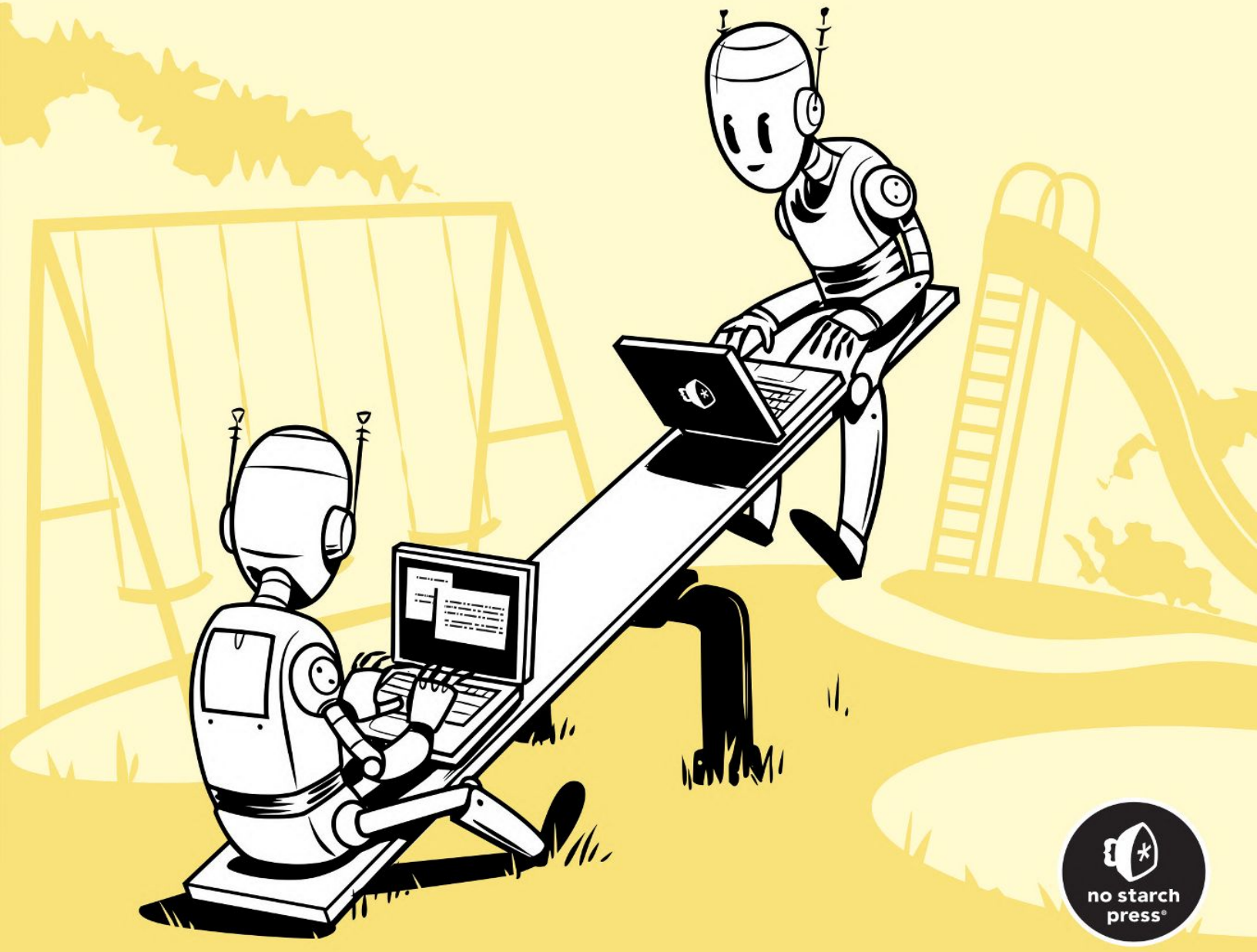
SECOND EDITION

PYTHON PLAYGROUND

GEEKY PROJECTS FOR THE CURIOUS PROGRAMMER

MAHESH VENKITACHALAM



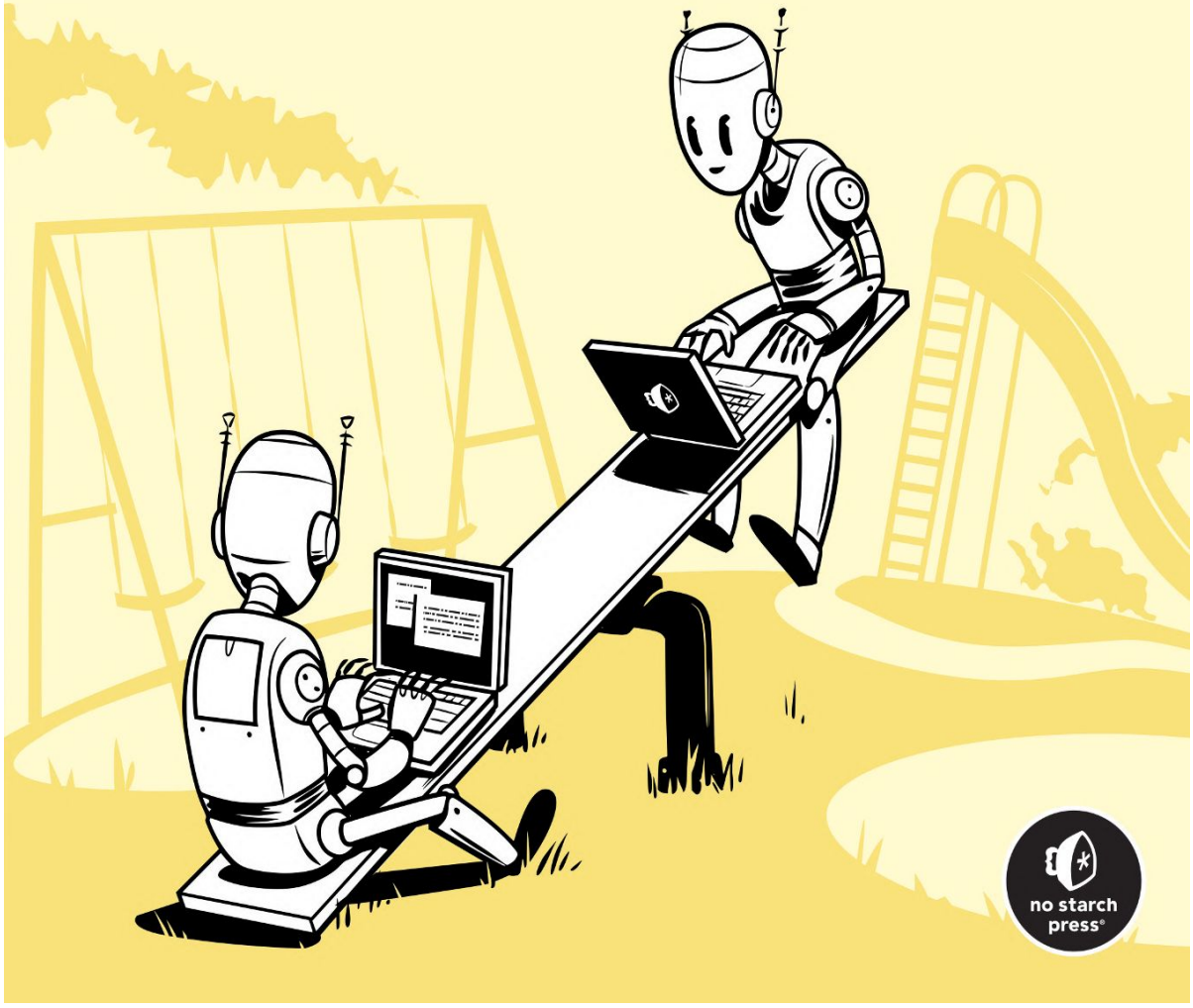
no starch press®

SECOND EDITION

PYTHON PLAYGROUND

GEEKY PROJECTS FOR THE CURIOUS PROGRAMMER

MAHESH VENKITACHALAM



no starch press®

PRAISE FOR THE FIRST EDITION

“If you want to become adept at doing clever things with Python, I doubt you’ll find a better group of projects or more useful help for understanding how the language works.”

—Network World

“This is a book that belongs in every Python programmer’s library.”

—*Full Circle Magazine*

“Packed with interesting projects.”

—iProgrammer

“*Python Playground* targets programmers who want to further improve their skills and knowledge of the language. The book does a good job of explaining all relevant details and makes sure that readers get a clear picture of what is going on.”

—InfoQ

“*Python Playground* has excellent projects for the science-minded programmer, the programming-minded science enthusiast, and everyone in between.”

—Al Sweigart, author of *Automate the Boring Stuff with Python*

“One of those rare programming books that is fun to read, and yet suited for the intermediate / advanced Python programmer. I reckon even the novices should take a stab . . . this book is enjoyable from start to finish.”

—Karthikeyan Chellappa, Goodreads reviewer

“Loaded with entertaining and interesting ideas . . . not a typical programming book at all. It’s for the hobbyist programmer looking for a bit of fun. Don’t pass it up!”

—F. Howard, Amazon reviewer

PYTHON PLAYGROUND

2nd Edition

Geeky Projects for the Curious Programmer

by Mahesh Venkitachalam



no starch
press®
San Francisco

PYTHON PLAYGROUND, 2ND EDITION. Copyright © 2024 by Mahesh Venkitachalam.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

First printing

27 26 25 24 23 1 2 3 4 5

ISBN-13: 978-1-7185-0304-5 (print)
ISBN-13: 978-1-7185-0305-2 (ebook)

Publisher: William Pollock
Managing Editor: Jill Franklin
Production Manager: Sabrina Plomitallo-González
Production Editor: Sydney Cromwell
Developmental Editor: Nathan Heidelberger
Cover Illustrator: Josh Ellingson
Interior Design: Octopod Studios
Technical Reviewers: Eric Mortenson and Xander Soldaat
Copyeditor: Kim Wimpsett
Proofreader: Audrey Doyle

Figure 12-4 has been adapted from “Timing diagram of I^2S” by wdwd and reproduced under CC BY 3.0, *https://creativecommons.org/licenses/by/3.0/*. Figure 12-6, courtesy of Raspberry Pi Ltd., has been reproduced under CC BY-ND 4.0, *https://creativecommons.org/licenses/by-nd/4.0/*.

For information on distribution, bulk sales, corporate sales, or translations, please contact No Starch Press® directly at info@nostarch.com or:

No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 1.415.863.9900
www.nostarch.com

The Library of Congress has catalogued the first edition as follows:

Venkitachalam, Mahesh.
  Python playground : geeky projects for the curious programmer / by Mahesh Venkitachalam.
  pages cm
  Includes index.
  ISBN 978-1-59327-604-1 -- ISBN 1-59327-604-4

1. Python (Computer program language) 2. Electronic apparatus and appliances--Automatic control. 3. Arduino (Programmable controller)--Programming. 4. Raspberry Pi (Computer)--Programming. I. Title.
QA76.73.P98.V46 2015
005.13'3--dc23
2014046103

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

For my parents,

A.V. Venkitachalam and N. Saraswathy,

for giving me the greatest gift of all—

an education.

&

For Hema

$H = M^2A$

About the Author

Mahesh Venkitachalam is a computer graphics and embedded systems consultant with over 20 years of experience. He is also the founder of Electronut Labs, a company known for developing innovative open source hardware. He maintains a blog on programming and electronics at *https://electronut.in*.

About the Technical Reviewers

Eric Mortenson has a BS and PhD in mathematics from the University of Wisconsin–Madison. He has held research and teaching positions at Pennsylvania State University, the University of Queensland, and the Max Planck Institute for Mathematics. He's currently an associate professor in mathematics and computer science at St. Petersburg State University.

Xander Soldaat is a former MINDSTORMS Community Partner for LEGO MINDSTORMS. He was an IT infrastructure architect and engineer for 18 years before becoming a full-time software developer. He has recently come full circle and gone back to his Linux roots as an OpenShift cloud success architect at Red Hat. In his spare time he likes to tinker with robots, 3D printing, and home-built retro-computers.

BRIEF CONTENTS

Acknowledgments
Introduction

Part I: Warming Up

Chapter 1: The Koch Snowflake
Draw a beautiful fractal pattern

Chapter 2: Spirographs
Animate spirograph-like curves

Part II: Simulating Life

Chapter 3: Conway’s Game of Life
Implement a cellular automaton

Chapter 4: Musical Overtones with Karplus-Strong
Synthesize realistic plucked string sounds

Chapter 5: Flocking Boids
Simulate the behavior of a flock of birds

Part III: Fun With Images

Chapter 6: ASCII Art
Convert images into text-based art

Chapter 7: Photomosaics

Reproduce an image from a grid of smaller images

Chapter 8: Autostereograms
Embed a 3D image in a 2D pattern

Part IV: Enter 3D

Chapter 9: Understanding OpenGL
Explore the basics of 3D graphics and shaders

Chapter 10: Conway's Game of Life on a Torus
Run a cellular automaton in 3D

Chapter 11: Volume Rendering
Visualize MRI and CT scan data

Part V: Hardware Hacking

Chapter 12: Karplus-Strong on a Raspberry Pi Pico
Build a playable electronic instrument

Chapter 13: Laser Audio Display with a Raspberry Pi
Design a laser light show that responds to music

Chapter 14: IoT Garden
Monitor the state of a garden over Bluetooth

Chapter 15: Audio ML on Pi
Train and deploy an embedded speech recognition system

Appendix A: Python Installation
Set up Anaconda Python with all the required modules
Appendix B: Raspberry Pi Setup

Index

CONTENTS IN DETAIL

Acknowledgments
Introduction
Who Is This Book For?
What's in This Book?
What's New to the Second Edition?
Why Python?
The Code in This Book

Part I: Warming Up

1 The Koch Snowflake

How It Works
- Using Recursion
- Computing the Snowflake
- Drawing with turtle Graphics

Requirements
The Code
- Calculating the Points
- Recursing
- Drawing a Flake
- Writing the main() Function

Running the Snowflake Code
Summary
Experiments!
The Complete Code

2 Spirographs

How It Works

Understanding Parametric Equations
Drawing Curves with turtle Graphics
Requirements
The Code
Drawing the Spiros
Coordinating the Animation
Saving the Curves
Parsing Command Line Arguments and Initialization
Running the Spirograph Animation
Summary
Experiments!
The Complete Code

Part II: Simulating Life

3 Conway's Game of Life
How It Works
Requirements
The Code
Representing the Grid
Setting the Initial Conditions
Enforcing the Boundary Conditions
Implementing the Rules
Sending Command Line Arguments to the Program
Initializing the Simulation
Running the Game of Life Simulation
Summary
Experiments!
The Complete Code

4 Musical Overtones with Karplus-Strong
How It Works
The Simulation
The WAV File Format

The Minor Pentatonic Scale
Requirements
The Code
Implementing the Ring Buffer with deque
Implementing the Karplus-Strong Algorithm
Writing a WAV File
Playing WAV Files with pyaudio
Creating Notes and Parsing Arguments
Running the Plucked String Simulation
Summary
Experiments!
The Complete Code

5 Flocking Boids
How It Works
Requirements
The Code
Initializing the Simulation
Setting Boundary Conditions
Drawing a Boid
Applying the Rules of the Boids
Influencing the Simulation
Incrementing the Simulation
Parsing Arguments and Instantiating the Boids
Running the Boids Simulation
Summary
Experiments!
The Complete Code

Part III: Fun with Images

6 ASCII Art
How It Works
Requirements

The Code
Defining the Grayscale Levels and Grid
Computing the Average Brightness
Generating the ASCII Content from the Image
Creating Command Line Options
Writing the ASCII Art Strings to a Text File
Running the ASCII Art Generator
Summary
Experiments!
The Complete Code

7 Photomosaics
How It Works
Splitting the Target Image
Averaging Color Values
Matching Images
Requirements
The Code
Reading In the Input Images
Calculating the Average Color Value of an Image
Splitting the Target Image into a Grid
Finding the Best Match for a Tile
Creating an Image Grid
Creating the Photomosaic
Writing the main() Function
Running the Photomosaic Generator
Summary
Experiments!
The Complete Code

8 Autostereograms
How It Works
Perceiving Depth in an Autostereogram
Working with Depth Maps
Shifting Pixels

Requirements
The Code
Creating a Tile from Random Circles
Repeating a Given Tile
Creating Autostereograms
Providing Command Line Options
Running the Autostereogram Generator
Summary
Experiments!
The Complete Code

Part IV: Enter 3D

9 Understanding OpenGL
How OpenGL Works
Geometric Primitives
3D Transformations
Shaders
Vertex Buffers
Texture Mapping
The OpenGL Context
Requirements
The Code
The RenderWindow Class
The Scene Class
Utility Functions
Running the OpenGL Application
Summary
Experiments!
The Complete Code

10 Conway's Game of Life on a Torus
How It Works
Computing Vertices

Computing Normals for Lighting
Rendering
Coloring the Triangle Strips
Controlling the Camera
Mapping the Grid to the Torus
Requirements
The Code
Rendering the Torus
Implementing the Game of Life Simulation
Creating the Camera
Putting Everything Together
Running the 3D Game of Life Simulation
Summary
Experiments!
The Complete Torus Rendering Code
The Complete Game of Life Simulation Code
The Complete Camera Code
The Complete RenderWindow Code

11 Volume Rendering
How It Works
The Data Format
Ray Generation
The OpenGL Window
Requirements
The Code
Generating a 3D Texture
Generating Rays
Implementing the Ray Casting Algorithm
Showing 2D Slices
Putting the Code Together
Running the Program
Summary
Experiments!

The Complete 3D Texture Code
The Complete Ray Generation Code
The Complete Volume Ray Casting Code
The Complete 2D Slicing Code
The Complete Main File Code

Part V: Hardware Hacking

12 Karplus-Strong on a Raspberry Pi Pico

How It Works
- Input and Output
- The I2S Protocol

Requirements
- Hardware Setup
- MicroPython Setup

The Code
- Setting Up
- Generating the Notes
- Playing a Note
- Writing the main() Function

Running the Pico Code
Summary
Experiments!
The Complete Code

13 Laser Audio Display with a Raspberry Pi

How It Works
- Generating Patterns with a Laser
- Analyzing Audio with the Fast Fourier Transform

Requirements
- Setting Up the Raspberry Pi
- Constructing the Laser Display
- Hooking Up the Hardware

The Code

Setting Up
Controlling the Hardware
Processing the Audio
Testing the Motors
Putting It All Together
Running the Laser Display
Summary
Experiments!
The Complete Code

14 IoT Garden
How It Works
Bluetooth Low Energy
The Bottle Web Framework
The SQLite Database
Requirements
Raspberry Pi Setup
CircuitPython Setup
If This Then That Setup
The Code
The CircuitPython Code
The BLE Scanner Code
The Web Server Code
The Main Program File
Running the IoT Garden
Summary
Experiments!
The Complete CircuitPython Code
The Complete BLE Scanner Code
The Complete Python Web Server Code
The Complete Main Program Code

15 Audio ML on Pi
A Machine Learning Overview

How It Works
Spectrograms
Inference on the Raspberry Pi
Requirements
The Code
Training the Model in Google Colab
Using the Model on the Raspberry Pi
Running the Speech Recognition System
Summary
Experiments!
The Complete Code

A Python Installation
Installing Source Code for the Book's Projects
Installing Python and Python Modules
Windows
macOS
Linux

B Raspberry Pi Setup
Setting Up the Software
Testing Your Connection
Logging in to Your Pi with SSH
Installing Python Modules
Working Remotely with Visual Studio Code
Index

ACKNOWLEDGMENTS

Writing a book is like running a marathon, or so I've been told. What I do know is that writing this book tested the limits of my endurance, and I couldn't have done it without my personal cheerleading squad of close friends and family.

First, I'd like to thank my wife, Hema, for her constant love, encouragement, and patience throughout the two years it took to complete this work. I thank my friend Raviprakash Jayaraman for being a co-conspirator in all my dubious projects, for being a technical reviewer for this book, and for interesting lunches, movies, and trips to the S.P. Road Zoo. I thank my friend Seby Kallarakkal for pushing me to pursue this book and for all the interesting discussions we had together. I am grateful to my pal Dr. Santosh Hemachandra for helpful discussions on fast Fourier transforms. I'd like to thank Karthikeyan Chellappa for helping me test the installation of Python modules and for our running sessions around Kaikondrahalli Lake. I would also like to thank Matthew Denham (with whom I conversed on Reddit) for his help on the mathematics of spirographs.

I'd like to thank Tyler Ortman and Bill Pollock of No Starch Press for accepting my book proposal, and Serena Yang for her professional work on the first edition of the book. I'd like to thank Nicholas Kramer for his technical review of the first edition.

For the second edition, I would like to thank Nathan Heidelberger for his meticulous editing. I was able to improve many sections from the first edition thanks to his thoroughness. I would also like to thank Eric Mortenson for his technical review of the software section of the book, and

Xander Soldaat for his technical review of the hardware section. Thanks a second time to my friend Raviprakash Jayaraman for further help testing the code across multiple operating systems. I’m also grateful to my son and officemate, Aryan Mahesh, for his goofy jokes, music recommendations, and in-depth sci-fi book and movie debates throughout the writing of this book.

I thank my parents, A.V. Venkitachalam and N. Saraswathy, for providing me with an education far beyond their financial means. Finally, I thank all my teachers who inspired me. I hope to remain a student all my life.

INTRODUCTION



Welcome to the second edition of *Python Playground*! Within these pages, you'll find 15 exciting projects designed to encourage you to explore the world of programming with Python. The projects cover a wide range of topics, such as drawing Spirograph-like patterns, 3D rendering, projecting laser patterns in sync with music, and speech recognition using machine learning. In addition to being fun in and of themselves, these projects offer a lot of room for expansion and are designed to be jumping-off points for you to explore your own ideas.

Who Is This Book For?

Python Playground was written for anyone curious about how to use programming to understand and explore ideas. The projects in this book assume that you know basic Python syntax and basic programming concepts and that you're familiar with high school mathematics. I've done my best to explain in detail the math you need for all projects.

This book isn't intended to be your first book on Python. I won't walk you through the basics. I will, however, show you how to use Python to solve a variety of real-world problems in a series of nontrivial projects. As you work through the projects, you'll explore the nuances of the Python programming language and learn how to work with some popular Python libraries. But perhaps even more important, you'll learn how to break down a problem into parts, develop an algorithm to solve that problem, and then implement a solution from the ground up using Python.

It can be difficult to solve real-world problems because they are often open-ended and require expertise in various domains. But Python offers the tools to facilitate problem-solving. Overcoming difficulties and finding solutions to real problems are the most important parts of your journey on the way to becoming an expert programmer.

What's in This Book?

Let's take a quick tour through the chapters in this book. Part I features a few projects to get you started.

Chapter 1: The Koch Snowflake Uses recursive functions and `turtle` graphics to draw an interesting fractal shape

Chapter 2: Spirographs Harnesses parametric equations and `turtle` graphics to draw curves like the ones generated with the toy Spirograph

Part II features projects that use mathematical models to create computer simulations of real-world phenomena.

Chapter 3: Conway's Game of Life Implements a famous cellular automaton using `numpy` and `matploblib`, producing an evolving, lifelike system based on a few simple rules

Chapter 4: Musical Overtones with Karplus-Strong Realistically simulates the sound of a plucked string instrument and plays back the sounds with `pyaudio`

Chapter 5: Flocking Boids Uses `numpy` and `matplotlib` to implement the Boids algorithm and simulate the flocking behavior of birds

The projects in Part III will introduce you to reading and manipulating 2D images with Python.

Chapter 6: ASCII Art Introduces `Pillow`, a fork of the Python Imaging Library (PIL), by converting images into text-based ASCII art

Chapter 7: Photomosaics Stitches together a grid of smaller images to create a recognizable larger image

Chapter 8: Autostereograms Uses depth maps and pixel manipulation to embed the illusion of a 3D image in a 2D image

In Part IV, you'll learn to use shaders and the OpenGL library to quickly and efficiently render 3D graphics by working directly with your graphics processing unit (GPU).

Chapter 9: Understanding OpenGL Introduces the basics of using OpenGL to create simple 3D graphics

Chapter 10: Conway's Game of Life on a Torus Revisits the simulation from Chapter 3 and brings it to life on the surface of a 3D torus

Chapter 11: Volume Rendering Implements a volume ray casting algorithm to render volumetric data, a technique commonly used for medical imaging such as MRI and CT scans

Finally, in Part V, you'll work with a Raspberry Pi and other electronic components to deploy Python on embedded systems.

Chapter 12: Karplus-Strong on a Raspberry Pi Pico Creates a playable electronic instrument by implementing the Karplus-Strong algorithm from Chapter 4 on a tiny Raspberry Pi Pico microcontroller using MicroPython

Chapter 13: Laser Audio Display with a Raspberry Pi Harnesses Python on a Raspberry Pi to control two rotating mirrors and a laser, producing a laser light show that responds to sound

Chapter 14: IoT Garden Uses Bluetooth Low Energy to connect a Raspberry Pi with Adafruit hardware running CircuitPython, creating an IoT system to monitor temperature and humidity levels in your garden

Chapter 15: Audio ML on Pi Introduces the exciting world of machine learning with TensorFlow by implementing a speech recognition system on your Raspberry Pi

Each chapter ends with an "Experiments!" section featuring suggestions for how you can expand the project or further explore the topic at hand.

What's New to the Second Edition?

This second edition features five new projects, including the Koch snowflake project in Chapter 1 and the 3D version of Conway's Game of Life in Chapter 10. The most significant updates are in the hardware section, which has been

streamlined to focus exclusively on Raspberry Pi–based systems, rather than a mix of Raspberry Pi and Arduino. As such, every project in Part V either is new (Chapter 12, Chapter 14, Chapter 15) or has been thoroughly redesigned (Chapter 13).

Relying just on Raspberry Pi simplifies the setup process for the hardware projects, and it keeps the book's focus on Python programming. There's no longer a need to switch between Python and code written in the Arduino programming language (a version of C++). With the updated Part V, you'll also get a taste for programming with MicroPython and CircuitPython, two flavors of Python optimized for running on resource-constrained embedded systems.

Other significant updates to the second edition include:

- Using `pyaudio` rather than `pygame` to play WAV files in Chapter 4
- Comparing the performance of a linear search algorithm and a k-d tree data structure when finding the best image matches for the photomosaic in Chapter 7
- Guidance on how to create your own depth maps for generating autostereograms in Chapter 8
- Streamlined installation instructions in Appendix A using the standard Anaconda Distribution of Python

Beyond these specific updates, the entire text has been reviewed, corrected, and clarified, and the code has been updated as needed to reflect changes to Python since the first edition was released.

Why Python?

Python is an ideal language for exploring programming. As a multiparadigm language, it provides you with a lot of flexibility in how you structure your programs. You can use Python as a *scripting* language to simply execute code, as a *procedural* language to organize your program into a collection of functions that call one another, or as an *object-oriented* language that uses classes, inheritance, and modules to build up a hierarchy. This flexibility allows you to choose the programming style most suited to a particular project.

When you develop using a more traditional language like C or C++, you have to compile and link your code before you can run it. With Python, you can run it directly after editing. (Under the hood, Python compiles your code into an intermediate bytecode that is then run by the Python interpreter, but these processes are transparent to you, the user.) In practice, the process of modifying and running your code over and over is much less cumbersome with Python.

Python has a very small set of simple and powerful data structures. If you already understand strings, lists, tuples, dictionaries, list comprehensions, and basic control structures such as `for` and `while` loops, you're off to a great start. Python's succinct and expressive syntax makes it easy to do complex operations with just a few lines of code. And once you're familiar with Python's built-in and third-party modules, you'll have an arsenal of tools to tackle real problems like the ones covered in this book. There are standard ways to call C/C++ code from Python and vice versa, and because you can find libraries to do almost anything in Python, it's easy to combine Python with other language modules in bigger projects. This is why Python is considered a great *glue* language—it makes it easy to combine diverse software components. The 3D graphics

projects in Part IV demonstrate how Python can work side by side with the C-like OpenGL Shading Language, and in Chapter 14 you'll even mix in a little bit of HTML, CSS, and JavaScript to create a web interface for your IoT garden monitor. Real software projects often use a mix of software technologies, and Python fits very well into such layered architectures.

Python also provides a handy tool in the form of the Python interpreter. It lets you easily check code syntax, do quick computations, and even test code under development. When I write Python code, I keep three windows open: a text editor, a shell, and a Python interpreter. As I develop code in the editor, I import my functions or classes into the interpreter and test them as I go.

I also use the interpreter to get a feel for new modules before I incorporate them into my code. For example, while developing code for the IoT garden project in Chapter 14, I wanted to test the `sqlite3` database module. I brought up the Python interpreter and tried the following to make sure I understood how to create and add entries to a database:

```
>>> import sqlite3
>>> con = sqlite3.connect('test.db')
>>> cur = con.cursor()
>>> cur.execute("CREATE TABLE sensor_data (TS datetime, ID
    text, VAL numeric)")
>>> for i in range(10):
...   cur.execute("INSERT into sensor_data VALUES
    (datetime('now'),'ABC', ?)", (i, ))
>>> con.commit()
>>> con.close()
>>> exit()
```

Then, to make sure it worked, I did the following to retrieve some of the data I added:

```
>>> con = sqlite3.connect('test.db')
>>> cur = con.cursor()
>>> cur.execute("SELECT * FROM sensor_data WHERE VAL > 5")
>>> print(cur.fetchall())
[('2021-10-16 13:01:22', 'ABC', 6), ('2021-10-16 13:01:22',
  'ABC', 7),
 ('2021-10-16 13:01:22', 'ABC', 8), ('2021-10-16 13:01:22',
  'ABC', 9)]
```

This example demonstrates the practical use of the Python interpreter as a powerful tool for development. You don't need to write a complete program to do quick experiments; just open up the interpreter and get going. This is just one of the many reasons why I love Python and why I think you will too.

The Code in This Book

I've done my best throughout this book to walk you through the code for each project in detail, piece by piece. You can either enter the code yourself or download the complete source code (using the Download Zip option) for all programs in the book at *https://github.com/mkvenkit/pp2e*.

You'll find several exciting projects in the following pages. I hope you have as much fun playing with them as I had creating them. Don't forget to explore further with the experiments presented at the end of each project. I wish you many happy hours of programming with *Python Playground*!

PART I

WARMING UP

In the beginner’s mind there are many possibilities; in the expert’s mind there are few.

—Shunryu Suzuki

1
THE KOCH SNOWFLAKE



We'll start our Python adventures by figuring out how to draw an interesting shape called the *Koch snowflake*, invented by Swedish mathematician Helge von Koch in 1904. The Koch snowflake is a *fractal*—a type of figure that repeats itself as you zoom in to it.

Fractals derive their repeating nature from *recursion*, a technique where something is defined in terms of itself. In particular, you draw a fractal using a *recursive algorithm*, a repeating process where one repetition's output becomes the input of the next repetition.

As you work through this chapter, you'll learn:

- The basics of recursive algorithms and functions
- How to create graphics using the `turtle` module
- A recursive algorithm to draw the Koch snowflake
- Some linear algebra

How It Works

Figure 1-1 shows what the Koch snowflake looks like. Notice how the large branch in the middle is repeated on a smaller scale by branches on the left and right. Similarly, the large branch in the middle is itself made up of smaller branches that echo the larger shape. This is the repeating, self-similar nature of a fractal.

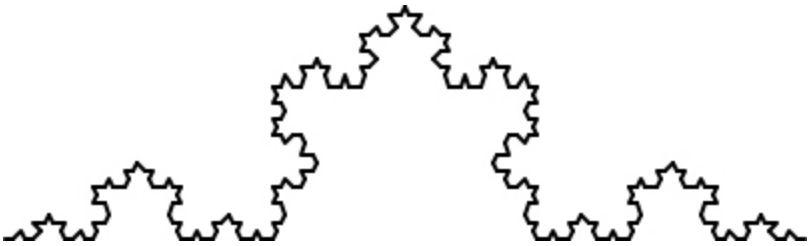


Figure 1-1: *The Koch snowflake*

If you know how to calculate the points that form the basic shape making up the snowflake, you can develop an algorithm to perform the same calculations recursively. This way, you'll draw smaller and smaller versions of that shape, building up the fractal. In this section, we'll look generally at how recursion works. Then we'll consider how to apply recursion, along with some linear algebra and Python's `turtle` module, to draw the Koch snowflake.

Using Recursion

To get a feel for how recursion works, let's take a look at a simple recursive algorithm: computing the factorial of a number. The factorial of a number can be defined by a function, as shown here:

$$f(N) = 1 \times 2 \times 3 \times \ldots \times (N - 1) \times N$$

In other words, the factorial of N is just the product of the numbers 1 through N. You can rewrite this as:

$$f(N) = N \times (N - 1) \times \ldots \times 3 \times 2 \times 1$$

which can again be rewritten as:

$$f(N) = N \times f(N - 1)$$

Wait, what did you just do? You defined *f* in terms of itself! That's recursion. Calling $f(N)$ will end up calling $f(N - 1)$, which will end up calling $f(N - 2)$, and so on. But how do you know when to stop? Well, you have to define $f(1)$ as 1, and that will be the deepest step of the recursion.

Here's how to implement the recursive factorial function in Python:

```
def factorial(N):
  ❶ if N == 1:
        return 1
    else:
      ❷ return N * factorial(N-1)
```

You handle the case where *N* is equal to 1 by simply returning `1` ❶, and you implement the recursive call by calling `factorial()` again ❷, this time passing in `N-1`. The function will keep calling itself until *N* equals 1. The net effect is that when the function returns, it will have computed the product of all numbers 1 through *N*.

In general, when you're trying to implement an algorithm using recursion, follow these steps:

1. Define a base case where the recursion ends. In our factorial example, you did this by defining $f(1)$ as 1.
2. Define the recursive step. For this you need to think about how to express the algorithm as a recursive

process. In some algorithms, there can be multiple recursive calls from a function—as you'll see soon.

Recursion is a helpful tool for problems that can be naturally partitioned into smaller versions of themselves. The factorial algorithm is a perfect example of this partitioning, and as you'll soon see, so is drawing the Koch snowflake. That said, recursion isn't always the most efficient way to solve a problem. In some cases, it would make sense to re-implement the recursive algorithm in terms of loops. But the fact remains that recursive algorithms are often more compact and elegant compared to their loopy counterparts.

Computing the Snowflake

Now let's look at how to construct the Koch snowflake. Figure 1-2 shows the basic pattern for drawing the snowflake. I'll call this pattern a *flake*. The basis of the figure is the line segment $\overline{AB}$ of length d. The segment is split into three equal parts, $\overline{AP_1}$, $\overline{P_1P_3}$, and $\overline{P_3B}$, each of which has a length r. Instead of directly connecting points P_1 and P_3, these points are connected through P_2, which is chosen such that P_1, P_2, and P_3 form an equilateral triangle of side length r and height h. Point C, the midpoint of P_1 and P_3 (and by extension of A and B), falls directly beneath P_2, such that $\overline{AB}$ and $\overline{CP_2}$ are perpendicular.

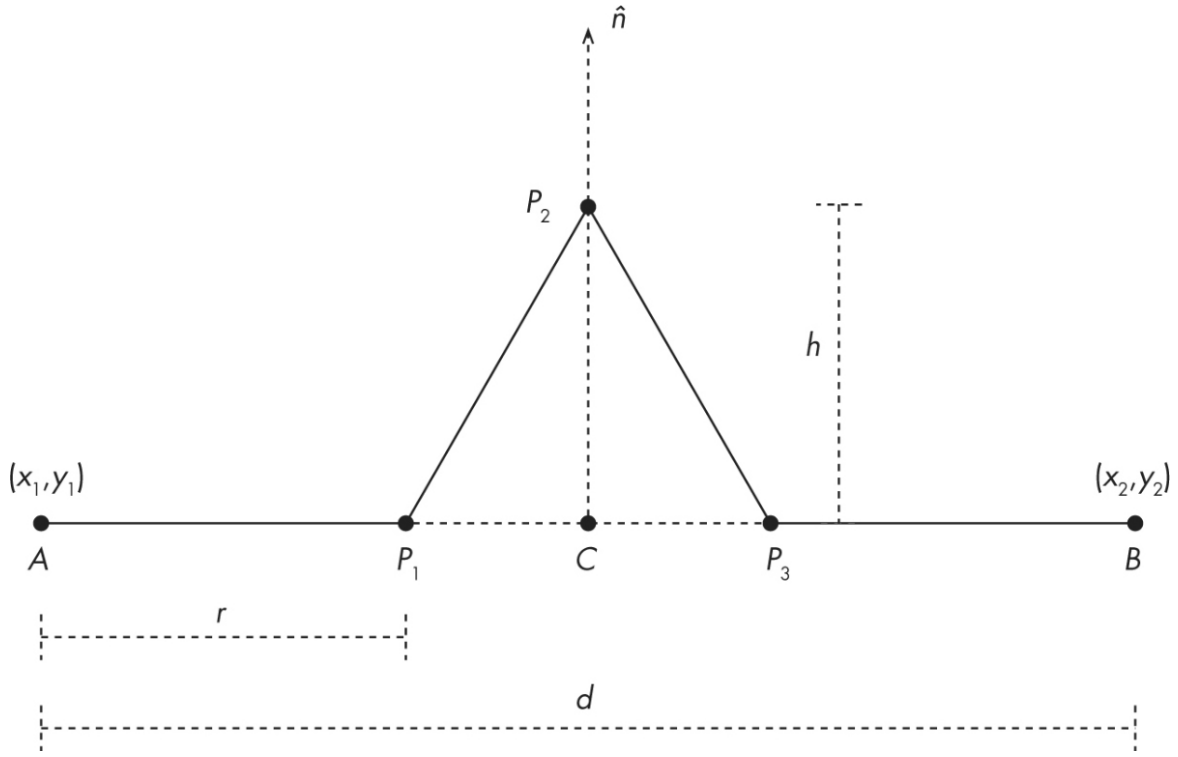


Figure 1-2: *The basic pattern for drawing a Koch snowflake*

Once you understand how to calculate the points shown in Figure 1-2, you'll be able to recursively draw smaller and smaller flakes to reproduce the Koch snowflake. Essentially, your goal is this: given points *A* and *B*, you want to compute the points P_1, P_2, and P_3 and join them up as shown in the figure. To calculate those points, you'll need to use some linear algebra, a mathematical discipline that lets you compute distances and figure out coordinates of points based on *vectors*, quantities that have both magnitude and direction.

Here's a simple formula from linear algebra that you'll be using. Say you have a point *A* in 3D space and a unit vector $\hat{n}$ (a *unit vector* is a vector with a length of 1 unit). Point *B* at a distance *d* along this unit vector is given by:

$$B = A + d \times \hat{n}$$

You can easily verify this with an example. Take the case where A = (5, 0, 0) and $\hat{n}$ = (0, 1, 0). What are the coordinates for a point B that's 10 units away from A along $\hat{n}$? Using the previous formula, you get:

$$B = (5, 0, 0) + 10 \times (0, 1, 0) = (5, 10, 0)$$

In other words, to get from A to B, you move 10 units along the positive y-axis.

Here's another result you'll use—let's call it the *perpendicular vector trick*. Say you have a vector $\vec{A}$ = (a, b). If you have another vector $\vec{B}$ that's perpendicular to $\vec{A}$, it can be expressed as $\vec{B}$ = ($-b$, a). You can verify that this trick works by taking the dot product of $\vec{A}$ and $\vec{B}$. To take the dot product of a pair of two-dimensional vectors, multiply the first components from each vector, then multiply the second components from each vector, and finally add the results together. In this case, the dot product of $\vec{A}$ and $\vec{B}$ is:

$$\vec{A} \cdot \vec{B} = (a \times -b) + (b \times a) = -ab + ab = 0$$

The dot product of two perpendicular vectors will always be zero, so $\vec{B}$ is indeed perpendicular to $\vec{A}$.

With this in mind, let's return to the flake in Figure 1-2. How can you calculate the position of P_2, given the coordinates for points A and B? You know that P_2 falls h distance away from point C along unit vector $\hat{n}$. Your first linear algebra formula tells you:

$$P_2 = C + h \times \hat{n}$$

Now let's put those variables in terms that you know. First, C is the midpoint of line $\overline{AB}$, so $C = (A + B) / 2$. Next, h is the height of an equilateral triangle with side length r. The Pythagorean theorem tells you:

$$h = \frac{\sqrt{3}}{2} r$$

In this case, r is simply a third of the distance from A to B. If A has coordinates (x_1, y_1) and B has coordinates (x_2, y_2), you can calculate the distance between them as:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Then simply divide d by 3 to get r.

Finally, you need a way to express $\hat{n}$. You know that $\hat{n}$ is perpendicular to vector $\vec{AB}$, and you can express $\vec{AB}$ by subtracting point A's coordinates from point B's:

$$\vec{AB} = (x_2 - x_1, y_2 - y_1)$$

The magnitude of $\vec{AB}$ is given by $d = |\vec{AB}|$. You can now use the perpendicular vector trick to express $\hat{n}$ in terms of A and B:

$$\hat{n} = \frac{(-(y_2 - y_1), x_2 - x_1)}{|\vec{AB}|} = \frac{(y_1 - y_2)}{d}, \frac{(x_2 - x_1)}{d}$$

Next you need to compute P_1 and P_3. For this you're going to use another result from linear algebra. Let's say you have a line $\overline{AB}$ and a point C on the line. Let a be the distance of C from A and b be the distance of C from B. The point C is given by:

$$C = \frac{(b \times A) + (a \times B)}{a + b}$$

To understand this formula, think about what happens if C is the midpoint of A and B, meaning a and b would be the same. In this case, you can intuit that C ought to equal (A + B) / 2. Substitute all the bs for as in the previous equation. You'll get:

$$C = \frac{(a \times A) + (a \times B)}{a + a} = \frac{A + B}{2}$$

With this new formula in mind, you can now compute P_1 and P_3. These points divide line $\overline{AB}$ into thirds, meaning the distance from P_1 to B is twice the distance from P_1 to A (b = $2a$), and the distance from P_3 to A is twice the distance from P_3 to B (a = $2b$). Feeding this into the formula, you can therefore calculate the points as:

$$P_1 = \frac{2 \times A + B}{2} \quad \text{and} \quad P_3 = \frac{A + 2 \times B}{3}$$

Now you have everything you need to draw the first level of the snowflake fractal. Once you decide on *A* and *B*, you know how to compute the points P_1, P_2, and P_3. But what happens at the second level of the fractal? You take each individual line segment from the flake at the first level (Figure 1-2) and replace it with a smaller flake. The result is shown in Figure 1-3.

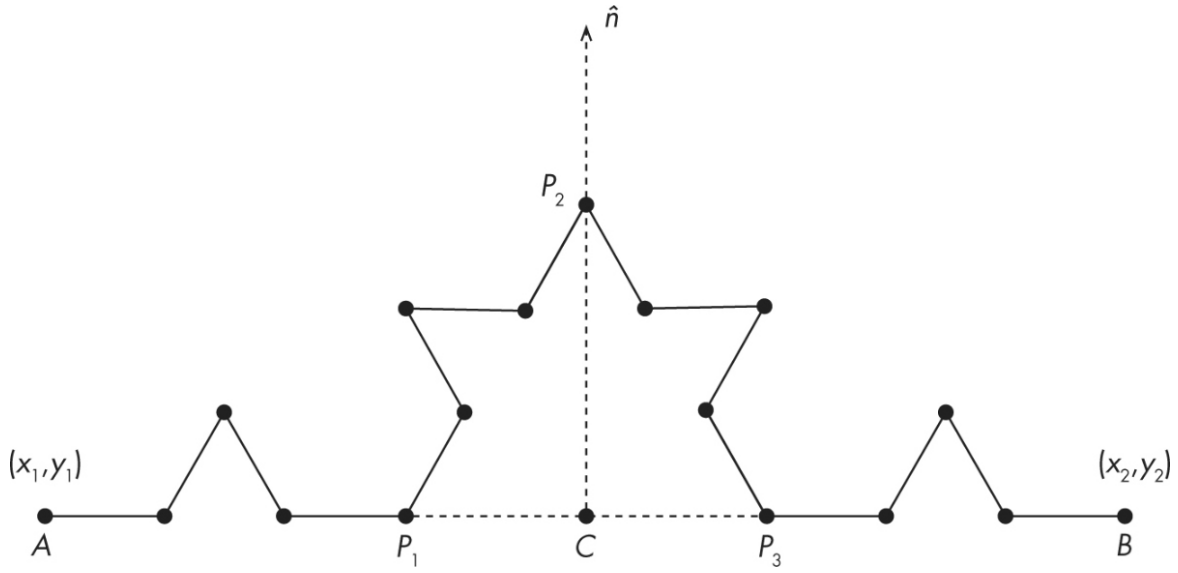


Figure 1-3: *The second step of Koch snowflake construction*

Notice how each of the four line segments from Figure 1-2, $\overline{AB}_1$, $\overline{P_1P_2}$, $\overline{P_2P_3}$, and $\overline{P_3B}$, has become the basis for a new flake. In the Koch snowflake program, you'll be able to use the endpoints of each line segment, for example, *A* and P_1, as new values for *A* and *B* and recursively perform the same calculations used to arrive at the points in Figure 1-2.

At each level of the fractal, you'll subdivide the snowflake again, drawing smaller and smaller self-similar figures. This

is the *recursive step* of the algorithm, which you'll repeat until you reach a *base case*. This should happen when $\overline{AB}$ is smaller than a certain threshold—say, 10 pixels. When you hit that threshold, just draw the line segments and stop recursing.

To make the final output a bit fancy, you can draw three linked flakes as the first level of the fractal. This will give you the hexagonal symmetry of an actual snowflake. Figure 1-4 shows what the starting drawing will look like.

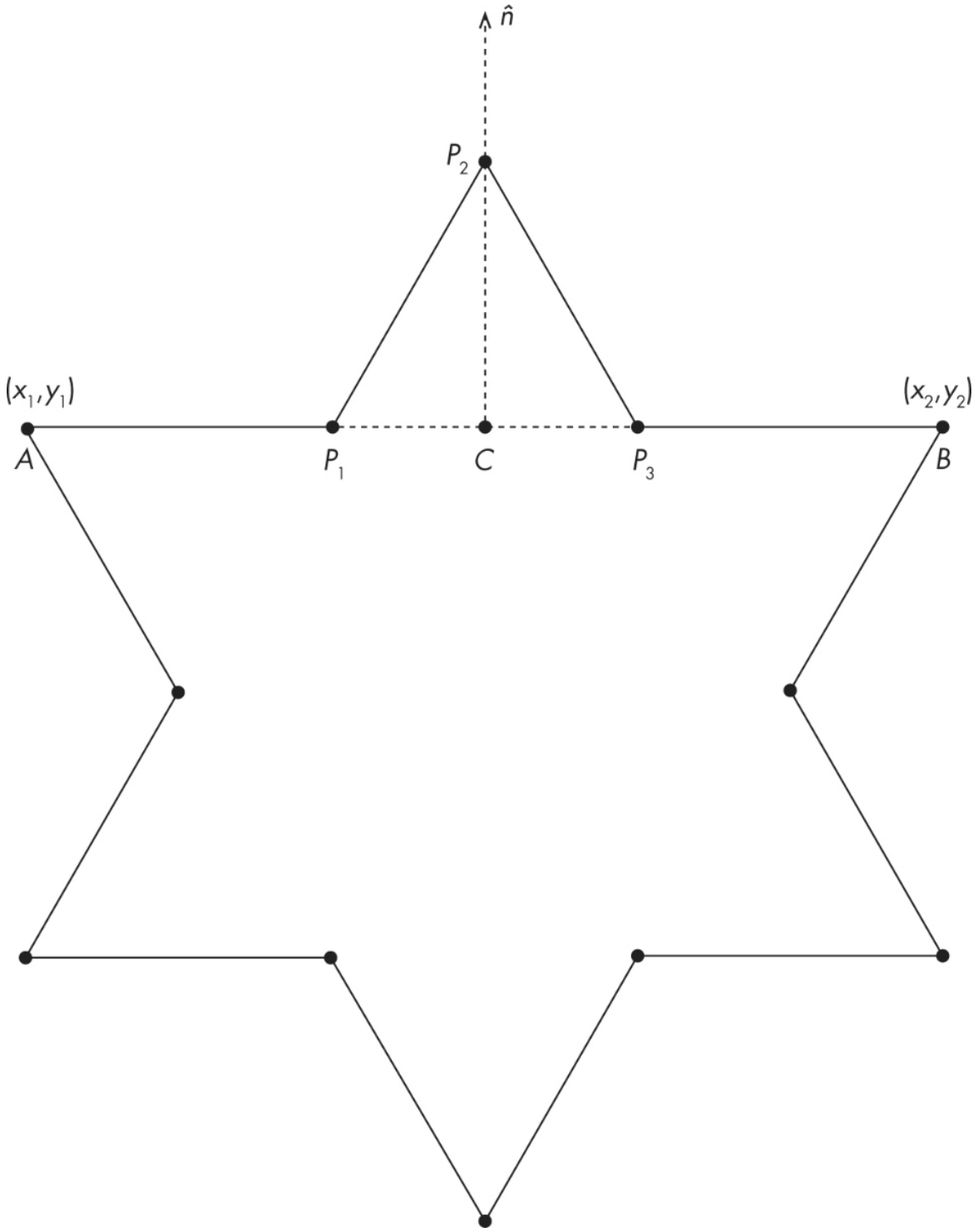


Figure 1-4: *Combining three snowflakes*

Now that you know how to calculate the coordinates for making the snowflake, let’s see how to use those coordinates in Python to actually draw an image.

Drawing with turtle Graphics

In this chapter, you'll use Python's `turtle` module to draw the snowflake; it's a simple drawing program modeled after the idea of a turtle dragging its tail through the sand, creating patterns. The `turtle` module includes methods you can use to set the position and color of the pen (the turtle's tail) and many other useful functions for drawing.

As you'll see, all you need is a handful of graphics functions to draw the Koch snowflake. In fact, from the standpoint of `turtle`, drawing the snowflake is almost as easy as drawing a triangle. To prove it, and to give you a feel for how `turtle` works, the following program uses `turtle` to draw said triangle. Enter the code, save it as *test_turtle.py*, and run it in Python:

```
import turtle

def draw_triangle(x1, y1, x2, y2, x3, y3, t):
    # go to start of triangle
  ❷ t.up()
  ❸ t.setpos(x1, y1)
  ❹ t.down()
    t.setpos(x2, y2)
    t.setpos(x3, y3)
    t.setpos(x1, y1)
    t.up()

def main():
    print('testing turtle graphics...')

  ❺ t = turtle.Turtle()
  ❻ t.hideturtle()

  ❼ draw_triangle(-100, 0, 0, -173.2, 100, 0, t)

  ❽ turtle.mainloop()

# call main
if __name__ == '__main__':
    main()
```

You start by importing the `turtle` module ❶. Next, you define the `draw_triangle()` method, whose parameters are three pairs of x-coordinates and y-coordinates (the three corners of a triangle), as well as `t`, a `turtle` object. The method starts by calling `up()` ❷. This tells Python to move the pen up; in other words, take the pen off the virtual paper so that it won't draw as you move the turtle. You want to position the turtle before you start drawing. The `setpos()` call ❸ sets the position of the turtle to the first pair of x- and y-coordinates. Calling `down()` ❹ sets the pen down, and for each of the subsequent `setpos()` calls, a line is drawn as the turtle moves to the next set of coordinates. The net result is a triangle drawing.

Next you declare a `main()` function to actually do the drawing. In it, you create the `turtle` object for drawing ❺ and hide the turtle ❻. Without this command, you'd see a small shape representing the turtle at the front of the line being drawn. You then call `draw_triangle()` to draw the triangle ❼, passing in the desired coordinates as arguments. The call to `mainloop()` ❽ keeps the `tkinter` window open after the triangle has been drawn. (`tkinter` is Python's default GUI library.)

Figure 1-5 shows the output of this simple program.

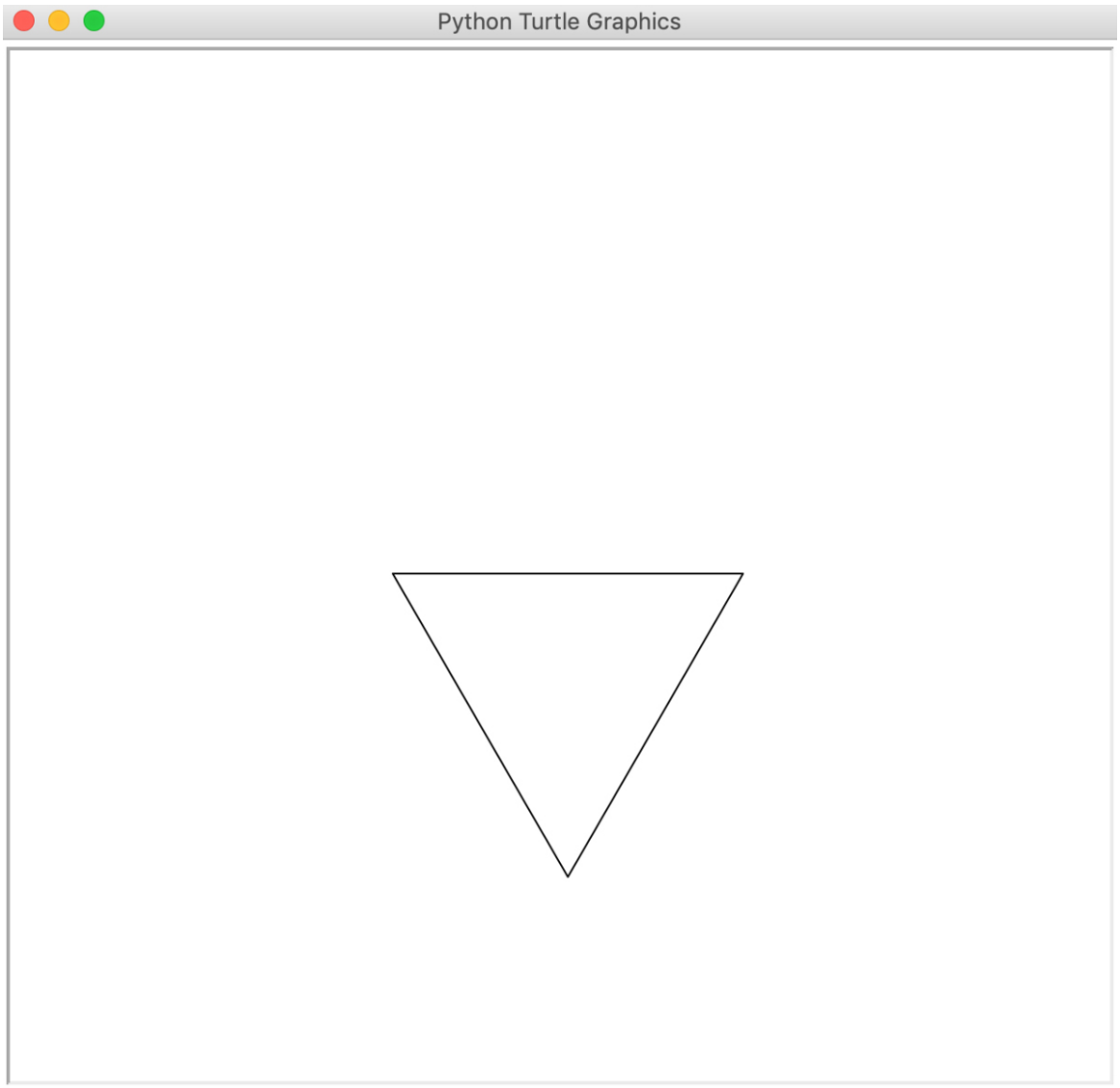


Figure 1-5: *The output of a simple* `turtle` *program*

You now have everything you need for the project. Let's draw some flakes!

Requirements

In this project, you'll use the Python `turtle` module to draw the snowflake.

The Code

To draw the Koch snowflake, define a recursive function, `drawKochSF()`. This function computes P_1, P_2, and P_3 in terms of *A* and *B* from Figure 1-2 and then recursively calls itself to perform the same calculation for smaller and smaller line segments until it reaches the smallest base case. Then it draws the flakes using `turtle`. For the full project code, skip ahead to "The Complete Code" on page 16. The code is also available in the book's GitHub repository at *https://github.com/mkvenkit/pp2e/blob/main/koch/koch.py*.

Calculating the Points

Begin the `drawKochSF()` function by calculating the coordinates for all the points needed to draw the basic flake pattern shown in Figure 1-2.

```
def drawKochSF(x1, y1, x2, y2, t):
    d = math.sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2))
    r = d/3.0
    h = r*math.sqrt(3)/2.0
    p3 = ((x1 + 2*x2)/3.0, (y1 + 2*y2)/3.0)
    p1 = ((2*x1 + x2)/3.0, (2*y1 + y2)/3.0)
    c = (0.5*(x1+x2), 0.5*(y1+y2))
    n = ((y1-y2)/d, (x2-x1)/d)
    p2 = (c[0]+h*n[0], c[1]+h*n[1])
```

You define `drawKochSF()`, passing in the x- and y-coordinates for the endpoints of a line segment $\overline{AB}$, which forms the basis for one of the sides of the snowflake, as shown in Figure 1-4. You also pass in the `turtle` object `t`, which you use for the actual drawing. Then you compute all the parameters shown in Figure 1-2, as discussed in the "Computing the Snowflake" section, starting with `d`, the

distance from *A* to *B*. Dividing `d` by 3 gives you `r`, the length of each of the four line segments that makes up a flake. You use `r` to find `h`, the height of the "cone" at the heart of the flake.

You calculate the rest of the parameters as tuples containing an x- and a y-coordinate. The `p3` and `p1` tuples describe the two points at the base of the cone portion of the flake. Point `c` is the midpoint of `p1` and `p3`, and `n` is the unit vector perpendicular to line $\overline{AB}$. Along with `h`, they help you calculate `p2`, the apex of the flake's cone.

Recursing

The next part of the `drawKochSF()` function uses recursion to break down the first-level flake into smaller and smaller versions of itself.

```
❶ if d > 10:
      # flake #1
    ❷ drawKochSF(x1, y1, p1[0], p1[1], t)
      # flake #2
      drawKochSF(p1[0], p1[1], p2[0], p2[1], t)
      # flake #3
      drawKochSF(p2[0], p2[1], p3[0], p3[1], t)
      # flake #4
      drawKochSF(p3[0], p3[1], x2, y2, t)
```

First you check for the recursion-stopping criteria ❶. If `d`, the length of segment $\overline{AB}$, is greater than 10 pixels, you continue the recursion. You do this by calling the `drawKochSF()` function again—four times! With each call, you pass in a different set of arguments corresponding to the coordinates for one of the four line segments that make up a flake, which you calculated at the start of the function. At ❷, for

example, you call `drawKochSF()` for the segment $\overline{AB}_1$. The other function calls are for segments $\overline{P_1P_2}$, $\overline{P_2P_3}$, and $\overline{P_3B}$. Within each of these recursive calls, you'll perform a new set of calculations based on the new values for points *A* and *B*, and if `d` is still greater than 10 pixels, you'll make another four recursive calls to `drawKochSF()`, and so on.

Drawing a Flake

Now let's look at what happens if segment $\overline{AB}$ is less than 10 pixels. This is the base case for the recursive algorithm. Since you're below the threshold, you aren't going to recurse. Instead, you actually draw the four line segments that make up a single flake pattern and return from the function. You use the `up()`, `down()`, and `setpos()` methods from the `turtle` module, which you learned about in the "Drawing with `turtle` Graphics" section.

```
    else:
        # draw cone
        t.up()
      ❶ t.setpos(p1[0], p1[1])
        t.down()
        t.setpos(p2[0], p2[1])
        t.setpos(p3[0], p3[1])
        # draw sides
        t.up()
      ❷ t.setpos(x1, y1)
        t.down()
        t.setpos(p1[0], p1[1])
        t.up()
      ❸ t.setpos(p3[0], p3[1])
        t.down()
        t.setpos(x2, y2)
```

First you draw the cone formed by points `p1`, `p2`, and `p3` ❶. Then you draw lines $\overline{AB}_1$ ❷ and $\overline{P_3B}$ ❸. Since you already performed all the required calculations at the start of the function, drawing is simply a matter of passing the appropriate coordinates to the `setpos()` method.

Writing the main() Function

The `main()` function sets up a `turtle` object and calls `drawKochSF()`.

```
def main():
    print('Drawing the Koch Snowflake...')

    t = turtle.Turtle()
    t.hideturtle()

    # draw
    try:
      ❶ drawKochSF(-100, 0, 100, 0, t)
      ❷ drawKochSF(0, -173.2, -100, 0, t)
      ❸ drawKochSF(100, 0, 0, -173.2, t)
  ❹ except:
        print("Exception, exiting.")
        exit(0)

    # wait for user to click on screen to exit
  ❺ turtle.Screen().exitonclick()
```

In Figure 1-4, you saw how you were going to draw three of the snowflakes to get a hexagonally symmetric image as the final output. You do this by making three calls to `drawKochSF()`. The coordinates used for points *A* and *B* are `(-100, 0)`, `(100, 0)` for the first snowflake ❶, `(0, -173.2)`, `(-100, 0)` for the second ❷, and `(100, 0)`, `(0, -173.2)` for the third ❸. Notice that these are the same coordinates you used earlier to draw a triangle in your *test_turtle.py*

program. Try to work out the coordinates for yourself. (Hint: $-173.2 \approx 100\sqrt{3}$.)

The `drawKochSF()` calls are enclosed in a Python `try` block to catch any exceptions that happen during drawing. For example, if you close the window while the drawing is still in process, an exception is thrown. You catch it in the `except` block ❹, where you print a message and exit the program. If you allow the drawing to complete, you'll get to `turtle.Screen().exitonclick()` ❺, which will wait until you close the window by clicking anywhere inside it.

Running the Snowflake Code

Run the code in a terminal as follows. Figure 1-6 shows the output.

```
$ python koch.py
```

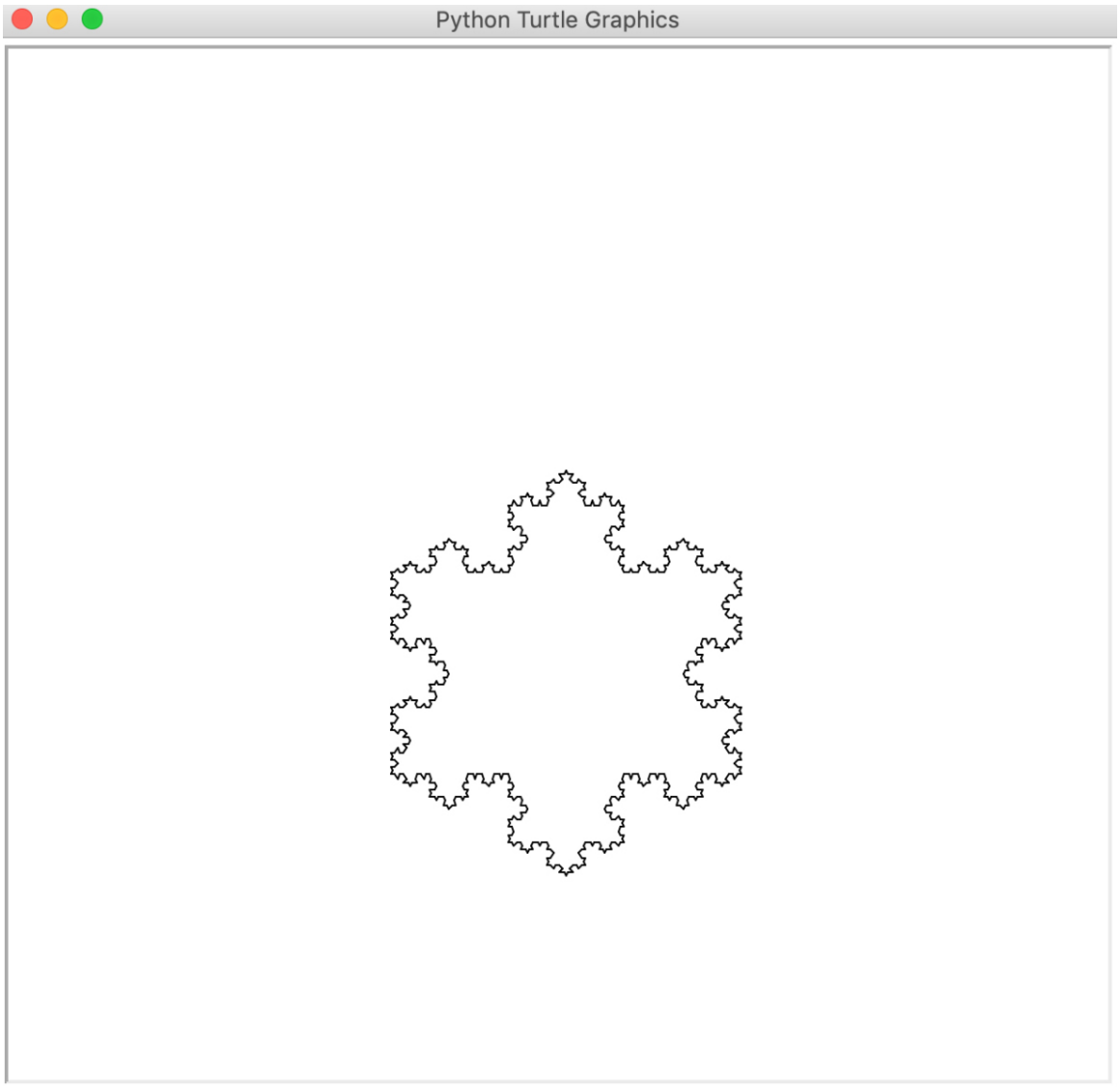


Figure 1-6: *The Koch snowflake output*

There’s your beautiful snowflake!

Summary

In this chapter, you learned the basics of recursive functions and algorithms. You also learned how to draw simple graphics with Python’s `turtle` module. You put these

concepts together to create a nice drawing of an interesting fractal called the Koch snowflake.

Experiments!

Now that you have completed one fractal drawing, let’s look at another interesting one called the *Sierpiński triangle*, named after the Polish mathematician Wacław Sierpiński. Figure 1-7 shows what it looks like.

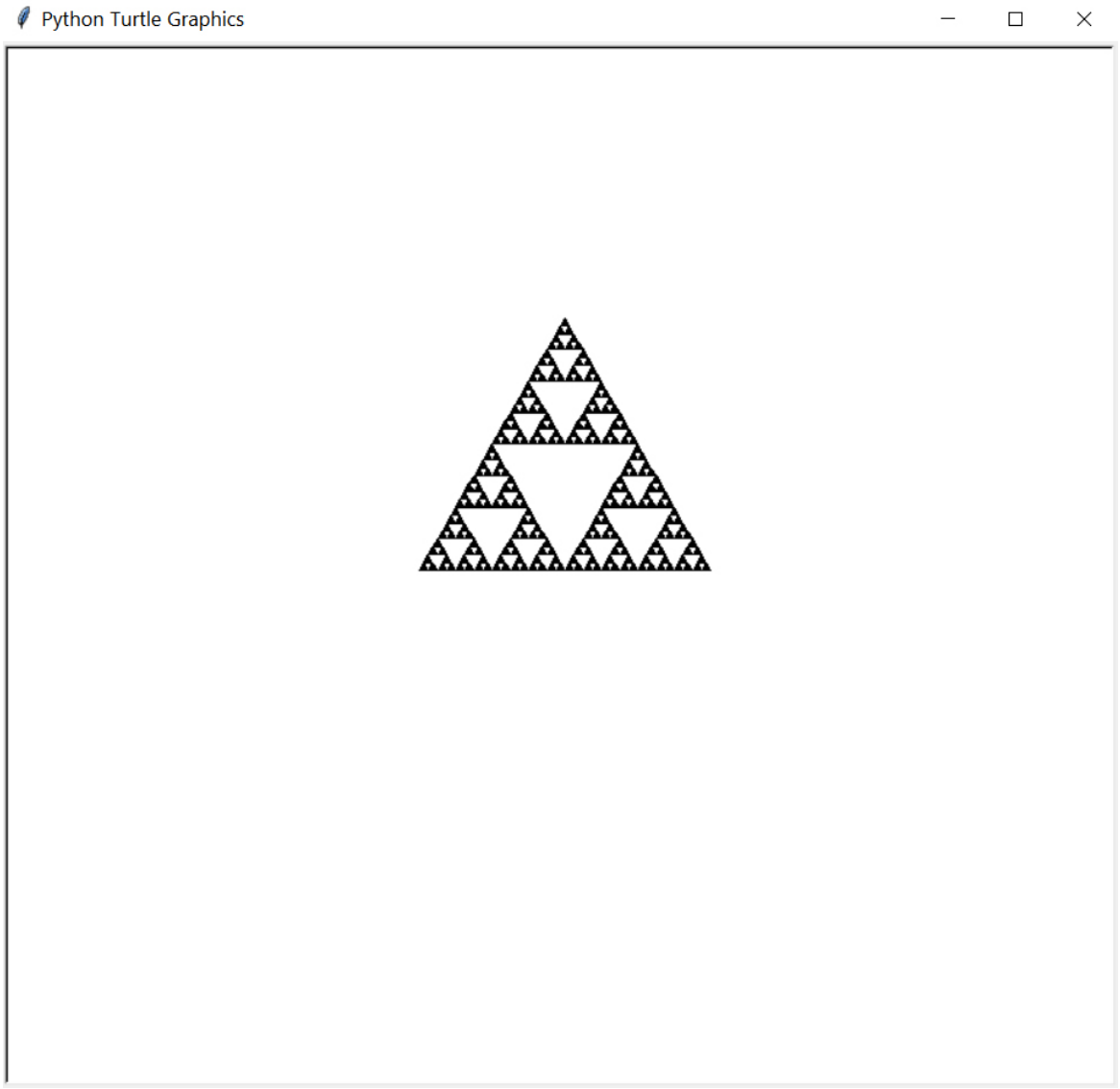


Figure 1-7: *The Sierpiński triangle*

Try drawing the Sierpiński triangle with `turtle` graphics. You can use a recursive algorithm like you did to draw the Koch snowflake. If you look at Figure 1-7, you'll see that the large triangle is divided into three smaller triangles, with an upside-down triangular hole in the middle. Each of the three smaller triangles is itself divided into another three triangles plus a hole in the middle, and so on. That gives you a hint on how to split up your recursion.

(The solution to this problem is in the GitHub repository for the book *https://github.com/mkvenkit/pp2e/blob/main/koch/koch.py*)

The Complete Code

Here's the complete code listing for this project:

```
"""
koch.py

A program that draws the Koch snowflake.

Author: Mahesh Venkitachalam
"""

import turtle
import math

# draw the recursive Koch snowflake
def drawKochSF(x1, y1, x2, y2, t):
    d = math.sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2))
    r = d/3.0
    h = r*math.sqrt(3)/2.0
    p3 = ((x1 + 2*x2)/3.0, (y1 + 2*y2)/3.0)
    p1 = ((2*x1 + x2)/3.0, (2*y1 + y2)/3.0)
    c = (0.5*(x1+x2), 0.5*(y1+y2))
    n = ((y1-y2)/d, (x2-x1)/d)
    p2 = (c[0]+h*n[0], c[1]+h*n[1])
    if d > 10:
        # flake #1
        drawKochSF(x1, y1, p1[0], p1[1], t)
        # flake #2
        drawKochSF(p1[0], p1[1], p2[0], p2[1], t)
        # flake #3
        drawKochSF(p2[0], p2[1], p3[0], p3[1], t)
        # flake #4
        drawKochSF(p3[0], p3[1], x2, y2, t)
```

```
    else:
        # draw cone
        t.up()
        t.setpos(p1[0], p1[1])
        t.down()
        t.setpos(p2[0], p2[1])
        t.setpos(p3[0], p3[1])
        # draw sides
        t.up()
        t.setpos(x1, y1)
        t.down()
        t.setpos(p1[0], p1[1])
        t.up()
        t.setpos(p3[0], p3[1])
        t.down()
        t.setpos(x2, y2)

# main() function
def main():
    print('Drawing the Koch Snowflake...')

    t = turtle.Turtle()
    t.hideturtle()

    # draw
    try:
        drawKochSF(-100, 0, 100, 0, t)
        drawKochSF(0, -173.2, -100, 0, t)
        drawKochSF(100, 0, 0, -173.2, t)
    except:
        print("Exception, exiting.")
        exit(0)

    # wait for user to click on screen to exit
    turtle.Screen().exitonclick()

# call main
if __name__ == '__main__':
    main()
```

2
SPIROGRAPHS



You can use a Spirograph toy (shown in Figure 2-1) to draw mathematical curves. The toy consists of two different-sized rings with plastic gear teeth, one large and one small. The small one has several holes. You put a pen or pencil through one of the holes and then rotate the smaller wheel inside the larger one (which has gear teeth on its inside), keeping the wheels in contact with each other, to draw an endless number of complex and wonderfully symmetric patterns.

In this project, you'll use Python to create an animation of Spirograph-like curves. The program will use parametric equations to describe the motion of a Spirograph's rings and draw the curves (which I call *spiros*). You'll save the completed drawings as PNG image files. Either the program will draw random spiros or you can use command line options to draw a spiro with specific parameters.

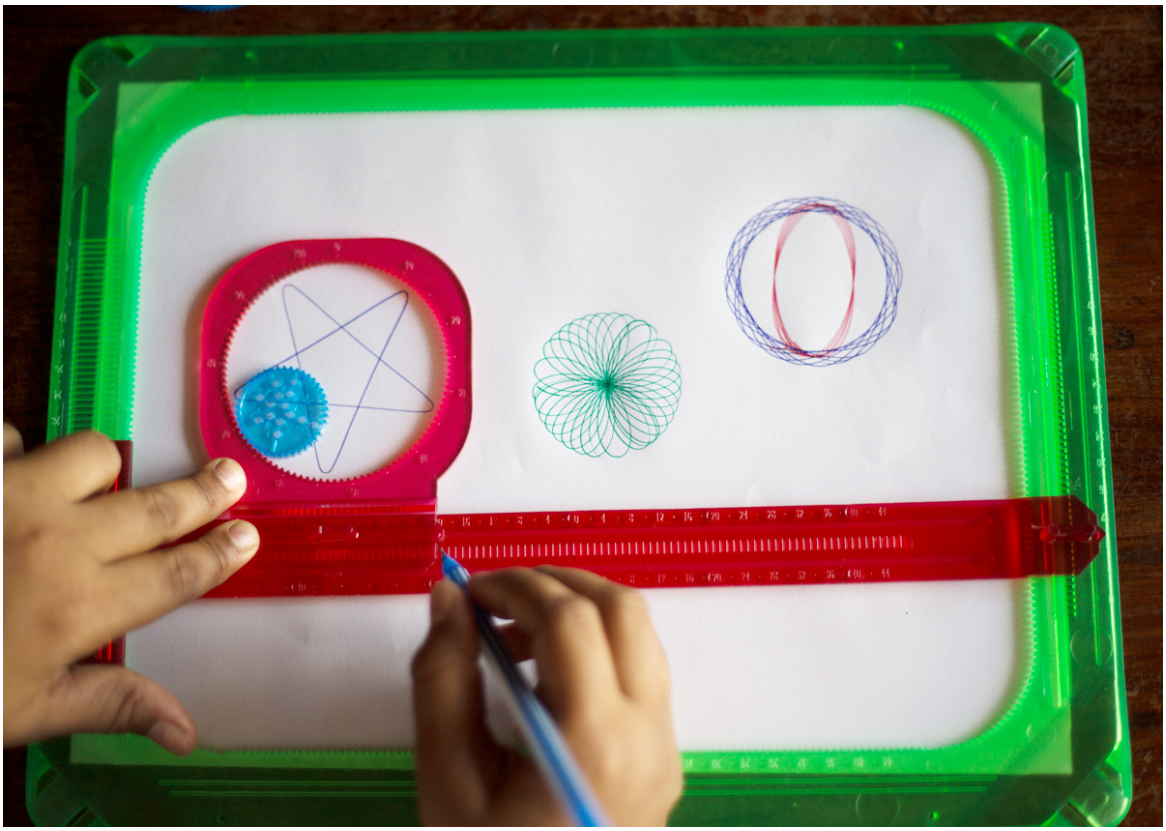


Figure 2-1: *A Spirograph toy*

In this project, you'll learn how to draw spiros on your computer. You'll also learn how to do the following:

- Use parametric equations to generate curves.
- Draw a curve as a series of straight lines using the `turtle` module.
- Use a timer to animate graphics.
- Save graphics to image files.

A word of caution: I've chosen to use the `turtle` module to draw spiros mainly for illustrative purposes and because it's fun, but `turtle` is slow and not ideal for creating graphics when performance is critical. (What do you expect from turtles?) If you want to draw something quickly, there are

better ways to do so, and you'll explore some of these options in upcoming projects.

How It Works

This project hinges around using *parametric equations*, equations that express the coordinates of the points on a curve as functions of one or more variables, called *parameters*. You'll plug values for the parameters into equations to calculate points that form a spiro pattern. Then you'll feed those points to the `turtle` module to draw the curves.

Understanding Parametric Equations

To understand how parametric equations work, we'll begin with a simple example: a circle. Consider a circle with radius r, centered at the origin of a two-dimensional plane. That circle consists of all the points whose x- and y-coordinates satisfy the equation $x^2 + y^2 = r^2$. This isn't a parametric equation, however. A parametric equation would give us all possible values of x and y, based on changes in some other variable (the parameter).

Now, consider the following equations:

$$x = r \cos(\theta)$$
$$y = r \sin(\theta)$$

Together, these equations are a *parametric* representation of our circle, where the parameter is θ, the angle of point (x, y) relative to the positive x-axis. Any value of (x, y) in these equations will satisfy the original $x^2 + y^2 = r^2$ equation. As you vary θ from 0 to 2π, the x- and y-coordinates generated

by these equations will form the circle. Figure 2-2 shows this scheme.

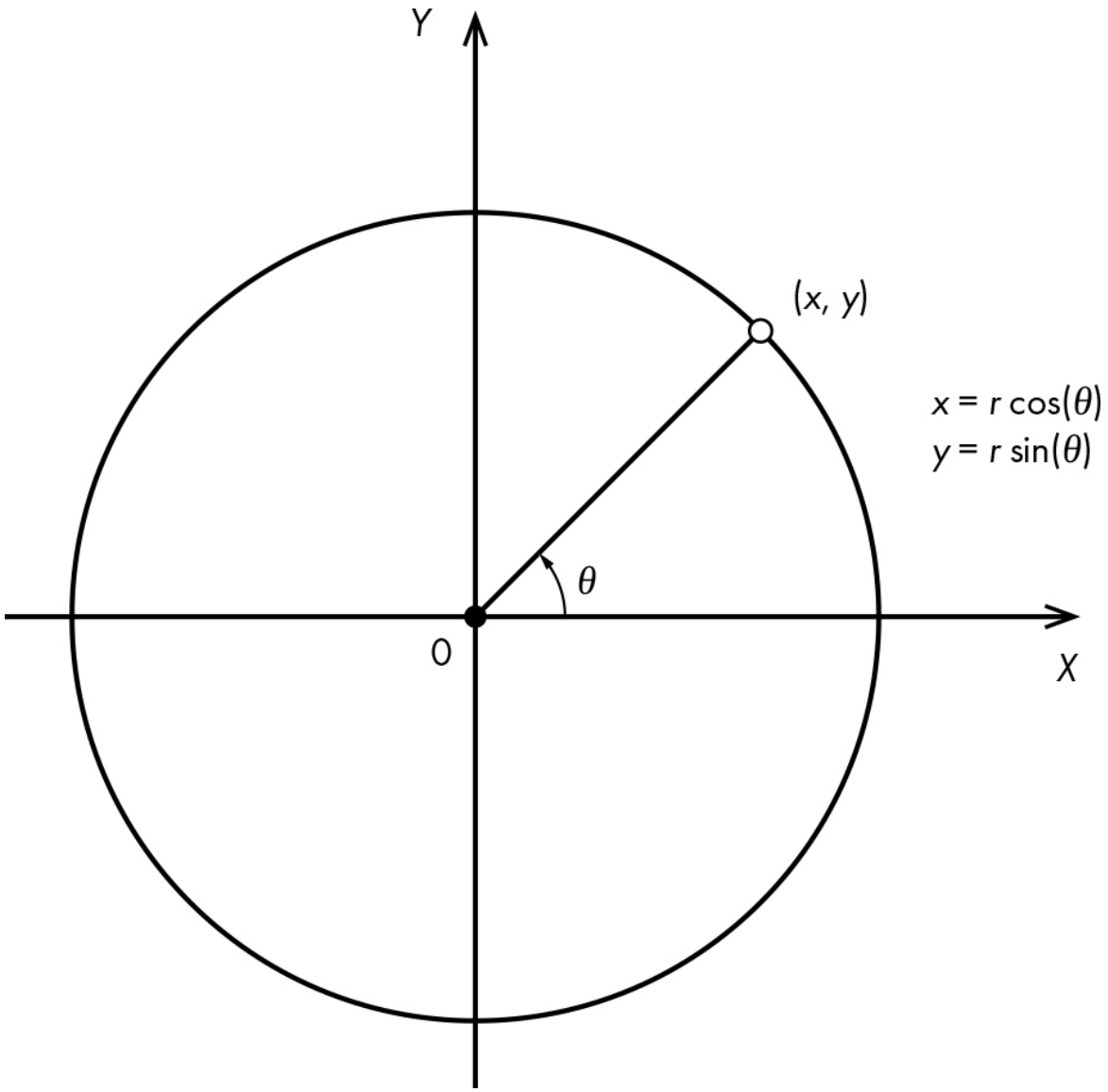


Figure 2-2: *Describing a circle with a parametric equation*

Remember, these two equations apply to a circle centered at the origin of the coordinate system. You can put a circle at any point in the XY plane by translating the center of the circle from the point (0, 0) to the point (a, b). The more general parametric equations then become $x = a + r\cos(\theta)$ and $y = b + r\sin(\theta)$.

Developing parametric equations that model a Spirograph toy isn't that much different from developing parametric equations for a circle, since at heart a Spirograph simply draws two interlocking circles. Figure 2-3 shows a mathematical model of Spirograph-like motion. The model has no gear teeth; they're used in Spirograph toys only to prevent slippage, and in the ideal world of mathematical modeling, you don't have to worry about anything slipping.

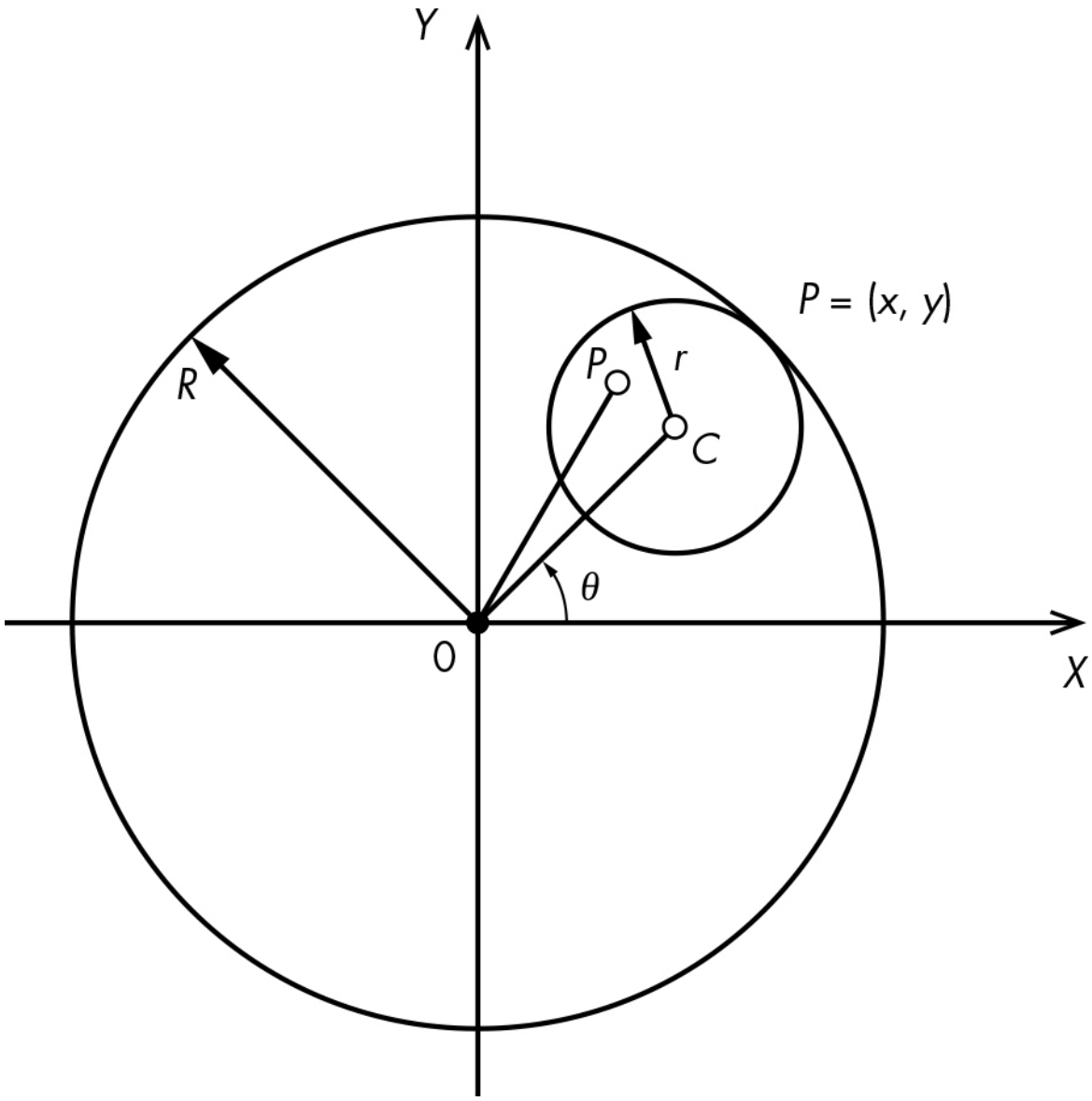


Figure 2-3: *A mathematical model of a Spirograph toy*

In Figure 2-3, *C* is the center of the smaller circle, *P* is the pen's tip, and *q* is the angle of *C* relative to the positive x-axis. The radius of the bigger circle is *R* and that of the smaller circle is *r*. You express the ratio of the radii as variable *k*, as follows:

$$k = \frac{r}{R}$$

Line segment $\overline{PC}$ tells you how far the pen tip is from the center of the smaller circle. You express the ratio of $\overline{PC}$ to the smaller circle's radius *r* as the variable *l*, like so:

$$l = \frac{\overline{PC}}{r}$$

You can now combine these variables into the following parametric equations that represent the x- and y-coordinates of point *P* (the pen) as the smaller circle rotates inside the larger one:

$$x = R\left((1 - \mathrm{k})\cos(\theta) + lk\cos\left(\frac{1 - \mathrm{k}}{k}\theta\right)\right)$$

$$y = R\left((1 - \mathrm{k})\sin(\theta) - lk\sin\left(\frac{1 - \mathrm{k}}{k}\theta\right)\right)$$

NOTE *These curves are called* hypotrochoids*. Although the equations may look a bit scary, the derivation is pretty straightforward. See the Wikipedia page on Spirographs if*

you'd like to explore the math: http://en.wikipedia.org/wiki/Spirograph*.*

Figure 2-4 shows an example curve drawn with these equations. For this curve, I set R to 220, r to 65, and l to 0.8. By choosing different values for these three parameters and then incrementing angle θ, you can produce an endless variety of fascinating curves.

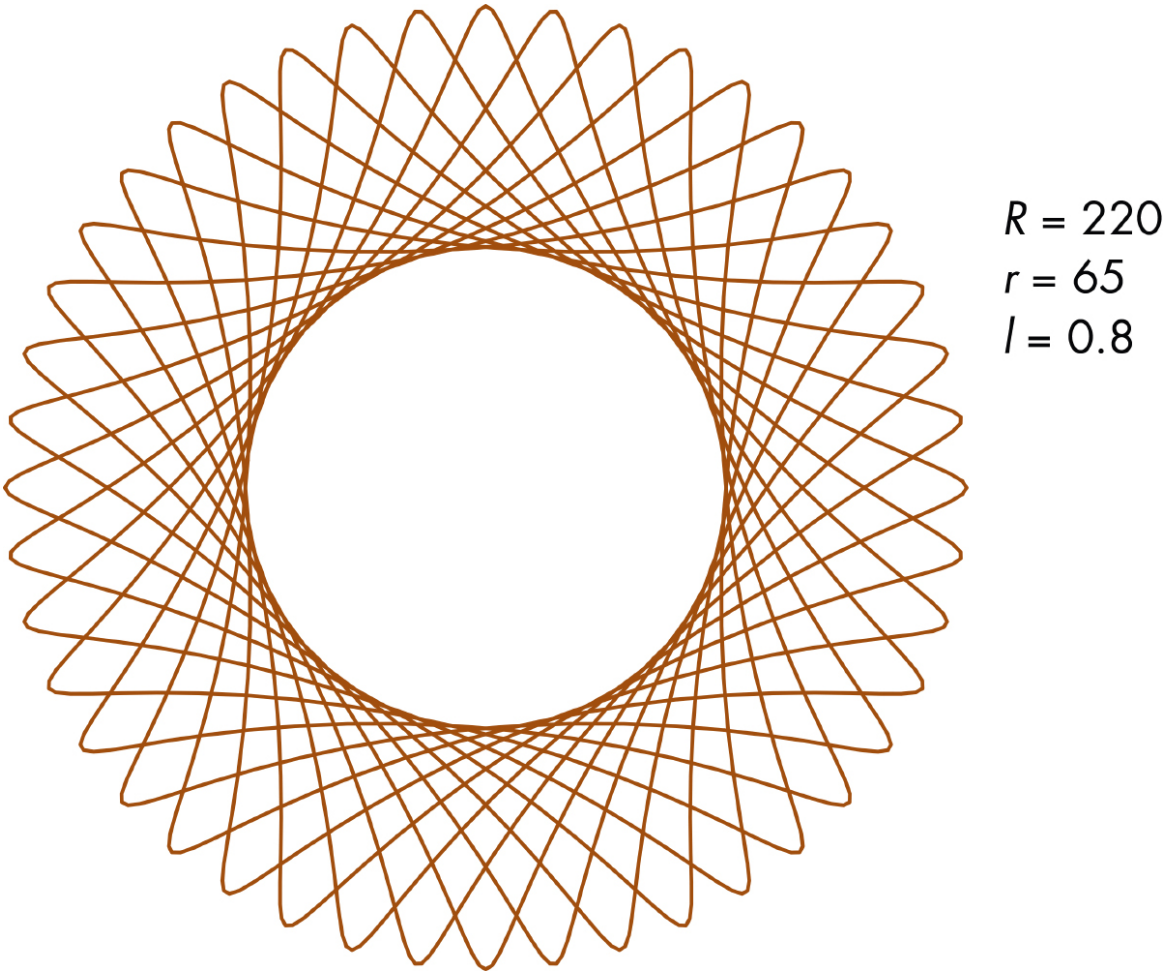


Figure 2-4: *A sample curve*

The only task left is to determine when to stop drawing, since Spirographs can require many revolutions of the smaller circle around the larger circle to form a complete pattern. You can calculate the *periodicity* of the Spirograph

(how long before the Spirograph starts repeating itself) by looking at the ratio of the radii of the inner and outer circles:

$$\frac{r}{R}$$

Reduce this fraction by dividing the numerator and denominator by the *greatest common divisor (GCD)*. Then the numerator tells you how many periods the curve needs to complete itself. For example, in Figure 2-4, the GCD of (*r*, *R*) is 5:

$$\frac{r}{R} = \frac{65}{220} = \frac{(65/5)}{(220/5)} = \frac{13}{44}$$

This tells you that the curve will start repeating itself after 13 revolutions of the smaller circle around the larger circle. The 44 in the denominator tells you the number of times the smaller circle revolves around its own center, which gives you a hint as to the shape of the curve. If you count the petals (or lobes) in the drawing in Figure 2-4, you'll see there are exactly 44!

Once you express the radii ratio in the reduced form *r*/*R*, the range for the parameter θ to draw the spiro is $[0, 2\pi r]$. This tells you when to stop drawing a particular spiro. In the case of Figure 2-4, you'd stop when θ reaches 26π (that is, $2\pi \times 13$). Without knowing the ending range of the angle, you'd end up looping around, repeating the curve unnecessarily.

Drawing Curves with turtle Graphics

Python's `turtle` module doesn't have a method for drawing curved lines. Instead, you'll draw a spiro as a collection of straight lines between different points calculated with the parametric equations discussed in the previous section. As long as the change in angle θ from one point to the next is relatively small, the result will appear curved.

To demonstrate, the following program draws a circle with `turtle`. It uses our basic parametric equations for a circle, $x = a + r\cos(\theta)$ and $y = b + r\sin(\theta)$, to calculate points along the circle, and it connects those points with straight lines. Technically, the program actually produces an N-sided polygon, but because the angle parameter will change in small increments, N will be very large, and the polygon will look like a circle. Enter the following code, save it as *drawcircle.py*, and run it in Python:

```
import math
import turtle

# draw the circle using turtle
def drawCircleTurtle(x, y, r):
    # move to the start of circle
    turtle.up()
  ❶ turtle.setpos(x + r, y)
    turtle.down()

    # draw the circle
  ❷ for i in range(0, 365, 5):
      ❸ a = math.radians(i)
      ❹ turtle.setpos(x + r*math.cos(a), y + r*math.sin(a))

drawCircleTurtle(100, 100, 50)
turtle.mainloop()
```

Here you define the `drawCircleTurtle()` function, whose parameters are the center of the circle to be drawn, (x, y), and the circle's radius, r. The function starts by moving the turtle into position at the first point on the circle's horizontal

axis: (x + r, y) ❶. The calls to `up()` and `down()` prevent the turtle from drawing while it's getting into position. Next, you start a loop using `range(0, 365, 5)`, which increments the variable `i` in steps of 5 from 0 to 360 ❷. The `i` variable is the angle parameter you'll pass into the parametric circle equations, but first you convert it from degrees to radians ❸. (Most computer programs require radians for angle-based calculations.)

Compute the next set of circle coordinates using the two parametric equations, and you set the turtle's position accordingly ❹. This draws a straight line from the last turtle position to the newly calculated one. Since you're changing the angle parameter by just 5 degrees at a time, the straight lines will create the appearance of a round circle.

Now that you have your function, you call it to draw a circle ❺. Calling `turtle.mainloop()` keeps the `tkinter` window open so that you can admire your work. (`tkinter` is the default GUI library used by Python.)

You're now ready to draw some spiros! You'll use the same `turtle` approach illustrated earlier. All that has to change are the details of the parametric equations used to calculate the points.

Requirements

You'll use the following to create your spiros:

- The `turtle` module for drawing
- `Pillow`, a fork of the *Python Imaging Library (PIL)*, to save the spiro images

The Code

First you'll define a class `Spiro` to draw the curves. You can use this class either to draw a single curve with customizable parameters or as part of an animation that draws several random spiros concurrently. To coordinate the animation, you'll define another class called `SpiroAnimator`. At the top level of the program, you'll write a function to save your drawings as image files, and you'll use the `main()` function to take in user input and set the drawing in motion.

To see the full project code, skip ahead to "The Complete Code" on page 36. You can also download the code for this project from *https://github.com/mkvenkit/pp2e/blob/main/spirograph/spiro.py*.

Drawing the Spiros

The `Spiro` class features methods for drawing an individual spiro pattern. Here's the `Spiro` class's constructor:

```
class Spiro:
    # constructor
    def __init__(self, xc, yc, col, R, r, l):

        # create the turtle object
      ❶ self.t = turtle.Turtle()
        # set the cursor shape
      ❷ self.t.shape('turtle')
        # set the step in degrees
      ❸ self.step = 5
        # set the drawing complete flag
      ❹ self.drawingComplete = False

        # set the parameters
        self.setparams(xc, yc, col, R, r, l)

        # initialize the drawing
```

```
        self.restart()
```

The `Spiro` constructor creates a new `turtle` object ❶. This way, each individual `Spiro` object will have its own `turtle` object associated with it, meaning you can create multiple `Spiro` objects to draw a bunch of spiros simultaneously. You set the shape of the turtle cursor to a turtle ❷. (You'll find other choices in the `turtle` documentation at *https://docs.python.org/3/library/turtle.xhtml*.) You set the angle increment for the parametric drawing to 5 degrees ❸ and create a Boolean `drawingComplete` flag for indicating when the spiro is done ❹. This flag will be useful when multiple `Spiro` objects are drawing in tandem; it allows you to keep track of whether a particular spiro is complete. You finish the constructor by calling two setup methods, as discussed next.

The Setup Methods

The `Spiro` class's `setparams()` and `restart()` methods both help with some setup that needs to be done before a spiro pattern can be drawn. Let's look at the `setparams()` method first:

```
def setparams(self, xc, yc, col, R, r, l):
    # the Spirograph parameters
    self.xc = xc
    self.yc = yc
    self.R = int(R)
    self.r = int(r)
    self.l = l
    self.col = col
    # reduce r/R to its smallest form by dividing with the GCD
  ❶ gcdVal = math.gcd(self.r, self.R)
  ❷ self.nRot = self.r//gcdVal
    # get ratio of radii
```

```
    self.k = r/float(R)
    # set the color
    self.t.color(*col)
    # store the current angle
  ❸ self.a = 0
```

First you store the coordinates of the center of the spiro (`xc` and `yc`). Then you convert the radius of each circle (`R` and `r`) to an integer and store the values. You also store `l`, which defines the position of the pen, and `col`, which determines the spiro's color. Next, you use the `gcd()` method from Python's built-in `math` module to compute the GCD of the radii ❶. You use this information to determine the periodicity of the curve, which you save as `self.nRot` ❷. Finally, you set the starting value of the angle parameter, `a`, to `0` ❸.

The `restart()` method continues with the setup by resetting the drawing parameters for the `Spiro` object and getting it into position to draw a spiro. This method makes it possible to reuse the same `Spiro` object to draw multiple spiros, one after the other, as part of the program's animation. The program will call `restart()` each time the object is ready to draw a new spiro. Here's the method:

```
def restart(self):
    # set the flag
    self.drawingComplete = False
    # show the turtle
    self.t.showturtle()
    # go to the first point
  ❶ self.t.up()
  ❷ R, k, l = self.R, self.k, self.l
    a = 0.0
  ❸ x = R*((1-k)*math.cos(a) + l*k*math.cos((1-k)*a/k))
    y = R*((1-k)*math.sin(a) - l*k*math.sin((1-k)*a/k))
  ❹ self.t.setpos(self.xc + x, self.yc + y)
  ❺ self.t.down()
```

You reset the `drawingComplete` flag to `False`, indicating the object is ready to draw a new spiro. Then you show the turtle cursor, in case it was hidden. Next lift up the pen ❶ so you can move to the first position at ❹ without drawing a line. At ❷, you're just using some local variables to keep the code compact. Then you feed those variables to the spiro parametric equations to compute the x- and y-coordinates of the curve's starting point, using `0` as an initial value for angle `a` ❸. Finally, once the turtle is in place, you set the pen down so the turtle can start drawing the spiro ❺.

The draw() Method

If you use command line options to set the parameters of a spiro, the program will draw just that one spiro, using the `Spiro` class's `draw()` method. This method draws the entire spiro all in one go, as a continuous sequence of straight-line segments:

```
def draw(self):
    # draw the rest of the points
    R, k, l = self.R, self.k, self.l
  ❶ for i in range(0, 360*self.nRot + 1, self.step):
        a = math.radians(i)
      ❷ x = R*((1-k)*math.cos(a) + l*k*math.cos((1-k)*a/k))
        y = R*((1-k)*math.sin(a) - l*k*math.sin((1-k)*a/k))
        try:
          ❸ self.t.setpos(self.xc + x, self.yc + y)
        except:
            print("Exception, exiting.")
            exit(0)
    # drawing is now done so hide the turtle cursor
  ❹ self.t.hideturtle()
```

Here you iterate through the complete range of the parameter `i`, which is expressed in degrees as 360 times `nRot` ❶. You use the parametric equations to compute the x- and y-coordinates for each value of the `i` parameter ❷, calling the turtle's `setpos()` method ❸ to draw the line from one point to the next. This method is enclosed in a `try` block so that if an exception arises—such as the user closing the window in the middle of the drawing process—you can catch it and exit gracefully. Finally, you hide the cursor because you've finished drawing ❹.

The update() Method

If you don't use any command line options, the program will draw multiple random spiros as an animation. This approach requires some restructuring of the drawing code we just looked at. Instead of drawing an entire spiro in one go, you need a method that draws just a single-line segment of the spiro. Then you'll call that method at every time step in the animation. This `update()` method of the `Spiro` class fits the bill:

```
def update(self):
    # skip the rest of the steps if done
  ❶ if self.drawingComplete:
        return
    # increment the angle
  ❷ self.a += self.step
    # draw a step
    R, k, l = self.R, self.k, self.l
    # set the angle
  ❸ a = math.radians(self.a)
    x = self.R*((1-k)*math.cos(a) + l*k*math.cos((1-k)*a/k))
    y = self.R*((1-k)*math.sin(a) - l*k*math.sin((1-k)*a/k))

    try:
      ❹ self.t.setpos(self.xc + x, self.yc + y)
```

```
    except:
        print("Exception, exiting.")
        exit(0)
    # if drawing is complete, set the flag
❺   if self.a >= 360*self.nRot:
        self.drawingComplete = True
        # drawing is now done so hide the turtle cursor
        self.t.hideturtle()
```

You first check to see whether the `drawingComplete` flag is set ❶; if not, you continue through the rest of the code. You increment the current angle ❷, calculate the (*x*, *y*) position corresponding to the current angle ❸, and move the turtle there, drawing the line segment in the process ❹. This is just like the code inside the `for()` loop in the `draw()` method, except it happens only once.

When I discussed the Spirograph parametric equations, I talked about the periodicity of the curve. A Spirograph starts repeating itself after a certain angle. You finish the `update()` function by checking whether the angle has reached the full range computed for this particular curve ❺. If so, you set the `drawingComplete` flag because the spiro is finished. Finally, you hide the turtle cursor so you can see your beautiful creation.

Coordinating the Animation

The `SpiroAnimator` class will let you draw several random spiros simultaneously as an animation. This class coordinates the activity of multiple `Spiro` objects with randomly assigned parameters, using a timer to regularly call each `Spiro` object's `update()` method. This technique updates the graphics periodically and lets the program process events such as button presses, mouse clicks, and so on.

Let's look at the `SpiroAnimator` class's constructor first:

```
class SpiroAnimator:
    # constructor
    def __init__(self, N):
        # set the timer value in milliseconds
      ❶ self.deltaT = 10
        # get the window dimensions
      ❷ self.width = turtle.window_width()
        self.height = turtle.window_height()
        # restarting
      ❸ self.restarting = False
        # create the Spiro objects
        self.spiros = []
        for i in range(N):
            # generate random parameters
          ❹ rparams = self.genRandomParams()
            # set the spiro parameters
          ❺ spiro = Spiro(*rparams)
            self.spiros.append(spiro)
        # call timer
      ❻ turtle.ontimer(self.update, self.deltaT)
```

The `SpiroAnimator` constructor sets `deltaT` to `10`, which is the time interval in milliseconds you'll use for the timer ❶. You then store the dimensions of the turtle window ❷ and initialize a flag that will be used to indicate that a restart is in progress ❸. In a loop that repeats *N* times (`N` is passed into `SpiroAnimator` in the constructor), you create new `Spiro` objects ❺ and add them to a `spiros` list. Before creating each `Spiro` object, you call the `genRandomParams()` helper method ❹ to randomly assign the spiro's parameters (we'll look at this method next). The `rparams` here is a tuple that you need to pass into the `Spiro` constructor. However, the constructor expects multiple arguments, so you use the Python `*` operator to unpack the tuple into a series of arguments.

Finally, you set the `turtle.ontimer()` method to call `update()` after `deltaT` milliseconds ❻ , which sets the animation in motion.

Generating Random Parameters

You'll use the `genRandomParams()` method to generate random parameters to send to each `Spiro` object as it's created to produce a wide variety of curves. You'll also call this method each time a `Spiro` object finishes drawing a spiro and is ready to start drawing a new one:

```
def genRandomParams(self):
    width, height = self.width, self.height
    R = random.randint(50, min(width, height)//2)
    r = random.randint(10, 9*R//10)
    l = random.uniform(0.1, 0.9)
    xc = random.randint(-width//2, width//2)
    yc = random.randint(-height//2, height//2)
    col = (random.random(),
           random.random(),
           random.random())
  ❶ return (xc, yc, col, R, r, l)
```

To generate random numbers, you use three methods from Python's `random` module: `randint()`, which returns random integers in the specified range; `uniform()`, which does the same for floating-point numbers; and `random()`, which returns a float between 0 and 1. You set `R` to a random integer between 50 and the value of half the smallest dimension of your window, and you set `r` to between 10 percent and 90 percent of `R`. Then you set `l` to a random fraction between 0.1 and 0.9.

You next select a random point on the screen to place the center of the spiro by selecting random x- and y-coordinates

(`xc` and `yc`) from within the screen boundaries. You assign a random color `col` to the curve by setting random values for the red, green, and blue color components (these are defined on a scale from 0 to 1). Finally, all of your calculated parameters are returned as a tuple ❶.

Restarting the Animation

The `SpiroAnimator` class has its own `restart()` method for restarting the animation to draw a new set of spiros:

```
def restart(self):
    # ignore restart if already in the middle of restarting
  ❶ if self.restarting:
        return
    else:
        self.restarting = True
    for spiro in self.spiros:
        # clear
        spiro.clear()
        # generate random parameters
        rparams = self.genRandomParams()
        # set the spiro parameters
        spiro.setparams(*rparams)
        # restart drawing
        spiro.restart()
    # done restarting
  ❷ self.restarting = False
```

This method loops through all the `Spiro` objects. For each one, you clear the previous drawing and randomly generate a new set of spiro parameters. Then you use the `Spiro` object's setup methods, `setparams()` and `restart()`, to assign the new parameters and get the object ready to draw the next spiro. The `self.restarting` flag ❶ prevents this method from being called before it's had a chance to complete,

which could happen if the user repeatedly hits the spacebar. The flag is reset at the end of the method so that the next restart won't be ignored ❷.

Updating the Animation

The following code shows the `update()` method in `SpiroAnimator`, which is called by the timer every 10 milliseconds to update all the `Spiro` objects used in the animation:

```
def update(self):
    # update all spiros
  ❶ nComplete = 0
    for spiro in self.spiros:
        # update
      ❷ spiro.update()
        # count completed spiros
      ❸ if spiro.drawingComplete:
            nComplete += 1
    # restart if all spiros are complete
  ❹ if nComplete == len(self.spiros):
        self.restart()
    # call the timer
    try:
      ❺ turtle.ontimer(self.update, self.deltaT)
    except:
        print("Exception, exiting.")
        exit(0)
```

The `update()` method uses a counter `nComplete` to track the number of `Spiro` objects that have finished drawing ❶. The method loops through the list of `Spiro` objects and updates them ❷, which draws one more line segment in each spiro. You increment the counter if a `Spiro` is done ❸.

Outside the loop, you check the counter to determine whether all the objects have finished drawing ❹. If so, you restart the animation with fresh spiros by calling the `restart()` method. The `update()` method ends with a call to the `turtle` module's `ontimer()` method ❺, which calls `update()` again after `deltaT` milliseconds. This is what keeps the animation going.

Showing or Hiding the Cursor

You use the following method of the `SpiroAnimator` class to toggle the turtle cursors on and off. Turning them off makes the drawing go more quickly.

```
def toggleTurtles(self):
    for spiro in self.spiros:
        if spiro.t.isvisible():
            spiro.t.hideturtle()
        else:
            spiro.t.showturtle()
```

This method uses built-in `turtle` methods to hide the cursor if it's visible or to show the cursor if it isn't. Later, you'll see how this `toggleTurtles()` method is triggered by keypresses while the animation is running.

Saving the Curves

After all your hard work generating spiros, it would be nice to have a way to save the results. The stand-alone `saveDrawing()` function saves the contents of the drawing window as a PNG image file:

```
def saveDrawing():
    # hide the turtle cursor
```

```
    ❶ turtle.hideturtle()
      # generate unique filenames
    ❷ dateStr = (datetime.now()).strftime("%d%b%Y-%H%M%S")
      fileName = 'spiro-' + dateStr
      print('saving drawing to {}.eps/png'.format(fileName))
      # get the tkinter canvas
      canvas = turtle.getcanvas()
      # save the drawing as a postscript image
    ❸ canvas.postscript(file = fileName + '.eps')
      # use the Pillow module to convert the postscript image
file to PNG
    ❹ img = Image.open(fileName + '.eps')
    ❺ img.save(fileName + '.png', 'png')
      # show the turtle cursor
      turtle.showturtle()
```

You hide the turtle cursors so that you won't see them in the final drawing ❶. Then you use `datetime()` to generate unique, timestamp-based names for the image files (in the *day-month-year-hour-minute-second* format) ❷. You append this string to *spiro-* to generate the filename.

The `turtle` program uses user interface (UI) windows created by `tkinter`, and you use the `canvas` object of `tkinter` to save the window in the Embedded PostScript (EPS) file format ❸. Because EPS is vector based, you can use it to print your images at high resolution, but the PNG format is more versatile, so you use `Pillow` to open the EPS file ❹ and save it as a PNG file ❺. Finally, you unhide the turtle cursors.

Parsing Command Line Arguments and Initialization

Most projects in this book have command line arguments for customizing the code. Rather than trying to parse them by hand and creating a mess, delegate this mundane task to

Python's `argparse` module. That's what you do in the first part of the spiro program's `main()` function:

```
def main():
  ❶ parser = argparse.ArgumentParser(description=descStr)

    # add expected arguments
  ❷ parser.add_argument('--sparams', nargs=3, dest='sparams',
  required=False,
                        help="The three arguments in sparams:
  R, r, l.")

    # parse args
  ❸ args = parser.parse_args()
```

You create an `ArgumentParser` object to manage the command line arguments ❶. Then you add the `--sparams` argument to the parser ❷. It consists of three components, for the *R*, *r*, and *l* parameters of a spiro. You use the `dest` option to specify the variable name the values should be stored under once the arguments are parsed, and `required=False` means this argument is optional. You call the `parse_args()` method ❸ to actually parse the arguments. This makes the arguments available as properties of the `args` object. In this case, the values of the `--sparams` argument will be available through `args.sparams`.

NOTE *You'll follow the same basic pattern described here throughout the book to create and parse each project's command line arguments.*

The `main()` function continues by setting up some `turtle` parameters:

```
    # set the width of the drawing window to 80 percent of the
  screen width
  ❶ turtle.setup(width=0.8)
```

```
    # set the cursor shape to turtle
    turtle.shape('turtle')

    # set the title to Spirographs!
    turtle.title("Spirographs!")
    # add the key handler to save our drawings
❷   turtle.onkey(saveDrawing, "s")
    # start listening
❸   turtle.listen()

    # hide the main turtle cursor
❹   turtle.hideturtle()
```

You use `setup()` to set the width of the drawing window to 80 percent of the screen width ❶. (You could also give `setup()` specific height and origin parameters.) Then you set the cursor shape to a turtle and set the title of the program window to *Spirographs!* Next, you use `onkey()` with your `saveDrawing()` function to instruct the program to save the drawing when you press the S key on your keyboard ❷. Calling `listen()` makes the drawing window listen for user events (like keypresses) ❸. Finally, you hide the turtle cursor ❹.

The rest of the `main()` function proceeds as follows:

```
    # check for any arguments sent to --sparams and draw the
  Spirograph
❶   if args.sparams:
❷       params = [float(x) for x in args.sparams]
        # draw the Spirograph with the given parameters
        col = (0.0, 0.0, 0.0)
❸       spiro = Spiro(0, 0, col, *params)
❹       spiro.draw()
    else:
        # create the animator object
❺       spiroAnim = SpiroAnimator(4)
        # add a key handler to toggle the turtle cursor
        turtle.onkey(spiroAnim.toggleTurtles, "t")
```

```
        # add a key handler to restart the animation
        turtle.onkey(spiroAnim.restart, "space")

    # start the turtle main loop
❻   turtle.mainloop()
```

You first check whether any arguments were given to `--sparams` ❶; if so, the program will just draw the one spiro defined by those arguments. The arguments currently exist as strings, but you need them to be interpreted as numbers. You use a list comprehension to convert them into a list of floats ❷. (A *list comprehension* is a Python construct that lets you create a list in a compact and powerful way. For example, `a = [2*x for x in range(1, 5)]` creates a list of the first four even numbers.) Then you use the parameters to construct a `Spiro` object ❸ (with the help of the Python `*` operator, which unpacks the list into a series of arguments) and call `draw()` to draw the spiro ❹.

If no arguments were specified at the command line, you enter random animation mode. For this, you create a `SpiroAnimator` object ❺, passing it the argument `4`, which tells it to draw four spiros at once. Then you use two `onkey` calls to capture additional keypresses. Pressing the T key will show or hide the turtle cursors with the `toggleTurtles()` method, while pressing the spacebar (`space`) will call `restart()` to interrupt the animation at any point and start drawing four different random spiros. Finally, you call `mainloop()` to tell the `tkinter` window to stay open, listening for events ❻.

Running the Spirograph Animation

Now it’s time to run your program:

```
$ python spiro.py
```

By default, the *spiro.py* program draws four random spiros simultaneously, as shown in Figure 2-5. Pressing S saves the drawing, pressing T toggles the cursors, and pressing the spacebar restarts the animation.

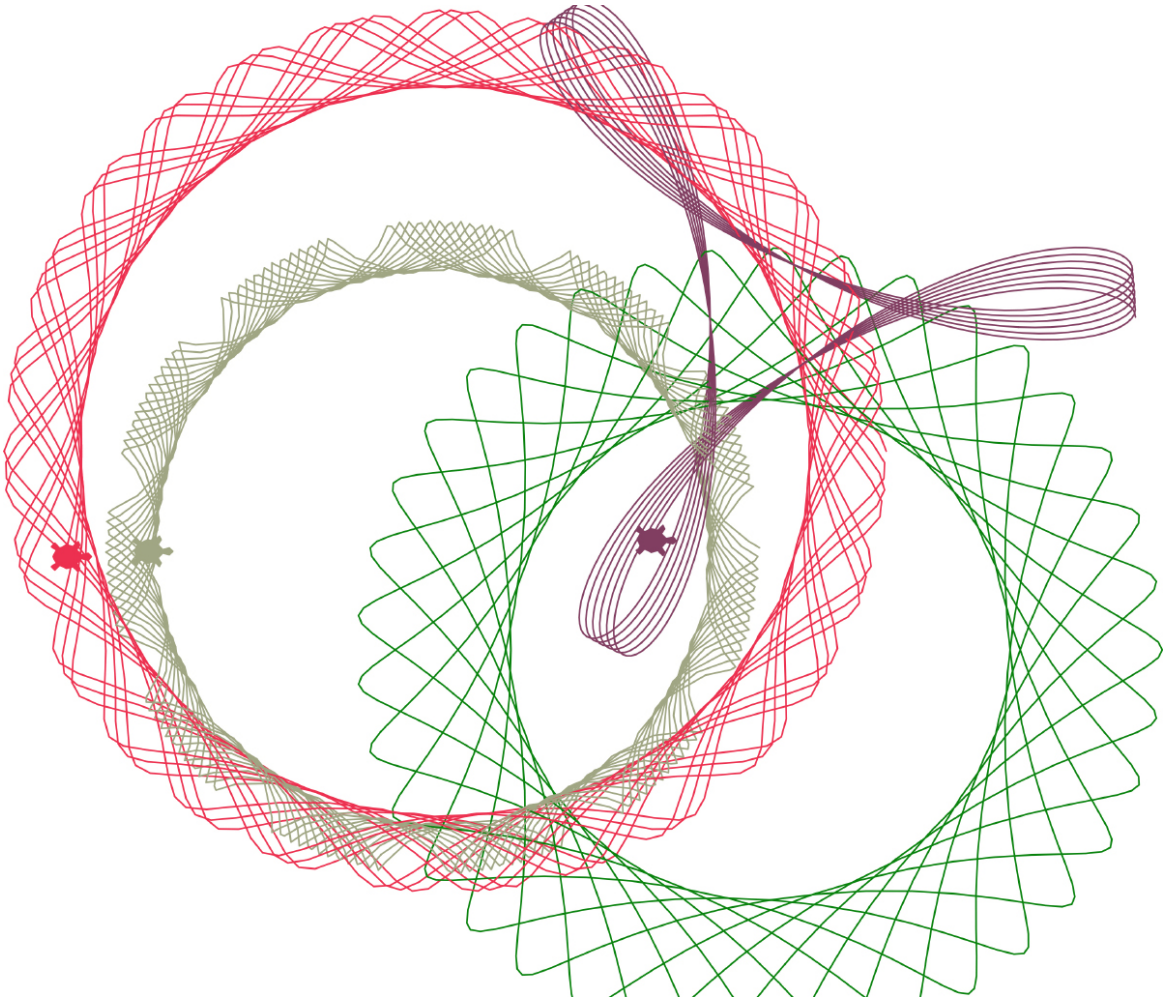


Figure 2-5: *A sample run of* spiro.py

Now run the program again, this time passing in parameters at the command line to draw a particular spiro:

```
$ python spiro.py --sparams 300 100 0.9
```

Figure 2-6 shows the output. As you can see, this code draws a single spiro with the parameters specified by the user, in contrast to Figure 2-5, which displays an animation of several random spiros.

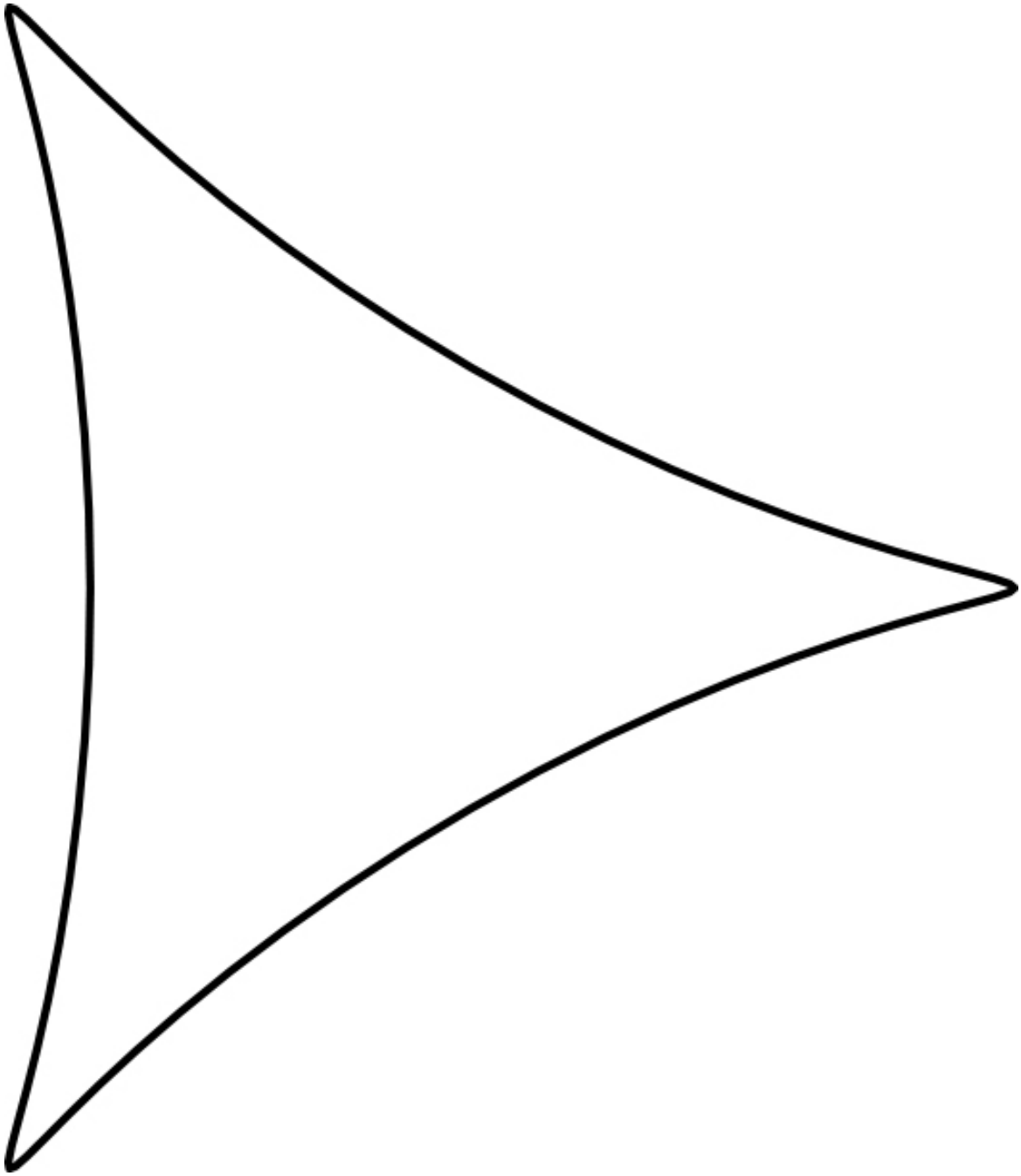


Figure 2-6: *A sample run of* spiro.py *with specific parameters*

Have fun experimenting with different parameters to see how they influence the resulting curves.

Summary

In this project, you learned how to create Spirograph-like curves. You also learned how to adjust the input parameters to generate a variety of different curves and to animate them on the screen. I hope you enjoy creating these spiros. (You'll find a surprise in Chapter 13, where you'll learn how to project spiros onto a wall!)

Experiments!

Here are some ways to experiment further with spiros:

1. Now that you know how to draw circles, write a program to draw random *spirals*. Find the equation for a *logarithmic spiral* in parametric form and then use it to draw the spirals.
2. You might have noticed that the turtle cursor is always oriented to the right as the curves are drawn, but that's not how turtles move! Orient the turtle so that, as the curve is being drawn, it faces in the direction of drawing. (Hint: calculate the direction vector between successive points for every step and reorient the turtle using the `turtle.setheading()` method.)

The Complete Code

Here’s the complete Spirograph program:

```
"""
spiro.py

A Python program that simulates a Spirograph.

Author: Mahesh Venkitachalam
"""

import random, argparse
import numpy as np
import math
import turtle
import random
from PIL import Image
from datetime import datetime

# a class that draws a spiro
class Spiro:
    # constructor
    def __init__(self, xc, yc, col, R, r, l):

        # create own turtle
        self.t = turtle.Turtle()
        # set cursor shape
        self.t.shape('turtle')
        # set step in degrees
        self.step = 5
        # set drawing complete flag
        self.drawingComplete = False

        # set parameters
        self.setparams(xc, yc, col, R, r, l)

        # initialize drawing
        self.restart()

    # set parameters
    def setparams(self, xc, yc, col, R, r, l):
        # spirograph parameters
        self.xc = xc
        self.yc = yc
```

```
        self.R = int(R)
        self.r = int(r)
        self.l = l
        self.col = col
        # reduce r/R to smallest form by dividing with GCD
        gcdVal = math.gcd(self.r, self.R)
        self.nRot = self.r//gcdVal
        # get ratio of radii
        self.k = r/float(R)
        # set color
        self.t.color(*col)
        # current angle
        self.a = 0

    # restart drawing
    def restart(self):
        # set flag
        self.drawingComplete = False
        # show turtle
        self.t.showturtle()
        # go to first point
        self.t.up()
        R, k, l = self.R, self.k, self.l
        a = 0.0
        x = R*((1-k)*math.cos(a) + l*k*math.cos((1-k)*a/k))
        y = R*((1-k)*math.sin(a) - l*k*math.sin((1-k)*a/k))
        try:
            self.t.setpos(self.xc + x, self.yc + y)
        except:
            print("Exception, exiting.")
            exit(0)
        self.t.down()

    # draw the whole thing
    def draw(self):
        # draw rest of points
        R, k, l = self.R, self.k, self.l
        for i in range(0, 360*self.nRot + 1, self.step):
            a = math.radians(i)
            x = R*((1-k)*math.cos(a) + l*k*math.cos((1-k)*a/k))
```

```
            y = R*((1-k)*math.sin(a) - l*k*math.sin((1-k)*a/k))
            try:
                self.t.setpos(self.xc + x, self.yc + y)
            except:
                print("Exception, exiting.")
                exit(0)
        # done - hide turtle
        self.t.hideturtle()

    # update by one step
    def update(self):
        # skip if done
        if self.drawingComplete:
            return
        # increment angle
        self.a += self.step
        # draw step
        R, k, l = self.R, self.k, self.l
        # set angle
        a = math.radians(self.a)
        x = self.R*((1-k)*math.cos(a) + l*k*math.cos((1-
k)*a/k))
        y = self.R*((1-k)*math.sin(a) - l*k*math.sin((1-
k)*a/k))
        try:
            self.t.setpos(self.xc + x, self.yc + y)
        except:
            print("Exception, exiting.")
            exit(0)
        # check if drawing is complete and set flag
        if self.a >= 360*self.nRot:
            self.drawingComplete = True
            # done - hide turtle
            self.t.hideturtle()

    # clear everything
    def clear(self):
        # pen up
        self.t.up()
        # clear turtle
```

```
        self.t.clear()

# a class for animating spiros
class SpiroAnimator:
    # constructor
    def __init__(self, N):
        # timer value in milliseconds
        self.deltaT = 10
        # get window dimensions
        self.width = turtle.window_width()
        self.height = turtle.window_height()
        # restarting
        self.restarting = False
        # create spiro objects
        self.spiros = []
        for i in range(N):
            # generate random parameters
            rparams = self.genRandomParams()
            # set spiro params
            spiro = Spiro(*rparams)
            self.spiros.append(spiro)
        # call timer
        turtle.ontimer(self.update, self.deltaT)

    # restart spiro drawing
    def restart(self):
        # ignore restart if already in the middle of restarting
        if self.restarting:
            return
        else:
            self.restarting = True
        # restart
        for spiro in self.spiros:
            # clear
            spiro.clear()
            # generate random parameters
            rparams = self.genRandomParams()
            # set spiro params
            spiro.setparams(*rparams)
            # restart drawing
```

```
            spiro.restart()
        # done restarting
        self.restarting = False

    # generate random parameters
    def genRandomParams(self):
        width, height = self.width, self.height
        R = random.randint(50, min(width, height)//2)
        r = random.randint(10, 9*R//10)
        l = random.uniform(0.1, 0.9)
        xc = random.randint(-width//2, width//2)
        yc = random.randint(-height//2, height//2)
        col = (random.random(),
               random.random(),
               random.random())
        return (xc, yc, col, R, r, l)

    def update(self):
        # update all spiros
        nComplete = 0
        for spiro in self.spiros:
            # update
            spiro.update()
            # count completed ones
            if spiro.drawingComplete:
                nComplete+= 1
        # if all spiros are complete, restart
        if nComplete == len(self.spiros):
            self.restart()
        # call timer
        try:
            turtle.ontimer(self.update, self.deltaT)
        except:
            print("Exception, exiting.")
            exit(0)

    # toggle turtle on/off
    def toggleTurtles(self):
        for spiro in self.spiros:
            if spiro.t.isvisible():
```

```
                spiro.t.hideturtle()
            else:
                spiro.t.showturtle()

# save spiros to image
def saveDrawing():
    # hide turtle
    turtle.hideturtle()
    # generate unique filename
    dateStr = (datetime.now()).strftime("%d%b%Y-%H%M%S")
    fileName = 'spiro-' + dateStr
    print('saving drawing to {}.eps/png'.format(fileName))
    # get tkinter canvas
    canvas = turtle.getcanvas()
    # save postscript image
    canvas.postscript(file = fileName + '.eps')
    # use PIL to convert to PNG
    img = Image.open(fileName + '.eps')
    img.save(fileName + '.png', 'png')
    # show turtle
    turtle.showturtle()

# main() function
def main():
    # use sys.argv if needed
    print('generating spirograph...')
    # create parser
    descStr = """This program draws spirographs using the
  Turtle module.
    When run with no arguments, this program draws random
  spirographs.

    Terminology:
    R: radius of outer circle.
    r: radius of inner circle.
    l: ratio of hole distance to r.
    """
    parser = argparse.ArgumentParser(description=descStr)

    # add expected arguments
```

```
    parser.add_argument('--sparams', nargs=3, dest='sparams',
  required=False,
                        help="The three arguments in sparams:
  R, r, l.")

    # parse args
    args = parser.parse_args()

    # set to 80% screen width
    turtle.setup(width=0.8)

    # set cursor shape
    turtle.shape('turtle')

    # set title
    turtle.title("Spirographs!")
    # add key handler for saving images
    turtle.onkey(saveDrawing, "s")
    # start listening
    turtle.listen()

    # hide main turtle cursor
    turtle.hideturtle()

    # check args and draw
    if args.sparams:
        params = [float(x) for x in args.sparams]
        # draw spirograph with given parameters
        # black by default
        col = (0.0, 0.0, 0.0)
        spiro = Spiro(0, 0, col, *params)
        spiro.draw()
    else:
        # create animator object
        spiroAnim = SpiroAnimator(4)
        # add key handler to toggle turtle cursor
        turtle.onkey(spiroAnim.toggleTurtles, "t")
        # add key handler to restart animation
        turtle.onkey(spiroAnim.restart, "space")

    # start turtle main loop
    turtle.mainloop()

# call main
```

```
if __name__ == '__main__':
    main()
```

PART II

SIMULATING LIFE

First, let's assume the cow is a sphere . . .

—Anonymous physics joke

3
CONWAY'S GAME OF LIFE



You can use a computer to study a system by creating a mathematical model for that system, writing a program to represent the model, and then letting the model evolve over time. There are many kinds of computer simulations, but I'll focus on a famous one called Conway's Game of Life, the work of the British mathematician John Conway. The Game of Life is an example of a *cellular automaton*, a collection of colored cells on a grid that evolve through a number of time steps according to a set of rules defining the states of neighboring cells.

In this project, you'll create an $N \times N$ grid of cells and simulate the evolution of the system over time by applying the rules of Conway's Game of Life. You'll display the state of the game at each time step and provide ways to save the output to a file. You'll set the initial condition of the system to either a random distribution or a predesigned pattern.

This simulation consists of the following components:

- A property defined in one- or two-dimensional space
- A mathematical rule to change this property for each step in the simulation

- A way to display or capture the state of the system as it evolves

The cells in Conway's Game of Life can be either ON or OFF. The game starts with an initial condition, in which each cell is assigned one of these states. Then, mathematical rules determine how each cell's state will change over time. The amazing thing about Conway's Game of Life is that with just four simple rules the system evolves to produce patterns that behave in incredibly complex ways, almost as if they were alive. Patterns include "gliders" that slide across the grid, "blinkers" that flash on and off, and even replicating patterns.

Of course, the philosophical implications of this game are also significant because they suggest that complex structures can evolve from simple rules without following any sort of preset pattern.

Here are some of the main concepts covered in this project:

- Using `matplotlib imshow` to represent a 2D grid of data
- Using `matplotlib` for animation
- Using the `numpy` array
- Using the `%` operator for boundary conditions
- Setting up a random distribution of values

How It Works

Because the Game of Life is built on a grid of nine squares, every cell has eight neighboring cells, as shown in Figure 3-1. A given cell (i, j) in the simulation is accessed on a grid [i][j], where i and j are the row and column indices,

respectively. The value of a given cell at a given instant of time depends on the state of its neighbors at the previous time step.

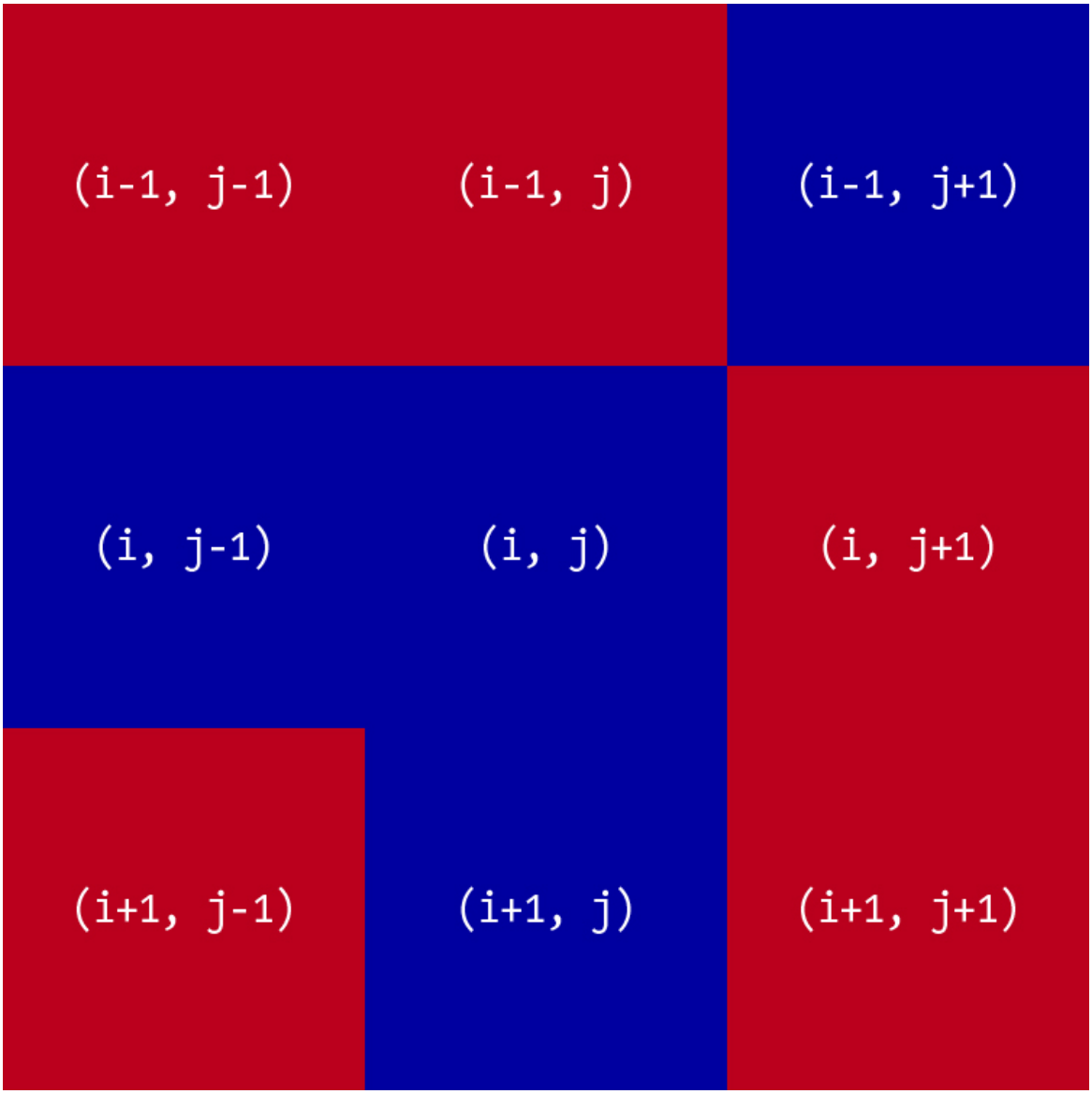


Figure 3-1: *A central cell (*i, j*) with eight neighboring cells*

Conway’s Game of Life has four rules:

1. If a cell is ON and has fewer than two neighbors that are ON, it turns OFF.

2. If a cell is ON and has either two or three neighbors that are ON, it remains ON.
3. If a cell is ON and has more than three neighbors that are ON, it turns OFF.
4. If a cell is OFF and has exactly three neighbors that are ON, it turns ON.

These rules are meant to mirror some basic ways that a group of organisms might fare over time: underpopulation and overpopulation kill cells by turning a cell OFF when it has fewer than two neighbors or more than three, but when the population is balanced, cells stay ON and reproduce by turning another cell from OFF to ON.

I said that each cell has eight neighboring cells, but what about cells at the edge of the grid? Which cells are their neighbors? To answer this question, you need to think about *boundary conditions*, the rules that govern what happens to cells at the edges or boundaries of the grid. I'll address this question by using *toroidal boundary conditions*, meaning that the square grid wraps around as if its shape were a torus. As shown in Figure 3-2, the grid is first warped so that its horizontal edges (A and B) join to form a cylinder, and then the cylinder's vertical edges (C and D) are joined to form a torus. Once the torus has been formed, all cells have neighbors because the whole space has no edge.

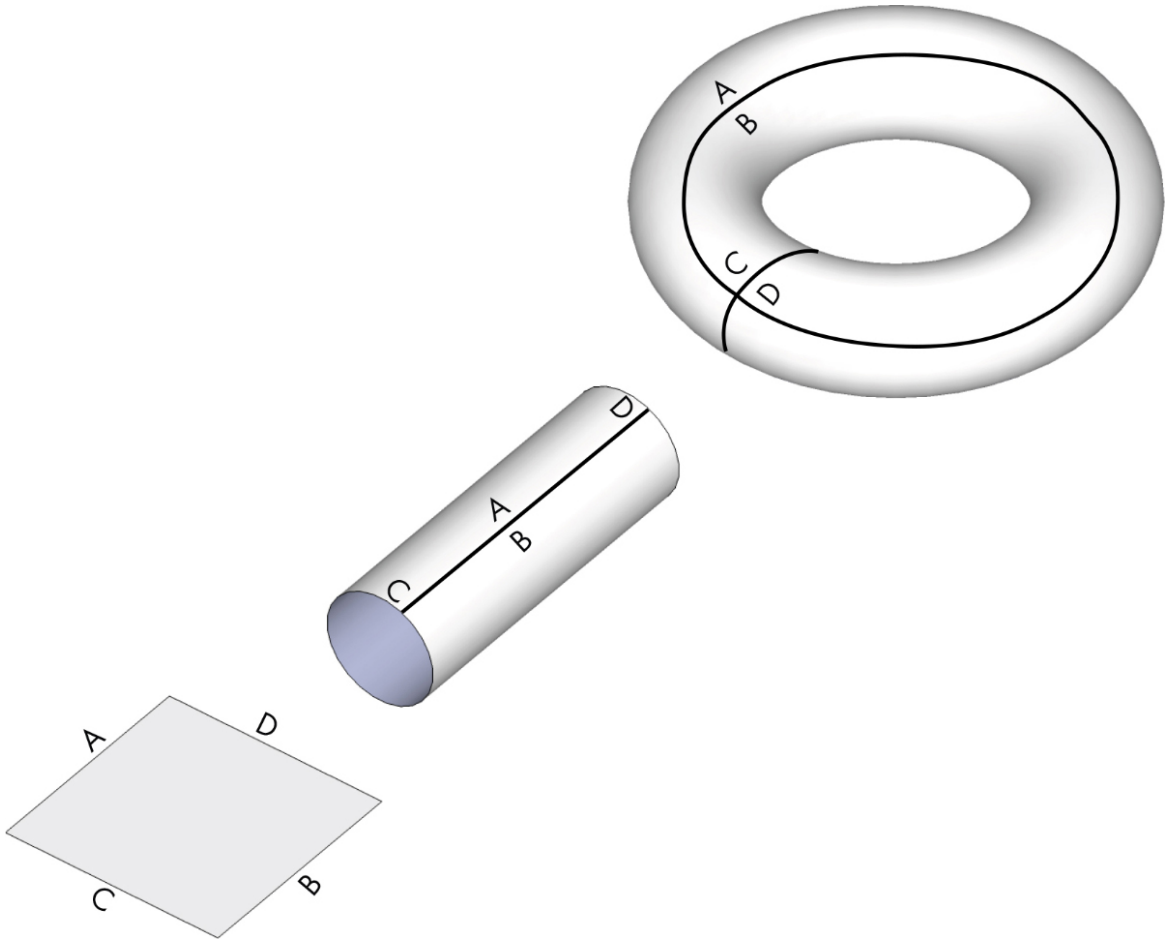


Figure 3-2: *A conceptual visualization of toroidal boundary conditions*

Toroidal boundary conditions are common in 2D simulations and games. For example, the game *Pac-Man* uses them. If you go off the top of the screen, you reappear on the bottom. If you go off the left side of the screen, you reappear on the right side. You'll follow the same logic for the Game of Life simulation: for the cell in the top-left corner of the grid, for example, the neighbor directly above it will be the cell in the bottom-left corner, and the neighbor directly to the left of it will be in the cell in the top-right corner.

Here's a description of the algorithm you'll use to apply the four rules and run the simulation:

1. Initialize the cells in the grid.
2. At each time step in the simulation, for each cell (*i*, *j*) in the grid, do the following:
    a. Update the value of cell (*i*, *j*) based on its neighbors, taking into account the boundary conditions.
    b. Update the display of grid values.

Requirements

You'll use `numpy` arrays and the `matplotlib` library to display the simulation output, and you'll use the `matplotlib animation` module to update the simulation.

The Code

We'll examine key aspects of the program piece by piece, including how to represent the simulation grid using `numpy` and `matplotlib`, how to set the initial conditions for the simulation, how to account for toroidal boundary conditions, and how to implement the Game of Life rules. We'll also look at the program's `main()` function, which sends command line arguments to the program and sets the simulation into motion. To see the full project code, skip ahead to "The Complete Code" on page 56. You can also download the code from *https://github.com/mkvenkit/pp2e/tree/main/conway*.

Representing the Grid

To represent whether a cell is alive (ON) or dead (OFF) on the grid, you'll use the values `255` and `0` for ON and OFF, respectively. You'll display the current state of the grid using the `imshow()` method in `matplotlib`, which represents a matrix of numbers as an image. To get a feel for how it works, let's try a simple example inside the Python interpreter. Enter the following:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> x = np.array([[0, 0, 255], [255, 255, 0], [0, 255, 0]])
>>> plt.imshow(x, interpolation='nearest')
>>> plt.show()
```

You define a 2D `numpy` array of shape (3, 3) ❶, meaning the array has three rows and three columns. Each element of the array is an integer value. You then use the `plt.imshow()` method to display this matrix of values as an image ❷, passing in the interpolation option as `'nearest'` to get sharp edges for the cells (otherwise, they'd be fuzzy). Figure 3-3 shows the output of this code.

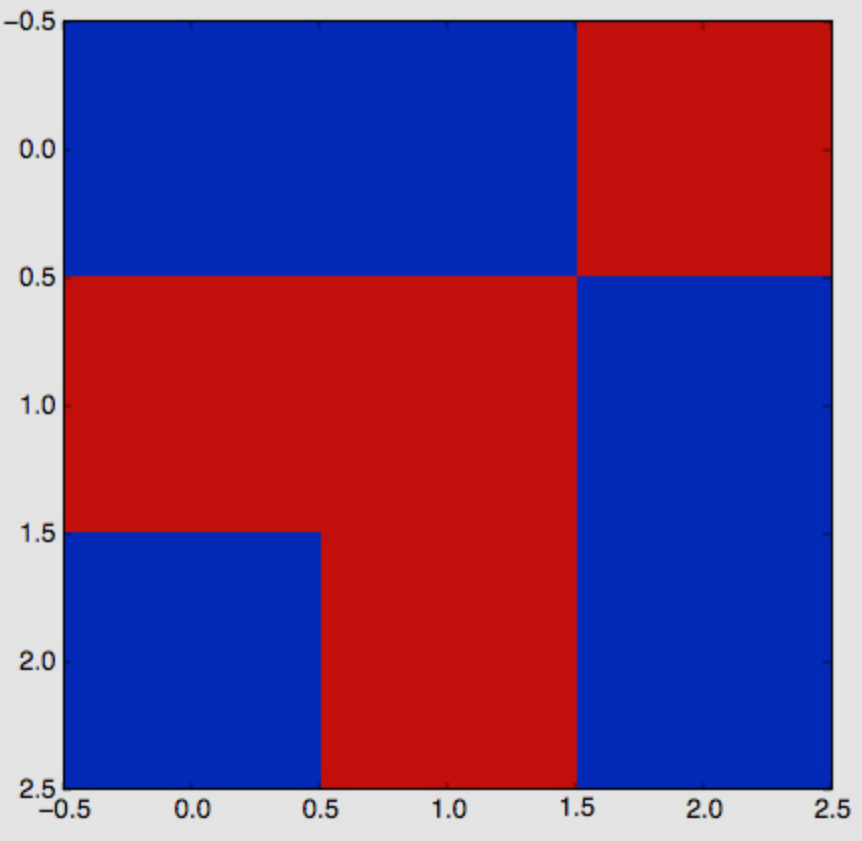


Figure 3-3: *Displaying a grid of values*

Notice that squares with a value of `0` (OFF) are given a darker color, while squares with a value of `255` (ON) are given a lighter color.

Setting the Initial Conditions

To begin the simulation, set an initial state for each cell in the 2D grid. You can use a random distribution of ON and OFF cells and see what kinds of patterns emerge, or you can add some specific patterns and see how they evolve. We'll look at both approaches.

For a random initial state, use the `choice()` method from the `random` module in `numpy`. Enter the following in the Python interpreter to see how it works:

```
>>> np.random.choice([0, 255], 4*4, p=[0.1, 0.9]).reshape(4, 4)
```

The output will be something like this:

```
array([[255, 255, 255, 255],
       [255, 255, 255, 255],
       [255, 255, 255, 255],
       [255, 255, 255, 0]])
```

`np.random.choice()` chooses a value from the given list `[0, 255]`, with the probability of the appearance of each value given in the parameter `p=[0.1, 0.9]`. Here you ask for 0 to appear with a probability of 0.1 (or 10 percent) and for 255 to appear with a probability of 90 percent. (The two values in `p` must add up to 1.) The `choice()` method creates a one-dimensional array, in this case with 16 values (specified with `4*4`). You use `reshape()` to make it a two-dimensional array with four rows and four columns.

To set up the initial condition to match a particular pattern instead of just filling in a random set of values, first use `np.zeros()` to initialize the grid with all zeros:

```
grid = np.zeros(N*N).reshape(N, N)
```

This creates an $N \times N$ array of zeros. Now define a function to add a pattern at a particular row and column in the grid:

```
def addGlider(i, j, grid):
    """adds a glider with top left cell at (i, j)"""
  ❶ glider = np.array([[0, 0, 255],
                       [255, 0, 255],
                       [0, 255, 255]])
  ❷ grid[i:i+3, j:j+3] = glider
```

You define the glider pattern (an observed pattern that moves steadily across the grid) using a `numpy` array of shape (3, 3) ❶. Then you use the `numpy` slice operation to copy the `glider` array into the simulation's two-dimensional grid ❷, with the pattern's top-left corner placed at the coordinates you specify as `i` and `j`.

Now you can add a glider pattern to the grid of zeros with a call to the `addGlider()` function you just defined:

```
addGlider(1, 1, grid)
```

You specify coordinates of (1, 1) to add the glider near the top-left corner of the grid, which is (0, 0). Note that for `grid[i, j]`, `i` starts at `0` and runs from top to bottom, and `j` starts at `0` and runs from left to right.

Enforcing the Boundary Conditions

Now we can think about how to implement the toroidal boundary conditions. First, let's see what happens at the right edge of a grid of size $N \times N$. The cell at the end of row i is accessed as `grid[i, N-1]`. Its neighbor to the right is `grid[i, N]`, but according to the toroidal boundary conditions, the value accessed as `grid[i, N]` should be replaced by `grid[i, 0]`. Here's one way to do that:

```
if j == N-1:
    right = grid[i, 0]
else:
    right = grid[i, j+1]
```

Of course, you'd need to apply similar boundary conditions to the left, top, and bottom sides of the grid, but doing so would require adding a lot more code because

each of the four edges of the grid would need to be tested. A much more compact way to implement the boundary conditions is with Python's modulus (`%`) operator, demonstrated here in the Python interpreter:

```
>>> N = 16
>>> i1 = 14
>>> i2 = 15
>>> (i1+1)%N
15
>>> (i2+1)%N
0
```

As you can see, the `%` operator gives the remainder for the integer division by `N`. In this example, `15%16` yields `15`, but `16%16` yields `0`. You can use the `%` operator to make the values wrap around at the right edge by rewriting the grid access code like this:

```
right = grid[i, (j+1)%N]
```

Now when a cell is on the edge of the grid (in other words, when `j = N-1`), asking for the cell to the right with this method will give you `(j+1)%N`, which sets `j` back to `0`, making the right side of the grid wrap to the left side. When you do the same for the bottom of the grid, it wraps around to the top:

```
bottom = grid[(i+1)%N, j]
```

Implementing the Rules

The rules of the Game of Life are based on the number of neighboring cells that are ON or OFF. To simplify the

application of these rules, you can just calculate the total number of neighboring cells in the ON state. Because ON corresponds to a value of 255, simply sum the values of all the neighbors and divide by 255 to get the number of ON cells. Here's the relevant code:

```
total = int((grid[i, (j-1)%N] + grid[i, (j+1)%N] +
             grid[(i-1)%N, j] + grid[(i+1)%N, j] +
             grid[(i-1)%N, (j-1)%N] + grid[(i-1)%N, (j+1)%N] +
             grid[(i+1)%N, (j-1)%N] + grid[(i+1)%N,
  (j+1)%N])/255)
```

For any given cell (*i*, *j*), you sum the value of its eight neighbors, using the `%` operator to account for toroidal boundary conditions. Dividing the result by 255 gives you the number of neighbors that are ON, which you store in the variable `total`. Now you can use `total` to apply the Game of Life rules:

```
# apply Conway's rules
if grid[i, j] == ON:
  ❶ if (total < 2) or (total > 3):
        newGrid[i, j] = OFF
else:
  ❷ if total == 3:
        newGrid[i, j] = ON
```

Any cell that is ON is turned OFF if it has fewer than two neighbors that are ON or if it has more than three neighbors that are ON ❶. The code in the `else` branch applies only to OFF cells: a cell is turned ON if exactly three neighbors are ON ❷. The changes are applied to the corresponding cells in `newGrid`, which starts as a copy of `grid`. Once every cell has been evaluated and updated, `newGrid` contains the data for displaying the next time step in the simulation. You can't

make changes directly to `grid`, or the states of the cells would keep changing as you try to evaluate them.

Sending Command Line Arguments to the Program

Now you can begin writing the `main()` function of the simulation, which starts by sending command line arguments to the program:

```
def main():
    # command line arguments are in sys.argv[1],
  sys.argv[2], ...
    # sys.argv[0] is the script name and can be ignored
    # parse arguments
  ❶ parser = argparse.ArgumentParser(description="Runs
  Conway's Game of Life
                                        simulation.")
    # add arguments
  ❷ parser.add_argument('--grid-size', dest='N',
  required=False)
  ❸ parser.add_argument('--interval', dest='interval',
  required=False)
  ❹ parser.add_argument('--glider', action='store_true',
  required=False)
    args = parser.parse_args()
```

You create an `argparse.ArgumentParser` object to add command line options to the code ❶, and then you add various options to it in the following lines. The option at ❷ specifies the simulation grid size *N*, and the option at ❸ sets the animation update interval in milliseconds. You also create an option to start the simulation with a glider pattern ❹. If this option isn't set, the simulation will start with random ON and OFF values.

Initializing the Simulation

Continuing through the `main()` function, you come to the following section, which initializes the simulation:

```
  # set grid size
❶ N = 100

  # set animation update interval
❷ updateInterval = 50
  if args.interval:
      updateInterval = int(args.interval)

  # declare grid
  grid = np.array([])
  # check if "glider" demo flag is specified
❸ if args.glider:
      grid = np.zeros(N*N).reshape(N, N)
      addGlider(1, 1, grid)
❹ else:
      # set N if specified and valid
      if args.N and int(args.N) > 8:
          N = int(args.N)
      # populate grid with random on/off - more off than on
      grid = randomGrid(N)
```

This portion of the code applies any parameters called at the command line, once the command line options have been parsed. First, a default grid size of 100×100 cells ❶ and an update interval of 50 milliseconds ❷ are set, in case these options aren't set at the command line. Then you set up the initial conditions, either a random pattern by default ❹ or a glider pattern ❸.

Finally, you set up the animation:

```
  # set up the animation
❶ fig, ax = plt.subplots()
  img = ax.imshow(grid, interpolation='nearest')
```

```
  ❷ ani = animation.FuncAnimation(fig, update, fargs=(img,
grid, N, ),
                                  interval=updateInterval,
                                  save_count=50)

    plt.show()
```

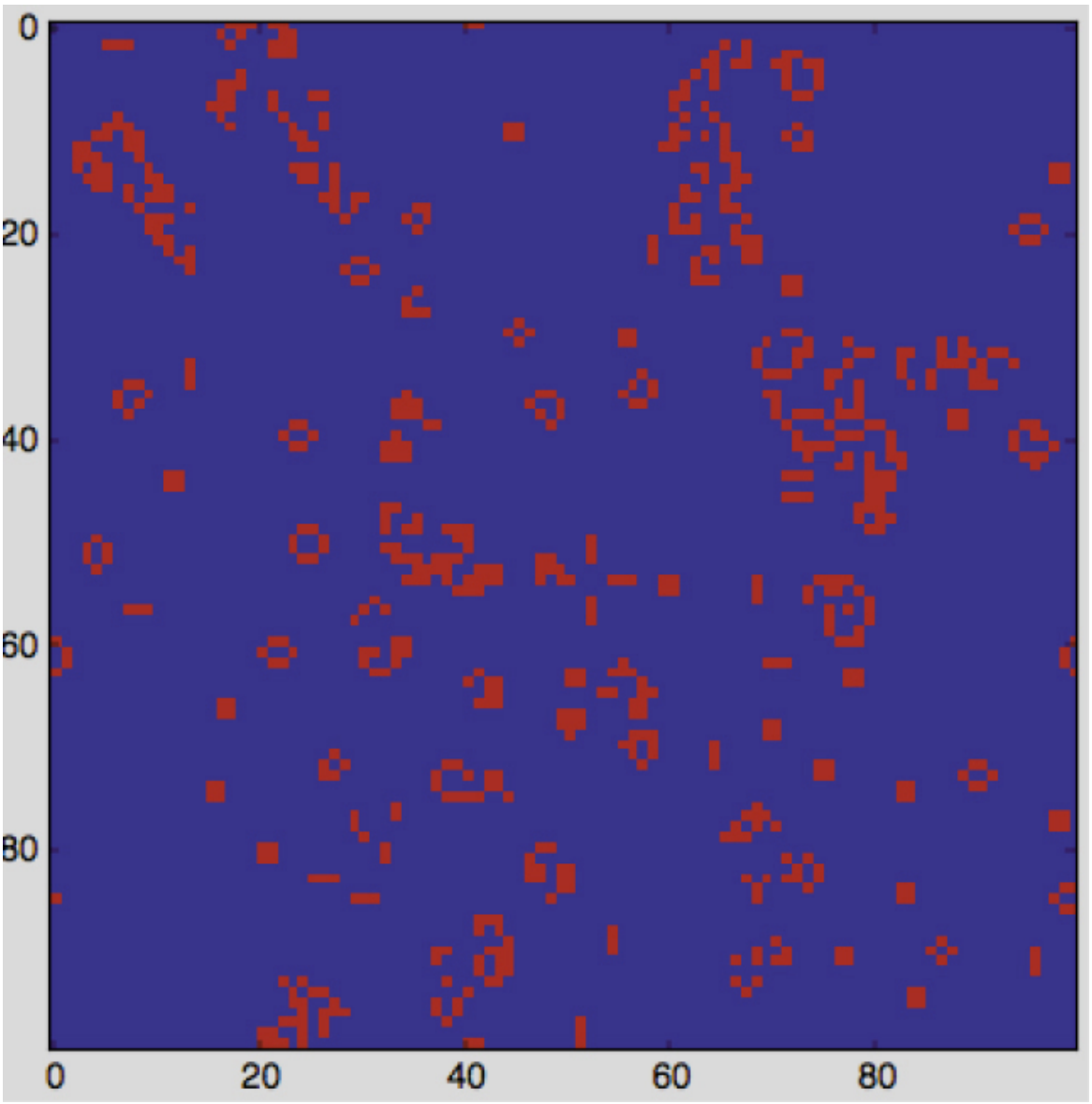
Still within the `main()` function, you configure the `matplotlib` plot and animation parameters ❶ . Then you set `animation.FuncAnimation()` to regularly call the function `update()`, defined earlier in the program, which updates the grid according to the rules of Conway's Game of Life using toroidal boundary conditions ❷.

Running the Game of Life Simulation

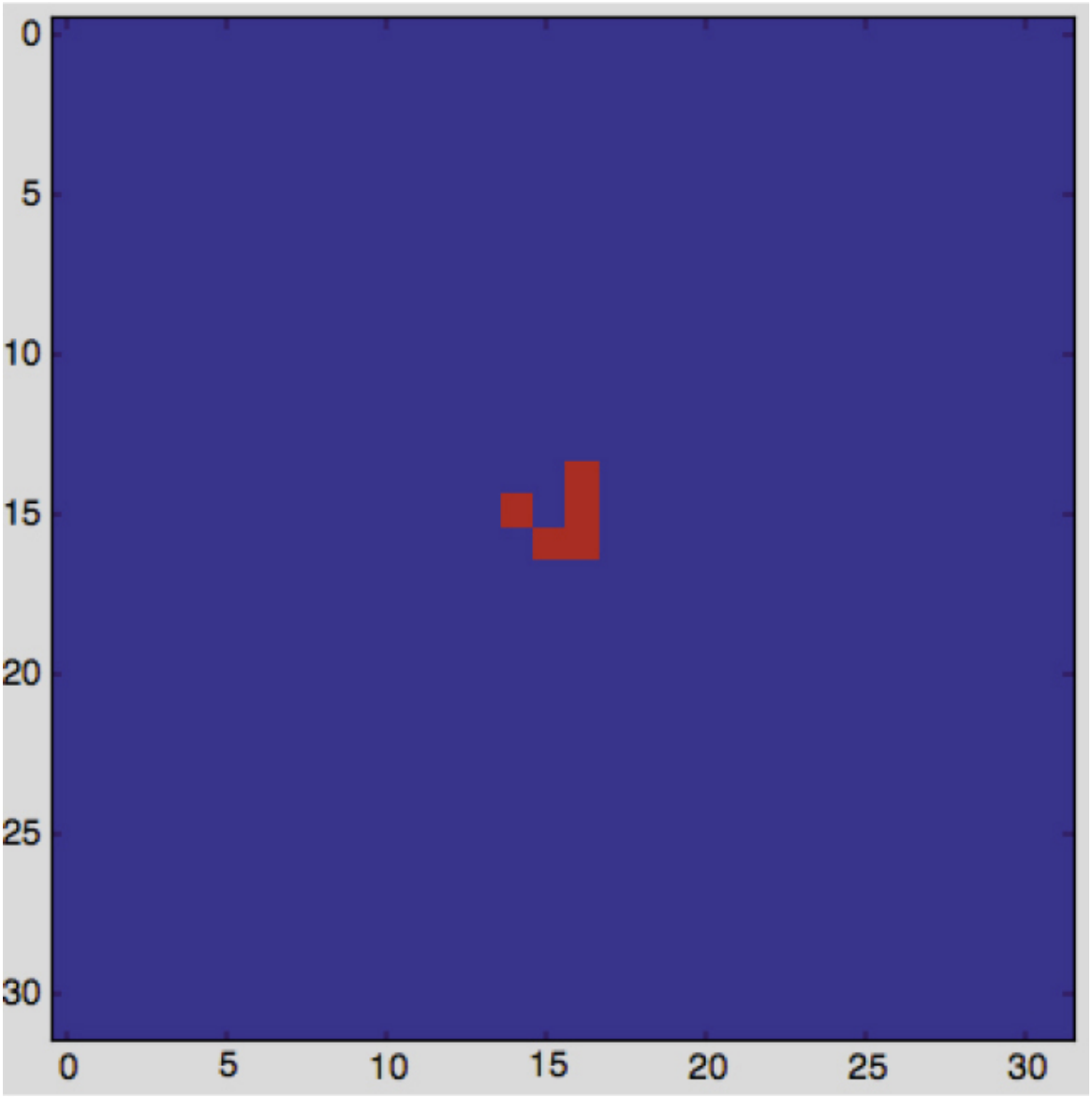
Now run the code:

```
$ python conway.py
```

This uses the default parameters for the simulation: a grid of 100×100 cells and an update interval of 50 milliseconds. As you watch the simulation, you'll see how it progresses to create and sustain various patterns over time, as in Figure 3-4 (a) and (b).



(a)



(b)

Figure 3-4: *The Game of Life in progress*

Figure 3-5 shows some of the patterns to look for in the simulation. Besides the glider, look for a three-cell blinker and static patterns such as a block or loaf shape.

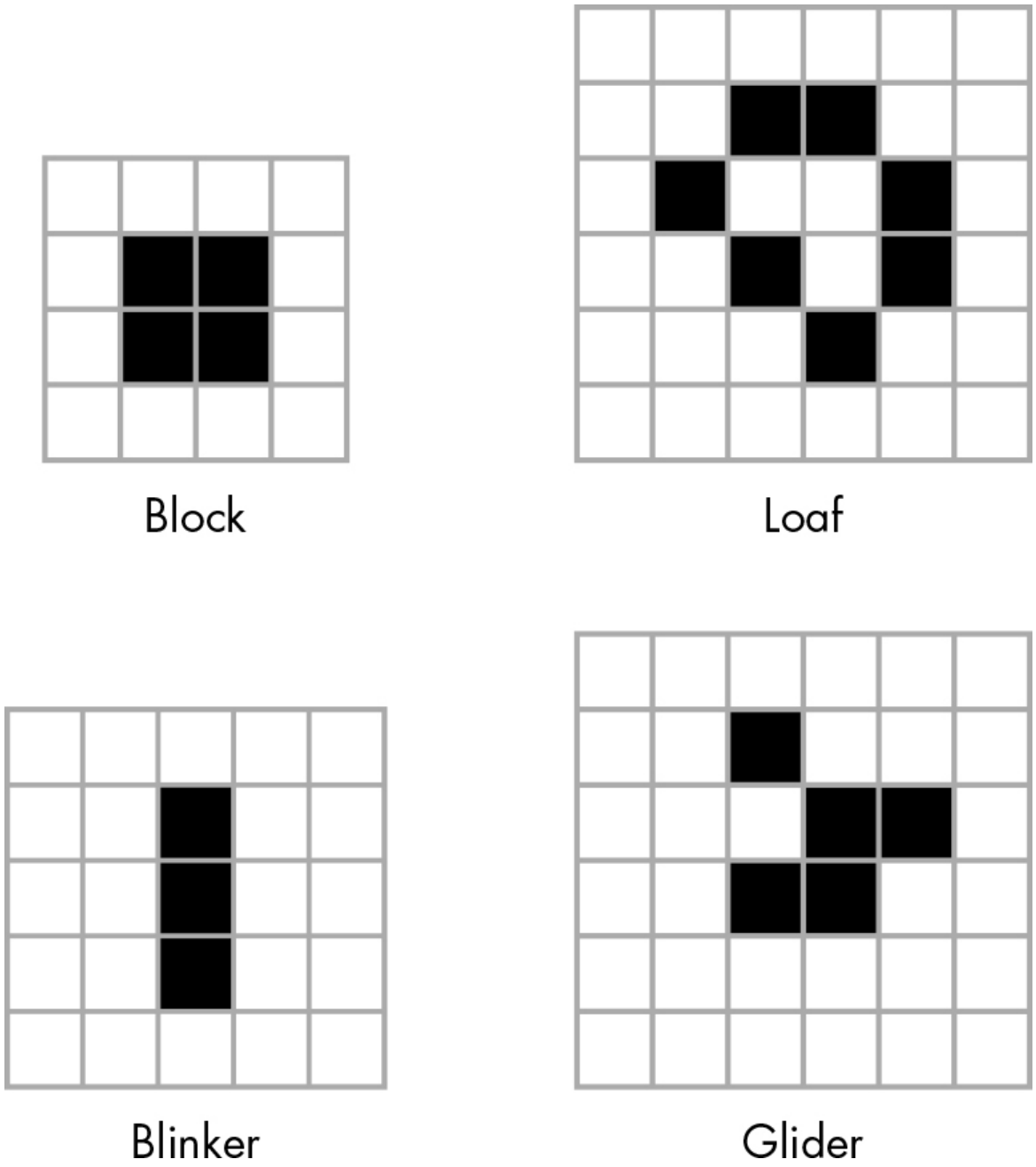


Figure 3-5: *Some patterns in the Game of Life*

Now change things up a bit by running the simulation with these parameters:

```
$ python conway.py --grid-size 32 --interval 500 --glider
```

This creates a simulation grid of 32×32, updates the animation every 500 milliseconds, and uses the initial glider pattern shown in the bottom right of Figure 3-5.

Summary

In this project, you explored Conway's Game of Life. You learned how to set up a basic computer simulation based on some rules and how to use `matplotlib` to visualize the state of the system as it evolves.

My implementation of Conway's Game of Life emphasizes simplicity over performance. You can speed up the computations in the Game of Life in many different ways, and a tremendous amount of research has been done on how to do this. You'll find a lot of this research through a quick internet search.

Experiments!

Here are some ways to experiment further with Conway's Game of Life:

1. Write an `addGosperGun()` method to add the pattern shown in Figure 3-6 to the grid. This pattern is called the *Gosper Glider Gun*. Run the simulation and observe what the gun does.

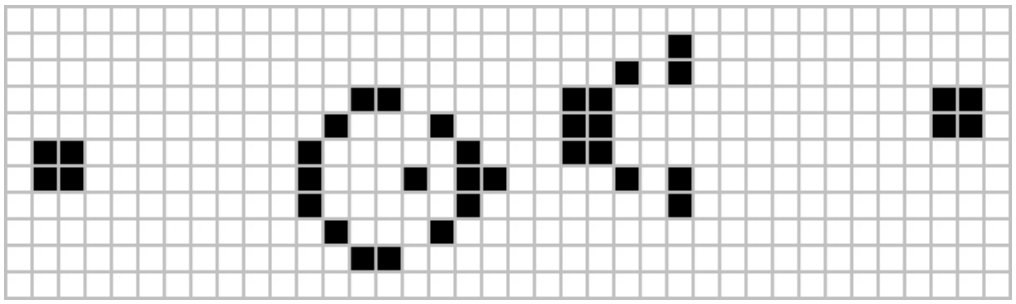


Figure 3-6: *The Gosper Glider Gun*

2. Write a `readPattern()` method that reads in an initial pattern from a text file and uses it to set the initial conditions for the simulation. You can use Python methods such as `open` and `file.read` to do this. Here's a suggested format for the input file:

```
8
0 0 0 255...
```

The first line of the file defines *N*, and the rest of the file is just *N*×*N* integers (0 or 255) separated by whitespace. This exploration will help you study how any given pattern evolves with the rules of the Game of Life. Add a `--pattern-file` command line option to use this file while running the program.

The Complete Code

Here's the complete code for the Game of Life project:

```
"""
conway.py
A simple Python/matplotlib implementation of Conway's Game of
  Life.
Author: Mahesh Venkitachalam
```

```
"""

import sys, argparse
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

def randomGrid(N):
    """returns a grid of NxN random values"""
    return np.random.choice([255, 0], N*N, p=[0.2,
  0.8]).reshape(N, N)

def addGlider(i, j, grid):
    """adds a glider with top left cell at (i, j)"""
    glider = np.array([[0,    0, 255],
                       [255,  0, 255],
                       [0,  255, 255]])
    grid[i:i+3, j:j+3] = glider

def update(frameNum, img, grid, N):
    # copy grid since we require 8 neighbors for calculation
    # and we go line by line
    newGrid = grid.copy()
    for i in range(N):
        for j in range(N):
            # compute 8-neighbor sum
            # using toroidal boundary conditions - x and y wrap
  around
            # so that the simulation takes place on a toroidal
  surface
            total = int((grid[i, (j-1)%N] + grid[i, (j+1)%N] +
                         grid[(i-1)%N, j] + grid[(i+1)%N, j] +
                         grid[(i-1)%N, (j-1)%N] + grid[(i-1)%N,
  (j+1)%N] +
                         grid[(i+1)%N, (j-1)%N] + grid[(i+1)%N,
  (j+1)%N])/255)
            # apply Conway's rules
            if grid[i, j]  == 255:
                if (total < 2) or (total > 3):
                    newGrid[i, j] = 0
            else:
                if total == 3:
```

```
                    newGrid[i, j] = 255
    # update data
    img.set_data(newGrid)
    grid[:] = newGrid[:]
    # need to return a tuple here, since this callback
    # function needs to return an iterable
    return img,

# main() function
def main():
    # command line args are in sys.argv[1], sys.argv[2]...
    # sys.argv[0] is the script name itself and can be ignored
    # parse arguments
    parser = argparse.ArgumentParser(description="Runs Conway's
  Game of Life
                                      simulation.")
    # add arguments
    parser.add_argument('--grid-size', dest='N',
  required=False)
    parser.add_argument('--interval', dest='interval',
  required=False)
    parser.add_argument('--glider', action='store_true',
  required=False)
    parser.add_argument('--gosper', action='store_true',
  required=False)
    args = parser.parse_args()

    # set grid size
    N = 100

    # set animation update interval
    updateInterval = 50
    if args.interval:
        updateInterval = int(args.interval)

    # declare grid
    grid = np.array([])
    # check if "glider" demo flag is specified
    if args.glider:
        grid = np.zeros(N*N).reshape(N, N)
        addGlider(1, 1, grid)
    elif args.gosper:
```

```
        grid = np.zeros(N*N).reshape(N, N)
        addGosperGliderGun(10, 10, grid)
    else:
        # set N if specified and valid
        if args.N and int(args.N) > 8:
            N = int(args.N)
        # populate grid with random on/off - more off than on
        grid = randomGrid(N)

    # set up animation
    fig, ax = plt.subplots()
    img = ax.imshow(grid, interpolation='nearest')
    ani = animation.FuncAnimation(fig, update, fargs=(img,
  grid, N, ),
                                  frames = 10,
                                  interval=updateInterval)

    plt.show()

# call main
if __name__ == '__main__':
    main()
```

4
MUSICAL OVERTONES WITH KARPLUS-STRONG



One of the main characteristics of any musical sound is its pitch, or *frequency*. This is the sound's number of vibrations per second in hertz (Hz). For example, the fourth string of an acoustic guitar produces a D note with a frequency of 146.83 Hz. You can approximate this sound by creating a sine wave with a frequency of 146.83 Hz on a computer, as shown in Figure 4-1.

Unfortunately, if you play this sine wave on your computer, it won't sound anything like a guitar. Nor will it sound like a piano, or any other real-world musical instrument for that matter. What makes a computer sound so different from a musical instrument when playing the same note?

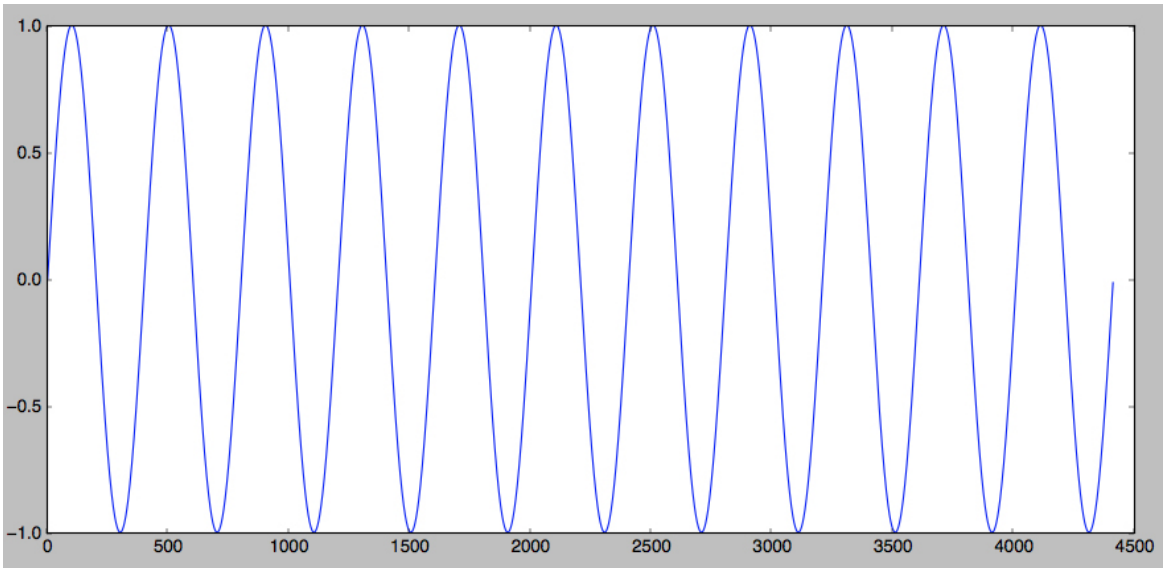


Figure 4-1: *A sine wave at 146.83 Hz*

When you pluck a string on a guitar, the instrument produces a mix of frequencies with varying intensity. The sound is most intense when the note is first struck, and the intensity dies off over time. In the case of plucking a guitar's D string, the dominant frequency you hear, called the *fundamental frequency*, is 146.83 Hz, but the sound also contains certain multiples of that frequency called *overtones*. In fact, the sound of any note on any instrument is composed of a fundamental frequency and overtones, and it's the combination of these different frequencies at different intensities that makes a guitar sound like a guitar, a piano sound like a piano, and so on. By contrast, a pure sine wave generated by a computer contains only a fundamental frequency, and no overtones.

You can see the evidence of overtones in *spectral plots*, like the one in Figure 4-2 representing the D string of a guitar. A spectral plot shows all the frequencies present in a sound at a particular moment in time, as well as the intensity of those frequencies. Notice that there are many different peaks in the spectral plot shown in the figure, telling us that there are many frequencies present in the

sound of the guitar's D string being plucked. Near the far left of the plot, the highest peak represents the fundamental frequency. The other peaks, representing the overtones, are less intense, but they still contribute to the quality of the sound.

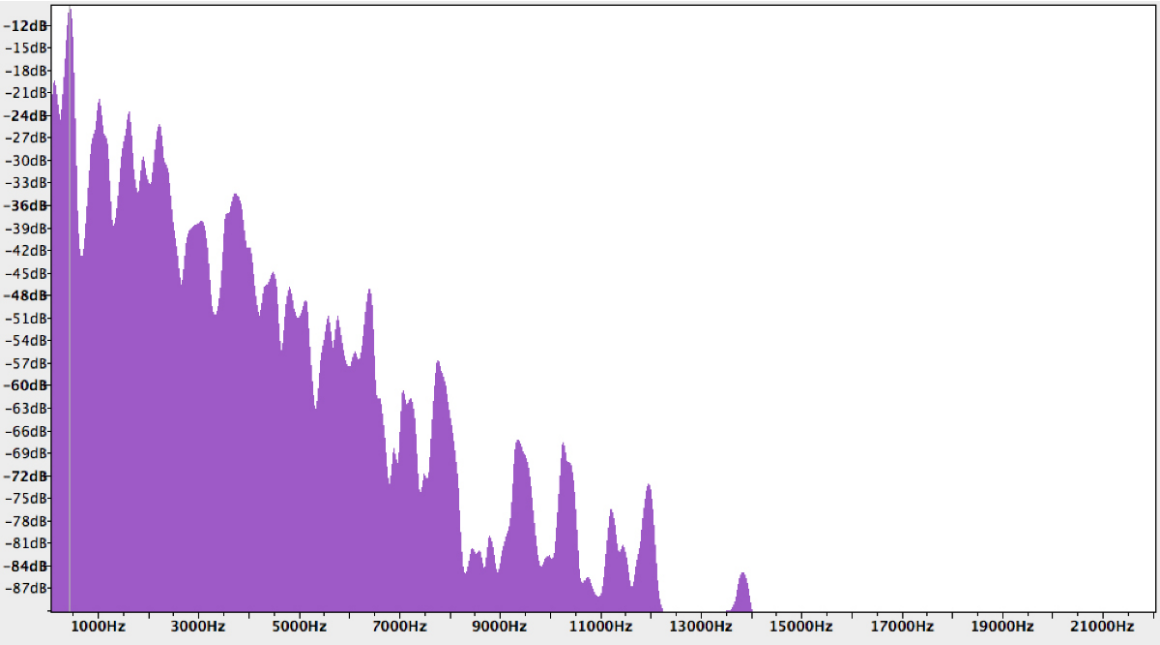


Figure 4-2: *A spectral plot of the note D3 played on a guitar*

As you can see, to simulate the sound of a plucked string instrument on the computer, you need to be able to generate both the fundamental frequency and the overtones. The trick is to use the Karplus-Strong algorithm. In this project, you'll generate five guitar-like notes of a musical scale (a series of related notes) using the Karplus-Strong algorithm. You'll visualize the algorithm used to generate these notes and save the sounds as WAV files. You'll also create a way to play them at random and learn how to do the following:

- Implement a ring buffer using the Python `deque` class.
- Use `numpy` arrays.

- Play WAV files using `pyaudio`.
- Plot a graph using `matplotlib`.
- Play the pentatonic musical scale.

How It Works

Imagine a string tied down at both ends, like a string on a guitar. When you pluck this string, it vibrates for a while, making a sound, and then settles back down to its resting position. At any given point in time while the string is vibrating, different parts of the string will be at different displacements from their resting position. These displacements can also be thought of as amplitudes of the sound wave produced by the vibrating string. The Karplus-Strong algorithm is a series of steps for generating and updating a series of these displacement or amplitude values to represent the motion of a wave along a plucked string. Play back those values as a WAV file and you get a pretty convincing simulation of a plucked string sound.

The Karplus-Strong algorithm stores displacement values in a *ring buffer* (also known as a *circular buffer*), a fixed-length buffer (just an array of values) that wraps around itself. In other words, when you reach the end of the buffer, the next element you access will be the first element in the buffer. (See “Implementing the Ring Buffer with deque” on page 66 for more about ring buffers.)

The length (N) of the ring buffer is related to the fundamental frequency of the note you want to simulate according to the equation $N = S/f$, where S is the sampling rate (more on this later) and f is the frequency. At the start of the simulation, the buffer is filled with random values in the range $[-0.5, 0.5]$, which you might think of as representing the random displacement of a string when it’s

first plucked. As the simulation progresses, the values are updated according to the steps of the Karplus-Strong algorithm, which we'll outline next.

In addition to the ring buffer, you'll use a *samples buffer* to store the intensity of the sound at any particular time. This buffer represents the final sound data, and it's built up based on the values in the ring buffer. The length of the samples buffer and the sampling rate determine the length of the sound clip.

The Simulation

During each time step in the simulation, a value from the ring buffer is stored in the samples buffer, and then the values in the ring buffer are updated in a kind of feedback scheme, as shown in Figure 4-3. Once the samples buffer is full, you write its contents to a WAV file so the simulated note can be played back as audio. For each time step of the simulation, you follow these steps, which together make up the Karplus-Strong algorithm:

1. Store the first value from the ring buffer in the samples buffer.
2. Calculate the average of the first two elements in the ring buffer.
3. Multiply this average value by an attenuation factor (in this case, 0.995).
4. Append this value to the end of the ring buffer.
5. Remove the first element of the ring buffer.

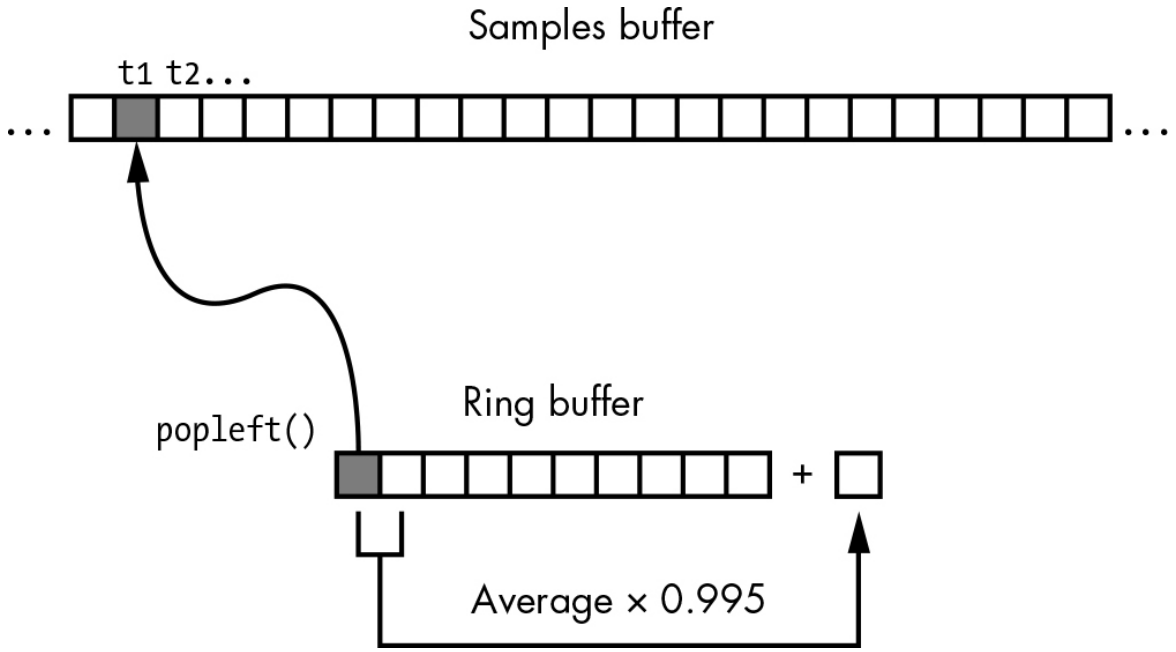


Figure 4-3: *A ring buffer and the Karplus-Strong algorithm*

This feedback scheme is designed to simulate a wave traveling through a vibrating string. The numbers in the ring buffer represent the energy of the wave at each point on the string. According to physics, the fundamental frequency of a vibrating string is inversely proportional to its length. Since we're interested in generating sounds of a certain frequency, we choose a ring buffer length inversely proportional to the desired frequency (this is the $N = S/f$ formula mentioned earlier). The averaging that happens in step 2 of the algorithm acts as a *low-pass filter* that cuts off higher frequencies and allows lower frequencies through, thereby eliminating higher harmonics (that is, larger multiples of the fundamental frequency) because you're mainly interested in the fundamental frequency. The attenuation factor in step 3 simulates the loss of energy as the wave travels back and forth along the string. This corresponds to the fading of the sound over time.

The samples buffer that you add to in step 1 of the simulation represents the amplitude of the generated sound over time. Storing the attenuated values at the end of the

ring buffer (step 4) and removing the first item from the ring buffer (step 5) ensures that a steady stream of gradually attenuating values will be passed to the samples buffer to build up the simulated sound.

Let's look at a simple example of the Karplus-Strong algorithm in action. Table 4-1 represents a ring buffer at two consecutive time steps. Each value in the ring buffer represents the amplitude of the sound, which is the same as the displacement of a point on a plucked string from its rest position. The buffer has five elements, and they are initially filled with some numbers.

Table 4-1: A Ring Buffer at Two Time Steps in the Karplus-Strong Algorithm

Time step 1	0.1	−0.2	0.3	0.6	−0.5
Time step 2	−0.2	0.3	0.6	−0.5	−0.04975

As you go from time step 1 to time step 2, you apply the Karplus-Strong algorithm as follows. The first value in the first row, 0.1, is removed, and all subsequent values from time step 1 are added in the same order to the second row, which represents time step 2. The last value in time step 2 is the attenuated average of the first and second values of time step 1, which is calculated as $0.995 \times ((0.1 + -0.2) \div 2) = -0.04975$.

The WAV File Format

The *Waveform Audio File Format (WAV)* is used to store audio data. This format is convenient for small audio

projects because it’s simple and doesn’t require you to worry about complicated compression techniques.

In its simplest form, a WAV file consists of a series of values, where each value represents the amplitude of the stored sound at a given point in time. Each value is allotted a fixed number of bits, called the *resolution*. You’ll use 16-bit resolution in this project. WAV files also have a set *sampling rate*, which is the number of times the audio is *sampled*, or read, every second. In this project, you use a sampling rate of 44,100 Hz, the rate used in audio CDs. In sum, when you generate a WAV file simulating the sound of a plucked string, it will contain 44,100 16-bit values for every second of audio.

For this project, you’ll be using Python’s `wave` module, which includes methods for working with WAV files. To get a feel for how it works, let’s generate a five-second audio clip of a 220 Hz sine wave using Python. First you represent a sine wave using this formula:

$$A = \sin(2\pi ft)$$

Here, A is the amplitude of the wave, f is the frequency, and t is the current time index. Now you rewrite this equation as follows:

$$A = \sin(2\pi fi/R)$$

In this equation, i is the index of the sample, and R is the sampling rate. Using these two equations, you can create a five-second WAV file for a 200 Hz sine wave as follows. (This code is available in *sine.py* in the chapter’s GitHub repository.)

```
import numpy as np
```

```
import wave, math

sRate = 44100
nSamples = sRate * 5
x = np.arange(nSamples)/float(sRate)
vals = np.sin(2.0*math.pi*220.0*x)
data = np.array(vals*32767, 'int16').tostring()
file = wave.open('sine220.wav', 'wb')
file.setparams((1, 2, sRate, nSamples, 'NONE', 'uncompressed'))
file.writeframes(data)
file.close()
```

You create a `numpy` array of numbers from `0` to `nSamples - 1` and divide those numbers by the sample rate to get the time value, in seconds, when each sample of the audio clip is taken ❶. This array represents the *i*/*R* portion of the sine wave equation discussed earlier. Next, you use the array to create a second `numpy` array, this one containing sine wave amplitude values, again following the sine wave equation ❷. The `numpy` array is a fast and convenient way to apply functions such as the `sin()` function to many values at once.

The computed sine wave values in the range [−1, 1] are scaled to 16-bit values and converted to a string so they can be written to a WAV file ❸. Then you set the parameters for the WAV file; in this case, it's a single-channel (mono), 2-byte (16-bit), uncompressed format ❹. Finally, you write the data to the file ❺ . Figure 4-4 shows the generated *sine220.wav* file in Audacity, a free audio editor. As expected, you see a sine wave of frequency 220 Hz, and when you play the file, you hear a 220 Hz tone for five seconds. (Note that you need to use the Zoom tool in Audacity to see the sine wave as shown in Figure 4-4.)

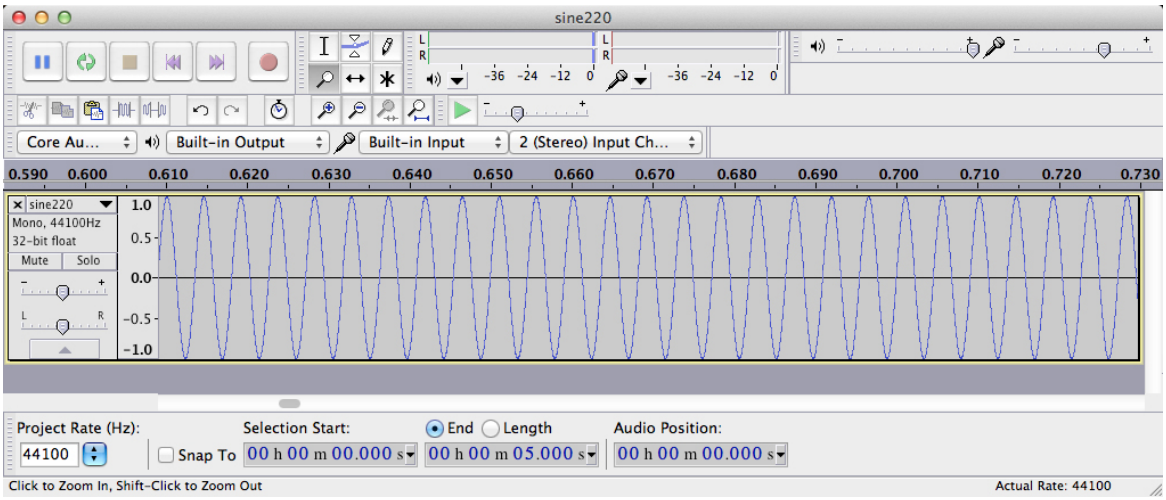


Figure 4-4: *A sine wave at 220 Hz, zoomed in*

In your project, once you've filled the samples buffer with audio data, you'll write it to a WAV file using the same pattern illustrated in Figure 4-4.

The Minor Pentatonic Scale

A *musical scale* is a series of notes in increasing or decreasing pitch (frequency). Often, all notes in a piece of music are chosen from a particular scale. A *musical interval* is the difference between two pitches. A *semitone* is a basic building block of a scale and is the smallest musical interval in Western music. A *tone* is twice the length of a semitone. The *major scale*, one of the most common musical scales, is defined by the interval pattern *tone-tone-semitone-tone-tone-tone-semitone*.

We'll briefly go into the pentatonic scale here, since you'll be generating musical notes in that scale. This section will explain the source of the frequency numbers used in the final program to generate notes with the Karplus-Strong algorithm. The *pentatonic scale* is a five-note musical scale. A variant of this scale is the *minor pentatonic scale*, which is defined by the interval pattern *(tone+semitone)-tone-tone-*

(tone+semitone)-tone. Thus, the C minor pentatonic scale consists of the notes C, E-flat, F, G, and B-flat. Table 4-2 lists the frequencies of the five notes of a C minor pentatonic scale that you'll generate using the Karplus-Strong algorithm. (Here, C4 designates C in the fourth octave of a piano, or *middle C*, by convention.)

Table 4-2: Notes in a Minor Pentatonic Scale

Note	**Frequency (Hz)**
C4	261.6
E-flat	311.1
F	349.2
G	392.0
B-flat	466.2

One aspect of this project will be stringing together random sequences of notes to create melodies. One of the reasons we're focusing on a minor pentatonic scale is that the notes of this scale sound pleasing no matter what order they're played in. Thus, the scale is particularly conducive to generating random melodies in a way that other scales, such as a major scale, are not.

Requirements

In this project, you'll use the Python `wave` module to create audio files in the WAV format. To implement the Karplus-Strong algorithm, you'll use the `deque` class from the Python `collections` module as a ring buffer and a `numpy` array as a samples buffer. You'll also use `matplotlib` to visualize the simulated guitar string, and you'll play back the WAV files with the `pyaudio` module.

The Code

Now let's develop the various pieces of code required to implement the Karplus-Strong algorithm and then put them together for the complete program. To see the full project code, skip ahead to "The Complete Code" on page 74. You can also download the code from the book's GitHub repository at *https://github.com/mkvenkit/pp2e/tree/main/karplus*.

Implementing the Ring Buffer with deque

Recall from earlier that the Karplus-Strong algorithm uses a ring buffer to generate a musical note. You'll implement the ring buffer using a `deque` container (pronounced "deck"), which is part of Python's `collections` module of specialized container data types. You can insert and remove elements from the beginning (head) or end (tail) of a `deque` (see Figure 4-5). This insertion and removal process is a `O(1)`, or a "constant time" operation, which means it takes the same amount of time regardless of how big the `deque` container gets.

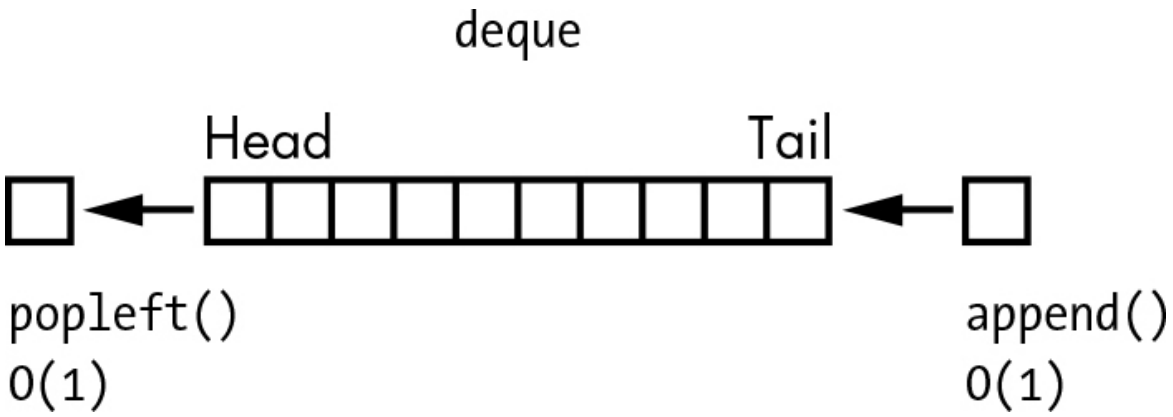


Figure 4-5: *A ring buffer implemented using* `deque`

The following code shows an example of how you would use `deque` in Python:

```
>>> from collections import deque
>>> d = deque(range(10), maxlen=10)
>>> print(d)
deque([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
>>> d.append(10)
>>> print(d)
deque([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], maxlen=10)
```

You create the `deque` container by passing in a list created with the `range()` function ❶. You also specify the maximum length `maxlen` of the `deque` as `10`. Next, you append the element `10` to the end of the `deque` container ❷. When you then print the `deque`, you can see that `10` has been appended to the end of the `deque`, while the first element, `0`, has automatically been removed to maintain the `deque` container's maximum length of 10 elements. This scheme will allow you to simultaneously implement steps 4 and 5 of the Karplus-Strong algorithm—adding a new value at the end of the ring buffer while removing the first value.

Implementing the Karplus-Strong Algorithm

You'll now implement the Karplus-Strong algorithm in the `generateNote()` function, using a `deque` container to implement the ring buffer and a `numpy` array to implement the samples buffer. In the same function, you'll also visualize the algorithm using `matplotlib`. The plot will show how the amplitudes of the plucked string change over time, in effect showing how the string moves as it vibrates.

You begin with some setup:

```
# initialize plotting
fig, ax = plt.subplots(1)
line, = ax.plot([], [])

def generateNote(freq):
    """generate note using Karplus-Strong algorithm"""
    nSamples = 44100
    sampleRate = 44100
  ❸ N = int(sampleRate/freq)
  ❹ if gShowPlot:
        # set axis
        ax.set_xlim([0, N])
        ax.set_ylim([-1.0, 1.0])
        line.set_xdata(np.arange(0, N))

    # initialize ring buffer
  ❺ buf = deque([random.random() - 0.5 for i in range(N)],
  maxlen=N)
    # init samples buffer
  ❻ samples = np.array([0]*nSamples, 'float32')
```

First you create a `matplotlib` figure ❶ and a line plot ❷, which you'll fill with data. Then you begin the `generateNote()` function definition, which takes the frequency of the note to be generated as a parameter. You set the number of samples in the sound clip and the sample rate to both be 44,100, which means the resulting clip will be one second long. Then you divide the sample rate by the desired

frequency to set the length `N` of the Karplus-Strong ring buffer ❸. If the `gShowPlot` flag is set ❹, you initialize the x and y range of the plot and initialize the x values to `[0, ... N-1]` using the `arange()` function.

You next initialize the ring buffer as a `deque` container with random numbers in the range [−0.5, 0.5], setting the maximum length of the `deque` to `N` ❺. You also initialize the samples buffer as a `numpy` array of floats ❻. You set the length of the array to be the number of samples the sound clip will contain.

Next comes the heart of the `generateNote()` function, where you implement the steps of the Karplus-Strong algorithm and create the visualization:

```
    for i in range(nSamples):
      ❶ samples[i] = buf[0]
      ❷ avg = 0.995*0.5*(buf[0] + buf[1])
      ❸ buf.append(avg)
        # plot of flag set
      ❹ if gShowPlot:
            if i % 1000 == 0:
                line.set_ydata(buf)
                fig.canvas.draw()
                fig.canvas.flush_events()

    # samples to 16-bit to string
    # max value is 32767 for 16-bit
  ❺ samples = np.array(samples * 32767, 'int16')
  ❻ return samples.tobytes()
```

Here you iterate over each element in the samples buffer and carry out the steps of the Karplus-Strong algorithm. With each iteration, you copy the first element in the ring buffer to the samples buffer ❶. Then you perform the low-pass filtering and attenuation by averaging the first two elements in the ring buffer and multiplying the result by

0.995 ❷. This attenuated value is appended to the end of the ring buffer ❸. Since the `deque` representing the ring buffer has a maximum length, the `append()` operation also removes the first element from the buffer.

The `samples` array is converted into a 16-bit format by multiplying each value by 32,767 ❺ (a 16-bit signed integer can take values only from −32,768 to 32,767, and 0.5 × 65,534 = 32,767). Then the array is converted to a byte representation for the `wave` module, which you'll use to save this data to a file ❻.

As the algorithm is running, you visualize how the ring buffer evolves ❹. For every thousand samples, you update the `matplotlib` graph with the values in the ring buffer, and this shows how the data changes with time.

Writing a WAV File

Once you have the audio data, you can write it to a WAV file using the Python `wave` module. Define a `writeWAVE()` function to carry this out:

```
def writeWAVE(fname, data):
    # open file
  ❶ file = wave.open(fname, 'wb')
    # WAV file parameters
    nChannels = 1
    sampleWidth = 2
    frameRate = 44100
    nFrames = 44100
    # set parameters
  ❷ file.setparams((nChannels, sampleWidth, frameRate,
  nFrames,
                    'NONE', 'noncompressed'))
  ❸ file.writeframes(data)
    file.close()
```

You create a WAV file ❶ and set its parameters using a single-channel, 16-bit, noncompressed format ❷. Then you write the data to the file ❸.

Playing WAV Files with pyaudio

Now you'll use the Python `pyaudio` module to play the WAV files generated by the algorithm. `pyaudio` is a high-performance, low-level library that gives you access to sound devices on a computer. For convenience, you encapsulate the code in a `NotePlayer` class, as shown here:

```
class NotePlayer:
    # constructor
    def __init__(self):
        # init pyaudio
      ❶ self.pa = pyaudio.PyAudio()
        # open stream
      ❷ self.stream = self.pa.open(
                format=pyaudio.paInt16,
                channels=1,
                rate=44100,
                output=True)
        # dictionary of notes
      ❸ self.notes = []
```

In the `NotePlayer` class's constructor, you first create the `PyAudio` object that you'll use to play the WAV file ❶. Then you open a 16-bit single-channel `PyAudio` output stream ❷. You also create an empty list that you'll later fill with filenames of the five pentatonic note WAV files ❸.

In Python, when all references to an object have been deleted, the object is destroyed by a process called *garbage collection*. At that time, the object's `__del__()` method, also

known as a *destructor*, is called, if one is defined. Here's the `NotePlayer` class's destructor:

```
def __del__(self):
    # destructor
    self.stream.stop_stream()
    self.stream.close()
    self.pa.terminate()
```

This method ensures that the `PyAudio` stream is cleaned up when the `NotePlayer` object is destroyed. Failing to provide a `__del__()` method for a class can cause problems when objects are repeatedly created and destroyed, since some system-wide resources (like `pyaudio` in this case) may not be cleaned up properly.

The remaining methods of the `NotePlayer` class are devoted to building up a list of possible notes and playing them. First, here's the `add()` method, which is used to add a WAV filename to the class:

```
def add(self, fileName):
    self.notes.append(fileName)
```

The method takes a filename corresponding to one of the generated WAV files as a parameter and adds it to the `notes` list you initialized in the class's constructor. The class will draw on this list when it wants to play a WAV file.

Next, let's look at the `play()` method used to play a note:

```
def play(self, fileName):
    try:
        print("playing " + fileName)
        # open WAV file
      ❶ wf = wave.open(fileName, 'rb')
        # read a chunk
```

```
    ❷ data = wf.readframes(CHUNK)
      # read rest
      while data != b'':
        ❸ self.stream.write(data)
        ❹ data = wf.readframes(CHUNK)
      # clean up
    ❺ wf.close()
  except BaseException as err:
    ❻ print(f"Exception! {err=}, {type(err)=}.\nExiting.")
      exit(0)
```

Here you open the desired WAV file using Python's `wave` module ❶. Then you read `CHUNK` frames (defined globally as 1,024 in this case) from the file into `data` ❷. Next, within a `while` loop, you write the contents of `data` to the `PyAudio` output stream ❸ and read the next chunk of data from the WAV file ❹. Writing to the output stream has the effect of playing the audio through the default audio device of your computer, which is typically a speaker. You read the data in chunks to maintain the sample rate at the output side. If the chunks are too large and you take too much time in between reading and writing, the audio won't sound right.

The `while` loop continues for as long as there's more data to read—that is, until `data` is empty. At that point, you close the WAV file object ❺. You handle any exceptions that may happen during the playback process (for example, the user pressing `CTRL`-C) by printing the error ❻ and exiting the program.

Finally, the `NotePlayer` class's `playRandom()` method picks a random note from the five notes you've generated and plays it:

```
def playRandom(self):
    """play a random note"""
    index = random.randint(0, len(self.notes)-1)
    note = self.notes[index]
```

```
    self.play(note)
```

The method selects a random WAV filename from the `notes` list and passes that filename to the `play()` method to be played.

Creating Notes and Parsing Arguments

Now let's look at the program's `main()` function, which creates the notes and handles various command line options to play the notes:

```
def main():
--snip--
    parser = argparse.ArgumentParser(description="Generating
  sounds with
                                                 Karplus-Strong
  Algorithm")
    # add arguments
    parser.add_argument('--display', action='store_true',
  required=False)
    parser.add_argument('--play', action='store_true',
  required=False)
    args = parser.parse_args()

    # show plot if flag set
  ❶ if args.display:
        gShowPlot = True
        plt.show(block=False)

    # create note player
  ❷ nplayer = NotePlayer()

    print('creating notes...')
    for name, freq in list(pmNotes.items()):
        fileName = name + '.wav'
      ❸ if not os.path.exists(fileName) or args.display:
            data = generateNote(freq)
            print('creating ' + fileName + '...')
```

```
            writeWAVE(fileName, data)
        else:
            print('fileName already created. skipping...')

        # add note to player
      ❹ nplayer.add(name + '.wav')

        # play note if display flag set
        if args.display:
          ❺ nplayer.play(name + '.wav')
            time.sleep(0.5)

    # play a random tune
    if args.play:
        while True:
            try:
              ❻ nplayer.playRandom()
                # rest - 1 to 8 beats
              ❼ rest = np.random.choice([1, 2, 4, 8], 1,
                                        p=[0.15, 0.7, 0.1,
0.05])
                time.sleep(0.25*rest[0])
            except KeyboardInterrupt:
                exit()
```

First you set up some command line options for the program using `argparse`, as discussed in earlier projects. The `--display` option will play each of the five notes in turn, while visualizing each note's waveform using `matplotlib`. The `--play` option generates a random melody using the five notes.

If the `--display` command line option was used ❶, you set up a `matplotlib` plot to show how the waveform evolves during the Karplus-Strong algorithm. The `plt.show(block=False)` call ensures that the `matplotlib` display method doesn't block. This way, when you call this function, it will return immediately and go on to the next statement. This is the behavior you need, since you're manually updating the plot every frame.

You next create an instance of the `NotePlayer` class ❷. Then you generate WAV files of the five notes in the C minor pentatonic scale. The frequencies for the notes are defined in the global dictionary `pmNotes`, which looks like this:

```
pmNotes = {'C4': 262, 'Eb': 311, 'F': 349, 'G': 391, 'Bb': 466}
```

To generate the notes, you iterate through the dictionary, first constructing a filename for the note using the dictionary key plus the *.wav* extension—for example, *C4.wav*. You use the `os.path.exists()` method to see whether the WAV file for a particular note has been created ❸. If so, you skip the computation for that note. (This is a handy optimization if you're running this program several times.) Otherwise, you generate the note using the `generateNote()` and `writeWAVE()` functions you defined earlier. Once the note is computed and the WAV file created, you add the note's filename to the `NotePlayer` object's list of notes ❹, and then you play the note if the `--display` command line option is used ❺.

If the `--play` option is used, the `playRandom()` method in `NotePlayer` repeatedly plays a note at random from the five notes ❻. For a note sequence to sound even remotely musical, you need to add rests between the notes played, so you use the `random.choice()` method from `numpy` to choose a random rest interval ❼. This method also lets you choose the probability of the rest interval, which you set so that a two-beat rest is the most probable and an eight-beat rest the least probable. Try changing these values to create your own style of random music!

Running the Plucked String Simulation

To run the code for this project, enter this in a command shell:

```
$ python ks.py --display
```

As you can see in Figure 4-6, the `matplotlib` plot shows how the Karplus-Strong algorithm converts the initial random displacements to create waves of the desired frequency.

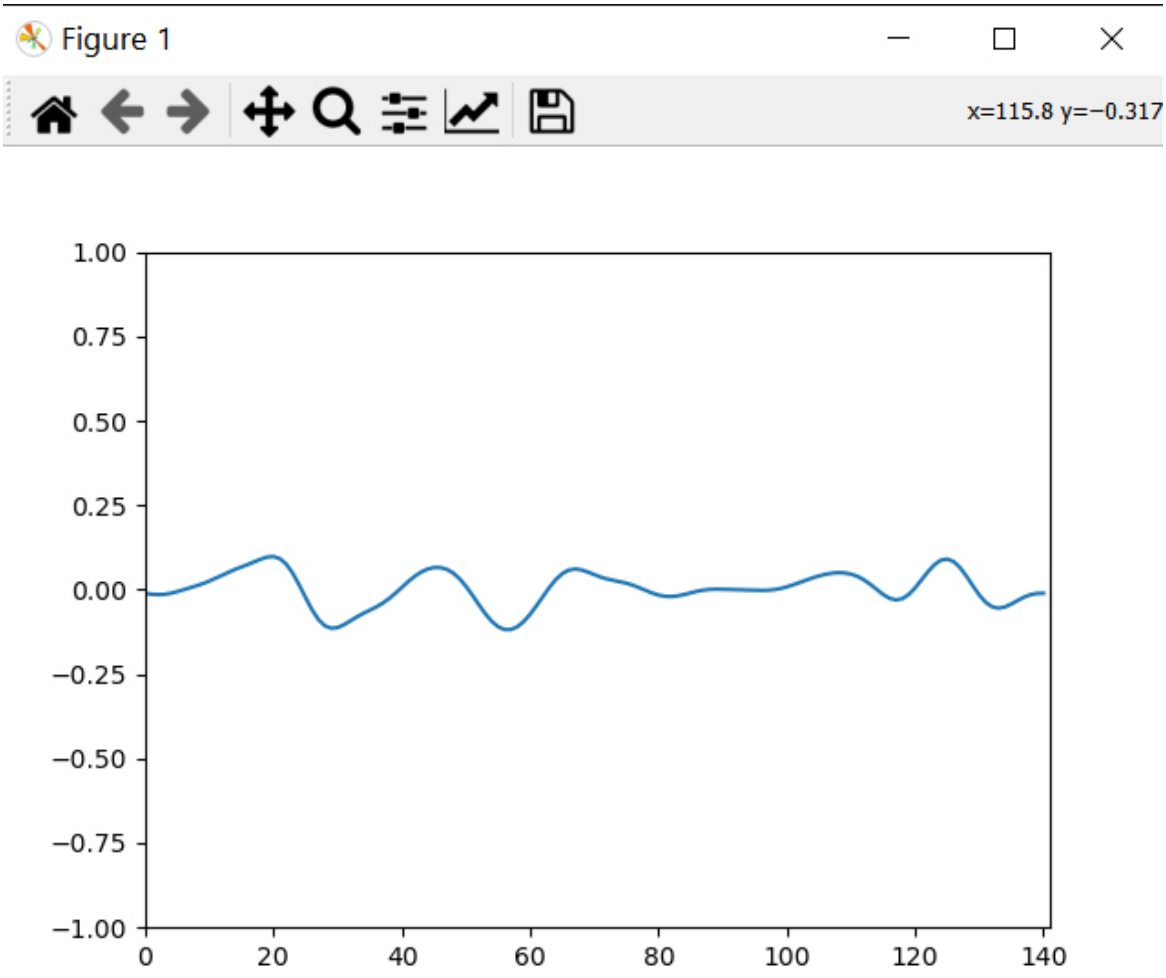


Figure 4-6: *A sample run of the plucked string simulation*

Now try playing a random sequence of notes using this command:

```
$ python ks.py --play
```

This should play a random note sequence using the generated WAV files of the pentatonic musical scale.

Summary

In this project, you used the Karplus-Strong algorithm to simulate the sound of plucked strings and played notes from generated WAV files. You learned how to implement the Karplus-Strong algorithm using a `deque` container as a ring buffer. You also learned about the WAV file format and how to play WAV files using `pyaudio`, as well as how to use `matplotlib` to visualize a vibrating string. You even learned about the pentatonic musical scale!

Experiments!

Here are some ideas for experiments:

1. I've stated that the Karplus-Strong algorithm creates realistic plucked string sounds by generating overtones as well as the fundamental frequency of the note. But how do you know it's working? By creating spectral plots of your WAV files, like the one in Figure 4-2. You can use the free program Audacity to do this. Open one of the WAV files in Audacity, and select **Analyze▸Plot Spectrum**. You should see that the sound contains many frequencies.
2. Use the techniques you learned in this chapter to create a method that replicates the sound of two strings of different frequencies vibrating together.

Remember, the Karplus-Strong algorithm produces a ring buffer full of sound amplitude values. You can combine two sounds by adding their amplitudes together.

3. Replicate the sound of two strings vibrating together, as described in the previous experiment, but add a time delay between the first and second string plucks.
4. Write a method to read music from a text file and generate musical notes. Then play the music using these notes. You can use a format where the note names are followed by integer rest time intervals, like this: C4 1 F4 2 G4 1 . . .

The Complete Code

Here's the complete code for this project:

```
"""
ks.py

Uses the Karplus-Strong algorithm to generate musical notes
in a pentatonic scale.

Author: Mahesh Venkitachalam
"""

import sys, os
import time, random
import wave, argparse
import numpy as np
from collections import deque
import matplotlib
# to fix graph display issues on macOS
matplotlib.use('TkAgg')
from matplotlib import pyplot as plt
import pyaudio
```

```
# show plot of algorithm in action?
gShowPlot = False

# notes of a pentatonic minor scale
# piano C4-E(b)-F-G-B(b)-C5
pmNotes = {'C4': 262, 'Eb': 311, 'F': 349, 'G':391, 'Bb':466}

CHUNK = 1024

# initialize plotting
fig, ax = plt.subplots(1)
line, = ax.plot([], [])

# write out WAV file
def writeWAVE(fname, data):
    """write data to WAV file"""
    # open file
    file = wave.open(fname, 'wb')
    # WAV file parameters
    nChannels = 1
    sampleWidth = 2
    frameRate = 44100
    nFrames = 44100
    # set parameters
    file.setparams((nChannels, sampleWidth, frameRate, nFrames,
                    'NONE', 'noncompressed'))
    file.writeframes(data)
    file.close()

def generateNote(freq):
    """generate note using Karplus-Strong algorithm"""
    nSamples = 44100
    sampleRate = 44100
    N = int(sampleRate/freq)

    if gShowPlot:
        # set axis
        ax.set_xlim([0, N])
        ax.set_ylim([-1.0, 1.0])
        line.set_xdata(np.arange(0, N))

    # initialize ring buffer
```

```
    buf = deque([random.random() - 0.5 for i in range(N)],
  maxlen=N)
    # init sample buffer
    samples = np.array([0]*nSamples, 'float32')
    for i in range(nSamples):
        samples[i] = buf[0]
        avg = 0.995*0.5*(buf[0] + buf[1])
        buf.append(avg)
        # plot of flag set
        if gShowPlot:
            if i % 1000 == 0:
                line.set_ydata(buf)
                fig.canvas.draw()
                fig.canvas.flush_events()

    # samples to 16-bit to string
    # max value is 32767 for 16-bit
    samples = np.array(samples * 32767, 'int16')
    return samples.tobytes()

# play a WAV file
class NotePlayer:
    # constructor
    def __init__(self):
        # init pyaudio
        self.pa = pyaudio.PyAudio()
        # open stream
        self.stream = self.pa.open(
                format=pyaudio.paInt16,
                channels=1,
                rate=44100,
                output=True)
        # dictionary of notes
        self.notes = []
    def __del__(self):
        # destructor
        self.stream.stop_stream()
        self.stream.close()
        self.pa.terminate()

    # add a note
```

```
    def add(self, fileName):
        self.notes.append(fileName)
    # play a note
    def play(self, fileName):
        try:
            print("playing " + fileName)
            # open WAV file
            wf = wave.open(fileName, 'rb')
            # read a chunk
            data = wf.readframes(CHUNK)
            # read rest
            while data != b'':
                self.stream.write(data)
                data = wf.readframes(CHUNK)
            # clean up
            wf.close()
        except BaseException as err:
            print(f"Exception! {err=},
  {type(err)=}.\nExiting.")
            exit(0)

    def playRandom(self):
        """play a random note"""
        index = random.randint(0, len(self.notes)-1)
        note = self.notes[index]
        self.play(note)

# main() function
def main():
    # declare global var
    global gShowPlot

    parser = argparse.ArgumentParser(description="Generating
  sounds with
                                                 Karplus-Strong
  Algorithm.")
    # add arguments
    parser.add_argument('--display', action='store_true',
  required=False)
    parser.add_argument('--play', action='store_true',
  required=False)
```

```
    args = parser.parse_args()

    # show plot if flag set
    if args.display:
        gShowPlot = True
        # plt.ion()
        plt.show(block=False)

    # create note player
    nplayer = NotePlayer()

    print('creating notes...')
    for name, freq in list(pmNotes.items()):
        fileName = name + '.wav'
        if not os.path.exists(fileName) or args.display:
            data = generateNote(freq)
            print('creating ' + fileName + '...')
            writeWAVE(fileName, data)
        else:
            print('fileName already created. skipping...')

        # add note to player
        nplayer.add(name + '.wav')

        # play note if display flag set
        if args.display:
            nplayer.play(name + '.wav')
            time.sleep(0.5)

    # play a random tune
    if args.play:
        while True:
            try:
                nplayer.playRandom()
                # rest - 1 to 8 beats
                rest = np.random.choice([1, 2, 4, 8], 1,
                                        p=[0.15, 0.7, 0.1,
  0.05])
                time.sleep(0.25*rest[0])
            except KeyboardInterrupt:
                exit()

# call main
```

```
if __name__ == '__main__':
    main()
```

5
FLOCKING BOIDS



Look closely at a flock of birds or a school of fish, and you'll notice that although the group is composed of individual creatures, the group as a whole seems to have a life of its own. The birds in a flock align with each other as they move and flow over and around obstacles. They break formation when disturbed or startled, but then they regroup, as if controlled by some larger force.

In 1986, Craig Reynolds created a realistic-looking simulation of the flocking behavior of birds called the *Boids model*. One remarkable thing about the Boids model (named after the stereotypical New Yorker's pronunciation of the word *birds*) is that only three simple rules govern the interaction between individuals in the flock, yet the model produces remarkably realistic flocking behavior. The Boids model is widely studied and has even been used to animate computer-generated swarms like the marching penguins in the movie *Batman Returns* (1992).

In this project, you'll use Reynolds's three rules to create a Boids simulation of the flocking behavior of N birds and plot their positions and directions of movement over time. You'll also provide a method to add a bird to the flock, as well as a scatter effect that you can use to study the impact of a local disturbance on the flock. Boids is called an N-body

simulation because it models a dynamic system of *N* particles that exert forces on each other.

How It Works

The three core rules of the Boids simulation are as follows:

Separation Keep a minimum distance between the boids.

Alignment Point each boid in the average direction of movement of its local flockmates.

Cohesion Move each boid toward the center of mass of its local flockmates.

Boids simulations can add other rules too, such as ones to avoid obstacles or scatter the flock when it’s disturbed, as you’ll learn in the following sections. To create the Boids animation, you’ll do the following for every time step in the simulation:

1. For all boids in the flock:
    a. Apply the three core rules.
    b. Apply any additional rules.
    c. Apply all boundary conditions.
2. Update the positions and velocities of the boids.
3. Plot the new positions and velocities.

As you’ll see, these simple steps create a flock with evolving, complex behavior.

Requirements

These are the Python modules you'll be using in this simulation:

- `numpy` arrays to store the positions and velocities of the boids
- The `matplotlib` library to animate the boids
- `argparse` to process command line options
- The `scipy.spatial.distance` module, which has some really neat methods for calculating distances between points

I chose to use `matplotlib` for boids as a matter of simplicity and convenience. To draw a huge number of boids as quickly as possible, you might use something like the OpenGL library. We'll explore graphics in more detail in Part III of this book.

The Code

You'll encapsulate the behavior of a group of boids in a class called `Boids`. First you'll set the initial positions and velocities of the boids. Next, you'll set up the boundary conditions for the simulation, look at how the boids are drawn, and implement the Boids simulation rules discussed earlier. Finally, you'll add some interesting events to the simulation by allowing the user to add boids and scatter the flock. To see the full project code, skip ahead to "The Complete Code" on page 96. You can also download it from the book's GitHub repository at *https://github.com/mkvenkit/pp2e/blob/main/boids/boids.py*.

Initializing the Simulation

The Boids simulation needs to compute the position and velocities of the boids at each step by pulling information from `numpy` arrays. At the beginning of the simulation, you use the `Boids` class's `__init__()` method to create those arrays and initialize all boids in approximately the center of the screen, with their velocities set in random directions.

```
import argparse
import math
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from scipy.spatial.distance import squareform, pdist
from numpy.linalg import norm

width, height = 640, 480

class Boids:
    """class that represents Boids simulation"""
    def __init__(self, N):
        """initialize the Boids simulation"""
        # init position & velocities
      ❷ self.pos = [width/2.0, height/2.0] +
                   10*np.random.rand(2*N).reshape(N, 2)
        # normalized random velocities
      ❸ angles = 2*math.pi*np.random.rand(N)
      ❹ self.vel = np.array(list(zip(np.cos(angles),
  np.sin(angles))))
        self.N = N
```

First you import the modules required for the program and set the width and height of the simulation window on the screen ❶. Then you begin declaring the `Boids` class. Within the class's `__init__()` method, you create a `numpy` array called `pos` to store the x- and y-coordinates of all the boids ❷. For the initial value of each pair of coordinates, you start with the center of the window, `[width/2.0, height/2.0]`, and add a random displacement of up to 10 units. The code

`np.random.rand(2*N)` creates a one-dimensional array of 2*N* random numbers in the range [0, 1], which you multiply by 10 to scale to a range of [0, 10]. The `reshape()` call converts the one-dimensional array into a two-dimensional array of shape (*N*, 2), perfect for storing *N* pairs of x- and y-coordinates. Notice, too, the `numpy` broadcasting rules in action here: the 1×2 array `[width/2.0, height/2.0]` representing the center of the window is added to each element in the *N*×2 array to randomly offset each boid's position from the center.

Next, you create an array of random unit velocity vectors (these are vectors of magnitude 1.0, pointing in random directions) for each boid using the following method: given an angle *t*, the pair of numbers (cos(*t*), sin(*t*)) lie on a circle of radius 1.0, centered at the origin (0, 0). If you draw a line from the origin to a point on this circle, it becomes a unit vector that depends on the angle *t*. So if you choose *t* at random, you end up with a random velocity vector. Figure 5-1 illustrates this scheme.

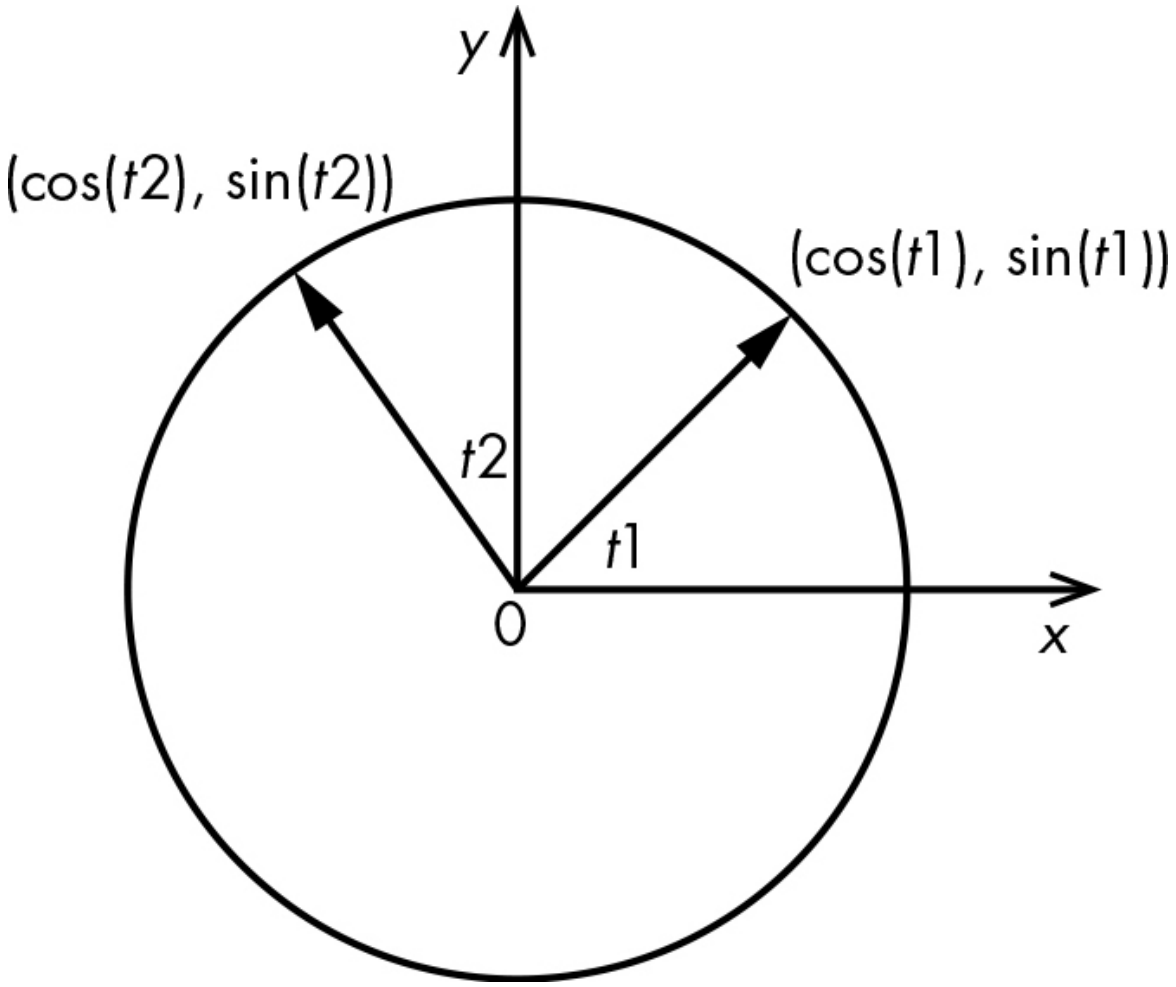


Figure 5-1: *Generating random unit velocity vectors*

Returning to the code, you put this method into action by first generating an array of *N* random angles in the range [0, 2π] ❸. Then you create an array of random unit velocity vectors by calculating the cosine and sine of those angles ❹. You group the coordinates of each vector using Python's built-in `zip()` method. The following is a simple example of `zip()`. This joins two lists into a list of tuples. The `list()` is needed because just calling `zip` will create only an iterator—you need all the elements in the list.

```
>>> list(zip([0, 1, 2], [3, 4, 5]))
[(0, 3), (1, 4), (2, 5)]
```

In summary, you've generated two arrays that will be useful throughout the simulation, `pos` and `vel`. The first contains random positions clustered within a 10-pixel radius around the center of the screen, and the second contains unit velocities pointing in random directions. This means that at the start of the simulation, the boids will all hover around the center of the screen, pointed in random directions.

The `__init__()` method continues by declaring some constant values that will help govern the simulation:

```
        # min dist of approach
    ❶ self.minDist = 25.0
        # max magnitude of velocities calculated by "rules"
    ❷ self.maxRuleVel = 0.03
        # max magnitude of final velocity
    ❸ self.maxVel = 2.0
```

Here you define the minimum distance of approach between two boids ❶. You'll use this value later to apply the separation rule. Then you define `maxRuleVel`, which limits how much a boid's velocity can be changed each time one of the simulation rules is applied ❷. You also define `maxVel`, which sets an overall limit on the boids' velocities ❸.

Setting Boundary Conditions

Birds fly in a boundless sky, but the boids must play in limited space. To create that space, you'll set boundary conditions, as you did with the toroidal boundary condition in the Game of Life simulation in Chapter 3. In this case, you'll apply a *tiled boundary condition* (actually the continuous space version of the boundary condition you used in Chapter 3).

Think of the Boids simulation as taking place in a tiled space: when a boid moves out of a tile, it moves in from the opposite direction to an identical tile. The main difference between the toroidal and tiled boundary conditions is that this Boids simulation won't take place on a discrete grid; instead, the birds move over a continuous region. Figure 5-2 shows what those tiled boundary conditions look like.

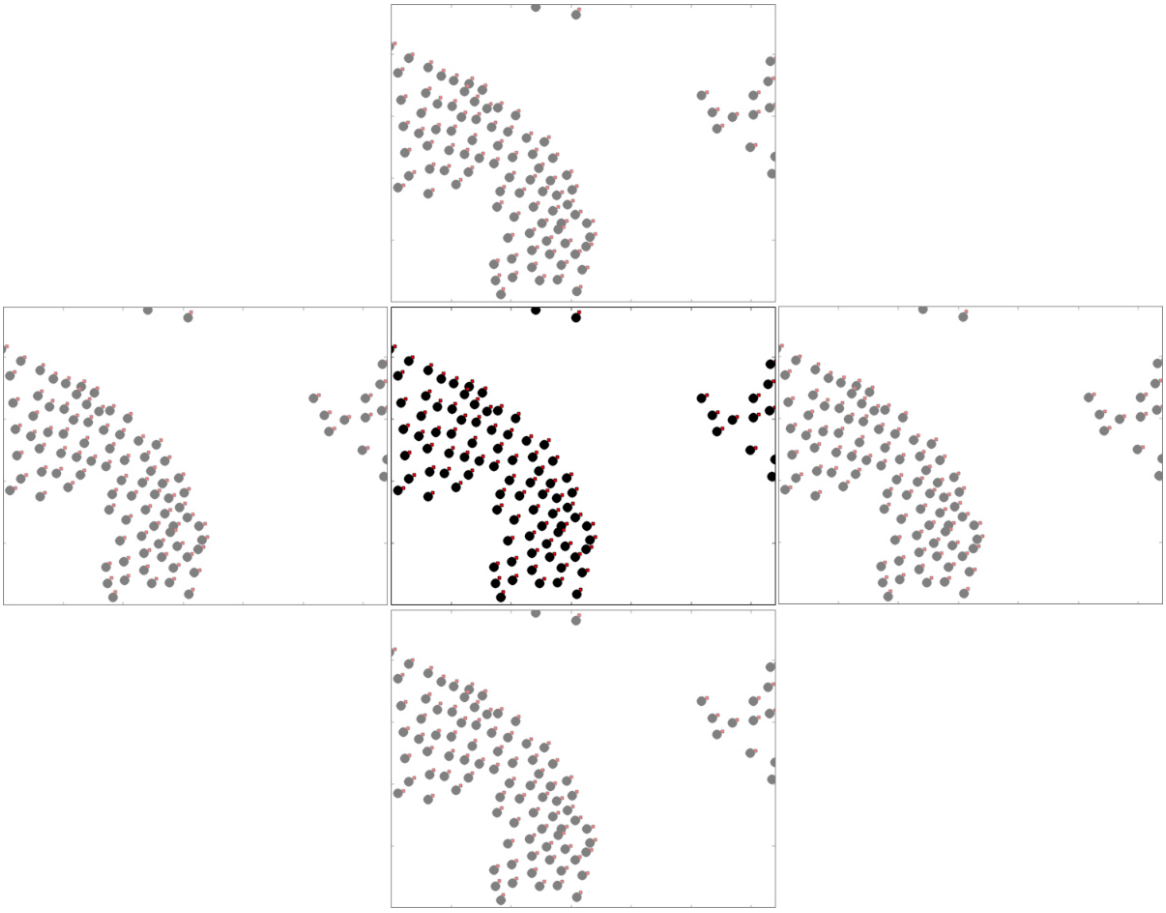


Figure 5-2: *Tiled boundary conditions*

Look at the tile in the middle. The birds flying out to the right are entering the tile on the right, but the boundary conditions ensure that they actually come right back into the center tile through the tile at the left. You can see the same thing happening at the top and bottom tiles.

You implement the tiled boundary conditions for the Boids simulation as a method on the `Boids` class:

```
def applyBC(self):
    """apply boundary conditions"""
    deltaR = 2.0
    for coord in self.pos:
      ❶ if coord[0] > width + deltaR:
            coord[0] = - deltaR
        if coord[0] < - deltaR:
            coord[0] = width + deltaR
        if coord[1] > height + deltaR:
            coord[1] = - deltaR
        if coord[1] < - deltaR:
            coord[1] = height + deltaR
```

This method applies the tiled boundary conditions to each set of boid coordinates in the `pos` array. For example, if an x-coordinate is greater than the width of the window ❶, you set it back to the left edge of the window. The `deltaR` in this line provides a slight buffer, which allows the boid to move slightly outside the window before it starts coming back in from the opposite direction, thus producing a better visual effect. You perform a similar check at the left, top, and bottom edges of the window.

Drawing a Boid

To build the animation, you need to know each boid's position and velocity and have a way to indicate both the position and direction of motion at each time step.

Plotting the Boid's Body and Head

To animate the boids, you use `matplotlib` and a little trick to plot both the position and velocity. Draw each boid as two circles, as shown in Figure 5-3. The larger circle represents the body, and the smaller one represents the head. Point *P* marks the center of the body. For our purposes, you can consider *P* to be the position of the boid, and you'll set it using coordinates from the `pos` array discussed earlier. Point *H* is the center of the head. You calculate the position of *H* according to the formula $H = P + k \times V$, where *V* is the velocity of the boid and *k* is a constant representing the distance from the center of the body to the center of the head. This way, the boid's head will be aligned with its direction of motion at any given time, which visually communicates the boid's direction of movement more clearly than just drawing the body alone.

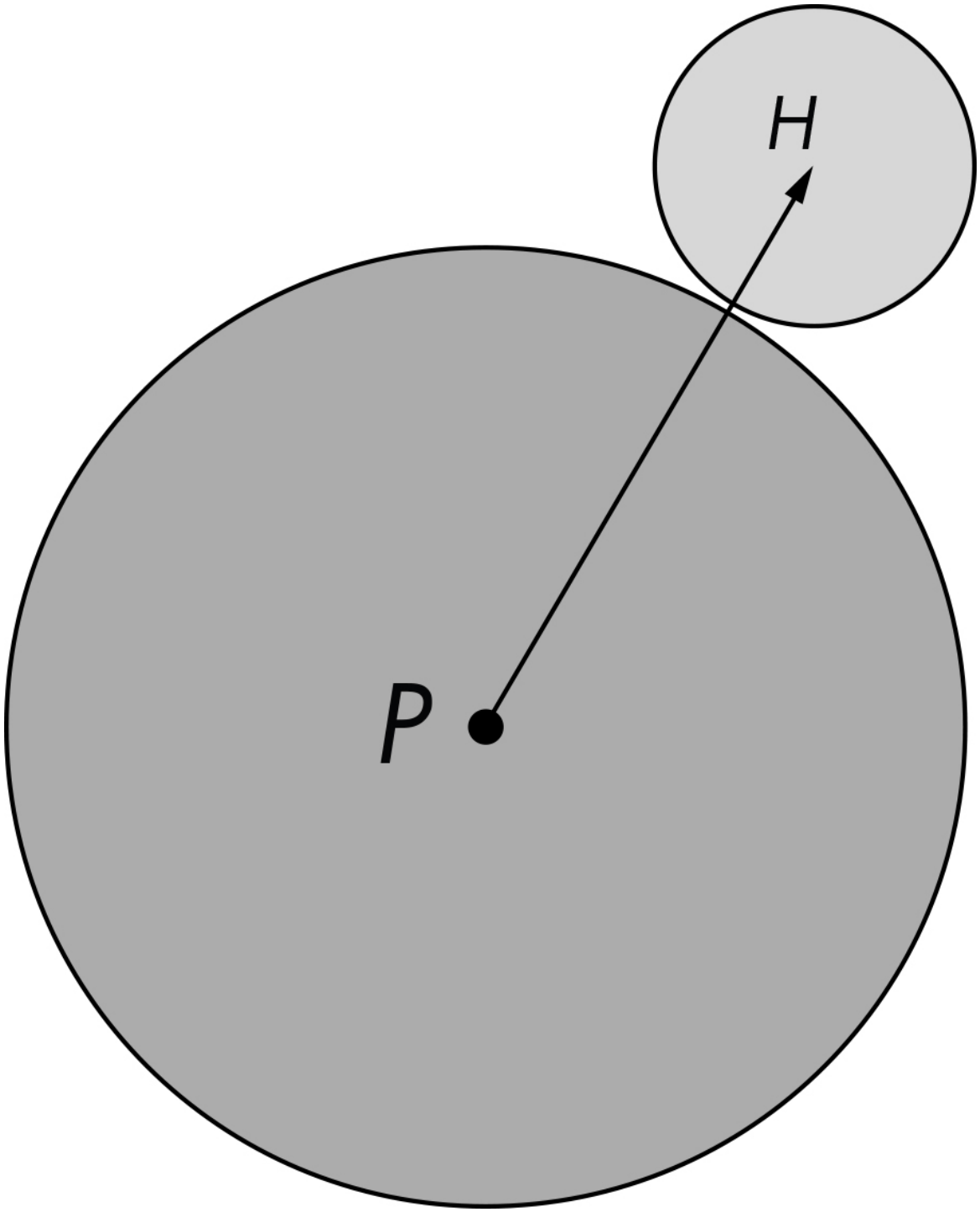


Figure 5-3: *Representing a boid*

In the following snippet from the program's `main()` function, you draw the boid's body and head as circular markers using `matplotlib`:

```
fig = plt.figure()
ax = plt.axes(xlim=(0, width), ylim=(0, height))

pts, = ax.plot([], [], markersize=10, c='k', marker='o',
ls='None')
head, = ax.plot([], [], markersize=4, c='r', marker='o',
ls='None')
anim = animation.FuncAnimation(fig, tick, fargs=(pts[0], head,
boids),
                               interval=50)
```

You set the size and shape of the markers for the boid's body (`pts`) ❶ and head (`head`) ❷. The `'k'` and `'r'` strings specify the colors black and red, respectively, and `'o'` produces a circular marker. The `ax.plot()` method returns a list of `matplotlib.lines.Line2D` objects. The , syntax in these lines picks up the first and only element in this list.

You next initialize a `matplotlib animation.FuncAnimation()` object ❸, which sets up a callback function `tick()` to be called for every frame of the animation (we'll look at this function later in the chapter). The `fargs` parameter lets you specify the arguments of the callback function, and you also set the time interval (50 milliseconds in this case) at which this function will be called. Now that you know how to draw the body and the head, let's see how to update their positions.

Updating the Boid's Position

Once the animation starts, you need to update both the boid's position and the location of the head, which tells you the direction in which the boid is moving. You do so with this code:

```
vec = self.pos + 10*self.vel/self.maxVel
```

```
head.set_data(vec.reshape(2*self.N)[::2], vec.reshape(2*self.N)
  [1::2])
```

First you calculate the position of the head by applying the $H = P + k \times V$ formula discussed earlier. You use a k value of 10 units in the direction of the velocity (`vel`). You then update (`reshape`) the `matplotlib` axis (`set_data`) with the new values of the head position. The `[::2]` picks out the even-numbered elements (x-axis values) from the velocity list, and the `[1::2]` picks out the odd-numbered elements (y-axis values).

Applying the Rules of the Boids

In this section, we'll look at how to implement the three rules of the Boids simulation—separation, alignment, and cohesion—to recalculate the boids' velocities at each time step. We'll start by focusing just on the separation rule. The goal is to generate a new velocity vector for each boid that pushes it away from its nearby flockmates, defined as all the boids within a certain radius R. Given two boids i and j with positions P_i and P_j, $P_i - P_j$ produces a new velocity vector for boid i that points away from boid j. We'll call this a *displacement vector*. To calculate a new velocity vector V_i for boid i that on average pushes it away from *all* its nearby flockmates, simply sum all the displacement vectors between boid i and each boid within radius R. In other words, $V_i = (P_i - P_1) + (P_i - P_2) + \ldots (P_i - P_N)$, provided the distance between boids i and j is less than R. You can write this more formally as:

$$V_i = \begin{cases} \Sigma\,(P_i - P_j)\ if\ |P_i - P_j| < R \\ 0\ otherwise \end{cases}$$

Notice that implementing this rule—and, indeed, implementing the other Boids rules as well—involves calculating the distance between each boid and every other boid to determine which boids are local flockmates. The traditional way to do this in Python would be to use a pair of nested loops to iterate through the boids. As you'll see, however, `numpy` arrays provide more efficient methods that bypass the need for loops. We'll implement both approaches and compare the results and then apply what we've learned to the actual simulation code.

Using Nested Loops

First, let's define a function `test1()` that implements the separation rule in a straightforward way, using loops:

```
def test1(pos, radius):
    # fill output with zeros
    vel = np.zeros(2*N).reshape(N, 2)
    # for each pos
  ❶ for (i1, p1) in enumerate(pos):
        # velocity contribution
        val = np.array([0.0, 0.0])
        # for each other pos
      ❷ for (i2, p2) in enumerate(pos):
            if i1 != i2:
                # calculate distance from p1
                dist = math.sqrt((p2[0]-p1[0])*(p2[0]-p1[0]) +
                                 (p2[1]-p1[1])*(p2[1]-p1[1]))
                # apply threshold
              ❸ if dist < radius:
                  ❹ val += (p2 - p1)
        # set velocity
        vel[i1] = val
    # return computed velocity
    return vel
```

This code uses a nested pair of loops. The outer loop ❶ goes through each boid in the `pos` array. The inner loop ❷ computes the distance between the current boid and each other boid in the array. If the distance is less than the threshold defined as the function's `radius` parameter ❸, you calculate the displacement vector as discussed earlier and add the result to `val` ❹. At the end of each cycle of the inner loop, `val` holds a new velocity that will push the current boid away from its neighbors. You store that velocity back in the `vel` array.

Using numpy Methods

Now let's define a function `test2()` that does the same thing "the `numpy` way," avoiding loops and using highly optimized `numpy` methods. You'll also use methods from the `scipy.spatial.distance` module to efficiently calculate the distance between points:

```
def test2(pos, radius):
    # get distance matrix
  ❶ distMatrix = squareform(pdist(pos))
    # apply threshold
  ❷ D = distMatrix < radius
    # compute velocity
  ❸ vel = pos*D.sum(axis=1).reshape(N, 1) - D.dot(pos)
    return vel
```

You use the `squareform()` and `pdist()` methods (defined in the `scipy` library) to calculate the distances between every possible pair of points in the `pos` array ❶. For an array of N points, `squareform()` gives you an $N \times N$ matrix, where any given entry M_{ij} is the distance between points P_i and P_j. Let's consider a quick example of how that looks. In this code, you call the methods on an array of three points:

```
>>> import numpy as np
>>> from scipy.spatial.distance import squareform, pdist
>>> x = np.array([[0.0, 0.0], [1.0, 1.0], [2.0, 2.0]])
>>> squareform(pdist(x))
array([[0.        , 1.41421356, 2.82842712],
       [1.41421356, 0.        , 1.41421356],
       [2.82842712, 1.41421356, 0.        ]])
```

Since you provide an array of three points, the result is a 3×3 matrix of distance calculations. The values in the first row, for example, tell you the distance between the first point (`[0.0, 0.0]`) and each point in the array. The zeros running diagonally down the array correspond to the distance between each point and itself.

Returning to the `test2()` function, you next filter the matrix based on whether the distance is less than the specified `radius` ❷. Using the same example array of three points, you have the following:

```
>>> squareform(pdist(x)) < 1.4
array([[ True, False, False],
       [False,  True, False],
       [False, False,  True]])
```

The `<` comparison creates a Boolean matrix of `True/False` values corresponding to the original distance matrix—`True` if a distance is less than the given threshold (in this example, 1.4).

Back in `test2()`, you use a modified version of the equation for V_i discussed earlier, broadcast over the entire `pos` array ❸. That equation can be rewritten as:

$$V_i = mP_i - \sum_{d<D} P_k$$

Here, the second summation term on the right includes only the points *P* that satisfy the distance criteria. The number of elements in the summation term is *m*. This equation can again be written as:

$$V_i = mP_i - \sum D_{ij} P_j$$

where D_{ij} is row *i* of the Boolean matrix you generated ❷, *m* is the number of `True` values in that row, and P_j is all the points *P* that fall within the specified radius of the current boid.

The `D.sum` method ❸ adds up the `True` values in the Boolean matrix in a column-wise fashion, giving you *m* from the equation. The reshape is required because the result of the sum is a one-dimensional array of *N* values (shape (*N*,)), and you want it to be of shape (*N*, 1) so it's compatible for multiplication with the position array. The `D.dot(pos)` part of the line ❸ is taking the dot product (multiplication) of the Boolean matrix and the array of boid positions, which corresponds to the $D_{ij}P_j$ part of the equation.

Comparing Approaches

Comparing the two approaches, `test2()` is much more compact than `test1()`, but its real advantage is speed. Let's use the Python `timeit` module to evaluate the performance of the two functions. First, enter the code for the functions `test1()` and `test2()` in a file named *test.py*, as shown here:

```
import math
import numpy as np
from scipy.spatial.distance import squareform, pdist, cdist

N = 100
width, height = 640, 480
pos = np.array(list(zip(width*np.random.rand(N),
  height*np.random.rand(N))))

def test1(pos, radius):
--snip--

def test2(post, radius):
--snip--
```

Now use the `timeit` module in a Python interpreter session to compare the performance of the two functions:

```
>>> from timeit import timeit
>>> timeit('test1(pos, 100)', 'from test import test1, N, pos,
     width, height', number=100)
7.880876064300537
>>> timeit('test2(pos, 100)', 'from test import test2, N, pos,
     width, height', number=100)
0.036969900131225586
```

On my computer, the `numpy` code without loops runs about 200 times faster than the code that uses explicit loops! But why? Aren't they both doing more or less the same thing?

As an interpreted language, Python is inherently slower than compiled languages like C. The `numpy` library brings the convenience of Python and performance nearly equal to that of C by providing highly optimized methods that operate on arrays of data. In general, you'll find that `numpy` works best when you reorganize your algorithm as steps that operate on entire arrays at once, without looping through individual elements to perform computations.

Writing the Final Method

Now that you've compared the two approaches, you're ready to use what you've learned to write a final version of the method that applies all three rules of the simulation and returns updated velocities for all the boids. The `applyRules()` method, part of the `Boids` class, uses the optimized `numpy` techniques discussed earlier.

```
def applyRules(self):
    # get pairwise distances
  ❶ self.distMatrix = squareform(pdist(self.pos))
    # apply rule #1: separation
    D = self.distMatrix < self.minDist
  ❷ vel = self.pos*D.sum(axis=1).reshape(self.N, 1) -
  D.dot(self.pos)
  ❸ self.limit(vel, self.maxRuleVel)

    # distance threshold for alignment (different from
  separation)
  ❹ D = self.distMatrix < 50.0

    # apply rule #2: alignment
  ❺ vel2 = D.dot(self.vel)
    self.limit(vel2, self.maxRuleVel)
  ❻ vel += vel2

    # apply rule #3: cohesion
  ❼ vel3 = D.dot(self.pos) - self.pos
    self.limit(vel3, self.maxRuleVel)
  ❽ vel += vel3

    return vel
```

You compute the pairwise distance matrix between the boids using the `squareform()` and `pdist()` methods from the `scipy` library, as discussed earlier ❶. When you apply the separation rule using `numpy` methods ❷, each boid is pushed away from neighboring boids within a distance of `minDist` (25 pixels). The calculated velocities are clamped, or

restricted, to a certain maximum value using the `Boids` class's `limit()` method ❸, which we'll look at later. Without this restriction, the velocities would increase with each time step, and the simulation would go haywire.

You next generate a new Boolean matrix using a distance threshold of 50 pixels rather than 25 ❹. You'll use this broader definition of neighboring flockmates to apply the alignment and cohesion rules. The alignment rule is implemented so that each boid is influenced by and aligns itself with the average velocity of its neighbors. You get that average simply by taking the dot product of `D` (the Boolean matrix) and the velocity array ❺. Once again, you restrict the calculated velocities to a maximum so they don't increase indefinitely. (Using the compact `numpy` syntax makes all these computations simple and fast.)

Finally, you apply the cohesion rule by adding the positions of all the neighboring boids and then subtracting the position of the current boid ❼. This produces a velocity vector for each boid that points to the *centroid* or geometric center of its neighbors. Again, you limit the velocities to keep them from getting out of control.

Each of the three rules produces its own velocity vector for each boid. At ❻ and ❽, you add these vectors together, producing an overall velocity vector for each boid that reflects the influence of all three simulation rules. You store the final velocity vectors in the `vel` array.

Limiting the Velocities

In the previous section, you saw how the `limit()` method was called after applying each rule to keep the boids' velocities from getting out of control. Here's that method:

```
def limit(self, X, maxVal):
```

```
    """limit the magnitude of 2D vectors in array X to
  maxValue"""
  ❶ for vec in X:
        self.limitVec(vec, maxVal)
```

This method is designed to take in an array of velocity vectors, extract each individual vector ❶, and pass it along to the `limitVec()` method, which looks like this:

```
def limitVec(self, vec, maxVal):
    """limit the magnitude of the 2D vector"""
  ❶ mag = norm(vec)
    if mag > maxVal:
      ❷ vec[0], vec[1] = vec[0]*maxVal/mag, vec[1]*maxVal/mag
```

You use the `norm()` function from the `numpy` library to calculate the magnitude of the vector ❶. If it exceeds the maximum, you scale the x and y portions of the vector in proportion to the vector's magnitude ❷. The maximum value was defined as `self.maxRuleVel = 0.03` as part of the initialization of the `Boids` class.

Influencing the Simulation

The core rules in the Boids simulation will cause the boids to automatically exhibit flocking behavior. But let's make things more interesting by allowing the user to influence the simulation as it runs. Specifically, you'll create the ability to add boids to the flock or make the flock scatter with the click of your mouse.

The first step to inject events into the simulation while it's running is to add an *event handler* to the `matplotlib` canvas. This is a piece of code that calls a function every time a certain event, such as a mouse click, happens. Here's how to do it:

```
cid = fig.canvas.mpl_connect('button_press_event',
  boids.buttonPress)
```

You use the `mpl_connect()` method to add a button press event handler to the `matplotlib` canvas. This handler will call the `buttonPress()` method of the `Boids` class every time a mouse button is pressed in the simulation window. Next, you need to define the `buttonPress()` method.

Adding a Boid

The first part of the `buttonPress()` method adds a boid to the simulation at the location of your cursor and assigns that boid a random velocity, when the *left* mouse button is pressed.

```
def buttonPress(self, event):
"""event handler for matplotlib button presses"""
# left-click to add a boid
if event.button is 1:
  ❷ self.pos = np.concatenate((self.pos,
                               np.array([[event.xdata,
  event.ydata]])),
                               axis=0)
    # generate a random velocity
    angles = 2*math.pi*np.random.rand(1)
    v = np.array(list(zip(np.sin(angles), np.cos(angles))))
  ❸ self.vel = np.concatenate((self.vel, v), axis=0)
  ❹ self.N += 1
```

First you ensure that the mouse event is a left-click ❶. Then you append the mouse location given by (`event.xdata`, `event.ydata`) to the array of boid positions ❷. You also generate a random velocity vector, add it to the array of boid velocities ❸, and increment the count of boids by 1 ❹.

Scattering the Boids

The three simulation rules keep the boids in a flock as they move around. But what happens when the flock is disturbed? To simulate this situation, you can introduce a "scatter" effect: when you right-click in the simulation window, the flock will scatter from the location of the click. You can think of this as how the flock might respond to the sudden appearance of a predator or a loud noise that spooks the birds. You implement this effect as a continuation of the `buttonPress()` method:

```
    # right-click to scatter boids
❶ elif event.button is 3:
        # add scattering velocity
        self.vel += 0.1*(self.pos - np.array([[event.xdata,
event.ydata]]))
```

Here you check whether the mouse button press is a right-click event ❶. If so, you change the velocity for every boid by adding a vector that points away from the point where the disturbance arose (that is, where the mouse was clicked). You calculate this vector much like you calculated the displacement vectors for the separation rule. If P_i is the position of a boid and P_m is the point where the mouse was clicked, $P_i - P_m$ is a vector that points away from the mouse click. You multiply this vector by `0.1` to keep the magnitude of the disturbance small. Initially, the boids will fly away from that point, but as you'll see, the three rules prevail, and the boids will coalesce again as a flock.

Incrementing the Simulation

At each time step in the simulation, you need to apply the rules to calculate the boids' new velocities, update the

boids' positions based on those velocities, enforce the boundary conditions, and redraw everything in the display window. You can coordinate all this activity from the `tick()` function, which will be called at each frame of the `matplotlib` animation.

```
def tick(frameNum, pts, head, boids):
    """update function for animation"""
    boids.tick(frameNum, pts, head)
    return pts, head
```

The stand-alone `tick()` function simply calls the `tick()` method of the `Boids` class. The latter is defined as follows:

```
def tick(self, frameNum, pts, head):
    """update the simulation by one time step"""
    # apply rules
  ❶ self.vel += self.applyRules()
  ❷ self.limit(self.vel, self.maxVel)
  ❸ self.pos += self.vel
  ❹ self.applyBC()
    # update data
  ❺ pts.set_data(self.pos.reshape(2*self.N)[::2],
                 self.pos.reshape(2*self.N)[1::2])
  ❻ vec = self.pos + 10*self.vel/self.maxVel
  ❼ head.set_data(vec.reshape(2*self.N)[::2],
                  vec.reshape(2*self.N)[1::2])
```

This method is where everything comes together. You apply the boid rules using the `applyRules()` method that we already looked at ❶. Then you limit the computed velocities of the boids using the `self.maxVel` threshold ❷. (Even though you limited the velocity vector generated by each individual rule, the overall velocity determined by adding all three rules together may still be too large.) You next compute the updated positions of the boids by adding the new velocity

vectors to the old array of positions ❸. For example, if a boid was at position [0, 0] and has a velocity vector of [1, 1], its new position after one time step would be [1, 1]. You apply the boundary conditions for the simulation by calling `applyBC()` ❹.

The call to `pts.set_data()` ❺ updates the `matplotlib` axis with the boids' new positions. The `[::2]` picks out the even-numbered elements (x-axis values) from the `pos` array, and the `[1::2]` picks out the odd-numbered elements (y-axis values). This will redraw the larger circles representing the boids' bodies. Next, you need to draw the smaller circles representing the boids' heads. You calculate the position of each boid's head so it will point in the boid's direction of motion by applying the $H = P + k \times V$ formula discussed earlier ❻. Recall that P is the center of a boid's body, k is a constant representing the distance from the center of the body to the center of the head (you use a value of 10 units), and V is the boid's velocity. Once you have the new head positions, you draw them via the same technique you used to draw the bodies ❼.

Parsing Arguments and Instantiating the Boids

The program's `main()` function begins by handling command line arguments and instantiating the `Boids` class:

```
def main():
    # use sys.argv if needed
    print('starting boids...')

    parser = argparse.ArgumentParser(description="Implementing
  Craig
                                     Reynolds's Boids...")
    # add arguments
  ❶ parser.add_argument('--num-boids', dest='N',
  required=False)
```

```
    args = parser.parse_args()

    # set the initial number of boids
❷   N = 100
    if args.N:
        N = int(args.N)

    # create boids
❸   boids = Boids(N)
```

You use the familiar `argparse` module to create a command line option for setting the initial number of boids in the simulation ❶. If no argument is provided at the command line, the simulation defaults to 100 boids ❷. You set the simulation in motion by creating an object of the `Boids` class ❸.

The `main()` function continues with the code to create and animate a `matplotlib` plot. We've already discussed this code in "Plotting the Boid's Body and Head" on page 84.

Running the Boids Simulation

Let's see what happens when you run the simulation. Enter the following:

```
$ python boids.py
```

The Boids simulation should start with all the boids clustered around the center of the window. Let the simulation run for a while, and the boids should start to flock as they form a pattern similar to the one shown in Figure 5-4.

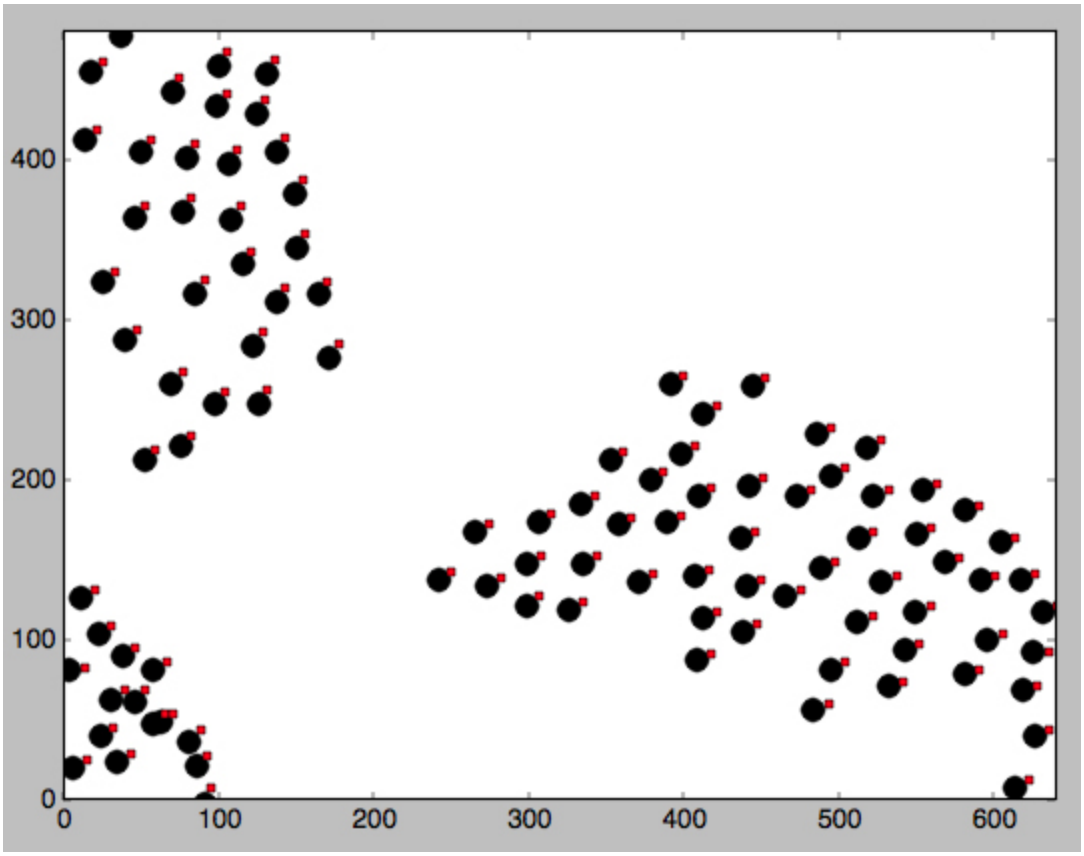


Figure 5-4: *A sample run of the Boids simulation*

Left-click the simulation window. A new boid should appear at that location, and its velocity should change as it encounters the flock. Now right-click. The flock should initially scatter from your cursor but then recoalesce.

Summary

In this project, you simulated the flocking of birds (or boids) using the three rules proposed by Craig Reynolds. You looked at how operating on an entire `numpy` array at once is much faster than performing the same operations inside an explicit loop. You used the `scipy.spatial` module to perform fast and convenient distance calculations, and you implemented a `matplotlib` trick that uses two markers to

represent both the position and the direction of points. Finally, you added interactivity to the simulation in the form of an event handler to respond to button presses within the `matplotlib` plot.

Experiments!

Here are some ways you might further explore flocking behavior:

1. Implement obstacle avoidance for your flock of boids by writing a new method called `avoidObstacle()` and applying it right after you apply the three rules, as follows:

```
self.vel += self.applyRules()
self.vel += self.avoidObstacle()
```

   The `avoidObstacle()` method should use a predefined tuple (x, y, R) to add an additional velocity term to a boid, pushing it away from the obstacle location (x, y), but only when the boid is within radius R of the obstacle. Think of this as the distance at which a boid sees the obstacle and steers away from it. You can specify the (x, y, R) tuple using a command line option.
2. What happens when the boids fly through a strong gust of wind? Simulate this by adding a global velocity component to all the boids at random time steps in the simulation. The boids should temporarily be affected by the wind but return to the flock once the wind stops.

The Complete Code

Here’s the complete code for the Boids simulation:

```
"""
boids.py

An implementation of Craig Reynolds's Boids simulation.

Author: Mahesh Venkitachalam
"""

import argparse
import math
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from scipy.spatial.distance import squareform, pdist
from numpy.linalg import norm

width, height = 640, 480

class Boids:
    """class that represents Boids simulation"""
    def __init__(self, N):
        """initialize the Boids simulation"""
        # init position & velocities
        self.pos = [width/2.0, height/2.0] +
                   10*np.random.rand(2*N).reshape(N, 2)
        # normalized random velocities
        angles = 2*math.pi*np.random.rand(N)
        self.vel = np.array(list(zip(np.cos(angles),
     np.sin(angles))))
        self.N = N
        # min dist of approach
        self.minDist = 25.0
        # max magnitude of velocities calculated by "rules"
        self.maxRuleVel = 0.03
        # max magnitude of final velocity
        self.maxVel = 2.0

    def tick(self, frameNum, pts, head):
```

```
    """update the simulation by one time step"""
    # apply rules
    self.vel += self.applyRules()
    self.limit(self.vel, self.maxVel)
    self.pos += self.vel
    self.applyBC()
    # update data
    pts.set_data(self.pos.reshape(2*self.N)[::2],
                 self.pos.reshape(2*self.N)[1::2])
    vec = self.pos + 10*self.vel/self.maxVel
    head.set_data(vec.reshape(2*self.N)[::2],
                  vec.reshape(2*self.N)[1::2])

def limitVec(self, vec, maxVal):
    """limit magnitude of 2D vector"""
    mag = norm(vec)
    if mag > maxVal:
        vec[0], vec[1] = vec[0]*maxVal/mag,
 vec[1]*maxVal/mag

def limit(self, X, maxVal):
    """limit magnitude of 2D vectors in array X to
 maxValue"""
    for vec in X:
        self.limitVec(vec, maxVal)

def applyBC(self):
    """apply boundary conditions"""
    deltaR = 2.0
    for coord in self.pos:
        if coord[0] > width + deltaR:
            coord[0] = - deltaR
        if coord[0] < - deltaR:
            coord[0] = width + deltaR
        if coord[1] > height + deltaR:
            coord[1] = - deltaR
        if coord[1] < - deltaR:
            coord[1] = height + deltaR

def applyRules(self):
    # get pairwise distances
```

```
        self.distMatrix = squareform(pdist(self.pos))
        # apply rule #1 - separation
        D = self.distMatrix < self.minDist
        vel = self.pos*D.sum(axis=1).reshape(self.N, 1) -
    D.dot(self.pos)
        self.limit(vel, self.maxRuleVel)

        # different distance threshold
        D = self.distMatrix < 50.0

        # apply rule #2 - alignment
        vel2 = D.dot(self.vel)
        self.limit(vel2, self.maxRuleVel)
        vel += vel2;

        # apply rule #1 - cohesion
        vel3 = D.dot(self.pos) - self.pos
        self.limit(vel3, self.maxRuleVel)
        vel += vel3

        return vel

    def buttonPress(self, event):
        """event handler for matplotlib button presses"""
        # left-click - add a boid
        if event.button == 1:
            self.pos = np.concatenate((self.pos,
                                       np.array([[event.xdata,
    event.ydata]])),
                                       axis=0)
            # random velocity
            angles = 2*math.pi*np.random.rand(1)
            v = np.array(list(zip(np.sin(angles),
    np.cos(angles))))
            self.vel = np.concatenate((self.vel, v), axis=0)
            self.N += 1
        # right-click - scatter
        elif event.button == 3:
            # add scattering velocity
            self.vel += 0.1*(self.pos - np.array([[event.xdata,
    event.ydata]]))

def tick(frameNum, pts, head, boids):
```

```
    """update function for animation"""
    boids.tick(frameNum, pts, head)
    return pts, head

# main() function
def main():
    # use sys.argv if needed
    print('starting boids...')

    parser = argparse.ArgumentParser(description=
                                     "Implementing Craig
     Reynolds's Boids...")
    # add arguments
    parser.add_argument('--num-boids', dest='N',
     required=False)
    args = parser.parse_args()

    # number of boids
    N = 100
    if args.N:
        N = int(args.N)

    # create boids
    boids = Boids(N)

    # setup plot
    fig = plt.figure()
    ax = plt.axes(xlim=(0, width), ylim=(0, height))

    pts = ax.plot([], [], markersize=10,
                  c='k', marker='o', ls='None')

    head, = ax.plot([], [], markersize=4,
                    c='r', marker='o', ls='None')
    anim = animation.FuncAnimation(fig, tick, fargs=(pts[0],
     head, boids),
                                   interval=50)

    # add a "button press" event handler
    cid = fig.canvas.mpl_connect('button_press_event',
     boids.buttonPress)

    plt.show()

# call main
```

```
if __name__ == '__main__':
    main()
```

PART III

FUN WITH IMAGES

You can observe a lot by watching.

—Yogi Berra

6
ASCII ART



In the 1990s, when email ruled and graphics capabilities were limited, it was common to include a signature in your email that contained a graphic made of text, commonly called *ASCII art*. (ASCII is simply a character-encoding scheme.) Figure 6-1 shows a couple of examples. Although the internet has made sharing images immeasurably easier, the humble text graphic isn’t quite dead yet.

ASCII art has its origins in typewriter art created in the late 1800s. In the 1960s, when computers had minimal graphics processing hardware, ASCII was used to represent images. These days, ASCII art continues as a form of expression on the internet, and you can find a variety of creative examples online.



Figure 6-1: *Examples of ASCII art*

In this project, you'll use Python to create a program that generates ASCII art from graphical images. The program will let you specify the width of the output (the number of columns of text) and set a vertical scale factor. It also supports two mappings of grayscale values to ASCII characters: a sparse 10-level mapping and a more finely calibrated 70-level mapping.

To generate your ASCII art from an image, you'll learn how to do the following:

- Convert images to grayscale using `Pillow`, a fork of the Python Imaging Library (PIL).
- Compute the average brightness of a grayscale image using `numpy`.
- Use a string as a quick lookup table for grayscale values.

How It Works

This project takes advantage of the fact that from a distance, we perceive grayscale images as the average value of their brightness. For example, in Figure 6-2, you can see a grayscale image of a building and, next to it, an image filled with the average brightness value of the building image. If you look at the images from across the room, they will look similar.

ASCII art is generated by splitting an image into tiles and replacing each tile with an ASCII character, based on the tile's average brightness value. Brighter tiles are replaced with sparser ASCII characters (that is, characters that contain a lot of whitespace), such as a period (`.`) or colon (`:`), while darker tiles are replaced with denser ASCII characters, such as an ampersand (`@`) or dollar sign (`$`). From a distance, since our eyes have limited resolution, we sort of see the "average" values in ASCII art while losing the details that would otherwise make the art look less real.



Figure 6-2: *The average value of a grayscale image*

This program will take a given image and first convert it to 8-bit grayscale so that each pixel has a grayscale value in the range [0, 255] (the range of an 8-bit integer). Think of this 8-bit value as the pixel's *brightness*, with 0 being black,

255 being white, and the values in between being shades of gray.

Next, it will split the image into a grid of $M \times N$ tiles (where M is the number of rows and N the number of columns in the ASCII image). The program will then calculate the average brightness value for each tile in the grid and match it with an appropriate ASCII character by predefining a *ramp* (an increasing set of values) of ASCII characters to represent grayscale values in the range [0, 255]. It will use these ramp values as a lookup table for the brightness values.

The finished ASCII art is just a bunch of lines of text. To display the text, you'll use a constant-width (also called *monospace*) font such as Courier because unless each text character has the same width, the characters in the image won't line up properly along a grid, and you'll end up with unevenly spaced and scrambled output.

The *aspect ratio* (the ratio of width to height) of the font used also affects the final image. If the aspect ratio of the space taken up by a character is different from the aspect ratio of the image tile the character is replacing, the final ASCII image will appear distorted. In effect, you're trying to replace an image tile with an ASCII character, so their shapes need to match. For example, if you were to split your image into square tiles and then replace each of the tiles with a font where characters are taller than they are wide, the final output would appear stretched vertically. To address this issue, you'll scale the rows in your grid to match the Courier aspect ratio. (You can send the program command line arguments to modify the scaling to match other fonts.)

In sum, here are the steps the program takes to generate the ASCII image:

1. Convert the input image to grayscale.

2. Split the image into $M \times N$ tiles.
3. Correct M (the number of rows) to match the image and font aspect ratio.
4. Compute the average brightness for each image tile and then look up a suitable ASCII character for each.
5. Assemble rows of ASCII character strings and print them to a file to form the final image.

Requirements

In this project, you'll use `Pillow`, the friendly fork of the Python Imaging Library, to read in the images, access their underlying data, and create and modify them. You'll also use the `numpy` library to compute averages.

The Code

You'll begin by defining the grayscale levels used to generate the ASCII art. Then you'll look at how the image is split into tiles and how average brightness is computed for those tiles. Next, you'll work on replacing the tiles with ASCII characters to generate the final output. Finally, you'll set up command line parsing for the program to allow users to specify the output size, output filename, and so on.

For the full project code, skip to "The Complete Code" on page 109. You can also download the code for this project from *https://github.com/mkvenkit/pp2e/blob/main/ascii/ascii.py*.

Defining the Grayscale Levels and Grid

As the first step in creating your program, define the scales you'll use to convert image brightness values to ASCII characters as global values.

```
# 70 levels of gray
gscale1 = "$@B%8&WM#*oahkbdpqwmZO0QLCJUYXzcvunxrjft/\|()1{}[]?-
     _+~<>i!lI;:,\"^`'. "
# 10 levels of gray
gscale2 = "@%#*+=-:. "
```

The value `gscale1` is a 70-level grayscale ramp, while `gscale2` is a simpler 10-level grayscale ramp. Both of these values are stored as strings, with a range of characters that progress from darkest to lightest. The program will use the `gscale2` ramp by default, but you'll include a command line option to use the more nuanced `gscale1` ramp instead.

NOTE *To learn more about how characters are represented as grayscale values, see Paul Bourke's "Character Representation of Grey Scale Images" at* http://paulbourke.net/dataformats/asciiart/*.*

Now that you have your grayscale ramps, you can set up the image. The following code opens the image using `Pillow` and splits it into a grid:

```
# open the image and convert to grayscale
image = Image.open(fileName).convert("L")
# store the image dimensions
W, H = image.size[0], image.size[1]
# compute the tile width
w = W/cols
# compute the tile height based on the aspect ratio and scale
  of the font
h = w/scale
# compute the number of rows to use in the final grid
```

```
rows = int(H/h)
```

First, `Image.open()` opens the input image file, and `Image.convert()` converts the image to grayscale. The `"L"` stands for *luminance*, a measure of the brightness of an image. You store the width and height (measured in pixels) of the input image ❶. Then you compute the width of a tile for the number of columns (`cols`) specified by the user ❷. (The program uses a default of 80 columns if the user doesn't set another value in the command line.) You use floating-point, not integer, division to avoid truncation errors while calculating the dimensions of the tiles.

Once you know the width of a tile, you compute its height using the vertical scale factor passed in as `scale` ❸. This way, each tile will match the aspect ratio of the font you're using to display the text so that the final image won't be distorted. The value for `scale` can be passed in as an argument, or it's set to a default of `0.43`, which works well for displaying the result in Courier. Having calculated the height of each row, you compute the number of rows in the grid ❹.

Computing the Average Brightness

Next, you need a way to compute the average brightness for a tile in the grayscale image. The function `getAverageL()` does the job.

```
def getAverageL(image):
    # get the image as a numpy array
  ❶ im = np.array(image)
    # get the dimensions
    w,h = im.shape
    # get the average
  ❷ return np.average(im.reshape(w*h))
```

The image tile is passed into the function as a PIL `Image` object. You convert the image into a `numpy` array ❶, at which point `im` becomes a two-dimensional array containing the brightness values of the image's pixels. You store the dimensions (width and height) of the array and then use `numpy.reshape()` to convert the two-dimensional array into a flat one-dimensional array whose length is a product of the original array's width and height (`w*h`). You pass the reshaped array to `numpy.average()`, which sums the array values and computes the average brightness level of the entire image tile ❷.

Generating the ASCII Content from the Image

The main part of the program generates the ASCII content from the image:

```
# an ASCII image is a list of character strings
aimg = []
# generate the list of tile dimensions
for j in range(rows):
    y1 = int(j*h)
    y2 = int((j+1)*h)
    # correct the last tile
    if j == rows-1:
      ❸ y2 = H
    # append an empty string
  ❹ aimg.append("")
  ❺ for i in range(cols):
        # crop the image to fit the tile
        x1 = int(i*w)
        x2 = int((i+1)*w)
        # correct the last tile
        if i == cols-1:
            x2 = W
```

```
        # crop the image to extract the tile into another Image
object
      ❻ img = image.crop((x1, y1, x2, y2))
        # get the average luminance
      ❼ avg = int(getAverageL(img))
        # look up the ASCII character for grayscale value (avg)
        if moreLevels:
          ❽ gsval = gscale1[int((avg*69)/255)]
        else:
          ❾ gsval = gscale2[int((avg*9)/255)]
        # append the ASCII character to the string
      ❿ aimg[j] += gsval
```

In this section of the program, the ASCII image is first stored as a list of strings, which you initialize ❶. Next, you iterate through the rows of image tiles ❷, calculating the top and bottom y-coordinates of each image tile in a given row as *y1* and *y2*. These are floating-point calculations, but you truncate them to integers before passing them to an image-cropping method.

Next, because dividing the image into tiles creates edge tiles of the same size only when the image width is an integer multiple of the number of columns, you correct for the bottom y-coordinate of the tiles in the last row by setting the y-coordinate to the image's actual height (`H`) ❸. By doing so, you ensure that the bottom edge of the image isn't truncated.

You add an empty string into the ASCII image list as a compact way to represent the current image row ❹. You'll fill in this string next. Essentially, you're treating the string as a list of characters that you can append to. Then you iterate over all the tiles in a given row of the image, column by column ❺. You compute the left and right x-coordinates of each tile as *x1* and *x2*. When you get to the last tile in the row, you set the right x-coordinate to the width of the image

(`w`), for the same reasons you corrected the final y-coordinate to the image's height.

You've now calculated (*x1*, *y1*) and (*x2*, *y2*), the coordinates of the top-left and bottom-right corners of the current image tile. You pass these coordinates to `image.crop()` to extract the tile from the complete image ❻. Then you pass that tile (which takes the form of a PIL `Image` object) to the `getAverageL()` function ❼, defined in "Computing the Average Brightness" on page 105, to get the average brightness of the tile. You scale the average brightness value from [0, 255] to [0, 9], the range of values for the default 10-level grayscale ramp ❾. You then use `gscale2` (the stored ramp string) as a lookup table for the relevant ASCII character. The line at ❽ is similar, except it scales the brightness value to the [0, 69] range of the 70-level grayscale ramp. This line will be used only when the `moreLevels` command line flag has been set. Finally, you append the looked-up ASCII character, `gsval`, to the text row ❿, and the code loops until all rows are processed.

Creating Command Line Options

Next, define some command line options for the program. This code uses the built-in `argparse.ArgumentParser` class:

```
parser = argparse.ArgumentParser(description="descStr")
# add expected arguments
parser.add_argument('--file', dest='imgFile', required=True)
parser.add_argument('--scale', dest='scale', required=False)
parser.add_argument('--out', dest='outFile', required=False)
parser.add_argument('--cols', dest='cols', required=False)
parser.add_argument('--morelevels', dest='moreLevels',
  action='store_true')
```

You include the following options:

--file Specifies the image file to input. This is the only required argument.

--scale Sets the vertical scale factor for a font other than Courier.

--out Sets the output filename for the generated ASCII art. Defaults to *out.txt*.

--cols Sets the number of text columns in the ASCII output.

--morelevels Selects the 70-level grayscale ramp instead of the default 10-level ramp.

Writing the ASCII Art Strings to a Text File

Finally, take the generated list of ASCII character strings and write those strings to a text file:

```
# open a new text file
f = open(outFile, 'w')
# write each string in the list to the new file
for row in aimg:
    f.write(row + '\n')
# clean up
f.close()
```

You use the built-in `open()` function to open a new text file for writing ❶. Then you iterate through each string in the `aimg` list and write it to the file ❷. When you're done, you close the file object to release system resources ❸.

Running the ASCII Art Generator

To run your finished program, enter a command like the following one, replacing *data/robot.jpg* with the relative path

to the image file you want to use:

```
$ python ascii.py --file data/robot.jpg --cols 100
```

Figure 6-3 shows the ASCII art that results from sending the image *robot.jpg* (at the left). Try adding the `--morelevels` option to see how the 70-level grayscale ramp compares to the 10-level ramp.

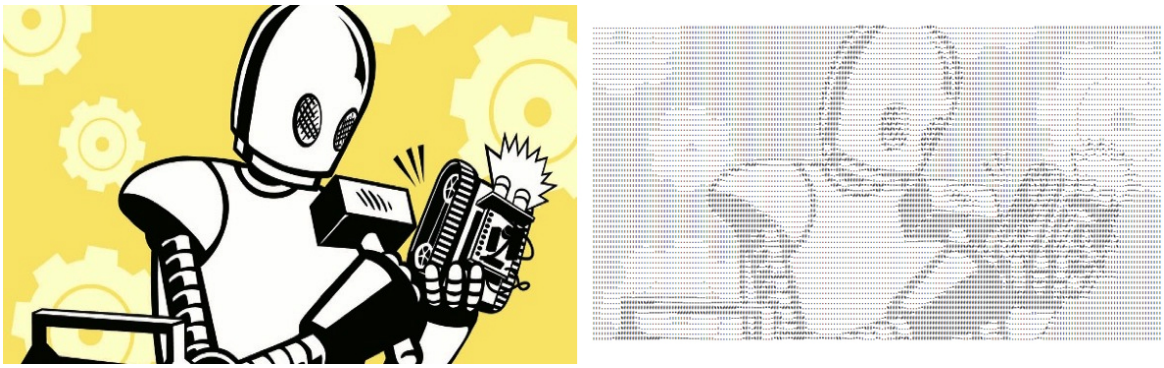


Figure 6-3: *A sample run of* ascii.py

Now you're all set to create your own ASCII art!

Summary

In this project, you learned how to generate ASCII art from any input image. In the process, you learned how to split an image into a grid of tiles, how to compute the average brightness value of each tile, and how to replace each tile with a character based on the brightness values. Have fun creating your own ASCII art!

Experiments!

Here are some ideas for exploring ASCII art further:

1. Run the program with the command line option `--scale 1.0`. How does the resulting image look? Experiment with different values for `scale`. Copy the output to a text editor and try setting the text to different fixed-width fonts to see how doing so affects the appearance of the final image.
2. Add a command line option `--invert` to the program to invert the generated ASCII images so that black appears white, and vice versa. (Hint: try subtracting the tile brightness value from 255 during lookup.)
3. In this project, you created lookup tables for grayscale values based on two hardcoded character ramps. Implement a command line option to pass in a different character ramp to create the ASCII art, like so:

```
$ python ascii.py --map "@$%^`."
```

   This should create the ASCII output using the given six-character ramp, where `@` maps to a brightness value of 0 and `.` maps to a value of 255.

The Complete Code

Here is the complete ASCII art program.

```
"""
ascii.py

A Python program that convert images to ASCII art.

Author: Mahesh Venkitachalam
"""

import sys, random, argparse
import numpy as np
```

```
import math

from PIL import Image

# grayscale level values from:
# http://paulbourke.net/dataformats/asciiart/

# 70 levels of gray
gscale1 = "$@B%8&WM#*oahkbdpqwmZO0QLCJUYXzcvunxrjft/\|()1{}[]?-
     _+~<>i!lI;:,\"^`'. "
# 10 levels of gray
gscale2 = '@%#*+=-:. '

def getAverageL(image):
    """
    given PIL Image, return average value of grayscale value
    """
    # get image as numpy array
    im = np.array(image)
    # get shape
    w,h = im.shape
    # get average
    return np.average(im.reshape(w*h))

def convertImageToAscii(fileName, cols, scale, moreLevels):
    """
    given Image and dims (rows, cols) returns an m*n list of
     Images
    """
    # declare globals
    global gscale1, gscale2
    # open image and convert to grayscale
    image = Image.open(fileName).convert('L')
    # store dimensions
    W, H = image.size[0], image.size[1]
    print("input image dims: {} x {}".format(W, H))
    # compute width of tile
    w = W/cols
    # compute tile height based on aspect ratio and scale
    h = w/scale
    # compute number of rows
    rows = int(H/h)
```

```
print("cols: {}, rows: {}".format(cols, rows))
print("tile dims: {} x {}".format(w, h))

# check if image size is too small
if cols > W or rows > H:
    print("Image too small for specified cols!")
    exit(0)

# an ASCII image is a list of character strings
aimg = []
# generate list of dimensions
for j in range(rows):
    y1 = int(j*h)
    y2 = int((j+1)*h)
    # correct last tile
    if j == rows-1:
        y2 = H
    # append an empty string
    aimg.append("")
    for i in range(cols):
        # crop image to tile
        x1 = int(i*w)
        x2 = int((i+1)*w)
        # correct last tile
        if i == cols-1:
            x2 = W
        # crop image to extract tile
        img = image.crop((x1, y1, x2, y2))
        # get average luminance
        avg = int(getAverageL(img))
        # look up ASCII char
        if moreLevels:
            gsval = gscale1[int((avg*69)/255)]
        else:
            gsval = gscale2[int((avg*9)/255)]
        # append ASCII char to string
        aimg[j] += gsval

# return image
return aimg
```

```
# main() function
def main():
    # create parser
    descStr = "This program converts an image into ASCII art."
    parser = argparse.ArgumentParser(description=descStr)
    # add expected arguments
    parser.add_argument('--file', dest='imgFile',
     required=True)
    parser.add_argument('--scale', dest='scale',
     required=False)
    parser.add_argument('--out', dest='outFile',
     required=False)
    parser.add_argument('--cols', dest='cols', required=False)
    parser.add_argument('--
     morelevels',dest='moreLevels',action='store_true')

    # parse args
    args = parser.parse_args()

    imgFile = args.imgFile
    # set output file
    outFile = 'out.txt'
    if args.outFile:
        outFile = args.outFile
    # set scale default as 0.43, which suits a Courier font
    scale = 0.43
    if args.scale:
        scale = float(args.scale)
    # set cols
    cols = 80
    if args.cols:
        cols = int(args.cols)

    print('generating ASCII art...')
    # convert image to ASCII text
    aimg = convertImageToAscii(imgFile, cols, scale,
     args.moreLevels)

    # open file
    f = open(outFile, 'w')
    # write to file
    for row in aimg:
```

```
        f.write(row + '\n')
    # clean up
    f.close()
    print("ASCII art written to {}.".format(outFile))

# call main
if __name__ == '__main__':
    main()
```

7
PHOTOMOSAICS



When I was in the sixth grade, I saw a picture like the one shown in Figure 7-1 but couldn't quite figure out what it was. After squinting at it for a while, I eventually figured it out. (Turn the book upside down, and view it from across the room. I won't tell anyone.)

The puzzle works because of how the human eye functions. The low-resolution, blocky image shown in the figure is hard to recognize up close, but when it is seen from a distance, you know what it represents because your eyes perceive less detail, which makes the edges smooth.

A *photomosaic* is an image that works according to a similar principle. You take a *target* image, split it into a grid of rectangles, and replace each rectangle with another, smaller image that matches that section of the target. When you look at a photomosaic from a distance, all you see is the target image, but if you come closer, the secret is revealed: the image actually consists of many tiny images!

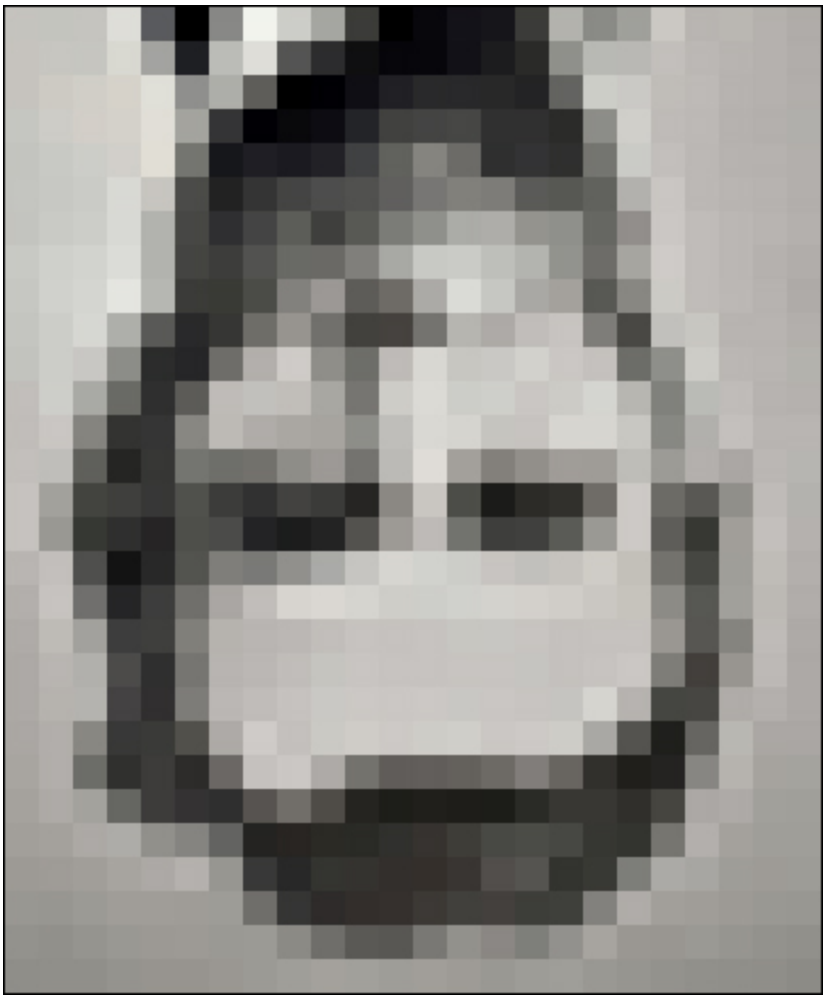


Figure 7-1: *A puzzling image*

In this project, you'll create a photomosaic using Python. You'll divide a target image into a grid and replace each block in the grid with a suitable image to create a photomosaic of the original. You'll be able to specify the grid dimensions and choose whether input images can be reused in the mosaic.

As you work on the project, you'll learn how to do the following:

- Create images using the Python Imaging Library (PIL).
- Compute the average RGB value of an image.

- Crop images.
- Replace part of an image by pasting in another image.
- Compare RGB values using a measurement of average distance in three dimensions.
- Use a data structure called a *k-d tree* to efficiently find the image that best matches a section of the target image.

How It Works

To create a photomosaic, begin with a blocky, low-resolution version of the target image (because the number of tile images would be too great in a high-resolution image). The user inputs the dimensions $M \times N$ (where M is the number of rows and N is the number of columns) of the mosaic. Next, build the mosaic according to this methodology:

1. Read the input images, which will be drawn on to replace the tiles in the original image.
2. Read the target image and split it into an $M \times N$ grid of tiles.
3. For each tile, find the best match from the input images.
4. Create the final mosaic by arranging the selected input images in an $M \times N$ grid.

Splitting the Target Image

We'll start by looking at how to split the target image into an $M \times N$ grid of tiles. Follow the scheme shown in Figure 7-2.

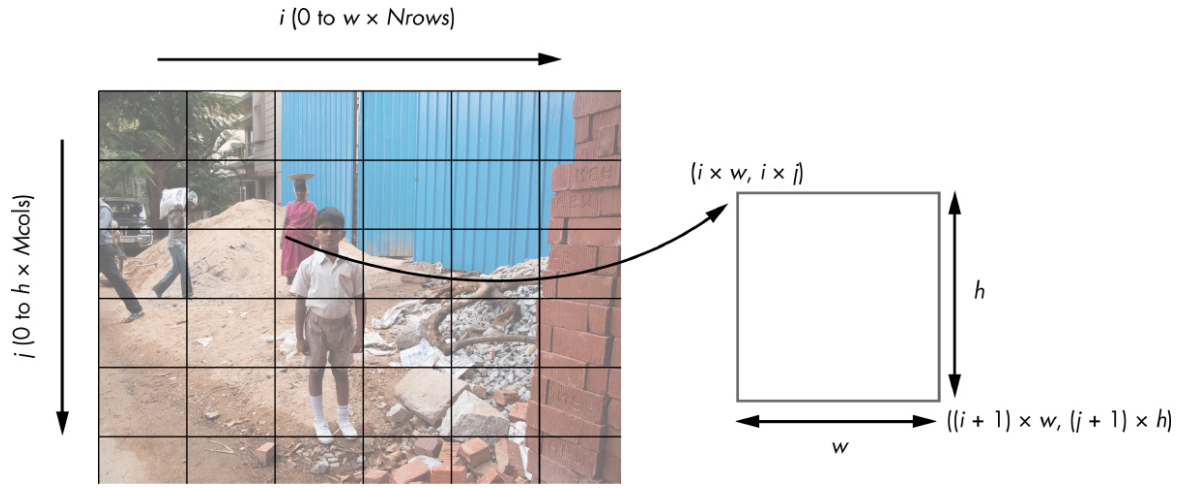


Figure 7-2: *Splitting the target image*

We split the original image into a grid of tiles with N columns arranged along the x-axis and M rows arranged along the y-axis. Each tile is represented by an index (i, j) and is w pixels wide and h pixels high. According to this scheme, the original image is $w \times N$ pixels wide and $h \times M$ pixels high.

The right side of Figure 7-2 shows how to calculate the pixel coordinates for a single tile from this grid. The tile with index (i, j) has a top-left corner coordinate of ($i \times w$, $i \times j$) and a bottom-right corner coordinate of (($i + 1$) × w, ($j + 1$) × h). These coordinates can be used with the PIL to crop and create tiles from the original image.

Averaging Color Values

Every pixel in an image has a color that can be represented numerically by its red, green, and blue values. In this case, you are using 8-bit images, so each of these three color components has an 8-bit value in the range [0, 255]. You can therefore determine the average color of an image by taking the average of the red, green, and blue values for all of the image's pixels. Given an image with a total of N pixels, the average RGB is calculated as follows:

$$(r,g,b)_{\text{avg}} = \left(\frac{r_1 + r_2 + \ldots + r_N}{N} , \frac{g_1 + g_2 + \ldots + g_N}{N} , \frac{b_1 + b_2 + \ldots + b_N}{N} \right)$$

Like the RGB for an individual pixel, the average RGB for a whole image is a triplet, not a scalar or single number, because the averages are calculated separately for each color component. You calculate the average RGB to match the tiles from the target image with replacements from among the input images.

Matching Images

For each tile in the target image, you need to find a matching image from the images in the input folder specified by the user. To determine whether two images match, use the average RGB values. The best match is the input image with the average RGB value closest to that of the tile from the target image.

The simplest way to find the best match is to calculate the distance between the average RGB values as if they were points in 3D space. After all, each average RGB consists of three numbers, which you can think of as x-, y-, and z-axis coordinates. You can thus use the following formula from the geometry for calculating the distance between two 3D points:

$$D_{1,2} = \sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2}$$

Here you compute the distance between the points (r_1, g_1, b_1) and (r_2, g_2, b_2). Given a target average RGB value (r_1, g_1, b_1), you can plug a list of average RGB values from the input images into the previous formula as (r_2, g_2, b_2) to find

the closest matching image. However, there might be hundreds or even thousands of input images to check. We should therefore give some thought to how to efficiently search the set of input images to find the best match.

Using Linear Search

The simplest approach to searching for a match is a *linear search*. In this method, you just iterate through all the RGB values one by one and find the one with the minimum distance to the query value. The code will look something like this:

```
min_dist = MAX_VAL
for val in vals:
    dist = distance(query, val)
    if dist < MAX_VAL:
        min_dist = dist
```

You go through each value in the list `vals` one by one and calculate the distance between that value and `query`. If the result is less than `min_dist` (which was initialized as the maximum possible distance between two points), you update `min_dist` with the distance you just calculated. After checking every item in `vals`, `min_dist` will contain the smallest distance in the whole dataset.

Although a linear search method is easy to understand and implement, it isn't very efficient. If there are *N* values in the `vals` list, the search will take an amount of time proportional to *N*. You can achieve much better performance with a different data structure and search algorithm.

Using k-d Trees

A *k-d tree*, or *k-dimensional tree*, is a data structure that partitions a space of *k* dimensions—that is, it divides the space into a number of non-overlapping subspaces. This data structure provides a way to sort and search through datasets whose members are points in *k*-dimensional space. The dataset is represented as a *binary tree*: each point in the dataset becomes a node in the tree, and each node can have two child nodes. In other words, each node in the tree divides the space into two parts, called *subtrees*. One part points to the left of the node (the node's left child and its descendants), and the other points to the right of the node (the node's right child and its descendants).

Each node of the tree is associated with one of the dimensions of the space, and that's the dimension used to determine if points belong in the node's left subtree or right subtree. If a node is associated with the x-axis, for example, points whose x-values are less than that node's x-value will be put in the node's left subtree, and points whose x-values are greater than the node's x-value will be put in the right subtree. A common method to select the dimension associated with each node is to cycle through them as you move down the levels of the tree. For example, in the case of a three-dimensional k-d tree, you could set the dimensions to be x, y, z, x, y, z, and so on, moving down the tree. Nodes at the same tree height will have the same splitting dimension.

Let's look at a simple example of a k-d tree. Say you have the following set of points, *P*:

$$P = \{(5, 3), (2, 4), (1, 2), (6, 6), (7, 2), (4, 6), (2, 8)\}$$

In this case, you'd build a two-dimensional k-d tree, since each member of *P* describes a point in two-dimensional space. You start by associating the first node, or the *root* node, (5, 3), with the x-dimension. Then you add the next

point, (2, 4), as a left child of the root node, since the point's x-coordinate, 2, is less than 5, the x-coordinate of the root. The node (2, 4), being on the second level of the k-d tree, will use the y-dimension for partition. The next point in the list is (1, 2). Starting again at the root, 1 < 5, so you go to the left child of the root node. You then compare (1, 2) with (2, 4) using the y-dimension. Since 2 < 4, you add (1, 2) as the left child of (2, 4).

If you continue in this fashion for all the points in *P*, you'll create the tree and space partitioning shown in Figure 7-3.

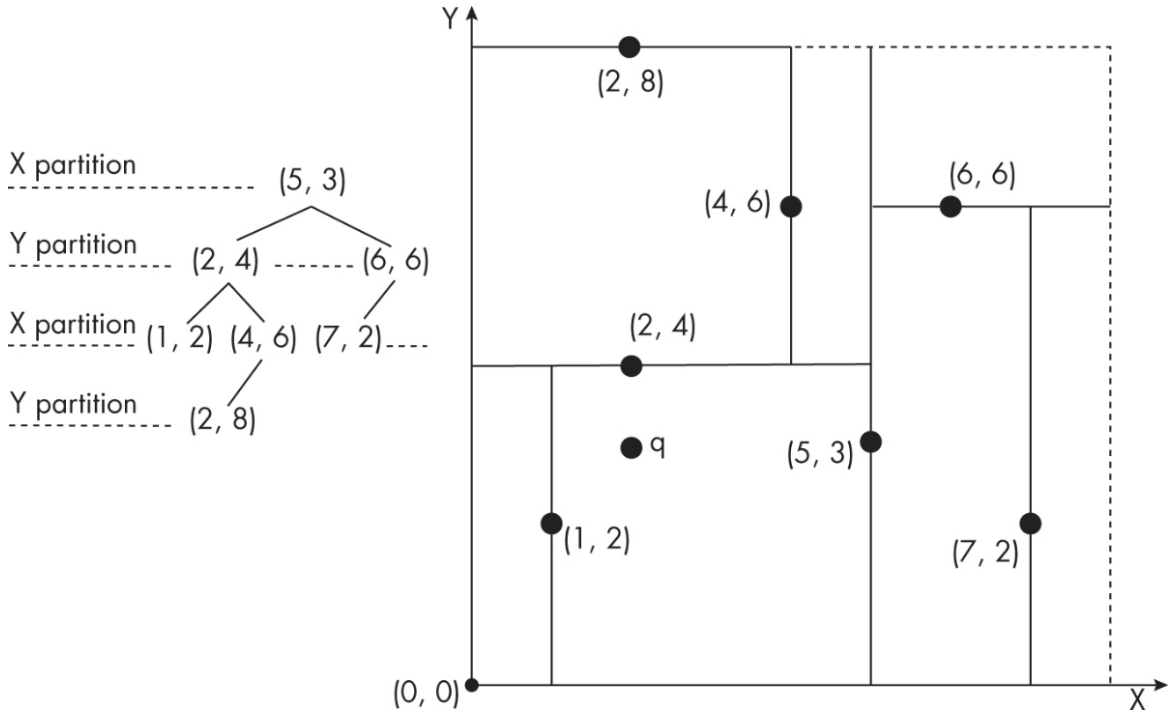


Figure 7-3: *An example of a k-d tree*

The top image of Figure 7-3 shows the space partitioning scheme for the tree we just discussed. Starting with point (5, 3), you split the space in two along the x-dimension by drawing a vertical line through that point. Next, you use point (2, 4) to split the left half of the first partition along the y-dimension by drawing a horizontal line through the point, stopping when the line hits the vertical line. Continue in this

fashion with the remaining points, and you'll get the partitioning scheme shown in the figure.

Why should you care about k-d trees? The answer is that once you arrange a dataset this way, you can search through it much more quickly. Specifically, a *nearest-neighbor search*—finding the point closest to a queried point—is much faster with a k-d tree than a linear search. For a dataset of N values, the average nearest-neighbor search of a k-d tree takes a time proportional to $\log(N)$, much less than the time proportional to N that a linear search would take.

To demonstrate, let's try to find the point from P nearest to point q, (2, 3), which is shown in Figure 7-3. Looking at the figure, you can see that point (2, 4) is the match. The nearest-neighbor algorithm will find the match by traversing down the tree from (5, 3) to (2, 4). The algorithm knows, for example, that the right subtree of the root can be skipped, since q's x-coordinate is less than the root node's x-coordinate. The spatial partitioning scheme thus lets you skip a larger number of comparisons than with a linear search. This is what makes the k-d tree useful for our problem.

How can you use a k-d tree in the photomosaic code? You could try to write an implementation from scratch, but there's an easier option: the `scipy` library already has a built-in k-d tree class. We'll look at how to leverage this class later in the chapter.

Requirements

For this project, you'll use `Pillow` to read in the images, access their underlying data, and create and modify the images. You'll also use `numpy` to manipulate image data and `scipy` to search the image data using a k-d tree.

The Code

You'll begin by reading in the input images that you'll draw on to create the photomosaic. Next, you'll compute the average RGB value of the images, split the target into a grid, and find the image that best matches each tile in the grid. Finally, you'll assemble the image tiles to create the actual photomosaic. To see the complete project code, skip ahead to "The Complete Code" on page 129. You can also find the code at *https://github.com/mkvenkit/pp2e/tree/main/photomosaic*.

Reading In the Input Images

First read in the input images from a given folder. Here's how to do that:

```
def getImages(imageDir):
    """
    given a directory of images, return a list of Images
    """
  ❶ files = os.listdir(imageDir)
    images = []
    for file in files:
      ❷ filePath = os.path.abspath(os.path.join(imageDir,
  file))
        try:
            # explicit load so we don't run into resource
  crunch
          ❸ fp = open(filePath, "rb")
            im = Image.open(fp)
            images.append(im)
            # force loading the image data from file
          ❹ im.load()
            # close the file
          ❺ fp.close()
        except:
```

```
            # skip
            print("Invalid image: %s" % (filePath,))
    return images
```

You first use `os.listdir()` to gather the filenames in the *imageDir* directory in a list called `files` ❶. Next, you iterate through each file in the list and load it into a PIL `Image` object.

You use `os.path.abspath()` and `os.path.join()` to get the complete filename of the image ❷. This idiom is commonly used in Python to ensure that your code will work with both relative paths (for example, *\foo\bar*) and absolute paths (*c:\foo\bar*), as well as across operating systems with different directory-naming conventions (\ in Windows versus / in Linux).

To load the files into PIL `Image` objects, you could pass each filename to the `Image.open()` method, but if your photomosaic folder had hundreds or thousands of images, doing so would be highly resource intensive. Instead, you can use Python to open each image and pass the file handle `fp` into PIL using `Image.open()`. Once the image has been loaded, close the file handle and release the system resources.

You open the image file using `open()` ❸ and then pass the handle to `Image.open()` and store the resulting image, `im`, in a list called `images`. Calling `Image.load()` ❹ force-loads the image data inside `im` because `open()` is a lazy operation. It identifies the image but doesn't actually read all the image data until you try to use the image. You finish by closing the file handle to release system resources ❺.

Calculating the Average Color Value of an Image

Once you've read in the input images, you need to calculate each image's average color value. You also need to find the

average color value for each section of the target image. Create a function `getAverageRGB()` to handle both tasks.

```
def getAverageRGB(image):
    """
    return the average color value as (r, g, b) for each input
  image
    """
    # get each tile image as a numpy array
  ❶ im = np.array(image)
    # get the shape of each input image
  ❷ w,h,d = im.shape
    # get the average RGB value
  ❸ return tuple(np.average(im.reshape(w*h, d), axis=0))
```

The function takes in an `Image` object—it could be one of the input images or a section of the target image—and uses `numpy` to convert it into a data array ❶. The resulting `numpy` array has the shape (`w`, `h`, `d`), where `w` is the width of the image, `h` is the height, and `d` is the depth, which, in the case of RGB images, is three units (one each for R, G, and B). You store the `shape` tuple ❷ and then compute the average RGB value by reshaping the array into a more convenient form with shape (`w*h`, `d`) so that you can compute the average using `numpy.average()` ❸. (You performed a similar operation in Chapter 6 to get the average brightness of a grayscale image.) You return the result as a tuple.

Splitting the Target Image into a Grid

Now you need to split the target image into an $M \times N$ grid of smaller images. Let's create a function to do that:

```
def splitImage(image, size):
    """
```

```
    given the image and dimensions (rows, cols), return an m*n
  list of images
    """
  ❶ W, H = image.size[0], image.size[1]
  ❷ m, n = size
  ❸ w, h = int(W/n), int(H/m)
    # image list
    imgs = []
    # generate a list of images
    for j in range(m):
        for i in range(n):
            # append cropped image
      ❹ imgs.append(image.crop((i*w, j*h, (i+1)*w, (j+1)*h)))
    return imgs
```

First you gather the dimensions of the target image ❶ and the grid size ❷. Then you calculate the dimensions of each tile in the target image using basic division ❸. Next you need to iterate through the grid dimensions and cut out and store each tile as a separate image. Calling `image.crop()` ❹ crops out a portion of the image using the upper-left and lower-right image coordinates as arguments (as discussed in “Splitting the Target Image” on page 115). You end up with a list of images—first, all the images in the first row of the grid, from left to right; then all the images in the second row of the grid; and so on.

Finding the Best Match for a Tile

Now let’s find the best match for a tile from the folder of input images. We’ll look at two ways of doing this: using a linear search and using a k-d tree. For the linear search method, you create a utility function, `getBestMatchIndex()`, as follows:

```
def getBestMatchIndex(input_avg, avgs):
```

```
    """
    return index of the best image match based on average RGB
  value distance
    """

    # input image average
    avg = input_avg

    # get the closest RGB value to input, based on RGB distance
    index = 0
  ❶ min_index = 0
  ❷ min_dist = float("inf")
  ❸ for val in avgs:
      ❹ dist = ((val[0] - avg[0])*(val[0] - avg[0]) +
                (val[1] - avg[1])*(val[1] - avg[1]) +
                (val[2] - avg[2])*(val[2] - avg[2]))
      ❺ if dist < min_dist:
            min_dist = dist
            min_index = index
        index += 1

    return min_index
```

You're trying to search `avgs`, a list of the average RGB values of the input images, to find the one closest to `input_avg`, the average RGB value of one of the tiles in the target image. To start, you initialize the closest match index to 0 ❶ and the minimum distance to infinity ❷. Then you loop through the values in the list of averages ❸ and start computing distances ❹ using the standard formula shown in "Matching Images" on page 116. (You skip taking the square root to reduce computation time.) If the computed distance is less than the stored minimum distance `min_dist`, it's replaced with the new minimum distance ❺. This test will always pass the first time, since any distance will be less than infinity. At the end of the iteration, `min_index` is the index of the average RGB value from the `avgs` list that is closest to `input_avg`. Now you can use this index to select the matching image from the list of input images.

Now let's find the best matches using a k-d tree instead of a linear search. Here's the function:

```
def getBestMatchIndicesKDT(qavgs, kdtree):
    """
    return indices of best Image matches based on RGB value
  distance
    uses a k-d tree
    """
    # e.g., [array([2.]), array([9], dtype=int64)]
  ❶ res = list(kdtree.query(qavgs, k=1))
  ❷ min_indices = res[1]
    return min_indices
```

The `getBestMatchIndicesKDT()` function takes two arguments: `qavgs` is the list of average RGB values for each tile in the target image, and `kdtree` is the `scipy KDTree` object created using a list of average RGB values from the input images. (We'll be creating the `KDTree` object in "Creating the Photomosaic" on page 124.) You use the `KDTree` object's `query()` method to get the points in the tree that are closest to the ones in `qavgs` ❶. Here, the `k` parameter is the number of nearest neighbors to the queried point you want to return. You just need the closest match, so you pass in `k=1`. The return value from the `query()` method is a tuple consisting of two `numpy` arrays with the distances and indices of the matches. You need the indices, so you pick the second value from the result ❷.

Notice that the `query()` method ❶ allows you to pass in a list of query points instead of just one. This actually runs faster than querying results one by one, and it means you'll have to call the `getBestMatchIndicesKDT()` function only once, whereas you'll have to call the linear search `getBestMatch()` function many times, once for each tile in the photomosaic.

The complete program will include an option to choose which of the previous two functions to use, the linear search version or the k-d tree version. It will also have a timer to test which search method is faster.

Creating an Image Grid

You need one more utility function before moving on to photomosaic creation. The `createImageGrid()` function will create a grid of images of size $M \times N$. This image grid is the final photomosaic image, created from the list of selected input images.

```
def createImageGrid(images, dims):
    """
    given a list of images and a grid size (m, n), create a
  grid of images
    """
  ❶ m, n = dims

    # sanity check
    assert m*n == len(images)

    # get the maximum height and width of the images
    # don't assume they're all equal
  ❷ width = max([img.size[0] for img in images])
    height = max([img.size[1] for img in images])

    # create the target image
  ❸ grid_img = Image.new('RGB', (n*width, m*height))

    # paste the tile images into the image grid
    for index in range(len(images)):
      ❹ row = int(index/n)
      ❺ col = index - n*row
      ❻ grid_img.paste(images[index], (col*width, row*height))
    return grid_img
```

The function takes two parameters: a list of images (the input images you chose based on the closest RGB match to the individual tiles of the target image) and a tuple with the photomosaic's dimensions (the number of rows and columns you want it to have). You gather the dimensions of the grid ❶ and then use `assert` to see whether the number of images supplied to `createImageGrid()` matches the grid size. (The `assert` method checks assumptions in your code, especially during development and testing.) Then you compute the maximum width and height of the selected images ❷, since they may not all be the same size. You'll use these maximum dimensions to set the standard tile size for the photomosaic. If an input image won't completely fill a tile, the spaces between the tiles will show as solid black by default.

Next, you create an empty `Image` sized to fit all images in the grid ❸; you'll paste the tile images into this. Then you fill the image grid by looping through the selected images and pasting them into the appropriate spot on the grid using the `Image.paste()` method ❻ . The first argument to `Image.paste()` is the `Image` object to be pasted, and the second is the top-left coordinate. Now you need to figure out in which row and column to paste an input image into the image grid. To do so, you express the image index in terms of rows and columns. The index of a tile in the image grid is given by $N \times row + col$, where N is the number of cells along the width and (row, col) is the coordinate in the grid; at ❹, you determine the row from the previous formula, and at ❺, the column.

Creating the Photomosaic

Now that you have all the required utilities, let's write the main function that creates the photomosaic. Here's the start of the function:

```
def createPhotomosaic(target_image, input_images, grid_size,
                      reuse_images, use_kdt):
    """
    creates photomosaic given target and input images
    """

    print('splitting input image...')
    # split target image
  ❶ target_images = splitImage(target_image, grid_size)

    print('finding image matches...')
    # for each target image, pick one from input
    output_images = []
    # for user feedback
    count = 0
  ❷ batch_size = int(len(target_images)/10)

    # calculate input image averages
    avgs = []
  ❸ for img in input_images:
        avgs.append(getAverageRGB(img))

    # compute target averages
    avgs_target = []
  ❹ for img in target_images:
        # target subimage average
        avgs_target.append(getAverageRGB(img))
```

The `createPhotomosaic()` function takes as input the target image, the list of input images, the size of the generated photomosaic (number of rows and columns), and flags indicating whether an image can be reused and whether to use a k-d tree to search for image matches. The function begins by calling `splitImage()` ❶ to split the target into a grid of smaller image tiles. Once the image is split, you’re ready to start finding matches for each tile from the images in the input folder. Because this process can be lengthy, however, it’s a good idea to provide feedback to users to let them know that the program is still working. To help with this

feedback, you set `batch_size` to one-tenth the total number of tile images ❷. The choice of one-tenth is arbitrary and simply a way for the program to say "I'm still alive." Each time the program processes a tenth of the images, it will print a message indicating that it's still running.

To find image matches, you need the average RGB values. You iterate over the input images ❸ and use your `getAverageRGB()` function to compute the average RGB value for each one, storing the results in the list `avgs`. Then you do the same for each tile in the target image ❹, storing the average RGB values into the list `avgs_target`.

The function continues with an `if...else` statement to find RGB matches using either a k-d tree or a linear search. Let's look at the `if` branch first, which runs if the `use_kdt` flag was set to `True`:

```
    # use k-d tree for average match?
    if use_kdt:
        # create k-d tree
      ❶ kdtree = KDTree(avgs)
        # query k-d tree
      ❷ match_indices = getBestMatchIndicesKDT(avgs_target,
  kdtree)
        # process matches
      ❸ for match_index in match_indices:
          ❹ output_images.append(input_images[match_index])
```

You create a `KDTree` object using the list of average RGB values from the input images ❶ and retrieve the indices of the best matches by passing in `avgs_target` and the `KDTree` object to your `getBestMatchIndicesKDT()` helper function ❷. Then you iterate through all the matching indices ❸, find the corresponding input images, and append them to the list `output_images` ❹.

Next, let's look at the `else` branch, which performs a linear search for matches:

```
    else:
        # use linear search
      ❶ for avg in avgs_target:
            # find match index
          ❷ match_index = getBestMatchIndex(avg, avgs)
          ❸ output_images.append(input_images[match_index])
            # user feedback
          ❹ if count > 0 and batch_size > 10 and count %
  batch_size == 0:
                print('processed %d of %d...' %(count,
  len(target_images)))
            count += 1
            # remove selected image from input if flag set
          ❺ if not reuse_images:
                input_images.remove(match)
```

For the linear search, you start iterating through the average RGB values of the target image tiles ❶. For each tile, you search for the closest match in the list of averages for the input images using `getBestMatchIndex()` ❷. The result is returned as an index, which you use to retrieve the `Image` object and store it in the `output_images` list ❸. For every `batch_size` number of images processed ❹, you print a message to the user. If the `reuse_images` flag is set to `False` ❺, you remove the selected input image from the list so that it won't be reused in another tile. (This works best when you have a wide range of input images to choose from.)

All that remains in the `createPhotomosaic()` function is to arrange the input images into the final photomosaic:

```
    print('creating mosaic...')
    # draw mosaic to image
  ❶ mosaic_image = createImageGrid(output_images, grid_size)
```

```
    # return mosaic
    return mosaic_image
```

You use the `createImageGrid()` function to build the photomosaic ❶. Then you return the resulting image as `mosaic_image`.

Writing the main() Function

The `main()` function of the program takes in and parses command line arguments, loads all the images, and does some additional setup. Then it calls the `createPhotomosaic()` function and saves the resulting photomosaic. As the photomosaic is built, Python times how long the process takes, allowing you to compare the performance of the k-d tree with that of the linear search.

Adding the Command Line Options

The `main()` function supports these command line options:

```
    # parse arguments
    parser = argparse.ArgumentParser(description='Creates a
photomosaic from
                                     input images')
    # add arguments
    parser.add_argument('--target-image', dest='target_image',
required=True)
    parser.add_argument('--input-folder', dest='input_folder',
required=True)
    parser.add_argument('--grid-size', nargs=2,
dest='grid_size',
                        required=True)
    parser.add_argument('--output-file', dest='outfile',
required=False)
    parser.add_argument('--kdt', action='store_true',
required=False)
```

This code contains three required command line parameters: the name of the target image, the name of the input folder of images, and the grid size. The fourth parameter is for the optional filename for the output. If the filename is omitted, the photomosaic will be written to a file named *mosaic.png*. The fifth argument is a Boolean flag that enables the k-d tree search instead of linear search for matching average RGB values.

Controlling the Size of the Photomosaic

Once all the images are loaded, one issue to address in the `main()` function is the size (in pixels) of the resulting photomosaic. If you were to blindly paste the input images together based on matching tiles in the target, you could end up with a huge photomosaic that is much bigger than the target. To avoid this, resize the input images to match the size of each tile in the grid. (This has the added benefit of speeding up the average RGB computation since you'll be using smaller images.) Here's the section of the `main()` function that handles this task:

```
        print('resizing images...')
        # for given grid size, compute the maximum width and
 height of tiles
      ❶ dims = (int(target_image.size[0]/grid_size[1]),
                int(target_image.size[1]/grid_size[0]))
        print("max tile dims: %s" % (dims,))
        # resize
        for img in input_images:
          ❷ img.thumbnail(dims)
```

You compute the target dimensions based on the specified number of rows and columns in the grid ❶; then you use the

PIL `Image.thumbnail()` method to resize the input images to fit those dimensions ❷.

Timing the Performance

When the program is run, you'll want to know how long it takes to execute. Use the Python `timeit` module for this purpose. The approach for finding execution time is outlined here:

```
import timeit
# start timing
start = timeit.default_timer()
# run some code here...
--snip--
# stop timing
stop = timeit.default_timer()
print('Execution time: %f seconds' % (stop - start, ))
```

You record the start time using the `timeit` module's default timer ❶. Then, after running some code, you record the stop time ❷. Computing the difference gives you the execution time measured in seconds.

Running the Photomosaic Generator

Let's first run the program using the default linear search approach. The photomosaic will consist of a grid of 128×128 images:

```
$ python photomosaic.py --target-image test-data/cherai.jpg --
  input-folder
  test-data/set6/ --grid-size 128 128
reading input folder...
```

```
starting photomosaic creation...
resizing images...
max tile dims: (23, 15)
splitting input image...
finding image matches...
processed 1638 of 16384...
processed 3276 of 16384...
processed 4914 of 16384...
processed 6552 of 16384...
processed 8190 of 16384...
processed 9828 of 16384...
processed 11466 of 16384...
processed 13104 of 16384...
processed 14742 of 16384...
processed 16380 of 16384...
creating mosaic...
saved output to mosaic.png
done.
Execution time:    setup: 0.402047 seconds
Execution time: creation: 2.123931 seconds
Execution time:    total: 2.525978 seconds
```

Figure 7-4(a) shows the target image, and Figure 7-4(b) shows the resulting photomosaic. You can see a close-up of the photomosaic in Figure 7-4(c). As you can see in the output, it takes about 2.1 seconds ❶ to find the best match for each of the 16,384 tiles in the photomosaic using a linear search. That's not bad, but we can do better.

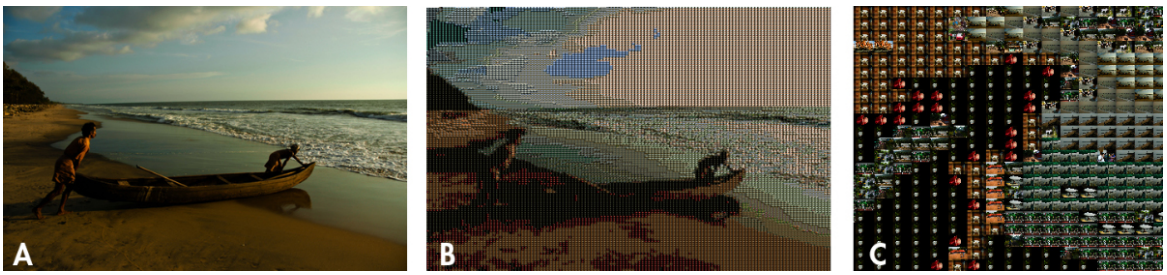


Figure 7-4: *A sample run of the photomosaic generator*

Now run the same program with the `--kdt` option, which enables the use of a k-d tree to search for image matches. Here are the results:

```
$ python photomosaic.py --target-image test-data/cherai.jpg --
  input-folder
  test-data/set6/ --grid-size 128 128 --kdt
reading input folder...
starting photomosaic creation...
resizing images...
max tile dims: (23, 15)
splitting input image...
finding image matches...
creating mosaic...
saved output to mosaic.png
done.
Execution time:    setup: 0.410334 seconds
Execution time: creation: 1.089237 seconds
Execution time:    total: 1.499571 seconds
```

The photomosaic creation time has dropped from about 2.1 seconds to less than 1.1 seconds ❶ with a k-d tree. That's almost twice the speed!

Summary

In this project, you learned how to create a photomosaic, given a target image and a collection of input images. When viewed from a distance, the photomosaic looks like the original image, but up close, you can see the individual images that make up the mosaic. You also learned about an interesting data structure, the k-d tree, which significantly sped up the process of finding the closest match for each tile in the mosaic.

Experiments!

Here are some ways to further explore photomosaics:

1. Write a program that creates a blocky version of any image, similar to Figure 7-1.
2. With the code in this chapter, you created the photomosaic by pasting the matched images without any gaps in between. A more artistic presentation might include a uniform gap of a few pixels around each tile image. How would you create the gap? (Hint: factor in the gaps when computing the final image dimensions and when pasting the images in `createImageGrid()`.)

The Complete Code

Here's the complete code for the project:

```
"""
photomosaic.py

Creates a photomosaic given a target image and a folder of
  input images.

Author: Mahesh Venkitachalam
"""

import os, random, argparse
from PIL import Image
import numpy as np
from scipy.spatial import KDTree
import timeit

def getAverageRGBOld(image):
    """
    given PIL Image, return average value of color as (r, g, b)
```

```
    """
    # no. of pixels in image
    npixels = image.size[0]*image.size[1]
    # get colors as [(cnt1, (r1, g1, b1)), ...]
    cols = image.getcolors(npixels)
    # get [(c1*r1, c1*g1, c1*g2), ...]
    sumRGB = [(x[0]*x[1][0], x[0]*x[1][1], x[0]*x[1][2]) for x
  in cols]
    # calculate (sum(ci*ri)/np, sum(ci*gi)/np, sum(ci*bi)/np)
    # the zip gives us [(c1*r1, c2*r2, ...), (c1*g1,
  c1*g2, ...), ...]
    avg = tuple([int(sum(x)/npixels) for x in zip(*sumRGB)])
    return avg

def getAverageRGB(image):
    """
    given PIL Image, return average value of color as (r, g, b)
    """
    # get image as numpy array
    im = np.array(image)
    # get shape
    w,h,d = im.shape
    # get average
    return tuple(np.average(im.reshape(w*h, d), axis=0))

def splitImage(image, size):
    """
    given Image and dims (rows, cols) returns an m*n list of
  Images
    """
    W, H = image.size[0], image.size[1]
    m, n = size
    w, h = int(W/n), int(H/m)
    # image list
    imgs = []
    # generate list of images
    for j in range(m):
        for i in range(n):
            # append cropped image
```

```
            imgs.append(image.crop((i*w, j*h, (i+1)*w,
  (j+1)*h)))
    return imgs

def getImages(imageDir):
    """
    given a directory of images, return a list of Images
    """
    files = os.listdir(imageDir)
    images = []
    for file in files:
        filePath = os.path.abspath(os.path.join(imageDir,
  file))
        try:
            # explicit load so we don't run into resource
  crunch
            fp = open(filePath, "rb")
            im = Image.open(fp)
            images.append(im)
            # force loading image data from file
            im.load()
            # close the file
            fp.close()
        except:
            # skip
            print("Invalid image: %s" % (filePath,))
    return images

def getBestMatchIndex(input_avg, avgs):
    """
    return index of best Image match based on RGB value
  distance
    """

    # input image average
    avg = input_avg

    # get the closest RGB value to input, based on x/y/z
  distance
    index = 0
    min_index = 0
    min_dist = float("inf")
```

```
    for val in avgs:
        dist = ((val[0] - avg[0])*(val[0] - avg[0]) +
                (val[1] - avg[1])*(val[1] - avg[1]) +
                (val[2] - avg[2])*(val[2] - avg[2]))
        if dist < min_dist:
            min_dist = dist
            min_index = index
        index += 1

    return min_index

def getBestMatchIndicesKDT(qavgs, kdtree):
    """
    return indices of best Image matches based on RGB value
  distance
    using a k-d tree
    """
    # e.g., [array([2.]), array([9], dtype=int64)]
    res = list(kdtree.query(qavgs, k=1))
    min_indices = res[1]
    return min_indices

def createImageGrid(images, dims):
    """
    given a list of images and a grid size (m, n), create
    a grid of images
    """
    m, n = dims

    # sanity check
    assert m*n == len(images)

    # get max height and width of images
    # i.e., not assuming they are all equal
    width = max([img.size[0] for img in images])
    height = max([img.size[1] for img in images])

    # create output image
    grid_img = Image.new('RGB', (n*width, m*height))

    # paste images
    for index in range(len(images)):
        row = int(index/n)
```

```
        col = index - n*row
        grid_img.paste(images[index], (col*width, row*height))
    return grid_img

def createPhotomosaic(target_image, input_images, grid_size,
                      reuse_images, use_kdt):
    """
    creates photomosaic given target and input images
    """
    print('splitting input image...')
    # split target image
    target_images = splitImage(target_image, grid_size)
    print('finding image matches...')
    # for each target image, pick one from input
    output_images = []
    # for user feedback
    count = 0
    batch_size = int(len(target_images)/10)
    # calculate input image averages
    avgs = []
    for img in input_images:
        avgs.append(getAverageRGB(img))
    # compute target averages
    avgs_target = []
    for img in target_images:
        # target subimage average
        avgs_target.append(getAverageRGB(img))
    # use k-d tree for average match?
    if use_kdt:
        # create k-d tree
        kdtree = KDTree(avgs)
        # query k-d tree
        match_indices = getBestMatchIndicesKDT(avgs_target,
  kdtree)
        # process matches
        for match_index in match_indices:
            output_images.append(input_images[match_index])
```

```
    else:
        # use linear search
        for avg in avgs_target:
            # find match index
            match_index = getBestMatchIndex(avg, avgs)
            output_images.append(input_images[match_index])
            # user feedback
            if count > 0 and batch_size > 10 and count %
  batch_size == 0:
                print('processed {} of {}...'.format(count,
                                                     len(target
  _images)))
            count += 1
            # remove selected image from input if flag set
            if not reuse_images:
                input_images.remove(match)

    print('creating mosaic...')
    # draw mosaic to image
    mosaic_image = createImageGrid(output_images, grid_size)

    # return mosaic
    return mosaic_image

# gather our code in a main() function
def main():
    # command line args are in sys.argv[1], sys.argv[2]...
    # sys.argv[0] is the script name itself and can be ignored

    # parse arguments
    parser = argparse.ArgumentParser(description='Creates a
  photomosaic
                                                      from
  input images')
    # add arguments
    parser.add_argument('--target-image', dest='target_image',
  required=True)
    parser.add_argument('--input-folder', dest='input_folder',
  required=True)
    parser.add_argument('--grid-size', nargs=2,
  dest='grid_size',
                        required=True)
```

```
  parser.add_argument('--output-file', dest='outfile',
required=False)
  parser.add_argument('--kdt', action='store_true',
required=False)

  args = parser.parse_args()

  # start timing
  start = timeit.default_timer()

  ###### INPUTS ######

  # target image
  target_image = Image.open(args.target_image)

  # input images
  print('reading input folder...')
  input_images = getImages(args.input_folder)

  # check if any valid input images found
  if input_images == []:
      print('No input images found in %s. Exiting.' %
(args.input_folder, ))
      exit()

  # shuffle list - to get a more varied output?
  random.shuffle(input_images)

  # size of grid
  grid_size = (int(args.grid_size[0]),
int(args.grid_size[1]))

  # output
  output_filename = 'mosaic.png'
  if args.outfile:
      output_filename = args.outfile

  # reuse any image in input
  reuse_images = True

  # resize the input to fit original image size?
  resize_input = True

  # use k-d trees for matching
  use_kdt = False
  if args.kdt:
```

```
        use_kdt = True

    ##### END INPUTS #####

    print('starting photomosaic creation...')

    # if images can't be reused, ensure m*n <= num_of_images
    if not reuse_images:
        if grid_size[0]*grid_size[1] > len(input_images):
            print('grid size less than number of images')
            exit()

    # resizing input
    if resize_input:
        print('resizing images...')
        # for given grid size, compute max dims w,h of tiles
        dims = (int(target_image.size[0]/grid_size[1]),
                int(target_image.size[1]/grid_size[0]))
        print("max tile dims: %s" % (dims,))
        # resize
        for img in input_images:
            img.thumbnail(dims)

    # setup time
    t1 = timeit.default_timer()

    # create photomosaic
    mosaic_image = createPhotomosaic(target_image,
input_images, grid_size,
                                     reuse_images, use_kdt)

    # write out mosaic
    mosaic_image.save(output_filename, 'PNG')

    print("saved output to %s" % (output_filename,))
    print('done.')

    # creation time
    t2 = timeit.default_timer()

    print('Execution time:    setup: %f seconds' % (t1 -
start, ))
    print('Execution time: creation: %f seconds' % (t2 - t1, ))
    print('Execution time:    total: %f seconds' % (t2 -
start, ))
```

```
# standard boilerplate to call the main() function to begin
# the program.
if __name__ == '__main__':
    main()
```

8
AUTOSTEREOGRAMS



Stare at Figure 8-1 for a minute. Do you see anything other than random dots? Figure 8-1 is an *autostereogram*, a two-dimensional image that creates the illusion of three dimensions. Autostereograms usually consist of repeating patterns that resolve into three dimensions on closer inspection. If you can't see any sort of image, don't worry; it took me a while and a bit of experimentation before I could. (If you aren't having any luck with the version printed in this book, try the color version in the *images* folder of the book's GitHub repository. The footnote to the caption reveals what you should see.)

In this project, you'll use Python to create autostereograms. Here are some of the concepts covered in this project:

- Linear spacing and depth perception
- Depth maps
- Creating and editing images using `Pillow`
- Drawing into images using `Pillow`



Figure 8-1: *A puzzling image that might gnaw at you*[1]

The autostereograms you'll generate in this project are designed for "wall-eyed" viewing. The best way to see them is to focus your eyes on a point behind the image (such as a wall). Almost magically, once you perceive something in the patterns, your eyes should automatically bring it into focus, and when the three-dimensional image "locks in," you'll have a hard time shaking it off. (If you're still having trouble viewing the image, see Gene Levin's article "How to View Stereograms and Viewing Practice"[2] for help.)

How It Works

An autostereogram starts as an image with a repeating tiled pattern. The hidden 3D image is embedded into it by changing the linear spacing between the repeating patterns, thereby creating the illusion of depth. When you look at repeating patterns in an autostereogram, your brain can interpret the spacing as depth information, especially if there are multiple patterns with different spacing.

Perceiving Depth in an Autostereogram

When your eyes converge at an imaginary point behind the image, your brain matches the points seen by your left eye with a different group seen by your right eye, and you see these points lying on a plane behind the image. The perceived distance to this plane depends on the amount of spacing in the pattern. For example, Figure 8-2 shows three rows of *A*s. The *A*s are equidistant within each row, but their horizontal spacing increases from top to bottom.

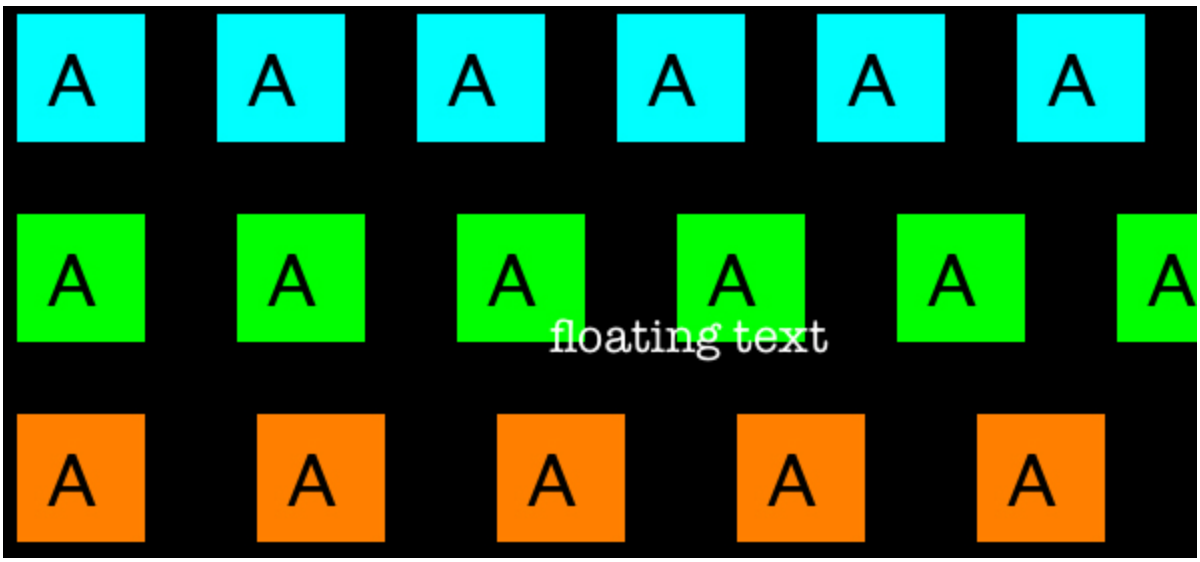


Figure 8-2: *Linear spacing and depth perception*

When this image is viewed "wall-eyed," the top row in Figure 8-2 should appear to be behind the paper, the middle row should look like it's a little behind the first row, and the bottom row should appear farthest from your eye. The text that says *floating text* should appear to "float" on top of these rows.

Why does your brain interpret the spacing between these patterns as depth? Normally, when you look at a distant object, your eyes work together to focus and converge at the same point, with both eyes rotating inward to point directly at the object. But when viewing a "wall-eyed"

autostereogram, focus and convergence happen at different locations. Your eyes focus on the autostereogram, but your brain sees the repeated patterns as coming from the same virtual (imaginary) object, and your eyes converge on a point behind the image, as shown in Figure 8-3. This combination of decoupled focus and convergence allows you to see depth in an autostereogram.

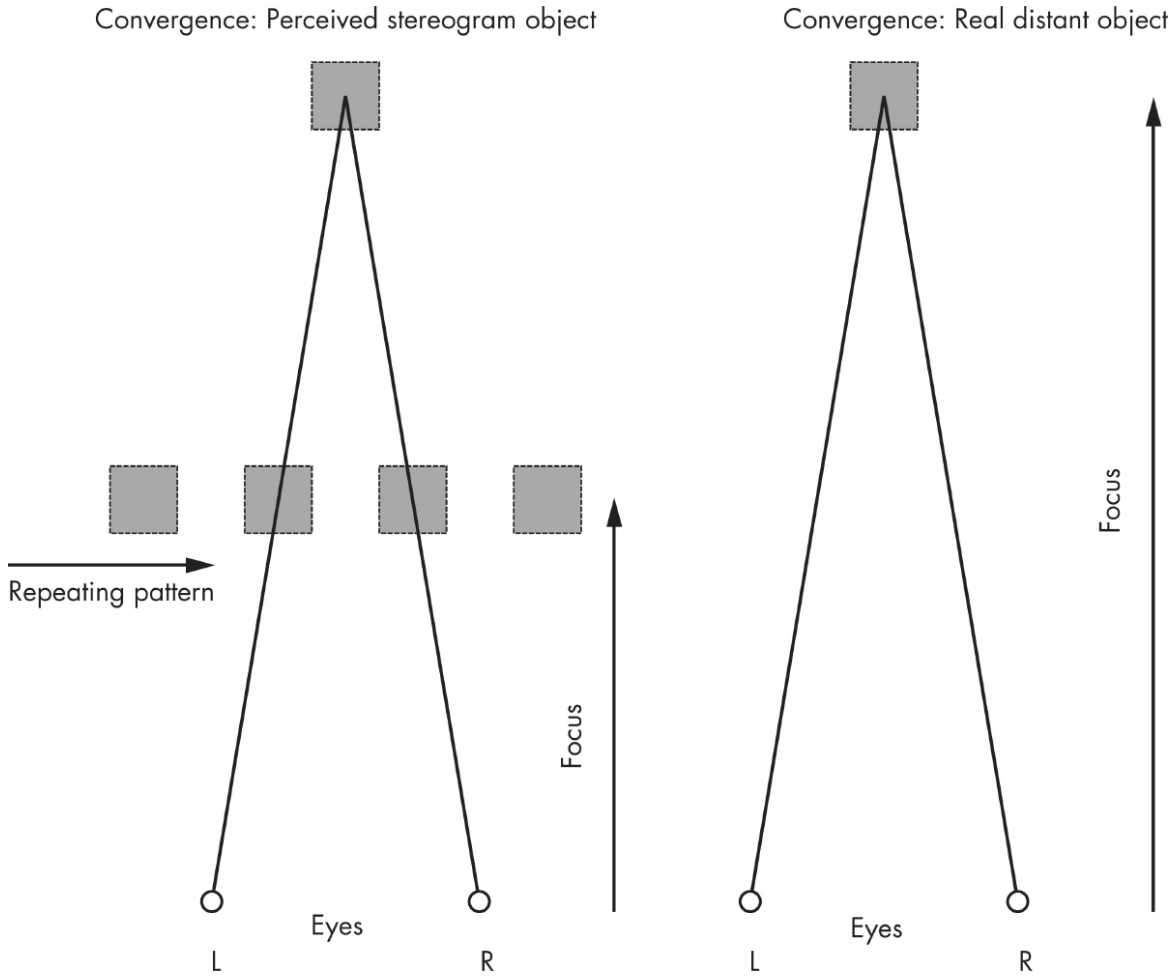


Figure 8-3: *Seeing depth in autostereograms*

The perceived depth of the autostereogram depends on the horizontal spacing of pixels. Because the first row in Figure 8-2 has the closest spacing, it appears in front of the other rows. However, if the spacing of the points were varied in the image, your brain would perceive each point at

a different depth, and you could see a virtual three-dimensional image appear.

Working with Depth Maps

The hidden image in an autostereogram comes from a *depth map*, an image in which the value of each pixel represents a depth value, which is the distance from the eye to the part of the object represented by that pixel. A depth map is often shown as a grayscale image, with light areas for nearby points and darker areas for points farther away, as shown in Figure 8-4.

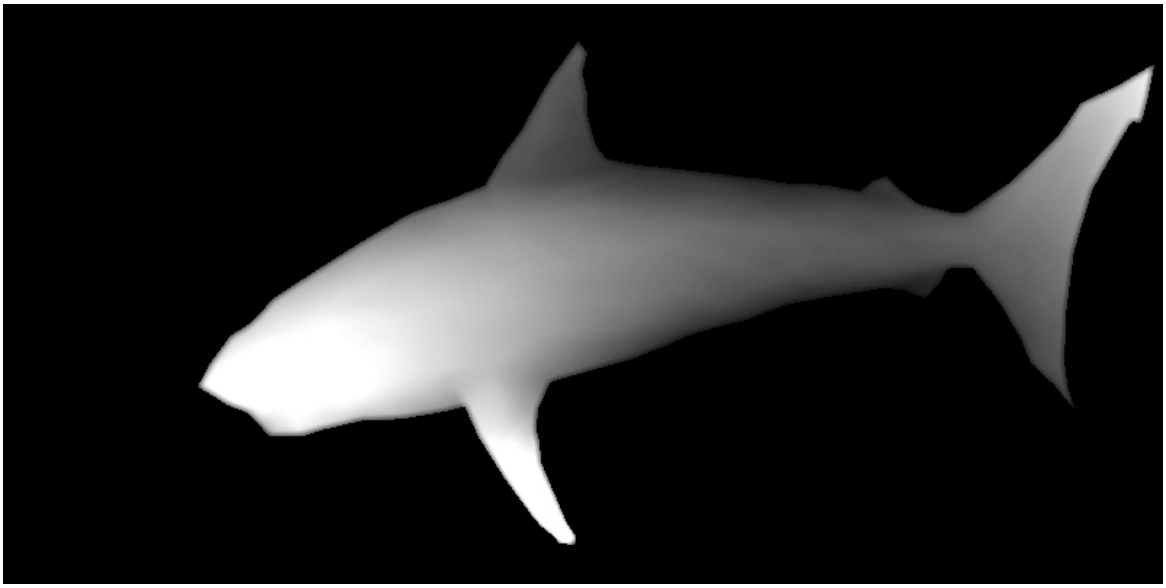


Figure 8-4: *A depth map*

Notice that the nose of the shark, the lightest part of the image, seems closest to you. The darker area toward the tail seems farthest away. (By the way, the image in Figure 8-4 is the same depth map used to create the first autostereogram shown in Figure 8-1.)

Because the depth map represents the depth or distance from the center of each pixel to the eye, you can use it to

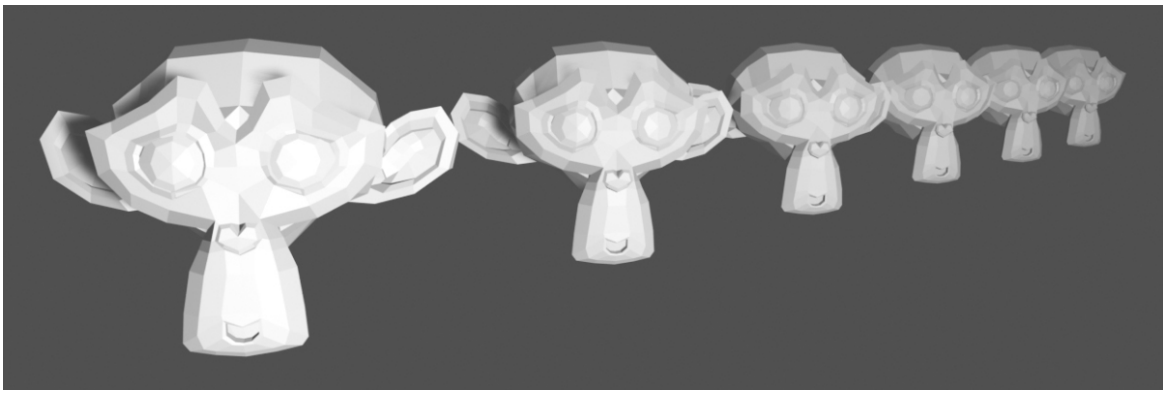
get the depth value associated with a pixel location in the image. You know that horizontal shifts are perceived as depth in images. So if you shift a pixel in a (patterned) image proportionally to the corresponding pixel's depth value, you would create a depth perception for that pixel consistent with the depth map. If you do this for all pixels, you'll end up encoding the entire depth map into the image, creating an autostereogram.

Depth maps store depth values for each pixel, and the resolution of the value depends on the number of bits used to represent it. Because you'll be using common 8-bit images in this chapter, depth values will be in the range [0, 255].

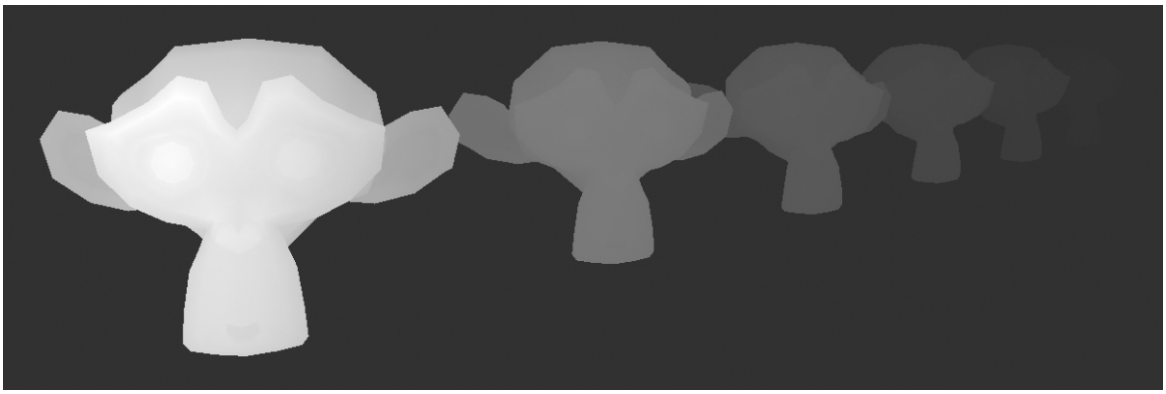
For the purposes of this project, I've posted several example depth maps to the book's GitHub repository. You can download the maps and use them as input for generating autostereograms. However, you may also want to try making your own depth maps to create some fancier images. There are two approaches you can take: using synthetic images created with 3D modeling software or using photographs taken with a smartphone camera.

Creating Depth Maps from 3D Models

If you create a 3D model of something using a 3D computer graphics program like Blender, you can also use the program to generate a depth map of the model. Figure 8-5 shows an example.



(a)



(b)

Figure 8-5: *A 3D model (a) and its associated depth map (b)*

Figure 8-5(a) shows a 3D model rendered using Blender, and Figure 8-5(b) shows the depth map created from this model. Search YouTube for "Blender depth map in 5 minutes!" for a tutorial[3] from Jonty Schmidt on how to do this. The key is to color the image based on the Z-distance from the camera.

Creating Depth Maps from Smartphone Photographs

These days, many smartphone cameras have a *portrait mode*, which captures depth information along with the photograph to selectively blur out the background. If you could get hold of this depth data, you'd have a depth map

of the photograph that you can use to create an autostereogram! Figure 8-6 shows an example.



(a)



(b)

Figure 8-6: *A portrait mode photo (a) and depth map (b) from an iPhone 11 camera*

Figure 8-6(a) shows a photograph taken in portrait mode with an iPhone 11, and Figure 8-6(b) shows the corresponding depth map. The depth map was created with the open source software ExifTool using this command:

```
exiftool -b -MPImage2 photo.jpg > depth.jpg
```

This command extracts the depth information from the metadata in the file *photo.jpg* and saves it to the file *depth.jpg*. Download ExifTool from *https://exiftool.org* to try the process for yourself. The command works for photos from an iPhone, but you can use similar techniques to extract the depth data from images taken with other types

of phones. There are also various apps available on the Android and iOS app stores that can help you with this. Here's one online depth map extractor that works with portrait mode images from a variety of phone models: *http://www.hasaranga.com/dmap*.

Shifting Pixels

We've looked at how our brains perceive the spacing between repeating elements in an image as depth information, and we've seen how depth information is conveyed through a depth map. Now let's look at how to shift the pixels in a tiled image in proportion to the values in a depth map. This is the key step in creating an autostereogram.

A tiled image is created by repeating a smaller image (the tile) in the x- and y-directions, although for depth perception, we're concerned only with the x-direction. If the tile that makes up the image is w pixels wide, you know the color values of the image's pixels will repeat every w pixels in the x-direction for any given row. Put another way, the color of the pixel in some row at point i along the x-axis can be expressed as:

$$C_i = C_{i-w} \text{ for } i \geq w$$

Let's consider an example. Given a tile width of 100 pixels, for a pixel with an x-axis position of 140, the equation tells you that $C_{140} = C_{140-100} = C_{40}$. This means the color value of the pixel at x-position 140 is the same as that of the pixel at x-position 40, due to the repetition of the image. (For values of i less than w in the previous formula, the color is just C_i, since the tile hasn't repeated yet.)

The goal is to shift pixels in the tiled image according to the values in the depth map. Let δ_i be the value at x-position i in the depth map. The shifted color value of the corresponding pixel in the tiled image is given by:

$$C_i = C_{i-w+\delta_i}$$

Returning to the example where the tile width is 100 pixels, given a pixel at x-position 140 and a corresponding depth map value of 10, the formula says that $C_{140} = C_{140 - 100 + 10} = C_{50}$. Because of the depth map, the color of the pixel at position 140 should be changed to match the color of the pixel at position 50. Since C_{50} is the same as C_{150}, this is effectively taking the pixel at x-position 150 and shifting it 10 pixels to the left. As a result, the repetition between positions 50 and 150 has become 10 pixels narrower, and your brain will perceive this change as depth information.

To create the complete autostereogram, you'll repeat this shifting process across the width of the image and over all the rows. You'll see how the shifting is actually implemented when you review the code.

Requirements

In this project, you'll use `Pillow` to read in images, access their underlying data, and create and modify images.

The Code

The code for this project will follow these steps to create an autostereogram:

1. Read in a depth map.
2. Read in a tile image or create a “random dot” tile. This will serve as the basis for the autostereogram’s repeating pattern.
3. Create a new image by repeating the tile. The dimensions of this image should match those of the depth map.
4. For each pixel in the new image, shift the pixel by an amount proportional to the depth value associated with the corresponding pixel in the depth map.
5. Write the resulting autostereogram to a file.

To see the complete project, skip ahead to “The Complete Code” on page 147. You can also download the full code listing for this chapter from *https://github.com/mkvenkit/pp2e/tree/main/autos*.

Creating a Tile from Random Circles

The user will have the option to provide a tile image at the start of the program (I’ve uploaded an image based on an M.C. Escher drawing to GitHub for this purpose). If one isn’t provided, create a tile with random circles using the `createRandomTile()` function.

```
def createRandomTile(dims):
    # create image
  ❶ img = Image.new('RGB', dims)
  ❷ draw = ImageDraw.Draw(img)
    # set the radius of a random circle to 1% of
```

```
    # width or height, whichever is smaller
❸   r = int(min(*dims)/100)
    # number of circles
❹   n = 1000
    # draw random circles
    for i in range(n):
        # -r makes sure that the circles stay inside and aren't
cut off
        # at the edges of the image so that they'll look better
when tiled
❺       x, y = random.randint(r, dims[0]-r), random.randint(r,
dims[1]-r)
❻       fill = (random.randint(0, 255), random.randint(0,
255),
                random.randint(0, 255))
❼       draw.ellipse((x-r, y-r, x+r, y+r), fill)
    return img
```

First you create a new Python Imaging Library (PIL) `Image` object with the dimensions given by `dims` ❶. Then you use `ImageDraw.Draw()` ❷ to draw circles inside the image with an arbitrarily chosen radius (`r`) of 1/100th of either the width or the height of the image, whichever is smaller ❸. (The Python `*` operator unpacks the width and height values in the `dims` tuple so that they can be passed into the `min()` method.)

You set the number of circles to draw to `1000` ❹ and then calculate the x- and y-coordinates of the center of each circle by calling `random.randint()` to get random integers in the range [`r`, *`width`*-`r`] and [`r`, *`height`*-`r`] ❺. Offsetting the range by `r` ensures the generated circles will fall entirely inside the boundaries of the tile. Without it, you could end up drawing a circle right at the edge of the image, which means it would be partly cut off. If you tiled such an image to create the autostereogram, the result wouldn’t look good because the circles at the edge between two tiles would have no spacing between them.

Next, you select a fill color for each circle by randomly choosing RGB values from the range [`0`, `255`] ❻. Finally, you use the `ellipse()` method in `draw` to draw each of your circles ❼. The first argument to this method is a tuple defining the bounding box of the circle, which is given by the top-left and bottom-right corners as (`x-r`, `y-r`) and (`x+r`, `y+r`), respectively, where (`x`, `y`) is the center of the circle and `r` is its radius. The other argument is the randomly chosen fill color.

You can test this method in the Python interpreter as follows:

```
>>> import autos
>>> img = autos.createRandomTile((256, 256))
>>> img.save('out.png')
>>> exit()
```

Figure 8-7 shows the output from the test.

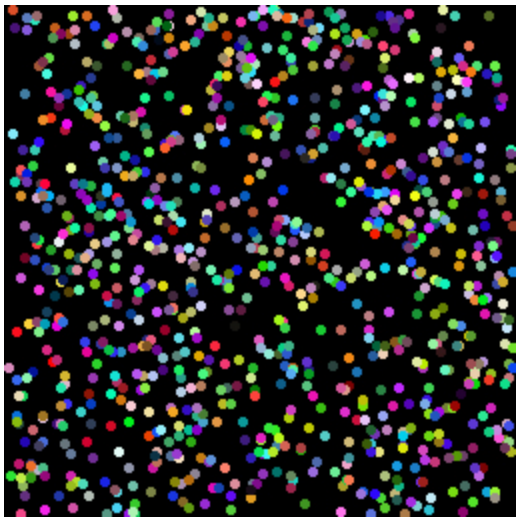


Figure 8-7: *A sample run of* `createRandomTile()`

As you can see in Figure 8-7, you've created an image with random dots that you can use as the autostereogram's tiled pattern.

Repeating a Given Tile

Now that you have a tile to work with, you can create an image by repeating that tile. This will form the basis of your autostereogram. Define a `createTiledImage()` function to do the work.

```
def createTiledImage(tile, dims):
    # create the new image
  ❶ img = Image.new('RGB', dims)
    W, H = dims
    w, h = tile.size
    # calculate the number of tiles needed
  ❷ cols = int(W/w) + 1
  ❸ rows = int(H/h) + 1
    # paste the tiles into the image
    for i in range(rows):
        for j in range(cols):
          ❹ img.paste(tile, (j*w, i*h))
    # output the image
    return img
```

The function takes in an image that will serve as the tiled pattern (`tile`), and the desired dimensions of the output image (`dims`). The dimensions are given as a tuple in the form (*`width`*, *`height`*). You create a new `Image` object using the supplied dimensions ❶. Next, you store the width and height of both the individual tile and the overall image. Dividing the overall image dimensions by those of the tile gives you the number of columns ❷ and rows ❸ of tiles you need to have in the image. You add 1 to each calculation to make sure that the last column of tiles on the right and the last row of tiles on the bottom aren’t missed when the output image dimension isn’t an exact integer multiple of the tile dimension. Without this precaution, the right and bottom of the image might be cut off. Finally, you loop through the

rows and columns and fill them with tiles ❹ . You determine the location of the top-left corner of the tile by multiplying `(j*w, i*h)` so it aligns with the rows and columns, just as you did in the photomosaic project. Once complete, the function returns an `Image` object of the specified dimensions, tiled with the input image `tile`.

Creating Autostereograms

Now let's create some autostereograms. The `createAutostereogram()` function does most of the work. Here it is:

```
def createAutostereogram(dmap, tile):
    # convert the depth map to a single channel if needed
  ❶ if dmap.mode != 'L':
        dmap = dmap.convert('L')
    # if no image is specified for a tile, create a random
  circles tile
  ❷ if not tile:
        tile = createRandomTile((100, 100))
    # create an image by tiling
  ❸ img = createTiledImage(tile, dmap.size)
    # create a shifted image using depth map values
  ❹ sImg = img.copy()
    # get access to image pixels by loading the Image object
  first
  ❺ pixD = dmap.load()
    pixS = sImg.load()
    # shift pixels horizontally based on depth map
  ❻ cols, rows = sImg.size
    for j in range(rows):
        for i in range(cols):
          ❼ xshift = pixD[i, j]/10
          ❽ xpos = i - tile.size[0] + xshift
          ❾ if xpos > 0 and xpos < cols:
              ❿ pixS[i, j] = pixS[xpos, j]
```

```
    # display the shifted image
    return sImg
```

First you convert the provided depth map (`dmap`) into a single-channel grayscale image if needed ❶. If the user doesn't supply an image for the tile, you then create a tile of random circles using the `createRandomTile()` function you defined earlier ❷. Next, you use your `createTiledImage()` function to create a tiled image that matches the size of the supplied depth map image ❸. You then make a copy of this tiled image ❹. This copy will become the final autostereogram.

The function continues by using the `Image.load()` method on the depth map and the output image ❺. This method loads image data into memory, allowing you to access an image's pixels as a two-dimensional array in the form `[i, j]`. You store the image dimensions as a number of rows and columns ❻, treating the image as a grid of individual pixels.

The core of the autostereogram creation algorithm lies in the way you shift the pixels in the tiled image according to the information gathered from the depth map. To do this, you iterate through the tiled image and process each pixel. First you look up the value of the corresponding pixel from the depth map and divide this value by 10 to determine a shift value for the tiled image ❼. You divide by 10 because you're using an 8-bit depth map here, which means the depth varies from 0 to 255. If you divide these values by 10, you get depth values in the approximate range of 0 to 25. Since the depth map input image dimensions are usually in the hundreds of pixels, these shift values work fine. (Play around by changing the value you divide by to see how it affects the final image.)

Next, you use the formula discussed in "Shifting Pixels" on page 140 to calculate the x-axis coordinate to look for the pixel's new color value ❽. Pixels with a depth map value of 0

(black) won't be shifted and will be perceived as the background. After checking to make sure you're not trying to access a pixel that's not in the image (which can happen at the image's edges because of the shift) ❾, you replace each pixel with its shifted value ❿.

Providing Command Line Options

The `main()` function of the program provides some command line options to customize the autostereogram.

```
def main():
    # create a parser
    parser =
  argparse.ArgumentParser(description="Autostereograms...")
    # add expected arguments
  ❶ parser.add_argument('--depth', dest='dmFile',
  required=True)
    parser.add_argument('--tile', dest='tileFile',
  required=False)
    parser.add_argument('--out', dest='outFile',
  required=False)
    # parse args
    args = parser.parse_args()
    # set the output file
    outFile = 'as.png'
    if args.outFile:
        outFile = args.outFile
    # set tile
    tileFile = False
    if args.tileFile:
        tileFile = Image.open(args.tileFile)
```

As with previous projects, you define the command line options for the program using `argparse`. The one required argument is the name of the depth map file ❶. There are also two optional arguments, one to provide an image file to

use as the tile pattern and the other to set the name of the output file. If a tile image isn't specified, the program will generate a tile of random circles. If the output filename isn't specified, the autostereogram is written to a file named *as.png*.

Running the Autostereogram Generator

Now let's run the program using a depth map of a stool (*stool-depth.png*), which you'll find in the *data* folder of this project's GitHub repository:

```
$ python autos.py --depth data/stool-depth.png
```

Figure 8-8 shows the depth map image on the left and the generated autostereogram on the right. Because you haven't supplied a graphic for the tile, this autostereogram is created using random tiles.

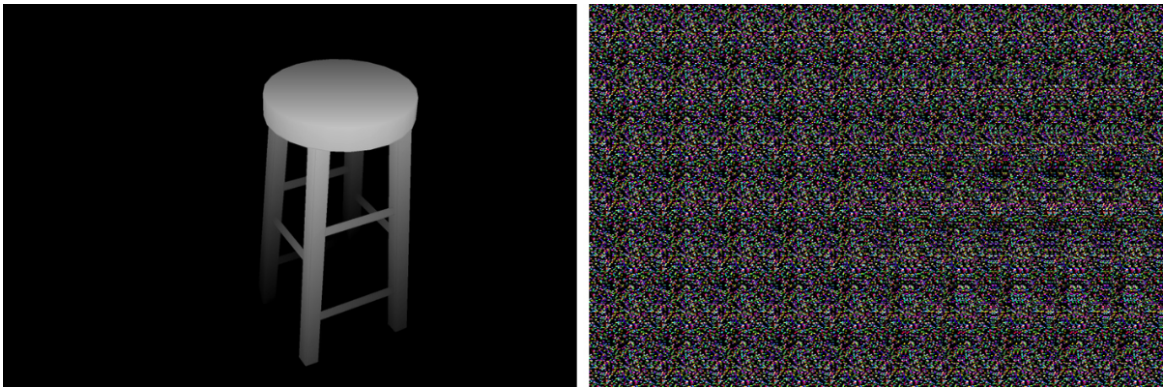


Figure 8-8: *A sample run of* autos.py

Now let's give a tile image as input. Use the *stool-depth.png* depth map as you did earlier, but this time, supply the image *escher-tile.jpg* for the tiles.

```
$ python autos.py --depth data/stool-depth.png —-tile
  data/escher-tile.jpg
```

Figure 8-9 shows the output.

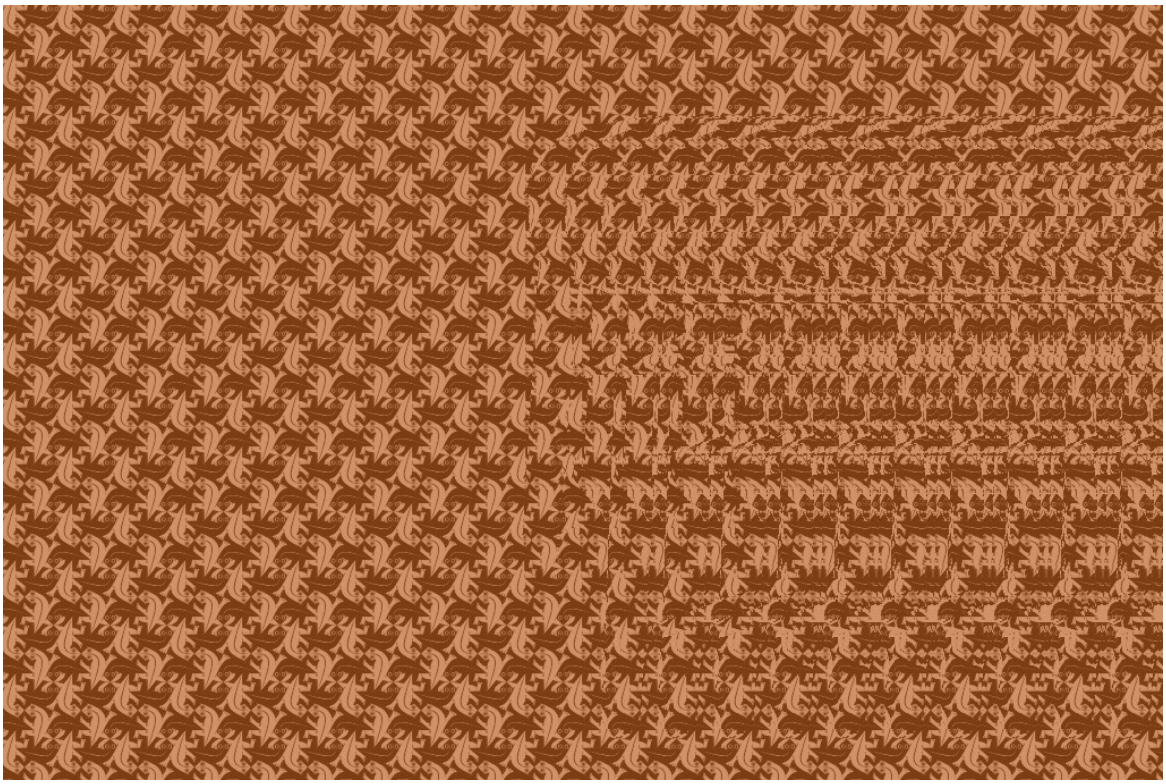


Figure 8-9: *A sample run of* autos.py *using tiles*

Enjoy creating moiré autostereograms using the images I've provided on GitHub or using your own depth maps!

Summary

In this project, you learned how to create autostereograms. Given a depth map image, you can now create either a random dot autostereogram or one tiled with an image you supply.

Experiments!

Here are some ways to further explore autostereograms:

1. Write code to create an image similar to Figure 8-2 that demonstrates how changes in the linear spacing in an image can create illusions of depth. (Hint: use image tiles and the `Image.paste()` method.)
2. Add a command line option to the program to specify the scale to be applied to the depth map values. (Remember that the code divides the depth map value by 10.) How does changing the value affect the autostereogram?

The Complete Code

Here's the complete autostereogram project code:

```
"""
autos.py

A program to create autostereograms.

Author: Mahesh Venkitachalam
"""

import sys, random, argparse
from PIL import Image, ImageDraw

# create spacing/depth example
def createSpacingDepthExample():
    tiles = [Image.open('test/a.png'),
  Image.open('test/b.png'),
             Image.open('test/c.png')]
    img = Image.new('RGB', (600, 400), (0, 0, 0))
    spacing = [10, 20, 40]
    for j, tile in enumerate(tiles):
```

```
        for i in range(8):
            img.paste(tile, (10 + i*(100 + j*10), 10 + j*100))
    img.save('sdepth.png')

# create image filled with random dots
def createRandomTile(dims):
    # create image
    img = Image.new('RGB', dims)
    draw = ImageDraw.Draw(img)
    # calculate radius - % of min dimension
    r = int(min(*dims)/100)
    # number of dots
    n = 1000
    # draw random circles
    for i in range(n):
        # -r is used so circle stays inside - cleaner for
  tiling
        x, y = random.randint(r, dims[0]-r), random.randint(r,
  dims[1]-r)
        fill = (random.randint(0, 255), random.randint(0, 255),
                random.randint(0, 255))
        draw.ellipse((x-r, y-r, x+r, y+r), fill)
    # return image
    return img

# create a larger image of size dims by tiling the given image
def createTiledImage(tile, dims):
    # create output image
    img = Image.new('RGB', dims)
    W, H = dims
    w, h = tile.size
    # calculate # of tiles needed
    cols = int(W/w) + 1
    rows = int(H/h) + 1
    # paste tiles
    for i in range(rows):
        for j in range(cols):
            img.paste(tile, (j*w, i*h))
    # output image
    return img
```

```
# create a depth map for testing:
def createDepthMap(dims):
    dmap = Image.new('L', dims)
    dmap.paste(10, (200, 25, 300, 125))
    dmap.paste(30, (200, 150, 300, 250))
    dmap.paste(20, (200, 275, 300, 375))
    return dmap

# given a depth map (image) and an input image, create a new
  image
# with pixels shifted according to depth
def createDepthShiftedImage(dmap, img):
    # size check
    assert dmap.size == img.size
    # create shifted image
    sImg = img.copy()
    # get pixel access
    pixD = dmap.load()
    pixS = sImg.load()
    # shift pixels output based on depth map
    cols, rows = sImg.size
    for j in range(rows):
        for i in range(cols):
            xshift = pixD[i, j]/10
            xpos = i - 140 + xshift
            if xpos > 0 and xpos < cols:
                pixS[i, j] = pixS[xpos, j]
    # return shifted image
    return sImg

# given a depth map (image) and an input image, create a new
  image
# with pixels shifted according to depth
def createAutostereogram(dmap, tile):
    # convert depth map to single channel if needed
    if dmap.mode != 'L':
        dmap = dmap.convert('L')
    # if no tile specified, use random image
    if not tile:
        tile = createRandomTile((100, 100))
```

```
    # create an image by tiling
    img = createTiledImage(tile, dmap.size)
    # create shifted image
    sImg = img.copy()
    # get pixel access
    pixD = dmap.load()
    pixS = sImg.load()
    # shift pixels output based on depth map
    cols, rows = sImg.size
    for j in range(rows):
        for i in range(cols):
            xshift = pixD[i, j]/10
            xpos = i - tile.size[0] + xshift
            if xpos > 0 and xpos < cols:
                pixS[i, j] = pixS[xpos, j]
    # return shifted image
    return sImg

# main() function
def main():
    # use sys.argv if needed
    print('creating autostereogram...')
    # create parser
    parser =
  argparse.ArgumentParser(description="Autostereograms...")
    # add expected arguments
    parser.add_argument('--depth', dest='dmFile',
  required=True)
    parser.add_argument('--tile', dest='tileFile',
  required=False)
    parser.add_argument('--out', dest='outFile',
  required=False)
    # parse args
    args = parser.parse_args()
    # set output file
    outFile = 'as.png'
    if args.outFile:
        outFile = args.outFile
    # set tile
    tileFile = False
```

```
    if args.tileFile:
        tileFile = Image.open(args.tileFile)
    # open depth map
    dmImg = Image.open(args.dmFile)
    # create stereogram
    asImg = createAutostereogram(dmImg, tileFile)
    # write output
    asImg.save(outFile)

# call main
if __name__ == '__main__':
    main()
```

---

1 The hidden image is a shark.

2 *http://colorstereo.com/texts_.txt/practice.htm*

3 *https://www.youtube.com/watch?v=oqpDqKpOChE*

PART IV

ENTER 3D

In One Dimensions, did not a moving Point produce a Line with two *terminal points?*
In Two Dimensions, did not a moving Line produce a Square with four *terminal points?*
In Three Dimensions, did not a moving Square produce—did not the eyes of mine behold it—
that blessed being, a Cube, with eight *terminal points?*

—Edwin A. Abbott, *Flatland: A Romance of Many Dimensions*

9
UNDERSTANDING OPENGL



In this project, you'll create a simple program that displays a texture-mapped square using OpenGL and GLFW. OpenGL is a software interface to your graphics processing unit (GPU), and GLFW is a windowing toolkit. You'll also learn how to use the C-like OpenGL Shading Language (GLSL) to write *shaders*—code that executes in the GPU. Shaders bring immense flexibility to computations in OpenGL. I'll show you how to use GLSL shaders to transform and color geometry as you create a rotating, textured polygon (as shown in Figure 9-1).

GPUs are optimized to perform the same operations on huge amounts of data repeatedly, in parallel, which makes them much faster than central processing units (CPUs) for rendering computer graphics. In addition, they're being used for general-purpose computing, and specialized languages now let you harness your GPU hardware for all sorts of applications. You'll leverage the GPU, OpenGL, and shaders in this project.

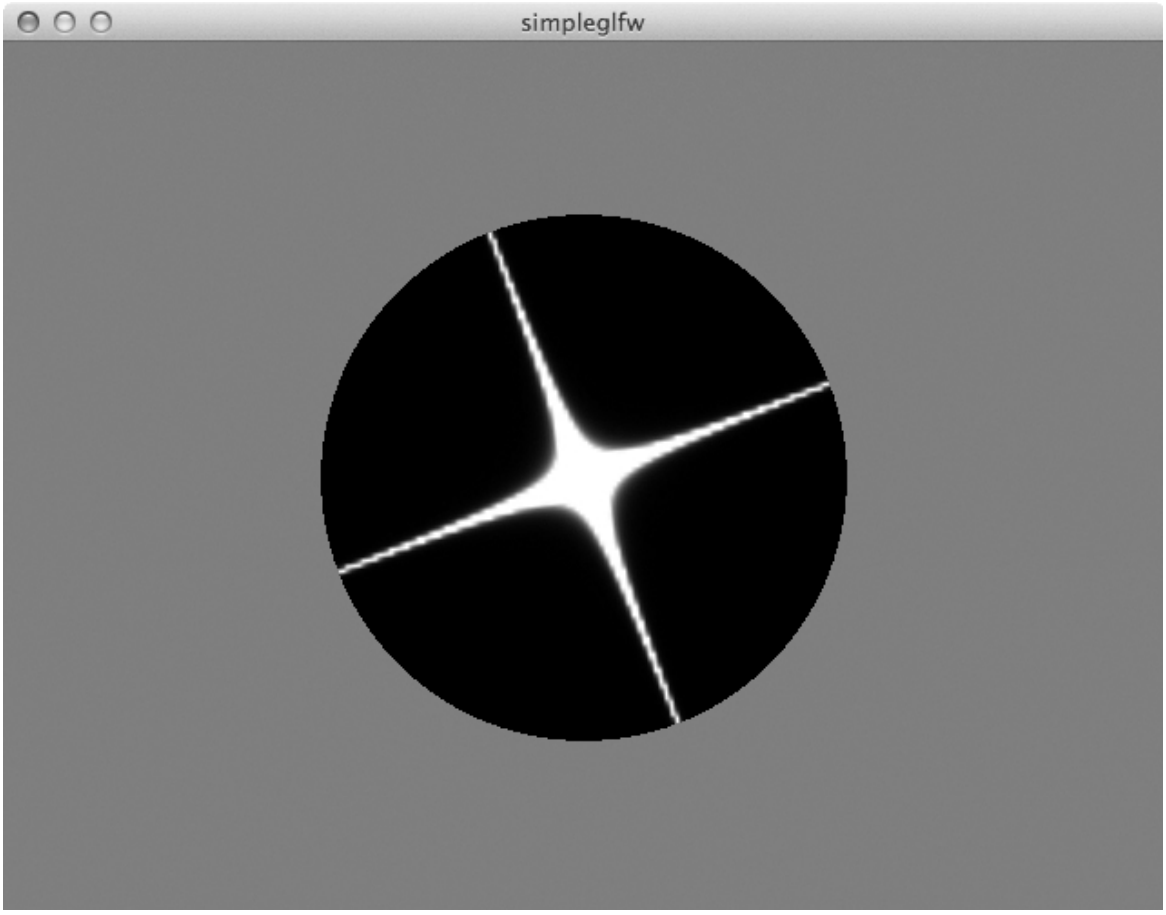


Figure 9-1: *The final image for the project in this chapter—a rotating polygon with a star image. This square polygon boundary is clipped to a black circle using a shader.*

Python is an excellent “glue” language. There are a vast number of Python *bindings* available for libraries written in other languages, such as C, that allow you to use these libraries in Python. In this chapter and in Chapters 10 and 11, you’ll use `PyOpenGL`, the Python binding to OpenGL, to create computer graphics.

Here are some of the concepts introduced in this project:

- Using the GLFW windowing library for OpenGL
- Using GLSL to write vertex and fragment shaders
- Performing texture mapping
- Using 3D transformations

First, let’s take a look at how OpenGL works.

NOTE *OpenGL went through a major transition a few years ago. It went from using a fixed function graphics pipeline to a programmable pipeline with a dedicated shading language. We refer to the latter as* modern OpenGL*, and that’s what we’ll be using in this book. Specifically, we’ll use OpenGL version 4.1.*

How OpenGL Works

Modern OpenGL makes graphics appear on your screen through a sequence of operations commonly known as the *3D graphics pipeline*. Figure 9-2 shows a simplified representation of the OpenGL 3D graphics pipeline.

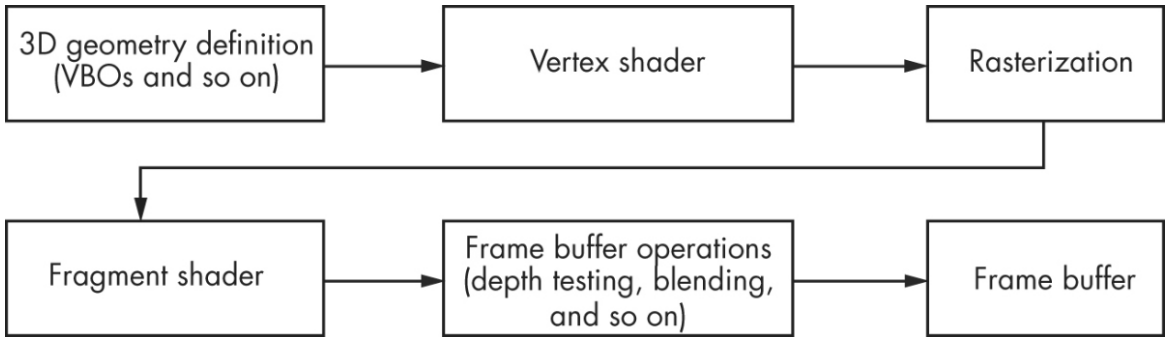


Figure 9-2: *The (simplified) OpenGL graphics pipeline*

At their heart, computer graphics boil down to computing color values for the pixels on your screen. Say you want to make a triangle appear. In the first step of the pipeline, you define the 3D geometry by defining the vertices of the triangle in 3D space and specifying the colors associated with each vertex. These vertices and colors are held in data structures called *vertex buffer objects (VBOs)*. Next, you transform the vertices: the first transformation places the vertices in 3D space, and the second projects the 3D

coordinates onto 2D space for display on a 2D screen. The color values for the corresponding vertices are also calculated in this step based on factors such as lighting, typically in code called the *vertex shader*.

Next, the geometry is *rasterized* (converted from a 3D representation to 2D pixels), and for each pixel (or *fragment*, to be more accurate), another block of code called the *fragment shader* is executed. Just as the vertex shader operates on 3D vertices, the fragment shader operates on the 2D fragments after rasterization. I say *fragment* rather than *pixel* since a pixel is what is displayed on the screen, whereas a fragment is the output of computations in the fragment shader, and depending on the next step in the pipeline, a fragment may be discarded before it becomes a pixel on the screen.

Finally, each fragment passes through a series of frame buffer operations, where it undergoes *depth buffer testing* (checking whether one fragment obscures another), *blending* (mixing two fragments with transparency), and other operations that combine its current color with what is already on the frame buffer at that location. These changes end up on the final frame buffer, which is typically displayed on the screen.

Geometric Primitives

Because OpenGL is a low-level graphics library, you can't ask it directly to draw a cube or a sphere, though libraries built on top of it can do such tasks for you. OpenGL understands only low-level geometric primitives, such as points, lines, and triangles.

Modern OpenGL supports only the primitive types `GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, and `GL_TRIANGLE_FAN`. Figure 9-3 shows how

the vertices for the primitives are organized. Each vertex has a 3D coordinate such as (*x*, *y*, *z*).

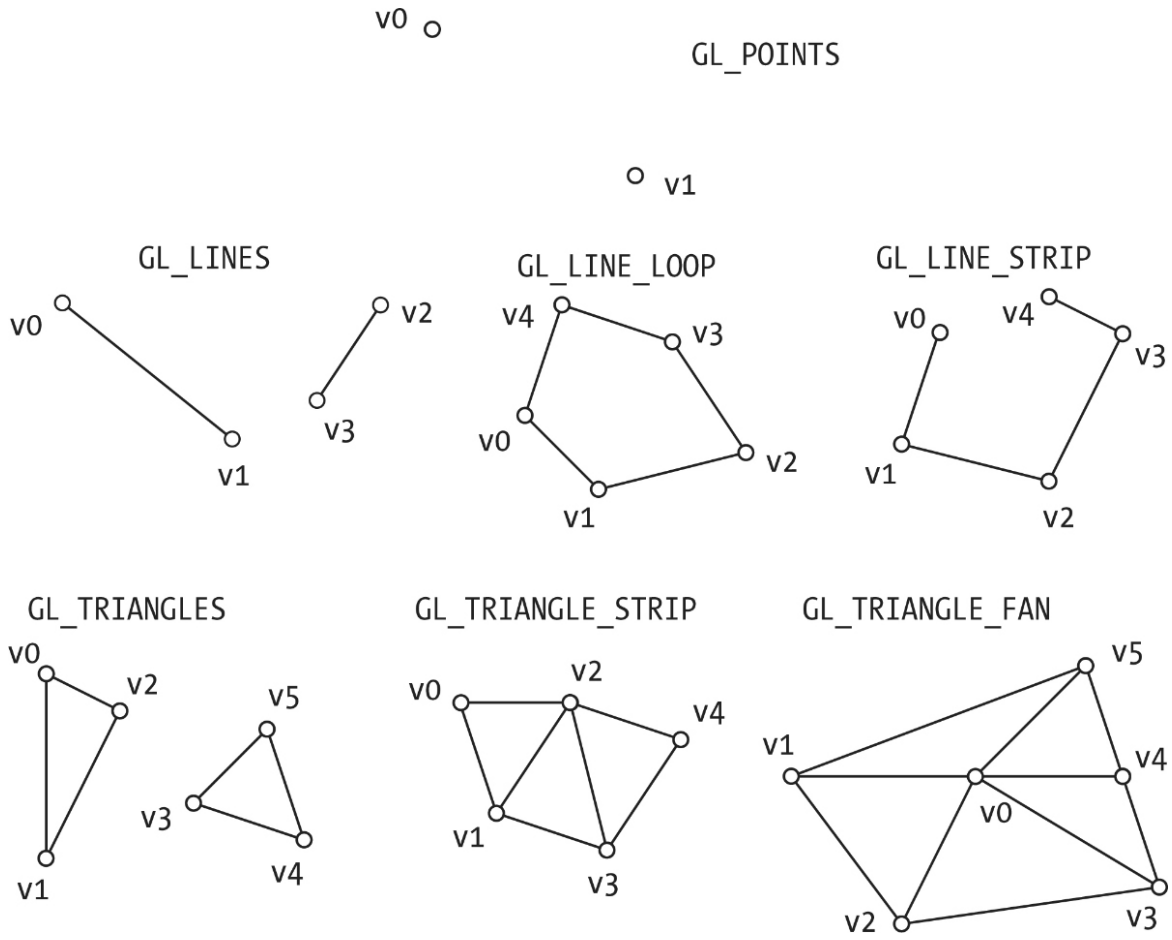


Figure 9-3: *OpenGL primitives*

To draw a sphere in OpenGL, first define the geometry of the sphere mathematically and compute its 3D vertices. Then assemble the vertices into basic geometric primitives; for example, you could group each set of three vertices into a triangle. You then render the vertices using OpenGL.

3D Transformations

You can’t learn computer graphics without learning about 3D transformations. Conceptually, these are quite simple to understand. You have an object—what can you do to it? You

can move it, stretch (or squash) it, or rotate it. You can do other things to it too, but these three tasks—translation, scale, and rotation—are the operations or transformations most commonly performed on an object. In addition to these commonly used transformations, you'll use a perspective projection to map the 3D objects onto the 2D plane of the screen. These transformations are all applied on the coordinates of the object you are trying to transform.

While you're probably familiar with 3D coordinates in the form (x, y, z), in 3D computer graphics you use coordinates in the form (x, y, z, w), called *homogeneous coordinates*. (These coordinates come from a branch of mathematics called *projective geometry*, which is beyond the scope of this book.) Homogeneous coordinates allow you to express common 3D transformations such as translation, scale, and rotation as 4×4 matrices. But for the purposes of these OpenGL projects, all you need to know is that the homogeneous coordinate (x, y, z, w) is equivalent to the 3D coordinate (x/w, y/w, z/w, 1.0). A 3D point (1.0, 2.0, 3.0) can be expressed in homogeneous coordinates as (1.0, 2.0, 3.0, 1.0).

Here's an example of a 3D transformation using a 4×4 matrix. See how the matrix multiplication translates a point (x, y, z, 1.0) to ($x + t_x$, $y + t_y$, $z + t_z$, 1.0):

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix}$$

Since this operation translates a point in space, the 4×4 matrix involved is called a *translation matrix*.

Let's now look at another useful matrix for 3D transformation—a rotation matrix. The following matrix rotates a point (x, y, z, 1.0) counterclockwise around the x-axis by θ radians:

$$R_{\theta,x} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

But here's something to keep in mind: if you're going to apply this rotation inside shader code, the matrix will be stored in *column-major format*, which means you should declare it as follows:

```
// rotational transform
mat4 rot =  mat4(
    vec4(1.0,  0.0,          0.0,         0.0),
    vec4(0.0,  cos(uTheta),  sin(uTheta), 0.0),
    vec4(0.0, -sin(uTheta),  cos(uTheta), 0.0),
    vec4(0.0,  0.0,          0.0,         1.0)
);
```

Notice that, in the code, the matrix is flipped along its diagonal compared to the definition of $R_{\theta,x}$.

Two terms that you will encounter often in OpenGL are *modelview* and *projection* transformations. With the advent of customizable shaders in modern OpenGL, modelviews

and projections are just generic transformations. Historically, in old-school versions of OpenGL, the modelview transformations were applied to your 3D model to position it in space, and the projection transformations were used to map the 3D coordinates onto a 2D surface for display, as you'll see in a moment. Modelview transformations are user-defined transformations that let you position your 3D objects, and projection transformations are projective transformations that map 3D onto 2D.

The two most commonly used 3D graphics projective transformations are *orthographic* and *perspective*, but here you'll use only perspective projections, which are defined by a *field of view* (the extent to which the eye can see), a *near plane* (the plane closest to the eye), a *far plane* (the plane farthest from the eye), and an *aspect ratio* (the ratio of the width to the height of the near plane). Together, these parameters constitute a camera model for a projection that determines how the 3D figure will be mapped onto a 2D screen, as shown in Figure 9-4. The truncated pyramid shown in the figure is the *view frustum*. The *eye* is the 3D location where you place the camera. (For orthographic projection, the eye will be at infinity, and the pyramid will become a rectangular cuboid.)

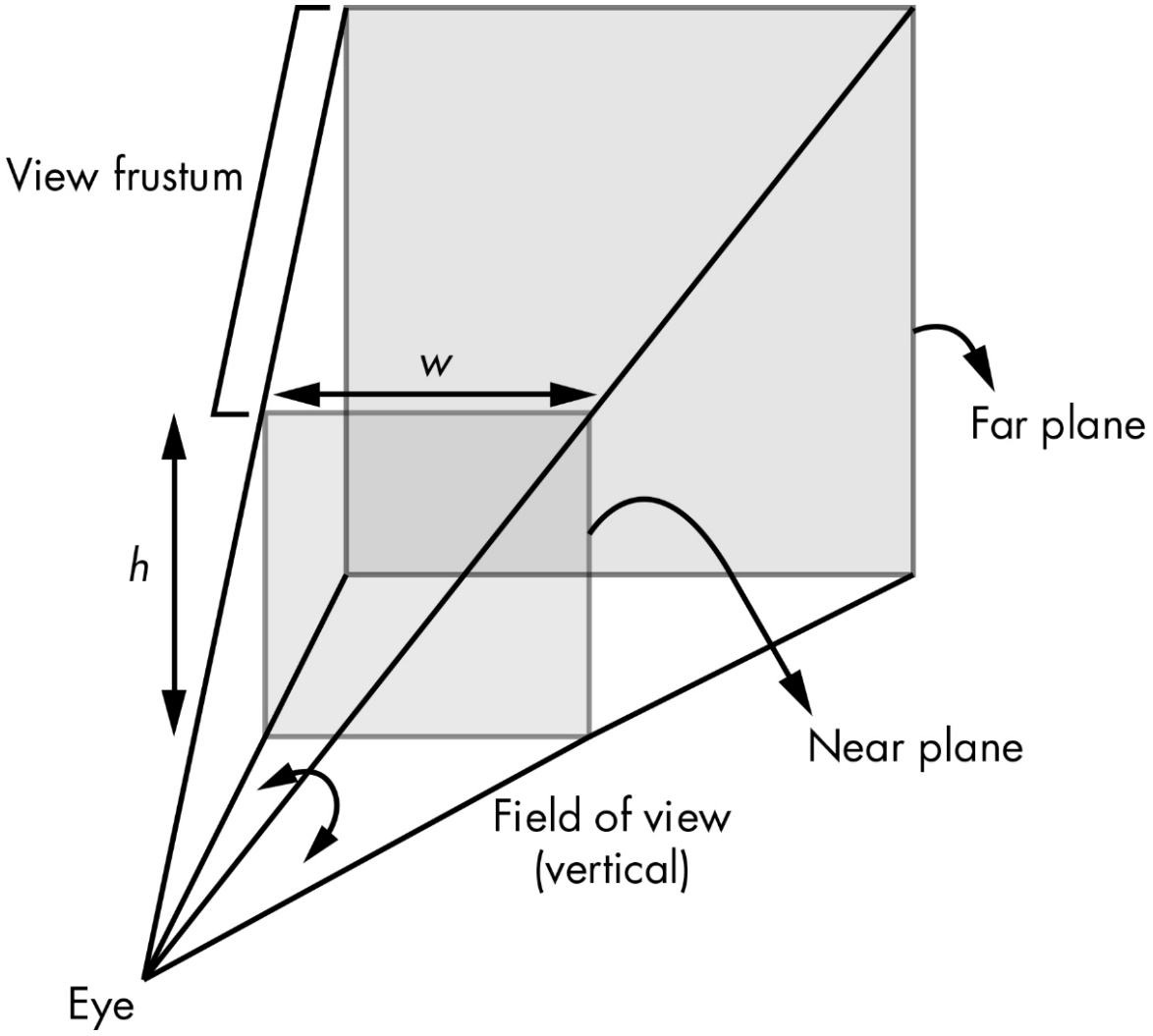


Figure 9-4: *A perspective projection camera model*

Once the perspective projection is complete and before rasterization, the graphics primitives are clipped (or cut out) against the near and far planes shown in Figure 9-4. The near and far planes are chosen such that the 3D objects you want to appear onscreen lie inside the view frustum; otherwise, they will be clipped away.

Shaders

You've seen how shaders fit into the modern OpenGL programmable graphics pipeline. Now let's look at a simple

pair of vertex and fragment shaders to get a sense of how GLSL works.

A Vertex Shader

Here is a simple vertex shader that computes the position and color of a vertex:

```
# version 410 core

in vec3 aVert;

uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;

out vec4 vCol;

void main() {
    // apply transformations
  ❻ gl_Position = uPMatrix * uMVMatrix * vec4(aVert, 1.0);
    // set color
  ❼ vCol = vec4(1.0, 0.0, 0.0, 1.0);
}
```

You first set the version of GLSL used in the shader to version 4.1 ❶. Then you define an input named `aVert` of type `vec3` (a 3D vector) for the vertex shader using the keyword `in` ❷ . You next define two variables of type `mat4` (4×4 matrices), which correspond to the modelview ❸ and projection ❹ matrices. The `uniform` prefix to these variables indicates that they do not change during execution of the vertex shader for a given rendering call on a set of vertices. You use the `out` prefix to define the output of the vertex shader, which is a color variable of type `vec4` (a 4D vector to store red, green, blue, and alpha channels) ❺.

Now you come to the `main()` function, where the vertex shader program starts. The value of `gl_Position` is computed by transforming the input `aVert` using the uniform matrices

passed in ❻. The GLSL variable `gl_Position` is used to store the transformed vertices. You set the output color from the vertex shader to red with no transparency by using the value (1, 0, 0, 1) ❼. You'll use this as input in the next shader in the pipeline.

A Fragment Shader

Now let's look at a simple fragment shader that computes the fragment color based on the vertex color passed in:

```
# version 410 core

in vec4 vCol;

out vec4 fragColor;

void main() {
    // use vertex color
  ❹ fragColor = vCol;
}
```

After setting the version of GLSL used in the shader ❶, you set `vCol` as the input to the fragment shader ❷. This variable, `vCol`, was set as output from the vertex shader. (Remember, the vertex shader executes for every vertex in the 3D scene, whereas the fragment shader executes for every fragment on the screen.) You also set the fragment shader's output color variable `fragColor` ❸.

During rasterization (which occurs between the vertex and fragment shaders), OpenGL converts the transformed vertices to fragments, and the color of the fragments lying between the vertices is calculated by interpolating the color values at the vertices; `vCol` in the previous code is this interpolated color. You set the fragment shader's output to be the same as the interpolated color going into the fragment shader ❹. By default, and in most cases, the

intended output of the fragment shader is the screen, and the color you set ends up there (unless it's affected by operations, such as depth testing, that occur in the final stage of the graphics pipeline).

For the GPU to execute the shader code, it needs to be compiled and linked to instructions that the hardware understands. OpenGL provides ways to do this and reports detailed compiler and linker errors that will help you develop the shader code. The compilation process also generates a table of locations or indices for the variables declared in your shaders so you can connect them to variables in your Python code.

Vertex Buffers

Vertex buffers are an important mechanism used by OpenGL shaders. Modern graphics hardware and OpenGL are designed to work with large amounts of 3D geometry. Consequently, several mechanisms are built into OpenGL to help transfer data from the program to the GPU. A typical setup to draw 3D geometry in a program will do the following:

1. Define arrays of coordinates, colors, and other attributes for each vertex of the 3D geometry.
2. Create a vertex array object (VAO) and bind to it.
3. Create vertex buffer objects (VBOs) for each attribute, defined on a per-vertex basis.
4. Bind to the VBO and set the buffer data using the predefined arrays.
5. Specify the data and location of vertex attributes to be used in the shader.
6. Enable the vertex attributes.

7. Render the data.

After you define the 3D geometry in terms of vertices, you create and bind to a vertex array object. VAOs are a convenient way to group geometry as multiple arrays of coordinates, colors, and so on. Then, for each attribute of each vertex, you create a vertex buffer object and set your 3D data into it. The VBO stores the vertex data in the GPU memory. Now, all that's left is to connect the buffer data so you can access it from your shaders. You do this through calls that use the location of the variables employed in the shader.

Texture Mapping

Now let's look at texture mapping, an important computer graphics technique that you'll use in this chapter. *Texture mapping* is a way to give a scene a realistic feel with the help of a 2D picture of a 3D object (like the backdrop in a play). A texture is usually read from an image file and is stretched to drape over a geometric region by mapping the 2D coordinates (in the range [0, 1]) onto the 3D coordinates of the polygons. For example, Figure 9-5 shows an image draped onto one face of a cube. (I used `GL_TRIANGLE_STRIP` primitives to draw the cube faces, and the ordering of the vertices is indicated by the lines on the face.)

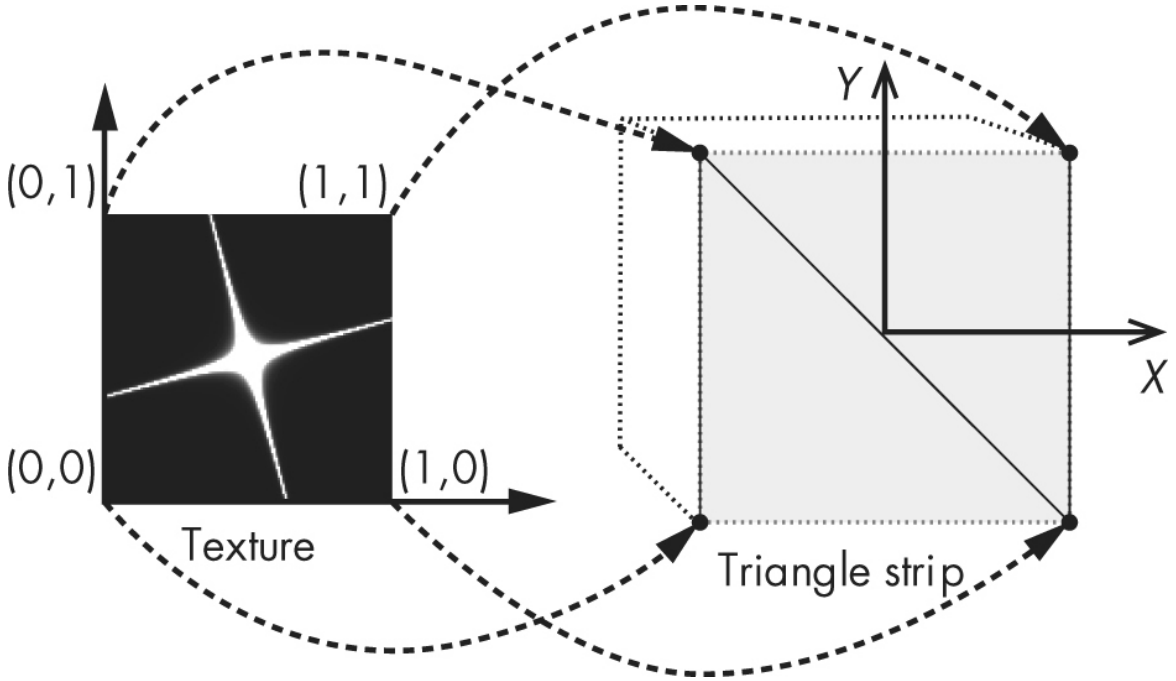


Figure 9-5: *Texture mapping*

In Figure 9-5, the (0, 0) corner of the texture is mapped to the bottom-left vertex of the cube face. Similarly, you can see how the other corners of the texture are mapped, with the net effect that the texture is “pasted” onto this cube face. The geometry of the cube face itself is defined as a triangle strip, and the vertices zigzag from the bottom to the top left and from the bottom to the top right. Textures are extremely powerful and versatile computer graphics tools, as you’ll see in Chapter 11.

The OpenGL Context

Now let’s talk about how to get OpenGL to draw stuff on the screen. The entity that stores all the OpenGL state information is called the *OpenGL context*. Contexts have a viewable, window-like area where the OpenGL drawings go, and you can have multiple contexts per process or run of an application, but only one context per thread can be current

at a time. (Fortunately, the window toolkit will take care of most of the context handling.)

For your OpenGL output to appear in a window onscreen, you need the help of the operating system. For these projects, you'll use GLFW, a lightweight cross-platform C library that lets you create and manage OpenGL contexts, display the 3D graphics in a window, and handle user input such as mouse clicks and keypresses. (Appendix A covers the installation details for this library.)

Because you're writing code in Python and not C, you'll also use a Python binding to GLFW (*glfw.py*, available in the *common* directory in the book's code repository), which lets you access all the GLFW features using Python.

Requirements

You'll use `PyOpenGL`, a popular Python binding for OpenGL, for rendering, and you'll use `numpy` arrays to represent 3D coordinates and transformation matrices.

The Code

In this project, you'll build a simple Python application for displaying a rotating, textured polygon using OpenGL. To see the complete project code, skip ahead to "The Complete Code" on page 172. The complete code for our simple OpenGL application resides in two files. The main project code discussed in this chapter is in *simpleglfw.py*, which can be found at *https://github.com/mkvenkit/pp2e/tree/main/simplegl*. The helper functions are in *glutils.py*, which can be found in the GitHub repository's *common* directory.

The RenderWindow Class

The `RenderWindow` class manages the creation of the window that displays the OpenGL graphics. It initializes GLFW, sets up OpenGL, manages rendering, and sets up callbacks to receive keyboard input.

Creating an OpenGL Window

The first order of business of the `RenderWindow` class is to set up GLFW so you have an OpenGL window to render into. The class's initialization code addresses this task:

```
class RenderWindow:
    """GLFW rendering window class"""
    def __init__(self):

        # save current working directory
        cwd = os.getcwd()

        # initialize glfw
      ❶ glfw.glfwInit()

        # restore cwd
        os.chdir(cwd)

        # version hints
      ❷ glfw.glfwWindowHint(glfw.GLFW_CONTEXT_VERSION_MAJOR,
 4)
        glfw.glfwWindowHint(glfw.GLFW_CONTEXT_VERSION_MINOR, 1)
        glfw.glfwWindowHint(glfw.GLFW_OPENGL_FORWARD_COMPAT,
 GL_TRUE)
        glfw.glfwWindowHint(glfw.GLFW_OPENGL_PROFILE,
                            glfw.GLFW_OPENGL_CORE_PROFILE)

        # make a window
        self.width, self.height = 800, 600
        self.aspect = self.width/float(self.height)
      ❸ self.win = glfw.glfwCreateWindow(self.width,
 self.height,
                                           b'simpleglfw')
```

```
        # make the context current
    ❹ glfw.glfwMakeContextCurrent(self.win)
```

You initialize the GLFW library ❶, and then, starting at ❷, you set the OpenGL version to the OpenGL 4.1 core profile. You next create an OpenGL-capable window with the dimensions 800×600 ❸ . Finally, you make the context current ❹, and you're ready to make OpenGL calls.

Next, still within the `__init__()` definition, you make some initialization calls:

```
        # initialize GL
    ❶ glViewport(0, 0, self.width, self.height)
    ❷ glEnable(GL_DEPTH_TEST)
    ❸ glClearColor(0.5, 0.5, 0.5, 1.0)
```

Here you set the viewport or screen dimensions (width and height) where OpenGL will render your 3D scene ❶. Then you turn on depth testing with `GL_DEPTH_TEST` ❷ and set the color the background should become when `glClear()` is issued during rendering ❸. You choose 50 percent gray with an `alpha` setting of 1.0. (Alpha is a measure of the transparency of a fragment—1.0 means fully opaque.)

Setting Callbacks

You finish the `__init__()` definition by registering event callbacks for user interface events within the GLFW window so you can respond to keypresses:

```
        # set window callbacks
        glfw.glfwSetKeyCallback(self.win, self.onKeyboard)
```

This code sets callbacks for keypresses. Every time one of these events happens, the function registered as a callback, `onKeyboard()`, is executed. Let's look at the definition of that keyboard callback function now:

```
def onKeyboard(self, win, key, scancode, action, mods):
    # print 'keyboard: ', win, key, scancode, action, mods
  ❶ if action == glfw.GLFW_PRESS:
        # ESC to quit
        if key == glfw.GLFW_KEY_ESCAPE:
          ❷ self.exitNow = True
        else:
            # toggle cut
          ❸ self.scene.showCircle = not self.scene.showCircle
```

The `onKeyboard()` callback is called every time a keyboard event happens. The arguments to the function arrive filled with useful information such as what type of event occurred (key-up versus key-down, for example) and which key was pressed. The code `glfw.GLFW_PRESS` says to look only for key-down, or `PRESS`, events ❶. You set an exit flag if the `ESC` key is pressed ❷ . If any other key is pressed, you toggle a `showCircle` Boolean ❸ . This variable will be used in the fragment shader to keep or discard fragments outside the circle area.

Defining the Main Loop

The `RenderWindow` class also defines the main loop of the program through its `run()` method. (GLFW doesn't provide a default program loop.) The `run()` method updates the OpenGL window at a preset time interval. After calling the render methods to draw the scene, it also polls the system for any pending window or keyboard events. Let's look at the method definition:

```
def run(self):
    # initializer timer
  ❶ glfw.glfwSetTime(0)
    t = 0.0
  ❷ while not glfw.glfwWindowShouldClose(self.win) and not
  self.exitNow:
        # update every x seconds
      ❸ currT = glfw.glfwGetTime()
        if currT - t > 0.1:
            # update time
            t = currT
            # clear
          ❹ glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
            # set viewport
          ❺ self.width, self.height =
  glfw.glfwGetFramebufferSize(self.win)
          ❻ self.aspect = self.width/float(self.height)
          ❼ glViewport(0, 0, self.width, self.height)
```

In the main loop, `glfw.glfwSetTime()` resets the GLFW timer to 0 ❶. You'll use this timer to redraw the graphics at regular intervals. You initiate a `while` loop ❷ that exits only if the window is closed or `exitNow` is set to `True`. When the loop exits, `glfw.glfwTerminate()` is called to shut down GLFW cleanly.

Inside the loop, `glfw.glfwGetTime()` gets the current timer value ❸, which you use to calculate the elapsed time since the last drawing. By setting a desired interval here (in this case, to 0.1 second or 100 milliseconds), you can adjust the rendering frame rate. Next, `glClear()` clears the depth and color buffers and replaces them with the set background color to get ready for the next frame ❹.

You query and set the window width and height using the `glfwGetFramebufferSize()` function ❺. You do this in case the user has changed the window's size. Note that in some systems (such as a MacBook with Retina display) the

window size and frame buffer size can be different, so to be safe, always query the latter. You next compute the aspect ratio of the window ❻, which you'll use later to set the projection matrix. Then you clear the viewport using the new frame buffer dimensions you retrieved ❼.

Now let's look at the remaining part of the `run()` method:

```
          # build projection matrix
        ❶ pMatrix = glutils.perspective(45.0, self.aspect,
0.1, 100.0)

        ❷ mvMatrix = glutils.lookAt([0.0, 0.0, -2.0], [0.0,
0.0, 0.0],
                                    [0.0, 1.0, 0.0])
          # render
        ❸ self.scene.render(pMatrix, mvMatrix)
          # step
        ❹ self.scene.step()

        ❺ glfw.glfwSwapBuffers(self.win)
          # poll for and process events
        ❻ glfw.glfwPollEvents()
  # end
  glfw.glfwTerminate()
```

Still within the `while` loop, you compute the projection matrix using the `perspective()` method defined in *glutils.py* ❶. The projection matrix is the transformation that maps a 3D scene to a 2D screen. Here you ask for a 45-degree field of view and a near/far plane distance of 0.1/100.0. Then you set the modelview matrix using the `lookAt()` method ❷, also defined in *glutils.py*. The default OpenGL view puts your eye at the origin looking along the negative z-direction. The modelview matrix created by the `lookAt()` method transforms the vertices such that the view matches up with the eye position and orientation specified by the call. You set the eye position to (0, 0, −2), looking at the origin (0,

0, 0) with an “up” vector of (0, 1, 0). Next, you call the `render()` method on the `scene` object ❸, passing in these matrices, and you call `scene.step()` so it can update the variables necessary for the time step ❹. (The `Scene` class, which we’ll look at next, encapsulates the setup and rendering of the polygon.) The `glfwSwapBuffers()` call ❺ swaps the back and front buffers, thus displaying your updated 3D graphic, and the `glfwPollEvents()` call ❻ checks for any UI events and returns control to the `while` loop.

The Scene Class

Now let’s look at the `Scene` class, which is responsible for initializing and drawing the 3D geometry. Here’s the start of the class declaration:

```
class Scene:
    """ OpenGL 3D scene class"""
    # initialization
    def __init__(self):
        # create shader
      ❶ self.program = glutils.loadShaders(strVS, strFS)
      ❷ glUseProgram(self.program)
```

In the `Scene` class constructor, you first compile and load the shaders. For this, you use the utility method `loadShaders()` ❶ defined in *glutils.py*, which provides a convenient wrapper around the series of OpenGL calls required to load the shader code from strings, compile it, and link it into an OpenGL program object. Because OpenGL is a state machine, you need to set the code to use a particular “program object” (because a project could have multiple programs) using the `glUseProgram()` call ❷.

The `__init__()` method continues by connecting the variables in the Python code with those in the shaders:

```
        self.pMatrixUniform =
    glGetUniformLocation(self.program, b'uPMatrix')
        self.mvMatrixUniform =
    glGetUniformLocation(self.program, b'uMVMatrix')
        # texture
        self.tex2D = glGetUniformLocation(self.program,
    b'tex2D')
```

This code uses the `glGetUniformLocation()` method to retrieve the locations of the variables `uPMatrix`, `uMVMatrix`, and `tex2D` defined inside the vertex and fragment shaders. These locations can then be used to set the values for the shader variables.

Defining the 3D Geometry

The next part of the `Scene` class's `__init__()` method defines the 3D geometry for the scene. You first define the geometry for the polygon, which will take the form of a square:

```
        # define triangle strip vertices
      ❶ vertexData = numpy.array(
            [-0.5, -0.5, 0.0,
             0.5, -0.5, 0.0,
             -0.5, 0.5, 0.0,
             0.5, 0.5, 0.0], numpy.float32)

        # set up vertex array object (VAO)
      ❷ self.vao = glGenVertexArrays(1)
        glBindVertexArray(self.vao)
        # vertices
      ❸ self.vertexBuffer = glGenBuffers(1)
        glBindBuffer(GL_ARRAY_BUFFER, self.vertexBuffer)
        # set buffer data
      ❹ glBufferData(GL_ARRAY_BUFFER, 4*len(vertexData),
  vertexData,
```

```
                    GL_STATIC_DRAW)
        # enable vertex array
    ❺ glEnableVertexAttribArray(0)
        # set buffer data pointer
    ❻ glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
None)
        # unbind VAO
    ❼ glBindVertexArray(0)
```

First you define the array of vertices of the triangle strip used to draw the square ❶. Think of a square of a side length of 1.0 centered at the origin. The bottom-left vertex of this square has the coordinates (−0.5, −0.5, 0.0); the next vertex (the bottom-right one) has the coordinates (0.5, −0.5, 0.0); and so on. The order of the four coordinates is that of a `GL_TRIANGLE_STRIP`. Essentially, you're creating the square by defining two right triangles with a shared hypotenuse.

Next, you create a VAO ❷. Once you bind to this VAO, all upcoming calls will be bound to it. You then create a VBO to manage the rendering of the vertex data ❸. Once the buffer is bound, you set the buffer data from the vertices you've defined ❹.

Now you need to enable the shaders to access this data. For that, you call `glEnableVertexAttribArray()` ❺. You use an index of 0 because that is the location you have set in the vertex shader for the vertex data variable. Calling `glVertexAttribPointer()` sets the location and data format of the vertex attribute array ❻. The index of the attribute is 0, the number of components is 3 (you use 3D vertices), and the data type of the vertex is `GL_FLOAT`. You then unbind the VAO ❼ so other related calls don't interfere with it. In OpenGL, it's a best practice to reset states when you're done. OpenGL is a state machine, so if you leave things in a mess, they will remain that way.

The following code loads an image of a star as an OpenGL texture:

```
        # texture
        self.texId = glutils.loadTexture('star.png')
```

The texture ID returned will be used later in rendering.

Rotating the Square

Next you need to update variables in the `Scene` object to make the square rotate on the screen. Use the class's `step()` method:

```
# step
def step(self):
    # increment angle
  ❶ self.t = (self.t + 1) % 360
```

At ❶, you increment the angle variable `t` and use the modulus operator (`%`) to keep this value within [0, 360]. This variable will be used to update the rotation angle in the vertex shader.

Rendering the Scene

Now let's look at the `Scene` object's main rendering code:

```
def render(self, pMatrix, mvMatrix):
    # use shader
  ❶ glUseProgram(self.program)

    # set projection matrix
  ❷ glUniformMatrix4fv(self.pMatrixUniform, 1, GL_FALSE,
  pMatrix)
```

```
    # set modelview matrix
    glUniformMatrix4fv(self.mvMatrixUniform, 1, GL_FALSE,
mvMatrix)
    # set shader angle in radians
❸   glUniform1f(glGetUniformLocation(self.program, 'uTheta'),
                math.radians(self.t))
    # show circle?
❹   glUniform1i(glGetUniformLocation(self.program,
b'showCircle'),
                self.showCircle)

    # enable texture
❺   glActiveTexture(GL_TEXTURE0)
❻   glBindTexture(GL_TEXTURE_2D, self.texId)
❼   glUniform1i(self.tex2D, 0)

    # bind VAO
❽   glBindVertexArray(self.vao)
    # draw
❾   glDrawArrays(GL_TRIANGLE_STRIP, 0, 4)
    # unbind VAO
❿   glBindVertexArray(0)
```

First you set up the rendering to use the shader program ❶. Starting at ❷, you set the computed projection and modelview matrices in the shader using the `glUniformMatrix4fv()` method. Then you use the `glUniform1f()` method to set `uTheta` in the shader program ❸. You use `glGetUniformLocation()` as before to get the location of the `uTheta` angle variable from the shader, and you use the Python `math.radians()` method to convert the angle from degrees to radians. Next, you use `glUniform1i()` to set the current value of the `showCircle` variable in the fragment shader ❹. OpenGL has a concept of multiple texture units, and `glActiveTexture()` ❺ activates texture unit 0 (the default). You bind the texture ID you generated earlier from the *star.png* image to activate it for rendering ❻. The `sampler2D` variable in the fragment shader is set to texture unit 0 ❼.

You continue by binding to the VAO you created previously ❽. Now you see the benefit of using VAOs: you don't need to repeat a whole bunch of vertex buffer–related calls before the actual drawing. You then call `glDrawArrays()` to render the bound vertex buffers ❾. The primitive type is a triangle strip, and there are four vertices to be rendered. Finally, you unbind the VAO at ❿, which is always a good coding practice.

Defining the GLSL Shaders

Now let's look at the most exciting part of the project—the GLSL shaders. First, here's the vertex shader, which computes the position and texture coordinates of the vertices:

```
# version 410 core

layout(location = 0) in vec3 aVert;

uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform float uTheta;

out vec2 vTexCoord;

void main() {
    // rotational transform
  ❹ mat4 rot = mat4(
               vec4(1.0, 0.0, 0.0, 0.0),
               vec4(0.0, cos(uTheta), -sin(uTheta), 0.0),
               vec4(0.0, sin(uTheta),  cos(uTheta), 0.0),
               vec4(0.0, 0.0,          0.0,         1.0)
               );
    // transform vertex
  ❺ gl_Position = uPMatrix * uMVMatrix * rot * vec4(aVert,
  1.0);
    // set texture coordinate
  ❻ vTexCoord = aVert.xy + vec2(0.5, 0.5);
```

```
}
```

You use the `layout` keyword ❶ to set explicitly the location of the vertex attribute `aVert`—to 0, in this case. This attribute lets the vertex shader access the vertices that you defined for the polygon. Starting at ❷, you declare three `uniform` variables for the projection and modelview matrices and the rotation angle. These will be set from the Python code. You also set a 2D vector `vTexCoord` as an output from this shader ❸. This will be available as an input to the fragment shader.

In the `main()` method in the shader, you set up a rotation matrix ❹, which rotates around the x-axis by a given angle, `uTheta`. You compute `gl_Position` ❺ using a concatenation of the projection, modelview, and rotation matrices. This gives you the position of the output vertex from the shader. You then set up a 2D vector as a texture coordinate ❻. You may recall that you defined the triangle strip for a square centered at the origin with side 1.0. Because texture coordinates are in the range [0, 1], you can generate these from the vertex coordinates by adding (0.5, 0.5) to the x- and y-values. This also demonstrates the power and immense flexibility of shaders for your computations. Texture coordinates and other variables aren't sacrosanct; you can set them to just about anything.

Now let's look at the fragment shader, which computes the output pixels of our OpenGL program:

```
# version 410 core

in vec2 vTexCoord;

uniform sampler2D tex2D;
uniform bool showCircle;

out vec4 fragColor;

void main() {
    if (showCircle) {
```

```
        // discard fragment outside circle
      ❺ if (distance(vTexCoord, vec2(0.5, 0.5)) > 0.5) {
            discard;
        }
        else {
          ❻ fragColor = texture(tex2D, vTexCoord);
        }
    }
        else {
          ❼ fragColor = texture(tex2D, vTexCoord);
        }
}
```

You start by defining inputs to the fragment shader—in this case the texture coordinate you set as output in the vertex shader ❶. Recall that the fragment shader operates on a per-pixel basis, so the values set for these variables are those for the current pixel, interpolated across the polygon. You declare a `sampler2D` variable ❷, which is linked to a particular texture unit and is used to look up the texture value, and a Boolean uniform flag `showCircle` ❸, which is set from the Python code. You also declare `fragColor` as the output from the fragment shader ❹. By default, this goes to the screen (after final frame buffer operations such as depth testing and blending).

Within the `main()` method, if the `showCircle` flag is not set ❼, you use the GLSL `texture()` method to look up the texture color value using the texture coordinate and the sampler. In effect, you're just texturing the triangle strip using the star image. If, however, the `showCircle` flag is `true` ❺, you use the GLSL built-in method `distance()` to check how far the current pixel is from the center of the polygon. It uses the (interpolated) texture coordinates for this purpose, which are passed in by the vertex shader. If the distance is greater than a certain threshold (0.5 in this case), you call the GLSL `discard()` method, which drops the current pixel. If the

distance is less than the threshold, you set the appropriate color from the texture ❻. Basically, what this does is ignore pixels that are outside a circle with a radius of 0.5 centered at the midpoint of the square, thus cutting the polygon into a circle when `showCircle` is set.

Utility Functions

I've referred to several utility functions defined for you in *glutils.py* to make working with OpenGL easier. Let's look at an example of one of those functions now. The `loadTexture()` function loads an image into an OpenGL texture:

```
def loadTexture(filename):
    """load OpenGL 2D texture from given image file"""
  ❶ img = Image.open(filename)
  ❷ imgData = numpy.array(list(img.getdata()), np.int8)
  ❸ texture = glGenTextures(1)
  ❹ glBindTexture(GL_TEXTURE_2D, texture)
  ❺ glPixelStorei(GL_UNPACK_ALIGNMENT, 1)
  ❻ glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
  GL_CLAMP_TO_EDGE)
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
  GL_CLAMP_TO_EDGE)
  ❼ glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
  GL_LINEAR)
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
  GL_LINEAR)
  ❽ glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, img.size[0],
  img.size[1],
                 0, GL_RGBA, GL_UNSIGNED_BYTE, imgData)
    return texture
```

The `loadTexture()` function uses the Python Imaging Library (PIL) `Image` module to read the image file ❶. Then it gets the data out of the `Image` object onto an 8-bit `numpy` array ❷ and creates an OpenGL `texture` object ❸, which is a prerequisite

to doing anything with textures in OpenGL. You next perform the now familiar binding to the `texture` object ❹ so all further texture-related settings apply to this object. You set the unpacking alignment of data to 1 ❺, which means the image data will be considered to be 1-byte or 8-bit data by the hardware. Starting at ❻, you tell OpenGL what to do with the texture at the edges. In this case, you direct it to just clamp the texture color to the edge of the geometry. (In specifying texture coordinates, the convention is to use the letters `S` and `T` for the axes instead of `x` and `y`.) At ❼ and the following line, you specify the kind of interpolation to be used when the texture is stretched or compressed to map onto a polygon. In this case, *linear filtering* is specified. Finally, you set the image data in the bound texture ❽. At this point, the image data is transferred to graphics memory, and the texture is ready for use.

Running the OpenGL Application

Here is a sample run of the project:

```
$ python simpleglfw.py
```

You saw the output in Figure 9-1. Be sure to try some keypresses to toggle the circle on and off.

Summary

Congratulations on completing your first program using Python and OpenGL! Through this project, you've learned about creating 3D transformations, using the OpenGL 3D graphics pipeline, and using GLSL vertex and fragment shaders to create interesting 3D graphics. You've begun

your journey into the fascinating world of 3D graphics programming.

Experiments!

Here are some ideas for modifying this project:

1. The vertex shader in this project rotates the square around the x-axis (1, 0, 0). Can you make it rotate around the y-axis (0, 0, 1)? You can do this in one of two ways: first, by modifying the rotation matrix in the shader, or second, by computing this matrix in the Python code and passing it as a *uniform* into the shader. Try both!
2. In the project, the texture coordinates are generated inside the vertex shader and passed to the fragment shader. This is a trick, and it works only because of the convenient values chosen for the vertices of the triangle strip. Pass the texture coordinates as a separate attribute into the vertex shader, similar to how the vertices are passed in. Now, can you make the star texture *tile* across the triangle strip? Instead of displaying a single star, you want to produce a 4×4 grid of stars on the square. (Hint: use texture coordinates greater than 1.0 and set `GL_TEXTURE_WRAP_S/T` parameters in `glTexParameterf()` to `GL_REPEAT`.)
3. By changing just your fragment shader, can you make your square look like Figure 9-6? (Hint: use the GLSL `sin()` function.)

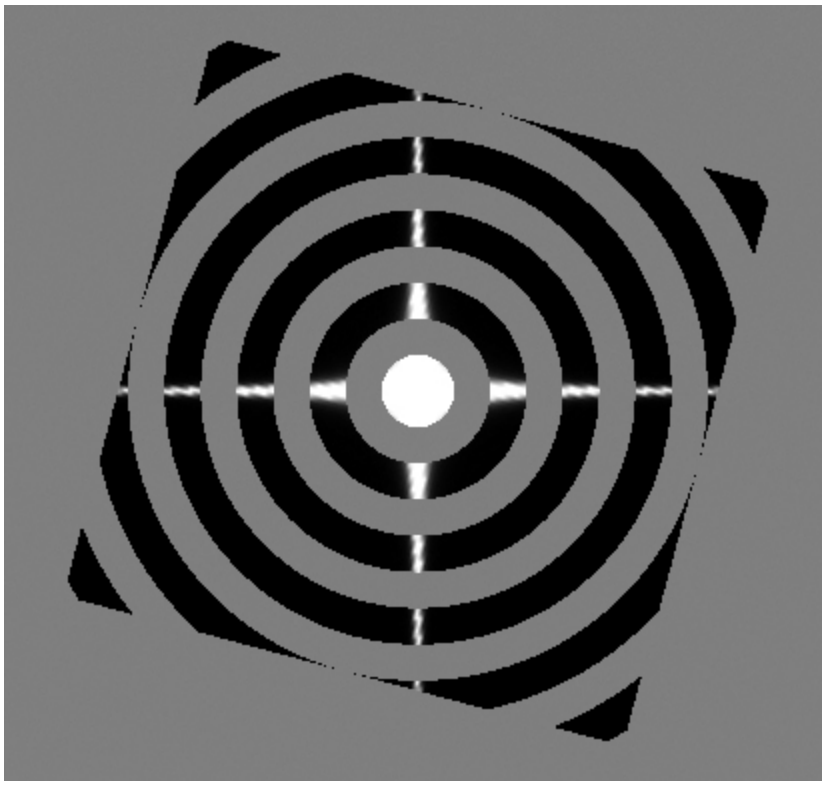


Figure 9-6: *Using the fragment shader to block out concentric circles*

The Complete Code

Here's the complete *simpleglfw.py* code:

```
"""
simpleglfw.py

A simple Python OpenGL program that uses PyOpenGL + GLFW to get
  an
OpenGL 4.1 context.

Author: Mahesh Venkitachalam
"""

import OpenGL
from OpenGL.GL import *

import numpy, math, sys, os
import glutils
```

```
import glfw

strVS = """
# version 410 core

layout(location = 0) in vec3 aVert;

uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform float uTheta;

out vec2 vTexCoord;

void main() {
  // rotational transform
  mat4 rot =  mat4(
        vec4(1.0,  0.0,          0.0,         0.0),
        vec4(0.0,  cos(uTheta),  sin(uTheta), 0.0),
        vec4(0.0, -sin(uTheta),  cos(uTheta), 0.0),
        vec4(0.0,  0.0,          0.0,         1.0)
        );
  // transform vertex
  gl_Position = uPMatrix * uMVMatrix * rot * vec4(aVert, 1.0);
  // set texture coord
  vTexCoord = aVert.xy + vec2(0.5, 0.5);
}
"""

strFS = """
# version 410 core

in vec2 vTexCoord;

uniform sampler2D tex2D;
uniform bool showCircle;

out vec4 fragColor;

void main() {
  if (showCircle) {
    // discard fragment outside circle
    if (distance(vTexCoord, vec2(0.5, 0.5)) > 0.5) {
      discard;
    }
    else {
```

```
      fragColor = texture(tex2D, vTexCoord);
    }
  }
  else {
    fragColor = texture(tex2D, vTexCoord);
  }
}
"""

class Scene:
    """ OpenGL 3D scene class"""
    # initialization
    def __init__(self):
        # create shader
        self.program = glutils.loadShaders(strVS, strFS)
        glUseProgram(self.program)
        self.pMatrixUniform =
  glGetUniformLocation(self.program,
                                                b'uPMatrix')
        self.mvMatrixUniform =
  glGetUniformLocation(self.program,
                                                b'uMVMatrix')
        # texture
        self.tex2D = glGetUniformLocation(self.program,
  b'tex2D')
        # define triangle strip vertices
        vertexData = numpy.array(
            [-0.5, -0.5, 0.0,
              0.5, -0.5, 0.0,
              -0.5, 0.5, 0.0,
              0.5, 0.5, 0.0], numpy.float32)
        # set up vertex array object (VAO)
        self.vao = glGenVertexArrays(1)
        glBindVertexArray(self.vao)
        # vertices
        self.vertexBuffer = glGenBuffers(1)
        glBindBuffer(GL_ARRAY_BUFFER, self.vertexBuffer)
        # set buffer data
```

```
        glBufferData(GL_ARRAY_BUFFER, 4*len(vertexData),
vertexData,
                     GL_STATIC_DRAW)
        # enable vertex array
        glEnableVertexAttribArray(0)
        # set buffer data pointer
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
None)
        # unbind VAO
        glBindVertexArray(0)

        # time
        self.t = 0

        # texture
        self.texId = glutils.loadTexture('star.png')

        # show circle?
        self.showCircle = False

    # step
    def step(self):
        # increment angle
        self.t = (self.t + 1) % 360

    # render
    def render(self, pMatrix, mvMatrix):
        # use shader
        glUseProgram(self.program)

        # set proj matrix
        glUniformMatrix4fv(self.pMatrixUniform, 1, GL_FALSE,
pMatrix)

        # set modelview matrix
        glUniformMatrix4fv(self.mvMatrixUniform, 1, GL_FALSE,
mvMatrix)

        # set shader angle in radians
        glUniform1f(glGetUniformLocation(self.program,
'uTheta'),
                    math.radians(self.t))

        # show circle?
```

```
        glUniform1i(glGetUniformLocation(self.program,
  b'showCircle'),
                    self.showCircle)

        # enable texture
        glActiveTexture(GL_TEXTURE0)
        glBindTexture(GL_TEXTURE_2D, self.texId)
        glUniform1i(self.tex2D, 0)

        # bind VAO
        glBindVertexArray(self.vao)
        # draw
        glDrawArrays(GL_TRIANGLE_STRIP, 0, 4)
        # unbind VAO
        glBindVertexArray(0)

class RenderWindow:
    """GLFW rendering window class"""
    def __init__(self):

        # save current working directory
        cwd = os.getcwd()

        # initialize glfw - this changes cwd
        glfw.glfwInit()

        # restore cwd
        os.chdir(cwd)

        # version hints
        glfw.glfwWindowHint(glfw.GLFW_CONTEXT_VERSION_MAJOR, 4)
        glfw.glfwWindowHint(glfw.GLFW_CONTEXT_VERSION_MINOR, 1)
        glfw.glfwWindowHint(glfw.GLFW_OPENGL_FORWARD_COMPAT,
  GL_TRUE)
        glfw.glfwWindowHint(glfw.GLFW_OPENGL_PROFILE,
                            glfw.GLFW_OPENGL_CORE_PROFILE)

        # make a window
        self.width, self.height = 800, 600
        self.aspect = self.width/float(self.height)
        self.win = glfw.glfwCreateWindow(self.width,
  self.height,
                                         b'simpleglfw')
        # make context current
```

```
        glfw.glfwMakeContextCurrent(self.win)

        # initialize GL
        glViewport(0, 0, self.width, self.height)
        glEnable(GL_DEPTH_TEST)
        glClearColor(0.5, 0.5, 0.5, 1.0)

        # set window callbacks
        glfw.glfwSetKeyCallback(self.win, self.onKeyboard)

        # create 3D
        self.scene = Scene()

        # exit flag
        self.exitNow = False

    def onKeyboard(self, win, key, scancode, action, mods):
        # print 'keyboard: ', win, key, scancode, action, mods
        if action == glfw.GLFW_PRESS:
            # ESC to quit
            if key == glfw.GLFW_KEY_ESCAPE:
                self.exitNow = True
            else:
                # toggle cut
                self.scene.showCircle = not
self.scene.showCircle

    def run(self):
        # initializer timer
        glfw.glfwSetTime(0)
        t = 0.0
        while not glfw.glfwWindowShouldClose(self.win) and not
self.exitNow:
            # update every x seconds
            currT = glfw.glfwGetTime()
            if currT - t > 0.1:
                # update time
                t = currT
                # clear
                glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT)

                # set viewport
```

```
                self.width, self.height =
                                    glfw.glfwGetFramebufferSize
  (self.win)
                self.aspect = self.width/float(self.height)
                glViewport(0, 0, self.width, self.height)

                # build projection matrix
                pMatrix = glutils.perspective(45.0,
  self.aspect, 0.1, 100.0)

                mvMatrix = glutils.lookAt([0.0, 0.0, -2.0],
  [0.0, 0.0, 0.0],
                                          [0.0, 1.0, 0.0])
                # render
                self.scene.render(pMatrix, mvMatrix)

                # step
                self.scene.step()

                glfw.glfwSwapBuffers(self.win)
                # poll for and process events
                glfw.glfwPollEvents()
        # end
        glfw.glfwTerminate()

    def step(self):
        # step
        self.scene.step()

# main() function
def main():
    print("Starting simpleglfw. "
          "Press any key to toggle cut. Press ESC to quit.")
    rw = RenderWindow()
    rw.run()

# call main
if __name__ == '__main__':
    main()
```

10
CONWAY'S GAME OF LIFE ON A TORUS



In Chapter 3, we implemented Conway's Game of Life using Python and the `matplotlib` library. You may recall one interesting aspect of that project: it used toroidal boundary conditions. Figure 3-2 on page 48 showed how we were effectively treating the flat, 2D grid of the simulation as a 3D toroidal surface, thanks to how the boundary conditions stitched the edges together. In the previous chapter, you were introduced to OpenGL and learned how to render 3D objects. Let's now put together your experience with the Game of Life and OpenGL to re-create the Conway's simulation in 3D, on an actual torus.

In this project, you'll start by computing the 3D geometry of a torus. Then you'll arrange the vertices of the torus in a way that's easy to draw and color in OpenGL. You'll set up a revolving camera to view the torus from all sides and also implement some basic lighting in the shaders. Finally, you'll adapt the Game of Life code from Chapter 3 to color the grid on the torus. As the simulation proceeds, you'll see the Game of Life come alive on the surface of the torus!

Here are the main topics covered in this project:

- Using matrix math to construct the 3D geometry of a torus
- Implementing a coloring scheme for the Game of Life grid on the torus
- Implementing a revolving camera in OpenGL
- Implementing simple lighting in OpenGL

How It Works

Before getting into the code, let's consider how you can render, light, and view a 3D torus using OpenGL. It all begins with computing the vertices that make up the torus.

Computing Vertices

A torus is essentially a collection of circles, or rings, arranged in a circle around a central point. However, you can't actually draw circles with OpenGL; they need to be *discretized*, or represented as a series of vertices connected by straight lines. The simplified model in Figure 10-1 illustrates how you can begin defining the torus as a collection of vertices.

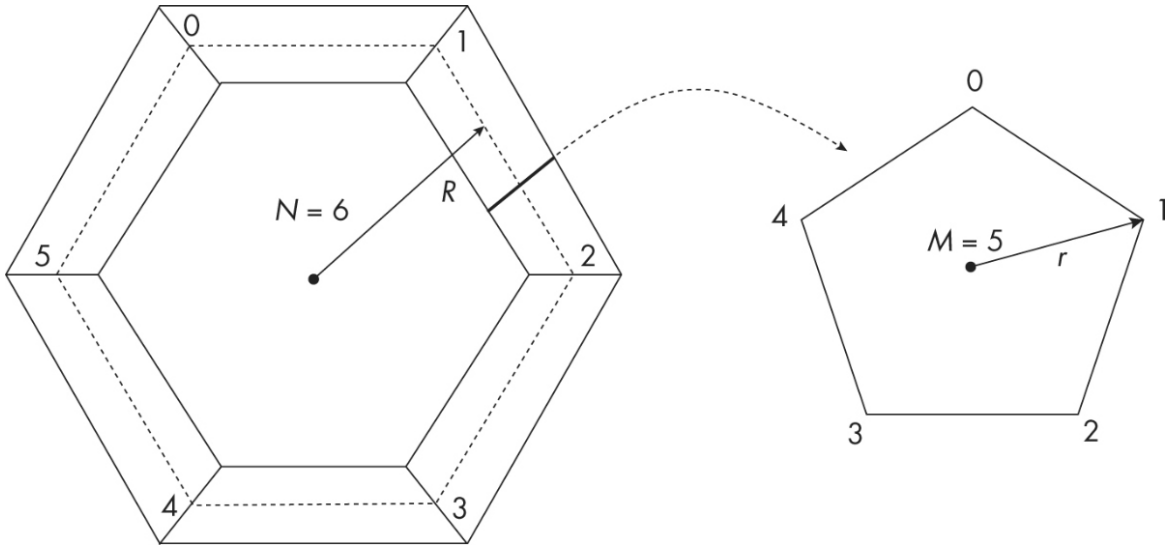


Figure 10-1: *A torus rendering model. The simplified torus is on the left. An individual "ring" that makes up the torus is on the right.*

The right side of Figure 10-1 shows a ring with radius r discretized to have $M = 5$ points. The left side of Figure 10-1 models a simple torus with major radius R by arranging $N = 6$ such discretized rings (labeled 0 through 5) around a central point. (The *major radius* is the distance from the center of the torus's hole to the center of its outer ring.) Don't worry about the blocky nature of the torus shown in the figure. It gets smoother as you increase the values of N and M.

The torus is filled out by drawing bands to connect adjacent rings. You'll draw the bands using `GL_TRIANGLE_STRIP` primitives, and each cell in the Game of Life simulation will consist of two adjacent triangles on a strip, which together form a *quad*, or quadrilateral. When a cell is ON you'll color its quad black, and when a cell is OFF you'll color it white.

To compute the vertices of the torus, you first need to define a coordinate system for it. Let's assume that the torus is on the XY plane, centered at the origin, with the line through the center of the torus aligned along the z-axis, as shown in Figure 10-2.

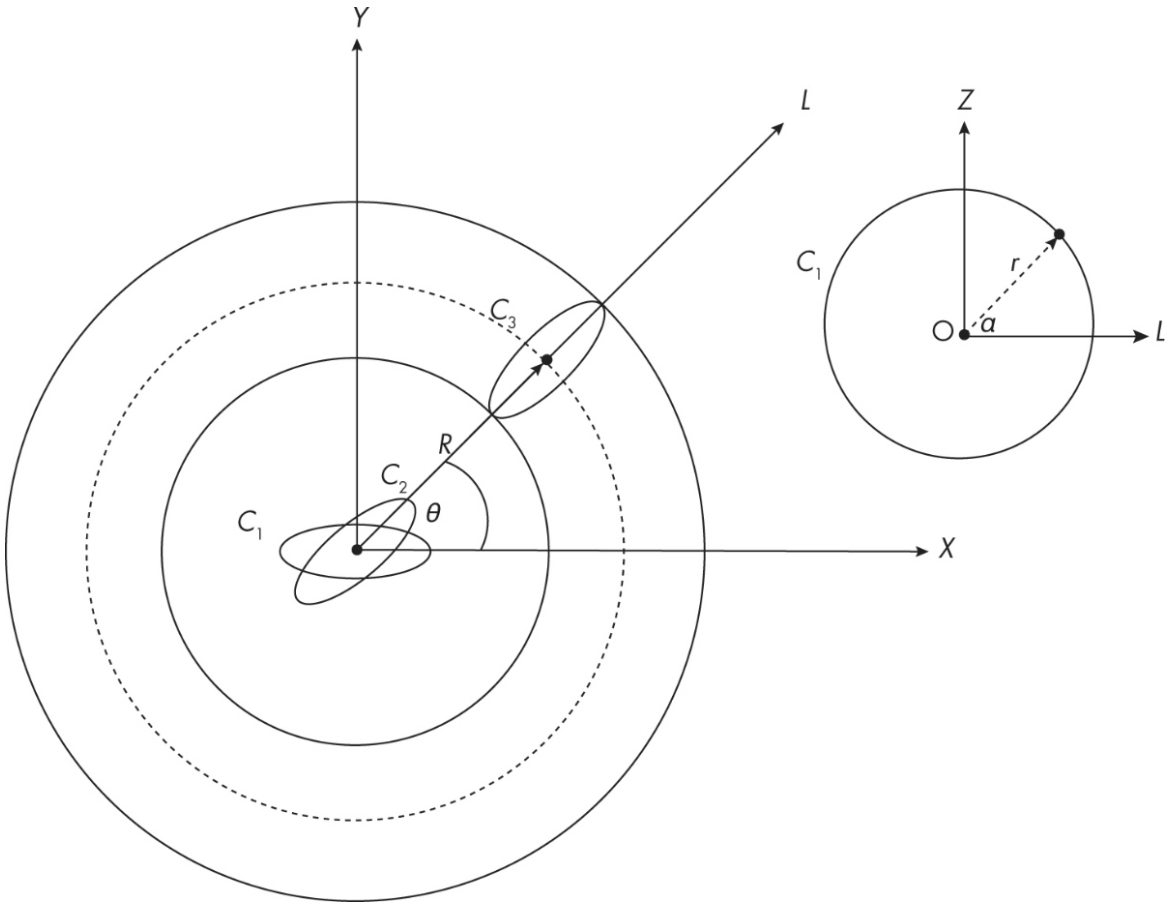


Figure 10-2: *The rendering strategy for the torus*

The vertices for a circle C_3 on the torus at an angle θ relative to the x-axis can be computed as follows:

1. Compute the vertices for a circle C_1 of radius r in the XZ plane and with the center at the origin.
2. Rotate the circle C_1 around the z-axis by the angle θ. This gives you the circle C_2.
3. Translate the circle C_2 by R amount along the angle θ to get the circle C_3 in the correct position on the torus.

You may recall the use of parametric equations to define a circle in the spirograph project in Chapter 2. We're going to

use the same concept here. The vertices marking the circumference of a circle C_1 of radius r on the XZ plane centered at the origin are given by:

$$P = (r\cos(\alpha), 0, r\sin(\alpha))$$

Here, α is the angle made by the point P with respect to the x-axis. As α varies from 0 to 360 degrees (or 2π radians), a circle will be formed by the P points. Notice that the y-coordinates of the points are zero in the previous equation. This is expected, since the circle is on the XZ plane.

You must now rotate the points around the z-axis by angle θ. The rotation matrix for this operation is given by:

$$R_{\theta,z} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0.0 & 0.0 \\ \sin(\theta) & \cos(\theta) & 0.0 & 0.0 \\ 0.0 & \cos(\theta) & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}$$

Once you rotate the points, you need to translate them to the right place. This is done using the following translation matrix. (This format was discussed in Chapter 9.)

$$T = \begin{pmatrix} 1.0 & 0.0 & 0.0 & R\cos(\theta) \\ 0.0 & 1.0 & 0.0 & R\sin(\theta) \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}$$

Thus, the transformed points on a ring on the torus are given by:

$$P' = T \times R_{\theta,Z} \times P$$

which is the same as:

$$P' = \begin{pmatrix} 1.0 & 0.0 & 0.0 & R\cos(\theta) \\ 0.0 & 1.0 & 0.0 & R\sin(\theta) \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix} \cdot \begin{pmatrix} 1.0 & 0.0 & 0.0 & R\cos(\theta) \\ 0.0 & 1.0 & 0.0 & R\sin(\theta) \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix} \cdot \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1.0 \end{pmatrix}$$

In the previous equation, P is first multiplied by the rotation matrix, which aligns it correctly, and is then multiplied by the translation matrix, which "pushes" the points to the right place on the torus. Notice that P is represented with *homogeneous* coordinates (x, y, z, 1.0), discussed in the previous chapter.

Computing Normals for Lighting

For the torus to look nice, you'll need to apply lighting to the geometry, which in turn means you'll need to compute the *normal vectors* for the points P calculated in the previous

section. The lighting on a surface depends on the orientation of the surface to the incoming light, and the orientation can be quantified by the normal vector, which is the vector perpendicular to the surface at a particular point. Take a look at Figure 10-3 to see an example.

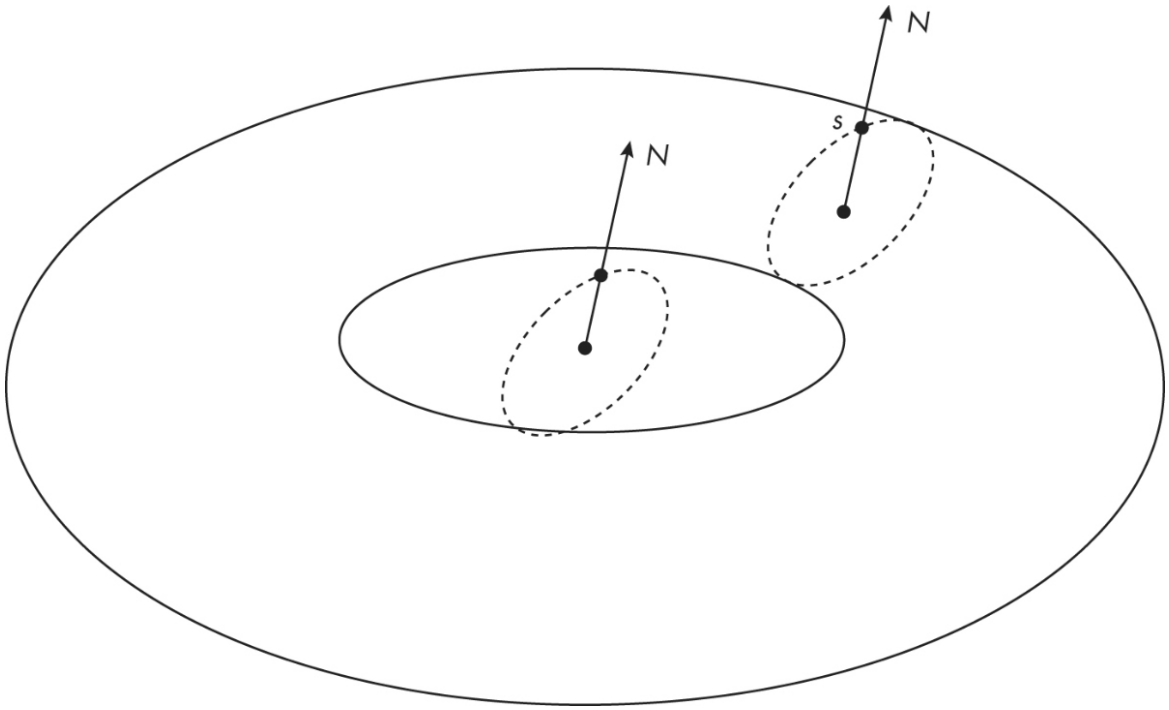


Figure 10-3: *Computing normal vectors*

Because of the geometry of the torus, the normal vector for point *s* on a ring is in the same direction as the line connecting point *s* to the center of the ring. This means that the normal vectors are the same as rotated points. The translation doesn't matter, as the direction of the normals is unaffected by it. You can therefore compute the normals as follows:

$$N = R_{\theta,Z} \times P$$

Note that you need to *normalize* the normal vectors before doing any lighting computations. You do this by

dividing the normal by its magnitude.

The actual lighting will come from a single light source at a fixed position. This will be defined in the vertex shader.

Rendering

Now that you have the vertices and normals for the torus, let's talk about how to render it using OpenGL. You first need to split it up into bands, as shown in Figure 10-4. Each band is the region between two adjacent rings on the torus.

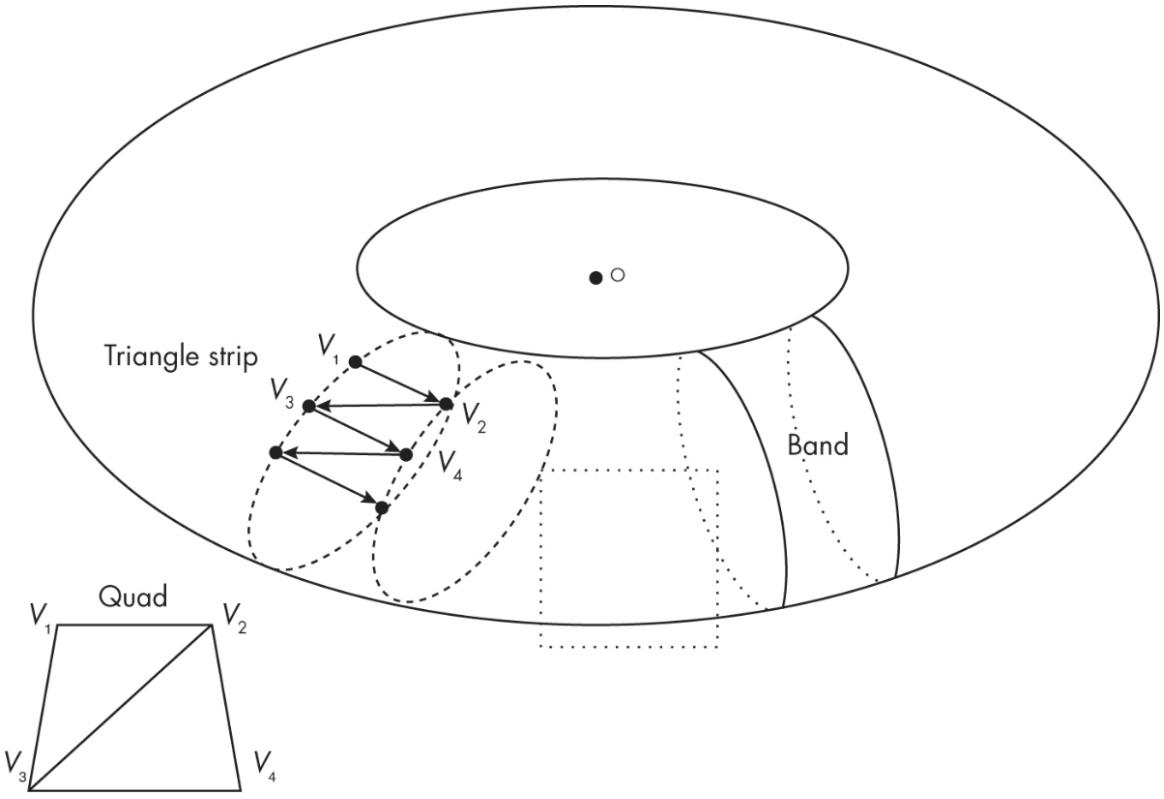


Figure 10-4: *Rendering the torus with triangle strips*

Each of the bands is rendered using OpenGL as a `GL_TRIANGLE_STRIP` primitive. Besides forming the building blocks of the torus, these triangle strips also provide a convenient way to create the Game of Life simulation grid: each cell in the grid is represented by a quad formed from

two adjacent triangles in the strip. Figure 10-5 gives a closer look at one band in the torus.

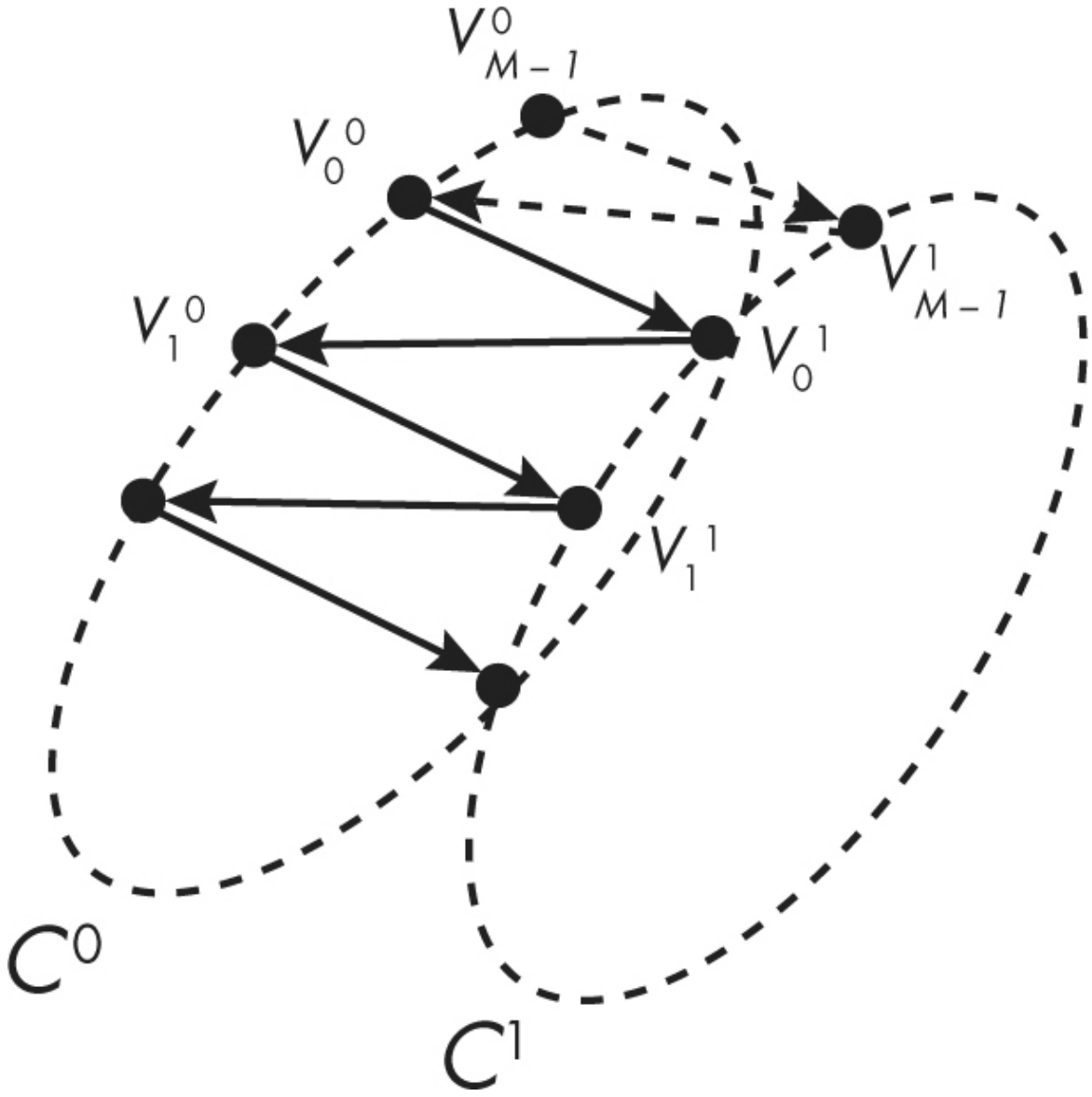


Figure 10-5: *Rendering a band from triangle strips*

The band is formed by the adjacent rings C^0 and C^1. Each ring has M vertices. The triangle strip making up the band is formed as M pairs of vertices that zigzag back and forth between the rings:

$$V_0^0 \rightarrow V_0^1 \rightarrow V_1^0 \rightarrow V_1^1 \ldots, V_{M-1}^0 \rightarrow V_{M-1}^1$$

But there's one additional pair of vertices you need to add: $V_0^0 \rightarrow V_0^1$. You need to repeat the first two vertices to close out the gap at the end of the band. Hence, the total number of vertices in the triangle strip that forms the band is given by $2 \times M + 2$.

The band shown in Figure 10-5 was composed of rings $C^0 \rightarrow C^0$. The torus is divided into N bands, where N is the number of rings:

$$(C^0 \rightarrow C^1 \rightarrow C^1 \rightarrow C^2 \ldots, C^{N-1} \rightarrow C^0)$$

Notice how the last band wraps around by coming back to the first ring, C^0. This means the total number of vertices needed to render the torus is given by $N \times (2 \times M + 2)$. You'll see more details of the implementation as you look at the code.

Now let's take a look at the coloring scheme for the torus.

Coloring the Triangle Strips

You need to color the cells in the Game of Life simulation individually. You know that each cell is a quad—two triangles that are part of a larger triangle strip. As an example, the quad made of vertices $V_0^0 \rightarrow V_0^1 \rightarrow V_1^0 \rightarrow V_1^1$ is composed of two triangles: $V_0^0 \rightarrow V_0^1 \rightarrow V_1^0$ and $V_0^1 \rightarrow V_1^0 \rightarrow V_1^1$. Each vertex has a corresponding color associated with it, which is a triplet of the form (r, g, b), denoting the red, green, and blue components of the color. By default, the color of the

first vertex in the quad (in this case, V_0^0) sets the value for the first triangle in the quad, and the color of the second vertex (V_0^1) sets the value for the second triangle in the quad. As long as you set these two colors to be identical, you'll color the quad uniformly. We'll discuss OpenGL's vertex color convention further when we look at the code.

NOTE *The OpenGL function called* `glProvokingVertex()` *changes the convention of which color value is mapped to the vertex.*

Controlling the Camera

To view the torus, you'll create a camera that revolves around the origin of the 3D scene and looks down at an angle from above. Figure 10-6 shows the camera setup.

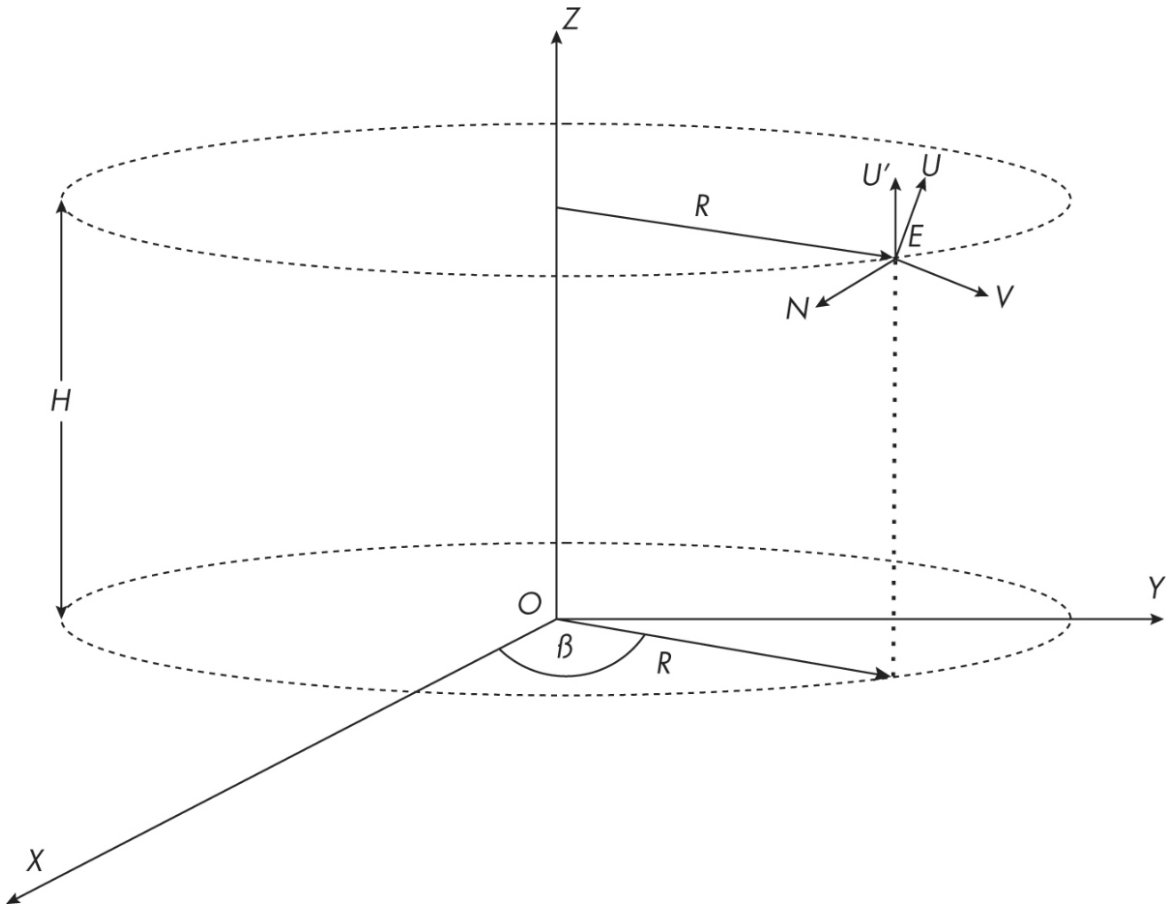


Figure 10-6: *Implementing a revolving camera*

Think of the camera, represented by point *E*, as being placed on a cylinder of radius *R* at a height *H* from the origin. The camera points toward the origin *O*. The camera is defined by the mutually perpendicular vectors *V*, *U*, and *N*, where *V* is the *view vector* that points from *E* to *O*; *U* is the *up vector*, which is up relative to the camera; and *N* is a vector perpendicular to both *V* and *U*. For every time step, you move the camera by a constant distance along the rim of the cylinder. This movement is parameterized by the angle *β*, as shown in Figure 10-6. As you learned in Chapter 9, you use a `lookAt()` method to set up the view, which takes three parameters: the eye, the center, and the up vector. The center is just the origin: (0, 0, 0). The 3D coordinates of the eye are given by:

$$E = (R \cos(\beta), R \sin(\beta), H)$$

As the camera moves along the rim of the cylinder, it will keep its direction pointed to O, and the up vector U will keep changing as well. To compute the up vector U, start with an initial guess U' that's parallel to the z-axis. Then find N, the vector perpendicular to the plane given by U' and V. It can be calculated as:

$$N = V \times U'$$

N is given by the cross product of V and U'. Now, what happens if you take the cross product of N and V? You get a vector perpendicular to the NV plane, which is the up vector U that you're looking for!

$$U = N \times V = (V \times U') \times V$$

Once U is calculated, make sure that it's normalized before use. Once that's done, you'll have everything you need to set up the camera with `lookAt()`: E (the eye), O (the center), and U (the up vector).

Mapping the Grid to the Torus

Finally, let's look at how the 2D Game of Life simulation grid maps to the 3D torus because of the grid's toroidal boundary conditions. Figure 10-7 shows the mapping.

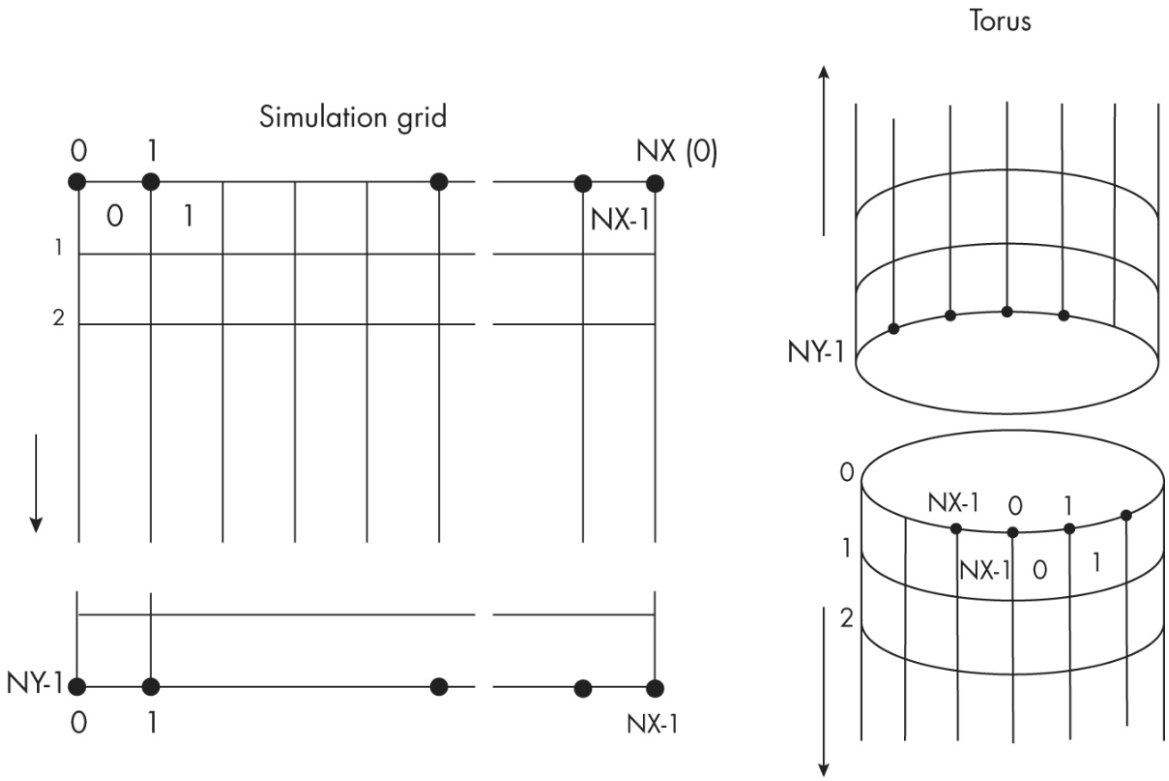


Figure 10-7: *Mapping the simulation grid to a torus*

The two-dimensional Game of Life grid has *NX* columns and *NY* rows. You can see on the right of the figure how the *NX* points that divide each row wrap around the tube of the torus. The point indices go from 0 to $NX - 1$. The next index, *NX*, is the same as 0 because of the wrap. A similar wrap happens in the y-direction, where you have *NY* cells: the point with index *NY* is the same as the point with index 0.

You saw earlier how each ring in the torus is discretized to have *M* points. To go from the two-dimensional grid to the torus, set $NX = M$. Similarly, set $NY = N$, where *N* is the number of bands on the torus.

Requirements

We'll use `PyOpenGL` and GLFW for OpenGL rendering, as in Chapter 9, and `numpy` for matrix/vector computations.

The Code

The code for this project is organized into several files:

torus.py This contains the geometry computation and rendering code for the torus.

gol.py This implements Conway's Game of Life, adapted from Chapter 3.

camera.py This contains the implementation for the revolving camera for viewing the torus.

gol_torus.py This is the main file that sets up OpenGL and GLFW and calls the rendering code from other modules.

The full project code is available online at *https://github.com/mkvenkit/pp2e/blob/main/gol_torus*.

Rendering the Torus

We'll first consider the code for rendering the torus, which is encapsulated in a class called `Torus` defined in the file *torus.py*. To see the complete code listing, skip ahead to "The Complete Torus Rendering Code" on page 203.

Defining the Shaders

First, define the GLSL shaders for the torus. Here's the vertex shader, which takes the per-vertex attributes of position, color, and normal and computes the transformed inputs for the fragment shader:

```
strVS = """
# version 410 core
layout(location = 0) in vec3 aVert;
layout(location = 1) in vec3 aColor;
layout(location = 2) in vec3 aNormal;
```

```
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
flat out vec3 vColor;
out vec3 vNormal;
out vec3 fragPos;
void main() {
  // transform vertex
  gl_Position = uPMatrix * uMVMatrix * vec4(aVert, 1.0);
fragPos = aVert;
  vColor = aColor;
  vNormal = aNormal;
}
"""
```

You define the vertex shader code as a string stored in `strVS`. The attribute variables for the shader are `aVert`, `aColor`, and `aNormal`, representing the coordinates, color, and normal vector for each vertex. Notice the `flat` qualifier when you initialize `vColor`, one of the outputs of the vertex shader ❶, indicating that this variable won't be interpolated in the fragment shader. In effect, we're saying that this variable will remain constant across the primitive (one of the triangles in a triangle strip). This ensures that every Game of Life cell will be a single color. This type of shading across a primitive is called *flat shading*. The next vertex shader output is `vNormal` ❷, which by default will be interpolated in the fragment shader. You need this so you can compute lighting across a primitive, but later you'll see how to modify this shader code to support flat shading. The other output is called `fragPos` ❸. In the main shader code, you set this output to `aVert` ❹ so it can be passed into the fragment shader for lighting calculations. The shader also computes `gl_Position` and passes to the fragment shader the color and normal data as it is received.

Here's the fragment shader, which applies lighting and computes the final color of the fragment. It's defined as

another string, called `strFS`.

```
strFS = """
# version 410 core
flat in vec3 vColor;
in vec3 vNormal;
in vec3 fragPos;
out vec4 fragColor;
void main() {
vec3 lightPos = vec3(10.0, 10.0, 10.0);
vec3 lightColor = vec3(1.0, 1.0, 1.0);
vec3 lightDir = normalize(lightPos - fragPos);
  float diff = max(dot(vNormal, lightDir), 0.0);
  vec3 diffuse = diff * lightColor;
  float ambient = 0.1;
vec3 result = (ambient + diffuse) * vColor.xyz;
  fragColor = vec4(result, 1.0);
}
"""
```

Notice that the color, normal, and fragment position variables, which were outputs from the vertex shader, are now inputs to the fragment shader. In the main shader code, you define the position ❶ and color ❷ for a light source. Then you compute the light direction ❸. The resulting color ❹ is a mix of ambient and diffuse components of the light and is set as the output from the fragment shader.

Keep in mind that `fragPos` and `vNormal` are computed for each fragment via interpolation, whereas `vColor` is constant for a given primitive. The net effect is that the intrinsic color of a primitive (a triangle strip, in this case) remains constant, while the perceived color varies across the primitive based on its orientation with respect to the light source. This is exactly what you need to set each Game of Life cell to a solid color, while varying that color to create the appearance of lighting.

Initializing the Torus Class

Now let's look at the initialization code in the `Torus` class constructor:

```
class Torus:
    """ OpenGL 3D scene class"""
    # initialization
  ❶ def __init__(self, R, r, NX, NY):
        global strVS, strFS
        # modify shader for flat shading
        # create shader
      ❷ self.program = glutils.loadShaders(strVS, strFS)
        glProvokingVertex(GL_FIRST_VERTEX_CONVENTION)
        self.pMatrixUniform =
  glGetUniformLocation(self.program,
                                                    b'uPMatrix')
        self.mvMatrixUniform =
  glGetUniformLocation(self.program,
                                                     b'uMVMatrix
  ')
        # torus geometry
        self.R = R
        self.r = r
        # grid size
        self.NX = NX
        self.NY = NY
        # no. of points
      ❸ self.N = self.NX
        self.M = self.NY
        # time
        self.t = 0
        # compute parameters for glMultiDrawArrays
        M1 = 2*self.M + 2
      ❹ self.first_indices = [2*M1*i for i in range(self.N)]
        self.counts = [2*M1 for i in range(self.N)]
        # colors: {(i, j) : (r, g, b)}
        # with NX * NY entries
      ❺ self.colors_dict = self.init_colors(self.NX, self.NY)
```

```
        # create an empty array to hold colors
      ❻ self.colors = np.zeros((3*self.N*(2*self.M + 2), ),
np.float32)
        # get vertices, normals, indices
      ❼ vertices, normals = self.compute_vertices()
      ❽ self.compute_colors()
        # set up vertex buffer objects
      ❾ self.setup_vao(vertices, normals, self.colors)
```

The `Torus` class has the following parameters in its constructor ❶: the radius `R` of the outer ring of the torus, the radius `r` of the torus tube, and `NX` and `NY` for the number of Game of Life simulation cells in the x- and y-directions. The constructor's first order of business is to load the shaders. You use the `loadShaders()` method ❷ defined in the common *glutils.py* file. In the subsequent lines, you store the variables passed into the `Torus` constructor in instance variables such as `self.R` so they can be accessed later from other methods. Then you set `N`, the number of points on the outer circle of the torus, to be `NX`, the number of cells in the x-direction ❸. You similarly set `M`, the number of points along the smaller radius `r` of the torus, to `NY`. This scheme is discussed in the "Mapping the Grid to the Torus" section.

Next, you do some additional preparation for rendering the triangle strips that will form the bands along the outer ring of the torus. You'll eventually be using the `glMultiDrawArrays()` OpenGL method to render all the triangle strips at once. This method is an efficient way to draw multiple triangle strip primitives using just one function call. As you saw in the "Rendering" section, each triangle strip has $2M + 2$ vertices, and you have N of these strips. So, the starting indices for these triangle strips will be $[0, (2M + 2), (2M + 2) \times 2, \ldots, (2M + 2) \times N]$. Accordingly, you set `first_indices` and `counts` ❹ , which will be required parameters when you call `glMultiDrawArrays()`.

The `init_colors()` method ❺ initializes `color_dict`, which maps each grid cell to a color—black or white. We'll look at the details of the `init_colors()` method soon. You initialize the `numpy` array `colors` to zeros ❻. You'll later populate this array with the correct values. You conclude the constructor by computing the vertices and normals for the torus ❼, as well as the colors ❽, and by setting up the vertex array object (VAO) for rendering the torus ❾.

Now let's look at the `Torus` class's `init_colors()` method that was just mentioned:

```
def init_colors(self, NX, NY):
    """initialize color dictionary"""
    colors = {}
    c1 = [1.0, 1.0, 1.0]
    for i in range(NX):
        for j in range(NY):
          ❶ colors[(i, j)] = c1
    return colors
```

The `init_colors()` method creates a dictionary called `colors` mapping from simulation cell index (*i*, *j*) to the color that should be applied to the cell. To begin, you simply set all the cell color values to `c1`, which is just plain white ❶. As the Game of Life simulation unfolds, the values in this dictionary will be updated to turn cells on and off.

Calculating the Vertices

The next few methods we'll consider work together to compute all the torus vertices. We begin with the `compute_vertices()` method:

```
def compute_vertices(self):
        R, r, N, M = self.R, self.r, self.N, self.M
```

```
        # create an empty array to hold vertices/normals
        vertices = []
        normals = []
        for i in range(N):
            # for all M points around a ring
            for j in range(M+1):
                # compute angle theta of point
              ❶ theta = (j % M) *2*math.pi/M
                #---ring #1------
                # compute angle
              ❷ alpha1 = i*2*math.pi/N
                # compute transforms
              ❸ RM1, TM1 = self.compute_rt(R, alpha1)
                # compute points
              ❹ Pt1, NV1 = self.compute_pt(r, theta, RM1, TM1)
                #---ring #2------
                # index of next ring
              ❺ ip1 = (i + 1) % N
                # compute angle
              ❻ alpha2 = ip1*2*math.pi/N
                # compute transforms
                RM2, TM2 = self.compute_rt(R, alpha2)
                # compute points
                Pt2, NV2 = self.compute_pt(r, theta, RM2, TM2)
                # store vertices/normals in right order for
GL_TRIANGLE_STRIP
              ❼ vertices.append(Pt1[0:3])
                vertices.append(Pt2[0:3])
                # add normals
                normals.append(NV1[0:3])
                normals.append(NV2[0:3])
        # return vertices and colors in correct format
      ❽ vertices = np.array(vertices, np.float32).reshape(-1)
        normals = np.array(normals, np.float32).reshape(-1)
        # print(vertices.shape)
        return vertices, normals
```

The `compute_vertices()` method begins by creating empty lists to store the vertices and normals. Then you compute the vertices and normals for the torus, using a nested loop to implement the strategy we discussed in the "Rendering" section. The outer loop iterates over the `N` rings that make up the torus. The inner loop iterates over the `M` points on each of those rings. Within the loops, you first compute the angle `theta` subtended by a point on the ring with index `j` ❶. You use `j % M` and have the inner loop iterate over range `[0, M+1)` so that when *j* is equal to `M`, `(j % M)` rolls back to `0`. This is to complete the last segment of the ring.

The torus is rendered as a set of bands (triangle strips), and each band consists of two adjacent rings of points. You compute `alpha1`, the angle subtended by the first ring in a band, at index `i` ❷, and use `alpha1` to compute the rotation and translation matrices for this first ring using the `compute_rt()` method ❸. Then you pass these matrices to the `compute_pt()` method to calculate the vertex and normal for the point on the ring at angle `theta` ❹. We'll look at how the `compute_rt()` and `compute_pt()` methods work soon.

Next, you move on to the adjacent ring at index `i+1`, using `ip1 = (i+1) % N` to ensure that you roll back to zero at the end ❺. You compute the angle `alpha2` of the ring at index `ip1` ❻ and then compute the vertex and normal for the point on the `ip1` ring at angle `theta`, just as you did for the first ring.

Beginning at ❼, you append the vertices and normals for the adjacent rings to the lists you created at the beginning of the method. You pick only the first three coordinates of each vertex and normal, as in `Pt1[0:3]`, since all the matrix transformations are done with homogeneous coordinates in the form (*x*, *y*, *z*, *w*), and you need only (*x*, *y*, *z*). This action stores the vertices and normals in a Python list of triplets of the form `[[x1, y1, z1], [x2, y2, z2], ...]`. However, OpenGL expects vertex attributes to be provided in a flat array with a known size. Therefore, you convert the

`vertices` and `normals` lists into `numpy` arrays of 32-bit floats ❽, using `reshape(-1)` to ensure that they're flat arrays of the form `[x1, y1, z1, x2, y2, z2, ...]`.

Now let's look at the `compute_rt()` and `compute_pt()` methods that helped you compute the vertices and normals. We'll start with `compute_rt()`, which calculates the rotation and translation matrices needed to render a given ring in the torus:

```
def compute_rt(self, R, alpha):
        # compute position of ring
      ❶ Tx = R*math.cos(alpha)
        Ty = R*math.sin(alpha)
        Tz = 0.0

        # rotation matrix
      ❷ RM = np.array([
            [math.cos(alpha), -math.sin(alpha), 0.0, 0.0],
            [math.sin(alpha), math.cos(alpha), 0.0, 0.0],
            [0.0, 0.0, 1.0, 0.0],
            [0.0, 0.0, 0.0, 1.0]
            ], dtype=np.float32)

        # translation matrix
      ❸ TM = np.array([
            [1.0, 0.0, 0.0, Tx],
            [0.0, 1.0, 0.0, Ty],
            [0.0, 0.0, 1.0, Tz],
            [0.0, 0.0, 0.0, 1.0]
            ], dtype=np.float32)
        return (RM, TM)
```

You first calculate the translation components of the matrix ❶, using parametric equations. Then you create the rotation matrix ❷ and translation matrix ❸ as `numpy` arrays. You've seen these matrices before in the "Computing

Vertices" section. You return the arrays at the end of the method.

Here's the other helper method, `compute_pt()`, which uses the translation and rotation matrices to determine the vertex and normal vector of a given point on a ring of the torus:

```
def compute_pt(self, r, theta, RM, TM):
    # compute point coords
  ❶ P = np.array([r*math.cos(theta), 0.0, r*math.sin(theta),
  1.0],
                 dtype=np.float32)
    # print(P)
    # apply rotation - this also gives us the vertex normals
  ❷ NV = np.dot(RM, P)
    # apply translation
  ❸ Pt = np.dot(TM, NV)
    return (Pt, NV)
```

You compute the point `P` at angle `theta` on a ring lying on the XZ plane ❶. Then you apply a rotation to this point by multiplying it by the rotation matrix ❷. This also gives you the point's normal vector. You multiply the normal by the translation matrix to give you the vertex on the torus ❸.

Managing Cell Colors

Now we'll examine some methods that help set the colors of the cells on the torus. First is the `compute_colors()` method, which we originally called as part of the `Torus` class's constructor. It sets the color of each triangle in the triangle strips that make up the torus, based on the values determined by the Game of Life simulation.

```
def compute_colors(self):
```

```
    R, r, N, M = self.R, self.r, self.N, self.M

    # the points on the ring are generated on the X-Z plane
    # then they are rotated and translated into the correct
    # position on the torus

    # for all N rings around the torus
    for i in range(N):
        # for all M points around a ring
        for j in range(M+1):
            # j value
            jj = j % M
            # store colors - same color applies to (V_i_j,
V_ip1_j)
          ❶ col = self.colors_dict[(i, jj)]
            # get index into array
          ❷ index = 3*(2*i*(M+1) + 2*j)
            # set color
          ❸ self.colors[index:index+3] = col
          ❹ self.colors[index+3:index+6] = col
```

This method follows the logic described in “Coloring the Triangle Strips” on page 185 to update the values in the `colors` array, which was initialized as an array of all zeros. You retrieve the color for cell `(i, jj)` from `colors_dict`, the dictionary mapping cells to colors that you created earlier ❶. (You define `jj = j % M` so it rolls over to zero at the end.) Then you compute the index into the `colors` array at which you should update the new computed values ❷. Each pair of rings that makes up a band has `2*(M+1)` vertices, and there are `N` of these pairs. Starting at each location in the array, you store three sequential values (the RGB components of a cell’s color). Hence, the index of the *j*th color in a ring for the *i*th segment of the torus will be given by `3*(2*i*(M+1) + 2*j)`. Note that you use `j` and not `jj` in computing the index, since you’re storing the computed values here and you don’t want the index to roll over to zero. Now that you have the index, you update the `colors`

array with the new computed values. You update the array both at `[index:index+3]` ❸ and at `[index+3:index+6]` ❹ since each cell on the torus is a quad, made of two adjacent triangles.

Let's now look at `recalc_colors()`, a method for updating the color values stored on the GPU at each step in the Game of Life simulation:

```
def recalc_colors(self):
    # get colors
    self.compute_colors()
    # bind VAO
    glBindVertexArray(self.vao)
    glBindBuffer(GL_ARRAY_BUFFER, self.colorBuffer)
    # set buffer data
  ❶ glBufferSubData(GL_ARRAY_BUFFER, 0, 4*len(self.colors),
  self.colors)
    # unbind VAO
    glBindVertexArray(0)
```

For every step of the simulation, the colors of the cells are updated, which means you need to update the colors of all the triangle strips on the torus, and you need to do it efficiently so you don't slow down the rendering. The `recalc_colors()` method does this using the OpenGL `glBufferSubData()` method ❶ . The vertices, normals, and colors are stored in attribute arrays on the GPU. The vertices and normals don't change, so you computed them only once in the beginning, with a call to `compute_vertices()` in the class's constructor method. When the colors change, `glBufferSubData()` updates the color attribute arrays rather than creating them afresh.

Drawing the Torus

Finally, here's the `render()` method that draws the torus:

```
def render(self, pMatrix, mvMatrix):
    # use shader
  ❶ glUseProgram(self.program)
    # set proj matrix
  ❷ glUniformMatrix4fv(self.pMatrixUniform, 1, GL_FALSE,
  pMatrix)
    # set modelview matrix
  ❸ glUniformMatrix4fv(self.mvMatrixUniform, 1, GL_FALSE,
  mvMatrix)
    # bind VAO
  ❹ glBindVertexArray(self.vao)
    # draw
  ❺ glMultiDrawArrays(GL_TRIANGLE_STRIP, self.first_indices,
                      self.counts, self.N)
    # unbind VAO
    glBindVertexArray(0)
```

This method is similar to the render methods you saw in the previous chapter. You make a call to use the shader program ❶ and set the projection ❷ and modelview ❸ matrix uniform variables. Then you bind to the vertex array object ❹, which you created by calling `setup_vao()` in the class's constructor. The VAO has all the attribute array buffers you need. Next, you use the `glMultiDrawArrays()` method to draw `N` triangle strips ❺. You already computed `first_indices` and `counts` in the `Torus` constructor.

Implementing the Game of Life Simulation

In Chapter 3, you implemented Conway's Game of Life (GOL) by using `matplotlib` to visualize the updated values of the simulation grid. Here you'll adapt the earlier implementation to update a dictionary of cell colors instead, which will be used to update the colors of the torus. The relevant code is encapsulated in a class called `GOL`, declared in the file *gol.py*. To see the complete code listing, skip

ahead to “The Complete Game of Life Simulation Code” on page 209.

The Class Constructor

First, let’s look at the `GOL` class constructor:

```
class GOL:
  ❶ def __init__(self, NX, NY, glider):
        """GOL constructor"""
        # a grid of NX x NY random values
        self.NX, self.NY = NX, NY
        if glider:
          ❷ self.addGlider(1, 1, NX, NY)
        else:
          ❸ self.grid = np.random.choice([1, 0], NX * NY,
                                         p=[0.2,
  0.8]).reshape(NX, NY)
```

The `GOL` constructor takes as input the grid dimensions `NX` and `NY`, as well as a Boolean flag `glider` ❶. If this flag is set, you initialize the simulation grid with the “glider” pattern using the `addGlider()` method ❷. Since we already discussed this method in Chapter 3, we won’t examine it here. If the `glider` flag isn’t set, you just initialize the grid to random ones and zeros ❸.

The `GOL` class uses an `update()` method to update the simulation grid at each time step. Again, this is identical to the previous implementation.

The get_colors() Method

The `get_colors()` method is what distinguishes this chapter’s Game of Life implementation from that of Chapter 3. The method builds up a dictionary mapping of each Game of Life

cell to its color value at a given step in the simulation: black for ON or white for OFF. This dictionary is passed to the `Torus` object when the scene is updated.

```
def get_colors(self):
    colors = {}
  ❶ c1 = np.array([1.0, 1.0, 1.0], np.float32)
  ❷ c2 = np.array([0.0, 0.0, 0.0], np.float32)
    for i in range(self.NX):
        for j in range (self.NY):
            if self.grid[i, j] == 1:
                colors[(i, j)] = c2
            else :
                colors[(i, j)] = c1
    return colors
```

Here you iterate through all the cells in the simulation grid and set the RGB colors based on whether the grid value is `0` or `1`. The possible colors are defined as `c1` for white ❶ or `c2` for black ❷. These colors will be used while rendering the torus.

Creating the Camera

In “Controlling the Camera” on page 185, we discussed how to build a camera that orbits around the torus. Now let’s look at the implementation. The code is encapsulated in the class `OrbitCamera`, which is declared in the file *camera.py*. To see the complete code listing, skip ahead to “The Complete Camera Code” on page 211.

Constructing the Class

Here’s the constructor for the `OrbitCamera` class:

```
class OrbitCamera:
    """helper class for viewing"""
    def __init__(self, height, radius, beta_step=1):
      ❶ self.radius = radius
      ❷ self.beta = 0
      ❸ self.beta_step = beta_step
      ❹ self.height = height
        # initial eye vector is (-R, 0, -H)
        rr = radius/math.sqrt(2.0)
      ❺ self.eye = np.array([rr, rr, height], np.float32)
        # compute up vector
      ❻ self.up = self.__compute_up_vector(self.eye )
        # center is origin
      ❼ self.center = np.array([0, 0, 0], np.float32)
```

You start by setting the camera parameters passed into the `OrbitCamera` constructor. These include the camera's orbit radius ❶ and `beta`, the angle that the view vector (projected on the XY plane) makes with the x-axis ❷. You also set the amount `beta` should increment with each time step of the camera rotation ❸ and the height of the camera from the XY plane ❹.

Next, you set the initial value of the eye position to be midway between the positive x- and positive y-axis, at a distance `R` from the origin, suspended at the specified `height` ❺. You can calculate this as:

$$E = \left(\frac{R}{\sqrt{2}}, \frac{R}{\sqrt{2}}, height \right)$$

Finally, you compute the camera's up vector ❻ and set the center as the origin (0, 0, 0) ❼. Remember that these

are the pieces of information, along with the eye position, that OpenGL needs to model a camera.

Calculating the Up Vector

Here's the method that you call in the `OrbitCamera` class constructor to compute the up vector:

```
def __compute_up_vector(self, E):
    # N = (E x k) x E
    Z = np.array([0, 0, 1], np.float32)
  ❶ U = np.cross(np.cross(E, Z), E)
    # normalize
  ❷ U = U / np.linalg.norm(U)
    return U
```

The `__compute_up_vector()` method calculates the up vector `U` based on the method we discussed earlier in "Controlling the Camera" on page 185. Specifically, you use cross products and the initial up vector guess of (0, 0, 1) to compute the correct up vector ❶. Then you normalize the up vector ❷ before returning it.

Rotating the Camera

The `OrbitCamera` class's `rotate()` method is called every time you need to rotate the camera around the torus by one step. Here's the method's definition:

```
def rotate(self):
    """rotate by one step and compute new camera parameters"""
  ❶ self.beta = (self.beta + self.beta_step) % 360
    # recalculate eye E
  ❷ self.eye =
  np.array([self.radius*math.cos(math.radians(self.beta)),
```

```
                        self.radius*math.sin(math.radians(self
.beta)),
                        self.height], np.float32)
    # up vector
❸ self.up = self.__compute_up_vector(self.eye)
```

You increase the angle `beta` by the increment `beta_step`, using the `%` operator to ensure that the angle rolls over to 0 when it reaches 360 degrees ❶. Then you use the new `beta` value to compute the updated eye position ❷, and you use the new eye position to compute the new up vector with the `__compute_up_vector()` method ❸.

Putting Everything Together

You've built all the classes necessary for rendering the torus. Now you need some code to bring those classes together, create and manage the OpenGL window, and coordinate the rendered objects. Create the class `RenderWindow` (defined in *gol_torus.py*) for this purpose. It's similar to the `RenderWindow` class used in Chapter 9, so we'll discuss only the parts of the code that are unique to the current project. To see the complete code listing, skip ahead to "The Complete RenderWindow Code" on page 211.

The main() Function

Before we examine the `RenderWindow` class, let's look the program's `main()` function, which sets the whole simulation in motion. This function is also defined in *gol_torus.py*.

```
def main():
    print("Starting GOL. Press ESC to quit.")
    # parse arguments
```

```
    parser = argparse.ArgumentParser(description="Runs Conway's
  Game of Life
                                         simulation on a Torus.")
    # add arguments
  ❶ parser.add_argument('--glider', action='store_true',
  required=False)
    args = parser.parse_args()
    glider = False
    if args.glider:
      ❷ glider = True
  ❸ rw = RenderWindow(glider)
  ❹ rw.run()
```

You add a command line argument called `--glider` so you can bring up the torus with just a glider pattern on it ❶ and set the corresponding flag ❷. Then you create a `RenderWindow` object ❸, which initializes all the other objects needed for the program, and start the rendering with a call to the `RenderWindow` object's `run()` method ❹.

The RenderWindow Constructor

The constructor on the `RenderWindow` class starts with the standard GLFW OpenGL setup you saw in Chapter 9, including setting the window size, calling the render methods, and handling window and keyboard events. Then the constructor goes on with the following Game of Life–specific initializations:

```
class RenderWindow:
    def __init__(self, glider):
    --snip--
       # create 3D
        NX = 64
        NY = 64
        R = 4.0
        r = 1.0
```

```
    ❶ self.torus = Torus(R, r, NX, NY)
    ❷ self.gol = GOL(NX, NY, glider)
      # create a camera
    ❸ self.camera = OrbitCamera(5.0, 10.0)
      # exit flag
    ❹ self.exitNow = False
    # rotation flag
    ❺ self.rotate = True
      # skip count
    ❻ self.skip = 0
```

First you set some parameters for the simulation, including the number of cells in the grid and the inner and outer radiuses of the torus. Then you create the `Torus` object using these parameters ❶, as well as the `GOL` object that will manage the simulation ❷ . You also create the orbiting camera at a radius of 5 units from the origin and a height of 10 units from the XY plane ❸.

Next, you set the exit flag used to quit the program ❹, and you initialize the rotation flag to `True` ❺. Finally, you set a `skip` variable ❻, which you'll use to control the frequency at which the simulation updates. You'll see how the `skip` variable works later in this section.

The run() and step() Methods

The `run()` method of the `RenderWindow` object is responsible for running the simulation, with help from the `step()` method. Let's take a look at the `run()` method first:

```
def run(self):
    # initializer timer
    glfw.glfwSetTime(0)
    t = 0.0
  ❶ while not glfw.glfwWindowShouldClose(self.win) and not
  self.exitNow:
```

```
        # update every x seconds
        currT = glfw.glfwGetTime()
      ❷ if currT - t > 0.05:
            # update time
            t = currT
            # set viewport
          ❸ self.width, self.height =
glfw.glfwGetFramebufferSize(self.win)
            self.aspect = self.width/float(self.height)
            glViewport(0, 0, self.width, self.height)

            # clear
            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
            # build projection matrix
            pMatrix = glutils.perspective(60.0, self.aspect,
0.1, 100.0)

            mvMatrix = glutils.lookAt(self.camera.eye,
self.camera.center,
                                      self.camera.up)
            # render
          ❹ self.torus.render(pMatrix, mvMatrix)

            # step
          ❺ if self.rotate:
                self.step()
            glfw.glfwSwapBuffers(self.win)
            # poll for and process events
            glfw.glfwPollEvents()
    # end
    glfw.glfwTerminate()
```

The rendering scheme is designed to keep rendering frames in a loop until the window is closed or the ESC key is pressed ❶. Before you proceed, you check whether the time elapsed since the last render is greater than 0.05 seconds ❷ . This helps maintain a maximum frame rate. Starting at ❸, you perform some standard OpenGL operations, such as setting the viewport, clearing the screen, and computing the current transformation that needs to be set into the

vertex shader. Then you render the torus ❹ and call the `step()` method ❺, which will rotate the camera and update the Game of Life simulation by one time step. Once the rendering is done, you swap the OpenGL buffers and poll for further window events. If you exit the loop, you call the `glfwTerminate()` method for cleanup.

Here's the `step()` method that increments the camera and the simulation:

```
def step(self):
  ❶ if self.skip == 9:
        # update GOL
      ❷ self.gol.update()
      ❸ colors = self.gol.get_colors()
      ❹ self.torus.set_colors(colors)
        # step
      ❺ self.torus.step()
        # reset
      ❻ self.skip = 0
    # update skip
  ❼ self.skip += 1
    # rotate camera
  ❽ self.camera.rotate()
```

Every time this method is called, it rotates the camera by one step ❽. You also want to update the Game of Life simulation, but doing so at the same rate that the camera moves would not be visually pleasing. You therefore use the `skip` variable to slow down the simulation by a factor of 9 relative to the camera motion. This variable starts from `0` and is incremented each time the `step()` method is called ❼. When `skip` gets to `9` ❶, you update the simulation by one time step. To do this, you first call the `GOL` class's `update()` method ❷, which turns cells on or off according to the Conway's Game of Life rules. Then you get the updated cell colors from the simulation ❸, set them to the torus ❹, and

call `torus.step()` ❺, which will update the attribute buffers with the new colors. Finally, you reset the `skip` variable to `0` so the process can repeat ❻.

Running the 3D Game of Life Simulation

Now we're ready to run the code. Enter the following at the terminal:

```
$ python gol_torus.py
```

Figure 10-8 shows the output.



Figure 10-8: *A rendering of the Game of Life on a torus*

The program will open a window that shows the torus you meticulously constructed, with the Game of Life simulation running on its surface! As the simulation evolves, try to find some of the familiar Game of Life patterns you saw in Chapter 3. Notice that the light direction remains constant while the camera orbits the torus. As the camera turns, you'll be able to see the light and dark portions of the torus.

Now let's try the glider option:

```
$ python gol_torus.py --glider
```

Figure 10-9 shows the output.

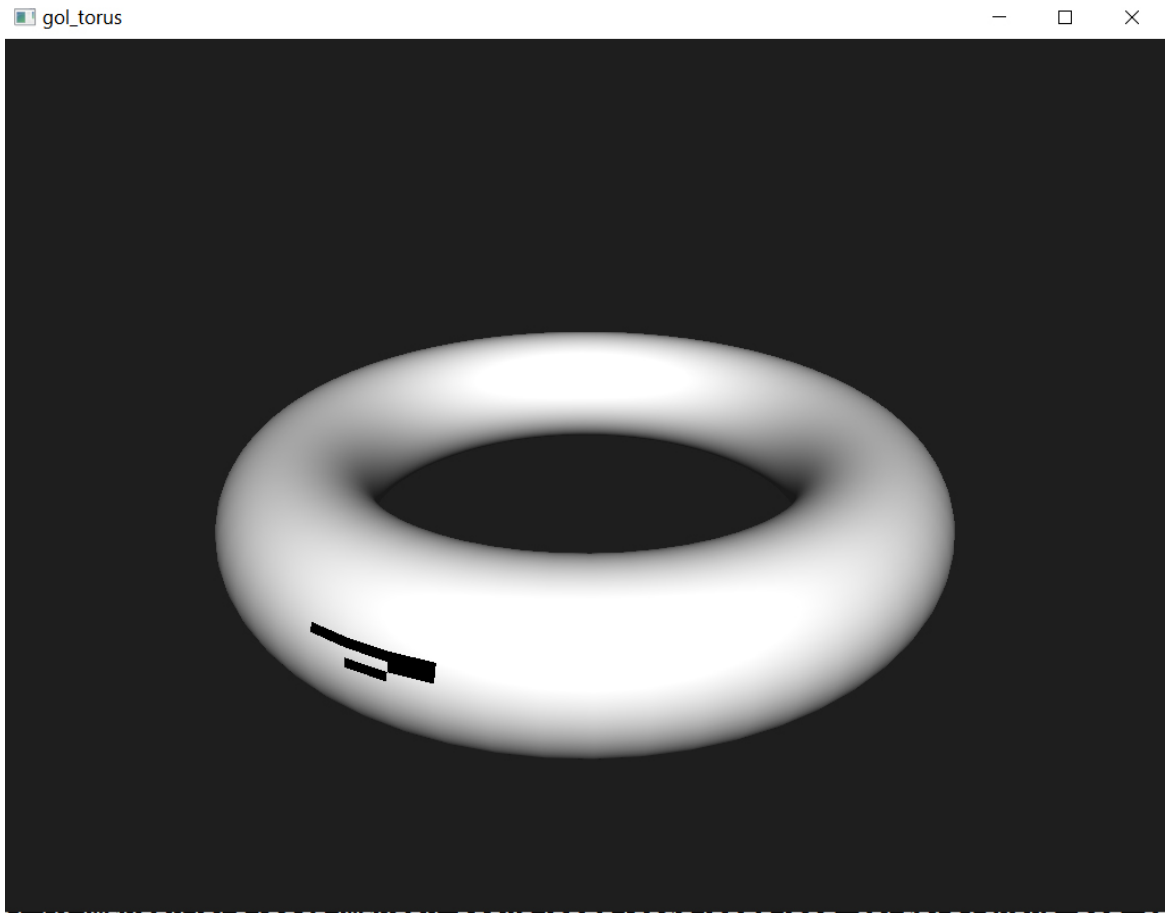


Figure 10-9: *A Game of Life glider on a torus*

Sit back and enjoy watching the lone glider make its way along the surface of the torus!

Summary

In this chapter, you implemented Conway's Game of Life on a torus. You learned how to compute the vertices of a torus and how to render it using OpenGL, and you saw how code can be adapted from one context (a flat rendering of the Game of Life simulation) to another (a 3D rendering of the same simulation). In the process, I hope you've gotten a more intuitive feeling for how the toroidal boundary conditions we discussed in Chapter 3 work.

Experiments!

Here are a few experiments you can try with this project:

1. In this chapter's implementation, the torus is lit by a single light source. Try adding another light source in the shader code. Now the computed color of a vertex will be the sum of contributions from both light sources. Try changing the positions and colors of the light sources and see the effect on the torus lighting.
2. To get a representative view of the simulation, you defined a camera that revolves around the z-axis of the torus, in a plane parallel to the XY plane. Now create a camera that flies over the torus instead. Your camera will start by looking down at the torus along the negative z-axis and will move in a circle along the XZ plane, at a fixed distance from the center of the torus. Think about how you will compute the eye point, view direction, and up vector for each step of the movement.

The Complete Torus Rendering Code

Here's the complete listing for the file *torus.py*:

```
"""
torus.py

A Python OpenGL program that generates a torus.

Author: Mahesh Venkitachalam
"""

import OpenGL
from OpenGL.GL import *
```

```
import numpy as np
import math, sys, os
import glutils

import glfw

strVS = """
# version 330 core

layout(location = 0) in vec3 aVert;
layout(location = 1) in vec3 aColor;
layout(location = 2) in vec3 aNormal;

uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;

flat out vec3 vColor;
out vec3 vNormal;
out vec3 fragPos;

void main() {
  // transform vertex
  gl_Position = uPMatrix * uMVMatrix * vec4(aVert, 1.0);
  fragPos = aVert;
  vColor = aColor;
  vNormal = aNormal;
}
"""

strFS = """
# version 330 core
flat in vec3 vColor;

in vec3 vNormal;
in vec3 fragPos;

out vec4 fragColor;

void main() {
  vec3 lightPos = vec3(10.0, 10.0, 10.0);
  vec3 lightColor = vec3(1.0, 1.0, 1.0);
  vec3 lightDir = normalize(lightPos - fragPos);
  float diff = max(dot(vNormal, lightDir), 0.0);
  vec3 diffuse = diff * lightColor;
  float ambient = 0.1;
```

```
  vec3 result = (ambient + diffuse) * vColor.xyz;
  fragColor = vec4(result, 1.0);
}
"""

class Torus:
    """ OpenGL 3D scene class"""
    # initialization
    def __init__(self, R, r, NX, NY):
        global strVS, strFS

        # create shader
        self.program = glutils.loadShaders(strVS, strFS)

        glProvokingVertex(GL_FIRST_VERTEX_CONVENTION)

        self.pMatrixUniform =
  glGetUniformLocation(self.program,
                                                    b'uPMatrix')
        self.mvMatrixUniform =
  glGetUniformLocation(self.program,
                                                     b'uMVMatrix
  ')

        # torus geometry
        self.R = R
        self.r = r
        # grid size
        self.NX = NX
        self.NY = NY
        # no. of points
        self.N = self.NX
        self.M = self.NY

        # time
        self.t = 0

        # compute parameters for glMultiDrawArrays
        M1 = 2*self.M + 2
        self.first_indices = [2*M1*i for i in range(self.N)]
        self.counts = [2*M1 for i in range(self.N)]

        # colors: {(i, j) : (r, g, b)}
        # with NX * NY entries
```

```
        self.colors_dict = self.init_colors(self.NX, self.NY)

        # create an empty array to hold colors
        self.colors = np.zeros((3*self.N*(2*self.M + 2), ),
np.float32)

        # get vertices, normals, indices
        vertices, normals = self.compute_vertices()
        self.compute_colors()
        # set up vertex buffer objects
        self.setup_vao(vertices, normals, self.colors)

    def init_colors(self, NX, NY):
        """initialize color dictionary"""
        colors = {}
        c1 = [1.0, 1.0, 1.0]
        for i in range(NX):
            for j in range (NY):
                colors[(i, j)] = c1
        return colors

    def compute_rt(self, R, alpha):
        # compute position of ring
        Tx = R*math.cos(alpha)
        Ty = R*math.sin(alpha)
        Tz = 0.0

        # rotation matrix
        RM = np.array([
            [math.cos(alpha), -math.sin(alpha), 0.0, 0.0],
            [math.sin(alpha), math.cos(alpha), 0.0, 0.0],
            [0.0, 0.0, 1.0, 0.0],
            [0.0, 0.0, 0.0, 1.0]
            ], dtype=np.float32)

        # translation matrix
        TM = np.array([
            [1.0, 0.0, 0.0, Tx],
            [0.0, 1.0, 0.0, Ty],
            [0.0, 0.0, 1.0, Tz],
            [0.0, 0.0, 0.0, 1.0]
            ], dtype=np.float32)
```

```
        return (RM, TM)

    def compute_pt(self, r, theta, RM, TM):
        # compute point coords
        P = np.array([r*math.cos(theta), 0.0,
r*math.sin(theta), 1.0],
                     dtype=np.float32)
        # print(P)
        # apply rotation - this also gives us the vertex
normals
        NV = np.dot(RM, P)
        # normalize
        # NV = NV / np.linalg.norm(NV)
        # apply translation
        Pt = np.dot(TM, NV)

        return (Pt, NV)

    def compute_vertices(self):
        """compute vertices for the torus
           returns np float32 array of n coords (x, y, z):
shape (3*n, )
        """

        R, r, N, M = self.R, self.r, self.N, self.M

        # create an empty array to hold vertices/normals
        vertices = []
        normals = []

        # the points on the ring are generated on the X-Z plane
        # then they are rotated and translated into the correct
        # position on the torus

        # for all N rings around the torus
        for i in range(N):

            # for all M points around a ring
            for j in range(M+1):

                # compute angle theta of point
                theta = (j % M) *2*math.pi/M

                #---ring #1------

                # compute angle
```

```
                alpha1 = i*2*math.pi/N
                # compute transforms
                RM1, TM1 = self.compute_rt(R, alpha1)
                # compute points
                Pt1, NV1 = self.compute_pt(r, theta, RM1, TM1)

                #---ring #2------
                # index of next ring
                ip1 = (i + 1) % N

                # compute angle
                alpha2 = ip1*2*math.pi/N
                # compute transforms
                RM2, TM2 = self.compute_rt(R, alpha2)
                # compute points
                Pt2, NV2 = self.compute_pt(r, theta, RM2, TM2)

                # store vertices/normals in right order for
GL_TRIANGLE_STRIP
                vertices.append(Pt1[0:3])
                vertices.append(Pt2[0:3])

                # add normals
                normals.append(NV1[0:3])
                normals.append(NV2[0:3])

        # return vertices and colors in correct format
        vertices = np.array(vertices, np.float32).reshape(-1)
        normals = np.array(normals, np.float32).reshape(-1)
        # print(vertices.shape)
        return vertices, normals

    def compute_colors(self):
        """compute vertices for the torus
           returns np float32 array of n coords (x, y, z):
shape (3*n, )
        """

        R, r, N, M = self.R, self.r, self.N, self.M

        # the points on the ring are generated on the X-Z plane
        # then they are rotated and translated into the correct
        # position on the torus
```

```
      # for all N rings around the torus
      for i in range(N):

          # for all M points around a ring
          for j in range(M+1):

              # j value
              jj = j % M

              # store colors - same color applies to (V_i_j,
V_ip1_j)
              col = self.colors_dict[(i, jj)]
              # get index into array
              index = 3*(2*i*(M+1) + 2*j)
              # set color
              self.colors[index:index+3] = col
              self.colors[index+3:index+6] = col

  def setup_vao(self, vertices, normals, colors):
      # set up vertex array object (VAO)
      self.vao = glGenVertexArrays(1)
      glBindVertexArray(self.vao)

      # --------
      # vertices
      # --------
      self.vertexBuffer = glGenBuffers(1)
      glBindBuffer(GL_ARRAY_BUFFER, self.vertexBuffer)
      # set buffer data
      glBufferData(GL_ARRAY_BUFFER, 4*len(vertices),
vertices,
                   GL_STATIC_DRAW)
      # enable vertex attribute array
      glEnableVertexAttribArray(0)
      # set buffer data pointer
      glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
None)

      # normals
      # --------
      self.normalBuffer = glGenBuffers(1)
      glBindBuffer(GL_ARRAY_BUFFER, self.normalBuffer)
      # set buffer data
```

```
        glBufferData(GL_ARRAY_BUFFER, 4*len(normals), normals,
                     GL_STATIC_DRAW)
        # enable vertex attribute array
        glEnableVertexAttribArray(2)
        # set buffer data pointer
        glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, 0,
None)

        # --------
        # colors
        # --------
        self.colorBuffer = glGenBuffers(1)
        glBindBuffer(GL_ARRAY_BUFFER, self.colorBuffer)
        # set buffer data
        glBufferData(GL_ARRAY_BUFFER, 4*len(colors), colors,
                     GL_STATIC_DRAW)
        # enable color attribute array
        glEnableVertexAttribArray(1)
        # set buffer data pointer
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0,
None)

        # unbind VAO
        glBindVertexArray(0)

    def set_colors(self, colors):
        self.colors_dict = colors
        self.recalc_colors()

    def recalc_colors(self):

        # get colors
        self.compute_colors()

        # bind VAO
        glBindVertexArray(self.vao)
        # --------
        # colors
        # --------
        glBindBuffer(GL_ARRAY_BUFFER, self.colorBuffer)
        # set buffer data
        glBufferSubData(GL_ARRAY_BUFFER, 0, 4*len(self.colors),
self.colors)
```

```
        # unbind VAO
        glBindVertexArray(0)

    # step
    def step(self):
        # recompute colors
        self.recalc_colors()

    # render
    def render(self, pMatrix, mvMatrix):
        # use shader
        glUseProgram(self.program)

        # set proj matrix
        glUniformMatrix4fv(self.pMatrixUniform, 1, GL_FALSE,
pMatrix)

        # set modelview matrix
        glUniformMatrix4fv(self.mvMatrixUniform, 1, GL_FALSE,
mvMatrix)

        # bind VAO
        glBindVertexArray(self.vao)
        # draw
        glMultiDrawArrays(GL_TRIANGLE_STRIP,
self.first_indices,
                          self.counts, self.N)
        # unbind VAO
        glBindVertexArray(0)
```

The Complete Game of Life Simulation Code

Here's the full code listing for the file *gol.py*.

```
"""
gol.py

Implements Conway's Game of Life.
```

```
Author: Mahesh Venkitachalam
"""

import numpy as np

class GOL:
    """GOL - class that implements Conway's Game of Life
    """
    def __init__(self, NX, NY, glider):
        """GOL constructor"""
        # a grid of NX x NY random values
        self.NX, self.NY = NX, NY
        if glider:
            self.addGlider(1, 1, NX, NY)
        else:
            self.grid = np.random.choice([1, 0], NX * NY, p=
     [0.2, 0.8]).reshape(NX, NY)

    def addGlider(self, i, j, NX, NY):
        """adds a glider with top left cell at (i, j)"""
        self.grid = np.zeros(NX * NY).reshape(NX, NY)
        glider = np.array([[0,    0, 1],
                           [1,  0, 1],
                           [0,  1, 1]])
        self.grid[i:i+3, j:j+3] = glider

    def update(self):
        """update the GOL simulation by one time step"""
        # copy grid since we require 8 neighbors for
     calculation
        # and we go line by line
        newGrid = self.grid.copy()
        NX, NY = self.NX, self.NY
        for i in range(NX):
            for j in range(NY):
                # compute 8-neighbor sum
                # using toroidal boundary conditions - x and y
     wrap around
                # so that the simulation takes place on a
     toroidal surface
                total = (self.grid[i, (j-1) % NY] +
     self.grid[i, (j+1) % NY] +
```

```
                    self.grid[(i-1) % NX, j] +
    self.grid[(i+1) % NX, j] +
                    self.grid[(i-1) % NX, (j-1) % NY] +
    self.grid[(i-1) % NX, (j+1) % NY] +
                    self.grid[(i+1) % NX, (j-1) % NY] +
    self.grid[(i+1) % NX, (j+1) % NY])
            # apply Conway's rules
            if self.grid[i, j]  == 1:
                if (total < 2) or (total > 3):
                    newGrid[i, j] = 0
            else:
                if total == 3:
                    newGrid[i, j] = 1
        # update data
        self.grid[:] = newGrid[:]

    def get_colors(self):
        """returns a dictionary of colors"""
        colors = {}
        c1 = np.array([1.0, 1.0, 1.0], np.float32)
        c2 = np.array([0.0, 0.0, 0.0], np.float32)
        for i in range(self.NX):
            for j in range (self.NY):
                if self.grid[i, j] == 1:
                    colors[(i, j)] = c2
                else :
                    colors[(i, j)] = c1
        return colors
```

The Complete Camera Code

Here's the complete code in the file *camera.py*:

```
"""
camera.py

A simple camera class for OpenGL rendering.
```

```
Author: Mahesh Venkitachalam
"""

import numpy as np
import math

class OrbitCamera:
    """helper class for viewing"""
    def __init__(self, height, radius, beta_step=1):
        self.radius = radius
        self.beta = 0
        self.beta_step = beta_step
        self.height = height
        # initial eye vector is (-R, 0, -H)
        rr = radius/math.sqrt(2.0)
        self.eye = np.array([rr, rr, height], np.float32)
        # compute up vector
        self.up = self.__compute_up_vector(self.eye )
        # center is origin
        self.center = np.array([0, 0, 0], np.float32)

    def __compute_up_vector(self, E):
        """compute up vector
        N = (E x k) x E
        """
        # N = (E x k) x E
        Z = np.array([0, 0, 1], np.float32)
        U = np.cross(np.cross(E, Z), E)
        # normalize
        U = U / np.linalg.norm(U)
        return U

    def rotate(self):
        """rotate by one step and compute new camera
  parameters"""
        self.beta = (self.beta + self.beta_step) % 360
        # recalculate eye E
        self.eye =
  np.array([self.radius*math.cos(math.radians(self.beta)),
                              self.radius*math.sin(math.radians(
  self.beta)),
```

```
                            self.height], np.float32)
        # up vector
        self.up = self.__compute_up_vector(self.eye)
```

The Complete RenderWindow Code

The complete code listing for *gol_torus.py*, including the `RenderWindow` class and the `main()` function, follows.

```
"""
gol_torus.py

Python OpenGL program that displays a torus.

Author: Mahesh Venkitachalam
"""

import OpenGL
from OpenGL.GL import *

import numpy, math, sys, os
import argparse
import glutils

import glfw

from torus import Torus
from camera import OrbitCamera
from gol import GOL

class RenderWindow:
    """GLFW Rendering window class"""
    def __init__(self, glider):

        # save current working directory
        cwd = os.getcwd()

        # initialize glfw - this changes cwd
        glfw.glfwInit()

        # restore cwd
        os.chdir(cwd)
```

```
    # version hints
    glfw.glfwWindowHint(glfw.GLFW_CONTEXT_VERSION_MAJOR, 3)
    glfw.glfwWindowHint(glfw.GLFW_CONTEXT_VERSION_MINOR, 3)
    glfw.glfwWindowHint(glfw.GLFW_OPENGL_FORWARD_COMPAT,
GL_TRUE)
    glfw.glfwWindowHint(glfw.GLFW_OPENGL_PROFILE,
glfw.GLFW_OPENGL_CORE_PROFILE)

    # make a window
    self.width, self.height = 640, 480
    self.aspect = self.width/float(self.height)
    self.win = glfw.glfwCreateWindow(self.width,
self.height, b'gol_torus')
    # make context current
    glfw.glfwMakeContextCurrent(self.win)

    # initialize GL
    glViewport(0, 0, self.width, self.height)
    glEnable(GL_DEPTH_TEST)
    #glClearColor(0.2, 0.2, 0.2, 1.0)
    glClearColor(0.11764706, 0.11764706, 0.11764706, 1.0)

    # set window callbacks
    glfw.glfwSetMouseButtonCallback(self.win,
self.onMouseButton)
    glfw.glfwSetKeyCallback(self.win, self.onKeyboard)

    # create 3D
    NX = 64
    NY = 64
    R = 4.0
    r = 1.0
    self.torus = Torus(R, r, NX, NY)
    self.gol = GOL(NX, NY, glider)

    # create a camera
    self.camera = OrbitCamera(5.0, 10.0)

    # exit flag
    self.exitNow = False

    # rotation flag
    self.rotate = True
```

```
    # skip count
    self.skip = 0

def onMouseButton(self, win, button, action, mods):
    # print 'mouse button: ', win, button, action, mods
    pass

def onKeyboard(self, win, key, scancode, action, mods):
    # print 'keyboard: ', win, key, scancode, action, mods
    if action == glfw.GLFW_PRESS:
        # ESC to quit
        if key == glfw.GLFW_KEY_ESCAPE:
            self.exitNow = True
        elif key == glfw.GLFW_KEY_R:
            self.rotate = not self.rotate

def run(self):
    # initializer timer
    glfw.glfwSetTime(0)
    t = 0.0
    while not glfw.glfwWindowShouldClose(self.win) and not
 self.exitNow:
        # update every x seconds
        currT = glfw.glfwGetTime()
        if currT - t > 0.05:
            # update time
            t = currT

            # set viewport
            self.width, self.height =
 glfw.glfwGetFramebufferSize(self.win)
            self.aspect = self.width/float(self.height)
            glViewport(0, 0, self.width, self.height)

            # clear
            glClear(GL_COLOR_BUFFER_BIT |
 GL_DEPTH_BUFFER_BIT)

            # build projection matrix
            pMatrix = glutils.perspective(60.0,
 self.aspect, 0.1, 100.0)

            mvMatrix = glutils.lookAt(self.camera.eye,
 self.camera.center, self.camera.up)
```

```
                # render
                self.torus.render(pMatrix, mvMatrix)

                # step
                if self.rotate:
                    self.step()

                glfw.glfwSwapBuffers(self.win)
                # poll for and process events
                glfw.glfwPollEvents()
        # end
        glfw.glfwTerminate()

    def step(self):

        if self.skip == 9:
            # update GOL
            self.gol.update()
            colors = self.gol.get_colors()
            self.torus.set_colors(colors)
            # step
            self.torus.step()
            # reset
            self.skip = 0

        # update skip
        self.skip += 1
        # rotate camera
        self.camera.rotate()

# main() function
def main():
    print("Starting GOL. Press ESC to quit.")
    # parse arguments
    parser = argparse.ArgumentParser(description="Runs Conway's
     Game of Life simulation
                                                  on a Torus.")
    # add arguments
    parser.add_argument('--glider', action='store_true',
     required=False)
    args = parser.parse_args()

    # set args
```

```
    glider = False
    if args.glider:
        glider = True

    rw = RenderWindow(glider)
    rw.run()

# call main
if __name__ == '__main__':
    main()
```

11
VOLUME RENDERING



Magnetic resonance imaging (MRI) and computed tomography (CT) scans are diagnostic processes that create *volumetric data*, data that consists of a set of 2D images showing cross sections through a 3D volume. *Volume rendering* is a computer graphics technique used to construct 3D images from this type of volumetric data. Although volume rendering is commonly used to analyze medical scans, it can also be used to create 3D scientific visualizations in academic disciplines such as geology, archaeology, and molecular biology.

The data captured by MRI and CT scans typically follows the form of a 3D grid of dimensions $N_x \times N_y \times N_z$. In other words, there are N_z 2D "slices," where each slice is an image of size $N_x \times N_y$. Volume rendering algorithms are used to display the collected slice data with some type of transparency, and various techniques are used to accentuate the parts of the rendered volume that are of interest.

In this project, you'll look at a volume rendering called *volume ray casting*, which takes full advantage of the graphics processing unit (GPU) to perform computations using OpenGL Shading Language (GLSL) shaders. Your code

executes for every pixel onscreen and leverages the GPU, which is designed to do parallel computations efficiently. You'll use a folder of 2D images consisting of slices from a 3D data set to construct a volume-rendered image using the volume ray casting algorithm. You'll also implement a method to show 2D slices of the data in the x-, y-, and z-directions so users can scroll through the slices using the arrow keys. Keyboard commands will let the user toggle between the 3D rendering and the 2D slices.

Here are some of the topics covered in this project:

- Using GLSL for GPU computations
- Creating vertex and fragment shaders
- Representing 3D volumetric data and using the volume ray casting algorithm
- Using `numpy` arrays for 3D transformation matrices

How It Works

There are various ways to render a 3D data set. In this project, you'll use the volume ray casting method, which is an *image-based* rendering technique used to generate the final image from the 2D slices, pixel by pixel. In contrast, typical 3D rendering methods are *object based*: they begin with a 3D object representation and then apply transformations to generate the pixels in the projected 2D image.

In the volume ray casting method that you'll use in this project, for each pixel in the output image, a ray is shot into the discrete 3D volumetric data set, which is typically represented as a cuboid. As the ray passes through the volume, the data is sampled at regular intervals, and the samples are combined, or *composited*, to compute the color

value or intensity of the final image. (You might think of this process as similar to stacking a bunch of transparencies on top of each other and holding them up against a bright light to see a blend of all the sheets.)

While volume ray casting rendering implementations typically use techniques such as applying gradients to improve the appearance of the final render, filtering to isolate 3D features, and using spatial optimization techniques to improve speed, you'll just implement the basic ray casting algorithm and composite the final image by X-ray casting. (My implementation is largely based on the seminal paper on this topic by Kruger and Westermann, published in 2003.[1])

The Data Format

For this project, you'll use medical data from 3D scans from the Stanford Volume Data Archive.[2] This archive offers a few excellent 3D medical data sets (both CT and MRI) of TIFF images, one for each 2D cross section of the volume. You'll read a folder of these images into an OpenGL 3D texture; this is sort of like stacking a set of 2D images to form a cuboid, as shown in Figure 11-1.

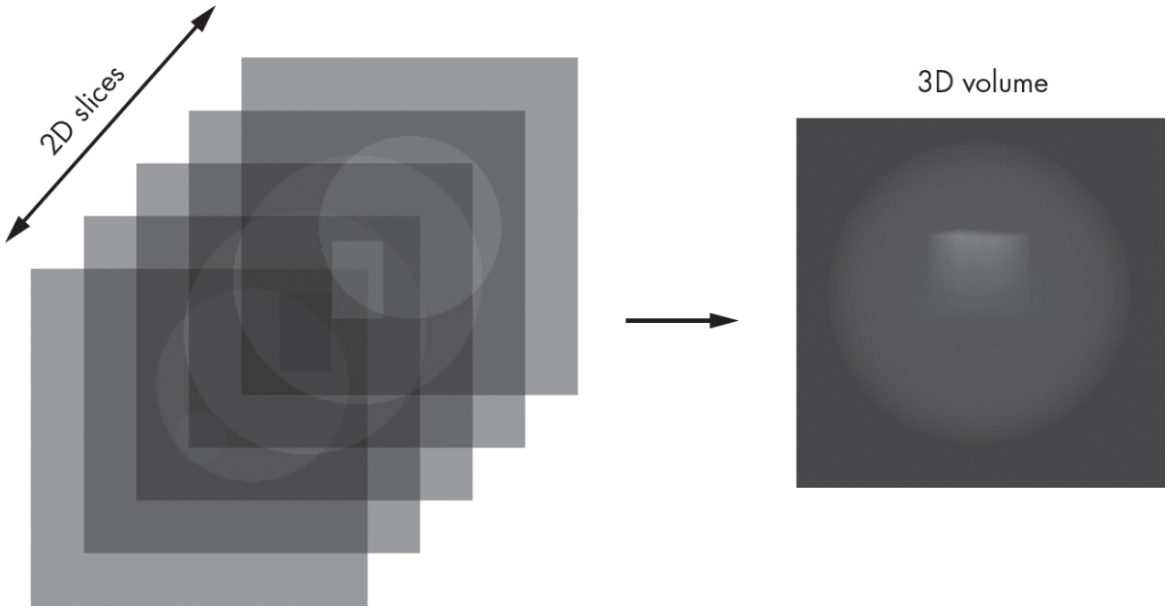


Figure 11-1: *Building 3D volumetric data from 2D slices*

Recall from Chapter 9 that a 2D texture in OpenGL is addressed with a 2D coordinate (*s*, *t*). Similarly, a 3D texture is addressed using a 3D texture coordinate of the form (*s*, *t*, *p*). As you'll see, storing the volumetric data as a 3D texture allows you to access the data quickly and provides you with interpolated values required by your ray casting scheme.

Ray Generation

Your goal in this project is to generate a perspective projection of the 3D volumetric data, as shown in Figure 11-2. The figure shows the OpenGL view frustum, as discussed in Chapter 9. Specifically, it shows how a ray from the eye enters this frustum at the near plane, passes through the cubic volume (which contains the volumetric data), and exits from the rear at the far plane.

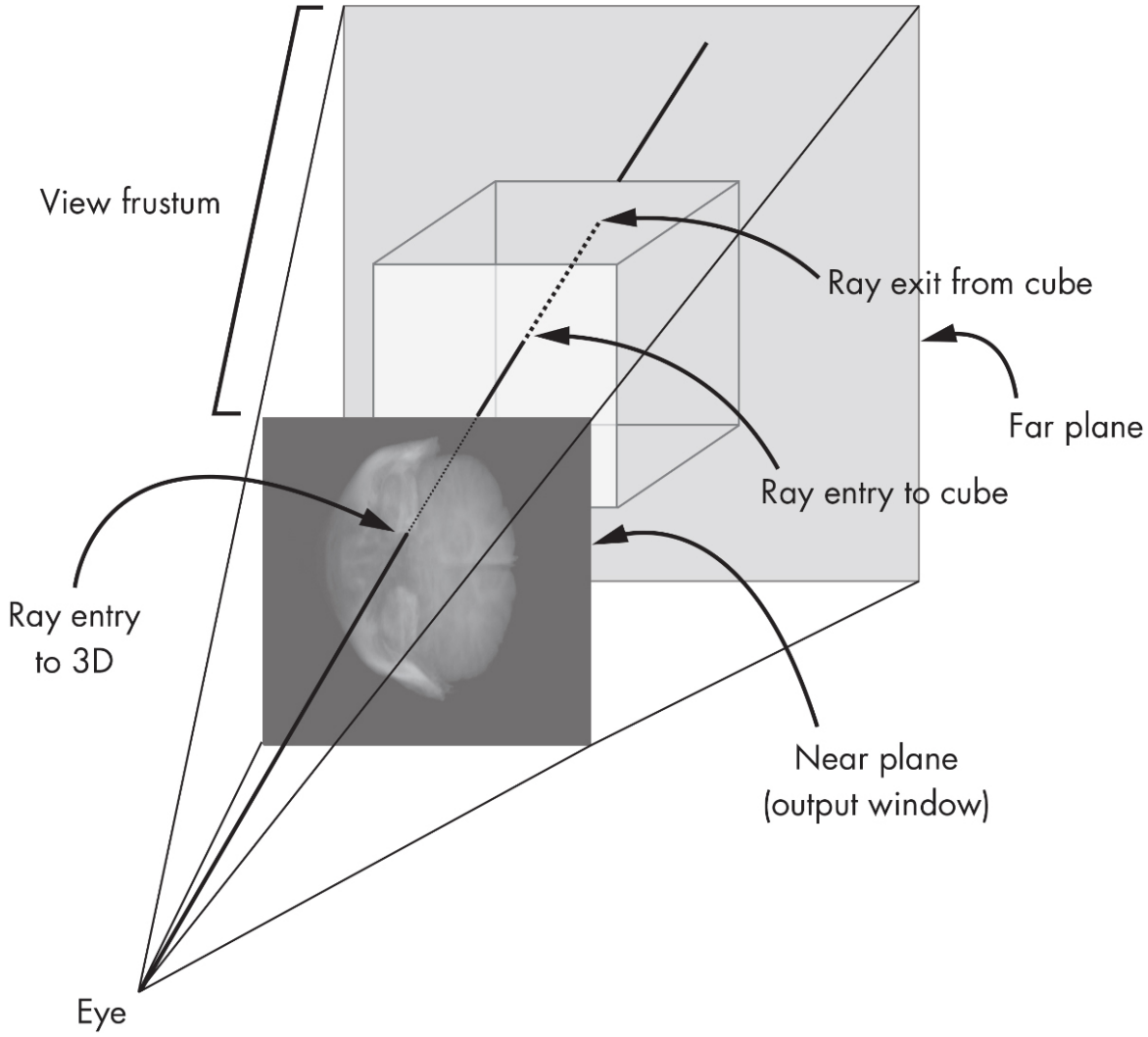


Figure 11-2: *A perspective projection of 3D volumetric data*

To implement ray casting, you need to generate rays that go into the volume. For each pixel in the output window shown in Figure 11-2, you generate a vector *R* that goes into the volume you consider a unit cube (which I'll refer to as the *color cube*) defined between the coordinates (0, 0, 0) and (1, 1, 1). You color each point inside this cube with the RGB values equal to the 3D coordinates of the cube. The origin is colored (0, 0, 0), or black; the (1, 0, 0) corner is red; and the point on the cube diagonally opposite the origin is colored (1, 1, 1), or white. Figure 11-3 shows this cube.

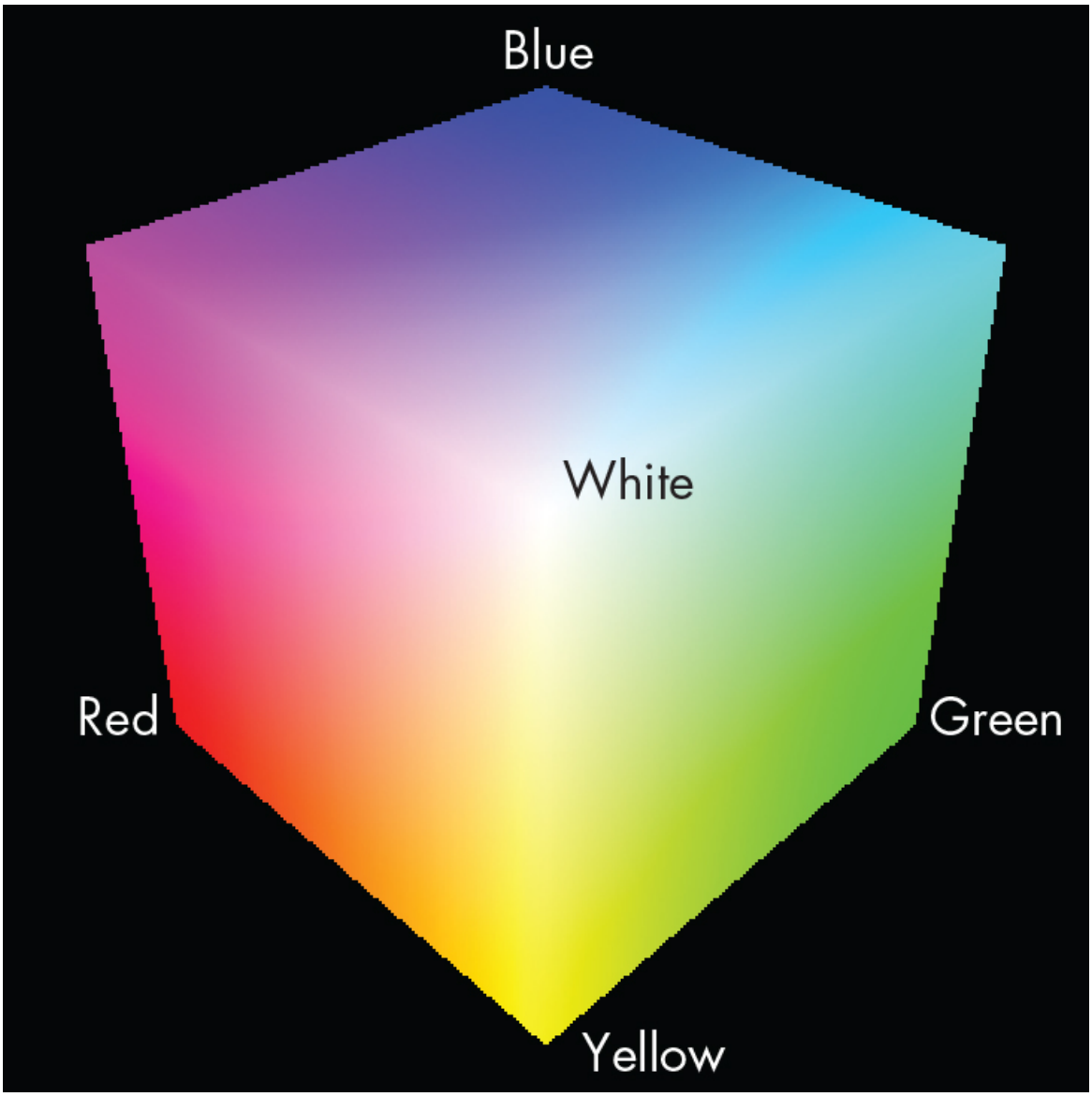


Figure 11-3: *A color cube*

NOTE *In OpenGL, a color can be represented as a strip of 8-bit unsigned values (*r, g, b*), where* r, g, *and* b *are in the range [0, 255]. It can also be represented as a triplet of 32-bit floating-point values (*r, g, b*), where* r, g, *and* b *are in the range [0.0, 1.0]. These representations are equivalent. For example, the red color (255, 0, 0) in the former is the same as (1.0, 0.0, 0.0) in the latter.*

To draw the cube, first draw its six faces using the OpenGL primitive `GL_TRIANGLES`. Then color each vertex and use the interpolation provided by OpenGL when it rasterizes polygons to take care of the colors between each vertex. For example, Figure 11-4(a) shows the three front-faces of the cube. The back-faces of the cube are drawn in Figure 11-4(b) by setting OpenGL to cull front-faces.

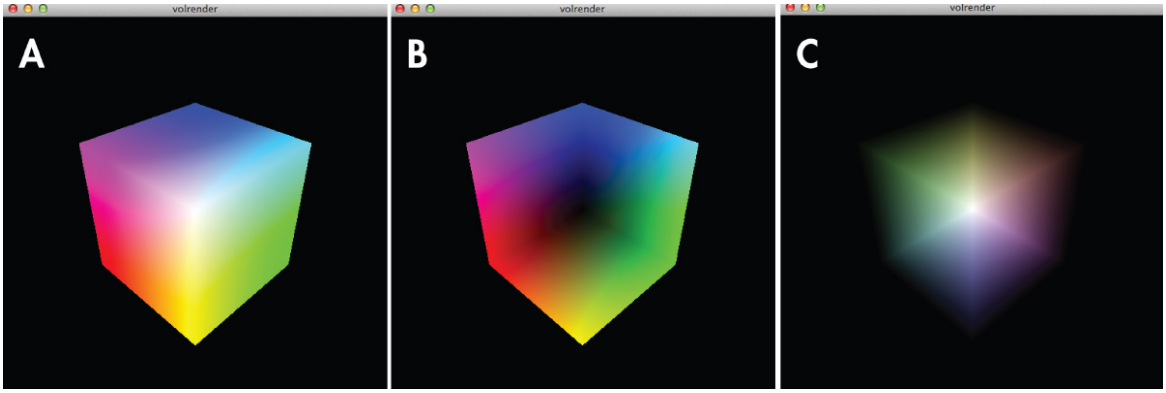


Figure 11-4: *The color cube used to compute rays*

If you subtract the colors in Figure 11-4(a) from Figure 11-4(b) by subtracting $(r, g, b)_{\text{front}}$ from $(r, g, b)_{\text{back}}$, you actually compute a set of vectors that go from the front to the back of the cube because each color (r, g, b) on this cube is the same as the 3D coordinate at that color's location. Figure 11-4(c) shows the result. (Negative values have been flipped to positive for the purposes of this illustration because negative numbers cannot be displayed as colors directly.) Reading the color value (r, g, b) of a pixel, as shown in Figure 11-4(c), gives the (r_x, r_y, r_z) coordinates for the ray passing into the volume at that point.

Once you have the casting rays, you render them into an image or 2D texture for later use with OpenGL's frame buffer object (FBO) feature. After this texture is generated, you can access it inside the shaders that you'll use to implement the ray casting algorithm.

Ray Casting in the GPU

To implement the ray casting algorithm, you first draw the back-faces of the color cube into an FBO. Next, the front-faces are drawn on the screen. The bulk of the ray casting algorithm happens in the fragment shader for this second rendering, which runs for each pixel in the output. The ray is computed by subtracting the front-face color of the incoming fragment from the back-face color of the color cube, which is read in from a texture. The computed ray is then used to accumulate and compute the final pixel value using the 3D volumetric texture data, available within the shader.

Showing 2D Slices

In addition to the 3D rendering, you show 2D slices of the data by extracting the 2D cross section from the 3D data perpendicular to the x-, y-, or z-axis and applying that as a texture on a quad. Because you store the volume as a 3D texture, you can easily get the required data by specifying the texture coordinates (s, t, p). OpenGL's built-in texture interpolation gives you the texture values anywhere inside the 3D texture.

The OpenGL Window

As in your other OpenGL projects, this project uses the GLFW library to display the OpenGL window. You'll use handlers for drawing, for resizing the window, and for keyboard events. You'll use keyboard events to toggle between volume and slice rendering, as well as for rotating and slicing through the 3D data.

Requirements

You'll use `PyOpenGL`, a popular Python binding for OpenGL, for rendering. You'll also use the Python Imaging Library (PIL) to load the 2D images from the volumetric data set, and you'll use `numpy` arrays to represent 3D coordinates and transformation matrices.

The Code

You'll begin by generating a 3D texture from the volumetric data read in from the image files. Next, you'll look at a color cube technique for generating rays from the eye that point into the volume, which is a key concept in implementing the volume ray casting algorithm. You'll look at how to define the cube geometry as well as how to draw the back- and front-faces of this cube. You'll then explore the volume ray casting algorithm and the associated vertex and fragment shaders. Finally, you'll learn how to implement 2D slicing of the volumetric data.

This project has seven Python files:

glutils.py Contains the utility methods for OpenGL shaders, transformations, and so on

makedata.py Contains utility methods for creating volumetric data for testing

raycast.py Implements the `RayCastRender` class for ray casting

raycube.py Implements the `RayCube` class for use in `RayCastRender`

slicerender.py Implements the `SliceRender` class for 2D slicing of volumetric data

volreader.py Contains the utility method to read volumetric data into the OpenGL 3D texture

volrender.py Contains the main methods that create the GLFW window and the renderers

We'll cover all but two of these files in this chapter. The *makedata.py* file lives with the other project files for this chapter at *https://github.com/mkvenkit/pp2e/tree/main/volrender*. The *glutils.py* file can be downloaded from *https://github.com/mkvenkit/pp2e/tree/main/common*.

Generating a 3D Texture

The first step is to read the volumetric data from a folder containing images, as shown in the following code. To see the complete *volreader.py* code, skip ahead to "The Complete 3D Texture Code" on page 241. You can also find the *volreader.py* file at *https://github.com/mkvenkit/pp2e/tree/main/volrender*. Note that the `loadTexture()` function in this file is used to open an image file, read the contents, and create an OpenGL texture object out of it, which is subsequently used in rendering.

```
def loadVolume(dirName):
    """read volume from directory as a 3D texture"""
    # list images in directory
  ❶ files = sorted(os.listdir(dirName))
    print('loading images from: %s' % dirName)
    imgDataList = []
    count = 0
    width, height = 0, 0
    for file in files:
      ❷ file_path = os.path.abspath(os.path.join(dirName,
  file))
        try:
            # read image
          ❸ img = Image.open(file_path)
            imgData = np.array(img.getdata(), np.uint8)
```

```
            # check if all images are of the same size
          ❹ if count is 0:
                width, height = img.size[0], img.size[1]
                imgDataList.append(imgData)
            else:
              ❺ if (width, height) == (img.size[0],
img.size[1]):
                    imgDataList.append(imgData)
            else:
                print('mismatch')
                raise RunTimeError("image size mismatch")
            count += 1
            # print img.size
      ❻ except:
            # skip
            print('Invalid image: %s' % file_path)

    # load image data into single array
    depth = count
  ❼ data = np.concatenate(imgDataList)
    print('volume data dims: %d %d %d' % (width, height,
depth))
```

The `loadVolume()` method first lists the files in the given directory using the `listdir()` method from the `os` module ❶. Then you iterate through the image files themselves, loading them one at a time. To do this, you append the current filename to the directory using `os.path.abspath()` and `os.path.join()` ❷, eliminating the need to deal with relative filepaths and operating system–specific path conventions. (You often see this useful idiom in Python code that traverses files and directories.) Next, you use the `Image` class from the PIL to load the current image into an 8-bit `numpy` array ❸. If the file specified isn't an image or if the image fails to load, an exception is thrown, which you catch by printing an error ❻.

Because you're loading these image slices into a 3D texture, you need to ensure they all have the same

dimensions (width × height), which you confirm at ❹ and ❺. You store the dimensions for the first image and compare them against new incoming images. Once all the images are loaded into individual arrays, you create the final array containing the 3D data by joining these arrays using the `concatenate()` method from `numpy` ❼.

The `loadVolume()` function continues by loading the array of 3D image data into an OpenGL texture:

```
    # load data into 3D texture
  ❶ texture = glGenTextures(1)
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1)
    glBindTexture(GL_TEXTURE_3D, texture)
    glTexParameterf(GL_TEXTURE_3D, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE)
    glTexParameterf(GL_TEXTURE_3D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE)
    glTexParameterf(GL_TEXTURE_3D, GL_TEXTURE_WRAP_R,
GL_CLAMP_TO_EDGE)
    glTexParameterf(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR)
    glTexParameterf(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR)
  ❷ glTexImage3D(GL_TEXTURE_3D, 0, GL_RED,
                 width, height, depth, 0,
                 GL_RED, GL_UNSIGNED_BYTE, data)
    # return texture
  ❸ return (texture, width, height, depth)
```

Here you create an OpenGL texture ❶ and set parameters for filtering and unpacking. Then you load the 3D data array into the OpenGL texture ❷. The format used here is `GL_RED`, and the data format is `GL_UNSIGNED_BYTE` because you have only one 8-bit value associated with each pixel in the data. Finally, you return the OpenGL texture ID and the dimensions of the 3D texture ❸.

Generating Rays

The code for generating the rays is encapsulated in a class called `RayCube`. This class is responsible for drawing the color cube and has methods to draw the back-faces of the cube to an FBO or texture and to draw the front-faces of the cube to the screen. To see the complete *raycube.py* code, skip ahead to "The Complete Ray Generation Code" on page 242. You can also find the *raycube.py* file at *https://github.com/mkvenkit/pp2e/tree/main/volrender*.

First, let's define the shaders used by this class. The shaders will be compiled as part of the `RayCube` class's constructor:

```
strVS = """
# version 410 core

layout(location = 1) in vec3 cubePos;
layout(location = 2) in vec3 cubeCol;

uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
out vec4 vColor;

void main()
{
    // set back-face color
    vColor = vec4(cubeCol.rgb, 1.0);

    // transformed position
    vec4 newPos = vec4(cubePos.xyz, 1.0);

    // set position
    gl_Position = uPMatrix * uMVMatrix * newPos;

}
"""
strFS = """
# version 410 core

in vec4 vColor;
```

```
out vec4 fragColor;

void main()
{
    fragColor = vColor;
}
"""
```

You define the vertex shader used by the `RayCube` class ❶. This shader has two input attributes, `cubePos` and `cubeCol`, which are used to access the position and color values of the vertices, respectively. The modelview and projection matrices are passed in with the uniform variables `uMVMatrix` and `uPMatrix`, respectively. The `vColor` variable is declared as output because it needs to be passed on to the fragment shader, where it will be interpolated. The fragment shader ❷ sets the fragment color to the (interpolated) value of the incoming `vColor` set in the vertex shader.

Defining the Color Cube Geometry

Now let's look at the geometry of the color cube, defined in the `RayCube` class's constructor:

```
class RayCube:
    def __init__(self, width, height):
    --snip--
        # cube vertices
      ❶ vertices = numpy.array([
                0.0, 0.0, 0.0,
                1.0, 0.0, 0.0,
                1.0, 1.0, 0.0,
                0.0, 1.0, 0.0,
                0.0, 0.0, 1.0,
                1.0, 0.0, 1.0,
                1.0, 1.0, 1.0,
                0.0, 1.0, 1.0
```

```
            ], numpy.float32)
    # cube colors
❷   colors = numpy.array([
            0.0, 0.0, 0.0,
            1.0, 0.0, 0.0,
            1.0, 1.0, 0.0,
            0.0, 1.0, 0.0,
            0.0, 0.0, 1.0,
            1.0, 0.0, 1.0,
            1.0, 1.0, 1.0,
            0.0, 1.0, 1.0
            ], numpy.float32)

    # individual triangles
❸   indices = numpy.array([
            4, 5, 7,
            7, 5, 6,
            5, 1, 6,
            6, 1, 2,
            1, 0, 2,
            2, 0, 3,
            0, 4, 3,
            3, 4, 7,
            6, 2, 7,
            7, 2, 3,
            4, 0, 5,
            5, 0, 1
            ], numpy.int16)
```

You define the cube geometry ❶ and the colors ❷ as `numpy` arrays. Notice that the values in these two definitions are the same. As we discussed earlier, the color of each pixel in the color cube corresponds to that pixel's 3D coordinates. The color cube has six faces, each of which can be drawn as two triangles, for a total of 6×6, or 36, vertices. But rather than specify all 36 vertices, you specify just the cube's eight corners ❶ and then define the triangles formed with those corners using an `indices` array ❸, as illustrated in Figure 11-

5. The first two sets of three indices, for example, (4, 5, 7) and (7, 5, 6), define the triangles on the top face of the cube.

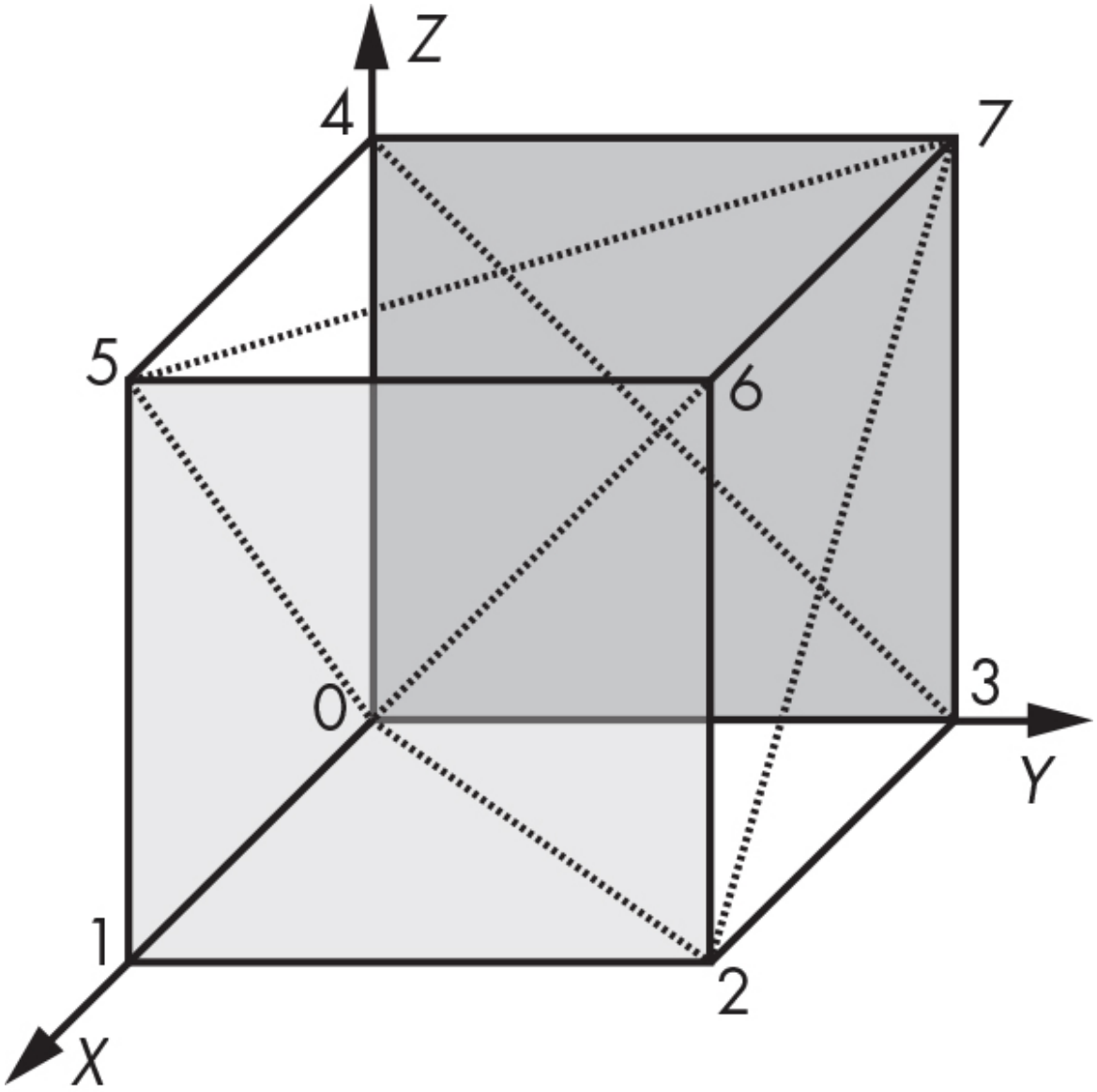


Figure 11-5: *Using indexing, a cube can be represented as a collection of triangles, with each face composed of two triangles.*

Next, still within the `RayCube` class constructor, you need to put the vertex information into buffers:

```
        # set up vertex array object (VAO)
```

```
        self.vao = glGenVertexArrays(1)
        glBindVertexArray(self.vao)

        # vertex buffer
        self.vertexBuffer = glGenBuffers(1)
        glBindBuffer(GL_ARRAY_BUFFER, self.vertexBuffer)
        glBufferData(GL_ARRAY_BUFFER, 4*len(vertices),
    vertices, GL_STATIC_DRAW)

        # vertex buffer — cube vertex colors
        self.colorBuffer = glGenBuffers(1)
        glBindBuffer(GL_ARRAY_BUFFER, self.colorBuffer)
        glBufferData(GL_ARRAY_BUFFER, 4*len(colors), colors,
    GL_STATIC_DRAW)

        # index buffer
        self.indexBuffer = glGenBuffers(1)
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,
    self.indexBuffer); ❶
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, 2*len(indices),
    indices,
                     GL_STATIC_DRAW)
```

As with previous projects, you create and bind to a vertex array object (VAO) and then define the buffers it manages. One difference here is that the `indices` array is given the designation `GL_ELEMENT_ARRAY_BUFFER` ❶, which means the elements in its buffer will be used to index and access the data in the color and vertex buffers.

Creating the Frame Buffer Object

Now let's jump to the `RayCube` class method that creates the frame buffer object, where you'll direct your rendering:

```
def initFBO(self):
    # create frame buffer object
    self.fboHandle = glGenFramebuffers(1)
    # create texture
```

```
  self.texHandle = glGenTextures(1)
  # create depth buffer
  self.depthHandle = glGenRenderbuffers(1)

  # bind
  glBindFramebuffer(GL_FRAMEBUFFER, self.fboHandle)

  glActiveTexture(GL_TEXTURE0)
  glBindTexture(GL_TEXTURE_2D, self.texHandle)

  # set parameters to draw the image at different sizes
❶ glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR)
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR)
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE)
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE)

  # set up texture
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, self.width,
self.height,
               0, GL_RGBA, GL_UNSIGNED_BYTE, None)

  # bind texture to FBO
❷ glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0,
                         GL_TEXTURE_2D, self.texHandle, 0)

  # bind
❸ glBindRenderbuffer(GL_RENDERBUFFER, self.depthHandle)
  glRenderbufferStorage(GL_RENDERBUFFER,
GL_DEPTH_COMPONENT24,
                        self.width, self.height)

  # bind depth buffer to FBO
  glFramebufferRenderbuffer(GL_FRAMEBUFFER,
GL_DEPTH_ATTACHMENT,
                            GL_RENDERBUFFER,
self.depthHandle)

  # check status
❹ status = glCheckFramebufferStatus(GL_FRAMEBUFFER)
  if status == GL_FRAMEBUFFER_COMPLETE:
```

```
        pass
        # print "fbo %d complete" % self.fboHandle
    elif status == GL_FRAMEBUFFER_UNSUPPORTED:
        print "fbo %d unsupported" % self.fboHandle
    else:
        print "fbo %d Error" % self.fboHandle
```

Here you create a frame buffer object, a 2D texture, and a render buffer object; then you set up the texture parameters ❶. The texture is bound to the frame buffer ❷, and at ❸ and in the lines that follow, the render buffer sets up a 24-bit depth buffer and is attached to the frame buffer. Next, you check the status of the frame buffers ❹ and print a status message if something goes wrong. Now, as long as the frame buffer and render buffer are bound correctly, all of your rendering will go into the texture.

Rendering the Back-Faces of the Cube

Here is the code for rendering the back-faces of the color cube:

```
def renderBackFace(self, pMatrix, mvMatrix):
    """renders back-face of ray-cube to a texture and returns
  it"""
    # render to FBO
  ❶ glBindFramebuffer(GL_FRAMEBUFFER, self.fboHandle)
    # set active texture
    glActiveTexture(GL_TEXTURE0)
    # bind to FBO texture
    glBindTexture(GL_TEXTURE_2D, self.texHandle)

    # render cube with face culling enabled
  ❷ self.renderCube(pMatrix, mvMatrix, self.program, True)

    # unbind texture
  ❸ glBindTexture(GL_TEXTURE_2D, 0)
    glBindFramebuffer(GL_FRAMEBUFFER, 0)
```

```
    glBindRenderbuffer(GL_RENDERBUFFER, 0)

    # return texture ID
  ❹ return self.texHandle
```

First you bind the FBO ❶, set the active texture unit, and bind to the texture handle so that you can render to the FBO. Then you call the `RayCube` class's `renderCube()` method ❷, which we'll look at soon. It has a face-culling flag as an argument, allowing you to draw either the front-face or the back-face of the cube using the same code. You set the flag to `True` to make the back-faces appear in the FBO texture.

Next, you make the necessary calls to unbind from the FBO so that other rendering code is unaffected ❸. Finally, you return the FBO texture ID ❹ for use in the next stage of the algorithm.

Rendering the Front-Faces of the Cube

The following code is used to draw the front-faces of the color cube during the second rendering pass of the ray casting algorithm:

```
def renderFrontFace(self, pMatrix, mvMatrix, program):
    """render front-face of ray-cube"""
    # no face culling
    self.renderCube(pMatrix, mvMatrix, program, False)
```

This method simply calls `renderCube()`, with the face-culling flag set to `False` so the front-faces will appear.

Rendering the Whole Cube

Now let's look at the `renderCube()` method, which draws the color cube discussed previously:

```
def renderCube(self, pMatrix, mvMatrix, program, cullFace):
    """renderCube uses face culling if flag set"""

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

    # set shader program
    glUseProgram(program)

    # set projection matrix
    glUniformMatrix4fv(glGetUniformLocation(program,
  b'uPMatrix'),
                       1, GL_FALSE, pMatrix)

    # set modelview matrix
    glUniformMatrix4fv(glGetUniformLocation(program,
  b'uMVMatrix'),
                       1, GL_FALSE, mvMatrix)

    # enable face culling
    glDisable(GL_CULL_FACE)
  ❶ if cullFace:
        glFrontFace(GL_CCW)
        glCullFace(GL_FRONT)
        glEnable(GL_CULL_FACE)

    # bind VAO
    glBindVertexArray(self.vao)

    # animated slice
  ❷ glDrawElements(GL_TRIANGLES, self.nIndices,
  GL_UNSIGNED_SHORT, None)

    # unbind VAO
    glBindVertexArray(0)

    # reset cull face
    if cullFace:
        # disable face culling
        glDisable(GL_CULL_FACE)
```

You clear the color and depth buffers, select the shader program, and set the transformation matrices. Then you set a flag to control face culling ❶, which determines whether the cube’s front-face or back-face is drawn. Notice that you

use `glDrawElements()` ❷ because you're using an index array to render the cube, rather than a vertex array.

Resizing the Window

Because the FBO is created for a particular window size, you need to re-create it when the window size changes. To do that, you create a resize handler for the `RayCube` class, as shown here:

```
def reshape(self, width, height):
    self.width = width
    self.height = height
    self.aspect = width/float(height)
    # re-create FBO
    self.clearFBO()
    self.initFBO()
```

The `reshape()` method is called when the OpenGL window is resized. It checks the new window dimensions and then clears and re-creates the FBO.

Implementing the Ray Casting Algorithm

Next, you'll implement the ray casting algorithm in the `RayCastRender` class. The core of the algorithm happens inside the fragment shader used by this class, which also uses the `RayCube` class to help generate the rays. To see the complete *raycast.py* code, skip ahead to "The Complete Volume Ray Casting Code" on page 248. You can also find this file at *https://github.com/mkvenkit/pp2e/tree/main/volrender*.

Begin in the `RayCastRender` constructor by creating a `RayCube` object and loading the shaders:

```
class RayCastRender:
    def __init__(self, width, height, volume):
        """RayCastRender construction"""

        # create RayCube object
      ❶ self.raycube = raycube.RayCube(width, height)

        # set dimensions
        self.width = width
        self.height = height
        self.aspect = width/float(height)

        # create shader
      ❷ self.program = glutils.loadShaders(strVS, strFS)
        # texture
      ❸ self.texVolume, self.Nx, self.Ny, self.Nz = volume

        # initialize camera
      ❹ self.camera = Camera()
```

The constructor creates an object of type `RayCube` ❶, which is used to generate rays. You load the shaders used by the ray casting ❷ and then set the OpenGL 3D texture and dimensions ❸, which were passed into the constructor as the tuple `volume`. Next, you create a `Camera` object ❹, which you'll use to set up the OpenGL perspective transformation for the 3D rendering.

NOTE *The `Camera` class, also declared in* raycast.py*, is basically the same as the one used in Chapter 10. You'll see it in the complete code listing on page 248.*

Here is the rendering method for `RayCastRender`:

```
def draw(self):

    # build projection matrix
  ❶ pMatrix = glutils.perspective(45.0, self.aspect, 0.1,
  100.0)
```

```
  # modelview matrix
❷ mvMatrix = glutils.lookAt(self.camera.eye,
self.camera.center,
                            self.camera.up)

  # render

  # generate ray-cube back-face texture
❸ texture = self.raycube.renderBackFace(pMatrix, mvMatrix)

  # set shader program
❹ glUseProgram(self.program)

  # set window dimensions
  glUniform2f(glGetUniformLocation(self.program,
b"uWinDims"),
              float(self.width), float(self.height))

  # bind to texture unit 0, which represents back-faces of
cube
❺ glActiveTexture(GL_TEXTURE0)
  glBindTexture(GL_TEXTURE_2D, texture)
  glUniform1i(glGetUniformLocation(self.program,
b"texBackFaces"), 0)

  # texture unit 1: 3D volume texture
❻ glActiveTexture(GL_TEXTURE1)
  glBindTexture(GL_TEXTURE_3D, self.texVolume)
  glUniform1i(glGetUniformLocation(self.program,
b"texVolume"), 1)

  # draw front-face of cubes
❼ self.raycube.renderFrontFace(pMatrix, mvMatrix,
self.program)
```

First you set up a perspective projection matrix for the rendering, using the `glutils.perspective()` utility method ❶. Then you set the current camera parameters into the `glutils.lookAt()` method ❷. Next, the first pass of the rendering is done ❸, using the `renderBackFace()` method in `RayCube` to draw the back-faces of the color cube into a texture. (This method also returns the ID of the generated texture.)

You continue by enabling the shaders for the ray casting algorithm ❹. Then you set up the textures for use in the shader program. The texture returned at ❸ is set up as texture unit 0 ❺, and the 3D texture created from the volumetric data you read in is set up as texture unit 1 ❻. Finally, you render the front-faces of the cube using the `renderFrontFace()` method in `RayCube` ❼. When this code is executed, the shaders for `RayCastRender` will act on the vertices and fragments.

The Vertex Shader

Now you come to the shaders used by `RayCastRender`. Let's look at the vertex shader first:

```
strVS = """
# version 410 core

layout(location = 1) in vec3 cubePos;
layout(location = 2) in vec3 cubeCol;

uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;

out vec4 vColor;

void main()
{
    // set position
  ❹ gl_Position = uPMatrix * uMVMatrix * vec4(cubePos.xyz,
  1.0);

    // set color
  ❺ vColor = vec4(cubeCol.rgb, 1.0);
}
"""
```

First you set the input variables of position and color ❶. The layout uses the same indices as defined in the `RayCube`

vertex shader because `RayCastRender` uses the VBO defined in that class to draw the geometry, and the locations in the shaders have to match. Then you define the input transformation matrices ❷ and set a color value as the shader output ❸. The usual transformation computes the built-in `gl_Position` output ❹, before you set the output as the current color of the cube vertex ❺. The latter will be interpolated across vertices to give you the correct color in the fragment shader.

The Fragment Shader

The fragment shader is the star of the show. It implements the core of the ray casting algorithm.

```
strFS = """
# version 410 core

in vec4 vColor;

uniform sampler2D texBackFaces;
uniform sampler3D texVolume;
uniform vec2 uWinDims;

out vec4 fragColor;

void main()
{
    // start of ray
  ❶ vec3 start = vColor.rgb;

    // calculate texture coordinates at fragment,
    // which is a fraction of window coordinates
  ❷ vec2 texc = gl_FragCoord.xy/uWinDims.xy;

    // get end of ray by looking up back-face color
  ❸ vec3 end = texture(texBackFaces, texc).rgb;

    // calculate ray direction
  ❹ vec3 dir = end - start;

    // normalized ray direction
```

```
    vec3 norm_dir = normalize(dir);

    // the length from front to back is calculated and
    // used to terminate the ray
    float len = length(dir.xyz);

    // ray step size
    float stepSize = 0.01;

    // X-ray projection
    vec4 dst = vec4(0.0);

    // step through the ray
  ❺ for(float t = 0.0; t < len; t += stepSize) {

        // set position to endpoint of ray
      ❻ vec3 samplePos = start + t*norm_dir;

        // get texture value at position
      ❼ float val = texture(texVolume, samplePos).r;
        vec4 src = vec4(val);

        // set opacity
      ❽ src.a *= 0.1;
        src.rgb *= src.a;

        // blend with previous value
      ❾ dst = (1.0 - dst.a)*src + dst;

        // exit loop when alpha exceeds threshold
      ❿ if(dst.a >= 0.95)
            break;
    }

    // set fragment color
    fragColor = dst;
}
"""
```

The input to the fragment shader is the cube vertex color. The fragment shader also has access to the 2D texture generated by rendering the color cube, the 3D texture containing the volumetric data, and the dimensions of the OpenGL window.

While the fragment shader executes, you send in the front-faces of the cube, so by looking up the incoming color value ❶, you get the starting point of the ray that goes into this cube. (Recall the discussion in “Ray Generation” on page 217 about the connection between the colors in the cube and the ray directions.)

You calculate the texture coordinate of the incoming fragment on the screen ❷. Here, dividing the location of the fragment in window coordinates by the window dimensions maps the location to the range [0, 1]. The ending point of the ray is obtained by looking up the back-face color of the cube using this texture coordinate ❸.

You next calculate the ray direction ❹ and then calculate the normalized direction and length of this ray, which will be useful in the ray casting computation. Then you loop through the volume using the ray’s starting point and direction until it hits the ray’s endpoint ❺. In this loop, you compute the ray’s current position inside the data volume ❻ and look up the data value at this point ❼. You then perform the blending equation at ❽ and ❾, which gives you the X-ray effect. You combine the `dst` value with the current value of the intensity (which is attenuated using the alpha value), and the process continues along the ray. The alpha value keeps increasing until it equals the maximum threshold of 0.95 ❿, at which point you exit the loop. The end result is a sort of average opacity through the volume at each pixel, which produces a “see-through” or X-ray effect. (Try varying the threshold and alpha attenuation to produce different effects.)

Showing 2D Slices

In addition to showing the 3D view of the volumetric data, you want to show 2D slices of the data in the x-, y-, and z-directions onscreen. The code for this is encapsulated in a

class called `SliceRender`, which creates 2D volumetric slices. To see the complete *slicerender.py* code, skip ahead to "The Complete 2D Slicing Code" on page 251. You can also find the *slicerender.py* file at *https://github.com/mkvenkit/pp2e/tree/main/volrender*.

Here's the initialization code in the `SliceRender` class constructor that sets up the geometry for the slices:

```
class SliceRender:
    def __init__(self, width, height, volume):
    --snip--
        # set up vertex array object (VAO)
        self.vao = glGenVertexArrays(1)
        glBindVertexArray(self.vao)

        # define quad vertices
      ❶ vertexData = numpy.array([0.0, 1.0, 0.0,
                                  0.0, 0.0, 0.0,
                                  1.0, 1.0, 0.0,
                                  1.0, 0.0, 0.0],
  numpy.float32)

        # vertex buffer
        self.vertexBuffer = glGenBuffers(1)
        glBindBuffer(GL_ARRAY_BUFFER, self.vertexBuffer)
        glBufferData(GL_ARRAY_BUFFER, 4*len(vertexData),
  vertexData,
                     GL_STATIC_DRAW)
        # enable arrays
        glEnableVertexAttribArray(self.vertIndex)
        # set buffers
        glBindBuffer(GL_ARRAY_BUFFER, self.vertexBuffer)
        glVertexAttribPointer(self.vertIndex, 3, GL_FLOAT,
  GL_FALSE, 0, None)

        # unbind VAO
        glBindVertexArray(0)
```

This code sets up a VAO to manage the VBO, as in earlier examples. You define the geometry of a square in the XY plane ❶. (The vertex order is that of the `GL_TRIANGLE_STRIP`, introduced in Chapter 9.) You'll use this same geometry regardless of whether you're showing slices perpendicular to *x*, *y*, or *z*. All that changes between these cases is the data plane that you pick to display from within the 3D texture. We'll return to this idea when we look at the vertex shader.

Here's a method to render the 2D slices:

```
def draw(self):
    # clear buffers
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    # build projection matrix
  ❶ pMatrix = glutils.ortho(-0.6, 0.6, -0.6, 0.6, 0.1, 100.0)
    # modelview matrix
  ❷ mvMatrix = numpy.array([1.0, 0.0, 0.0, 0.0,
                            0.0, 1.0, 0.0, 0.0,
                            0.0, 0.0, 1.0, 0.0,
                            -0.5, -0.5, -1.0, 1.0],
  numpy.float32)
    # use shader
    glUseProgram(self.program)

    # set projection matrix
    glUniformMatrix4fv(self.pMatrixUniform, 1, GL_FALSE,
  pMatrix)

    # set modelview matrix
    glUniformMatrix4fv(self.mvMatrixUniform, 1, GL_FALSE,
  mvMatrix)

    # set current slice fraction
  ❸ glUniform1f(glGetUniformLocation(self.program,
  b"uSliceFrac"),
                float(self.currSliceIndex)/float(self.currSlice
  Max))
    # set current slice mode
  ❹ glUniform1i(glGetUniformLocation(self.program,
  b"uSliceMode"),
```

```
                self.mode)

    # enable texture
    glActiveTexture(GL_TEXTURE0)
    glBindTexture(GL_TEXTURE_3D, self.texture)
    glUniform1i(glGetUniformLocation(self.program, b"tex"), 0)

    # bind VAO
    glBindVertexArray(self.vao)
    # draw
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4)
    # unbind VAO
    glBindVertexArray(0)
```

Each 2D slice is a square, which you build up using an OpenGL triangle strip primitive. This code goes through the render setup for the triangle strip. Note that you implement the orthographic projection using the `glutils.ortho()` method at ❶. You set up a projection that adds a 0.1 buffer around the unit square representing the slice.

When you draw something with OpenGL, the default view (without any transformation applied) puts the eye at (0, 0, 0) looking down the z-axis with the y-axis pointing up. Applying the translation (−0.5, −0.5, −1.0) to your geometry centers it on the z-axis ❷. You set the current slice fraction ❸ (where, for example, the 10th slice out of 100 would be 0.1), set the slice mode ❹ (to view the slices in the x-, y-, or z-direction, as represented by the integers 0, 1, and 2, respectively), and set both values to the shaders.

The Vertex Shader

Now let’s look at the vertex shader for `SliceRender`:

```
strVS = """
# version 410 core

in vec3 aVert;
```

```
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;

uniform float uSliceFrac;
uniform int uSliceMode;

out vec3 texcoord;

void main() {

    // x slice
    if (uSliceMode == 0) {
      ❶ texcoord = vec3(uSliceFrac, aVert.x, 1.0-aVert.y);
    }
    // y slice
    else if (uSliceMode == 1) {
      ❷ texcoord = vec3(aVert.x, uSliceFrac, 1.0-aVert.y);
    }
    // z slice
    else {
      ❸ texcoord = vec3(aVert.x, 1.0-aVert.y, uSliceFrac);
    }

    // calculate transformed vertex
    gl_Position = uPMatrix * uMVMatrix * vec4(aVert, 1.0);
}
"""
```

The vertex shader takes the triangle strip vertex array as input and sets a texture coordinate as output. The current slice fraction and slice mode are passed in as uniform variables `uSliceFrac` and `uSliceMode`.

The shader has three branches, depending on the slice mode. For example, if `uSliceMode` is `0`, you calculate the texture coordinates for an *x* slice ❶. Because you're slicing perpendicular to the x-direction, you want a slice parallel to the YZ plane. The 3D vertices coming in to the vertex shader also double as the 3D texture coordinates because they are in the range [0, 1], so the texture coordinates are given as (f, V_x, V_y), where f is the fraction of the slice

number in the direction of the x-axis and where V_x and V_y are the vertex coordinates. Unfortunately, the resulting image will appear upside down because the OpenGL coordinate system has its origin at the bottom left, with the y-direction pointing up; this is the reverse of what you want. To resolve this problem, you change the texture coordinate t to $(1 - t)$ and use $(f, V_x, 1 - V_y)$ ❶. You use similar logic to compute the texture coordinates for y- ❷ and z- ❸ direction slices if the `uSliceMode` value is `1` or `2`, respectively.

The Fragment Shader

Here is the fragment shader:

```
strFS = """
# version 410 core

in vec3 texcoord;

uniform sampler3D texture;

out vec4 fragColor;

void main() {
    // look up color in texture
  ❸ vec4 col = texture(tex, texcoord);
  ❹ fragColor = col.rrra;
}
"""
```

The fragment shader declares `texcoord` as input ❶, which was set as output in the vertex shader. The texture sampler is declared as `uniform` ❷. You look up the texture color using `texcoord` ❸ and set `fragColor` as the output ❹. (Because you read in your texture only as the red channel, you use `col.rrra`.)

A User Interface for 2D Slicing

Now you need a way for the user to slice through the data. Do this using a keyboard handling method on the `SliceRender` class:

```
def keyPressed(self, key):
    """keypress handler"""
  ❶ if key == 'x':
        self.mode = SliceRender.XSLICE
        # reset slice index
        self.currSliceIndex = int(self.Nx/2)
        self.currSliceMax = self.Nx
    elif key == 'y':
        self.mode = SliceRender.YSLICE
        # reset slice index
        self.currSliceIndex = int(self.Ny/2)
        self.currSliceMax = self.Ny
    elif key == 'z':
        self.mode = SliceRender.ZSLICE
        # reset slice index
        self.currSliceIndex = int(self.Nz/2)
        self.currSliceMax = self.Nz
    elif key == 'l':
      ❷ self.currSliceIndex = (self.currSliceIndex + 1) %
  self.currSliceMax
    elif key == 'r':
        self.currSliceIndex = (self.currSliceIndex - 1) %
  self.currSliceMax
```

When the X, Y, or Z key on the keyboard is pressed, `SliceRender` switches to the *x*, *y*, or *z* slice mode. You can see this in action for the *x* slice, for example ❶, where you set the appropriate mode, set the current slice index to the middle of the data, and update the maximum slice number.

When the left or right arrow key on the keyboard is pressed, you page through the slices. For instance, the slice

index is incremented ❷ when the left arrow key is pressed. The modulo operator (`%`) ensures that the index "rolls over" to 0 when you exceed the maximum value.

Putting the Code Together

Let's take a quick look at the main file in the project, *volrender.py*. This file uses a class `RenderWin`, which creates and manages the GLFW OpenGL window. (I won't cover this class in detail because it's similar to the class used in Chapters 9 and 10.) To see the complete *volrender.py* code, skip ahead to "The Complete Main File Code" on page 254. You can also find the *volrender.py* file at *https://github.com/mkvenkit/pp2e/tree/main/volrender*.

In the initialization code for this class, you create the renderer as follows:

```
class RenderWin:
    def __init__(self, imageDir):
    --snip--
        # load volume data
      ❶ self.volume = volreader.loadVolume(imageDir)
        # create renderer
      ❷ self.renderer = RayCastRender(self.width, self.height,
  self.volume)
```

Here you read the 3D data into an OpenGL texture using the `loadVolume()` function we discussed earlier ❶. Then you create an object of type `RayCastRender` to display the data ❷.

The Keypress Handler

The `RenderWindow` class needs its own keyboard handler method for toggling between volume and slice rendering and for closing the window. This method also passes along

keypresses to the `RayCastRender` and `SliceRender` classes' keyboard handlers, to either rotate the camera or navigate through the 2D slices.

```
def onKeyboard(self, win, key, scancode, action, mods):
    # print 'keyboard: ', win, key, scancode, action, mods
    # ESC to quit
    if key is glfw.GLFW_KEY_ESCAPE:
        self.renderer.close()
        self.exitNow = True
    else:
      ❶ if action is glfw.GLFW_PRESS or action is
  glfw.GLFW_REPEAT:
            if key == glfw.GLFW_KEY_V:
                # toggle render mode
              ❷ if isinstance(self.renderer, RayCastRender):
                    self.renderer = SliceRender(self.width,
  self.height,
                                                self.volume)
                else:
                    self.renderer = RayCastRender(self.width,
  self.height,
                                                  self.volume)
                # call reshape on renderer
                self.renderer.reshape(self.width, self.height)
            else:
                # send keypress to renderer
              ❸ keyDict = {glfw.GLFW_KEY_X: 'x',
  glfw.GLFW_KEY_Y: 'y',
                            glfw.GLFW_KEY_Z: 'z',
  glfw.GLFW_KEY_LEFT: 'l',
                            glfw.GLFW_KEY_RIGHT: 'r'}
                try:
                    self.renderer.keyPressed(keyDict[key])
                except:
                    pass
```

Pressing ESC quits the program. You set other keypresses to work whether you have just pressed the key down or you are keeping it pressed ❶. If the V key is pressed, you toggle the renderer between volume and slice ❷, using Python's `isinstance()` method to identify the current class type. To handle any other keypress events (X, Y, Z, or the left and right arrows), you use a dictionary ❸ and pass the keypress to the current renderer's `keyPressed()` handler method. We looked at the slice renderer's `keyPressed()` method in "A User Interface for 2D Slicing" on page 237, for example.

NOTE *I'm choosing not to pass in the `glfw.KEY` values directly and using a dictionary to convert these to character values instead, because it's a good practice to reduce dependencies in source files. Currently, the only file in this project that depends on GLFW is* volrender.py*. If you were to pass GLFW-specific types into other code, they would need to import and depend on the GLFW library too. Then, if you were to switch to a different OpenGL windowing toolkit, the code would become messy.*

Running the Program

Here is a sample run of the application using data from the Stanford Volume Data Archive:

```
$ python volrender.py --dir mrbrain-8bit/
```

You should see something like Figure 11-6.

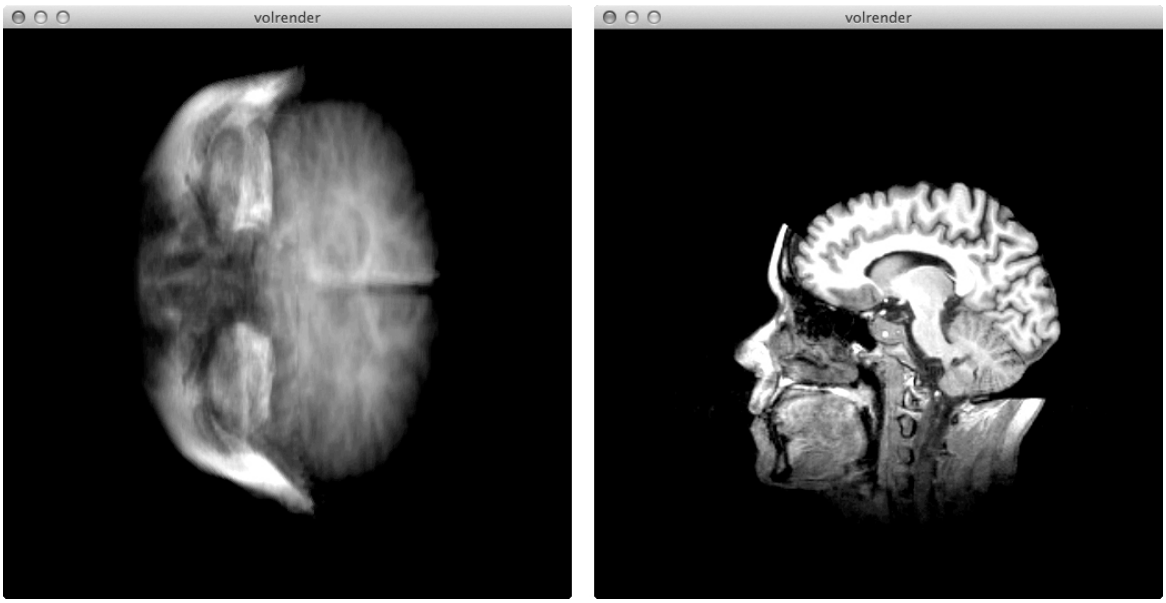


Figure 11-6: *A sample run of* volrender.py*. The image on the left is the volumetric rendering, and the image on the right is a 2D slice.*

As the application runs, use the V key to toggle between volume and slice rendering. In slice mode, use the X, Y, and Z keys to change the slicing axis, and use the arrow keys to change the slicing location.

Summary

In this chapter, you implemented the volume ray casting algorithm using Python and OpenGL. You learned how to use GLSL shaders to implement this algorithm efficiently, as well as how to create 2D slices from the volumetric data.

Experiments!

Here are a few ways you could keep tinkering with the volume ray casting program:

1. Currently, it's hard to see the boundary of the volumetric data "cube" in the ray casting mode. Implement a class `WireFrame` that draws a box around this cube. Color the x-, y-, and z-axes red, green, and blue, respectively, and give each its own shaders. You'll use `WireFrame` from within the `RayCastRender` class.
2. Implement data scaling. In the current implementation, you're drawing a cube for the volume and a square for 2D slices, which assumes you have a symmetric data set (that the number of slices are the same in each direction), but most real data has a varying number of slices. Medical data, in particular, often has fewer slices in the z-direction, with dimensions such as 256×256×99, for example. To display this data correctly, you have to introduce a scale into your computations. One way to do so is to apply the scale to the cube vertices (3D volume) and square vertices (2D slice). The user can then input the scaling parameters as command line arguments.
3. Our volume ray casting implementation uses X-ray casting to calculate the final color or intensity of a pixel. Another popular way to do this is to use *maximum intensity projection (MIP)* to set the maximum intensity at each pixel. Implement this in your code. (Hint: in the fragment shader of `RayCastRender`, modify the code that steps through the ray to check and set the maximum value along the ray, instead of blending values.)
4. Currently, the only UI you have implemented is rotation around the x-, y-, and z-axes. Implement a zoom feature so pressing I/O will zoom in/out of the volume-rendered image. You could do this by setting the appropriate camera parameters in the

`glutils.lookAt()` method, with one caveat: if you move your view inside the data cube, the ray casting will fail because OpenGL will clip the front-faces of the cube; the ray computation needed for ray casting requires both the front- and back-faces of the color cube to be rendered correctly. Instead, zoom by adjusting the field of view in the `glutils.projecton()` method.

The Complete 3D Texture Code

Here's the full *volreader.py* code listing.

```
import os
import numpy as np
from PIL import Image

import OpenGL
from OpenGL.GL import *

from scipy import misc

def loadVolume(dirName):
    """read volume from directory as a 3D texture"""
    # list images in directory
    files = sorted(os.listdir(dirName))
    print('loading images from: %s' % dirName)
    imgDataList = []
    count = 0
    width, height = 0, 0
    for file in files:
        file_path = os.path.abspath(os.path.join(dirName,
  file))
        try:
            # read image
            img = Image.open(file_path)
            imgData = np.array(img.getdata(), np.uint8)
```

```
            # check if all are of the same size
            if count is 0:
                width, height = img.size[0], img.size[1]
                imgDataList.append(imgData)
            else:
                if (width, height) == (img.size[0],
img.size[1]):
                    imgDataList.append(imgData)
                else:
                    print('mismatch')
                    raise RunTimeError("image size mismatch")
            count += 1
            # print img.size
        except:
            # skip
            print('Invalid image: %s' % file_path)

    # load image data into single array
    depth = count
    data = np.concatenate(imgDataList)
    print('volume data dims: %d %d %d' % (width, height,
depth))

    # load data into 3D texture
    texture = glGenTextures(1)
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1)
    glBindTexture(GL_TEXTURE_3D, texture)
    glTexParameterf(GL_TEXTURE_3D, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE)
    glTexParameterf(GL_TEXTURE_3D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE)
    glTexParameterf(GL_TEXTURE_3D, GL_TEXTURE_WRAP_R,
GL_CLAMP_TO_EDGE)
    glTexParameterf(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR)
    glTexParameterf(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR)
    glTexImage3D(GL_TEXTURE_3D, 0, GL_RED,
                 width, height, depth, 0,
                 GL_RED, GL_UNSIGNED_BYTE, data)
    # return texture
```

```
    return (texture, width, height, depth)

# load texture
def loadTexture(filename):
    img = Image.open(filename)
    img_data = np.array(list(img.getdata()), 'B')
    texture = glGenTextures(1)
    glPixelStorei(GL_UNPACK_ALIGNMENT,1)
    glBindTexture(GL_TEXTURE_2D, texture)
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
  GL_CLAMP_TO_EDGE)
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
  GL_CLAMP_TO_EDGE)
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
  GL_LINEAR)
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
  GL_LINEAR)
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, img.size[0],
  img.size[1],
                 0, GL_RGBA, GL_UNSIGNED_BYTE, img_data)
    return texture
```

The Complete Ray Generation Code

Here's the full code listing for the `RayCube` class.

```
import OpenGL
from OpenGL.GL import *
from OpenGL.GL.shaders import *

import numpy, math, sys
import volreader, glutils

strVS = """
# version 330 core

layout(location = 1) in vec3 cubePos;
layout(location = 2) in vec3 cubeCol;

uniform mat4 uMVMatrix;
```

```
uniform mat4 uPMatrix;
out vec4 vColor;

void main()
{
    // set back-face color
    vColor = vec4(cubeCol.rgb, 1.0);

    // transformed position
    vec4 newPos = vec4(cubePos.xyz, 1.0);

    // set position
    gl_Position = uPMatrix * uMVMatrix * newPos;

}
"""
strFS = """
# version 330 core

in vec4 vColor;
out vec4 fragColor;

void main()
{
    fragColor = vColor;
}
"""

class RayCube:
    """class used to generate rays used in ray casting"""

    def __init__(self, width, height):
        """RayCube constructor"""

        # set dims
        self.width, self.height = width, height

        # create shader
        self.program = glutils.loadShaders(strVS, strFS)

        # cube vertices
        vertices = numpy.array([
                0.0, 0.0, 0.0,
                1.0, 0.0, 0.0,
                1.0, 1.0, 0.0,
```

```
        0.0, 1.0, 0.0,
        0.0, 0.0, 1.0,
        1.0, 0.0, 1.0,
        1.0, 1.0, 1.0,
        0.0, 1.0, 1.0
        ], numpy.float32)

# cube colors
colors = numpy.array([
        0.0, 0.0, 0.0,
        1.0, 0.0, 0.0,
        1.0, 1.0, 0.0,
        0.0, 1.0, 0.0,
        0.0, 0.0, 1.0,
        1.0, 0.0, 1.0,
        1.0, 1.0, 1.0,
        0.0, 1.0, 1.0
        ], numpy.float32)

# individual triangles
indices = numpy.array([
        4, 5, 7,
        7, 5, 6,
        5, 1, 6,
        6, 1, 2,
        1, 0, 2,
        2, 0, 3,
        0, 4, 3,
        3, 4, 7,
        6, 2, 7,
        7, 2, 3,
        4, 0, 5,
        5, 0, 1
        ], numpy.int16)

self.nIndices = indices.size

# set up vertex array object (VAO)
self.vao = glGenVertexArrays(1)
glBindVertexArray(self.vao)

# vertex buffer
```

```
    self.vertexBuffer = glGenBuffers(1)
    glBindBuffer(GL_ARRAY_BUFFER, self.vertexBuffer)
    glBufferData(GL_ARRAY_BUFFER, 4*len(vertices),
vertices, GL_STATIC_DRAW)

    # vertex buffer - cube vertex colors
    self.colorBuffer = glGenBuffers(1)
    glBindBuffer(GL_ARRAY_BUFFER, self.colorBuffer)
    glBufferData(GL_ARRAY_BUFFER, 4*len(colors), colors,
GL_STATIC_DRAW);

    # index buffer
    self.indexBuffer = glGenBuffers(1)
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,
self.indexBuffer);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, 2*len(indices),
indices,
                 GL_STATIC_DRAW)

    # enable attrs using the layout indices in shader
    aPosLoc = 1
    aColorLoc = 2

    # bind buffers
    glEnableVertexAttribArray(1)
    glEnableVertexAttribArray(2)

    # vertex
    glBindBuffer(GL_ARRAY_BUFFER, self.vertexBuffer)
    glVertexAttribPointer(aPosLoc, 3, GL_FLOAT, GL_FALSE,
0, None)

    # color
    glBindBuffer(GL_ARRAY_BUFFER, self.colorBuffer)
    glVertexAttribPointer(aColorLoc, 3, GL_FLOAT, GL_FALSE,
0, None)
    # index
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, self.indexBuffer)

    # unbind VAO
    glBindVertexArray(0)

    # FBO
    self.initFBO()
```

```
def renderBackFace(self, pMatrix, mvMatrix):
    """renders back-face of ray-cube to a texture and
 returns it"""
    # render to FBO
    glBindFramebuffer(GL_FRAMEBUFFER, self.fboHandle)
    # set active texture
    glActiveTexture(GL_TEXTURE0)
    # bind to FBO texture
    glBindTexture(GL_TEXTURE_2D, self.texHandle)

    # render cube with face culling enabled
    self.renderCube(pMatrix, mvMatrix, self.program, True)

    # unbind texture
    glBindTexture(GL_TEXTURE_2D, 0)
    glBindFramebuffer(GL_FRAMEBUFFER, 0)
    glBindRenderbuffer(GL_RENDERBUFFER, 0)

    # return texture ID
    return self.texHandle

def renderFrontFace(self, pMatrix, mvMatrix, program):
    """render front-face of ray-cube"""
    # no face culling
    self.renderCube(pMatrix, mvMatrix, program, False)

def renderCube(self, pMatrix, mvMatrix, program, cullFace):
    """render cube using face culling if flag set"""

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

    # set shader program
    glUseProgram(program)

    # set projection matrix
    glUniformMatrix4fv(glGetUniformLocation(program,
 b'uPMatrix'),
                       1, GL_FALSE, pMatrix)

    # set modelview matrix
    glUniformMatrix4fv(glGetUniformLocation(program,
 b'uMVMatrix'),
                       1, GL_FALSE, mvMatrix)

    # enable face culling
```

```
    glDisable(GL_CULL_FACE)
    if cullFace:
        glFrontFace(GL_CCW)
        glCullFace(GL_FRONT)
        glEnable(GL_CULL_FACE)

    # bind VAO
    glBindVertexArray(self.vao)

    # animated slice
    glDrawElements(GL_TRIANGLES, self.nIndices,
 GL_UNSIGNED_SHORT, None)

    # unbind VAO
    glBindVertexArray(0)

    # reset cull face
    if cullFace:
        # disable face culling
        glDisable(GL_CULL_FACE)

def reshape(self, width, height):
    self.width = width
    self.height = height
    self.aspect = width/float(height)
    # re-create FBO
    self.clearFBO()
    self.initFBO()

def initFBO(self):
    # create frame buffer object
    self.fboHandle = glGenFramebuffers(1)
    # create texture
    self.texHandle = glGenTextures(1)
    # create depth buffer
    self.depthHandle = glGenRenderbuffers(1)

    # bind
    glBindFramebuffer(GL_FRAMEBUFFER, self.fboHandle)

    glActiveTexture(GL_TEXTURE0)
    glBindTexture(GL_TEXTURE_2D, self.texHandle)

    # set parameters to draw the image at different sizes
```

```
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR)
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR)
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE)
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE)

    # set up texture
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, self.width,
self.height,
                 0, GL_RGBA, GL_UNSIGNED_BYTE, None)

    # bind texture to FBO
    glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0,
                           GL_TEXTURE_2D, self.texHandle,
0)

    # bind
    glBindRenderbuffer(GL_RENDERBUFFER, self.depthHandle)
    glRenderbufferStorage(GL_RENDERBUFFER,
GL_DEPTH_COMPONENT24,
                          self.width, self.height)

    # bind depth buffer to FBO
    glFramebufferRenderbuffer(GL_FRAMEBUFFER,
GL_DEPTH_ATTACHMENT,
                              GL_RENDERBUFFER,
self.depthHandle)

    # check status
    status = glCheckFramebufferStatus(GL_FRAMEBUFFER)
    if status == GL_FRAMEBUFFER_COMPLETE:
        pass
        # print "fbo %d complete" % self.fboHandle
    elif status == GL_FRAMEBUFFER_UNSUPPORTED:
        print("fbo %d unsupported" % self.fboHandle)
    else:
        print("fbo %d Error" % self.fboHandle)

    glBindTexture(GL_TEXTURE_2D, 0)
    glBindFramebuffer(GL_FRAMEBUFFER, 0)
```

```
        glBindRenderbuffer(GL_RENDERBUFFER, 0)
        return

    def clearFBO(self):
        """clears old FBO"""
        # delete FBO
        if glIsFramebuffer(self.fboHandle):
            glDeleteFramebuffers(1, int(self.fboHandle))

        # delete texture
        if glIsTexture(self.texHandle):
            glDeleteTextures(int(self.texHandle))

    def close(self):
        """call this to free up OpenGL resources"""
        glBindTexture(GL_TEXTURE_2D, 0)
        glBindFramebuffer(GL_FRAMEBUFFER, 0)
        glBindRenderbuffer(GL_RENDERBUFFER, 0)

        # delete FBO
        if glIsFramebuffer(self.fboHandle):
            glDeleteFramebuffers(1, int(self.fboHandle))

        # delete texture
        if glIsTexture(self.texHandle):
            glDeleteTextures(int(self.texHandle))

        # delete render buffer
        """
        if glIsRenderbuffer(self.depthHandle):
            glDeleteRenderbuffers(1, int(self.depthHandle))
            """

        # delete buffers
        """
        glDeleteBuffers(1, self._vertexBuffer)
        glDeleteBuffers(1, &_indexBuffer)
        glDeleteBuffers(1, &_colorBuffer)
        """
```

The Complete Volume Ray Casting Code

Here's the full *raycast.py* code listing.

```
import OpenGL
from OpenGL.GL import *
from OpenGL.GL.shaders import *

import numpy as np
import math, sys

import raycube, glutils, volreader

strVS = """
# version 330 core

layout(location = 1) in vec3 cubePos;
layout(location = 2) in vec3 cubeCol;

uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;

out vec4 vColor;

void main()
{
    // set position
    gl_Position = uPMatrix * uMVMatrix * vec4(cubePos.xyz,
  1.0);

    // set color
    vColor = vec4(cubeCol.rgb, 1.0);
}
"""

strFS = """
# version 330 core

in vec4 vColor;

uniform sampler2D texBackFaces;
uniform sampler3D texVolume;
uniform vec2 uWinDims;
```

```
out vec4 fragColor;

void main()
{
    // start of ray
    vec3 start = vColor.rgb;

    // calculate texture coords at fragment,
    // which is a fraction of window coords
    vec2 texc = gl_FragCoord.xy/uWinDims.xy;

    // get end of ray by looking up back-face color
    vec3 end = texture(texBackFaces, texc).rgb;

    // calculate ray direction
    vec3 dir = end - start;

    // normalized ray direction
    vec3 norm_dir = normalize(dir);

    // the length from front to back is calculated and
    // used to terminate the ray
    float len = length(dir.xyz);

    // ray step size
    float stepSize = 0.01;

    // X-ray projection
    vec4 dst = vec4(0.0);

    // step through the ray
    for(float t = 0.0; t < len; t += stepSize) {

        // set position to endpoint of ray
        vec3 samplePos = start + t*norm_dir;

        // get texture value at position
        float val = texture(texVolume, samplePos).r;
        vec4 src = vec4(val);

        // set opacity
        src.a *= 0.1;
        src.rgb *= src.a;

        // blend with previous value
        dst = (1.0 - dst.a)*src + dst;
```

```
        // exit loop when alpha exceeds threshold
        if(dst.a >= 0.95)
           break;
    }

    // set fragment color
    fragColor =  dst;
}
"""

class Camera:
    """helper class for viewing"""
    def __init__(self):
        self.r = 1.5
        self.theta = 0
        self.center = [0.5, 0.5, 0.5]
        self.eye = [0.5 + self.r, 0.5, 0.5]
        self.up = [0.0, 0.0, 1.0]

    def rotate(self, clockWise):
        """rotate eye by one step"""
        if clockWise:
            self.theta = (self.theta + 5) % 360
        else:
            self.theta = (self.theta - 5) % 360
        # recalculate eye
        self.eye = [0.5 +
  self.r*math.cos(math.radians(self.theta)),
                    0.5 +
  self.r*math.sin(math.radians(self.theta)),
                    0.5]

class RayCastRender:
    """class that does Ray Casting"""

    def __init__(self, width, height, volume):
        """RayCastRender constr"""

        # create RayCube object
        self.raycube = raycube.RayCube(width, height)

        # set dimensions
        self.width = width
```

```
        self.height = height
        self.aspect = width/float(height)

        # create shader
        self.program = glutils.loadShaders(strVS, strFS)
        # texture
        self.texVolume, self.Nx, self.Ny, self.Nz = volume

        # initialize camera
        self.camera = Camera()

    def draw(self):

        # build projection matrix
        pMatrix = glutils.perspective(45.0, self.aspect, 0.1,
100.0)

        # modelview matrix
        mvMatrix = glutils.lookAt(self.camera.eye,
self.camera.center,
                                  self.camera.up)
        # render

        # generate ray-cube back-face texture
        texture = self.raycube.renderBackFace(pMatrix,
mvMatrix)

        # set shader program
        glUseProgram(self.program)

        # set window dimensions
        glUniform2f(glGetUniformLocation(self.program,
b"uWinDims"),
                    float(self.width), float(self.height))

        # texture unit 0, which represents back-faces of cube
        glActiveTexture(GL_TEXTURE0)
        glBindTexture(GL_TEXTURE_2D, texture)
        glUniform1i(glGetUniformLocation(self.program,
b"texBackFaces"), 0)

        # texture unit 1: 3D volume texture
        glActiveTexture(GL_TEXTURE1)
        glBindTexture(GL_TEXTURE_3D, self.texVolume)
```

```
        glUniform1i(glGetUniformLocation(self.program,
  b"texVolume"), 1)

        # draw front-face of cubes
        self.raycube.renderFrontFace(pMatrix, mvMatrix,
  self.program)

        #self.render(pMatrix, mvMatrix)

    def keyPressed(self, key):
        if key == 'l':
            self.camera.rotate(True)
        elif key == 'r':
            self.camera.rotate(False)

    def reshape(self, width, height):
        self.width = width
        self.height = height
        self.aspect = width/float(height)
        self.raycube.reshape(width, height)

    def close(self):
        self.raycube.close()
```

The Complete 2D Slicing Code

Here is the full 2D slicing code listing.

```
import OpenGL
from OpenGL.GL import *
from OpenGL.GL.shaders import *
import numpy, math, sys

import volreader, glutils

strVS = """
# version 330 core

in vec3 aVert;

uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
```

```
uniform float uSliceFrac;
uniform int uSliceMode;

out vec3 texcoord;

void main() {

    // x slice
    if (uSliceMode == 0) {
        texcoord = vec3(uSliceFrac, aVert.x, 1.0-aVert.y);
    }
    // y slice
    else if (uSliceMode == 1) {
        texcoord = vec3(aVert.x, uSliceFrac, 1.0-aVert.y);
    }
    // z slice
    else {
        texcoord = vec3(aVert.x, 1.0-aVert.y, uSliceFrac);
    }

    // calculate transformed vertex
    gl_Position = uPMatrix * uMVMatrix * vec4(aVert, 1.0);
}
"""

strFS = """
# version 330 core

in vec3 texcoord;

uniform sampler3D tex;

out vec4 fragColor;

void main() {
    // look up color in texture
    vec4 col = texture(tex, texcoord);
    fragColor = col.rrra;
}

"""

class SliceRender:
    # slice modes
    XSLICE, YSLICE, ZSLICE = 0, 1, 2
```

```
def __init__(self, width, height, volume):
    """SliceRender constructor"""
    self.width = width
    self.height = height
    self.aspect = width/float(height)

    # slice mode
    self.mode = SliceRender.ZSLICE

    # create shader
    self.program = glutils.loadShaders(strVS, strFS)

    glUseProgram(self.program)

    self.pMatrixUniform =
glGetUniformLocation(self.program, b'uPMatrix')
    self.mvMatrixUniform =
glGetUniformLocation(self.program, b"uMVMatrix")

    # attributes
    self.vertIndex = glGetAttribLocation(self.program,
b"aVert")

    # set up vertex array object (VAO)
    self.vao = glGenVertexArrays(1)
    glBindVertexArray(self.vao)

    # define quad vertices
    vertexData = numpy.array([0.0, 1.0, 0.0,
                              0.0, 0.0, 0.0,
                              1.0, 1.0, 0.0,
                              1.0, 0.0, 0.0],
numpy.float32)
    # vertex buffer
    self.vertexBuffer = glGenBuffers(1)
    glBindBuffer(GL_ARRAY_BUFFER, self.vertexBuffer)
    glBufferData(GL_ARRAY_BUFFER, 4*len(vertexData),
vertexData,
                 GL_STATIC_DRAW)
    # enable arrays
    glEnableVertexAttribArray(self.vertIndex)
    # set buffers
    glBindBuffer(GL_ARRAY_BUFFER, self.vertexBuffer)
```

```
    glVertexAttribPointer(self.vertIndex, 3, GL_FLOAT,
GL_FALSE, 0, None)

    # unbind VAO
    glBindVertexArray(0)

    # load texture
    self.texture, self.Nx, self.Ny, self.Nz = volume

    # current slice index
    self.currSliceIndex = int(self.Nz/2);
    self.currSliceMax = self.Nz;

def reshape(self, width, height):
    self.width = width
    self.height = height
    self.aspect = width/float(height)

def draw(self):
    # clear buffers
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    # build projection matrix
    pMatrix = glutils.ortho(-0.6, 0.6, -0.6, 0.6, 0.1,
100.0)
    # modelview matrix
    mvMatrix = numpy.array([1.0, 0.0, 0.0, 0.0,
                            0.0, 1.0, 0.0, 0.0,
                            0.0, 0.0, 1.0, 0.0,
                            -0.5, -0.5, -1.0, 1.0],
numpy.float32)
    # use shader
    glUseProgram(self.program)

    # set projection matrix
    glUniformMatrix4fv(self.pMatrixUniform, 1, GL_FALSE,
pMatrix)

    # set modelview matrix
    glUniformMatrix4fv(self.mvMatrixUniform, 1, GL_FALSE,
mvMatrix)

    # set current slice fraction
    glUniform1f(glGetUniformLocation(self.program,
b"uSliceFrac"),
```

```
                float(self.currSliceIndex)/float(self.currS
liceMax))
    # set current slice mode
    glUniform1i(glGetUniformLocation(self.program,
b"uSliceMode"),
                self.mode)

    # enable texture
    glActiveTexture(GL_TEXTURE0)
    glBindTexture(GL_TEXTURE_3D, self.texture)
    glUniform1i(glGetUniformLocation(self.program, b"tex"),
0)

    # bind VAO
    glBindVertexArray(self.vao)
    # draw
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4)
    # unbind VAO
    glBindVertexArray(0)

def keyPressed(self, key):
    """keypress handler"""
    if key == 'x':
        self.mode = SliceRender.XSLICE
        # reset slice index
        self.currSliceIndex = int(self.Nx/2)
        self.currSliceMax = self.Nx
    elif key == 'y':
        self.mode = SliceRender.YSLICE
        # reset slice index
        self.currSliceIndex = int(self.Ny/2)
        self.currSliceMax = self.Ny
    elif key == 'z':
        self.mode = SliceRender.ZSLICE
        # reset slice index
        self.currSliceIndex = int(self.Nz/2)
        self.currSliceMax = self.Nz
    elif key == 'l':
        self.currSliceIndex = (self.currSliceIndex + 1) %
self.currSliceMax
    elif key == 'r':
```

```
            self.currSliceIndex = (self.currSliceIndex - 1) %
    self.currSliceMax

    def close(self):
        pass
```

The Complete Main File Code

Here is the full code listing for the main file.

```
import sys, argparse, os
from slicerender import *
from raycast import *
import glfw

class RenderWin:
    """GLFW Rendering window class"""
    def __init__(self, imageDir):

        # save current working directory
        cwd = os.getcwd()

        # initialize glfw; this changes cwd
        glfw.glfwInit()

        # restore cwd
        os.chdir(cwd)

        # version hints
        glfw.glfwWindowHint(glfw.GLFW_CONTEXT_VERSION_MAJOR, 3)
        glfw.glfwWindowHint(glfw.GLFW_CONTEXT_VERSION_MINOR, 3)
        glfw.glfwWindowHint(glfw.GLFW_OPENGL_FORWARD_COMPAT,
    GL_TRUE)
        glfw.glfwWindowHint(glfw.GLFW_OPENGL_PROFILE,
                            glfw.GLFW_OPENGL_CORE_PROFILE)

        # make a window
        self.width, self.height = 512, 512
        self.aspect = self.width/float(self.height)
        self.win = glfw.glfwCreateWindow(self.width,
    self.height, b"volrender")
```

```
    # make context current
    glfw.glfwMakeContextCurrent(self.win)

    # initialize GL
    glViewport(0, 0, self.width, self.height)
    glEnable(GL_DEPTH_TEST)
    glClearColor(0.0, 0.0, 0.0, 0.0)

    # set window callbacks
    glfw.glfwSetMouseButtonCallback(self.win,
 self.onMouseButton)
    glfw.glfwSetKeyCallback(self.win, self.onKeyboard)
    glfw.glfwSetWindowSizeCallback(self.win, self.onSize)

    # load volume data
    self.volume =  volreader.loadVolume(imageDir)
    # create renderer
    self.renderer = RayCastRender(self.width, self.height,
 self.volume)

    # exit flag
    self.exitNow = False

def onMouseButton(self, win, button, action, mods):
    # print 'mouse button: ', win, button, action, mods
    pass

def onKeyboard(self, win, key, scancode, action, mods):
    # print 'keyboard: ', win, key, scancode, action, mods
    # ESC to quit
    if key is glfw.GLFW_KEY_ESCAPE:
        self.renderer.close()
        self.exitNow = True
    else:
        if action is glfw.GLFW_PRESS or action is
 glfw.GLFW_REPEAT:
            if key == glfw.GLFW_KEY_V:
                # toggle render mode
                if isinstance(self.renderer,
 RayCastRender):
                    self.renderer = SliceRender(self.width,
 self.height,
```

```
                                                        self.volume
)
                else:
                    self.renderer =
RayCastRender(self.width, self.height,
                                                          self.volu
me)
                # call reshape on renderer
                self.renderer.reshape(self.width,
self.height)
            else:
                # send keypress to renderer
                keyDict = {glfw.GLFW_KEY_X: 'x',
glfw.GLFW_KEY_Y: 'y',
                           glfw.GLFW_KEY_Z: 'z',
glfw.GLFW_KEY_LEFT: 'l',
                           glfw.GLFW_KEY_RIGHT: 'r'}
                try:
                    self.renderer.keyPressed(keyDict[key])
                except:
                    pass

def onSize(self, win, width, height):
    # print 'onsize: ', win, width, height
    self.width = width
    self.height = height
    self.aspect = width/float(height)
    glViewport(0, 0, self.width, self.height)
    self.renderer.reshape(width, height)

def run(self):
    # start loop
    while not glfw.glfwWindowShouldClose(self.win) and not
self.exitNow:
        # render
        self.renderer.draw()
        # swap buffers
        glfw.glfwSwapBuffers(self.win)
        # wait for events
        glfw.glfwWaitEvents()
    # end
```

```
        glfw.glfwTerminate()

# main() function
def main():
    print('starting volrender...')
    # create parser
    parser = argparse.ArgumentParser(description="Volume
     Rendering...")
    # add expected arguments
    parser.add_argument('--dir', dest='imageDir',
     required=True)
    # parse args
    args = parser.parse_args()

    # create render window
    rwin = RenderWin(args.imageDir)
    rwin.run()

# call main
if __name__ == '__main__':
    main()
```

---

1 J. Kruger and R. Westermann, “Acceleration Techniques for GPU-based Volume Rendering,” IEEE Visualization, 2003.

2 *https://graphics.stanford.edu/data/voldata/*

PART V

HARDWARE HACKING

Those parts of the system that you can hit with a hammer (not advised) are called hardware; those program instructions that you can only curse at are called software.

—Anonymous

12
KARPLUS-STRONG ON A RASPBERRY PI PICO



In Chapter 4, you learned how to make plucked string sounds using the Karplus-Strong algorithm. You saved the generated sounds as WAV files and played notes from a pentatonic musical scale on your computer. In this chapter, you'll learn how to shrink that project to fit on a tiny piece of hardware: the Raspberry Pi Pico.

The Pico (see Figure 12-1) is built using an RP2040 microcontroller chip, which has just 264KB of random access memory (RAM). Compare that to the tens of gigabytes of RAM on the typical personal computer! The Pico also has 2MB of flash memory on a separate chip, in contrast to a normal computer's hundreds of gigabytes of hard disk space. Despite these limitations, however, the Pico is still extremely capable. It can perform many useful services, while also being much cheaper and less power-hungry than a regular computer. Your watch, your air conditioning unit, your clothes dryer, your car, your phone—tiny microcontrollers like the RP2040 are everywhere!

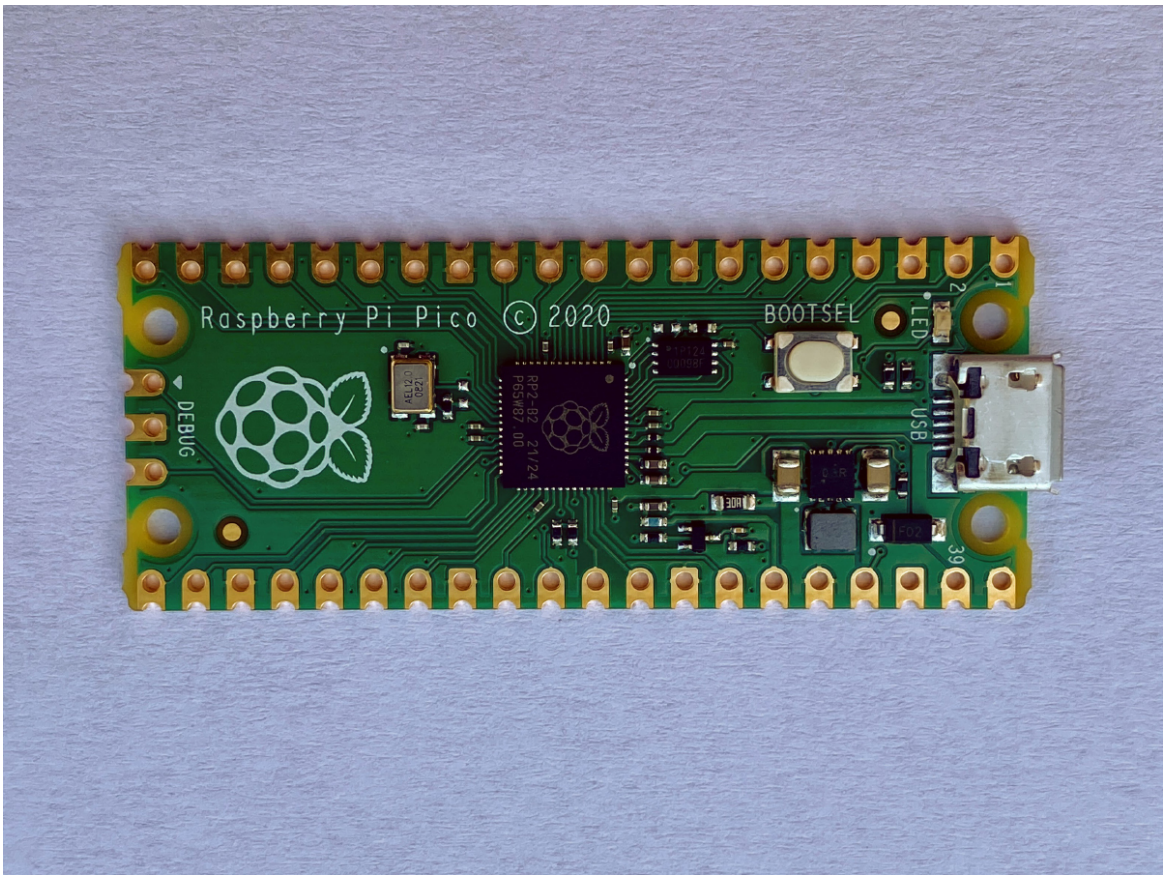


Figure 12-1: *The Raspberry Pi Pico*

The goal for this project is to use the Raspberry Pi Pico to create a musical instrument with five buttons. Pressing each button will play a note from a pentatonic scale, generated with the Karplus-Strong algorithm. Some of the concepts you'll learn from this project are:

- Programming a microcontroller using MicroPython, an implementation of Python optimized to run on devices like the Pico
- Building a simple audio circuit on a breadboard using the Pico
- Using the I2S digital audio protocol and an I2S amplifier to send audio data to a speaker

- Implementing the Karplus-Strong algorithm from Chapter 4 on a resource-constrained microcontroller

How It Works

We discussed the Karplus-Strong algorithm in detail in Chapter 4, so we won't revisit it here. Instead, we'll focus on what makes this version of the project different. Your program from Chapter 4 was designed to run on a laptop or desktop computer. Thanks to the computer's ample RAM and hard disk resources, it had no problems creating WAV files using the Karplus-Strong algorithm and playing audio through speakers with `pyaudio`. The challenge now is to fit the project code onto a resource-constrained Raspberry Pi Pico. This will require the following modifications:

- Using a smaller audio sampling rate to reduce memory requirements
- Using a simple binary file to store raw generated samples rather than a WAV file
- Using the I2S protocol to send out the audio data to an external audio amplifier
- Using memory management techniques to avoid copying the same data repeatedly

We'll discuss the specifics of these modifications as they arise.

Input and Output

To make the project interactive, you'll want the Pico to generate sounds in response to user input. You'll need to wire five push buttons to the Pico for this purpose, since the

Pico doesn’t have a keyboard or mouse. (You’ll use a sixth push button to run the program.) We also need to figure out how to produce the sound output, since unlike a personal computer, the Pico board doesn’t have any built-in speakers. Figure 12-2 shows a block diagram of the project.

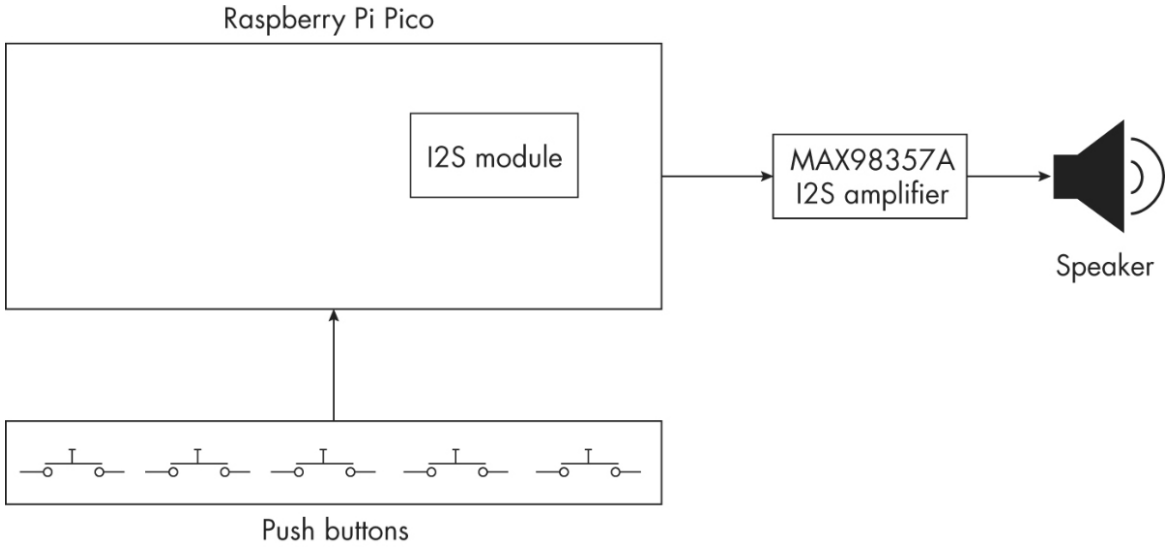


Figure 12-2: *A block diagram of the project*

When you press a button, the MicroPython code running on the Pico will generate a plucked string sound using the Karplus-Strong algorithm. The digital sound samples produced by the algorithm will be sent to a separate MAX98357A amplifier board, which decodes the digital data into an analog audio signal. The MAX98357A also amplifies the analog signal, which allows you to connect its output to an external 8-ohm speaker so you can hear the audio. Figure 12-3 shows the Adafruit MAX98357A board.

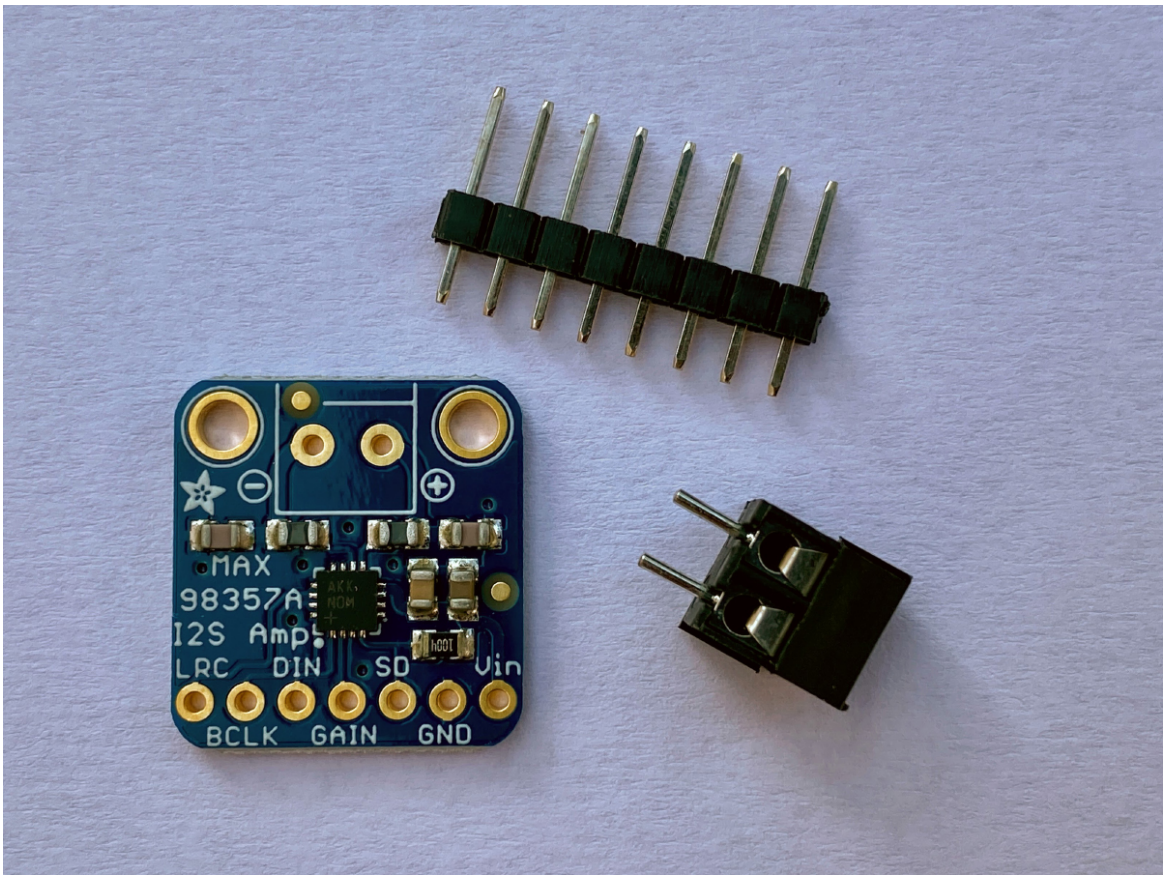


Figure 12-3: *An Adafruit MAX98357A I2S amplifier board*

The Pi Pico needs to send data to the amplifier board in a certain format for it to successfully be interpreted as an audio signal. Enter the I2S protocol.

The I2S Protocol

The *Inter-IC Sound (I2S) protocol* is a standard for sending digital audio data between devices. It's a simple, convenient way to get quality audio output from a microcontroller. The protocol transmits audio using three digital signals, which are shown in Figure 12-4.

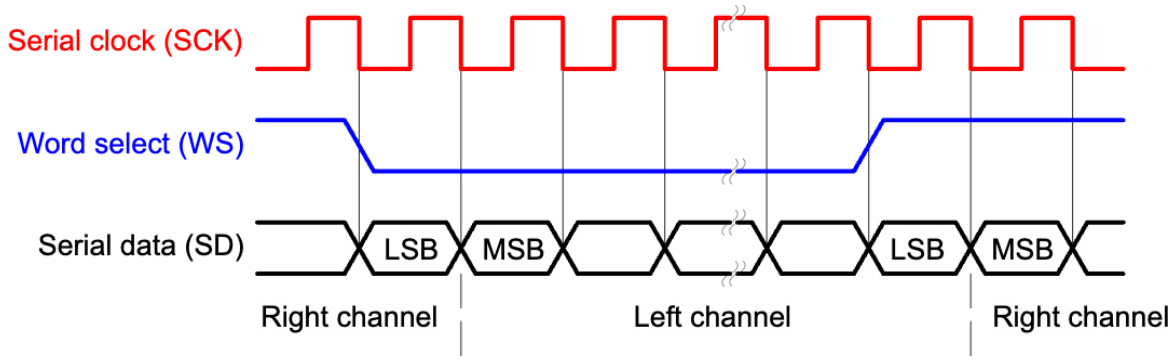


Figure 12-4: *The I2S protocol*

The first signal, SCK, is the *clock*, a signal that alternates between high and low at a fixed speed. This sets the rate of data transmission. Next, WS is the *word select* signal. It steadily alternates between high and low to indicate which audio channel, left or right, is being sent at any given moment. Finally, SD is the *serial data* signal, which carries the actual audio information, in the form of N-bit binary values representing the amplitude of the sound.

To understand how this works, let's consider an example. Say you want to send stereo audio at a sampling rate of 16,000 Hz and you want the amplitude of each sound sample to be a 16-bit value. The frequency of WS should be the same as the sampling rate, since that's the rate at which you're sending each amplitude value. This way, the WS signal will alternate between high and low 16,000 times per second; when it's high, SD will send the amplitude value of one audio channel, and when it's low, SD will send the amplitude value of the other audio channel. Since each amplitude value for each channel is made up of 16 bits, SD has to transmit at a rate that's 16 × 2 = 32 times faster than the sampling rate. The clock controls the rate of transmission, so SCK's frequency must be 16,000 Hz × 32 = 512,000 Hz.

For this project, the Pico will be the I2S transmitter, so it will generate the SCK, WS, and SD signals. MicroPython actually has a fully implemented `I2S` module for the Pico, so

much of the work generating the signals will be done for you, behind the scenes. As you've already seen, the Pico will send the signals to a MAX98357A board, which is specifically designed to receive audio data through the I2S protocol. Then the board converts the I2S data into an analog audio signal that can be played through a speaker.

Requirements

You'll program the project for the Raspberry Pi Pico using MicroPython. You'll need the following hardware:

- One Raspberry Pi Pico board based on the RP2040 chip
- One Adafruit MAX98357A I2S breakout board
- One 8-ohm speaker
- Six push buttons
- Five 10 kΩ resistors
- One breadboard
- An assortment of hookup wires
- One Micro USB cable for uploading code to the Pico

Hardware Setup

You'll assemble the hardware on a breadboard. Figure 12-5 shows the hookup.

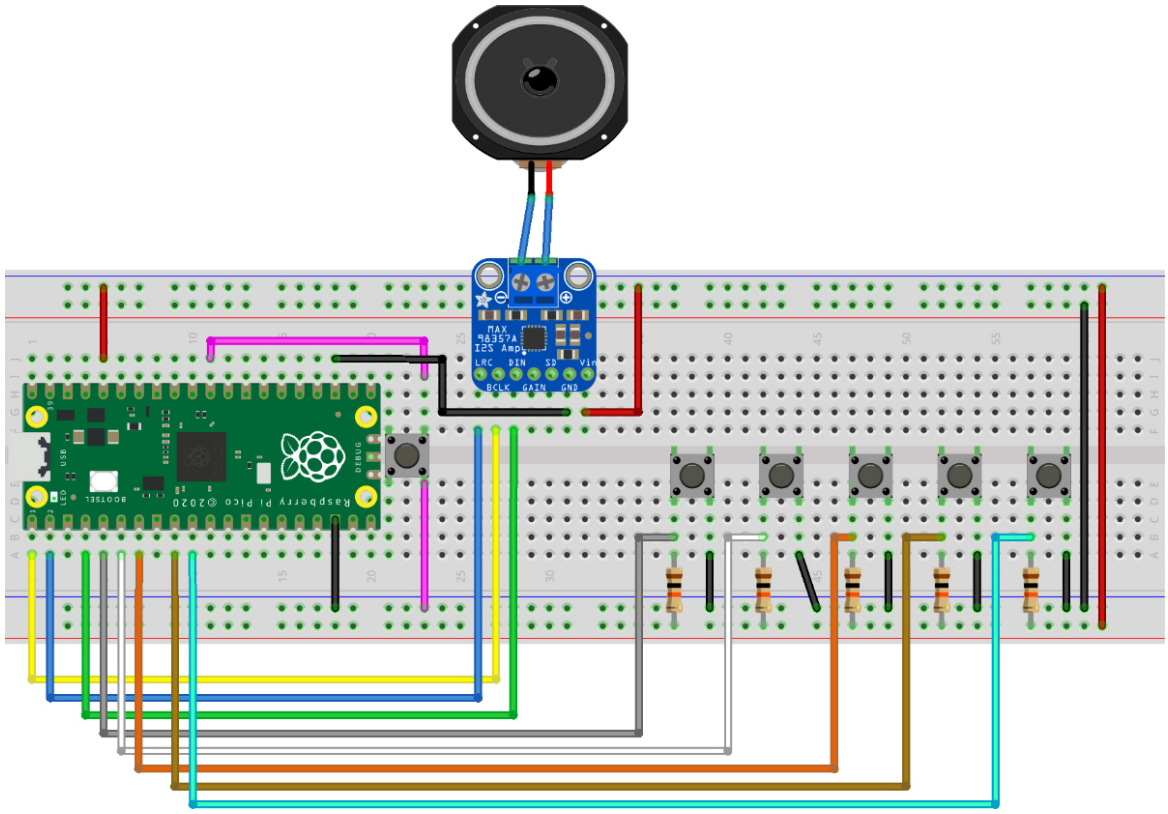


Figure 12-5: *The hardware hookup*

Figure 12-6 shows a pin diagram of the Pico from the official datasheet, which is a handy reference for your hookup.

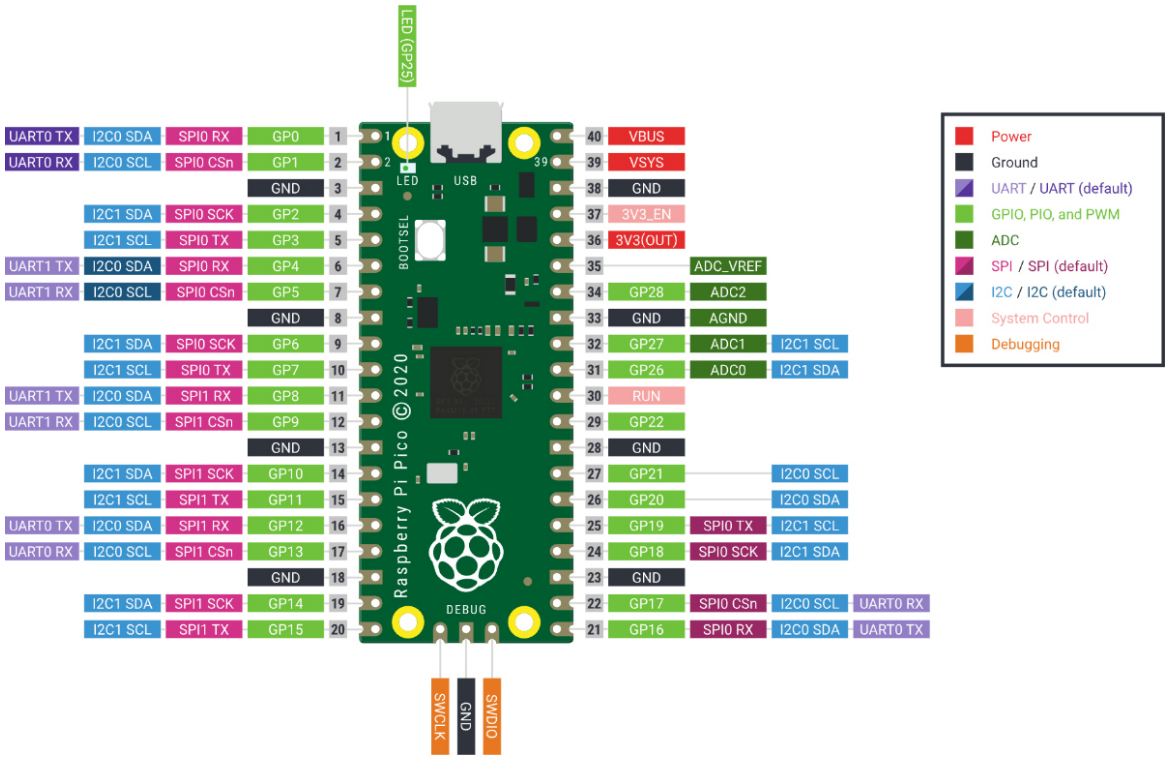


Figure 12-6: *A pin diagram from the Raspberry Pi Pico datasheet*

Table 12-1 summarizes the electrical connections you need to implement on the breadboard. Figure 12-5 shows these connections.

Table 12-1: Electrical Connections

Pico pin	**Connection**
GP3	Push button 1 (other pin to VDD via 10 kΩ resistor)
GP4	Push button 2 (other pin to VDD via 10 kΩ resistor)
GP5	Push button 3 (other pin to VDD via 10 kΩ resistor)

Pico pin	Connection
GP6	Push button 4 (other pin to VDD via 10 kΩ resistor)
GP7	Push button 5 (other pin to VDD via 10 kΩ resistor)
RUN	Push button 6 (other pin to GND)
GP0	MAX98357A BCLK
GP1	MAX98357A LRC
GP2	MAX98357A DIN
GND	MAX98357A GND
3V3(OUT)	MAX98357A Vin

Once you've hooked up the hardware, your project should look like Figure 12-7.

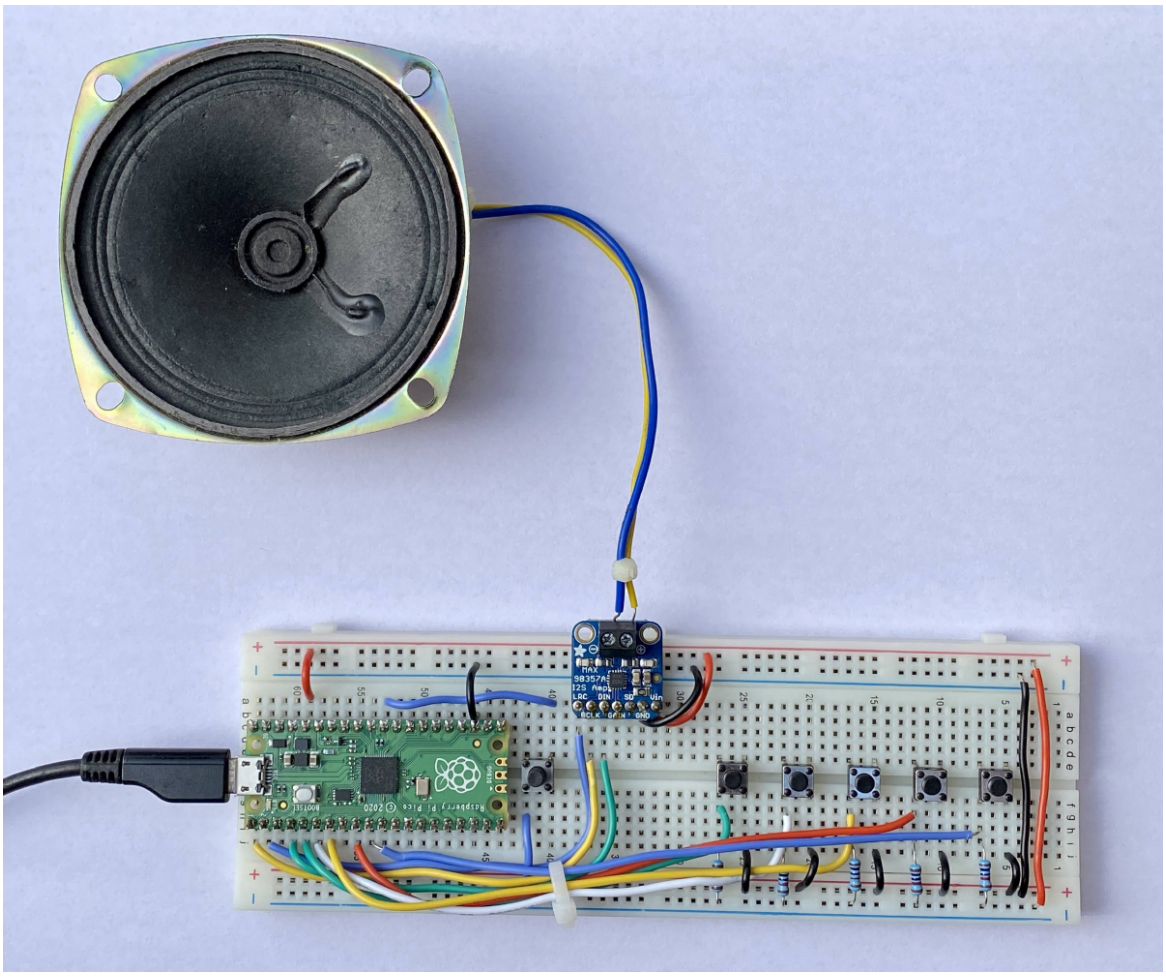


Figure 12-7: *The fully built hardware*

Before you start using your Pico, however, you need to set up MicroPython.

MicroPython Setup

Setting up your Raspberry Pi Pico with MicroPython is quite straightforward. Follow these steps:

1. Visit *https://micropython.org*, go to the Download page, and find the Raspberry Pi Pico.
2. Download the UF2 binary file (version 1.18 or later) containing the MicroPython implementation for the

Pico.

3. Press the white BOOTSEL button on the Pico, and while holding this button down, use your Micro USB cable to connect the Pico to your computer. Then release the button.
4. You should see a folder called *RPI-RP2* pop up on your computer. Drag and drop the UF2 file into this folder.

Once the copying is done and the Pico reboots, you're all set to code the Pico using MicroPython!

The Code

The code consists of some initial setup, followed by functions for generating and playing the five notes. Then everything comes together in the program's `main()` function. To see the full program, skip ahead to "The Complete Code" on page 275. The code is also available on GitHub at *https://github.com/mkvenkit/pp2e/blob/main/karplus_pico/karplus_pico.py*.

Setting Up

The code begins with some basic setup. First, import the required MicroPython modules:

```
import time
import array
import random
import os

from machine import I2S
from machine import Pin
```

You import the `time` module for its “sleep” functionality to create timed pauses during the execution of the code. The `array` module will let you create arrays for sending the sound data via I2S. An array is a more efficient version of a Python list, since it requires all members to be of the same data type. You’ll use the `random` module to fill the initial buffer with random values (the first step in the Karplus-Strong algorithm), and you’ll use the `os` module to check if a note has already been saved in the filesystem. Finally, the `I2S` module will let you send sound data, and the `Pin` module lets you set up the pin outputs of the Pico.

You complete the setup by declaring some useful information:

```
# notes of a minor pentatonic scale
# piano C4-E(b)-F-G-B(b)-C5
pmNotes = {'C4': 262, 'Eb': 311, 'F': 349, 'G':391, 'Bb':466}

# button to note mapping
btnNotes = {0: ('C4', 262), 1: ('Eb', 311), 2: ('F', 349), 3:
('G', 391),
            4: ('Bb', 466)}

# sample rate
SR = 16000
```

Here you define a dictionary `pmNotes` that maps the name of a note to its integer frequency value ❶. You’ll use the names of the notes to save files containing the sound data, and you’ll use the frequency values to generate the sounds using the Karplus-Strong algorithm. You also define a dictionary `btnNotes` that maps the ID of each push button (represented as the integers 0 through 4) to a tuple that has the corresponding note name and frequency value ❷. This dictionary controls which note is played when the user presses each button.

Finally, you define the sampling rate as 16,000 Hz ❸. This is the number of sound amplitude values per second that you'll be sending out via I2S. Notice that this is much lower than the sampling rate of 44,100 Hz used in Chapter 4. This is because of the limited memory on the Pico compared to a standard computer.

Generating the Notes

You generate the five notes of the pentatonic scale with the help of two functions: `generate_note()` and `create_notes()`. The `generate_note()` function uses the Karplus-Strong algorithm to calculate the amplitude values for a single note, while `create_notes()` coordinates generating all five notes and saving their sample data to the Pico's filesystem. Let's consider the `generate_note()` function first. (You implemented a similar function in Chapter 4, so this might be a good time to review the original implementation.)

```
# generate note of given frequency
def generate_note(freq):
    nSamples = SR
    N = int(SR/freq)
    # initialize ring buffer
  ❶ buf = [2*random.random() - 1 for i in range(N)]
    # init sample buffer
  ❷ samples = array.array('h', [0]*nSamples)
    for i in range(nSamples):
      ❸ samples[i] = int(buf[0] * (2 ** 15 - 1))
      ❹ avg = 0.4975*(buf[0] + buf[1])
        buf.append(avg)
        buf.pop(0)
  ❺ return samples
```

The function starts by setting `nSamples`, the length of the samples buffer that will hold the final audio data, to `SR`, the sampling rate. Since `SR` is the number of samples per second, this implies that you'll be creating a one-second audio clip. Then you compute `N`, the number of samples in the Karplus-Strong ring buffer, by dividing the sampling rate by the frequency of the note being generated.

Next, you initialize your buffers. First you create the ring buffer with random initial values ❶. The `random.random()` method returns values in the range [0.0, 1.0], so `2*random.random() - 1` scales the values to the range [−1.0, 1.0]. Remember, you need both positive and negative amplitudes for the algorithm. Notice that you're implementing the ring buffer as a regular Python list, instead of as a `deque` object like you used in Chapter 4. MicroPython's `deque` implementation has restrictions and doesn't give you what you need for a ring buffer. Instead, you'll just use the regular `append()` and `pop()` list methods to add and remove elements from the buffer. You also create the samples buffer as an `array` object of length `nSamples` filled with zeros ❷. The `'h'` argument specifies that each element in this array is a *signed short*, a 16-bit value that can be positive or negative. Since each sample will be represented by a 16-bit value, this is exactly what you need.

Next, you iterate over the items in the `samples` array and build up the audio clip using the Karplus-Strong algorithm. You take the first sample value in the ring buffer and scale it from range [−1.0, 1.0] to range [−32767, 32767] ❸. (The range of a 16-bit signed short is [−32767, 32768]. You scale the amplitude values to be as high as possible, which will give you the highest possible volume of sound output.) Then you calculate an attenuated average of the first two samples in the ring buffer ❹. (Here, `0.4975` is the same as `0.995*0.5` from the original implementation.) You use `append()` to add the new amplitude value to the end of the ring

buffer, while using `pop()` to remove the first element, thus maintaining the buffer's fixed size. At the end of the loop, the samples buffer is full, so you return it for further processing ❺.

NOTE *Using `append()` and `pop()` to update the ring buffer works, but it isn't an efficient method of computation. We'll look more at optimization in "Experiments!" on page 273.*

Now let's consider the `create_notes()` function:

```
def create_notes():
    "create pentatonic notes and save to files in flash"
  ❶ files = os.listdir()
  ❷ for (k, v) in pmNotes.items():
        # set note filename
      ❸ file_name = k + ".bin"
        # check if file already exists
      ❹ if file_name in files:
            print("Found " + file_name + ". Skipping...")
            continue
        # generate note
        print("Generating note " + k + "...")
      ❺ samples = generate_note(v)
        # write to file
        print("Writing " + file_name + "...")
      ❻ file_samples = open(file_name, "wb")
      ❼ file_samples.write(samples)
      ❽ file_samples.close()
```

You don't want to have to run the Karplus-Strong algorithm every time the user presses a button to play a note, as that would be too slow. Instead, this function creates the notes the first time the code is run and stores them in the Pico's filesystem as *.bin* files. Then, as soon as

the user presses a button, you'll be able to read the appropriate file and output the sound data via I2S.

You begin by using the `os` module to list the files on the Pico ❶. (There's no "hard disk" on the Pico. Rather, a flash chip on the Pico board is used to store data, and MicroPython provides a way to access this data like a normal filesystem.) Then you iterate through the items in the `pmNotes` dictionary, which maps note names to frequencies ❷. For each note, you generate a filename based on its name in the dictionary (for example, *C4.bin*) ❸. If a file with that name exists in the directory ❹, you've generated that note already, so you can skip to the next one. Otherwise, you generate the sound samples for that note using the `generate_note()` function ❺. Then you create a binary file with the appropriate name ❻ and write the samples to it ❼. Finally, you clean up by closing the file ❽.

The first time you run the code, `create_notes()` will run the `generate_note()` function to create a file for each note using the Karplus-Strong algorithm. This will create the files *C4.bin*, *Eb.bin*, *F.bin*, *G.bin*, and *Bb.bin* on the Pico. On subsequent runs, the function will find these files still in place, so it won't need to create them again.

Playing a Note

The `play_note()` function plays one of the notes from the pentatonic scale by outputting the samples using the I2S protocol. Here's the definition:

```
def play_note(note, audio_out):
    "read note from file and send via I2S"
  ❶ fname = note[0] + ".bin"
    print("opening " + fname)
    # open file
    try:
```

```
        print("opening {}...".format(fname))
      ❷ file_samples = open(fname, "rb")
    except:
        print("Error opening file: {}!".format(fname))
        return

    # allocate sample array
  ❸ samples = bytearray(1000)
    # memoryview used to reduce heap allocation
  ❹ samples_mv = memoryview(samples)

    # read samples and send to I2S
    try:
      ❺ while True:
          ❻ num_read = file_samples.readinto(samples_mv)
            # end of file?
          ❼ if num_read == 0:
                break
            else:
                # send samples via I2S
              ❽ num_written =
audio_out.write(samples_mv[:num_read])
  ❾ except (Exception) as e:
        print("Exception: {}".format(e))

    # close file
  ❿ file_samples.close()
```

The function has two arguments: `note`, a tuple in the form `('C4', 262)` conveying the note name and frequency, and `audio_out`, an instance of the `I2S` module used for sound output. You first create the appropriate *.bin* filename based on the name of the note to be played ❶. Then you open the file ❷. You expect the file to exist at this point, so if the open fails, you just return from the function.

The rest of the function outputs the audio data via I2S, working in batches of 1,000 samples. To mediate the data transfer, you create a MicroPython `bytearray` of 1,000 samples ❸ and a `memoryview` of the samples ❹. This is a

MicroPython optimization technique to prevent the whole array from being copied when a slice of the array is passed into other functions such as `file_samples.readinto()` and `audio_out.write()`.

NOTE *A* slice *of an array represents a range of values within that array. For example, `a[100:200]` is a slice representing array values `a[100]` through `a[199]`.*

Next, you start a `while` loop to read samples from the file ❺. In the loop, you read a batch of samples from the file into the `memoryview` object using the `readinto()` method ❻, which returns the number of samples read (`num_read`). You output the samples from the `memoryview` object via I2S using the `audio_out.write()` method ❽. The `[:num_read]` slice notation ensures you write out the same number of samples you read in. You handle any exceptions at ❾. You're done outputting data when you get to the point where zero samples are read into the `memoryview` object ❼, in which case you can break out of the `while` loop and close the *.bin* file ❿.

Writing the main() Function

Now let's look at the `main()` function, which brings all the code together:

```
def main():
    # set up LED
  ❶ led = Pin(25, Pin.OUT)
    # turn on LED
    led.toggle()
    # create notes and save in flash
  ❷ create_notes()

    # create I2S object
  ❸ audio_out = I2S(
        0,                  # I2S ID
```

```
        sck=Pin(0),         # SCK Pin
        ws=Pin(1),          # WS Pin
        sd=Pin(2),          # SD Pin
        mode=I2S.TX,        # I2S transmitter
        bits=16,            # 16 bits per sample
        format=I2S.MONO,    # Mono - single channel
        rate=SR,            # sample rate
        ibuf=2000,          # I2S buffer length
    )

    # set up btns
❹ btns = [Pin(3, Pin.IN, Pin.PULL_UP),
             Pin(4, Pin.IN, Pin.PULL_UP),
             Pin(5, Pin.IN, Pin.PULL_UP),
             Pin(6, Pin.IN, Pin.PULL_UP),
             Pin(7, Pin.IN, Pin.PULL_UP)]

    # "ready" note
❺ play_note(('C4', 262), audio_out)
    print("Piano ready!")

    # turn off LED
❻ led.toggle()

    while True:
        for i in range(5):
            if btns[i].value() == 0:
              ❼ play_note(btnNotes[i], audio_out)
                break
      ❽ time.sleep(0.2)
```

The function starts by setting up the Pico's onboard LED ❶. It's toggled ON at the start to indicate the Pico is busy initializing. Next, you call the `create_notes()` function ❷. As we discussed, this function will create the *.bin* files for the notes only if they don't already exist in the filesystem. To manage the audio output, you instantiate the `I2S` module as `audio_out` ❸. The module requires a number of input parameters. The first parameter is the I2S ID, which is `0` for the Raspberry Pi Pico. Next come the pin numbers

corresponding to the clock (SCK), word select (WS), and data (SD) signals. We discussed these signals in “I2S Protocol” on page 262. You then set the I2S mode to `TX`, indicating this is an I2S transmitter. Next, you set `bits` to `16`, indicating the number of bits per sample, and `format` to `MONO`, since there’s only one audio output channel. You set the sampling rate to `SR`, and lastly, you set the value for the internal I2S buffer `ibuf` to `2000`.

NOTE *A smooth audio experience requires an uninterrupted stream of data output. MicroPython uses a special hardware module in the Pico called Direct Memory Access (DMA) for this. DMA can transfer data from the memory to the I2S output without involving the CPU directly. The CPU just needs to keep an internal buffer (`ibuf` in the code) filled with data and is free to do other things while the DMA is doing its job. The size of the internal buffer is typically set to at least twice the size of the audio output so the DMA doesn’t run out of data to transfer, which would result in distorted audio. In this case, you’ll be transferring 1,000 bytes to I2S at a time, so you set `ibuf` to twice that.*

Next, you need to set up the buttons so that notes will play when the buttons are pressed. For this, you create a list of `Pin` objects called `btns` ❹. For each button in the list, you specify the pin number, the data direction of the pin (`Pin.IN`, or input, in this case), and whether the pin has a pull-up resistor. In this case, all the push button pins have a 10 kΩ resistor pull-up on them. This means that by default the pins’ voltages are “pulled up” to VDD, or 3.3 V, and when the buttons are pushed, the voltage drops to GND, or 0 V. You’ll use this fact to detect button presses.

Once the setup is done, you play a C4 note using the `play_note()` function to indicate the Pico is ready to accept button presses ❺, and you also toggle the onboard LED to

OFF ❻. Then you start a `while` loop to monitor for button presses. Within this loop, you use a `for` loop to check whether any of the five buttons have a value of `0`, indicating the button is pressed. If so, you look up the note corresponding to that button in the `btnNotes` dictionary and play it using `play_note()` ❼. Once the note is done playing, you break out of the `for` loop and wait for 0.2 seconds ❽ before continuing with the outer `while` loop.

Running the Pico Code

Now you're ready to test your project! To run the code on the Pico, it's useful to install two pieces of software. The first is Thonny, an open source, easy-to-use Python integrated development environment (IDE), which you can download from *https://thonny.org*. Thonny makes it easy to copy your project code to the Raspberry Pi Pico and manage files on the Pico. A typical development cycle is as follows:

1. Connect your Pico to your computer via USB.
2. Open Thonny. Click the Python version number in the bottom-right of the window and change the interpreter to **MicroPython (Raspberry Pi Pico)**.
3. Copy your code to Thonny and click the red **Stop/Restart** button to stop the code from running on the Pico. This will show the Python interpreter at the bottom of the IDE.
4. Edit your code in Thonny.
5. When you're ready to save the file, select **File▸Save As**, and you'll be prompted to save it on the Raspberry Pi Pico. The next dialog will also list the files on the Pico. Save your code as *main.py*. You

can also use this dialog to right-click and delete existing files on the Pico.

6. After saving the file, press the extra push button you hooked up to the RUN pin on the Pico, and your code will start running.
7. Anytime you want to edit the code, click the **Stop/Restart** button in the IDE and Thonny will drop you to the Python interpreter on the Pico.

The other useful piece of software for working with the Pico is CoolTerm, which you can download from *http://freeware.the-meiers.org*. CoolTerm lets you monitor the Pico's serial output. All print statements from your program will end up here. To use CoolTerm, ensure that you aren't "stopped" in Thonny. The Pico code should be in the running state, since the Pico can't connect to both Thonny and CoolTerm at the same time.

Once you have the code running, press the push buttons one by one and you'll hear the notes of a nice pentatonic scale coming out of your speaker. Figure 12-8 shows the serial output for a typical session.

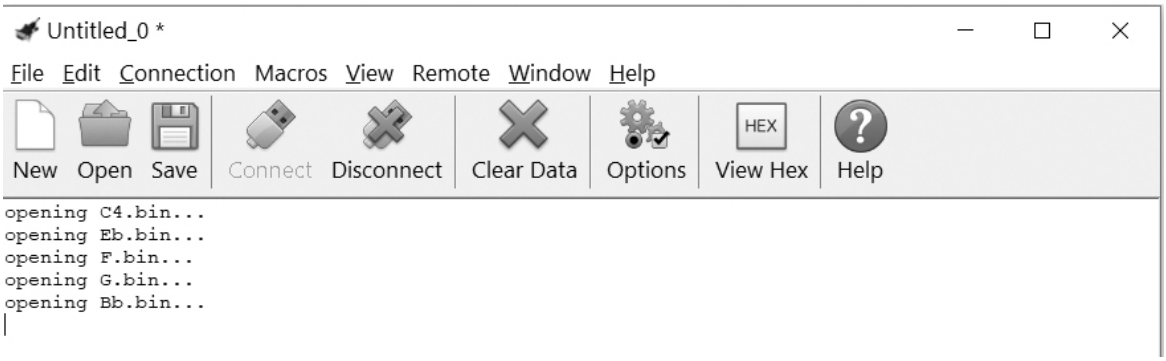


Figure 12-8: *A sample Raspberry Pi Pico output in CoolTerm*

See what melodies you can compose and play using your digital instrument's five push buttons!

Summary

In this chapter, you adapted your Karplus-Strong algorithm implementation from Chapter 4 to run on a tiny microcontroller and built a digital musical instrument using a Raspberry Pi Pico. You learned how to run Python (in the form of MicroPython) on the Pico, as well as how to transmit audio data using the I2S protocol. You also learned about the limitations of adapting code from a personal computer to a resource-constrained device like the Pico.

Experiments!

1. The MAX98357A I2S board lets you increase the volume (gain) of the output. Look at the datasheet of this board and try to boost the sound coming from the speaker.
2. The current implementation of `generate_note()` isn't very fast. It didn't matter that much for this project, because you generate the notes only once. Still, can you make the method faster? Here are a few strategies to try:

    a. Instead of using `append()` and `pop()` operations on the `buf` list, turn the list into a circular buffer by keeping tracking of the current position in the list and incrementing it using the modulo operation `%N`.

    b. Use integer operations instead of floating point. You'll have to think about how the initial random values will be generated and scaled.

    The Language Reference page in the MicroPython documentation (*https://docs.micropython.org*) has an article on maximizing the speed of your code.

The documentation also suggests how you can test your results. First, define a function to measure timing:

```
def timed_function(f, *args, **kwargs):
    myname = str(f).split(' ')[1]
    def new_func(*args, **kwargs):
        t = time.monotonic()
        result = f(*args, **kwargs)
        delta = time.monotonic() - t
        print('Function {} Time = {:f}
s'.format(myname, delta))
        return result
    return new_func
```

Then use `timed_function()` as a *decorator* for the function you want to time:

```
# generate note of given frequency
@timed_function
def generateNote(freq):
    nSamples = SR
    N = int(SR/freq)
    --snip--
```

When you call `generateNote()` in your main code, you'll see something like this in the serial output:

```
Function generateNote Time = 1019.711ms
```

3. When you press a hardware push button, it doesn't just go from ON to OFF, or the reverse. The spring-loaded contacts inside the button bounce between ON and OFF multiple times in a fraction of a second, triggering multiple software events for one physical

press of the button. Think about how this could affect your project, and then read up on *debouncing*, a class of techniques for mitigating the problem. What steps can you take to debounce your buttons?

4. When you press a button, a new note isn't played until the current note is done playing. How can you abruptly stop playing the current note when a new button is pressed and switch to playing the new note immediately?

The Complete Code

Here's the full code listing for this project.

```
"""
karplus_pico.py

Uses the Karplus-Strong algorithm to generate musical notes in
  a
pentatonic scale. Runs on a Raspberry Pi Pico. (MicroPython)

Author: Mahesh Venkitachalam
"""

import time
import array
import random
import os

from machine import I2S
from machine import Pin

# notes of a minor pentatonic scale
# piano C4-E(b)-F-G-B(b)-C5
pmNotes = {'C4': 262, 'Eb': 311, 'F': 349, 'G':391, 'Bb':466}

# button to note mapping
```

```
btnNotes = {0: ('C4', 262), 1: ('Eb', 311), 2: ('F', 349), 3:
  ('G', 391),
            4: ('Bb', 466)}

# sample rate
SR = 16000

def timed_function(f, *args, **kwargs):
    myname = str(f).split(' ')[1]
    def new_func(*args, **kwargs):
        t = time.ticks_us()
        result = f(*args, **kwargs)
        delta = time.ticks_diff(time.ticks_us(), t)
        print('Function {} Time = {:6.3f}ms'.format(myname,
  delta/1000))
        return result
    return new_func

# generate note of given frequency
# (Uncomment line below when you need to time the function.)
# @timed_function
def generate_note(freq):
    nSamples = SR
    N = int(SR/freq)
    # initialize ring buffer
    buf = [2*random.random() - 1 for i in range(N)]

    # init sample buffer
    samples = array.array('h', [0]*nSamples)
    for i in range(nSamples):
        samples[i] = int(buf[0] * (2 ** 15 - 1))
        avg = 0.4975*(buf[0] + buf[1])
        buf.append(avg)
        buf.pop(0)

    return samples

# generate note of given frequency - improved method
def generate_note2(freq):
    nSamples = SR
    sampleRate = SR
    N = int(sampleRate/freq)
```

```
    # initialize ring buffer
    buf = [2*random.random() - 1 for i in range(N)]

    # init sample buffer
    samples = array.array('h', [0]*nSamples)
    start = 0
    for i in range(nSamples):
        samples[i] = int(buf[start] * (2**15 - 1))
        avg = 0.4975*(buf[start] + buf[(start + 1) % N])
        buf[(start + N) % N] = avg
        start = (start + 1) % N

    return samples

def play_note(note, audio_out):
    "read note from file and send via I2S"
    fname = note[0] + ".bin"

    # open file
    try:
        print("opening {}...".format(fname))
        file_samples = open(fname, "rb")
    except:
        print("Error opening file: {}!".format(fname))
        return

    # allocate sample array
    samples = bytearray(1000)
    # memoryview used to reduce heap allocation
    samples_mv = memoryview(samples)

    # read samples and send to I2S
    try:
        while True:
            num_read = file_samples.readinto(samples_mv)
            # end of file?
            if num_read == 0:
                break
            else:
                # send samples via I2S
                num_written =
  audio_out.write(samples_mv[:num_read])
```

```
    except (Exception) as e:
        print("Exception: {}".format(e))

    # close file
    file_samples.close()

def create_notes():
    "create pentatonic notes and save to files in flash"
    files = os.listdir()
    for (k, v) in pmNotes.items():
        # set note filename
        file_name = k + ".bin"
        # check if file already exists
        if file_name in files:
            print("Found " + file_name + ". Skipping...")
            continue
        # generate note
        print("Generating note " + k + "...")
        samples = generate_note(v)
        # write to file
        print("Writing " + file_name + "...")
        file_samples = open(file_name, "wb")
        file_samples.write(samples)
        file_samples.close()

def main():

    # set up LED
    led = Pin(25, Pin.OUT)
    # turn on LED
    led.toggle()

    # create notes and save in flash
    create_notes()

    # create I2S object
    audio_out = I2S(
        0,                  # I2S ID
        sck=Pin(0),         # SCK Pin
        ws=Pin(1),          # WS Pin
        sd=Pin(2),          # SD Pin
        mode=I2S.TX,        # I2S transmitter
```

```
        bits=16,            # 16 bits per sample
        format=I2S.MONO,    # Mono - single channel
        rate=SR,            # sample rate
        ibuf=2000,          # I2S buffer length
    )

    # set up btns
    btns = [Pin(3, Pin.IN, Pin.PULL_UP),
            Pin(4, Pin.IN, Pin.PULL_UP),
            Pin(5, Pin.IN, Pin.PULL_UP),
            Pin(6, Pin.IN, Pin.PULL_UP),
            Pin(7, Pin.IN, Pin.PULL_UP)]

    # "ready" note
    play_note(('C4', 262), audio_out)
    print("Piano ready!")

    # turn off LED
    led.toggle()

    while True:
        for i in range(5):
            if btns[i].value() == 0:
                play_note(btnNotes[i], audio_out)
                break
        time.sleep(0.2)

# call main
if __name__ == '__main__':
    main()
```

13
LASER AUDIO DISPLAY WITH A RASPBERRY PI



In Chapter 12, you used a Pico, a tiny microcontroller, to generate musical tones. In this chapter, you'll use a much more powerful embedded system, the Raspberry Pi, to produce interesting laser patterns based on audio signals.

The previous chapter's Pico had an RP2040 microcontroller with dual ARM Cortex-M0 processors running at speeds of up to 133 MHz, with 264KB of random access memory (RAM) and 2MB of nonvolatile storage on an external flash chip. The Raspberry Pi 3B+, in comparison, has a much more capable ARM Cortex-A53 processor operating at 1.4 GHz, with 1GB of RAM and storage of several gigabytes, depending on the SD card you use. While this still pales in comparison to a standard desktop or laptop computer, the Pi is nonetheless capable of running a Linux-based operating system and full-fledged Python, unlike the Pico.

In this chapter, you'll use Python on the Raspberry Pi to read an audio file in the WAV format, perform computations based on the real-time audio data, and use that data to adjust the speed and direction of rotation of two motors in a laser display rig. You'll attach mirrors to the motors to

reflect the beam from an inexpensive laser module, producing Spirograph-like patterns that change in response to the audio. You'll also simultaneously stream the audio to a speaker so you can hear the WAV file playing as you watch your laser light show.

This project will push your knowledge of Python further as you learn to use the Raspberry Pi. Here are some of the topics we'll cover:

- Generating interesting patterns with a laser and two rotating mirrors
- Getting frequency information from a signal using the fast Fourier transform (FFT)
- Computing FFTs using `numpy`
- Reading audio from WAV files
- Outputting audio data using `pyaudio`
- Driving motors with a Raspberry Pi
- Toggling a laser module on/off with a metal-oxide-semiconductor field-effect transistor (MOSFET)

How It Works

You'll use a Raspberry Pi to work with the audio data and control the hardware. Figure 13-1 shows a block diagram of what you'll create in this project.

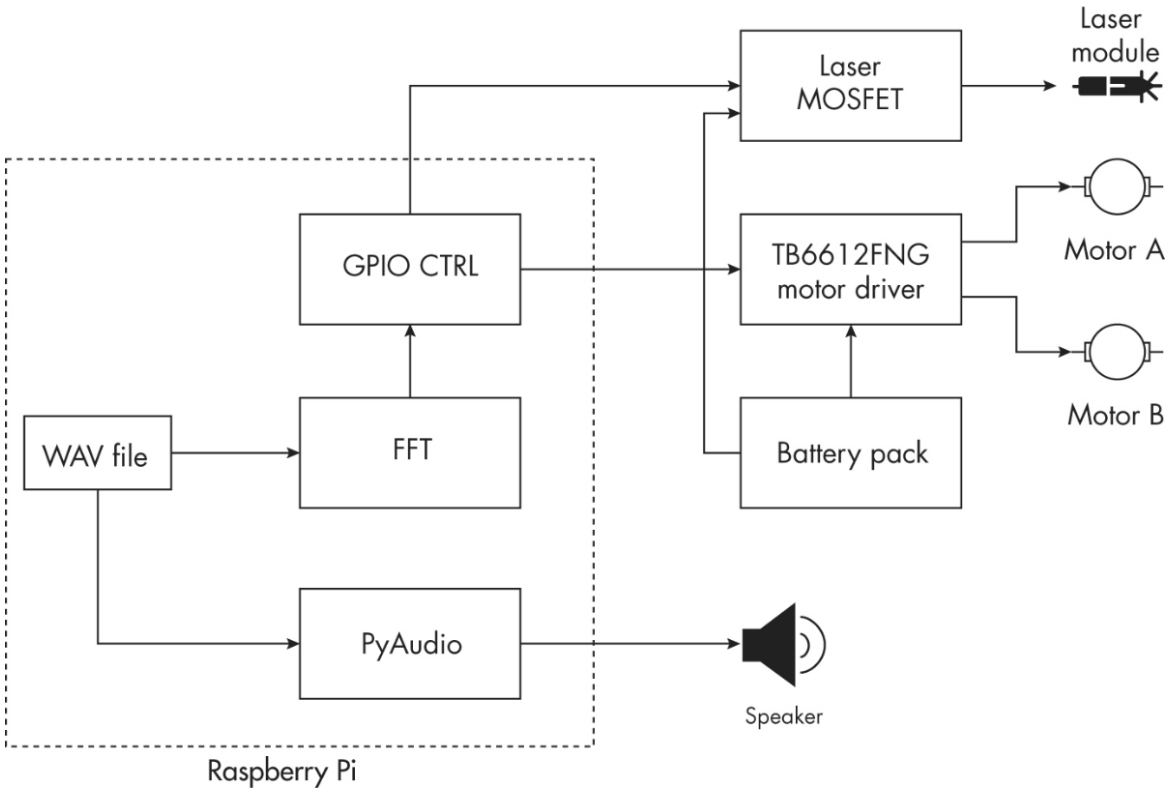


Figure 13-1: *A block diagram of the laser audio project*

The Raspberry Pi will use a WAV file in two ways. It will play the file through an attached speaker by way of `pyaudio`, while also analyzing the audio data in real time using a mathematical technique called a *fast Fourier transform (FFT)*. The Pi will use data from the FFT to drive the motors and laser via its general-purpose input/output (GPIO) pins, but to protect the Pi from damage, you won't hook it up directly to these external components. Instead, you'll connect it indirectly through a motor driver board and a MOSFET. Before you begin, let's consider in more detail how some of these aspects of the project will work.

Generating Patterns with a Laser

To generate the laser patterns in this project, you'll use a laser module and two mirrors attached to the shafts of two

small DC motors, as shown in Figure 13-2. Think of the laser as an intense beam of light that remains focused on a tiny point, even when projected over a large distance. This focus is possible because the beam is organized so that its waves travel in one direction only and are in phase with each other. If you shine the laser at the surface of a flat mirror (mirror A in Figure 13-2), the reflection projected will remain a fixed point, even as the motor spins. Because the plane of reflection of the laser is perpendicular to the spinning axis of the motor, it’s as if the mirror isn’t rotating at all.

Now, say the mirror is attached at an angle relative to the motor shaft, as shown on the right of Figure 13-2 (mirror B). As the shaft rotates, the projected point will trace an ellipse, and if the motor is spinning fast enough, the viewer will perceive the moving dot as a continuous shape.

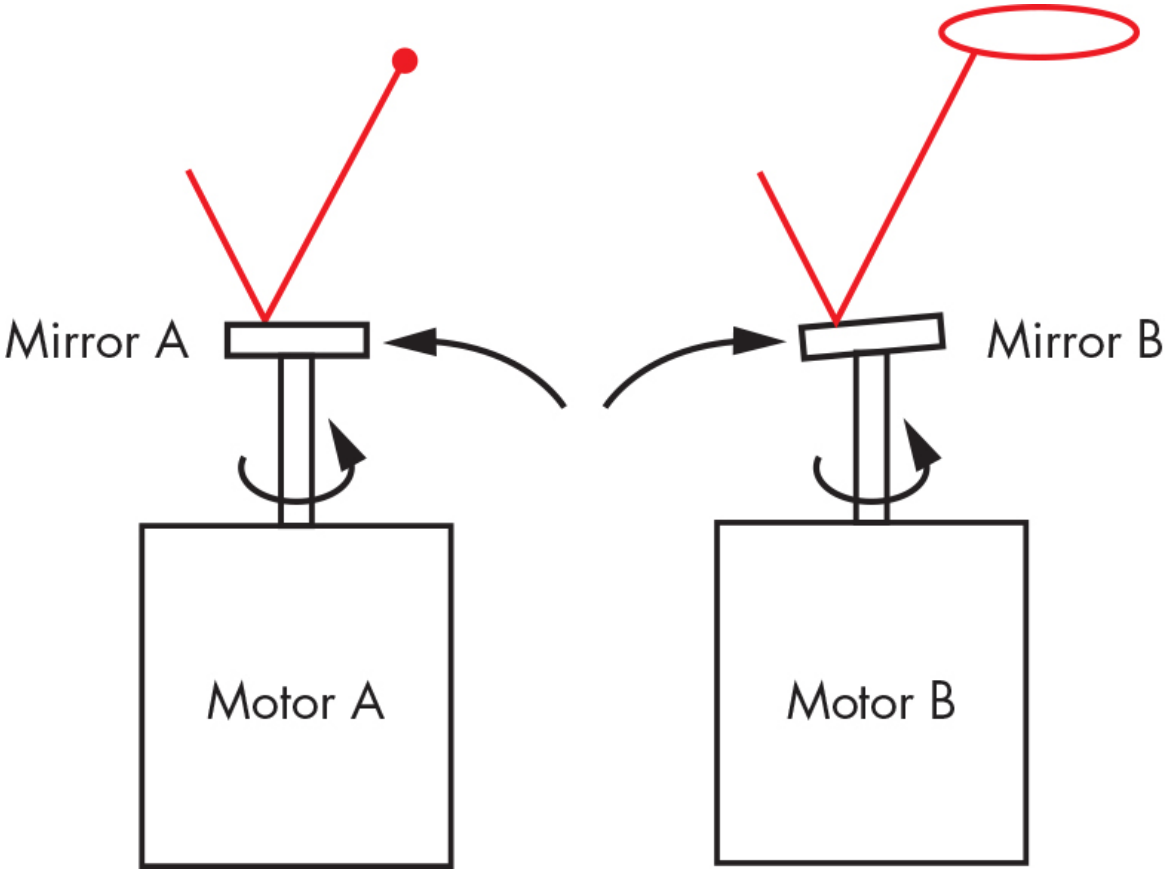


Figure 13-2: *The flat mirror (mirror A) reflects a single dot. The reflection off the slanted mirror (mirror B) creates a circle as the motor spins.*

What if both mirrors are slanted and you arrange them so that the point reflected off mirror A is projected onto mirror B? Now when motors A and B spin, the pattern created by the reflected point will be a combination of the two rotational movements of motors A and B, producing interesting patterns, as shown in Figure 13-3.

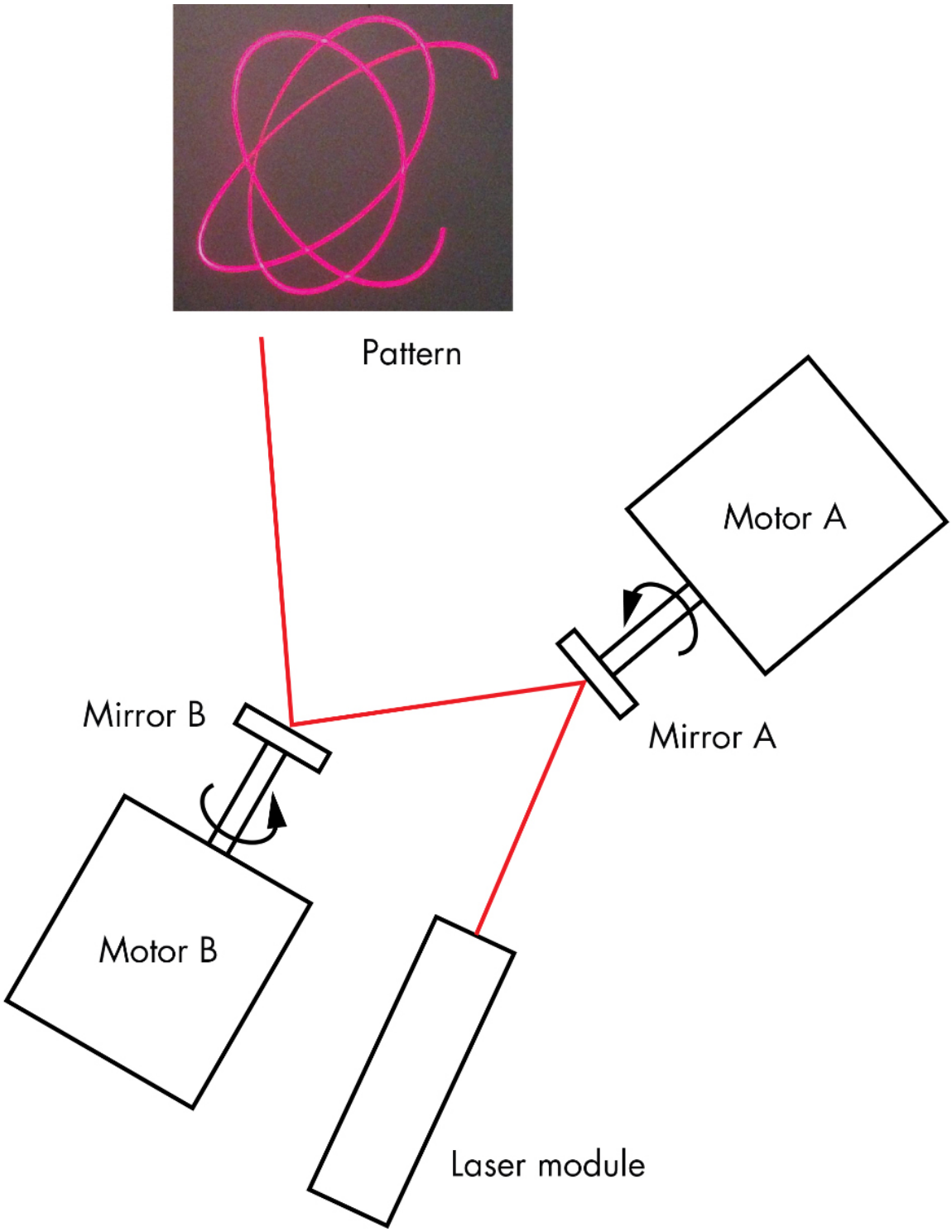


Figure 13-3: *Reflecting laser light off two rotating, slanted mirrors produces interesting, complex patterns.*

The exact patterns produced will depend on the speed and direction of rotation of the two motors, but they will be

similar to the hypotrochoids produced by the Spirograph you explored in Chapter 2.

Motor Control

You'll use the Raspberry Pi to control the speed and direction of your motors through a technique called *pulse width modulation (PWM)*. This is a way to power a device (such as a motor) by sending digital pulses that switch on and off quickly, such that the device "sees" a continuous voltage. The signal sent to the device has a fixed frequency, but the fraction of time that the digital pulse is on, called the *duty cycle*, can vary. The duty cycle is expressed as a percentage. To illustrate, Figure 13-4 shows three signals with the same frequency but different duty cycles—25 percent, 50 percent, and 75 percent.

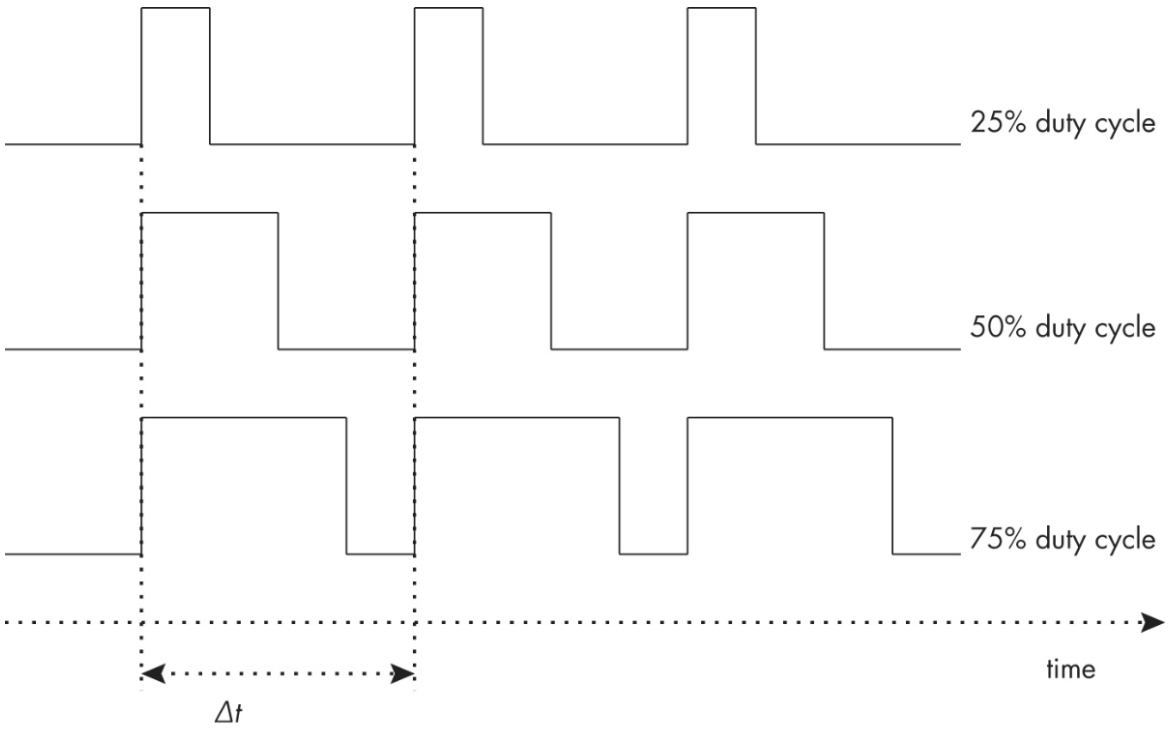


Figure 13-4: *PWM signals with different duty cycles*

The higher the duty cycle percentage, the longer the pulse is on for each cycle of the signal. The motor receiving the signal will perceive those longer pulses as a higher continuous voltage. By manipulating the duty cycle, you can thus provide varying power levels to the motors in this project, which will result in variations in motor speed and changes in the laser pattern.

NOTE *PWM has many applications beyond motor control. It could also be used, for example, to control the brightness of dimmable LEDs.*

Motors operate at a relatively high voltage, but the Raspberry Pi can handle only so much current before it’s damaged. You’ll be using a TB6612FNG motor driver breakout board, similar to the ones shown in Figure 13-5, to act as an intermediary between the Pi and the motors, keeping the Pi safe. Many variants of this board are available, and you can choose any one of them, as long as you’re careful to wire it up correctly.

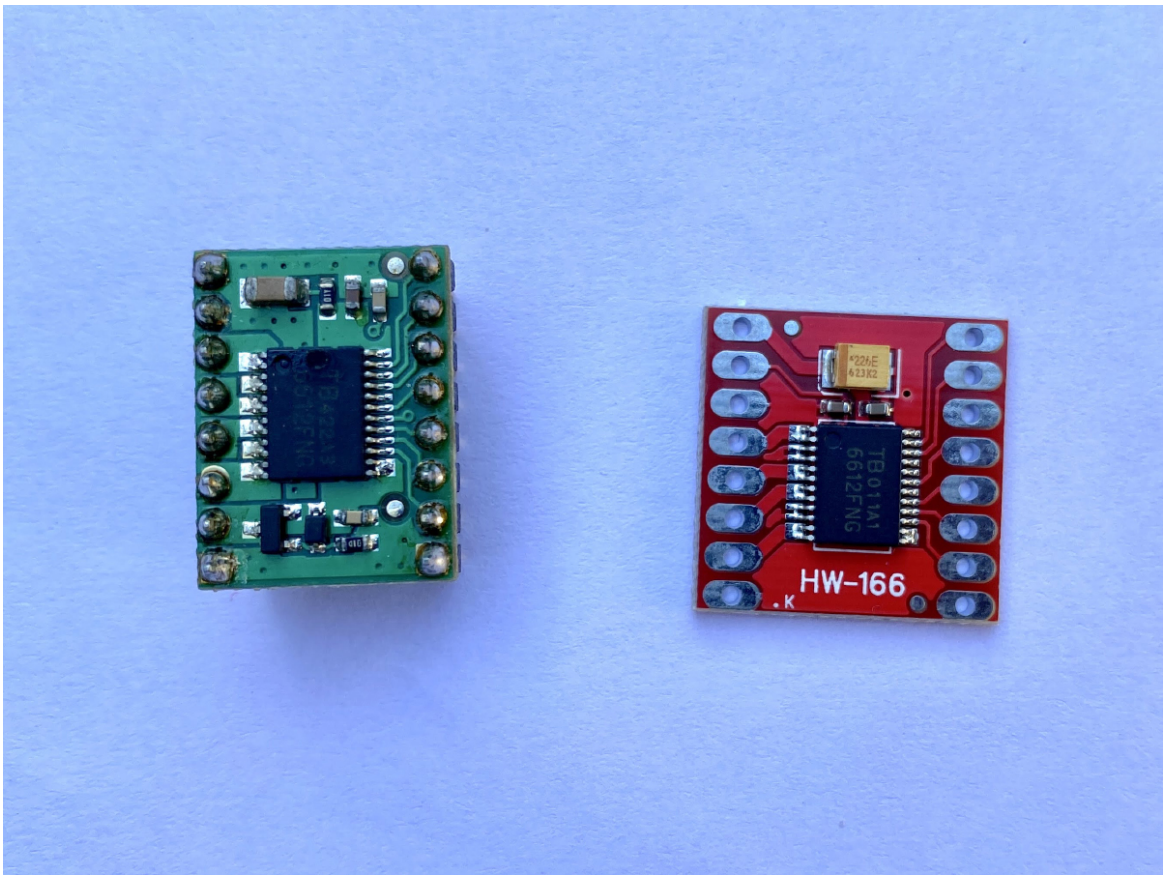


Figure 13-5: *TB6612FNG motor driver breakout printed circuit boards (PCBs)*

The bottom of the breakout board should have pin information. It's also a good idea to look at the TB6612FNG chip datasheet, which you can download from the internet. The *A* and *B* in the pin names denote the two motors. The IN pins control the direction of the motors, the 01 and 02 pins supply power to the motors, and the PWM pins use pulse width modulation to control the motor speeds. By writing to these pins, you can control both the direction and speed of rotation for each motor, which is exactly what you need for this project. We won't go into the details of how this board works, but if you're curious, you can start by reading up on an *H-bridge*, a common circuit design that uses MOSFETs to control motors.

NOTE *You could replace this breakout part with any motor control circuit you're familiar with, as long as you modify the code appropriately.*

Laser Module

For the laser, you'll use an inexpensive laser module breakout PCB similar to the one shown in Figure 13-6.

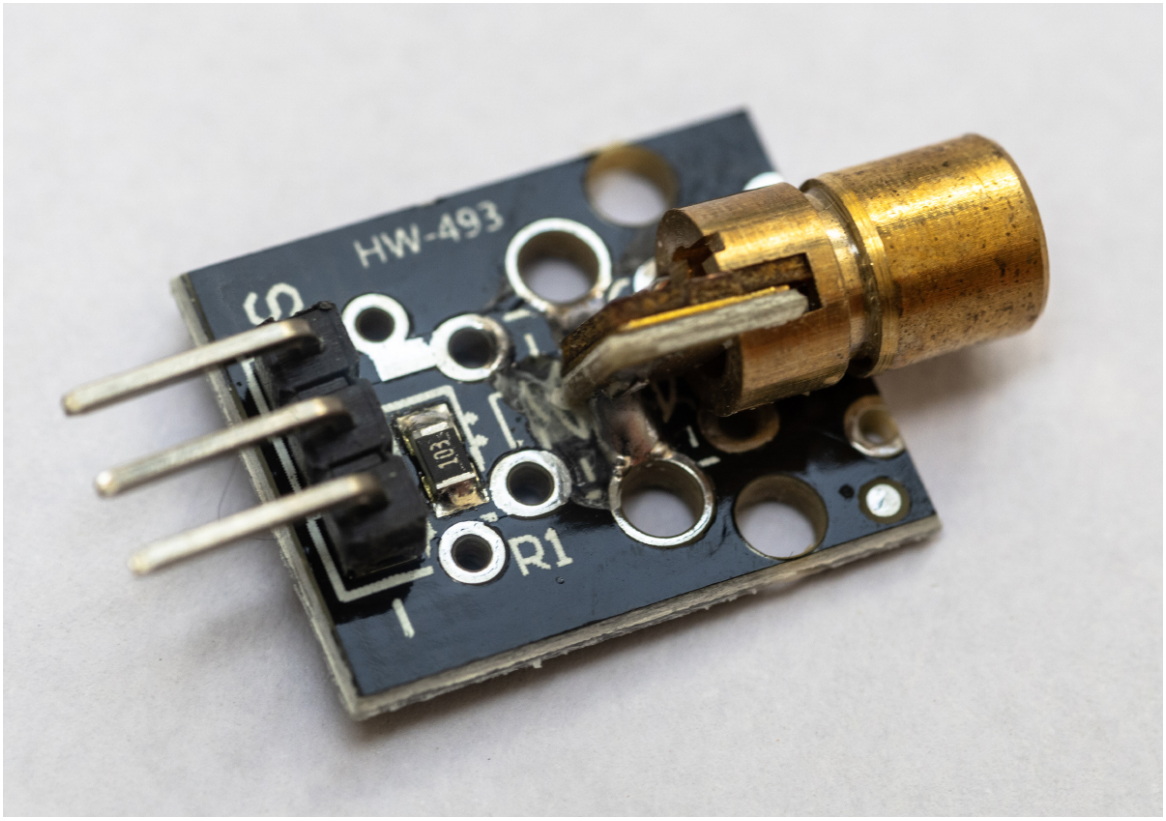


Figure 13-6: *A laser module*

Different variants of laser modules are available. You want one with a 650 nanometer (nm) red laser that operates at 5 volts (V). (The 650 nm refers to the laser's wavelength.) Be sure you understand the polarity and connections of this board before you use it for the project. Test it separately using a 5 V power supply.

MOSFET

To turn the laser module on and off using the Raspberry Pi, you'll use an *N-channel MOSFET*, which you can think of as an electrically controlled switch. You can use almost any N-channel MOSFET for the project, but the BS170 is cheap and readily available. Figure 13-7 shows the pin numbering for the MOSFET, as well as how to connect it to the laser module and the Raspberry Pi.

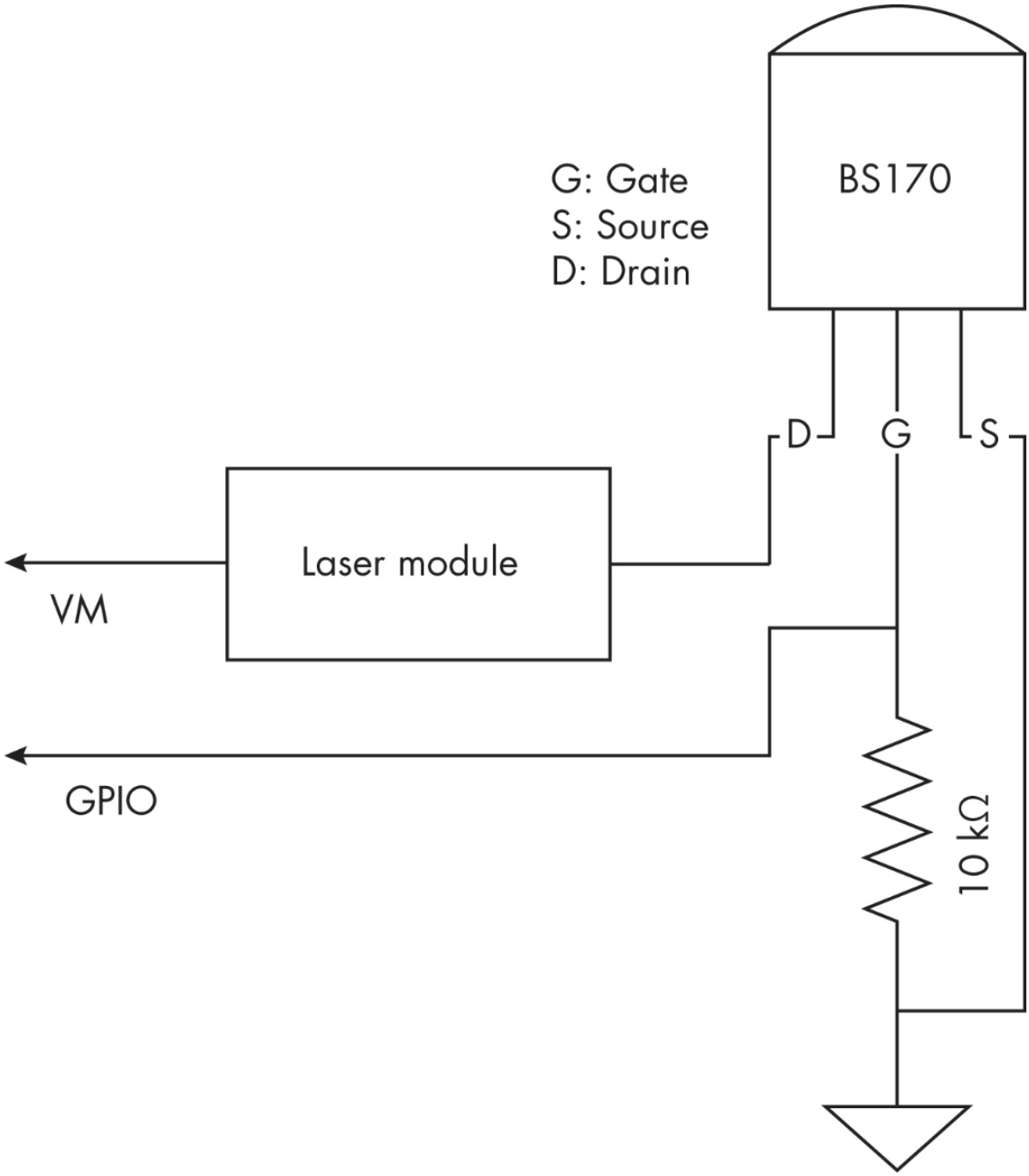


Figure 13-7: *BS170 MOSFET connections*

The 10 kΩ resistor “pulls” the gate pin of the MOSFET to the ground, so it isn’t triggered when the Raspberry Pi GPIO pin is in a floating state (for example, after a GPIO cleanup). When you send a HIGH to the GPIO, the MOSFET switch turns on, effectively connecting the laser module to VM and GND and powering it on.

Why do you need a MOSFET? Can't you just connect the laser module directly to a GPIO pin of the Raspberry Pi? That's not a great idea, because the MOSFET can take a lot more current than your Raspberry Pi. Using a MOSFET isolates your Pi from a situation that causes a current spike on the load. Better to burn your cheap MOSFET than your comparatively expensive Raspberry Pi! In general, the MOSFET trick is a good thing to remember whenever you want to control an external device with your Raspberry Pi.

Analyzing Audio with the Fast Fourier Transform

Because the ultimate goal in this project is to control motor speeds based on audio input, you need to be able to analyze the audio in real time. Recall from Chapter 4 that tones from an acoustic instrument are a mix of several frequencies, or overtones. In fact, any sound can be decomposed into its constituent frequencies using a *Fourier transform*. When the Fourier transform is applied to digital signals, the result is called the *discrete Fourier transform (DFT)* because digital signals are composed of many discrete samples. In this project, you'll use Python to implement a *fast Fourier transform (FFT)* algorithm to compute the DFT. (Throughout this chapter, I'll use *FFT* to refer to both the algorithm and the result.)

Figure 13-8 shows a simple example of an FFT. The top frame of the figure shows the waveform of a signal that combines just two sine waves. This plot is in the *time domain* because it's showing how the signal's amplitude varies over time. The bottom frame of the figure shows the FFT corresponding to that signal. The FFT is in the *frequency domain*; it's showing what frequencies are present in the signal at a given moment in time.

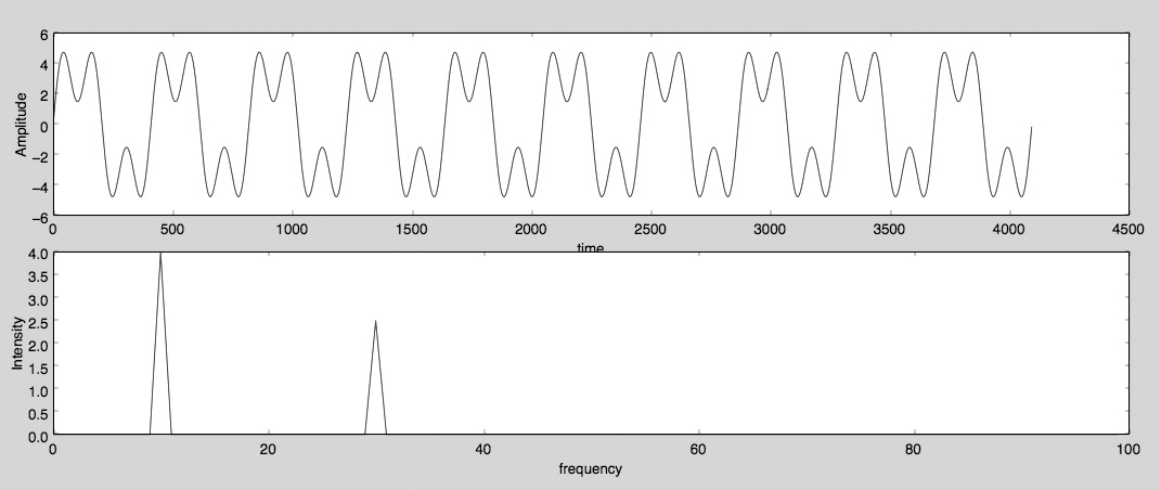


Figure 13-8: *An audio signal containing multiple frequencies (top) and its corresponding FFT (bottom)*

The wave in the top frame can be expressed by the following equation, which sums the two sine waves:

$$y(t) = 4\sin(2\pi 10t) + 2.5\sin(2\pi 30t)$$

Notice the 4 and 10 in the expression for the first wave—4 is the amplitude of the wave, and 10 is the frequency (in hertz). Meanwhile, the second wave has an amplitude of 2.5 and a frequency of 30 Hz. Take a look at the FFT in the bottom frame of the figure and you'll see it has two peaks, at 10 Hz and 30 Hz. The FFT has revealed the signal's component frequencies. The FFT also identifies the relative amplitude of each frequency; the intensity of the first peak is about twice that of the second peak.

Now let's look at a more realistic example. Figure 13-9 shows a complex audio signal in the top frame and the corresponding FFT in the bottom frame. Notice that the FFT contains many more peaks at a variety of intensities, indicating that the signal contains many more frequencies.

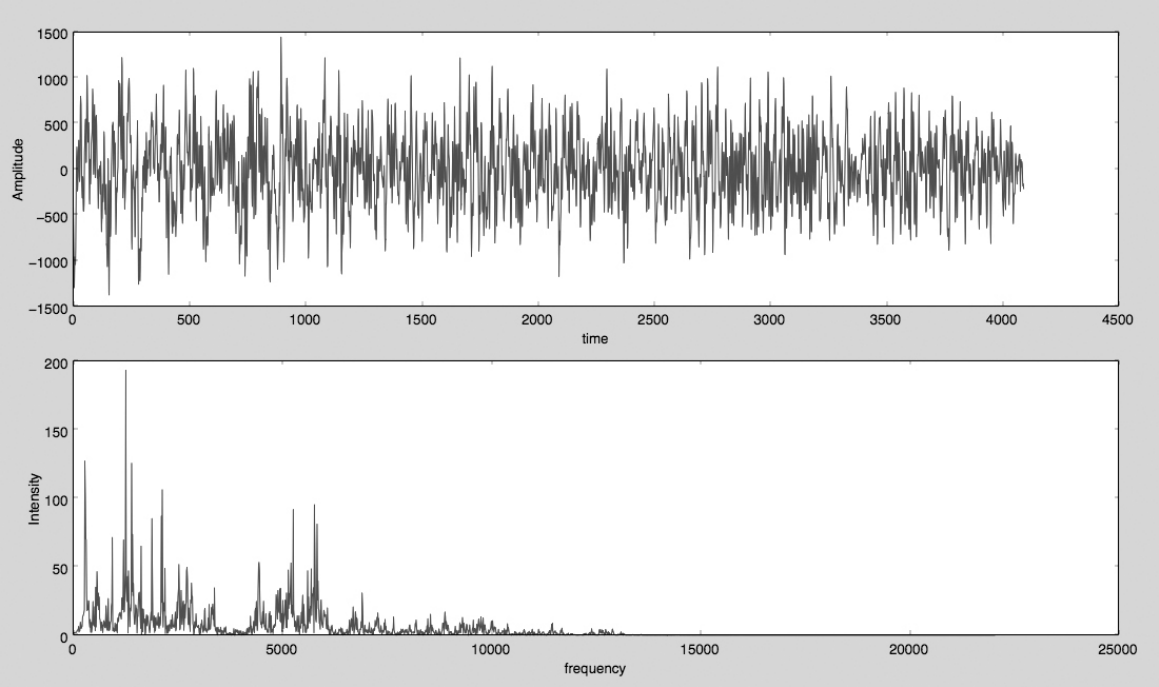


Figure 13-9: *The FFT algorithm takes an amplitude signal (top) and computes its component frequencies (bottom).*

To compute an FFT, you need a set of samples. The choice of the number of samples is a bit arbitrary, but a small sample size wouldn’t give you a good picture of the signal’s frequency content and might also mean a higher computational load because you would need to compute more FFTs per second. On the other hand, a sample size that’s too large would average out the changes in the signal, so you wouldn’t be getting a “real-time” frequency response for the signal. For the purposes of this project, a sample size of 2,048 will work. At a sampling rate of 44,100 Hz, 2,048 samples represent about 0.046 seconds of audio.

You’ll use `numpy` to compute an FFT to split the audio data into its constituent frequencies, and then you’ll use that information to control the motors. First you’ll split the range of frequencies (in Hz) into three bands: [0, 100], [100, 1000], and [1000, 2500]. You’ll compute an average amplitude level for each band, and each value will affect the motors and resulting laser pattern differently, as follows:

- Changes in the average amplitude of low frequencies will affect the speed of the first motor.
- Changes in the average amplitude of middle frequencies will affect the speed of the second motor.
- When high frequencies peak above a certain threshold, the first motor will change direction.

With these rules, the laser pattern will change in response to the audio signal.

Requirements

In this project, you'll use the following Python modules:

- `RPi.GPIO` to set up PWM and control the output of the pins
- `time` for pausing between operations
- `wave` to read WAV files
- `pyaudio` to process and stream audio data
- `numpy` for FFT computations
- `argparse` to process command line arguments

You'll also need the following items to build the project:

- One Raspberry Pi 3B+ or newer
- One 5 V adapter to power the Raspberry Pi
- One powered speaker with AUX (line-in) input (most Bluetooth speakers have an AUX input these days)
- One TB6612FNG motor breakout board
- One laser module breakout board

- One 10 kΩ resistor
- One BS170 N-channel MOSFET or equivalent
- Two DC motors like the ones used in a small toy, rated for 9 V
- Two small mirrors, approximately 1 inch or less in diameter
- One 3.7 V 18650 2000 mAh (3C) lithium-ion battery with holder (or use four AA batteries on a holder)
- Two 3D-printed parts to fix the mirrors onto the motor shafts (optional)
- One rectangular base, about 8 inches by 6 inches, to mount the hardware
- Some LEGO bricks to raise the motors and laser module off the base so the mirrors can spin freely
- A hot glue gun
- Superglue to fix the mirrors to the motor shafts
- A soldering iron
- A breadboard
- Wires to make connections (single-core hookup wires with male pins on both sides work nicely)

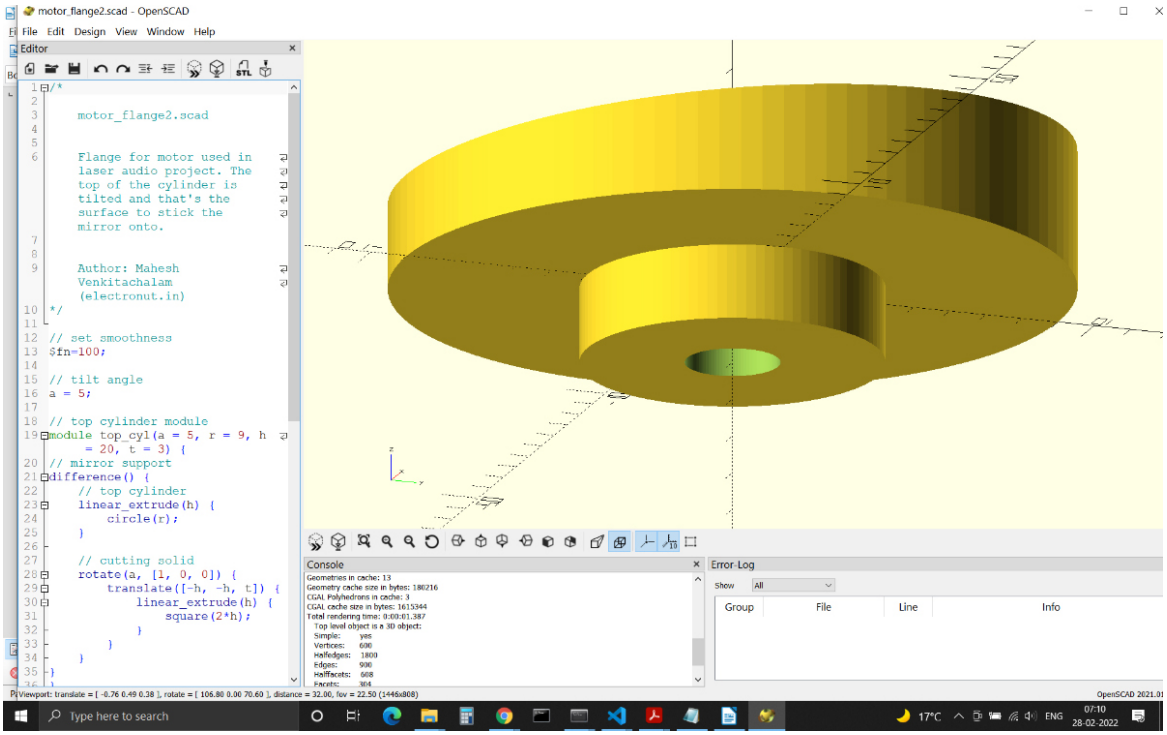
Setting Up the Raspberry Pi

To set up your Raspberry Pi, see Appendix B. Follow the instructions in the appendix, and make sure you've installed the `numpy` and `pyaudio`Python packages required for this project. You'll write code on the Raspberry Pi via a Secure Shell (SSH). You can set up Microsoft Visual Studio Code to work remotely on the Pi using an SSH from your laptop or desktop computer. This is also explained in Appendix B.

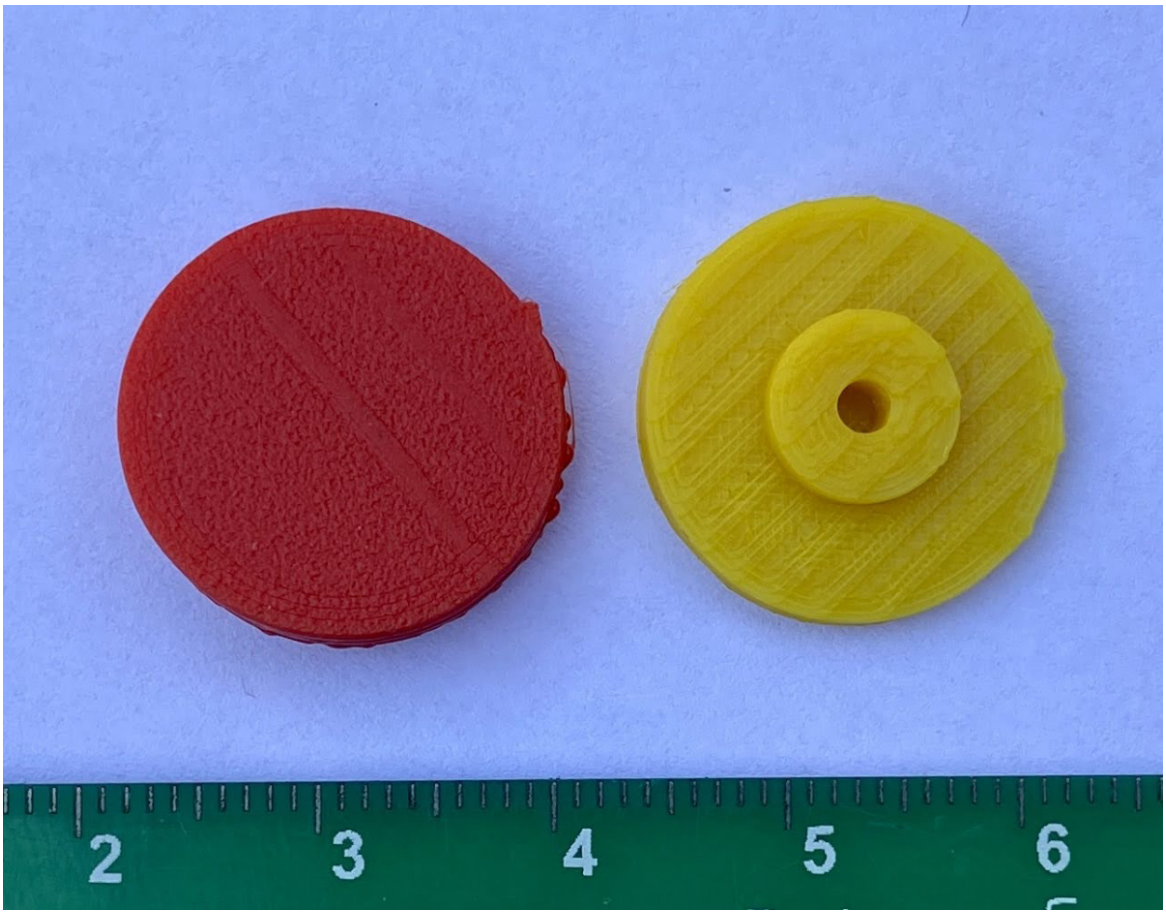
Constructing the Laser Display

Before you hook up all the hardware, you should prepare the motors and the laser module for the laser display. The first order of business is to attach the mirrors to the motors. Each mirror has to be at a slight angle relative to the motor shaft. One way to do this would be to use hot glue. To attach the mirror, place it face down on a flat surface and put a drop of hot glue in the center. Carefully dip the motor shaft in the glue, keeping it at a slight angle to the perpendicular with respect to the mirror, until the glue hardens.

A better way would be to use a motor flange with a slanted face to which you can easily stick the mirror. But where would you find such a part? You can make it yourself using 3D printing! Figure 13-10(a) shows a 3D design I created using the free, open source program called OpenSCAD. You can download the design from this book's GitHub repository. Figure 13-10(b) shows the 3D printed parts. The mirror the laser strikes first will use the flange with the lesser tilt (5 degrees), and the second mirror will use the one with more tilt (10 degrees).


motor_flange2.scad - OpenSCAD
File Edit Design View Window Help
Editor
/*
motor_flange2.scad
Flange for motor used in laser audio project. The top of the cylinder is tilted and that's the surface to stick the mirror onto.
Author: Mahesh Venkitachalam (electronut.in)
*/
// set smoothness
$fn=100;
// tilt angle
a = 5;
// top cylinder module
module top_cyl(a = 5, r = 9, h = 20, t = 3) {
// mirror support
difference() {
// top cylinder
linear_extrude(h) {
circle(r);
}
// cutting solid
rotate(a, [1, 0, 0]) {
translate([-h, -h, t]) {
linear_extrude(h) {
square(2*h);
}
}
}
}
Console
Geometries in cache: 13
Geometry cache size in bytes: 186216
CGAL Polyhedrons in cache: 3
CGAL cache size in bytes: 1615344
Total rendering time: 0:00:01.387
Top level object is a 3D object:
Simple: yes
Vertices: 600
Halfedges: 1800
Edges: 900
Halffacets: 608
Facets: 304
Error-Log
Show All
Group File Line Info
Type here to search
17°C
ENG
07:10
28-02-2022

(a)



(b)

Figure 13-10: *An OpenSCAD model (a) and 3D-printed flanges (b)*

Print the flanges yourself if you have a 3D printer, or get them printed from a 3D printing service. (Either way, it will be inexpensive.) Once you have the parts, use superglue to attach the flanges to the motor shafts and the mirrors to the flanges. Figure 13-11 shows the fully assembled parts.

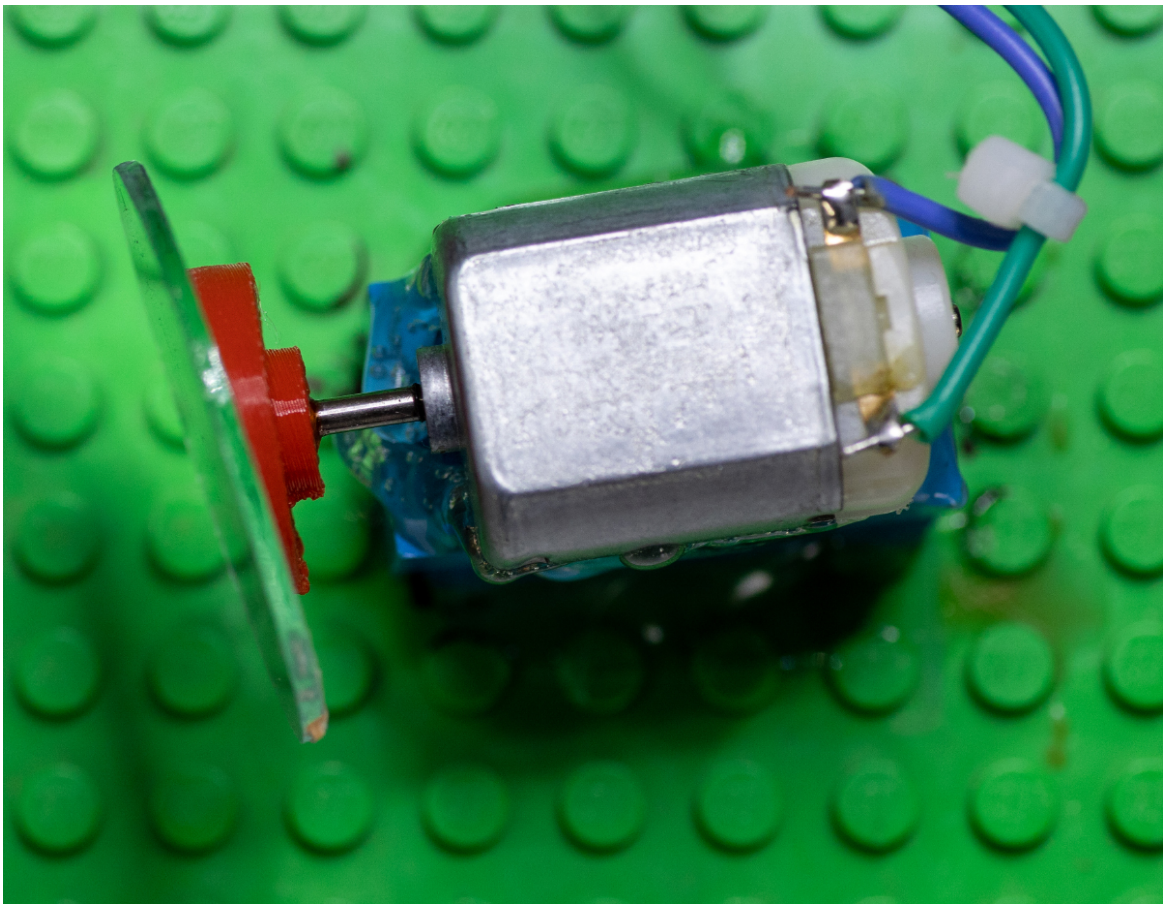


Figure 13-11: *Attach the mirrors to each motor shaft at a slight angle.*

To test the assembly, spin the mirror with your hand while shining the laser module at it. You should find the reflection of the laser dot moves in an ellipse when projected on a flat surface. Do the same for the second mirror. It should create a wider ellipse because of the larger angle with respect to the motor shaft.

Aligning the Mirrors

Next, align the laser module with the mirrors so that the laser reflects from mirror A to mirror B, as shown in Figure 13-12. Be sure that the reflected laser light from mirror A stays within the circumference of mirror B for mirror A's entire range of rotation. (This will take some trial and error.) To test the arrangement, manually rotate mirror A. Also, be

sure to position mirror B so that the light reflected from its surface will fall on a flat surface (like a wall) for the full range of rotation of both mirrors.

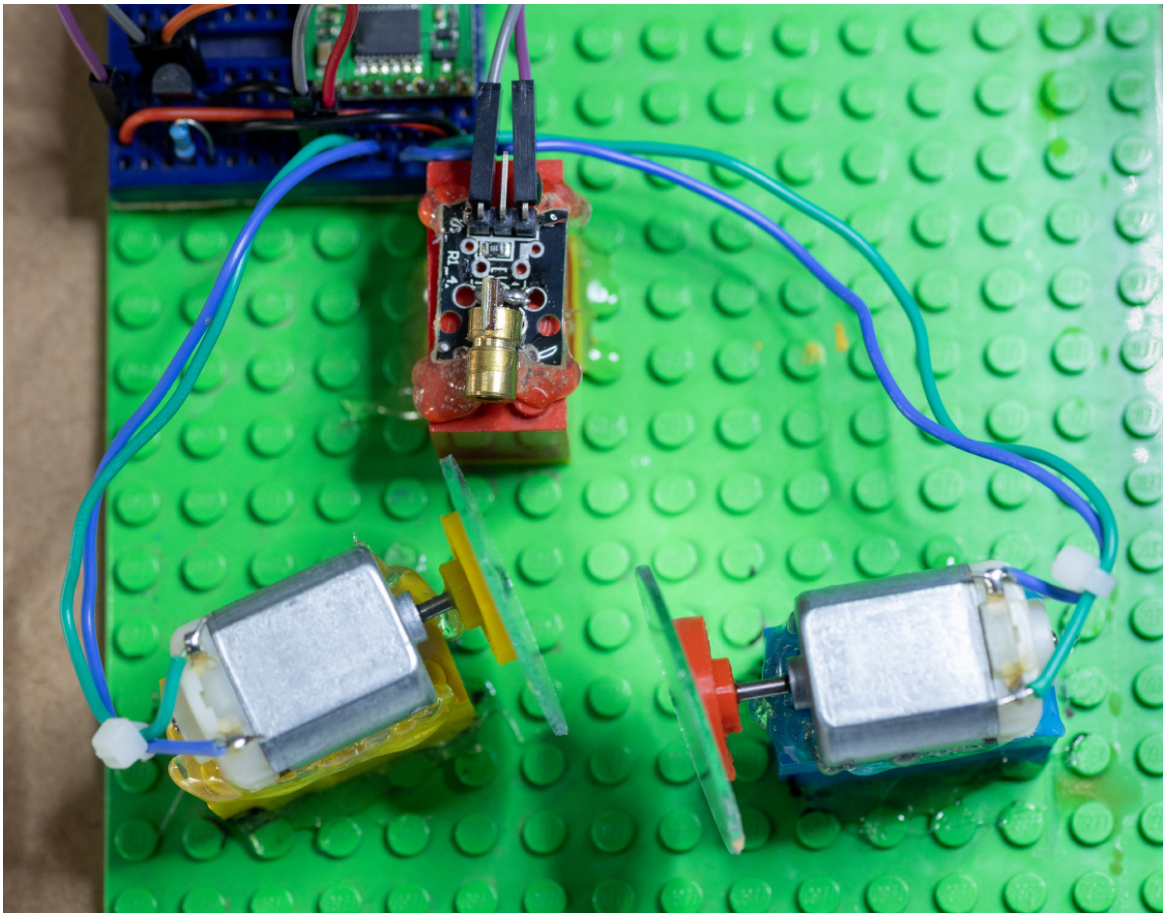


Figure 13-12: *The alignment of the laser and the mirrors*

NOTE *As you tweak the alignment, you'll need to keep the laser pointer on. You can do this by running the project code as follows:* `python laser_audio.py --test_laser`. *This command simply turns on the MOSFET controlling the laser module, as we'll discuss later in the chapter.*

Once you're happy with the placement of the mirrors, hot-glue the laser module and the two motors with attached mirrors onto three identical blocks (LEGO blocks work great!) to raise them up so that the motors will be able to rotate freely. Next, place the blocks on the mounting board,

and when you're happy with their arrangement, mark the location of each by tracing their edge with a pencil. Then hot-glue the blocks onto the board. Or use a LEGO baseplate and simply attach the LEGO blocks to the baseplate.

Powering the Motors

If your motors didn't come with wires attached to their terminals (most don't), solder wires to both terminals, being sure to leave sufficient wire (say, 6 inches) so that you can attach the motors to the motor driver board. The motors can be powered by a 3.7 V lithium-ion battery, or a 4x AA battery pack.

Hooking Up the Hardware

Now to hook up the hardware. You need to connect the Raspberry Pi, the motor driver board, the MOSFET, the laser module board, and the motors. The Raspberry Pi has a collection of GPIO pins for connecting to other hardware. To understand the pin layout, I highly recommend you visit the website *https://pinout.xyz*. It gives you a handy visual reference and explains the various pin functions.

NOTE *There are a few different conventions for referring to pin numbers on a Raspberry Pi. For this project, we'll use the* BCM pin-numbering *convention.*

Table 13-1 lists the connections you need to make.

Table 13-1: Hardware Wiring Connections

From	To

From	To
Raspberry Pi GPIO 12	TB6612FNG PWMA
Raspberry Pi GPIO 13	TB6612FNG PWMB
Raspberry Pi GPIO 7	TB6612FNG AIN1
Raspberry Pi GPIO 8	TB6612FNG AIN2
Raspberry Pi GPIO 5	TB6612FNG BIN1
Raspberry Pi GPIO 6	TB6612FNG BIN2
Raspberry Pi GPIO 22	TB6612FNG STBY
Raspberry Pi GND	TB6612FNG GND
Raspberry Pi 3V3	TB6612FNG VCC
Raspberry Pi GPIO 25	BS170 GATE (also to GND via 10 kΩ resistor)
Raspberry Pi GND	BS170 SOURCE
Laser module GND	BS170 DRAIN
Laser module VCC	Battery pack VCC (+)
Battery pack GND (−)	TB6612FNG GND

From	To
Battery pack VCC (+)	TB6612FNG VM
Motor #1 connector #1 (polarity doesn’t matter)	TB6612FNG A01
Motor #1 connector #2 (polarity doesn’t matter)	TB6612FNG A02
Motor #2 connector #1 (polarity doesn’t matter)	TB6612FNG B01
Motor #2 connector #2 (polarity doesn’t matter)	TB6612FNG B02
Raspberry Pi 3.5 mm audio jack	AUX input of powered speaker

Figure 13-13 shows everything wired up.

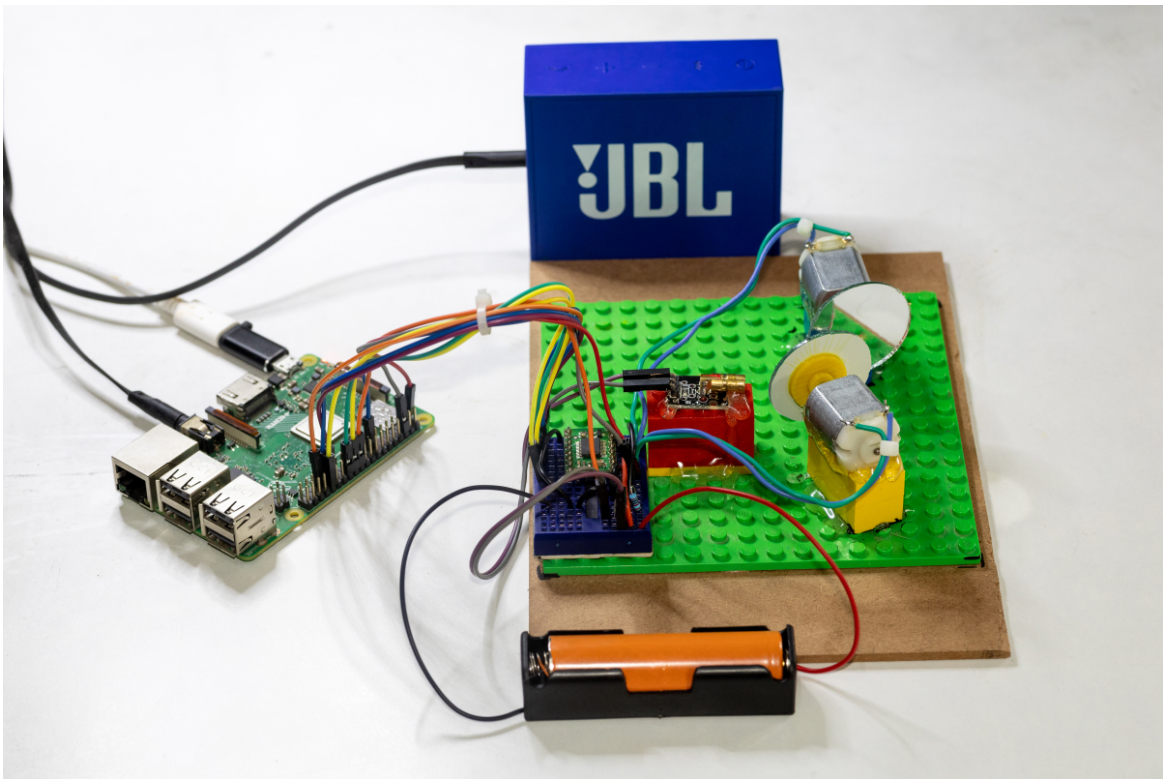


Figure 13-13: *The completely wired-up laser display*

Now let’s look at the code.

The Code

The code for this project is in the file *laser_audio.py*. You’ll begin with some basic setup. Then you’ll define functions to operate and test the motors and laser, as well as a function to process audio data from a WAV file and control the motors based on that data. You’ll bring everything together and accept command line options through the `main()` function. To see the full program, skip ahead to “The Complete Code” on page 305. You can also download the code at *https://github.com/mkvenkit/pp2e/tree/main/laser_audio*.

Setting Up

Start by importing the required modules:

```
import RPi.GPIO as GPIO
import time
import argparse
import pyaudio
import wave
import numpy as np
```

The `RPi.GPIO` module lets you use the pins of the Raspberry Pi. You'll use the `time` module to put delays in your code, and you'll use `argparse` to add command line arguments to the program. The `pyaudio` and `wave` modules will help you read data from WAV files and output an audio stream. Finally, you'll use `numpy` to compute the FFT of the audio data.

Next, you initialize a few global variables:

```
# define pin numbers
# uses TB6612FNG motor driver pin naming
PWMA = 12
PWMB = 13
AIN1 = 7
AIN2 = 8
BIN1 = 5
BIN2 = 6
STBY = 22
LASER = 25
```

This code stores the pin numbers for all the Raspberry Pi pins used in the project. `PWMA`, `PWMB`, `AIN1`, `AIN2`, `BIN1`, `BIN2`, and `STBY` are pins that connect to the TB6612FNG motor driver. The `LASER` pin will connect to the gate of the MOSFET, which can turn the laser module on and off. Note that you're using the BCM pin-numbering convention here.

Continue with a few more global variables:

```
# global PWM objects
pwm_a = None
pwm_b = None
# size of audio data read in
CHUNK = 2048
# FFT size
N = CHUNK
```

Here you initialize the variables `pwm_a` and `pwm_b` that will represent `PWM` objects, which you'll use to control the motors. You set them to `None` here because it's too early in the code to create the actual `PWM` objects. You also set `CHUNK`, the number of audio data samples you'll read in from the WAV file at a time, and `N`, the number of samples you'll use to compute the FFT.

You finish setting up by initializing the GPIO pins. This is necessary to use the pins. Define a function `init_pins()` for this purpose:

```
def init_pins():
    """set up pins"""
  ❶ global pwm_a, pwm_b
    # use BCM pin numbering
  ❷ GPIO.setmode(GPIO.BCM)
    # put pins into a list
    pins = [PWMA, PWMB, AIN1, AIN2, BIN1, BIN2, STBY, LASER]
    # set up pins as outputs
  ❸ GPIO.setup(pins, GPIO.OUT)
    # set PWM
    pwm_a = GPIO.PWM(PWMA, 100)
    pwm_b = GPIO.PWM(PWMB, 100)
```

First you indicate that `pwm_a` and `pwm_b` are global variables ❶, since you'll be setting them inside this function. Then you set the pin mode to the BCM numbering convention ❷. Next, you put the pin variables you set earlier into a `pins` list so you can declare them all to be output pins with one call ❸. Finally, you create two `PWM` objects and assign them to your global `pwm_a` and `pwm_b` variables. The argument `100` is the frequency, in hertz, of the signal that will drive each motor. You'll vary the duty cycles of those signals to control the motors' speeds using pulse width modulation.

Controlling the Hardware

You need some helper functions to control the laser module and the motors. Let's look first at the function that toggles the laser module:

```
def laser_on(on):
    # pin 25 controls laser ctrl mosfet
    GPIO.output(LASER, on)
```

This function takes a parameter `on` that will be a Boolean `True/False` value. You pass that parameter to the `GPIO.output()` method to set the `LASER` pin on (`True`) or off (`False`). This will trigger the MOSFET to turn the laser module on or off.

Next, define a function `start_motors()` to turn the motors on at the start of the project:

```
def start_motors():
    """start both motors"""
    # enable driver chip
  ❶ GPIO.output(STBY, GPIO.HIGH)
    # set motor direction for channel A
  ❷ GPIO.output(AIN1, GPIO.HIGH)
```

```
  ❸ GPIO.output(AIN2, GPIO.LOW)
    # set motor direction for channel B
    GPIO.output(BIN1, GPIO.HIGH)
    GPIO.output(BIN2, GPIO.LOW)
    # set PWM for channel A
    duty_cycle = 10
  ❹ pwm_a.start(duty_cycle)
    # set PWM for channel B
    pwm_b.start(duty_cycle)
```

First you set the `STBY` (standby) pin to `HIGH` ❶, which in effect turns on the motor driver. Then you set the `AIN1` and `AIN2` pins to `HIGH` ❷ and `LOW` ❸, respectively. This will cause motor A to spin in one direction. (Swapping the `HIGH/LOW` values between the two pins would cause the motor to spin in the opposite direction.) You do the same thing for motor B. Lastly, you use your `PWM` objects to set the speed of the motors ❹. You set the duty cycle (which correlates to the motor speed) to 10 percent, a relatively low value, since this is just an initializing call.

You also need a function to stop the motors from spinning at the end of the project. Here's the definition:

```
def stop_motors():
    """stop both motors"""
    # stop PWM
  ❶ pwm_a.stop()
  ❷ pwm_b.stop()
    # brake A
    GPIO.output(AIN1, GPIO.HIGH)
    GPIO.output(AIN2, GPIO.HIGH)
    # brake B
    GPIO.output(BIN1, GPIO.HIGH)
    GPIO.output(BIN2, GPIO.HIGH)
    # disable driver chip
  ❸ GPIO.output(STBY, GPIO.LOW)
```

To stop the motors from spinning, you first stop the PWM signals going to the `PWMA` ❶ and `PWMB` ❷ pins. Then you set the `AIN1`, `AIN2`, `BIN1`, and `BIN2` pins all to `HIGH`, which has the effect of "braking" each motor to a stop. Finally, you disable the motor driver by setting the `STBY` pin to `LOW` ❸. Standby mode saves power when the motors aren't required to function.

You need one more helper function to set the speed and direction of both motors. You'll use this function to adjust the motors based on your real-time audio analysis.

```
def set_motor_speed_dir(dca, dcb, dira, dirb):
    """set speed and direction of motors"""
    # set duty cycle
  ❶ pwm_a.ChangeDutyCycle(dca)
    pwm_b.ChangeDutyCycle(dcb)
    # set direction A
  ❷ if dira:
        GPIO.output(AIN1, GPIO.HIGH)
        GPIO.output(AIN2, GPIO.LOW)
  ❸ else:
        GPIO.output(AIN1, GPIO.LOW)
        GPIO.output(AIN2, GPIO.HIGH)
    if dirb:
        GPIO.output(BIN1, GPIO.HIGH)
        GPIO.output(BIN2, GPIO.LOW)
    else:
        GPIO.output(BIN1, GPIO.LOW)
        GPIO.output(BIN2, GPIO.HIGH)
```

The `set_motor_speed_dir()` function takes four parameters: `dca` and `dcb` determine the duty cycle for each motor, while `dira` and `dirb` are Booleans that set the motors' direction of rotation. You use the `ChangeDutyCycle()` method to update the duty cycles (speeds) of the motors to the values passed into the function ❶. Then you attend to the motors' directions. If `dira` is `True` ❷, you set pins `AIN1` and `AIN2` to `HIGH` and `LOW`,

respectively, which will let motor A spin in one direction. However, if `dira` is `False` ❸, you set the pins the opposite way, which will spin the motor in the other direction. You do the same thing for motor B using the `dirb` parameter.

Processing the Audio

The heart of this project is the `process_audio()` function, which reads audio data from a WAV file, outputs an audio stream with `pyaudio`, analyzes the audio data by computing an FFT, and uses the resulting analysis to control the motors. We'll look at the function in sections.

```
def process_audio(filename):
    print("opening {}...".format(filename))
    # open WAV file
  ❶ wf = wave.open(filename, 'rb')
    # print audio details
  ❷ print("SW = {}, NCh = {}, SR =
  {}".format(wf.getsampwidth(),
        wf.getnchannels(), wf.getframerate()))
    # check for supported format
  ❸ if wf.getsampwidth() != 2 or wf.getnchannels() != 1:
        print("Only single channel 16 bit WAV files are
  supported!")
        wf.close()
        return
    # create PyAudio object
  ❹ p = pyaudio.PyAudio()
    # open an output stream
  ❺ stream =
  p.open(format=p.get_format_from_width(wf.getsampwidth()),
                    channels=wf.getnchannels(),
                    rate=wf.getframerate(),
                    output=True)
    # read first frame
```

You start by using the `wave` module to open the audio file passed into the `process_audio()` function ❶. The `wave.open()` function returns a `Wave_read` object, which you'll use to read data from the WAV file. You print out some information about the WAV file read in ❷: `SW` is the sample width in bytes, `NCh` is the number of channels in the audio, and `SR` is the sampling rate. To keep the project simple, you'll support only single-channel, 16-bit WAV files as input. You check for these specs ❸, and if the input doesn't match, you return from the function.

Next, you create the `PyAudio` object that you'll use to stream data from the WAV file to the output ❹. Then you open a `pyaudio` output stream (as indicated by the `output=True` argument), configuring it to have the same sample width, number of channels, and sample rate as the WAV file ❺. For the Raspberry Pi, the default sound output is the 3.5 mm audio jack on the board. As long as your speaker is plugged into this jack, you'll hear the sound output.

Here's the next part of the function:

```
❶ data = wf.readframes(CHUNK)
❷ buf = np.frombuffer(data, dtype=np.int16)
  # store sample rate
❸ SR = wf.getframerate()
  # start motors
  start_motors()
  # laser on
  laser_on(True)
```

Here you read `CHUNK` samples from the WAV file into variable `data` ❶. Remember, you've set `CHUNK` to `2048`, and each sample is 2 bytes wide, so you'll be reading a total of 2,048 16-bit values. You read only one chunk of data because of how the function's main loop is structured, as you'll see soon.

The `readframes()` method returns a `bytes` object, but you use the `numpy` library's `frombuffer()` function to convert the `bytes` object into a `numpy` array of 16-bit integers called `buf` ❷. You store the sampling rate (the `wave` module calls it the *frame rate*) in variable `SR` ❸; you'll need it later. Then you call the `start_motors()` and `laser_on()` functions, which we've already discussed, to get the motors and the laser module going.

Next, you enter the function's main loop, which outputs the audio and performs the FFT. The loop works on one chunk of audio data at a time, which is why you read only a single chunk of data in the previous code listing. Notice that the loop occurs inside a `try` block. Later, you'll write an `except` block to handle any problems that arise during the execution of the loop.

```
    # read audio data from WAV file
    try:
        # loop till there is no data to be read
      ❶ while len(data) > 0:
            # write stream to output
          ❷ stream.write(data)
            # ensure enough samples for FFT
          ❸ if len(buf) == N:
              ❹ buf = np.frombuffer(data, dtype=np.int16)
                # do FFT
              ❺ fft = np.fft.rfft(buf)
              ❻ fft = np.abs(fft) * 2.0/N
                # calc levels
                # get average of 3 frequency bands
                # 0-100 Hz, 100-1000 Hz, and 1000-2500 Hz
              ❼ levels = [np.sum(fft[0:100])/100,
                          np.sum(fft[100:1000])/900,
                          np.sum(fft[1000:2500])/1500]
```

The main loop repeats until `data` is empty ❶, meaning you've gotten to the end of the WAV file. In the loop, you write the current chunk of data to the `pyaudio` output stream ❷. This will allow you to hear the WAV file you're processing at the same time that you're driving the motors. Then you check whether you have `N` samples in the current chunk of data to calculate the FFT ❸ (you set `N` to `2048`, the same as the chunk size, at the start of the code). This check is required because the last chunk of data read in may not have the full number of samples needed for the FFT. In that case, you'll simply skip calculating the FFT and updating the motors, since the audio file is basically over anyway.

Next, you load the audio data into a `numpy` array of 16-bit integers ❹. With the data in this format, computing the FFT is quite straightforward: you simply use the `rfft()` method from the `numpy.fft` module ❺. This method takes a signal composed of *real numbers* (like the audio data) and computes the FFT, which generally results in a set of *complex numbers*. You want to keep working with real numbers, however, so you use the `abs()` method to get the magnitudes of these complex numbers, which are real ❻. The `2.0/N` is a normalization factor you use to map the FFT values to the expected range.

Continuing the loop, you extract the relevant information from the FFT to control the motors. To analyze the audio signal, you split the frequency range into three bands: 0 to 100 Hz (bass), 100 to 1,000 Hz (midrange), and 1,000 to 2,500 Hz (treble). You're especially interested in the bass and midrange frequency bands, which roughly correspond to the beat and the vocals in a song, respectively. You compute the average amplitude value of the frequencies in each band by using the `numpy.sum()` method and dividing the result by the number of frequencies in that band ❼. You store the three averages in a Python list.

Notice that you're doing two different things in the `while` loop: sending audio to the output and computing the FFT of the same audio. You're able to do this and maintain the audio output because the `numpy` FFT computation is fast enough—it finishes before the current chunk of audio data has finished playing. Try an experiment: put a time delay right after FFT, and see what happens to the sound output!

Now you need to convert the average amplitudes from the FFT into motor speeds and directions, still within the `while` loop started in the previous listing. The speeds need to be percentages, while the directions need to be `True/False` values.

```
  # speed1
❶ dca = int(5*levels[0])  percent 60
  # speed2
❷ dcb = int(100 + levels[1])  percent 60
  # dir
  dira = False
  dirb = True
❸ if levels[2] > 0.1:
      dira = True
  # set motor direction and speed
❹ set_motor_speed_dir(dca, dcb, dira, dirb)
```

First you take the value from the lowest frequency band, scale it by a factor of `5`, convert it to an integer, and use the modulus operator (`percent`) to ensure that the value lies within the [0, 60] range ❶. This value controls the speed of motor A. Then you add `100` to the value from the middle frequency band and again use the modulus operator to place it in the [0, 60] range ❷. This value controls the speed of motor B.

NOTE *It's not a good idea to run the motors too fast at first, which is why this code limits the motors' speeds to*

60 percent. Once you're confident the display is working, you can try increasing the speed threshold at ❶ and ❷.

By default, you set `dira` to `False` to run motor A in one direction, but you switch to the other direction if the value from the highest frequency band crosses above a threshold of `0.1` ❸. Meanwhile, you keep motor B's direction constant by setting `dirb` to `True`. Finally, you call your `set_motor_speed_dir()` function to run the motors at the speeds and directions you've calculated ❹.

NOTE *There's no particularly elegant rule governing how you convert the FFT information to motor speeds and directions. The FFT values change constantly with the audio signal, so any method you come up with will change the laser pattern in response to the music. I arrived at the method described here through trial and error; I looked at FFT values while playing various types of music and chose calculations that produced a nice variety of patterns. I encourage you to play with the calculations and create your own conversions. There are no wrong answers here, as long as your method puts the motor speeds in the [0, 100] range (or less to avoid high speeds) and sets the directions to be* `True` *or* `False`*.*

Here's the remainder of the `process_audio()` function:

```
            # read next
          ❶ data = wf.readframes(CHUNK)
    ❷ except BaseException as err:
          print("Unexpected {}, type={}".format(err, type(err)))
    ❸ finally:
            print("Finally: Pyaudio clean up...")
            stream.stop_stream()
            stream.close()
            # stop motors
            stop_motors()
```

First you conclude the `while` loop by reading in the next chunk of audio data to be processed ❶. Recall that the `while` loop unfolds inside a `try` block. The `except` block ❷ catches any exceptions that may arise during the loop. For example, pressing CTRL-C while the program is running will throw an exception and halt the loop, as will any errors reading the data. You end by doing some cleanup in a `finally` block ❸ that will execute whether or not any exceptions are thrown. In this block, you stop the `pyaudio` output stream and close it, and you call your `stop_motors()` function to stop the motors from spinning.

Testing the Motors

For testing purposes, it would be useful to be able to manually set the speed and direction of the motors and view the resulting laser pattern. Here's a `test_motors()` function that makes this possible:

```
def test_motors():
    """test motors by manually setting speed and direction"""
    # turn laser on
  ❶ laser_on(True)
    # start motors
  ❷ start_motors()
    # read user input
    try:
        while True:
          ❸ print("Enter dca dcb dira dirb (eg. 50 100 1 0):")
            # read input
            str_in = input()
            # parse values
          ❹ vals = [int(val) for val in str_in.split()]
            # sanity check
            if len(vals) == 4:
```

```
            ❺ set_motor_speed_dir(vals[0], vals[1], vals[2],
vals[3])
          else:
              print("Input error!")
  except:
      print("Exiting motor test!")
❻ finally:
      # stop motors
      stop_motors()
      # turn laser off
      laser_on(False)
```

You begin by turning on the laser ❶ and starting the motors ❷. Then you enter a loop to get information from the user. The loop prompts the user to enter four integer values ❸: `dca` and `dcb` are the duty cycles (speeds) for the motors (from 0 to 100), and `dira` and `dirb` are the motor directions (0 or 1). You wait for the input, and then parse it, using `split()` to divide the input string into separate strings based on whitespace and a list comprehension to convert each substring into an integer ❹. After a sanity check to ensure that you did, in fact, get four numbers as input, you run the motors using the provided values ❺.

This runs in a loop, so you can try inputting various speed and direction values to see the result. Since the `while` loop is placed inside a `try` block, pressing CTRL-C will throw an exception and exit from the test when you're ready. Then, in the `finally` block ❻, you stop the motors and turn the laser off.

Putting It All Together

As usual, the `main()` function takes in command line arguments and sets the project in motion. Let's look at the command line arguments first:

```
def main():
    """main calling function"""
    # set up args parser
  ❶ parser = argparse.ArgumentParser(description="A laser
  audio display.")
    # add arguments
    parser.add_argument('--test_laser', action='store_true',
  required=False)
    parser.add_argument('--test_motors', action='store_true',
  required=False)
    parser.add_argument('--wav_file', dest='wav_file',
  required=False)
    args = parser.parse_args()
```

Here you follow the familiar pattern of creating an `ArgumentParser` object to parse command line arguments for the program ❶. The program will support three different command line arguments. The `--test_laser` option just turns the laser on and is useful when you're building the motor and laser assembly. The `--test_motors` option is for testing the motors, and the `--wav_file` option lets you specify the WAV file to be read in for the laser audio display.

Here's the rest of the `main()` function:

```
    # initialize pins
  ❶ init_pins()

    # main loop
    try:
      ❷ if args.test_laser:
            print("laser on...")
            laser_on(True)
            try:
                # wait in a loop
                while True:
                    time.sleep(0.1)
            except:
                # turn laser off
```

```
            laser_on(False)
    ❸ elif args.test_motors:
            print("testing motors...")
            test_motors()
    ❹ elif args.wav_file:
            print("starting laser audio display...")
            process_audio(args.wav_file)
  except (Exception) as e:
      print("Exception: {}".format(e))
      print("Exiting.")

  # turn laser off
❺ laser_on(False)
  # call at the end

❻ GPIO.cleanup()
  print("Done.")
```

You call the `init_pins()` function you defined earlier to initialize the Raspberry Pi's GPIO pins ❶. Next, you go through and process the command line arguments. If the user typed in the `--test_laser` argument, `args.test_laser` will be set to `True`. You handle this case by turning on the laser and waiting for the user to press CTRL-C to terminate the loop ❷. Similarly, you handle the `--test_motors` option by calling `test_motors()` ❸. To start the laser audio display, the user needs to use the `--wav_file` command line argument. In this case ❹, you call the `process_audio()` function. Once again, all this is embedded in a `try` block, so when the user presses CTRL-C, you break out of the loop for any of the three modes. To finish, you turn the laser off ❺ and do a GPIO cleanup ❻ before exiting the program.

Running the Laser Display

To test the project, assemble the hardware, make sure the battery pack is connected, and position everything so the laser is projecting onto a flat surface such as a wall. Then use SSH to log in to your Raspberry Pi as discussed in Appendix B and run the program from a shell. I recommend testing the laser display part first by running the program in test mode.

WARNING *The project has high-speed spinning mirrors. Please wear appropriate eye protection or cover the setup with a transparent box before running the program to avoid injuries.*

Here's a sample run of the test mode:

```
$ python laser_audio.py --test_motors
testing motors...
Enter dca dcb dira dirb (eg. 50 100 1 0):
30 40 0 1
Enter dca dcb dira dirb (eg. 50 100 1 0):
40 30 1 0
```

You can use this test to run both motors through various combinations of speeds and directions. You should see different laser patterns projected onto your wall as you change the values. To stop the program and the motors, press CTRL-C. Note that if you enter duty cycle (speed) values greater than 80, the motors will spin really fast. Be careful!

If the test succeeds, you're ready to move on to the real show. Copy a WAV file of your favorite music to the Raspberry Pi. Remember that to keep things simple, the program accepts only single-channel WAV files in 16-bit format. You can convert any audio file to this format using the free software Audacity. (A sample file is also available in

the project's GitHub repository.) When the audio file is in place, run the program as follows, substituting your desired filename after the `--wave_file` option:

```
python3 laser_audio.py --wav_file bensound-allthat-16.wav
```

You should see the laser display produce lots of interesting patterns that change in time with the music, as shown in Figure 13-14.

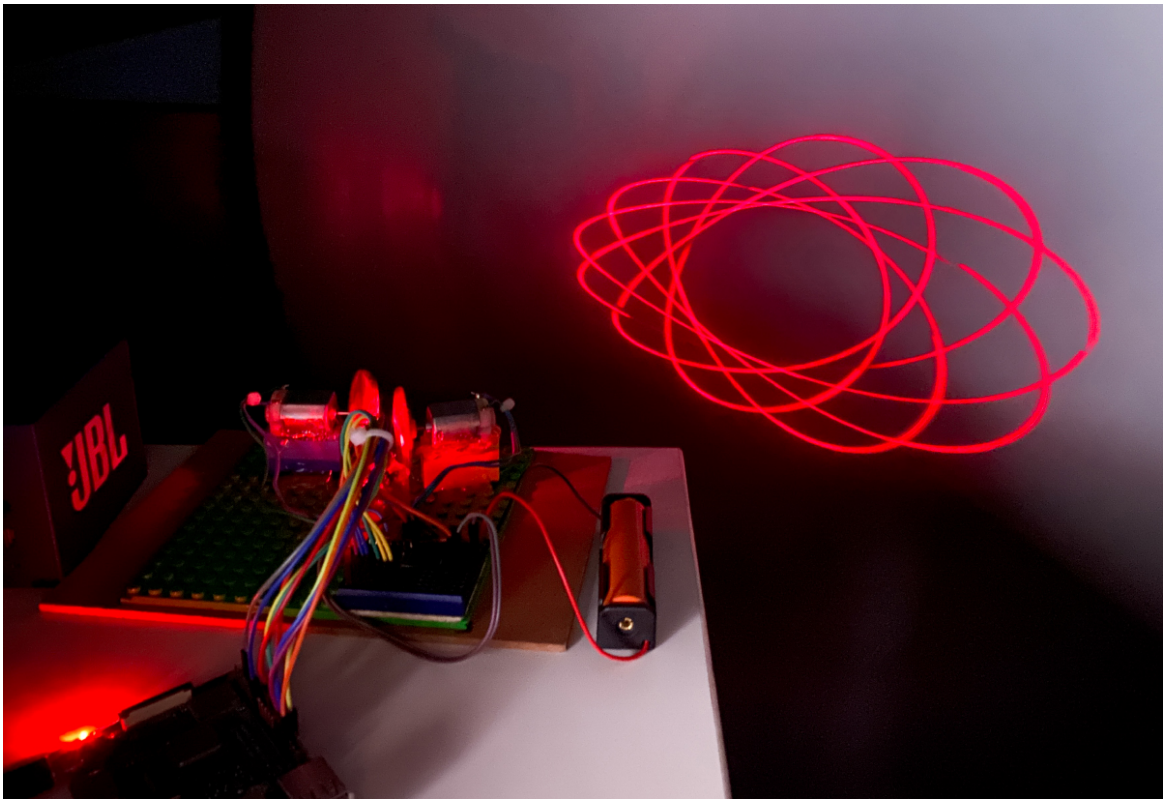


Figure 13-14: *The complete wiring of the laser display and a pattern projected on the wall*

Try experimenting with different WAV files, or with different calculations for converting the FFT information to motor settings, to see how the visualization responds.

Summary

In this chapter, you upped your Python and hardware skills by building a fairly complex project. You learned how to control motors with Python, a Raspberry Pi, and a motor driver. You used `numpy` to compute the FFT of audio data, and you used `pyaudio` to stream audio output in real time. You even learned to control a laser using a MOSFET!

Experiments!

Here are some ways you can modify this project:

1. The program used an arbitrary scheme to convert the FFT values into motor speed and direction data. Try changing this scheme. For example, experiment with different frequency bands and criteria for changing motor directions.
2. In this project, you converted frequency information gathered from the audio signal into motor speed and direction settings. Try making the motors move according to the overall "pulse" or volume of the music instead. For this, you can compute the *root mean square (RMS)* value of the amplitude of the signal. This computation is similar to the FFT calculation. Once you read in a chunk of audio data and put it into a `numpy` array `x`, you can compute the RMS value as follows:

```
rms = np.sqrt(np.mean(x**2))
```

   Also, remember that the amplitude in your project was expressed as a 16-bit signed integer, which can

have a maximum value of 32,768 (a useful number to keep in mind for normalization). Use this RMS amplitude in conjunction with the FFT to generate a greater variation of laser patterns.

3. You know now that the frequency content, and hence the FFT of the audio data, changes in sync with the audio. Can you create a real-time visualization like the one shown in Figure 13-15, of both the audio data and the FFT as the audio plays through a speaker? This is intended to run on your computer, not your Raspberry Pi.

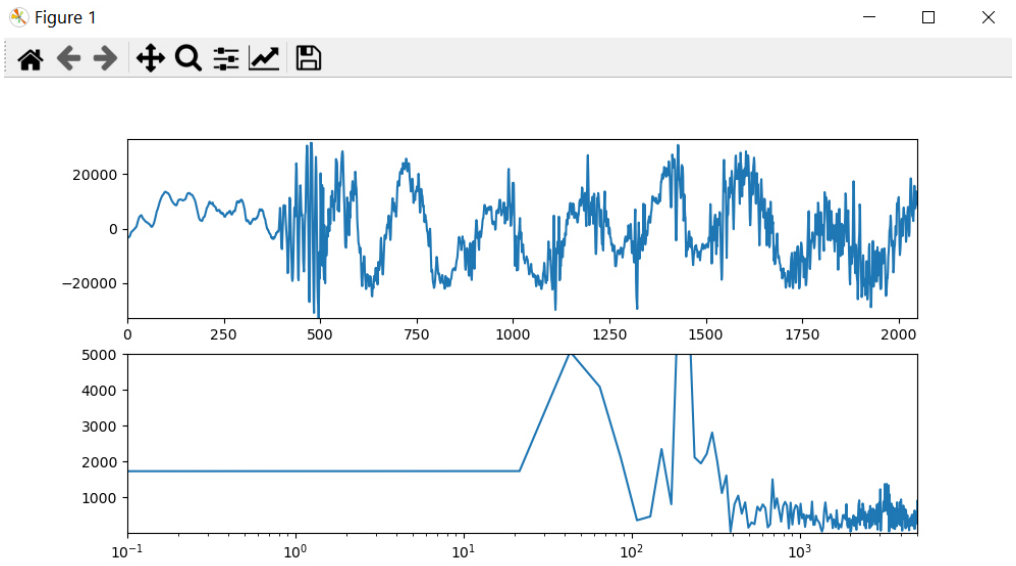


Figure 13-15: *A real-time FFT visualization*

Here are a few hints to solve this problem:

- Use `matplotlib` for plotting.
- Use Python's `multiprocessing` package so your music streaming output and plotting can happen simultaneously.
- Use the `numpy.fft.rfftfreq()` method to get the frequencies corresponding to the FFT values, for ease of plotting.

(The solution code for this experiment is in the book's GitHub repository, but give it a shot yourself first!)

The Complete Code

Here's the complete Python code for this project:

```
"""
laser_audio.py

Creates a laser display that changes in time to music.
Uses Python on a Raspberry Pi.

Author: Mahesh Venkitachalam
"""

import RPi.GPIO as GPIO
import time
import argparse
import pyaudio
import wave
import numpy as np

# define pin numbers
# uses TB6612FNG motor driver pin naming
PWMA = 12
PWMB = 13
AIN1 = 7
AIN2 = 8
BIN1 = 5
BIN2 = 6
STBY = 22
LASER = 25

# global PWM objects
pwm_a = None
pwm_b = None

# size of audio data read in
```

```
CHUNK = 2048
# FFT size
N = CHUNK

def init_pins():
    """set up pins"""
    global pwm_a, pwm_b

    # use BCM pin numbering
    GPIO.setmode(GPIO.BCM)

    # put pins into a list
    pins = [PWMA, PWMB, AIN1, AIN2, BIN1, BIN2, STBY, LASER]

    # set up pins as outputs
    GPIO.setup(pins, GPIO.OUT)

    # set PWM
    pwm_a = GPIO.PWM(PWMA, 100)
    pwm_b = GPIO.PWM(PWMB, 100)

def laser_on(on):
    """turn laser MOSFET on/off"""
    # pin 25 controls laser ctrl mosfet
    GPIO.output(LASER, on)

def test_motors():
    """test motors by manually setting speed and direction"""
    # turn laser on
    laser_on(True)

    # start motors
    start_motors()

    # read user input
    try:
        while True:
            print("Enter dca dcb dira dirb (eg. 50 100 1 0):")
            # read input
            str_in = input()
            # parse values
            vals = [int(val) for val in str_in.split()]
            # sanity check
            if len(vals) == 4:
```

```
                set_motor_speed_dir(vals[0], vals[1], vals[2],
  vals[3])
            else:
                print("Input error!")
    except:
        print("Exiting motor test!")
    finally:
        # stop motors
        stop_motors()
        # turn laser off
        laser_on(False)

def start_motors():
    """start both motors"""
    # enable driver chip
    GPIO.output(STBY, GPIO.HIGH)
    # set motor direction for channel A
    GPIO.output(AIN1, GPIO.HIGH)
    GPIO.output(AIN2, GPIO.LOW)
    # set motor direction for channel B
    GPIO.output(BIN1, GPIO.HIGH)
    GPIO.output(BIN2, GPIO.LOW)
    # set PWM for channel A
    duty_cycle = 0
    pwm_a.start(duty_cycle)
    # set PWM for channel B
    pwm_b.start(duty_cycle)

def stop_motors():
    """stop both motors"""
    # stop PWM
    pwm_a.stop()
    pwm_b.stop()
    # brake A
    GPIO.output(AIN1, GPIO.HIGH)
    GPIO.output(AIN2, GPIO.HIGH)
    # brake B
    GPIO.output(BIN1, GPIO.HIGH)
    GPIO.output(BIN2, GPIO.HIGH)
    # disable driver chip
```

```
    GPIO.output(STBY, GPIO.LOW)

def set_motor_speed_dir(dca, dcb, dira, dirb):
    """set speed and direction of motors"""
    # set duty cycle
    pwm_a.ChangeDutyCycle(dca)
    pwm_b.ChangeDutyCycle(dcb)
    # set direction A
    if dira:
        GPIO.output(AIN1, GPIO.HIGH)
        GPIO.output(AIN2, GPIO.LOW)
    else:
        GPIO.output(AIN1, GPIO.LOW)
        GPIO.output(AIN2, GPIO.HIGH)
    if dirb:
        GPIO.output(BIN1, GPIO.HIGH)
        GPIO.output(BIN2, GPIO.LOW)
    else:
        GPIO.output(BIN1, GPIO.LOW)
        GPIO.output(BIN2, GPIO.HIGH)

def process_audio(filename):
    """reads WAV file, does FFT and controls motors"""

    print("opening {}...".format(filename))

    # open WAV file
    wf = wave.open(filename, 'rb')

    # print audio details
    print("SW = {}, NCh = {}, SR =
  {}".format(wf.getsampwidth(),
        wf.getnchannels(), wf.getframerate()))

    # check for supported format
    if wf.getsampwidth() != 2 or wf.getnchannels() != 1:
        print("Only single channel 16 bit WAV files are
  supported!")
        wf.close()
        return

    # create PyAudio object
    p = pyaudio.PyAudio()
```

```
  # open an output stream
  stream =
p.open(format=p.get_format_from_width(wf.getsampwidth()),
                 channels=wf.getnchannels(),
                 rate=wf.getframerate(),
                 output=True)

  # read first frame
  data = wf.readframes(CHUNK)
  buf = np.frombuffer(data, dtype=np.int16)

  # store sample rate
  SR = wf.getframerate()

  # start motors
  start_motors()

  # laser on
  laser_on(True)

  # read audio data from WAV file
  try:
      # loop till there is no data to be read
      while len(data) > 0:
          # write stream to output
          stream.write(data)
          # ensure enough samples for FFT
          if len(buf) == N:
              buf = np.frombuffer(data, dtype=np.int16)
              # do FFT
              fft = np.fft.rfft(buf)
              fft = np.abs(fft) * 2.0/N
              # calc levels
              # get average of 3 frequency bands
              # 0-100 Hz, 100-1000 Hz, and 1000-2500 Hz
              levels = [np.sum(fft[0:100])/100,
                          np.sum(fft[100:1000])/900,
                          np.sum(fft[1000:2500])/1500]
              # speed1
              dca = int(5*levels[0])  percent 60
              # speed2
              dcb = int(100 + levels[1])  percent 60
```

```
                # dir
                dira = False
                dirb = True
                if levels[2] > 0.1:
                    dira = True
                # set motor direction and speed
                set_motor_speed_dir(dca, dcb, dira, dirb)
            # read next
            data = wf.readframes(CHUNK)

    except BaseException as err:
        print("Unexpected {}, type={}".format(err, type(err)))

    finally:
            print("Finally: Pyaudio clean up...")
            stream.stop_stream()
            stream.close()
            # stop motors
            stop_motors()
            # close WAV file
            wf.close()

def main():
    """main calling function"""

    # set up args parser
    parser = argparse.ArgumentParser(description="A laser audio
  display.")
    # add arguments
    parser.add_argument('--test_laser', action='store_true',
  required=False)
    parser.add_argument('--test_motors', action='store_true',
  required=False)
    parser.add_argument('--wav_file', dest='wav_file',
  required=False)
    args = parser.parse_args()

    # initialize pins
    init_pins()

    # main loop
    try:
        if args.test_laser:
```

```
            print("laser on...")
            laser_on(True)
            try:
                # wait in a loop
                while True:
                    time.sleep(1)
            except:
                # turn laser off
                laser_on(False)
        elif args.test_motors:
            print("testing motors...")
            test_motors()
        elif args.wav_file:
            print("starting laser audio display...")
            process_audio(args.wav_file)
    except (Exception) as e:
        print("Exception: {}".format(e))
        print("Exiting.")

    # turn laser off
    laser_on(False)

    # call at the end
    GPIO.cleanup()
    print("Done.")

# call main
if __name__ == '__main__':
    main()
```

14
IOT GARDEN



We live in an era where our phones talk to light bulbs and where toothbrushes want to access the internet. This is possible through the *Internet of Things (IoT)*, networks of everyday devices embedded with sensors that communicate with each other and the internet, usually in a wireless fashion. In this chapter, you'll build your own IoT sensor network to monitor the temperature and humidity conditions in your garden. The network will consist of one or more low-power devices running Python code and transmitting real-time sensor data wirelessly to a Raspberry Pi. The Pi will log the data and make it available over a local web server. You'll be able to view the sensor data through a web browser, as well as receive real-time alerts on your mobile device when extreme conditions occur.

Some of the concepts you'll learn about through this project are:

- Putting together a low-power IoT sensor network
- Understanding the basics of the Bluetooth Low Energy (BLE) protocol
- Building a BLE scanner on the Raspberry Pi
- Using a SQLite database to store sensor data

- Running a web server on the Pi using `Bottle`
- Using If This Then That (IFTTT) to send alerts to your mobile phone

How It Works

The IoT devices you'll use for this project are Adafruit BLE Sense boards, which have built-in temperature and humidity sensors. The devices periodically take measurements and transmit the sensor data wirelessly using Bluetooth Low Energy (BLE), which we'll discuss soon. This data is picked up by the Raspberry Pi running a BLE scanner. The Pi uses a database to store and retrieve the data, and it also runs a web server so it can display the data on a web page. Additionally, the Pi has logic to detect anomalies in the temperature and humidity data and to send an alert to a user's mobile device via the IFTTT service when such anomalies occur. Figure 14-1 summarizes the architecture of the project.

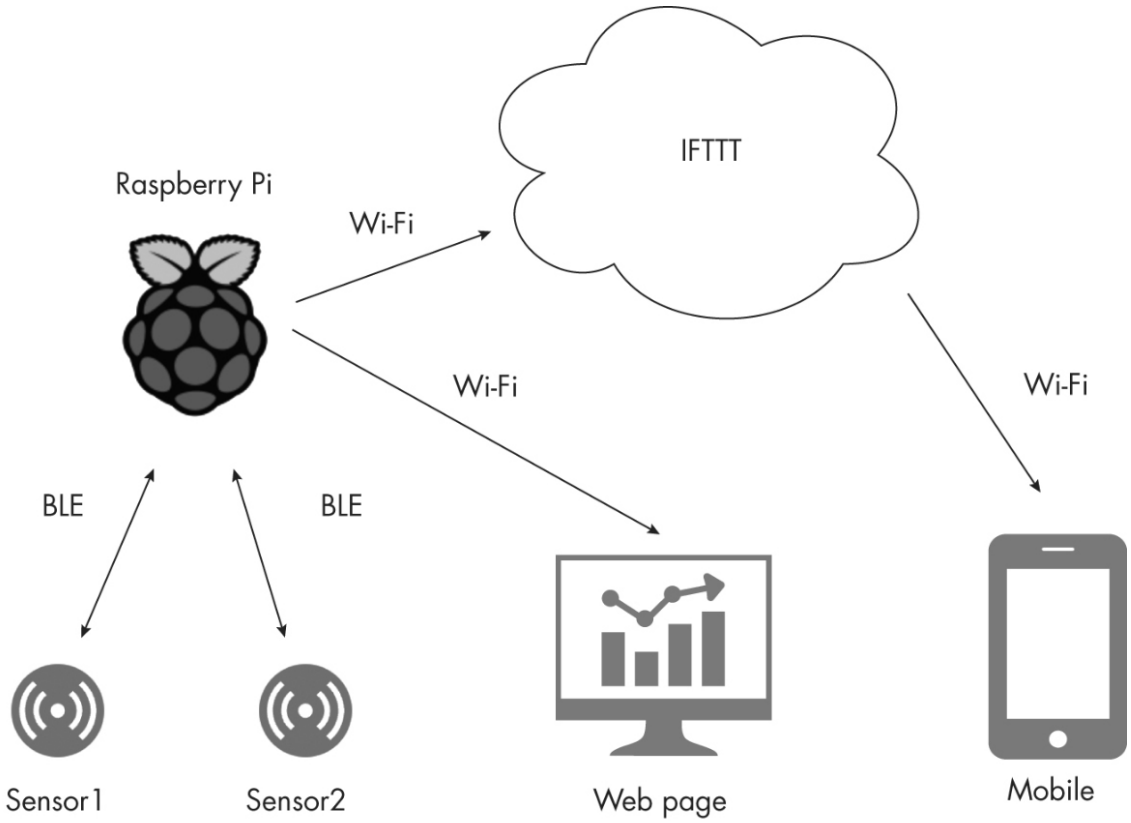


Figure 14-1: *The IoT garden system architecture*

You may wonder why you need the Raspberry Pi at all. Why can't the sensor devices talk directly to the internet instead of going through the Pi? The answer lies in power consumption. If the devices were talking directly to the internet via a protocol like Wi-Fi, they would typically consume more than 10 times the power compared to using BLE. This is important, because IoT devices are usually powered by batteries, and we expect them to last a very long time. The arrangement in Figure 14-1, where you have a network of low-power wireless devices talking to a gateway (the Raspberry Pi, in this case), is a common architecture in the world of IoT.

Another feature of this project's architecture is that your data remains in your hands—inside a database on your Pi, to be precise. You aren't blasting it over the internet, but rather confining it to your local network. This may not be

critical for basic garden data, but it's still good to know that an IoT device doesn't always *need* to send everything over the internet for it to be useful. Privacy and security are two good reasons for exposing your data to the internet only when it's really necessary. In this case, you'll still make use of the internet a little bit to send the IFTTT alerts.

Bluetooth Low Energy

BLE is a subset of the same wireless technology standard that enables Bluetooth headphones and speakers, but it's optimized for low-power, battery-operated devices. BLE is how your smartphone speaks to your smartwatch or your fitness tracker, for example. Devices that communicate over BLE can be categorized as either *central* or *peripheral*. Usually, central devices are more capable hardware such as laptops and phones, while peripheral devices are less capable hardware such as fitness bands and beacons. In this project, the Raspberry Pi is the central device, and the Adafruit BLE Sense boards are the peripherals.

NOTE *The distinction between central and peripheral devices isn't always clear-cut. Modern BLE chips allow the same piece of hardware to function as a central device, a peripheral, or a combination of both.*

A BLE peripheral makes its presence known via *advertisement packets*, as shown in Figure 14-2. These packets of data, which are typically sent out every few milliseconds, contain information such as the name of the peripheral, its transmission power, its manufacturer data, whether the central device can connect to it, and so on. The central device continuously scans for advertisement packets, and it can use information in the packets to establish communication with the peripheral.

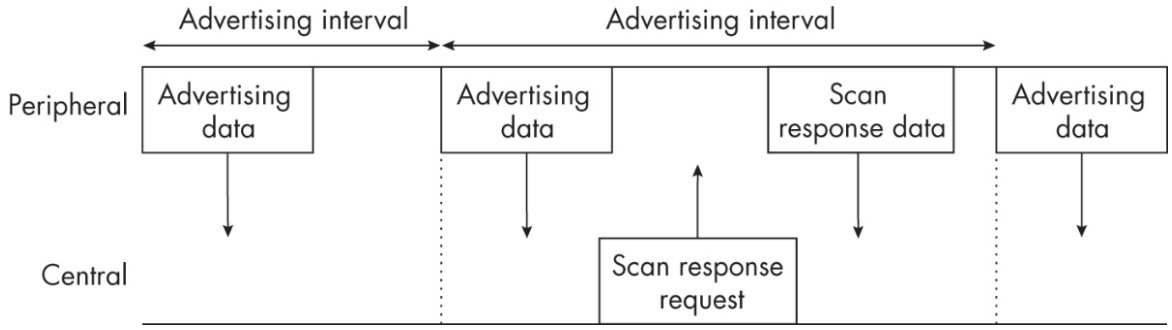


Figure 14-2: *The BLE advertising scheme*

The amount of data in an advertisement packet is limited to just 31 bytes to conserve the peripheral's battery, but the peripheral can optionally send an additional packet of information via a separate transmission called a *scan response*. The peripheral indicates whether a scan response is available as part of its normal advertisement packet. If the central device wants the extra data, it then sends a scan response request, which prompts the peripheral to take a break from sending advertisement packets and send the scan response instead. For this project, though, you need very little data from the sensors, so you can just put the temperature and humidity data directly in the advertisement packets.

On the Raspberry Pi side, you'll build a BLE scanner using BlueZ, the official Bluetooth protocol stack on Linux. Specifically, you'll make use of the following three command line programs:

hciconfig Resets BLE on the Pi during program initialization

hcitool Scans for BLE peripherals

hcidump Reads the advertisement data from the BLE peripherals

`hciconfig` and `hcitool` come as part of the Raspberry Pi OS installation, but you'll need to install `hcidump` from a terminal, as follows:

```
$ sudo apt-get install bluez-hcidump
```

Here’s what a typical command line session on the Raspberry Pi looks like with these tools. First, run `hcidump` in a shell to get ready to output data packets once the scan begins:

```
pi@iotsensors:~ $ sudo hcidump --raw
HCI sniffer - Bluetooth packet analyzer ver 5.50
device: hci0 snap_len: 1500 filter: 0xffffffff
```

This tells you that `hcidump` is waiting for BLE input. Next, run the `lescan` command with `hcitool` in a different shell to start scanning for BLE devices:

```
pi@iotsensors:~ $ sudo hcitool lescan
LE Scan...
DE:74:03:D9:3D:8B (unknown)
DE:74:03:D9:3D:8B IOTG1
36:D2:35:5A:BF:B0 (unknown)
8C:79:F5:8C:AE:DA (unknown)
5D:9F:EC:A0:09:51 (unknown)
5D:9F:EC:A0:09:51 (unknown)
60:80:0A:83:18:40 (unknown)
--snip--
```

This indicates that the scanner has detected a whole lot of BLE devices (they’re everywhere these days!). The moment you run the `lescan` command, `hcidump` starts printing out advertisement packet data, so your `hcidump` shell should now be full of messages like the following:

```
< 01 0B 20 07 01 10 00 10 00 00 00
> 04 0E 04 01 0B 20 00
< 01 0C 20 02 01 01
```

```
> 04 0E 04 01 0C 20 00
> 04 3E 1B 02 01 02 01 8B 3D D9 03 74 DE 0F 0E FF 22 08 0A 31
  FE 49 4F 54 47 31 1B 36 30 CB
> 04 3E 16 02 01 04 01 8B 3D D9 03 74 DE 0A 02 0A 00 06 09 49
  4F 54 47 31 CB
> 04 3E 23 02 01 03 01 03 58 0A 00 6A 35 17 16 FF 06 00 01 09
  21 0A 13 71 DA 7D 1A 00 52 6F 63 69 6E 61 6E 74 65 C5
> 04 3E 1F 02 01 03 01 B9 D4 AE 7E 01 0E 13 12 FF 06 00 01 09
  21 0A 9E 54 20 C5 51 48 6D 61 6E 64 6F BE
```

The messages are output as hexadecimal bytes (rather than human-readable text) because you started `hcidump` with the `--raw` option.

This example illustrated using the BlueZ tools manually at the command line. For the project, you'll instead execute the commands from inside the Python code running on your Pi. The Python code will also read the advertisement packets and get the sensor data.

The Bottle Web Framework

To monitor the sensor data via a web interface, you'll need to have the Pi run a web server. To do this, you'll use `Bottle`, a Python web framework with a simple interface. (In fact, the entire library consists of a single source file named *bottle.py*.) Here's the code needed to serve a simple web page from a Pi using `Bottle`:

```
from bottle import route, run

@route('/hello')
def hello():
  ❷ return "Hello Bottle World!"

run(host='iotgarden.local', port=8080, debug=True)
```

This code first defines a route to a URL or path (in this case, `/hello`) where the client can send data requests ❶, and it uses the `route()` method from `Bottle` as a Python decorator to bind to the `hello()` function, which will act as a handler for that route. This way, when the user navigates to the route, `Bottle` will call the `hello()` function, which returns a string ❷. The `run()` method ❸ starts the `Bottle` server, which can now accept connections from clients. Here we're assuming that the server is running on port 8080 on a Pi called `iotgarden`. Notice that the `debug` flag is set to `True` to make it easier to diagnose problems.

Run this code on your Wi-Fi–connected Pi, open a browser on any computer connected to the local network, and visit *http://<iotgarden>.local:8080/hello/*, substituting in your Pi's name as appropriate. `Bottle` should serve you a web page with the text "Hello Bottle World!" With just a few lines of code, you've created a web server.

PYTHON DECORATORS

A *decorator* in Python is `@` syntax that takes a function as an argument (such as `route()` in our `Bottle` example) and returns another function (such as `hello()`). A decorator provides a convenient way to "wrap" one function using another function. For example, this code

```
@wrapper
  def myFunc():
      return 'hi'
```

is equivalent to doing the following:

```
myFunc = wrapper(myFunc)
```

Functions are first-class objects in Python that can be passed like variables.

Note that you'll be using `Bottle` routing functions slightly differently in your project compared to this simple example, because you'll be binding the routes to class methods rather than free functions like `hello()` shown previously. There will be more on this later.

The SQLite Database

You need a place to store the sensor data so you can retrieve it in the future. You could write the data to a text file, but the retrieval process would quickly get cumbersome. Instead, you'll store the data with SQLite, a lightweight, easy-to-use database perfect for embedded systems like the Raspberry Pi. To access SQLite in Python, you'll use the `sqlite3` library.

SQLite databases are manipulated using SQL, a standard language for database systems. The SQL statements are written as strings in your Python code. You don't need to be a SQL expert to use SQLite for this project, however. You'll need only a few commands, which we'll discuss as they arise. To get a feel for how it works, let's look at a simple example of using SQLite in a Python interpreter session. First, here's how to create a database and add some entries to it:

```
>>> import sqlite3
>>> con = sqlite3.connect('test.db') ❶
>>> cur = con.cursor() ❷
>>> cur.execute("CREATE TABLE sensor_data (TS datetime, ID
    text, VAL numeric)") ❸
>>> for i in range(10):
```

```
...   cur.execute("INSERT into sensor_data VALUES
     (datetime('now'),'ABC', ?)", (i, )) ❹
>>> con.commit() ❺
>>> con.close()
>>> exit()
```

Here you call the `sqlite3.connect()` method with the name of the database (`test.db` in this case) ❶. This method either returns a connection to an existing database or creates a new one if the database with the given name doesn't exist. Then you create a *cursor* using the connection object ❷. This is a construct that lets you interact with the database to create tables, make new entries, and retrieve data. You use the cursor to execute a SQL statement that creates a database table called `sensor_data` with the following columns: `TS` (timestamp), which is of the type `datetime`; `ID` of type `text`; and `VAL` of type `numeric` ❸. Next, you add 10 entries to this database by executing SQL `INSERT` statements in a `for` loop. Each statement adds the entry with the current timestamp, the string `'ABC'`, and the loop index `i` ❹. The `?` is a formatting placeholder used by SQLite, and the actual values are specified using a tuple. Finally, you commit the changes to the database to make them permanent ❺, before closing the database connection.

Now let's retrieve some values from the database:

```
>>> con = sqlite3.connect('test.db')
>>> cur = con.cursor()
>>> cur.execute("SELECT * FROM sensor_data WHERE VAL > 5")
>>> print(cur.fetchall())
[('2021-10-16 13:01:22', 'ABC', 6), ('2021-10-16 13:01:22',
  'ABC', 7),
 ('2021-10-16 13:01:22', 'ABC', 8), ('2021-10-16 13:01:22',
  'ABC', 9)]
```

Once again, you establish a connection to the database and create a cursor to interact with it. Then you execute a `SELECT` SQL query to retrieve some data ❶. In this query, you ask for all rows (`SELECT *`) from the `sensor_data` table for which the entry in the `VAL` column is greater than `5`. You print the results of the query, which are accessible through the cursor's `fetchall()` method.

Requirements

On the Raspberry Pi, you'll need the `bottle` module to create a web server, the `sqlite3` module to work with a SQLite database, and `matplotlib` to plot the sensor data. The BLE Sense boards don't have enough computing power to run a full version of Python, so instead you'll program them using CircuitPython, an open source derivative of MicroPython maintained by Adafruit. We're using CircuitPython for this project, rather than MicroPython as you saw in Chapter 12, since the former has more library support for the Adafruit-made devices.

You'll also need the following hardware for this project:

- One or more Adafruit Feather Bluefruit nRF52840 Sense boards, as per your need
- One Raspberry Pi 3B+ (or newer) board with SD card and power supply adapter
- One 3.7 V LiPo battery or USB power supply per BLE Sense board

Figure 14-3 shows the required hardware.

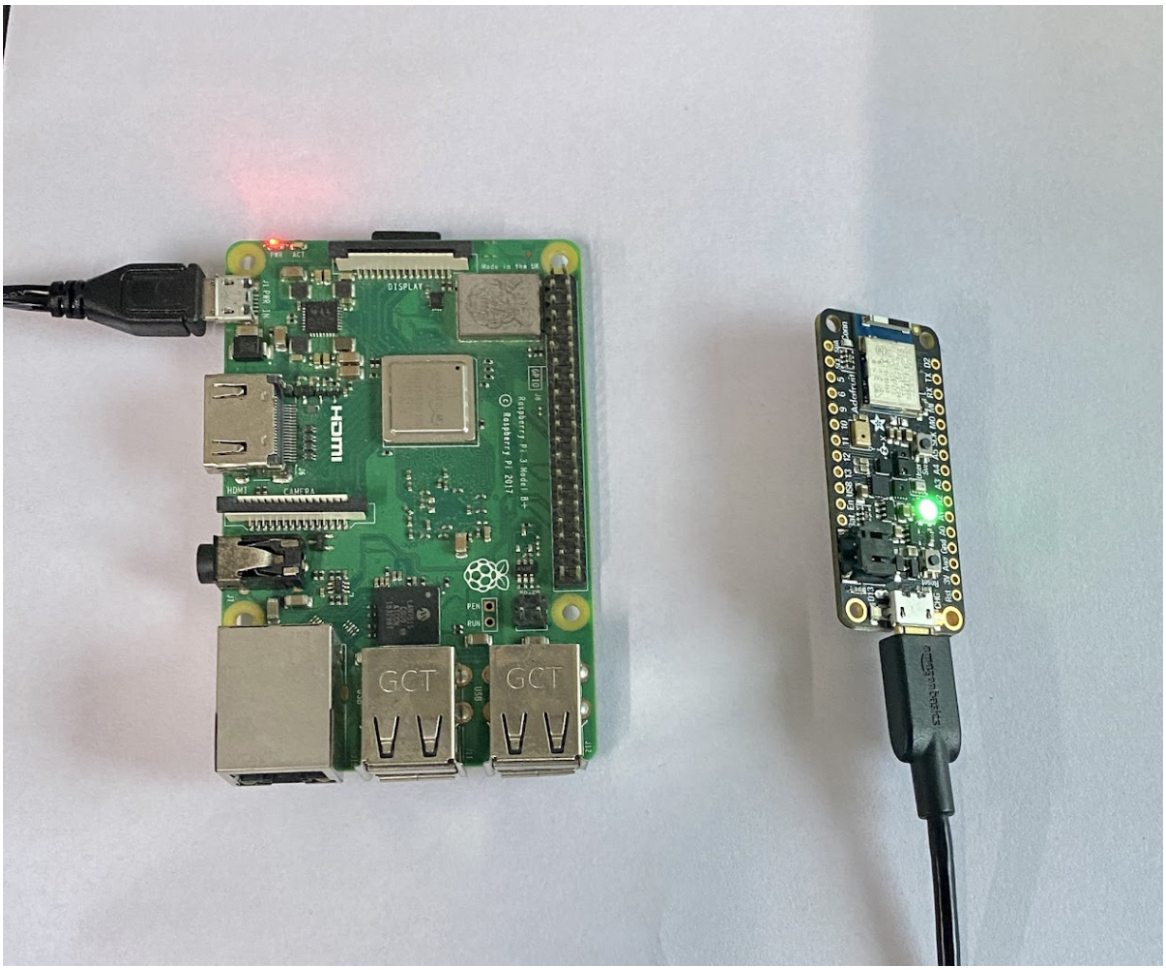


Figure 14-3: *The hardware required for the project*

You can hook up your Raspberry Pi indoors, close to your garden, and place your BLE Sense boards with suitable power supply units and protective enclosures in the garden.

Raspberry Pi Setup

To start this project, you'll need to set up your Raspberry Pi. Please follow the instructions in Appendix B. The project code that follows assumes that you've named you Pi `iotgarden`, which allows you to access it on the network as `iotgarden.local`.

CircuitPython Setup

To install CircuitPython, follow these steps for each of your Adafruit BLE Sense boards:

1. Visit *https://circuitpython.org/downloads*, search for your Bluefruit Sense board, and download the board's CircuitPython installer file, which has a *.uf2* extension. Take note of the CircuitPython version number you're downloading.
2. Connect the Adafuit board to a USB port on your computer and double-click the Reset button on the board. The LED on the board should turn green, and you should see a new drive appear on your computer called FTHRSNSBOOT.
3. Drag the *.uf2* file into the FTHRSNSBOOT drive. Once the file is copied, the LED on the board will flash, and a new drive called CIRCUITPY will appear on your computer.

Next, you need to install the required Adafruit libraries on your board. Here's how:

1. Visit *https://circuitpython.org/libraries* and download the *.zip* file with the library bundle corresponding to your version of CircuitPython.
2. Unzip the downloaded file and copy the following files/folders into a folder called *lib* inside the CIRCUITPY drive. (Create the *lib* folder if it doesn't exist.)
    - *adafruit_apds9960*
    - *adafruit_ble*
    - *adafruit_bme280*

- *adafruit_bmp280.mpy*
- *adafruit_bus_device*
- *adafruit_lis3mdl.mpy*
- *adafruit_lsm6ds*
- *adafruit_register*
- *adafruit_sht31d.mpy*
- *neopixel.mpy*

3. Press Reset on the board and you're all set to use the board for this project.

By default, CircuitPython will run code from any file in CIRCUITPY named *code.py*. You'll need to copy to the drive the *ble_sensors.py* file discussed in the following section and rename it *code.py* to run the project.

If This Then That Setup

IFTTT is a web service that lets you create automated responses to specific actions. You'll use IFTTT to send alerts to your mobile phone when something is really off with the temperature or humidity levels your sensors are picking up. Follow these steps to get set up to receive IFTTT alerts:

1. Visit the IFTTT website (*https://ifttt.com*) and sign up for an account.
2. Download the IFTTT app to your smartphone and set it up.
3. While signed into your IFTTT account in your browser, click **Create**. Then click the **Add** button in the If This box.
4. Search for and select **Webhooks** on the Choose a Service page that comes up. Then select **Receive a**

Web Request with a JSON Payload.

5. Under Event Name, enter **TH_alert** (note the capitalization) and press **Create Trigger**.
6. You should now be back on the Create page. Click the **Add** button in the Then That box.
7. Search for and select **Notifications**. Then click **Send a Notification from the IFTTT App**.
8. In the page that comes up, add the text “T/H Alert!” to the Message box. Then click **Add Ingredient** and select **OccuredAt**. Click **Add Ingredient** again and select **JsonPayload**.
9. Click the **Create Action** button to return to the Create screen. Then click **Continue** and **Finish** to finalize the alert.

You’ll need your IFTTT key to trigger alerts from your Python code. To look it up, follow these steps:

1. On the IFTTT website, click the round account icon in the top-right corner of the screen and select **My Services**.
2. Click the **Webhooks** link, and then click the **Documentation** button.
3. A page should load with your key written across the top. Take note of the key. You’ll also find information on this page about how to send a test alert to your smartphone. Be sure to fill in **TH_alert** as the event name if you run a test.

The Code

The code for this project is spread over these Python files:

ble_sensors.py The CircuitPython code that runs in the Adafruit BLE Sense boards. It reads from the temperature and humidity sensors and puts the data in BLE advertisement packets.

BLEScanner.py Implements the BLE scanner on the Raspberry Pi, using BlueZ tools to read advertisement data. This code also sends the IFTTT alerts.

server.py Implements the `Bottle` web server on the Raspberry Pi.

iotgarden.py The main program file. This code sets up the SQLite database and starts the BLE scanner and the web server.

In addition to the Python files, the project has a subfolder called *static* with some extra files used by the `Bottle` web server:

static/style.css The style sheet for the HTML returned by the server

static/server.js The JavaScript code returned by the server that fetches sensor data

The full code for the project is available at *https://github.com/mkvenkit/pp2e/tree/main/iotgarden*.

The CircuitPython Code

The CircuitPython code running on the BLE Sense board has a straightforward purpose: it reads data from the built-in temperature and humidity sensors and puts that data in the board's BLE advertisement packets. This deceptively simple task requires you to import a surprising number of modules. To see the complete code listing, skip ahead to "The Complete CircuitPython Code" on page 343. The code is also available at *https://github.com/mkvenkit/pp2e/blob/main/iotgarden/ble_sensors/ble_sensors.py*.

```
import time, struct
import board
import adafruit_bmp280
import adafruit_sht31d
from adafruit_ble import BLERadio
from adafruit_ble.advertising import Advertisement,
LazyObjectField
from adafruit_ble.advertising.standard import ManufacturerData,
                                              ManufacturerDataF
  ield
import _bleio
import neopixel
```

You import Python's built-in `time` and `struct` modules for sleeping and packing data, respectively. The `board` module gives you access to the `I2C` library, which allows the BLE chip on the board to communicate with the sensors using the I^2C protocol (pronounced "eye-squared-C"). The `adafruit_bmp280` and `adafruit_sht31d` modules are required for communicating with the sensors, and the `BLERadio` class is for enabling BLE and sending advertisement packets. The imports at ❶ and ❷ give you access to the Adafruit BLE advertising modules necessary for creating your own custom advertisement packets featuring the sensor data. Additionally, you'll use the `_bleio` module to get the MAC address of the device and `neopixel` to control the board's LED.

Preparing BLE Packets

Next, define a class called `IOTGAdvertisement` to help create the BLE advertisement packets. The `adafruit_ble` library already has an `Advertisement` class that handles BLE advertisements. You create `IOTGAdvertisement` as a subclass of `Advertisement` to use the parent class's features while adding your own customization:

```
class IOTGAdvertisement(Advertisement):
    flags = None
  ❶ match_prefixes = (
        struct.pack(
            "<BHBH",  # prefix format
            0xFF,     # 0xFF is "Manufacturer Specific Data" as
  per BLE spec
            0x0822,   # 2-byte company ID
            struct.calcsize("<H9s"), # data format
            0xabcd # our ID
        ), # comma required - a tuple is expected
    )
  ❷ manufacturer_data = LazyObjectField(
        ManufacturerData,
        "manufacturer_data",
        advertising_data_type=0xFF, # 0xFF is "Manufacturer
  Specific Data"
                                    # as per BLE spec
        company_id=0x0822,          # 2-byte company ID
        key_encoding="<H",
    )
    # set manufacturer data field
  ❸ md_field = ManufacturerDataField(0xabcd, "<9s")
```

The BLE standard is very particular, so this code may look intricate, but all it's really doing is putting some custom data in the advertisement packet. First you fill a tuple called `match_prefixes` ❶, which the `adafruit_ble` library will use to manage various fields in the advertisement packet. The tuple has only one element, a packed structure of bytes that you create using the Python `struct` module. Next, you define the `manufacturer_data` field ❷, which will use the format described at ❶. The manufacturer data field is a standard part of a BLE advertisement packet that has some space for whatever custom data the manufacturer (or the user) wants to include. Finally, you create a custom `ManufacturerDataField`

object ❸ , which you'll keep updating as sensor values change.

Reading and Sending Data

The `main()` function of the CircuitPython program reads and sends the sensor data. The function begins with some initializations:

```
def main():
    # initialize I2C
  ❶ i2c = board.I2C()

    # initialize sensors
  ❷ bmp280 = adafruit_bmp280.Adafruit_BMP280_I2C(i2c)
  ❸ sht31d = adafruit_sht31d.SHT31D(i2c)

    # initialize BLE
  ❹ ble = BLERadio()

    # create custom advertisement object
  ❺ advertisement = IOTGAdvertisement()
    # append first 2 hex bytes (4 characters) of MAC address to
  name
  ❻ addr_bytes = _bleio.adapter.address.address_bytes
    name = "{0:02x}{1:02x}".format(addr_bytes[5],
  addr_bytes[4]).upper()
    # set device name
  ❼ ble.name = "IG" + name
```

First you initialize the `I2C` module ❶ so the BLE chip can communicate with the sensors. Then you initialize the modules for the temperature (`bmp280`) ❷ and humidity (`sht31d`) ❸ sensors. You also initialize the BLE radio ❹, which is required for transmitting the advertisement packets, and create an instance of your custom `IOTGAdvertisement` class ❺.

Next, you set the name of the BLE device to the string `IG` (for IoT Garden) followed by the first four hexadecimal digits

(or two bytes) of the device's MAC address ❼. For example, if the MAC address of the device is `de:74:03:d9:3d:8b`, the name of the device will be set to `IGDE74`. To do this, you first get the MAC address as bytes ❻. The bytes are in reverse order of the string representation, however—in our example MAC address, for instance, the first byte would be `0x8b`. What you're looking for are the first two bytes, `0xde` and `0x74`, which are at indices `5` and `4` in `address_bytes`, respectively. You use string formatting to convert these bytes to string representation and convert them to uppercase using `upper()`.

Now let's look at the rest of the initialization:

```
  # set initial value
  # will use only first 5 chars of name
❶ advertisement.md_field = ble.name[:5] + "0000"
  # BLE advertising interval in seconds
  BLE_ADV_INT = 0.2
  # start BLE advertising
❷ ble.start_advertising(advertisement, interval=BLE_ADV_INT)

  # set up NeoPixels and turn them all off
   pixels = neopixel.NeoPixel(board.NEOPIXEL, 1,
                              brightness=0.1,
auto_write=False)
```

Here you set an initial value for your custom manufacturer data ❶. For this, you concatenate the first five characters of the device name followed by four bytes of zeros. You'll update the first two bytes with the sensor data. As for the remaining two bytes, there's an exercise waiting for you in "Experiments!" on page 343 where you can put them to use.

Next, you set the BLE advertisement interval (`BLE_ADV_INT`) to `0.2`, meaning the device will send out an advertisement packet every 0.2 seconds. Then you call the method to start sending advertisement packets ❷, passing your custom

advertisement class and the time interval as arguments. You also initialize the `neopixel` library to control the LED on the board. The `board.NEOPIXEL` argument sets the pin number for the neopixel LED, and `1` represents the number of LEDs on the board. The brightness setting is `0.1` (the maximum being `1.0`), and setting the `auto_write` flag to `False` means you'll need to call the `show()` method explicitly for the values to take. (You'll see this in action soon.)

The `main()` function continues with a loop that reads the sensor data and updates the BLE packets:

```
    # main loop
    while True:
        # print values - this will be available on serial
      ❶ print("Temperature: {:.1f}
C".format(bmp280.temperature))
        print("Humidity: {:.1f}
%".format(sht31d.relative_humidity))
        # get sensor data
      ❷ T = int(bmp280.temperature)+ 40
        H = int(sht31d.relative_humidity)
        # stop advertising
      ❸ ble.stop_advertising()
        # update advertisement data
      ❹ advertisement.md_field = ble.name[:5] + chr(T) +
chr(H) + "00"
        # start advertising
      ❺ ble.start_advertising(advertisement,
interval=BLE_ADV_INT)
        # blink neopixel LED
        pixels.fill((255, 255, 0))
        pixels.show()
        time.sleep(0.1)
        pixels.fill((0, 0, 0))
        pixels.show()
        # sleep for 2 seconds
      ❻ time.sleep(2)
```

You begin the loop by printing out the values read by the sensors ❶. You can use this output to confirm that your sensors are putting out reasonable values. To see the values, connect the board via USB to your computer and use a serial terminal application such as CoolTerm, as explained in Chapter 12. The output should look something like this:

```
Temperature: 26.7 C
Humidity: 55.6 %
```

Next, you read the temperature and humidity values from the sensors ❷, converting both to integers since you have only one byte to represent each value. You add 40 to the temperature to accommodate negative values. The BMP280 temperature sensor on the board has a range of −40°C to 85°C, so adding 40 converts this range to [0, 125]. You'll be back to the correct range on the Raspberry Pi once the data is parsed from the BLE advertisement data.

You have to stop BLE advertising ❸ so you can change the data in the packets. Then you set the manufacturer data field with the first five characters of the device name, followed by one byte each of temperature and humidity values ❹. You're using `chr()` here to encode each 1-byte value into a character. You also set the last two bytes in the data field to be zeros. Now that you've updated the sensor values in the packet, you restart advertising ❺. This way, the scanner on the Pi will pick up the new sensor values from the BLE board.

To provide a visual indicator that the device is alive, you blink its neopixel LED by turning it on for 0.1 second. The `fill()` method sets a color using an (R, G, B) tuple, and `show()` sets the value to the LED. Finally, you add a two-second delay before restarting the loop and checking the sensors again ❻. During that delay, the board will continue

sending out the same advertisement packet every 0.2 seconds, as per its advertisement interval.

NOTE *When you've tested the code and are ready to deploy your IoT device, I recommend commenting out the* `print()` *statements and the neopixel code to conserve power. Remember, BLE is all about low energy!*

The BLE Scanner Code

The code to have your Raspberry Pi listen for and process sensor data over BLE is encapsulated in a class called `BLEScanner`. To see the complete code listing, skip ahead to "The Complete BLE Scanner Code" on page 345. You'll also find this code in the book's GitHub repository at *https://github.com/mkvenkit/pp2e/blob/main/iotgarden/BLEScanner.py*.

Here's the constructor for this class:

```
class BLEScanner:
    def __init__(self, dbname):
        """BLEScanner constructor"""
        self.T = 0
        self.H = 0
        # max values
        self.TMAX = 30
        self.HMIN = 20
        # timestamp for last alert
      ❶ self.last_alert = time.time()
        # alert interval in seconds
      ❷ self.ALERT_INT = 60
        # scan interval in seconds
      ❸ self.SCAN_INT = 10
      ❹ self._dbname = dbname
      ❺ self.hcitool = None
        self.hcidump = None
```

```
❻ self.task = None
  # ------------------------------------------------
  # peripheral allow list - add your devices here!
  # ------------------------------------------------
❼ self.allowlist = ["DE:74:03:D9:3D:8B"]
```

You start by defining instance variables `T` and `H` to keep track of the latest temperature and humidity values read from the sensors. Then you set threshold values for triggering the automated alerts. If the temperature goes above `TMAX` or the humidity goes below `HMIN`, the program will issue an alert using IFTTT. You create the `last_alert` variable at ❶ to store the time when the most recent alert was sent, and at ❷ you set the minimum interval between alerts. This is so you don't keep sending yourself alerts continuously when an alert condition is met. At ❸, you set the scan interval in seconds to control how often the Pi will scan for BLE devices. Next, you save the name of the SQLite database that was passed into the constructor ❹. You need this to save values from the sensors.

At ❺ and the following line, you create a couple of instance variables to store process IDs for the `hcitool` and `hcidump` programs, which will run later. At ❻, you create a `task` instance variable. Later, you'll be creating a separate thread to run this task, which will be doing the scanning, while the main thread of the program runs the web server. Finally, at ❼ you create a list of the BLE devices you want to listen to. When you run the BLE scanner, you're likely to pick up many BLE peripherals, not just the Adafruit Sense boards. Keeping a list with your sensor device IDs makes the BLE data easier to parse. Later in the chapter, I'll show you how to look up the device IDs so you can add them here to your allowlist.

Working with the BlueZ Tools

As we discussed in “Bluetooth Low Energy” on page 313, this project uses BlueZ, Linux’s official Bluetooth protocol stack, to scan for BLE data. The `BLEScanner` class needs methods for working with these tools. First we’ll examine the `start_scan()` method, which sets up the BlueZ tools for BLE scanning.

```
def start_scan(self):
    """starts the BlueZ tools required for scanning"""
    print("BLE scan started...")
    # reset device
  ❶ ret = subprocess.run(['sudo', '-n', 'hciconfig', 'hci0',
  'reset'],
                           stdout=subprocess.DEVNULL)
    print(ret)

    # start hcitool process
  ❷ self.hcitool = subprocess.Popen(['sudo', '-n', 'hcitool',
                                     'lescan', '--duplicates'],
                                    stdout=subprocess.DEVNULL)
    # start hcidump process
  ❸ self.hcidump = subprocess.Popen(['sudo', '-n', 'hcidump',
  '--raw'],
                                    stdout=subprocess.PIPE)
```

First you reset the BLE device of the Raspberry Pi using the `hciconfig` tool ❶. You use the Python `subprocess` module to run this process. The `subprocess.run()` method takes the process arguments as a list, so this call executes the command `sudo -n hciconfig hci0 reset`. The output of this process `stdout` is set to `DEVNULL`, which just means you don’t care about messages printed out by this command. (The `-n` flag in `sudo` makes it noninteractive.) You next use a different `subprocess` method called `Popen()` to run the command `sudo -n hcitool lescan --duplicates` ❷. This process scans for BLE peripheral devices. The `--duplicates` flag ensures that the same device can come up in the scan list repeatedly. You

need this, since the sensor data in the advertisement packets keeps changing, and you need the latest values.

NOTE *The difference between `subprocess.run()` and `subprocess.Popen()` is that the former waits for the process to complete, whereas the latter returns immediately while the process runs in the background.*

Finally, you use `subprocess.Popen()` to run another command: `sudo -n hcidump --raw` ❸. As we discussed earlier in the chapter, this command intercepts and prints out the advertisement data as hexadecimal bytes. Notice that `stdout` is set to `subprocess.PIPE` in this case. This means you can read the output from this process similar to how you read the contents of a file. More on this in "Parsing the Data" below.

Now let's look at the `stop_scan()` method, which kills the processes begun in the `start_scan()` method when you're ready to stop scanning for BLE packets.

```
def stop_scan(self):
    """stops BLE scan by killing BlueZ tools processes"""
    subprocess.run(['sudo', 'kill', str(self.hcidump.pid), '-
  s', 'SIGINT'])
    subprocess.run(['sudo', '-n', 'kill',
  str(self.hcitool.pid),
                    '-s', "SIGINT"])
    print("BLE scan stopped.")
```

Here you kill off the `hcidump` and `hcitool` processes using `pid`, their process IDs. The command `sudo -n kill pid -s SIGINT` kills a process with the given `pid` by sending it the `SIGINT` interrupt signal.

Parsing the Data

The scanner needs methods for parsing the BLE data it receives. First we'll consider the `parse_hcidump()` method, which parses the output from the `hcidump` process:

```
def parse_hcidump(self):
    data = ""
    (macid, name, T, H) = (None, None, None, None)
    while True:
      ❶ line = self.hcidump.stdout.readline()
      ❷ line = line.decode()
        if line.startswith('> '):
            data = line[2:]
        elif line.startswith('< '):
            data = ""
        else:
            if data:
              ❸ data += line
              ❹ data = " ".join(data.split())
              ❺ fields = self.parse_data(data)
                    success = False
                  ❻ try:
                        macid = fields["macid"]
                        T = fields["T"]
                        H = fields["H"]
                        name = fields["name"]
                        success = True
                    except KeyError:
                        # skip this error, since this indicates
                        # invalid data
                      ❼ pass
                    if success:
                      ❽ return (macid, name, T, H)
```

Within a `while` loop, you start reading one line at a time from `self.hcidump.stdout` using the `readline()` method ❶ , much like you'd read lines from a file. To understand the code that follows, it helps to know a bit about the data

being parsed. This is what a typical output from `hcidump` looks like:

```
> 04 3E 1B 02 01 02 01 8B 3D D9 03 74 DE 0F 0E FF 22 08 0A 31
  FE 49 4F 54 47 31 1B 36 30 CB
```

The output is split across two lines, and it starts with a `>` character. You want to take these two lines and combine them to get a single string such as `"04 3E 1B 02 01 02 01 8B 3D D9 03 74 DE 0F 0E FF 22 08 0A 31 FE 49 4F 54 47 31 1B 36 30 CB"`. To do this, you convert the bytes output from `readline()` to a string using the `decode()` method ❷. Then you use some logic to build up the final string that you want. If the line starts with `>`, you know it's the first of the set of two lines making up an `hcidump` entry, so you store the line and go on to the next one. Lines that start with `<` are ignored. If a line starts with neither `>` nor `<`, then it's the second line of the advertisement, and you join the lines together ❸.

The resulting data will have newline characters in the middle and the end. You get rid of those using a combination of the `split()` and `join()` string methods ❹. This example illustrates how the scheme works:

```
>>> x = "ab cd\n ef\tff\r\n"
>>> x.split()
['ab', 'cd', 'ef', 'ff']
>>> " ".join(x.split())
'ab cd ef ff'
```

Notice from the first line of output how the `split()` method automatically splits up the string at the whitespace characters, producing a list of substrings and removing the whitespace characters in the process. This gets rid of the unwanted newline characters, but it also gets rid of the spaces, which you want to keep. That's where the `join()`

method comes in. It merges the list items back into a single string, with a space between each substring, as you can see in the second output line.

Returning to the `parse_hcidump()` method, you now have a complete BLE advertisement packet stored as a string in variable `data`. You call the `parse_data()` method on this string to get the device details in the form of a `fields` dictionary ❺. We'll look at this method soon. Then you retrieve the MAC ID, name, temperature, and humidity values from the dictionary. This code is enclosed in a `try` block ❻ in case the data isn't what you expect. In that case, an exception will be thrown, and you skip that advertisement packet by calling `pass` ❼. If you successfully retrieve all the values, they're returned as a tuple ❽.

Now let's take a look at the `parse_data()` method you used in `parse_hcidump()` to build the `fields` dictionary:

```
def parse_data(self, data):
    fields = {}
    # parse MACID
  ❶ x = [int(val, 16) for val in data.split()]
  ❷ macid = ":".join([format(val, '02x').upper() for val in
  x[7:13][::-1]])
    # check if MACID is in allowlist
  ❸ if macid in self.allowlist:
        # look at 6th byte to see PDU type
      ❹ if (x[5] == 0x02): # ADV_IND
          ❺ fields["macid"] = macid
            # parse data
          ❻ fields["T"] = x[26]
            fields["H"] = x[27]
          ❼ name = "".join([format(val, '02x').upper() for val
  in x[21:26]])
          ❽ name = bytearray.fromhex(name).decode()
            fields["name"] = name
    return fields
```

You start by defining an empty dictionary `fields`, where you'll store the parsed data. Then you split the data into a list of hexadecimal values ❶ and extract the MAC ID of the peripheral that sent the advertisement packet ❷. Here's a sample run of these statements to illustrate how they work:

```
>>> data = "04 3E 1C 02 01 02 01 8B 3D D9 03 74 DE 10 0F FF 22
           08 0B 31 FE 49 4F 54 47 31 61 62 63 64 BD"
>>> x = [int(val, 16) for val in data.split()]
>>> x
[4, 62, 28, 2, 1, 2, 1, 139, 61, 217, 3, 116, 222, 16, 15, 255,
  34,
 8, 11, 49, 254, 73, 79, 84, 71, 49, 97, 98, 99, 100, 189]
>>> x[7:13][::-1]
[222, 116, 3, 217, 61, 139]
>>> [format(val, '02x').upper() for val in x[7:13][::-1]]
['DE', '74', '03', 'D9', '3D', '8B']
>>> ":".join([format(val, '02x').upper() for val in x[7:13]
  [::-1]])
'DE:74:03:D9:3D:8B'
```

Notice how the data is first split into a list of decimal numbers (`x`). Then you use a list comprehension and `format()` to create two-character string representations of the numbers, taking just bytes 7 through 12 to extract the MAC ID. The `[::-1]` reverses the MAC ID, since it comes in the opposite order in the packet data. Finally, the `join()` method merges the bytes making up the MAC ID into a single string, using colons as separators.

Continuing with `parse_data()`, you check if the extracted MAC ID matches a device on your allowlist ❸. If not, you ignore the data. Then you check the fifth byte in the data to ensure that the packet type is `ADV_IND` ❹. This ensures that the data is a regular advertisement packet, and not a scan response. Next, you store the MAC ID in the `fields` dictionary ❺, along with the temperature and humidity values ❻,

which you read from the appropriate indices in the data list. You also read the five-character device name you put in the BLE packet ❼, similar to how you read the MAC ID. Then you call `decode()` to convert the bytes to a string ❽. Finally, you return the `fields` dictionary to the caller.

Sending Alerts

Once you've parsed the data from a BLE advertisement packet, you need to send an IFTTT alert if the sensor readings are concerning. Define a `send_alert()` method for this purpose:

```
def send_alert(self):
    # check T, H
  ❶ delta = time.time() - self.last_alert
  ❷ if ((self.T > self.TMAX) or (self.H < self.HMIN)) and
                                        (delta > self.ALERT_INT):
        print("Triggering IFTTT alert!")
      ❸ key = 'ABCDEF' # USE YOUR KEY HERE!
      ❹ url =
  'https://maker.ifttt.com/trigger/TH_alert/json/with/key/' +
  key
        json_data = {"T": self.T, "H": self.H}
      ❺ r = requests.post(url, data = json_data)
        # save last alert
        self.last_alert = time.time()
```

First you compute how much time has elapsed since the last time an IFTTT alert was sent ❶. Then you check your alert conditions ❷. An alert will be triggered either if the current temperature is above `TMAX` or if the humidity has fallen below `HMIN`, provided a sufficient amount of time has passed since the last alert. Since the goal is to monitor the health of your garden, checking if it's too hot or too dry makes sense, but you're welcome to modify this check with

your own alert criteria. You next put together the IFTTT Webhooks URL ❹, using your user key set at ❸. (Make sure your key matches the one you obtained in “If This Then That Setup” on page 319.) Then you set up a simple JSON string with the sensor data and post it to the IFTTT URL ❺. You finish by updating the `last_alert` instance variable with the current time, for future use.

Conducting a Scan

The `scan_task()` method coordinates all the activity required to conduct a BLE scan. This method also stores the scanned data to the SQLite database. Here’s the method definition:

```
def scan_task(self):
    """the scanning task which is run on a separate thread"""
    # start BLE scan
  ❶ self.start_scan()
    # get data
  ❷ (macid, name, self.T, self.H) = self.parse_hcidump()
    # correct temperature offset
    self.T = self.T - 40
    print(self.T, self.H)
    # stop BLE scan
  ❸ self.stop_scan()

    # send alert if required
  ❹ self.send_alert()
    # write to db
    # connect to database
    con = sqlite3.connect(self._dbname)
    cur = con.cursor()
    devID = macid
    # add data
    with con:
      ❺ cur.execute("INSERT INTO iotgarden_data VALUES (?, ?,
 ?, ?, ?)",
                  (devID, name, datetime.now(), self.T, self.H))
```

```
    # commit changes
    con.commit()
    # close db
    con.close()
    # schedule the next task
❻ self.task = Timer(self.SCAN_INT, self.scan_task)
❼ self.task.start()
```

This method harnesses methods you've already defined. First you start the BlueZ tools by calling `start_scan()` ❶. Then you call `parse_hcidump()` to parse the advertisement data ❷, storing the retrieved values in a tuple. Right after that, you correct for the temperature offset added in the IoT device by subtracting `40`. (Recall that this offset was to accommodate negative temperature values.) Calling `stop_scan()` stops the BLE scanning ❸, and calling `send_alert()` sends an IFTTT alert if required ❹.

Next, you establish a connection to your SQLite database, allowing you to insert a row of values consisting of the MAC ID, device name, current time, temperature, and humidity level ❺. You then create a `Timer` object from the `threading` module and set it up to call the same `scan_task()` method after a time interval `SCAN_INT` ❻. Finally, you start the timer ❼. This way, once the time interval has passed, `scan_task()` will be executed in a new thread that will run parallel to the rest of the program, and the cycle will be repeated.

The Web Server Code

In this section, we'll look at the code in *server.py*, which implements a web server on the Raspberry Pi using `Bottle`. This code will generate a web page displaying the latest sensor values, as well as a plot of the data. Along with Python, you'll be using small doses of HTML, CSS, and JavaScript in your code, but you don't need to be an expert

in web development to understand the project. At a high level, HTML provides *structure* for a web page, CSS determines the *style* of presentation, and JavaScript facilitates *actions* on the page.

To see the complete code listing, skip ahead to "The Complete Python Web Server Code" on page 349. You can also find this code at *https://github.com/mkvenkit/pp2e/blob/main/iotgarden/server.py*.

Creating and Running the Server

The Python code to manage the web server is encapsulated in a class called `IOTGServer`. Here's the class's constructor:

```
class IOTGServer:
    def __init__(self, dbname, host, port):
        self._dbname = dbname
        self._host = host
        self._port = port
        # create bottle object
      ❶ self._app = Bottle()
```

The `IOTGServer` constructor takes in and stores the database name, hostname, and port number. The constructor also creates a `Bottle` instance ❶, which you'll use to implement the web server.

We briefly explored how the `Bottle` web framework works earlier in the chapter. As you saw, using `Bottle` involves defining routes to web resources and binding those routes to handler functions that will be called when someone visits that route. The `IOTGServer` class's `run()` method does just that.

```
def run(self):
```

```
    # ----------
    # add routes:
    # ----------
    # T/H data
❶   self._app.route('/thdata')(self.thdata)
    # plot image
❷   self._app.route('/image/<macid>')(self.plot_image)
    # static files - CSS, JavaScript
❸   self._app.route('/static/<filename>')(self.st_file)
    # main HTML page
❹   self._app.route('/')(self.main_page)

    # start server
❺   self._app.run(host=self._host, port=self._port)
```

You start by creating four routes, pairing each with its own handler method. The `/thdata` route ❶ returns the latest sensor data for all scanned devices in JSON format. The `/image/<macid>` route ❷ is for an image showing a plot of the sensor data. The plot image will be dynamically created from values in the SQLite database using `matplotlib`. The `<macid>` portion of the route uses `Bottle` URL template syntax to create a placeholder for your device's MAC ID. As you'll see later, the actual ID will be filled in by the JavaScript code. The route at ❸ is a little different: `Bottle` allows you to serve *static* files (files that you already have on disk) using the `/static` keyword in the route. In this case, you'll be serving JavaScript and CSS files using this scheme. Finally, the `/` route ❹ is for the main HTML page, which will be loaded when you run the server. The method ends with a call to the `run()` method on the `Bottle` instance to start the server ❺.

Serving the Main Page

Now we'll look at `main_page()`, the handler method bound to the `Bottle` route for the main HTML page. This method will

be called when the user navigates to *http://<iotgarden>.local:8080/* in a web browser.

```
    def main_page(self):
        """main HTML page"""
      ❶ response.content_type = 'text/html'
        strHTML = """
<!DOCTYPE html>
<html>
<head>
<link href="static/style.css" rel="stylesheet">
<script src="static/server.js"></script>
</head>
<body>
<div id = "title">The IoT Garden </div>
<hr/>
<div id="sensors"></div>
</body>
</html>"""
        return strHTML
```

You set the content type of the response the method will return to be either text or HTML ❶. Then you put together the HTML as a multiline string declared within triple quotes (`"""`). In the HTML code, you load a CSS style sheet file ❷ and JavaScript file ❸. These files, which will help style the page and fetch the latest sensor data, will be served using the `/static` route we discussed earlier. You next declare an empty `<div>` element ❹, which is a section or division in an HTML document, assigning it an ID of `sensors`. This will be populated dynamically by the code in the JavaScript file, as you'll see later.

Here's the `/static` route's handler method, which serves the JavaScript and CSS files for the main page:

```
def st_file(self, filename):
```

```
    """serves static files"""
    return static_file(filename, root='./static')
```

The JavaScript and CSS files are in a */static* subfolder with respect to the Python code. You use the `Bottle` framework's `static_file()` method to serve the file from this subfolder with the given filename.

Retrieving the Sensor Data

As you've seen, there are two `Bottle` routes associated with sensor data: `/image/<macid>`, which retrieves a plot of a device's data, and `/thdata`, which retrieves the most recent sensor data for all devices. We'll look at the handler methods associated with those routes now, starting with the `plot_image()` method, which is bound to the `/image/<macid>` route.

```
def plot_image(self, macid):
    """create a plot of sensor data by reading database"""
    # get data
  ❶ data = self.get_data(macid)
    # create plot
    plt.legend(['T', 'H'], loc='upper left')
  ❷ plt.plot(data)
    # save to a buffer
  ❸ buf = io.BytesIO()
  ❹ plt.savefig(buf, format='png')
    # reset stream position to start
    buf.seek(0)
    # read image data as bytes
  ❺ img_data = buf.read()
    # set response type
    response.content_type = 'image/png'
    # return image data as bytes
  ❻ return img_data
```

The method takes in the MAC ID of the device whose sensor data you want to plot. You call the `get_data()` helper method ❶, which we'll look at next, to retrieve that device's data from the SQLite database. The method returns a list of tuples with temperature and humidity readings in the form `[(T, H), (T, H), ...]`. You use `matplotlib` to plot this data ❷.

Normally, you'd call `plt.show()` to display a `matplotlib` plot on your computer, but in this case, your web server needs to send this data out as image bytes so the plot can be viewed in a browser. You use Python's `io.BytesIO` module to create a buffer that will act as a file stream to hold the image data ❸ and then save the plot to buffer in the PNG format ❹. Next, you reset the stream with `buf.seek(0)`, which in turn sets you up to read the image bytes from the beginning ❺. After setting the response return type to a PNG image, you return the image bytes ❻.

Here's the `get_data()` method that you called as part of `plot_image()`. It retrieves all temperature and humidity readings from the device with the given MAC ID.

```
def get_data(self, macid):
    # connect to database
    con = sqlite3.connect(self._dbname)
    cur = con.cursor()
    data = []
  ❶ for row in cur.execute("SELECT * FROM iotgarden_data
                           where DEVID = :dev_id LIMIT 100",
  {"dev_id" : macid}):
      ❷ data.append((row[3], row[4]))
    # commit changes
    con.commit()
    # close db
    con.close()
    return data
```

After establishing a connection to your SQLite database, you issue a query to get the 100 most recent rows in the database with a `DEVID` equal to the MAC ID passed into this method ❶. The rows returned from the database have fields in the form `(DEVID, NAME, TS, T, H)`. You pick up just the last two elements of each row (a temperature reading and a humidity reading) and append them as a tuple to the `data` list ❷. You end up with the list of tuples that the `plot_image()` method expects.

The other handler method that works with sensor data is `thdata()`, the handler for the `/thdata` route. This method returns the latest temperature and humidity values for each of your Adafruit BLE peripheral devices:

```
def thdata(self):
    """connect to database and retrieve latest sensor data"""
    # connect to database
    con = sqlite3.connect(self._dbname)
    cur = con.cursor()
    macid = ""
    name = ""

    # set up a device list
    devices = []

    # get unique device list from db
  ❶ devid_list = cur.execute("SELECT DISTINCT DEVID FROM
  iotgarden_data")
    for devid in devid_list:
      ❷ for row in cur.execute("SELECT * FROM iotgarden_data
            where DEVID = :devid ORDER BY TS DESC LIMIT 1",
            {"devid" : devid[0]}):
          ❸ devices.append({'macid': macid, 'name': name,
  'T' : T, 'H': H})

        # commit changes
    con.commit()
    # close db
    con.close()
```

```
    # return device dictionary
  ❹ return {"devices" : devices}
```

Here you run the following query on your SQLite database: `SELECT DISTINCT DEVID FROM iotgarden_data` ❶. This returns all the unique device IDs in the database. For example, if you have three Adafruit boards set up for this project, the query will return the device IDs for all three of them. For each device ID you've found, you run the following database query: `SELECT * FROM iotgarden_data where DEVID = :devid ORDER BY TS DESC LIMIT 1` ❷. This gets you the latest (by timestamp) row of data available for the given device ID. You add the retrieved information into a `devices` list ❸. Each element in the list is a dictionary. This list is mapped to a `"devices"` key in a dictionary and returned ❹. The format followed here is JSON—a nested dictionary of lists and dictionaries—which will be convenient to parse in the JavaScript code.

The JavaScript

Next, let's take a look at the JavaScript code in file *static/server.js*. This file is included in the HTML returned by the `main_page()` handler method, so the JavaScript code will run locally in a web browser on the user's machine (typically not the Raspberry Pi) when they visit the main project page. The code uses `Bottle` paths to dynamically add the sensor data to the home page's HTML.

```
// async function that fetches data from server
async function fetch_data() {
  ❶ let response = await fetch('thdata');
  ❷ devices_json = await response.json();
    console.log('updating HTML...');
  ❸ devices = devices_json["devices"];
  ❹ let strHTML = "";
  ❺ var ts = new Date().getTime();
```

```
  ❻ for (let i = 0; i < devices.length; i++) {
        // console.log(devices[i].macid)
        strHTML = '<div class="thdata">';
        strHTML += '<span>' + devices[i].name + '(' +
  devices[i].macid
            + '): </span>';
        strHTML += '<span>T = ' + devices[i].T +
                   ' C (' + (9.0*devices[i].T/5.0 + 32.0) + '
  F),</span>';
        strHTML += '<span> H = ' + devices[i].H + ' % </span>';
        strHTML += '</div>'; // thdata
        // create image div
      ❼ strHTML += '<div class="imdiv"><img src="image/' +
            devices[i]["macid"] + '?ts=' + ts + '"></div>';
        // add divider
        strHTML += '<hr/>';
    }

    // set HTML data
  ❽ document.getElementById("sensors").innerHTML = strHTML;
};
```

You first define a JavaScript function called `fetch_data()`. The `async` keyword in the definition indicates that you can call the `await` method from this function. `async` and `await` are modern JavaScript features that facilitate asynchronous programming. An example of asynchronous programming is when you request data from a server over a network, as you'll be doing here. You don't know when you'll get a response from the server, and you don't want to wait around for it. With `async` and `await`, you can go off and do other things and get notified when the response arrives. But when you get the response, it could be useful data, or it could indicate an error.

You make an asynchronous call using `await` to get the `/thdata` resource using the JavaScript `fetch()` method ❶. When this call reaches the server, it will end up calling the

route handler `thdata()` in the `IOTGarden` class in *server.py*. You then make another asynchronous call ❷ to get the response from the call ❶. This call will return only when the response is received. You expect that response to be JSON data, and you retrieve the contents of the `"devices"` key from the data ❸ . This will be a list of sensor data values from your devices.

Next, you create an empty string that you'll use to build up the HTML for showing the sensor data ❹. You use the JavaScript `Date()` method to get a current timestamp ❺, which you'll soon put to use in a little trick to ensure that the plot image loads correctly. Then you loop through all the devices ❻ and create the required HTML. Notice especially how you create an HTML `<img>` element for displaying the `matplotlib` plot ❼, supplying the `Bottle` route `/image/<macid>` as the location from which the image should be retrieved.

Once you've built up the HTML string, you add it to the home page's HTML ❽. Specifically, you set the HTML into the `<div>` with the ID of `"sensors"`. This was the empty `<div>` you created in `main_page()`, the `IOTGarden` class's handler method for the main page route in the server.

To better illustrate what the code in the `for` loop at ❻ is doing, here's an example of the output HTML string produced in one iteration through the loop. Note that the output has been formatted for readability.

```
<div class="thdata">
    <span>IGDE74(DE:74:03:D9:3D:8B): </span>
    <span>T = 26 C, (73.4 F),</span>
    <span>H = 55 % </span>
</div>
<div class="imdiv">
  ❸ <img src="image/DE:74:03:D9:3D:8B?ts=1635673486192">
</div>
<hr>
```

You first have a `<div>` of class `thdata` ❶, which contains three `<span>` elements holding the device name and MAC ID, the latest temperature reading, and the latest humidity reading. Then you have another `<div>` of class `imdiv` ❷. This `<div>` contains the `<img>` element for displaying the plot ❸. The `src` of this element is set as `image/DE:74:03:D9:3D:8B?ts=1635673486192`. The first part of this, `image/DE:74:03:D9:3D:8B`, is the route to the `plot_image()` method in `IOTGarden`, which takes the device's MAC ID as an argument. The `ts` part is a current timestamp, which tricks the browser into not caching the image. Browsers use caching to skip loading web resources that they think they've already loaded recently, but you want to ensure that the plot image gets updated regularly. Adding the timestamp to the image's URL makes the URL different every time, so the browser will keep retrieving this resource.

The HTML string ends with a horizontal line ❹ to act as a separator between data for each device. If you have multiple Adafruit BLE devices, you'd see a similar block of HTML for the next device after this first one.

The *static/server.js* file concludes with this JavaScript code:

```
// fetch once on load
window.onload = function() {
    fetch_data();
};

// now fetch data every 30 seconds
setInterval(fetch_data, 30000)
```

Here you set an anonymous function to be called as soon as the web page loads ❶. This function will call `fetch_data()`, the asynchronous function you defined earlier. This ensures that sensor data will be displayed immediately when the user visits the page. You use the JavaScript `setInterval()`

function ❷ to call `fetch_data()` every 30,000 milliseconds—or every 30 seconds, that is. This way the user will see real-time updates to both the plot and the latest sensor readings.

The CSS

CSS controls the appearance of a web page through a *style sheet*, which sets rules for how different HTML elements should be rendered. These rules rely on a *box model*, where every element in your HTML is considered a rectangular box. You can control almost every aspect of a box's appearance by specifying colors, transparency, margins, borders, text fonts, layout qualifiers, and so on. The CSS rules for the project's main page are in the file *static/style.css*.

```
html {
    background-color: gray;
}

body {
    min-height: 100vh;
    max-width: 800px;
    background-color: #444444;
    margin-left: auto;
    margin-right: auto;
    margin-top: 0;
}

h1 {
    color: #aaaaaa;
    font-family: "Times New Roman", Times, serif;
}

#title {
    font-family: "Times New Roman", Times, serif;
    font-size: 40px;
    text-align: center;
```

```
}

.thdata {
    display: flex;
    justify-content: center;
    width: 80%;
    color: #aaaaaa;
    font-family: Arial, Helvetica, sans-serif;
    font-size: 24px;
    margin: auto;
}

.imdiv {
    display: flex;
    justify-content: center;
}
```

We won't dwell on the details of this CSS file, but notice the code blocks at ❶ and ❷. These lay out rules for the display of HTML elements with a `class` attribute of `thdata` and `imdiv`, respectively. These are `<div>` elements generated by the JavaScript file we just looked at.

The Main Program File

The main program file *iotgarden.py* coordinates all the code running on the Raspberry Pi. This file is responsible for creating and managing the SQLite database, starting the BLE scanner, running the `Bottle` web server, and accepting command line arguments. To see the complete code listing, skip ahead to "The Complete Main Program Code" on page 351. You can also find this code at *https://github.com/mkvenkit/pp2e/blob/main/iotgarden/iotgarden.py*.

The Database Setup

The main program uses function `setup_db()` to prepare the SQLite database. You'll call this function the first time you run the code, or anytime you want to clear the database of old data and start fresh.

```
def setup_db(dbname):
    """set up the database"""
    # connect to database - will create new if needed
  ❶ con = sqlite3.connect(dbname)
    cur = con.cursor()
    # drop if table exists
  ❷ cur.execute("DROP TABLE IF EXISTS iotgarden_data")
    # create table
  ❸ cur.execute("CREATE TABLE iotgarden_data(DEVID TEXT, NAME
  TEXT,
                  TS DATETIME, T NUMERIC, H NUMERIC)")
```

You start by connecting to the SQLite database with a given name and path ❶. If the database doesn't exist yet, this call will create it. You clear off the `iotgarden_data` table if it already exists in the database ❷. Then you create a new table of that name ❸. The table is given the following fields: `DEVID` (a text field for the device's MAC ID), `NAME` (a text field for the name of the device), `TS` (a timestamp of type `DATETIME`), and numeric `T` (temperature) and `H` (humidity) fields.

The main program also has a utility function `print_db()` for listing the current contents of the database. This can be useful for debugging purposes, or if you want to view all the sensor data as simple text output rather than a graphical plot.

```
def print_db(dbname):
    """prints contents of database"""
    # connect to database
    con = sqlite3.connect(dbname)
```

```
    cur = con.cursor()
  ❶ for row in cur.execute("SELECT * FROM iotgarden_data"):
        print(row)
```

After connecting to the database, you execute a query to gather all rows from the table of sensor data ❶, which you then print out, one row at a time.

The main() Function

Now let's look at the `main()` function:

```
def main():
    print("starting iotgarden...")
    # set up cmd line argument parser
    parser = argparse.ArgumentParser(description="iotgarden.")
    # add arguments
  ❶ parser.add_argument('--createdb', action='store_true',
  required=False)
  ❷ parser.add_argument('--lsdb', action='store_true',
  required=False)
  ❸ parser.add_argument('--hostname', dest='hostname',
  required=False)
    args = parser.parse_args()

    # set database name
  ❹ dbname = 'iotgarden.db'

    if (args.createdb):
        print("Setting up database...")
      ❺ setup_db(dbname)
        print("done. exiting.")
        exit(0)

    if (args.lsdb):
        print("Listing database contents...")
      ❻ print_db(dbname)
        print("done. exiting.")
        exit(0)
```

```
  # set hostname
❼ hostname = 'iotgarden.local'
  if (args.hostname):
      hostname = args.hostname

  # create BLE scanner
❽ bs = BLEScanner(dbname)
  # start BLE
  bs.start()

  # create server
❾ server = IOTGServer(dbname, hostname, 8080)
  # run server
  server.run()
```

You use a `parser` object to add command line options for creating or resetting the database ❶ and for printing out the database contents ❷. You also add a `--hostname` option ❸, which lets you use a different hostname—this is useful if you name your Pi something other than `iotgarden`.

Next, you declare the filename for the SQLite database ❹. Then, if the `--createdb` command line option was used, you call the `setup_db()` function discussed earlier ❺. You print out the database contents if the `--lsdb` command line option is set ❻. You then set the hostname to `iotgarden.local` by default ❼, but you override this if a different hostname was set in the command line.

To finish, you create an object of your `BLEScanner` class ❽ and set it in motion with its `start()` method. Similarly, you create the `Bottle` server ❾ and launch it by calling the `run()` method. The scanner and web server will run in parallel with each other.

Running the IoT Garden

The code for this project lives in two places. First, there's the CircuitPython code in *ble_sensors.py*, which needs to be renamed *code.py* and uploaded to the Adafruit BLE Sense boards, as discussed in "CircuitPython Setup" on page 318. Once you get each BLE board up and running, you'll need to determine its MAC address. For this, connect the board to power, and run the following on your Raspberry Pi:

```
$ sudo hcitool lescan
```

Here's my output as an example:

```
LE Scan...
57:E0:F5:93:AD:B1 (unknown)
57:E0:F5:93:AD:B1 (unknown)
DE:74:03:D9:3D:8B (unknown)
DE:74:03:D9:3D:8B IOTG1
27:FE:36:49:F0:2E (unknown)
7A:17:EB:3C:04:A5 (unknown)
7A:17:EB:3C:04:A5 (unknown)
```

Here, my board's MAC ID `DE:74:03:D9:3D:8B` appears next to `IOTG1`. (It will likely be a different ID for you.) Take note of this ID and add it to the allowlist in your BLE scanner code.

The rest of the code goes on the Raspberry Pi, which you can work with using SSH and VS Code, as discussed in Appendix B. When you're ready to try it, run the following from the code directory:

```
$ sudo python iotgarden.py
```

You'll see a stream of messages on your shell similar to the following output:

```
starting iotgarden...
```

```
BLE scan started...
CompletedProcess(args=['sudo', 'hciconfig', 'hci0', 'reset'],
  returncode=0)
04 3E 1C 02 01 02 01 8B 3D D9 03 74 DE 10 0F FF 22 08 0B CD AB
  49 4F 54 47 31
1A 3A 30 30 C9 26 58
BLE scan stopped.
Bottle v0.12.19 server starting up (using WSGIRefServer())...
Listening on http://iotgarden.local:8080/
Hit Ctrl-C to quit.
```

Now, open a browser window on any computer on the same local network, and navigate to *http://<iotgarden>.local:8080/*, substituting your Raspberry Pi name as appropriate. You should see output similar to Figure 14-4.

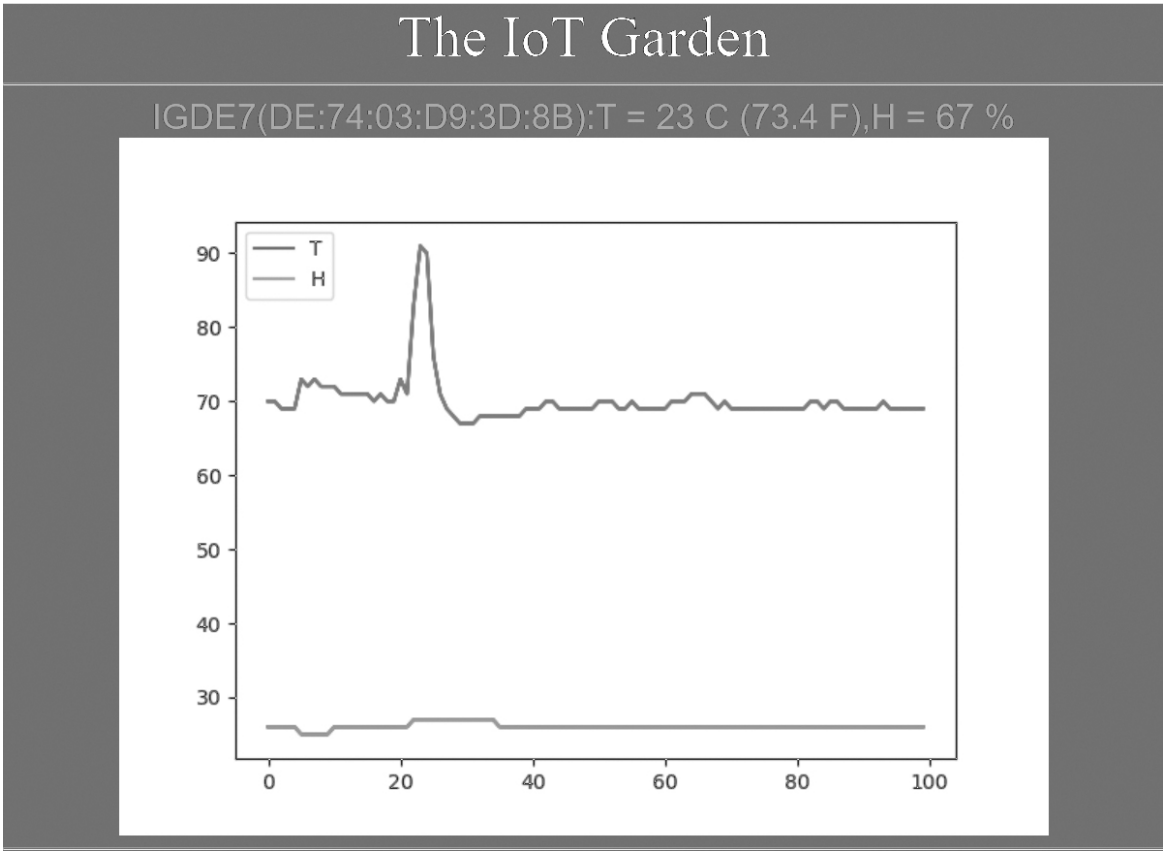


Figure 14-4: *The browser output from the IoT garden project*

The browser output will show a web page with the device ID and the latest temperature and humidity values, along with a graph of the latest 100 values from the device. This information will be repeated for each device that you have configured. The graph will refresh automatically every 30 seconds to show you the latest data.

If you don't want to wait for an exceptionally hot or dry day to test the IFTTT alert system, put your finger on the temperature sensor (or warm it up slightly using any other method, taking care not to damage the board) so that it exceed the temperature threshold `self.TMAX` set in *BLEScanner.py*. You should see a message like this in your shell on the Raspberry Pi:

```
Triggering IFTTT alert!
```

In a few seconds, you should also get an alert on the IFTTT app on your phone.

Summary

We've covered a lot of ground in this chapter! You learned about a multitier IoT architecture consisting of hardware, software, and the cloud. You learned how to use CircuitPython to get data from sensors and transmit it via the BLE wireless protocol. You also learned how to run a simple web server on your Raspberry Pi and how to display sensor data in a web browser using HTML, JavaScript, and CSS. You even learned how to get live alerts from your IoT garden using the IFTTT service.

Experiments!

1. For simplicity, the web page you created to display your sensor data is quite limited, but it provides a good framework you can build on to create more sophisticated visualizations of the data, especially since you're using a structured database like SQLite to store the sensor values. Here are a few suggestions to improve the page:
    - Instead of showing data from all the BLE Sense boards at once, create a pull-down menu to select the board whose data you want to display.
    - Implement a way to set how many days of data you want to show in the plot. You can use the JavaScript DatePicker, along with a custom `Bottle` route and a special SQLite query to implement this.
    - Customize your JavaScript code to update the latest sensor values every few seconds and only refresh the plot on a longer time scale.
2. You wrote the CircuitPython code to transmit the temperature and humidity data as single-byte integers, but what if you want more accuracy? How could you transmit a value like `26.54` over BLE? (Hint: there are two unused bytes in the manufacturer data field of the BLE packet. For a value like `26.54`, you could store the `54` in a byte and divide it by `100` upon receipt in the BLE scanner code.)
3. Power consumption is a major concern in IoT. Think about how you can save battery power on the devices. One method is to slow down the BLE advertisement interval. Another option you can explore is to use the Arduino C++ library instead of CircuitPython on the Adafruit boards. This will allow you to reduce the power consumption of the device

by putting it to deep sleep during times when it isn't reading the sensor data or transmitting BLE messages.

The Complete CircuitPython Code

Here's the complete *ble_sensors.py* code listing:

```
"""
ble_sensors.py

CircuitPython code for Adafruit BLE Sense boards. This program reads
data from the built-in Temperature and Humidity sensors, and puts the
data in the BLE advertisement packet.

Author: Mahesh Venkitachalam
"""

import time
import struct
import board
import adafruit_bmp280
import adafruit_sht31d
from adafruit_ble import BLERadio
from adafruit_ble.advertising import Advertisement, LazyObjectField
from adafruit_ble.advertising.standard import ManufacturerData, ManufacturerDataField
import _bleio
import neopixel

# derived from adafruit_ble class Advertisement
class IOTGAdvertisement(Advertisement):
    flags = None
    match_prefixes = (
        struct.pack(
            "<BHBH",  # prefix format
```

```
            0xFF,     # 0xFF is "Manufacturer Specific Data" as
     per BLE spec
            0x0822,   # 2-byte company ID
            struct.calcsize("<H9s"), # data format
            0xabcd # our ID
        ), # comma required - a tuple is expected
    )
    manufacturer_data = LazyObjectField(
        ManufacturerData,
        "manufacturer_data",
        advertising_data_type=0xFF, # 0xFF is "Manufacturer
     Specific Data" as per BLE spec
        company_id=0x0822,          # 2-byte company ID
        key_encoding="<H",
    )
    # set manufacturer data field
    md_field = ManufacturerDataField(0xabcd, "<9s")

def main():

    # initialize I2C
    i2c = board.I2C()

    # initialize sensors
    bmp280 = adafruit_bmp280.Adafruit_BMP280_I2C(i2c)
    sht31d = adafruit_sht31d.SHT31D(i2c)

    # initialize BLE
    ble = BLERadio()

    # create custom advertisement object
    advertisement = IOTGAdvertisement()
    # append first 2 hex bytes (4 characters) of MAC address to
     name
    addr_bytes = _bleio.adapter.address.address_bytes
    name = "{0:02x}{1:02x}".format(addr_bytes[5],
     addr_bytes[4]).upper()
    # set device name
    ble.name = "IG" + name

    # set initial value
    # will use only first 5 chars of name
    advertisement.md_field = ble.name[:5] + "0000"
```

```
    # BLE advertising interval in seconds
    BLE_ADV_INT = 0.2
    # start BLE advertising
    ble.start_advertising(advertisement, interval=BLE_ADV_INT)

    # set up NeoPixels and turn them all off
    pixels = neopixel.NeoPixel(board.NEOPIXEL, 1,
                               brightness=0.1,
     auto_write=False)

    # main loop
    while True:
        # print values - this will be available on serial
        print("Temperature: {:.1f}
     C".format(bmp280.temperature))
        print("Humidity: {:.1f}
     %".format(sht31d.relative_humidity))
        # get sensor data
        # BMP280 range is -40 to 85 deg C, so add an offset to
     support
        # negative temperatures
        T = int(bmp280.temperature) + 40
        H = int(sht31d.relative_humidity)
        # stop advertising
        ble.stop_advertising()
        # update advertisement data
        advertisement.md_field = ble.name[:5] + chr(T) + chr(H)
     + "00"
        # start advertising
        ble.start_advertising(advertisement,
     interval=BLE_ADV_INT)
        # blink neopixel LED
        pixels.fill((255, 255, 0))
        pixels.show()
        time.sleep(0.1)
        pixels.fill((0, 0, 0))
        pixels.show()

        # sleep for 2 seconds
        time.sleep(2)

# call main
```

```
if __name__ == "__main__":
    main()
```

The Complete BLE Scanner Code

Here's the complete listing for the BLE scanner code, in file *BLEScanner.py*:

```
"""
BLEScanner.py

This class uses BlueZ hciconfig, hcitool, and hcidump tools to
    parse
advertisement data from BLE peripherals. It then stores them in
    a database.
This class also sends alerts via the IFTTT service.

Author: Mahesh Venkitachalam
"""

import sqlite3
import subprocess
from threading import Timer
import sys
import os
import time
import requests
from datetime import datetime

class BLEScanner:
    def __init__(self, dbname):
        """BLEScanner constructor"""
        self.T = 0
        self.H = 0
        # max values
        self.TMAX = 30
        self.HMIN = 20
        # time stamp for last alert
        self.last_alert = time.time()
```

```
    # alert interval in seconds
    self.ALERT_INT = 60
    # scan interval in seconds
    self.SCAN_INT = 10
    self._dbname = dbname
    self.hcitool = None
    self.hcidump = None
    self.task = None
    # ------------------------------------------------
    # peripheral allow list - add your devices here!
    # ------------------------------------------------
    self.allowlist = ["DE:74:03:D9:3D:8B"]

def start(self):
    """start BLE scan"""
    # start task
    self.scan_task()

def stop(self):
    """stop BLE scan"""
    # stop timer
    self.task.cancel()

def send_alert(self):
    """send IFTTT alert if sensor data has exceeded the
 thresholds"""
    # check T, H
    delta = time.time() - self.last_alert
    # print("delta: ", delta)
    if ((self.T > self.TMAX) or (self.H < self.HMIN)) and
 (delta > self.ALERT_INT):
        print("Triggering IFTTT alert!")
        key = '6zmfaOBei1DgdmlOgOi6C' # USE YOUR KEY HERE!
        url =
 'https://maker.ifttt.com/trigger/TH_alert/json/with/key/'
 + key
        json_data = {"T": self.T, "H": self.H}
        r = requests.post(url, data = json_data)
        # save last alert
        self.last_alert = time.time()

def start_scan(self):
```

```
    """starts the BlueZ tools required for scanning"""
    print("BLE scan started...")
    # reset device
    ret = subprocess.run(['sudo', '-n', 'hciconfig',
'hci0', 'reset'],
                         stdout=subprocess.DEVNULL)
    print(ret)

    # start hcitool process
    self.hcitool = subprocess.Popen(['sudo', '-n',
'hcitool', 'lescan', '--duplicates'],
                                    stdout=subprocess.DEVNU
LL)

    # start hcidump process
    self.hcidump = subprocess.Popen(['sudo', '-n',
'hcidump', '--raw'],
                                    stdout=subprocess.PIPE)

def stop_scan(self):
    """stops BLE scan by killing BlueZ tools processes."""
    subprocess.run(['sudo', 'kill', str(self.hcidump.pid),
'-s', 'SIGINT'])
    subprocess.run(['sudo', '-n', 'kill',
str(self.hcitool.pid), '-s', "SIGINT"])
    print("BLE scan stopped.")

def parse_data(self, data):
    """parses hcdump string and outputs MACID, name,
manufacturer data"""
    fields = {}
    # parse MACID
    x = [int(val, 16) for val in data.split()]
    macid = ":".join([format(val, '02x').upper() for val in
x[7:13][::-1]])
    # check if MACID is in allowlist
    if macid in self.allowlist:
        # look at 6th byte to see PDU type
        if (x[5] == 0x02): # ADV_IND
            print(data)
            fields["macid"] = macid
            # set pkt type
```

```
            #fields["ptype"] = "ADV_IND"
            # parse data
            fields["T"] = x[26]
            fields["H"] = x[27]
            name = "".join([format(val, '02x').upper() for
 val in x[21:26]])
            name = bytearray.fromhex(name).decode()
            fields["name"] = name
    return fields

def parse_hcidump(self):
    """parse output from hcidump"""
    data = ""
    (macid, name, T, H) = (None, None, None, None)
    while True:
        line = self.hcidump.stdout.readline()
        line = line.decode()
        if line.startswith('> '):
            data = line[2:]
        elif line.startswith('< '):
            data = ""
        else:
            if data:
                # concatenate lines
                data += line
                # a tricky way to remove whitespace
                data = " ".join(data.split())
                # parse data
                fields = self.parse_data(data)
                success = False
                try:
                    macid = fields["macid"]
                    T = fields["T"]
                    H = fields["H"]
                    name = fields["name"]
                    success = True
                except KeyError:
                    # skip this error, since this indicates
                    # invalid data
```

```
                    pass
                if success:
                    return (macid, name, T, H)

    def scan_task(self):
        """the scanning task which is run on a separate
     thread"""
        # start BLE scan
        self.start_scan()
        # get data
        (macid, name, self.T, self.H) = self.parse_hcidump()
        # correct temperature offset
        self.T = self.T - 40
        print(self.T, self.H)
        # stop BLE scan
        self.stop_scan()

        # send alert if required
        self.send_alert()

        # write to db
        # connect to database
        con = sqlite3.connect(self._dbname)
        cur = con.cursor()
        devID = macid
        # add data
        with con:
            cur.execute("INSERT INTO iotgarden_data VALUES (?,
     ?, ?, ?, ?)",
                (devID, name, datetime.now(), self.T, self.H))
        # commit changes
        con.commit()
        # close db
        con.close()

        # schedule the next task
        self.task = Timer(self.SCAN_INT, self.scan_task)
        self.task.start()

# use this for testing the class independently
def main():
    print("starting BLEScanner...")
```

```
    bs = BLEScanner("iotgarden.db")
    bs.start()
    data = None
    while True:
        try:
            (macid, name, T, H) = bs.parse_hcidump()
            # exit(0)
        except:
            bs.stop()
            print("stopped. Exiting")
            exit(0)

        print(macid, name, T, H)
        time.sleep(10)

if __name__ == '__main__':
    main()
```

The Complete Python Web Server Code

Before we move on to the JavaScript and CSS files that round out the web server code, here's a complete listing of the Python portion of the web server code, in file *server.py*.

```
"""
server.py

This program creates a Bottle.py based web server. It also
     creates a
plot from the sensor data.

Author: Mahesh Venkitachalam
"""

from bottle import Bottle, route, template, response,
     static_file
from matplotlib import pyplot as plt
import io
```

```
import sqlite3

class IOTGServer:

    def __init__(self, dbname, host, port):
        """constructor for IGServer"""
        self._dbname = dbname
        self._host = host
        self._port = port
        # create bottle object
        self._app = Bottle()

    def get_data(self, macid):
        # connect to database
        con = sqlite3.connect(self._dbname)
        cur = con.cursor()
        data = []
        for row in cur.execute("SELECT * FROM iotgarden_data
     where DEVID = :dev_id LIMIT 100",
                               {"dev_id" : macid}):
            data.append((row[3], row[4]))
        # commit changes
        con.commit()
        # close db
        con.close()

        return data

    def plot_image(self, macid):
        """create a plot of sensor data by reading database"""
        # get data
        data = self.get_data(macid)
        # create plot
        plt.legend(['T', 'H'], loc='upper left')
        plt.plot(data)
        # save to a buffer
        buf = io.BytesIO()
        plt.savefig(buf, format='png')
        # reset stream position to start
        buf.seek(0)
        # read image data as bytes
        img_data = buf.read()
```

```
        # set response type
        response.content_type = 'image/png'
        # return image data as bytes
        return img_data

    def main_page(self):
        """main HTML page"""
        response.content_type = 'text/html'
        strHTML = """
<!DOCTYPE html>
<html>
<head>
<link href="static/style.css" rel="stylesheet">
<script src="static/server.js"></script>
</head>
<body>
<div id = "title">The IoT Garden </div>
<hr/>
<div id="sensors"></div>
</body>
</html>"""
        return strHTML

    def thdata(self):
        """connect to database and retrieve latest sensor
    data"""
        # connect to database
        con = sqlite3.connect(self._dbname)
        cur = con.cursor()
        # set up a device list
        devices = []
        # get unique device list from db
        devid_list = cur.execute("SELECT DISTINCT DEVID FROM
    iotgarden_data")
        # print(devid_list)
        for devid in devid_list:
            for row in cur.execute("SELECT * FROM
    iotgarden_data where DEVID = :devid
                ORDER BY TS DESC LIMIT 1",
                {"devid" : devid[0]}):
```

```
            devices.append({'macid': row[0], 'name':
 row[1], 'T' : row[3], 'H': row[4]})

        # commit changes
        con.commit()
        # close db
        con.close()
        # return device dictionary
        return {"devices" : devices}

    def st_file(self, filename):
        """serves static files"""
        return static_file(filename, root='./static')

    def run(self):
        # ----------
        # add routes:
        # ----------
        # T/H data
        self._app.route('/thdata')(self.thdata)
        # plot image
        self._app.route('/image/<macid>')(self.plot_image)
        # static files - CSS, JavaScript
        self._app.route('/static/<filename>')(self.st_file)
        # main HTML page
        self._app.route('/')(self.main_page)

        # start server
        self._app.run(host=self._host, port=self._port)
```

The Complete Main Program Code

Here's a listing for the complete main program code in file *iotgarden.py*.

```
"""
iotgarden.py
```

```
Main program for the IoT Garden project. Sets up database,
  starts the Bottle
web server, and the BLE scanner.

Author: Mahesh Venkitachalam
"""

import argparse
import sqlite3
from BLEScanner import BLEScanner
from server import IOTGServer

def print_db(dbname):
    """prints contents of database"""
    # connect to database
    con = sqlite3.connect(dbname)
    cur = con.cursor()
    for row in cur.execute("SELECT * FROM iotgarden_data"):
        print(row)

def setup_db(dbname):
    """set up the database"""
    # connect to database - will create new if needed
    con = sqlite3.connect(dbname)
    cur = con.cursor()
    # drop if table exists
    cur.execute("DROP TABLE IF EXISTS iotgarden_data")
    # create table
    cur.execute("CREATE TABLE iotgarden_data(DEVID TEXT, NAME
  TEXT,
                TS DATETIME, T NUMERIC, H NUMERIC)")

def main():
    print("starting iotgarden...")

    # set up cmd line argument parser
    parser = argparse.ArgumentParser(description="iotgarden.")
    # add arguments
    parser.add_argument('--createdb', action='store_true',
  required=False)
    parser.add_argument('--lsdb', action='store_true',
  required=False)
```

```
    parser.add_argument('--hostname', dest='hostname',
  required=False)
    args = parser.parse_args()

    # set database name
    dbname = 'iotgarden.db'

    if (args.createdb):
        print("Setting up database...")
        setup_db(dbname)
        print("done. exiting.")
        exit(0)

    if (args.lsdb):
        print("Listing database contents...")
        print_db(dbname)
        print("done. exiting.")
        exit(0)

    # set hostname
    hostname = 'iotgarden.local'
    if (args.hostname):
        hostname = args.hostname

    # create BLE scanner
    bs = BLEScanner(dbname)
    # start BLE
    bs.start()

    # create server
    server = IOTGServer(dbname, hostname, 8080)
    # run server
    server.run()

# call main
if __name__ == "__main__":
    main()
```

15
AUDIO ML ON PI



In the past decade, *machine learning (ML)* has taken the world by storm. It's everywhere from facial recognition to predictive text to self-driving cars, and we keep hearing about novel applications of ML seemingly every day. In this chapter, you'll use Python and TensorFlow to develop an ML-based speech recognition system that will run on an inexpensive Raspberry Pi computer.

Speech recognition systems are already deployed in a huge number of devices and appliances in the form of voice assistants such as Alexa, Google, and Siri. These systems can perform tasks ranging from setting reminders to switching on your home lights from your office. But all of these platforms require your device to be connected to the internet and for you to sign up for their services. This brings up issues of privacy, security, and power consumption. Does your light bulb *really* need to be connected to the internet to respond to a voice command? The answer is *no*. With this project, you'll get a sense of how to design a speech recognition system that works on a low-power device, without the device needing to be connected to the internet.

Some of the concepts you'll learn about through this project are:

- Using a machine learning workflow to solve a problem
- Creating an ML model with TensorFlow and Google Colab
- Streamlining an ML model for use on a Raspberry Pi
- Processing audio and generating spectrograms with the short-time Fourier transform (STFT)
- Leveraging multiprocessing to run tasks in parallel

A Machine Learning Overview

It's impossible to do justice to a topic as vast as machine learning in a single section of a single book chapter. Instead, our approach will be to treat ML as just another tool for solving a problem—in this case, how to distinguish between different spoken words. In truth, ML frameworks like TensorFlow have become so mature and easy to use these days that it's possible to effectively apply ML to a problem without being an expert in the subject. So in this section, we'll only briefly touch upon the ML terminology relevant to the project.

ML is a small part of the larger computer science discipline of *artificial intelligence (AI)*, although when AI is mentioned in the popular press, ML is usually what they mean. ML itself is made of various subdisciplines that involve different approaches and algorithms. In this project, you'll use a subset of ML called *deep learning*, which harnesses *deep neural networks (DNNs)* to identify features and patterns in large sets of data. DNNs have their origin in *artificial neural networks (ANNs)*, which are loosely based on neurons in our brains. ANNs consists of a bunch of *nodes* with multiple inputs. Each node also has a *weight* associated with it. The output of an ANN is typically a nonlinear

function of the inputs and weights. This output can be connected to the input of another ANN. When you have more than one layer of ANNs, the network becomes a deep neural network. Typically, the more layers the network has—that is, the deeper it goes—the more accurate the learning model becomes.

For this project, you'll be using a *supervised learning* process, which can be divided into two phases. First is the *training phase*, where you show the model several inputs and their expected outputs. For example, if you were trying to build a human presence detection system to recognize whether or not there's a person in a video frame, you would use the training phase to show examples of both cases (human versus no human), with each example labeled correctly. Next is the *inference phase*, where you show new inputs and the model makes predictions about them based on what it learned during training. Continuing the example, you'd show your human presence detection system new video frames, and the model would predict whether or not there's a human in each frame. (There are also *unsupervised learning* processes, in which the ML system attempts to find patterns by itself, based on unlabeled data.)

An ML model has many numerical *parameters* that help it process data. During training, these parameters are adjusted automatically to minimize errors between the expected values and the values the model predicts. Usually a class of algorithms called *gradient descent* is used to minimize the error. In addition to the parameters of an ML model, which are adjusted during training, there are also *hyperparameters*, variables that are adjusted for the model as a whole, such as which neural network architecture to use or the size of your training batch. Figure 15-1 shows the neural network architecture I've chosen for this project.

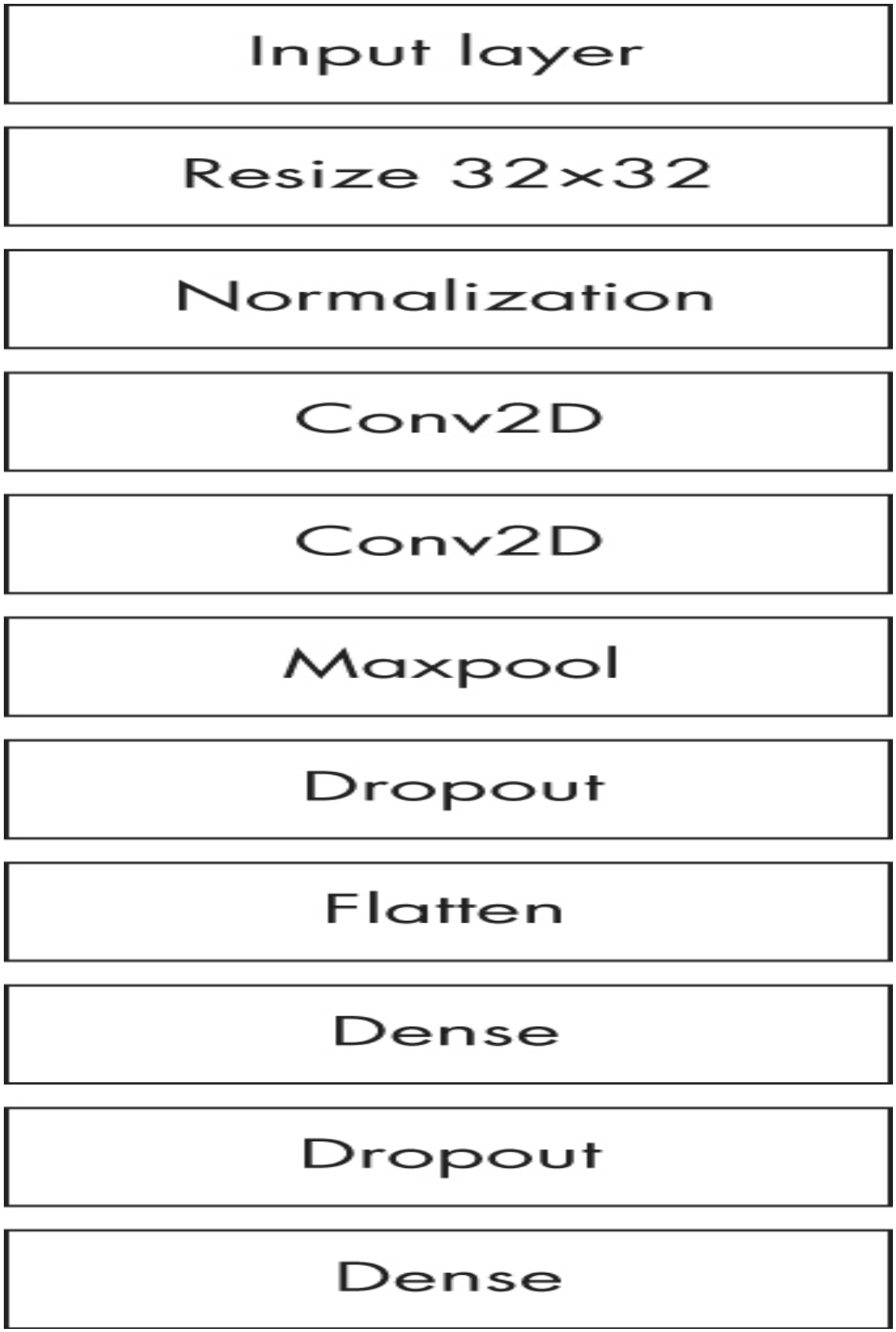

Input layer
Resize 32x32
Normalization
Conv2D
Conv2D
Maxpool
Dropout
Flatten
Dense
Dropout
Dense

Figure 15-1: *The neural network architecture for the speech recognition project*

Each layer in the network architecture represents some form of processing on the data that helps improve the model's accuracy. The design of the network isn't trivial, but just defining each layer won't tell us much about how it works. A broader question to consider is *why* I've chosen this particular network. The answer is that one needs to determine the best network architecture for the project at hand via experimentation. It's common to try different neural network architectures and see which one produces the most accurate results after training. There are also architectures published by ML researchers that are known to perform well, and that's a good place to start for practical applications.

NOTE *For more information on machine learning, I highly recommend the book* Deep Learning: A Visual Approach *by Andrew Glassner (No Starch Press, 2021). The book will gives you a good intuition about the subject without getting too much into the math or code. For a comprehensive, hands-on approach, I also recommend the online ML courses on Coursera taught by Andrew Ng.*

How It Works

In this project, you'll use Google's TensorFlow machine learning framework to train a neural network using a collection of audio files containing speech commands. Then you'll load an optimized version of the trained model onto a Raspberry Pi equipped with a microphone so the Pi can recognize the commands when you speak them. Figure 15-2 shows a block diagram for the project.

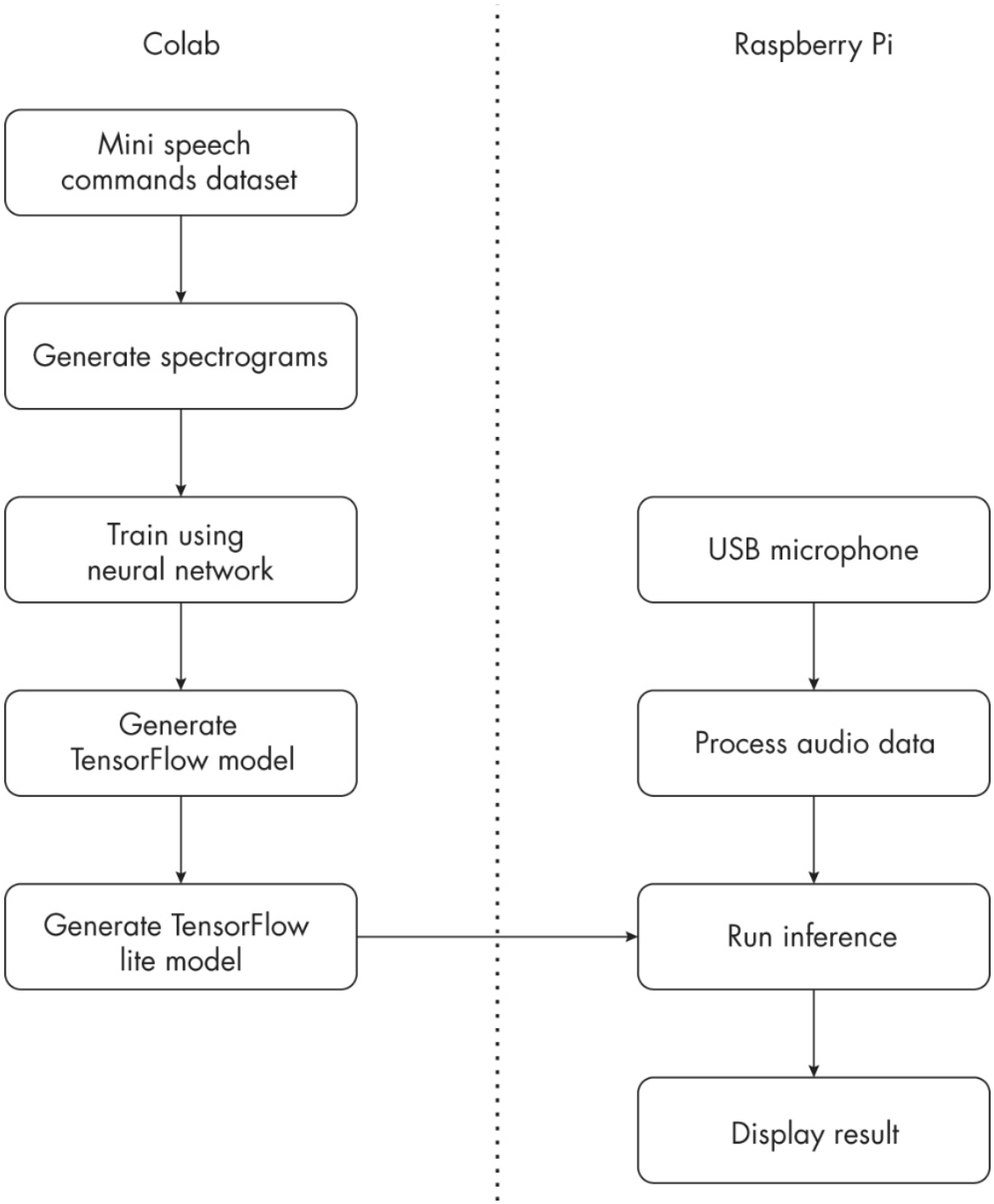


Figure 15-2: *A block diagram of the speech recognition project*

For the training portion of the project, you'll work in Google Colab (short for Colaboratory), a free cloud-based service that lets you write and run Python programs in your web browser. There are two advantages to using Colab.

First, you don’t need to install TensorFlow locally on your computer, nor deal with incompatibility issues related to various versions of TensorFlow. Second, Colab runs on machines that are likely much more powerful than yours, so the training process will go more quickly. For training data, you’ll use the Mini Speech Commands dataset from Google, a subset of a larger Speech Commands dataset published in 2018. It consists of thousands of sample recordings of the words *yes*, *no*, *up*, *down*, *left*, *right*, *stop*, and *go*, all standardized as 16-bit WAV files with a 16,000 Hz sampling rate. You’ll generate a *spectrogram* of each recording, an image that shows how the frequency content of the audio changes over time, and use those spectrograms to train a deep neural network (DNN) via TensorFlow.

NOTE *The training portion of this project takes inspiration from Google’s official TensorFlow example called “Simple Audio Recognition.” You’ll use the same neural network architecture as that example. However, the rest of the project deviates significantly from Google’s example, since our goal is to recognize live audio on a Raspberry Pi, whereas the latter runs inference on existing WAV files.*

Once the training is complete, you’ll convert the trained model to a simplified format called TensorFlow Lite, which is designed to run on less capable hardware such as embedded systems, and load that streamlined model onto the Raspberry Pi. There you’ll run Python code to continuously monitor the audio input from a USB microphone, take spectrograms of the audio, and perform inference on that data to identify the spoken commands from the training set. You’ll print out the commands that the model identifies to the serial monitor.

Spectrograms

A key step in this project is generating spectrograms of the audio data—both the preexisting data used to train the model and the real-time data encountered during inference. In Chapter 4, you saw how a spectral plot reveals the frequencies present in an audio sample at a particular moment in time. Then, in Chapter 13, you learned how spectral plots are calculated with a mathematical tool called a *discrete Fourier transform (DFT)*. A spectrogram is essentially just a series of spectral plots, generated through a sequence of Fourier transformers, which together reveal how the frequency content of some audio data evolves over time.

You need a spectrogram, rather than a single spectral plot, of each audio sample because the sound of human speech is incredibly complex. Even in the case of a single word, the frequencies present in the sound change significantly—and in distinctive ways—as the word is spoken. For this project, you'll be working with one-second-long audio clips, each consisting of 16,000 samples. If you computed a single DFT of the entire clip in one go, you wouldn't get an accurate picture of how the frequencies change over the course of the clip, and thus you wouldn't be able to reliably identify the word being spoken. Instead, you'll divide the clip into a bunch of overlapping intervals and compute the DFT for each of these intervals, giving you the series of spectral plots needed for a spectrogram. Figure 15-3 illustrates this type of computation, called a *short-time Fourier transform (STFT)*.

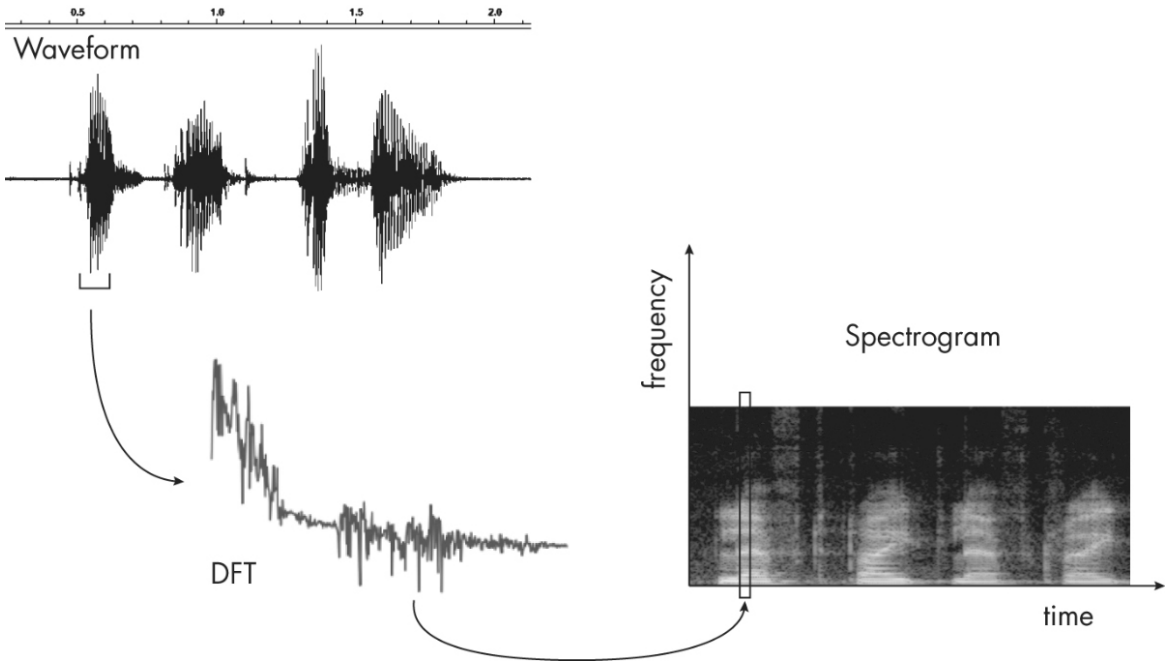


Figure 15-3: *Computing the spectrogram of a signal*

The STFT gives you *M* DFTs of the audio, taken at even time intervals. Time is shown along the x-axis of the spectrogram. Each DFT gives you *N* frequency bins and the intensity of the sound within each of those bins. The frequency bins are mapped to the y-axis of the spectrogram. Thus, the spectrogram takes the form of an *M*×*N* image. Each column of pixels in the image represents one of the DFTs, with color used to convey the intensity of the signal in a given frequency band.

You might be wondering why we need to use Fourier transforms at all for this project. Why not use the waveforms of the audio clips directly, instead of extracting the frequency information from those waveforms? For an answer, consider Figure 15-4.

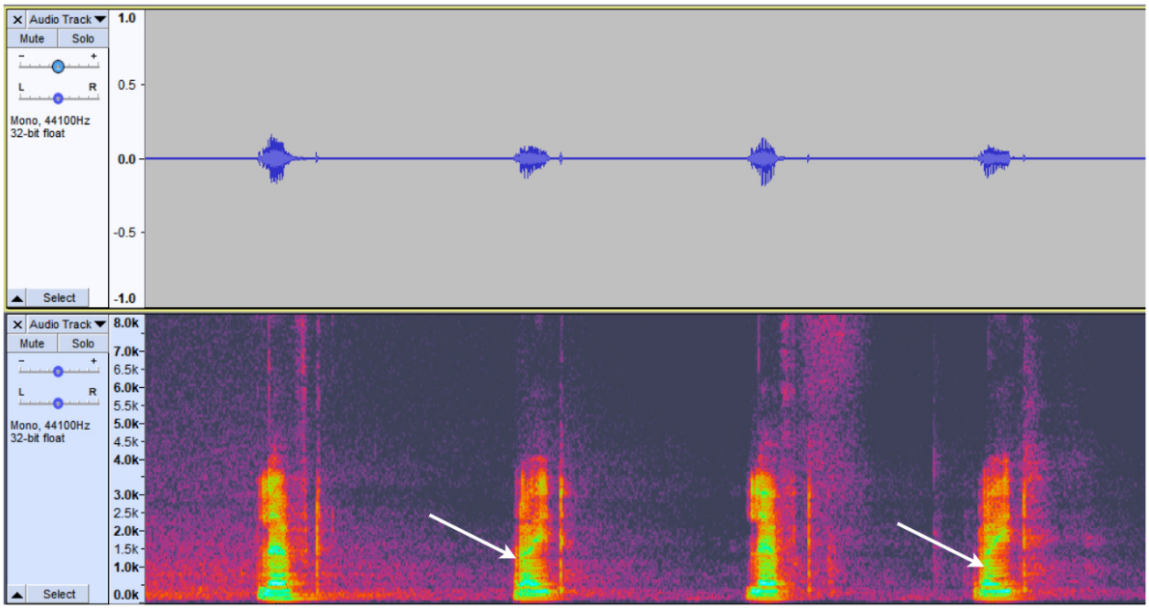


Figure 15-4: *The waveform and spectrogram of speech samples*

The top half of the figure shows the waveform of a recording made by speaking the sequence “Left, right, left, right.” The bottom half of the figure shows a spectrogram of that recording. Looking just at the waveform, you can see that the two *left*s look vaguely similar, as do the two *right*s, but it’s hard to pick out strong identifying characteristics of each word’s waveform. By contrast, the spectrogram reveals more visual features associated with each word, like the bright C-shaped curve (shown by the arrows) in each instance of *right*. We can see these distinctive features more clearly with our own eyes, and a neural network can “see” them as well.

In the end, since a spectrogram is essentially an image, taking spectrograms of the data turns a speech recognition problem into an image classification problem, allowing us to leverage the rich set of ML techniques that already exist for classifying images. (Of course, a waveform can be treated as an image too, but as you’ve seen, a spectrogram is better at capturing the “signature” of the audio data and hence more suited for ML applications.)

Inference on the Raspberry Pi

The code on the Raspberry Pi must accomplish several tasks: it needs to read the audio input from the attached microphone, compute the spectrogram of that audio, and do inference using the trained TensorFlow Lite model. Here's one possible sequence of operations:

1. Read microphone data for one second.
2. Process data.
3. Do inference.
4. Repeat.

There's a big problem with this approach, however. While you're busy with steps 2 and 3, more speech data could be coming in, which you'll end up missing. The solution is to use Python multiprocessing to perform different tasks concurrently. Your main process will just collect the audio data and put it in a queue. In a separate, simultaneous process, you'll take this data out of the queue and run inference on it. Here's what the new scheme looks like:

Main Process

1. Read microphone data for one second.
2. Put data into the queue.

Inference Process

1. Check if there's any data in the queue.
2. Run inference on the data.

Now the main process won't miss any audio input, since putting data into the queue is a very quick operation. But

there’s another problem. You’re collecting one-second audio samples continuously from the microphone and processing them, but you can’t assume that all spoken commands will fit cleanly into those one-second intervals. A command could come at an edge and be broken up across two consecutive samples, in which case it probably won’t be identified during inference. A better approach is to create overlapping samples, as follows:

Main Process

1. For the very first frame, collect a two-second sample.
2. Put the two-second sample into the queue.
3. Collect another one-second sample.
4. Create a two-second sample by moving the latter half of the sample from step 2 to the front and replacing the second half with the sample from step 3.
5. Return to step 2.

Inference Process

1. Check if there’s any data in the queue.
2. Do inference on a one-second portion of the two-second data based on peak amplitude.
3. Return to step 1.

In this new scheme, each sample placed into the queue is two seconds long, but there’s a one-second overlap between consecutive samples, as illustrated in Figure 15-5. This way, even if a word is partially cut off in one sample, you’ll get the full word in the next sample. You’ll still run inference on only one-second clips, which you’ll center on

the point in the two-second sample that has the highest amplitude value. This is the part of the sample most likely to contain a spoken word. You need the clips to be one second long for consistency with the training data.

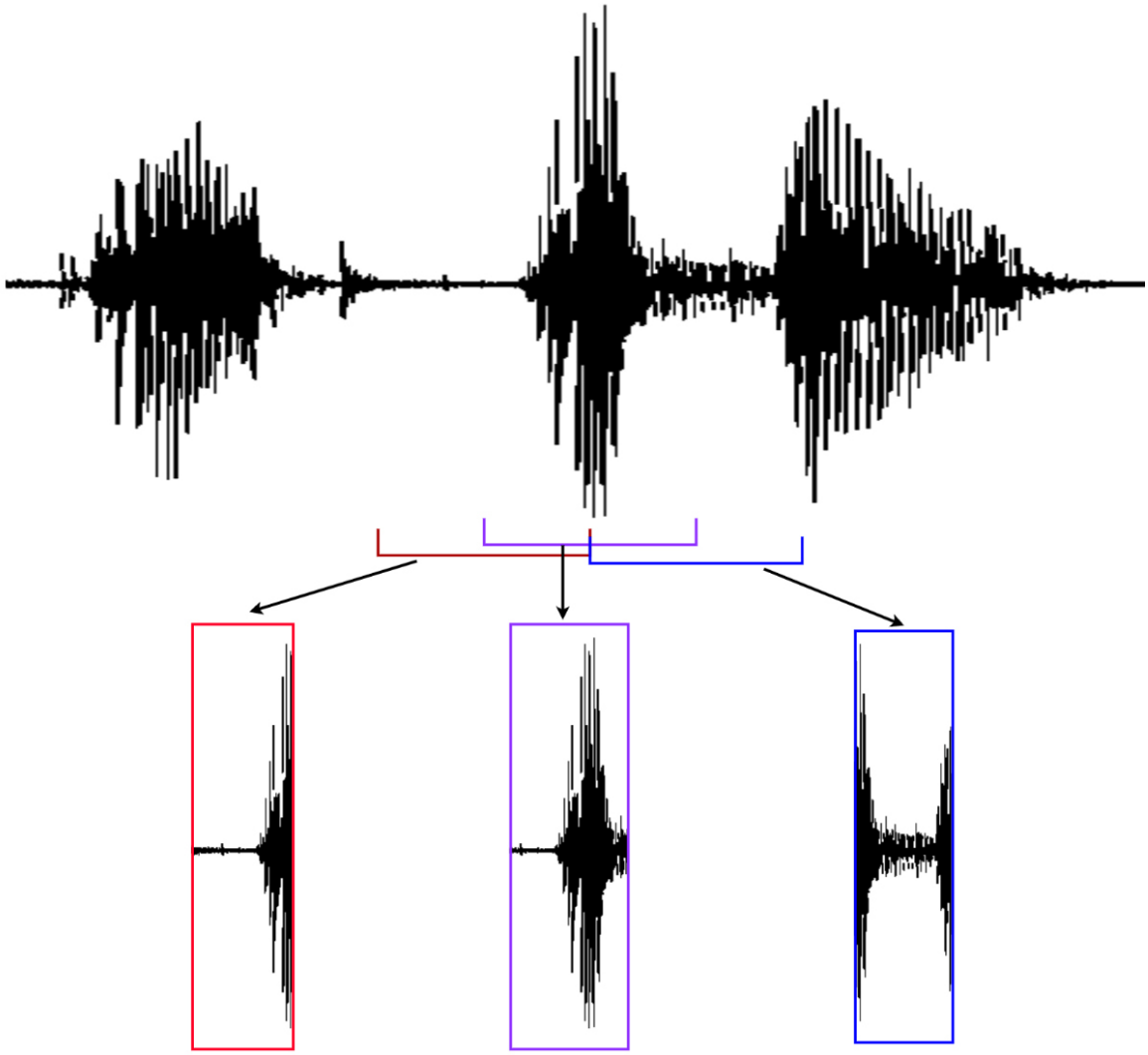


Figure 15-5: *The two-frame overlapping scheme*

Through this combination of multiprocessing and overlapping samples, you'll design a speech recognition system that minimizes missing inputs and improves inference results.

Requirements

For this project, you'll need to sign up with Google Colab to train your ML model. On the Raspberry Pi, you'll need the following Python modules:

- `tflite_runtime` for running the TensorFlow inference
- `scipy` for computing the STFT of audio waveforms
- `numpy` arrays for handling audio data
- `pyaudio` for streaming audio data from the microphone input

The installation for these modules is covered in Appendix B. You'll also use Python's built-in `multiprocessing` module for running ML inference in a separate thread from the audio processing thread.

In the hardware department, you'll need the following:

- One Raspberry Pi 3B+ or newer
- One 5 V power supply for the Raspberry Pi
- One 16GB SD card
- One single-channel USB microphone compatible with the Pi

Various types of USB microphones are compatible with the Raspberry Pi. Figure 15-6 shows an example.



Figure 15-6: *A USB microphone for the Raspberry Pi*

To check if your Pi can recognize your USB microphone, SSH into your Pi and run the following command:

```
$ dmesg -w
```

Now plug your microphone into a USB port on the Pi. You should see something similar to the following output:

```
[26965.023138] usb 1-1.3: New USB device found, idVendor=cafe,
     idProduct=4010, bcdDevice= 1.00
[26965.023163] usb 1-1.3: New USB device strings: Mfr=1,
     Product=2, SerialNumber=3
[26965.023179] usb 1-1.3: Product: Mico
[26965.023194] usb 1-1.3: Manufacturer: Electronut Labs
[26965.023209] usb 1-1.3: SerialNumber: 123456
```

The information should match the specs of your microphone, indicating it's been correctly identified.

The Code

The code for this project exists in two parts: the training portion, which you'll run in Google Colab, and the inference portion, which you'll run on your Raspberry Pi. We'll examine these parts one at a time.

Training the Model in Google Colab

In this section, we'll look at the Google Colab code needed to train the speech recognition model. I recommend working with Colab in the Chrome web browser. You'll begin by getting set up and downloading the training dataset. Then you'll run some code to get to know the data. You'll clean up the data to prepare it for training and explore how to generate spectrograms from the data. Finally, you'll put what you've learned to work by creating and training the model. The end result will be a *.tflite* file, a streamlined TensorFlow Lite version of the trained model that you can load onto your Raspberry Pi. You can also download this file from the book's GitHub repository at *https://github.com/mkvenkit/pp2e/tree/main/audioml*.

A Google Colab notebook consists of a series of cells where you enter one or more lines of Python code. Once you've entered your desired code into a cell, you run it by clicking the Play icon in the top-left corner of the cell. Any output associated with that cell's code will then appear beneath the cell. Throughout this section, each code listing will represent a complete Google Colab cell. The cell's

output, if any, will be shown in gray at the end of the listing, beneath a dashed line.

Setting Up

You begin your Colab notebook with some initial setup. First you import the required Python modules:

```
import os
import pathlib
import matplotlib.pyplot as plt
import numpy as np
import scipy
import scipy.signal
from scipy.io import wavfile
import glob

import tensorflow as tf
from tensorflow.keras.layers.experimental import preprocessing
from tensorflow.keras import layers
from tensorflow.keras import models
from tensorflow.keras import applications
```

In the next cell, you do some initialization:

```
# set seed for random functions
seed = 42
tf.random.set_seed(seed)
np.random.seed(seed)
```

Here you initialize the random functions you'll be using to shuffle the input filenames.

Next, download the training data:

```
data_dir = 'data/mini_speech_commands'
data_path = pathlib.Path(data_dir)
```

```
filename = 'mini_speech_commands.zip'
url =
     "http://storage.googleapis.com/download.tensorflow.org/dat
     a/mini_speech_commands.zip"
if not data_path.exists():
    tf.keras.utils.get_file(filename, origin=url, extract=True,
     cache_dir='.',
                            cache_subdir='data')
```

This cell downloads the Mini Speech Commands dataset from Google and extracts the data into a directory called *data*. Since you're using Colab, the data will be downloaded to the filesystem on the cloud, not to your local machine, and when your session ends, these files will be deleted. While the session is still active, however, you don't want to have to keep downloading the data every time you run the cell. The `tf.keras.utils.get_file()` function caches the data so you won't need to keep downloading it.

Getting to Know the Data

Before you start training your model, it would be useful to take a look at what you just downloaded to get to know your data. You can use Python's `glob` module, which helps you find files and directories through pattern matching:

```
glob.glob(data_dir + '/*')
```

```
['data/mini_speech_commands/up',
 'data/mini_speech_commands/no',
 'data/mini_speech_commands/README.md',
 'data/mini_speech_commands/stop',
 'data/mini_speech_commands/left',
 'data/mini_speech_commands/right',
 'data/mini_speech_commands/go',
 'data/mini_speech_commands/down',
 'data/mini_speech_commands/yes']
```

You pass `glob` the `'/*'` pattern to list all the first-level directories within the *data* directory (`*` is a wildcard character). The output shows you that the dataset comes with a *README.md* text file and eight subdirectories for the eight speech commands you'll be training the model to identify. For convenience, you create a list of the commands:

```
commands = ['up', 'no', 'stop', 'left', 'right', 'go', 'down',
  'yes']
```

In your machine learning model, you'll be matching audio samples to a `label_id` integer denoting one of the commands. These integers will correspond to the indices from the `commands` list. For example, a `label_id` of `0` indicates `'up'`, and a `label_id` of `6` indicates `'down'`.

Now take a look at what's in those subdirectories:

```
wav_file_names = glob.glob(data_dir + '/*/*')
np.random.shuffle(wav_file_names)
print(len(wav_file_names))
for file_name in wav_file_names[:5]:
    print(file_name)
```

```
8000
data/mini_speech_commands/down/27c30960_nohash_0.wav
data/mini_speech_commands/go/19785c4e_nohash_0.wav
data/mini_speech_commands/yes/d9b50b8b_nohash_0.wav
data/mini_speech_commands/no/f953e1af_nohash_3.wav
data/mini_speech_commands/stop/f632210f_nohash_0.wav
```

You use `glob` again, this time showing it the pattern `'/*/*'` to list all the files in the subdirectories ❶. Then you randomly shuffle the returned list of filenames to reduce any bias in the training data ❷. You print the total number of files found, as well as the first five filenames. The output

indicates that there are 8,000 WAV files in the dataset, and it gives you some idea of how the files are named—for example, *f632210f_nohash_0.wav*.

Next, take a look at some individual WAV files from the dataset:

```
filepath =
    'data/mini_speech_commands/stop/f632210f_nohash_1.wav' ❶
rate, data = wavfile.read(filepath) ❷
print("rate = {}, data.shape = {}, data.dtype =
    {}".format(rate, data.shape, data.dtype))

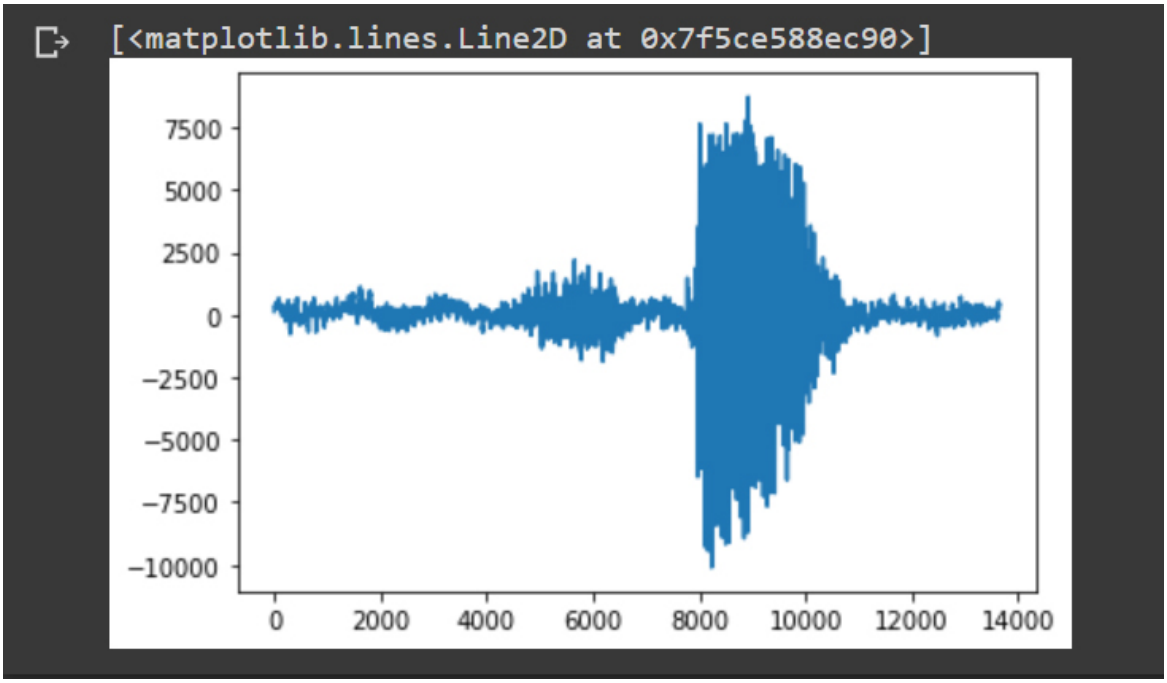
filepath = 'data/mini_speech_commands/no/f953e1af_nohash_3.wav'
rate, data = wavfile.read(filepath)
print("rate = {}, data.shape = {}, data.dtype =
    {}".format(rate, data.shape, data.dtype))
```

```
rate = 16000, data.shape = (13654,), data.dtype = int16
rate = 16000, data.shape = (16000,), data.dtype = int16
```

You set the name of a WAV file you want to look at ❶ and use the `wavefile` module from `scipy` to read data from the file ❷ . Then you print the sampling rate, shape (number of samples), and type of the data. You do the same for a second WAV file. The output shows that the sampling rates of both the WAV files are 16,000, as expected, and that each sample is a 16-bit integer for both—also expected. However, the shape indicates the first file has only 13,654 samples, and this is a problem. To train the neural network, each WAV file needs to have the same length; in this case, you'd like each recording to be one second, or 16,000 samples, long. Unfortunately, not all the files in the dataset fit that standard. We'll look at a solution to this problem shortly, but first, try plotting the data from one of these WAV files:

```
filepath =
  'data/mini_speech_commands/stop/f632210f_nohash_1.wav'
```

```
rate, data = wavfile.read(filepath)
plt.plot(data)
```



You use `matplotlib` to create a plot of the audio waveform ❶. The WAV files in this dataset contain 16-bit signed data, which can range from −32,768 to +32,767. The y-axis of the plot shows you that the data in this file ranges from only around −10,000 to +7,500. The plot's x-axis also underscores that the data is short of the necessary 16,000 samples—the axis runs only to 14,000.

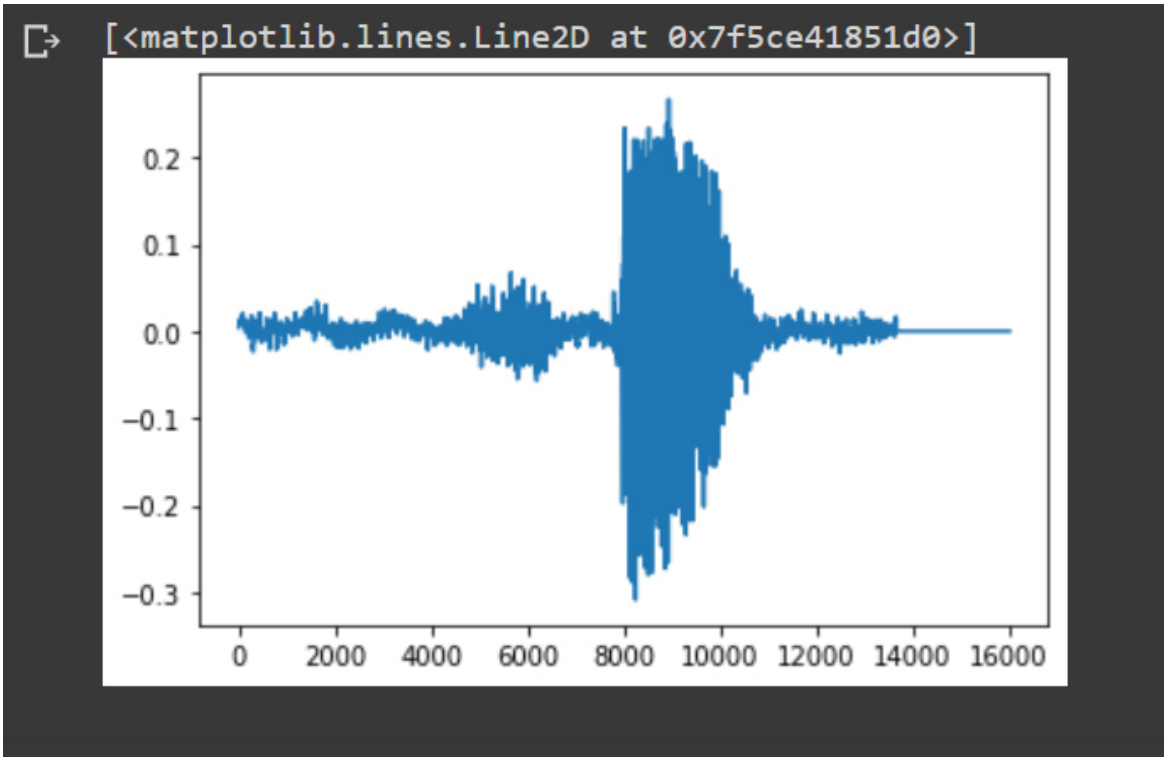
Cleaning the Data

You've seen that the dataset needs to be standardized so that each clip is one second long. This type of preparatory work is called *data cleaning*, and you can do it by padding the audio data with zeros until it reaches a length of 16,000 samples. You should clean the data further by *normalizing* it —mapping the value of each sample from range [−32,768, +32,767] to range [−1, 1]. This type of normalization is

crucial for machine learning, as keeping the input data small and uniform will help the training. (For the mathematically curious, large numbers in the inputs will cause problems in the convergence of gradient descent algorithms used to train the data.)

As an example of data cleaning, here you apply both padding and normalization to the WAV file you viewed in the previous listing. Then you plot the results to confirm that the cleaning has worked.

```
padded_data = np.zeros((16000,), dtype=np.int16)
padded_data[:data.shape[0]] = data
norm_data = np.array(padded_data/32768.0, dtype=np.float32)
plt.plot(norm_data)
```



You create a 16-bit `numpy` array of length 16,000 filled with zeros ❶. Then you use the array slice operator, `[:]`, to copy the contents of the too-short WAV file into the beginning of

the array ❷. Here `data.shape[0]` gives you the number of samples in the original WAV file, since `data.shape` is a tuple in the form `(13654,)`. You now have one second of WAV data, consisting of the original audio data followed by a padding of zeros as needed. You next create a normalized version of the data by dividing the values in the array by 32,768, the maximum value a 16-bit integer could have ❸. Then you plot the data.

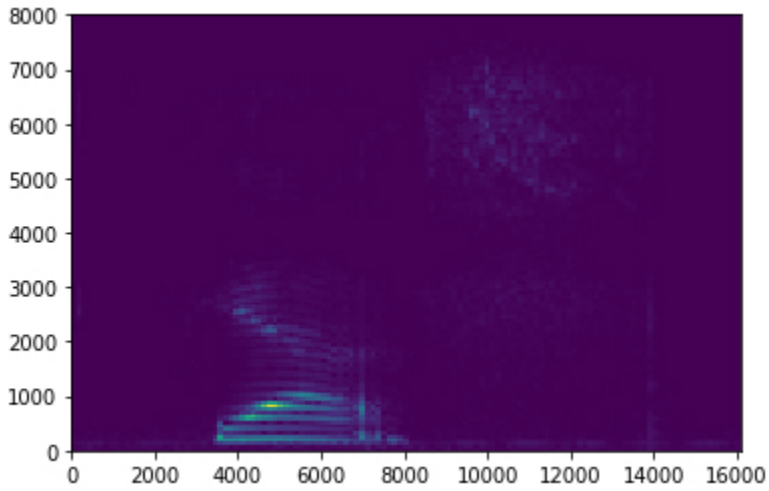
The x-axis of the output shows that the data has been padded to extend to 16,000 samples, with the values from around 14,000 to 16,000 all being zero. Also, the y-axis shows that the values have all been normalized to fall nicely within the range of (−1, 1).

Looking at Spectrograms

As we've discussed, you won't be training your model on the raw data from the WAV files. Instead, you'll generate spectrograms of the files and use them to train the model. Here's an example of how to generate a spectrogram:

```
filepath =
  'data/mini_speech_commands/yes/00f0204f_nohash_0.wav'
rate, data = wavfile.read(filepath)
f, t, spec = scipy.signal.stft(data, fs=16000, nperseg=255,
                               noverlap = 124, nfft=256)
spec = np.abs(spec)
print("spec: min = {}, max = {}, shape = {}, dtype =
  {}".format(np.min(spec),
                                          np.max(spec),
  spec.shape, spec.dtype))
X = t * 129*124
plt.pcolormesh(X, f, spec)
```

```
spec: min = 0.0, max = 2089.085693359375, shape = (129, 124),
  dtype = float32
```



You pick an arbitrary WAV file from the *yes* subdirectory and extract its data using the `wavfile` module from `scipy`, as before. Then you use the `scipy.signal.stft()` function to compute the spectrogram of the data ❶. In this function call, `fs` is the sampling rate, `nperseg` is the length of each segment, and `noverlap` is the number of overlapping samples between consecutive segments. The `stft()` function returns a tuple comprising three members: `f`, an array of frequencies; `t`, an array of the time intervals mapped to the range [0.0, 1.0]; and `spec`, the STFT itself, a grid of 129×124 complex numbers (these dimensions are given as `shape` in the output). You use `np.abs()` to convert the complex numbers in `spec` into real numbers ❷. Then you print some information about the computed spectrogram. Next, you create an array `X` to hold the sample numbers corresponding to the time intervals ❸. You get these by multiplying `t` by the dimensions of the grid. Finally, you use the `pcolormesh()` method to plot the grid in `spec`, using the values in `X` as the grid's x-axis and the values in `f` as the y-axis ❹.

The output shows the spectrogram. This 129×124 grid of values (an image), and many more like it, will be the input for the neural network. The bright spots around 1,000 Hz and lower, starting around 4,000 samples in, are where the

frequency content is most prominent, while darker areas represent less prominent frequencies.

NOTE *Notice that the y-axis in the spectrogram images goes up to only about 8,000 Hz. This is a consequence of the* sampling theorem *in digital signal processing, which states that the maximum frequency that can be accurately measured in a digitally sampled signal is half the sampling rate. In this case, that maximum frequency works out to 16,000/2 = 8,000 Hz.*

Training the Model

You're now ready to turn your attention to training the ML model, and that largely means leaving behind Python libraries like `numpy` and `scipy` in favor of TensorFlow methods and data structures like `tf.Tensor` and `tf.data.Dataset`. You've been using `numpy` and `scipy` so far because they've provided a convenient way to explore the speech commands dataset, and in fact you could continue using them, but then you'd miss out on the optimization opportunities provided by TensorFlow, which is designed for large-scale ML systems. You'll find that TensorFlow has near-identical functions for most of the computations you've done until now, so the transition will be smooth. For our purposes, when I refer to a *tensor* in the upcoming discussion, understand that it's similar to talking about a `numpy` array.

To train the ML model, you need to be able to extract the spectrogram and label ID (the spoken command) from the filepath of an input audio file. For that, first create a function that computes an STFT:

```
def stft(x):
  ❶ f, t, spec = scipy.signal.stft(x.numpy(), fs=16000,
  nperseg=255,
                                   noverlap=124, nfft=256)
```

```
  ❷ return tf.convert_to_tensor(np.abs(spec))
```

The function takes in `x`, the data extracted from a WAV file, and computes its STFT using `scipy`, as before ❶. Then you convert the returned `numpy` array to a `tf.Tensor` object and return the result ❷. There is, in fact, a TensorFlow method called `tf.signal.stft()` that's similar to the `scipy.signal.stft()` method, so why not use it? The answer is that the TensorFlow method won't be available on the Raspberry Pi, where you'll be using the slimmed-down TensorFlow Lite interpreter. Any preprocessing you do during the training phase should be identical to any preprocessing you do during inference, so you need to ensure that you use the same functions in Colab as you'll use on the Pi.

Now you can make use of your `stft()` function in a helper function that extracts the spectrogram and label ID from a filepath.

```
def get_spec_label_pair(filepath):
    # read WAV file
    file_data = tf.io.read_file(filepath)
    data, rate = tf.audio.decode_wav(file_data)
    data = tf.squeeze(data, axis=-1)
    # add zero padding for N < 16000
  ❶ zero_padding = tf.zeros([16000] - tf.shape(data),
  dtype=tf.float32)
    # combine data with zero padding
  ❷ padded_data = tf.concat([data, zero_padding], 0)
    # compute spectrogram
  ❸ spec = tf.py_function(func=stft, inp=[padded_data],
  Tout=tf.float32)
    spec.set_shape((129, 124))
    spec = tf.expand_dims(spec, -1)
    # get label
  ❹ cmd = tf.strings.split(filepath, os.path.sep)[-2]
  ❺ label_id = tf.argmax(tf.cast(cmd == commands, "uint32"))
```

```
    # return tuple
    return (spec, label_id)
```

You start by reading the file using `tf.io.read_file()` and decoding the WAV format using the `tf.audio.decode_wav()` function. (The latter is comparable to the `scipy.io.wavfile.read()` function you used previously.) You then use `tf.squeeze()` to change the shape of the `data` tensor from (*N*, 1) to (*N*,), which is required for functions coming ahead. Next, you create a tensor for zero-padding the data ❶. Tensors are immutable objects, however, so you can't copy the WAV data directly into a tensor full of zeros, as you did earlier with `numpy` arrays. Instead, you create a tensor with the exact number of zeros you need to pad the data, and then you concatenate it with the data tensor ❷.

You next use `tf.py_function()` to call the `stft()` function you defined earlier ❸. In this call, you also need to specify the input and the data type of the output. This is a common method for calling a non-TensorFlow function from TensorFlow. You then do some reshaping of the tensor returned by `stft()`. First you use `set_shape()` to reshape it to (129, 124), which is necessary because you're going from a TensorFlow function to a Python function and back. Then you run `tf.expand_dims(spec, -1)` to add a third dimension, going from (129, 124) to (129, 124, 1). The extra dimension is needed for the neural network model you'll be building. Finally, you extract the label (for example, `'no'`) associated with the filepath ❹ and convert the label string to the integer `label_id` ❺, which is the index of the string in your `commands` list.

Next, you need to get the input files ready for training. Recall that you had 8,000 audio files in the subdirectories and that you randomly shuffled their filepath strings and put them into a list called `wav_file_names`. Now you'll partition the data into three: 80 percent, or 6,400 files, for training;

10 percent, or 800 files, for validation; and the other 10 percent for testing. Such partitioning is a common practice in machine learning. Once the model is trained using training data, you can use the validation data to tweak the model's accuracy by changing the hyperparameters. The testing data is used only for checking the final accuracy of the (tweaked) model.

```
train_files = wav_file_names[:6400]
val_files = wav_file_names[6400:7200]
test_files = wav_file_names[7200:]
```

Now you load the filepath strings into TensorFlow `Dataset` objects. These objects are critical to working with TensorFlow; they hold your input data and allow for data transformations, and all this can happen at a large scale:

```
train_ds = tf.data.Dataset.from_tensor_slices(train_files)
val_ds = tf.data.Dataset.from_tensor_slices(val_files)
test_ds = tf.data.Dataset.from_tensor_slices(test_files)
```

Take a look at what you just created:

```
for val in train_ds.take(5):
    print(val)
```

```
tf.Tensor(b'data/mini_speech_commands/stop/b4aa9fef_nohash_2.wa
    v', shape=(), dtype=string)
tf.Tensor(b'data/mini_speech_commands/stop/962f27eb_nohash_0.wa
    v', shape=(), dtype=string)
--snip--
tf.Tensor(b'data/mini_speech_commands/left/cf87b736_nohash_1.wa
    v', shape=(), dtype=string)
```

Each `Dataset` object contains a bunch of tensors of type `string`, each holding a filepath. What you really need,

however, are the `(spec, label_id)` pairs corresponding to those filepaths. You create those here:

```
train_ds = train_ds.map(get_spec_label_pair)
val_ds = val_ds.map(get_spec_label_pair)
test_ds = test_ds.map(get_spec_label_pair)
```

You use `map()` to apply your `get_spec_label_pair()` function to each `Dataset` object. This technique of mapping a function to a list of things is common in computing. Essentially, you're going through each filepath in the `Dataset` object, calling `get_spec_label_pair()` on it, and storing the resulting `(spec, label_id)` pair in a new `Dataset` object.

Now you further prepare the dataset for training by splitting it up into smaller batches:

```
batch_size = 64
train_ds = train_ds.batch(batch_size)
val_ds = val_ds.batch(batch_size)
```

Here you set the training and validation datasets to have a batch size of 64. This is a common technique for speeding up the training process. If you tried to work with all 6,400 training samples and 800 validation samples at once, it would require a huge amount of memory and would slow down the training.

Now you're finally ready to create your neural network model:

```
input_shape = (129, 124, 1)
num_labels = 8
norm_layer = preprocessing.Normalization()
norm_layer.adapt(train_ds.map(lambda x, _: x))

model = models.Sequential([
```

```
    layers.Input(shape=input_shape),
    preprocessing.Resizing(32, 32),
    norm_layer,
    layers.Conv2D(32, 3, activation='relu'),
    layers.Conv2D(64, 3, activation='relu'),
    layers.MaxPooling2D(),
    layers.Dropout(0.25),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(num_labels),
])
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 resizing_3 (Resizing)       (None, 32, 32, 1)         0
 normalization_3 (Normalization)  (None, 32, 32, 1)    3
 conv2d_5 (Conv2D)           (None, 30, 30, 32)        320
 conv2d_6 (Conv2D)           (None, 28, 28, 64)        18496
 max_pooling2d_3 (MaxPooling2D)  (None, 14, 14, 64)    0
 dropout_6 (Dropout)         (None, 14, 14, 64)        0
 flatten_3 (Flatten)         (None, 12544)             0
 dense_6 (Dense)             (None, 128)               1605760
 dropout_7 (Dropout)         (None, 128)               0
 dense_7 (Dense)             (None, 8)                 1032
=================================================================
Total params: 1,625,611
```

```
Trainable params: 1,625,608
Non-trainable params: 3
```

You set the shape of the input into the first layer of the model ❶ and then set the number of labels ❷, which will be the number of units in the model's output layer. Next, you set up a normalization layer for the spectrogram data. This will scale and shift the data to a distribution centered on 1 with a standard deviation of 1. This is a common practice in ML that improves training. Don't let the `lambda` scare you ❸. All it's doing is defining an anonymous function that picks out just the spectrogram from each `(spec, label_id)` pair in the training dataset. The `x, _: x` is just saying to ignore the second element in the pair and return only the first element.

You next create the neural network model, one layer at a time ❹. The layers correspond to the architecture we viewed earlier in Figure 15-1. Finally, you print out a summary of the model ❺, which is shown in the output. The summary tells you all the layers in the model, the shape of the output tensor at each stage, and the number of trainable parameters in each layer.

Now you need to compile the model. The compilation step sets the optimizer, the loss function, and the data collection metrics for the model:

```
model.compile(
    optimizer=tf.keras.optimizers.Adam(),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_log
  its=True),
    metrics=['accuracy'],
)
```

A *loss function* is a function used to measure how well a neural network is doing by comparing the output of the model to the known correct training data. An *optimizer* is a

method used to adjust the trainable parameters in a model to reduce the losses. In this case, you're using an optimizer of type `Adam` and a loss function of type `SparseCategoricalCrossentropy`. You also get set up to collect some accuracy metrics, which you'll use to check how the training went.

Next, you train the model:

```
EPOCHS = 10
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=EPOCHS,
    callbacks=tf.keras.callbacks.EarlyStopping(verbose=1,
     patience=15), ❶
)
```

```
Epoch 1/10
100/100 [==============================] - 38s 371ms/step -
     loss: 1.7219 - accuracy: 0.3700
                                        - val_loss: 1.2672 -
     val_accuracy: 0.5763
Epoch 2/10
100/100 [==============================] - 37s 368ms/step -
     loss: 1.1791 - accuracy: 0.5756
                                        - val_loss: 0.9616 -
     val_accuracy: 0.6650
--snip--
Epoch 10/10
100/100 [==============================] - 39s 388ms/step -
     loss: 0.3897 - accuracy: 0.8639
                                        - val_loss: 0.4766 -
     val_accuracy: 0.8450
```

You train the model by passing the training dataset `train_ds` into `model.fit()`. You also specify the validation dataset `val_ds`, which is used to evaluate how accurate the model is. The training takes place over 10 *epochs*. During

each epoch, the complete set of training data is shown to the neural network. The data is randomly shuffled each time, so training across multiple epochs allows the model to learn better. You use the `callback` option ❶ to set up a function to exit the training if it turns out that the training loss is no longer decreasing with each epoch.

Running this Colab cell will take some time. The progress will be shown on the screen as the training is in process. Looking at the output, the `val_accuracy` listed under Epoch 10 shows that the model was about 85 percent accurate at running inference on the validation data by the end of the training. (The `val_accuracy` metric corresponds to the validation data, while `accuracy` corresponds to the training data.)

Now you can try the model by running inference on the testing portion of the data:

```
test_audio = []
test_labels = []

for audio, label in test_ds:
    test_audio.append(audio.numpy())
    test_labels.append(label.numpy())

test_audio = np.array(test_audio)
test_labels = np.array(test_labels)

y_pred = np.argmax(model.predict(test_audio), axis=1)
y_true = test_labels

test_acc = sum(y_pred == y_true) / len(y_true)
print(f'Test set accuracy: {test_acc:.0%}')
```

```
25/25 [==============================] - 1s 35ms/step
Test set accuracy: 84%
```

You first fill in two lists, `test_audio` and `test_labels`, by iterating through the test dataset `test_ds` ❶. Then you create

`numpy` arrays from these lists ❷ and run inference on the data ❸. You compute the test accuracy by summing up the number of times the predictions matched the true value and dividing them by the total number of items ❹. The output shows an accuracy of 84 percent. Not perfect, but good enough for this project.

Exporting the Model to the Pi

Congratulations! You have a fully trained machine learning model. Now you need to get it from Colab onto your Raspberry Pi. The first step is to save it:

```
model.save('audioml.sav')
```

This saves the model to a file on the cloud called *audioml.sav*. Next, convert that file to the TensorFlow Lite format so you can use it on your Pi:

```
converter =
 tf.lite.TFLiteConverter.from_saved_model('audioml.sav')
tflite_model = converter.convert()

with open('audioml.tflite', 'wb') as f:
    f.write(tflite_model)
```

You create a `TFLiteConverter` object, passing in the saved model filename ❶. Then you do the conversion ❷ and write the simplified TensorFlow model to a file called *audioml.tflite* ❸. Now you need to download this *.tflite* file from Colab onto your computer. Running the following snippet will give you a browser prompt to save the *.tflite* file:

```
from google.colab import files
files.download('audioml.tflite')
```

Once you have the file, you can transfer it to your Raspberry Pi using SSH as we've discussed in other chapters.

Using the Model on the Raspberry Pi

Now we'll turn our attention to the Raspberry Pi portion of the code. This code uses parallel processing to take in audio data from the microphone, prepare that data for your trained ML model, and show the data to the model to perform inference. As usual, you can write the code on your local machine and then transfer it to your Pi via SSH. To view the complete code, see "The Complete Code" on page 389. You can also download the code from *https://github.com/mkvenkit/pp2e/tree/main/audioml*.

Setting Up

Start by importing the required modules:

```
from scipy.io import wavfile
from scipy import signal
import numpy as np
import argparse
import pyaudio
import wave
import time
from tflite_runtime.interpreter import Interpreter
from multiprocessing import Process, Queue
```

Next, you initialize some parameters that are defined as global variables:

```
VERBOSE_DEBUG = False
CHUNK = 4000
```

```
FORMAT = pyaudio.paInt16
CHANNELS = 1
SAMPLE_RATE = 16000
RECORD_SECONDS = 1
NCHUNKS = int((SAMPLE_RATE * RECORD_SECONDS) / CHUNK)
ND = 2 * CHUNK * NCHUNKS
NDH = ND // 2
# device index of microphone
dev_index = -1
```

`VERBOSE_DEBUG` is a flag you'll use in many places in the code. For now, you set it to `False`, but if set to `True` (via a command line option), it will print out a lot of debugging information.

NOTE *I've omitted the `print()` statements for debugging from the code listings that follow. You can find them in the full code listing and on the book's GitHub repository.*

The next global variables are for working with the audio input. `CHUNK` sets the number of data samples read at a time using `PyAudio`, and `FORMAT` specifies that the audio data will consist of 16-bit integers. You set `CHANNELS` to `1`, since you'll be using a single-channel microphone, and `SAMPLE_RATE` to `16000` for consistency with the ML training data. `RECORD_SECONDS` indicates that you'll be grouping the audio into one-second increments (which you'll stitch together into overlapping two-second clips, as discussed earlier). You calculate the number of chunks in each one-second recording as `NCHUNKS`. You'll use `ND` and `NDH` to implement the overlapping technique—more on that later.

Finally, you initialize the device index number of the microphone to `-1` ❶. You'll need to update this value at the command line once you know your microphone's index. Here's a function to help you figure that out. You'll be able to call this function as a command line option.

```
def list_devices():
    """list pyaudio devices"""
    # initialize pyaudio
  ❶ p = pyaudio.PyAudio()
    # get device list
    index = None
  ❷ nDevices = p.get_device_count()
    print('\naudioml.py:\nFound the following input devices:')
  ❸ for i in range(nDevices):
        deviceInfo = p.get_device_info_by_index(i)
        if deviceInfo['maxInputChannels'] > 0:
            print(deviceInfo['index'], deviceInfo['name'],
                  deviceInfo['defaultSampleRate'])
    # clean up
  ❹ p.terminate()
```

You initialize `PyAudio` ❶ and get a count of the audio devices it detects ❷. Then you iterate through the devices ❸. For each one, you retrieve information about the device using `get_device_info_by_index()` and print out devices with one or more input channels—that is, microphones. You finish by cleaning up `PyAudio` ❹.

Here's what a typical output of the function looks like:

```
audioml.py:
Found the following input devices:
1 Mico: USB Audio (hw:3,0) 16000.0
```

This indicates there's an input device called Mico with a default sample rate of 16,000 and an index of `1`.

Taking In Audio Data

One of the main tasks for the Pi is to continuously take in the audio input from the microphone and break it up into

clips that you can run inference on. You create a `get_live_input()` function for this purpose. It takes in the `interpreter` object needed to work with the TensorFlow Lite model. Here's the start of the function:

```
def get_live_input(interpreter):
    # create a queue object
  ❶ dataq = Queue()
    # start inference process
  ❷ proc = Process(target = inference_process, args=(dataq,
  interpreter))
    proc.start()
```

As we discussed in "How It Works," you'll need to use separate processes for reading the audio data and doing the inference to avoid missing any input. You create a `multiprocessing.Queue` object that the processes will use to communicate with each other ❶ . Then you create the inference process using `multiprocessing.Process()` ❷ . You specify the name of the handler function for the process as `inference_process`, which takes the `dataq` and `interpreter` objects as arguments (we'll view this function later). You next start the process so the inference will run parallel to the data capture.

You continue the `get_live_input()` function by initializing `PyAudio`:

```
    # initialize pyaudio
  ❶ p = pyaudio.PyAudio()
    print('opening stream...')
  ❷ stream = p.open(format = FORMAT,
                    channels = CHANNELS,
                    rate = SAMPLE_RATE,
                    input = True,
                    frames_per_buffer = CHUNK,
                    input_device_index = dev_index)
```

```
  # discard first 1 second
❸ for i in range(0, NCHUNKS):
      data = stream.read(CHUNK, exception_on_overflow =
False)
```

You create a `PyAudio` object `p` ❶ and open an audio input stream ❷ , using some of your global variables as parameters. Then you discard the first one second of data ❸. This is to disregard any spurious data that comes in when the microphone is enabled for the first time.

Now you're ready to start reading the data:

```
  # count for gathering two frames at a time
❶ count = 0
❷ inference_data = np.zeros((ND,), dtype=np.int16)
  print("Listening...")
  try:
    ❸ while True:
        chunks = []
      ❹ for i in range(0, NCHUNKS):
          data = stream.read(CHUNK, exception_on_overflow
= False)
          chunks.append(data)
        # process data
        buffer = b''.join(chunks)
      ❺ audio_data = np.frombuffer(buffer, dtype=np.int16)
```

You initialize `count` to `0` ❶. You'll use this variable to keep track of the number of one-second frames of audio data read in. Then you initialize a 16-bit array `inference_data` with zeros ❷ . It has `ND` elements, which corresponds to two seconds of audio. You next enter a `while` loop to process the audio data continuously ❸. In it, you use a `for` loop ❹ to read in one second of audio data, one chunk at a time, appending those chunks to the list `chunks`. Once you have a full second of data, you convert it into a `numpy` array ❺.

Next, still within the `while` loop started in the previous listing, you implement the technique we discussed in "How It Works" to create overlapping two-second audio clips. You get help from your `NDH` global variable.

```
if count == 0:
    # set first half
  ❶ inference_data[:NDH] = audio_data
    count += 1
elif count == 1:
    # set second half
  ❷ inference_data[NDH:] = audio_data
    # add data to queue
  ❸ dataq.put(inference_data)
    count += 1
else:
    # move second half to first half
  ❹ inference_data[:NDH] = inference_data[NDH:]
    # set second half
  ❺ inference_data[NDH:] = audio_data
    # add data to queue
  ❻ dataq.put(inference_data)
```

The very first time a one-second frame is read in, it's stored in the first half of `inference_data` ❶. The next frame that comes in is stored in the second half of `inference_data` ❷. Now you have a full two seconds of audio data, so you put `inference_data` into the queue for the inference process to pick it up ❸. For every subsequent frame, the second half of the data is moved to the first half of `inference_data` ❹, the new data is set to the second half ❺, and `inference_data` is added to the queue ❻. This creates the desired one-second overlap between each consecutive two-second audio clip.

The `while` loop occurs inside a `try` block. To exit the loop, you just need to press `CTRL`-C and trigger the following `except` block:

```
    except KeyboardInterrupt:
        print("exiting...")
    stream.stop_stream()
    stream.close()
    p.terminate()
```

This `except` block performs some basic cleanup by stopping and closing the stream and by terminating `PyAudio`.

Preparing the Audio Data

Next, you'll create a few functions to prepare the audio data for inference. First is `process_audio_data()`, which takes in a raw two-second clip of audio data pulled from the queue and extracts the most interesting one second of audio from it, based on peak amplitude. We'll look at this function across several listings:

```
def process_audio_data(waveform):
    # compute peak to peak based on scaling by max 16-bit value
  ❶ PTP = np.ptp(waveform / 32768.0)
    # return None if too silent
  ❷ if PTP < 0.3:
        return []
```

You want to skip doing any inference on the microphone audio input if nobody is talking. There will always be some noise in the environment, however, so you can't simply look for the signal to be 0. Instead, you'll skip inference if the peak-to-peak amplitude (the difference between the highest value and the lowest value) of the audio is below a certain threshold. For this, you first divide the audio by `32768` to normalize it to a range of (−1, 1), and you pass the result to `np.ptp()` to get the peak-to-peak amplitude ❶ . The normalization makes it easier to express the threshold as a

fraction. You return an empty list (which will bypass the inference process) if the peak-to-peak amplitude is below `0.3` ❷ . You may need to adjust this threshold value depending on the noise level of your environment.

The `process_audio_data()` function continues with another technique for normalizing any audio data that won't be skipped:

```
    # normalize audio
    wabs = np.abs(waveform)
    wmax = np.max(wabs)
❶ waveform = waveform / wmax
    # compute peak to peak based on normalized waveform
❷ PTP = np.ptp(waveform)
    # scale and center
❸ waveform = 2.0*(waveform - np.min(waveform))/PTP – 1
```

When you normalized the data before skipping quiet audio samples, you divided the audio by 32,768, the maximum possible value of a 16-bit signed integer. In most cases, however, the peak amplitude of the audio data will be well below this value. Now you want to normalize the audio such that its maximum amplitude, whatever that may be, is scaled to 1. To do this, you first determine the peak amplitude in the audio signal and then divide the signal by that amplitude value ❶. Then you compute the new peak-to-peak value of the normalized audio ❷ and use this value to scale and center the data ❸ . Specifically, the expression `(waveform – np.min(waveform))/PTP` will scale the waveform values to the range (0, 1). Multiplying this by 2 and subtracting 1 will put the values in the range (−1, 1), which is what you need.

The next part of the function extracts one second of audio from the data:

```
    # extract 16000 len (1 second) of data
❶   max_index = np.argmax(waveform)
❷   start_index = max(0, max_index-8000)
❸   end_index = min(max_index+8000, waveform.shape[0])
❹   waveform = waveform[start_index:end_index]
    # padding for files with less than 16000 samples
    waveform_padded = np.zeros((16000,))
    waveform_padded[:waveform.shape[0]] = waveform
    return waveform_padded
```

You want to make sure you're getting the most interesting one second of the data, so you find the array index where the audio amplitude is at the maximum ❶. Then you try to grab 8,000 values before ❷ and after ❸ this index to get a full second of data, using `max()` and `min()` to ensure that the start and end indices don't fall out of range of the original clip. You use slicing to extract the relevant audio data ❹. Because of the `max()` and `min()` operations, you may end up with less than 16,000 samples, but the neural network strictly requires each input to be 16,000 samples long. To address this problem, you pad the data with zeros, using the same `numpy` techniques you saw during training. Then you return the result.

Figure 15-7 summarizes the `process_audio_data()` function by showing an example waveform at the various stages of processing.

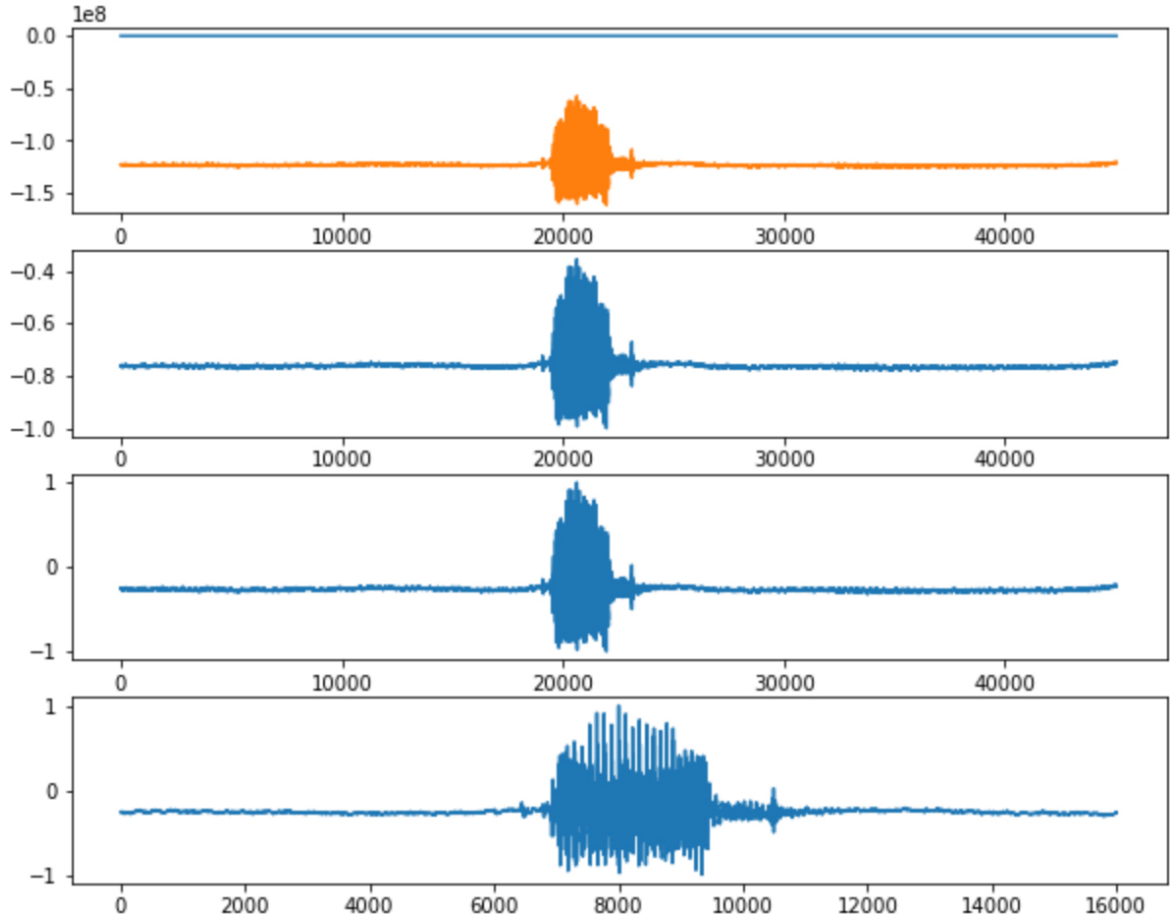


Figure 15-7: *The audio preparation process at various stages*

The top waveform in Figure 15-7 shows the unprocessed audio. The second waveform shows the audio with the values normalized to range (−1, 1). The third waveform shows the audio after a shift and scale—notice on the y-axis how the waveform now fills the entire (−1, 1) range. The fourth waveform consists of 16,000 samples extracted from the third one, centered on the peak amplitude.

Next, you need a `get_spectrogram()` function for computing the spectrogram of the audio data:

```
def get_spectrogram(waveform):
  ❶ waveform_padded = process_audio_data(waveform)
  ❷ if not len(waveform_padded):
        return []
```

```
    # compute spectrogram
  ❸ f, t, Zxx = signal.stft(waveform_padded, fs=16000,
  nperseg=255,
        noverlap = 124, nfft=256)
    # output is complex, so take abs value
  ❹ spectrogram = np.abs(Zxx)
    return spectrogram
```

You call your `process_audio_data()` function to prepare the audio ❶. If the function returns an empty list (because the audio is too quiet), `get_spectrogram()` returns an empty list as well ❷ . Next, you compute the spectrogram with `signal.stft()` from `scipy`, exactly as you did when training the model ❸. You then calculate the absolute value of the STFT ❹ to convert from complex numbers—again, as you did during training—and return the result.

Running Inference

The heart of this project is using your trained model to run inference on the incoming audio data and identify any spoken commands. Recall that this occurs in a separate process from the code for taking in audio data from the microphone. Here's the handler function that coordinates this process:

```
def inference_process(dataq, interpreter):
    success = False
    while True:
      ❶ if not dataq.empty():
            # get data from queue
          ❷ inference_data = dataq.get()
            # run inference only if previous one was not
  successful
          ❸ if not success:
                success = run_inference(inference_data,
  interpreter)
```

```
            else:
                # skipping, reset flag for next time
              ❹ success = False
```

The inference process runs continuously inside a `while`. Within this loop, you check if there's any data in the queue ❶, and if so, you retrieve it ❷. Then you run inference on it with the `run_inference()` function, which we'll look at next, but only if the `success` flag is `False` ❸. This flag keeps you from responding to the same speech command twice. Recall that because of the overlap technique, the second half of one audio clip will be repeated as the first half of the next clip. This lets you catch any audio commands that might be split across two frames, but it means that once you have a successful inference, you should skip the next element in the queue because it will have a portion of the audio from the previous element. When you do a skip like this, you reset `success` to `False` ❹ to start running inference again on the next piece of data that comes in.

Now let's look at the `run_inference()` function, where the inference is actually carried out:

```
def run_inference(waveform, interpreter):
    # get spectrogram data
  ❶ spectrogram = get_spectrogram(waveform)
    if not len(spectrogram):
        return False
    # get input and output tensors details
  ❷ input_details = interpreter.get_input_details()
  ❸ output_details = interpreter.get_output_details()
```

The function takes in the raw audio data (`waveform`) for interacting with your TensorFlow Lite model (`interpreter`). You call `get_spectrogram()` to process the audio and generate the spectrogram ❶, and if the audio was too quiet, you

return `False`. Then you get the input ❷ and output ❸ details from the TensorFlow Lite interpreter. These tell you what the model is expecting as input and what you can expect from it as output. This is what `input_details` looks like:

```
[{'name': 'serving_default_input_5:0', 'index': 0, 'shape':
    array([1, 129, 124,   1]),
  'shape_signature': array([ -1, 129, 124,   1]), 'dtype':
    <class 'numpy.float32'>,
  'quantization': (0.0, 0), 'quantization_parameters':
    {'scales': array([], dtype=float32),
  'zero_points': array([], dtype=int32), 'quantized_dimension':
    0}, 'sparsity_parameters': {}}]
```

Notice that `input_details` is an array with a dictionary inside it. The `'shape'` entry is especially of interest: `array([1, 129, 124, 1])`. You've already ensured that your spectrogram, which will be the input to the interpreter, is shaped to this value. The `'index'` entry is just the index of the tensor in the tensor list inside the interpreter, and `'dtype'` is the expected data type of the input, which in this case is `float32`, a signed 32-bit float. You'll need to reference both `'index'` and `'dtype'` later in the `run_inference()` function.

Here's `output_details`:

```
[{'name': 'StatefulPartitionedCall:0', 'index': 17, 'shape':
    array([1, 8]), 'shape_signature':
  array([-1,  8]), 'dtype': <class 'numpy.float32'>,
    'quantization': (0.0, 0),
  'quantization_parameters': {'scales': array([],
    dtype=float32), 'zero_points':
  array([], dtype=int32), 'quantized_dimension': 0},
    'sparsity_parameters': {}}]
```

Notice the `'shape'` entry in this dictionary. It shows that the output will be an array of shape (1, 8). The shape

corresponds to the label IDs of the eight speech commands.

You continue the `run_inference()` function by actually running inference on the input data:

```
    # set input
❶ input_data = spectrogram.astype(np.float32)
❷ interpreter.set_tensor(input_details[0]['index'],
input_data)

    # run interpreter
    print("running inference...")
❸ interpreter.invoke()
    # get output
❹ output_data = interpreter.get_tensor(output_details[0]
['index'])
❺ yvals = output_data[0]
    print(yvals)
```

First you convert the spectrogram data to 32-bit floating point values ❶. Recall that your audio data started as 16-bit integers. The scaling and other processing operations converted the data to 64-bit floats, but as you saw in `input_details`, the TensorFlow Lite model requires 32-bit floats, which is the reason for the conversion. You next set the input value to the appropriate tensor inside the interpreter ❷. Here the `[0]` accesses the first (and only) element in `input_details`, which as you saw is a dictionary, and `['index']` retrieves the value under that key in the dictionary to specify which tensor you're setting. You run inference on the input using the `invoke()` method ❸. Then you retrieve the output tensor using similar indexing to the input ❹ and get the output itself by extracting the first element from the `output_data` array ❺. (Since you provided only one input, you expect only one output.) Here's an example of what `yvals` looks like:

```
[  6.640185  -26.032831  -26.028618  8.746256  62.545185  -0.56
     98182  -15.045679  -29.140179 ]
```

These eight numbers correspond to the eight commands you trained the model with. The values indicate the likelihood of the input data being each word. In this particular array, the value at index `4` is by far the largest, so that's what the neural network is predicting as the most probable answer. Here's how you interpret the result:

```
    # Important! This should exactly match training labels/ids.
    commands = ['up', 'no', 'stop', 'left', 'right', 'go',
  'down', 'yes']
    print(">>> " + commands[np.argmax(output_data[0])].upper())
```

You define a `commands` list in the same order as you used during training. It's important to keep the order consistent across training and inference, or you'll end up misinterpreting the results! Then you use `np.argmax()` to get the index of the highest value in the output data and use that index to pick up the corresponding string from `commands`.

Writing the main() Function

Now let's look at the `main()` function, which brings everything together:

```
def main():
    # globals set in this function
    global VERBOSE_DEBUG
    # create parser
    descStr = "This program does ML inference on audio data."
    parser = argparse.ArgumentParser(description=descStr)
    # add a mutually exclusive group
  ❶ group = parser.add_mutually_exclusive_group(required=True)
    # add mutually exclusive arguments
```

```
❷ group.add_argument('--list', action='store_true',
required=False)
❸ group.add_argument('--input', dest='wavfile_name',
required=False)
❹ group.add_argument('--index', dest='index',
required=False)
  # add other arguments
❺ parser.add_argument('--verbose', action='store_true',
required=False)
  # parse args
  args = parser.parse_args()
```

You start by setting `VERBOSE_DEBUG` as a global, since you'll be setting it in this function and don't want it to be treated as a local variable. Then you create a familiar `argparse.ArgumentParser` object and add a mutually exclusive group to the parser ❶, since some of your command line options won't be compatible with each other. Those are the `--list` option ❷, which will list all the `PyAudio` devices so you can get your microphone's index number; the `--input` option ❸, which lets you specify a WAV file to use as input instead of live data from the microphone (useful for testing); and the `--index` option ❹, which starts capturing audio and running inference using the microphone with the specified index. You also add the non–mutually exclusive `--verbose` option ❺ to print out detailed debug information as the program is run.

Next, you create the TensorFlow Lite interpreter so you can use the ML model:

```
  # load TF Lite model
  interpreter = Interpreter('audioml.tflite')
  interpreter.allocate_tensors()
```

Here you create an `Interpreter` object, passing it the *audioml.tflite* file with the model you created during

training. Then you call `allocate_tensors()` to prepare the necessary tensors for running the inference.

The `main()` function finishes with branches for the different command line arguments:

```
    # check verbose flag
    if args.verbose:
        VERBOSE_DEBUG = True
    # test WAV file
    if args.wavfile_name:
      ❶ wavfile_name = args.wavfile_name
        # get audio data
      ❷ rate, waveform = wavfile.read(wavfile_name)
        # run inference
      ❸ run_inference(waveform, interpreter)
    elif args.list:
        # list devices
      ❹ list_devices()
    else:
        # store device index
      ❺ dev_index = int(args.index)
        # get live audio
      ❻ get_live_input(interpreter)
    print("done.")
```

If the `--input` command line option is used, you get the name of the WAV file ❶ and read its contents ❷. The resulting data is passed along for inference ❸. If the `--list` option is used, you call your `list_devices()` function ❹. If the `--index` option is used, you parse the device index ❺ and start processing live audio by calling the `get_live_input()` function ❻.

Running the Speech Recognition System

To run the project, gather your Python code and the *audioml.tflite* file into a folder on your Pi. For testing, you can also download the *right.wav* file from the book's GitHub repository and add that to the folder. You can work with your Pi via SSH, as explained in Appendix B.

First, try using the `--input` command line option to run inference on a WAV file:

```
$ sudo python audioml.py --input right.wav
```

Here's the output:

```
running inference...
[  6.640185  -26.032831  -26.028618    8.746256   62.545185   -
  0.5698182
 -15.045679  -29.140179 ]
>>> RIGHT
run_inference: 0.029174549999879673s
done.
```

Notice that the program has correctly identified the *right* command recorded on the WAV file ❶.

Now plug your microphone into the Pi and use the `--list` option to determine its index number, as shown here:

```
$ sudo python audioml.py --list
```

Your output should be similar to the following:

```
audioml.py:
Found the following input devices:
```

```
1 Mico: USB Audio (hw:3,0) 16000.0
done.
```

In this example, the microphone has index `1`. Use that number to run the `--index` command to do some live speech detection! Here's an example run:

```
$ sudo python audioml.py --index 1
--snip--
opening stream...
Listening...
running inference...
[-2.647918    0.17592785
  -3.3615346   6.6812882   4.472283   -3.7535028
  1.2349942   1.8546474 ]
>>> LEFT
run_inference: 0.03520956500142347s
running inference...
[-2.7683923 -5.9614644
  -8.532391   6.906795  19.197264  -4.0255833
  1.7236844 -4.374415 ]
>>> RIGHT
run_inference: 0.03026762299850816s
--snip--
^C
KeyboardInterrupt
exiting...
done.
```

After starting the program and getting the "Listening . . ." prompt, I spoke the words *left* and *right*. The output at ❶ and ❷ indicates that the program was able to identify the commands correctly.

Try running the program with the `--verbose` option to see more information about how it's working. Also, try speaking different commands in quick succession to verify whether

the multiprocessing and overlapping techniques are working.

Summary

This chapter introduced you to the world of machine learning. You learned how to train a deep neural network to recognize speech commands using the TensorFlow framework, and you converted the resulting model to a TensorFlow Lite format for use on a resource-constrained Raspberry Pi. You also learned about spectrograms and the importance of processing input data before ML training. You practiced using Python multiprocessing, reading USB microphone input on a Raspberry Pi using `PyAudio`, and running a TensorFlow Lite interpreter for ML inference.

Experiments!

1. Now that you know how to process speech commands on a Raspberry Pi, you can build an assistive device that responds to those commands by doing more than printing out the identified words. For example, you could use the commands *left*, *right*, *up*, *down*, *stop*, and *go* to control a camera (or laser!) mounted on a pan/tilt mount. Hint: you'll need to retrain the ML model with just these six commands. You'll also need to get a two-axis pan/tilt bracket with two servo motors attached. The servos will be connected to the Raspberry Pi and controlled based on the inference results.
2. Read about the *mel spectrogram*, a variant of the spectrogram you used for this project that's better suited for human speech data.

3. Try modifying the neural network by adding or removing some layers. For example, remove the second Conv2D layer. See how the changes affect the training accuracy of the model and the inference accuracy on the Pi.
4. This project used an ad hoc neural network, but there are also pretrained neural networks available that you could leverage. For example, read up on MobileNet V2. What changes are needed to adapt your project to use this network instead?

The Complete Code

Here's a complete listing of the code that goes on the Raspberry Pi, including the `print()` statements for verbose debugging. The Google Colab notebook code can be found at *https://github.com/mkvenkit/pp2e/blob/main/audioml/audioml.ipynb*.

```
"""
    simple_audio.py

    This programs collects audio data from an I2S mic on the
  Raspberry Pi
    and runs the TensorFlow Lite interpreter on a per-build
  model.

    Author: Mahesh Venkitachalam
"""

from scipy.io import wavfile
from scipy import signal
import numpy as np
import argparse
import pyaudio
import wave
```

```
import time

from tflite_runtime.interpreter import Interpreter
from multiprocessing import Process, Queue

VERBOSE_DEBUG = False
CHUNK = 4000                # choose a value divisible by
  SAMPLE_RATE
FORMAT = pyaudio.paInt16
CHANNELS = 1
SAMPLE_RATE = 16000
RECORD_SECONDS = 1
NCHUNKS = int((SAMPLE_RATE * RECORD_SECONDS) / CHUNK)
ND = 2 * SAMPLE_RATE * RECORD_SECONDS
NDH = ND // 2
# device index of microphone
dev_index = -1

def list_devices():
    """list pyaudio devices"""
    # initialize pyaudio
    p = pyaudio.PyAudio()
    # get device list
    index = None
    nDevices = p.get_device_count()
    print('\naudioml.py:\nFound the following input devices:')
    for i in range(nDevices):
        deviceInfo = p.get_device_info_by_index(i)
        if deviceInfo['maxInputChannels'] > 0:
            print(deviceInfo['index'], deviceInfo['name'],
  deviceInfo['defaultSampleRate'])
    # clean up
    p.terminate()

def inference_process(dataq, interpreter):
    """infererence process handler"""
    success = False
    while True:
        if not dataq.empty():
            # get data from queue
            inference_data = dataq.get()
```

```
            # run inference only if previous one was not
  success
            # otherwise we will get duplicate results because
  of
            # overlap in input data
            if not success:
                success = run_inference(inference_data,
  interpreter)
            else:
                # skipping, reset flag for next time
                success = False

def process_audio_data(waveform):
    """Process audio input.
    This function takes in raw audio data from a WAV file and
  does scaling
    and padding to 16000 length.
    """

    if VERBOSE_DEBUG:
        print("waveform:", waveform.shape, waveform.dtype,
  type(waveform))
        print(waveform[:5])

    # compute peak to peak based on scaling by max 16-bit value
    PTP = np.ptp(waveform / 32768.0)

    if VERBOSE_DEBUG:
        print("peak-to-peak (16 bit scaling): {}".format(PTP))

    # return None if too silent
    if PTP < 0.3:
        return []

    # normalize audio
    wabs = np.abs(waveform)
    wmax = np.max(wabs)
    waveform = waveform / wmax

    # compute peak to peak based on normalized waveform
    PTP = np.ptp(waveform)

    if VERBOSE_DEBUG:
        print("peak-to-peak (after normalize): {}".format(PTP))
```

```
        print("After normalization:")
        print("waveform:", waveform.shape, waveform.dtype,
  type(waveform))
        print(waveform[:5])

    # scale and center
    waveform = 2.0*(waveform - np.min(waveform))/PTP - 1

    # extract 16000 len (1 second) of data
    max_index = np.argmax(waveform)
    start_index = max(0, max_index-8000)
    end_index = min(max_index+8000, waveform.shape[0])
    waveform = waveform[start_index:end_index]

    # padding for files with less than 16000 samples
    if VERBOSE_DEBUG:
        print("After padding:")

    waveform_padded = np.zeros((16000,))
    waveform_padded[:waveform.shape[0]] = waveform

    if VERBOSE_DEBUG:
        print("waveform_padded:", waveform_padded.shape,
               waveform_padded.dtype, type(waveform_padded))
        print(waveform_padded[:5])

    return waveform_padded

def get_spectrogram(waveform):
    """computes spectrogram from audio data"""

    waveform_padded = process_audio_data(waveform)

    if not len(waveform_padded):
        return []

    # compute spectrogram
    f, t, Zxx = signal.stft(waveform_padded, fs=16000,
  nperseg=255,
        noverlap = 124, nfft=256)
    # output is complex, so take abs value
    spectrogram = np.abs(Zxx)

    if VERBOSE_DEBUG:
        print("spectrogram:", spectrogram.shape,
  type(spectrogram))
```

```
        print(spectrogram[0, 0])

    return spectrogram

def run_inference(waveform, interpreter):
    # start timing
    start = time.perf_counter()

    # get spectrogram data
    spectrogram = get_spectrogram(waveform)

    if not len(spectrogram):
        if VERBOSE_DEBUG:
            print("Too silent. Skipping...")
        return False

    if VERBOSE_DEBUG:
        print("spectrogram: %s, %s, %s" % (type(spectrogram),
              spectrogram.dtype, spectrogram.shape))

    # get input and output tensors details
    input_details = interpreter.get_input_details()
    output_details = interpreter.get_output_details()

    if VERBOSE_DEBUG:
        print("input_details: {}".format(input_details))
        print("output_details: {}".format(output_details))

    # reshape spectrogram to match interpreter requirement
    spectrogram = np.reshape(spectrogram, (-1,
  spectrogram.shape[0],
                                           spectrogram.shape[1]
  , 1))

    # set input
    input_data = spectrogram.astype(np.float32)
    interpreter.set_tensor(input_details[0]['index'],
  input_data)

    # run interpreter
    print("running inference...")
    interpreter.invoke()

    # get output
    output_data = interpreter.get_tensor(output_details[0]
  ['index'])
```

```
    yvals = output_data[0]
    if VERBOSE_DEBUG:
        print(output_data)

    print(yvals)

    # Important! This should exactly match training labels/ids.
    commands = ['up', 'no', 'stop', 'left', 'right', 'go',
  'down', 'yes']
    print(">>> " + commands[np.argmax(output_data[0])].upper())

    # stop timing
    end = time.perf_counter()
    print("run_inference: {}s".format(end - start))
    # return success
    return True

def get_live_input(interpreter):
    """this function gets live input from the microphone
    and runs inference on it"""

    # create a queue object
    dataq = Queue()
    # start inference process
    proc = Process(target = inference_process, args=(dataq,
  interpreter))
    proc.start()

    # initialize pyaudio
    p = pyaudio.PyAudio()

    print('opening stream...')
    stream = p.open(format = FORMAT,
                    channels = CHANNELS,
                    rate = SAMPLE_RATE,
                    input = True,
                    frames_per_buffer = CHUNK,
                    input_device_index = dev_index)

    # discard first 1 second
    for i in range(0, NCHUNKS):
        data = stream.read(CHUNK, exception_on_overflow =
  False)
```

```
  # count for gathering two frames at a time
  count = 0
  inference_data = np.zeros((ND,), dtype=np.int16)
  print("Listening...")
  try:
      while True:
          # print("Listening...")

          chunks = []
          for i in range(0, NCHUNKS):
              data = stream.read(CHUNK, exception_on_overflow
= False)
              chunks.append(data)

          # process data
          buffer = b''.join(chunks)
          audio_data = np.frombuffer(buffer, dtype=np.int16)

          if count == 0:
              # set first half
              inference_data[:NDH] = audio_data
              count += 1
          elif count == 1:
              # set second half
              inference_data[NDH:] = audio_data
              # add data to queue
              dataq.put(inference_data)
              count += 1
          else:
              # move second half to first half
              inference_data[:NDH] = inference_data[NDH:]
              # set second half
              inference_data[NDH:] = audio_data
              # add data to queue
              dataq.put(inference_data)

          # print("queue: {}".format(dataq.qsize()))

  except KeyboardInterrupt:
      print("exiting...")

  stream.stop_stream()
```

```
    stream.close()
    p.terminate()

def main():
    """main function for the program"""
    # globals set in this function
    global VERBOSE_DEBUG
    # create parser
    descStr = "This program does ML inference on audio data."
    parser = argparse.ArgumentParser(description=descStr)
    # add a mutually exclusive group
    group = parser.add_mutually_exclusive_group(required=True)
    # add mutually exclusive arguments
    group.add_argument('--list', action='store_true',
  required=False)
    group.add_argument('--input', dest='wavfile_name',
  required=False)
    group.add_argument('--index', dest='index', required=False)
    # add other arguments
    parser.add_argument('--verbose', action='store_true',
  required=False)

    # parse args
    args = parser.parse_args()

    # load TF Lite model
    interpreter = Interpreter('audioml.tflite')
    interpreter.allocate_tensors()

    # check verbose flag
    if args.verbose:
        VERBOSE_DEBUG = True

    # test WAV file
    if args.wavfile_name:
        wavfile_name = args.wavfile_name
        # get audio data
        rate, waveform = wavfile.read(wavfile_name)
        # run inference
        run_inference(waveform, interpreter)
    elif args.list:
        # list devices
```

```
        list_devices()
    else:
        # store device index
        dev_index = int(args.index)
        # get live audio
        get_live_input(interpreter)

    print("done.")

# main method
if __name__ == '__main__':
    main()
```

A
PYTHON INSTALLATION



This appendix covers how to install Python, as well as the external modules and code used in the book. Installation for the hardware projects using Raspberry Pi is covered in Appendix B. The projects in this book have been tested with Python 3.9.

Installing Source Code for the Book's Projects

You can download source code for the book's projects from *https://github.com/mkvenkit/pp2e*. Use the Download ZIP option at this site to retrieve the code.

Once you download and extract the code, you need to add the path to the *common* folder in the downloaded code (generally *pp-master/common/*) to your `PYTHONPATH` environment variable so that modules can find and use these Python files.

On Windows, you can do this by creating a `PYTHONPATH` environment variable or adding to one if it already exists. On macOS, you can add this line to your *.profile* file in your home directory (or create a file there if needed):

```
export PYTHONPATH=$PYTHONPATH:path_to_common_folder
```

Fill in the path to the common folder as appropriate.

Linux users can do something similar to macOS, in their *.bashrc*, *.bash_profile*, or *.cshrc/.login* files as appropriate. Use the `echo $SHELL` command to see the default shell.

Installing Python and Python Modules

I recommend using the Anaconda distribution of Python to run the book's projects, since it already comes with most of the Python modules you'll need. This section covers the installation process on Windows, macOS, and Linux.

Windows

Visit *https://www.anaconda.com* and download Anaconda Distribution for Windows. Once the installation is complete, bring up the Anaconda prompt (type **Anaconda prompt** in the search bar), which you'll use to run your programs. Just `cd` into the book's code directory and you're ready to go.

It's also useful to add the location of Anaconda and its supporting files into your `Path` environment variable (type **Edit Environment Variables** in the search bar):

```
C:\Users\mahes\anaconda3
C:\Users\mahes\anaconda3\Scripts
C:\Users\mahes\anaconda3\Library\bin
```

In this case, my Anaconda installation directory was `C:\Users\mahes\anaconda3\`. Modify this as necessary.

Installing GLFW

For the OpenGL-based 3D graphics projects in this book, you need the GLFW library, which you can download at *http://www.glfw.org*. On Windows, after you install GLFW, set a `GLFW_LIBRARY` environment variable (type **Edit Environment Variables** in the search bar) to the full path of the installed *glfw3.dll* so that your Python binding for GLFW can find this library. The path will look something like *C:\glfw-3.0.4.bin.WIN32\lib-msvc120\glfw3.dll*.

To use GLFW with Python, you use a module called `pyglfw`, which consists of a single Python file called *glfw.py*. You don't need to install `pyglfw` because it comes with the source code for the book; it's in the *common* directory. Just in case you need to install a more recent version, here is the source: *https://github.com/rougier/pyglfw*.

You also need to ensure that your graphics card drivers are installed on your computer. This is a good thing in general since many programs (especially games) make use of the graphics processing unit (GPU).

Installing Additional Modules

You need to install a few additional modules that aren't part of the standard Anaconda Distribution. Run the following commands at the Anaconda prompt:

```
conda install -c anaconda pyaudio
conda install -c anaconda pyopengl
```

macOS

Visit *https://www.anaconda.com* and download Anaconda Distribution for macOS. Once it's installed, bring up a Terminal application window and enter **`which python`**. The

output should point to the version of Anaconda Python. If not, add the path manually to your *.profile* file. For example:

```
export PYTHONPATH=your_anaconda_install_dir_path:$PYTHONPATH
```

Fill in the path to your Anaconda installation directory as appropriate.

Installing GLFW

For the OpenGL-based 3D graphics projects in this book, you need the GLFW library, which you can download at *http://www.glfw.org*. Choose the macOS **Pre-compiled Binaries** option and copy the downloaded folder to your *Home* folder. In my case, for example, my *Home* folder is */Users/mahesh/*.

Now you need to add the following to your *.profile*, making changes to your path as appropriate:

```
export GLFW_LIBRARY=/Users/mahesh/glfw-3.3.8.bin.MACOS/lib-
    universal/libglfw.3.dylib
```

You may get a security warning when you first try to run a program that uses GLFW. You need to allow it to run in System Preferences under Security.

Installing Additional Modules

Next, you need to install some additional modules that aren't part of the standard Anaconda Distribution. In a Terminal window, run the following commands:

```
conda install -c anaconda pyaudio
conda install -c anaconda pyopengl
```

Linux

Linux usually comes with Python, as well as all the development tools needed to build the required packages, built in. So you don't need to install Anaconda Python. On most Linux distributions, you should be able to use `pip3` to get the packages required for the book. You can install a package using `pip3` like this:

```
sudo pip3 install matplotlib
```

The other way to install a package is to download the module source distribution for it, which is usually in a *.gz* or *.zip* file. Once you unzip these files into a folder, you can then install them as follows:

```
sudo python setup.py install
```

Use one of these methods for each package needed for the book.

B
RASPBERRY PI SETUP



This appendix covers setting up a Raspberry Pi so you can use it for the projects in Chapters 13, 14, and 15. The projects work with a Raspberry Pi 3 Model B+ or Raspberry Pi 4 Model B. The setup instructions are the same for both. In addition to the board, you'll need a compatible power supply and a micro SD card with a 16GB capacity or higher.

Setting Up the Software

There are several ways to set up your Pi. These steps outline one of the simplest methods, using Raspberry Pi Imager:

1. Download the Raspberry Pi Imager from the Raspberry Pi website at *https://www.raspberrypi.com/software*.
2. Insert your SD card into your computer. (Depending on your system, you may need a micro SD card adapter for this.)
3. Open Pi Imager and click the **Choose OS** button. Figure B-1 shows the resulting dialog.

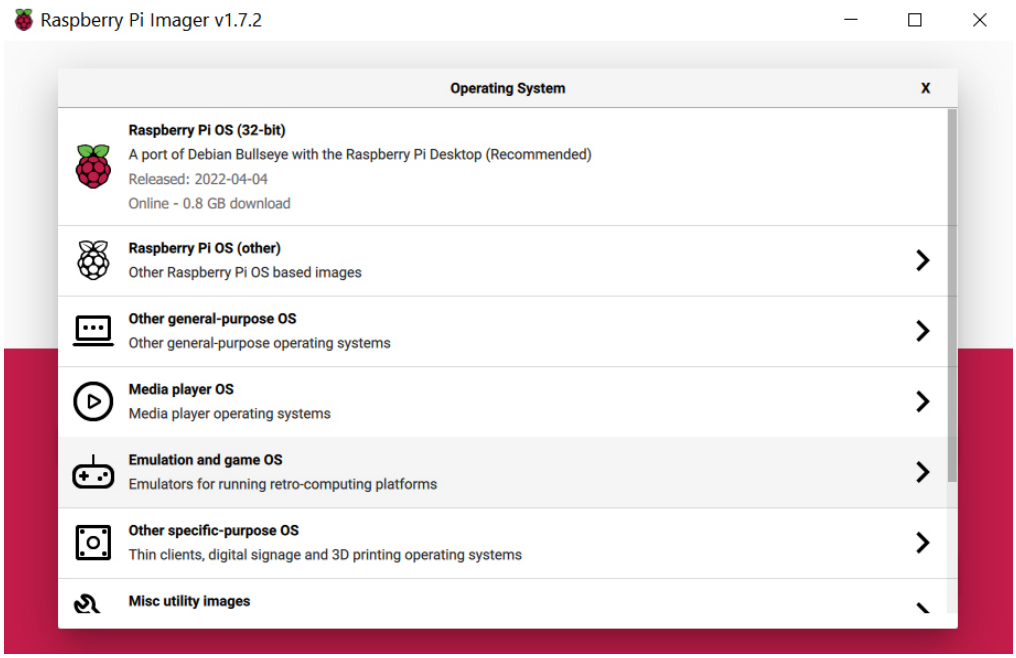


Figure B-1: *The Choose OS dialog in Raspberry Pi Imager*

4. Click the **Raspberry Pi OS** option.
5. Click the **Choose Storage** button. You should get a screen like the one in Figure B-2.

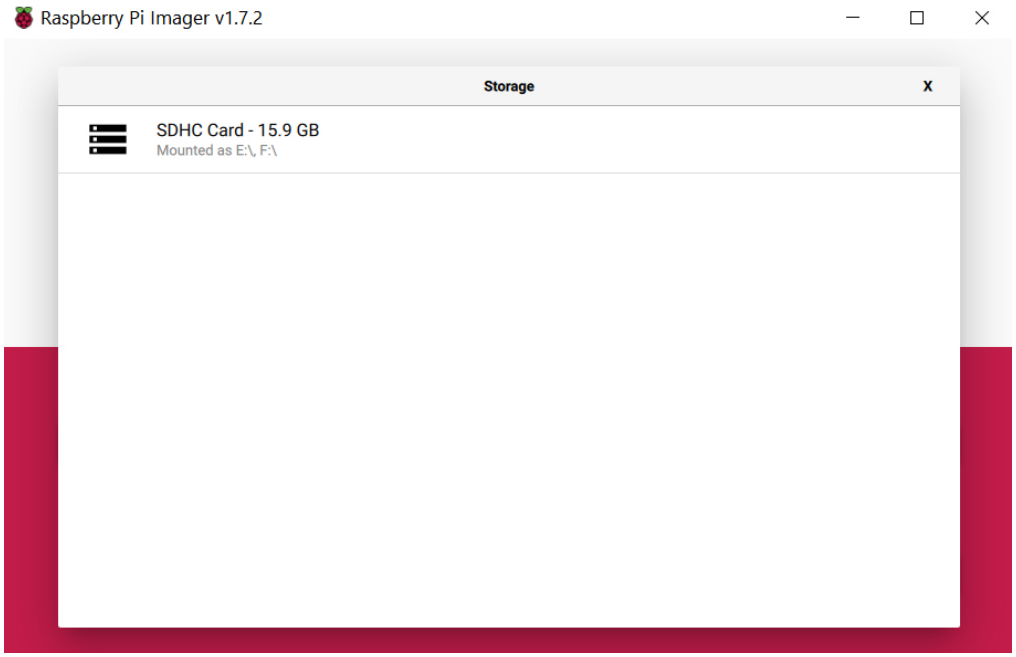


Figure B-2: *The Choose Storage dialog in Raspberry Pi Imager*

6. The screen should list your SD card. Click it.
7. Click the gear icon to open the Advanced Options dialog, as shown in Figure B-3.

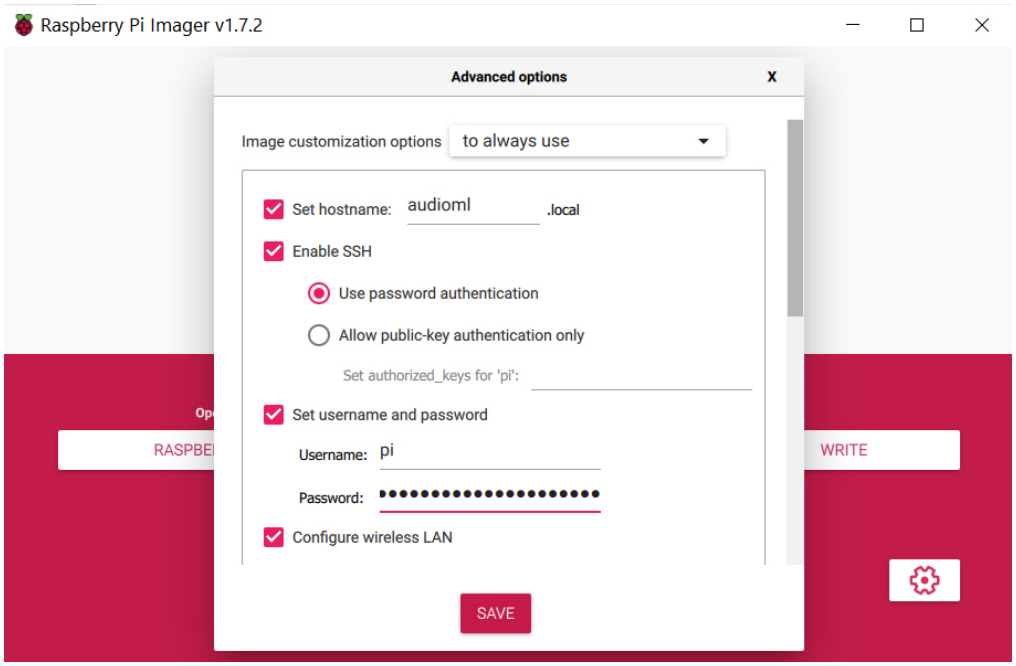


Figure B-3: *The Advanced Options dialog in Raspberry Pi Imager*

8. Enter a name for your Pi in the Set Hostname box. I've set the name to `audioml` in Figure B-3. Thanks to a service called Avahi that's enabled on the Raspberry Pi OS installation by default, you'll be able to reach your Pi over the local network by appending `.local` to the device name you choose—for example, `audioml.local`. This is much more convenient than remembering and using an IP address.
9. In the same dialog, set your username and password, and enable SSH. Then scroll down to see the Wi-Fi connection options, as shown in Figure B-4.

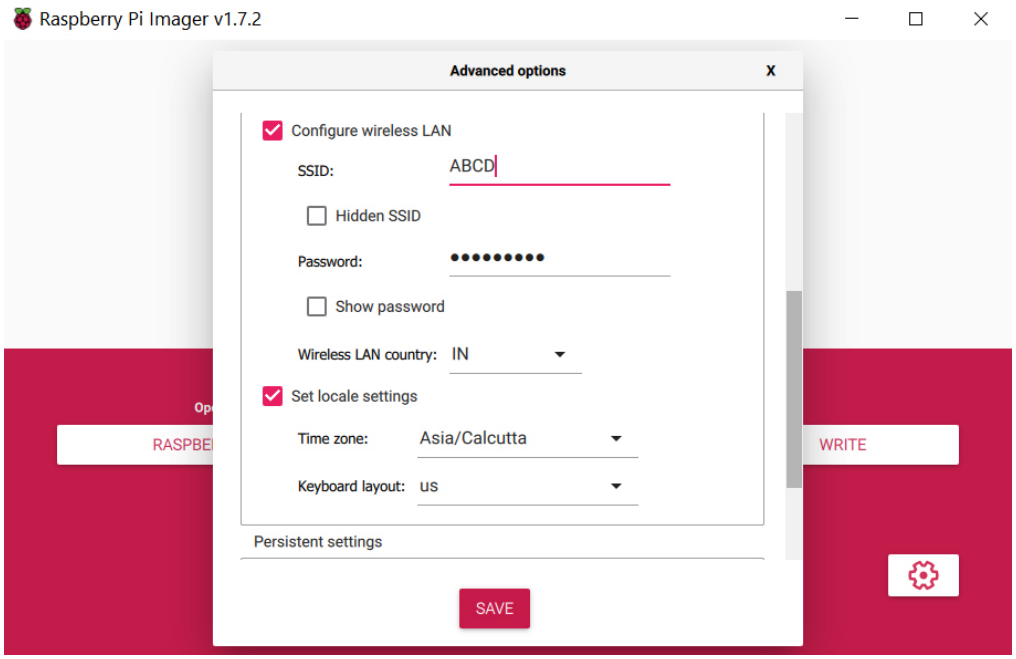


Figure B-4: *The Wi-Fi details in Raspberry Pi Imager*

10. Enter your Wi-Fi details, similar to what's shown in Figure B-4. Once you're done, click **Save**, and then click the **Write** button to write all this information to your SD card.
11. When the SD card is ready, insert it into your Pi. Then boot up your Pi and it will automatically connect to your Wi-Fi network.

Now you should be able to log in to your Pi remotely using Secure Shell (SSH), as we'll discuss soon.

Testing Your Connection

To check whether your Pi is connected to the local network, enter **`ping`** at the command line on your computer, followed by your Pi's device name. For example, here's what a `ping` session looks like from a Windows command shell:

```
$ ping audioml.local

Pinging audioml.local [fe80::e3e0:1223:9b20:2d6f%6] with 32
  bytes of data:
Reply from fe80::e3e0:1223:9b20:2d6f%6: time=66ms
Reply from fe80::e3e0:1223:9b20:2d6f%6: time=3ms
Reply from fe80::e3e0:1223:9b20:2d6f%6: time=2ms
Reply from fe80::e3e0:1223:9b20:2d6f%6: time=3ms

Ping statistics for fe80::e3e0:1223:9b20:2d6f%6:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 2ms, Maximum = 66ms, Average = 18ms
```

This `ping` output shows the number of bytes sent and the time it took to get a reply. If you see the message `Request timeout...` instead, then you know your Pi isn't connected to the network. In this case, try searching the internet for troubleshooting strategies. On a Windows computer, for example, you might try opening a command prompt as an administrator and entering the **`arp -d`** command. This clears the ARP cache. (ARP is a protocol for detecting other computers over a network.) Then try the `ping` command again. If that fails, it might be a good idea to hook up a monitor and keyboard to your Pi to check if it's really able to connect to the internet.

Logging in to Your Pi with SSH

You can hook up a keyboard, mouse, and monitor to your Pi to work with it directly, but for the purposes of this book, the most convenient way to work is to use SSH to log in to your Pi remotely from your desktop or laptop computer. If you do this not only frequently but also from the same computer, you'll probably find it annoying to enter the password every single time. With the `ssh-keygen` utility that comes with SSH,

you can set up a public/private key scheme so you can securely log in to your Pi without entering the password. For macOS and Linux users, follow the next procedure. (For Windows users, PuTTY lets you do something similar. Search for “Generating an SSH key with PuTTY” to learn more.)

1. From a terminal on your computer, enter the following to generate a key file:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/xxx/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/xxx/.ssh/id_rsa.
Your public key has been saved in /Users/xxx/.ssh/id_rsa.pub.
The key fingerprint is:
--snip--
```

2. Copy the key file to the Pi. You can use the `scp` command for this, which is part of SSH. Enter the following, replacing your Pi’s IP address as appropriate:

```
$ scp ~/.ssh/id_rsa.pub pi@192.168.4.32:.ssh/
The authenticity of host '192.168.4.32 (192.168.4.32)' can't
be established.
RSA key fingerprint is
f1:ab:07:e7:dc:2e:f1:37:1b:6f:9b:66:85:2a:33:a7.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.4.32' (RSA) to the list of
known hosts.
pi@192.168.4.32's password:
id_rsa.pub                                    100%  398     0.4K
B/s   00:00
```

3. Log in to the Pi and verify the key file was copied over, again substituting your Pi's IP address:

```
$ ssh pi@192.168.4.32
pi@192.168.4.32's password:

$ cd .ssh
$ ls
id_rsa.pub  known_hosts

$ cat id_rsa.pub >> authorized_keys
$ ls
authorized_keys  id_rsa.pub  known_hosts
$ logout
```

The next time you log in to the Pi, you won't be asked for a password. Also, note that I used an empty passphrase in `ssh-keygen` in this example, which isn't secure. This setup may be fine for Raspberry Pi hardware projects in which you aren't very concerned about security, but you may want to consider using a passphrase.

Installing Python Modules

Most of the Python modules you need for the projects in Chapters 13, 14, and 15 are already part of the Raspberry Pi installation. For the rest, install them by running the following commands one by one after you SSH into your Pi:

```
$ sudo pip3 install bottle
$ sudo apt install python3-matplotlib
$ sudo apt-get install python3-scipy
$ sudo apt-get install python3-pyaudio
$ sudo pip3 install tflite-runtime
```

This should get you going with all the projects that use the Raspberry Pi in the book.

Working Remotely with Visual Studio Code

Once you have SSH access to your Pi, you could edit your source code on your computer and transfer it to the Pi via the `scp` command, but this gets cumbersome quickly. There's a better way. Visual Studio Code (VS Code) is a popular code editor from Microsoft. This software supports a huge number of plug-ins or extensions to enhance its capabilities. One of them, the Visual Studio Code Remote - SSH extension, will let you connect to your Pi and edit your files directly from your computer. You can find the installation details for this extension at *https://code.visualstudio.com/docs/remote/ssh*.

INDEX

Symbols

2D slice, 216, 220
3D graphics pipeline, 154
3D printing, 289
3D texture, 217
3D transformations, 156–158
8-bit image, 103
16-bit data, 65, 69
`_bleio`, 322
`%` (modulus operator), 51

A

Adafruit
  `adafruit_ble` library
    `Advertisement` class, 321
    `BLE_ADV_INT`, 323
    `BLERadio`, 322
    `LazyObjectField`, 322
    `ManufacturerDataField`, 322
    `start_advertising()`, 323
    `stop_advertising()`, 324
  `adafruit_bmp280`, 322
  `adafruit_sht31d`, 322
  Feather Bluefruit nRF52840, 318
amplitude, 63
anonymous function, 374
Arduino sketch, 293
`argparse` module, 33, 52, 72
  `add_argument()`, 33, 199, 301, 340, 385
  `ArgumentParser`, 33, 94, 107, 126, 145, 199, 301, 340, 385
    `add_mutually_exclusive_group()`, 385
    `parse_args()`, 33, 199, 301, 340, 385
artificial life, 45
ASCII art, 101–102

aspect ratio, 103
brightness, 103, 104, 105
command line options, 107
font, 103
generating, 102–104, 106–107
ramp, 103, 104
tiles, 103
writing to text file, 107
aspect ratio, 103, 158
`assert` method, 123
Audacity, 65, 74
audio signal, 287
autostereograms, 135
command line options, 145
creating tiled image, 144
depth map, 138
depth perception, 137, 138
example, 136, 146
linear spacing, 136, 137
random dots, 142
repeating pattern, 137, 143
wall-eyed viewing, 137
average color, 115
avoiding loops, 87

B

BLE, 313
advertisement, 313
central, 313
peripheral, 313
scan response, 314
blending, 155, 216, 233
Bluetooth Low Energy. *See* BLE
BlueZ, 314
`fromhex`, 329
`hciconfig`, 314
`hcidump`, 314
output, 328
`hcitool`, 314
parsing data, 327

starting the scan, 326
stopping the scan, 327
`board` module
creation, 332
I2C, 322
route, 332
serving static files, 332
Boids simulations, 79
adding a boid, 91–92
animation, 92
boundary conditions, 83
drawing, 84–86
initial conditions, 81
obstacle avoidance, 95
rules, 80, 86
scattering, 92
tiled boundary conditions, 83
`Bottle` module, 315
`route()`, 315
running, 315, 316
web framework, 315
BS170 MOSFET, 285
`bytearray.fromhex()`, 329
bytecode, xxiv

C

cellular automaton, 45
central processing unit (CPU), 153
centroid, 90
CircuitPython, 318
`collections` module, 66
color cube, 218
color representation, 219
common folder, 395
complex numbers, 298
computer-generated swarms, 79
computer simulation boundary conditions, 47
constant time, 66
Conway, John, 45
Conway’s Game of Life, 45

boundary conditions, 47
glider, 50
Gosper Glider Gun, 55
initial conditions, 50
patterns, 54
rules, 47, 48
toroidal boundary conditions, 47, 51
CoolTerm, 273
CPU (central processing unit), 153
CT scan, 215

D

DC motor, 281
debouncing, 274
decorator, 274, 316
deep learning, 356
depth encoding, 138
depth map, 138
depth perception, 137
`deque`
class, 66
container, 66
example, 67
Direct Memory Access (DMA), 272
discrete Fourier transform (DFT), 286
`dmesg` command, 364

E

email signature, 101
Embedded PostScript (EPS), 32
epoch, 375

F

face culling, 219
factorial, 4
far plane, 158
fast Fourier transform (FFT), 286
amplitude, 286
example, 286, 287
frequency, 287

sampling rate, 287
FBO (frame buffer object), 220
field of view, 158
file handles, 120
fractal, 15
fragment shader, 155
frame buffer objects (FBO), 220
frequency, 59
fundamental, 60, 62

G

Game of Life. *See* Conway's Game of Life
geometric primitive, 158
GitHub repository for book, 395
GLFW, 154, 220, 238
`glfwCreateWindow()`, 163
`glfwGetFramebufferSize()`, 200
`glfwGetTime()`, 164, 200
`glfwInit()`, 162
`glfwMakeContextCurrent()`, 163
`glfwPollEvents()`, 165
`glfwSetKeyCallback()`, 163
`glfwSetTime()`, 164, 200
`glfwSwapBuffers()`, 165
`glfwTerminate()`, 165, 200
`glfwWindowHint()`, 162
`glfwWindowShouldClose()`, 164, 200
installation, 396, 397
keyboard events, 163
`GL_LINE_LOOP`, 155–156
`GL_LINES`, 155–156
`GL_LINE_STRIP`, 155–156
`glob` module, 366
`GL_POINTS`, 155–156
GLSL (OpenGL Shading Language), 153, 155, 158, 159, 160, 168, 169, 170, 216, 223, 224, 231, 232
compiling, 165
computing position, 169
`discard()` method, 169, 170
example, 158

fragment shader, 159, 169, 189, 224, 232, 236
gl_FragCoord, 232
gl_Position, 159, 169, 188, 223, 231, 236
length, 232
mat4, 157, 159, 168, 169, 188, 223, 231
normalize(), 189, 204, 232
sampler2D, 168, 169, 232
sampler3D, 232, 237
texture, 169, 232, 237
uniform, 159, 168, 169, 188, 223, 231, 232, 235, 236
vec2, 168, 169, 173
vec3, 158, 168, 172, 188, 189, 223, 232
vec4, 157, 159, 169, 173, 188, 223, 224, 231, 232, 236, 237
vertex shader, 158, 159, 168, 188, 231, 235
glTexImage3D(), 223
GL_TRIANGLE_FAN, 155–156
GL_TRIANGLES, 155–156, 219
GL_TRIANGLE_STRIP, 155–156, 161, 181, 184
glue language, xxv
Google Colab, 365
download file, 376
graphics processing unit (GPU), 153, 216
grayscale images and values, 102
greatest common divisor (GCD), 23
guitar, 60

H

harmonics, 63
homogeneous coordinates, 156, 182
hot glue, 289
hypotrochoid, 22, 282

I

I2S protocol, 262
amplitude, 263
SCK signal, 262
SD signal, 262
waveform, 262
WS signal, 262
IFTTT (If This Then That), 312

sending alerts, 330, 342
setup, 319–320
illusions, 135
image-based rendering, 216
inference on Raspberry Pi, 376
`input()`, 300
Internet of Things (IoT), 311
`Io` module
`io.BytesIO`, 334
IoT garden, 311
`adafruit_ble`, 321
`adafruit_bmp280`, 321
`adafruit_sht31d`, 321
architecture, 312
BLE scanner, 325
complete code, 345
`Bottle` module
complete code, 349
creating routes, 332
main HTML page, 333
CircuitPython code, 321, 343
code files, 320
complete code, 351
CSS, 338
getting sensor data, 335
hardware, 318
IFTTT alert, 342
JavaScript, 336
`async`, 336
`await`, 336
`Date()`, 336
DatePicker, 343
`fetch_data()`, 336
`getElementById()`, 336
`onload`, 337
`setInterval()`, 338
Raspberry Pi setup, 318
requirements, 317
running, 341

sensor data, 333
SQLite setup, 339
web server complete code, 331

K

Karplus on Pico
assembled hardware, 265
block diagram, 261
code listing, 266
complete code, 275
electrical connections, 265
generating notes, 267
hardware hookup, 264
hardware requirements, 263
implementation, 260
main function, 270
pentatonic scale (minor), 267
playing notes, 269
ring buffer, 268
running the code, 272
serial output, 273
Thonny, 272–273
Karplus-Strong algorithm, 62
low-pass filter, 63
k-d tree, 114
definition, 117
example, 117
nearest-neighbor search, 118
Koch snowflake, 3
combining, 9
complete code listing, 16
computing, 5
fractal, 3, 8
GitHub code, 11
recursion, 3
base case, 5, 8
recursive step, 5, 8
recursive algorithm, 3
running the code, 14

L

`lambda`, 374
laser, 279
laser audio
  aligning mirrors, 289
  attaching mirrors, 290
  block diagram, 280
  complete code, 293, 305
  constructing laser display, 289
  display, 303
  hardware
    completed, 292
    connections, 291, 292
    control, 294
    requirements, 288
  laser and mirrors, 282
  powering motors, 291
  processing audio, 296
  Raspberry Pi setup, 289
  required software, 288
  running the code, 302–303
  setup, 293
  testing the motors, 300
  test run, 302
laser module, 284
laser pointer, 281
linear algebra, 6
linear search, 116
linear spacing, 136, 137
list comprehension, 34
loops, avoiding, 87
loss function, 374
luminance, 105

M

MAC address, 323
machine learning
  deep neural networks (DNN), 356
  gradient descent, 357
  hyperparameters, 357

  inference, 356
  supervised learning, 356
  training, 356
  unsupervised learning, 357
magnetic resonance imaging (MRI), 215
magnitude of vectors, 82
major scale, 65
mapping, 102
`math` module
  `sqrt`, 87, 197
`matplotlib`, 72
  library, 48
    `animation`, 48, 53
    `imshow`, 49
    `interpolation`, 49
  module
    `animation`, 85
    `mpl_connect`, 91
matrix multiplication, 157
MAX98357A, 261
medical data, 217
microcontroller, 259
MicroPython
  `array` module, 267
  `bytearray`, 269
  `close`, 269, 270
  `I2S` module, 263
    creating, 271
    internal buffer, 271
    write, 270
  list
    `append()`, 268
    `pop()`, 268
  `memoryview`, 269, 270, 276
  open, 269
  `os` module
    `listdir`, 268
  `Pin` module, 270
    button input, 271

toggle, 271
`random` module, 267
`readinto`, 270
setup, 266
slice, 270
timing, 274
write, 269
`min()` method, 142
Mini speech commands dataset, 359
minor pentatonic scale, 66
mirrors, 281
attaching, 289
lasers, 282
modelview matrix, 224
modulus `(%)` operator, 51
motor driver, 284
`multiprocessing` module, 305
`Process()`, 378
`Queue`, 378
`empty()`, 383
`get()`, 383
`put()`, 380
musical scale
major, 65
minor pentatonic, 66
tone, 65

N

N-body simulation, 80
near plane, 158
`neopixel`, 321
`show()`, 324
normalize vector, 183
nRF52840, 318
`numpy` module, 119, 162
`abs()`, 297, 381, 382
`argmax()`, 375, 381, 385
arrange, 64
array, 49, 50, 53, 85, 87, 92, 105, 120, 166, 222
`reshape()`, 81

average, 120
average(), 105
broadcasting, 82
cos(), 81
cross, 198
FFT. *See* fast Fourier transform
  rfft(), 297
  rfftfreq(), 305
frombuffer(), 297
image, 106
linalg
  norm, 198
math
  radians, 198
max(), 381
min(), 381
optimization, 87
ptp(), 380
random, 50, 72–73, 81
  choice, 73, 196
  shuffle, 367
sin(), 64, 81
sum(), 298
zeros(), 190, 379, 381

O

object-oriented language, xxiv
OpenGL
  3D graphics pipeline, 154
  3D transformations, 156–158
  clamp texture, 171
  color representation, 219
  context, 161
  displaying, 161
  face culling, 219
  geometric primitive, 158
  glActiveTexture(), 168, 226
  glBindBuffer(), 166, 195, 225, 234
  glBindFramebuffer(), 226
  glBindRenderbuffer(), 226

glBindTexture(), 168, 170, 222, 226
glBindVertexArray(), 166, 168, 195, 225, 228, 234
glBufferData(), 166, 225, 234
glBufferSubData(), 195
GL_CCW, 217
glCheckFramebufferStatus(), 227
glClear(), 164, 200
glClearColor(), 163
GL_CULL_FACE, 228
glDrawArrays(), 168
glDrawElements(), 228
GL_ELEMENT_ARRAY_BUFFER, 226
glEnable(), 163
glEnableVertexAttribArray(), 166, 234
glFramebufferRenderbuffer(), 227
glFramebufferTexture2D(), 226
glGenBuffers(), 166, 225, 234
glGenFramebuffers(), 226
glGenRenderbuffers(), 226
glGenTextures(), 170, 222, 226
glGenVertexArrays(), 166, 225, 234
glGetUniformLocation(), 166, 190
glMultiDrawArrays(), 191, 195
glPixelStorei(), 170, 222
glRenderbufferStorage(), 226
GLSL (OpenGL Shading Language), 153, 155, 158, 159, 160, 168, 169, 170, 223, 224, 231, 232
  compiling, 165
  computing position, 169
  discard() method, 169, 170
  example, 158
  fragment shader, 159, 169, 189, 224, 232, 236
  gl_FragCoord, 232
  gl_Position, 159, 169, 188, 223, 231, 236
  length, 232
  mat4, 157, 159, 168, 169, 188, 223, 231
  normalize(), 189, 204, 232
  sampler2D, 168, 169, 232
  sampler3D, 232, 237
  texture, 169, 232, 237

uniform, 159, 168, 169, 188, 223, 231, 232, 235, 236
vec2, 168, 169, 173
vec3, 158, 168, 172, 188, 189, 223, 232
vec4, 157, 159, 169, 173, 188, 223, 224, 231, 232, 236, 237
vertex shader, 158, 159, 168, 188, 231, 235
glTexImage2D(), 170, 226
glTexImage3D(), 223
glTexParameterf(), 170, 222
glTexParameteri(), 226
GL_TRIANGLES, 219
GL_TRIANGLE_STRIP, 155, 161, 184
glUniform1i(), 168
glUniformMatrix4fv(), 167, 195, 228
glUseProgram(), 165, 167, 195
glVertexAttribPointer(), 166, 234
glViewport(), 163, 200
linear filtering, 171
modelview, 157
projection, 157
rasterization, 155
texture unit, 168
triangle strip, 161
vertex array object (VAO), 160
vertex buffer object (VBO), 160
OpenSCAD, 289
optimizer, 374
orthographic projection, 158
os module
listdir(), 119, 221
path, 72, 119, 222
overtones, 60

P

parallel processing, 153
parametric equations, 20
for a Spirograph, 20–24
pattern, 281
pentatonic scale, 65
minor, 267
performance analysis, 88

perspective projection, 158
photomosaics, 113
  averaging color values, 115, 120
  command line options, 126
  creating image grid, 123–124
  distance, 122
  grid, 114, 120
  matching images, 116, 121–122
  measuring distance, 116
  reading input images, 119
  RGB values, 116
  splitting target image, 115, 120
Pico, 259–263
  pin diagram, 264
  setup with MicroPython, 266
PIL (Python Imaging Library), 25
`Pillow` module, 25, 104, 119, 141
procedural language, xxiv
projection matrix, 224
public/private key scheme, 402
pulse width modulation, 282
PuTTY, 402
`PWM` objects, 282, 284
`pyaudio` module
  `close()`, 299
  creating, 296, 378
  `get_device_count()`, 378
  `get_device_info_by_index()`, 378
  `open()`, 296, 379
  read, 379
  `stop_stream`, 299
  `terminate()`, 378
  `write()`, 297
`pygame` module, 66, 69
`PyOpenGL` module, 162, 187, 220
Python Imaging Library (PIL), 25
  `convert() method`, 105
  `Image`, 104, 105
    `convert()`, 104, 105, 144

copy(), 56, 144
crop(), 106, 107, 121
load(), 119, 120, 144, 145
new(), 123, 142, 143
open(), 32, 104, 105, 119, 120
paste(), 123, 124, 143, 147
pixel access, 144
size(), 105, 120, 123, 127, 143
thumbnail(), 127
ImageDraw, 142
luminance, 105
Python installation
Anaconda
macOS, 397
Windows, 396
PYTHONPATH, 396, 397
Python interpreter, xxv

R

random module
choice(), 73
hideturtle(), 31–32
randint(), 142
showturtle(), 31–32
range() method, 24
Raspberry Pi
BCM pin numbering, 291
hostname, 401
Imager, 399
installing Python modules, 404
local, 401
logging in with SSH, 402
Pico, 259–263
pin diagram, 264
setup with MicroPython, 266
ping, 402
pinout, 291
scp, 403
setup, 399
Visual Studio Code, 404

rasterization, 155
ray, 216
  generation, 217–218
ray casting algorithm, 216, 220, 229
reflection, 281
relative paths, 119
`requests` module
  `post()`, 330
resolution, 113
  sound, 64
Reynolds, Craig, 79
RGB values, 116
ring buffer, 67
root mean square (RMS), 304
RP2040 chip, 259
`RPi` module
  `GPIO`, 293
    output, 294, 295
    PWM, 294
    `setmode()`, 294
    `setup()`, 294
  `GPIO.PWM`
    `ChangeDutyCycle()`, 295
    `start()`, 295
    `stop()`, 295

S

sampling rate, 64
sampling theorem, 370
`scipy` module
  `spatial.distance` module, 88
  `spatial.pdist()` module, 88
  `spatial.squareform()`, 87
  `stft()`, 371, 382
`scipy.spatial` module, 80, 87
  `KDTree`
    creation, 125
    `query()`, 122
scripting language, xxiv
Secure Shell (SSH), 402

semitones, 65
shaders, 153, 158
  fragment, 159
  vertex, 158–159
shaft, 281
short-time Fourier transform (STFT), 360
Sierpiński, Wacław, 15
Sierpiński triangle, 15
sine wave, 59, 64
sound
  amplitude, 63
  frequency, 59
  fundamental frequency, 60
  overtones, 60
spectral plot, 74
spectrogram, 359
speech recognition, 355
  audio preparation, 382
  block diagram, 358
  cleaning data, 368
  code for training, 365
  complete code, 389
  deep neural network (DNN) architecture, 357
  downloading training data, 366
  exporting model, 376
  inference strategy, 362
  listing input devices, 377
  Mini speech commands dataset, 366
  preparing audio data, 380
  required hardware, 364
  required software, 364
  running, 386
  running inference, 383
  spectrogram example, 369
  training the model, 370
spiral, 36
Spirograph equations, 21
  periodicity, 23–24
`sqlite` module, 316

close(), 317
commit(), 317
connect(), 317
CREATE TABLE, 317
cursor(), 317
execute(), 317
fetchall(), 317
ID, 317
SELECT, 317
sqlite3, 316
TS, 317
VAL, 317
ssh-keygen, 402
STFT (short-time Fourier transform), 360
string module
decode(), 327
split(), 300, 327
startswith(), 327
system resources, 120

T

TB6612FNG, 283
connecting, 291
TensorFlow
allocate_tensors(), 386
argmax(), 371
astype(), 384
audio.decode_wav(), 371
concat(), 371
convert_to_tensor(), 371
expand_dims(), 371
keras.utils.get_file(), 366
py_function(), 371
shape(), 371
squeeze(), 371
Tensor, 371
tf.data.Dataset
batch(), 373
from_tensor_slices(), 372
map(), 373

  tf.io.read_file(), 371
  tf.keras.callbacks.EarlyStopping(), 375
  tf.keras.layers
    adapt(), 373
    Conv2D(), 373
    Dense(), 373
    Dropout(), 373
    Flatten(), 373
    MaxPooling2D(), 373
  tf.keras.layers.experimental.Normalization(), 373
  tf.keras.losses.SparseCategoricalCrossentropy(), 374
  tf.keras.Model
    fit(), 375
    save(), 376
    summary(), 373
  tf.keras.models.Sequential(), 373
  tf.keras.optimizers.Adam(), 374
  tflite_runtime
    get_input_details(), 383
    get_output_details(), 383
    get_tensor(), 384
    Interpreter, 386
    invoke(), 384
    tflite_runtime, 377
  tf.lite.TFLiteConverter, 376
  tf.string.split(), 371
  zeros(), 371
TensorFlow Lite, 359, 376
text-based graphics, 101
texture mapping, 160–161
texture unit, 168
timeit module, 89, 127
time module
  sleep(), 72, 301
  time(), 330
timing, 88
tkinter module, 10, 25
  canvas, 32
tones, 65

torus
  calculating vertices, 191
  camera, 185
    complete code, 211
  cell colors, 194
  coloring, 185
  complete code, 203
  computing normals, 183
  computing vertices, 180
  creating the camera, 197
  fragment shader, 189
  Game of Life, 196
    complete code, 209
    rendering, 202
  mapping simulation grid, 187
  rendering, 183
  `RenderWindow`
    complete code, 211
  rotating camera, 198
  running the code, 201
  Torus class, 190
  transformation matrix, 182
  vertex shader, 188
translation matrix, 156
`turtle` module, 24
  `down()`, 24
  drawing a circle, 24
  drawing a spirograph, 27
  graphics, 9
    `down()`, 10
    `mainloop()`, 10
    `setpos()`, 10
    `tkinter()`, 10
    `up()`, 10
  `hideturtle()`, 31–32
  hiding the cursor, 31–32
  `listen()`, 33
  `mainloop()`, 24
  `onkey()`, 33

ontimer(), 29
setheading(), 36
setpos(), 24
setting the cursor, 25–26
setup(), 33–34
showturtle(), 31–32
title(), 33
up(), 24
window_height(), 29
window_width(), 29

U

USB microphone, 364

V

vectors
magnitude of, 82
velocity, 82
vertex array object, 160
vertex buffer object, 160
vertex shader, 155
volume rendering, 215
2D slices, 220, 233, 234, 235, 236, 237
3D texture coordinate, 217
color cube, 218, 220, 225, 226, 227, 228
geometry definition, 224
maximum intensity projection, 240
ray casting, 216, 220, 221, 229, 230, 231, 232, 233
algorithm, 216, 220, 229
reading data, 221–223
scaling, 240
volumetric data, 215

W

wavefile.read(), 367, 386
wave module, 64
getframerate(), 296, 297
getnchannels(), 296
getsampwidth(), 296
open(), 296

`readframes()`, 297
WAV file format, 64–65
creating, 64
playing, 69
writing, 69

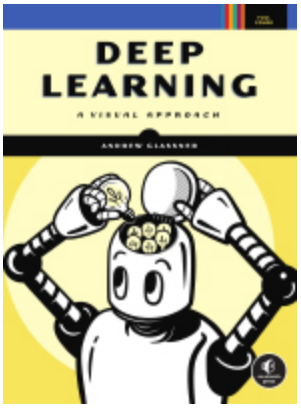
Z

`zip()` method, 82

RESOURCES

Visit *https://nostarch.com/python-playground-2nd-edition* for errata and more information.

More no-nonsense books from NO STARCH PRESS



DEEP LEARNING

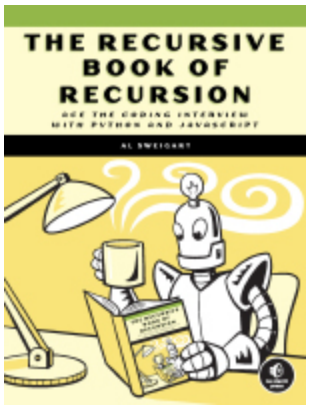
A Visual Approach

BY ANDREW GLASSNER

768 PP., $99.99

ISBN 978-1-7185-0072-3

Full color



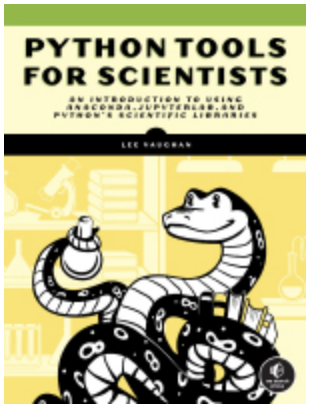
THE RECURSIVE BOOK OF RECURSION

Ace the Coding Interview with Python and JavaScript

BY AL SWEIGART

328 PP., $39.99

ISBN 978-1-7185-0202-4



PYTHON TOOLS FOR SCIENTISTS

An Introduction to Using Anaconda, JupyterLab, and Python’s Scientific Libraries

BY LEE VAUGHAN

744 PP., $49.99

ISBN 978-1-7185-0266-6



DEAD SIMPLE PYTHON

Idiomatic Python for the Impatient Programmer

BY JASON C. MCDONALD

752 PP., $59.99

ISBN 978-1-7185-0092-1



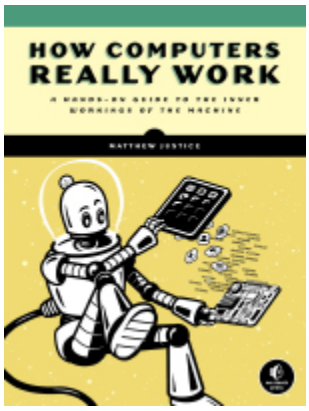
BEYOND THE BASIC STUFF WITH PYTHON

Best Practices for Writing Clean Code

BY AL SWEIGART

384 PP., $34.95

ISBN 978-1-59327-966-0



HOW COMPUTERS REALLY WORK

A Hands-On Guide to the Inner Workings of the Machine

BY MATTHEW JUSTICE

392 PP., $39.95

ISBN 978-1-7185-0066-2

PHONE:

800.420.7240 OR
415.863.9900

EMAIL:

SALES@NOSTARCH.COM

WEB:

WWW.NOSTARCH.COM