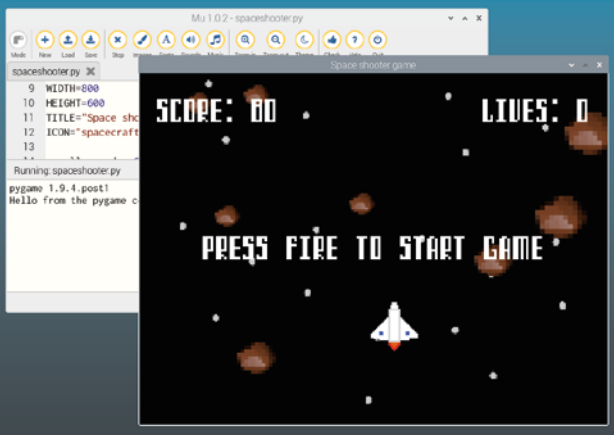
TECHNOLOGY IN ACTION™



Beginning Game Programming with Pygame Zero

Coding Interactive Games on Raspberry Pi Using Python

Stewart Watkiss

Apress®

Beginning Game Programming with Pygame Zero

Coding Interactive Games on Raspberry Pi Using Python

Stewart Watkiss

Apress®

Beginning Game Programming with Pygame Zero: Coding Interactive Games on Raspberry Pi Using Python

Stewart Watkiss
Redditch, UK

ISBN-13 (pbk): 978-1-4842-5649-7 ISBN-13 (electronic): 978-1-4842-5650-3
https://doi.org/10.1007/978-1-4842-5650-3

Copyright © 2020 by Stewart Watkiss

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Aaron Black
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-5649-7. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

For my children Oliver and Amelia.
You are the inspiration in my life.

Table of Contents

About the Author

Stewart Watkiss is a keen maker and programmer. He has a master's degree in electronic engineering from the University of Hull and a master's degree in computer science from Georgia Institute of Technology.

He has over 20 years of experience in the IT industry, working in computer networking, Linux system administration, technical support, and cyber security. While working toward Linux certification, he created the web site `www.penguintutor.com`. The web site originally provided information for those studying toward certification but has since added information on electronics, projects, and learning computer programming.

Stewart often gives talks and runs workshops at local Raspberry Pi events. He is also a STEM Ambassador and Code Club volunteer, helping to support teachers and children learning programming.

About the Technical Reviewer

Sai Yamanoor is an embedded systems engineer working for an industrial gases company in Buffalo, NY. His interests, deeply rooted in DIY and open source hardware, include developing gadgets that aid behavior modification. He has published two books with his brother, and in his spare time, he likes to contribute to build things that improve quality of life. You can find his project portfolio at http://saiyamanoor.com.

Acknowledgments

My family has been very supportive in my maker activities and while writing this book. Thank you to my wife Sarah for her support and to my children Oliver and Amelia who have been a source of inspiration and help while writing the book. Oliver has been particularly helpful in testing the games and giving me feedback, and my daughter's knowledge of music was a great help while writing about making sounds.

I'd also like to thank the team behind the Raspberry Pi including the Raspberry Pi Foundation and the community that has grown around it. I've also been inspired by the work of Nicholas Tollervey who created the Mu editor that is used throughout the book and Daniel Pope who created Pygame Zero, without which the book wouldn't have been possible.

I'm also grateful to all the support from the team at Apress, to Jessica Vakili for her support in putting the book together, and to Sai Yamanoor for the technical review. There are also many other people who helped to contribute through reviews and getting the book production ready.

Introduction

This book is designed for anyone wanting to learn programming through making fun games. It will also be useful for someone who has already learned the basics of programming and wants to learn how to add fun graphics and create their own games.

It is focused on making the games rather than teaching programming theory. In this book, you're more likely to see code on how gravity affects a missile's trajectory rather than the most efficient way to search through data. Even then the code is kept simple as games should be more about playability rather than complex physics.

The book starts with a simple text-based game to cover the basics of programming in Python. It then quickly moves on to creating simple graphical games in Pygame Zero. The book introduces object-oriented programming to make it easier to make more complex games. It also explains how you can create your own graphics and sounds.

Throughout the book, you will get to apply the new techniques in a variety of 2D games. As well as some new games, there are some variations on class games including a space shooter game and battleships.

The games are designed to run on the Raspberry Pi, although they can be used on other platforms that support Python 3 with Pygame Zero.

The games you make will be playable and hopefully fun to play. They are only the beginning. If all you ever do is copy the code from this book, then you are only going to learn so much, but by adapting and improving these games, they can become more enjoyable as well as helping you learn more than you

ever will from just typing out code that's written down for you. For each of the games, there is a list of suggestions for you to develop the games further.

All the code and resource files used in the book are available from the page to accompany the book at https://www.apress.com/gb/book/9781484256497.

CHAPTER 1

Creating Computer Games

Writing computer games is a great way to make programming enjoyable, but it does have its disadvantages. The main disadvantage is that to make a working game you need to write a lot of code which takes a lot of time. A full working game is usually too much for a beginner programming book. Fear not, as this book uses worked examples and takes advantage of the simplicity of Python and Pygame Zero to make it as painless as possible. In this book you will create a few different games to illustrate different programming techniques.

Creating a game is more than just writing code. This book covers some of the other aspects of creating a computer game as well as the programming.

First you need an idea. That idea then needs to be developed to come up with a set of rules and controls. It will likely need additional resources such as images and sounds. You will then need to write the code to make it happen. Next (and now comes the fun part) you need to test it to find out what works and how it can be improved. You then go back to the start to redefine the idea and repeat the programming cycle.

In this chapter you'll also find out about Python and Pygame Zero and some of the reasons that make it suitable for game programming.

© Stewart Watkiss 2020
S. Watkiss, *Beginning Game Programming with Pygame Zero*,
https://doi.org/10.1007/978-1-4842-5650-3_1

Inspiration Rather Than Imitation

The first step is about coming up with an idea. For this you may take inspiration from games you have played, which can be existing computer games, card games, board games, or playground games. Or you could come up with a completely new game, perhaps taking inspiration from activities in the real world. If you are looking to create a game based on something that has already been done before, then you do need to be careful about infringing on other people's intellectual property, including copyright, patents, and trademarks.

Like many laws, the rules protecting games and computer programs are complex and vary among different countries. It would not be possible to provide real guidance on the complex legal intricacies, but there are some general rules that you should follow.

Copyright can protect various aspects of work such as words, graphics, code, and music. Copyright does not however cover the idea of the game or how it's played. The work is automatically copyrighted when it is created and doesn't normally need a specific copyright notice or registration, although that can provide additional protection.

Patents are far more complex and can cover ideas and concepts. Patents are intended for inventions, and in the case of game programming, they can be granted for specific technical aspect of a game. For example, there are patents covering the way that directions are shown in a car racing game and how players are identified in a soccer game. It's incredibly difficult to know about what patents may relate to a game you are developing. If you are creating a commercial game, then you may want to look at getting professional advice on patents.

Trademarks are a way to protect names and logos, and in the case of computer games, they can include the appearance of the characters. This may prevent you from using a recognizable character if that character

is protected under a trademark. If you want to use any character that is protected under a trademark, then you will need to get a license granting you permission from the trademark owner.

Playing Games

The best way to learn about what makes a good game is to play them. Rather than just playing one game, play lots of different ones. Play good games and bad games and think about what makes the game good and bad.

Are you getting bored playing the game or does it have you hooked so you can't drag yourself away from the screen? Which games make you want to keep playing and why?

As mentioned previously you don't just need to take inspiration from computer games. Play some board games as well. Think about what works well and what doesn't. Think about the differences between playing a game using physical objects and when it is on a computer screen; there are likely to be both advantages and disadvantages to both.

Create the Resources

When looking at additional resources, you will likely be thinking about graphics and sound effects. There are other resources that you may need including introductory videos, tutorials, and background music.

For most games you are going to want to include graphics. The look and size of these graphics can determine the programming. For example, if you have a character that needs to move around the screen, then you will need to know how the character moves (whether its feet move) and the amount of space that is needed for that character to move around. It therefore makes sense to at least create an outline of any graphics prior to starting programming.

Sound effects can sometimes be left until later in the project, although they are often still an important part of creating an overall game. If leaving them to be added later, then it is still a good idea to think about when they will be used and what impact they will have when designing the game.

Development Cycle

The main buzzword relating to programming is agile. Agile programming is a way of developing software creating code in small increments implementing a feature at a time and then going back to add more code. The term agile programming is normally used to refer to a programming technique used for developing software across a team with regular reviews and team meetings (called scrums), but a similar technique can be used when programming on your own.

Some key points about developing code using an agile style methodology:

1. Gather requirements. Meet with end users or review your ideas with yourself as though you are the customer.
2. Plan the development. Split the work into small chunks that can be implemented a bit at a time.
3. Design the code to complete the current feature.
4. Write the code.
5. Test the code. As well as testing the standalone code, test how it interacts with other parts.
6. Assess whether the code is still in line with the requirements.
7. Return to 1. Consider the code that has been created. Is that compatible with what it is trying to achieve?

Keep repeating this cycle for each part of the code you develop. You then reach a release version once all the required parts have been implemented. Follow the same cycle when adding more features or improving the code.

Some things that are useful when using agile programming:

- Design interfaces between how the different parts of the code interact.
- Work in short code sprints with incremental releases.
- Perform regular short reviews of what has been completed during the last step and what you will be creating next. Reviews are normally performed daily in a work environment but differ if you are working in your spare time.
- Perform test-driven development by having specific tests that the code needs to pass. Automated tests are popular in agile programming, but you can also test manually.
- Refactor code regularly; review code for improvements for clarity/performance.
- Regularly check with the users (or yourself if it's a personal project) to see that the design is in line with the expectations.
- Use rubber duck debugging (see Chapter 11).

The games in this book are created based around agile programming. There will not be any of the code reviews specifically listed in the book, but you will see how the code is built up starting a feature at a time.

Making Programming Enjoyable

Whether you have a full-time job writing computer games, or it's something you do in your spare time, programming should be something you enjoy. I find a great deal of satisfaction from creating something that I would like to play myself.

While you can try and think of the concepts in advance, you may not know whether you enjoy the game until you get to play it. It's then when you get to tune the game to make sure it is the right difficulty or if there are features that you will want to add. This is discussed more in Chapter 4 when you will see some of the techniques used to improve on an initial game design.

Python and Pygame Zero

Python is a popular programming language used throughout education and in industry. It is available across a number of different computer operating systems including Apple Mac OS X, Microsoft Windows, and Linux. Some of the benefits of learning Python are it is easy to learn, uses less code (compared with some other languages), and can help teach good programming techniques.

Pygame is a library that can be used within Python to make graphical game programming easier. Pygame Zero is a library that uses Pygame but makes graphical game programming even easier than Pygame by reducing the amount of code needed. Using these, it is possible to create characters on the screen and move them around very easily.

This book uses version 3.7 of Python running on Linux which is the current version on the Raspberry Pi. The games should work across different computer systems and more recent versions of Python with Pygame Zero installed.

There are different styles for programming in Python. In this book the first few programs are written using primarily functional programming techniques, but then the later programs will be based around object-oriented programming. The functional programming style is generally considered easier to learn when starting programming, but once you start creating longer programs, then it is often easier to write and understand the code when written using object-oriented programming.

Compiled vs. Interpreted

Different computers and operating systems work in different ways. If you are creating a game designed for a phone or tablet (using a touch screen), then you may need to design the interface differently than if you are designing a game for a game console with a game controller. Also, different processors inside the computer and different ways that the operating system works mean that it can be difficult to write games that will work across multiple computers.

When writing computer code, you will normally use a programming language which uses a text-based language. Computers can't run that directly, and the code needs to be converted into the machine code that the computer can understand. When using a computer language such as C, the code must be converted to machine code before you can run the program. This is known as a compiled language and the program needs to be compiled into machine code that matches the computer architecture it will run on.

Python does this differently by converting the code to the machine language using an interpreter. This is done while the program is running. The benefit of this is that as long as there is an interpreter for the computer you want to run the code, you don't normally need to do anything extra to run it on that computer. The disadvantage is that interpreted languages can run slower because it needs to convert this code while it is running.

This performance won't be an issue with any of the games in this book, but you should be aware of it if programming a graphics-intensive game.

There is also a hybrid where the code is compiled to an intermediate form, but then still needs an interpreter (or something similar) for it to run on each particular computer architecture. This is how Java works using the Java Virtual Machine to convert from the Java Bytecode to machine language the computer can understand.

As Python is interpreted, it should be able to run on a variety of different computers without needing any changes. Unfortunately, it can sometimes be a little tricky to install the Python interpreter and the Pygame Zero libraries on some platforms. Fortunately, there is a simpler solution using the Mu editor which is the preferred editor for those starting with Pygame Zero programming.

Choosing a Programming Environment

In this book the games have been designed for a Raspberry Pi, which is a small, inexpensive computer designed specifically for those learning computing and computer programming. There are different variants of the Raspberry Pi including the tiny Raspberry Pi Zero and the fully featured Raspberry Pi 4. You can use any model of Raspberry Pi for the games in this book, although I would suggest using a Raspberry Pi 2 or better for performance reasons. If you are also using the Raspberry Pi for designing images for the games (as explained in Chapter 5), then a Raspberry Pi 4 may be advantageous, but it is not a requirement.

The Raspberry Pi is ideal for learning Python as most of the software you need is already pre-installed. The programs will still run on other computers and you are free to develop the code on another platform if you prefer, but there are a few extra steps involved on other systems.

Python programs are text files, and as such, you can create them in any text editor. If you've not programmed with Python before, then I suggest

you start with the Mu editor. The Mu editor is not the most powerful editor available, but its simplicity makes it ideal for getting started. It also handles most of the setup including Pygame Zero.

If using a Raspberry Pi, then latest versions of Raspbian include Mu, but if it's not already installed, then you can install Mu from a command shell. Start the command shell by clicking the black terminal icon at the top of the screen.

Then enter the following commands:

```
sudo apt update
sudo apt install mu-editor
```

You can then run Mu from the Raspbian menu system. From the start menu, select the programming menu, then click Mu, which should look like Figure 1-1.

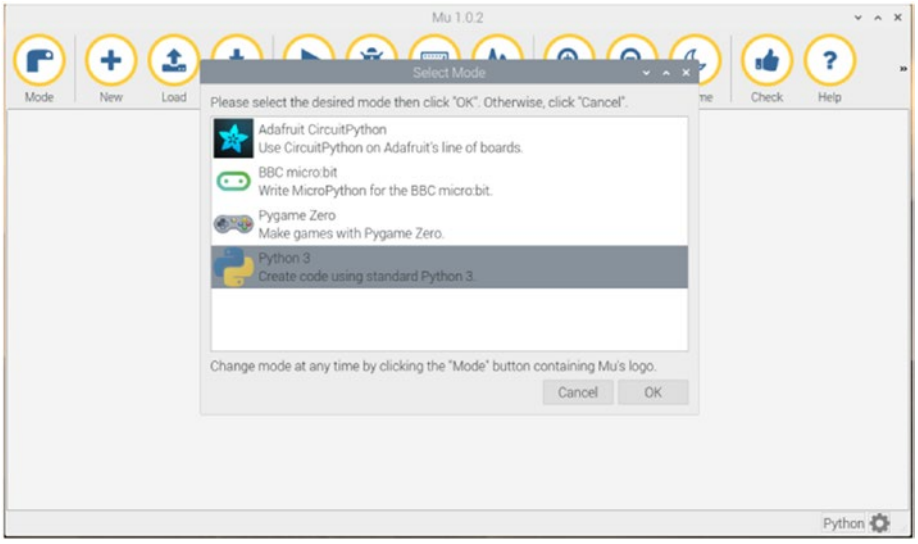


Figure 1-1. *A screenshot of the Mu editor*

If you would like to install Mu on other operating systems, then you can download the Mu editor from https://codewith.mu/. When installing under Windows, the recommendation on the Mu web site is to install for "this user only". That will make it easier to add any modules that may be required later.

The Mu editor has different modes which are useful for different programming environments. This book uses the Python 3 and Pygame Zero modes.

When you have more experience, you may want to change to a more powerful editor. If using a Raspberry Pi, then you have a number to choose from and you can run the programs directly from the command line. If you are using a different environment, then you may need to set up a native Python environment with Pygame Zero.

Summary

This chapter has looked at some of the things you should think of before you start programming. It has given suggestions on where you can get inspiration from and a warning about some of the pitfalls that you should avoid around other people's intellectual property.

It has explained what Python is and why Pygame Zero is a good choice for those starting out in game programming.

In the next chapter, you will get started with creating code and create a command-line game using Python.

CHAPTER 2

Getting Started with Python

To get started with programming Python, this chapter begins with some basic command-line programming. This will create a simple text-based game that can be played using the keyboard. This is only the beginning; from the next chapter onward, you will be able to create graphical games that are fun to play.

Using the Mu Editor

When you first start the editor, it will ask you which mode to start in. The modes that you will use for the projects in this book are Python 3 and Pygame Zero. If you have already run the editor before, then it will start in the mode last used, in which case you can change the mode using the mode button on the top left of the editor.

For this chapter you will create basic text-based program, so you should select Python 3. In future chapters, you should use Pygame Zero.

When you first start Mu, there should be an empty screen with a comment `# Write your code here :-)`.

The # at the beginning of the line means that this is a comment and would be ignored. Comments are really useful for programmers to explain how the program works, but Python just ignores them. You can

© Stewart Watkiss 2020
S. Watkiss, *Beginning Game Programming with Pygame Zero*,
https://doi.org/10.1007/978-1-4842-5650-3_2

delete that line for now, but when you write your own code, I suggest you add comments to explain how the code works as that can be useful in understanding the code in future.

To get started, you can create a basic program called "Hello World". It is one of the smallest programs that you can create. This is literally one line of code as shown here:

```
print ("Hello World")
```

Replace the comment in the Mu editor with this print statement. You will then need to save the program before running it; I'd suggest saving it in the default folder (/home/pi/mu_code) and calling it helloworld.py. If you try to run the code before saving, then you will be prompted to save it first.

After saving the file, click Run and you will see the program running in the bottom part of the screen. In this case it prints Hello World to the text-based screen area. This is shown in Figure 2-1.

Figure 2-1. *The Hello World program running in the Mu editor*

Once you have finished, click the Stop icon to stop the program from running.

This is the most common way of running a Python program from Mu. Another alternative is to run the program from a Raspbian Linux command shell. Save the current program using the Save button. You will see where the file is saved by looking at the status message at the bottom of the editor, in this case

```
/home/pi/mu_code/helloworld.py
```

To run this program from the command line, launch the terminal program from the Raspbian menu launcher. The terminal is a text-based interface used to communicate with the operating system including starting other programs. You can change to the folder that the program is stored in by using the cd command. The filename consists of the directory which consists of all the characters up to the last "/" character (note that the directory separator on Linux faces the opposite way to the folder separator used on the Windows operating system).

In this case the directory path is /home/pi/mu_code/ and the filename is helloworld.py. To change to the directory and run the program, enter the following commands:

```
cd /home/pi/mu_code/
python3 helloworld.py
```

Your program will now run and display the same "Hello World" text as you saw previously in the Mu output screen. This is shown in Figure 2-2.

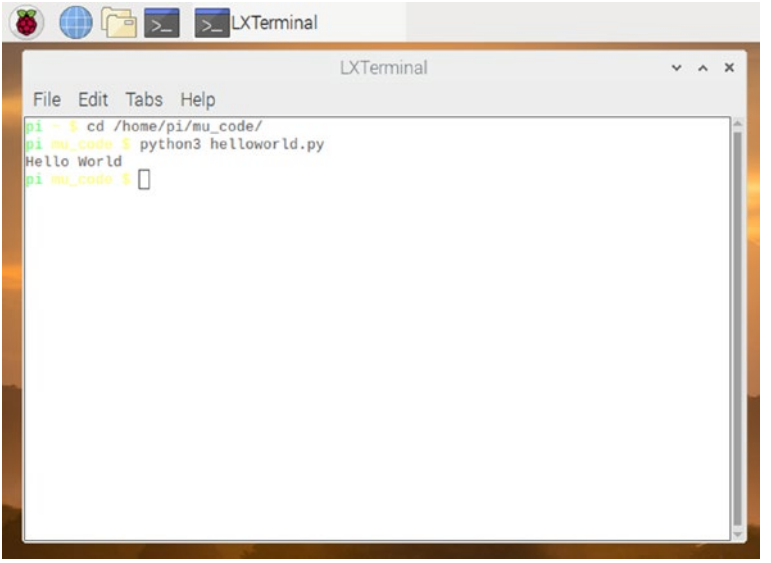


Figure 2-2. *Running the Hello World code from the command line*

Another way to run Python code is using the REPL. It stands for read-eval-print loop (but the name is not important). What the REPL does is it provides a way of running Python code in an interactive mode. This can be useful to test running small amounts of code prior to including it in your programs.

To run the same code in the REPL, click REPL in the Mu editor menu bar. You must be in the Python 3 mode to see that menu option. If the REPL icon is not shown, then use the mode icon on the Mu menu bar to change mode. After clicking the REPL icon, there will be an interactive shell at the bottom of the screen. Note that if your previous programming is still running, then it will show the program output and the REPL side by side, and if so, then click the Stop button which will give the REPL the full width of the editor.

You will see a prompt in the REPL screen which will normally show "IN [1]:". Enter the previous program code at the prompt

```
print ("Hello World")
```

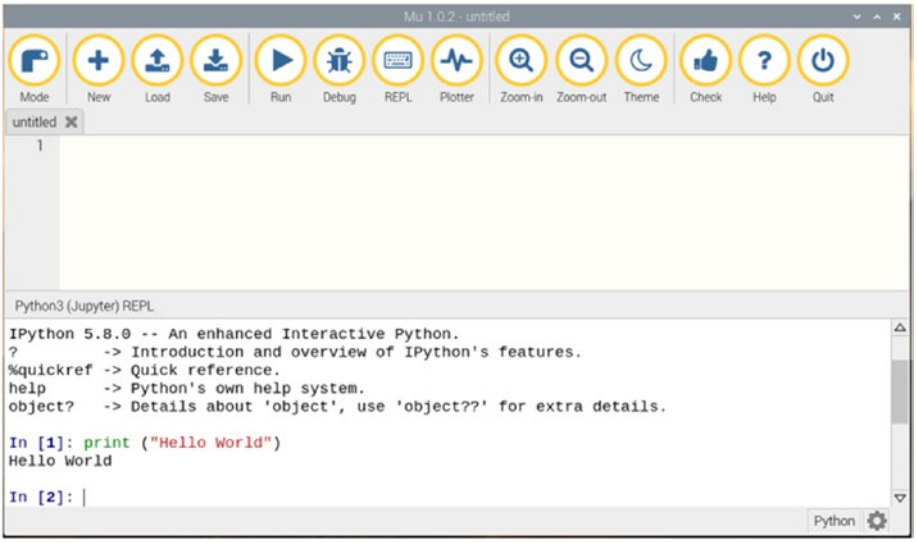


Figure 2-3. *The REPL in the Mu editor running the Hello World code*

Then press Enter to see the effect of running that instruction. This is shown in Figure 2-3.

You can also access the REPL by running `python3` from the command line. In that case the REPL prompt is shown by three greater than characters ">>>". If running from the command line, then you need to press Ctrl-D to exit.

Python Programming

When creating a Python program, you need to follow a certain structure so that the program can run correctly. This first game will cover some of the rules that need to be followed for a Python program to work.

The game is a simple joke quiz. The program will ask the player a question with a joke answer. If the player answers the question correctly, then they will be congratulated; otherwise, they will be given the punchline.

Click the New button in Mu to create a new file and enter the code in Listing 2-1.

Listing 2-1. Joke quiz program

```
1 print ("Welcome to the Python joke program")
2 player_guess = input ("Why couldn't the engineer fix the
  computer?\n")
3 if (player_guess == "too many bits"):
4     print ("Well done!")
5 else:
6    print ("too many bits")
```

The code must be entered exactly as shown, except for the numbers on the left which should not be typed in (and are shown by default in the margin of the Mu editor). The numbers are included to make it easier to explain the code or to help fix problems if the code doesn't work correctly. They should not be included as they don't form part of the code.

Python code is case sensitive, so print, Print, and PRiNT are completely different as far as Python is concerned. The spacing is also important. Lines 1, 2, 3, and 5 should start in the first character position on the left-hand side of the editor. Lines 4 and 6 should be indented by four spaces; the editor will help by auto-indenting after it sees a colon ":" character. Mu also inserts four spaces whenever you press the Tab button.

Save and then run the program. When you first click Save, you will need to give it a filename. Name the file joke.py or another appropriate name. You can then click the Play button to run the program.

The program will print "Welcome to the Python joke" followed by "Why couldn't the engineer fix the computer?". At this point the player needs to have a guess. Enter `I don't know`. The computer will then respond with "too many bits".

At this point I'd like to apologize for such an awful joke. I'm sure that you can do much better, so feel free to change the text between the quotes to your favorite joke.

If you run the program a second time, then you already know the answer, so you can type in the answer when prompted as shown in Figure 2-4.

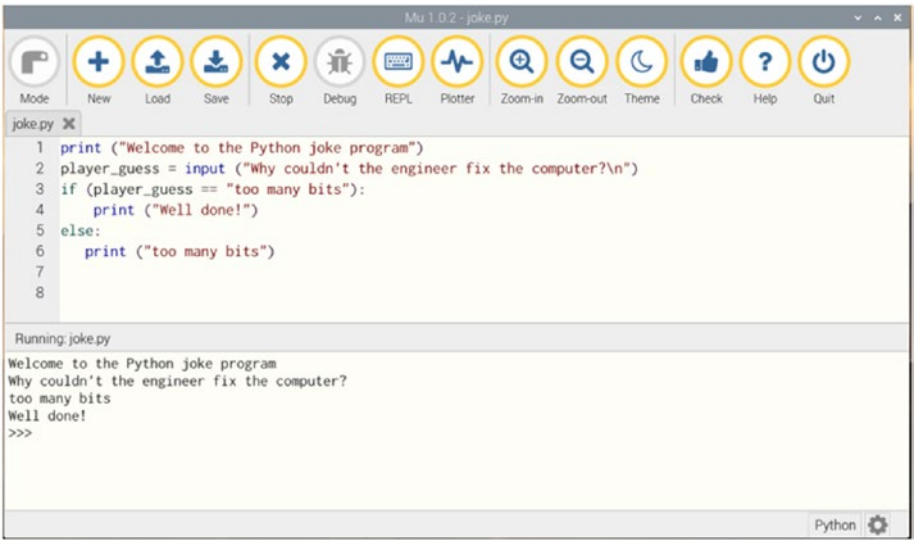


Figure 2-4. *The output of the joke.py game*

To explain how the code works, it is useful to look at it one line at a time.

The first line is `print ("Welcome to the Python joke program")`.

This code runs a function called print. A function is a block of code that performs a certain function. In this case the print function is included with Python and contains code that can print text to the screen. You can identify print as a function because of the brackets. Some functions need one or more values inside the brackets which are known as arguments, but not all functions use arguments. In the case of print, it takes a single argument which is a text string. The quotes around the text indicate that the text is to be used as a text string rather than a variable.

Note Functions and methods. You may see references to both functions and methods in this book. In Python a method is like a function but is contained within a class and operates on an object. Python uses both depending upon the context. These are object-oriented programming terms and are explained in Chapter 9.

Line 2 uses the `input` function, which displays a message to the user and then waits for the user to respond with an input. The argument is a text string, like that used in line 1. The function returns a string value containing what the player entered as their guess. The returned value is stored in a variable called `player_guess`.

The argument text string includes a special sequence `\n` at the end. This is an escape sequence that moves the cursor onto the next line. This is required, as unlike the print function, the output function does not add the new line automatically. Both variables and the escape sequences will be explained later in this chapter when looking at variables.

Line 3 compares the variable that is stored in `player_guess` and sees if it matches the text string `"too many bits"`. If it does match, then it runs the block of code that is indented, which in this case is line 4. Line 4 is the `print` function again which gives the player the message `"Well done!"`.

Line 5 is an `else` which is the opposite of the if on line 3. If the condition in line 3 is not met, then it runs the indented block of text after the else clause which is line 6. Line 6 uses the `print` function to `print "too many bits"`.

The `if` and `else` clauses are conditional statements, which are explained in more details later.

There are some more things that can be added to improve the program, but first it will be useful to understand how data can be stored inside a computer program.

Variables

A common way of storing information in a computer program is using variables. You can imagine a variable as a box that you can store something in, but rather than storing physical objects, the variables store information.

The following example will create a new variable called `my_variable` and store the value 7 inside the variable.

```
my_variable = 7
```

The variable name must start with a letter or an underscore character. The rest of the variable name can then include letters, numbers, and underscores. Variable names are always case sensitive, so a variable called `My_Variable` would be different to `my_variable`.

In some programming languages, you need to specify what you will be storing in the variable, such as whether it will be a number or a string. In Python this is dynamic so a variable can change type as required. It is important to know about the different variable types as it is often necessary to convert between the different variable types.

The main variable types used in Python are

Integers (int) which store whole numbers without any fractions.

Examples of integers are 3, 3948392, and –237 (they don't need to be positive numbers).

Floating-point numbers (float) which store numbers that include fractions or a decimal point. Examples of floating-point numbers include 2.99, 3.14159, –345.2, and 1.0.

Character (chr) which refers to a single character of text. In Python characters are stored in unicode, so as well as being able to store standard text like 'a' and digits such as '3', there are many different characters such as Greek letters or letters with an accent symbol. Note that the '3' character is not the same as the number 3; when stored as a character, it normally needs to be converted to a number before any arithmetic operations can be performed on it.

Strings (str) are used to hold text. They are stored as a collection of characters which are strung together. You can imagine this as a string of letter beads where each bead has a letter to make up a word (see Figure 2-5). A string can be any length from an empty string (zero characters) to an entire book (if you wanted to).

Booleans (bool) can represent either true of false. The fact that they can only hold those two values means that they are useful for making simple yes and no decisions.

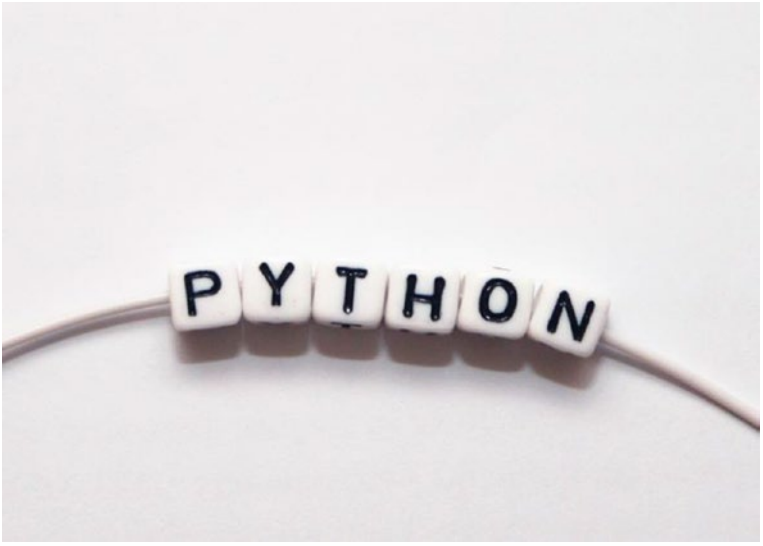


Figure 2-5. *A string of characters making up the word PYTHON*

The text in brackets for each of the variable types (such as int) is the name of the built-in function that is used to convert a different variable into that variable type. For instance, if you have an integer, but want it to be a string, then you can use the str function. You can then use the type function to see what type of variable is being stored.

To see this in action, you can enter some commands into the REPL. Within Mu, click the REPL button and then enter the text that follows the >>> characters shown here. The response is shown in bold.

```
>>> variable1 = 10
>>> print (variable1)
10
>>> type(variable1)
int
```

```
>>> variable2 = str(variable1)
>>> print(variable2)
10
>>> type(variable2)
str
```

As you can see, variable1 and variable2 appear to show the same value when printed using the print function, but they are stored as different variable types.

If you tried to join the two variables using the + operator, then you will get an error as shown here:

```
>>> variable3 = variable1 + variable2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

A common reason for needing to convert between variable types is if you wanted to print the value stored in an integer or float variable. This can be achieved by printing the output of the str function.

```
>>> print ("The value of variable1 is " + str(variable1))
The value of variable1 is 10
```

It's also important to understand when numbers are stored as strings and integers/floats when you need to perform arithmetic operations on them. You can see this yourself by testing some operations in the REPL.

```
>>> integer1 = 1
>>> integer2 = 2
>>> integer3 = integer1 + integer2
>>> print (integer3)
3
```

```
>>>
>>> string1 = "1"
>>> string2 = "2"
>>> string3 = string1 + string2
>>> print (string3)
12
```

As you can see, when adding the integer numbers together, you get the arithmetic sum which is in this case 1 + 2 = 3. If they are stored as strings, then the second string is appended to the first one giving the string "12".

That example shows why it is important to differentiate between numbers and numbers within a string, but what about floating-point numbers instead of integers? In fact, why do we even need integers as a floating-point number can hold any integer value, just with zero after the decimal point? There are two main reasons. The first is for efficiency; it takes less space to store integer numbers and they are much easier for the computer to manipulate. The other is about inaccuracies due to rounding up values. To store floating-point numbers, particularly those created as the result of a division, the computer may need to round the value. If you then create a different variable with the same amount but using a different technique, then it may be rounded off differently giving a value that is almost the same number, but not quite. As a result, it is not normally considered safe to check for a floating-point value being equal to another. If you want to check a floating-point value for a certain value, you should always compare to see if is within a certain range rather than assuming it is an exact value. If you only need whole numbers, then it is better to store them as integers.

It's quite common to want to increase or decrease numbers by a set amount. For example, when a player scores, then you may need to increment the score variable. You can achieve this with the following code:

```
score = score + 1
```

This will work, but there is a shortcut that allows you to increase an existing variable. In Python you can use += to increase and -= to decrease a variable. You can see this by testing this in the REPL.

```
>>> score = 0
>>> score += 1
>>> score
1
>>> score += 2
>>> score
3
>>> score -= 1
>>> score
2
```

The += and -= shortcut is used a lot by programmers, and you will rarely see the longer format in programs.

The += also works with strings, which will append the new string to the end of the first. This is demonstrated as follows:

```
>>> var1 = "string 1"
>>> var1 += " string 2"
>>> var1
'string 1 string 2'
```

Strings and Format

As mentioned previously strings are a group of characters. The strings don't need to be limited to normal text characters as they can also make use of special character sequences. You have already seen the escape sequence \n which inserts a newline character; there are also others such

as \' and \" which are used when you want to include a quotation mark within the string and \\ which is used when you want to include the \ character within the string.

To create a string is as simple as putting some text in quotes (either single or double quotes). "This is a string", 'This is also a string'. The only difference between using single and double quotation marks in Python is that if you want to use the same quotation mark within the string, then you will need to use the escape character first.

As shown previously you can also add strings together using the plus sign +. This combines the two strings into a new single string (referred to by some other programming languages as concatenation). If you want to include a non-string variable, then convert it into a string first as shown here:

```
>>> string1 = "Your score is "
>>> score = 10
>>> string2 = string1 + str(score) + "points"
>>> print (string2)
Your score is 10 points
```

There are some alternative techniques which can be used for formatting strings. The first is known as the printf-style formatting. The Python documents now discourage using printf-style formatting as it is easy to make a mistake when using it. It's useful to recognize this formatting if you come across it in someone else's code. If you come across a string with a % after the closing quotes, then they are using printf-style formatting:

```
>>> score = 20
>>> "Your score is %d points" % (score)
'Your score is 20 points'
```

An improved way of formatting string is to use str.format(). This uses braces {} to show where a variable should be inserted. To create the same as the preceding example, you would enter

```
>>> score = 30
>>> "Your score is {} points".format(score)
'Your score is 30 points'
```

An even better way is using the new f-strings. These include the name of the variable within the main part of the string rather than having to add it to the end.

```
>>> score = 40
>>> f"Your score is {score} points"
'Your score is 40 points'
```

Unfortunately, the f-strings are only available in recent versions of Python (version 3.6 or later) and will fail on older versions. Raspbian did not include a compatible version prior to the Buster image (2019). It is now possible to use f-strings on a Raspberry Pi, but the usage will restrict is to those running recent versions of Raspbian. It will also take some time for other computers to upgrade to the latest version of Python, so you may be better using str.format or concatenating the strings together using a + character. The code in this book uses mainly the concatenation or the str.format method depending upon which is most readable.

There are also lots of built-in string methods which help when manipulating text. For example, if you want to compare text ignoring the difference between upper- and lowercase letters, then you can convert the string to lowercase by using the str.lower method. This is included in the improved code for the joke program shown in Listing 2-2.

Listing 2-2. Updated joke quiz program

```
print ("Welcome to the Python joke program")
player_guess = input ("Why couldn't the engineer fix the
computer?\n")
if (player_guess.lower() == "too many bits"):
    print ("Well done!")
else:
   print ("too many bits")
```

The updated version of the joke quiz program will now accept the answer regardless of whether the player uses any capital letters or not.

Lists

The variables mentioned previously are great for storing a single piece of information, but are a bit limiting when you need to store more information. For that Python provides lists.

For example, if you want to have a number of different questions for a quiz game, then instead of creating different variables called question1, question2, and so on, you can create a single list called questions. The following two lists show five questions and answers for quiz. This will be used to create the first game.

```
answers = ["Tetris", "Picade", "Python", "Sega", "luigi"]

questions = [
    "What Russian tile matching game was popular in the 1980s?",
    "What is the name of the Raspberry Pi arcade machine from
    Pimoroni?",
    "What programming language has a logo featuring two snakes?",
    "Which company created Sonic The Hedgehog?",
    "What is the name of Mario's twin brother?"
    ]
```

I've put the answers first as they are shorter so easier to follow. The answers list contains five strings. The square brackets denote this as a list, and the individual entries are separated by commas.

If the entries are more than a few words long, then it's often easier to read the code by placing each entry on a separate line. As you can see from the questions list, this follows the same format as the answers with the square brackets and the separating commas, but each entry is placed on a new line and there are four space characters at the beginning of each line to indicate that this is part of the same block.

The individual entries can be accessed by using the name of the list followed by the index position in square brackets. As is common with most programming languages, the index starts at position 0. The following example shows how the first question and answer can be printed to the screen:

```
>>> answers = ["Tetris", "Picade", "Python", "Sega", "luigi"]
>>>
>>> questions = [
...     "What Russian tile matching game was popular in the
       1980s?",
...     "What is the name of the Raspberry Pi arcade machine
       from Pimoroni?",
...     "What programming language has a logo featuring two
       snakes?"
...     "Which company created Sonic The Hedgehog?",
...     "What is the name of Mario's twin brother?"
...     ]
>>> print (questions[0])
What Russian tile matching game was popular in the 1980s?
>>> print (answers[0])
Tetris
```

You can also update one of the questions by referring to it by its index. To correct the intentional mistake of not starting Luigi with a capital letter, we can update it as follows:

```
answers[4] = "Luigi"
```

To add a question to the list, use the append method.

```
>>> questions.append("What is the name of the giant barrel
throwing ape in Nintendo's classic game?")
>>> questions.append("Donkey Kong")
```

You can also create an empty list just by using []. To store the players guesses, you could use

```
>>> guesses = []
```

If you decide you want to delete an entry, then the del statement can be used to delete an entry as the specified index. For example, to remove the second question, use

```
>>> del questions[1]
```

This will move the rest of the entries in the list to fill the gap, so if you wanted to keep the two arrays in sequence, then you would need to do the same to the answers list.

There is much more that you can do with lists, including inserting entries at a specified position, removing entries based on their value, or even re-ordering the entire list. For more details see the Python documentation, a link is included in Appendix B.

Note Python also has a different data storage object type known as an array. It works in a similar way to lists, but first needs to be imported. Arrays do have some advantages such as if you need to perform mathematical operations over an entire array. Arrays are beyond the scope of this book. If you want to find out more, see the links in Appendix B.

Dictionaries

Lists can be a useful way to organize data when you want to access it based on its index position, but sometimes you want to associate the information with a word instead. In this case you can use a dictionary where each entry is associated to a key instead of a numerical position.

You can think of this just like a traditional dictionary book, where it is indexed by a word and then provides a description. The dictionary in Python can use any string for the index, which is known as the key. The description can be any kind of variable or object and is known as the value.

A dictionary is created in a similar way to a list but uses braces {} around the dictionary and uses key value pairs.

```
>>> dictionary1 = {'key1':'value1', 'key2':'value2'}
```

The individual entries are then referenced using the key instead of the numerical index that we used in a list

```
>>> print (dictionary1['key2'])
value2
```

An example would be if you have a game with a different welcome message depending upon a user selected language. You could use the user's language as the key.

```
>>> welcome_message = {'english':'Welcome',
    'french':'Bienvenue', 'german':'Herzlich willkommen'}
>>> language = 'french'
>>> print (welcome_message[language])
Bienvenue
```

Tuples

Another type of data structure commonly used in Python is the tuple. The best way to think of a tuple is as a list that cannot be changed once created (in programming "jargon" this is known as being immutable). These are commonly used in Python as return values, where more than one value needs to be returned, or to represent an object that has multiple values such as x,y coordinates.

To create a tuple, you just create a list of values surrounded by brackets. For example, the following could represent the position of a spaceship where x = 10 and y = 15.

```
position1 = (10,15)
```

You will see examples of where tuples are used in Chapter 3 when creating an actor on the screen.

Conditional Statements (if, elif, else)

Conditional statements provide a way to change the execution of code. They work by testing for a certain condition and only running parts of the code if that condition is met.

You have already seen an if statement in the earlier code in Listing 2-1. The section of the code dealing with the if statement is repeated here:

```
3 if (player_guess.lower() == "too many bits"):
4     print ("Well done!")
5 else:
6    print ("too many bits")
```

In this case the code on line 4 is only run "if" the condition is met. The code on line 6 is only run if the condition is not met as defined by the "else".

The if statement evaluates any tests or instructions up to the colon. This is known as the conditional expression. It determines whether the output of the conditional statement is true or false. If it is true, then it runs the block of text indented after the if. If it is not true, then it will skip that block of text.

The "else" clause and associated block of code is optional. When that is included, then that code will only run when the "if" condition is not met.

The indentation of the block of text is important. I recommend each block is moved in by four spaces for each indent. In Mu this is usually done automatically, and pressing the Tab button will automatically replace it with the correct number of spaces. In other editors pressing the Tab key may generate a tab character instead of four spaces; this will prevent the code from running.

When adding an if statement, the value you need to evaluate may not necessarily be a true or false answer, in which case you can use a comparison operator to change it to a true or false answer. Consider a game where you add different amounts of points as the player progresses through the game. A silver coin adds 1 point, a gold coin adds 5 points, and a bag of coins adds 10 points. If the player reaches 100 points, then they get a level up. This is easy to achieve within the add score code using

```
if (score == 100):
    level += 1
    print ("Level up to "+str(level))
```

There is however a problem with this code. If the player has reached 98 points and then collects a bag of coins which earns them 10 points, then their score will increase to 108 points. The comparison will never be true as the score will have increased too quickly, and they will not have met the condition where score was equal to 100.

Instead you need to check to see if the score is either equal to or greater than 100. The angle brackets "< >" can be used to check whether something is less than or greater than a value. So

```
if (score > 99):
```

will check for 100 or higher. Alternatively, you could combine that with an equals to compare to it being greater than or equal to. So

```
if (score >= 100):
```

will work if the score is equal to 100 or if the score is greater than 100.

A summary of the different comparisons is shown in Figure 2-6.

Operator Symbol	Operation	Comments
==	Equals	Tests to see if the values are identical.
!=	Not equals	Tests to see if the values are not not identical.
>	Greater than	Tests that the value on the left is greater than the value on the right.
<	Less than	Tests that the value on the left is less than the value on the right.
>=	Greater than or equal to	Tests that the value on the left is greater than or equal to the value on the right.
<=	Less than or equal to	Tests that the value on the left is less than or equal to the value on the right.

Figure 2-6. *Common comparison operators*

If you change the code using greater than in place of equality test, then you may also need to update related parts of the code. If greater than or equals was substituted in the earlier code, it would increase the level every time that the player scored a point after 100. So instead of just increasing the level, the code needs to check within certain bounds such as

```
if (score >= 100 and score < 200):
    level = 1
    print ("Level up to "+str(level))
```

This adds another test which is the logical and operator. Using the and statement, the condition is only met when both the left-hand and right-hand side are true.

This is used in the form of

```
if (condition1 and condition2):
```

Another logical operator is the or operator which will evaluate to true if either condition is true. This is summarized in Figure 2-7.

Logical Operator	Comments
and	True if both conditions are true.
or	True if any of the conditions are true.
not	Invert. If true then return false, if false then return true.
True	Always evaluate to true.
False	Always evaluate to false.

Figure 2-7. *Logical operators*

At first glance, including True and False may feel superfluous, but sometimes they can be useful. Typically, these can be used as a condition in a loop (a True operator in a loop creates a forever loop), or it can be useful to use either of these temporarily when debugging code.

There are other ways of evaluating a true or false condition. This may be through a function that returns a value or by entering a variable directly.

In these cases, if a value is equal to false or zero, then it is evaluated as false. For any other return value, the value evaluates as true. This may be that the return value is positive or negative, or a string is non-empty. This can cause a little bit of confusion when trying to understand how a value is going to be interpreted. If there is some ambiguity, then I recommend comparing it against a known value to make it clear.

Simple Quiz Game

After covering some of the basics, you should now be ready to create a simple quiz using the list of questions and answers created earlier.

Enter the code in Listing 2-3 into a new file. Ignore the line numbers which are included to make the code easier to explain.

Listing 2-3. Simple quiz game – quiz0.1.py

```
 1 # Simple quiz game
 2
 3 # Score starts at 0 - add one for each correct answer
 4 score = 0
 5
 6 # List of questions
 7 questions = [
 8     "What Russian tile matching game was popular in the 1980s? ",
 9     "What is the name of the Raspberry Pi arcade machine from
       Pimoroni? ",
10     "What programming language has a logo featuring two snakes? ",
11     "Which company created Sonic The Hedgehog? ",
12     "What is the name of Mario's twin brother? "
13     ]
14
15 # Answers - correspond to each question
```

```
answers = ["Tetris", "Picade", "Python", "Sega", "Luigi"]

print ("Welcome to the computer game quiz")

# Ask the first questions, store response in player_guess
player_guess = input (questions[0])
if (player_guess.lower() == answers[0].lower()):
    # If correct say so and add 1 point
    print ("Correct")
    score += 1
else:
    print ("Incorrect")

# Ask the second question
player_guess = input (questions[1])
if (player_guess.lower() == answers[1].lower()):
    # If correct say so and add 1 point
    print ("Correct")
    score += 1
else:
    print ("Incorrect")

# Ask the third questions
player_guess = input (questions[2])
if (player_guess.lower() == answers[2].lower()):
    # If correct say so and add 1 point
    print ("Correct")
    score += 1
else:
    print ("Incorrect")

print ("You scored {} points".format(score))
```

This program is included in the accompanying source code named quiz0.1.py.

The code starts with some comments prefixed with the # character.

Line 4 creates the score variable and sets its initial value to 0.

Lines 6 to 16 add the questions and answers, as explained previously.

After giving a welcome message to the player (18), lines 21 to 27 ask the first question and check if it is correct using an if statement. You will see on line 22 that both the player's answer and the correct answer are converted to lowercase (.lower function) so that it doesn't matter if the player inputs the answer using capital letters or not.

Lines 29 to 36 ask the second question and then lines 38 to 45 ask the third question.

Finally line 47 tells the player how well they did.

If you look at lines 21 to 27, 30 to 36, and 39 to 45, you will notice that some of the code is repeated between the blocks. Except for the question number, the block of text for the first question is the same as that for the second and third. This is quite a lot of wasted code for just three questions, but imagine if there were more questions. If you have to add eight additional lines of code (including a comment) for every new question, then that is going to add up to a lot of code. This is where loops come in useful.

Loops – While, For

After conditional statements, one of the most important things that code needs to do is to repeat actions. This is usually done in the form of a loop.

The quiz code in Listing 2-3 showed how repeating code can increase the amount of code that needs to be written. It also means that if you want to make a change to the code, then changes will need to be made across multiple lines which is a waste of time and increases the risk of mistakes.

Loops are even more important when it comes to code that needs to keep running. If you have an arcade machine, then it would not be much

good if the whole machine needed to be rebooted after each person has finished playing. For most computer games after the "game over", you expect to have the option to play again without needing to restart.

When creating loops in Python, there are essentially two different types of loop. The `while` loop is the easiest to construct and so will be covered first.

While Loop

The `while` loop can be shown through a demonstration

```
num_times = 0
while (num_times < 10):
    print ("This is line number "+str(num_times))
    num_times += 1
```

If you enter the code in the Mu REPL and hit Enter, then you should see the following:

```
This is line number 0
This is line number 1
This is line number 2
This is line number 3
This is line number 4
This is line number 5
This is line number 6
This is line number 7
This is line number 8
This is line number 9
```

This repeats the command ten times.

The main thing to consider is the while loop which will run "while" the variable num_times is less than 10. To run this as a loop, num_times variable must be updated during each loop.

In this case the variable is incremented once during each loop, but sometimes the variable may change differently. It may be that the loop needs to run while the player's score is less than a certain value or until a certain trigger is reached. There will be further examples of loops in the code used in later games.

For Loop

An alternative is the for loop. Typically, a `for` loop is often used to iterate over a list. This makes it useful when you want to give it a list and run some code for each of the items in the list.

Again, this is easiest demonstrated through an example

```
questions = [
     "What Russian tile matching game was popular in the 1980s?",
     "What is the name of the Raspberry Pi arcade machine from
     Pimoroni?",
     "What programming language has a logo featuring two snakes?"
     "Which company created Sonic The Hedgehog?",
     "What is the name of Mario's twin brother?"
     ]
for this_question in questions:
    print (this_question)
```

which will print out each of the questions in turn showing the following output:

```
What Russian tile matching game was popular in the 1980s?
What is the name of the Raspberry Pi arcade machine from
Pimoroni?
What programming language has a logo featuring two snakes?
Which company created Sonic The Hedgehog?
What is the name of Mario's twin brother?
```

Looking at the code in the for loop, what it is doing is iterating over the list questions and storing the current value in a temporary variable called `this_question`. It then prints the content of `this_question`.

Another example is where you want to run a loop a fixed number of times. This is an alternative to the while loop used previously:

```
for x in range(0,10):
    print ("This is line number "+str(x))
```

This time the for loop uses the range function which allows it to iterate over a range of numbers. Effectively it is like having a list of numbers from the first argument to the second argument (not including second argument). This will give a list going 0,1,2,3,4,5,6,7,8,9. There is a third parameter which can be used to change the size of the step between the numbers.

So `range(0,10,2)` will only show the even numbers between 0 and 9.

The format for the function is

`range(start, stop, step)`
`start` `(optional if only one parameter is used) - the first number included`
`stop` `(required) - the maximimum value is one less than the stop value`
`step` `(optional) - the difference between each value`

The values can be negative. If you wanted to count down, then the step could be –1 to count down one per iteration.

Some other programming languages have different `for` and `foreach` loops. The Python `for` loop is like the `foreach` loop in other programming languages, but with the `range` function, it can act like a `for` loop from other programming languages.

Forever Loop – while True

A special case with the while loop is that it can be run with the condition set as True. This means that the loop will run forever.

```
while True:
    print ("Program is still running")
```

I don't recommend you run the preceding code as it will just keep running forever. Actually, forever is perhaps an exaggeration (but is a term used in some other programming languages); the loop actually runs until you stop the program externally, the computer stops, or the end of the world, whichever comes first!

If you do run the program, then you can cancel using the Stop button in Mu or Ctrl-C if running Python from the command line. The Ctrl-C will send a signal telling Python to stop running and give a KeyboardInterrupt error message. It is quite common to include a forever loop in command-line programs, although less so in Pygame Zero where the forever loop is handled in the background.

You may also see other programs using `while 1`. As 1 evaluates as True, then that is the same.

Changing Loop Flow – break and continue

What happens if you want an "almost forever" loop? Perhaps you want the program to continue running forever, except if the player requests to quit. There are two statements that can be used to change the flow within a loop (which applies to the for loop as well) which are break and continue.

A break statement will cause the program to exit the loop at that point and then run the code outside of the loop. A continue statement causes the code to jump back to the start of the loop, re-evaluate the expression, and then run the loop again (if the condition is met) or exit the loop (if the condition is not met).

Functions

A function is a way of defining a block of code so that it can be used elsewhere within the program. These can be built in and included in libraries or you can create your own.

One of the most popular Python functions is the print function, which has already been used in many of the examples in this chapter. At its most basic use, the print function takes a single string which the function displays in the console.

```
print ("string")
```

Essentially what happens when you call a function is that the current program flow pauses. Any arguments provided are passed to the function, the code in the function runs, and then when the function is complete, the flow returns to the previous point in the code.

You can create your own functions as shown in Listing 2-4.

Listing 2-4. Example of a Python function

```
1 def ask_question (question, answer):
2    player_guess = input(question)
3    if (player_guess.lower() == answer.lower()):
4        print ("Correct")
5        return 1
6    else:
7        print ("Incorrect")
8        return 0
```

Again, note that the line numbers would not be in the code.

This code won't do anything if you try to run it but should instead be included as part of a bigger program. In that case the line numbers would not normally start from 0 as a function would not normally be the first entry in an executable the file.

This is the same code that was used earlier in Listing 2-3 (lines 20 to 27) but using a function called `ask_question` and passing the question and answer as an argument instead of accessing the list directly.

The first line uses the "def" statement which identifies this as a function. The next item on line 1 is `ask_question` which is the name of the function. The function name follows a similar convention as variables, for example, it cannot start with a number and the convention recommends using underscore characters instead of a space. The brackets are used to include any arguments that need to be passed to the function. In this case there are two arguments: question and answer. The final character on line 1 is the ":" character which denotes the start of the function and the content of the function needs to be indented below.

Arguments do not need to be used in functions, but the brackets are still required if there are no arguments. An important thing to know when passing arguments to functions is that the function makes a local copy of the data passed as an argument, so any changes made to those variables are lost when the function returns.

The body of the function is then the same as the previous code except for the return statements (which will be explained shortly), and that instead of using the entries from the list code in the function uses the values provided in the argument. This means that instead of having to duplicate code, different arguments can be passed to the function.

The return statements on lines 5 and 8 used to end the function and return to the main code. Return statements are not always necessary as if you reach the end of the function there is an implied return, but a return can be added if you would like the code to return before reaching the end of the function or if the function needs to pass a value back. A return statement is often followed by a value or variable to be returned, but if not (or if there is no return statement), then a special value is returned which is "None".

Variable Scope

Variables can be created in the main part of the code or inside a function; the scope defines where the variable can be updated which can be either local or global. If the variable is created inside a function, then it will be a local variable which is only available inside that function. This is also the case for arguments which are copied into a local function. This allows multiple variables with the same name, which is an important feature for code reuse. It also prevents accidentally changing a variable in another function.

Sometimes you will need to access variables that are created elsewhere. For example, if there is a variable that holds the score, then that may need to be updated by any functions that need to update that score. To achieve this, the global keyword should be used within the function so that it can access the variable as a global variable.

This is easiest to understand through an example. Listing 2-5 shows example code to demonstrate the use of local and global variables.

Listing 2-5. Code demonstrating variable scope

```
variable1 = 1
variable2 = 1

def local_function (variable1):
    variable1 += 1
    variable2 = 5
    print ("variable1 in local_function {}".format(variable1))
    print ("variable2 in local_function {}\n".format(variable2))

def global_function (argument1):
    global variable1, variable2
    variable1 = argument1 + 10
    variable2 = 15
    print ("variable1 in global_function {}".format(variable1))
    print ("variable2 in global_function {}\n".format(variable2))
```

```
print ("variable1 in top level-code {}".format(variable1))
print ("variable2 in top level-code {}\n".format(variable2))

local_function (variable1)

print ("variable1 in top level-code {}".format(variable1))
print ("variable2 in top level-code {}\n".format(variable2))

global_function (variable1)

print ("variable1 in top level-code {}".format(variable1))
print ("variable2 in top level-code {}".format(variable2))
```

When this is run, it will produce the output shown in Listing 2-6.

Listing 2-6. Output of code demonstrating variable scope

```
variable1 in top-level code 1
variable2 in top-level code 1

variable1 in local_function 2
variable2 in local_function 5

variable1 in top-level code 1
variable2 in top-level code 1

variable1 in global_function 11
variable2 in global_function 15

variable1 in top-level code 11
variable2 in top-level code 15
```

There are two variables which are created at the top level of the code (outside of any functions). There are two functions; the `local_function` demonstrates local variables and the `global_function` shows how the global variables can be altered instead. There is no significance in the naming other than to make it clear which is being referred to. Any function can have any combination of local or global variables.

The variables are both set to 1 and that is confirmed by the first print statements. The first variable is passed as an argument to `local_function` which is defined as a local variable only visible inside that function. That value is increased to 2 which is displayed inside the function, but after the function finishes the original variable is unchanged. Another variable called variable2 is created and set to 5. When used within the `local_function`, it shows the value of 5, but this variable only exists within the function, and outside of that function, the value of `variable2` remains as 1.

In `global_function`, `variable1` is passed as an argument but is stored as a local variable named `argument1`. Both `variable1` and `variable2` are set as global through the global statement, and when they are updated inside that function, it also updates the value in the global top-level variable.

There is one more thing. If a variable is created at the top level and then read within a function without using the global statement, then the value of the top-level variable will be read. That same variable name cannot then be used as the name of a local variable.

Global variables are something that should, where possible, be avoided. The reason for this is that having multiple places update variables can result in code that is difficult to understand and debug. This is sometimes referred to as a "bad smell" if you see too many global variables in code. When using Pygame Zero (starting from the next chapter), you will see that there are quite a few global variables used. This is a nature of Pygame Zero in that the code runs within different functions which are part of Pygame Zero, and it is very difficult to pass variables into those functions without using global variables. Fortunately, object-oriented programming makes this easier, but that won't be discussed until Chapter 9. For the next few chapters, please accept that there will be a number of global variables but that the situation will change later in the book.

Refactoring the Code

Now that you've learned the theory of some additional programming techniques, you can put this into action with a new improved quiz. The player of the game won't notice any difference in this version, but I like to think of it as being "better code".

This is known as refactoring the code. Refactoring is where changes are made to the structure of the code that does not normally add any additional functionality but makes the code cleaner and easier to understand. It can also be used to make it easier to add new features.

The new code is shown in Listing 2-7, which I've called quiz0.2.py.

Listing 2-7. Refactored version of simple quiz game – quiz0.2.py

```
# Simple quiz game
def ask_question (question, answer):
    player_guess = input(question)
    if (player_guess.lower() == answer.lower()):
        print ("Correct")
        return 1
    else:
        print ("Incorrect")
        return 0

# List of questions
questions = [
    "What Russian tile matching game was popular in the 1980s? ",
    "What is the name of the Raspberry Pi arcade machine from
    Pimoroni? ",
    "What programming language has a logo featuring two snakes? ",
    "Which company created Sonic The Hedgehog? ",
    "What is the name of Mario's twin brother? "
    ]
```

```
# Answers - correspond to each question
answers = ["Tetris", "Picade", "Python", "Sega", "Luigi"]

while True:
    print ("Welcome to the computer game quiz")

    # Score starts at 0 - add one for each correct answer
    score = 0

    for i in range (0,len(questions)):
        if (ask_question (questions[i], answers[i]) == True):
            score += 1

    print ("You scored {} points\n".format(score))
```

This improved quiz starts with a function called `ask_question`. This function asks the player the question, checks for a correct response, and returns a 1 or 0 depending upon whether the provided answer is correct or not. The function is at the beginning of the code as it needs to be defined before it is called.

The questions and answers are then stored as lists. The order of the lists is such so that the question and answer have the same index. The rest of the code is wrapped in a while True loop, so that after the questions have been answered, the quiz goes back to the start.

Further Improvements

As with all the games in this book, this is a working game, but with scope for improvement. Some ideas for improvement are adding more questions (or changing them for a topic you are interested in), choosing questions to appear at random, and changing the output to give a different phrase depending upon the number of questions answered correctly.

Have a go at adding these and see if you can make the game more entertaining. You will need to use the random module to select the questions at random; you can find details on the Internet or you could return here after the next chapter where it is explained. In the supplied source code, I have included an example incorporating all of these as quiz0.3.py, although I suggest you have a go at making your own changes before you look at the code.

Summary

This chapter has been a very brief run through of the Python programming language. There is not enough space in this book to explain Python in detail. From now on the book will assume some familiarity of the Python programming language. If you need more information about getting started in Python, then I suggest the book *Beginning Python* by M. L. Hetland, published by Apress.

I have also included links to the official Python documentation in Appendix B.

The next chapter will move on to creating graphical games using Pygame Zero.

CHAPTER 3

Pygame Zero

So far, the programs have been text-based, but if we just stuck with text-based games, they would not have the same appeal as graphical ones. The rest of this book is about graphical games, which will be created using Pygame Zero.

To understand what Pygame Zero is, you first need to understand what pygame is. Pygame is a programming library for Python. It is designed for creating multimedia applications (such as games) easier. It also works across multiple platforms making it easier to share the games you make with other computers.

While pygame made it much easier to create games in Python, it still needs a reasonable amount of standard code, known as boilerplate code, before you can get started.

Pygame Zero is a more recent programming library which uses pygame but removes the need for much of the boilerplate code making it even easier to create games. Designed for use in education to help teach programming, it is a great way to get started creating computer games.

Pygame Zero Development

At the time of writing, Pygame Zero is an active project with improvements on a regular basis. Fortunately, most changes have maintained backward compatibility, but some new features will not work on all installations of Pygame Zero. If you are confident that the games you write will only be run

© Stewart Watkiss 2020
S. Watkiss, *Beginning Game Programming with Pygame Zero*,
https://doi.org/10.1007/978-1-4842-5650-3_3

on current or later versions, then you can include these into your game design, but if it is more important that your game will run on a wider range of computers, then you may want to restrict yourself to features compatible with older versions of the Pygame Zero libraries.

One example is that you can now include file path and filename extensions on resources (such as image files), but that does not work on older versions of Pygame Zero (prior to summer 2018). The code in this book has been tested with a recent version of Pygame Zero, but where I am aware of an issue with backward compatibility, I have tried to write the code so as to work with older versions as well.

Compass Game

The first graphical game will be known as Compass Game. The compass game is inspired by a game that is played by the Cub Scouts who I volunteer with; this in turn is a variation of a game known as Captain's Coming. The game is used to help teach the four cardinal directions (points of a compass). In the real (non-computer) game, a label is placed on each of the walls of the scout hall. The Cubs are given a direction, and they must run to the appropriate wall. Additional instructions can also be given, such as Captain's Coming, where the Cubs must stand still and make a salute.

In this chapter you will create a computer version of this game where the player is given an instruction that they must follow. The player must move their character in the direction stated. The game is shown in Figure 3-1.

Figure 3-1. *Screenshot of Compass Game*

This will provide an opportunity to learn about Pygame Zero and how to make a character appear to walk around the screen. This will be created using an agile methodology, adding a feature at a time to create the game.

Required Files

There are several image files that are needed for this project. These need to be in an image directory directly below your source code for the game.

You will need the files from the source code in the directory chapter3 and then the sub-directory images. If you are using the Mu editor, then they should be copied to the directory `/home/pi/mu_code/images`.

If you look in the chapter3 directory in the supplied source code, you will see a number of Python files prefixed with compassgame. The game is going to go through multiple iterations, and these files are used in the different stages in the development of the game. If you are following the instructions on your computer, then you create the game using just a

single file called compassgame.py which will then evolve throughout this chapter. The files provided in the source code can be used if you want to jump straight to the code for each stage rather than typing it yourself.

Running Mu in Pygame Zero Mode

The game should be created as a new file in Mu. You will need to change the mode to Pygame Zero. This is achieved by clicking the Mode icon on the top left of the editor. This is shown in Figure 3-2.

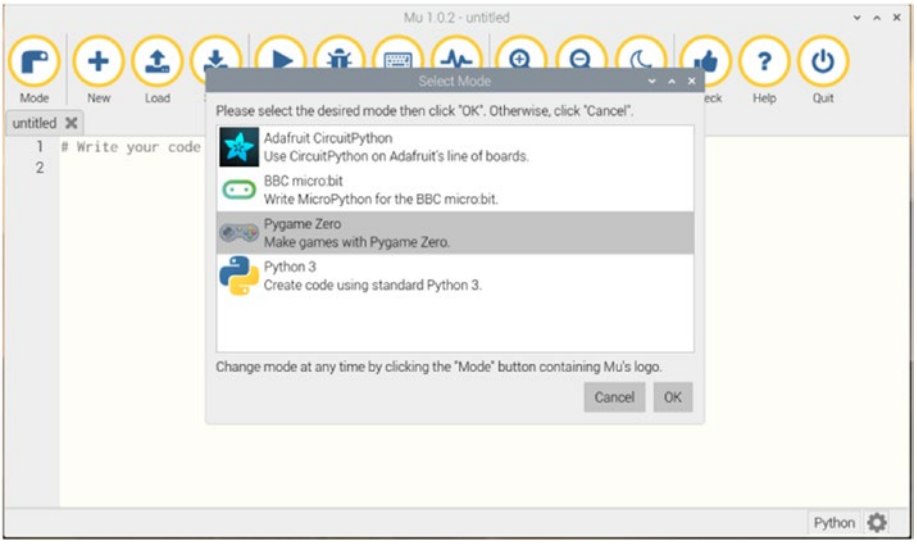


Figure 3-2. *Changing to Pygame Zero mode in the Mu editor*

Start by adding the following two lines to the file and then save it as compassgame.py.

```
WIDTH = 800
HEIGHT = 600
```

Then click the Play button in Mu and you should see a black screen, 800 pixels wide and 600 pixels high. This first example acts as a good demonstration of why Pygame Zero is so easy to use. Just defining the dimensions of the screen is enough to create a game window. In fact, this could have just been launched using an empty file, as those are the default values. This is less than the equivalent code using Pygame and it's easier to understand.

You can close the program by clicking the x in the top right or by pressing Stop from the Mu menu bar.

If you are not using Mu, then the file can be created in any other editor, but should be run from the command line using the following command:

```
pgzrun compassgame.py
```

This code is available in the source code in the file compassgame-layout1.py.

Tip Remember, if the menu has a run menu item instead of play, then you need to switch to Pygame Zero mode. Click the Mode button at the top left to select your mode.

Adding a Background Image

Now that you know how to create a basic Pygame Zero application, it's time to add something a bit more interesting. You can start by replacing the plain black background with something a bit more interesting.

Replace the current code with the code in Listing 3-1.

Listing 3-1. Simple Pygame Zero program with image background

```
WIDTH = 800
HEIGHT = 600

BACKGROUND_IMG = "compassgame_background_01"

def draw():
    screen.blit(BACKGROUND_IMG, (0,0))
```

The code is available in the supplied source code as compassgame-layout2.py.

Click the Play button and you should now see the same screen as before, but it will now have a green background image. If it doesn't work, make sure you have copied the images into the correct directory. There should be a file compassgame_background_01.png in the mu_code/images directory.

The code works by creating a variable BACKGROUND_IMG which has the name of the file to display. The image is entered as the filename without any path information or the .png suffix. On recent versions of Pygame Zero, you can use the full filename if you prefer, but to maintain compatibility with older versions of Pygame Zero, the files must be in the image folder and not include the suffix. This is the same for any image files used as Actors and backgrounds.

The line `def draw():` is defining the draw function. This is a Python function that Pygame Zero calls approximately 60 times per second. It should be used to tell Pygame Zero what should be displayed on the screen.

The function calls `screen.blit` which displays a bitmap image at the appropriate position (in this case 0,0 starting at the top left of the screen).

Note The reason for using such long filenames is because by default Mu puts all the code into the same directory. If you create multiple Python programs, they all share the same image directory. Naming them like this makes it obvious which files are for which program.

If you are using a different editor or have organized your game into a dedicated directory, then you may want to remove the compassgame_ prefix from the start of the filenames.

The image filenames also include a number which will allow us to change the look of the background or person.

Adding an Actor

In computer graphics, characters and other objects are known as sprites. In the case of Pygame Zero, it uses a more "friendly" name calling sprites Actors. I will often refer to these as sprites as that is the correct computing term, but remember when defining these in Pygame Zero to add them as an Actor object.

A sprite is an image used in a computer game that is often created from a bitmap image. These often take the forms of characters (people, animals, aliens, etc.), but they could also be used for objects that the players need to interact with such as obstacles, balls, or bullets fired from a weapon.

In this case you can start with a single sprite representing the player character. Later you can add more sprites to act as obstacles to add a challenge.

You will need several images for the sprites for the player character, so that you can show it facing different directions and to make it appear to move. The minimum needed would be an image with the character facing each of the following directions: front, right, left, and rear. To make

the movement a little more realistic, additional images can be used with the legs moving between the images. In Chapter 5, you will get to see how to design your own sprite characters, but for now you can use the sprites included in the source code. This is easiest by copying the files from the image directory in the source code in the image sub-directory within mu_ code. The sprite for this game is a person, but it could be replaced with an animal or with a different character completely such as a car.

To create a sprite, use the Actor object, with an image file.

```
player = Actor('imagefile')
```

The same rules apply about the image as previously mentioned for background images. If you want maximum compatibility, use the name of the image located in the image directory and without an extension. If using a recent version of Pygame Zero, you can include the extension and a path to the file location.

To position the sprite in a specific position of the screen, you can add the location as a tuple afterward. The following will create a sprite in the center of the screen:

```
player = Actor('compassgame_person_down_1', (WIDTH/2,HEIGHT/2))
```

The coordinate system starts in the top left-hand corner of the screen. The x-coordinate increases to the right and the y-coordinate increases downward. This is different to how graphs and maps work. The image in Figure 3-3 shows the game screen with some key coordinates marked.

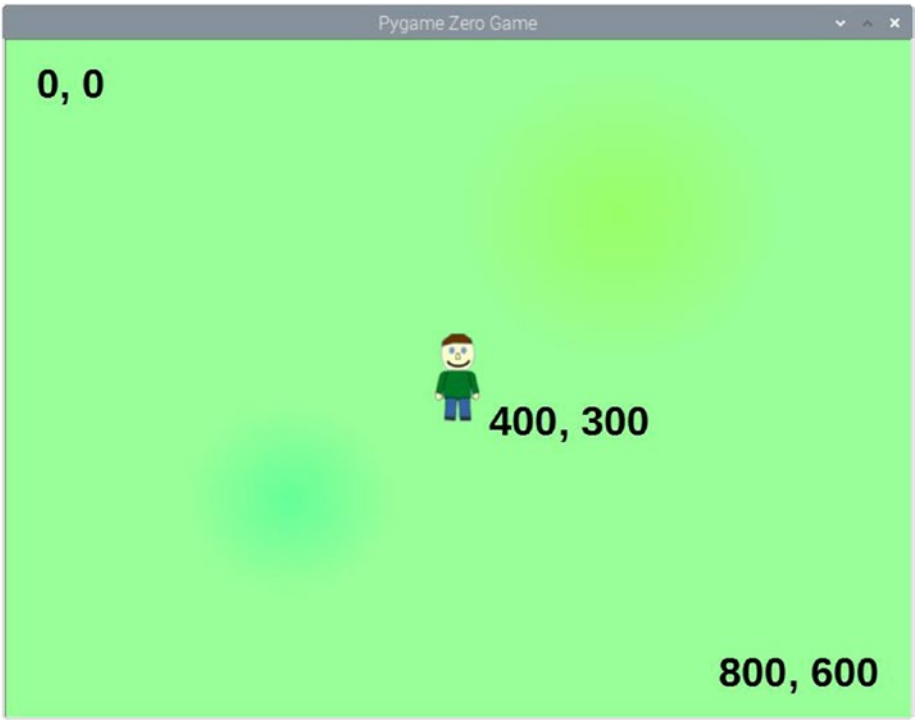


Figure 3-3. *The Pygame Zero screen coordinates*

As well as creating the actor, you need to include code to draw it onto the screen. This is achieved by putting the following entry inside the draw function:

```
player.draw()
```

The code to demonstrate this is shown in Listing 3-2, which is included in the source code as compassgame-player.py.

Listing 3-2. Simple Pygame Zero program with player actor

```
WIDTH = 800
HEIGHT = 600

BACKGROUND_IMG = "compassgame_background_01"

#Player character
player = Actor('compassgame_person_down_1', (WIDTH/2,HEIGHT/2))

def draw():
    screen.blit(BACKGROUND_IMG, (0,0))
    player.draw()
```

Moving the Sprite Around the Screen

Now that you have created a sprite (Actor), you can read the keys from the keyboard and make the player move in the direction of the key press.

To make it easier to test whether a key is pressed, Pygame Zero provides an attribute for each key. To test if the up arrow key is pressed, you should check the value of "keyboard.up". If the value is true, then the up key is pressed, if it is false, then it is not pressed.

You wouldn't use this method for getting text input from a player, because it doesn't tell you the order that the keys being pressed. It is however useful for game programming where there is just a small number of keys that can be pressed and where multiple keys can be pressed at the same time (such as up and right to move diagonally).

When you know which direction to move the player, then you can just change the x and y attributes to move the character a certain number of pixels in that direction.

The code to move the character is shown in Listing 3-3. Replace the current code with this updated code.

Listing 3-3. Code to allow the character to move around the screen

```
WIDTH = 800
HEIGHT = 600

BACKGROUND_IMG = "compassgame_background_01"

#Player character
player = Actor('compassgame_person_down_1', (WIDTH/2,HEIGHT/2))
# Direction that player is facing
direction = 'down'

def draw():
    screen.blit(BACKGROUND_IMG, (0,0))
    player.draw()

def update():
    # Need to be able to update global variable direction
      global direction

    # Check for direction keys pressed
    # Can have multiple pressed in which case we move in all
      the directions
    # The last one in the order below is set as the direction
      to determine the
    # image to use
    new_direction = ''
    if (keyboard.up):
        new_direction = 'up'
        move_actor(new_direction)
    if (keyboard.down):
        new_direction = 'down'
        move_actor(new_direction)
```

```
    if (keyboard.left) :
        new_direction = 'left'
        move_actor(new_direction)
    if (keyboard.right) :
        new_direction = 'right'
        move_actor(new_direction)
    # If new direction is not "" then we have a move button
      pressed
    # so set appropriate image
    if (new_direction != '') :
        # Set image based on new_direction
        player.image = "compassgame_person_"+new_direction+"_1"
        direction = new_direction

def move_actor(direction, distance = 5):
    if (direction == 'up'):
        player.y -= distance
    if (direction == 'right'):
        player.x += distance
    if (direction == 'down'):
        player.y += distance
    if (direction == 'left'):
        player.x -= distance

    # Check not moved past the edge of the screen
    if (player.y <= 30):
        player.y = 30
    if (player.x <= 12):
        player.x = 12
    if (player.y >= HEIGHT - 30):
        player.y = HEIGHT - 30
    if (player.x >= WIDTH - 12):
        player.x = WIDTH - 12
```

This is included in the source code named compassgame-movement1.py.

You should be able to follow most of the code by now, but there are a few new things which may need explaining.

The new_direction variable is a local variable inside the update function. It is used to hold the direction of the last key that it detected was pressed (so if you pressed up and right, it would hold right). This is used so that the character doesn't change between up and right when both keys are pressed, but also will be useful later when making the character's legs move. As new_direction is stored as a string, it can be included in the player image using the following line:

```
player.image = "compassgame_person_"+new_direction+"_1"
```

If the player is facing right, this will show the image `compassgame_person_right_1.png`.

A new function has been added called move_actor. As its name suggests, this moves the position of the actor on the screen. The first argument is the direction to move. The second argument for the function is defined as "`distance = 5`". This means that if a value is provided to the function, then that value will be stored in the distance variable, but if nothing is passed in the argument, then the distance variable will be set to 5. This can be useful when you want to include a default value for an argument.

When moving the actor, the code checks the position to make sure that the character does not go beyond the edge of the screen. It uses a y offset value of 30 pixels and an x offset value of 12 pixels so that the whole of the actor remains on the screen.

Making the Movements More Realistic

If you run the code in Listing 3-3, you will see the character move around and face the direction they are walking, but it does not look particularly realistic. As the legs are not moving, it appears as though the player is

sliding rather than walking. To make the movement look a little more realistic, we can change the image to show the person's legs moving.

The technique used is similar to the way that cartoons are made, where each frame is individually drawn with a slight movement. The frames are then shown one after another to make a moving image. In a typical cartoon, they may create around 20 images for each second of movement. In theory it would be possible to have the image change on every run of the update function, which is around 60 times per second; however, to keep this simple, the code will only update on every 5th time that the update function is called. This will give a frame rate of 12 frames per second. To achieve this requires four images for each direction that the player is moving in. Figure 3-4 shows the four images used for the right direction.

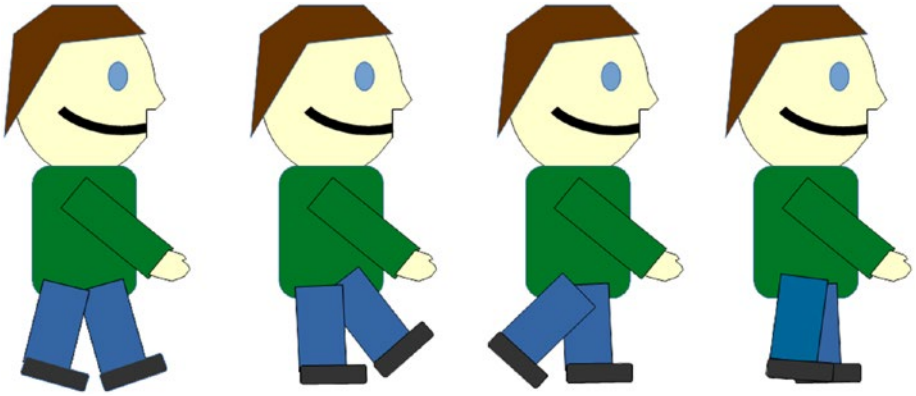


Figure 3-4. *The four sprite images for walking to the right*

In this example only the legs are moving, but you could have the arms moving as well to make it a little more realistic.

Using 4 images per direction needs 16 unique images. If you wanted to increase the frame rate, then you can increase the number of images. If you wanted to move the character twice as often, you would half the delay between each image and double the number of images to 32.

Previously the image was changed by updating the actor attribute. To allow for different images to be displayed, this can be changed to a function call to a new function called set_actor_image which will determine the correct image based on the direction of travel and the appropriate image in the sequence.

To update your previous code to show the character working, perform the following steps.

Add a new global variable near the top of the code called player_step_count. It can be placed after the definition of the direction variable.

```
player_step_count = 1
```

Replace the line

```
player.image = "compassgame_person_"+new_direction+"_1"
```

with

```
set_actor_image (new_direction)
```

Then add the following code to the bottom of the file:

```
# Show image matching new_direction and current step count
def set_actor_image (new_direction):
    global player, player_step_count

    player_step_count += 1
    if player_step_count >= 4:
        player_step_count = 1

    player.image = "compassgame_person_"+new_direction+"_"+
    str(player_step_count)
```

The updated code is included in the source code as compassgame-movement2.py. If you run the code now, then you will see the legs move, but it will be far too fast. It still needs the code to slow the movement down by only replacing the image on every 5th frame.

This is achieved by allowing the player_step_count to count up until five times the number of images and then dividing the image number by 5. The code will then discard any remainder and then add 1 (to start the image numbering from 1 instead of 0).

```
This is best illustrated by working through some examples.

With player_step_count set to 0
Divide player_step_count (0) by the delay (5) giving 0.0
Discard anything after the decimal place which gives 0
Add 1 to get image number 1

With player_step_count set to 1
Divide player_step_count (1) by the delay (5) giving 0.2
Discard anything after the decimal place which gives 0
Add 1 to get image number 1

With player_step_count set to 5
Divide player_step_count (5) by the delay (5) giving 1.0
Discard anything after the decimal place which gives 1
Add 1 to get image number 2

With player_step_count set to 19
Divide player_step_count (19) by the delay (5) giving 3.8
Discard anything after the decimal place which gives 3
Add 1 to get image number 4

With player_step_count set to 20, the maximum value has been
exceeded so set back to 0 and recalculate the value.
```

Most of this uses basic operations, but to discard the value after the decimal point, you will need the function floor() which is included in the math module. The floor function is defined as returning the largest integer value less than or equal to x.

The math module includes several mathematical functions, which can be useful when creating games. More details are available from https://docs.python.org/3.5/library/math.html.

To import the math module, add the following line to the top of the code:

```
import math
```

Then update the set_actor_image function (which was added to the bottom of the code) to match the following:

```
# Show image matching new_direction and current step count
def set_actor_image (new_direction):
    global player, player_step_count

    step_delay = 5
    player_step_count += 1

    if player_step_count >= 4 * step_delay:
        player_step_count = 1

    player_step_position = math.floor(player_step_count / step_
    delay) +1
    player.image = "compassgame_person_"+new_direction+"_"+
    str(player_step_position)
```

The updated file is included as compassgame-movement3.py in the source code.

If you run the updated code, you should see the legs move at a more realistic speed.

Keeping Game State

An important concept in programming is to be able to keep track of the state that the program is in. This is where the program needs to keep track of what has happened in the past and which influences how it will then handle future events.

If you think in terms of a board game, then the initial state may be when you have got the game out of the box and are placing the appropriate counters into each position. Once the game is set up, then there may be another state to determine who will be playing first (perhaps based on rolls of the dice).

Then when the game starts, the status will change between each person in turn for them to roll the device, move to the next position, and carry out any actions required. Finally, there will be some winning state when a player reaches the goal.

In a computer game, this is something that needs to be tracked using one or more variables. The game code can then handle key presses differently if it was displaying a menu screen rather than if the game was already in progress. The variable could be anything from a single number that has a specific meeting to a complete class with multiple properties.

For this game the code will need to track two things. One is the status of the game, so that it doesn't keep moving the character around the screen when the game ends, and the other is which direction the player needs to move in. These could be combined into a single variable, but to make it easier to follow this example uses two separate variables, one called game_state and the other target_direction.

The first variable is called game_state and tracks the different stages in the game. These are an empty string '' when the game has not yet started, the string 'playing' when the game is in progress, and a string 'end' when the game has finished.

In summary:

- '' – Game not started
- 'playing' – Game in process
- 'end' – Game over

For the target direction, the variable can be the different cardinal directions (four primary directions on a compass). These are 'north', 'east', 'south', and 'west' as shown in Figure 3-5.

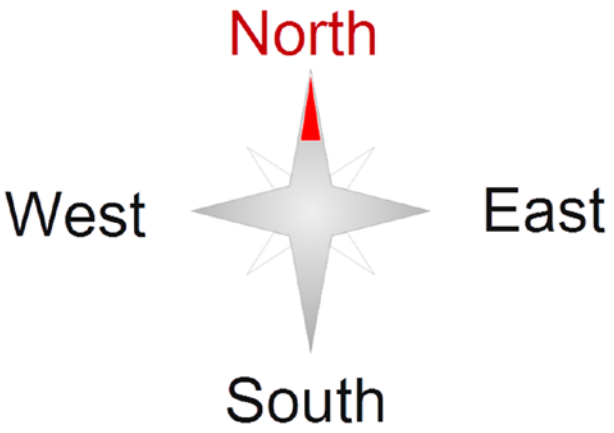


Figure 3-5. *The four points on a compass*

The code will be updated to generate a random direction. So, the random module needs to be imported by adding the following entry to the top of the file:

```
import random
```

Add the variables by adding the following lines near the top of the file (such as just after the BACKGROUND_IMG entry):

```
game_state = "
target_direction = "
```

Near the top of the update function, replace the `global direction` line with the following:

```
    global direction, game_state, target_direction

    # If state is not running then we give option to start or
      quit
    if (game_state == " or game_state == 'end'):
        # Display instructions (in draw() rather than here)
        # If space key then start game
```

```
        if (keyboard.space):
            game_state = "playing"
            target_direction = get_new_direction()
        # If escape then quit the game
        if (keyboard.escape):
            quit()
        return
```

At the bottom of the file, add the following function:

```
def get_new_direction():
    move_choices = ['north', 'east', 'south', 'west']
    return random.choice(move_choices)
```

This code will handle the state for starting the game.

If the game is not in progress, then it waits for the player to press the Start key, which in this case is the space key. If that is pressed, then it sets the status to playing and assigns a new target_direction.

The get_new_direction function has a list of the different directions and uses the random choice to choose one of the directions at random.

This is available in the source code as compassgame_state1.py.

You can now run the game again. Remember you will now need to press the space bar before you the player can be moved around.

The next thing to add is a way of telling the player which way to go. This can be done by using screen.draw.text() which will display text on the screen. Replace the current draw function with the following code:

```
def draw():
    screen.blit(BACKGROUND_IMG, (0,0))
    # If game not running then give instruction
    if (game_state == "):
        # Display message on screen
```

```
        screen.draw.text("Press space bar to start",
        center=(WIDTH/2,HEIGHT/2), fontsize=60, shadow=(1,1),
        color=(255,255,255), scolor="#202020")
    elif (game_state == 'end'):
        screen.draw.text("Game Over\nPress space bar to start
        again", center=(WIDTH/2,HEIGHT/2), fontsize=60,
        shadow=(1,1), color=(255,255,255), scolor="#202020")
    else:
        screen.draw.text(target_direction, center=(WIDTH/2,50),
        fontsize=60, shadow=(1,1), color=(255,255,255),
        scolor="#202020")
        player.draw()
```

The new draw function shows three different blocks of text depending upon the game state. The first block is when `game_state = ''`, in which case it instructs the player to press the space bar to start the game. The second is controlled by an elif (else if) which checks for the end of the game, and the third block is when the game is in progress. There is no need to check for the playing game state because if it's not the previous two states, then it must be in the state playing.

The `player.draw` is only called when the game is playing, because otherwise the text overlaps over the player. It is not yet possible to reach the end of the game. That will be something that will be implemented later.

The interesting thing about this code is the part that displays the text. Here are details of the first entry, but the others all work in a similar way:

```
screen.draw.text("Press space bar to start",
center=(WIDTH/2,HEIGHT/2), fontsize=60, shadow=(1,1),
color=(255,255,255), scolor="#202020")
```

The text method takes a string to show and a position; the rest of the arguments are optional. The position can be entered by just putting a tuple as the second argument, such as (10,10). In this case it looks better with the text in the center of the screen, so the tuple is passed to the center argument. It uses half the WIDTH and HEIGHT values to determine the position.

The other optional arguments used here are

- fontsize – Used to set the size of the font; the default value is 24.
- shadow – Adds a shadow to the text; the values are the x and y offsets for the shadow position.
- color – The color of the text.
- scolor – The color of the shadow.

As you can see, the code uses different ways to enter the color. You can use a few different color formats such as (r,g,b) where (255,255,255) is white or html color strings where "#202020" is a light gray color. See Chapter 6 for more details about how colors can be created.

Note The random numbers created by the random module are pseudo-random. Computers struggle with creating true random numbers, so instead they have a way of generating numbers that appear to be random to the end user. Depending upon the operating system and hardware, it may include less deterministic sources such as time and movement of the mouse to make it less predictable. This is usually sufficient for games, but if using it for cryptographic purposes, you may want to look at alternative random sources.

Detecting Collisions

If you've followed the code so far (or run compassgame-movement3.py), then you should now have a character that can work around the screen. The next stage is to detect when the player moves to the correct side of the screen. In this case it's enough for the character to be near that side rather than at the extreme edge as that is a bit more natural than having to actually stop at the side. One of the ways to achieve is to create code to look at the position of the character and check to see if it reaches a certain threshold. While that's a valid way of doing this, it is a bit inflexible, if you change the size of the character (perhaps to something that has a different height to width ratio) as you may need to update the code to handle that. Instead there is a nice feature available in Pygame Zero that allows it to check for a collision.

Unfortunately, the Pygame Zero documentation doesn't provide much information on detecting collisions. Pygame Zero uses the standard Pygame methods which are well documented in the pygame documentation (see the links in Appendix B).

The collision detection is often used to detect if two sprites (Actors) collide. To understand this, you need to be aware that all sprites in Pygame have a Rect property. That is something that is automatically created when you create an Actor through Pygame Zero. The Rect is a virtual rectangle that you cannot see. It is the minimum size rectangle that will fully include the size of the image. This is shown in Figure 3-6 where a representation of the bounding rectangle has been added around the actor.

Figure 3-6. *Actor with bounding rectangle*

The collideRect method can be used to determine if one rectangle overlaps the other. For example, consider a car game where two cars are moving around the game area and you want to know if they crash (collide) into each other. If you have actors called car1 and car2, you can detect to see if they have hit using

```
if car1.collideRect(car2):
    print ("Car 1 and Car 2 have collided")
```

Back to the game, we are currently working on "compass game"; for this particular detection, we don't actually need to create an actor to collide with, we just need to know when they are near to an edge of the screen. Instead we can create a simple Rect object with the appropriate dimensions. Then if the actor collides with one of those Rects, we know they are in that area. The image in Figure 3-7 shows the layout of the game with the rectangles shown on the playing grid. In this image the rectangles have been exaggerated to make them easier to see.

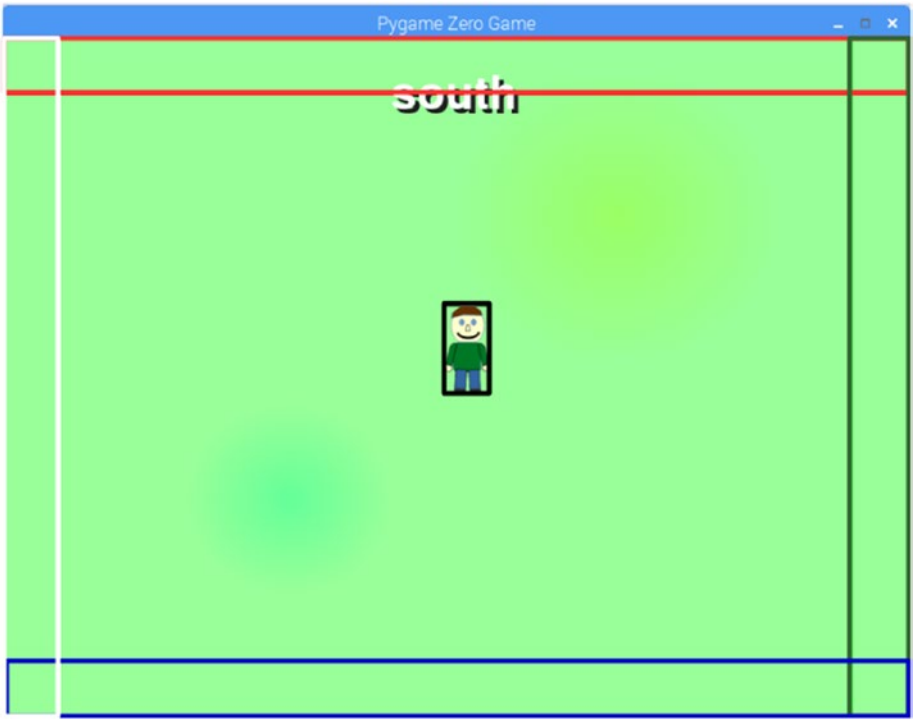


Figure 3-7. *Collision rectangles to detect the player approaching the edge of the screen*

You will see that the rectangles overlap. That's not a problem with this game as we only check to see if the player has reached one of these, but it is something you may need to be aware of when creating other games.

The following code is used to create the rectangles, which can be added before the draw function:

```
#Rectangles for compass points for collision detection to
ensure player is in correct position
box_size = 50
north_box = Rect((0, 0), (WIDTH, box_size))
east_box = Rect((WIDTH-box_size, 0), (WIDTH, HEIGHT))
south_box = Rect((0, HEIGHT-box_size), (WIDTH, HEIGHT))
west_box = Rect((0, 0), (box_size, HEIGHT))
```

The preceding rectangles are invisible, which is what we want. It is a good idea to temporarily display the rectangles as it can help show if any of the rectangles are in the wrong place. To do so you can add the following to the draw function (after screen.blit). This also includes a box around the player and uses a different color for each of the rectangles.

```
screen.draw.rect(north_box, (255,0,0))
screen.draw.rect(east_box, (0,255,0))
screen.draw.rect(south_box, (0,0,255))
screen.draw.rect(west_box, (255,255,255))
screen.draw.rect(Rect(player.topleft, player.size), (0,0,0))
```

The source code is included as compassgame-collide1.py. It is a good idea to have a little play with that version so you can see the rectangle around the actor move as you move around the screen. Whenever the box around the player overlaps with one of the other rectangles, that can be detected as a collision.

To detect the collisions, you can add the following code to the bottom of the update function:

```
if (player.colliderect(north_box)):
    print ("Collided with North")
if (player.colliderect(south_box)):
    print ("Collided with South")
if (player.colliderect(east_box)):
    print ("Collided with East")
if (player.colliderect(west_box)):
    print ("Collided with West")
```

This is included as compassgame-collide2.py in the source code. Now if you run the program and if you watch in the console in Mu (or in the terminal if you launched it from there), you will see several print messages whenever the player enters one of the rectangles.

This is good for testing, but you should now delete the blocks of code with the colliderect and draw.rect statements before the next stage.

Using rectangles is a convenient way of performing collision detection and works well enough for this game. If using other games, you may need to consider how the sprites interact particularly if they have a lot of "white space" around them. If you have a shape that doesn't fill the rectangle, then it can be frustrating for players if a player does not actually touch the other object, but that the rectangles overlap. It is instead possible to test on a specific point using the collidepoint method or to implement more accurate collision detection in your own code.

Change in Direction

Now you can add the code to handle the situation when the player reaches their target. Once they reach the required area, the player needs to be told where they need to go next. The player should then move to the new location before being told the next target and so on.

After deleting the code that printed out the collision notification, add the following in its place at the bottom of the update function:

```
    if (reach_target(target_direction)):
        target_direction = get_new_direction()
```

Also add the following after the update function:

```
def reach_target(target_direction):
    if (target_direction == 'north'):
        if (player.colliderect(north_box)):
            return True
        else:
            return False
```

```
    elif (target_direction == 'south'):
        if (player.colliderect(south_box)):
            return True
        else:
            return False
    elif (target_direction == 'east'):
        if (player.colliderect(east_box)):
            return True
        else:
            return False
    elif (target_direction == 'west'):
        if (player.colliderect(west_box)):
            return True
        else:
            return False
```

The extra code in the update function will check to see if the player has reached their target destination using the function reach_target.

The reach_target function returns true if the player collides with the box associated with the current target direction. If not, then it returns false.

This code is available as compassgame-collide3.py.

If you run the game, you should see instructions at the top of the screen, and if you go to the side specifying them, then you will get a new instruction.

Keeping Score

To add a scoring mechanism, there just needs to be a variable that is updated each time that the target is reached. To implement this, create a new global variable to hold the current score.

```
# Current score for this game
score = 0
```

This needs to be a global variable, so within the update function, update the global line to read

```
global direction, game_state, target_direction, score
```

The score needs to be reset at the start of each game, so add `score = 0` into the block of text where the game_state is set to "playing".

To increase the score, at the bottom of the update function, add score += 1 within the if statement that checks if the target is reached.

So

```
if (reach_target(target_direction)):
    target_direction = get_new_direction()
```

becomes

```
if (reach_target(target_direction)):
    target_direction = get_new_direction()
    score += 1
```

This will keep track of the score. To display it on the screen, you can update the draw function to also display the score. First add it as a global to the start of the draw, then add the following in the final else text block to show the score while the game is playing. You can place it just before the call to player.draw().

```
screen.draw.text('Score '+str(score),
fontsize=60, center=(WIDTH-130,50), shadow=(1,1),
color=(255,255,255), scolor="#202020")
```

You can also add the final score in the end of the game section.

```
screen.draw.text("Game Over score "+str(score)+
"\nPress space to start", fontsize=60,
center=(WIDTH/2,HEIGHT/2), shadow=(1,1),
color=(255,255,255), scolor="#202020")
```

The draw function will then look like the following:

```
def draw():
    global score
    screen.blit(BACKGROUND_IMG, (0,0))

    # If game not running then give instruction
    if (game_state == ''):
        # Display message on screen
        screen.draw.text("Press space bar to start",
        center=(WIDTH/2,HEIGHT/2), fontsize=60, shadow=(1,1),
        color=(255,255,255), scolor="#202020")
    elif (game_state == 'end'):
        screen.draw.text("Game Over "+str(score+"\nPress
        space bar to start again", center=(WIDTH/2,HEIGHT/2),
        fontsize=60, shadow=(1,1), color=(255,255,255),
        scolor="#202020")
    else:
        screen.draw.text(target_direction, center=(WIDTH/2,50),
        fontsize=60, shadow=(1,1), color=(255,255,255),
        scolor="#202020")
        screen.draw.text('Score '+str(score),
        fontsize=60, center=(WIDTH-130,50), shadow=(1,1),
        color=(255,255,255), scolor="#202020")
        player.draw()
```

This is included in the source code as compassgame-score.py.

If you run the updated code, you will see the score increase as you reach each target.

Adding a Countdown Timer

Finally, there needs to be something to make it a challenge. Otherwise, you can just keep going forever between the sides. Without any form of challenge, I'm sure most would get bored very quickly.

To add a challenge, there will be a timer so that the player needs to move around the screen in a set amount of time. The timer will start at a fixed time, for example, 10 seconds, giving the player time to reach the target. If they succeed, then the timer is reset but decremented slightly to make it a little harder. If they are unable to complete in the time, then it's game over.

A crude way of calculating the time is to consider how regularly the update function is run. In Pygame Zero the update function is normally called 60 times per second or approximately 0.016 seconds, so by counting the number of times the function is called, you can work out how long the player has had to complete the task. The problem with this is that the frequency of the loops is not guaranteed; if the computer is busy, then it may take longer between updates giving the player an unfair advantage. Instead the code should track how much time has elapsed since the last time the update function is called. This can be achieved by adding a parameter to the update() method to find out how long since the last run. To do this, replace update() with `update(time_interval)`. The time_interval variable will be set with the number of seconds since the last time update was run (which should be approximately 0.016).

To implement this, add the following global variables:

```
# Number of seconds to play when the timer starts
  timer_start = 10.9
# number of seconds to decrement the timer each time we score a
  point
timer_decrement = 0.2
# This is the actual timer set to the initial start value
timer = timer_start
```

Add the timer variable to the global variable list in the update method (there is no need to add the other new variables as we don't need to change those).

In the block of code which handles when the keyboard.space key is pressed for the start of the game, add

```
timer = timer_start
```

Just before the direction keys are pressed, decrement the timer and check we haven't gone below 0.9.

```
# Update timer with difference from previous
timer -= time_interval
# Check to see if timer has run out
if (timer < 0.9):
    game_state = 'end'
    return
```

Then after the score is increased (each time the target is reached), the timer needs to be reset (but including a decrement based on the current score).

```
# Update timer - subtracting timer decrement for each
  point scored
timer = timer_start - (score * timer_decrement)
```

Finally, to see timer on the screen, add timer as a global to the draw function and add the following displayed at the same time that the Score is shown on the screen.

```
screen.draw.text('Time: '+str(math.floor(timer)),
fontsize=60, center=(100,50), shadow=(1,1),
color=(255,255,255), scolor="#202020")
```

You may be wondering why the timer is set to 10.9 seconds for a 10-second countdown.

This is because the print uses the floor function to strip off any fractions and display the timer in whole seconds. The player will expect the game to end as soon as the timer display reaches zero and not continue to count for a further second if we instead tested for the timer being above zero. Also, the player will also expect the timer to stay on 10 for 1 second and not go to 9 once we subtract the first time interval. Starting the timer at 10.9 seconds and ending at less than 1 second will be almost exactly 10 seconds, and the user will see the values from 10 to 0.

Final Code for Compass Game Version 0.1

You will now have a complete game that you can play. When you reach the end, then it will tell you your score. You can then press space to try the game and see if you can beat that score. The complete listing of the game, so far, is included in Listing 3-4. This is also included in the source code as compassgame-v0.1.py.

Listing 3-4. Compass game. A simple Pygame Zero program with image background

```
import random
import math

WIDTH = 800
HEIGHT = 600

BACKGROUND_IMG = "compassgame_background_01"

game_state = "
target_direction = "

#Player character
player = Actor('compassgame_person_down_1', (WIDTH/2,HEIGHT/2))
# Which image is being displayed
```

```
player_step_count = 1
# Direction that player is facing
direction = 'down'

# Number of seconds to play when the timer starts
timer_start = 10.9
# number of seconds to decrement the timer each time we score a
  point
timer_decrement = 0.2
# This is the actual timer set to the initial start value
timer = timer_start

#Rectangles for compass points for collision detection to
ensure player is in correct position
box_size = 50
north_box = Rect((0, 0), (WIDTH, box_size))
east_box = Rect((WIDTH-box_size, 0), (WIDTH, HEIGHT))
south_box = Rect((0, HEIGHT-box_size), (WIDTH, HEIGHT))
west_box = Rect((0, 0), (box_size, HEIGHT))

# Current score for this game
score = 0

def draw():
    global score, timer
    screen.blit(BACKGROUND_IMG, (0,0))

    # If game not running then give instruction
    if (game_state == ''):
        # Display message on screen
        screen.draw.text("Press space bar to start",
        center=(WIDTH/2,HEIGHT/2), fontsize=60, shadow=(1,1),
        color=(255,255,255), scolor="#202020")
    elif (game_state == 'end'):
```

```
        screen.draw.text("Game Over "+str(score)+"\nPress
        space bar to start again", center=(WIDTH/2,HEIGHT/2),
        fontsize=60, shadow=(1,1), color=(255,255,255),
        scolor="#202020")
    else:
        screen.draw.text(target_direction, center=(WIDTH/2,50),
        fontsize=60, shadow=(1,1), color=(255,255,255),
        scolor="#202020")
        screen.draw.text('Score '+str(score),
        fontsize=60, center=(WIDTH-130,50), shadow=(1,1),
        color=(255,255,255), scolor="#202020")
        screen.draw.text('Time: '+str(math.floor(timer)),
        fontsize=60, center=(100,50), shadow=(1,1),
        color=(255,255,255), scolor="#202020")
        player.draw()

def update(time_interval):
    # Need to be able to update global variable direction
    global direction, game_state, target_direction, score,
    timer_start, timer_decrement, timer

    # If state is not running then we give option to start or
      quit
    if (game_state == '' or game_state == 'end'):
        # Display instructions (in draw() rather than here)
        # If space key then start game
        if (keyboard.space):
            game_state = "playing"
            timer = timer_start
            target_direction = get_new_direction()
        # If escape then quit the game
```

```
        if (keyboard.escape):
            quit()
        return

    # Update timer with difference from previous
    timer -= time_interval
    # Check to see if timer has run out
    if (timer < 0.9):
        game_state = 'end'
        return

    # Check for direction keys pressed
    # Can have multiple pressed in which case we move in all
      the directions
    # The last one in the order below is set as the direction
      to determine the
    # image to use
    new_direction = ''
    if (keyboard.up):
        new_direction = 'up'
        move_actor(new_direction)
    if (keyboard.down):
        new_direction = 'down'
        move_actor(new_direction)
    if (keyboard.left) :
        new_direction = 'left'
        move_actor(new_direction)
    if (keyboard.right) :
        new_direction = 'right'
        move_actor(new_direction)
    # If new direction is not "" then we have a move button
      pressed
```

```
    # so set appropriate image
    if (new_direction != '') :
        # Set image based on new_direction
        set_actor_image (new_direction)
        direction = new_direction

    if (reach_target(target_direction)):
        target_direction = get_new_direction()
        score += 1
        # Update timer - subtracting timer decrement for each
          point scored
        timer = timer_start - (score * timer_decrement)

def reach_target(target_direction):
    if (target_direction == 'north'):
        if (player.colliderect(north_box)):
            return True
        else:
            return False
    elif (target_direction == 'south'):
        if (player.colliderect(south_box)):
            return True
        else:
            return False
    elif (target_direction == 'east'):
        if (player.colliderect(east_box)):
            return True
        else:
            return False
    elif (target_direction == 'west'):
        if (player.colliderect(west_box)):
            return True
```

```
        else:
            return False

def move_actor(direction, distance = 5):
    if (direction == 'up'):
        player.y -= distance
    if (direction == 'right'):
        player.x += distance
    if (direction == 'down'):
        player.y += distance
    if (direction == 'left'):
        player.x -= distance

    # Check not moved past the edge of the screen
    if (player.y <= 30):
        player.y = 30
    if (player.x <= 12):
        player.x = 12
    if (player.y >= HEIGHT - 30):
        player.y = HEIGHT - 30
    if (player.x >= WIDTH - 12):
        player.x = WIDTH - 12

# Show image matching new_direction and current step count
def set_actor_image (new_direction):
    global player, player_step_count

    step_delay = 5
    player_step_count += 1

    if player_step_count >= 4 * step_delay:
        player_step_count = 1
```

```
    player_step_position = math.floor(player_step_count / step_
    delay) +1
    player.image = "compassgame_person_"+new_direction+"_"+
    str(player_step_position)

def get_new_direction():
    move_choices = ['north', 'east', 'south', 'west']
    return random.choice(move_choices)
```

The complete game is about 170 lines of code, including comments and blank lines. This may sound a lot, but it's much less than it would have been in many other programming languages.

Summary

This chapter has introduced Pygame Zero as well as creating a first graphical game. The code is quite long, which reflects the effort involved in creating a game, but it's much shorter than the equivalent code that would be needed in many other programming languages.

The game is quite basic at the moment and will be developed further in the next chapter which is on game design.

CHAPTER 4

Game Design

Hopefully you've had chance to play the game from Chapter 3 before moving on to this chapter. What did you think of it?

If your experience is like mine, then the first few goes were quite fun, but then the enjoyment dropped off somewhat. Two reasons for this: one is that once you've memorized the moves, it's quite straightforward to play and is in fact a bit too simple, and the other thing is that because of the way the timer reduces with every level, it quickly gets to the point where there is too little time to make the move, which means you get end up with a similar score on each game.

In this chapter we will look at what makes a game interesting to play and how we can make a few changes to improve the game. This forms the basis of game design.

What Makes a Game Enjoyable?

Before we look at adding any code, think about the games you have played and what makes them enjoyable. Here are a few of the things I came up with, perhaps you can think of other factors:

- Challenging but achievable
- Choices and consequences
- Rewards and progress

© Stewart Watkiss 2020
S. Watkiss, *Beginning Game Programming with Pygame Zero*,
https://doi.org/10.1007/978-1-4842-5650-3_4

- Likeable characters
- Storyline/historical relevance
- Educational (sometimes)
- Takes an appropriate level of time to play
- Inclusivity
- Age appropriate

These are not required for all games. Think of them as being guidelines that make you think about the game design, but without being too restrictive. Being aware of when you can include these features can make a game more enjoyable. These may also relate to each other, such as how rewards can help overcome a challenge or where progress is used to reveal the next part in a storyline.

When designing a new game, it's a good idea to work through these and think about how they can be implemented in the game. If you don't think it's important to your game, then that's fine.

There is no single answer to all these features, and it really depends upon what type of game you want to create and who your target audience is.

Challenging but Achievable

When you are playing a game, you want to be able to feel you have achieved something. This is often achieved by having a challenge in the game that you need to overcome. The challenge may be a skill; it may be about quick reactions; or it may involve having to use brain power to solve a puzzle.

There are some games that are popular that don't provide a challenge, but they normally provide something else. If you think about the paint by number apps, they are not what you would normally consider challenging, but instead are relaxing or therapeutic; some games may be creative rather

than competitive such as *Minecraft* in creative mode. Arguably you could say that the lack of challenge means that they wouldn't be classed as games, which is something to think about.

In most games there is a balance between it being easy to play and challenging enough to feel like you have achieved something. Make a game too easy and the player may get bored and look for a new challenge elsewhere, make it too hard and they may give up thinking they can't progress any further.

In general, you will want the game to start easy, so that the player understands how to play without facing too much challenge. Then as they progress through the game, it should get harder to make it challenging and give the player a sense of achievement.

When thinking about how to make a game challenging, you should think about whether the game will be predictable or whether there will be random elements. A predictable game would react in exactly the same way each time it is played. This means that every game has the same level of difficulty, but with lots of practice a player may learn the level. With a random element, the game is less predictable, and the player will need to adapt their play to fit the game.

Choices and Consequences

Some games create choices that the player needs to make. Some choices just change the look or feel of the game (perhaps a different color costume), but I am really talking about choices that determine the play of the game. These could be a choice of direction, a decision of whether to battle or choose diplomacy, or what technology to pursue. This is a particularly good way of making a game challenging and having the player feel in control of the game. If providing a choice, then there should normally be a consequence to the choice that the player made which determines how they progress through the game.

Rewards and Progress

When a game includes a challenge, then it's useful to reward the player which gives them a feeling of satisfaction that it was worth the effort. The reward can be just progressing through the levels (level up), or it could involve unlocking a new character or a power-up. These power-ups often can work in conjunction with the challenge where they help in completing the next level.

Likeable Characters

Many computer games put you in the role of a particular character or control of a team of characters. A character in your game may be specially created for your game, or it may be related to an existing franchise such as film or TV.

You may want to try and create a game that relates to your favorite film, perhaps a Harry Potter Wizard game, but you are likely to come across copyright issues. If it is for an existing franchise, then you need to be aware of the copyright and licensing restrictions. In general, if you use anything based around a place from a film, TV, or well-known character, then you will need to get permission from the owner of the franchise.

If you create your own characters, then you can give them their own personality and traits so that players can associate with them. In some cases, the characters can become personalities in their own right, just think about the Lara Croft character who started as a video character and was made into a film.

Also remember that characters don't have to be people. They could be creatures or vehicles, or you could even make inanimate objects come to life.

Storyline/Historical Relevance

One thing that is often optional is whether the game follows a storyline or is set in a historical story. A story can help the player to relate more to the game and make them feel a part of the story. This can be a powerful motivation to keep playing the game.

A historical relevance is where you base your game around a real moment in history. A popular one is to have a game that is associated with a historic battle or an important time in history such as birth of the railways.

There are however many games that don't have any kind of storyline and you just play for fun. It all depends upon the type of game you want to create.

Educational

Another optional aspect is whether the game is educational or not. This can include traditional children's educational games such as addition and multiplication games, adult "brain games", games to help teach you to play a musical instrument, or perhaps games that include references to historical events.

These can be an obvious goal or just a subtle feature to the game play. This can then tie into the reward, but instead of just a badge on the screen, the player can have the feeling that they have learned something that they can use away from the computer. They could also be very subtle, perhaps learning history through the storyline or by learning how to overcome an obstacle.

Takes an Appropriate Level of Time to Play

When thinking about how long it takes to play, you need to think of how the player is going to be playing. Is this a game you expect them to sit down at for a long time or something they may use to pass away a few minutes that they have spare during the day?

You should also think about whether the game can be saved and how long it can go between saves. It can be very frustrating if you have spent a long time trying to complete a level, but then don't have the time to finish it. If you can save and resume that level, that may avoid the frustration of needing to be elsewhere.

Inclusivity

There are several ways that a game can be made more inclusive of other people. This may include additional/simplified controls for those with a disability that may find traditional keyboard controls difficult to use. Or it may include the ability to have different characters to represent the gender or skin color of those that may play the game.

It can be just as important to make sure you don't use any negative stereotypes. In the past female characters have been used as damsels in distress, waiting for a male knight to come to her rescue. Thankfully, these are now becoming less common with more female characters taking a lead role in films and computer games.

Keeping these thoughts in mind when developing the game, there may be some simple things that can be implemented to make the game more appealing to a more diverse group of people.

Age Appropriate

Finally, I will mention that a game should be age appropriate. The games in this book are all designed to be family-friendly. If you are aiming for older players, then you may use less family friend language, but that would may make it less suitable for others. There is a similar thing with the amount of violence or the realism of harm inflicted. The target age of the game should also be reflected in the type of graphics used which will be considered more in the next chapter on graphics.

Improving Compass Game

Taking some of these suggestions, there are some things that can be done to make compass game a bit better. It won't be possible to implement all these ideas in this chapter, but you will be able to add three new features to improve the gameplay:

1. Improve the timer so that there is more chance of completing even when the score is quite high.
2. Add some random obstacles to make the game more challenging.
3. Add a high score, which saves the highest achieved score.

These are about making the game more challenging, but also include aspects of a reward in terms of saving the high score.

Note The code used in this chapter needs the same resources as Chapter 3. You will need to copy the source code from Chapter 4 to the same directory as Chapter 3's source code.

Updated Timer

The problem with the game timer is that it decrements linearly, counting down the same length of time each go. This works well initially, but then after around 38 points scored, it gets so difficult; it is practically impossible to complete the task. What is needed is a timer function which reduces the time quite quickly at first (to create an element of challenge), but that over time it decreases less quickly giving a reasonable chance of still being able to complete the task.

This will involve some math. We will keep this simple at this stage. The formula to be used is x / (x + h). Here x is the score and h is an offset amount. We will use an offset of 10. This formula increases quickly at first, but then as x gets larger, it tends toward the value 1. To get the time for the timer, we then subtract this from the start time.

To determine the appropriate values, this was tested using the Python plot module. I won't go into details on how the code works, but the source code is provided in a file called timedecaygraph.py. If you look in the source code, you should be able to see how it works. If you would like to try running the code, you will first need to install the plotly module. Future versions of the Mu editor will include a way of installing modules, but that is not available at the time of writing. To add a module, perform one of the following:

- On a Raspberry Pi, you can install the module using

  ```
  sudo pip3 install plotly
  ```

- On other Linux distributions

  Install either the same as previously or

  ```
  sudo pip install plotly
  ```

- On Windows

  You will need to tell pip the location of the pkgs that Mu is using.

  On my computer, that is achieved using

  ```
  pip install plotly --target="c:\users\stewart\AppData\
  Local\Mu\pkgs"
  ```

  You will need to replace `stewart` with the username that Mu was installed under.

- On Mac OS X

  First create a separate directory to run the program and copy in the timedecaygraph.py file.

  Create a file called setup.cfg with the following:

  ```
  [install]
  Prefix=
  ```

Then install the package using

```
pip3 install plotly --upgrade --target /Applications/
mu-editor.app/Contents/Resources/app_packages
```

Once you have installed plotly, you can then run timedecaygraph.py from within Mu (change the mode from Pygame Zero to Python 3 first).

Depending upon your system, it may open the results in a web browser, but on others you may need to save the output as an html file and then opening it with your web browser manually.

Through adjusting the formula values, I found that the following formula worked well:

```
start_value + 1.5 - (start_value * (i/ (i + 10)))
```

See the screenshot in Figure 4-1 showing how this new formula compares with the linear decay.

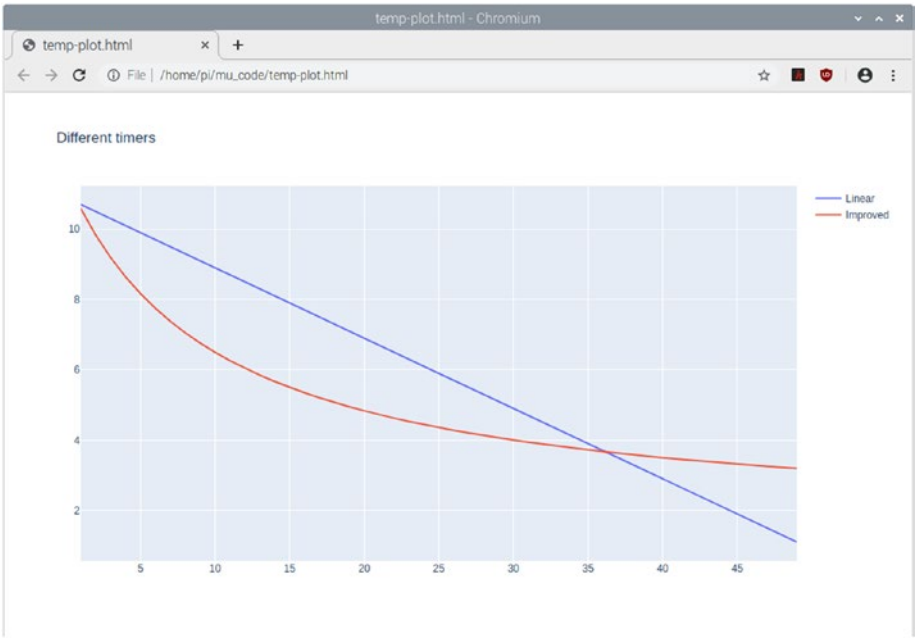


Figure 4-1. *Screenshot of graph showing different decay formulas*

As you can see from the graph, the improved formula initially decreases much quicker than the linear decay, but the decay is much smaller as the score increases.

To implement this in the code, load the current version of code from the end of the previous chapter (compassgame-v0.1.py).

Remove the timer_decrement variable as that is no longer required.

Then in the update function, replace the following entry

```
timer = timer_start - (score * timer_decrement)
```

with

```
timer = timer_start + 1.5 - (timer_start * (score/ (score + 10)))
```

The value of 10 sets the decay speed and 1.5 is used to increase the offset. These could be changed to variables if you want to be able to fine tune the values.

This is included in the source code as compassgame-timer2.py.

Adding Obstacles

The next thing we can do is to add a bit more of a challenge through adding obstacles that the player must avoid. This can be done by adding new levels. The first level does not have any obstacles, level 2 adds some obstacles, level 3 adds some different obstacles, and so on. The screenshot in Figure 4-2 shows how the game will look with some obstacles to avoid.

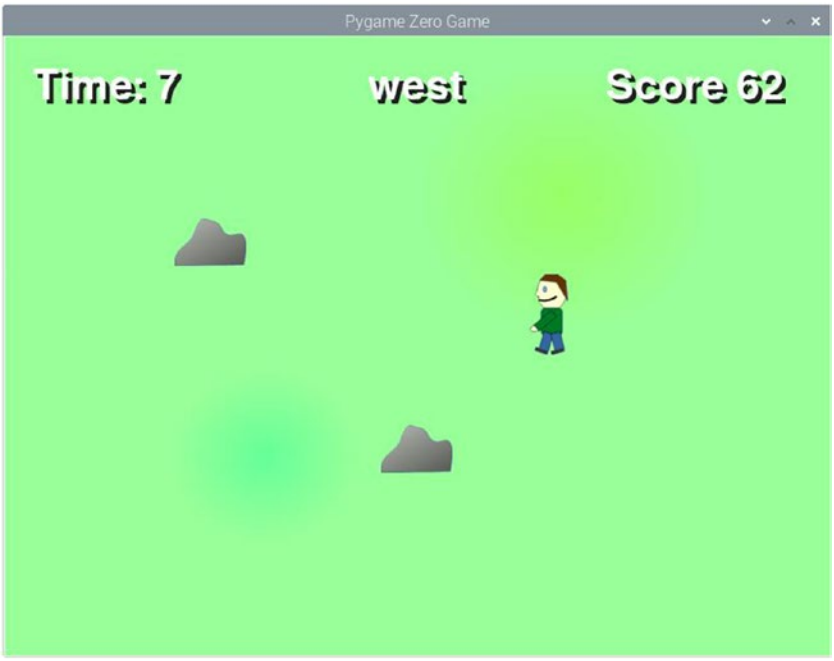


Figure 4-2. *Compass game with obstacles to avoid*

There are several changes needed for adding the obstacles. Start with the code from the end of Chapter 3 (compassgame-v0.1.py). The first is to add some more variables and definitions near the top of the file:

```
OBSTACLE_IMG = "compassgame_obstacle_01"
# Current score for this game
score = 0
# Score for each level
score_per_level = 20

# What level are we on
level = 1

#Obstacles - these are actors, but stationary ones - default
positions
```

```
obstacles = []
# Positions to place obstacles Tuples: (x,y)
obstacle_positions = [(200,200), (400, 400), (500,500),
(80,120), (700, 150), (750,540), (200,550), (60,320), (730,
290), (390,170), (420,500) ]
```

To display the obstacles, add this to the draw function making sure it's not within any of the if-else clauses.

```
    for i in range (0,len(obstacles)):
        obstacles[i].draw()
```

Add a new set_level function which creates obstacle Actors. This can be toward the end of the tile.

```
def set_level(level_number):
    global level, obstacles, obstacle_positions

    level = level_number

    # Reset / remove all obstacles
    obstacles = []
    if (level < 1):
        return
    # Add appropriate number of obstacles - up to maximum
      available positions
    for i in range (0,len(obstacle_positions)):
        # If we have already added more than the obstacle level
          number then stop adding more
        if (i >= level_number - 1):
            break
        obstacles.append(Actor(OBSTACLE_IMG, obstacle_
        positions[i]))
```

This function is to be called whenever the level increases. As well as updating the global variable for the level number, it also creates the obstacles to be avoided.

The obstacles list starts out as empty, so no obstacles are drawn. When the level is changed above level 1, then new obstacles are created. These are added as Actors, but unlike our player, they won't be able to move around the screen.

You will need to make sure that the obstacle image exists; otherwise, the program may hang with no error message making it difficult to know what has gone wrong.

Update the if(reach_target(target_direction)): block of code which is located near the bottom of the update function.

```
if (reach_target(target_direction)):
    target_direction = get_new_direction()
    score += 1
    # check if we need to move up a level
    if (score >= level * score_per_level):
        set_level(level + 1)
    # Level score is the number of points scored in this
      level
    level_score = score - ((level - 1) * score_per_level)
    # Update timer - subtracting timer decrement for each
      point scored
    timer = timer_start + 1.5 - (timer_start * (level_
    score/ (level_score + 10)))
```

In this code the level increases every 20 levels. There will be no obstacles until 20 points are scored, then one obstacle will be added, and the second obstacle at 40 points and so on. This gives a reasonable level of difficulty for each level but is a lot of time to be playing when testing the game during development. You may want to reduce the value of score_per_level to 10 so that you can test that the obstacles are created correctly

without needing to play for a long time. This is a common thing to do when developing games. In some games these are coded into the game as special "cheat codes" which would be used to jump direct to a certain level or add certain power-ups to help with testing.

The updated code is provided as compassgame-obstacle1.py in the source code. You can test the code and the obstacles will appear after the scoring 20 points, but the player is able to walk straight through them. Clearly some extra code is needed to do something when the player bumps into them. This is done by adding the following block of code at the end of the update function:

```
# detect if collision with obstacle (game over)
for current_obstacle in obstacles:
    if player.colliderect(current_obstacle):
        game_state = "end"
        return
```

This is the same as the code that is used to detect when the player reaches one of the sides of the game area but using a loop to compare against each of the obstacles in the list. If the player collides with an obstacle, then the game is set to the "end" state which triggers the end of the game. The code so far is included as compassgame-obstacle2.py in the source code.

Adding a High Score

The next feature is to add a high score. This tells the player what the previously attained highest score is and gives the player something to aim for. Typically, a high score will store multiple values along with their name or initials, but for now you should start with a single highest score value. One thing about a high score is that it needs to be saved somewhere so that it's not lost when the computer is switched off. This will therefore cover

how to save data to a file on disk and how to read it back. In the case of the Raspberry Pi, instead of a physical hard disk, it will be stored on an SD card, but using Python it is accessed in the same way as if it was on a disk.

In recent versions of Pygame Zero, there is a storage function which provides a simple way of storing information. At the time of writing, the function is not fully documented in the Pygame Zero documents. While the traditional Python file operations are more difficult to use, they are a useful tool for any Python programming. I recommend learning the method used here which will be useful for future Python programming.

Add the following new global variable near the top of the file:

```
HIGH_SCORE_FILENAME = "compassgame_score.dat"
```

Add two new functions, one to retrieve the high score from the disk (get_high_score) and the other to save the latest high score (set_high_score). These can be added at the bottom of the file.

```
# Reads high score from file and returns as a number
def get_high_score():
    file = open(HIGH_SCORE_FILENAME, 'r')
    entry = file.readline()
    file.close()
    high_score = int(entry)
    return high_score

# Writes a high score to the file
def set_high_score(new_score):
    file = open(HIGH_SCORE_FILENAME, 'w')
    file.write(str(high_score))
    file.close()
```

The get_high_score function reads a value from a file. First it opens the file using the open function. The first argument is the filename, the second is one or more characters to denote what mode the file should be opened

in. In this case 'r' is for read, other common modes are write 'w' and append 'a'. By default, the file is opened in the default text mode, but you could access the file in binary mode by using the 'b' option. For example, to open a file as read-only binary mode, you would use 'rb'.

The file is returned as a file object which can then be used for reading the file. The function uses the file object with the readline method which will read a line from the file. Subsequent calls to readline will read in further lines. In this case we only have a single entry, so it only needs to be called once.

As the high score has been stored into a text file, it will be a string rather than as a number. As we need to be able to compare it to a number, it needs to be converted from a character to an integer using the int function. The resulting value is then returned.

You will also notice that there is a line file.close() which closes the file when the function has finished reading it. This is needed to free the file up, so that it can be opened by this or another program later.

The set_high_score function works in a similar way to get_high_score, but it is writing to the file instead of reading from it. First the global variable high_score is updated and then it opens the file in write mode and writes the high score value converted to a string. Then the file is closed.

Inside the update function, add the following code just before the line `score = 0`:

```
high_score = get_high_score()
if (score > high_score) :
    set_high_score(score)
```

Where this is placed in the code means that the new high score is not saved until after the next game is started. This is done to keep the code simple and make it easier to read. You may like to look at checking this once the game ends instead.

Finally, the code is needed to display the high score when the game is over. Replace the current print statement for "Game Over" with the following two lines:

```
high_score = get_high_score()
screen.draw.text("Game Over\nScore "+str(score)+"\nHigh
score "+str(high_score)+"\nPress map or duck button
to start", fontsize=60, center=(WIDTH/2,HEIGHT/2),
shadow=(1,1), color=(255,255,255), scolor="#202020")
```

Try and Except

If you try and run the code now, then it will not work. Unfortunately, it fails without giving an error message, which can be frustrating. The reason for this is that there is no error checking on the file access. When the code tries to read in the high score file for the first time, then it doesn't exist. You could add code to check to see if the file exists or not, but then there are other things that can go wrong during file operations. For example, the file may exist, but the value is corrupt. To avoid having to put in lots of different checks, we can use Python exception handling with the try except blocks of code.

The try except has three steps. First the "try" block will run the code; if there are any errors (exceptions), then they can be handled using "except" blocks, and then the "finally" block will run whether an exception has occurred or not.

Listing 4-1 shows a generic example of code used for handling an exception.

Listing 4-1. Example of a try except exception handling

```
try:
    operation_that_may_fail()
except:
    print ("An exception occurred")
finally:
    print ("I run regardless")
```

Here the code tries to run operation_that_may_fail. If it triggers an exception, then the except code will run. The finally block runs regardless.

You can also catch only certain exceptions. The following code shows how you would only catch IO errors:

```
except IOError:
```

You can also use multiple except blocks for different kinds of errors. When an exception occurs, you can access the exception attributes as follows:

```
except Exception as e:
```

This will provide an Exception value in the variable e. You can display this to the console screen using `print (e)`. Exception handling is explained further in Chapter 11.

To use the try except exception handling on the access of the high score file, you can replace the two high score functions with the following new code:

```
# Reads high score from file and returns as a number
def get_high_score():
    try:
        file = open(HIGH_SCORE_FILENAME, 'r')
        entry = file.readline()
        file.close()
```

```
        high_score = int(entry)
    except Exception as e:
        print ("An error occurred reading the high score file :"
        + str(e))
        high_score = 0
    return high_score

# Writes a high score to the file
def set_high_score(new_score):
    global high_score
    high_score = new_score
    try:
        file = open(HIGH_SCORE_FILENAME, 'w')
        file.write(str(high_score))
        file.close()
    except Exception as e:
        print ("An error occurred writing to the high score
        file :" + str(e))
```

The updated code is named compassgame-highscore.py.

The way that the exception is handled in the code means that if there is an exception, the program continues to run. In the case of a read error, the high_score is just given a value of zero. This is acceptable here because the game can still be played without saving the high score. On some programs, a failure to save the data may be a critical issue and would therefore result in other actions, possibly including terminating the program.

A simple high score like this can add additional game play for a while, but eventually you will reach a point where it is difficult or even impossible to beat the score. Many games overcome this by adding different elements or by earning credits when you play which can be used to buy objects to make it easier to gain a higher score. In a military game, this may be armor or a more powerful weapon. That is beyond the scope of this book as it

would need a lot of additional code to include a reward-based system but is something you may want to consider when designing your own games.

This game has just implemented a few of the ideas. This is enough to make the game a bit more enjoyable. The compass game is never going to be a particularly good game in its current form as it is a little too simplistic. It is however a good game for demonstrating how to include graphics into a game and the basics of computer animation. The new features should give you an idea of how to implement some of these features to make your own game more interesting.

Summary

You have now seen how some additional elements can change the game play and make a game more interesting. This has been achieved by adding a new feature at a time which is a feature of agile programming.

This chapter has also shown how you can include timing elements to add a challenge element. It has then shown how files can be read and written to and how to handle errors that can occur when accessing files.

The next chapter will look at how graphics can be created and used in games.

CHAPTER 5

Graphic Design

The visual graphics are a key part of any game. They are what set the scene, set the tone of the game, and determine whether a game is visually appealing. The level of detail varies greatly between games, from the original pong games which had a simple block bat and ball to modern commercial games which may involve realistic video footage.

In an ideal world, all developers would also be great artists or have an artist that can create the graphics for them. That is not always the case, so this chapter looks at some simple ways of creating graphics suitable for use in games. Even if you have a professional artist, some programmers may create basic images known as programmer art, which is used as a place holder to demonstrate the game prior to the professional artwork being created.

To keep it simple, this book will mainly cover simple pixel art-based characters and simple 2D images. These graphics would be suitable for that retro 1980s feel or consistent with the style used in many indie games. It will also look at some other tools that are useful if you want to create some more complex 2D or 3D graphics.

The level of detail that you include in a game will depend upon your own artistic talent (or that of your graphic designer if part of a multi-person team) and the amount of time devoted to creating the graphics. Even if you are not particularly artistic when it comes to drawings, you can still create some simple cartoon style images. I created the graphics for all the games in this book; while they are unlikely to win any prizes for realism, they show that you can create some simple graphics without needing to be a professional artist.

© Stewart Watkiss 2020
S. Watkiss, *Beginning Game Programming with Pygame Zero*,
https://doi.org/10.1007/978-1-4842-5650-3_5

Creating a Theme

Before you start creating your graphics, you should decide on the style and theme of your graphics. When starting out programming, it is often a good idea to start off with simple images as those work well in the simple Actor objects that Pygame Zero uses. These will also need much less processing power compared to the lifelike characters you see in commercial AAA games. This doesn't mean your characters need to be lifeless as you can still give the characters their own style and personality.

Some other things to consider:

- What kind of environment is the game based in? Games could be based on land, on the sea, or even in space. Each location has its own challenges and advantages regarding the graphics.
- Will the graphics be realistic? Graphics can be created that create realism or that can take you to a fantasy world.
- Will the game be family friendly? If you want the game to be suitable for young children, then you should avoid violence, bad language, and other inappropriate content. If the game does include some level of violence or destruction, then comic-style violence is more suitable for children than if you use lifelike images. You may consider having a family-friendly mode with more appropriate graphics for younger players.

- Can the characters be customized? If the main character is a person, then players may like to choose a character that they can associate with. This may be through providing a different gender, color of skin, hair color, or choice in clothes. If instead the character is an animal or fantasy creature, then maybe there could be an option for different animals or creatures. This can also apply for inanimate objects, in the case of a vehicle, a different make, model, or color.

Having decided on the theme, you can create images for the background and the characters in the game.

File Formats

There are different file formats that can be used for images. The two most common are bitmap and vector formats which will be considered here.

Bitmap Images

The images that have been used so far have all been bitmap images. Bitmap images (also known as raster images) are created as individual pixels which are the smallest individual block of the image. The bitmap defines the color of each of the pixels making up the image.

This is shown in Figure 5-1. This is a simple image of 10 x 10 pixels with a white background and a black rectangle. The squares that are colored white would be stored as a white pixel, and those that are colored black would be stored as a black pixel.

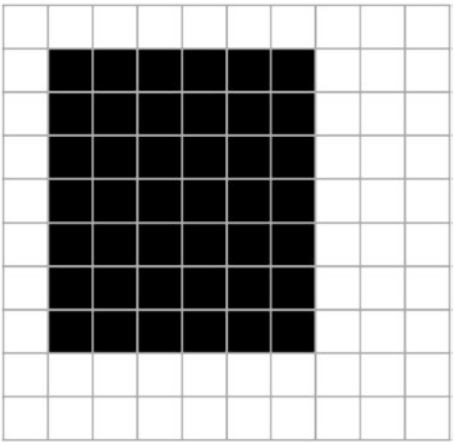


Figure 5-1. *Simple bitmap image*

This is a trivial image. Bitmap images usually consist of a lot more pixels, and so storing the color of each pixel can result in very large file sizes. For example, the background image used in the compass game is 800 x 600 pixels, which is 480,000 pixels. If 3 bytes are used to represent the color (which is typical), then that image would be about 1.4 MB in size. You could prove this yourself by converting the image to a Windows Bitmap (.bmp) image format. To avoid such large file sizes, image formats often support compression.

The two most popular image formats that are used in Pygame Zero are PNG (.png) and JPEG (.jpg). The PNG (Portable Network Graphics) format supports lossless compression. This reduces the file size but keeps all the data in the image intact. The JPEG format (created by Joint Photographic Experts Group) uses lossy compression, which removes some of the information in the file while making it look as close as possible to the original. The lossy compression often makes the files smaller but can result in loss of quality.

JPEG files are good for large images where compression is a priority. This makes them a useful format for photos.

PNG has good compression with no loss of quality and has support for transparency, so it is usually a good choice for game programming.

Vector Images

An alternative to a bitmap image is a vector image. Instead of storing details of each of the pixels, a vector image stores instructions on how to create the image from shapes. In the case of the image previously used in Figure 5-1, the file format would instead describe how to create the image using a rectangle.

Listing 5-1 shows pseudo-code for how the bitmap image could be drawn as a vector image.

Listing 5-1. Example of a try except exception handling

```
Create blank page 10 pixels x 10 pixels
Set the page color to white
Draw a rectangle starting at position 1,1 which is 6 x 7 pixels
in size.
Color the rectangle black
```

Tip Pseudo-code is used to describe how a program works. It cannot be run directly in any programming language as it doesn't have the correct vocabulary or syntax that a normal programming language needs. It is useful for explaining how the code will work.

The main advantages of a vector image are as follows:

- The shapes can be edited and moved without losing any information where overlapping other shapes.
- When zooming in on a shape, it continues to be crisp, whereas a bitmap becomes pixelated.
- Usually the file size is smaller.

A popular format for vector images is SVG (Scalable Vector Graphics) which is a generic file format. There are lots of other vector file formats, which are normally associated with a particular editing application (such as ODG which is used in LibreOffice Draw).

Pygame Zero is not able to display these images in the same way that it can with bitmap images. Vector images have to be converted to bitmap images when designing the game, converted using code that can understand the vector image format, or to have code that instructs Pygame Zero to create the images using its built-in shape tools. Each of these methods will be covered in this or the next two chapters.

Useful Tools

There are many tools that can be used to design computer graphics. The examples shown here are all freely available and will work on the Raspberry Pi. For some of these, an example of how to create an image is shown.

LibreOffice Draw

Draw is one of the applications included in the LibreOffice Office suite. It is included by default in the Raspberry Pi NOOBs image and available for other operating systems from the web site `www.libreoffice.org/`.

Draw is useful for creating 2D vector images, which can then be converted into bitmap images for using in Pygame Zero.

The screenshot in Figure 5-2 shows a person created in Draw. The figure on the right has been separated into its different components to demonstrate how these were created using basic shapes.

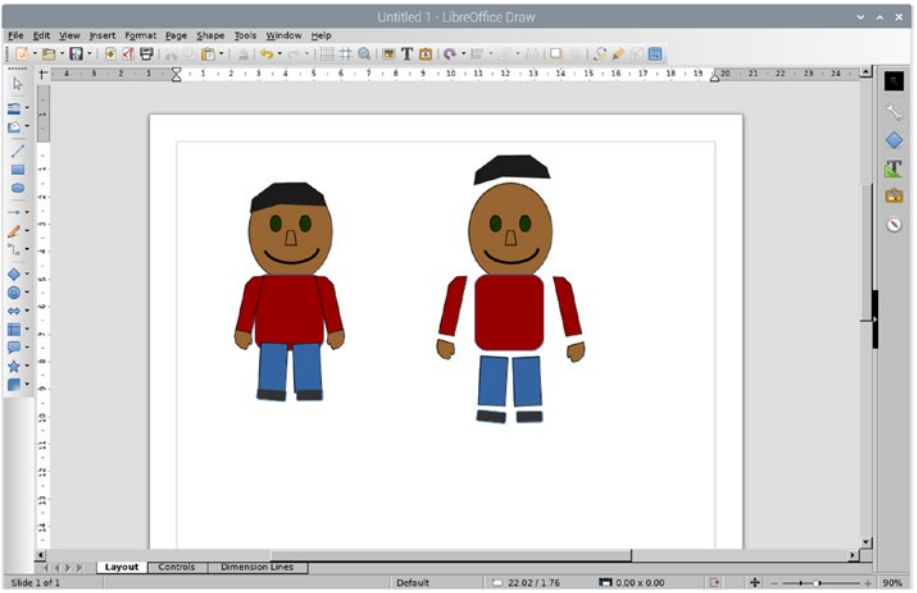


Figure 5-2. *Person sprite image created in LibreOffice Draw*

There are several different shapes that can be used which are shown in Figure 5-3. For more complex shapes, the draw tool includes an option for creating an irregular polygon using a collection of lines which can be formed in any shape.

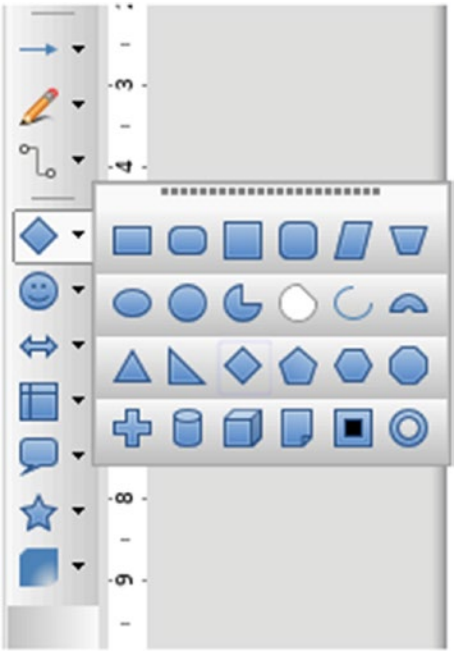


Figure 5-3. *Simple shape drawing tools in LibreOffice Draw*

After designing the sprite, it can be exported to a PNG file using the export option. If you tick the "selection" checkbox, then it will just export the selected objects. That can be useful if you create multiple images in the same document.

Inkscape

LibreOffice Draw is a good program, but for a more professional drawing application, there is another free alternative in the form of Inkscape. Inkscape is a vector drawing program which compares itself to Adobe Illustrator and CorelDRAW. It isn't included by default in the NOOBS install, but can be installed using

```
sudo apt install inkscape
```

Inkscape is also available for other operating systems and can be downloaded from https://inkscape.org/. The screenshot in Figure 5-4 shows Inkscape with a drawing of a car.

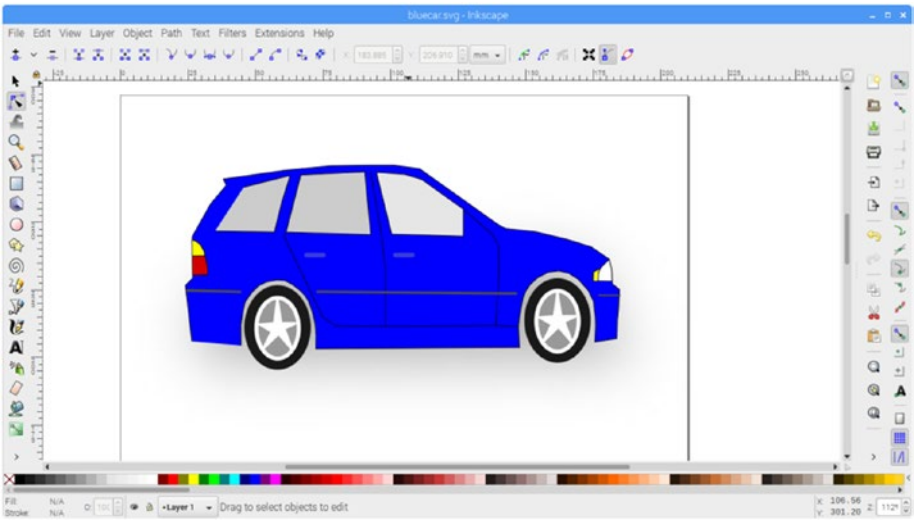


Figure 5-4. *Car image created in Inkscape*

Inkscape is a bit harder to use than LibreOffice Draw, but more powerful. If you are not already familiar with a vector drawing program, then you may like to try LibreOffice Draw first and then use Inkscape when ready to move to the next level. An example of how it works differently is that LibreOffice Draw has a polygon tool for creating irregular polygons, whereas in Inkscape this is achieved by using the pencil tool. To create a polygon, draw the first line, then start each subsequent line from the end of the previous line. When complete, clicking the beginning of the first line will result in a polygon which you can fill with color.

The Inkscape files are saved directly as SVG files which makes them useful for sharing with other applications and the images can be exported as PNG bitmap files for use in Pygame Zero.

GIMP

GIMP (GNU Image Manipulation Program) is a bitmap editor. It is a powerful tool with lots of features, but due to this, it can be difficult to learn. It can be installed on the Raspberry Pi using

```
sudo apt install gimp
```

On other operating systems, you can download a version at www.gimp.org. There are many ways that GIMP can be used for creating graphics. Two examples are shown here, one creating a background image from a drawing or photo and the other showing how it can be used to create simple pixel art suitable for sprites.

Creating a Computer Image from a Drawing or Photo

This example will show the principles behind creating a computer graphic image from a drawing or photo. This can be used to take concept artwork and make it into a background for a game. In this case I have created a computer graphic image of a castle from a photo of a castle. The photo image is first loaded into GIMP and resized to the size of the finished image as shown in Figure 5-5.

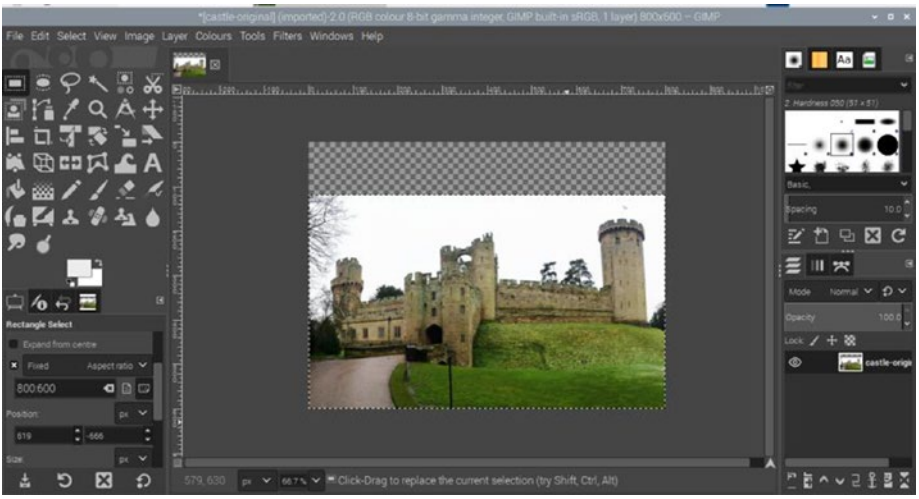


Figure 5-5. *GIMP with photo of a castle*

You will see that there is a transparent area at the top of the image (checkerboard pattern). This is due to the image being resized to achieve the desired aspect ratio.

The image will be created on a new layer and the photograph eventually removed. The new layer has been created using the layer tool as shown in Figure 5-6.

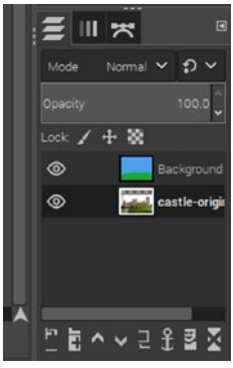


Figure 5-6. *GIMP layer dialog with new layer*

The new layer has been divided into two areas showing green land with a blue sky. The main tools that are used are the free select tool (lasso) and the fill tool (bucket); these are both highlighted in Figure 5-7.

Figure 5-7. *GIMP tools dialog showing free select and fill tools*

The order and opacity of the layers can be adjusted so that it is possible to see the photo in the background and then the outline drawn using the free select tool. You can zoom in and out using Ctrl and the mouse roller wheel. You can move around the image using the scroll bars. If you accidentally click in the wrong place, then use the backspace key on the keyboard. The selection is shown in Figure 5-8, where you can see a faint outline of the shape of the castle.

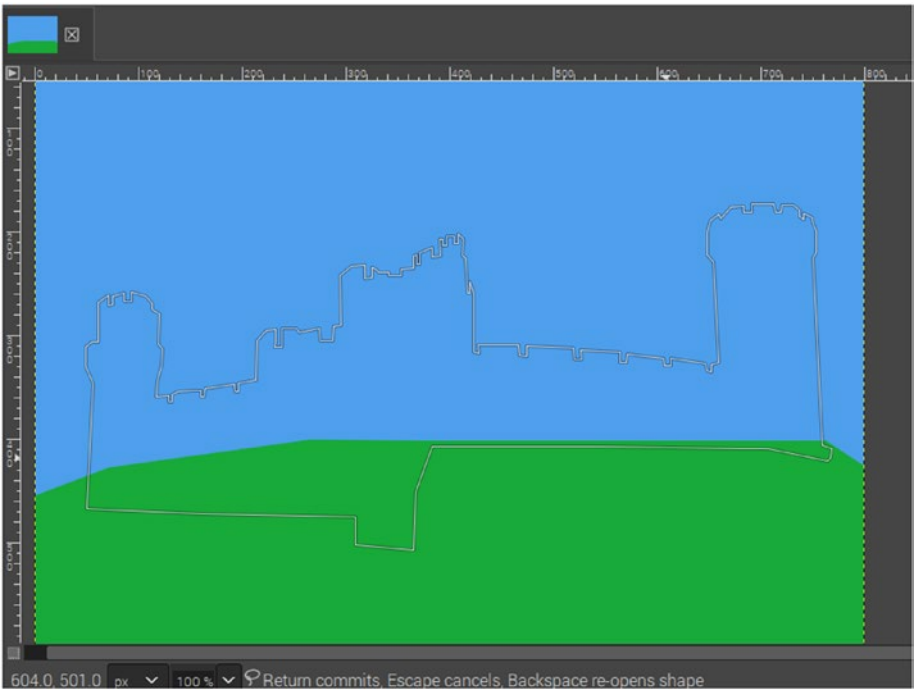


Figure 5-8. *Selection of castle outline in GIMP*

The fill tool is then used to fill the outline with the appropriate color. This is repeated to add more details, such as the door and windows. The image can be saved as a GIMP XCF file which will allow you to continue editing it and exported as a PNG file for use in Pygame Zero. The exported image of the castle is shown in Figure 5-9.

Figure 5-9. *Exported image of the castle*

This process is repeated until the appropriate level of detail is achieved. I've added a bridge, the road, and a darker green for the far side of the dry moat.

It is also possible to draw onto the image using a pencil or paintbrush. I've used the paintbrush tool to add some clouds. These have been drawn using a soft brush on two layers with partial transparency to give it a softer appearance.

Creating a Pixel Art Sprite

An alternative is to create the image completely from scratch using your own imagination. In this example, a simple pixel art sprite of a spacecraft. Start by creating a new image. Set the size to the appropriate level of detail (in this example 32 x 32 pixels), and under the more options dialog, choose the background as transparency.

You can then zoom in to the image, and using the pen with size set to 1 individually, color the relevant pixels. I started by creating a simple outline shape as shown in Figure 5-10.

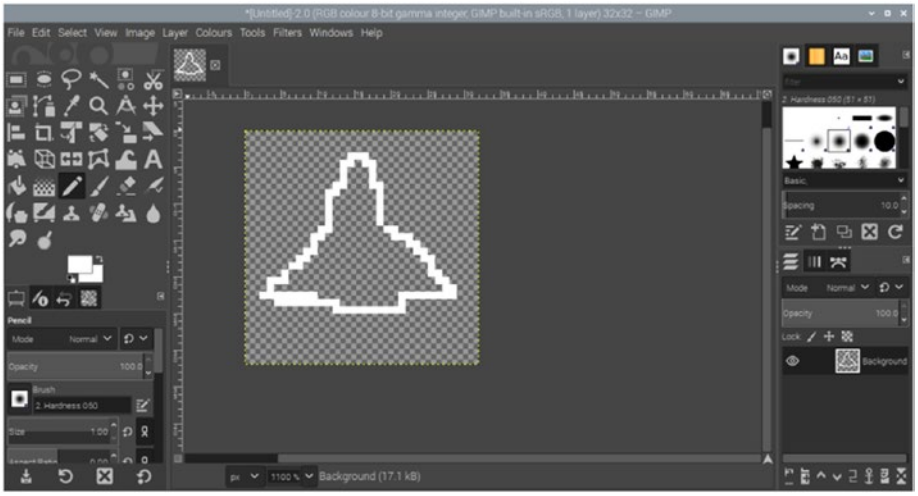


Figure 5-10. *Creating a pixel art sprite in GIMP*

To make it easier to create symmetry, I added a temporary layer with a line showing the middle of the image. You can then count the same number of pixels for each side of the line. This is shown in Figure 5-11.

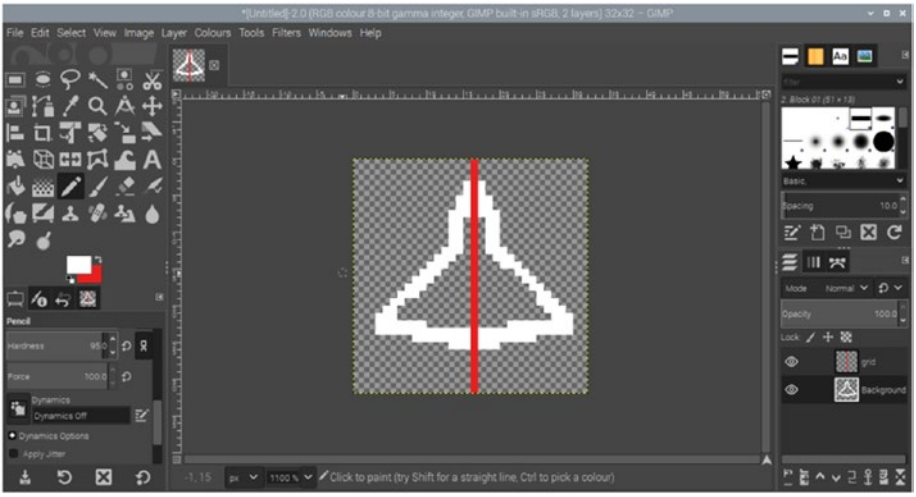


Figure 5-11. *Creating a pixel art sprite in GIMP with a line of symmetry*

Continue adding detail as necessary. Once complete, the image can be exported as a PNG file as shown in Figure 5-12. You should normally leave any unused pixels as transparent when exporting the image, but I have colored the background gray to make it easier to see the white image.

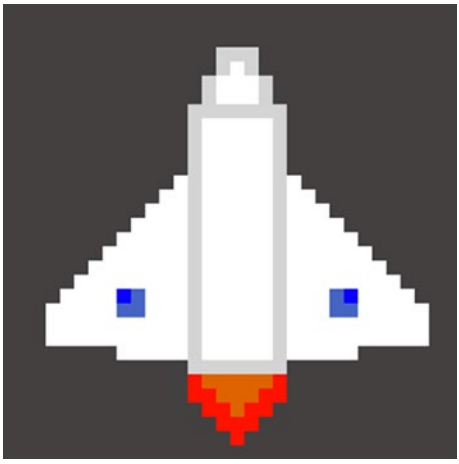


Figure 5-12. *Pixel art spacecraft*

Blender

The tools discussed so far are designed for 2D images. Blender is a 3D design tool. This can be useful for creating 3D games, but that is beyond the scope of this book; instead, I will show an example of how it can be used to create a more 3D appearance by applying lighting and shadows to a 3D model and then exporting it as a 2D image.

Blender is a professional design tool that is available for free. It can be installed on the Raspberry Pi.

```
sudo apt install blender
```

If running on a Raspberry Pi, then I suggest getting a Raspberry Pi 4 with 4GB of memory; it will run on older versions but is very slow and barely usable. For other operating systems, the program can be downloaded from www.blender.org.

Blender is an incredibly powerful tool, but it can be difficult to learn. It has tools all around the screen with multiple pull-down menus in different places, and the mouse operations work differently to the 2D tools. As a result, it can be very confusing for new users.

If you do learn it, then it can be useful. You may want to start by working through some short tutorials looking at certain aspects rather than trying to take it all in during one project.

Creating a 3D object is beyond the scope of this book, but it may be useful to understand how you can use objects created in Blender in your games. The following steps show how you can export a Blender model as a 2D image suitable for use in a game.

The image in Figure 5-13 shows a simple missile/bullet image created for a game. It is made up of a cylinder and cone. As a basic 2D object without shading, it would look very basic, but by applying a light source so that you can see the shadows, it can take on a more 3D appearance.

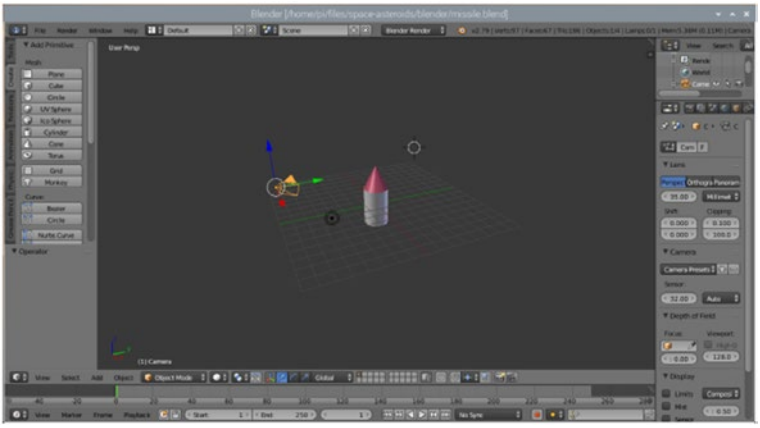


Figure 5-13. *Blender with 3D model of a missile*

After designing the image, the object can be rendered to a 2D image, then saved as an image file as shown in Figure 5-14.

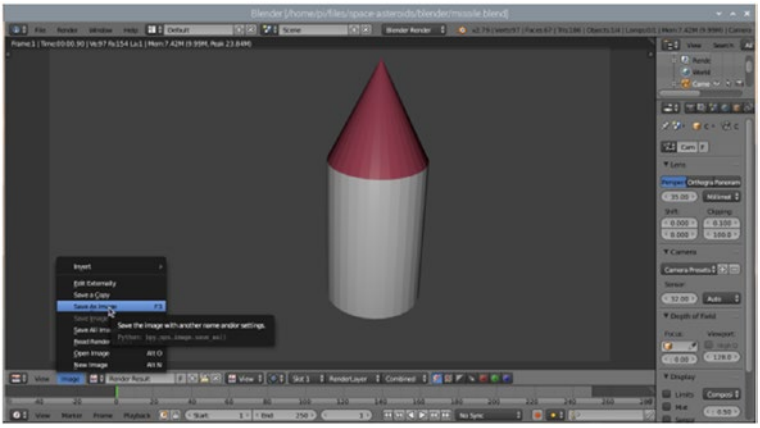


Figure 5-14. *Blender with Save As Image menu option*

Create Using Code

The tools so far have looked at ways of creating images in a tool that are then exported for use in Pygame Zero. An alternative is to generate the image in Pygame Zero using code. This can make use of the shape drawing tools within Pygame Zero.

Chapter 7 includes a game created completely from scratch using this technique. A screenshot of the game is shown in Figure 5-15.

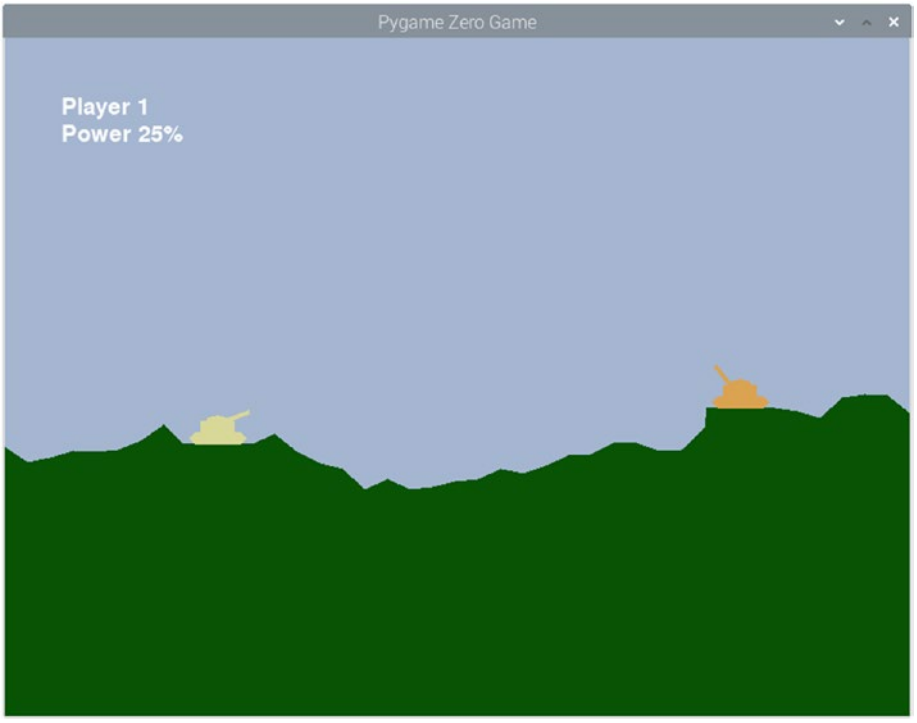


Figure 5-15. *Screenshot of tank game created using code*

The graphics in this game are basic, but more detail could be added to make them more realistic.

Other Sources

If you don't want to create your own images, then you could get some graphics created by someone else. You will need to check the licenses for the graphics allowing you to use them in your game. Some licenses may put restrictions on how the graphics can be used, modified, and distributed. They may also impose different licenses depending upon whether your game is monetized.

The following is a small selection of sources that may be useful; be aware that some of these sites can use different licenses for different images or may use licenses that restrict how the images may be used:

- Open Game Art – https://opengameart.org/
- Kenny – https://kenney.nl/
- Pixabay – https://pixabay.com/
- Itch.io free game assets – https://itch.io/game-assets/free

This is not an exhaustive list. A search using an Internet search engine will list other sites with graphics suitable for use in your own games.

Summary

This has shown some common tools that can be used for creating images for use in computer game programming. It is beyond the scope of this book to go into details of how they are used, but it has included an overview of some of the techniques that you may want to use when creating graphics. It has also included a few suggestions of sites that may have suitable graphics that can be used.

The next chapter will look at how colors are used in Pygame Zero and some techniques for using colors in game programming.

CHAPTER 6

Colors

In Chapter 3 there was a brief mention that there are different ways of defining colors. This chapter will look at the different ways that colors can be used in Pygame Zero. You will also see how the mouse can be used to interact with a program.

This chapter will use some code examples, but this chapter is not about creating a specific game; it is about learning new tools and techniques that may be useful in future.

Color Mixing

To understand the color model, it is useful to look at different ways that colors can be defined. At a young age, you would have learned that you can make up different colors from mixing different colored paints together. Through that you learned that the primary colors were blue, red, and yellow. If you look at the ink in a color printer, then you will still see this in action, but using cyan (light blue), magenta (light red), and yellow. You will also see that you have a black ink to give a true black color. This is known as the CMYK color model.

The CMYK model works well for printers because it is a subtractive color model. You start with a pale color (often white paper) and the ink that is added prevents colors from being reflected. By adding ink in specific quantities, you can filter out unwanted light to get the color you want.

© Stewart Watkiss 2020
S. Watkiss, *Beginning Game Programming with Pygame Zero*,
https://doi.org/10.1007/978-1-4842-5650-3_6

The RGB scheme used on computer screens is the opposite. Instead of blocking colors, it starts with a black screen and adds colored light to reach the desired color. Because the colors are being added rather than subtracted, it uses different colors to the subtractive color scheme. The colors used on a computer screen are red, green, and blue (RGB). There are other color schemes and there are modules in Python that can convert between the different color models, but essentially it is just RGB that is needed for most game programming.

In Pygame Zero RGB values are often entered as a tuple listing the three different color components as numbers from 0 to 255. For example, to represent orange you could use (255, 165, 0) which has 255 for the red component (the maximum), 165 for the green component, and 0 for the blue component. It can also be entered as a hexadecimal value which is the same as if it was defined in HTML or CSS. This shows the same three values but converted to hexadecimal (base 16) instead of decimal. For orange this would be #ffa500. There are also some 657 different words that can be used for colors ranging from "aliceblue" to "yellowgreen". A small selection of the color codes is shown in Figure 6-1.

aquamarine1	127,255,212	#7fffd4
black	0,0,0	#000000
blue	0,0,255	#0000ff
magenta	255,0,255	#ff00ff
gray	190,190,190	#bebebe
green	0,255,0	#00ff00
limegreen	50,205,50	#32cd32
maroon	176,48,96	#b03060
navy	0,0,128	#000080
brown	165,42,42	#a52a2a
purple	160,32,240	#a020f0
red	255,0,0	#ff0000
lightgray	211,211,211	#d3d3d3
orange	255,165,0	#ffa500
white	255,255,255	#ffffff
yellow	255,255,0	#ffff00
violet	238,130,238	#ee82ee

Figure 6-1. *List of color codes*

The code to generate this list is in Listing 6-1 and included in the source code as color-demo.py. The demonstration program displays the word, RGB, and HTML values for a selection of colors. It shows them on both a black and white background to make the colors visible.

Listing 6-1. Code to display a selection of color words with color codes

```
# Program to demonstrate some of the color words including in
  Pygame / Pygame Zero
import pygame

WIDTH = 800
HEIGHT = 600

colors = ['aquamarine1', 'black', 'blue', 'magenta', 'gray',
'green', 'limegreen', 'maroon', 'navy', 'brown', 'purple',
'red', 'lightgray', 'orange', 'white', 'yellow', 'violet']

def draw():
    screen.draw.filled_rect(Rect((400,0),(400,600)),(255,255,255))
    line_number = 0
    for color in colors:
        print_color (color, line_number)
        line_number += 1

def print_color (colorname, line_number):
    color_rgb_string = "{},{},{}".format(pygame.
    Color(colorname).r, pygame.Color(colorname).g, pygame.
    Color(colorname).b)
    color_html_string = "#{:02x}{:02x}{:02x}".format(pygame.
    Color(colorname).r, pygame.Color(colorname).g, pygame.
    Color(colorname).b)
    screen.draw.text(colorname, (20,30*(line_number+1)),
    color=colorname)
    screen.draw.text(color_rgb_string, (130,30*(line_
    number+1)), color=colorname)
    screen.draw.text(color_html_string, (250,30*(line_
    number+1)), color=colorname)
```

```
    screen.draw.text(colorname, (420,30*(line_number+1)),
    color=colorname)
    screen.draw.text(color_rgb_string, (530,30*(line_
    number+1)), color=colorname)
    screen.draw.text(color_html_string, (650,30*(line_
    number+1)), color=colorname)
```

The code uses Pygame Zero to display the text, but accesses the pygame.Color list directly. The list of colors is not available through the Pygame Zero documentation, but a link is included in Appendix B to the Pygame source code where you can see all the colors defined.

Bouncing Ball

As a further demonstration of the use of color, I have created a short program which shows a ball bouncing around the screen. The ball changes color as it moves around. I won't use this in a game, but I will explain the technique used which could be useful if you wanted to make a game that relies on bouncing a ball such as Breakout. A screenshot of the program is shown in Figure 6-2.

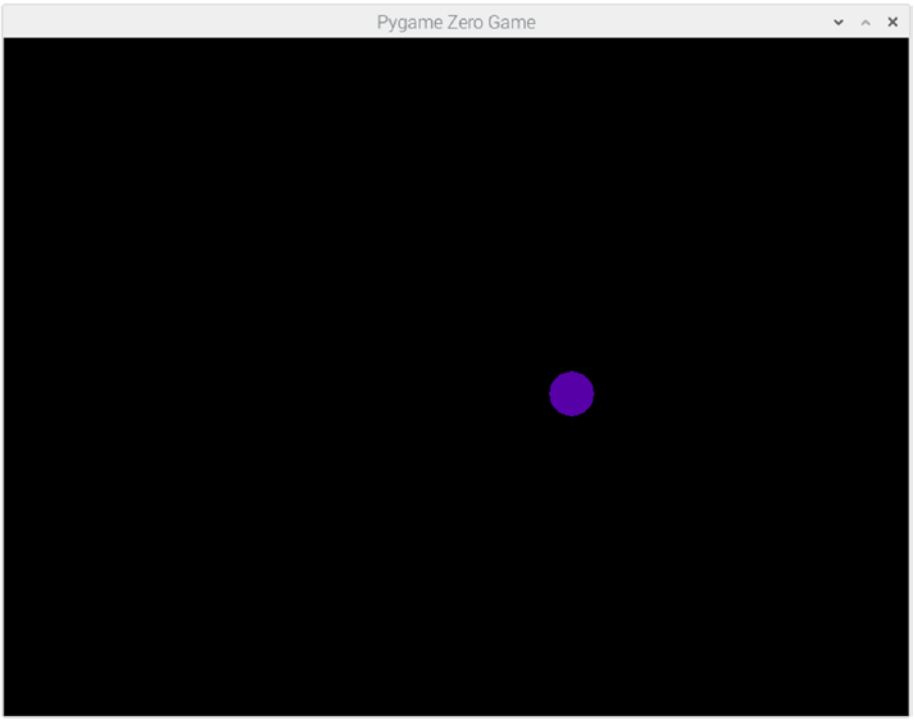


Figure 6-2. *Color bouncing ball*

The code for this is shown in Listing 6-2 and is included in the supplied source code as bouncingball.py.

Listing 6-2. Code to display a selection of color words with color codes

```
WIDTH = 800
HEIGHT = 600

# starting positions
ball_x = 400
ball_y = 300
ball_speed = 5
```

```
# Velocity separated into x and y components
ball_velocity = [0.7 * ball_speed, 1 * ball_speed]
ball_radius = 20
ball_color_pos = 0

def draw():
    screen.clear()
    draw_ball()

def update():
    global ball_x, ball_y, ball_velocity, ball_color_pos
    ball_color_pos += 1
    if (ball_color_pos > 255):
        ball_color_pos = 0
    ball_x += (ball_velocity[0])
    ball_y += (ball_velocity[1])
    if (ball_x + ball_radius >= WIDTH or ball_x - ball_radius
    <= 0):
        ball_velocity[0] = ball_velocity[0] * -1
    if (ball_y + ball_radius >= HEIGHT or ball_y - ball_radius
    <= 0):
        ball_velocity[1] = ball_velocity[1] * -1

def draw_ball():
    color = color_wheel (ball_color_pos)
    screen.draw.filled_circle ((ball_x,ball_y), ball_radius,
    color)

# Cycle around a color wheel - 0 to 255
def color_wheel(pos):
    if pos < 85:
        return (pos * 3, 255 - pos * 3, 0)
```

```
    elif pos < 170:
        pos -= 85
        return (255 - pos * 3, 0, pos * 3)
    else:
        pos -= 170
        return (0, pos * 3, 255 - pos * 3)
```

As with all Pygame Zero code, the code is based around the draw and update functions.

The update function handles the movement of the ball. The ball has a velocity (combination of speed and direction) which is stored in terms of the change in x and y for each run of the update function. Using the default speed of 5, the ball will move 3.5 pixels in the X direction and 5 pixels in the Y direction each time the function is called. When the ball hits a wall, then it's velocity in the appropriate direction will be reversed.

The draw function runs the draw_ball function which draws the ball using screen.draw.filled_circle. It works out the color for the ball from a color_wheel function.

The color wheel is created in three phases. The first phase starts with no red light, full green light, and no blue light. Then red light is increased, and blue light decreased as you move around this phase.

The second phase is where the red light decreases and blue light increases, with no green light.

The third phase is where green light increases and blue light decreases, with no red light.

This uses just one slice around the wheel with a fixed amount of brightness. The total number of colors available is over 16 million, but because it only takes one slice, the color_wheel function will return one of 256 different colors each time it is called. Using the next color, each time the ball is drawn means that the ball will change color as it moves around the screen.

Background Color Selector

To help visualize the different colors, the next program will provide a means of viewing colors associated with different color codes.

The program allows the user to select a color, and it will be displayed across the bottom half of the window. This is shown in Figure 6-3.

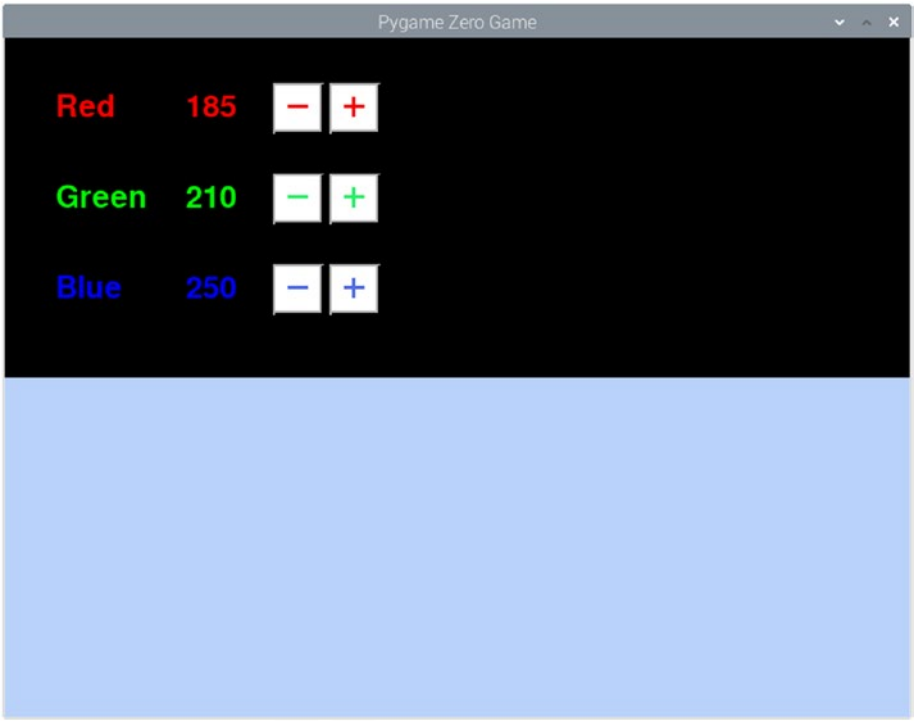


Figure 6-3. *Color selector program*

Like the rest of this chapter, it won't involve creating a complete game, but it will demonstrate techniques that can be used in creating games. This includes how to handle mouse events to create games using the mouse.

Handling Mouse Events

When the mouse is moved, clicked, or dragged, then it causes an event to be triggered. These then call mouse event functions which you can implement in your own code. These functions are on_mouse_down, on_mouse_up, and on_mouse_move. If you implement these functions in your Pygame Zero code, then they will be called whenever one of the events is triggered.

Looking at the function on_mouse_down, it is triggered each time that one of the mouse buttons is pressed. The function can have two arguments; if they are included in the function, then they will be provided with the position of the mouse and the mouse button pressed.

An example function is shown in Listing 6-3.

Listing 6-3. Code to handle mouse press

```
def on_mouse_down(pos, button):
    if (button == mouse.LEFT):
       print ("Mouse pressed, position {} {}".format((pos[0]),
       pos[1]))
```

Using this code each time the left button is pressed, it will print out the coordinates of the mouse to the console. If there are actors on the screen, then it is possible to detect whether the mouse is over one of those actors using the actor collidepoint method. This is different than if using a conventional (non-game) application. In a game you normally want the action (such as pressing a button, firing a laser, or turning a card) as soon as the mouse is clicked. In a conventional application, to press a button, you normally press on the button and then also need to release it while the mouse is over the same point. This means keeping track of whether the button was during the on_mouse_down, then waiting until after a on_mouse_up is called. As this is a game programming book, it will just cover the first, but it's something you may want to consider if using Pygame Zero for a non-game application.

Creating the Color Selector

The color selector creates a filled_rectangle with the selected color. The rectangle takes up half of the program window. This is like the filled_circle used previously, except it uses a Rect object. The color is set based on variables for color_red, color_green, and color_blue. The value of each of those is set using plus and minus button using the on_mouse_down function. These buttons are images which are created as actor objects the same as if creating a character or other sprite.

The code for the color selector is shown in Listing 6-4.

Listing 6-4. Color selector program

```
WIDTH = 800
HEIGHT = 600

color_red = 0
color_green = 0
color_blue = 0

change_amount = 5

BOX = Rect((0,300),(800,300))

button_minus_red = Actor("button_minus_red", (260,63))
button_plus_red = Actor("button_plus_red", (310,63))
button_minus_green = Actor("button_minus_green", (260,143))
button_plus_green = Actor("button_plus_green", (310,143))
button_minus_blue = Actor("button_minus_blue", (260,223))
button_plus_blue = Actor("button_plus_blue", (310,223))

def draw() :
    screen.clear()

    screen.draw.text("Red", (45,45), fontsize=40, color="red")
    screen.draw.text(str(color_red), (160,45), fontsize=40,
    color="red")
```

```
    screen.draw.text("Green", (45,125), fontsize=40,
    color="green")
    screen.draw.text(str(color_green), (160,125), fontsize=40,
    color="green")
    screen.draw.text("Blue", (45,205), fontsize=40,
    color="blue")
    screen.draw.text(str(color_blue), (160,205), fontsize=40,
    color="blue")

    button_minus_red.draw()
    button_plus_red.draw()
    button_minus_green.draw()
    button_plus_green.draw()
    button_minus_blue.draw()
    button_plus_blue.draw()

    screen.draw.filled_rect (BOX, (color_red,color_green,
    color_blue))

def update() :
    pass

def on_mouse_down(pos, button):
    global color_red, color_green, color_blue
    if (button == mouse.LEFT):
        if (button_minus_red.collidepoint(pos)):
            color_red -= change_amount
            if (color_red < 1):
                color_red = 0
        elif (button_plus_red.collidepoint(pos)):
            color_red += change_amount
            if (color_red > 255):
                color_red = 255
```

```
elif (button_minus_green.collidepoint(pos)):
    color_green -= change_amount
    if (color_green < 1):
        color_green = 0
elif (button_plus_green.collidepoint(pos)):
    color_green += change_amount
    if (color_green > 255):
        color_green = 255
elif (button_minus_blue.collidepoint(pos)):
    color_blue -= change_amount
    if (color_blue < 1):
        color_blue = 0
elif (button_plus_blue.collidepoint(pos)):
    color_blue += change_amount
    if (color_blue > 255):
        color_blue = 255
```

The on_mouse_down function handles all the button presses. There is a block of text for each button which looks to see if the button collides with the position of the mouse. If a collision is detected, then it increases or decreases the value of the appropriate color by 5. The reason for changing by 5 rather than 1 is to reduce the number of button clicks needed, although that does mean that only a subset of colors can be displayed.

Summary

This chapter has looked at how colors are created in Pygame Zero and how the colors can be used. The bouncing ball program showed how the colors can be used. The color selector provides a way of creating different colors and how to use the mouse to interact with the program. The code used in these programs can be used as a building block for creating games.

In the next chapter, the colors will be used to create another game using vector images.

CHAPTER 7

Tank Game Zero

The last few chapters have covered some theory; now you will get a chance to apply some of those techniques into a new game. The game is an artillery battle game called Tank Game Zero – a battle to destroy your enemy's tank.

The game will use some of the features learned in the previous chapters and expand on those. It will use dynamic vector graphics to create sprites and the background image. It will also cover a new technique for tracking a trajectory for shells fired from a tank's gun.

Rather than cover the program line by line, I've explained some of the different techniques used to create the game. These will come together toward the end of this chapter to create a working game.

The game is a two-player turn-based game. Player 1 will fire a shell toward the enemy tank to try and destroy it. If that is unsuccessful, then player 2 has a go. This repeats until one of the players' shells successfully hits the opponent's tank.

Vector Image of Tank

Instead of using bitmap images, the game is created using the built-in Pygame Zero shapes. This includes the landscape which is created as a polygon and the tanks which are created using simple shapes. The basic outline for the tank is shown in Figure 7-1.

© Stewart Watkiss 2020
S. Watkiss, *Beginning Game Programming with Pygame Zero*,
https://doi.org/10.1007/978-1-4842-5650-3_7

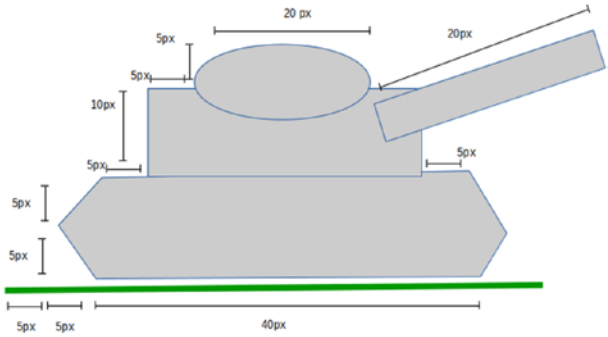


Figure 7-1. *Tank shape created using vector shapes*

In the code the bottom part of the tank is known as the track, which is created as a polygon; the main part is known as the hull, created as a rectangle; the top is the turret which is an ellipse; and the gun is a rectangle shape but created as a polygon.

This is going to need additional code to work out the position of the tank and the relative coordinates of the different shapes. The math for drawing the gun position is going to be quite involved so that is put into a separate function. The code to draw one of the tanks is shown in Listing 7-1. This is included in the source code as tankshape.py.

Listing 7-1. Code to display a tank created using shapes

```
import math
import pygame

WIDTH=800
HEIGHT=600

left_tank_position = 50,400
left_gun_angle = 20

def draw():
    draw_tank ("left", left_tank_position, left_gun_angle)
```

```
def draw_tank (left_right, tank_start_pos, gun_angle):
    (xpos, ypos) = tank_start_pos
    tank_color = (216, 216, 153)

    # The shape of the tank track is a polygon
    # (uses list of tuples for the x and y co-ords)
    track_positions = [
        (xpos+5, ypos-5),
        (xpos+10, ypos-10),
        (xpos+50, ypos-10),
        (xpos+55, ypos-5),
        (xpos+50, ypos),
        (xpos+10, ypos)
    ]
    # Polygon for tracks (pygame not pygame zero)
    pygame.draw.polygon(screen.surface, tank_color, track_
    positions)

    # hull uses a rectangle which uses top right coords and
      dimensions
    hull_rect = Rect((xpos+15,ypos-20),(30,10))
    # Rectangle for tank body "hull" (pygame zero)
    screen.draw.filled_rect(hull_rect, tank_color)

    # Despite being an ellipse pygame requires this as a rect
    turret_rect = Rect((xpos+20,ypos-25),(20,10))
    # Ellipse for turret (pygame not pygame zero)
    pygame.draw.ellipse(screen.surface, tank_color, turret_rect)

    # Gun position involves more complex calculations so in a
      separate function
    gun_positions = calc_gun_positions (left_right, tank_start_
    pos, gun_angle)
```

```
    # Polygon for gun barrel (pygame not pygame zero)
    pygame.draw.polygon(screen.surface, tank_color,
    gun_positions)

# Calculate the polygon positions for the gun barrel
def calc_gun_positions (left_right, tank_start_pos, gun_angle):
    (xpos, ypos) = tank_start_pos
    # Set the start of the gun (top of barrel at point it joins
      the tank)
    if (left_right == "right"):
        gun_start_pos_top = (xpos+20, ypos-20)
    else:
        gun_start_pos_top = (xpos+40, ypos-20)
    # Convert angle to radians (for right subtract from 180 deg
      first)
    relative_angle = gun_angle
    if (left_right == "right"):
        relative_angle = 180 - gun_angle
    angle_rads = math.radians(relative_angle)
    # Create vector based on the direction of the barrel
    # Y direction *-1 (due to reverse y of screen)
    gun_vector = (math.cos(angle_rads), math.sin(angle_rads) * -1)

    # Determine position bottom of barrel
    # Create temporary vector 90deg to existing vector
    if (left_right == "right"):
        temp_angle_rads = math.radians(relative_angle - 90)
    else:
        temp_angle_rads = math.radians(relative_angle + 90)
    temp_vector =  (math.cos(temp_angle_rads), math.sin(temp_
    angle_rads) * -1)
```

```
    # Add constants for gun size
    GUN_LENGTH = 20
    GUN_DIAMETER = 3
    gun_start_pos_bottom = (gun_start_pos_top[0] + temp_
    vector[0] *
        GUN_DIAMETER, gun_start_pos_top[1] + temp_vector[1] *
        GUN_DIAMETER)

    # Calculate barrel positions based on vector from start
    position
    gun_positions = [
        gun_start_pos_bottom,
        gun_start_pos_top,
        (gun_start_pos_top[0] + gun_vector[0] * GUN_LENGTH,
            gun_start_pos_top[1] + gun_vector[1] * GUN_LENGTH),
        (gun_start_pos_bottom[0] + gun_vector[0] * GUN_LENGTH,
            gun_start_pos_bottom[1] + gun_vector[1] * GUN_LENGTH),
    ]

    return gun_positions
```

The program first imports some modules. One is the math module and the other is pygame. The reason for needing to import pygame is that while the game is designed for Pygame Zero, there are some features that are not currently available in Pygame Zero. Importing pygame enables the code to make use of functionality in the pygame module.

Next there are some global variables for the position of the tank and the angle for the gun. These refer to the left tank; in the final game, there will be two tanks and variable names are consistent to how they will be as the game is developed.

The draw function is a single entry that draws the tank by calling the `draw_tank` function. There is no update function as that is not needed at this point.

The task of drawing the tank goes to the `draw_tank` function. The first argument to the function is a word "left" or "right". This is not used in this code as currently it only creates the left tank, but it's often better to include any future arguments where it is known they will be needed later. The other arguments represent the position of the tank and the angle that the gun is pointing at.

The `draw_tank` function first defines the shape which represents the tank tracks. This is created as a polygon. A polygon can be any closed shape with at least three sides which makes it ideal for irregular shapes.

```
track_positions = [
    (xpos+5, ypos-5),
    (xpos+10, ypos-10),
    (xpos+50, ypos-10),
    (xpos+55, ypos-5),
    (xpos+50, ypos),
    (xpos+10, ypos)
]
pygame.draw.polygon(screen.surface, tank_color, track_
positions)
```

The `track_positions` list is created with all the vertices that represent the shape (each of the corners). Pygame Zero does not currently include the code to create polygons. To overcome this limitation, the Pygame method is used instead. Instead of beginning `screen.draw` as is used in Pygame Zero, the method is `pygame.draw.polygon` and the surface to draw on is passed as the first argument using `screen.surface`.

The next shape is a rectangle which can be drawn directly from Pygame Zero.

```
hull_rect = Rect((xpos+15,ypos-20),(30,10))
screen.draw.filled_rect(hull_rect, tank_color)
```

The `hull_rect` is Rect object, which has a tuple to represent the starting position (left, top) and a tuple to represent the size of the rectangle in pixels (width, height). That is then passed along with the color to `screen.draw.filled_rect`.

The turret is created as an ellipse. Pygame Zero does not currently support an ellipse (only having a circle), so this also needs to be created using Pygame. The ellipse is defined as a rectangle (Rect object), which contains the ellipse.

```
turret_rect = Rect((xpos+20,ypos-25),(20,10))
pygame.draw.ellipse(screen.surface, tank_color, turret_rect)
```

The last item in the draw function is to draw the gun barrel. This is a rectangle which is rotated to reflect the selected angle. As this is drawn at an angle, it is created as a polygon. The math in determining the positions of the vertices is quite involved so it is broken out into a separate function `calc_gun_positions`. The gun is shown in Figure 7-2 showing how the gun is positioned on the tank and the gun angle.

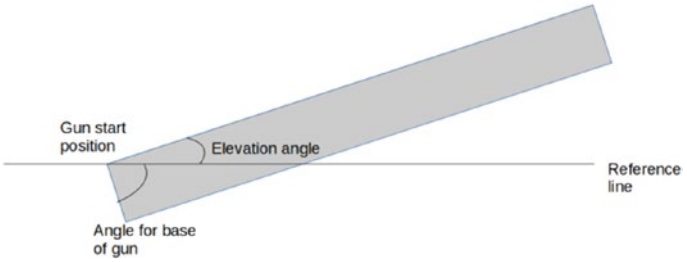


Figure 7-2. *Tank shape created using vector shapes*

The `calc_gun_positions` function has been written to support the tank being on the left of the screen (with the gun pointing to the right) or on the right of the screen (with the gun pointing to the left). This is done by first setting the appropriate start position for the top of the barrel where it overlaps the hull of the tank. The `gun_angle` is the number of degrees from the reference line shown in Figure 7-2. If the tank is on the right, then the gun angle is converted to a relative angle by subtracting 180 degrees.

The angle is then converted to radians as that is what the math module uses for the trigonometric functions. The `gun_vector` is then created based on the cosine for the change in the x axis and the sine for the change in the y axis. That vector gives the relative x and y changes and can be multiplied by the length of the gun to calculate the position of the vertex at the top of the gun. A similar technique is used to find the bottom position, which is at an angle of 90 degrees (minus or plus depending upon whether it is right or left) compared to the gun vector. Finally, a list is created called `gun_positions`, which is returned to the draw function to create the polygon.

Creating a Dynamic Landscape

In the previous code, the tank is just positioned in a stationary position hovering in the air. This next part will create the landscape for the tanks to stand on. Rather than create a static landscape which is the same each time the game is played, a dynamic landscape will be created. This will show how a dynamic landscape can be generated using random numbers. An example landscape is shown in Figure 7-3.

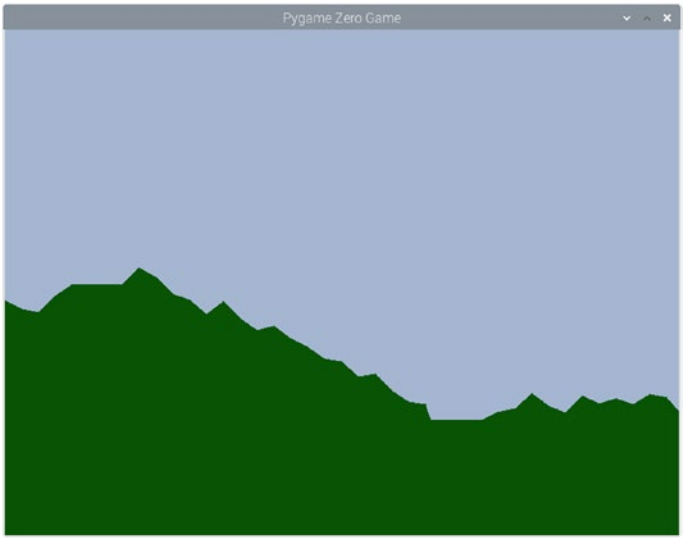


Figure 7-3. *Dynamic landscape for the tank game*

The landscape will be generated as a polygon. You may be thinking that you can just use a random number value to determine the value of the y axis. It is not quite that simple as the random number would result in sharp differences between each point causing the landscape to be too rugged and unrealistic. Instead the landscape is created by calculating a random value as a difference from the previous position. This gives a more gradual change. I've also created a flat area on both the left and right which is where the tanks will be positioned. The code for this is shown in Listing 7-2. The code is included in the source code as tanktrajectory.py.

Listing 7-2. Code to generate a random landscape for the tank game

```
import random
import pygame

WIDTH=800
HEIGHT=600
```

```
SKY_COLOR = (165, 182, 209)
GROUND_COLOR = (9,84,5)

# How big a chunk to split up x axis
LAND_CHUNK_SIZE = 20
# Max that land can go up or down within chunk size
LAND_MAX_CHG = 20
# Max height of ground
LAND_MIN_Y = 200

# Position of the two tanks - set to zero, update before use
left_tank_position = (0,0)
right_tank_position = (0,0)

def draw():
    screen.fill(SKY_COLOR)
    pygame.draw.polygon(screen.surface, GROUND_COLOR, land_
    positions)

# Setup game - allows create new game
def setup():
    global left_tank_position, right_tank_position, land_
    positions
    # Setup landscape (these positions represent left side of
      platform)
    # Choose a random position
    # The complete x,y co-ordinates will be saved in a
    # tuple in left_tank_rect and right_tank_rect
    left_tank_x_position = random.randint (10,300)
    right_tank_x_position = random.randint (500,750)

    # Sub divide screen into chunks for the landscape
    # store as list of x positions (0 is first position)
    current_land_x = 0
```

```
current_land_y = random.randint (300,400)
land_positions = [(current_land_x,current_land_y)]
while (current_land_x < WIDTH):
    if (current_land_x == left_tank_x_position):
        # handle tank platform
        left_tank_position = (current_land_x, current_
        land_y)
        # Create level ground for the tank to sit on
        # Add another 50 pixels further along at same y
          position
        current_land_x += 60
        land_positions.append((current_land_x, current_
        land_y))
        continue
    elif (current_land_x == right_tank_x_position):
        # handle tank platform
        right_tank_position = (current_land_x, current_
        land_y)
        # Create level ground for the tank to sit on
        # Add another 50 pixels further along at same y
          position
        current_land_x += 60
        land_positions.append((current_land_x, current_
        land_y))
        continue
    # Checks to see if next position will be where the
      tanks are
    if (current_land_x < left_tank_x_position and current_
    land_x +
        LAND_CHUNK_SIZE >= left_tank_x_position):
        # set x position to tank position
        current_land_x = left_tank_x_position
```

```
        elif (current_land_x < right_tank_x_position and
        current_land_x +
            LAND_CHUNK_SIZE >= right_tank_x_position):
            # set x position to tank position
            current_land_x = right_tank_x_position
        elif (current_land_x + LAND_CHUNK_SIZE > WIDTH):
            current_land_x = WIDTH
        else:
            current_land_x += LAND_CHUNK_SIZE
        # Set the y height
        current_land_y += random.randint(0-LAND_MAX_CHG,
        LAND_MAX_CHG)
        # check not too high or too low
        # Note the reverse logic as high y is bottom of screen
        if (current_land_y > HEIGHT):   # Bottom of screen
            current_land_y = HEIGHT
        if (current_land_y < LAND_MIN_Y):
            current_land_y = LAND_MIN_Y
        # Add to list
        land_positions.append((current_land_x, current_land_y))
    # Add end corners
    land_positions.append((WIDTH,HEIGHT))
    land_positions.append((0,HEIGHT))

# Setup the game (at end so that it can see the other
  functions)
setup()
```

After setting up some constants and variables, the setup function is called. The instruction to call `setup` is at the bottom of the file. This is because in Python the function must be defined before it's called, so by placing it at the bottom of the file, all earlier functions are already loaded.

Before creating the ground, the tank positions need to be calculated. This is so that the code can ensure that the tank is mounted on a level section of ground. The x position of the tanks is set based on a random integer; the y position will be added later once the ground is calculated. The background is then split into chunks of a fixed size. If a tank will be on the next section, then the chunk is ended at the position so that the level section can be created. The next chunk is then created with the y axis left unchanged.

If the current section won't have a tank on it, then it will be changed by a random amount. All these positions are added to a list which is then used by the draw function to draw the polygon.

Calculating the Trajectory

When the shell is fired from the gun, it does not follow a straight line. This is because of several factors, the main influence being gravity. Ignoring the other factors, then the force of gravity pulling it toward earth would result in the path of the shell forming a parabola as it first gains height and then starts to fall back toward earth.

In the real world, the path would be distorted because of air resistance and any wind resistance that it encounters. To keep it fairly simple, this program will just consider gravity. This will be handled by a function `update_shell_position` and a `draw_shell` function. To illustrate this, I have created a program tanktrajectory.py which will display the entire path for a certain set of values. The path is shown in Figure 7-4 with the colors modified to improve the contrast.

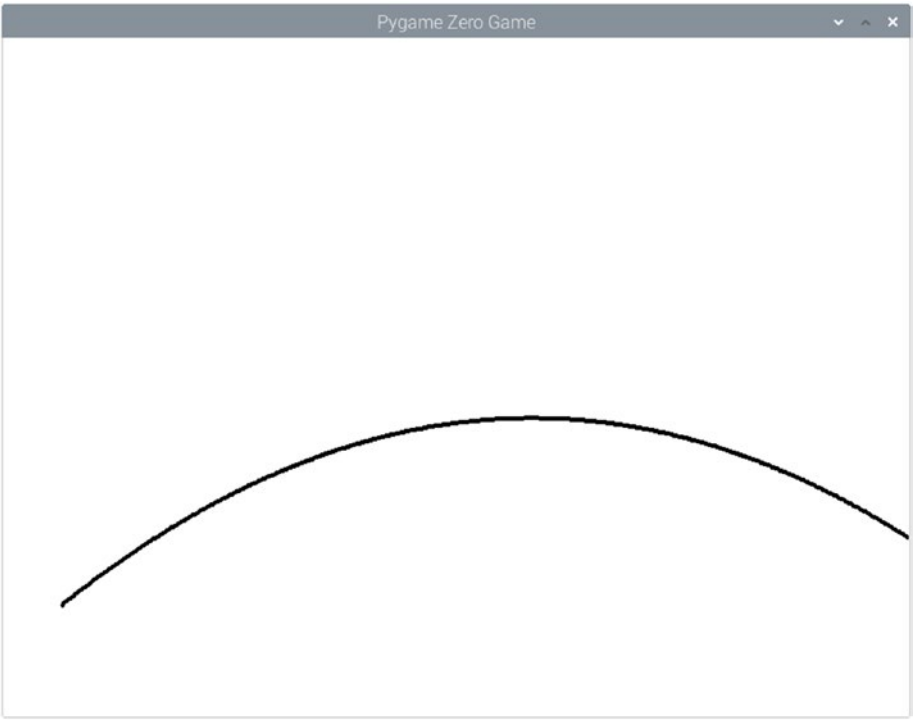


Figure 7-4. *Example trajectory of a tank shell being fired*

The code to demonstrate this is shown in Listing 7-3.

Listing 7-3. Code to demonstrate trajectory for a tank shell being fired

```
import math
import pygame

WIDTH=800
HEIGHT=600

SKY_COLOR = (165, 182, 209)
SHELL_COLOR = (255,255,255)
shell_start_position = (50,500)
```

```
left_gun_angle = 50
left_gun_power = 60
-
shell_positions = []

def draw_shell (position):
    (xpos, ypos) = position
    # Create rectangle of the shell
    shell_rect = Rect((xpos,ypos),(5,5))
    pygame.draw.ellipse(screen.surface, SHELL_COLOR,
    shell_rect)

def draw():
    screen.fill(SKY_COLOR)
    for this_position in shell_positions:
        draw_shell(this_position)

def update_shell_position (left_right):
    global shell_power, shell_angle, shell_start_position,
    shell_current_position, shell_time

    init_velocity_y = shell_power * math.sin(shell_angle)

    # Direction - multiply by -1 for left to right
    if (left_right == 'left'):
        init_velocity_x = shell_power * math.cos(shell_angle)
    else:
        init_velocity_x = shell_power * math.cos(math.pi -
        shell_angle)
    # Gravity constant is 9.8 m/s^2 but this is in terms of
      screen so instead use a suitable value
    GRAVITY_CONSTANT = 0.004
    # Constant to give a sensible distance on x axis
    DISTANCE_CONSTANT = 1.5
```

```
    # time is calculated in update cycles
    shell_x = shell_start_position[0] + init_velocity_x *
    shell_time * DISTANCE_CONSTANT
    shell_y = shell_start_position[1] + -1 * ((init_velocity_y *
    shell_time) -
        (0.5 * GRAVITY_CONSTANT * shell_time * shell_time))

    shell_current_position = (shell_x, shell_y)
    shell_time += 1

def setup_trajectory():
    global shell_positions, shell_current_position, shell_
    power, shell_angle, shell_time

    shell_current_position = shell_start_position

    shell_angle = math.radians (left_gun_angle)
    shell_power = left_gun_power / 40
    shell_time = 0

    while (shell_current_position[0] < WIDTH and shell_current_
    position[1] < HEIGHT):
        update_shell_position("left")
        shell_positions.append(shell_current_position)
setup_trajectory()
```

The `setup_trajectory` function is used to demonstrate the trajectory and won't be included in the game. It sets the angle and then creates a while loop which calculates all the positions that the shell will go through before hitting the ground or going off the right-hand side of the screen.

The `update_shell_position` function starts with calculating the initial velocity in the x and y directions. This is based on the power and angle of the gun.

There then needs to be two constants: a value that represents GRAVITY_CONSTANT (amount of force pulling down toward earth) and DISTANCE_CONSTANT which influences how far the shell travels in the x direction on each step. The value for gravity is 9.8 m/s2, but that is assuming real distance measured in meters. In the case of a computer screen, we have a virtual distance measured in pixels. The value used is created using trial and error to get a value that looks realistic and gives a suitable curve. The same trial and error method is used for the DISTANCE_CONSTANT. These values are then included in the following algorithms to determine the position of the shell at each time interval.

```
shell_x = shell_start_position[0] + init_velocity_x *
shell_time * DISTANCE_CONSTANT
shell_y = shell_start_position[1] + -1 * ((init_velocity_y *
shell_time) -
    (0.5 * GRAVITY_CONSTANT * shell_time * shell_time))
```

This doesn't include any factor for air resistance, wind resistance, or any of the other forces acting upon the shell except for gravity.

This demonstration program shows all the shell positions simultaneously, but in the game only a single shell will be drawn at a time, which will move slowly across the screen.

Detecting a Collision

In the earlier game, the collisions were based on the Rect collide feature. Although a useful technique, it does not have the accuracy needed for this game. An alternative technique is to detect when the shell collides with a tank or the ground by looking for the color of the pixel to see if it matches with the color of the tank or ground. For this to work, the color of the ground and of each of the tanks needs to be unique. Listing 7-4 shows the function that will be used to detect the collision.

Listing 7-4. Function to detect collision with tank or ground

```
def detect_hit (left_right):
    global shell_current_position
    (shell_x, shell_y) = shell_current_position
    # Add offset (3 pixels)
    # offset left/right depending upon direction of fire
    if (left_right == "left"):
        shell_x += 3
    else:
        shell_x -= 3
    shell_y += 3
    offset_position = (math.floor(shell_x), math.
    floor(shell_y))

    # Check whether it's off the screen
    # temporary if just y axis, permanent if x
    if (shell_x > WIDTH or shell_x <= 0 or shell_y >= HEIGHT):
        return 10
    if (shell_y < 1):
        return 1

    # Get color at position
    color_pixel = screen.surface.get_at(offset_position)
    if (color_pixel == GROUND_COLOR):
        return 11
    if (left_right == 'left' and color_pixel == TANK_COLOR_P2):
        return 20
    if (left_right == 'right' and color_pixel == TANK_COLOR_P1):
        return 20

    return 0
```

This code creates an offset just in front of the shell, so as not to look at its own color. It then checks to see if that position is off the screen. If it has gone above the top of the screen, then that is just a temporary situation, so it returns a different value to if it goes off the right or left side of the screen. The code then uses the following line to read the value of the pixel at the offset position:

```
color_pixel = screen.surface.get_at(offset_position)
```

This returns the value of the pixel at the offset position. If that value is a match with the color of a tank or the ground, then it returns an appropriate value.

In this function the values returned are just values which have been chosen to represent the different conditions. If you are writing code that will be reused in other programs, then it is usually a good idea to create a constant to make it easier to see what that value means. For example, when looking at the status of the mouse in Chapter 6, a test was made to see if the value of the button was equal to `mouse.LEFT`. The value of `mouse.LEFT` is just a number, which happens to be 1. It is generally easier to remember `mouse.LEFT` rather than having to remember the number that is generated for each of the different buttons. As this is only used for this particular function, the real value is returned but comments have been included in the code to explain what those values mean.

Complete Game Code

There is quite a bit of additional code still, but most of that includes techniques that have already been demonstrated in earlier chapters.

As with most programs, the state of the game needs to be tracked to know which player is currently active or to display the appropriate message. This is done by setting appropriate text in the variable `game_state`.

The different states are listed in comments at the start of the program; they are

- "start" – Timed delay before start
- "player1" – Waiting for player to set position
- "player1fire" – Player 1 fired
- "player2" – Player 2 set position
- "player2fire" – Player 2 fired
- "game_over_1" – Show that player 1 won
- "game_over_2" – Show that player 2 won

These states have appropriate codes in the update or draw functions to make sure that the game gives the correct prompts or handles the input appropriately.

The `player_keyboard` function is called from the update function to check to see if any keys are pressed. If the up or down buttons are pressed, then the gun elevation angle is adjusted; if the left or right buttons are pressed, then the power is adjusted (as a percentage of maximum power), and if space is pressed, then the shell is fired. There is an additional test to see if the left-shift key is pressed, which is another option instead of the space to fire the shell. This is included so that the game can work with the Picade or other Raspberry Pi-based arcade machines which map that key to a physical button.

There is a setup function used for all the code that needs to be run when the game is first run. This creates the landscape as well as setting values for many of the variables that will be needed later. There is also additional code to display messages to the user. The code for the complete game is shown in Listing 7-5.

Listing 7-5. Complete code for Tank Game Zero

```
import math
import random
import pygame

WIDTH=800
HEIGHT=600

# States are:
# start - timed delay before start
# player1 - waiting for player to set position
# player1fire - player 1 fired
# player2 - player 2 set position
# player2fire - player 2 fired
# game_over_1 / game_over_2 - show who won 1 = player 1 won etc.
game_state = "player1"

# Color constants
SKY_COLOR = (165, 182, 209)
GROUND_COLOR = (9,84,5)
# Different tank colors for player 1 and player 2
# These colors must be unique as well as the GROUND_COLOR
TANK_COLOR_P1 = (216, 216, 153)
TANK_COLOR_P2 = (219, 163, 82)
SHELL_COLOR = (255,255,255)
TEXT_COLOR = (255,255,255)

# How big a chunk to split up x axis
LAND_CHUNK_SIZE = 20
# Max that land can go up or down within chunk size
LAND_MAX_CHG = 20
# Max height of ground
LAND_MIN_Y = 200
```

```
# Timer used to create delays before action (prevent accidental
  button press)
game_timer = 0

# Angle that the gun is pointing (degrees relative to
  horizontal)
left_gun_angle = 20
right_gun_angle = 50
# Amount of power to fire with - is divided by 40 to give scale
  10 to 100
left_gun_power = 25
right_gun_power = 25
# These are shared between left and right as we only fire one
  shell at a time
shell_power = 1
shell_angle = 0
shell_time = 0

# Position of shell when fired (create as a global - but update
  before use)
shell_start_position = (0,0)
shell_current_position = (0,0)

# Position of the two tanks - set to zero, update before use
left_tank_position = (0,0)
right_tank_position = (0,0)

# Draws tank (including gun - which depends upon direction and
  aim)
# left_right can be "left" or "right" to depict which position
  the tank is in
# tank_start_pos requires x, y co-ordinates as a tuple
# angle is relative to horizontal - in degrees
```

```
def draw_tank (left_right, tank_start_pos, gun_angle):
    (xpos, ypos) = tank_start_pos

    # Set appropriate color for the tank
    if (left_right == "left"):
        tank_color = TANK_COLOR_P1
    else:
        tank_color = TANK_COLOR_P2

    # The shape of the tank track is a polygon
    # (uses list of tuples for the x and y co-ords)
    track_positions = [
        (xpos+5, ypos-5),
        (xpos+10, ypos-10),
        (xpos+50, ypos-10),
        (xpos+55, ypos-5),
        (xpos+50, ypos),
        (xpos+10, ypos)
    ]
    # Polygon for tracks (pygame not pygame zero)
    pygame.draw.polygon(screen.surface, tank_color, track_
    positions)

    # hull uses a rectangle which uses top right co-ords and
      dimensions
    hull_rect = Rect((xpos+15,ypos-20),(30,10))
    # Rectangle for tank body "hull" (pygame zero)
    screen.draw.filled_rect(hull_rect, tank_color)

    # Despite being an ellipse pygame requires this as a rect
    turret_rect = Rect((xpos+20,ypos-25),(20,10))
    # Ellipse for turret (pygame not pygame zero)
    pygame.draw.ellipse(screen.surface, tank_color, turret_rect)
```

```
    # Gun position involves more complex calculations so in a
      separate function
    gun_positions = calc_gun_positions (left_right, tank_start_
    pos, gun_angle)
    # Polygon for gun barrel (pygame not pygame zero)
    pygame.draw.polygon(screen.surface, tank_color, gun_
    positions)

def draw_shell (position):
    (xpos, ypos) = position
    # Create rectangle of the shell
    shell_rect = Rect((xpos,ypos),(5,5))
    pygame.draw.ellipse(screen.surface, SHELL_COLOR, shell_rect)

# Calculate the polygon positions for the gun barrel
def calc_gun_positions (left_right, tank_start_pos, gun_angle):
    (xpos, ypos) = tank_start_pos
    # Set the start of the gun (top of barrel at point it joins
      the tank)
    if (left_right == "right"):
        gun_start_pos_top = (xpos+20, ypos-20)
    else:
        gun_start_pos_top = (xpos+40, ypos-20)

    # Convert angle to radians (for right subtract from 180 deg
      first)
    relative_angle = gun_angle
    if (left_right == "right"):
        relative_angle = 180 - gun_angle
    angle_rads = relative_angle * (math.pi / 180)
    # Create vector based on the direction of the barrel
    # Y direction *-1 (due to reverse y of screen)
    gun_vector = (math.cos(angle_rads), math.sin(angle_rads) * -1)
```

```
    # Determine position bottom of barrel
    # Create temporary vector 90deg to existing vector
    if (left_right == "right"):
        temp_angle_rads = math.radians(relative_angle - 90)
    else:
        temp_angle_rads = math.radians(relative_angle + 90)
    temp_vector =  (math.cos(temp_angle_rads), math.sin(temp_
    angle_rads) * -1)

    # Add constants for gun size
    GUN_LENGTH = 20
    GUN_DIAMETER = 3
    gun_start_pos_bottom = (gun_start_pos_top[0] + temp_
    vector[0] * GUN_DIAMETER, gun_start_pos_top[1] + temp_
    vector[1] * GUN_DIAMETER)

    # Calculate barrel positions based on vector from start
      position
    gun_positions = [
        gun_start_pos_bottom,
        gun_start_pos_top,
        (gun_start_pos_top[0] + gun_vector[0] * GUN_LENGTH,
        gun_start_pos_top[1] + gun_vector[1] * GUN_LENGTH),
        (gun_start_pos_bottom[0] + gun_vector[0] * GUN_LENGTH,
        gun_start_pos_bottom[1] + gun_vector[1] * GUN_LENGTH),
    ]

    return gun_positions

def draw():
    global game_state, left_tank_position, right_tank_position,
    left_gun_angle, right_gun_angle, shell_start_position
    screen.fill(SKY_COLOR)
```

```
    pygame.draw.polygon(screen.surface, GROUND_COLOR, land_
    positions)
    draw_tank ("left", left_tank_position, left_gun_angle)
    draw_tank ("right", right_tank_position, right_gun_angle)
    if (game_state == "player1" or game_state == "player1fire"):
        screen.draw.text("Player 1\nPower "+str(left_gun_
         power)+"%", fontsize=30, topleft=(50,50), color=(TEXT_
         COLOR))
    if (game_state == "player2" or game_state == "player2fire"):
        screen.draw.text("Player 2\nPower "+str(right_gun_
        power)+"%", fontsize=30, topright=(WIDTH-50,50),
        color=(TEXT_COLOR))
    if (game_state == "player1fire" or game_state ==
    "player2fire"):
        draw_shell(shell_current_position)
    if (game_state == "game_over_1"):
        screen.draw.text("Game Over\nPlayer 1 wins!",
        fontsize=60, center=(WIDTH/2,200), color=(TEXT_COLOR))
    if (game_state == "game_over_2"):
        screen.draw.text("Game Over\nPlayer 2 wins!",
        fontsize=60, center=(WIDTH/2,200), color=(TEXT_COLOR))
def update():
    global game_state, left_gun_angle, left_tank_position,
    shell_start_position, shell_current_position, shell_angle,
    shell_time, left_gun_power, right_gun_power, shell_power,
    game_timer
    # Delayed start (prevent accidental firing by holding start
      button down)
    if (game_state == 'start'):
        game_timer += 1
```

```
    if (game_timer == 30):
        game_timer = 0
        game_state = 'player1'
# Only read keyboard in certain states
if (game_state == 'player1'):
    player1_fired = player_keyboard("left")
    if (player1_fired == True):
        # Set shell position to end of gun
        # Use gun_positions so we can get start position
        gun_positions = calc_gun_positions ("left", left_
        tank_position, left_gun_angle)
        shell_start_position = gun_positions[3]
        shell_current_position = gun_positions[3]
        game_state = 'player1fire'
        shell_angle = math.radians (left_gun_angle)
        shell_power = left_gun_power / 40
        shell_time = 0
if (game_state == 'player1fire'):
    update_shell_position ("left")
    # shell value is whether the shell is inflight, hit or
      missed
    shell_value = detect_hit("left")
    # shell_value 20 is if other tank hit
    if (shell_value >= 20):
        game_state = 'game_over_1'
    # 10 is offscreen and 11 is hit ground, both indicate
      missed
    elif (shell_value >= 10):
        game_state = 'player2'
if (game_state == 'player2'):
    player2_fired = player_keyboard("right")
```

```
        if (player2_fired == True):
            # Set shell position to end of gun
            # Use gun_positions so we can get start position
            gun_positions = calc_gun_positions ("right", right_
            tank_position, right_gun_angle)
            shell_start_position = gun_positions[3]
            shell_current_position = gun_positions[3]
            game_state = 'player2fire'
            shell_angle = math.radians (right_gun_angle)
            shell_power = right_gun_power / 40
            shell_time = 0
    if (game_state == 'player2fire'):
        update_shell_position ("right")
        # shell value is whether the shell is inflight, hit or
          missed
        shell_value = detect_hit("right")
        # shell_value 20 is if other tank hit
        if (shell_value >= 20):
            game_state = 'game_over_2'
        # 10 is offscreen and 11 is hit ground, both indicate
          missed
        elif (shell_value >= 10):
            game_state = 'player1'
    if (game_state == 'game_over_1' or game_state == 'game_
    over_2'):
        # Allow space key or left-shift (picade) to continue
        if (keyboard.space or keyboard.lshift):
            game_state = 'start'
            # Reset position of tanks and terrain
            setup()
```

```
def update_shell_position (left_right):
    global shell_power, shell_angle, shell_start_position,
    shell_current_position, shell_time

    init_velocity_y = shell_power * math.sin(shell_angle)

    # Direction - multiply by -1 for left to right
    if (left_right == 'left'):
        init_velocity_x = shell_power * math.cos(shell_angle)
    else:
        init_velocity_x = shell_power * math.cos(math.pi -
        shell_angle)

    # Gravity constant is 9.8 m/s^2 but this is in terms of
      screen so instead use a sensible constant
    GRAVITY_CONSTANT = 0.004
    # Constant to give a sensible distance on x axis
    DISTANCE_CONSTANT = 1.5
    # Wind is not included in this version, to implement then
    decreasing wind value is when the wind is against the fire
    direction
    # wind > 1 is where wind is against the direction of fire.
    Wind must never be 0 or negative (which would make it
    impossible to fire forwards)
    wind_value = 1

    # time is calculated in update cycles
    shell_x = shell_start_position[0] + init_velocity_x *
    shell_time * DISTANCE_CONSTANT
    shell_y = shell_start_position[1] + -1 * ((init_velocity_y
    * shell_time) - (0.5 * GRAVITY_CONSTANT * shell_time *
    shell_time * wind_value))
    shell_current_position = (shell_x, shell_y)

    shell_time += 1
```

```
# Detects if the shell has hit something.
# Simple detection looks at color of the screen at the position
# uses an offset to not detect the actual shell
# Return 0 for in-flight,
# 1 for offscreen temp (too high),
# 10 for offscreen permanent (too far),
# 11 for hit ground,
# 20 for hit other tank
def detect_hit (left_right):
    global shell_current_position
    (shell_x, shell_y) = shell_current_position
    # Add offset (3 pixels)
    # offset left/right depending upon direction of fire
    if (left_right == "left"):
        shell_x += 3
    else:
        shell_x -= 3
    shell_y += 3
    offset_position = (math.floor(shell_x), math.floor(shell_y))

    # Check whether it's off the screen
    # temporary if just y axis, permanent if x
    if (shell_x > WIDTH or shell_x <= 0 or shell_y >= HEIGHT):
        return 10
    if (shell_y < 1):
        return 1

    # Get color at position
    color_pixel = screen.surface.get_at(offset_position)
    if (color_pixel == GROUND_COLOR):
        return 11
    if (left_right == 'left' and color_pixel == TANK_COLOR_P2):
        return 20
```

```
    if (left_right == 'right' and color_pixel == TANK_COLOR_P1):
        return 20
    return 0

# Handles keyboard for players
# If player has hit fire key (space) then returns True
# Otherwise changes angle of gun if applicable and returns
  False
def player_keyboard(left_right):
    global shell_start_position, left_gun_angle, right_gun_
    angle, left_gun_power, right_gun_power

    # get current angle
    if (left_right == 'left'):
        this_gun_angle = left_gun_angle
        this_gun_power = left_gun_power
    else:
        this_gun_angle = right_gun_angle
        this_gun_power = right_gun_power

    # Allow space key or left-shift (picade) to fire
    if (keyboard.space or keyboard.lshift):
        return True
    # Up moves firing angle upwards, down moves it down
    if (keyboard.up):
        this_gun_angle += 1
        if (this_gun_angle > 85):
            this_gun_angle = 85
    if (keyboard.down):
        this_gun_angle -= 1
        if (this_gun_angle < 0):
            this_gun_angle = 0
```

```
    # left reduces power, right increases power
    if (keyboard.right):
        this_gun_power += 1
        if (this_gun_power > 100):
            this_gun_power = 100
    if (keyboard.left):
        this_gun_power -= 1
        if (this_gun_power < 10):
            this_gun_power = 10

    # Update the appropriate global (left / right)
    if (left_right == 'left'):
        left_gun_angle = this_gun_angle
        left_gun_power = this_gun_power
    else:
        right_gun_angle = this_gun_angle
        right_gun_power = this_gun_power

    return False

# Setup game - allows create new game
def setup():
    global left_tank_position, right_tank_position, land_
    positions
    # Setup landscape (these positions represent left side of
      platform)
    # Choose a random position
    # The complete x,y co-ordinates will be saved in a tuple in
      left_tank_rect and right_tank_rect
    left_tank_x_position = random.randint (10,300)
    right_tank_x_position = random.randint (500,750)

    # Sub divide screen into chunks for the landscape
```

```
# store as list of x positions (0 is first position)
current_land_x = 0
current_land_y = random.randint (300,400)
land_positions = [(current_land_x,current_land_y)]
while (current_land_x < WIDTH):
    if (current_land_x == left_tank_x_position):
        # handle tank platform
        left_tank_position = (current_land_x, current_
        land_y)
        # Add another 50 pixels further along at same y
          position (level ground for tank to sit on)
        current_land_x += 60
        land_positions.append((current_land_x, current_
        land_y))
        continue
    elif (current_land_x == right_tank_x_position):
        # handle tank platform
        right_tank_position = (current_land_x, current_
        land_y)
        # Add another 50 pixels further along at same y
          position (level ground for tank to sit on)
        current_land_x += 60
        land_positions.append((current_land_x, current_
        land_y))
        continue
    # Checks to see if next position will be where the
      tanks are
    if (current_land_x < left_tank_x_position and current_
    land_x + LAND_CHUNK_SIZE >= left_tank_x_position):
        # set x position to tank position
        current_land_x = left_tank_x_position
```

```
        elif (current_land_x < right_tank_x_position and
        current_land_x + LAND_CHUNK_SIZE >= right_tank_x_
        position):
            # set x position to tank position
            current_land_x = right_tank_x_position
        elif (current_land_x + LAND_CHUNK_SIZE > WIDTH):
            current_land_x = WIDTH
        else:
            current_land_x += LAND_CHUNK_SIZE
        # Set the y height
        current_land_y += random.randint(0-LAND_MAX_CHG,LAND_
        MAX_CHG)
        # check not too high or too lower (note the reverse
          logic as high y is bottom of screen)
        if (current_land_y > HEIGHT):   # Bottom of screen
            current_land_y = HEIGHT
        if (current_land_y < LAND_MIN_Y):
            current_land_y = LAND_MIN_Y
        # Add to list
        land_positions.append((current_land_x, current_land_y))
    # Add end corners
    land_positions.append((WIDTH,HEIGHT))
    land_positions.append((0,HEIGHT))

# Setup the game (at end so that it can see the other
  functions)
setup()
```

Rather than typing all the code yourself, you will find a copy with the book source code named tankgame.py.

You may notice that there is some code that is repeated. This is because there is some code for when player 1 is playing and very similar code for player 2. This is something that is generally best avoided; not only does it mean more typing, it also makes it more difficult to remember to update the code for both tanks and to debug if things go wrong. This is something that could be refactored in a future version and is something that object-oriented programming can help with, which is covered in Chapter 9.

Improving the Game

This game has the starting of making an enjoyable game. In fact, there are several commercial games that are based on the concept of the artillery game. Many use a tank, but others replace the tanks with other objects, such as a catapult against a castle wall or worms with a variety of different weapons. There is even a game which uses a catapult to fire different birds at pigs that are trying to steal their eggs.

So now you've learned the concepts involved, can you think of ways to make the game more enjoyable? Here are some of my thoughts:

- Have multiple lives or different amount of damage level required (health).
- Change the order of which player goes first so that player 1 doesn't always have the advantage.
- Add wind resistance with different amounts of wind.
- Add sound effects or background music.
- Show an explosion when the shell hits.
- Add a computer player option.
- Have different shapes of tanks or different colors.

- Different tanks could have different amount of power vs. health to give a choice between more powerful gun and better resistance against hits.
- Earn points to spend on tank upgrades.
- Allow the tank to move.
- Multiple tanks.
- Different weapons.
- Replace the tanks with a different object or creature.

You could add these features to the existing code or use the concepts you have learned to create another game.

Summary

This chapter has covered various techniques including drawing vector images, creating dynamic landscapes, calculating a trajectory, and other steps involved in creating a game. The tank game will be used again during the next chapter which will add some sound effects and background music.

CHAPTER 8

Sound

Adding sound to a game will add an additional dimension and can help bring the game come to life. This can be achieved by adding special effect sounds or adding background music to set the mood. You may also use the sound as a key component in the game.

As well as looking at how music can be added to a game through Pygame Zero, this chapter will also look at ways of creating the sound effects or music and some of the tools that can be used to process the sounds.

This chapter starts with looking at how you can create your own sounds and music. If you are just interested in using sound effects or music that have been created by someone else, you can skip to later in the chapter where the sounds are added to a Pygame Zero game.

Recording Sound Effects

For realistic sound effects, they are often created by recording real sounds. It may not however be possible to record the effect you are creating in the game. If you don't happen to own a challenger tank, then you may need to look at something that sounds like a tank rather than recording a real tank. If you are creating a futuristic sci-fi game, then you may need to look at sounds being computer generated.

Even if you can record the exact effect that you want, that may not sound quite right for a game. One of the things I looked at was how you could create the sound of a steam train. I have several preservation

© Stewart Watkiss 2020
S. Watkiss, *Beginning Game Programming with Pygame Zero*,
https://doi.org/10.1007/978-1-4842-5650-3_8

railways within a reasonable distance, so I visited them to record the sounds. One problem is that there is a lot of additional background noise from people, pets, and other things around such as car traffic. Also, the sounds recorded while realistic did not match the sound that you may expect or that fit in with what is happening in the game. For example, when recording the sound of the train, the sound of the locomotive was accompanied by lots of different noises, such as the carriages clanging and the sound of the wheels squealing against the track. I found a better sound was achieved by recording the sound of the locomotive when it was uncoupled from the train rather than when it was pulling a train.

You probably won't be wanting to carry a Raspberry Pi, screen, and accessories around when you want to record sounds. In that case you can use a portable recorder, perhaps a mobile phone either using a video recorder or using an audio recording tool. Details are provided on how you can convert and edit suitable audio formats using Audacity if you have captured them with a mobile phone.

Creating Artificial Sound Effects

If you can't record the real sound effects, then it may be possible to create an equivalent sound using household items. Here are a few examples:

- The crunch of walking feet using a shoe in a tray of gravel.
- The clip-clop of horse hooves by tapping coconut shells together.
- Explosions based on fireworks. If local laws don't permit consumer fireworks, then you could record a professional display.
- Water sounds created in a bathtub.

I have used artificial sound effects in creating the sounds for the tank game. The sound of the tank firing is based on popping a balloon, with the time slowed down. The sound of the explosion was recorded at a public firework display.

You can also create sound effects synthetically using music creation tools such as Sonic Pi. It is possible to use different shaped waveforms and adding audio effects to create various sounds, particularly useful for sci-fi type effects.

There are some web sites with examples of how you can create artificial sound effects. Two examples are listed here, but there are others.

- EpicSound – `www.epicsound.com/sfx/`
- The Art of Foley – `www.marblehead.net/foley/specifics.html`

Recording Audio on the Raspberry Pi

The Raspberry Pi does not include an audio input. If you want to record sounds directly on a Raspberry Pi, then you will need an audio input device. The most common methods would be either a USB microphone (as shown in Figure 8-1) or a USB audio adapter with a microphone socket.

Figure 8-1. *Raspberry Pi with USB microphone*

Before recording sounds, you should test that audio is working on the Raspberry Pi by playing sounds through a TV or external speaker. The aplay command can be used using the following commands:

```
aplay /usr/share/sounds/alsa/Front_Left.wav
aplay /usr/share/sounds/alsa/Front_Right.wav
```

These commands test for stereo through the left and right speakers. If there is no sound, then the sound icon on the top right of the desktop provides a choice of Analog (headphone jack) or HDMI. Alternatively, it can be changed through the terminal configuration tool.

```
sudo raspi-config
```

Choose advanced options and then Audio which gives the option of using

- Auto
- Force 3.5mm (“headphone”) jack
- Force HDMI

Connecting a USB Microphone

After connecting the microphone, you should run the dmesg from the terminal to see details of the connected device. The dmesg tool will show messages from the kernel ring buffer logs.

```
dmesg
```

At the bottom, you should see an entry like the messages shown in Listing 8-1.

Listing 8-1. Partial output of dmesg showing USB microphone

```
[ 3407.526441] usb 1-1.3: new full-speed USB device number 4
using xhci_hcd
[ 3407.670531] usb 1-1.3: New USB device found, idVendor=0c76,
idProduct=1690, bcdDevice= 1.00
[ 3407.670539] usb 1-1.3: New USB device strings: Mfr=0,
Product=1, SerialNumber=0
[ 3407.670544] usb 1-1.3: Product: USB PnP Device(Echo-058)
[ 3407.677945] input: USB PnP Device(Echo-058) as /devices/
platform/scb/fd500000.pcie/pci0000:00/0000:00:00.0/0000:01:00.
0/usb1/1-1/1-1.3/1-1.3:1.2/0003:0C76:1690.0007/input/input15
[ 3407.746906] hid-generic 0003:0C76:1690.0007: input,hidraw3:
USB HID v1.00 Device [USB PnP Device(Echo-058)] on usb-
0000:01:00.0-1.3/input2
```

```
[ 3407.844707] usb 1-1.3: Warning! Unlikely big volume range
(=496), cval->res is probably wrong.
[ 3407.844724] usb 1-1.3: [50] FU [Mic Capture Volume] ch = 1,
val = 0/7936/16
[ 3407.847365] usbcore: registered new interface driver snd-
usb-audio
```

This example is using a Fifine Technology USB microphone. It uses the driver Echo-058.

You can also see the device by right-clicking the sound icon at the top right of the desktop as shown in Figure 8-2.

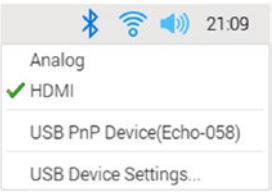


Figure 8-2. *Raspberry Pi sound settings with USB microphone*

Using arecord

Once the microphone is connected, then there are a few different tools that can be used to record sounds. For a simple command-line tool, arecord is included in the standard NOOBS image.

To use arecord, find the device by running arecord –l, which will give an output like that in Listing 8-2.

Listing 8-2. Output of arecord –l command

```
arecord -l
***** List of CAPTURE Hardware Devices ****
card 1: DeviceEcho058 [USB PnP Device(Echo-058)], device 0: USB
Audio [USB Audio]
```

```
Subdevices: 1/1
Subdevice #0: subdevice #0
```

The card number (in this case 1) and the device number (in this case 0) form the basis of the device reference which in this case is hw:1,0. The plughw plugin needs to be used; in this case, the device is plughw:1,0.

The following command will create a wav file, 16-bit little endian, with a maximum duration of 60 seconds, saved as a file audiorecord.wav:

```
arecord -D plughw:1,0 -t wav -f S16_LE -d 60 audiorecord.wav
```

An alternative to using the command line is the graphical application Audacity, which will be covered next.

Audacity

Audacity is a powerful tool which can be used for recording and editing audio. Here you will see how Audacity can be used for recording audio on the Raspberry Pi, converting audio formats, extracting audio from video files, and trimming audio files.

Audacity is not included by default on the Raspberry Pi but can be installed using

```
sudo apt install audacity
```

This will add an option to the "Sound & Video" menu. For other operating systems, you can download Audacity from www.audacityteam.org.

A screenshot of the program is shown in Figure 8-3.

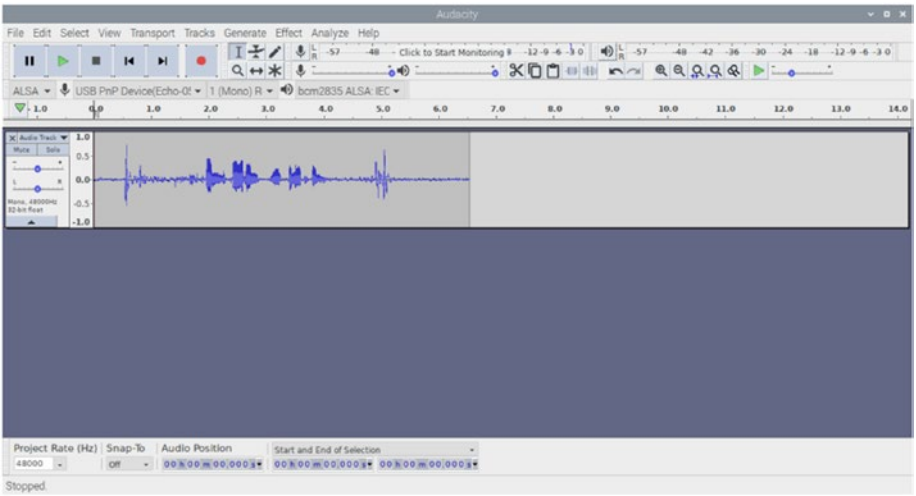


Figure 8-3. *Screenshot of Audacity audio editor*

Here are a few suggestions of things you may like to try which will help familiarize yourself with some of the features of Audacity.

Recording Sounds with Audacity

Audacity can record directly from a microphone. The microphone can be selected, recording started, and recording stopped using the graphical user interface.

- Launch Audacity and it will not show any sound waveforms.
- Ensure that the microphone is selected as the input device (shown alongside the microphone icon).
- Click the red record button and talk into the microphone or record nearby sounds.
- Stop recording.
- Export the audio as a suitable sound format (WAV and OGG are good formats for use in Pygame Zero).

Convert Audio Formats

Audacity can read from multiple different audio file formats and then convert them to another when you export them. This may be to convert from an MP3 file or an M4A file (often used on mobile phones) to a WAV or OGG file.

- Close any existing project.
- Load an audio file using open from the file menu.
- Choose export to save as a different audio format.

Extract Audio from Video Files

As well as reading audio files, Audacity can extract the audio from video format files such as MP4 and AVI. The process is the same as converting audio formats except that you select video as the source instead of an audio file.

Trim Audio Files

Often when creating audio files, you will have additional recording before and after the sounds you want.

- Open the audio file.
- Select the part to be trimmed using the mouse along the waveform.
- Press the Delete key.
- Export the updated sound to a suitable file format.

This has covered some useful features but has only scratched the surface of what Audacity can do. It can handle multiple tracks and provides filters that allow you to apply different effects to the sounds.

Creating Music with Sonic Pi

There are multiple options for creating music. A useful tool that is included on the Raspberry Pi is Sonic Pi.

Sonic Pi is a code-based music creation and performance tool. It is designed for live music performances but can also be used to compose music that can then be used as background music in computer games. A screenshot of the interface is shown in Figure 8-4. It is considered a programming tool so is on the programming menu in Raspbian.

Figure 8-4. *Screenshot of Sonic Pi music creation tool*

The program has several buffer text edit tabs where code can be entered. The code is based on Ruby which is quite different from Python. It's not possible to go into detail in this book, but an example will be given of how it can be used to create background music.

Music in Sonic Pi is often created using samples which can be manipulated in code. It can also be used by entering musical notes to play a tune using different sample instruments. An example piece of music is included in Listing 8-3.

Listing 8-3. Code to create music in Sonic Pi

```
piano_notes = (ring :r, :c4, :e4, :f4, :g4, :r, :r, :r,
               :r, :c4, :e4, :f4, :g4, :r, :r, :r,
               :r, :c4, :e4, :f4, :g4, :e4, :c4, :e4,
               :d4, :r, :r, :e4, :e4, :d4, :c4, :c4,
               :e4, :g4, :g4, :g4, :f4, :r, :r, :e4,
               :f4, :g4, :e4, :c4, :d4, :c4)

live_loop :piano do
  use_synth :piano
  tick
  play piano_notes.look, attack: 0.2, release: 0.1, amp: 0.5
  sleep 0.25
end
```

Enter the code into one of the buffers and press Run.

This code works by playing musical notes which are stored in an array (list), which is played in the loop. The tune is a simplified version of *When the Saints Go Marching In*. It's a traditional song which doesn't have any copyright issues.

Another example is shown in Listing 8-4 which is an original composition as an example of a different way of creating music in Sonic Pi.

Listing 8-4. Another musical tune created in Sonic Pi

```
# Example tune for Sonic-Pi
tune1_notes = (ring :c4, :d4, :e4, :f4, :g4, :f4, :d4, :c3)
dsaw_notes = (ring :e4, :r, :g4, :r, :a4, :b4, :r, :a4, :b4,
:r, :d5, :r, :b4, :d5, :r, :b4, :r,  :e4, :r, :g4, :r, :a4,
:b4, :r, :a4, :b4, :r, :d5, :r, :b4, :d5, :r, :b4, :r, :g4, :r,
:e4, :r, :e4, :r, :e4, :r, :g4, :r)
piano_notes = (ring :r, :f4, :r, :a4, :r, :g4, :r, :b4)
```

```
with_fx :reverb, room: 1, mix: 0.3 do
  live_loop :tune1 do
    8.times do
      tick
      play tune1_notes.look, release: 0.1, amp: 0.6
      sleep 0.25
    end
  end
end

with_fx :echo do
  live_loop :dsaw do
    use_synth :mod_dsaw
    play dsaw_notes.look, attack: 0.2, release: 0.1, amp: 0.05
    sleep 0.125
  end
end

with_fx :flanger do
  live_loop :piano do
    use_synth :piano
    play piano_notes.look, attack: 0.2, release: 0.1, amp: 0.5
    sleep 0.125
  end
end
```

This uses three different loops with some special effects. This creates a tune that could be used as a background music for a game.

To record the music as a WAV file that can be used in Pygame Zero, click the record button before starting the music, then click the record button again to stop recording, and save it as a file. You will then need to trim out any unwanted silence at the beginning or end using Audacity.

The code is based on Ruby which is very different from Python and is beyond the scope of this book. To learn more about Sonic Pi, there is a good tutorial included in the program. Look in the bottom left corner of Sonic Pi for more details.

Downloading Free Sounds and Music

There are many places where you can download free sounds and music. These include recordings of live effects as well as original music which is made available for free use. Whenever you get sound or music from one of these sites, you need to check that the license allows for your intended use.

Two popular sites for sound effects are Sound Bible (`http://soundbible.com/`) and Freesound (`https://freesound.org`). Most of the sound effects listed on the sites are under an Attribution license which means you can use for most purposes as long as you credit the creator. Some of the samples do restrict the sounds to personal use only, so you may need to be careful with those.

If you are looking for music, then there are several links on the Creative Commons web site `http://bit.ly/ccmusic1`. This site links to other web sites known to have free music, but you will need to check for any restrictions on use.

Adding Sound Effects in Pygame Zero

Having created or downloaded an appropriate sound effect, the next stage is to add it to your games. The sounds can be in WAV or OGG formats.

To play sounds in Pygame Zero, first create a new sub-directory called sounds and copy your sound effects in there. The format of the command to play the sound is `sounds`, followed by the filename (without any extension) and by the appropriate method such as play.

To play the sound "explode.wav", you would use

```
sounds.explode.play()
```

This method should only be used for short sound effects. It loads the entire sound file into memory and can have a significant performance impact if you try to use it on long music files. If you want to play longer pieces of music, then see "Playing Music in Pygame Zero" later in this chapter.

I have included two sound effects in the sounds sub-directory called tankfire.wav and explode.wav. These are used to add some sound effects to the tank game created in the last chapter.

To add the sound of the tank gun firing, add the `sounds.tankfire.play()` entry when the game state is set to `'player1fire'`.

```
game_state = 'player1fire'
sounds.tankfire.play()
```

For the explosion when the shell hits, add `sounds.explode.play()` when the game state is set to `'game_over_1'`.

```
game_state = 'game_over_1'
sounds.explode.play()
```

This should be repeated for `'player2fire'` and `'game_over_2'`. All the required files are included in the supplied source code.

Playing Music in Pygame Zero

When you need some music to play longer, then there is a music player option. The built-in music object provides the ability to play music by loading the track a bit at a time. It only allows a single track to play at a time but can be combined with sounds to have special effects playing at the same time as background music. The music files should be stored in a directory called `music`.

This is a relatively new feature in Pygame Zero and comes with a warning. The music support depends upon the computer system and how well they support playback of a particular codec. It should work with MP3, OGG, and WAV files. MP3 music cannot be played on certain Linux systems, which may be due to patents that have now expired. There have also been reported issues with OGG files. It would seem that WAV may be the safer option, although that may be just that there have been less reports of issues. WAV files are uncompressed which can result in large file sizes.

To play a music track, call music.play with the name of the music track. For instance, if you have a track saved in the music directory called backing.ogg, then you can play it using

```
music.play('backing')
```

The track will then play continuously in the background. If you only wanted the track to play once, such as at the end of a game, then you can use the `play_once` method instead.

```
music.play_one('victorymusic')
```

In either case it will stop any previous track or any in the queue. If you would like to add another track to play next after the current one, then you can use `music.queue`.

It is possible to `stop`, `pause`, and `unpause` the music as well as changing the volume through `set_volume` prefixing the method name with the music object.

Piano Game Created with Tones

Another alternative with Pygame Zero is to play computer-generated sounds using the built-in tone generator. The tone generator can be a useful way for creating sounds, but it uses synthesized sounds and is not as good quality as could be created using sampled sounds. It was added in version 1.2 of Pygame Zero, which is included in the latest version of Raspbian and Mu. It may not work on some older versions.

The tone generator allows you to select the pitch and duration for the tone. These do take a short time to generate (several milliseconds per note), so are better created in advance. This is achieved using `tone.create` with the pitch and duration. For example, to play middle C (4th octave), you would load the tone using

```
middle_c = tone.create('C4', 0.5)
```

Then play using

```
middle_c.play()
```

To make this into a game, I have used the tone generator for a simple piano-based game. The game will allow you to play music using a virtual keyboard and provide a game where the player presses the appropriate key to play a tune. A screenshot is shown in Figure 8-5.

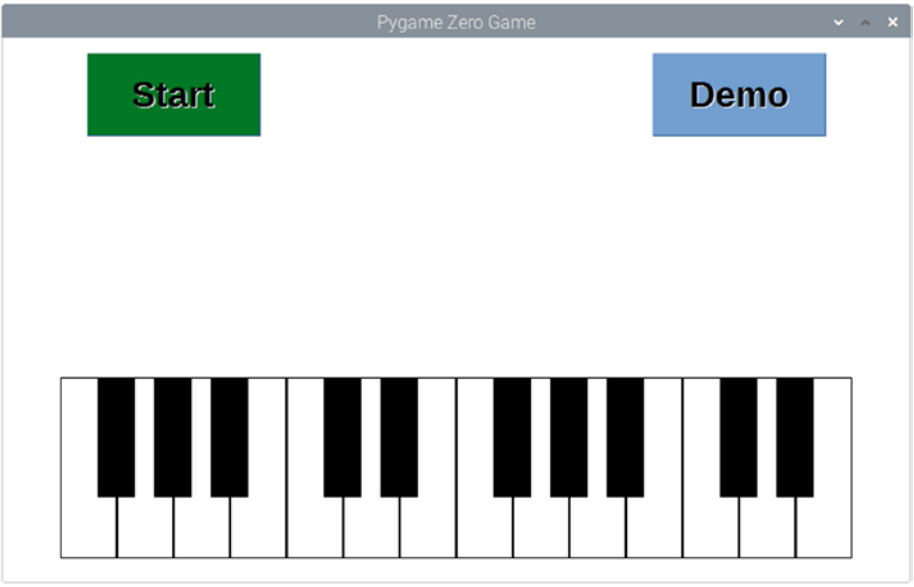


Figure 8-5. *Screenshot of Piano Game*

Clicking any of the keys will play the appropriate note. Clicking the Demo button will play a demonstration of the tune. Clicking Start will play the game; clicking the correct key when the note reaches the target line will score a point.

This game is designed for use with the Raspberry Pi touch screen. It can still be used with a mouse but is harder to play when you need to move the mouse pointer. A limitation to this game is that the player can only press one key at a time. This is a limitation of Pygame Zero, which does not support multi-touch. If you wanted to use multi-touch, then you would need to look at a different programming framework such as Kivy, but that is beyond the scope of this book.

The code for the complete game is shown in Listing 8-5. The buttons are created using shapes so there are no image or sound files required.

Listing 8-5. Code for Piano Game

```
# Piano Game
# Screen resolution based on Raspberry Pi 7" screen
WIDTH = 800
HEIGHT = 410

# Notes are stored as quarter time intervals
# where no note is played use "
# There is no error checking of the tune, all must be valid notes
# When the saints go marching in
tune = [
    ", 'C4', 'E4', 'F4', 'G4', ", ", ", ", 'C4', 'E4', 'F4',
    'G4', ", ", ",
    ", 'C4', 'E4', 'F4', 'G4', 'E4', 'C4', 'E4', 'D4', ", ",
    'E4', 'E4', 'D4', 'C4', 'C4',
```

```
    'E4', 'G4', 'G4', 'G4', 'F4', '', '', 'E4', 'F4', 'G4', 'E4'
    , 'C4', 'D4', 'C4'
    ]

# State allows 'menu' (waiting), 'demo' (play demo), 'game'
(game mode), 'gameover' (show score)
state = 'menu'
score = 0
note_start = (50,250)
note_size = (50,160)
# List of notes to include on noteboard
notes_include_natural = ['F3','G3','A3','B3','C4','D4','E4','F4
','G4','A4','B4','C5','D5','E5']
# List of sharps (just reference note without sharp)
notes_include_sharp = ['F3','G3','A3','C4','D4','F4','G4','A4',
'C5','D5']
note_rect_sharp = {}
note_rect_natural = {}
notes_tones = {}

beats_per_minute = 116
# Crotchet is a quarter note
# 1 min div by bpm
time_crotchet = (60/beats_per_minute)
time_note = time_crotchet/2

# how long has elapsed since the last note was started - or a
  rest
time_since_beat = 0
# The current position that is playing in the list
# A negative number indicates that the notes are shown falling,
# but hasn't reached the play line
note_position = -10
```

```
button_demo = Actor("button_demo", (650,40))
button_start = Actor("button_start", (150,40))
# Setup notes
def setup():
    global note_rect_natural, note_rect_sharp, notes_tones
    i = 0
    sharp_width = 2*note_size[0]/3
    sharp_height = 2*note_size[1]/3
    for note_ref in notes_include_natural:
        note_rect_natural[note_ref] = Rect(
            (note_start[0]+(note_size[0]*i),note_
            start[1]),(note_size)
            )
        # Add note
        notes_tones[note_ref]=tone.create(note_ref, time_note)
        # Is there a sharp note?
        if note_ref in notes_include_sharp:
            note_rect_sharp[note_ref] = Rect(
                (note_start[0]+(note_size[0]*i)+sharp_width,
                note_start[1]),
                (sharp_width,sharp_height)
                )
            # Create version in Note#Octave eg. C#4
            note_ref_sharp = note_ref[0]+"#"+note_ref[1]
            notes_tones[note_ref_sharp]=tone.create(note_ref_
            sharp, time_note)
        i+=1

def draw():
    screen.fill('white')
    button_demo.draw()
    button_start.draw()
```

```
    draw_piano()
    if (state == 'demo' or state == 'game'):
        draw_notes()
        # draw line for hit point
        screen.draw.line ((50, 220), (WIDTH-50, 220), "black")
    if (state == 'game'):
        screen.draw.text("Score {}".format(score),
        center=(WIDTH/2,50), fontsize=60,
            shadow=(1,1), color=("black"), scolor="white")
    if (state == 'gameover'):
        screen.draw.text("Game over. Score {}".format(score),
        center=(WIDTH/2,150), fontsize=60,
            shadow=(1,1), color=("black"), scolor="white")
def draw_notes():
    for i in range (0, 10):
        if (note_position + i < 0):
            continue
        # If no more notes then finish
        if (note_position + i >= len(tune)):
            break
        draw_a_note (tune[note_position+i], i)

# position is how far ahead
# 0 = current_note, 1 = next_note etc.
def draw_a_note(note_value, position):
    if (len(note_value) > 2 and note_value[2] == 's'):
        sharp = True
        note_value = note_value[0:2]
    else:
        sharp = False
    if (position == 0) :
        color = 'green'
```

```
        else:
            color = 'black'
        if note_value != '':
            if sharp == False:
                screen.draw.filled_circle((note_rect_natural[note_
                value].centerx, 220-(15*position)), 10, color)
            else:
                screen.draw.filled_circle((note_rect_sharp[note_
                value].centerx, 220-(15*position)), 10, color)
                screen.draw.text("#", center=(note_rect_sharp[note_
                value].centerx+20, 220-(15*position)),
                    fontsize=30, color=(color))
def update(time_interval):
    global time_since_beat, note_position, state
    time_since_beat += time_interval
    # Only update when the time since last beat is reached
    if (time_since_beat < time_crotchet):
        return

    # reset timer
    time_since_beat = 0

    if state == 'demo':
        note_position += 1
        if (note_position >= len(tune)):
            note_position = -10
            state = 'menu'
        # Play current note
        if (note_position >= 0 and tune[note_position] != ''):
            notes_tones[tune[note_position]].play()

    elif state == 'game':
        note_position += 1
```

```
        if (note_position >= len(tune)):
            note_position = -10
            state = 'gameover'

def draw_piano():
    for this_note_rect in note_rect_natural.values() :
        screen.draw.rect(this_note_rect, 'black')
    for this_note_rect in note_rect_sharp.values() :
        screen.draw.filled_rect(this_note_rect, 'black')

def on_mouse_down(pos, button):
    global state, note_position, score
    if (button == mouse.LEFT):
        if button_demo.collidepoint(pos):
            note_position = -10
            state = "demo"
        elif button_start.collidepoint(pos):
            note_position = -10
            state = "game"
        else:
            # First check sharp notes as they overlap the
            natural keys
            for note_key, note_rect in note_rect_sharp.items():
                if (note_rect.collidepoint(pos)):
                    note_key_sharp = note_key[0]+"#"+note_
                    key[1]
                    if (note_key_sharp == tune[note_position]):
                        score += 1
                    notes_tones[note_key_sharp].play()
                    return
            for note_key, note_rect in note_rect_natural.
            items():
```

```
                if (note_rect.collidepoint(pos)):
                    if (note_key == tune[note_position]):
                        score += 1
                    notes_tones[note_key].play()
                    return
setup()
```

I won't go through this line by line, but I will go through some of the key parts of how the code works.

Starting from the top, you will see that the screen resolution is set to a HEIGHT of only 410. This is because of the resolution of the 7-inch screen after subtracting the top menu bar and window decoration.

The tune is an array which lists the notes that need to be played. In this case it is for *When the Saints Go Marching In*. The music originates from around the late 19th to early 20th century. You could replace that with a more modern tune, but in that case, you would need to take into consideration any copyright issues if you then redistributed the game. The tune needs to be quite simple as only one note can be played at a time, and it will only play quarter notes (crotchets) and rests. In this case the music has been simplified and altered slightly. Chords have been replaced with single notes and the sustain removed on longer notes. The tune should still be recognizable. The notes are stored in the list as strings which are based on the note and the octave, where C4 is middle C. If there is a sharp, then that can be indicated by adding an # between the note and the octave number.

There are several other variables and two Actors which represent the two buttons which are created as images. The tempo is determined by the number of `beats_per_minute`, which is then converted into the length of time between each beat, measured in seconds. In the case of 116 beats per minute, that is the number of quarter notes (crotchets) in a minute. This works out as 0.51 seconds between each quarter note which is each entry in the list. The update function is called approximately every 0.016 seconds,

which should provide a reasonably accurate timing. The note duration is stored in the variable `time_note`, which is half of the time between notes so that the notes don't merge if played quickly.

Another variable is the `note_position` which is used to indicate the position of the array where the current note is. The variable starts at –10 because that allows the notes to fall from the top of the screen. Only when the `note_position` reaches 0 will that note be played (if playing the demo) or the player needs to click the note (in the game). After the variables are the functions followed by a call to `setup`. This is because the functions need to be loaded into memory before the `setup` function tries to use them. Even though the call to `setup` is the last line of the file, it is still run before Pygame Zero runs the update and draw functions.

The `setup` function creates the `rect` objects needed to create the keyboard and pre-loads all the notes of the keyboard. The keys are created as two separate lists, the accidentals (sharp and flat keys) are the black keys and the natural keys are white. The accidentals are referred to as sharp keys in the code as they are created offset from previous natural key, so the sharp named C3 is C#3.

Each of the notes that will be used is pre-loaded into the dictionary `notes_tones` using the code

```
notes_tones[note_ref]=tone.create(note_ref, time_note)
```

This prevents delays when the note is placed. Once created, it can be played by using

```
notes_tones['C3'].play()
```

The `draw_piano` function calls `screen.draw.rect` for the natural keys and `screen.draw.filled_rect` for the accidentals.

The `on_mouse_down` function handles the clicks on the buttons, which sets the state to demo or game as appropriate. It also detects if any of the keys on the piano keyboard are pressed, and if so, it starts the note playing. If in game mode it increases the score if the correct key being pressed.

The `update` function checks to see if enough time has expired for the next note. It uses the argument `timer_interval` which gives the amount of time that has passed since the update function was last run. It uses this to track the time since the last note was played. If it has not reached the time in `time_crotchet`, then it returns from the function. If the timer has exceeded that time, then it can update the `note_position` if it is in either the demo or game states.

The `draw` function displays the buttons, keyboard, and any notes or text that needs to be displayed. A line is drawn as the target for when the note should be played. This uses `screen.draw.line` which uses the start and end coordinates. It also displays the score during the game and the game over message when complete.

This is a simple fun game but would need quite a lot more to create a game that could be used to help teach someone how to play the piano. As mentioned previously the lack of multi-touch is quite limiting. There are still things that you could do to improve the game, such as lighting up the keys when they should be pressed (by using filled_rect with an appropriate color) and providing a way to change the tempo. It's also limited in playing only quarter notes, which could be changed but would involve loading multiple versions of each note depending upon the duration of the note.

Summary

This chapter has covered a few different ways of making and using sounds and music in Sonic Pi. This has included using the Raspberry Pi as a recording device or for converting and editing sounds recorded on another device. It has also covered creating your own music using Sonic Pi.

It then covered the three different ways of playing sounds through Pygame Zero. Sound effects played using the sound object, music played with the music object, and tone using the tone object.

The next chapter is on object-oriented programming, showing an alternative way to creating software using Python.

CHAPTER 9

Object-Oriented Programming

The programs so far have been primarily using a procedural style of programming. The procedural coding style is a good way to learn programming, but there are benefits to using object-oriented programming, which will be covered next. A useful thing about Python is that it supports many different coding styles, even allowing multiple styles in the same code. You have already been using some object-oriented code when making use of Python modules, including Pygame Zero.

After explaining the main concepts of object-oriented programming, this chapter will start a new game. This is based on the classic game "matching pairs", sometimes called memory game. In this game there are several cards face down on the table. Each card has a picture with a matching pair. You need to find the pairs by turning over two cards in each turn. If you are successful, then you keep those cards and score a point.

What Is Object-Oriented Programming?

Object-oriented programming (OOP) is a different style of programming which is based around data and operations on that data. You have already seen this throughout this book when interacting with Pygame Zero. An Actor is an instance of an object. An Actor can be manipulated by changing

© Stewart Watkiss 2020
S. Watkiss, *Beginning Game Programming with Pygame Zero*,
https://doi.org/10.1007/978-1-4842-5650-3_9

its attributes such as pos (which changes its position on the screen) and can have operations performed on it such as the draw method (which draws it on the screen).

The four main concepts in object-oriented programming are encapsulation, data abstraction, polymorphism, and inheritance. These are known as four pillars of object-oriented programming:

- **Encapsulation** is about keeping the internal state private to protect the data. Python doesn't have true encapsulation which is normally achieved using private variables and methods. Python does however have a convention of using __ (double underscore) before the name to prevent accidental use of a private variable or method.
- **Data abstraction** is an extension of encapsulation which makes it easier to hide the details of internal operations. This helps create a simple, more stable interface.
- **Polymorphism** allows a child to act as though it is its parent. This is a way to allow better code reuse through sharing code. It can also provide different implementations of methods based on the input parameters.
- **Inheritance** allows the reuse of parts of code between similar objects.

This book will concentrate on the specific aspects that object-oriented programming provides which make it easier to design and program games. It will show how encapsulation and abstraction can be used to make game programming simpler and easier to write and understand. It will also give an example of how inheritance can help reduce the amount of code that needs to be written by making use of existing code. It will also help

reduce the number of global variables which make the code difficult to understand and to debug when things go wrong.

OOP Classes and Objects

Object-oriented programming is based around objects. In the real world, we think of objects as physical things, such as a laptop, a phone, or a book. In programming objects can be anything that stores data or what you interact with. Built-in objects include the screen, an Actor, or a sound, and you can create your own objects for just about anything. The object holds the data as internal variables of the object which can be read and manipulated through the object attributes. Most objects also have operations (methods) that can be performed on the object.

Creating a Class, Attributes, and Methods

To create objects, there needs to be a blueprint that tells the computer what to do with the object. The blueprint is known as a class. Each object is known as an instance of the class. Normally a class is created in a separate file named the same as the class, ending with .py. The code in Listing 9-1 shows a skeleton class that could be used to represent a ball. The class has the name Ball which follows a convention of using an initial capital letter for a class name. If there are multiple words in the class, then the first letter of any word is also capitalized such as MyClass. The filename is normally the same as the class name, but all lowercase. In this example the class name is Ball, so the file is called ball.py. You don't need to split each class into a file; you could have them in the existing file or a file with multiple classes, but it's usually a good idea to have one file per class.

Listing 9-1. Example of a OOP class

```
class Ball():
    shape = "sphere"

    def __init__(self, position, radius, color):
        self.position = position
        self.radius = radius
        self.color = color

    def draw(self, screen):
          screen.draw.filled_circle(self.position, self.radius,
          self.color)
```

The file starts with a class definition to state that this is an object-oriented class. This is the blueprint for creating Ball objects.

The first variable listed is called `shape` which has the value "sphere". This is a class variable. There is only one instance of the variable which spans all instances of the class. It is most often used for values that don't change, although in the next chapter, you will see an example of a class variable which is edited by multiple objects.

It is more common to have instance variables, which are unique to each instance of the class. Instance variables are created within methods and are prefixed with the `self` keyword. The instance variables that are created prefixed with self will be available to all methods defined in the class. Local variables can also be created without the self keyword, and they will behave in the same way as local variables in procedural programming.

In this class there are two methods. A method is essentially the same as a function except that they perform operations on the object and have access to the data within the object. They are created using the def keyword similar to how functions are defined.

The first method is called `__init__`. This is known as the constructor and is called whenever you create an instance of the class. There is a method called `draw`, which will draw the ball on the screen. You can see that both the methods have `self` as the first argument. The `self` keyword is used to represent the instance of the class and is used by methods to access the data within the instance of the class. You do not provide anything on the argument when calling the method; instead, `self` is passed automatically to the method and is used to access the instance variables.

The constructor method (`__init__`) is run when an object is first created. It is often used to set up any variables. In this case it takes three values for the position, size, and color. The arguments in the method are always held as local variables so these are copied into `self.position`, `self.radius`, and `self.color`. These are stored in the object and can be read and written to by any of the other methods as required. There is no need to mark these as global; they are automatically available to all methods through the self keyword.

The next method is `draw` which draws the ball on the screen as a filled circle. It can access all the variables that were previously set through the constructor which are `self.position`, `self.radius`, and `self.color`. There is one anomaly with this method. Previously when calling `screen.draw` operations, it used the built-in screen object. That works from within the draw function in the top-level function that Pygame Zero uses, but when using a separate object, the reference to the screen object needs to be provided as an argument.

Creating an Instance of a Class (Object)

After creating the Class, you can create an instance of the class. This is like creating a physical object using the Class as its blueprint. This is shown in Listing 9-2 which creates an executable program balldemo.py.

Listing 9-2. Creating instances of a class

```
from ball import Ball

WIDTH = 800
HEIGHT = 600

ball1 = Ball((400,300),10,"red")

def draw():
    ball1.draw(screen)
```

This is a basic program which when run will display the ball in the center of the screen. The first line imports the class. It reads it from the file ball.py (in the same directory as the main program file) and from that imports the class Ball. Once imported, you can use the class.

A new object is created called ball1, which is an instance of the class. To create the instance, it uses the name of the class followed by the arguments listed in the constructor. This creates the new instance and runs the __init__ method.

Within the draw function, the draw method is called on the instance ball1 which draws the ball. The built-in screen needs to be passed to the draw method of ball1 so that it is able to draw to the screen.

This creates one instance of a ball, but you could create a second instance using

```
ball2 = Ball((100,100),20,"green")
```

Then add to the draw method using

```
    ball2.draw(screen)
```

which draws a large green ball in the top left of the screen.

Accessing Attributes of an Object

The variables within an object are known as attributes. As you have seen, the variables in the class definition are prefixed with self which refers to the instance of the class.

If you are outside of the class, then you can replace self with the instance name. In the case of the instance ball1, you can access its variable using

```
ball1.color
```

If you wanted to check the value, then you could use

```
if ball1.color == "red":
...
```

Or if you wanted to change the value, then you could use

```
ball1.color = "orange"
```

This applies to all the variables created in the __init__ method, or any other methods as long as the variables are prefixed with self.

Terminology

One thing about object-oriented programming is all the new terminology it uses. Here is a recap of some of the terminology that has been covered to help make it clearer. Figure 9-1 shows the relationship between the class and the instances.

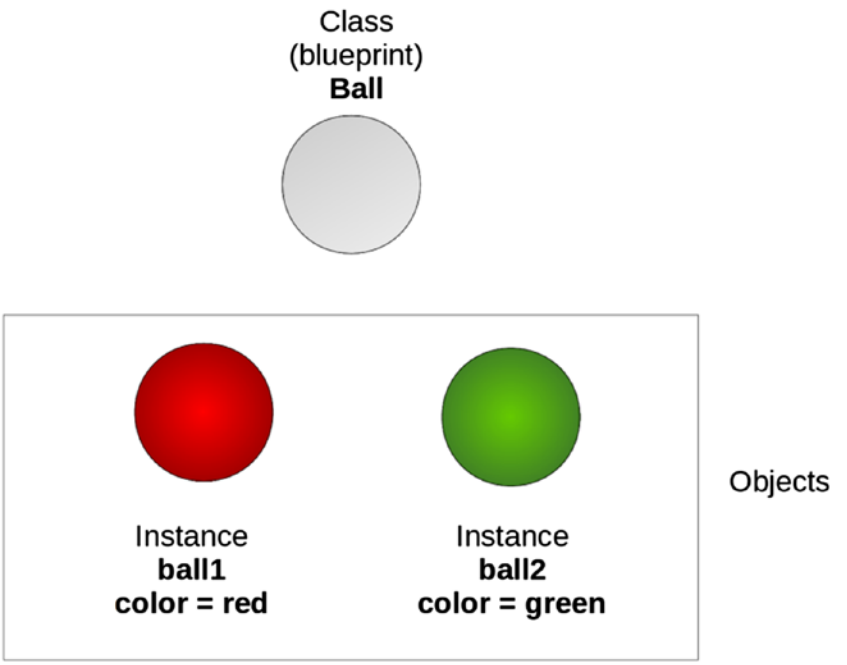


Figure 9-1. *Class to instance relationship*

The Class shown at the top of the diagram is a blueprint for creating the objects. It defines how the class will be created, what attributes they have, and the operations that can be performed on them. You cannot normally use the class directly and instead need to create specific objects known as instances. In this example we created two instances known as ball1 and ball2. These were both created from the same blueprint, so will behave in a similar way, but they have their own set of attributes (stored as the instance variables). Using this ball1 color is set to red, whereas ball2 color is set to green.

Encapsulation and Data Abstraction

As mentioned previously two of the benefits of object-oriented programming are encapsulation and data abstraction. The benefit of this to the programmer is that it separates the internal structure of the class from the code that is making use of that. This can make it easier when different programmers work on the same project, and it can make it easier to make changes in the future.

One scenario that this can be useful is when multiple people are working on the same project. If the programmers agree in advance on the interface to the class, then they can work independently. This has implications beyond an individual project; it also helps with creating libraries of code that can be used by others.

Another scenario is where you want to make changes to the code in the future. If you wanted to add a new feature or improve existing code, then it separates that class from others. If you keep the interface the same, you can change any of the internal code in the ball class to draw the ball in a completely different way.

Python does not enforce the data abstraction as strictly as some other programming languages. It is possible to change any of the instance variables from outside of the class which could result in a loss of the data abstraction. If you want to make some variables hidden from outside the class, then you can hide them by prefixing them with __ (two underscore characters). Even if using this, it is not full data abstraction. Python is useful for object-oriented programming but relies on the programmers to create a stable interface and to use good programming best practices.

Inheritance

Inheritance allows the creation of a child class which inherits some of its attributes and operations from a parent class. This is a way of avoiding the duplication of code. This has the advantage of saving typing from the programmer but more importantly can help reduce the number of bugs.

To demonstrate this, you can imagine a flight game which has an airplane class to represent the plane. If a game had different kinds of planes, then they may have different things that those planes may be able to do. This is illustrated in Figure 9-2 which shows three very different types of plane: a passenger plane, a cargo plane, and a fighter plane.

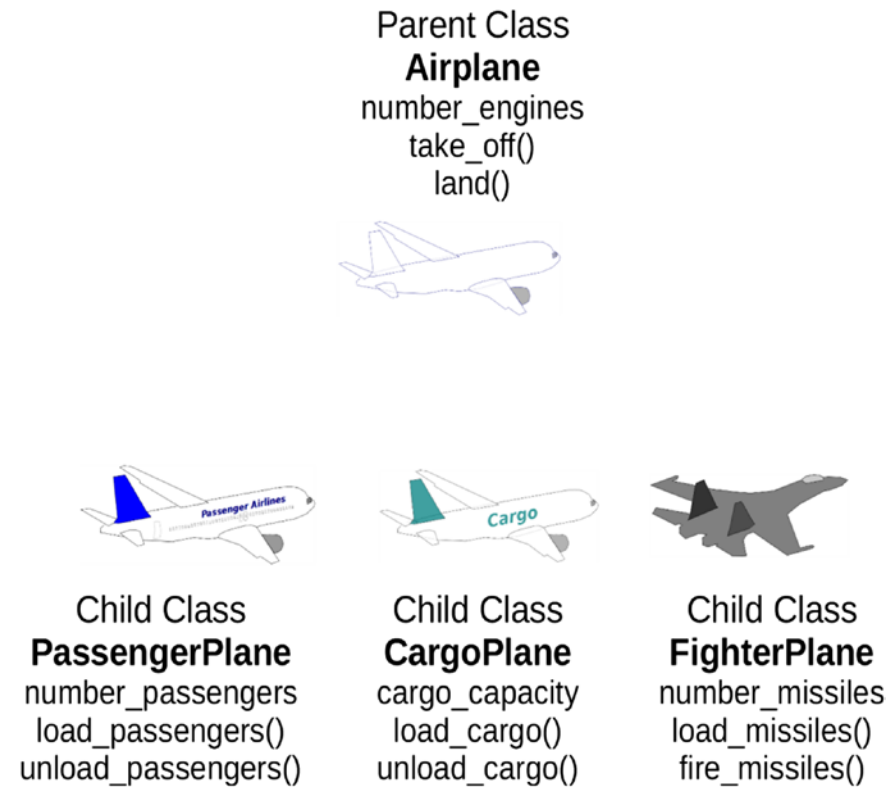


Figure 9-2. *Inheritance using parent and child classes*

These are all different types of planes so will have some things in common. Other things may only apply to certain types of plane. All planes have a number of engines and the ability to take off and land, so those can all be configured in the Airplane (parent) class. There are some other attributes and operations that are unique to certain types of balls. For example, the passenger plane has a number of passengers, but that does not make sense for a fighter plane. The fighter plane can load and fire missiles, but neither passenger nor cargo planes would need that ability.

Inheritance works by defining the common attributes and methods in the parent class and then adding any unique features into the child class. Code that uses the child class can use the operations that are inherited from the parent or that are in the child class. The code in Listing 9-3 is a demonstration of how a child class refers to its parent.

Listing 9-3. Inheritance demonstration showing a child class

```
from airplane import Airplane

class PassengerPlane(Airplane):

    def __init__(self):
        Airplane.__init__(self)
        self.number_passengers = 0

    def load_passengers (number_passengers):
        self.number_passengers = number_passengers

    def unload_passengers ():
        self.number_passengers = 0
```

This code inherits from the Airplane class, adding a new attribute called `number_passengers` and two new methods called `load_passengers` and `unload_passengers`. An instance of the passenger plane can be created using

```
plane1 = PassengerPlane()
```

This will then have access to its own methods such as

```
plane1.load_passengers(20)
```

as well as the parent methods such as

```
plane1.take_off()
```

Design for Object-Oriented Programming

When I first started using object-oriented programming, one of the challenges I found was deciding what objects to define. In the case of something that represents an object in the real world, it's obvious. In the plane example, it's obvious that would be considered an object, but what about something less tangible? Is the player's score an object, or is it an attribute? If it's an attribute, then what's the object that the score belongs to?

In some cases, there is no definitive answer to whether something should be an object or not. It depends upon the type of game, how it interacts with other objects, and the programmer's personal preference. I will show the technique I use which you may find helpful when designing your own games. It's not mandatory, and with experience, you may not need to do this, but it is a technique I often use when creating a new program.

This technique helps show what classes to create and what their attributes and operations should be. First start by writing down, in one or two paragraphs, what the game does and how it will work. You should actually write this down (or type it into a computer) and write in full sentences rather than bullet points. Don't try and do this from memory as you need to see the words for the next step. Now look through the description you have written and find all the nouns. Write the nouns down as possible class names. Next look at all the adjectives, then apply to those nouns and write them under the associated noun. These will be the attributes (variables). Then look for the active verbs and how they relate to

the nouns. List these under their associated noun; these will become the operations (methods).

Here's a quick summary:

- Nouns (names) – Classes
- Adjectives (describing words) – Attributes (variables)
- Active verbs (action words) – Operations (methods)

These words then provide suggestions for the classes, attributes, and operations. Note that these are suggestions only; you should then apply your own judgment on whether those are all necessary. It may be that some items are similar to each other or that some of the nouns are not significant enough to have their own class. It does however give you a starting point to design your class structure. You can always revisit the structure as you develop the game.

Matching Pairs Memory Game

Now that you are familiar with some of the aspects of object-oriented programming, it's time to put that into practice with another game. This is a digital version of a traditional card-based memory game. The game is normally played using picture cards with each card being one of a matching pair. The cards are placed face down, and players take it in turn to turn over two cards trying to find the matching pairs. A screenshot of the game is shown in Figure 9-3.

Figure 9-3. *Screenshot of matching pairs memory game*

In the traditional game, there are normally two or more people taking it in turns to try to win the most points, but in this version, there will be just one player who will play against the clock. To create this using object-oriented programming, I first followed the design methodology explained earlier. I have written down the following explanation of the game.

This is a memory game. The game starts with a number of cards which are face down. Each card has a picture on it which matches one other card. The player turns over two cards and looks to see if they match. If they do match, then the player's score is increased. If they don't match, then the cards are reset and the player has another go. There is a timer which counts down over time, and if the timer value is zero, then the game ends. If the player matches all pairs on a level, then the player is awarded bonus points and the cards are dealt again.

I have then identified the nouns marked with (), which will become the classes. Adjectives are marked with []; they will be the attributes. Active verbs are marked with {}; they will become the operations.

This is a (memory game). The game starts with a number of (cards) which are [face down]. Each (card) has a [picture] on it which matches one other card. The (player) {turns over} two cards and looks to see if they {match}. If they do match, then the player's (score) is {increased}. If they don't match, then the cards are {reset} and the player has another go. There is a (timer) which counts down over time, and if the [timer value] is zero, then the game ends. If the player matches all pairs on a (level), then the player is awarded [bonus points] and the cards are (dealt) again.

Remember that these are guidelines rather than fixed rules. You can use your own discretion when identifying the appropriate words and phrases, or you can do that at a later stage. This is just a way to make it easier for the programmer to decide on how to create the different classes. I have then made them into the following:

Memory Game

Attributes: N/A

Operations: reset; deal

Player

Attributes: score; bonus points

Operations: N/A

Card

Attributes: face down; picture

Operations: matches (another card); reset

Timer

Attributes: time remaining

Operations: count down; reach zero

Level

Attributes: N/A

Operations: N/A

This should only be considered a starting point. You should now review and see if there are any you want to change now. You can revisit this when implementing the code as it may need to change.

There are some things that made sense to change immediately. One thing is that memory game defines the overall game. We can create this without defining it as a separate class which will allow it to use the Pygame Zero draw and update functions; it will also need a variable for tracking the state of the game.

Another that is worth considering is that the bonus points don't need to be an attribute but can instead be added to the normal score; it makes sense for adding bonus points to be an operation instead of an attribute. Also, the level probably doesn't need to be a class, but can be incorporated into the memory game or player classes.

Having worked through that exercise, you should have an idea of what classes will be needed and some of the attributes and operations. As you write the code, you may decide that there are other classes, attributes, or operations needed. You can add any additional attributes or operations as you create the code.

There are two versions of code in the supplied source code, one is the initial version and the other is an updated version. To try out the code, you will need some card pictures. The source code for this book includes some photographs I took around the Lake District in the United Kingdom. These are used to create the cards, but you could use your own photos or pictures to personalize the game.

Creating the Classes

There is no fixed order that you create the code. I usually start by creating some of the classes before creating the main program. This means that I can perform some testing on those classes before writing the main program. The classes I created are explained here.

Timer Class

The first class is the Timer class. The timer class is used to track how much time has lapsed so that the game has to be completed within the allotted time. This is a very simple class but is useful to demonstrate how the class is created. The code is shown in Listing 9-4 and is saved in a file called timer.py.

Listing 9-4. Timer class

```
import math
import time

class Timer():

    def __init__(self, start_count):
        self.start_count = start_count
        self.start_time = time.time()

    # start count down, with optional parameter to replace the
      start_count value
    # -1 is used as a "magic number", this method should only
      be called with positive number
    # if it isn't given a number then -1 indicates no new time give
    def start_count_down(self, new_time = -1):
```

```
        if (new_time >= 0):
            self.start_count = new_time
        self.start_time = time.time()

    def get_time_remaining(self):
        current_time = self.start_count + self.start_time -
        time.time()
        if (current_time <= 0):
            return 0
        return math.ceil(current_time)
```

The file starts by importing the two modules, math and time. As the name suggests, these provide mathematical and time functions.

The class is defined by the entry

```
class Timer():
```

This creates the class definition for a regular class. The class name is Timer. The constructor is defined as the __init__ method. Its first argument is self, which is always included in a class constructor. It then takes one argument which is the start_count value. This is a countdown timer with the start_count value as the number of seconds to count down from.

```
def __init__(self, start_count):
        self.start_count = start_count
        self.start_time = time.time()
```

The constructor also creates a variable `self.start_time` which is passed the number of seconds since the epoch. On a Linux system, the epoch is 00:00:00 1970-01-01 UTC (January 1, 1970). The actual time is not important for this game, but it is used as a reference point to measure the amount of time which has elapsed.

The `start_count_down` method is used to start the timer. It includes the usual self argument. It then has an argument `new_time = -1`. By setting `new_time` to have a value in the arguments, it makes the argument optional. If an argument is passed, then that will be placed in the `new_time` variable; if there is no argument passed, then the variable will take the value –1.

```
def start_count_down(self, new_time = -1):
```

If `new_time` is changed, then that is placed in the `self.new_time` variable. The method then restarts the timer by storing the current time (in seconds since the epoch) in the `self.start_time` variable.

The `get_time_remaining` method returns the number of seconds remaining before the counter reaches zero, or zero if the countdown time has already been exceeded. The `math.ceil` function is used to round the time up to the nearest whole second. This makes it so that the countdown always shows a whole number of seconds and only decrements when a full second has passed.

Card Class

The next class is the card class which displays the card to the player. This is shown in Listing 9-5 and should be saved as card.py. This class also demonstrates how inheritance works as it extends the Actor class.

Listing 9-5. Card class with inheritance

```
from pgzero.actor import Actor

# Card is based on an Actor (uses inheritance)
class Card(Actor):

    def __init__(self, name, back_image, card_image):
        Actor.__init__(self, back_image, (0,0))
        self.name = name
        self.back_image = back_image
```

```
        self.card_image = card_image
        # Status can be 'back' (turned over) 'front' (turned
        up) or 'hidden' (already used)
        self.status = 'back'

    # Override Actor.draw
    def draw(self):
        if (self.status == 'hidden'):
            return
        Actor.draw(self)

    def turn_over(self):
        if (self.status == 'back'):
            self.status = 'front'
            self.image = self.card_image
        elif (self.status == 'front'):
            self.status = 'back'
            self.image = self.back_image
        # Attempt to turn over a hidden card - ignore
        else:
            return

    def hide(self):
        self.status = 'hidden'

    # When unhide set it to back image
    def unhide (self):
        self.status = 'back'
        self.image = self.back_image

    def is_hidden (self):
        if self.status == 'hidden':
            return True
        return False
```

```
    # Is it turned to face forward
    def is_faceup (self):
        if self.status == 'front':
            return True
        return False

    def reset (self):
        self.unhide()

    def set_position(self, x, y):
        self.x = x
        self.y = y

    def equals (self, othercard):
        if self.name == othercard.name:
            return True
        return False
```

The first entry on the file is to import the Actor class which is in pgzero.actor.

```
from pgzero.actor import Actor
```

This is something that the Pygame Zero normally loads automatically, but because this is in a separate class file, it needs to be imported explicitly.

It then defines a new class called Card:

```
class Card(Actor):
```

The word "Actor" in brackets indicates that this is to be a child of the Actor class.

The new class imports the methods from the parent class which can then be overridden. The `__init__` method is included which overrides the constructor method. It includes the reference to self and then three variables. The name variable is a label used to check for a matching pair; the other arguments are used to pass the image filenames to the Card class.

```
    def __init__(self, name, back_image, card_image):
        Actor.__init__(self, back_image, (0,0))
        self.name = name
        self.back_image = back_image
        self.card_image = card_image
        self.status = 'back'
```

There is a variable called status which will track if the card is face up (front) or face down (back) or if it is already used (hidden).

You will also see within that block of code that the `Actor.__init__` method is then called. This is the same method as that of an Actor being created without being part of a child class. In this case the call is made directly to the parent's `__init__` method by prefixing it with the name of the parent class. If there is no `__init__` method, then the parent's `__init__` will be called instead.

The next method in the Card class is the draw method, which also overrides the method from the parent class. This is created so that it only displays the card if the status is not equal to hidden. If the card is not hidden, then it makes the call to the parent's draw method by calling `Actor.draw(self)`.

There are then some methods that perform operations on the Card object. These methods don't exist in the parent class. They are methods that are specifically for cards and in most cases wouldn't make sense on other Actors that aren't Cards.

The card has two different images. The card starts by showing the `back_image`, but it is changed to the `card_image` when the `turn_over` method is called. This is done by changing the `self.image` property, which is a feature of the Actor class. Some of the other methods are primarily getting and setting the values of variables. For example, the `hide` and `unhide` methods are used to change the value of the hidden variable, and the `is_hidden` method returns the value of the hidden variable. These methods are not actually required as it's possible to change the hidden

variable directly. There are pros and cons to whether you update and read the variables directly or using methods. The Python mantra is usually to take the simpler option of updating the variable directly, whereas for some other programming languages, it is encouraged to have getter and setter methods whenever you need to access a variable in an object.

I usually prefer to use methods to access variables. The main advantage is that it supports the concept of data abstraction. Imagine that at a future date you decided to add an option to partially hide a card. For example, you may add a feature that a card can only be turned over if it hasn't been turned over in the previous turn; if a card has been used in the previous turn, then it should be grayed out to show that it cannot be used. To achieve this, you may change the hidden variable so that instead of being a Boolean which can only hold two states (True or False), you use a number to represent the amount of transparency. If you only use methods to access the values, then you could add this as a new feature without breaking the way that the library is used. This is particularly useful when you reuse the same code between different programs.

The equals method compares the name of the current card with the name of another card. The argument `othercard` will be passed the object from which it can check the name of the other card.

GamePlay Class

At this point I decided not to create a separate Players class as it would just hold a single variable for the score. It is not normally worth creating a class for just one variable.

Initially I incorporated all the score and state tracking into the main program file. When I did the program file, it became long and difficult to understand how it worked. This is known as a bad smell. To avoid this, I created a new class called the GamePlay class. This is known as refactoring the code, which is when the code is updated, but not normally adding any

additional functionality. It's normally a case reorganizing and changing the code to make it easier to read, or perhaps more efficient.

Note Bad smell is a programming term which indicates a bad code design. It is not usually a bug, but may slow down development, make it hard to understand the code, or increase the risk of bugs in the future.

Another advantage of creating the GamePlay class is that it separates the user score from the main code and should make it easier to make into a two-player game at a later stage.

The GamePlay class is shown in Listing 9-6 and is saved as gameplay.py.

Listing 9-6. GamePlay class

```
# State is tracked as a number, but to make the code readable
  constants are used
STATE_NEW = 0               # Game ready to start, but not running
STATE_PLAYER1_START = 1     # Player 1 to turn over card
STATE_PLAYER1_CARDS_1 = 2   # Card 1 turned over
STATE_PLAYER1_CARDS_2 = 30 # Card 2 turned over
STATE_END = 50

# Number of seconds to display high score before allowing click
  to continue
TIME_DISPLAY_SCORE = 3

class GamePlay:

    def __init__ (self):
        # These are what we need to track
        self.score = 0
        self.state = STATE_NEW
```

```
        # These are the cards that have been turned up.
        self.cards_selected = [None, None]

    # If game has not yet started
    def is_new_game(self):
        if self.state == STATE_NEW:
            return True
        return False

    def is_game_over(self):
        if self.state == STATE_END:
            return True
        return False

    def set_game_over(self):
        # player gets to see high score
        self.state = STATE_END

    def is_game_running(self):
        if (self.state >= STATE_PLAYER1_START and self.state <
        STATE_END):
            return True
        return False

    def start_game(self):
        self.score = 0
        self.state = STATE_PLAYER1_START

    def set_new_game(self):
        self.state = STATE_NEW

    def is_pair_turned_over(self):
        if (self.state == STATE_PLAYER1_CARDS_2):
            return True
        return False
```

```
    # Return the index position of the specified card
    def get_card(self, card_number):
        return self.cards_selected[card_number]

    # Point scored, so add score and update state
    def score_point(self):
        self.score += 1
        self.state = STATE_PLAYER1_START

    # Not a pair - just update state
    def not_pair(self):
        self.state = STATE_PLAYER1_START

    # If a card is clicked then update the state accordingly
    def card_clicked(self, card_index):
        if (self.state == STATE_PLAYER1_START):
            self.cards_selected[0] = card_index
            self.state = STATE_PLAYER1_CARDS_1
        elif (self.state == STATE_PLAYER1_CARDS_1):
            self.cards_selected[1] = card_index
            self.state = STATE_PLAYER1_CARDS_2
```

The main things that the GamePlay class provides are tracking the state of the game and keeping track of the score. The file starts by creating some constants which are used to denote the different states. These aren't necessary, but `state == STATE_PLAYER1_START` is more readable than `state == 1`. The constants are all in capitals to make it clear that they are constants and shouldn't be changed, but as far as Python is concerned, these are just variables. The value of the variables isn't important as long as they are always referenced using the constant.

The `__init__` method is used to create the score and state variables. The next variable `cards_selected` is a list which tracks which of the cards has been turned face up. It starts with each of the values as `None`. None is a

special variable type that indicates that no value has been set. It is needed so that the two entries exist so that the card number can be stored in them.

The methods included are mainly about providing the status of the game. For example, the method `is_new_game` will return a value of True if the game is about to start; otherwise, it will return False. These are provided as it makes it easier to understand what the code is doing compared to checking against the status code.

The one method that is a little more complex is the `card_clicked` method. This method looks at the current state to determine whether the card that has been clicked is the first or the second card and updates the appropriate entry in `cards_selected`.

Program File

Having created the class files, the program file is much simpler. It's still quite long, but shorter than if all the code was in a single file. The code is shown in Listing 9-7.

Listing 9-7. Memory game main program file

```
# Memory Card Game - PyGame Zero
import random

from card import Card
from timer import Timer
from gameplay import GamePlay

# These constants are used to simplify the game
# For more flexibility these could be replaced with
  configurable variables
# (eg. different number of cards for different difficulty levels)
NUM_CARDS_PER_ROW = 4
X_DISTANCE_BETWEEN_CARDS = 120
```

```
Y_DISTANCE_BETWEEN_CARDS = 120
CARD_START_X = 220
CARD_START_Y = 130
TIME_LIMIT = 60

TITLE = "Lake District Memory Game"
WIDTH = 800
HEIGHT = 600

cards_available = {
    'airafalls' : 'memorycard_airafalls',
    'ambleside' : 'memorycard_ambleside',
    'bridgehouse' : 'memorycard_bridgehouse',
    'derwentwater' : 'memorycard_derwentwater',
    'ravenglassrailway' : 'memorycard_ravenglassrailway',
    'ullswater' : 'memorycard_ullswater',
    'weatherstone' : 'memorycard_weatherstone',
    'windermere' : 'memorycard_windermere'
    }

card_back = "memorycard_back"

## Setup instance variables
count_down = Timer(TIME_LIMIT)
game_state = GamePlay()
all_cards = []
# Create individual card objects, two per image
for key in cards_available.keys():
    # Add to list of cards
    all_cards.append(Card(key, card_back, cards_available[key]))
    # Add again (to have 2 cards for each img)
    all_cards.append(Card(key, card_back, cards_available[key]))
```

```
## Functions are defined here - the rest of the initialization
## is at the bottom of the file

# Shuffle the cards and update their positions
# Do not draw as this is called before the screen is properly setup
def deal_cards():
    # Create a temporary list of card indexes that is then shuffled
    keys = []
    for i in range (len(all_cards)):
        keys.append(i)
    random.shuffle(keys)

    # Setup card positions
    xpos = CARD_START_X
    ypos = CARD_START_Y
    cards_on_row = 0
    for key in keys:
        # Reset (ie. unhide if hidden and display back)
        all_cards[key].reset()
        all_cards[key].set_position(xpos,ypos)
        xpos += X_DISTANCE_BETWEEN_CARDS

        cards_on_row += 1
        # If reached end of row - move to next
        if (cards_on_row >= NUM_CARDS_PER_ROW):
            cards_on_row = 0
            xpos = CARD_START_X
            ypos += Y_DISTANCE_BETWEEN_CARDS

def update():
    if (game_state.is_new_game()):
        pass
    elif (game_state.is_game_over()):
        pass
```

```
    else:
        if (count_down.get_time_remaining()<=0):
            game_state.set_game_over()

# Mouse clicked
def on_mouse_down(pos, button):
    # Only interested in the left button
    if (not button == mouse.LEFT):
        return
    # If new game then this click is to start the game
    if (game_state.is_new_game()):
        game_state.start_game()
        # start the timer
        count_down.start_count_down(TIME_LIMIT)
        deal_cards()
        return
    # If game over then this click is to get to new game screen
    if (game_state.is_game_over()):
        # Make sure the timer has reached zero (short delay to
        see state)
        if (count_down.get_time_remaining()<=0):
            game_state.set_new_game()
        return

    ## Reach here then we are in game play
    # First check for both already clicked and this is a click
      to test
    if (game_state.is_pair_turned_over()):
        if (all_cards[game_state.get_card(0)].equals(all_
        cards[game_state.get_card(1)])):
            # Add points and hide the cards
            game_state.score_point()
```

```
            all_cards[game_state.get_card(0)].hide()
            all_cards[game_state.get_card(1)].hide()
            # Check if we are at the end of this level (all
            cards done)
            if (end_level_reached()):
                deal_cards()
        # If not match then turn both around
        else:
            all_cards[game_state.get_card(0)].turn_over()
            all_cards[game_state.get_card(1)].turn_over()
            game_state.not_pair()
        return

    ## Otherwise we just turn over the next card if clicked
    for i in range (len(all_cards)):
        if (all_cards[i].collidepoint(pos)):
            # Ignore if card hidden, or has already been turned up
            if (all_cards[i].is_hidden() or all_cards[i].is_
            faceup()):
                return
            all_cards[i].turn_over()
            # Update state
            game_state.card_clicked(i)

# If reach end of level ?
def end_level_reached():
    for card in all_cards:
        if (not card.is_hidden()):
            return False
    return True
```

```
def draw():
    screen.fill((220, 220, 220))
    if (game_state.is_new_game()):
        screen.draw.text("Click mouse to start", fontsize=60,
        center=(WIDTH/2,HEIGHT/2), shadow=(1,1),
        color=(255,255,255), scolor="#202020")
    if (game_state.is_game_over()):
        screen.draw.text("Game Over\nScore: "+str(game_state.
        score), fontsize=60, center=(WIDTH/2,HEIGHT/2),
        shadow=(1,1), color=(255,255,255), scolor="#202020")
    if (game_state.is_game_running()):
        for card in all_cards:
            card.draw()
        screen.draw.text("Time remaining: "+str(count_down.
        get_time_remaining()), fontsize=40, bottomleft=(50,50),
        color=(0,0,0))
        screen.draw.text("Score: "+str(game_state.score),
        fontsize=40, bottomleft=(600,50), color=(0,0,0))

### End of functions - start of initialization code
deal_cards()
```

Unlike the other files, the main program file is not created as a separate class. This is different to some other programming languages which would require everything to be object-oriented. In the case of Python, that's optional, and in the case of Pygame Zero, it's easier to not use a separate class in the main part of the program. Instead the program makes use of the Pygame Zero hooks such as the `draw` and `update` functions.

To understand the program, it's useful to take a look at the overall file. The imports and variables are defined at the top of the file, along with the initialization of the class instances. The functions are in the middle, and

then additional code that runs during the initialization of the program is at the bottom after the line

```
### End of functions - start of initialization code
```

The program first imports the random module and then the three classes created previously: Card, Timer, and GamePlay. There are several constants defined which are used for the spacing of the cards and game settings such as the duration of the timer. There is also a variable for the filename of the image for the back of the cards as well as a dictionary with the filenames for the different card images. These settings would typically be stored in a separate configuration file, but to keep it simple, they have been included in the memory.py file. There is then an empty list created called `all_cards` which will hold the instances of the Card class.

The creation of the instances for the classes is handled next. The `Timer` and `GamePlay` classes only need a single instance created by a normal assignment.

```
count_down = Timer(TIME_LIMIT)
game_state = GamePlay()
```

For the Cards class, there needs to be an instance for each of the cards that will be displayed. A for loop is used to create these and append them to the `all_cards` list. This is a list of Card objects. Two instances are created for each card to have the matching pairs in the list.

```
all_cards = []
# Create individual card objects, two per image
for key in cards_available.keys():
    # Add to list of cards
    all_cards.append(Card(key, card_back, cards_available[key]))
    # Add again (to have 2 cards for each img)
    all_cards.append(Card(key, card_back, cards_
    available[key]))
```

The functions are listed after this, followed by the call to `deal_cards` at the bottom of the file. This needs to be placed after the `deal_cards` function is defined; otherwise, it will cause an error. Placing it at the end makes the code easier to follow.

The `deal_cards` function works by creating a list of all keys from the cards. It then calls the `random.shuffle` function which mixes the cards up into a random order. It then updates each of the cards with their coordinates based on the spacing between the cards. The cards can be accessed by using their index in the list as shown in the following example entry:

```
all_cards[key].set_position(xpos,ypos)
```

Next is the update function. It first checks to see if the game is either new or finished. If that is the case, it does nothing which is indicated by the `pass` keyword. Using pass does nothing, but it can be useful as a placeholder if you plan to add additional code in the future. If the game is in progress, then it calls the `get_time_remaining` method on the timer and changes the game state if the end of the game is reached.

```
if (count_down.get_time_remaining()<=0):
            game_state.set_game_over()
```

Most of the code to update the game is driven by the mouse action and so is in `on_mouse_down` rather than the `update` function.
The on_mouse_down function is handled differently on whether the user is playing the game. If the game is not in progress, then the click changes the state of the game, such as to start the game. If the game is in progress, then it will first test for both cards already being turned over. If it is, then it tests to see if the two cards match and either hides the cards (if they do) or resets the cards back to face down. If both cards are not yet turned over, then it checks to see if a card has been clicked using the `collidepoint` method and if so turns the card over and updates the game state. There is also a check against the `end_level_reached` function which checks to see

if all cards have been turned over and if so shuffles the cards ready for the player to start again.

The draw function puts some messages on the screen and if appropriate calls the draw method for each of the cards to display them on the screen.

You may have noticed that there are no global variables that are updated in any of the functions. The class instances do act like global variables, but because they are updated using the methods for the classes means it is less likely to create obscure bugs compared to updating global variables directly.

This completes the game. There is plenty of scope for improving the game. You could improve the look of the game by having different card patterns available or change the difficulty by changing the number of cards or the length of time to play the game. You could also look changing the game into a two-player game, or instead of playing against the clock, have the player compete against the computer. Using the object-oriented techniques is likely easier than if it was done using procedural coding style.

Summary

Object-oriented programming is an alternative to procedural programming closely associating the data with the methods to work on them. This is particularly useful for code reuse and to help organize the structure of the program as the amount of code increases. This chapter has explained some of the key concepts of object-oriented programming and how they can be implemented in Python. It includes a game which demonstrates how to implement many of those concepts.

The next chapter will look at adding artificial intelligence to games to create a computer-based competitor.

CHAPTER 10

Artificial Intelligence

Artificial intelligence (AI) in computer games is programming to make the computer behave as though it is intelligent. Typically, this may be showing intelligence behind a character or object that is controlled by the computer.

This is not normally the same as machine learning which is what people often associated with artificial intelligence. Machine learning is a type of artificial associated with other systems such as speech recognition or pattern recognition.

In a computer game, artificial intelligence could be as simple as a pre-determined route that the enemy takes, or it could include some complicated algorithm that tracks the players' movement and responds in a lifelike way. To work well, it needs to be set at an appropriate level of difficulty for the player. The problem with machine learning is that if you use it to create an opponent, then it may become unbeatable rather than just challenging. Machine learning may be more suitable for creating realistic backgrounds or special effects.

When I refer to artificial intelligence, I'm really looking at algorithms that can be used to create a computer player at the appropriate level. This chapter will look at some examples of simple artificial intelligence that can be applied to games, with some theory around how to make a computer player as well as some code examples.

© Stewart Watkiss 2020
S. Watkiss, *Beginning Game Programming with Pygame Zero*,
https://doi.org/10.1007/978-1-4842-5650-3_10

Memory Game with AI

The memory game from Chapter 9 is currently a case of trying to beat the clock. This gives a little challenge but is not the same as playing against an opponent. Instead it is possible to create an artificial intelligence player to play against. To design the AI player, think about how people normally play the game, what the challenges are, and what strategies they use to win. As its name suggests, the challenge in the game is memory. If you can memorize all the cards as they are turned over, then the chance of winning the game is greatly increased. There is also an element of luck which we can be factored into the AI. I will also show the AI can be adjusted to create different difficulties.

In my first attempt at rewriting the code, I added and changed code in the existing files. As this developed, the code became long and confusing, a classic case of a bad smell. To fix this, I refactored the code, adding new classes to simplify the program. As the code increases, then there are multiple different classes, and it becomes harder to keep track of the files. One way to make this easier to understand is to create a diagram that shows the relationship between the classes. To do this, I created a UML class diagram which is shown in Figure 10-1. The diagram is only an approximation as it would be overcrowded to include all the attributes and methods. It also shows the top-level memory.py file as a class, which isn't correct. Despite not being a "pure" UML file, it is useful for showing how the program works.

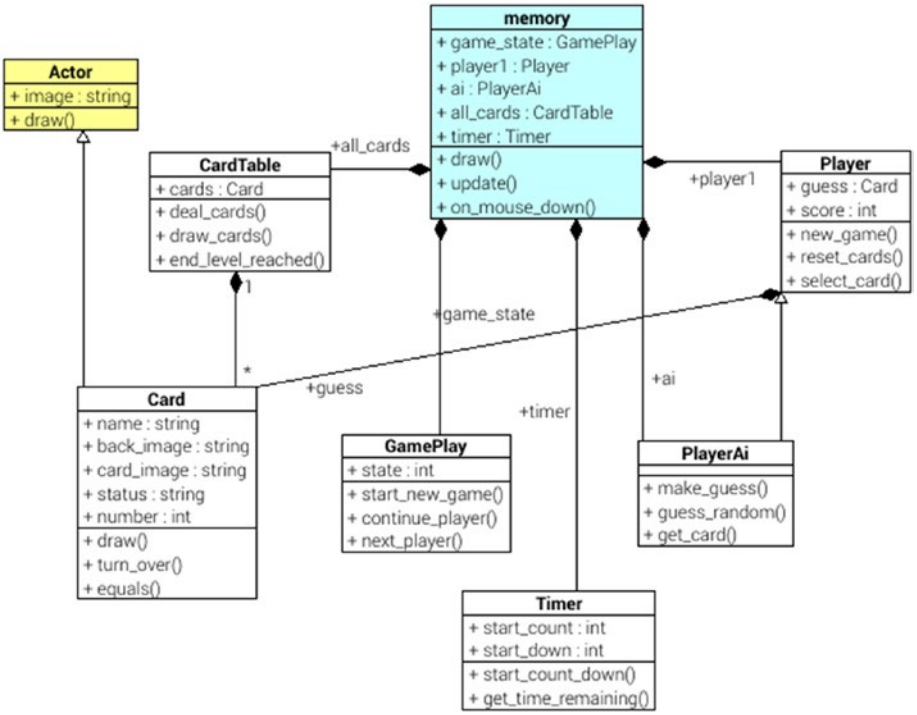


Figure 10-1. *UML class diagram for the memory game*

The lines and arrows on the diagram show the relationships between the classes. The triangular arrowheads represent inheritance where a child inherits attributes and methods from a parent. The filled diamonds show a composition relationship, also known as a "has a" relationship as the class has one or more instances of the class. Most of the compositions are 1 to 1 (the numbers have been left off for simplicity), but the CardTable contains multiple instances of the cards in a 1 to many relationship (1:∗).

The code files are included in the source code in a directory called memorygame2. The first class is the Card class. This class has only had a few changes from the previous version. The code is shown in Listing 10-1.

Listing 10-1. Card class for AI memory game

```
from pgzero.actor import Actor

# Card is based on an Actor (uses inheritance)
class Card(Actor):

    def __init__(self, name, back_image, card_image):
        Actor.__init__(self, back_image, (0,0))
        self.name = name
        self.back_image = back_image
        self.card_image = card_image
        # Status can be 'back' (turned over) 'front' (turned
          up) or 'hidden' (already used)
        self.status = 'back'
        # Number is unique number based on position
        # count left to right, top to bottom
        # updated after dealt
        self.number = None

    # Override Actor.draw
    def draw(self):
        if (self.status == 'hidden'):
            return
        Actor.draw(self)

    def turn_over(self):
        if (self.status == 'back'):
            self.status = 'front'
            self.image = self.card_image
        elif (self.status == 'front'):
            self.status = 'back'
            self.image = self.back_image
        # Attempt to turn over a hidden card - ignore
```

```
    else:
        return

def hide(self):
    self.status = 'hidden'

# When unhide set it to back image
def unhide (self):
    self.status = 'back'
    self.image = self.back_image

def is_hidden (self):
    if self.status == 'hidden':
        return True
    return False

# Is it turned to face forward
def is_faceup (self):
    if self.status == 'front':
        return True
    return False

# Is it turned to face down
def is_facedown (self):
    if self.status == 'back':
        return True
    return False

def reset (self):
    self.unhide()

def set_position(self, x, y):
    self.x = x
    self.y = y
```

```
    def equals (self, othercard):
        if self.name == othercard.name:
            return True
        return False
```

A new class is the CardTable class which has been created to simplify some of the code from the memory.py file. It contains the list of all the cards. It also includes methods to set up the table, deal the cards, and then draw them all on the screen. There is a method that returns all the cards which are face down, which is needed for the AI to know which cards it can pick from. There is also a method to test to see if the end of the level is reached (all cards are successfully paired).

The code for the CardTable class is shown in Listing 10-2.

Listing 10-2. CardTable class file

```
import random
from card import Card

class CardTable:

    def __init__ (self, card_back, cards_available):
        self.cards = []
        # Create individual card objects, two per image
        for key in cards_available.keys():
            # Add to list of cards
            self.cards.append(Card(key, card_back, cards_
            available[key]))
            # Add again (to have 2 cards for each img)
            self.cards.append(Card(key, card_back, cards_
            available[key]))

    def draw_cards(self):
        for this_card in self.cards:
```

```
        this_card.draw()

# Set the table settings
def setup_table(self, card_start_x, card_start_y, num_
cards_per_row, x_distance_between_cards, y_distance_
between_cards):
    self.card_start_x = card_start_x
    self.card_start_y = card_start_y
    self.num_cards_per_row = num_cards_per_row
    self.x_distance_between_cards = x_distance_between_cards
    self.y_distance_between_cards = y_distance_between_cards

# Returns all cards that are face down as Card objects
def cards_face_down(self):
    selected_cards = []
    for this_card in self.cards:
        if (this_card.is_facedown()):
            selected_cards.append(this_card)
    return selected_cards

# Shuffle the cards and update their positions
def deal_cards(self):
    # Create a temporary list of card indexes that is then
      shuffled
    keys = []
    for i in range (len(self.cards)):
        keys.append(i)
    random.shuffle(keys)

    # Setup card positions
    xpos = self.card_start_x
    ypos = self.card_start_y
    cards_on_row = 0
```

```
        # Give each card number based on position
        # count left to right, top to bottom
        card_number = 0
        for key in keys:
            # Reset (ie. unhide if hidden and display back)
            self.cards[key].reset()
            self.cards[key].number = card_number
            self.cards[key].set_position(xpos,ypos)
            xpos += self.x_distance_between_cards

            cards_on_row += 1
            # If reached end of row - move to next
            if (cards_on_row >= self.num_cards_per_row):
                cards_on_row = 0
                xpos = self.card_start_x
                ypos += self.y_distance_between_cards
            card_number += 1

    # If reach end of level
    def end_level_reached(self):
        for card in self.cards:
            if (not card.is_hidden()):
                return False
        return True

    def check_card_clicked (self, pos):
        for this_card in self.cards:
            # If not facedown then skip
            if (not this_card.is_facedown()):
                continue
            if (this_card.collidepoint(pos)):
                return this_card
        return None
```

The GamePlay class is a simplified version of the previous GamePlay class. The score attribute has been removed as that is now handled by the Player classes to provide a score for each of the players. There are additional state attributes and methods to handle the second player. The code for GamePlay class is included in Listing 10-3.

Listing 10-3. GamePlay class file

```
# State is tracked as a number, but to make the code readable
  constants are used
STATE_NEW = 0               # Game ready to start, but not running
STATE_PLAYER1_START = 10    # Player 1 to turn over card
STATE_PLAYER1_CARDS_1 = 11 # Card 1 turned over
STATE_PLAYER1_CARDS_2 = 12 # Card 2 turned over
STATE_PLAYER2_START = 20    # Player 2 starts go
STATE_PLAYER2_WAIT = 21     # Delay before Card 1 turned over
STATE_PLAYER2_CARDS_1 = 22 # Card 1 turned over
STATE_PLAYER2_CARDS_2 = 23 # Card 2 turned over
STATE_END = 50

# Number of seconds to display high score before allowing click
  to continue
TIME_DISPLAY_SCORE = 3

class GamePlay:

    def __init__ (self):
        self.state = STATE_NEW

    # If game has not yet started
    def is_new_game(self):
        if self.state == STATE_NEW:
            return True
        return False
```

```
    def is_game_over(self):
        if self.state == STATE_END:
            return True
        return False

    def is_player_1(self):
        if (self.state >= STATE_PLAYER1_START and self.state <=
        STATE_PLAYER1_CARDS_2):
            return True
        return False

    def is_player_2(self):
        if (self.state >= STATE_PLAYER2_START and self.state <=
        STATE_PLAYER2_CARDS_2):
            return True
        return False

    def is_player_2_start(self):
        if (self.state == STATE_PLAYER2_START):
            return True
        return False

    def is_player_2_wait(self):
        if (self.state == STATE_PLAYER2_WAIT):
            return True
        return False

    def is_player_2_card1(self):
        if (self.state == STATE_PLAYER2_CARDS_1):
            return True
        return False
```

```
def is_player_2_card2(self):
    if (self.state == STATE_PLAYER2_CARDS_2):
        return True
    return False

def set_player_2_wait(self):
    self.state = STATE_PLAYER2_WAIT

def set_player_2_card1(self):
    self.state = STATE_PLAYER2_CARDS_1

def set_player_2_card2(self):
    self.state = STATE_PLAYER2_CARDS_2

def start_game(self):
    self.state = STATE_PLAYER1_START

def set_game_over(self):
    # player gets to see high score
    self.state = STATE_END

def is_game_running(self):
    if (self.state >= STATE_PLAYER1_START and self.state <
    STATE_END):
        return True
    return False

# Continue with current player (matched correctly)
def continue_player (self):
    if self.state <= STATE_PLAYER1_CARDS_2:
        self.state = STATE_PLAYER1_START
    else:
        self.state = STATE_PLAYER2_START
```

```
    # Switch to next player (not matched)
    def next_player (self):
        if self.state <= STATE_PLAYER1_CARDS_2:
            self.state = STATE_PLAYER2_START
        else:
            self.state = STATE_PLAYER1_START

    def set_new_game(self):
        self.state = STATE_NEW

    def is_pair_turned_over(self):
        if (self.state == STATE_PLAYER1_CARDS_2):
            return True
        return False

    # If a card is clicked then update the state accordingly
    def card_clicked(self):
        if (self.state == STATE_PLAYER1_START):
            self.state = STATE_PLAYER1_CARDS_1
        elif (self.state == STATE_PLAYER1_CARDS_1):
            self.state = STATE_PLAYER1_CARDS_2
```

The Timer class is the same as previously, but it is used in a different way. Instead of being used as a timer for the player to play against, it's used to add a delay for the AI player so that the human player can see the cards that the computer was turning over. The Timer class is shown in Listing 10-4.

Listing 10-4. Timer class file

```
import math
import time

class Timer():

    def __init__(self, start_count):
```

```
        self.start_count = start_count
        self.start_time = time.time()

    # start count down, with optional parameter to replace the
      start_count value
    # -1 is used as a "magic number", this method should only
      be called with positive number
    # if it isn't given a number then -1 indicates no new time give
    def start_count_down(self, new_time = -1):
        if (new_time >= 0):
            self.start_count = new_time
        self.start_time = time.time()

    def get_time_remaining(self):
        current_time = self.start_count + self.start_time -
        time.time()
        if (current_time <= 0):
            return 0
        return math.ceil(current_time)
```

The Player class was considered in the earlier version but was not necessary at the time. With the addition of the AI, it was more useful to have a separate class for the player. This is a simple class which holds the score for the player and the card selection. It uses the card class having an instance of the card passed during the `select_card` method and then returning it using the `get_card` method. This is the reason for the composition between the Player and the Card class shown in Figure 10-1. The code for the Player class is shown in Listing 10-5.

Listing 10-5. Player class file

```
from card import Card

class Player():

    def __init__ (self):
        # Track which cards are turned over
        self.guess = [None, None]
        self.score = 0

    def score_point (self):
        self.score += 1

    # Returns a single card object - either 0 or 1
    def get_card (self, card_number):
        return self.guess[card_number]

    # Reset cards held in hand, but does not hide / turn_over card
    def reset_cards(self):
        self.guess[0] = None
        self.guess[1] = None

    def select_card(self, card):
        if (self.guess[0] == None):
            self.guess[0] = card
        else:
            self.guess[1] = card

    # Returns the number of cards that are selected
    def num_cards_selected(self):
        if (self.guess[0] == None):
            return 0
```

```
        elif (self.guess[1] == None):
            return 1
        else:
            return 2
```

The last of the class files contains the PlayerAi class which inherits from the Player class adding the ability for the AI player to make a random guess. This is a very basic form of AI which will be expanded on later. The code is included in Listing 10-6.

Listing 10-6. Player class file

```
import random
from player import Player

class PlayerAi (Player):

    def __init__(self):
        Player.__init__(self)

    def make_guess(self, available_cards):
        self.guess_random(available_cards)

    def guess_random (self, available_cards):
        this_guess = random.choice(available_cards)
        this_guess.turn_over()
        self.select_card(this_guess)

    def get_card (self, card_number):
        return self.guess[card_number]
```

Finally, the memory.py file has been updated. The user interaction is still handled within the `on_mouse_down` function, but it now includes the AI player in the update function. Each time that the AI player performs an operation, there is a delay triggered by `timer.start_count_down`, which effectively

pauses the AI from any operations until `timer.get_time_remaining` shows that the time has been exceeded. This is shown in Listing 10-7.

Listing 10-7. The main memory.py file with basic AI

```
# Memory Card Game - PyGame Zero
import random

from card import Card
from gameplay import GamePlay
from player import Player
from playerai import PlayerAi
from timer import Timer
from cardtable import CardTable

# These constants are used to simplify the game
# For more flexibility these could be replaced with
  configurable variables
# (eg. different number of cards for different difficulty
  levels)
NUM_CARDS_PER_ROW = 4
X_DISTANCE_BETWEEN_CARDS = 120
Y_DISTANCE_BETWEEN_CARDS = 120
CARD_START_X = 220
CARD_START_Y = 130

TITLE = "Lake District Memory Game"
WIDTH = 800
HEIGHT = 600

cards_available = {
    'airafalls' : 'memorycard_airafalls',
    'ambleside' : 'memorycard_ambleside',
    'bridgehouse' : 'memorycard_bridgehouse',
```

```
    'derwentwater' : 'memorycard_derwentwater',
    'ravenglassrailway' : 'memorycard_ravenglassrailway',
    'ullswater' : 'memorycard_ullswater',
    'weatherstone' : 'memorycard_weatherstone',
    'windermere' : 'memorycard_windermere'
    }

card_back = "memorycard_back"

## Setup instance variables
game_state = GamePlay()
player1 = Player()
ai = PlayerAi()
# Timer is used for AI thinking time
timer = Timer(2)
all_cards = CardTable(card_back, cards_available)
all_cards.setup_table(CARD_START_X, CARD_START_Y, NUM_CARDS_
PER_ROW, X_DISTANCE_BETWEEN_CARDS, Y_DISTANCE_BETWEEN_CARDS)
all_cards.deal_cards()

def update():
    if (game_state.is_player_2_start()):
        timer.start_count_down()
        game_state.set_player_2_wait()
    if (game_state.is_player_2_wait()):
        if (timer.get_time_remaining() <= 0):
            ai.make_guess(all_cards.cards_face_down())
            timer.start_count_down()
            game_state.set_player_2_card1()
    # card 1 turned
    elif (game_state.is_player_2_card1()):
        if (timer.get_time_remaining() <= 0):
            ai.make_guess(all_cards.cards_face_down())
```

```
            timer.start_count_down()
            game_state.set_player_2_card2()
    # Card 2 selected - wait then check if matches
    elif (game_state.is_player_2_card2()):
        if (timer.get_time_remaining() <= 0):
                if ai.get_card(0).equals(ai.get_card(1)):
                    # If match add points and hide the cards
                    ai.score_point()
                    ai.get_card(0).hide()
                    ai.get_card(1).hide()
                    ai.reset_cards()
                    # Game Over
                    if (all_cards.end_level_reached()):
                        game_state.set_game_over()
                    # If user guess correct then they get
                      another attempt
                    else:
                        game_state.continue_player()
                # If not match then turn both around
                else:
                    ai.get_card(0).turn_over()
                    ai.get_card(1).turn_over()
                    ai.reset_cards()
                    game_state.next_player()

# Mouse clicked
def on_mouse_down(pos, button):
    # Only interested in the left button
    if (not button == mouse.LEFT):
        return
    # If new game then this click is to start the game
    if (game_state.is_new_game() or game_state.is_game_over()):
```

```
    game_state.start_game()
    all_cards.deal_cards()
    player1.score = 0
    ai.score = 0
    return

## Reach here then we are in game play
# Is it player1's turn
if (game_state.is_player_1()):
    # Check for both already clicked and this is a click to
      test
    if (game_state.is_pair_turned_over()):
        if (player1.get_card(0).equals(player1.get_card(1))):
            # If match add points and hide the cards
            player1.score_point()
            player1.get_card(0).hide()
            player1.get_card(1).hide()
            player1.reset_cards()
            # End of game
            if (all_cards.end_level_reached()):
                game_state.set_game_over()
            # If user guess correct then they get another
              attempt
            else:
                game_state.continue_player()
        # If not match then turn both around
        else:
            player1.get_card(0).turn_over()
            player1.get_card(1).turn_over()
            player1.reset_cards()
            game_state.next_player()
        return
```

```
        # Check if clicked on a card
        card_clicked = all_cards.check_card_clicked(pos)
        if (card_clicked != None):
            card_clicked.turn_over()
            player1.select_card(card_clicked)
            # Update state
            game_state.card_clicked()

def draw():
    screen.fill((220, 220, 220))
    if (game_state.is_new_game()):
        screen.draw.text("Click mouse to start", fontsize=60,
        center=(WIDTH/2,HEIGHT/2), shadow=(1,1),
        color=(255,255,255), scolor="#202020")
    if (game_state.is_game_over()):
        screen.draw.text("Game Over\nPlayer 1 score:
        "+str(player1.score)+"\nPlayer 2 (AI) score: "+str(ai.
        score), fontsize=60, center=(WIDTH/2,HEIGHT/2),
        shadow=(1,1), color=(255,255,255), scolor="#202020")
    if (game_state.is_game_running()):
        # Set colors based on which player is selected
        if (game_state.is_player_1()):
            player1_color = (0,0,0)
            player2_color = (128,128,128)
        else:
            player1_color = (128,128,128)
            player2_color = (0,0,0)
        all_cards.draw_cards()
        screen.draw.text("Player 1: "+str(player1.score),
        fontsize=40, bottomleft=(50,50), color=player1_color)
        screen.draw.text("Player 2 (AI): "+str(ai.score),
        fontsize=40, bottomleft=(550,50), color=player2_color)
```

```
        # Display computer status during ai turns
        if (game_state.is_player_2_wait() or game_state.is_
        player_2_card1()):
            screen.draw.text("Thinking which card to
            pick", fontsize=40, center=(WIDTH/2,HEIGHT/2),
            shadow=(1,1), color=(255,255,255),
            scolor="#202020")
```

This can be run and played but is extremely easy to beat. The game just chooses a random card each time, so until there are only a few cards left, the probability of them getting a match is very small.

A Good Memory

To make the game more challenging, we can have the computer remember the guesses that are made. As the player code is separated from the rest of the code, there are only two files that need to be updated. These are the files containing the Player class and the PlayerAi class. The two modified files will be listed here, but the complete source code is included in the memory3 directory.

First there needs to be somewhere to store the cards that have been seen. If the cards seen are stored in the PlayerAi class, then it will only see the cards that the AI player turns over. If the cards are instead stored in the Player class, then it's possible to store all the cards that are turned over by the human player as well as the AI player.

Saving the list as a class variable instead of an instance variable will make it visible to all instances, including all instances of child classes. This is done by placing the variables at the top of the class as shown here:

```
class Player():
    card_memory = {}
    click_order = []
```

There are two variables created here: `card_memory` is a dictionary to hold the card with an index of the card's name and `click_order` is a list that remembers the order that the cards are clicked. The second is not actually required at the moment, but adding it now will simplify the next stage.

To update the class variables whenever a card is revealed needs the following to be added to the `select_card` method:

```
Player.card_memory[card.number] = card
```

As this method is inherited by PlayerAi, then it will be called each time the human player or the computer player turns a card over. The variables also need to be reset at the start of a new game which is implemented in a static method `reset_cards`. The updated player.py file is shown in Listing 10-8.

Listing 10-8. Updated Player class to add improved AI

```
from card import Card

class Player():

    # Index of cards that ai remembers
    # Stored as dictionary as cards will be missing or be forgotten
    card_memory = {}
    click_order = []

    def __init__ (self):
        # Track which cards are turned over
        self.guess = [None, None]
        self.score = 0

    @staticmethod
    def new_game():
        Player.card_memory = {}
        Player.click_order = []
```

```
    def score_point (self):
        self.score += 1

    # Returns a single card object - either 0 or 1
    def get_card (self, card_number):
        return self.guess[card_number]

    # Reset cards held in hand, but does not hide / turn_over card
    def reset_cards(self):
        self.guess[0] = None
        self.guess[1] = None

    def select_card(self, card):
        if (self.guess[0] == None):
            self.guess[0] = card
        else:
            self.guess[1] = card
        Player.card_memory[card.number] = card

    # Returns the number of cards that are selected
    def num_cards_selected(self):
        if (self.guess[0] == None):
            return 0
        elif (self.guess[1] == None):
            return 1
        else:
            return 2
```

There are three different methods added to the PlayerAi class showing different ways that improved memory can be implemented. The updated source code is shown in Listing 10-9. Each of the new methods is explained later.

Listing 10-9. Updated PlayerAi class to add improved AI

```
import random
from player import Player

class PlayerAi (Player):

    def __init__(self):
        Player.__init__(self)

    def make_guess(self, available_cards):
        #self.guess_random(available_cards)
        #self.guess_remember_all(available_cards)
        #self.guess_remember_sometimes(available_cards)
        self.guess_remember_recent(available_cards)

    def guess_random (self, available_cards):
        this_guess = random.choice(available_cards)
        this_guess.turn_over()
        self.select_card(this_guess)

    def guess_remember_all (self, available_cards):
        # If first guess then use random
        if (self.guess[0] == None):
           self.guess_random(available_cards)
           return
        # Search to see if we have seen a matching card
        for search_card in Player.card_memory.values():
            # ignore if current card - or card has been hidden
            since
            if (search_card == self.guess[0] or search_card.
            is_hidden()):
                continue
            # Check to see if the card matches
```

```
            if (search_card.equals(self.guess[0])):
                search_card.turn_over()
                self.select_card(search_card)
                return
        # If not found the matching card then use random
        self.guess_random(available_cards)

    def guess_remember_sometimes (self, available_cards):
        # If first guess then use random
        if (self.guess[0] == None):
           self.guess_random(available_cards)
           return
        # Random whether make a proper guess or random guess
        if (random.randint(1,10) < 5):
            self.guess_random(available_cards)
            return
        # Search to see if we have seen a matching card
        for search_card in Player.card_memory.values():
            # ignore if current card - or card has been hidden
              since
            if (search_card == self.guess[0] or search_card.
            is_hidden()):
                continue
            # Check to see if the card matches
            if (search_card.equals(self.guess[0])):
                search_card.turn_over()
                self.select_card(search_card)
                return
        # If not found the matching card then use random
        self.guess_random(available_cards)

    def guess_remember_recent (self, available_cards):
```

```
        # If first guess then use random
        if (self.guess[0] == None):
           self.guess_random(available_cards)
           return
        # Get last 4 cards that were clicked
        # These are just card numbers
        recent_cards = Player.click_order[:-4]
        # Search to see if one of those is a matching card
        for search_card in Player.card_memory.values():
            # ignore if current card - or card has been hidden
              since
            if (search_card == self.guess[0] or search_card.
            is_hidden()):
                continue
            # ignore if not a recent card
            if (search_card.number not in recent_cards):
               continue
            # Check to see if the card matches
            if (search_card.equals(self.guess[0])):
                search_card.turn_over()
                self.select_card(search_card)
                return
        # If not found the matching card then use random
        self.guess_random(available_cards)

    def get_card (self, card_number):
        return self.guess[card_number]
```

The first of the new methods is `guess_remember_all`, which remembers every card that is turned over. The method starts by choosing a random card. If the corresponding pair has already been turned over, then it will turn over the corresponding card. This is handled by looking for the

card in `Player.card_memory.values` which returns the list of all values in the dictionary. The key parts of that method are listed as follows:

```
for search_card in Player.card_memory.values():
```

This is a for loop that cycles through all the values in the dictionary. The value is a Card object which is held in the variable `search_card`. It then checks for a match using

```
if (search_card.equals(self.guess[0])):
```

If it matches the previously turned over card, then it turns the matching card over using

```
search_card.turn_over()
self.select_card(search_card)
```

As this method remembers every card turned over, it is a very difficult level to beat. If you have a good memory or are very lucky with your choice of card, then it is possible to beat this, but it is frustratingly difficult.

The next method is called `guess_remember_sometimes`. As its name suggests, this remembers previous cards but only sometimes. This is based on a random check to determine whether to search for the card from memory or not. This is essentially the same as the `guess_remember_all` except for the following additional code:

```
if (random.randint(1,10) < 5):
    self.guess_random(available_cards)
    return
```

It creates a random number between 1 and 10. If the number is less than 5, then it performs a random guess. If the number is 5 or greater, then it searches for the card in the memory. The value to compare against (in this case 5) can be adjusted up and down to improve the probability of a successful guess.

This gives a reasonable level of difficulty, but it is not particularly realistic. The reason being is that human players are usually much better at remembering a card that was turned over recently compared to one that was turned over some time ago.

The final method is called `guess_remember_recent`. This provides the computer player with a short-term memory. All the cards that are turned over are still stored in the dictionary, but the computer only uses the most recent ones listed in the `Player.click_order` variable when checking for a match.

This is achieved by creating a separate list which only holds the last four entries of the `click_order` list.

```
recent_cards = Player.click_order[:-4]
```

Then when checking for a match, it uses the following to skip any cards that are not in the `recent_cards` list:

```
if (search_card.number not in recent_cards):
                continue
```

You can try adjusting the number of recent cards that the computer looks through to change the difficulty.

There are other things that you could do to make it appear more realistic. For example, you could combine these techniques to have a computer player which behaves more naturally by having a memory that is very good for recent cards but is randomly less likely to guess correctly as more cards are turned over. This has been left as an exercise for the reader.

If you look at the `make_guess` method, you can see that the different methods are all commented out except for the `guess_remember_recent` method. This provides a way for you to try out the different methods and compare them. Just remove the "`#`" character which is commenting out the one you want to test and comment out the others.

```
    def make_guess(self, available_cards):
        #self.guess_random(available_cards)
        #self.guess_remember_all(available_cards)
        #self.guess_remember_sometimes(available_cards)
        self.guess_remember_recent(available_cards)
```

One thing that you could do is to have the player choose the difficulty. Think about how you could add that. I've added another version of the memory game in the source code which includes that option. It's stored in the directory memory4; think about how you would add that first before looking at the supplied code.

Battleships

Another example of how you can create artificial intelligence can be seen in the game Battleships. This is the classic game you have almost certainly played at some stage. Originally a paper-based game where you had to try and sink your opponent's ships, this is now commonly played as a board game using model ships and plastic pegs to show when a ship has been hit or missed. This is shown in Figure 10-2.

Figure 10-2. *Traditional Battleship board game*

This game will be a computer version of this classic game which is used to demonstrate artificial intelligence. The intelligence involved in playing battleships is something that most of us do sub-consciously. It's true that a lot of the game is based on luck, but without a strategy, a non-intelligent computer version will almost certainly lose against a human opponent.

There are three main strategies that can be considered when playing battleships:

- Random – Playing each turn randomly is the most basic strategy. The odds of successfully hitting each of the opponent's ships are very low until many of the positions have already been tried.
- Random with ship awareness – This second strategy is where you fire shots randomly until successfully hitting an opponent's ship. Upon hitting the opponent's ship, you fire against adjacent positions until the ship is

sunk. After the ship is sunk, then you start again trying random positions.

- Probability analysis – This is the ultimate strategy where a computer opponent can work out the probability for the remaining ships being in a particular position.

In this game I have implemented the second strategy of random shots with ship awareness. The reason is that the first level is far too easy for most players and the third is likely to be too difficult to beat.

To keep the code short, the version listed in the book has fixed positions for the ships of both the human and computer players. This allows me to demonstrate the way that the computer player works without having to list a lot of additional code. I have however included a second version in the source code which is a complete game where the player gets to position their own ships and the computer chooses random positions for its ships. The version listed in this book is in the directory battleship, and the more complete version is in the directory battleship2.

There are six Python files included in this game, plus some images in the image folder. The game uses a similar object-oriented programming methodology to the memory game. The file battleship.py is the main executable file. There is a fleet for each player, with the fleet consisting of five ships. Each ship is a child of the Actor class. There is a grid class which handles the grid position and translates the position on the grid to the position on the screen. Finally, the Ai class is the one that is of most interest for this chapter as that is where the intelligence is coded.

I will give a quick run through of each of the files finishing with an explanation of the Ai class at the end.

The first file is the battleship.py main program file. This is shown in Listing 10-10.

Listing 10-10. The main battleship.py program file for Battleship game

```
from fleet import Fleet
from grid import Grid
from ai import Ai

WIDTH = 1024
HEIGHT  = 768

# Start of your grid (after labels)
YOUR_GRID_START = (94,180)
# Start of enemy grid
ENEMY_GRID_START = (544,180)
GRID_SIZE = (38,38)

player = "player1"

grid_img_1 = Actor ("grid", topleft=(50,150))
grid_img_2 = Actor ("grid", topleft=(500,150))

own_fleet = Fleet(YOUR_GRID_START, GRID_SIZE)
enemy_fleet = Fleet(ENEMY_GRID_START, GRID_SIZE)

## Manually position ships position random or allow
## player to choose.
own_fleet.add_ship("destroyer",(7,0),"horizontal")
own_fleet.add_ship("cruiser",(1,1),"horizontal")
own_fleet.add_ship("submarine",(1,4),"vertical")
own_fleet.add_ship("battleship",(4,5),"horizontal")
own_fleet.add_ship("carrier",(9,3),"vertical")

enemy_fleet.add_ship("destroyer",(5,8),"horizontal", True)
enemy_fleet.add_ship("cruiser",(3,4),"vertical", True)
enemy_fleet.add_ship("submarine",(4,1),"horizontal", True)
enemy_fleet.add_ship("battleship",(8,3),"vertical", True)
enemy_fleet.add_ship("carrier",(1,1),"vertical", True)
```

```
# Don't need a player1 object
# Player 2 represents the AI player
player2=Ai()

def draw():
    screen.fill((192,192,192))
    grid_img_1.draw()
    grid_img_2.draw()
    screen.draw.text("Battleships", fontsize=60,
    center=(WIDTH/2,50), shadow=(1,1), color=(255,255,255),
    scolor=(32,32,32))
    screen.draw.text("Your fleet", fontsize=40,
    topleft=(100,100), color=(255,255,255))
    screen.draw.text("The enemy fleet", fontsize=40,
    topleft=(550,100), color=(255,255,255))
    own_fleet.draw()
    enemy_fleet.draw()
    if (player == "gameover"):
        screen.draw.text("Game Over", fontsize=60,
        center=(WIDTH/2,HEIGHT/2), shadow=(1,1),
        color=(255,255,255), scolor=(32,32,32))

def update():
    global player
    if (player == "player2"):
        grid_pos = player2.fire_shot()
        result = own_fleet.fire(grid_pos)
        player2.fire_result (grid_pos, result)
        # If ship sunk then inform Ai player
        if (result == True):
            if (own_fleet.is_ship_sunk_grid_pos(grid_pos)):
                player2.ship_sunk(grid_pos)
```

```
                # As a ship is sunk - check to see if all ships
                  are sunk
                if own_fleet.all_sunk():
                    player = "gameover"
                    return

        # If reach here then not gameover, so switch back to
          main player
        player = "player1"

def on_mouse_down(pos, button):
    global player
    if (button != mouse.LEFT):
        return
    if (player == "player1"):
        if (enemy_fleet.grid.check_in_grid(pos)):
            grid_location = enemy_fleet.grid.get_grid_pos(pos)
            #print (Grid.grid_to_string(grid_location))
            enemy_fleet.fire(grid_location)
            if enemy_fleet.all_sunk():
                player = "gameover"
            else:
                # switch to player 2
                player = "player2"
```

This file imports some of the classes and creates the instances of the main classes. This includes two grids and corresponding fleets, one for the human player's fleet positions and the other for the computer player. The ships are added to the fleet through hard-coded positions for both the human and computer fleets. This is to reduce the amount of code at this stage. The draw function loops through the various objects and calls each of their draw methods, as well as displaying the status text.

There is a separation between the human and Ai code. The update function handles the computer player, whereas the on_mouse_down function handles the interaction with the human player.

The fleet class handles tracking the ships and the shots that are fired. It includes methods for testing to see if a ship is sunk (in which case, it is set to visible) and to test if the entire fleet is sunk which is the trigger for game over. The code for fleet.py is shown in Listing 10-11.

Listing 10-11. Fleet class for Battleship game

```
import math
from grid import Grid
from ship import Ship
from pgzero.actor import Actor

class Fleet:

    def __init__ (self, start_grid, grid_size):
        self.start_grid = start_grid
        self.grid_size = grid_size
        self.ships = []
        self.grid = Grid(start_grid, grid_size)
        self.shots = []

    # Is there a ship at this position that has sunk
    def is_ship_sunk_grid_pos (self, check_grid_pos):
        # find ship at that position
        for this_ship in self.ships:
            if (this_ship.includes_grid_pos(check_grid_pos)):
                return this_ship.is_sunk()
        # If there is no ship at this position then return False
        return False
```

```
    def add_ship (self, type, position, direction, hidden=False):
        self.ships.append(Ship(type, self.grid, position,
        direction, hidden))

    # check through ships to see if any still floating
    def all_sunk (self):
        for this_ship in self.ships:
            if not this_ship.is_sunk():
                return False
        return True

    # Draws entire fleet (each of the ships)
    def draw(self):
        for this_ship in self.ships:
            this_ship.draw()
        for this_shot in self.shots:
            this_shot.draw()

    def fire (self, pos):
        # Is this a hit
        for this_ship in self.ships:
            if (this_ship.fire(pos)):
                # Hit
                self.shots.append(Actor("hit",topleft=self.
                grid.grid_pos_to_screen_pos(pos)))
                #check if this ship sunk
                if this_ship.is_sunk():
                    # Ship sunk so make it visible
                    this_ship.hidden = False
                return True
        self.shots.append(Actor("miss",topleft=self.grid.grid_
        pos_to_screen_pos(pos)))
        return False
```

One of the main things that the fleet class provides is the list of all the ships belonging to that fleet. This is in the list `self.ships` and is created based on the Ship class. It also holds all the shots that have been fired as a list of Actors representing either a hit or miss.

The Ship class is shown in Listing 10-12. It is a child of the Actor class with some additional code to handle the placement of the ship on the appropriate grid and to handle when the ship is hidden or visible.

Listing 10-12. Ship class for Battleship game

```
from pgzero.actor import Actor
from grid import Grid

# Ship is referred to using an x,y position

class Ship (Actor):

    def __init__ (self, ship_type, grid, grid_pos, direction,
    hidden=False):
        Actor.__init__(self, ship_type, (10,10))
        self.ship_type = ship_type
        self.grid = grid
        self.image = ship_type
        self.grid_pos = grid_pos
       self.topleft = self.grid.grid_pos_to_screen_pos((grid_pos))
       # Set the actor anchor position to center of the first square
        self.anchor = (38/2, 38/2)
        self.direction = direction
        if (direction == 'vertical'):
            self.angle = -90
        self.hidden = hidden
        if (ship_type == "destroyer"):
            self.ship_size = 2
            self.hits = [False, False]
```

```
        elif (ship_type == "cruiser"):
            self.ship_size = 3
            self.hits = [False, False, False]
        elif (ship_type == "submarine"):
            self.ship_size = 3
            self.hits = [False, False, False]
        elif (ship_type == "battleship"):
            self.ship_size = 4
            self.hits = [False, False, False, False]
        elif (ship_type == "carrier"):
            self.ship_size = 5
            self.hits = [False, False, False, False, False]

    def draw(self):
        if (self.hidden):
            return
        Actor.draw(self)

    def is_sunk (self):
        if (False in self.hits):
            return False
        return True

    def fire (self, fire_grid_pos):
        if self.direction == 'horizontal':
            if (fire_grid_pos[0] >= self.grid_pos[0] and
                fire_grid_pos[0] < self.grid_pos[0]+self.ship_
                size and
                fire_grid_pos[1] == self.grid_pos[1]):
                self.hits[fire_grid_pos[0]-self.grid_pos[0]] =
                True
                return True
        else:
```

```
            if (fire_grid_pos[0] == self.grid_pos[0] and
                fire_grid_pos[1] >= self.grid_pos[1] and
               fire_grid_pos[1] < self.grid_pos[1]+self.ship_size):
               self.hits[fire_grid_pos[1]-self.grid_pos[1]] = True
                return True
        return False

    # Does this ship cover this grid_position
    def includes_grid_pos (self, check_grid_pos):
        # If first pos then return True
        if (self.grid_pos == check_grid_pos):
            return True
        # check x axis
        elif (self.direction == 'horizontal' and
            self.grid_pos[1] == check_grid_pos[1] and
            check_grid_pos[0] >= self.grid_pos[0] and
            check_grid_pos[0] < self.grid_pos[0] + self.ship_
            size):
            return True
        elif (self.direction == 'vertical' and
            self.grid_pos[0] == check_grid_pos[0] and
            check_grid_pos[1] >= self.grid_pos[1] and
            check_grid_pos[1] < self.grid_pos[1] + self.ship_
            size):
            return True
        else :
            return False
```

The Ship class uses a ship type to determine the size of the ship. This is based on the name of the ship, such as destroyer (two grid positions) or battleship (four grid positions). It also updates the anchor position. This has nothing to do with a nautical anchor used in a ship but instead relates

to the anchor position of the Pygame Zero Actor. By default, the anchor is the center of the image, but in this case, it is set as the center of the first grid position (top, left) that the ship occupies. This position is used for placement of the ship and its rotation. It makes it easier to position the ship on the grid, and so that when a ship is placed vertically, it is rotated within the grid column.

The constructor then creates a list corresponding to each of the grid positions called `self.hits`. The list is set to False for each of the positions, which are then updated to True whenever one of them is hit. If they are all set to True, then the ship is considered sunk. This can be tested using the `is_sunk` method.

The fire method determines whether the fire hits the ship by looking at whether its grid position matches any of the positions that the ship occupies and updates the status accordingly. The `includes_grid_position` method performs a similar check but is used to check whether a ship exists in that position and does not change its status.

The methods in the Fleet and Ship class use a grid position rather than the screen location. The Grid class is used to convert the screen position from the mouse click to the grid position on one of the two grids. It is used by both the Ship class and the `on_mouse_down` function in battleship.py. The Grid class is shown in Listing 10-13.

Listing 10-13. Grid class for Battleship game

```
import math

class Grid:
    # Grid dimensions are in terms of screen pixels
    def __init__ (self, start_grid, grid_size):
        self.start_grid = start_grid
        self.grid_size = grid_size
```

```
# Does co-ordinates match this grid - if so which screen_
  position
def check_in_grid (self, screen_pos):
    if (screen_pos[0] < self.start_grid[0] or
        screen_pos[1] < self.start_grid[1] or
        screen_pos[0] > self.start_grid[0] + (self.grid_
        size[0] * 10) or
        screen_pos[1] > self.start_grid[1] + (self.grid_
        size[1] * 10)):
            return False
    else:
        return True

def get_grid_pos (self, screen_pos):
    x_offset = screen_pos[0] - self.start_grid[0]
    x = math.floor(x_offset / self.grid_size[0])
    y_offset = screen_pos[1] - self.start_grid[1]
    y = math.floor(y_offset / self.grid_size[1])
    if (x < 0 or y < 0 or x > 9 or y > 9):
        return None
    return (x,y)

# Gets top left of a grid position - returns as screen position
def grid_pos_to_screen_pos (self, grid_pos):
    x = self.start_grid[0] + (grid_pos[0] * self.grid_size[0])
    y = self.start_grid[1] + (grid_pos[1] * self.grid_size[1])
    return (x,y)
```

This is handled using the start position of the grid, stored in `grid_pos`, and the size of each grid square, stored in `grid_size`. The floor method from the math module is used to round the values down to the nearest whole number.

The final class is the Ai class which is where the computer player is implemented. This is the key part for this chapter, so it will be explained in more detail. The code is shown in Listing 10-14.

Listing 10-14. Ai class for Battleship game

```
import random
from grid import Grid

# Provides Ai Player
class Ai:

    NA = 0
    MISS = 1
    HIT = 2

    def __init__ (self):
        # Create 2 dimension list with no shots fired
        # access using [x value][y value]
        # Pre-populate with NA
        self.shots = [ [Ai.NA for y in range(10)] for x in
        range(10) ]
        # Hit ship is the position of the first successful hit
          on a ship
        self.hit_ship = None

    def fire_shot(self):
        # If not targeting hit ship
        if (self.hit_ship == None):
            return (self.get_random())
        else:
            # Have scored a hit - so find neighboring positions
            # copy hit_ship into separate values to make easier
              to follow
```

```
            hit_x = self.hit_ship[0]
            hit_y = self.hit_ship[1]
            # Try horizontal if not at edge
            if (hit_x < 9):
                for x in range (hit_x+1,10):
                    if (self.shots[x][hit_y] == Ai.NA):
                        return (x,hit_y)
                    if (self.shots[x][hit_y] == Ai.MISS):
                        break
            if (hit_x > 0):
                for x in range (hit_x-1,-1, -1):
                    if (self.shots[x][hit_y] == Ai.NA):
                        return (x,hit_y)
                    if (self.shots[x][hit_y] == Ai.MISS):
                        break
            if (hit_y < 9):
                for y in range (hit_y+1,10):
                    if (self.shots[hit_x][y] == Ai.NA):
                        return (hit_x,y)
                    if (self.shots[hit_x][y] == Ai.MISS):
                        break
            if (hit_y > 0):
                for y in range (hit_y-1,-1, -1):
                    if (self.shots[hit_x][y] == Ai.NA):
                        return (hit_x,y)
                    if (self.shots[hit_x][y] == Ai.MISS):
                        break
            # Catch all - shouldn't get this, but just in case
              guess random
            return (self.get_random())

    def fire_result(self, grid_pos, result):
```

```
        x_pos = grid_pos[0]
        y_pos = grid_pos[1]
        if (result == True):
            result_value = Ai.HIT
            if (self.hit_ship == None):
                self.hit_ship = grid_pos
        else:
            result_value = Ai.MISS
        self.shots[x_pos][y_pos] = result_value

    def get_random(self):
        # Copy only non-used positions into a temporary list
        non_shots = []
        for x_pos in range (0,10):
            for y_pos in range (0,10):
                if self.shots[x_pos][y_pos] == Ai.NA:
                    non_shots.append((x_pos,y_pos))
        return random.choice(non_shots)

    # Let Ai know that the last shot sunk a ship
    # list_pos is provided, but not currently used
    def ship_sunk(self, grid_pos):
        # reset hit ship
        self.hit_ship = None
```

After the import and the class definition, there are three class variables called NA, MISS, and HIT. These are used as constants and just make the rest of the code easier to understand. Reading the code, it's easier to understand that Ai.MISS represents a miss than just using the number 1, the same for NA (no shot fired at that position) and HIT.

After that, there is the usual constructor __init__ which has an entry

```
self.shots = [ [Ai.NA for y in range(10)] for x in range(10) ]
```

This is a way of creating a 2D list and pre-populating it with Ai.NA. This will end up with a list which looks like this:

```
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

This has an entry for every position in the grid which can be accessed using `self.shots[x-pos][y-pos]`. If you look at the grid in the preceding format, then the x and y axis are switched (y goes across and x goes down), but that is only how it is represented in the print listing. The important thing is how it is accessed using x, y positions.

The other variable created in the constructor is `self.hit_ship`. This will keep track of the position of the last time a shot successfully hit a target. It is reset to None when the ship is sunk.

When a shot is fired, there are multiple stages:

1. The fire_shot method is called which works out the next "guess" for where to fire the shot. The Ai class does not know if that shot is successful or not at this stage.

2. battleship.py then calls the fire method from the fleet class, which adds the appropriate hit or miss Actor and returns a True or False to indicate whether the shot hit a target or not.

3. The fire_result method is called which allows the Ai class to update the shots list to know whether the shot resulted in a hit.

4. If the ship has been sunk, then the ship_sunk method is called so that the Ai class knows that it doesn't need to keep targeting that ship.

The reason for needing to do this in multiple stages is because the Ai cannot see where the ships of the enemy are located. As a result, it does not know whether its shots were successful or not.

The first thing that the `fire_shot` does is to look to see if it knows the location of a ship that is not yet sunk. It does that by looking at whether `self.hit_ship` is set to None. If it does not know the location of an enemy ship, then it takes a random guess using the `get_random` method which is shown as follows:

```
def get_random(self):
    # Copy only non-used positions into a temporary list
    non_shots = []
    for x_pos in range (0,10):
        for y_pos in range (0,10):
            if self.shots[x_pos][y_pos] == Ai.NA:
                non_shots.append((x_pos,y_pos))
    return random.choice(non_shots)
```

This uses the `random.choice` method to choose from available positions. Before it can call that, it needs a list showing only the shots that have not already been tried, which is what the rest of that code does. It creates a `non_shots` list, and then using a nested for loop checks all the grid locations and adds any grid positions to the `non_shots` list that have not already been tried. The grid position is then returned to `fire_shot` which in turn uses it that location as its return value.

If there is already a ship that was hit recently, but which has not been sunk, then there will be a position in `self.hit_ship`. In that case the code tries four different directions until it finds a suitable grid position to try next. A suitable position is any location that it has not been tried and is adjacent to a successful shot. This can be seen in the following excerpt from the code:

```
if (hit_x < 9):
    for x in range (hit_x+1,10):
        if (self.shots[x][hit_y] == Ai.NA):
            return (x,hit_y)
        if (self.shots[x][hit_y] == Ai.MISS):
            break
```

If the x position of the hit_ship is less than 9 (not at the right-hand side of the grid), then it will loop across all positions to the right. If it comes across a position that has the value of NA, then that is a valid shot and so it returns that position. If instead it comes across a MISS, then it knows that the ship is not in that direction so it uses a break to exit from the for loop which moves the code on to check the next direction.

The other if statements do the same thing but looking in the other directions until a valid shot is found.

There is a final entry so that if none of the four directions apply, then it returns a random guess instead. This should never be called, as until the ship is sunk, there should always be a valid position to try. There are differences of opinions on whether adding "just in case" code is a good idea. My rationale is that if there is some situation I haven't thought of or perhaps a mistake in the code, then this will allow the game to continue without giving the user an error. The counter argument is that this could hide a problem with the code further up, where the game continues to run, but not in the way it was intended.

The `fire_result` method is called when the result of the shot is known. It updates the grid location with whether the hit was successful or not. It also updates the value of `hit_ship` if the shot was a hit and the value of `hit_ship` is currently set to None. The last method is `ship_sunk` which resets the value of `hit_ship` to None after the ship is successfully sunk.

This code implements the strategy quite well, but there are a couple of things that could be improved. One is that the Ai always tries the positions in the same order (horizontal and then vertical). If the player understands this, then they could gain an advantage by always placing ships away from the left and right edges and always vertically. This would only make a small difference but could be fixed by using a random decision on which direction to try first. It can also be tricked where there are two ships touching, where it hits one ship first but then sinks the second ship. It will not go back to finish off the first ship it hit. These do not stop the game from working but would be a good challenge for the reader to create an improved version.

As I warned in the beginning, graphical game programming uses lots of code. To get this far has needed 300 lines of code, but that hasn't included the ability for the user to place their own ships or for the computer to choose positions for its ships. This is something you may like to have a go at implementing yourself, or you can look in the folder battleship2 in the source code where I have created another version which implements that feature as well as some other improvements.

A screenshot of the final game is shown in Figure 10-3.

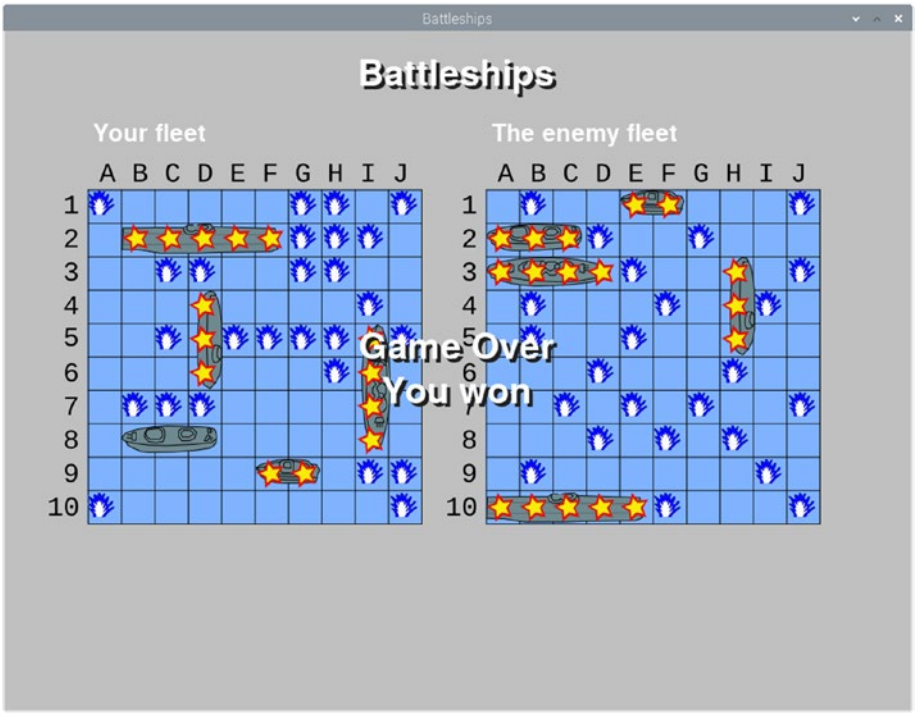


Figure 10-3. *Complete Battleship game*

Summary

This chapter has looked at ways that computers can be made to behave like a human player. In both the examples listed, the artificial intelligence has been created to mimic the same process that a human would go through when playing that game.

When designing some computer programs, you may be looking to make the computer as "clever" as possible. The problem the computer being too intelligent is that the computer can analyze the possible outcomes can making it too difficult to beat. When creating computer

games, it's important to think about the level of difficulty to make it challenging, but not too difficult.

There is scope to improve these games by creating different difficulty levels or by making the game appear more human-like. You may like to have a go at tweaking the AI or thinking of how you could add AI to other games.

CHAPTER 11

Improvements and Debugging

This final chapter will look at a few additional techniques for making improvements to your code. It will also provide some help with debugging when things go wrong. The final game will be a 2D top-down space shooter game. This should help give you the confidence to create your own games using the knowledge acquired from this book.

Additional Techniques

Throughout this book, there have been several different techniques introduced for creating games. Some of these are then widely used across multiple games, whereas others may only benefit a certain type of game. There are plenty of other things that can improve gameplay, make the game appear more professional, or save you time. I have added a few more here which can help improve the number of programming techniques you can use.

© Stewart Watkiss 2020
S. Watkiss, *Beginning Game Programming with Pygame Zero*,
https://doi.org/10.1007/978-1-4842-5650-3_11

More About Pygame Zero

The official documentation for Pygame Zero is available online at https://pygame-zero.readthedocs.io/en/stable/. The documentation is very useful when first learning Pygame Zero, but it is limited in what it provides.

There are some features that are not included in the official documentation. One example of this is the TITLE variable. This is like the WIDTH and HEIGHT variables which have been used to change the size of the window, but in this case, the TITLE replaces the title on the title bar of the game window. There is also an ICON option that can be used to add a thumbnail icon to the application on the task bar. Listing 11-1 shows both these in action on an example program.

Listing 11-1. Program with TITLE and ICON options

```
WIDTH = 400
HEIGHT = 200
TITLE = "My Game Title"
ICON = "spacecrafticon.png"
```

The file referred to in the ICON needs to be in the directory that the application is running in. This is normally the same directory as the executable file, but in the case of Mu, you may need to copy it to the mu_code directory. Ideally the icon should be a PNG file of 32 x 32 pixels in dimension.

Figure 11-1 shows the program running on a Raspberry Pi. The game on the left does not include the TITLE or ICON entries so has the default "Pygame Zero Game" and default icon. The one on the right is titled "My Game Title" and includes a spacecraft icon.

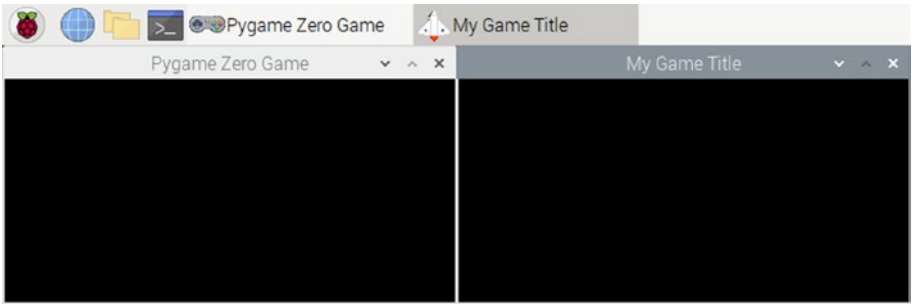


Figure 11-1. *Running programs with and without the TITLE and ICON entries*

If these are undocumented, then how do you find out about them? I think it's worth considering that Pygame Zero is still in an early stage and is being developed over time. It's possible that these features may have been added to the documentation by the time you read this. It may also be the case that there are more new features that have not yet made it into the documentation.

One way to find out about these features is to look at programs created by other people. That way, you can see what others have discovered. Another place is to look at the source code for Pygame Zero. The source code is on GitHub at `https://github.com/lordmauve/pgzero`. The source code is quite advanced code so it can be difficult to read for less experienced programmers. It can sometimes be useful when looking for something specific.

More About Pygame

In addition to Pygame Zero, you can make calls to methods in the parent library Pygame. This has already been used in the tank game in Chapter 7. An example is `pygame.draw.polygon` which made use of the Pygame libraries directly.

You can find more about Pygame at the official documentation at `www.pygame.org/docs/`.

Adding Fonts

In the previous games, the text shown on the screen has been using the default font. You can make use of other fonts by creating a `fonts` directory within your game directory and copying the fonts there. You can either copy existing fonts into that folder or add custom fonts without installing them on the system. On Linux (including the Raspberry Pi), you can normally use the system fonts by copying them from /usr/share/fonts/truetype. Alternatively, you can find many free fonts by searching on the Internet. Other systems, such as Windows, are likely to have copyright restrictions on many of the fonts. You should avoid any non-free fonts if you intend to share your games with others. You may also need to include the copyright information from the font when you distribute your game.

There is an anomaly with how fonts are installed, in that for Pygame Zero, the filename for the font must be all in lowercase. The font file must also be a True Type font ending with a .ttf extension. You will need to rename the font file when copying it into the fonts directory to remove any uppercase letters.

Once the file is in the font directory, you can use it by using the filename (without the .ttf extension). This is shown in the following code:

```
screen.draw.text("This is using the Deja Vu Sans Font",
fontname="dejavusans", fontsize=40, topleft=(30,30),
color=(255,255,255))
```

This uses the Deja Vu font which is available as standard on the Raspberry Pi or can be downloaded free from https://dejavu-fonts.github.io/.

Scrolling Screen

Several games can make use of a scrolling background. This is often used where a player stays stationary on the screen (or moves within the confines of the screen), but the background moves to make it appear that the player

is moving. This can be scrolling from side to side (often used in platform games where a player walks from left to right) or from top to bottom (used to show that a vehicle such as a plane is moving toward the top of the screen).

Depending upon the game, you may have a single background image which is repeated, or you may have multiple images which are designed to scroll from one to another. Typically, these images are the same size of the screen, but that's not necessary.

One way of creating this effect is to use `screen.blit` which has already been used for the background images in most of the games in this book. This is used to show an image on the screen. Using an offset in the image position will show the part of the image that overlaps with the screen. The diagram in Figure 11-2 shows how positioning two identical images with different offsets results in what appears to be a continuous image.

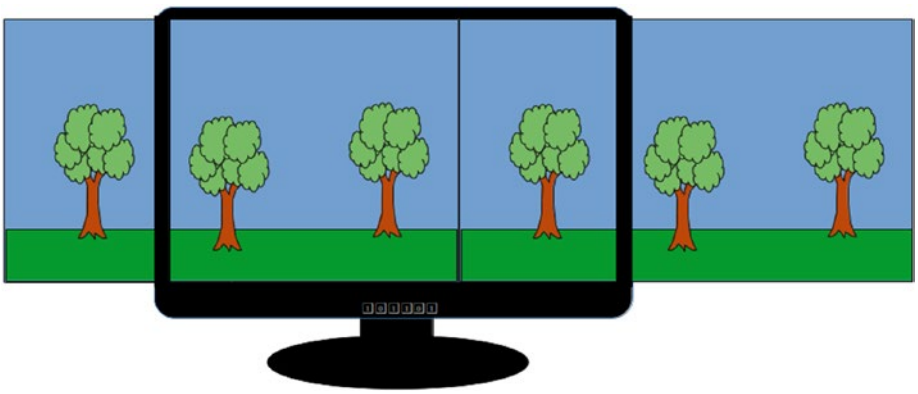


Figure 11-2. *Creating a scrolling screen*

The code in Listing 11-2 shows how this can be implemented. It uses a `scroll_position` which is the x position of the first image and the second image follows directly afterward, starting just off-screen. The `scroll_speed` can be adjusted to make the scroll go faster or slower as appropriate.

Listing 11-2. Creating a scrolling background

```
scroll_speed = 2
scroll_position = 0

def draw ():
    screen.blit("background_scroll", (scroll_position,0))
    screen.blit("background_scroll", (scroll_position+800,0))

def update():
    global scroll_position
    scroll_position -= scroll_speed
    if (scroll_position <= -800):
        scroll_position = 0
```

Reading from a CSV config file

Reading and writing to a file was covered in Chapter 4 when saving the high score for a game. In that case it was just a single entry. When there is more information stored, then the data needs to be stored in a way that the information can be easily retrieved.

There are many different file formats that can be used, each of which has its pros and cons. A simple format is to store the information as comma-separated values, known as a CSV file. In this format each line of the file holds multiple values which are separated by a comma, such as in the following line:

```
String value,1,2,3.1
```

In this example there is a string, followed by two integer numbers and one floating-point number. An important thing to note is that the numbers are stored as strings, so you cannot manipulate the values until they have been converted to numbers.

The first step is to split the file into separate components. As these are separated using a comma, you can use the `string.split` method which divides a string based on a character. In this case it would split the line based on the positions of the commas. What if there is a comma within the string? If there is a comma within the string, then the CSV format places quotes around the string to indicate that the comma within the quotes should not be split. In the following entry, the values are the same, but this time there is a comma in the string.

```
"String, value",1,2,3.1
```

It is now no longer possible to use the split method as that will ignore the quotes and result in the string being split into two values.

The solution is to use a module that knows how to handle a CSV file. Python includes the csv module which can do that. To demonstrate this, we need a CSV file. The file in Listing 11-3 is a simplified version of the enemies file that will be used in the space shooter game.

Listing 11-3. Sample CSV file for reading demo

```
0.3,asteroid,asteroid_sml,200,0,4
0.9,asteroid,asteroid_sml,100,0,4
0.9,asteroid,asteroid_med,400,0,3
1.2,asteroid,asteroid_sml,750,0,4
```

This is used to create asteroids that need to be dodged or destroyed. The first field is when the asteroid appears on the screen (in seconds), a keyword "asteroid" to indicate it is an asteroid, an image filename, the x and y coordinates, and finally the velocity of the asteroid (in pixels per time interval).

The file is saved as csvdemo.csv. The extension indicates it's a CSV file, but it can have a different extension. In the game it will be named enemies.dat to indicate it's a form of data file. The extension doesn't make any difference to how the file is handled in the program, but if you name the

file .csv, then it may be possible to open the file in a spreadsheet or similar application. This is something you probably don't want the players to be able to do.

The file has been created in a text editor. It is not possible to use Mu to edit the file as it only allows you to edit Python files, but there is a text editor app on the Raspberry Pi and Linux distributions, or using other operating systems the editor may be known as Notepad or TextEdit.

The source code for reading the file is shown in Listing 11-4.

Listing 11-4. Code to read in a CSV configuration file

```
import csv
import sys

configfile = "csvdemo.csv"

try:
    with open(configfile, 'r') as file:
        csv_reader = csv.reader(file)
        for enemy_details in csv_reader:
            start_time = float(enemy_details[0])
            # value 1 is type
            image = enemy_details[2]
            start_pos = (int(enemy_details[3]),
                int(enemy_details[4]))
            velocity = float(enemy_details[5])
            print ("Start time {}, Image {}, Start Pos {},
            Velocity {}".format(start_time, image, start_pos,
            velocity))
except IOError:
    print ("Error reading configuration file "+configfile)
    # Just end as cannot play without config file
    sys.exit()
```

```
except:
    print ("Corrupt configuration file "+configfile)
    sys.exit()
```

This is a standard Python 3 executable rather than being a Pygame Zero file. To run the code in Mu, you will need to change the mode. The code uses the `with` keyword before the file open command but is otherwise similar to the way that files were read before. When using the `with` keyword, there is no need to explicitly close the file. This can be useful in the event of a problem reading the file as closing it is handled automatically.

The entire operation is enclosed in a try except clause which will try and catch any errors. In this case there is nothing that can be done in the event of an error as the program cannot do anything without the data from the file. If the error is due to an IOError reading the file, then it gives a different error message than if the file is corrupt.

The CSV file is handled using the `csv.reader` which parses the file and places into a `csv_reader` which stores the data as a 2D list. Where numerical values are required, these are converted using int or float as appropriate. These will trigger an exception if the data is not in the correct format, so they are also included in the try clause.

Joysticks and Gamepads

The games so far have been designed to be played with the mouse or keyboard. The next step would be to add support for joysticks or gamepads. Unfortunately, Pygame Zero does not yet support gamepads, although it is listed on the roadmap as a potential future feature. Until that is added, it is possible to use a gamepad to emulate key presses using QJoyPad. This can be downloaded from http://qjoypad.sourceforge.net/. The gamepad would need to be configured on each computer it is used.

For a more authentic arcade gaming experience, the Picade or Picade console can provide a joystick that can act as though it is a keyboard.

Creating Arcade Games for Picade

If you want to get the full arcade game experience for your games, then the Picade is a compact arcade machine based around the Raspberry Pi. It is available as a console which needs to be connected to a TV or monitor or as a complete arcade cabinet with built-in screen.

A photo of the Picade arcade cabinet is shown in Figure 11-3.

Figure 11-3. *Pimoroni Picade running the space shooter game*

The Picade uses a HAT which is mounted on the Raspberry Pi. The HAT is then connected to a joystick and arcade buttons (switches) mounted on the top and side of the cabinet. The HAT translates the button presses into signals sent to the Raspberry Pi as though they were from a keyboard. You can also get the HAT separately and use that to create your own cabinet. An alternative is to use a different board that can emulate key presses such as a Makey Makey or Arduino.

The image in Figure 11-4 shows the keys associated with the joystick and each of the buttons on the Picade.

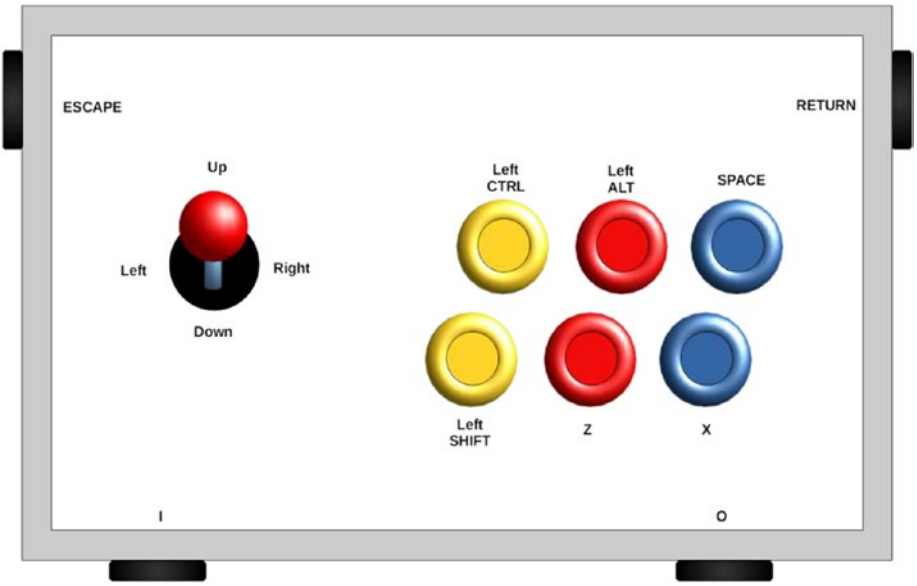


Figure 11-4. *Pimoroni Picade button layout to key presses*

Most of the games designed for a keyboard use the direction keys, which are mapped to the joystick on the Picade. The buttons are a bit more obscure, so to make the game playable on a standard keyboard and a Picade, then it can be useful to allow two different keys to be pressed to provide compatibility for both Picade and normal keyboard.

This is achieved using a boolean "or" in the check for the key press. The following code is taken from the tank game in Chapter 7, which allows either the keyboard space key or the Picade bottom yellow button (Left Shift) to be used to fire a shell.

```
if (keyboard.space or keyboard.lshift):
            game_state = 'start'
```

You may also want to consider making the key codes to be configurable through a config file.

Another thing to be aware of is that the Picade has a tradition 4:3 screen size, and while it can play games designed for a different screen size, the resolutions 800 x 600 and 1024 x 768 are a good choice.

Games designed for the Picade can be run on any Raspberry Pi using the keyboard instead of the joystick and buttons. The Picade usually runs RetroPie which is discussed next.

RetroPie

RetroPie provides a way of playing retro computer games on a Raspberry Pi. This can be a Picade or a regular Raspberry Pi. RetroPie is usually used for playing old computer games through emulators. It does not normally include any games by default due to potential copyright issues with commercial games.

As well as running emulator games, RetroPie can also run games created in Pygame or Pygame Zero. Adding support for your games to work in RetroPie could make it available to a wider audience. RetroPie can be downloaded and installed following the instructions at `https://retropie.org.uk/`.

RetroPie does not include Pygame Zero by default, but Pygame Zero can be installed using

```
sudo apt install python3-pgzero
```

You can add a new menu to install your own games. To add a new menu to the system, add the coding in Listing 11-5 to the file `/etc/emulationstation/es_systems.cfg` before the `</systemList>` entry.

Listing 11-5. Define new menu for RetroPie

```
<system>
  <name>pgzero</name>
  <fullname>Pygame Zero</fullname>
  <path>/home/pi/RetroPie/roms/pgzero</path>
  <extension>.sh</extension>
  <command>%ROM%</command>
  <theme>pgzero</theme>
</system>
```

There also needs to be an entry in the appropriate themes folder. There is a file included in the source code. This can be extracted by following these instructions:

```
cd ~
tar -xvzf pgzero-retro-theme.tgz
cd /etc/emulationstation/themes/carbon
sudo cp -r ~/retropietheme/* .
```

When installed, there will be a menu for Pygame Zero.

To install a game on RetroPie, create a folder in the roms directory which is usually ~/RetroPie/roms. In my case I created one called pgzero. In that directory, create a simple shell script to launch the program. The script file is shown in Listing 11-6.

Listing 11-6. Script file for launching the compass game ~/RetroPie/roms/pgzero/CompassGame.py

```
cd ~/compassgame
pgzrun compassgame.py
```

The script file also needs executable permission, which can be done using

```
chmod +x ~/RetroPie/roms/pgzero/CompassGame.py
```

The game can now be selected from the main menu. A screenshot of the menu is shown in Figure 11-5.

Figure 11-5. *Pygame Zero menu on RetroPie*

Debugging

When things go wrong in a program, it is known as a bug. Early bugs could include mechanical problems which included a dead moth that prevented a relay from closing. Nowadays, it usually refers to errors in computer code. This can be anything that negatively affects the way that the program runs compared to the way it is expected to behave. This can range from the program not running at all to a minor error where an actor may need to move two extra pixels before it's detected. It could also be a performance issue where a program runs slower than it should.

The details of how to test and debug programs could easily fill an entire book. This book will look at a few techniques relating to debugging and performance.

Error Messages

The first thing to check is to see if there are any error messages. In Mu these are normally displayed in a panel at the bottom of the screen. They will however be lost when you click stop. An alternative is to try running the program from the command line and see if you get an error message.

Sometimes the message given will be obvious and help you find the problem straight away. With some error messages, you may need to do some investigation. For example, a typical error message may include

```
KeyError: "No image found like 'batleship'. Are you sure the
image exists?"
```

The first thing to check is that the name matches an expected file. In this case there is a typo with the word battleship spelt incorrectly.

If the name was correct, then you should check to see if the file is in the correct directory. In the case of an image, it should be in the `images/` sub-directory. Also be aware that in some cases files are referred to from other places, which would be relative to the location of the program file, or in some cases in the directory, the program appears to run from (such as the `~/mu_code` directory).

Other errors may refer to syntax errors in your code. They will often give you the line number, but beware that the error may be earlier in the code than it says. For example, this is part of an invalid syntax error message:

```
  File "battleship.py", line 19
    grid_img_2 = Actor ("grid", topleft=(500,150))
               ^
SyntaxError: invalid syntax
```

The error appears to indicate that the problem is on line 19. However, looking at the code at lines 18 and 19 shows that the error is actually on line 18.

```
18. grid_img_1 = Actor ("grid", topleft=(50,150)
19. grid_img_2 = Actor ("grid", topleft=(500,150))
```

In this case there is a missing closing bracket on line 18. As a result of the missing bracket, the interpreter thinks that line 19 is a continuation of line 18 and that line 19 has the error. This is a common occurrence, so always make an effort to check to look for an error in the line prior to the one with the error.

Also don't forget to make sure that Mu is in Pygame Zero mode. If you get an error which says "NameError: name 'Actor' is not defined", then it maybe because you are trying to run in Python 3 mode instead.

Check for Variable Names

Another common problem is to mistype a variable name. If you try and store something into a different variable, then Python will just create that as a new variable. As a result, code that refers to the correct variable will not see the updates. Remember that variable names are case sensitive, so using the wrong case has the same effect.

You should also check that the variable is accessible in the current scope. If you try and update a variable that's not included in the globals, then it will create a local variable and not update the global variable.

Print Statements

A useful tool when trying to understand a program's behavior is the use of print commands. These can be used to display a message to the console while the program is running in the graphical display.

By adding a number of print commands, you can follow the status of the variables as the game progresses and see what happens.

IDE Debugging Tools

The Mu editor does include some basic debugging tools which can be an improvement on adding print statements. There is a debug mode within Mu (next to the play button). You can set breakpoints within the code by clicking the line number. The breakpoints are indicated by a red circle. From the menu, you can run to the breakpoints, or step over and some of the variables are shown in a new pane on the right-hand side.

As your programming progresses, you may want to look at a professional IDE (integrated development environment). Unfortunately, the setup for most IDEs with Pygame Zero is difficult, so you may want to stick with Mu for now, but it is something you may want to look at in the future.

Rubber Duck Debugging

Sometimes the program doesn't behave in the way you expect it to, and you can't see why. If this is the case, then it's useful to walk through how the program is supposed to work. A good way to do this is to talk out loud describing the way that it should work while stepping through the code. This can be done to an inanimate object such as a rubber duck. The idea is that while talking through the way that the code works, you may realize why it is not working as expected. It is surprising how effective this is. My favorite debug duck is shown in Figure 11-6, but you don't need to use a duck; any other object works just as well.

Figure 11-6. *Pycon UK debug duck*

Performance

One of the things about Python is that it is an interpreted language. This means that the text-based code you write is converted to code that the computer understands at runtime. This compares with compiled languages where this is done before the program is run. Generally interpreted languages are slower than if the program is compiled first which may contribute to performance issues.

The games that have been created so far in this book are quite short and so shouldn't result in performance issues, but as you increase the number of actors and resources being used, you may find the code starts to run slow.

Some of the code in this book has already made allowances for running at different speeds by checking the time since the update function last ran, but that may not be enough to stop the game from being unresponsive.

When writing code, the priority is normally about making the code as simple as possible and so it is easy to understand how it works. That helps limit the number of bugs and make it easier to maintain, but it may not result in the most efficient code.

There are steps that can be done to improve performance of a program. The first thing is to identify where the performance issues may be. Without understanding where the issue is, then resources may be wasted on optimizing code that is rarely used or where the computer is idle and will not notice the performance improvement. Normally you will need to look within loops that are called regularly during the running of the program.

The next thing is to make sure that you have some way of testing to see if your changes improve the performance. Sometimes changes made may sound like they will improve performance, but actually make it slower.

Here are a few suggestions on ways which may improve performance:

- If an existing Python library already exists, then use that (it's likely already been optimized).
- Check for loops that are consuming lots of resources.
- Avoid global variables.
- When in a function return once you are complete rather than continuing through code that is not necessary.
- Use code patterns (find code that others have created that has already considered performance).
- Redesign the algorithm.

These are just a few suggestions which may or may not improve performance. The last tip is vague and really depends upon what code you are creating. If you are doing something that others may have already done, such as sorting information, then look at what code others have created. It may be that some algorithms work better with a small amount of data rather than with a lot of data.

Space Shooter Game

The final game in this book is a space shooter top-down game. In this a spacecraft flies around shooting at obstacles that block its path. The game takes in a lot of the techniques that have been discussed throughout the book. It is designed for the Picade but can work equally well using keyboard controls. To fit in with the theme of an arcade machine, the game has an intentional retro feel including bitmap images, block font for the score, and tinny sound effects.

The design for the game simulates an asteroid field that the spacecraft must navigate around or blast its way through. The asteroids are referred to as enemies as a possible future addition would be to also include enemy spaceships which fly across the screen.

A screenshot of the game is shown in Figure 11-7.

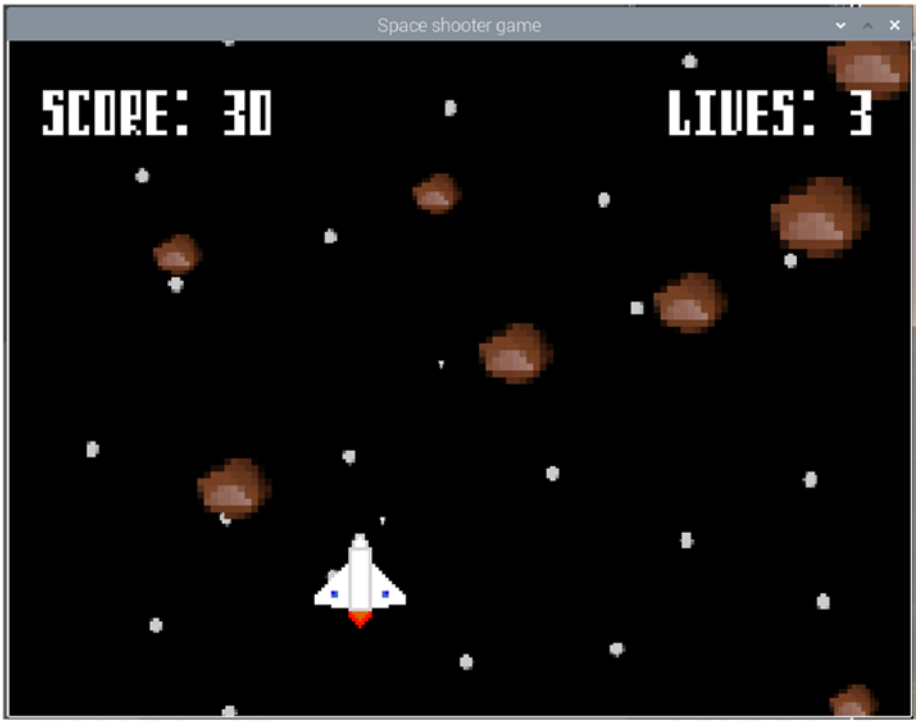


Figure 11-7. *Space shooter game*

The source code is split across multiple files for the different classes. The spaceship is defined as a subclass of the Actor class. This is shown in Listing 11-7.

Listing 11-7. Spaceship class in file spaceship.py

```
from pgzero.actor import Actor

class SpaceShip(Actor):

    def set_speed (self, movement_speed):
        self.movement_speed = movement_speed

    def move (self, direction):
        if (direction == "up"):
            self.y -= self.movement_speed
        elif (direction == "down"):
            self.y += self.movement_speed
        elif (direction == "left"):
            self.x -= self.movement_speed
        elif (direction == "right"):
            self.x += self.movement_speed
        # Make sure that the ship remains on the screen
        if self.x < 20:
            self.x = 20
        if self.x > 780:
            self.x = 780
        if self.y < 20:
            self.y = 20
        if self.y > 580:
            self.y = 580
```

This is essentially an object-oriented version of the code for the character in the compass game. This is simpler than the compass game

as the image doesn't change when the spacecraft moves. A possible improvement would be to add different images if you wanted the ship to look like it was banking over when moving to the side or for the flame to get bigger when it's accelerating forward.

The next file is the Asteroid class. This is a child of the Actor class handling the drawing of asteroids on the screen. This is shown in Listing 11-8.

Listing 11-8. Asteroid class in file asteroid.py

```
from pgzero.actor import Actor
import time
from constants import *

class Asteroid(Actor):

    def __init__ (self, screen_size, start_time, image, start_
    pos, velocity):
        Actor.__init__(self, image, (start_pos))
        self.screen_size = screen_size
        self.start_pos = start_pos
        self.start_time = start_time
        self.velocity = velocity
        self.status = STATUS_WAITING

    def update(self, level_time, time_interval):
        if self.status == STATUS_WAITING:
            # Check if time reached
            if (time.time() > level_time + self.start_time):
                # Reset to start position
                self.x = self.start_pos[0]
                self.y = self.start_pos[1]
                self.status = STATUS_VISIBLE
```

```
        elif self.status == STATUS_VISIBLE:
            self.y+=self.velocity * 60 * time_interval

    def reset(self):
        self.status = STATUS_WAITING

    def draw(self):
        if self.status == STATUS_VISIBLE:
            Actor.draw(self)

    def hit(self):
        self.status = STATUS_DESTROYED
```

This class extends the Actor class by adding a few variables and methods. The `start_time` is the time that the asteroid appears on the screen relative to the start of each level. The asteroids can have different images depending upon the size of the asteroid. The `start_pos` determines where the asteroid starts on the screen, and then the velocity is the speed that the asteroid moves toward the bottom of the screen, which is a measure of the number of pixels that the asteroid moves.

The update method handles when the asteroid becomes visible and moves the asteroid relative to its velocity. The reset method hides the asteroid. The hit method updates the status showing whether the asteroid has been destroyed. The draw method tests to see if the asteroid should be visible and if so then calls the parent draw method to show it on the screen.

There are several constants required which are stored in the constants.py file. This is so that they can be made available across multiple files and classes. This is shown in Listing 11-9.

Listing 11-9. Shared constants in file constants.py

```
# Status for each of the enemies
STATUS_WAITING = 0
STATUS_VISIBLE = 1
```

```
STATUS_DESTROYED = 2
STATUS_OFFSCREEN = 3

# Delay in seconds for messages on screen
DELAY_TIME = 2
```

This could be used similar to a system configuration file, but care needs to be taken when editing the file as it's a Python file and any errors could stop the program from running with an obscure error message.

The Asteroid class defines a single asteroid. The Enemies class provides a collection of Asteroids so that multiple instances can be handled at the same time. This is shown in Listing 11-10.

Listing 11-10. Enemies class in file enemies.py

```
import sys
import time
import csv
from constants import *
from pgzero.actor import Actor
from asteroid import Asteroid

# Enemies is anything that needs to be destroyed
# Could be an asteroid or an enemy fighter etc.

class Enemies:

    def __init__(self, screen_size, configfile):
        self.screen_size = screen_size
        self.asteroids = []
        # Time that this level started
        self.level_time = time.time()
        self.level_end = None
        # Load the config file
```

```
    try:
        with open(configfile, 'r') as file:
            csv_reader = csv.reader(file)
            for enemy_details in csv_reader:
                if enemy_details[1] == "end":
                   self.level_end = float(enemy_details[0])
                elif enemy_details[1] == "asteroid":
                    start_time = float(enemy_details[0])
                    # value 1 is type
                    image = enemy_details[2]
                    start_pos = (int(enemy_details[3]),
                        int(enemy_details[4]))
                    velocity = float(enemy_details[5])
                    self.asteroids.append(Asteroid(start_
                    time, image, start_pos, velocity))
    except IOError:
       print ("Error reading configuration file "+configfile)
        # Just end as cannot play without config file
        sys.exit()
    except:
        print ("Corrupt configuration file "+configfile)
        sys.exit()

# Next level reset time
def next_level (self):
    self.level_time = time.time()
    for this_asteroid in self.asteroids:
        this_asteroid.reset()

def reset (self):
    self.level_time = time.time()
    for this_asteroid in self.asteroids:
```

```
            this_asteroid.reset()
    # Updates positions of all enemies
    def update(self, time_interval):
        # Check for level end reached
        if (self.level_end != None and
            time.time() > self.level_time + self.level_end):
                self.next_level()

        for this_asteroid in self.asteroids:
            this_asteroid.update(self.level_time, time_interval)

    # Draws all active enemies on the screen
    def draw(self, screen):
        for this_asteroid in self.asteroids:
            this_asteroid.draw()

    # Check if a shot hits something - return True if hit
    # otherwise return False
    def check_shot(self, shot):
        # check for any visible objects colliding with shot
        for this_asteroid in self.asteroids:
            # skip any that are not visible
            if this_asteroid.status != STATUS_VISIBLE:
                continue
            if (this_asteroid.colliderect(shot)):
                this_asteroid.hit()
                return True
        return False

    # Check if crashed - return True if crashed
    # otherwise return False
    def check_crash(self, spacecraft, collide_points=None):
        for this_asteroid in self.asteroids:
```

```
            # skip any that are not visible
            if this_asteroid.status != STATUS_VISIBLE:
                continue
            # Crude detection based on rectangles
            if (this_asteroid.colliderect(spacecraft)):
               # More accurate detection, but more time consuming
                # (optional if collide_points default to None)
                if (collide_points == None):
                    this_asteroid.status = STATUS_DESTROYED
                    return True
                for this_point in collide_points:
                    if this_asteroid.collidepoint(
                        spacecraft.x+this_point[0],
                        spacecraft.y+this_point[1] ):
                          this_asteroid.status = STATUS_DESTROYED
                            return True
    return False
```

The class has been named and written so that it can be extended to other enemies, not just asteroids. Much of the __init__ method is devoted to reading the configuration file. The configuration file uses comma-separated variations that define when each enemy appears, where they appear, and the speed at which they travel. This is the same as the previous code in Listing 11-4 but adds an extra option "end" to signify when the end of the level is reached and instead of printing to the screen a new instance of the Asteroid object is created. This is stored in the asteroids list.

Other methods handle changing a level including resetting all the enemies. The update method checks for the end of level time reached, but otherwise just calls the update for each of the asteroids. The draw method cycles through the draw of any asteroids that have been created. The check_shot and check_crash methods check to see if any shots or the spacecraft has hit an asteroid. If either of these has occurred, then the

asteroid is set to destroyed. The check_crash method uses a new technique to detect a collision. Previously the collision has used colliderect which uses a rectangle that encompasses the entire spacecraft. The problem with this is that due to the large area at the top of the image which is not part of the ship, the collision occurs too soon. This can be seen in Figure 11-8 where there is still a significant gap between the spacecraft and the asteroid, but their rectangles overlap.

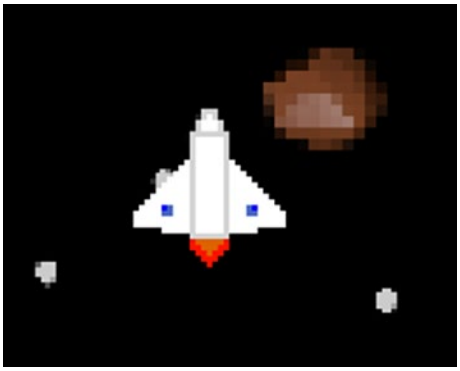


Figure 11-8. *Problem with colliderect on irregular shapes*

To overcome this problem, a list of points is used which is based on the extremities of the spacecraft.

The Player class is used for variables relating to the player. The code is included in Listing 11-11.

Listing 11-11. Player class in file player.py

```
class Player:
    def __init__ (self):
        self.lives = 3
        self.score = 0

    def reset (self):
        self.lives = 3
        self.score = 0
```

As you can see, this is a very simple class with only a few lines of code. It is used to store the number of lives that a player has remaining and to track the score. This is to avoid having global variables which are difficult to manage. Instead there is a single instance of the player class which can be used to hold the number of lives and the score.

The Shot class is a child of the Actor class used to track the shot. This is shown in Listing 11-12.

Listing 11-12. Player class in file player.py

```
from pgzero.actor import Actor

class Shot(Actor):

    def update(self, time_interval):
        self.y-=3 * 60 * time_interval
```

The shot is basically an actor with the image of the shot fired. Most of the functionality needed for the shot is provided from the parent class, but an update method is provided to move the position of the Actor on each refresh.

The rest of the code is in the spaceshooter.py file which is shown in Listing 11-13.

Listing 11-13. Space shooter main program file spaceshooter.py

```
import time
from constants import *
from spaceship import SpaceShip
from player import Player
from shot import Shot
from enemies import Enemies

WIDTH=800
HEIGHT=600
```

```
TITLE="Space shooter game"
ICON="spacecrafticon.png"

scroll_speed = 2

player = Player()

spacecraft = SpaceShip("spacecraft", (400,480))
spacecraft.set_speed(4)

enemies = Enemies((WIDTH,HEIGHT), "enemies.dat")

# List to track shots
shots = []
# shot last fired  timestamp - to ensure don't fire too many shots
shot_last_fired = 0
# time in seconds
time_between_shots = 0.5

scroll_position = 0

# spacecraft hit points
# positions relative to spacecraft center which classes as a collide
spacecraft_hit_pos = [
    (0,-40), (10,-30), (-10,-30), (13,-15), (-13,-15), (25,-3),
    (-25,-3),
    (46,12), (-46,12), (25,24), (-25,24), (10,27), (-10,27), (0,27) ]

# Status
# "start" = Press fire to start
# "game" = Game in progress
# "gameover" = Game Over
status = "start"

# value for waiting when asking for option
wait_timer = 0
```

```
def draw ():
    # Scrolling background
    screen.blit("background", (0,scroll_position-600))
    screen.blit("background", (0,scroll_position))

    enemies.draw(screen)

    spacecraft.draw()
    # Shots
    for this_shot in shots:
        this_shot.draw()

    screen.draw.text("Score: {}".format(player.score),
    fontname="computerspeak", fontsize=40, topleft=(30,30),
    color=(255,255,255))
    screen.draw.text("Lives: {}".format(player.lives),
    fontname="computerspeak", fontsize=40, topright=(770,30),
    color=(255,255,255))

    if status == "start" or status == "start-wait":
        screen.draw.text("Press fire to start game",
        fontname="computerspeak", fontsize=40,
        center=(400,300), color=(255,255,255))
    elif status == "gameover" or status == "gameover-wait":
        screen.draw.text("Game Over", fontname="computerspeak",
        fontsize=40, center=(400,200), color=(255,255,255))

def update(time_interval):
    global status, scroll_position, shot_last_fired, wait_timer
    # Allow Escape to quit straight out of the game regardless
    of state of the game
    if keyboard.escape:
        sys.exit()
    # Wait on fire key press to start game
```

```
    if status == "start":
        # start timer
        wait_timer = time.time() + DELAY_TIME
        status = "start-wait"
    if status == "start-wait":
        if (time.time() < wait_timer):
            return
        if keyboard.space or keyboard.lshift:
            player.reset()
            enemies.reset()
            status = "game"
    elif status == "gameover":
        # start timer
        wait_timer = time.time() + DELAY_TIME
        status = "gameover-wait"
    elif status == "gameover-wait":
        if (time.time() < wait_timer):
            return
        if keyboard.space or keyboard.lshift:
            status = "start"
    elif status == "game":
        # Scroll screen
        scroll_position += scroll_speed
        if (scroll_position >= 600):
            scroll_position = 0

        # Update existing shots
        for this_shot in shots:
            # Update position of shot
            this_shot.update(time_interval)
            if this_shot.y <= 0:
                shots.remove(this_shot)
```

```
    # Check if hit asteroid or enemy
    elif enemies.check_shot(this_shot):
        player.score += 10
        # remove shot (otherwise it continues to hit
          others)
        shots.remove(this_shot)
        sounds.asteroid_explode.play()

if enemies.check_crash(spacecraft, spacecraft_hit_pos):
    player.lives -= 1
    if player.lives < 1:
        status = "gameover"
        return
    else:
        sounds.space_crash.play()

# Update enemies after checking for a shot hit
enemies.update(time_interval)

# Handle keyboard
if keyboard.up:
    spacecraft.move("up")
if keyboard.down:
    spacecraft.move("down")
if keyboard.left:
    spacecraft.move("left")
if keyboard.right:
    spacecraft.move("right")
if keyboard.space or keyboard.lshift:
    # check if time since last shot reached
   if (time.time() > shot_last_fired + time_between_shots):
        # rest time last fired
        shot_last_fired = time.time()
```

```
            shots.append(Shot("shot",(spacecraft.x,spacecra
            ft.y-25)))
            # Play sound of gun firing
            sounds.space_gun.play()
```

Much of this code should be familiar as it uses similar techniques to those used in other games or in the listings earlier in this chapter.

The main class instances are the player, the spacecraft, and the enemies. There is also a list for tracking the shots that are fired. The list spacecraft_hit_pos is used for the collidepoint positions of the spacecraft.

The draw function includes the background scrolling code from Listing 11-2. It then calls each of the draw methods from the enemies and spacecraft objects. It also displays text as required, which uses the Computer Speak font. Details of the font are available from https://fontstruct.com/fontstructions/show/1436469 and license details included in the fonts directory.

The update function handles the state of the game, calling the various update methods and updates the position of the background scroll images.

It updates the shots and removes any shots that have gone off the top of the screen. It also checks to see if the spacecraft has crashed and update the lives or change the state to game over. The rest of the code handles key presses and movement of the craft and creates a new instance of the Shot class when the fire button is pressed.

The spaceshooter.py file also adds the sound effects through three files located in the `sounds` directory: `asteroid_explode.wav`, `space_crash.wav`, and `space_gun.wav`. These sound files are based on files under a Creative Commons license which are from the freesound library. I have edited the sounds using Audacity to change the pitch and filter out a limited frequency range. Details of the source are included in the license.txt file.

There is another file need to determine when the asteroids should appear. This is a file called enemies.dat. Unfortunately, Mu can only be used to edit files ending with .py so another text editor should be used to

edit the file. This is a configuration file which determines when each of the asteroids will appear. The configuration file is shown in Listing 11-14.

Listing 11-14. Space shooter enemies configuration file enemies.dat

```
0.3,asteroid,asteroid_sml,200,0,4
0.9,asteroid,asteroid_sml,100,0,4
0.9,asteroid,asteroid_med,400,0,3
1.2,asteroid,asteroid_sml,750,0,4
1.2,asteroid,asteroid_sml,400,0,4
1.6,asteroid,asteroid_lge,350,0,4
2.0,asteroid,asteroid_med,200,0,4
2.4,asteroid,asteroid_sml,150,0,2.5
2.5,asteroid,asteroid_med,450,0,4
2.7,asteroid,asteroid_med,605,0,4
3.0,asteroid,asteroid_lge,720,0,4
3.1,asteroid,asteroid_sml,380,0,4
3.6,asteroid,asteroid_lge,770,0,4
3.8,asteroid,asteroid_sml,200,0,3
3.8,asteroid,asteroid_sml,100,0,4
4.1,asteroid,asteroid_med,400,0,4
4.4,asteroid,asteroid_sml,750,0,4.5
5.0,asteroid,asteroid_sml,400,0,4
5.0,asteroid,asteroid_lge,350,0,3
5.0,asteroid,asteroid_med,200,0,4
5.2,asteroid,asteroid_sml,150,0,4
5.2,asteroid,asteroid_sml,600,0,3
5.2,asteroid,asteroid_med,620,0,4
5.2,asteroid,asteroid_med,450,0,5
5.5,asteroid,asteroid_lge,720,0,4
5.6,asteroid,asteroid_sml,380,0,4
6.0,asteroid,asteroid_lge,770,0,4
9.0,end
```

There are three different images referenced in the file, asteroid_sml.png, asteroid_med.png, and asteroid_lge.png, which are in the `images` directory.

There is also an icon file called spacecrafticon.png used for the icon shown on the application title bar.

The files enemies.dat and spacecrafticon.png need to be in the directory where the program is running. When running from the command line, this is normally where the spaceshooter.py file is located. If running the game from the Mu editor, then these two files will need to be in the `mu_code` directory.

Summary

The space shooter game has used various techniques that have been covered throughout the book. There are still other features that could be added. One feature that would be useful would be a high score such as the one added to the compass game. Another would be for a different type of enemy, perhaps one that could fire back instead of just crashing into the spacecraft. If you don't like the retro feel, then you could change the images to ones with better quality graphics.

Where Next?

Looking back through the games in the book, you can see that many of the games use the same or similar techniques. These are the essential skills needed to start creating games, but there is still lots more to learn. The best way to learn is by having a go write some code and create your own game.

You could start with one of the games from this book adding new features. You could start with one of the existing games and change the main player image to completely transform the game. Perhaps changing the spacecraft for a racing car and replacing the background with a racetrack that the car must navigate.

Feeling more adventurous? Now, that you have seen these implemented in different games, you will hopefully have learned enough to design and create your own games. The appendicies have some useful links to more information that you may find useful; this includes the Pygame Zero code as well as Pygame which can be used together.

Hopefully this has shown that programming games is something that is open to everyone with some programming experience.

I will be developing some of these games further or feel free to create your own versions. Any new versions I make will be shared using social media PenguinTutor on Twitter, Facebook, and YouTube. Feel free to share any improvements you make or anything that you create inspired by this book.

APPENDIX A

Quick Reference

This is a quick reference summary of some useful keywords, modules, and methods that are useful when programming in Pygame Zero.

Pygame Zero

Useful Keywords

```
WIDTH = 800             # Width of screen in pixels
HEIGHT = 600            # Height of screen in pixels
TITLE = "Title of game" # Title bar text
ICON = "filename"       # ICON image
```

Actor (Sprite)

Basic sprite operations:

```
sprite = Actor ('filename')     # Create sprite
sprite.topright = x_pos,y_pos   # Move top right to position
sprite.x = x_pos                # Change x position
sprite.y = y_pos                # Change y position
sprite.image = 'newfilename')   # Change image
```

© Stewart Watkiss 2020
S. Watkiss, *Beginning Game Programming with Pygame Zero*,
https://doi.org/10.1007/978-1-4842-5650-3

Detect collisions:

```
sprite.collidepoint(pos)    # Collide with position
sprite.colliderect(rect)    # Collide with rect or another sprite
```

Background Image or Color

For a background image, include this in the draw function:

```
screen.blit("imagefile", (0,0)) # Place image at 0,0
```

For a background color, include this in the draw function:

```
screen.fill ((red,green,blue))
```

Sound Effects

Play a sound effect from the sounds folder with a file filename.wav or filename.mp3.

```
sounds.filename.play()
```

Mouse Events

A common event is to check for a button click.

```
def on_mouse_down(pos, button):
    print ("Mouse button {}, clicked at {}".format(button, pos))
```

Other useful functions include on_mouse_up and on_mouse_move. The button numbers are as follows:

1. Left button
2. Middle button
3. Right button

4. Scroll forward
5. Scroll backward

Keyboard Events

Detect a keyboard event using similar methods to the mouse handling.

```
def on_key_down (key, mod, unicode):
    print ("Key {}, mod {}, char {}".format(key, mod, unicode))
```

The key value is the numerical value assigned to the key, mod is a bitmask for modifiers that are pressed at the time (shift has value 1), and unicode is the letter that is pressed.

You can also test if specific keys are pressed using

```
keyboard.<keyname>
```

Displaying Text

There are lots of options that can be applied when displaying text. Many of these are optional. Some common ones are shown here, but there are more.

```
screen.draw.text(
    "The text",
    (x_pos, y_pos),
    fontname="computerspeak", fontsize=40,
    color=(red,green,blue),
    shadow=(2,2), scolor=(red,green,blue)
    )
```

Python 3

These are lots of different techniques that are used in game programming. These are some useful ones.

Lists

Create a list.

```
list1 = ["value0", "value1", "value2"]
```

Access a list through numerical index.

```
print (list[1])
```

Index for lists starts at 0.

Dictionaries

Dictionaries are lists with a key for the index.

```
dictionary1 = {'key1':'value1', 'key2':'value2'}
```

Access using the key as the index.

```
print (dictionary1['key1'])
```

Access all keys.

```
dictionary1.keys()
```

Access all values.

```
dictionary1.values()
```

Conditional Statements (if, elif, else)

Conditional statements are used to run the appropriate action depending upon the boolean output of the condition.

```
if (condition1):
    action1()
elif (condition2):
    action2()
else:
    action3()
```

Loops

While loop across set number of loops:

```
num_times = 0
while (num_times < 10):
    print ("This is line number "+str(num_times))
    num_times += 1
```

For loop across range of values:

```
for x in range(0,10):
    print ("This is line number "+str(x))
```

For loop over a list:

```
for this_entry in this_list:
    print ("This entry "+this_entry)
```

To exit a loop:

```
break
```

To continue to the start of the loop:

```
continue
```

Python 3 Modules

These are part of the core modules which are included in all Python installs.

Random

First the random module must be imported.

```
import random
```

Random number between 0.0 and 1.0:

```
random.random()
```

Random integer:

```
random.randint (0,10)
```

Select a random entry.

```
random.choice (list)
```

Math

The math module includes numerous mathematical functions. First import the module.

```
import math
```

Round the value in x up to the nearest whole integer.

```
math.ceil(x)
```

Round the value in x down to the nearest whole integer.

```
math.floor(x)
```

Convert angle x from radians to degrees.

```
math.degrees(x)
Convert angle x from degrees to radians.
```

```
math.radians(x)
```

Trigonometry functions using angle in radians.

```
math.cos(x)
math.sin(x)
math.tan(x)
```

Pi constant for π:

```
math.pi
```

Time

The time module provides the time relative to the epoch. The epoch is system dependent which for Linux is January 1, 1970, 00:00:00 (UTC). The time module is useful for relative times for use within a game. For the actual date and time, see the datetime module. First import the module.

```
import time
```

Get time as a floating-point number in seconds from the epoch.

```
time.time()
```

Suspend the program for a number of seconds. This is not normally used for games created using Pygame Zero.

```
time.sleep(secs)
```

DateTime

The datetime module is used for the date and time.

```
import datetime
```

Get current date and time. When printed, display using yyyy-mm-dd hh:mm:ss.microseconds.

```
now =  datetime.datetime.now()
```

Get the different components of the date.

```
year = now.year
month = now.month
day = now.day
hour = now.hour
minute = now.minute
second = now.second
microsecond = now.microsecond
```

Format the date and time into a string (yyyy-mm-dd hh:mm:ss).

```
now.strftime("%Y-%m-%d %H:%M.%S")
```

The different options can be rearranged into an appropriate date format.

APPENDIX B

More Information

These are some links to the official documentation where more information can be found.

Python

Python 3 documentation:

https://docs.python.org/3/

Python 3 string methods:

https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str

Python 3 data structures (including lists and dictionaries):

https://docs.python.org/3/tutorial/datastructures.html

Python 3 control flow (if statement and loops):

https://docs.python.org/3/tutorial/controlflow.html

Pygame Zero

Pygame Zero documentation:

https://pygame-zero.readthedocs.io/en/stable/

© Stewart Watkiss 2020
S. Watkiss, *Beginning Game Programming with Pygame Zero*,
https://doi.org/10.1007/978-1-4842-5650-3

Pygame

Pygame documentation:

www.pygame.org/docs/

Pygame color names (this is a link to the actual source code for Pygame):

https://github.com/pygame/pygame/blob/master/src_py/colordict.py

Index

A

B

C

© Stewart Watkiss 2020
S. Watkiss, *Beginning Game Programming with Pygame Zero*,
https://doi.org/10.1007/978-1-4842-5650-3

D

E

F

G, H

I

J, K

L

M, N

O

P, Q

R

S

T

U

V

W, X, Y, Z