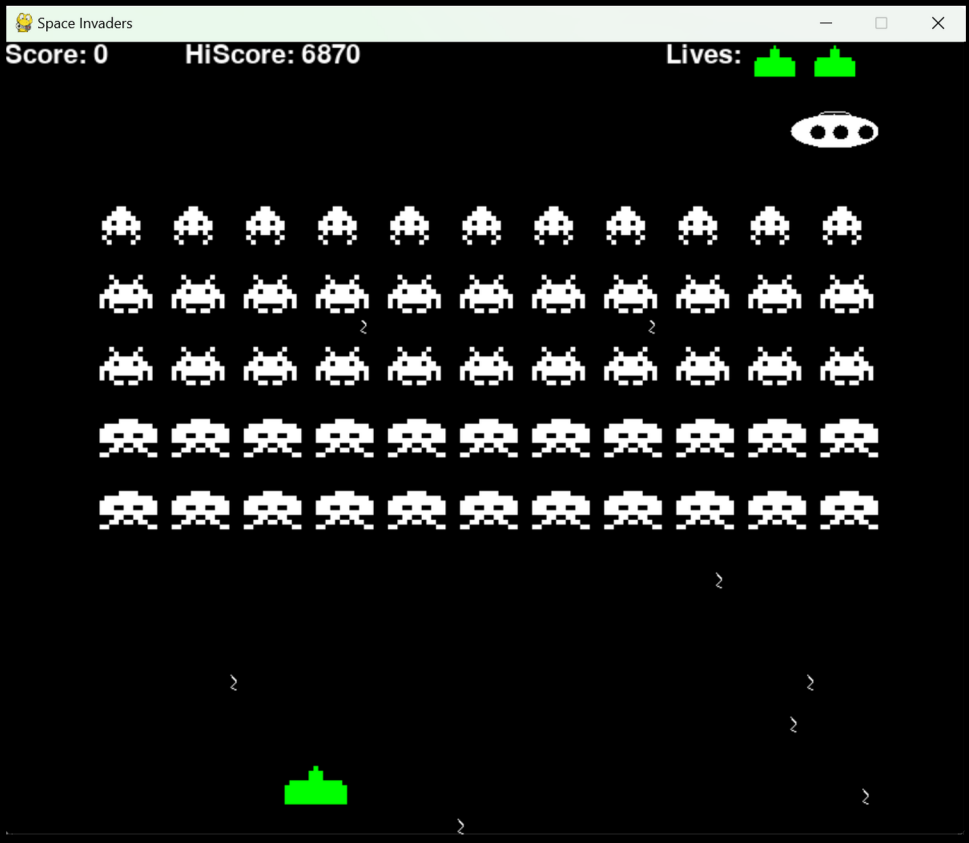
Creating Video Games Using PyGame



Mike Gold

Creating Video Games using PyGame

With Step by Step Examples

Mike Gold

This book is for sale at
http://leanpub.com/creatingagameusingpygame

This version was published on 2023-05-02 ISBN 979-8-89034-116-7

Table of Contents

Setting up Python and Pygame

Welcome to the world of PyGame and Python programming! This book will provide you with a comprehensive introduction to the PyGame library and teach you how to create your own, custom games using the Python language. We will start with a basic overview of Python and the PyGame library, before moving on to designing, writing, and debugging our own game. From adding graphics and sounds, to creating animations and power-ups, we will cover everything that you need to know to create your own rich, interactive game. Finally, we will go through the process of debugging and testing our game, before publishing it for the world to enjoy. So, let's get started and learn how to make your own game with PyGame and Python!

Getting Started

Installing Python

You can find the latest version of Python at Python.org[1]. There are both 32-bit and 64-bit versions available. Once you have clicked on the Download button, run the executable that you downloaded at follow the instructions to install the latest python on your machine.

[1] https://www.python.org/downloads/

Installing VSCode

Visual Studio Code is available for Windows, MacOS, Linux operating systems. You can download visual studio code from https://code.visualstudio.com/download. Choose the appropriate download for your OS and then run the installation. Once you've installed Visual Studio Code, you'll want to install Python and Pylance extensions.

Python Extension:

The Python extension for Visual Studio Code provides a wide range of features to help make Python development in VS Code easier, including linting, debugging, IntelliSense code completion, code formatting, refactoring, unit testing, and more. The extension is open source and available for free, and it can be installed by searching for it in the VS Code extension market. With the Python extension, developers can quickly and easily create and manage their Python projects, and also take advantage of a wide range of advanced features.

Pylance Extension:

Pylance is a Visual Studio Code extension that provides an enhanced Python language support, including fast feature-rich IntelliSense, linting, project-wide analysis, and debugging. Pylance uses the Language Server Protocol (LSP) to communicate with the language server, and it supports a wide range of features such as autocomplete, code refactoring, code navigation, and error diagnostics. Pylance also provides an auto-import feature which can automatically add imports for symbols when you type them in your code. Pylance is a great tool for Python developers to quickly and efficiently write code.

To install the extensions, go to the extensions symbol on the left hand bar of Visual Studio Code and search the marketplace for Pylance. Click on it and Install the extension into VisualStudio Code. Also look for an extension called Python and install that as

well.

Installing Pygame

Pygame is an open source library for making games in Python. It has a wide range of features and functions that make it easy to get started making games.

You can find the documentation for Pygame at pygame.org[2].

To get started using Pygame, you will need to install it. The easiest way to install pygame is from the terminal inside of VSCode. Click the terminal at the top of the menu and type the following line:

pip install pygame

if you don't have pip already installed, you will have to go to https://bootstrap.pypa.io/get-pip.py and download the file into your python application directory. To figure out where python is installed, you can actually ask python! Go to the terminal in Visual Code and type

```
python
```

You'll see the >>> prompt. Put in the following code

```
>>> import os
>>> import sys
>>> os.path.dirname(sys.executable)
```

This will spit out the path where you'll place your get-pip.py file.

e.g. C:\Python310 on windows

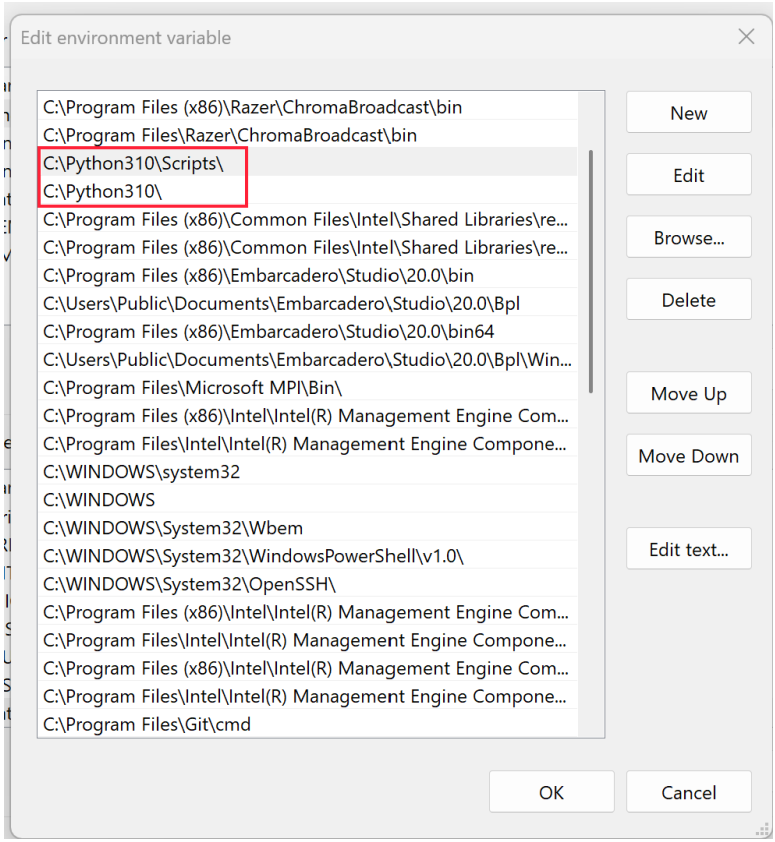
Place get-pip.py in the revealed path and then run

[2] http://www.pygame.org/docs/

```
1  py get-pip.py
```

⚠ Note: You may need to add the python path your environment variables path

The two paths I have are shown below



Intro To Python

In the chapters to file, we will be programming in Python, so we need to give you the foundation to understand the language constructs we will be using as well as how to run them. The following chapter will guide you through the use of the most common pieces of the language and what we will be utilizing to build our game. First let's answer a few common questions about Python.

History of Python

Python was created by Guido van Rossum and first released in 1991. Python is a high-level, interpreted, general-purpose programming language. Python has become popular due to its clear syntax and readability. Python also supports modules and packages, which allows for code reusability.

Python is an interpreted language, which means that it is compiled at run-time. This allows Python code to be more forgiving of errors and makes debugging easier. Python also supports a number of open source systems and frameworks, such as Django and Flask.

Python is often used for scientific computing, web development, machine learning, and automation. Python has a large and active community, making it easy to find help and support online. Python is used by organizations such as Google, Yahoo, and NASA.

What makes Python Different than other languages?

Python is an interpreted language, which makes it easier to get started with than other languages such as C or Java. It is also dynamically typed, meaning you don't need to declare a type when creating a variable. This makes the language more expressive and can reduce the complexity of some applications. Python is also highly extensible, which means that it can be extended with existing libraries and new modules written in C, C++, or other languages. Additionally, Python's syntax is relatively simple and easy to learn.

What types of applications are built with Python?

Python is used in a wide variety of applications, including desktop GUI applications, web applications, software development, scientific and numeric computing, and artificial intelligence and machine learning. Many of the most popular websites and services such as YouTube, Instagram, Quora, and Dropbox were built using Python.

Why should I learn Python?

As discussed, Python is a powerful and versatile programming language with a wide range of applications and uses. It is easy to learn and has a high readability level, making it a great choice for beginners, yet it is also popular with experienced developers. It is a versatile language, meaning it can be used for a variety of tasks - from web development to data science and machine learning. Python also has a strong community of developers and users, so there is always support and new tools available. Additionally,

Python is an open-source language, meaning that it can be used for free and is accessible to anyone with internet access.

Now that you know a little bit about the language, let's create our first Python program, just to get your feet wet. We are going to jump into python.

Let's start with a simple program that prints Hello World:

Create a New Folder in VSCode called **HelloWorld**. Then create a new file called **HelloWorld.py** and add the following line.

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

Save your file. Go to your terminal in VSCode (the bash terminal) and run the python with the following command:

```
py -m HelloWorld
```

You should see the following output in your terminal window:

```
Hello World
```

Not Bad! If you got this far, you are up and running. Let's turn up the dial a little bit. Let's write a program that writes Hello World 10 times. For this we will use a for loop. A for loop let's us loop through a range of values and each time through the loop print 'Hello World'. Alter your HelloWorld.py file to the code below and run.

```
for number in range(5):
    print ('Hello World')
```

This program produces the following output

```
Hello World
Hello World
Hello World
Hello World
Hello World
```

Note that our for loop has a number inside. Each time the number goes through the loop it increments to the next number. We can show this in our print statement by using string interpolation. Change HelloWorld.py to this code:

```
for number in range(5):
    print (f'Hello World #{number}')
```

This program produces this output after being run:

```
Hello World #0
Hello World #1
Hello World #2
Hello World #3
Hello World #4
```

Notice that the range function starts at 0 and ends at 4 and not 5. If we wanted our hello world to count to five, we could just add one to the number

```
for number in range(5):
    print (f'Hello World #{number+1}')
```

This produces an output that numbers Hello World 1-5:

```
Hello World #1
Hello World #2
Hello World #3
Hello World #4
Hello World #5
```

The if statement

What if we only wanted to print out even 'Hello World's? We can now introduce the if statement which allows us to make some decisions of which of the Hello Worlds gets printed

```
for number in range(5):
    numberToPrint = number + 1
    if numberToPrint % 2 == 0:
       print (f'Even Hello World #{numberToPrint}')
```

This code introduces the if statement for making decisions. In this case, the if statement uses the mod function (%) to determine if there are any remainders when the next number is divided by 2. If the remainder of nuumberToPrint divided by 2 is zero, the print will get executed. So for example 2 % 2 has no remainders so it passes the mod test of numberToPrint % 2 == 0 and will print the Even Hello World #2. On the other hand, 5 % 2 equals 1, so it fails the test of being equal to 0 since 0 does not equal 1. The print will be skipped for 5.

So after running the program, the code will print "Even Hello World #2", "Even Hello World #4". It will skip printing "Even Hello World #1", "Even Hello World #3", and "Even Hello World #5" since none of those numbers are even and meet the criteria of the mod function.

```
Even Hello World #2
Even Hello World #4
```

The else statement

if we want a more complete answer to our even number print out, we can also print whether the number is even or odd, we'll use the else statement to help us here:

```
for number in range(5):
    numberToPrint = number + 1
    if (numberToPrint) % 2 == 0:
       print (f'Even Hello World #{numberToPrint}')
    else:
       print (f'Odd Hello World #{numberToPrint}')
```

The else statement is executed when the condition in the if statement is false. It's used to execute different code when the condition is not true. In the example above, the else statement prints out a message with the word *Odd Hello World #{numberToPrint}* when the number is odd.

```
Odd Hello World #1
Even Hello World #2
Odd Hello World #3
Even Hello World #4
Odd Hello World #5
```

elif

In Python, the **elif** statement (short for "else if") is a conditional statement that allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions

evaluates to TRUE. The **elif** statement follows the same syntax as the if statement, but with one additional keyword: elif. For example, the following code will check if the numberToPrint is divisible by 3, and if not, it will check if the numberToPrint is even. If neither of those is true, the **else** will kick in and it will print that its neither even or divisible of 3:

```
for number in range(5):
    numberToPrint = number + 1
    if numberToPrint % 3 == 0:
       print (f'{numberToPrint} is divisible by 3')
    elif numberToPrint % 2 == 0:
       print (f'{numberToPrint} is even')
    else:
       print (f'{numberToPrint}
        Not even and not divisible by 3')
```

Here is the output for the code illustrating how if elif else works:

```
1 Not even and not divisible by 3
2 is even
3 is divisible by 3
4 is even
5 Not even and not divisible by 3
```

The while loop

A while loop allows us a lot of flexibility over looping through data:

Sometimes it gives us too much flexibility! The following while loop would run forever:

```
while True:
  print('Hello World')
```

Here the condition is always true, so it would never end the loop. While loops end when the condition after the while is false. The other way to break out of the loop is with a break statement:

```
while True:
  print('Hello World')
  break
```

The output for this loop is:

```
Hello World
```

because the program will still enter the while loop and print 'Hello World', but right after it hits the print statement, it will hit the break, whihc it will cause it to break out of the loop.

we can show the power of the while loop, by rewriting our for loop above:

```
number = 1
while number <= 5:
    print (f'Hello World #{number}')
    number = number + 1
```

The while loop starts with a number equal to 1, so the while loop guard condition is true because 1 is less than or equal to 5. The print statement executes with 1 and next the number is incremented by 1 to become 2. The program will then loop back and since 2 is still less than or equal to 5 it will execute the print again and increment the number to 3. It will continue to do so until it no longer meets the guard condition. When number equals 6 it is no longer less than or equal to five, so it will break out of the loop. Here are the results from running this program:

```
Hello World #1
Hello World #2
Hello World #3
Hello World #4
Hello World #5
```

Python Lists

Python Lists are data structures that store a collection of items that can be of different data types. Items in the list are accessed using an index. Lists allow for efficient storage of data by allowing for quick retrieval and modification of data stored in them. Lists can be used for various tasks such as sorting, searching, and manipulating data.

We can do the same thing we did in our previous loops by looping through a list:

```
numbers = [1, 2, 3, 4, 5]
for number in numbers:
    print (f'Hello World #{number}')
```

numbers is the list of items numbering 1 through 5. We loop through the numbers the same way we looped through the values returned from the range function.

outputs:

```
Hello World #1
Hello World #2
Hello World #3
Hello World #4
Hello World #5
```

Adding, Inserting, and removing from a list

In Python, lists can be modified using the append, insert, and remove methods. The append method adds an element to the end

of the list, the insert method adds an element at a specified index, and the remove method removes the element at a specified index. For example, the following code will add the element 'c' to the end of the list, add the element 'd' at index 2, and remove the element with value 'a':

```
letters = ['a', 'b', 'e']
print(letters, 'starting letters')
letters.append('c')
print('append c: ',letters)
letters.insert(2, 'd')
print('insert d at position 2: ', letters)
letters.remove('a')
print('remove a: ',letters)
```

output:

```
starting letters:  ['a', 'b', 'e']
append c:  ['a', 'b', 'e', 'c']
insert d at position 2:  ['a', 'b', 'd', 'e', 'c']
remove a:  ['b', 'd', 'e', 'c']
```

These methods are very useful when you want to modify a list without having to create a new list from scratch.

Two Dimensional List

A two-dimensional list in Python is a type of data structure that stores data in multiple dimensions.[1] It is essentially a list of lists, where each inner list is a row of data, and each outer list is a column of data. Two-dimensional lists are useful for data that is organized in rows and columns, such as tables, spreadsheets, and matrixes. To access data in a two-dimensional list, you need to reference the

[1]This is well described on stack overflow: https://stackoverflow.com/questions/2397141/how-to-initialize-a-two-dimensional-array-in-python

index of the row and column. For example, to access the item in the third row and the fourth column of a two-dimensional list, you could use the following code:

```
my_list = [[1,2,3,4], [5,6,7,8], [9,10,11,12]]

value = my_list[2][3]
print(value)
```

output for this code:

```
12
```

Twelve is located in the third array, in the 4th element of that array (remember that lists are indexed starting at 0, so we start counting at 0)

```
value = my_list[0][0]
print(value)
```

The above code outputs 1 since 1 is at the 0 index which is the first list in the array of lists and the 0th element into that first array.

Another way to think of a 2 dimensional arrays is in terms of rows and columns. This code can be thought of as accessing the number 1 which appears in the first row and the first column of a two-dimensional array. The variable my_list is a two-dimensional array, and the two indices are used to specify the row and column of the item we wish to access. The first index (0) specifies row zero which consists of [1,2,3,4]. The second index (0) specifies column zero. The code prints the value of the item at row 0, column 0, which in this case will be 1.

List Comprehension

List comprehension is a concise way to create lists in Python. It is a syntactic construct that allows you to create a new list by specifying the elements you want to include, often derived from existing lists or other iterable objects. List comprehensions can also include optional conditions to filter the elements. They are more readable and often faster than equivalent code using loops.

Mapping a list

Suppose you have a list of numbers and you want to create a new list with the squares of those numbers. Using list comprehension:

```
numbers = [1, 2, 3, 4, 5]

squares = [x**2 for x in numbers]
print(squares)
```

output: [1, 4, 9, 16, 25]

Using list comprehension to create a new list by squaring each number from an existing list is equivalent to looping over the numbers, squaring them, and appending the result to a new list, but list comprehension provides a more compact and concise syntax.

Filtering a list

If you want to create a new list containing only the even numbers from the original list, you can add a filtering condition to the list comprehension. Below is an example to extract all the even numbers from the numbers list.

```
even_numbers = [x for x in numbers if x % 2 == 0]
print(even_numbers)
```

output: [2, 4]

List comprehension applies a filter to the numbers list using the condition **if x % 2 == 0** to selectively include only even numbers in the new even_numbers list. Any numbers that do not satisfy the condition are excluded from the resulting list.

Two-dimensional lists

List comprehension can be used to create or modify 2D lists as well. For example, if you have a 2D list (matrix) and you want to create its transpose, you can use nested list comprehensions:

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

transpose = [[row[i] for row in matrix] for i in range(le\
n(matrix[0]))]
print(transpose)
```

output: [[1, 4, 7], [2, 5, 8], [3, 6, 9]]

In this example, the outer list comprehension iterates over the indices of the columns, while the inner list comprehension iterates over the rows. The resulting transpose list is a new 2D list containing the transposed elements of the original matrix.

Functions

In Python, functions are pieces of reusable code that can take one or more input values, perform some operations on them, and return a result. Functions are used to make code more organized, readable, and reusable. Functions can be defined using the def keyword, and they can take any number of parameters. For example, the following function takes two parameters, x and y, and returns the sum of the two:

```
def add(x, y):
    return x + y

sum = add(2, 3)
print(sum) # 5
```

Functions can also take optional parameters, which are parameters that have default values. For example, the following function takes two parameters, x and y, and an optional parameter named z, which has a default value of 0:

```
def add(x, y, z=0):
    return x + y + z

sum = add(2, 3)
print(sum) # 5
```

Python also supports anonymous functions, which are functions that do not have a name and are defined using the lambda keyword. Anonymous functions can take any number of parameters, and they always return a single expression. The following is an anonymous function for adding two numbers:

```
sum = lambda a, b: a + b
```

We can call the lambda function simply by calling the variable we assigned to it:

```
  result = sum(4,5)
  print('4 + 5 = ', result)
```

Functions can also have variable-length arguments, which allows them to accept any number of arguments. These arguments are stored in a tuple. For example, the following function takes any number of arguments and prints them out:

```
def print_args(*args):
    for arg in args:
        print(arg)

print_args('a', 'b', 'c', 'd')
```

The output of this function will be:

```
a b c d
```

Additionally, functions can also return multiple values. Instead of returning a single value, you can return a tuple containing multiple values. This allows you to return multiple pieces of data from a single function call. For example, the following function returns two values:

```
def get_info():
    name = "John"
    age = 25
    return (name, age)

name, age = get_info()
print(name) # John
print(age) # 25
```

Functions are a powerful and versatile tool in Python, and they can be used to write code that is more organized, readable, and reusable. With functions, you can break up your code into smaller chunks and organize them into meaningful, self-contained units. This allows you to easily reuse the code in different parts of your program, and it makes debugging and troubleshooting much easier. Additionally, functions can also be used to simplify complex logic and make code more readable. Finally, functions can also be used to improve performance by allowing code to be run in parallel, which can speed up execution time.

Tuples

In Python, tuples are immutable sequences of objects. They are typically used to store related pieces of information, such as coordinates or a record of data. Tuples are created using parentheses, and they can contain any type of object, including other tuples. Tuples can also be used to return multiple values from a function. For example, you can use the following code to return two values from a function:

```
def get_info():
    name = "John"
    age = 25
    return (name, age)


name, age = get_info()
print(name) # John
print(age) # 25
```

Tuples are also useful for grouping related data together. For example, if you have a list of coordinates, you can use a tuple to store each coordinate in one place.

Tuples are also a great way to create an efficient and secure dictionary. When you create a dictionary with tuples, the order of the elements in each tuple will determine the order of the keys in the dictionary. This can help you avoid accidentally overwriting or deleting data.

Classes

Classes in Python are like templates for creating objects. They are the basic building blocks for any object-oriented programming language. A class defines the properties, behavior, and attributes

of an object. Classes also provide methods, which are functions that act upon the data in the class. Classes are typically used to create objects that represent real world objects, like a car or a person. Objects created from classes can have their own values and methods which can be used to carry out tasks.

Here is a simple Person class example in python:

```
class Person:
  def __init__(self, name, age, gender):
    self.name = name
    self.age = age
    self.gender = gender

  def introduce(self):
    print(f'Hello, my name is {self.name}')
```

The Person class defines the attributes has and functions that a person can perform. Below is an example of instantiating a new object called tim from the Person class. After we create tim, we can call the introduce method for tim.

```
tim = Person("Tim", 28, "Male")
tim.introduce()
```

output for calling introduce is shown below:

```
Hello, my name is Tim
```

The introduce method uses the name we used to construct the Person tim with. If we wanted to construct a person named Mary, we would do the following:

```
mary = Person("Mary", 30, "Female")
```

mary is another instance of a Person, just like tim, but Mary will give a different response if she introduces herself:

```
mary.introduce()
```

output:

```
Hello, my name is Mary
```

We will be using classes extensibly when building a PyGame because games are much easier to manage when they are organized into classes.

Here are some built in classes used in PyGame:

pygame.Surface

pygame.Surface is a fundamental class in the pygame library that represents a rectangular, 2D area in memory where you can draw, manipulate, and store pixel data. Surfaces are used for various graphical operations in pygame, such as rendering images, text, and shapes.

pygame.sprite.Sprite

This is the base class for visible game objects. All visible game objects are derived from this class.

pygame.rect.Rect

This class is used to define a rectangular area of the screen. It is used to store and manipulate the size, position and location of a rectangular area. It is used to determine collisions between objects.

pygame.time.Clock

This class is used to manage time and game loops. It helps you keep track of time and manage the game's frame rate.

pygame.font.Font

This class is used to render text on the screen. It helps you render text with a specified font, size, style and color.

Intro To PyGame

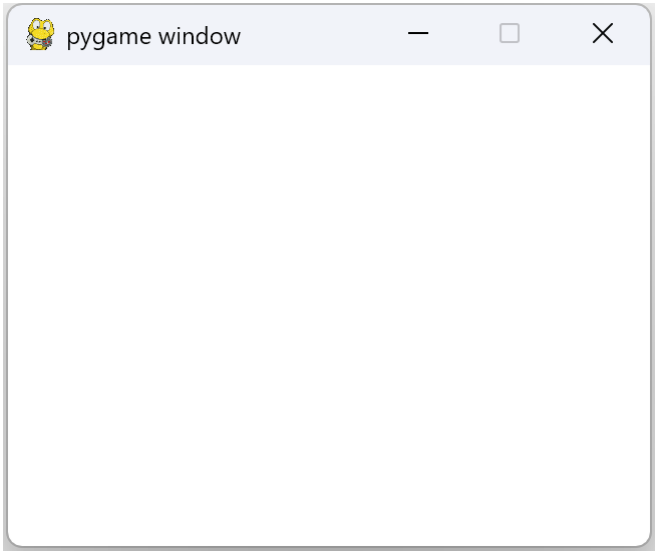
Pygame is a Python module designed specifically for game development. It provides a set of tools and libraries for building games and multimedia applications. Pygame was first created by Pete Shinners in 2000, as a side project to explore Python's multimedia capabilities. He released the first version of Pygame in March 2000, which included basic functionality such as image loading, sound playback, and event handling. Over the years, Pygame has grown and evolved with the Python language, adding new features and capabilities. In 2007, the Pygame community created the Pygame Subset for Android, which allowed Pygame applications to run on Android devices. Today, Pygame is widely used by game developers and enthusiasts alike, and its popularity continues to grow as Python becomes increasingly popular as a programming language for game development.

To get started with PyGame, you can use the following code to fill the a 320x240 pixel window with a white background:

```
1  import pygame
2
3  # Initialize the game
4
5  pygame.init()
6
7  # Create the screen
8  gamewindow = pygame.display.set_mode((320, 240))
9
10 WHITE = (255, 255, 255)
11
12 while True:
13     for event in pygame.event.get():
```

```
14          if event.type == pygame.QUIT:
15              sys.exit()
16      gamewindow.fill(WHITE)
17      pygame.display.flip()
```

The **pygame.init()** initializes pygame and the pygame.display gets us access to the game window. **while True:** is our game loop that loops through our game forever or until someone closes the pygame screen which triggers a quit event. The **pygame.display.flip** function updates the contents of the entire display. It is typically used after drawing or updating the display to make sure that the changes are visible. This function is also known as a "screen flip" or "page flip", as it swaps the front and back buffers, which contain the contents of the display.



Hello World

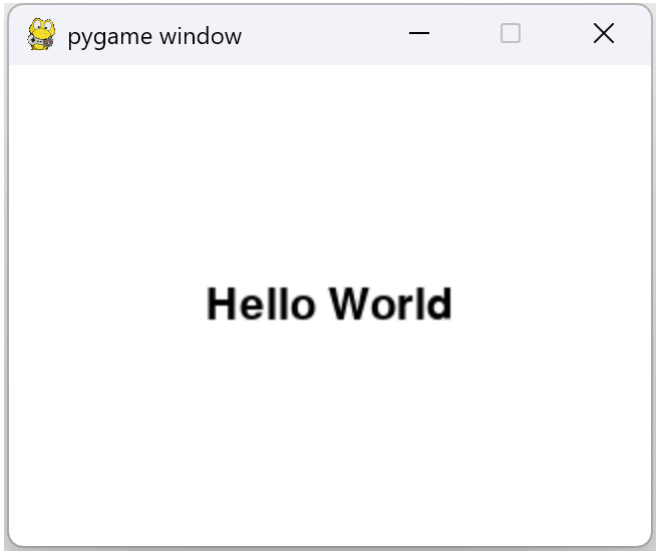
To add the hello world text to our blanks screen, we need to create a font object. The **font = pygame.font.Font(None, 32)** line creates a font object with size 32. This font object can then be used to render text onto a display surface. The None argument specifies that the default font should be used. If a font file is passed in, then that font will be used instead. The font size is specified in points, where 1 point is 1/72 of an inch.

The **font.render()** function is used to render text onto a display surface. It takes three arguments: the text to be rendered, whether the text should be anti-aliased, and the color of the text. In this case, the font.render() function is used to render the text "Hello World" with anti-aliasing turned on and the color black. The rendered text is then stored in the text surface.

To put the text on the screen, we first need to blit the text object onto the screen surface.We will use the blit function to draw the text surface onto the screen at the center. The screen.blit() function takes two arguments: the surface to be drawn, and the position at which it should be drawn. The first argument in this case is the text surface that was created on the previous font.render. The second argument is a tuple containing the coordinates of the center of the screen, which is calculated by subtracting half of the width and height of the text surface from the width and height of the screen. This allows the text to be centered on the screen.

```
import pygame

# Initialize the game

pygame.init()

# Create the screen
screen = pygame.display.set_mode((320, 240))
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)

# create the font object
font = pygame.font.Font(None, 32)

while True:
    screen.fill(WHITE)  # fill the background

    # check for quit event
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    # create the text surface
    text = font.render("Hello World", True, BLACK)

    # blit the text surface to the center of the screen
    screen.blit(text, ((screen.get_width() -
           text.get_width())/2,
           (screen.get_height() - text.get_height()) / 2))

    # update the screen
    pygame.display.flip()
```

This code will produce the following output window:



Blinking Hello World

Let's say we want to blink **Hello World** black and red every second. We could add a time delay in our game loop and alternate different color text objects. Here we take advantage of a python list to alternate between colors when creating the text object:

```
1  import pygame
2
3  # Initialize the game
4
5  pygame.init()
6
7  # Create the screen
8  screen = pygame.display.set_mode((320, 240))
9  WHITE = (255, 255, 255)
10 BLACK = (0, 0, 0)
```

```
11  RED = (255, 0, 0)
12
13  HelloWorldColors = [BLACK, RED]
14
15  # create the font object
16  font = pygame.font.Font(None, 32)
17
18  count = 0
19  while True:
20      count = count + 1
21      screen.fill(WHITE)  # fill the background
22
23      # check for quit event
24      for event in pygame.event.get():
25          if event.type == pygame.QUIT:
26              sys.exit()
27
28      # create the text surface
29      # and alternate the color using either RED or BLAcK
30      text = font.render("Hello World", True,
31                HelloWorldColors[count % 2])
32
33      ### wait 1 second (or 1000 milliseconds)
34      pygame.time.delay(1000)
35
36      # blit the text surface to the screen
37      screen.blit(text, ((screen.get_width() -
38                          text.get_width())/2,
39                          (screen.get_height() -
40                            text.get_height()) / 2))
41
42      # update the screen
43      pygame.display.flip()
```

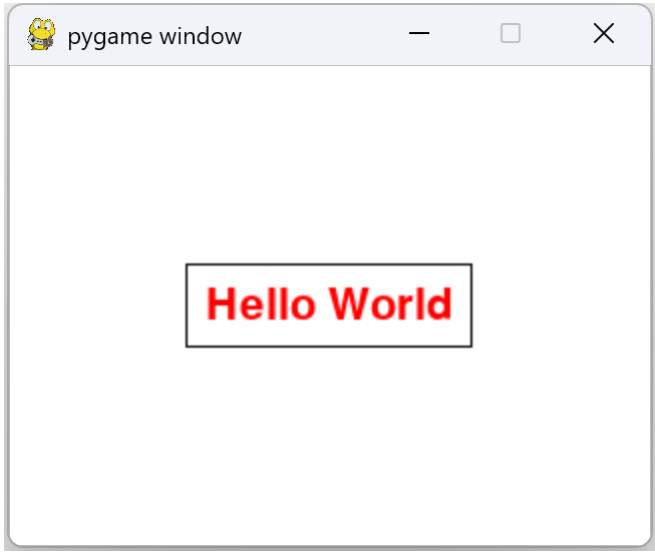
surrounding with a border

What if we wanted to place a black border around the blinkin ghello world? The code code shown below is used to draw a black rectangular border around the text surface. The **pygame.draw.rect()** function takes five arguments: the display surface to draw on, the color of the rectangle, the coordinates of the top left corner of the rectangle, the width and height of the rectangle, and the thickness of the line. In this case, the top left corner of the rectangle is calculated by subtracting 10 from the coordinates of the center of the screen, and the width and height of the rectangle is calculated by adding 20 to the width and height of the text surface. The 1 argument specifies that the line should be 1 pixel thick.

```
while True:
    count = count + 1
    screen.fill(WHITE)  # fill the background

    # check for quit event
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    # create the text surface
    text = font.render("Hello World", True,
                 HelloWorldColors[count % 2])
    pygame.time.delay(1000)

    # surround text with black rectangular border
    pygame.draw.rect(screen, BLACK,
                ((screen.get_width() -
                text.get_width())/2 - 10,
                (screen.get_height() -
                text.get_height()) / 2 - 10,
                text.get_width() + 20,
```

```
                text.get_height() + 20), 1)

    # blit the text surface to the screen
    screen.blit(text, ((screen.get_width() -
                        text.get_width())/2,
                        (screen.get_height() -
                        text.get_height()) / 2))

    # update the screen
    pygame.display.flip()
```

This results in the following screen:



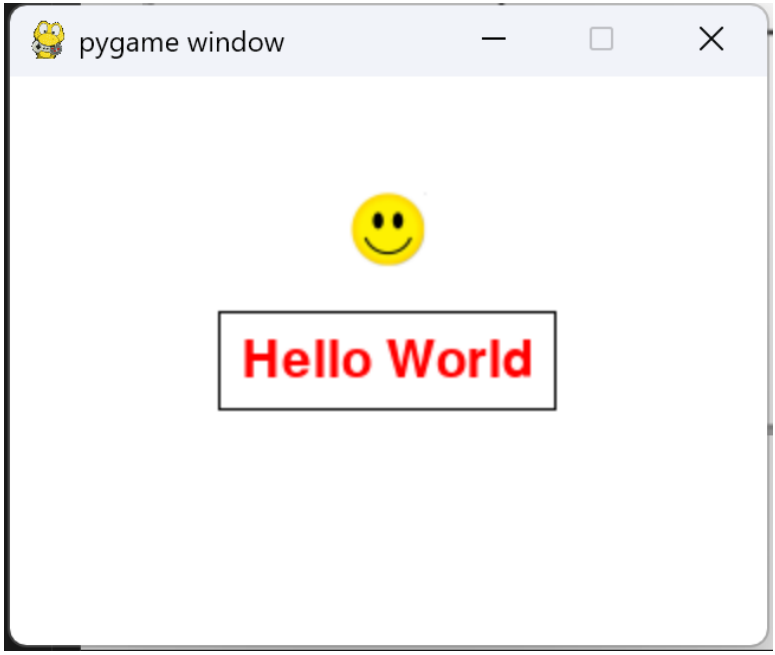
Adding an Image

Adding an image to the view is as easy loading it and blitting it to the screen surface. You will probably be using a lot of images when creating your games, so its good to know how to draw an image. In

the code below, we load an image of a smiley face and then blit it centered above the Hello World Text:

```
    # draw an image of a smiley above the border with a s\
ize of 32x32

    screen.blit(pygame.image.load("resources/smiley.png"),
                (screen.get_width()/2 - 16,
                (screen.get_height()
                - text.get_height()) / 2 - 60))
```

This results in the following screen rendering:



Adding Sound to our Game

Pygame comes complete with the ability to draw shapes, fonts, and images to the screen. It also comes with facility to play sounds and music. The following code shows you how to add a beep sound to the Hello World Screen every time it blinks.

First we need to initialize the pygame sound module before the game loop:

```
# Initialize the mixer
# to play sound
pygame.mixer.init()
pygame.mixer.music.load("resources/shortbeep.mp3")
```

Inside our game loop we can play the sound each time through our game loop after everything is drawn. There is already a 1 second delay we have added to the game loop, so the shortbeep file will be played every second.

```
count = 0
while True:
    count = count + 1
    screen.fill(WHITE)  # fill the background

    # check for quit event
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    # create the text surface
    text = font.render("Hello World", True, HelloWorldCol\
ors[count % 2])

    # 1 second delay
```

```
16      pygame.time.delay(1000)
17
18      # surround text with black rectangular border
19      pygame.draw.rect(screen, BLACK,
20        ((screen.get_width() - text.get_width())/2 - 10,
21        (screen.get_height() -
22        text.get_height()) / 2 - 10,
23        text.get_width() + 20,
24        text.get_height() + 20), 1)
25
26      # draw an image of a smiley above the border with a s\
27  ize of 32x32
28
29      screen.blit(pygame.image.load("resources/smiley.png"),
30                  (screen.get_width()/2 - 16,
31                      (screen.get_height() -
32                      text.get_height()) / 2 - 60))
33
34      # blit the text surface to the screen
35      screen.blit(text, ((screen.get_width() -
36              text.get_width())/2,
37              (screen.get_height() - text.get_height()) / 2\
38  ))
39
40  # play the beep
41      pygame.mixer.music.play()
42
43      # update the screen
44      pygame.display.flip()
```

Responding to the Keyboard

You may have noticed that we have added event process the beginning of our game loop shown in the code below. Currently

the event processing only looks to see if we quit the game or not.

```
  # check for quit event
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
```

The event processing can also allow us to read keyboard and mouse events and use them in our game. Let's first see if we can use the left mouse button to indicate where to place smiley. When the left mouse is pressed down, it will trigger the pygame.MOUSEBUTTONDOWN event inside the game loop. The event loop will get the mouse position from the mouse, and we can use that to place smiley at the mouse position on the screen

```
    for event in pygame.event.get():
        if event.type == pygame.MOUSEBUTTONDOWN:
            mouse_pos = pygame.mouse.get_pos()
...

# draw an image of a smiley above the border with a size \
of 32x32
    if mouse_pos != None:
        screen.blit(pygame.image.load(
            "resources/smiley.png"),
            (mouse_pos[0] - 16, mouse_pos[1] - 16))
    else:
        screen.blit(pygame.image.load(
            "resources/smiley.png"),
            (screen.get_width()/2 - 16,
            (screen.get_height() -
            text.get_height()) / 2 - 60))
```

So now when you click somewhere in the game window, you'll see smile move to where you clicked! You may notice there is a delay

from when you click and when smiley actually gets painted. That's because we added the 1 second delay in the code for sound. There is actually a better way to handle the blinking of Hello World and the beep sound every second so it does not interfere with retrieving events. The way to do this is to add a clock instead of a delay, and only execute the beep and the blink when the clock reaches the 1 second mark on the clock. The best way to illustrate the use of the clock is by seeing the code:

```
    time = pygame.time

count = 0
oneSecondMarkReached = False
lastTime = 0

while True:
    # increment the count every second
    if oneSecondMarkReached:
        count = count + 1

    screen.fill(WHITE)  # fill the background

    # check for quit event
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
        elif event.type == pygame.MOUSEBUTTONDOWN:
            mouse_pos = pygame.mouse.get_pos()

    # create the text surface
    text = font.render("Hello World", True,
               HelloWorldColors[count % 2])

    # surround text with black rectangular border
    pygame.draw.rect(screen, BLACK,
```

```
27          ((screen.get_width() - text.get_width())/2 - 10,
28          (screen.get_height() -
29          text.get_height()) / 2 - 10,
30          text.get_width() + 20,
31          text.get_height() + 20), 1)
32
33  # draw an image of a smiley above the border with a size \
34  of 32x32
35      if mouse_pos != None:
36          screen.blit(pygame.image.load(
37              "resources/smiley.png"),
38              (mouse_pos[0] - 16, mouse_pos[1] - 16))
39      else:
40          screen.blit(pygame.image.load(
41              "resources/smiley.png"),
42              (screen.get_width()/2 - 16,
43              (screen.get_height() -
44                text.get_height()) / 2 - 10 - 50))
45
46      # blit the text surface to the screen
47      screen.blit(text,
48         ((screen.get_width() -
49             text.get_width())/2,
50             (screen.get_height()
51                 - text.get_height()) / 2))
52
53      if oneSecondMarkReached:
54          pygame.mixer.music.play()
55
56      # update the screen
57      pygame.display.flip()
58
59      # reset the oneSecondMarkReached flag
60      oneSecondMarkReached = False
61
```

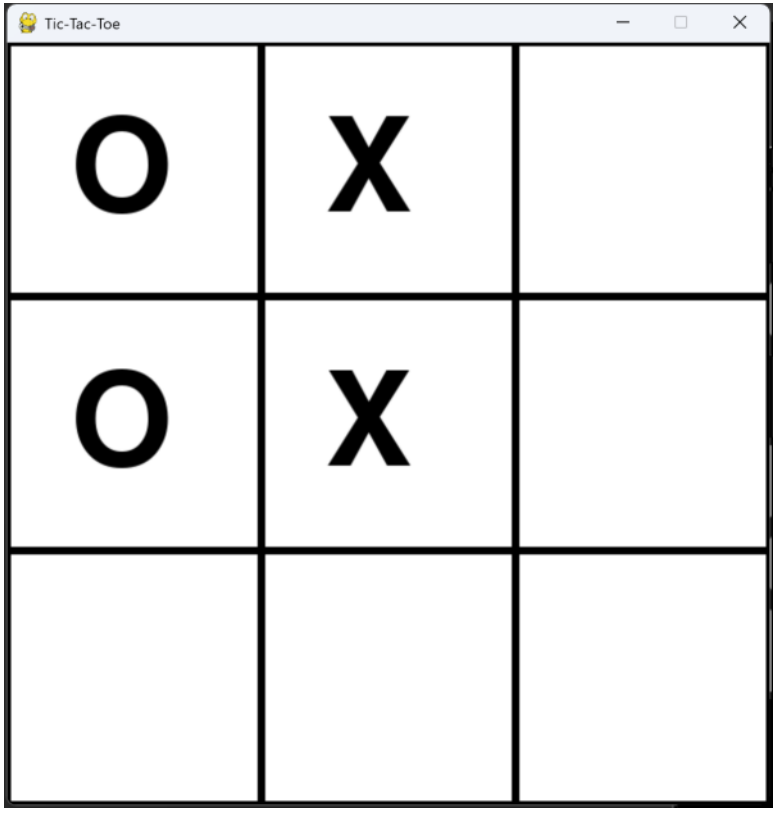
```
62      # inform the program every time
63      # the 1 second mark is reached
64
65      currentTime = time.get_ticks()
66      if currentTime - lastTime > 1000:
67          lastTime = currentTime
68          oneSecondMarkReached = True
```

We needed to alter the code slightly so the blink happens every second and the beep happens every second. We use the oneSecondMarkReached flag and set it every 1000 ticks (1 sec in time). It then gets reset once it has executed the beep and the performed the color change to the "Hello World" text.

Conclusion

We've examine a bunch of concepts to get us started using the many game related elements provided by the pygame library. We learned how to fill the background on the screen, draw text, load and draw an image, and play music and sound. In the next chapter we will dive right into creating our first game, tic-tac-toe.

Tic Tac Toe in PyGame



Intro

Welcome to the chapter on writing a Tic Tac Toe game with PyGame. In this chapter, we will explore the basics of the PyGame library and how to use it to write a simple two-player Tic Tac Toe game. We will cover how to draw the game board, how to detect

user input, and how to implement a basic AI to play against. By the end of this chapter, you should have a working Tic Tac Toe game that you can play against the computer. So let's get started!

Main Loop

The following code is a main game loop for a tic-tac-toe game implemented with the PyGame library. It runs an event processing loop to check for user input, then draws the game board. If the game is not over, it checks if the player has placed an X, then waits half a second to simulate the AI thinking before it places an O. After it places an O, it checks if anyone has won the game, and if no one has won, it checks if it is a draw. Finally, it updates the display.

```
######################################################
# Main Game Loop
######################################################
while True:
    if game_over:
        pygame.display.flip()
        pygame.time.delay(1000)
        draw_game_over_screen()
        # Run the event processing to check for quit
        check_for_quit_event()
    else:
        game_window.fill(WHITE) # white background
         # check for quit and mouse down
        run_event_processing()
        # Check for win or draw
        game_over = check_for_win_or_draw()
        draw_the_board()  # Draw the game board
        pygame.display.flip()  # Update the display

        # Check if anyone won after X was placed
```

```
        if game_over:
            continue

        # AI Goes here to place O
        if X_placed:
            # Wait for 1/2 second to make it
            # look like AI is thinking
            pygame.time.delay(500)
            O_placed = run_algorithm_to_place_O()
            game_over = check_if_anyone_won()
            # Draw the board again to show the
            # O we just placed
            draw_the_board()
            X_placed = False

    # Update the display
    pygame.display.flip()

    # limit the loop to 60 frames a second
    clock.tick(60)
```

Processing Events

Underlying the contents of the game loop are several functions that leverage pygame to do the heavy lifting. Let's look first at the function DoEventProcessing. This code is a function in PyGame which runs an event processing loop to check for user input and mouse clicks. When the user clicks on the board, it handles the mouse down event for X and sets the X_placed flag to True. It also checks to see if the user chooses to quit the game. The Quit Event is triggered when the user closes the window.

```
def run_event_processing():
    global X_placed
    global game_over

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()  # quit the game
            quit()
        if event.type == pygame.MOUSEBUTTONDOWN:
            # Populate X on the Board
            handle_mouse_down_for_x()
            X_placed = True
```

Now let's take a look at the handle_mouse_down_for_x function being called. This code is used to handle the mouse down event for placing an X on the tic-tac-toe board. It uses the PyGame library's mouse.get_pos() function to get the mouse position, then divides the row and column by the grid width and height to get the row and column of the click. Finally, it sets the corresponding position in the board array to "X".

```
def handle_mouse_down_for_x():
    (row, col) = pygame.mouse.get_pos()
    row = int(row / grid_width)
    col = int(col / grid_height)
    board[row][col] = "X"
```

Drawing the Board

The function draw_the_board is used to draw the Tic Tac Toe board in its current state. It loops through all the rows and columns of the board and calls the draw_game_board_square() function to draw each square. Then, it checks if the board at that row and column

contains an “X” or an “O” and calls the draw_tic_tac_toe_letter() function to draw the corresponding letter.

```
def draw_the_board():
    for row in range(grid_size):
        for col in range(grid_size):
            draw_game_board_square(row, col)
            # Render letter X
            if (board[row][col] == "X"):
                draw_tic_tac_toe_letter(row, col, 'X')
           # Render letter O
            if (board[row][col] == "O"):
                draw_tic_tac_toe_letter(row, col, 'O')
```

Drawing the Game Square

This code is used to draw the game board square at the specified row and column. It uses the PyGame library’s Rect() function to create a rectangle object with the given row, column, width, and height. Then, it uses the draw.rect() function to draw the rectangle on the game window with a black color and a line width of 3.

```
def draw_game_board_square(row, col):
    rect = pygame.Rect(col * grid_width, row *
                       grid_height,
                       grid_width,
                       grid_height)
    pygame.draw.rect(game_window, BLACK, rect, 3)
```

Drawing the Tic-Tac-Toe Letter

This code is used to draw the letter ‘X’ or ‘O’ at the specified row and column. It uses the PyGame library’s font.render() function to render the letter as a Surface object and sets the color to black. Then,

it uses the game_window.blit() method to draw the letter at the specified row and column, with the row and column multiplied by the grid width and height, plus a quarter of the grid width and height to center it.

```
def draw_tic_tac_toe_letter(row, col, letter):
    letter_piece = font.render(letter, True, BLACK)
    game_window.blit(
        letter_piece, (row * grid_width + grid_width/4,
                       col * grid_height + grid_height/4))
```

"AI" for placing an O

For simplification, the algorithm for placing an O is to just look for the next available square. We will improve this later in the chapter, but this strategy should at least allow you to play the game against a computer opponent.

```
#######################################################
# A very simple algorithm to place O on the board.
# Loop through the entire board and look for the first
# available square. Place the O there.
#######################################################
def run_algorithm_to_place_O():
    for rowo in range(grid_size):
        for colo in range(grid_size):
            if (board[rowo][colo] == 0):
                board[rowo][colo] = "O"
                return True

    return False
```

Check for a Win

The following code checks for a win on the board. It looks to see if there are three of the same characters in a row on the board (horizontally, vertically, and diagonally).

```
def check_if_anyone_won():
    global winner
    # Check if someone won horizontally
    for row in range(3):
        if board[row][0] == board[row][1]
                == board[row][2] != 0:
            winner = board[row][0]
            return True
    # Check if someone won vertically
    for col in range(3):
        if board[0][col] == board[1][col]
                == board[2][col] != 0:
            winner = board[0][col]
            return True
    # Check if someone won diagonally
    if board[0][0] == board[1][1]
            == board[2][2] != 0:
        winner = board[0][0]
        return True
    if board[0][2] == board[1][1]
           == board[2][0] != 0:
        winner = board[0][2]
        return True

    # no one won, return false
    return False
```

Check for a Draw

We also need to check if there are no more places to place an X or O and neither player won the game. The way we do this is to create a new function that checks to see if the board is full. If nobody won and the board is full, then its a draw.

```
def check_if_board_is_full():
    for row in range(3):
        for col in range(3):
            if board[row][col] == 0:
                return False
    return True


#####################################################
# Check if there is a draw by checking if the board is
# full and no one has won
#####################################################

def check_if_draw():
    return not (check_if_anyone_won()) and
       check_if_board_is_full()
```

Handling the Game Over State

Once we determined if there is a win, a lose, or a draw, we set the game_over flag to True. When we detect that the game is over, we want to display a game over screen instead of a tic-tac-toe board. In our main loop we check the game_over flag, and if its true, we draw the game over screen instead of the tic-tac-toe board:

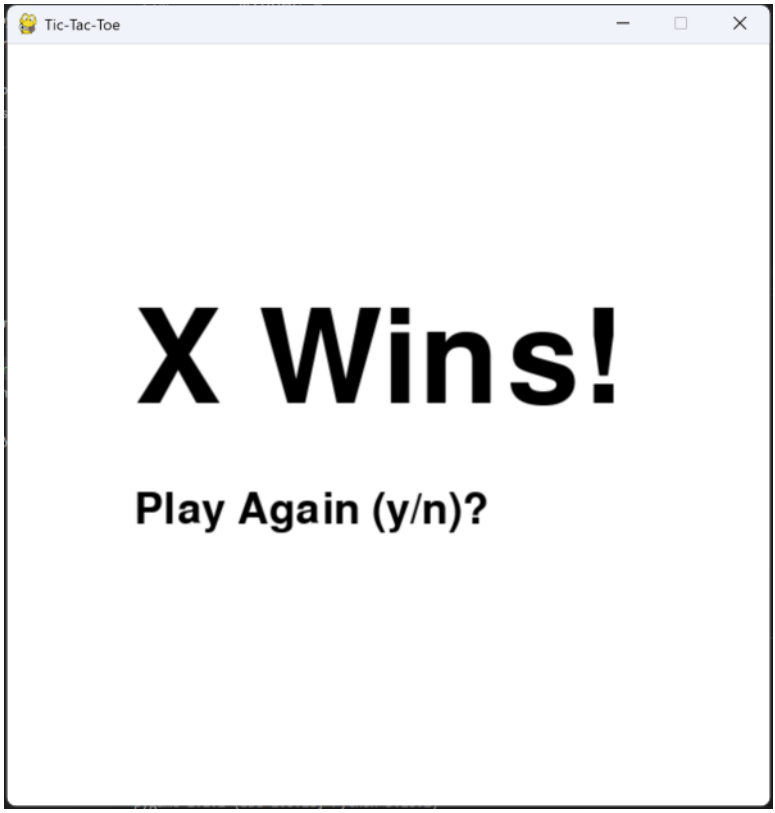
```
if game_over:
        # draw the game over screen
        pygame.display.flip()
        pygame.time.delay(1000)
        draw_game_over_screen()
        check_for_quit_event()  # Run the event processin\
g to check for quit
    else:
        # draw the tic tac to board
```

The following python code draws the game over screen instead of the tic-tac-toe board, once we determined that the game has finished. It checks the winner string and displays the appropriate message as to what happened in the game based on that string. The Game Over Screen also conveys to the player the option of playing a new game or not:

```
#######################################################
# Draw the game over screen showing who won
#######################################################
def draw_game_over_screen():
    game_window.fill(WHITE)
    if winner == "X":
        text = font.render('X Wins!', True, BLACK)
    elif winner == "O":
        text = font.render('O Wins!', True, BLACK)
    else:
        text = font.render('Draw!', True, BLACK)

    playAgainText = smallfont.render(
          'Play Again (y/n)?', True, BLACK)

    game_window.blit(text,
        (window_width/2 - 200, window_height/2 - 100))

```

```
19      game_window.blit(playAgainText,
20          (window_width/2 - 200, window_height/2 + 50))
```

The resulting game over screen image is shown below:



Playing Again

In order to allow the player to play a new game, we need to clear the state of the current game when the user hits the y key.

We reset the global game state with a new function called **initialize_game_values**. This method gets triggered if the user hits the

'y' key in the game over state:

```
def check_for_quit_event():
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_y:
                initialize_game_values()
                game_window.fill(WHITE)
                return True
            elif event.key == pygame.K_n:
                pygame.quit()
                quit()
```

Once a y is detected in the Event Loop, then we initialize the game board and start over. We have not stopped the game loop, so the game loop will automatically paint the board based on the reset variable state once initialize_game_values is called.

```
def initialize_game_values():
    global board
    global game_over
    global X_placed
    global O_placed
    global winner
    global clock

    game_over = False
    X_placed = False
    O_placed = False
    winner = ''

    board = [
```

```
15          [0, 0, 0],
16          [0, 0, 0],
17          [0, 0, 0],
18      ]
19
20      clock = pygame.time.Clock()
```

A better AI

Recall our discussion about implementing a more advanced AI for the "O" player in the game? This enhanced algorithm is designed to never lose! Here are the steps for this refined algorithm:

1. Count the number of moves made so far.
2. If it's the second move (only one move has been made): a. Place "O" in the center of the board, if it's empty. b. If the center is occupied, place "O" in the first available corner.
3. For all empty positions on the board: a. Check if placing "O" in the current position would result in a win for the "O"player. If so, place "O" and return True. b. Check if placing "O" in the current position would block the "X" player from winning. If so, place "O" and return True.
4. If "O" started in a corner, place "O" in the first available corner and return True.
5. Place "O" in the first available non-corner side position and return True.
6. If none of the above conditions apply, place "O" in the first available position and return True.
7. If no empty positions are available, return False.

The algorithm uses a series of rules to decide the optimal position for placing "O" on the Tic-Tac-Toe board. It takes into account the current state of the board and makes decisions based on winning or

blocking the opponent from winning, as well as prioritizing corners and non-corner sides depending on the situation.

Here is the python code. Note we broke it down into 3 functions **run_better_algorithm_to_place_O**, **is_winning_move**, and **get_empty_positons** :

```
# check if placing a piece in the row
# and column results in a winning move
def is_winning_move(player, row, col):
    n = len(board)
    # Check row
    if all(board[row][j] == player
        for j in range(n)):
        return True
    # Check column
    if all(board[i][col] == player
        for i in range(n)):
        return True
    # Check main diagonal
    if row == col and all(board[i][i]
       == player for i in range(n)):
        return True
    # Check secondary diagonal
    if row + col == n - 1 and
        all(board[i][n - i - 1]
          == player for i in range(n)):
        return True
    return False

# return empty positions on the board in a list
def get_empty_positions():
    empty_positions = []
    for i, row in enumerate(board):
        for j, cell in enumerate(row):
            if cell == 0:
```

```
30                  empty_positions.append((i, j))
31      return empty_positions
32
33
34  def run_better_algorithm_to_place_O():
35      grid_size = len(board)
36      empty_positions = get_empty_positions()
37      num_moves = sum(1 for row in board for
38              cell in row if cell != 0)
39
40      # Second move: Place "O" in center or corner
41      if num_moves == 1:
42          center = grid_size // 2
43          if board[center][center] == 0:
44              board[center][center] = "O"
45              return True
46          else:
47              for row, col in
48                  [(0, 0), (0, grid_size - 1),
49                  (grid_size - 1, 0),
50                  (grid_size - 1, grid_size - 1)]:
51                  if board[row][col] == 0:
52                      board[row][col] = "O"
53                      return True
54
55      # Try to win or block X from winning
56      for row, col in empty_positions:
57          # Check if placing "O" would win the game
58          board[row][col] = "O"
59          if is_winning_move("O", row, col):
60              return True
61          board[row][col] = 0
62
63      # Check if placing "O" would block X from winning
64      for row, col in empty_positions:
```

```
65          board[row][col] = "X"
66          if is_winning_move("X", row, col):
67              board[row][col] = "O"
68              return True
69          board[row][col] = 0
70
71      # Place "O" in a corner if it started in a corner
72      if board[0][0] == "O"
73          or board[0][grid_size - 1] == "O"
74          or board[grid_size - 1][0] == "O"
75          or board[grid_size - 1][grid_size - 1]
76               == "O":
77          for row, col in
78              [(0, 0), (0, grid_size - 1),
79              (grid_size - 1, 0),
80              (grid_size - 1, grid_size - 1)]:
81              if board[row][col] == 0:
82                  board[row][col] = "O"
83                  return True
84
85      # Place "O" in a non-corner side
86      for row, col in empty_positions:
87          if row not in [0, grid_size - 1]
88             and col not in [0, grid_size - 1]:
89              board[row][col] = "O"
90              return True
91
92      # Place "O" in any available space
93      for row, col in empty_positions:
94          board[row][col] = "O"
95          return True
96
97      return False
```

Conclusion

Having learned the basics of creating a game with pygame, we can now take our skills to the next level by streamlining our code through object-oriented programming. By organizing our game objects into classes, we can simplify and streamline our code, making it easier to manage and maintain. In the upcoming chapter, we'll explore this technique in more detail and show you how to implement it in your own games.

Using Classes in Pygame

Introduction

Tic Tac Toe and other games we write in Pygame are condusive to being broken down into smaller objects which the program can act upon. Classes in Python can help developers create re-usable and maintainable code. By organizing the code into classes, the code is easier to read and understand and allows for more efficient debugging and testing. Additionally, it allows developers to easily modify and add new pieces to the game without having to rewrite the code, which is especially useful when developing complex games. Classes also help keep the code organized and make it easier to implement new features. Finally, using classes helps developers create more efficient code since they can easily use the same code for similar game pieces.

Let's dive into creating classes for our game. A Python class representing a letter (X or O) that can be drawn on the screen in pygame can be defined as follows:

Here is an example of a Pygame class that inherits from the Sprite class and draws an 'X' or an 'O' on the screen:

```
import pygame


class LetterSprite(pygame.sprite.Sprite):

    def __init__(self, letter, row, column,
                 grid_width, grid_height):
        # initialize the sprite base class
        super().__init__()
        font = pygame.font.Font(None, 150)
        # render the font to an image surface
        self.image = font.render(letter, True, (0, 0, 0))
        # determine the image boundaries on the board
        self.rect = self.image.get_rect().move(
            row * grid_width + grid_width / 3,
            column * grid_height + grid_height / 3)

    def update(self):
        pass

    def draw(self, surface):
        letter_piece = self.image
        surface.blit(letter_piece, self.rect)
```

The Letter class takes a 5 arguments: the letter itself, which is stored as an instance variable, the row and column where the letter is placed on the board, and the grid dimensions. The **init** constructor does most of the hard work. It creates the image from the default font and calculates the rect position from the row, column, and grid dimensions. Since there is no movement from its current position, the letter does not need to have an update method, so we don't do anything with update. The draw() method takes only one argument: screen, which is the game surface. It then uses the pygame.Surface as a means to blitting the Letter onto the game screen.

We can construct our LetterSprite as soon as their is a mousedown

event from the player, then we can use the Group class in pygame to collect all the x's we add to the board.

```
def handle_mouse_down_for_x():
    (row, col) = pygame.mouse.get_pos()
    row = int(row / grid_width)
    col = int(col / grid_height)
    board[row][col] = "X"
    letterX = LetterSprite('X', row, col,
          grid_width, grid_height)
    group.add(letterX)
```

The reason we add the X's to the group, is when we want to draw all the game pieces, we simply call group.draw(surface) and it will draw all the game pieces for us at once. As we shall soon see, we can do the same thing with the "O"'s as well!

Now we can remove 90% of the code that draws X's and O's and it will boil down to one line of code: group.draw(game_window)

```
def draw_the_board():

    group.draw(game_window)

    for row in range(grid_size):
        for col in range(grid_size):
            draw_game_board_square(row, col)
```

Notice that we are still looping to create game squares. We can create Sprites for the game squares as well:

```
import pygame

# Create the sprite
class GameBoardSquareSprite(pygame.sprite.Sprite):
    def __init__(self, color, row, column, width, height):
        super().__init__()
        self.width = width
        self.height = height
        # Create a surface for the sprite
        self.image = pygame.Surface([width, height])
        # make the background game tile white
        self.image.fill((255, 255, 255))
        self.rect = self.image.get_rect().move(row*width,
            column*height)
        # Draw the rectangle to the sprite surface
        pygame.draw.rect(self.image, color, pygame.Rect(
            0, 0, width, height),  2)

    # Draw the sprite on the screen

    def draw(self, surface):
        surface.blit(self.image, 0, 0)
```

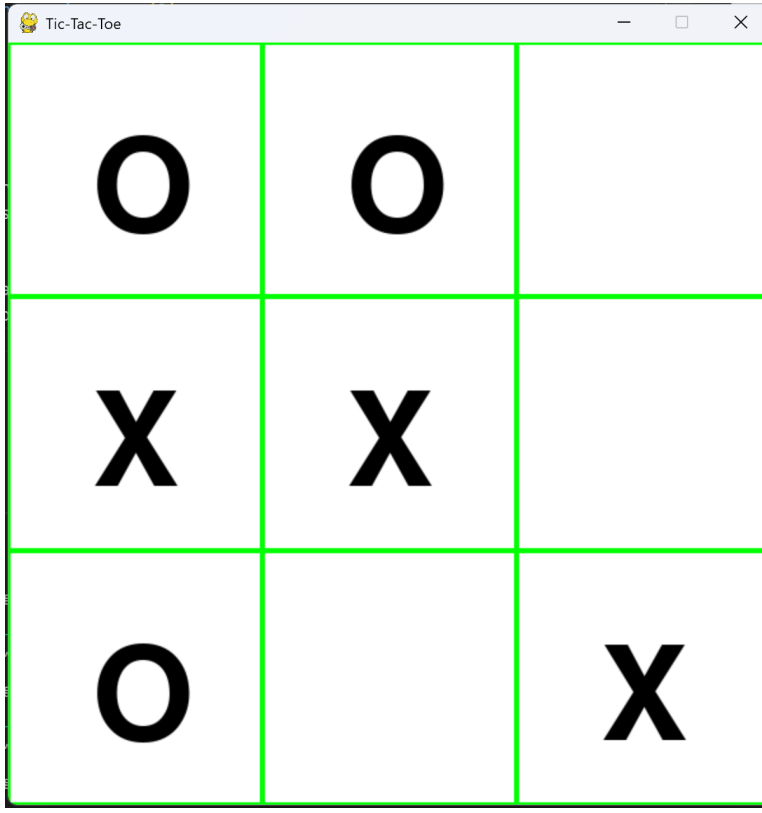
Now in our initialize_game_board, we'll add the game tiles to the group:

```
def initialize_game_board():
    for row in range(3):
        for column in range(3):
            game_board_square = GameBoardSquareSprite(
                (0, 255, 0),  row, column,
                 grid_width, grid_height)
            group.add(game_board_square)
```

When group.draw is called, it will draw the tiles as well as the X's and O's played. Our draw_the_board function now looks like this:

```
def draw_the_board():
    group.draw(game_window)
```

Since we chose to make our game board squares green, the resulting board looks like the figure below:



Refactoring the Game Logic

In order to modularize the code even more, we can pull all the game logic out of the main python module and into a class called GameBoard. The GameBoard class will check for wins, losses, and draws, as well as give us a way to populate the board with our guess.

It can also control the algorithmic logic for placing the O's.

```
class GameBoard:
    def __init__(self, grid_size):
        self.grid_size = grid_size
        self.winner = ''
        self.initialize_board()

    ########################################################
    # Initialize the board with zeroes
    ########################################################

    def initialize_board(self):
        self.board = [
            [0, 0, 0],
            [0, 0, 0],
            [0, 0, 0],
        ]

#####################################################
# Check if someone won in any row, column or diagonal
#####################################################

```

```
22      def check_if_anybody_won(self):
23          # Check if someone won horizontally
24
25          for row in range(3):
26              if self.board[row][0] == self.board[row][1]
27                == self.board[row][2] != 0:
28                  self.winner = self.board[row][0]
29                  return True
30
31          # Check if someone won vertically
32          for col in range(3):
33              if self.board[0][col] == self.board[1][col]
34                == self.board[2][col] != 0:
35                  self.winner = self.board[0][col]
36                  return True
37
38          # Check if someone won diagonally
39          if self.board[0][0] == self.board[1][1]
40          == self.board[2][2] != 0:
41              self.winner = self.board[0][0]
42              return True
43          if self.board[0][2] == self.board[1][1]
44          == self.board[2][0] != 0:
45              self.winner = self.board[0][2]
46              return True
47
48          return False
49
50      ########################################################
51      # Check if the board is full
52      ########################################################
53
54      def check_if_board_is_full(self):
55          for row in range(3):
56              for col in range(3):
```

```
57                  if self.board[row][col] == 0:
58                      return False
59          return True
60
61      ########################################################
62      # Check if there is a draw by checking if the
63      # board is full and no one has won
64      ########################################################
65
66      def check_if_draw(self):
67          return not (self.check_if_anybody_won()) and
68             self.check_if_board_is_full()
69
70      ########################################################
71      # Place the X
72      ########################################################
73      def place_X(self, row, col):
74          self.board[row][col] = "X"
75
76      ################################################
77      # Used by run_better_algorithm_to_place_O to
78      # determine if placing the piece in the row or column
79      # on the board results in the winning move.  This
80      # is used for determining blocking as well as winning
81      # for the "O" opponent
82      ################################################
83      def is_winning_move(self, player, row, col):
84          n = len(self.board)
85          # Check row
86          if all(self.board[row][j] == player
87             for j in range(n)):
88              return True
89          # Check column
90          if all(self.board[i][col] == player
91             for i in range(n)):
```

```
92              return True
93          # Check main diagonal
94          if row == col and all(self.board[i][i] ==
95              player for i in range(n)):
96                  return True
97          # Check secondary diagonal
98          if row + col == n - 1 and
99              all(self.board[i][n - i - 1]
100             == player for i in range(n)):
101             return True
102         return False
103
104     ################################################
105     # Used by the run_better_algorithm_to_place_O method
106     # to collect all the available positions on the board
107     ################################################
108     def get_empty_positions(self):
109         empty_positions = []
110         for i, row in enumerate(self.board):
111             for j, cell in enumerate(row):
112                 if cell == 0:
113                     empty_positions.append((i, j))
114         return empty_positions
115
116 ######################################################
117 # Uses an algorithm to decide where to place an O
118 # This algorithm never loses
119 ######################################################
120     def run_better_algorithm_to_place_O(self):
121         grid_size = len(self.board)
122         empty_positions = self.get_empty_positions()
123         num_moves = sum(1 for row in self.board for
124                         cell in row if cell != 0)
125
126         # Second move: Place "O" in center or corner
```

```
127             if num_moves == 1:
128                 center = grid_size // 2
129                 if self.board[center][center] == 0:
130                     self.board[center][center] = "O"
131                     return (True, center, center)
132                 else:
133                     for row, col in [(0, 0),
134                         (0, grid_size - 1),
135                         (grid_size - 1, 0),
136                         (grid_size - 1, grid_size - 1)]:
137                         if self.board[row][col] == 0:
138                             self.board[row][col] = "O"
139                             return (True, row, col)
140 
141             # Try to win or block X from winning
142             for row, col in empty_positions:
143                 # Check if placing "O" would win the game
144                 self.board[row][col] = "O"
145                 if self.is_winning_move("O", row, col):
146                     return (True, row, col)
147                 self.board[row][col] = 0
148 
149             # Check if placing "O" would block X from winning
150             for row, col in empty_positions:
151                 self.board[row][col] = "X"
152                 if self.is_winning_move("X", row, col):
153                     self.board[row][col] = "O"
154                     return (True, row, col)
155                 self.board[row][col] = 0
156 
157             # Place "O" in a corner if it started in a corner
158             if self.board[0][0] == "O"
159                 or self.board[0][grid_size - 1] == "O"
160                 or self.board[grid_size - 1][0] == "O"
161                 or self.board[grid_size - 1][grid_size - 1]
```

```
162                 == "O":
163             for row, col in [(0, 0), (0, grid_size - 1),
164                              (grid_size - 1, 0),
165                              (grid_size - 1, grid_size - 1)]:
166                 if self.board[row][col] == 0:
167                     self.board[row][col] = "O"
168                     (True, row, col)
169                     return (True, row, col)
170
171         # Place "O" in a non-corner side
172         for row, col in empty_positions:
173             if row not in [0, grid_size - 1]
174             and col not in [0, grid_size - 1]:
175                 self.board[row][col] = "O"
176                 return (True, row, col)
177
178         # Place "O" in any available space
179         for row, col in empty_positions:
180             self.board[row][col] = "O"
181             return (True, row, col)
182
183         return (False, -1, -1)
```

Now we can call all these functions for checking for who won and placing X's and O's on the board from the main game file, and the main game file is a lot cleaner.

In our initialize_game_values we construct the board as follows:

```
1     board = GameBoard(grid_size)
```

Then anywhere we use the board, we simply call its methods

Below is the call to the **run_better_algorithm_to_place_O** GameBoard method where we place the O sprite from the main game program. The algorithmic method on the board returns a tuple

indicating if we were able to find a place for the the piece on the board and if so what row and column it was placed.

```
(O_placed, rowo, colo) =
    board.run_better_algorithm_to_place_O()
if O_placed:
                letterO = LetterSprite(
                    'O', colo, rowo,
                    grid_width,
                    grid_height)
                group.add(letterO)
```

We can also access any internal attributes of the GameBoard class, like the winner of the game:

```
 if board.winner == "X":
        text = font.render('X Wins!', True, BLACK)
    elif board.winner == "O":
        text = font.render('O Wins!', True, BLACK)
    else:
        text = font.render('Draw!', True, BLACK)
```

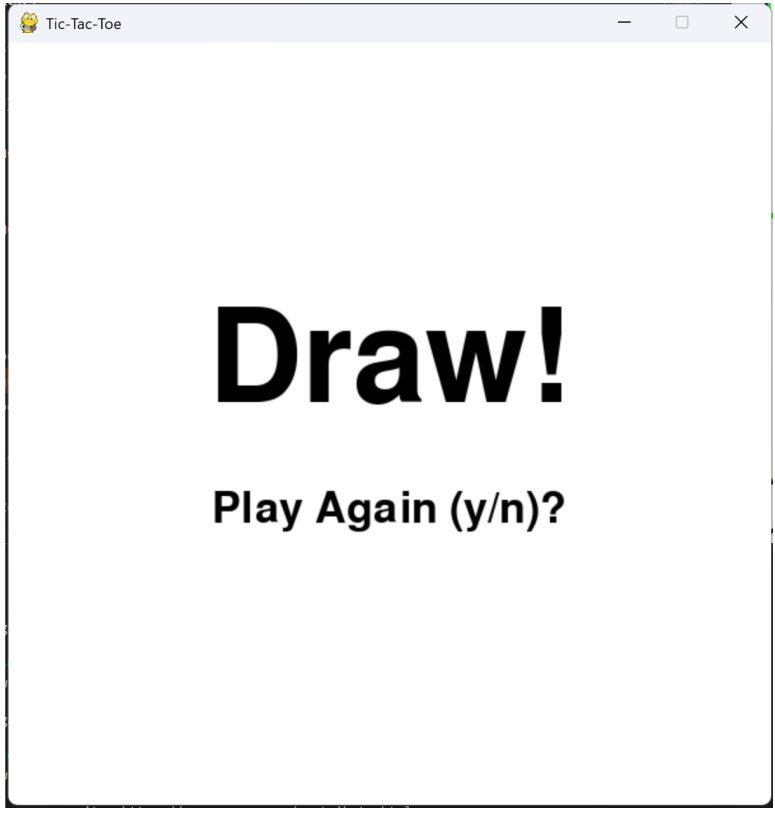
Looking at this code, its actually an opportunity to refactor into a method that returns the winner string for the Game Over Screen. So we'll add a new method get_winner_display_message to GameBoard:

```
 def get_winner_display_message(self):
        if self.winner == 'X':
            return 'X Wins!'
        elif self.winner == 'O':
            return 'O Wins!'
        else:
            return 'Draw!'
```

and then call it from the draw_game_over_screen function in our main pygame program.

```
def draw_game_over_screen():
    game_window.fill(WHITE)
    winnerMessage = board.get_winner_display_message()

    text = font.render(winnerMessage, True, BLACK)

    # get the width of the text so we can
    # center horizontally
    text_width = text.get_width()

    playAgainText = smallfont.render('Play Again (y/n)?',
      True, BLACK)

   # get the width of the play again prompt
   # so we can center it horizontally
    playAgainText_width = playAgainText.get_width()

    game_window.blit(
        text, (window_width/2 - text_width/2,
          window_height/2 - 100))

    game_window.blit(playAgainText,
        (window_width/2 - playAgainText_width/2,
        window_height/2 + 50))
```

This code refactoring took the responsibility of having to know how the winner was determined out of the main program and stuck it in a black box that we could call from our board object.

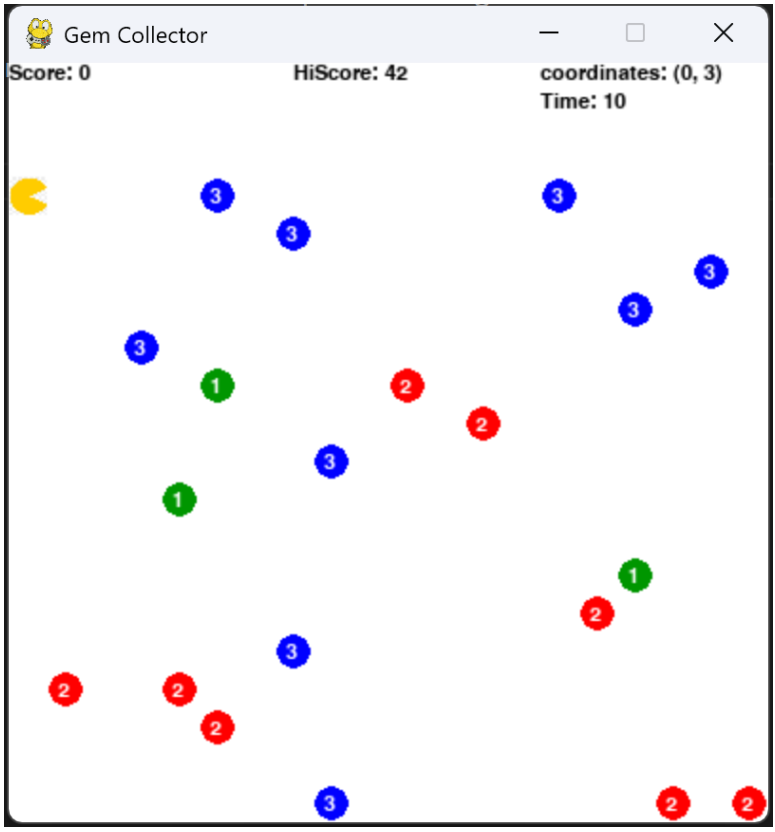


Conclusion

In this chapter, we explored the use of classes in pygame to enhance the organization and maintainability of your code. We achieved this by employing the GameBoard class to handle game logic and data management while assigning the responsibility of drawing to sprite

classes. Through their methods and properties, these classes were integrated into the main loop. Furthermore, we shifted much of the low-level drawing to Sprite classes, decluttering the main program. In the upcoming chapter, we will introduce a game where players must gather gems within a specified time frame using arrow keys. We will incorporate many concepts previously discussed in earlier chapters.

Chapter 6 - Stone Eater



Introduction

In the previous chapter we created a tic-tac-toe game to play against the computer. Stone Eater is a game you play against the clock! The object of the game is to eat as many valuable stones as you can

before the clock runs out. Each stone is worth eather 1,2, or 3 points, so you'll want to eat stones that have a higher value. Stone Eater introduces a few new concepts in our game. In this chapter we'll learn how to handle keyboard events, animate our player character, and play sound to add another dimension to our game.

The Game Design

The Classes

We will take advantage of classes for our game. There are multiple sprite classes we will use in the game, each sprite representing a game object. There will be a sprite used to draw the stone-eater as well as a sprite used to draw all the gems in the game. Also we will have sprites for each of the stats we use in the game: score, hi score, coordinates, and time. We will also create a general message sprite to post text like "play again?" Also, as we did in tic-tac-toe, we'll create a game board class that will control all the game logic and game state for when a player eats a stone. Below are list of the classes we just mentioned:

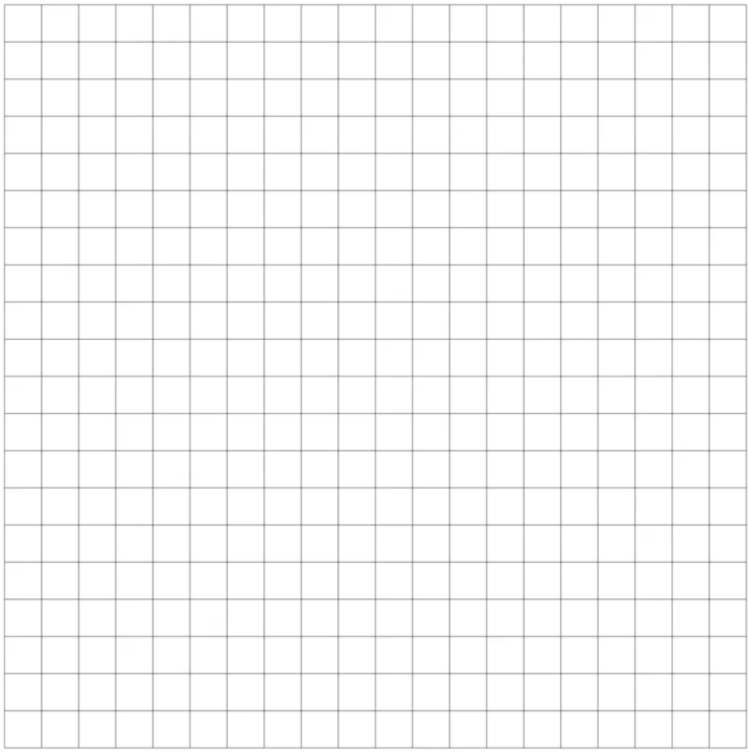
PlayerSprite StoneSprite GameBoard ScoreSprite HiScoreSprite TimeSprite CoordinateSprite MessageSprite

Game Layout

Like Tic-Tac-Toe, the stone eater game is layed out like a grid. The game board in this game is a 20x20 grid with each cell having a width of 20 pixels. When the player moves up, down, left, or right, the stone eater moves to the next adjacent cell in the grid. Gems are also placed randomly inside the grid in different cells.

Although the game board is a 20 x 20 grid, we only use the bottom 17 rows to leave room for the scoring sprites at the top of the game.

We could have done this differently, by placing the grid below the scores, but this method works as well as long as we limit the player from going above the 3rd row in the grid.



Initializing the Game

Before we start the game loop, like any game, we need to initialize the game pieces. The **initialize_game_state** function sets up the initial state for the Stone Eater game. This function starts by declaring several global variables that will be used throughout the game: **gems_collected**, **gems_score**, **start_time**, **times_up**, and **already_played**.

The function then empties the gem_group to remove any existing gems on the game board. It sets the **start_time** to the current time and sets **times_up** to False. The player's initial position is set to the 3rd row and the first column of the game board. The game time and score are updated to their starting values of time_limit and 0, respectively. The **gems_collected** and gems_score variables are set to 0, and **already_played** is set to False. Finally, the function calls the **initialize_board** method to reset the game board logic to its initial state.

```
def initialize_game_state():
    global gems_collected, gems_score, start_time,
     times_up, already_played
    print("Initializing game state")
    gem_group.empty()
    start_time = pygame.time.get_ticks()
    times_up = False
    player.row = player_limit
    player.column = 0
    player.update()
    game_time.update_time(time_limit, 0)
    score.update_score(0)
    gems_collected = 0
    gems_score = 0
    already_played = False

    game_board.initialize_board()


initialize_game_state()
```

Game Loop

As seen in tic-tac-toe the game is composed of receiving events and drawing sprites based on the state of the board and the events.

Below is the a rough architecture of the game loop which drives the entire game of stone eater.

```
while running:
  # Get the keyboard event to move the player
  # update the player sprite position if a key was pressed
  # determine if the player collided with a stone sprite
  # if the player collided,
  #    remove the stone and play a sound
  #
  # update the scoring sprites
  # draw all the gems,
  # if the time is up
  #   draw play again (y/n) message
  #   update high score
  #   play end of game music
  # else
  #     draw player and gametime sprites
  # draw all the gem sprites
  # draw all the scores sprites,
  # loop back
```

Now let us look at the real game loop. Below is the full code of the stone eater game loop which you'll notice looks similar to the tic-tac-toe game loop. Just as we described in our pseudocode, the loop handles events from the user, updates to the game board accordingly, and draws everything on the game board. It takes advantage of sprite groups to perform both the update and drawing of the sprites each time through the loop. The game loop also plays sounds when appropriate: we play a sound each time a stone is eaten and we play music when time runs out and the game is complete.

```
# Main game loop
running = True
while running:

    running = process_events()  # process the keyboard

    # Check if player has picked up a gem
    if game_board.check_for_gem(player) and
       (times_up == False):
        # gem found, update score
        gems_collected += 1
        gems_score += game_board.get_cell_value(
          player.row, player.column)
        score.update_score(gems_score)
        # remove the gem from the board and the gem sprite
        game_board.remove_gem(player)
        which_sprite = detect_collisions(player,
           gem_group, piece_size)
        remove_sprite_from_group(which_sprite, gem_group)
        got_coin_sound.play()

    # Update coordinates
    coordinates.update_coordinatees(player.row, player.co\
lumn)

    # Update time
    game_time.update_time(time_limit,
                pygame.time.get_ticks() - start_time)

    # Check if time is up
    if (pygame.time.get_ticks() -
       start_time > time_limit * 1000)
       and (times_up == False):
        times_up = True

```

```
36      # empty the screen
37      window.fill(WHITE)
38
39      # Check if the time up flag is set
40      if times_up:
41          # set the current score, to compare to
42          # the hi score
43          # and play the end of game music once
44          if already_played == False:
45              hi_score.current_score = gems_score
46              victory_sound.play()
47              already_played = True
48
49          gems_collected_message.update_message(
50              f'You collected {str(gems_collected)} gems!')
51
52          gems_collected_message.update()
53          gems_collected_message.draw(window)
54          play_again_message.draw(window)
55      else:
56          # draw the player and game time
57          player.draw(window)
58          game_time.draw(window)
59
60      # draw the gems
61      gem_group.draw(window)
62
63      # update the stats
64      # (score, hiscore, coords, time left)
65      score_group.update()
66
67      # draw the stats
68      score_group.draw(window)
69
70      # display all the graphics
```

```
71      # we just drew
72      pygame.display.flip()
```

The code takes full advantage of sprites and sprite groups. The gems are in a group and the scores are in a group, so they can be drawn at once.

Detecting Key Strokes

Inside our event loop, we check to see if the user pressed any of the arrow keys so we can move the stone eater left, right, up, or down depending on which arrow key the player pressed. We do this by looping through all the events in the pygame event queue and see if any of them are keydown. If they are, then we check to see which keyboard key was chosen and compare it to the keys we are interested in. For example we look to see if the up arrow (pygame.K_UP) was chosen.

Once we determined they picked an up arrow, we also check to see if the player is trying to go off the board because we don't want the player going beyond the game board. In the case of key up arrow, we limit the player to the third row of the grid so the player doesn't start moving onto the game statistics. Once we determine that the player is within the game board, then we move the player one cell in the direction of the key that was pressed. For an up arrow, we subtract one from the player row to move them up a row in the game board grid.

```
for event in pygame.event.get():
        if event.type == pygame.QUIT or
        (event.type == pygame.KEYDOWN and
         event.key == pygame.K_n and times_up == True):
            running = False
        elif event.type == pygame.KEYDOWN:
            # Check if player has moved
            if event.key == pygame.K_UP
              and player.row > player_limit:
                player.row -= 1
                player.update()
            elif event.key == pygame.K_DOWN
              and player.row < GRID_LENGTH - 1:
                player.row += 1
                player.update()
            elif event.key == pygame.K_LEFT and
              player.column > 0:
                player.column -= 1
                player.update()
            elif event.key == pygame.K_RIGHT and
              player.column < GRID_LENGTH - 1:
                player.column += 1
                player.update()
            elif event.key == pygame.K_y and
               times_up == True:
                initialize_game_state()
```

The Game Board

Like in tic-tac-toe, the game board in stone eater is used to place the gems, and also to track where the gems are located and determine if they have been eaten or not. Below are the methods of the Game Board class and their purpose.

init (constructor) **initialize_board** - places the initial stones on the

board **check-for-gem** - check if the row and column has a gem in it **remove_gem** - removes the gem from the board **get_cell_value** - get the value of the cell at the row and column specified

When we initialize the board, we place gems at random unoccupied spots on the board.

```
  def initialize_board(self):
        # fill the empty grid with a 20 x 20 matrix of 0's
        self.grid = []
        for i in range(self.grid_size):
            self.grid.append([])
            for j in range(self.grid_size):
                self.grid[i].append(0)

        # Place gems randomly on the self.grid
        num_gems = 20
        for i in range(num_gems):
            gem_placed = False
            while not gem_placed:
                row = random.randint(self.player_limit,
                   self.grid_size - 1)
                column = random.randint(0,
                  self.grid_size - 1)
                if self.grid[row][column] == 0:
                    self.grid[row][column]
                       = random.randint(1, 3)
                    gem_placed = True
                    # add stone sprites to the gem group
                    # as we place them on the board
                    self.gem_group.add(StoneSprite(
                        self.colors[
                         self.grid[row][column]-1],
                         row, column, self.piece_size,
                          self.grid[row][column]))
```

Below is the entire class that includes all the methods described above. The GameBoard class makes it easier to do all the game logic as it relates to the board and hides the internal grid mapping from the game loop so the game loop doesn't have to think about it.

```
class GameBoard:
    def __init__(self, size, piece_size,
      player_limit, gem_group):
        self.grid_size = size
        self.piece_size = piece_size
        self.player_limit = player_limit
        self.grid = []
        self.gem_group = gem_group
        # gem colors
        GREEN = (0, 150, 0)
        RED = (255, 0, 0)
        BLUE = (0, 0, 255)
        self.colors = [GREEN, RED, BLUE]

    def initialize_board(self):
        # fill the empty grid with a 20 x 20 matrix of 0's
        self.grid = []
        for i in range(self.grid_size):
            self.grid.append([])
            for j in range(self.grid_size):
                self.grid[i].append(0)

        # Place gems randomly on the self.grid
        num_gems = 20
        for i in range(num_gems):
            gem_placed = False
            while not gem_placed:
                row = random.randint(self.player_limit,
                  self.grid_size - 1)
                column = random.randint(
                   0, self.grid_size - 1)
```

```
                if self.grid[row][column] == 0:
                    self.grid[row][column] =
                      random.randint(1, 3)
                    gem_placed = True
                    # add stone sprites to the gem group
                    # as we place them on the board
                    self.gem_group.add(StoneSprite(
                        self.colors[
                         self.grid[row][column]-1],
                         row, column,
                         self.piece_size,
                         self.grid[row][column]))

    def check_for_gem(self, player):
        if self.grid[player.row][player.column] > 0:
            return True
        else:
            return False

    def remove_gem(self, player):
        self.grid[player.row][player.column] = 0

    def get_cell_value(self, row, column):
        return self.grid[row][column]
```

Game Sprites

Each game sprite draws an object in the game. All sprites have the following contract in their structure:

```
class MySprite
    __init__
    def update(self)
    def draw(self, surface)
```

We don't always have to implement **update**, because its possible that the game sprite doesn't move or change in any way. For example, stones, once they are created, do not change graphically or in their position, so there is no reason to update them. The player sprite, on the other hand is moving with each key stroke, so it must be updated constantly in the game upon detecting an arrow key. The **draw** function is used to draw the sprite, so it is always used. Let's look at the sprite for a stone and the sprite for a player:

The stone sprite below, has most of it's code in the constructor. That is because once we define its image, it never changes. Even drawing the sprite is predetermined early on in the constructor. All the draw function has to do is blit the image created in the construct and blit the value of the gem (1,2, or 3). The update function does absolutely nothing if its called.

Let's take a closer look at the constructor (**__init__**) since this is where the meat of the class is located. The constructor creates a font object, and starts with a blank 20x20 image. It then fills the image with white and draws a filled circle onto its surface. The color of the circle will depend on the color passed into the sprite (either, red, green, or blue). After the constructor draws the circle, it renders the white font on top of the circle and blits it into the center of the circle. Finally, it moves the rectangle to the row and column passed into the constructor. Moving the rectangle will move the entire stone to the row and column position on the board.

```
1  class StoneSprite(pygame.sprite.Sprite):
2      def __init__(self, color, row, column,
3          piece_size, gem_value):
4           super().__init__()
5           WHITE = (255, 255, 255)
6           BLACK = (0, 0, 0)
7           small_font = pygame.font.Font(None, 16)
8
9           self.row = row
```

```
10          self.column = column
11
12          self.piece_size = piece_size
13          # Create a surface for the sprite
14          self.image = pygame.Surface(
15             [piece_size, piece_size])
16          self.image.fill(WHITE)
17
18          # Draw the rectangle to the sprite surface
19          pygame.draw.circle(self.image, color,
20             (piece_size/2, piece_size/2),
21             int(piece_size/2.2))
22          self.gem_value = small_font.render(
23              str(gem_value), True, WHITE)
24          self.image.blit(self.gem_value, (piece_size/3,
25             piece_size/4))
26          self.rect = self.image.get_rect().move(
27              column*piece_size,
28              row*piece_size)
29
30
31      def update(self):
32          pass
33
34      # Draw the sprite on the screen
35      def draw(self, surface):
36          surface.blit(self.image, self.rect)
37          surface.blit(self.gem_value, self.rect)
```

Now let's look at the stone eater sprite. In this sprite we introduce a new library called pyganim (pygame animation). You will need to install the pyganim library using pip install:

```
1  pip install pyganim
```

The animation libary makes it easier for us to animate our stone

eater without having to handle it in the game loop. The way we will animate the stone eater is to have it open and close its mouth to have it look like its eating stones (kinda like PacMan !). We only need two images to do this, the eater with their mouth open and the eater with their mouth closed.

The pygame animation library let's us animate this easily using the PygAnimation method which take the images paired with the time of the frame in milliseconds. For our pacman we alternate between the two images every 250 milliseconds or a quarter of a second. This will give us the desired effect of the eater open and closing its mouth. Also, because our images are rather large, we need to scale them down to the size of the grid cell. We could either do this manually, by resizing the images or we can use the scale function provided to us by the animation library. We chose to reduce the size using the scale function.

In order to play the animation, we simply call play on the animation object and it will run the animation of openign and closing of the eaters mouth throughout the entire game.

```
import pygame
import pyganim

class PlayerSprite(pygame.sprite.Sprite):
    def __init__(self, row, column, piece_size):
        super().__init__()

        self.row = row
        self.column = column
        self.piece_size = piece_size
        self.anim = pyganim.PygAnimation(
            [("pacopen.png", 250), ("pacclose.png", 250)])
        self.anim.scale((piece_size, piece_size))
        self.anim.play()
        self.rect = pygame.Rect(
            column*piece_size, row*self.piece_size,
            self.piece_size, self.piece_size)

    def update(self):
        self.rect = self.anim.getRect().move(
            self.column*self.piece_size,
            self.row*self.piece_size)

    def draw(self, surface):
        self.anim.blit(surface, self.rect)
```

Notice in the eater object, the update function is not empty. The reason for this is because every time an arrow key is pressed in the game, we update the players row or column position. We must call update on the player object in order to move its position rectangle to the new grid cell location determined by the key pressed. To draw the player, we simply blit the animated object to the game surface and place it at the rect position on the board.

The other sprite that we frequently update is the time sprite. This sprite will show the remaining time the user has left in the game. In

the main loop we update the time remaining each iteration through the loop:

```
 # Update time
    game_time.update_time(time_limit,
         pygame.time.get_ticks() - start_time)
```

The TimeSprite updates its internal time, and later uses that time in the update function to render that time into a font object containing the time left:

```
import pygame


class TimeSprite(pygame.sprite.Sprite):
    def __init__(self):
        super().__init__()
        BLACK = (0, 0, 0)
        self.time = 0
        self.small_font = pygame.font.Font(None, 16)
        self.image = self.small_font.render(
            f'Time: {self.time}', True, BLACK)
        self.rect = self.image.get_rect().move(280, 15)


    def update(self):
        BLACK = (0, 0, 0)
        # update the time image
        self.image = self.small_font.render(
            f'Time: {self.time}',
            True, BLACK)
        self.rect = self.image.get_rect().move(280, 15)


    def draw(self, surface):
```

```
25          # Draw the time on the screen
26          surface.blit(self.image, self.rect)
27
28      def update_time(self, time_limit, time_in_millisecond\
29  s):
30
31          # calculate the time remaining
32          calculated_time = int(time_limit -
33            (time_in_milliseconds / 1000))
34
35          # no need to go below 0
36          if calculated_time < 0:
37              calculated_time = 0
38          self.time = calculated_time
```

The ScoreSprite is similar to the TimeSprite. It contains a function to update the score, and then in the update function, it creates the image for drawing the score using self.score

```
1   import pygame
2
3   class ScoreSprite(pygame.sprite.Sprite):
4       def __init__(self):
5           super().__init__()
6           BLACK = (0, 0, 0)
7           self.score = 0
8           self.small_font = pygame.font.Font(None, 16)
9           # need initial image to determine rect
10          self.image = self.small_font.render(
11              f'Score: {self.score}', True, BLACK)
12          # get rect bounding the score
13          self.rect = self.image.get_rect().move(0, 0)
14
15      def update(self):
16          BLACK = (0, 0, 0)
```

```
17          self.image = self.small_font.render(
18              f'Score: {self.score}', True, BLACK)
19          # recalculate the rectangle
20          # since the image changed
21          self.rect = self.image.get_rect().move(0, 0)
22
23      def draw(self, surface):
24          # Draw the sprite on the screen
25          surface.blit(self.image, self.rect)
26
27
28      def update_score(self, score):
29          self.score = score
```

Tracking the High Score

The **HiScoreSprite** is in charge of tracking the high score of the game. It is different than the Score Sprite in that it remembers the highest score and only updates when a higher score is reached. The score is stored in a file, so its remembered, even if the user turns off their computer.

The **HiScoreSprite** class starts by initializing the score variable, which is the highest score achieved by a player. It opens a file called hiscore.txt and reads the score stored in it, converting it to an integer and saving it as the score variable.

The class then sets up the display of the score, creating a small font and rendering the text "HiScore: [score]" to an image. The rect attribute is set to a position on the screen, in this case (150, 0). The **update** method calls the **update_high_score** method to update the hi score. Similar to the Score sprite, the **draw** method draws the hi score on the screen.

The **update_high_score** method is used to update the highest score in the game. It compares the new score to the current highest score,

and if the new score is higher, it updates the score variable and the text displayed on the screen, writes the new score to the hiscore.txt file, and saves it. If the new score is not higher, it does nothing.

```
import pygame


class HiScoreSprite(pygame.sprite.Sprite):
    def __init__(self):
        super().__init__()
        WHITE = (255, 255, 255)
        BLACK = (0, 0, 0)
        f = open('files/hiscore.txt', 'r')
        self.hi_score = int(f.read())
        print(f'read hi score of {self.hi_score}')
        f.close()
        self.current_score = -1
        self.small_font = pygame.font.Font(None, 16)
        self.image = self.small_font.render(
            f'HiScore: {self.hi_score}', True, BLACK)
        self.rect = self.image.get_rect().move(150, 0)
    # Draw the sprite on the screen

    def update(self):
        self.update_high_score(self.current_score)

    def draw(self, surface):
        surface.blit(self.image, self.rect)

    def update_high_score(self, score):
        BLACK = (0, 0, 0)
        if self.hi_score < score:
            self.hi_score = score
            self.image = self.small_font.render(
                f'HiScore: {self.hi_score}', True, BLACK)
            self.rect = self.image.get_rect().move(150, 0)
```

```
33              print(f'write hi score of {self.hi_score}')
34              f = open('files/hiscore.txt', 'w')
35              f.write(str(score))
36              f.close()
37          else:
38              pass
```

Detecting Collisions

There is a built in function called **rect.colliderect** to detect collisions between sprites. We can loop through all the stones on the board and use the colliderect function to determine if one of the stones collides with the player rect on the board.

The function we will create is called detect_collisions and it takes three arguments:

playerSprite: a PlayerSprite object that represents the player character. **group**: a group of StoneSprites representing the stones on the board **piece_size**: an integer that represents the length and width of each sprite measured in number of pixels.

The purpose of the function is to check if the player sprite has collided with any of the sprites in the group.

The function starts by looping over all the sprites in the group using the **.sprites()** method. For each sprite, it creates a rectangle representation of the player sprite (**playerRect**) using the **pygame.Rect** constructor. The position of the player sprite is calculated by multiplying its row and column properties by piece_size, and the size of the rectangle is set to (piece_size, piece_size).

The colliderect method of the playerRect object is then called on the rect property of the current StoneSprite being looped over. This method returns True if the player sprite rectangle and the current sprite rectangle intersect, meaning that a collision has occurred.

If a collision is detected, the current stone sprite is returned from the function. If the loop completes without finding a collision, None is returned to indicate that no collision has occurred.

```
def detect_collisions(playerSprite: PlayerSprite, group: \
pygame.sprite.Group, piece_size: int):

    for sprite in group.sprites():
        # detect collision with a sprite
        playerRect = pygame.Rect((playerSprite.column *
            piece_size,
            playerSprite.row * piece_size),
            (piece_size, piece_size))
        if playerRect.colliderect(sprite.rect):
            return sprite
    return None
```

Why didn't we just pull the playerRect right of the sprite itself? Because we used the animation library and scaled the images of the stone eater, the rect for some reason did not also scale itself as well. In order to get around this issue, we can simply recreate the stone eater position rect based on the row, column and the piece_size.

Space Invasion in PyGame

Introduction

Space Invaders is a classic video game developed by Taito Corporation in 1978. The game was designed by Tomohiro Nishikado, who was inspired by the hit game Breakout and the sci-fi classic movie "Star Wars".

The game was initially released in Japan, but it quickly gained popularity around the world, becoming a cultural phenomenon in the 1980s. Its simple gameplay and iconic 8-bit graphics made it a favorite among players and established it as a classic of the arcade era.

In the game, players control a spaceship that must defeat waves of alien invaders that descend from the top of the screen. The game's difficulty increases as the player progresses, with faster and more aggressive alien attacks.

Space Invaders was not only a hit in arcades but also helped launch the video game industry and sparked a wave of space-themed games. It has been ported to numerous platforms over the years, including home consoles and personal computers, ensuring its continued popularity and status as a gaming icon.

In this chapter, we'll describe how to recreate the classic game using pygame.

How to play

The goal of the game is to defeat waves of alien invaders that are descending from the top of the screen by shooting them with a laser cannon controlled by the player. The player must move the cannon left or right to dodge the alien's attacks and to line up shots. The cannon can only fire one bullet at a time, so players need to time their shots carefully to avoid being overwhelmed.

The aliens move back and forth across the screen and gradually move closer to the player's cannon. If the aliens reach the bottom of the screen, the game is over. The player must try to destroy all the aliens before they reach the bottom to progress to the next level.

As the player progresses through the levels, the aliens become faster and more aggressive, and new types of aliens with different abilities appear. Some aliens move more quickly or unpredictably, while others require multiple shots to be defeated.

The game has a limited number of lives, so players must try to avoid being hit by the alien's attacks. If the player's cannon is hit by an alien's laser beam, they lose a life, and the game ends when all lives are lost.

Overall, Space Invaders is a simple but addictive game that requires quick reflexes and precise timing to succeed.


MIDWAY'S
a Bally Co.
SPACE
INVADERS
SCORE<1> HI-SCORE SCORE<2>
490 3490
PLAY
SPACE INVADERS
SCORE ADVANCE TABLE
=? MYSTERY
=30 POINTS
=20 POINTS
=10 POINTS
1 PLAYER
START GAME
2 PLAYER
LASER BASE
CONTROL
FIRE
LASER
DEFENSE INSTRUCTIONS

The Main Loop

Similar to how we set up our main loop for the stone eater game, we need to do the same sort of thing for space invasion. The loop consists of responding to keyboard events in the game as well as drawing the various sprites and detecting collisions between them. To make the main loop easier to read, we've broken down the code to several high level functions.

The first high level function, **process_events**, processes the keyboard events for the player moving and shooting. Here are the other high level functions and their descriptions:

handle_scoring - updates the current score on the screen based on the players score. This function also updates the high score, and the lives indicator.

handle_alien_movement - This function guides the aliens across the screen and helps the aliens reverse direction when they hit the edge of the screen. They also move down every time they hit the edge.

handle_player_movement - handles player movement according to the arrow key they are pressing. The left arrow moves the player left until they release the key and same with the right arrow.

handle_bullet - This function handles an active bullet coming from the player trying to hit the aliens. It moves the bullet up the screen at a certain speed each time through the game loop. It also checks to see if the bullet has collided with an alien with the function **handle_alien_hit**.

check_for_bomb_activation - this function checks based on a randomly generated value whether or not an alien has released a bomb. If an alien has released a bomb it is recorded in an array to handle its movement later.

handle_active_bombs loops through the bomb array and draws the active bombs. Also checks to see if the bomb hit the player by

calling **handle_player_hit**

draw_aliens - Draws all the aliens. Aliens are drawn by looping through the rows of aliens stored in **alien_groups** and then looping through each alien in the row. A row is a pygame group.

handle_alien_speedup - As the player kills more and more aliens, the aliens speed up after a certain threshold. Currently that threshold is when there are only 25 aliens, then when there are only 5 aliens and finally when there is just 1 alien left.

```
while running:
    (player_x, player_y) = player.position
    window.fill(BLACK)

    running = process_events()  # process the keyboard

    if (game_over):
        show_game_over_prompt()
        pygame.display.flip()
        continue

    # scoring
    handle_scoring(window, score, player_score)

    # move the aliens
    handle_alien_movement()

    # move the player
    handle_player_movement(window_width, player_left,
      player_right, player, player_x)

    # move the bullet
    if bullet_active:
        handle_bullet(bullet, bullet_active)


```

```
    # check for bomb activation every 2 seconds
    check_for_bomb_activation()

    # update active bombs
    handle_active_bombs(active_bombs)

    # draw the aliens and check for removal
    draw_aliens(window, alien_groups)


    # check if its time to speed up the aliens
    # based on the number of aliens left
    handle_alien_speedup(total_aliens)

    # show the display
    pygame.display.flip()

    # update the game time
    game_time = pygame.time.get_ticks() - start_time
```

Game Sprites

As with the stone eater, its easiest to build space invasion by creating sprites. This game contains the following sprites.

ImageSprite - The base class for all sprites that load an image and track position **PlayerSprite** - The player cannon you control for shooting aliens **BombSprite** - The sprite representing the image of the bomb dropped by an alien **BulletSprite** - The sprite showing the bullet that is shot at the invaders **InvaderSprite** - Draws the invader using two images to animate its movement. **MessageSprite** - Used to draw the game over message **ScoreSprite** - Draws the score of the game at the top of the screen **HighScoreSprite** - Same sprite we used in the stone eater to track the high score **LivesSprite** - Draws the number of lives left for the player

Player Sprite

Let's first take an in depth look at the player sprite. This code defines a class called PlayerSprite that extends ImageSprite. The PlayerSprite class represents a player-controlled sprite that can move left or right, be killed, and explode. It has the following attributes:

dead: a Boolean flag indicating whether the sprite is dead or not. **speed**: a float representing the speed at which the sprite moves left or right. **death_time**: an integer representing the time at which the sprite was killed. **animate_explosion**: a PygAnim object representing an animation of the sprite exploding.

The PlayerSprite class has the following methods:

init(self, name, x, y): the constructor for the PlayerSprite class. It initializes the sprite's position and sets its attributes to their initial values. **update**: a method that updates the sprite's position. **kill**: a method that kills the sprite, triggering an explosion animation. **draw**: a method that draws the sprite on a given surface. If the sprite is not dead, it calls the superclass's draw() method to draw the sprite image. If the sprite is dead, it plays the explosion animation and checks if enough time has passed for the explosion to be complete. **move_left**: a method that moves the sprite to the left by adjusting its position. **move_right**: a method that moves the sprite to the right by adjusting its position.

```
import pygame
import pyganim
from ImageSprite import ImageSprite

class PlayerSprite(ImageSprite):
    def __init__(self, name, x, y):
        super().__init__(name, x, y)
        self.dead = False
        self.speed = .1
        self.death_time = 0
        self.animate_explosion = pyganim.PygAnimation(
            [("images/shipexplosion/frame1.gif", 250),
             ("images/shipexplosion/frame2.gif", 250),
             ("images/shipexplosion/frame3.gif", 250),
             ("images/shipexplosion/frame4.gif", 250),
             ("images/shipexplosion/frame5.gif", 250),
             ("images/shipexplosion/frame6.gif", 250),
             ("images/shipexplosion/frame7.gif", 250),
             ("images/shipexplosion/frame8.gif", 250),],
             loop=False)

    # just call the super class to adjust the rect
    def update(self):
        super().update()

    # Draw the sprite on the screen
    def kill(self):
        self.animate_explosion.play()
        self.dead = True
        self.death_time = pygame.time.get_ticks()


    def draw(self, surface):
        if not self.dead:
            super().draw(surface)
```

```
36          else:
37              self.animate_explosion.blit(surface,
38                 self.rect)
39              if (pygame.time.get_ticks() -
40                self.death_time) > 5000:
41                  self.dead = False
42
43      def move_left(self):
44          (x, y) = self.position
45          self.position = (x - self.speed, y)
46
47      def move_right(self):
48          (x, y) = self.position
49          self.position = (x + self.speed, y)
```

Player Explosion

The animate_explosion object uses the pyganim library to handle the animating the ship explosion by quickly drawing each of the 8 frames of the explosion once through. The animation is initialized with all the information it needs to play the explosion frames and how long each frame will be shown:

```
1   self.animate_explosion = pyganim.PygAnimation(
2               [("images/shipexplosion/frame1.gif", 250),
3                ("images/shipexplosion/frame2.gif", 250),
4                ("images/shipexplosion/frame3.gif", 250),
5                ("images/shipexplosion/frame4.gif", 250),
6                ("images/shipexplosion/frame5.gif", 250),
7                ("images/shipexplosion/frame6.gif", 250),
8                ("images/shipexplosion/frame7.gif", 250),
9                ("images/shipexplosion/frame8.gif", 250),],
10               loop=False)
```

To play the animation, we simply call the **play** method on animate_explosion inside our kill method:

```
1  def kill(self):
2          self.animate_explosion.play()
3          self.dead = True
4          self.death_time = pygame.time.get_ticks()
```

Invader Sprite

Next lets take a look at the InvaderSprite that draws the animated alien moving across the screen. The InvaderSprite class inherits from pygame.sprite.Sprite, which is a base class for all sprites in Pygame. The **init**() method initializes various instance variables such as the two image sprites (name1 and name2) that are used to draw and animate the alien, the explosion image sprite, the parent row represented by a sprite group, the speed of the alien, its current direction (either left or right), its initial position, and its points. The **update()** method updates the position of the two image sprites representing the alien based on its current position. The draw() method draws the current image sprite on the game surface. The **move_left()**, **move_right()**, and **move_down**() methods move the alien left, right, or down respectively. The **switch_image()** method switches between the two image sprites of the alien depending on the image number passed into this method. The **get_width()** and **get_height()** methods return the width and height of the current image sprite of the alien. The **kill()** method switches the image sprite to the explosion sprite and marks the alien as dead.

```
import pygame
from ImageSprite import ImageSprite
from BombSprite import BombSprite


class InvaderSprite(pygame.sprite.Sprite):
    def __init__(self, name1, name2, x, y, parent, points\
):
        super().__init__()
        self.imageSprite1 = ImageSprite(name1, x, y)
        self.imageSprite2 = ImageSprite(name2, x, y)
        self.explosion =    ImageSprite('explosion', x, y)
        self.imageSprite = self.imageSprite1
        self.parent = parent
        self.speed = .01
        self.currentDirection = 'right'
        self.position = (x, y)
        self.rect = self.imageSprite.image.get_rect()
          .move(self.position)
        self.dead = False
        self.death_time = 0
        self.bomb_active = False
        self.points = points

    # update the position of the 2 sprites
    # representing the alien
    def update(self):
        self.imageSprite.rect = self.imageSprite
            .image.get_rect()
            .move(self.position)
        self.imageSprite1.rect = self.imageSprite.rect
        self.imageSprite2.rect = self.imageSprite.rect

    # Draw the sprite on the screen

```

```
36      def draw(self, surface):
37          self.imageSprite.draw(surface)
38
39      def move_left(self):
40          (x, y) = self.position
41          self.position = (x - self.speed, y)
42
43      def move_right(self):
44          (x, y) = self.position
45          self.position = (x + self.speed, y)
46
47      def move_down(self):
48          (x, y) = self.position
49          self.position = (x, y + 10)
50
51      # switch between the 2 images representing the alien
52      def switch_image(self, imageNumber):
53          if self.dead == True: return
54          if (imageNumber == 1):
55              self.imageSprite = self.imageSprite1
56          else:
57              self.imageSprite = self.imageSprite2
58
59      def get_width(self):
60          return self.imageSprite.get_width()
61
62      def get_height(self):
63          return self.imageSprite.get_height()
64
65      def kill(self):
66          self.imageSprite = self.explosion
67          self.imageSprite.draw(self.imageSprite.image)
68          self.imageSprite.update()
69          self.dead = True
70          self.death_time = pygame.time.get_ticks()
```

How is the alien animated as it moves?

The InvaderSprite class uses a trick here to perform the animation of the open close of the alien. At any given time, the self.imageSprite property holds a reference to either the alien open claw image or the alien closed claw image. The program either passes a 1 or 0 into the **switch_image** method and assigns the image sprite accordingly two one of the two images. When it comes time to draw the alien, whatever the imageSprite is assigned to at the time, will be the one that gets drawn on the surface.

Bullet Sprite

When the player hits the up arrow, a green bullet is made active that can shoot an alien if it collides with one. The bullet image is created simply by filling the bullet's rectangular surface with green color.

```
class BulletSprite(pygame.sprite.Sprite):
    def __init__(self, x, y, bullet_width,
           bullet_height, speed):
        super().__init__()
        WHITE = (255, 255, 255)
        GREEN = (0, 255, 0)
        BLACK = (0, 0, 0)
        small_font = pygame.font.Font(None, 16)

        self.position = (x, y)
        self.speed = speed

        # Create a surface for the sprite
        self.image = pygame.Surface(
           [bullet_width, bullet_height])
        self.image.fill(GREEN)
```

```
17          # Draw the rectangle to the sprite surface
18          self.rect = self.image.get_rect().move(x, y)
19
20      # move the sprite according to the bullet's position
21      def update(self):
22          (x, y) = self.position
23          self.rect = self.image.get_rect().move(x, y)
24
25      # Draw the sprite on the screen
26      def draw(self, surface):
27          surface.blit(self.image, self.rect)
```

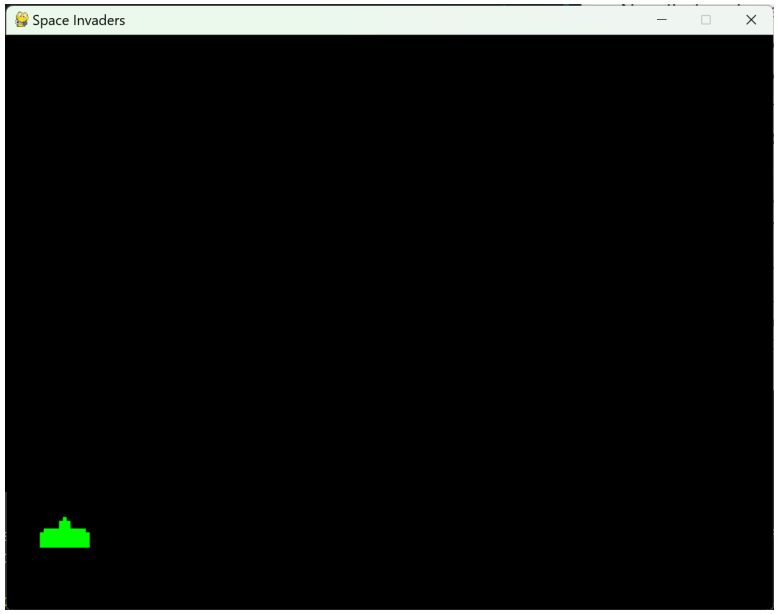
Bomb Sprite

The bomb sprite is the sprite that drops down from the aliens. It is drawn to the image surface as a series of diagnol white lines used to form the shape of a lightning bolt.

```
1   class BombSprite(pygame.sprite.Sprite):
2       def __init__(self, x, y, bullet_width,
3          bullet_height, speed, parent):
4           super().__init__()
5           WHITE = (255, 255, 255)
6           GREEN = (0, 255, 0)
7           BLACK = (0, 0, 0)
8           small_font = pygame.font.Font(None, 16)
9
10          self.position = (x, y)
11          self.speed = speed
12          self.parent = parent
13
14          # Create a surface for the sprite
15          self.image = pygame.Surface(
16            [bullet_width, bullet_height])
```

```
17          pygame.draw.lines(self.image, WHITE, True,
18              [(0, 0), (5, 5), (0, 10), (10, 15)], 1)
19
20          # Draw the rectangle to the sprite surface
21          self.rect = self.image.get_rect().move(x, y)
22
23      # update the bomb according to the current position
24      def update(self):
25          (x, y) = self.position
26          self.rect = self.image.get_rect().move(x, y)
27
28      # Draw the sprite on the screen
29
30      def draw(self, surface):
31          surface.blit(self.image, self.rect)
```

Moving the Player



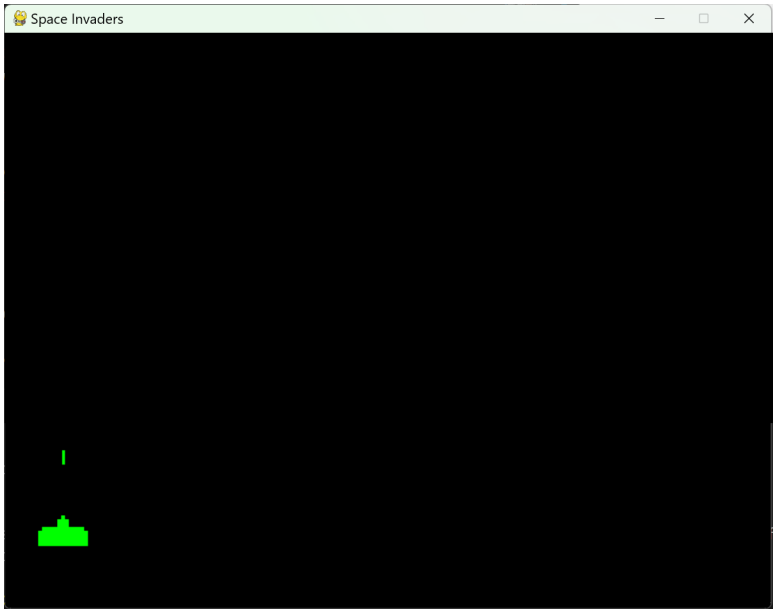
Now that we have our player sprite, we can move the player according to key presses. In our process_events method we'll capture the key presses. If the user hits the left arrow, we'll mark a flag that says the player is moving left. When they release the left arrow, the flag will be cleared. The same thing happens for the right arrow. If the user hits the right arrow, a flag will be marked to indicate the player is moving right. Once the user releases the right arrow, this flag will be cleared.

```
def process_events():
    global player_left, player_right
    (player_x, player_y) = player.position
    running = True
    for event in pygame.event.get():
        if event.type == pygame.KEYDOWN:
            # Check if player has moved
            if event.key == pygame.K_LEFT
              and player_x > 0:
                player_left = True
                player_right = False
            elif event.key == pygame.K_RIGHT and
               player_x < window_width:
                player_right = True
                player_left = False
        elif event.type == pygame.KEYUP:
            player_left = False
            player_right = False
    return running
```

Inside our game loop, we use the **player_left** and **player_right** flags set by the process_events method to move the player. We consolidate the behavior of the player in a method called **handle_player_movement** which takes the parameters necessary to move the player sprite. Note that we check the boundaries and if the player will go beyond the boundaries of the screen, we don't allow movement. Also note that if the player is marked dead, we don't need to move it. A player is moved at a rate according to the speed of the player. Once the speed is added or subtracted from the player position, the player sprite is updated and redrawn.

```
def handle_player_movement(window_width, player_left,
  player_right, player, player_x):
    if (player.dead):
        pass
    elif player_left:
        if (player_x - player.speed) > 0:
            player.move_left()
    elif player_right:
        if (player_x + player.speed) <
            window_width - player.get_width():
            player.move_right()

    player.update()
    player.draw(window)
```

Firing the bullet



The bullet is fired using the up arrow key. Once a bullet is fired, it cannot be refired until the bullet has either hit an alien or moved past the top of the screen. The bullet is activated by the up arrow, so we look for this in our process_event method:

```
def process_events():
    global player_left, player_right, bullet_active
    (player_x, player_y) = player.position
    running = True
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.KEYDOWN:
            # Check if player has moved
            if event.key == pygame.K_UP:
```

```
                if bullet_active == False:
                    bullet_active = True
                    bullet.position = (player_x + 30,
                       player_y - 20)
                    bullet_fire_sound.play()
                    ...
    return running
```

Once we've activated the bullet, we can handle its state and movement in a method in the main loop called **handle_bullet**. In the main loop, if the bullet is active, we call handle_bullet to draw the moving bullet. Handle bullet takes the bullet sprite and bullet_active flag we set in the process_events method. The bullet's y-position is set by subtracting the bullet speed from the bullet's current y-position and updating the bullet's position. If the bullet_y position is off the top of the screen (at y=0), we set the bullet_active flag to false

```
def handle_bullet(bullet, bullet_active):
    (bullet_x, bullet_y) = bullet.position
    bullet_y = bullet_y - bullet.speed
    bullet.position = (bullet_x, bullet_y)
    bullet.update()
    bullet.draw(window)
    if (handle_alien_hit(bullet_x, bullet_y)):
        bullet_active = False
        bullet.position = (0, 0)

    if (bullet_y < 0):
        bullet_active = False

    return bullet_active
```

Checking for alien hits

The **handle_bullet** method also checks to see if we hit an alien by calling the **handle_alien_hit** method with the bullet's current coordinates. handle_alien_hit not only checks all the aliens to see if any one of them was hit by the bullet, it also handles killing the alien. handle_alien_hit loops through all the alien rows and each alien in each row and checks if the bullet position is within an alien target. If it is, the alien sprite is killed and an explosion sound is played. Also the player's score is updated

```
def handle_alien_hit(bullet_x, bullet_y):
    global gems_collected, player_score, bullet,
       alien_groups
    for alien_group in alien_groups:
        for alien in alien_group:
            (x, y) = alien.position
            if bullet_x > x and
              and bullet_x < x + alien.get_width()
              and bullet_y > y and
                  bullet_y < y + alien.get_height():
                alien.kill()
                alien.death_time = pygame.time.get_ticks()
                alien_dying.play()
                player_score += alien.points
                return True
    return False
```

When an alien dies, it is replaced with an alien explosion image and marked as dead as shown in the InvaderSprite below. Also the time of death is marked so we can keep the explosion going for a set amount of time.

```
class InvaderSprite(pygame.sprite.Sprite):
    ...
    def kill(self):
        self.imageSprite = self.explosion
        self.imageSprite.draw(self.imageSprite.image)
        self.imageSprite.update()
        self.dead = True
        self.death_time = pygame.time.get_ticks()
```

In our main loop, we call **check_for_removal** each time through the loop. If an alien has been dead for more than 1/4 second(or 250 milliseconds), then we remove it from the row. If all aliens have been eliminated from the row, then we remove the row itself.

```
def check_for_removal(alien):
        if alien.death_time > 0
          and alien.death_time + 250 <
             pygame.time.get_ticks():
          alien.parent.remove(alien)
          if (len(alien.parent) == 0):
              alien_groups.remove(alien.parent)
```

Drawing the aliens

In order to detect alien hits, you need to have aliens to hit! In this section we'll describe how the aliens are drawn. We already looked at the invader sprite, the next step is to draw all the different kinds of invaders in different rows on the screen using the InvaderSprite class. Notice that the bottom two rows of invaders are both the same sprite and are worth 10 points, and the 2nd and third row are a different type of invader worth 20 points. The top row is also unique invader worth 30 points.

Initially we'll create the configuration of aliens you see in the figure above using the **create_aliens** method. In our game loop we'll update the alien movement using the **handle_alien_movement** function and we'll draw the aliens using the **draw_aliens** method.

create_aliens draws all 5 rows of aliens. The code defines a list

called alien_names that contains the names of different types of aliens in the game. Each type of alien name refers to the image of the alien created with the InvaderSprite. Their are two file images per InvaderSprite: the open alien sprite and the closed alien sprite ending in a c.

The function create_aliens() is used to create the actual aliens in the game. It starts by creating an empty list called alien_groups.

Next, the function uses a loop to create five rows of aliens. Within each row, the function uses another loop to create 11 aliens in that row. When creating an alien, it constructs an InvaderSprite with both the open and close versions of a particular alien.

The InvaderSprite constructor also assigns each alien a position on the screen to the InvaderSprite, which is determined by its row and column in the grid of aliens. Also assigned in the constructor is the parent group and the points the alien is worth when its hit. Each alien is added to a pygame.sprite.Group() object, which is a container for multiple sprites in Pygame.

Finally, each pygame.sprite.Group() object is added to the alien_-groups list, which is used to keep track of all the alien rows in the game.

Overall, this code sets up the different types of aliens in a game and creates them on the screen in a grid-like pattern using Pygame's sprite functionality.

```
## dictionary for scoring points
score_dict =  {
    'invader1': 30,
    'invader2': 20,
    'invader3': 10
}

alien_names = ['invader1', 'invader2', 'invader2', 'invad\
er3', 'invader3' ]
```

```
10
11  def create_aliens():
12      global alien_groups
13      alien_groups = []
14      for i in range(0, 5):
15          alien_group = pygame.sprite.Group()
16          for j in range(0, 11):
17              alien = InvaderSprite(alien_names[i],
18                  alien_names[i] + 'c',
19                  30 + (j * 60), 60 + i*60, alien_group,
20                  score_dict[alien_names[i]])
21              alien_group.add(alien)
22          alien_groups.append(alien_group)
```

Well we've painted our rows of aliens, now how do we move them? For that we call **handle_alien_movement** in our main loop. This method starts by looking for the leftmost alen and rightmost alien. The reason it finds them, is because it needs to know which alien will trigger the aliens to switch direction and move down a notch. We also need to know the bottommost alien to tell when the aliens land. The **move_aliens** function called inside of **handle_alien_movement** performs the actual alien movement which we will discuss in a bit. The next part of the code loops through all the aliens and performs the animation of them opening and closing their claws. The loop calls **switch_image** on each invader sprite, and passes the total game time so far divided by the blink rate and then modulus 2 which will generate a 1 or 0. The 1 or 0 represents whether the alien opens their claws or close their claws. The higher the blink rate, the slower the alien will open and close their claws. Later we will vary the blink rate as the alien population diminishes to speed up the aliens animation.

We can also use the game time to determine when to play the alien sound as it moves across the screen. We check a `py game_time modulus 400` to play the sound approximately every 1/2 second when the result is zero. The final bit of code sets the position of

all the aliens determine by the flags that were calculated by the **move_aliens** method. Whatever, the flags are set to, all aliens will follow the direction of those flags, since all the aliens move in tandem across the screen.

The last piece of code `py move_aliens_down = False` sets the flag that directs the aliens to move down to false. We want to reset this flag after we already moved the aliens down, because we only want to direct the aliens to move down one row, and then continue on either left or right. Otherwise the aliens would move down rather quickly!

```
def handle_alien_movement():
    global game_time, move_aliens_down, alien_groups,
           move_aliens_right
    alien_rightmost = find_rightmost_alien()
    alien_leftmost = find_leftmost_alien()
    alien_bottommost = find_bottommost_alien()
    (move_aliens_right, move_aliens_down) =
        move_aliens(
          alien_leftmost,
          alien_rightmost,
          alien_bottommost,
          move_aliens_right,
          move_aliens_down)

    # do animation
    for alien_group in alien_groups:
        for next_alien in alien_group:
            next_alien.switch_image(
                 int(game_time/blink_speed) % 2 )
            next_alien.update()

    # play alien sound every half second
    if game_time % 400 == 0 and aliens_exist():
        alien_movement.play()
```

```
25
26      for alien_group in alien_groups:
27          for alien in alien_group:
28              (x,y) = alien.position
29              if move_aliens_right:
30                  alien.move_right()
31              else:
32                  alien.move_left()
33              if move_aliens_down:
34                  alien.move_down()
35              alien.update()
36
37      # reset the move alien down, we only want them to
38      # move down one row.
39      move_aliens_down = False
```

move_aliens , shown below, shows how we calculate the movement determination of the set of all aliens. First we get the position of the farthest most left alien called **first_alien** and the farthest most right alien called **last_alien**. Then if we are currently moving the aliens right, we check to see if the aliens hit the right side of the screen. If they hit the boundary while moving right, then its time to switch direction and move down. As a result, if the alien's hit the right wall, we'll set the **move_right** flag to false to indicate we are now moving left. We will also set the **move_down** flag to true, unless of course, we already hit the bottom of the screen.

We also do a similar check on the **first_alien** coordinates if we are currently moving left (or not moving right). if the first alien position falls beyond the left side of the screen, its time to switch the aliens' direction and move them right as well as down.

```
def move_aliens(leftmost, rightmost, bottommost, move_rig\
ht, move_down):
    global game_time

    last_alien = rightmost
    first_alien = leftmost

    # don't do anything if the first and
    # last alien are empty
    if (last_alien is None) or (first_alien is None):
        return (move_right, move_down)

    # get the position coordinates for the first
    # and last alien
    (last_alien_x, last_alien_y) = last_alien.position
    (first_alien_x, first_alien_y) = first_alien.position

    # if we are already moving right, determine if
    # we should continue,
    # or move down and reverse direction
    if move_right:
        if last_alien_x + last_alien.speed >=
            window_width - (last_alien.rect.width + 5):
            move_right = False
            if last_alien_y + last_alien.speed <         \

              window_height - last_alien.rect.height:    \
                        if (bottommost.position[1] <
                  window_height - 50):
                    move_down = True

    return move_right, move_down

    # if we are already moving left, determine if
    # we should continue, or move down and reverse direct\
```

```
ion
    if not move_right:
        if first_alien_x - first_alien.speed <= 0:
            move_right = True
            if first_alien_y + first_alien.speed <
             window_height - first_alien.rect.height:
              if (bottommost.position[1] <
                window_height - 50):
                    move_down = True



    return move_right, move_down
```

Bombing the player

The game wouldn't be very challenging if the aliens just moved across the screen like sitting ducks, so let's add some alien bombs to the picture. The bombing from aliens above is performed with two functions in our main loop: **check_for_bomb_activation()** and **handle_active_bombs**. The first method checks whether or not an alien is going to release a bomb. In order to determine that if an alien is bombing us, we loop through all the aliens, and generate a random number for that alien. If the random number falls below the bombing frequency, then the alien is bombing and we mark that alien as having an active bomb. We also mark the position of the alien at the time of determination.

```
def check_for_bomb_activation():
     global game_time, active_bombs,
       alien_groups, bomb_frequency
     if (game_time/1000 % 2 == 0):
        if len(alien_groups) <= 0: return
        # find all aliens that currently have
        # ability to bomb below them
        bombing_aliens = [alien
         for alien_group in alien_groups
           for alien in alien_group.sprites()]
        for alien in bombing_aliens:
            # don't drop a bomb if the alien
            # is already dropping one
            # gotta give our player a fighting chance!
            if (alien.bomb_active == False):
                # if the random generated number
                # is below the bomb frequency
                # the alien is dropping
                activate_bomb = random.randint(0, 100)
                   < bomb_frequency
                if activate_bomb and
                   alien.bomb_active == False:
                    alien.bomb_active = True
```

```
                    newestBomb =
                      BombSprite( 0, 0, 5, 15, .03, alien)
                    newestBomb.position
                      = (alien.position[0] +
                         alien.get_width()/2,
                      alien.position[1]
                         + alien.get_height())
                    active_bombs.append(newestBomb)
```

The next function, **handle_active_bombs** will continue to move each active bomb downward from the horizontal position it started from. if the bomb position exceeds the window height we remove the bomb, and mark the bomb as inactive. We also need to check if the player is hit by the bomb and handle the player's death. If the player is killed, we essentially do the same thing we did when the bomb hit the bottom of the screen. We mark the bomb as inactive, reset its position, and remove the bomb from the collection of active bombs. Note the way we remove bombs is to add the bombs to a separate removal list and then remove them after. The reason we do this is because we don't want to alter the active_bomb list while we are looping through it. Removing items from a list while you loop through those items can cause strange behavior.

```
def handle_active_bombs(active_bombs):
    bombs_to_remove = []
    for bomb in active_bombs:
        (bomb_x, bomb_y) = bomb.position
        bomb_y = bomb_y + bomb.speed
        bomb.position = (bomb_x, bomb_y)
        bomb.update()
        bomb.draw(window)
        if (bomb_y > window_height):
            bomb.parent.bomb_active = False
            bomb.position = (0, 0)
            bombs_to_remove.append(bomb)
```

```
        if (handle_player_hit(bomb_x, bomb_y)):
            bomb.parent.bomb_active = False
            bomb.position = (0, 0)
            bombs_to_remove.append(bomb)

    # remove bombs marked for removal
    # from the active bombs list
    if (len(bombs_to_remove) > 0):
        if (len(active_bombs) > 0):
            for bomb in bombs_to_remove:
                active_bombs.remove(bomb)
```

Speeding up the aliens

As we are progressing through the game, we want to challenge the player by threatening to land the aliens faster as they traverse across the screen. We created a method called **handle_alien_speedup** and call it in the main loop to increase the speed of the aliens based on how many aliens are left. The code below will retrieve the total number of aliens by calling **total_aliens**, and based on this total, we'll change the speed of the aliens. If the aliens have reached a total of 20, 5, or 1, the speed of the aliens will be increased as well as the animation speed of the alien opening and closing their claws(this makes the aliens seem more menacing!)

```
def handle_alien_speedup(total_aliens):
    global blink_speed, bomb_frequency,
      first_speed_up, second_speed_up, third_speed_up

    if (total_aliens() == 20):
      if first_speed_up == False:
        blink_speed = 200
        bomb_frequency = 10
        speed_up_aliens()
        first_speed_up = True
```

```
11
12      if (total_aliens() == 5):
13        if second_speed_up == False:
14          blink_speed = 100
15          bomb_frequency = 20
16          speed_up_aliens()
17          second_speed_up = True
18
19      if (total_aliens() == 1):
20        if third_speed_up == False:
21          bomb_frequency = 40
22          blink_speed = 50
23          speed_up_aliens(2.0)
24          third_speed_up = True
```

The code provided defines a function called **speed_up_aliens**, which takes an optional argument factor with a default value of 1.0. This function loops through all the aliens that exist in different alien groups, and increases their speed by a factor that is calculated using the given factor parameter. If no factor parameter is passed, the speed of all aliens in all groups will be increased by a fixed factor of 0.01. Since the initial speed of all aliens is .01, the first *speed up* doubles the speed of the aliens.

The function **speed_up_aliens** first loops through each alien_-group in alien_groups list which represents each row of aliens. Within each alien_group, the function loops through each alien in that group, and then updates the speed attribute of the alien by adding the result of 0.01 multiplied by the factor value to the current speed. The result of this calculation will be added to the current value of the alien.speed attribute, effectively increasing the speed of the alien by the calculated factor.

Note that the factor parameter is optional, and the function will still work even if no argument is passed. In this case, the default factor value of 1.0 will be used, and all aliens will have their speed

increased by .01 * 1.0 or .01. If a factor value is passed to the function, the speed of all aliens will be increased by that factor instead. The last alien is sped up 3 times its original speed (.01 + .01 * 2.0)

```
def speed_up_aliens(factor = 1.0):
    for alien_group in alien_groups:
        for alien in alien_group:
            alien.speed = alien.speed + .01 * factor
```

Adding in Scoring

As we play the game, we'd like to track certain statistics for the user and display them. In the space invasion game, we'll track Score, High Score, and Lives. We already created Score and HiScore Sprites in our stone eater game and we can use the same classes in our space invasion game. Lives is the only new class and is shown below. This class will show how many lives are remaining in the form of 3 ship badges displayed in the upper right hand corner. As a ship is destroyed, one ship badge is removed until there are none left, at which point the game is over.

```
import pygame

class LivesSprite(pygame.sprite.Sprite):
    def __init__(self, window_width):
        super().__init__()
        WHITE = (255, 255, 255)
        self.window_width = window_width
        self.lives = 3
        self.livesImage = pygame.image.load(
          'images/man.gif')
        self.livesImage =
         pygame.transform.scale(self.livesImage,
           (40, 32))
```

```
14          self.rect = pygame.Rect(0, 0, 0, 0)
15          self.small_font = pygame.font.Font(None, 32)
16          self.image = self.small_font.render(
17              f'Lives: {self.lives}', True, WHITE)
18      # Draw the sprite on the screen
19
20      def update(self):
21          WHITE = (255, 255, 255)
22          self.image = self.small_font.render(
23              f'Lives:', True, WHITE)
24          self.rect = self.image.get_rect()
25              .move(self.window_width - 250, 0)
26
27      def draw(self, surface):
28          surface.blit(self.image, self.rect)
29          for i in range(self.lives):
30              surface.blit(self.livesImage,
31               (self.window_width - 180 + i * 50, 0))
32
33      def update_lives(self, lives):
34          self.lives = lives
```

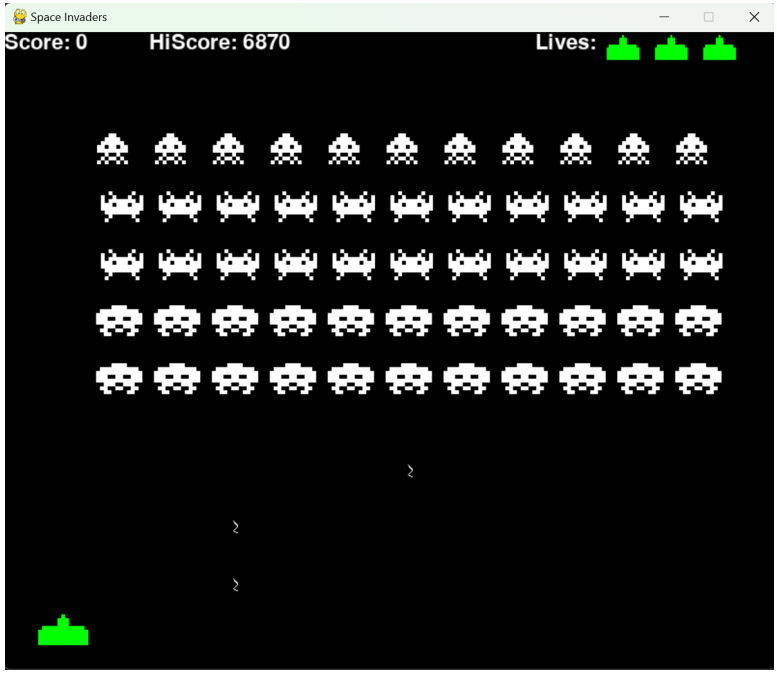
We can use the same image that we use for our player, as we do for our lives indicator, we just need to shrink it down using the pygame.transform.scale method.

Here is a detailed explanation of the code:

Here's a detailed explanation of the code:

1. **class LivesSprite(pygame.sprite.Sprite)**: defines a new class called LivesSprite that inherits from pygame.sprite.Sprite.
2. **def init(self, window_width)**: defines the constructor for the LivesSprite class. It takes the window width as a parameter to help determine where to place the indicator at the top of the screen.

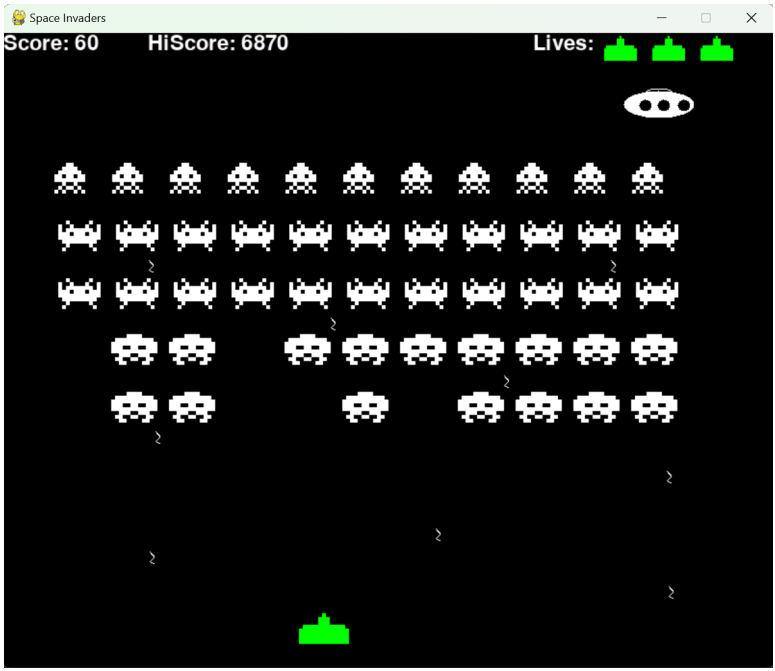
3. **super().init()** initializes the superclass (pygame.sprite.Sprite) to ensure that the LivesSprite class inherits all necessary attributes and methods.
4. **WHITE = (255, 255, 255)** defines a white color tuple used for text rendering.
5. **self.window_width = window_width** stores the window width in an instance variable.
6. **self.lives = 3** initializes the number of lives to 3.
7. **self.livesImage = pygame.image.load('images/man.gif')** loads the image for the player's life representation from the 'images/man.gif' file.
8. **self.livesImage = pygame.transform.scale(self.livesImage, (40, 32))** shrinks the life image to the size (40, 32).
9. **self.rect = pygame.Rect(0, 0, 0, 0)** initializes a rectangular area for the lives display text.
10. **self.small_font = pygame.font.Font(None, 32)** creates a font object of size 32 for rendering the text *Lives*.
11. **self.image = self.small_font.render(f'Lives: {self.lives}'**, True, WHITE) renders the initial text for the lives display.
12. **def update(self)**: defines the update method, which updates the *Lives* display text.
13. **def draw(self, surface)**: defines the draw method, which draws the lives display text and the life images on the screen. The surface parameter is the surface on which to draw the lives display.
14. **def update_lives(self, lives)**: defines the update_lives method, which updates the number of lives which is updated from the main loop when a player dies. The lives parameter is the new number of lives.
15. In summary, the LivesSprite class is responsible for displaying the number of lives a player has in a Space Invaders game. It handles the rendering of both the text and the life images on the game screen.



In the handle_player_hit method inside our main program, if the player has been hit by an alien bomb, we decrement the number of lives and update the LivesSprite to reflect the lost life:

```
handle_player_hit(bomb_x, bomb_y):
...
player_lives = player_lives - 1
lives_indicator.update_lives(player_lives)
```

Launching the UFO



In the classic arcade game Space Invaders, the saucer, also known as the UFO or mystery ship, appears at the top of the screen and moves horizontally across the screen at regular intervals. The saucer typically appears every 25 to 30 seconds, but this can vary depending on the specific version of the game or the stage the player is in. The purpose of the saucer is to provide an opportunity for the player to earn bonus points by shooting it down as it moves from one side of the screen to the other. In our version, we'll launch the saucer every 20 seconds.

To code this up, we'll start by creating a SaucerSprite. The Sprite will be animated using 3 saucer images which will give the illusion of the saucer spinning. Here is the breakdown of the SaucerSprite

code:

Here's a breakdown of the class and its methods:

- **init**: The constructor method initializes the saucer sprite with three different images (name1, name2, and name3) for the saucer, an initial position (x, y), and an optional level parameter. It sets the initial image sprite, generates a random score for the saucer, calculates the speed based on the level, and initializes other relevant attributes.
- **reset**: This method resets the saucer to a given position and level, updating the speed, points, and other attributes.
- **update**: This method updates the position of the saucer sprite and its three image sprites, as long as the saucer is not dead.
- **draw**: This method draws the saucer sprite on the given surface. If the saucer is dead, it draws the saucer's score instead of the sprite.
- **move_left**: This method moves the saucer sprite to the left by its speed value, provided the saucer is not dead.
- **switch_image**: This method switches the saucer's image sprite between the three provided images based on the given image number.
- **get_width** and **get_height**: These methods return the width and height of the saucer sprite, respectively.
- **kill**: This method sets the saucer's dead attribute to True and records the time when the saucer was killed using Pygame's get_ticks() function. The reason it records the time of death, is to allow the sprite time to show the number of points scored on the screen after the saucer dies.

Here is the full code for the saucer sprite:

```
import random
import pygame
from ImageSprite import ImageSprite


class SaucerSprite(pygame.sprite.Sprite):
    def __init__(self, name1, name2, name3, x, y, level =\
 1):
        super().__init__()
        self.active = False
        self.imageSprite1 = ImageSprite(name1, x, y)
        self.imageSprite2 = ImageSprite(name2, x, y)
        self.imageSprite3 = ImageSprite(name3, x, y)
        self.imageSprite  = self.imageSprite1
        self.explosion = pygame.font.Font(None, 32)
        self.imageSprite = self.imageSprite1
        self.points =  random.randint(1, 6) * 50
        self.saucerScore = self.explosion.render(
            str(self.points), True, (255, 255, 255))
        self.speed = .05 * (.9 + level/10.0)
        self.position = (x, y)
        self.rect = self.imageSprite.image
            .get_rect().move(self.position)
        self.dead = False
        self.death_time = 0

    def reset(self, x, y, level = 1):
        self.imageSprite = self.imageSprite1
        self.points =  random.randint(1, 6) * 50
        self.saucerScore = self.explosion
            .render(str(self.points), True,
              (255, 255, 255))
        self.speed = .05 * (.9 + level/10.0)
        self.currentDirection = 'left'
        self.position = (x, y)
```

```
36          self.rect = self.imageSprite.image
37              .get_rect().move(self.position)
38          self.dead = False
39          self.death_time = 0
40
41      # update the position of the 3 sprites
42      # representing the ufo
43      def update(self):
44          self.rect = self.imageSprite.rect
45          if self.dead == True: return
46          self.imageSprite.rect = self.imageSprite.image
47              .get_rect().move(self.position)
48          self.imageSprite1.rect = self.imageSprite.rect
49          self.imageSprite2.rect = self.imageSprite.rect
50          self.imageSprite3.rect = self.imageSprite.rect
51          self.rect = self.imageSprite.rect
52
53      # Draw the sprite on the screen
54
55      def draw(self, surface):
56          if self.dead == True:
57              surface.blit(self.saucerScore, self.rect)
58          else:
59              self.imageSprite.draw(surface)
60
61      def move_left(self):
62          if self.dead == True: return
63          (x, y) = self.position
64          self.position = (x - self.speed, y)
65
66      # switch between the 3 images representing the saucer
67      def switch_image(self, imageNumber):
68          if self.dead == True: return
69          if (imageNumber == 1):
70              self.imageSprite = self.imageSprite1
```

```
71          elif (imageNumber == 2):
72              self.imageSprite = self.imageSprite2
73          else:
74              self.imageSprite = self.imageSprite3
75
76
77      def get_width(self):
78          return self.imageSprite.get_width()
79
80      def get_height(self):
81          return self.imageSprite.get_height()
82
83      def kill(self):
84          self.dead = True
85          self.death_time = pygame.time.get_ticks()
```

How will we decide when to launch the UFO? In the original game, the saucer would move across the screen approximately every 25 seconds. We'll do the same in our game inside the game loop:

```
1       #launch the saucer every 20 seconds if the game is no\
2   t over.
3       if game_over == False and game_time % 20000 == 0 and \
4   game_time > 0:
5           start_saucer()
```

The `start_saucer` function initializes our saucer and prepares it to move across the screen. It also plays a special saucer sound:

```
def start_saucer():
    if saucer.active == False:
        saucer.reset(window_width - 50, 50, level)
        saucer.active = True
        saucer_sound.play()
        saucer.position = (window_width - 50, 50)
```

Once the saucer is initialized, we need to keep it moving across the screen. We do that through the `handle_saucer_movement` function in the main loop.

```
def handle_saucer_movement():
    global game_time
    saucer_show_score_time = 1000

    if saucer.active:
        saucer.move_left()
        saucer.update()
        saucer.draw(window)
        saucer.switch_image(
         int(game_time/saucer_blink_speed) % 3)
        (saucer_x, saucer_y) = saucer.position
        if (saucer_x <= -100):
            saucer.dead = True
            saucer.position = (0, 0)

    if saucer.dead:
        current_saucer_death_time
         = pygame.time.get_ticks()
             - saucer.death_time
        if current_saucer_death_time >
           saucer_show_score_time:
            saucer.active = False
```

This function is responsible for moving, updating, and drawing a

saucer (or UFO) from a Space Invaders-like game. It also handles the case when the saucer is "dead" (hit by a projectile, for instance).

Here's a step-by-step explanation of the code:

1. **saucer_show_score_time = 1000**: This line sets a variable that represents the time (in milliseconds) for which the saucer's score will be shown on the screen after it has been killed.
2. **if saucer.active**: This condition checks if the saucer is active (visible and moving on the screen).
3. **saucer.move_left()**: If the saucer is active, this method moves the saucer to the left.
4. **saucer.update()**: This method updates the saucer's position and its image sprites.
5. **saucer.draw(window)**: The saucer is drawn on the given surface (the window).
6. **saucer.switch_image(int(game_time/saucer_blink_speed) % 3)**: This line switches the saucer's image sprite based on the current game time and a variable called saucer_blink_speed, creating an animation effect.
7. The next lines check if the saucer has moved off the screen (to the left). If so, the saucer is considered dead, and its position is reset to (0, 0).
8. **if saucer.dead**: This condition checks if the saucer is dead.
9. **current_saucer_death_time = pygame.time.get_ticks() - saucer**.death_time: This line calculates the time elapsed since the saucer was killed.

if current_saucer_death_time > saucer_show_score_time: This condition checks if the time elapsed since the saucer's death is greater than the time for which the saucer's score should be shown. If true, the saucer is set to inactive (saucer.active = False), meaning it will not be displayed or updated until it's reset and activated again.

In summary, the **handle_saucer_movement** function ensures that the saucer moves across the screen, updates its position, and switches its images while active. When the saucer is dead, it shows the score for a set duration before setting it to inactive.

Checking if we hit the saucer

When we shoot a bullet from our player, the bullet can either hit an alien, hit the saucer, or go all the way to the top of the screen. We need to handle the case where the bullet hits the saucer moving across the screen:

Inside of the handle_bullet function, we need to check if the bullet collided with the saucer:

```
def handle_bullet(bullet, bullet_active):
    ...

    if (handle_saucer_hit(bullet_x, bullet_y)):
        bullet_active = False
        bullet.position = (0, 0)
```

The `handle_saucer_hit` function checks for saucer collision with the bullet. If it collides, the function calls the kill function on the SaucerSprite. The kill function, puts the saucer in a dead state which allows the saucer to show the score for a few seconds after it was shot. This function also plays a sound to indicate the saucer was hit and it adds the score from the saucer bonus points.

```
def handle_saucer_hit(bullet_x, bullet_y):
    global player_score, bullet, saucer
    (x, y) = saucer.position
    # check if bullet collides with saucer
    if bullet_x > x
        and bullet_x < x + saucer.get_width()
        and bullet_y > y
        and bullet_y < y + saucer.get_height():
            saucer.kill()
            saucer_dying.play()
            player_score += saucer.points
        return True
    return False
```

Conclusion

In this chapter, we delved into the development of a dynamic first-person 2D shooter game, Space Invasion. Our journey encompassed essential game components such as user input, sound design, enemy movement, collision detection, and captivating animation.

Though Space Invasion is slightly more intricate compared to our previous projects, it serves as an excellent blueprint for crafting your own real-time gaming experiences. As you design unique sprites, you can unleash your creativity by combining and adapting them across a multitude of game genres and concepts. This endeavor not only expands your skillset but also opens up a world of possibilities for innovation in the gaming landscape.

Appendix

Source Code

In order to access the source code for my book, please visit the GitHub repository at https://github.com/microgold/pygames. This repository contains all the relevant code examples, organized in a clear and concise folder structure to make it easy for you to navigate through the content. To get started, simply clone or download the repository to your local machine and follow the instructions provided in the README.md[1] file. Should you have any questions or encounter any issues, feel free to submit an issue on the repository, and I will be more than happy to help you out. Happy coding!

Where to Find Images

Open Game Art[2] has free images to pick from

Itch.io[3] has a mix of free and paid game assets

Where to Find Sounds

FreeSound.org[4] has a host of free sounds you can download to use in your game.

[1] https://github.com/microgold/pygames/blob/master/ReadMe.md
[2] https://opengameart.org/
[3] https://itch.io/
[4] https://freesound.org/

Also SoundBible.com[5] has royalty-free free sounds you can look through.

Other Resources

Python and Pygame can be challenging to learn, but there are many excellent resources to help you get started. We recommend:

For Python

Python Tutorial - Learn Python Programming (Step by Step)[6]

Python Tutorial - Full Course for Beginners[7]

Python.org[8]: The official website of the Python programming language provides a wealth of information for beginners, including tutorials, documentation, and community resources.

Codecademy[9]: Codecademy offers a free Python course for beginners.

Coursera[10]: Coursera offers free Python courses from top universities and institutions.

Learn Python the Hard Way[11]: A free online book that provides a hands-on, exercise-driven approach to learning Python. []

Python for Everybody[12]: A series of free online courses offered by the University of Michigan that provide an introduction to Python programming.

[5] https://soundbible.com/
[6] https://www.youtube.com/watch?v=XGf2GcyHPhc
[7] https://www.youtube.com/watch?v=rfscVS0vtbw
[8] https://www.python.org/
[9] https://www.codecademy.com/learn/learn-python
[10] https://www.coursera.org/courses?query=python
[11] https://learnpythonthehardway.org/python3/
[12] https://www.py4e.com/lessons

W3Schools[13]: A website that offers a variety of tutorials and resources for learning Python, as well as other programming languages and web development technologies.

More PyGames

Brick Breaker[14]

Sudoku[15]

Snake Game[16]

[13]https://www.w3schools.com/python/
[14]https://www.geeksforgeeks.org/brick-breaker-game-in-python-using-pygame/
[15]https://www.geeksforgeeks.org/building-and-visualizing-sudoku-game-using-pygame/
[16]https://www.edureka.co/blog/snake-game-with-pygame/