Pygame Car Crash Tutorial - Part 2

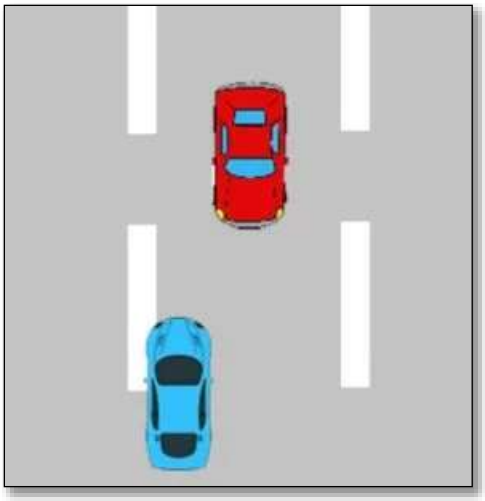
Contents

Time required: 30 minutes

Preview of the Game

Here's a sneak peak of the game that we are going to work on.

Car Crash Demo Video



Car Crash is simple arcade type game. The object is to move your blue car back and forth to avoid the oncoming red cars.

Assets

There is a **CarCrashAssets.zip** file attached to the assignment. This has all the images and sounds we will use in the game.

1. Create a folder named **assets** in your game directory.
2. Extract the files and copy them to the **assets** folder.

Config Module

1. Create a Python file called **config.py**
2. Insert the following code.

```
"""
    Filename: config.py
    Author:
    Date:
    Purpose: Global variables and constants for the entire program
"""

# Initialize constants for screen size
WIDTH = 400
HEIGHT = 600
```

Import Config Module

1. Save **car_crash_1.py** as **car_crash_2.py**
2. Add the following code.

```
"""
    Name: car_crash_2.py
    Author:
    Date:
    Purpose: Draw the playing field
"""
# pip install pygame-ce
# Import pygame library
import pygame
# Import exit for a clean program shutdown
from sys import exit
import config
```

3. There is a new import statement: **import config**
4. Remove **HEIGHT** and **WIDTH** from the **car_crash_2.py** program.

Classes and Objects

We'll be using Classes and Objects for our program. Whether it's GUI, PyGame or any other large application the Object-Oriented Programming approach is almost always the best idea. Using Classes, we'll be using methods to store blocks of code that are to be repeated several times throughout the game.

Let's start with displaying the background image.

```
15  class CarCrash:
16      def __init__(self):
17          # Initialize the pygame library
18          pygame.init()
19
20          # Create the game surface (window)
21          self.surface = pygame.display.set_mode(
22              (config.WIDTH, config.HEIGHT)
23          )
24
25          # Set window caption
26          pygame.display.set_caption("Car Crash")
27
28          # Setup computer clock object to control the speed of the game
29          self.clock = pygame.time.Clock()
30
31          # Only allow these events to be captured
32          # This helps optimize the game for slower computers
33          pygame.event.set_allowed(
34              [
35                  pygame.QUIT,
36                  pygame.KEYDOWN
37              ]
38          )
39
40          # Load background image from file into an image variable
41          self.background = pygame.image.load(
42              "./assets/street.png").convert_alpha()
```

1. Create the CarCrash class.
2. The **__init__** method creates the CarCrash class.
3. **pygame.init()** initializes the PyGame library.
4. Set the display mode using **pygame.display.set_mode** to create a window (drawing surface). The window dimensions are defined by **WIDTH** and **HEIGHT** from the config module**.** This display surface is stored in the **self.surface** variable.
5. Set the window caption.
6. **self.clock** is used to control the speed of the game.
7. **pygame.event.set_allowed:** Only allow these events to be captured, this optimizes the game.

8. Load **street.png** from file into an image variable → **self.background**
    a. This image will be used as the playing field.
    b. **convert_alpha()** is used to optimize the image for faster blitting.

What is a Game Loop?

A game loop is a fundamental structure in game development that continuously repeats, or loops, to handle various tasks and update the game state. It's the heart of a video game, responsible for coordinating and executing the sequence of actions that make up the gameplay. The main components of a typical game loop include:

1. **Input**
    a. Gather input from the player, such as keyboard, mouse, or controller inputs.
2. **Update**
    a. Update the game state and entities based on the input and the passage of time.
    b. This includes handling player movement, updating AI behavior, checking for collisions, and managing other aspects of the game logic.
3. **Render**
    a. Draw or render the current state of the game to the screen.
    b. This involves updating the graphics, animations, and any visual elements that the player interacts with.
4. **Timing**
    a. Manage the timing of the loop to ensure a consistent frame rate.
    b. This helps in creating a smooth and responsive gaming experience.

The loop continues to iterate, repeating these steps, creating the illusion of continuous motion and interaction. The game loop runs as long as the game is active, providing real-time updates and responses to player input.

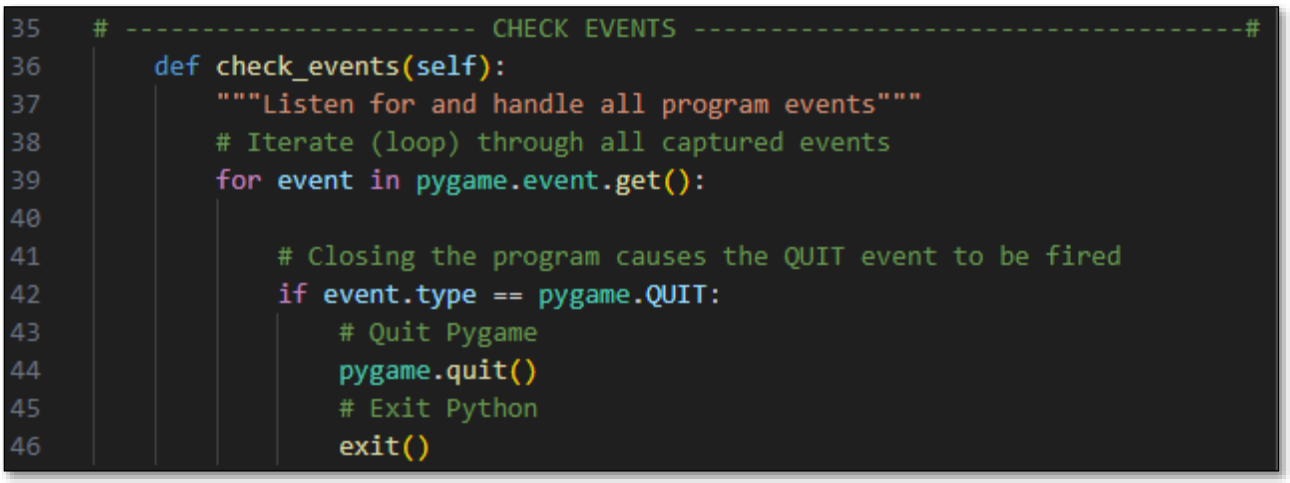
Here's a simplified example in pseudo-code:

```
while game_is_active:
    # Process player input
    handle_input()

    # Update game state
    update_game_state()

    # Render the current state
    render()

    # Control the frame rate and wait if needed
    control_frame_rate()
```

The exact implementation of a game loop may vary based on the game engine or framework being used, but the fundamental structure remains similar across most games.

Listen for Events

Modify the following code in your **car_crash_2.py** file.



This method listens for and handles all program events, such as keyboard presses, mouse clicks, or program closing.

Game Loops and Game States

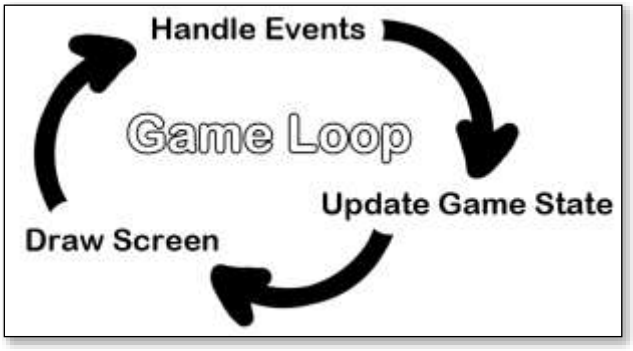
Every game that has what is called a "Game Loop". It's a standard game development concept.

The Game Loop is where all the game events occur, update, and get drawn to the screen. Once the initial setup and initialization of variables is out of the way, the Game Loop begins where the program keeps looping over and over until an event of type **QUIT** occurs.

A game loop (also called a main loop) is a loop where the code does three things:

1. Handles events
2. Updates the game state
3. Draws the game state to the screen

Since the game state is usually updated in response to events (such as mouse clicks or keyboard presses) or the passage of time, the game loop is constantly checking and re-checking many times a second for any new events that have happened. Inside the main loop is code that looks at which events have been created (with PyGame, this is done by calling the **pygame.event.get()** function). The main loop also has code that updates the game state based on which events have been created. This is usually called event handling.



Car Crash Game Loop

Add this code below the **check_events** method.

```
58      # ------------------------ GAME LOOP --------------------------------- #
59      def game_loop(self):
60          """Infinite Game Loop"""
61          while True:
62              self.check_events()
63
64              # ---------------- DRAW ON SURFACE --------------------------- #
65              # Draw everything on the surface first
66              # Fill the surface with the background image loaded earlier
67              self.surface.blit(self.bg, (0, 0))
68
69              # ------------------ UPDATE DISPLAY ---------------------------- #
70              # From the surface, update Pygame display to reflect any changes
71              pygame.display.update()
72
73              # Cap game speed at 60 frames per second
74              self.clock.tick(60)
75
76
77  # Create game instance/object
78  car_crash = CarCrash()
79  # Start the game
80  car_crash.game_loop()
```

self.check_events() called the check_events method each time through the game loop.

blit

In PyGame, the term "blit" stands for Block Transfer. It refers to the process of copying the pixel values from one surface onto another. In the context of game development or graphical applications using PyGame, the **blit** method is commonly used to draw images onto a screen or another surface.

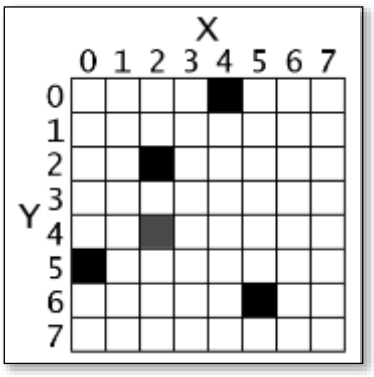
```
# ---------------- DRAW ON SURFACE --------------------------- #
# Draw everything on the surface first
# Fill the surface with the background image loaded earlier
self.surface.blit(self.bg, (0, 0))
```

- **self.surface** - The destination surface onto which you want to draw the image.
- **self.background** - The surface containing the image you want to draw.

- **(x, y)** – The coordinates specifying where on the destination surface the top-left corner of the source image should be placed. (0, 0) is in the upper left corner of the screen.

Pixel Coordinates

The window that the program creates is composed of little square dots on the screen called pixels. Each pixel starts off as black but can be set to a different color. Imagine that instead of a Surface object that is 300 pixels wide and 300 pixels tall, we just had a Surface object that was 8 pixels by 8 pixels. If that tiny 8x8 Surface was enlarged so that each pixel looks like a square in a grid, and we added numbers for the X and Y axis, then a good representation of it could look something like this:



We can refer to a specific pixel by using a Cartesian Coordinate system. Each column of the X-axis and each row of the Y-axis will have an "address" that is an integer from 0 to 7 so that we can locate any pixel by specifying the X and Y axis integers.

For example, in the above 8x8 image, we can see that the pixels at the XY coordinates (4, 0), (2, 2), (0, 5), and (5, 6) have been painted black, the pixel at (2, 4) has been painted gray, while all the other pixels are painted white. XY coordinates are also called points. If you've taken a math class and learned about Cartesian Coordinates, you might notice that the Y-axis starts at 0 at the top and then increases going down, rather than increasing as it goes up. This is just how Cartesian Coordinates work in PyGame (and almost every programming language).

The PyGame framework often represents Cartesian Coordinates as a tuple of two integers, such as (4, 0) or (2, 2). The first integer is the X coordinate and the second is the Y coordinate.

Colors

Colors are a big part of any game development framework or engine.

PyGame uses the RGB system of colors. This stand for Red, Green and Blue respectively. These three colors combined (in varying ratios) are used to create all the colors you see on computers, or any device that has a screen.

Some of these colors are predefined in PyGame. The **pygame_color_picker.py** program is in the assets zip file.

There are three primary colors of light: red, green and blue. (Red, blue, and yellow are the primary colors for paints and pigments, but the computer monitor uses light, not paint.) By combining different amounts of these three colors you can form any other color.

In PyGame, we represent colors with tuples of three integers. The first value in the tuple is how much red is in the color. An integer value of 0 means there is no red in this color, and a value of 255 means there is the maximum amount of red in the color. The second value is

for green and the third value is for blue. These tuples of three integers used to represent a color are often called RGB values.

Because you can use any combination of 0 to 255 for each of the three primary colors, this means PyGame can draw 16,777,216 different colors (that is, 256 x 256 x 256 colors).

For example, we will create the tuple (0, 0, 0) and store it in a variable named BLACK. With no amount of red, green, or blue, the resulting color is completely black. The color black is the absence of any color. The tuple (255, 255, 255) for a maximum amount of red, green, and blue to result in white. The color white is the full combination of red, green, and blue. The tuple (255, 0, 0) represents the maximum amount of red but no amount of green and blue, so the resulting color is red. Similarly, (0, 255, 0) is green and (0, 0, 255) is blue.

The values for each color range from 0 – 255, a total of 256 values. You can find the total number of possible color combinations by evaluating 256 x 256 x 256, which results in a value well over 16 million.

You can mix the amount of red, green, and blue to form other colors. Here are the RGB values for a few common colors:

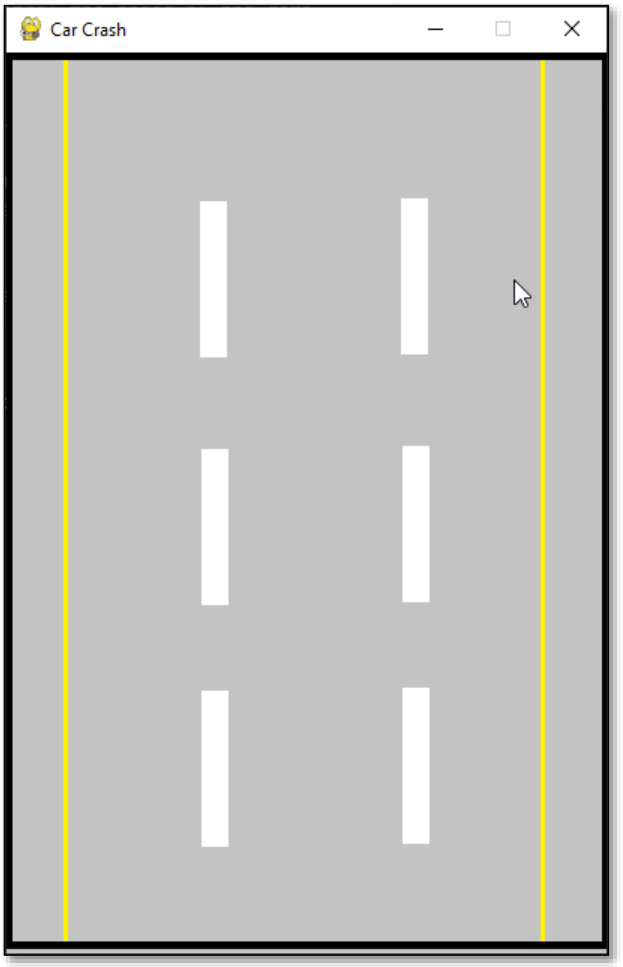
Color	RGB Values
Aqua	(0, 255, 255)
Black	(0, 0, 0)
Blue	(0, 0, 255)
Fuchsia	(255, 0, 255)
Gray	(128, 128, 128)
Green	(0, 128, 0)
Lime	(0, 255, 0)
Maroon	(128, 0, 0)
Navy Blue	(0, 0, 128)
Olive	(128, 128, 0)
Purple	(128, 0, 128)
Red	(255, 0, 0)
Silver	(192, 192, 192)
Teal	(0, 128, 128)
White	(255, 255, 255)
Yellow	(255, 255, 0)

To use colors in PyGame, we first create Color objects using RGB values. RGB values must be in a tuple format, with three values, each corresponding to a respective color. Here are some examples below showing some color objects in PyGame.

```
BLACK = pygame.Color(0, 0, 0)          # Black
WHITE = pygame.Color(255, 255, 255)    # White
GREY = pygame.Color(128, 128, 128)     # Grey
RED = pygame.Color(255, 0, 0)          # Red
```

The **fill(color)** method is used to fill in objects. For instance, assigning a rectangle the color green will only turn the borders green. If you use the **fill()** method and pass a green color object, the rectangle will become completely green.

Example run:



Assignment Submission

1. Attach a screenshot showing the operation of the program.
2. Zip up the program files folder and submit in Blackboard.