Universidad Carlos III de Madrid

Programming
16480

Final Project Report

Authors:
Salma Alejandra Caisaguano Barzallo
Richard Lim

December 16th, 2022

Universidad
Carlos III de Madrid

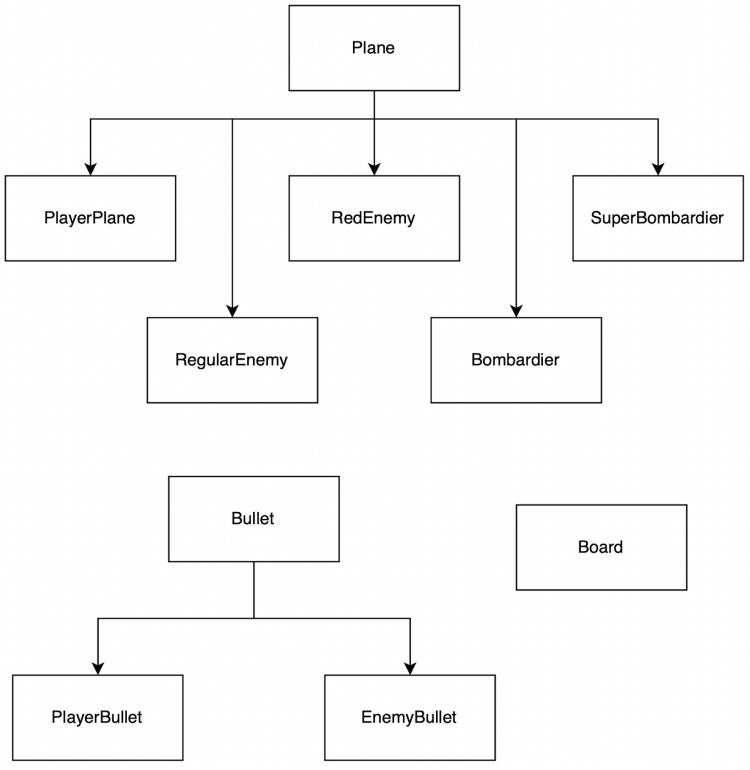
Table of Contents:

1. Summary

This document aims to outline the work completed on the final project for the Programming class. In this document, we will explain each different class and the components that go into each. We will go over a brief description of the most relevant algorithms and the work performed. In the end, we will give out final thoughts on the project and the work performed.

2. Design of Classes

In total, there are 10 different classes we used to re-create the 1942 game. These classes were all created to organize the design of the game. Rather than having all of the code in one file or one class, it was easier to keep track of all of the different elements using classes. We used various techniques we learned throughout the completion of the Programming course to aid us in creating this project.



a. Board

The most important class is the ***Board*** class. This is because it contains all of the necessary code to create the game while calling all of the other classes. Within the **__init__** method, the most important elements are the lists of each enemy and the **for** loops that append enemies into each respective list. For instance, the self.regular_enemies element is created as an empty list, and the **for** loop adds 20 regular enemies to that list at different locations. This process is repeated for each of the different types of enemies.

The **update** method contains a large part of the code of the game. To begin with, it is updated and executed every frame. This is important because this allows the game to be updated immediately. The first important components in this method are the keys pressed on the keyboard. For each arrow key pressed, an update is sent to the **PlayerPlane** class and updates the location of the plane depending on which arrow key was pressed. In addition, if the space key is pressed, it updates the **PlayerBullet** class and generates 2 bullets in different locations every time the key is pressed.

The next important component in the **update** method is the creation of bullets for the enemies. To do this, there are different **for** loops that check the elements in each list of enemies, and if the enemy's **y** value is at a certain location, it will create an enemy bullet. For each element in the lists **pbullet_list1, pbullet_list2, and enemiesbullet_list**, it will either move the bullet up the board or down the board, depending on if it is an enemy bullet or player bullet, and update the board.

The next important component in the **update** method is the scoring system. This is done with **for** and **if** loops. If the location of the bullet is the same location as an enemy, the game will update by not drawing the enemy that was shot, deleting the

bullet that hit the enemy, and updating the score. This is done for each bullet in the list of enemy bullets.

The game is organized into 4 stages, each one with two different groups of enemies:

1. Regular and red enemies
2. Regular and red enemies
3. Regular enemies and bombardiers
4. Regular enemies and superbombardier

For the **regular_enemy** planes, a new regular enemy will spawn only when the previous regular enemy has reached a certain y-position or when the last enemy of a different class was no longer on the board. The same goes for **red_enemies,** which, in addition, followed a V-formation. The **bombardier** and **super_bombardier** planes will only generate and appear on the board when the previous one had left the board. The **draw** method is responsible for creating all of the visual components on the board. This includes, the color of the background of the board, the player plane, all the bullets, and all of the different enemies.

b. Plane

The ***Plane*** class is a mother class and only contains an **__init__** method. The method requires 2 elements, the **x** position of a plane on the board, and the **y** position of a plane on the board.

i. PlayerPlane

The ***PlayerPlane*** class calls upon the ***Plane*** class to generate a plane at a designated position. The **move** method takes in 2 components: **direction** and **size,** both of which are received from the **board** class. The **direction** is updated whenever an arrow key is pressed and the **size** is the **player plane's**

width of the plane if the direction is left or right and its height if the direction is up or down.

ii. RegularEnemy

The ***RegularEnemy*** class also calls upon the mother ***Plane*** class. The **__init__** method calls upon the **REGULAR_ENEMY_SPRITE** from the **Constants** file. It also creates a boolean value called **self.inboard** and sets the value to true.

The **direction** method moves the plane down 3 pixels each frame. The **update** method updates the **self.inboard** boolean value and changes it to false if the plane leaves the board. The **draw** method continues to draw the sprite if the plane is on the board.

iii. RedEnemy

The ***RedEnemy*** class contains many similar components to the ***RegularEnemy*** class, but there are a few new components. In the **__init__** method, a boolean value called **self.nextplane** is created and the value is set to false. In the **update** method, if the **y** value of the **red enemy** is greater than 40, the **self.nextplane** is set to true and generates another **red enemy**.

iv. Bombardier

The ***Bombardier*** class also contains many similar components to the ***RegularEnemy*** class, but there are a few new components. In the **__init__** method, a boolean value called **self.lives** is created and the value is set to a random number between 6 and 10 (as the player plane shoots twice at a time). In the **move** method, the plane will only move if the value for **self.lives** is greater than 0. The rest of the components are the same as the others.

v. SuperBombardier

The ***SuperBombardier*** contains the same components as the ***Bombardier*** class. There are no new components, the only thing that changes are its lives(10-14).

c. Bullet

The ***Bullet*** class is a mother class. The **__init__** method requires 2 elements, the **x** position of a bullet on the board, and the **y** position of a bullet on the board.

i. PlayerBullet

The most important component of the ***PlayerBullet*** class is the creation of the 2 lists of bullets. Similar to the classes of the **enemy planes**, the ***PlayerBullet*** class contains 3 methods: **__init__, move, update, and draw**. The **__init__ method** class gets the sprite of the bullets from the boolean values and sets both to true. Then it appends or adds the **self** value to each list. The **move** method takes in a direction in the form of a string, and if the direction is "up", the bullet will move 2 pixels up every frame. The update method updates the value of **self.inboard** and changes it to false if the bullets leave the board, and removes the bullets from the lists if they leave the board. And the **draw** method draws the bullet on the board.

ii. EnemyBullet

The ***EnemyBullet*** class is the same as the ***PlayerBullet*** class except for its movement, which is "down".

3. Relevant Algorithms

The most relevant algorithm is the ***Board*** class. As mentioned before, this class contains all of the necessary code to run the game.

Another relevant algorithm is the 3 different lists of bullets. These are important because they are what allow the player to interact with the enemies. Without these lists, the player will be unable to kill the enemy and the enemy will be unable to kill the player. The **for** and **if** loops that are used to check if a bullet hits an enemy or if an enemy bullet hits the player are important as well.

The last relevant algorithm is the list of enemy planes. The list of enemy planes allows the **RegularEnemy, RedEnemy, Bombardier, and SuperBombardier** classes to create and draw the enemies on the board. Without these lists, none of the enemies would ever appear on the board.

4. Work Performed

Unfortunately, we were not able to accomplish some enemies' characteristics. For instance, we could not figure out, how red enemies could perform circular formations.
In addition, we could not animate the player's helix or manage to implement its loops.

Nevertheless, we were mostly successful in creating a completely functioning game. Once the program is run, the **player plane** appears in the middle of the board and the enemies begin to appear from the top of the board and move downward. There are 4 different waves of enemies. Only when all of the enemies are killed or have left the board will the next wave begin. If the player is hit by one of the enemy bullets, the player will lose 1 life. The player begins the game with 3 lives and ends it when either the player has lost all his lives or has won (reached the top of the board). The player can move using the arrow keys and shoot with the space key.

As for the score, there are two values in the ***Board*** class that are called **self.score** and **self.high_score** which are initially set to 0. As the player begins to kill enemies, points are added to the **self.score.** By the end, the total sum of the score is the final score. If **self.score** is

higher than **self.high_score,** then its value would be modified to the new score. Both results will be displayed on the screen at the end of the game, along with “WINNER”,(if the player was able to win), or with “GAME OVER” if not.

5. Conclusion

As written in the previous section, we were successful in creating a mostly functioning game with a moving and shooting player and different kinds of enemies that appear in waves. However, there were some elements we were unable to complete.

While implementing the game, there were a few problems we ran into but we were able to fix most of them. For example, we had a problem with the creation and drawing of the bullets. We were unable to show the bullets on the screen and were unable to detect if the bullets hit an enemy plane. Through problems like this, we were able to use the knowledge we gained throughout this course, the help from our professors, and the GitHub for Pyxel to solve these problems.