PROGRAMMING

FINAL PROJECT

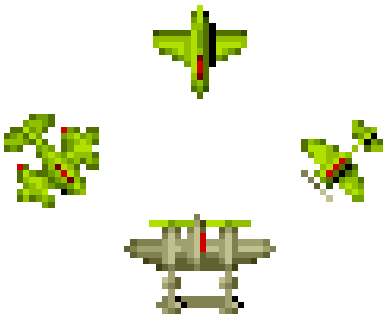
Enica King & Estefany González

Data Science and Engineering

Group 96

ABSTRACT

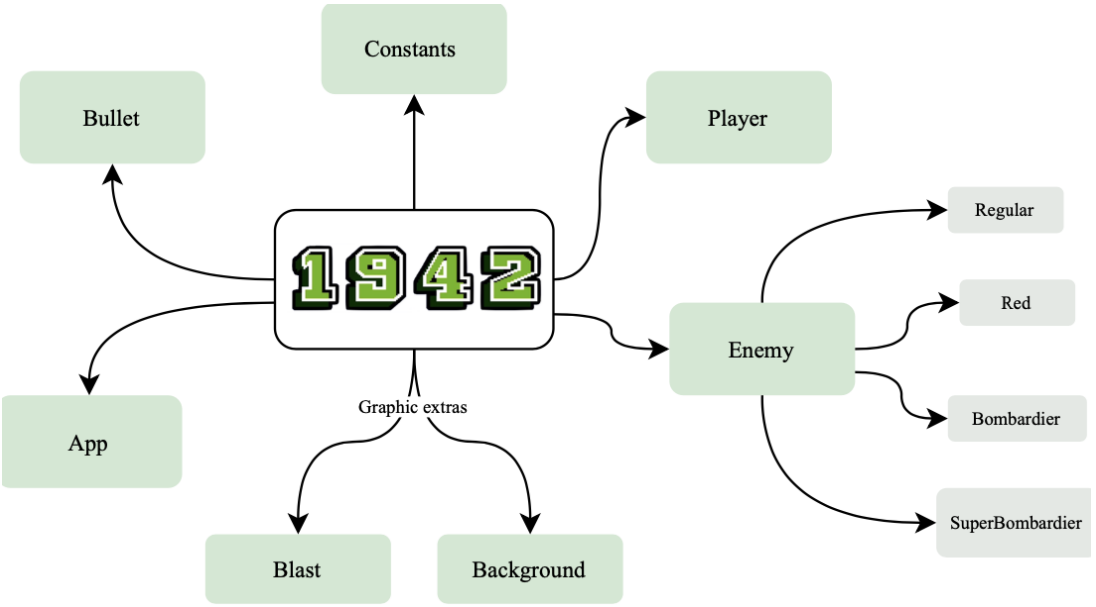
This project aims to mimic the classic retro game 1942, using the open-source Python package *'pyxel'*. Guided by the rubric and inspired by documented gameplay, the project has resulted in a fully functional 'level' designed within the game itself. The main coding approach has been one of succinctness, code reusability, and faithfulness to the original game's design.



✧ TABLE OF CONTENTS ✧

Class Design

The project code implements a hierarchical organization according to object-oriented principles such as inheritance (4 subclasses), encapsulation (conversion of functions into methods by storing within classes), etc.



Player

Contains all information pertaining to the player sprite controlled by the player. The player moves according to arrow keys, shoots when the user presses the key space or performs a loop (maximum 3 times) when presses the Z key. Inside the draw function of this class, we can find the animation of the loop and the movement of the helix.

Enemy

The mother class enemy contains all the variables which are common among all the enemy sprites the player faces, such as its position and if it is alive.

Regular/Red/Bombardier/SuperBombardier

Subclasses of Enemy that contain their own attributes pertinent to their update and draw functions.

Bullet

In this class we have differentiated the player bullets, which go upwards and the enemy bullets which go downwards. Their respective sprites can be pellets or color-coded bullets according to the sprite type.

Constants

The only file that is not a class, storing all the constants such as height and width of sprites, colors used, score system and bonuses, etc.

Blast

This additional class describes the motion of an explosion when a bullet comes into contact with the player plane or when the player meets an enemy.

Background

This class generates an infinite loop of downward moving stars. The color of the stars is related to their speed, so the user gets the illusion of depth and feel that they are flying forward through space.

App

The most important class, that calls upon all previous classes and loads the images and sounds. When the game is activated, it is updated and drawn frame by frame, the background and player sprite called separately as unique entities, and the others (bullets, blasts and enemies) grouped into nested lists within the master_list variable.

It is responsible for keeping track of the sprites on screen, registering keyboard presses to control the player or set off game modes, scores, high scores, enemy generation, and level length. It is also divided into three game modes called scenes: title, play, and gameover.

Relevant Methods/Attributes

The most relevant methods derived over the course of this project were related with the movement and animation of the sprites: the **draw** and **update** functions for each class.

offset attribute in several enemy subclasses to randomize the side-to-side and up-to-down movement of the regular and bombardier enemy planes, respectively.

halfway attribute, a boolean for the regular enemy plane that flips the height parameter on pyxel.blt to animate flip without need of manual animation

checkcollision method defined in the main class App to analyze interactions between 1) player bullets and enemies, 2) enemy bullets and the player, and 3) regular/red enemies and the player to analyze sprite interactions. If either entity was inside the other, done by checking the overlap of the coordinates, it would disappear.

frame_counter attribute of App to monitor regularity of enemy sprite generation and act as a timer for the game level duration.

The most important methods are six methods in the App class that make up the update and draw functions for each scene that make up the backbone of the game. The background draws and updates regardless of the scene.

update_title_scene: checks for the button 'P' in order to start a new game.

update_play_scene: the longest method in the project, that monitors level length, enemy generation, sprite interactions and keeps track of point systems and player lives.

update_gameover_scene: removes all entities previously generated and resets important game stats in preparation for the next game: player lives and starting position, score, lives, and game timer.

draw_title_scene: displays highscore, game name, and lets player know to press 'P' to play.

draw_play_scene: draws all entities, player turns left, score and high score, and the player lives as hearts that disappear as they are lost.

draw_gameover_scene: adds score bonus for lives left, updates high score and displays congratulatory message if the player has lives left (game over otherwise). Gives the option to quit the game or start a new one.

Relevant Algorithms

The most relevant algorithms are those that were considered breakthroughs to solving problems of code functionality.

To organize the sprites three lists were created within a nested masterlist: **blast_list**, **bullet_list** and **enemy_list**. Then in the main code, in each frame the masterlist was iterated through using tailored functions to update, draw, remove dead entities, and clear the lists in preparation for a new game.

The point system is worth noting, as constants that are invoked simultaneously with the **checkcollisions** method and add different amounts to the score based on the enemy type. This also adds a bonus at the end of the game for the player lives left.

A neat detail is for the red enemies, the swarm is stored within a list, and each time a red enemy is defeated, the list checks for the elimination of the entire swarm, and if so, awards the player an extra turn.

The most notable algorithm within the code is the usage of **scenes** TITLE, PLAY, GAMEOVER to cycle through game 'modes', in this order. Buttons and conditions monitored the transition from one scene to the next, and the three different setups of the game (waiting for the user to start the game, the game itself and the game over scene) allow the code to be compartmentalized.

Lastly, the game was simplified by deleting unnecessary sprites that have gone outside of the frame, thus simplifying the game.

Work Performed

We managed to comply with all the project prerequisites and even implemented some additional features. The project is designed as the eighth level of the original game.

The program starts on the loading screen, displaying the logo of 1942 and the highscore, and when P is pressed it transitions into gameplay by generating the player sprite, turns, lives, the score and highscore. The player can control the plane by moving and shooting.

The duration of the game is set to 2000 frames, and during this regular and red enemy swarms are randomly generated. Bombardiers and Super Bombardiers have a minimum frame count required to generate, given how for shorter, easier levels they do not appear.

When players touch regular or red enemies, or any enemy bullets, they lose lives. They can avoid this entirely by looping (3 times maximum) for temporary invincibility while the animation is in effect. Furthermore, by destroying an entire Red enemy swarm, the player is granted an extra turn.

The player can shoot down the incoming enemy planes to prevent being shot at, which increases the score depending on the type of enemy plane shot down.

The game level ends either when the attribute frame_counter reaches the predetermined limit, or the player runs out of lives. At this point, the game transitions into the final scene, and updates the high score if necessary. It gives the player the option to play the level again or quit.

Some additional features that we implemented were sound effects for the player shooting, and the collision blast, a blast Class that animates sprite interactions, an infinite starry sky background, and displaying the player lives as hearts in the top-left corner of the screen.

Nonetheless, there were some features of the original game that became too complicated to implement. For instance, the Red and Bombardier bullets are supposed to honing, and the SuperBombardier bullets a spray that fans out. However, this would require the creation of separate classes of the bullets, reception of player position as parameters, etc. We also slightly modified the flight paths of the enemies; the Bombardier no longer turns, the regulars fly more irregularly, and the red enemies do not do complete loops. We think though, that the point of this was to create differently moving enemy sprites, rather than copying the original game.

Conclusion

Summary

Through much hard work and dedication, we have been able to create a game level with both scalability and modularity.

Scalability: The inclusion of the constants GAME_FRAME_LENGTH, GAME_WIDTH, and GAME_HEIGHT allow for the modification of the game level duration and dimensions of the screen. Furthermore, the background is an infinite loop that can adapt to these constants.

Modularity: The compartmentalization of the code into classes, and file constants.py that can be modified greatly contributes to the modularity of the project. Furthermore, the use of the random package to randomize enemy generation, flight paths, and background star positions avoids the monotony of the game.

Obstacles

Many problems arose throughout the project.

In the beginning, we had a very hard time setting up the game. To this end, the video tutorials and demo code uploaded by pyxel's creator, *kitao*, were of great help. From this springboard, we were able to build the rest of the game very efficiently. However, all his code was in one file, which made the required division into files per class and subsequent necessary import statements difficult.

The implementation of graphics was another of the main issues; understanding how to load the images and use the image bank, the use of colkey, and how to animate sprite movements. For instance, we could not get the red enemy planes to loop until we had the idea of using the mathematical sine function to create the flight path.

Lastly, frustrating random bugs that we could not understand because they were not necessarily errors, which required us to comb through the code. For instance, on the initial game over screen the generated enemies would persist until their flight paths had run out instead of being cleared, and the player game over blast would expand infinitely. This turned out to be a misplaced hashtag that didn't allow the entities to update.

Takeaways

This project helped us to mature in our understanding of object-oriented programming. We struggled to transform classes into methods, invoke methods within the same class, ensure the declaration of attributes, and create subclasses.

However, all the obstacles were overcome (some with a Herculean effort) and the end result is a functional version of the 1942 shooter game, and important lessons for us as we were able to put many theoretical concepts into practice. It was also quite enjoyable and rewarding to see the results of our hard work.