Elijah Gaytan

Eg28423

Simulation and Game Programming:

Creating a Top-Down Space Shooter in Python

I: Introduction

In this lab, the primary motivation and purpose is to design a game or simulation utilizing the popular SDL library, pygame, within the Python scripting language. For 2d games, Pygame makes the game-making process substantially easier by simplifying the process by which images or sprites can be implemented into the game, providing modules to play sounds and music, manage surfaces and backgrounds within the environment, and run and queue events that are integral to running the game. Designing a game in Python utilizing the pygame library makes the lab a suitable introduction to the development of video games as a whole.

Primarily, the main goal of this lab was to create a Space Shoot 'em up style game, in the style of games such as the popular Space Invaders and Project Starfighter. The game's main idea is to survive and get the highest score possible by destroying enemy ships, which spawn at random intervals and locations towards the top of the screen. There are four different types of ships,ranging from frigates to Capital ships, each with varying levels of health, speed, and, to an extent, behavior. Upon being destroyed, each ship earns the player a different amount of points. The basic mechanics of the game

follow a relatively common schema. The player utilizes the arrow keys to shoot and the space bar to shoot a bullet. There are no limit to the amount of bullets that can be on screen; players can shoot as many bullets as they possibly can, and some enemies will likewise do the same. The player has health bar, which completely depletes after 10 hits, but can be restored by randomly spawning heart pieces, and, in addition, also possesses an energy bar, which regenerates over time. Energy, at the moment, is currently only used in one maneuver, and that is to ignite afterburners, or, in other words, Dash, by pressing the Z key.

II: Results and Discussion

Implementing some of the basic ideas of the game definitely proved to be a challenge, especially given a lack of programming experience in the Python language.

Implementation of all game mechanics was done through several classes; these classes include entities such as the player, player and enemy bullets, items, and enemies. In particular, everything related to the game's assets and visuals were constantly being updated every frame (the game runs by default at 60fps) in the game-state loop. This includes important information such as the player's health and energy bars, the transition of bullets across the screen, and the movement of the player himself.

Visuals in the game were implemented using the pygame.sprites module. Every sprite in the screen is added to an all\_sprites\_list, or a list that contains the location of every sprite currently in the game. Every sprite that is added to the is an object that has an update() method, and the update method is called for every sprite in the game on each frame. This is necessary to redraw sprites on the screen, given that each update() method of the object each in the list specifies movement of a different type (i.e movement down the y-axis). Individuals objects the game are also added to a list that corresponds to their class, as well. This is necessary for detecting collision with bullets that may hit enemies, or bullets that go off-screen. Since every sprite is surrounded by a rectangle, the distance formula is used to measure the range of impact between the centerX of the rectangles for each bullet in the list of all bullets for each enemy, and if the distance is less than a specified value, than the bullet is removed from the all\_sprites\_list and bullets\_list, and the enemy, if its HO is 0, is likewise removed as well.

Enemy spawning in the game works based on the python RandInt method. Each class of enemy has a different chance of spawning. Each enemy is assigned (in the main method) a particular static range of number between the randInt minrange and maxrange. The game state then generates random numbers between a specified range each frame, and if the random number generated is equal to the range of the number of the enemy as defined in the class, then the enemy is randomly spawned with an x-value between (0, Width).

Each type of enemy is also initialized with a different amount of HP, NUMOFHITS, given its class. Generally, the harder the enemy is, the wider the range of integers that can be generated. Easier ships are more common enemies and thus correspondingly have a smaller range of integers in which they can be generated.

Certain enemies can also fire as well. These bullets employ the same hit-collision techniques that the player’s bullets do, and are added to a separate list.

Finally, the final images that are updated are the background, hp bar, and energy bar. These images are blit to the screen every frame, with changes being made to the health and energy within the loop depending how much health the player has. Because the player can restore health through randomly spawning pickups, the health meter must be updated at the end of the loop.

Conclusions:

This lab has taught me much about game programming in general. There are a lot of variables that go into making even a simple shooter game, and it is important to update these variables every frame if the game is to be accurate. In addition, managing hit detection and collision can be challenging, particularly when you use a variety of sprites at a myriad of sizes. This can complicate matters, even when using the distance formula. Generally speaking, a lot of work goes into updating the game state, especially when you must account for so many variables, such as health, ammo, mana, or whatever resource you may want to use. Creating more in-depth games such as Rpgs I assume can also be particularly difficult, especially when you must consider the large amount of events such a game would have.