Mini Project 4

Mary Keenan and Lauren Pudvan

Project Overview

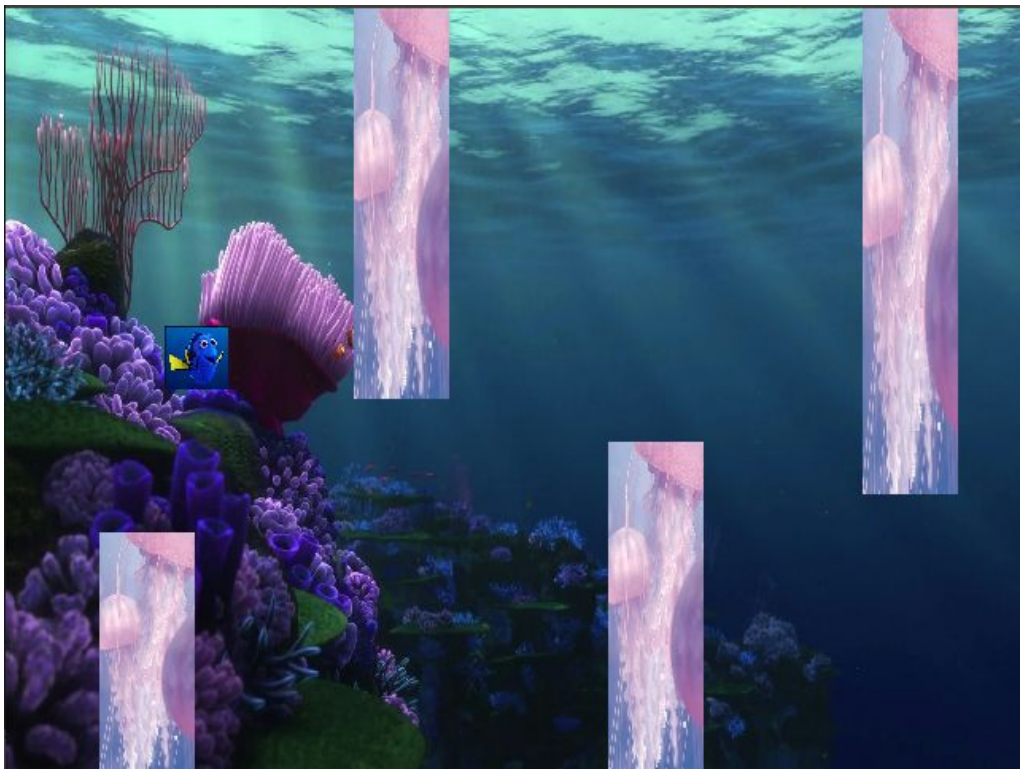
We created a spin-off of Flappy Bird called Swimmy Fish. It stars a fish named Dory who is trying to pass through a field of jellyfish without getting stung. The player can use the up and down keys to move Dory to evade the incoming jellyfish, which approach her from both the top and the bottom of the screen. When Dory hits a jellyfish, the game ends. We used the pygame package to create the game -- Dory and the jellyfish are sprites, and we use collision detection to sense when Dory hits a jellyfish.

Results

We were able to make a convincing spin-off of Flappy Bird, replete with background music, images, and a goofy storyline. We were able to implement collision detection between Dory and the jellyfish with PyGame's colliderect. If Dory hits any of the jellyfish, the game ends.

We also figured out how to make the jellyfish move similar to how the pipes move in relation to the bird in Flappy Bird. Adjusting the length of time that elapses between each frame of the game changes the difficulty level -- a faster game equals faster jellyfish, which in turn means the user has to move Dory faster.

Swimmy Fish in Action



Implementation

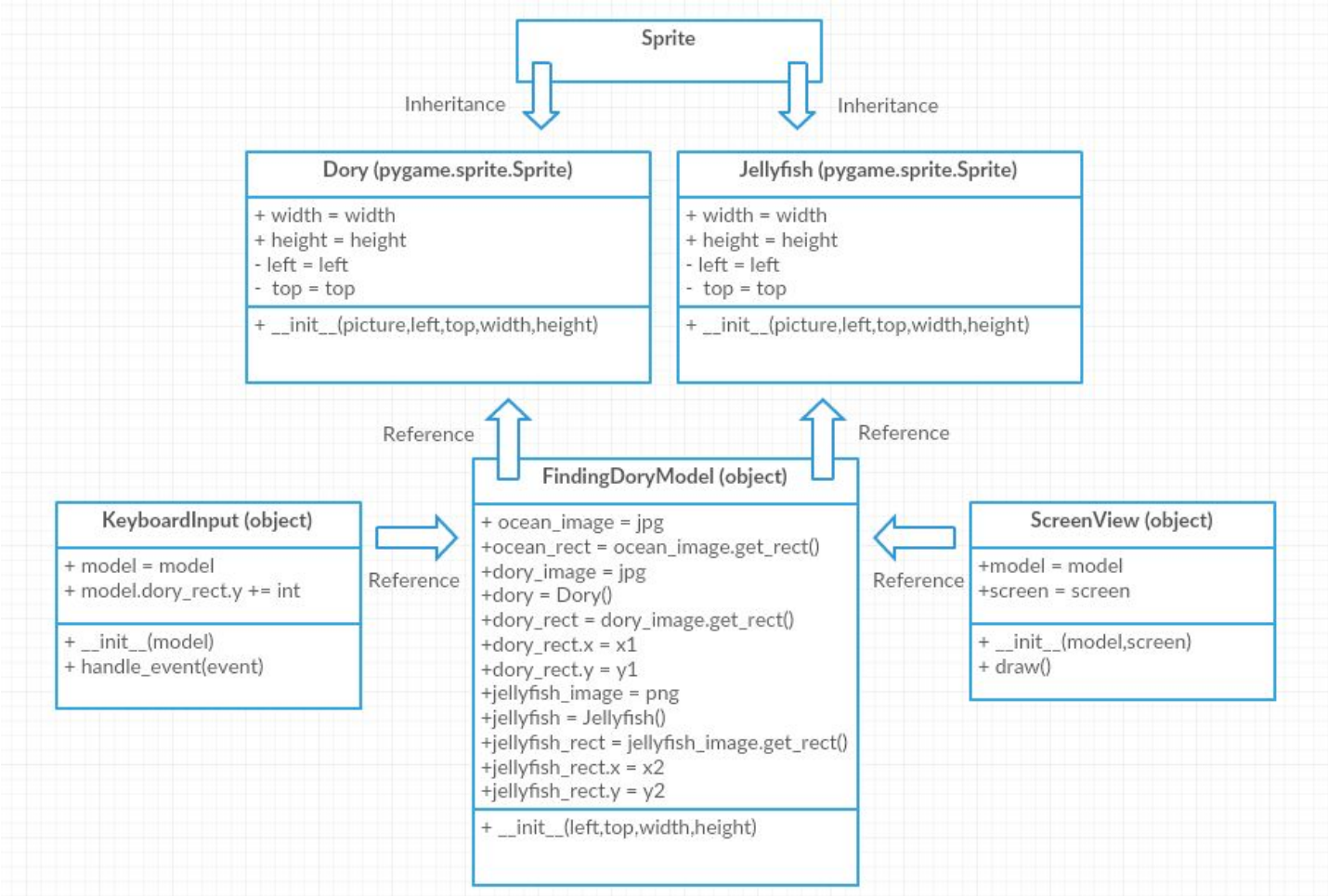
Our code has three main parts to it: the model, view, and controller. The FindingDoryModel class contains the bulk of the code -- it's what creates, sizes, and places the images for one Dory, four jellyfish, and the ocean. This class contains no method but "__init__" and has many attributes, as can be seen in our UML Class Diagram below.The ScreenView class draws Dory, the jellyfish, and the ocean to the screen with the method "draw". It's attributes are also listed below. The controller class, KeyboardInput has a "handle_event" method that takes the user's input and changes Dory's y-position.

We changed Dory and Jellyfish to sprites because we thought that would be the easiest way to detect collisions between Dory and the jellies, but we had trouble implementing that and ended up using colliderect anyway. If we had more jellyfish (we have four), this would have been cumbersome, because each jellyfish needs its own collision test to see if it hit Dory, but four was manageable. This is actually why we wanted to do spritecollide originally -- you can put sprites in a group and test **once** to see if any of the group-sprites collide with another sprite whereas colliderect tests a collision between two rectangles -- not two groups. So you have to run it multiple times, but it works.

We were able to get the jellyfish to move by steadily decreasing their x position as the program runs (in the while loop). When their x-position hits 0, it's reset to 640 and the jellyfish gets a new height. This is how we're able to get a seemingly infinite number of jellyfish to attack Dory when we really only have four -- two on top and two on bottom. Four are on-screen at a time, and they just continually move off the screen to the left before restarting on the right side again.

One decision we had to make was whether or not to have Dory sink. In Flappy Bird, the force of gravity pulls the bird down and it has to push itself up to stay in the air -- in our game, the force of buoyancy pushes Dory up and she has to swim to stay down. We chose this because dead fish float. Ergo, it seems logical to assume Dory would rise to the surface of the ocean rather than sink to the bottom if she wasn't actively swimming.

UML Class Diagram



Reflection

We think our project was successful, because we created our MVP+ and our game is pretty fun to play, if we do say so ourselves. We're happy with how our game looks, and the soundtrack is a fun plus. We could have gone an extra step and added a start and/or end screen, but it didn't seem like that would add much to the game. Our project was scaled well because we were able to achieve our MVP and learning goals and we did a good job evenly distributing the work throughout the project (I think it's good to note that although we set an even pace for working on the project, most of the things we worked on didn't actually start working until the last day or two of the project, which is part of the reason we're so happy with it right now!). Our plan for testing the code was to start off with a basic game and make small

improvements and test along the way (incremental development). By testing between small changes, we could catch errors and locate them easily.

We started the project working next to each other, using Floobits to share code. Once we set up the basic model -- moving blocks to represent the Flappy Bird pipes and a square that could move up and down (the bird), we divvied up the rest of the work. We made a list of improvements we wanted to see in the game and picked which one we wanted to work on. When we made progress, we updated the code on Floobits and picked something new from our list to work on. This worked pretty well, but one problem that arose was trying to integrate our code. Each time the code was updated, one of us had to manually read through the changes the other made and fit it into their code. A small benefit of this was that we got a closer look at the new code and learned how the partner achieved the goal. In the future it would be more efficient to use github because of its version control.