Scott Mackinlay and Keenan Zucker

Software Design, Spring 2015

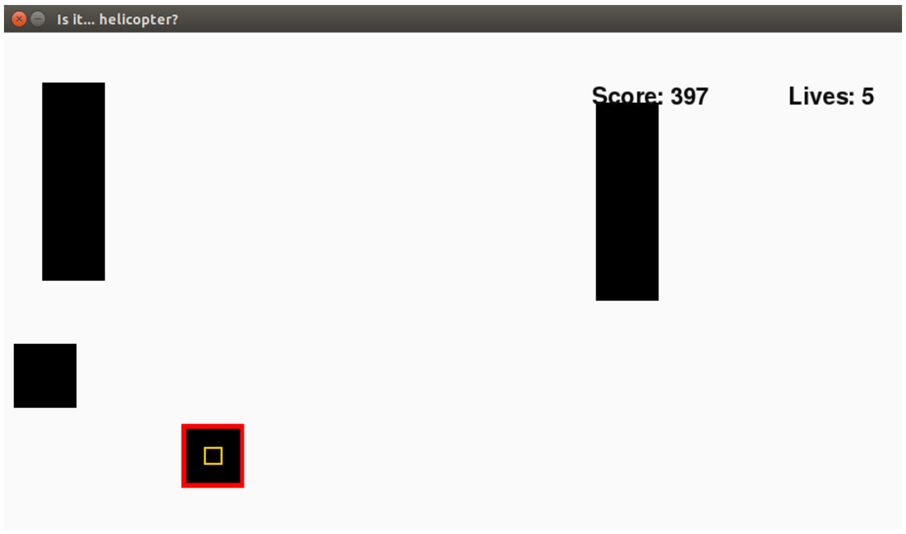
Project Overview

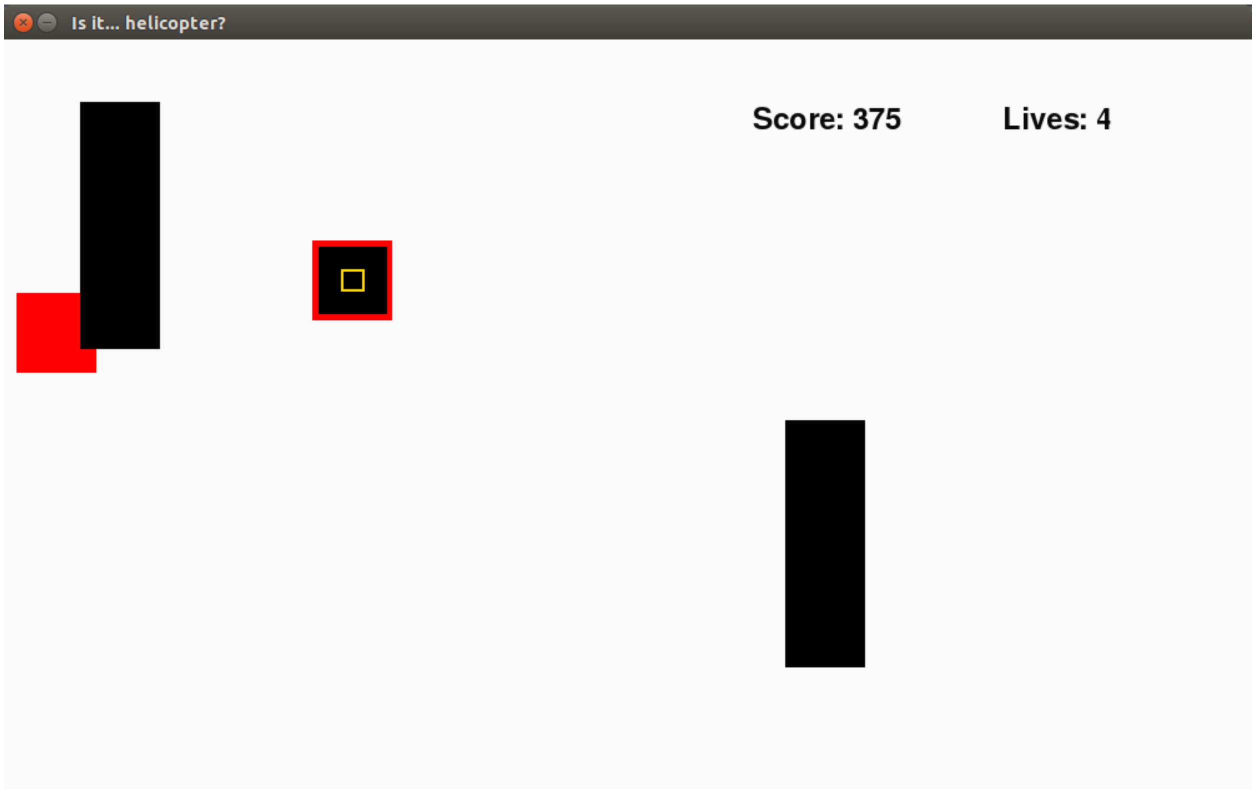
We both enjoyed playing an old classic called 'Helicopter' when we were younger, which is a one button game in which the character has to navigate through randomly generated walls. We wanted to create a similar feeling game, but with a simpler visual design and the option to control the character with only audio.

Results

We successfully created a continuously side scrolling game! We decided on keeping the visual design very simple, with the character simply as a square rather than a picture. Within the window (shown below, the score and number of lives can be located in the upper right hand corner. A randomly generated set of walls approach you from the right with the added bonus of a smart enemy (affectionately known as a "baddie") who spawns at your current location and rushes at you faster than normal walls. When your character collides with either a wall, the baddie, or the upper or lower dimensions of the screen, then you become hit and 'burn' for two seconds, giving you two seconds of invincibility to regain your composure and get back into the flow of gameplay.

The game can be controlled by using the spacebar, causing your character moves up while releasing spacebar allows your character to fall. Primarily, we allowed our code to take audio input control. This means that the computer uses the microphone to sample the audio input, and if the audio exceeds a certain threshold, the character moves upwards. Anything below the threshold lets the character fall. Happy yelling!





This is the helicopter at the moment of impact with a wall. It becomes red to show that you messed up and that you have a moment to position yourself.

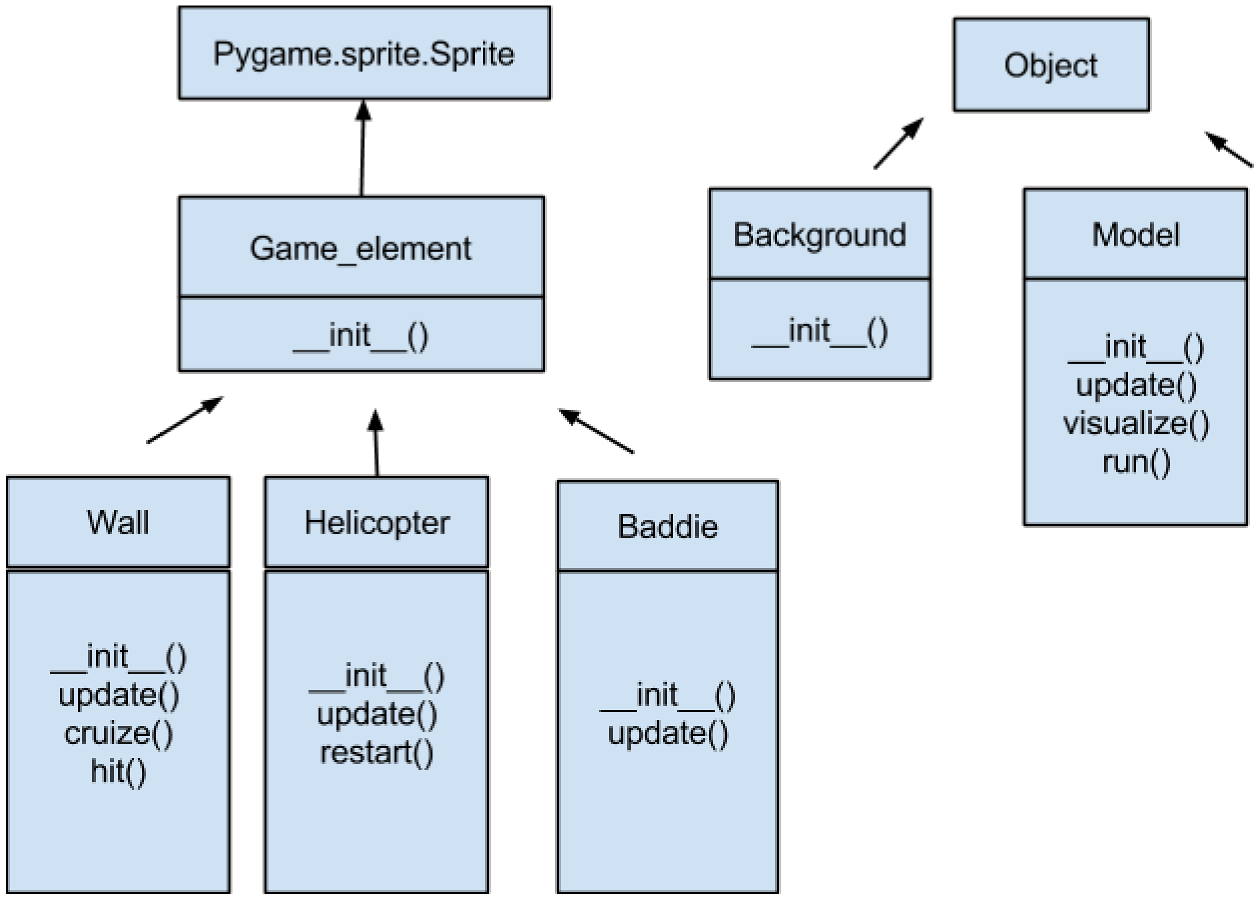
The last feature to be added to the game was the idea that it should get progressively harder as time goes on. Essentially this translated into an increased velocity of the walls as they approach the character. Be warned, it gets real fast.

Implementation.

From an architectural point of view, our code contains a game element class from which three game objects inherit their basic properties. In addition, we have a model and background class that handle the input and visualization for the game. The model is broken into three sections: update, visualize, and run. Run is the indefinite while loop that calls update and visualize which change the game objects positions and render them to the screen respectively.

We decided to have all of our game objects (the helicopter, the wall, and the "baddie") to inherit from a common super. We came to this conclusion after realizing that the code for each of these classes was redundant, especially when it came to the init methods. Alternatively, we could have just repeated the code three times but that would have been unnecessary, lame, and bad coding practice.

In addition, we broke our model up into sections. Initially, the model was a function that was called once and contained a while loop to keep the game rolling. Because this resulted in one massive function controlling our game, we found that debugging and navigating to the correct section of code to change/add was exceedingly difficult. To fix our predicament, we broke the model into three distinct sections: update, visualize, and run. This had the pleasant side effect of allowing us to better control what type of input was required (keyboard vs audio) without having to alter large sections of code.



Reflection

Our organization on this project was interesting. On our first pass, we had a class for the objects, like helicopter, wall, and baddie, but basically wrote everything else in main. This gave us functional code, but it wasn't organized in an intuitive or clean way. Knowing this, we decided to re-organize using a Model class and a Background class. We also decided to create a Game_Element class from which the objects helicopter, wall, and baddie inherited from.

I think our project was appropriately scoped, as we did replicate the game in a pretty successfully way. And not only did we recreate the game, we made it much better:

1. The movement is fluid and smooth
2. Hit detection works well
3. There are smart bad guys to make play interesting
4. The game increases in difficulty with time
5. The audio input control works well too.

We learned that we should probably have written down our UML diagram early on, before we even started writing code. This would have helped us keep a frame of where we would put each section of code, since we ended up doing a fair amount of simply trying to move code around and copy/paste to organize retroactively. Making the UML diagram at the end didn't really help us much as it would have at the beginning to guide our organization from day one.

As far as teamwork goes, we initially did it the hard way! Our first couple iterations were emailed back and forth, which is terrible practice! We eventually got our git repository working, and after that, it worked a lot better. One thing to improve would be making more branches. Instead of this, we would usually just create a new file to test with, and see how it went, then adding the changes into the clean code if it was successful. Our communication was pretty good, and I felt like we both had a common goal of getting a cool game working. We tried pair programming a few times, which worked pretty well, but definitely took slower overall.