Software Design Project Reflection: The Fall

Judy Xu & Becca Getto

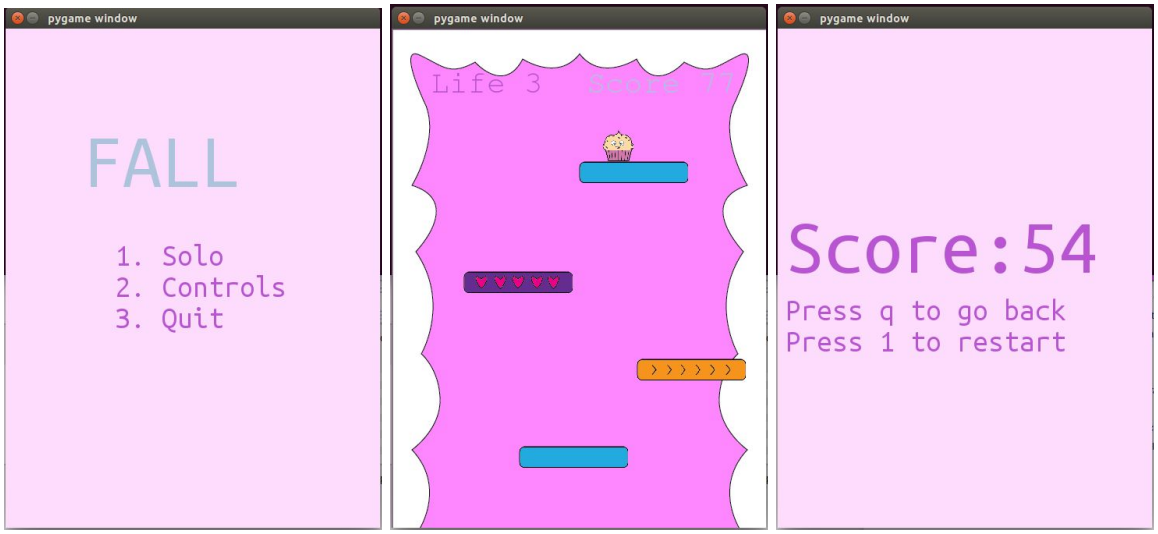
March 9th, 2016

Project Overview

We created our own version of the NS Shaft game that we renamed Fall. The game consists of a player who must continually jump down from plank to plank as they scroll up the screen. The player receives points for the number of planks have been created while they are alive. There are various plank types such as spike planks that reduce life, flip planks that the player falls through after a moment of standing on it, fast and slow planks that change the player's speed, and heart planks that increase life. The player continues to jump from plank to plank gaining points until they die by losing all of their life point, colliding with the top of the screen, or falling off the bottom.

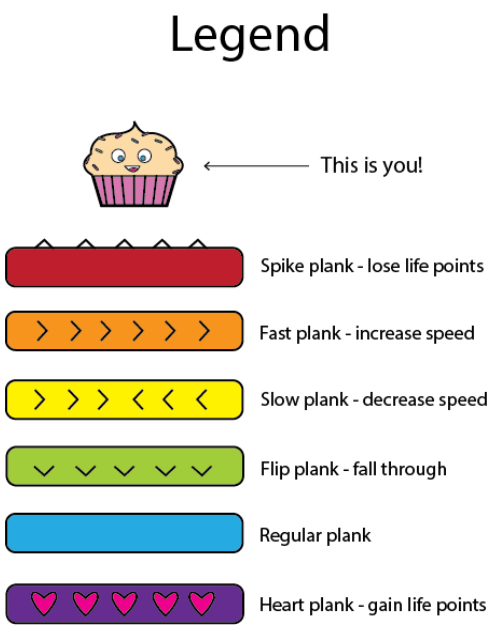
Results

We were able to create our version of the NS Shaft game with our own set of custom graphics and a variety of variations from the original game. Our version of the fall opens with a menu screen giving the options to play solo, learn the control, or quit the game.



In solo play mode, the cupcake starts standing on a plank on the top of the screen. The planks immediately begin to scroll upwards as new planks are created at the bottom of the screen. The user

must move the cupcake player left and right to jump down from plank to plank in order to earn points. The various types of planks exhibit different behaviors as described in the legend below. The player earns points for the amount of time they are able to stay alive in the game. The player loses by either running out of life points due to standing on spike planks, by falling off the bottom of the screen, or by being pushed up to the top of the screen by the plank. When a player loses, a screen displaying the score as well as the options to go back or to restart appear.

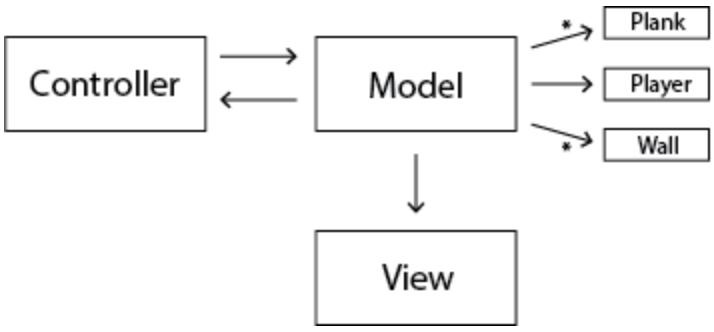


Implementation

We utilized the Model, View, Controller architecture to implement our program. The Model is the most complex aspect of the code and contains the Plank, Player and Wall classes as well as the update method. The Plank class defines the characteristics of the planks and has the attributes: left, top, width, height, and plank type. There are 6 types of planks: regular, spike, flip, heart, fast and slow. This class also contains a method movey for moving the planks up the screen. The Player class defines the characteristics of the player and contains the attributes: left, top, width, and height. It contains two methods , movex and movey, for moving the player around the screen. In the full graphic version of the code, we removed the Wall class and used an image in its place. The Fall model has many methods to make planks, move planks, check the status of planks and to determine if the player has died.

The update method contains much of the functionality of gameplay. It updates the model at each time step with information regarding which planks are on the screen, their position, the player position, player speed, player life and the game score. Throughout gameplay, the player will fall downwards until a collision occurs. The update function checks the player and plank positions to determine if there has been a collision. If a collision has occurred, the type of plank is also noted. A different collision behavior occurs for each type of plank and the behavior continues for the entire time the player is on that plank. The regular plank simply stops the downward motion of the player and the player begins to move upwards with the plank, spike planks cause the player to lose life point, flip planks stop the players downward fall for a moment before the player falls through, fast planks increase the player speed while slow planks decrease it, and heart planks increase the life points of the player. The update function keeps track of the score of the game by keeping track of the gameplay time. The start time if each game trial is recorded and subtracted from the entire time the program has been running to determine the score. Initially we had decided to keep track of the score simply by counting the number of planks that had been created, but we later decided to delete the planks once they left the screen to increase the speed of the program. Once this was implemented, we needed to switch to using the time elapsed as a measure of the score instead.

The Controller has access to the model and is used to control the position of the player. The Controller checks for an input either via mouse or keyboard and uses this information to change the x position of the player in the Model. The View also has access to the Model and is used to draw all of the planks as well as the player, background and walls on the screen. The View class also displays the game score and the player's life on the screen. Prior to the start of game play, the View displays the start menu and will display the game, the instructions, or quit the game depending on what the user inputs.



Reflection

From the start of our project, we mapped out in detail everything we wanted to accomplish, what our minimum viable deliverable was, and all of our potential extensions. This gave us a great framework to begin working from and helped us to decide what aspect of the code to implement in what order. We had a clear sense of all of the necessary classes and decided to use the Model, View, Controller architecture to scaffold our program. All of this gave us a relatively clear direction in our implementation.

Initially, we began writing the code together using the pair coding format. This was a great system in that we could communicate freely while we were creating code and we were both able to contribute to and understand the program. While coding together would have been ideal, our schedules did not allow for us to be in the same room at all times when we needed to be working. We broke up various tasks and implemented sections of code on our own then uploaded them to Cloud9 so we could both have access to what the other wrote. Judy was very excited about the project and was able to implement large sections of the program on her own. Becca also implemented a few features on her, but as a much weaker programmer, she was not able to follow the changing code very well. As the project got further along, it become harder for Becca to contribute as she didn't understand every detail of some of the methods Judy implemented. Unfortunately, Judy ended up contributing a larger section of the code to the project and Becca was less able to. Becca began to focus on the graphical aspects of the game as she felt she needed to pull her weight on the team. While we are both relatively new coders, Judy has become much more comfortable programming than Becca which lead to an uneven distribution of work. Hopefully in future projects work could be distributed more evenly despite differences in knowledge and comfort with programming.