CSCI 205 - Software Engineering and Design
Final Project - Super Omar.io
Omar El-Etr, Morgan Muller, Mateen Qureshi, and Laura Poulton
May 1, 2017

Design Manual

Introduction

Our project, Super Omar.io, is a platform-based single-player game. The main software platform we used was JavaFX, and any classes, APIs, libraries, or interfaces used in the program are all part of the JavaFX framework. All the graphics used in the game were created by our team using either Adobe Illustrator or Adobe Photoshop.

When the game starts running, the GameMain class creates an instance of the GameWorld abstract class called SuperOmario, and sets up the game scene. The GameWorld class creates the game loop using the JavaFX class AnimationTimer. The AnimationTimer object gameLoop updates the game animation at a specified number of frames per second. Every frame, an event handler calculates the amount of time since the last frame and uses this calculated frame duration to update the objects in the game accordingly. In SuperOmario, the GameWorld is created and the JavaFX pane, scene, and root node are added and initialize. The AnimationTimer is started and then immediately paused. The start screen image is shown over the game scene until the player presses the ‘S’ key to start the game, and then the image is removed from the scene nodes and the AnimationTimer starts again.

Every object in the game, including the player and the background, is called a sprite. Every frame, the AnimationTimer event handler runs a method called updateSprites which updates the position and/or velocity of the sprite images within the game scene using their current velocity and/or acceleration, and the frame duration time. The AnimationTimer event handler then runs a method called checkCollisions which checks for any intersecting sprites and changes the sprite behavior or removes the sprite as necessary. GameWorld is an abstract class, so each instance of GameWorld can have different implementations of updateSprites and checkCollisions, as well as the initialize method. In SuperOmario, the initialize method adds all the sprites to the root node of the game, displays all necessary text, and uses event handlers to update the velocity, acceleration, and/or behavior of different sprites according to specific user key presses and key releases.

Every sprite in the game uses bindings to keep its size and position relative to the size of the game scene, so the window is somewhat resizeable. Once the game ends, with the player either winning or losing, the appropriate end screen image is shown, and the AnimationTimer is stopped. If the user chooses to restart the game, the GameMain class runs again and creates a

new GameWorld and starts a new AnimationTimer. The user can then play the game as many times as they want.

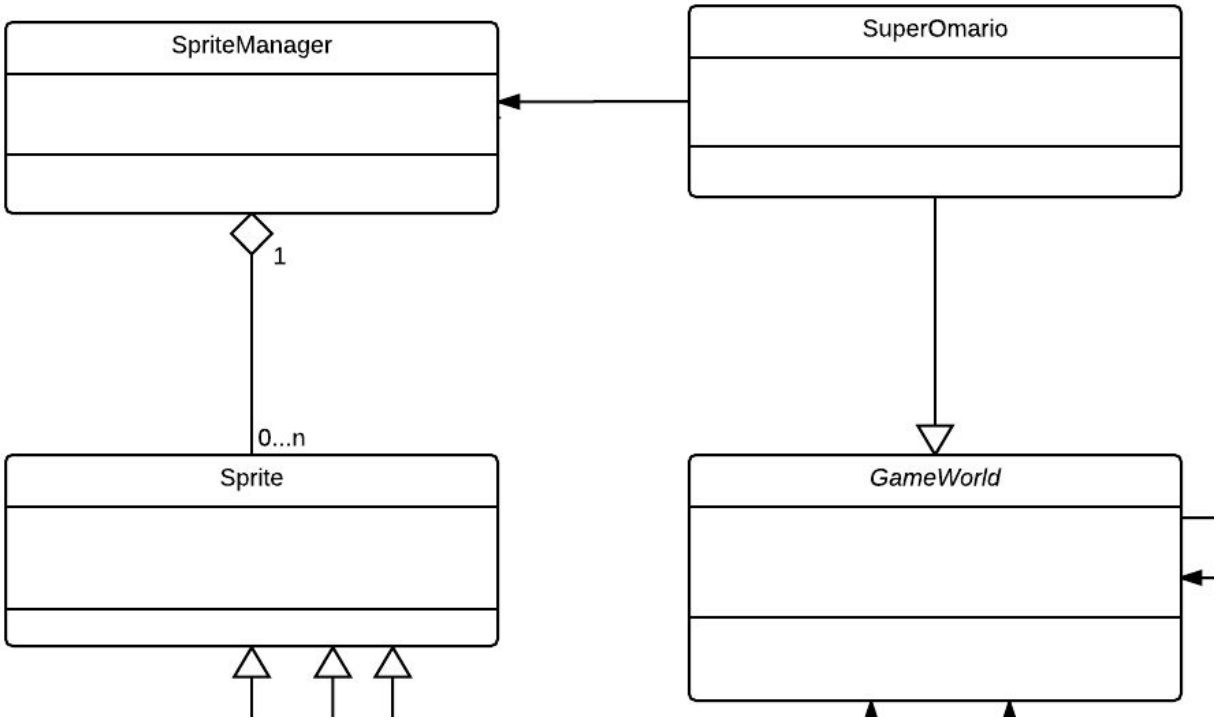
Completed User Stories

1. As a user, I have a character
2. As a user, I want to be able to move my character back and forth across the screen
3. As a user, I want a background with a variety of objects
4. As a user, my character appears to run and jump like a real person
5. As a user, I want the game to have enemies and obstacles
6. As a user, I want objects in the game to be dynamic and interesting
7. As a user, I can win or lose this game
8. As a user, I want to play a well-developed, finished game
9. As a user, I don't want the game to be too easy or too difficult
10. As a user, I want to see some kind of measurement as to my level of success in completing the game

These user stories are centered around creating an enjoyable (and playable) game. The first user story was accomplished by extending the Sprite class to make a Player that could be instantiated and controlled with user input in a wide variety of ways. Within the SuperOmario class, we implement an event handler that listens for keyboard input and controls the movement of not only the Player object (implementing 1,2, and 4), but also controls the movement of the background scrolling, which helps to fulfill user story 6. In order to implement 3, 5, and 6, we created Platform, Background, Enemy, and Coffee classes that could all be specifically controlled to make an environment that really felt like a platform-based game. We were able to use the Enemy class to create "bluescreen" enemies that had to be avoided in order to successfully complete the game. We also used the Coffee class to make little cups of coffee that the user could pick up and add to their score. The Platform class simply acted as surfaces for the player to stand on, while the Background was simply the picture that sits in the back of the scene. We made these two classes as extensions of the Sprite class in order to make it easier to accomplish our goal of creating a scrolling background. The less concrete user stories were mostly accomplished by adjusting certain existing elements of the game. For example, to achieve user story 9, we adjusted the padding on the intersections between the player and the enemies and adjusted the jump velocity to make the game challenging and yet possible to complete. Through these methods and similar ones, we made changes and design decisions to complete the above user stories.

Object Oriented Design

As we brainstormed during the first week, we created a very basic level UML diagram which was accepted as the fundamental design by all of us. This UML was similar to the one given below:



We recognized the need for Sprite classes early on. This would go on to enable us to implement the functionality we wanted for game elements based on the user stories. For instance, consider the following user stories:

- As a user, I want to be able to move my character back and forth across the screen
- As a user, I want the game to have enemies and obstacles

Observe that enemies and the character both consist of animated object moving across the screen. We realized this was a case where inheritance could be employed to reuse existing code in an efficient manner. We then built upon this development to create a collection of CRC cards, which we modified slightly as the project progressed. Regarding the in-game characters, we created the following CRC cards:

Player

- Creates an object represented by the player avatar - Controls the gravity of the player avatar - Controls the acceleration/deceleration - Updates position and dimensions - Check if the player avatar is on the ground	- GameWorld

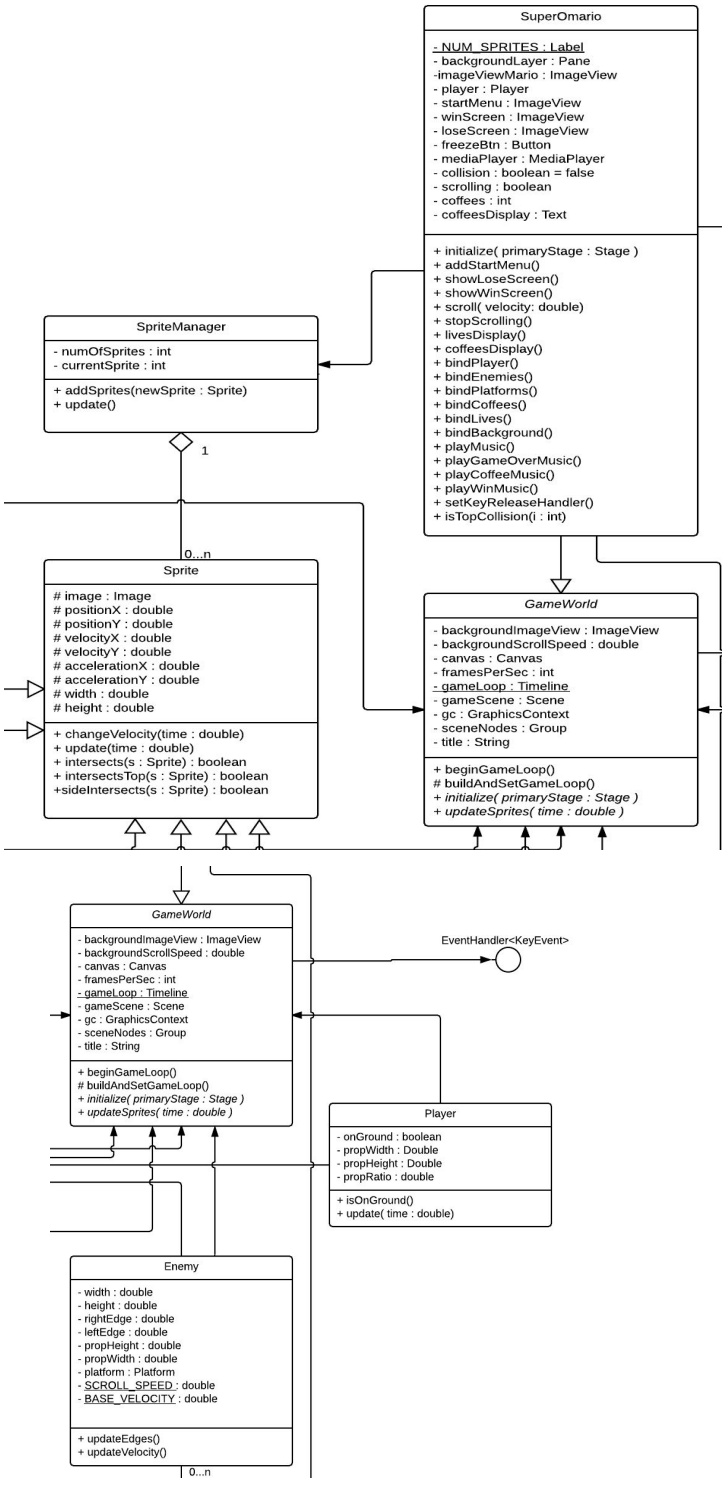
Enemy	
- Creates a new enemy in the game - Handles the visual representation - Handles the speed of the enemy - Handles the oscillatory motion of the enemy	- EnemyManager - GameWorld - Platform

These two classes were designed in response to the following user stories:

- As a user, I have a character
- As a user, my character appears to run and jump like a real person
- As a user, I want the game to have enemies and obstacles

Furthermore, we had Sprite as the parent class of both Player and Enemy, while a separate class named SpriteManager allowed us to have a ‘container’ for the Sprites present in the game and gave us the ability to remove them as required.

These elements were marked as the fundamental components and we proceeded to incorporate it into the UML in the following manner:

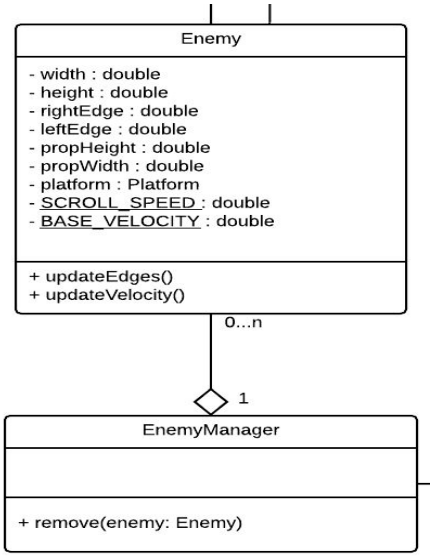


However, while there is one player, there are multiple enemies, all of which share the majority of their functionality and attributes apart from position and existence. More specifically, there was an aggregation of Enemy objects that had to be controlled. We developed a class named

EnemyManager for this purpose with its responsibilities and collaborations denoted by the following CRC card:

EnemyManager	
- Adds an enemy to each platform - Remove an enemy once it is eliminated	- Enemy - GameWorld - SuperOmario - BackgroundManager

This was then incorporated into the UML as follows:



Throughout the Design process, we aimed to minimize coupling and maximize cohesion. Consequently the Sprite descendent classes we designed had a few well-defined methods along with low associativity. These classes included, besides Player and Enemy, Background, Platform, Coffee, Life, and WinFlag. These classes were designed to address the following user stories:

- As a user, I want a background with a variety of objects
- As a user, I want objects in the game to be dynamic and interesting

The CRC cards for those classes are as follows:

Coffee	
- Create a coffee object - Base bindings off of ratio between image dimensions, and initial game dimensions - Make the coffee visible - Update the coffee's velocity - Make it hover - Update the coffee	- GameWorld - SuperOmario

Platform	
- Put Platforms in the game for the Player to move upon - Adjust the proportions of the platforms to the Game Scene	- GameWorld - BackgroundManager

Life	
- Represent the number of lives available to a player - Deduct from the number of lives when player dies	- GameWorld

Background	
- Sets the background image - Sets position of background at beginning of the game Scene	- GameWorld

These sprites were, in turn, managed by the BackgroundManager which is summarized by the following CRC:

BackgroundManager	
- Contains all the Platforms - Contains all the Coffees - Remove a life if player dies - Remove a coffee if player gains one	- GameWorld - Platform - Coffee - Life

The next part was to get a functional GUI which integrated animation. While this was controlled mostly by the SuperOmario class, including the initialization of the GameLoop, some of the animation specifications were handled by the ImageViewSprite class:

ImageViewSprite	
- Displays a Sprite - Sets the viewport and determine where it needs to be positioned - Sets the FPS	- SuperOmario - AnimationTimer