***Coliseum - Detailed Design Document***

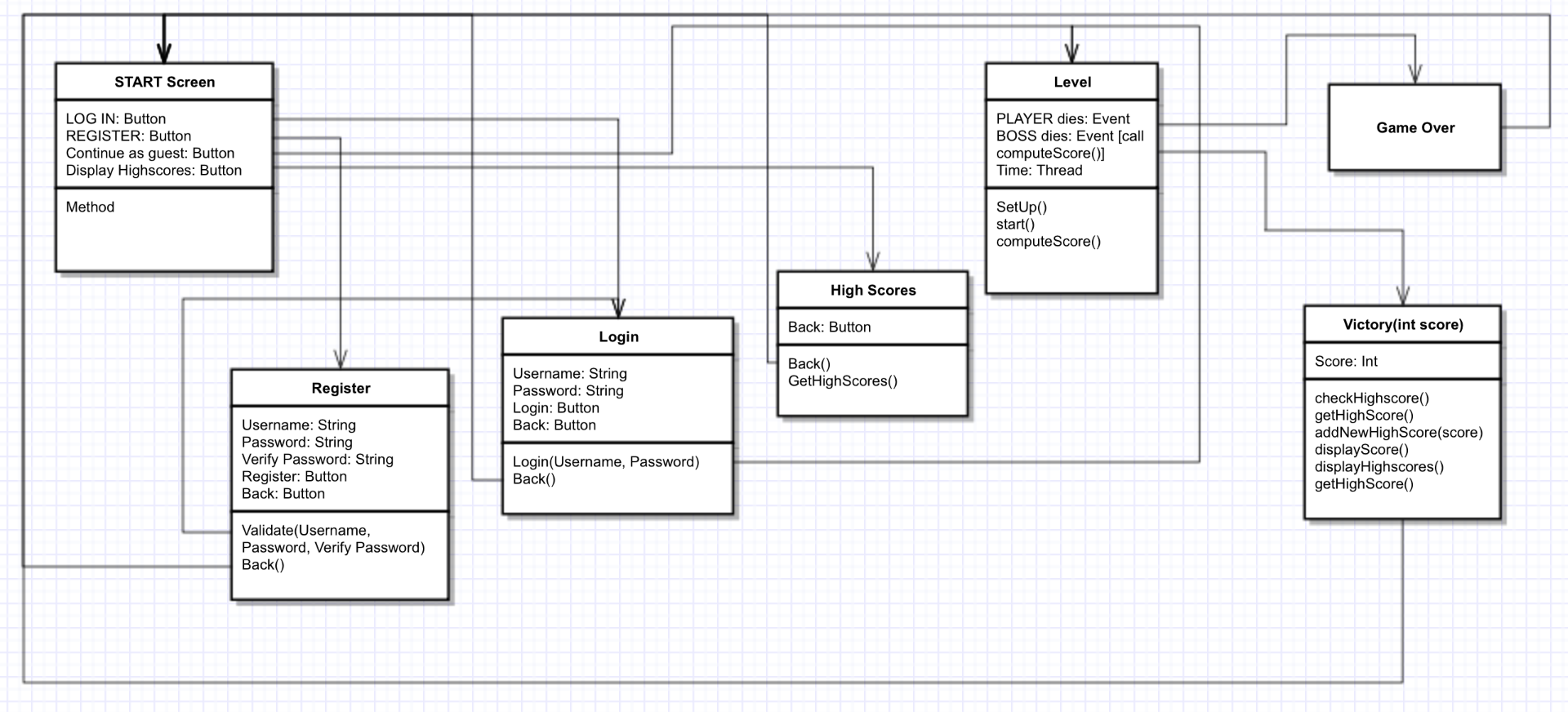
Jun Yen Leung - junyenle@usc.edu

Ethan Barker - esbarker@usc.edu

Jake Leventhal - jglevent@usc.edu

Uri Rahimi - urahimi@usc.edu

Dylan Appleton - dappleto@usc.edu

**Section 1: Broad Program Structure (Class Structure)**

**1a: Flow and Overview**

* **Program Flow From Application Run:**
  + **START screen →** 
    - Has Login, Register, Continue as guest, and Display high scores buttons (functionality below).
    - **Login screen (on Login button click)**
      * Has Username field, Password field, Login button, and Back button.
      * Login button calls login() function, which will check the username and password against the server.
      * Further explained in data section below.
      * If successful, redirects to LEVEL.
      * Back button redirects to START screen.
    - **Register screen (on Register button click)**
      * Has Username field, Password field, Register button, and Back button
      * Register button calls validate() to check Username and Password fields’ validity (check password not null and that username not taken).
        + If validated successfully, calls on the RegisteringUserThread on the server to register user.
      * Further explained in the data section below.\
      * If successful, redirects to login screen.
      * Back button redirects to START screen.
    - **Continue as Guest (button only, no screen - on Continue as guest button click)**
      * Calls login() on a dummy user that does not have high score saving permission.
      * Redirects player to LEVEL.
    - **Display Highscores screen (on Display highscores button click)**
      * Calls getHighScore(), which queries the server and returns the list of highscores.
      * Has back button, which redirects to START screen.
      * Further explained in the data section below.
  + **LEVEL →**
    - setUp() will prepare all source media for the LEVEL and display it on the screen. Player movement is locked while this is running.
    - start() will unlock player character movement as well as start the BOSS AI.
    - there will be a slot called gameOver listening for one of two events:
      * BOSS dies, in which case we run computeScore() and redirect to VICTORY. computeScore() computes the player’s score as a function of time spent and damage taken.
      * Player dies, in which case we redirect to GAME OVER
    - there will be a thread that keeps track of the time the player has been on this LEVEL for.
    - More level details in section 1b below.
  + **VICTORY →** 
    - Displays text (see GUI below)
    - checkHighscore() function which calls getHighScore() on the server and checks if the score returned by computeScore() above is a new highscore. If so, call addNewHighScore(int) on the server.
    - displayScore() function will display the score returned by computeScore() on the screen.
    - displayHighscores() will display the top 10 scores returned by getHighScore().
  + **GAME OVER →**
    - Displays text (see GUI below)
    - Redirects to START screen.

**1b: LEVEL specifics**

Within Unity, every class that uses a C# script automatically inherits from MonoBehaviour which provides a number of functionalities from Unity that allows the object to be functional with the game. In the actual level, there will be two main classes, the player class and the boss class. Both will have a RigidBody2D. This is an element in Unity that allows the object to interact with the physics of the game. Both will also have a number of BoxCollider2D’s. These are an an object in unity that acts as a collision box or hitbox. It can push other object that intersect with it away and/or trigger a script (for example, the collider on the player’s weapon will have a script that decreases the boss’s health, which will be a private float variable.) both will contain a SpriteRenderer which is the object that displays the current sprite to the screen. They will have an Animator object which iterates through an animation and sets the current sprite to the next frame in the animation. The animations are controlled through a Controller object, which chooses what animation to trigger and when.

Other than the boss and the player, there will also be a main camera object. It will contain the Unity Camera object. Along with these will have the background which will simply have a sprite renderer and a float mParallax which will be used to calculate how the camera positions that object with parallax scrolling to add depth to the game. There will also be a bridge object that has a sprite renderer and a float mParallax. It will also have a BoxCollider2D so that the player is able to stand on the bridge.

**Section 2: Hardware / Software Requirements**

**2a: Programs / Tools**

* The majority of actual application design will be done in Unity.
* Sketches, concept art, and game art will be drawn in Adobe Photoshop using Wacom Intuos 4.
* Model vectoring will be done in Inkscape / Photoshop and converted into animation sprite sheets in Inkscape.
* Voice lines will be recorded in Audacity.

**2b: Libraries**

* MongoDB Java Driver
* FreeTTS 1.2.2. A text to speech library used to communicate verbally with the players.

**2c: Programming Languages**

* Server logic: Java
* Client logic: C#

**Section 3: Data Management**

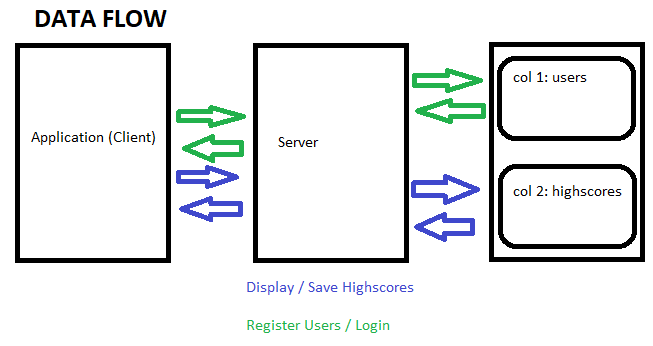
**3a: Database Schema**

* Two-collection MongoDB database.
  + Highscores collection to hold high scores (indexed by score).
  + Users collection to hold registered users’ names and hashed passwords (indexed by username).
  + See 3b for more details on data structure.

**3b: Data Structure**

* One database collection will store usernames and hashed passwords that we will be accessing and modifying as users are registering.
* Another collection will hold the ranking, name of the player, and high score.
* Sample login data collection:{  
   username: “ttrojan”,  
   password: hashed password here  
  },  
  {  
   username: “bbruin”,  
   password: hashed password here  
  }
* Sample highscore data collection:  
  {  
   score: 5080,  
   username: “ttrojan”  
  },  
  {  
   score: 8082,  
   username: “bbruin”  
  }

**3c: Pushing and Pulling Data**

****

**3ci: Saving Highscores**

* New highscores will be added to the database right when gameplay ends.
* After completing a game, a call is made to the server to see if the user’s high score is higher than the current high score with the getHighScore() function, which returns an integer.
  + If the high score returned is greater than the score that the user just received, the addNewHighScore(int score) function will be called.
    - This function will display a text pop-up that notifies the user that he/she received a new high score along with the score that was received, and his/her previous high score.
    - After displaying the data, the user’s high score in the Mongo database will be overwritten.

**3cii: Pulling Highscores**

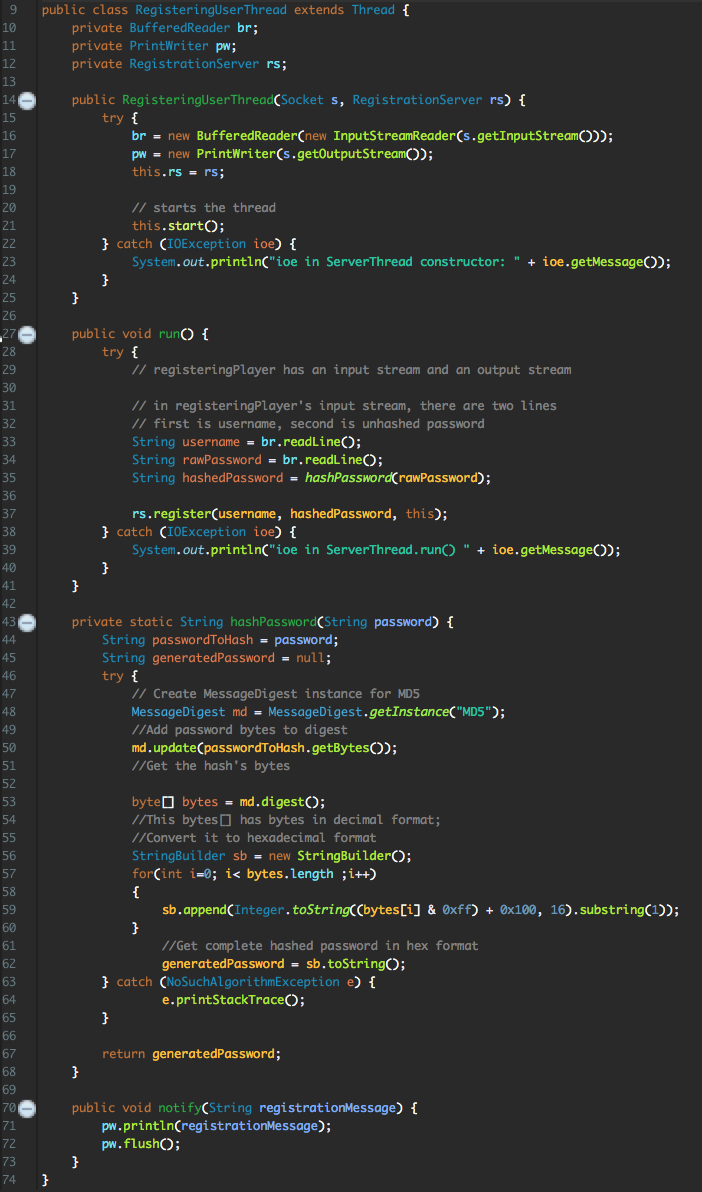
* The database for high scores will be structured in a while loop that we can retrieve the highest score and compare it to the current user’s high score and if need be trigger a text to speech notification notifying the user that they have beaten the previous high score, or their own previous high score.
* Database access will be done using the MongoDB Java Driver.

**3ciii: User Registration**

* At the home screen (start screen), users will be prompted to either login or register which both will be making two separate calls.
  + Registering: There will be three fields for the user to enter data into. First, the user will need to input a username, password, and password confirmation. Upon registering, a validate() function will be called to make sure that the user has entered valid input.
    - If the username that the user is registering with is already taken, an error next to the username field will be displayed.
    - If the password field contains text shorter than 6 characters, an error will be displayed next to the password field.
    - If the text in the password field is not equal to the text in the confirm password field, an error message will be displayed next to the confirm password field.
    - If all the fields are valid, a RegisteringUserThread object is created to represent the user that is registering. This thread contains member variables for a PrintWriter, a BufferedReader, and a reference to the registration server.
      * When the RegisteringUserThread is created, the username and password to register with is read from its socket’s input stream.
      * The password is then hashed so that I can be stored securely using the RegisteringUserThread’s hashPassword(String rawPassword) function.
        + This function performs bitwise operations on the password to transform it to a 32 character hashed version of the Password.
      * The registration server’s register(String username, String password, RegisteringUserThread registeringUser) method is then called.
        + This adds the new username and password to the Mongo database and then notifies the user of successful or failed registration via the registeringUser’s notify(String registrationMessage) method.

This method simply flushes the registrationMessage to the RegisteringUserThread’s output stream so that it can be displayed on the client’s front end.

* + - After the registration is successful, we will trigger a text-to-speech event notifying the player that registration was a success.
    - The code for what RegisteringUserThread, including hashPassword(String rawPassword) and notify(String registrationMessage) functions.

****

* + - After registration, redirect to the login screen.

**3civ: User Login**

* The user login will simply check and verify that the username and hashed password matches a corresponding existing entry in the username/password database table via the login() method.
  + This will make a call to the LoginServer which will make the actual database check.
    - If there is a match for the username and the hashed password, the user will be logged in.
    - Otherwise, an error message will be displayed notifying the user that the username and password combination is invalid.
* Once this has been accomplished we will trigger a text-to-speech event welcoming the player to the game.

**Section 4: Graphics**

**4a: GUI Design**

**4a0: Global Specifications**

* The application will run in fullscreen only. The GUI will be designed for 1080p (1920x1080 pixels) screens.

**4ai: Pre-LEVEL**

* The first screen shown to the user (START screen) should have four buttons: Log in, Register, Display High Scores, and Continue as Guest.
* Behind these buttons, there should be START screen art.
* The START screen should look like this:

****

* The Login and Register screens should look like this (except that “Register” should read “Login” if logging in):

****

**4aii: LEVEL**

* The playing arena on which the actual gameplay will take place (LEVEL) will be a long platform with the hostile agent (BOSS) in the center. The BOSS will be located at the center of the platform and make attacks from the midground(explained in later sections). The camera may move with the player character, depending on the LEVEL.
* The player’s health will be displayed in the top left of the screen.
* There will be a background, on which there will be a painting of mountains.
* The platform will exist on the foreground.
* The foreground is the only level where character hitboxes will exist.
* The LEVEL should look like this:

****

**4aiii: Post-LEVEL**

* The post-LEVEL screen that displays depends on whether or not the player won the game.
* If the player won, the VICTORY screen should be displayed. The VICTORY screen should show the player’s score as well as the list of high scores (see section 3 on data management). This screen should eventually fade back to the START screen (within 10-20 seconds).
* If the player lost, the GAME OVER screen should be displayed. This screen should print “GAME OVER” and then fade back to the START screen (within 10-20 seconds).
* The VICTORY screen should look like this:

****

* The GAME OVER screen should look like this:

****

**4b: Modelling and Animation**

**4bi: Environment**

* The LEVEL’s background (the sky and mountains in the figure in 4aii above) should have some looping animations. For instance, there should be moving clouds. The LEVEL art will be painted in Photoshop.

**4bii: Player Character (PC)**

* The PC will be drawn by hand, vectored in Photoshop / Inkscape and converted into sprite sheets for animation in Inkscape.
* PC actions will be animated in the same way.
* The PC should look roughly like this:

****

**4biii: Hostile Agent (BOSS)**

* Like the PC, the BOSS will be drawn by hand, vectored in Photoshop / Inkscape, and converted into sprite sheets for animation in Inkscape.
* Critically, the BOSS needs two sets of sprites: one for its main body (which will live in the midground) and one for the body parts that will attack (which will live in the foreground). For instance, his fists, which will need to punch at the player, must exist in the foreground.
* Then, each BOSS action also needs two sets of animations, for the background BOSS movement (e.g. raising arm) and the corresponding foreground movement (e.g. fist coming down on platform).
* The BOSS should look roughly like this:

****

* As an example of foreground models, the BOSS’ fist should look roughly like this:

****

**4biv: Flavor Animations**

* Miscellaneous looping animations (e.g. clouds, waterfalls on the mountains) will be drawn in Photoshop. These will be applied in separate layers with equal parallax factors to the layers that they correspond to so as to seamlessly integrate them into the static art.

**Section 5: Sound**

**5a: Soundtrack**

* All soundtracks will be taken from free soundtrack sources online.

**5b: Sound effects**

* All sound effects will be taken from free soundtrack sources online.
* BOSS voices will be recorded by us using Audacity or similar software.

**Section 6: Game Mechanics**

**6a: Player Character (PC)**

**6ai: Movement**

* The player will be able to move left if the left key or the ‘a’ key is held. The player will move right if the right key or the ‘d’ key is held. When the spacebar is pressed, the player will jump, and the jump height can be controlled by how long the spacebar is held. If shift is pressed along with a left or right direction button, the player will do a dash in that direction. The dash will also cause the player’s BoxCollider2D to be turned off for the duration of the dash so that the player can go through the boss’s attacks without being hurt.

**6aii: Attacks**

* If the player clicks the left mouse button, an attack will be triggered. First it will check whether the player is in the air or on the ground. The player will have a different attack depending on whether they are on the ground or in the air. A BoxCollider2D will be created over the player’s weapon, and an animation will be triggered. If the box collider intersects with the boss’s box collider, the boss will lose hp from his health variable.

**6aiii: Taking Hits / Vitality**

* If the player’s box collider intersects with one of the box colliders for a boss’s attack, the player’s HP variable will be decreased and the player’s velocity will be adjusted so that they move back in the direction of the attack a bit. If the player’s HP reaches 0, the game over screen will be triggered.

**6biv: Comments**

**6b: Hostile Agent (BOSS)**

**6bi: Movement**

* The boss will not move except for its attack animations.

**6bii: Attacks**

* While the boss’s body will appear in the midground, the attacks of the boss will appear in the foreground. This will be achieved by syncing the sprites of the boss’s body with the sprites of the boss’s arm for the attack. The boss will decide what attack to use by calculating the distance between itself and the player. We do this by subtracting the boss’s x and y position from the player’s. We can also find the angle from the boss and the player by using atan 2. The boss will have 2 attacks. If the player is in the air, the boss will do a stab towards the player. If the player is near them and on the ground, the boss will pound his fist down on top of the player. And if the player is far away and on the ground, the boss will sweep his arm along the bridge. All of these attacks will have box colliders, and if they collide with the players box collider, the player’s health variable will be decreased.

**6biii: Taking Hits / Vitality**

* As touched on in 6aii, if the boss’s health variable will be decreased if its box collider intersect with one of the player’s attacks’ box collider. If the boss’s HP score reaches 0, the victor screen will be triggered, and the player’s score will be pushed to the server.

**6biv: Comments**

**6c: Environment**

**6ci: PC Collision**

* There is a bridge object that has a box collider. This will be the ground that the player stands on. The player will also be able to collide with the boss’s attacks which will decrease his health and send him backwards.

**6cii: BOSS Collision**

* The only collision for the actual boss will be its collision boxes intersecting with the player’s attacks, which is discussed in 6biii.

**6ciii: Depth Through Parallax Scrolling**

* Every object will have a float mParallax which is the parallax factor. The camera will determine that object’s position by finding the x and y position of the center of the object, then subtracting from that center the mParallax of the object multiplied by the x and y coordinates of the center of the camera.

**6d: Scoring**

* Once the boss’s health reaches 0, the score will be calculated. We will have an ideal time which will be the fastest possible time that the boss could be defeated if the player just spammed attack and could not get hit. Then the player’s time will be subtracted from the ideal time. That number (in seconds) will be divided by the percent of health the player lost (meaning the lower percent, the higher the resulting number). Lastly this number will be multiplied by 1000 to increase the score to give the player the sense that their accomplishment means more.