Minigame Dungeon

Design and Planning Document

Document Revision History

Rev. 1.0 <2019-10-07>: initial version

Contents

1. System Architecture

1.1 Overview

We have a mobile game built with Unity's Entity-Component System, a standard unity architecture that is easy to implement, that is run with a flask server.

1.2 Mobile Game

The mobile game's system architecture will include Unity's Entity-Component System, a minigame interface, and a user interface.

1.2.1 Unity's Entity-Component System[1]

The Unity Engine uses the Entity-Component System (ECS) which is comprised of:

A. Entities, which group components and systems. Colloquially, these are called *game objects.*
B. Components, which hold the data of the game
C. Systems, which provide logic to the game.

We can look at minigames as an example of ECS. A minigame is ultimately an *entity.* The ui objects, their positions on the screen and success criteria are data contained in components. The gameplay, mechanics, and decision making are systems which are defined in C# scripts.

Another important part of the ECS is the concept of *prefabs*[2]. Prefabs allow you to replicate a previously defined entity and its components and systems. Since entities cannot be created during runtime, prefabs are an important element that allows us to more easily and efficiently manage our data.

1.2.2 Gameplay Overview

After clicking play a randomly picked minigame appears. Once the current minigame ends, it will flash a black screen to transition to the next minigame. At the end of all the minigames a screen will pop up that you have made it through the dungeon and won, and then it will give you the options to go back to sign up or go back to the homescreen.

---

[1] https://docs.unity3d.com/Packages/com.unity.entities@0.1/manual/ecs_core.html
[2] https://www.google.com/search?client=firefox-b-1-d&q=unity+prefabs

List of minigames (we are going to do at least 2 any more are stretch goals):

Minigame #1 - Dodge the arrow: Tile the phone to move the character away from the incoming arrows.

Minigame #2 - Swim: Tap quickly for the character to move to the surface before the timer runs out.

Minigame #3 - Shoot the bird: Aim and power the bow to shoot the flying bird.

Minigame #4 - Jump the logs: Tap when the logs are close to jump over the logs.

Minigames #5 - Catapult Time: Aim and power the catapult at the castle and bring it down.

Minigame #6 - Dodge the Boulders: Tilt or tap the phone to move the character out of the way from the falling boulder.

Minigame #7 - Run from the Boulder: like Indiana Jones tap quickly to run from the boulder.

Minigame #8 - Block the Boulder: Tap the corresponding lever to move the barrier to block the incoming boulder.

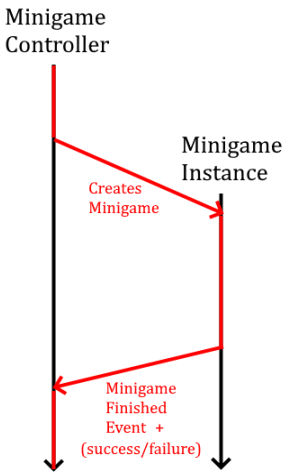
Minigame #9 - Throw the Javelin: In the time quickly tap as many times as you can to throw the javelin as far as possible.

1.2.3 Minigame Interface

Following Unity's Entity-Component model, each minigame can be an entity prefab. A singleton minigame controller, called the MinigameController, contains a list of minigame prefabs. Each prefab contains a Minigame component which allows it to, when finished, use Minigame.finish(bool) to fire off an event in the MinigameController that it has finished and that the MinigameController is clear to destroy it, play a transition, and move on to the next minigame.

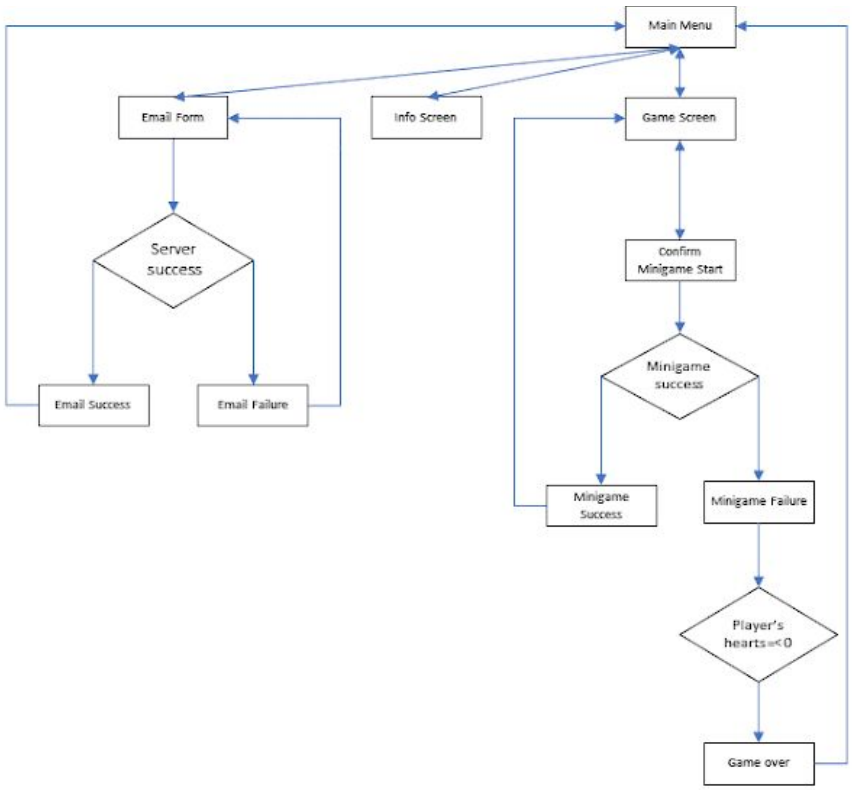
Other than the Minigame.finish(bool) interface, minigames are allowed to implement any other features specific to the minigame.

All minigames, since they follow Unity's Entity-Component System, will also have Start and Update methods for running their own game logic.



1.2.4 User Interface

The user interface will facilitate gameplay by directing the user and giving them responsive feedback to communicate internal responses. UI flow is shown in the following diagram.



Main menu - The main menu is the first screen the user sees. It has 3 buttons that change the screen the user sees

1. Email Form
2. Information
3. Game Screen

Email Form - The email form screen is the screen that allows the user to submit an email to the server to sign up for GDD. There is a text input box for users to enter their email and a button to send the email. When the email is sent a boolean value is returned. If the server returns true, then the email was valid and submitted so the *Email Success* screen is shown. If the server returns false then there was some error and the *Email Failure* screen is shown.

Email Success - The email success screen is a pop-up over the Email Form screen which has one button that returns the user to the main menu.

Email Failure - The email success screen is a pop-up over the Email Form screen which has one button that returns the user to the email form screen to try again.

Information - The information screen explains to the player what GDD is and credits us as the creators of this app. There is one button which returns the player to the main menu.

Game Screen - The game screen displays the "dungeon" that the player goes through and allows the player to start a minigame. The screen shows how many minigames the player has completed and how many hearts There are 3 buttons:

1. A pause button

2. A button to return to the main menu, which loads the main menu
3. A button to begin the next minigame, which loads the *Confirm Minigame Start* screen

Confirm Minigame Start - This is a popup which allows the player to make sure that they want to start playing. It has 2 buttons:

1. An exit button that returns the player to the game screen
2. A button which allows the ui to be transferred over to the specific minigame requested

The minigame creator will be responsible for all ui in the minigame as this project is purposefully meant to be expanded upon. One requirement is that the minigame return a boolean which says if the player won the minigame or not. If the player won the minigame, the *Minigame Success* screen is shown. Otherwise, the *Minigame Failure* screen is shown.

Minigame Success - The minigame success screen congratulates the player and shows their new level. There is one button which returns them to the game screen.

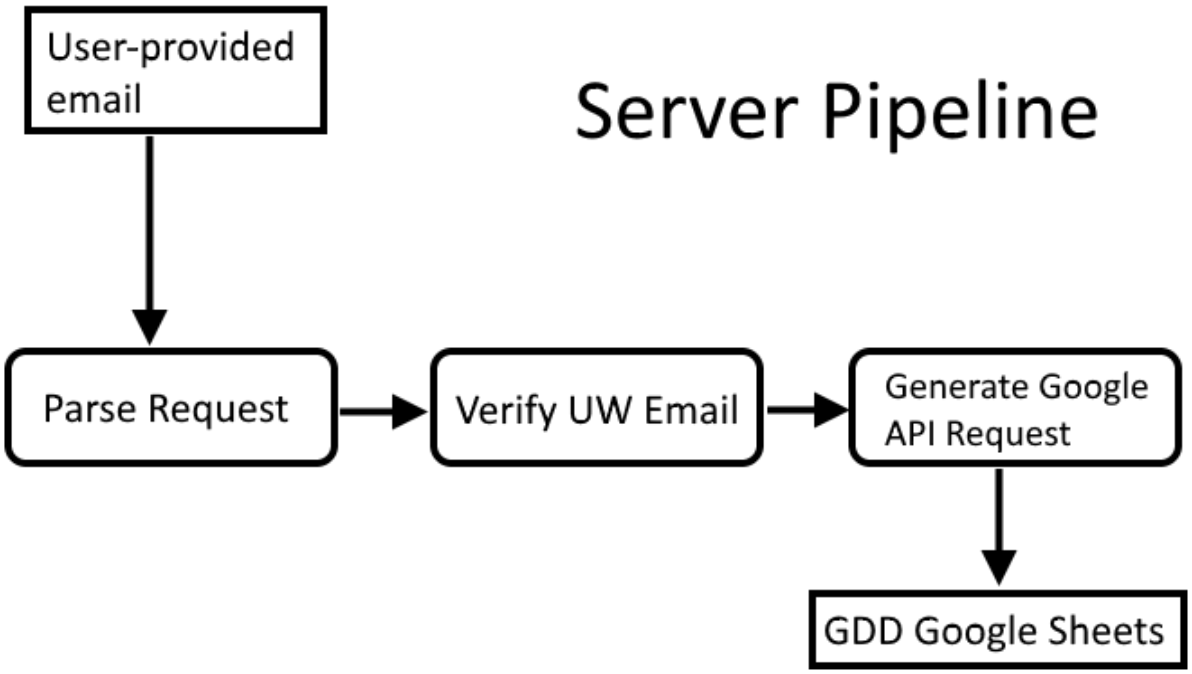
Minigame Failure - The minigame failure screen informs the player that their heart has decreased. There is one button that allows the player to continue. When this button is pressed, the decrement and then the isDead functions in PlayerHealth are called. If the isDead returns true, the *Game Over* screen is called. Otherwise, the player is returned to the game screen.

Game Over - The game over screen informs the player that their run has ended. There is a continue button which returns them to the main menu.

1.3 Server Pipeline

The primary purpose of the web server is to take HTTP requests containing the emails of users who wish to sign up for GDD, verify that they are UW emails, and then append them to the GDD Google Sheets page.

The Web Server that the Mobile Game sends HTTP requests to will use the pipe-and-filter architecture. This architecture was chosen because it best fits the server's use case: users provide emails to the server, and these emails must then be filtered and appended to a Google Sheets page. Users cannot fetch data from the server or interact with it in any other way, which makes other architectures such as MVC irrelevant.



For the stretch goal of adding user confirmation and unsubscription, the pipeline will be integrated with an event-driven model where receiving a User Confirmation or User Unsubscription event with the correct token causes the server to append to the google sheets page. HTTPS support is also a stretch goal.

1.4 Design Risks

One risk with our current design is there could be too few minigames so it could get boring and seem uneventful. There are also risks with the designs of our minigames. We want the minigames we create to be challenging enough to be interesting but not too challenging that it's not fun. A major risk is that Github has issues with Unity and can make it hard to compile the code.

2. Design Details

2.1 Mobile Game

After clicking play, the player walks forward and a randomly picked minigame appears.

List of minigames (we are going to do at least 2, any more are stretch goals):

Minigame #1 Dodge the arrow: Tile the phone to move the character away from the incoming arrows.

Minigame #2 Swim: Tap quickly for the character to move to the surface before the timer runs out.

Minigame #3 Shoot the bird: Aim and power the bow to shoot the flying bird.

Minigame #4 Jump the logs: Tap when the logs are close to jump over the logs.

Minigames #5 Catapult Time: Aim and power the catapult at the castle and bring it down.

Minigame #6 Dodge the Boulders: Tilt or tap the phone to move the character out of the way from the falling boulder.

Minigame #7 Run from the Boulder: like Indiana Jones tap quickly to run from the boulder.

Minigames #8 Block the Boulder: Tap the corresponding lever to move the barrier to block the incoming boulder.

Minigames #9 Throw the Javelin: In the time quickly tap as many times as you can to throw the javelin as far as possible.

If the user succeeds at the minigame, the player's score will increase, and the player will move onto the next room, where another minigame will appear.

If the user fails at the minigame, the player's score will not increase, and the player will take damage (as shown by a heart breaking above their head, and the heart count on the UI

decreasing). If this would cause the user's health to fall to 0, the game over screen is shown instead of moving onto another minigame.

On the game over screen, users can choose to either restart the game, or sign up for GDD.

2.1.1 User Interface

The UI will be controlled by a UIController

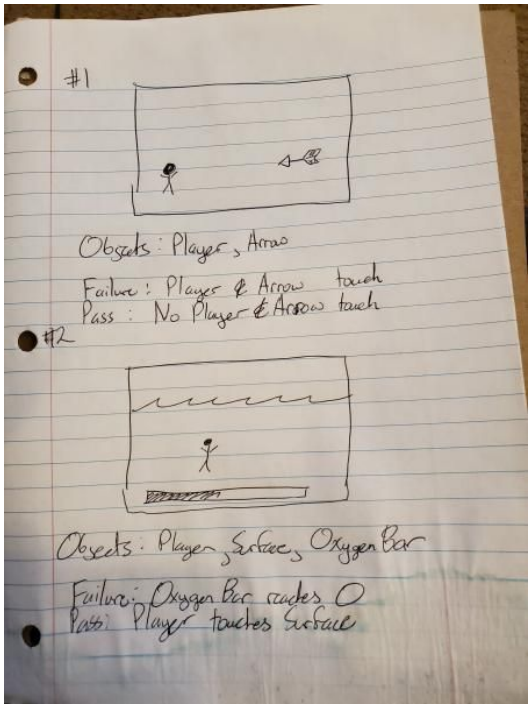
UIController
UIController.ShowScreen (string screenName): void
Instantiates screenName

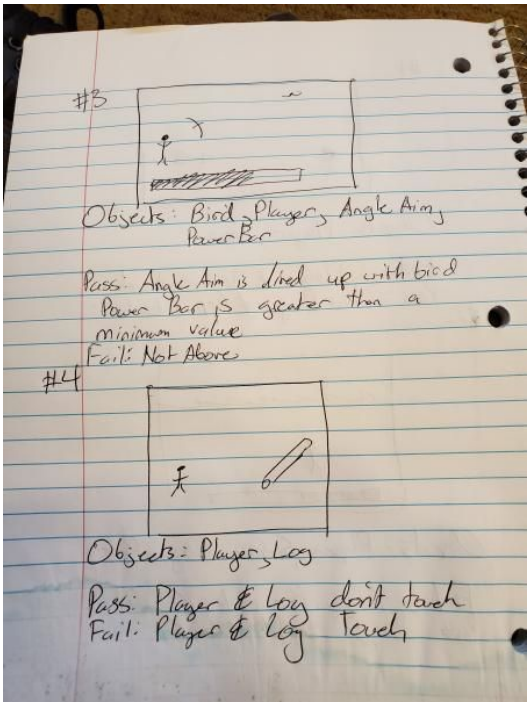
UIController.PauseGame(bool isPaused): void
Pauses the game if the player leaves the application

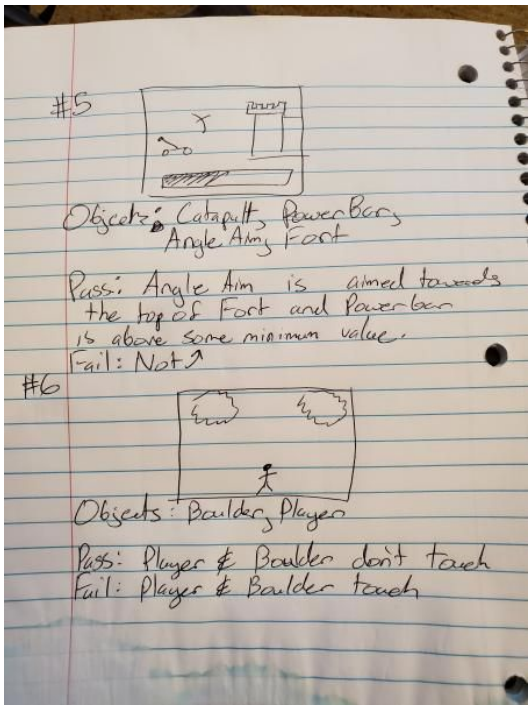
UIController.StartMinigame ( ): void
Starts a minigame by running a selection algorithm and instantiating the minigame.

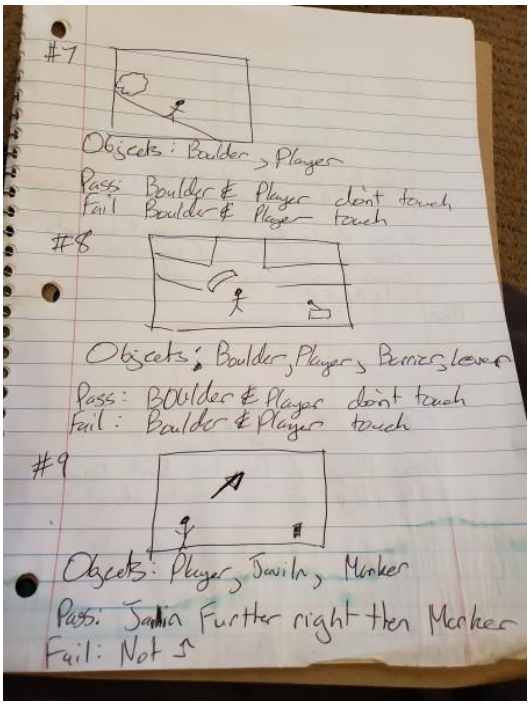
UIController.MinigameWon (boolean success)
Returns feedback based on whether the minigame was won or lost

2.1.2 Gameplay









2.1.2.1 Player

Components

PlayerHealth: MonoBehaviour
Interacts with the UI to display the player's health. Causes a game over if health causes to 0.

PlayerAnimation: MonoBehaviour
Controls the player's movement animations during transitions and minigames

Methods

PlayerHealth.reset(): void
Resets player health to full. Called on game start/restart.

PlayerHealth.decrement(): void
Removes one heart from player health. Called on minigame failure.

PlayerHealth.isDead(): boolean
Returns whether or not the player's health is equal to or less than 0.

PlayerAnimation.trigger(String): void
Triggers a specific player animation to play, such as Jump

2.1.2.2 MinigameController

Components

MinigameController: MonoBehaviour
Contains an Unity-inspector maintained list of valid minigames that can be chosen at random. Contains a state machine that goes through the states:
Transition to Minigame -> Play Minigame -> Finish Minigame -> Transition to new Minigame -> etc, until player health runs out.
Causes UI to display game over screen if player health runs out.

Methods

MinigameController.startMinigame(): void

Choose and create a minigame at random.

MinigameController.finishMinigame(bool): void
Called by the Minigame component. Notifies the MinigameController that the minigame has finished and that it is free to destroy it and play a transition.

MinigameController.transition(bool): bool
Triggers a specific transition to play, either success or failure. At the end of this, player a new minigame or goes to game over screen, depending on health.

2.1.2.3 Minigame (Blueprint)

Components

Minigame: MonoBehaviour

Implements the interface that the minigame ues to talk to the MinigameController via finish(bool).

ExampleMinigameComponent: MonoBehaviour

Minigame-specific components, such as one that handles player platforming in a platforming minigame or handles screen input for a swiping minigame, is pre-loaded into each minigame prefab. These components will be different on each minigame, since each minigame is different.

We will also have blueprints for each class of minigame, such as a platformer blueprint for platformer minigames, to facilitate creating more of them.

ExamplePlatformer: MonoBehaviour

Contains a list of solid blocks, a start point, and a goal. Enables physics on the player, and allows the player to move left, right, and jump.

Methods

Minigame.finish(bool): void
Notifies the MinigameController that the minigame has finished.

2.1.2.4 Transition

Components

Transition: MonoBehaviour

Handles player animations during minigame transitions.

Methods

Transition.playTransition(bool): void
Plays a success or failure animation from the Unity Animator.

2.1.3 HTTP Request to Server

The Sign-Up button in the user interface, when clicked, submits an HTTP request of the form: HTTP POST: http://scrollingnumbers.com:42069/signup?email=<user_email>
The POST request will have no body, and will only have the email URL argument. This request is implemented using the UnityWebRequest library.

For the HTTPS stretch goal, all of the HTTP request will instead use HTTPS.

2.2 Flask Server

The Flask Server will be a python module that can be run as a Flask Server.

Attributes

SCOPES: Tuple of Strings
Used to notify Google that we want both read and write permissions on the GDD Google Sheets page.

SPREADSHEET_ID: String
Identifies the GDD Email list signup Google Sheets page that the server appends to.

SHEET_RANGE: String

Google Sheets range identifier for where on the Google Sheets page the user emails column is located.

SPREADSHEET: Spreadsheet Object
Google Sheets API object used to communicate with the Google Sheets API.

SMTP: SMTP Object
Python smtplib object used to verify UW emails and send confirmation emails.

Methods

@app.route("/signup")
def signup(): Flask HTTP Response
Allows in HTTP requests to /signup?email=<user email>. Runs the server pipeline on the received email by checking if its a valid email address, checking that it's an existing UW email, and checking that it does not already exist in the GDD Google Sheets page. If all these conditions pass, it appends the email to the GDD Google Sheets page email column and sends a confirmation email to the target email address.

def check_email(): boolean
Ensures that the email is a valid email address using a regex. Returns true if valid, false if not.

def check_email_new(): boolean
Ensures that the email does not already exist in the google sheets page. Returns true if it is a new email, false if not.

def check_email_uw(): boolean
Ensures that the email is a valid UW email by contacting the wisc.edu SMTP server. Returns true if it is an existing UW email, false if not.

def add_to_sheet(): boolean
Appends the email to the GDD Google Sheets page. Returns true on success, false on failure.

def send_confirmation_email(): boolean
Appends the email to the GDD Google Sheets page. Returns true on success, false on failure.

Stretch Goal Methods

def store_user_token(email, token): boolean
Stores an email token on the local server sqlite database. Returns true on success, false on failure.

def confirm_user(email, token): boolean
Appends the email to the GDD Google Sheets page if the email token is correct. Returns true on success, false on invalid token or network error.

def unsubscribe_user(email, token): boolean
Removes the email from the GDD Google Sheets page if the email token is correct. Returns true on success, false on invalid token or network error.

2.4 Alternative Design Ideas

We considered multiple design ideas before settling on our current version which is above. We thought about having a list of minigames that you chose from, but chose to to have them be continuous so that it keeps users engaged and playing the game. We also considered forcing users to play the game and then at the end would sign up for the club, but instead added a feature at the initial home screen to sign up for the club to maximize sign ups.

3. Implementation Plan

3.1 Overview

This project will be broken into three iterations. Iteration one will focus on getting everything set up and having somewhat of a working prototype. Iteration two in summary will be all about testing and implementing minigames. Finally, in iteration three we hope to make everything pretty and complete stretch goals if iteration one and two go smoothly.

Time units for tasks in the iterations will be based off the table below.

T-shirt size	Time Required
XS	1 hr
S	2 hrs
M	4 hrs
L	6 hrs
XL	8 hrs

3.2 Iteration 1

Get things set up and rolling. Should have a main menu screen, an option to sign up for the club, and two to three minigames. Will be setting up all of the UI. Need to create scene transitions between these. Set up all minigame utilities including inputs. Will need to set up the server for signing up and have the option to unsubscribe from emails. There will be unit tests for subscribing and unsubscribing from the email list.

Stage 1		Menu	
Task	**Time Units**	**Dependencies**	**Assigned Person(s)**
Task 1.1 Create an application with a menu screen with an info button, and an email sign up button, and a play button.	S		Lyn
Task 1.2 Confirm that the application opens and closes	XS	Iteration1: 1.1	Vinoth
Stage 2		**Google Sheets**	
Task	**Time Units**	**Dependencies**	**Assigned Person(s)**
Task 2.1 Contact client and get an API key that can write to their Google sheets page	XS		Emma
Task 2.2 Formulate a Google sheets API request that can append one email to it	S	Iteration1: 2.1	Vinoth
Stage 3		**Flask Server**	
Task	**Time Units**	**Dependencies**	**Assigned Person(s)**
Task 3.1 Create the sign up Flask route	S		Evans

Task 3.2 Deploying the Flask server to the target machine	S	Iteration1: 3.1	Evans
Task 3.3 Setting up port forwarding in DNS	S	Iteration1: 3.2	Evans
Stage 4		**Minigame Controllers**	
Task	**Time Units**	**Dependencies**	**Assigned Person(s)**
Task 4.1 Implement *MinigameController.startMinigame()*	S		Jimmy
Task 4.2 Implement *MinigameController.finishMinigame(bool)*	S	Iteration1: 4.1	Jimmy
Task 4.3 Implement *MinigameController.transition(bool)*	M	Iteration1: 4.1, 4.2	Jimmy
Stage 5		**Player Methods**	
Task	**Time Units**	**Dependencies**	**Assigned Person(s)**
Task 5.1 Implement *PlayerHealth.reset()*	S		Jimmy
Stage 6		**Create Minigames**	
Task	**Time Units**	**Dependencies**	**Assigned Person(s)**
Task 6.1 Create minigame1	XL	Iteration1: 4.1	Emma, Lyn
Task 6.2 Implement *Minigame.finish(bool)*	S	Iteration1: 4.1, 4.2	Jimmy
Task 6.3 Create	L	Iteration1: 4.1	Jimmy

minigame2			
Stage 7		**Testing**	
Task 7.1 Server unit testing	M	Iteration1: 3.3	Vinoth

3.3 Iteration 2

The focus of Iteration 2 is on get things ready to test for users and the client. Iteration 2 will include implementation of 10 minigames.

Stage 1		**HTTP Request to Server**	
Task	**Time Units**	**Dependencies**	**Assigned Person(s)**
Taske 1.1 Set up HTTPS	L	Iteration1: 3.3	Evans
Task 1.2 Set up user confirmation	M	Iteration1: 3.3	Evans
Stage 2		**Player Methods**	
Task	**Time Units**	**Dependencies**	**Assigned Person(s)**
Task 2.1 Implement *PlayerHealth.decrement()*	S	Iteration1: 5.1	Lyn
Task 2.2 Implement *PlayerAnimation.trigger(String)*	S		Lyn
Stage 3		**Create Minigames**	
Task	**Time Units**	**Dependencies**	**Assigned Person(s)**
Task 3.1 Create	M	Iteration1: 6.1	Vinoth

minigame 3			
Task 3.2 Create minigame 4	M	Iteration1: 6.1	Vinoth
Task 3.3 Create minigame 5	M	Iteration1: 6.1	Emma
Task 3.4 Create minigame 6	M	Iteration1: 6.1	Emma
Task 3.5 Create minigame 7	M	Iteration1: 6.1	Emma
Task 3.6 Create minigame 8	M	Iteration1: 6.1	Emma
Task 3.7 Create minigame 9	M	Iteration1: 6.1	Emma
Task 3.8 Create minigame 10	M	Iteration1: 6.1	Emma
Stage 4		**Testing**	
Task 4.1 Integration Testing	M	Everything previous to this task	Jimmy
Task 4.2 Begin user testing	M	Iteration2: 4.1	Vinoth

3.4 Iteration 3

Make everything pretty! Complete stretch goals such as adding extra minigames, have the users confirm that they want to sign up (in case someone uses the wrong email), and other finishing details. We will be adding more tasks and details to this iteration throughout iterations one and two.

Stage 1		Stretch Goals	
Task	**Time Units**	**Dependencies**	**Assigned Person(s)**
Task 1.1 Create a score system for the Minigames	M	Iteration1: 6.2	Lyn
Task 1.2 Create a transition between the minigames	M	Iteration1: 6.2	Lyn
Task 1.3 Create more animations for the player character	M		Lyn
Task 1.4 Email Verification	L	Iteration1: 2.1	Jimmy
Task 1.5 Create extra minigames	L	Iteraion1: 6.1	Everyone!
Stage 2		**Bug Fixing**	
Task	**Time Units**	**Dependencies**	**Assigned Person(s)**
2.1 Continue Integration testing	M	Iteration2: 4.1	Lyn
2.2 Continue Unit testing	M	Iteration2: 4.1	Evans
2.3 Continue beta testing with GDD club	M	Iteration3: 2.1	Jimmy, Vinoth
Stage 3		**Google Play Store**	
Task	**Time Units**	**Dependencies**	**Assigned Person(s)**
Task 3.1 Submit the final apk onto the Google Play Store	L	Iteration3: 2.3	Jimmy, Emma

4. Testing Plan

We will be working together in person to test our project as we go, testing several times a week as we add functionality. We want to use unit tests for testing the UI and the server and the functionality of the app. A lot of the minigame testing will have to be done by inspection.

4.1 Unit Testing

The goal of unit testing to see that the components are able to give the output that we expect. Unit testing will ensure that any changes we make to the modules will not break the program. We will be relying heavily on unit tests for our program.

In iteration 1, we will use unit testing on the server and UI. In iteration 2, we will continue to use unit testing to verify that our server and UI function, but we will do extensive integration testing on the minigames and game experience as a whole.

4.1.1 Server Side Unit Tests

The server is developed using Flask. We can write unit tests by using the request library and python's unit testing framework PyUnit to test the server we have written. To do this, we will test server connectivity, email verification and email-adding via three unit tests:

Server Connectivity: send a request to http://scrollingnumbers.com/signup. We expect to receive error code 400.

Email Verification: First, remove eschen3@wisc.edu from the google sheets page. Then, send a request to http://scrollingnumbers.com/signup?email=eschen3@wisc.edu. We expect 200 OK. Then, send a request to http://scrollingnumbers.com/signup?evansschen@gmail.com. We expect error code 400 since this is not a UW email. Finally, send a request to http://scrollingnumbers.com/signup?email=eschen3@wisc.edu again. We expect error code 400 since this is now a duplicate.

Email-adding: First, remove eschen3@wisc.edu from the google sheets page. Verify via the google sheets API that it is gone. Then, add eschen3@wisc.edu to the google sheets page via a request to /signup. Ensure that it is on the google sheets page via the google sheets API.

4.1.2 Android Unit Tests

We will use Unity to write the Android app. We will write unit tests for this in C# using UnityTest, Unity's built-in testing framework.

The main unit test that we will conduct is that the MinigameController creates a minigame, and then responds to a Minigame.finish call. This is done via a dummy Minigame that instantly finishes. We then inspect MinigameController state to ensure that it cleans up the Minigame and proceeds to the next.

4.1.3 UI Unit Tests

The UI can easily be tested with Unity's built in testing-framework. We will test the following:

Screen-swapping: the correct screens are instantiated based on pseudo-button clicks.

Death: When the player's health is decreased to 0, the game over screen shows.

Minigame Correctly Won: If the UI receives a true variable, level should equal level+1. If the UI receives false, health should equal health - 1.

4.2 Integration Testing

It is not feasible to unit-test minigames on a method-by-method basis, so all minigames will be tested by the minigame creator once the minigame is integrated into the full Unity project. Thus, we will do integration testing on each minigame as it is integrated with the MinigameController.

Additionally, integration testing includes integration between the mobile game UI and the Flask Server. We will perform this integration testing to ensure that pressing the Signup button with a valid/invalid email creates the correct result on the GDD google sheets page.

4.3 Performance Testing

There are no performance requirements for the mobile game itself. For the server, we will use top to ensure that memory and performance requirements are met as the server handles requests. We will also time the response from the server between the HTTP request from the client to the server's response.

4.4 Compatibility Testing

We will test our app on Android devices of varying screen size, using PC emulators such as BlueStacks, as well as borrowed tablets and personal phones.

4.5 Regression Testing

Regression testing will be conducted at the end of every iteration, as well as every push to the master branch on our GitHub. This regression testing includes a run of all unit tests, as well as a manual integration test of all affected entities in the mobile game.

4.6 Beta Testing

We plan to beat up our app and play the game ourselves many times, following a list of possible breaking points that we will write once we have the game more developed. We also are going to have our client, the president of the GDD club and the club members play the app and give us feedback on our work. We will do this during iterations two and three and then go through all of that feedback and take it into account to improve with our final drafts of the app.

5. FAQs

5.1. Are all parts of the document in agreement with the product requirements?

Yes. No parts of this document disagree with the product requirements.

5.2. Are all the external interfaces to the system specified in detail?

Yes throughout entire document.

5.3. Is the architecture of the system described, with the major components and their interfaces?

See Section One.

5.4. Is there sufficient detail in the design to start Iteration 1?

See sections 3.2 and 3.3 for details on iteration 1 and 2 for details on design.

5.5. Is there a reasonable detail in the design for features in future iterations?

See section 2 for details on the design and 3.3 and 3.4 for details on future iterations.

5.6. Is there risk associated with the proposed design? Is it discussed in the document?

See section 1.4 on the design risks.

5.7. Are the limitations of the specified implementation sufficiently documented?

See section 1.3 for limitations for user confirmation and unsubscribing, section 1.4 for limitations with Github and Unity, section 4 for limitations with having to test the minigames with test by inspection, and see all references to “stretch goals” throughout the doc which are limited by time.

5.8. Have alternative designs been considered and documented?

See section 2.4 on alternative design ideas.

5.9. Are the different testing activities covered and described?

See section 4 on testing.

5.10. Does the design take into account testability of the various units, components, and subsystems?

See section 4 for all testing notes.

5.11. Are the sets of features for each iteration identified?

See section 3 for all the details on the iterations and specific tasks to accomplish the features.

5.12. Is the plan for Iteration 1 sufficiently complete to start the implementation?

Yes, see section 3.2 for details on Iteration 1.

5.13. Are the testing activities scheduled at the appropriate times?

We will be doing testing every Friday at our regular meeting time, and will also have an extra focus of testing during Iteration 2, see section 3.3.

5.14. Are the subteams identified and has enough thought been given to parallelization of effort and team dependencies?

See section 3 with iteration details which specifies specific people to work on tasks and specific phases to give order to when tasks should be completed.

5.15. Is the solution at a fairly consistent and appropriate level of detail?

Used numbered organizations for consistency and the document is over 25 pages long.

5.16. Is the solution clear enough to be turned over to an independent group for implementation and still be understood?

Details throughout specifying everything you could possibly need to build this application.

5.17. Is the document making good use of semi-formal notation (UML, diagrams, etc.)?

See sections 1.2.3, 1.2.4, 1.3, 2.1.2, and 3.

5.18. Is the document identifying common architectural or design patterns, where appropriate?

See sections 1 and 2.