Minigame Dungeon

Requirements and Specification Document

2019-09-25, version 1.0

Project Abstract

Minigame Dungeon will be a mobile game where the user plays an adventurer exploring a dungeon. The gameplay consists of a series of quick minigames such as: navigating the player around a maze, jumping to avoid some obstacles, or swiping across the screen to attack a monster. The player has 5 seconds to complete each minigame, and on success, the player's score will increase. If the player fails, the adventurer will take damage. Then, the player will advance to the next minigame. If the adventurer runs out of health, then the adventurer will die, and the current score will be displayed on the screen. The player may then play again, exit to the main menu, or sign up for Game Design and Development (GDD) by entering their UW email.

Document Revision History

Rev. 1.0 2019-09-25: initial version

Customer

Our client, Mary Xu, is the president of GDD at UW-Madison, which caters to students at the University of Wisconsin Madison who are interested in gaming or game development. Mary's primary need in this project is a way to promote GDD and expand membership.

We will be working with her for the duration of the semester to get guidance on requirements and feedback on the development process. She will be the one who will use the system once it is complete to market the club and recruit new members.

Competitive Landscape

Our competitors for this project are somewhat odd. Since the purpose/use case of this app is to convince other students to join GDD in situations like club fairs, our competitors are really other clubs like the Software Design club or the Undergrad Projects Lab.

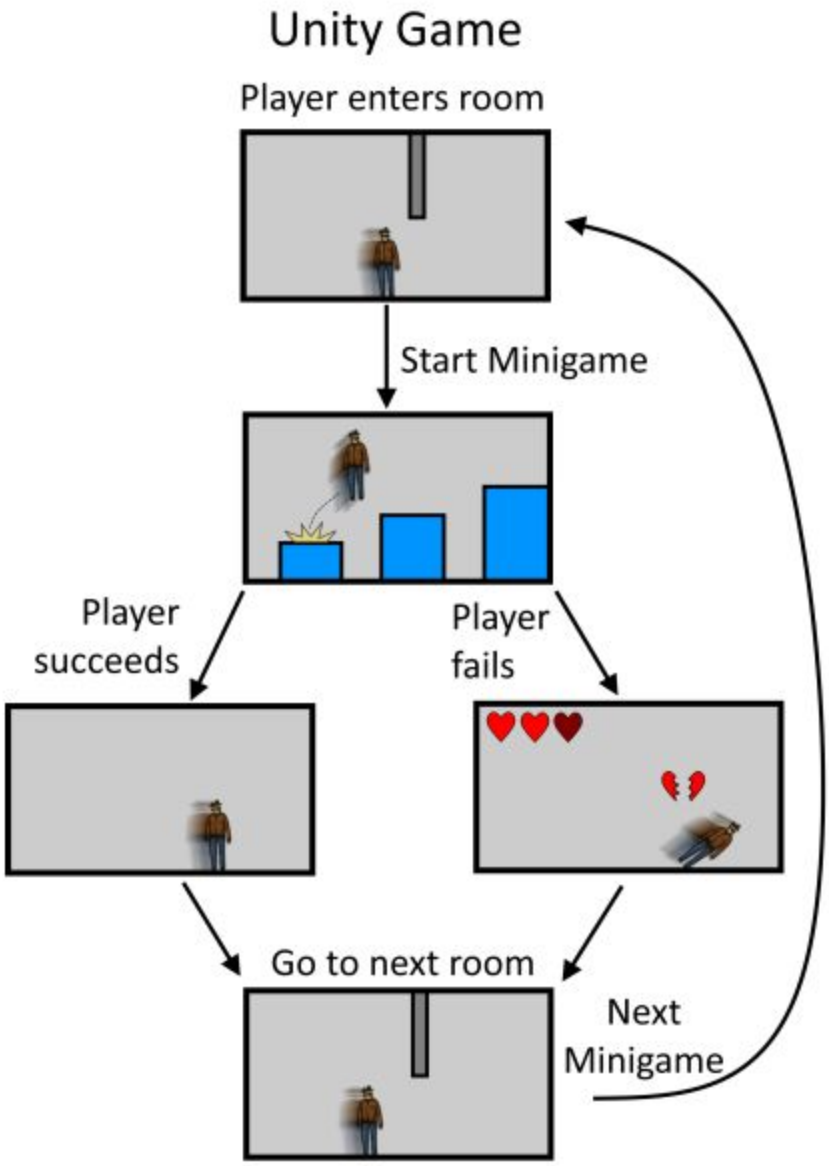
Two of the strongest competitors are Badgerloop and Wisconsin Robotics since they also have a product potential users can interact with. To be able to effectively draw users we will need a snappy product that can be portable to org fairs which is why we chose this solution.

The technological competitors are entertainment apps that can run on android. The flaw of most of these entertainment apps is that they are similar to each other. The strength of monetized entertainment apps is that they are meant to be profitable and so investment in marketing and advertising is justified because an increase in users increases revenue. That said, since our product is not meant to be profitable per say and instead is built for a single customer, we do not have to engage in marketing schemes.

User Requirements

Bold are higher priority; Italics are lower priority

- Main menu
    - Play dungeon game
        - **Play series of minigames**
            - **Have at least one minigame**
            - *Have 20+ minigames*
        - Once dead
            - Option to return to main menu
            - Option to play again
            - Option to sign up for GDD with wiscmail
    - **Sign up for GDD with wiscmail**
    - Information



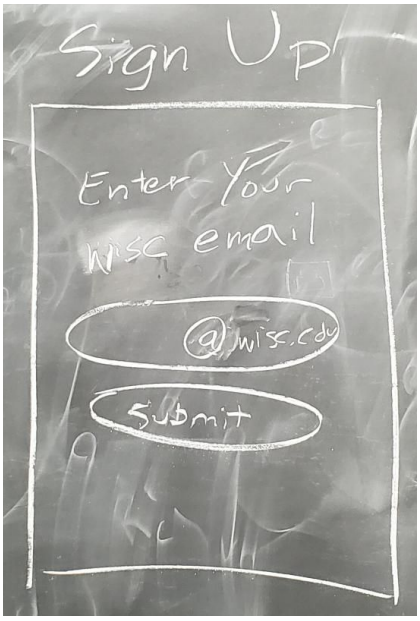
Use Cases

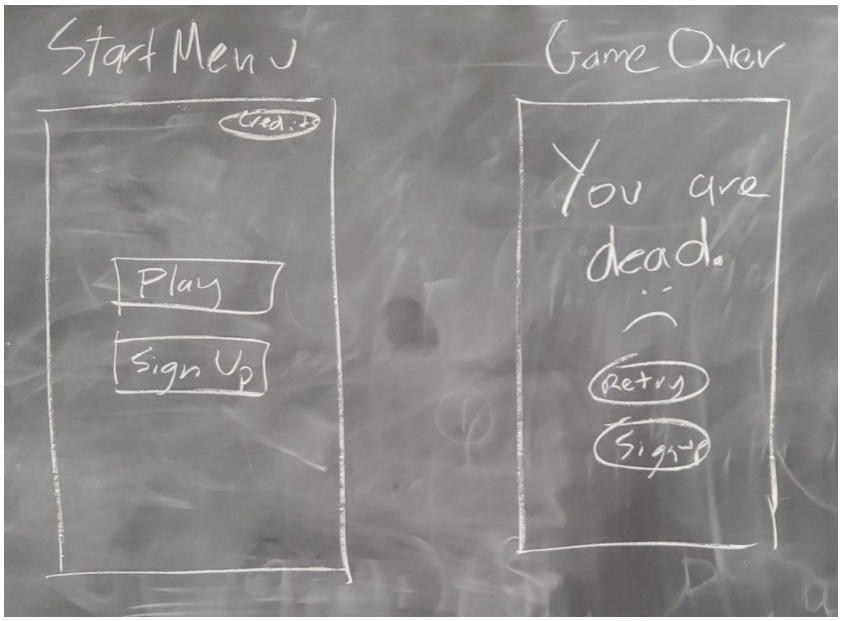
Must have	**Useful**	**Optional**
Start the application and close the application - Unit test that shows everything is started properly - Unit test that shows everything closes properly	Die if you fail to much, when you die you can play again - Unit test that reduces hearts to catch if you die - Unit test that makes you die and then starts a new game. All	Many minigames in sequence, have an order for them - Ensure that distribution of minigames is fairly uniform

	data should be reset	
Play a minigame - Test that game works as expected and can transition into and out of the game	Multiple minigames in sequence - Be able to exit out of one minigame and progress to another	Pause button during games - Test to ensure game can seamlessly resume after pausing
Sign up for GDD - Unit test with various emails		

User Interface Requirements

Input login information text boxes. Buttons for play, play again, sign up, main menu. Various taps swipes and moving device around for minigames. All minigame required actions should be shown with either an arrow for direction of swiping, shrinking circle for taps or a moving phone icon for moving the phone. All interfaces should have a similar style and look good.





Security Requirements

Users must only be able to input a correctly formed email to the Flask server. Additionally, all emails added to the Google Sheets must be an existing UW email. Any email that is malformed or does not exist as a UW email must be rejected by the server. Any duplicate emails that already exist on the Google Sheets page must also be rejected. Users must not be able to access GDD's Google Sheets

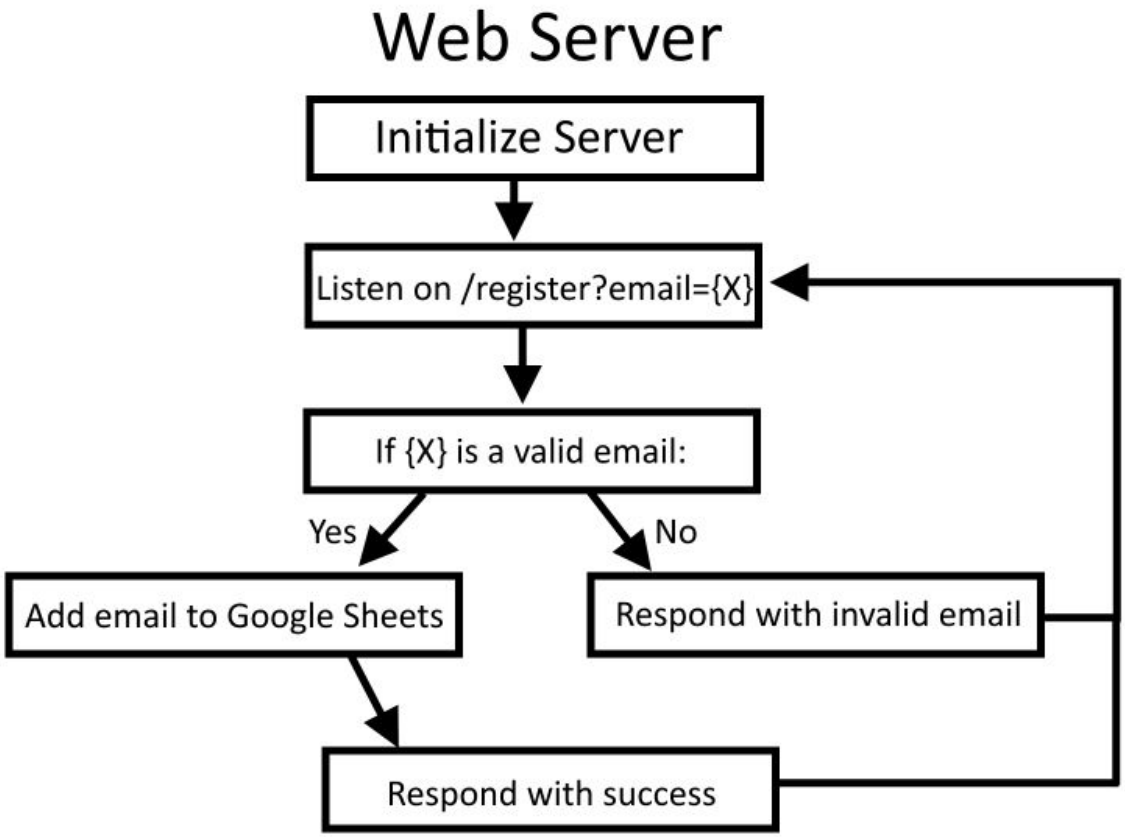
page, and users must not be able to obtain any other user's email address. The Flask server may be vulnerable to high traffic or denial of service attacks, but the Google Sheets page itself must never be accessed by a user or written to by a user other than for adding their UW email.

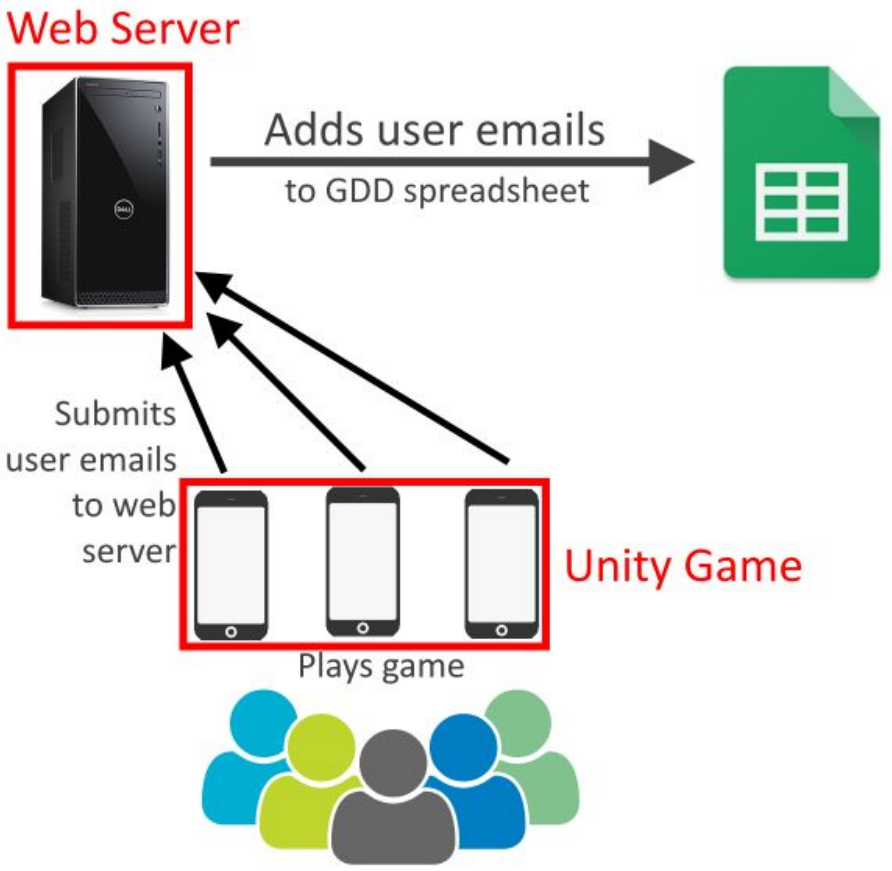
System Requirements

The web server part of the project must be created using the Python language, depending on the Flask library for server hosting and the Google Sheets API for access to the Google Sheets page. Additionally, the smtplib library must be used to check for valid UW emails, and the server must run on a Linux machine, limited to 8GB of memory. The amount of time between when the web server receives a request and when it outputs either success or failure back to the client must be less than 1 minute. The Google Sheets page must be limited to 20,000 entries, each with emails less than 500 characters long.

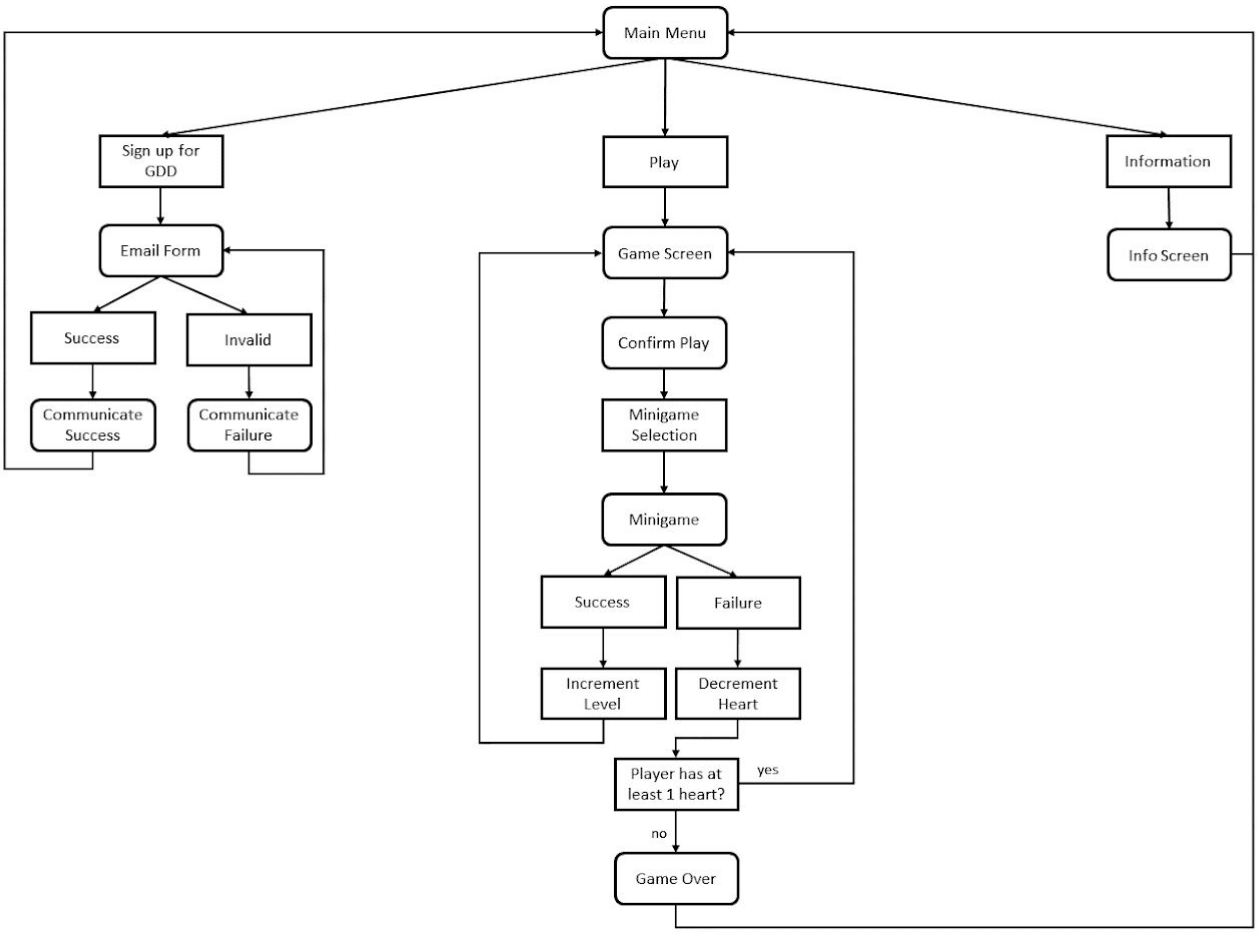
The mobile game part of the project must depend on the Unity Engine, must need to depend on the Android SDK for building for Android, and must need to be published to the Google Play Store. The mobile application itself must be limited to 1GB of memory. Due to the large variations in phone quality, there are no performance requirements for the mobile game itself. However, the game must be compatible with Android versions newer than Android 4.2 (Jellybean).

Specification


Web Server
Initialize Server
Listen on /register?email={X}
If {X} is a valid email:
Yes
No
Add email to Google Sheets
Respond with invalid email
Respond with success



On the Web server, it first configures Flask to receive web requests. Then, it listens for requests until a request arrives. When a web request arrives, it retrieves the email from the URL. If the email is not provided, or the email is malformed, it will return error code 400. Then, the server will use smtplib to send a registration confirmation email. If the email succeeds, the server returns success back to the user's app. If the confirmation email fails to send, then the server will return failure back to the user's app.



From a high level, this project consists of a Unity game, which the user will play and possibly use to register for GDD, as well as a web server, which receives these user emails from the app via HTTP request and submits them to the GDD Google Sheets page via the Google Sheets API.