**Software Implementation and Testing Document**

**For**

**Group Tanx**

Increment 3

**Authors**:

Blake A

Andrew T

Nolan M

Alex G

# Programming Languages (5 points)

We are using Javascript across our project with the Phaser interface. We chose this because we are not particularly well versed with game design and this is a developer-friendly tool that allows people with our experience level to build functional projects efficiently and in a straightforward manner. Their documentation and source code is found in github and they provide extensive support for starting the development.

The game menu is being built using html and css for its structure and stylings. The stages also use html to hop from scene to the stage selection menu, so players cannot just jump from level to level without completing the previous levels. We also used save-keys to make sure players couldn’t access stages and skip them. This allowed us to format the game in the browser and let players easily view the necessary menus and stages for the player.

(This change occurred during iteration 1, but we have left documentation here for reference.)

This has changed from our initial plan of using C++’s SDL library, since we could get working quicker with Javascript. The main reason for this switch was the ease of starting up a project and getting the team on the same page. Additionally, the resources published online are a little more helpful for getting to know the system, as they are a little more modern and oriented for a project similar to ours.

# Platforms, APIs, Databases, and other technologies used (5 points)

We used Phaser’s library to build our game in JavaScript. We chose this because of the simplicity to get it up and running. Instead of focusing on implementing a more difficult and complex system, we got Phaser running quickly and turned our attention to the small details that hopefully made our game stand out in how smooth and well rounded the game feels.

We built the game’s menu, stage selection, game scene references in portions in HTML, so that the user can see a simple format that plainly displayed the options to advance in the game. For simple elements, we have found this to be much more useful, as the design elements are very straightforward and easy to add. It also allows for a much easier and navigable UI so users would know how to get from one stage to another easily. We also used keys to access the levels in order to prevent players from skipping around and playing different levels without having completed or attempting other levels.

We are using MAMP and XAMPP with Apache servers to host our testing and game design through our localhost. This allowed us to have things working in a web browser without having to open up a window or download another feature. Within the browser, we also use the developer tools that come with whichever browser we chose. This helped display errors in our code and allow us to see exactly where in our code we were having issues with. Without this, debugging would have been much more difficult as whenever the game crashed, we would get a blank screen or have features in the game not work.

We shared our progress on coding in GitHub to streamline working together with pulls and commits. For sharing the different files we were working on (like this file and other requirements), we used google drive, as it allows simultaneous editing on documents and sharing them with ease.

We also used Atom as our IDE because of its simplicity and usefulness across all of our intended uses. Atom also helps users keep track of scoping and mass finding/replacing of words so we could edit our code quickly and efficiently.

# Execution-based Functional Testing (10 points)

We had tested our functional requirements through simple, straightforward tests. We would use a test environment then to make sure certain aspects worked inside the AI and Player classes so we didn’t have to deal with conflicting ideas and multiple moving parts. Once we figured out the issues, we would then implement all of these classes in the levels themselves to ensure certain features worked and test out if a level was missing a core function or not. We ensured that no feature would interfere with another feature, or if for some reason it did, we would make sure that certain things were implemented in a certain way to preserve scoping issues and unreferenced variable problems. For example, the AI relies on player position to shoot directly at the player, so the player class must have been instantiated before the AI classes were called to ensure there were no issues when creating the AI.

# Execution-based Non-Functional Testing (10 points)

We pretty much ran all of the non-functional testing with the functional testing as we ensured everything ran with the code given. The only real non-functional item we didn’t test specifically with the functional code was the game menus as those for a while were tested to make sure the menus linked and worked amongst themselves. Otherwise, everything was tested as our non-functional requirements and they often overlapped.

# Non-Execution-based Testing (10 points)

We did some code review whenever someone had an error that they could not figure out or if some code looked verbose and ineffectual. Most individuals took amongst themselves to make sure their code was well structured and could be implemented correctly. We would hold meetings about twice a week towards the end of the semester, gradually increasing to meeting everyday or twice a day to ensure that everyone’s code was working and made progress. Again, whenever an issue came up that they couldn’t figure out, one or two others would look at the issue and come up with a fix in a reasonable amount of time.