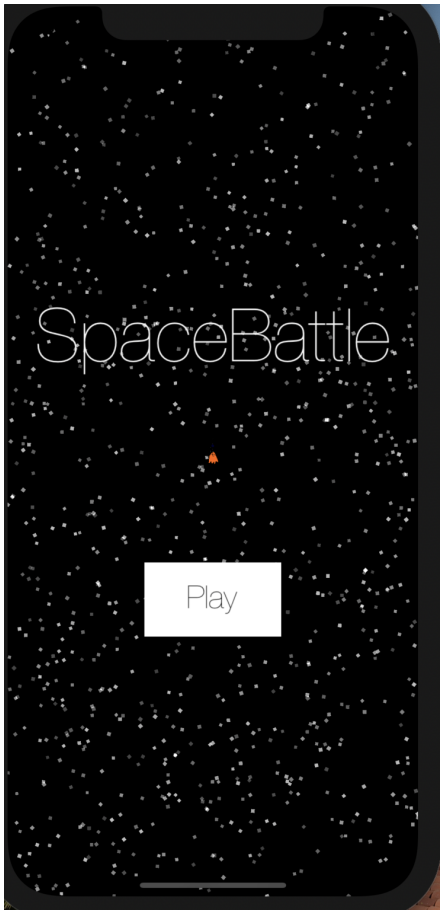
The Space Battle project is authored by Peter Chun, Ryan Frederick, and Ka Wong. When presented with the project, the group began by brainstorming various ideas ranging from an app for UMD's bus system to a mobile game about a character roaming around shooting enemies. The team ended up choosing the latter. During the following brainstorming session, the original concept of the game changed as we gained more knowledge about what could be achievable within the ios platform within our given time frame.

The concept of the game was to have the main character be able to roam around in a closed-off map that had enemies in various locations spawning. The player could control the character's movement and shoot at enemies to accumulate points. These points could then be spent after playing a run to purchase upgrades for their ship. This way, as the player progressed through the game they would be able to achieve higher scores with their upgrades. This progression would keep the game interesting for the player longer and avoid it becoming too routine. Since the game Among Us was recently huge in popularity we decided to go with a standard space theme. Including sprites of space monsters, lasers, and star backgrounds.



Our original proposal had several minimal goals including player movement, player projectile shooting, creating a single map, a health system, screen transitions for new game and game over, leaderboard, sentry enemies that shot lasers back, random spawning, and sound effects for shooting and damage taking. As our development continued in the following weeks,

some of these goals took different iterations. However, in the end, all of these minimal goals were met, and the fact that these goals were written down helped us stay focused.

During the first few weeks of development, the team divided up the tasks and began our individual endeavors. There was a lot of research into how Swift sprite kit-based games function. There were a lot of questions regarding things like how the sprites would interact with each other, how the map would be designed, how to change to a different screen? The answers to these questions began to become more clear as we made more progress and gained more experience writing code for a swift-based game.

The theme of the game was initially thought to be set in a spaceship, with a spacesuited person as the main character. However, because the laser shooting is based on the direction the character is pointing towards, we wanted to change the image model to something that better reflected the direction it was pointed. We chose to use a spaceship instead as it has a pointed side and still fits the overall theme. The sprites we utilized in the game were free pngs provided by sites like OpenGameArt.org, which we put into the assets folder to be used by our SKSpriteNodes.

For this project we relied on the SpritKit and GameKit imports to use as the framework for our game. Rather than setting up horizontal and vertical stacks we had to create Nodes and edit their positions to create our UI and objects. One of the interesting concepts that we had to learn was about bit masking and how that affected collisions and physics. Each of the sprites could be assigned their own bitmask of 1’s and 0’s. And, depending on which bitmask is assigned, different sprites could interact with each other. There are properties like isDynamic, which tell whether the sprites move in the plane after colliding with another object. We initially

had some issues with this property, because if we wanted to check whether a laser hit an enemy or player, one of those sprites would have to have the property isDynamic set to true. At first, we had both objects set to true, which caused the enemies and players to bounce around after being hit, which looked cool, but wasn't intended. Eventually, we realized we had to only make the laser beam's isDynamic true.

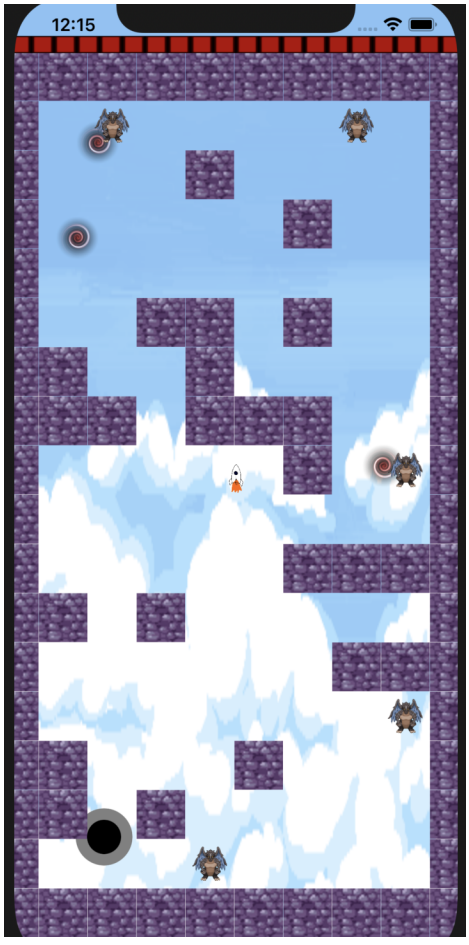
Initially, we considered moving the character using four buttons on the bottom left of the screen, which would move the character left, right, up, or down. However, the lack of diagonal movement made the game less interesting when navigating through the map. It felt more similar to PacMan rather than flying around whilst dodging lasers. So, we went with a joystick approach which consisted of a lower and upper SKSpriteNode. When the upper part of the joystick is touched it signals a flag that the ship could be moving. The velocity of the ship was then determined by how far the upper joystick was from the lower joystick in both the X and Y directions. With these X and Y magnitudes we then calculated the angle of Z-rotation to be applied to the ship so it would face the direction of its movement. This was done by taking the arctangent of the Y over the X. However, the angle was slightly off as we wanted the resting position to be when the ship was facing straight up. So, we printed out the angle when the joystick was going straight up to figure out that we had to subtract 1.57 from the angle.

Furthermore, when developing the movement with the joystick we had the code which updated the player's position in the touchesMoved function. Doing it this way produced smooth movement but only when the joystick was being adjusted, once the player moved it all the way in one direction the movement of the ship would completely stop. Instead, we set a moving flag in the touchesMoved to true and set it to false in the touchesEnded function. This way the update function could be used to always be moving the player when the moving flag was set to true. A

last location was stored from the last time the joystick was touched, so that the ship could continue moving even when the joystick got maxed out. Adding the joystick greatly improved the movement of the ship from the initial version. The increased complexity of the movement made the game slightly more difficult, but far more satisfying to control the ship. The use of four directional buttons was not only simplistic, but also felt somewhat clunky to transition from one direction to another.

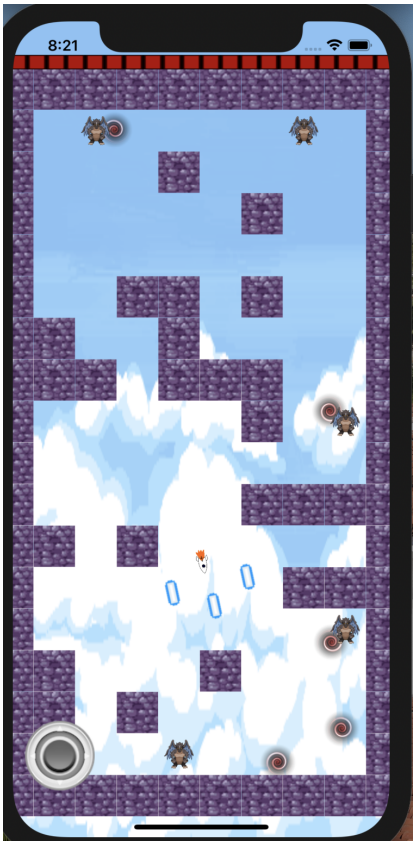
Sound effects were also a big part of the game, we set sound effects for the player's beam, when enemies die, and when a player is hit by an enemy vortex. The sound effects were surprisingly easy to implement, the only thing we had to be careful of was to put them in the correct file directory.

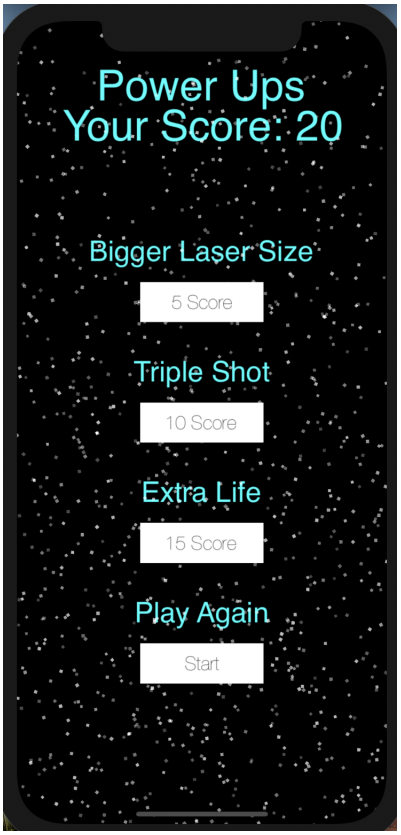
The map creating and initial spawning of the enemies was done in a somewhat unique fashion. We took a .txt file and populated it with characters arranged in a grid-like map that would eventually be converted into sprites. This gave us the flexibility to make easy and quick changes to the map and allowed us to make easy adjustments when we had issues with the resolution of different screen sizes. The x's in the .txt file represent walls on the map and the v's represent the enemies. Initially, we didn't think to have the enemies spawn at random at first, so they were manually placed where they were. Thus, when it came time to program a way to have enemies spawn at random, we had to think creatively. At first, we had issues



with enemies spawning on top of walls, other enemies, and outside of the map. Eventually, we realized that we could leverage the blank spaces that were in the .txt file to set bounds to where new enemies could spawn. This proved to be a good method as it allowed enemies to spawn at random without any way for them to spawn on top of a wall or another enemy.

Following the end of a game, the player is then transitioned to a shop screen where they can use the score they earned from the last game. The shop screen offers power ups including a larger laser size, triple shot, and an extra life. The larger laser size edits the scale of the bullets shot out of the ship. The triple shot power up shoots an extra two bullets towards the left and right diagonals of the front of the ship. These two power ups can also be used together to shoot three all larger sized bullets at once. The shop.swift file checks to see if the player has enough





score once a button is pressed, if so it changes a variable for the corresponding power up to true. Since the variable has now been changed, the Game Scene goes down condition paths for the

individual power ups to apply them to future games. For the UI of the shop page, it was predominantly done with a combination of SKSpriteNodes and SKLabelNodes. The buttons are SKSpriteNodes which have been configured with names. So, the touchesBegan function checks which function was hit, if the conditions are met, and changes the corresponding label text of the button.

Some features still remain to be completed, these included the scoreboard to collect and sort the user's score history. To adjust our plans to the issues that we had not expected by working with SpriteKit for the first time, we decided to focus on our core gameplay. We wanted to develop the features that the player would actually be using rather than spending time map building new levels with lacking features. There was also an attempt to create a tabs bar at the bottom of the window for selecting between play mode, score board, etc. We got trouble while setting up the tabs features since we certainly need to learn more about handling clicking events to switch and expand to different tabs smoothly, and how to use SpriteKit instead of SwiftUi to render this new component into our existing app without crashing. So we moved away from a tabular view and created multiple game scenes which we could use SKTransitions to swap between.

This project was a great learning experience of how project plans can consistently change in slight ways and that planning features is just a starting point. Having an idea of features you want to implement is a great start, but it's also important to think about the best ways to combine work towards the end. Swift is a versatile language with many libraries, so something like creating UI for an application can be done in many different ways. Deciding whether to approach game movement using physics vectors or by smoothly animating the sprites from one CGPoint to another is something we did not think about at the beginning of the project. During our

development of our application it was easier to plan out the specifics of such cases as we began to work on each feature. Our initial plans of using SwiftUI similar to what we had done previously in class was quickly changed into SKScenes with Sprite Kit. However, by continuing to discuss what we were working on with each other we were able to always combine our work and end up with our final product.