Stellar Reaction

A Game by:
Leon Frickensmith
Dylan Schlaht
Henry Lee
Shaan Iqbal
Yuto Otaguro
Alexander McArthur
Andrew Borg
Sahil Suhag
Michael Wojciak

UIUC CS-428 Final Project Documentation

Table of Contents

Project Description

Stellar Reaction is a real-time, multiplayer, space combat video game. Gameplay consists of four teams of Human or AI players controlling space ships, which are used to attack enemy ships, and to capture objective points to earn money.

Modular Ships

Ships consist entirely of modules which provide all the functionality to a ship. More modules can be bought at a store for some amount of money, but there is a limit to the number of modules that can be put on a ship, and on where they can be placed. Individual modules control all of the following: Camera Zooming, Cloaking, Firing Weapons, Energy Production, Ship Propulsion, and Teleportation.

Combat

Combat features ships firing at one another, and using their environment to their advantage. Damage is dealt to a ship by reducing the health of the ships modules. When modules have less than 25% health, they no longer provide whatever functionality they used to. In addition, there are hazards around the different maps that can be used to one’s advantage. A player can control which weapons fire at a given time by toggling their weapon control groups to on or off.

Hazards

Black Holes pull objects toward it that come too close. When an object approaches the center of the Black Hole, damage is dealt to the object, potentially destroying a ship. Ships can escape a Black Hole with the right technique and tools however. Asteroids are another hazard that players should be wary of. They have a large amount of mass, and will damage a player's ship if they run into it. Dense asteroid fields can provide a route of escape for a losing ship, or a graveyard for careless pilots.

Weapons

There are four classes of weapons available to the player. Lasers, Projectiles, Missiles, and Grappling Hooks. Lasers consume lots of energy, but have the advantage of being good at medium range combat due to their time of flight being nearly instant. They are good for damaging specific modules of the enemy ship. Projectile weapons can be tricky to aim, but deal large amounts of damage, making them ideal for close range combat. Missiles are designed for long range combat. They target the closest object to the player's cursor when fired. They will track the target until they hit it or run out of fuel and explode. A player can used quick turning and obstacles to avoid a Missile. Grappling Hooks pull the target ship towards the player's ship, enabling a player to catch a fleeing opponent.

The Process

The team followed the XP process by making use of pair programming, iterative development, and refactoring, among others, using Slack for communication, and GitHub for source code control.

The code management system we used was fairly simple. For each iteration, each pair would create a branch that they would use to work on their user story. Once a pair felt they were finished with their user story we would merge the code from their branch into the master branch. The code that was in the master branch was tagged at the end of each iteration. The main Slack channel was used for team announcements, and any general discussion. Individual members used it to contact each other to set up meetings for pair programming work.

At the beginning of the project, the team had a large meeting to brainstorm ideas that we wanted to add to the game. From these ideas we formed user stories for the first few iterations. Iterations lasted for about two weeks, and the team planned out the user stories in advance of the next iteration. Each member would then pick the stories they were interested in for a given iteration. In the second half of the semester, the team began to decide the user stories on the first Sunday of the iteration. This change was a necessary one because the team's developmental goals would shift based on the progress that was made during the previous iteration. One challenge of iterative development was the somewhat short iterations. Sometimes, it is difficult to estimate the amount of time that is needed to implement a certain story. This, compounded with the shorter iteration durations, can make it difficult for a pair to finish implementing their story while also writing maintainable code.

The team worked in pairs and trios, and tried to make sure that each team member was paired with a new person each iteration. This allowed us to enhance our team communication as well as our collective code ownership. Once the pairs were determined, it was up to the pairs to decide when they wanted to meet and work on the project. Each pair kept track of their meeting times and what they had accomplished on the Wiki page. One issue of pair programming is that you need to meet physically with another team member. Finding common meeting times to work on the user story and meeting for an appropriate amount of time sometimes proved to be a challenge.

For each iteration, each pair was assigned to refactor the previous iteration's code. Furthermore, we ensured that the code that was being refactored had been worked on by one member of the pair in the previous iteration. This way, the other person in the pair was exposed to new code they had not seen, and at the same time, the pair could work together to improve the code quality. This also proved to be a challenge for the person having to familiarize themselves with code they had not seen before.

The team did not strictly follow test-driven development as described by XP. The tests for each user story were usually written after implementation to ensure the desired functionality was present. The main issues with testing came with understanding and working with the Google Test framework. Many of the team members did not have experience using the framework before, so we had to familiarize ourselves with it to be able to setup automated tests for the project. Furthermore, there were a few instances where changes that one pair made to

their code caused the tests of another pair to fail. This almost always ended up being a minor change but caused problems when merging the code at the end of the iteration.

At the end of each iteration, a meeting was held with the team's TA in which each pair would give a demo of their user story while also explain their implementation at a high level. Each pair would also quickly go over the unit tests for their user story. This was perhaps the most important part of the process as the TA provided us with valuable feedback as to how we could improve our project for the final release.

User Stories

The user stories focused on adding new elements and features to the game to enhance the gameplay experience.

Iteration 1

- Write Proposal - Mainly consisted of getting an idea and setting realistic targets to achieve over the course of six iterations. The process of idea gathering was fairly simple as we picked up the idea of developing a game that was initially created by Leon, which helped us greatly as we already had a lot of the framework set up for us.

- Form Team - This required everyone to decide on the project they wanted to work on from the list of projects showcased during lecture. After the formation of the final team for the project, a formal meeting with all group members was arranged where current state of the project was discussed and future goals for the project were set.

- Prepare for Iteration 1 - Everyone was introduced to the framework and existing code of the project, and given time to download the required software. This included deciding on the user stories for the upcoming weeks.

Iteration 2

- Module Controls - Groupings made for weapons so they do not all activate per mouse click. Can enable/disable certain weapons.

- Stealth Module - Implementation of a stealth module that would activate on a key press for five seconds and make the ship transparent to the enemy teams. While in stealth, the ship’s energy is also continually drained.

- Hazard Field - Implementation of asteroids, so as to improve the background and give more of a space-like feel to the game, and also to add obstacles to the user ships.

- Missile Weapon Module - Implementation of a new missile launcher module as a weapon for the ship. Missiles have significantly longer range than other weapons in the game and also have the ability to lock on.

Iteration 3

- Death Mechanics - When the main reactor of a ship is destroyed, it will respawn after three seconds. After respawning, the ship is held in place for 1 more second before giving the controls back to the player or AI. All modules are healed upon respawn.

- Game Mode - Objectives added to the game with implementation of victory/losing conditions. Included the display of player score in HUD.

- Automatically Loaded Blueprints - `BlueprintLoader` now finds all files under a particular subdirectory to load, instead of relying on a file roster. General refactor of `BlueprintLoader` to contain less code.

- Weapon Types - Implementation of two different types of weapons: area of effect explosive weapons and piercing weapons.

- Mini Map - Implementation of a radar that detects nearby enemies, allies, capture stations, and hazards in a defined scope displayed on the bottom-right of the screen.

Iteration 4

- Shield Modules - Implementation of a shield that deflects the projectiles aimed at it. The shield can be manually turned on and off. The shield drains energy with each hit and turns off when the user is out of energy.

- AI players - AI Players to control ships to fight other players, and each other. Initial implementation had a working AI but the need for refactoring was evident.

- Black Holes Hazard - Implementation of a dangerous entity that pulls objects towards it and potentially does damage to them.

- Teleportation Module - Implementation of a module that would teleport the user ship in the direction of the mouse. There is a maximum distance that the ship can teleport and the ship cannot teleport inside of another ship.

- Shotgun Weapon - Weapon added that would fire a spray of projectiles while satisfying weapon inaccuracy. Initial implementation didn't fully satisfy the inaccuracy and hence was one of the issues that required further inspection.

- Grapple Weapon - Weapon implementation on the ship that would harpoon the target and pull it towards the ship. Because the combat takes place in space, both ships are pulled toward each other.

Iteration 5

- Fix issues on the GitHub issues page - Fix many of the small things described on the issues page so as to refine the game and refactor some old components that required it.

- Map Improvement - Created a functional map with asteroid fields, black holes, planets, spawn points for each team, map boundaries, and capture points. This also included creating graphics for black holes, planets, capture points, and spawn points.

- Dynamic Art System - Background art is loaded into the game from a level configuration file, and can appear as far away objects by simulating the effects of parallax.

- No Friendly Fire - Members of the same team cannot damage each other.

- Display money on the HUD - Show and continuously update the player's money on their HUD.

- Scoreboard - Shows the name, score, and money of each player in the game upon key press.

Iteration 6

- Fix Issues on GitHub Issue Page - Same as Iteration 5

- Further improve the AI - Improvement to the original AI implementation by making the AI ship choose on random whether to attack a ship or go to a capture point. Refactoring the original AI problems of the ship getting stuck and attacking an enemy ship without deviating from a decided path.

- Polish UI - Improve usability and visuals.

- Refactor & Documentation - work on final documentation, remove unused code, add comments to public functions, etc.

Architecture & Design

Stellar Reaction has two large components that every other part of the system is related to. The first component is the `Game` class. All objects are contained within `Game`, and `Game` provides a global access point to many utilities needed at a global level, such as Textures, Sounds, Animations, IO-Messaging, Networking, and the game's Window, as well as holding the game's GUI and a single `Universe`.

The `Universe` is the other major component, as it contains all objects that the player sees when actually playing the game. The `Universe` holds blueprints, which are loaded when the game is launched, as well as `Controllers`, which are used to sync player commands between networked games. When a game launches, a new `Universe` is created, with the old one being destroyed.

The biggest thing the `Universe` contains are the `Chunks` that represent ships in the game and the `Modules` that represent the weapons and add-ons to ships. `Chunks` act as containers for `Modules` which provide all the functionality to a ship. `Chunks` are used mainly for representing a ship in the game, however hazards such as black holes and asteroids are also derived from it.

All game objects in the game have a blueprint which allows convenient creation of multiple large objects that have many member variables, as well as multiple instances of these objects with different values. It also allows the modification of data without having to recompile the program, as they are loaded from JSON files via `BlueprintLoader`. For example, a weapon blueprint contains the amount of damage, energy consumption, laser color, and sprite for the weapon. These can be modified, and a player only has to relaunch the game to see the changes.

Framework

The entire project was done in Visual Studio 2013 C++ Community Edition, a free IDE with a compiler that supports C++11 features. This had a large influence on the design of the project, since C++ puts the burden of memory management on the developer. Visual Studio however provides many features to make this easier, as well as C++11's smart pointers. Visual Studio has an advanced, built-in debugger, which allows the viewing of the programs variables at runtime, and providing detailed information in the event of a crash. Smart pointers remove the need to call delete on anything, helping reduce memory leaks, and general pointer mishaps.

The game engine uses Box2D for its physics engine, SFML for graphics, and TGUI to help with GUI aspects. Box2D and SFML behave very differently, which required writing wrappers such as `BodyComponent`, `FixtureComponent`, and `Camera`, to prevent having to switch between coordinate systems so frequently. We also used Google Test framework which provided an easy way for us to write unit tests.

Damage Sequence Diagram

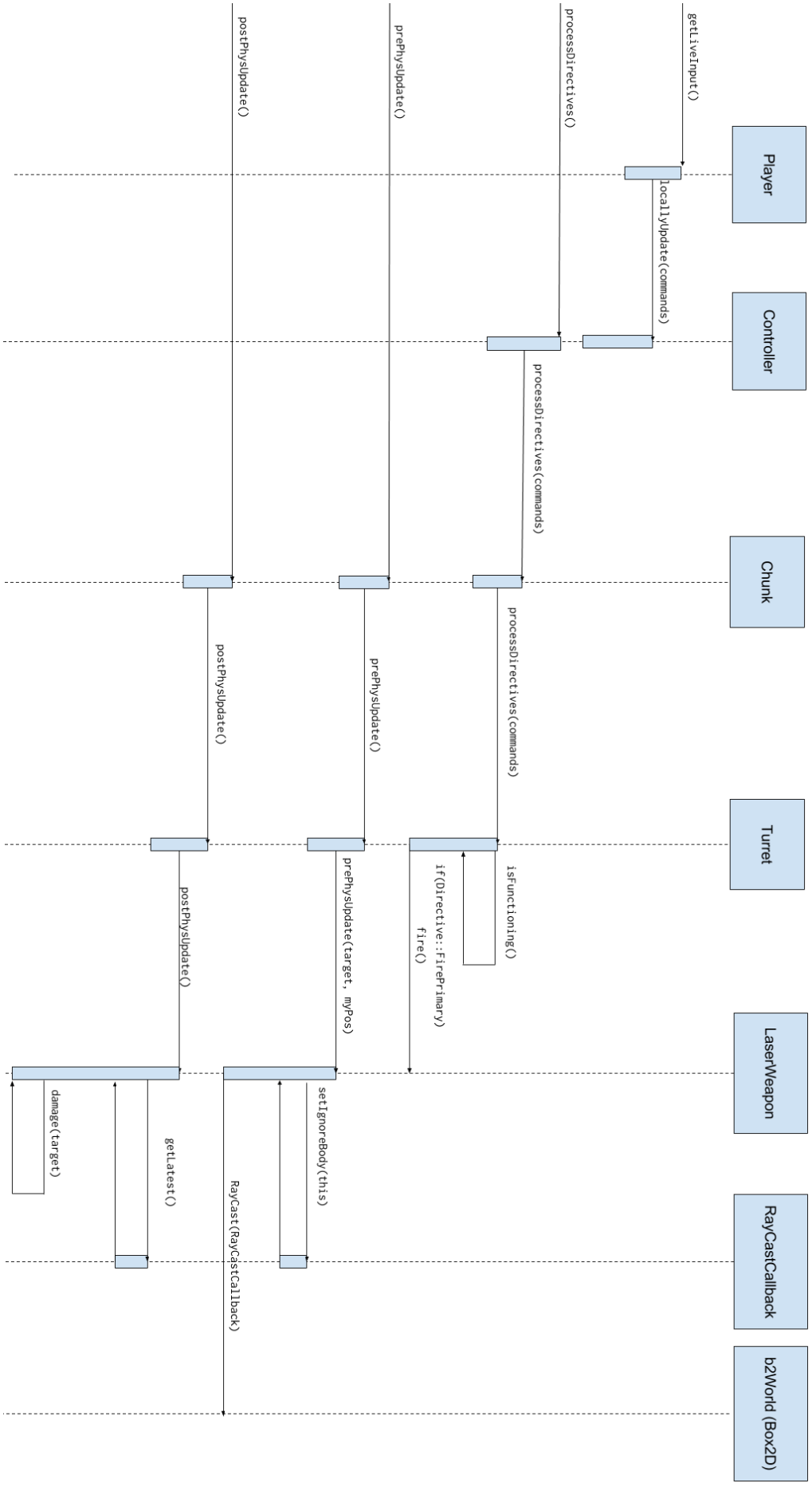

Figure: 1

New Game Sequence Diagram

To Next Page

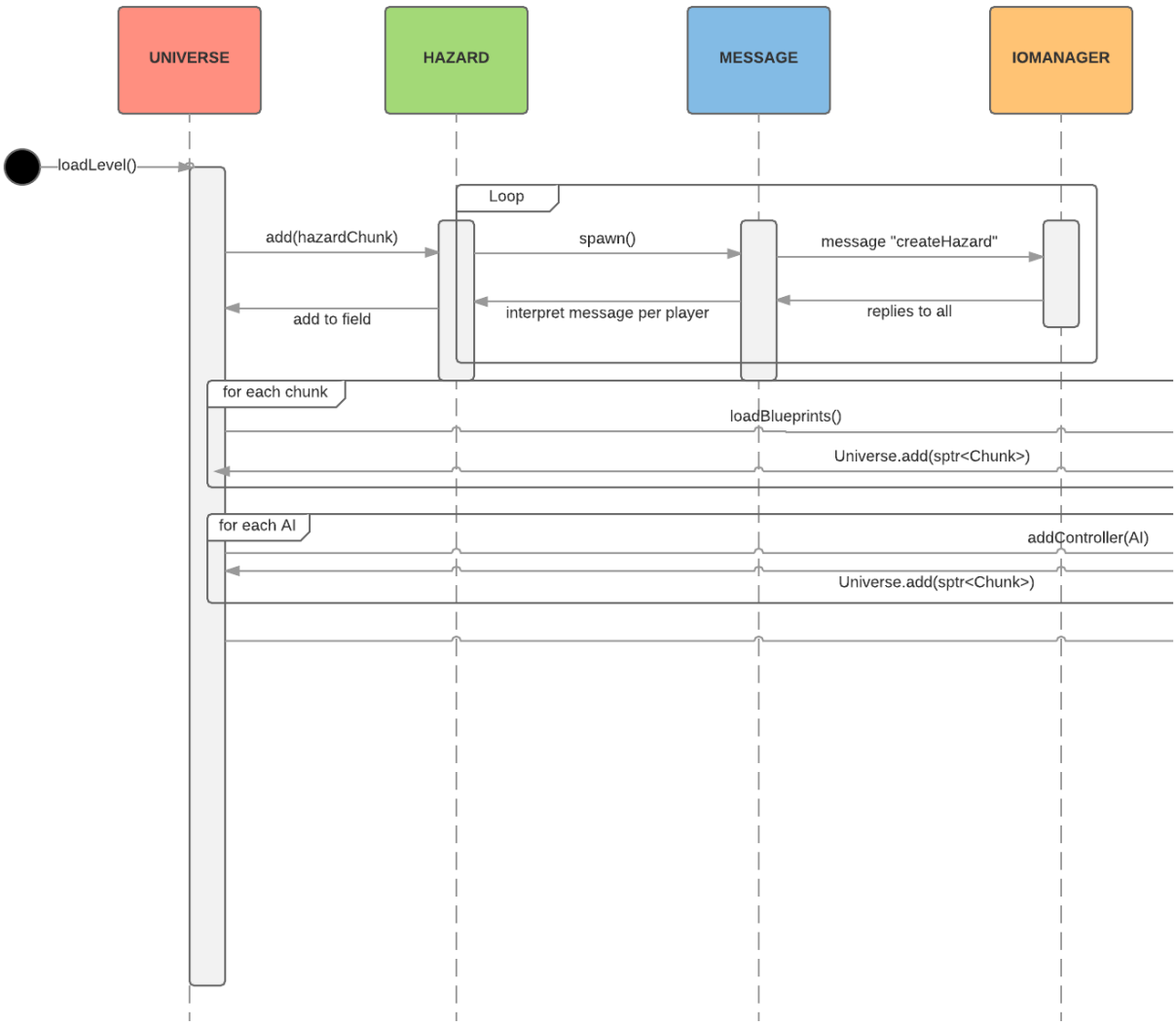

Figure: 2.1

To Previous Page

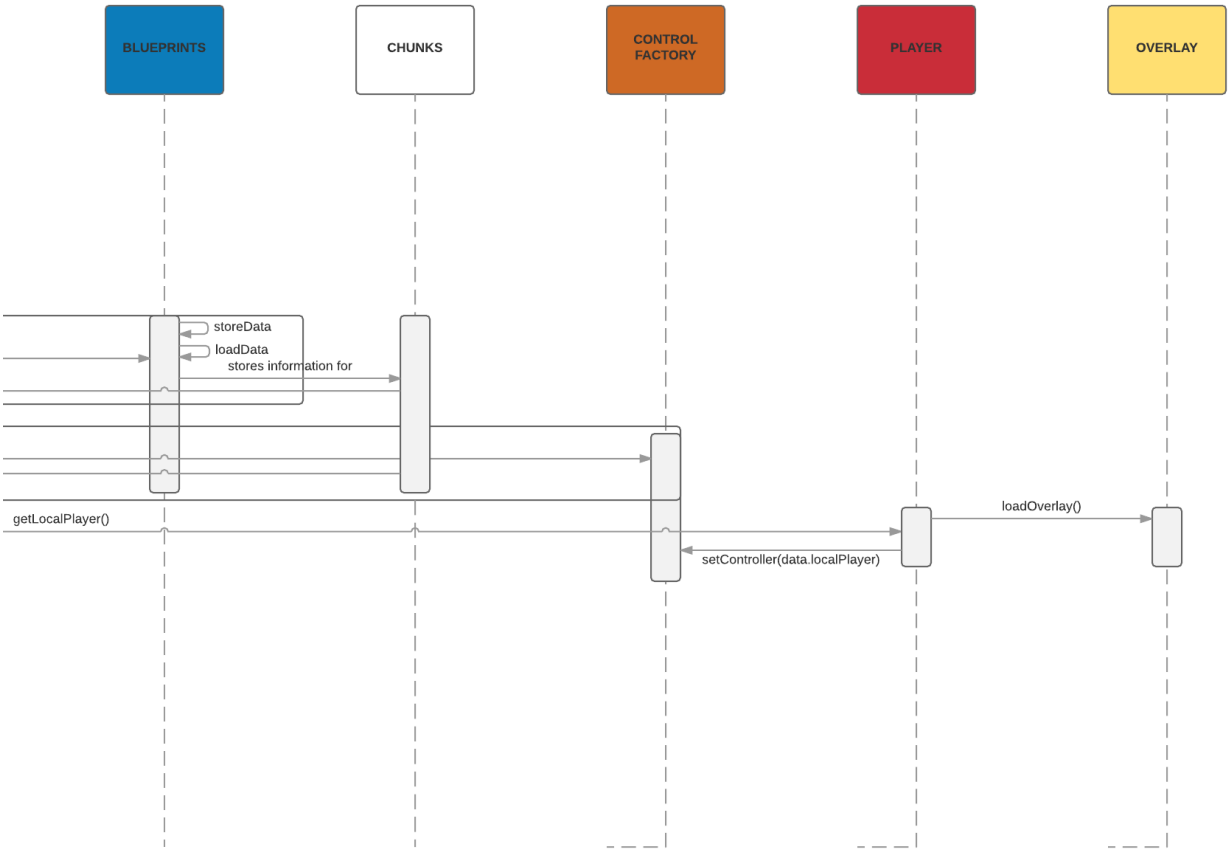

Figure: 2.2

Class Diagram

To Next Page

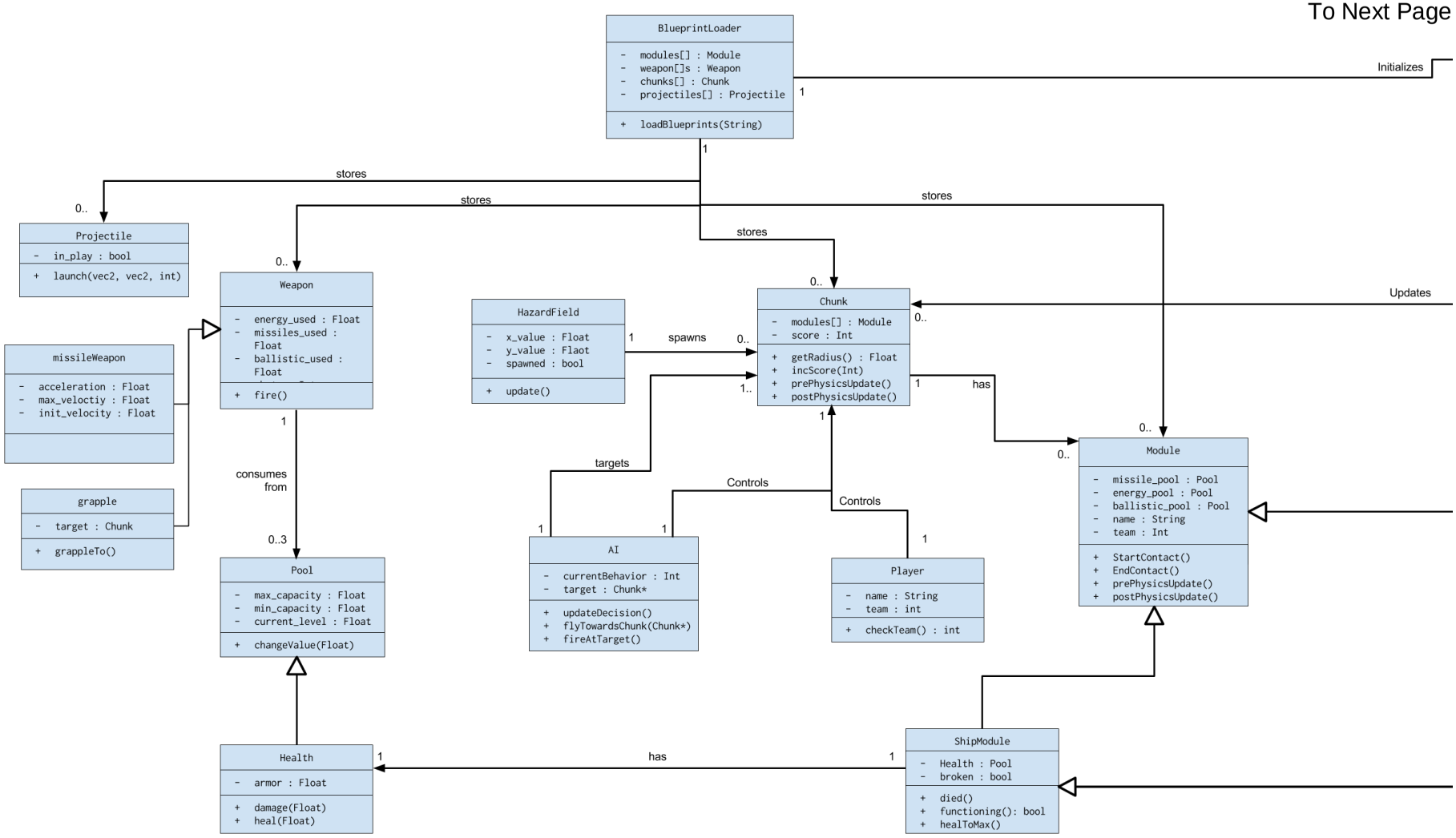

Figure: 3.1

To Previous Page

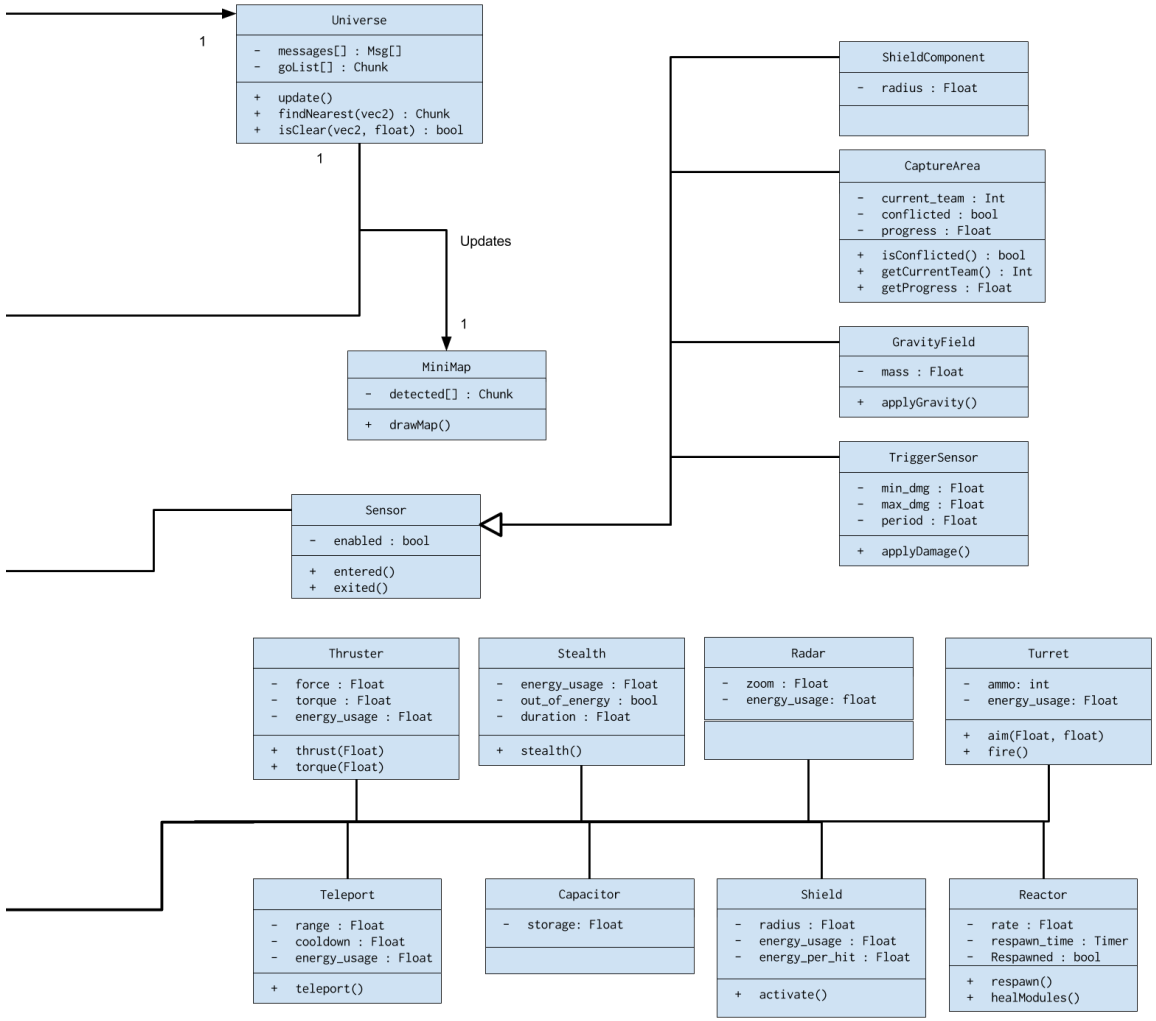

Figure: 3.2

Reflections and Lessons Learned

From working on this project, I learned the value of having documentation and unit testing. Stellar Reaction began with a large chunk of legacy code. If we didn't happen to have the creator of that code in our team, we would have been completely lost. Now, I feel if someone else were to take over that wouldn't be the case.
- Alex McArthur

I started this project before I knew about this class as a learning experience and did not put as much effort into documentation as would have been ideal. This made it more difficult for the other contributors to work, as questions would pop up that only I could answer. I did however put a lot of effort into design which paid off enormously. Making new types of objects was very easy due to a good set of Base objects implementing most complex functionality, and loading data into the game from file was facilitated by an easy set of function calls. Auto-registered classes made the overhead of creating and drawing graphics objects almost zero. Despite putting a lot of effort into the design, this was the first time I attempted to make a multiplayer game, which resulted in a non-ideal networking structure, which could be improved upon. Another thing I learned from this project was to limit project scope early on. Because I did not have a concrete goal set on gameplay, it caused me to write code that was too general, and ultimately did not contribute as much to the final project as it could have. This common error is described in a famous article titled, "Write Games, Not Engines". If only I would have found that a bit sooner. I am excited for the project, because it becomes more of a game every day, and the list of necessary features grows smaller. Besides even better AI, a multiplayer browser, some art, and few modules, the game doesn't need much more to be completed.
- Leon Frickensmith

This project has been a great learning experience for me. Not only did I get to work on developing a game which was something that I aspired to do since I was a kid, but I also got to polish my thinking skills along with learning how to use GitHub and Visual Studio. It helped me understand how to work in a large group to achieve objectives and helped me realize the importance of coherence within a team, as that can be the make or break for any project. We worked fairly well as a team and stuck by our deadlines and were also lucky to have Leon to guide us through the project.
- Sahil Suhag

I've learned that understanding a pre-existing code base doesn't necessarily have to begin with understanding the underlying structure first. Instead of trying to grasp all the systems and which feature they occupy, it is more efficient and reasonable to be dedicated on one system. It is a difference between breadth and depth, and each person is likely to have their own preferences. For me, it was depth because working on one system was bound to force me to understand a different one. On personal note, I feel like I should have looked into more testing functions. A lot of things I have worked on in this project had manual testing as a more

adequate and reasonable approach (UI-related and level loading) but since we learned about GUI testing in class, I could have looked more into it. Unfortunately the testing framework we employed did not have GUI testing.
- Yuto Otaguro

The main challenge I had working on this project was figuring out how all the systems in the game worked, and how they interacted with each other. When implementing user stories, a lot of the time was actually spent figuring out how existing things worked before being able to add the desired functionality. This is where I think pair programming helped a lot. Having someone else being able to go through the code with you is much easier than trying to figure out everything alone. With the added documentation the project now has, I feel that this process of understanding the code base will be much easier for someone new to the project.
- Shaan Iqbal

I, personally, enjoyed working on this semester long project quite a bit. It was, however, challenging to jump into such a large code base at the beginning of the project but I grew quite familiar with it faster than I had expected. The group project aspect of this class has introduced me to all the complexities and problems of working in large groups that I may encounter in the future. Problems like these are inevitable and this experience will not be forgotten. Furthermore, I found that learning the strengths and weaknesses of each team member has brought out the importance of collaboration and finding the unique roles that each individual may fit into.
- Dylan Schlaht

One thing that I learned from this project and the process we went through was how much documentation mattered in a fairly large project like this. Often times in iterations what was most challenging was figuring out how to add on to existing code and determining where the new functionality belongs. Most of the time the issues would be resolved through asking Leon, who was the project leader and also had the most knowledge about the code base. Although this was one method of solving the problem, I think that it would have been much easier if we had better documentation to begin with. If the documentation was sufficient, then there would be less time spent on asking questions and/or attempting to learn the code base itself. That said, there was one thing that I appreciated in this project and that was the modularity within the code. It was easy to implement and derive from some classes and that made some of the user stories a lot simpler than they would have been otherwise.
- Henry Lee

This project was a lot of fun to work as well as a great learning experience. Before this class I had only one other experience working with an existing codebase with many people contributing to it. I had not used GitHub very often before and was used to other types of version control like subversion so it was useful to see how they differed. The biggest thing that I learned, however, was the process that goes into creating an actual game. I had always played around with the idea of game development but never completed anything of note. Working on such a

large game with many people really made me realize everything that goes into the development and how important documentation is.
- Michael Wojciak

From this project, I learned much about building off of a piece of software that wasn't fully functional and adding functionality as it was being built. Having tests and documentation completed as functionality is added is a godsend, since understanding large codebases can be a daunting task. After having worked to document the large amount of code that was undocumented, I realized that the work could have, and should have, been done by the original authors as it allows them to convey not only the function of the code but as well as the intent of the code. Furthermore, having an expert of a project available to assist in understanding of the code base greatly eases the acclimation process and aids in getting new programmers on the project up to speed.
- Andrew Borg