KARELIA UNIVERSITY OF APPLIED SCIENCES
Business Information Technology

Pietu Hyvärinen

FROM A GAME JAM GAME TO A FULL RELEASE

Thesis
September 2018

THESIS

September 2018

Bachelor of Business IT

Author

Pietu Hyvärinen

Title

From a Game Jam Game to a Full Release

Abstract

The goals of this thesis were to develop a new game based on a game developed at the Global Game Jam of 2018, describe how the development processes differ between these two games and examine how the final games are different. The game was developed using the Unity game engine.

In the design process, a method of looking through lenses was used to give more perspective on the design. The implementation process was conducted using good programming practices and in the polishing process, an extra layer of polish was applied. Finally, the game was published to Itch.io.

The new game was visually as well as gameplay, sound and code-wise superior to the game jam game, indicating that the development processes of the new game were more structured and thought-out. The main factors differentiating the development processes and the quality of these two games were, the time available for development, having more expertise and the possibility to use money for the assets of the new game.

Language

English

Pages 64

Keywords

game jam, game development, game design, programming

CONTENT

1 INTRODUCTION 4

2 TOOLS 5

2.1 Game Engine 5
2.2 Version Control 6
2.3 IDE 7
2.4 Project Management 7
2.5 Design tools 8

3 DESIGN PROCESS 8

3.1 Scope 10
3.2 Looking through the lenses 12
3.3 Designed features 19

4 IMPLEMENTATION PROCESS 24

4.1 Writing Clean Code 25
4.2 Patterns 29
4.3 Main Feature Implementation 33

5 POLISHING PROCESS 45

5.1 Sounds 47
5.2 Juiciness 47
5.3 Optimization 49

6 PUBLISHING PROCESS 50

6.1 Steam 51
6.2 Itch.io 52
6.3 Publishing to Itch.io 53

7 RESULTS 56

7.1 Development processes 57
7.2 Visuals 58
7.3 Features 59

8 CONCLUSION 60

REFERENCES 63

1 Introduction

Global Game Jam is an annual event taking place in January. People gather to jam sites that are placed all over the world and to make games together. A central theme is given to all the participants that their game must be built upon. Themes are not usually very restrictive, and they often consist of only one word, which can be interpreted in many ways. The 2018's theme was transmission. After the theme is announced, participants brainstorm together for game ideas that could be built based on the theme. When the brainstorming has concluded, the participants can pitch their ideas to others and then people form teams based on those ideas. The idea behind Global Game Jam is to network with people and have fun. If you want to, you can make a game by yourself, but this is not the purpose of the Global Game Jam. Teams have 48 hours to make and release the game to the Global Game Jam website.

The game that we built as a team of six developers at Global Game Jam 2018 is called *Tinfoil Deflector*. Tinfoil Deflector is a small game where the player moves a circular object around a planet and tries to absorb or deflect incoming hostile beams by hitting them with the controlled object. The theme transmission is represented by sound files that each of the beam plays on their way towards the planet (Figure 1).

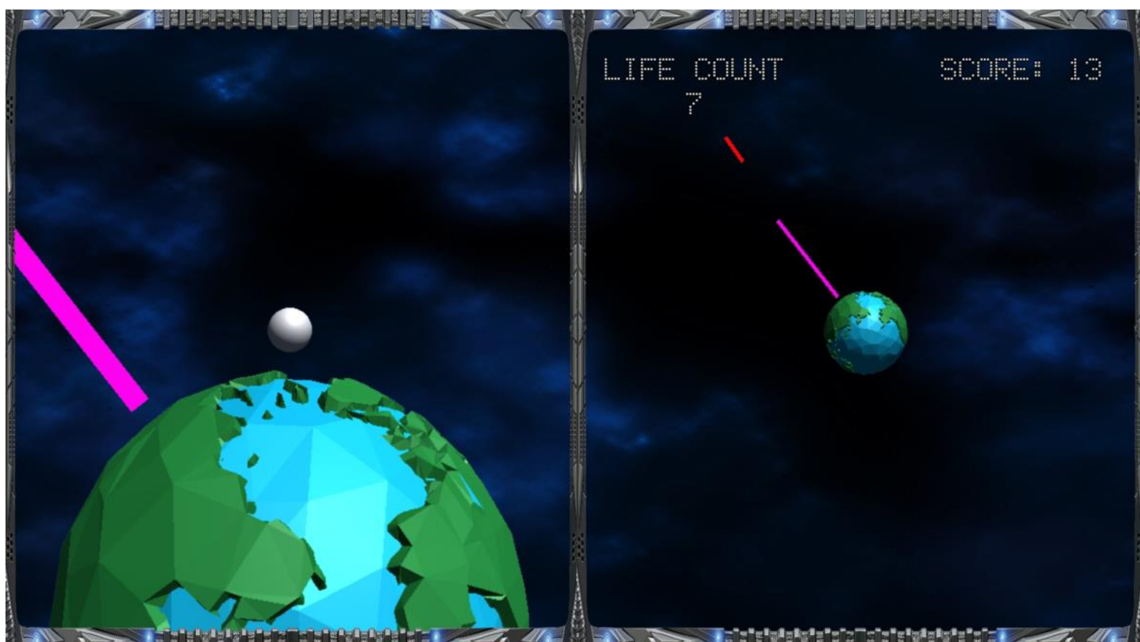

Figure 1. Tinfoil Deflector's game scene

This thesis goes through the whole process of creating a game that is based on the game made at the Global Game Jam of 2018. The game that is developed based on Tinfoil Deflector is called *You Are the Light*. You Are the Light retains the core of Tinfoil Deflector while adding more content and polishing in the process.

The purpose of this thesis is to demonstrate and analyze the process of developing a new game based on an already existing game. The focus is on the development of the new game, but there are some general descriptions about the game jam game development processes. The two processes of game development are compared with each other while considering their different purposes, resources and the theory of game development involved. Also, the differences between the visuals, gameplay and features of the two completed games are surveyed.

In chapter 2 *Tools*, the tools used in this project are described. In chapter 3 *Design process,* the design process and the use of a method of “looking through lenses” to refine the design, are explained. In chapter 4 *Implementation Process,* a few programming patterns and good programming practices and examples of the code written for You Are the Light are examined. In chapter 5 *Polishing process,* there is a description of what polishing a game means and a few examples of what was polished in You Are the Light are given. In chapter 6 *Publishing process*, two different publishing platforms are examined and the process of publishing to a publishing platform Itch.io is described.

2 Tools

2.1 Game Engine

Unity was chosen as the game engine for this project because I have used it for quite some time and Unity is powerful, Unity Asset Store has a lot of high-quality assets and above all, Unity has a very large community of developers, so it is easier to find answers to questions from different discussion boards. Unity can be used to make 2D and 3D games and it supports publishing to multiple platforms. Unity engine itself is programmed in C++, but it has a wrapper allowing the users to write their scripts in C#. Unity Technologies released version 2018.1

of their game engine which has a new set of powerful features like entity component system and Shader Graph, but these features are not relevant for this project, so the version of the engine used in this project is the previous iteration of the engine, Unity 2017.4. Unity 2017.4 is flagged as LTS which stands for Long Term Support, and as such Unity Technologies will support this version of Unity for two years to come.

Another game engine that could have been chosen is Unreal Engine. Unreal Engine's scripting language C++ is considered a bit harder to grasp than C#. Unreal Engine has a visual scripting tool called blueprints. Blueprints is a node-based visual scripting tool where you make code nodes and drag lines from one node to another forming connections between the nodes. Blueprint system is great for artists who do not know how to write code, because with blueprints they can contribute to the making of functionality with easy-to-use visual scripts. Unreal engine is very popular, but the community is not as big as Unity's and that makes it more challenging to get answers to problems easily. Many large companies use Unreal Engine as their game engine, but the few times I have tried it I have felt that it is too bloated for my use cases and is more suitable for more ambitious projects and bigger teams.

2.2 Version Control

Unity has its own version control system (VCS) called Unity Collaborate, which is made for Unity projects. You have to connect to the service through the editor's services tab and then you can upload your project files with a single click. Collaborate is not as robust as *Git*, but it is easy to use, is made for Unity projects and has enough space available for my needs, so it was chosen as the VCS for this project.

Git is the most popular version control systems available. It takes some time to become familiar with the workflow and to understand all the commands available, but it is worth it. You can easily return to a past state of the project. You can upload your files to an online repository via different services like GitHub or GitLab, so they are safe in the cloud. You can use Git straight from the command line or through different graphical user interfaces like GitKraken or SourceTree. I

tried to use Git and GitLab as my version control tool and service for the prototype version of this project, but there was some trouble uploading large graphical files to a repository. Sometimes the push to a repository fails to go though no matter how many times tried if the graphic files are too big in size. This made me choose the Unity Collaborate.

2.3 IDE

Rider is an Integrated Development Environment (IDE) developed by JetBrains. Rider can be used for anything that is written in .NET Framework, .NET Core or anything that is Mono based. Rider is fast compared to Visual Studio and it is cross platform (Taylor 2018). Rider has a lot of code completion features and refactoring tools that quicken the workflow considerably. Rider has a monthly fee unless you are a student. Rider was chosen as the IDE for this project because it does more than the competitors and a free license is offered for students. Ultimately it doesn't matter what text editor or IDE you use, but you should find the one that supports your workflow the best.

Visual Studio community edition is free to use. Visual Studio now ships with Unity installation. The IDE is powerful and full of great features, but Rider has some crucial features, such as suggestions for better code structure and changing to the suggested structure with a few simple commands.

Visual Studio Code is a lightweight text editor and it is lightning fast. Being lightweight it lacks features that come with full-fledged IDEs like Visual Studio and Rider, but you can download extensions to the editor through the marketplace. The extensions can bring the text editor on par with basic IDE functionality, but it cannot attain the same level of features as Rider.

2.4 Project Management

Trello was chosen as the project management tool. Trello is a simplistic application you can use on desktop, browser and phone. With Trello you can manage your tasks with ease. You make a board for your project, make relevant lists for the tasks or “cards” as they are called and then fill the lists with the cards. Common practice is to make a list of all the tasks called “backlog” and then make a “doing” and “done” lists. You move the tasks you are working on to the doing list and when they are done, you move them to the done list.

Jira is another project management software that I have used for school projects. Jira is supposed to be used for agile development projects and especially for Scrum. While Jira is a good tool, it is far too complex for a project with a single developer. Jira shines when the project is large, complex and there is a customer involved.

2.5 Design tools

Draw.io was chosen as the diagramming tool for this project. Draw.io runs in the browser so it is easy to access it from anywhere you want. You can save your diagrams straight to Google Drive if wanted. Draw.io is also simple enough and easy to use for small projects like this one.

Another diagramming software I have used is Visual Paradigm. Visual Paradigm allows the creation of more complex diagrams compared to Draw.io, but as not that many diagrams were needed for this project Draw.io was the more suitable contender.

3 Design Process

Designing a game is an iterative process. A designer cannot possibly think about all the outcomes of his design before taking it to practice. Design may change throughout the development process if the features designed feel out of place or outright do not work. It is hard to evaluate if a designed feature works or not when

you are not able to test it. Prototyping is how different designs can be tested before going into full development.

Tinfoil Deflector was designed as a group. We collectively brainstormed the ideas for the game and designed the features keeping the small scope in mind. The design was being restricted and influenced by the given theme, Transmission. Our group used most of the first day of the game jam in the design, but we did not have time to prototype the features. There was no comprehensive documentation work done on the design, but we used a Trello board for task management.

I made a few prototypes of You Are the Light in the spring of 2018 before working on the final product. These prototypes flew in the trashcan, but I gathered valuable experience from these prototypes and found out what worked and what did not. Outside of prototyping, the design still changed during the development process owing to time constraints so that the game could be launched in a reasonable time span.

My design process has different phases. First, one has to come up with an idea, then take notes of the idea, prototype the idea, test the prototype and then make changes to the design if needed. This is the process for features that can be prototyped at this time. (Figure 2)

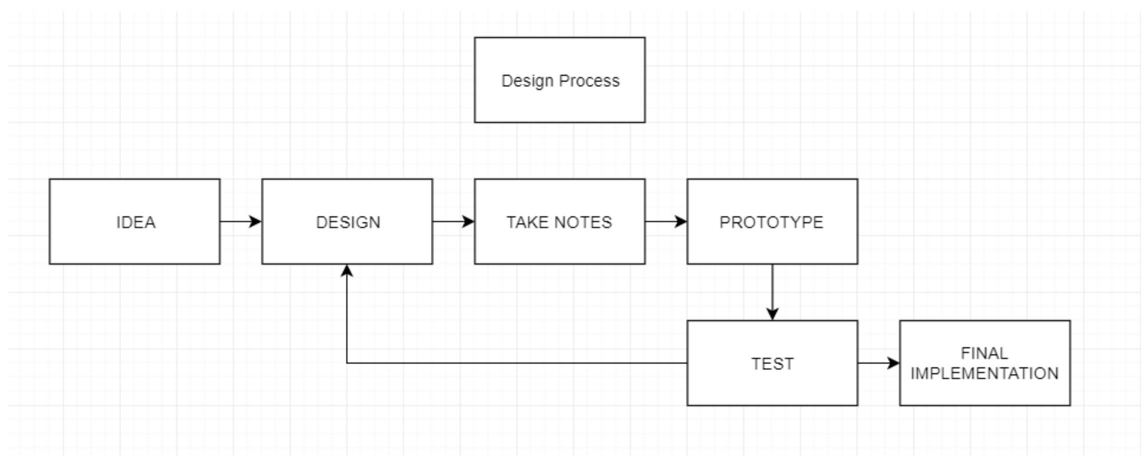

Figure 2. Design flow if the feature can be prototyped at the time.

If the designed feature cannot be prototyped at this stage because the feature would depend on other features that are not yet implemented, the design process just goes from an idea to design and taking notes. (Figure 3)

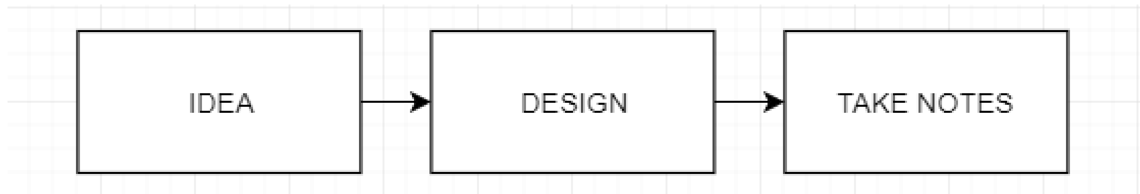

Figure 3. Design flow if the feature cannot be prototyped at the time.

The ideas that I came up with for this project were heavily influenced by the "looking through the lenses" methodology that is explained later in this chapter. I often scribble the idea for a feature on a piece of paper first when it emerges and then start to think more about the design and put the more refined thoughts onto a Trello card as a checklist (Figure 4). Trello makes it easy to track what you have to do and in what order if the cards are organized correctly.

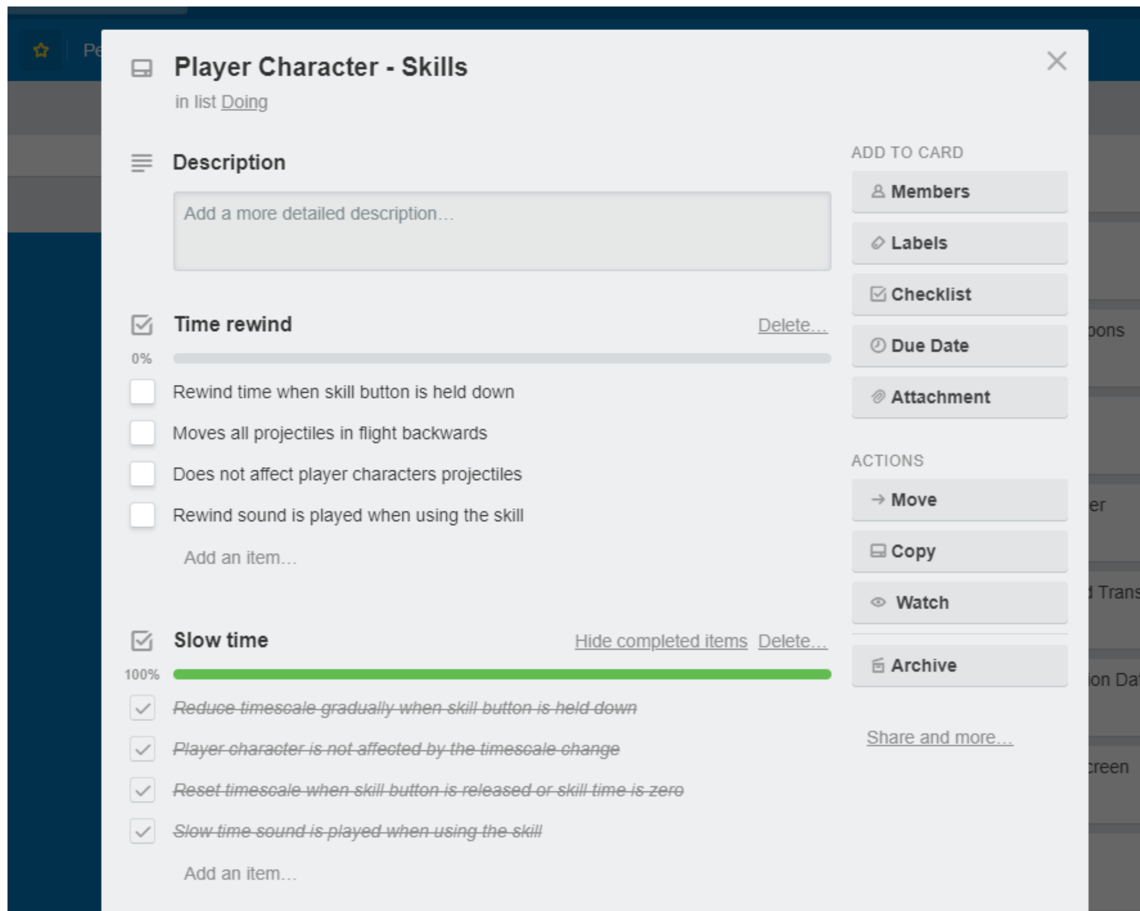

Figure 4. Trello card with checklists for each of the skills.

3.1 Scope

Scope is the result of different resources and needs the project has. To define a scope, a developer needs to know what he wants to do, how much money, time, knowledge and people he has for the project and evaluate the goals based on these resources available. Scope is not set in stone, it can change over time, for example, if you get more money during the development you can do more than what the original scope allowed. There are different ways to define a project's scope and one of them is to divide your game into elements. There are four elements that need to be considered: mechanics, story, aesthetics and technology. Now you need to think about your game and define, for example, what kind of mechanics does the game need? Do the same to story, aesthetics and technology. List all the features. After you have considered these different categories you can evaluate them. You might want to use three different grades for evaluation: low, medium and high.

Defining the scope of the original product, Tinfoil Deflector, was quite simple. We had thematical and time restrictions coming from outside and no money. We wanted to make the player move around a planet, spawn beams from random positions around the game field to move towards the planet, the beams have different effects (a number of these effects can be reduced or added based on time available) and we had to find different free transmission sounds from the web. As our resources we had 48 hours of time, 6 developers of varying skill level and no money. Originally, we had the idea of a minigame within our game that would be triggered upon certain conditions but that was deleted from our feature list due to the time limitation.

The scope of the thesis game, You Are the Light, was a bit different resource-wise from the game jam game. A total of three months was dedicated to make the game and to write the report based on the making process. The game had a budget of 100 euros and all of the budgeted funds were used. I worked alone, and my skill level at the time could be considered advanced. As the development time I figured that one month is sufficient enough and the other two months were left to writing this report.

Software projects may grow bigger over time, for example, if a customer wants more features to be added when the development has already started. You Are the Light was done by myself so there was no external pressure to add more features, but I still felt the need to add some. When you add more features than

there was in the original design or plan, this is called scope creep. More often than not scope creep forces the project to take a different path that may result in crossing the resources available. To prevent scope creep, it is best to document all the requirements of the project that are possible at the time. Small changes are naturally going to happen and are acceptable but when coming to bigger changes that have a large impact on the project you must evaluate whether the new features and changes are possible resource wise. Lacomb (2017) suggests in an article written that: “Breaking the project into small easily digestible pieces is a good way to keep track of how things are going.” I used this method with my design and can assert its importance. The proper use of tools such as Trello were especially important in itemizing features and tasks throughout the project, which helped keep the scope in check.

3.2 Looking through the lenses

Jesse Schell in his book *The Art of Game Design: A Book of Lenses (2008)* approaches game design by looking at the game from as many different perspectives as possible. He calls these different perspectives *lenses*. These lenses are tools to be used when designing a game. Most of the lenses are not suitable for the scope of this thesis as many of them revolve around the story aspects of a game and are meant for larger productions than this, but some of the lenses that are best suited for this project’s needs were picked. Next, Tinfoil Deflector is viewed through these lenses and evaluated to get a better view of what must be done to ensure a better experience in You Are the Light. This is a means of retrospection central to my thesis by which I evaluate the already made game to improve the new one.

3.2.1 The Lens of Essential Experience

Looking through this lens a developer needs to think about the experience that the game gives to the player. What is the experience a developer wants the player

to have, what is essential to that experience and how can the game capture that essence? A developer might be creating a different kind of game than what he wants to deliver as an experience. If the game does not seem to be delivering the kind of experience intended, the game must change. (Schell 2008, 21.)

While the Tinfoil Deflector is fun to play, we did not think deeply enough about the experience it was supposed to give to the player. There is some tension due to the inevitable loss in Beam Defender, but the action is a bit too slow paced.

You Are the Light is supposed to deliver a fast-paced, exhausting and exhilarating experience. This experience must be enforced to the fullest potential. To attain this player experience, every action in the game must have an impact, pressure must be built on the player and moving objects ought to be fast and the effects ought to be explosive.

3.2.2 The Lens of Fun

Games are usually supposed to be fun to play. Looking through this lens a developer is supposed to find out how to maximize the game's fun factor. What parts of the game are fun and why? What parts need more fun injected into them? The definition of what is fun varies person to person and a developer can't please all of the players. (Schell 2008, 27.)

Tinfoil Deflector is a fun little game, but it is lacking content and player action. The only actions the player can do in Tinfoil Deflector are movement actions and toggling the shield if the player gets it as a beam effect. To increase the fun, the player needs to have more actions at his disposal.

To make the game more fun, in You Are the Light the player competes for a new high score against himself and other players. At a bare minimum, You Are the Light is going to have indication of the last high score the player attained and nice visual effects and indications when the player has topped his own high score. If there is enough time, a leaderboard system will be implemented which lists high scores of all the players who have played the game and this way the player can compete against other players for the top spots of the list.

Shooting and blowing up things is considered fun and these actions are the bread and butter of many games. Usually hitting objects by shooting at them is

something that requires skill, which is also the case in You Are the Light. Shooting must feel powerful so a proper feedback to the player through sounds and visual effects is to be placed.

3.2.3 The Lens of Endogenous Value

Games often have some sort of scoring system or something that accumulates points based on player actions. This lens makes a developer think about the things that give player value. What is the player after in the game? How can a developer make valuable things more valuable? How is the value and player's motivations connected? (Schell 2008, 32.)

In Tinfoil Deflector the player gains one point for each second he survives. This design makes it so that the only way for the player to get a better score is to survive for a longer time. There is no reward for any kind of skillful actions that the player might be able to pull off.

In You Are the Light the player competes for a high score. The player get points for hitting and destroying hostile projectiles and for hitting and destroying a boss. Projectiles move towards the planet, which the player is trying to protect, so the points are accumulated through the process of defending the planet. Bullets behave in different ways and they make it possible to gain more points on some occasions; for example, a laser gun bullet can bounce off hostile projectiles a few times, thus generating more score if the angle is right for this to happen. This allows for more room in the ways that the player can get more points to raise high score with.

3.2.4 The Lens of Elemental Tetrad

This lens is supposed to define what the game is truly made of. There are four elements: mechanics, aesthetics, story and technology. Does the design include all the elements? Could the design be improved by enhancing some of the elements? Are the elements reinforcing each other? (Schell 2008, 43.)

Tinfoil deflector is a very barebones game. There just is not much to it. There is no revolutionary technology used, there is a small backstory for the events, there is only a handful of mechanics and the aesthetics are not pleasing to the eye.

You Are the Light improves on half of the elements, although some of them are in a less notable role than the others. The mechanics are the most important element for this game. Due to scope, the mechanics get most of the attention in design and development. There is a backstory, but it is not reinforced in any way or shown to the player. The story is just that the player is the planet's last hope, the guarding light, and it is trying to defend the planet from incoming cosmic threats. Aesthetics play the second most important part of the game. I wanted the game to look good despite not having the graphical skills to get to the vision I want, so the assets bought from the Unity Asset Store helped a lot. The assets purchased have a somewhat unified theme and look to them. There is not any new or exciting technology used in the game.

3.2.5 The Lens of Goals

Looking through this lens helps a developer to determine if the goals of the game are balanced and appropriate. What is the main goal of the game? Is the goal clear to the player? Is the goal achievable and rewarding? (Schell 2008, 149.)

In Tinfoil Deflector the goal is to survive for as long as possible. Eventually the player will still lose because there is no objective that enables a win. Game ends only when the planet has taken enough hits and the incoming beams are spawning ever faster and gaining more speed at each interval. This is not rewarding for the player.

Improving upon this, there will be bosses in You Are the Light, which makes it possible to win the game. The main goal then becomes defeating the boss. Defeating the boss will give the player a multiplier to the score achieved and the end screen effects are more cheerful and indicate that the player has won the game. The notable difficulty of the game combined with its gentle learning curve also contribute to the reward the player may get from beating the game.

3.2.6 The Lens of Chance

Looking through this lens a developer must determine what elements of the game involve randomness and risk. What in the game is truly random? Are there parts that just feel random but are not really? Is the randomness serving the player or taking away from the joy? What is the relationship of skill and randomness in the game? (Schell 2008, 169.)

Tinfoil Deflector has randomness in it. The beams that shoot towards the planet give randomized effects to the character when hit, and the beams' speed is set to random value in range when spawned. This could the good kind of randomness except for the fact that the player has to block the beams to avoid hit point loss, meaning that the player is not in control of the effects applied to the character. Even though the beams' effects are random, the player can still determine what the color means after hitting them a few times. Effects and colors match for the duration of the game and in the next game the effects and colors are randomized again. Beam speed randomization is fine because the variance is small enough, but it still brings the right amount of unpredictability to the game.

You Are the Light's beams do not have the same kind of randomness as Tinfoil Deflector's. In this game, beams do not apply effects to the player because the player is not in control of the effects as the player must block the beams to avoid hit point loss. There are many other kinds of randomness, though mostly on the visual side of things. Skybox is picked at random at the start of the game, so is the soundtrack that will play through the session. Planet graphics are also random. These have no effect on how the game plays, but they give a nice touch. Boss's looks and behavior are randomized so the player does not know what kind of boss there is incoming. It has to be made sure that the bosses are beatable with whatever skill and weapon the player picked up on the selection screen.

3.2.7 The Lens of Juiciness

A user interface can be called “juicy” albeit it may sound weird, although saying that interface is “dry” sounds correct if there is not enough feedback given to the user. A developer needs to think about if the interface gives the player continuous feedback for the actions they perform and that the motion created by the player actions is interesting and powerful. (Schell 2008, 233.)

Juiciness is about giving good visual and sound feedback for the actions that the player does. These actions can be anything from clicking buttons on the menu or shooting a gun or getting some sort of notification when the player gains points.

Tinfoil Deflector is not juicy at all. There is not even a highlight on the buttons when hovering over them on the main menu. Beams cause a particle effect and camera shake when they hit the surface of the planet, but the effects are bland and sound effects do not correspond to the events in any interesting or powerful way. The score is ticking away on the upper right corner of the screen in a steady pace second by second, but the player’s attention is never directed at that corner for the lack of effects.

For You Are the Light to offer the fast and intense experience wanted, the feedback from all the collisions, shooting, weapon recharging and scoring must be as juicy and explosive as possible. Most of the juiciness is added in the polishing phase of development when all the crucial parts have already been put together and everything is working. There will be more on polishing and juiciness at a dedicated chapter later.

3.2.8 The Lens of Dynamic State

When playing a game, a player needs to make decisions. These decisions are based on the information at hand. Defining what information and how it is shown to the player is a crucial part of the design and gameplay. Changes to what is shown can have a great impact on the game for the better or for the worse. What are the objects in the game? What are the attributes of the objects? What are possible states for the attributes and what triggers change of the state? What state only the game knows? What state is known by the player? Would changing who knows about a state impact the game in any way? (Schell 2008, p. 140.)

In Tinfoil Deflector only a handful of information is shown to the player. There are beams shooting towards the planet. Beams have colors that indicate what effect they have on the character when it hits them and the speed at which they travel. Speed is not shown to the player, but the color is, although the colors are mixed up each time a new game is started to make them less predictable. If there was a speed value on top of each beam, the player could more easily evaluate which beam is going to hit the planet first. Current score is shown in the top right corner of the screen. Seeing the score brings tension if you are about to make a new record so hiding it might make the game staler. Same attributes apply to the planet's health amount shown to the player.

In You Are the Light there is more information on the screen. Skills and weapons bring recharge sliders to the table. The player needs to know for how much longer he can use his skills and how much time it takes for them to recharge. Same goes for the weapons. There is also the planet's total health shown on the planet because the game is fast-paced and the information about health is crucial and it has to be where the player can easily see it.

3.2.9 The Lens of Skill

To use this lens a developer has to consider the skills that the player needs to play the game. Which skills are dominant? Are these skills making the experience a developer wants to deliver come alive? Are these skills easier for some people to utilize and does it make the game feel unfair? Can these skills be improved with practice? (Schell 2008, 153.)

One of the things that makes games fun is learning and getting better at them. Skill level needed should be balanced, not too easy and not too hard. Many games have a difficulty setting that can be changed through the options menu.

Tinfoil Deflector does not have a difficulty setting available for the player to choose a suitable level of challenge. Tinfoil Deflector's viewport is divided in half and the camera on the left is closer to the planet showing only the planet and the character circling around it. Right camera is far away from the planet so that the player can see all the beams coming at it (Figure 1). The player has to be able to divide his attention between the two cameras to succeed in the game. The

player also needs to evaluate the beams' distances to the planet to know which beam to take care of first.

In You Are the Light there is only one camera. Two cameras, while being kind of a unique feature, is a more annoying than a fun mechanic. With the addition of skills and weapons, the player now needs to manage the timing when using these and the precision of some of them. The player no longer gains points based on how many seconds he survives, but by destroying or bouncing back incoming hostile projectiles. It is possible to execute combos with some of the weapons which makes it possible to gain more score if the player has enough skill in determining the angles where two projectiles collide. The player will get better at the game by playing it. The player gets better at determining speeds of different objects, angles, recharge times of their weapons and skills and how long it takes for them to get to the position to counter the beams.

3.2.10 The Lens of Time

Lens of time is used in determining if the experience the game is giving is too short or too long. What determines the length of the game's gameplay activities? How can you change it if the game ends too early or goes on for too long? Timing is a difficult thing to get right but getting it right is crucial to the experience. (Schell 2008, 189.)

In both games Tinfoil Deflector and You Are the Light the length of the session is based on the player skill to an extent. Game ends when the planet's hit points are reduced to zero. What keeps the game from going on for too long is that the beams are spawning faster over the time of each session. The longer the game goes on the faster beams are being spawned. Also, in You Are the Light the boss becomes more difficult as time passes to disable the farming of score from shooting back the boss's own projectiles.

3.3 Designed features

Tinfoil Deflector has only a handful of features. The player character has controls for movement and a reflection shield with space key when a certain effect is applied to the character. Beams with random effects are spawned around the planet and they travel towards it. Score is gained for each second until the planet takes enough hits, which is when the game ends.

Keeping a realistic scope in mind, only a limited number of features were designed for You Are the Light. For the player character there are three weapons and three skills to choose from and for bosses there are three different kinds of behavior. Main menu has basic graphical and sound options and a screen for picking a skill and weapon. End screen is shown at the end of the game with data about the playthrough and a leaderboard that lists the scores of the players.

3.3.1 Menu

Menu is the first thing the player sees when starting up the game, so it needs to be fluid, of good visual quality, clear and easy to use. You Are the Light’s menu has an abstract particle effect at the center of the screen so that the first impression of the game arouses interest. Clicking the main buttons transitions the camera around the particle effect for some extra visual fidelity.

There will be 4 buttons on the main screen. Buttons are for starting the game, options menu, credits and quitting the game. *Start game* button takes the player to the character customization screen where the player can choose one skill and one weapon. *Options* button takes the player to options menu which has 2 more buttons, one for graphics options and one for sound options. Graphics options have sliders that control values for different graphical settings, for example, bloom and contrast. Sound options have 3 sliders, one for master volume, one for music volume, and one for sound effects (Figure 5).

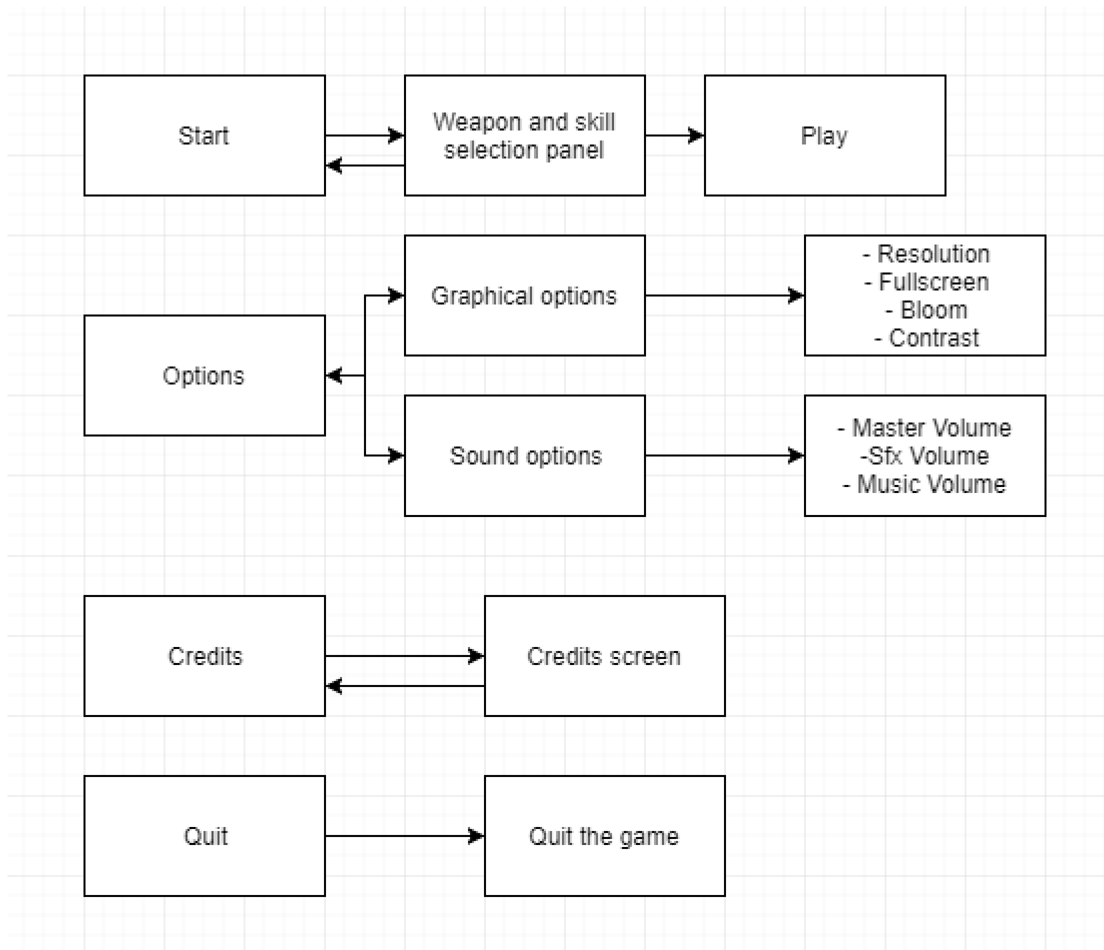

Figure 5. Menu flow-chart

All the UI-elements are themed similarly so that the menu's layout stays unified and does not cause confusion, meaning that, for example, buttons have the same graphics. There are also click sounds on the buttons to inform the player of his actions, for example, if the action was successful.

3.3.2 Weapons

There are three weapons in the game. Each weapon is used with left mouse click. There is a slider on the screen to show the cooldown of the weapon in use.

Laser gun is a weapon with a fast reload and instantaneous projectile release. Projectile is fast and moves in a straight line in the opposite direction of the character in relation to the planet.

Rocket launcher has a longer reload time, but it shoots homing missiles that track the closest target. Missiles are slower than the laser gun's bullets, so the player

still has to determine what the closest target is and does the missile have time to hit its target.

Pulse gun is a chargeable weapon, so the player needs to hold down the left mouse button to charge up the weapon. The pulse projectile gets larger for as long as the player holds down the button and expands to the accumulated size when the button is released. If the button is held for too long the projectile launches on its own.

3.3.3 Skills

There are three skills in the game. Each skill is used with right mouse button. Some of them have usage time, so the time runs out when holding skill button, and some of them have a cooldown.

Teleportation is a skill with a small cooldown that teleports the character to the other side of the planet. A useful skill when two hostile projectiles are coming towards the planet from opposite directions.

Slow time is a skill with a longer cooldown that slows down hostile projectiles for the duration that the skill is used. Slow time gives the player more time to react and to make his move.

Time rewind is a skill with a long recharge time and can be used to rewind a short amount of time. Time rewind affects beams, the boss and the planet, and not the character or the projectiles that the character has shot.

3.3.4 Planet

The player needs to protect the planet to avoid losing the game. Each hostile projectile that hits the planet reduces its hit points by one. Planet graphics are randomized so they will be different each time the game is played. The planet rotates around itself but that doesn’t affect the collider attached to it.

When the planet’s health reaches zero, explosion particles are spawned. Also, the explosion is accompanied by a loud bang sound effect and a screen shake to

give a satisfying result. End screen is activated when there has been some time for the effects to run their course.

3.3.5 Boss behaviors

Bosses have randomized behaviors. There are three shoot patterns and three movement patterns and each time the game is played one shoot pattern and one movement pattern is chosen for the boss. Boss graphics are also randomized. The boss will always have a red orb in the middle of it that represents the hitbox of the boss. When the boss moves, it leaves a trail behind it and this trail also stays the same despite the graphic randomization that happens.

For movement patterns there is a circular movement, teleportation and diagonal movement. When the boss teleports, a flash particle effect is spawned at the location of disappearance and reappearance. There is also a sound effect playing that represents the teleportation. At the end of the teleportation the boss shoots projectiles towards the planet and then teleports again and this continues for a randomized number of times. When the boss moves continuously it also shoots at a steady pace. Diagonal movement is fast and when the boss reaches the corner where it is moving, it shoots a burst.

When the boss's health drops to zero, the same kind of explosion happens as when the planet is destroyed. The boss's health is represented by a health bar that is in the upper center of the screen. When the boss takes damage there is an indication on the health bar and a screen shake.

3.3.6 End screen and leaderboard

End screen shows up either when the player has destroyed the boss or when the planet's hit points reach zero. There are three values shown on the end screen: the length of the session, score and the number of destroyed beams. When the

numerical values are shown there is a visual effect in place that shows that the scores are being calculated and a corresponding sound plays in the background.

Leaderboard shows as a list, and the player's score is highlighted on the list. If the player gains a new high score there is feedback for that in the form of tweened elements and sound effects.

4 Implementation Process

Implementing the features for Tinfoil Deflector was a somewhat chaotic process. There was no time for any kind of architectural design for the code and thus everything was just scrambled together as fast as possible, but that is part of the fun of game jams. The team was also a bit sleep deprived and that can be seen in the quality of the code written for the game.

With You Are the Light time was less of an issue and so the code is more organized and modular. Good programming practices were used and there was more architectural design when thinking about the larger features that were to be implemented.

Implementation process goes through the steps of first picking a feature to be fully implemented based on whether the feature can be implemented independently or if the feature is crucial for other features to be implemented. If the feature is large and other features depend on it, the architecture of that feature must be well designed to avoid pitfalls during the development. More simple features that do not depend on other features can be developed and designed in a "vacuum" of sorts. I manually test the feature when working on it. When the feature is working as intended, I move to the next feature on the list (Figure 6).

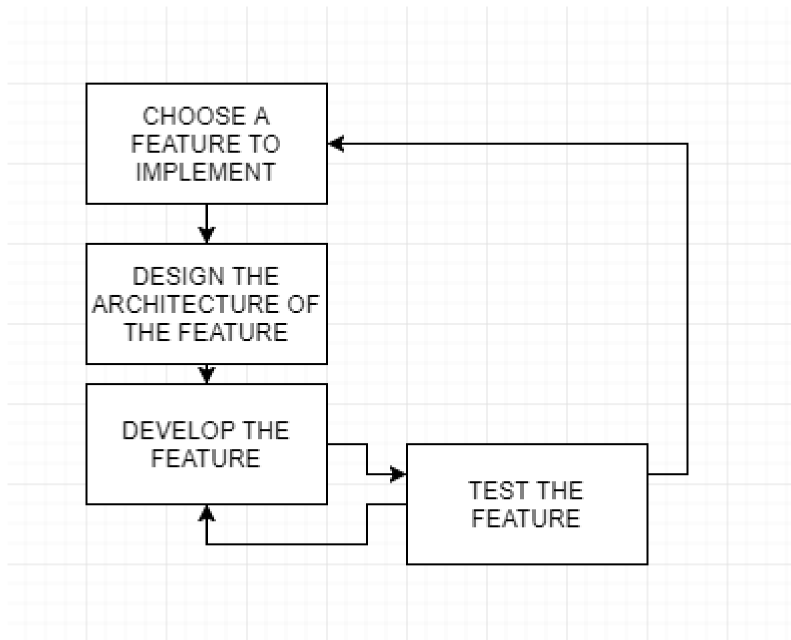

Figure 6. Workflow of the implementation process.

Manually testing everything is a bit tedious and there are solutions to make that process more fluid, like the Unity test runner and writing unit tests. Writing a good and comprehensive test suite takes time and I rather put that time in the development to get out more features. This would bite me in the end if the project was much larger, but fortunately this is a very small project. If the game is developed further and it gets more complex and larger, unit tests are a must.

4.1 Writing Clean Code

Robert Martin (also known as Uncle Bob) is a software engineer and an author. I have read his book called *Clean Code* (2009). *Clean Code* is about professional agile software craftsmanship and touches topics like proper naming, function structure, commenting code, formatting code, data structures, error handling, writing unit tests and many more.

Software developers on different discussion forums like Reddit, have different opinions about Robert Martin's teachings. Others think that Uncle Bob is a god-like figure and they keep *Clean Code* as their bible. Others think that his principles are not practical and that it is impossible to expect the amount of discipline he would want programmers to have. His thoughts on Test Driven

Development (TDD) are especially divisive on the community. TDD is a practice in which tests are written before the actual code for the software. Some people think that TDD is too hardcore methodology and that it takes too much time from the development. There is no denying that writing a full test suite to test all the pieces of code makes developing the software easier and safer the bigger it gets, but it takes quite a lot of time, and time is money for companies. Using TDD might save time in the end if used properly and if the project is developed for a long period of time or is subjectable to constant changes.

Most of the Robert Martin's principles are logical, practical and on point. I do not think that his opinions are the word of god, but there are some very good principles and ideas that all developers should try to follow and a few of those are described. I wanted to make You Are the Light's code quality better following these principles to ensure that the game is easy to develop further in the future. Naming variables and functions and writing functions with good structure are important parts of software development and how well you do these tasks affects how much commenting you need to do, which hopefully is none. These practices were not particularly well applied in the development of Tinfoil Deflector. Naming in Tinfoil Deflector's code is almost acceptable but there are some hiccups like using the "_" prefix for private variables or a class named Planet that only rotates the object that it is attached to which could have been named just as a "Rotator". Many of the functions in Tinfoil Deflector are too long and they should be extracted to smaller functions to make the code more readable.

4.1.1 Naming

Naming things is an important aspect of software development and should be treated with great care. We name folders, files, classes, variables, functions and function arguments. That is a whole lot of naming to do so naming must be done well for the code to be easily readable and logical. (Martin 2009, 17-18)

Name must reveal the intention of the named subject. Name of a variable, function or a class should tell the person who reads the code why it exists, what

it does and how it is used. Making up this kind of good name takes time, but ultimately saves more time in the end. Everyone who reads your code has an easier time understanding what the code does if the naming is done right. If you must leave a comment for the name, it does not reveal its intent well. (Martin 2009, 18.)

Names should not give false clues about its intention. If you have an array of customers you should not name it customerList because it is not a list, it is an array. People reading the code might think that customerList is a List and try to use it that way for their own purposes which probably leads to errors and loss of work time. (Martin 2009, 19.)

Names should be pronounceable. Developers should not use prefixes and short hands in naming. This kind of naming makes it harder to understand the intention and it makes it hard to discuss about these things with other people. (Martin 2009, 22) Classes and objects should have noun or noun phrase names. Class names unlike method names should not be a verb. (Martin 2009, 25.)

Writing funny names for variables and functions might be fun at the time and to all the current participants, but when other people outside that group try to read your code, they do not have the slightest clue what the funny named functions and variables are supposed to mean and used for. You should have one word in use per concept. For example, if you name your get methods getSomething, do not use fetchSomething somewhere else. (Martin 2009, 26.)

4.1.2 Comments

Leaving a comment is a failure of sorts. If you need to leave a comment to describe something, you have failed to express yourself clear enough in the code. Comments are a necessary evil because programmers can’t always express themselves well enough. Each time you leave or are about to leave a comment you should think if you could express it somehow in the code. (Martin 2009, 55.)

Comments are a bad thing because they often mislead the reader. Code changes and evolves and thus comments grow old and they might not be accurate anymore when time passes and code changes. Inaccurate comments are worse

than no comments. They mislead the reader and hide the true functionality of the code. Programmers should keep their comments updated and accurate but unfortunately this is not the case. Truth about what the code does can only be found in the code, so the code should be as expressive as possible. (Martin 2009, 55.)

Writing redundant comments is a form of bad commenting. If the code already explains itself well enough why would you have to comment it? For example, if you have a class with hitPoints variable, don't write a comment above it that describes that these are the hit points the object has. (Martin 2009, 61.)

Commenting everything is a bad practice as it clutters the code, lies and leads to confusion and disorganization. You should name everything correctly, so the intention is revealed reading the name, not a comment (Martin 2009, 63). Some people like to leave journal comments at the top of the file that describes the changes made to that file. This should not be needed in the modern world where we have perfectly valid version control systems available. (Martin 2009, 63-64.)

Do not leave commented out code hanging around. People reading the code do not know if they can delete it or not. Source control takes care of this as well nowadays, so you can always go back to your code if you need it. (Martin 2009, 68-69.)

Some comments are necessary or beneficial. However, the best kind of comment is the one that you found a way not to write at all. Beneficial comments can be ones that go beyond useful information and provides the intent behind a decision made or for clarification of an argument or return type that is bound to be obscure because of a standard library functionality. Sometimes it's good to warn about the consequences of using a function, for example, if it takes too long to run or that it is not suitable for test usage. IDE's can interpret TODO comments and usually highlights them differently from other comments. TODO comments should explains why the function is not fully implemented and give information what should be added in it. (Martin 2009, 56-58.)

4.1.3 Functions

Functions should be small. Functions should do one thing and one thing only to they keep their small size. Small functions are easier to read, easier to name and easier to understand. (Martin 2009, 34.)

Code should be able to be read like a top-down narrative. This is called The Stepdown Rule. Every function should be followed by those that are at the next level of abstraction. Learning to write functions this way is crucial for keeping them small. (Martin 2009, 37.)

It is difficult to write a small switch or if-else statement. They make functions grow in line size and it also makes it more difficult for the function to do only one thing. Switch statements can be tolerated if they only appear once, are used to create polymorphic objects and are hidden behind an inheritance relationship so that the rest of the system can’t see them. (Martin 2009, 37.)

Name functions descriptively. Smaller the function, easier it is to name it. A long expressive name is better than a short puzzling name or a long descriptive comment. (Martin 2009, 39.)

Best amount of function arguments is none. One or two arguments is acceptable, but three arguments should be avoided at all costs. Arguments make testing more difficult as they increase the amount of test cases needed. (Martin 2009, 40.)

Passing a Boolean into a function as an argument is a terrible practice. It implies already that the function does at least two things which conflicts with single responsibility principle. (Martin 2009, 41.)

4.2 Patterns

Here are few examples of programming patterns that were used in developing You Are the Light. Singleton pattern was also used in the game jam game, but the object pooling pattern was not, though it would have been extremely efficient there because of the number of projectile objects being instantiated at the same time.

Patterns are important because they give structure to the code, they are familiar to other developers making it easier for them to follow the code and they often boost the performance of the code like in the case of object pooling.

4.2.1 Singleton pattern

Singleton is a class that can be instantiated only once (Figure 7). Usually in singletons there is a static member called Instance, so there is a global access to it. (Machusak 2011.)

```
public static ProjectilePooler Instance;
public GameObject[] ProjectilePrefabs;

private readonly Dictionary<ProjectileType, Pool> projectilePools = new Dictionary<ProjectileType, Pool>();

// Singleton. Makes sure there is only one instance of this script
private void Awake(){
    if (Instance == null){
        Instance = this;
    }
    else{
        Destroy(this);
    }

    InitializePools();
    PoolProjectiles();
}
```

Figure 7. Singleton pattern used in ProjectilePooler class.

Singletons are useful because they can be accessed from anywhere in the code. For example, in Unity you do not have to find or get the singleton in any way, but you can refer to it with the class name and its static member called instance like this, ProjectilePooler.Instance, and then call a public method of the singleton class you want to use.

There are also some problems with using singletons. Singletons produce tight coupling. When you call the singleton from multiple places in the code, the code becomes dependent of the singleton. (Cosentino 2013.)

If you want to change the way the singleton class behaves you might have to rewrite large parts of your code to make it happen. The same goes if you notice during the development process that you need another instance of the class that is a singleton. In You Are the Light there are a few use cases for singletons. Projectile pooler is a singleton as there needs to be ever only one of them in the scene since every projectile pool needs to be accessible from the same place for clarity's sake.

4.2.2 Object Pooling

Memory management is important for performance in game development and even more so if you develop for mobile phones and consoles with more restricted memory capabilities than desktop computers. It is better to allocate the memory you need up-front when initializing the game rather than in the middle of the game during a frame. Instantiating loads of objects during runtime and then destroying them can cause hitches in the framerate if garbage collector runs and Unity's Instantiate function can be CPU heavy if used extensively. (Izzo, 2018.)

To avoid issues with memory allocation and possible hitching of the framerate due to garbage collector running we can resort to *object pooling*. Using object pooling you no longer instantiate and destroy objects at runtime, but you instantiate objects at the start of the game, put them on a list and then get the wanted object from the list when you need it. When you get the object from the list you activate it and when you no longer need it you disable the object and return it to the list, also referred to as the pool.

In You Are the Light there are multiple projectiles on the screen at the same time, so it is best to pool them to reduce the impact on performance. The pool itself holds a reference to the pooler and the index of the prefab it is based on. There is a list of objects in the Pool class. There is a GetPooledProjectile function that gets the last projectile in the list and returns it, and if there are no projectiles available, the pool calls the pooler to create a new projectile. All the projectiles hold a reference to the pool so AddToPool can be called when they are disabled. (Figure 8)

```
1   using System.Collections.Generic;
2   using UnityEngine;
3
4   public class Pool {
5       private readonly ProjectilePooler myProjectilePooler;
6       private readonly int myProjectilePrefabIndex;
7       private readonly List<GameObject> pooledObjects = new List<GameObject>();
8
9       public Pool(ProjectilePooler projectilePooler, int projectilePrefabIndex) {
10          myProjectilePooler = projectilePooler;
11          myProjectilePrefabIndex = projectilePrefabIndex;
12      }
13
14      public GameObject GetPooledProjectile() {
15          GameObject projectile = null;
16          if (pooledObjects.Count == 0) {
17              myProjectilePooler.PoolProjectile(myProjectilePrefabIndex, this);
18          }
19          projectile = pooledObjects[pooledObjects.Count - 1];
20          pooledObjects.RemoveAt(pooledObjects.Count - 1);
21          return projectile;
22      }
23
24      public void AddToPool(GameObject projectile) {
25          pooledObjects.Add(projectile);
26      }
27
28  }
29
```

Figure 8. Pool class

ProjectilePooler is responsible for initializing the pools. There is a dictionary that holds all the pools that are needed based on projectile prefabs placed in an array through the editor. In the start method the pools are initialized first and then the projectiles are pooled into the pools. (Figure 9)

```
private void InitializePools() {
    for (var i = 0; i < ProjectilePrefabs.Length; i++) {
        var projectileType = ProjectilePrefabs[i].GetComponent<IPoolableObject>().GetProjectileType();
        var pool = new Pool(this, i);

        projectilePools.Add(projectileType, pool);
    }
}

private void PoolProjectiles() {
    for (var prefabIndex = 0; prefabIndex < ProjectilePrefabs.Length; prefabIndex++) {
        var poolableObjectPrefab = ProjectilePrefabs[prefabIndex];
        var projectileCount = poolableObjectPrefab.GetComponent<IPoolableObject>().GetProjectileCount();
        var projectileType = poolableObjectPrefab.GetComponent<IPoolableObject>().GetProjectileType();
        var pool = projectilePools.FirstOrDefault(item => item.Key == projectileType).Value;

        for (var i = 0; i < projectileCount; i++) {
            PoolProjectile(prefabIndex, pool);
        }
    }
}

public void PoolProjectile(int prefabIndex, Pool pool) {
    var projectile = Instantiate(ProjectilePrefabs[prefabIndex], transform.position, Quaternion.identity);
    projectile.transform.parent = null;
    projectile.GetComponent<IPoolableObject>().SetPool(pool);
    projectile.SetActive(false);
    pool.AddToPool(projectile);
}
```

Figure 9. ProjectilePooler class.

There is also a function called GetPoolOfType (Figure 10) that returns a pool based on an enum argument. All the scripts that need a projectile pool call this

method and save the returned pool to a variable for later use. For example, all the weapons the player character possesses get their pool of projectiles this way.

```
public Pool GetPoolOfType(ProjectileType projectileType) {
    var poolToReturn = projectilePools.FirstOrDefault(projectile => projectile.Key == projectileType).Value;
    if (poolToReturn == null) {
        Debug.LogError("Pool to be returned is null");
    }

    return poolToReturn;
}
```

Figure 10. Function that returns a pool based on an argument.

4.3 Main Feature Implementation

These are the main features of You Are the Light that took most of the development time and are the most important ones. There are a lot of small classes in the project, but they are not examined here.

Some of the features here can be found in Tinfoil Deflector too, like the MusicPlayer or projectiles but their implementations differ greatly. Other features like options, bosses, skills, weapons and sound effects are new to You Are the Light and many of them were designed and included to the implementation list with the help of the process of “looking through the lenses”.

4.3.1 Options

Sound options are simple to make. There is a master mixer with three audio mixer groups, one for master volume, music volume and sound effects volume. There is a slider for each of those and the sliders value is bound to their volume value, so when the player changes the slider, the volume gets updated in the audio mixer group to the slider’s value.

Graphical options are a bit tougher to produce. I wanted the player to be able to change bloom, contrast, resolution and whether the screen is on full screen mode or not. Contrast and bloom are tied to the post processing profile which is a camera effect template of sorts. I wanted the player to see the change in bloom and contrast when they change the slider value, so I update the post processing

profile each time a change is made to the value of the slider. This is not efficient, but it enables the player to see the change, performance being the tradeoff. The methods are public because Unity's UI event triggers need public methods to be dragged onto the events taking the slider value as an argument. (Figure 11)

```
59      public void SetBloom(float bloomIntensity) {
60          var bloomSettings = postProcessingProfile.bloom.settings;
61          bloomSettings.bloom.intensity = bloomIntensity;
62          postProcessingProfile.bloom.settings = bloomSettings;
63          BloomIntensity = bloomIntensity;
64      }
65
66      public void SetContrast(float contrastValue) {
67          var colorGradingSettings = postProcessingProfile.colorGrading.settings;
68          colorGradingSettings.basic.contrast = contrastValue;
69          postProcessingProfile.colorGrading.settings = colorGradingSettings;
70          ContrastIntensity = contrastValue;
71      }
72
```

Figure 11. Public methods to set contrast and bloom values to the post processing profile.

Choosing the resolution is handled with a dropdown menu. First when the game is loaded the resolution dropdown is initialized. All the available resolutions are gathered in a list of strings so that they can be shown in a readable manner to the player. When the player picks a resolution from the list it is set as the current resolution based on the value of the index in the dropdown menu.

Options data is saved to a local folder on the player's computer as a json file. The json file contains an OptionsData object that holds the values of the saved sound and graphics settings.

4.3.2 Music Player

The MusicPlayer is responsible for playing all the music in the game. At the start of the game, a random soundtrack collection is picked from an array where they are stored. There are three soundtracks in each of the collections, one of them is an intro track, second is a continuous track and third is a more intense track. Each of the soundtracks are initialized. They are given an audiosource and the audiosource's variables are set to loop and volume is set to zero because the tracks are always faded in. Each of the audio sources are also set up with the

music audio mixer group so that the volume applied through options affects the music. Tracks can be crossfaded, so for example, when the music is supposed to get more intense the current track fades out and the more intense track fades in. (Figure 12)

```
public class MusicPlayer : MonoBehaviour {

    public AudioMixerGroup AudioMixerGroup;
    public SoundtrackCollection[] SoundtrackCollections;

    private SoundtrackCollection currentSoundtrackCollection;
    private readonly AudioSource[] audioSources = new AudioSource[4];

    private void OnEnable() {
        Planet.NearDeathEvent += Intensify;
        Planet.GameEndEvent += FadeOutAll;
    }

    private void OnDisable() {
        Planet.NearDeathEvent -= Intensify;
        Planet.GameEndEvent -= FadeOutAll;
    }

    private void Start() {
        currentSoundtrackCollection = SoundtrackCollections[Random.Range(0, SoundtrackCollections.Length)];
        InitSoundTracks();
        StartCoroutine(StartPlaying());

    }

    private void InitSoundTracks() {
        for (var i = 0; i < currentSoundtrackCollection.Soundtracks.Length; i++) {
            var source = gameObject.AddComponent<AudioSource>();
            source.clip = currentSoundtrackCollection.Soundtracks[i];
            source.loop = true;
            source.volume = 0;
            source.outputAudioMixerGroup = AudioMixerGroup;
            source.Play();
            audioSources[i] = source;
        }
    }

    private IEnumerator StartPlaying() {
        audioSources[0].DOFade(1f, 6f);
        yield return new WaitForSeconds(currentSoundtrackCollection.Soundtracks[0].length);
        CrossFade(0, 1);
    }

    private void CrossFade(int fadeOutIndex, int fadeInIndex) {
        audioSources[fadeOutIndex].DOFade(0f, 3f);
        audioSources[fadeInIndex].DOFade(1f, 3f);
    }

    private void FadeOutAll() {
        foreach (var audioSource in audioSources) {
            audioSource.DOFade(0, 15f);
        }
    }

    private void Intensify() {
        CrossFade(1, 2);
    }
}
```

Figure 12. MusicPlayer script

The MusicPlayer has subscribed to different events. When the Planet script's NearDeathEvent is fired, the music gets more intense and when GameEndEvent is fired all the music is faded out.

4.3.3 Planet and beam spawner

The Planet script is responsible for firing the events (Figure 13) when the game ends unfavorably for the player. The Planet script has a health variable and when it reaches zero the game ends. There is an event triggering when health is brought to zero and all the scripts that are subscribed to that event execute their associated methods, usually to disable ongoing effects so they do not interfere with the game ending.

```
public class Planet : MonoBehaviour {
    public delegate void HealthLoss(int health);
    public static event HealthLoss HealthLossEvent;

    public delegate void NearDeath();
    public static event NearDeath NearDeathEvent;

    public delegate void GameEnd();
    public static event GameEnd GameEndEvent;

    public delegate void PlanetExplosion();
    public static event PlanetExplosion PlanetExplosionEvent;
```

Figure 13. Planet scripts events.

There is a script called BeamSpawner (Figure 14) attached to the planet prefab. BeamSpawner spawns beams when the isActive variable's value is set to true. Beams are spawned in intervals and the interval is decreased in the Update method by Time.deltatime. There are multiple beams spawned inside an IEnumerator based on a random value that is passed to the coroutine each time it is run.

```
public class BeamSpawner : MonoBehaviour {

    [SerializeField] private float beamSpawnInterval = 2f;

    private float beamTime;
    private Pool beamPool;
    private bool isActive;

    private void Start () {
        beamPool = ProjectilePooler.Instance.GetPoolOfType(ProjectileType.SmallBeam);
    }

    private void Update () {
        if (!isActive) return;
        beamTime -=  Time.deltaTime;
        beamSpawnInterval -= 0.1f * Time.deltaTime;
        if (beamTime > 0) return;
        beamTime = beamSpawnInterval;
        StartCoroutine(SpawnBeamsRapidly(Random.Range(1, 4)));
    }

    public void Activate() {
        isActive = true;
        beamTime = beamSpawnInterval;
    }

    public void DeActivate() {
        isActive = false;
        beamTime = beamSpawnInterval;
    }

    private IEnumerator SpawnBeamsRapidly(int beamAmount) {
        for (var i = 0; i < beamAmount; i++) {
            if(isActive)
                SpawnBeam();
            yield return new WaitForSeconds(beamSpawnInterval / beamAmount);
        }
    }

    private void SpawnBeam() {
        var projectile = beamPool.GetPooledProjectile();
        projectile.GetComponent<Beam>().SetTarget(transform);
        projectile.transform.position = transform.position;
        projectile.transform.position = new Vector2(transform.position.x, transform.position.y) + (Random.insid
        projectile.transform.position = new Vector3(projectile.transform.position.x,
            projectile.transform.position.y, transform.position.z);
        projectile.SetActive(true);
    }
}
```

Figure 14. BeamSpawner script.

4.3.4 Character and skills

Character is one of the more important parts of the game. Character is controller through YALCharacterController (CharacterController class name is already in use in Unity) script which is responsible for reading input and triggering actions based on that input, like movement, shooting and skills. (Figure 15)

```
private void Update() {
    if (Input.GetAxisRaw(HorizontalMovement) < 0) {
        if (movementSpeed < 5) {
            movementSpeed += 50f * Time.deltaTime;
        }

        MoveRight();
    }
    else if (Input.GetAxisRaw(HorizontalMovement) > 0) {
        if (movementSpeed < 5) {
            movementSpeed += 50f * Time.deltaTime;
        }
        MoveLeft();
    }
    else {
        movementSpeed = 0;
    }

    if (Input.GetButtonDown(Skill)) {
        skill.OnButtonDown();
    }

    if (Input.GetButtonUp(Skill)) {
        skill.OnButtonUp();
    }

    if (Input.GetButtonDown(Shoot)) {
        weapon.OnButtonDown();
    }

    if (Input.GetButtonUp(Shoot)) {
        weapon.OnButtonUp();
    }
}
```

Figure 15. YALCharacterControllers Update method that reads input and triggers actions based on the input.

The player movement is based on angle calculations and changing the transform’s position rather than moving by giving a rigidbody some force. This is because the movement is circular, and it is difficult to produce that kind of movement with a rigidbody. Calculating the movement involves adding the movement speed variable multiplied with the unscaled delta time to a movement angle variable and then determining the sin and cos of the movement angle multiplied with the wanted distance from the planet and finally adding the calculated movement offset to the planets transform position to get the point where the player needs to be.

All the skills are inherited from a base abstract class called Skill. Skill has OnButtonDown and OnButtonUp methods that are used for the activation of the skill but the functionality for different skills reside in the inherited scripts. Skill could have been just an interface but there is enough functionality in Skill that I

thought it is better to be an abstract class to stay true to Don't Repeat Yourself (DRY) principle. (Figure 16.)

```
public abstract class Skill : MonoBehaviour {

    public SkillData SkillData;

    protected YALCharacterController CharacterController;

    public abstract void OnButtonDown();
    public abstract void OnButtonUp();

    public void SetSkillData(SkillData data) {
        SkillData = data;
    }

    public void SetCharacterController(YALCharacterController controller) {
        CharacterController = controller;
    }

    public Skill GetSkill() {
        return this;
    }
}
```

Figure 16. Skill abstract class.

Teleportation skill calculates the point opposite to the character's current position in relation to the planet's position and moves the character there. When the skill is used, the character disappears from the scene and a flash particle is instantiated. When the OnButtonUp is pressed, the character appears at the teleportation point across the planet and again a flash particle is instantiated (Figure 17).

```
private void TeleportStart() {
    SkillData.Timer = 0;
    isActive = true;
    startSoundSource.Play();
    Instantiate(portal, transform.position, Quaternion.identity);
    ShockWave.Get().StartIt(transform.position, 10f, 0.001f, 0.015f, 0.05f);
    character.SetActive(false);
    transform.position = GetTeleportPoint();
    CharacterController.ChangeAngle(3);

}

private void TeleportEnd() {
    isActive = false;
    endSoundSource.Play();
    Instantiate(portal, transform.position, Quaternion.identity);
    ShockWave.Get().StartIt(transform.position, 10f, 0.001f, 0.015f, 0.05f);
    character.SetActive(true);
}

private Vector3 GetTeleportPoint() {
    var distance = (CharacterController.GetDistanceFromPlanet()) * 2;
    var direction = CharacterController.CurrentPlanetTransform.position - transform.position;
    return transform.position + (direction.normalized * distance);
}
```

Figure 17. Teleportation skill scripts methods.

All the skills have a scriptable object called SkillData added to them as a variable. SkillData holds information about the skill's cooldown time and current recharge time, skill's start sound effect, end sound effect and the particle effect that happens when the skill is used. With the SkillData being a scriptable object, it is possible to just drag the SkillData through the inspector to a public variable in any script that wants to keep track of the skill recharge time, for example, the skill slider uses this data to adjust the value of the slider to be the same as in the SkillData to show the player when the skill is recharged again.

The SlowTime skill slows time for everything else except the player character. Using the skill reduces Unity's Time.timescale variable from 1.0 to 0.5 which means that everything in the game is slowed down to half of their original speed.

4.3.5 Projectiles

Projectiles are inherited from an abstract class because there is a lot of common functionality that is used for all of them. Projectile prefabs are tagged with either "PlayerProjectile" or "HostileProjectile", so they can be differentiated from each other. Projectile abstract class has a ScoreAction delegate and ScoreAction event OnHit that can be triggered from any of the projectile scripts that are inherited from Projectile. (Figure 18)

```
public class Projectile : MonoBehaviour, IPoolableObject {

    public delegate void ScoreAction(int score, Vector3 hitPosition);
    public static event ScoreAction OnHit;
    public ProjectileType MyProjectileType;
    public int MyProjectileCount;

    protected Transform CharacterTransform;

    private Pool myPool;


    public ProjectileType GetProjectileType() {
        return MyProjectileType;
    }

    public int GetProjectileCount() {
        return MyProjectileCount;
    }

    public void SetPool(Pool pool) {
        myPool = pool;
    }

    public void ReturnToPool() {
        myPool.AddToPool(gameObject);
    }

    public void SetCharacterTransform(Transform charTransform) {
        CharacterTransform = charTransform;
    }

    protected void TriggerOnHitEvent(int score, Vector3 hitPosition){
        if (OnHit != null) OnHit(score, hitPosition);
    }
}
```

Figure 18. Projectile abstract class from which all of the projectile scripts are inherited.

There is a script called FloatingScoreSpawner that is subscribed to the OnHit event and then, based on the event parameters, places a floating score object that shows the amount of score gained to the player. Scorekeeper script is also subscribed to this event and the script internally increases the score the player has accumulated based on the event parameters.

4.3.6 Bosses

Original boss behavior design included randomization of different movement and shoot sets to give the player a unique boss fight experience each time that the

game is played. This proved to be difficult to implement with the time restrictions on making the game and writing this thesis, so the design had to be changed. Also, the design was bad because it is boring to fight against a boss that moves and shoots in only one way throughout the fight.

Design that was implemented in place of the original one includes a BossController, BossMovementBehaviour and BossShootingBehaviour scripts. BossController changes the state of movement behavior and movement behavior changes the state of the shooting behavior. This is a simple state machine solution.

BossController has minimum and maximum state length variables. A random time value is generated from these two variables and the random value is the length of the state that is being changed to. The script is subscribed to the Planet script's NearDeathEvent which is triggered when the planet's health is at half. This sets the boss to be activated and triggers Activation coroutine that places the boss to correct position above the planet, instantiates spawn particles and enables the boss model so that the player can see the boss. (Figure 19)

```
public class BossController : MonoBehaviour {
    public bool IsActive;
    public GameObject Model;

    [SerializeField]private GameObject spawnParticles;
    [SerializeField]private float minStateLength;
    [SerializeField]private float maxStateLength;
    [SerializeField]private float distanceFromPlanet;

    private float timer;
    private BossMovementBehaviour movementBehaviour;

    private void Start(){
        movementBehaviour = GetComponent<BossMovementBehaviour>();
    }

    private void OnEnable() {
        Planet.NearDeathEvent += Activate;
        Planet.PlanetExplosionEvent += Disappear;
    }

    private void OnDisable() {
        Planet.NearDeathEvent -= Activate;
        Planet.PlanetExplosionEvent -= Disappear;

    }

    private void Update() {
        if (!IsActive) return;
        minStateLength -= 0.01f * Time.deltaTime;
        maxStateLength -= 0.01f * Time.deltaTime;
        timer -= Time.deltaTime;
        if (timer > 0) return;
        timer = Random.Range(minStateLength, maxStateLength);
        movementBehaviour.ChangeState(
            movementBehaviour.GetState() == MovementState.Circular
                ? MovementState.Teleportation
                : MovementState.Circular, timer);
    }

    private void Activate() {
        StartCoroutine(Activation());
    }

    private IEnumerator Activation() {
        yield return new WaitForSeconds(4.5f);
        var planetTransform = FindObjectOfType<Planet>().transform;
        movementBehaviour.Target = planetTransform;
        transform.position = new Vector3(planetTransform.position.x, planetTransform.position.y + dist
        Instantiate(spawnParticles, transform.position, Quaternion.identity);
        yield return new WaitForSeconds(0.1f);
        Model.SetActive(true);
        IsActive = true;
    }

    private void Disappear(){
        IsActive = false;
        Instantiate(spawnParticles, transform.position, Quaternion.identity);
        Model.SetActive(false);
    }
}
```

Figure 19. BossController script.

Movement behavior has circular and teleportation states. When the state is circular the boss circles around the planet at a set speed. When the state is changed to circular, shooting state is changed to continuous.

Changing to teleportation state fires up a coroutine that handles different parts of the teleportation. Teleportation has different phases that are hide, change place, show and shooting a burst and each of these take a certain amount of time to complete in relation to the time parameter. Shooting behaviors ChangeState

method takes a time value as a parameter so the length of the state can be calculated. Movement behavior gets a reference from the planet to its waypoints that are used as the teleport positions.

Shooting behavior has continuous and burst states. In continuous state the boss fires projectiles at a regular rate while burst state fires multiple projectiles in rapid succession.

4.3.7 Leaderboard

Leaderboard got ditched during the development due to a time shortage. In place of a leaderboard the game now has a simple high score system. High score is saved to a json file and the file is read when the end screen appears. If the player has accumulated a score greater than what is saved in the file, the new score is saved. This json solution makes it possible for the player to tweak the high score in the file directly, but as this is a single player game and the high scores are not compared with other players' high scores, the possibility of tweaking the score does not matter.

Leaderboard system can be implemented with Unity's social API with relative ease, but it still takes a large amount of time to change the appearance of the end screen and its elements to suit the leaderboard. Leaderboard also would have needed login features to be able to differentiate players from one another.

4.3.8 Sound Effects

Sound effects are scriptable objects. SoundEffect holds different variables corresponding to variables in the Unity's audiosource component. There is an Init method that takes a gameobject as a parameter. Init method adds audiosource component to the gameobject given as a parameter and sets that audiosources variable values to be the same as in the sound effect script and then returns the audiosource.

There is also GetRandomizedPitch which returns a random value between the minPitch and maxPitch variables, so it can be placed as the pitch of the audiosource before playing the sound effect.

CreateAssetMenu attribute makes it possible to make new scriptable object instances of the scriptable object through Unity's dropdown menus. Each of these scriptable object instances can be given different values through the editor because all of the variables are serialized. (Figure 20)

```
[CreateAssetMenu(fileName = "New Sound Effect", menuName = "Sound Effect")]
public class SoundEffect : ScriptableObject {
    [SerializeField]private  AudioClip soundEffect;
    [SerializeField]private  AudioMixerGroup audioMixerGroup;
    [SerializeField]private  bool canLoop;
    [SerializeField]private  bool playOnAwake;

    [SerializeField]private  float volume;
    [SerializeField]private  float pitch;
    [SerializeField]private  float minPitch;
    [SerializeField]private  float maxPitch;
    [SerializeField]private  float spatialBlend;

    public AudioSource Init(GameObject soundUser) {
        var audioSource = soundUser.AddComponent<AudioSource>();
        audioSource.loop = canLoop;
        audioSource.spatialBlend = spatialBlend;
        audioSource.volume = volume;
        audioSource.pitch = pitch;
        audioSource.playOnAwake = playOnAwake;
        audioSource.outputAudioMixerGroup = audioMixerGroup;
        audioSource.clip = soundEffect;
        return audioSource;
    }

    public float GetRandomizedPitch() {
        return Random.Range(minPitch, maxPitch);
    }
}
```

Figure 20. SoundEffect scriptable object.

5 Polishing Process

Polish is a subjective term. In a general sense, a polished game is one that is devoid of issues that pull the player out of the experience. Polishing can be seen as consistency of experience. If a game has beautiful graphics but is balanced horrendously and these balance issues pull the player out of the experience, the

game is not well polished. Polishing is the last 10 percent of work where everything is working in the game and it is time to focus on the details. Polishing is also one of the more time-consuming processes sometimes taking up the same time as the first 90 percent of development. (Zoss 2009, 1.)

The most important part of creating a polished game is time management. Developers achieve polish via allocating time in their schedules for polishing. Some see time allocated for polishing as a buffer time in a project schedule. Buffer time is meant to be used in dealing with doubt and uncertainty, even though the amount of polish required is regularly uncertain, these two uncertainties are not to be considered the same. It is important to leave time for polishing at the end of the project, but it is also important not to work on the game indefinitely. (Zoss 2009, 2.)

Polishing a game can include tasks like making sure the game is stable, performs well and the smoothing of some of the rougher edges. Code and assets need to be optimized so that memory consumption is kept at minimum. Online features are somewhat difficult to optimize and test until the later stages of development because majority of the game needs to be complete to be able to do so. (Zoss 2009, 3.)

Tinfoil Deflector got no polishing at all. The few effects happening on the screen feel bland and the sound effects do not fit well with the visual effects. There is not any kind of feedback given to the player for his actions. Tinfoil Deflector is also not optimized in any way though the game is not particularly heavy on the graphics or the CPU usage, there still would have been room for some optimization.

With the development of You Are the Light, polishing was kept in mind throughout the development process. The effects were hand-picked carefully, and it was made sure that sound effects and visual effects match each other and the actions that they are the products of. An extra layer of polish was added in the last steps of the development which included tweaking a number of effects, for example, particle sizes, frequency of tweens and volume of sound effects.

5.1 Sounds

There are lots of sounds in the game from music to different kinds of sound effects. Now they are just placed in the game and working, but there has not been an evaluation of how the sound levels are balanced. All the music comes from an asset package bought through Unity Asset Store and they are balanced between themselves but not with the sound effects that come from another package.

Sound effects are more difficult to balance because they differ from each other so greatly. These effects must be balanced by manually playing the game and each of the sound effects must be assessed separately. Balance of the volume depends on the experience that the sound is supposed to create; for example, when the planet explodes, the effect has to sound impactful and thus giving the explosion sound more volume might do the trick. (Figure 21)

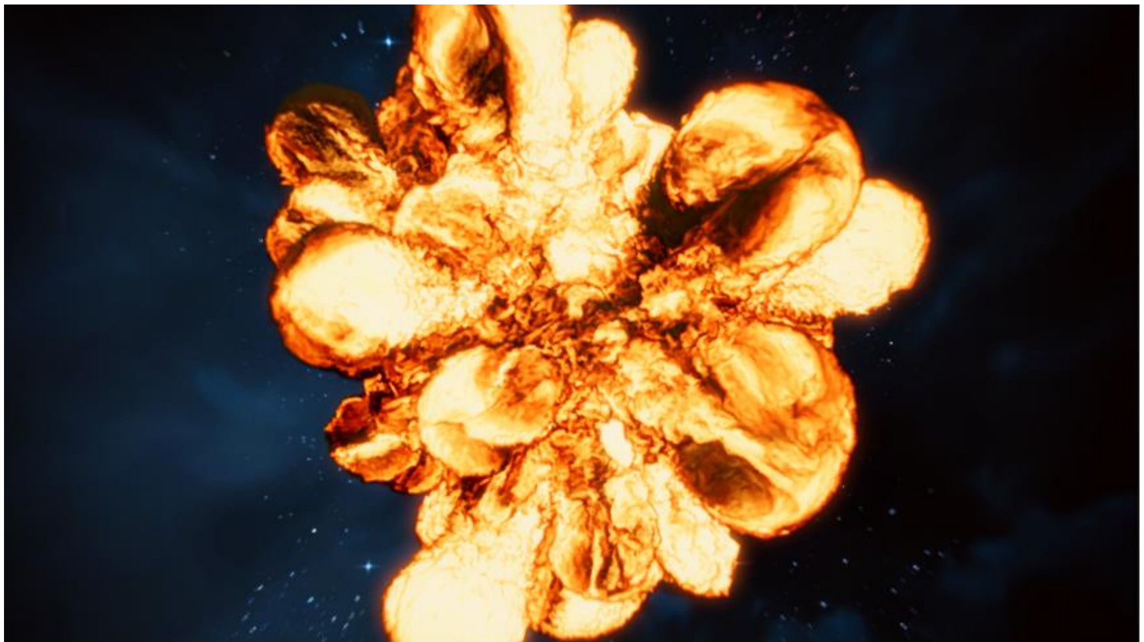

Figure 21. Planet exploding.

I put each of the sound effects' and music tracks' volume levels to half of the maximum and then either decreased or increased the volume as needed based on how they sounded in the game and how impactful the sound needed to be.

5.2 Juiciness

This is the time to add some more juiciness to the game. This can include things like adding more particle effects to an action, changing a camera filter when at low health, playing a sound effect or increasing scale of a text element when the player gains score.

One thing that was lacking juiciness was the effect when the player attains a new high score. (Figure 22) To address this issue, I tweened the text element's scale and gave it a more noticeable color when a new high score is made.

Figure 22. Default high score elements.

A library called DoTween is used for all the tween effects that are needed in the game. To tween the element you give DoTween an element you want to tween and then determine how you want to tween it. In Figure 23, can be seen how the high score text and numeral elements are tweened inside an IEnumerator and sound effects corresponding to the effects happening are played. Numeral element is scaled up, its numerals are scrambled, and color is changed to green in the TextTween method that is used elsewhere also. The text element that first reads as “HIGH SCORE” gets scrambled as well and its scale is brought up 1.5 of the original size and during the scramble the text is changed to “NEW HIGH SCORE”. Then, the text element is scaled back to the original scale but the score element is left at its tweened scale to be more noticeable.

```
private IEnumerator NewHighScoreTween(float tweenTime){
    calculationAudioSource.Play();
    TextTween(highScoreNumeralElement, score.ToString(), tweenTime);
    yield return new WaitForSeconds(tweenTime);
    calculationAudioSource.Stop();
    highScoreAudioSource.Play();
    highScoreTextElement.DOText("NEW HIGH SCORE:", tweenTime, true, ScrambleMode.All);
    highScoreTextElement.DOScale(1.5f, tweenTime);
    yield return new WaitForSeconds(tweenTime);
    highScoreAudioSource.Stop();
    highScoreTextElement.DOScale(1f, tweenTime);
}
```

Figure 23. NewHighScoreTween IEnumerator.

In the middle of the tween effect these high score elements look as in figure 24. Numeral text is already tweened and the new high score is being scrambled to show the new value after a certain time.

Figure 24. High score elements in the middle of the tween process

The boss encounter is one of the more exciting features in the game and it needs some juiciness to it. Camera shake is an easy way to add more impact to actions especially when something is being hit. When the player shoots at the boss and the projectile hits it, there is nothing happening; the boss just continues its routine like the hit does not matter. Adding a camera shake to a hit action gives the player feedback that a projectile has hit the boss and adds a nice visual effect. I also decided to tween the boss's health bar scale to 1.5 when a hit occurs to give the player even more feedback that the boss's health is decreasing with each hit.

5.3 Optimization

Optimization is an important part of game development to make the game run well on as many devices and setups as possible. Graphics and code optimization are the most impactful optimization targets. In Unity, there is a handy tool called *Profiler* that shows information about how much processing power is used and what uses it. (Figure 25)

Figure 25. Unity's Profiler tool window

Profiler shows CPU and memory usage and how much rendering impacts the game's performance. There are also some more minor profiling targets that are not examined here like audio and UI, because they have such a small impact in You Are the Light. Many of the readings in the figure num are higher than supposed to because of the editor overhead that Unity produces when using the profiler and the editor is running in the background. However, the overhead is shown when diving deeper into the readings, so it can be disregarded.

Looking at the profiler it seems that there is some garbage accumulation happening even though the projectiles are not instantiated and destroyed over time. There is still instantiation and destroying happening because most of the particles are used by instantiating them and destroying them at the end of the particle lifecycle. Particle spawning can be made more performant by pooling them the same way projectiles are pooled and only enabling and disabling them after the initial instantiation. This requires some changes to the code regarding pooling, like making a general abstract pooler class and then inheriting from that a projectile pooler and a particle pooler. Pool class can stay the way it is except for some naming generalization. This is such a minor issue that optimizing it is left for a later day in the future.

Framerate is looking stable in the profiler without big hits to it and the overall performance is good. Usually CPU usage spikes are caused by running heavy unoptimized Update() methods. You Are the Light is a small game and there has been an attempt to use as few as possible update methods to keep the performance good.

6 Publishing Process

The final step in making a game is to publish it. Where to publish depends on the purpose of the game. There are marketplaces that are more accessible to developers of all kinds and marketplaces that have fees and more strict processes of evaluating the game before it can be released to the public.

Companies of different sizes pick their publishing platforms differently. Big companies have more money to put into marketing and thus their games get more

sales and make it to the top of the charts more often, gaining more exposure than companies with lower budgets.

A developer must identify their needs for the game. Is the game supposed to be profitable? What is the game's audience like? What kind of cut is acceptable for the publishing platform company to take? Do you need easy to use deployment tools?

These are the most notable of the PC publishing platforms:

- Steam
- Itch.io
- Game Jolt
- Gog
- Humble Bundle
- Kongregate
- Gamers Gate
- Game House

Two of the more popular ones were picked for a closer examination, Steam and Itch.io. Steam is the market leader on PC game sales and Itch.io is a popular choice among indie developers.

6.1 Steam

Steam is the largest publishing platform for PC games (Figure 26). Steam used to have a program called Steam Greenlight that allowed developers to upload their games to Steam greenlight for $100. This fee was only to be paid once and after that you could upload as many games as you want to Steam Greenlight. You could not get the fee back. Then developers proceeded to make a page for their game and once the page was live, users could vote Yes, No or Maybe to express interest in the game. All the games were ranked by the yes votes and when the game broke into the top 5 on the list it was considered "greenlit" and got the permission to be released. (Perez, 2017.)

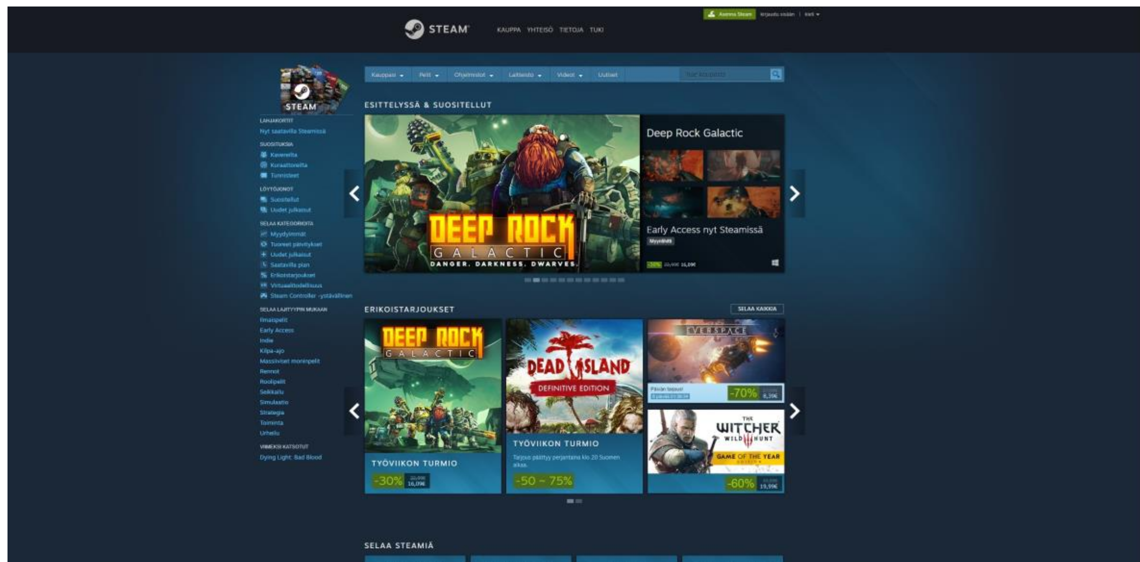

Figure 26. Steams storefront.

Nowadays, Steam has a new program for releasing a game called Steam Direct. Now developers need to pay $100 for each game they want to release, but now they can get it back if the game sells for over $1000 on Steam. After submitting the fee, a developer has to fill out their personal, tax and bank information. When the personal information is filled out, a developer can insert information about the game, such as its name, description and price. Now the game can be submitted for Valve's review process where they check if the game matches the description and that it is not malware. This process is supposed to take a few days. (Perez, 2017.)

After Steam Direct was launched the number of games on Steam has skyrocketed. 38% of all Steam games were released in 2016 alone (Perez, 2017). This makes it difficult to find the gems from the sea of games. When the number of games is this high it means that there is less exposure for all of the games, making it harder to get sales even if your game was good and polished. Steam takes a cut of 30% from the profits of a game.

6.2 Itch.io

Itch.io is a game hosting site where you can publish your games for free. The platform also has more customization options compared to Steam. (Figure 27) Company page can be modified by having a header image, color theme choices

and changing the order of the games that appear on the company page. You can also get access to the HTML by contacting Itch.io for even more freedom modifying the page. These options are also available for all the game pages you set up. (Fuller, 2017.)

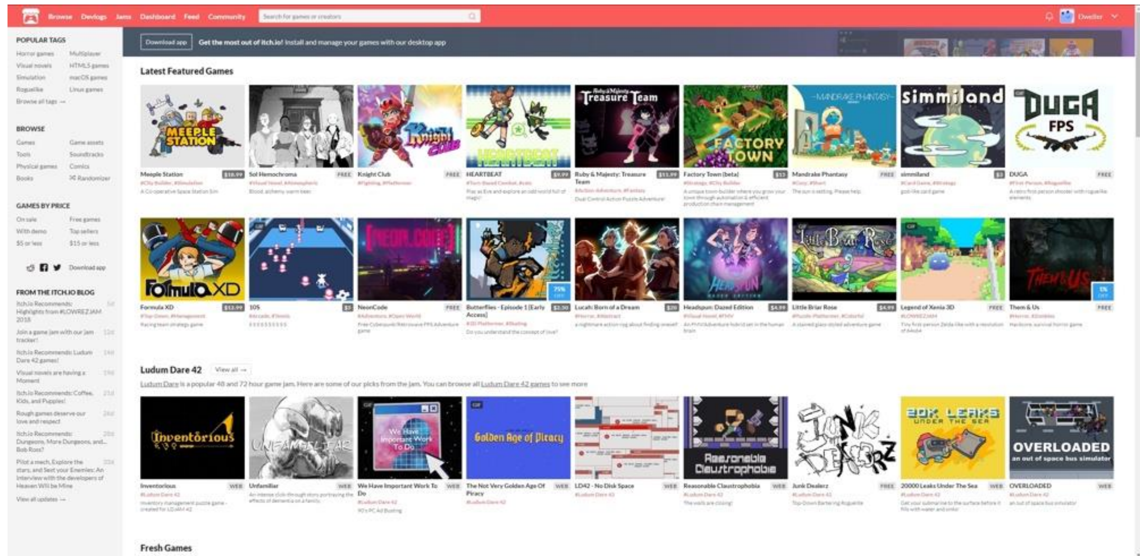

Figure 27. Itch.io storefront.

All the nice features do not keep bad games away from the platform, but the best-looking pages attract more people, so if a developer publishes a bad game, they have to actively make their pages look good to get views. (Fuller, 2017.)

Itch.io allows to set a price for your game but offers the option for consumers to pay more if they want to support a developer. A developer can keep all of the money coming in or decide to give Itch.io a cut of the profits.

6.3 Publishing to Itch.io

Publishing to Itch.io is easy. First you create an account to the service and then you can see your dashboard and create a new project. In the dashboard you can see a summary of all the views and downloads that have happened in twelve days. (Figure 28)

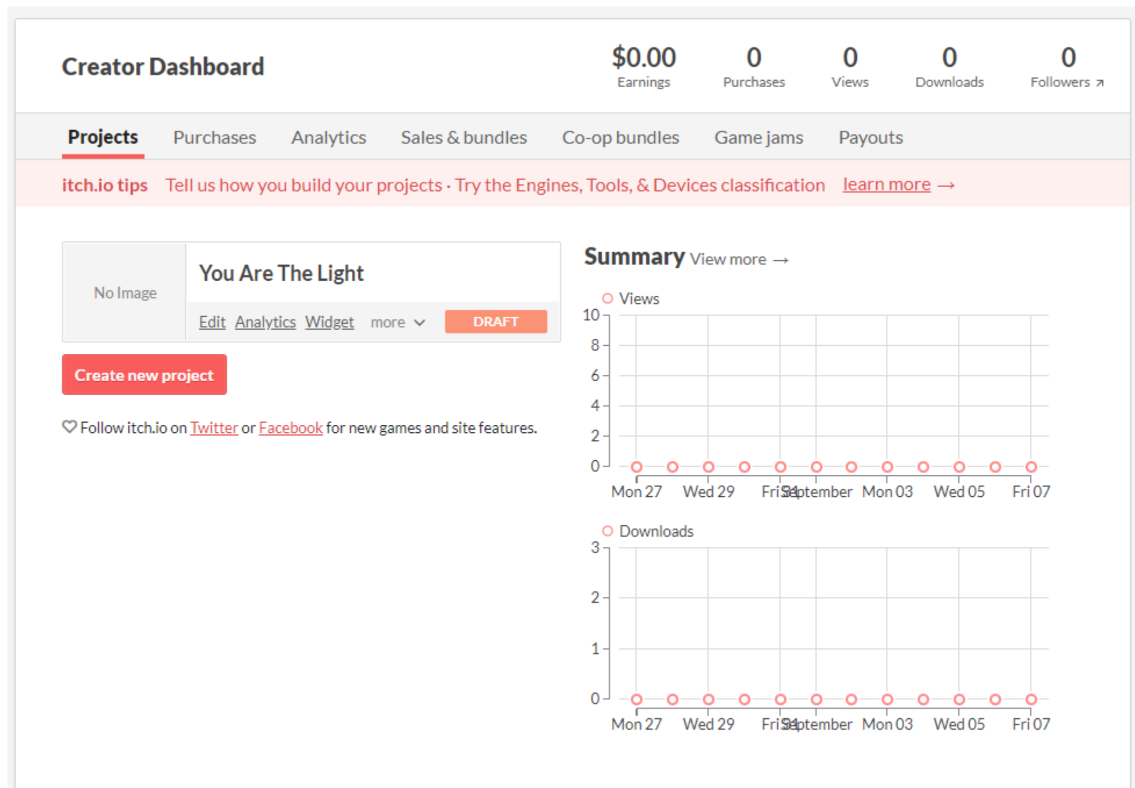

Figure 28. Itch.io dashboard

You can edit the project page extensively (Figure 29). Apart from setting up the basic information, like the title and description for the project, you can upload screenshots of the game, include tags, link to other publishing platforms where your game may reside, enable community features like comments or a full discussion board and change your pricing model.

For You Are the Light project there is a cover image added which shows on the right-hand side of the page when saved. The project is tagged with arcade, fast-paced and action tags and the description is filled with general information about the game. There is also a mention of upcoming features that were designed but did not get implemented yet. Pricing is set to allow no payments at all, not even donations. Game files are uploaded as zip-file, but you can upload files in a folder too if wanted. In figure 30 the final look of the game's page can be seen.

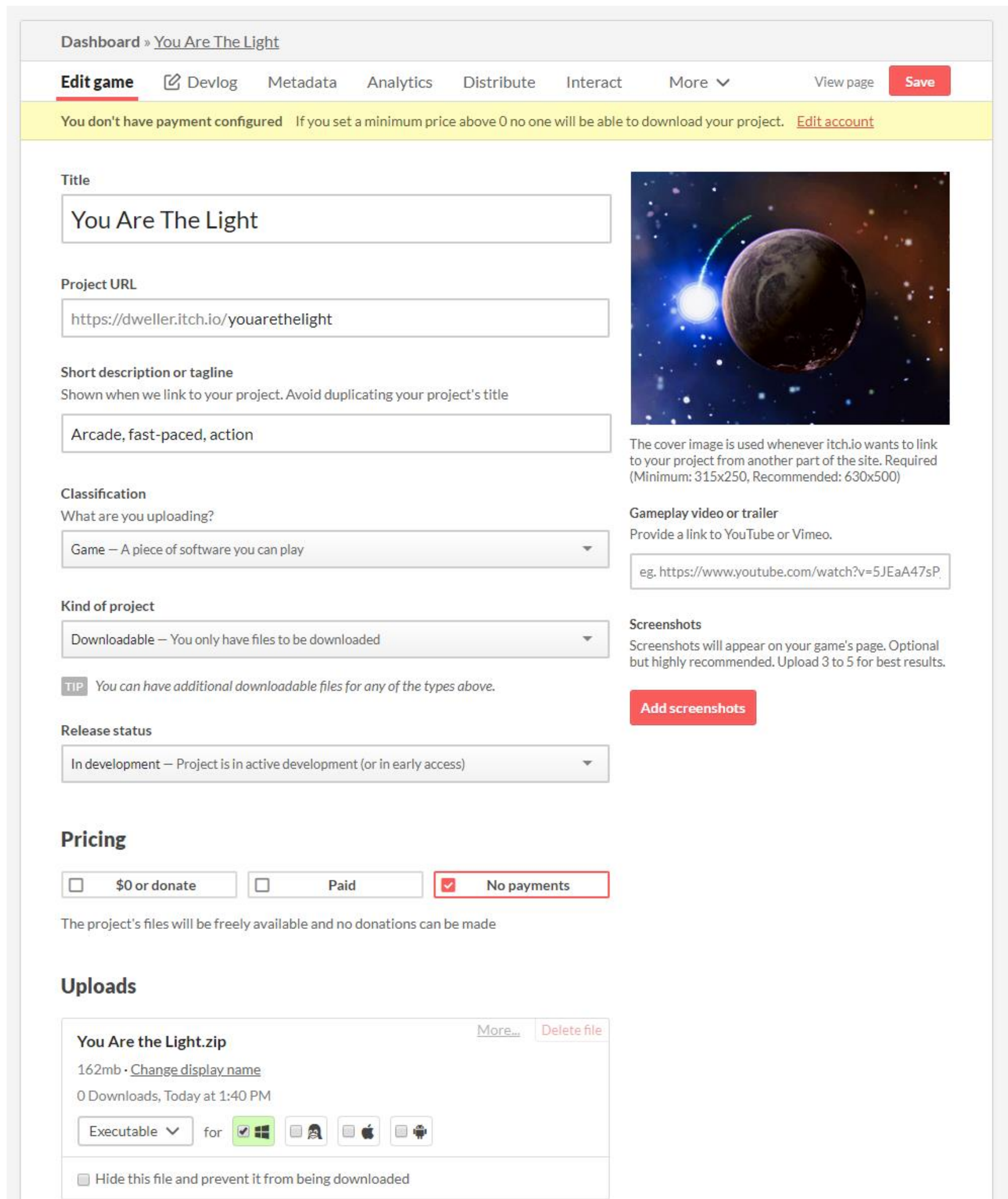

Figure 29. Projects edit page.

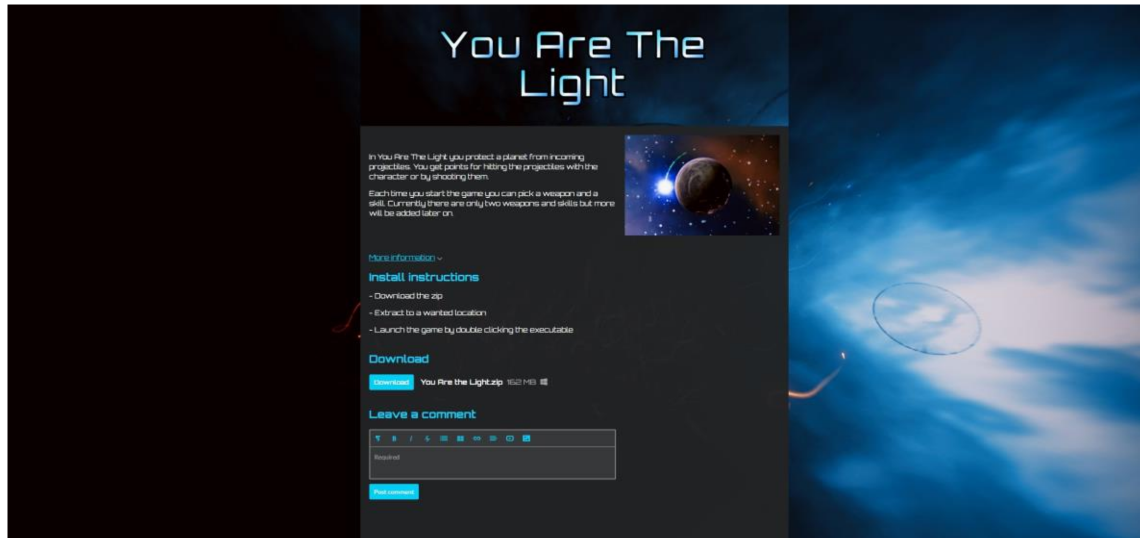

Figure 30. Final look of the games Itch.io page.

7 Results

Here is a summary of how the development of these two games was different from each other and which processes were similar, if any. As neither of these games have any notable story elements and no new mechanical marvels were used in the development, the results are centered on the development processes, visuals and features of these games.

Differences in development processes:

- Tinfoil Deflector's design was restricted by the theme 'Transmission'
- You Are the Light's design was restricted by Tinfoil Deflector's core gameplay design and the space theme.
- Overall the development processes of You Are the Light were more structured.
- More time and thought were put into You Are the Light's design and "looking through the lenses" method was used to great effect.
- Code structure and quality of You Are the Light is better, making further development easier in the future.

Differences in visuals and sounds:

- Tinfoil Deflector does not look good.
- You Are the Light looks polished and assets are of good quality.
- Tinfoil Deflector has only a few visual effects and they look and feel bad. Not enough feedback is given to the player.
- You Are the Light has a lot of visual effects that match the actions of the player and look and feel good. A lot of feedback is given to the player in form of various tween effects, sounds and particle effects.

Differences in features:

- Tinfoil Deflector has only a handful of features.
- You Are the Light retains the core features of Tinfoil Deflector like the circling around the planet and projectiles traveling towards it.
- You Are the Light adds many features to the game. There are bosses, skills and weapons. There is also quality of life features like options that are saved.

7.1 Development processes

The design part of Tinfoil Deflector's development went well, the scope of the game was reasonable, and the game was playable before it had to be uploaded to the game jam website. That is a great feat in itself for people participating to a game jam for the first time. Code, on the other hand, is full of dependencies that would make developing the game further a nightmare as there was no architectural design made. Polishing on the game is nowhere to be seen.

You Are the Light is designed with greater care. Different designs were carefully pondered and listed in Trello and there was research done on good design principles. Code architecture designed for You Are the Light makes further development possible because the code is structured well and is abstract and modular enough. Minor polishing was applied to You Are the Light before publishing the game.

Publishing processes were quite similar to each other. Both games' executables were uploaded to a website and general information about the games was written

there. Tinfoil Deflector’s site has a gameplay video linked to it, which You Are the Light does not have.

7.2 Visuals

Tinfoil Deflector’s visuals are bad (Figure 1). It was not possible to use paid Unity Asset Store assets on the project because assets are not to be distributed to people unless they work at the same company and share a workspace. Also, the game jam project and every project file, including graphical assets, have to be uploaded to a public repository at the end of the game jam, so everyone downloading the project would have gained access to the assets used. We resorted to downloading free assets from the Unity Asset Store that do not look that great.

You Are the Light looks much better than Tinfoil Deflector. The visuals are united and fit well with the space theme of the game. There is also a greater amount of visual feedback given to the player in the form of UI element tweens and particle effects. (Figure 31)

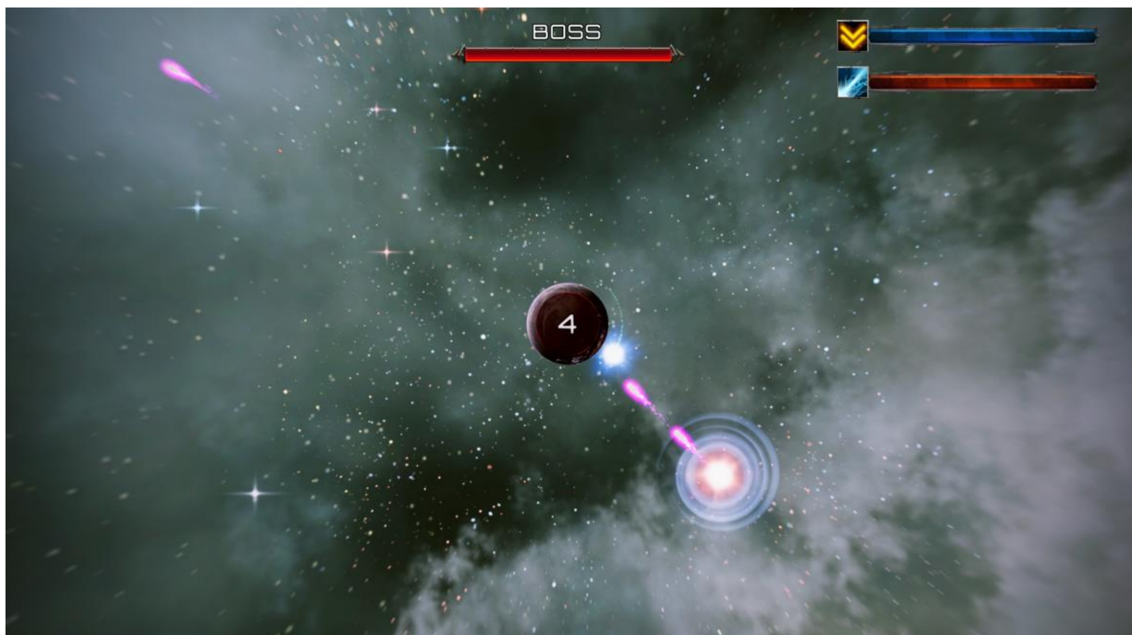

Figure 31. Final look of the game.

7.3 Features

Tinfoil Deflector has only a handful of features. The key features of the game, the two-camera system and beam transmission sounds, are quite unique but not fun. Having two cameras that divide the player's attention brings difficulty to the game at the cost of reduced enjoyment.

The two cameras and the transmission theme in You Are the Light were changed, but the core mechanics of protecting and circling around the planet stayed the same. Skills and weapons were added to the game to make more actions available for the player. Bosses were also introduced, making the game winnable when defeating the boss and giving something for the player to achieve. High score is saved in You Are the Light and getting score is no longer only dependent on the seconds survived.

Some of the designed features were cut from the final version of You Are the Light. Pulse gun was not implemented, Time Rewind skill also got deleted and leaderboard system was changed to a simpler high score system. Randomized boss behaviors were cut in the end too. These features were researched upon and designed and they will be implemented in the future and will be added to Itch.io when ready. Two of the cutout features would have had a great impact on the game if they would have made to the published version and those are the leaderboard and randomized boss behaviors.

Leaderboard would have brought the competition aspect to the game as the player would have tried to attain higher score than the other players have. With the high score system, the player now competes only against himself. If this was a real project at a real game studio there would have been made time to implement the leaderboard system before launching.

The randomized boss behaviors would have made the boss more interesting to fight against. Now there are only two kinds of actions, either circular movement with continuous shooting or teleportation with burst shooting.

8 Conclusion

Almost all the differences between the development of these two games boil down to one factor, time. When you have 48 hours to make a game there is only so much you can do. Designing features and code architecture properly is a key to a better game and that can be done well only if you have enough time at your disposal. The development of these games would have gone similarly if the team had been given more time on the game jam project.

The design method of looking at the game you are designing through different perspectives or “lenses” really helped to make a better product. Looking at your game from different perspectives that are laid out for you is a great way of making a better product. I would have never thought of some of the perspectives the lenses offered. It is hard to find as thorough guides to game design as the *Art of Game Design: A Book of Lenses (2008)*. I will use this method of design again in the future.

Visuals of You Are the Light are better than Tinfoil Deflector’s partly because of the paid assets used in the former. These paid assets are made by highly talented artists. The quality of the graphical assets is still not everything. What makes Tinfoil Deflector’s visual as horrible as they are, is the fact that they are not united in any way. A game can look good even if the assets do not, if the assets fit the theme of the game and look like they have come from the pen of the same artist. A lot of research was done to find suitable assets for You Are the Light. Maybe if we had used more time for Tinfoil Deflector, we could have found more unified assets for the game. Visual effects like particles, tweens and camera shakes looking good in You Are the Light is the result of time used tweaking them. Also, some external libraries were used, especially for the tweens, to get these kinds of results. Tweaking all the visual effects involves a lot of manual testing and tweaking. With every change of a variable you need to check if the visual effect looks and fits better than before.

I was a little disheartened about cutting the two-camera system and the transmission theme when designing You Are the Light, since these were the parts of the game that were at least somewhat unique. You Are the Light does not have

anything unique in it. It is a generic arcade shooter with a simple high score system. The two-camera system in Tinfoil Deflector divides the player’s attention too much and it is in the end just an artificial distraction and an annoyance. Although You Are the Light is not unique, it is a complete and polished experience and it takes some willpower to get through the boring parts of development where it seems that all the work is just the tiniest modifications of already existing features and you are not creating anything new.

If You Are the Light was to be a commercial product, a chapter about marketing would have been added to this thesis. Marketing is a huge topic and examining it could easily be a subject of a thesis on its own. Marketing includes subjects like gathering a community, managing the community and advertisement.

It is common to use designs and game mechanics from other games that are already proven to be good. When you think about it, every sequel of a game franchise is based on another, already developed game. For example, a new version of Grand Theft Auto (GTA) developed by Rockstar is an iteration of the previous ones. Rockstar might cut or add some features, change the location where the game is set in or improve the graphics and code quality. Of course, this is done in a much larger scale than it was done here. If thinking like this then You Are the Light could be called *Tinfoil Deflector 2* which it basically is.

If we are to examine games that are more refined versions of their game jam counterparts, Goat Simulator is a fantastic example. Goat Simulator is a crazy game where you play as a goat and wreck the world almost in a GTA style. Goat Simulator was not meant to be a commercial release, but the developers had so much fun playing it and the game got so much attention from consumers that they apparently had to release it. Goat Simulator was developed at a Coffee Stains Studios internal game jam and while not being exactly the same thing as developing a game at a Global Game Jam with strangers the experience is still comparable. (Wawro 2014.)

Boss behavior design was changed at the latest parts of the development. The original design being that boss would have one move set and one shoot set in use would have made the boss quite boring to fight against. The design was changed so that the boss has two move sets and two shoot sets in use during the same game, but this kind of functionality would have needed more features designed, developed and tested if it were to be randomized and there just was

not time for that. So, in the end the boss only ever has the same two move sets and shoot sets and the behavior is similar in each game.

I am somewhat disappointed that some of the features designed for You Are the Light had to be cut. Leaderboard system and randomized boss behaviors would have brought a lot of longevity to the game and upped the fun factor and competitiveness of the game. Not all hope is lost though, because of the code quality of You Are the Light is good enough allowing further development with relative ease. More skills and weapons can be added to the game easily as the weapons, skills and projectiles are abstracted. The code quality of the boss behaviors does not outright allow randomized behavior because the behaviors were written in a hurry. The code for the boss behaviors must be rewritten to attain better composition.

There was more time to develop this thesis game than we had at the Global Game Jam, but time was still an issue. Time was lost due to being unable to work because of a heat wave circling the city for three weeks. Sitting at a desk, looking at your screen for hours and hours with no air conditioning or a powerful cooling system proved to be impossible when the heat got hold of the apartment.

I am proud of the level of polish that the game has. During the development it was constantly kept in mind that the game is supposed to be polished. It had to look, sound and feel good and impactful and looking at the final product, it does all that. A great amount of time was spent on finding suitable particle effects and sound effects and then tweaking them. Tween effects of the UI elements were also carefully adjusted throughout the development.

Code quality is good in You Are the Light, but it could be better. There are some fixes to be made especially on the object pooling. The object pooling is dedicated to projectiles, but during the development it came apparent that pooling was needed for particle effects as well. There is of course always room for improvement, but overall the quality of the game is great and at some point, you just have to stop the development and decide when good enough is good enough and release the product.

Game jams are great events to get ideas for games because they often include group brainstorming sessions that produce more ideas than brainstorming on your own. Game jams more often than not are restricted by a time limit, so the developed games form out to be rather simplistic due to time limits. A simple

game concept can be developed further becoming more complex, but a game jam game suits better as a base for a smaller game rather than a large one.

References

Cosentino, N. 2013. Singletons: Why Are They Bad? Codeproject. https://www.codeproject.com/Articles/634723/Singletons-Why-Are-They-Bad 7.8.2018

Fuller, J. 2017. Why you should use Itch.io over Steam Direct. Gamasutra. https://www.gamasutra.com/blogs/JamesFuller/20170605/299317/Why_you_should_use_Itchio_over_Steam_Direct.php 30.8.2018

Izzo, S. 2018. Type-safe object pool for Unity. Gamasutra. https://www.gamasutra.com/blogs/SamIzzo/20180611/319671/Typesafe_object_pool_for_Unity.php 10.8.2018

Lacomb, J. 2017. What Is Scope Creep? And How Does It Impact Your Project Management? Workzone https://www.workzone.com/blog/what-is-scope-creep/ 6.6.2018

Machusak, E. 2011. The Actual Singleton Pattern. Gamasutra. https://www.gamasutra.com/blogs/EvanMachusak/20110228/89068/The_Actual_Singleton_Pattern.php 10.8.2018

Martin, R. 2009. Clean Code. Pearson Education, Inc. 17-19, 22, 25-26, 34, 37, 39-41, 55-58, 61, 63-64, 68-69.

Moran, D. 2016. 5 Leading Game Engines for indie game developers. Gamasutra.https://www.gamasutra.com/blogs/EvanMachusak/20110228/89068/The_Actual_Singleton_Pattern.php 30.8.2018

Perez, S. 2016. Steam Greenlight vs. Steam Direct: What indies need to know. Gamasutra.https://www.gamasutra.com/blogs/SheenaPerez/20170710/301248/Steam_Greenlight_vs_Steam_Direct_What_indies_need_to_know.php 30.8.2018

Schell, J. 2008. The Art of Game Design: A Book of Lenses. Morgan Kaufman Publishers. 21, 27, 32, 43, 140, 149, 153, 169, 189, 233.

Taylor, J. What is Rider? Codeshare. https://codeshare.co.uk/blog/what-is-jetbrains-rider/ 6.6.2018

Wawro, A. 2014. Q&A: The weird, wacky success that is Goat Simulator. Gamasutra.https://www.gamasutra.com/view/news/215628/QA_The_weird_wacky_success_that_is_Goat_Simulator.php 22.9.2018

Zoss, J. 2009. The Art Of Game Polish: Developers Speak. Gamasutra. https://www.gamasutra.com/view/feature/132611/the_art_of_game_polish_developers_.php 29.8.2018