Squirrels Gone Nuts: Adapting Software Engineering Principles to Solo Game Development

A Manuscript

Submitted to

the Department of Computer Science

and the Faculty of the

University of Wisconsin–La Crosse

La Crosse, Wisconsin

by

Logan Larson

in Partial Fulfillment of the

Requirements for the Degree of

Master of Software Engineering

March, 2024

Squirrels Gone Nuts: Adapting Software Engineering Principles to Solo Game Development

By Logan Larson

We recommend acceptance of this manuscript in partial fulfillment of this candidate's requirements for the degree of Master of Software Engineering in Computer Science. The candidate has completed the oral examination requirement of the capstone project for the degree.

Prof. Kenny Hunt Examination Committee Chairperson	Date
Prof. Rig Das Examination Committee Member	Date
Prof. Elliott Forbes Examination Committee Member	Date

Abstract

This manuscript describes the development of a 2D, multiplayer video game employing various software development techniques and project management strategies throughout the duration of the project. The game consists of short play sessions where players move their characters throughout uniquely designed levels with the goal of eliminating the other players in the game. This report explores the application of software engineering principles to the process of game development and the challenges that arise from blending art with engineering.

Acknowledgements

I would like to express my gratitude to my advisor, Kenny Hunt, whose guidance and support were invaluable throughout the development of "Squirrels Gone Nuts." Special thanks to the faculty and staff of Computer Science at UW-La Crosse, whose resources and encouragement were crucial to my development efforts. I would also like to acknowledge the contributions of the participants in the playtesting sessions. Their feedback was essential in refining the game mechanics and enhancing the overall player experience. Lastly, I wish to extend my thanks to my family and friends for their constant support and encouragement on this project.

Table of Contents

List of Figures

Glossary

2D (Two-Dimensional)

Refers to graphics and gameplay that occur on a flat plane, having length and width but no depth. In video games, this typically describes scenes or actions viewed from a side or top-down perspective.

Arena Shooter

A subgenre of shooter games characterized by confined play areas where players engage in fast-paced, shoot-out battles focusing on quick action and precision.

Entity Component System (ECS)

An architectural pattern used in game development that separates logic and data, enabling flexible handling of game objects and behaviors without the inheritance complexities of traditional object-oriented programming.

Scrum

An agile development methodology where work is broken into sprints, or short phases of work, followed by review and adjustment. It emphasizes iterative progress through collaboration.

Evolutionary Prototyping

A model in software development where an initial prototype is evolved through iterative modifications and improvements based on user feedback and testing.

Kanban

A method for managing knowledge work with an emphasis on just-in-time delivery while not overloading the team members. In software development, it involves visual management with cards and boards to track work progress.

Matchmaking System

A system in multiplayer online games that groups players into sessions based on their skill level or other criteria to ensure fair and balanced gameplay.

MMR (Matchmaking Rating)

A numerical value assigned to each player based on their gameplay performance. It is used in competitive gaming to match players of similar skill levels.

User Story

A feature of Agile software development, describing functionality from an end-user's perspective. It helps developers to focus on delivering value based on user needs.

Lag Compensation

Techniques employed in online gaming to minimize or counteract the effects of lag or latency, thus ensuring that player actions are responsive and consistent despite delays.

Client-Side Prediction

A technique used in networked computer games where the game client predicts the results of certain actions without waiting for server confirmation, to make the game appear more responsive.

Server Reconciliation

A network programming technique used in multiplayer video games to ensure that the server's authoritative game state resolves conflicts between multiple clients' states. It corrects discrepancies between a client's predicted state and the server's calculated state, ensuring consistent gameplay across all players.

Entity Interpolation

A method used in real-time multiplayer games to smoothly render the movement of objects or entities between updates received from a server. It interpolates positions and states based on previous and current data to reduce the jarring effects of network latency.

Polish Criteria

Elements of game development that enhance the aesthetic and sensory appeal of a game, which are not essential for basic functionality but are crucial for enriching the player experience.

1. Introduction

1.1. Project Inspiration and Overview

"Squirrels Gone Nuts", is a video game where you play as a squirrel and shoot other squirrels with nut-themed weapons. I can take a guess at what you're thinking: "There's no way this kid did his Software Engineering Master's capstone project on a video game about squirrels with guns." However, I invite you on the journey behind "Squirrels Gone Nuts", a project that blends inventive game design with the principles of software engineering.

1.1.1. Inspiration

Before stepping foot on UW - La Crosse's campus my freshman year, I knew I wanted to make a video game for this capstone project. I had been making games since I was around 10 years old, and I saw the MSE program as an opportunity to build and finish a long term project. Throughout college however, I let those dreams fall to the wayside as I grew more interested in web development among other areas of computer science. At one point, I even planned on graduating early with just my Bachelor's because I was not enjoying classes (mainly due to COVID). But, I stuck with it and before I knew it, my capstone project proposal was due. I had a few project ideas floating around, but that urge to make a video game found its way back into my head. One fall day, I was walking back from class observing the squirrels running around on campus, and thought, "Wouldn't it be fun to run around as one of them?" Thus, "Squirrels Gone Nuts" was born.

1.1.2. Overview

"Squirrels Gone Nuts" is a 2D, multiplayer, arena shooter where you play as a squirrel that can traverse every visible surface in the game. Let's break that down. '2D' is the view from which this game is observed. The specific version of a 2D view is a side-on view, commonly called a side-scroller. Similar to games like Super Mario Bros and Jetpack Joyride. The game features online multiplayer, so players from around the world can connect and play with each other. 'Arena shooter' is the genre of the game, and it refers to a few key mechanics and design decisions that set arena shooters apart from other shooter genres. Typically an arena shooter consists of a small amount of players (4-12) fighting in an enclosed area with the focus on fast-paced movement and precise shooting. The character controlled by the player is a squirrel which offers a unique perspective on platformer movement in a crowded market of standard platform movement controllers.

1.2. Project Objectives

There were two categories of objectives for this project. First being the tangible product-based objectives pertaining to the game itself, and second being applying software engineering principles and life cycle development models to independent game development.

1.2.1. Game Objectives

For the objectives of the game itself, I dreamed and set many lofty goals that are on par with the top competitors in the market. What I failed to realize was there were teams of 50+ people working on those games, and I am a one man team. Suffice to say, I over scoped the game which is very common in the indie developer scene. As much as it hurts my ego to say, the game was over scoped both in terms of complexity and timeline. When thinking of the games features, I assumed I had more experience and understanding of how those features could be implemented than I actually did. In reality, each step of this journey met me with unforeseen challenges and resulted in many deadline extensions. The Dunning-Kruger effect was in full effect throughout most of this project, but maybe that is the reason the game is where it is today. Had I not believed I was capable of implementing the features I wanted in the game, I probably would have settled for a much simpler and less fun game.

1.2.2. Software Engineering Objectives

While the development of the game itself was my primary objective for the project, I also wanted to determine how software engineering principles could be applied to the game development process especially as a solo developer. Throughout the project, I iterated through and adapted many different software life cycle models.

Successfully integrating software engineering principles into game development proved to be a significant achievement. That is not to say development was consistent and smooth throughout. I went through many iterations of life cycle models until I landed on one that complemented my workflow and the project's needs. Exploring various software development models turned out to be crucial in overcoming the technical challenges faced at different stages of the project. Each part of the game's development called for a unique approach, and being able to adapt the life cycle model improved the architecture and the velocity of the project.

In Section 2., Software Development Processes, I'll discuss the advantages and trade-offs of each model I examined. I'll also share the adjustments I implemented in developing my current software process.

1.3. Relevance and Need

The arena shooter genre, once at the pinnacle of gaming with classics like "Quake" and "Doom" has seen a noticeable dip in its popularity. This downturn, I argue, isn't due to a dwindling player base but rather a stagnation in innovation. The early successes set a high standard, yet as the industry evolved, many developers shifted their focus to broader markets, leaving arena shooters in a state of creative limbo. Veteran players dominate the arena due to the steep learning curve for the movement mechanics. This has created an unwelcoming environment for newcomers slowly diminishing the player base of newer titles.

1.3.1. Innovation

With "Squirrels Gone Nuts," my ambition was to tackle the high barrier to entry head-on while ensuring the game remained rewarding and engaging for both newcomers and veterans.

The key to this was innovation, particularly through the game's movement system and mechanics, ensuring everyone starts on equal footing but with enough depth to keep gameplay interesting.

Originally I envisioned "Squirrels Gone Nuts" as a 3D third-person shooter, but a pivotal discussion led to a shift towards a 2D platformer. This decision, influenced by my experience level and desire for innovation, allowed for a more focused approach to game mechanics, particularly in how players move and interact within the game's environment.

1.3.2. Redefining Movement

With the change of perspective, I had to shift the design of the movement mechanics from a something that mimicked a squirrels movement to a stylized and simplified version in two dimensions. Due to the nature of a squirrel's movement incorporating both vertical and horizontal movements from climbing trees to running on the ground. I knew the perspective needed to allow for this freedom of movement which is why I decided to use the side-on perspective.

Throughout my game development journey, I have been interested in unique camera movements to enhance immersion. So, I thought of a simple yet impactful camera movement for the player. The camera matches the player's orientation. When the player runs up a wall, the camera rotates to follow the player's rotation to make it look as though the wall has become the ground. This helps the player orient themselves better in the world for the same reason as it is easier to accurately drive a real car vs a remote control car. This sort of camera movement is common for 2D top-down games, but, to my knowledge, is the first of its kind for a side-on game.

In real-life, squirrels predominantly move around on all fours, with the exception of when they are using their front two legs to eat. I took this idea and adapted it into my game by allowing player's to traverse the world in different modes. The first two modes were sprinting and shooting. In sprint mode, the player is quadrupedal and able to move fast, but they cannot fire their weapon. In shoot mode, the player stands bipedal and moves slower, but the player equips their current weapon and is able to shoot the other players with said weapon. As the project progressed, I added in another mode that has little connection to real life squirrel movement, sliding. When transitioning into slide mode, the player retains the momentum from the previous mode. While sliding, movement input is locked, meaning the player can't directly influence the speed or direction of the character. However, sliding is physics based, so if the player is on an incline, they can gain momentum as they slide down the incline. The combination of these different movement modes creates a unique system that requires quick decision making based on the trade offs of each mode.

Another integral mechanic to the movement system is jumping. This is how the player can change which surface they are connected to. When the jump input is pressed, the player leaves the current surface they are standing on and follows a trajectory based on the player's momentum and the world's gravity. After leaving the ground, a landing spot is predicted and the player's character rotates to match the surface of the landing spot.

The final movement mechanic is an airborne mechanic that relies on the weapon shooting system. While airborne, the player is given the option to switch to shooting mode and fire their weapon. When this occurs, a force is applied to the player in the opposite direction

of the shot, a recoil force. This recoil force varies by weapon type and allows for interesting and emergent player movement behavior. A side-effect of this mechanic grants the player the ability to fly by shooting towards the direction of descent.

1.4. Expectations: Game and Report

1.4.1. Market Expectations

"Squirrels Gone Nuts" stands as a symbol of innovation within the arena shooter genre, yet I approach its release with measured expectations. In today's competitive market, I am realistic about the game's commercial prospects with hopes of niche success. The primary goal was never to dominate the charts but to innovate within a genre that has significantly shaped my gaming experience.

The journey and development of "Squirrels Gone Nuts" has been a great learning experience, providing me with skills and knowledge that extend far beyond this project. This endeavor has reignited my passion for pushing the envelope in game design and mechanics within the arena shooter genre. Although "Squirrels Gone Nuts" may not revolutionize the genre, it represents a significant step towards future innovations that might.

1.4.2. Transitioning to Technical Insights

With the foundation laid and expectations set for "Squirrels Gone Nuts," the report now transitions into a detailed examination of the technical journey behind the game. The upcoming sections are crafted to provide an overview of the methodologies, challenges, and solutions encountered during the development process.

This report aims to dissect the process of intersecting software development processes with game development. Offering a detailed account that balances technical rigor with the practical realities of bringing a creative vision to life.

2. Software Development Process

2.1. Introduction to Software Life Cycle Models

The software development process is understood and documented through the concept of software life cycle models. Software life cycle models provide a guide for the parties involved in creating a software product. Coordinating people, events and timelines is handled by the selection of a life cycle model. Each model has a different approach for achieving the end result of functional software. There are many pros and cons to each approach depending on a multitude of factors ranging from team size to project requirements.

2.2. Criteria for Choosing a Life Cycle Model

Throughout the course of this project I gathered an understanding of what criteria are necessary for the choosing of a life cycle model. The following are some of the reasons I deem important when selecting a life cycle model in the context of solo game development.

2.2.1. Gathering Requirements

The process of developing a game, while similar, has many differences from developing software. In software development, typically there is a client with specific needs. Therefore, there can be rigidity in when the development team meets with the client to gather requirements. As for game development, the client is the player and, more often than not, the needs of the player are far more abstract than a client's. The player has an idea of what he or she wants but it is up to the designer and developers to figure out what they want and how to implement it. So, it is not as easy as setting up a meeting between players and developers and gathering the requirements for the game.

2.2.2. Art and Engineering

Game development is the blend of art and engineering, so the final product needs to be functionally correct while also being aesthetically pleasing to the player. Customer facing software applications also have the requirement of being visual appealing, but to a much lower degree. Software solves a functional need for a user while a game solely relies on its engagement and immersion which requires more aesthetic appeal. While I know that wasn't entirely necessary for this capstone project, I still had goals of building an aesthetically pleasing game. Therefore, I needed to factor this into the selection of a life cycle model.

2.2.3. Finding the Fun

Something unique to game development, in regards to picking the life cycle model, is the concept referred to as "finding the fun". Video games are entertainment and, as mentioned before, are not required to be functionally useful in the same way as software products are. Therefore a video game development life cycle requires flexibility to "find the fun". Let me define "finding the fun". It is an abstract concept designers and developers use to describe the gut feeling they have when implementing a mechanic, and want to change or add to it

because they think it will make the game more fun. For example, lets say in the original design document, the game is supposed to have a jump mechanic. The developer goes ahead and implements the jump mechanic. Then the developer tries adding a double jump into the game because they think it will be more fun for the player. This is the developer expressing his gut feeling of finding the fun. The life cycle I chose for my game needed to have the flexibility for me to "find the fun". I only started to realize this about halfway through development which is why I changed life cycle models.

2.2.4. Solo Development

The final major influence on which life cycle model I chose was that I developed the project on my own. There are many aspects of development that become easier as a solo developer. Team coordination is not required, no roadblocks from waiting on team members to finish work, and creative decisions are resolved much faster. But, other aspects become more difficult like the lack of accountability and brainstorming solutions to difficult problems. Throughout the course of the project, I realized that building in accountability mechanisms helps tremendously when developing a game by yourself. As for the brainstorming difficulties, I regularly participate in Discord servers which is basically a network of game developers just like me that are willing to help me solve bugs and brainstorm ideas for the game.

2.3. Journey through Different Life Cycle Models

I wish I could say I picked the perfect life cycle model for this project and I had no issues with it during development, but with most things in life this was not the case. I was constantly evaluating and adapting the life cycle models to fit my needs. Since the duration of the project was so long, I was able to run experiments for how well a life cycle model was working for me. Then I was able to adapt the model to remedy its weaknesses and play to its strengths. In some cases, I changed life cycle models entirely.

2.3.1. Evolutionary Prototyping Model

In the beginning of the project, I did not consider many models because I did not know what to look for. As previously mentioned, the criteria discussed above was discovered throughout the course of the project.

In my mind, the game was easily divided into sections of related concepts: movement, shooting, multiplayer, etc. So, I went looking for a model that fit that idea of building upon each previous concept, and I found the evolutionary prototyping model. And, I only got as far as the name because I completely misinterpreted the process of this life cycle model. Right from the start, I adapted this model into my own version of a life cycle model. I was under the assumption that the evolutionary prototyping model broke the project down into distinct sections and then each section was developed as a mini-waterfall. I realize now that is not the intended use of the model, but I can't go back now. Here is the journal entry outlining the process I followed for the first two evolutions of development.

Software Lifecycle Model

Wednesday, April 27, 2022, 1:16 PM

As of the day I got my project approved (4/27/2022), I am thinking about using the Evolutionary prototyping model.

I chose this model because the game can be divided into many distinct "evolutions": movement, combat, multiplayer, etc.

Each evolution will follow this rough outline:

1. Requirements & Specification
   (a) Define features and systems as requirements
   (b) Organize the requirements
   (c) Expand requirements details
   (d) Write formal requirements document
   (e) Estimate time needed for implementation
2. Design
   (a) Divide system into subsystems and decide how subsystems will interact
   (b) User interface design (if applicable)
   (c) Database design (if applicable)
   (d) Player input design (if applicable)
3. Modeling
   (a) Architectural modeling - design patterns, component diagrams
   (b) Structural modeling - class diagrams, ERDs
   (c) Behavioral modeling - behavioral class diagrams, DFDs, Petri nets
4. Implementation
   (a) Just code
5. Testing
   (a) Play test with friends
   (b) Fix bugs

While this may not be a formal specification of a life cycle model, I do believe this model helped me tremendously in the early stages of development. To me, it seems foolish to start implementing and iterating right away without any designing or modeling. Especially for a project with such a large scope it is critical to set a proper foundation. A project that has a clearly defined architecture is much easier to extend and modify later in development rather than systems that are built on a week-to-week basis. Maybe this is just my opinion or I don't apply my software engineering skills well enough in real-time. There is a similar phenomenon in writing approaches: outline writing vs discovery writing. And personally, I like setting the outline for the project and filling in the gaps rather than discovering the gaps and filling them in real-time.

Another benefit of this approach is reducing the scope of the project to each "evolution"

instead of the project as a whole. When starting a project of large scope, it is natural to get overwhelmed with everything that needs to get done. By breaking the project up into distinct evolutions, I was able to manage the overwhelm and focus on a single section of the project at a time. Somewhat like a modular development workflow where each section of the application is built in solitude and only towards the end of development are the modules integrated. But with this project, the components built in one evolution tended to have dependencies on the previous so it was more like the project was evolving rather than being built in modules.

At the start, I was only using Microsoft OneNote to manage the project. Later, I expanded to using Notion alongside OneNote. OneNote is a virtual note taking application that provides a hierarchy of sections and pages for organizing ideas. The general structure of OneNote is a user has multiple notebooks, each notebook has multiple sections, and each section has multiple pages. In the case of my project, there is one notebook for the entire project, and each evolution of the project has a separate section. Each section contains pages related to each stage of the evolutionary prototyping process.

Here are some artifacts from the first two evolutions of the game: *Movement* and *State Management and Synchronization*.

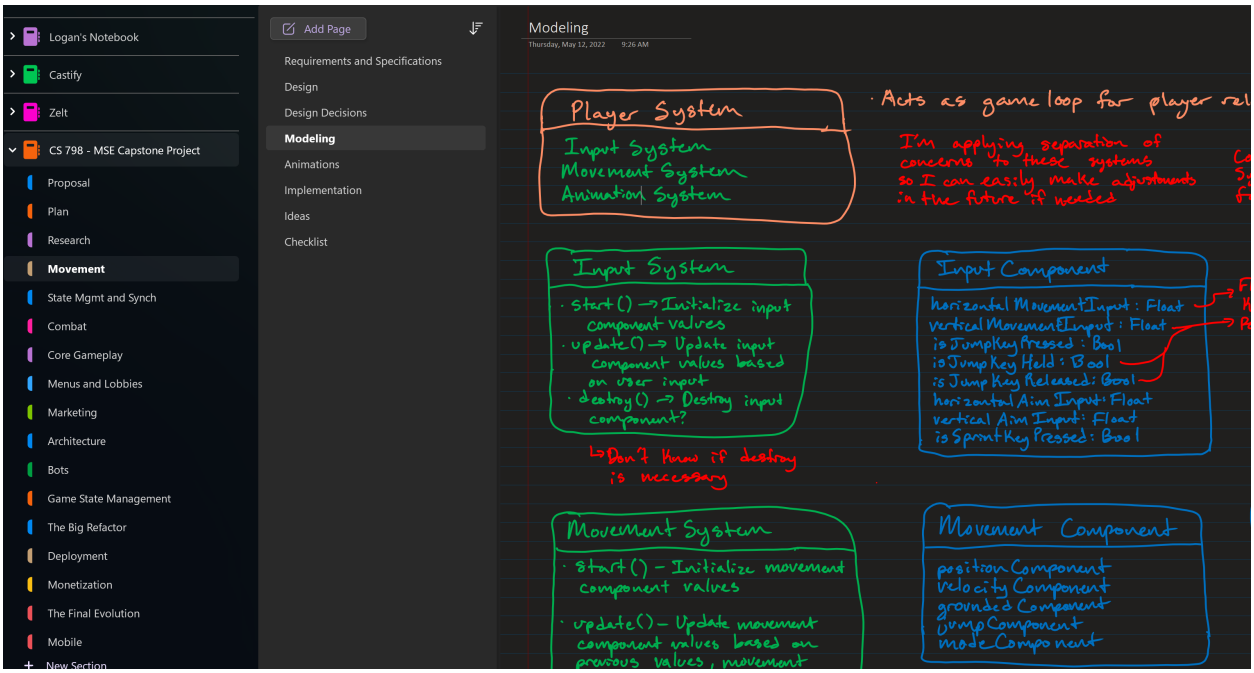


Figure 1. Movement evolution modeling page.

Initially I had planned for the state management and synchronization evolution to be implemented towards the end of development. But, I heeded the advice from the many game developers that came before me and implemented it early on in development. This ended up being a great decision because I had grossly underestimated the amount of effort required to implement multiplayer. It would have been a massive undertaking to retrofit multiplayer capabilities into the game late in development.

At the start of the *State management and Synchronization* evolution, I began writing

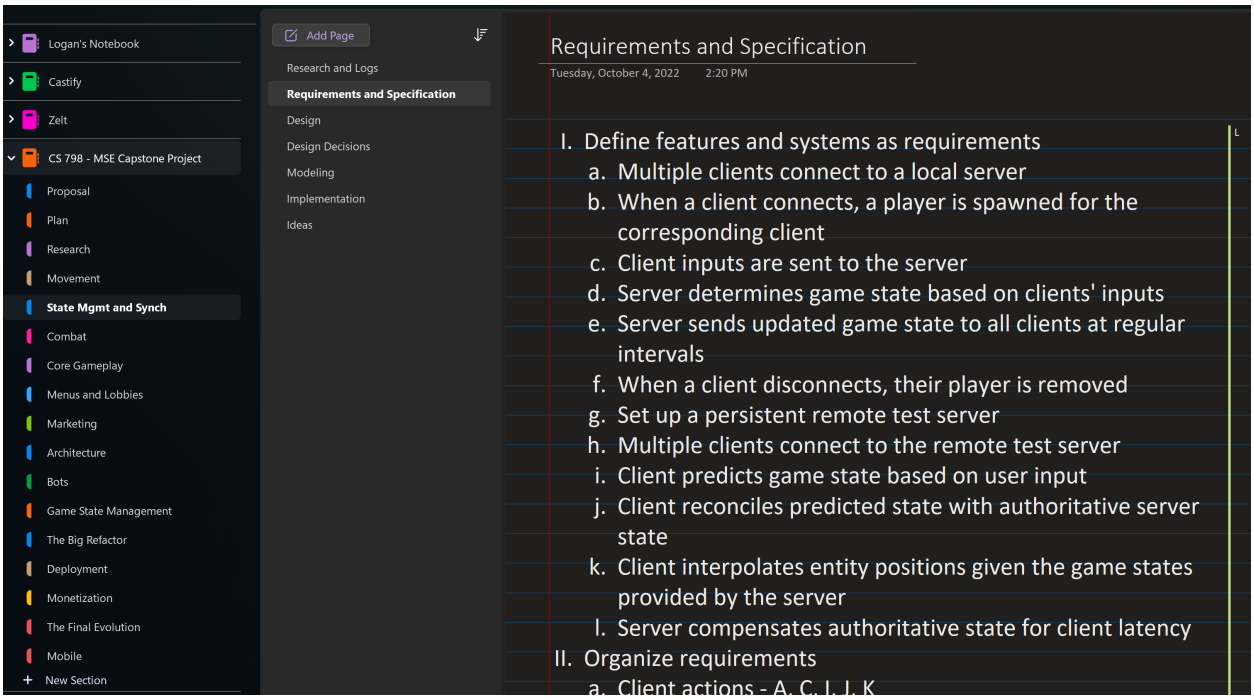


Figure 2. State management and synchronization evolution requirements and specifications page.

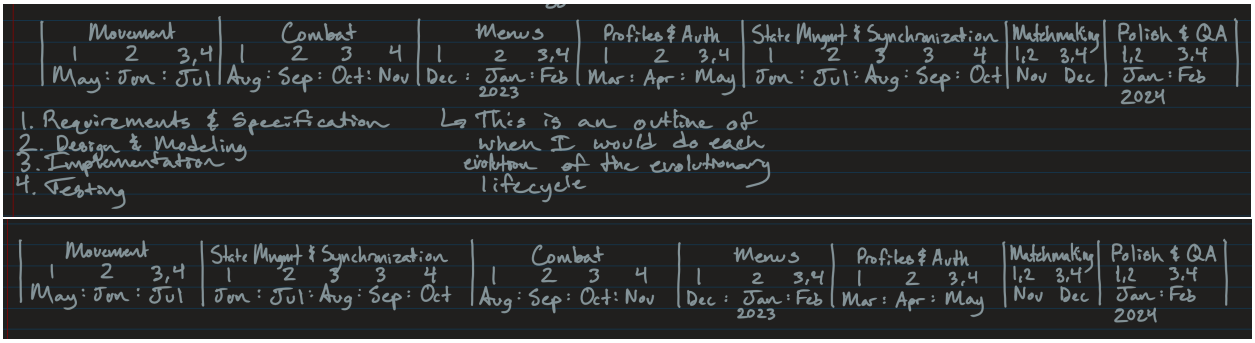


Figure 3. Project timeline estimation before starting the project and after the first two evolutions.

development logs. These acted as recaps of the work accomplished for the day and plans for moving forward. Although these are not required by any certain life cycle model, I think they are invaluable for recording the progress of a project and documenting growth as a developer. This practice also makes re-acclimating to the project much easier. For example, there would be stretches of time where I didn't touch the project, but by documenting the progress daily, I was able to jump right back into development rather than needing to spend time figuring out where I left off.

As for improvements to the modified evolutionary prototyping model, a simple improvement would be to allow changes to the requirements, design, and models during the implementation process. In the planning periods, I would develop a big grandiose plan for how the software will seamlessly fit together, but as soon as I started implementing the plan, I would realize I didn't account for something and now the whole design needs revisiting. So I think for a process like this to be very effective the models and designs need to be *live* documents with version control just like code. The version history is key to understanding timelines and how changes in the models coincide with changes in the code. By the end of each evolution, so much had changed from designs to implementation, and the models no longer reflected the current state of the project.

Something that would have made this process easier is making the designs and models more easily extensible and modifiable. I drew the designs in OneNote with my stylus because that is how I like to brainstorm. But, for future reference I will move the models to a tool that allows for easier modification and tracking of changes.

Another issue I found with this model was the lack of accountability. Since the deadlines were rather loose I found myself slacking off because I had given myself such a large buffer for the work. I only noticed this after I moved to the scrum approach because that approach has consistent deadlines and expectations of work done by those deadlines. So I think building in accountability mechanisms for this life cycle such as shorter timescale work expected deadlines would be another good improvement to the model.

I will take a page from agile here and say there should have been a larger evaluation stage of the process at the end of the evolution. I should have taken a moment and reflected on the process to see where improvements could be made rather than assuming it was working the best it could and moving to the next evolution. I think this is partially the reason why I ended up switching models entirely after the second evolution.

2.3.2. Evolutionary Scrum Model

Rather than recalling from over a year ago the reasons why I switched life cycle models, here is the journal entry explaining why I switched life cycle models.

Switching Project Management Models
Wednesday, February 8, 2023, 9:13 PM

> For the first two evolutions of the project, *Movement* and *State Management and Synchronization*, I used a standard evolutionary prototyping model. Where each evolution consisted of a mini waterfall cycle. This proved to be

really helpful for the movement evolution because it was also the evolution it which I needed to design the project structure. This also worked well in the state management and synchronization evolution because there was a good deal of designing and modeling that needed to take place before I started implementing the system. However, I have now come to the point where the bulk of the designing and modeling is complete, so I am proposing a switch from *waterfall evolutions* to *scrum evolutions*. I believe moving to this model aligns better with the current state of the project and how development is likely to proceed on the project. Another reason for switching is I am currently in Software Project Management where we are learning about the scrum model and I am starting to grasp the benefits the model provides.

For each evolution I have a set time frame I would like to complete the evolution in. For example, I would like the *Combat* evolution to take 4 months. Each sprint lasts 2 weeks. So, the *Combat* evolution will last 8 sprints.

A typical *scrum evolution* will go as follows.

At the beginning of the evolution, I will generate a list all of the prospective user stories for that evolution and their story point values. I then total up all the story points and divide them by the number of sprints in that evolution minus one. I like to leave one buffer sprint at the end for unforeseen challenges and uncertainty. For example, the total story points for the *Combat* evolution is 75. With there being 7 working sprints, I should aim to complete 10-11 story points worth of user stories each sprint to keep project velocity constant.

Each sprint will start by prioritizing and grooming the backlog. I will use the MoSCoW method for prioritizing. With that, I select user stories for the current sprint and the next sprint. By selecting user stories for the next sprint, I give myself a chance to work ahead on relatively important and high priority tasks. Then for the user stories in the current sprint backlog, I expand them into sprint tasks and test cases. A user story has acceptance criteria that defines when the user story is considered done. After all that setup, I implement the sprint tasks. After the implementation, I demo the product to Prof. Hunt for review.

This life cycle model proved to be very effective in producing consistent progress. The main perceived benefit of this model was the accountability. Every two weeks, I had a self-imposed due date for features or bug fixes. This seems to improve the consistency of progress with a small compromise for the architecture's robustness. I say this because there is no longer a large focus of development effort on the planning of the project's architecture. Instead the focus is on implementing and iterating to arrive at a functional product. Figure 4 displays a portion of the initial planning process for the evolution. This list was revisited every sprint planning session: adding, modifying, and removing user stories.

Figure 5 shows the process I used to plan the sprints using the MoSCoW method. The top number in the column header is the sprints corresponding to each of the MoSCoW categories. The other number in the column header in parentheses is the number of story

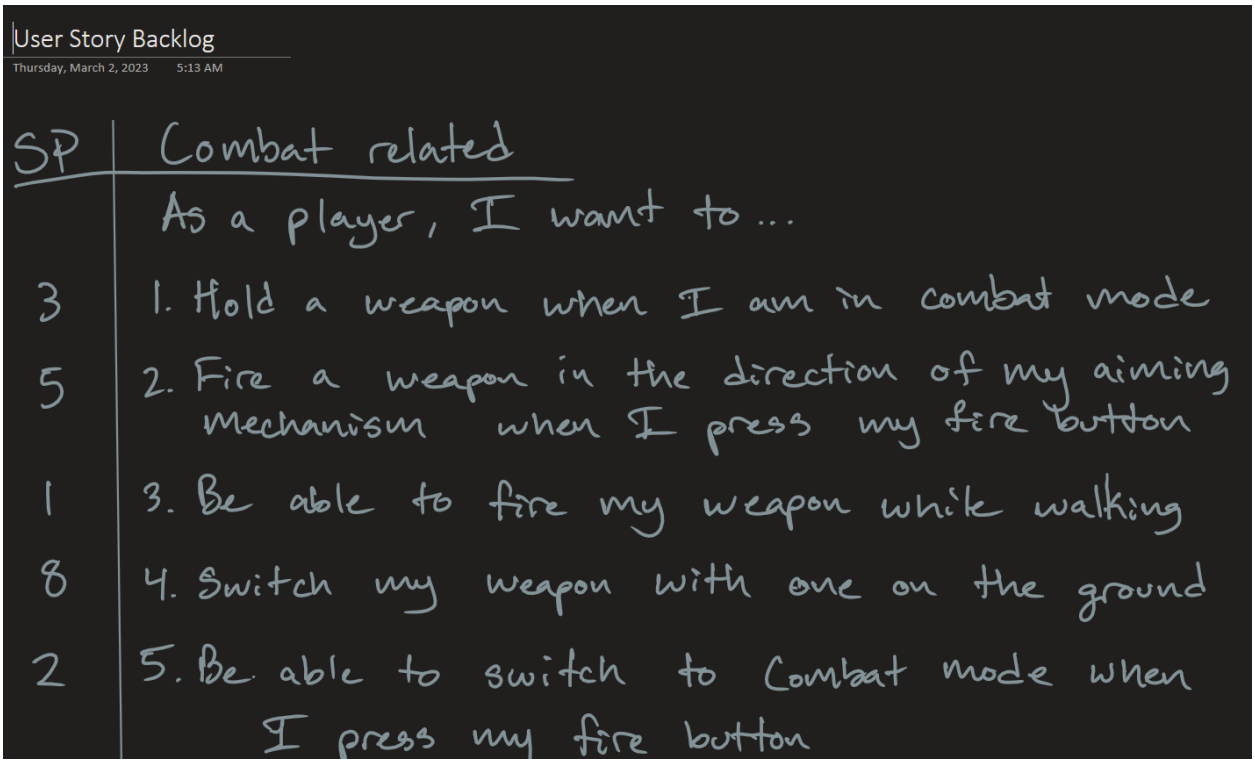


Figure 4. User Story Backlog for Combat Evolution.

points for all the user stories in the column. The numbers in the column bodys correlate with numbered user stories in the evolution's user story backlog.

The final document used to plan the sprints, Figure 6 was something I called tasks and tests. These represented the steps to take when implementing the user stories. Also they provided acceptance criteria for me to know when a user story was fulfilled enough to move on. I also added another category of acceptance criteria called polish criteria. These were not essential to building a functional product, but were things that would improve game feel and overall polish to the game.

During this time, I introduced Notion to the project management tools list. Notion provided me with a extensible Kanban board. The board view provided clear oversight to the project, and I was able to gauge the project's velocity much easier. My version of the Kanban board contained seven different columns: evolution backlog, next sprint backlog, current sprint backlog, in progress, testing, polishing, and done. Cards would originate in the evolution backlog after the initial evolution planning session. Then proceed to move along the board based on current state of the card relative to its acceptance and polish criteria. Figure 7 displays the details contained in a card on the *Combat* evolution's Kanban board.

I used this approach for the next two evolutions of the game: *Combat* and *Core Gameplay*. All development efforts during this time were relatively smooth. Progress was slow, but consistent because I was able to tackle small pieces at a time rather than implementing the full system in one go.

Then, I attempted to use the model for the *Menus and Lobbies* evolution, but that proved

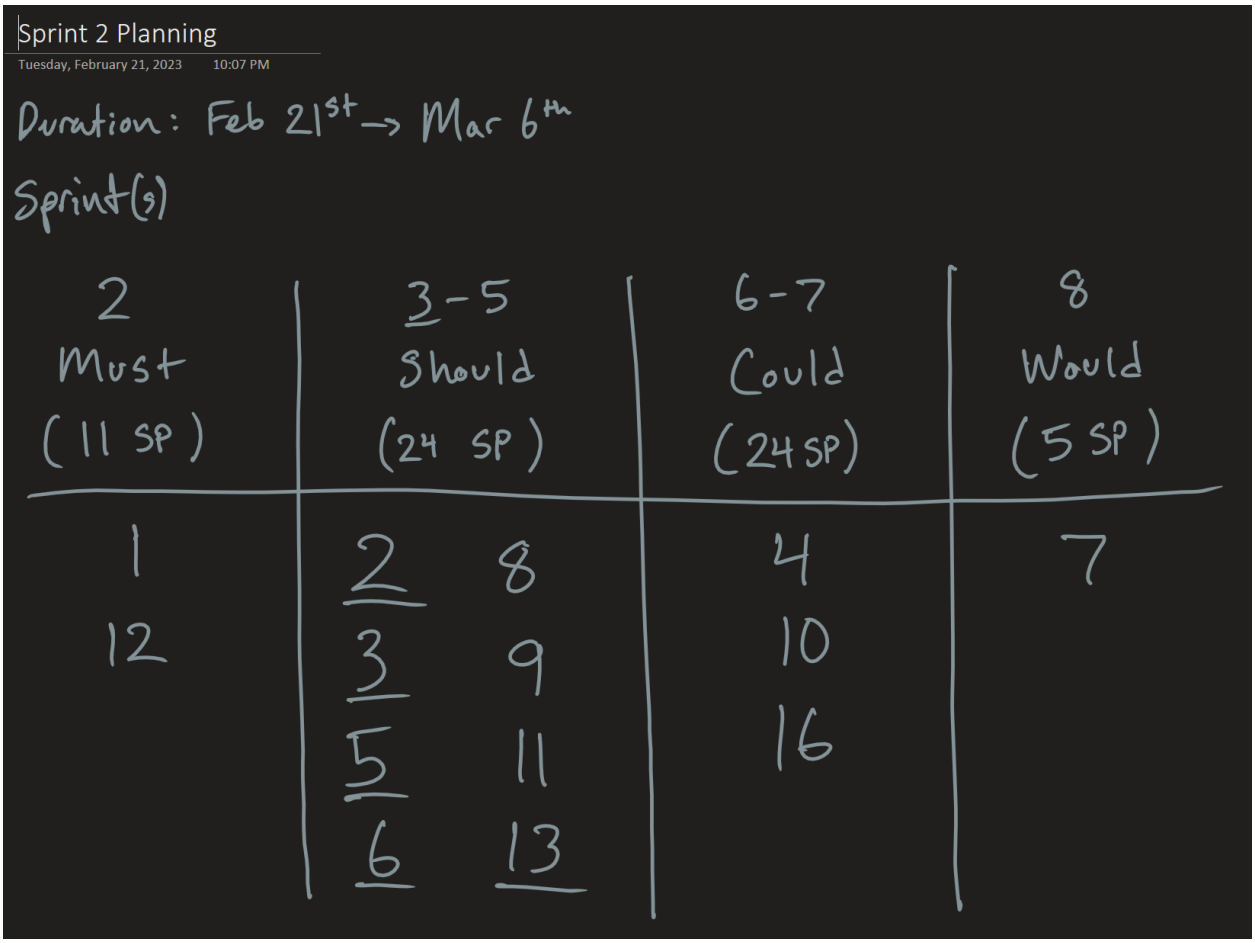


Figure 5. Sprint Planning Example.

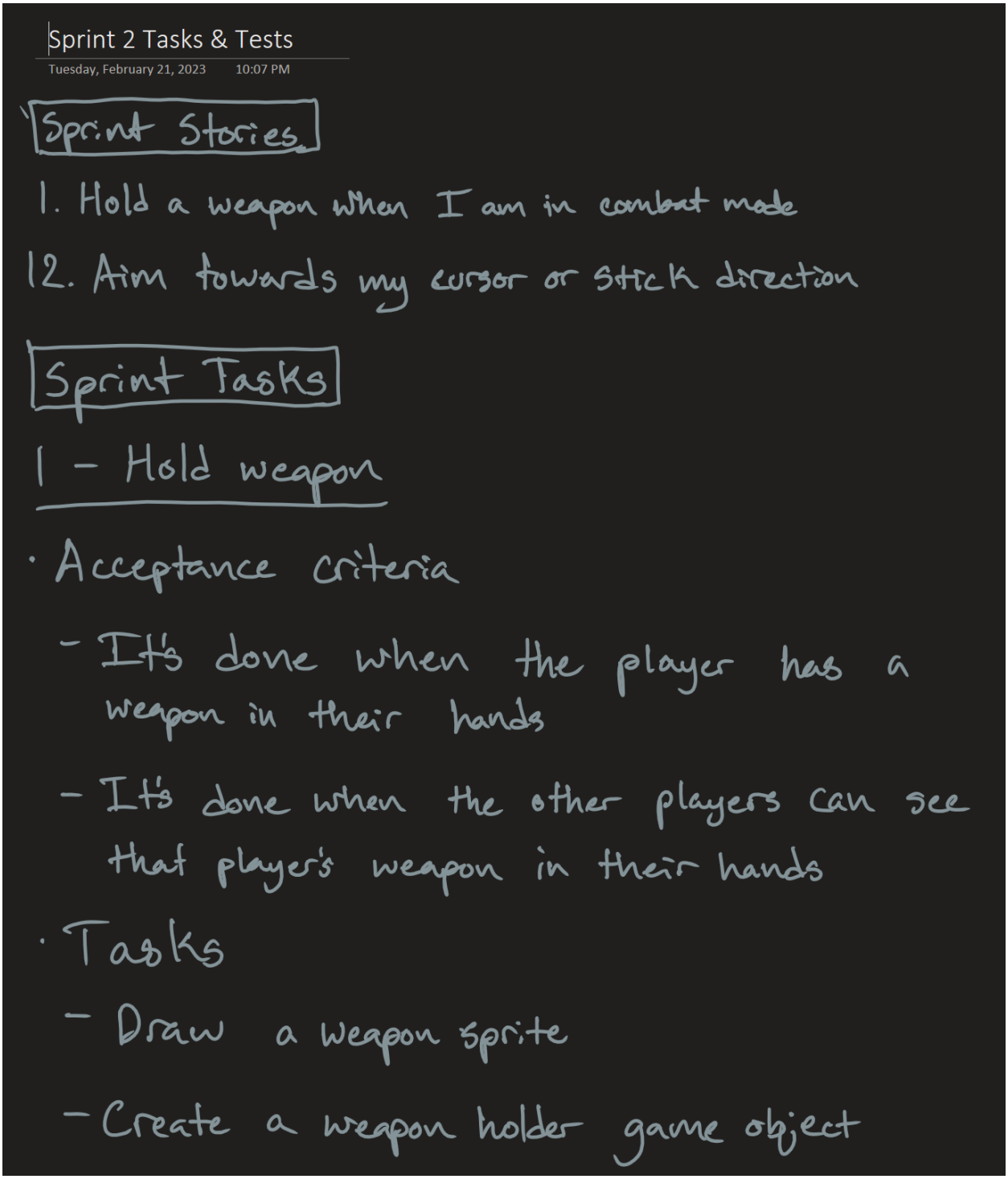


Figure 6. Sprint Tasks and Test Cases.

8. Bullets damage enemies

Status: Done

Priority: Should

Story points: 8

User story: As a player, I want to damage enemies with my weapon's bullets

Add a property

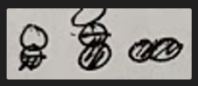
L Add a comment...

Acceptance Criteria

- [x] ~~It's done when the player has a health bar/indicator~~
- [x] ~~It's done when a bullet decreases the hit target's health~~
- [x] ~~It's done when a player's health is reflected across the server to clients~~

Polish Criteria

- [x] ~~It's polished when a bullet does more damage for headshots~~
- [x] ~~It's polished when damage counters are shown above the player that was hit~~
- [x] ~~It's polished when a player receiving damage has a visual damage indicator~~



- [ ] It's polished when a sound is played when a player is hit by a bullet
  - [ ] Distinct sound for a headshot

Tasks

- [ ] Create PlayerHealth component that reflects players' health across server
- [ ] Create visual health indicator that corresponds to player health
  - Number above player's head or simple health bar

Requ

Figure 7. Evolutionary Scrum Sprint Backlog Item.

to be difficult. At this point, I was again going to be navigating uncharted territory in the game development landscape. I had general ideas about how I was going to implement the lobby system, but those ideas ended up being far off from the current implementation today. Because I had little understanding about what I was going to implement, I had a difficult time even coming up with user stories to implement. This and some other reasons explained in the next section caused the project to come to a halt. As far as project management techniques, I think reverting to the previous development lifecycle model for this evolution would have benefited me greatly. There was so much I didn't know about how lobby systems were implemented, and I think a proper research and planning phase would have been very beneficial.

Overall, I think the Evolutionary Scrum Model is a good model for solo game developers. It is helpful to break the game down into evolutions so the developer does not get lost in the big picture for the game. Also, having a visible task list on a KanBan board is helpful for visualizing progress. Finally a tip I would give is try to recognize when there are sections of the project that need careful architecture design or research and planning before implementing. And, give yourself grace when tackling those problems because rushing through them will only hurt the project in the long run.

2.3.3. Losing My Way

In early December 2023, I fully stopped development on "Squirrels Gone Nuts". The project was a mess with build times over 5 minutes and "hot reload" times being over 30 seconds every time I saved a file. Also the project did not follow good software engineering principles because everything was tightly coupled and when one thing broke so did the entire project. I am not completely blaming scrum for this but it definitely held some responsibility with the lack of planning and documentation for the project. Another reason why development halted was that I was moving into a new evolution and I tried to jump right in to the scrum sprints without any research and planning. A lobby and matchmaking system was something I had zero experience with and it was very daunting to start tackling tasks without having any idea where to start. I think going back to the previous life cycle model would have benefited me a lot during this period of development.

Anyways I didn't touch the project for a good month. Then in January 2024, I opened a new project and started rebuilding the project. I did a small amount of preliminary planning for the new project's architecture. I used plenty of code and assets from the previous project, but I carefully inspected each script and tailored it to follow better software engineering principles. By February, I had gotten the project almost back to its original state. Also this was the first time I started working on the project full-time so progress was happening very rapidly. During this time, I completely lost my way with adhering to a software development life cycle, and I essentially woke up each day and wrote out what tasks I wanted to accomplish and set out to do them.

Although this is not a recommended approach, I found it to be extremely effective in the short-term. Especially for the task of refactoring the project. I was not adding any new features so I had the benefit of hindsight while implementing these features. However, issues started occurring when I continued trying to use this "model" after I had rebuilt the project to its previous state. I continued setting tasks for only the current day well into February

and I started to see the diminishing returns of this technique. Without structure and project management I had no foresight into the project's timeline. I overscoped the amount of work I expected I could do by the time I was supposed to finish the project. This was also a time where I was trying to learn new technologies and implement them simultaneously. Which led to me running into a bug that I got stuck on for over 3 weeks. The combination of learning new software and lack of project management led to March being one of the worst months of development on the project. Obviously not ideal timing with this being my final semester.

2.3.4. Returning to Scrum

I knew something needed to change. So, I found my way back to the evolutionary scrum model. I set out one final evolution that was mainly focused on tying up loose ends and testing the project.

2.4. Approach Moving Forward

Given this experience of testing and using different life cycle models, I can confidently say there is no single best approach to managing the development of a game especially for independent development. A few takeaways. I think breaking the project into evolutions helped tremendously with managing the scope of the project, and I will continue to do this for my projects in the future. When approaching an evolution or problem that has a known state space, that is there are few techniques and technologies to learn to complete the task. Then I recommend using an agile scrum approach because the development can be easily broken down and developed just by putting in the work. On the other hand, when the upcoming development is relatively new in terms of challenging your understanding as a developer, then I recommend taking some time to learn the technologies and best practices before implementation. During the *State Management and Synchronization* evolution I made three small prototypes implementing new multiplayer concepts before programming those concepts into the main project. Finally, reflect on and adjust the software development process often. There are always areas for improving efficiency and effectiveness of the project management process.

3. Requirements

3.1. Overview and General Project Requirements

In the previous section, I provided a chronological explanation of the different software development processes used over the course of the project. For this section and subsequent sections hereafter, I will only provide the relevant differences between the processes without chronological ordering.

The following are the general requirements of the project as stated in the project proposal.

1. Create a simple movement system that allows for creativity
2. Develop a fast-paced environment through player mechanics and map design
3. View the game in 2D as a side-view platformer
4. Players operate in two modes: combat (bipedal) and parkour (quadrupedal)
5. Players can run on any surface like they are on the ground
6. Connect players to game servers via matchmaking system

By design, these requirements are vague. From past game design experience, I knew the original idea I set out to build would probably differ from the final result. Some aspects would be better and some would be worse. So, I attempted to set the overall project requirements in a way that they could be adapted as the development required. That is not to say I allowed myself to completely pivot the game halfway through development. It just allowed for the freedom to "find the fun" in the game as development progressed.

3.2. Evolutionary Prototyping Requirements Process

While following the evolutionary prototyping model, the requirements were brainstormed in the initial phase of each evolution, and I followed the formula shown in Figure 8 to document them. I wrote out this formula for gathering and organizing requirements before the *Movement* evolution started. Since I interpreted the methodology to follow *mini-waterfall's*, I assumed I should gather the requirements into a formal requirements document and dismissed the idea of using user stories. Later, I would realize gathering requirements in the form of user stories is just as valid, and in the case of video games, more effective at fully conveying the requirement.

Requirements and Specification

1. Define features and systems as requirements
2. Organize the requirements
3. Expand requirements' details
4. Write formal requirements document
5. Estimate time needed for implementation

Figure 8. Evolutionary Prototyping Model - Requirements and Specifications Procedure

The following list demonstrates the end result of the *Requirements and Specifications* process in the form of a formal requirements document.

Movement Requirements

1. Grounded Movement
   (a) Lateral Movement
      - Move *left* or *right* relative to body orientation when movement input is pressed
      - Accelerate relative to acceleration variable
      - Limit top-speed to set amount
      - On key release, apply friction to reduce speed

   (b) Sprinting
      - When sprint input is held, increase top speed by sprint multiplier
      - On release, don't apply sprint multiplier
      - Sprinting is only allowed in parkour mode

   (c) Grounded Orientation
      - When *grounded*, match player orientation to surface below's normal vector
      - *Grounded* is a property that returns true when the player is within a height threshold above the ground

   (d) Grounded height above ground
      - When grounded, set height above ground to certain amount

2. Player State
   (a) Parkour mode
      - Player is quadrupedal
      - Increased movement speed

   (b) Combat mode
      - Player is bipedal
      - Reduced movement speed

- Other functionality will be added, but it is irrelevant at the time

3. Jumping

   - On jump key press, grounded is false
   - On jump key press, impulse force is added to player movement in direction of cursor
   - If cursor direction is between 180 and 360 relative to the player body, no jump is performed
   - Jumping can only be performed in *Parkour mode*
     - If in combat mode, switch to parkour mode then perform jump

4. Airborne Movement

   (a) Landing prediction

   - When in air, use projectile motion physics to predict landing spot
   - With known landing spot, rotate player body to match surface normal of landing spot

   (b) Forces applied

   - Movement mechanic will be added during combat
   - Must allow for forces to be applied to player while airborne

3.3. Evolutionary Scrum Requirements Process

During the evolutionary scrum requirements process, there again was an initial phase of brainstorming. However, this time the requirements were not documented in a formal requirements document. Instead they took the form of user stories. This proved to be a much more useful strategy for brainstorming in my opinion. With formal requirements, I was trying to jump ahead and figure out exactly what the player should do from a mechanical perspective. I could use the framework of "As a player, I want to..." to express what the player wants to do instead of what they should be doing.

Here are examples of both scenarios:

1. Formal Requirement

   - The player moves horizontally when pressing the left and right arrow keys and calculates the body rotation to match the ground so that the player can run on all surfaces.

2. User Story

   - As a player, I want to run on all surfaces.

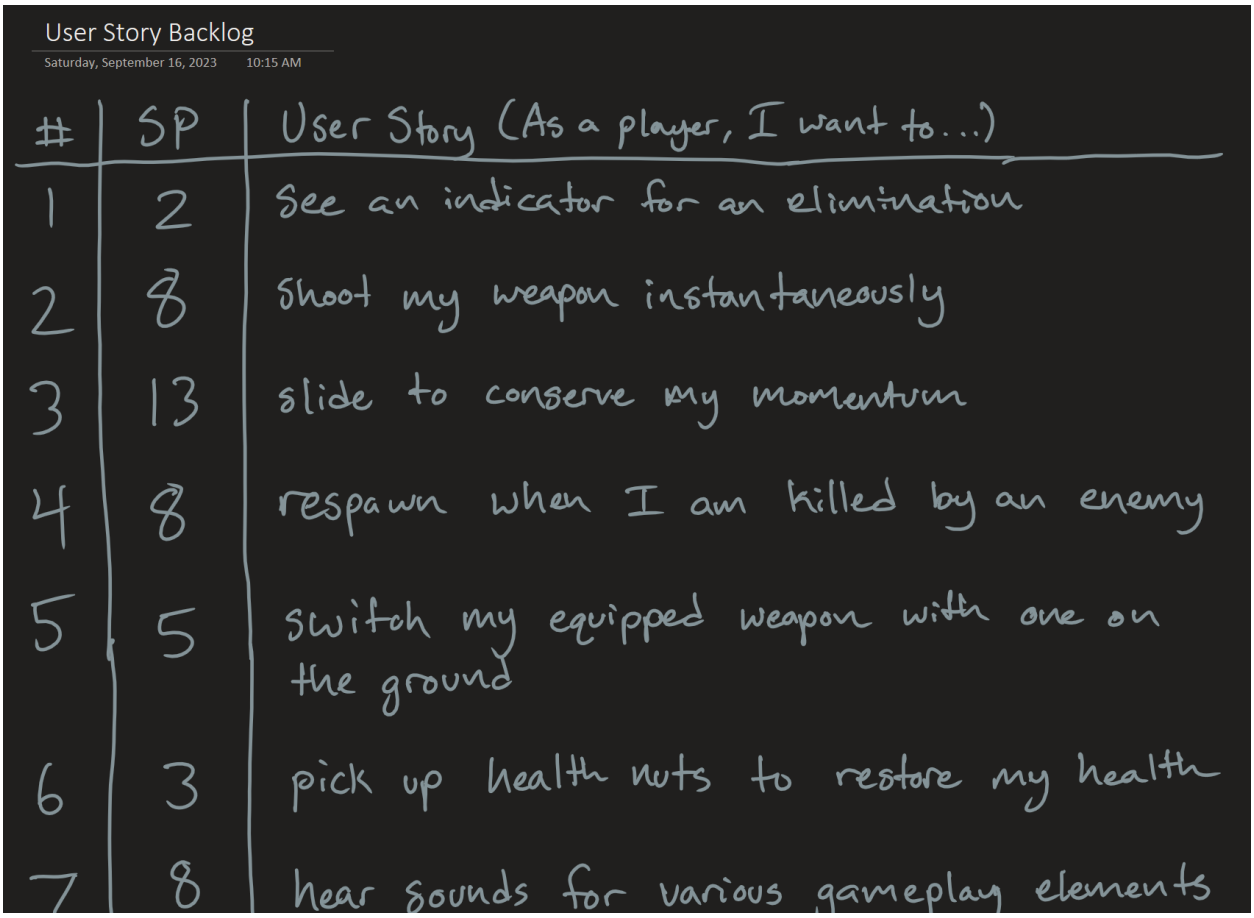


Figure 9. Core Gameplay Evolution User Story Backlog

By removing the technical aspect of the requirement, the implementation details remain separate from the idea. This leaves the implementation solely up to the developer. As a solo developer, it can be difficult to separate the roles of game designer and game developer, but user stories provide a simple framework to do just that.

In the context of game development, there is another benefit of user stories as opposed to formal requirements. User stories emphasize a player centered approach which focuses on the player's wants vs the software's needs. In turn, user stories guide the development path while still providing a connection to the emotional and entertainment needs of the player.

Figure 9 shows the user stories for the *Core Gameplay* evolution.

3.4. Final Thoughts

Going into the project, I had a rough idea of how I wanted the game to look and play. This is a pleasant change from having to guess and translate the client's ideas into requirements. So, the requirements phase of the project was really a matter of articulating my thoughts and vision instead of the traditional route of gathering requirements from the client.

4. Design

There are many components that make up a video game: graphics, physics, animation, audio, networking, and the list goes on. Luckily, modern game engines come equipped with many of these features off the shelf. While there are games built from scratch by solo developers, Minecraft being the most popular, most games today are built using sophisticated game engines like Unity or Unreal.

Considering a game engine is utilized to handle most of the low-level functionality necessary to make a game, the high-level components of this project include the following:

- Client interface
- Game server
- Database

The client interface needs to support 2D graphics capabilities and audio, IO, and networking interfaces. Then there is the game server that runs a simulated game environment based on the clients' inputs and communicates the updated game state to the clients. The database stores user profile data and manages real-time updates for the lobby states.

4.1. Data Storage Design

In the scope of this project, there was only a small amount of data storage necessary to fulfill the requirements, so external microservices were used to manage the data and reduce my development load. The authentication system and lobby management system were provided by Beamable [1], a 3rd party live ops service. The authentication mechanism was an anonymous and automatic login based on client's unique machine ID. That same service supplied the lobby management system which acts as a relay server between clients and the game server. Aggregating and synchronizing the clients into a virtual lobby before the matches take place. Both of these services were available through Beamable's API, so as far as data design, there was none.

4.2. Architecture Design

Going into the project I knew I needed to produce designs and models, but I didn't know the underlying architecture of the software I was using. I started designing the game around the assumption I would have full architectural control. In other words, my initial designs assumed I was not using a game engine that had predefined architectural patterns. As a result, I designed the preliminary project architecture around this model called the Entity Component System (ECS) [2]. This is a game architecture that is commonly implemented in game engines because it separates the data from the computation. However, Unity [3], the game engine I chose, does not follow this architecture at least not fully. I was designing the game without understanding what system I needed to program the game into. This resulted in many complications when it came time to implement the designs.

On the surface, it seems that Unity follows the patterns of ECS. There are scenes which are comprised of game objects similar to entities and these game objects are made up of components. However the components in Unity are intended to handle both the data and the computation. This marrying of the two concepts into one script creates tight coupling between the data and computation. Resulting in tightly coupled game objects which reduces the modularity of the scripts produced.

The following is a journal entry discussing decisions around ECS.

Reasons for Choosing ECS for Architecture and Modeling
Wednesday, May 18, 2022, 9:35 AM

1. Unity follows this structure already
   (a) One difference is Unity combines *components* with *systems* into one class
2. ECS makes more sense for game development
   (a) Eliminates messy inheritance trees
   (b) More accurately represents a game state
3. Will allow for small amounts of necessary network data to be sent
   (a) Simplifies the objects being sent over the network

As seen in the entry, I knew Unity combined the notion of components and systems into one unit, but I failed to foresee the implications it would cause. In retrospect, I was borderline obsessed with this architecture to such an extent that I designed my own *modeling* technique for ECS as seen in Figure 10. I used this technique to model the components in the movement evolution as seen in Figure 11.

During the implementation, I tried splitting data and computation into separate scripts, but this was inefficient and led to drastically more coupling than intended. The core idea of the ECS pattern is that the systems act independently from the entities. Systems are executed by the main game loop and are only concerned with performing computations on the components. This design philosophy is elegant, but Unity requires that the systems be placed on the entities themselves which undermines the entire point of the ECS.

After this failed attempt at formal design during the movement evolution, I abandoned it altogether. This was a mistake looking back on the project. I tried to comply with the agile philosophy where designs emerge from competent programming and intentional refactoring. The result was incomprehensible design diagrams and poorly organized code that required many refactoring sessions. Due to my novice knowledge of Unity, I believe the poor organization was inevitable.

The development of this game would have benefited from regular refactoring sessions. Ones where the project architecture was fully taken into consideration and components were carefully crafted to adhere to Unity's architectural design philosophy. Also, I think consuming more literature on the topic would have been very helpful. Instead of diving into the first solution I found and abandoning all others. Recently, I started reading the book

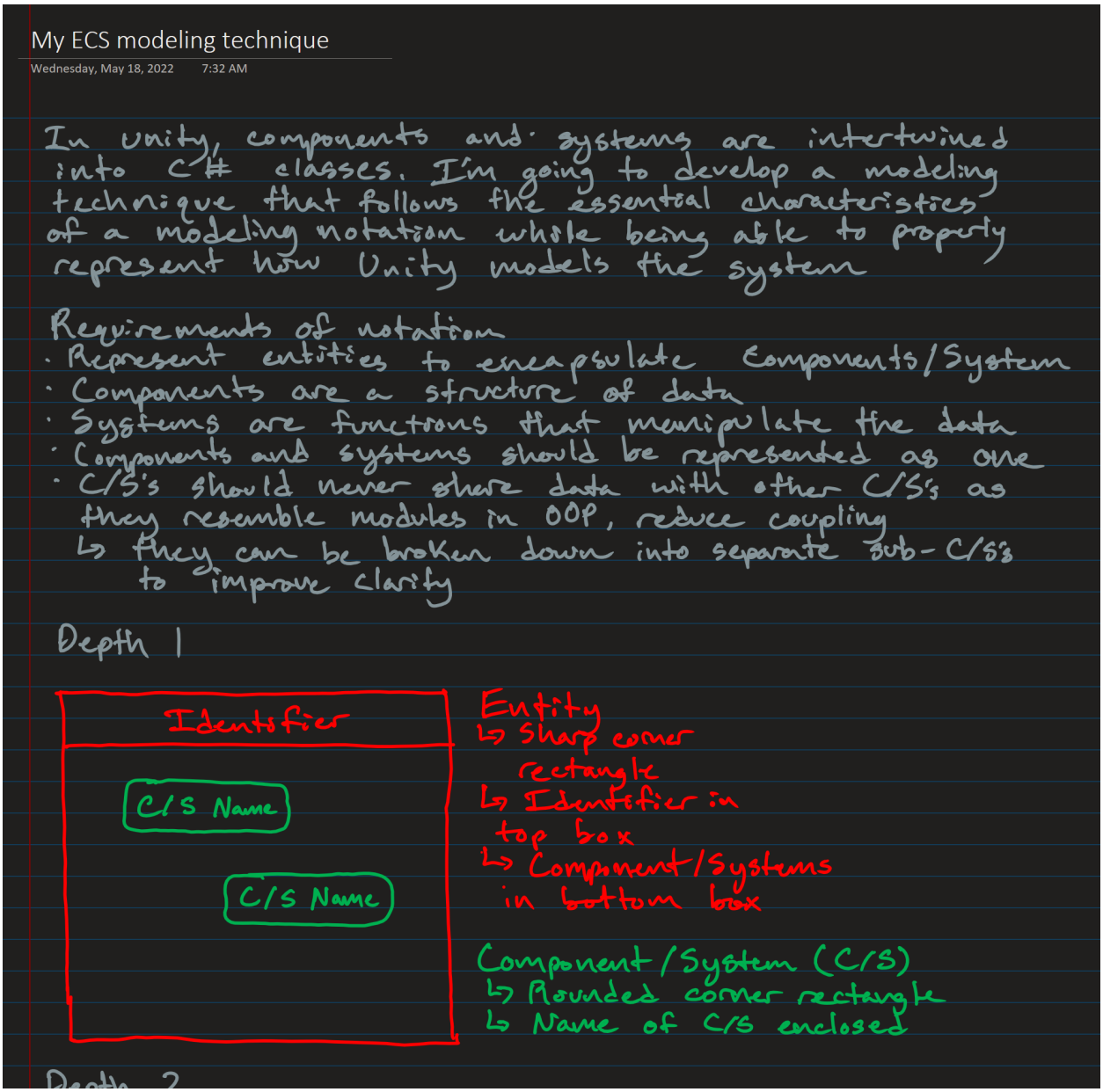


Figure 10. My ECS Modeling Technique

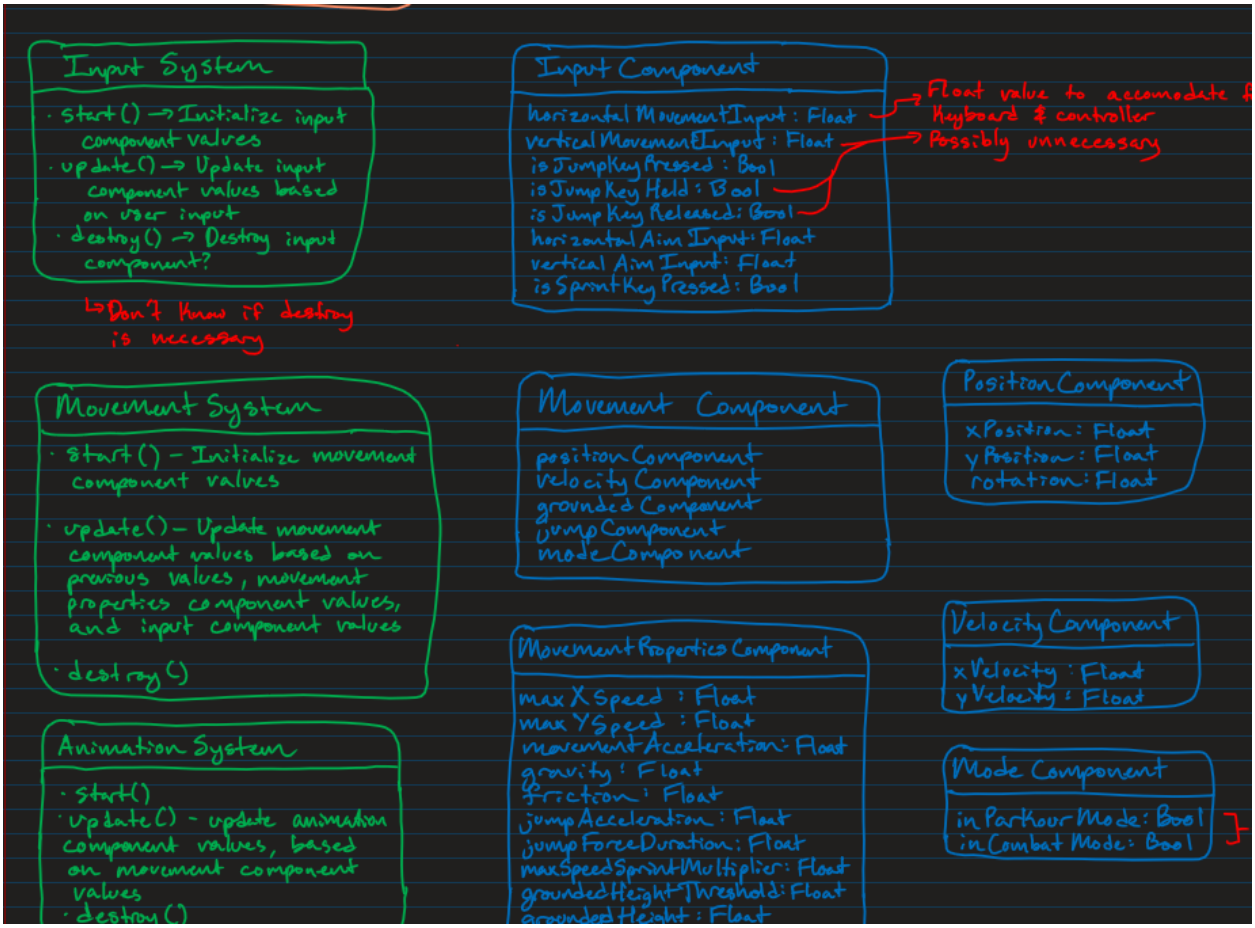


Figure 11. Movement Evolution ECS Models

"Game Programming Patterns" [4], and if I had known about this book before starting the project, I predict I would have made a lot less mistakes in terms of design.

The one thing Unity did get right about the pattern was favoring composition over inheritance. Especially in game development, composition is favored over inheritance because it is more extensible and easier to refactor with changes to the requirements. In Unity, game objects or entities are represented as collections of components and other child game objects. Although these components contain both data and computation, they still adhere to the principle of separation of concerns, enabling a more modular and flexible approach to game development.

4.2.1. Reflections on Technology Influencing Design

A common theme throughout this entire process is unfamiliarity with the technology causing incorrect design assumptions. I don't think this could have been avoided unless I built my own engine from scratch which would have brought along with it a whole host of other problems.

With the benefit of hindsight, I would have chosen a different game engine that aligns better with my intuitions for architecture design. Unity is very unopinionated in terms of its architecture guidelines. From this project and others, I learned that I prefer opinionated languages and software tools. Even if the implementations tend to be more verbose, I enjoy the ability to develop a consistent mental framework for architecture design within the constraints of the tool. For example, in my second to last semester, I took it upon myself to learn Golang with my machine learning class. Within a week or two of learning it, I felt confident that I could design complex software architecture in Golang because it offers a simple and consistent design philosophy. Whereas to this day with Unity, I can set out to build a new project with all the components laid out and still not properly design the architecture so that it doesn't become too coupled down the line.

There is an engine called Bevy written in Rust, and it enforces the ECS architecture. Many times throughout the project I considered switching game engines because of architecture design issues and other factors. However, I fell victim to the sunken cost fallacy and stayed with Unity. Had I chosen to use Bevy or had I written my own engine, progress in the beginning would have been much slower. But, I believe the return on investment would have been much greater by the end of the project solely due to the architectural control.

4.3. Design Decisions

I document a large portion of the projects I work on, but my documentation process rarely fits into formally defined process. My notion of *Design Decisions* are another one of these cases. In development, there are crossroads and a decision has to be made about using a specific technology or process to solve a problem. I document these decisions in the form of a journal entry. Usually this is a constant flow of consciousness to the page pouring out all the current knowledge I have on a particular topic and why I chose a certain technology or process over the other. There are plenty of times where this decision is reverted in the light of new information. But, I think this process helps me organize the information I gathered and come to reasonable conclusions on the topic.

Design Decisions for Combat Evolution

Tuesday, February 7, 2023, 12:55 PM

So I am currently working on the camera positioning and feel. I know this is more related to the movement evolution but whatever.

I experimented with making the camera dynamic to follow the midpoint between the mouse and the player. However this is somewhat jarring. I think it is just too much movement especially since it is controlled by the player. It doesn't feel like I am in control when it is enabled.

So I think am going to try a different technique. Instead of panning the camera around the player to give them the chance to see more, I am going to zoom out the camera when the player is moving fast and zoom it back in when the player's velocity is lower. I think this will give the player a good sense of speed and it will be nice to know where you are in relation to the rest of the map when you are moving fast.

I implemented the new system and I think it feels really good. Definitely going to have to run it by some play-testers to get a final verdict. I also moved away from the Cinemachine and I will have to clean up that part of the project now that I'm not using it. I didn't solve the jitter issue but I think I will have a better chance at solving it now that I'm not using Cinemachine.

5. Implementation

5.1. Technology

5.1.1. Game Engine: Unity

The choice of game engine has been eluded to many times in this report. If the last section wasn't clear enough, the one decision I regret was choosing Unity [3]. The choice was based entirely on familiarity even though I had only built small games and prototypes with Unity. I can't fully discount Unity though because it does simplify much the game development process. Unity satisfies all of the functional requirements of the project by providing a graphics engine, physics engine, and a suite of development tools that streamline the creation of games. In reality, I would have likely not achieved as much as I did, had I not used Unity, and there is good reason why it is the most popular game engine today.

5.1.2. Networking Solution: FishNetworking

Out of all the technology choices in this project, the networking solution was the one I spent the most time researching. There were many requirements to fulfill and FishNetworking [5] met all the needs. Here are compiled journal entries detailing my thought process and all the solutions I considered.

Design Decisions for State Management and Synchronization Evolution

Thursday, October 6, 2022

I am starting the unity Netcode tutorials to evaluate if this is the networking system I want to go with.

Tuesday, October 18, 2022

I looked into networking alternatives and here is what I found:

- Unet: the old networking solution that Unity provides. It is now being deprecated because of the new system
- Mirror: an opensource fork of Unet that has been improved upon the many pitfalls of Unet. However it was mainly built with MMOs and low tick rate games in mind, so it is not the best solution for a fast-paced environment like mine.
- Photon Fusion: this was the go-to solution for developers that were building a fast-paced high tick rate game because of the features provided by it such as client-side prediction and lag compensation. However like all good things it isn't free.
- In-house: some developers report writing their own networking solution and have really enjoyed the solution they came up with because it is a

lot more lightweight compared to general purpose solutions. However through anecdotal evidence they have reported working on their projects for multiple years and that is something I don't have.

Tuesday, October 25, 2022

Netcode for GameObjects

I am going to use Unity's built in networking solution for my state management and synchronization. I chose this for a few reasons. First being it is built by Unity so I know it will be compatible with their current versions and it is intended to be their all-in-one networking solution for the foreseeable future. Second, it has plenty of documentation because the solution has been in 'beta' for a while with developers ironing out the kinks. So, after a few years of development the documentation has become pretty robust. Thirdly, it is going to save me a lot of time rather than developing my own in house solution. I considered it throughout this research phase, but there never seemed to be good reason to do it. Especially when Unity is going to provide this working and relatively lightweight solution with probably five or more developers working on it as I write this.

Wednesday, November 2, 2022

So, I was wrong. In my initial findings I thought Netcode for GameObjects was going to be the perfect solution for all my needs. Long story short, it's not. I was going to start listing out my requirements and specifications when I realized I don't know how to design a multiplayer architecture. So, I looked it up and found plenty of fantastic resources. There were a few things that stuck out to me that I knew I would need in my game: client-side prediction, server reconciliation, entity interpolation and lag compensation. Only entity interpolation is implemented in the current version of Netcode for GameObjects with the others on the roadmap.

After a bit more research, I found an option that I think will suite my needs. FishNetworking. It has client-side prediction, server reconciliation, entity interpolation all for free. Lag compensation is available for a fraction of the cost of Photon Fusion. There also is plenty of documentation and tutorials as well as an active discord server. From taking a quick look at the documentation, the development process and experience seems very similar to Netcode for GameObjects, if not better.

Saturday, November 12, 2022

> Today I followed those concise tutorials I found. Very helpful in understanding some techniques used when designing and making a game using FishNet. I threw FishNet into SGN and it just worked. I don't know how to explain it, it just worked flawlessly.

5.1.3. Hosting Provider: Hathora

The server hosting provider was iterated through a few times throughout the project. At first I was using a service called PlayFlow Cloud [6]. I primarily used it for the initial multiplayer testing because the service offers a free testing server.

The next hosting provider I moved to was intended to be the all-in-one final solution, Microsoft's Azure PlayFab [7]. This service provides much more beyond server hosting. They provide multiple API's for user authentication, lobbies, matchmaking and plenty of other live ops services. I moved the project to this solution and got a solid implementation working with the API's. The only problem was that I didn't read the fine print for the server hosting. The service provided 750 free core hours per month which is great because that covers running the server nonstop. The problem is PlayFab requires the usage of a server with minimum of 2 cores. So halfway through the month PlayFab would start charging for the server even when it wasn't in use. After that first bill, I knew I needed a new solution. But, I was nervous to give up the live ops services.

I started searching for new solutions and came across a newer hosting provider called Hathora [8]. This service promised on-demand servers that only charged based on usage, and they would shutdown the servers after all connections were closed. As a solo developer without high expectations for a large playerbase, this is a huge benefit. On the chance 'Squirrels Gone Nuts' does explode in popularity, the service promises the servers are globally scalable. Hathora also gave me 500 dollars in free credit to use for testing and prelaunch.

5.1.4. Live Ops: Beamable

After finding a hosting provider, I was still looking for a live ops service to replace PlayFab. I had considered implementing my own but I knew time was running slim on the project and honestly I didn't want to implement all those things. One day, I was on Hathora's discord server and I noticed people were talking about this service called Beamable [1]. I checked out their website and it felt like a golden egg landed in my lap. This service provided everything that Azure PlayFab had and more and at a cheaper price. The Beamable services currently in the project are the authentication and lobby management services.

5.2. Process

5.2.1. Gameplay Loops

Before explaining the order in which the project was assembled, I need to explain the framework behind dividing the project into its components.

> "In game design, a gameplay loop is a repeatable sequence of actions the players engage in that makes up the primary flow of your players experience that keeps the them playing over and over again." [9]

There are three categories of gameplay loops: primary, secondary, and tertiary. Each encompass sequences of player actions on timescales ranging from seconds to days. The primary gameplay loop is the sequence of actions happening on a second to second basis. For simplicity, I will use Tetris as an example. The primary loop in Tetris is rotating and placing blocks to leave no gaps in the stack. The primary loop is generally regarded as the most important loop because the player engages with these actions the most. If the player doesn't enjoy the primary gameplay loop, they won't ever make it to the secondary and tertiary loops. The secondary gameplay loop takes place on a minute to minute basis. In Tetris, the secondary loop is setting up the stack such that you can perform a 4 line clear with the long piece. This loop generally keeps the player engaged throughout the play session. The final gameplay loop is the tertiary loop. This loop is focused on actions over hours and days. For Tetris, this is the player attempting to beat their high score. This loop provides engagement over longer time periods spanning multiple sessions and is typically what keeps the player coming back to the game.

5.2.2. Project Components

As explained in the Software Development Process section, the project was divided into many *evolutions*. The division and ordering of these evolutions was based the progression of the gameplay loops from primary to tertiary. Since the hook of the game would be the movement system, I set it as the first evolution. I wanted to allocate adequate time to getting it *right* because it was a major contributor to the primary game play loop. Following the *Movement* evolution was the *State Management and Synchronization* evolution. Initially, I had planned for the *Combat* evolution to be next, but I figured it would be easier to implement the networking earlier rather than later. Soon after was the *Combat* evolution which also played an integral role in developing the primary loop. With the conclusion of the *Combat* evolution, I could move on from the primary gameplay loop and focus on the secondary loops. The *Core Gameplay* evolution focused on respawning, picking up health nuts, free-for-all gamemode, and other quality of life changes. Then the *Menus and Lobbies* evolution which is also categorized as a factor of the secondary gameplay loop. This involved players hosting, finding and joining lobbies. Finally, the tertiary gameplay loops. These were listed as stretch goals for the project and I haven't reached them in development yet, but the plans are to implement the ranked system and unlockable weapon and player skins through challenges before release.

5.2.3. Kanban

During the first two evolutions of the project, following the evolutionary prototyping methodology, I didn't have a specific process for implementation. After writing the requirements and designing the architecture, I would simply write code to fulfill those requirements and designs. This resulted in a loose timeline for these implementation phases, and deadline

extensions were common. The only thing for tracking progress were the development logs and the Git history. Both of which are not ideal for tracking progress visually.

After moving to evolutionary scrum, I started using a Kanban board to track the progress of the user stories. The board provided a visual overview of the progress I was making throughout an evolution. This was helpful in organizing activities from the day-to-day implementation to the overall evolution planning. I enjoyed this version of project planning and implementation due to the small achievements sprinkled throughout the process whenever a user story was marked complete. The Kanban board was a healthy way to view progress and keep motivation high because I could break the project down into simple components and not be overwhelmed by the big picture.

5.3. Changes to Requirements

Earlier I talked about this concept of *finding the fun* and how it brings a unique perspective to the definition of a game's requirements. Due to this there were plenty of changes to the requirements.

Back in the 3D version of the game, I intended the player to have precise control over which direction their player jumped. This would have been a novel movement system for the genre. When I transitioned the game to 2D, I thought I needed to retain this idea. So I set out to build just that mechanic. When the player was quadrupedal the player would be able to jump in the direction of the cursor. On paper this idea sounds really cool being able to pull off crazy stunts like jumping in the opposite direction of player movement. But in practice, the controls were unintuitive and acted as more of a hindrance than benefit. Unforeseen challenges such as what to do when the player tries to jump into the ground started to make that mechanic difficult to implement. I ended up abandoning the mechanic in favor of a more traditional jump mechanic that just adds an impulse to the player in the direction above the player. Although simpler, the mechanic turned out to be very expressive in its own right. Players became accustomed to the mechanic and were able to perform interesting move-sets due to the consistency of the mechanic.

This is just a single example of a change in the requirements to which there were many throughout the course of the project.

5.3.1. Added Functionality

Two unplanned mechanics found their way into the game during implementation. The first being sliding and the second being airborne knockback.

At the start of the project, there were two modes the squirrel could exist in: sprinting and shooting. During the *Core Gameplay* evolution, however, I added another mode called sliding. This mechanic allows the player to conserve momentum at the cost of not being able to shoot or directly increase speed. I say directly because the player is still subject to gravity, so when the player is on an incline the squirrel will increase speed like a ball rolling down a hill. Nowadays, this sort of mechanic is common in first person shooters, and I thought it would be a great fit for a game based on fast movement.

The airborne knockback mechanic was set be implemented during the *Movement* evolution, but I abandoned the idea due to complexity and dependence on the weapon systems

which weren't implemented at the time. The mechanic essentially applies a force to the player when they shoot their weapon in the air. I revisited the mechanic during the *Combat* evolution, but again the complexity proved to be a major time sink. It wasn't until the large refactor at the start of 2024 that I realized the mechanic was far more trivial than previously anticipated. Without going too far into the weeds, the movement system is a client predictive system where the player sees their inputs reflected in the game before the server authorizes the game state. The weapon system is also client predictive but it is purely event based whereas the movement system operates on synchronization mechanism between client and server. These two systems don't mesh well together, so during the refactor I moved the weapon system to also operate on the same synchronization mechanism.

5.3.2. Unfulfilled Goals

I had set many lofty goals at the beginning of this project and plan to continue working towards those goals post graduation. Two stretch goals in the proposal were a ranked matchmaking system and a profile system.

Starting with the ranked matchmaking system, I am deeply interested in the problem of matchmaking players based on skill. There is plenty of nuance to this topic and it is central to many discussions around competitive games. In November 2023, I attended a game development conference in Madison. One of the talks I was at was about the concept of hidden MMR (Matchmaking Rating) which is how most games implement their ranked systems. Players have a visible rank and a hidden rating, and the latter is used by the matchmaking algorithms. This concept leads to frustration among the players because while their visible rank is low the hidden rating could be much higher leading them to be paired with more skilled opponents preventing a natural growth in rank as their skills progress. There are also issues with stagnation in the ranks where the algorithm determines the player's skill level and provides little opportunity for change to that level even as the player's skills progress. The presenter of the talk argued there is no need to differentiate the visible rank from the hidden rating if the MMR adjustments are fair. I would have liked to design and implement a fair MMR system into my game because I have always enjoyed competitive games and I believe I could improve the algorithms used today.

As for the profile system, I implemented the simple authentication mechanism using Beamable, but that hardly scratches the surface of what I initially intended with the profile system. By release I would like to implement a proper authentication mechanism, player statistics, friends system, and storage for in-game items. All of these features are available through Beamable API's, so it is a matter of taking the time to implement them.

6. Testing and Verification

6.1. Overview of Testing Approaches

In developing "Squirrels Gone Nuts," a variety of informal testing approaches were utilized, each serving a crucial role in enhancing game reliability and performance. The primary methodologies included integration and system testing, which focused on the interaction between various game components during the development phases. Usability feedback, derived from playtesting sessions, provided invaluable insights into player interaction and game interface design. Regression testing was also employed, particularly useful following iterative updates to ensure new features did not disrupt existing functionalities. These testing methods, though informal, were pivotal in ensuring the game components not only functioned together but also resonated well with player expectations.

6.2. Test Development and Execution

The development of test cases for "Squirrels Gone Nuts" was centered around anticipated user interactions and gameplay mechanics. These test cases were somewhat informally documented, captured through development logs and direct feedback during playtesting sessions. Testing execution was predominantly manual, utilizing tools like the Unity Editor, which allowed for real-time bug tracking and immediate adjustments. This manual testing approach provided a hands-on way to identify and resolve issues quickly. Looking forward, there is a plan to formalize the test documentation process and incorporate automated testing methods to enhance the efficiency and coverage of tests, ensuring a more robust and reliable experience.

6.3. Testing Frequency and Coverage

Testing was conducted intermittently, aligned with significant development milestones and prior to public playtesting sessions. This approach was chosen to balance the need for thorough testing with the constraints of an agile development process. The coverage of these tests primarily focused on functional requirements such as gameplay mechanics and network performance. However, non-functional aspects like performance optimization and hardware compatibility, particularly concerning the upcoming Nintendo Switch port, were not as rigorously tested. The absence of formal unit testing has been recognized as a significant gap, which is planned to be addressed in future iterations of the game development to ensure a comprehensive testing framework.

6.4. Reflections and Future Testing Strategies

Reflecting on the testing processes employed during the development of "Squirrels Gone Nuts," it is clear that while the informal testing approach facilitated rapid iteration and feedback integration, it also highlighted the necessity for a more structured testing regimen. Future testing strategies will include the integration of systematic unit testing and an

increased emphasis on test automation. These enhancements are aimed at improving the robustness and consistency of the game across different platforms, particularly in preparation for the planned console ports. By adopting a more structured approach to testing, the goal is to not only maintain the game's quality across various platforms but also to prepare the game for a broader and more successful release.

7. Validation

Software is validated by answering the following question: "Does the product built satisfy the high-level requirements?" This question is answered through various techniques comparing the end result to the initial requirements. With software development cycles becoming more iterative, validation occurs more often in the development process. This is especially true in game development. The requirements are defined by satisfying the player's desires, and these desires can be hard to identify. Furthermore, the game might meet the designer's requirements, but still not meet the needs of the players. So, the game must be constantly validated and iterated on.

To start this section, I will outline the requirements stated in the project proposal. Then, I will demonstrate the validation techniques employed throughout development. Finally, I will examine these processes and note improvements for future projects.

7.1. Overview of the Project Proposal

The project proposal denotes a few key goals for the project from a software validation perspective. I set out to build a game summarized by this statement: "a web-based multiplayer videogame that produces a competitive and fast-paced environment for players." Initially, I planned to deploy to a web-based environment, but I quickly pivoted to a traditional software application. This will be explained further in the Deployment 9. section, and for validation purposes, it is irrelevant.

The first goal was to create a multiplayer game; satisfied by connecting multiple players to a single game state. I also set out to create a competitive and fast-paced environment. These are more difficult to measure as they are feelings experienced by the player rather than functional requirements.

The project did not have an external client, so the requirements were my interpretation of the player's needs.

7.2. Validation Assessment Techniques

I used two techniques to validate the game. For the concrete requirements, I validated them through the notion of Acceptance Criteria. I validated the more abstract requirements through User Acceptance Testing.

7.2.1. Acceptance Criteria

A user story's acceptance criteria are a list of functional requirements that must be satisfied for it to be marked complete. The acceptance criteria were generated in the sprint planning phase. After backlog grooming, I would examine each user story on the next sprint's backlog and list out its acceptance criteria. In Agile environments, stakeholders are consulted in the setting of the acceptance criteria. But, the only stakeholders on the project were potential players and me because this was a solo endeavour. So, I did not involve any stakeholders in the setting of these criteria.

I also invented another set of criteria I coined Polish Criteria. Polish criteria are not essential to creating a functional game, but they satisfy the artistic and game feel requirements. These were not required to be completed to mark a user story as complete. However, the more polish criteria I completed, the richer the game felt.

I used Notion to organize the user stories and track the fulfillment of the criteria. In the Kanban board, I designated a column titled *Polish*. When a user story's acceptance criteria was satisfied but not the polish criteria, I would move the card to the *Polish* column. If all the cards were moved to this column and time was left in the sprint, I would implement the polish items. But if there was no time left in the sprint, I just moved the cards to the *Done* column because the next set of user stories needed attention to maintain velocity.

Here is an example user story from the *Combat* evolution:

"As a player, I want to fire a weapon in the direction of my aiming mechanism when I press the fire button."

Acceptance Criteria:

1. It's done when the player fires their weapon and a bullet is spawned
2. It's done when the player can see bullets
3. It's done when the player can see enemies bullets
4. It's done when the bullet firing uses lag compensation
5. It's done when a bullet's direction is toward the player's aiming mechanism

Polish Criteria:

1. It's polished when the weapon has a firing sound
2. It's polished when the weapon has a recoil animation after firing
3. It's polished when a weapon has a muzzle flash visible to all players

It is worthy to note that for the *Movement* and *State Management and Synchronization* evolutions, I did not generate acceptance criteria. I only started writing out the acceptance criteria when I moved to the *Evolutionary Scrum* model. In the *Combat* evolution, which was the first one using the evolutionary scrum section, I had not yet created the notion of polish criteria. This caused a blending of criteria and confusion on whether a user story was considered complete or not. I later divided the criteria into acceptance and polish and it provided much more clarity to the whole process.

Something I did not consider at the time was using external feedback to verify the acceptance criteria. In retrospect, I think it would have been useful to provide the acceptance criteria to playtesters to get their feedback on the fulfillment of the criteria. This would have given playtesters a framework/lens to view the game through rather than just playing and giving general feedback.

7.2.2. User Acceptance Testing

As stated before, User Acceptance Testing focused on validating the abstract requirements of the game. These include the game feel and player engagement. To validate these, I invited my friends to playtest the game with me. These were open play sessions where we would play a few games. In the first few playtests, I focused more on the multiplayer actually functioning properly. But by the last one, my friends and I were just playing the game to see who could win the most matches.

I conducted unscripted interviews after the playtest sessions where I would ask the playtesters to give me their thoughts. I had a OneNote page setup to take notes, mainly noting bugs, feature suggestions and general feedback. Since these players were not QA testers by trade, they often couldn't tell if something was a bug or if it was intentional. I noted this after the first few playtests which led to me asking them to record their gameplay. This proved to be a good strategy for finding bugs, but also observing how the player interacted with the game. Also, I participated in the playtests alongside my friends, and this helped me find bugs that are far more difficult to discover by yourself. For example, one of the playtests had an issue with the respawning system always spawning players in one of two spawn points. This is something I had difficulty testing until I ran a playtest with 5 other players.

I tried creating a regular playtesting schedule around the development cycle, but I found myself scrambling and crunching before every playtesting session because of the lack of integration testing. I would delay and then skip playtests which resulted in very infrequent playtests. Looking back I wish I would have improved the unit and integration testing so that I could have performed regular playtests. I am confident in my continuous integration and deployment skills in the web and mobile development realm, but I have yet to explore it in game development.

Despite the infrequent playtests, there were many great insights gained from them. One being, the weapon balancing, which is how powerful the guns are in relation to one another. Weapon balancing is a difficult scenario to test with one individual because it is more of a feeling gained from actual play. In multiplayer shooters, it is common to have to balance weapons even after release because if one gun is over powering the others, then the game becomes unfair to players who don't have that gun. Weapon balance issues were by far the most common feedback from players which I took as a positive because that meant the other areas of the game were working well.

There were also plenty of feature suggestions from the playtesters that have made it into the game. I can assure you the game would not be where it is today without them. Here are a few suggestions from the playtesters:

- Kill feed
- Sliding
- Health nuts
- Scoreboard

7.3. Conclusions and Future Improvements

There were two validation methods used to ensure Squirrels Gone Nuts met my promises in the initial project proposal. The acceptance criteria validated the day-to-day operations of development whereas the user acceptance testing validated the overall project goals. Both of these methods were effective in identifying the successes and shortcomings of the game.

Moving forward, I would like to implement more frequent playtests on a schedule aligned with the chosen lifecycle model. I foresee this benefiting development in more ways than one. First, it would provide steady feedback which makes it easier to realize when a mechanic is not working before sinking too much time into it. Secondly, consistent feedback surfaces bugs that are difficult to test independently like the respawn issue stated earlier. Finally, playtests fuel my motivation. Hearing feedback from players enjoying the game motivates me to fix the areas where they see need for improvement.

8. Security

In the current state of the game, there are few security concerns. The only user data stored is the player's username. I opted to use a 3rd party service, Beamable [1] for authentication and storage of player data. The player is anonymously authenticated using the device ID upon launching the game. After authentication, the player's username is retrieved via Beamable's API. Beamable provides a secure API that handles the authentication and transfer of data, and it assures the use of proper security practices in their documentation.

Beamable Securty Overview
https://docs.beamable.com/docs/security-overview

Authentication
For user auth, Beamable implements OAuth2 with the following grant flows for token-based auth:

- Password
- Authorization Code
- Federated/Social Login
- Anonymous/Guest

For server auth, we use signed requests using an MD5 Digest of the app secret and other information. The signature is then included in the header of the request.

For password storage, Beamable uses the BCrypt password hashing function, which encompasses a random salt and a difficulty factor in hash, protecting against both brute force and rainbow-table attacks.

Encryption
All data is secured at rest and in flight by industry-standard SSL/TLS encryption within the Beamable platform.

Figure 12. Beamable Security Overview - Authentication and Encryption

I mentioned there are only a few concerns for the current state of the game. I plan on expanding the authentication mechanism later in development which could increase the security concerns. Beamable provides a mechanism for authenticating using username and password that I plan to use come release. Still there appears to be no valid attack vector because only encrypted data transfer occurs through the Beamable API.

From an infrastructure perspective, the game servers are of interest for an attacker. If they are compromised, then there is no game. The main attack vector for this would be a Distributed Denial of Service (DDoS) attack. Hathora [8], the game server hosting provider, assures protection against DDoS attacks.

9. Deployment

The deployment for this project was simple because Unity provides a versatile build system. The build system compiles the entire game into a single executable and can target all the major gaming platforms. This tends to be implied with a multiplayer video game, but the client must have a computer connected to the internet. As for hardware, the following are the minimum specification recommendations:

- OS: Windows 7 or later
- Processor: Intel Core 2 Duo, 2GHz or AMD Athlon 64 X2 2GHz
- Memory: 2 GB RAM
- Graphics: DirectX 10 or OpenGL 3.0 compatible GPU with 512MB VRAM
- DirectX: Version 10
- Network: Broadband Internet connection
- Storage: 1 MB available space

9.1. Deployment for Playtests

During development, I used a website called itch.io [10] to facilitate the distribution of the executable for the playtests. This site is a game distribution platform aimed towards indie developers. Playtesters would simply visit my itch.io profile, download and extract the zip file containing the executable, and run the game.

9.2. Deployment for Release

For release, I am choosing to publish on Steam [11], a digital distribution platform. Steam is one of the most popular platforms for buying and playing PC games. I am planning to release the game in November 2024. There are a few tasks I need to do before release, and I will go into further detail of my plans in Section 11..

9.3. Reasons for Changing from Web to Steam

The decision to switch from web deployment to a traditional application happened early in development. There are a few reasons behind the decision. Web-based games are far less lucrative than traditional games. This is evident by the fact that a majority of games are published to platforms other than the web. Also, the revenue model for web games is typically advertisement based, and I did not want to have ads in my game. However, money was not the only reason for the switch in deployment strategy. Unity's WebGL build requires a different transport layer for the network communications. This likely would have increased the complexity of the project. Also, Steam provides many built-in promotion mechanisms. With a web-based game, I would have to do all the marketing whereas Steam has built in mechanisms to show your game to players who might enjoy it.

9.4. Future Plans

About halfway through development, my plan was to publish the game on all the major gaming platforms: Xbox, PlayStation, Nintendo, and PC. I still have that ideal in mind but on a longer timescale. Initially, Squirrels Gone Nuts will be exclusively available on Steam. Shortly after launch, I plan to publish on the Nintendo Switch. At that point, if there is substantial traction, I will look into publishing on the major consoles.

Another aspect of deployment I would like to improve in the future is the server deployments. For now, I manually compile the build and upload it to Hathora, the server hosting provider. Eventually, I would like to setup a CI/CD pipeline to automatically compile and upload server builds. I have set up a CI/CD pipeline for my web and mobile app Castify, but it would be a new endeavour for me in the realm of game development.

10. Challenges

10.1. Designer vs. Developer

Something specific to my case was that I was playing the role of both the developer and the designer. Again pros and cons of each. Being the same person I could design mechanics I knew were feasible to implement within a certain time frame. But, I also got lost in designer mode while developing. I would implement a feature and think, "I should expand this feature" and then sink 3 more hours into it to end up with no progress. I addressed this by dividing the roles based on phases in the life cycle mode. During the requirements and planning phase, I would focus specifically on game design. Then, I would implement the requirements and make notes of new features I wanted to implement, but I would push those features into the next sprint. This allowed me as a designer to evaluate those ideas from a fresh perspective during the planning phase. When I was able to separate my designer and developer roles I was able to achieve progress at a higher velocity.

10.2. Continuous Learning and Adaptation

This was my third ever project in Unity. The first game I made in Unity was for an FBLA (Future Business Leaders of America) competition my senior year of high school. I participated in the Computer Game and Simulation Programming event. I made a game where you walk around the office from the show "The Office" and answer questions related to FBLA. Not exactly the most fun game, but it was a fun experience. The second game I made was during my sophomore year of college for my astrophysics capstone course. I worked in a team with two other Computer Science students, and we made a Lunar Lander simulator that is still used today by the physics department in astronomy labs. Both were great experiences, but nothing compared to the project I was going to take on.

When I started this project my experience with Unity was novice at best. This made development difficult, especially for a project with such a large scope. It is common knowledge in the game development community that you should not make a multiplayer game for your first game. The following are reasons often stated: increased scope, networking complexity, and difficult testing. Now this technically wasn't my first game in Unity, but it felt like the first time I really set out to understand Unity. As a result, I learned so many things along the way. This made it difficult to adjust the project to these new findings as the project got bigger and bigger. Which is what led to the refactor at the beginning of 2024. Even now, I have been finding plenty of great techniques to better manage the project's architecture but it will be very difficult to retrofit these findings at this stage. The positive takeaway is that I know my next game will have a much improved architecture.

10.3. Prototyping for Understanding

One thing I would do different is create throwaway prototypes more often. During the *State Management and Synchronization* evolution I created a simple multiplayer space shooter based on the arcade game asteroids. Building it allowed me to understand how the server authoritative networking worked. It was a simple game, but it provided me with so much

insight into how I should implement it in the main project. I think next time I would try to create the game in such a way that I can have these prototypes in the game. Let me explain. I would try to make the game very modular and be able to construct small demos of certain features independently of the entire game. This way I could get the benefits of prototyping without having to throw away the prototype. Instead the scripts and features would already be in the game just separated into components that I could add to the existing game. so in broad strokes I would try to improve the project architecture from the beginning to promote modularity and single responsibility.

11. Conclusions and Future Work

11.1. Project Summary

"Squirrels Gone Nuts" set out with two primary objectives: develop an innovative and engaging game, and apply software engineering principles to independent game development. The project resulted in a unique 2D multiplayer arena shooter where players control agile squirrels wielding nut-themed weapons. The game distinguished itself through inventive movement mechanics, such as sliding and shooting recoil, as well as a dynamic camera system that rotates to follow player movement on any surface.

Development was not without challenge; ambitious goals led to overscoping and required adjustments in both complexity and timeline. An iterative process was integral to overcoming these hurdles and ensuring a quality final product.

By blending the creative aspects of game design with software engineering principles, the project achieved a delicate balance that resulted in unique gameplay features. The movement system offers an unprecedented freedom of traversal for platforming games. The other game mechanics complement each other to provide a fresh and engaging experience within the arena shooter genre.

Ultimately, "Squirrels Gone Nuts" demonstrates how adaptability and persistence lead to success when working solo on a large-scale project. The knowledge and insights gained from this journey will shape my future projects and innovations in both software and game development.

11.2. Future Work

I mentioned in the Deployment section that I am planning to release the game in November 2024. Before that, I am going to participate in the October Steam Next Fest in hopes of attracting players. Next Fest is a virtual festival hosted by Steam [11] three times a year where developers can show off their unreleased games to millions of players. I am planning on using this as a promotional mechanism for the game. Then in early November, I will be attending the Midwest Game Development Conference (MDEV) in Madison. I attended last year for the first time and it was an excellent experience. This year I am applying for a booth, so players can experience Squirrels Gone Nuts first hand. For demonstrating the game at a booth in a conference, I will need to create a snapshot of the game easily playable in a short session.

There are a few items on my to do list before release. I have broken them down into evolutions the same way I did throughout the project.

1. **Gamemodes**
   I have a few gamemodes planned beyond the standard free-for-all (FFA) gamemode. All the other modes besides FFA will be team based. The team based modes will feature 3 teams of 3 players instead of the standard 2 team dynamic found in other multiplayer games. This adds a layer of strategic depth to the gamemodes and has been proven in practice by the game The Finals. Here are the other modes I have planned:

- Team Deathmatch: The only objective is to eliminate players on the opposing teams. The first team to reach a set amount of eliminations wins the game.
- Capture the Flag: Each team is assigned a flag and a base. Players must capture the opposing teams' flags and bring them to their base. The first team to reach a set amount of captures wins the game.
- King of the Hill: Teams will fight over control of areas around the arena. Score points by having majority control of the "hill". The first team to reach a set amount of points wins the game.
- Capture and Conquer: This is a new mode of my own devising with a few more layers of strategy. The gamemode combines Capture of the Flag and King of the Hill. Each team is assigned a flag and base. There is a center area that represents the "hill". Players must capture the opposing teams' flags and place them on the center hill. When a there is a flag on the hill and a team has majority control of the hill, that team takes points from the team whose flag was captured.

2. **Player Profiles and Stats**
   Player profiles will track plenty of data including kill/death ratios, win/loss records, and progress on weapon skin challenges. This system will enable players to monitor their growth and accomplishments in the game.

3. **Matchmaking and Parties**
   The matchmaking system will employ an algorithm to match players and teams based on skill levels, ensuring competitive and balanced gameplay. Players can form parties with friends, entering a pre-matchmaking lobby to ensure they play on the same team in team-based modes, enhancing the social and strategic aspects of the game.

4. **Bots**
   To support practice and offline play, there will be AI-controlled bots with adjustable difficulty levels. These bots will mimic human-like behaviors. They can be used to fill multiplayer lobbies and will provide a way for players to enjoy the game even if there are not many players online.

5. **Tutorial and Demo Build**
   A tutorial will be available to help new players familiarize themselves with game controls and mechanics. Additionally, a demo version of the game will be made for Steam Next Fest and the MDEV Conference. It will have limited content but enough features to showcase the game's unique elements and mechanics.

11.3. Final Remarks

The goal of this project was to build a 2D, multiplayer, shooter with an emphasis on fast-paced gameplay anchored in a unique movement system. By my judgement, evidenced by this report, this goal was fulfilled. Hopefully the future players of "Squirrels Gone Nuts" agree.

Bibliography

[1] *Beamable Documentation*. Beamable, 2024. URL: https://docs.beamable.com/docs/beamable-overview.

[2] Wikipedia contributors. *Entity component system*. 2024. URL: https://en.wikipedia.org/wiki/Entity_component_system.

[3] *Unity Documentation*. Unity Technologies, 2024. URL: https://docs.unity.com/.

[4] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 2014.

[5] *Fish-Networking Documentation*. FirstGearGames, 2024. URL: https://fish-networking.gitbook.io/docs.

[6] *PlayFlow Cloud Documentation*. PlayFlow, LLC, 2024. URL: https://docs.playflowcloud.com/.

[7] *Azure PlayFab Documentation*. Microsoft, 2024. URL: https://learn.microsoft.com/en-us/gaming/playfab/.

[8] *Hathora Documentation*. Hathora, Inc., 2024. URL: https://hathora.dev/docs.

[9] Alexander Brazie. *Designing The Core Gameplay Loop: A Beginner's Guide*. 2024. URL: https://gamedesignskills.com/game-design/core-loops-in-gameplay/#:~:text=What%20is%20a%20gameplay%20loop,aim%2C%20fire%2C%20advance%2C%20repeat.

[10] *itch.io Homepage*. itch corp, 2024. URL: https://itch.io/.

[11] *Steam Homepage*. Valve Corporation, 2024. URL: https://store.steampowered.com/.