Heady Stuff: Agile Development Practices in Unreal Engine Game Development

Jacob Hreshchyshyn

Ruben Acuña, Committee Director

John Hentges, Committee Member

A Barrett Thesis Submitted in Partial Fulfillment of the Requirements for Software Engineering

May 1, 2022

1. Introduction

This paper will demonstrate that the Agile development process helps to ensure incremental work on an Unreal Engine game project is achieved by presenting a product produced in Unreal Engine along with my experience in utilizing Scrum to facilitate the game's development. Section 2 discusses project goals and motivations for using Agile, using Unreal Engine, and for the choice of genre in the final product. Section 3 contextualizes these goals by presenting the history of Unreal Engine, the novel applications of Unreal Engine, and the use of Unreal Engine in the development of *Heady Stuff*. Section 4 presents findings from the project's development by describing my use of Agile and by presenting the steps taken in learning Unreal Engine. Section 4 continues by highlighting important development considerations in the use of Blueprints, C++, and HLSL in Unreal Engine. The section ends with the presentation of project feedback, its incorporation in the final product, and the resources used to assist development. Section 5 compares the workflow, help resources, and applications of Unreal Engine with those of Unity, another highly popular game engine. Lastly, Section 6 performs a post-mortem on the overall development process by considering how well Agile development processes were upheld along with how much of the original plans in the Design Document was present in the final product. Additionally, the section presents the major challenges encountered during project development. These challenges will help in proposing possible best practices for game development in Unreal Engine.

2. Project Goals

2a. Why Agile Development

Agile development can best be expressed in the Agile Manifesto, which highlights the value of "Individuals and interactions over processes and tools, Working software over comprehensive documentation, Customer collaboration over contract negotiation, Responding to change over following a plan" (Beck et al. 2001).[1] Agile development methodologies encapsulate a wide range of software development practices that focus on the rapid development of good software while deemphasizing the need to produce documentation. Most importantly, these methodologies foster adaptivity and relationships between developers and other stakeholders, like customers of the developed software. These methodologies incorporate short work cycles, usually 2-3 weeks, to facilitate such rapid development along with revisions of the working product due to the iterative nature of these methodologies (Wrike 2022).[2] Due to strict time constraints for developing a complete game along with the need to produce a game with an appropriate level of polish with respect to playability, I deemed it logical to utilize a development methodology that would best ensure that the project be completed within the time frame while also promoting the need to satisfy the customer (the game's playtesters) and other project stakeholders (including my committee).

1. Kent Beck, et al., 2001, "Manifesto for Agile Software Development," Manifesto for Agile Software Development, https://agilemanifesto.org/.

2. Wrike, 2022, "What Is Agile Methodology in Project Management?" Wrike, https://www.wrike.com/project-management-guide/faq/what-is-agile-methodology-in-project-management/.

Additionally, I took heavy inspiration from Scrum due to my experience with it in other classes in the Software Engineering curriculum. The scrum.org website provides a comprehensive definition of Scrum and its framework (see Figure 1).

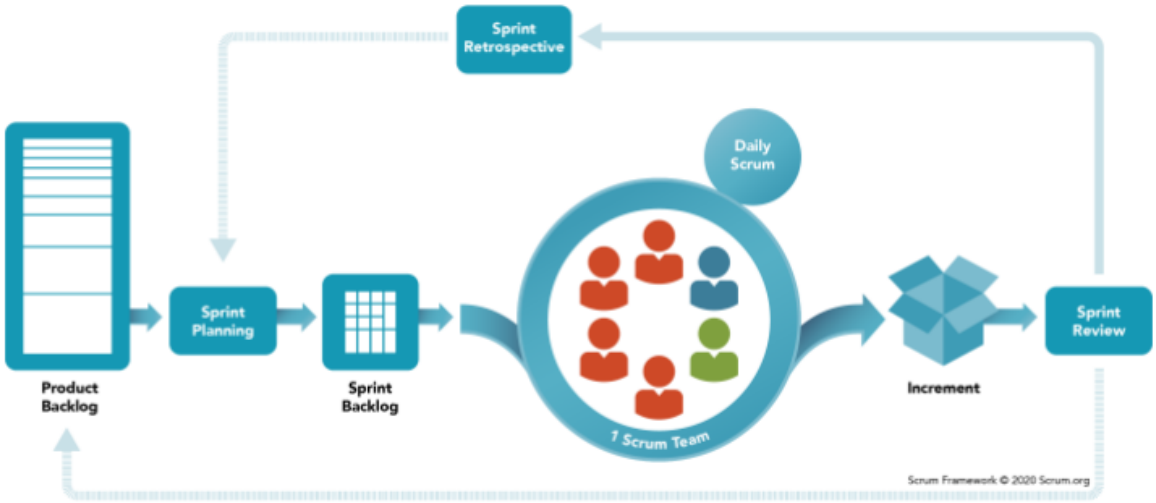


Figure 1. A visual representation of the Scrum framework (Scrum.org 2022).[3]

The site provides the following requirements for Scrum.

In a nutshell, Scrum requires a Scrum Master to foster an environment where:

1. A Product Owner orders the work for a complex problem into a Product Backlog.
2. The Scrum Team turns a selection of the work into an Increment of value during a Sprint.
3. The Scrum Team and its stakeholders inspect the results and adjust for the next Sprint.
4. Repeat (Scrum.org 2022)[4]

3. Scrum.org, 2022, "What is Scrum?" Scrum.org, https://www.scrum.org/resources/what-is-scrum.
4. Scrum.org, 2022, "What is Scrum?" Scrum.org, https://www.scrum.org/resources/what-is-scrum.

The utilization of this framework in previous experiences helped in maintaining constant progress on software projects, especially due to the use of 2-week long Sprints. These Sprints assist in identifying the features that must be implemented to meet each goal, the tasks that must be completed to implement each feature, and the overall workload of the project. Thus, in addition to enabling developers using Scrum to gather feedback on project progress, Sprints not only help encourage constant effort on a software engineering project through rapid development cycles, but also provide a history of the work done on a project. While there exist other Agile techniques that facilitate rapid development, such as Extreme Programming, I decided against these techniques due to my unfamiliarity with their processes and Scrum's higher presence in industry. Additionally, I wanted to determine how well Scrum processes, which I have seen used in industry and in academia, can adapt to a game development context. These are the primary reasons I decided to incorporate Scrum methodologies in the development of this project.

2b. Why Unreal Engine

Primary motivations in using Unreal Engine include my desire to learn Unreal Engine game development through the experience of working with it. This interest stemmed from my previous opportunities in working with game engines and frameworks such as GameMaker Studio 2 (YoYo Games Ltd. 2022),[5] Unity (Unity Technologies 2022),[6] and MonoGame (MonoGame Team 2022).[7] Because I already worked with these tools, I was interested in learning how to use other popular game engines, thereby further expanding my toolset. Seeing the popularity of games made with Unreal Engine, such as

5. YoYo Games Ltd., 2022, "Easily Make Video Games with GameMaker," YoYo Games, https://gamemaker.io/en/gamemaker.
6. Unity Technologies, 2022, "Unity," Unity Real-Time Development Platform | 3D, 2D VR & AR Engine, https://unity.com/.
7. MonoGame Team, 2022, "MonoGame," MonoGame.net, https://www.monogame.net/.

Fortnite, along with the free access to development with Unreal Engine, I decided to capitalize on the Barrett Creative Project opportunity by making a game with Unreal Engine.

Additionally, Unreal Engine has many appealing qualities, some of which will be expounded upon further in the discussion on Unreal Engine's history and its use within and outside of the game development industry. However, I can indicate here that I was already aware of the impressive visual capabilities of Unreal Engine due to my exposure of Epic Games' many promotional videos (Unreal Engine 2022)[8] showcasing the features of their product. Further, I found value in learning how to develop with Unreal Engine due to its use in AAA titles such as Tekken 7 (BANDAI NAMCO Europe 2017),[9] Final Fantasy VII Remake (Minotti 2015),[10] and Kingdom Hearts 3 (Corriea 2014).[11] Thus, the value I saw in learning Unreal Engine comes from my knowledge of its use in the AAA game industry, which is where I would like to work after the termination of my career at ASU. Since Unreal Engine is a widely used game engine and is popular among independent developers in a similar way that Unity is popular, I would be able to access a wealth of resources to assist in the game development process. These are the primary reasons I decided to work with Unreal Engine in the undertaking of the Barrett Creative Project.

8. Unreal Engine, 2022, "Virtual Production Sizzle Reel 2022 | Unreal Engine," YouTube, https://www.youtube.com/watch?v=_oMH_gv7r60&list=PLZlv_N0_O1gYBVBadlg6vUKXAazMSY2r8.
9. BANDAI NAMCO Europe, 2017, "Tekken 7 - PS4/XB1/PC - Unreal Engine 4 & Tekken Part 1 (Q&A Dev Interviews)," YouTube, https://www.youtube.com/watch?v=vo-FE-nlPXs.
10. Mike Minotti, 2015, "Final Fantasy VII Remake uses Unreal Engine 4 instead of an in-house tool," VentureBeat, https://venturebeat.com/2015/12/07/final-fantasy-vii-remake-using-unreal-engine-4-instead-of-an-in-house-one/.
11. Ray Corriea, 2014, "Kingdoms Hearts 3 switched to Unreal Engine 4 due to rendering difficulties," Polygon, https://www.polygon.com/2014/10/7/6938125/kingdoms-hearts-3-unreal-engine-4.

2c. Why I Made the Game I Made

It is worth discussing why I opted to develop an arcade-style platformer with an FPS perspective. Further details are expressed in section 10.c in the Design Document for *Heady Stuff*. The games referenced for this project are *Beat Saber*, a VR game in which the player slices incoming cubes to the beat of the music, *Teardown*, a sandbox heist game featuring a fully destructible environment, *Super Mario Galaxy*, a space-themed 3D platformer, and *Uncharted 2: Among Thieves*, an action-adventure game about treasure-hunting. This collection of reference games demonstrates interesting elements that can be drawn upon for a new experience. From *Beat Saber*, I borrowed the idea of reacting to hazards that are flying towards the player. I enjoyed this mechanic because it introduced a frantic element to the gameplay that would be appropriate for an arcade-style game involving fast reflexes. This also contributed to the decision that the game use an FPS perspective in order to better convey that frantic energy. From *Teardown*, I borrowed the idea of fragmenting objects, which I implemented for the hazards that are destroyed as the player punches them. From *Super Mario Galaxy*, I borrowed the idea of a space theme, which is present in the strange environment the player plays in as well as the black hole hazard that the player must avoid in the game. Lastly, from *Uncharted 2: Among Thieves*, I borrowed the idea of utilizing post-processing techniques to color the view of the player as the player takes damage. I felt that this was a visually appealing and effective means of conveying the idea of getting hurt by incoming hazards and decided that it was worth including in *Heady Stuff*. Ultimately, I picked features from this collection of games because of the enjoyable

experiences I had playing most of them and because I felt that the individual features I selected from these games could be implemented with relative ease, especially since I would be utilizing an engine that features a host of tools for visual effects. Thus, it was through referencing and implementing unique gameplay elements from each of these titles that the game took shape the way it did.

3. The Unreal Engine

3a. The History of Unreal Engine

We will now discuss Unreal Engine, its history, its use within and outside the gaming industry, and how I used the engine. Unreal Engine as we know it today can be traced back to a puzzle game developed by Tim Sweeney in 1991 called *ZZT* (Thomsen 2010).[12] The object-oriented implementation of the game in Turbo Pascal is what laid the framework for the implementation of Unreal Engine and its development process. In an interview with IGN, Tim Sweeney explains this foundation. He says, "*ZZT* served as a conceptual blueprint for Unreal . . . A game engine with a high-productivity, what-you-see-is-what-you-get tools pipeline, bundled with a programming language aimed at simplifying gameplay logic" (Thomsen 2010).[13] Following the creation of *ZZT*, Tim Sweeney with his company, Epic, would produce an early version of the first iteration of Unreal, which they would successfully monetize through licensing deals. Subsequent iterations of Unreal Engine would emerge with major developments in the console market. For example, Unreal Engine 2 was made with the intention of providing multi-platform

12. Mike Thomsen, 2010, "History of the Unreal Engine," IGN, https://www.ign.com/articles/2010/02/23/history-of-the-unreal-engine.
13. Mike Thomsen, 2010, "History of the Unreal Engine," IGN, https://www.ign.com/articles/2010/02/23/history-of-the-unreal-engine.

development on systems like the PlayStation 2, Xbox, and Gamecube (Thomsen 2010).[14] Later, with Unreal Engine 3, Epic focused on tailoring the engine to support development on HD consoles such as the Xbox 360 and PlayStation 3. Unreal Engine 3 would also later offer support for development on mobile platforms to meet the rising popularity of mobile gaming (Thomsen 2010).[15] Unreal Engine 4, in addition to meeting the demands of the next generation of console gaming, sought to improve the development experience of those who are not programmers. One of the most prominent examples of this is UE4's introduction of the Blueprint system, a powerful WYSIWYG scripting system that can be used for implementing logic for in-game actors and shaders (Stafford 2014).[16] In 2021, Unreal Engine 5 was made available, introducing even more improvements in developing large-scale games. These improvements, among others, include Nanite, a virtualized geometry system that eases the process of dealing with LODs in art, Lumen, an advanced lighting system, and the World Partition system, a system that eases the creation of open worlds by mapping sublevels onto a universe (L'Italien 2021).[17]

3b. Novel Applications of Unreal Engine

Unreal Engine has been utilized in a multitude of AAA and independent titles over the years. However, Unreal Engine also has many use cases outside the space of video game development. One of these use cases is in film production, a prominent example being the production of *The Mandalorian* (Farris 2020).[18] John

14. Mike Thomsen, 2010, "History of the Unreal Engine," IGN, https://www.ign.com/articles/2010/02/23/history-of-the-unreal-engine.
15. Mike Thomsen, 2010, "History of the Unreal Engine," IGN, https://www.ign.com/articles/2010/02/23/history-of-the-unreal-engine.
16. Matt Stafford, 2014, "Transitioning from UE3 to UE4," Unreal Engine, https://www.unrealengine.com/en-US/blog/transitioning-from-ue3-to-ue4?sessionInvalidated=true.
17. Ryan L'Italien, 2021, "Unreal Engine 5 | Why 2022 Will Be Released with UE5 | Perforce," Perforce Software, https://www.perforce.com/blog/vcs/unreal-engine-5.
18. Jeff Farris, 2020, "Forging new paths for filmmakers on The Mandalorian," Unreal Engine, https://www.unrealengine.com/en-US/blog/forging-new-paths-for-filmmakers-on-the-mandalorian.

Farris, an Epic Games game engineer and technical lead, details the innovative system Unreal Engine 4 enabled for providing a better sense of interactivity with the imaginary environments present in the television show during production. He says, "By the time shooting began, Unreal Engine was running on four synchronized PCs to drive the pixels on the LED walls in real time" (Farris 2020).[19] The LED walls Farris describes fill up the entire room and visualize the virtual environments rendered in Unreal Engine. Advantages of this type of visualization include how light emanating from the LEDs interact with actors on set realistically, thereby enhancing the visual appeal of the show while also leaving little need for actors to imagine the strange scenarios they have to act out. Farris continues by describing how "three Unreal operators could simultaneously manipulate the virtual scene, lighting, and effects on the walls. . . . This virtual production workflow was used to film more than half of *The Mandalorian* Season 1, enabling the filmmakers to eliminate location shoots, capture a significant amount of complex VFX shots with accurate lighting and reflections in-camera, and iterate on scenes together in real time while on set" (Farris 2020).[20] Thus, the film production of *The Mandalorian* acts as a notable case study in demonstrating the versatility of Unreal Engine beyond the game development industry.

Unreal Engine today has many applications well beyond the domain of entertainment. One of its most notable, and perhaps impactful, applications is in the development of Digital Twins. This topic is well beyond the scope of this paper. However, exploring Unreal Engine's *Pulse* episode on the topic (Unreal Engine 2021)[21] is a great place to start learning about the topic and how Unreal Engine is contributing to the

19. Jeff Farris, 2020, "Forging new paths for filmmakers on The Mandalorian," Unreal Engine, https://www.unrealengine.com/en-US/blog/forging-new-paths-for-filmmakers-on-the-mandalorian.
20. Jeff Farris, 2020, "Forging new paths for filmmakers on The Mandalorian," Unreal Engine, https://www.unrealengine.com/en-US/blog/forging-new-paths-for-filmmakers-on-the-mandalorian.
21. Unreal Engine. 2021. "Digital Twins: Building Cities of the Future | The Pulse | Unreal Engine." YouTube. https://www.youtube.com/watch?v=i12kObBpz-E.

development of Digital Twins. To briefly summarize the episode's discussion, Digital Twins, despite currently having an ambiguous definition, involves the collection of data on things like the built environment of city infrastructure and feeding that data to a visual model. Doing so facilitates the iterative development of that infrastructure in order to best achieve outcomes that improve living conditions, mitigate environmental impacts from city development, and many other applications. Unreal Engine is currently being used to host high-fidelity models of cities like Wellington and Shanghai to achieve this concept.

3c. Use of Unreal Engine for Heady Stuff

For the Barrett Creative Project, I used Unreal Engine 4. I initially attempted to pursue development using C++. However, I transitioned to working primarily with Blueprints due to their usefulness in rapid logic implementation along with the vast quantity of online resources that use the Blueprint WYSIWYG for teaching Unreal Engine game development. I also used Blueprints for developing the shaders used to produce unique lighting effects on certain materials. I was also able to make use of Custom Nodes, which allowed me to define shader logic in High-Level Shader Language (HLSL). A comparison between Blueprints and Custom Nodes will be made later when I describe my overall experience during project development. In addition to these primary means of implementing behavior in the project, I also utilized Unreal Engine's built-in sequencer system, which enabled the animation of a cinematic camera that starts by visualizing the stage on which the player plays from a distance, then spins around until reaching the top of the stage, leading into the start of the main game state (see Figure 2).

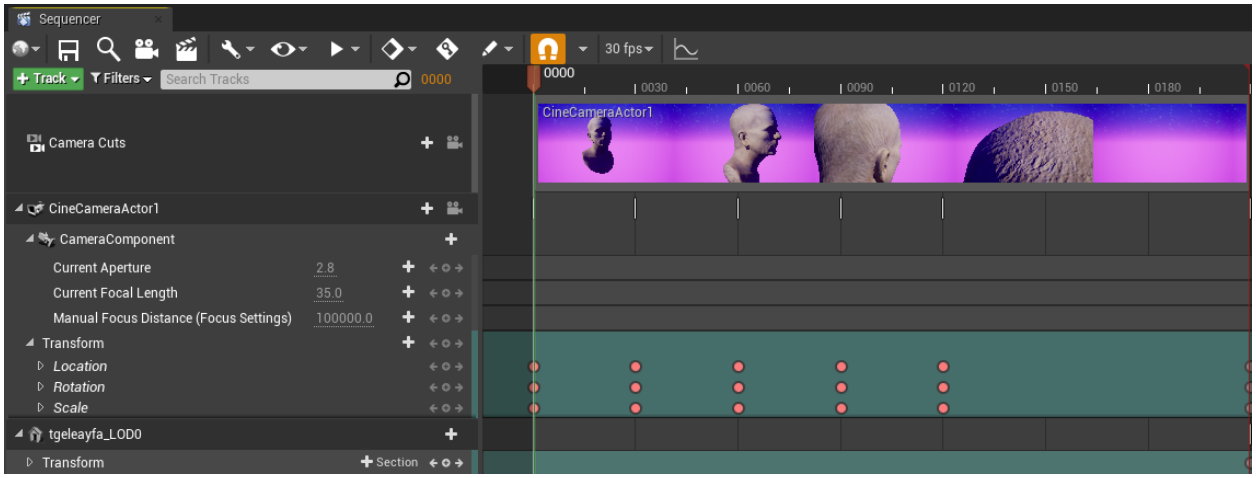


Figure 2. The sequencer, which in this case is used to animate a cinematic camera that spins around the stage, represented by a massive head.

Lastly, I made use of Unreal Engine's built-in widget editor. Widgets, which are canvases that display UI elements, help represent game elements such as HUDs or menus. The widget editor allows the user to drag visual elements, like buttons or text boxes, onto a canvas and anchor them to certain locations on the canvas to assist in properly rendering visual elements on screens of varying resolutions. Events can then be assigned to these visual elements to enable interactivity with the developed widget.

4. Project Development

4a. Agile in Practice

We will now examine my overall experience in developing *Heady Stuff*, starting with my employment of good software development processes like version control and Scrum. The Scrum development process focuses on meeting certain development goals within short development cycles called Sprints. These development goals are broken down into

User Stories (e.g. “As a Player, I want to be able to punch incoming hazards so that I can do more to avoid them besides running away”), which represent the desired outcomes of a Sprint and are stored in a backlog (see Figure 1). A Scrum process would feature both a product backlog, which stores all project User Stories, and a Sprint backlog, which contains a subset of the User Stories from the product backlog. These Sprint backlog User Stories are meant to be completed by the end of the Sprint increment. During a Sprint, User Stories are displayed on a task board (See Figure 3) that presents tasks associated with each User Story as well as the state of task completion (e.g. To-Do, Doing, Done).

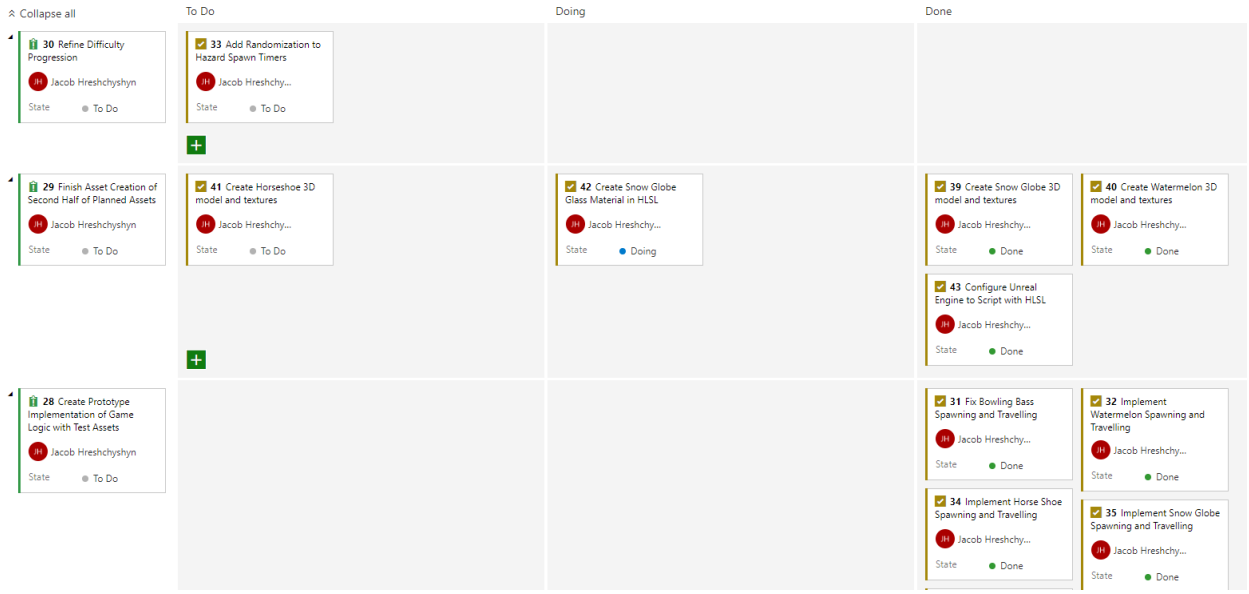


Figure 3. Example of a Sprint task board in Azure DevOps. The green boxes at the left represent User Stories. The yellow boxes represent tasks that correspond to User Stories in the same row. The tasks are separated by the completion states To-Do, Doing, and Done.

In order to best incorporate such a development workflow into my project, I originally considered using Taiga (Taiga Agile, LLC. 2022),[22] an Agile project management tool that I previously used. Taiga would have been a good tool for this project since it easily provides

22. Taiga Agile, LLC., 2022, “Taiga,” Taiga: Your opensource agile project management software, https://www.taiga.io/.

all the Scrum-related techniques described earlier. This led to my initial decision to use Taiga for this project.

On the other hand, there were difficulties in deciding on a version control technique for the Unreal Engine project. One of the main issues with versioning any Unreal Engine project is dealing with large binary files, which would normally be too large for version control tools like Git to process and record a development history. Additionally, I wanted to make use of Unreal Engine 4's built-in version control integration tool, which would link up with the version control system being used to commit changes and store those changes on a remote repository. This led me to investigate solutions like Perforce (Perforce Software, Inc. 2019),[23] which I ultimately decided would be too complicated to work with and integrate with my own project.

Eventually, I came across two development tools that would prove useful in integrating both version control and Scrum methodologies in the development of *Heady Stuff*. The first of these tools is Git LFS (Software Freedom Conservancy 2022),[24] an extension of Git that enables the versioning of large files. This would prove effective in versioning the large files produced during project development. Additionally, because Git LFS is an extension of Git, I was able to apply my previous experience working with Git in versioning *Heady Stuff*, which meant that I would not need to spend an excessive amount of time learning a new versioning system. Further, using Git meant that I would be able to utilize more familiar remote storage solutions. I initially considered using GitHub (GitHub, Inc. 2022)[25] as such a remote storage solution since I have used this for many earlier software development projects. However, due to my recent acceptance as a

23. Perforce Software, Inc., 2019, "How to Use Unreal Engine 4 | Tutorial | Perforce," Perforce Software, https://www.perforce.com/blog/vcs/how-use-unreal-engine-4-perforce.
24. Software Freedom Conservancy, 2022, "Git Large File Storage." Git Large File Storage | Git Large File Storage (LFS) replaces large files such as audio samples, videos, datasets, and graphics with text pointers inside Git, while storing the file contents on a remote server like GitHub.com or GitHub Enterprise, https://git-lfs.github.com/.
25. GitHub, Inc., 2022, "About · GitHub," GitHub. https://github.com/about.

Full-Stack Apprentice at a local software development company, I found Azure DevOps (Microsoft 2022)[26] as the second development tool that would easily merge version control with Scrum methodologies for *Heady Stuff*. Not only does Azure DevOps enable the hosting of large Git repositories, it also incorporates the same Scrum tools that Taiga does, including backlogs, task boards, burndown charts, etc. Thus, I decided to adopt the combination of Git LFS, which enabled the versioning of *Heady Stuff* as a Git repository, with Azure DevOps, which stored that Git repository remotely and facilitated Scrum development methodologies. In practice, the combination proved to be effective in achieving project goals and recording project progress for my committee to monitor.

4b. Learning Unreal Engine

Like the process of determining how to employ good software development practices in this project, there were numerous challenges in learning how to use Unreal Engine itself. One of these challenges was the determination of an appropriate workflow for implementing game logic to game entities using Blueprints, C++ and shaders.

One of the primary means of implementing game logic in Unreal Engine is through the use of Blueprints. Blueprints are visual representations of the logic of a game object, or actor, in Unreal Engine (see Figure 4).

26. Microsoft, 2022, "Azure DevOps Services," Microsoft Azure, https://azure.microsoft.com/en-us/services/devops/#overview.

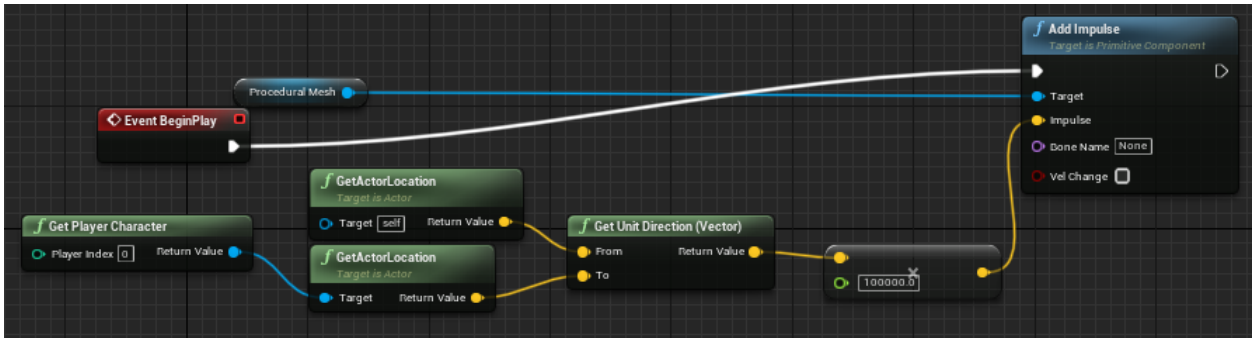


Figure 4. Part of a Blueprint for the black hole hazard in *Heady Stuff*. The Blueprint contains nodes that represent game events like when play begins for the actor (Event BeginPlay), and logic nodes, such as Add Impulse, which applies a specified force towards a Target object, the Procedural Mesh of the actor.

Blueprints are built in Unreal Engine's Blueprint editor. Logic nodes are created either by dragging from inputs or outputs of other nodes or by right-clicking, then selecting the node with the desired behavior. While it can be difficult to do low-level configuration with Blueprint nodes, they are very useful for rapidly generating working game logic, making them ideal for prototyping and incremental development.

In addition to Blueprints, Unreal Engine 4 also enables more experienced programmers to utilize C++ to create actors with their corresponding components and behaviors. This presence of these options is one of the reasons that I found Unreal Engine 4 to have a steeper learning curve than other game engines I worked with in the past, like Unity and GameMaker Studio 2. Similar to Unreal Engine 4, GameMaker Studio 2 allows users to implement game logic using either their own drag-and-drop tool or using GameMaker Language (GML). However, the inherent verbosity of C++ along with the difficulty in acquiring learning resources in developing with C++ in Unreal makes Unreal a more difficult learning experience.

Aside from the complexity introduced by C++, Unreal Engine 4 complicates the development workflow by requiring Blueprints to be used when initializing game objects in levels (there is no null Blueprint node). This means that, after creating and implementing a C++ base class, a Blueprint class that extends the C++ class must be created if it is to be used in the scene. This can lead to complications when adding additional features to that same game object since logic can be introduced either on top of the Blueprint that extends the C++ class or within the C++ class itself, meaning that the developer must now keep track of two different development environments, one for Blueprints and one for C++. Fortunately, there exists some customization in the development environment one chooses for implementing the C++ code. I initially began working with Visual Studio (Microsoft 2022)[27] to handle the C++ compilation and development, but transitioned to Rider for Unreal Engine (JetBrains s.r.o. 2022),[28] an IDE for C++ development in Unreal Engine, due to its faster compilation times and because it was less bloated than Visual Studio.

There were also challenges in learning how to effectively integrate HLSL code in shader development. Part of the problem with Unreal Engine 4's default means of implementing HLSL code for a shader material is the development environment provided for writing HLSL. Like most game objects in Unreal, shader materials are implemented using their own set of Blueprint nodes that encode a variety of effects, like Fresnel effects, etc. In order for a developer to implement their own HLSL shader, the developer must make use of the Custom Node. The problem with making use of the Custom Node is that the development of HLSL code takes place in a small text box under the attributes section of the selected Custom Node (see Figure 5).

27. Microsoft, 2022, "Visual Studio," Visual Studio: IDE and Code Editor for Software Developers and Teams, https://visualstudio.microsoft.com/.
28. JetBrains s.r.o., 2022, "Rider for Unreal Engine," JetBrains, https://www.jetbrains.com/lp/rider-unreal/.

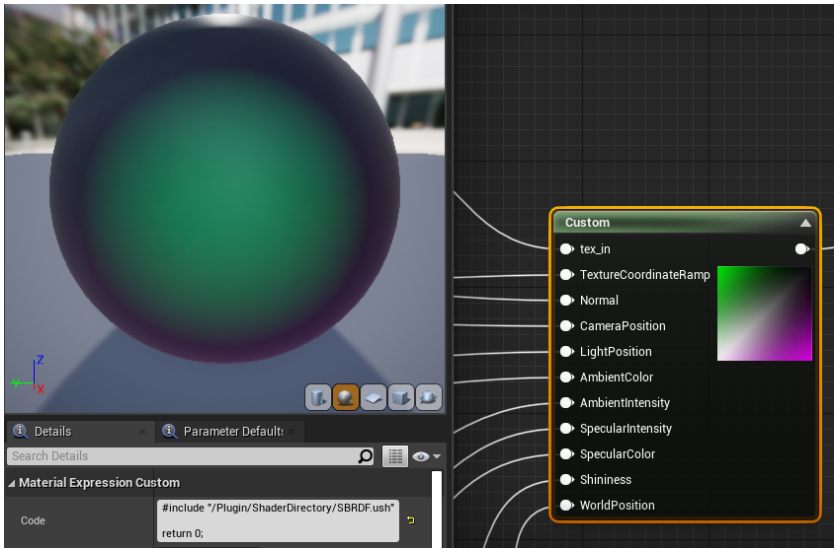


Figure 5. Shader development with the Custom Node. HLSL code is normally written in the small Code field.

This makes it rather difficult to incorporate one's own custom HLSL code in a project unless the code is somewhat short. To circumvent this issue, configuration can be performed to enable the importation of .ush files, which are shader files that can be processed by Unreal Engine. These can contain the HLSL code originally written in the Custom Node text box. This allows the developer to modify their HLSL code in a more robust editor like Visual Studio Code (Microsoft 2022),[29] which is what I did after referencing Baker's video on the topic (Baker 2021).[30] Once again, this produces two separate development environments for implementing shader materials, potentially complicating the development of shaders. However, as before, this offers more customizations in the development process, enabling the developer to optimize their development workflow in other ways. For example, Visual Studio Code in HLSL development allowed me to make use of a plugin that assists in linting the HLSL code I develop.

29. Microsoft, 2022, "Visual Studio Code," Visual Studio Code - Code Editing, Redefined, https://code.visualstudio.com/.
30. Alessa Baker, 2021, "Writing HLSL Code OUTSIDE The Custom Node in UE4," YouTube, https://www.youtube.com/watch?v=V3BVsYV7ge0.

4c. Additional Blueprint, C++, and HLSL Development Considerations

There exist additional implementation considerations to make when choosing between Blueprints and HLSL Custom Nodes for shader material development, starting with the disadvantages of utilizing Blueprints alone for shader development. To help address these considerations, examinations of *Heady Stuff's* implementations of its black hole hazard (see section 6d.) shader and SBRDF (NVIDIA Corporation 2007)[31] shader will take place.

One of the primary disadvantages of Blueprints is that it can become very easy for the Blueprints to become disorganized, which can lead to maintainability issues. There is a chance that the Blueprint logic begins to visually manifest the idea of "spaghetti code" (see Figure 6).

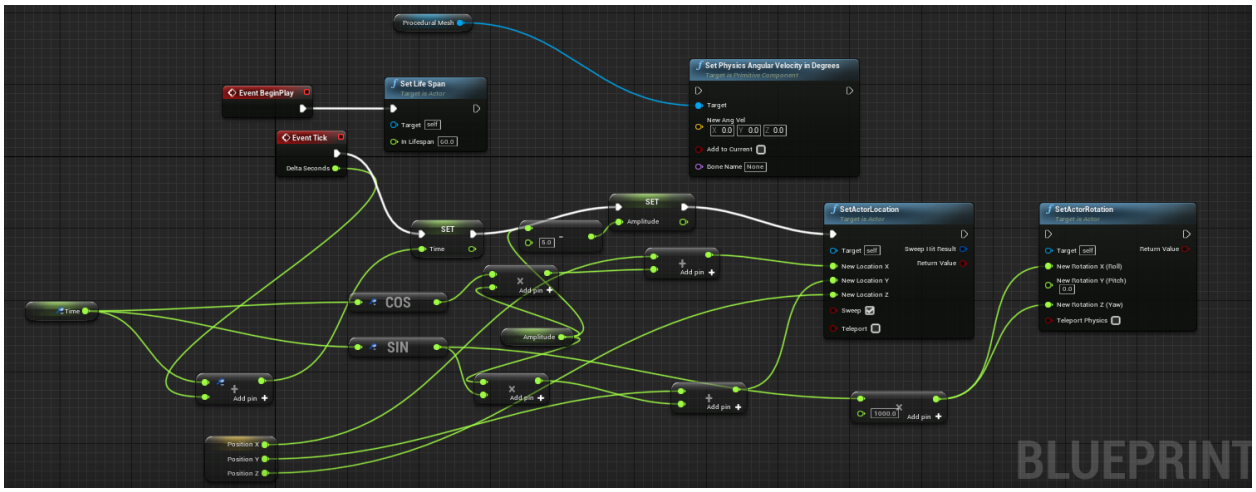


Figure 6. A more literal representation of "spaghetti code" in the movement Blueprint for the horseshoe hazard.

This can result in the temptation to encode that logic manually, whether it be through C++ for actor logic or through HLSL for shader materials. In fact, with shader materials, it is

31. NVIDIA Corporation, 2007, "Chapter 18. Spatial BRDFs," NVIDIA Developer, https://developer.nvidia.com/gpugems/gpugems/part-iii-materials/chapter-18-spatial-brdfs.

possible to encode the entirety of the shader logic in a single custom node. Figure 7 demonstrates this approach with the implementation of the SBRDF shader.

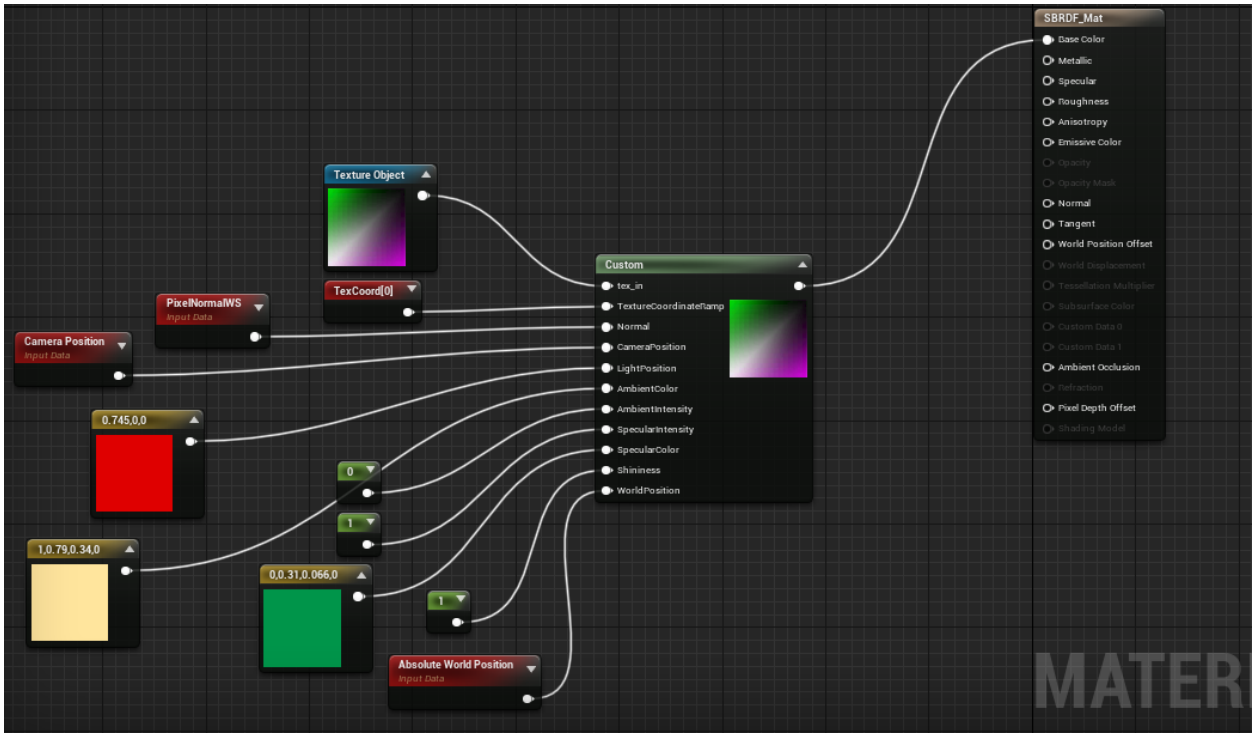


Figure 7. A Custom Node-encoded SBRDF shader material. Uses a texture to vary a lighting effect over a surface.

As demonstrated in my SBRDF shader material implementation, I created a Custom Node with eleven input channels and a single output channel that applies the shader effect to the base color of the material. There is no need for additional processing nodes. I simply need access to the input data for the Custom Node, which I would have needed anyway if I were to implement this shader using Blueprints alone. Thus, from a maintainability and legibility perspective, the use of Custom Nodes seems preferable in shader material development for this example.

However, there exist performance problems with the Custom Node. In one of Epic's Live Training videos on YouTube, Sam Deiter, a Senior Technical Writer at Epic Games, discusses why most shader development in Unreal should not happen in the Custom

Node. He says, “99.999% of the time, all you’re going to need to use is either the built-in material nodes or material functions that come with the Unreal Engine. [The Custom Material Node] is not going to spit out optimized code” (Unreal Engine 2016).[32] The Unreal Engine documentation on the Custom Node provides further information on the optimization processes behind shader compilation and how the Custom Node avoids that optimization.

> Using the custom node prevents constant folding and may use significantly more instructions than an equivalent version done with built in nodes! Constant folding is an optimization that UE4 employs under the hood to reduce shader instruction count when necessary. For example, an expression chain of Sin >Mul by parameter > Add to something can and will be collapsed by UE4 into a single instruction, the final add. This is possible because all of the inputs of that expression (parameter) are constant for the whole draw call, they do not change per-pixel. UE4 cannot collapse anything in a custom node, which can produce less efficient shaders than an equivalent version made out of existing nodes. As a result, it is best to only use the custom node when it gives you access to functionality not possible with the existing nodes. (Epic Games 2022)[33]

The shader material I developed for the black hole hazard in *Heady Stuff* acts as a good example of utilizing functionality not possible with the existing material nodes because of its use of a for-loop (see Figure 8).

32. Unreal Engine, 2016, “Custom Material Node: How to use and create Metaballs | Live Training | Unreal Engine,” YouTube, https://www.youtube.com/watch?v=HaUAfgrZjlU&t=560s.
33. Epic Games, 2022, “Custom Expressions | Unreal Engine Documentation,” Unreal Engine 5 Documentation, https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/Materials/ExpressionReference/Custom/.

```
RaymarchingStruct RM;
float4 Color = 0;
float3 pos = WorldPosition;
float3 Normal = 0;
for(int i = 0; i < MaxSteps; i++)
{
    if(SceneDepth < length(pos - CameraPosition))
    {
        break;
    }
    float d1 = RM.RaymarchingSphere(pos);

    float d2 = SceneDepth - length(pos - CameraPosition);

    float distance = RM.opSmoothUnion(d1, d2, Smooth);

    if(distance < 0.01)
    {
        Color = 1;
        Normal = RM.RaymarchingNormal(pos);
        break;
    }

    pos += CameraVector * 1; // later replaces distance with 1
}
```

Figure 8. A portion of the black hole hazard shader, which utilizes a raymarching technique (Elalaoui 2019)[34] and a for-loop to achieve this effect. Each iteration of the loop intensifies the smooth union effect responsible for merging the black hole object's surface with that of other objects.

While such control structures are possible in actor Blueprint nodes, they are not available in the built-in shader material editor.

One final important consideration to make when choosing to develop shader materials using HLSL in the Custom Node is platform compatibility. *Heady Stuff* is a PC game, so there was little need to worry about potential compatibility issues resulting from shaders being developed in HLSL. However, Ryan Brucks, a Senior Technical Artist at Epic Games, during the same Live Training video, relates how he needed to rewrite some HLSL code for a PlayStation 4 project. He discusses how he needed to use one of Unreal Engine's debug tools to help him find a declared, but uninitialized, integer variable in his HLSL shader, which worked on other platforms except the PlaysStation 4.

34. Marien Elalaoui, 2019, "UE4 Tutorial - Blob with Raymarching," YouTube, https://www.youtube.com/watch?v=grmZ0I5-CgA.

To summarize the appropriate uses of Blueprints, C++, and HLSL, the best way to implement game and shader logic in Unreal Engine 4 is to start by using the built-in development tools. For implementing actor logic, it is a good idea when first creating a project in Unreal Engine 4, to create a C++ project. Doing so would allow the developer to create and edit Blueprint classes using the provided WYSIWYG while also having the opportunity to leverage C++ as a development tool to create logic not easily implementable using Blueprints. This way, one can incorporate a workflow of prototyping game logic with Blueprints, then refining or expanding upon that logic in C++. With respect to shader development, users should be able to create the majority of visual effects they need using the built-in shader methods available in the material editor. I suggest making use of Custom Nodes and HLSL only if the shader being developed requires control structures, like for-loops, that the built-in material nodes cannot provide.[35] For developers expert in developing HLSL code, it might be worthwhile to utilize Custom Nodes as a means of prototyping shaders. However, due to the ease of utilizing existing material nodes to produce shader materials quickly within the material editor, one would still be better off prototyping within the material editor rather than in HLSL. Thus, my recommendation for a workflow in Unreal Engine 4 is to prototype game and shader logic by first using the visual tools, then transition to the advanced development tools to refine and expand upon that logic.

35. Unreal Engine, 2016, "Custom Material Node: How to use and create Metaballs | Live Training | Unreal Engine," YouTube, https://www.youtube.com/watch?v=HaUAfgrZjlU&t=560s.

4d. Receipt of Feedback

In addition to the experience acquired in learning Unreal Engine and working with the development tools provided by Unreal Engine, I also sought to acquire feedback on how enjoyable the game is for people playing the game. For about three quarters of the development life cycle of *Heady Stuff*, the majority of this feedback came from my Creative Project committee. This direct committee feedback was designed to maintain good software development practices while also keeping the committee abreast of project progress. Thus, meetings with my committee acted as Sprint Planning Meetings, in which I received feedback on how others perceived the state of project progress, thereby informing me on what User Stories should be created for the subsequent Sprint.

One of the primary pieces of feedback reiterated many times throughout project development was to focus on the finalization of the game's "critical path", that is, the longest path a player must take to get from the beginning to the end of the game. Up until the ninth Sprint of the project life cycle, my committee members still perceived the state of the game to appear more like a tech demo or physics simulation rather than a game. One of the largest contributions to this deficiency was the lack of a progression system in the way hazards were being instantiated in the game. The game would simply start and a large collection of objects would bombard the player at a constant rate for as long as the player stayed alive. With this in mind, I developed a progression system using timed Blueprint functions to progressively introduce new hazards to the arena as time went on in-game. Couple this progression system with a clear beginning to the game and a clear end state (i.e. an arcade-style Game Over screen), and I had produced a critical path for my game.

Aside from feedback from my committee members, I sought out family members who might be interested in trying *Heady Stuff* in order to understand their first-hand impressions of the game and get a better sense of how I can improve the experience for them. My father proved to be a valuable resource due to his ability to effectively express what felt wrong when playing the game while also appreciating the unique features present in the game. One piece of feedback he provided was his indication of not receiving a good sense of feedback when a thrown punch collides with a hazard. He felt that, while it helps to see the hazard split apart when a punch connects with a hazard, it still felt as though he were punching air. To address this issue, I decided to incorporate various breaking sound effects that would play when broken up by a punch. This helped immensely in communicating the idea of destroying objects with one's fists.

Another piece of feedback my father provided was the fact that the game was somewhat disorienting. Because the player stood in the center of an arena that gets bombarded from all sides by incoming hazards that target the player, it often becomes difficult for the player to react to hazards that approach the player. To mitigate this issue, my father actually suggested the idea of incorporating a minimap feature that helped to visualize those incoming hazards. I had initially planned on a different hazard detection system that would use cursors that visualized three hazards that were closest to the player at any given time. However, the minimap idea was a much better suggestion since it would not only help to visualize all hazards that approached the player, but would also help to visualize the arena on which the player is standing. This is vital since the player is able to walk off the arena, which would result in a Game Over. Thus, the minimap would kill two birds with one stone by visualizing all the vital dangers the player would face

during a play session. This led to me incorporating such a top-down minimap feature, leading to a positive reception from my father.

In addition to this feedback, I incorporated feedback received from a team member from my Capstone project. He had a positive experience with the game since he played the most finalized version. His only criticism was that the tutorial text that showed on screen was too dark and that the text should be white in order to contrast well with the relatively dark environment.

4e. Resources Used in Assisting Development

The last thing worth discussing in my experience working with Unreal Engine is the collection of supplemental assetsI used to assist my development of *Heady Stuff*. One of the notable resources I used was Quixel (Quixel 2022).[36] Quixel is an online collection of development and art assets acquired by Epic Games in 2019. Because of this acquisition, their entire asset library became available for free for any Unreal Engine developer. Quixel allowed me to source the remaining art assets required in the *Heady Stuff* Design Document, namely, assets for the watermelon and horseshoe hazards. I ultimately made use of Quixel in sourcing the remaining assets in order to make up for lost time spent using Blender for asset production. While I had some experience with 3D modeling tools, I realized during development that it would take too long to produce the high-quality models I needed for *Heady Stuff*. Switching to models sourced from Quixel allowed me to incorporate the high-quality assets I needed to move forward with hazard logic implementation. For anyone seeking to develop games in Unreal Engine, they should certainly be informed of Quixel.

36. Quixel, 2022, “We capture the world so you can create your own,” Quixel, https://quixel.com/about.

In addition to art assets like 3D models and textures, I needed a way to create and modify sound effects to better provide a sense of impact for punching hazards, as described earlier. To achieve this, I made use of Freesound (Freesound 2022)[37] and Audacity (Muse Group 2022).[38] Like Quixel, Freesound is a repository containing sound files with various licenses attributed to each sound effect, which allowed me to source sound effects that belonged either in the public domain or that required some attribution. Aside from obtaining sound effects, I needed a way to modify those sound effects so that they could be used effectively in a game environment. Thus, I used Audacity, a free-to-use sound editor that allows users to record and manipulate audio. Using Audacity, I was able to shorten the sound effects I acquired from Freesound so that I could eliminate any unnecessary lingering noise.

In addition to finding these resources, I discovered a wealth of useful resources on YouTube to assist in my development process with Unreal Engine. A good example of this is a raymarching HLSL shader tutorial (Elalaoui 2019)[39] that I utilized to assist in the development of the black hole shader material. Using this tutorial in combination with the application of a starry texture I acquired from Quixel allowed me to produce the black hole shader present in the game, thereby demonstrating the presence of the Unreal Engine community and its efforts to share development knowledge. Another example of this presence can be seen in my implementation of fragmenting hazards in the game. For this, I referenced a tutorial on procedural mesh slicing in Unreal Engine that showed an application for an FPS-style game (UnrealCG 2019).[40] Through this tutorial, I discovered that procedural mesh slicing can be achieved by assigning a procedural mesh component

37. Freesound, 2022, “Freesound,” Freesound - Freesound, https://freesound.org/.
38. Muse Group, 2022, “Credits,” Audacity, https://www.audacityteam.org/about/credits/.
39. Marien Elalaoui, 2019, “UE4 Tutorial - Blob with Raymarching,” YouTube, https://www.youtube.com/watch?v=grmZ0I5-CgA.
40. UnrealCG, 2019, “Mesh Slicing In Unreal Engine,” YouTube, https://www.youtube.com/watch?v=oIdKxYYQBdw.

to a game object, and then by executing a mesh-splitting Blueprint node associated with the procedural mesh component. Because of this discovery, I saved a considerable amount of time that I would have otherwise spent on designing and debugging my own solution to this problem. These experiences in utilizing Unreal Engine features presented through such tutorials help to demonstrate the community of support around Unreal Engine development while also demonstrating instances in which Unreal Engine can assist in providing simple solutions to what would otherwise be difficult and time-consuming problems.

5. Unreal Engine vs. Unity

5a. Workflow and Logic Implementation

Next, we compare what I learned developing with Unreal Engine with my past experience with Unity. We can start by discussing the general workflows I employed during development in these engines. As discussed earlier, my recommended workflow for Unreal Engine is to employ an iterative, prototype-based approach in which general game logic is first implemented using the visual development tools, like Blueprints. Once the groundwork for game logic is implemented and the need arises for additional customizations or optimizations, the developer should transition to the more advanced development tools available, like C++ and HLSL. In my experience working with Unity, I found that I employed a similar iterative, prototype-based approach. However, this

approach was facilitated due to Unity's heavier emphasis on Entity-Component-System (ECS) development.

One can understand ECS as Unity's main means of implementing game logic, which involves the creation of individual C# script components that contain discrete collections of data relating to the state or behaviors of a game object. For example, if I were to develop a simple platformer game, I could start by creating a C# script that was responsible for processing player inputs and moving the player according to those inputs. If I then wanted to develop a system for the player to be able to take damage, I could create separate C# scripts that implement that game logic. I could then take each of the individual C# scripts I created and attach them to a Capsule primitive object as components. (To clarify, Unity allows for the rapid creation of simple shapes called primitives. A Capsule is one of those possible shapes.) This modular approach to implementing game logic differs slightly from Unreal Engine. While components do exist in Unreal Engine and allow for some semblance of modular game development, the representation of that game logic (if working solely with Blueprints) exists entirely in the same behavior chart that is associated with the entire actor, not for individual components. Thus, it is generally simpler to rapidly create game logic in Unity due to the relative ease of an ad-hoc style of game development that the engine provides in its component system.

To illustrate the development workflow of Unity, I used the engine to create a vertical slice of *Heady Stuff* (see Figure 9).

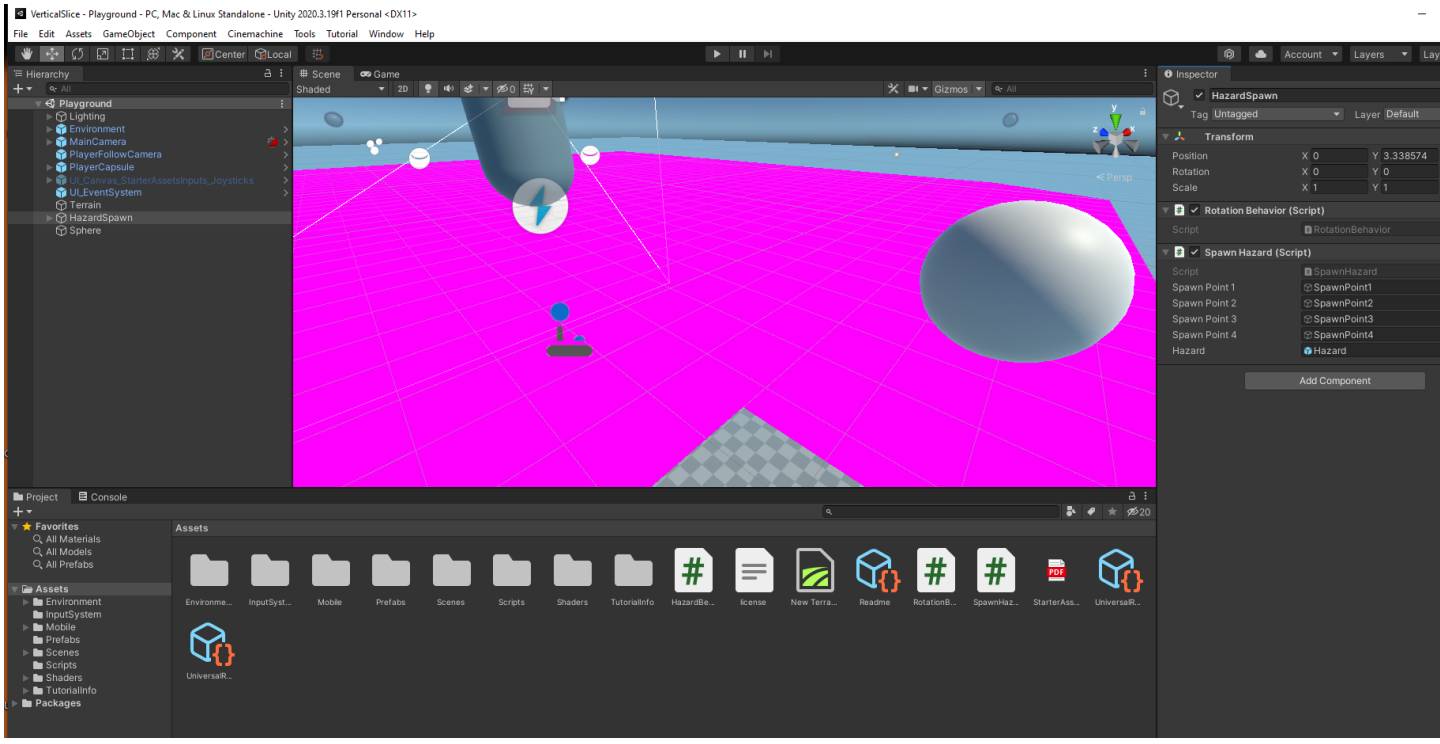


Figure 9. The main workspace of Unity. The bottom left and middle show the game's assets and file structure. The top left shows the collection of game objects in the level, called a Scene. The top right shows the Inspector, which displays attributes and components associated with a selected game object. The top middle shows the edited Scene.

Figure 9 illustrates its ECS qualities in its Inspector, which displays a Transform component and two C# script components attached to a HazardSpawn object. As the two scripts demonstrate, one can add logic modularly to game objects (see Figure 10).

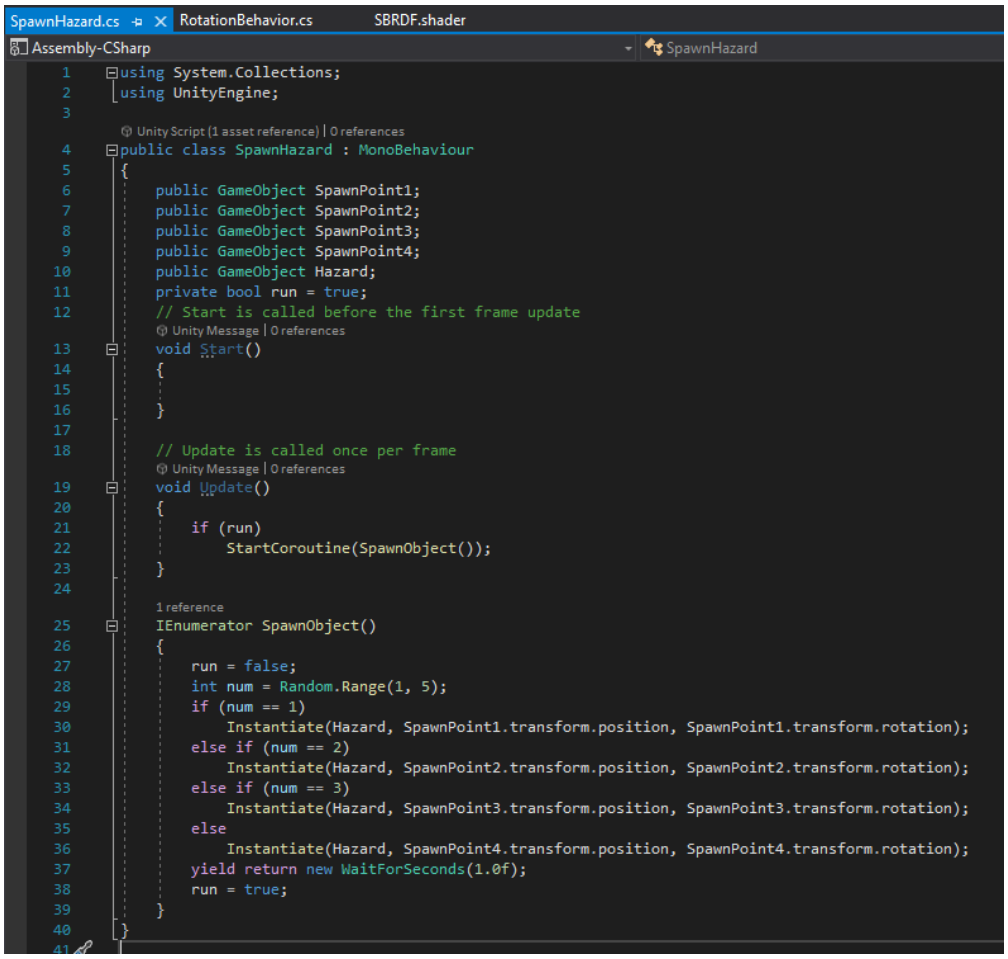


Figure 10. A C# script featuring references to game objects, an Update method that executes once every frame, and a coroutine that spawns hazards at a set time interval.

Figure 10 demonstrates the implementation of game logic in Unity. By default, Unity C# scripts contain a class that implements the MonoBehavior interface, which requires the implementation of the methods Start and Update. Start is a method that behaves like Unreal Engine's Event BeginPlay node and executes game logic at the beginning of play for that game object. Update executes logic once every frame. Using these methods along with one's own variables, custom methods, and coroutines, one is able to modularly develop game logic in Unity. This can be challenging for developers less comfortable with programming.

However, the use of C# as a programming language makes the development experience easier than that of C++ with Unreal Engine (see Figure 11).

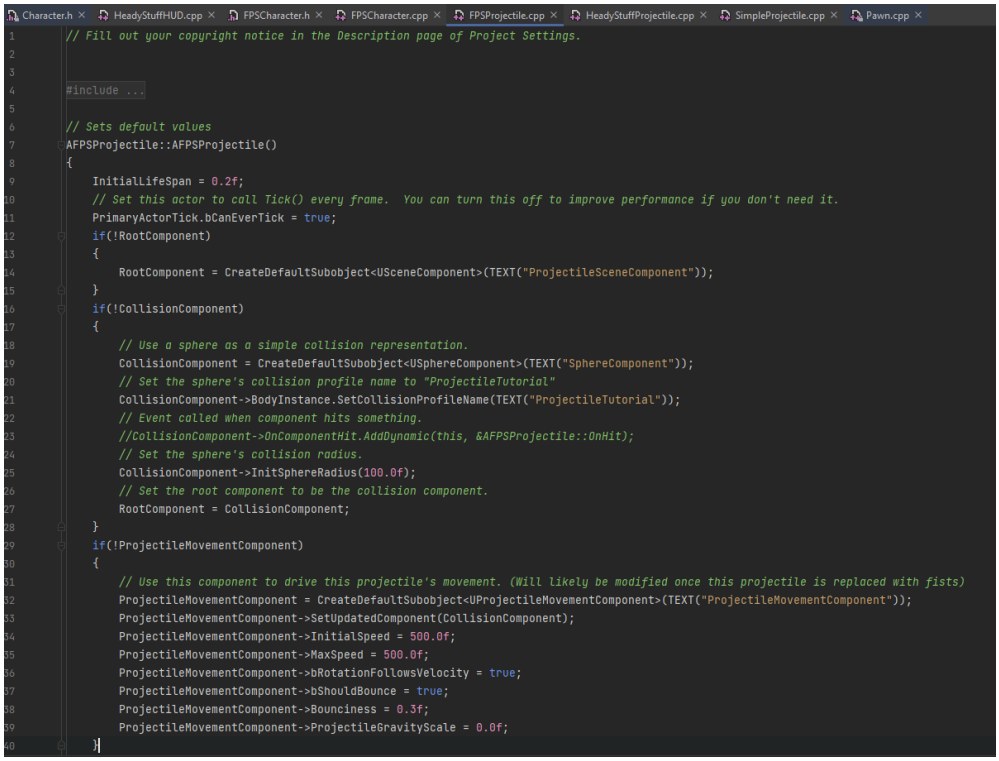


Figure 11. A sample of the FPSProjectile behavior in Unreal Engine's C++ implementation. Notice line 32's creation of a ProjectileMovementComponent, demonstrating a more complicated ECS development process.

Additionally, shader development in Unity is largely achieved by writing HLSL code in Unity's .shader file format (see Figure 12).

rd.cs RotationBehavior.cs SBRDF.shader

neous Files (Global Scope)

```
Shader "CTD/SBRDF"
{
    Properties
    {
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _Ramp2D ("BRDF Ramp", 2D) = "gray" {}
    }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
        #pragma surface surf Ramp
        // Use shader model 3.0 target, to get nicer looking lighting
        #pragma target 3.0

        sampler2D _MainTex;
        sampler2D _Ramp2D;
        struct Input
        {
            float2 uv_MainTex;
        };

        half4 LightingRamp(SurfaceOutput s, half3 lightDir, half3 viewDir, half atten)
        {
            float NdotL = dot(s.Normal, lightDir);
            float NdotE = dot(s.Normal, viewDir);

            float diff = (NdotL * 0.3) + 0.5;
            float2 brdfUV = float2(NdotE * 0.8, diff);
            float3 BRDF = tex2D(_Ramp2D, brdfUV.xy).rgb;

            float4 c;
            c.rgb = BRDF;
            c.a = s.Alpha;
            return c;
        }

        void surf (Input IN, inout SurfaceOutput o)
        {
            half4 c = float4(0.5, 0.5, 0.5, 1);
            o.Albedo = c.rgb;
            o.Alpha = c.a;
        }
        ENDCG
    }
    FallBack "Diffuse"
```

Figure 12. SBRDF shader logic in Unity's .shader file. Processing logic occurs in the LightingRamp function.

Comparing the shader logic in Figure 12 with that of the Unreal Engine project (see Figure 13) shows that the logic is largely the same between the two.

```
struct VertexShaderOutput {
    float4 Position;
    float4 Color;
    float3 Normal;
    float3 WorldPosition;
    float2 TextureCoordinateRamp;
};

VertexShaderOutput output;

// Beginning of Vertex Shader
output.WorldPosition = WorldPosition;
output.Normal = Normal;
output.Color = 0;
output.TextureCoordinateRamp = TextureCoordinateRamp;

// Beginning of Pixel Shader
float3 N = normalize(output.Normal.xyz);//normal
float3 V = normalize(CameraPosition - output.WorldPosition.xyz);//view direction
float3 L = normalize(LightPosition);//light direction
float3 R = reflect(-L, N);

float NdotL = dot(N, L);
float NdotE = dot(N, V);

float diff = (NdotL * 0.3) + 0.5;
float2 brdfUV = float2(NdotE * 0.8, diff);
float3 BRDF = Texture2DSample(tex_in, tex_inSampler, output.TextureCoordinateRamp);
float4 ambient = AmbientColor * AmbientIntensity;
float4 specular = SpecularIntensity * SpecularColor * pow(max(0, dot(V, R)), Shininess);
float4 color = saturate(ambient + float4(BRDF, 0) + specular);
//color.rgb = BRDF;
color.a = 0;
return color;
```

Figure 13. SBRDF shader logic compatible with Unreal Engine and contained in the shader material's Custom Node. The code is comparable to that shown in Figure 12.

The main differences have to do with how data is stored and passed to the necessary processing inputs. Both shaders produce the same results (see Figures 13 and 14).

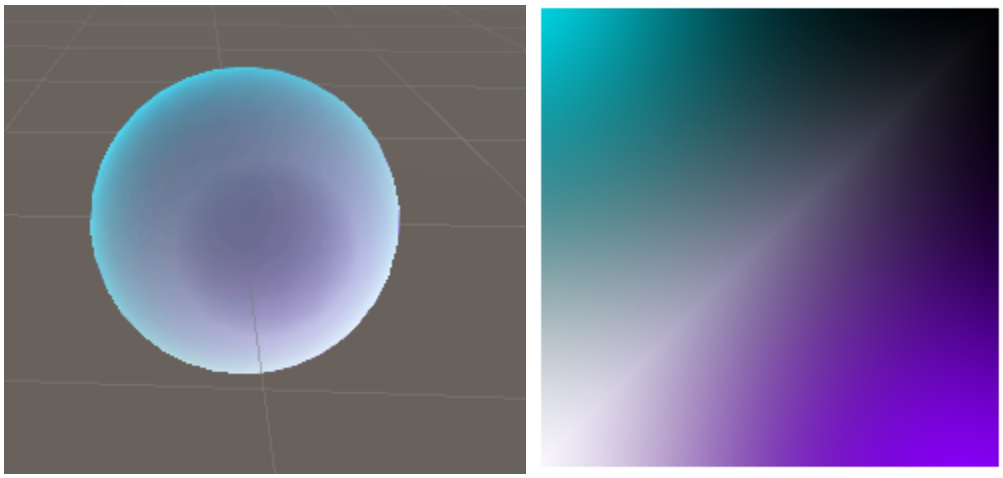


Figure 13. Result of .shader file SBRDF logic in Unity.

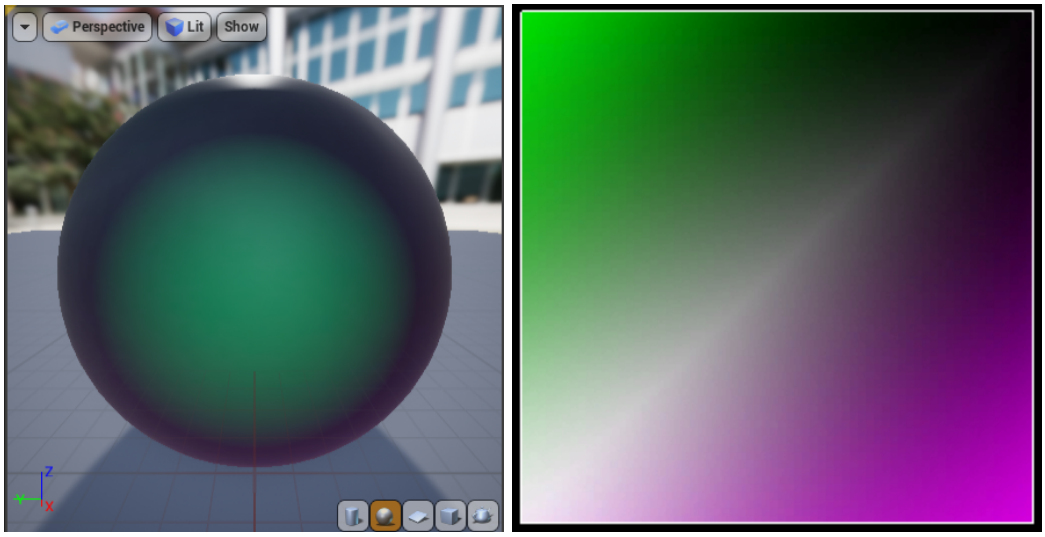


Figure 14. Result of SBRDF shader logic in Unreal. Color is different due to the use of a different sample texture, shown on the right. The processing between the two shaders is identical.

Thus, both engines demonstrate the capabilities necessary for a prototype-based workflow. However, comparing these two engines demonstrates Unity's relative superiority in such a workflow, at least with respect to the workflow's speed.

5b. Resource Availability

In addition to the relative ease of prototyping with Unity, I also found its documentation of resources to be generally better than that for Unreal Engine. While both engines have a vast wealth of help resources online and a community of support surrounding game development with both engines, there are two advantages to Unity's documentation over Unreal Engine that I could uncover. First, Unity often does a better job of providing example code to illustrate the intention behind functions available in the engine's framework. An example of this virtue is present in the documentation for the ForceMode enumeration (Unity Technologies 2022).[41] Not only does this documentation provide a description of what ForceMode is and how it is used, but it also provides a

41. Unity Technologies, 2022, "Unity - Scripting API: ForceMode," Unity - Manual, https://docs.unity3d.com/ScriptReference/ForceMode.html.

lengthy section of sample code demonstrating the different ForceModes and how to use them. This contrasts with Unreal Engine's documentation on the Slice Procedural Mesh node (Epic Games 2022),[42] which was vital in implementing mesh splitting in *Heady Stuff*. The documentation provides a description of the node's inputs and outputs and also provides a screenshot of the node on its own. I think it would have been useful for the documentation to include screenshots of a working use case of the Slice Procedural Mesh node in order to obtain a better understanding of how to use the inputs of the node effectively to produce a result similar to what I sought to develop. I was fortunate to find another tutorial on YouTube that allowed me to leverage this node's features effectively. However, the lack of good examples on the engine's built-in functions does present a weakness in Unreal Engine's documentation compared to Unity's.

The second advantage Unity has in its documentation has to do with the fact that Unity has a single primary source for implementing game logic (C#), whereas Unreal Engine has two (Blueprints and C++). Any assistance a developer might need with the Unity C# API can be found in Unity's official online documentation. For Unreal Engine, Blueprint documentation can also be found online. Additionally, there exist many official tutorials that utilize C++ for the purpose of demonstrating examples like creating an FPS game (Epic Games 2022).[43] However, when it came to individual functions, I found little to no official online documentation. I eventually discovered that documentation on various C++ data structures and functions do exist within the source code of Unreal Engine as comments. This is certainly better than no documentation at all. However, in my experience working with Unity, I cannot recall any instances of finding API documentation

42. Epic Games, 2022, "Slice Procedural Mesh | Unreal Engine Documentation," Unreal Engine 5 Documentation, https://docs.unrealengine.com/4.26/en-US/BlueprintAPI/Components/ProceduralMesh/SliceProceduralMesh/.
43. Epic Games, 2022, "First Person Shooter Tutorial | Unreal Engine Documentation," Unreal Engine 5 Documentation, https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/CPPTutorials/FirstPersonShooter/.

I needed within Unity's source code. While documentation contained in source code is not a bad practice, for convenience purposes, I appreciate Unity's success in its organized documentation of their API online.

5c. Applications of One Engine Over Another

These differences in how documentation is handled helps reinforce my belief that Unreal Engine is moreso designed for use among intermediate or experienced developers rather than novice aspiring game developers. Unity, on the other hand, is better for those who are new to game development and looking for ways to begin that learning process. However, it should be noted that, while both engines are capable of producing AAA games, Unreal Engine is easier to work with when it comes to making games that look very good. As described earlier, the visual capabilities of the engine were one of my motivations to pursue development in Unreal for this creative project. From what I can tell, both engines can be used to reach the same level of graphical fidelity. However, Unreal Engine allows users to have good graphics for their games with initially minimal retooling on the part of the developer. In addition, Unreal Engine contains some built-in features, such as the Slice Procedural Mesh node described earlier (see Figures 15 and 16), which would either require manual implementation of such a feature through vector math or through defining breaks in the mesh beforehand in Unity (see Figures 17 and 18).

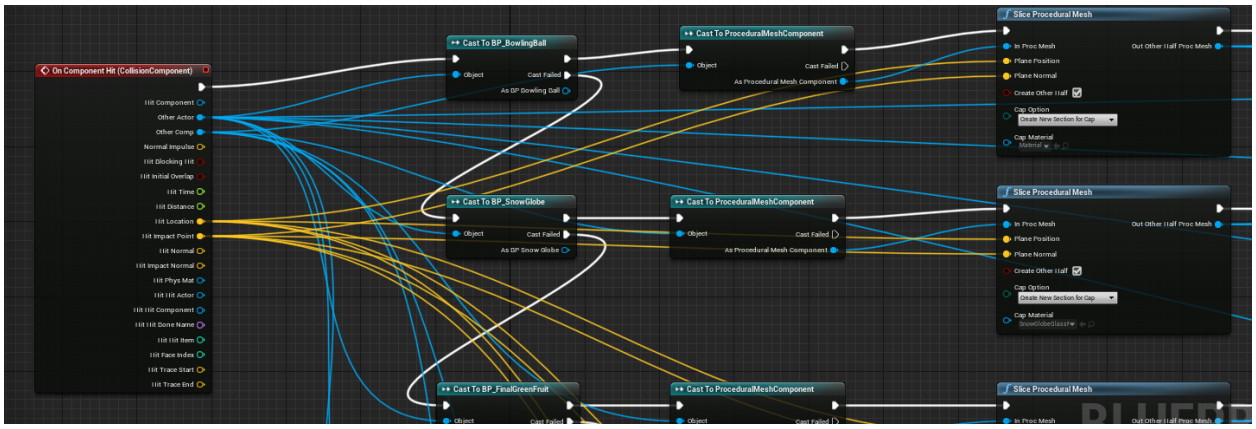


Figure 15. A portion of the mesh slicing logic in Unreal. The bulk of that logic is contained in the provided Slice Procedural Mesh node.

Figure 16. Result of Slice Procedural Mesh node on watermelon hazard.

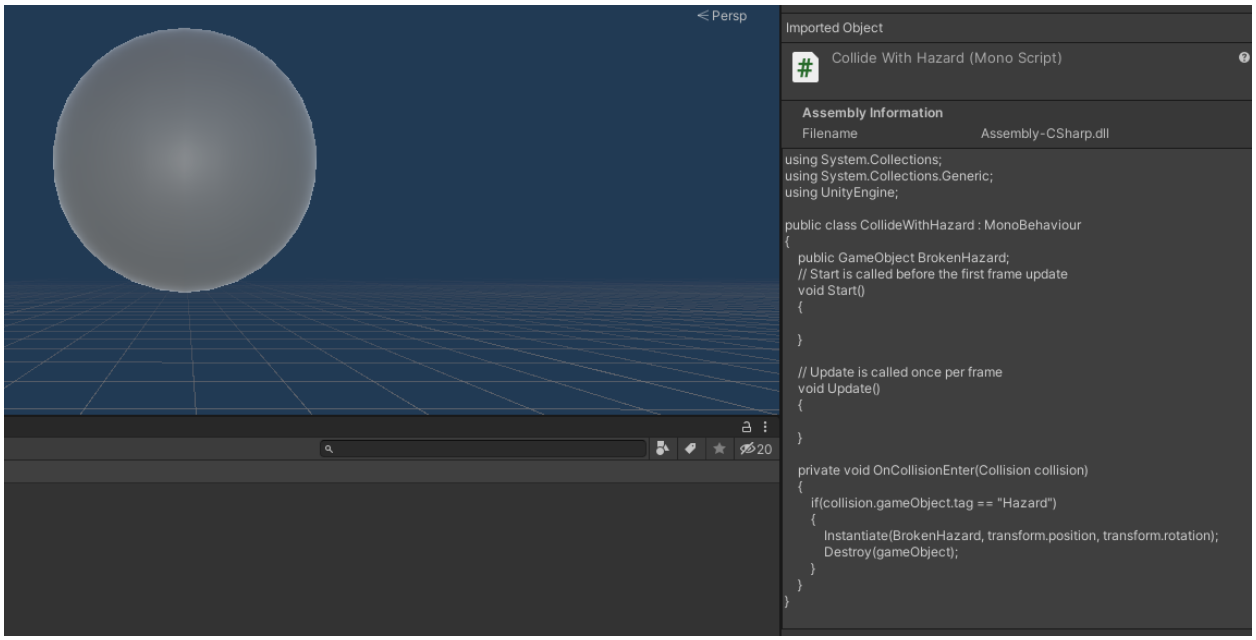


Figure 17. Simple, but less robust, logic for mesh slicing in Unity. On collision with the player's fist, a duplicate of the hazard with pre-defined splits is instantiated at the position of the original hazard. The original hazard is then destroyed.

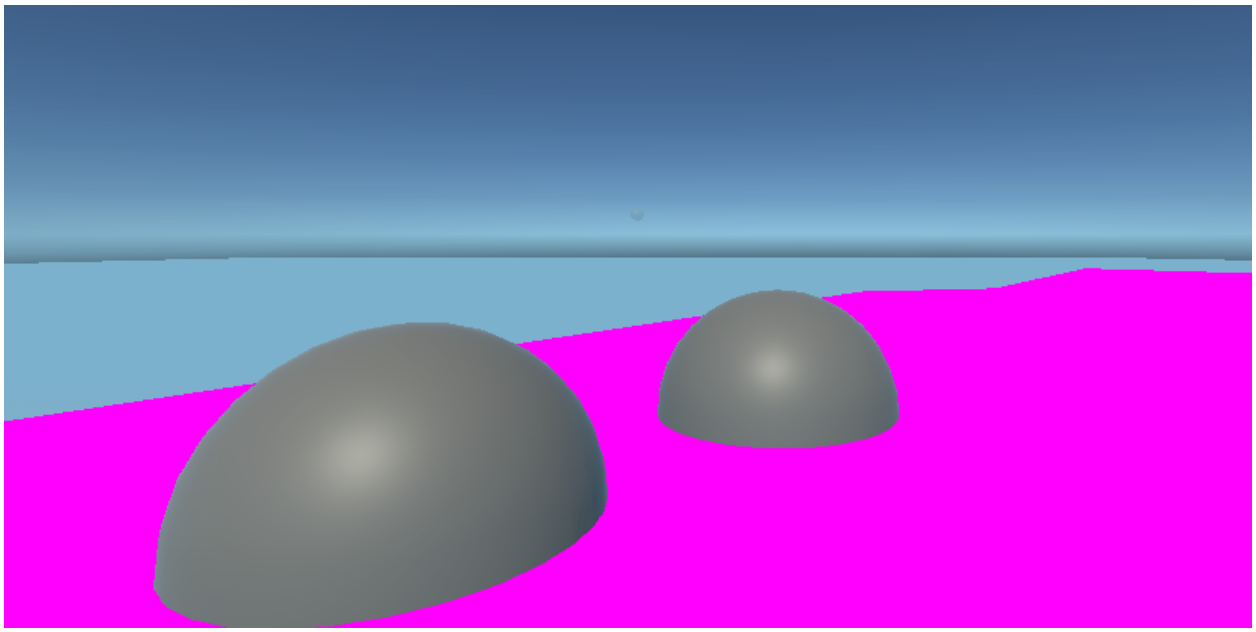


Figure 18. Result of pre-defined splits in Unity vertical slice.

6. Post-Mortem

6a. Adherence to Agile Practices

We will now begin a brief post-mortem on the development of *Heady Stuff*, starting with my adherence to the Scrum development process. Overall, I did well in adhering to this process. As evidenced by the following two burndown charts (see Figures 19 and 20) tracking the completion of tasks throughout the entire project, 79% of all tasks were completed. While it is expected that 100% of all project tasks be completed, some tasks in earlier Sprints were not moved to the Done column on the task board. Some Sprints copied over those uncompleted tasks to new Sprints, in which they were completed during that increment. While not explicitly a Scrum workflow, this was a practice adopted in other software development classes. Thus, despite the uncompleted tasks from earlier increments, the completion of 79% of all project tasks demonstrates a high task completion rate.

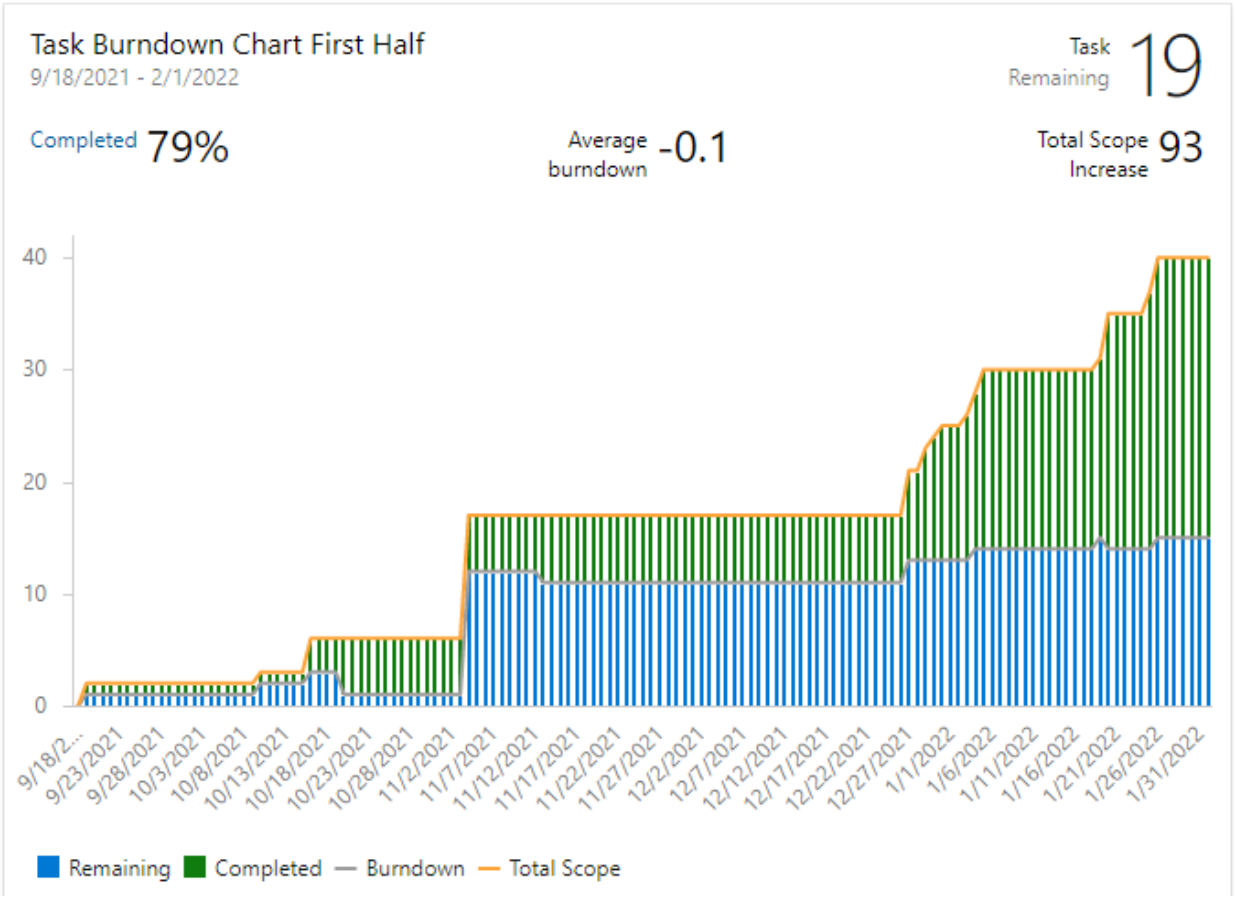


Figure 19. Task completion burndown chart of 9/18/2021 - 2/1/2022 time frame of project.

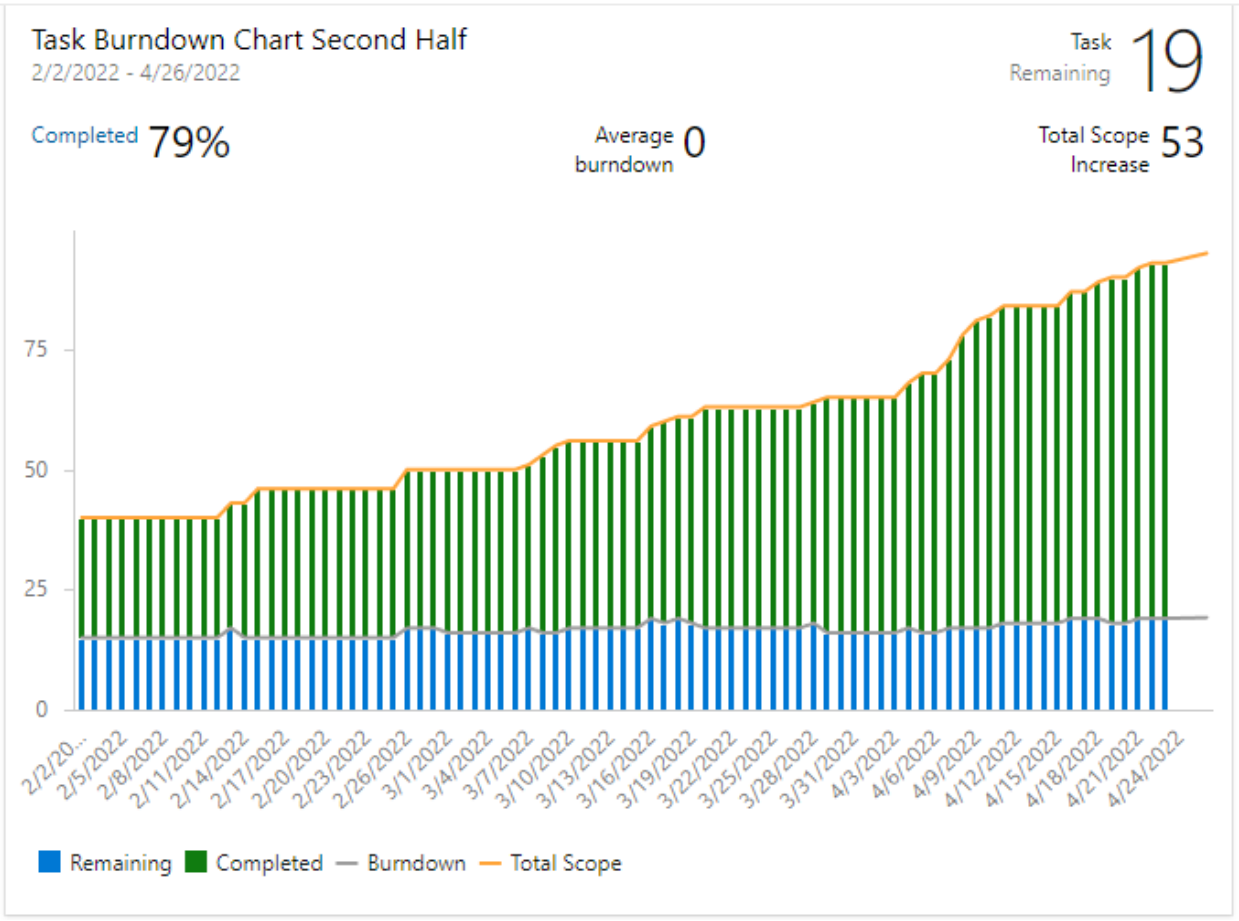


Figure 20. Task completion burndown chart of 2/2/2022 - 4/26/2022 time frame of project.

However, it is worth examining each Sprint's task completion burndown to get a better sense of my overall progress. Below are the task completion burndowns of Sprints 1 - 10 (excluding the final Sprint since the thesis completion is one of that Sprint's tasks). Starting with Sprint 1, the backlog shows a reasonable amount of progress with just one task that was left unfinished (see Figure 21).

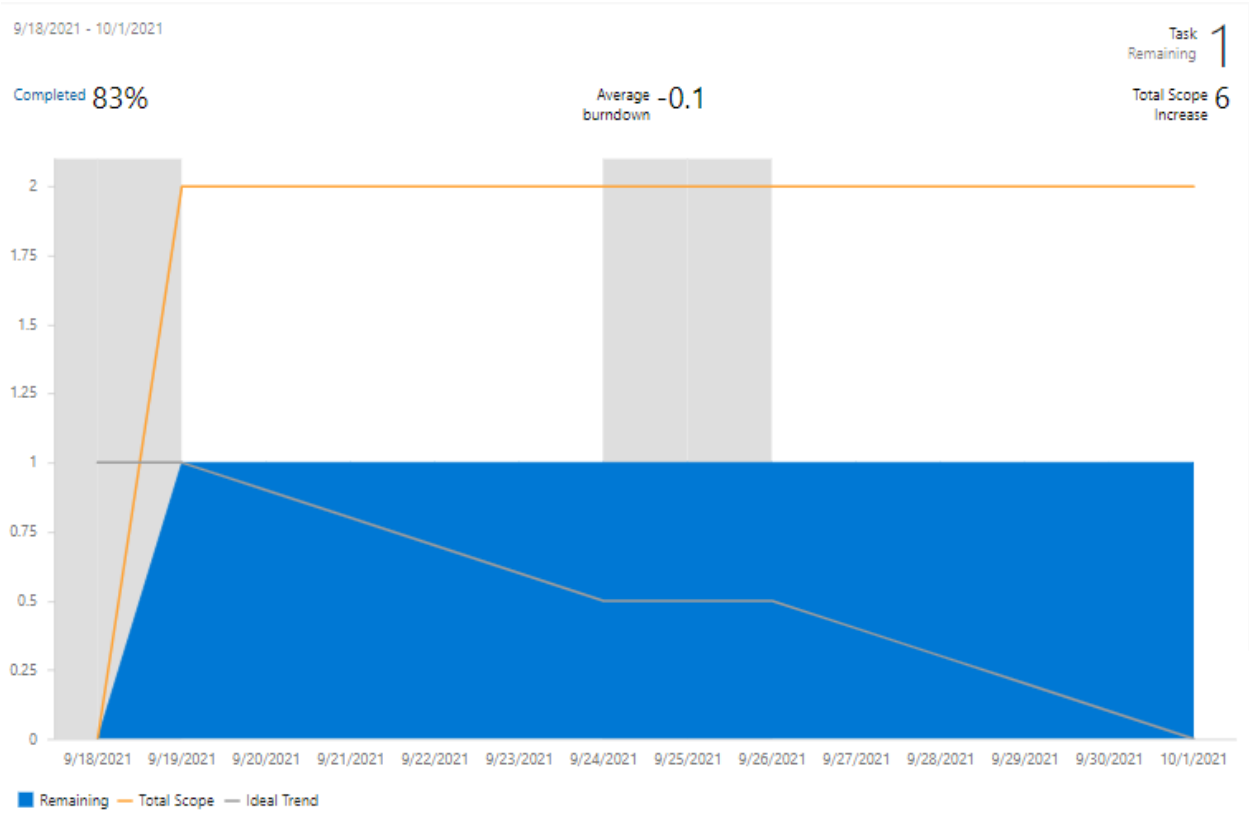


Figure 21. Task completion burndown of Sprint 1 showing 83% task completion with just 1 unfinished task, a good start to the project.

However, Sprints 2 and 3 showed some significant signs of failing to meet Sprint goals with no tasks present during Sprint 2 and only 9% of the tasks being completed on Sprint 3 (see Figures 22 and 23).

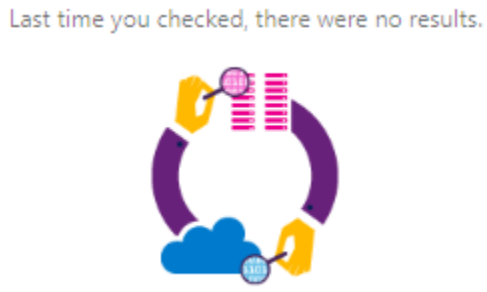


Figure 22. Task completion burndown of Sprint 2. There is no burndown because there is no record of task creation.

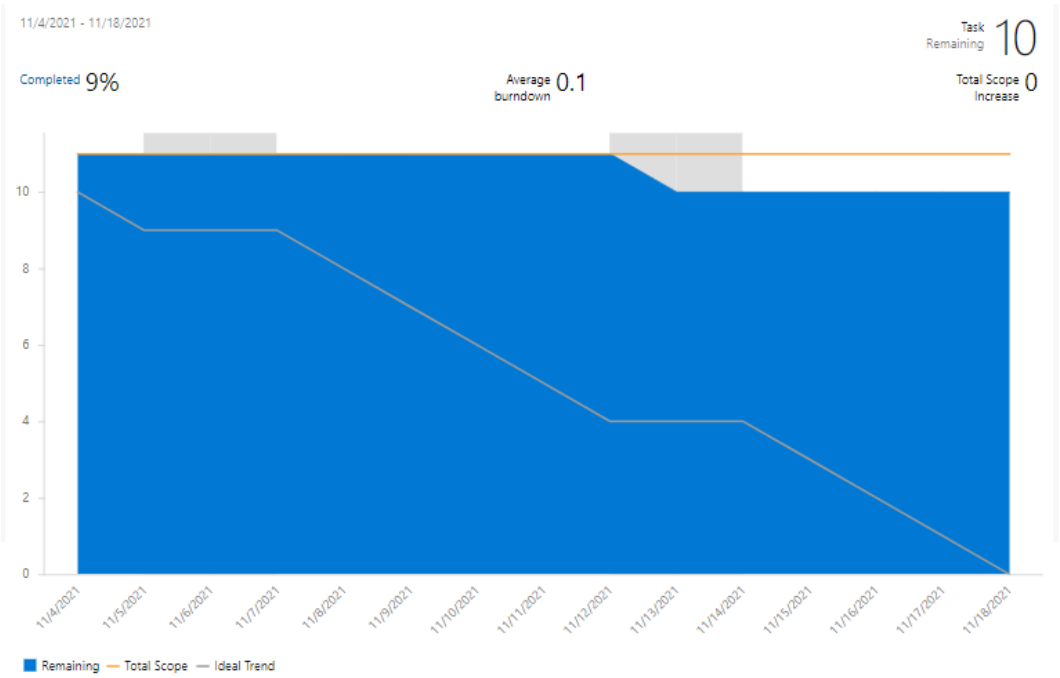


Figure 23. Task completion burndown of Sprint 3 showing 9% task completion with 10 tasks remaining. This poor showing has put the project behind by about 4 weeks.

This is largely due to significant technical challenges I experienced during this portion of the project, which will be examined later in the post-mortem. Ultimately, the lack of progress during these Sprints put me behind schedule by about four weeks. Fortunately, I rebounded with more consistent progress in the following Sprints. Sprint 4 saw an increase in both task scope and task completion that assisted in getting me closer to the initial goals of creating project assets required to represent in-game hazards (see Figure 24).

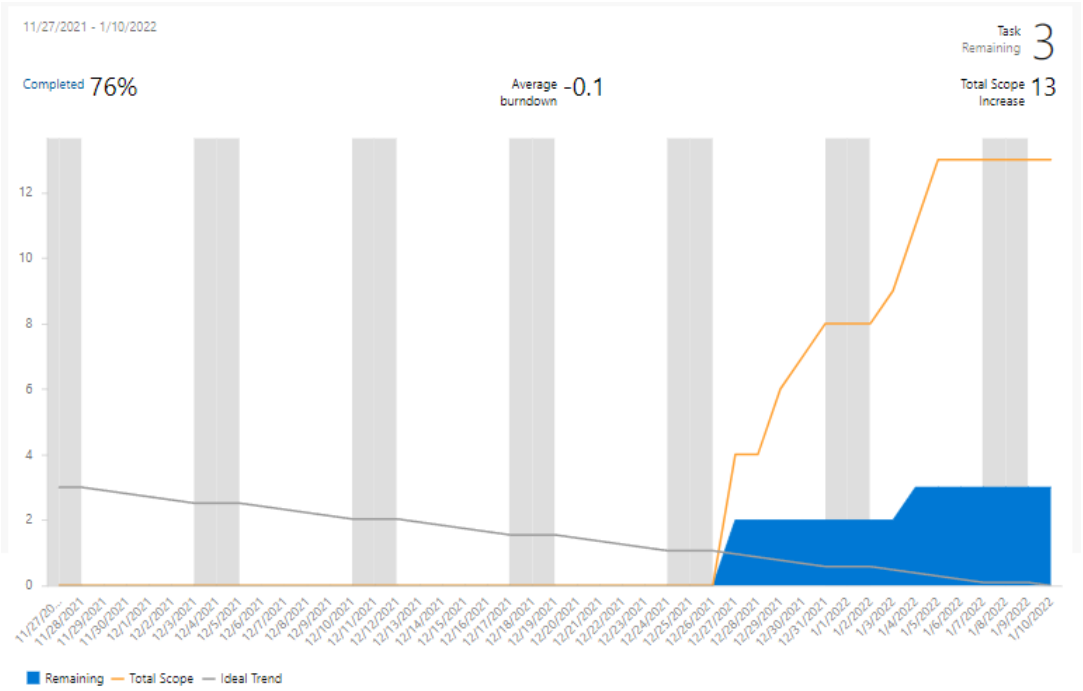


Figure 24. Task completion burndown of Sprint 4 showing 76% task completion with just 3 out of 13 tasks remaining. The project trajectory improves.

I successfully maintained a consistent workflow in the Sprints leading up to Sprint 10. While this helped me make up some ground in the time lost during Sprints 2 and 3, there were still numerous polishing issues that needed to be addressed. Thus, Sprint 10 marks the Sprint with the highest task scope in the project with 19 total tasks, of which 89% were completed (see Figure 25).

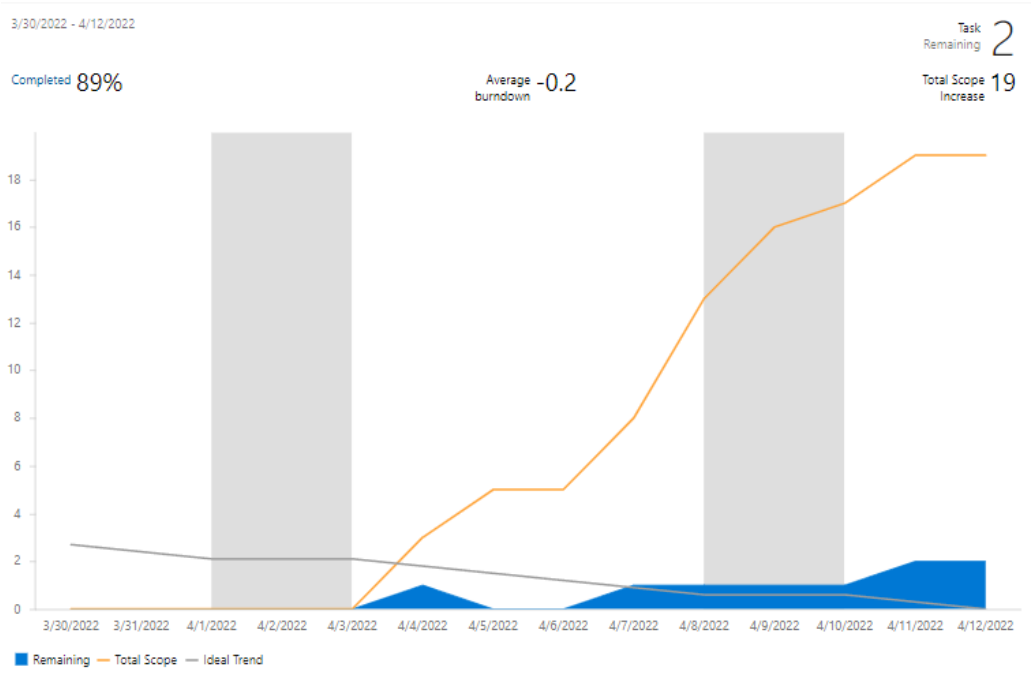


Figure 25. Task completion burndown of Sprint 10, the penultimate Sprint, showing 89% task completion with just 2 out of 19 tasks remaining. This Sprint has the highest scope of tasks due to the need for tweaking and finalizing the project.

This Sprint helped fully correct the time deficit and has put me back on the path of completing this Creative Project by the end of the semester, as originally planned. Thus, aside from the four weeks of inactivity near the beginning of project development, I have done well in adhering to the Scrum process from a task completion standpoint.

With respect to the interpersonal aspects of Scrum, such as Sprint Planning Meetings, I and my committee have also done very well. Throughout the entire life cycle of this project, we needed to postpone only one meeting. In fact, I met with my committee more frequently in the latter stages of development in order to better provide feedback on project progress. We ultimately decided to have weekly meetings during the last two Sprints of the project in order to ensure that the experience of the project be tweaked and optimized in line with the expectations presented in the Design Document and with the impressions of those playing the game, including the committee members. This collection

of meetings helped everyone remain on the same page about the state of project development, thereby producing the intended effect of conventional Scrum Standup Meetings.

Thus, I generally succeeded in adhering to Scrum practices during the development of *Heady Stuff*. While significant deviations from the process occurred near the beginning of development, decisive corrections brought the project trajectory back on track and in line with the expected task completion output. Most importantly, channels of communication were maintained between myself and my committee members, who have been exceedingly flexible with their own schedules to facilitate my own commitment to the Scrum process.

6b. Adherence to Design Document

Next, we will discuss the final product's adherence to the Design Document, which is accessible in this paper's Appendix. First, by reviewing sections 2.b and 2.c, we see specifications for a control system, which indicates that the player should be able to move using conventional WASD movement, pause with Escape, and Jump with Spacebar. It also specifies the need to view surroundings using the mouse and the need to use the Left and Right mouse buttons to throw punches to the left and right of the player. The document also specifies the ability to click both the Left and Right mouse buttons simultaneously to throw a middle punch. The final product deviates from these specifications only in the execution of a middle punch, which is done using the Middle mouse button (i.e. the scroll wheel button), instead of clicking both Left and Right mouse buttons. This was done to reduce the number of physical execution interactions from two to one. This decision came from my

own playtesting, in which I determined that the assignment of a middle punch action to a single mouse button would be simpler to perform and simpler to implement.

Next, section 2.d describes a health system that conveys damage taken by using visual post-processing filters. The document also describes three intensities for each damage filter and how long it takes to transition between filters. In the final product, this system was simplified by relying on a single health variable to determine the intensity of the post-processing filter. There are no other major deviations from this mechanic, and the main idea of the player's vision changing as the player takes damage in-game is still conveyed effectively. The change was made to convey the idea that the player is able to take more than three hits, which would have corresponded to the three intensity levels. Additionally, a gradual transition to a more intense post-processing effect proved simpler to implement than defining three discrete intensities.

Next, section 2.e describes the stamina system for punches thrown by the player. It specifies the need for two independent stamina bars associated with their own corresponding fist. Starting with 50 stamina points, a stamina bar will deplete by 5 points each time the fist associated with that bar is thrown. Both bars deplete when throwing a middle punch. In addition, a multiplier is added to that depletion value if the arm is used repeatedly. Lastly, the section specifies that the recovery of stamina occurs at a rate of 5 points per second. The only deviation from this specification in the final product is the recovery rate, which occurs at a rate of 1 point per second. This change was made in order to adjust the difficulty of the game. From feedback received from friends who played the game, a recovery rate of 5 points a second proved to make it too easy to avoid hazards with punches alone. Otherwise, the final product perfectly adheres to this specification.

Section 3 of the specification describes the appearance and behavior of the game's hazards. Most deviations in this section are minor. For example, the bowling ball hazard moves quickly, not slowly, in the final product since it spawns later in the sequence of attacking hazards. Additionally, because the watermelon used an asset sourced from Quixel, the final shape of the watermelon was more spherical than initially anticipated in the specification. Thus, the watermelon does not have a particularly notable rotation behavior. Another deviation has to do with the visual appearance of the black hole. Due to implementation challenges, the shader effect of the black hole causes the material to blend in with environmental features in the game rather than bending light like the refraction effect of the snow globe does (see Figure 8 for additional details on the black hole shader's implementation). Otherwise, the behavior and appearance specified in the document are reflected well in the final product.

Next are the sections on the layout of the game's arena and the art direction for the game's main HUD. The final product does not deviate from the original specifications for the design of the arena. However, the final product has many deviations from the original vision of the HUD. Minor changes include the addition of stamina bars and the repositioning of the score count on-screen. However, the largest deviation involves how alerts on incoming hazards are conveyed to the player. The original specification describes a system for visualizing the three closest incoming hazards with an arrow and cursor system. However, as described earlier, this was changed to incorporate a minimap to help simultaneously visualize incoming hazards from all directions as well as the stage on which the player is playing. This is the largest deviation from the original Design Document.

The remaining sections include state machines for transitioning between game states, player behavior, and hazard behavior as well as a list of expected assets to develop. Minor deviations exist in the 6.a state machine with some transitions not existing in the final product in order to remove redundancy. Lost transitions include a restart transition from the pause menu to the transition scene as well as a transition from the end of the tutorial to the main menu. The former situation allows the player to select ”Exit Game”, which transitions to the Game Over screen, from which they can choose to return to the main menu or restart. The latter situation allows the player to go straight to the transition scene once the tutorial ends. The 6.c state machine differs slightly as well with hazards in the final product not being destroyed on collision with the ground. This transition was rethought in consideration of how it would look strange to see an object removed completely without fragmenting when it collided with the arena. The 6.b state machine specifications, meanwhile, are largely preserved in the final product.

With respect to the list of assets to develop, there are several examples of poorly-informed specifications that do not consider how they might actually be implemented in Unreal Engine. Some of these, like the HUD Projectile Detection Scripts, have already been addressed. However, others like the GUI Interaction Scripts, fail to capture how to implement such interactivity for something like a Main Menu. As indicated previously, this is largely due to a lack of experience on my part with the workflow of Unreal Engine. Thus, because this list acts more like an attempt at anticipating lower-level requirements for the project and because we have already examined the final product’s

representation of the important features specified in the Design Document, one can conclude that the final product adheres well to the Design Document.

6c. Major Challenges

In the remaining portion of this post-mortem, I will present some of the major challenges I encountered during the project life cycle that will help clarify the failure to adhere to Agile Scrum processes during the second and third sprints. Aside from attempting to balance a newly-obtained Full-Stack Apprenticeship with a project-heavy collection of courses, I encountered serious hardware challenges with my work laptop, whose specifications are described in the Design Document. If one were to refer to the official hardware documentation for Unreal Engine 4 (Epic Games, Inc. 2022),[44] one would find that I possessed a system that met the recommended requirements, though falls considerably short of the typical system requirements for a machine used at Epic (all requirements were described for Unreal Engine 4.27. My project was running on Unreal Engine 4.26). However, knowing that the development laptop that I had used throughout my entire career at ASU met the recommended requirements, I thought that this would suffice for my development purposes. Before development properly began, I had only 8 GB RAM available, which was a large contributing factor to my initial difficulties in getting Unreal Engine and a C++ IDE like Visual Studio or Rider open at the same time. This led me to purchase an extra set of RAM for $66.64 to upgrade my laptop to 16 GB. This allowed me to run both Unreal Engine and Rider at the same time, but was not enough to address crashing issues that only worsened as the project increased in size. It was during Sprints 2 and 3 that constant crashing led me to purchase an upgraded system with an

44. Epic Games, Inc, 2022, "Hardware and Software Specifications | Unreal Engine Documentation," Unreal Engine 5 Documentation, https://docs.unrealengine.com/4.27/en-US/Basics/InstallingUnrealEngine/RecommendedSpecifications/.

AMD Ryzen 7 3.8 GHz processor, an NVIDIA RTX 3070, and 32 GB RAM with a Windows 64-bit OS for $2,497.78 on November 13, 2021. I quickly purchased such a capable system so that I could feel confident in developing with Unreal Engine without encountering any additional hardware issues. This is why one can see that, on November 27th, 2021, the beginning of Sprint 4 and around 5 days after I first received the machine, there was such a large increase in task completion. The new system eliminated the constant crashing issue I had and helped me redirect the downward trend of the project task completion rate. Such was the most critical challenge I faced during the development of *Heady Stuff*.

8. Conclusion

8a. What I Would Do Differently

In conclusion, the development of *Heady Stuff* was a challenging, yet rewarding experience. Over the course of the project, I learned much about the difficulties of scoping project requirements, about the process of independently learning how to use an unfamiliar game engine, and about the application of good development practices, like Scrum, in game development. If I had to redo this project, I would probably adjust my focus on what I would be developing. Instead of planning to produce all of my own art assets, such as 3D models, shaders, etc., I would plan to employ a kit-bashing approach. In other words, I would plan on utilizing resources like Quixel from the start for sourcing art assets so that I could focus more on the programming and game design aspects of the project. Additionally, if I were to redefine my goal as one about learning how to develop in

an unfamiliar game engine, I might have considered working with an engine other than Unreal due to the system requirements initially being beyond my means (though I do not regret the fact that I have a machine more capable of high-end development with tools like Unreal). Lastly, I would either have given myself more time to work on this project or I would have reduced the scope of this project. Ultimately, this project has shown me that, while I may have had a broad set of experiences in different aspects of game development before approaching this project, it was unrealistic of me to expect an easy time incorporating each of those experiences into a single project in an unfamiliar development environment and with such a constrained time frame. While I have largely succeeded in producing the game specified in my Design Document, there still remain things that can be done to improve the project beyond the submission of this project.

8b. Future Work

In the future, I would like to introduce more music and sound effects to bring the experience more life. I would try to make better use of the Score system by creating a simple leaderboard system that is hosted on a web server. I would be able to incorporate that aspect of network functionality into the game without having to refactor the existing player behavior, making this a viable and worthwhile addition. I would like to further refine the game's UI by modifying the font of some text blocks and by embellishing certain visual features, like the stamina bars and minimap. I would also like to refine some of the game's 3D models, such as the player mesh, along with the lighting in the game's stages. Lastly, as these changes progress, I would like to host *Heady Stuff* on itch.io (itch corp 2022)[45] so that a wider variety of players can enjoy the game for themselves.

45. itch corp., 2022, "itch.io," Download the latest indie games - itch.io, https://itch.io/.

9. References

Baker, Alessa. 2021. “Writing HLSL Code OUTSIDE The Custom Node in UE4.” YouTube. https://www.youtube.com/watch?v=V3BVsYV7ge0.

BANDAI NAMCO Europe. 2017. “Tekken 7 - PS4/XB1/PC - Unreal Engine 4 & Tekken Part 1 (Q&A Dev Interviews).” YouTube. https://www.youtube.com/watch?v=vo-FE-nlPXs.

Beck, Kent, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, et al. 2001. “Manifesto for Agile Software Development.” Manifesto for Agile Software Development. https://agilemanifesto.org/.

Corriea, Ray. 2014. “Kingdoms Hearts 3 switched to Unreal Engine 4 due to rendering difficulties.” Polygon. https://www.polygon.com/2014/10/7/6938125/kingdoms-hearts-3-unreal-engine-4.

Elalaoui, Marien. 2019. “UE4 Tutorial - Blob with Raymarching.” YouTube. https://www.youtube.com/watch?v=grmZ0I5-CgA.

Epic Games. 2022. “Custom Expressions | Unreal Engine Documentation.” Unreal Engine 5 Documentation. https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/Materials/ExpressionReference/Custom/.

Epic Games. 2022. “First Person Shooter Tutorial | Unreal Engine Documentation.” Unreal Engine 5 Documentation.

https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/CPPTutorials/FirstPersonShooter/.

Epic Games. 2022. “Slice Procedural Mesh | Unreal Engine Documentation.” Unreal Engine 5 Documentation. https://docs.unrealengine.com/4.26/en-US/BlueprintAPI/Components/ProceduralMesh/SliceProceduralMesh/.

Epic Games, Inc. 2022. “Hardware and Software Specifications | Unreal Engine Documentation.” Unreal Engine 5 Documentation. https://docs.unrealengine.com/4.27/en-US/Basics/InstallingUnrealEngine/RecommendedSpecifications/.

Farris, Jeff. 2020. “Forging new paths for filmmakers on The Mandalorian.” Unreal Engine. https://www.unrealengine.com/en-US/blog/forging-new-paths-for-filmmakers-on-the-mandalorian.

Freesound. 2022. “Freesound.” Freesound - Freesound. https://freesound.org/.

GitHub, Inc. 2022. “About · GitHub.” GitHub. https://github.com/about.

itch corp. 2022. “itch.io.” Download the latest indie games - itch.io. https://itch.io/.

JetBrains s.r.o. 2022. “Rider for Unreal Engine.” JetBrains. https://www.jetbrains.com/lp/rider-unreal/.

L'Italien, Ryan. 2021. “Unreal Engine 5 | Why 2022 Will Be Released with UE5 | Perforce.” Perforce Software. https://www.perforce.com/blog/vcs/unreal-engine-5.

Microsoft. 2022. “Azure DevOps Services.” Microsoft Azure. https://azure.microsoft.com/en-us/services/devops/#overview.

Microsoft. 2022. “Visual Studio.” Visual Studio: IDE and Code Editor for Software Developers and Teams. https://visualstudio.microsoft.com/.

Microsoft. 2022. “Visual Studio Code.” Visual Studio Code - Code Editing. Redefined. https://code.visualstudio.com/.

Minotti, Mike. 2015. “Final Fantasy VII Remake uses Unreal Engine 4 instead of an in-house tool.” VentureBeat. https://venturebeat.com/2015/12/07/final-fantasy-vii-remake-using-unreal-engine-4-instead-of-an-in-house-one/.

MonoGame Team. 2022. “MonoGame.” MonoGame.net. https://www.monogame.net/.

Muse Group. 2022. “Credits.” Audacity. https://www.audacityteam.org/about/credits/.

NVIDIA Corporation. 2007. “Chapter 18. Spatial BRDFs.” NVIDIA Developer. https://developer.nvidia.com/gpugems/gpugems/part-iii-materials/chapter-18-spatial-brdfs.

Perforce Software, Inc. 2019. “How to Use Unreal Engine 4 | Tutorial | Perforce.” Perforce Software. https://www.perforce.com/blog/vcs/how-use-unreal-engine-4-perforce.

Quixel. 2022. “We capture the world so you can create your own.” Quixel. https://quixel.com/about.

Scrum.org. 2022. “What is Scrum?” Scrum.org. https://www.scrum.org/resources/what-is-scrum.

Software Freedom Conservancy. 2022. “Git Large File Storage.” Git Large File Storage | Git Large File Storage (LFS) replaces large files such as audio samples, videos, datasets, and graphics with text pointers inside Git, while storing the file contents on a remote server like GitHub.com or GitHub Enterprise. https://git-lfs.github.com/.

Stafford, Matt. 2014. “Transitioning from UE3 to UE4.” Unreal Engine. https://www.unrealengine.com/en-US/blog/transitioning-from-ue3-to-ue4?sessionInvalidated=true.

Taiga Agile, LLC. 2022. “Taiga.” Taiga: Your opensource agile project management software. https://www.taiga.io/.

Thomsen, Mike. 2010. “History of the Unreal Engine.” IGN. https://www.ign.com/articles/2010/02/23/history-of-the-unreal-engine.

Unity Technologies. 2022. “Unity.” Unity Real-Time Development Platform | 3D, 2D VR & AR Engine. https://unity.com/.

Unity Technologies. 2022. “Unity - Scripting API: ForceMode.” Unity - Manual. https://docs.unity3d.com/ScriptReference/ForceMode.html.

UnrealCG. 2019. “Mesh Slicing In Unreal Engine.” YouTube. https://www.youtube.com/watch?v=oIdKxYYQBdw.

Unreal Engine. 2016. “Custom Material Node: How to use and create Metaballs | Live Training | Unreal Engine.” YouTube. https://www.youtube.com/watch?v=HaUAfgrZjlU&t=560s.

Unreal Engine. 2021. “Digital Twins: Building Cities of the Future | The Pulse | Unreal Engine.” YouTube. https://www.youtube.com/watch?v=i12kObBpz-E.

Unreal Engine. 2022. “Virtual Production Sizzle Reel 2022 | Unreal Engine.” YouTube. https://www.youtube.com/watch?v=_oMH_gy7r60&list=PLZlv_N0_O1gYBVBadlg6vUKXAazMSY2r8.

Wrike. 2022. “What Is Agile Methodology in Project Management?” Wrike. https://www.wrike.com/project-management-guide/faq/what-is-agile-methodology-in-project-management/.

YoYo Games Ltd. 2022. “Easily Make Video Games with GameMaker.” YoYo Games. https://gamemaker.io/en/gamemaker.

10. Appendix

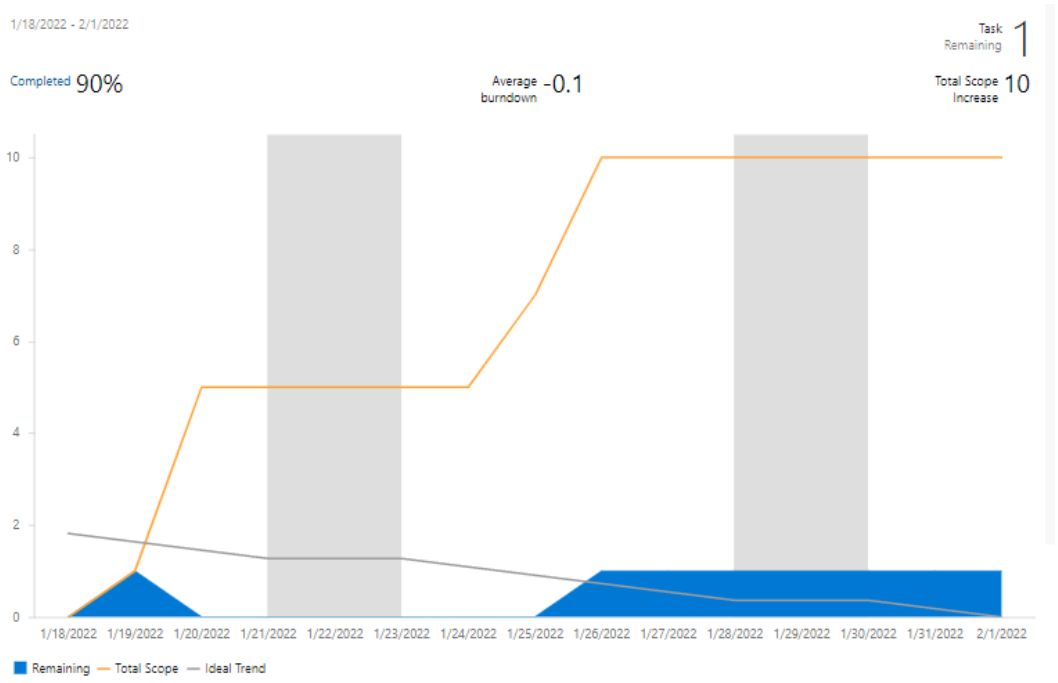


Figure 26. Task completion burndown of Sprint 5 showing 96% task completion with just 1 out of 10 tasks remaining. Consistent effort is now being made.

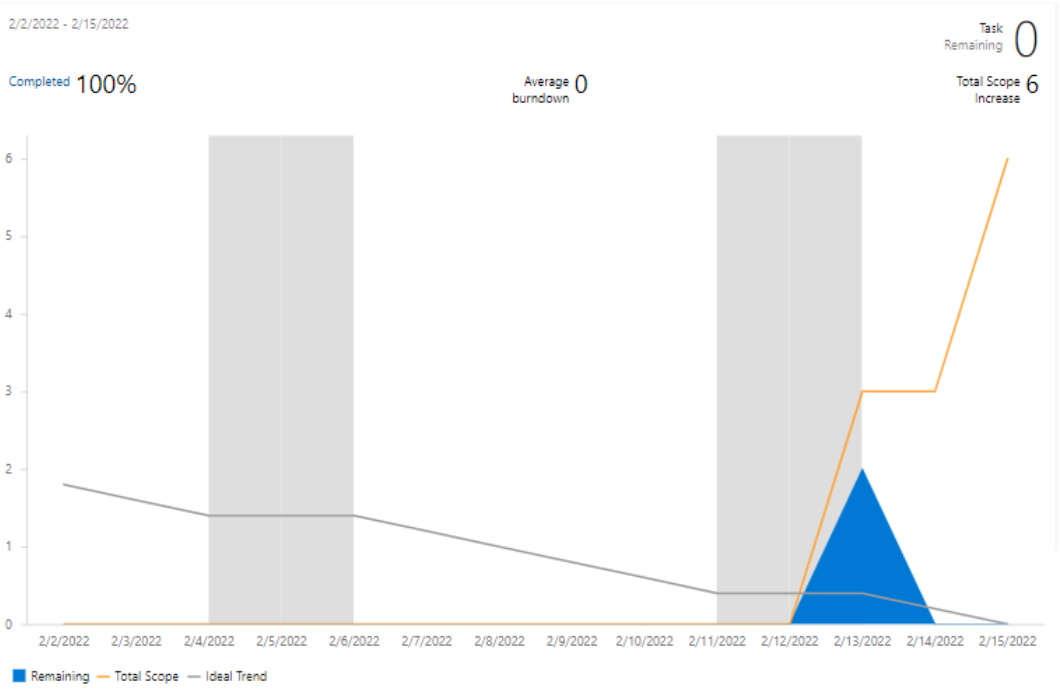


Figure 27. Task completion burndown of Sprint 6 showing 100% task completion with all 6 tasks completed.

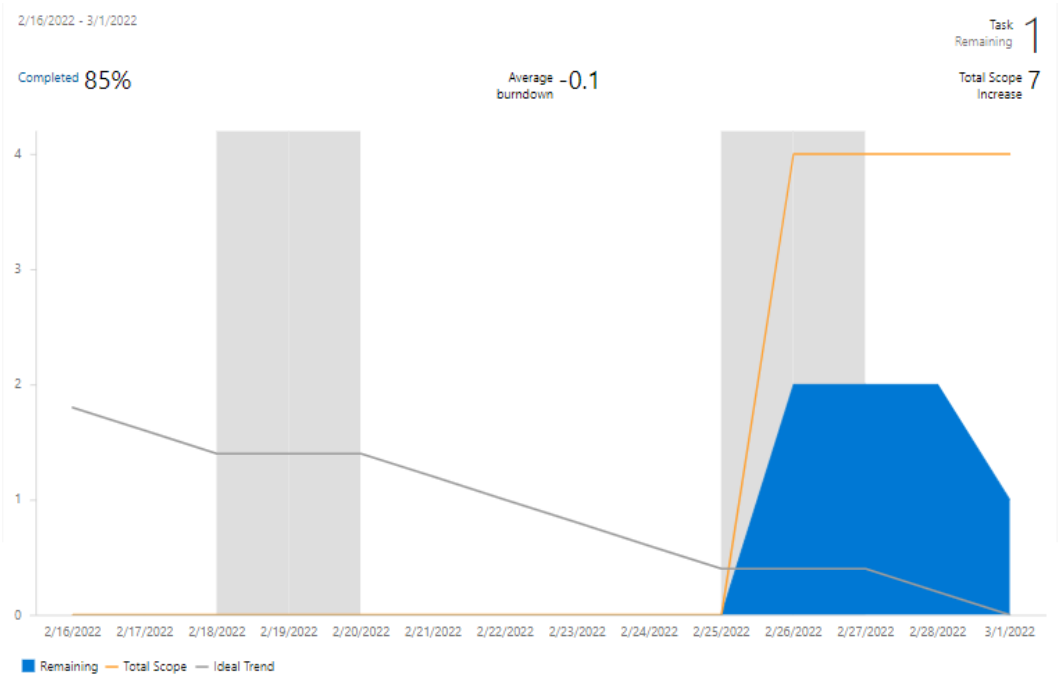


Figure 28. Task completion burndown of Sprint 7 showing 85% task completion with just 1 out of 7 tasks remaining.

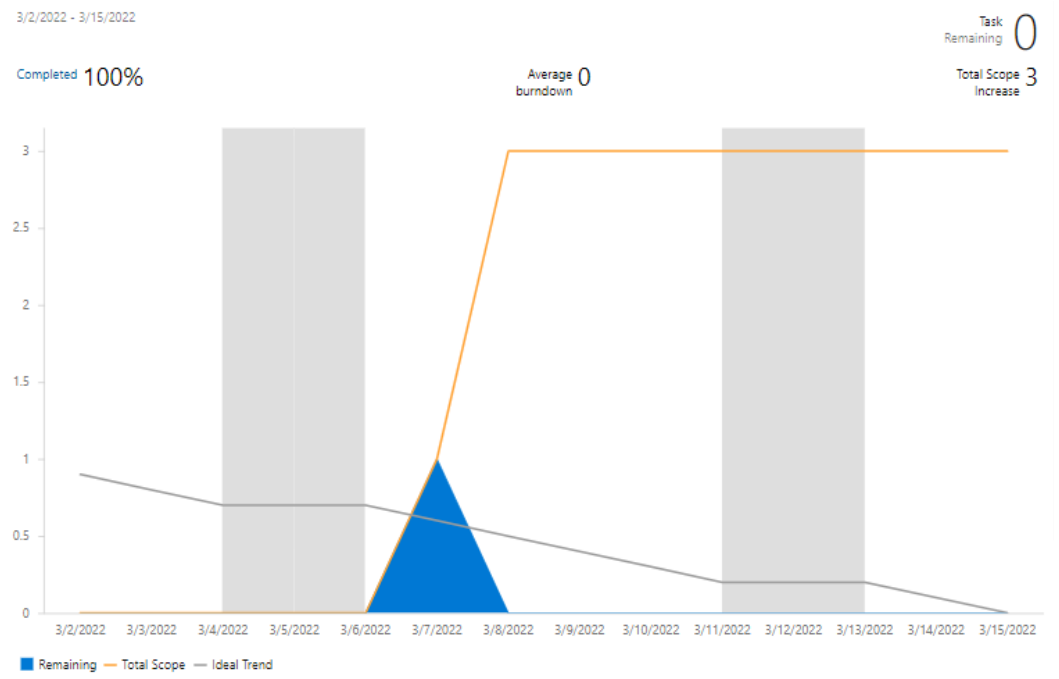


Figure 29. Task completion burndown of Sprint 8 showing 100% task completion with all 3 tasks completed. Task output is rather low. However, consistent effort is maintained.

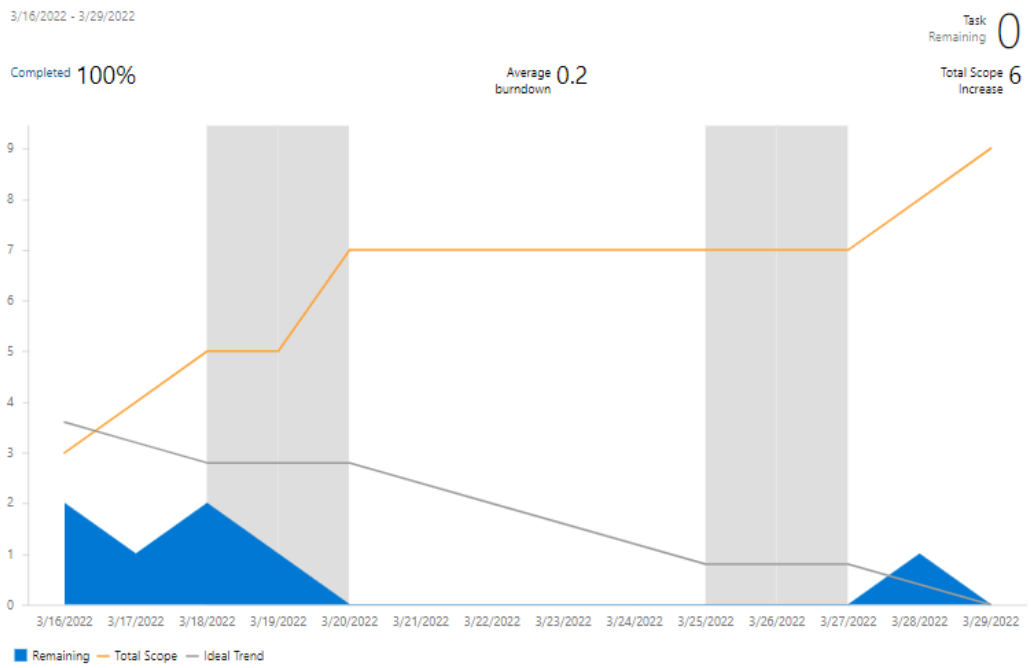


Figure 30. Task completion burndown of Sprint 9 showing 100% task completion with all 6 tasks completed. Task output is on par with the expected output from a Computing Capstone sprint.

	Logic Blueprints	C++	Shader Blueprints	HLSL
Use of Control Structures (i.e. loops)	Yes	Yes	No	Yes
Constant Folding Optimization	N/A	N/A	Yes	No
Risk of Visual Spaghetti Code	Yes	No	Yes	No

Figure 31. Table for easy comparison between Logic Blueprints, C++, Shader Blueprints, and HLSL.

Heady Stuff

1. GAME OVERVIEW

1.a Two-Sentence Pitch

A first-person arcade game featuring simple melee combat that challenges the player to fragment a variety of incoming flying hazards while standing on a head-shaped arena called the Prime Mover. The fast-paced gameplay has the player punch hazards in order to accumulate points.

1.b Two-Minute Pitch

Heady Stuff is a first-person arcade game to be built in Unreal Engine. The game will take place in a space-themed environment on top of a head-shaped arena called the Prime Mover. The first-person controller will stand on top of the Prime Mover and attempt to survive a barrage of randomly shaped hazards that are hurtling towards the player from all directions.

Being an arcade game, the game will have no obvious victory state and will end when the player has been struck by too many hazards in quick succession, similar to how deaths are handled in games like Uncharted. Thus, the main object of the game is to obtain a high score, which can be done through the slow accumulation of points as the player avoids obstacles and through the faster, though riskier, tactic of punching hazards as they approach the player. One of the main features of this game is that hazards will fragment using Unreal's physics simulation around the player as they are punched, meaning that they still pose a threat to the player after they are broken up.

The game will be built in Unreal Engine 5 in order to gain exposure to some of the best technologies used in the game development industry while also increasing the odds of producing a game that has good visual polish. This project will also serve as a testing environment for the Scrum agile methodology in game development.

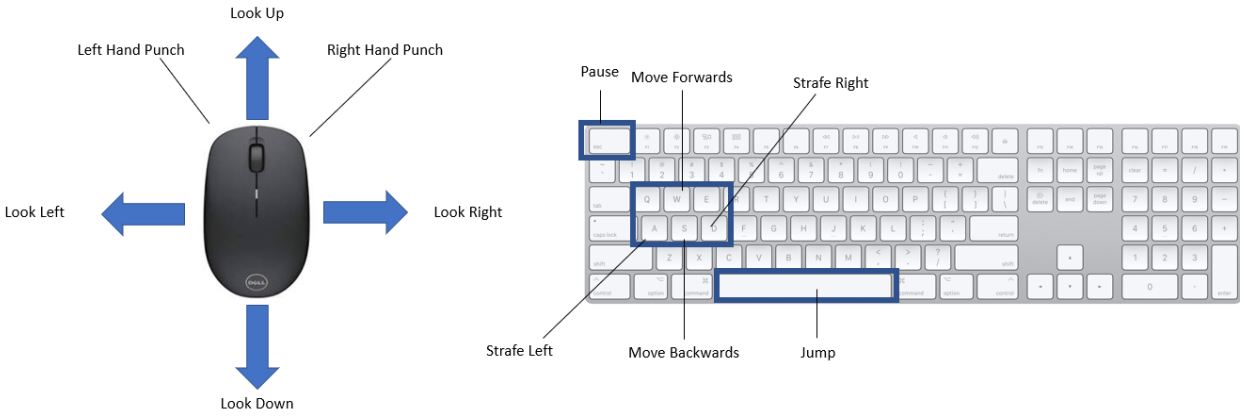
2. GAMEPLAY OVERVIEW

2.a Level Structure Overview

When the game begins, the player will be greeted by a title screen containing options to start the game, play a tutorial to learn the controls, and quit the game. On starting the game, the camera will start by showing the entire head arena and then zoom in towards the player model standing on the head, thereby transitioning to a first-person view where player inputs are accepted.

The arena will contain a simple circular layout that is open to a barrage of incoming hazards that spawn from beyond the arena, giving the player a good view of incoming objects

2.b Basic Input Layout



2.c Player Character Controls

As described in the Basic Input Layout subsection, the player controls will reflect a standard FPS control scheme. WASD controls will handle moving forwards, backwards, left, and right while mouse movement will control the FPS camera view.

The main control aspect that distinguishes this project from other FPS games is that the click buttons, namely, Left Click and Right Click, are responsible for how the player punches, which is the main defense against incoming hazards. These punches, when executed, will land in differing positions, one landing left from center screen and the other landing right from center screen. This means that the player will need to throw punches deliberately in order to effectively avoid incoming hazards. Worth noting is that clicking both Left and Right mouse buttons at the same time will allow the player to throw punches with both hands simultaneously, allowing for good coverage against projectiles. However, all punches come at the cost of stamina, which will be further described in the Stamina System subsection.

Aside from the main player controls, the Escape key will be used to bring up a menu during gameplay, allowing the player to quit easily.

2.d Health System

The health system will resemble one seen in the Uncharted games. When the player gets hit by an incoming hazard, filters will be applied to the camera view to communicate to the player that they've taken damage. As the player is continuously damaged, the filters become more intense until the player dies, causing a game over after a single try.

There will be three levels of filter intensity, each corresponding to how much damage the player took. In order to fully recover from the first level of intensity, the player must not get hit by hazards for 1 second. In order to fully recover from the second level of intensity, the player must not get hit by hazards for 2 seconds. Finally, in order to fully recover from the second level of intensity, the player must not get hit by hazards for 4 seconds (these values are not final). Once the player is fully recovered, the filter intensities are reset and the player has another chance at progressing farther in the game without dying so easily.

However, as indicated earlier, when the player takes too much damage and passes beyond the third level of intensity, the player dies, receives a game over screen, and is presented with options to quit the game or retry.

2.e Stamina System

The Stamina System will be primarily associated with the punching mechanics of the player character. In order to discourage the player from spamming the punching mouse buttons in order to easily avoid taking damage from hazards, each arm will have its own stamina bar, each of which contains 50 stamina points. After a punch from one of the arms is thrown, 5 stamina points will be deducted from that arm's stamina bar. Punches thrown in quick succession from a single arm will begin with a 5 point stamina deduction followed by a deduction of that base value plus an increasing multiplier value (e.g. first punch = 5 point deduction, second punch = 5 + 1 * 0.2 point deduction, third punch = 5 + 2 * 0.2 point deduction, etc.)

To recover stamina, punches cannot be thrown, thereby allowing stamina to return at a rate of around 5 points per second.

3. HAZARDS OVERVIEW

3.a Snow Globe

The Snow Globe hazard will be one of the more basic projectiles to attack the player from beyond the arena. It will be decorated with the mesh of a snow globe made in Blender and will include material effects to allow light to refract when travelling through the glass part of the globe.

It will spawn from outside the arena and move towards the player in a vertical sinusoidal motion. The player can either avoid the hazard or punch the hazard to avoid taking damage.

3.b Bowling Ball

The Bowling Ball hazard will be a large, but slow-moving hazard. It will contain the mesh of a bowling ball and will have some standard lighting effects that include specular highlights to emphasize the smoothness of the object.

It will spawn from outside the arena and move slowly towards the player in a straight line. The player can either avoid the hazard or punch the hazard to avoid taking damage.

3.c Watermelon

The Watermelon hazard will be a large hazard that rotates about the thick section of the watermelon, thereby giving substantial breadth to the damaging portions of the object. Ideally, the object will initially appear with an ordinary-looking Watermelon mesh that reveals its red pulp when it fragments under the player's punch.

It will spawn from outside the arena and move quickly towards the player in a straight line while rotating about its thick section (that is, not about an axis through the melon's minor radius). The player can either avoid the hazard or punch the hazard to avoid taking damage.

3.d Horseshoe

The Horseshoe hazard will be a mid-sized hazard. It will contain the mesh of a Horseshoe while including materials that make it look rough and metallic.

It will spawn from outside the arena and move towards the center of the arena in an orbiting spiral motion. The player can either avoid the hazard or punch the hazard to avoid taking damage.

3.e Black Hole

The Black Hole hazard will be a large objects that moves towards the center of the arena in a straight line. Materials will likely be the primary tool in simulating the light-bending effects of a black hole.

While its movement is ordinary, the Black Hole will also have a gravitational effect that pulls the player towards it. If the player gets too close to the hazard, the player dies. This means that, when a Black Hole emerges, the only option available to the player is to avoid it. The player will not be able to fragment a Black Hole.

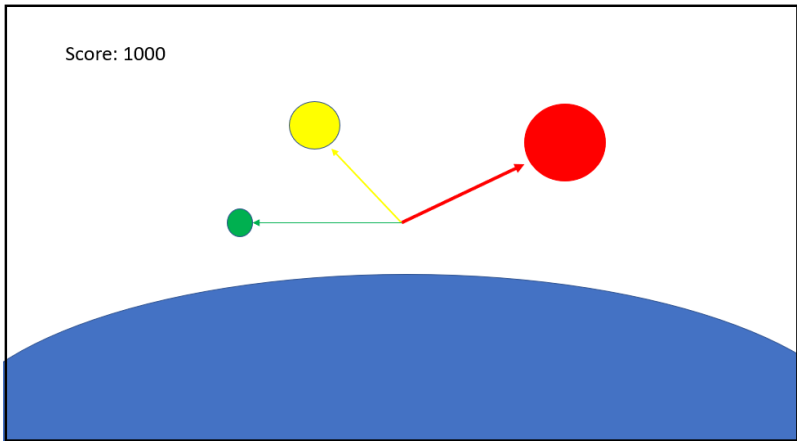
4. ENVIRONMENT AND LEVEL DESIGN OVERVIEW

4.a Basic Layout of Prime Mover Arena

As indicated earlier, the layout of the arena will be in the shape of the top of a human skull. This means that there might be an elliptical boundary around the flatter portions of the skull that still leaves room for slight slopes at the edges of the boundary while leaving good walking space for the player to move around and visualize incoming hazards.

5. ART DIRECTION DOCUMENTATION

5.a GUI/HUD

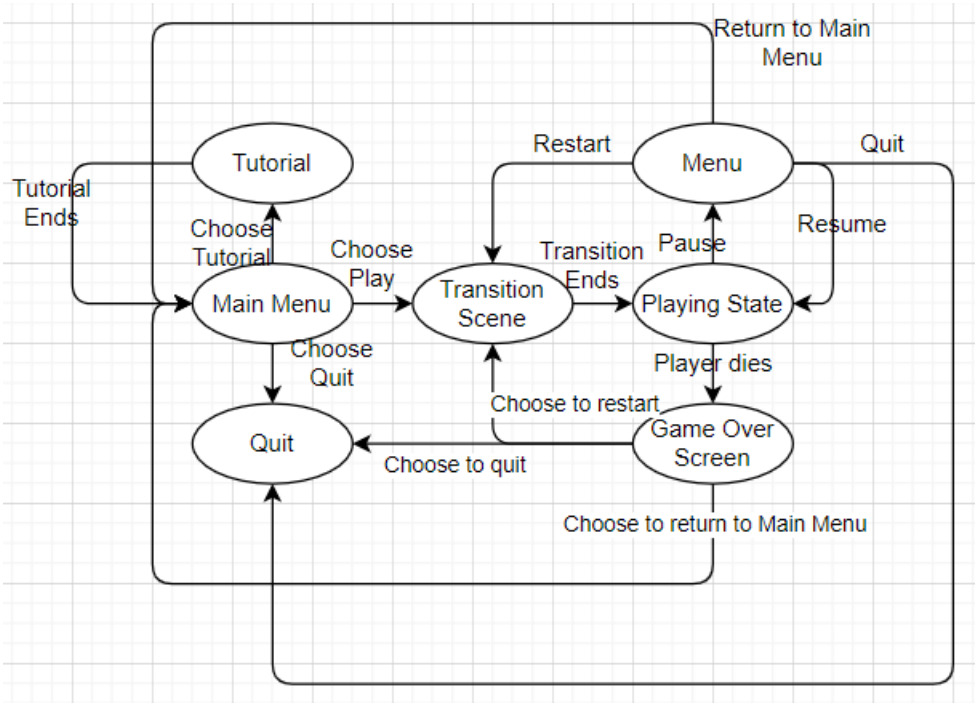


The above image demonstrates the general concept for the GUI and HUD elements during gameplay. The blue space is not part of the GUI or HUD and instead helps visualize the arena space that the player will be playing on.

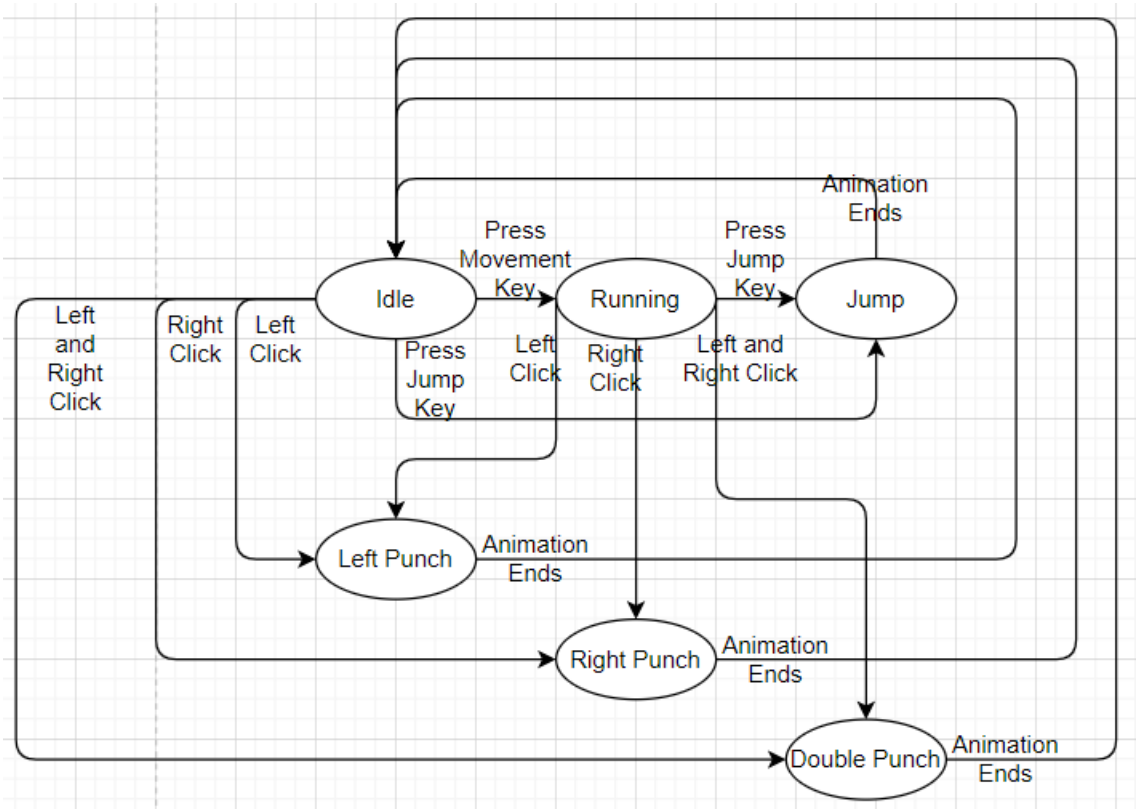
First, there will be a section that displays the score, which rises as the player successfully avoids and destroys hazards. There will also be some sort of GUI that alerts the players to hazards approaching the player. To prevent the player from becoming disoriented, instead of informing the player about all hazards that are approaching, there will be at most three arrow markers stemming from the center of the screen leading towards the three closest projectiles. The closest projectile will be marked with a red arrow while the second and third closest projectiles will be marked with a yellow and green arrow, respectively. When projectiles are behind the player, the arrow markers will point to either the left or right side of the screen to indicate to the player the fastest way to turn to see the incoming projectile.

6. GAME STATE MACHINES

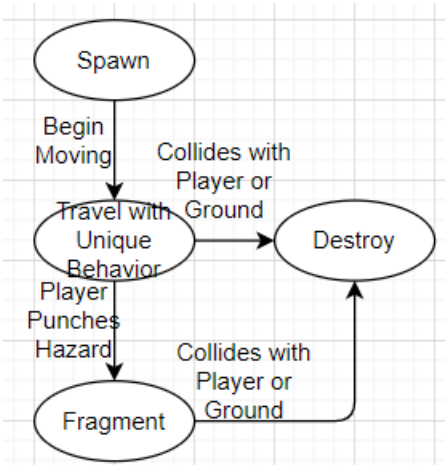
6.a Overall State Machine



6.b Player State Machine



6.c Hazard State Machine



7. LIST OF EXPECTED GAME OBJECTS, SCRIPTS, AND MATERIALS

7.a Objects

- 3rd Person Player Mesh
- 1st Person Arms Mesh
- Head Arena Mesh

- Snow Globe Mesh
- Bowling Ball Mesh
- Watermelon Mesh
- Horseshoe Mesh
- Main Menu GUI
- Pause Menu GUI
- Projectile HUD

7.b Scripts

- Player Movement Scripts based on Actor
- Hazard Movement Scripts based on Actor
- Hazard Spawning Scripts
- Scene Navigation Scripts
- GUI Interaction Scripts
- HUD Projectile Detection Scripts
- Shader Scripts contained in Materials

7.c Materials

- 3rd Person Player Material
- 1st Person Arms Material
- Head Arena Material
- Snow Globe Materials
- Bowling Ball Material
- Watermelon Materials
- Horseshoe Material
- Black Hole Material

8. TIMELINE/MILESTONES

Being an agile project, there will be no strict phases for when assets are made and game logic is implemented. This means that milestones will be determined in meetings with Professors Acuña and Hentges, then completed during a Sprint per Scrum procedures. However, since the Prospectus required a general project schedule, one can refer to this project's Prospectus to help frame the general goals of the project.

9. BIOGRAPHIES

9.a Professor Ruben Acuña

Professor Acuña will act as the director for this creative project. Being a Software Engineering Lecturer, he will help infuse a Software Engineering flavor to the project by helping ensure that good agile practices are used in the development of the project in addition to the standard expectations of good coding practices.

9.b Professor John Hentges

Professor Hentges will act as the Second Committee Member for this creative project. His expertise lies in Digital and Media Art as well as Game Studies. Additionally, his first-hand experience in the games industry will be instrumental in guiding both the game design and graphical aspects of the project.

9.c Jacob Hreshchyshyn

Jacob is the developer who will be working on the agile creative project. He will be responsible for developing the assets, logic, and game design of the project as defined in this document. He will also be responsible for adhering to the Scrum process in developing the game and will work in close collaboration with Professors Acuña and Hentges to ensure good progress on the project.

10. APPENDIX

10.a Supported Operating Systems

As of the time of this writing, the only supported operating system will be Windows 10. There is a possibility that cross-compilation to Linux systems may be possible in the future.

Currently, I am unable to build for MacOS since I lack the hardware to build for those systems. However, if I can find someone to assist in building a project for MacOS, I will build for MacOS. For now, however, the only expected operating system to deliver for will be Windows 10.

10.b Hardware Resource Envelope

Since I am building on my own laptop, I will expect the game to perform well on a system with specifications similar to those of my laptop.

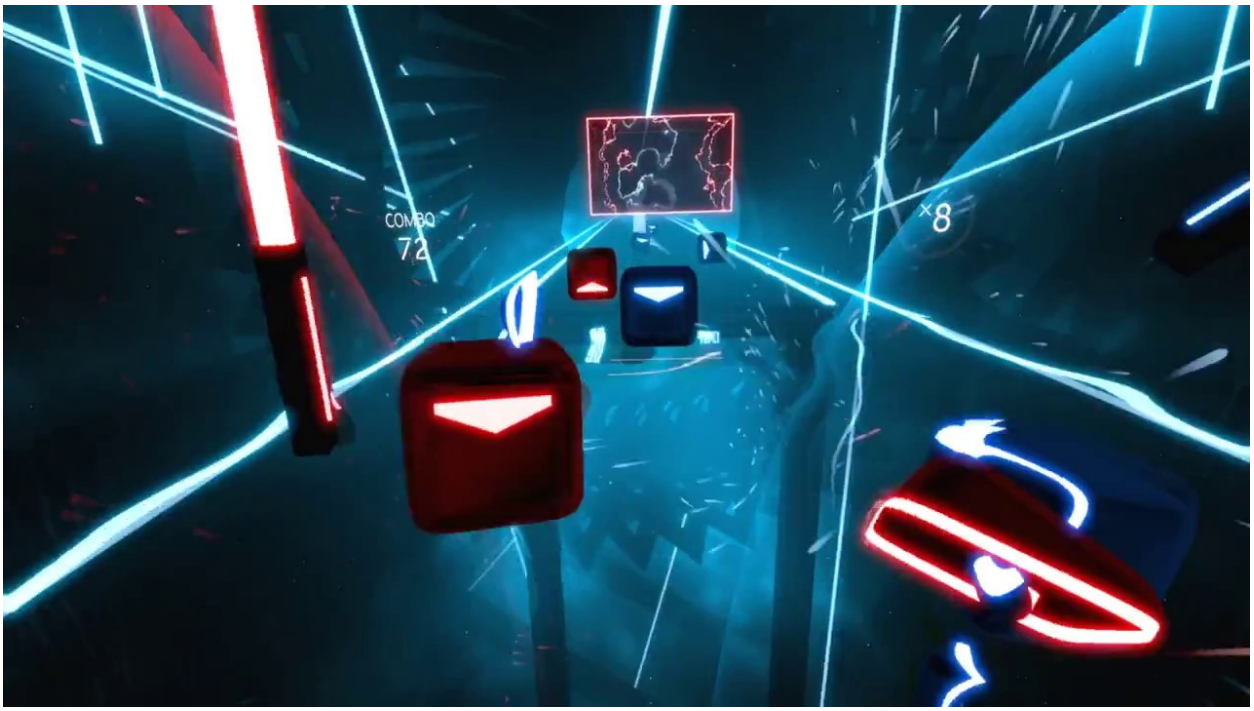
My specifications are the following:

4-core CPU at 2.60 GHz.

16 GB of system RAM.

NVIDIA GeForce GTX 960M.

10.c Reference Games



Beat Saber – While this creative project will not be a VR game, the challenge of destroying hazards flying towards the player resembles Beat Saber's gameplay of swinging a sword at red and blue blocks in time to music.

Teardown – A sandbox heist game featuring a fully destructible environment. While the scale of this creative project will not match the scale of Teardown, the idea of the destructibility of incoming hazards will be borrowed from this game.

Super Mario Galaxy – A 3D platformer with unique gravity mechanics, a cool space theme, and a fantastic orchestral score. The space theme of the game will be referenced for this creative project.

Uncharted 2: Among Thieves – This reference game could have been any of the Uncharted games. The main referenced aspect will be the health system where the more damage the player takes, the more altered the view becomes until the character dies.