Computer Science Research Notebook

Game Design With Unreal Engine

Dang Le

2022 - 2023

Table of Contents

Preliminary Project Plan

September 13, 2022

Problem statement: Define the problem you are trying to solve in 1-2 paragraphs.

Before a game is released to the market in a playable stage, many milestones are set and reached to organize a large system of collaborations from many different departments to achieve an end product. Of the many aspects in a game, my main focus will be within the code, processes it undergoes, and its applications. As an idea proposal doesn't complete a game, someone has to prototype and someone has to program in a language. This 2022 - 2023 school year will be dedicated to researching and exploring the complex process of game development, going in depth on the programming aspects, and the use of a game engine.

Interest: Why are you personally interested in solving this problem?

Over the course of the past few decades, video games have evolved at an exponential rate. With my childhood favorite, Minecraft, being one of the most famous games to have been made during this time along with me gaining regular access to the internet, my childhood and middle school years revolved around my desire to play video games. It was bound that I would eventually get curious about the process. In 8th grade, I started my personal exploration with Lua and OOP (Object Oriented Programming). Since then, I have been more and more interested in professional game development.

Relevance: Who else (what communities of people) would benefit from a solution to this problem?

Although the game development industry would benefit greatly from the use of game engines and programming, many other industries are starting to get involved. One such community would be the movie production industry. As animated films are becoming more popular, engines such as Unreal Engine have adapted to be able to render films rather than just cutscenes for a game.

Research Aspect: What topics/skills will you be researching during this project?

My primary focus will be in the use of C++ and Visual Scripting in game design. More specifically, I plan to go in depth in the applications of visual scripting, such as the prototyping process, as well as the logical process of using Unreal Engine's nodes available for game designers. Following this, research in the use of C++ will be done to an extent in which game design is possible.

Existing Research: What is already known about this problem? What solutions already exist?

As game designers and large companies propose their ideas and move onto the development stages of a game, the issue of funding and time come in. Should time and resources be invested into custom foundations of a game, built from scratch? Is it worth it? Noticing these issues, companies such as Epic Games and Unity Technologies invested their resources into creating Unreal Engine and Unity for companies and developers to have a foundation to work with.

Computational Artifact: What will you be creating? What will it be able to do?

Over the course of 9 months, projects and reports will be submitted following substantial progress in a multitude of formats. Many include the creation of short play-test demonstrations, essays/write ups, and long term projects. With some milestones being a combination of more than one work, these are intended to display my research and findings, as well as demonstrate my ability to apply knowledge summarized from daily entries in this digital journal.

Risks: What are the aspects of your project that you are least confident about? What resources or support do you need to be successful?

In long term research, getting stuck on large concepts or losing track of the main goal can be a concern. This holds especially true for independent research. As such, guidance in any form will be appreciated. I believe the best form of guidance for me will be a forum or course I can look to as a resource for help when stuck.

Confidence Level: Currently, how sure are you that this is the project you want to work on for the next 9 months? Do you have other ideas that you are considering?

I can say with confidence and certainty that this project is what I want to work on for the next 9 months. Prior to this school year starting, I had already planned to learn how to utilize Unreal on my own throughout the school year. As this opportunity came, I decided to use a more professional excuse of “School” to work towards my desire to learn Unreal. As of now, no ideas of topics intrigue me as much as Unreal, not even close.

Unreal Engine & Alternatives

September 15, 2022 - September 21, 2022

As game engines are the foundation and ground base for many professional projects, the engine should be able to provide a wide range of tools for developers to utilize. With the purpose of efficiency for developers, the engine should be performant and simple to an extent. Assessing Unreal Engine (UE5) & Unity (2022), my main focus will be within four aspects specifically for the practicality of 3D game design targeted for PC.

- Physics
    - Simulations
- Rendering
    - Lighting
    - Materials & Textures
- Scripting
    - Language(s)
- Developer Tools
    - World Design
    - Visual Effects (VFX)
    - Assets Provider

Notes

	Unreal Engine (UE5)	**Unity (2022)**
Physics	Physics Solvers - PhysX (Integration) - Chaos (Built-in) Simulations - Object Destruction - Cloth - Strand-Based Hair & Fur - Fluids	Physics Solvers - PhysX (Integration) - Havok (3rd Party Plugin) Simulations - Cloth - Strand-Based Hair & Fur
Rendering	Nanite: Allows for importing of meshes composed of trillions of polygons while maintaining real-time frame rates - Virtual Shadow Maps (VSM) Lumen - Dynamic global illumination - Photoreal ray tracing (Shadows, reflections, ect.) - No baking required	Lighting - Global illumination - Shadow Maps Level of Detail (LOD) Render Pipelines - Built-in - URP (Low end devices) - HDRP (High end devices) - Custom

	Level of Detail (LOD)	
Scripting	Visual Scripting - Blueprints Language(s) - C++ (Objectively complex; Rewarding)	Visual Scripting - Bolt Language(s) - C# (Simplistic)
Developer Tools	Material Editor World & Terrain - Realistic or Stylized clouds & atmosphere - Procedural Foliage Tool (Environmental details) - World Partition (Large scale worlds) - Terrain editor - Customizable brushes via Blueprints Modeling - Built-in sculpting, UV creation, and baking tools Niagara - VFX - Fluid Simulations Marketplace	Shader Graph Terrain (Unity Package) - More variability with asset store Particle Editor Asset Store DOTS (Data Oriented Tech Stack) - Data Oriented

Other Notes	- Powerful graphics reduces practicality on mobile devices
	- Overall better for VR
	- 2D game capabilities
	- Pricings
	  - Free
	  - 5% royalty after $1 million in revenue mark
		- Use intended for mobile devices
		- Has VR support
		- 2D game focused capabilities
		- Pricings
		  - Personal Plan: Free
		  - Plus Plan: $399 /yr per seat (Must switch to Pro plan after $200k in revenue)
		  - Pro Plan: $1,800 /yr per seat

Settling on a Game Engine; Unreal Engine or Unity?

An essay comparing Unreal Engine 5 and Unity 2022 and for which purposes you would use each. Compared aspects are physics, rendering, scripting, and developer tools for mainly 3D game design and use as a learning tool.

Entry Summary

When it comes to deciding between Unreal Engine 5 and Unity release 2022, one of the most important points to take into consideration would be your supported platform(s) and game type. For the sake of simplicity and consistency, I compared UE5 and Unity 2022 on practicality of 3D game design targeted for PC using the criteria of physics, rendering, scripting, and development tools.

In short, when it comes to a learning tool, Unity has a lead over Unreal Engine. With more simplistic functions and a large community, Unity acts as a perfect starting point to build skills. Scripting in either C# or Bolt (Visual Scripting), the basics to understanding OOP can be nurtured. Unity isn't limited to the basics however. With an arsenal of tools for developers to use such as cloth simulation and the built in shader graph, game design can easily reach professional levels. Can't find or understand something? The asset store has thousands of assets free and paid for learning, reference, and use. The large community also has built a forum of answers to questions you'll likely ask.

On the other side of the ring, we have Unreal Engine 5. To say the least, UE5 is a beast that can't be compared in graphics and built-in tools. With built-in Nanite and Lumen, global illumination can light up scenes of millions of skyscrapers, composed of millions of polygons, running real time effortlessly. Developers are provided incredible tools without the need of hitting the marketplace. However, Unreal is more fit for the advanced developers. As games are written in C++, memory management can get complex and the number of powerful tools can appear intimidating. But in the right hands of the right people, Unreal stands at the top of the podium.

Starting Up Workflow

September 23, 2022 - September 29, 2022

Install Unreal Engine 5 through either Epic Games launcher or by GitHub source. To begin working in C++, an IDE is required, preferably Visual Studio 2022.

Visual Studio 2022 Installation/Setup

1. After running the installer, under Workloads, select .NET desktop development, Desktop development with C++, and Game development with C++
    a. Additional Individual Component for Unreal Engine 4 from source: .NET Framework 4.6.2 targeting pack
    b. Additional Individual Component for Unreal Engine 5 from source: .NET Core 3.1 Runtime (LTS)
2. Install Visual Studio 2022 & Uninstall older versions of Visual Studio
3. The following are for **source downloads** only
    a. Open UE5 file directory in command prompt
    b. Run *GenerateProjectfiles.bat -2022* (The year flag is for VS-Code versions)
    c. Open the file *UE5.sln* in Microsoft Visual Studio Version Selector
        i. If prompted with *Target framework not supported*, select *Update target to .NET Framework 4.8* or the recommended option and check the *Remember my choice* box
    d. On the top bar, set the following:
        i. Solution Configuration - Development Editor
        ii. Solution Platform - Win64
    e. On the right explorer, right click UE5 and select *Clean*
    f. Again, right click UE5, this time selecting *Debug → Start New Instance*
    g. Create/Open a project
    h. In VS-Code, select *Build* by right clicking the project
4. The following are suggested Visual Studio 2022 settings:
    a. In Visual Studio Code, select *Tools → Options → Projects and Solutions → General* and **Disable** *Always show Error List if build finishes with errors*
    b. In options, select Text Editor → All Languages → Scroll Bars and **Enable** *Use map mode for vertical scroll bar*
    c. In *Text Editor*, navigate to *C/C++ → Advanced*
        i. Under Browsing/Navigation, **Enable** *Disable External Dependencies Folders*
        ii. Under IntelliSense, **Disable** *Disable IntelliSense*
    d. In *Text Editor*, navigate to *C/C++ → View*
        i. Under Inactive Code, **Disable** *Show Inactive Blocks*
    e. Navigate to *Debugging → General*
        i. **Disable** *Enable Edit and Continue and Hot Related*
    f. Right click the toolbar and navigate to *Customize → Commands → Toolbar* and change *Build* to *Standard*; Select Solution Configuration in the Preview and set width to 200 in Modify Selection

Github Setup/Syncing

1. Install a Git Client (Desktop, Kraken, ect.)
2. Install Git if not already installed
3. Install Git LfS
4. Repository Creation Approach One:
    a. On the main page of github, create a new Repository
        i. **Enable** *Add .gitIgnore* with the Unreal Engine template
    b. Click *Code* and either select *Open with (Git Application)* or copy and paste the HTTPs
    c. In your Git application, clone the repository
    d. Cut and paste the project files into the same root directory as the *.git* folder
    e. Assure everything works, then commit and push the repository
5. Repository Creation Approach Two:
    a. Create a local repository in your Git Application
        i. Create with the existing project folder
        ii. **Enable** *Add .gitIgnore* with the Unreal Engine template
    b. Assure everything works, then commit and publish the repository
6. Install Git LfS on repository
    a. Open Terminal
    b. Run *git lfs install*

Project Setup

1. In Unreal Engine select *Edit* → *Editor Preferences* → *Source Code* → *Source Code Editor* and set it to Visual Studio 2022
2. You can now uninstall Visual Studio 2019 that was required for project creation

Moving Project Directories

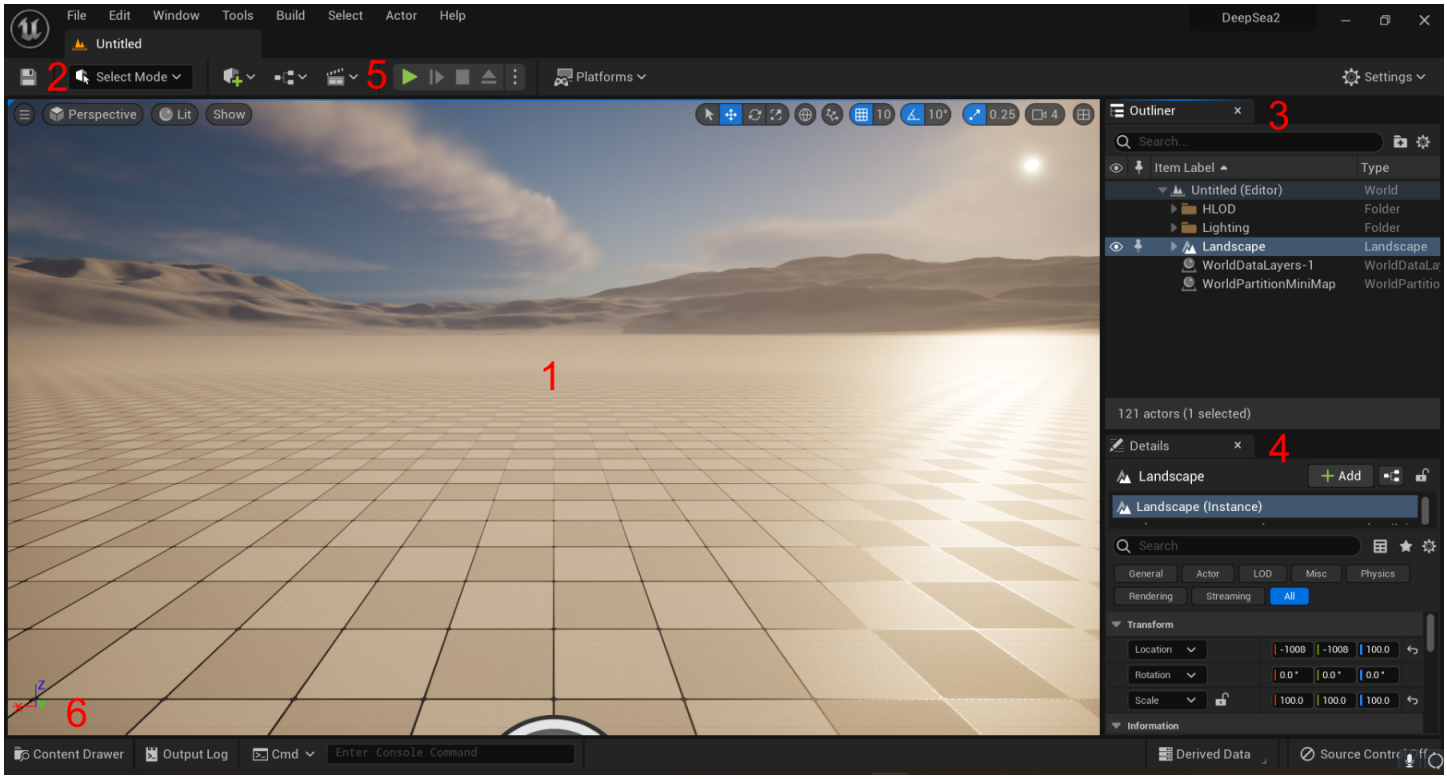
1. Locate project in directory and move the folder
2. Restart Unreal Engine
3. Browse for Project Files and select the new location

Unreal Engine Familiarization

October 4, 2022

Interface Navigation

1. Viewport
    a. Perspectives
    b. View Modes
    c. Viewport Options
2. Editor Modes
3. World Outliner
4. Details
5. Toolbar
    a. Play Testing
    b. Blueprint Shortcut
6. Content Drawer - Majority of game development occurs in here
    a. Assets Importer
    b. Asset Collection
    c. C++ Classes



Blueprints

The Blueprint Window can be accessed through the toolbar or navigating to the Content Drawer and opening a blueprint.

Navigation

- Viewport
    - Used to view and edit component details
- Event Graph
    - Start events of blueprints
    - Bulk of logic
- Functions, Variables, and Events
    - Found in the left windows

The Suspicious Statue

October 5, 2022 - October 8, 2022

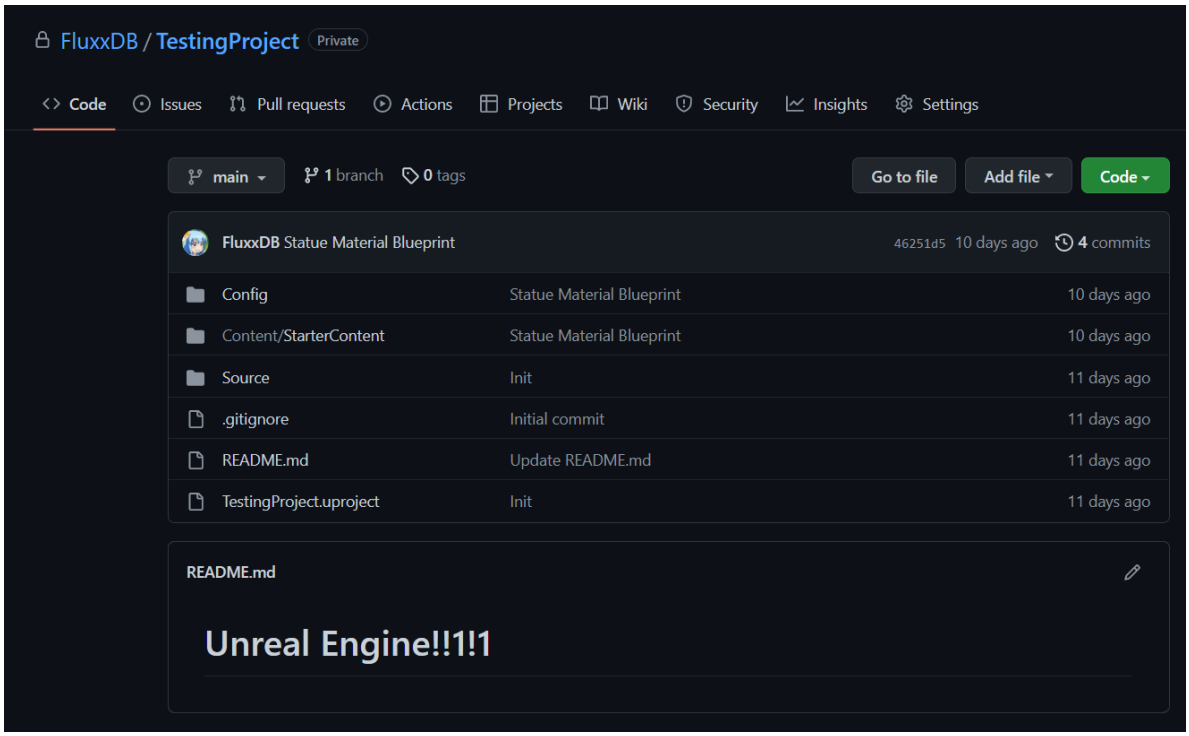
To start, experimentation can be said to be the best way to learn and get a deeper understanding. Over the span of three days, I was able to make a statue spin at a constant rate while being positionally locked to the table. After five seconds, the statue blew up and flew away.

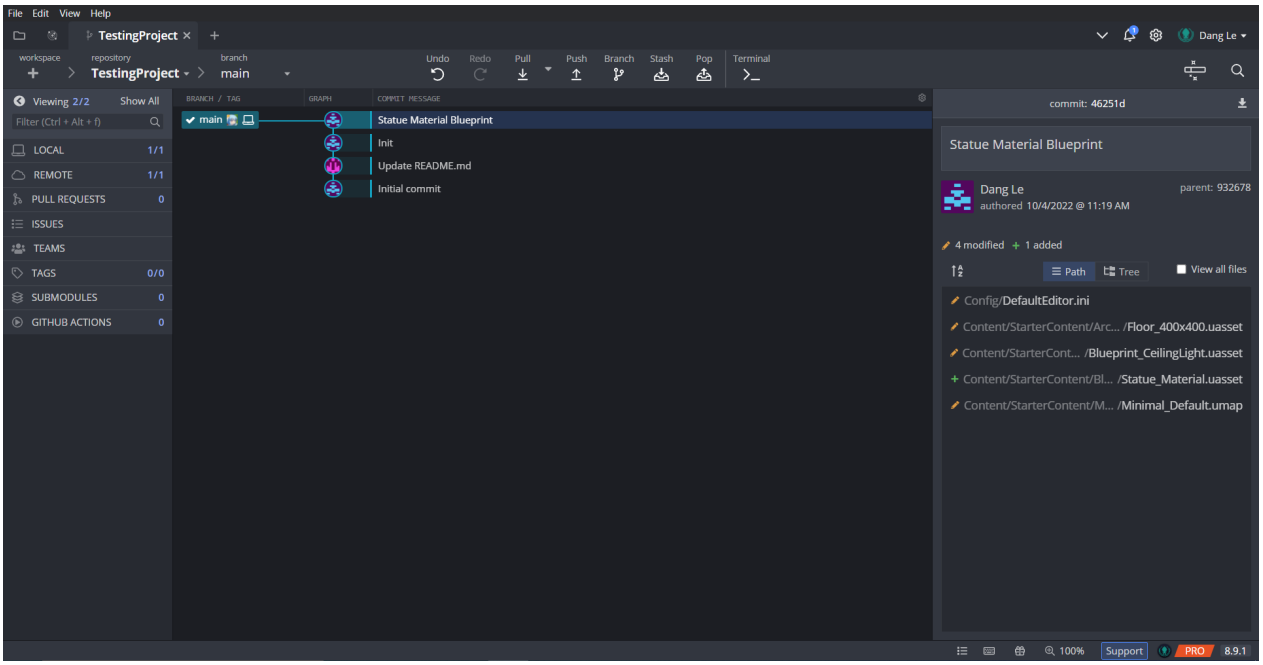
Technical Goals

1. Familiarization with blueprint graphs
2. Utilize blueprint components
3. Modify basic properties (position and rotation) with blueprints
4. Utilize blueprint functions and events

Project Startup

To synchronize work between my Laptop and Desktop, I used Github to host the repository and GitKraken to pull/push. This way, work can be saved and accessed from nearly anywhere.



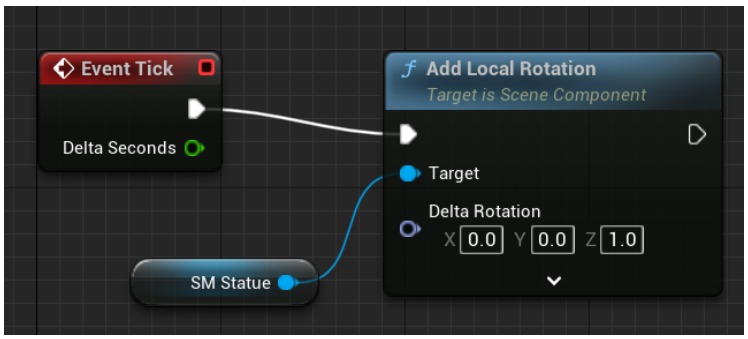


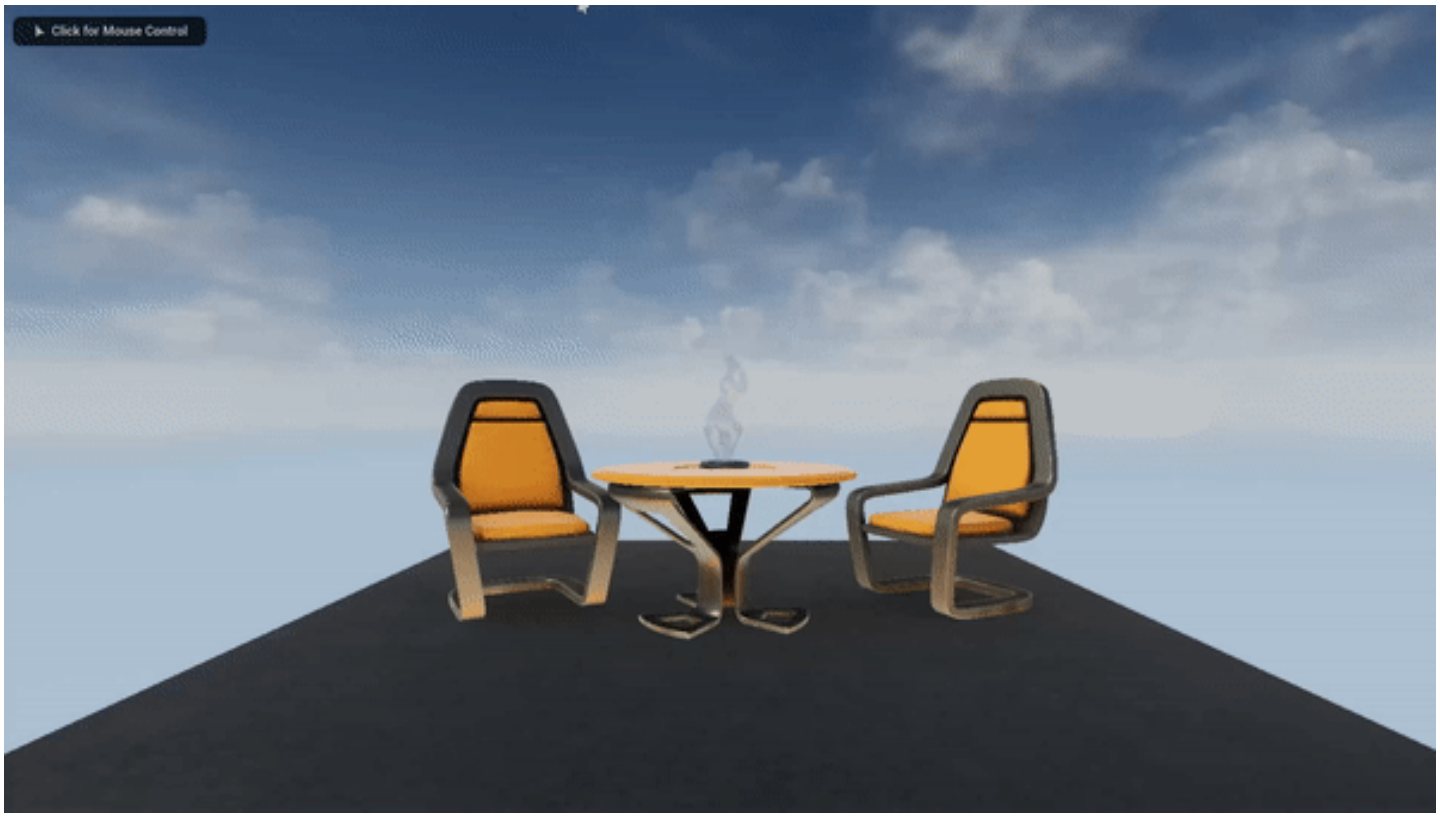
Blueprinting

Taking on the challenge to complete the statue without any specific tutorials resulted in quite a bit of frustration at the start. Navigating the content drawer, I wasn't quite sure how to get an Actor into a blueprint or even access it. However, after some playing around, I had discovered a few things:

1. The components inside of a blueprint are like a pool of assets, actors included, to be accessed and modified with the blueprint
2. Variables can be made beforehand, storing default values, but should not be used to reference actors in the components tab
3. Multiple graphs reference different sections of the logic
    a. Used to separate from Functions, Events, ect.

With these tips in mind, I headed to Unreal Engine's documentation page to look for a method to make the statue spin. I settled with using a loop and constantly applying Angular Force. To test this, I used the *Event Tick* event to call the *Add Local Rotation* node:



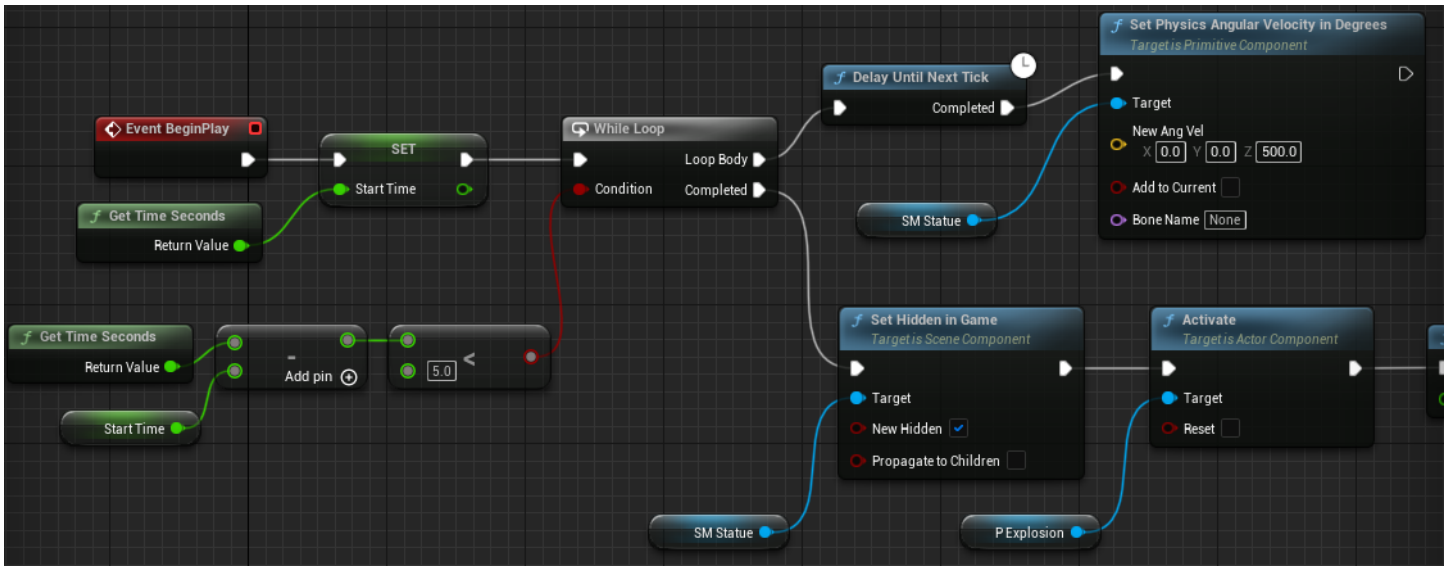


Although the statue now spins, it wasn't quite what I needed. I needed the statue to stop spinning at some point so linear AND angular force can be added (explosion blast). Now to the drawing board:

Logic

1. Loop to keep statue spinning and pinned to a position
    a. While Loop
    b. (CurrentTime - StartTime > 5) Condition
    c. Apply constant amount of force
2. Break away from loop after five seconds
    a. Connection to *Completed* pin of while loop node
3. Apply linear and angular force to statue
    a. Function to apply force a single time
4. Emit the explosion particle
5. Clean up
    a. Hide the statue (No particle interruption)
    b. Wait *x* seconds
    c. Destroy the statue
    d. Destroy the particle

Although I wasn't very familiar with all of the node names or categorizing, it was easy to grasp with the aid of Unreal Engine's documentation. Thus, the first prototype blueprint was completed (I omitted the *Fling* function to test the logic and confirm there were no errors):



(Logic following While Loop hidden due to length; It's just a *Delay* node into a *Destroy* node for cleanup)

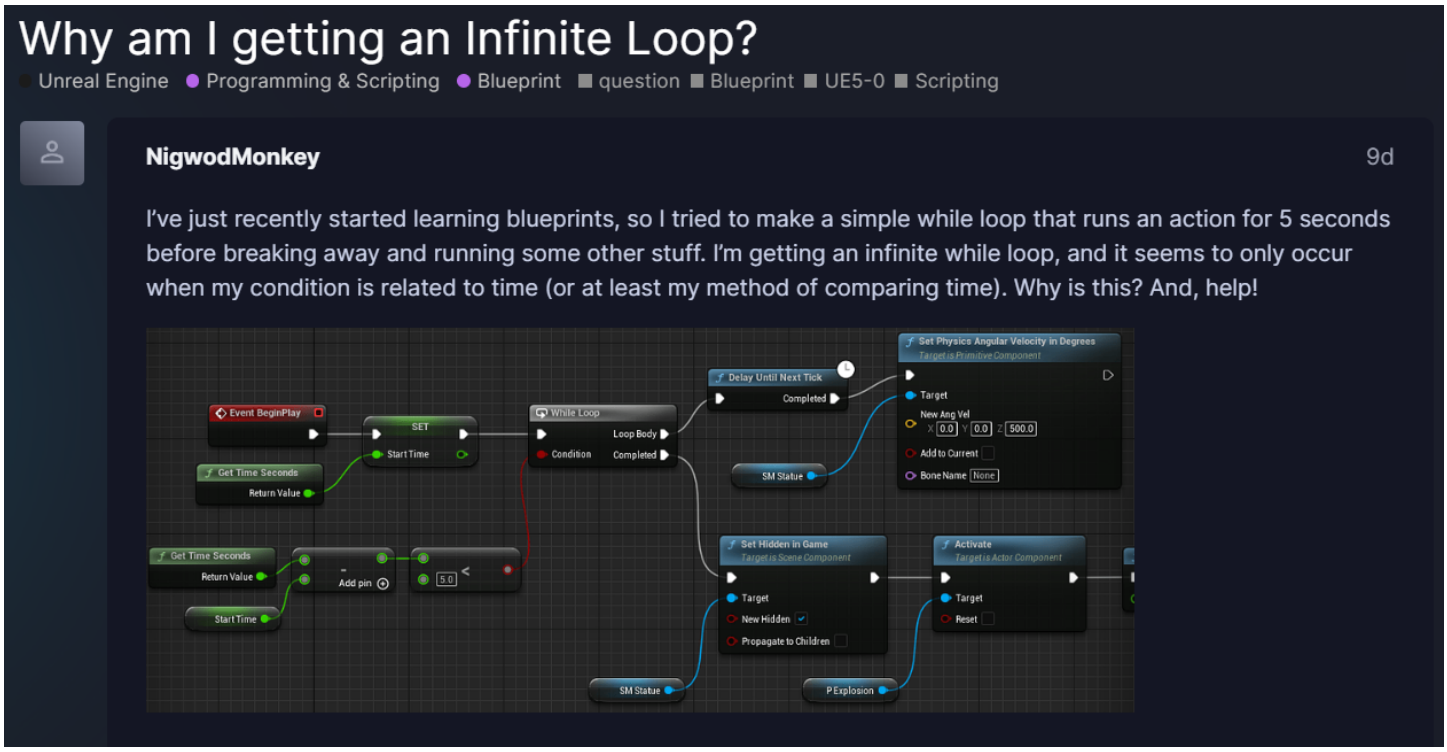
Good News: I now have a piece of logic that was constructed from 0 knowledge and makes sense
Bad News: It didn't work

In fact, the first run never works in programming. I'm all too familiar with this feeling. However, this time was a little more frustrating than usual. With traditional programming, this would have normally been something I could do in 10 minutes TOPS. It had already taken me over an hour to figure this stuff out, and I'm now about to spend the next 3 hours trying different variations of the while loop condition.

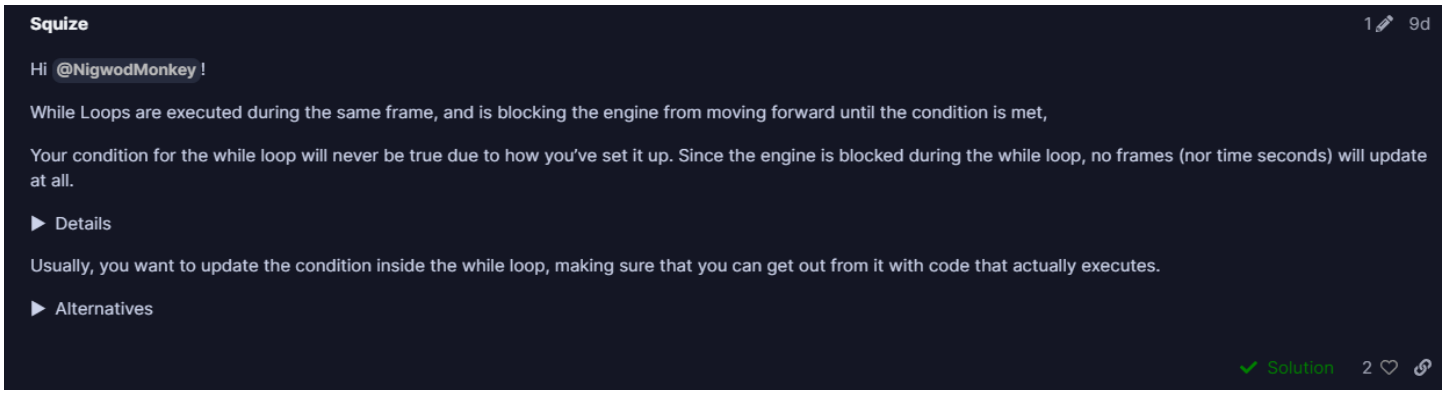
Despite going through hours of iterations of the while loop condition, nothing worked, except for 1 condition. Now, I had wanted the condition to be based off of time, which is why I was frustrated that of the 20 or so conditions I had tested, the only ONE condition that WASN'T based off of the game time worked. Why?

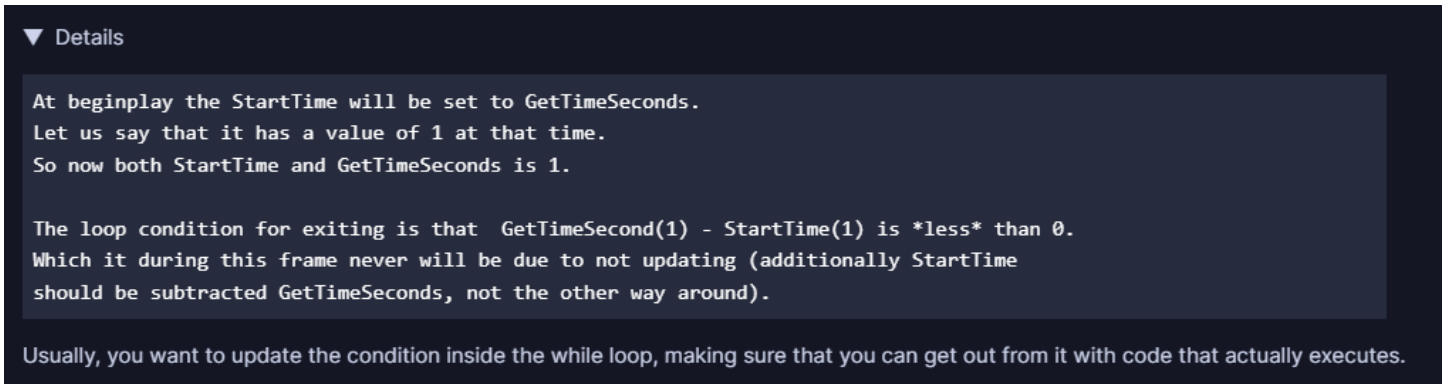
Developer Forum Post

Although I had wanted to finish this statue without the use of external resources, it doesn't seem like I'll be able to solve this problem any time soon. It was 1am, and it was about time I asked for some help.



Within an hour, I had an answer. With the response from Squize, I was able to get back to work after a good night's rest. Thank you Squize the Norwegian Automation Engineer (Profile Link).

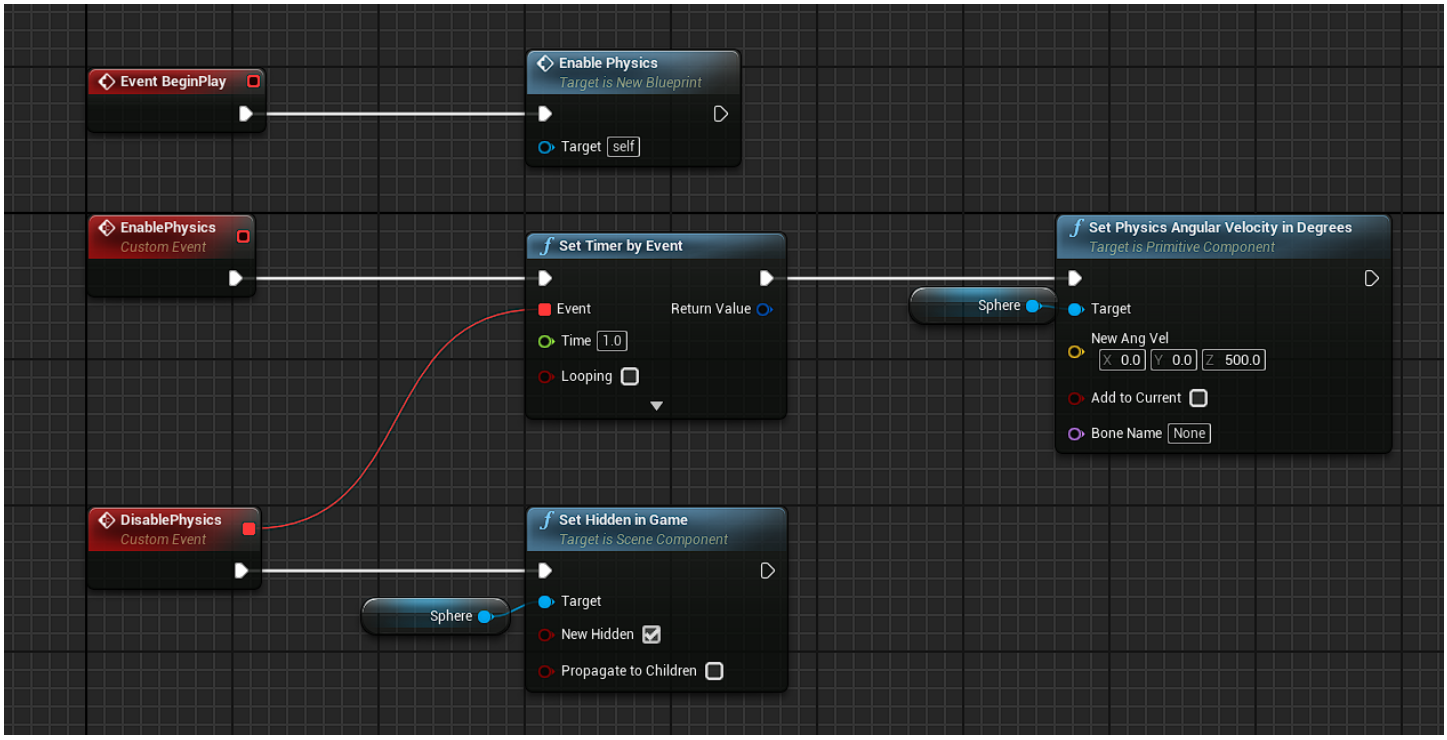




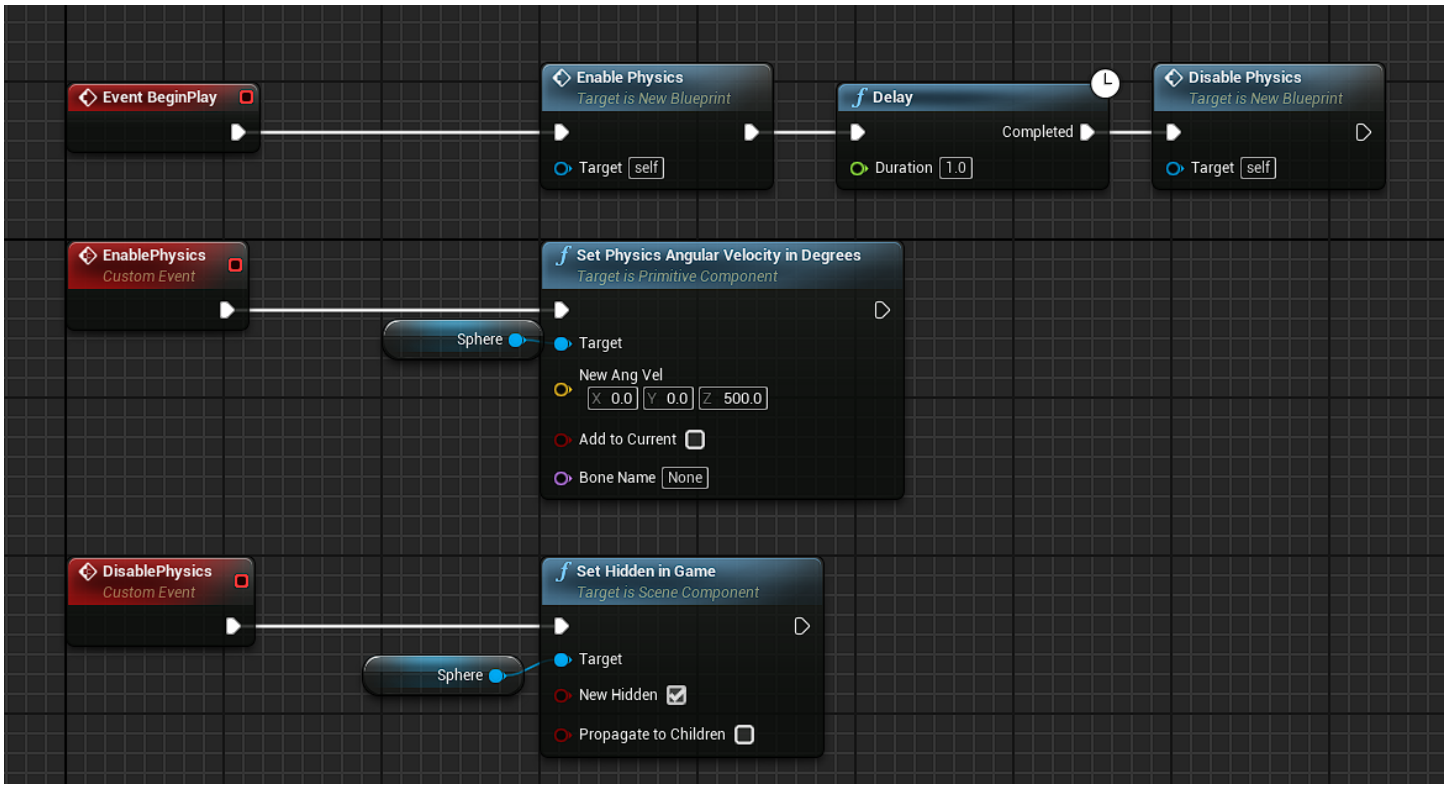
(Squize even gave suggestions after answering my question)

Alternatives Provided by Squize:

By Timer



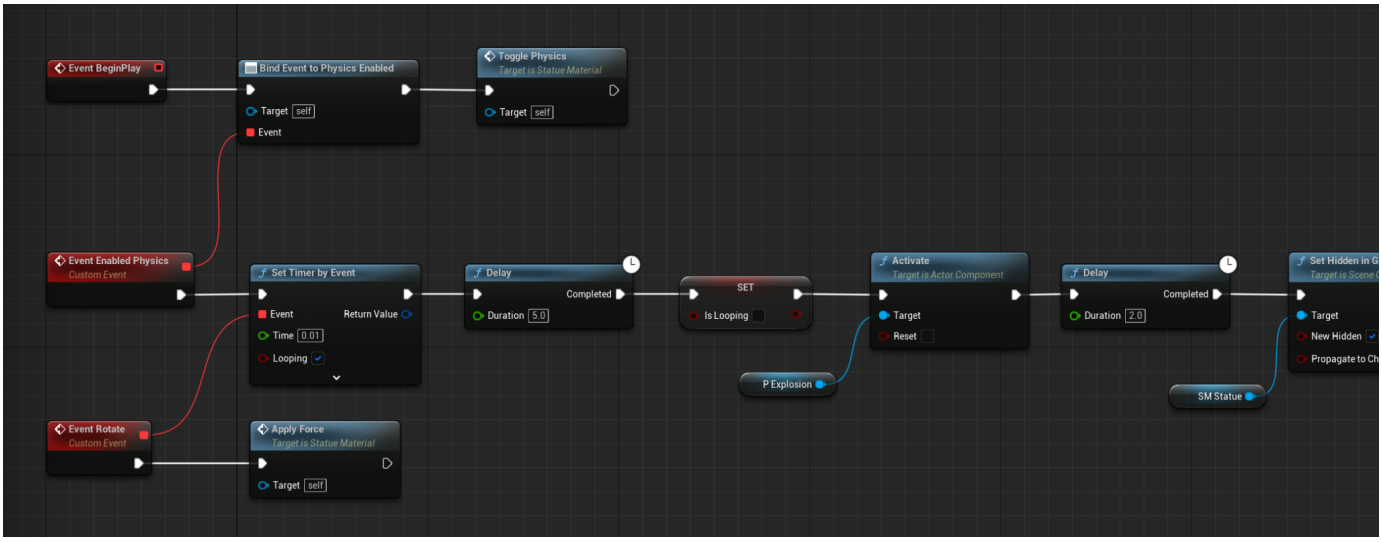
By Delay



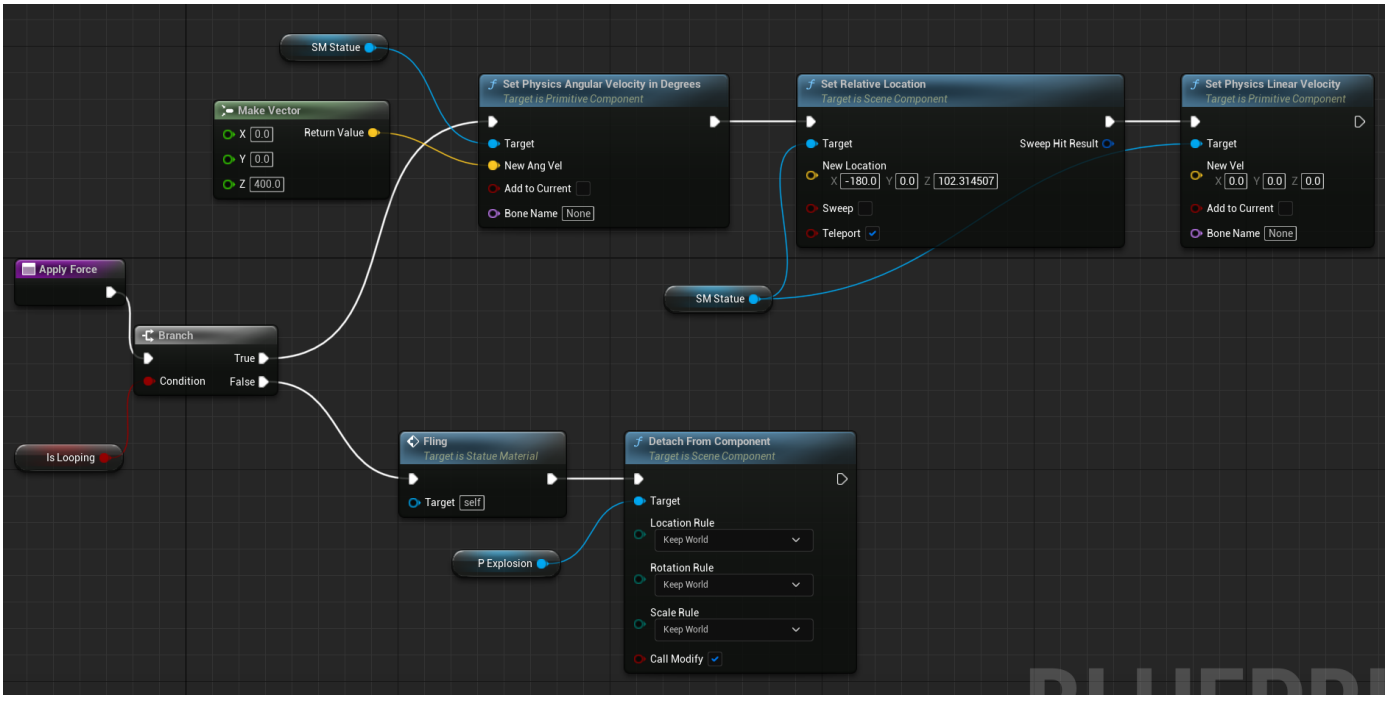
Taking into account the Timer Alternative, I started working on the new piece of logic, scrapping everything I had. This time, I had to be a little more clever, turning the timer into a loop that constantly updates the statue's rotation and position.

Logic Revision

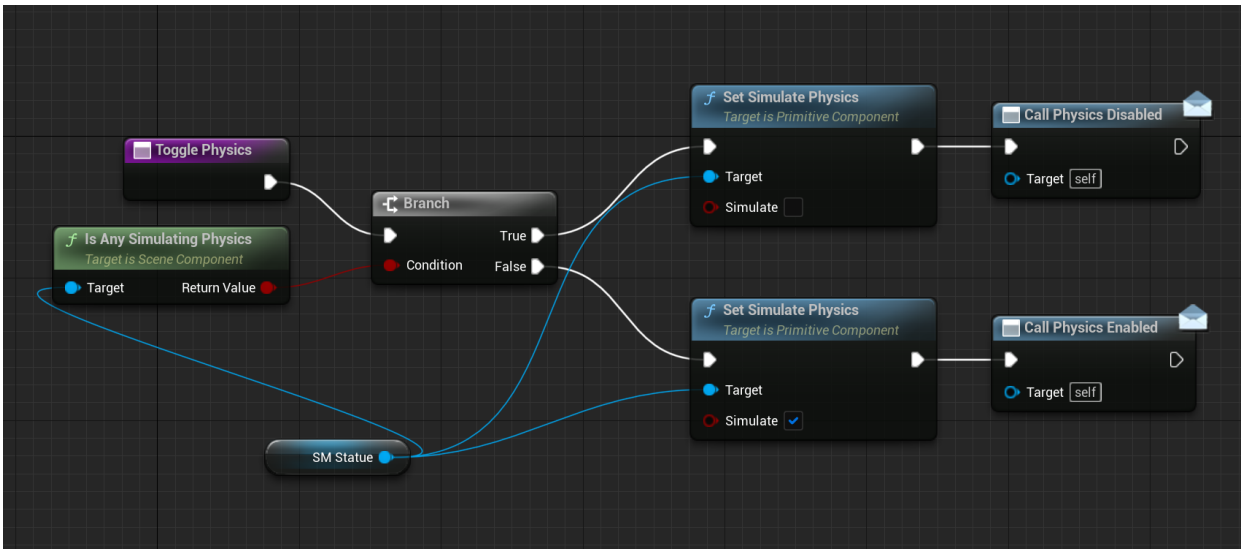
1. Toggle physics on game run
2. Physics Enabled Event Fires
    a. Timer repeatedly calls Rotate Event
        i. Rotate Event calls Apply Force
    b. 5 second delay start
    c. Disable timer loop
3. Play Explosion Particle
4. Call fling function
5. Clean up



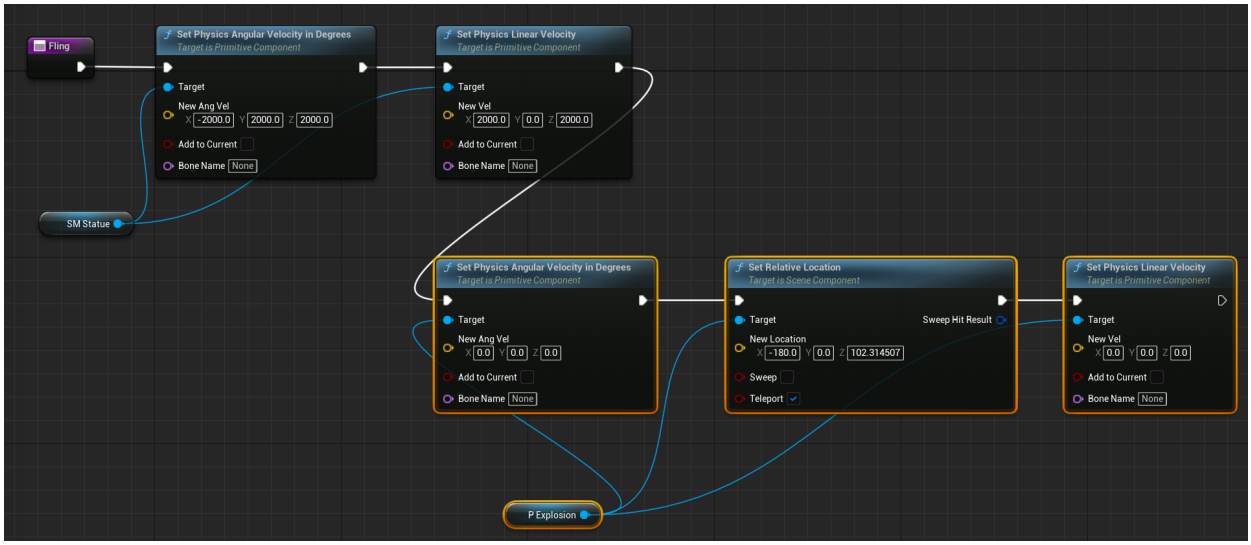
(Event based logic; Cleanup was omitted)



(Apply Force Function)



(Toggle Physics Function)



(Fling Function)

Entry Summary

With this tiny assignment/task, the goal was to learn a bit about how nodes flow and connect, how blueprints interact with components, and how to modify basic properties like position and rotation. Along the way, I not only reached my goal by completing the statue, but also learned two important things. First, the Unreal Engine Developer Community is an EXCELLENT resource for questions, help, and posts relating to Unreal Engine in general, with reliable responses coming in quick and concise. Second, when a while loop is executed, Unreal Engine stops at the current frame to continuously execute the code until the condition is met. For example, in my instance, game uptime was used as the condition to break out of the while loop when it exceeds five seconds. However, time will never increase to be greater than five if the frame is frozen (thus time being stuck at zero indefinitely).

In addition to reaching my goal, I was able to learn more about niches specific to Unreal Engine as well as find a reliable source for help. Thus, I would call this entry a success.

Rocket Science 101

October 12, 2022 - June 2, 2023

With the completion of The Suspicious Statue, I'm starting to grasp the basics to Blueprints. To further develop a good foundation and move onto more challenging material, I've decided now is a good time to start a longer term project to explore blueprints. The gameplay goal of this project is to develop a first person lab game, where cartoon-styled rocket components come together to the desire of the player.

Technical Goals

1. Blender Implementation (Meshes)
2. Blueprint Systems
    a. Carry & Interact
    b. Movement
    c. Rocket Components
    d. Rocket Launch
3. SFX Implementation
4. VFX Development
    a. Particles
    b. Mesh Outlining
5. World Creation
    a. Atmosphere Layers
6. Basic UI
    a. Guide Text
    b. Interaction Text

Gameplay Goals

1. First Person
2. Physics Based Carry System
    a. Interact to activate item in hand
3. Stylized World
4. Rocket Launch Milestones
    a. Lift-Off
    b. Clouds Atmosphere
    c. High Atmosphere
    d. Exit Atmosphere
    e. Moon Crash
5. Rocket Components
    a. Engine
    b. Thrusters
    c. Fins
    d. Cosmetic Paint (Spray Can)

Project Startup

Following typical procedure, Github will be hosting the repository while GitKraken will be used to push/pull. This time, Blender will also be needed, so I've installed it on my Laptop for remote working.

Grab/Carry System

With the carry system, I had intended it to be a simple click to grab and E to interact. However, it quickly spun out of hand and got much more complex than I had imagined. Regardless, I managed to figure it out with the help of developer forums and youtube videos.

Logic

1. Listen to Left Click Pressed Input
    a. Raycast to mouse world position
    b. Grab object based on obstruction point
    c. While Loop grabbed object's position to mouse world position

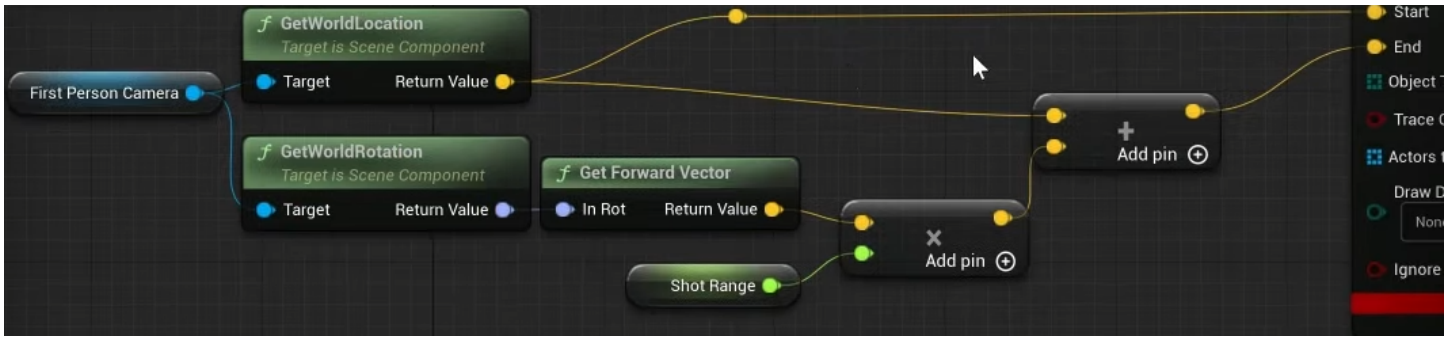
2. Listen to Left Click Release Input
    a. Break out of while loop
3. Listen to E Pressed Input
    a. Break out of while loop
    b. Check if object is interactable
        i. True: Apply linear force to object in mouse world direction
        ii. False: TBD

Shortly after getting started, I had already run into a problem. In Unreal Engine 5, line traces are used in place of raycasts. The difference lies in the fact that raycasts are rays, hence they have no end point and extend indefinitely. Line traces do the same thing as raycasts, but have a defined endpoint. This becomes an issue when the mouse isn't locked to the center of the screen with the first person camera. Should the mouse be locked to the center of the screen, I could simply line trace from the camera to the mouse world position. However, the mouse moves around, so I must line trace with the camera information alone.

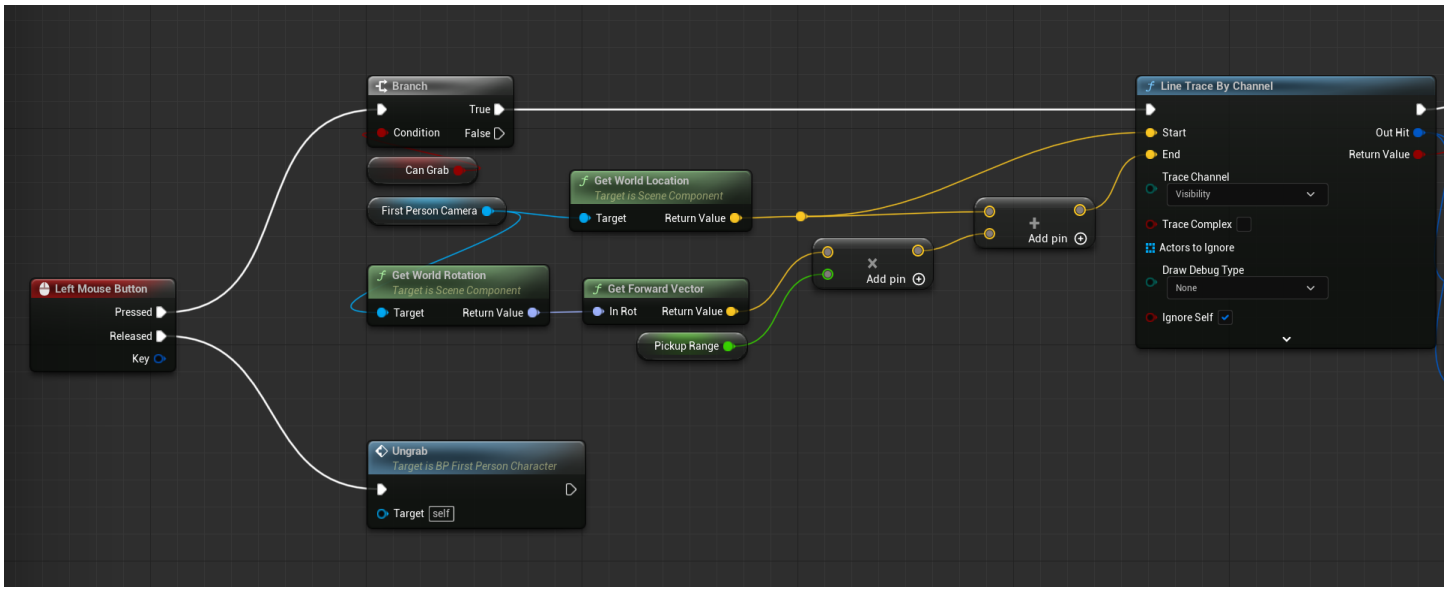
Unreal Engine 5's *Line Trace By Channel* node takes in a starting and ending vector position. The issue is, the camera has only a position value and a rotation value. To get the end position, I would need to either do math or figure out a node that can transform a rotator type value to a vector 3 direction, which can then be multiplied. For the next few hours, I try to mathematically solve this problem with a familiar approach, physics vectors (thanks Mrs. Wissler). Ultimately, I abandoned this method for a few reasons.

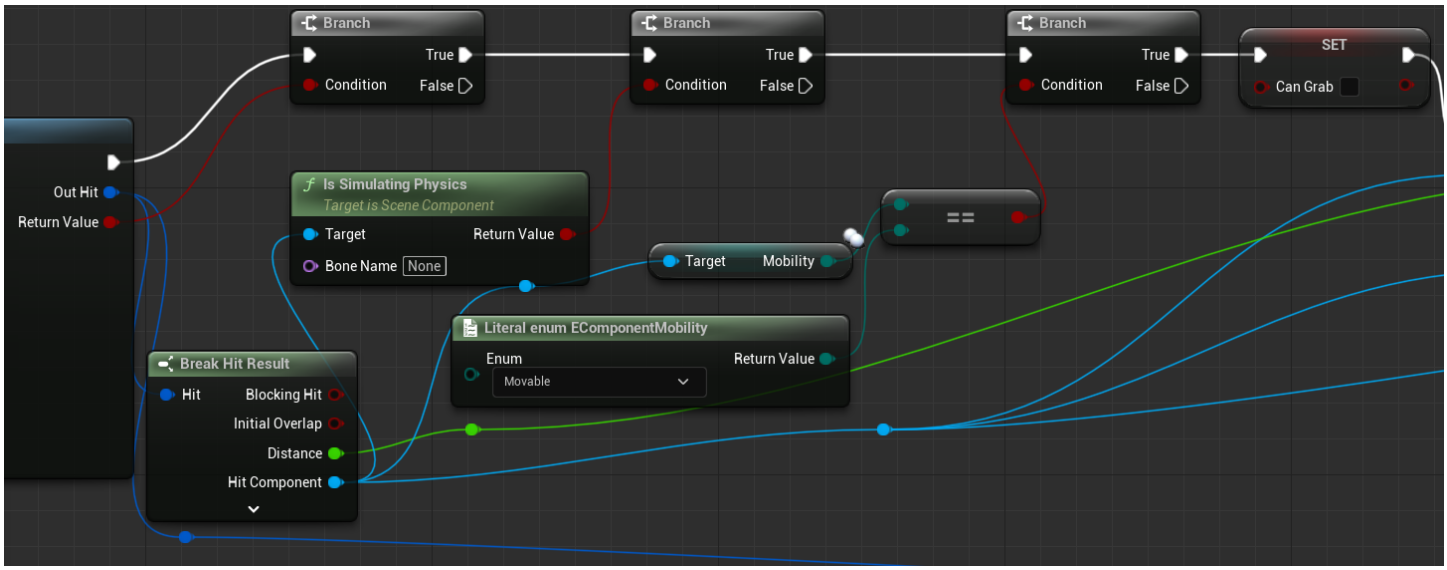
1. I was unfamiliar with doing trigonometry in 3 dimensions
2. It was complicated, messy, and not performant
3. I had a new idea

Before moving onto this new idea, I gave Mrs. McFadden a message to see if I could later come back and solve this problem the mathematical way. More on this later. After consulting with a friend well versed in Unreal Engine 5, he sent me an image of a node doing exactly what I had needed (thanks Andrew):

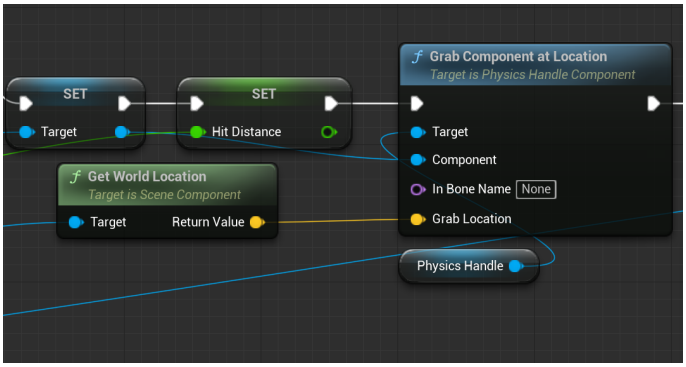


With the help of the image, I was able to get the hit detection working. I had also further implemented checks to assure the target grabbed is able to simulate physics.

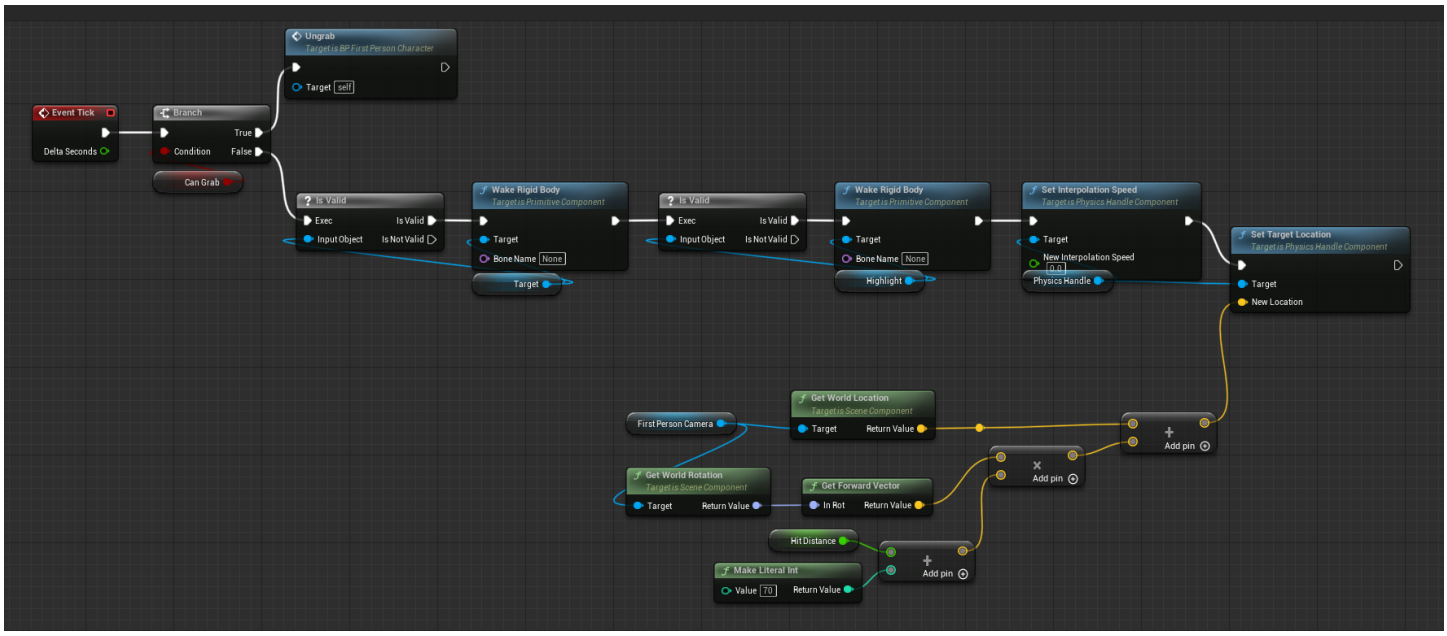




Now I needed to implement the while loop to keep the target object's position where the player is facing. However, using a while loop would get quite messy and tedious with the condition. Instead, I settled with a tick event (logic executed every frame) and decided to use the physics handle component to deal with the target position of the object. With the help of [Lusiogenic's guide], the general structure of this grab blueprint has been fairly easy.



(Adding Target to Physics Handle)



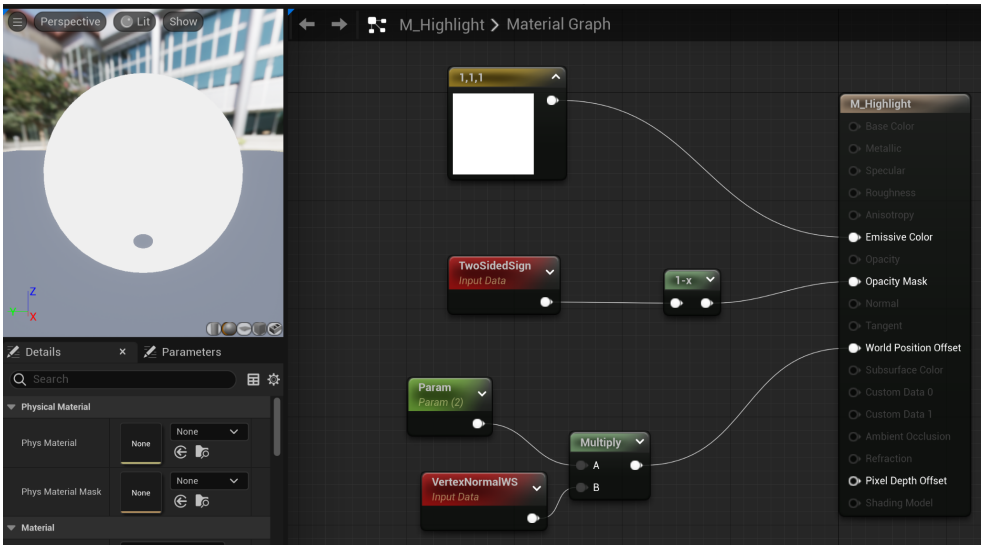
(Event tick logic; Setting target object position)

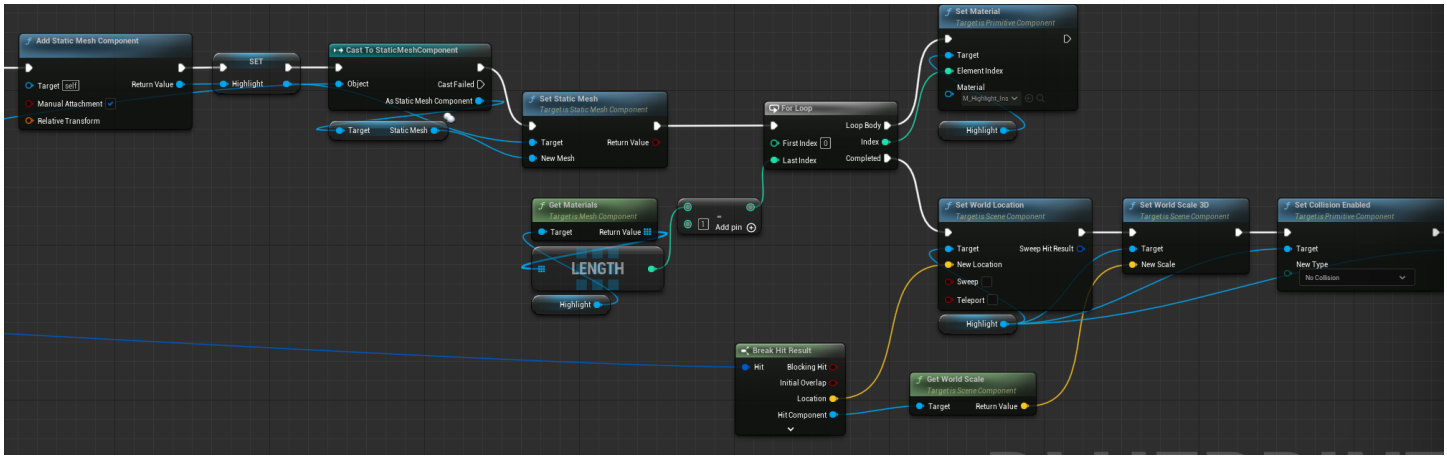
Feedback & Fine Tuning

As this grab system was slightly inspired by an old game I used to play on roblox called Lumber Tycoon 2, I had initially planned to have the object grabbed by the point at which the line trace collision occurred, giving a hanging/dangling effect. However, as shown in the images above, I decided that having the objects grabbed by their center of mass to be more visually appealing.

Highlight

Additionally, in Lumber Tycoon 2, wherever you grabbed, there appeared a dot to indicate that the object was being grabbed. This dot doesn't look very logical when the object is grabbed from the center rather than a point, so I decided to highlight the object instead. Once again calling upon my good friend Andrew, he sent me a good video reference. With the guidance of the video and a developer forum post, I was able to make a highlight material and correctly highlight objects (clone and change material of target object):





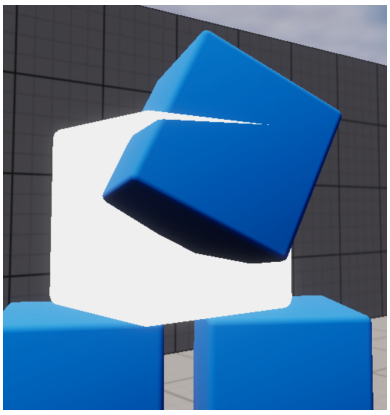
With this incomplete logic, the highlight has only been spawned and doesn't stick to the target object. With this, I tried numerous methods to try and get the highlight to follow the object, but this proved more difficult than I imagined.

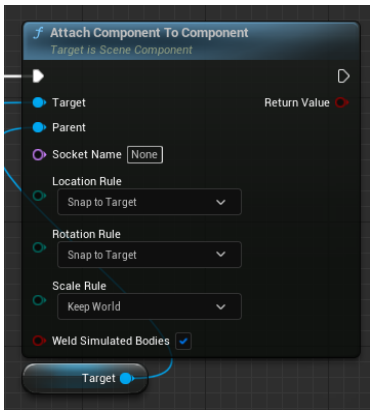
Attempted Fixes:

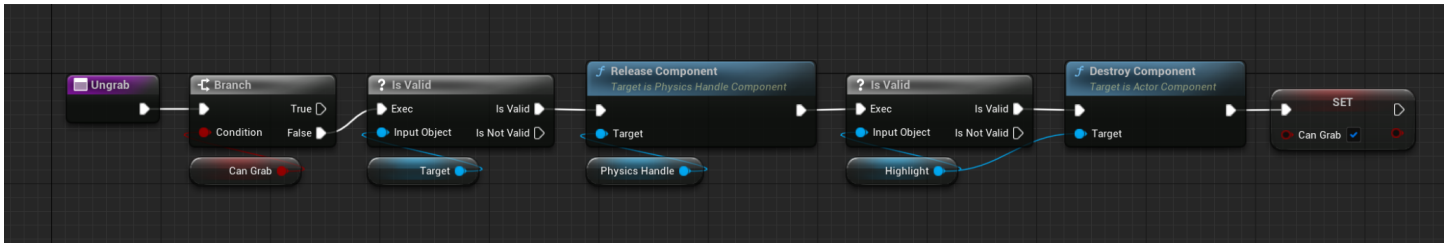
- Update highlight position every tick
  - Not performant
  - Highlight lagged behind target object
- Add highlight to physics handle
  - Both highlight and object couldn't simulate physics any longer (Still not sure why)
- Physics Constraint
  - This didn't even make any sense

After a good night's sleep followed by some developer forum searches, I found a node that did exactly what I needed. Turns out I was overcomplicating such a simple solution.

With a working highlight and physics carry, I finished up the ungrab function to clean up after every grab. I added basic checks (Is Valid) to assure components exist before deleting them, preventing runtime errors (like a good programmer).



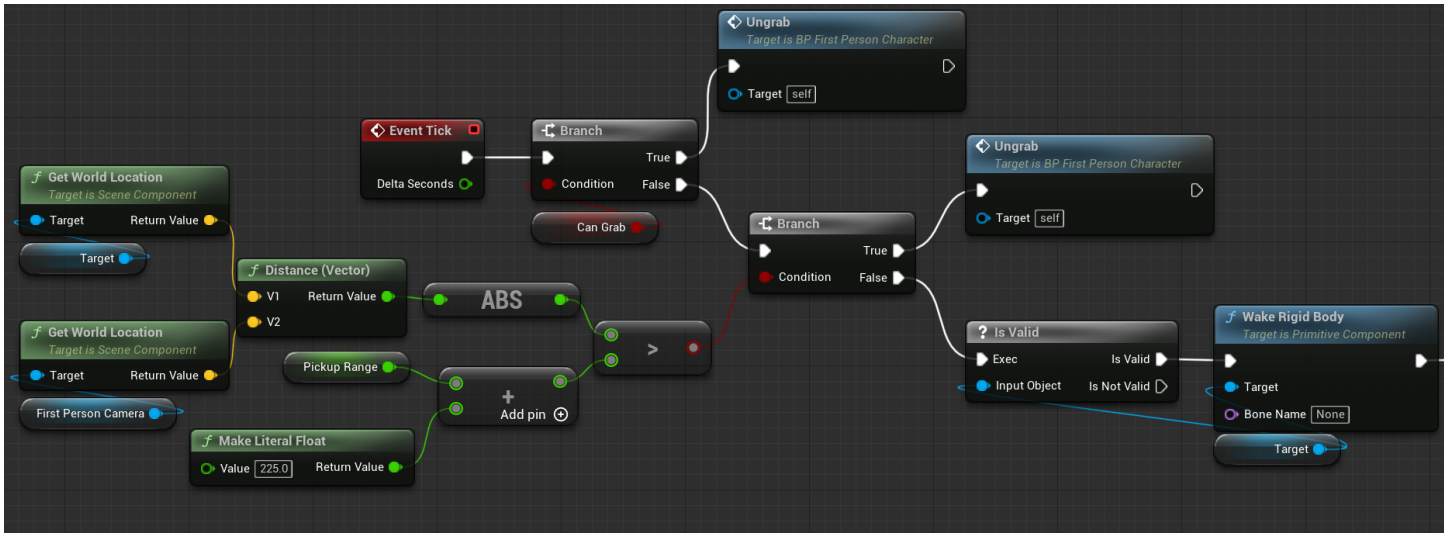




(Ungrab function)

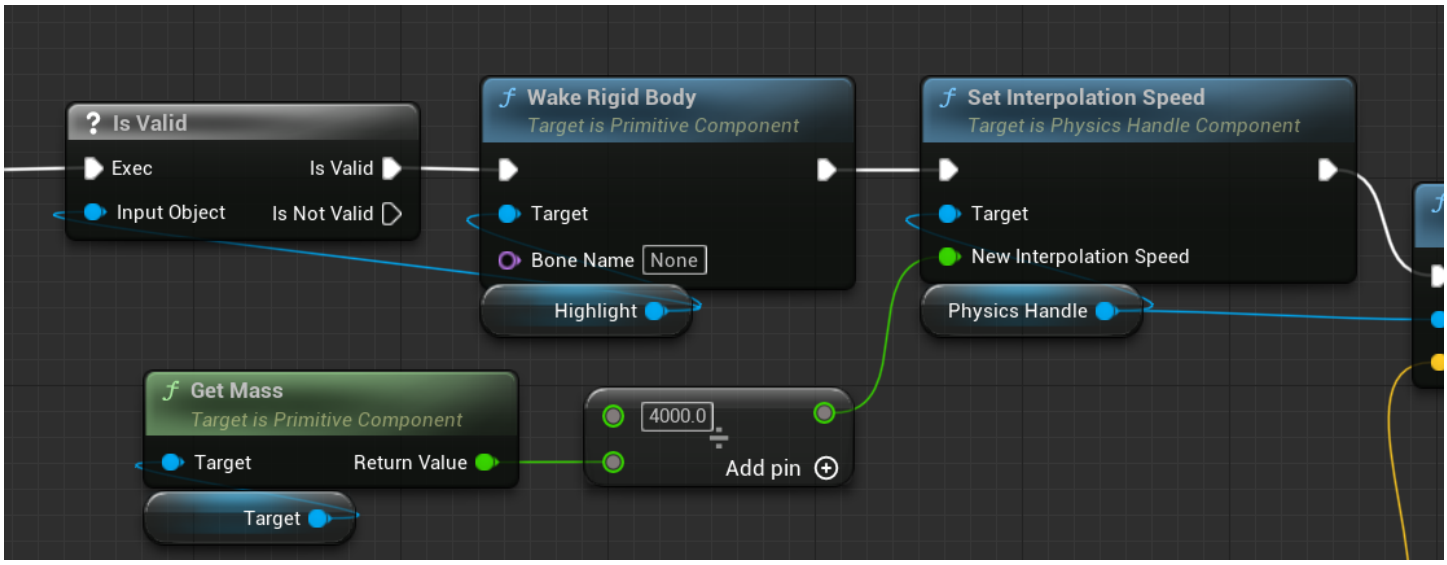
Max Carry Distance

To make the carry system seem a little more real, I added a simple if statement each tick to check the distance between the camera and the target object. If the object is too far away, ungrab is called.



Simulated Weight

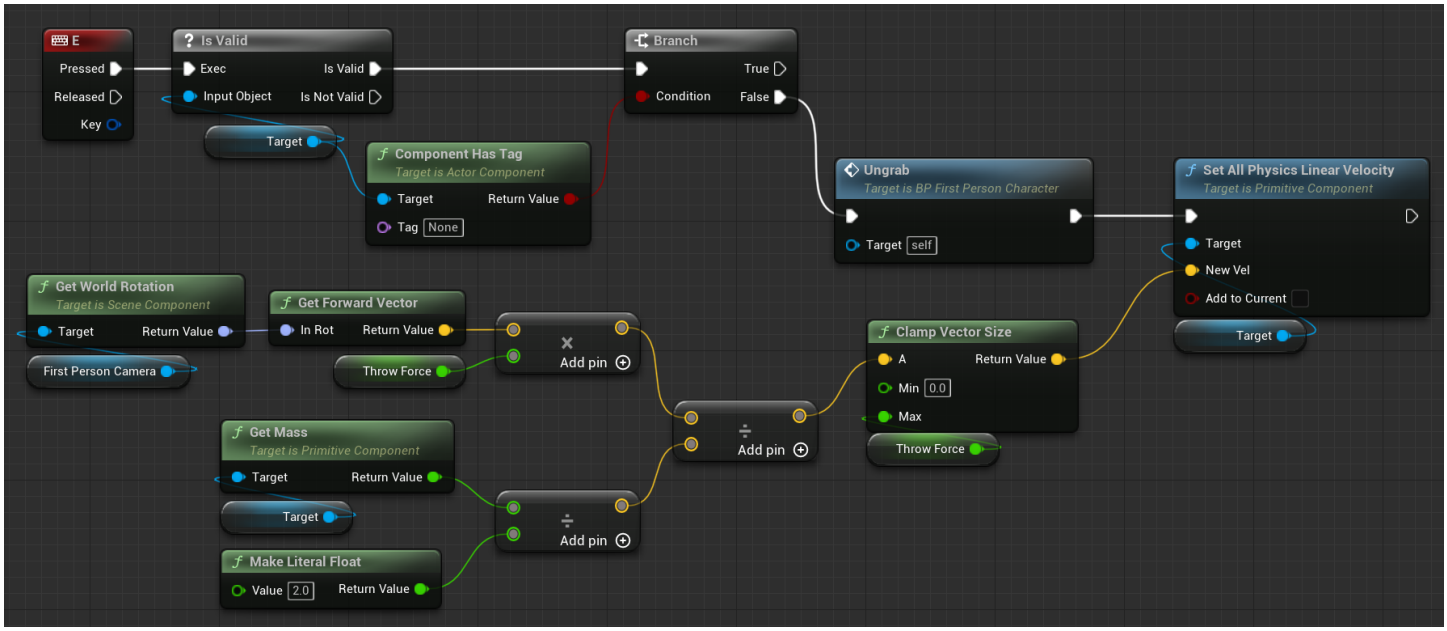
Carrying objects of different sizes doesn't feel "realistic" when you can't feel the weight of the object. Considering this is a virtual game, "feeling" the weight isn't possible, but faking the feeling of objects having weight is. The interpolation speed of the target position is now based on the mass of the object divided from the max carry force.



Target Object Interaction

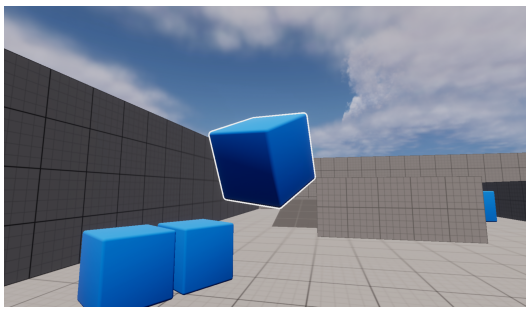
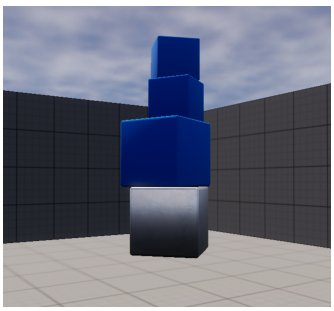
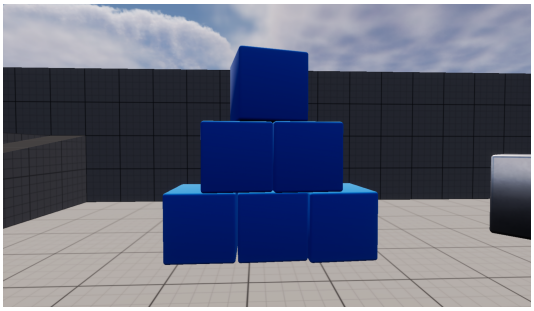
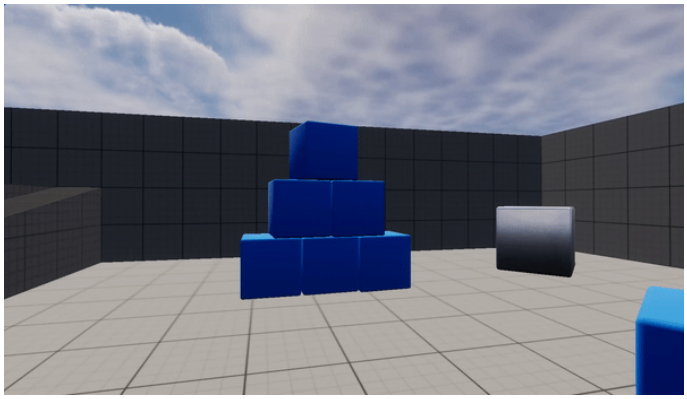
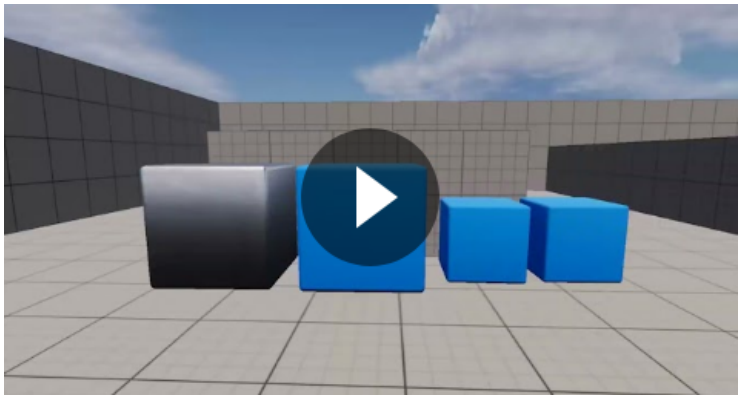
When an item is in your hand, you can click E to interact with the object. If the object is an interactable object, clicking E will activate whatever it's supposed to do. Since I'm not yet prepared for this, I focused on the aspect of clicking E to throw non-interactable objects.

To simulate weight on the object being thrown, the heavier the object, the shorter the distance it travels. This was achieved by dividing the throw force by the mass of the object.



Line Trace Endpoint Revision (Mathematical Method)

Demonstration/Showcase

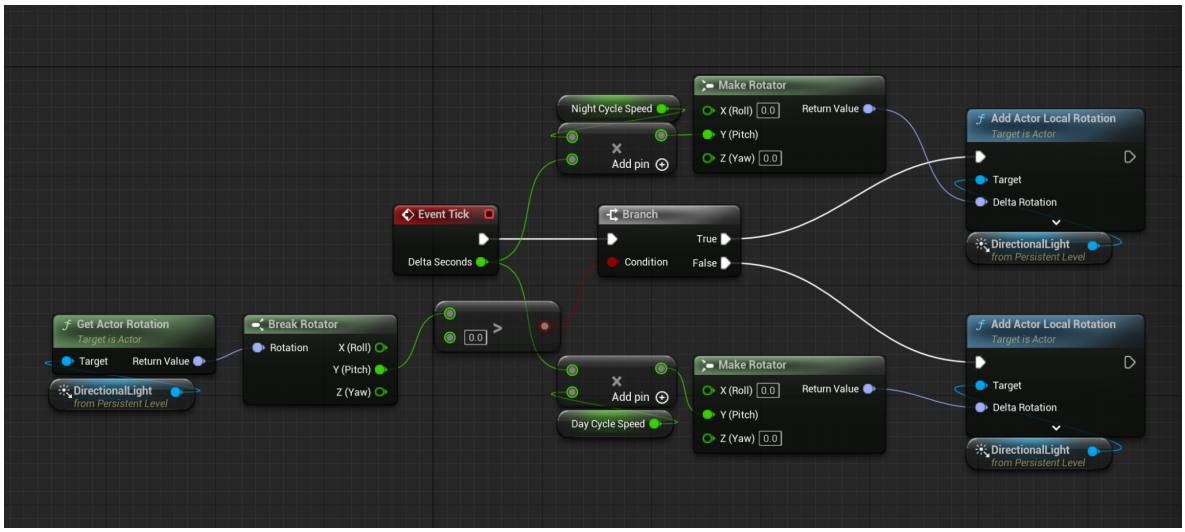
Details

As I decided on the next big thing to work on for this project, I wanted to get some smaller details of this game out of the way. Starting with a day/night cycle, I wanted to give players a sense of time.

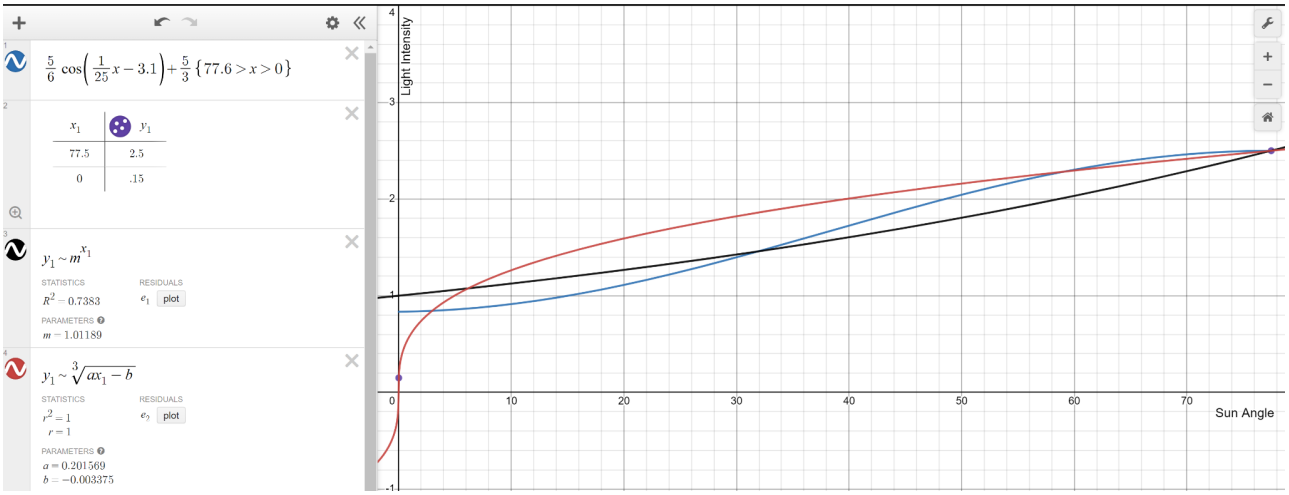
Day/Night Cycle

This was something that I took an approach at with zero knowledge, so I believed watching a youtube video or two would quickly get me started. Starting with *I See 3D's* 90 Second Guide, the logic seemed simple enough. Connect to an event tick, use the delta seconds to add to the sun's Y-Axis rotation, then update the sun's position. The issue is, *I See 3D's* template seems to have been using a blueprint for the light source. This meant the update node that I needed to call didn't exist for me. Despite looking through four or five more videos after *I See 3D's* none seemed to be as simple as his. As I didn't need or want anything complex, I took the initiative to make my own cycle blueprint.

Within 10 minutes, I had a working cycle blueprint with variables for night and day lengths:

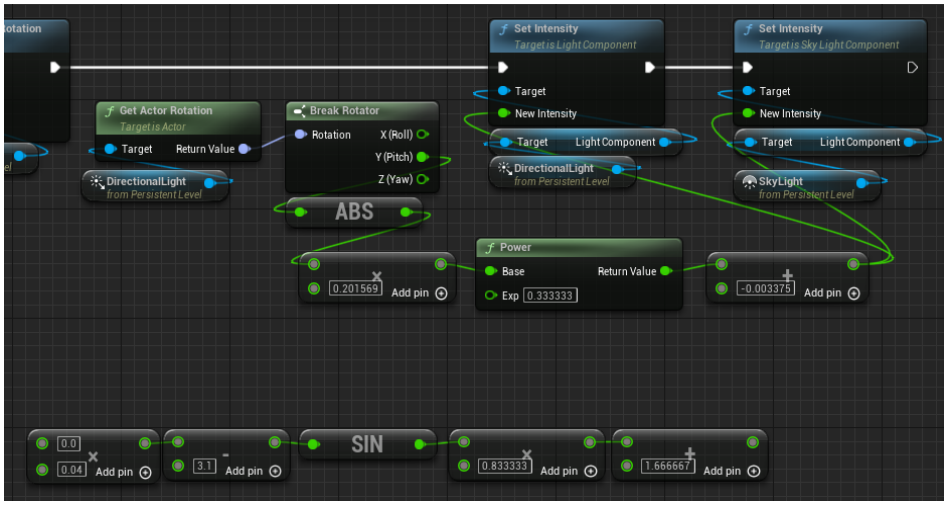


To further add to the feel of "time passage," I needed a function to interpolate the brightness of the world correlating to the sun angle. Running some regressions in desmos, I had three interpolation models that fit:

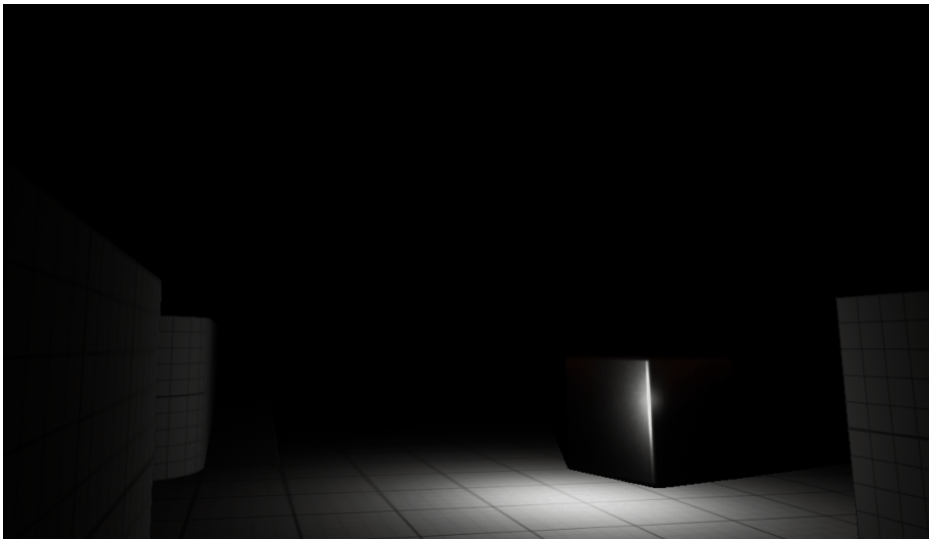


This was where I ran into a minor annoyance with using blueprint nodes: <u>when implementing functions, nodes can chain into long blocks of logic when its C++ counterpart can complete the same task within lines.</u>

In the end, I settled with the cube root function, as it gave the most subtle day interpolations and sharp nights.



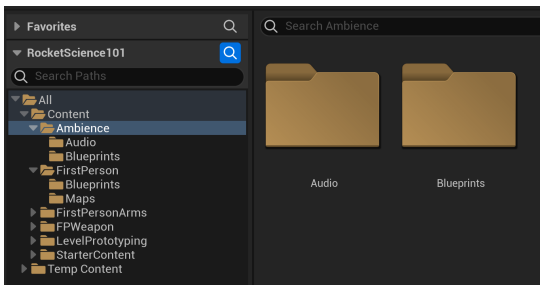
(The bottom equation is a cosine function, while the one used is a cube root)



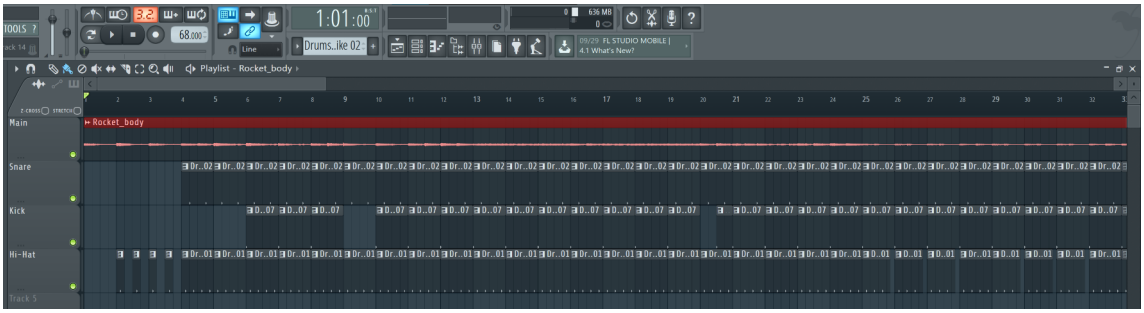
Ambience & Music

Adding to the depth of the game and user experience, color, music/ambience, sound design, and world design are the pillars I believe are the most important aside from the main components of the game. I have made some cool plans for tackling world design and color, but that's for another section. Today, I'm working on music and ambience.

As for the ambience setup, to the right is how I've organized my folders. Setup is quite simple consisting of only two folders.



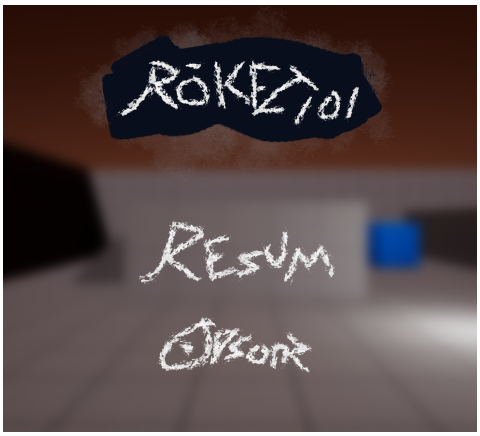
Although I'm not a professional musician with perfect pitch, I do play the guitar and was able to put together a simple track for the menu screen in FL Studio.

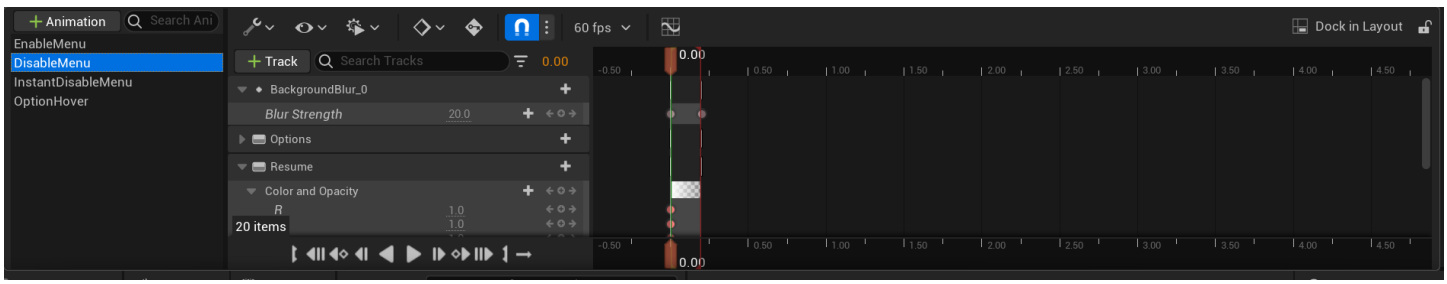


To manage all soundtracks and ambience sounds in an organized fashion, I've made an Ambience blueprint where all logic pertaining to controlling soundtracks and ambiences will be. As for the menu's sake, I only need a toggle function (to fade in/out audio) and an update volume function to sync the option volume slider with the volume.

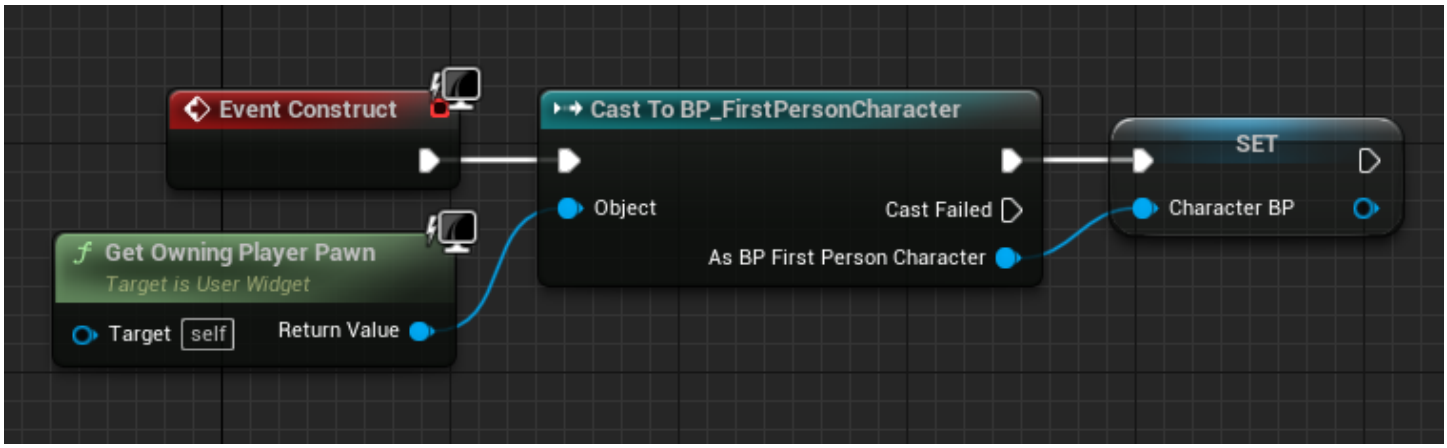
Menu Screens

Since I have yet to make the loading screen and determine how players load up the game, I decided to push the main welcome/menu screen for later and work on the pause screen now. Putting my school issued Adobe Photoshop license to good use, I made some simple icons and a logo for the game. With some time and exploration in Unreal Engine's widget blueprint, I quickly found how to animate and work with UI. To say the least, it's much easier to utilize than with other engines or even web development (libraries and frameworks like Rodux made this process a mess).

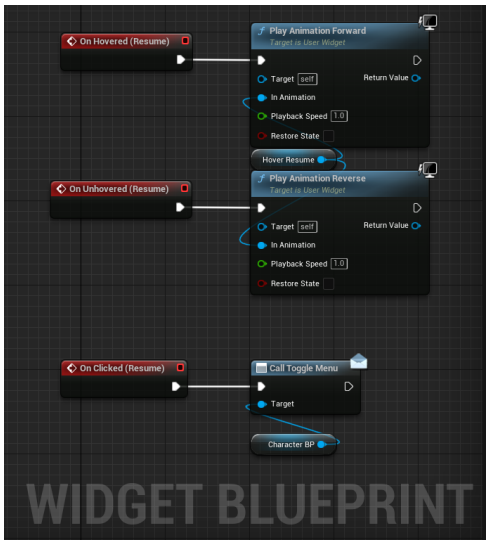




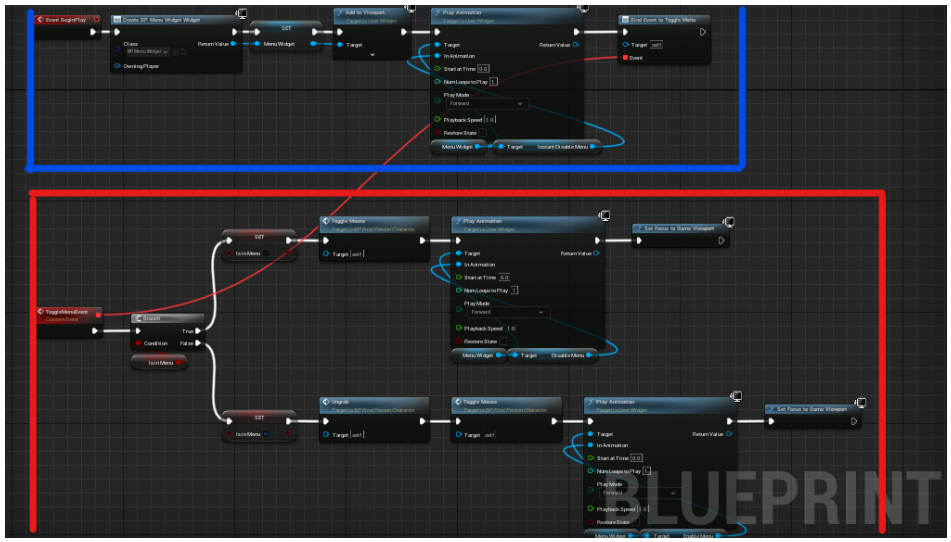
In any given widget blueprint, a UI editor is provided alongside with an event graph to control animations and functions. For this menu specifically, I needed to let the character controller know when the menu had been toggled so that movement and interactions can be disabled accordingly. However, I've previously had an issue with communication between blueprints, and I once again needed to implement this (blueprints will never replace C++). With the help of a Developer Forum post by JoSF, I was able to figure out fairly easily how to assign a target blueprint to a variable. This is my implementation of assigning the character blueprint to a variable in the widget event graph:



Now with access to the character blueprint, I can make calls to any functions/events within the blueprint. Here's my set up for toggling menu screens:

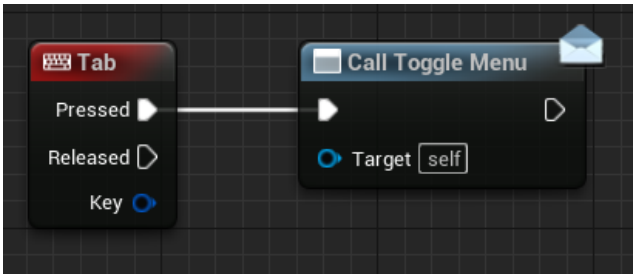


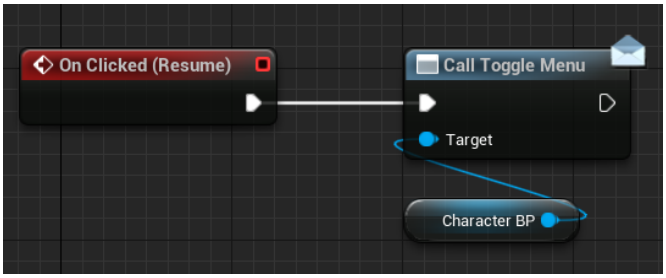
Top two events are hover animations; Last event is disable menu call on clicked event



The logic highlighted in blue is like the main function of a Java driver class, setting up the widget on initialization as well as binding the "toggle menu" event to the logic highlighted in red.

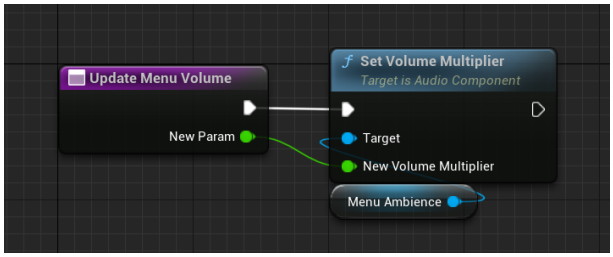
To call the binded event, a simple call can be made from either within the blueprint itself (left) or in another blueprint (like in the widget event graph to the right) given the target blueprint object.





At this point, I had a working prototype of the menu as well as some simple music to go along with it. Putting everything together in one single blueprint would become a giant mess however, so I opted for making a music/soundtrack controller blueprint.

This ambience blueprint was then hooked up to a slider in the options menu to control the menu soundtrack volume. Future ambience/soundtrack will also be quick and easy to set up.



Personally, UI design and management has always been something I dislike doing, much rather preferring straight development of the game's core. So with the option screen still in my way, I had Connor draw some icons and scanned them into the game.

With this out of the way, the core of the game is the next item on the list: Rocket components and lift-off mechanics.

Rocket Component System

With the goal of the game being to exit the atmosphere (hopefully intact), players need to assemble a rocket with components found scattered around the world. Before moving on with Blueprints as usual though, I'd like to try something different.

Over the past few months, working with Blueprints has had its ups and downs. However, as of lately, I've been seriously missing the robustness of traditional keyboard and IDE. Visual scripting has its own limitations and is fairly new to the industry, so for this part of the project, I've decided to settle back into Visual Studio, working with C++.

Planning

Every rocket is built upon a rocket body. To keep track of a rocket's rating, the body will record a value called potential, hidden under the hoods of the game. Depending on the components tacked on, the score is either increased or decreased. Each launch milestone has a rocket potential threshold to meet, allowing the rocket to pass that specific stage. Otherwise, 💥. Below in parentheses and italics will be the effect of each component on the rocket's potential score.

Components

1. Rocket Body
    - ~~a. Cone Body *(+15)*~~ ✔
    - ~~b. Icicle Body *(+10)*~~ ✔
    - ~~c. Driver Body *(+22)*~~ ✔
    - ~~d. Triangle Beam *(+15)*~~ ✔
    - ~~e. Rectangle Beam *(+10)*~~ ✔
    - ~~f. Dog *(+23)*~~ ✔
    - ~~g. Shuttle Body *(+25)*~~ ✔
    - ~~h. Yoga Ball *(-22)*~~ ✔
2. Engine
    - ~~a. Car Engine *(+7)*~~ ✔
    - ~~b. Radiator *(+5)*~~ ✔
    - ~~c. Blender *(+5)*~~ ✔
    - ~~d. Motor *(+5)*~~ ✔
    - ~~e. Toaster *(+14)*~~ ✔
    - ~~f. Budai *(+30)*~~ - *Sketchfab*
      Model Edited ✔
    - ~~g. Fuel Tank *(+26)*~~ ✔
    - ~~h. Magical Fire Gem *(50)*~~ ✔
3. Thrusters
    - ~~a. Single Cone Thruster *(+12)*~~ ✔
    - ~~b. Twin Rockets *(+18)*~~ ✔
    - ~~c. Driver Rocket *(+27)*~~ ✔
    - ~~d. Triplet Thrusters *(+10)*~~ *(Shuttle Body)* ✔
    - ~~e. Golden Umbrella *(-45)*~~ ✔
    - ~~f. Desk Fan *(+28)*~~ ✔
4. Fins
    - ~~a. Round Triangle Fins *(+7)*~~ ✔
    - ~~b. Sharp Triangle Fins *(+5)*~~ ✔
    - ~~c. C4 *(-100)*~~ ✔
    - ~~d. Witch's Broom *(+17)*~~
    - ~~e. Cat *(-7)*~~
    - ~~f. Cutting Board *(+10)*~~

Rocket Launch Milestones (Potential Score Thresholds)

a. Lift-Off *(15)*
b. Atmosphere *(50)*
c. Moon Crash *(100)*

Component Modeling Assigning:

- **Me**
- **Pumpum**
- **Sourced**

Learning C++

Types & Files

Preprocessor Statements - *Executed before compilation; often to import files (ie. header file)*

- *<Angular brackets> are to search include path*
- *“Quotations” are to search all of the project*

Header Files - *A place to store declarations to later be imported into .cpp files that need access to them*

Namespaces - *The library actually called and used*

```
C/C++
#include <iostream> // Preprocessor Statement

using namespace std; // Namespace

int main() {
	cout << "Hello, World" << endl;
}
```

Narrowing Conversions - *The loss of information through conversions*

- *Prevent with list-form:* $\{ x \}$

const vs. constexpr:

- const - *Evaluates value at run time and ensures no change*
- constexpr - *Evaluates at compile time and ensures no change; stores in more stable, read-only, memory*
    - *constexpr **functions** can evaluate non-constant parameters*

```
C/C++
const double s1 = sum(v); // Compiles
constexpr double s2 = sum(v); // Error
```

Pointers - *Reference directly to a memory address, not the value*

- *Value at memory address can be accessed later*

```
C/C++
char* pointer = &v[3]; // Points to the memory address of an array's 4th element
char value = *pointer; // Dereferences the pointer/memory address (accesses the value)
```

*NOTE: While arrays **cannot** be passed into functions as parameters, you can pass pointers to arrays*

Memory

Bytes - *One byte stores 8 bits of data*

NOTE: Variables can be broken down simply to "memory allocation" (types are different in memory size)

```
C/C++
// Types
bool // 1 Byte (Any value other than 0 is 'true')
char // 1 Byte
int // 4 Bytes
float // 4 Bytes (signified by appending 'F' or 'f')
double // 8 Bytes

// Modifiers
unsigned // Uses the saved 'sign' byte to store more data
short // 2 Bytes
long // Usually 4 Bytes
long long // 8 Bytes
```

*NOTE: Type sizes are dependent on compiler (and therefore machines), so '*sizeof(type)*' can be called*

Pointers - *Can be initialized with the **starting** value of the memory block written to*

- ***memset(pointer, value, size)** - Fills a block of memory given the starting memory address (a pointer), value to fill with, and size*

```
C/C++
char* buffer = new char[8]; // Since a char is one byte, this allocates 8 bytes of memory
memset(buffer, 0, 8); // Sets the 8 bytes of memory to 0's
delete[] buffer; // Deletes the pointer
```

References - *An alias for an existing variable*

*NOTE: References are similar to pointers, but are **not** variables. (Everything references can do, pointers can too)*

```
C/C++
void Increment(int value) { value++; }
void Increment(int& value) { value++; }

int main ()
{
	int a = 5;
	Increment(a); // "a" is copied into the function; "a" remains 5 (pass by value)

	int b = 5;
```

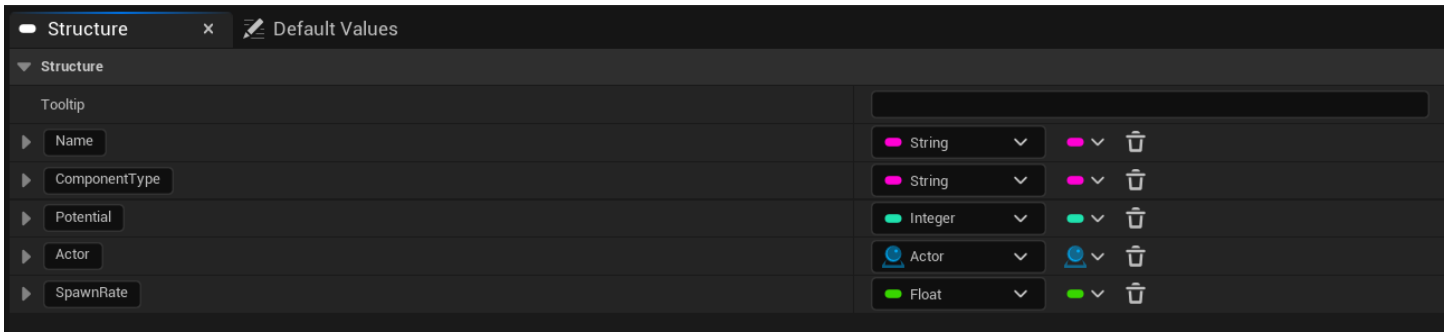
```
        Increment(b); // A reference to "b" is passed; "b" is 6 (hence, pass by reference=)
}
```

Syntax

Pointers & References		Function Calls	
type*****	*// Pointer of a type*	namespace**::**	*// Calls a static function*
*****variable	*// Dereferences a pointer*	class**->**	*// Overloads parameters*
&value	*// Memory address of value*	**.**method()	*// Calls a method*
type**&**	*// Reference of a type*		

Handling Data

While looking through Unreal's documentation, I came across something quite interesting. Unreal provides an object class called a *Structure* and a *Data Table*. These two do exactly as their names imply: provide a customizable template of data and a list/collection of that data. The data table can then be read via blueprint or C++ later in runtime and become malleable data. For our game, I've made a structure of rocket components and a data table storing all of the planned out components:



Name - The display name of the component

Component Type - A string to be initialized as an Enum in C++

Potential - The effect of the component on the overall rocket

Actor - The mesh/model associated with the component

Spawn Rate - How often the component occurs in the world

Data Table | Data Table Deta...

Search

	Row Name	Name	ComponentType	Potential	Actor	SpawnRate
1	Cone_Body	Cone Body	ERocketBody	15	None	0.053571
2	Icicle_Body	Icicle Body	ERocketBody	10	None	0.035714
3	Driver_Body	Driver Body	ERocketBody	22	None	0.053571
4	Pyramid_Body	Pyramid Body	ERocketBody	2	None	0.035714
5	Rectangle_Body	Rectangle Body	ERocketBody	-10	None	0.017857
6	Dog_Body	Dog	ERocketBody	23	None	0.003571
7	Shuttle_Body	Shuttle Body	ERocketBody	25	None	0.046429
8	Yoga_Body	Yoga Ball Body	ERocketBody	-22	None	0.003571
9	Car_Engine	Car Engine	ERocketEngine	7	None	0.071429
10	Radiator_Engine	Radiator Engine	ERocketEngine	5	None	0.017857
11	Blender_Engine	Blender Engine	ERocketEngine	5	None	0.017857
12	Motor_Engine	Motor Engine	ERocketEngine	5	None	0.064286
13	Toaster_Engine	Toaster Engine	ERocketEngine	14	None	0.003571
14	Budai_Engine	Budai	ERocketEngine	30	None	0.001786
15	Fuel_Engine	Fuel Tank Engine	ERocketEngine	26	None	0.069643
16	Gem_Engine	Magical Fire Gem	ERocketEngine	-50	None	0.003571
17	Cone_Thruster	Single Cone Thruste	ERocketThruster	12	None	0.060714
18	Twin_Thruster	Twin Thrusters	ERocketThruster	18	None	0.060714
19	Driver_Thruster	Driver Thruster	ERocketThruster	27	None	0.060714
20	Triplet_Thruster	Triplet Thrusters	ERocketThruster	10	None	0.060714
21	GUmbrella_Thrus	Golden Umbrella	ERocketThruster	-45	None	0.003571
22	Fan_Thruster	Desk Fan	ERocketThruster	28	None	0.003571
23	Rounded_Fins	Rounded Fins	ERocketFins	7	None	0.121429
24	Sharp_Fins	Sharp Fins	ERocketFins	5	None	0.121429
25	C4_Fins	C4	ERocketFins	-100	None	0.003571
26	Witch_Fins	Witch's Broom	ERocketFins	17	None	0.003571
27	Cat_Fins	meow meow	ERocketFins	-7	None	0.003571
28	Board_Fins	Cutting Board Fins	ERocketFins	10	None	0.003571

After "successfully" setting up the data table and structure, I had trouble processing the data in C++. How do I traverse a data table? Or even better, how do I import it? Surprisingly enough, Unreal Engine's documentation is quite garbage when it comes to this. There was quite literally nothing on reading Unreal assets from Visual Studio. Luckily, I found a reliable youtube video after lots of googling, which ended up pointing me in the right direction.

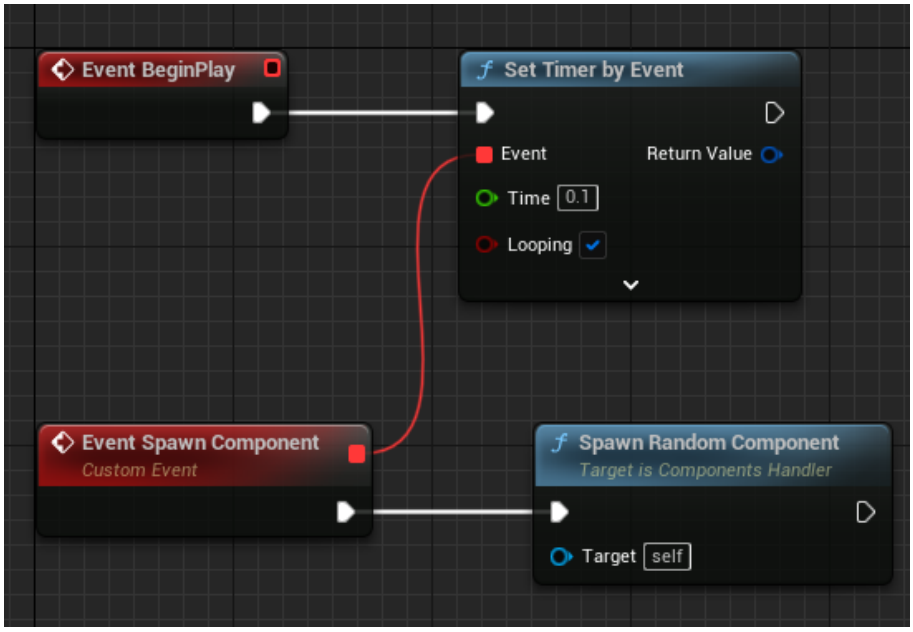
Turns out, I had to define my struct in Visual Studio and import that struct into Unreal. This is so that 1 - C++ has a "copy" of the struct class and 2 - Unreal could sync with Visual Studio on what struct type the data table consists of. Here's how I implemented the data table and struct:

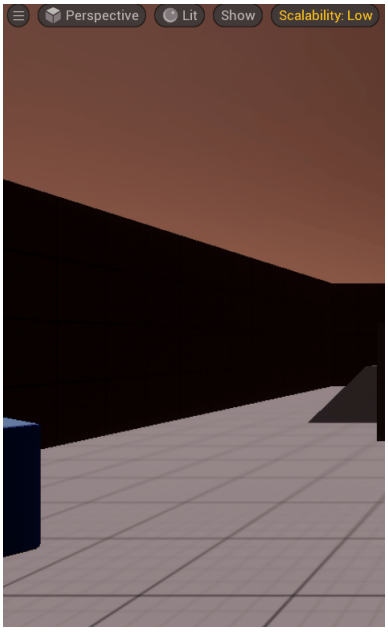
```C/C++
#include "Engine/DataTable.h"
#include "Containers/Array.h"
#include "Math/UnrealMathUtility.h"

USTRUCT(BlueprintType)
struct FRocketComponent : public FTableRowBase
{
	GENERATED_BODY()

	UPROPERTY(EditAnywhere, BlueprintReadOnly)
		FString Name;

	UPROPERTY(EditAnywhere, BlueprintReadOnly)
		FString ComponentType;

	UPROPERTY(EditAnywhere, BlueprintReadOnly)
		int32 Potential;

	UPROPERTY(EditAnywhere, BlueprintReadOnly)
		AActor* Actor;

	UPROPERTY(EditAnywhere, BlueprintReadOnly)
		double SpawnRate;
};

UCLASS()
class ROCKETSCIENCE101_API AComponentsHandler : public AActor
{
	GENERATED_BODY()

// ... Some in between Unreal Generated stuff

public:
	// Called every frame
	virtual void Tick(float DeltaTime) override;

	UFUNCTION(BlueprintCallable, Category = "Rocket Component System")
	void SpawnRandomComponent();

	UPROPERTY(EditAnywhere);
	class UDataTable* RocketComponentsDataTable;
};
```

The struct simply consists of properties and the type of properties in the struct (I later imported to Unreal and copied the data from the previous data table). The public declarations are a variable to add the data table manually (shown later) and declaring a function.

```cpp
// Spawns a random component at a random position
void AComponentsHandler::SpawnRandomComponent()
{
    if (!GEngine) return;

    if (!RocketComponentsDataTable) return;

    const TArray<FName> RocketComponents = RocketComponentsDataTable->GetRowNames();
    const int RocketComponentsSize = RocketComponents.Num();

    const FString& ContextString(TEXT("Rocket Component Context"));
    const FName index = RocketComponents[FMath::FRandRange(0, RocketComponentsSize - 1)];
    const FRocketComponent* RocketComponent = RocketComponentsDataTable->FindRow<FRocketComponent>(index, ContextString, true);

    GEngine->AddOnScreenDebugMessage(-1, 3, FColor::Magenta, RocketComponent->Name);
}
```

Temporary function to test and demonstrate workflow of utilizing both C++ and Blueprints.





After saving and compiling, my C++ class can be dragged into the world like any other actor. This actor will also have an event graph just like other blueprint actors. In the event graph, C++ functions that I have exposed can be accessed (as shown to the left).

In this example, I've hooked a simple timer loop to an event that calls the C++ function *Spawn Random Component* that I wrote earlier. The function simply picks a random rocket component from the data table and prints out the name.

Algorithms

Now with the table of weights and components all sorted out, I need a way to efficiently look through all of the components within the list and select one based on the randomized weight. The logical and mathematical way of doing this would be to multiply all of the weights by a number such that all of the weights become whole numbers. Going through the whole number of each item, you can pick out a random component. However,

this method isn't performant and requires an unnecessary amount of processing. WIth that in mind, I've come up with two methods of sorting through a weighted list:

Method One - Boundaries

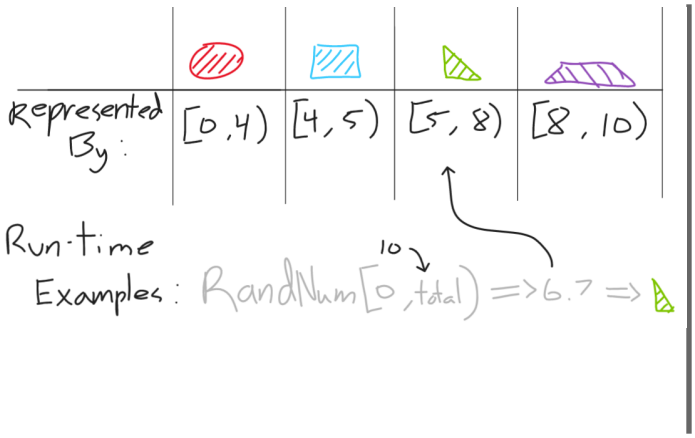
1. Create a running total of all weights
2. Loop through all components
   a. Add component weight to total
   b. Use the weight to create a bound
   c. Example:
      Comp. A: [0 - 0.123], Comp. B: (0.123, 0.232]

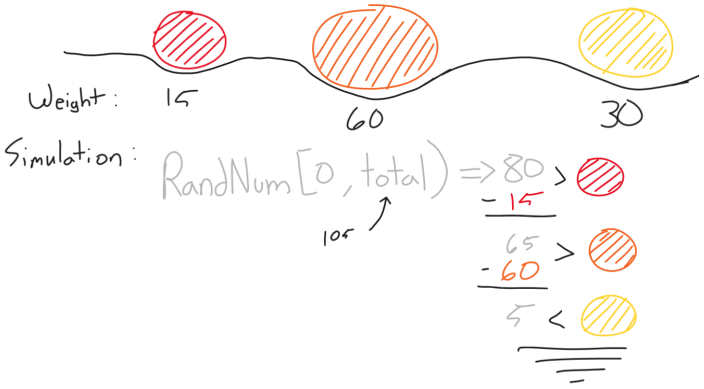
1. Pick a random number (0 - total weight)
2. Find corresponding boundary
3. Grab & Return corresponding component

Method Two:

1. Create a running total of all weights
2. Loop through all components
   a. Add component weight to total

1. Pick a random number (0 - total weight)
2. Loop through all components
   a. If Random Weight is less than component weight:
      i. Grab & Return component
   b. Else:
      i. Subtract random number by component weight





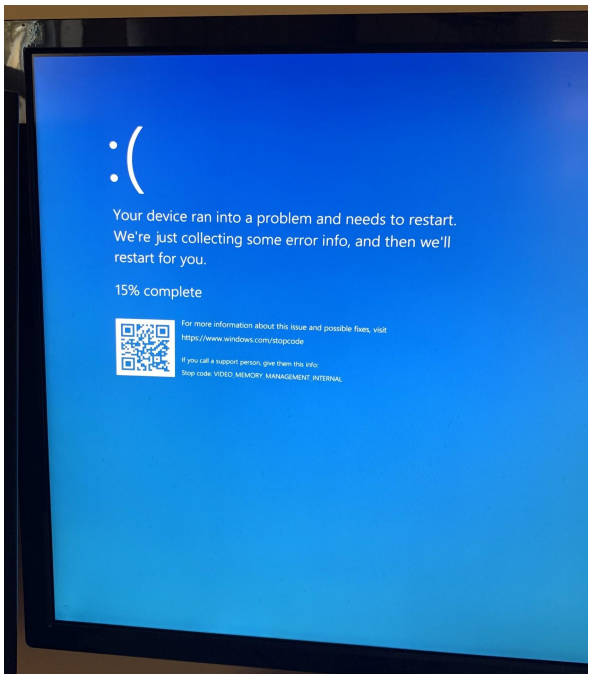
Ultimately, I picked method two for one main reason- Ease of implementation

With method one, I needed to create bounds with the initial setup loop, followed by another loop later to traverse the array and find a bound that includes the random number. On the other hand, I was much more familiar with method two, which happens to be the same way the hit game *Call of Duty* implements their weighted random table for their mystery gun box. Method two requires only two basic steps: adding up all of the weights and an if statement within a loop.

Implementation

```
C/C++
// BeginPlay Method
for (int i = 0; i < RocketComponents.Num(); i++)
{
	const FRocketComponent* component = RocketComponentsDataTable->FindRow<...>(...);
	totalWeight += component->SpawnRate
	componentsWeightBounds.Add(component->SpawnRate, *component);
}

// SpawnRandomComponent Method
if (componentWeights == nullptr) return;
for (auto pair : componentWeights)
{
	if (randomWeight <= pair.Key)
	{
		selectedComponent = pair.Value;
		GEngine->AddOnScreenDebugMessage(... , pair.Value)
		break;
	}
	else
	{
		randomWeight -= pair.Key;
	}
}
```

Logically looking through this code, everything matches the algorithm and I have checks in place to prevent errors, since errors with pointers are scary 😱. So then I hit the compile and run button:



Not only had Unreal crashed, I somehow blue screened my entire PC, 🫠. After reviewing my code the error Unreal spit out, removing the null pointer comparison seemed to have fixed the issue.

But after being able to run the code and see the prints, there's a new issue:

I love C++! 🫡

After nearly two hours of debugging, I believe I have an explanation for why different characters are printing: inside the for loop, a temporary variable is created to access the values of the map (*pairs* in my case). When I try to print out this value, by the time it reaches Unreal, the memory has already been freed by C++ to save processing power. This causes the pointer to reference a value in the memory that is now overridden which happens to be these characters (it often printed nothing as well). Here's the revised version:

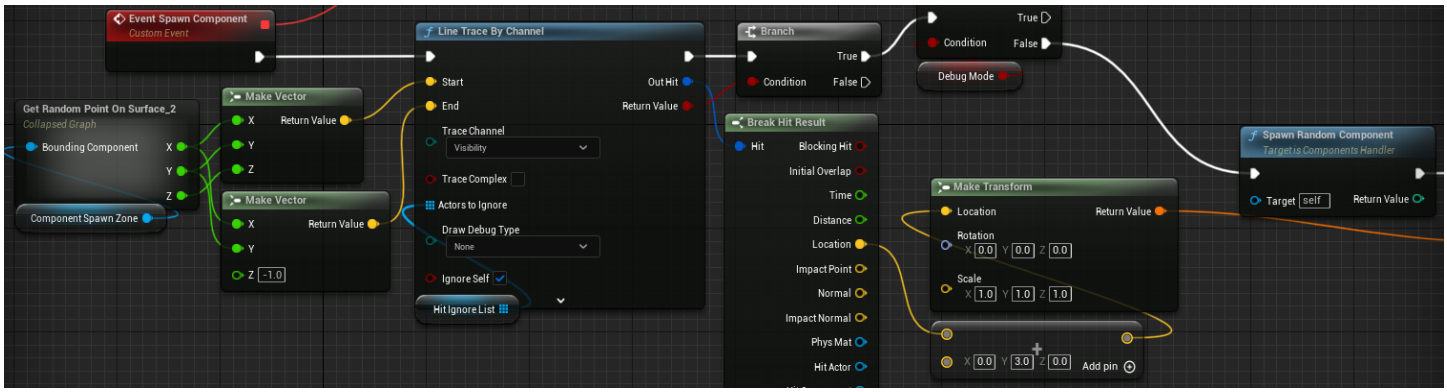
```
C/C++
#include "ComponentsHandler.h"
#include <iostream>
#include <string>
#include <Containers/UnrealString.h>

using namespace std;

// Sets default values
AComponentsHandler::AComponentsHandler()
{
	// Set this actor to call Tick() every frame.  You can turn this off to improve
performance if you don't need it.
	PrimaryActorTick.bCanEverTick = true;
}

// Called when the game starts or when spawned
void AComponentsHandler::BeginPlay()
{
	Super::BeginPlay();

	// Guarding if for default class parameters
	if (!RocketComponentsDataTable) return;

	const FString& ContextString(TEXT("Rocket Component Context"));
	const TArray<FName> RocketComponents = RocketComponentsDataTable->GetRowNames();

	// Assigns weight to every component
	for (int i = 0; i < RocketComponents.Num(); i++)
	{
		const FRocketComponentStruct* component =
RocketComponentsDataTable->FindRow<FRocketComponentStruct>(RocketComponents[i],
ContextString);
		totalWeight += component->SpawnRate * 100;

		// Adding to a map where <key, value> = <Component Weight, Pointer to Datatable
row>
		componentWeightBounds.Add(component->SpawnRate * 100, *component);
	}
}

// Called every frame
```

```
void AComponentsHandler::Tick(float DeltaTime)
{
	Super::Tick(DeltaTime);

}

// Spawns a random component at a random position
int AComponentsHandler::SpawnRandomComponent()
{
	FRocketComponentStruct selectedComponent;

	const TArray<FName> RocketComponents = RocketComponentsDataTable->GetRowNames();
	const int RocketComponentsSize = RocketComponents.Num();

	const FString& ContextString(TEXT("Rocket Component Context"));
	double randomWeight = FMath::FRandRange(0, totalWeight);
	//GEngine->AddOnScreenDebugMessage(-1, 3, FColor::Magenta,
FString::Printf(to_string(randomWeight)));
	int index = 0;

	for (auto pair : componentWeightBounds)
	{
		if (randomWeight <= pair.Key)
		{
			selectedComponent = *componentWeightBounds.Find(pair.Key);
			break;
		}
		else
		{
			index++;
			randomWeight -= pair.Key;
		}
	}

	// GEngine->AddOnScreenDebugMessage(-1, 3, FColor::Magenta, TEXT("Spawn Random
Component"));

	return index;
}
```

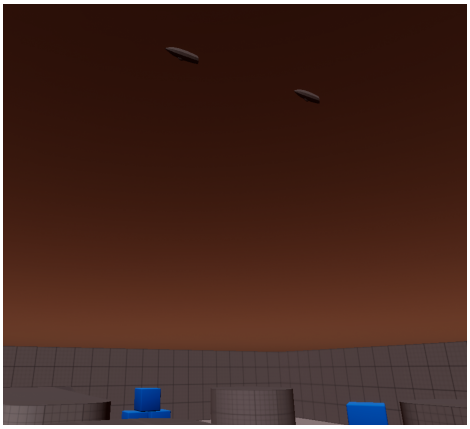
Spawning

To keep systems grouped and organized, I'll be discussing the spawning system of Rocket Components before the creation of the rocket component meshes. However, chronologically, the section discussing Meshes and Actors was done before the spawning system.

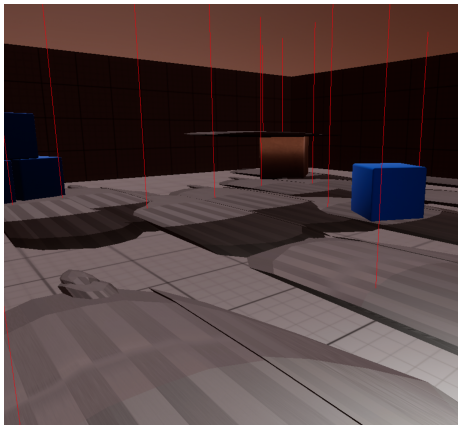
So how does one go about placing objects around a map with uneven terrain? This is a question I've asked myself about four years ago when I was working on my first game on a different engine. Ultimately, I found my answer and worked it out: raycast from the sky down and place an object on the collision point. Similarly, I'll be using the same method to scatter rocket components using the weighted *SpawnRandomComponent* method written in C++. Earlier in this project, I had already worked with raycasts (line trace in Unreal), so this part went fairly smoothly.



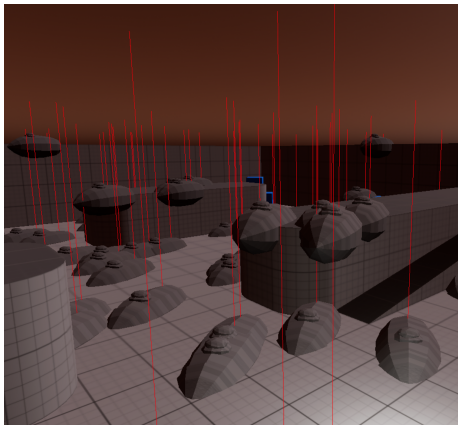
The spawning of the static mesh for pure testing sake. Once all meshes are complete and converted to actors, I'll be revising this logic to accommodate. Like all first program runs, the program never runs as intended. In fact, the first few runs were quite a mess:



Meshes (at the top) were spawning in the wrong place.



I thought the normal node (of hit) could translate to rotation



🎉🥳🎊

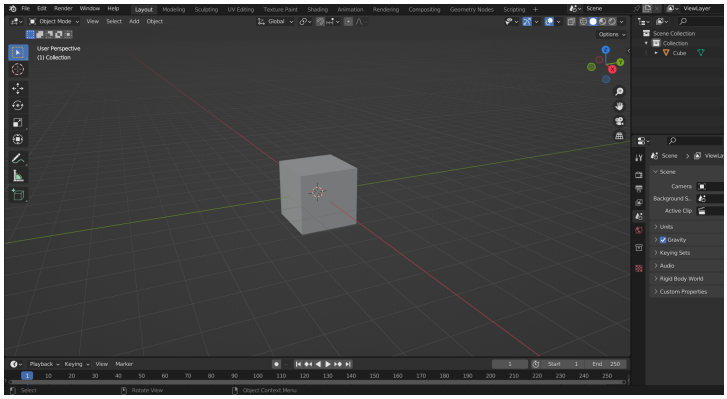
Game Design

Although I won't be going over all of the game design I researched such as music composition, stylized worlds, or UI design, I'll be going over the larger aspects of what I did implement with my friends.

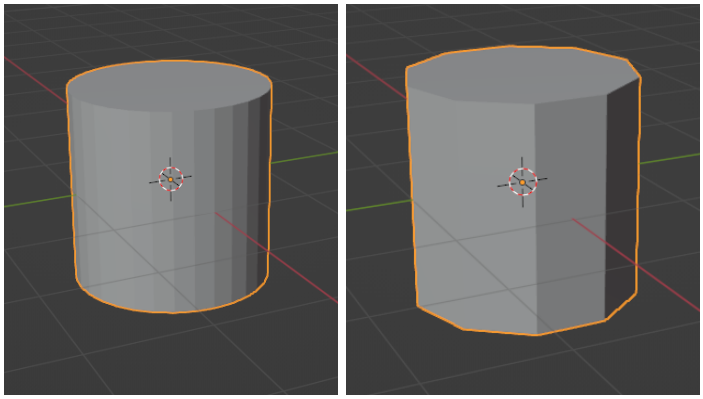
Meshes and Actors

Stylization plays a major role in modeling and texturing. For our case, we decided a low poly, almost cartoon styled, world is best. As I'm the one modeling and texturing all of the components, this takes almost all of the burden of modeling off of my shoulders.

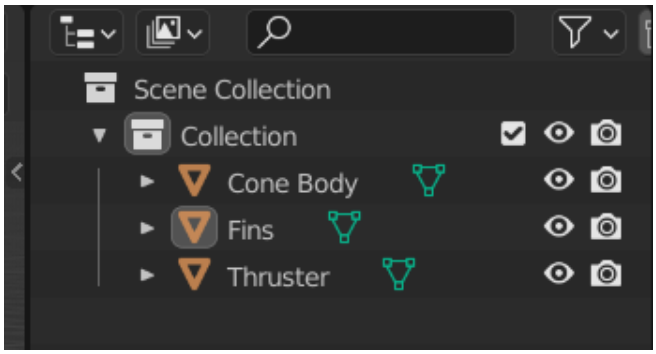
Starting with a basic rocket composed of *Round Fins, Cone Body, and Single Cone Thruster*, I modeled and assembled the rocket in Blender to see how it would look in game. Modeling itself isn't hard (to me), especially when the meshes are low poly, so I'll be glossing over the process:



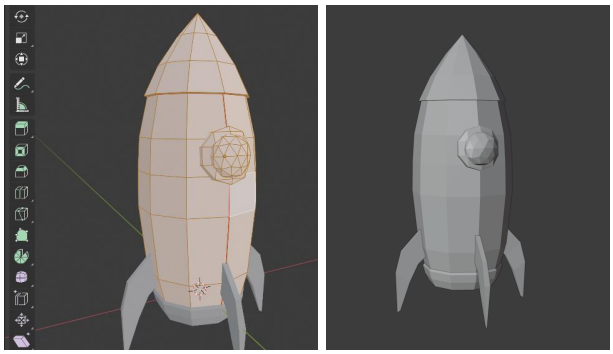
Every blender save opens up with a cube. Since our rocket is a cylinder, we delete this and add a new cylinder.



Since we're trying to create a low poly look, we lower the vertices count from 32 to 10

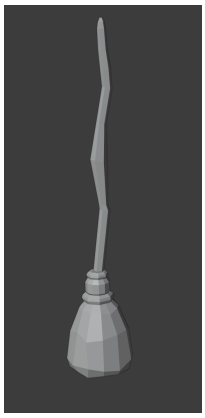


Keeping each component separated in the collection bin, I can later export the mesh as one file but import the file as multiple meshes/pieces. I'll demonstrate this later.

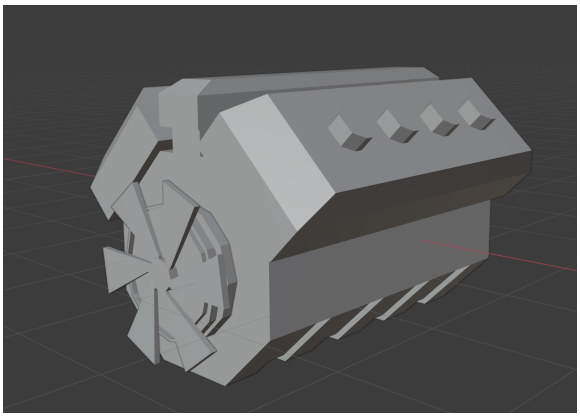


Wow! A cool Rocket! Who made this masterpiece?!?!

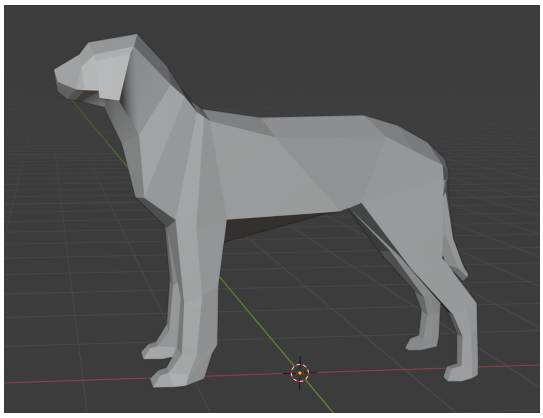
Here are some models that Pumpum and I made (untextured):



Witch's Broom

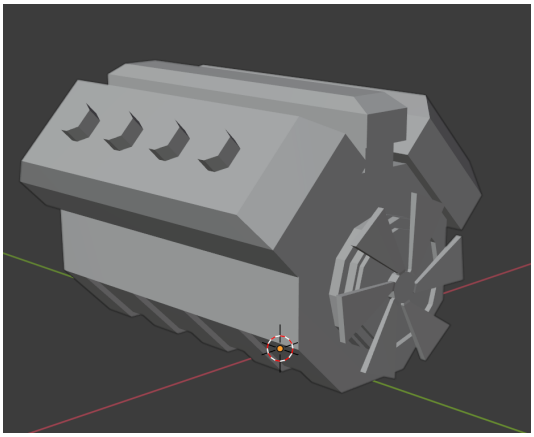


Car Engine

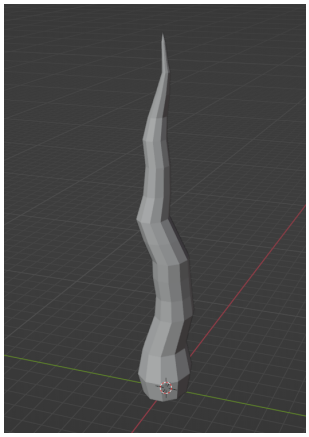


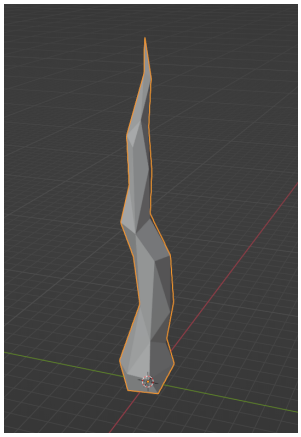
Dog

No idea
(Andrew made it)



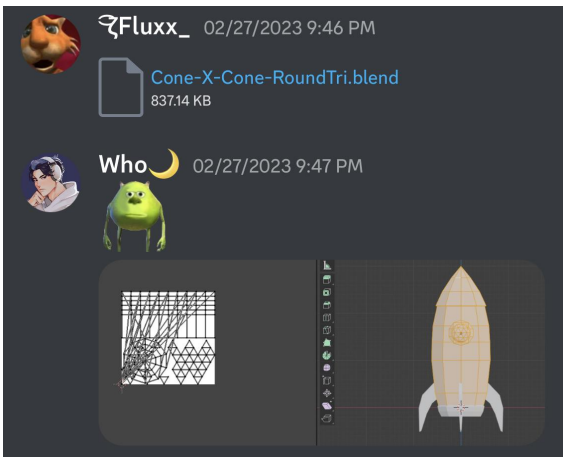
Car Engine Again

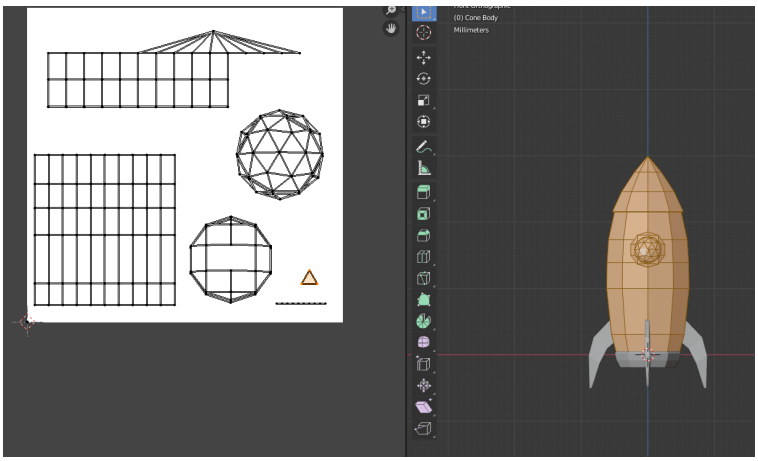




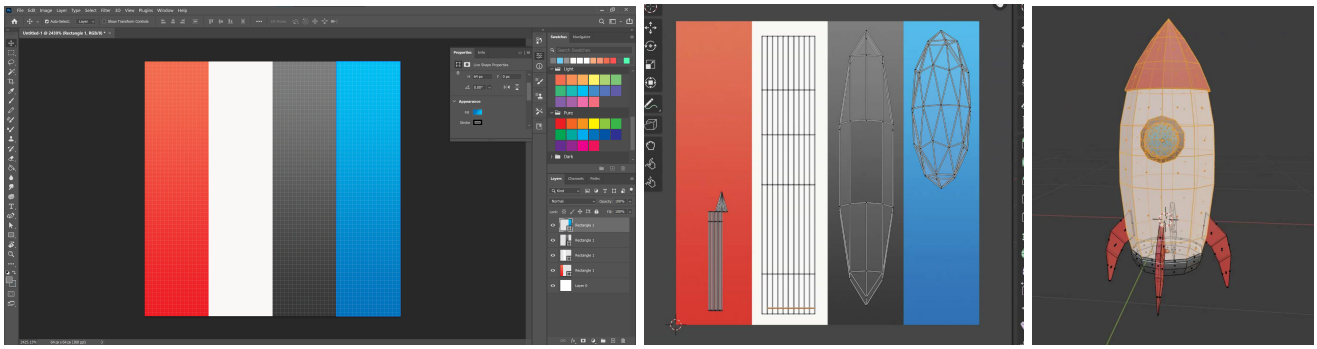
Geometry importance:
Base model vs. Decimated model

Now that we have a model, the general shape of the mesh/object is laid out. However, to add color to a 3D shape, we go through a process called UV Unwrapping. This is where you tell programs where and how you want to slice a 3D object and lay it out on a flat surface, coordinating every pixel of the flat UV to a point in 3D space. This is a whole other art itself and I hate doing it, mainly because I suck at it though. I tried adding seams so the program knows where to make cuts and all but the unwrapping still ended up horrible (denoted by the non-symmetrical and wonky shapes). In the end, I gave up and got Sam, a professional 3D modeler, to help me out:

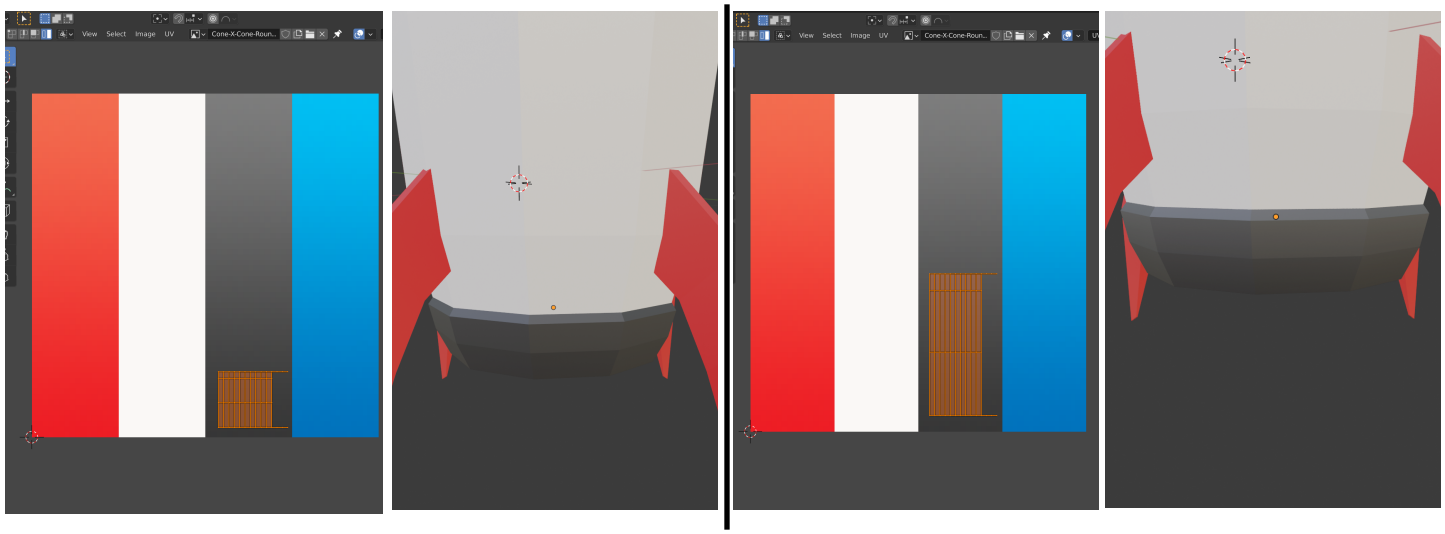




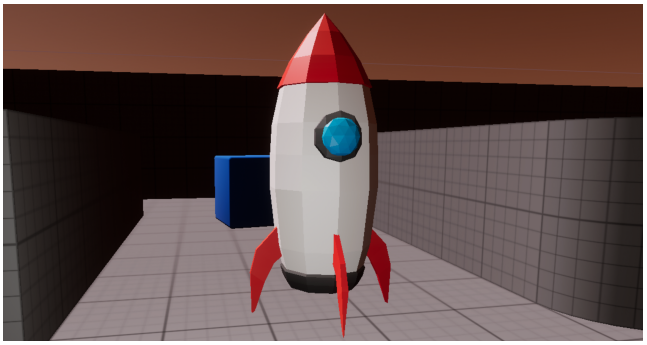
Putting my school issued photoshop license to good use I threw together texture using a technique called Gradient Texturing. It sounds fancy but it's just an amateur's way to quickly slap textures onto a simple model (like low poly meshes).



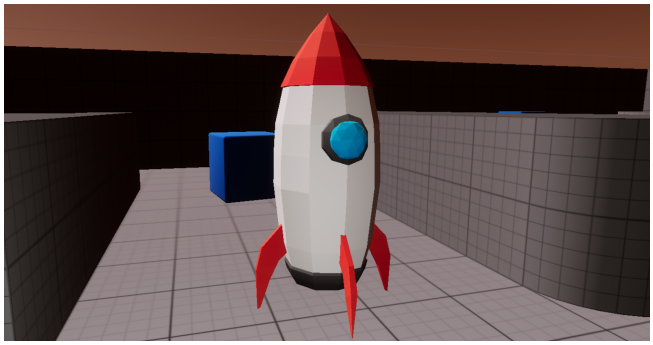
The advantage of this technique is that you'll get a nice subtle blend between seams, such as between this thruster and the rocket body. This effect is more prominent when blending diverse colors (like soil and grass).



Next was to get the model imported to Unreal, a task made easy from the prior set up I did with the *.obj* file settings. Once you drag and drop the file into the content browser of Unreal, you'll be prompted with some settings before everything is finally imported as multiple meshes, all separated for you.



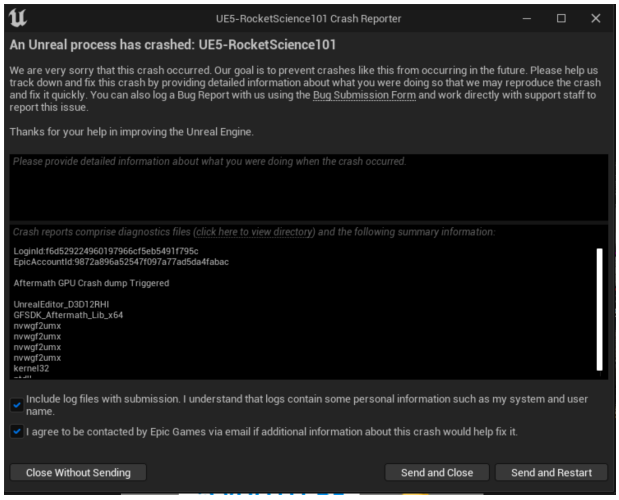
The meshes all have the right textures with the correct UV map, but it's still missing 3 crucial maps for texturing: Roughness, Normal, and Metallic. These define and give meshes the ability to interact with lighting, creating the illusion the texture is 3D (such as the shine at the tip of the rocket).



I won't be getting into how to generate/create these maps since they require a lot of time and skill, much more than needed for these simple rockets. Likewise with how they work. Instead, all that I need to do is create a constant value and plug it into the roughness map.

Looking at the textures raw, these are what they look like. These images use the original texture to demonstrate the roughness map because it just somehow worked out that half of the texture is shiny.

Oh, and of course, game development wouldn't be game development without some good ol' crashes and errors.



Converting to actors

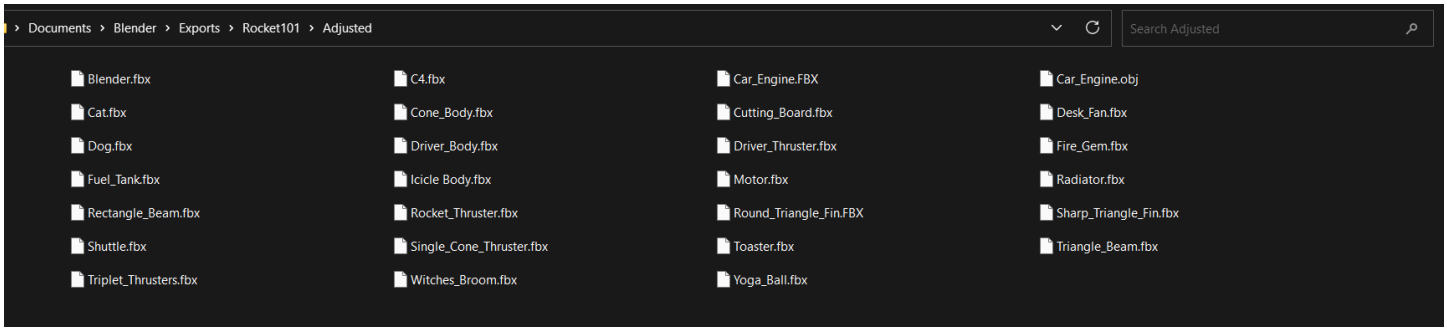
After importing the meshes to Unreal, these components are stand-alone "Static Meshes." In Unreal, we don't process meshes in a stand alone manner. The amount of things you can do with them just are too limiting. Instead, we need to convert the mesh to an object type called an actor. In Unreal, actors are how physical objects interact with each other and exist. However, with 20+ meshes, giving each of them their own actor is too repetitive and would take forever. This is why I opted for an overarching component blueprint class instead. Here's the initialization breakdown:

SetComponent Function - Gets called on creation of the RocketComponent blueprint by the RocketComponentsHandler to assign the correct mesh, potential, and all other information from the data table (has a "Data Row" parameter).

PreloadSockets Function - Only allows components of type "ERocketBody" to create open available sockets to snap components onto. Otherwise, we'd have overlapping sockets that overcomplicate things unnecessarily.

Mesh Reconfiguration

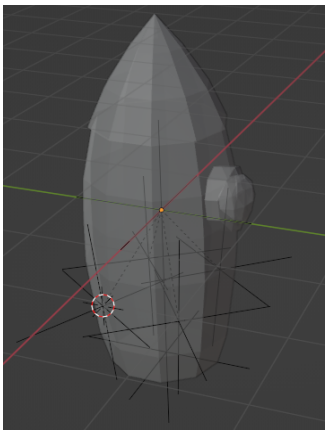
While working on sockets snapping together for the actor components, there were major flaws in the way I was setting the meshes up that I noticed. Despite my efforts in trying to work around them, the best solution I came to was to just reconfigure every mesh by hand. With Andrew and Pumpum not able to help me, I had to do this on my own, taking a painstakingly long 2 weeks.

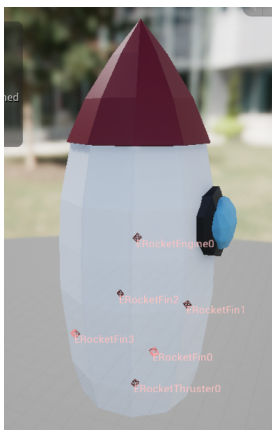


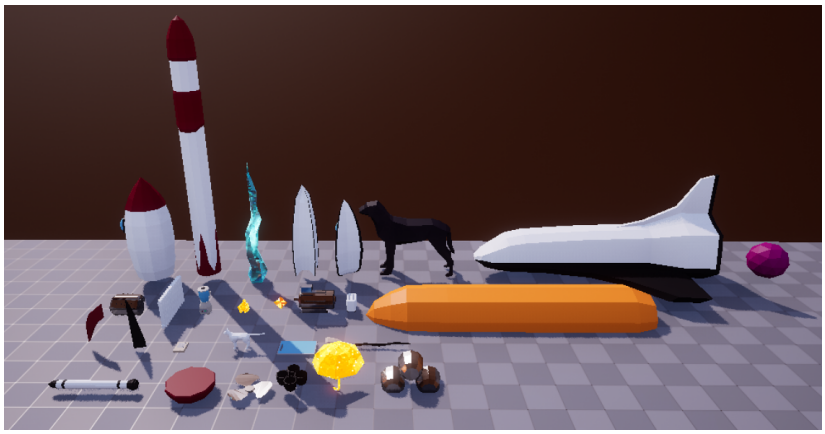
The first driving force of the reconfiguration is that all of the meshes needed to be converted from *.obj* to *.fbx*, meaning I had to import each mesh individually to blender and re-export it under the *.fbx* format.

The second driving force is that Unreal's socket editor isn't sufficient. The inaccuracy of the placement method doesn't allow for me to place sockets flush against the mesh. I also had no access to the individual vertices of the mesh, meaning I had to way of copying the location of what is guaranteed to be flush against the mesh. Instead, blender gives me direct access to the vertices and only requires a naming convention for Unreal to recognize that *Empty* types are sockets (Socket_NAME). Meshes also need to be in *.fbx* format to contain socket data when importing to Unreal.

The third and last reason for the reconfiguration is to clean up the meshes. Since these were made by two different people, me and Pumpum, the meshes' origins are in different places relative to the mesh. Ideally, they should always be at the center of mass, but some of our meshes had their origin set to the very bottom of the mesh for whatever reason. This causes issues when we try to snap the components together, since it is reliant on the mesh's location which is relative to the origin. While fixing this, I also took the liberty to finish texturing all of the meshes and scaling them properly (a yoga ball shouldn't be larger than a rocket). Here's everything completed





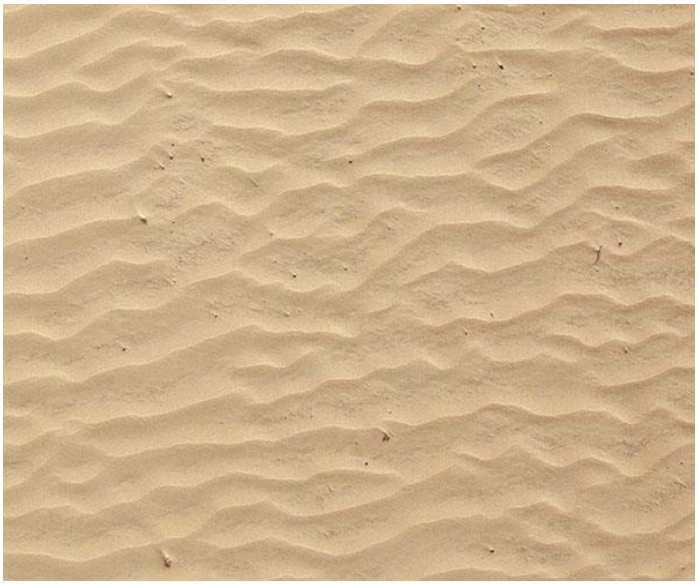


World Design

Developed by Andrew C. — Documented by Dang L.

In a fantasy novel, the author needs to build a convincing atmosphere for the readers to feel immersed in an existing world. Likewise, to sell the illusion that we're on another planet, I've tasked Andrew with creating terrain and atmosphere similar to that of Mars. His approach will be to procedurally generate the map so that nothing has to be done manually by hand. This approach also allows us the opportunity to create an infinite map should we have time.

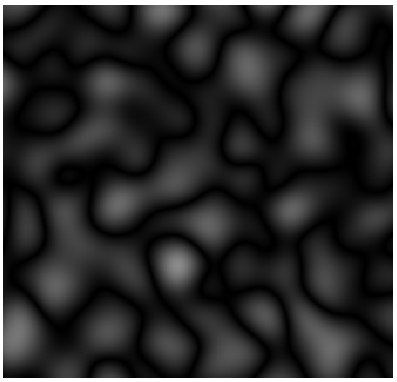
Starting with sand generation, it can be broken down into two main components (I'm no sand expert, these are my own names for them): the waves/ripples and the shape.



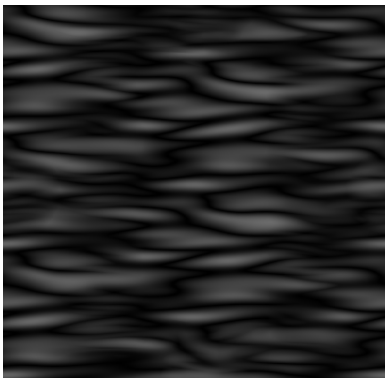
Rather than creating a mesh with millions of polygons and triangles to include folds and bends for the waves, we'll be using the same technique/method to texturize the rocket roughness – PBR textures.

This allows the rendering of the texture to interact with light, selling the illusion of a wavy plane. Using a form of perlin noise, a map called plasma noise can be made which has the sharp edges that fit our wavy feel better than just plain old perlin noise.

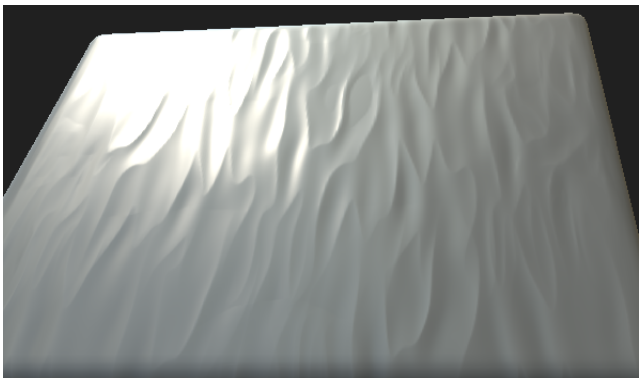
Lastly, Andrew threw in a Y scale to squish the noise map, making the texture more accurate to sand waves.



Soft Plasma Noise

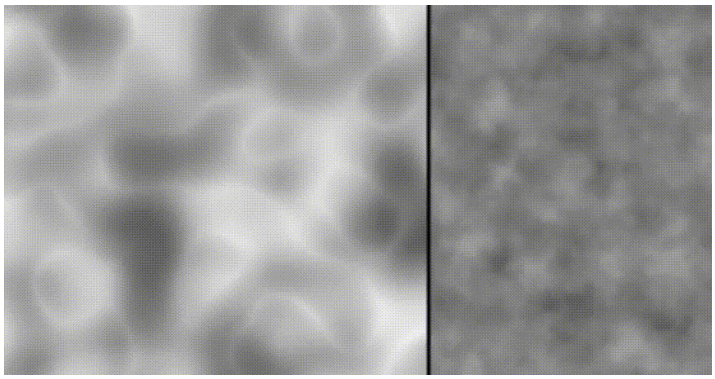


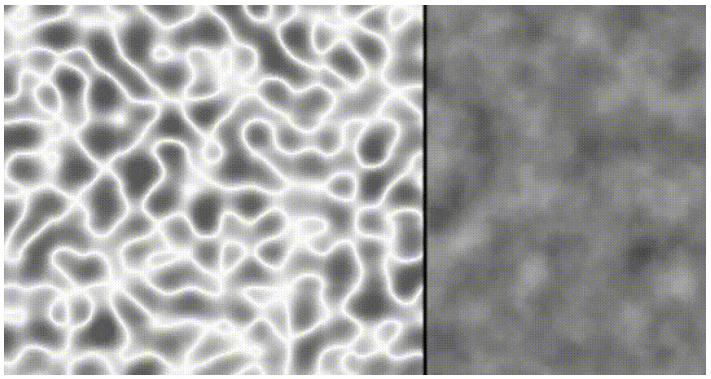
Squished Noise Map



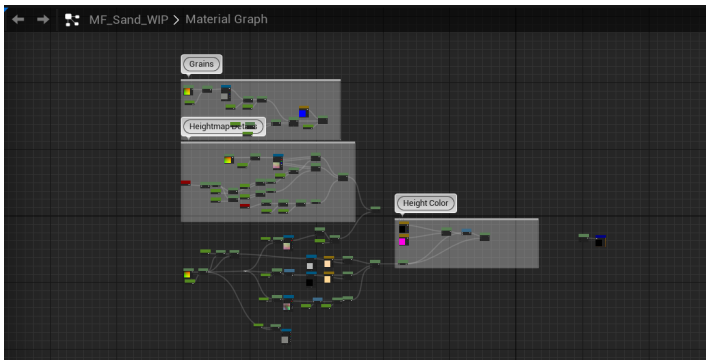
Noise Map applied to Normal Texture

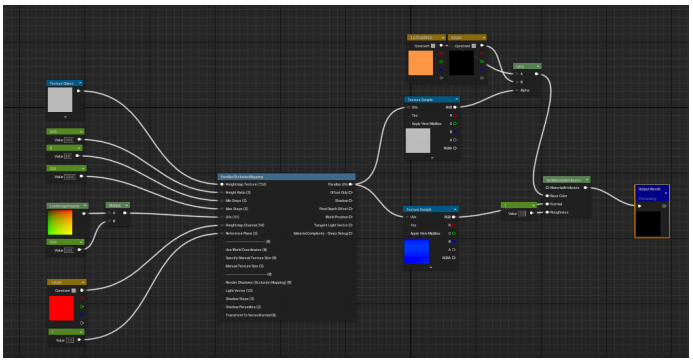
Remember, this plane is simply just a flat surface made of one single polygon. When we render the texture however, the light interacts with the surface creating darker and lighter areas to sell the illusion of physical waves. It does a good job doesn't it?



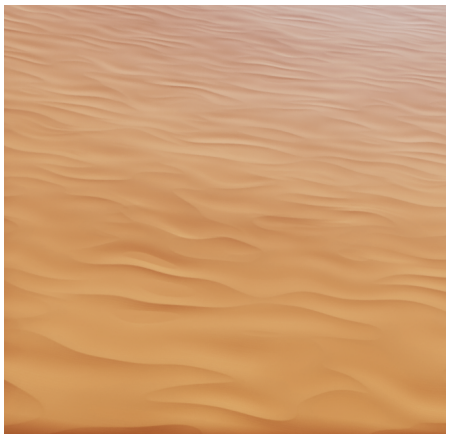


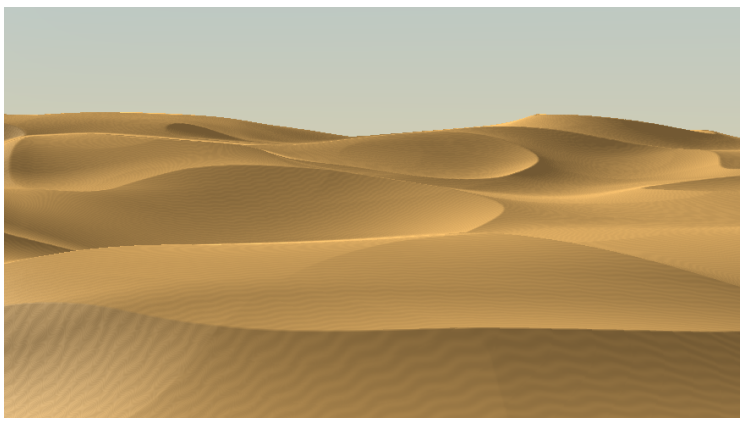
The left GIF is the final layering of all the maps needed to generate the terrain. You can see each part quite well, where the sharp white lines are the creases of the dunes and the dark spots are the divots and valleys.





Here, Andrew is connecting all of the nodes together to actually have the map generated by Unreal. The right image specifically is the color map linking to the terrain. End result:





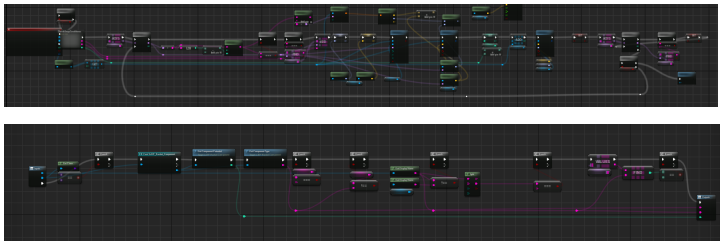
Rocket Launch

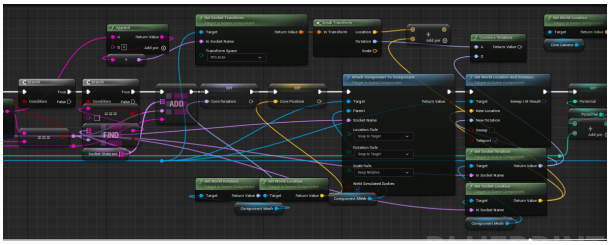
With the sockets finally out of the way and completed, it was time to have rocket component actors able to snap together and recognize the rocket as a “full rocket,” followed by actually lifting it off the ground.

Recognizing a Rocket & Launch

Extending off of the overarching RocketComponent blueprint, here is a breakdown of the snapping mechanism and how the rocket is able to determine how far it can go:

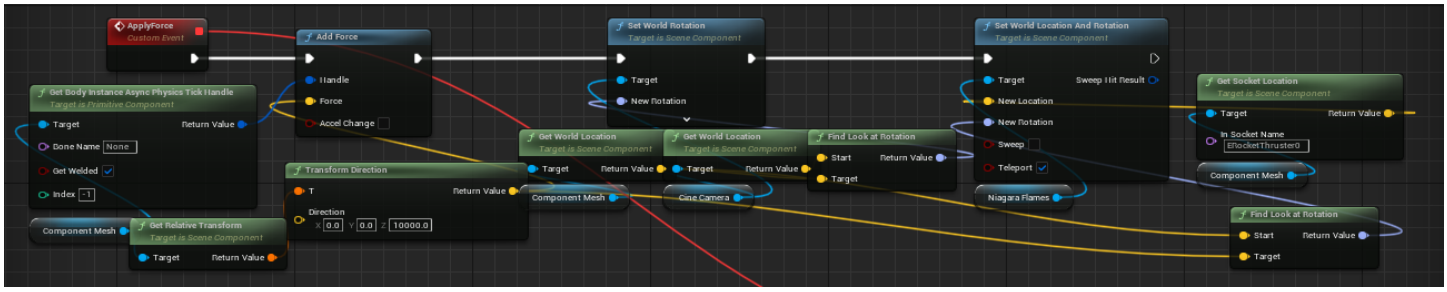
On Component Begin Overlap (Bounding Box) Event - Fires when other actors come in contact with the bounding box of the mesh (set through SetComponent). Has arguments for Colliding Actor and Colliding Component (Components are the non-blueprint things that make up an actor, like a static mesh). After a long series of checks for compatibility, the actor (assured to be ERocketBody) loops through all of its sockets to see which are available. Of the open sockets, if the colliding component matches in socket type, they snap together at the socket’s location and rotation.





RocketFinished Event - Custom event fired when the rocket no longer has an open slot (checked at the end of On Component Begin Overlap). Switches cameras out of the first person controller and disables player input. Calls corresponding stage check functions to check for the correct amount of potential and decide next actions. If the rocket passes, it fires the event *ApplyForce* every 0.0001 seconds until the next check is done, in which it is paused to await for further actions.

Apply Force Event - Applies a force in the -Z direction relative to the rocket (to ignore rotation and always point “down” relative to the rocket). Also points the camera at the rocket and updates the flame particle’s position to the bottom of the rocket



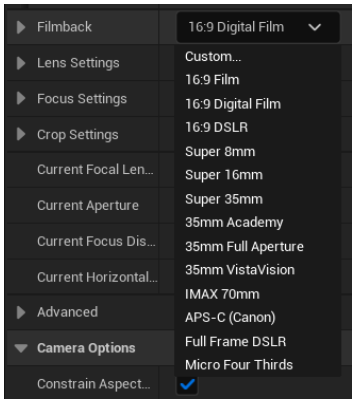
CheckStage(1, 2, 3) Function - Cleans up any audio effects lingering and checks if the rocket has enough flight potential to pass. If it does, it’ll play the corresponding audio. Otherwise, it calls the *Explode* method.

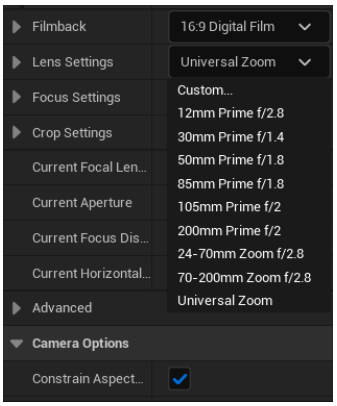
Explode Function - Plays the explosion audio and enables the corresponding particles. Disables physics, makes all attached sockets invisible, and cleans up all playing audio.

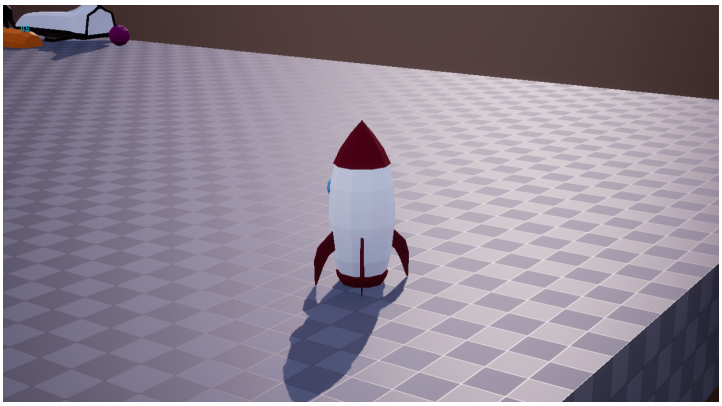
Cutscenes

Using a combination of different cameras, particle effects, and audio effects, I achieved pretty cool cinematic feeling cutscenes. The only thing I'd add if I had time was camera shake to emphasize the rumble of the rocket and when it passes the camera.

Cine Camera - Unreal was not joking when they said they'd go for realism. With the default cinematic camera alone, they had over 10 presets to emulate just lenses alone for the photography and videography industry. Obviously, I'm not a professional videographer so I just picked out a lens that looked nice and had a cool aspect ratio. This ultimately led to a nice transition from the first person camera to the cine camera due to the conflicting aspect ratios, which Unreal compensated with black bars on the side (giving a cinematic vibe).

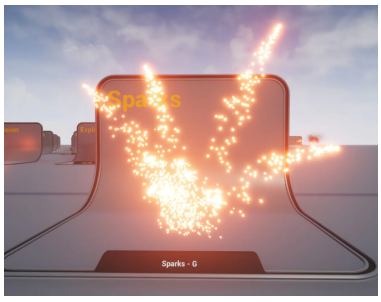


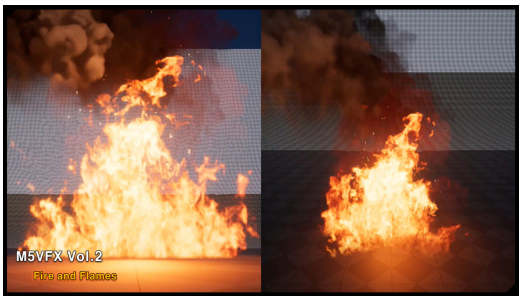


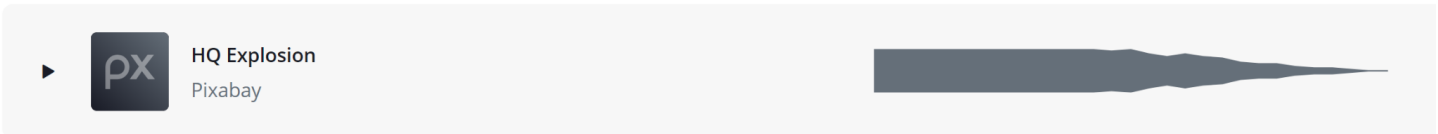


To give a more "Helicopter chase" feel rather than a realistic scene out of a movie, I placed the camera a far distance from the rocket, zoomed it in to cut down on its field of view, then had the camera follow the center of the rocket as it flew. God, I love the freedom and beauty of game development.

Audio and Visual Effects - Both were sourced from non-copyrighted material (one from an online sound bank and the other from Unreal marketplace). The particle was a bit more annoying to set up, as it was difficult to align the particle with the -Z axis of the rocket. However, once it worked, it worked beautifully.

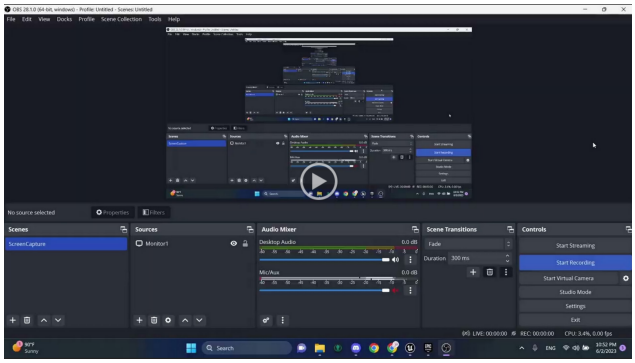
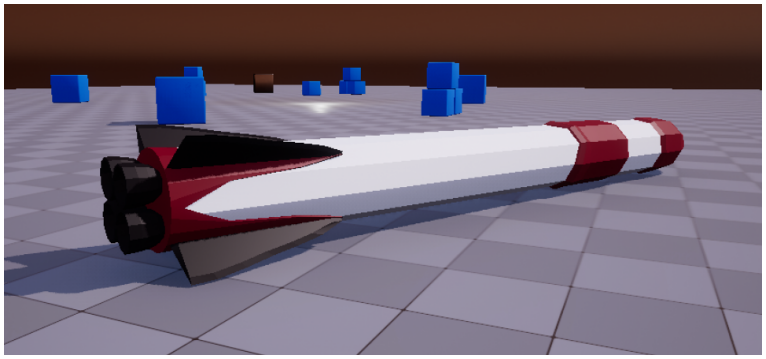
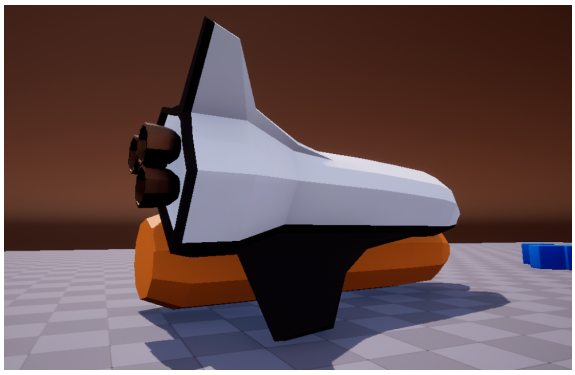
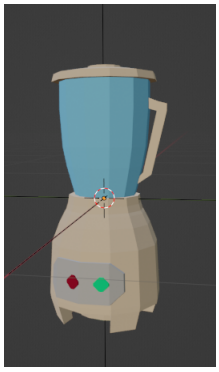
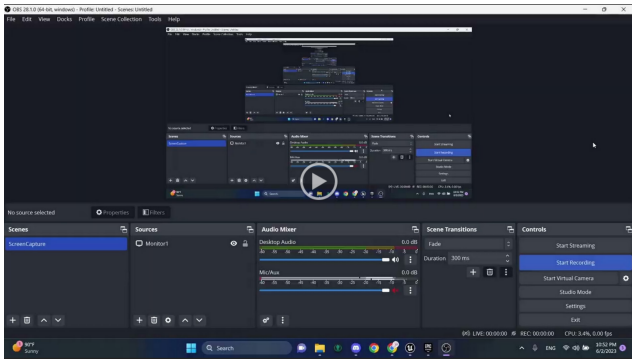
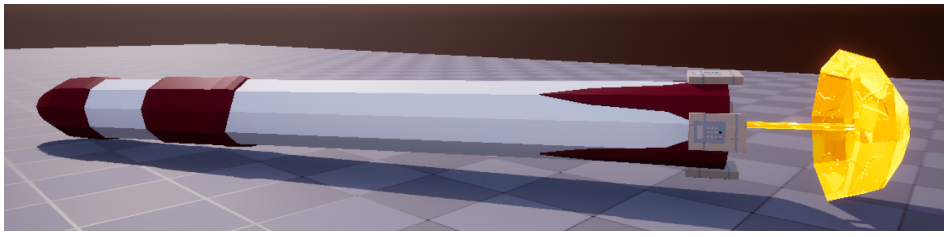


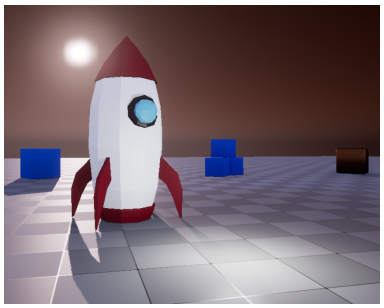
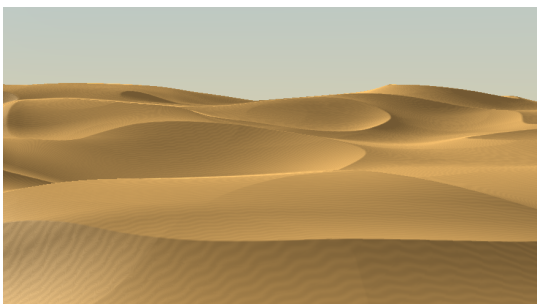
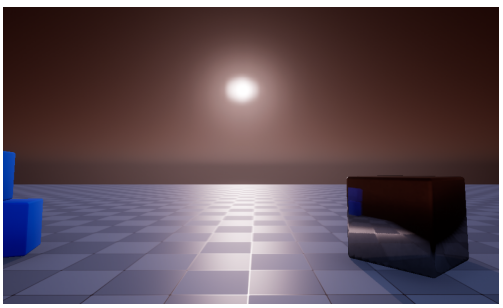
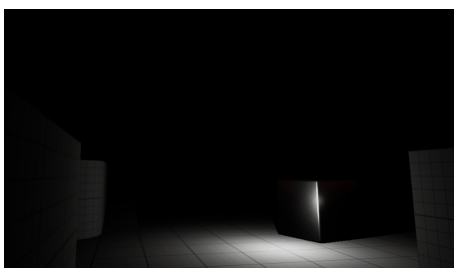
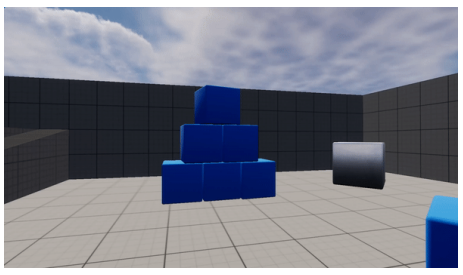
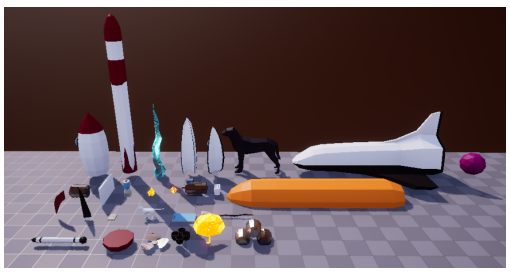




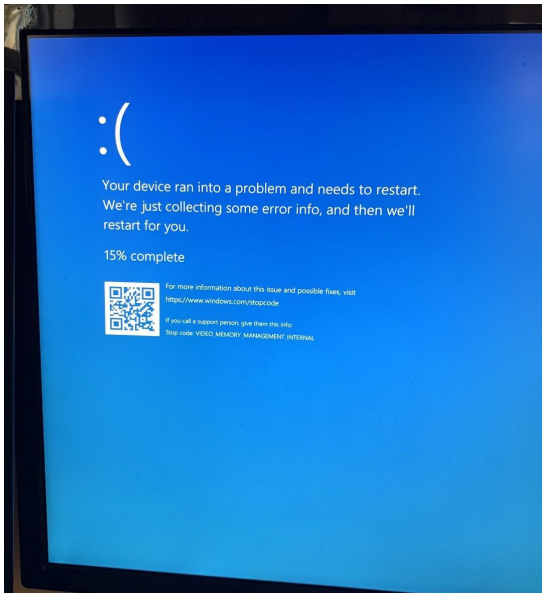
Finished Product

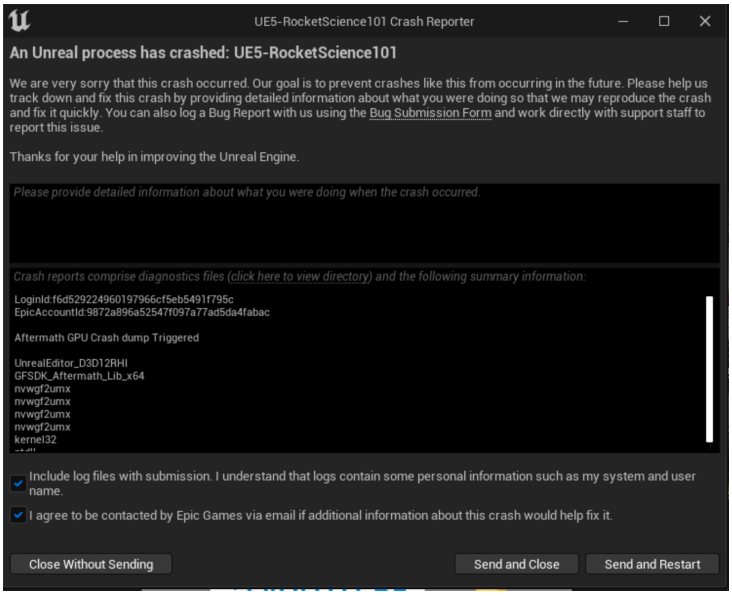
Thank you for an awesome year with Unreal! I'll continue to work on the game over the summer with goals of releasing it on Steam if I can salvage it from its messy state and improve performance. No promises though :3

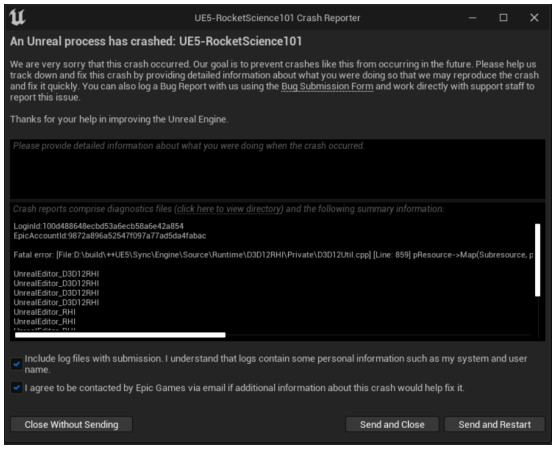







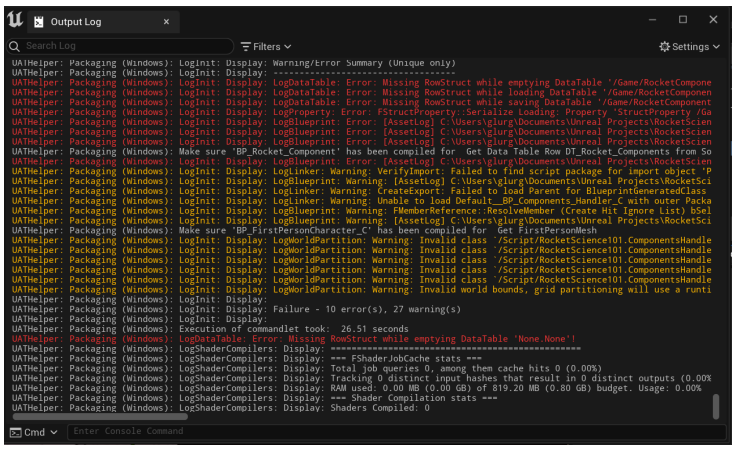








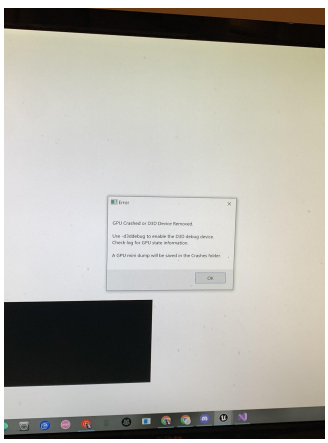
Collection of Crashes

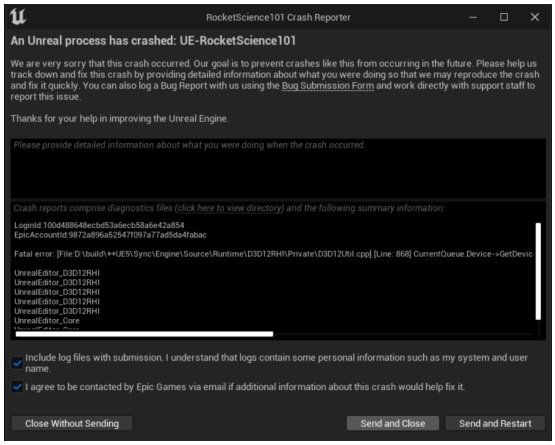


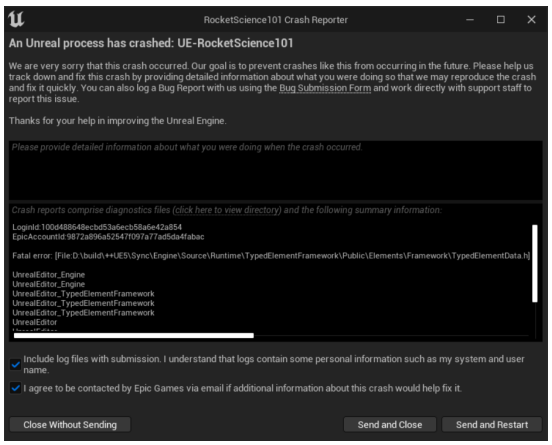












I also found that Unreal likes to crash when you leave it idling in the background for too long, for whatever reason. It's best to just save whenever possible 😭.

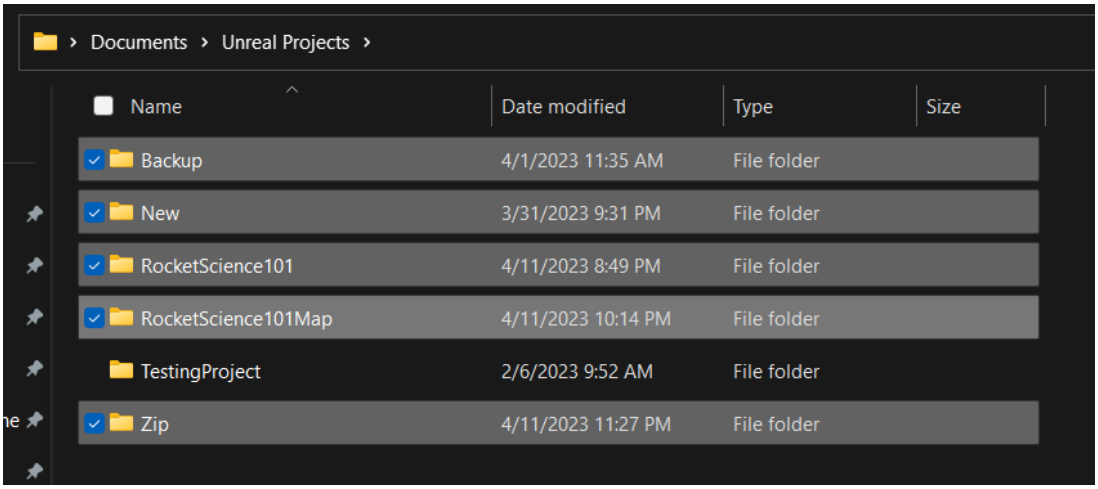
Funny thing is, Unreal crashed in the background while I wrote that and some other stuff ^

Project Transfer

I couldn't really find an appropriate place to mention this while documenting my development process since it didn't fit into any of the categories or titles, so I'll bring it up here. Before starting the RocketComponent system, I decided on making the DataTable first. This was a very, very bad idea. With the C++ struct compiled and imported into Unreal's DataTable object, I started the RocketComponent blueprint and everything seemed to work fine, until I realized I needed to make changes to the DataTable.

Normally, a simple rewriting of the C++ struct, recompilation, and editing of the actual DataTable object should fix the issue. However, from then on, I've had the same recurring issue no matter what fixes I had attempted. Every time I boot up Unreal Engine, the compiler defaults back to the original struct that was first compiled. This means the DataTable was always broken and couldn't be used, essentially corrupt. It didn't matter if I deleted the DataTable object and made a new one or created a new struct and rebuilt the DataTable, the compiler was just messed up.

Ultimately, Andrew proposed that I just did a full project transfer. Since Unreal has the option to "Migrate" UE Assets (Unreal packages, transfers, and unpackages the assets for you), I had to create a new project and transfer all files over, except for the DataTable and C++ class. This also meant that EVERYTHING needed to be recompiled, taking my poor laptop well over 4 hours. To make matters worse, I had to do this project transfer three more times since I still had issues with the compiler, again. With where I'm at now, I have over five copies of the game at various stages in development. I also desynced my github in the process, which I'll have to reconstruct later on. For now, I've just been working on my laptop.



Sources

Unreal Engine & Alternatives

- https://www.unrealengine.com/en-US/features
- https://www.unrealengine.com/en-US/unreal-engine-5
- https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Physics/ChaosPhysics/Overview/
- https://www.unrealengine.com/en-US/faq
- https://forums.unrealengine.com/t/unreal-5-memory-usage-is-different-than-what-windows-is-showing-which-is-correct/268152
- https://docs.unrealengine.com/5.0/en-US/key-concepts-in-niagara-effects-for-unreal-engine/

- https://store.unity.com/compare-plans
- https://docs.unity3d.com/Manual/PhysicsSection.html
- https://docs.unity3d.com/Manual/PackagesList.html
- https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@12.1/manual/master-stack-hair.html
- https://docs.unity3d.com/Manual/class-Cloth.html
- https://docs.unity3d.com/Manual/LightingOverview.html
- https://unity.com/features/shader-graph
- https://docs.unity3d.com/Manual/PartSysUsage.html
- https://docs.unity3d.com/Manual/render-pipelines-overview.html
- https://docs.unity3d.com/Packages/com.unity.entities@0.51/manual/index.html
- https://docs.unity3d.com/Packages/com.havok.physics@0.1/manual/index.html

- https://www.gamedesigning.org/career/video-game-engines/
- https://www.youtube.com/watch?v=aExdxF4OKBo
- https://program-ace.com/blog/unity-vs-unreal/
- https://gamedevacademy.org/unity-vs-unreal/

- https://youtu.be/8xJRr6Yr_LU
- https://www.youtube.com/watch?v=5J0CczTshKY

- https://forums.unrealengine.com/t/why-am-i-getting-an-infinite-loop/662662
- https://www.google.com/url?q=https://docs.unrealengine.com/5.1/en-US/using-sockets-with-static-meshes-in-unreal-engine/&sa=D&source=docs&ust=1685893857428100&usg=AOvVaw2ClcSJusE7tnXlOGwKBZ5M
- https://www.google.com/url?q=https://docs.unrealengine.com/5.0/en-US/BlueprintAPI/Transformation/AttachActorToActor/&sa=D&source=docs&ust=1685893857428218&usg=AOvVaw1JfLelPvulllg7QVG5yuKe