BACHELOR OF ENGINEERING (HONOURS) IN ENGINEERING SOFTWARE

ACADEMIC YEAR 2018/2019

The design and development of a VR Application for teaching electronic fundamentals to 1st Year Engineering Students

Author: Albin Jacob

Supervisors: Andrew Donnellan

Date: 18th May 2022

Declaration

"This report is prepared as partial fulfilment towards graduation requirements for Bachelor of Engineering (Honours) in Electronic Engineering, at the Technological University Dublin. I declare that the contents of this report, and the project to which it refers, are entirely the work of the author and have not been submitted for a degree in any other institute."

Signed……Albin Facob …….

Date…….18/05/2022……..……

Abstract

Virtual reality (VR) is a cutting-edge technological advancement that allows users to interact with computer-generated surroundings. It's the key to seeing, hearing, and feeling the past, present, and future. They can look at it from various angles, reach into it, grab it, and reshape it. In the last decade, VR has provided numerous alternative learning opportunities, through modern products such as the Oculus Rift and other wearable Virtual Reality technologies that are being introduced into society. Virtual equipment makes use of a user's spatial information by outfitting them with physical counterparts of common wearables such as headsets, gloves, headphones, and so on. Currently, there is little systematic research on how immersive VR has been used in higher education that considers the use of such equipment.

Epic Games' game engine Unreal Engine, this project investigates this dilemma and creates an adventure game environment with scenarios covering the fundamentals of 3 logic gates that involve the physical use of HMDs and other VR equipment for 1st Year Engineering Students. The VR framework developed with Unreal Engine offers an extensive, unified framework for developing virtual reality apps with the engine. This project constructed a tailored reality game type experience that incorporates electronic engineering fundamentals and allowed the students to learn and apply these fundamentals through problem-solving scenarios using VR equipment & reality.

Acknowledgements

I would like to express my gratitude & appreciation to all those who gave me the possibility to complete this report. I would like to express my deep sense of gratitude & indebtedness due to my supervisor Dr Andrew Donnellan for his invaluable encouragement, stimulating suggestions and support from an early stage of this research & helped me all the time during the designing process & in writing this report. I also sincerely thank him for the time spent proofreading & correcting my many mistakes.

I wish to acknowledge the Technological University Dublin, and the faculty members without whom this project would have been a distant reality. I am immensely obliged to my friends for their elevating inspiration & encouraging support in the completion of my project.

Last, but not least, my parents are also an important inspiration for me. So, with due regard, I express my gratitude to them.

Table of Contents

Table of Figures

1. Introduction

The goal of this project is to develop a virtual reality learning environment (VRLE) for first-year electronic students to learn how logic gates work. Recent advancements in VR technology enable this new potential learning technique with numerous educational applications. Since commercial VR headsets are widely used for entertainment purposes, many people's first experiences with VR came from video games and other widely distributed media, which are widely advertised and well known, resulting in increased popularity. However, thanks to significant advances in technology, including this technology is now available in a mobile format, VR now has broader application possibilities. This mobile format enables universities and other third-level education institutes to deploy these VR learning environments at a low cost. So, by utilizing this advancement, this project aims to create a learning environment that can be used by both commercial VR headsets such as the Vive or Oculus, as well as mobile format VR such as Google Cardboard or Gear VR. This allows students to go home and immerse themselves in the VR experience, fully understanding the concept of what is being tried to be taught usually through the use of theory.

Logic gates are the core elements of any modern device. It is a single-output electronic circuit with one or more inputs. The relationship between the input and the output is ruled by logic. Based on this, logic gates are labelled as AND gates, OR gates, NOT gates, and so on. The problem that this project is attempting to solve is to take the theory side of logic gates and create an immersive problem-solving environment that allows the student to understand why a certain type of gate is used in a specific scenario. Since VR can present environments in 3D, is interactive, and can provide audio, visual, and even haptic feedback it is very practical in presenting learning materials in 3D and can be especially beneficial when teaching subjects where the learning materials must be visualized. So, this project can demonstrate to students how each gate has its own properties, what happens if the gate is switched with another and how the output differs, or how that gate will break the system and cause it to malfunction. Though visualizing is one of the most obvious advantages of VR, it can also be accomplished with a simple video. (McMahan, 2003) talked about the connection between immersion, presence, and engagement. They investigated what it means for a learner to be immersed, as well as immersion & engagement in 3D virtual environments, to demonstrate how 3D virtual environments can be used to improve learner engagement. Videos are passive

learning objects, while VR allows for direct interaction with the environment. Interactivity and feedback can be beneficial for all subjects, as there are specific advantages to interactive learning because it promotes active learning rather than passive learning.

The utility of VR in education may also be determined by the type of learning. According to learning styles theories, there are various ways to learn, and some people learn better with some methods than others because they have different approaches to information processing. Learning styles are classified into three types: visual, auditory, and kinaesthetic. Because VR headsets support complex visual renderings, audio, and movement tracking, all three of these learning styles can be targeted in a single application. Having a single learning environment that can accommodate multiple learning styles could be very beneficial because it would be suitable for a much wider range of individuals, particularly the younger generation of students who are already involved in these environments from a young age via consoles and such.

To make this learning experience unique while remaining relatable to the audience, the scenarios are designed in the style of an adventure game. As per (Frank, 2012) while educational leaders have initially been cautious to use gameplay or virtual worlds in the classroom, there is a growing interest across broad and diverse parts of the educational establishment to investigate the use of digital games as serious learning and assessment tools. In this game, the player will be immersed in an ancient temple ruins environment, where they must explore and find switches that can control the state of input 1 or 2 of a specific logic gate, which will solve a specific problem, such as opening a door. Advocates of game-based learning in higher education point to digital games' ability to teach and reinforce skills necessary for future jobs such as collaboration, problem-solving, and communication.
This adventure-style game has been designed and developed using Epic Games' Unreal engine. The game is created in an engine using blueprints and node-based scripting via the provided VR framework and then tested using HMDs such as the oculus.

This report discusses the history of game development engines and virtual reality. Then, summarize the exploration of education through virtual reality, as well as evaluate the use of games for learning and assessment. Finally, it will present the design of the game environment as well as the implementation and development agenda to move the field forward.

2. Background

2.1 Game Development Engines

A gaming engine is a software development environment that includes settings and configurations that optimize and simplify the development of video games in a variety of programming languages. Game engines are primarily used by developers to create games for video game consoles and other types of computers. As described by (ARM, n.d.) it is also known as a "game architecture" or "game framework." The core functionality of most gaming engines includes;

1. A 2D or 3D graphics rendering engine that is compatible with various import formats.
2. A physics engine that simulates real-world activities,
3. Artificial intelligence (AI) that responds to the player's actions automatically
4. A sound engine that controls sound effects
5. An animation engine

Collision detection, scripting, networking, streaming, memory management, threading, localization support, scene graph, and video support for cinematics are also features of advanced engines. This project prompts the use of a gaming engine to create the virtual reality map and the environment from which the user can explore and learn. Unreal Engine, Unity, Lumberyard, and Cry Engine are some of the most popular & widely used game engines in the current market. The following chapters discuss Unreal Engine and Unity in detail, comparing the major differences and complexities between them as well as explaining why a certain engine was chosen to create the virtual reality learning environment in this project.

2.1.1 Unreal Engine

Unreal Engine (UE), which is owned by Epic Games. According to (Unreal Engine, n.d.) it is essentially a game development multi-platform engine designed for businesses of all sizes that helps to transform ideas into engaging visual content using real-time technology. The original version was released in 1998, and it is still used in some of the most popular games 19 years later. The UE's strength is its ability to be customized to the point where games can be transformed into truly unique experiences. It can be used for enterprise applications and cinematic experiences to high-quality games across PC, Console, Mobile, Virtual and Augmented Reality. UE 4 is the latest version of the platform it is designed for a wide range

of platforms such as Windows, macOS, Linux, SteamOS, HTML5, iOS, Android, Nintendo Switch, PlayStation 4, Xbox One, Magic Leap One, and Virtual reality (SteamVR/HTC Vive, Oculus Rift, PlayStation VR, Google Daydream, OSVR and Samsung Gear VR) corresponding to (Unreal Engine, n.d.).

This engine is known for the development of high-tech Triple-A games as it is best for projects with a long-life cycle. These options give UE a greater chance when compared to the other engines listed above, but this engine also has some drawbacks, such as a steep learning curve to master to create a highly professional game. Unreal Engine also prefers that the user works within their structure of doing things. If you're working on a large-scale budget game, you'll need a licensed copy of Unreal Engine to make a game for the public, and you'll have to pay a 5% tax once the game is published and profitable.

2.1.2 Unity

Unity is a multi-platform game engine that simplifies the creation of interactive 3D content. Unity, a gaming engine developed by Unity Technologies in 2005, is also a favourite choice of many large organizations today due to its excellent functionality, high-quality content, and ability to be used for any type of game. It can also illustrate both 2D and 3D content. Unity's all-in-one editor is compatible with Windows, Mac, Linux, iOS, Android, Switch, Xbox One, PlayStation 4, Tizen, and other platforms. It is especially popular for iOS and Android mobile game development, and it has been used in games like Pokémon Go and Call of Duty: Mobile, among others. The user-friendly interface streamlines development and reduces the need for training. As a result, it is popular among indie game developers and is thought to be simple to use for new developers. The Unity Asset Store is a massive repository of tools and content that is constantly updated.
Other than video games, the engine has been used in industries such as film, automotive, architecture, engineering, and construction, as well as in the United States Armed Forces. Developers have realized that game engines can be used successfully for non-game applications. As discussed by (Haas, 2014) architects, for example, can quickly prototype ideas, artists can create interactive art installations, and researchers can use them to visualize data. Another reason for Unity's popularity is its ability to deploy to a wide range of target platforms while using the same code and assets. It only takes two mouse clicks to get your game running on another platform. One will switch platforms, while the other will build and

run. Unity also has three browser-based options: the native unity web player, Google's native client, and Flash Player, which is now discontinued.

2.1.3 Differences between UE & Unity

The first significant distinction between Unity and Unreal Engine is their programming languages. C# is used by Unity in both the editor and the additional plugins. Unreal Engine is written in C++, but when creating game code, you'll use a combination of Blueprint a proprietary language exclusive to Epic Games and C++. UE's visual scripting option called Blueprints, allows you to manage the expansion process and lessen the amount of time spent on it, because of the various functions, rendering technology is more efficient and faster than in other engines. (Scripting, n.d.). According to the documentation provided by UE (Unreal Engine, n.d.) The Blueprint Scripting system in UE is a complete gameplay scripting system based on the concept of creating gameplay elements from within Unreal Editor using a node-based interface. It is used to define object-oriented (OO) classes or objects in the engine, just like many other common scripting languages.

This leads to the second significant difference, the ease of use, as determined by a test conducted by (Smid, 2017). Users found Unity to be slightly easier to use, due mainly to its native C# coding language, which should be relatively familiar to all developers, and its overall workspace layout. It's a simpler platform to "dive in" and start creating on than Unreal Engine, which has a slightly steeper learning curve. The Unreal engine will be functional by artists who have prior programming experience. This knowledge is highly valued and well-suited, but for novice users or freelancers attempting to build a prototype to show to investors and raise funding, unity is preferred.

While prototyping, the best VFX, Animation & Rendering are required to showcase to clients and investors, which leads to the third difference comparison. While both platforms generate higher visual effects, the majority of users in the test operated by (Smid, 2017) discovered that Unreal Engine has a significant advantage over Unity in terms of visual effects quality. Unreal Engine provides better characteristics for the different types of graphics & environments in multiple amounts of scenarios. It can generate photorealistic visualizations that immerse players and allow them to freely travel through a beautiful new world, as well as incorporate high-quality assets from a variety of sources. As described by (Eldad, 2021) unreal also provides a marketplace full of assets that can be used for prototyping, as well as

additional plugins and community support forms which are rapidly growing. Unreal Engine has more tools and functionality for a wide range of situations that are not present in other engines of a similar type. Unreal's animation quality is unrivalled, thanks to its powerful rendering capabilities and top-tier visual effects. It was the clear user favourite due to the high quality of its animation tools and animation renderings.

In terms of pricing, according to Unity Technologies, there is a free version of the software available, but to unlock all its functionality, the user needs to upgrade to the pro version, which has a monthly subscription fee of €75 per month. Unreal Engine, on the other hand, has a distinct pricing model, according to Epic Games. Although the software is free, Unreal Engine is entitled to a 5% royalty fee on all game sales once your game is released. (Unreal Engine, n.d.)

While Unity is an excellent platform for indie designers who want to start creating right away and don't want to owe the platform a royalty off their games on the back end, thanks to its native C# language and a large community of other developers and designers. Unreal Engine is all about fine-tuned graphics and lightning-fast render speeds, making it ideal for enterprise-level game developers or indie developers who want that extra-fine quality on their games and don't mind owing to the royalty on the back end. The learning curve is steeper but learning to play by Unreal Engine's guidelines benefits guests with mind-blowing graphical prowess that appears to be limitless. All of these factors contribute to the decision to use the Unreal engine for this project. It is ideal for creating a VRLE due to its fine-tuned graphics and render speeds.

2.2 VR

The use of computer technology to create a simulated environment is known as virtual reality (VR). VR immerses the user in an experience. Instead of looking at a screen, users are immersed in and able to interact with 3D worlds. As described by (Bardi, 2019), the computer is transformed into a gatekeeper to this artificial world by simulating as many senses as possible, including vision, hearing, touch, and even smell and the only limitations to near-real VR experiences, are the availability of content and cheap computing power.

Many researchers believe that virtual reality is made up of blocks such as synthetic experience, virtual worlds, artificial worlds, and artificial reality. The following are some of the researchers' definitions of Virtual Reality;

- "Virtual reality refers to immersive, interactive, multi-sensory, viewer-centred, three-dimensional computer-generated environments and the combination of technologies required to build these environments." (Cruz-Neira, 1992)
- "Real-time interactive graphics with three-dimensional models, combined with a display technology that gives the user the immersion in the model world and direct manipulation." (Fuchs & Bishop, 1992)
- "The illusion of participation in a synthetic environment rather than external observation of such an environment. VR relies on three-dimensional, stereoscopic head-tracker displays, hand/body tracking and binaural sound. VR is an immersive, multi-sensory experience." (Gigante, 1993)
- "Virtual reality lets you navigate and view a world of three dimensions in real-time, with six degrees of freedom. In essence, virtual reality is a clone of physical reality." (Scheweber, 1995)

Although these definitions differ, they all essentially refer to the same thing. They all imply that VR is an interactive and immersive experience in a simulated world. Figure 2.2.1 depicts the relationship between autonomy, interaction, and presence in virtual reality. It's also known as Zeltzer's cube. Autonomy refers to a

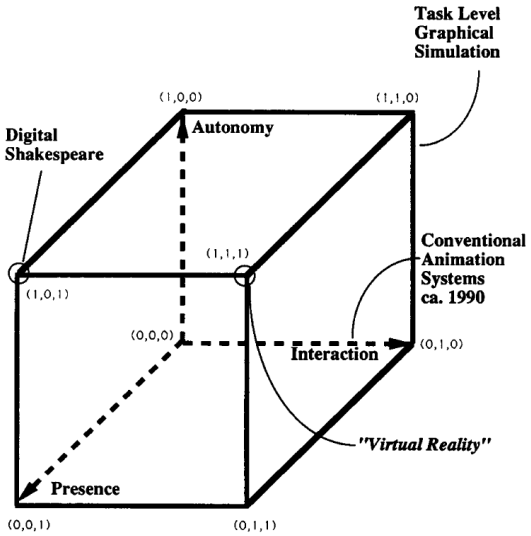


Figure 2.2.1 - Zeltzer's Cube

system's ability to receive and respond to external stimuli, such as user actions. Interaction describes the extent to which a user uses all of the commands available in the system. The presence of a system describes how much tactile feedback it provides. Each variable is assigned a value between 0 and 1 and plotted on a three-axis coordinate system. The point at 0,0,0 represents no virtual reality, while the point at 1,1,1 represents the pinnacle of virtual reality.

2.2.1 Applications of VR

A computer generates sensory impressions that are delivered to the human senses in a virtual environment system. The type and quality of these impressions determine the level of immersion and presence in VR. Cab simulation, projected reality, augmented reality, telepresence, and desktop virtual reality are all examples of virtual reality. Cab simulation is used in training, such as for aeroplane pilots. Projected reality employs projection technology to create a system that matches the resolution, colour, and flicker-free of workstation screens. Augmented reality uses head-mounted displays (HMDs) to superimpose virtual 3-D objects in the real world. Along with the computer-generated graphics, the outside world is visible. Manipulate objects in the virtual world, and the results appear in a remote location in the real world, using telepresence technology. Traditional virtual reality systems are subsets of desktop virtual reality systems. It employs traditional computer input and output devices such as a keyboard, mouse, and monitor rather than a head-mounted display, but spatial awareness is lacking. (Zheng, et al., 1998)

Ideally, high-resolution, high-quality, and consistent information should be presented to all the user's senses across all displays. The user can observe and manipulate the simulated environment in the same way that we do in real life, without having to learn how the complex user interface works. This enables users to use VR in a variety of industries to simulate an immersive understanding such as;

- Data and Architectural Visualization
- Modelling, designing, and planning
- Training and education
- Cooperative working
- Entertainment

2.2.2 VR Technology

VR requires more resources than standard desktop systems to provide a fully immersive experience. For improved user interaction, additional input, and output hardware devices, as well as special drivers, are required. Electronic and mechanical engineering, cybernetics, database design, real-time and distributed systems, simulation, computer graphics, human engineering, stereoscope, human anatomy, and even artificial life are all part of VR. The primary goal of virtual reality is to immerse the participant in a virtual environment that gives the participant the sensation of "being there." This necessitates the integration of the human sensory and muscular systems with the "virtual environment." A VR system is made up of three different types of hardware: Sensors, effectors, and reality simulators are the first three main components. The sensors include head position sensors that detect the operator's body movement and data glove sensors that measure finger bend and flex, capture the user's actions, and instruct the computer on how to respond to the inputs. HMDs and other effectors stimulate the operator's senses. The reality simulator connects the sensors and effectors to create sensory experiences for sight, hearing, and touch in the user. Because VR displays environments in 3D, it is interactive, and it can provide audio, visual, and even haptic feedback. HMDs provide visual and audio input.

Display, latency, refresh rate, and tracking, if required, are all factors that contribute to a good HDM. To transfer images to the brain, HMDs commonly use an LCD panel, which is the same type of panel found in smartphones, modern televisions, and computer monitors. Organic Light-Emitting Diode (OLED) displays are also used in advanced HMDs which provide better colours and graphics with lower blue light factors. The time between input and output is referred to as latency. That is the amount of time it takes for the image in your virtual reality world to catch up to your new head position each time. VR technology requires very low latencies to fool your brain into thinking you are in an immersive world. A truly exceptional experience typically has a latency of 20ms or less; anything more and users will notice an unnatural lag. The refresh rate describes how quickly a display's content can change over a given period. Modern LCD computer monitors can do this at 60 frames per second (60Hz or 60fps), and the more frames used in each given second, the smoother and crisper the motion appears. Because virtual reality is intended to impose a sense of immersion and realism, crisp 'real looking' motion and the absence of motion blur are critical to the overall experience. A minimum of 60 frames per second is required to achieve this, but some HMDs

with rates as high as 120Hz already exist. Your HMD requires accurate head tracking technology to recognize your head's position and transform it into other data. We can now put a multi-axis accelerometer on a chip, allowing infrared tracking cameras to accurately watch markers on the HMD and relay positional data to the computer, thanks to advancements in smartphone technology. Eye-tracking is the tracking of your eye movement. It detects where your eyes are looking and can extrapolate that information into visual data. With this technology, HMD changes the depth of field of the visuals on-screen to simulate natural vision more closely. Virtual characters respond to your gaze, or the user can use their eyes to select items in your virtual world quickly. Users will be able to use their headphones in conjunction with the HMD for audio. Some HMDs now come with their optional headphones, while others do not. Positional, multi-speaker audio that creates the illusion of a three-dimensional world is a technology that already exists, such as surround sound speakers that can be seamlessly integrated into existing HMDs.

The following are some of the best HMDS on the market right now, along with their specifications. Facebook's Oculus Rift and HTC Vive were the three best-selling VR headsets according to (Dexter & Ridley, 2021). Oculus was a pioneer in Virtual Reality hardware for video games. Originally funded on Kickstarter in 2012 and engineered with the assistance of John Carmack. Facebook acquired Oculus in 2014, bringing the company's high-end VR HMD to market for consumers. In 2019, the company released the budget-friendly stand-alone Oculus Quest and the advanced tethered Rift. The following are the specs of the Oculus Rift controller provided by the company & (Bardi, 2019)

Oculus Rift

Figure 2.2.2.2 - Oculus Rift

Price: $399

Display Technology: OLED

Resolution: 2160×1200 (1080×1200 per eye) at 90 Hz

Field of View: 110 degrees

Additional Equipment Required - High-end PC

Tracking System: 6DOF (3-axis rotational tracking + 3-axis positional tracking) through USB-connected IR LED sensor, tracks via the “constellation” method

Controller: Xbox One game controller, Oculus Touch motion-tracked controllers

Audio: Integrated 3D audio headphones

Weight: 460 grams
Platform System: Windows, macOS, and GNU/Linux
Connectivity: HDMI 1.3, USB 3.0, USB 2.0

Since its consumer release in 2016, the HTC Vive has been one of the best VR HMDs on the market. HTC's Vive was the first VR HMD to support SteamVR (Popular Gaming Platform). Since its release, the Vive has been in fierce competition with the Oculus Rift, as both headsets are aimed at the same top end of the VR enthusiast market. The Vive has proven to be a reliable workhorse for enterprise solutions while also providing one of the best consumer VR experiences on the market. The Vive was first released in 2016 and has gone through several iterations, which brought out the Vive Pro, Vive Pro Eye, and HTC Vive Cosmos were released in 2018.

HTC Vive
Price: $499
Display Technology: Dual AMOLED 3.6'' diagonal
Resolution: 1080 x 1200 pixels per eye (2160 x 1200 pixels combined) at 90 Hz
Field of view: 110 degrees
Additional Equipment Required: High-end PC
Sensors: SteamVR Tracking, G-sensor, gyroscope, proximity, and front-facing camera
Audio: Integrated microphone, stereo 3.5 mm headphone jack
Input: Multifunction trackpad, Grip buttons, dual-stage trigger, System button, Menu button
Weight: 470 grams
Platform System: Windows, macOS, and GNU/Linux
Connectivity: HDMI, USB 2.0, Power, Bluetooth, Micro-USB charging port

Figure 2.2.2.3 - HTC Vive

The oculus rift was utilized for this project to calibrate the VR environment movements of the character as well as the mapping of the interface keys. The project is pre-configured to work with the Oculus Rift, however, any of the previously stated VR HMDs can be used. The rift unit's three-axis rotational and positional tracking produces such fine data that the project can employ precise measurement for trigger boxes within the environment. The controllers enable the user to navigate the map and interact with the levers and switches.

2.2.3 Learning in a Virtual Environment

VR is important in education, and possibly at its best, when visualizing situations that are not generally available to the student; therefore, it has yielded apparent success in training in safety-critical situations, but it is just as valid in other situations such as planetary exploration (McGreevy, 1993) or fashion design (Thalman, 1994). It appears that, at the very least, VR systems should be utilized to replicate an idea that is difficult for the user to visualize. This could be due to a lack of experience or abstraction, which can be reinforced for the user by utilizing VR's visualisation strength. Furthermore, the system must foster independent thinking while also enhancing the extent to which a student learns and retains certain information. (Mavor, 1995). It is also preferable that the system boosts the student's motivation to learn. In essence, this fits the psychological theory of constructivism (Peter J. Fensham, 1994), which advocates an experimental approach to learning. Constructivism has grown in popularity since the late 1960s (Mavor, 1995), and computer learning in general, and VR, lend themselves to such an approach. Motivation is fundamental in all learning, and constructivists argue that an exploratory approach to learning can at least sustain and possibly boost motivation. As a result, if the program can raise motivation, it will improve students' learning.

The use of VR in education may also be determined by the sort of learning. According to learning styles theories, there are multiple ways to learn, and some people learn better with some methods than others because they have distinct approaches to information processing. According to the widely used visual-auditory–kinaesthetic learning styles paradigm (Walter B. Barbe, 1981), there are three types of learning styles: visual, auditory, and kinaesthetic. Because VR headsets support complex visual renderings, audio, and movement tracking, all three of these learning methods can be targeted in a single application. Though there has been significant debate about learning styles theories, as noted below, having a single learning environment that can accommodate many learning styles could be very useful because it would be acceptable for a much larger spectrum of individuals.

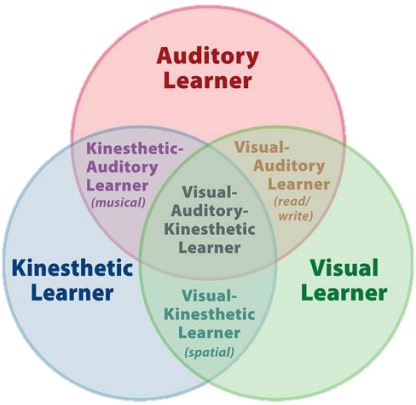


Figure 2.2.3.4 - Types of Learning Styles

Other learning style models emphasize the importance of learning through multiple perceptual modalities, many of which can be targeted in VR (Cassidy, 2004). It has been

proposed that having a variety of learning methods is useful; (Jorge Gaytan, 2007) concluded that using a variety of instructional approaches to appeal to students' learning preferences is beneficial. As students enjoy numerous modes of information presentation, VR activities could be developed to offer multiple learning methods so that learners can select to engage with the learning materials in the manner that most interests them. (Heidi L Lujan, 2006).

(M. Bellamy, 2011) did a case study by simulating genuine experiments with basic online interactive simulations. Eighty-three per cent of their students said they found the online simulations helpful or very helpful, and their presenters said the students seemed considerably better prepared and willing to answer questions after doing the simulations. These and other instances demonstrate the value of simulated settings as alternatives to real-life scenarios.

To summarize these points, it is critical that the strengths of VR be used in learning and, as a result, the system: 1. exploit the visualisation of the world it depicts; 2. encourage the student to explore and learn by constructing their knowledge patterns; and 3. Increase the learner's motivation to learn by giving the student a sense of presence in the environment.

2.3 Class Relationships

Class diagrams are your system's or subsystem's blueprints. A class diagram illustrates a class's attributes and operations, as well as the limitations imposed on the system. Class diagrams are used to model the system's objects, depict the relationships between them, and define what those objects perform and the services they provide. Class diagrams are useful throughout the system design process. Since they're the only UML diagrams that can be mapped directly to object-oriented languages, class diagrams are frequently utilized in the modelling of object-oriented systems.

The typical class diagram is divided into three sections: Upper section, Middle section & Bottom section. The upper sections contain the class name, this section is always required, regardless of whether it's discussing the classifier or an object. The middle sections contain the class's attributes & class operations are included in the bottom part.

Some advantages of class diagrams are that they can be used to represent the object model of complicated systems. It decreases maintenance time by offering a visual representation of how an application is structured before coding. It presents a general schematic of an application to aid comprehension. It depicts a detailed chart by emphasizing the desired code to be programmed.

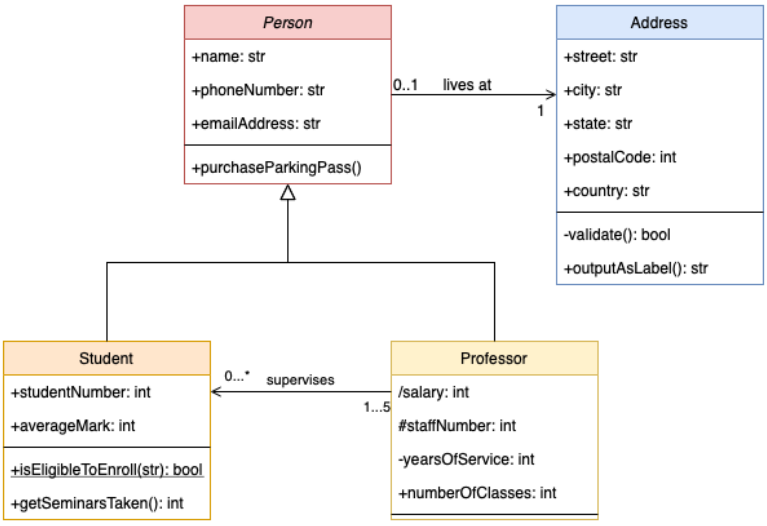


Figure 2.3.5 - Sample Class Diagram

In UML, relationships are used to indicate a connection between structural, behavioural, or grouping elements. It is also known as a link since it describes how two or more items can relate to each other during system execution. Association, Dependency, Generalization and Composition are the four types of UML relationships.

An association relationship is described as a group of people who work together to achieve a shared goal and have a formal structure. It denotes a binary relationship between two things describing an activity. It is an object-to-object relationship. A doctor, for example, may be

associated with several patients. There are several types of associations: one to one, one to many, unidirectional, and bidirectional. When just one object accesses the attributes of the other, it is said to be unidirectional. When both objects have access to each other, it is said to be bi-directional.

In UML diagrams, associations are represented by a solid line connecting two classes. The role names describe the class's role in the associations. The name of the role generally leads to the name of the reference/pointer in the other class. Each association might have a different number at each end. These numbers are known as multiplicity values, and they represent how many objects are involved in the interaction.

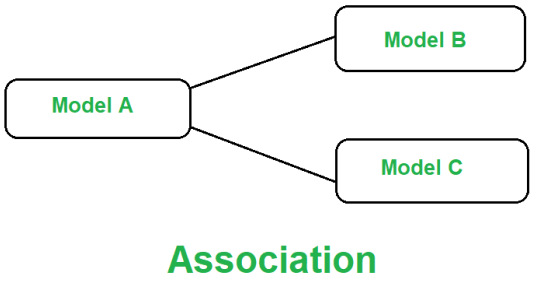


Figure 2.3.6 - Association in UML

As the name implies, two or more parts are dependent on each other in a dependency relationship. A dependency occurs when one class is dependent on another. A dependency, according to the UML definition, is "a using relationship in which a change to an independent object may affect another dependent object." If we make a change to one element in this type of relationship, it is quite likely that the change will influence all of the other elements as well. UML Diagrams will include dependency using a dotted line to demonstrate that there are certain "using" relationships between two classes, but not all dependencies will be displayed because there are simply too many. Dependency is regarded as a weaker relationship that produces no new members in the classes. It is often implemented using member functions with variables that are objects of the other class.

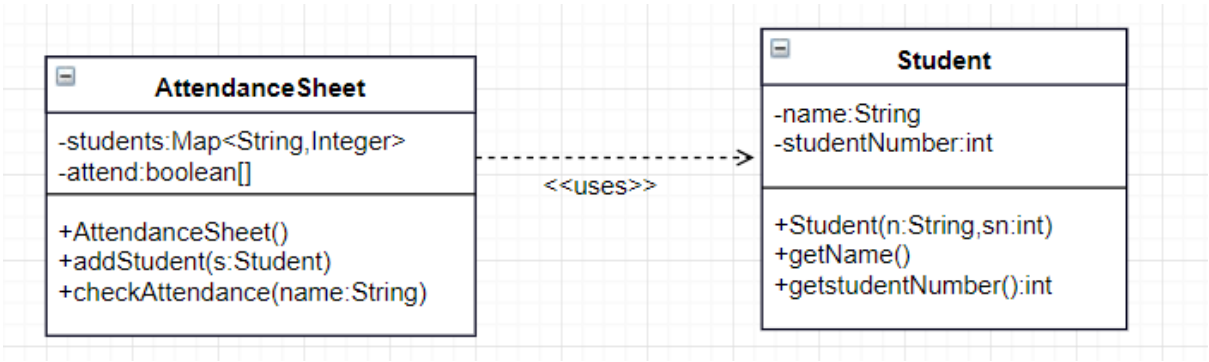


Figure 2.3.7 - Dependency in UML

In generalization, also known as a parent-child relationship. One element is a subset of another more general component. It can be used in its place. It is mostly used to denote

inheritance. Inheritance refers to a class's ability to derive attributes and properties from another class. This is commonly known as an "is-a" relationship. The data members and function members of the superclass or base class are passed down to the subclass or derived class. A child of any parent can access, edit, or inherit the functionality specified inside the parent object through inheritance. A child object can extend its functionality and inherit the structure and behaviour of its parent object.

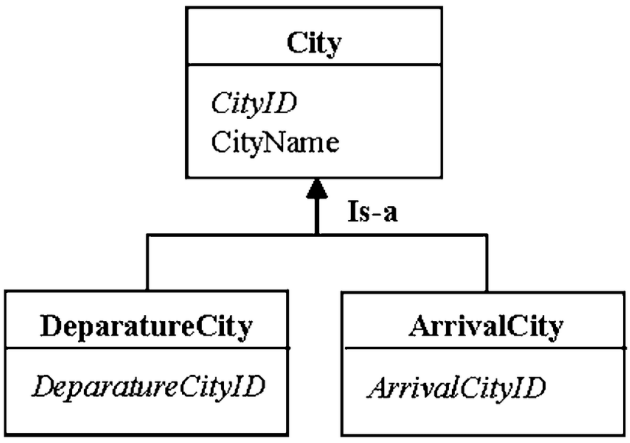


Figure 2.3.8 - Inheritance in UML

Whole/part relationships are what composition relationships are. The "part" is housed within the class that represents the "whole." This usually means that the "part" is constructed by the "whole's" constructor and destroyed by the "whole's" destructor. A "part" item can only be associated with one "whole." Typically, the "part" object is member data of the "whole." An automobile is the most common example of composition. When an automobile is produced, it has a body, but when it is destroyed, the body is also destroyed. The composition relationship is frequently referred to as a "has-a" relationship.

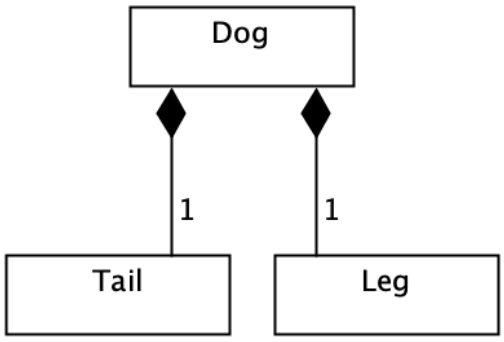


Figure 2.3.9 - Composition in UML

Taking this information into consideration, it was decided that this project would be created in Epic Games' Unreal Engine and pre-configured for use with the Oculus Rift. A basic game design was constructed to incorporate the visual, auditory, and kinesthetic learning styles. A UML class diagram was also created to organise the characters and items in the game, which is discussed in the following section.

3. Design

This application is intended for use with students who have a rudimentary understanding of logic gates. It could be used as a revision tool or to visualize the outputs of the individual gates. Knowing will inform the students on what to do. Even though there are instructions in all the maps, knowing the essentials will allow students to explore the environment while keeping the application's aims in mind. This program can potentially be used as an introduction lab for future virtual reality labs. This application will assist the learner in getting a feel for virtual reality, such as how to move about and interact.

So, when it came to planning this project, the possibilities were endless. The game design could have been anything as long as it had the fundamental logic gates. It might be designed in a first-person action or adventure manner. The possibilities were endless. The difficulty of these activities was also considered when developing the elements and maps. A-Train Track Operator, a Component Manufacturer, a Construction Site Traffic Manager, a Bomb Disposal Unit, and a Gate Operator were among the final suggestions.

So, after much deliberation, it was decided that this project would be built on a character that undertook an adventure with levers and switches that opened doors to different realms. These distinct universes would correspond to three distinct logic gates. The design was chosen since it checks all the senses that must be stimulated for the best game experience. If the game was based on some of the above recommendations, it would not take into consideration mobility and would instead focus on interaction with the screen. As a result, that is how this decision was made.

The next phase in the application's design was to decide which logic gates to use in the application's maps. Because there are seven gates on which the maps might be based, it was determined that the AND & OR gates should be the first two levels because they represent the steppingstones to the remaining gates. The third level would be a gate that combined these two as well as a third gate, of which the NOR gate was picked. It was required to design how the inputs of the gates functioned and how the output would be shown. So, a map was generated for each gate, with each map taking a different approach to how the logic gates' inputs are handled. Because it was an adventure game, it was necessary to choose a theme for this environment so that all the elements could be constructed around that theme. This application's environment was an abandoned old temple on floating islands in the sky.

3.1 Map Design

As previously indicated, a map was constructed for each gate. So, three maps were created in all, each having a distinct floor plan to add a variety to the different levels. The maps all feature a floating island with an abandoned temple, two input methods, a forcefield, status lights, and a teleporter to the next level. When the user enters the correct sequence via the input methods, the forcefield disappears and the status lights indicate that the user's sequence was successful, and they can advance to the next level via the teleporter, which is the temple's door. The two input methods were an old-fashioned pull lever and a floor-mounted switch. These are discussed more in the following section. The maps were categorized as follows:

- Map 1 – AND Gate
- Map 2 – OR Gate
- Map 3 – NOR Gate

As indicated, a rough draft of these maps was generated. A fundamental idea was developed from the preliminary sketch, which can then be implemented. One is built with two floating islands, one slightly higher than the other, as seen on the map. In this game, the player begins on the bottom island and must activate the levers on the walls in front of them in the correct order. When the game begins, the player sees the AND logic, which they can refer to if they mistype the solution. Each lever is built into a standing wall, as indicated in the sketch, and this wall will also have instructions for the player to know how to interact with it. It also indicates which input names are appropriate. A status led is also built into the standing wall. This informs the player of the lever's status If the lever is pressed, the light will turn green; otherwise, it will be off and display nothing. When the input is deemed correct, the forcefield disappears and the gate with the forcefield illuminates green to notify the user that they can proceed. This indicates to the player walks up the steps, through the gate, and into the temple's door. When this occurs, the teleporter will transport the player to the next map in the game.

The input techniques are the only significant design distinction between map two and map one. In this level, instead of levers, there are floor switches that must be activated by standing on them, as opposed to levers in map 1. As illustrated in the initial concept for map 2, these triggers are on a separate floating island from the main one, however once activated in the correct order, the forcefield disappears and the player can continue as in the previous level.

Map 3 includes the NOR gate as well as the AND & OR gate. This map differs from the previous two in that it contains two smaller temples that house the NOR gate's inputs. To obtain access to the input methods of map 3, the player must solve the combination of the OR house & AND house to activate the NOR inputs and then progress up the stairs to the end of the game. To solve the NOR gate challenge, the players must open both houses to determine which input of the nor button is already triggered and which input to untrigger. The NOR inputs on this map are handled with floor switches, however, the AND & OR inputs are made with levers located in front of the temple. The final map was constructed in this manner so that the user thinks, uses, and tests their knowledge of prior levels as the level of difficulty increases and allows them to process the workings of the logic gates.

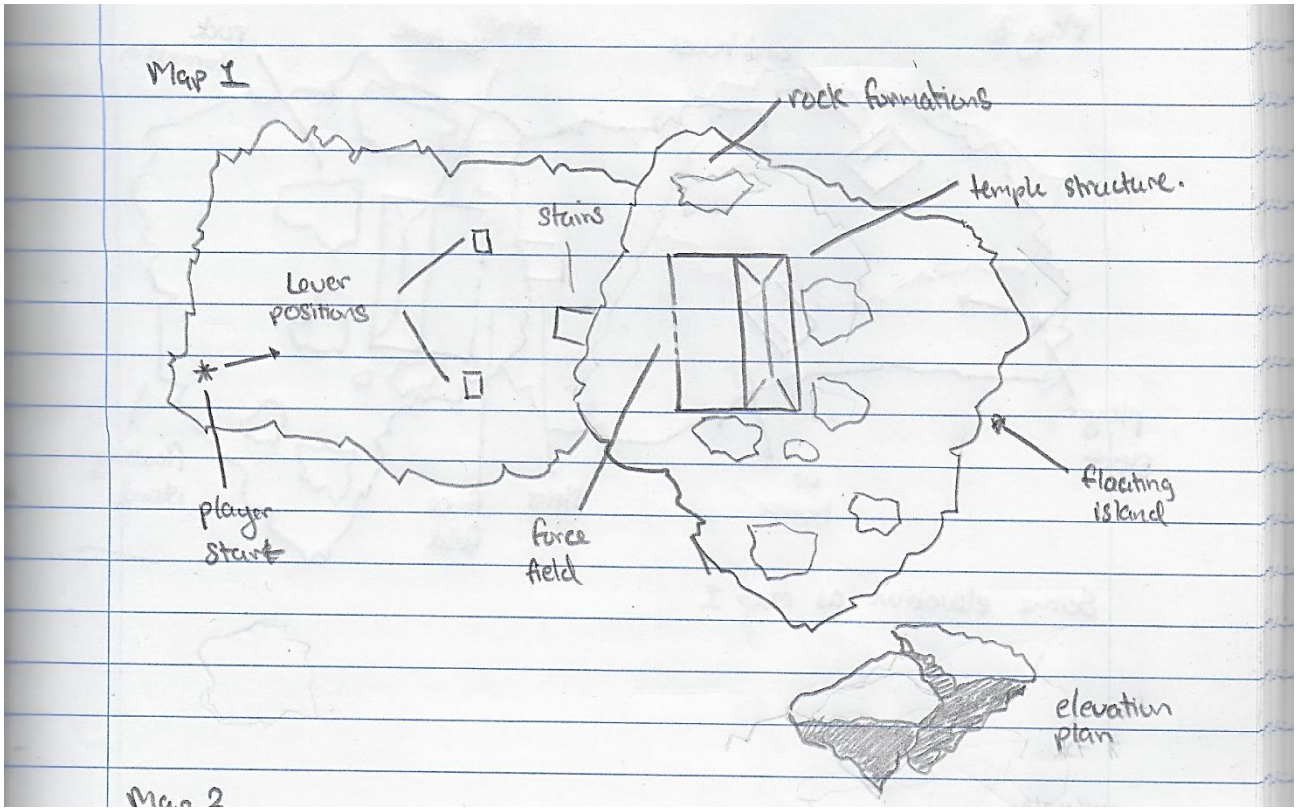


Figure 3.1.10 - Rough Sketch MAP-1

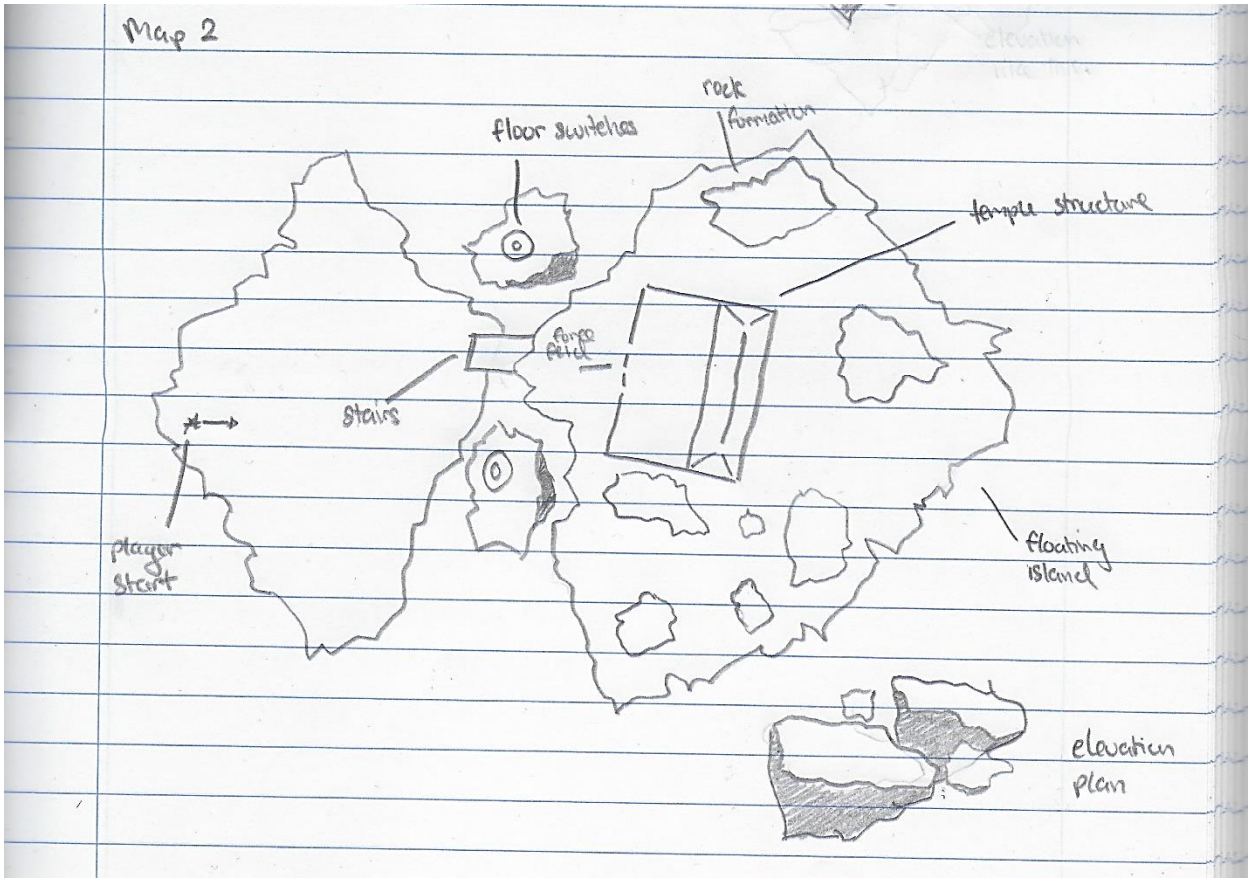


Figure 3.1.11 - Rough Sketch MAP-2

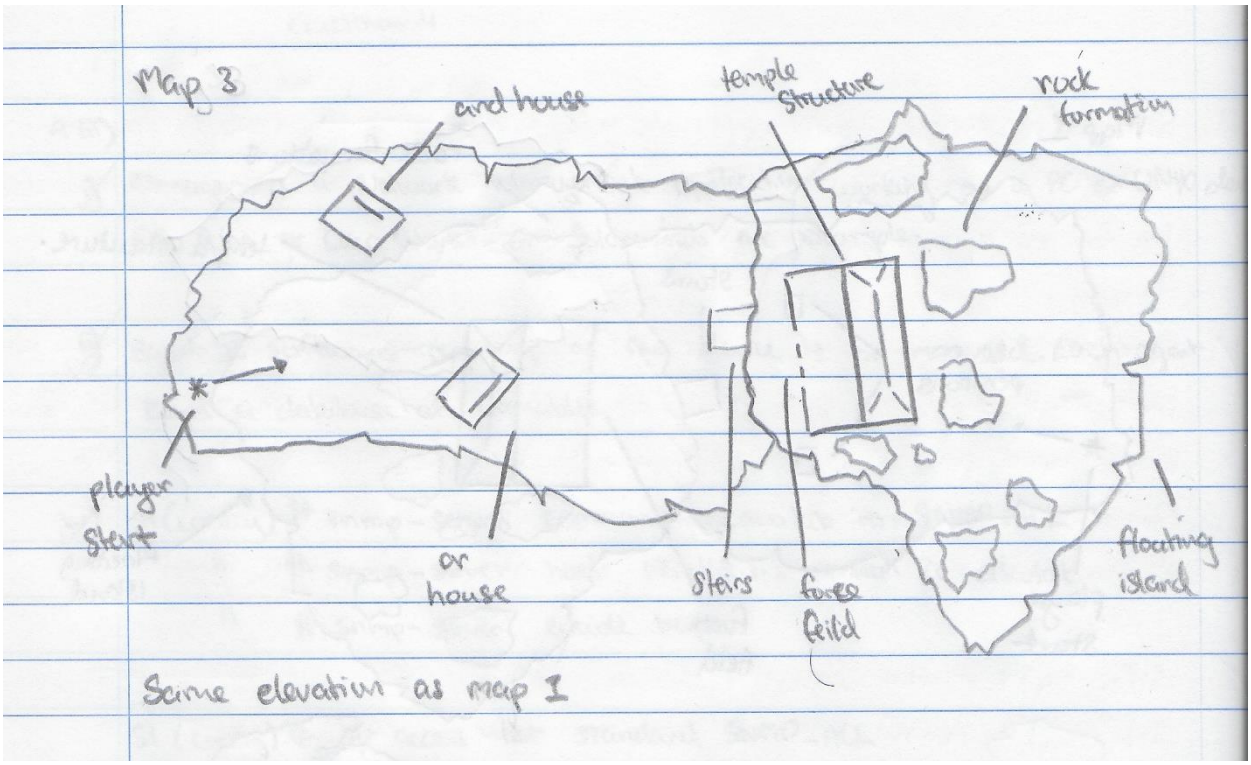


Figure 3.1.12 - Rough Sketch MAP-3

3.2 Switches & Levers Design

As noted in the part on map design, the input methods are a significant component of this application's architecture. Instead of having something as simple as a light switch. It was necessary to build a design that also suited the application's environment and concept. As a result, two distinct sorts of input techniques were created: a lever and a switch. The design of the lever is based on an old medieval style pull lever. There were several alternative designs developed, but one design really complemented the theme of the application. To enhance the immersion of the experience it was determined that this lever should include moving parts that be animated to show whether the lever was activated or not. The sketch below depicts the various elements of the lever and how they could move when triggered.

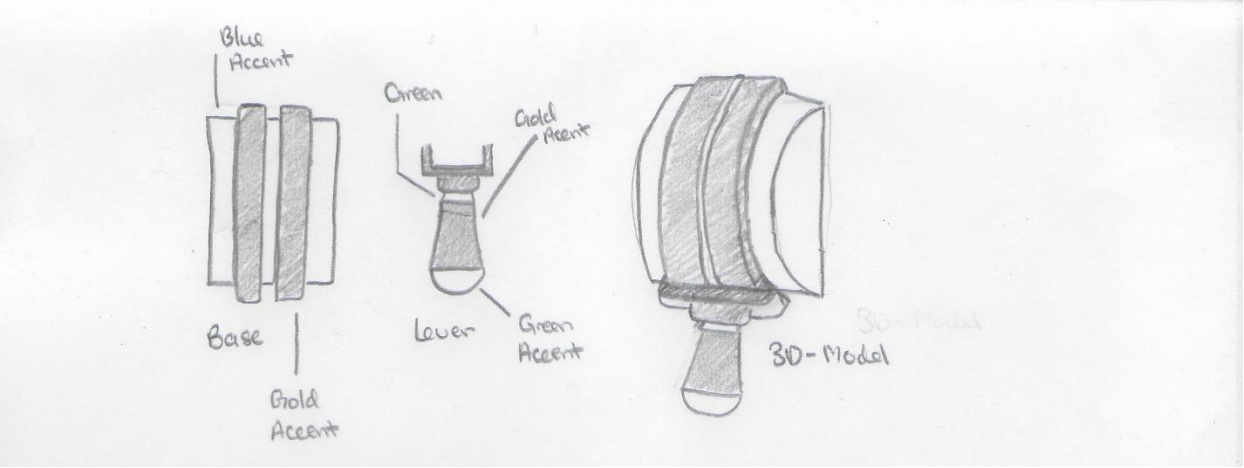


Figure 3.2.13 - Pull Lever Design Sketch

The switch, like the pull lever, had to reflect the theme of the application. So, after several iterations, the following was selected as the final design. The switch is intended for floor implementation because it requires the player to stand on it to be activated. A status light is included inside the switch. When the player walks on the switch, the status illuminates green and remains illuminated for as long as the player remains on top of the switch. It was decided to incorporate the status light inside the switch because it ensures that the players look down on the ground of the environment to ensure that they have full access & movement to the VR equipment. When the player stands on the raised piece of the switch, it will slowly slide inside the switch to demonstrate that it has been pressed and will slowly push back out once the player steps off.

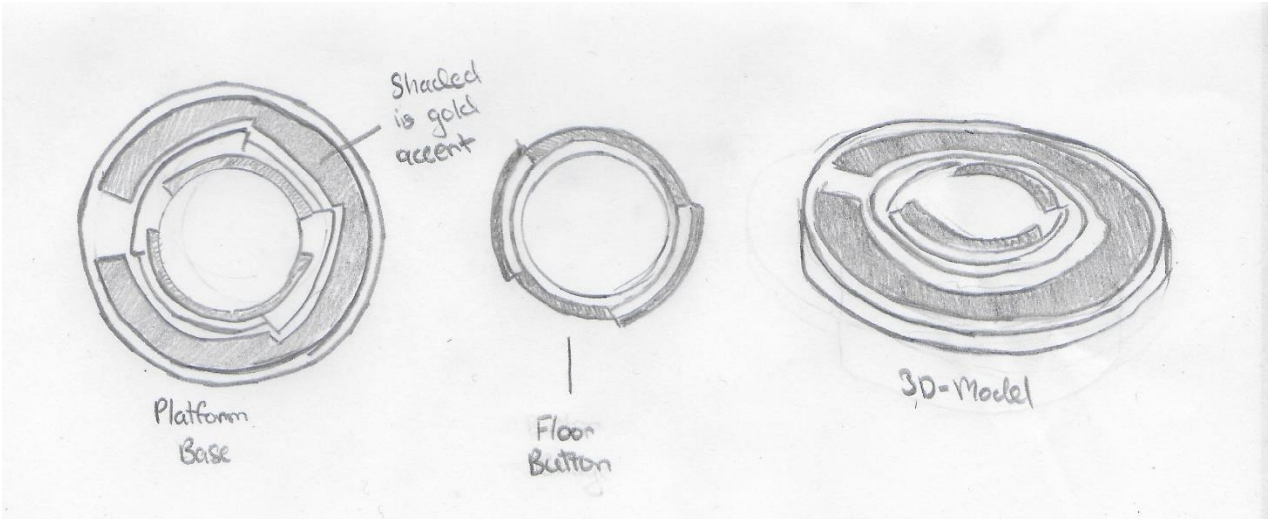


Figure 3.2.14 - Switch Design Sketch

3.3 Game Flow Chart

Although the design portion of this application appears rather hectic, the application's actual working design is quite simple. As illustrated in the flowchart, map 1 sets up the character; this just checks in the system if the application is operating in VR or as a standard game. The following step is to load the map with the correct system based on the information from the previous phase. The next step is to see if the levers were activated in the correct order, and if so, the appropriate status light will illuminate.

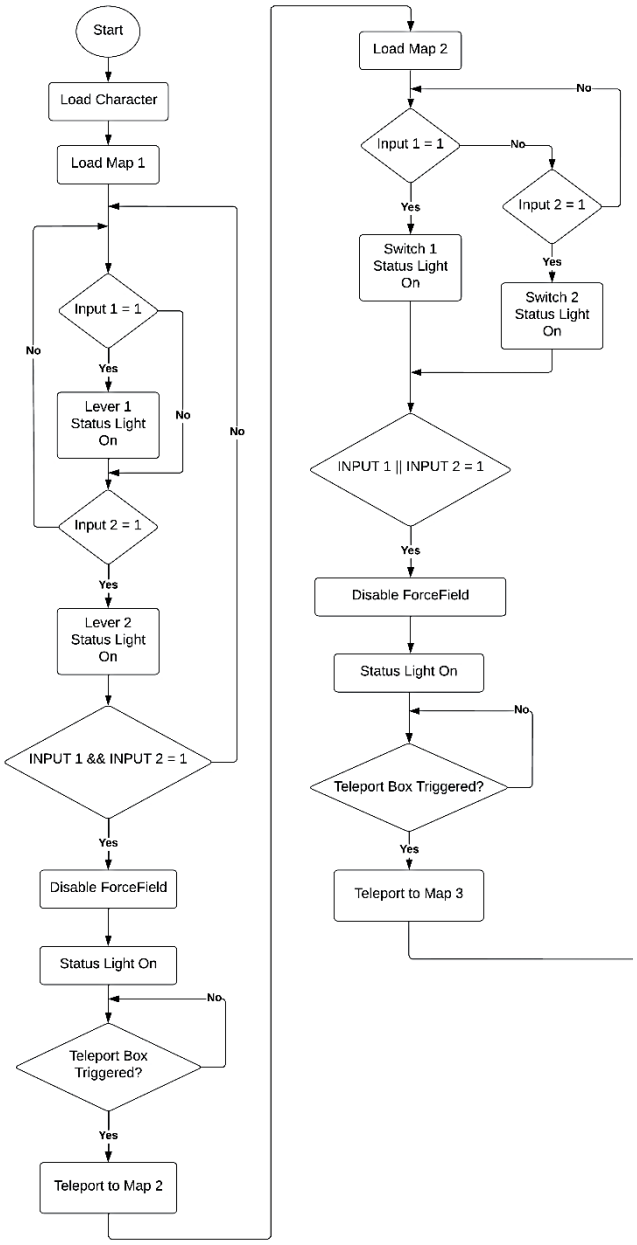


Figure 3.3.15 - Map 1&2 Flow Chart

However, the program will continue to check if the levers in this situation are both triggered, and if these circumstances are fulfilled, the forcefield will be disengaged and the gate status light will turn on, allowing the player to go towards the teleporter. If the teleporter is activated, the player will be transported to the next map; however, the teleporter is only activated when the player moves into the correct position.

The teleporter transports the player to map 2 - the OR gate map. This map follows the same method as map 1 but is somewhat modified to meet the OR gate conditions but uses the same type of operating process. The checking statements are the conditions that have been modified in this flowchart. Instead of requiring both inputs to be activated, the OR gate simply requires one input to be triggered. ` So the application checks to see if this statement has been completed, and if so, it proceeds to the next map.

The third map has the most requirements. When the level loads and begins, one of the NOR inputs is set to 1, and the player must determine which input is triggered and open the forcefield on the OR & AND gate house to obtain access to the NOR Input. As a result, this

flow chart incorporates condition factors from the previous two maps for the separate houses. These criteria are active at the same time as the NOR condition is waiting to be met; once met, this map continues the same production cycle as the others. In this situation, both inputs of the NOR gate must be set to 0 for the criteria to be met. This report's appendix contains a full-sized flowchart.

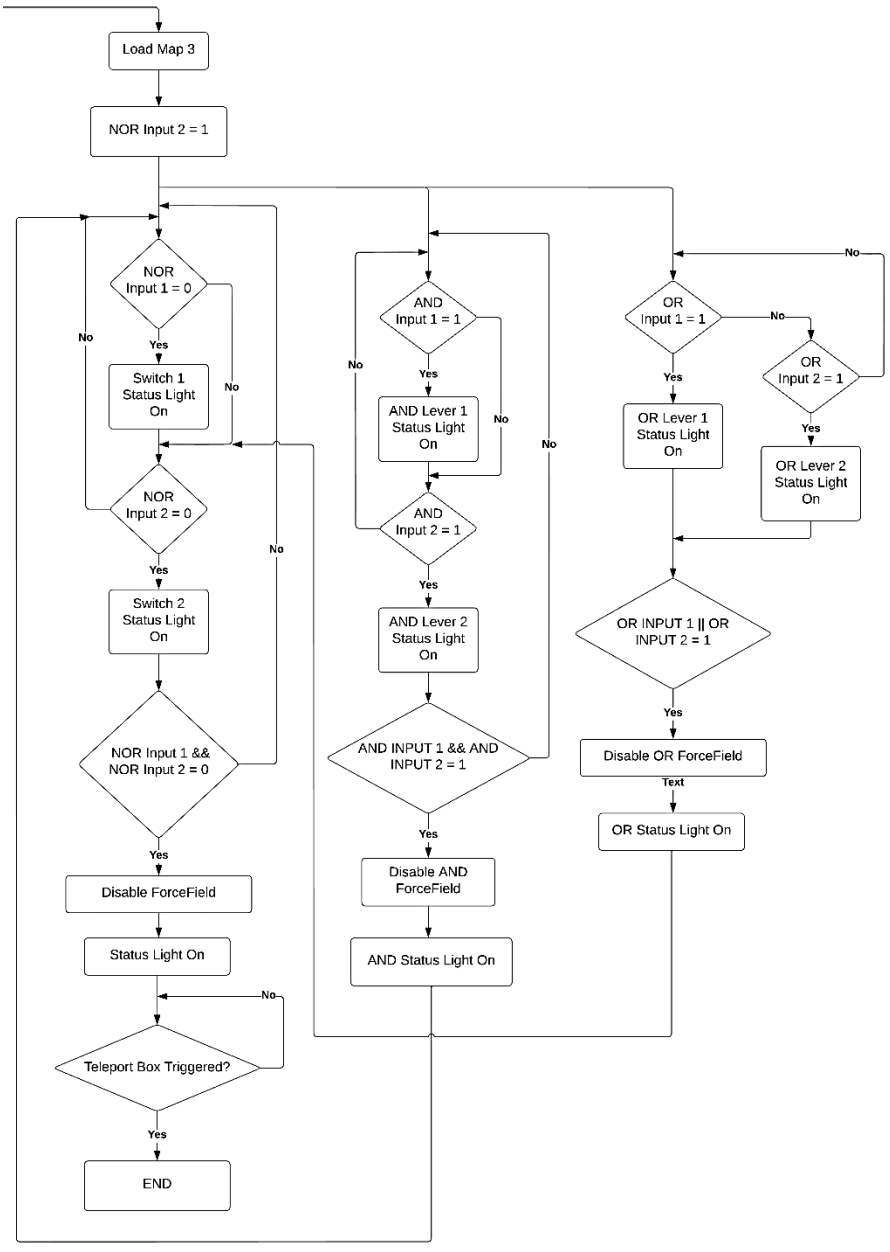


Figure 3.3.16 – Map 3 Flow Chart

3.4 Class Diagram

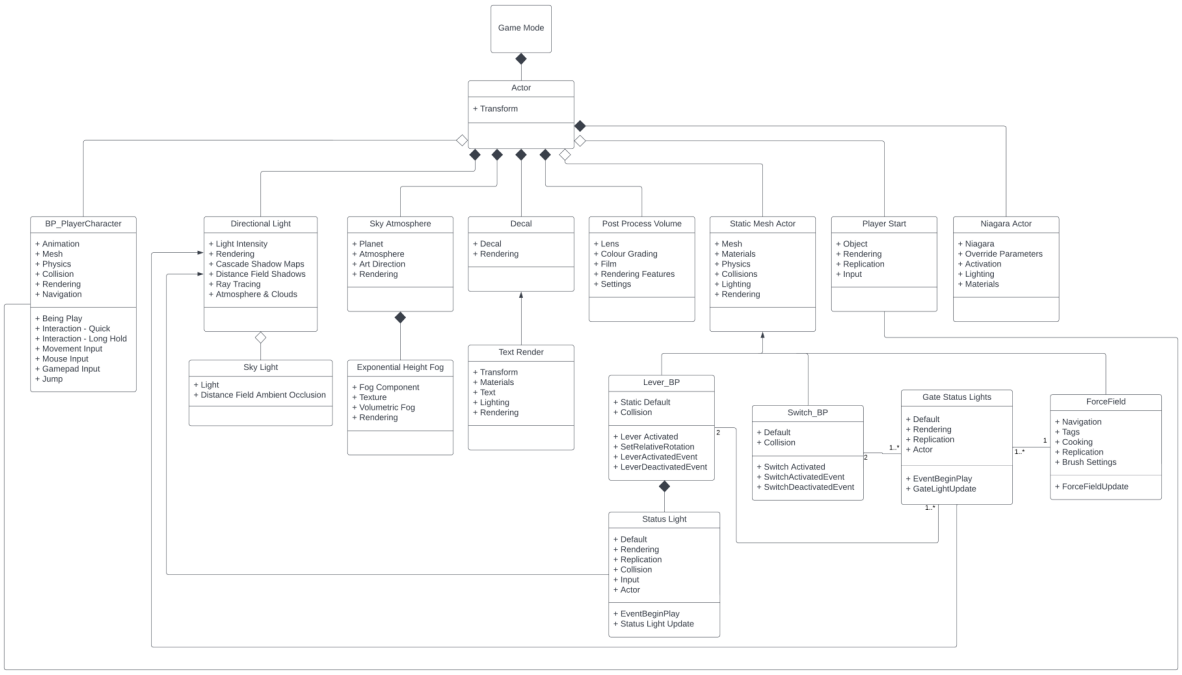


Figure 3.4.17 - Overall Class Diagram

The class diagram for the intended application is shown above. As you can see, there are numerous classes in this design. The main class in this application is game mode. The game mode class is an Unreal Engine built-in class that loads the default settings every time the application is run, such as the game default map, VR settings if an HMD unit is connected, frame rate, and so on. If this class is not properly initialized, it will fail to launch the game properly, resulting in in-game crashes and lags. It may also cause issues when attempting to initialize the remaining classes & assets.

When the game mode class has finished loading, the next most important class is the actor class. Every element or object built in Unreal Engine is comprised of an actor, and the actor class provides the transform property to all other elements. Without the actor class, the majority of the classes collapse. As explained in the previous chapter, these classes have a composition relationship with the actor class. This composition relationship is sometimes referred to as the whole-part relationship. As a result, the classes associated with the black filled diamond cannot exist without the actor class (as a whole) as their lifecycle at the end of scope of the actor class. This class has one attribute, the transform attribute, which is used to modify each element in the application. There are three variables in the transform attribute: location, rotation, and scale. Each object element can be moved around, rotated at a specific angle, and scaled up or down using this. Each of these variables accepts an x,y and z plane

value and modifies the objects on that plane. The transform additionally contains a CapsuleComponent variable with three options: Static, Stationary, and Movable. Static objects cannot be changed or moved in the game; stationary objects can be changed but not moved. Moveable signifies that the object in the game can be altered and moved. For example, a building wall would be created as a static object, but the character figure would be created as a moving object so that the character could move around based on the player's input.

A non-filled diamond connects some of the classes to the actor classes. These classes are in aggregation with the actor. An aggregation relationship follows the same procedures as a composition relationship, except the kid can exist without the parent. The classes that have an aggregation relationship with the actor class are the classes with which the player interacts the most, such as the player character itself, static mesh actor, which is used to construct the levers or switches, and player start, which is the position from which the player always starts when the application is launched. These classes were set up as an aggregate relationship. These are custom developed classes for special functions within the program, hence it may exist without the parent class. It was also chosen to make these relationships aggregated such that if the actor class goes out of scope or crashes, it will prevent the main classes from crashing and therefore prevent a full shutdown, but the other classes can be rebooted with a relaunch. These classes might not even also use the transform attribute since they are constantly updating from another attribute.

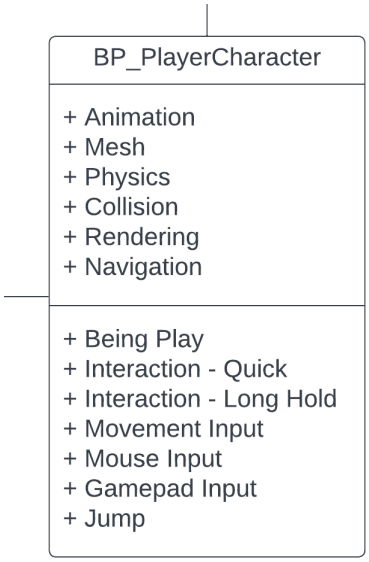


Figure 3.4.18 - BP_PlayerCharacter Class

The BP_PlayerCharacter class manages the character's actions and movements throughout the game. Because this application is in the first person, the character referred to is the actual player, and this class handles how the player's inputs are transformed into the character's actions. This class has six properties as well as six Blueprint Functions. In Unreal Engine, attributes are specific characteristics and unique variables that each class has based on their background actor. Blueprint functions are operations performed by the class when triggered by the player or any other element in the program. The animation attribute is used to connect the object's animation blueprint, in this example the character's animation blueprint. The animation blueprint specifies how the mesh

should move when a specific button is pressed. As an example, consider how the character moves when walking, jumping, or engaging with a lever.

The mesh attribute takes the appearance of the figure; because this is a first-person application, the player will not see the entire appearance of the character; however, to cast nice shadows or when looking down at feet and hands, it is necessary to have a design for these objects even though the player will not see the entire character. The character's appearance can be built with various third-party applications, imported, changed into a mesh file, and used. The physics and collision properties collaborate to create a realistic environment that nonetheless performs realistically. It replicates physics and restrictions like gravity and limits. These properties prevent the character from falling through the island's ground or walking through barriers, whereas the physics attribute causes the character to fall if they walk over the island's edge.

The rendering attribute allows the developer to choose whether or not the mesh and capsule components are visible in the editor. However, it also includes the option to hide an actor in the game, so that any box triggers or volumes, in general, can be hidden in the game while still existing and functioning, but the mesh is not visible in the game. The navigation attribute is more of an obstacle avoidance ability. When enabled and given the map area, this feature prevents the player from becoming trapped on the edge of a map or the corner of an object when attempting to walk around. This characteristic is extremely important for the character class.

The character class's functions or operations are as follows. The first function, BeginPlay, is used to load up the character with the necessary mesh and animation blueprints, preparing it to be cast to the player start class. The second function is Interaction - Quick, which examines if the player pressed an interaction key, commonly "e" or “L1 or R1” on the VR controller and how long it was pre for; if it was less than the set value, this function is invoked. The third function (Interaction - Long Hold) operates similarly, but it is only activated if the key is held down for a longer period than the set value. The quick interaction feature is intended to be used in conjunction with the pull lever and the long hold for the floor switch.

The following functions in this class control the character's motion ability. The movement function scans the keyboard for the keys "W", "A", "S", "D" or "Arrow Up”, “Arrow Left”, “Arrow Down", "Arrow Right" and moves the character accordingly. W or the Up arrow pushes the character forward, S or the Down arrow moves the character backwards, A

or the Left arrow moves the character to the left, and D or the Right arrow moves the character to the right. The mouse input capability enables the character to gaze around by moving the mouse. The character field of view moves with the mouse's yaw and pitch input, allowing the player to see up, down, left, or right. This field of view enables the character to move in any direction. If the mouse is not moved, the character will only travel in a cross pattern, but if the field is moved, the character will move in a range of motion. The gamepad input function combines the previous two functions, but only when utilizing a VR HMD with controllers. The left-hand joystick controls yaw and pitch, allowing the player to look around, while the right-hand joystick controls character movement. When the spacebar is pressed, the character will jump to a specific height, which can be used to dodge obstacles that the player may discover. The gravity parameter in this class's physics attribute will return the character to the surface rather than floating in the sky. These instructions were simplified and reduced to single buttons on keyboards and controllers, which is very useful for people with physical disabilities. Instructions are also given at each level for anyone who is deaf and cannot hear the narration component.

This class has two types of relationships: aggregation and association. As previously stated, the character class has an agreement relationship with the actor class, and it also has an association relationship with a Player Start class. When fully loaded, the begin Play function gives this character to the Player Start class.

The next custom class is the Player Start, which contains four attributes, one of which was described in the previous class. The object attribute accepts the Player Start Tag variable, which is essential if the program has many characters; this tag variable accepts one character and assigns that character's starting position to where the Player Start object has been set.

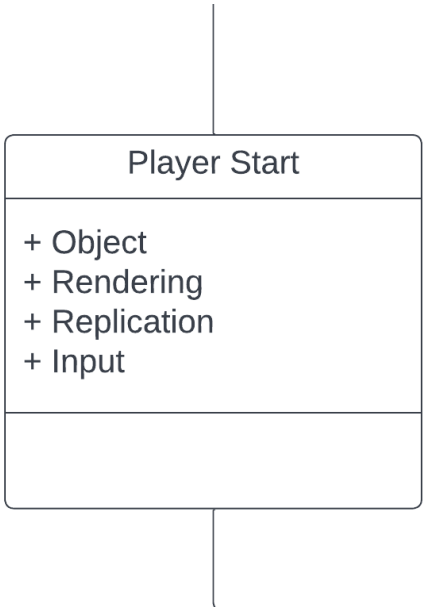


Figure 3.4.19 - Player Start Class

During map loading, the replication attribute in this class loads the actor onto the network client. If several playerStart objects are constructed, this attribute is linked to the object attribute. This feature ensures that all character’s spawn and are loaded into the launching map rather than being abandoned. In this class, the input

attribute is used to specify a default character who will send any type of input or response to this class. For example, if the character somehow doesn't load properly, this class can produce an error message informing the player of the error. It also contains an input priority variable, which may be modified to increase the visibility of this input in a stack. The playerStart class likewise has two relationships: one with the actor class and one with the PlayerCharacter class, as previously mentioned.

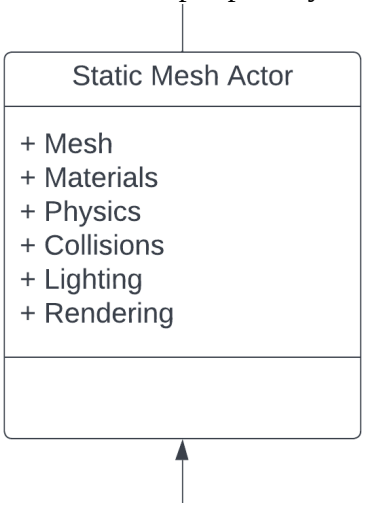


Figure 3.4.20 - Static Mesh Actor Class

Static Meshes are a basic sort of renderable geometry in Unreal Engine. The StaticMeshActor is used to populate the environment world with these models. The following custom classes are derived from the Static Mesh Actor class. A static mesh class had six attributes, some of which were already discussed above; the remaining attributes are discussed here. The Material attribute enables the developer to load material textures for the static mesh object. For example, the floating island is a static mesh object to which a rock texture can be applied via the materials attribute, giving the island the natural appearance of such a formation.

This class's lighting attribute allows you to activate shadows for a physical element. It can also override the existing lighting configuration and generate a customised lightmass. A Lightmass generates lightmaps that include complicated light interactions such as area shadowing and diffuse interreflection. It is used to compute sections of the lighting contribution of stationary and static mobility lights. This class also has two relationships: an aggregation with the actor class and, as previously stated, this static mesh actor class is inherited by three additional classes: Lever BP, Switch BP, and Forcefield. By inheriting this class, the other class gains access to the class's attributes as well as their own.

The next custom class in this application is the Lever BP class. The lever class is composed for the pull lever stated before in this report. The lever has two characteristics and inherits the attributes of the static mesh actor; this class also contains four functions. The static default attribute is a select option to which the lever should begin in the triggered position, this attribute allows changes to be made to the input values for each map. The lever activated functions work just as the name implies: when the character interacts with the

lever and triggers it, this function is triggered. The setRelativeRotation function is used to animate the lever when it is triggered. When the character pulls the lever, it moves in the opposite direction of its initial location. If the lever handle is facing up, this function will rotate it to face down, and vice versa.

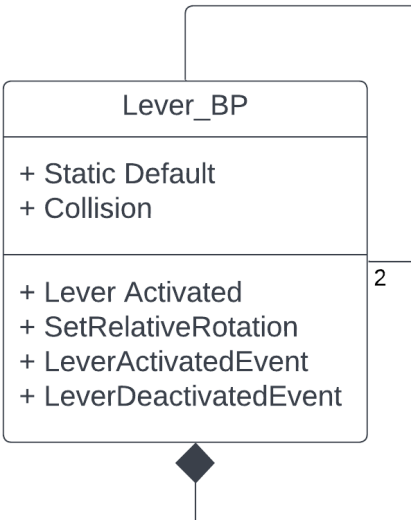


Figure 3.4.21 - Lever_BP Class

LeverActivatedEvent and LeverDeactivateEvent function just as their names suggest. In actuality, these two functions work similarly to the Lever Activated function, however instead of activating the function, these functions cause an event to occur, which may be numerous things at the same time. For example, when the event is triggered, all the lights on the maps become red and the forcefield pulsates. The distinction between an event and a standard function trigger is that an event can be global, meaning that it can cause events to occur in multiple classes with the same name. The deactivated event performs the same function as the above, but only when the lever is deactivated.

This class has three relationships, one of which was previously discussed as the inheritance relationship to the static mesh actor class. The second relationship that this class has is with the gate status light class. This link occurs because the functions in the gate status light require lever class objects. This association relationship has a multiplicity value of one-to-many gate status lights for every two levers.

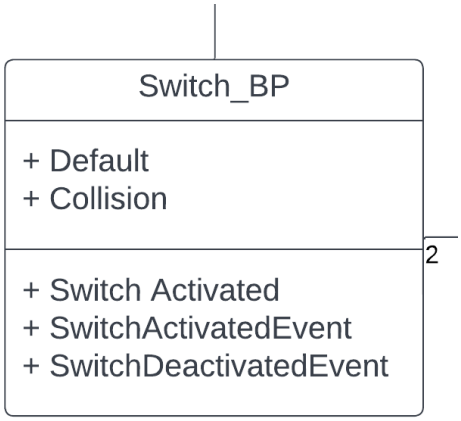


Figure 3.4.22 - Switch_BP Class

The Switch BP is the next custom class. This class works identically to the lever BP class, but the difference that necessitated the separation of these two classes was the properties that these two input methods required for the character to interact. The switch BP required a box trigger, which had to be connected to this class's default attribute. As a result, the way the functions were triggered changed, necessitating the use of two classes. This class has two relationships: one with the static mesh actors as previously indicated, and another with the gate status lights class for the same

reasons as the Lever_BP. This class also has multiplicity values, with one-to-many gate status lights for every two switches.

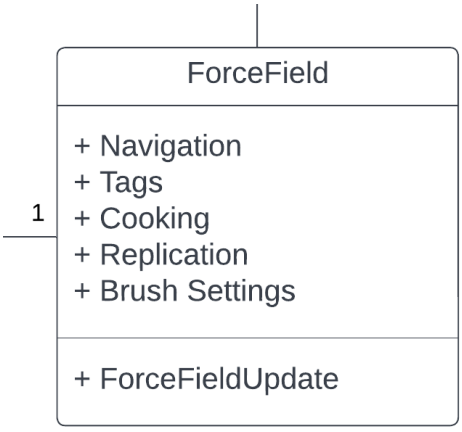


Figure 3.4.23 - ForceField Class

The forcefield class is the final to inherit from the static mesh actor. This class has six properties. The tags attribute is used to group and categorize an array of tags that can be used to access a specific forcefield when scripting in a different class. The cooking attribute has an editor-only setting that, when enabled, only reveals the forcefield to the developer. For normal players, the forcefield exists but is invisibly and will not allow characters to pass onto the teleported as normal. The brush parameters enable developers to construct a blocking volume that appears in a specific form, thickness, hollowed or even tessellated. This can be used to provide numerous forcefields piled on top of each other with various appearances. The force field update function determines whether the two input methods were triggered in the correct order and whether the logic condition was met. If this returns true, the forcefield is removed, allowing the character to progress to the next level. This class also has two relationships: one with the static mesh actors, as previously indicated, and another with gate status lights. The forcefield has an association with the gate status light, and when it turns on, the forcefield disappears. There are one too many gates status lights for each forcefield.

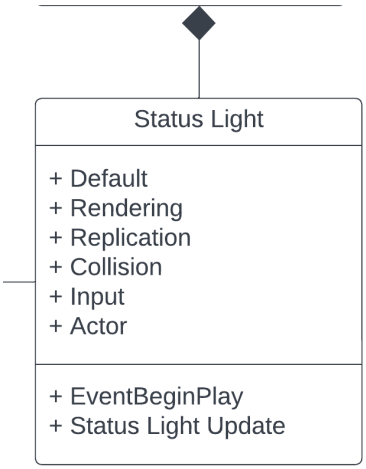


Figure 3.4.24 - Status Light Class

The status light class was built to display the lever's status. The status class's actor attribute accepts the lever object to which the status light is associated. When the application is launched, the Event Begin Play function checks whether the levers have been triggered to illuminate if the lever has been triggered. When a lever or switch is triggered, the status light update function is called and the status light object checks to see if the condition has changed, in this case, if the lever was triggered or untriggered, and vice versa. The status light class has two relationships: one is an inheritance relationship with a directional light class (a

prebuilt class with UE) and has access to its attributes, and the other is a composition relationship with the Lever BP class because if the lever does not exist, the status light class does not need to exist either because there is nothing to show the status of.

The gate status light works exactly like the status light, except instead of only one actor, the gate status object requires two actors to check if the conditions in the function statements have been met. This class has four relationships: an inheritance relationship with the directional light class like the status light class, and three association relationships, one with the Lever BP class, one with the Switch BP class, and one with the forcefield class. The previous explanation of the connected classes stated the existence of all relationships.

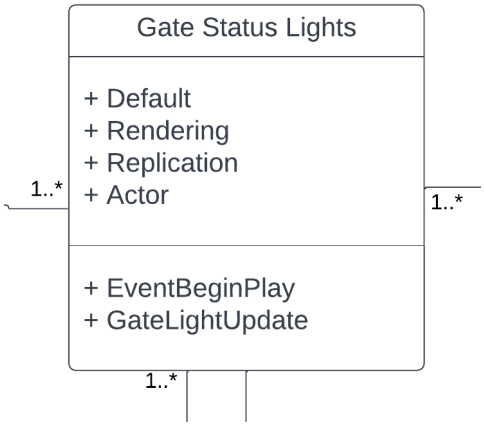


Figure 3.4.25 - Gate Status Lights

3.5 Approach to Implementation

This application can be created in a base version utilizing the above design and approach because it fits the present project aims. An application that incorporates any type of student's visual, auditory, and kinesthetic learning styles and assists them with problem-solving through logic gate principles. The benefits of this approach are that all the learners need to remember about a specific gate is embedded in the levels. This enables any student (from beginner to advanced) to learn at their own pace. They can walk about at their convenience and experiment with alternative configurations to discover why each input equation behaves differently according to the truth table for the logic table. One of the approach's disadvantages is that some student minds regard its application as too incredibly simple or foundational. If they have prior familiarity with various VR environments. Letting users make custom maps by giving them the assets and allowing them to pick a logic gate and create a level using their own activity is an easy way to avoid this. This custom map can be tested and, if appropriate, integrated into the program, and by making this an extra quest or lab, students' custom maps can gradually extend the levels of the application. This design will also allow future developers to utilize this software as a basic system and put on more new levels that cover the rest of the logic gates or add new topics of interest to the game as the character progresses through the game.

The implementation stage of the aforesaid design is the next step in the design of this application. This implementation will begin with Unreal Engine, and for some specific designs and textures - components available in the marketplace may be required to make the environment realistic, such as a grass texture for the surface of the floating island. The coding element, of this design, will be created in Unreal Engine's blueprints component. This will enable full animation, interaction, and statement verification. If external coding is required, it can be integrated with the script, if blueprints do not offer the possibility to build it. The above-mentioned components obtained via the UE marketplace are covered by the engine's EULA copyright. Any custom user-created assets downloaded from the marketplace, on the other hand, will receive full credit in the product credits. If this application is made available to the public, Epic Games must be notified using the web form if you are generating money from it. After generating one million USD in gross revenue, it is necessary to pay UE a royalty fee of 5%.

4. Implementation

To begin implementing the above-mentioned concept, a step-by-step method was designed with several critical factors that were necessary for each map/level. Each map requires an atmosphere, which is made up of fog, sunlight, clouds, and shadows, among other things. These atmosphere characteristics make the VR environment appear more lifelike. Following the atmosphere elements, the accompanying crucial elements necessary in each level are the physical features that the character will see and experience, such as floating islands, temples, staircases, and so on. Following that is the aesthetic part of the design, which includes elements that make the environment look less basic and dull, such as Rock Formations and Post Process Volume. After creating the environment with the aforementioned elements. The following set of elements is those with which the character interacts or has a role in influencing the properties of the elements. Things like forcefields, levers, switches, status lights, teleporters, and so on. Then there are custom elements designed specifically to aid the player proceeds through the levels, such as instructions, truth tables, and elements that are presented on the screen (like on the floor or a wall). The final set of elements required in each level is the character and the player's start. After these two elements are implemented, the environment has met the required criteria; anything else added simply adds atmosphere to the level. This strategy ensures that all essential parts are developed initially for the environment and that a working version of the application is established.

4.1 Creating a new project & map

The complete program was built with Unreal Engine 4.27.2. Following the launch of UE, the following procedures are used to generate the proper setup for the VR application. A new project must be established using the game category option, which instructs UE to import their game packages, textures, and some game-related meshes. The VR template must be selected from the next menu, as well as the option to enable starter content, as this imports, the game packages and textures indicated above.

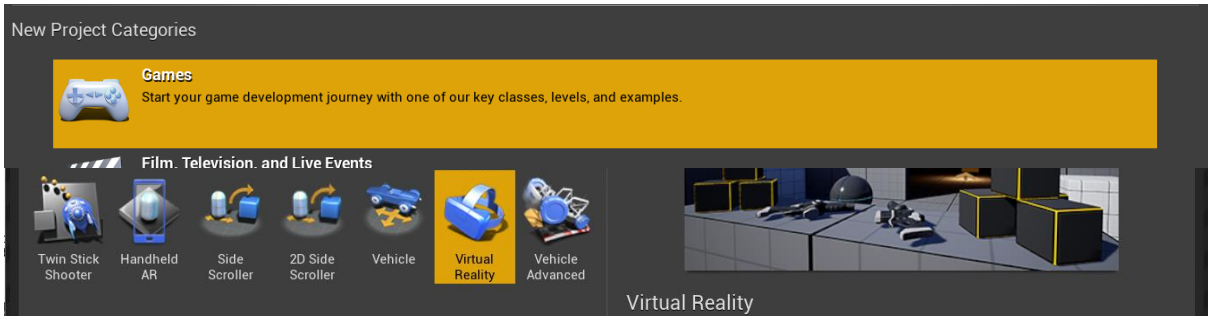


Figure 4.1.26 - Unreal Engine Menu Settings

After the project has launched and loaded, a new level is created using the toolbar at the top of the software, the file tab, or the shortcut CTRL+N. Given that this is map 1 or level 1, it is important to configure the game mode of this project because a default map has been established. Setting the default map in game mode is critical because this is the map that will be launched when the application is launched. The game mode may be found in the project settings; to get there, click the edit tab in the toolbar and then select project settings from the dropdown. This will open another window with all of the project's configuration options, and on the left of this window is a panel with a variety of settings options; from that list, select "Maps & Modes." There is a section labelled default maps within the maps & modes panel, as well as a variable called "game default map." Select the newly produced map from the dropdown input box and save and exit the settings. Once generated, the character must also be linked to the default pawn class variable in the default modes section just above.

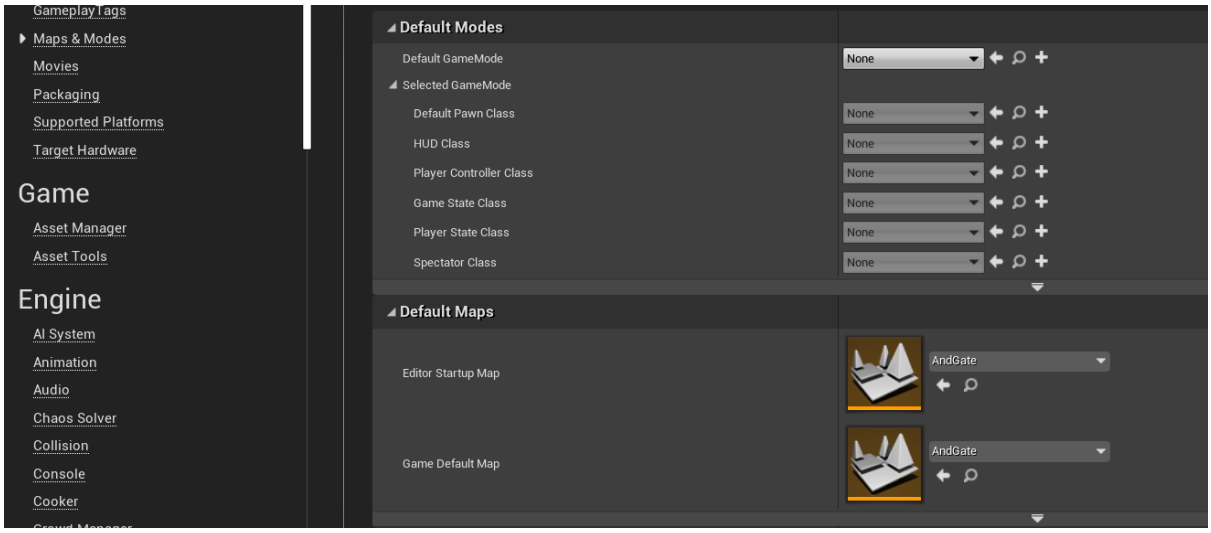


Figure 4.1.27 - Maps & Mode Settings

4.2 Floating Islands

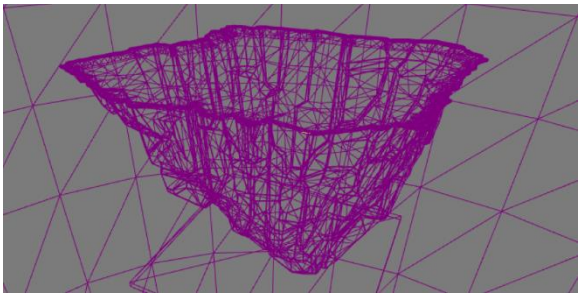


Figure 4.2.28 - Island Wire Mesh

Since the game is set in the sky on floating islands, the initial concept generated was a floating island. The mesh structure of the floating island in Map 1 is depicted. This structure was created and then uploaded to the UE Marketplace free assets called the “Fantasy Rocks” by The Pocket Dimension. Importing this wire mesh into the project is as simple as just drag and drop the package file into UE. These wire meshes will be created into a Static Mesh Actor object so referring back to the static mesh class it will gain those attributes. Wire Meshes can be also can be also

created within UE using the built-in architecture tab. This architecture tab enables the developer to create wire mesh architectures for the construction of custom components. The island's unusual shape was achieved by flipping the mesh upside & stretching the wire mesh into the desired shape. This allows the developer to create various islands to have distinct appearances. To construct the overall shape, this mesh structure was combined with the plane class. The plane class uses the mesh structure to build an outline, which the player can stand without falling through. The finished piece gives the appearance of a floating island, as illustrated below. Rather than having the surface of these islands be empty and only covered in stone. In addition, a grass overlay wire mesh was also imported and created using the above process. A grass overlay wire mesh was also imported and constructed using the previous procedure. Figure 4.2.30 shows how to line up both the grass and island formation parts to make a full island. The next step in making the floating island is to finish it off with materials.

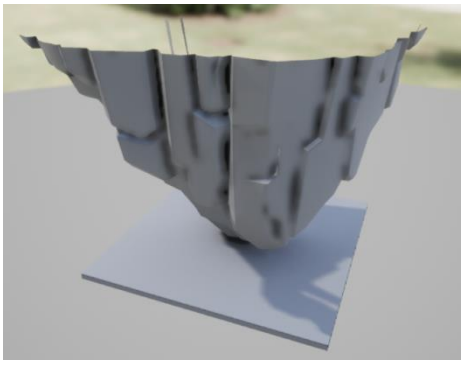


Figure 4.2.29 - Island Wire Mesh

A materials package was required to produce the finishing look for both the island and the grass overlay element. After hours of digging, a texture library including textures for an animated gaming scene was discovered. It is an improved version of the Unreal Engine Starter Content. It can be found on the market under the name "Starting Kit Advanced." Once downloaded and unpacked, transfer the folder to the same area where the game file is saved to avoid the textures not being discovered when the application is opened if the folder is no longer in the previously indicated location. The materials folder will be shown in the UE Editor's content browser. If the folder does not appear, it can be added to the project by selecting "Add/Import."

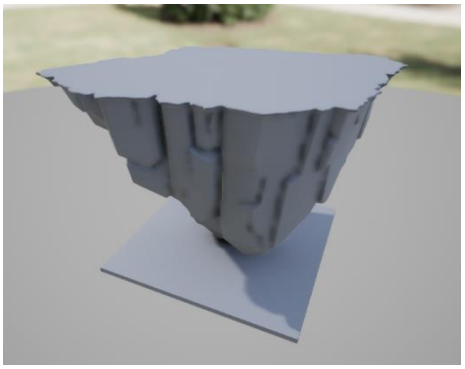


Figure 4.2.30 - Island with Grass

The next step is to connect the necessary material to the wire mesh. As previously indicated, the wire mesh is a static mesh actor. The materials attribute makes it simple to attach the texture to the wire mesh by clicking the mesh element and looking in the details panel for the wire mesh object's properties. One of these attributes is the materials, and by using the dropdown, the appropriate material may be selected and linked with the mesh. After scanning through the materials, the most appropriate material that also matched the theme of this application, for the island formation was "ML_Island_BOT_L," and the most appropriate

material for the grass was "Grass_BaseInstance." Figure 4.2.31 shows how the elements appear after the materials were linked. Once that is connected, a floating island is formed. A series of floating islands can be used to gently build up the environment by duplicating this original item and altering it by rotating, scaling up or down, and inserting it in different positions, as illustrated in Figure 4.2.32. The following phase in implementing the design is to build the atmosphere characters mentioned above, such as light, fog, and so on.

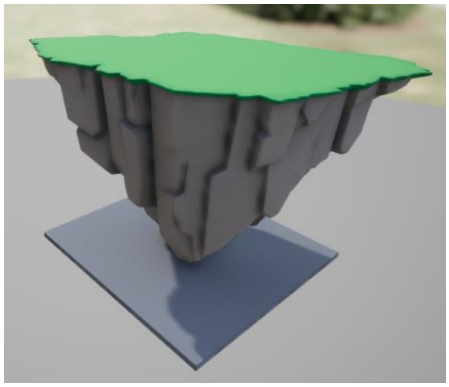


Figure 4.2.31 - Island with Materials

Figure 4.2.32 - Floating Islands

4.3 Lighting & Atmosphere

It is required to include a light source within the project for these elements to be seen. The main lighting in the scene, in this example the sun, should be represented by a light source. The light source element in this application is a directional light class object. To create a direction light, go to the classes panel on the left side of the editor, select the lights tab, and then select a direction light actor from the panel. The classes panel contains all the prebuilt actors and classes in UE. In the world outliner panel on the right, rename this item to a light source. The editor's world outliner panel displays all of the things that are currently present within the specified level.

By dragging this light source item onto the map and selecting it, you can now customize the light source's attributes. Light, Distance Field Shadows & Cascaded Shadow Maps, and other attributes should be tweaked to achieve the desired effect. should be tweaked to achieve the desired effect. For all the maps in this application, the light attribute is set to 5, the light colour is the default, the source angle is "0.5357," and the temperature is set to 5500.0. The

rest of the variables were left alone. Distance Field Shadows are used to compute efficient area shadowing from dynamic light sources when shadowing from movable light sources is provided via object Distance Fields for each rigid mesh. The default value in this attribute does not need to be altered, but the distance field shadows box must be selected. Cascaded Shadow Maps is a well-known technique for reducing aliasing by giving better quality depth textures near the observer and lower resolution far away. Aliasing is the visible stair-stepping of edges in a picture caused by low resolution. The smoothing of jagged edges in digital images by averaging the colours of pixels at a border is known as anti-aliasing. Most of the parameters in the cascaded shadow maps attribute can be left at their default values, but the Transition Fraction is set to 0.1 to limit the presence of shadow aliasing when the player moves about quickly. Figure 4.3.33 shows how the environment appears after adding a light source.

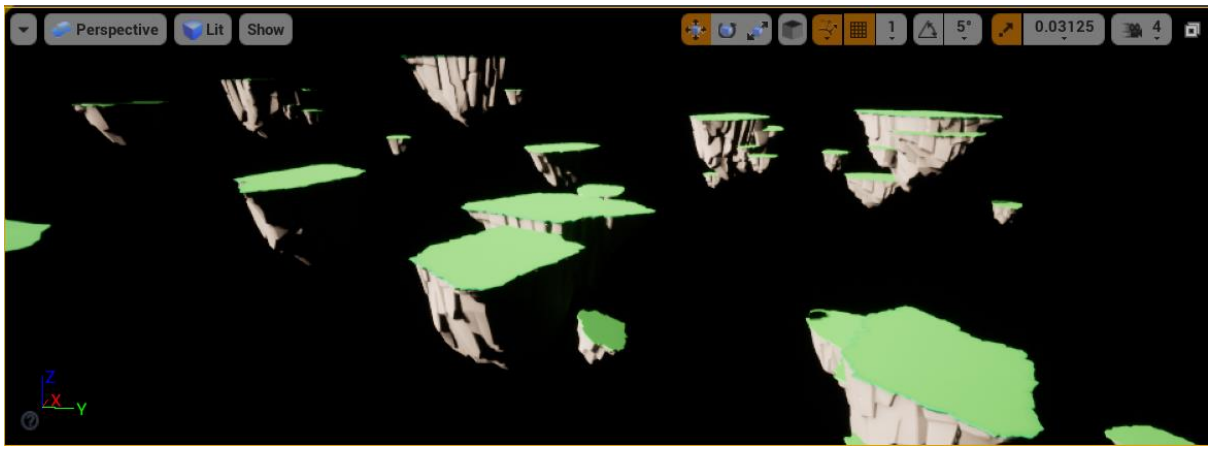


Figure 4.3.33 - Islands with Light Source

As seen in Figure 4.3.33, because the light source is solely from the light source, only half of the elements are lit up; to solve this problem, a skylight actor must be added to the map. The Sky Light gathers distant elements of your level and applies them as a light to the scene. That implies the sky's appearance and lighting/reflections will be consistent. In this example, it will add a weak light to the darkened sides of objects cast by sunlight, allowing the player to see all sides of any object from any perspective. As seen in Figure 4.3.33, because the light source is solely from the light source, only half of the elements are lit up; to solve this problem, a skylight actor must be added to the map. A skylight actor can only exist after a direction light has been placed to prevent it from being used as the main light source. The Sky Light gathers distant elements of your level and applies them as a light to the scene. That implies the sky's appearance and lighting/reflections will be consistent. In this example, it

will add a weak light to the darkened sides of objects cast by sunlight, allowing the player to see all sides of any object from any perspective. Using the search box in the classes panel, you can locate a skylight object. Once the object is added to the map, the settings must be customized to produce the best results, however, because this is an interactive map, there is an option within the light attribute of the object called "Source Type." This source type automatically updates the skylight settings as the game runs, which is advantageous because it decreases the processing power required by the system running the application. To accomplish this, the "SLS Captured Scene" option from the source type dropdown menu must be selected.

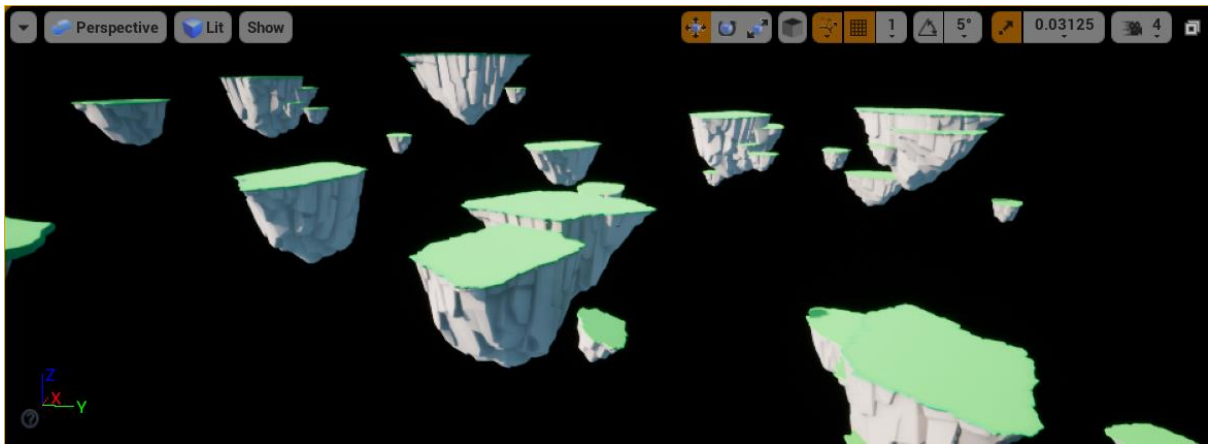


Figure 4.3.34 - Light Source & Sky Light

The next component to add to the map's realism is a sky atmosphere object. The Sky Atmosphere component is a way of physically simulating the sky and atmosphere. It's adaptable enough to generate an Earth-like atmosphere complete with sunrise and sunset. Using the sky atmosphere, the developer can set the sky characteristics to a specific time of day, for example, this entire application can operate during a sunset feel if necessary. Planet & Atmosphere - Rayleigh is the characteristic that needs to be adjusted for this object when it is placed on the maps. The planet property relates to the sphere style used to produce the map. The transform mode must be changed from "Planet Top at the Component Transform" to "Planet Top at the Component Transform," which will cause the map's curvature to exhibit a horizon line slightly beyond the eye level. This impact increases the realistic value of the application even more. Rayleigh scattering of sunlight in the Earth's atmosphere creates diffuse sky radiation, which is responsible for the blue colour of the daylight and twilight sky. Rayleigh was enabled in this application to help create a peaceful ambient vibe. Figure 4.3.35 depicts the level with all the pieces that have been implemented thus far.

Figure 4.3.35 - Environment with Sky Atmosphere

The next two parts work together to produce a complete atmospheric impression; the clouds, and fog, form an environment for the map that appears to be alive, much like the earth's ecosystem. A component called Exponential Height Fog is used to add fog to the map. To add fog, a sky atmosphere must first be built because the fog object cannot exist without knowing the limitations of the sky atmosphere object. After selecting the fog object in the classes panel, it is created. The fogging effect will be visible immediately, but because the weather for this application is sunny, not much fog is necessary, just enough to compliment the clouds. As a result, the fog object's following attributes must be changed: Falloff in Fog Density and Fog Height. The fog density, as the name suggests, regulates the density of the fog, and because the application only requires a little quantity, the value is set to "0.01," and the height falloff controls how the density of the fog increases as the height decreases. This value has been set to 0.2. This gives the program the appearance of floating with the clouds.

It is necessary to produce clouds for the floating with the clouds to make sense. Because creating clouds involves experience and power to render and develop, the cloud actors in the project were imported from a marketplace asset called "Good Sky" made by Uneasy Games. This package includes several cloud sets as well as a property that animates cloud movement in a light breeze. After importing the cloud package into the editor, the application must look natural by picking the appropriate sets of clouds. After determining the optimal set, the kill distance variable in the override parameters must be set to the appropriate value for the best range of motion. The kill distance for the clouds in this application was set to 5000 such that the clouds moved quickly enough to see a difference in short periods but not while looking straight at the clouds. After creating one cloud, duplicate it several times, and then scale or rotate it. to generate a range of clouds that can be placed in various regions of the maps

Figure 4.36 depicts the environment after the fog and clouds parts have been inserted. These produced elements have provided a sufficient environment for the rest of the level to be built on. The next elements developed are those that the character will see and experience physically.

Figure 4.3.36 - Environment with Fog & Clouds

4.4 Abandoned Temple

The abandoned temple is the key visual element with which the character will engage. The temple is made up of separate components such as a door frame, windows, walls, floors, and a roof. Each of the parts listed above has its wire mesh. A package was downloaded from the UE marketplace to obtain an abandoned medieval style of architecture. Dongli3D's Modular Medieval Fantasy includes all the necessary materials and components. The temple can begin construction once the package has been downloaded and put into the project. Starting with the floor and allocating the floor that best fits the concept of the application is required. The floor chosen for this application was called "Kit Floor." Because the floor was already associated with a material that was extremely appropriate to the concept of this application, the only customization required for the floor element was scaling and navigating the floor to position on the surface of the floating island. When several floors are placed on the surface of the floating island, a foundation for the temple is formed as shown in Figure 4.3.39. Since the foundation has been laid, the next phase is to build the temple's walls. Keeping the temple's appearance consistent throughout is essential, thus a wall structure from the assets package,

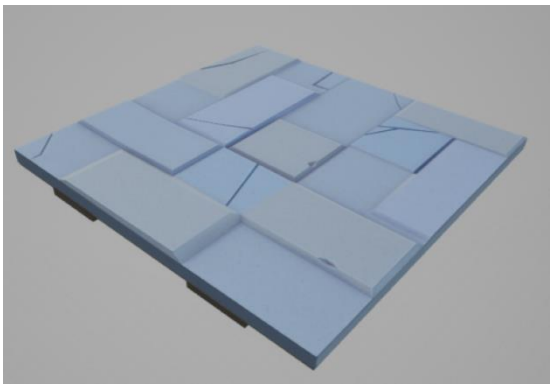


Figure 4.4.37 - Kit Floor

"Kit Wall Straight" was picked for this application, and the wall material matched the material of the floor. This wall is utilized to build the temple's walls as well as the temple compound's walls by replicating and then rotating these walls. Figure 4.4.39 shows the walls & floors of the temple.

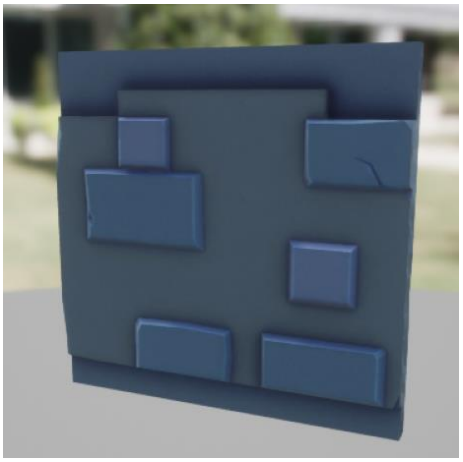


Figure 4.4.38 - Kit Wall Straight

The door and window frames are the next features of the temple. As previously done for the temple's floor and walls. The assets package is used to pick the doors and window frames. The package's door frame is designated "Kit Wall Door" for this application, and the window frame is called "Kit Wall Window." Windows are employed to create the illusion of natural light entering the sanctuary. The natural light that enters the temple will cast shadows within it. This is a visually appealing sight to see within virtual reality. Figure 4.4.40 depicts the temple following the installation of the door and window frame.

Figure 4.4.39 - Wall & Floors of Temple

Figure 4.4.40 - Doors & Windows of Temple

The temple roof is the final component that completes the temple building. This is also imported from the assets package, which is where the remainder of the temple was built. The kit includes three different types of roof designs that, when joined, form a complete roof. Corner IN, Corner OUT, and Roof Straight are the structures. Create the roof structures shown in Figure 4.4.44 by copying these three roof structures and rotating them as needed.

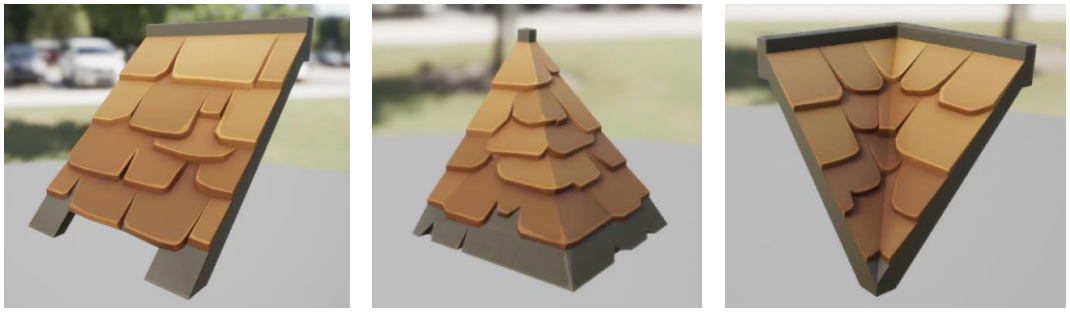


Figure 4.4.41 - Roof Straight, Corner OUT & Corner IN

Figure 4.4.42 - Temple with Roof Structure

Because the island where the temple was built was slightly higher than the island where the character began. The character needed a way to get to the elevated island. There were steps designed in the temple assets package that could be used if the temple was multistorey, but for this application, the staircase was constructed as the route for the character to get to the temple from one floating island to the other. Figure 4.4.43 depicts this stair design.

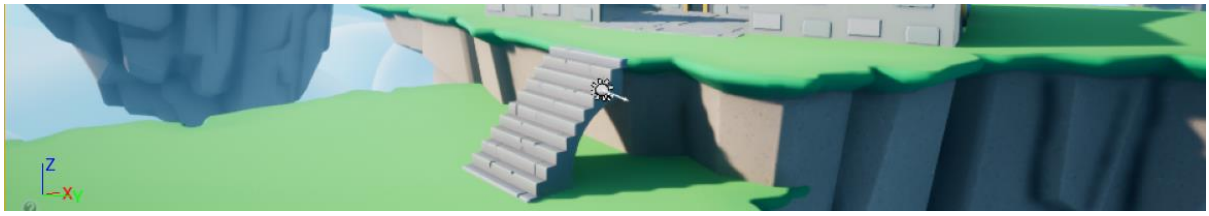


Figure 4.4.43 - Staircase to Temple

4.5 Forcefield

The gate of the temple completes the temple's appearance; in this example, the gate is represented by a forcefield. Because the forcefield is simply created with a static mesh actor and a blocking volume object. It would appear too plain in comparison to the remainder of the temple. Fortunately, the asset package for the temple architecture included a gate frame, so the temple gate could be built by using this frame and combining the forcefield and blocking volume. The gate wall frame and the gate frame itself are both contained within the forcefield frame. The materials assigned matched with the rest of the temple details because these two elements were picked and imported from the asset package. The wall frame utilized in this application is named "Large Gate Frame," while the gate frame is named "Large Gate." Figure 4.5.44 depicts both parts independently.

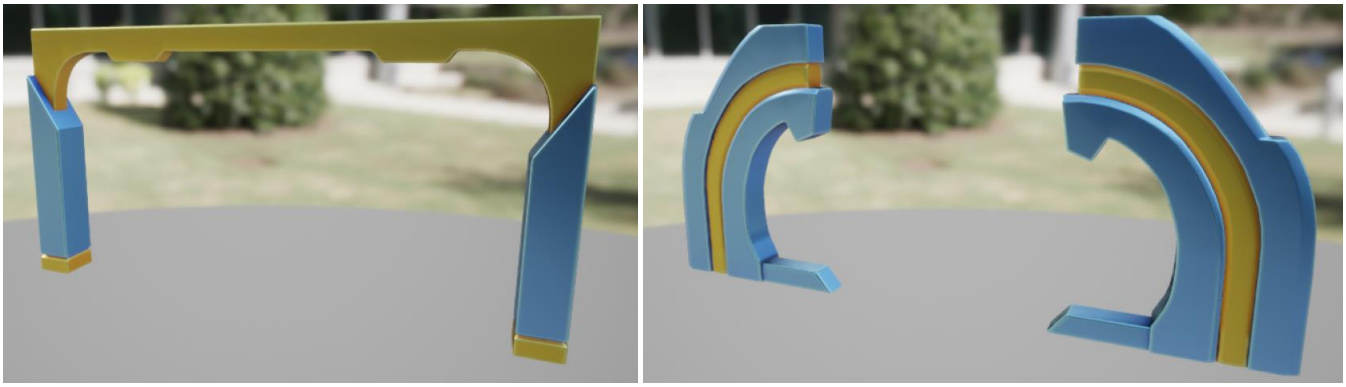


Figure 4.5.44 - Gate Frame & Gate

To achieve the appearance of the gate, the physical appearance of a forcefield was required. Satchel Quinn's "Sci-Fi Barrier Toolkit" library of forcefields was discovered after searching the marketplace. The twin pulse barrier included in this package best complemented the application's theme; nonetheless, the barrier needed to be customized to match the appearance of the rest of the gate as well as the temple. This was accomplished by reducing the oscillation variable of the forcefield and changing the material's colour property to match the blue accent from the gate frame. After making these adjustments and saving the material change, the forcefield matched the feel of the rest of the application. The forcefield's physical appearance was created by these three elements. The next stage was to design an element that prevented the player from entering the area. This effect is accomplished by employing a blocking volume actor. As long as the blocking was in place, the character would be unable to enter a specific area, in this case, the temple's courtyard. So the forcefield needs a blueprint to verify if the map's input methods were triggered in the correct order, and if they were,

destruct the blocking volume. The blocking volume actor can be located by using the search box in the classes panel. Before building a blueprint, the transform attribute is altered after it has been found and imported to the map. It was necessary to convert the object's mobility from stationary to static in the forcefield's transform attribute. After that, the blocking volume should be adjusted to cover the entire gate frame as well as a few above the gate frame to prevent the character from simply leaping over. Figure 4.5.45 depicts how this might appear in the editor.

Figure 4.5.45 - Forcefield with Blocking Volume

The following step is to construct a blueprint. The blueprint for the previously built forcefield element is being created in this application. To build an element's blueprint. Select the element from the global outliner panel, and in the details panel, there should be a blue button that indicates "Blueprint/Add Script." By pressing this, the element will be turned into a blueprint class, and the developer will be able to begin adding functions to it. Two functions must be programmed for the forcefield. When the game is launched, a function should run, and another function is an update function that checks to see if the statement’s value with the function has changed, and if so, execute the function again.

Starting with the Event BeingPlay function, this function accepts two variables, in this instance the relevant map's input method. In map 1, for example, this function would take in the two levers. To add the following actions to the blueprint, simply right-click on the editor's grid space, and these actions can be searched by name. This variable is cast to that element's blueprint to determine whether the activated function has been triggered via a Boolean variable that turns true if the lever has been triggered. Because each logic gate has two input methods, this process is repeated for the second lever. If the logic operator powered Boolean produces a true statement, the forcefield is hidden in-game by triggering the element's

visibility attribute. The forcefield element does not need to be provided in the function because it is running the blueprint. The action "set hidden in game" will choose the called element. It is the forcefield in this blueprint. The update function follows the same procedure, except it accepts an additional variable and performs an additional operation. The extra variable it takes in is the blocking volume that was constructed, and the extra action it has added is that when the correct logic sequence is triggered after hiding the forcefield in the game, the extra action deletes the blocking volume. As a result, the character can pass through the gate and enter the temple. The input methods in the maps, whether levers or switches, call the update function. When one of the inputs is activated, the update function of the appropriate forcefield is called, and the process described above is followed.

Figure 4.5.46 - Forcefield Blueprint

Figure 4.5.46 depicts the two functions present in the blueprint editor for map 2 – OR Gate. To incorporate the previously described input methods and forcefield into the blueprints There were custom variables created. It is remarkably simple to create a custom variable in blueprints. Navigate within the blueprint editor to my blueprint panel on the left side of the editor, and there should be a row named variables within that panel. To add a custom variable, simply press the plus symbol next to the name and a variable is formed; now rename the variable to something more appropriate. By clicking on the variable in the list, the editor now allows each variable to be modified in the details panel for a variety of reasons. The variable type is the most important setting to modify. In this application, the variable types for the input methods should be set to an actor, and there is a blocking volume option within

the variable type for the blocking volume. By selecting these types, it prevents other items from being passed and only allows components that match the variable type's description to be passed. The variable selection will appear in the map editor's attribute list of the element, and elements can be provided into the blueprint via this.

4.6 Instructions

A most essential element of this design was developing instructions to be displayed to the player when the application was launched. Because the playerstart element has not yet been deployed. A location for the instructions is visible from the rough sketches of the map. The instructions must be placed on a surface to be visible. A custom surface must be utilized to do this, but there is currently a specific wall-sized surface included with the project from the temple components package. The wire mesh from that wall can be reused, and a custom material can be added to the wire mesh to create a custom wall on which to print the instructions. To reuse that wall, simply locate it in the content browser and drag it onto the map; once this is completed, modify the material of the wall so it does not resemble the temple wall. The material utilized for this project was a material prebuilt into the Unreal Editor, which can be found in the content browser by searching "BasicShapeMaterial." After it has been applied, this wall can be replicated and scaled up or down to create various surfaces for displaying instructions or anything else. Figure 4.4.47 depicts how these customized walls appear on the map when the material has been modified.

The next step is to add text or photos to the wall's surface. A text render object should be utilized to add text to the map. This object can be found using the search function in the classes panel. When this object is dragged in, text can be added to the object's text attribute. The text attribute also lets you modify the font, alignment, colour, and other aspects of the text to make it completely customizable. Once modified, the text only needs to be put slightly in front of the wall surface, creating the appearance that it is printed on the wall, but the text render is simply overlaying the surface of the wall. Figure 4.4.47 depicts the appearance of the text as it is superimposed on top of the wall.

To include graphics in the maps, simply import them into the UE via the content browser. Image file formats such as jpg or png are preferred. Once imported, right-click on the graphic in the content browser and select "make material" from the texture operations menu. This will generate a new file containing the graphic but in element material format. After you've created this material file, double-click it to open it in a material editor. When the editor is

opened, look for the detail panel on the left side. There should be a material property in the details panel, and a few of the variables should be modified for the greatest quality. The surface should be converted to a deferred decal in the material domain variable. This will result in an error indicating that the blend mode should also be adjusted. The second variable that must be changed is the blend mode from opaque to translucent, followed by the decal blend mode from translucent to DBuffer translucent colour. If the image has a black or clear background, the texture sample's alpha node should be linked to the graphic material's opacity node. This will remove the background when the decal object is displayed. A decal object is required to display the image inside the environment. The search box in the classes panel can be used to locate a decal object. Once the decal object has been imported into the map, go to the details panel, and choose the material file that was made with the graphic in the material attribute. The graphic will be updated and displayed by the decal object. Now position and rotate the graphic according to what should appear to the characters. Figure 4.6.47 illustrates how the decal appears on the previously constructed custom wall.

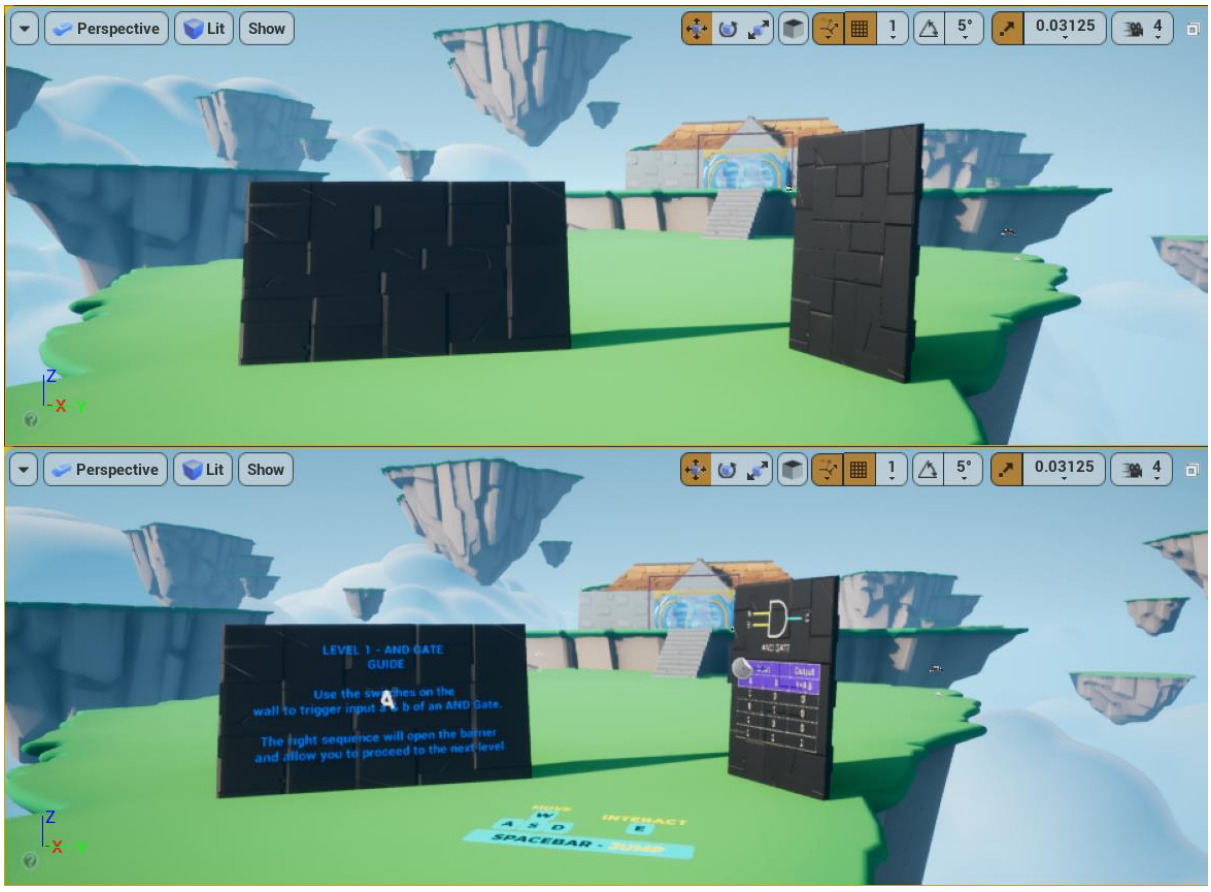


Figure 4.6.47 - Text & Decals

4.7 Aesthetic Elements

The next phase in design implementation is to add the minutiae to the environment before creating and implementing the interactive parts. These aspects are included to make the map visually appealing rather than bland and uninspiring, as well as to produce a genuine look to the environment; two elements are added to this application to do this. A post-process volume and rock formations The same basic wire mesh used for the floating island was utilized to make the rock formations, however, it was turned 180 degrees so that the peaked section of the formation faced the sky. After that, the mesh was scaled down to a smaller size and distorted to generate a variety of sized rock formations.

The post-process volume was the next element added to the environment. Post-processing is used to allow developers to fine-tune the overall look and feel of the scene. Camera lens, luts, bloom, ambient occlusion, and tone mapping are examples of elements and effects. The search box in the classes panel can be used to locate a post process volume. This object can be placed anywhere on the map because its influence applies to the entire map. Because this is a VR application, the developer may not be able to alter a few lens qualities; however, if this application was made in the third person, the lens attributes will allow the developer to produce a specific look by using specific lenses and bokeh compression, among other things. There is no specific attribute for this object that should be enabled, deactivated, or altered; instead, the attributes of this object can be customized to the developer's preferences. Bloom, Exposure, Depth of Field, and Colour Grading are some of the most significant qualities to consider in this object. The Bloom effect adds light fringes to the edges of bright regions in an image. Exposure is the amount of light that reaches your camera's sensor, or in this instance, the character's eyes. Depth of Field blurs a scene based on how far it is in front of or behind a focal point. The depth effect is used to direct the viewer's attention to a certain subject of the shot. Colour grading is the process of changing or correcting the final image's colour and brightness, similar to applying a filter to the final image.

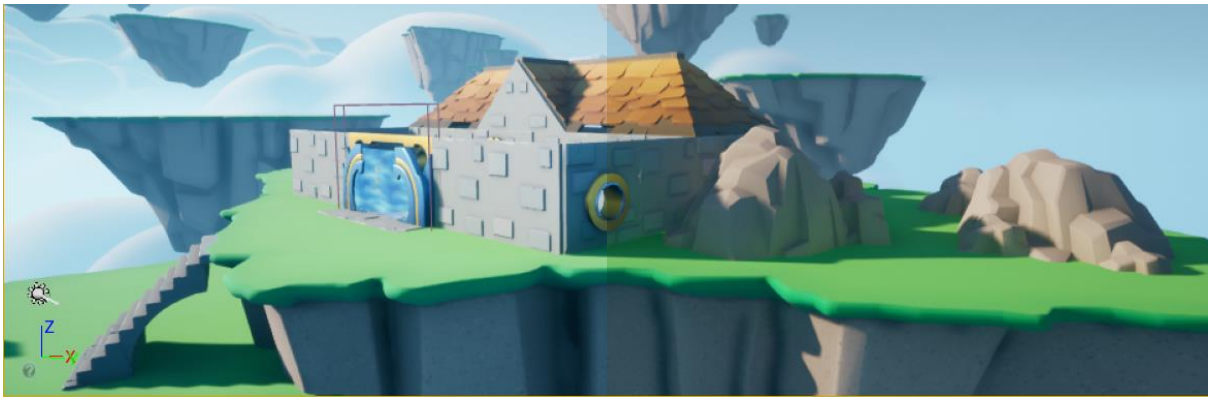


Figure 4.7.48 - Before & After with Rocks Formation & Post Process

4.8 Interactive Elements

The levers and switches are the next pieces to be generated during the implementation stage. Each input method consists of three parts: the base, the movable, and the blueprint. The base & the movable of the lever in this application was imported from a marketplace assets package. Kristy Bores' Buttons, Levers, Knobs package included a variety of buttons and levers that can be customized to meet the concept of this application. After importing the package into UE and picking the most appropriate base from the "lever base" list, it is time to alter the base to meet the theme of this project. By double-clicking the base's static mesh file in the content browser. This will open the architecture tab and allow users to customize the base. The lever base already matched the rest of the environment in this project, but the colours of the base needed to be adjusted to match the rest of the theme, so the base colour was matched to the accents of the temple door frame to maintain a consistent aesthetic throughout the project. The movable handle that revolved along the axis of the base piece when triggered was the next component added to the lever. The lever "Lever Handle" was likewise chosen from the previously mentioned assets bundle. The handle also needed to be tailored to fit the colours of the temple, thus it was given a golden accent. Figure 4.8.49 depicts the appearance of the lever on a customized wall. After customizing and saving both items. Another file in the package exists that automatically connected these two sections to form the lever as a whole, and this is the file to which the preset blueprint is attached. This file is known as "Lever BP" in this project. Insert this element into the map and use the blueprint editor to define the actions that will take place when the character pulls the lever.

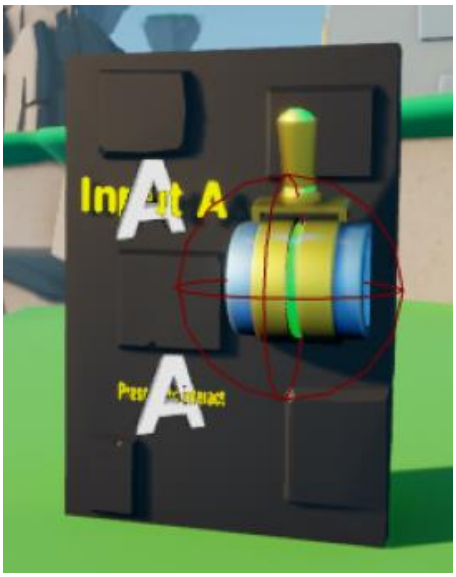


Figure 4.8.49 - Implemented Lever

When the blueprint is loaded, a series of predefined actions are displayed. None of these actions must be changed because they govern the interaction between the character and the lever as well as the animation of the handle, but a few more actions must be added to the blueprint when triggered. This lever blueprint activates the previously stated update function in the forcefield blueprint. The status light and a gate status light, like a forcefield, also have update functions; these functions are explained further in their respective sections. To activate these update routines in the blueprint, a few actions must be included following the

switch action. When the switch action triggers the "activated" node, the blueprint uses a "get all actors of class" action to select actors of a specific class, such as the lever status light class. This action is then attached to a for each loop action, which goes through each lever status light actor and then triggers the status light update function if it exists for that actor. This procedure must be carried out again for the gate status light and the forcefield actors. So, if a lever is activated, the blueprint performs the prior operation, which triggers the update functions in the other blueprints. Figure 4.8.50 depicts the blueprint of the lever class with close-ups of additional actions that were implemented.

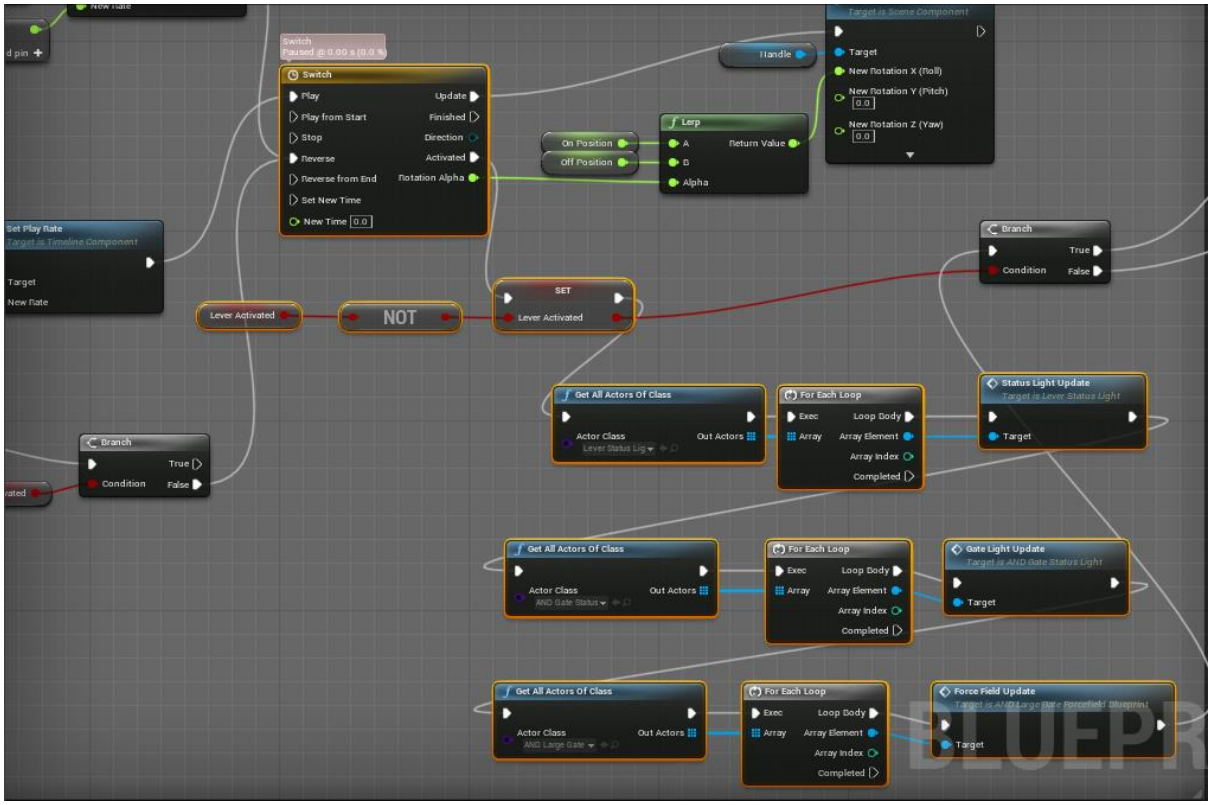


Figure 4.8.50 - Lever BP - Extra Actions Highlighted

The switch is the second type of input technique. Although the switch has a different blueprint than the lever, the method is the same. The switch includes three components as well: the base, the button, and the blueprint. After searching through the assets package where the lever was discovered, a suitable base for the switch was also discovered, and just like the lever base, the switch base fit the theme of the application by customizing the base material to match the colour of the temple design. The base element in the package is called "button_ring". Next, a button is needed to be connected to the base, and one was located in the same package, but this button had a unique feature in that it had a built-in status light, eliminating the need for an external status light. In the package, this button is known as

"button_push". After selecting both the base and the button, the file containing the blueprint for the switch was located, and the selected base and button were picked in the variables to connect them to the blueprint. The blueprint's name is "playerbutton_bp". Before installing the button on the map, the blueprint must be updated to activate the gate status light and forcefield update functions. There are pre-configured actions when the blueprint loads. These activities are there to respond to the interactions of the characters. Such as the button animation of being pushed into the base or rising back up when released. An action must be placed in the blueprint's "player in volume" section for this button to activate the update functions of the forcefield and gate. Following the press button action in the blueprint, a node must be placed to the "get all actors of class" action, and the procedure that was executed for the lever blueprint must also be duplicated here, but only for the gate status light and forcefield actors, as the status light is not required. Figure 4.8.52 depicts the switch's blueprint as well as close-ups of additional operations that were implemented.

Figure 4.8.51 - Triggered Switch

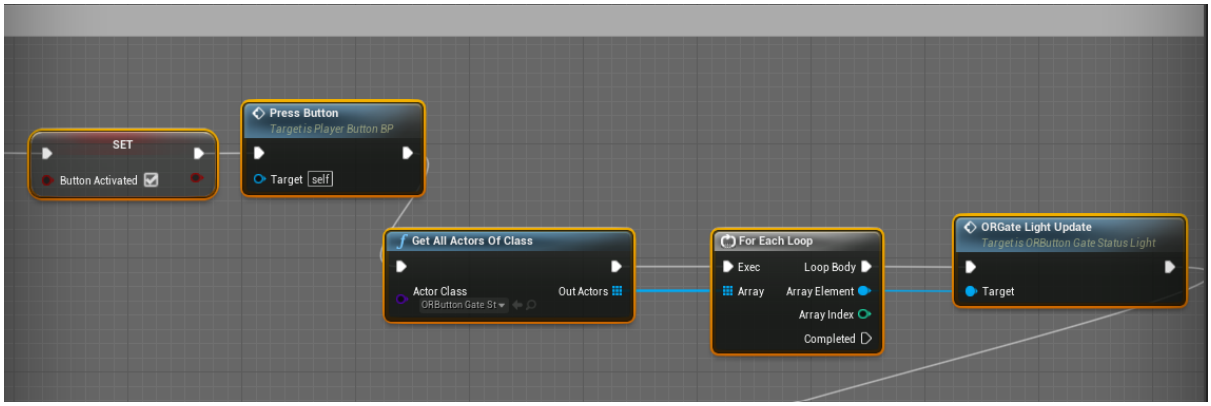


Figure 4.8.52 - Switch BP – Extra Actions Highlighted

4.9 Computing Functions

The status lights, teleporting function, and player start object are the next design elements to be implemented. The status lights serve as visual cues to the character that something has been accomplished. There are two sorts of status lights: conventional status lights that indicate whether or not the lever has been triggered, and gate status lights that indicate whether or not the logic sequence entered was accurate and the forcefield has been disengaged, allowing the player to enter the temple. A point light is used to implement status lights. Point Lights function similarly to real-world light bulbs, emitting light in all directions. A point light actor can be found in the lights section of the classes panel. To light up when a lever is pulled, this actor must be turned into a blueprint. For this application, the light colour variable of the point light must be modified before creating a blueprint. The light's default colour is white, however, for this application, it was changed to green to indicate that it was successfully activated. A blueprint can be developed once that change has been made.

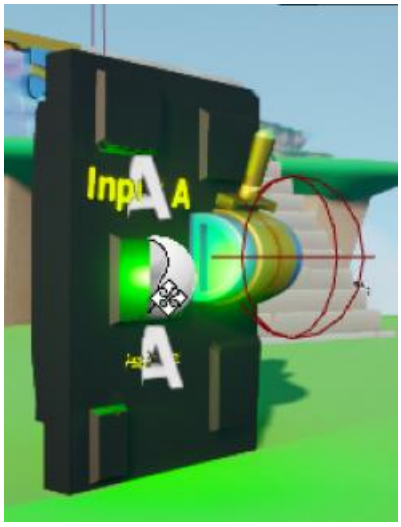


Figure 4.9.53 - Status Light

This design is similar to the forcefield blueprint in that it includes an actor variable, which in this case is the lever to which the status light will be tied. The variable is then cast to its primary blueprint, and when the lever is activated, an operation called "Set Visibility" turns on the point light, as opposed to the forcefield where the actor was hiding. This point light also includes an update function that executes the same script but is only called by the lever blueprint function. This is necessary since the first function only works when the game is first launched, whereas the update function causes the status light to activated & deactivated. Figure 4.9.54 depicts the status light's blueprint.

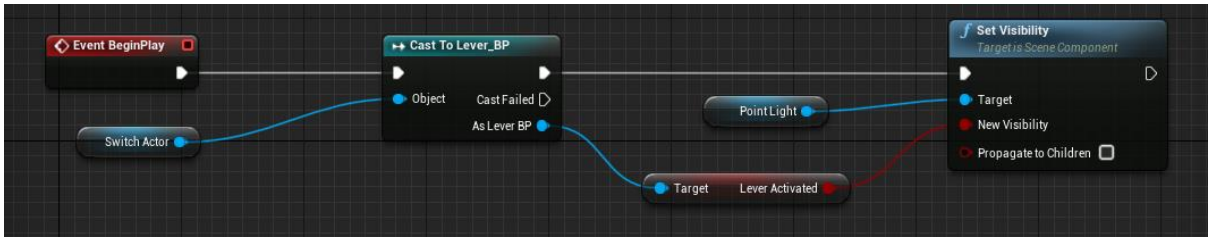


Figure 4.9.54 - Status Light BP

The gate status light operates on the same principles as the status light; it is the same, green-coloured point light as the status light; the only difference is that the blueprint is different.

Because point lights have limited coverage, they can be spread to cover a specific region by reproducing the light a few times, as seen in figure 4.9.55. The gate status light blueprint is a hybrid of the status light and the forcefield. The gate status light, like the forcefield blueprint, takes in two actors and checks if both actors have been activated and if they pass the logical operator statement required for that specific map. Instead of hiding the actor as in the forcefield, it follows the status light blueprint and turns on the light if the conditions have been met. Figure 4.9.56 depicts the gate status light's blueprint. As previously stated, the gate status light also has an update function, but it just executes the same script when triggered by a lever or switch.

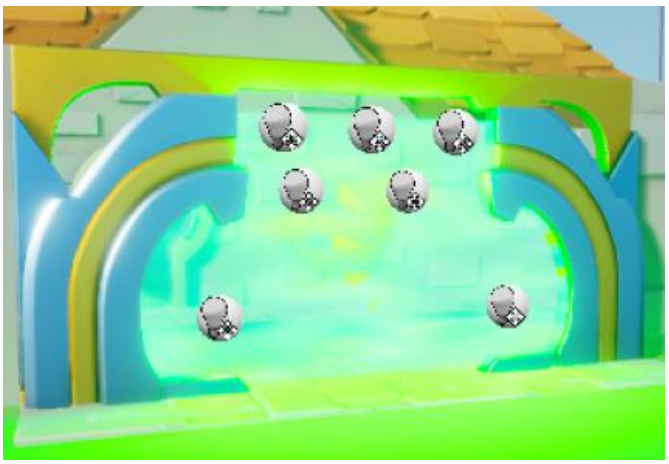


Figure 4.9.55 - Gate Status Lights

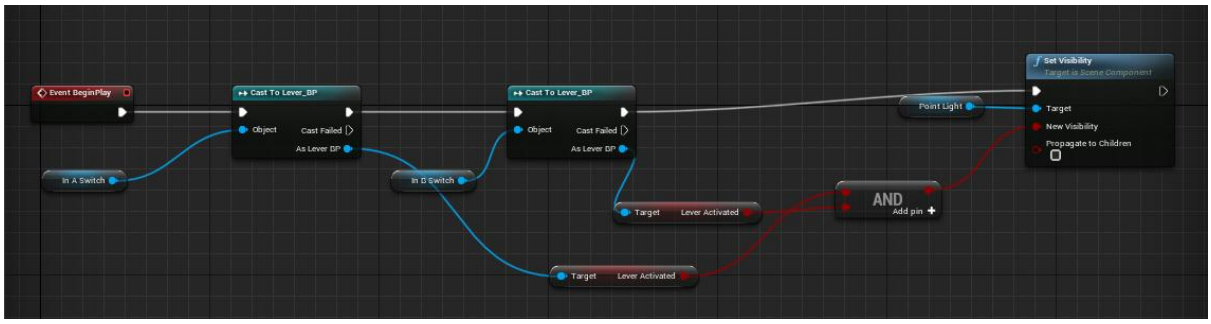


Figure 4.9.56 - Gate Status BP

The mechanism for moving the character from one map to the next is the next element to be implemented. This element is known as the teleporter in the project. The teleporter is made up of two objects: a symbol and a box trigger with a blueprint. The symbol is merely a decoration to indicate to the player that this is a trigger to proceed to the next map. The symbol is positioned inside the temple so that once the character enters the temple's door, they are transported to the next map. An element needed to be built for the symbol, however, after browsing through the asset package for the temple, there were a few props that could be used to beautify the inside of the temple. This

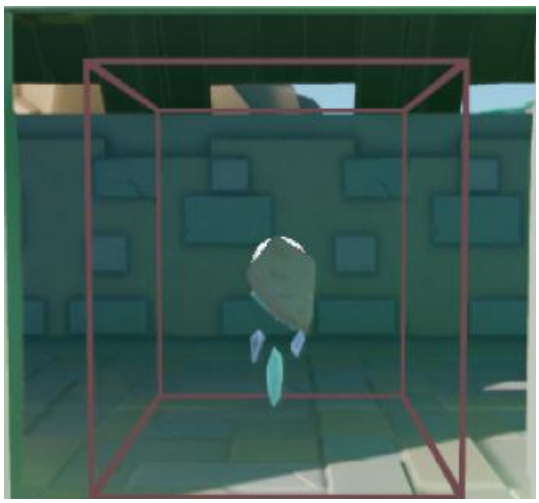


Figure 4.9.57 - Teleporter Symbol & Box Trigger

prop was titled "Powerup Shards," and it seemed to fit the theme while yet being distinctive enough to stand out. The element was placed inside the temple, followed by the box trigger. Triggers are actors that cause an event to occur when they interact with another object in the level. When the box trigger detects that the character has identified the symbol, the blueprint script will execute and transport the character to the next level. The box trigger can be searched in the classes panel by using the search box; once located, a blueprint script can be applied. The script consisted of three actions. Begin Overlap on Component, Cast to Character, and Open Level (by Name). The whole script is created by linking these actions in order, then attaching another actor node from the component action to the object node on the character actor, and then selecting the level to which the character must be teleported to in the open level action. These three actions work together to see if the character has overlapped into the box trigger, and if so, the map specified in the variable is opened. Figure 4.9.58 depicts the blueprint and how the actions are linked.

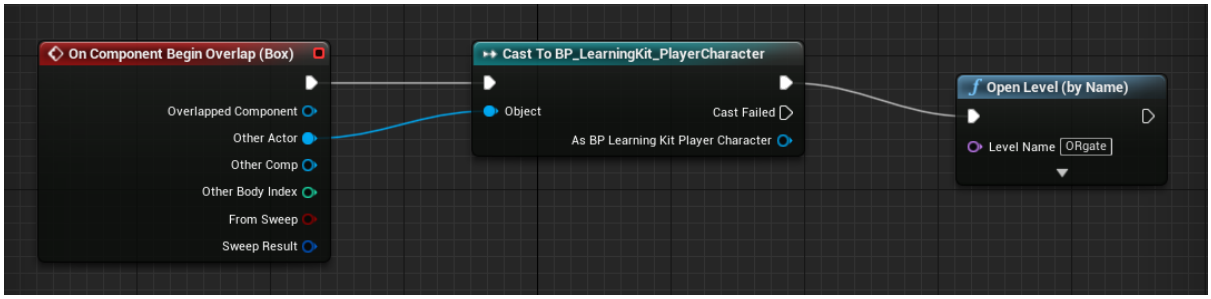


Figure 4.9.58 - Teleporter BP

The player start object is the final piece to be added before the character is introduced. This object works similar to a position marker, allowing the software to determine where the player character should begin when the application launches. The player start actor can be discovered by searching in the classes

Figure 4.9.60 - Player Start Actor

Figure 4.9.59 - Player Starting View

panel. This Actor is placed in the level to indicate where the player should begin when the game starts. A blue arrow should show on the actor symbol once it has been placed into the level. This blue arrow indicates the direction the character will face when the game begins. So, by rotating and modifying this blue arrow, you may change what items are visible to him. Figure 4.9.59 depicts the blue arrow that is being referred to, whereas Figure 4.9.60 depicts what the character in Level 1 will see.

4.10 Character

Normally, the character element in VR is controlled by a floating camera actor. This camera actor will use sensor information from VR technology such as headsets to control the camera angle that is displayed to the user. It was decided to attach another physical element to the camera object for this particular case. This physical aspect was a dummy body so that the user would have a character body in this program, and it was determined to do as follows so that animations and materials could be linked to the character body to make a more immersive application. As previously stated, each custom element in UE begins with a wire mesh; however, for a character element, a skeleton mesh is used instead of a wire mesh. Skeletal Meshes are frequently used to represent characters or other animated objects in Unreal Engine 4. The 3D models, physics, and animations are developed in external modelling and animation application such as 3DSMax, Maya, Softimage, or Blender before being imported into Unreal Engine 4 and saved into packages via the Unreal Editor's Content Browser. The character for this application was obtained from the UE marketplace as part of a package named "Wizards of Magic," which was created by Dunegon Mason. This character element came preloaded with a skeletal mesh, material, physics, and animations, as well as a blueprint to connect all of the components. A closer look at the skeletal mesh reveals that it replicates a human skeletal, with each moving joint and connection to other body parts, with mesh constructed of each moving part of this character. There are 75 distinct moving pieces in this particular character, which are all joined together by a skeleton mesh to form the character. As previously said, the character comes with pre-built animations; nevertheless, for these animations to appear, each moving component is transformed, and these movements are tracked by a motion controller and translated into keyframes for the transform attributes. When an animation is triggered, each moving component simply travels to the predetermined keyframe and then resets and repeats this action. Figure 4.10.61 includes multiple joints on the left side as well as the skeleton mesh.

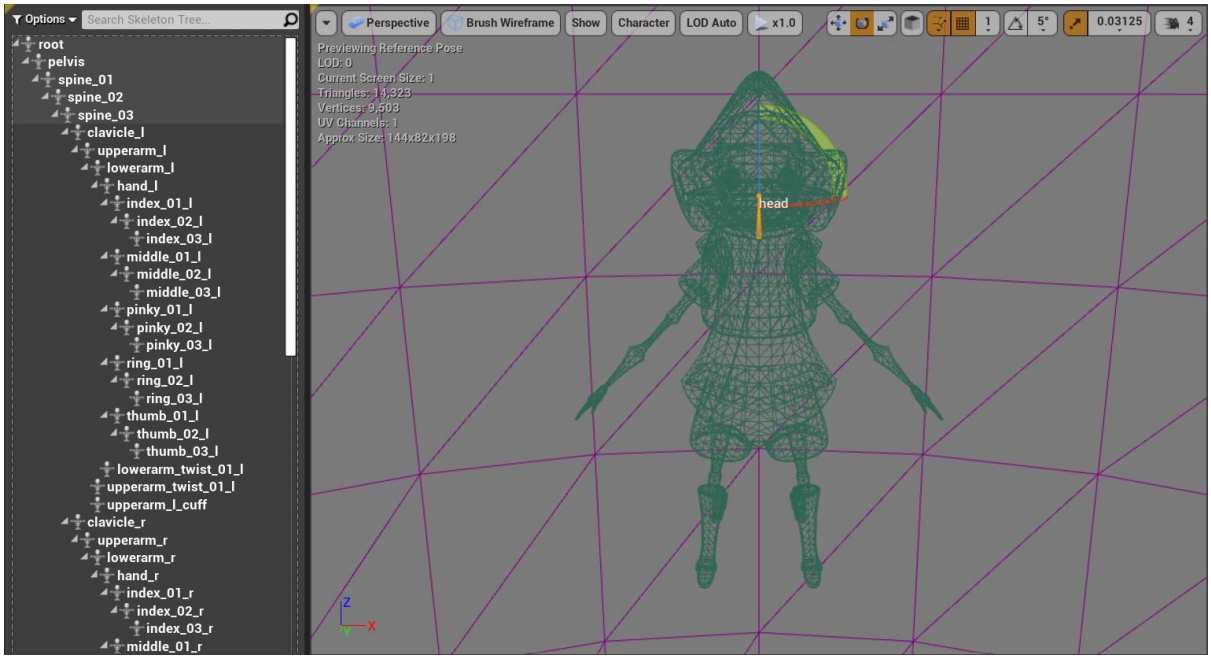


Figure 4.10.61 - Skeletal Mesh

Attaching materials to the mesh was the next step in character implementation. There were 8 sections for materials to be attached to the body of this character. Body, hood, face, shirt and shorts, arms and legs, boots, tent, and rucksack were among the sections. Although the character came pre-configured with materials for these portions, the colours of these materials were changed to match the rest of the theme. These materials are changed by opening them in the asset editor using the content browser and changing the hue or tone of the material using the colour attributes in the Details panel. Figure 4.10.62 displays each segment separately in the following order. Face, hood, body, shirt and shorts, arms and legs, boots, tent, and bag are all included.

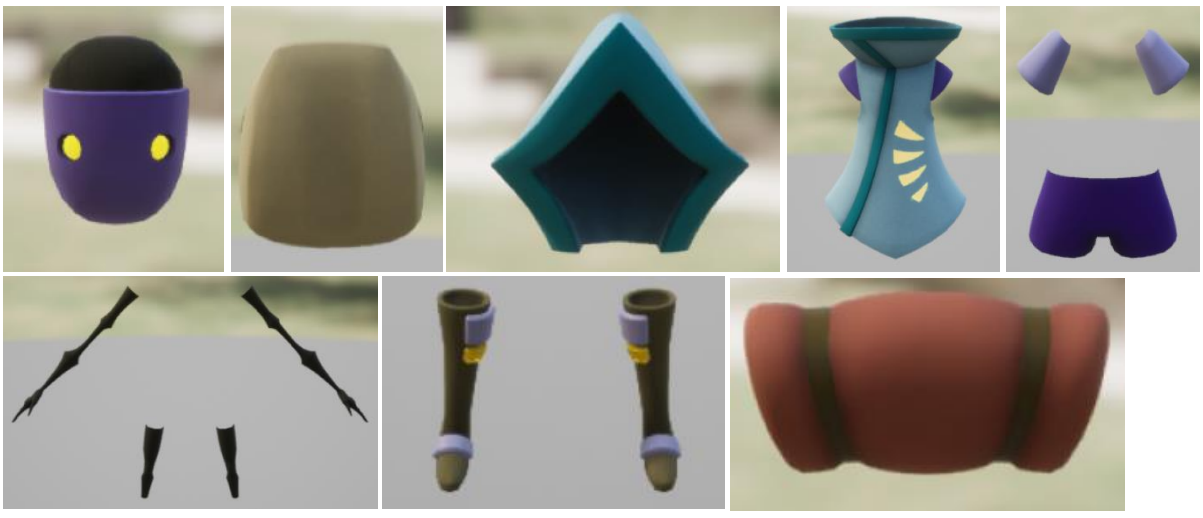


Figure 4.10.62 - Character Materials

The animation stage came next in the character implementation process. As previously stated, the character included pre-configured animation as part of the bundle. There were 34 different animations pre-built for this character. To import these character animations, they must first be imported into UE's content browser. These animations can then be linked to a specific skeleton utilizing the skeletal mesh animation system. The animations were not altered for this project. All of the animations were checked within the animation system once they were imported and linked to ensure that the right connections in the skeleton mesh were selected during the automation process.

The physics feature of the character is the next level of implementation. When the character interacts with other parts, the physics composition of the character is essential. The physics design prevents the character from stepping through walls, the character's hand from phasing through the lever when pulling it down, and so on. When testing for overlap between two items, the physics design also interacts with the box triggers. This physics also plays an important role in maintaining the character upright and preventing the character's joints from buckling. It almost works as the character's glue. In this scenario, the character came with a predefined physics design that connects all the body sections and includes gravity features. by taking a closer look at one of the physical design scripts illustrated in Figure 4.10.63. The constrictions of the head to the spine 03 element can be noticed. This indicates that, like in the human body, the head of the body is linked to the upper spine of the body. This is merely one of the character's physical designs; there are 17 other requirements with the character's body. Connections for the pelvis, spines, upper arms, lower arms, hands, head, thighs, calves, and feet were all split between the left and right sides of the body.

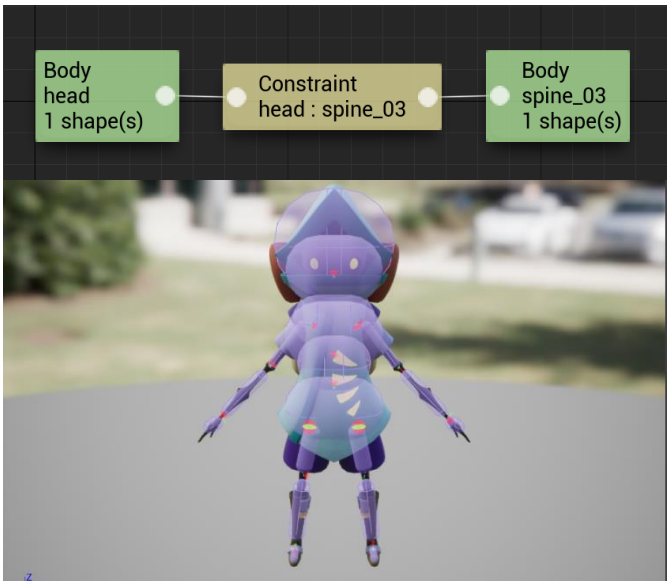


Figure 4.10.63 - Physics Design

Once all the preceding processes have been completed, the character's blueprint is created. This blueprint determines how the character behaves based on user input, as well as which animations to perform and how the interaction with other elements occurs. The character's

blueprint was imported with a pre-configured blueprint and a particular number of pre-set scripts such as Being Play: Setup, Jump, Interaction, and so on. The majority of the actions were left alone because they functioned seamlessly with the rest of the features in the environment, but a new action was designed for the implementation of the VR controller.

The design referred to this action as "Gamepad Input." Figure 64 depicts this portion of the blueprint. When the controller provides input, this action has an event that takes the current value of the TurnRightRate axis once per frame. When TurnRightRate returns a negative value, the character turns to the left. This value is combined with the base turn rate value and the frame delta time. The base turn rate is a variable that can be modified; in many applications, it is referred to as sensitivity. These three numbers are multiplied and then given to the controller's yaw input. Yaw is commonly characterized as a moving object twisting or oscillating about a vertical axis. In this game, the multiplied value controls the character's yaw, which in this case rotates the character left or right. This operation is repeated for the character's gazing up and down movement, therefore the value multiplied by the LookUpRate determines the pitch of the character. A pitch motion is an upward or downward movement of an object's snout. As previously stated, these events may be found by right-clicking within the blueprint editor and searching for these actions in the drop-down box. Please keep in mind that this search box is case sensitive.

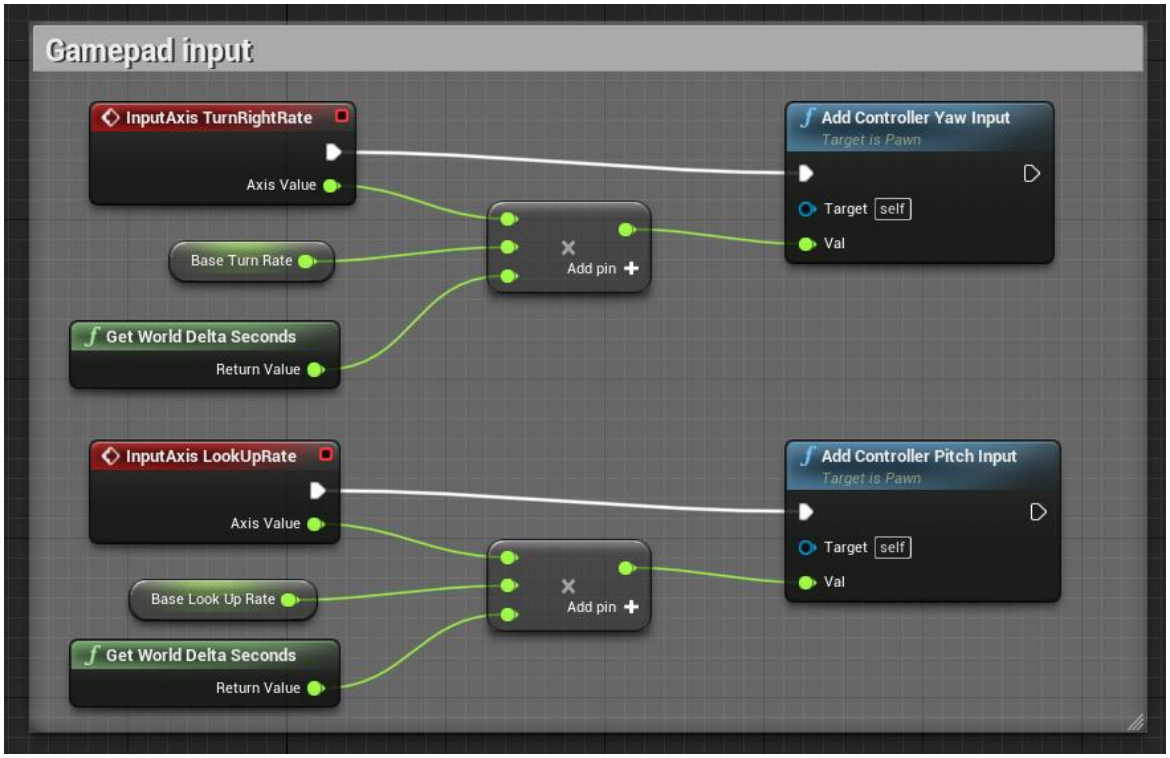


Figure 4.10.64 - Gamepad Input

5. Results

So, after completing all of the prior stages, the next step is to deploy the application and test that all of the elements function as expected. The three maps can be distinguished from the preceding phases in their own right; in this case, the final appearance of the three maps in this application is depicted in Figure 5.67. To launch an application, the levels must first be built. UE computes lighting and visibility data, generates navigation networks, and updates brush models. It also creates shaders for each element. A shader in Unreal is made up of HLSL code and the contents of a material graph. Unreal creates numerous shader permutations based on factors such as shading mode and usages when building a material. It can only be launched if all the levels have been finished. The play button in the UE toolbar can be used to run the application. Using the option box next to the play button, you can launch the application in a variety of formats. The application is launched in the UE editing viewport by default, but it can also be opened in a new editor window, as a standalone game, or with a VR preview, which is only available if a VR headset is connected to the device running the application. This application was run for testing reasons via the standalone game option, which allows the standalone window to be forced to quit via task manager if the game stalls or crashes.

Three major difficulties were discovered after running the application through several rounds of variables. The first fault was discovered within the AND gate level. The blocking volume was destroyed after only one of the levers was pushed, allowing the character to enter the temple. This was discovered to be a flaw in the blueprint. What was happening was that once the AND logic check was finished, the white connected nodes were completed even if it was false. This problem was solved by implementing a branch action in which the logical operator statement had to be true for the action to destroy the blocking volume to be carried out. Figure 5.65 depicts the original blueprint as well as the revised blueprint.

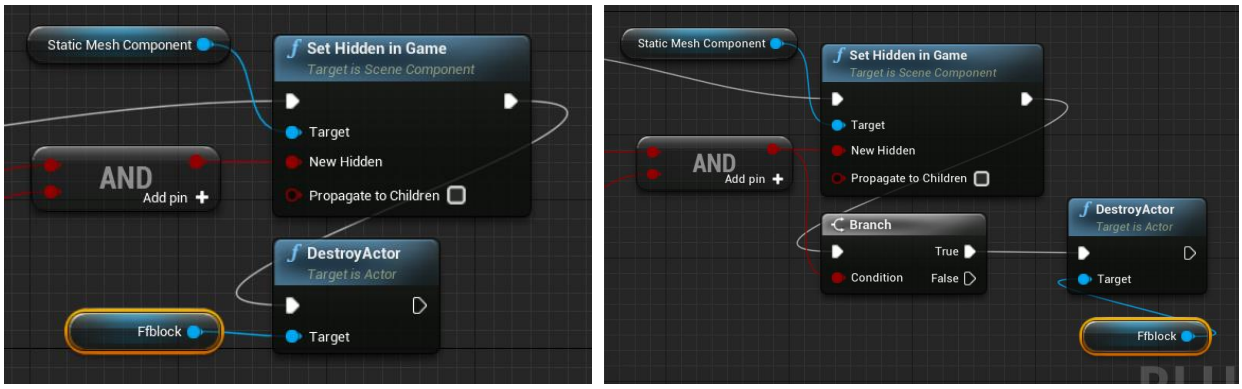


Figure 5.65 - Original & Updated Blueprint

The next issue discovered during testing was that the application was utilizing too much ram on the device that was running it, and this caused the device to lag as well as raise the CPU temps as soon as the application was loaded. To combat this, Level 1 of this application was used as a test subject, and each element in the level 1 map was destroyed before running the application to identify which element was causing the ram issue, which turned out to be the character object. As a result, the various stages of the character were disabled, and the application was tested. So, when the animation stage of the character was implemented, the application's ram usage increased, and the fault was traced back to the animation stage of the character. Then it was shown that even though the character's 34 pre-configured animations were not used. Because all animations were being rendered and shaded every time the application was launched, the ram usage was quite high. To remedy this, simply link the animations that were utilized to the character and leave all the unused animations in the content browser to be linked at a later stage if necessary.

The last issue discovered when testing the application was at level 3 - the NOR gate level. The issue was particularly noticeable on the OR house levers; when the character triggered one of the levers, the second lever was also triggered. The problem was initially assumed to be a blueprint issue, however, even after many adjustments and recreations of the levers, the problem persisted. Then it was supposed to be the creation of the lever object when they were replicated and some hidden link was made, but that wasn't the case either. The actual issue was that the levers were too close together. So, while the character was in the centre of the two switches, attempting to engage with one of them. Both switches were being activated in the same command, therefore being triggered at the same time, and the repair was to just increase some space between the levers, and the problem was solved. Figure 5.66 depicts the switches' original positions as well as the fixed space between them.

Figure 5.66 - Lever Positions

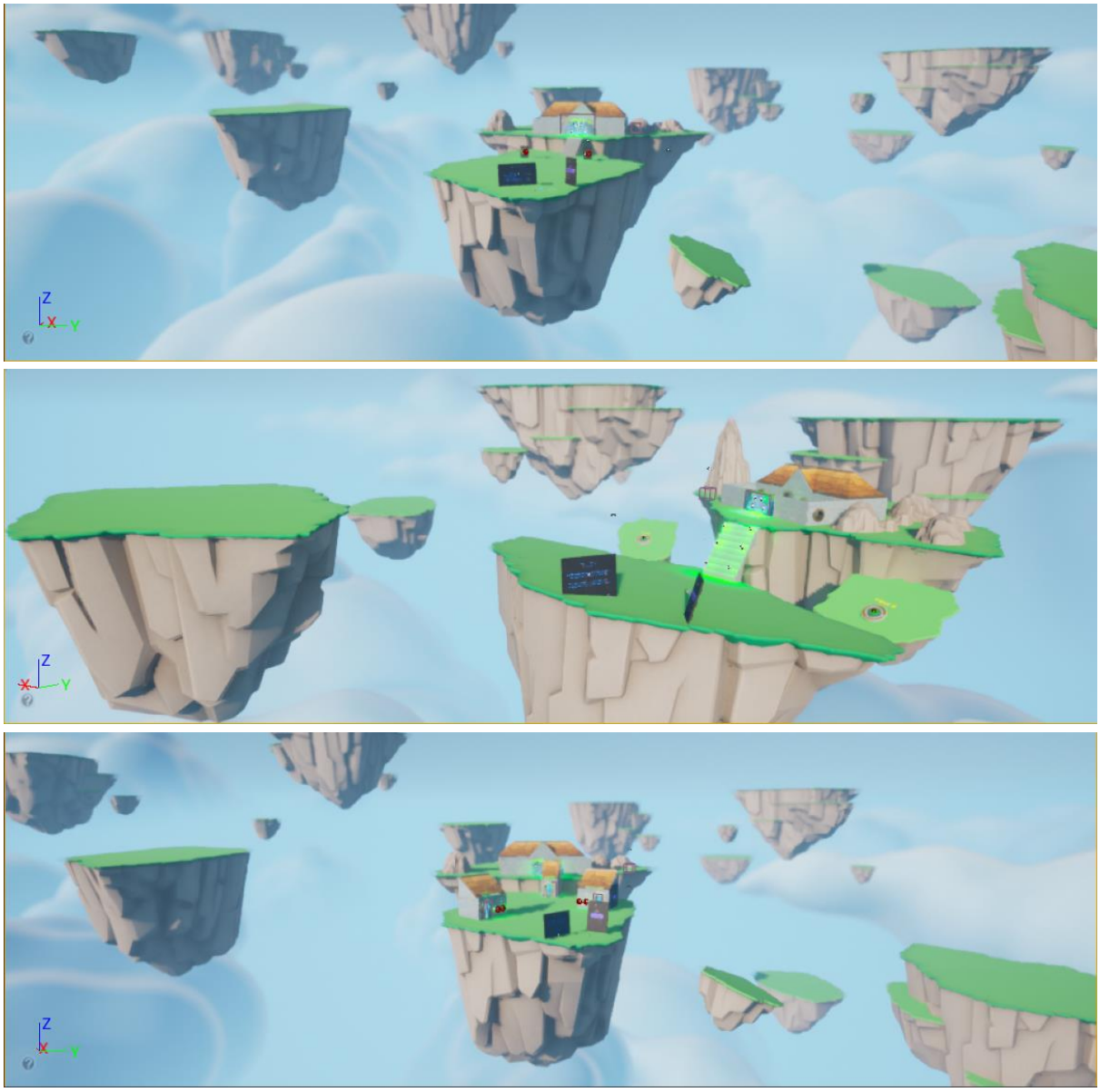


Figure 5.67 - Map After Implementation

6. Conclusion

Overall, the application built for this project achieves the goal of assisting first-year engineering students in learning the principles of logic gates through VR and the primary three learning styles. This application can also be made available in a mobile version for these pupils to utilize as a learning tool at home. The most difficult part was constructing the application's basic components, but once these foundations were in place, the rest of the application's development went relatively smoothly. If a longer period had been available for this project, it would have been utilized to develop levels that covered all the logic gates, followed by a final level with a vast map that covered all of the logic gates, allowing the game to end.

Instead of having a character spawn randomly on a map, a tale would have been constructed for the character in the game to make it more appealing to the user. Another feature that may have been created is the ability for students to create custom maps by dragging and dropping pre-made structures and pieces into an environment. Students can also use pre-set blueprints to add logical functionality to custom levels, which they can then share with other students to test out. This allows them to experience and wonder which logical gate to use in their custom map, giving them a better understanding, and students who decide to play the custom map get to immerse themselves even more in VR. Another VR future objective is to develop another application that will allow students to build circuits on breadboards with various components, just like they would in a typical lab environment. This might greatly assist students in predesigning circuits in VR and testing out multiple configurations before physically creating the board. This allows students to make numerous mistakes and then create or design an actual circuit board and have it manufactured once they have a working solution.

This project has also helped me understand a lot more about VR and how the VR space works, which has been especially useful with the rise of the Metaverse and virtual space, and this project served as an introduction to the Web 3.0 area. The project assisted with several non-technical difficulties, such as time management and how to operate with limited resources. So, not only did the project achieve its goal of educating first-year engineering students about logic gates, but the application also has a plan and numerous other development possibilities.

Bibliography

ARM, n.d. *What is a Gaming or Game Engine?.* [Online]
Available at: https://www.arm.com/glossary/gaming-engines
[Accessed Monday December 2021].

Bardi, J., 2019. *What is Virtual Reality? [Definition and Examples].* [Online]
Available at: https://www.marxentlabs.com/what-is-virtual-reality/
[Accessed 29 December 2021].

Cassidy, S., 2004. Learning Styles: An overview of theories, models, and measures. *Educational Psychology,* 24(4), pp. 419-444.

Cruz-Neira, C., 1992. Audio Visual Experience Automatic Virtual Environment. In: *Communications of ACM, Vol. 35.* s.l.:s.n., p. 65.

Dexter, A. & Ridley, J., 2021. *The best VR headset in 2022.* [Online]
Available at: https://www.pcgamer.com/best-vr-headset/
[Accessed 01 January 2022].

Eldad, A., 2021. *Unity vs Unreal, What Kind of Game Dev Are You?.* [Online]
Available at: https://www.incredibuild.com/blog/unity-vs-unreal-what-kind-of-game-dev-are-you#:~:text=Unity%20vs%20Unreal%3A%20key%20differences&text=They%20say%20that%20coders%20prefer%20Unity%20while%20artists%20tend%20to%20like%20Unreal.&text=A%20lot%20of%20graphics
[Accessed 21 Ocotober 2021].

Frank, J., 2012. *A Literature Review of Gaming in,* s.l.: Pearson.

Fuchs, H. & Bishop, G., 1992. *Research Directions in Virtual Environments.,* Carolina: University North Carolina.

Gigante, M., 1993. Virtual Reality Systems. In: *Virtual Reality: Definitions, History and Applications.* s.l.:Academic Press, pp. 3-14.

Haas, J. K., 2014. *A History of the Unity Game Engine,* s.l.: Worcester Polytechnic Institute.

Heidi L Lujan, S. E. D., 2006. First-year medical students prefer multiple learning styles. *Advances in Physiology Education,* 30(1), pp. 13-16.

Jorge Gaytan, B. C. M., 2007. Effective Online Instructional and Assessment Strategies. *American Journal of Distance Education,* 21(3), pp. 117-132.

M. Bellamy, A. W., 2011. *Using Online Practicals to Support Lab Sessions,* s.l.: Ed Share Soton.

Mavor, N. I. D. &. A. S., 1995. *Virtual Reality: Scientific and Technological Challenges.* s.l.:National Research Council.

McGreevy, M. W., 1993. *Virtual Reality Applications and Explorations,* London: Academic Press.

McMahan, A., 2003. Immersion, Engagement and Presence: A Method for Analyzing 3-D Video Games. In: M. Wolf & B. Perron, eds. *The Video Game Theory Reader.* s.l.:s.n., pp. 67-86.

Peter J. Fensham, R. F. G. R. T. W., 1994. *The Content of Science: A Constructivist Approach to Its Teaching and Learning.* s.l.:Psychology Press.

Scheweber, E. v., 1995. Virtually Here. *PC Magazine,* pp. 168-198.

Scripting, B. V., n.d. *Unreal Engine Documents.* [Online]
Available at: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/
[Accessed 21 October 2021].

Smid, A., 2017. *Comparison of Unity and Unreal Engine*, Czech: Technical University in Prague.

Thalman, N., 1994. Tailoring Clothes for Virtual Actors. In: *MacDonald and Vince.* s.l.:s.n., p. Chapter 13.

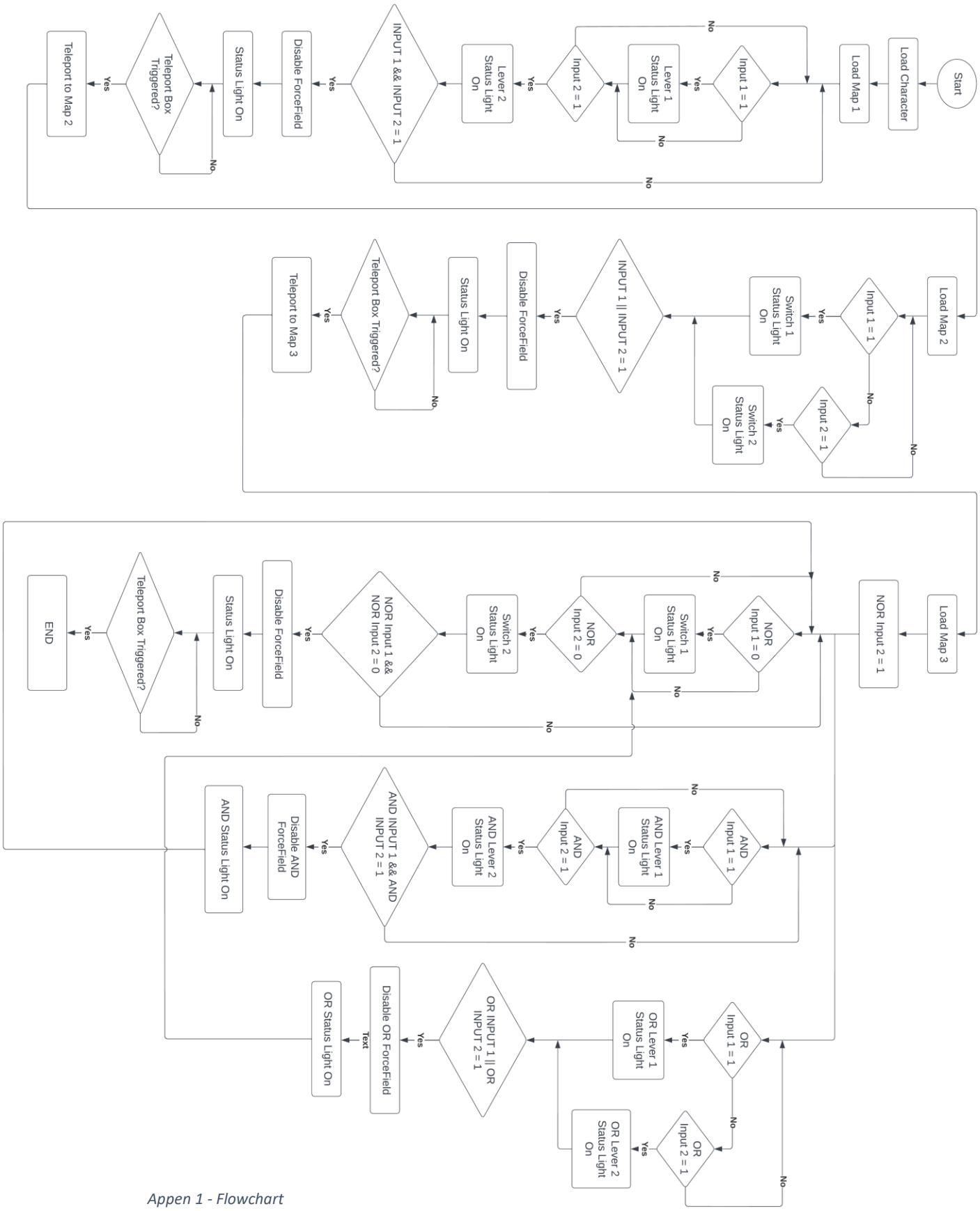
Unreal Engine, n.d. *Features of Unreal Engine.* [Online]
Available at: https://www.unrealengine.com/en-US/features
[Accessed 21 October 2021].

Unreal Engine, n.d. *Unreal Engine 4 Documentation.* [Online]
Available at: https://docs.unrealengine.com/4.27/en-US/
[Accessed Monday December 2021].

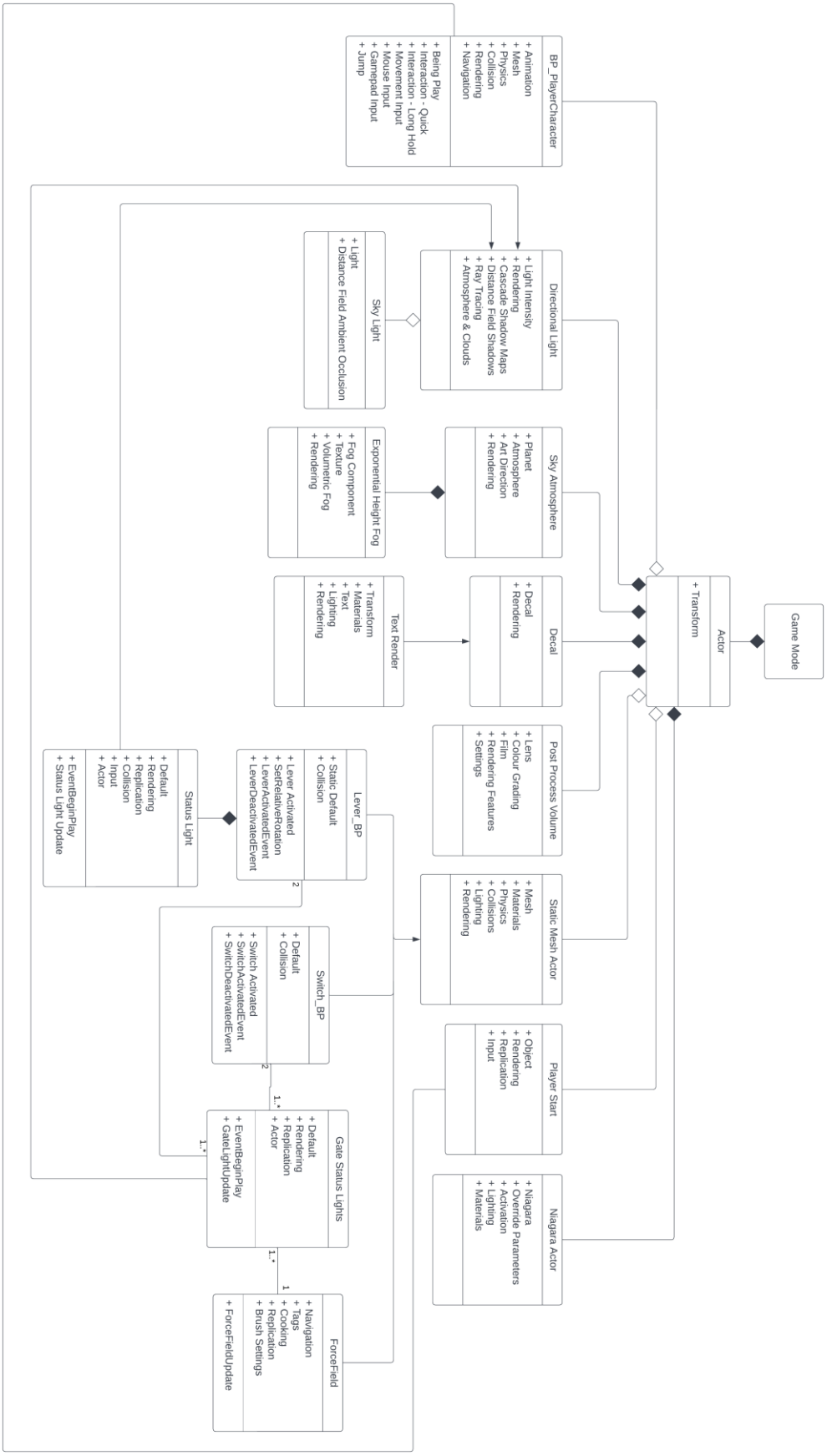
Walter B. Barbe, R. H. S. M. N. M., 1981. Modality Strengths. *Academic Therapy,* 16(3).

Zheng, J. M., Chan, K. W. & Gibson, I., 1998. Virtual Reality. *IEEE Potentials,* 17(2), pp. 20-23.

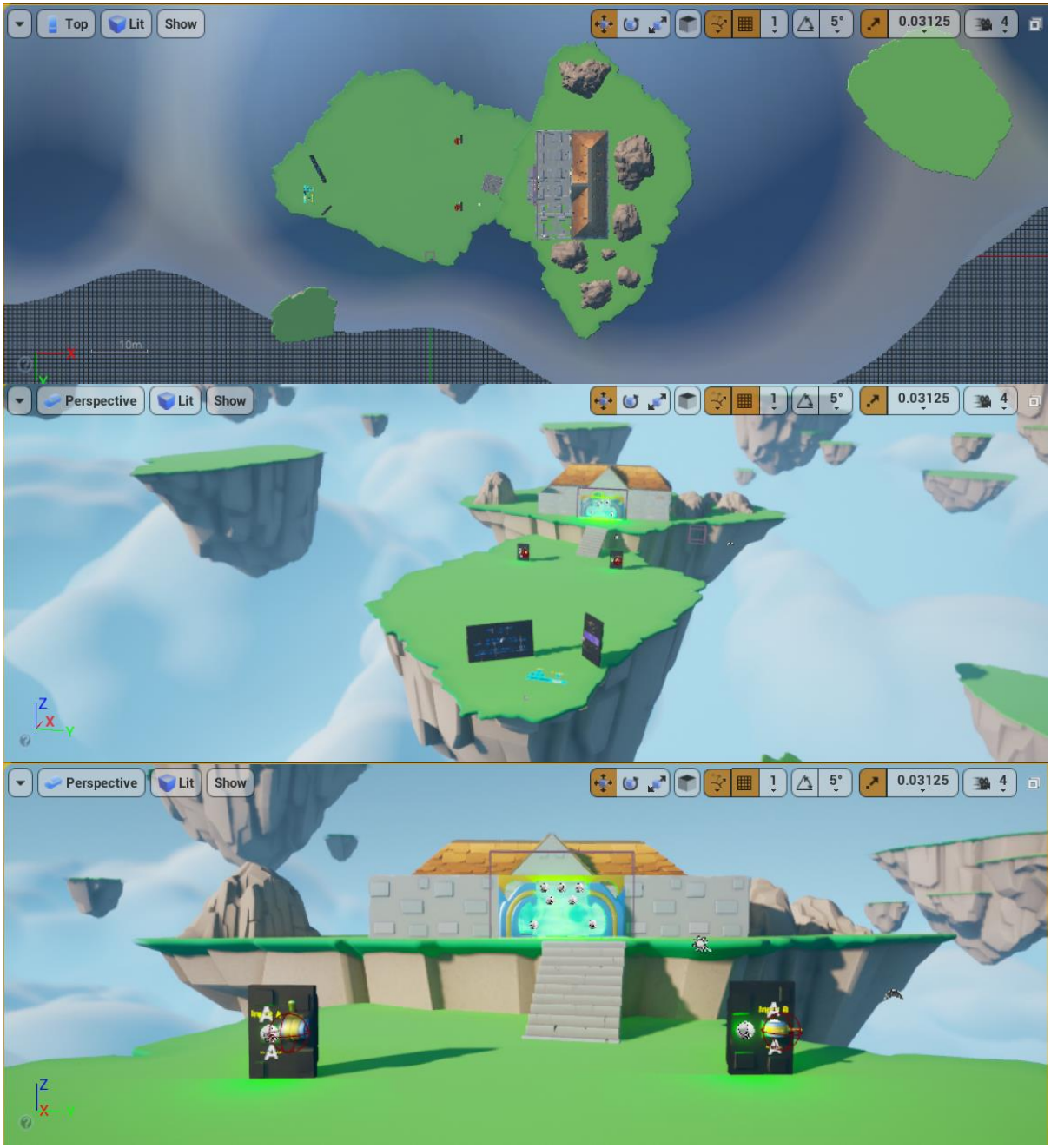
Appendix



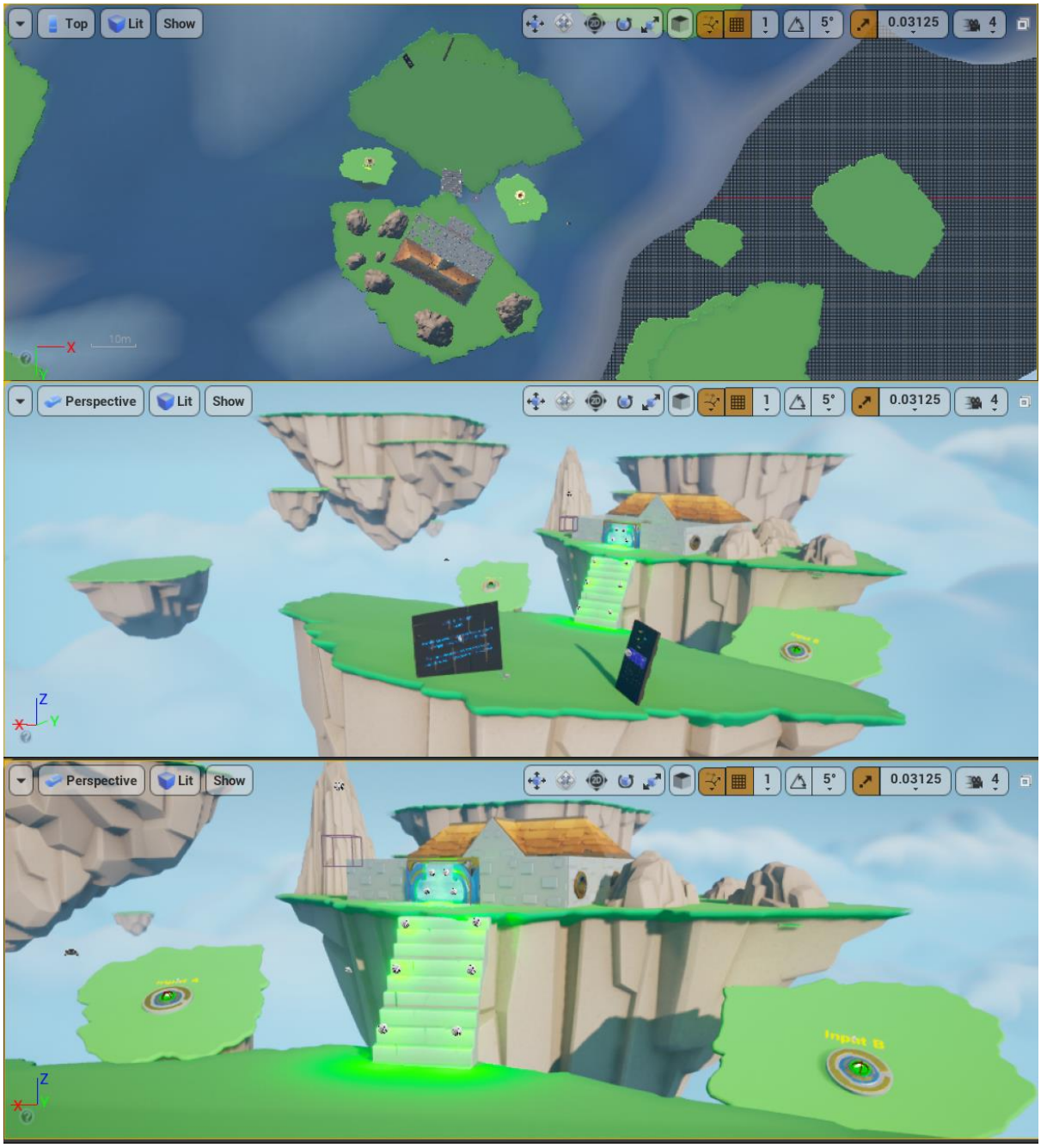
Appen 1 - Flowchart



Appen 2 - Class Diagram

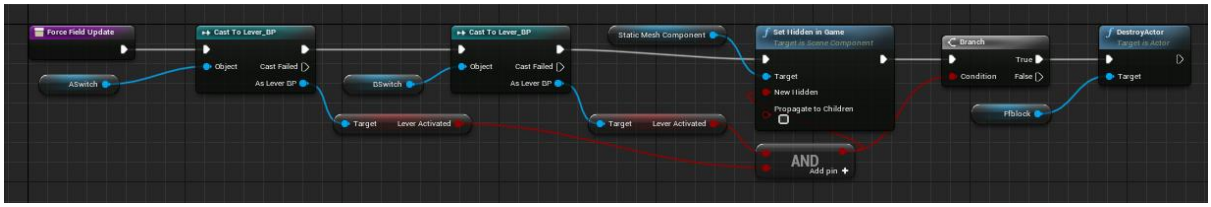


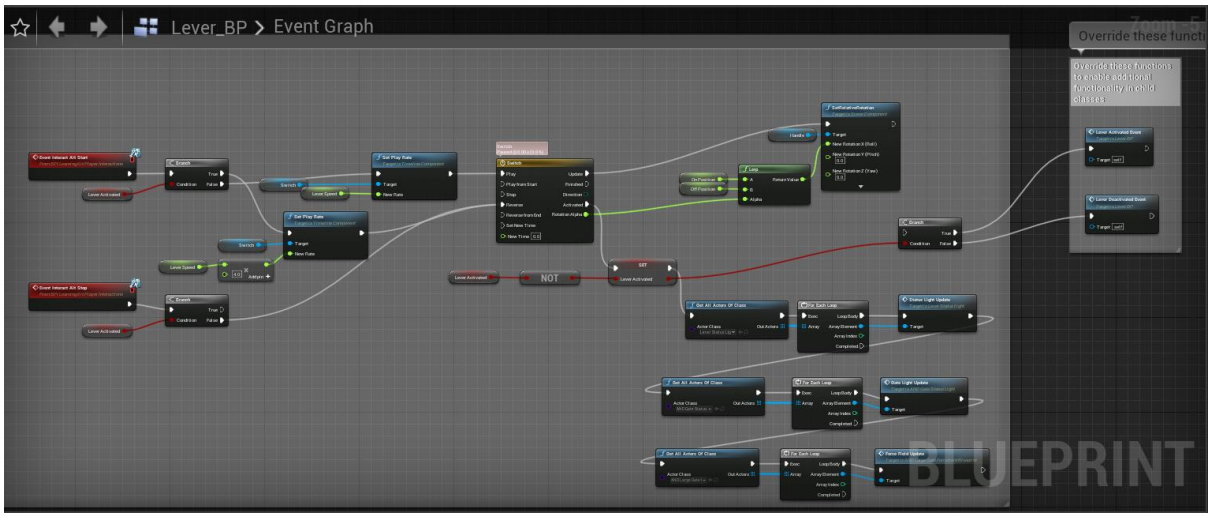
Appen 3 - Level 1

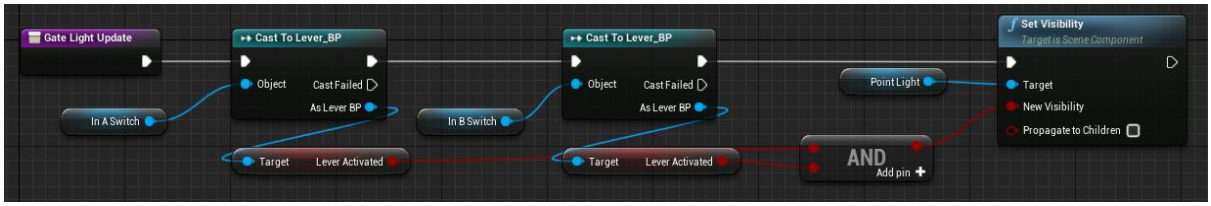


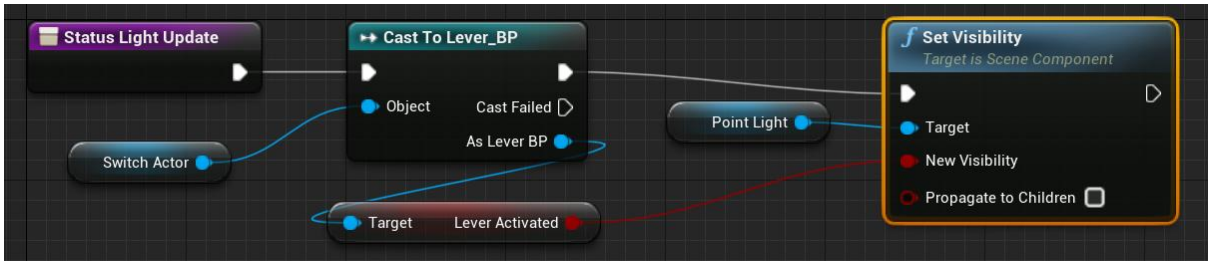
Appen 4 - Level 2

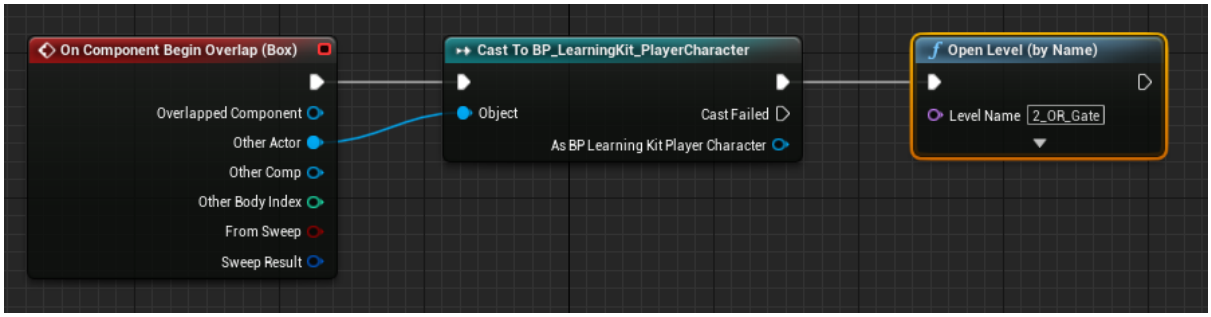
Appen 5 - Level 3



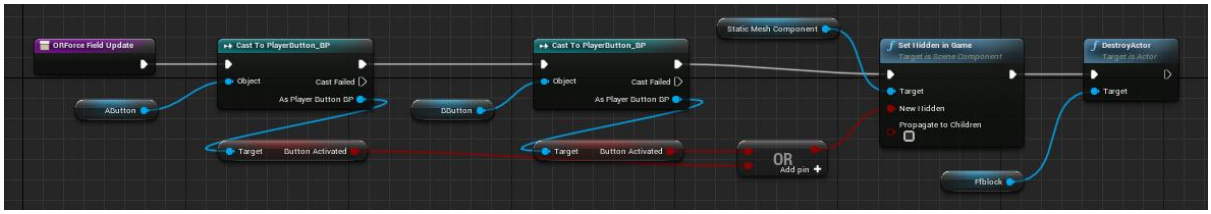


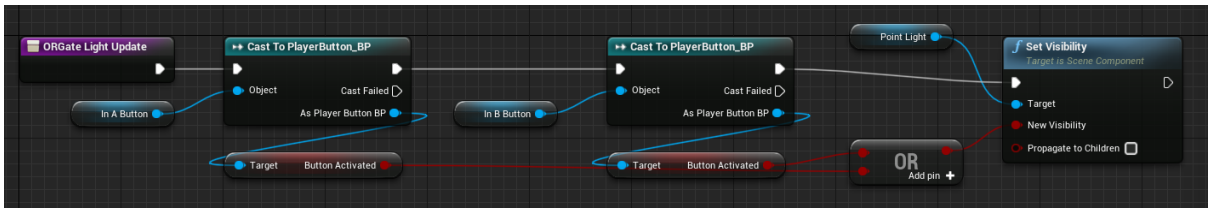


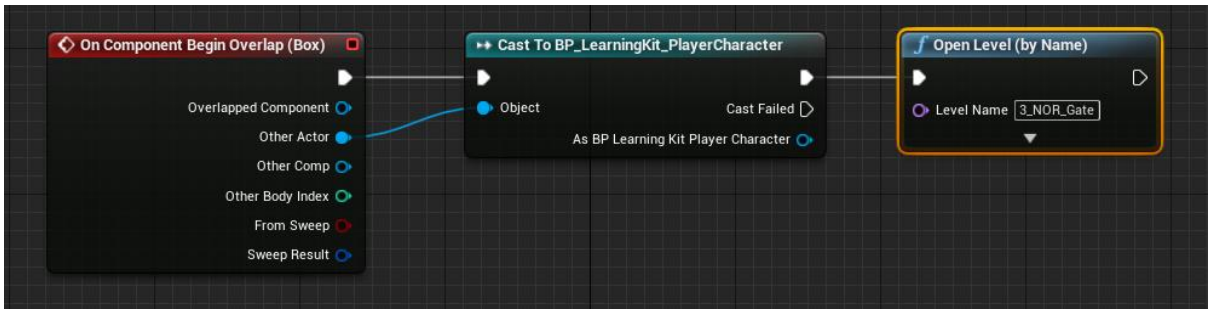


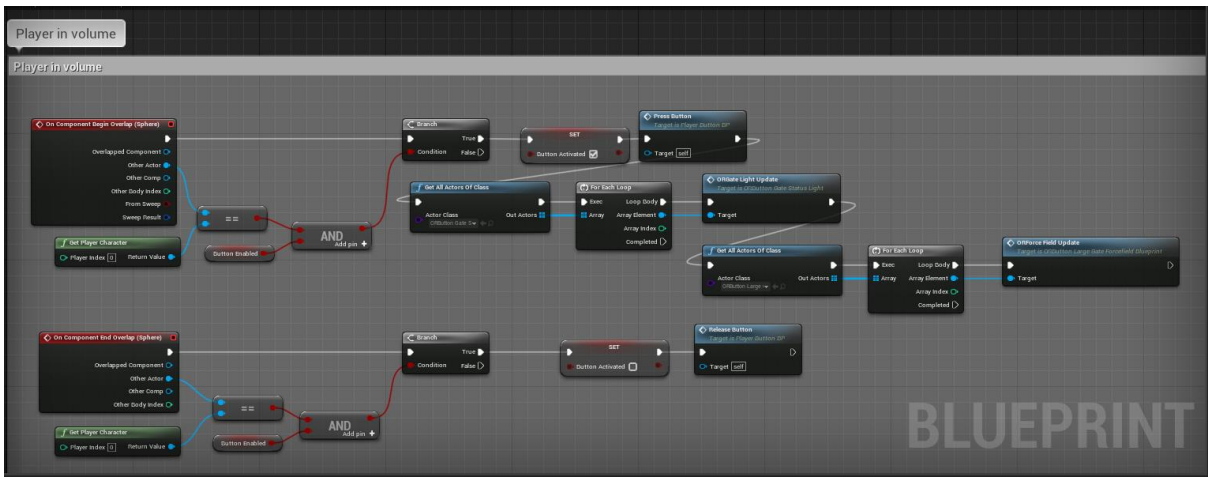


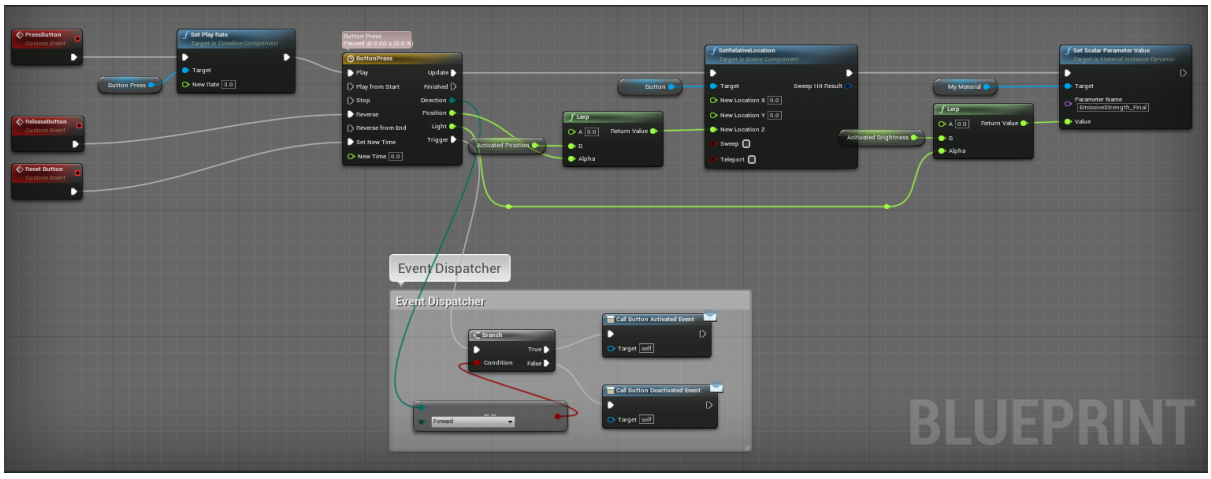
Appen 6 - Level 1 Blueprints

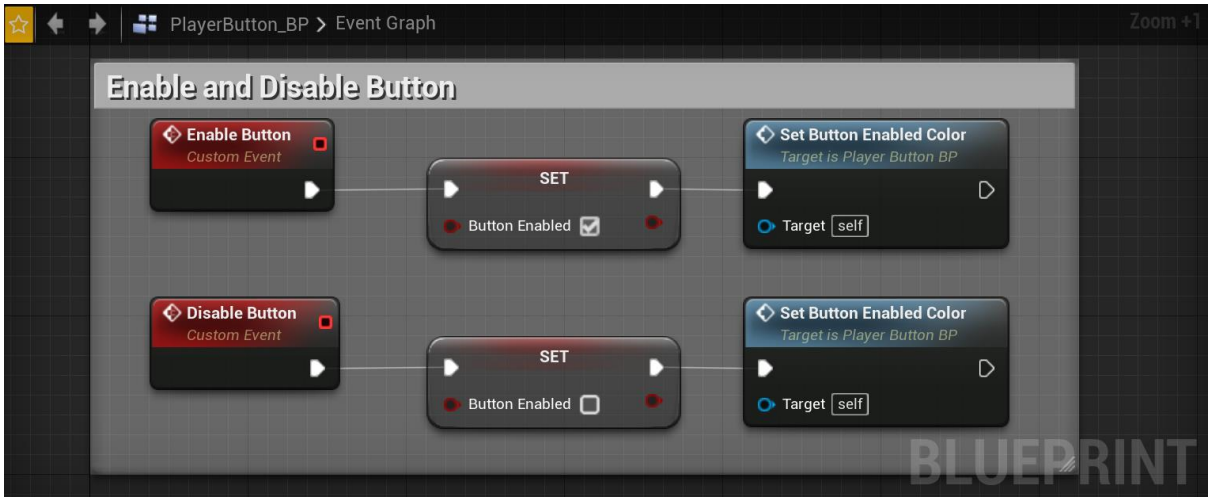

OffForce Field Update
Cast To PlayerButton_BP
Object
Cast Failed
As Player Button BP
ADutton
BDutton
Target
Button Activated
Static Mesh Component
Set Hidden in Game
Target is Scene Component
Target
New Hidden
Propagate to Children
DestroyActor
Target is Actor
Target
OR
Add pin
FBlock

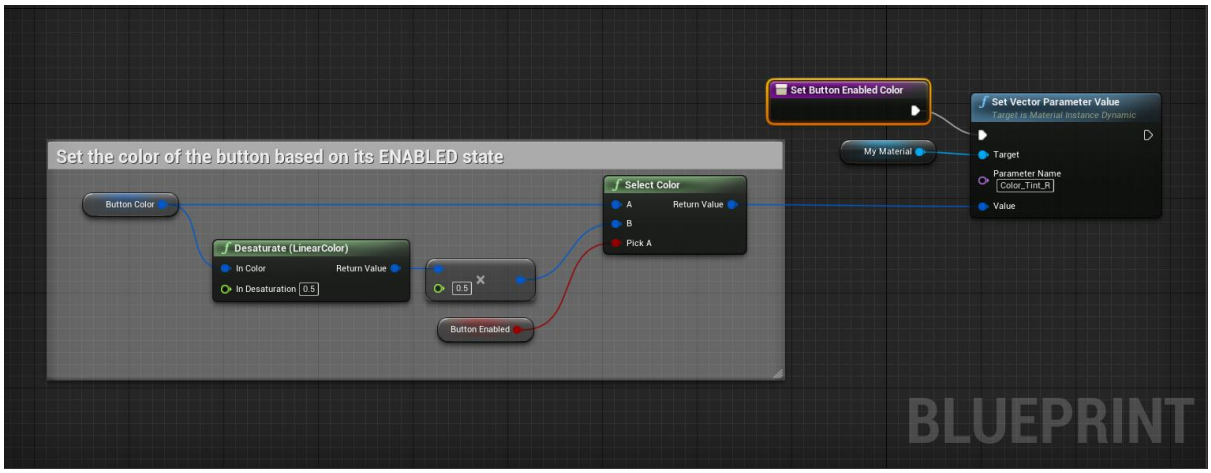

ORGate Light Update
Cast To PlayerButton_BP
Object
Cast Failed
As Player Button BP
In A Button
In B Button
Target
Button Activated
Point Light
Set Visibility
Target is Scene Component
Target
New Visibility
Propagate to Children
OR
Add pin


On Component Begin Overlap (Box)
Overlapped Component
Other Actor
Other Comp
Other Body Index
From Sweep
Sweep Result
Cast To BP_LearningKit_PlayerCharacter
Object
Cast Failed
As BP Learning Kit Player Character
Open Level (by Name)
Level Name
3_NOR_Gate

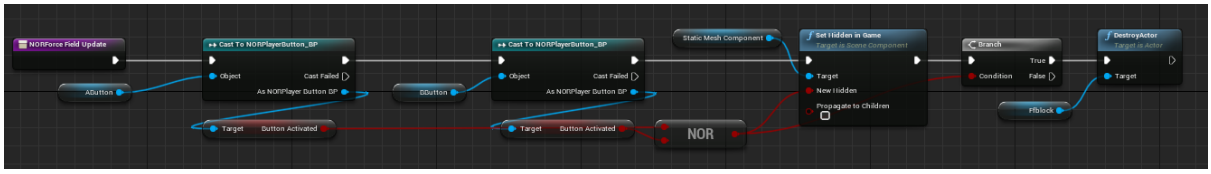

Player in volume
Player in volume
BLUEPRINT

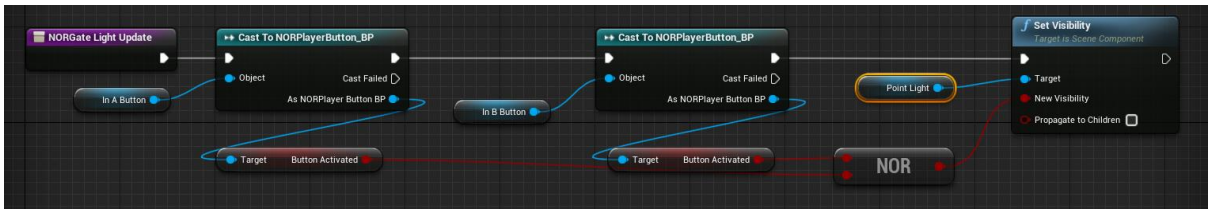

Event Dispatcher
Event Dispatcher
BLUEPRINT

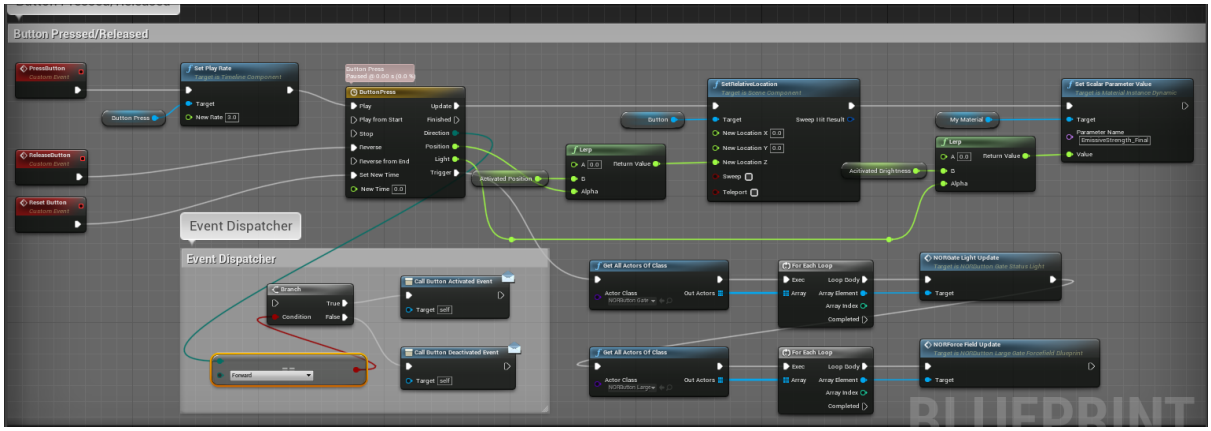


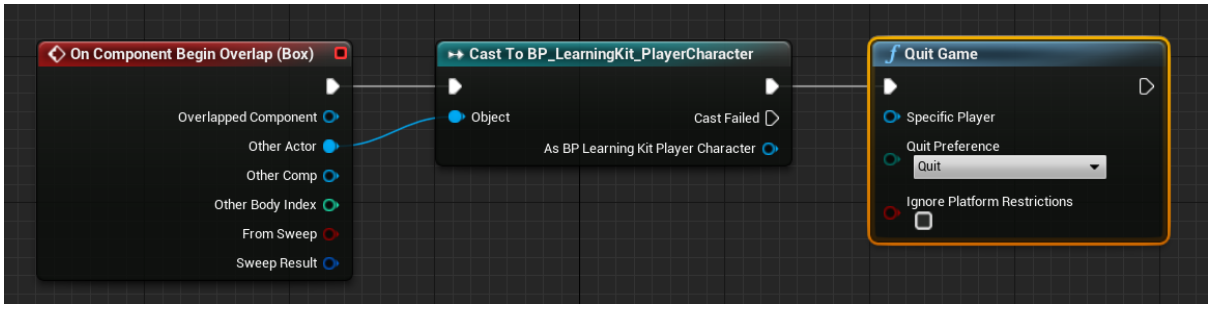


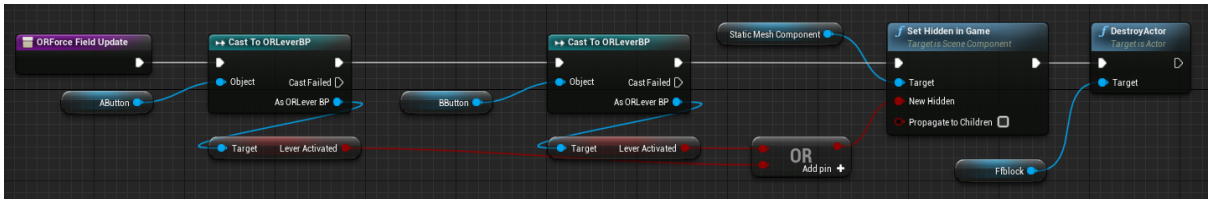
Appen 7 - Level 2 Blueprints

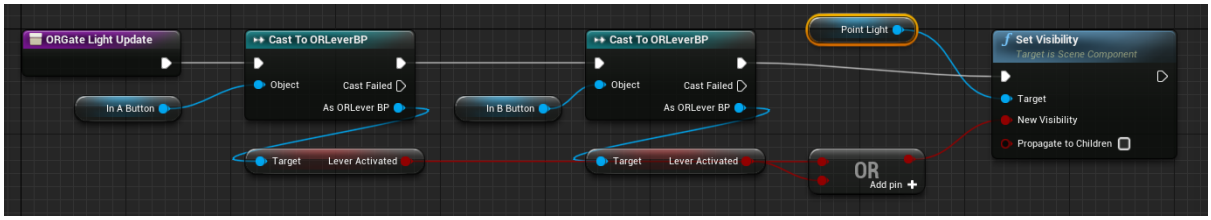












Appen 8 - Level 3 Blueprints