Dimensional Doors

A collaborative VR project

Mei Maddox
William Pembleton

December 2019
For use by faculty and students at New College of Florida

Table of Contents

(All sections link to the page listed to its right. Simply click on the title of interest to navigate)

Project Overview

For those who've dabbled in the world of Minecraft, you might have encountered a mod called Dimensional Doors. In this mod, the player runs into doors scattered throughout the world which teleport them to different dimensions (thus the name), each with their own little puzzle or idea. This project was born out of curiosity to experience the interesting mod in virtual reality.

Feel free to check out the video demo on YouTube and/or clone the repo on GitHub and use the project for your own purposes!

Software Download and Installation

The following software was used for the creation of this project (links are provided):

- Git / Github Desktop
- Unreal Engine 4 (UE4)
- Blender

The software will be explained in more detail but for now you'll just need to set up a few accounts: one for Github and one for the Epic game store/ Unreal Engine.

Unreal Engine 4 (UE4)

Unreal Engine is the main program that was used for this project; it runs the VR game and handles most things for us. When creating your own project feel free to download install the newest version of the engine (Dimensional Doors was created in version 4.23.*). Differences between the versions shouldn't be so drastic as to make this tutorial out of date. Go ahead and download the engine; it's 20-30 gigs so set aside some time for this.

Git / Github Desktop

Github Desktop is an interface for the tool Git - a software used for version control. Similar to the Cloud, it allows you to remotely access current working projects and upload changes to that project from any computer device, eliminating the need to copy files between work (VR Lab) and home (personal computer). Github Desktop hides most complexities of Git, making it an easy medium with which to learn Git. There are sources on the internet and Github (like GitHub Education) that teach you how to use git; it is free to use at your discretion, however, note we opted to rely on Google and trial and error.

After downloading Github Desktop, log in and create a repository on your computer.

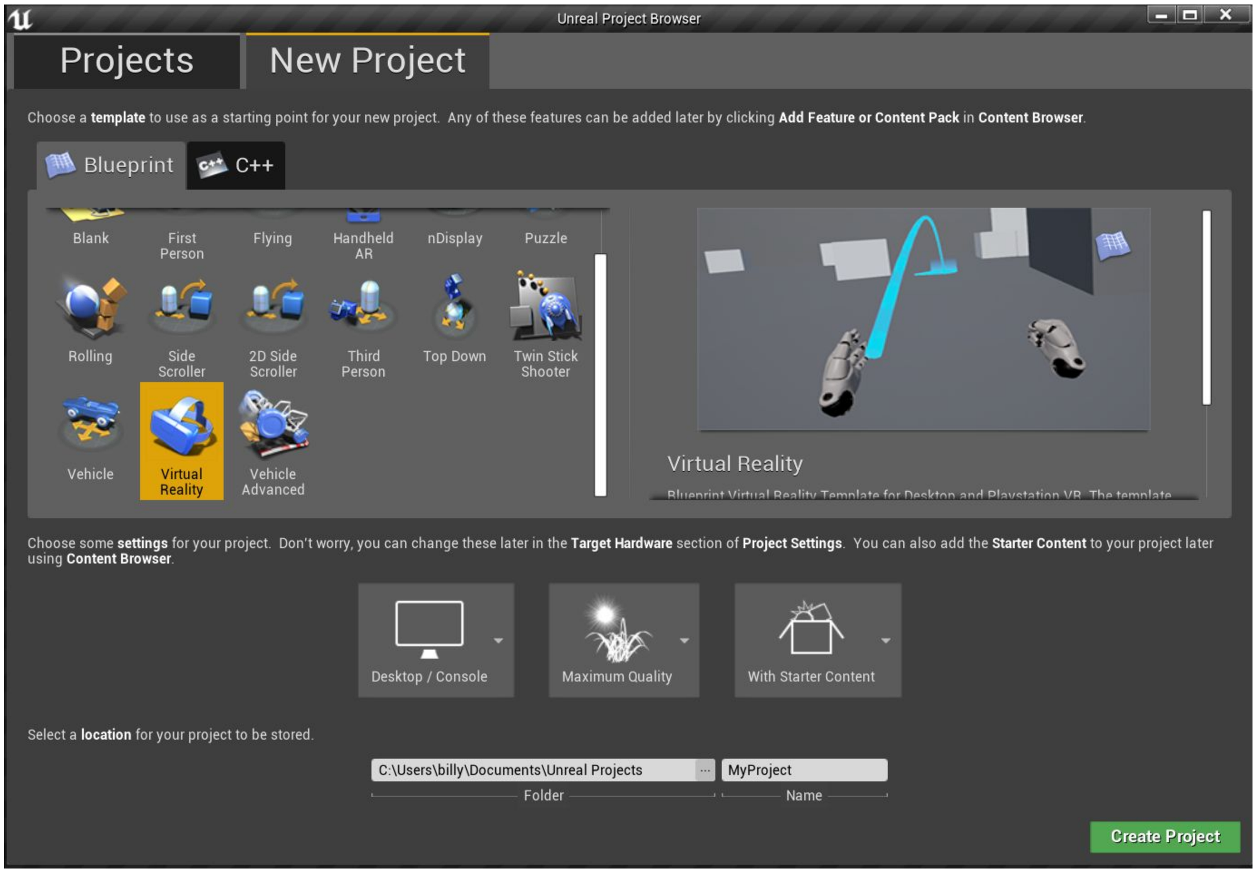
Blender

Blender is a free and open-source program that allows you to create and manipulate models in a 3D environment. This, or similar software, is necessary to learn should you wish to include custom objects/models in your project. Otherwise, you can opt to purchase/obtain pre-made objects from websites such as TurboSquid and cgtrader. Although Unreal also offers 3D suite functionality it is not streamlined for it, making a dedicated program like Blender ideal for easy and fast creation.

The newest version of Blender currently available is version 2.8. We recommend watching the First Steps - Blender 2.80 Fundamentals video on Youtube. It appears to be a remake of an older tutorial to fit the new UI, therefore providing a solid foundation for using Blender. It covers all/most things in Blender so watch what you think you need.

General Project Creation

Launch UE4 and create a new project. UE4 supports project creation in two formats: Blueprints and C++ code. For this tutorial, we opted to use Blueprints. A blueprint can be thought of as a class in Object-Oriented languages. Choose the Virtual Reality starting template and leave starter content on - this adds content such as some generic assets, music and sound fx to help you get started. Set the location for the project as your Git repository.


Unreal Project Browser

Projects | New Project

Choose a **template** to use as a starting point for your new project. Any of these features can be added later by clicking **Add Feature or Content Pack** in **Content Browser**.

Choose some **settings** for your project. Don't worry, you can change these later in the **Target Hardware** section of **Project Settings**. You can also add the **Starter Content** to your project later using **Content Browser**.

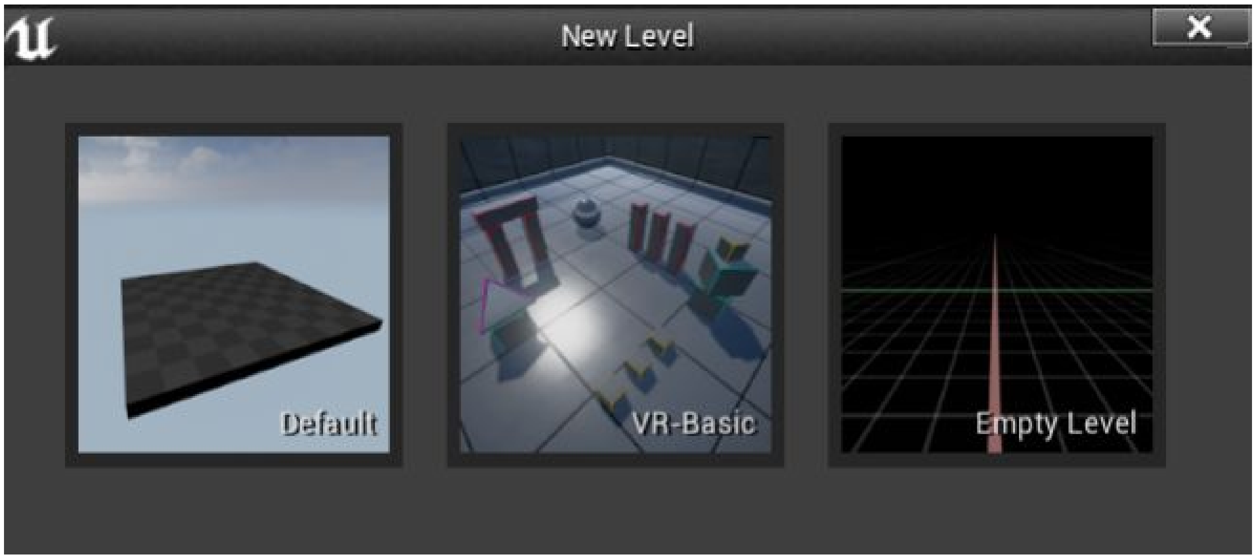
Desktop / Console | Maximum Quality | With Starter Content

Select a **location** for your project to be stored.

Folder: C:\Users\billy\Documents\Unreal Projects
Name: MyProject

Map (Level) Creation

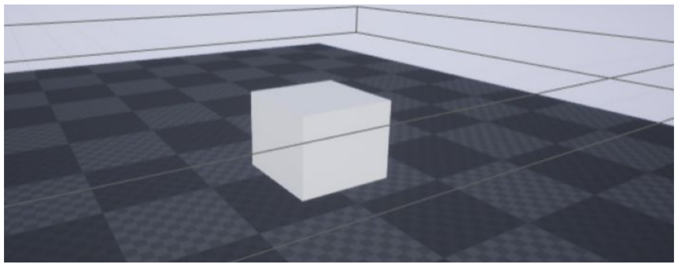
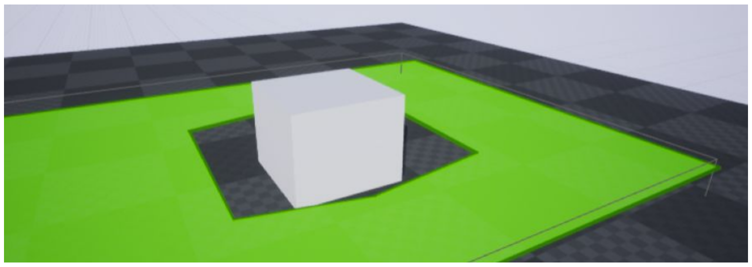
Now it's a good time to make a new level. There are two ways to create a new level: from the content browser and from the file menu. Level creation within the content browser creates a blank level which has no lighting so seeing any object placed within it is harder without creating a light source first. We recommend using the file menu. To do this, go to File → New Level → Default. Feel free to delete unneeded aspects such as atmospheric fog or the sky, etc.. Make sure to save the level if you wish to have it in the project. Saving the level triggers a popup that prompts you for the location you'd like to store the newly created level in. To keep the project organized, we recommend creating a folder for all levels called Maps.



Pawn

A pawn is the basic controlled actor class within Unreal, controlled either by a player or AIController. Floating point movement can be added to basic pawns to allow for movement along all axis freely. Characters are a type of pawn that have gravity/physics enabled therefore constricting movement to the slope of the terrain. Movement is further defined for characters, allowing for different settings of speed and animation depending on whether the character is walking, crouching, swimming, flying, etc.

Automated movement using a native method in Unreal relies on an object called the *Nav Mesh Bounds Volume*. This object defines a region of valid planar movement within your level based off collision with objects pawns can stand upon. The image . This same volume is also used to enable *player teleportation*. Pressing ‘p’ while the object is selected brings up a visual representation of all the valid places (See picture below). Make sure to adjust the size, scale, and location of the volume to accommodate your needs.

The pawn representing the player is a bit unique: extra functionality is offered with the inclusion of hands. To access this pawn, browse to Content/VirtualRealityBP/Blueprints, and drag the MotionControllerPawn into your level. Pressing ‘end’ snaps the pawn camera to the floor. Make sure to select the pawn in the level and scroll through the settings and set the controller to “Player 0.” Without this, you won’t be able to control the pawn! Setting this within the level allows you to create multiple instances of the same pawn for each player that can be controlled independently of each other. Unless you are implementing multiplayer functionality, you can change this setting within the pawn blueprint itself to save yourself work.

Test Run

Before proceeding, test to make sure everything works in VR. Near the top right of the tool bar, there is a button that says Play with an arrowhead. Click the dropdown next to it and choose “VR Preview”. If everything went well you should have hands in VR that mimic your actual hands. If the camera height is inaccurate, you can offset this within the pawn blueprint.

Mesh (Objects)

There are two types of mesh: static and skeletal. The shape of a static mesh is absolute/non-dynamic. Note, the mesh can be moved, rotated, and scaled, however it cannot have one vertex move separate of the unit as a whole. It’s counterpart, however, does offer that functionality. Skeletal meshes are made up of vertexies that can be moved and animated independently of each other. Skeletal meshes are typically used to represent and animate creatures or people within the world.

Project Specifics

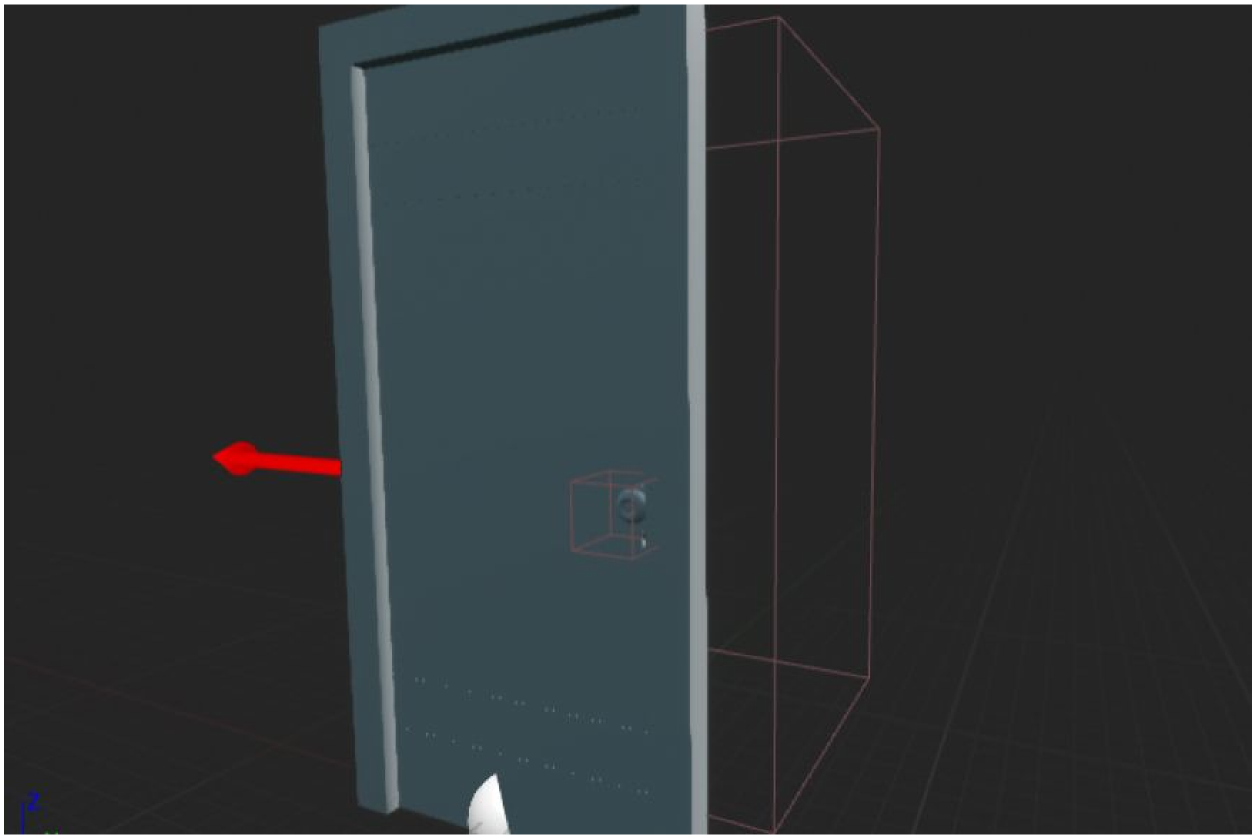
Scalability and replayability were two main goals of the game. In other words, the design should easily accommodate the addition of new levels without placing too many constraints or requirements on how the level should function, minimize the code dependencies for individual/specific levels, and maintain a certain level of randomness. This section details the challenges and general rationality behind design challenges choices within the project as a whole. Further explanation of how certain features are implemented are also provided below.

Door

The door acts as a gateway to another level. All the functionality for the generation of another level was placed within the door model itself. Therefore, the only restriction on a level is that it contains a door reachable by the player at some point in time. As you can see from the image below, the door asset contains two trigger-boxes and an arrow component. The trigger-box on the door knob opens the door while the trigger-box on the backside opens a new level; both of these events only trigger if certain conditions are met.

- *Opening the Door*: Opening the door happens by having an arrow component be the parent of the actual door (not the frame). Then you can rotate the arrow component around so that the static mesh comes with it. The actual rotation happens once a hand comes into contact with the trigger box surrounding the handle. From there we check if the hand is closed. After that we run into a Move Component To node, this allows us to basically transform a component however we want, for us, we want to rotate the arrow component by 90 in the Z direction over .5 seconds.

- *Level Loading:* A custom function was created to load the next level. Because we are traversing the array in sequential order, finding the index of the next level is as easy as incrementing the index of the current level by one. A quick bounds check ensures the game won't crash should all levels be traversed and teleports you back to Spawn after shuffling the array of levels. This way the sequence of levels will be different should the player wish to complete the game again.



Game Instance and Global Variables

Variables created in blueprints, whether it be an object or a level, are all protected local variables. In fact, global variables are defaulty unavailable in a project. To enable the creation of global variables a special *Game Instance* blueprint must be created and set as the game instance blueprint within the project settings. Unlike other blueprints, there exists only one instance within the whole project and any reference to it does not create a copy but points to the original instance.

For this project, the blueprint was called "MyGI" and mainly stored an array of all levels. If a new level is created, simply add the level name to the array. At the beginning of the game, the array is shuffled to ensure different level sequence order between playthroughs.

Pawn Alterations

The original class of the MotionControllerPawn is *Pawn*. It doesn't support movement outside of displacement within the physical world. This, however, constrains movement to the horizontal plane and can create height inconsistencies if the level floor is sloped (player walk through the floor). To ensure player movement remains consistent with the slope of terrain, gravity was added to the pawn via reclassing to *Character.* Gravity cannot be added by mere *enable gravity* toggle to a collision sphere as the only movement option for basic pawn is floating point movement.

Furthermore, requiring the player to physically move constricts the size of the level to the dimensions of the room. Normally, teleportation compensates for this drawback by offsetting the pawn position relative to the in-game world. Unfortunately, teleportation could allow for the player to bypass sections of the level. Therefore, teleportation was removed and support for movement via the Motion Controllers was added. This was accomplished by adding a Motion Controller axis event to read joystick input and use it as a scalar factor for vector movement. Speed is capped at .3 but controls for adjusting speed were added as well.

Note, that re-parenting the pawn to *Character* creates a collision sphere that prohibits movement into actors (such as walls). For the start of each level, the player must be located in the middle of the physical room otherwise the collision will be offset. This weird quirk of VR doesn't occur with the default pawn as no collision exists at all (player can walk through any object). So any offset from the pawn in VR isn't noticeable.

A modified version of this pawn was used in the Kitchen level - see kitchen for details.

Level Creation

Each map is a separate level. All levels are organized together within a folder called Maps.

Spawn

The player always starts in the spawn room. The purpose of this level is to allow players to be in a room where they can just play around with the controls and get comfortable in VR. The walls of the room were created using box brushes. Brushes are useful during early level design as they allow for rapid prototyping. It is recommended, however, that they be replaced by static meshes for final production as static meshes are far more efficiently rendered.

Cave

Within this level the player is taken to a cave system and required to walk over a pit of bubbling slime. There are ambient whoosh noises to simulate air circulation throughout a cave paired with the sounds of bubbling from the slime below the bridge. The player is required to navigate across the bridge, avoiding any holes/missing boards. Should the player accidentally fall through a gap in the bridge or off it entirely, they would hit the slime and trigger their death - respawn at the beginning of the level.

- <u>Ambient Sounds</u>: The sound assets are edited audio clips from Youtube. Ambient noises are set up by creating a cue from the sounds assets with looping audio and adjusting the pitch and volume. The cues are then placed in the world. Initially, a cue has a spherical range with two radiuses - inner and fallout. The sound is audible within the bounds of the fallout radius, increasing in volume as you near the inner radius where it is heard at full volume. These settings can be adjusted by selecting the Override Attenuation box. For the purpose of this project, the attenuation settings for the bat noises were significantly smaller, only allowing for perception near the door.

- Landscapes: Initially, the project utilized the landscape feature in Unreal. The landscape was generated using a heightmap - black and white picture representing elevation - obtained from google images. Using the sculpt tool (and holding <shift>), a divot was created in the center to accommodate the slime pool and the bridge. The polygon tool is useful to reduce material stretching. Although the cave landscape was realistic, the amount of detail unfortunately caused a significant load time when attempting to open another world. Landscapes are large assets typically meant to represent the whole world map as opposed to just a small area. Landscape design works well with level streaming as it only renders/loads sections of the world at one time, normally the sections adjacent to the player. This allows for good exploration in an open-world setting. For the purpose of the project, however, a "landscape" was created in Blender and imported as an asset. Unfortunately, the number of polygons were significantly lower and stretched -without a way to repolygonize - causing poor material placement. The resulting cave look is therefore merely a brown color.
- Slime (Materials): The slime pit is merely a plane with a material. To make the slime material, a texture (picture) with an appearance similar to that of molten lava was blended with a light green and used as its emissive color - base color that eliminates light. This creates a texture that looks the same except the color continuum stays within hues of green. To create the illusion of bubbling, the texture needs to be tessellated. To do this, the panned output of a noise texture was used as the factor for world displacement. A Panner allows for texture movement across a surface. Because the surface is a plane, this creates the illusion of 2d movement. 3d height of the surface is achieved through tessellating the desired texture based off the greyscale from the noise texture. In simple terms, it creates an offset for the material based off the original surface. The result: a dynamic and random appearing bubble/waving lit green surface.
- Death: The death sequence is fairly straightforward. Because gravity is enacted on the pawn, the player will fall upon walking off the bridge and eventually land on the slime plane. This hit triggers an event that plays an acid burning sound, fades the camera to green, teleports the player to their starting position within the level, then un-fades the camera. The sound is merely an asset that is called within the blueprint to be played as opposed to placed in the world like the other sounds. Camera fade is a simple function with a defined fade color, fade duration, and alpha scales. Teleportation turns out to be the most complex aspect of the death. It does not utilize the teleportation feature of the default player as the player is not intended to be able to teleport at their whim. Therefore, it uses a function called *SetLocationAndRotaiton* that manually "teleports" an actor. The player's initial start position and rotation (direction player is looking) is recorded and saved as variables within the level. These variables are then called during the (slime) plane hit event and passed as parameters for the SetLocationAndRotation function.
- Bridge: Although the bridge appears to be one cohesive asset, it is, in fact, three different assets - one broken bridge and two complete bridges - laid end-to-end. In order to force the player to walk along the beam of the broken bridge or risk falling, the player collision for the entire asset had to be turned off because the "hole" is merely a visual representation but doesn't exist when calculating collision (player could walk over the "hole" in the bridge as it's still considered part

of the asset). Turning off collision, however, means the player cannot walk upon the broken bridge at all. To remedy this issue, planes were placed underneath the bridge corresponding to the intended walkable areas. All assets have a toggle that indicate whether or not they are hidden within the level. Merely turning the toggle on for the plane, creates the illusion of a walkable bridge with holes even though the player would fall through the actual bridge asset if the planes were not beneath it.

Kitchen

The idea for this level was to create a level that plays with perspective. The plan was to let the player play around a bit until they realize there is a little bottle on the table that if you drink it you shrink down (If you've seen the old Alice in Wonderland movie Disney released in 1951 then you might recognize we're recreating the door scene).

First up is to create the kitchen scene. One of the ways to do this is to import a full scene into Unreal. The reasons that you shouldn't do this:

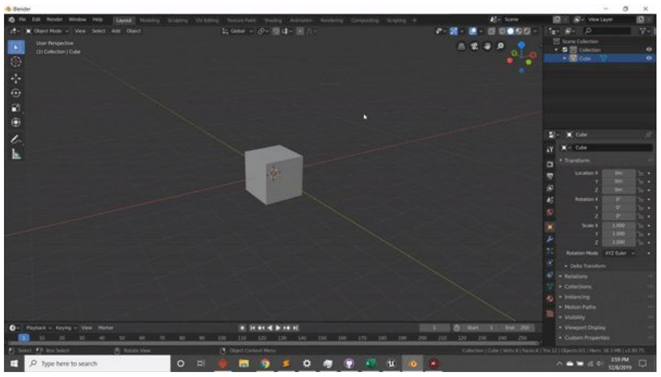
1. Objects in Unreal are imported individually which means if you want to import a whole scene with tons of objects into Unreal then you have to place each of those objects.
2. Materials brake often. So somewhere between the fbx file and importing materials are broken and are replaced with a completely white placeholder material. This means that you need to individually fix each material manually. This is compounded by the ton of objects problem from before.

The better way to do this is to import models 1 by 1 for what is needed. This reduces the problem dramatically because you can combine the meshes beforehand in Blender and make sure that the materials get imported properly.

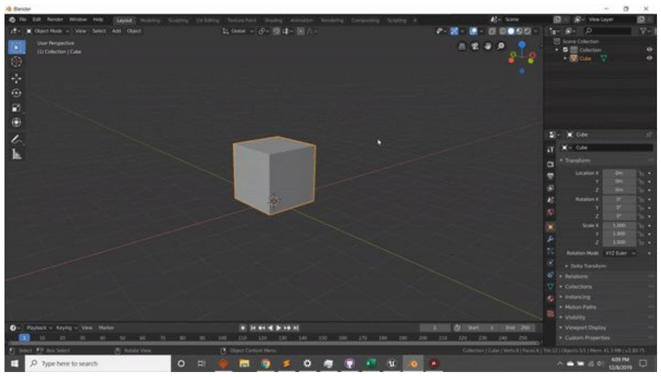
So let's talk about how scaling can be implemented. There were two ways that our team thought of to simulate the player shrinking. One of them was to scale the player and the other was to scale the world around them.

Our first attempt was to scale the world around the player. The reason that this idea didn't work was because of how scaling works in 3d software. When you scale (rotate or move) the object scales by something called its origin point. This is a point that was created in the 3d modeling software by which every other point is based on. This means that if you try to scale the object it scales in reference to its origin point. I made these little gifs to help explain. The little orange red and white circle in the center is the origin point for each object.

From Center



From Outer



The reason that scaling the world didn't work is because the objects need to scale and move relative to the player. The other problem is that if all objects scale up by a certain amount they wouldn't move (since the origin point isn't moving) and they would run into each other which isn't the effect that was planned.

The other way that one can implement scaling the world is to create the entire world as a blueprint. Which for us means the whole level can be scaled via one origin point. This was much closer to the intended effect, especially if the player shrinks down where the origin point of the level is. However, if the player isn't near the origin point of the level then the scaling is just a little bit off. For that reason, we didn't go with this effect

The final way to implement scaling is to scale down the MotionControllerPawn (the blueprint that is basically the player for VR). For this method, you can attach a trigger box to the camera. With the trigger box you can detect when an object runs into it. For detecting which object hits the trigger box you need to make a blueprint interface, and a bottle object which implements that interface. You can make a blueprint based off of the BP_PickupCube located in Content/VirtualRealityBP/Blueprints/BP_PickupCube. Duplicate the cube by right clicking and place it wherever you want. The reason for this is that the BP_PickupCube can be picked up with the players hands. Then in the content browser create a blueprint interface by right clicking and name it bottle. Then open up your duplicated object   I simply shrunk down the topmost component (blueprints work in a hierarchical system)

Ocean

What's better than the calming sounds of an ocean and schools of fish swimming by? Obviously running into a giant eel and shark! This level simulates life at the bottom of the sea. Surrounded by rocks, coral, seaweed, and a sunken submarine, the player wanders along the seafloor in search of the door. Schools of fish freely swim about and a shark roams above. Within the submarine itself lies the door and a large eel beside it. Fret not, nothing in this level attacks the player.

- <u>Asset Animation (Shark/Eel)</u>: Both the shark and eel are pawns with skeletal meshes. The eel is a basic pawn while the shark is an AICharacter. Both of these assets have movement animations. Creating these animations is a bit complex. First, a physics body must be created off of the skeleton. A physics body creates limitations on rotation for every joint - vertex - of the skeleton effectively simulating realistic bends and chained movement (displacement of one joint causes a bend/rotation in another, etc.). Using this physics body, an animation can easily be created. The beauty of animation lies in its simplicity; animation blueprints are merely looping movement over time. For this example only the shark will be explained, however, the same logic was applied to the eel. At t0, the shark starts completely straight. Slowly move the tail to one side and create a key. This creates a goal point for the joint selected (tail) to move to by time t1. The physics body handles the constriction of the other joints to simulate realistic tail movement to that position. Next, drag the same tail joint to the opposite side and create a key. Lastly, return the shark to its initial start position and create a key. The shark now loops in a swimming animation. This is the basics of animation.

    - Adding sound to the animation itself is useful because the frequency the sound gets played corresponds to the speed of the animation - increased movement speed equals faster movement sound. To add sound to the animation (in this case swimming audio), simply play the animation back and add "play sound notifications" at the desired times. These notifications are keys that don't create movement goal points but rather trigger other actions - in this case audio. A sound was played starting at the apex of tail rotation for both sides.
- Fish: As it can be taxing to position individual fish within the level, schools of fish were generated using an actor object. The particular class of fish generated is random but within each school there will always be *one* leader and a random amount of followers with unique speeds relative to the other fish to enhance realism. To simulate random movement (as opposed to having set target points for the leader fish to swim to), a series of nav meshes and hidden planes are positioned at various heights within the level. This pairing of nav mesh with plane is required as the function for selecting target points of navigation is reliant on collision with horizontal planar surfaces.
    - *Spawner*: The actor itself is merely a collision box that spawns actors at a random point within its bounds using the *SpawnActorAtLocation* and *GetBounds(ofActor)* method. The init function spawns one leader fish followed by a random amount of follow fish of class bluetuna or yellowtuna.
    - *Classes*: There are two general classes of fish - leader and follower - that extend the basic pawn class with added floating pawn movement. Because all the fish implement floating pawn movement, their movement isn't constricted to one plane due to gravity and can thus move up and down along the z axis as well. Both classes of fish have an arrow component, a static mesh (where the mesh itself remains undefined), and a collision sphere. The arrow component is important for making the fish point in the direction of movement, otherwise we could have sideways swimming fish! The collision sphere is useful for ensuring the fish won't swim into each other, effectively melding together, and triggering a scatter should the player bump into them - not implemented in the final product due to time constraints. The main difference between the fish lies in their movement function, both of which are recursive in nature and use the *AIMoveTo* method.
        - *Leader*: Uses the output of *GetRandomReachablePoint* as the target parameter.
        - *Follow*: Unique speed is assigned at initialization (spawn) by randomly setting the max speed to a number within a range. Movement itself is implemented by setting the target parameter to an actor (leader fish). To
        - ensure the fish doesn't follow the leader of a different school, the target is set through a variable at spawn and remains unaltered from that point on. A simple call to *GetActorsOfType* returns an array of all current references to leader fish in spawn order. The target variable is then set to the actor at the end of the array. Using the *AIMoveTo* method, unnaturally large rotation changes can occur (i.e. fish completes 90 degree turn instantly). To address this issue, you can use lerping and mathematically calculate the rotation changes. Due to time constraints, a fully working implementation isn't placed within the level for the final project, but the basics can be found in the blueprint file for "testfish".

        - *BlueTuna/YellowTuna (Follow)*: These classes are extensions of the FollowFish class (set class to FollowFish). These merely set the skeleton defined in the parent class to a particular fish model. In other words, they are FollowFish you can actually see within the level when spawned.
- Sand Material: The original sand material was imported from an online open-source ocean project. The material, however, was designed for a landscape of specific proportions. Placing it upon the surface of an actor of different scale caused massive compression and stretching: the material tessellated the surface creating large flashing spikes as the system couldn't render it properly or appeared as large glossy brown cubes. In other words, the material was designed to be placed upon a whole surface (conglomeration of polygons) as opposed to representing just one polygon on a surface. If the material is drawn on every single polygon as opposed to the surface as a whole, it won't have nearly as much material stretching. Therefore, scaling the object doesn't affect how it appears visually in terms of pattern/color. To do this, a scaling factor for micro and macro tiling is multiplied by a texture coordinate that eventual feeds into the texture itself. The specular parameter was also multiplied by a scaling factor. This reflectiveness gives the illusion of a wet surface.Visual Effects
    - *Water Caustics:* The water caustics effect is simple to implement; merely create a material and set it as the light function for a spotlight within the level. This material must be of light function domain for it to work properly. Simply pan three instances of the same caustics texture, creating offsets through addition and multiplication of static numbers to the vector output of the pan. To ensure the caustics are visible enough within the level, create an intensity parameter within the material. This, in conjunction with the intensity parameter for spotlights should be sufficient.
    - *Post-Process Effects:* Visual effects encompassing the entire camera view but not intended to trigger events or affect collision/physics within the level can be done through Post-Process Volumes. These volumes are essentially blendable camera layers that only alters what the camera sees and NOT anything within the level itself. Make sure that the rendering settings for the particular level you wish to have this volume in has Post-ProcessVolume set to a higher degree of all the others. Otherwise, some effects mainly the bloom and flare won't be visible within the level.
        - *Blur:* The goal of blurring is to simulate the distortion that occurs underwater - separate of object movement like seaweed. The blur is achieved through texture panning. Regardless of whether the texture is colored or has alpha, the texture is made to be completely translucent by setting the material domain to Post-Process and adding to the materials array within the post process volume rendering features. This effect isn't implemented within the level as the combined movement from the distortion and individual assets (i.e. seaweed, coral, fish, ocean waves, etc.) was a bit much and sometimes caused slight queasiness.
        - *Scene Tint:* Tinting the whole scene slightly turquoise adds to the realism of being on the ocean floor. This happens to be a parameter of the volume itself thereby allowing direct alteration without the need of creating a material.
        - *Bloom & Flare:* Light from the sun is never distinctly circular when stared at directly, rather it is circular in nature with a halo of fading light from the center.

The bloom effect creates that halo. Lens flare is normally associated with cameras as it simulates the scattering of light when viewing bright objects. Although this effect doesn't occur to the naked eye, it adds to the realism and is often used in video games. Both of these effects are parameters of the volume itself and thus can be changed directly

Walk Into Nothingness

This level is called "Walk into Nothingness" which is basically what you have to do. In this level we wanted to see if players would get motion sick if they were walking on nothing. So how we did this was to remove all lighting and external references people add by deleting: Atmospheric Fog, Light Source and the Sky Sphere. We added a platform for the user to stand on then a useless door which was just the model of the door without any code hooked up. The reason for the useless door is to give players hope for a second that they can go back but then realize that the door doesn't work and they have to face their fear of walking on nothing. Since the MotionControllerPawn is affected by gravity we had to add an invisible platform as the floor so the MotionControllerPawn does fall down to oblivion. One other thing that we had to do was to add invisible vertical planes at the end of the platform so that it forces players to physically walk to the door. Placing planes prohibits any movement from the MotionControllerPawn via controller input but still allows for the player to physically walk.

Closing Remarks

VR Projects are fun! The goal of the tutorial was to create a functional and fun VR game that could be replicated or added upon by future students. Feel free to expand on this project and create new levels.