Virtual Reality Multiplayer in Unreal Engine 5 with C++

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Bachelor thesis

Degree Programme in Business Information Technology
autumn, 2022

Roope Pennanen

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tietojenkäsittelyn koulutus		Tiivistelmä
Tekijä	Roope Pennanen	Vuosi 2022
Työn nimi	Unreal Engine 5:n VR moninpeli C++ kielellä	
Ohjaaja	Tommi Lahti	

TIIVISTELMÄ

Opinnäytetyön tavoitteena on antaa käyttäjille perusohjeet virtuaalitodellisuuteen Unreal Engine 5:ssa. Työssä tehdään virtuaalitodellisuusympäristö, jossa ainakin kaksi eri käyttäjää pystyy olemaan vuorovaikutuksessa toistensa kanssa. Työssä näytetään mitä ominaisuuksia virtuaalitodellisuudessa olisi hyvä olla, sekä miten yksinkertainen moninpeliominaisuus toteutetaan.

Lukijalta oletetaan perusymmärrystä C++-kielestä sekä Unreal Engine -alustasta. Työssä kerrotaan tarpeelliset tiedot virtuaalitodellisuudesta sekä moninpelistä, minkä jälkeen selitetään käyttännönosassa tehty demo. Työn rakenne on seuraavanlainen: erityyppiset liikkumisvaihtoehdot, VR-pahoinvoinnin vähentäminen, käyttöliittymä ja lopuksi moninpelin lisääminen.

Valmis työ on mahdollista nähdä linkitetyn videon kautta. Työ on tehty Unreal Engine 5 Early Access -versiossa, mutta on suositeltavaa odottaa vakaata versiota.

Avainsanat Virtuaalitodellisuus, Unreal Engine, Moninpeli

Sivut 51 sivua ja liitteitä 9 sivua

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Degree Programme in Business Information Technology		Abstract
Author	Roope Pennanen	Year 2022
Subject	VR Multiplayer in Unreal Engine 5 with C++	
Supervisors	Tommi Lahti	

ABSTRACT

The goal of this thesis is to give general directions to Unreal Engine users to create a simple virtual environment where at least two different users can interact with each other. One player uses a virtual reality character while others use third-person characters. The thesis will show what features should be implemented for a VR environment, how to make it into a multiplayer, as well as what to take into consideration when creating a VR game.

The reader should have a basic understanding of the C++ language as well as knowledge of the Unreal Engine framework. The thesis will demonstrate the necessary theory of virtual reality and multiplayer aspects of this thesis, then proceed with the implementation of the project. The structure of the thesis is as follows: Different types of VR character movement, reducing VR sickness, UI for VR, and finally adding the networking aspects of the project.

The final product can be found in the linked video in Chapter 6 that showcases the finished project. This project was done with the early access version of unreal engine 5, but it is recommended to wait for the full release, as some of the bugs will have been fixed.

Keywords Virtual Reality, Unreal Engine, Multiplayer

Pages 51 pages and appendices 9 pages

Glossary

C++	Low level programming language used by Unreal Engine.
Epic Games	Company responsible for the development of Unreal Engine
Haptics	Technology that gives the user a sense of touch by applying different types of forces.
Macro	Pattern that is replaced by something else.
Pawn	Subclass of actor that can receive inputs.
Replication	An aspect of networking, where actor or component data is synced with clients.
UI	User Interface
VR	Virtual Reality

Contents

Figures

Tables

Annexes

1 Introduction

This thesis was conducted to give the basic information of creating a virtual reality multiplayer environment in Unreal Engine. There are little to no existing guides on the subject publicly available. There are guides for virtual reality and networking in Unreal Engine separately, but no known guides that combine both with C++. The project was heavily influenced by a Udemy course (GameDev.tv team).

The project was done with the Unreal Engine 5 early access version, due to its relevancy at the time of writing this thesis. Unreal Engine 5 was chosen due to its potential to change multiple industries with the new technologies it brings. Nanite is by far the most important for virtual reality as it would allow orders of magnitude more polygons to work with compared to other game engines. Unfortunately, the early access version does not support nanite for virtual reality, but according to official statements from Epic Games, nanite should be ready for VR with the final release of the engine.

As the thesis is more focused on virtual reality and multiplayer, some features, like the 3rd person character implementation, will only be discussed very briefly in this thesis, as it is not relevant.

Research questions

- What features should be included in virtual reality?
- How to implement simple multiplayer in Unreal Engine 5?
- What should be taken into consideration when creating a VR game?

2 Introduction to Unreal Engine 5

This chapter gives a short and very general information about the engine and the required hardware and software. While this information should be known to the reader, the system specifications and software specifications have changed over time.

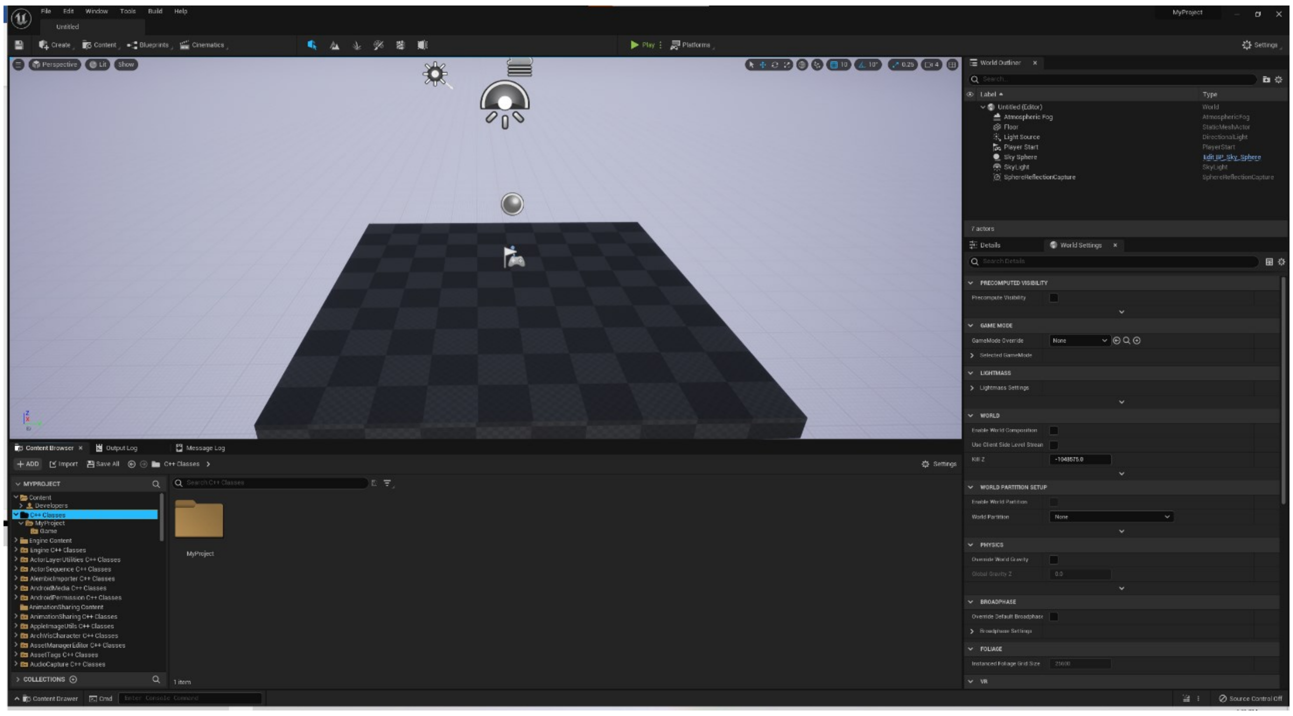
2.1 Basics

Unreal Engine 5 is the upcoming version of Unreal Engine, as it will be an upgrade from the previous Unreal Engine 4 in several ways. Unreal Engine is a proprietary game engine made by Epic Games and has been free to use since 2015 with source code available on GitHub. Unreal utilizes Blueprints visual scripting and C++ programming. C++ language used by Unreal Engine is much more human-readable than standard C++.

2.2 What's new?

The thesis won't be going into much detail about the changes in this thesis, since most of the new features will not be used. The only relevant new feature for this project is MetaSounds, which we will be using for the sound effects, which will be replicated for clients. Editor has seen some visual changes, to give it a more modern look as seen in Figure 1.

Figure 1. Unreal Engine 5 Editor.



2.3 Hardware

Virtual reality hardware used in this project is a Valve Index Head Mounted Display, two Valve Index Motion Controllers and two Valve Index Base Stations. Unreal Engine supports a variety of different virtual reality platforms. These platforms include Oculus VR, SteamVR, Windows Mixed Reality, Samsung Gear VR and Google VR (VR Platforms). Unreal Engine also has minimum and recommended system requirements for its editor. Recommended hardware is listed as follows: Processor: Quad-core Intel or AMD, 2.5 GHz or faster, Memory: 8 GB RAM, Video Card/DirectX Version: DirectX 11 or DirectX 12 compatible graphics card (Hardware and software specifications).

2.4 Software

For DirectX versions, DirectX 11 or DirectX 12. The minimum Windows version is Windows 7, and recommended version is Windows 10 64-bit. IDE used in this project is *Rider for Unreal Engine*

2021.1.3. The thesis won’t be covering IDE set-up, but Epic has documentation for setting up Visual Studio for Unreal Engine if interested. We will be using SteamVR in this project, but the required software may vary depending on the hardware used for virtual reality. We will be using the early access (EA) version of Unreal Engine 5.0.0 for this project. It is also recommended to make sure your graphics card drivers are up to date.

3 Virtual Reality in Unreal Engine

Virtual reality sickness is a serious issue for any virtual reality application, which needs to be fixed accordingly. The thesis will go into some depth as to how to reduce virtual sickness. In this project, SteamVR and OculusVR plugins are used. Unreal Engine also has OpenXR support, which is another virtual reality platform. We will not use OpenXR for this project due to the possible problems, since OpenXR is in a beta state for Unreal Engine 5 early access. Basics of Virtual Reality

3.1 Basics of Virtual Reality

Virtual reality is a simulated experience where the user receives artificially created inputs, mostly visual and audio, within a virtual environment. Most commonly used VR systems use a head-mounted display for visual, as well as audio inputs. The head-mounted display has two screens, one for each eye. Each image on the screen is rendered separately to give a sense of depth, in order to fool the user into thinking they are in a different environment. Many virtual reality systems require some types of motion controllers to be used for user input. These motion controllers often have built-in haptic feedback, which gives the user a sense of touch, usually by different types of vibrations. Some systems also need base stations to function properly. These base stations are devices that are set up in a room that track the movement of your headset and controllers.

3.2 Virtual Reality Sickness

Virtual reality sickness is a common form of sickness when using a head-mounted display. Symptoms are very similar to that of motion sickness, which a person might experience during car or sea travel. Common symptoms include nausea, dizziness, headaches, and disorientation. (Eunhee Chang et al., 2020)

3.2.1 Causes of Sickness

The underlying reason for VR sickness is conflicting sensory information. There are three senses that help with sensing movement. These senses are the visual, vestibular, and proprio sensory system. The visual system is simply what you can see with your eyes, vestibular being your inner ear, which helps to determine orientation and acceleration, and the proprio sensory, which is all your nerves working together to determine location of your body (Reason, 1975). In a VR environment, often times the user can see movement, but none of the other systems are detecting movement, meaning that the systems are in conflict with each other. This conflict is the main reason for feeling sick, but some users might fair better due to biological differences. Hardware can also be a contributing factor in feeling sick, as low resolution and low refresh rate displays can increase the feeling of sickness (Eunhee Chang et al., 2020).

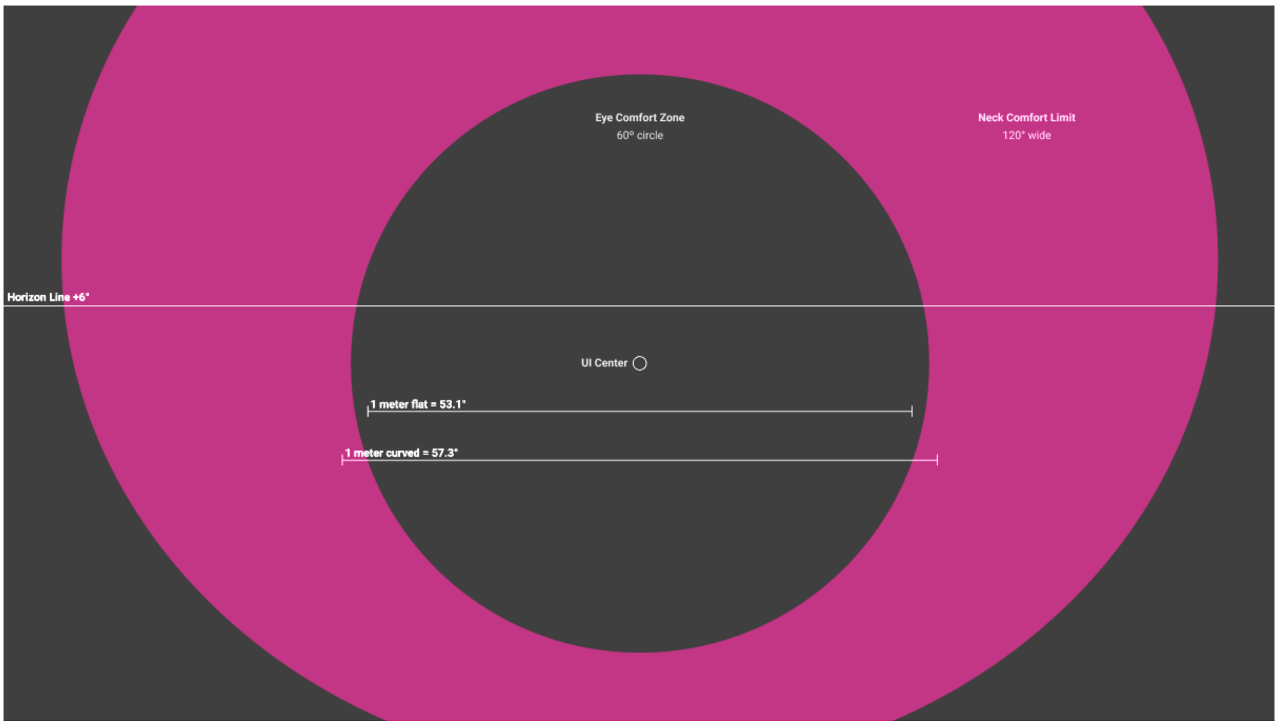
3.2.2 Techniques to reduce sickness

There are many ways to reduce possible sickness, but each technique comes with some drawbacks, so it is up to the developer to figure out the best techniques for their use case. As there are many different ways to reduce sickness, we will only mention techniques that are relevant to this project. Teleporting is a way to achieve movement in-game while not creating conflicts with the vestibular or the proprio sensory systems. The visual system is fooled by fading in and out during the teleportation process. As a downside, this technique can be immersion-breaking, so depending on the project it might not be suitable. Climbing can be used to move vertically, but it is unsuitable for most cases. It has the upside of simulating hand movement and having your hand as a reference point, thus fooling the visual and proprio sensory systems. The vestibular system in this case is ignored. Lastly, there are blinkers, which are dark areas around the user’s peripheral vision that block visual input so that the user perceives less motions. Blinkers are used when the user is moving fast and can be programmed to increase in size depending on the speed of the user. The biggest problems with this technique are the limited vision and limiting the user to only moving in the direction of the head. (GameDev.tv team n.d.)

3.3 Graphical User Interface in Virtual Reality

For the virtual reality graphical user interface, we will be using the Google Daydream sticker spreadsheet guide (Google Daydream). Google Daydream uses dmm as a measurement. 1 dmm is equal to 1 mm at a 1-meter distance, and 1 dmm is equal to 1 pixel. The dmm unit was created to make it easier to have UIs that scale correctly no matter the distance from the user.
The VR UI should be positioned 6 degrees downwards as people naturally look slightly downwards. The color of the UIs background should not be pure white, since it can be extremely bright for some users. Google suggests using #EEEEEE for light backgrounds and #212121 for darker backgrounds. Pure white is acceptable for texts and other elements that need to pop.
There exists comfort zones for both the eyes and neck that need to be taken into consideration when creating the UI in VR, as seen in Figure 2. For the eyes, it's a circle that is approximately 55 degrees in diameter, and for the neck, it is about 120 degrees wide.

Figure 2. Google Daydream comfort zones. (Google Daydream)



3.4 Unreal Engine settings for Virtual Reality

In this chapter, we will discuss settings that should be enabled or disabled in the Unreal Engine 5 EA build. Note that this information might change after the full release of the engine.
As for lighting, lumen needs to be disabled from the project settings. The current version of Unreal Engine 5 does not support lumen in virtual reality. This is done by going to **Edit > Project Settings > Rendering > Global Illumination**, and setting Dynamic Global Illumination Method to None. It is also required to change the reflection method, by going to **Edit > Project Settings > Rendering > Reflections**, and setting Reflection Method to None. It can be noted that the dropdown list has two options that are not recommended for virtual reality. Raytracing is marked as deprecated, but it can technically work for virtual reality, but it must be noted that it is extremely resource-heavy. As has been previously mentioned, achieving a stable and comfortable framerate of about 90 frames per second or more is recommended with most devices, meaning that unnecessary resource-heavy operations should be avoided. The second possible option is Screen Space, which is also not recommended due to known issues with it. Screen space reflections and screen space global illuminations have known issues of showing discrepancies between the displays for the two eyes.

Unreal recommends using Forward Renderer and enabling Instanced Stereo for virtual reality. Enable forward shading from **Edit > Project Settings > Rendering > Forward Renderer**, and enable Forward Shading. Forward rendering is preferred in virtual reality, due to it being faster than deferred rendering in most cases, and it provides better anti-aliasing options. Instanced Stereo can be enabled from **Edit > Project Settings > Rendering > VR**. Instanced Stereo can improve performance in virtual reality by allowing parallel rendering for both eyes. (Virtual reality best practices)

4 Multiplayer in Unreal Engine

Unreal Engine has been made with multiplayer in mind. The multiplayer is based on the server-client model, meaning that the server is the authority and the clients experience what the server experiences in most cases. To enable multiplayer, simply press the three vertical dots next to the preview button. Go down the list and set the player count to two or more.

4.1 Types of Multiplayer

For the net mode options there are standalone, listen server, and client. Standalone just means singleplayer mode, so even if you have player count set to 2 or more, the engine only loads more instances of the game without networking enabled. We will be using a listen server, which is where one of the players is the server and the client at the same time.

4.2 Replication

Replication is the process of replicating actors or components. Unreal Engine replicates basic character movement as default, but for others, it must be enabled manually. To mark an actor for replication, simply add bReplicates = true; in the actor’s constructor. Alternatively it is possible to use the following syntax to enable replication: Actor->SetReplicates(true);.

4.2.1 What to replicate

As bandwidth is a limited resource, we must only replicate what is needed. Relevant transform data, spawning of actors, audio and visual effects are usually important to replicate, but it depends heavily on the project. As a general rule, replication must be done so that the server and the client are always in sync with each other. In our project, the most important objects that need to be replicated are characters and their subobjects, splines created by the light painter, as well as chat.

4.2.2 What should not be replicated

As a general rule anything that other players shouldn't experience, should not be replicated. These usually include the user interface and certain types of audio. In this project, we will not be replicating the UI or haptic effects.

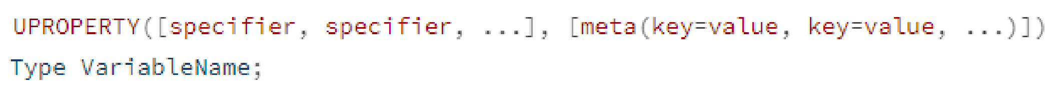
4.2.3 UPROPERTIES for multiplayer

UPROPERTY() is a macro that is used before declaring a property. Inside the UPROPERTY() you can add property specifiers that define the behavior of that property. For multiplayer purposes, the specifiers that are of interest are shown in Table 1. The description is taken from the official documentation (Property Specifiers). You can also see the UPROPERTY syntax in Figure 3, but it is noteworthy that you can leave the specifiers empty. Leaving the UPROPERTY specifier empty will still register the property to the garbage collection system.

Table 1. Property specifiers

Specifier	**Description**
NotReplicated	Skip replication. This only applies to struct members and parameters in service request functions.
Replicated	The property should be replicated over the network.
ReplicatedUsing=FunctionName	The ReplicatedUsing Specifier specifies a callback function which is executed when the property is updated over the network.

Figure 3. UPROPERTY syntax



```
UPROPERTY([specifier, specifier, ...], [meta(key=value, key=value, ...)])
Type VariableName;
```

4.2.4 UFUNCTIONS for multiplayer

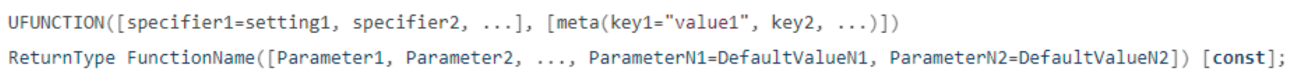
UFUNCTION() is a macro that can be used before declaring a function. Inside the UFUNCTION() you can add a function specifier to change the behavior of the function in various ways. For multiplayer purposes the specifiers that are of interest are shown in Table 2. Descriptions are taken from the official documentation (Function Specifiers). The syntax for the UFUNCTION specifiers can be seen in Figure 4.

Table 2. Function specifiers (Continues)

Specifier	**Description**
Client	The function is only executed on the client that owns the Object on which the function is called. Declares an additional function named the same as the main function, but with *_Implementation* added to the end. The autogenerated code will call the *_Implementation* method when necessary.
NetMulticast	The function is executed both locally on the server, and replicated to all clients, regardless of the Actor's NetOwner.
Reliable	The function is replicated over the network, and is guaranteed to arrive regardless of bandwidth or network errors. Only valid when used in conjunction with *Client* or *Server*
Server	The function is only executed on the server. Declares an additional function named the same as the main function, but with *_Implementation* added to the end, which is where the code should be written. The

	autogenerated code will call the *_Implementation* method when necessary.
Unreliable	The function is replicated over the network but can fail due to bandwidth limitations or network errors. Only valid when used in conjunction with *Client* or *Server*
WithValidation	Declares an additional function named the same as the main function, but with *_Validate* added to the end. This function takes the same parameters, and returns a bool to indicate wheter or not the call to the main function should proceed

Figure 4. UFUNCTION syntax



```
UFUNCTION([specifier1=setting1, specifier2, ...], [meta(key1="value1", key2, ...)])
ReturnType FunctionName([Parameter1, Parameter2, ..., ParameterN1=DefaultValueN1, ParameterN2=DefaultValueN2]) [const];
```

5 Project Implementation

The idea is to have a demo where at least two different players can interact with each other in a virtual world. One player will be the host and control a virtual reality character, while the clients joining will be using a 3rd person character. There is only one virtual reality character due to the lack of equipment. Players can see others and interact with each other by drawing or using the chat. The 3rd person character is imported from an official template.

5.1 Features

The virtual reality character can move, teleport and climb in the world. It can draw with the right motion controller when pressing a specified button. Blinkers will be used when moving and fading when teleporting. The user can also open the in-game menu and quit the session.

5.2 Remarks

Some of the issues are fixed later in the thesis in chapter 5.5. There are also known issues that aren’t addressed in this thesis due to time limitations or bugs with the engine. Every C++ class we create that will have a UstaticMeshComponent, will have a blueprint version of it. Simply right-click the class and select “**Create Blueprint class based on x**”. Notice that all blueprint classes should have BP_ as the prefix. You also need to go to your .build.cs file and add “NavigationSystem” and “HeadMountedDisplay” to the PublicDependencyModuleNames list.

5.3 Virtual Reality Features

The essential features that will be implemented in this chapter are character movement, Lightpainter, VR sickness reduction the UI. As mentioned before, the movement section consists of normal movement, climbing, and teleportation.

5.3.1 Character and Controllers

The first thing to do is to create a character for the virtual reality user. Create a new C++ class and derive it from a Character. We also need classes for the two types of controllers we will be using. We'll create a custom HandControllerBase class seen in Program Code 1 which can be used as the parent class for the normal handcontroller class and the UI handcontroller class. First, include the XRMotionControllerBase header, then in the constructor of the base class, we will create the UmotionControllerComponent subobject, set the root component, create a StaticMeshComponent, and then attach it to the MotionControllerComponent. We do not use the default device model mesh due to a bug with replication. We also need to create a function that is called from the VRCharacter class that sets the tracking source for each controller as seen in Program Code 2.

Program Code 1. AHandControllerBase().

```
AHandControllerBase::AHandControllerBase()
{
	PrimaryActorTick.bCanEverTick = false;
	bReplicates = true;
	MotionControllerComponent = 
CreateDefaultSubobject<UMotionControllerComponent>
(TEXT("MotionControllerComponent"));
	SetRootComponent(MotionControllerComponent);
	MotionControllerComponent->SetShowDeviceModel(false);

	SM_Component = CreateDefaultSubobject<UStaticMeshComponent>
(TEXT("StaticMesh"));
	SM_Component->SetupAttachment(MotionControllerComponent);

}
```

Program Code 2. SetHand function.

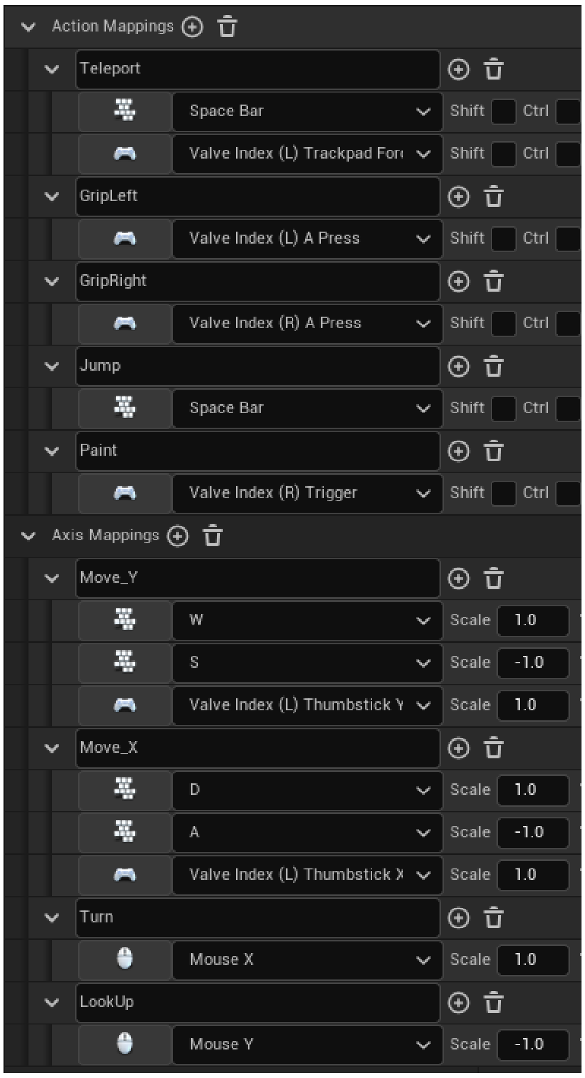
```
void AHandControllerBase::SetHand(FName Side)
{
	if(Side == "Left")
	{
		MotionControllerComponent->
SetTrackingMotionSource(FXRMotionControllerBase::LeftHandSourceId);
	}else if(Side == "Right")
	{
		MotionControllerComponent->
SetTrackingMotionSource(FXRMotionControllerBase::RightHandSourceId);
	}

}
```

5.3.2 Movement

The virtual reality character can move by moving the left joystick in any direction. It is also possible to move using the WASD keys on a keyboard. Teleporting is done by using by aiming with the left controller and pressing on the left trackpad to confirm the teleport location.

We need to add input and axis mappings for the character. Go to **Edit > Project Settings > Engine > Input**, and add action mappings for Teleport Grip Left, Grip Right, and Paint. We also need to add axis mappings for moving in the Y and X-axis, as well as turning and looking up for the 3rd person character as seen in Figure 5.

Figure 5. Input Mappings



```
Action Mappings
  Teleport
    Space Bar                          Shift  Ctrl
    Valve Index (L) Trackpad For       Shift  Ctrl
  GripLeft
    Valve Index (L) A Press            Shift  Ctrl
  GripRight
    Valve Index (R) A Press            Shift  Ctrl
  Jump
    Space Bar                          Shift  Ctrl
  Paint
    Valve Index (R) Trigger            Shift  Ctrl
Axis Mappings
  Move_Y
    W                                  Scale  1.0
    S                                  Scale  -1.0
    Valve Index (L) Thumbstick Y       Scale  1.0
  Move_X
    D                                  Scale  1.0
    A                                  Scale  -1.0
    Valve Index (L) Thumbstick X       Scale  1.0
  Turn
    Mouse X                            Scale  1.0
  LookUp
    Mouse Y                            Scale  -1.0
```

First, the normal movement for the character will be added. The VRCharacter can use either WASD or the motion controller to move. In the VRCharacter.cpp there is a SetupPlayerInputComponent function in which we will bind a delegate function to an action that we just defined in the editor input mappings. This means we need to create the functions for the two-movement axis. Notice that the Text parameter needs to be the same as the one you wrote in the axis mappings in the editor as shown both in Figure 5 and in Program Code 3.

Program Code 3. Binding movement axes.

```
PlayerInputComponent->BindAxis(TEXT("Move_Y"), this,
&AVRCharacter::MoveForward);
PlayerInputComponent->BindAxis(TEXT("Move_X"), this,
&AVRCharacter::MoveRight);
```

It is noteworthy that AddMovementInput function handles the movement automatically with characters, but not with base pawn classes. The Throttle parameter changes from positive to negative depending on the user input as shown in Program Code 4.

Program Code 4. Movement functions.

```
void AVRCharacter::MoveForward(const float Throttle)
{
	AddMovementInput(Throttle * Camera->GetForwardVector());
}

void AVRCharacter::MoveRight(const float Throttle)
{
	AddMovementInput(Throttle * Camera->GetRightVector());
}
```

Next, we’ll implement climbing for our virtual reality character. In the AVRCharacter::BeginPlay() we’ll spawn the controllers. HandControllerClass contains the blueprint version of the HandController.

We’ll attach each controller to the root component, Call the SetHand function that is in the HandControllerBase class with the corresponding side, and set “this” as the owner. We also pair both controllers, so that they have a reference to each other. (Annex 1)

For climbing, we need to have four functions to handle gripping and releasing for both hands as seen in Program Code 5. The keyaction needs to be IE_Pressed for the gripping functions and IE_Released for the release functions.

Program Code 5. Binding grip functions.

```
PlayerInputComponent->BindAction(TEXT("GripRight"), IE_Pressed, this,
&AVRCharacter::GripRight);
PlayerInputComponent->BindAction(TEXT("GripLeft"), IE_Pressed, this,
&AVRCharacter::GripLeft);
PlayerInputComponent->BindAction(TEXT("GripLeft"), IE_Released, this,
&AVRCharacter::ReleaseLeft);
PlayerInputComponent->BindAction(TEXT("GripRight"), IE_Released,
this, &AVRCharacter::ReleaseRight);
```

We'll use inline functions in the VRCharacter header file and move the gripping and releasing logic to the HandController class as shown in Program Code 6.

Program Code 6. Gripping inline functions.

```
void GripLeft() { LeftController->Grip(); }
void ReleaseLeft() { LeftController->Release(); }
void GripRight() { RightController->Grip(); }
void ReleaseRight() { RightController->Release(); }
```

In the HandController header file, we will add the necessary functions and properties for climbing. We need to get a reference to the other controller so that we can determine whether or not we are climbing or not, and to make it so that you can only grip with one controller at a time. The HapticEffect will be used when the controller overlaps with an actor that is tagged as Climbable. We also need functions to handle logic when we begin and end overlapping. (Annex 2)

In the cpp file we first create a function that returns a boolean which tells us whether or not we can climb or not as shown in Program Code 7. We create an array of AActor* that we put into the GetOverlappingActors function, which is part of the actor class. Actors get put into the array, which we then iterate through and return true if any of the actors have the tag "Climbable". This means that in the editor, we need to create an actor with the aforementioned tag. When in the blueprint editor, the tags can be found in **Details > Actor > Tags**.

Program Code 7. CanClimb function.

```
bool AHandController::CanClimb() const
{
	TArray<AActor*> OverlappingActors;
	GetOverlappingActors(OverlappingActors);
	for (AActor* OverlappingActor : OverlappingActors)
	{
		if (OverlappingActor->ActorHasTag(TEXT("Climbable")))
		{
			return true;
		}
	}

	return false;
}
```

In the AhandController::BeginPlay() we will add two dynamic functions that get called OnActorBeginOverlap and OnActorEndOverlap as seen in Program Code 8.

Program Code 8. Handcontroller overlap function bindings.

```
OnActorBeginOverlap.AddDynamic(this,
&AHandController::ActorBeginOverlap);
OnActorEndOverlap.AddDynamic(this,
&AHandController::ActorEndOverlap);
```

In the ActorBeginOverlap function, we call the CanClimb function and play the haptic effect on the correct controller if the player is the host. In the ActorEndOverlap we check if the other controller is still overlapping with a climbable object as shown in Program Code 9.

Program Code 9. HandController overlap function implementations.

```
void AHandController::ActorBeginOverlap(AActor* OverlappedActor,
AActor* OtherActor)
{
	bool bNewCanClimb = CanClimb();
	if (!bCanClimb && bNewCanClimb)
	{
		AVRCharacter* VRC = Cast<AVRCharacter>
(GetAttachParentActor());
		if(HasAuthority() && VRC && HapticEffect)
		{
			UGameplayStatics::GetPlayerController(this,0)->
PlayHapticEffect(HapticEffect, MotionControllerComponent->
GetTrackingSource());
		}
	}
	bCanClimb = bNewCanClimb;
}

void AHandController::ActorEndOverlap(AActor* OverlappedActor,
AActor* OtherActor)
{
	bCanClimb = CanClimb();
}
```

In the Grip function, as seen in Program Code 10, we first check if the player can climb and if the player is already climbing. We then set the current controllers bIsClimbing boolean to true and the other controllers to false. We save the starting location for later use and set the character’s movement mode to MOVE_Flying. Teleporting should be disabled while the player is climbing.

Program Code 10. Grip function.

```
void AHandController::Grip()
{
	if(!bCanClimb) return;
	if(!bIsClimbing)
	{
		bIsClimbing = true;
		SecondController->bIsClimbing = false;
		ClimbingStartLoc = GetActorLocation();
		ACharacter* Character = Cast<ACharacter>
(GetAttachParentActor());
		if(Character)
		{
			Character->GetCharacterMovement()->
SetMovementMode(MOVE_Flying);
		}
		AVRCharacter* Chara = Cast<AVRCharacter>
(GetAttachParentActor());
		if(Chara)
		{
			Chara->bCanTeleport = false;
		}
	}
}
```

In Program Code 11, bIsClimbing is set to false and movement mode is set back to MOVE_Falling.

Program Code 11. Release function.

```
void AHandController::Release()
{
	if(bIsClimbing)
	{
		bIsClimbing = false;
		ACharacter* Character = Cast<ACharacter>
(GetAttachParentActor());
		if(Character)
		{
			Character->GetCharacterMovement()->
SetMovementMode(MOVE_Falling);
		}
		if(!SecondController->bIsClimbing)
		{
			AVRCharacter* Chara = Cast<AVRCharacter>
(GetAttachParentActor());
			if(Chara)
			{
				Chara->bCanTeleport = true;
			}
		}
	}
}
```

The actual moving happens in the Tick function where we subtract the climbing start location from the actor location in order to get the controller movement vector. Then we add a world offset to the AVRCharacter that is opposite to the controller movement vector as shown in Program Code 12.

Program Code 12. Controller offset for climbing.

```
if(bIsClimbing)
{
	FVector HandControllerDelta = GetActorLocation() - 
ClimbingStartLoc;
	GetAttachParentActor()->AddActorWorldOffset(-
HandControllerDelta);
}
```

Teleportation will be the last movement type to be implemented. With teleportation, there will be a cylinder that we move by aiming the left controller, and there will be a dynamic spline that starts from the controller and goes into the cylinder forming an arch. This arch helps at aiming the cylinder. The cylinder location is used to determine the teleportation location. The location needs to be in the Navigation mesh. First, we'll add the binding in the AVRCharacter::SetupPlayerInputComponent as shown in Program Code 13.

Program Code 13. Binding TeleportPlayer function.

```
PlayerInputComponent->BindAction(TEXT("Teleport"), IE_Pressed, this,
&AVRCharacter::TeleportPlayer);
```

Next, we'll look at the things we need to add in the header file for teleportation (Annex 3). We need a mesh for both the cylinder and the spline and variables for teleportation calculations.

The cylinder will be updated every tick while the user can teleport and is not currently teleporting. In the UpdateTeleportCylinder function, a TArray of vectors is created to hold the path information and a single vector for holding the destination as shown in Program Code 14. These are then passed to the FindTeleportDestination function.

Program Code 14. UpdateTeleportCylinder function.

```
void AVRCharacter::UpdateTeleportCylinder()
{
	FVector Navloc;
	TArray<FVector> Path;
	bool bHasValidDestination = FindTeleportDestination(Path, 
Navloc);
	if (bHasValidDestination && bCanTeleport)
	{
		TeleporterCylinder->SetVisibility(true);
		TeleporterCylinder->SetWorldLocation(Navloc);
		DrawTeleportPath(Path);
	}
	else
	{
		TeleporterCylinder->SetVisibility(false);
		TArray<FVector> EmptyArray;
		DrawTeleportPath(EmptyArray);
	}
}
```

In the FindTeleportDestination function, we save the left controller location and forward vector so that we can use them in the projectile calculations for the teleportation arch. Firstly, the FPredictProjectilePathParams is created with the needed variables, some of which we set in the header file. We then use the UGameplayStatics::PredictProjectilePath function with the path parameters to get an FPredictProjectilePathResult type, which we will then use to fill the passed in OutPath TArray and the Location vector. Finally returning a boolean to signal whether cylinder is in a valid location. (Annex 4)

If the cylinder has a valid location and the user is able to teleport, then the cylinder visibility is set to true, and it is drawn with the information from the FindTeleportationDestination function.

In order to draw the trace, we first need to update the teleport spline, which is done in the UpdateTeleportSpline function.

Whenever UpdateTeleportCylinder is called we are calling the DrawTeleportPath function (Annex 5) and passing a reference to the array which we got from the FindTeleportDestination function. In order to draw the trace we first need to update the teleport spline, which is done in the UpdateTeleportSpline function as seen in Program Code 15.

Program Code 15. UpdateTeleportSpline function.

```
void AVRCharacter::UpdateTeleportSpline(const TArray<FVector>&
OutPath)
{
	TeleporterPath->ClearSplinePoints(false);
	FVector LocalPos;
	for (int32 i = 0; i < OutPath.Num(); i++)
	{
		LocalPos = TeleporterPath->
GetComponentTransform().InverseTransformPosition(OutPath[i]);
		TeleporterPath->AddPoint(FSplinePoint(i, LocalPos,
ESplinePointType::Curve), false);
	}
	TeleporterPath->UpdateSpline();
}
```

Lastly, the TeleportPlayer function needs to be created which is called whenever the player presses the designated teleport button. It checks whether the player can teleport and sets bTeleporting to true. Camera fade is initiated, and a timer is set so that after the fade-out has finished the FinishTeleport function is called.

In the FinishTeleport function, the player’s location is set to the TeleportCylinder. bTeleporting is set to false and the camera fade-in is initiated as shown in Program Code 16.

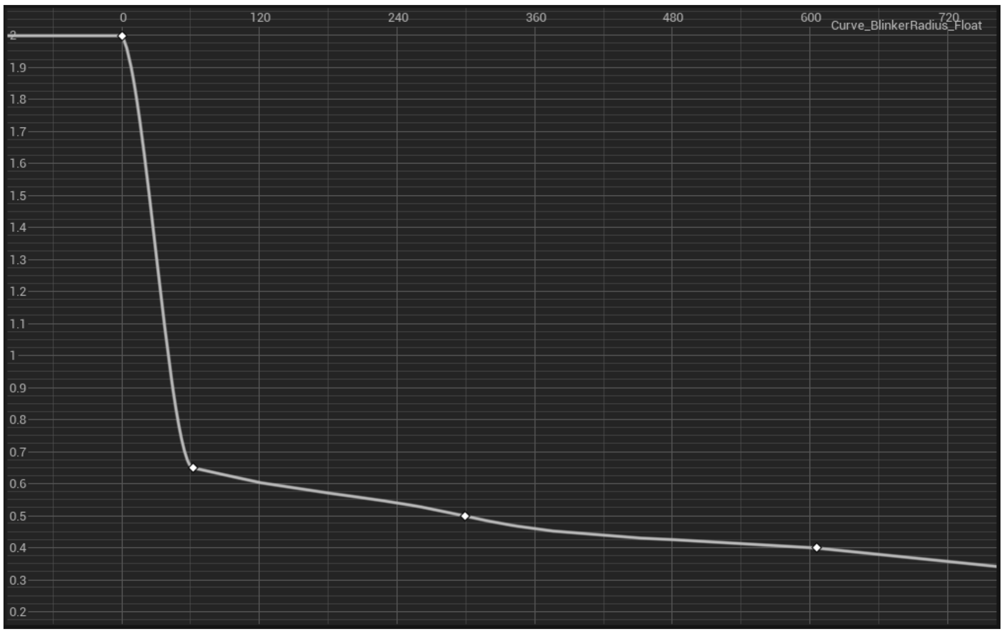
Program Code 16. FinishTeleport function.

```
void AVRCharacter::FinishTeleport()
{
	float Height = GetCapsuleComponent()->
GetScaledCapsuleHalfHeight();
	SetActorLocation(TeleporterCylinder->GetComponentLocation() +
FVector(0, 0, Height));
	if (PC != nullptr)
	{
		bTeleporting = false;
		PC->PlayerCameraManager->StartCameraFade(1, 0,
TeleportFadeTime, FLinearColor::Black);
	}
}
```

5.3.3 Reducing Motion Sickness

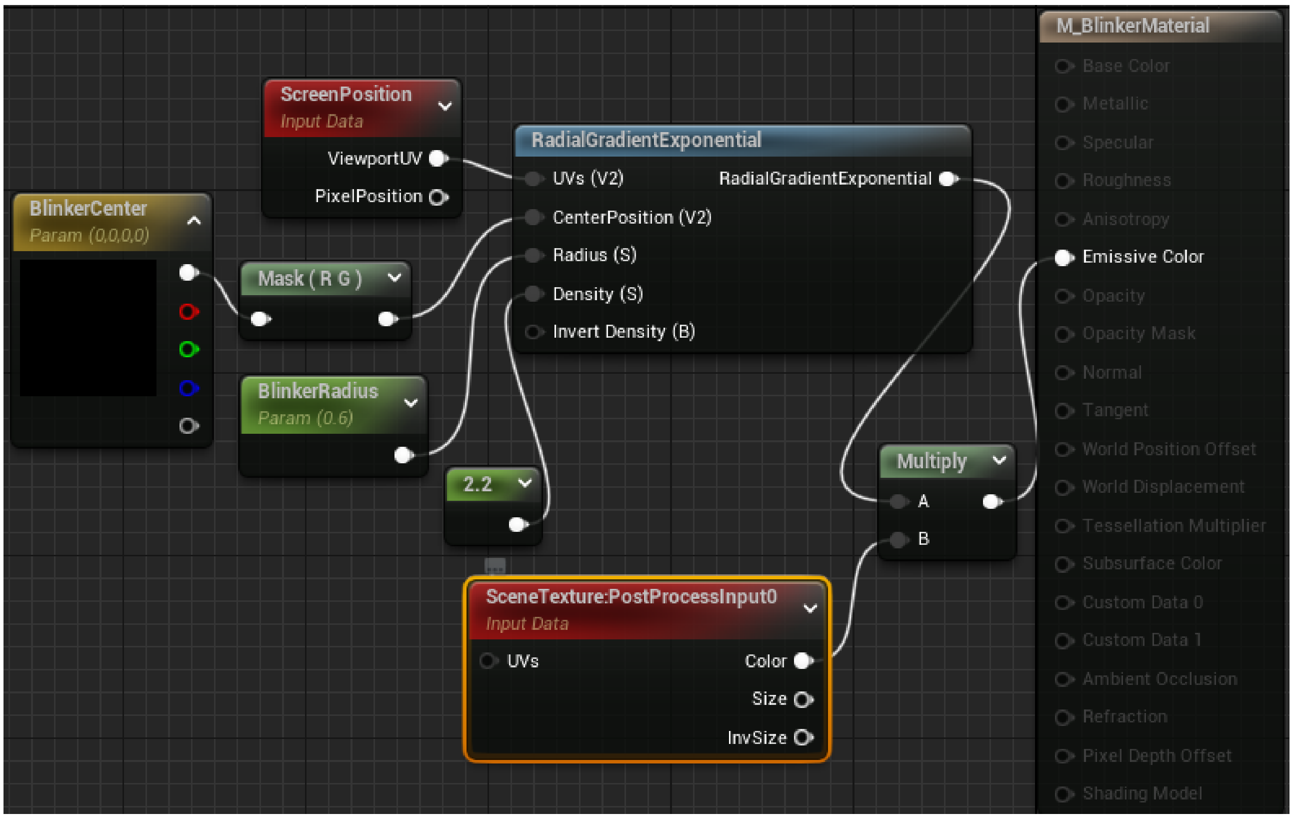
When moving, blinkers can be used to trick the user’s brain into thinking there is a lot less movement going on than without blinkers. This is done by using a float curve object that tells how much of the peripheral vision is blackened by a material assigned for the blinkers when moving. To add a float curve from the content browser, go **Miscellaneous > Curve > FloatCurve**. The naming convention for float curves advises using Curve as a prefix. You can see in Figure 6 that the radius in the y-axis gets smaller the bigger the velocity on the x-axis gets.

Figure 6. Float curve for blinker radius



We also need to create a dynamic material that we can edit on runtime like the one shown in Figure 7. According to Unreal naming conventions, all materials should have M_ as the prefix. Open the material editor by double-clicking the newly created material, and after that navigate to **Details > Material > Material Domain**, and select **Post Process** from the list.

Figure 7. Dynamic blinker material nodes



The UpdateBlinkers function is called from the AVRCharacter::Tick() and updated once per frame if enabled in the BP_VRCharacter.

The blinker material needs a vector value that represents the center. This value is calculated in GetBlinkerCenter function (Annex 6) which is called in the UpdateBlinkers function as seen in Program Code 17. The center is based on the player's movement speed and then projected into screenspace coordinates from world space coordinates.

Program Code 17. UpdateBlinkers function.

```
void AVRCharacter::UpdateBlinkers()
{
	if (RadiusVsVelocity == nullptr)
	{
		return;
	}
	float Speed = GetVelocity().Size();
	float Radius = RadiusVsVelocity->GetFloatValue(Speed);
	FVector2D Center = GetBlinkerCenter();
	MaterialInstanceDynamic->
SetVectorParameterValue(TEXT("BlinkerCenter"), FLinearColor(Center.X,
Center.Y, 0));
	if (Center.X < .3 || Center.X > .7) // If BlinkerCenter is too
far, increase radius
	{
		MaterialInstanceDynamic->
SetScalarParameterValue(TEXT("BlinkerRadius"), Radius * 2);
	}
	else
	{
		MaterialInstanceDynamic->
SetScalarParameterValue(TEXT("BlinkerRadius"), Radius);
	}
}
```

5.3.4 Light Painter

We need to create a class for the mesh that we create with the light painter. It will be made from splines that we create based on the location of the right controller while the trigger is pressed. The meshes are taken from the internet. The joint mesh is a 2 x 2 x 2 low polygon ball, and the Stroke mesh is a 1 x 2 x 2 low polygon cylinder. UinstancedStaticMeshComponents are used to optimize draw calls. (Annex 5)

In the cpp file, we'll setup the root component, create the default subobjects and attach them to the root as seen in Program Code 18.

Program Code 18. AStroke constructor.

```
AStroke::AStroke()
{
      PrimaryActorTick.bCanEverTick = false;
      bReplicates = true;

      Root = CreateDefaultSubobject<USceneComponent>(TEXT("Groot"));
      SetRootComponent(Root);
      StrokeMeshes =
CreateDefaultSubobject<UInstancedStaticMeshComponent>
(TEXT("StrokeMeshes"));
      StrokeMeshes->SetupAttachment(Root);

      JointMeshes =
CreateDefaultSubobject<UInstancedStaticMeshComponent>
(TEXT("JointMeshes"));
      JointMeshes->SetupAttachment(Root);

}
```

As shown in the header file (Annex 7), we need to create functions that calculate the position, rotation and scale of the stroke meshes, and the joint positions (Annex 8). We'll also need an Update function which is called from the Handcontroller class when the trigger is being pressed like seen in the following program code.

Program Code 19. UpdateStroke function.

```
void AStroke::UpdateStroke(FVector CursorLoc, bool bInvisible)
{
      ControlPoints.Add(CursorLoc);
      if(PreviousCursorLoc.IsNearlyZero())
      {
            PreviousCursorLoc = CursorLoc;
            return;
      }
      StrokeMeshes->AddInstance(GetNextSegmentLoc(CursorLoc,
bInvisible));
      JointMeshes->AddInstance(GetNextJointLoc(CursorLoc));
      PreviousCursorLoc = CursorLoc;
}
```

In the UpdateStroke function, we are adding the cursor locations, that are sent from the controller to a list, after which we add an instance of both the stroke and joint meshes. The invisible boolean is used for the first segment and segments that link other strokes together between trigger presses. The first invisible segment is created in the BeginPlay function as seen in Program Code 20.

Program Code 20. AStroke BeginPlay function.

```
void AStroke::BeginPlay()
{
	Super::BeginPlay();

	PreviousCursorLoc = FVector(1,1,1);
	StrokeMeshes->
AddInstance(GetNextSegmentLoc(FVector::ZeroVector, true)); //Create
the stroke instance and make the first segment invisible

}
```

In the Program Code 21, we'll spawn the stroke in the BeginPlay, call the UpdateStroke from the tick function whenever the trigger is being pressed and make sure that the first segment is invisible whenever we press the trigger. The ServerSpawnStroke_Implementation is discussed further in 5.4.2.

Program Code 21. AHandController function implementations.

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

	if(bIsClimbing)
	{
		FVector HandControllerDelta = GetActorLocation() - 
ClimbingStartLoc;
		GetAttachParentActor()->AddActorWorldOffset(-
HandControllerDelta);
	}
	if (!bTriggerReleased)
	{
		CurrentStroke->UpdateStroke(GetActorLocation(), false);
	}
}
void AHandController::ServerSpawnStroke_Implementation()
{
	CurrentStroke = GetWorld()->SpawnActor<AStroke>(StrokeClass);
	CurrentStroke->SetReplicates(true);
}
void AHandController::TriggerPressed()
{
	CurrentStroke->UpdateStroke(GetActorLocation(), true);
	bTriggerReleased = false;
}

void AHandController::TriggerReleased()
{
	bTriggerReleased = true;
}
```

5.4 UI Features

The UI is done with Blueprints instead of C++. The UI will be simple, as it will only have buttons to disconnect, play sound and go back to the game. It was supposed to have chat, but Unreal Engine does not have a built-in VR keyboard to handle user input, and the implementation of such a feature would take a significant portion of the project.

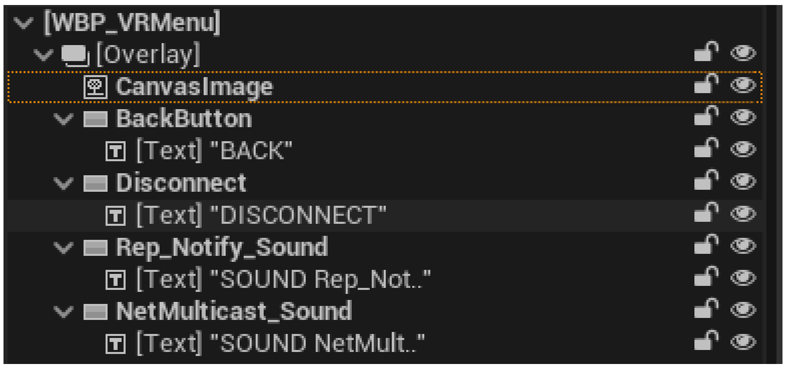
5.4.1 UI Handcontroller

First, we'll be creating a new C++ class that is derived from the HandControllerBase and also add UMG to the module list in the .build class. This class is used when interacting with the UI.

5.4.2 In-Game Menu

In order to create a working UI, we first need to create a Widget for it. Go to your content browser and search for **User Interface > Widget Blueprint**. According to the naming conventions, the prefix for widget Blueprints is WBP_ . When inside the Blueprint editor, delete the Canvas panel from the hierarchy and add an overlay. Under the overlay, we'll add an image and 4 buttons. Under the buttons add a text element as seen in Figure 8.

Figure 8. WBP_VRMenu hiararchy



```
[WBP_VRMenu]
  [Overlay]
    CanvasImage
    BackButton
      [Text] "BACK"
    Disconnect
      [Text] "DISCONNECT"
    Rep_Notify_Sound
      [Text] "SOUND Rep_Not.."
    NetMulticast_Sound
      [Text] "SOUND NetMult.."
```

First, we need to create the widget. This is done in the AVRCharacter::BeginPlay, by using the SpawnActor function as shown in Program Code 22.

Program Code 22. MenuActor initialization.

```
MenuActor = GetWorld()->SpawnActor<AActor>(MenuClass);
MenuActor->SetHidden(true);
```

We are spawning the menu based on where the VRCharacter location and rotation. The OpenMenu function is simply called whenever the A-button is pressed on the right-hand controller. This also activates the pointer functionality with the controller, which allows the user to point at elements within the menu widget as shown in Program Code 23.

Program Code 23. Open Menu function.

```
void AVRCharacter::OpenMenu()
{
	bMenuOpen = !bMenuOpen;
	if(bMenuOpen)
	{
		MenuActor->SetActorHiddenInGame(false);
		MenuActor->SetActorLocation(CameraActor->
GetActorLocation());
		MenuActor->SetActorRotation(FRotator(0,CameraActor->
GetActorRotation().Yaw, 0));
		RightController->WidgetPointer->Activate();
	}else
	{
		MenuActor->SetActorHiddenInGame(true);
		RightController->WidgetPointer->Deactivate();
	}
}
```

Binding actions to the buttons is done by creating a reference to the button in C++ and adding the meta = (BindWidget) property specifier as shown in Program Code 24. Note that the button name must be the same as it is in the Blueprint.

Program Code 24. NetMulticast_Sound declaration.

```
UPROPERTY(BlueprintReadOnly, VisibleAnywhere, meta = (BindWidget))
UButton* NetMulticast_Sound;
```

Next, we'll create a BlueprintCallable dynamic function when the button is clicked as shown in Program Code 25.

Program Code 25. ButtonOnClicked function.

```
void UUIUserWidget::ButtonOnClicked()
{
	NetMulticast_Sound->OnClicked.AddDynamic(this,
&UUIUserWidget::Multicast_PlaySound);
	Rep_Notify_Sound->OnClicked.AddDynamic(this,
&UUIUserWidget::SetPlaySoundBoolean);
	UE_LOG(LogTemp, Warning,
TEXT("UUIUserWidget::ButtonOnClicked"));
}
```

Finally, we need to allow the user to click the buttons with the pointer. This is done with the PressPointerKey and ReleasePointerKey functions by passing in the Ekeys::LeftMouseButton as can be seen in Program Code 26.

Program Code 26. ClickElement and ReleaseElement functions.

```
void AVRCharacter::ClickElement()
{
	if(bMenuOpen)
	{
		RightController->WidgetPointer->
PressPointerKey(EKeys::LeftMouseButton);
	}
}

void AVRCharacter::ReleaseElement()
{
	if(bMenuOpen)
	{
		RightController->WidgetPointer->
ReleasePointerKey(EKeys::LeftMouseButton);
	}
}
```

5.5 Multiplayer Features

As for the multiplayer features, a custom game mode and game states are needed. We'll also need to replicate the character movement and the drawing for it to be considered multiplayer. In a VR multiplayer, it is important to replicate the movement of the controllers as well as the headset.

5.5.1 GameMode and GameState

For the GameMode, we need the two-player classes that we use for spawning, as well as a function we use to spawn them.

Program Code 27. Player classes and SpawnPlayer function declaration.

```
TSubclassOf<APawn> FirstPlayerClass;
TSubclassOf<APawn> SecondPlayerClass;

void SpawnPlayer(APlayerController* PC);
```

In the GameMode constructor, as seen in Program Code 28, we'll get the blueprint version of the player classes.

Program Code 28. AVRGameMode constructor.

```
AVRGameMode::AVRGameMode()
	: Super()
{
	static ConstructorHelpers::FClassFinder<APawn>
PlayerPawnClassFinder(TEXT("/Game/Blueprints/BP_VRCharacter"));
	checkf(PlayerPawnClassFinder.Class, TEXT("BP_VRCharacter not
found!"))
	FirstPlayerClass = PlayerPawnClassFinder.Class;

	static ConstructorHelpers::FClassFinder<APawn>
PlayerPawnClassFinder2(TEXT("/Game/Blueprints/BP_FPSCharacter"));
	checkf(PlayerPawnClassFinder2.Class, TEXT("BP_FPSCharacter not
found!"))
	SecondPlayerClass = PlayerPawnClassFinder2.Class;

}
```

In the SpawnPlayer function, we'll first create the FactorSpawnParameters property and add a SpawnCollisionHandlingOverride that tries to adjust spawn location if the character is colliding with something. Next, the client character is spawned, and if the spawning is successful the client PlayerController will possess the spawned character as seen in Program Code 29.

Program Code 29. SpawnPlayer function.

```
void AVRGameMode::SpawnPlayer(APlayerController* PC)
{
	FActorSpawnParameters ActorSpawnParams;
	ActorSpawnParams.SpawnCollisionHandlingOverride =
ESpawnActorCollisionHandlingMethod::AdjustIfPossibleButDontSpawnIfCol
liding;
	PlayerLoc = FVector(FMath::RandRange(0, 2000),
FMath::RandRange(0, 2000), FMath::RandRange(50, 2000));

	AFPSCharacter* ClientPlayer = Cast<AFPSCharacter>(GetWorld()->
SpawnActor(SecondPlayerClass, &PlayerLoc,nullptr, ActorSpawnParams));
	ensureMsgf(ClientPlayer, TEXT("ClientPlayer cast failed!"));
	if(ClientPlayer)
	{
		PC->Possess(ClientPlayer);
	}
}
```

The host character will be spawned inside the BeginPlay function like seen in Program Code 30. As previously the spawn parameters and a random spawn location is created. Using the GetPlayerController function and giving it the PlayerIndex of 0, we can get the host player controller. If successful, we'll then spawn and possess the character using the VRCharacter class.

As the GameMode only exists in the server, we'll call the GameMode from the GameState with a server function.

Program Code 30. AVRGameMode BeginPlay function.

```
void AVRGameMode::BeginPlay()
{
	Super::BeginPlay();

	FActorSpawnParameters ActorSpawnParams;
	ActorSpawnParams.SpawnCollisionHandlingOverride =
ESpawnActorCollisionHandlingMethod::AdjustIfPossibleButDontSpawnIfCol
liding;
	PlayerLoc = FVector(FMath::RandRange(0, 2000),
FMath::RandRange(0, 2000), FMath::RandRange(50, 2000));

	APlayerController* PC =
UGameplayStatics::GetPlayerController(GetWorld(), 0); //Host player
controller
	ensureMsgf(PC, TEXT("PC1 is nullptr!"));
	if(PC)
	{
		AVRCharacter* HostPlayer = Cast<AVRCharacter>
(GetWorld()->SpawnActor(FirstPlayerClass, &PlayerLoc,nullptr,
ActorSpawnParams));
		ensureMsgf(HostPlayer, TEXT("HostPlayer cast failed!"));
		if(HostPlayer)
		{
			PC->Possess(HostPlayer);
		}
	}
}
```

As for the GameState, we'll create a server function so that the client can send its PlayerController pointer to the GameMode. It will also use the Reliable and WithValidation function specifiers like can be seen in Program Code 31.

Program Code 31. SpawnMe function declaration.

```
UFUNCTION(Server, Reliable, WithValidation)
void SpawnMe(APlayerController* PC);
```

In BeginPlay there is an if statement that checks if the player is the client. If successful it will call the SpawnMe function with the local PlayerController.

Program Code 32. Authority check for SpawnMe.

```
if(!HasAuthority())
{
        SpawnMe(UGameplayStatics::GetPlayerController(this, 0));
}
```

In the SpawnMe_Implementation we first check if the PlayerController is valid. We then get the current GameMode and call the previously created SpawnPlayer and pass in the PlayerController.

5.5.2 Movement Replication

Character movement is replicated automatically by the engine, but other aspects such as controllers need to be manually replicated. Controllers need to be set as actors, which then are attached to the VRCharacter, in order to replicate their movement. Subobject movement can't be normally replicated.

To replicate the handcontrollers we first set bReplicates to true in the HandController constructor or it can also be done in the HandControllerBase constructor. We also need to create a new function that keeps track of the replicated properties. This is done by adding the GetLifetimeReplicatedProps function. In this function, you first call the super function and after that write out the macro DOREPLIFETIME() and add the class name and the property you want to replicate inside the parenthesis as shown in Program Code 33.

Program Code 33. GetLifetimeReplicatedProps function for the AVRCharacter class.

```
void
AVRCharacter::GetLifetimeReplicatedProps(TArray< FLifetimeProperty >
& OutLifetimeProps) const
{
	Super::GetLifetimeReplicatedProps(OutLifetimeProps);

	DOREPLIFETIME(AVRCharacter, RightController);
	DOREPLIFETIME(AVRCharacter, LeftController);
}
```

5.5.3 Drawing Replication

The drawing replication is handled in the Stroke class. Since the spawning is done in the BeginPlay function of the HandController class, both the players will execute it. For the actual spline to be replicated, we need to mark the StrokeMeshes and JointMeshes to be replicated in the Stroke.h. As for all properties marked as replicated, they also need the specified GetLifetimeReplicaatedProps function to be implemented. We'll add the DOREPLIFETIME macro for both of the properties as can be seen in Program Code 34.

Program Code 34. GetLifetimeReplicatedProps function for the AStroke class.

```
void AStroke::GetLifetimeReplicatedProps(TArray< FLifetimeProperty >
& OutLifetimeProps) const
{
	Super::GetLifetimeReplicatedProps(OutLifetimeProps);

	DOREPLIFETIME(AStroke, JointMeshes);
	DOREPLIFETIME(AStroke, StrokeMeshes);
}
```

5.5.4 Sound replication

NetMulticast should be used for sound replication because only those players in the vicinity of the sound should receive the information. The VR Character can play a sound by clicking a button in the menu. First, we'll need to create the necessary functions and properties to play sounds and for their replication, as seen in Annex 9. The multicast version will be BlueprintCallable function as the Widget will be calling it whenever the button gets clicked.

In the UIUserWidget class we simply need to implement the necessary functions. The multicast version needs the _Implementation suffix, but the ReplicatedUsing version does not. We will play a sound simply by calling the PlaySound function and passing in the UsoundBase* object. We need to assign sounds in the widget blueprint. For the ReplicatedUsing function, we also need a function that changes the boolean, as the change in the value will trigger the function to be called both on the server and the clients. For listen servers, the ReplicatedUsing function will not be called on the client that is acting as a server. To get around this, simply execute the functionality on the server normally without ReplicatedUsing.

5.6 Finishing touches

This project has had many problems, most of which have been solved. Some of the fixes are workarounds rather than fixes. Some of the problems are not addressed in this document but might have been solved later. In this chapter, we'll discuss optimization, bug fixes, and issues that have yet to be solved.

5.6.1 Optimization

The major optimization that has already been done is the use of instanced static meshes. Without the use of them, each time a new segment to the painting mesh was added, it would increase the number of draw calls.

The second way of optimizing was to use an invisible segment so that every time the trigger is pressed, the game would have no need to spawn another instance of the stroke mesh.

5.6.2 Bug Fixes

The first issue that we'll fix is syncing the player position and VR camera position so that the player can't walk through geometry. This issue is due to the camera location and the player capsule location, being out of sync. The needed offset is calculated in the AVRCharacter Tick function as seen in Program Code 35.

Program Code 35. Camera offset calculations.

```
FVector CameraOffset = Camera->GetComponentLocation() -
GetActorLocation();
CameraOffset.Z = 0;
AddActorWorldOffset(CameraOffset);
VRRoot->AddWorldOffset(-CameraOffset);
```

The host's movement can extremely laggy and the client may see the host rubberbanding. This might be caused by the HandController meshes CollisionPreset being set to Block All Dynamic. Changing this to Overlap All should solve this issue.

When using postprocessing, such as fading or blinkers, there would be visual artifacts present, such as warping. This problem was fixed after updating graphics drivers.

When starting the project as a Blueprint project, you might not have the option to create a c++ class. This is fixed by adding a c++ component to any of the objects in the level. This generates necessary files for you, and after that, you can create a c++ class.

Launching the game in VR Preview will crash the editor Giving the following error: *"Assertion failed: ClearValue.ColorBinding == EClearBinding::EColorBound [File:D:/build/++UE5/Sync/Engine/Source/Runtime/Windows/D3D11RHI/Private/D3D11Commands.cpp] [Line: 1129] Texture: MotionBlur.SceneColor does not have a color bound for fast clears* ". The workaround solution for this is to disable Motion Blur from **Edit > Project Settings > Rendering > Default Settings > Motion Blur**.

5.6.3 Known Issues

You might experience a bug with shadows in VR, where a large shadow will appear when an object is being rendered with the right eye. This shadow does not appear with the left eye. It was tested that this problem does not occur with the latest Unreal Engine 4 version and might be fixed in a later version of Unreal Engine 5. This was tested on the early access version, and not the full release or the preview version.

6 Summary

The goal of the thesis was to provide an example of how to create a virtual reality environment with multiplayer features in C++ with unreal engine 5. The features that were essential from a virtual reality perspective were movement, blinkers, and the user interface. These features needed to be tailored specifically to the virtual reality character, mostly to avoid motion sickness. Additional features of virtual reality were climbing, and writing. From a multiplayer perspective, the goal was to implement a simple way of interacting with another player with C++, which was successful.

As for the future, this thesis provided an excellent way of introducing Unreal Engine as a basis for a virtual reality project. I learned how to set up a simple multiplayer in C++ in a virtual reality project. I learned how to create different types of movement for a virtual reality character, creating splines dynamically and efficiently, and I learned what makes a good user interface in virtual reality.

The demo video can be seen on YouTube (Pennanen R.). The demo video shows all the features created in this thesis.

References

Chang, E., Kim, H. T., & Yoo, B. (2020). Virtual reality sickness: A review of causes and measurements. International Journal of Human–Computer Interaction, 36(17), 1658-1682. https://doi.org/10.1080/10447318.2020.1778351

Epic Games. (n.d.). VR platforms. https://docs.unrealengine.com/4.27/en-US/SharingAndReleasing/XRDevelopment/VR/VRPlatforms/

Epic Games. (n.d.). Virtual reality best practices. https://docs.unrealengine.com/4.27/en-US/SharingAndReleasing/XRDevelopment/VR/VRBestPractices/#light

Epic Games. (n.d.). Function specifiers. https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/GameplayArchitecture/Functions/Specifiers/

Epic Games. (n.d.). Hardware and software specifications. https://docs.unrealengine.com/4.27/en-US/Basics/InstallingUnrealEngine/RecommendedSpecifications/

Epic Games. (n.d.). Property specifiers. https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/GameplayArchitecture/Properties/Specifiers/

Epic Games. (n.d.). APawn::AddMovementInput. https://docs.unrealengine.com/4.26/en-US/API/Runtime/Engine/GameFramework/APawn/AddMovementInput/

GameDev.tv team. (n.d.). Unreal VR Dev. Udemy. https://www.udemy.com/course/unrealvr/

Howard, M. C., & Van Zandt, E. C. (2021). A meta-analysis of the virtual reality problem: Unequal effects of virtual reality sickness across individual differences. Virtual Reality, 25(4), 1221-1246. https://doi.org/10.1007/s10055-021-00524-3

Reason, J. T., & Brand, J. J. (1975). Motion sickness. Academic Press.

Pennanen R. (2022, February 5). VR project preview NetMode server unreal engine 5 SteamVR. YouTube. https://youtu.be/pF5vMNFGddk

Unreal engine: Naming convention guide. (2022, June 6). Tom Looman. https://www.tomlooman.com/unreal-engine-naming-convention-guide/

Material management plan

All gathered material will be stored for a minimum of one year on the student's personal computer. No sensitive or personal material was gathered for this thesis, so there is no need to manage such information.

Annex 1: AVRCharacter::BeginPlay()

```
LeftController = GetWorld()->SpawnActor<AHandController>
(HandControllerClass);
if (LeftController != nullptr)
{
	LeftController->AttachToComponent(VRRoot,
FAttachmentTransformRules::KeepRelativeTransform);
	LeftController->SetHand("Left");
	LeftController->SetOwner(this); // FIX FOR 4.22
}

RightController = GetWorld()->SpawnActor<AHandController>
(HandControllerClass);
if (RightController != nullptr)
{
	RightController->AttachToComponent(VRRoot,
FAttachmentTransformRules::KeepRelativeTransform);
	RightController->SetHand("Right");
	RightController->SetOwner(this); // FIX FOR 4.22
}

LeftController->PairController(RightController);
RightController->PairController(LeftController);
```

Annex 2: HandController.h for climbing

```
void Grip();
void Release();

void PairController(AHandController* OtherController);

//COMPONENTS
UPROPERTY(VisibleAnywhere)
UMotionControllerComponent* MotionController;
UPROPERTY(EditDefaultsOnly)
UHapticFeedbackEffect_Base* HapticEffect;

// CALLBACKS
UFUNCTION()
void ActorBeginOverlap(AActor* OverlappedActor, AActor* OtherActor);
UFUNCTION()
void ActorEndOverlap(AActor* OverlappedActor, AActor* OtherActor);

// HELPERS
bool CanClimb() const;

// STATE
bool bCanClimb = false;
bool bIsClimbing = false;

//VECTORS
FVector ClimbingStartLoc;

//ACTORS
UPROPERTY()
AHandController* SecondController;
```

Annex 3: VRCharacter.h for teleportation

```
//COMPONENTS

UPROPERTY(EditDefaultsOnly)
class UStaticMeshComponent* TeleporterCylinder;
UPROPERTY(VisibleAnywhere)
class USplineComponent* TeleporterPath;
UPROPERTY(EditDefaultsOnly)
class UStaticMesh* TeleportArchMesh;
UPROPERTY(EditDefaultsOnly)
class UMaterialInterface* TeleportArchMaterial;
UPROPERTY(EditAnywhere)
TArray<class USplineMeshComponent*> SplineMeshPool;

//BP EDITABLE VARIABLES
UPROPERTY(EditAnywhere)
float MaxTeleportDistance = 100;
UPROPERTY(EditAnywhere)
float TeleportProjectileSpeed = 500;
UPROPERTY(EditAnywhere)
float TeleportSimTime = 3;
UPROPERTY(EditAnywhere)
float TeleportFadeTime = 1;
UPROPERTY(EditAnywhere)
FVector TeleportProjectionExtent = FVector(100, 100, 100);
UPROPERTY(EditAnywhere)
float TeleportProjectileRadius = 10;

bool bTeleporting = false;

bool bCanTeleport = true; //Public

FVector StartPos, EndPos, StartTan, EndTan;

//TELEPORTATION FUNCTIONS
void DrawTeleportTrace(FVector Start, FVector End);
void DrawTeleportPath(TArray<FVector>& OutPath);
bool FindTeleportDestination(TArray<FVector>& OutPath, FVector&
Location);
void FinishTeleport();
void TeleportPlayer();
void UpdateTeleportCylinder();
void UpdateTeleportSpline(const TArray<FVector>& OutPath);
```

Annex 4: FindTeleportDestination function

```
bool AVRCharacter::FindTeleportDestination(TArray<FVector>& OutPath,
FVector& Location)
{
	FVector Start = LeftController->GetActorLocation();
	FVector Look = LeftController->GetActorForwardVector();
	Look = Look.RotateAngleAxis(45, LeftController->
GetActorRightVector());
	FVector End = Start + Look * MaxTeleportDistance;

	FPredictProjectilePathParams
PathParams(TeleportProjectileRadius, Start, TeleportProjectileSpeed *
Look,
					TeleportSimTime,
ECC_Visibility, this);
	FPredictProjectilePathResult PathResult;
	bool bHit = UGameplayStatics::PredictProjectilePath(this,
PathParams, PathResult);

	if (!bHit)
	{
		return false;
	}

	FNavLocation NavLoc;
	bool bOnNavMesh =
UNavigationSystemV1::GetNavigationSystem(GetWorld())->
ProjectPointToNavigation(
		PathResult.HitResult.Location, NavLoc,
TeleportProjectionExtent);

	if (!bOnNavMesh)
	{
		return false;
	}

	for (FPredictProjectilePathPointData PointData :
PathResult.PathData)
	{
		OutPath.Add(PointData.Location);
	}

	Location = NavLoc;

	return bHit && bOnNavMesh;
}
```

Annex 5: DrawTeleportPath function

```
void AVRCharacter::DrawTeleportPath(TArray<FVector>& OutPath)
{
	UpdateTeleportSpline(OutPath);

	for(USplineMeshComponent* SplineMeshComponent : SplineMeshPool)
	{
		SplineMeshComponent->SetVisibility(false);
	}

	int32 SegmentNum = OutPath.Num() - 1;
	for (int i = 0; i < SegmentNum; ++i)
	{
		if(SplineMeshPool.Num() <= i)
		{
			USplineMeshComponent* SplineMesh = 
NewObject<USplineMeshComponent>(this, 
"SplineMesh"+SplineMeshPool.Num()+1);
			SplineMesh->
SetMobility(EComponentMobility::Movable);
			SplineMesh->
AttachToComponent(TeleporterPath,FAttachmentTransformRules::KeepRelat
iveTransform);
			SplineMesh->SetStaticMesh(TeleportArchMesh);
			SplineMesh->SetMaterial(0,TeleportArchMaterial);
			SplineMesh->RegisterComponent();
			SplineMeshPool.Add(SplineMesh);
		}

		TeleporterPath->
GetLocalLocationAndTangentAtSplinePoint(i, StartPos, StartTan);
		TeleporterPath->
GetLocalLocationAndTangentAtSplinePoint(i+1, EndPos, EndTan);


		USplineMeshComponent* SplineMesh = SplineMeshPool[i];
		SplineMesh->SetVisibility(true);
		SplineMesh->SetStartAndEnd(StartPos, StartTan, EndPos, 
EndTan);
	}
}
```

Annex 6: GetBlinkerCenter function

```
FVector2D AVRCharacter::GetBlinkerCenter()
{
	FVector MovementDir = GetVelocity().GetSafeNormal();
	if (MovementDir.IsNearlyZero())
	{
		return FVector2D(.5, .5);
	}
	FVector WorldStationaryLoc;
	if (FVector::DotProduct(Camera->GetForwardVector(), 
MovementDir) > 0)
	{
		WorldStationaryLoc = Camera->GetComponentLocation() + 
MovementDir * 1000;
	}
	else
	{
		WorldStationaryLoc = Camera->GetComponentLocation() - 
MovementDir * 1000;
	}
	FVector2D ScreenLoc;

	if (PC == nullptr)
	{
		return FVector2D(.5, .5);
	}

	PC->ProjectWorldLocationToScreen(WorldStationaryLoc, 
ScreenLoc);

	XVal = static_cast<int32>(ScreenLoc.X);
	YVal = static_cast<int32>(ScreenLoc.Y);

	int32 Size_X, Size_Y;
	PC->GetViewportSize(Size_X, Size_Y);

	Range = FVector2D(0, 1);
	VPSizeRangeX = FVector2D(0, Size_X);
	VPSizeRangeY = FVector2D(0, Size_Y);
	XMapped = FMath::GetMappedRangeValueClamped(VPSizeRangeX, 
Range, XVal);
	YMapped = FMath::GetMappedRangeValueClamped(VPSizeRangeY, 
Range, YVal);

	return FVector2D(XMapped, YMapped);
}
```

Annex 7: Stroke.h

```
UFUNCTION()
FTransform GetNextSegmentLoc(FVector CurrentLoc, bool bInvicible)
const;
UFUNCTION()
FTransform GetNextJointLoc(FVector CurrentLoc) const;
UFUNCTION()
FVector GetNextSegmentScale(FVector CurrentScale) const;
UFUNCTION()
FQuat GetNextSegmentRotation(FVector CurrentRot) const;
UFUNCTION()
FVector GetNextSegmentPosition(FVector CurrentPos) const;

UPROPERTY(VisibleAnywhere)
USceneComponent* Root;
UPROPERTY(VisibleAnywhere)
UInstancedStaticMeshComponent* StrokeMeshes;
UPROPERTY(VisibleAnywhere)
UInstancedStaticMeshComponent* JointMeshes;

FVector PreviousCursorLoc;
TArray<FVector> ControlPoints;

UFUNCTION()
void UpdateStroke(FVector CursorLoc, bool bTriggerPressed); //Public
```

Annex 8: Stroke functions

```
FTransform AStroke::GetNextSegmentLoc(FVector CurrentLoc, bool
bInvisible) const
{
	FTransform SegmentTransform;
	SegmentTransform.SetLocation(GetNextSegmentPosition(CurrentLoc)
);
	SegmentTransform.SetRotation(GetNextSegmentRotation(CurrentLoc)
);
	if(!bInvisible)
	{

	SegmentTransform.SetScale3D(GetNextSegmentScale(CurrentLoc));
	}else
	{
		SegmentTransform.SetScale3D(FVector::ZeroVector); //Make
the Segment that connects the new stroke invisible
	}
	return SegmentTransform;
}

FTransform AStroke::GetNextJointLoc(FVector CurrentLoc) const
{
	FTransform JointTransform;
	JointTransform.SetLocation(GetTransform().InverseTransformPosit
ion(CurrentLoc));
	return JointTransform;
}

FVector AStroke::GetNextSegmentScale(FVector CurrentScale) const
{
	return FVector((CurrentScale-PreviousCursorLoc).Size(), 1,1);
}

FQuat AStroke::GetNextSegmentRotation(FVector CurrentRot) const
{
	FVector Segment = CurrentRot - PreviousCursorLoc;
	FVector SegmentNormal = Segment.GetSafeNormal();
	return FQuat::FindBetweenNormals(FVector::ForwardVector,
SegmentNormal);
}

FVector AStroke::GetNextSegmentPosition(FVector CurrentPos) const
{
	return
GetTransform().InverseTransformPosition(PreviousCursorLoc);
}
```

Annex 9: UIUserWidget.h

```
public:

	UFUNCTION(NetMulticast, Unreliable, BlueprintCallable)
	void Multicast_PlaySound();

	void SpawnMenu();

protected:

	UPROPERTY(BlueprintReadOnly, VisibleAnywhere, meta =
(BindWidget))
	UOverlay* MenuOverlay;

	UPROPERTY(BlueprintReadOnly, VisibleAnywhere, meta =
(BindWidget))
	UButton* NetMulticast_Sound;

	UPROPERTY(BlueprintReadOnly, VisibleAnywhere, meta =
(BindWidget))
	UButton* BackButton;

	UPROPERTY(BlueprintReadOnly, VisibleAnywhere, meta =
(BindWidget))
	UButton* Disconnect;

	UFUNCTION(BlueprintCallable)
	void ButtonOnClicked();

	UPROPERTY(EditDefaultsOnly)
	USoundBase* SoundMulticast;

private:

	UPROPERTY(EditDefaultsOnly)
	TSubclassOf<UUIUserWidget> MenuOverlayClass;
```