Seamless Virtual Reality Locomotion

Masaya Todoroki

A thesis submitted to

Auckland University of Technology

In fulfilment of the requirements for the degree of

Master of Design

2018

Digital Design

Auckland University of Technology

Research Question:

"This research project explores creating an immersive locomotion system that combines multiple forms of locomotion, allowing the user to seamless move through a variety of terrains in a virtual environment, being created with the Unity 3d game engine and coding with C#".

Attestation of Authorship

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person (except where explicitly defined in the acknowledgements), nor material which to a substantial extent has been submitted for the award of any other degree or diploma of a university or other institution of higher learning.



Signed _____ Date 17/05/2018

Acknowledgments

I would like to thank my supervisor Gregory Bennett for all his help and guidance with this project.

I would also like to thank my boss Aliesha Staples at Staples VR for her support during my studies.

Finally, I'd like to thank So Young Kim and Gary Ng who have helped me gather data through playtesting this project.

Abstract

This project explores creating alternative ways of locomotion and combining these locomotion systems together for virtual reality (VR) gaming. This allows users to seamlessly switch between multiple locomotion modes as they move through the virtual environment depending on terrain and location. This project will be created using the Unity3d game engine and will be coded using C#. The project also needs to take into consideration how it's locomotion system would affect gameplay if it was to be adapted into a game. The primary consideration is making sure that input from controllers needs to be shared between both the locomotion system and the interaction with game objects. With VR content slowly increasing in quantity, many developers are creating new and unique ways of gameplay. However, locomotion is a key issue that consistently arises from many VR game experiences. This project will take a practice-based and practice-led research approach to discovering new possibilities for movement with VR. This exegesis will also provide code from each locomotion system created. These codes can be used as a guideline for creating certain locomotion's within a VR game.

Table of contents

1 – Introduction……9

2 – Contextual……11

2.1 - Locomotion System's in current VR Games……11

2.2 - Interaction……17

2.3 Motion Sickness……18

3 - Methodology……20

3.1 - Initial Plan……21

3.2 - Act……22

3.3 - Monitor……23

3.4 - Evaluate……23

3.5 - Revised plan……24

3.6 - Iterative Design……24

4 - Project Development/Documentation……26

4.1 - Walking/Running……26

4.2 - Swimming……36

4.3 - Jumping……45

4.4 - Climbing……47

4.5 - Grappling Hook Locomotion……54

4.7 - Combining locomotions together……58

5 - Discussion……60

6 - Conclusions/Implications……62

7 - Bibliography……63

Technical Terms

GameObject – Base class for all entities in Unity scenes. Every object in a scene is a GameObject.

Float – “floating point”, a numeric value which contains floating decimal points. E.g. 1.45

Int – “integer”, a numeric value which holds whole numbers E.g. 1

Raycast – a ray that gets sent from a position in a specific direction and can return information about what GameObject the ray has hit.

Vector2 – Contains two float values, *x* and *y*, mainly used to represent a 2d position.

Bool – “Boolean”, a binary variable that can only have one of two possibilities, 0 (false) or 1 (true)

Mechanic - Rules/Features of the game. E.g. locomotion is a type of game mechanic.

List of Figures

Figure 01. Image of typical HTC Vive room-scale setup……9

Figure 02. Action Research Diagram……20

Figure 03. Image of controller layout for Black 9. ……22

Figure 04. Vive Controller Diagram……26

Figure 05. Edited Controller Diagram for this project……27

Figure 06. Image of tracker attached to the hip……33

List of Videos

Video 01: Gameplay footage of Raw Data Teleporting……12

Video 02: Gameplay footage of Skyrim Vignette……13

Video 03: Gameplay footage of Hover Junkers Locomotion……14

Video 04: Gameplay footage of Skyrim Swimming……15

Video 05: Gameplay footage of Climbey Locomotion……16

Video 06: Moving Up and Down Locomotion……31

Video 07: Leaning Values……34

Video 08: Moving Up and Down Leaning Locomotion……34

Video 09: Crouching Example……35

Video 10: Swimming Locomotion……39

Video 11: Double Body Collider……41

Video 12: Final Swimming Locomotion……44

Video 13: Jumping Locomotion……46

Video 14: Ladder Fall……52

Video 15: Holding onto Ladder fall……53

Video 16: Grappling Hook Locomotion……55

Video 17: Pulling Grappling Hook Locomotion……57

Video 18: Combined Locomotion……59

Video 19: Gameplay Bug……60

1 - Introduction

Current Locomotion Issues

"VR locomotion, an important interaction component that enables navigation in VR environments"[1].

Room-scale VR headsets such as the HTC Vive, already have a form of locomotion built into it. With the ability to freely walk within a certain space, users can physically move their own body as they would in the real world to move to a certain place within the virtual world. However, the main problem that arises from this is that there is a limit to how far a person can move. These limitations are caused by the size of the room that user is using the headset in, or the limitation of distance that the Vive Lighthouses can transmit. Once the user moves beyond those limits, they will either hit into objects such as walls in the real world, or the tracking of the headset will be lost due to the headset not being able to see the Lighthouses anymore.

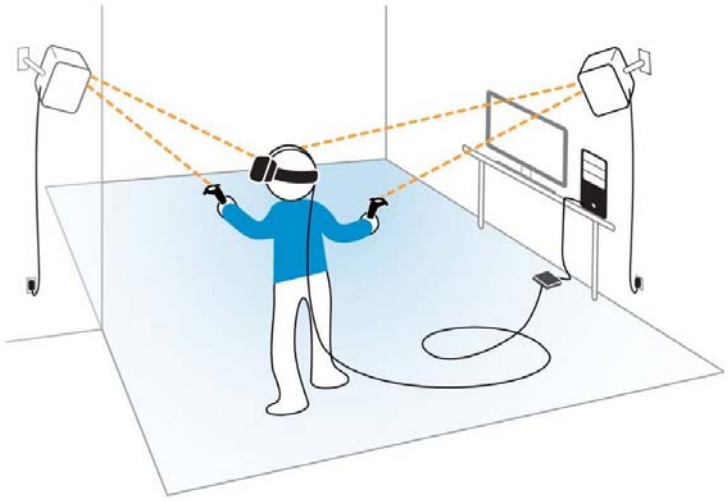


***Figure 01**. Image of typical HTC Vive room-scale setup.*

***Source:** HTC-Vive-manual-main.jpg, digital image. accessed May 14, 2018*
https://www.androidheadlines.com/2016/02/htc-vive-user-guide-shows-setup-play-area.html

[1] Costas Boletsis, "The New Era of Virtual Reality Locomotion: A Systematic Literature Review of Techniques and a Proposed Typology," September 28, 2017.

While having the room-scale locomotion ability does immerse the user as their movements realistically translate from the real world, this limitation of space is a huge disadvantage for creating video games. To counter this, developers have been using extra locomotion systems alongside the room-scale movement or have only allowed a very small play space available to the user. Currently limiting the play space and not having any extra locomotion is the most common technique used within VR gaming, this is shown through the countless wave base shooters, which require the user to stand in one spot for the entire game while shooting at targets or enemies. However, to allow the user to move even further within the Virtual World, most developers use a teleporting mechanism allowing users to quickly move around. Teleporting in VR works by holding down a button on the controller and pointing either at the ground or at a teleport point, after releasing the button, the user will then be transported to that location. This way of movement within a game environment can cause many problems though. Having the user being able to freely teleport gives the user more freedom of exploiting certain scenarios within a game, such as easily running away from enemies or sticking their controller through a wall and teleporting to places they weren't meant to be able to reach yet. Some developers have put limits or the teleporting aspect of their game, such as a timer only allowing players to teleport a certain amount of times. While this can be quite effective in some games, it reduces the amount of freedom that a user has and restricts their movement, giving a very unnatural feel to it. There have been more and more locomotion games that have been appearing lately, however, the main focus of these games is the locomotion itself, rather than having other features such as fighting enemies.

2 - Contextual

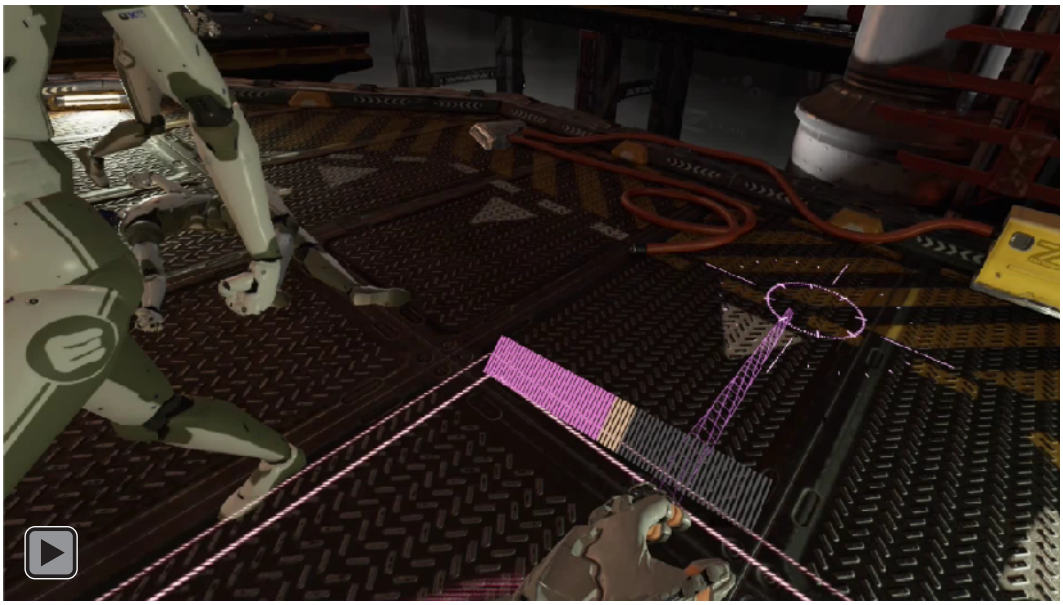
This contextual review takes the form of a survey of current VR games with the focus of exploring different forms of locomotions within them. Due to the current lack of published material available to this project specific issue, exploring the current state of locomotion in the currently available VR games will form the basis of practice for this project. This contextual review will also analyse how to further immerse the user through their interactions in the virtual world. It will also research into the current motion sickness issue that VR poses, and what techniques can be used to help reduce this issue.

2.1 - Locomotion Systems in current VR Games

This project's inspiration came from many different VR games that I have played using the HTC Vive. The very first game I played, "The Lab"[2], created by Valve. While immersive, it showed me the variable limitation of room scale VR. The amount that you can physically move around in was limited to the amount of real physical space you had. To solve this issue, The Lab using a teleport mechanic to allow players to move beyond their current location. This teleport function is a mechanic that is now one of the most widely used locomotion system. While it does allow the player to go beyond where they are able to physically move to, there are many issues with this system. "Budget Cuts"[3] created by Neat Corporation, is a small demo of a game which uses teleportation as its main form of locomotion. When moving around when there is no action happening, this form of locomotion is completely fine. However, when faced with an enemy attacking you, I have found this type of locomotion to be awkward

[2] *The Lab*, Microsoft Windows (Valve, 2016).
[3] *Budget Cuts*, Microsoft Windows (Neat Corporation, 2016).

and a bit chaotic. This is due to the instant positional change from teleporting, which means you need to reanalyse each time you are in a new position of where everything currently is in the game, which can be quite disorienting. I also found this issue in the game “Raw Data”[4] which has a lot more intense action than “Budget Cuts”[5]. The reason behind this issue is since teleporting involves physically pointing towards the certain location that you wish to teleport to. Comparing this towards a traditional 2d video game where the player only needs to move their thumb on the analog stick on a controller, or use a finger to press a key on a keyboard, teleporting in VR requires a lot of coordination and effort to get to a certain location when under pressure. This involves the VR user to physically move their whole arm to point at the exact position they wish to teleport to.



Video 01: *Gameplay footage of Raw Data Teleporting*

Not every VR game uses teleportation. Some games use a more traditional movement option, which is called ‘smooth movement’. On the HTC Vive Controller, the touchpad can be used as input for X and Y axis values, which is the same as a gamepad controller analog

[4] *Raw Data*, Microsoft Windows (Survios, 2017).
[5] *Budget Cuts*.

stick. This smooth movement allows developers to transfer over the movement that is wildly already known from non VR games into VR games. This type of locomotion is becoming one of the most popular forms of locomotion from gamers. However, developers have previously shied away from this due to motion sickness that could occur. This is due to the disconnect between the movement happening that the user can see and the physical movement that the user is currently making.

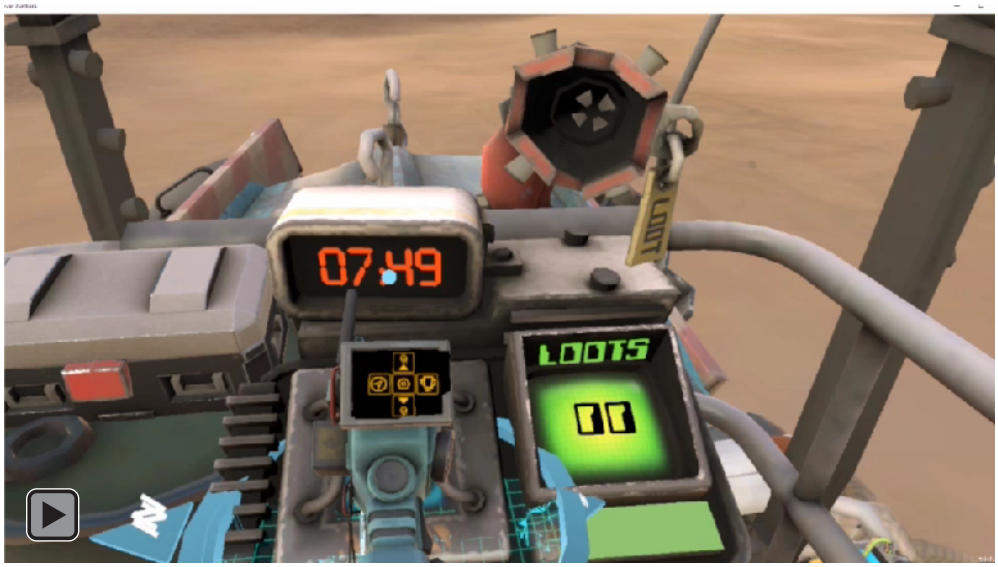
As VR has aged, more people have become more adept at this form of locomotion. For newcomers, developers have added options to help with motion sickness. The most common option is to add a black vignette to the camera. Researchers at Columbia University[6] found out that reducing the field of view for the user would result in motion sickness being drastically reduced. This is due to the reduced visual cues for the brain to register. By adding this vignette to the camera while a player is moving, reduces the amount of image being shown to the user. This contextual review explores motion sickness further in the 2.3 Motion Sickness section.



Video 02: *Gameplay footage of Skyrim Vignette*

[6] Ajoy Fernandes and Steven Feiner, "Combating VR Sickness through Subtle Dynamic Field-Of-View Modification," 2016.

Another creative way of locomotion used is to allow players to move the space through interactions with the environment they are in. An example of this is through the game “Hover Junkers”[7] created by Stress Level Zero. In Hover Junkers, you pilot a hover ship and defend and attack opposing players. Players are able to move through the environment by steering the ship. Players choose the size of their ship which is closest to the size of their play area.

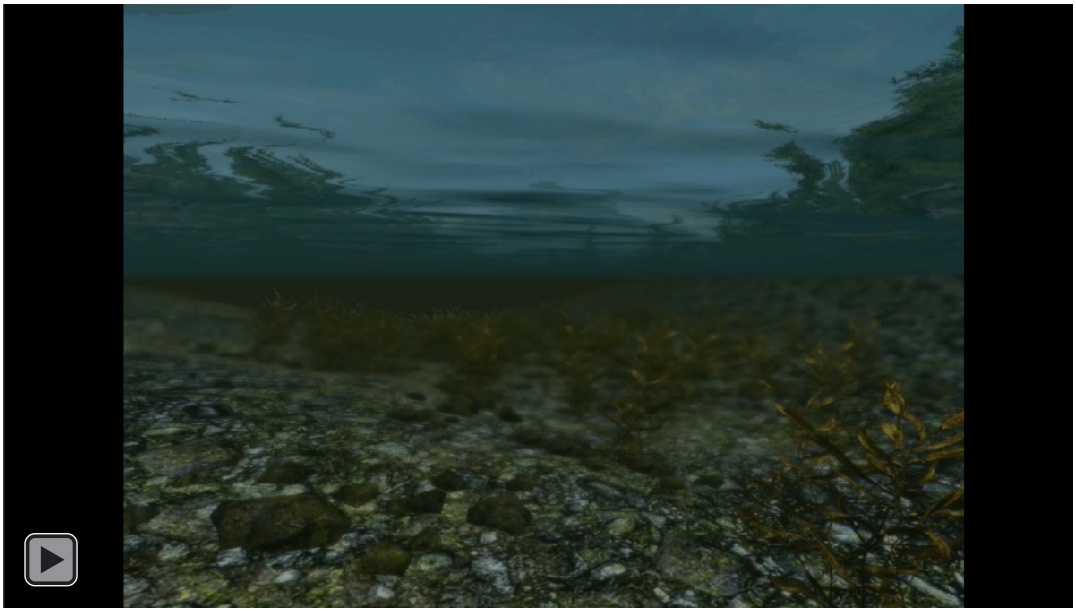


Video 03: *Gameplay footage of Hover Junkers Locomotion*

While there are quite a few underwater VR games, I’ve yet to find many that use actual swimming motion as the source of movement. One project that I came across was a project called “Atlantis”[8] created by Bryan Anenberg, Harrison Ho, and Karen Yang, which uses very similar swimming motion to the motion that I am using in this project. Upon watching a video of their project, it showed one of the issues that my project is attempting to solve. While outside of the water, the motions used for movement is the exact same as when underwater, allowing the user to swim through the air. “The Elder

[7] *Hover Junkers*, Microsoft Windows (Stress Level Zero, 2016).
[8] Bryan Anenberg, Harrison Ho, and Karen Yang, *Atlantis - VR Swimming Simulation & Puzzle Game*, Microsoft Windows, 2017.

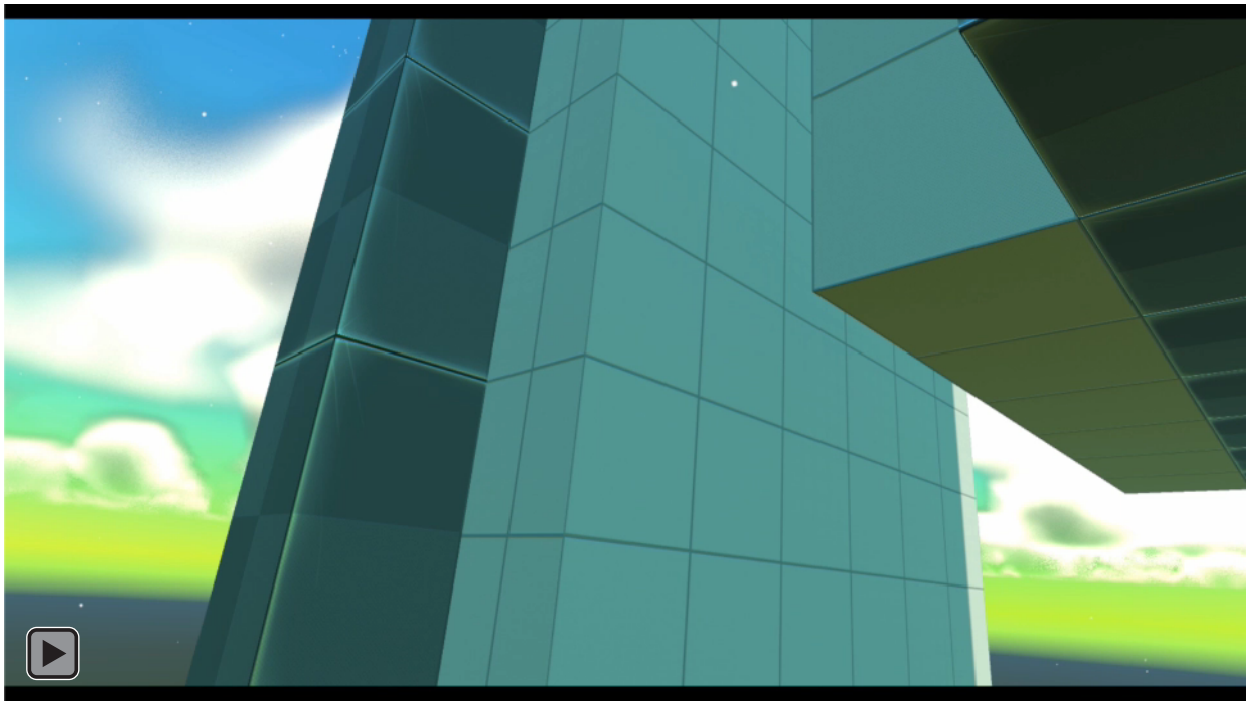
Scrolls V: Skyrim VR”[9] created by Bethesda Game Studios, is a game that was released on 11 November 2011 which has recently been released for PC VR. It’s huge open world map allows for the user to move between ground to water. This is the closest game that I have found which supports seamless transitions between different locomotion forms such as the transition between walking and swimming. While it does support for realistic swimming mechanics they, unfortunately, are not very user-friendly to a point where it’s almost impossible to use. The main issue with their swimming locomotion is that the forward direction is always changing depending on when your arms are moving, this causes you to move backward if you do not release the trigger buttons at the end of your stroke before bringing your arms back in front of you to perform another stroke. Alongside the in-game water current effect causing players to move with the current, it makes the realistic swimming in the game almost impossible to get to where you want to move to. After personally trying the swimming locomotion in the game, I ended up turning the feature into the more traditional movement with the trackpad.



Video 04: *Gameplay footage of Skyrim Swimming*

[9] *The Elder Scrolls V: Skyrim VR*, Microsoft Windows, (Bethesda Game Studios, 2018).

While researching all these different locomotion's from different games, they have all only had one or two options. The most combined locomotion system besides Skyrim consisted of walking/climbing and jumping from the game "Climbey"[10]. What my project intends to achieve, is to combine all the locomotion systems into one. Players will be able to ride vehicles similar to Hover Junkers, then either jump off or exit the vehicle and continue exploring the world. Another element missing currently is the transition between water and land. Games that allow for underwater exploration limit the player to the water. My project will break this limitation by allowing the player to emerge from the water and continue on foot on the land.



Video 05: *Gameplay footage of Climbey Locomotion*

[10] Brian Lindenhof, *Climbey*, Microsoft Windows (ShadowBrain Games, 2016).

2.2 - Interaction

This project will also take considerations discussed by Jason Jerald in his book, *"The VR Book: Human-Centered Design for Virtual Reality"*[11]. The VR book explores many different aspects of VR such as the "*illusion of physical interaction*"[12], which allows interaction to use substitutes feedback to mimic a realistic feel. Why this is important for locomotion is that my project is not just pushing down on the touchpad to move. It involves grabbing hold of the environment to climb and grabbing interactable objects to steer vehicles. By adding the illusion of physical interaction alongside the locomotion, I will be able to create a more immersive experience for the user. This illusion can be created by giving haptic feedback while a player is holding onto something or even just a visual highlight around the controller. Defined by Mel Slater and Sylvia Wilbur, "Immersion is the objective degree to which a VR system and application projects stimuli onto the sensory receptors of users in a way that is extensive, matching, surrounding, vivid, interactive, and plot informing."[13] When trying to make a VR experience immersive, the first feature that I will try to adhere to is making the experience match what you would experience in the real world. This is why room scale VR is so immersive, as it gives the user the ability to physically move around. Because of this reason, this project will attempt to further the user experience for locomotion in a direction that would not be commonly found in VR games.

[11] Jason Jerald, *The VR Book: Human Centered Design for Virtual Reality* (Association for Computing Machinery and Morgan & Claypool, 2015).
[12] Jerald.
[13] Mel Slater and Sylvia Wilbur, "A Framework for Immersive Virtual Environments (FIVE): Speculation on the Role of Presence in Virtual Environments.," 1997.

2.3 Motion Sickness

Jason Jerald also analyses the issues of motion sickness which is one of the main problems that this project will face, stating that motion sickness is *"the greatest risk for VR"*[14]. Ajoy S Fernandes and Steven K. Feiner researched and created a way of reducing motion sickness by creating a method which dynamically modifies the users Field of view. They have stated that "*It has been noted that decreasing field of view (FOV) tends to decrease VR sickness, though at the expense of sense of presence."*[15] By decreasing the field of view for the user, the amount of information through visual cues that is experienced by the user is reduced, as this method decreases "*the conflict between visual motion and felt motion"*[16]. However, as mentioned by Fernades and Feiner, decreasing the field of view also decreases the sense of presence, damaging the amount of immersion felt by the user. What the duo investigated was "*the utility of dynamically decreasing FOV in situations in which a larger FOV would be likely to cause VR sickness, and dynamically restoring it in situations in which VR sickness would be less likely to occur*"[17]. Furthermore, they wanted to be able to change the FOV in such a way that it was so subtle that users would not perceive that any change in the FOV was taking place. Thus, minimizing the decrease of the noticeable sense of presence that the user would experience with a lower FOV. The software testbed that they created involved having an in-game vignette appear on the screen as the user moved through the virtual environment. The vignette would also disappear as the user stops moving. These vignettes would not just appear/disappear straight away, but gradually fade in. Through multiple testing, they were able to find out that with the current headset they were using, which started at an initial FOV of 120 degrees, decreasing the FOV to 90

[14] Jerald, *The VR Book: Human Centered Design for Virtual Reality*.
[15] Fernandes and Feiner, "Combating VR Sickness through Subtle Dynamic Field-Of-View Modification."
[16] Jerald, *The VR Book: Human Centered Design for Virtual Reality*.
[17] Fernandes and Feiner, "Combating VR Sickness through Subtle Dynamic Field-Of-View Modification."

degrees the majority of participants were not able to notice any change in FOV. While decreasing the FOV to 80 degrees does reduce discomfort, many participants were able to detect that the FOV had changed. Not everyone is the same when it comes to motion sickness. Throughout researching into locomotion, I have found myself more adapt to different movement within VR which may cause motion sickness to some. However, with games that have used the dynamically changing FOV method, I have found myself being able to play them for a longer time than games without it.

3 - Methodology

As this project is about researching different locomotion's, the end result is to have a fully working locomotion prototype, therefore this project is both practice-based and practice-led. Linda Candy defines practiced-based as "*an original investigation undertaken in order to gain new knowledge partly by means of practice and the outcomes of that practice*"[18]. Meanwhile practice-led research "*is concerned with the nature of practice and leads to new knowledge that has operational significance for that practice"*[19]*.* Action research will play a big part in my methodology as "this type of research is an examination of the way in which practitioners reflect on their actions during and following their work."[20]

[21]

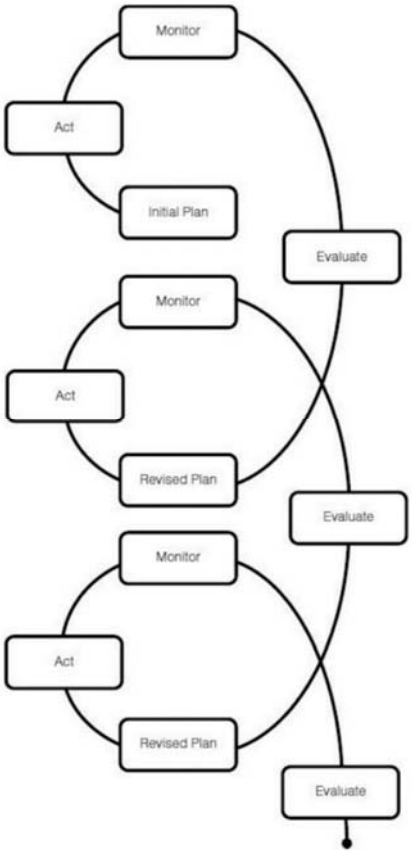


Figure 02*. Action Research Diagram*

Source: *digital image, accessed April 14, 2018. Research for Designers : A Guide to Methods and Practice* (SAGE publications Ltd, 2015)

[18] Linda Candy, "Practice Based Research: A Guide" (University of Technology Sydney, November 2006).
[19] Candy.
[20] Donald Schon, *The Reflective Practitioner: How Professionals Think In Action* (Basic Books, 1984).
[21] Gjoko Muratovski, *Research for Designers : A Guide to Methods and Practice* (SAGE publications Ltd, 2015).

The above diagram from Gjoko Muratovski's "Research for Designers"[22] is the base action research methodology that I have incorporated into my own methodology. When creating the different locomotion's, I first split them into their own cycles. Once I have evaluated and tweaked them to a point that I am satisfied with, I either start working on the next form of locomotion or start merging that locomotion with the others.

3.1 - Initial Plan

In this step, I start to research what similar locomotion systems are out there to the one that I want to make. For example, for my climbing locomotion, I went through the steam store and looked for different types of games that involved being able to climb. I would then play the game and critically analyse how the locomotion mechanics worked. This involves me writing down as I play how the interaction works with the game environment, and what I action I would need to take to make the locomotion work. I would also analyse the comfort levels that I experienced while playing and if I felt any nausea. From my written analysis, I would categorise positives and negatives from my gameplay experience. By creating these two categories, I can gather the positive feedback from my experience and incorporate it into my own locomotion system. However, sometimes negative feedback will also need to be added, as the feedback could come from the limitation of the hardware. If at this point I had already created other forms of locomotion, then I will also plan out how this new locomotion will be able to combine with the others. This is to make sure that there will be no conflicts with input from the controllers. Erik Bethke who wrote "*Game Development and Production*"[23] points out that "*A critical diagram to*

[22] Muratovski.
[23] Erik Bethke, *Game Development and Production* (Wordware Publishing, 2003).

create is the controller diagram."[24] The below diagram shows the control scheme from a cancelled game called "Black 9" [25]. He uses this diagram to demonstrate how to layout a controller when developing.



***Figure 03**. Image of controller layout for Black 9.*

***Source:** digital image, accessed April 14, 2018.* Erik Bethke, *Game Development and Production* (Wordware Publishing, 2003).

3.2 - Act

This step involves starting to write the code for the locomotion system. From my study and plan step, I already have an idea of how the code should be written. While this step is only about writing code, it is one of the steps that takes the longest and is also one of the steps that can cause most of the issues. As I continuously go through iterative cycles, the code continuously keeps changing as I

[24] Bethke.
[25] *Black 9*, Sony Playstation 2, Xbox (Taldren, n.d.).

find issues from testing and also add small tweaks to make the locomotion more fluid.

3.3 - Monitor

From here, I begin to playtest what I have created. While playtesting I need to make sure that the code is working properly. In this step, I need to not only test if it's working properly but also test if I am able to break the game. This is to find the limitations of the locomotion system that I have currently made. All the data collection that need to be evaluated will come from this step. This is a very important step as playtesting helps to "*determine weak points of the game*"[26] and also helps to bring out new ideas for improvement.

3.4 - Evaluate

After playtesting, I reflect upon what happened throughout the playtest. Jen McNiff and Jack Whitehead recommend using the following questions with an action research process:

- What am I doing?
- Do I need to improve anything?
- If so, what?
- How do I improve it?
- Why should I improve it?

Essentially, finding out what was good and what was bad from my playtest. One of the main evaluating areas for this step will be tweaks to int and float values that need to be changed in the code. These int and float values will be variables such as speed, and if these values need to be increased or decreased.

[26] Timofei Viktorov, "CHALLENGES IN GAME DEVELOPMENT Playtesting and Prototyping as Important Steps in Game Development" (Lahti University of applied sciences, 2015).

3.5 - Revised plan

After one cycle, I begin another cycle. This new cycle is a continuation from my previous cycle. From all the information that I have gathered in my reflection, I transfer that into the Study and Plan step of the new cycle. However, this time the research is more about the locomotion that I have created. This step still can involve researching into previously made games, to see the difference between mine and theirs. But its more about gathering what has gone wrong in the previous cycle, and finding a way to solve these issues. Sometimes a new cycle can be quite short, such as if I need to tweak some values to speed up/down movement speed. Or it could take quite some time as I find that my code doesn't work the way I want, so need to rewrite it all. Once I have tested and the locomotion system I have created is at a playable state. I can begin combining the locomotion systems together.

3.6 - Iterative Design

As this project is fundamentally a prototype for locomotion within a video game, it's good to get a base knowledge of game production methodologies. By using an iterative design methodology, game developers are able to quickly create prototypes and begin testing very early in development. Katie Salen and Eric Zimmerman who co-wrote the "Rule of Play – Game Design Fundamentals"[27], define iterative design as a "play-based design process. Emphasizing playtesting and prototyping, iterative design is a method in which design decisions are made based on the experience of playing a game while it is in development. In an iterative methodology, a rough version of the game is rapidly prototyped as early in the design

[27] Katie Salen and Eric Zimmerman, *Rules of Play - Game Design Fundamentals* (MIT Press, 2003).

process as possible”[28]. Through rapid prototyping, this project can get the foundations of each of the different locomotions. This allows getting an understanding of what does and doesn’t work for the current locomotion being worked on.

The benefit of using this methodology is that it allows for constant improvement to the project. By taking this iterative design approach, I can continuously improve and add to the current locomotion system that I have created. One issue with this methodology is knowing when to limit how many cycles to continue on with. As someone who is still constantly learning about C# coding for Unity 3d, I am always learning new ways to code which can improve performance.

[28] Salen and Zimmerman.

4 - Project Development/Documentation

4.1 - Walking/Running

Walking/Running is probably the main form of locomotion in any video game, so was the first locomotion that I began working on. From researching many VR games, the most common walking movement was through using the touchpad on the controller to control forwards and sideways movement. To begin creating my locomotion system, I needed to get an understanding of how the input from the controller's work. The below diagram shows a diagram of the Vive controller, numbers 1,2,7 and 8 are all the input options available for use within unity.

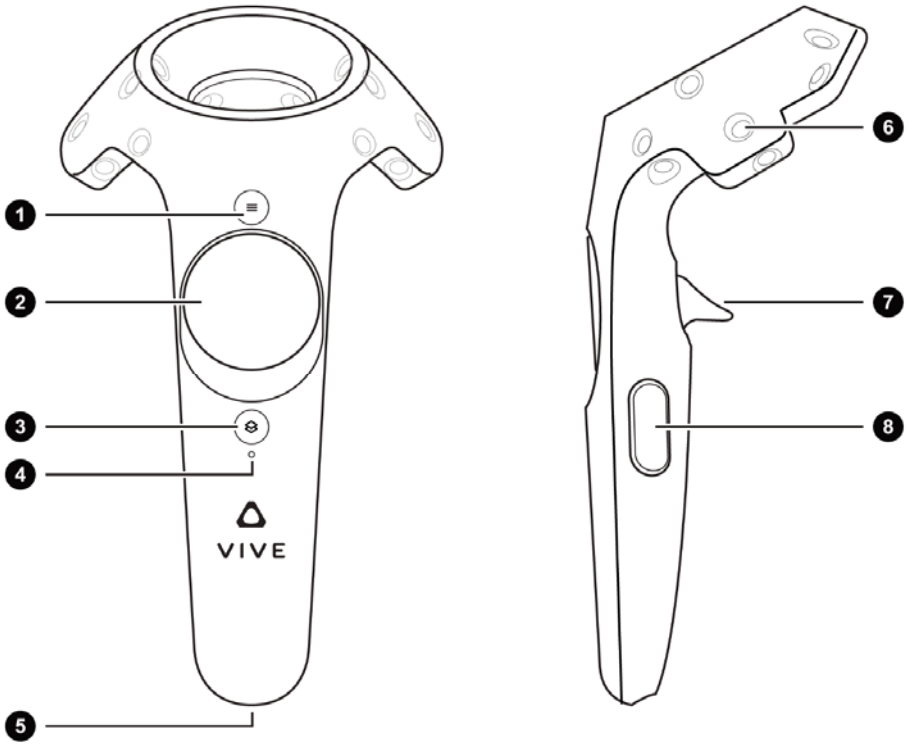


***Figure 04**. Vive Controller Diagram*

***Source:** OculusControllersViveControllers.png, digital Image, accessed January 08, 2018. https://docs.unity3d.com/560/Documentation/Manual/OpenVRControllers.html*

The trackpad, number 2, contains X and Y values depending on where the user's thumb is on it. The trackpad can also be clicked

down allowing for button inputs like on a standard gamepad controller.

Gathering the X and Y values from the trackpad and combining that value with clicking the trackpad down, can allow the thumbpad to be used as many different buttons e.g. you can calculate if the trackpad has been clicked on the left or right side.

During my initial plans, I needed to create a controller diagram to plan out what inputs on the controller would be used for each locomotion.

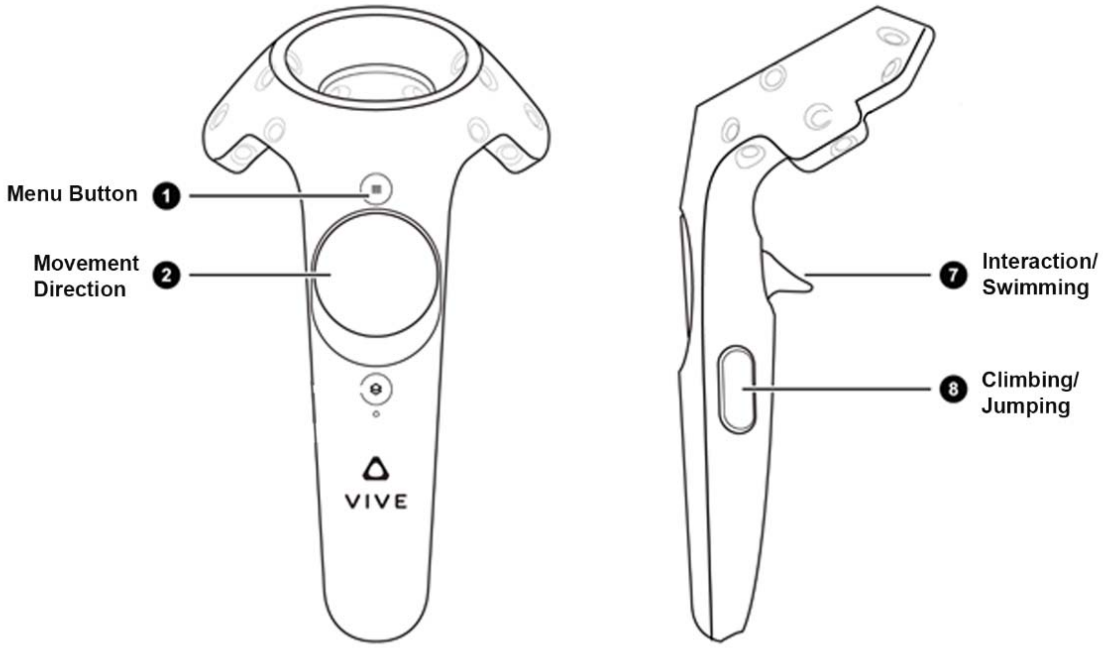


***Figure 05**. Edited Controller Diagram for this project*

(Menu Button) Menu Button: This will be used for opening a menu where the user access features like changing settings or exiting the game.

(Trackpad) Movement Direction: This will be used to control the direction that the user will travel.

(Trigger Button) Interaction/Swimming: This will be used for interacting with interactable objects in the environment. It will also be used for the swimming locomotion.

(Grip Button) Climbing Jumping: This will be used for allowing the player to Jump or Climb.

One of the main reasons for using Unity3d as my game engine for building this locomotion system is because Valve Corporation released a plugin called SteamVR Plugin. This plugin comes with everything you need for the tracking to be received by Unity 3d as it comes with a prefab called Camera Rig. The Camera Rig can be dragged and dropped into your scene and you will not have to worry about setting up the tracking for the headset and controllers. The below script shows my original setup for receiving inputs from the controller. This script will be attached to each controller in the Camera Rig Prefab.

```
private Valve.VR.EvrButtonId triggerButton =
Valve.VR.EVRButtonId.k_EButton_StreamVR_Trigger;

private Valve.VR.EvrButtonId touchButton =
Valve.VR.EVRButtonId.k_EButton_StreamVR_Touchpad;

private Valve.VR.EvrButtonId gripButton =
Valve.VR.EVRButtonId.k_EButton_StreamVR_Grip;

private SteamVR_Controller.Device controller;

private SteamVR_TrackedObject trackedObj;

private Vector2 TouchPadDirection;

void Start(){

      trackedObj = GetComponent<SteamVR_TrackedObject>();

      controller = SteamVR_Controller.Input((int) trackedObj.index);

}
```

```
Void Update(){
      if(controller.GetPressDown(triggerButton)){
            Debug.Log("Trigger Pressed");
      }
      TouchPadDirection =
controller.GetAxis(Valve.VR.EVRButtonId.k_EButton_SteamVR_Touchpad);
}
```

Using the forward direction of the headset and combining that with the vector 2 values from the trackpad, creating very simple locomotion can be easily achieved as the same values can be used to make movement in a normal non-VR game. The question from here is how can I make it a more immersive experience?

When creating each different locomotion, I always need to think about if what I have created is the most realistic way to create the most immersive experience. This is the revised plan step in my methodology. The question that needs to be answered is why should I improve it? While it is possible to move through the virtual environment beyond what the room scale movement provides, the current state of this type of locomotion was not immersive enough. I needed to make the movement more realistic. To make this type of locomotion more immersive, I decided to make the person have to run on the spot in conjunction with the trackpad values.

I use the changing value of the Y position of the headset to calculate the speed at which a person is running on the spot. This makes the user move faster through the virtual world by running on the spot faster.

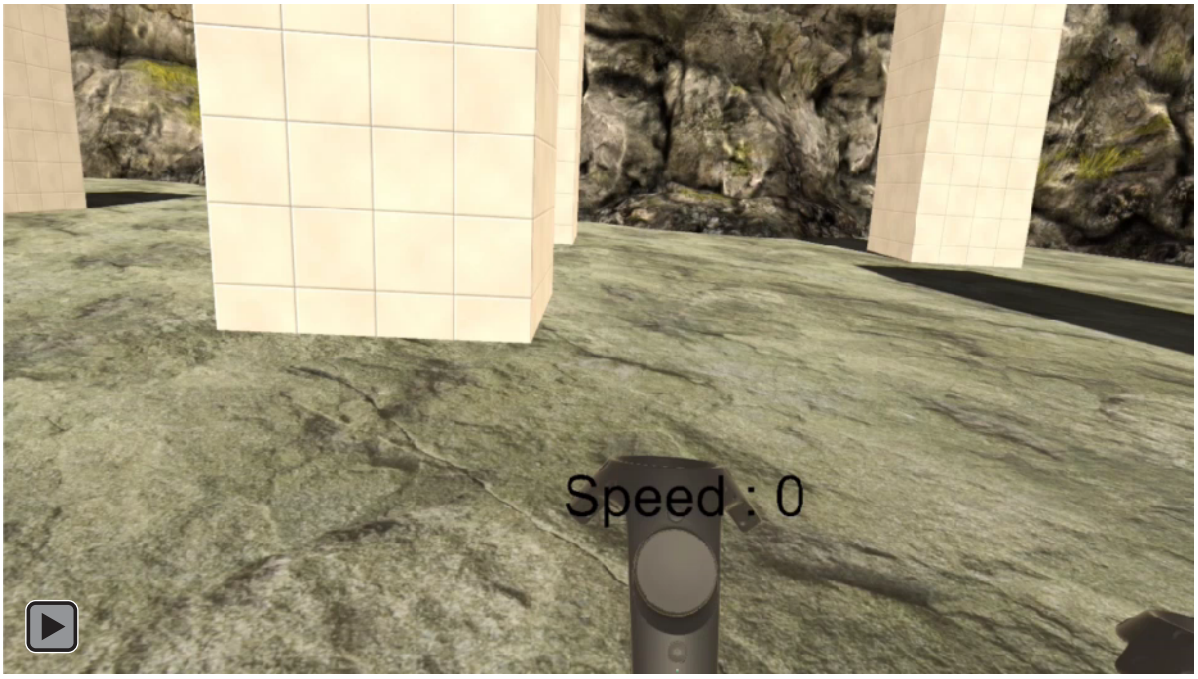
```
      void Update () {
            CalculateSpeed();
            Vector3 forward =
forwardDirection.TransformDirection(Vector3.forward) * trackpadY;
            Vector3 sideways =
forwardDirection.TransformDirection(Vector3.right) * trackpadX;
            moveDirection = (forward + sideways);
```

```
            moveDirection = new Vector3(moveDirection.x, 0,
moveDirection.y);

      moveDirection *= newSpeed * speed;

    }



public void CalculateSpeed(){


            prevDistance = Mathf.Abs(BounceTransform.localPosition.y -
PrevPos.y);
        if (Crouched == false) {
                  moveSpeed = prevDistance * movespeedOffset;
                  if (moveSpeed > 3) {
                        moveSpeed = 3;
                  }

                  if (newSpeed < moveSpeed) {

                        newSpeed = moveSpeed;

                  }

                  if (newSpeed > 0) {

                        newSpeed -= Time.deltaTime * 3;
                        if (newSpeed < 0) {

                              newSpeed = 0;

                        }
                  }
            }

      }
```

One extra line of code that was needed was a raycast check to see if the user was touching the ground. To create this raycast, an empty gameobject was first created, which would be a child of the Camera Rig. This empty gameobject would then follow the positional values of the headset, except the Y value. The Y value would have a slight positive offset as if the gameobject was at the same Y value as the CameraRig, the raycast may pass directly through the ground. While the Y local position of the gameobject had been set as 0.07, I set the length that the ray could cast as 0.17, this was due to how the

ground was not always flat and going up/down slopes would cause the script to calculate the user as not touching the ground. A layer mask also needed to be added to the raycast so that the player wouldn't be seen by it.

```
if (Physics.Raycast (GroundCheck.position, GroundCheck.TransformDirection (Vector3.forward), out hit, GroundDistance, Layermask))
```



Video 06: *Moving Up and Down Locomotion*

While personally, I had no issue with this type of movement, during an informal playtest, one of the users who were very new to VR wasn't able to continuously run on the spot while wearing a VR headset. This was due to fear of not being able to see the real world around them and thinking that they might run into something. After spending a bit more time within the virtual environment they were able to gain more confidence and had no more issues going forward.

Through my work, I participated in the Global Shanghai VR 48 Hour Hackathon. During my time at the hackathon, I could put my

methodology to the test through a rapid prototype of a locomotion system. As my team was given a Vive tracker to use for the hackathon, I based the locomotion system around the tracker being the source of movement. The Vive tracker is an accessory which can be attached to real world objects such as cameras, bodies, and props, allowing them to also be tracked in the virtual environment. Previous versions of my locomotion system use the position of the headset to calculate the forward direction and speed that a player moves at. Using the tracker gave me a lot more helpful feedback than I expected. First off, the main reason for me wanting to use a Vive tracker for the source of the movement is for the direction that the player moves in. Previously, the player would have to adjust the thumb position on the touchpad of the controller when turning their head to keep moving in a straight line. Also, in a different previous prototype, I experimented with using the controller as the forward direction which did seem to help the controls become more intuitive. However, using the controller as the source ran into many problems when a player was faced with swinging a sword around.

Upon testing with the Vive tracker, it became clearly apparent that using the tracker as the source of direction is by far the most intuitive way. Not a single participant seemed to struggle with directions that they wished to move in. What makes room-scale VR such a big step up from other VR systems, is that it gives the user the ability to move physically move around without pressing any buttons. If they want to move to a location in the virtual world, they simply walk over to it like they would in the real world. The next closest thing to this would be walking/running on the spot to reach that location. This is what hardware such as the Virtuix Omni is trying to achieve. These treadmill machines allow the player to mimic walking and the data through their movement will drive the direction they are moving. One of the main problems with these products is that the price is almost the same, if not more, than the VR headset

itself. While seeing the usefulness of the Vive tracker I began to test other ways in which it could be used for locomotion. From this, I purposed to create a semi hands free locomotion system without the use of a treadmill. The only assisted extra hardware I will be using will be the Vive tracker.

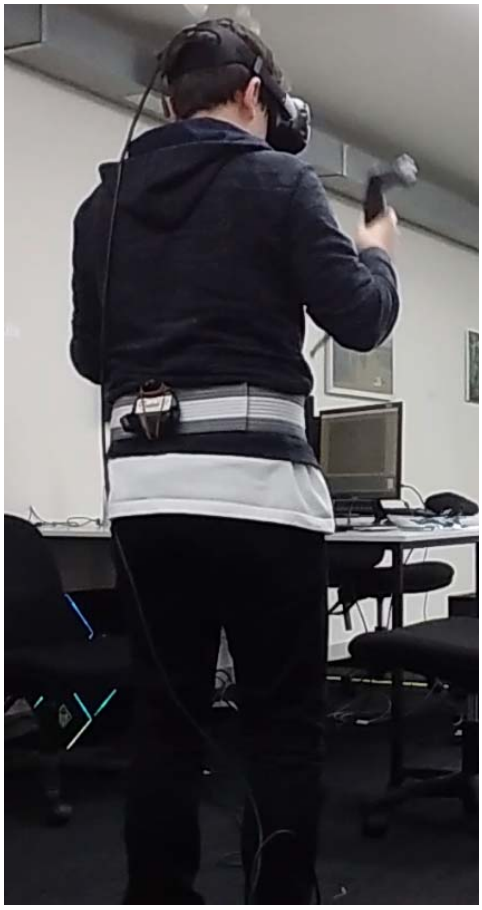


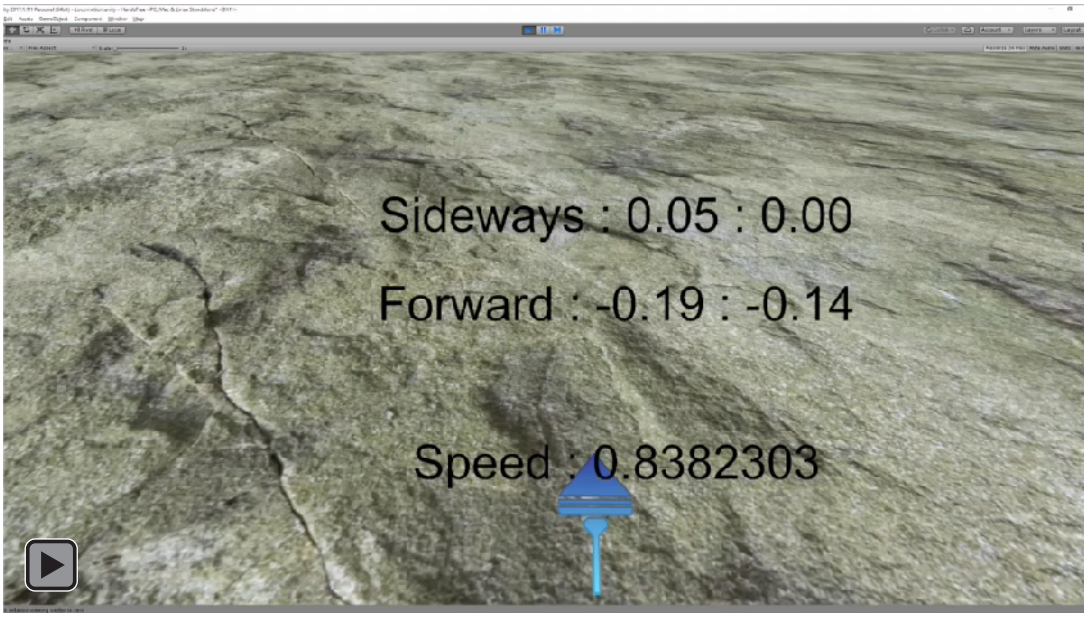
***Figure 06**. Image of tracker attached to the hip*

To create this hands free locomotion, I attached the Vive tracker to the hip of the user and then used the position differential of the tracker and the headset to calculate amount and direction that a user leans in. This lean value will then be translated into a vector2 value which can be used instead of the vector2 value from the trackpad. A calibration script was also created as the vector2 value needs to be set as zero if the user was standing completely straight.



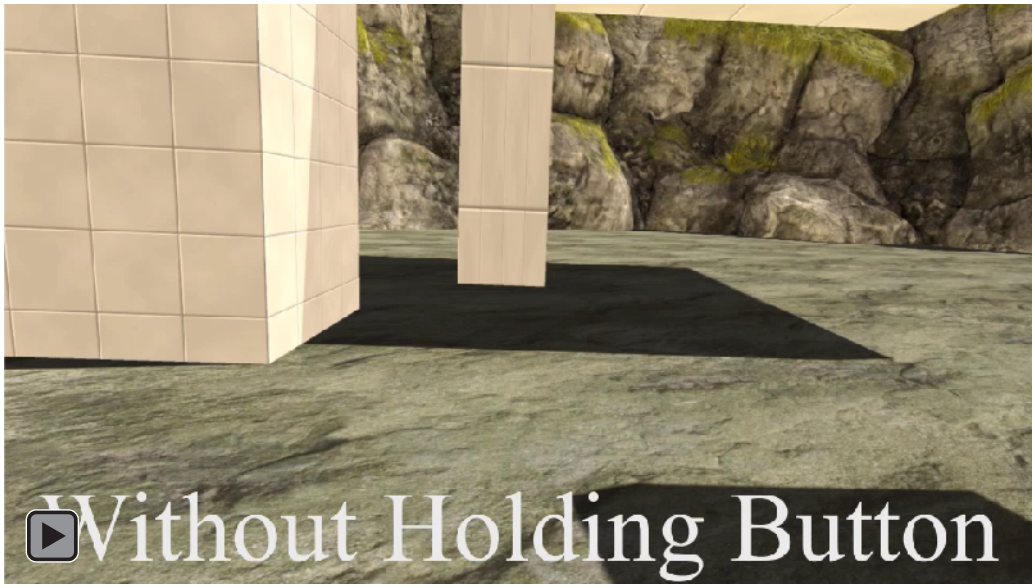
Video 07: *Leaning Values*

As we now have the Vector2 values which can replace the values from the trackpad, we can combine that value with the speed from running on the spot.



Video 08: *Moving Up and Down Leaning Locomotion*

One issue that has arisen from this hands-free locomotion, is that due to forward lean and speed of the y position of the headset, crouching down on the spot is near impossible. With the trackpad movement, all you need to do is not click the trackpad to stop the movement, however, the hands-free locomotion is always calculating. The solution to this was now that the trackpad click has now been freed up, we can use this button as a way for the user to crouch down without it trying to move in the leaned direction. To do this, we just needed to create a bool that would turn to true when the trackpad has been clicked.



Video 09: *Crouching Example*

4.2 - Swimming

The way that I wanted my swimming locomotion to work is something that had never been created before. While there has been some underwater locomotion VR games. All the games I have come across use the touch has the input for the control and direction of the movement. My swimming locomotion system is to allow the user to give a sense of realism as though they are swimming. Allowing players to move their arms through the water to give them the speed and direction.

My inspiration for this type of swimming came from the video game Final Fantasy X[29]. In Final Fantasy X, you can play a mini game called Blitzball, which is almost like an underwater version of American football. The main character Tidus uses breaststroke to swim through the water. After watching other swimming strokes, I decided to use the breaststroke for the swimming locomotion. There were two reasons why I chose this stroke. After watching videos about underwater swimming, many of these athletes were using the breaststroke to move through the water. The second reason was that in my mind I believe it to be the easiest stroke to create through code.

To begin, I needed to find a way to calculate the direction the player would move when moving. Normally VR developers have been using the forward direction of the head or controller as the forward movement. With my locomotion, I always want to give the player the ability to move in the straight line and can look where ever they want. Because of this, I decided to make the forward direction the forward direction from the player's body towards the center point of

[29] *Final Fantasy X*, Sony Playstation 2 (Sqaure Enix, 2001).

the two controllers. My first step was to calculate the middle point between the controllers.

```
Void Update ()
{
	MidObject.position = (RightHand.position + LeftHand.position)/2;
}
```

I then added an object that followed the head and would always face this middle position.

One flaw that I found with this, was that when the player does the breaststroke and their hands go behind their body, the forward direction would then be the complete opposite way.

To counter this flaw, I added an offset to the object that followed the head and made it be way behind the way so that the player's arms would never reach it. After more experimenting, this step was not needed as I would lock the direction the player would move when the player pushes both controllers. This way they would always travel in that one direction when doing a single breaststroke.

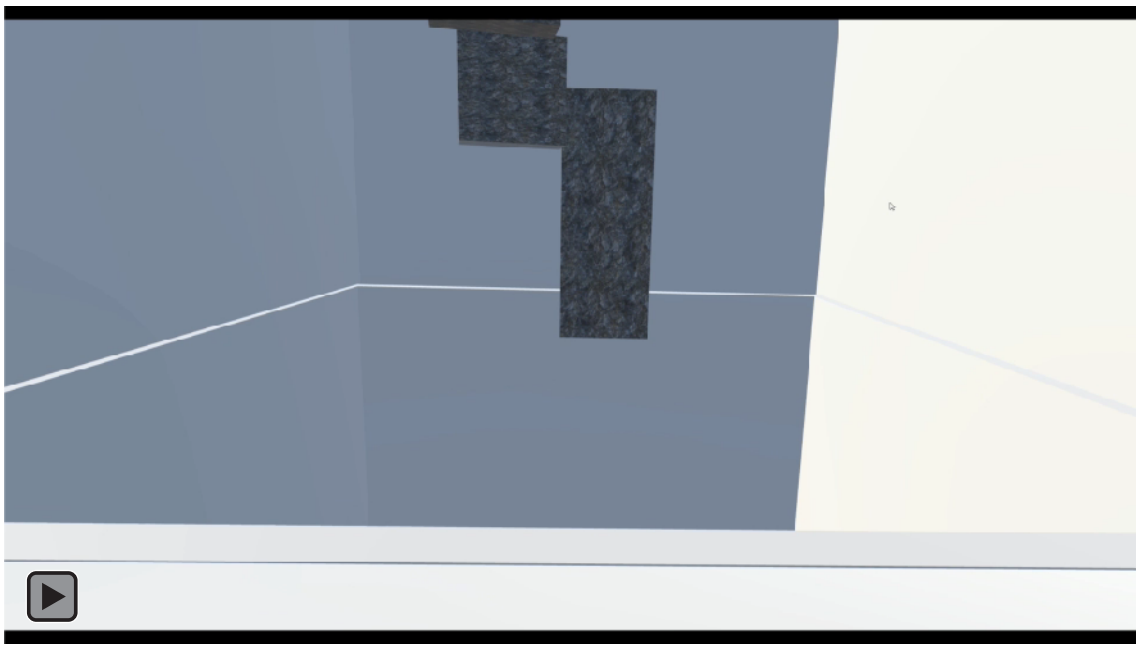
```
if (RightTrigger == true && LeftTrigger == true) {
	if (SetForward == false) {
		SwimLook.position = transform.position;
		Midpoint = (RightHand.position + LeftHand.position) / 2;
		Lookpoint = new Vector3 (Midpoint.x, Midpoint.y, Midpoint.z);
		SwimLook.LookAt (Lookpoint)l
		SetForward = true;
	}
```

```
If (SetPreDist == false) {
	SetPreDist = true;
	PreDistance = Distance;
}
If (Moving == false) {
	Moving = true;
}
Distance = Vector3.Distance (SwimDistance.position, Lookpoint);
if (PreDistance > Distance) {
	MoveSpeed += (PreDistance – Distance) * Speed;
	If (MoveSpeed > 5) {
		MoveSpeed = 2;
	}
}
PreDistance = Distance;
} else {
	Moving = false;
	SetPreDist = false;
}
```

The previous code I came up with was to make the empty gameObject called FaceDirection look at the midpoint. This direction that FaceDirection is in will be used to move the player in that direction. When creating my previous locomotion from last year, I locked the direction the player was moving in the direction their head was looking. Because of this, when a player wants to look to the side they will start travelling in that direction. Using the direction from FaceDirection will allow players to look around the world without changing the direction that they are travelling in.
I locked off the Y value to just get a feeling of how this movement felt in VR.

I was actually quite surprised how well this felt, more in the way that I did not feel any motion sickness. My only feedback to myself is to a bit of a glide towards the movement. Having it move and stop without any deacceleration did feel a bit weird. To fix this, I added code to decrease the MoveSpeed every frame which is the speed that the player is moving in. Decreasing this over time creates a drift to the movement. This gave the swimming a more natural feel towards the movement and made it feel really smooth.

```
if (MoveSpeed != 0){
	Player.Translate (SwimLook.forward * Time.deltaTime * MoveSpeed * (Speed/100f));
	MoveSpeed -= Time.deltaTime * 1.0f * StopSpeed;
	if (MoveSpeed <= 0) {
		MoveSpeed = 0;
		}
	}
```



Video 10: *Swimming Locomotion*

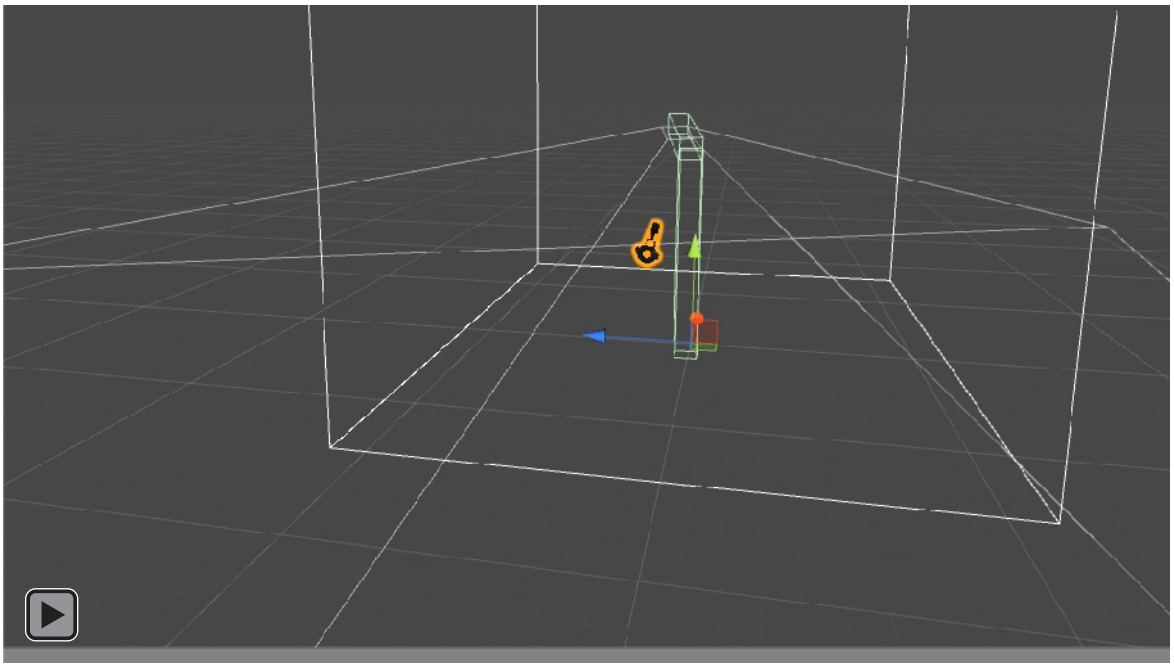
My next step to do is to create a collider so that the user can't swim through objects. In a normal game adding a collider would be quite simple, however, as this is VR it's not quite as straightforward. Because the player can freely walk around while in VR, adding a collider to the Camera Rig object would only stop them going through objects in they were to stand only in one spot. Adding the collider to the head doesn't work either as the controllers are not parented to the head, so if the head was to be pushed by a wall, the controllers would not follow.

The solution to this is to put the collider onto the Camera Rig and then create a script that would cause the collider to follow and change according to the head.

```
using UnityEngine;
using System.Collections;

public class ViveCollider : MonoBehaviour {

	public BoxCollider BC;
	public Transform Head;

	// Update is called once per frame
	void Update () {

		UpdateCollider ();

	}

	public void UpdateCollider(){

	BCsize = Head.localPosition.y;
	  if(BCsize <= 0.1f)
	  {
	      BCsize = 0.1f;
	  }
	      BC.size = new Vector3 (CS, BCsize, CS);
	      BC.center = new Vector3(Head.localPosition.x, BCsize/2,
Head.localPosition.z);


	}

}
```

This code makes the collider follow the position of the head by copying its local position. The collider also scales depending on where the player is located on the Y axis as well.

When this project started using the Vive tracker, this collider also had an upgrade. Due to the tracker's position on the hip, a collider can be created from the ground position to the hip position, then another collider can be created from the hip position to the head. This new collider setup allows the user to be able to stand on a platform and look over the edge without falling off. Previously, as the collider would follow the heads position, if a user looked over an edge, the colliders position would be over that edge, which would cause the user to fall down in the virtual environment. To set this collider up, an empty object with a box collider needs to be created and parented to the Camera Rig, this empty object will be the top collider that will follow the headset. The code will be exactly the same, only that this object will need to look at the hip tracker position. By having this object look at the hip tracker, it will rotate to match the diagonal position of the angle difference between the headset and hip tracker.



Video 11: *Double Body Collider*

Because my locomotion system isn't just for swimming, there is still quite a lot of work to be done. When a player moves their head out of the water I need to stop the player from being able to keep moving up, and also lock the positive Y direction if they want to keep on swimming. Without this, the player would be able to keep on swimming into the sky. To achieve this, a collider was added to both the water in the game and the headset. This collider needs to be set as a trigger or else it will stop the user from being able to go into the water. By attaching these colliders, we can register when the headset is in or out of the water. As we now have a collider on the body and the head, we can check if the body is still in the water while the head is above the surface. Now, all that needs to be done is to set the Y movement direction to 0 if it is positive. If the Y direction is positive, that means that the user is moving upwards. A negative value means that the user is moving downwards so we don't need to set that as 0 to allow the user to be able to dive back down into the water.

```
void Update()
{

    if (bodyInWater == true)
    {

        CalculateSwim();

        if (locomotion.Grounded == true)
        {
            RB.useGravity = true;
        }
        else
        {
            RB.useGravity = false;
        }

        CameraRig.Translate(moveDirection * moveSpeed * Speed);
        prevDistance = Distance;
        moveSpeed -= Time.deltaTime;
        if (moveSpeed <= 0)
        {
            moveSpeed = 0;
```

```
            }

        }

    }

    private void CalculateSwim()
    {
        midPosition = (RightHand.localPosition + LeftHand.localPosition) /
2;
        Distance = Vector3.Distance(midPosition,
playerCamera.localPosition);
        if(Distance < prevDistance)
        {
            float moveDistance = prevDistance - Distance;
            moveDistance *= 100;
            if (moveSpeed <= moveDistance)
            {
                moveSpeed = moveDistance;
            }
            if(headInWater == false)
            {
                if (moveDirection.y >= 0)
                {
                    moveDirection = new Vector3(moveDirection.x, 0,
moveDirection.z);
                }
            }

        }

    }

    public void BothTriggersPressed()
    {
        if (bodyInWater == true)
        {
            midPosition = (RightHand.localPosition +
LeftHand.localPosition) / 2;
            moveDirection = (midPosition -
playerCamera.localPosition).normalized;
            prevDistance = Vector3.Distance(midPosition,
playerCamera.localPosition);
            CanSwim = true;
        }
    }
```



Video 12: *Final Swimming Locomotion*

4.3 - Jumping

There were two ways which could be used for jumping for this project. Through the VR games that I have researched, there have been two different ways in which developers have created this locomotion. One was to have the users to press one of the buttons, which would have the user jump in the virtual world straight away. The other way was to have the user swing their arms while holding down a button and then letting go, the velocity that the user swung their arms would be used to calculate the distance that the user would jump. This project will be using the latter method as by having the user swing their arms gives a sense of more control over the locomotion which will give a more immersive experience.

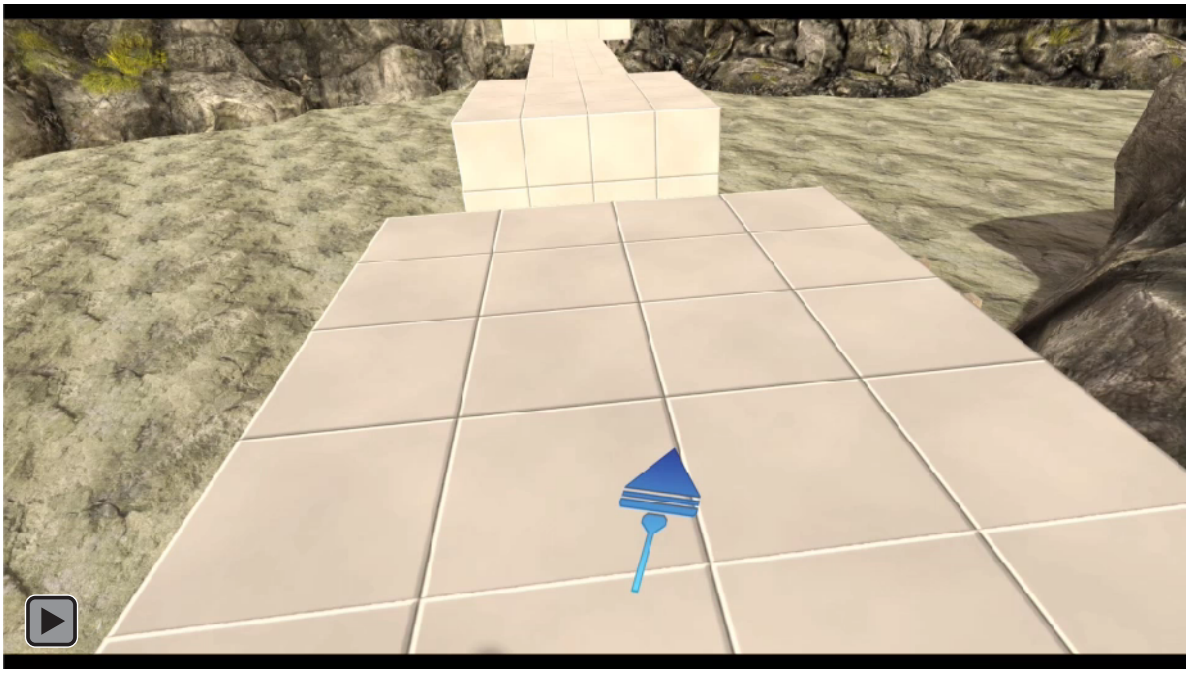
My initial plan for creating this locomotion was to use a similar approach that I had done with creating the swimming locomotion. The idea was to use create an empty game object which would be the middle position of the two controllers when both the grip buttons were pressed. At each frame, the current position would be referenced and then used in next frame with the new current position to calculate the direction that the middle position is travelling. The speed and direction would be multiplied together to give the height and direction that the user would travel in. When the user lets go of the grip buttons the locomotion would activate with the current calculated height and direction for that frame.

```
	void Update () {

		transform.position = (Left.position + Right.position) / 2;

		MidDirection = (transform.localPosition - PrevMid).normalized;
		MidSpeed = Vector3.Distance (PrevMid,
transform.localPosition);

		PrevMid = transform.localPosition;


	}
```

```
public void Jump(){

      Player.AddForce(MidSpeed * 8000);

}
```

After testing I found that the jump wasn't giving enough upwards force. Another issue that needed to be fixed was that the jump could continuously be performed, however, this was quite an easy fix as we can use the raycast from the walking/running script which checked if the user was touching the ground or not.

```
public void Jump(){

  if (Player.isGrounded == true)
  {
      Player.AddForce(Vector3.up * MidSpeed * 8000);
      Player.AddForce(Direction.TransformDirection(Vector3.forward)
* MidSpeed * 2000);
  }
}
```



Video 13: *Jumping Locomotion*

4.4 - Climbing

Climbing was the third locomotion system that I worked on after swimming. From creating the swimming locomotion first, it gave me more insight into how to start coding the climbing locomotion. My original idea was that when the user would grab the climbable object the code would reference that spot's position. From there, if the player moves their hand around, the distance from the original position would become an offset to where the user's body would move to.

After a few tests and tweaks, the code below is what I ended up with.

When the grip on the controllers gets pressed, it calls either RightClimbing or LeftClimbing depending on what hand it is. The function will then tell the other hand that it has grabbed the climbable object so that the other hand will let go.

```
if (isClimbing == true) {

if (LeftClimb == true) {


PosX += (HandPoint.x - LeftHand.position.x);
PosZ += (HandPoint.z - LeftHand.position.z);
PosY += (HandPoint.y - LeftHand.position.y);

} else {

PosX += (HandPoint.x - RightHand.position.x);
PosZ += (HandPoint.z - RightHand.position.z);
PosY += (HandPoint.y - RightHand.position.y);

}

transform.position = new Vector3 (PosX, PosY, PosZ);

}

public void LeftClimbing(Vector3 HandPos){

ClimbJump = 1.0f;

RightTouch.OtherHandClimb ();

LeftClimb = true;
```

```
if (isClimbing == false) {
      isClimbing = true;
}

RB.useGravity = false;
RB.isKinematic = true;

HandPoint = HandPos;
PosX = transform.position.x;
PosY = transform.position.y;
PosZ = transform.position.z;

      }



public void RightClimbing(Vector3 HandPos){

ClimbJump = 1.0f;

LeftTouch.OtherHandClimb ();

LeftClimb = false;

if (isClimbing == false) {
      isClimbing = true;
      }

RB.useGravity = false;
RB.isKinematic = true;

HandPoint = HandPos;
PosX = transform.position.x;
PosY = transform.position.y;
PosZ = transform.position.z;

}


public void LetGo(){

      if (isClimbing == true) {

            isClimbing = false;

                  RB.useGravity = true;
                  RB.isKinematic = false;
                  RB.AddForce (ClimbSpeed * ClimbJump);

      }
}
```

While this code worked well, there were two main issues with it. The first issue was that the player was able to pull themselves through walls which could then break the game as they may fall through the world if they let go. The second was that I would need to create climbable objects above platforms for users to be able to climb up something and stand on the top.

To fix the collider issue of the user being able to move their body through walls, the current code causes the player to instantaneously be at the new position, which would cause no collision with the collider on the climbable object. This means I would need to have the user move towards the new position instead of just setting that new position.

```
transform.position = Vector3.MoveTowards(transform.position, new
Vector3(OriPos.x - OffX, OriPos.y - OffY, OriPos.z - OffZ), 0.1f);
```

By using Vector3.MoveTowards I was able to fix this issue. I set the step value to 0.1 so that it was fast enough to feel the same as the previous code and also slow enough to be able to collide with objects.

To solve the issue of how to get the player to get on top of something while climbing, I researched into many of the current VR games that use climbing. All of them use a system where if the head was above ground and the let go, the player would then be moved up until their feet are on the ground.

```
public void LetGo(){

if (HoldingLeft == false && HoldingRight == false) {

      PP.UnParent ();
      GrabParentHold.parent = null;
```

```
        LS.Climbing = false;
        Holding = false;

        GroundDistance = Head.localPosition.y + 0.01f;
         RaycastHit hit;

        if (Physics.Raycast (Head.position, transform.TransformDirection
        (Vector3.down), out hit, GroundDistance)) {

        Debug.Log ("MovePlayerUp");
         moveUpLocation = hit.point;
        MoveUp = true;

        } else {

        RB.isKinematic = false;
        RB.useGravity = true;
        LowerBC.enabled = true;
        UpperBC.enabled = true;

              }
        }
}
```

When the player lets go, the script will call a function. In this function, it will shoot a raycast down towards the ground and detect how far away the ground is. If the ground is within a certain distance, then the player will be moved up.

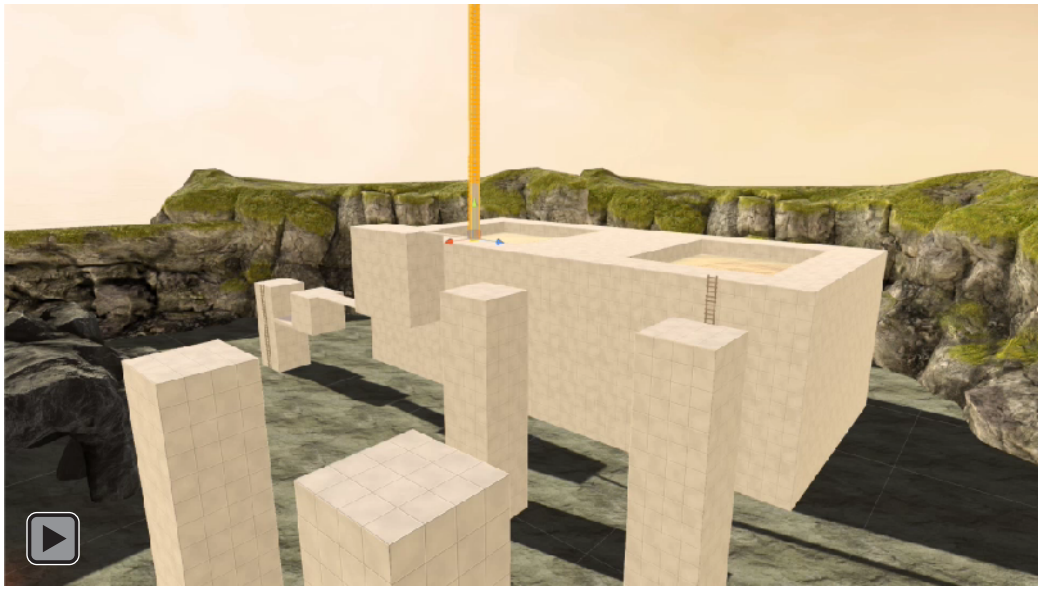
```
if (MoveUp == true) {

GroundDistance = Head.localPosition.y + 0.01f;

        if (Physics.Raycast (Head.position, transform.TransformDirection
        (Vector3.down), GroundDistance)) {

              Debug.Log ("MovePlayerUp");
              Player.Translate (Vector3.up * 0.05f);
                  if (moveUpLocation.y < Player.transform.position.y)
                  {
                      MoveUp = false;
                      RB.isKinematic = false;
                      RB.useGravity = true;
                      LowerBC.enabled = true;
                      UpperBC.enabled = true;
                  }
```

To try to fix the issue of being shot up into the sky, as soon the player releases both the grip buttons on the controller, I use the previous raycast to get the Vector3 position of the hit. From them, I can see if the Player is currently above that position or still below it. If the player is now above it, then moving up should stop.

```
if (Physics.Raycast (Head.position, transform.TransformDirection
(Vector3.down), out hit, GroundDistance))
{
	Debug.Log ("MovePlayerUp");
	moveUpLocation = hit.point;
	MoveUp = true;
}
```

```
if (Physics.Raycast (Head.position, transform.TransformDirection
(Vector3.down), GroundDistance)) {

	Debug.Log ("MovePlayerUp");
	Player.Translate (Vector3.up * 0.05f);
	if (moveUpLocation.y < Player.transform.position.y)
{
		MoveUp = false;
		RB.isKinematic = false;
		RB.useGravity = true;
		LowerBC.enabled = true;
		UpperBC.enabled = true;
	}
}
```

This project focuses on not only creating locomotions that are currently being used in VR gaming, but also on how to improve or enhance them as well. Climbing in VR is something that has always been quite immersive as the user physically moves to climb in the virtual world. Through researching climbing games, there are some that allow movement of the environment while climbing e.g. holding onto a platform as it moves. I haven't been able to find any examples of how rotational environment changes. To attempt to solve this issue I used a ladder that can be climbed, once the user reaches a certain height on the ladder, it would start to fall down.



Video 14: *Ladder Fall*

While you could just parent the Camera Rig to the ladder, this would cause the world rotational values of the Camera Rig to change as the ladder falls down. The solution to this is to create two empty game objects, one would position itself to the position that the grabbing hand is located at. After going to that position, this empty game object would become the parent of the Camera Rig. The second Game Object would also go to the position of the grabbing hand, but would also parent itself to the falling ladder. The first Game Object which now has the Camera Rig as a child would follow the position of the second Game Object.

```
public void LeftHold(Transform Obj){

  HoldingLeft = true;


    if (Holding == false) {
          GrabParent.position = LeftHand.position;
          GrabParentHold.position = GrabParent.position;
          GrabParentHold.parent = Obj;
          Player.parent = GrabParent;
          LS.Climbing = true;
          Holding = true;
          //RB.isKinematic = true;
          RB.useGravity = false;
          LowerBC.enabled = false;
```

```
            UpperBC.enabled = false;
            HeadBC.enabled = true;
        }
        RightUnHold ();

        OriPos = Player.localPosition;
        OriHandPos = LeftHand.localPosition;

}
```



Video 15: *Holding onto Ladder fall*

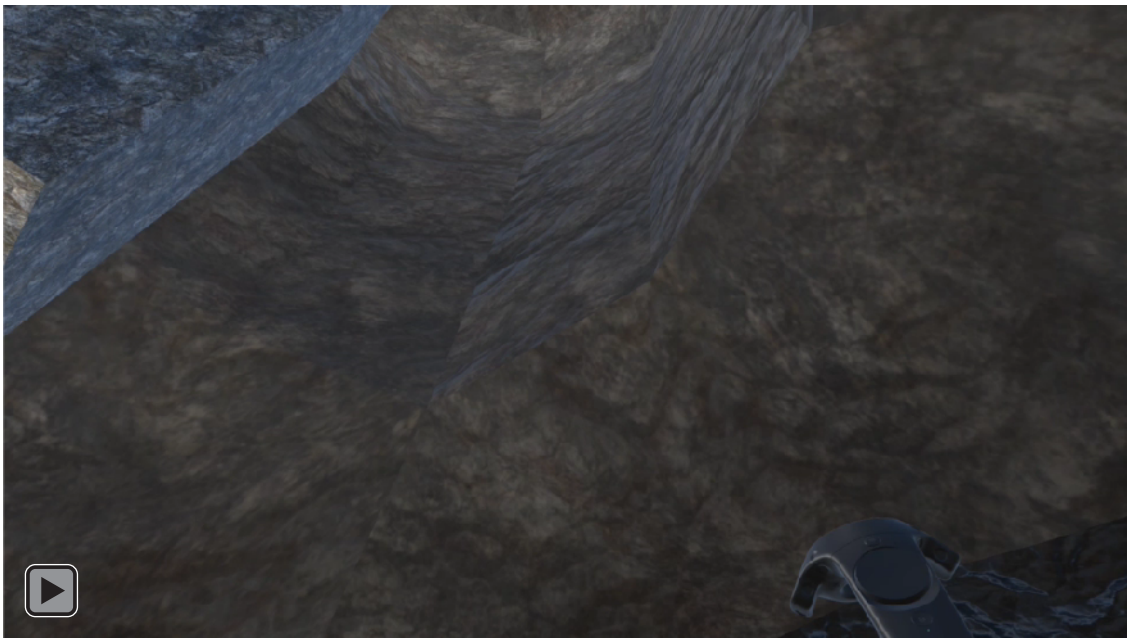
4.5 - Grappling Hook Locomotion

The VR game "*Windlands*"[30] created a locomotion system that allows users to shoot out grappling hooks which can then be used to swing around the virtual environment. I used this type of locomotion to research how to create locomotion that isn't traditionally used.

My initial plan step for this locomotion was to critically analyse how the locomotion worked within Windlands. This involved writing down step by step exactly what was happening when the locomotion activated. From this analysis, I was able to break down what was happening and transfer that into code. From what I gathered, there were essentially only two fundamental actions taking place for this locomotion. The first being a button press to activate the locomotion, second being the player moving towards the location that the grappling hook has attached to.

```
	void Update () {

		if (Calculate == true) {
				Force = (End.position -
Controller.position).normalized * speed;
				RB.AddForce (Force);
		}
	}


	public void Extend(){

		if (Physics.Raycast (Controller.position,
Controller.TransformDirection (Vector3.forward), out hit)) {

			End.parent = null;
			End.position = hit.point;
			Calculate = true;
			Mesh.SetActive (true);

		}

	}
```

[30] *Windlands*, Microsoft Windows (Psytec Games Ltd, 2016).



Video 16: *Grappling Hook Locomotion*

Now that a base version of Windland's locomotion had been created, it was time to try and improve it further. To try and create this type of locomotion even more immersive, I decided to give the user more control over it. To give more control to the user, the user will be able to shoot out the hook, however, they will only be able to move by quickly pulling back. The desired effect is that the user will actually feel as though they are slinging themselves into the air, similar to the arm swing with the jumping.

To create this new code, I needed to get the speed at which the controller was moving backwards. If this speed was over a certain threshold, then force can be added to the user in the direction that they pulled. One issue that arose when testing, was that gravity was greatly affecting the Y value that the user would travel in from the pull force. If the user was to pull downwards, which would move the user up, the force of gravity was too much. To solve this issue, I doubled the amount of force on the Y value only if the Y value was positive.

```
void Update () {

            if (Calculate == true) {
                  curLength = Vector3.Distance (Controller.position,
End.position);

                  Distance = Vector3.Distance (Controller.position,
PrevPos.position);
                  if (Distance >= 0.3f) {
                        Force = (PrevPos.position -
Controller.position).normalized * pullspeed;
                        if(Force.y > 0){
                        ExtraForce = new Vector3 (Force.x, Force.y * 2,
Force.z);
                        }
                        else{
                        ExtraForce = Force;
                        }
                        RB.AddForce (Force + ExtraForce);
                        PrevPos.position = Controller.position;

                  }

                  transform.LookAt (End);
                  scaleZ = (Vector3.Distance (transform.position,
End.position)) * Offset;
                  transform.localScale = new Vector3 (1, 1, scaleZ);

            }
      }

      public void Extend(){

            PrevPos.position = Controller.position;

            if (Physics.Raycast (Controller.position,
Controller.TransformDirection (Vector3.forward), out hit)) {

                  End.parent = null;
                  End.position = hit.point;
                  Calculate = true;
                  Mesh.SetActive (true);


            }

      }
```



Video 17: *Pulling Grappling Hook Locomotion*

4.7 - Combining locomotions together

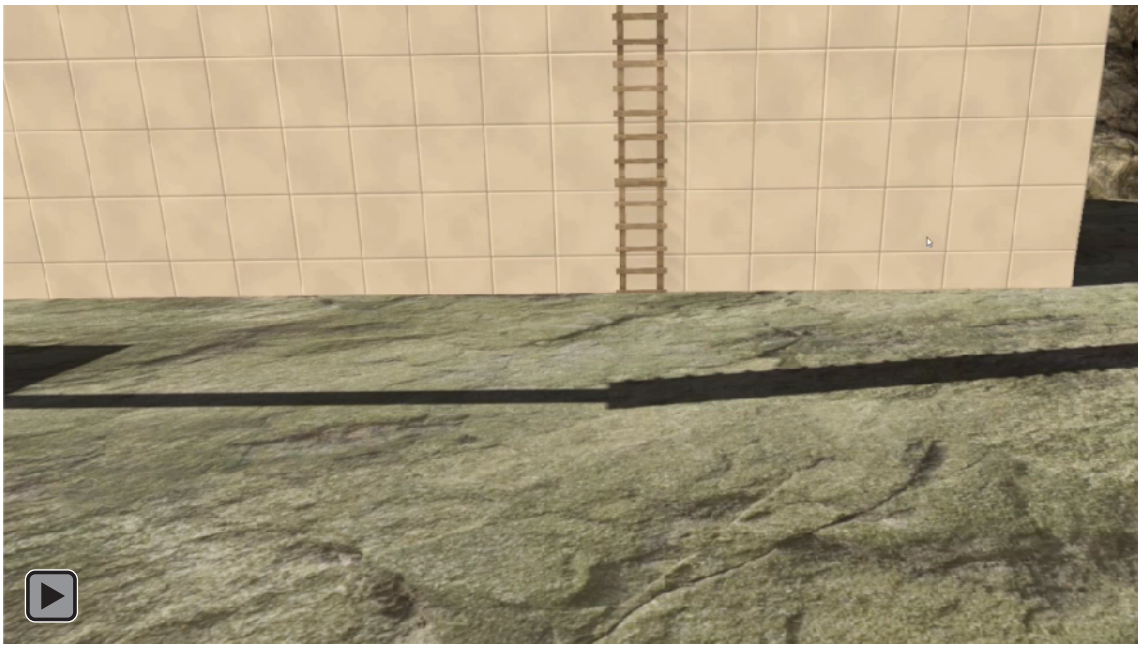
Through my methodology, I had already planned out in my initial plan step for each locomotion what inputs they would use. Because of this initial plan, I was able to avoid any conflicts between any of the different locomotions. This allowed for this part of development to only focus on the transitions. My initial plan step for this, of combining locomotions together, was to decide what needs to be a priority if all locomotions can be used at once. To decide on the order, I used the real world as my reference.

1. Climbing
2. Swimming
3. Walking

Climbing would be the first priority as you can't swim or walk while holding onto a ladder. Swimming came second as having walking as a priority would cause the user not to be able to swim at all if they were touching the ground in water. Once this priority list has been decided on, we need to set bools in the appropriate scripts stating what locomotions are currently being used.

```
if (Climbing == false) {

    if (LocomotionMode == 1) //Hands Free
    {
        if(hipTrackerOn == true)
        {
            HandsFreeMovement();
        }
        else
        {
            TrackpadExercise();
        }
    }
    if(LocomotionMode == 2) //Run On Spot with trackpad
    {
        TrackpadExercise();
    }
    if(LocomotionMode == 3) //only trackpad movement
    {
```

```
            TrackpadMovement();
        }
        transform.Translate(moveDirection);
    }
```



Video 18: *Combined Locomotion*

5 - Discussion

This project has researched many different forms of locomotion within VR gaming. To ensure that the locomotion systems work seamlessly together, action research has been used as part of the methodology. By using action research, I have been able to refine each aspect of the different locomotion forms to create a more effective locomotion system. As coding is a big part of this project, through the multiple cycles of action research, I was able to gain more knowledge of coding and more efficient ways of coding for VR. This has allowed me to go back to previous code for locomotion and improve it, allowing for an easier combination of locomotion forms. One of the major issues I had while creating this project was due to software bugs with plugins and Unity 3d. These bugs caused issues that were out of my control to fix and caused me to waste a lot of time trying to solve, which unfortunately were only able to be solved by creating a new project or changing the unity version if I had updated it.



Video 19: *Gameplay Bug*

The above video shows the player being bounced very far when they have jumped or when the player has fallen from a high place. The bug was an issue with the head being tracked within the camera rig prefab. If the Camera Rig prefab gets moved, as the headset is a child of the Camera Rig, the local positions of the headset should not be affected by any of the Camera Rig movement. However, a bug was causing the local position to be affected by any Camera Rig movement. This means that if the player is falling, this would be picked up as the player running on the spot at a very high speed, which caused the player to bounce away when they land on the ground.

6 - Conclusions/Implications

This project has shown multiple ways in which locomotion in VR gaming be not only combined but also be improved and enhanced. The scripts contained in this exegesis can be used as guidelines for creating locomotion systems and would be helpful to any Unity developers wanting to add certain locomotion into their games. Future implications for this project would be to incorporate the locomotion system I have created into an actual game. Incorporating this system into a game would then require a lot of resources dedicated to playtesting, as many tweaks would need to be made to variables such as speed. The methodology that I have used has shown the importance of the initial planning, as by doing so when combing the locomotion modes together for this project, there were no conflicts between any of the modes. Because of the issues I had with software bugs with Unity, one thing that I would change if I was to start this project again, would be to stay on the same Unity version the whole way through. A lot of wasted time came from changing my code to try to fix an issue which ended up being an issue with unity instead of the code.

7 – Bibliography

Anenberg, Bryan, Harrison Ho, and Karen Yang. *Atlantis - VR Swimming Simulation & Puzzle Game*. Microsoft Windows, 2017.

Bethke, Erik. *Game Development and Production*. Wordware Publishing, 2003.

Black 9. Sony Playstation 2, Xbox. Taldren, n.d.

Boletsis, Costas. "The New Era of Virtual Reality Locomotion: A Systematic Literature Review of Techniques and a Proposed Typology," September 28, 2017.

Budget Cuts. Microsoft Windows. Neat Corporation, 2016.

Candy, Linda. "Practice Based Research: A Guide." University of Technology Sydney, November 2006.

Fernandes, Ajoy, and Steven Feiner. "Combating VR Sickness through Subtle Dynamic Field-Of-View Modification," 2016.

Final Fantasy X. Sony Playstation 2. Sqaure Enix, 2001.

Hover Junkers. Microsoft Windows. Stress Level Zero, 2016.

Jerald, Jason. *The VR Book: Human Centered Design for Virtual Reality*. Association for Computing Machinery and Morgan & Claypool, 2015.

Lindenhof, Brian. *Climbey*. Microsoft Windows. ShadowBrain Games, 2016.

Muratovski, Gjoko. *Research for Designers : A Guide to Methods and Practice*. SAGE publications Ltd, 2015.

Raw Data. Microsoft Windows. Survios, 2017.

Salen, Katie, and Eric Zimmerman. *Rules of Play - Game Design Fundamentals*. MIT Press, 2003.

Schon, Donald. *The Reflective Practitioner: How Professionals Think In Action*. Basic Books, 1984.

Slater, Mel, and Sylvia Wilbur. "A Framework for Immersive Virtual Environments (FIVE): Speculation on the Role of Presence in Virtual Environments.," 1997.

The Elder Scrolls V: Skyrim VR. Microsoft Windows,. Bethesda Game Studios, 2018.

The Lab. Microsoft Windows. Valve, 2016.

Windlands. Microsoft Windows. Psytec Games Ltd, 2016.