Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

An Investigation into Locomotion and Interaction Mechanics in a Multiplayer Virtual Environment

Brian Whelan
15315707

April 22, 2019

Supervisor: Rachel McDonnell

A Final Year Project submitted in partial fulfilment
of the requirements for the degree of
B.A (Mod.) Integrated Computer Science

Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at `http://tcd-ie.libguides.com/plagiarism/ready-steady-write`.

Signed: _____ Date: _____

Abstract

With the rise in adoption of consumer-grade Virtual Reality systems, more users than ever have access to virtual environments. Current experiences generally rely on room-scale or seated tracking experiences, augmented by additional methods of locomotion such as teleportation or artificial locomotion. As has been the case with both entertainment and productivity applications, a desire for social and collaborative interaction has arisen as of late among Virtual Reality users, with which a host of potential problems arise in regards not only to local and remote perception, but also around interaction and consistency of interaction in a multiplayer environment.

This paper describes the investigation and implementation of three potential methods of locomotion and examines the sense of perception of self and others, both in the local context and in the context of remote clients. The paper also describes two competing methods of interaction within a virtual environment and compares both in regard to interactivity, preciseness and bandwidth usage. Participants were placed in pairs of two into a multiplayer virtual environment and asked to solve a series of tasks which made use of both methods of locomotion and methods of interaction.

Acknowledgements

I would like to thank my family, especially my parents, for the continuous love and support which has encouraged me throughout my life.

I would like to thank my supervisor, Rachel McDonnell, for her time, support and advice throughout the project, and Katja Zibrek, for her advice in regards to VR development and locomotion.

Finally, I'd like to thank my friends for the fun, support and friendship which has made the past four years all the more entertaining.

Contents

1 Introduction **1**

1.1 Research Question 1

1.2 Research Goals 1

1.2.1 Virtual Locomotion 1

1.2.2 Network Interaction 1

1.3 Technologies Used 2

1.3.1 Unity 2

1.3.2 Forge Networking Remastered 2

1.3.3 HTC Vive 3

1.3.4 SteamVR Plugin 3

2 State of the Art **4**

2.1 SteamVR Home 4

2.2 VRChat 4

2.3 Galvanic Vestibular Simulation 5

3 Implementation **6**

3.1 Introduction 6

3.2 SteamVR Input 6

3.3 The Player 7

3.3.1 The Player Prefab 7

3.3.2 The VRNetworkController 8

3.3.3 The Player Avatar 10

3.4 Locomotion 11

3.4.1 The ILocomotionState Interface 12

3.4.2 Method 1 - Teleportation: 13

3.4.2.1 Initial Use 13

3.4.2.2 Room-Relative Offset 14

3.4.2.3 Local Experience 16

3.4.2.4 Remote Experience 16

3.4.3 Method 2 - Artificial Locomotion 16

3.4.3.1 Initial Use 18

3.4.3.2 Local Experience 19

3.4.3.3 Remote Experience 19

3.4.4 Method 3 - Third Person Avatar 19

3.4.4.1 Navigating the Scene . . . 19
3.4.4.2 Handling Avatar Animation . . . 21
3.4.4.3 Local Experience . . . 23
3.4.4.4 Remote Experience . . . 23
3.4.5 Network Details . . . 23
3.4.5.1 Compressing Player Packets . . . 24
3.4.5.2 Serialising State . . . 24
3.4.5.3 De-Serialising and Interpolation . . . 25
3.5 Interaction . . . 25
3.5.1 Method 1 - Event-Based Interaction . . . 25
3.5.1.1 The EventInteractionManager . . . 26
3.5.1.2 Defining Interactables . . . 26
3.5.1.2.1 The Grab Command . . . 27
3.5.1.2.2 The Drop Command . . . 27
3.5.1.2.3 The Drop Snap Command . . . 27
3.5.1.3 Player Hands . . . 28
3.5.2 Method 2 - Physics-Based Interaction . . . 29
3.5.2.1 Interactables . . . 30
3.5.2.2 The PhysicsInteractionManager . . . 31
3.5.2.2.1 Client-Side Ownership . . . 32
3.5.2.2.2 Server-Side Filtering . . . 33
3.5.2.2.3 Receiving On Client . . . 35
3.5.2.3 Physics Hand . . . 35
3.5.2.4 Collisions . . . 36
3.5.3 Combining with Locomotion . . . 36

4 Evaluation **38**
4.1 Scenario . . . 38
4.1.1 Navigation . . . 38
4.1.2 Interaction . . . 40
4.2 Survey . . . 43
4.3 Results . . . 43
4.3.1 Participant Information . . . 43
4.3.2 Locomotion . . . 44
4.3.2.1 Ease of Travel . . . 47
4.3.2.2 Comprehension of Location . . . 48
4.3.2.3 Experience of Discomfort . . . 48
4.3.2.4 Remote View of Player Avatar . . . 49
4.3.3 Interaction . . . 50

4.3.3.1 Grabbing and Releasing 51
4.3.3.2 Reactions to Environment 51
4.3.3.3 Precision Placement 51
4.3.3.4 Participant Interaction 52
4.3.3.5 Network Results 52

5 Conclusion **55**

A 1 Appendix **57**
A 1.1 Survey Link 57
A 1.2 Participants 58
A 1.3 Locomotion 59
A 1.4 Interaction 61

List of Figures

1.1 The HTC Vive HMD and two Vive Wand controllers 3

3.1 Example of the SteamVR input binding UI 7
3.2 The hierarchy of the stock SteamVR player prefab 7
3.3 The hierarchy of the modified network-ready VR player prefab 8
3.4 Flow of a single instance of the Player Prefab. The green region indicates code executing locally, while yellow represents shared logic and blue represents remote-only logic 9
3.5 View of the Player Prefab as observed on a remote client 10
3.6 View of the calibration scene, with tutorial text and reference pose 11
3.7 Participant aiming towards desired location 13
3.8 Participant (blue) standing two meters to the right of room centre (pink) aims to teleport to destination (green) 15
3.9 Without offset, teleport places participant two meters right of destination 15
3.10 With offset, room is placed two meters left of destination. Participant is placed at expected position 15
3.11 Room centre (pink) with collider (green) and participant (blue) 18
3.12 Without offset, participant clips wall (black) as collider moves forward . 18
3.13 With offset, collider contacts wall and prevents participant clipping . . . 18
3.14 A user (right) aiming towards their desired location, with full-body avatar in motion 20
3.15 Blend tree state, with animation links and velocity parameters 22
3.16 Blending plot and trigger values (velx, vely, animation speed) 22
3.17 Flow of a command (Grab) being invoked on an Interactable 28
3.18 A user throwing an item 30

4.1 View from the overlook, with participant A holding the map, and participant B located at the beginning of the maze 39
4.2 View from the maze of participant A holding the map, from participant B located at the beginning of the maze 40
4.3 Participant A prepares to receive the silver key from participant B 41
4.4 Participant A places the silver key in the correct snap point 42
4.5 Percentage Responses for Teleporation 44
4.6 Percentage Responses for Third-Person Locomotion 45
4.7 Percentage Responses for Artificial Locomotion 46

4.8 Weighted scores for Locomotion responses 47
4.9 Weighted scores for Interaction responses 50

List of Tables

3.1 Values of CharacterController parameters 16
3.2 Remote avatar state fields and corresponding value sources 20
3.3 NetworkObject contract fields, corresponding types, and size of types . 23

4.1 Survey response values and corresponding ordinal value 43
4.2 Packet and Packet Payload sizes. 52
4.3 Packet and Packet Payload sizes. 53
4.4 Packet and Packet Payload sizes. 53

Terminology

GameObject:	An object existing in the current scene
NetworkObject:	A GameObject with a degree of replication over the network
VR:	Virtual Reality
Interactable:	An item that exists in the scene, which players can interact with
HMD:	Head mounted display, the main visor in a VR setup

1 Introduction

1.1 Research Question

"To examine methods of locomotion and interaction mechanics in a multiplayer virtual environment, to determine the advantages of each individual method of locomotion regarding local and remote perception, comfort and ease of use, and to examine methods of interaction in regard to interactivity, preciseness, and consistency in a networked environment"

1.2 Research Goals

The two primary goals of this research are:

1.2.1 Virtual Locomotion

The objective of the research with regard to locomotion is to examine a number of existing methods of locomotion in a single-user virtual environment and to explore how they translate to a virtual environment occupied by more than one user. Of particular interest is the perception of the world to the local user, in contrast to the perception of this user from the view of a remote client. The level of comfort experienced by the local user in each method of interaction was also investigated, in addition to the process of simulating a virtual user in a networked environment and the effect which these methods of locomotion had on maintaining a consistent state across the network.

1.2.2 Network Interaction

Involving multiple players in a virtual environment introduces a broad range of decisions to be made regarding the experience of the user in the virtual environment and introduces a range of new challenges in regard to the network. Of particular interest concerning interaction mechanics was ensuring a consistent environmental state for all players across the network. Ensuring a consistent environmental state for all players across the network was vital, while also ensuring that user interactions perform as expected for a user in their local environment. Factors such as bandwidth

usage, race conditions and consideration for latency were introduced to combat these issues.

1.3 Technologies Used

1.3.1 Unity

Unity is a cross-platform game engine which includes native support for Virtual Reality applications. Environments in Unity are divided into scenes, which can be considered "levels" in a game. Objects in the scene are represented as GameObjects, and each GameObject can have multiple components attached, which define operational logic for each particular GameObject.

1.3.2 Forge Networking Remastered

Forge Networking Remastered is an open-source networking framework built to support Unity. It provides high-level wrappers around the .NET Socket class and serialisation/de-serialisation methods for Unity-specific types. NetworkObjects represent a Unity GameObject in a networked scene and are referenced via a shared network ID across all clients. A NetworkObject is described by a Network Contract, which defines the fields which the NetworkObject should transmit across clients. For more fine-tuned control, Remote Procedure Calls can be invoked over the network with a byte array containing payload data.

As part of the research project, a bug-fix[1] regarding culture information invalidating JSON output and an improvement[2] to the way in which Network Contracts are generated were committed back to the project.

[1] https://github.com/BeardedManStudios/ForgeNetworkingRemastered/pull/198
[2] https://github.com/BeardedManStudios/ForgeNetworkingRemastered/pull/216

1.3.3 HTC Vive

The project was developed with the HTC Vive and two HTC Vive devices were used during the course of this research. Combined with SteamVR, the HTC Vive supports room-scale tracking, which can track any number of devices, such as the head mounted display (HMD) and two Vive Wand controllers used during the project. Room-scale tracking was an important factor in regard to locomotion and interaction in this project. Combined with the Vive Base Stations, the HTC Vive provides sub-millimetre precision tracking (1), which enabled the precise locomotion and interaction systems implemented in this project.

Figure 1.1: The HTC Vive HMD and two Vive Wand controllers

1.3.4 SteamVR Plugin

The SteamVR plugin is an extension for the native Virtual Reality support provided in Unity. It provides classes and types for interacting with the SteamVR Input system and also exposes Unity components for replicating SteamVR pose data from a tracked object such as a controller on to a Unity GameObject.

2 State of the Art

2.1 SteamVR Home

By default, SteamVR launches into an application called SteamVR Home. This provides a default environment for the user to explore in Virtual Reality before launching into a Virtual Reality enabled application via the in-environment dashboard. It also allows users to change between multiple environments or download user-made environments from Steam. It supports teleportation as its sole method of locomotion. It also allows the user to interact with items in the scene, which simulate physics and interact with other items and the environment.

2.2 VRChat

VRChat is a free-to-play multiplayer Virtual Reality application which supports the HTC Vive, Oculus Rift and Windows Mixed Reality devices. It allows for multiple users to occupy a shared virtual environment and to interact with each other. Players are represented by full-body avatars, which track either using full-body tracking, or estimating feet position. Two locomotion methods are supported by VRChat. The first, "Holoportation", is similar to the third-person avatar method discussed in this paper. Described as the "comfort" option, it provides a third-person avatar which travels in place of the user, with teleportation to move the user to the location of the third-person avatar. It also allows for Artificial Locomotion. It provides physics-based interactions in a networked context.

2.3 Galvanic Vestibular Simulation

Galvanic Vestibular Simulation (GVS) is a way of stimulating the vestibular system (inner ear) in order to invoke specific responses in the vestibular system, in this instance synchronised with the motion experienced by the user. As a result, it is possible to significantly reduce motion sickness and related symptoms such as nausea, by tricking the vestibular system into believing body motion exists where it does not. GVS is not currently available in any consumer-grade Virtual Reality devices, however small and lightweight wearable devices are under research (2).

3 Implementation

This chapter describes the implementation of the system created in order to investigate the proposed research topic. A copy of the source code is available on the included CD.

3.1 Introduction

The simulation consists of a virtual environment built to support multiple users. The virtual environment provides tasks which examine the use of locomotion and interaction in the environment. Participants are represented by a virtual avatar, which allows the user to explore the virtual environment using three varying methods of locomotion. It also allows users to interact with the environment through two different implementations of interaction mechanics.

3.2 SteamVR Input

In May 2018 (3), Valve announced a replacement for the traditional button-based input system provided by SteamVR. The new system, titled SteamVR Input, allows developers to define actions and have their application respond to the invocation of these actions, rather than relying on polling for hard-coded key values. A configuration is generated from the defined actions, allowing users to build binding configurations to map inputs on their chosen controller to actions defined by developers. This change allows developers to support a diverse range of controllers with no change to input logic.

Figure 3.1: Example of the SteamVR input binding UI

3.3 The Player

3.3.1 The Player Prefab

The SteamVR Unity plugin provides a simple prefab object for Unity, configured in a hierarchical manner as shown in figure 3.1. The root GameObject in the hierarchy represents the physical room centre, while the three children each individually represent a single tracked object, either the head mounted display, the left-hand controller, or the right-hand controller. Pose data for these three tracked objects is provided by SteamVR in local world-space coordinates, relative to the centre of the real-world room the user is situated in.

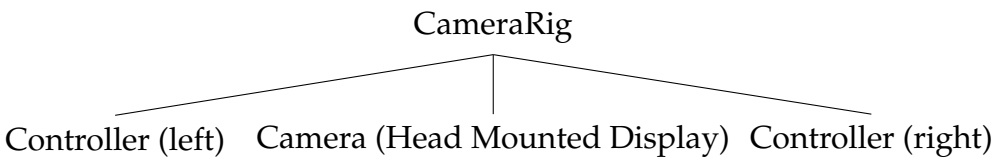


Figure 3.2: The hierarchy of the stock SteamVR player prefab

This configuration is not ideal from a multiplayer standpoint, as there are often cases

in which remote users need to be represented differently to local users. In order to better manage the differences between a local representation of a user and the remote representation of this user, a new prefab was constructed with the following hierarchy:

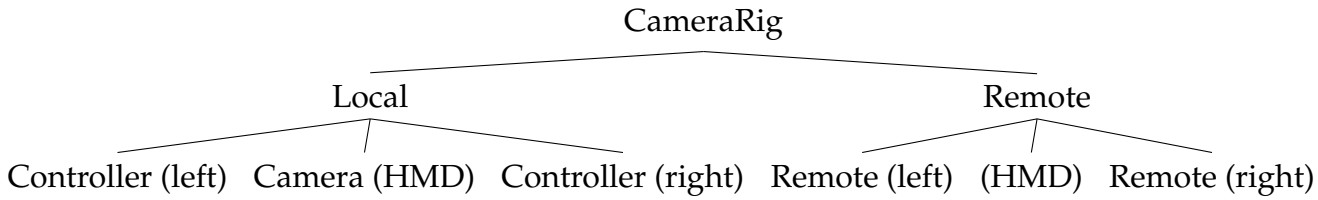


Figure 3.3: The hierarchy of the modified network-ready VR player prefab

The `CameraRig` in the hierarchy contains a component, `RemoteLocal`, which determines whether the entire hierarchy is under local-authority or remote-authority. Depending on this value, either the local or remote child will be enabled for the player instance. We then supply a component, `VRNetworkController` attached to the active child with the `NetworkObject` for the player. By segmenting local and remote aspects of the player avatar in this fashion, different perspectives of the player for both remote and local instances can be provided. It also allows for the division of logic based on whether it should run on local instances only, on remote instances only, or on both instances.

We define the `Local` child GameObject as representation of a local avatar, with the position of this GameObject mapping to the room centre. The three children represent both the head mounted display and the two tracked controllers for each hand, with pose information relative to the room centre.

Similarly, the `Remote` GameObject is defined as representing a user on a remote client. The root of this sub-hierarchy is kept as a method of easily disabling the entire sub-hierarchy when not needed and is not modified over the course of locomotion. The three children of this object receive positional and rotational data in world-space over the network, accurately representing the true world-space location of the user.

3.3.2 The VRNetworkController

Both the Local and Remote children in the hierarchy each have an attached component, `VRNetworkController`, which is used to handle player logic and store and retrieve state information over the network via the `NetworkObject`.

On both local and remote clients, we initialise the `VRNetworkController` through the `RemoteLocal` component and provide it with the `NetworkObject` representing the

player.

On the local client, the `VRNetworkController` initialises any implementations of `ILocomotionState`, which is an interface used to define a Locomotion method. It also initialises the Hand component attached to each controller GameObject for the current method of interaction.

Each frame, the local `VRNetworkController` applies the active `ILocomotionState` implementation to the current state of the player. The `ILocomotionState` implementation produces both a new local state (re-positioning the local GameObject) and a new remote state, which is injected into the `NetworkObject` and serialised over the network.

When a new state is received on the remote `VRNetworkController`, it is stored in the `NetworkObject`. The current state is interpolated towards the new state with an interpolant value of 0.15f, reducing jitter and giving the appearance of smooth movement on remote clients

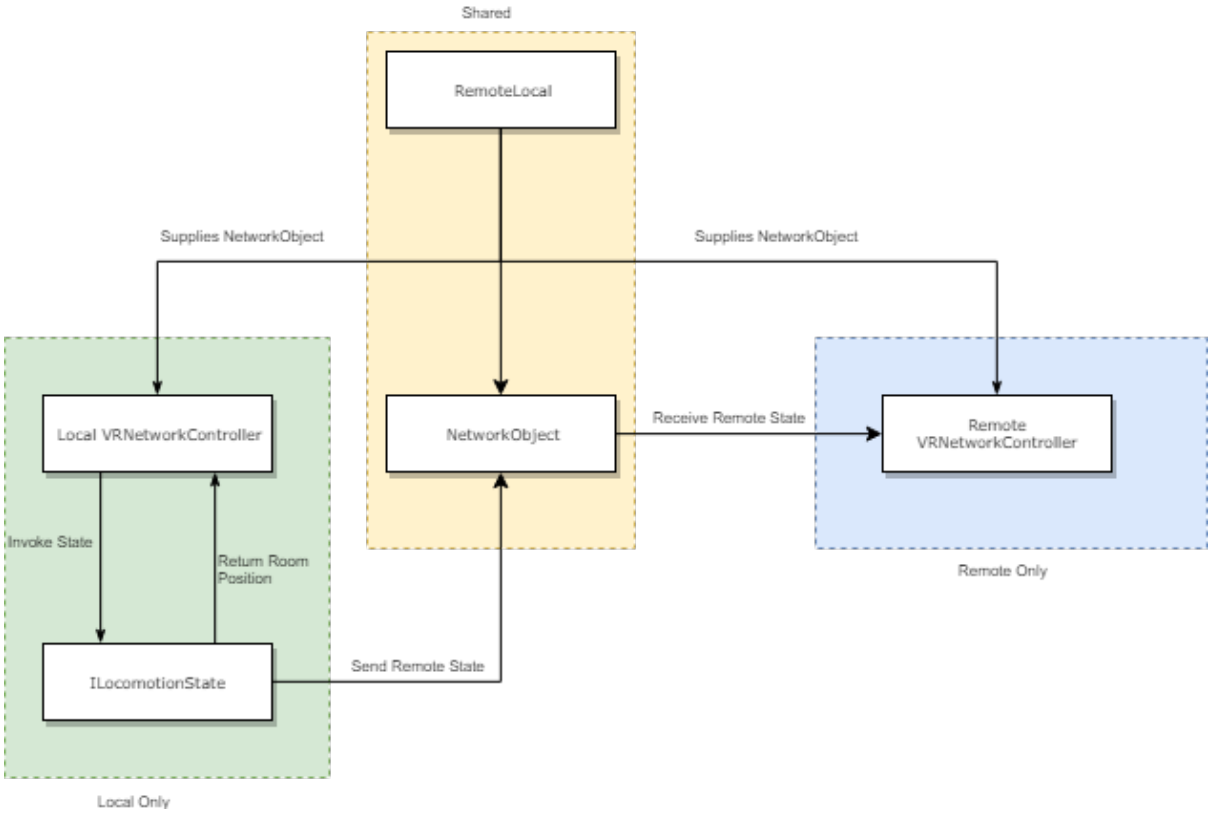


Figure 3.4: Flow of a single instance of the Player Prefab. The green region indicates code executing locally, while yellow represents shared logic and blue represents remote-only logic

3.3.3 The Player Avatar

Both sub-hierarchies also each contain a fourth child, the avatar body. Users both locally and remotely are represented via an upper-body avatar with an Inverse Kinematics solver applied to it, to predict arm, shoulder and head positions from tracked data. The body avatar used is a modified version of the "Space Robot Kyle" (4) model made available with Unity 4.

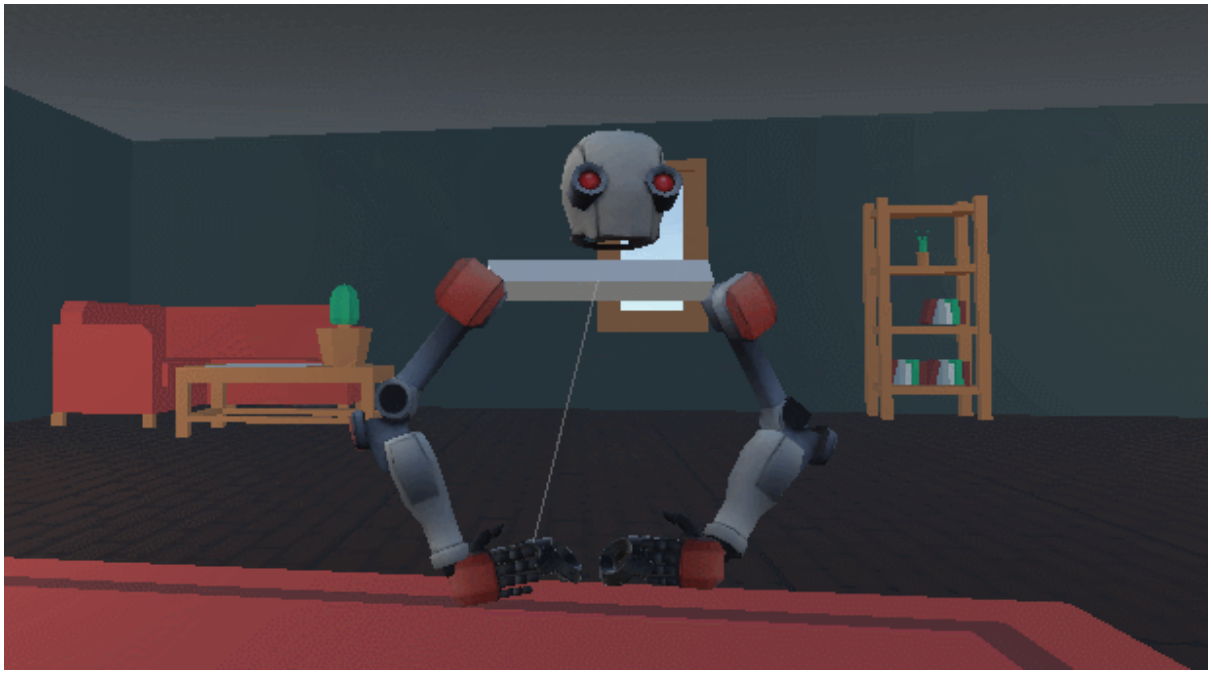


Figure 3.5: View of the Player Prefab as observed on a remote client

Initially, Inverse Kinematics was implemented using Unity's built-in Inverse Kinematics solver. However, it was found that the built-in solver only solves poses for arms and neck, with no consideration for relative shoulder or torso poses. This ultimately proved unusable as in certain cases, such as a user bending down or over-extending their arms, the lack of shoulder motion led to inaccurate solutions for the avatar.

To address this issue, a secondary solution was integrated as a replacement for the Unity IK solver. "VRArmIK", developed as part of work detailed in (5). The solution was modified to remove the use of the singleton pattern, in order to allow for multiple instances of the solver to be active in the current scene. This solution provided a number of additional improvements regarding shoulder and arm tracking, as follows:

- Shoulder rotation is calculated as the sum of the direction vectors from the position of the HMD to each left and right-hand controller.
- Shoulder roll is calculated based on the current height of the HMD relative to the standing height of the participant.
- Arm-Length is scaled based on the distance between both hands.

In order to calculate both the height and arm length of the participant, a calibration scene was inserted at the beginning of the application. The participant is asked to match a reference pose (T-Pose) shown on a full-body representation of their avatar and trigger the locomotion input action on both controllers. The height of the head mounted display was taken as participant height and the distance between both controllers taken as the width for arm length calculations.

Figure 3.6: View of the calibration scene, with tutorial text and reference pose

3.4 Locomotion

Three locomotion methods with varying control methods were implemented as part of the project. In order to allow for multiple movement methods to be applied by the controller, an interface was created to define a standardised way of invoking locomotion.

3.4.1 The ILocomotionState Interface

Each locomotion method definition implements an interface, `ILocomotionState`, which defines methods to enable the `VRNetworkController` to call upon it. By implementing each locomotion method on top of this interface, movement mechanics are standardised, enabling the hot-swapping of methods at runtime based on user preference. The following methods are defined by the `ILocomotionState` interface and implemented by all locomotion methods:

- `Apply(VRPlayerNetworkObject, SteamVR_Input_Sources)` - called each frame, the Apply method is used to produce a new state for the player. The first parameter specifies the `NetworkObject` for which the remote state should be placed in and the second parameter is the controller for which pose data should be retrieved if required.
- `FixedApply(VRPlayerNetworkObject, SteamVR_Input_Sources)` - called at a fixed timestep in line with the physics simulation, independently of the current real-time framerate. SteamVR demands a fixed timestep of 90Hz. This method is used for time-dependent operations, such as applying physics forces, playing audio at a fixed rate and applying motion based on input sampled in Apply().
- `CleanUp()` - called when the `ILocomotionState` is no longer the active locomotion state for the parent `VRNetworkController`. This method is used to reset any state and hide any visual side-effects of a locomotion state.

Each player avatar contains an attached controller, `VRNetworkController`, from which the `ILocomotionState` is supplied with required dependencies via its constructor. Each frame, the `VRNetworkController` invokes the Apply method of the current `ILocomotionState` to the current user state. The local hierarchy is directly affected by the application of the current `ILocomotionState`, generally through the re-positioning of the room centre. The `ILocomotionState` also provides the new remote avatar state to the `NetworkObject`, which handles serialisation and messaging over the network. The local and remote state is separated in this fashion, as the local avatar state and the remote avatar state are not always identical, the local user may perceive themselves differently to how they perceive a remote user in the same state.

3.4.2 Method 1 - Teleportation:

The first method of locomotion explored in this project is that of teleportation. Participants are free to roam via room-scale tracking. When they find that the boundaries of the physical room in which they inhabit restricts their ability to explore the virtual environment, they can make use of teleportation in order to re-position the room centre in the virtual environment.

3.4.2.1 Initial Use

Teleportation was implemented via the RangedTeleport class, which implements the `ILocomotionState` interface. In the Apply stage of RangedTeleport, while the Teleport SteamVR Input Action state is true, a ray-cast is fired with origin at the world-space position of the given controller, in the forward direction of the controller. The ray-cast has a limit of 10 units (1 unit = 1 meter) and is visible to users as a white laser scaled to reach the hit position of the ray-cast. The laser is created by placing a GameObject with attached cube mesh at the mid-point between the controller position and the hit point and performing a LookAt to rotate the cube to face the correct direction. The box is then scaled to the distance between the controller and the hit-point, on the Z-axis.

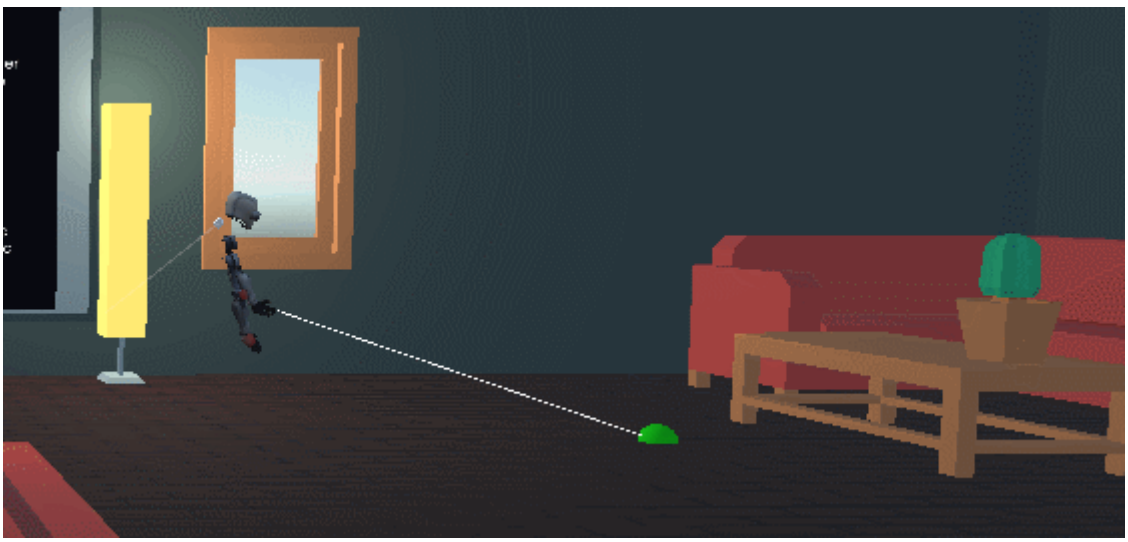


Figure 3.7: Participant aiming towards desired location

When the ray-cast collides with a surface, it first samples the hit point against the `NavMesh` defining the scene, to ensure the area is defined as walkable. The `NavMesh` is a Unity system in which surfaces can be defined as walkable or non-walkable. A resulting mesh is baked which defines which surfaces can be walked upon. This is

used traditionally for the A-star path-finding implementation included in Unity, but in this case is also used to ensure the aimed position is a valid surface.

Next, it ensures that the Normal of the surface at the hit point faces upwards in the world. This ensures that the user does not teleport to a point marked as walkable which could cause clipping (such as the vertical components of a staircase).

A targeting sphere exists whose position is interpolated towards the hit point each frame, ensuring smooth motion and giving a feeling of elasticity to the pointer. When the user targets a valid position, the sphere turns green to indicate the position is valid. If the target position is not valid, the sphere will change to a red colour.

Upon the state of the teleport input action changing to false, indicating the participant has released the action with a valid target point, the participant will be re-positioned to the hit point. A positive audio cue is also broadcast to the user to indicate success. If the targeting position is invalid, a negative audio cue is broadcast to the user and teleportation does not occur.

3.4.2.2 Room-Relative Offset

On a local client, the position of the headset and two controllers are retrieved directly from SteamVR and any attempt at manually overriding the SteamVR data is ignored. In order to correctly relocate the user in the virtual environment, the room position must instead be translated. Translating the room centre directly to the position the user aims at has the negative side-effect of placing the user in a different location to that of which they were aiming, unless they are standing in the very centre of the room. To address this, an offset is calculated between the head mounted display position and the room position before teleportation. On teleport, the room is relocated to the targeted position and the offset is then subtracted from this position to place the head mounted display and both controllers in the expected position.

```
// Calculate offset vector
Vector3 offset = roomCentre.position - HMD.position;
// Map offset vector to 2D plane
offset.y = 0;
// Apply offset to target position
roomCentre.position = hitPoint + offset;
```

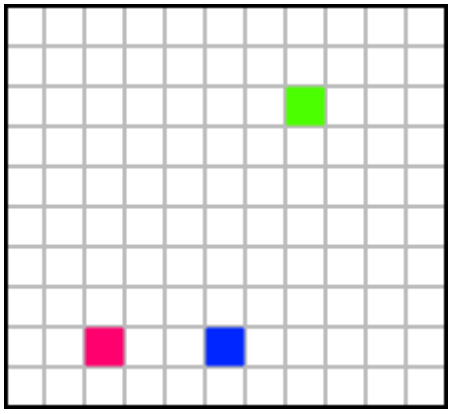


Figure 3.8: Participant (blue) standing two meters to the right of room centre (pink) aims to teleport to destination (green)

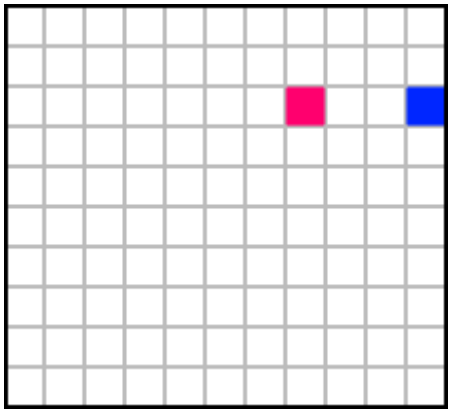


Figure 3.9: Without offset, teleport places participant two meters right of destination

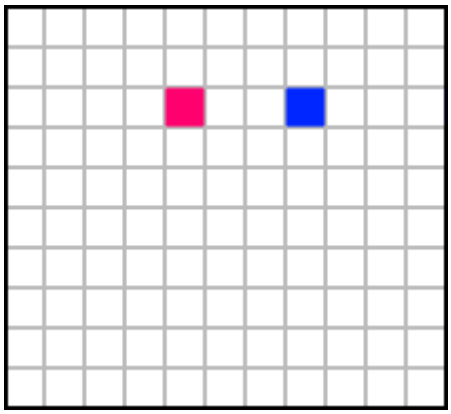


Figure 3.10: With offset, room is placed two meters left of destination. Participant is placed at expected position

3.4.2.3 Local Experience

Locally, on release of the teleport action, the user view will display an immediate "blink" to the targeted position. No additional screen fade-in/out or effect is applied. Any items users are carrying will also remain present in the hand of the user post-teleportation.

3.4.2.4 Remote Experience

On remote clients, the participant observing the remote participant performing a teleport does not see the laser used for aiming. Instead, on release, they experience the user travel at "warp-speed" to the new position. This allows the participant observing the teleportation to understand the change in position of the teleporting participant and occurs due to interpolation applied to remote instances of users to ensure the smooth appearance of motion.

3.4.3 Method 2 - Artificial Locomotion

The second method of locomotion implemented was Artificial Locomotion. Artificial Locomotion is the process of moving the virtual user using a traditional method of input, such as through joystick axes on a gamepad controller, rather than through spatial tracking. In this simulation, we implemented a method of Artificial Locomotion, which made use of a single input action and the pose of the controller on which the action was triggered, in order to construct a direction vector in which to artificially relocate the user.

Artificial Locomotion is implemented using Unity's built-in `CharacterController` component, which defines a controller which does not use rigidbody physics, but does react to collisions using a de-penetration system. A number of properties are defined for the `CharacterController`, which are used by the `CharacterController` during the Move step. In this implementation, the following values were used:

Slope Limit	45
Step Offset	0.3
Skin Width	0.08
Radius	0.1
Height	variable

Table 3.1: Values of CharacterController parameters

Each frame, the ArtificialLocomotionState initialises a direction vector of type Vector3 to zero. If the Movement input action is set to a state indicating it is pressed, the vector is assigned the current forward direction vector of the controller pose. The forward direction vector is normalised and held until the next FixedUpdate call, a method which runs repeatedly at a fixed interval. In this FixedUpdate call, the direction vector is passed to the SimpleMove method defined by `CharacterController`, which simulates a single step of motion by the provided vector with respect to colliders and gravity in the scene. While input capturing is performed each frame to ensure no lost inputs, the application of the result of the input is decoupled from the render step into FixedUpdate, which runs on a consistent fixed rate, to ensure a constant, frame-rate independent movement speed. A similar issue to that of the offset during teleportation was found in the implementation of Artificial Locomotion. As the tracking data is received from the hardware, the room centre must instead be moved in order to move the player view forward. The `CharacterController` component is bound to the root of the local hierarchy representing the room centre. This results in the collider possessed by the `CharacterController` being located at the position of the room centre. The result of the Move step performed by the `CharacterController` is applied to the GameObject representing the room centre, which in turn translates the player position. This introduces an issue whereby the tracked objects, including the player view, could clip through walls or other surfaces as a result of the root locomotion.

To resolve this issue, each frame the ArtificialLocomotionState adjusts the controller height to be equal to the current HMD height to the floor. This is calculated by subtracting the current height of the HMD from the current height of the `CharacterController` and then further subtracting half the current height of the controller to find the floor height. This height scaling occurs to give an accurate collider size for the current player.

Following this scaling, the `CharacterController` is re-centred from the root to be positioned below the HMD. This collider offset ensures that, even though the root of the room is being translated, the collider is located at the view position and thus the player receives the collision they would expect based on their location.

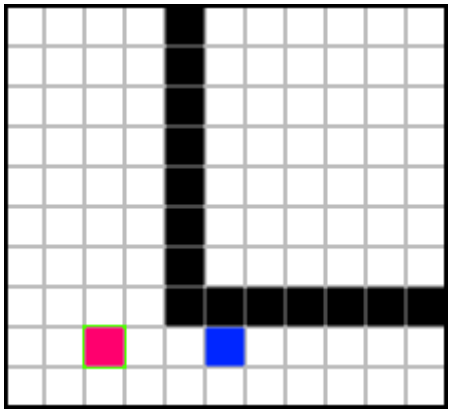


Figure 3.11: Room centre (pink) with collider (green) and participant (blue)

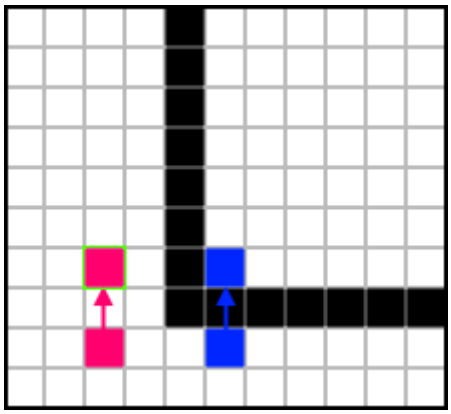


Figure 3.12: Without offset, participant clips wall (black) as collider moves forward

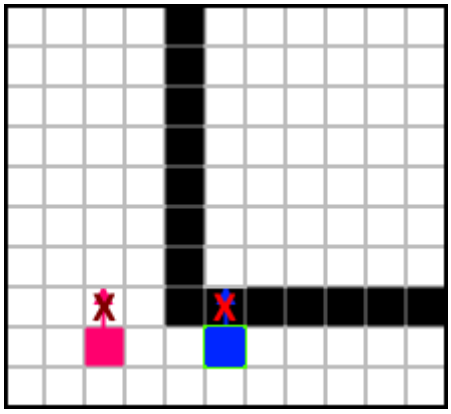


Figure 3.13: With offset, collider contacts wall and prevents participant clipping

3.4.3.1 Initial Use

To initiate artificial locomotion, the participant triggers the locomotion action on either controller. The participant then points the controller for which the action was triggered in the direction they want to travel. The `CharacterController` defines a Move method

which takes an input vector and simulates movement in this direction with respect to colliders in the scene.

3.4.3.2 Local Experience

Locally, the user experiences an artificial motion in the direction in which they point the controller. An audio cue is used to give the illusion of footstep sound effects to assist the user in understanding the locomotion they are experiencing.

3.4.3.3 Remote Experience

Remotely, an observing user witnesses the same translation of position and rotation as experienced by the local user.

3.4.4 Method 3 - Third Person Avatar

The final method of locomotion blends aspects of both previous methods of locomotion, and is inspired by a method of locomotion proposed by the developers of VRChat (6). In the third-person avatar locomotion method, the participant can create a third-person "hologram" representation of themselves which represents the player on remote clients and can navigate the world. When ready, the participant can then resume control, teleporting to the location of the third-person hologram.

When the participant triggers the movement input action, an identical laser-pointer to that in the teleportation locomotion method is made visible. The pointer acts in the same fashion as the teleportation pointer; however the hit point is used instead as the destination for a Unity `NavMeshAgent`.

3.4.4.1 Navigating the Scene

The `NavMeshAgent` is a Unity component which exposes functionality to navigate the current scene using the baked NavMesh describing the scene. A GameObject is defined underneath the local hierarchy of the Player prefab, which contains a `NavMeshAgent` component, a `CapsuleCollider`, a `Rigidbody` and additional Animator components to handle animation of the full-body avatar.

The `NavMeshAgent` supplies pose data back to the GameObject it is attached to. The `NavMeshAgent` is warped to the position of the player on invoke and set to an active state while the participant is aiming. It then navigates towards the hit point, using the

`NavMesh` to avoid obstacles in the scene. While this GameObject is in an active state, the `NavTeleportState` returns the following as the new remote avatar state:

Pose Variable	**Pose Data Source**
HMD Position	Full-Body Avatar Head Position
HMD Rotation	HMD Rotation
Left Controller Position	HMD-Relative Left Controller Position
Left Controller Rotation	Left Controller Rotation
Right Controller Position	HMD-Relative Right Controller Position
Right Controller Rotation	Left Controller Rotation

Table 3.2: Remote avatar state fields and corresponding value sources

When the participant releases the input action, they are shifted to the position at which the `NavMeshAgent` was located. From then on, the `NavTeleportState` will return SteamVR positional pose data as the player state.

One issue arising from this method of locomotion is that certain actions, such as the participant bending over or crouching, would not be properly reflected on the remote client, as they would be seen in-locomotion while in this pose. When returning from an in-locomotion state to the real tracked state, the two states vary. This would appear as a jarring transition for remote observers. To counter this, the local GameObject containing the full-body avatar is scaled relative to the current height of the local HMD and all pose data is sent relative to the head-position of the full-body avatar. This results in the proper blending of the in-locomotion state to the resting tracked state on remote clients.

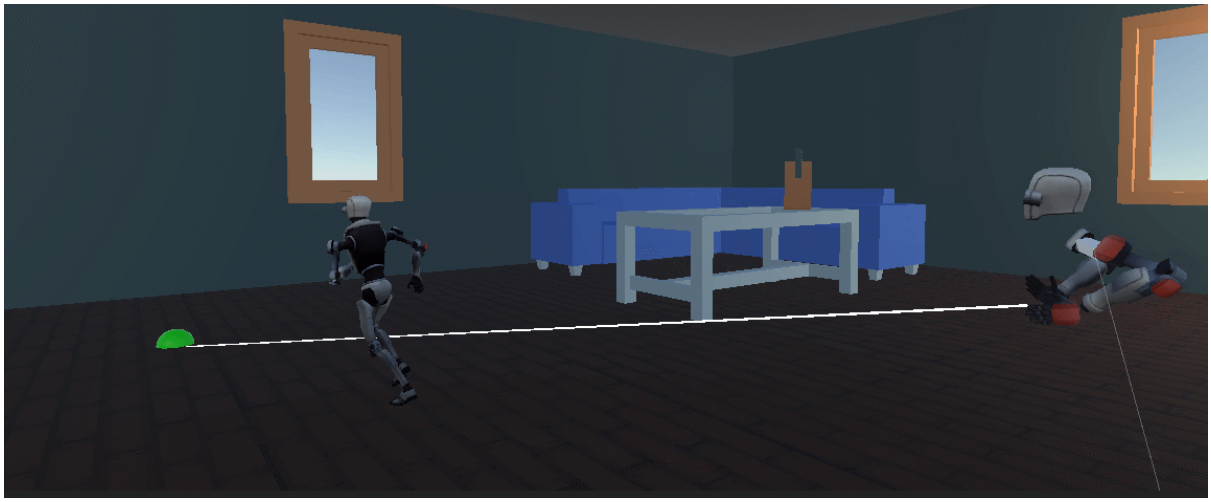


Figure 3.14: A user (right) aiming towards their desired location, with full-body avatar in motion

3.4.4.2 Handling Avatar Animation

In order to animate the full-body avatar, a blend tree is used. The blend tree is used to blend between multiple similar motions, for example a walking and running animation, based on given parameters provided to the blend tree, for example character velocity. The blend tree is used for the full-body avatar in order to blend between different run animations in order to ensure fluid and realistic motion. The blend tree describes seven different animation states, one for idling and six directional animations. We provide the blend tree with both the x and y components of the velocity of the `NavMeshAgent` as blend parameters and describe the blending of these animations by plotting each animation on a 2D plane whose axes are described by the velocity parameters. This can be seen in Figure 3.16.

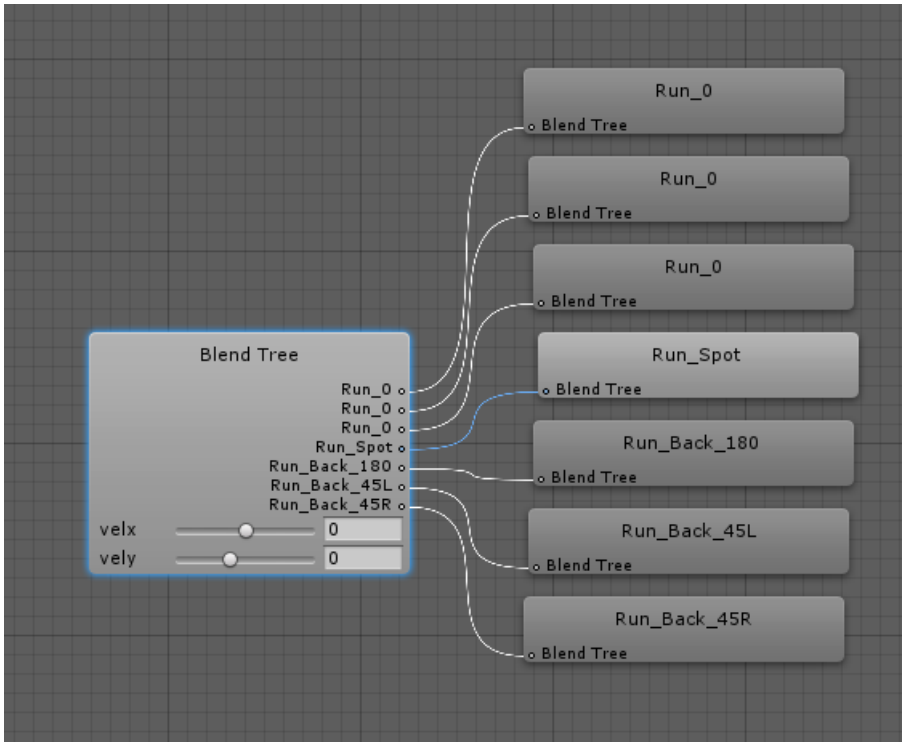


Figure 3.15: Blend tree state, with animation links and velocity parameters

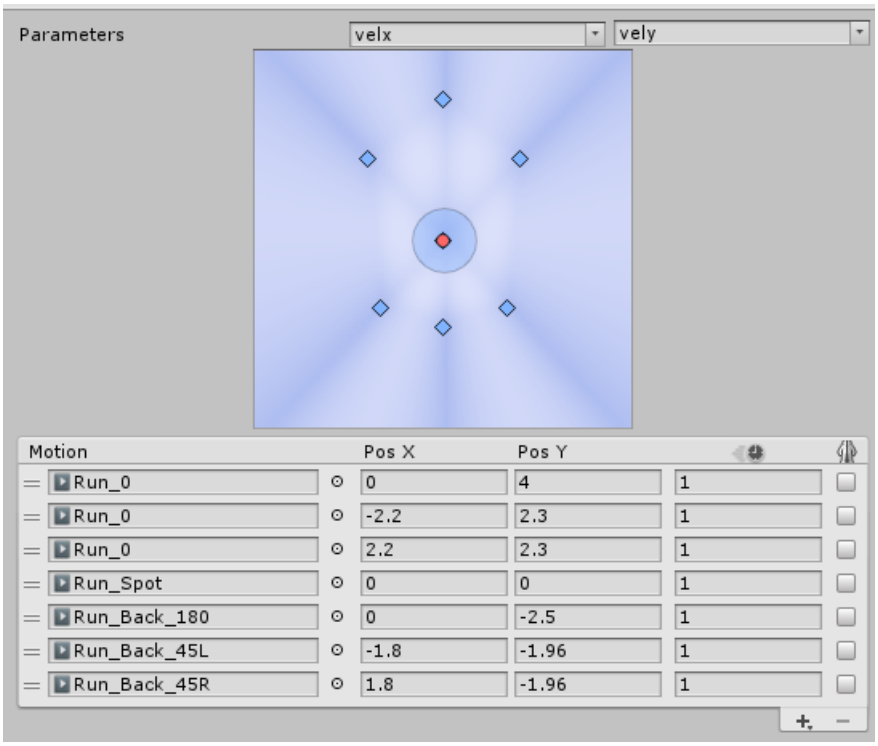


Figure 3.16: Blending plot and trigger values (velx, vely, animation speed)

3.4.4.3 Local Experience

Locally, the participant witnesses a third-person representation of themselves following their pointer. On release of the input action, they will be teleported to the location of the third-person representation.

3.4.4.4 Remote Experience

Remote participants view the participant using the method of locomotion at the position which the full-body avatar is located.

3.4.5 Network Details

Forge Networking contains a contract system to define NetworkObjects, which can be used by GameObjects in order to maintain a presence across the network. On instantiation, NetworkObjects are allocated an ID by the server, which is replicated on all clients. From here, values defined in the contract can be written to the NetworkObject, relayed through the server and retrieved on remote clients. Forge achieves this by serialising each field to a byte array and sending it via a UDP message at a fixed timestep.

We chose to have individual clients possess authority over their own avatars in the context of the network. Given that the pose information is provided directly from SteamVR and that the application is non-competitive, the logical approach was to trust the client to provide truthful positional data to the server, as the potential for cheating is a non-issue. This has the added benefit of providing a smooth experience to the local player, with no risk of "rubber-banding" as a result of mismatches between the server and client.

In order to replicate player motion across the network, a contract was created with Forge which defined the following properties:

Field	**Type**	**Size**
HMD Position	Vector3	12 bytes
HMD Rotation	Quaternion	16 bytes
Hand Positions	Vector3	12 bytes
Left Hand Rotation	Quaternion	16 bytes
Right Hand Rotation	Quaternion	16 bytes

Table 3.3: NetworkObject contract fields, corresponding types, and size of types

3.4.5.1 Compressing Player Packets

As it is the case that both the left-hand controller and right-hand controller will always be within close range of the head mounted display, it was possible to encode the controller positions as a delta from the position of the head mounted display. The position is subtracted from both the left-hand and right-hand controller positions, giving a resultant position with a much smaller range. From this, we encode each floating point component of these two 3D vectors as a 16-bit value (short) by quantising the vector. To convert a floating point number to a short, the value is multiplied by 1000.0f and cast to a short, placing it in the range of -32,768 to 32,767. To restore a short representation back to a floating point number, the short is cast to a float and divided by 1000.0f, returning a float in the range of +/- 32.

Given the smaller size, it became possible to pack the two vectors into the space of a single vector by bit-shifting each component of the two vectors. This was achieved by creating three integers, initialised to zero and performing a logical OR with each of the three integers and the corresponding 16-bit representation of each component of the first vector. Each integer was then bit-shifted 16 bits to the left and the components of the second vector were packed again using a logical OR. The result was three 32-bit integers containing the six 16-bit vector components.

Despite quantisation, no loss of precision was noticeable given the tracking precision of the SteamVR system. On the receiving end, the fields were split back into the respective components and restored to lower-precision floating point values. As a result, it was possible to reduce bandwidth per-packet by a small amount.

3.4.5.2 Serialising State

On each call to Apply by the `VRNetworkController` to the currently active `ILocomotionState`, the `ILocomotionState` will apply the locomotion logic it represents to the local user state and store the values of the new player state to the NetworkObject.

All NetworkObjects locally owned by the client tick at a fixed interval rate. On each tick, the current values of the fields defined in the NetworkObject contract are serialised to a byte array and written to a UDP packet alongside some routing information. This UDP packet is sent to the server, which relays the packet information to all other clients connected to the server. An interval rate of 33 ms was chosen for the player NetworkObject. This is relatively high for a non-competitive network experience and is generally used for much more fast-paced multiplayer experiences, such as first-person shooters.

Over the course of a second, a single network-enabled player packet with an update interval of 33 milliseconds (ms) would send 2520 bytes. In comparison, with quantisation, this value is reduced to 2160 bytes.

3.4.5.3 De-Serialising and Interpolation

On remote clients, the NetworkObject polls for incoming messages. After receiving a message, the NetworkObject consumes the payload of the incoming message and applies the state contained in the method to the fields defined in the NetworkObject. Remote instances of the `VRNetworkController` component then interpolate the current state of the player prefab towards the new state contained in their NetworkObject instance with an interpolant value of 0.15f, removing any jitter that would occur as a result of hard-applying the incoming state. Given this smoothing, it would be possible to use a much lower interval rate, or even a variable interval rate depending on client bandwidth availability and still view a smooth representation of all players.

3.5 Interaction

Interaction is an important aspect of developing a believable virtual environment. In a multiplayer environment, issues arise in regard to maintaining consistency of interaction for all players in the scene, so as to ensure the experience is as identical across all clients, while also protecting against potential race conditions such as two users attempting to grab the same item.

3.5.1 Method 1 - Event-Based Interaction

The first interaction method defined in the project was an event-based model of interaction with items in the environment. Interactable items in the scene are marked as kinematic and so do not react to external forces, such as collisions or gravity. These items can be grabbed by players, and when grabbed will be bound to the hand with which they were grabbed. On release, they will return to the world and remain kinematic, being positioned exactly where released. Grab and Drop commands are issued over the network in tandem with the local calls and replicated on remote clients. These commands are used to either make an item the child of a user's hand, or to remote the item as a child of a user's hand.

3.5.1.1 The EventInteractionManager

The core component of the event-based interaction system is the `EventInteractionManager`. This is a singleton component which is contained within the scene for which the Interactables are contained. Rather than providing each individual Interactable with its own NetworkObject, they are instead registered with the `EventInteractionManager`, which handles message passing for all items. This reduces overhead in regard to the instantiation of `NetworkObjects`, as only a single instance is required to manage any number of items. As kinematic items would not have a need to repeatedly pass state to other clients, instead relying only on reacting to commands invoked upon them, all commands are routed via the `EventInteractionManager`, leaving Interactables blind to the network.

The `EventInteractionManager` defines a list of current Interactables in the scene, which is used to reference individual items across multiple clients. It also defines three Remote Procedure Calls which are used to replicate the state of items across the network, along with methods to invoke these Remote Procedure Calls from a local client.

- `GrabItem(uint ownerID, uint itemID, bool isLeftHand)` - This event is invoked when a local user with ownerID grabs item with itemID. The third parameter is used to define whether the user grabbed the item with their left hand. This will invoke the Grab command on the item on remote clients.
- `DropItem(uint ownerID, uint itemID, Vector3 position, Quaternion rotation))` - The DropItem event is invoked when a local user with ownerID releases an item with itemID. On remote clients, this will invoke the Drop command on the Interactable.
- `DropItemSnap(uint itemID, uint snapID)` - This method is invoked when an item with itemID is consumed locally by a snap point with snapID. On remote clients, this will invoke the DropSnap command on the Interactable.

These three commands define the possible actions a user can take on an item existing in the scene using the Event-Based Interaction system.

3.5.1.2 Defining Interactables

Interactables contained in the scene are defined via a component, `Grabbable`. The `Grabbable` component defines state for the current Interactable and also defines methods used to invoke Grab and Drop events on the given item. Each GameObject with attached `Grabbable` is provided a static instance ID by the

`EventInteractionManager`, which is deterministic across all clients on the network and defined by the order in which they are registered before compilation of the project.

When a command is invoked on an Interactable, for example the Grab command, the logic defined by that command is performed on the local client.

3.5.1.2.1 The Grab Command

When a user grabs an Interactable, the IssueGrab method is invoked on the item. This takes the ownerID, the GameObject of the controller grabbing the item, and what hand that controller corresponds to, as parameters. The Interactable is set as a child of the hand, with the current world position of the Interactable being maintained as it transitions to a local coordinate system. This places the Interactable in the hand of the user at the exact point they grabbed it. The state of the Interactable is set to Held, which prevents other users from snatching the item from a user. The `EventInteractionManager` is then called upon to issue the Grab RPC to remote clients.

3.5.1.2.2 The Drop Command

When the user releases an Interactable, the IssueDrop method is invoked on the item. This takes a single parameter, the ID of the current holder of the item, and unparents the Interactable from the hand GameObject, translating the local coordinates into global coordinates to maintain the correct position. The `EventInteractionManager` is then called upon to issue the Drop RPC to remote clients.

3.5.1.2.3 The Drop Snap Command

Within the scene, Snap Points are defined which act as containers for items. When an item being held by a user enters within range of the Snap Point, it is interpolated towards the Snap Point and held within. Snap Points are used to define trigger zones which invoke events when certain conditions are met, for example placing the correct item. The IssueDropSnap method is invoked on the item when it enters a Snap Point. This accepts two parameters, the ID of the current holder of the item and the ID of the Snap Point, and unparents the Interactable from the hand GameObject, then interpolating it towards the destination position. The `EventInteractionManager` is then called upon to issue the DropSnap RPC to remote clients, which do the same.

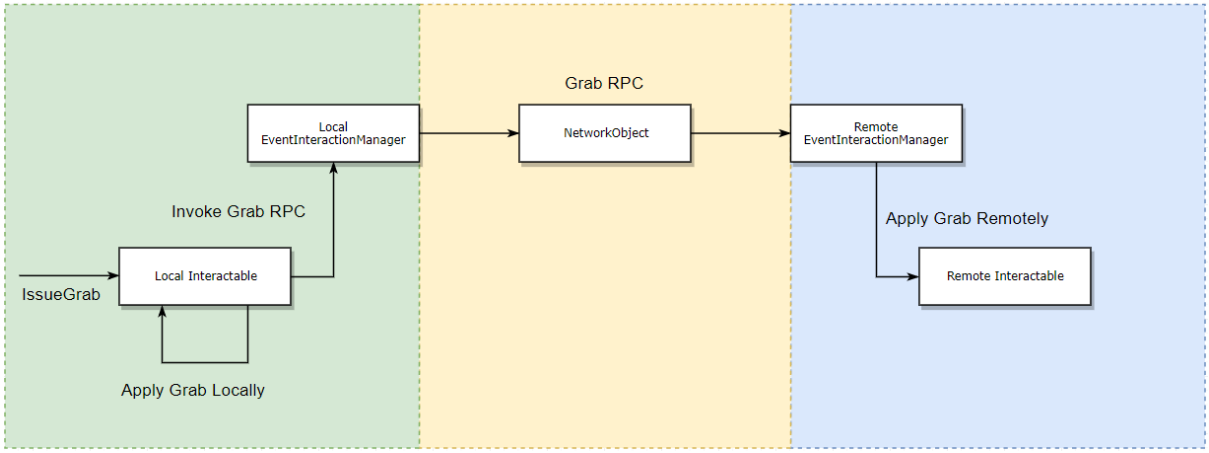


Figure 3.17: Flow of a command (Grab) being invoked on an Interactable

3.5.1.3 Player Hands

In order for participants to be able to invoke commands on an Interactable, a bridge was created between the controller and the Interactable. To do this, a component, `HandGrab`, was defined. This component allows users to grab and drop items in the scene.

The `HandGrab` component is attached to each GameObject, which represents a controller locally. The component also contains a reference to an attached BoxCollider, which is marked as a Trigger. PhysX defines the concept of trigger shapes. Trigger shapes do not interact in the collision simulation, but instead provide callbacks in response to other non-trigger shapes entering their volume. Three events exposed by Unity are used by the `HandGrab` component:

- `OnTriggerEnter` - called when a shape enters the volume of the trigger
- `OnTriggerStay` - called each frame for each collider contained inside the volume of the trigger and on the trigger itself
- `OnTriggerExit` - called when a shape exits the volume of the trigger

These callbacks are used to recognise when a potentially grabbable Interactable enters the volume of the controller. A tag, "Grabbable", was defined, which is used to validate whether an Interactable can be grabbed. If the other Collider entering or staying inside the volume is tagged as "Grabbable" the colliding field of the `HandGrab` component is set to the `Grabbable` component of the other object. When the `OnTriggerExit` callback is triggered, this field is set to null.

Each frame, a check is made for the following state:

- The grab input action was false in the previous frame and is now true
- The colliding `Grabbable` is not null
- The `heldItem` field is null (the hand is not already holding something)
- The colliding `Grabbable` is not held already
- The hand object field is set to itself (see 3.5.3)

If all these checks return true, then the Interactable is grabbed. The `heldItem` field is set to the `Grababble` contained in the colliding field and the holder of the `Grabbable` is set to this hand. The `IssueGrab` command is then invoked on the held item.

If any part of this check fails, a second check is performed:

- The grab input action was true on the previous frame and is now false
- The `heldItem` `Grabbable` is not null
- The `heldItem` field is not snapping to a snap point

If this check passes, the user is holding an Interactable and has released the controller input corresponding with the grab input action. The IssueDrop command is invoked on the Interactable and the `heldItem` field is set to null.

3.5.2 Method 2 - Physics-Based Interaction

The second method of interaction implemented as part of the research topic was that of a distributed physics-based interaction system, based on an implementation proposed by Glenn Fiedler and Oculus (7).

Interactables in the scene now contain non-kinematic rigidbodies and so respond to external forces such as gravity. They also respond to collisions from players and in certain cases other Interactables. Interactables are given the concept of ownership, represented by a client and a distributed simulation is built by having each client simulate only those Interactables for which they claim ownership. When a user grabs an item, ownership is passed to their client and they retain ownership and simulation control over the item until it is grabbed by another user.

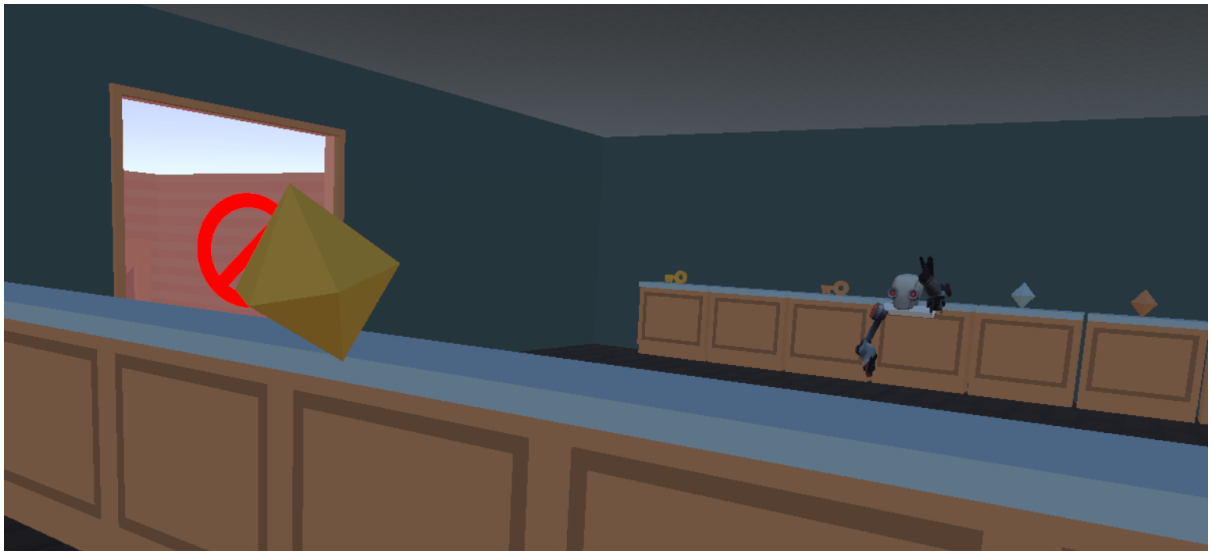


Figure 3.18: A user throwing an item

3.5.2.1 Interactables

To define a physics Interactable, a component, `PhysicsItem`, is attached to the GameObject. This component defines fields indicating whether the item is held, the hand holding the Interactable if it is being held and also two fields involving ownership. The first, ownerID, is an unsigned integer representing the connection ID of the client which is currently under authority of the Interactable. The second field, ownership, is a sequence number used to denote authority on the network. When a client wishes to take authority over an Interactable, they increment the authority sequence. The server makes use of this authority sequence to validate state updates.

It also defines a number of methods for control over the Interactable:

- `TakeOwnership()` - Increments the authority sequence and changes the ownerID to the current client.
- `SetOwnership(byte ownerSequence, uint ownerID)` - Sets the authority sequence to the ownerSequence field and changes the ownerID of the Interactable to the ownerID parameter. This is called on remote clients when state arrives from the server which invalidates the current authority state of the Interactable.
- `Consume(State s)` - Receives a `State` struct and sets the current state of the Interactable to the state defined within the parameter.

The `State` struct defines the structure for the state of the Interactable that is transmitted over the network. It is defined as follows:

```
public struct State
{
        public byte ownership;
        public uint ownerID;
        public Vector3 position;
        public Quaternion rotation;
        public bool rest;
        public Vector3 velocity;
        public Vector3 angularVelocity;
}
```

The `rest` boolean is used to determine whether the Interactable is at rest or in motion and depending on the value of this booleal, velocity and angular velocity may not be included in the state, as described in section 4.3.3.5.

Each frame, the `PhysicsItem` updates the position of the GameObject on the local client to the position of the `Transform` holding the item (see section 3.5.3). On remote clients, the GameObject position is interpolated towards the position of the hand GameObject holding the Interactable, with an interpolant value of 0.15f.

3.5.2.2 The PhysicsInteractionManager

A similar architectural approach to the Event-Based Interaction was taken when implementing Physics-Based Interaction. A `PhysicsInteractionManager` is defined, to which all Interactables are registered.

3.5.2.2.1 Client-Side Ownership When a client connects to the server, the client-side `PhysicsInteractionManager` invokes a method, `tick`, repeating 20 times a second. Each tick, a call is made to the IssueUpdate method, with a sequence number, beginning at zero, passed as a parameter. Following the call to IssueUpdate, the sequence number is incremented by one. The sequence number is stored as an unsigned integer.

In the IssueUpdate method, A C# `MemoryStream` is defined to which data can be written. A list of the PhysicsItems contained within the scene are iterated over, in order to write them to the packet payload. As the PhysicsItem list is static at runtime, clients are already aware of the number of PhysicsItems and so no size information is sent. Firstly, the sequence number is written to the `MemoryStream`. For each PhysicsItem, a check is made to ensure client ownership. A bit-field is maintained, represented as a byte array whose size is denoted as follows:

```
byteCount = (items.Count + 7)/8
```

This calculates the number of bytes needed to represent each item as a single bit. The `MemoryStream` position is shifted by `byteCount * 3` to accomodate for the space occupied by the bit-fields. If the `PhysicsItem` is owned by the current client, it should be included in the packet and so the bit corresponding to the ID of the `PhysicsItem` is set to one to indicate it has been included in the packet.

Next, a check is made regarding whether the `PhysicsItem` is currently being held by a user. If the item is held, the authority sequence, owner ID and handedness of the holding hand is written to the `MemoryStream`. A secondary bit-field, defining whether an item is held, has the bit at the ID of the `PhysicsItem` set to one to indicate it is held, in order to allow remote clients to properly de-serialise the payload.

If the item is not held, the position and rotation of the GameObject to which the PhysicsItem is attached is written to the `MemoryStream`. If the rigidbody attached to the GameObject is not at rest, a bit in a third bit-field used to determine whether a `PhysicsItem` is at rest is set to one. Following this, both the velocity and angular velocity of the rigidbody are written to the `MemoryStream`. If the rigidbody is at rest, the velocity and angular velocity are both zero and so do not need to be transmitted.

The index of the bit which corresponds to the respective `PhysicsItem` is calculated by incrementing a bit counter each time a `PhysicsItem` is iterated over. When the bit counter reaches eight, it is reset to zero and a byte counter is incremented by one. To access the corresponding bit, the byte counter is used to index into the byte array representing the bit-field and then a byte of value one is shifted to the left by the

number of bits specified in the bit counter. At the end of the iteration loop, the three bit-fields are written to the start of the payload by shifting the position of the `MemoryStream` back to zero and writing the three byte arrays directly. The `MemoryStream` is then converted to a byte array and sent to the server only via an unreliable RPC.

3.5.2.2.2 Server-Side Filtering On the server, when the RPC is received, the payload is received as a byte array and placed into a `MemoryStream`. The three bit-fields are read from the `MemoryStream`, alongside the sequence number of the packet. The server maintains a `Dictionary` containing a mapping of the client connection ID to the last sequence number received by the client. If the mapping for a given client is non-existent, an entry is created with the received sequence number as a value. Otherwise, the received sequence number is compared to the last received sequence number using an algorithm defined as part of PAWS (8), the TCP algorithm for protection against wrapped sequence numbers.

A method is defined, `CompareSequenceNumbers`, which takes two unsigned integers and the maximum value allowed by the sequence number being compared and returns one of the three following values:

- 1 - returned if the first sequence number is newer than the second
- 0 - returned if the sequence numbers are equal
- -1 - returned if the first sequence number is older than the second

The method body is defined as follows:

```
//1 if s1 is newer than s2
//0 if same
//-1 if s1 older than s2
static int CompareSequenceNumbers(uint s1, uint s2, uint maxValue)
{
    if (s1 == s2)
        return 0;
    else if (SequenceOlder(s1, s2, maxValue))
        return -1;
    else
        return 1;
}

//Is s older than t?
static bool SequenceOlder(uint s, uint t, uint maxValue)
{
    return 0 < (t - s) && (t - s) < maxValue;
}
```

If the incoming sequence number is older than the current sequence number, the incoming packet is out-of-date and is thus discarded. Otherwise, the last sequence number is updated to the incoming sequence number and the packet is processed.

If the item is marked as held in the bit-field, the authority sequence, owner ID and hand fields are retrieved from the byte array. Validation is performed to ensure the authority sequence is greater than the current authority sequence known by the server, indicating authority has been taken. If the authority sequence is the same, the validation falls back to checking the owner ID against the current owner of the `PhysicsItem`. If the owners are the same, then the state is valid. Otherwise, the state is rejected. If the authority sequence is older, then someone else is now the authority for this item and the state is rejected.

If the state is valid, then the ownership of the `PhysicsItem` is updated via `SetOwnership` and the item is set to the held state. The state is then copied to an outgoing `MemoryStream` for forwarding to other clients.

If the item is not held, the position and rotation of the item are read from the payload as a series of floats and reconstructed into a `Vector3` and `Quaternion`. If the item is not marked as at rest, the velocity and angular velocity are also read as a series of floats

and reconstructed. These reconstructed fields are used to construct an instance of the `State` enum, and this is consumed by the `PhysicsItem`. The state is then copied to the outgoing `MemoryStream`.

After all state has been validated and consumed, the outgoing MemoryStream is sent to all clients, ignoring the sender, as a byte array. This allows the server to validate incoming state from clients and solve any possible conflicts from clients.

3.5.2.2.3 Receiving On Client On remote clients, when the RPC is received from the server, the client de-serialises the packet based on the three bit-fields defined at the beginning of the packet. As the server has already performed filtering on state, all incoming state is valid, and so can be de-serialised without further validation.

If a given `PhysicsItem` is marked as held in the incoming payload, the ownership of the `PhysicsItem` is set, and the `heldPosition` field is set, which is the position the `PhysicsItem` occupies when it is marked as being held. This binds the `PhysicsItem` to the hand of the player holding the item.

If the `PhysicsItem` is not marked as held, an instance of the `State` enum is constructed from the payload. The state is placed in a `Queue` for the given `PhysicsItem`, which acts as a buffer to remove jitter. Each tick, the next state is removed from the `Queue` and consumed by the `PhysicsItem`.

3.5.2.3 Physics Hand

In order to allow users to grab items in the scene using this method of interaction, a component, `PhysicsHandGrab`, was defined. The component performs the same checks as performed by the `HandGrab` script regarding items entering the volume of the controller, however instead requires that the GameObject contains a `PhysicsItem` component rather than a `Grabbable` component.

Each frame, a check is made for the following state:

- The grab input action was false in the previous frame, and is now true
- The colliding `PhysicsItem` is not null
- The `heldItem` field is null (the hand is not already holding something)
- The colliding `PhysicsItem` is not held already
- The hand object field is set to itself (see 3.5.3)

If these conditions return true, then the Physics Interactable is considered to have been grabbed. The `heldItem` field is set to the attached `PhysicsItem` of the colliding Interactable, and if the Interactable is not owned by the grabbing user, ownership is taken via `TakeOwnership`. The rigidbody of the Interactable is made kinematic. The Interactable is marked as being held.

If any part of this check fails, a second check is performed:

- The grab input action was true on the previous frame, and is now false
- The `heldItem` `PhysicsItem` is not null

If these conditions return true, the Interactable has been released from the grip of the user. The Interactable is marked as no longer held, the rigidbody is made non-kinematic, and the velocity and angular velocity of the rigidbody are set to the velocity and angular velocity of the controller GameObject, retrieved from the SteamVR pose information.

3.5.2.4 Collisions

Interactables which share the same owner are able to collide with each other. As the resulting state of all Interactables under authority from the same owner is included in the same packet, state remains consistent. Interactables which are not owned by the local user are made partially transparent for that user, and will not interact with items that do not share an owner with it.

3.5.3 Combining with Locomotion

While both of these methods of interaction required no modification in order to be usable with both Teleportation and Artificial Locomotion, special cases were required regarding the Third-Person Avatar method of locomotion.

The remote view of a player using the third-person avatar method of locomotion differs to that of the local experienced of the user triggering the locomotion method. Locally, the player avatar remains at the position of the player relative to the room centre. Remotely, the player avatar is shown at the current location of the local third-person avatar.

Issues arise when the user triggering the third-person avatar is holding an Interactable in their hand. On the local user, the Interactable would remain at the position of the controller for which the hand is represented. Remotely, the Interactable would thus be shown in a different position, depending on the method of interaction used.

For the Event-Based Interaction method, the Interactable becomes a child of the hand of the remote player avatar. It would then be positioned at the correct location for the remote user, but in a different location to that of the local users view of the Interactable. If the local user were to release the Interactable before teleporting to the third-person avatar location, remote users would see the Interactable teleport from the hand of the remote player avatar to the position it was released at on the local client.

Similarly, for the Physics-Based Interaction method, the remote user receives the position of the Interactable from the server based on where the Interactable was located on the client under authority of the Interactable.

In order to resolve these issues, a `Transform` property, `handObject` is defined in both `HandGrab` and `PhysicsHandGrab`. A `Transform` is a Unity component attached to all GameObjects in a scene, used as a container to hold the position, rotation and scale of the GameObject it is attached to. This property was used to define the current position, rotation and scale of the GameObject for which the Interactable should be attached. When the third-person avatar locomotion method is triggered, the property for the given active component on each controller is set to the corresponding hand bone on the third-person body mesh. When the third-person avatar locomotion method completes, the property is restored to the controller `Transform`.

An event, `OnHandChanged`, is triggered when the property is set to a `Transform` that differs to the current value. This is used to correctly update the parent of Event-Based Interactables. As held Physics-Based Interactables update their position and rotation each frame from the property, no change is required on hand change.

This results in any Interactables being held properly transitioning to be held by the full-body avatar when the user triggers this method of locomotion. If the user releases an item while it is being held by the full-body avatar, the item will be dropped from the hand of the full-body avatar, thus correcting the difference in state between the local experience and the remote experience.

4 Evaluation

In order to evaluate both the locomotion and interaction methods defined above, a test scenario was devised which involved two participants working together to complete two tasks designed to make use of both locomotion and interaction mechanics. Participants were tasked with navigating and completing both tasks, using all three methods of movement and both methods of interaction. Following this, they were asked to complete a short survey about their time in Virtual Reality.

4.1 Scenario

The scenario begins with both users being placed in a shared space resembling a lobby area. Users were given time to explore the shared space before beginning the first task. The space contained several objects, which could be navigated, alongside a small number of Interactables. One wall contained a graphic, which described the control scheme for the scenario, and another contained tutorial text guiding the participants through one of two doors, depending on whether they were acting as the host or client.

After a short amount of time had passed, the participants were instructed to enter through the door that was visibly unlocked on their client, thus dividing the user into two channels. The first channel, entered by the host, participant A, contained a short hallway with a table and staircase. On the table was a map containing a colour-coded grid of 12x12 squares. Participant A was instructed to take the map and travel up the staircase to a balcony overlooking a large square room.

The second participant, participant B, was entered into this room. Contained on the floor of the room was a 12x12 grid of metal panels, with two blue tiles at the side closest to participant B, and a single blue tile at the exit on the other side of the room. The boundaries of tiles were clearly marked with a dark square outlining the border of tiles. These tiles matched the tiles featured on the map provided to participant A.

4.1.1 Navigation

Participant A was tasked with guiding participant B through the maze, by observing the movements of participant B through the maze, and by guiding participant B via

the map. Green tiles indicated squares which could be walked on, while red tiles indicated squares which would trigger a failure and teleport participant B back to the beginning of the maze if stepped on.

This task gave a clear analysis of both using each movement mechanic to navigate a defined space, while also requiring participant A to remotely observe the motion and direction of participant B in order to ensure they were traveling correctly based on instructions.

Following the completion of the maze, the process was flipped. Participant A was placed into a secondary maze, while participant B was provided with a similar map and brought to a balcony overlooking the second room. The process was repeated in order to ensure a symmetrical task environment for both users, when completing the survey at the end of the scenario.

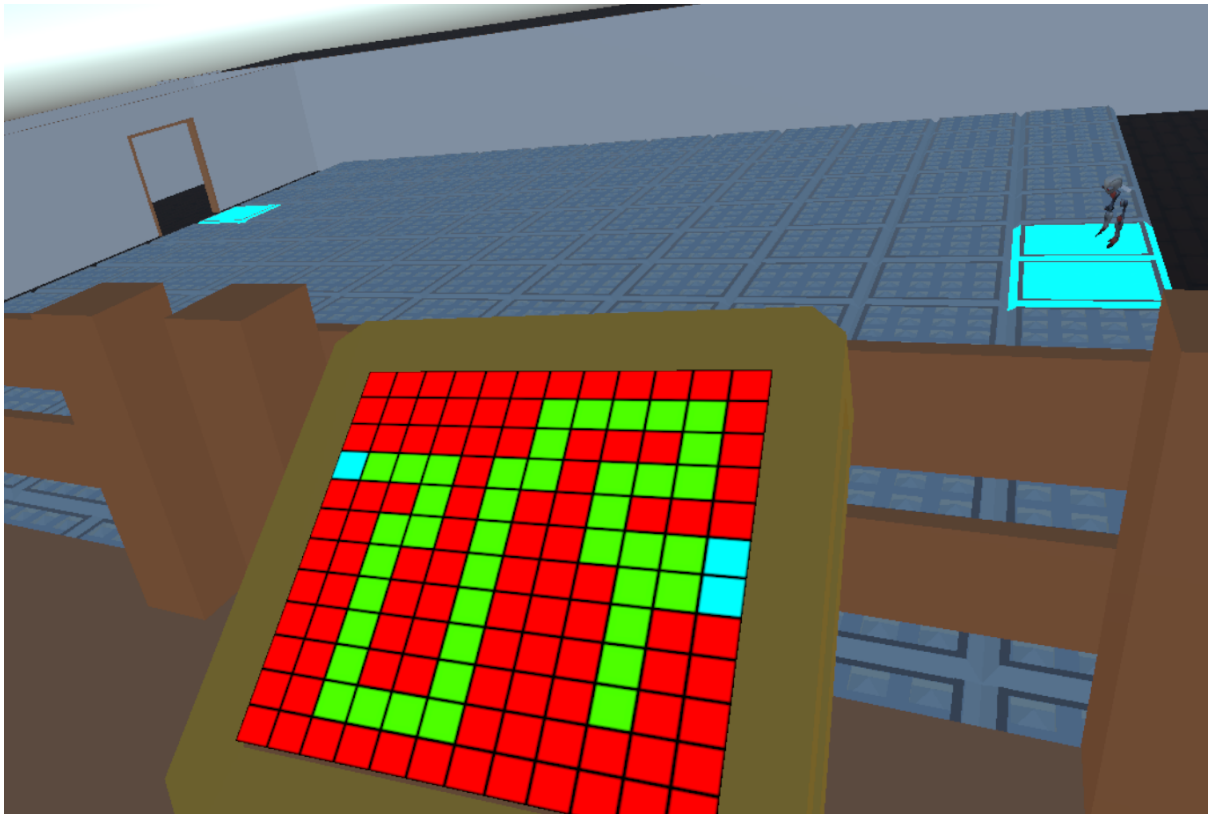


Figure 4.1: View from the overlook, with participant A holding the map, and participant B located at the beginning of the maze

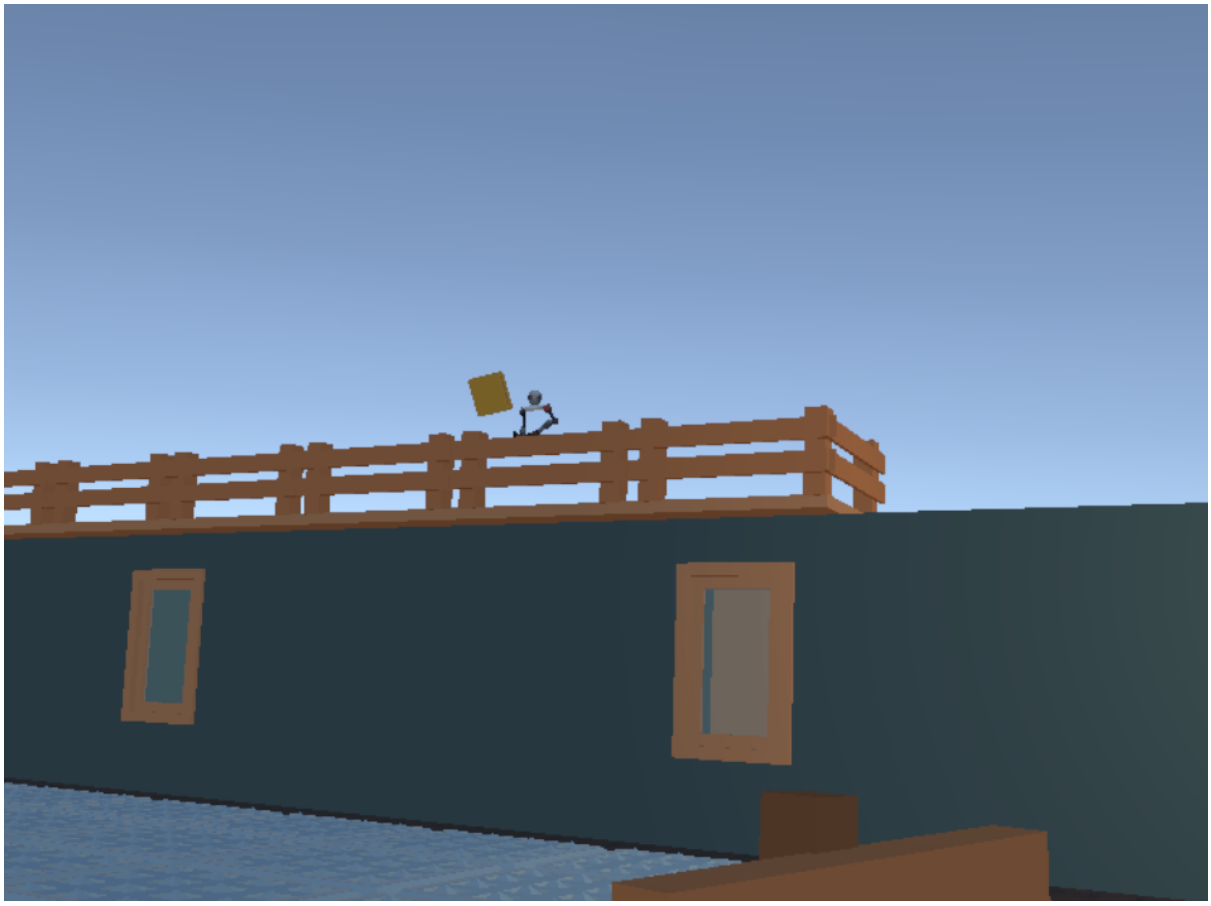


Figure 4.2: View from the maze of participant A holding the map, from participant B located at the beginning of the maze

4.1.2 Interaction

Following the completion of the first task, the participants were navigated back to a single room. The room was divided into two separate sections by a small 1.5 meter tall wall, with one participant placed on either side of the wall. In one section of the room, nine different items were placed on a row of pedestals. The items were divided into 3 types, `Keys`, `Gems` and `Coins`, with each type being represented by a tier, either `Gold`, `Silver` or `Bronze`.

Figure 4.3: Participant A prepares to receive the silver key from participant B

In the opposing section of the room, six pedestals were placed. Each pedestal contained the model of an item placed on the front panel, and a trigger point placed on top (either a snap point, or a physics trigger point, depending on the method of interaction being tested).

Participants were tasked with passing the required items labeled on the six pedestals from the nine contained on the opposite side of the room, across the small wall, and placing them into the individual trigger points. When an item was placed in the correct trigger point, the trigger point would turn green. When all trigger points had been turned green, the barriers of exit would open and the participants would be able to exit the room.

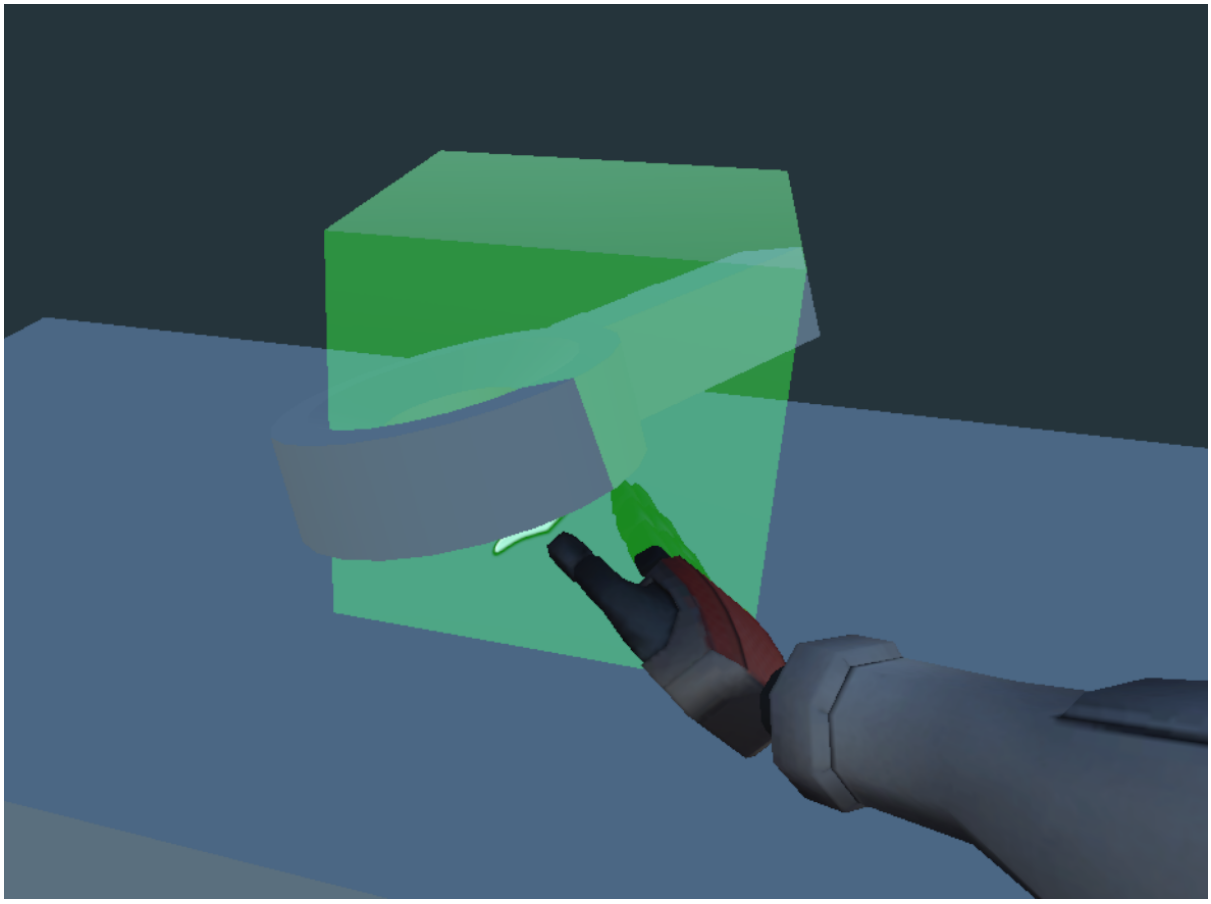


Figure 4.4: Participant A places the silver key in the correct snap point

Of particular interest in this task was whether or not the participants would have difficulty transferring items, whether they would use both hands for interaction or reserve one for locomotion only, and which locomotion method they would use in order to complete the task.

4.2 Survey

Following the interaction, both participants were asked to complete a survey with questions involving the scenario they had just completed. The questions were posed using a four-option Likert scale, with a legend as follows:

Value	**Ordinal**
Never	1
Rarely	2
Sometimes	3
Always	4

Table 4.1: Survey response values and corresponding ordinal value

Questions asked were divided into categories, based on the aspect of the scenario they associated with.

4.3 Results

4.3.1 Participant Information

Twelve participants were paired based on availability, and worked together to complete the scenario. Nine of the participants were male, and three were female. All participants originated from a STEM-related subject, and were selected from the student body in Trinity College Dublin.

Of the participants surveyed, eight participants had previously experienced Virtual Reality in some form, either via an OpenVR setup such as the HTC Vive, or a phone-based experience such as Google Cardboard, however none possessed their own Virtual Reality system. Ten of the participants indicated that they frequently play video games. Of these ten, seven played 1-2 hours a day and three played 3-5 hours a day.

Nine of the participants responded as not being prone to motion sickness, while three responded to being prone to motion sickness regularly.

4.3.2 Locomotion

Participants were asked six identical questions for each locomotion method following the scenario. The following results were observed for each method[1]:

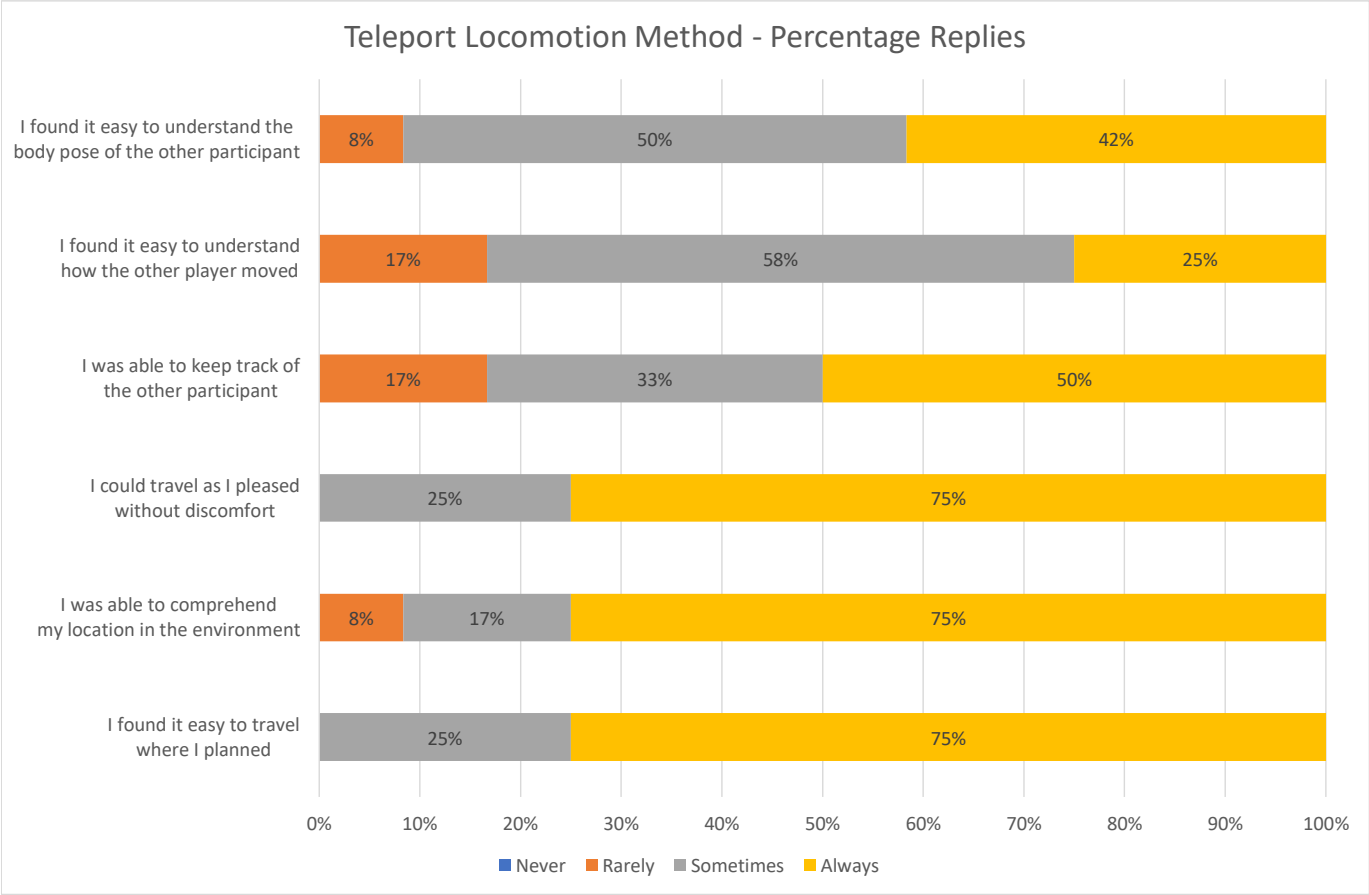


Figure 4.5: Percentage Responses for Teleporation

[1]Percentage values have been rounded down to nearest whole percentage

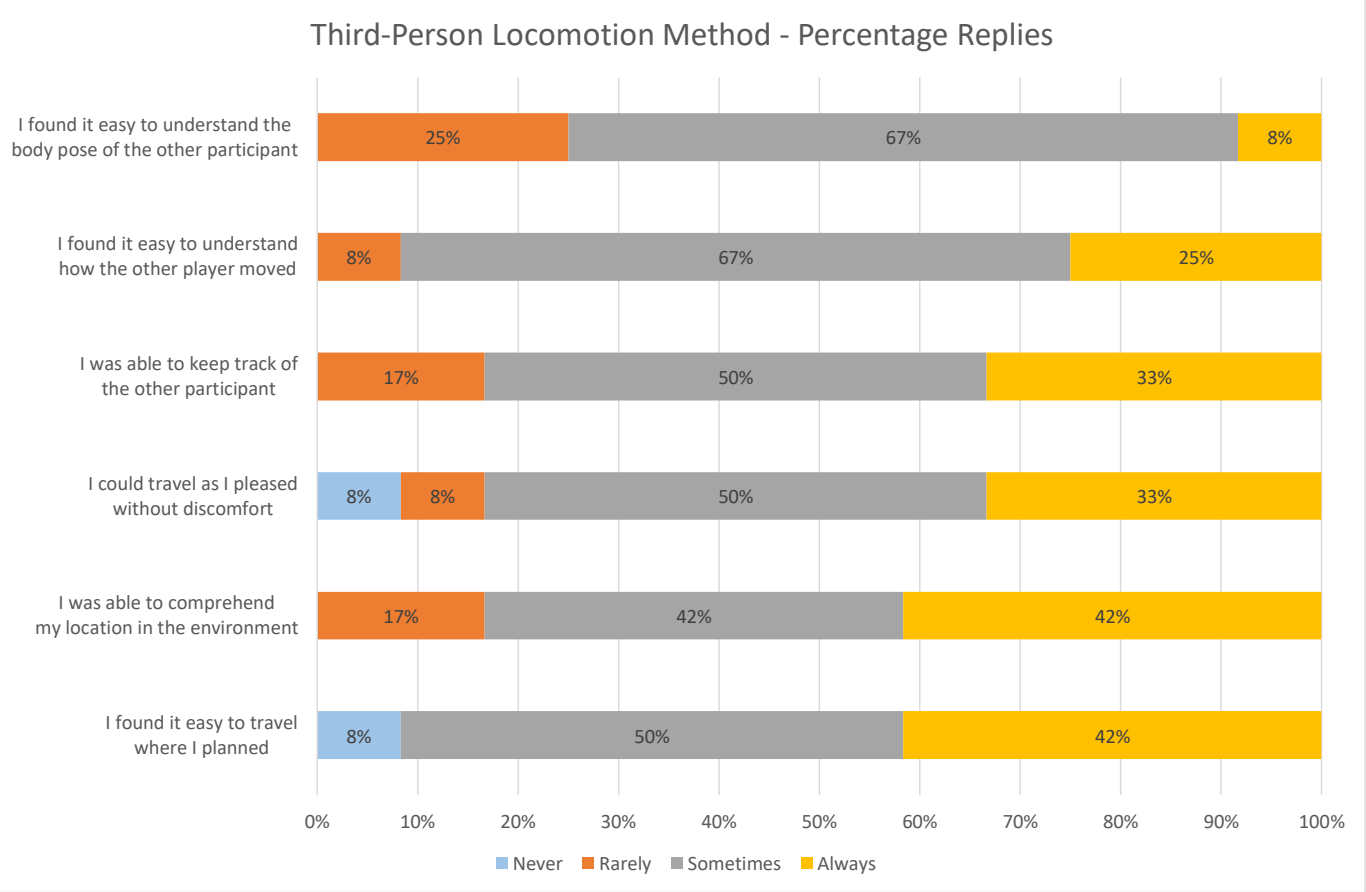


Figure 4.6: Percentage Responses for Third-Person Locomotion

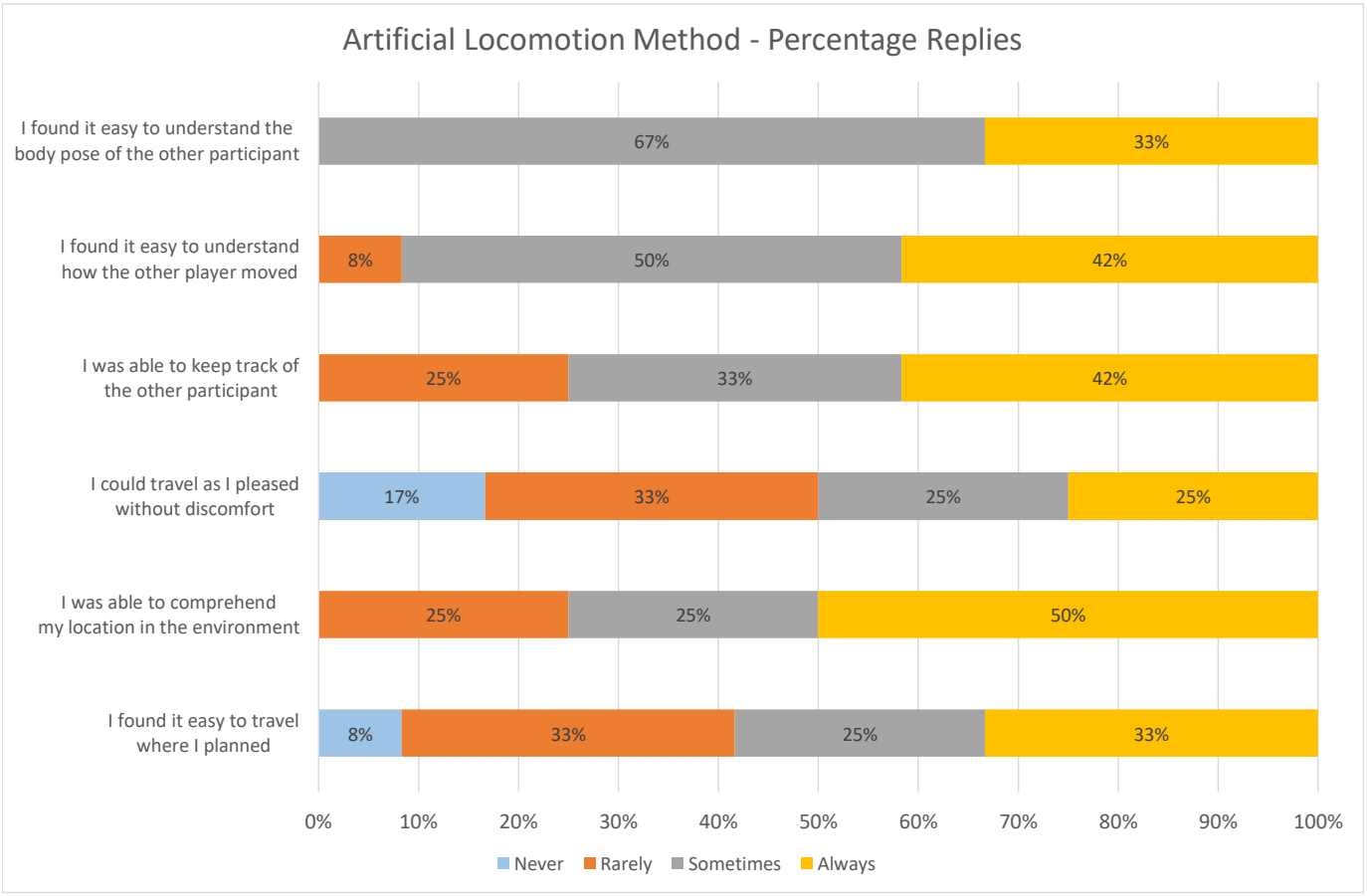


Figure 4.7: Percentage Responses for Artificial Locomotion

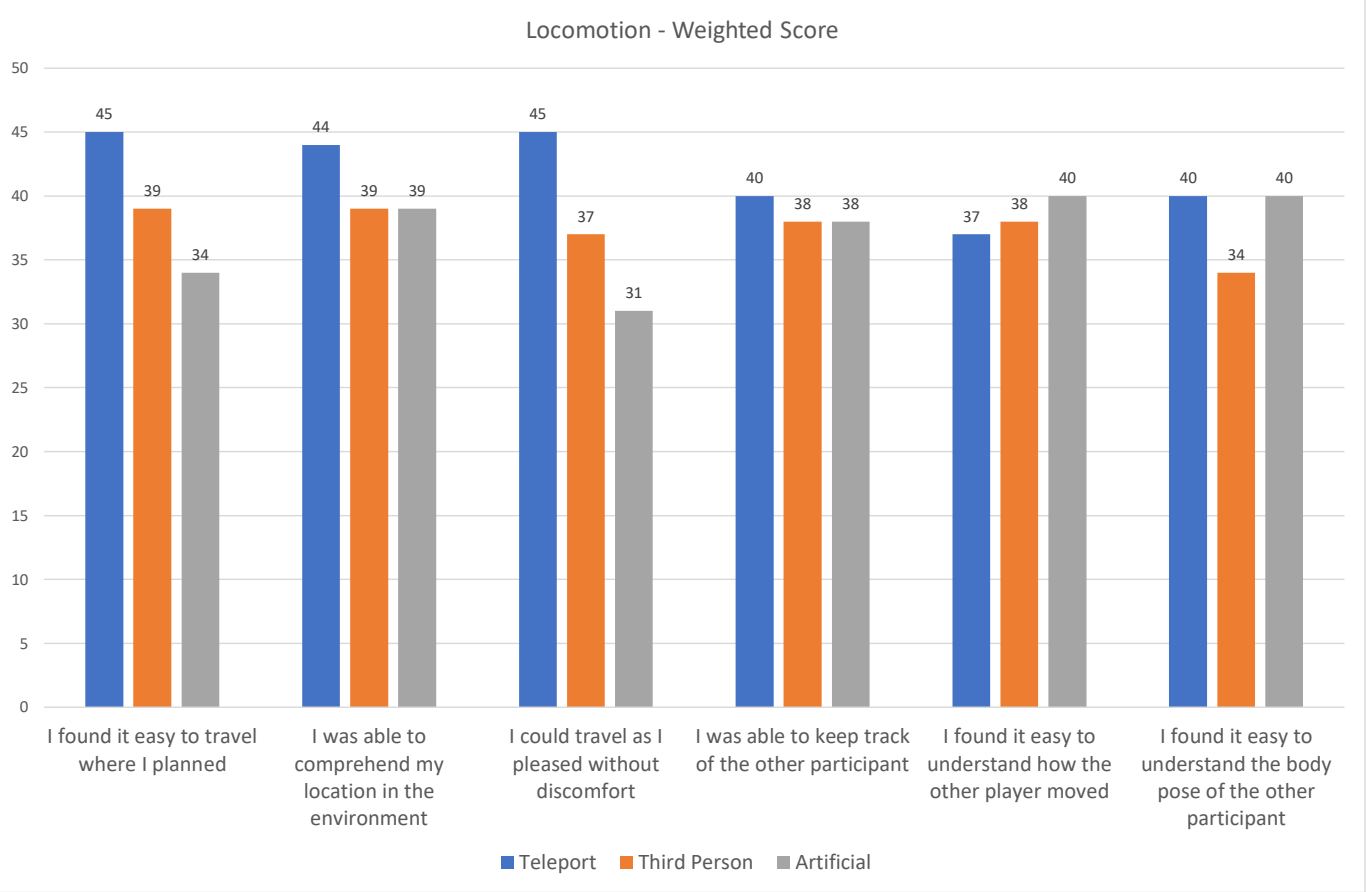


Figure 4.8: Weighted scores for Locomotion responses

Participant responses for each individual question were weighted by performing a sum of the ordinal values provided as a response to each question. Ordinal values can be found in Table 4.1. Overall, Teleportation was found to be the most preferred method of locomotion for the majority of categories among participants surveyed.

4.3.2.1 Ease of Travel

Participants responded that they found traveling to their intended destination to be easiest via teleportation, with a weighted value of 45. The Third-Person Avatar method of locomotion was the second favoured method of locomotion. The difference in ranking here is believed to be as a result of the speed of teleportation, which provides an instant translation of the user to their targeted destination. In

comparison, both Third-Person Avatar and Artificial Locomotion methods involve travel time. Artificial Locomotion, the slowest of the three methods in regard to travel time, ranked lowest with a weighted value of 34.

4.3.2.2 Comprehension of Location

Both Teleportation and Third-Person Avatar locomotion methods also supplied a laser pointer to allow participants to witness their desired movement, allowing participants to pre-plan their movement based on the destination. The second question polled participants on how well they felt they were able to comprehend their location in the environment using each method of locomotion. Again, teleportation was found to be the most successful in regard to participants comprehending their location, receiving a weighted score of 44. This is believed to be as a result of a combination of the laser pointer to show the resulting destination before teleportation, followed by instant travel. Both the Third-Person Avatar and Artificial methods of locomotion received an equal weighted score of 39. The lower ranking for the Third-Person Avatar method is believed to be as a result of users confusing the appearance of the full-body avatar with the second participant, and also with the full-body avatar blocking the view of the laser pointer while choosing a destination.

4.3.2.3 Experience of Discomfort

The third question polled was the level of discomfort experienced while traveling using each locomotion method. Again, teleportation was recorded as invoking the least amount of discomfort among users. The main source of discomfort when using a Virtual Reality system is that of motion sickness. Motion sickness occurs when there is a mismatch between visually perceived movement and the sense of movement recorded by the participant's vestibular system. Teleportation does not invoke a sense of visually perceived movement, as the change in position occurs over a single frame. In contrast, Artificial Locomotion occurs over multiple frames (while the user is triggering the locomotion method) and as a result causes a mismatch between the visually perceived movement, and the lack of body movement perceived by the vestibular system relative to the physical world. To compare, teleportation received the highest weighted value for this item, with a score of 45, while Artificial Locomotion received the lowest score of 31, indicating a high rate of discomfort. In comparison, Third-Person locomotion caused less discomfort compared to Artificial locomotion and more discomfort in comparison to teleportation. This is believed to be not as a result of the movement itself, given that the core of the Third-Person

locomotion is teleportation, but instead as a result of the sense of the third-person avatar appearing from within the participant and walking out from the user. This sense of disembodiment may have led to discomfort for some participants.

4.3.2.4 Remote View of Player Avatar

The final three questions involve the remote view of a player avatar. Overall, teleportation achieved the highest weighted score, however the difference between all three methods was very slight. Given the identical remote perception of both Third-Person locomotion and Artificial locomotion, the equal weighting is consistent with the experience perceived by the participants. Overall, users found it most difficult to understand how the remote participant moved with teleportation, which is believed to be as a result of the lack of travel time in regards to teleportation.

Overall, remote perception of a player avatar could be improved by the introduction of a full-body avatar, either through the use of additional tracked devices for the feet of the user, or by estimating foot position based on pose. This would integrate well with the Third-Person Avatar method of locomotion, as the use of full-body avatars would allow for blending between first-person and third-person perspectives. Third-Person locomotion received the worst weighted score in regards to the understanding of the remote body pose, which could be improved with the full-body tracking.

4.3.3 Interaction

Figure 4.9: Weighted scores for Interaction responses

Four questions were posed in regard to the two interaction methods implemented as part of the project.

4.3.3.1 Grabbing and Releasing

The first question involved the grabbing and releasing of Interactables within the scene. Given the identical nature of the process of grabbing and releasing Interactables regardless of interaction method, it was expected that the weighted scores would be very similar. After weighting the responses, the two scores were found to differ by a single point. Both systems scored in the upper range of scores, indicating that neither system was ultimately found to cause detriment to the participants during the scenario.

4.3.3.2 Reactions to Environment

The second question involved the reaction of Interactables to interactions in the scene. For this question, the Physics-Based Interaction system scored a higher weighted sum. This is believed to be as a result of the Physics-Based Interaction system being more true to real-world interactions, whereby the participant would expect Interactables to respond to gravity and collisions from other Interactables, in contrast to that of the Event-Based Interaction system. However, the difference is not significantly large enough given that both systems scored highly to indicate that the participants experienced a negative experience during the Event-Based Interaction scenario as a result of their expectations.

4.3.3.3 Precision Placement

It was expected that the Event-Based Interaction system would allow for the highest level of precision in regard to the placement of Interactables, given the lack of external stimulation on Interactables after they have been released by a participant. This expectation was reflected in the weighted score, where the Event-Based Interaction system scored marginally higher. However, given the small distance between both systems, and the overall high value of the weighted scores, it can be inferred that neither system was detrimental to the precise placement of Interactables during the scenario, especially with consideration towards the scenario tasks which relied on the precise placement of Interactables in trigger zones in order to complete the scenario.

4.3.3.4 Participant Interaction

The final question posed to participants involved the interaction between participants centring around Interactables, in particular passing items between participants, which was a large aspect of the Interaction aspect of the research scenario. During implementation, it was chosen to not allow participants to directly grab Interactables that are being held by another participant, in order to reduce potential ownership conflicts. In the context of the participant survey, both systems scored very similarly when weighted, with only a single point margin. However, overall, both systems scored significantly lower than in previous questions. This is believed to be as a result of the decision to disallow the direct handover of items, instead requiring a participant to first drop an item before another participant could grab the item.

4.3.3.5 Network Results

Event-Based Interaction relies solely on TCP messages in order to function. Each payload contains an initial 4 byte identifier to identify the packet type. The three possible payloads are of the following sizes:

- `GrabItem` - the Grab command contains 4 bytes storing the owner ID, 4 bytes storing the item ID, and a single byte indicating the handedness.
- `DropItem` - the Drop command contains 4 bytes storing the owner ID, 4 bytes storing the itemID, 12 bytes of positional data and 16 bytes of rotational data.
- `DropItemSnap` - the Drop Snap command contains 4 bytes storing the item ID, and 4 bytes storing the snap point ID.

Excluding TCP and Forge headers, this results in the following total payload sizes:

Packet	**Size**
GrabItem	13 bytes
DropItem	40 bytes
DropItemSnap	12 bytes

Table 4.2: Packet and Packet Payload sizes.

In comparison, Physics-Based Interaction uses significantly more bandwidth, as state must be streamed each tick for all items.

Each payload has an initial overhead of:

$$byteCount = ((n + 7)/8) * 3$$

where n is the number of items under authority for the given user. This is defined in order to store bit fields, which represent boolean values as a single bit rather than an entire byte.

To reduce overall total bandwidth, there are three possible ways in which the Interactable can be serialised.

An Interactable not held will serialise the following fields:

Field	**Size**
authority sequence	1 byte
owner ID	4 bytes
position	12 bytes
rotation	16 bytes
Not at Rest? Include:	
velocity	12 bytes
angular velocity	12 bytes

Table 4.3: Packet and Packet Payload sizes.

If the Interactable is at rest (velocity/angular velocity of zero), then these will not be included in the payload. This gives an at rest Interactable a payload footprint of 33 bytes. If the Interactable is in motion, it occupies a payload footprint of 61 bytes.

If the Interactable is held, it instead serialises to the following fields:

Field	**Size**
authority sequence	1 byte
holder ID	4 bytes
hand	1 byte

Table 4.4: Packet and Packet Payload sizes.

This results in a held item containing a payload footprint of only 6 bytes. Overall, The Event-Based Interaction provides lower total bandwidth usage, with, according to the weighted score, minimal loss of quality of use compared to Physics-Based Interaction. The scenario posed as part of the research was targeted at more precise placement of items, and ultimately it should be up to the given implementation as to whether or not

the preciseness of the Event-Based Interaction is required over the greater simulation interaction of the Physics-Based Interaction method.

Delta compression was also explored, whereby each state is built by taking the difference from the last received state for each other user. Given the lack of server authority in regard to state, it becomes more difficult to manage and acknowledge state for all players.

5 Conclusion

The aim of this project was to investigate multiple methods of locomotion in a multiplayer Virtual Environment, and compare the methods of locomotion in regard to comfort, ease of use, local and remote perceptions. The secondary aim of the project was to investigate two different methods of interaction with objects in a multiplayer Virtual Environment, with consideration for consistency, precision, expectations and bandwidth usage.

It was found that, of the participants surveyed, for a scenario which relied on precise movement, the most successful method of locomotion was that of teleportation. Considering the non-competitive nature of the scenario, there is little advantage gained by users as a result of the instant travel time. For a more competitive system, it may be the case that there is a requirement for a movement mechanic which introduces an artificial travel time, while also providing users with a better understanding of the movement of remote player avatars. Little difference was found between the two methods of interaction from a user perspective for the given scenario, and so it is recommended that an interaction method be chosen based on the requirements of the application, for example bandwidth restrictions and interaction requirements.

Given the small sample size used as part of the survey, it may be the case that the data is not entirely statistically significant. As a result, future work will involve performing a similar scenario with a larger sample size. The large majority of participants had not experienced Virtual Reality to any large degree, and so it would be interesting to consider a larger sample of participants who had a more involved experience with Virtual Reality.

Bibliography

[1] Ben Lang. Latest HTC Vives Are Shipping with Tweaked Base Stations, Redesigned Packaging – Road to VR. URL https://www.roadtovr.com/latest-vive-shipping-with-tweaked-base-stations-redesigned-packaging/.

[2] Misha Sra, Abhinandan Jain, and Pattie Maes. ACM ISBN 978-1-4503-5970-2. doi: 10.1145/3290605.3300905. URL https://doi.org/10.1145/3290605.3300905.

[3] SteamVR :: Controllers Controllers Controllers: Introducing SteamVR Input. URL https://steamcommunity.com/games/250820/announcements/detail/3809361199426010680. Accessed: 2019-03-21.

[4] Space robot kyle. https://assetstore.unity.com/packages/3d/characters/robots/space-robot-kyle-4696. Accessed: 2019-02-10.

[5] Mathias Parger, Joerg H Mueller, Dieter Schmalstieg, Markus Steinberger, and Markus Stein. Human Upper-Body Inverse Kinematics for Increased Embodiment in Consumer-Grade Virtual Reality. 18. doi: 10.1145/3281505.3281529. URL https://doi.org/10.1145/3281505.3281529.

[6] VRChat Developers. Introducing "Holoport" Locomotion – VRChat – Medium. URL https://medium.com/@vrchat/introducing-holoport-locomotion-9ada3abec63. Accessed: 2019-02-03.

[7] Glenn Fiedler. State Synchronization | Gaffer On Games. URL https://gafferongames.com/post/state_synchronization/.

[8] G Sadasivan, JN Brownlee, B Claise, and J Quittek. *Architecture for IP flow information export*. RFC Editor. URL https://tools.ietf.org/html/rfc7323{#}section-5.

A 1 Appendix

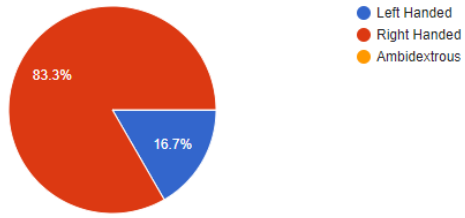
A 1.1 Survey Link

https://forms.gle/6aXJV811Y8jeNEiT6

A 1.2 Participants

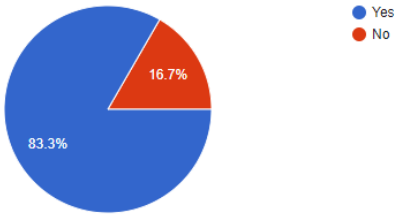
I am:

12 responses



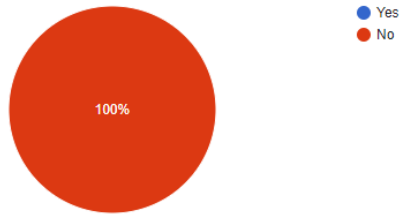
Do you play video games?

12 responses



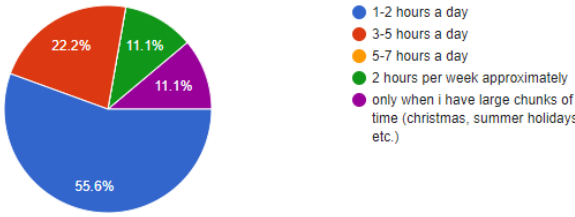
Do you own a virtual reality system?

12 responses



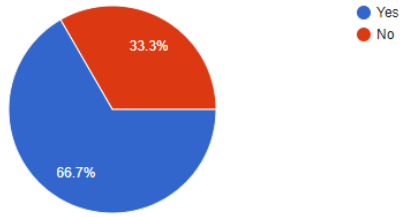
If yes, how often do you play?

9 responses



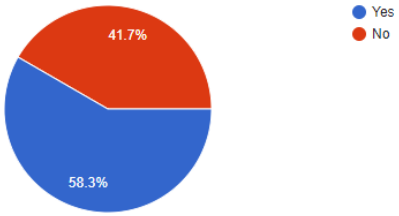
Have you experienced virtual reality before?

12 responses



Do you play fast paced games? (i.e. First-Person Shooters)

12 responses



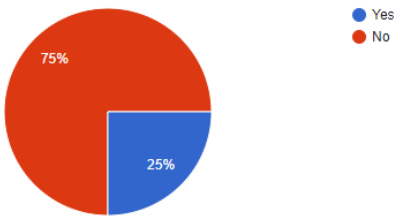
If yes, what was your experience?

7 responses

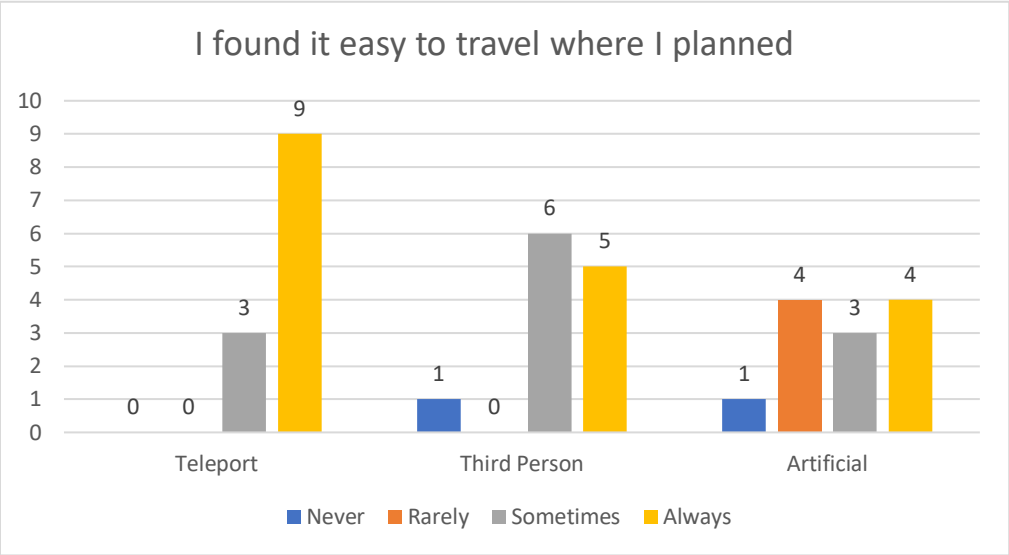
friends vr setup during parties
It's Amazing
phone VR
Really fun
A roller coaster video
I used it for work.
playing games

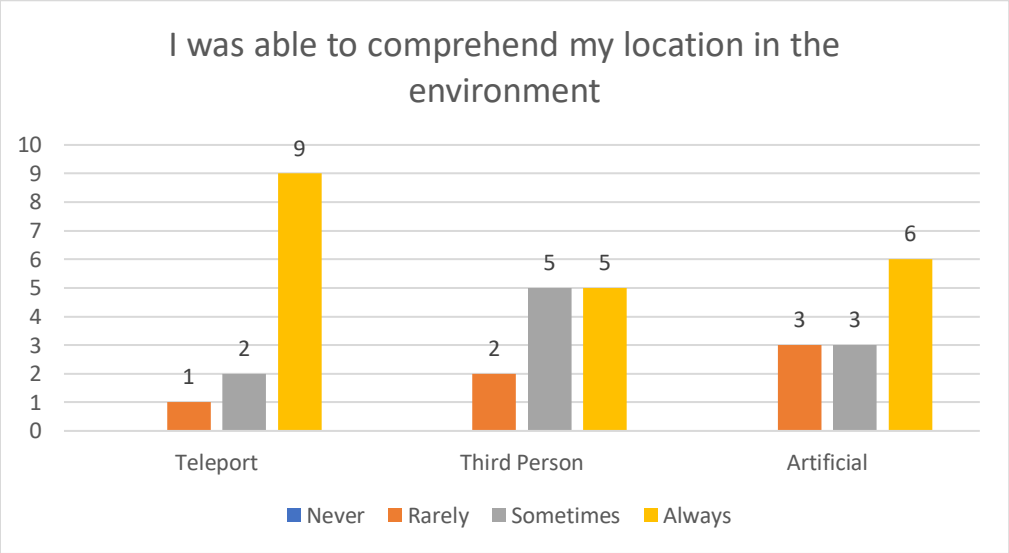
Are you prone to motion sickness?

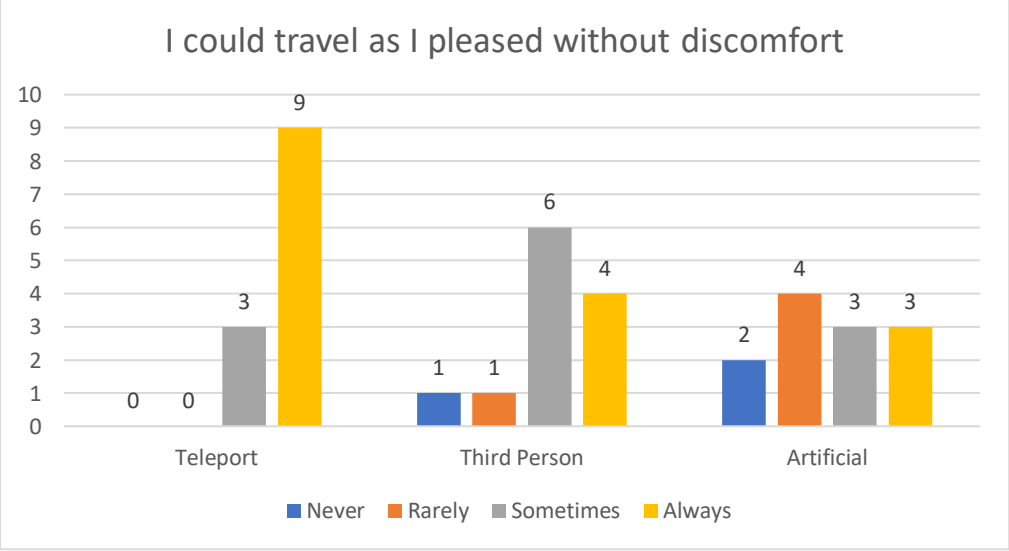
12 responses

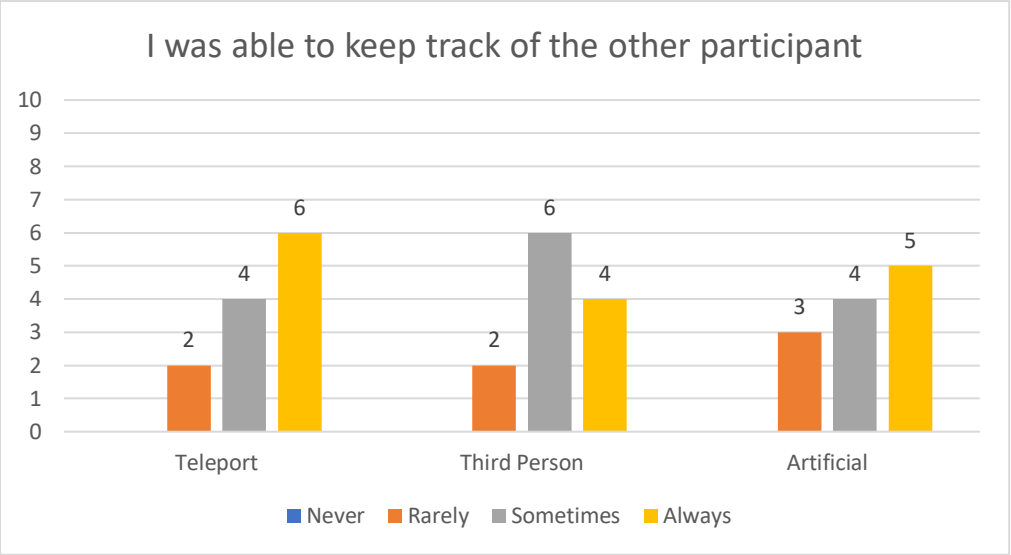


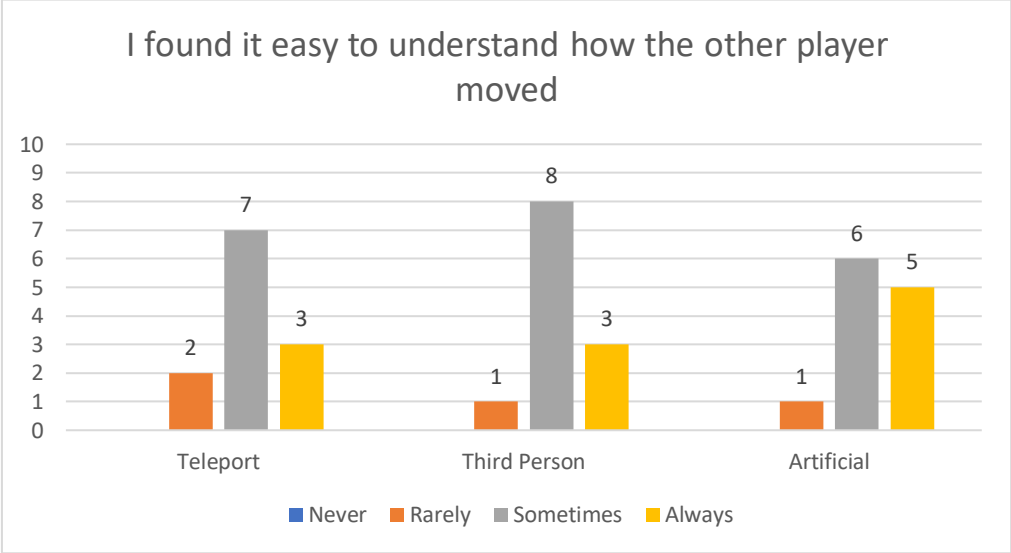
A 1.3 Locomotion

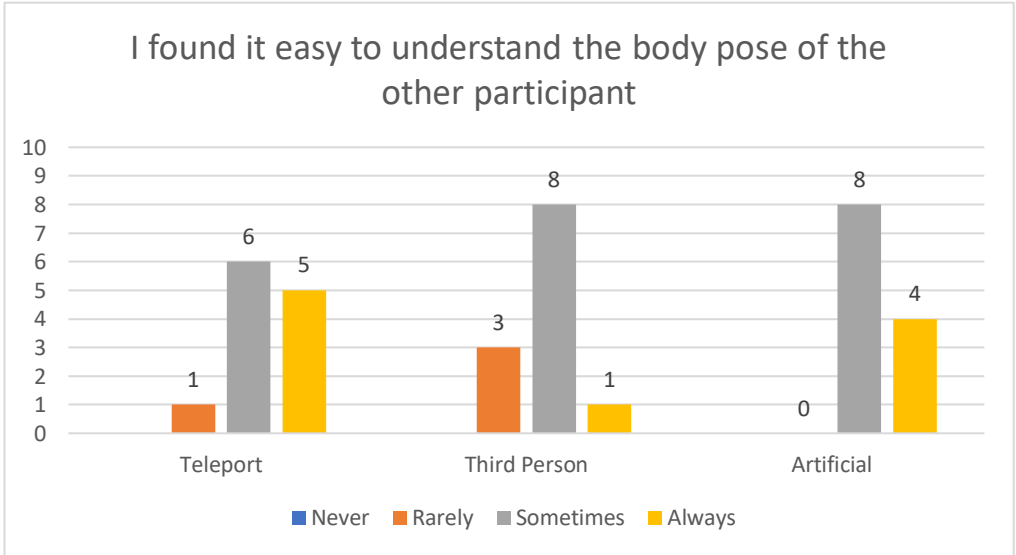




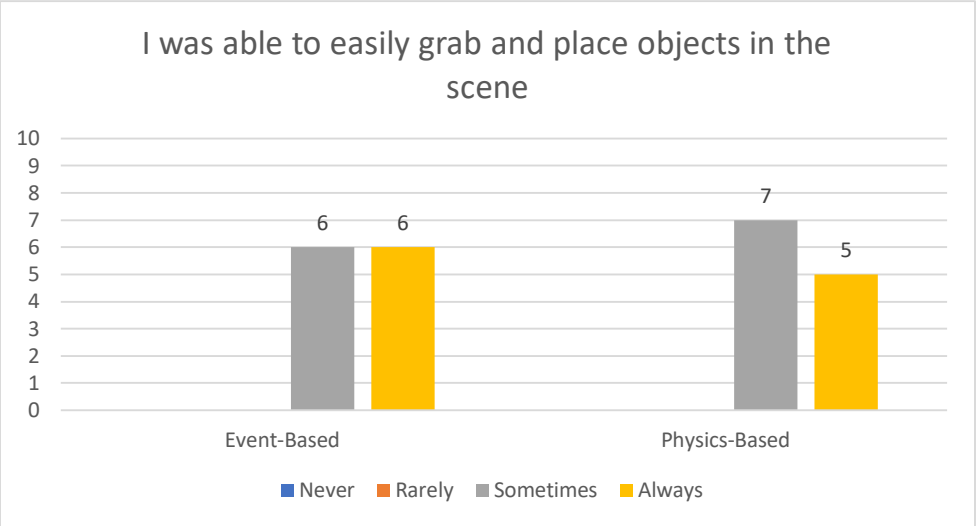


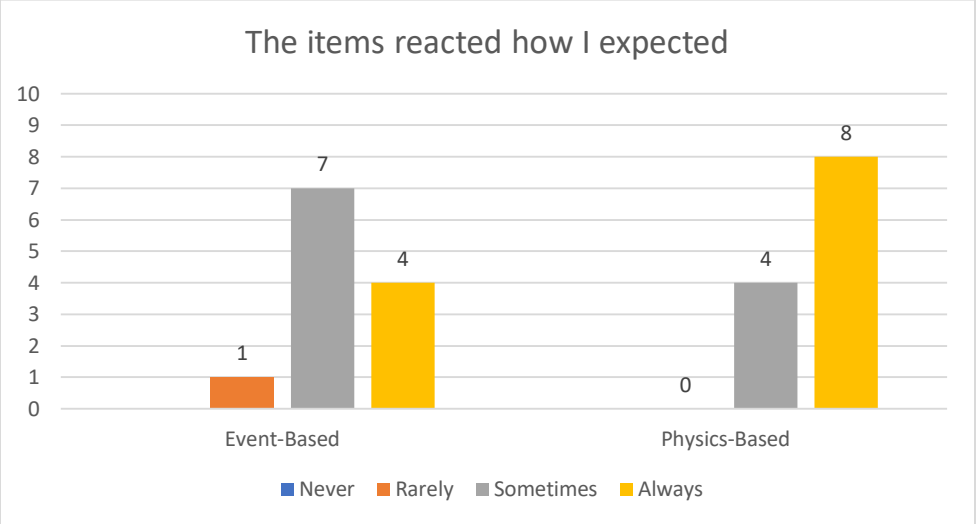

I was able to keep track of the other participant
10
9
8
7
6
5
4
3
2
1
0
2
4
6
2
6
4
3
4
5
Teleport
Third Person
Artificial
Never
Rarely
Sometimes
Always

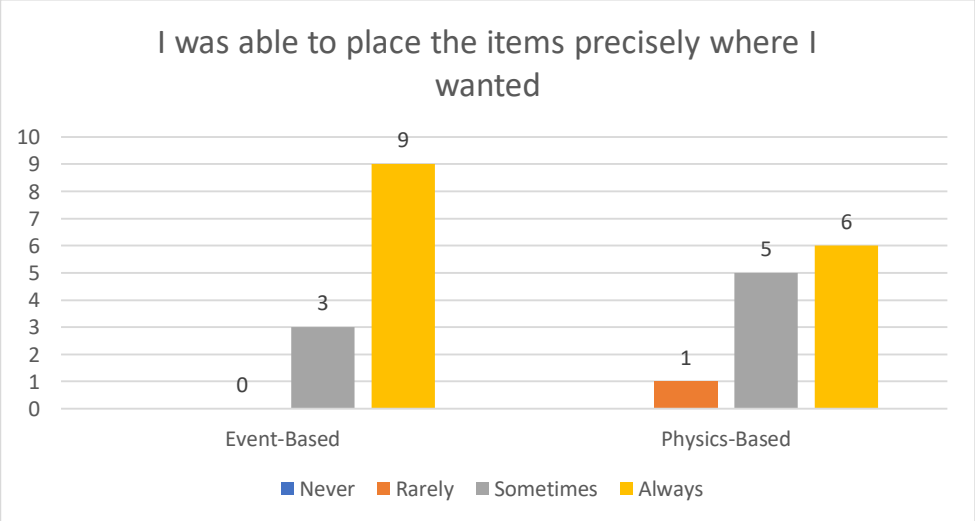

I found it easy to understand how the other player moved
10
9
8
7
6
5
4
3
2
1
0
2
7
3
1
8
3
1
6
5
Teleport
Third Person
Artificial
Never
Rarely
Sometimes
Always

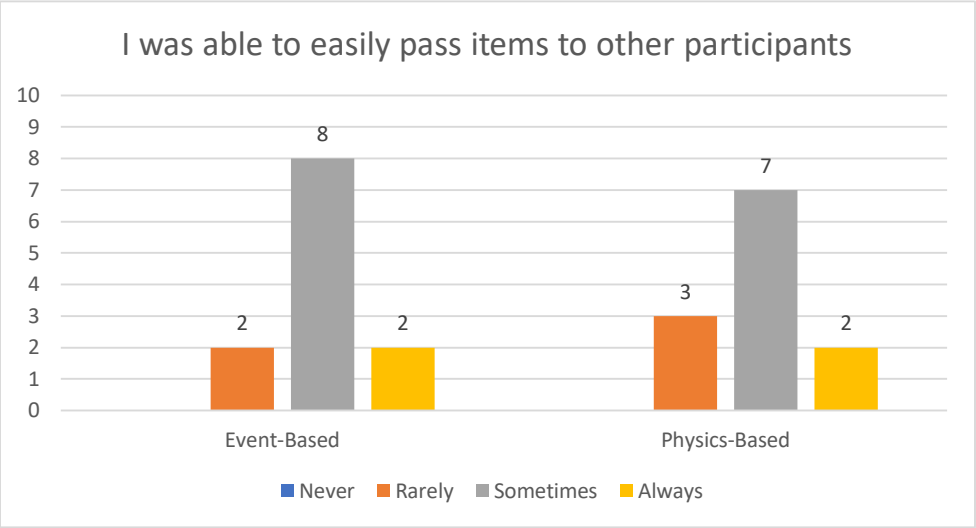

I found it easy to understand the body pose of the other participant
10
9
8
7
6
5
4
3
2
1
0
1
6
5
3
8
1
0
8
4
Teleport
Third Person
Artificial
Never
Rarely
Sometimes
Always

A 1.4 Interaction








I was able to easily pass items to other participants
10
9
8
7
6
5
4
3
2
1
0
2
8
2
3
7
2
Event-Based
Physics-Based
Never
Rarely
Sometimes
Always