Masaryk University
Faculty of Informatics

Virtual reality toolkit for the Unity game engine

Bachelor's Thesis

Vojtěch Juránek

Brno, Spring 2021

Masaryk University
Faculty of Informatics

Virtual reality toolkit for the Unity game engine

Bachelor's Thesis

Vojtěch Juránek

Brno, Spring 2021

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Vojtěch Juránek

Advisor: Mgr. Vojtěch Brůža

Acknowledgements

Firstly, I would like to thank my advisor, Mgr. Vojtěch Brůža for his patience, advice, friendly cooperation, and thorough feedback. Secondly, I would like to express my gratitude to Mgr. David Kuťák for giving me access to his VR toolkit project, and the HCI laboratory at FI MU for letting me borrow a Vive headset for an extended period of time.

Abstract

The thesis deals with the creation of a toolkit that facilitates the development of applications for virtual reality headsets. This toolkit is created in the Unity game engine utilizing the official XR Interaction Toolkit package (XRIT) developed by Unity. The first part analyzes the already existing third-party toolkits, the second part provides an overview of the features offered by XRIT, and the third part describes the design and implementation process of the new toolkit. This new toolkit encompasses various areas of virtual reality development, but primarily contains tools that can be used to create user interfaces. The assets included in this toolkit are designed to be modular and easy to use in other Unity projects. This work also contains a sample project which demonstrates several potential uses of this toolkit.

Keywords

C#, Unity, user interface, UI, virtual reality, VR, human-computer interaction

Contents

Introduction

Virtual reality (VR) is nowadays a rapidly evolving field of human-computer interaction technology. Although primarily used for gaming and entertainment [1, 2], VR is also utilized in education [3], medicine, military, and many other industries. With the release of affordable and easy-to-use VR headsets, such as Oculus Quest 2 or HP Reverb, one may speculate it will be only a matter of time until they are as common as televisions. In 2014, Facebook, Inc. acquired Oculus and has recently released a beta version of Facebook Horizon [4], a VR sandbox video game with social networking elements. Using VR for this purpose might be the next big step in social networking because it allows its users to share the same virtual spaces and communicate more directly. This might increase the popularity of VR even further.

Unity is a popular video game engine created by Unity Software Inc. and is used by industry professionals. The engine has supported development of VR applications since 2014, although most of the VR SDKs were supplied by the headsets' manufacturers. In 2019, Unity released the first version of XR Interaction Toolkit (XRIT). This package, although still in preview[1], provides all the required components for VR development, i.e., device tracking, input handling, and VR interaction architecture.

This bachelor thesis aims to introduce the reader to Unity and XRIT, analyze the existing VR Toolkits and concepts and implement a new one on top of XRIT. This toolkit should provide its future users with useful assets to help them develop their own VR applications.

The text is organized as follows. In Chapter 1, we take a short introduction to the Unity game engine and explain terms related to virtual reality. In Chapter 2, we try to analyze the already existing toolkits and other concepts in VR. In Chapter 3, we try to cover the XR Interaction Toolkit and its architecture. Chapter 4 describes the new VR toolkit and its implementation, and Chapter 5 presents a sample project created with the help of this toolkit.

1. Not yet officially released.

1 An Introduction to Unity and VR

This chapter contains a short summary of the Unity game engine and explains some terms related to VR [5]. It provides readers with minimal experience a basic understanding of the technology and concepts that are referred to later in this thesis.

1.1 Unity's Architecture

One of the most essential terms in Unity is the **GameObject**. A GameObject itself does not provide any functionality and serves as a container for **Components**, which perform specific tasks. Components are mostly isolated, but can access public variables and call public methods from other components. In some cases, a component can even explicitly require another component, meaning that it can not be attached to a GameObject alone. Unity already contains a large number of components; however, most of the interactions between them as well as new features need to be defined and implemented in user-written **scripts**. When a script is attached to a GameObject, it becomes a component. Unity scripts are written in the C# programming language.

A **Prefab** is a saved, pre-configured GameObject. By using prefabs, the developer can place multiple copies of the same GameObject across multiple scenes; or even projects, if the required components are provided as well. If the developer ever decides to modify the prefab, all the instantiated prefabs (GameObjects) in the project get modified as well.

Every GameObject has the `Transform` component, which represents its position, scale, and rotation. It is also important to note that there is a hierarchy between GameObjects (with parent, child, root relations) and the `Transform` is relative to the parent GameObject, which is used to distinguish between local space and global space. Unity's main interface is called the **Editor**, and consists of multiple windows. The scene View window provides the developer a visual representation of the current **scene**. The scene can be interpreted as one game "level" and consists of GameObjects. Another window describing the scene is the Hierarchy window, where the GameObjects

are represented by their names, and the developer can access their hierarchy. The **Inspector** window lists all the components attached to the currently selected GameObject and provides access to their public and serialized parameters. Components can be also manually disabled in the Inspector.

Unity currently supports two input systems; however, the older one is not officially recommended to use and will probably get removed in the future. The new input system, also called the action-based input system, makes input handling more abstract and universal. The system is based on actions stored in Input Action Assets; these actions need to be defined by the developer and can range from "GUI click" to "jump". The developer can then assign hardware and virtual inputs to these actions and subscribe functions to them that act as reactions. A typical example of this would be to create a *jump* action, assign the space and up keys to it, and make a virtual character jump when the action is performed.

Some of the components that will be reffered to later in this thesis are the `MeshFilter`, `MeshRenderer`, and `Animator` components. The `MeshFilter` component takes a 3D mesh and passes it to `MeshRenderer`, which renders it in the scene. `MeshRenderer` also lets the developer assign materials to the mesh and provides additional options for light blending and shadows. The `Animator` is primarily intended to animate GameObjects, but can trigger events as well. It requires an Animation Controller asset, which defines which animation clips to use and the transitions between them. These animation clips can be either created in Unity, or imported as a part of a 3D model file from another software.

1.2 Unity Physics

The Unity Physics system is currently an integration of the open-source PhysX SDK [6] by NVIDIA. Instead of being updated 60 times per second, which is the standard framerate for applications created in Unity, physics simulations are updated 50 times per second by default. This rate can be easily adjusted in Unity's settings to suit the developer's needs, although it is crucial to consider this change. Making the tick rate higher means more precise physics simulations at

the cost of higher computational costs, leading to worse performance, especially in VR. There are two main components in the architecture of Unity that are important for physics simulation; the `Rigidbody` component, and the different types of colliders. There are also some minor components, i.e., joints, that will be mentioned later.

A collider defines the shape of an object for the purpose of detecting collisions. There are primitive shapes, such as the `BoxCollider` and `SphereCollider`, but also `CapsuleCollider`, which is commonly used for more complex-shaped and longer objects that do not require precise collision detection. For more precise collision detection, there is the `MeshCollider` component. If a collider is marked as a trigger, it does not react as a physical object when colliding with other colliders. Instead, it sends messages that can be received by methods that handle the collisions.

The `Rigidbody` component controls the position and rotation of an object through physics simulation. The developer can move the object by applying forces, adjust its mass, change its normal and angular drag, and even constrain the object's position and rotation. Gravity is also applied by this component, but can be disabled.

1.3 Unity and VR

Unity is at the moment the most dominant game engine used for VR development on the market [7]. The first officially supported headset was Oculus Rift [8] in 2014, and since then, Unity has added support for all major VR platforms. This thesis does not address optional, specialized peripherals, such as trackers and hardware-based locomotion systems, even though some are supported by some of the toolkits described in Chapter 2 (e.g., Vive tracker support in SteamVR).

Virtual Reality allows the user to be visually immersed in a fully virtual world with no expected interaction with the real one. The term "VR Headset" can mean two things: the boxed product, which is usually the head-mounted display (HMD), two controllers, and sometimes base stations[1], or simply the HMD alone. It is also important to differentiate between three types of VR headsets. The most affordable headsets contain no electronic components and act only as adapters

1. Used for tracking the controllers and the HMD

for smartphones. Self-contained headsets, such as Oculus Quest 2, do not require to be connected to a PC and do not use base stations. The third type (depicted in Figure 1.1) needs to be connected to a PC and requires at least two base stations. The last type can generally offer more complex applications because it utilizes the PC's processing power; the previous one is far more mobile.

Source: vive.com [9]
Figure 1.1: The Vive Pro headset with two base stations and two controllers. The controllers are often referred to as "Vive Wands".

Augmented Reality (AR) overlays digital data over the real world. This can be done by projecting a digital image onto a lens in front of the user's eyes or by using the camera of a smartphone or a similar device. AR will not be discussed much further in this thesis.

Mixed Reality (XR) combines virtual reality elements and interactions with the real world. This can be achieved by allowing the user to see digital data on top of the real world and tracking the user's hand positions and gestures because controllers can be cumbersome in this situation. This means that the user can use their hands and sight to interact with the virtual world, as well as the real world.

2 VR Toolkits and Concepts

This chapter provides an overview of some of the major toolkits intended for VR development in Unity. Two of these toolkits are provided by companies to accompany their products; one is independent, yet widely used. It also presents some examples of interesting concepts found only in VR applications.

2.1 SteamVR

The SteamVR Unity plugin (SteamVR, for short) was released in 2015 [10] by Valve Software, the same company that (co-)introduced the Vive and Index headsets to market, and the developer of Steam, the most popular video game distribution service on the PC platform. The first official release that uses this toolkit is The Lab [11], a collection of minigames released on April 5, 2016, alongside the Vive headset. The Lab is also a part of the first setup process of the headset, which means it might have been the first VR experience for many of the headset's new owners.

According to the plugin's official documentation [12], the features of SteamVR can be split into four categories: the Render Models system, Skeleton Input and Skeleton Poser, SteamVR Input, and the Interaction System.

Render models simply means that a mesh can be attached to a tracked controller GameObject to represent the controller in virtual space, which is very important for giving the user visual feedback for their real-life hand movements. The underlying component not only can automatically detect the type of the physical controller and selects the appropriate 3D model, but also animates the buttons on the controller model to match their physical counterparts.

The **skeleton input** and **skeleton poser** are two more methods to represent the controllers in virtual space, this time by a human-like hand. Skeleton input tries to determine the poses of the user's fingers by interpreting the current button inputs. Some controllers, for example, the Valve Index controllers and Oculus quest controllers, have touch-sensitive buttons, which allows the system to be more precise, even though the user is not pressing any buttons, and has

only their fingers resting on them. Skeleton poser changes the poses of the user's fingers to one of the pre-determined poses that accords to the current situation, e.g., the fingers copy the currently held object's shape or get clenched into a fist when the user is preparing to punch something in a VR video game.

SteamVR input is very similar to Unity's new input system described in Section 1.1. In addition to the Input Actions system, it also supports the skeleton input system and haptic feedback.

The **interaction system** consists of several scripts and prefabs that were initially created as the basis for The Lab [12]. There are three main concepts of interaction in SteamVR — *hover* means that an interactable in range of a controller and can be interacted with, *attach* that an interactable is attached to a controller and *focus* signifies the last attached interactable because there can be more interactables attached to a single controller.

The architecture of the system can be described as follows:

- **Player** is the object that represents the user. It handles the tracking of the headset and the controllers.
- There are usually two **Hand** objects representing the left and right controller. The hands communicate via messages with interactables that are being hovered by the controllers.
- **Interactable** is the base class of an object that the user can interact with a Hand object. It can receive its messages and react accordingly.
- **ItemPackage** is a special kind of interactable that can be interacted with in different ways while being held. It technically replaces the functionality of the Hand object currently holding this interactable, so the Hand can not hover or grab other interactables.

SteamVR supports teleportation by pointing to a valid spot with a controller and pressing a button. There are two ways to create a valid spot. `TeleportArea` component uses a mesh, for example, a quad, to define a valid area. `TeleportPoint` is more restrictive and allows

teleportation only to a single position; however, every point has a set radius that defines the valid area.

To show an example of an ItemPackage, SteamVR contains a longbow with arrows and shootable targets. Shooting a bow is a very popular interaction in VR, and many games use it as a core mechanic. The longbow is depicted in Figure 2.1, which shows one of the example scenes from the SteamVR package.

Figure 2.1: A screenshot from a SteamVR example scene. Each station demonstrates a different feature and has its own teleport point and label.

2.2 VRTK v3

VRTK, or Virtual Reality Toolkit [13], is an open-source VR toolkit for Unity. Because it does not contain any low-level components, it relies on other Unity packages, such as SteamVR or the Oculus SDK. There are currently two versions available. VRTK v3 is recommended by its developers and is well documented; VRTK v4 is still in beta for more than two years. Because of this reason, the following section describes the features of VRTK v3. It also seems that the focus of the main developers is slowly shifting from VRTK to Tilia [14],

F

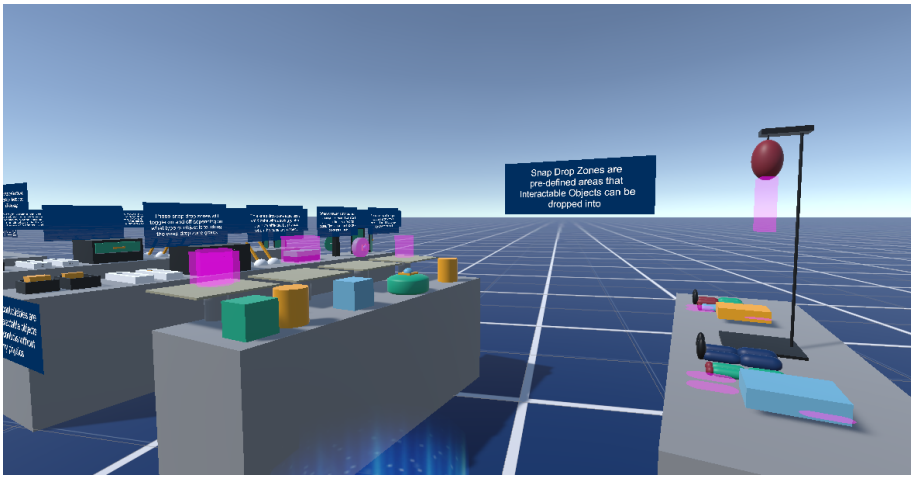


Figure 2.2: VRTK Snap Drop Zone examples. The pink translucent shapes are the triggers that act as the snap zones.

a different project, which splits VRTK into multiple small packages. This allows VR developers to choose only specific features and not to download the whole toolkit.

Because VRTK uses SteamVR as its base, the interaction architecture is very similar. The following is a quick overview of VRTK's most interesting features, which are divided into separate sections.

The **Prefabs** section contains drag-and-drop assets that provide various functionalities. `Console Viewer`, `Frames Per Second Viewer` and `Desktop Camera` can be very useful to debug code. The `Snap Drop Zone` interacts with interactables that are placed nearby. After dropping an interactable inside the zone, the interactable gets instantly attached to the center of the zone. This can be utilized in various ways, and some will be mentioned later. `Radial Menu` and `Panel Menu Controller` provide menus based on Unity's `Canvas` component. Figure 2.2 shows a screenshot from an example scene that demonstrates different kinds of interactions.

Locomotion contains scripts that allow the user to move around the scene. There is the `VRTK_DashTeleport` that builds on top of the teleport functionality by not teleporting the user instantly, but grad-

ually moving closer to the target position at a very high speed. `VRTK_ButtonControl` and `VRTK_TouchpadControl` use the controller's buttons for smooth movement. `VRTK_PlayerClimb` enables climbing, and `VRTK_StepMultiplier` multiplies the distance traveled by each physical step, and thus allowing the user to travel further.

Highlighters can be used to highlight certain objects placed in the scene. There are two ways to accomplish this, either by altering the object's material by using `VRTK_MaterialColorSwapHighlighter`, or by rendering an outline around the object using a component called `VRTK_OutlineObjectCopyHighlighter`.

The scripts from the **Presence** category deal with the user's tracking. `VRTK_BodyPhysics` handles the interactions between the user's body and the scene. It defines the floor, can alter collisions and detects specific kinds of movement in the real world, such as walking and crouching. `VRTK_HipTracking` tries to determine the position and rotation of the user's hips in relation to the HMD. This can be very useful for a belt inventory system. The following two scripts manage collisions of the user's head (when colliding with a wall, for example). `VRTK_HeadsetFade` fades the user's view into black, hopefully forcing them to return back to a valid position. `VRTK_PositionRewind` returns the user to a valid position automatically.

2.3 MRTK

MRTK, or Mixed Reality Toolkit [15], is a Unity package developed by Microsoft that is primarily intended for Mixed Reality headsets; although the package supports many VR headsets as well. The main difference between MRTK and the previous two toolkits is that MRTK supports hand-tracking (without the use of a physical controller), and most of the other features are designed in conjunction with this style of input.

MRTK offers an eye-tracking system as well as a hand tracking system. At this time, most of the VR headsets on the market do not support hand tracking input, with the exception of the Oculus Quest and Quest 2 headsets [16], and the eye-tracking system is supported only by specialized headsets. The only device that supports all of MRTK's features is currently Microsoft Hololens and Hololens 2 [17],

an MR headset intended primarily for professional use. Because of these reasons, it might be better for the developer to consider other solutions and toolkits when developing for other headsets.

The interaction architecture of this toolkit can be described as follows:

- **Controller** represents one of the user's hands, either holding a physical controller or not.
- **Pointers** can be attached to a controller and are used to interact with interactables. There are **Near Pointers**, that use either triggers or short raycasts, **Far Pointers** that are used for interacting at longer distances, and **Teleport Pointers** that are used for teleportation.
- **Focus** determines the interactable that can receive events from a pointer.

There are two main differences between MRTK and the previous two toolkits, the first being the hand and fingers tracking system. This is achieved using the front-facing cameras on some of the headsets trying to interpret the poses of the user's hands [18]. The tracking system uses a joint rig, where joint prefabs are assigned to specific locations on the user's hands and update their positions accordingly at runtime. There are two unique joints, one representing the user's index finger and is used for poking interactions, and the second one is the palm joint. The system can also detect several hand gestures, with the possibility to add more by the developer. Although still an experimental feature, it is possible to attach a rigidbody to each of the fingers' top joints and in this way, simulate the physical presence of the user's hands without the use of the mentioned pointers.

The other difference is MRTK's extensible support for UI, which can be divided into three distinct categories: buttons and sliders, menus, and tooltips.

The prefabs from the **Buttons and Sliders** family allow the developer to build complex user input systems. There are buttons based on Unity's canvas system, i.e., are two-dimensional, and also 3D buttons based on the interactable class, which can react to the index finger

joints described above. Sliders can be used to input numerical values and can be interacted with using the pinch gesture. The included keyboard prefab can be used for alphabetic and numeric input. Figure 2.3 shows a screenshot from the "Buttons" example scene. MRTK has the highest amount of example scenes and many of them even contain valuable information, such as how to create a custom button.

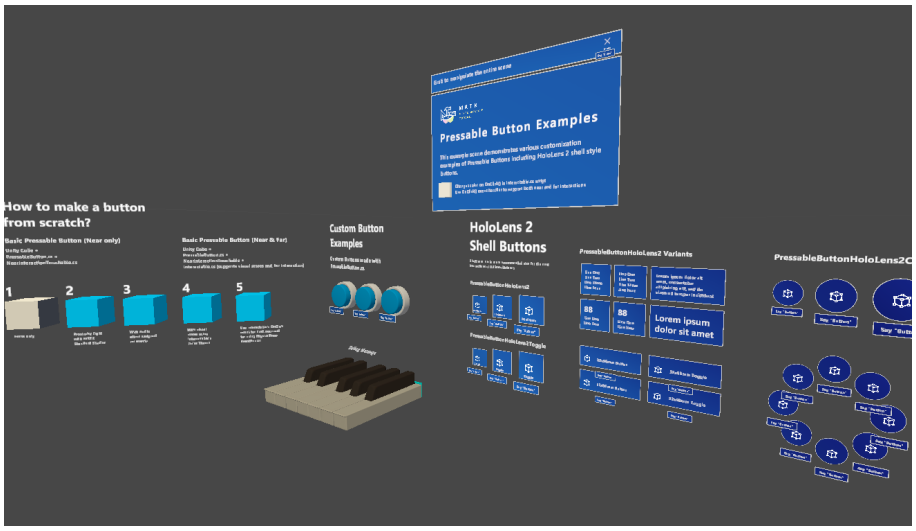


Figure 2.3: Buttons example scene from MRTK. Like the other toolkits, MRTK comes with multiple example scenes, each demonstrating a different category.

MRTK supports various types of **Menus**. There is the `Hand Menu`, which can be attached to the palm joint of one hand and can be interacted with by the other hand. The `Near Menu` has multiple configurations and can be either suspended in the air or made so that it will follow the user with a slight delay.

Tooltips can be attached to objects in a scene. They consist of a billboarding object, usually a descriptive text, and a line drawn from the text to the target object. Tooltips can be used for informative and tutorial purposes.

2.4 VR Concepts

Tomato Presence is a term first used by Owlchemy Labs [19] in an article about their VR game, Job Simulator. In this game, the player is tasked with ordinary jobs, and one of those jobs is a chef. The chef's hand is normally represented by hand-like 3D meshes. However, after picking up a tomato (or any other object), the mesh disappears, and the player's hand is now solely represented by the tomato, hence tomato presence. This concept can be seen in Figure 2.4 and is used in a large number of VR video games and other applications.

Source: Job Simulator
Figure 2.4: A player holding a tomato in their right hand. The position of the hand is perceived through the position of the tomato.

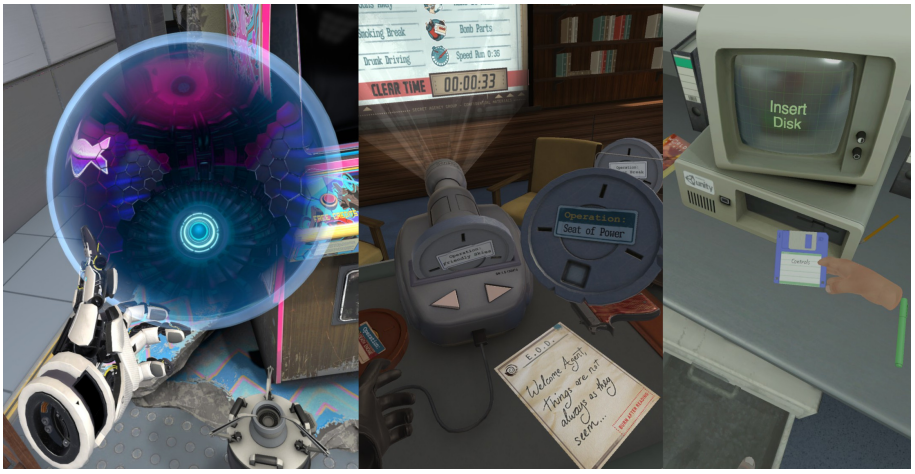
The **Watch menu** works as the main control panel for the player in Rec Room, a sandbox multiplayer VR video game. To show the menu, the player simply needs to hold one of the controllers in front of the headset. After a slight delay, to filter out false requests, the watch menu becomes visible, and the second controller can be used to interact with it. Because Rec Room is a multiplayer game, some privacy

Source: Rec Room
Figure 2.5: A player interacting with the watch menu. The image on the left shows the player's view, the one on the right is the view of another player.

measures are required. The menu is visible only for the player, other users can see only a silhouette — this can be observed in Figure 2.5. The first controller can then be moved away from the headset to close it. This is a very elegant solution because of many reasons, for example, the menu does not have to be assigned to a physical button on a controller, and the player's view is not cluttered by a permanently visible menu, either.

Grabbing and placing objects is the core mechanic of most VR video games. This mechanic is often utilized even in the form of a menu, where different objects represent different choices. The player can then grab one of these objects and put it in a particular position to invoke the option associated with the object. This can be seen in The Lab [11], where the player travels to different locations by moving virtual spheres close to their head. Another two examples come from the video games I Expect You To Die [20] and Surgeon Simulator: ER [21],

where the player can insert tape reels into a projector and diskettes into a computer, respectively. All of these examples can be seen in Figure 2.6.



Source: The Lab, I Expect You To Die, Surgeon Simulator: ER
Figure 2.6: Various examples of menus composed of interactive objects.

3 XR Interaction Toolkit

XR Interaction toolkit is an official package from Unity that provides means for developing VR applications without the use of third-party packages, such as the toolkits described in Chapter 2. The first version of XRIT was released in May of 2019, and since then, XRIT has received several major updates — some of them introduced new features, some have reworked the architecture. At the time of writing, the latest version is `1.0.0-pre.3` from March of 2021.

3.1 Architecture

XRIT's interaction architecture is similar to the architectures from the previous toolkits. There are **Interactors**, that are attached to **Controllers** and can interact with **Interactables**. These interactions are managed by the **Interaction Manager**.

3.1.1 Interactables

Interactables are the objects within the scene the user can interact with. Every interactable can attain one of three states. If an interactable is being *hovered*, an interactor reaches this interactable and is ready to interact with it, *Selected* signifies that user is actively interacting with said interactable, for example, is holding a grab interactable. The *activate* state can be reached only when the interactable is already in the *select* state and is useful for adding functionality to a currently held object. The default button on the controller to change the interactable's state to *selected* is the grip button, and to *activated* the trigger button.

The behavior of an interactable is defined by its callbacks. Every time the interactable changes its state, a sequence of events can get invoked. For example, the *OnHoverEnter* callback signifies that the interactable has started being *hovered*, and *OnHoverExit* the opposite. The developer can subscribe methods to these callbacks either by code, or in the editor. To ability to grab and hold objects is very important in VR, and the `XRGrabInteractable` component offers this feature. After *selecting* the interactable with a controller, the interactable gets attached to the controller (with some exceptions) and copies not only

its position and rotation, but also its velocity, which lets the interactable to be thrown after being *deselected*. One of the vital settings for this component is the tracking mode. There are three options; the first one directly changes the objects transform, the second option moves the object using the required `Rigidbody` component, and the last one uses velocity tracking and physics. Due to Unity's physics system, the first two options do not detect collisions with other colliders while the interactable is *selected*, so the user can pass these interactables through virtual solid objects, for example, walls. To make the interactable collide with other objects, the third option has to be used.

3.1.2 Interactors

Interactors are mostly attached to controllers, with one exception. `XRDirectInteractor` requires a trigger collider, which specifies its reach. Every interactable that collides with this trigger, is being *hovered* and can be *selected*. The `XRRayInteractor` component utilizes ray casting with `LineRender` as its visualization. The ray does not have to be only straight, Bézier and projectile curves are available as well. If the ray cast detects a valid interactable, the interactable is being *hovered* and can be *selected*. There is also the option to use sphere casting, instead of ray casting; sphere casting does not detect only the colliders that the ray directly intersects, but also the colliders in a set radius around the ray. This allows the user not to aim directly at interactables to interact with them, and thus makes indirect aim more forgiving, but could also render the interactor unusable, due the inability to aim at a certain interactable in an area crowded with other interactables.

After *selecting* a grab interactable with the ray interactor, it can react in two ways; either it snaps to the controller, as if it contained a direct interactor, or stay attached to the end of the ray. The object can be then translated along the ray with a joystick on the controller — the ability to move far objects is often referred to as "force grab". `XRSocketInteractor` does not have to be attached to a controller and functions practically the same way as the Snap Drop Zone from VRTK described in 2.1.

3.1.3 Interaction Manager

There has to be at least one `InteractionManager` component in every scene. It acts as an intermediary between the interactors and interactables in the scene, registers new interactables and deregisters removed ones and can force an interactor to *select* an interactable, but not *deselect* it. The lack of this feature seems as an oversight from XRIT's developers, but can be solved by creating a custom interaction manager through inheritance and implementing the appropriate function.

3.1.4 Controllers

Controllers provide tracking and input handling for the physical controllers. There are two types; `DeviceBasedController`, that still uses the old input system and, and `ActionBasedController`, that utilizes the new one. This component also provides the option to attach a 3D model representing the controller in the scene. This model can be either located already in the scene, or as a prefab that gets instantiated whenever the controller is enabled.

Currently, there is a very little support for controller vibration in XRIT. The method providing haptic feedback accepts two arguments, `amplitude` and `duration` and does not work properly with the Vive Wand controllers. No matter what arguments are used, the controller performs only a short and insignificant vibration. This is one of the disadvantages of using XRIT and the only solution at this time is to use a third-party package.

3.2 Locomotion and Presence

The **XR Rig** represents the user in the scene. XRIT provides several pre-made XR Rigs that can be added via a menu into the scene, the main difference between them is the type of input system used by the controllers and whether the rig is stationary (the user has to be standing still or sitting), or room-scale (the user can physically move). The *Camera Offset* GameObject determines the height of the user and *Main Camera* contains the camera and a component that provides tracking for it. An example of the hierarchy of an XR Rig can be seen in Figure 3.1.

Figure 3.1: The default XR Rig after being added into the scene.

XRIT offers various options for locomotion. The room-scale XR Rig allows the user to physically walk in the real world, with the movement being transferred into the virtual one. This system is mostly sufficient for applications with small scenes, but is very limiting for larger ones, due to the size of the real-world room the user is currently in. This problem can be solved by adding controller-operated locomotion, which lets the player travel much further.

`ContinuousMoveProvider` allows the user to move using a joystick (or a touchpad) on one of the controllers. This motion is smooth over time and is relative to the current rotation of the main camera. The component can either translate the XR Rig directly, or use the standard `CharacterController` component which can provide more features. This component is also required by `CharacterControllerDriver`, which lets the user crouch by modifying the height of the capsule collider attached to the XR Rig.

Even though the user can usually physically turn their head to rotate the camera, sometimes it is not possible due to the design of the application, for example, when the user has to be sitting down. `ContinuousTurnProvider` rotates the XR Rig smoothly over time, `SnapTurnProvider` rotates the XR Rig by a set angle value. These components are typically controlled by the joystick on one of the controllers.

The teleportation system consists of multiple components. The `TeleportationProvider` performs the teleportation itself by instantaneously changing the position of the XR Rig. The user can choose this position by pointing with a ray interactor at a teleportation interactable, either a single spot (an anchor) or a collider, commonly a flat mesh.

A valid area can be indicated by displaying a teleportation reticle, which the developer has to provide, at the end of the ray. This reticle is a permanent GameObject in the scene that gets enabled whenever the teleport interactor is active and is hovering a valid interactable.

3.3 AR Support and XR Device Simulator

XRIT also supports the development of augmented reality applications for smartphones and other touchscreen-based devices. The functionality is divided between several types of interactables that react to gestures. Each of these interactables provide a different kind of interaction, such as selection, translation, scaling, and rotation. There are several gestures already included, for example, *tapping* and *dragging*, and more can be added.

The XR Device Simulator system allows the developer to test their application without any VR hardware. It takes advantage of the new input system and rebinds the headset's actions to the mouse and keyboard. The developer can then move the camera and controllers in the scene, rotate them and even perform actions normally bound to the controllers' buttons.

4 Design and Implementation

Before starting the implementation process, I first looked into an already existing VR toolkit created by Mgr. David Kůťák and Mgr. Vojtěch Brůža mainly for the purpose of VR development at FI MUNI. It provides a large number of features and has been successfully used in various academic projects. However, this toolkit still uses the legacy ("old") input system, and many of the features have been developed before the release of XRIT. Some of these features are unique; nevertheless, most of the major elements, such as the interaction architecture and input handling, can be entirely replaced by XRIT. Because of this reason, I have decided to start from the ground up. The only piece of code MUVRE shares with the old toolkit is a script that contains custom `UnityEvent` definitions. Since the new toolkit does not share much with the old one, we have decided to choose a new name, and after some deliberation with my advisor, MUVRE (Masaryk University Virtual Reality Essentials) was chosen. At first, MUVRE was developed in Unity 2020, but was later downgraded to Unity 2019 due to reasons described later in this chapter.

All the 3D models in MUVRE and the example scenes were created using Blender [22], and the textures were retrieved from a website[1] offering PBR materials with CC0 licensing. Most of the 3D models' polycounts are relatively low to maintain better performance on standalone headsets, such as Oculus Quest 2. I tried to use as few textures as possible, so some of the materials use the same normal, base, and roughness maps with only the color of the base map adjusted. Few materials were created using a tool called Quixel Mixer [23], which is currently free to use, that can alter PBR materials and offers more controlled application of said materials to 3D models. Materials created using the software can be seen on the three Vive Wand variants, which will be mentioned in Section 4.3.

1. https://cc0textures.com/

4.1 User Interface Elements

User interface elements [24] are the basis of all menus (described more in subsection 4.2) and provide means of input and control for the user. Unity's UI system already provides most of the common UI elements, and XRIT integrates the system into VR. However, due to the flat nature of these elements and the need for the VR environments to be as immersive as possible, Unity's UI system is not always the ideal option.

To offer an alternative to the native Unity UI system, I have created a set of 3D user interface elements. Creating the 3D representations of the flat elements themselves can be simply done by using Blender; the main challenge is to make them intuitive to use without any loss of expected features. Since the operation of all the 3D elements should not get affected a different scale or rotation, the root GameObject of every UI element prefab is not directly interactable. Most of the required components are attached to a child GameObject, whose local space is not changed by modifying the parent transform. This means that a button is always pushed in the expected direction, even though it is facing downwards, for example.

Highlighting can be used to either attract the user's attention or to signify a state. There are currently two methods implemented for highlighting 3D UI elements or any other objects, and both of these methods can be seen in Figure 4.1. The first one changes the material of the interactive portion of the UI element; the second material needs to be supplied by the developer and should be preferably brighter than the normal material to achieve the highlight effect. The second method highlights only the outline of the desired element. It mostly relies on the free Quick Outline [25] asset from the Unity Asset store. This asset is optimized for VR and offers various ways to draw outlines of chosen color around objects.

4.1.1 Buttons

MUVRE currently contains three types of buttons. They are all derived from the `BaseButton` component, which primarily handles the event invoked by the button after being pressed, the sound it makes and haptic feedback.

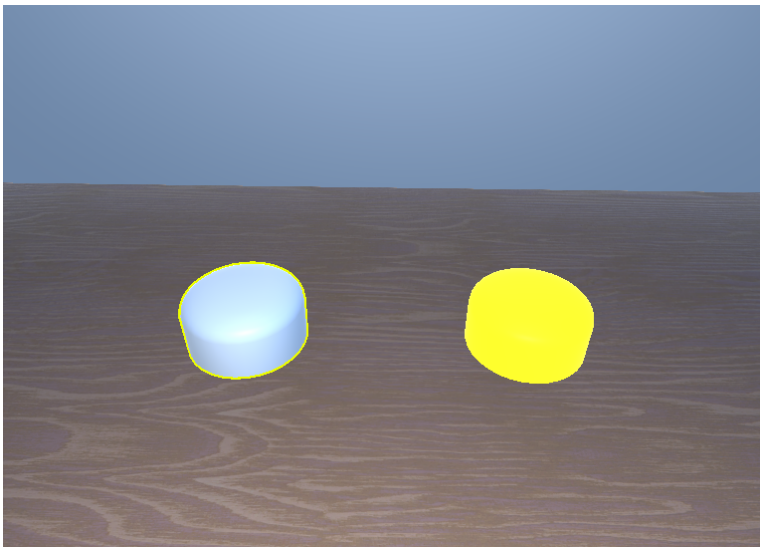


Figure 4.1: The left button is highlighted by an outline, the right by changing its material.

The first type of button, `PushableButton`, is intended primarily for direct interaction. When an interactor is *hovering* over the button and is pushed downwards, the button translates in the same direction. As stated before, all UI elements translate and rotate in local space; therefore, the interactor's position needs to be converted to the button's local space to match the direction of movement of both objects. After pushing the button far enough, the associated event gets invoked and the button cannot be pushed any further. To prevent unintended invokes, the button needs to be let back a percentage of the distance between the starting and the end position.

The `ClickableButton` is intended for direct as well as ray interactors. It does not react to the *hover* state, but rather to the *select* action. After performing the action, the button starts translating downwards along its local y-axis. After traveling a certain distance, the direction changes, and the button starts returning to its initial position. The time this animation takes can be set in the editor. This motion is achieved using linear interpolation of the button's local y-position in relation to time. There is also an option to use the button as a toggle button, which means the button object stays pushed down after the first click

and needs to be clicked again to return. Both of these clicks can invoke different events.

`ClickableButtonAnim` is almost identical to the `ClickableButton` button type, but instead of using translation along the y-axis, it uses Unity's `Animator` component. This allows more diverse animations to be applied to the button, but it also means that the developer needs to supply a valid `AnimatorController` asset with a pre-made animation clip. The toggle option is also present in this button, but requires two animation clips instead of one.

4.1.2 Keyboard

The keyboard is a common way to input alphanumeric values in VR [26] and also a great way to demonstrate the implemented buttons. Every key has one of the components derived from `BaseButton` attached to it, as well as the `KeyboardKey` component, which defines its type. The available types are: character, backspace, enter. Because of the interchangeability of these buttons, many types of keyboards can be created. For example, one keyboard could use `PushableButton` components as its keys and be moveable around the scene (this type can be seen in Figure 4.2); another one could use `ClickableButton` components that react only to a ray interactor and be a part of a static menu.

The `GenerateKeyboard` script allows the developer to create a custom keyboard. The space, enter and backspace keys need to be added manually, but all the other character keys can be generated. The developer has to choose the type of button to be used for the keys, the number of key rows, the horizontal and vertical spacing between the keys, and most importantly, write the values of the keys as strings (one string per row, one character per key). After pressing the "generate" button in the editor, the keyboard gets populated by keys. This keyboard can then be saved as a prefab and utilized later.

4.1.3 Sliders and Knob

Sliders are used to set numeric values. A slider consists of a handle, which can be grabbed and shifted, and a rail that constraints the handle's movement. The developer can specify a range of values, which

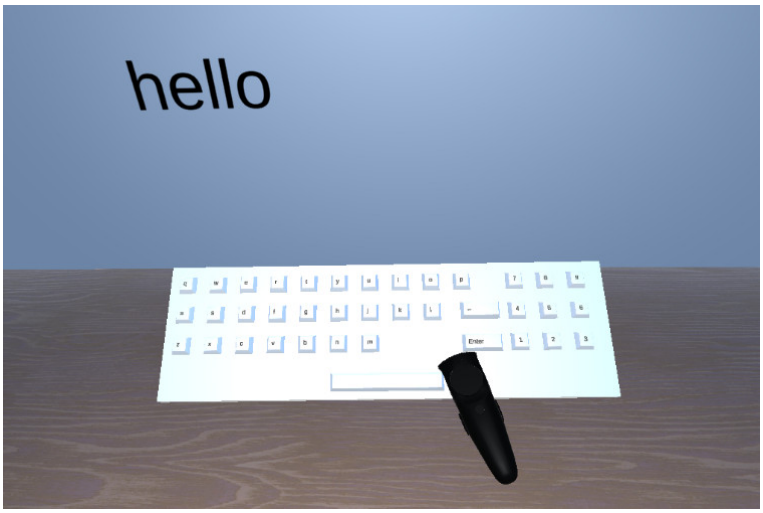


Figure 4.2: A user interacting with a keyboard. The written text is firstly displayed in a text field attached to the keyboard, where it can be still modified.

gets proportionately distributed across the rail. There two types of sliders implemented, the first uses floating-point numbers (continuous), the second integer values (discrete). Operating the first continuous type is straightforward; the user can grab the handle by either a direct or ray interactor and move it along the rail with a continuous motion. The second (discrete) type restricts the position of the handle to a series of pre-determined positions the handle can snap to. For example, if the range is from 0 to 10, there are 11 positions.

To allow intuitive interaction for both the direct and the ray interactor, he implementation has to be different. In the case of direct interactor, the implementation is similar to `PushableButton` — the handle copies the interactor's x-axis position. However, this does not work well with ray interactors. When a user interacts with the handle by a ray interactor, an empty GameObject is created in the same position as the handle, and the same movement constraints are applied. This GameObject is then parented to the current interactor, and the handle is set to follow its position. This secures that the handle reacts

naturally to the position and rotation of the interactor, but also stays on the rail.

Another UI element that can be used to set numeric values is the knob. It does not change its position, but rather the rotation and does so by calculating the angular difference of the current interactor's z-axis rotation between two frames. This angle is then used to rotate the knob and is also added to the knob's current value.

4.2 Menus and Tooltips

Menus [27] are used to provide the user with a set of options that can alter the application. Menus in VR are often diegetic, meaning that they act like real objects in the virtual world and fit its narrative. MUVRE currently contains the wrist-mounted menu, the watch menu, the radial menu, and the panel menu. Tooltips can also be placed within the scene, not to modify it, but to provide information.

4.2.1 Menus

Due to its accessibility and ease of use, the **wrist-mounted menu** [28] is very popular in VR applications and is also quite simple to implement. It does not use any special components, and the most basic version requires only a flat cuboid with UI elements attached to it. This cuboid can be then attached to a GameObject containing the `XRBaseController` component; which makes the menu follow the position and rotation of this controller, allowing the menu to be accessed by the second controller.

The **watch menu**, which was introduced in Section 2.4, shares some similarities with the wrist-mounted menu. It is also attached to a controller, but requires the user to be looking at it to be visible. This feature is implemented by using ray casting (in this instance, sphere casting) and expects a collider to be attached to the user's main camera. Every frame, an invisible, perpendicular ray is casted from the user's wrist that detects the first collider it crosses. If this collider is attached to the same GameObject as the main camera for a set amount of time, the menu becomes visible (depicted in Figure 4.3), and the user can

interact with it. If the raycast does not detect this collider for a set amount of time, the menu disappears.

Figure 4.3: An example of the watch menu. The "wrist" of the user is represented by a 3D watch model (in the center), and the menu fittingly displays the current time.

The **Radial Menu** [29] allows the user to select one out of multiple options. After holding down a button on a controller, a set of options gets displayed around the said controller. While still holding down the button, the user has to *hover* the controller over one of these options and release the button to select it. If no option has been marked, nothing will happen. An example of a radial menu can be seen in Figure 4.4.

Even though VR offers many alternatives, the **overlay menu** [29] (or panel menu) is often irreplaceable. This kind of menu can be brought up anytime by pressing a button on a controller and can be used, for example, to access the application's general settings or to quit the application. Due to its often large size its also easy to see and interact with. In non-VR applications with first-person view, the user loses control of the camera, and the menu obstructs their view. This does not work particularly well in VR; disabling the tracking for the HMD could potentially make the user nauseous [30] and covering their camera view with a menu breaks the immersion. The solution to this problem is not to overlay the menu over the camera, but to display a free-floating panel in the scene. The user can then freely move the camera around the scene, and because the menu is a 3D, diegetic object, the immersion does not break. However, this approach could cause a problem. Because the menu is an object in the scene,

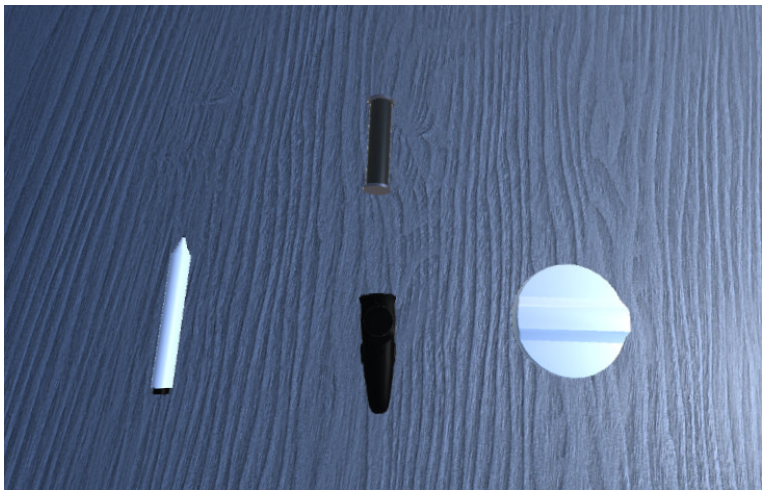


Figure 4.4: A typical use of the radial menu, which lets the user equip one of three items.

it could be obstructed by another object and thus made unusable. To make the menu always fully visible, a secondary camera is attached to the main camera. This camera is then set to the overlay mode and is added to the camera stack of the main camera. Because it is possible to set which objects are rendered by this camera and those objects are always rendered on top of other objects, the menu can never be obstructed.

4.2.2 Tooltips and the Tutorial System

The concept of tooltips has been already explained in Section 2.3 and in MUVRE they work in a similar way. The `LineRenderer` component provides the connecting line, which is then used by MUVRE's `LineConnector` to set the line between two transforms, one being the label; which consists of the `TextMeshPro` component and a white quad as a background for more contrast. The `Canvas` component with an `Image` could be used as well, but because tooltips mostly contain short texts that do not require aligning, the quad approach used instead. To make the tooltip readable from all angles, a simple billboarding effect was implemented, which rotates the label to always face the user's

main camera. There is also the option to use the same approach as for the panel menu, that is, render the tooltip by the overlay camera, which would ensure that the view of the tooltip is unobstructed.

The MUVRE Tutorial System takes advantage of tooltips and demonstrates one of their uses. After entering a new scene or a certain area, the tooltips get sequentially displayed with the line pointing at designated positions. However, to make the text harder to miss for the user, the label is not displayed near the related object in the scene. Instead it gets displayed near one of the controllers with a line pointing towards the related object.

4.3 Controllers and Controller Input

MUVRE currently contains a model of the Oculus Quest controller and three variants of the Vive Wand, all are shown in Figure 4.5. These variants were created to suit the developers' potential needs; one looks very similar to its physical counterpart, the second one simplifies the mesh by omitting the distinctive ring, and the third one has a narrowed end for more precise interactions. All of the buttons are separate meshes so that they can be individually addressed. The `ControllerButtonHighlighter` component utilizes this feature and can highlight individual buttons.

Figure 4.5: Controller models included in MUVRE. From the left: the full Vive Wand, the precise Vive Wand, the short Vive Wand, and the Oculus Quest Controller.

Due to the reasons described in Chapter 3, I have decided to use the action-based input system. An Input Action Asset called `ButtonMaps`

(depicted in Figure 4.6) was created that contains actions bound to all the possible buttons on a VR controller. This means that pressing the primary button on a Vive Wand acts exactly the same as pressing the primary button on an Oculus Quest controller. To process these inputs, the user can either directly subscribe methods to the actions in code or use the new `VRInputManager` component to subscribe to these actions in the editor. Each action in this component has three subscribable callbacks; *performed* is universal, *started* signifies press, and *cancelled* depress.

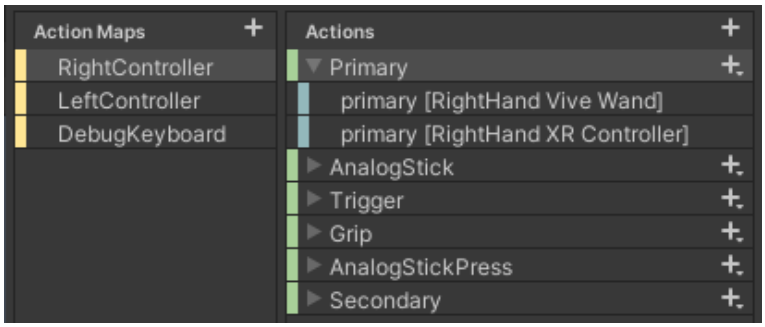


Figure 4.6: The ButtonMaps input action asset. There are separate action maps for the right and left controller. The actions in both of these maps are the same, the only difference being the input bindings.

The input system is the main reason why I had to revert from Unity `2020.3.0` back to Unity `2019.4.16` while developing this toolkit. Unity 2019 still uses the OpenVR [31] API to access the Vive hardware. This was changed in Unity 2020, where OpenVR is no longer supported, and OpenXR [32], an open-source alternative, is used instead. However, it seemed impossible to get any input from the menu button on the Vive Wand controller when using OpenXR version `1.0.14`, and sometimes the OpenXR option has completely disappeared from the `ButtonMaps` asset. This fact makes Unity 2020 partially unusable for potential developers that want to include support for Vive in their projects.

The `ControllerManager` component was created due to the fact that the `ActionBasedController` does not support multiple interactors to be attached to the same GameObject. The component contains

two lists of controllers, each for one hand. After pressing a button, the currently active controller gets disabled, and the next one in the list gets enabled. Switching the controllers this way is especially useful for the teleportation system because it eliminates the need to always have a teleport ray interactor on one of the controllers to be able to teleport. This controller can then be used as a direct interactor when teleportation is not needed.

4.4 Interactables

Interactables are the foundation of every interactive VR application. A new interactable can be created in one of two ways, either by creating a new component that inherits from the `XRBaseInteractable` or `XRGrabInteractable` component and overrides its methods, or by using one of these components in addition to another one, whose methods can be subscribed to the callbacks introduced in Section 3.1.1. Both approaches are used in MUVRE, depending on the complexity of the possible interactions.

4.4.1 Extending the Grab Interactable

`XRGrabInteractableExtended` builds upon `XRGrabInteractable` from XRIT and offers two major new features. The first feature is the ability to hide the controller model of the potential *selecting* interactor by disabling its `MeshRenderer` component and enabling it after the interaction ends. This simple solution basically implements the Tomato Presence concept described in Section 2.4.

The second feature changes the way a grab interactable attaches to an interactor. Both the interactor and the standard interactable have the `attachTransform` parameter, which is by default the GameObject's own transform, but can be changed. After *selecting* a grab interactable, the interactable changes its position and rotation by matching its `attachTransform` to the interactor's `attachTranform`. This implementation means that there is only one way to hold the grab interactable, but also causes a noticeable and abrupt snapping motion. The `XRGrabInteractableExtended` component addresses this by matching the interactor's `attachTranform` to the interactable's `attachTranform`

in advance. Because the transforms are already matched, the grab interactable does not rotate nor snap, and the interaction feels more natural.

4.4.2 Bow and Gun

The **bow** prefab is composed of multiple GameObjects with multiple components. The bow handle is a `XRGrabInteractable` that enables other components after being *selected* to prevent unintended interactions. The string is a `LineRender` with three connected positions. The middle position contains another `XRGrabInteractable` and the line is updated every frame, which makes the string react naturally to pulling. The middle position also contains the `XRSocketInteractor` component, so an arrow can be attached. The socket releases the arrow after pulling the string and releasing it; the arrow is then shot by adding a force pulse to the arrow's `Rigidbody` based on the distance and direction of the pull.

The **gun** prefab works similarly to the bow. After *selecting* and *activating* the interactable, a bullet is instantiated and shot forward. There is also the `Animator` component to animate the gun's mechanical parts and simulate recoil.

4.4.3 Marker and Whiteboard

The use of VR in education is rising, and because a way to write and create freehand illustrations is often required; a marker and whiteboard system was implemented. The marker uses a short raycast to detect whether it is touching a whiteboard object or not. If it is, the raycast hit is then converted into UV coordinates of the whiteboard's mesh. These coordinates are then used to draw a square of set size and color directly on the whiteboard's texture.

Because the marker has to be close to the whiteboard in order to function and is easiest to control when they are directly touching, the velocity tracking option needs to be used on the `XRGrabInteractable` component attached to the marker. This setting prevents the user from accidentally penetrating the whiteboard with it.

This implementation is in no way ideal and serves only as a proof of concept, because, for example, altering the texture every frame

would be very inefficient, especially in an online application, and the square shape could be very limiting.

4.4.4 Spawner

The `Spawner` component inherits from the `XRBaseInteractable` and is used to instantiate prefabs. After *selecting* the `Spawner` with an interactor, a set grab interactable is instantiated and the `InteractionManager` makes the same interactor `select` this interactable. One of the uses for this component is, for example, the archery quiver from Section 4.4.2.

4.5 Locomotion and Presence

Although XRIT already provides a variety of different locomotion options, I have decided to implement more features. The first is a **jump mechanic** for the `ContinuousMoveProvider`, which works by first performing a ground check, and if the check passes, an upward v because this system utilizes the `CharacterCOntroller` component, which does not use the standard physics system, gravity has to be manually applied to complete the jump.

The **climbing mechanic** works in conjunction with the standard `CharacterController` component as well. After *hovering* over and *selecting* a `ClimbInteractable`, the user needs to perform a pulling motion to climb up. When another `ClimbInteractable` is *selected* with the second interactor, the first one can be let go without falling down. The implementation is very simple. Firstly, whenever a `ClimbInteractable` is *selected*, the gravity affecting the user's XR Rig is disabled. Then, the velocity vector of the controller to which the interactor is attached is computed, negated, and applied to the `CharacterController` every tick. This makes the whole XR Rig move to the direction opposite to the pull direction.

The `CameraEffects` component currently provides only a fade-in and fade-out effect, but could be potentially extended later. This effect is currently utilized by the teleportation locomotion system and triggers whenever the user teleports. A potential implementation of this would be a post-processing shader, but a much simpler solution was used instead to make it more accessible for other developers — a

quad attached in front of the user's camera. Various camera effects can be then implemented by changing different aspects of the material assigned to this quad, in this instance by gradually changing the alpha value of a solid black material each frame. Another effect that can be implemented in a similar way is a camera vignette, which could reduce motion sickness [30] caused by continuous movement.

The same fading effect is also used in the `RoomFade` component. This component requires a trigger that encompasses a valid area where the user can move, i.e., a room. After the user moves their head outside the area, the `CameraEffects` component causes their view to fade out and stay faded out, fading in after the user returns inside the area.

The `PlayerBody` component attaches to a GameObject (a "neck") in the hierarchy of the XR Rig's camera and simulates the user's body movement and rotation. It locks the x and z rotational axes and uses linear interpolation to adjust the y axis for more natural-feeling motion. This means that every GameObject attached to this "neck" acts as if it was attached to the user's real body. This component is useful for creating a belt-based inventory system and could also be used to visually represent the user's body if a mesh is attached to the same GameObject.

5 Sample Project

The sample project consists of four scenes, each demonstrating different features MUVRE can offer. To introduce the user to the current scene's mechanics, there is a GameObject with the `Tutorial` component in every scene, and also several labels. To make the tutorial less intrusive, it can be disabled by pressing a button in the Main Room. The project uses mostly the scripts included in MUVRE (different kinds of menus, interactablas, and buttons), but also some new ones, that are going to be introduced in this chapter. To make the scenes more visually appealing, a skybox from the *Skybox Series Free* [33] asset pack was used.

5.1 Main Room

The Main Room is the initial scene of the sample project. Here the user can interact with all the user input UI elements described in Section 4, alongside the keyboard, currently set up to use the `PushableButton` components as its keys. To demonstrate the capabilities of the `XRGrabInteractableExtended` component, there are four grab interactables on one of the tables, each with a different behavior.

There is also a nightstand with two movable drawers. The drawer GameObject contains the `XRGrabInteractableExtended` component to enable interaction and `ConfigurableJoint` to constraint the drawer's movement. The attached `Rigidbody` component allows the drawer to react to physics and can be used to constrain the movement even further to lock it. The `Lock` component, which inherits from XRIT's `SocketInteractor`, will remove these constraints after the user inserts the key lying nearby.

To travel to a different scene, the user needs to place one of three objects into the Selector. This object has a component of the same name attached to it that inherits from the `XRSocketInteractor`. The component can detect which object is inserted into it, and after pressing a nearby button the scene changes to a different one represented by the inserted object.

5.2 Classroom

The classroom (shown in Figure 5.1) demonstrates features from MUVRE that could be used for educational purposes. The scene contains three grab interactables — a laser pointer, a whiteboard marker, and a whiteboard eraser. Both the marker and eraser utilize the `WhiteboardMarker` component and can interact with the whiteboard located in the scene.

The user can also access an item inventory, which is an extension of the `RadialMenu`. The `ClassroomRadialMenu` component checks whether the user is currently holding the whiteboard marker and displays one of two sets of options accordingly. If the user is currently holding the marker, they can change its color, and if not, they can equip one of the three interactables mentioned above.

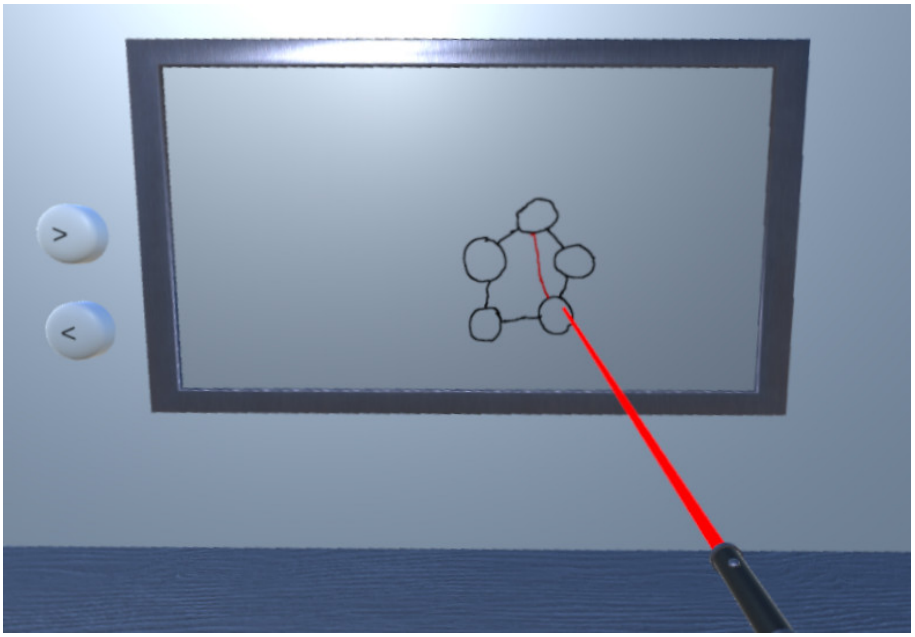


Figure 5.1: A user pointing with the laser pointer at the whiteboard. The two buttons on the left can be used to browse canvases.

5.3 Shooting Range

The shooting range (shown in Figure 5.2) acts as a demonstration of the bow and pistol interactables. To enter the shooting range, the user needs to open a door created using the `XRGrabInteractableExtended` component and `HingeJoint` to simulate the rotating motion of a door. The user's XR Rig contains a GameObject in its hierarchy that has the `PlayerBody` component attached to it. This needs to be done to properly attach a pistol holster (containing the `Socket` component) and an archery quiver (containing the `Spawner` component that instantiates arrows). There are several targets the user can shoot at with either the bow or the pistol. These targets are a part of a minigame, where the user needs to shoot as many as possible within a time limit. The time limit and the position of the targets can be configured with a control panel located on a desk located in front of the targets.

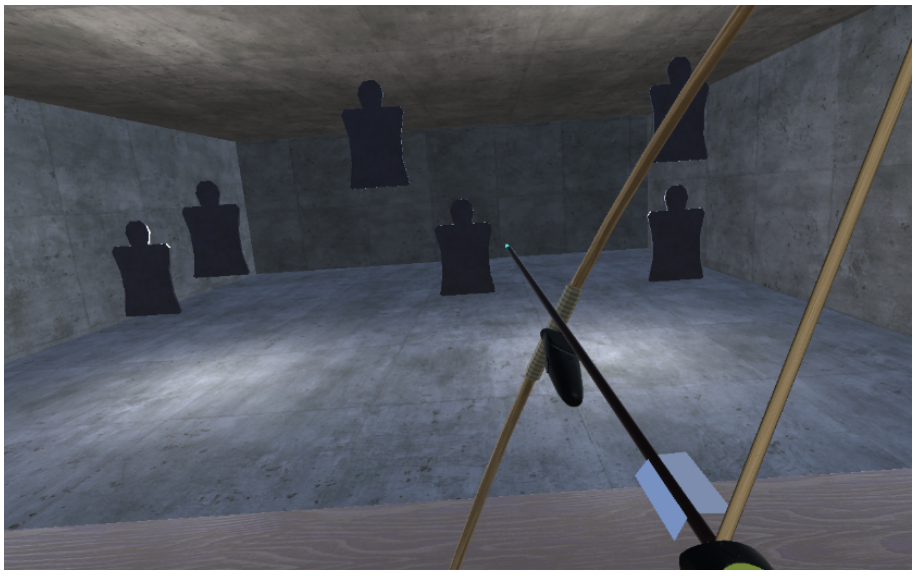


Figure 5.2: A screenshot from the shooting range scene. The player is currently aiming at targets with the bow.

5.4 Obstacle Course

The obstacle course scene demonstrates two different kinds of movement. Instead of the `TeleportationProvider` used in the previous scenes, the user's XR Rig now utilizes `ContinuousMoveController` alongside the `ContinuousMoveProvider`, `CharacterController`, and `CharacterControllerDriver` components. By combining these components, the user can move freely around the scene, jump between platforms and crouch under obstacles. `ClimbProvider` is attached to the rig to let the user use the climbing walls located within the scene. There is also a time trial minigame to make the scene more interesting. The user has to press a button at the start and traverse the obstacle course in the shortest time possible. The scene is depicted in Figure 5.3.

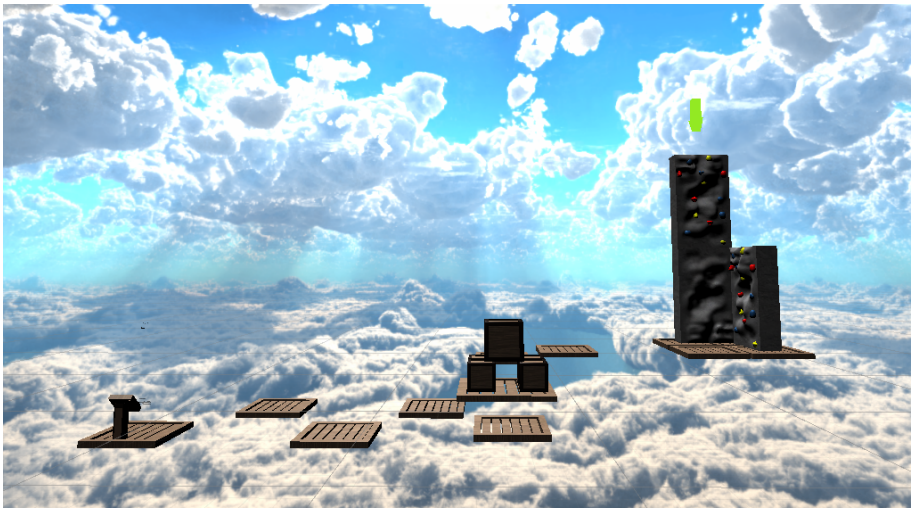


Figure 5.3: The obstacle course scene. The player has jump between wooden platforms, crouch under boxes, and climb up walls to reach the button marked by the green arrow.

6 Conclusion and Future Work

I have analyzed various third-party toolkits intended for VR development in Unity and explained the features of the XR Interaction Toolkit. I have then implemented and described a new toolkit (MUVRE) with XRIT as its base, and created a sample project to demonstrate my work. The toolkit includes various UI elements, such as buttons and sliders, and three types of menus. MUVRE also contains various interactable objects commonly found in other VR applications, improvements to XRIT's locomotion systems, and a new method for handling inputs from VR controllers.

The intent of MUVRE is to provide other developers with tools they can utilize in their own VR applications. This intent has been already fulfilled, because even during its development, MUVRE has been utilized in other projects, such as a VR video game developed by a small team as a part of a private game jam. The toolkit is designed to be easy to use, extendable, and platform-independent. Even though the current version fulfills the requirements specified in the original assignment of this thesis, new features will be added in the future, and the development of MUVRE will continue.

The crucial factor in choosing which features to implement is XRIT's steady development. Due to the fact that the package is still in preview, and Unity developers might add features that may completely replace some in MUVRE; I have decided not to implement significant changes that alter XRIT components, for example, adding the velocity tracking mode for controllers (which is currently available for interactables, and thus will be probably added later).

There are various areas in which MUVRE could be improved, such as the support for online applications. Some of the components can be easily serialized with great performance, and some need to be optimized more to be usable. In comparison to the other toolkits, certain elements of MUVRE are less automated, meaning that they need to be set-up manually in order to function properly. This issue is partially solved by the prefabs included in the project; however, creating a prefab is not always an option.

Bibliography

1. WOHLGENANNT, Isabell; SIMONS, Alexander; STIEGLITZ, Stefan. Virtual Reality. *Business and Information Systems Engineering*. 2020, vol. 62, pp. 455–461. ISSN 18670202. Available from DOI: 10.1007/s12599-020-00658-9.
2. *Virtual Reality Market Share & Trends Report, 2021-2028* [online]. Grand View Research, 2021 [visited on 2021-04-28]. Available from: https://www.grandviewresearch.com/industry-analysis/virtual-reality-vr-market.
3. JENSEN, Lasse; KONRADSEN, Flemming. A review of the use of virtual reality head-mounted displays in education and training. *Education and Information Technologies*. 2018, vol. 23, pp. 1515–1529. ISSN 15737608. Available from DOI: 10.1007/s10639-017-9676-0.
4. HACKL, Cathy. Social VR, Facebook Horizon And The Future Of Social Media Marketing. *Forbes*. 2020. Available also from: https://www.forbes.com/sites/cathyhackl/2020/08/30/social-vr-facebook-horizon--the-future-of-social-media-marketing/.
5. *Demystifying the Virtual Reality Landscape* [online]. Intel [visited on 2021-04-15]. Available from: https://www.intel.com/content/www/us/en/tech-tips-and-tricks/virtual-reality-vs-augmented-reality.html.
6. *Unity Physics Manual Page* [online]. Unity [visited on 2021-05-19]. Available from: https://docs.unity3d.com/Manual/UpgradeGuide5-Physics.html.
7. MARVIN, Rob. How Unity Is Building its Future on AR, VR, and AI. *PCMag*. 2018. Available also from: https://www.pcmag.com/news/how-unity-is-building-its-future-on-ar-vr-and-ai.
8. Calling all VR enthusiasts: target the Oculus Rift with Unity Free [online]. 2014 [visited on 2021-05-22]. Available from: https://blog.unity.com/technology/calling-all-vr-enthusiasts-target-the-oculus-rift-with-unity-free.
9. *VIVE | Discover Virtual Reality Beyond Imagination* [online] [visited on 2021-05-23]. Available from: https://www.vive.com/.

10. *SteamVR Unity Plugin Asset Store* [online]. Unity [visited on 2021-04-14]. Available from: https://assetstore.unity.com/packages/tools/integration/steamvr-plugin-32647.

11. *The Lab on Steam* [online] [visited on 2021-04-18]. Available from: https://store.steampowered.com/app/450390/The_Lab/.

12. *SteamVR Unity Plugin Documentation* [online]. Valve [visited on 2021-04-14]. Available from: https://valvesoftware.github.io/steamvr_unity_plugin.

13. *VRTK* [online] [visited on 2021-04-19]. Available from: https://vrtoolkit.readme.io/.

14. *Tilia* [online] [visited on 2021-04-20]. Available from: https://www.vrtk.io/tilia.html.

15. *MRTK Documentation* [online]. Microsoft [visited on 2021-04-15]. Available from: https://docs.microsoft.com/cs-cz/windows/mixed-reality/mrtk-unity/.

16. *Oculus support* [online]. Oculus [visited on 2021-04-15]. Available from: https://support.oculus.com.

17. NOOR, Ahmed K. The hololens revolution. *Mechanical Engineering*. 2016, vol. 138, pp. 30–35. ISSN 00256501. Available from DOI: 10.1115/1.2016-oct-1.

18. REHG, James M.; KANADE, Takeo. Visual tracking of high DOF articulated structures: An application to human hand tracking. In: Springer Verlag, 1994, vol. 801 LNCS, pp. 35–46. ISBN 9783540579571. ISSN 16113349. Available from DOI: 10.1007/bfb0028333.

19. *Tomato Presence!* [Online]. Owlchemy Labs [visited on 2021-04-29]. Available from: https://owlchemylabs.com/tomatopresence/.

20. *I Expect You To Die on Steam* [online] [visited on 2021-05-23]. Available from: https://store.steampowered.com/app/587430/I_Expect_You_To_Die/.

21. *Surgeon Simulator: Experience Reality on Steam* [online] [visited on 2021-05-23]. Available from: https://store.steampowered.com/app/518920/Surgeon_Simulator_Experience_Reality/.

22. *Blender - a 3D modelling and rendering package* [online] [visited on 2021-05-23]. Available from: https://www.blender.org/.

23. *Quixel Mixer* [online]. Quixel [visited on 2021-05-04]. Available from: https://quixel.com/mixer.

24. *User Interface Elements* [online]. usability.gov [visited on 2021-05-12]. Available from: https://www.usability.gov/how-to-and-tools/methods/user-interface-elements.html.

25. *Quick Outline Asset Store Page* [online] [visited on 2021-05-12]. Available from: https://assetstore.unity.com/packages/tools/particles-effects/quick-outline-115488.

26. BOLETSIS, Costas; KONGSVIK, Stian. Text Input in Virtual Reality: A Preliminary Evaluation of the Drum-Like VR Keyboard. *Technologies*. 2019, vol. 7, p. 31. ISSN 2227-7080. Available from DOI: 10.3390/technologies7020031.

27. PAAP, Kenneth R.; COOKE, Nancy J. *Design of Menus*. Elsevier, 1997. Available from DOI: 10.1016/b978-044481862-1.50090-x.

28. LI, Zhen; CHAN, Joannes; WALTON, Joshua; BENKO, Hrvoje; WIGDOR, Daniel; GLUECK, Michael. Armstrong: An Empirical Examination of Pointing at Non-Dominant Arm-Anchored UIs in Virtual Reality. In: ACM, 2021, pp. 1–14. ISBN 9781450380966. Available from DOI: 10.1145/3411764.3445064.

29. MONTEIRO, Pedro; COELHO, Hugo; GONÇALVES, Guilherme; MELO, Miguel; BESSA, Maximino. Comparison of radial and panel menus in virtual reality. *IEEE Access*. 2019, vol. 7, pp. 116370–116379. ISSN 21693536. Available from DOI: 10.1109/ACCESS.2019.2933055.

30. REGAN, Clare. An investigation into nausea and other side-effects of head-coupled immersive virtual reality. *Virtual Reality*. 1995, vol. 1, pp. 17–31. ISSN 13594338. Available from DOI: 10.1007/BF02009710.

31. *OpenVR Documentation* [online]. Valve [visited on 2021-05-17]. Available from: https://partner.steamgames.com/doc/features/steamvr/openvr.

32. *OpenXR* [online]. The Khronos Group Inc [visited on 2021-05-17]. Available from: https://www.khronos.org/openxr.

33. *Skybox Series Free Asset Store Page* [online] [visited on 2021-05-22]. Available from: `https://assetstore.unity.com/packages/2d/textures-materials/sky/skybox-series-free-103633`.

A Attachments

A.1 Build

The included Windows executable was built in Unity `2019.4.16` and is intended primarily for the Vive headset, although it may work with other headsets as well. Because it utilizes the SteamVR platform, a Steam account[1] is required, and the SteamVR platform package[2] has to be installed.

A.2 Unity Project

The included Unity Project requires Unity `2019.4.16` to function properly. This version can be downloaded in the Unity Hub[3] application. The project can then be imported by clicking the "ADD" button in the Unity Hub and selecting the folder of the project. There is also an HTML document with MUVRE documentation included in this folder.

1. `https://store.steampowered.com/join/`
2. `https://store.steampowered.com/app/250820/SteamVR/`
3. `https://unity3d.com/get-unity/download`