COMP256-002 - Lab 5 - Building Interactive VR Experiences

COMP256 - Special Topics in Gaming

Purpose: Set up a VR Project, explore $XR Interaction Toolkit$, deploy and test the VR Game/app on various platforms/simulators.

Contents

1. Intro

We will: Here, we try to create more complex VR applications, so we can craft immersive and interactive experiences for a wide range of use cases. We'll learn how to use Unity to create interactive VR apps:

- without code, and
- with C#.

1.1. Technical requirements

- We are going to use Unity Hub and Unity Editor 6000.x LTS (my version as of now is 6000.0.33f1 LTS).

1.2. Hardware requirements

- Meta Quest 2 headset
- Availability of a few Meta Links to be able to test the game(s)/app(s) on a PC/Laptop.

For Online classes there is no manadatory requirement for the physical headset (although having one will make a difference in the experienece). The lab(s) can be carried out using the **XR Device Simulator**. This can also help in-person classes to speed up the development.

2. Building interactive VR experiences without code

We have explored the demo scene(s) from VR Core template and XR Interaction Toolit demo scene. Let's build interactive VR experiences ourselves. We'll build a virtual car exhibition. The user should be able to walk around the ground via continuous movement (walking with the joystick) and via teleportation (only to teleportation anchors). Do the following:

- Open the XR project you have worked with before.
- Create a new empty scene named **COMP256_Lab5_{YourInitials}** and open it.
- Delete the **Main Camera**, as it is not needed.
- Create the environment:
  - Add a Cube; name it **ground**; reset it so is in origin); scale it to (20, 0.1, 20). This will form the foundation of our car exhibition.
- Import cars from the Asset Store.
- $Take\ Snapshot$

2.1. Importing cars from the Asset Store

Animating cars in our scene requires the prerequisite step of importing them. Let's walk through the process of importing the simple cars pack from the Unity Asset Store:

1. Go to the Unity Asset Store (https://assetstore.unity.com/). Search for the simple cars pack or, for a more direct route, access the package page via this link: https://assetstore.unity.com/packages/3d/vehicles/land/simple-cars-pack-97669.
2. After adding the package to your assets, find the **Open in Unity** button and click on it.
3. This action should prompt the Unity application to open, presenting you with the option to import the package. Please proceed with the Import operation.
4. Go to **Assets** → **Simple Vehicle Pack** → **Prefabs**.
5. Drop the **taxi** into the scene at **P:(−6,0,0)** and **S:(2,2,2)**.

⚠

If you encounter a magenta hue on the object it implies that the associated materials are missing or incompatible with the current rendering pipeline. To rectify this issue, go through the following steps:

1. **URP Installation**: Before proceeding, ensure you have the URP installed, as it's not included by default in the VR Template project. If it's not installed, go to **Window** → **Package Manager**. Then, search for the **URP** and install it.
2. Go to **Assets** → **Simple Vehicle Pack** → **Materials** → **Cars_1**.
3. With the Cars_1 material selected, alter the Shader drop-down menu in the Inspector window to **URP** → **Lit**.
4. Then, click on the small torus symbol next to Base, Metallic, and Occlusion Map, assigning the corresponding **cars_albedo** to the Base Map, **cars_metallic** to the Metallic Map, and **cars_AO** to the Occlusion Map. This should restore the car's normal appearance.
5. Repeat this process for **Bus_1**, **Bus_2**, and **Cars_2** materials, ensuring the corresponding bus_albedo, bus_albedo_2, bus_metallic, bus _AO, and cars_albedo_2 textures are assigned to the relevant maps.

You may have noticed that making manual adjustments can be quite time-consuming. Fortunately, Unity offers a more streamlined solution for such scenarios. The Render Pipeline Converter is designed to handle bulk conversions of assets and shaders to the URP format. Here's how to use it:

1. Go to **Window** → **Rendering** → **Render Pipeline Converter**.
2. In the opened window, select the materials or shaders you want to convert.
3. Click **Convert**. The tool will then attempt to automatically adjust your selected materials or shaders for compatibility with URP.

You might still need to verify each material to ensure it is converted correctly.

- Add the **Police_car** at **P:(6,0,0)**.
- Place all the vehicles from the asset into our scene.
- $Take\ Snapshot$

Let us now add the **player** and **button prefabs** in the scene.

2.2. Adding the player, teleport anchors, and button to the scene

Before we begin implementing our game logic into the VR experience, we must first add some additional GameObjects and prefabs to our scenes that are just as important for its functionality as the cars themselves. Let's start with adding the player.

2.2.1 Adding the player

- Go to **Assets** → **Samples** → **XR Interaction Toolkit** → **{version}** → **Starter Assets** → **Prefabs**
- Drag and Drop in the scene the **XR Origin (XR Rig)** prefab.
- Reset it to sit right at the center, at coordinates (0,0,0).

Successfully, we've integrated a player into our scene. Now, let's enhance the experience by setting up teleport anchors at various points of interest within our environment.

2.2.2 Setting up two teleport anchors

A way to prevent motion sickness is to restrict movement to teleportation. The XR Interaction Toolkit provides us with three teleportation prefab options:

- Teleport Anchor,
- Snapping Teleport Anchor, and
- Teleportation Area.

Think of **Snapping Teleport Anchor** as a specific point on our stage where the teleportation ray snaps to. In contrast, with **Teleport Anchor**, the teleportation ray will glide without snapping. However, both anchors essentially serve the same purpose: they’re designated spots where users can teleport. On the other hand, **Teleportation Areas** are expansive zones allowing user teleportation within their boundaries.

For this particular scene, we will be using two snapping teleport anchors to ensure users teleport only to specific locations. Here’s how we can set them up:

1. Go to **Assets → Samples → XR Interaction Toolkit → {version} → Starter Assets → DemoSceneAssets → Prefabs → Teleport**. Within this location, you’ll come across the just-described Teleport Anchor, Snapping Teleport Anchor, and Teleportation Area.
2. Drag the **Snapping Teleport Anchor** from the Teleport folder and drop it into the Scene Hierarchy window. Rename this to **Taxi Teleport Anchor**. Position it at coordinates **P:(−2, 0, 0)** and scale it to **S:(2, 1, 2)**.
3. Drag and drop another **Snapping Teleport Anchor** into the Scene Hierarchy window. Rename this one to Police_car Teleport Anchor. Position it at (2, 0, 0) and scale it to the same size as the previous anchor: (2, 1, 2).
4. $Take\,Snapshot$

Next, we’ll enhance the Taxi Teleport Anchor by adding a **push button** directly onto it.

2.2.3 Adding a push button for the taxi animation

To make our push button for the taxi animation stand out and be easily accessible, we’re going to place it on a pedestal created using a cylinder. This cylinder will serve as the button’s dedicated stand:

1. To create this pedestal, right-click in the Scene Hierarchy window and select 3D Object | Cylinder. Rename this cylinder Taxi Button Stand. Adjust its size to dimensions (0.2, 0.5, 0.2) and position it at (−2.7, 0.55, 0).
2. To find the push button, return to Assets | Samples | XR Interaction Toolkit | 2.5.1 | Starter Assets | DemoSceneAssets | Prefabs | Interactables. Once you've found it, drag and drop it onto the Taxi Button Stand in the Scene Hierarchy window. Make sure to scale it (5, 5, 5) and adjust its y-coordinate to 1.075 so it sits correctly on the pedestal.
3. $Take\,Snapshot$

2.2.4 Adding text buttons for police car animations

For our police car, we’ll create a canvas that holds two buttons, enabling users to either shrink or enlarge the police car. Setting up the canvas 1. Begin by creating a canvas: right-click in the hierarchy and select XR | UI Canvas. 2. Adjust its size dimensions. The correct size for our purpose is (0.001, 0.001, 0), making it a square. This scaling is essential for optimal viewing in the VR environment. 3. Position the canvas at coordinates (3,1,-1) and rotate it (0, 90, 0) towards the player’s direction. 4. If you're facing difficulties altering the UI Canvas properties, head to the Inspector window of the canvas component. Here, change the Render Mode from Screen Space – Overlay to World Space. Adjusting the Canvas components 1. With the canvas selected in the Scene Hierarchy, open the Inspector window. Click on Add Component to attach new components to the canvas. 2. First, add a Vertical Layout Group component. This ensures that any buttons you subsequently add will automatically align on top of each other. Within the Vertical Layout Group settings, do the following:

1. Set Spacing to 5.
2. Choose Middle Center for Child Alignment.
3. Check Child Force Expand for both width and height.
4. For visual appeal, add an Image component to the canvas. In the Image component's settings, set the source image to UISprite and choose a background color that complements your scene.

Adding the buttons 1. Navigate to Assets | Samples | XR Interaction Toolkit | 2.5.1 | Starter Assets | Prefabs | DemoSceneAssets | Prefabs | UI. Here, you should find the TextButton prefab. 2. Drag and drop this prefab onto the Canvas twice. This will automatically place two TextButton prefabs as child objects beneath the Canvas. Due to the Vertical Layout Group, they'll be aligned neatly. 3. Scale them to (0.4, 0.4, 0) and rename the first button as Scale Big Button and the second one as Scale Small Button.

2.2.5 Labeling the buttons

- Modify the child text object of each button by unfolding them twice.
  - Select the Text GameObject
  - Search for the text component.
  - Label one button as **Scale Big** and the other as **Scale Small**.
- Notice that by following these steps, you'll have a neatly organized UI ready for your police car animations. Now our stage is set, it's ready for the show.
- $Take\,Snapshot$

Up next, let's wire up the button events.

2.2.6 Interactable events that can be triggered

Interactable events, particularly those found in the XR Simple Interactable component, play a pivotal role in enhancing user experience. Such events, often tied to push buttons, facilitate diverse interactions within VR environments. There are many different options for interactable events. Imagine you are in a virtual reality car exhibition where you can interact with push buttons next to each car. Here's how these events might apply in this scenario:

- First Hover Entered: You approach a car and point at its information button for the first time. This event triggers a subtle glow around the button indicating it is selected.
- Hover Entered: Each time you point at the button, the event renews the glow, showing that the button is currently the focus of your attention.
- Hover Exited: As soon as your hand or pointer moves away from the button, the glow dissipates, indicating that the button is no longer selected.
- Last Hover Exited: The final time you move away from the button, the glow dissipates, and it seems as though the button subtly moves back to its original position, indicating that it's no longer in focus.
- First Select Entered: The first time you press the information button, this event is triggered. It might cause the car's information panel to appear with a smooth animation.
- Select Entered: Each time you press the button, the car's information panel appears, whether it's the first press or not.

- Select Exited: When you release the button, the panel begins to fade, indicating the end of the interaction.
- Last Select Exited: The final time you release the button, it seems as though the panel not only fades but also shrinks back into the button, signifying the interaction’s conclusion.
- Activated: This event could be tied to a button in the car that starts the engine. Upon pressing, you hear the car’s engine roar to life.
- Deactivated: When you press the button to turn off the car’s engine, this event is triggered. You hear the engine wind down to a stop.

This sequence of events provides a natural, intuitive interaction flow for the user, enhancing the immersive feel of the VR car exhibition. In the next section, we’ll delve into how to choreograph these interactions and bring animations to life without code.

2.2.7 Understanding animations and animator systems

Unity’s animation system functions like a mechanism for manipulating GameObjects by controlling their:

- movement,
- rotation,
- size,
- color, and so on.

It works by defining **keyframes**, which are specific states at certain times, and constructing an **animation clip** from these keyframes. Consider the example of a **virtual door**. An animation clip can illustrate the door’s movement from closed to open, and another clip can depict the door returning to its closed position. To ensure smooth transitions between these clips, Unity’s **Animator Controller** is used. This is a control system that directs the timing and sequence of the animation clips, transitioning between them based on specific conditions. For the door example, the Animator Controller would manage the clips for opening and closing the door and transitioning between them at the right moment, possibly based on user input. Here are other examples that can be crafted using Unity’s animation system:

- Object movement: For an object moving across the screen, the animation system can create the movement, but additional scripting may be required to control it based on user interaction
- Object rotation: A constantly spinning gear can be animated, but player-based control of the spin requires code
- Object scaling: Scaling animation can depict an object’s growth, but linking this to other factors, such as energy level, requires scripting
- Color change: An animation can be made for a changing light color, but synchronizing it with game conditions requires coding
- Camera movement: Camera movement between characters can be created using keyframes, but complex interactions with player movement might need additional code
- Blend shapes: Facial expressions can be controlled using blend shapes, but they may need to be scripted to correspond to specific game dialogues
- UI animations: A graphical UI element, such as a flashing button, can be animated, but stopping the animation when certain conditions are met (e.g., a race begins) requires a script

As seen in these examples, the animation system in Unity provides a powerful tool for bringing objects and scenes to life. However, adding scripts or coding to these animations unlocks a higher level of refinement and interactivity. In general, coding enables us to create animations in Unity that can respond to various inputs and game states, making them more dynamic. Coding also provides precise control over the animation, allowing for complex logic and customized interactions that cannot be achieved with Unity’s animation system alone. While Unity’s animation system may not have the same level of complexity and fine control that coding offers, it still holds a significant place as a primary tool for creating interactions within Unity. The visual interface and intuitive controls allow for a rapid and user-friendly way to animate GameObjects, even without delving into scripts. This makes it accessible to both novice and experienced developers, enabling them to add life and movement to their game scenes. In the next section of this lab, we will only be using the animation system, demonstrating how powerful and useful it can be on its own. Through hands-on examples, you'll see how it's possible to achieve engaging and interactive animations without relying on coding and how this tool can become a vital part of your XR development toolkit.

2.2.8 Animating a 360° car rotation

Let's dive into the world of animation. Think of a spinning top: it goes round and round, and we want our taxi to do the same when we push a button. The twist is that when we let go of the button, the taxi should stop spinning. Sound fun? Here's how we'll make it happen using the magic of Unity's animation system and UnityEvents without writing a line of code. First things first, we need to choreograph this spin for our taxi. Imagine it as a pirouette, a 360° turn on the taxi's y-axis. Go through the following steps to bring this dance to life:

1. Right-click in the Assets folder and select Create | Folder.
2. Select the taxi object in your scene and go to Window | Animation | Animation to bring up the Animation window. Click on Create, as if you're penning a new script for our taxi's dance, name it RotateCar, and save it in our newly minted Animations folder.
3. Define a 360° rotation for the y-axis by selecting Transform | Rotation and modifying your keyframes.
4. $Take\ Snapshot$
5. Set the initial keyframes at 0:00 with 0 as the rotation; this is our taxi's starting pose. Then, leap to 0:10 and give the Rotation.y value a 355° spin. Why not 360? That would be like turning on a dime, too abrupt for our gentle spin.
6. Click on the RotateCar animation clip in the project window to view its settings in the Inspector. In the Inspector, give the Loop Time box a check to tell our taxi to keep repeating its spin.
7. Attach an Animator Component to the Taxi and drag the newly created RotateCar animation clip into the Controller field of the Animator component.

This will start our Taxi's twirl right when we step into our virtual world. But remember, we want the spin to start only when we push the button. It's time to make that happen:

1. First, we need to open the Animator Controller. Select the Animator Controller for our Taxi and double-click it to open the Animator Controller window.
2. Like a blank canvas, right-click in the Animator window and select Create State | Empty to create a new empty state. Name this state Idle; it's the calm before the Taxi's spinning storm.
3. Next, right-click on the Idle state and set it as the Layer Default State. This tells the Animator Controller to start in this state, doing nothing when the scene begins.
4. We then make a bridge from the Idle state to the RotateCar state: the state that initiates our taxi's spin. To do this, right-click on Idle, choose Make Transition, and then click on RotateCar.
5. Now let's navigate to the Parameters tab and add a parameter named Rotate. This parameter will initiate the transition from the Idle state to the RotateCar state.
6. Click on the arrow representing the transition from Idle to RotateCar. In the Inspector window, under Conditions, add a new condition. Choose the Rotate parameter that we just created.
7. $Take\ Snapshot$
8. Next, we need to halt our taxi's spin once the button is released. Right-click on the RotateCar state, choose Make Transition, and then click on the Idle state.
9. Go back to the Parameters tab and this time add a parameter named StopRotation. This will initiate the transition from the RotateCar state back to Idle.
10. $Take\ Snapshot$
11. Click on the transition arrow that goes from the RotateCar to the Idle state this time. Under Conditions in the Inspector, add a new condition and select the StopRotation parameter as we

did with the Rotate parameter earlier.

12. $Take\ Snapshot$ of your Animator window,
13. Select the Taxi push button in the hierarchy and look for the Select Entered and Select Exited events. Click on the + and drag the taxi into the Runtime Only field. Next, we are calling for Animator.SetTrigger(). Think of it as the cue for our Taxi to start or stop its dance. When the button is pressed (Select Entered event), the cue is Rotate, and when the button is released (Select Exited event), the cue is StopRotation.
14. $Take\ Snapshot$
15. Playtest it. Your taxi is now set to dance to your button's beats, spinning a full circle each time you press it. Don't forget to try it out: give that button a good press and watch the taxi perform its 360° spin.

2.2.9 Scaling a police car

Imagine the compelling possibilities of incorporating two additional interactive features in our VR auto show: one button to miniaturize the car and another to magnify it. Similar to our earlier interactions, we will employ a set of procedures, but this time with 2D buttons on a canvas. First, we create two animation clips that capture the essence of the car's transformation as it scales up and down. Next, we create an Animator Controller: a crucial component that defines and manages the animation states. Lastly, we assign the corresponding trigger events to the button. Let's bring this magic to life with the following steps:

1. Select the police car in the hierarchy. Open the Animation window (Window | Animation | Animation) and create a new animation clip. Name it ScaleCar and store it in the Animations folder.
2. Now, let's choreograph the transformation. On the timeline in the Animation window, mark the beginning (0:00) and the end (1:00) with keyframes (the diamond icon). At the beginning, the scale of the car (Transform | Scale) should be at its smallest size, 0. At the end, the scale should be at its largest size; let's go with 3. This gives us an animation that sees the car scaling from minuscule to massive.
3. We don't want our car continuously growing and shrinking. So, in the Inspector, uncheck the Loop Time box.
4. Our police car needs an Animator Component to manage its transformation. Attach one if it hasn't been done automatically. Then, slot the ScaleCar animation clip into the Controller field of the Animator Component.
5. We've got the growing part down, now let's work on the shrinking. Create a second animation clip named ShrinkCar.
6. Similar to step 2, mark the beginning and the end with keyframes. This time, at the beginning, the scale of the car should be at its largest size (3). At the end, the scale should be at its smallest size (0). We've now reversed the transformation, creating an animation that shrinks the car from massive to minuscule.

By now, we've successfully created two animation clips and an Animator Controller. If we were to play the scene, the police car would scale up once thanks to the unchecked loop checkbox and the absence of the downscaling animation in our Animation Controller. We're one step closer to our transforming car. Ready for the next part? Let's dive in. Now, it's time to make our police car dance to our tune. We want it to change sizes when we wish and be idle when we don't. That's where the Animator Controller comes in. It's our maestro, dictating when the car grows, shrinks, or takes a breather. Here's how we bring this to life:

1. First, we need to open the Animator Controller. Locate the one assigned to the police car in the project window and double-click to open it.
2. Remember our shrinking dance? Drag and drop the ShrinkCar animation clip into the Animator Controller window. It now has both the growing and shrinking routines.

3. Now, we need a state of rest, a breather in between our dance routines. Right-click in the Animator Controller window, select Create State | Empty and name it Idle. It's like the calm before the storm.
4. We don't want our car to start dancing right away when the curtain rises. So, set the Idle state as the default state. This ensures that when the scene begins, our car stands still, awaiting its cue.
5. Our maestro is ready to conduct. Right-click on the Idle state and choose Make Transition. Then, click on the ScaleCar and another time to the ShrinkCar states. Now, we've set the stage for our car to transition from standing idle to growing big and shrinking small.
6. But what cues the transitions? We need to set trigger parameters for this. In the Animator window, click on the Parameters tab and add two new trigger parameters:
7. Name them ScaleBig and ScaleSmall.
8. Select the transition arrows. Under Conditions in the Inspector window, add a new condition for each of these triggers.

ScaleBig cues the transition from Idle to ScaleCar, while ScaleSmall cues the transition from Idle to ShrinkCar. The dance is never one-way. Our car needs to move smoothly from growing to shrinking and vice versa. So, create transitions from ShrinkCar to ScaleCar and the other way around. Click on the transition arrows. Under Conditions, add the ScaleBig trigger for the lower and the ScaleSmall trigger for the upper transition arrow. - $Take Snapshot$ - Try out the scene and watch as it gracefully grows and shrinks at your will by testing or deploying the scene onto your VR headset.

3. Building interactive VR experiences with C#

So far, we have added interactions to our scene just by using Unity's Animation system without writing a single line of code. In the next sections, you will learn how you can use scripting with C# to add even more complex breaths of air into your GameObjects and scene. The first question we are going to answer is this: when do we need to write C# code for our animations and interactions?

3.1. Understanding when to use C# for animations and interactions

It's important to understand that Unity's animation and animator systems and the use of C# scripting are not mutually exclusive. They are often used together, with the animator controlling predefined animations and C# adding interactivity based on user input or other game events. In the previous section, we rotated and scaled cars based using Unity's animation and animator systems. These systems are primarily used to create predefined animations. We can divide this process into the following three steps:

1. Initialization event: First, an event must occur to start the animation. This can be a button press, game event, collision, or any other trigger. In our example, to scale and rotate the cars, we used a physical push button and a 2D UI button. Unity's built-in animation and animator systems are largely designed around the idea of predefined animations that are triggered under specific conditions. These conditions are typically defined within the Animator Controller itself and are usually based on parameters that you set up ahead of time, such as a boolean to track if a character is jumping or a trigger that gets activated when a button is pressed. This means that, with the built-in system, we do not have the ability to trigger animations based on virtually any event or condition in our game and have to work with the available events.
2. Animation: Using Unity's animation system, we define what changes during the animation (position, rotation, scale, etc.) by creating keyframes. For instance, we changed the rotation and scale of the cars with keyframes.

3. Animator Controller: This is where you manage the different animations and transitions between them. You create states, each linked to a different animation, and define the conditions under which transitions occur. For example, we used a button press that initialized a transition of the car from a ShrinkCar state to a ScaleCar state.

This system is perfect for creating animations that are predefined and occur under specific conditions; for instance, character animations (walking, running, jumping) or environmental animations (door opening, elevator moving). While Unity's built-in system excels at predefined animations, C# comes into play when animations need to respond dynamically to user input or other game events. Here, we follow similar steps as before:

1. Initialization event: The event that starts an animation can be anything, such as button presses, user input, or changes in the game state. The biggest difference with Unity's built-in system is that with C# scripts, you have much greater flexibility and control over the initialization events for your animations. For instance, you can respond to complex sequences of input, such as a fighting game action combo. You can base animations on game logic or game state, such as an enemy's health level or the player's current score. You can trigger animations based on collisions, entering/exiting certain zones, or other physics events.
2. AI Decisions: You can initiate animations based on the decisions made by an AI system. In a multiplayer game, you can trigger animations based on network events, such as another player's actions.
3. Animation: Animations can be created in two ways:
    - You can use the animation system to create keyframes as before and then use C# to control the playback of these animations (for example, play, pause, stop, or alter speed).
    - Alternatively, you can use C# to modify the properties of GameObjects directly, creating animations programmatically.
4. State control: With C#, you gain more direct control over when and how animations and transitions occur. You can create conditions based on any aspect of your game's state, not just parameters in the Animator Controller. For example, you could change an NPC's animation based on the player's health or inventory.

Using C# for animations is ideal when the animation needs to respond in complex ways to the game state or user input. This is especially the case in the following scenarios:

- Real-time user input: You often use C# to animate GameObjects in response to real-time user input. For example, in a flight simulator, you might animate the plane's control surfaces (such as the ailerons, elevators, and rudder) based on the player's joystick input. Since the exact position of these surfaces depends on the player's input, it's not something that can be predetermined with keyframes.
- Physics-based animations: If your animation needs to incorporate or respond to physics, it often makes sense to animate it programmatically. For example, in a pool game, the balls move and spin based on physics calculations rather than predefined paths.
- Procedurally generated content: When the content of your game is procedurally generated, you often need to animate things programmatically because the exact nature of the animations can't be predetermined. For instance, in a rogue-like dungeon crawler, the layout of the dungeon and the placement of enemies are generated on the fly, so any animations related to these elements would also need to be generated at runtime.
- Complex AI behavior: When creating complex AI behavior, you might use C# to animate GameObjects based on the AI's decision-making processes. For instance, an enemy character might have an idle animation, a walk animation, and an attack animation, and you can use C# to decide which one to play based on the AI's current state and the player's position.

Now that we have understood the scenarios in which it is useful to use C# for animations and interactions, let's put our new knowledge to use by scaling a car using a slider in the following section. This would fall under the "real-time user input" category. When we want to scale a car using a slider, it's the user input (moving the slider) that's driving the animation (the scaling of the car). This kind of interaction can't be predefined with keyframes because the exact scale of the car depends on the player's input at any given moment. But before we add these to our scene, we first need to understand the very basics of the C# language, which are explained in the following section.

3.2. Understanding scripting with C# in Unity

Creating scripts for VR development in Unity involves using C#. As a high-level language, it simplifies many computing complexities, making it user-friendly compared to languages such as C++. Through its strong, static typing, C# helps you spot programming errors before running a game in Unity, which is incredibly helpful for developers. Additionally, it's good at managing memory use, as it minimizes the risk of memory leaks—a situation where a game eats up an increasing amount of memory, potentially causing crashes. C# is supported by Microsoft, which ensures you get reliable help, plenty of tools to work with, and access to a big community of other developers. And, importantly, just like Unity, C# works across many platforms. When using C# in Unity, your coding is mostly event-based. This means you override certain Unity functions that get triggered at specific times, such as the Start() or Update() functions of a game. Now, when it comes to using C# for VR development in Unity, there are three key object-oriented programming concepts you'll need to understand: variables, functions, and classes. Let's get to know these a bit better. Variables Variables are like storage boxes that your script uses to hold data. Every variable has a type, which tells you what kind of data it can hold. For example, if you're using the XR Interaction Toolkit, you might have variables to store things such as the position of a VR controller, the state of an object in the virtual world, and so on. Let's see an example of how we can use variables in the context of a C# script for Unity with the following code snippet:

```
public class XRGrab : MonoBehaviour
{
    public XRGrabInteractable grabInteractable;
    private bool isGrabbed = false;
}
```

In the preceding code, grabInteractable is an XRGrabInteractable object that represents a VR object that can be interacted with. isGrabbed is a private Boolean variable tracking whether the object is currently grabbed or not.

3.3. Functions

Think of functions (or methods) in C# as cooking recipes. Just like a recipe provides step-by-step instructions to cook a specific dish, a function in C# consists of a set of instructions that performs a specific task. You can reuse these recipes multiple times, either within the same script or across different scripts. In Unity, there are some special functions, such as unique cooking recipes, that are triggered at specific times during the life cycle of a script. Here is an overview of them:

- Awake(): This is like an alarm clock for your script. This function rings when the script first wakes up (or loads). Often, it's used to set up variables or the state of the game.
- Start(): This is the runner on their mark, ready to start the race. This function gets called right before the first frame of the game is displayed. It's also used to set things up, but unlike Awake(), it won't run if the script isn't enabled.
- Update(): This function is the heart of your game, beating once per frame. It's usually used for tasks that need to happen regularly, such as moving objects around, checking for user input, and so on.
- FixedUpdate(): This is like Update(), but it runs at a consistent pace, no matter the frame rate. It's typically used for physics-related tasks.
- LateUpdate(): This is the function that tidies up after everyone else. It runs after all Update() functions have done their thing, doing any tasks that need to happen after everything else.

- OnEnable(): This method is a special Unity method that gets called whenever the object this script is attached to becomes active in the game.

Now, when it comes to VR interactions, Unity's XR Interaction Toolkit provides its own set of special functions that react to VR actions. Here are a few important ones you'll want to know about:

- OnSelectEntered(): This function is called when you select an interactable object in VR. Imagine it as the moment when you point at a virtual object with your VR controller.
- OnSelectExited(): This function is the moment when you stop selecting an interactable object. Think of it as letting go of the object you were pointing at.
- OnActivate(): This is when you activate an interactable object. It's a bit like pressing a button while you're already holding the object.
- OnDeactivate(): This is when you deactivate the object. It's like letting go of the button you just pressed.
- OnHoverEntered(): This function gets called when you start hovering over an interactable object. This could be used to make the object light up, for instance.
- OnHoverExited(): This function is when you stop hovering over an object.

These functions can be combined in different ways to create all sorts of interactions in the virtual world. It's like stacking building blocks together to create something more complex. Classes Classes in C# are blueprints for creating objects. A class can contain fields (variables), methods (functions), and other members. All scripts in Unity are classes that inherit from a base class such as MonoBehaviour, ScriptableObject, or others to illustrate the characteristics of classes in C#. Let's go through an example class called XRGrab, which inherits from the base class, MonoBehaviour. The first section of this class consists of the following:

```
public class XRGrab : MonoBehaviour
{
    public XRGrabInteractable grabInteractable;
    private bool isGrabbed = false;
```

In the preceding example, the XRGrab class consists of two fields. public XRGrabInteractable grabInteractable is a reference to an XRGrabInteractable object. This script is typically attached to a GameObject that you want to make interactable in a VR or AR setting; for example, an object the user should be able to grab. isGrabbed is a boolean variable that keeps track of whether the interactable object is currently grabbed. The next section of our class consists of methods. The first method of our class is called OnEnable():

```
void OnEnable()
{
  grabInteractable.onSelectEntered.AddListener(Grabbed);
   grabInteractable.onSelectExited.AddListener(Released);
}
```

The OnEnable() method adds Grabbed() and Released() methods as listeners to the onSelectEntered and onSelectExited events of the grabInteractable object. This means when the user interacts with the object in the VR/AR world by selecting it, the Grabbed() method will be called, and when the user stops interacting with it and hence deselects it, the Released() method will be called. Let's have a look at what both of these methods might look like:

- Here's the Grabbed() method:

```
    void Grabbed(SelectEnterEventArgs args)
    {
        isGrabbed = true;
    }
```

The Grabbed() method sets isGrabbed to true, indicating that the object has been grabbed. - Here's the Released() method:

```
    void Released(SelectExitEventArgs args)
    {
        isGrabbed = false;
    }
}
```

The Released() method sets isGrabbed to false, indicating that the object has been released. As you can see, scripting in Unity using C# involves defining variables to hold data, implementing functions to manipulate that data or implement gameplay, and organizing these variables and functions into classes that represent objects or concepts in your game. The Unity engine then uses these scripts to drive the behavior of GameObjects within your scenes. In the realm of C#, class names typically follow the PascalCase convention, where each word begins with an uppercase letter and underscores are absent. You have the freedom to select any name that pleases you, as long as it complies with the general naming conventions of C#. Here are the naming guidelines:

- Names should commence with an uppercase letter instead of an underscore or a digit.
- Names can incorporate letters and digits but not underscores.
- Spaces or special characters should not be part of the name.
- Lastly, the name must not be a reserved word in C#. Words such as “class”, “int”, “void”, and so on are reserved and cannot be employed as a name. This is because they have specific meanings in C#, as they are part of its syntax. The compiler expects them to be used in specific ways, so using them as identifiers would cause confusion and lead to compilation errors.

And that’s a gentle introduction to scripting in Unity for VR development. Don’t worry if it still sounds a bit complex; like with any language, practice makes perfect, and this is exactly what you are going to do in the next section! Scaling a bus using a slider and C# We are about to embark on an exciting task: enriching our car exhibition experience by adding a dynamic scaling feature to a bus. Previously, we used fixed animations to scale a police car. However, this time, we will scale a bus in real time based on user interaction with a slider. With these steps, we will achieve our objective. So, let’s dive in:

1. First, we need to bring the bus into our scene. Go to Assets | Simple Vehicle Pack | Prefabs, drag and drop Bus_2 into the scene. Position it at coordinates (0,0,-6), with a −90° rotation on the y-axis.
2. Next, we require a slider and a descriptive text that indicates that the bus can be scaled using the slider. To accomplish this, we first need to create a canvas. Right-click in the hierarchy and select XR | UI Canvas, then resize it to (0.01,0.01,0), position it at (0,1.5,-5), and rotate it to (0,180,0). Now it is facing the player’s direction. Note that you must set the Render Mode of the Canvas component to World Space to manipulate its properties. Let’s rename the canvas as Bus Scale Canvas for clarity.

One thing is missing before adding UI elements to the canvas: the Vertical Layout Group component. Let’s add this in the Inspector and change the Child Alignment property to the Upper Center. This ensures that both the slider and text will be vertically aligned at the upper center of the canvas. 3. Next, we can add the slider. Right-click on the Bus Scale Canvas and select UI | Slider. This will add a slider as a child object of the canvas. The slider consists of several child objects that collectively form the slider’s visual and functional features. However, our focus is primarily on the slider object itself. In its Slider component in the Inspector, ensure that the Min Value is 0 and the Max Value is 5, allowing the bus to be scaled between these values. Let’s rename the slider to Bus Scale Slider for distinction. 4. For enhanced user experience, we will also place a text above the slider. Add a Text – TextMeshPro (TMP) child object to the Slider GameObject by right-clicking on Bus Scale Canvas and selecting UI | Text – TextMeshPro (TMP). Download the required TMP assets if prompted. Change Text Input to Bus Scale Slider, select H3 for the Text Style, select the Auto Size checkbox, and rename it Bus Scale Text. We have now set up the slider, which serves as our initialization event. - $Take\ Snapshot$

Following this, we will link this with the scaling animation and state control via a single C# script. The idea is to bind the slider’s value to the bus’s scale. Before proceeding with scripting, it’s advisable to maintain organization by creating a new folder:

1. Right-click in the Assets folder, choose Create | Folder, and name it Scripts.
2. Inside this new folder, create a new C# script and rename it BusScaler. This script will be our primary tool in bringing dynamic scaling to life.
3. Now double-click on the script. This will open it in the IDE that is installed on your computer. This is the code we are going to develop.

Writing the animation script The following explanations guide you on how the main components of the BusScaler script interact with each other:

1. Start with the following two lines of code to define your namespaces:

```
using UnityEngine;
using UnityEngine.UI;
```

The first step of any C# script in Unity is to include namespaces. A namespace is a collection of classes, structs, enums, delegates, and interfaces. This helps organize the code and provides a level of access control. The using keyword is used to include namespaces in the script. UnityEngine contains all the classes needed for creating games in Unity, while UnityEngine.UI contains classes for creating and manipulating UI elements. 2. The class declaration follows next. In our script, let's declare a new class named BusScaler with the following code sequence:

```
public class BusScaler : MonoBehaviour
```

This class inherits from MonoBehaviour, which is the base class for all Unity scripts. 3. Next, declare the variables of the BusScaler class with the following code sequence:

```
public GameObject bus;
public Slider slider;
```

The public keyword means these variables can be accessed from other scripts and can also be set from Unity's Inspector window. The bus variable will hold the bus object in the scene and the slider variable will hold the slider UI object in the scene. 4. Next, let's override the Awake() function to set up initial settings and references before the game starts. This includes setting the initial scale of the bus to 1 on the x-, y-, and z-axes. We do this with the following code snippet:

```
private void Awake()
{
  bus.transform.localScale = new Vector3(1f, 1f, 1f);
}
```

5. Our final step is to override the Update() function. Type or paste in the following lines of code:

```
private void Update()
{
  float scaleValue = slider.value;
  bus.transform.localScale = new Vector3(scaleValue, scaleValue, scaleValue);
}
```

This code retrieves the value of the slider (which ranges between its minimum and maximum values set in Unity's Inspector) and sets the scale of the bus to this value on the x-, y-, and z-axes. This means if the slider's value is 0.5, the bus will be half its original size. Testing our animation Now that our C# script is finished, there are only a few steps left to complete until we can test our animation:

1. In our C# script, the class BusScaler is derived from MonoBehaviour. To utilize this script, it needs to be associated with a GameObject within the Unity editor. To do this, right-click in the hierarchy and select Create | Empty GameObject. Rename this new object Bus Scaler Controller.
2. Next, you can drag your script and drop it into the Inspector panel of this new GameObject. This action will attach your script as a component.

3. Upon doing this, you’ll notice that the public fields bus and slider become visible in the script component within the Unity editor. These fields need to be populated with the actual bus GameObject and the slider object this script will interact with. To do this, simply drag and drop these objects into the corresponding field.
4. $Take\ Snapshot$
5. The final step before you can test your interactive slider is to place a teleport anchor in front of it. This can be done by going back to our toolbox and navigating to Assets | Samples | XR Interaction Toolkit:
6. Here, you’ll find the Teleport prefab. Drag and drop this to create a third teleport anchor, which we’ll call Bus Teleport Anchor.
7. Position this anchor at coordinates (0,0,-2.5) and adjust its size to (2,1,2). With this setup, the user can now teleport to a position in front of the slider, enabling them to scale the bus interactively.
8. $Take\ Snapshot$
9. Playtest
10. $Take\ Snapshot$

4. Summary

In this lab, we have learned how to:

- add interactivity into our VR scenes in Unity, using both:
  - code-dependent, and
  - code-free methods.
- decide when to opt for Unity’s animator system and when to leverage the power of C# for your interactive scene creation needs.
- trigger button events
- utilize the animator system, thereby equipping you with the skills to create simple yet effective interactions in your VR landscape without having to write a line of code.