2022

CA2 Report

VR & AR 2022

RYAN GOLDEN (STUDENT)

VR Project Design Document

11|02|2022
Ryan Golden

1 App Info

Tentative Title:	Archery Training

	Education & Training		Mental Health & Fitness
	Travel & Discovery		Media & Entertainment
	Productivity & Collaboration	✓	Gaming
	Art & Creativity		Other: _____

2 Pitch

To goal is for users to play:

Play a simple archery game, with score tracking and timers.

This will be especially fun in VR b/c:

Users will be able to dictate the direction and velocity of the arrow as if they were shooting a bow in real life.

At a high level, during the app, users will:

Be able to improve their accuracy and speed of which they handle the bow and be able to quickly set up for more shots. They will be able to improve their high score as more and more practice is done within the app.

This experience will be targeted at devices with:

6	degrees of freedom, giving users control over the	rotation	of their head & controllers.

3 Basics

The app will take place in:

A small custom built archery range

and the user will get around the scene with:

teleport	movement.

The user will be able to grab:

There will be sockets:

- Bow - Arrow - Quiver	- Bow Body and String - Arrows - Quiver

4 Events & Interactions

There will be haptic / audio feedback when:

- The user pulls and releases the bow string
- Notching the arrow

There will also be 3D sound from:

- The arrow being shot from the bow
- Various atmospheric sources

If the user is holding:

The bow	and presses the trigger,	A wooden pickup sound will play
The drawn string	and presses the trigger,	The string is released, and the arrow is fired
	and presses the trigger,	
		Suggestions: a UI change, a sound/video plays, a particle plays, an object is spawned or destroyed.

By default, the left hand will have a:

Direct	interactor.

and the right hand will have a:

Direct	interactor.

And you will not be able to toggle on a Ray interactor using the [thumbstick | button].

The main menu will be located:

On screen after hitting the allocated button. Alternatively, it will be accessible on a nearby wall.

and from the main menu, the user will be able to:

- Quit the game
- Change Audio settings
- Select other Game mode

[Optional] There will be additional UI elements for:

- Score
- Time
- Arrow Count (For Time mode)

5 Optimization & Publishing

To make the user experience more accessible / comfortable:

- The user is rooted in place, only movement involved is looking around
- The user will be able to control the bow with both their left and right hands
- Quiver can be placed on back or left somewhere else for easier access to arrows

Given that this app is targeting the [headset model], target metrics are:

Frames per second:	>= 60	FPS
Milliseconds per frame:	< 1000	ms (= 1,000 / FPS)
Triangles per frame:	_____ - _____	tris
Draw calls per frame:	_____ - _____	batches

Lighting strategy:

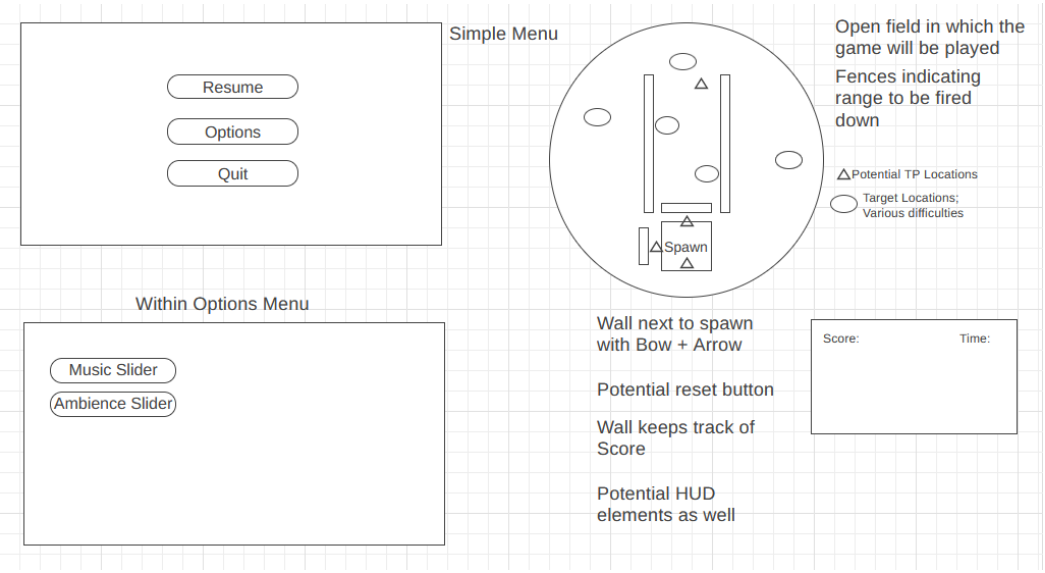
	All baked	✓	Mostly baked with some mixed		All real-time

Light probes [will | will not] also be used for more realistic mixed lighting.

6 Other features (Optional)

- Optional areas to explore / look around
- Ability to retrieve arrows manually or by hitting a switch
- Tutorial
-
-
-
-
-

7 Sketch (Optional)



TECHNICAL

The hardware chosen for this project was the Oculus Quest 2. This headset is both user and developer friendly, allowing for an easy development cycle. One of the best functions of the Quest 2 comes from the ability to use the Link cable which allows for real time testing of applications from within Unity. Simply put, it saves a lot of time since you don't need to build an application every single time you want to test out your product.

The Oculus Quest 2 is a very powerful headset, and was chosen because of this, alongside it's ease of portability and accessibility. The headset itself is completely wireless, unless you are specifically using the Link Cable (which has the added benefit of using the computers GPU power instead of the headsets internal systems). It has the added benefit of being incredibly affordable while still being able to run some of the newest high end VR games. Other headsets on the market can cost upwards of 600 dollars, apart from Sony's Playstation VR which starts at 300 dollars. Unfortunately, that also requires a Playstation Console to function, with the Oculus Quest being a standalone product for as little as 400 dollars. While weighing the options of other headsets, the Quest 2 easily comes out of top.

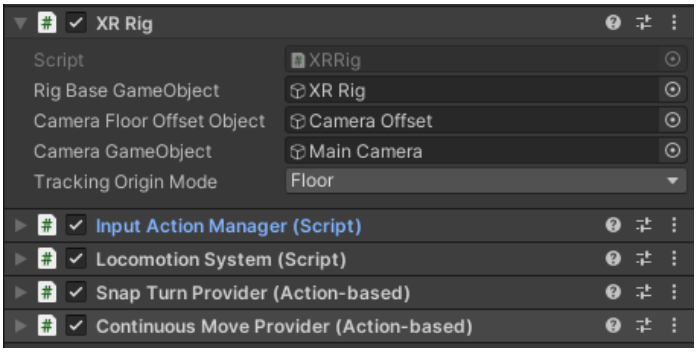
There are some setbacks to using the Quest 2 however. Other options have more robust systems for body and hand tracking. For example, the Valve Index has controllers that can fully track each individual fingers movement, allowing for incredibly precise interactivity within applications (Valve Index Controllers). Additionally, while the Quest uses built in trackers to track the players movement and location, other headsets such as the Valve Index or the HTC Vive use tracking stands, which come with both benefits and setbacks. The obvious setback of the base stations is that you are limited in the space that you can use the Headset, however users that have more then 2 base stations have much more consistent an accurate tracking on all pieces of hardware. Oculus has begun working on full hand tracking however, without the need for controllers at all by using the cameras built into the headset (Oculus News).

After deciding on the hardware, I then had to decide what framework to use for development of my app. In class, we had discussed and used AFrame, a web-based framework to develop a simple scene viewing application. Additionally, the Unreal Engine has many built in blueprints to get started with VR development. However, for the project it needed to be something more robust to allow for full interactivity. As such, the XR Interaction Toolkit was chosen as the core framework for this project, specifically build 1.0.0 Preview 3 (XR Interaction Toolkit).

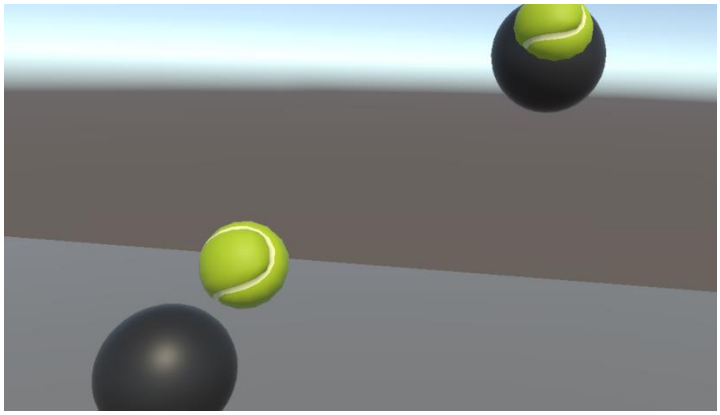
The XR Interaction Toolkit comes ready to install with every copy of Unity, only needing to be added to the scene using Unity's built-in plugin manager. This adds a plethora of premade scripts and components, to simplify the process of creating a Virtual Environment in Unity. It provides cross-platform controller support, simplifying the development across multiple VR Platforms, along with basic implementation of Unity's UI systems with VR and XR controllers. I followed a Unity tutorial which taught me the fundamentals of the XR Interaction Toolkit and used the finished product as a base to build my own application. For the project, I used Unity version 2020.3.18f1 as that was what the tutorial was using (Unity Learn).

DEVELOPMENT

sOnce the hardware and framework had been decided upon, I began work on my project. Using the room that had been created as part of our in-class work as a base, I created a new scene within that same project folder and added the core fundamentals using the XR Interaction Manager plugin. In the hierarchy, I added an XR rig object, which automatically changes the scene from a default unity scene to a VR scene, adding in a new directional light, an Input Action Manager and a default XR Interaction Manager. I removed the default one however and will explain why at a later point. With that, I added a few things to my XR Rig. Initially, I wanted to have set locations the player could teleport to, however I felt that constant locomotion would feel much more natural and give players a chance to really explore the scene I had created so I opted for continuous movement. I added the Locomotion System script to my XR Rig, as well as the Continuous Move Provider, mapping it to the left-hand controllers control stick for movement. I then added a Snap Turn Provider script to allow the player to rotate the camera using the right control stick.



With the core VR functionality added and a basic scene created, I began work on the player controllers. I didn't want to have a constant ray interactor coming from the players hands, so I changed each hand to a direct interactor, which requires the player to physically reach out and grab the objects they wish to interact with instead of simply pointing and clicking. Each hand was also given a Sphere Collider, and I set both colliders to be a Trigger, so that they could act as sockets or enable/disable scripts and objects within the scene. Adding some tennis balls into the scene and giving them the XR Grab Interactable script let me test my initial barebones build, and I began work on the main part of the project.



The bow and arrow would prove to be the most technical aspect of this project, as I didn't want to make a simple bow that spawns' arrows and shoot at a consistent velocity and trajectory. I wanted to make a bow that felt real, one that involved notching the arrow, pulling the string back and releasing to arch it across the scene. While this would have been a major task to undertake on my own due to the time constraints of the project and my own limited knowledge of Unity, I was lucky enough to find a tutorial online which provided exactly what I was looking for.

A Unity developer who goes by the name of Andrew had a video tutorial series (Bow and Arrow for Unity XR) which explained how he had developed and implemented a bow and

arrow in Unity using the XR Toolkit and had graciously provided people with the source code for his scripts and objects. His code came with a plethora of scripts, a handful of prefabs and models for the bow and arrow respectively, and a scene created to demonstrate the code itself. However, problems began to surface once I attempted to implement it into my own project. While the Unity Package with the bow and arrow worked perfectly in its own scene, any attempt at moving the prefabs and scripts over to my own scene was met with countless errors and issues. It wasn't until I exported it from his scene as a package was I able to get everything properly loaded into my scene.

Upon doing this however, another issue cropped up. This version of the bow and arrow had been developed with the XR Interaction Toolkit version 2.0, and as such was incompatible with my project, which was using the older version 1.0.0 Preview 3. I managed to find an older tutorial and attempted to follow that, but unfortunately that version of XR Toolkit was too old, and the same problems occurred with the code simply not functioning properly as it was trying to call on methods and functions that didn't exist. Eventually I found one of his tutorials that was built for Preview 4 of the XR Interaction Toolkit, which thankfully worked with Preview 3 as well. After creating a package from his premade scene and importing it into my project, I got to work on completing the code provided by working on the Notch script.

The Notch script is responsible for allowing all the separate Bow and Arrow scripts to properly interact with each other. It first creates a range value between 0 and 1 and creates the releaseThreshold value which dictates the minimum distance the string needs to be pulled for an arrow to be released. Once the Awake function is called, it grabs the PullMeasurer script so it can detect when the PullMeasurer is released, which in turn calls the ReleaseArrow function from within the script.

```
protected override void OnEnable()
{
    base.OnEnable();

    // Arrow is released once the puller is released
    PullMeasurer.selectExited.AddListener(ReleaseArrow);

    // Move the point where the arrow is attached
    PullMeasurer.Pulled.AddListener(MoveAttach);
}
```

The PullMeasurer script is responsible for calculating the distance that the String has been pulled back, as well as the target direction. The String prefab has been given 3 separate transform points (Start, Middle, and End) and uses these 3 values to limit how far the pullback animation goes, but also calculate the force created due to the String being released. This in turn is then applied to the Arrow object, causing it to fly different distances based on what the PullMeasurer returns.

```
private float CalculatePull(Vector3 pullPosition)
{
    // Direction, and length
    Vector3 pullDirection = pullPosition - start.position;
    Vector3 targetDirection = end.position - start.position;

    // Figure out out the pull direction
    float maxLength = targetDirection.magnitude;
    targetDirection.Normalize();

    // What's the actual distance?
    float pullValue = Vector3.Dot(pullDirection, targetDirection) / maxLength;
    pullValue = Mathf.Clamp(pullValue, 0.0f, 1.0f);
```

Within the Arrow script, the Launch function is called (after checking that the Arrow is in fact attached to the string). This in turn enables the SetLaunch function and the value that the PullMeasurer returned to Notch is sent to the ApplyForce function. Within ApplyForce, the value from PullMeasurer is set as a float value and is used to add force to the Arrow Prefab's rigid body component, which in turn gives it forward momentum.

```
private void Launch(Notch notch)
{
    // Double-check incase the bow is dropped with arrow socketed
    if (notch.IsReady)
    {
        SetLaunch(true);
        UpdateLastPosition();
        ApplyForce(notch.PullMeasurer);
    }
}
```

Back in the Notch script, once an Arrow is released a few things happen. First, the attach point for the Notch is immediately reset back to its beginning location (That being before the string is pulled). This also updates the string renderer. The notch is set back to a ready state once there is no arrow while the bow is being grabbed, and everything is waiting for the Notch to detect another Arrow object.

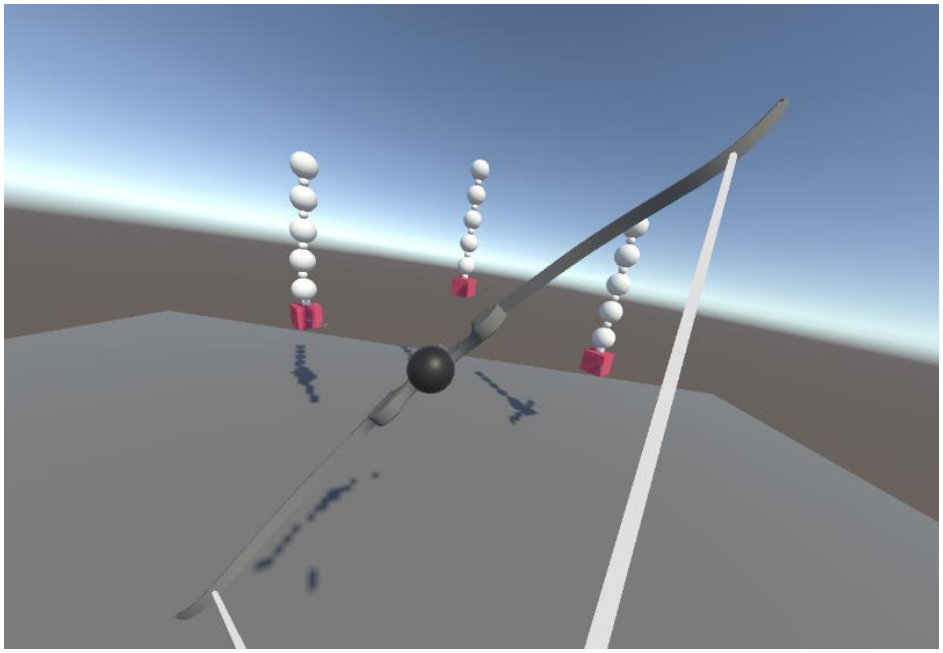
However, the Arrow Script is still functioning if an arrow object exists. While an arrow is flying, the CheckForCollision script is running, which repeatedly looks to see if the arrow object has come in contact with any other colliders in the scene as often as possible. If a collision is detected the position of the arrow's body and head are saved and physics for the object is disabled, freezing it in place.

```
private bool CheckForCollision()
{
    // Check if there was a hit
    if (Physics.Linecast(lastPosition, tip.position, out RaycastHit hit, layerMask))
    {
        TogglePhysics(false);
        ChildArrow(hit);
        CheckForHittable(hit);
    }

    return hit.collider != null;
}

private void TogglePhysics(bool value)
{
    // Disable physics for childing and grabbing
    rigidbody.isKinematic = !value;
    rigidbody.useGravity = value;
}
```

As I stated earlier, I deleted the premade XR Interaction Manager from the hierarchy. The reason for this is because by default, any object that requires an XR Interaction Manager will grab the base one created by the XR Toolkit, regardless of whether it exists in the scene. In order for the Notch script, and in turn every other script for the bow and arrow to function properly a custom Interaction Manager had to be created. Inside of this custom interaction manager, there is a Force Deselect function. This is what allows the String to be reset after an arrow is released, and will remove anything notched to the string if the bow object has been released as well.

```
// Need to cast to custom for Force Deselect
private CustomInteractionManager CustomManager => interactionManager as CustomInteractionManager;
```

This one line within the Notch script simply means that the Custom Interaction Manager can also be called using the default Interaction Manager, for ease of access.



With the scripts properly imported and modified to work within my scene, the core gameplay of my application was finished, and I had the beginnings of my Archery Training game.

After finally getting the bow and arrow to function properly within my project, I moved onto something less code intensive and began work on the environment. Since my game was based around archery, I wanted to create a small, forested area with an archery range that the player would be shooting down. From the Unity Asset store, I found a bundle of free assets from a user called Broken Vector. Their low poly prefabs were perfect for the overall aesthetic of the game I wished to create, and I utilised the Tree, Cliff, Fence and Rock packs respectively. The Cliffs were perfect to use as a natural world border, and the small valley I created was easily populated with the Trees and Rocks from the packs. I shrunk down some of the trees and used them as makeshift bushes as well, and the fences were used to create the actual archery range itself. Immediately upon testing the world however, I discovered that the arrows could simply phase through some object in the environment, primarily the trees, fences and rocks. This issue was solved by going to the prefabs that I used in my scene and adding Mesh Colliders to each of them.

With the environment partially done, I began working on the games point system. This would be the core gameplay element, with players being able to earn points and add them to their score by hitting targets down the range using the bow and arrow. I created the ArrowController script and initially as a proof of concept, I simply made it destroy the Arrow objects whenever they came in contact with an object in the scene under the tag “Target”. I did this to ensure that the arrows created were going to be affected by this script.

After testing, I created 3 tags for the targets: easy, medium and hard. Each of these tags would correspond to a different number of points that the player would earn by hitting them. In ArrowController script, I first created 3 public integers for each tag with a unique score set for them. Within the OnTriggerEnter function that was previously used to Destroy the projectiles, I call the public integer score from the ScoreController script, and added the value given to whichever target the player had hit. For example if they hit an easy target, the OnTriggerEnter function would cycle through if statements, and Unity’s CompareTag function would detect the tag “targetEasy”, and add the scoreEasy integer value to the players score (impact.Play(); simply plays a sound effect if the target is hit). The ScoreController script simply stores the value for score and updates the UI on screen when the player scores points. I repeated this if statement 3 times for each tag that the player could earn points from, scaling with difficulty.

```
private void OnTriggerEnter(Collider other)
{
    if(other.gameObject.CompareTag("targetEasy"))
    {
        ScoreController.score += scoreEasy;
        impact.Play();
    }
```

Once the players score was updating properly, I of course worked on a high score. This was achieved very simply. Within the ScoreController script I added a highscoreDisplay class that updates when a new high score is achieved. I then modified the if statement within the script to update alongside the score if the current score is higher then the previous high score. I used PlayerPrefs to store the high score between sessions. All this information is being sent directly to a UI component on a fence on the right-hand side of the scene.

```
{
    if (score > highscore)
    {
        highscore = score;
        PlayerPrefs.SetFloat("Highscore", highscore);
    }

    scoreDisplay.text = score.ToString();
    highscoreDisplay.text = highscore.ToString();
}
```

The UI components within the game are relatively simple. I began work on the welcome message along with the tutorial once I had finished the points system. The welcome message simply uses the XR Interaction Toolkits XR Canvas object, and I added two buttons to the initial screen. One that closes the message entirely, and another that begins a series of tutorial messages to teach the player how to use the bow and arrow properly. While the tutorial is active, a small video plays underneath the canvas as a visual aid on how to play the game. As the canvas had buttons, I had to go back and add a ray interactor to at least one of my controllers. I ended up adding it to the right controller and adding a toggle so that the ray would only show up once the controller’s primary button (On the Oculus Quest it’s the A button) is being held down. While held down, the trigger button allows the user to

interact with the buttons on the in-game UI. Each button has an on Click function set in the inspector, which either activates or deactivates an upcoming or current message on display.

Meanwhile, the score UI is slightly more complicated. I first created the Game UI canvas as a separate object in the hierarchy and attached the previously created Score Controller script to it. On the UI, I added 4 text objects to keep track of both score and high score. I dragged the score and high score counters into the empty slots in the Score Controller script, so that the text would update automatically in the scene. I also added a Timer text object to the UI and began work on an in-game timer.

I created a new script called CountdownTimer. This script sets a value for the variables currentTime and startingTime along with creating a variable called countdownText. In the Update function, 1 second is removed from currentTime every second relative to the scene using Time.deltaTime. This value is then sent to the currentTime variable as a String, which updates the display in game. Finally, inside of the if statement, when currentTime is less than or equal to zero, the time is repeatedly set to zero, and the countdownText is changed to display the users score at the end of the timer.

```
void Start()
{
    currentTime = startingTime;
}

void Update()
{
    currentTime -= 1 * Time.deltaTime;
    countdownText.text = currentTime.ToString("0");

    if(currentTime <= 0)
    {
        currentTime = 0;
        countdownText.text = currentTime.ToString("
    }
}
```

After this, I had to create a way to restart the timer. I created another script, this one called ResetScore, and only added two lines of code to it. Within the script, when the OnClick function is called, ResetScore sets the score variable in ScoreController to 0, and it sets the currentTime in CountdownTimer to 30, restarting the timer and the score at the same time.

While working on this however, I encountered a bug that allowed players to earn points even while the timer wasn't counting. Even though the UI wasn't updating, players could theoretically hit a target as many times as they wanted before starting the timer, and once they start it, every arrow is immediately counted towards the score. I struggled to fix this bug for a while, as I couldn't figure out a way to disable the scoring script while the timer wasn't active. I attempted to grab the ArrowController script from the Arrow object in order to disable it once the timer hits zero to no success.

```
//GetComponent<ArrowController>().enabled = false;
//GameObject.Find("Arrow").GetComponent<ArrowController>().enabled = false;
```

```
[13:26:00] Assets\CA2\Scripts\CountdownTimer.cs(29,62): error CS0119: 'ArrowController' is a type, which is not valid in the given context
```

The solution was far simpler than I thought. Within ArrowController, I added an Update function at the end. This update function simply checks what the currentTime is from the CountdownTimer script, and if it's zero or less, all targets score value is set to 0. This reset the second the timer is restarted, meaning players can only earn point when the Timer is counting down.

```
if (CountdownTimer.currentTime <= 0)
{
    scoreEasy = 0;
    scoreMedium = 0;
    scoreHard = 0;
}
```

With that, the core gameplay had been finished. I had a working point and high score system, as well as a timer that limits how long players can earn points for. The Bow and arrow worked very well, and allowed for precise aiming and control, and I had nearly finished creating the environment. First however, I had to fix the player movement. Up until this point, the player could pass through every single object in the scene, as the XR Rig itself had no collision. To try and fix this, I added a rigid body element to the XR Rig but that only resulted in the player being affected independently by gravity, and not having proper control. A capsule collider added to the XR Rig also failed to give the results I was hoping for. After searching online, I found the solution I needed. To the XR Rig, I had to add a Character Controller Driver script, along with the Character Controller component. This automatically gave the player collision with everything in the scene, and I simply reduced the size of the capsule to feel more comfortable.

Once I was happy with how the game looked and played, I went back to the environment and added a few more things to make the scene feel more fleshed out. I found another user on the Unity Asset Store by the name of JustCreate. They had a few more low poly assets that I could use for my project, most notably the smaller plants and grass. However, I didn't want to have hundreds of these grass objects in my scene, so I created four brand new prefabs using JustCreate's assets to use around my game. This was simply done to minimise the number of prefabs the game had to load in. I added a small number of mushrooms around the score UI wall, and a fallen log to act as a barrier. Additionally, I found a set of low poly targets on sketch fab, by the user Bram Van Hoof, and added them into the scene as the targets the player would be aiming for.

There were a few more things I had to add to my game. I first created invisible walls all around the environment to ensure the player could not accidentally get out of bounds or stuck. These are simply cubes with their mesh renderer disabled. After, I added some sound effects and ambience to the game. When the scene loads, a looping track is played in the background. There is no music in the game, just ambient noises. I added an audio source to the arrow prefab and in the Arrow Controller script I added a variable Audio Source called

impact. This grabs the AudioSource from within the arrow prefab, and every time a target is hit, the audio plays once. Additionally, I added another audio source to the Notch prefab, and within that Notch script I did the same thing. I pulled the Audio Source from the prefab object, and in the ReleaseArrow function, the audio is played.

Code Errors/Testing

While programming the Scripts for the project, most of my issues came from importing the Bow and Arrow code into my own project, and many of the issues simply stemmed from having incompatible versions. As such, the errors I experienced went mostly undocumented, as I never fixed any of the code, I simply found the correct version. However, the bow and arrow were part of the project that gave me the most trouble, which is why I was determined to get it working first.

Some of the issues that I came across were as follows.

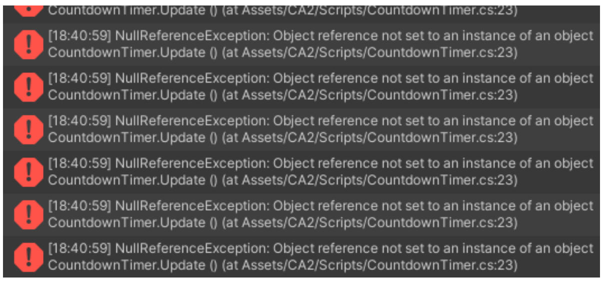
1. The models wouldn't import to the scene properly, the solution was to export them as a package from their original scene and import them to the new one
2. Broken code from the pre-made scripts I was using, discovered that the Bow and Arrow model I was using was for a newer version of XR Toolkit, this happened twice.
3. Attempted to add physics to the bow as well, bow and arrow immediately flew around the scene when notched together, had to disable the physics on the bow.
4. Once properly imported, the bow wouldn't fire at all. Had to create a custom XR Toolkit manager with a custom script to have the bow fire.

Another issue I came across was when updating the score UI, the timer and scores points would reset back to 0, making it impossible to see what your final score was. The solution was to simply make the score and timer rest when the button was pressed, instead of when the timer hit zero.

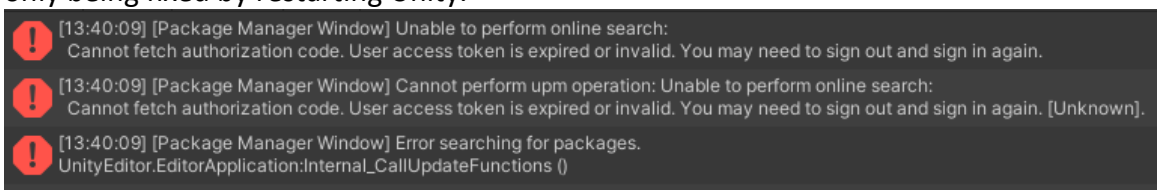
The points system originally kept track of arrows that hit target before the timer had begun, and so players could earn points by shooting a target multiple times, starting the timer, and then immediately being rewarded with those points. The solution was to simply se the targets point values to zero when the timer is at zero.

When adding Audio to the objects, I originally tried to call them directly from the Arrow script. However, as the arrow object does not exist until the player spawns one in, I could not modify the prefab too much. This had me stuck for a while, as I wasn't sure how I could make the arrow itself play a noise without using the Arrow script. Thankfully, I found a tutorial online, and the solution was to simply add an Audio Source directly onto the arrow and call it from another script on the object. For this I added it to the ScoreController script, and every time one of the targets is hit, the impact sound is played. I used the same solution for the Strings release sound, adding the Audio Source to the Notch object and within the script, playing that whenever the String is released.

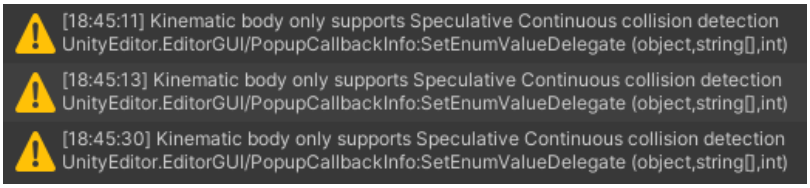
Some smaller errors that occurred were the countdown timer being set to a second object and throwing back hundreds of errors every frame. Deleting the script from the other object fixed this.



The package manager not loading the assets I had downloaded from the Asset store, and only being fixed by restarting Unity.



Or the players body not having any collision values and being capable of passing through every object in the scene, also limiting what option could be selected when trying to apply a rigid body to it. The solution was to use XR Toolkits Character Controller Driver and Component.



For user testing, I only managed to get one person to test out my game thoroughly, and they gave me a small number of feedback and improvements I could make. Primarily, they noted that the Score UI was very blurry, and it would be better if it was clearer. The reason for this was because I had originally used Text UI objects instead of Text Mesh Pro objects when creating the UI, and so I updated it to look sleeker and clearer. Additionally, they brought it to my attention that the high score didn't properly update unless the scene was reloaded. This too was something I fixed relatively quickly, I only had to add a line of code to the Update function within the ScoreController script that checks for a newly stored high score every frame and updates the number in real time.

```
highscoreDisplay.text = highscore.ToString();
```

Reflection

Working on this project has taught me the value of proper time management and work ethic. While I enjoyed the work while there were clear results on my screen, often I would get frustrated when a roadblock would crop up and stop me from progressing any further. Many times, during development I would overthink solutions for problems that I had, when the actual solutions ended up being incredibly simple, such as with the Score not being worth points while the timer wasn't active. I attempted to disable the script instead of simply changing the values of the score, which cost me a lot of time and frustration.

I am far more familiar with XR Toolkit as well as the Oculus Quest 2 from a developer standpoint. The XR Toolkit makes it incredibly easy to start creating intricate and enjoyable VR experiences with their premade scripts, and the Quest is a very easy and fun platform to work with. I can definitely see myself using these options in the future should the need arise, and with my heightened knowledge I could happily take on more difficult tasks.

Bibliography

Bow and Arrow for Unity XR. (n.d.). Retrieved from Youtube: https://www.youtube.com/watch?v=H0xTz4JtWiI&list=LL&index=1&t=1s

Oculus News. (n.d.). Retrieved from Oculus: https://developer.oculus.com/blog/hand-tracking-sdk-for-oculus-quest-available/

Unity Learn. (n.d.). Retrieved from Unity Learn: https://learn.unity.com/tutorial/explore-the-escape-room?uv=2020.3&projectId=5e4abf44edbc2a09bf60dceb#

Valve Index Controllers. (n.d.). Retrieved from Valve Index: https://www.valvesoftware.com/en/index/controllers

XR Interaction Toolkit. (n.d.). Retrieved from Unity Documentation: https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.2/manual/index.html