**Overview**

Patients who suffer from severe burns are often limited in mobility. In addition to the inextricable pain from the burns themselves, patients are often discouraged from their loss of mobility and feel a desire to explore. Studies show that video games in particular can both reduce the pain experienced by burn patients and increase overall mood by placing the patients in an immersive environment with elements contrary to the things that may have experienced during their injury (<https://www.hitl.washington.edu/projects/vrpain/>).

ReliefFinallyis a virtual reality-enabled adventure-based video game which aims to de-stress burn patients by placing them in calm environments where they are enabled to explore, unlock new environments, and interact with various elements (e.g. snow, water, wind, ice, etc.) that stand contrary to the reason for their injury.

**Setting up the project for development**

**Requirements:**

Must be running windows.

Must have a GearVR-compatible phone if you wish to test the game on android.

Follow this link: <https://cgcookie.com/2015/12/16/quick-start-guide-gear-vr-unity/>, or follow the summary below.

Download latest version of Unity (<https://unity3d.com/unity> -> Personal -> Learn More -> Try Personal -> download installer) and install.

Install Android Studio [https://developer.Android.com/Studio/index.html](https://developer.android.com/studio/index.html)

Find the file path of the Android SDK and save it somewhere.

Install most recent JDK (google java jdk and download the Java Se Development kit), find the file path of the JDK, should be in C:\ProgramFiles\Java and appear as jdk-somenumbers or jdksomenumbers) and save the file path somewhere.

Download project from github: <https://github.com/takoda1/ReliefFinally>.

Open Unity and open the root folder of ReliefFinally to open the project.

In the editor window, go to edit -> preferences.

Click External Tools.

Under Android, it should say SDK and JDK, put the saved file paths of the SDK and JDK into these boxes.

Next, in the editor, go to File -> Build Settings, click Android, then click Switch Platform.

In the same window, click on player settings, then the Android logo in the top bar.

Click Other Settings and under Identification, change the Minimum API level to 19 or above.

If you are using Visual Studio to edit code, make sure to also set the… (Experimental version 4.6 leads to crashes when run on the device)

In the same player settings window, click XR Settings and enable Virtual Reality Supported.

Now follow the directions in the **Building for the Gear VR section** of <https://cgcookie.com/2015/12/16/quick-start-guide-gear-vr-unity/>.

**Using Visual Studio to edit code**

<https://docs.unity3d.com/Manual/VisualStudioIntegration.html>

In Edit -> Project Settings -> Player, under configuration, make sure the Scripting Runtime Version is set to Stable(.NET 3.5 Equivalent). Builds to an android device will fail if the runtime version is 4.6 experimental as of 2/16/2018.

**Debugging the game on the Android device.**

Must have Android Studio or at least developer tools to debug.

It is easiest to have the Android device connected by USB for debugging. If you wish, you can also wirelessly connect your device for debugging by following the instructions under **Connect to a device over Wi-Fi** here: <https://developer.android.com/studio/command-line/adb.html>.

First you have to enable USB debugging on the Android device. In settings, find the Build Number setting, usually under About Phone, and click it at least seven times. This should enable Developer options. Next click on Developer options in the Settings screen. Scroll down in the Developer options screen and enable the USB debugging setting. Now, whenever the phone is connected to a machine it should prompt you to allow or prevent USB debugging from the said machine. Allow debugging from your machine when you connect the Android device.

There are two methods to see Logcat logs from the Android device. Non-gui Adb Logcat and Gui monitor. Both can be found deep within your file system and run from there, but it is easier to set them as system PATH variables and run them with short commands from the Command prompt (Instead of running C:\Users\username\AppData\Local\Android\sdk\platform-tools adb logcat, you can just type adb logcat and execute).

To set system PATH variables in windows 10, type View Advanced System Settings in the search bar, click Environment variables in the window that pops up, then under System variables, click Path to highlight it then click Edit. To add new environment variables, click New, then either type in the path or browse for it. The path for logcat and monitor.bat should be something like C:\Users\username\AppData\Local\Android\sdk\platform-tools and C:\Users\username\AppData\Local\Android\sdk\tools.

Now, open a command prompt (type cmd in search bar -> press enter) and type either adb logcat or monitor.bat and press enter to execute.

**Adb Logcat summary**

Keep in mind that I’m not very familiar with Logcat, this is a very general overview.

To view a specific log, click on the desired log to stop new ones from scrolling the logs up.

To make the logs continue scrolling after stopping the logs, press shift+enter.

To exit adb logcat press ctrl+c.

If you just type adb logcat, you get every single log from the android device, which is undesireable if you just want logs from the game. Instead, use the logcat options to filter for what you want: <https://developer.android.com/studio/command-line/logcat.html>.

**Monitor summary**

In the bottom half of the monitor, it should be displaying all logs from Logcat. In the search bar at the top, you can type in any string that you are looking for in the logs and it will automatically filter those logs for you to view.

**Project structure**

**Controllers**

Project compatible with either the GearVR controller or StratusXL controller.

GearVR Controller has no axis/input mappings, instead, inputs must be handled by code. Look at OVRGearVRControllerTest (Assets -> OVR -> Scripts -> Util) to see which code maps to which button.

I’ve already synced the StratusXL Controller with this project, but read below if you want to change button functionality.

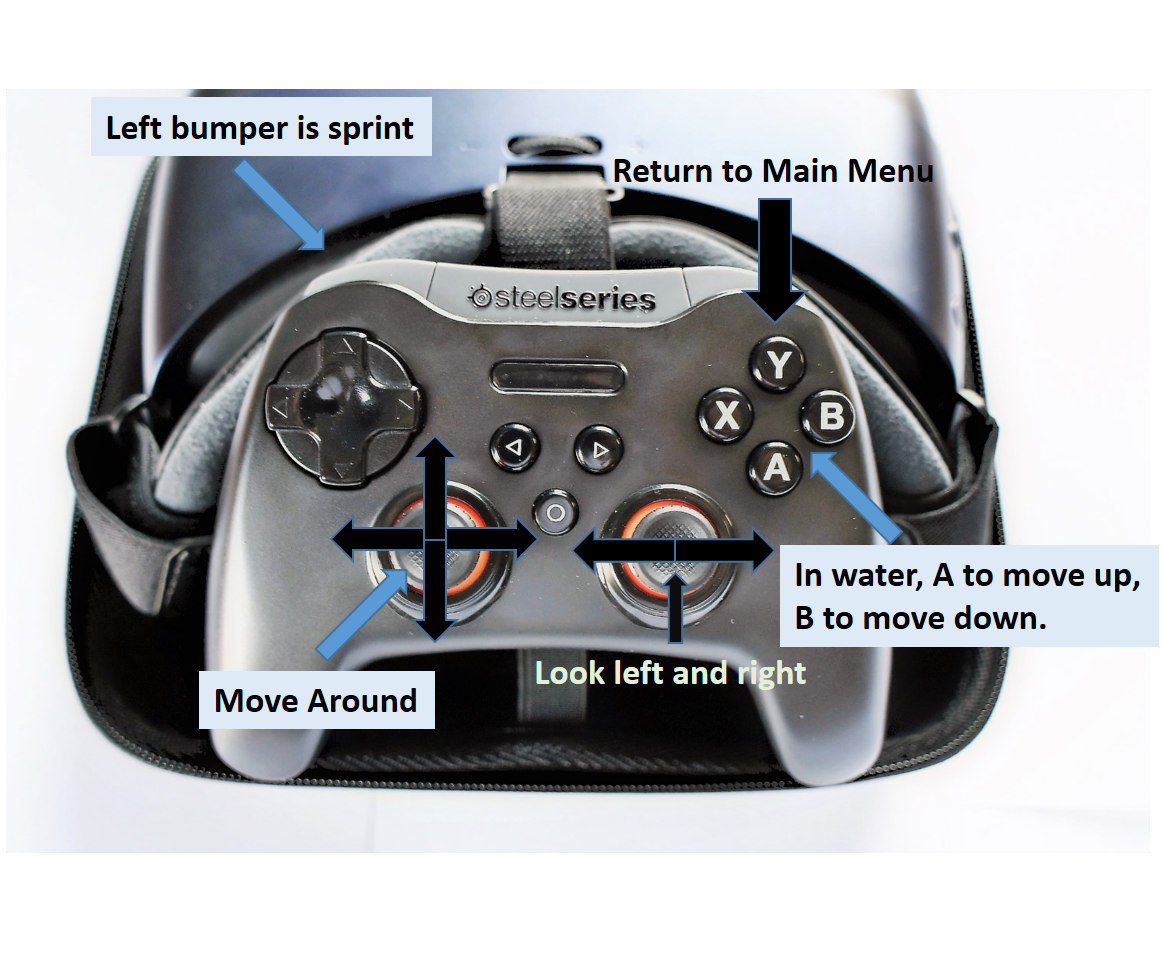
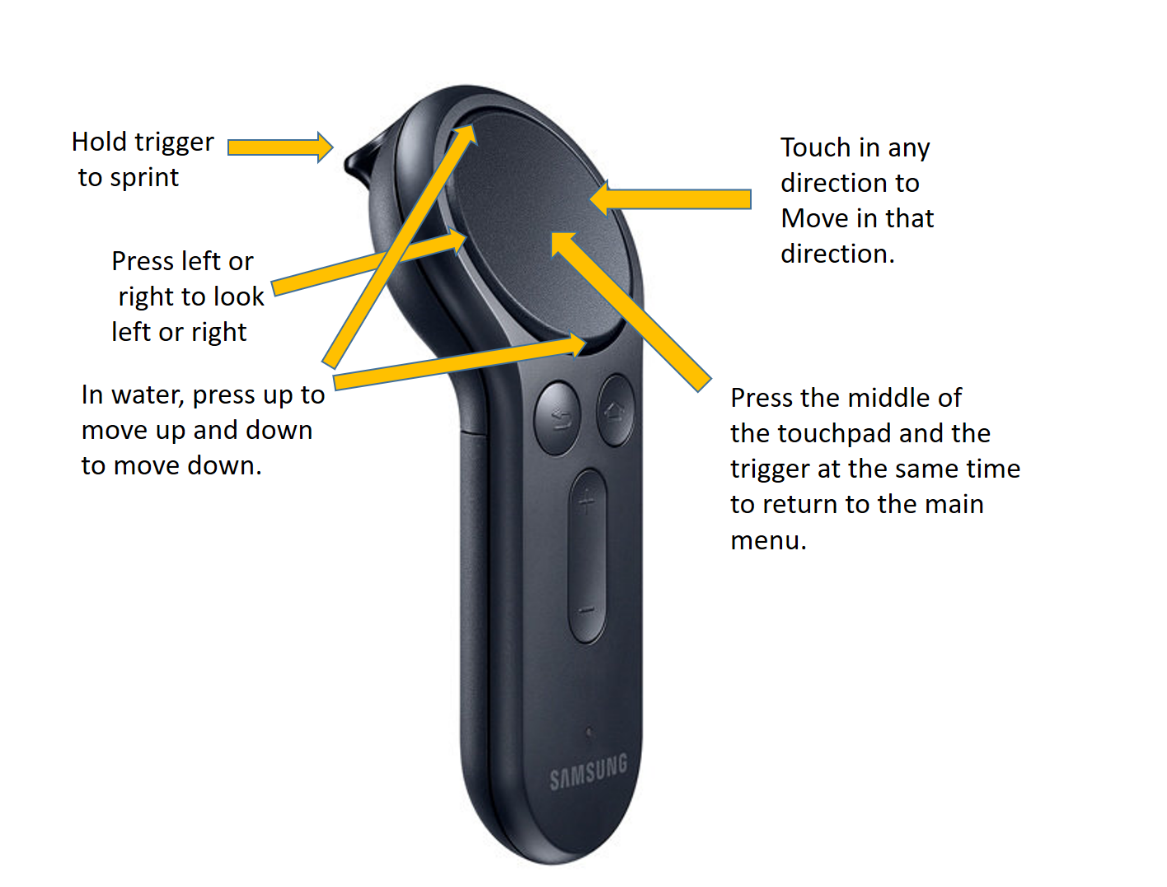
Syncing the StratusXL Controller with Unity:

<http://primeyoullc.net/steel-series-stratus-xl-game-controller-mapping-unity-3d/>

Button and axis mappings for the StratusXL for convenience.





These are the controls while playing in game.

**Scene Structure**

Scenes

MainMenuScene

InfoScene

TutorialScene

GrassyPlainsScene

BarnacleWatersScene

SnowyMountainScene

GrassyPlainsSceneControlled

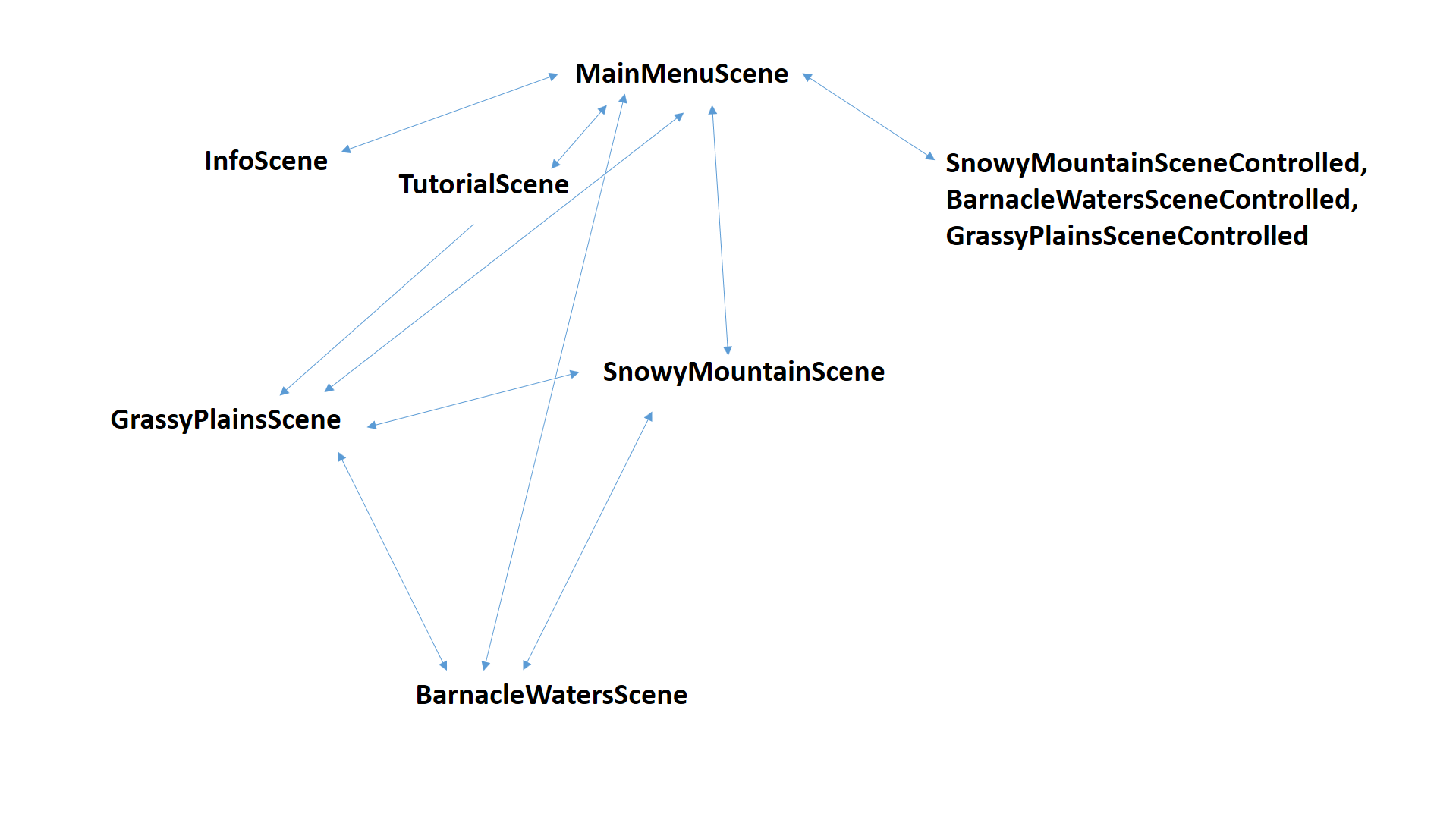
BarnacleWatersSceneControlled

SnowyMountainSceneControlled

Controls for the scenes

|  |  |  |
| --- | --- | --- |
| Scene | Look around with head | Controller movements |
| MainMenuScene | Y | N |
| InfoScene | Y | N |
| TutorialScene | Y | Y |
| GrassyPlainsScene | Y | Y |
| BarnacleWatersScene | Y | Y |
| SnowyMountainScene | Y | Y |
| GrassyPlainsSceneControlled | Y | N |
| BarnacleWatersSceneControlled | Y | N |
| SnowyMountainSceneControlled | Y | N |

Scene Interaction



Players can reach any scene from the MainMenuScene. Conversely, players can reach the MainMenuScene from any scene simply by pressing the trigger and the middle of the touchpad on the GearVR Controller or Y on the StratusXL. The end of the TutorialScene leads into the GrassyPlainsScene. The GrassyPlainsScene can be accessed from both SnowyMountainScene and BarnacleWatersScene without collecting any treasures. Treasures needed to access SnowyMountainScene are found in GrassyPlainsScene, treasures needed to access BarnacleWatersScene are found in SnowyMountainScene, and once all treasures needed to access a scene are found, the player can access that scene freely from either of the other two scenes excluding itself of course among GrassyPlainsScene, SnowyMountainScene, and BarnacleWatersScene. SnowyMountainScene and BarnacleWatersScene can be accessed directly from the MainMenuScene, but once you leave them for another scene among the three listed above, you won’t be able to return until you have collected the required treasures.

**UI Tips**

To have appropriate UI that follows a player around for VR, instead of using a regular canvas with Render Mode set as ScreenSpace …, change the render mode to World Space and find a distance away from the player suitable for reading. I used 42 units. To make text clearer, make the height and width of the text element huge but correspondingly scale it down by the appropriate amount, so with 100\* greater width you should have scale be .01. This way, text becomes much clearer.

**Individual Scene Explanations**

**MainMenuScene**

Overview

The player looks around from a center pedestal at rectangular menu items that each represent a specific scene and can enter the desired scene by looking at it, which pulls the menu item forward to indicate that it is being hovered over, and pressing the trigger on the GearVR controller or A on the StratusXL.

Oculus already has prefabs that handle all the details of a camera for each eye and how they are tracked by head motion, so I placed the OVRCameraRig (Assets -> OVR -> Prefabs), which allows the player to look around in a scene, on top of the center pedestal.

To make the rectangular menu items react while being looked at, you need a script attached to the camera that sends out rays and see if the rays have hit a desired object, that being a menu item. If a ray has hit a menu item, call the appropriate methods on the menu item. Therefore, the menu item should also have a script with methods that allow it to perform behaviors.

Scripts

For the menu item, I made a general script that reacts to clicks and gaze and that will be subclassed, InteractiveItem, with methods Over, Click, Out, and Down. MenuItemInteractiveItem then subclasses InteractiveItem and implements the behavior of a menu item. The script is then attached to any object that is to be a menu item. If you want to specifically see its behavior look at the code and comments in the script (Assets -> Scripts -> Menu Scripts).

The MenuEyeRaycaster (Assets -> Scripts -> Menu Scripts) script allows the user’s gaze and appropriate trigger presses to interact with IteractiveItem (and by extension MenuItemInteractiveItem due to subclassing). This script casts a ray in the direction of the forward direction of the object that it is attached to (the Center Camera in OVRCameraRig in our case) and checks if the “gaze” has just passed over or off an InteractiveItem, and calls the appropriate methods on the InteractiveItem, then updates variables that indicate what is being looked at. If the trigger or A is pressed while an InteractiveItem is being looked at, it calls the Click method on the respective InteractiveItem.

**InfoScene**

Overview

The player looks around from a center pedestal at graphics that indicate controls, as well as text describing controls. The same OVRCameraRig that is used in the MainMenuScene is used here as well. There is no interaction in this scene other than pressing the trigger and the middle of the touchpad on the GearVR Controller or Y on the StratusXL to return to the MainMenuScene.

**Important section on OVRPlayerController**

To allow a player’s controller inputs to allow for movement, it is simplest to use the prefab object provided by Oculus, OVRPlayerController (Assets -> OVR -> Prefabs). However, the script attached to the OVRPlayerController prefab, OVRPlayerController (Assets -> OVR -> Scripts -> Util), does not appropriately handle movements from the GearVR controller (it does for most of the StratusXL controls however because in OVRPlayerController, they use OVRInput.Axis2D.Primary/SecondaryThumbstick, which maps to the StratusXL’s analog sticks). So it must be slightly changed in order to handle GearVR controller inputs, which isn’t desirable because OVR updates their code OVR package every once in a while, so you have to add a chunk of code to OVRPlayerController every time a new update to the OVR package comes out. However, the alternative in making a new class to handle just GearVR controller inputs is a bit complicated because then you have two classes updating the CharacterController and cameraRig. So I decided it would be easier to just change the OVRPlayerController class slightly. In addition, if you want custom functionality that logically could be connected to a player’s movements (footsteps in our case), you probably don’t want additional changes to OVRPlayerController, so I had GearVRPlayerController subclass OVRPlayerController and implement footstep related code.

In OVRPlayerController, there is a section of code that looks like this:

public void UpdateMovement()

{

if (HaltUpdateMovement)

return;

bool moveForward = Input.GetKey(KeyCode.W) || Input.GetKey(KeyCode.UpArrow);

bool moveLeft = Input.GetKey(KeyCode.A) || Input.GetKey(KeyCode.LeftArrow);

bool moveRight = Input.GetKey(KeyCode.D) || Input.GetKey(KeyCode.RightArrow);

bool moveBack = Input.GetKey(KeyCode.S) || Input.GetKey(KeyCode.DownArrow);

MoveScale = 1.0f;

//PLACE CODE HERE

if ( (moveForward && moveLeft) || (moveForward && moveRight) ||

(moveBack && moveLeft) || (moveBack && moveRight) )

MoveScale = 0.70710678f;

...

You will place the following lines of code where I have indicated (//PLACE CODE HERE) if it isn’t there already.

#if UNITY\_ANDROID && !UNITY\_EDITOR

//if the gearVR controller is being used

if (OVRInput.GetActiveController() == OVRInput.Controller.LTrackedRemote ||

OVRInput.GetActiveController() == OVRInput.Controller.RTrackedRemote)

{

if (OVRInput.Get(OVRInput.Touch.PrimaryTouchpad) && !OVRInput.Get(OVRInput.Button.PrimaryTouchpad))

{

Vector2 touchPosition = OVRInput.Get(OVRInput.Axis2D.PrimaryTouchpad);

if (touchPosition.y > 0)

moveForward = true;

if (touchPosition.y < 0)

moveBack = true;

if (touchPosition.x > 0)

moveRight = true;

if (touchPosition.x < 0)

moveLeft = true;

}

if (OVRInput.Get(OVRInput.Button.PrimaryIndexTrigger))

{

MoveScale \*= 2.0f;

}

}

if(Input.GetButton("Button 4")) //button 4 maps to the left bumper on the controller

{

MoveScale \*= 2.0f;

}

//if using gearVR controller

if (OVRInput.Get(OVRInput.Button.PrimaryTouchpad))

{

Vector2 pressPosition = OVRInput.Get(OVRInput.Axis2D.PrimaryTouchpad);

if (pressPosition.y < .5 && pressPosition.y > -.5)

{

if (pressPosition.x < -.3)

{

buttonRotation -= RotationRatchet;

}

if (pressPosition.x > .3)

{

buttonRotation += RotationRatchet;

}

}

}

#endif

In addition, comment out these lines because they make it so that StratusXL bumpers rotate the player in discrete increments, which is undesirable because the dpad already does that and the bumpers are nice for sprinting.

bool curHatLeft = OVRInput.Get(OVRInput.Button.PrimaryShoulder);

if (curHatLeft && !prevHatLeft)

euler.y -= RotationRatchet;

prevHatLeft = curHatLeft;

bool curHatRight = OVRInput.Get(OVRInput.Button.SecondaryShoulder);

if(curHatRight && !prevHatRight)

euler.y += RotationRatchet;

prevHatRight = curHatRight;

Now, GearVRPlayerController (Assets -> Scripts -> Scene Scripts) simply subclasses OVRPlayerController. Then, on the OVRPlayerController prefab, replace the script OVRPlayerController with GearVRPlayerController instead.

**TutorialScene**

Overview

The player moves through a short scene with sections explaining controls, maps, treasures, and portals.

The slightly modified (described in the Section on OVRPlayerController above) prefab of OVRPlayerController (Assets -> OVR -> Prefabs) is used to allow the player to move around. There are 3 rectangular areas in the scene where if the player enters, corresponding tutorial text will start running across the screen. The player can continue through text by pressing the trigger on the GearVR Controller or A on the StratusXL.

Scripts

TypeText (Assets -> Scripts -> Tutorial Scripts)

To allow for running text functionality, I made a TypeText class that should be attached to a Text element. Whenever a text is set to the text field of TypeText using the method setText, it clears the attached Text element and begins adding one character from the set text at a time to the Text element’s display at a specified time interval.

ZoneText (Assets -> Scripts -> Tutorial Scripts)

To allow for certain areas of the map to display certain text, I made the simple class ZoneText which contains an array of Strings field that contains all the text a certain section of the tutorial is to have. It is then attached to a gameobject that acts as a trigger.

Tutorial (Assets -> Scripts -> Tutorial Scripts)

Tutorial should be attached to the parent gameobject that has the canvas that is to have text typed on it and that is to move around in the scene. This usually means whichever object is the player controller, so OVRPlayerController in our case. The script has a private TypeText field that is initialized with the TypeText object automatically on start by searching through child components for a TypeText object. When the player moves into a trigger zone, it checks if the zone has text, if calls the StartTyping(string[] text) method. This method sets the text field on its TypeText object one line at a time and waits between lines for the player to press the button that progresses text, trigger or A.

**GrassyPlains/SnowyMountains/BarnacleWaters Scenes**

Overview

A model of these scenes, including the GameObjects required and attached scripts:

|  |  |  |
| --- | --- | --- |
| GameObject Name(s) | GameObject Description | Attached Scripts |
| \_ProgressionManager  (Must be named this) | Empty gameobject that holds script that manages piece progression. | ReliefStats |
| Terrain | A terrain the player is to explore. |  |
| GearVRPlayerController (Prefab)  Assets -> Custom Assets -> Prefabs | The object that allows a player to look around, move, and interact with puzzle pieces and scene accessors. This is the original OVRPlayerController prefab but modified and turned into the new GearVRPlayerController prefab. | ReturnToMainMenu,  PickupShape,  TerrainAccessor,  GearVRPlayerController |
| Puzzle Pieces | Treasures that the player collects to gain access to other scenes. |  |
| Scene Accessors | Portals that the player walks through to travel to other scenes. |  |

GearVRPlayerController is simply OVRPlayerController with some extra attached scripts and audio sources, but I figured it would be convenient to make it a prefab for reuse. The GearVRPlayerController object, with PickupShape, TerrainAccessor, and GearVRPlayerController scripts attached, acts as the player and allows the player to move around and look around. Every time the player picks up a treasure object, PickupShape updates the UI and the ReliefStats data class. Every time the player walks through a portal, TerrainAccessor checks the ReliefStats data class to see if the required number of treasures for access to the respective scene have been collected, and if so, loads the scene, if not it updates the UI indicating that access is not granted.

Scripts (All found in Assets -> Scripts -> Scene Scripts)

ReliefStats

Class that holds static instance of itself as a variable and is initialized whenever the GameObject holding it is first enabled, and ensures that the static instance is not destroyed on a load of a new scene, thereby maintaining data. It holds data on the maximum pieces needed to gain access to a respective scene and the current number of pieces collected for each piece type in its private fields. It has methods Increment…Piece, …PiecesCollected, HasAccessTo…, Max…Pieces, and …PiecesLeft, with … indicating the piece name. These methods allow other classes to access and update piece progression data. This class must be attached to a gameobject called \_ProgressionManager, both to indicate its special purpose and to allow it to be deleted when returning to the main menu.

PickupShape

Attached to the player object, GearVRPlayerController. Whenever the player enters a trigger on an object with the appropriate tag (snowyPiece, barnaclePiece, newPiece), it will deactivate the object, increment the number of that piece collected in the ReliefStats instance, and update the UI indicating how many pieces the player has collected.

TerrainAccessor

Attached to the player object, GearVRPlayerController. Whenever the player collides with or passes through a trigger zone with the appropriate tag (snowyTerrainAccessor, barnacleTerrainAccessor, grassyTerrainAccessor), the scene will be loaded if the number of pieces required for access have been collected, otherwise it updates the UI indicating that access is not allowed.

**GrassyPlainsSceneControlled, SnowyMountainSceneControlled, BarnacleWatersSceneControlled**

Overview

To make a guided/on rails scene mirroring the original, make a copy of the original scene using save as. Then delete Puzzle pieces/treasures, delete EnvironmentAccessors (portals), and delete \_ProgressionManager. Add in the prefab AutoMoveOVRCameraRig (Assets -> custom assets -> Prefabs) at the same transform as GearVRPlayerController. Then add the background sound audiosource from the GearVRPlayerController object to AutoMoveOVRCameraRig, as well as any other non-interactive aesthetic components attached to GearVRPlayerObject, in the three scenes listed above, the only thing I needed to add from the GearVRPlayerController object was the background audiosource. Now delete GearVRPlayerController. Now create an empty game object and name it Positions, or something similar. Under that game object, sequentially create more empty game objects with transforms that form a path through the scene, and that eventually returns to the starting point.

**Optimization**

Check out this website:

<https://developer.oculus.com/blog/squeezing-performance-out-of-your-unity-gear-vr-game/>

This gives a general overview of optimization. However, with ReliefFinally having scenes that have large landscapes requiring long lines of vision, it is impossible to achieve fewer than 100,000 draw calls per frame.

A large terrain in of itself can bring fps down to 30, so that is the baseline fps to shoot for. If you reduce lines of sight by strategically molding the terrain such that the player is never able to have extended line of sight, that can help performance. If the terrain has a lot of grass/flowers and trees on it, you can increase performance by turning detail density down (in the terrain settings tab). Also make sure that you are not spreading the grass/flowers and trees too densely, as that is hard on the processor.

Another thing to keep in mind when designing new terrains is that you can increase performance by molding the terrain so that the player never has long lines of sight. Unity’s occlusion culling makes it so that any objects that are blocked by other objects from the camera’s view are not drawn. (<https://docs.unity3d.com/560/Documentation/Manual/OcclusionCulling.html>)