CallToHaptics.cs

Notes: A simple script which calls the Directional Haptics script when an attached trigger collider is entered. Not used anymore because Directional Haptics is not in function, but could still be used if Directional Haptics is reworked.

DirectionalHaptic.cs

Notes: Code that worked before bHaptics updated their software, but currently will not run without errors. Not deleted because there is some math in the script that can be used to calculate an angle between a 'sourceObject' and the Camera (which would be the player). The code used to work by modifying the strength of each vibration pad on each device to the value of the angle between the player and the source of the vibration, **but in their newest update, bHaptics no longer allows you to modify the strength of individual vibration pads directly. It can now only be done if we modify the bHaptics source code.**

This code can be made usable again via a workaround by changing each TactSource reference to a HapticSource reference, and then changing the values of HapticSource.clip.DotPoints[] of a custom SimpleHapticClip. The problem here though is that the DotPoints array is private, and the HapticSource.clip object is of type "HapticClip", rather than "SimpleHapticClip," so one would need to change the bHaptics source code in order to support the old functionality.

As you can see, it's a bit of an obtuse method, so we transitioned over to HapticSender and HapticReceiver. However, in doing so we have made the task of directional vibration much more complicated. It can still be done in a script which references a HapticSender's clips, but again, this is finicky because you have to use the same method, but this time the clips are stored in arrays whose sizes are determined in the inspector. Hopefully bHaptics adds back in better support for this functionality in the future, and this code can be used again.

HapticManager.cs

Notes: Used to exist on a gameobject that contained a reference to each device via TactSource (a script now called HapticSource in bHaptics' most recent update), but now depreciated. If in the future bHaptics re-allows the direct modification of the strength of individual vibration pads in code, this could be used again.

RemoteVibrate.cs

Notes: Placed on an object that will emit a vibration. It can be triggered remotely via any other script (none is written at this time).

Inside the script you will see depreciated functionality to modify the strength of vibration by 1 / distance from the object emitting the vibration. This functionality no longer works because again, bHaptics updated their software and we can no longer directly modify the strength of vibration of each pad in each device.

VibrateTrigger.cs

Notes: Used to be the way that haptic devices were vibrated, before the bHaptics software was updated and the workflow changed. Now it is unusable code, and can probably be deleted. Never was because I figured that bHaptics may one day re-introduce functionality to modify each vibration pad individually, and this code could have a purpose again.

BasicMover.cs

Notes: A movement script which gets placed on an object with a trigger collider, and gets given an object to move and a list of destinations. It can move the object along the destinations in one of two ways. When useBezierCurve is false, it linearly interpolates between each pair of points in the list of points. When useBezierCurve is true, it treats each point in the list as a point in the control polygon of a bezier curve, and creates a smooth curve with them.

Along with that, a raycast is shot down from the center of the object to determine the normal vector of the terrain, and it sets the object's normal vector to that. It's height is then ½ the size of the box collider of the object + the normal vector, so in the current set-up, the object being moved either needs a box collider; or, the object being moved needs an 'empty' parent object which contains a box collider.

When useBezierCurve is false, it looks forward by looking from the last position to the current position. When useBezierCurve is true, it looks forward by looking at the next 'point' on the line using the next t-value. This leads to running the getBezierPoint twice each frame, which leaves room for improvement.

The bezier curve implementation uses De Casteljau's Algorithm, but if we wanted a more complex pathing system, one could implement de Boor's B-Spline algorithm.

CallPlayerDeath.cs

Notes: Simple script which attaches to a trigger collider. Used to test the PlayerDeathHandler script, and can potentially be deleted. When the trigger is entered, calls the StartDeathCycle() function in the PlayerDeathHandler script attached to the player.

SceneRestore.cs

Notes: Works with the PlayerDeathHandler script to create a 'death cycle.' To use the script, simply add the objects in the scene that you want to save to the objectsToSave array in the inspector, and add a respawn point for the player to return to when they die at that given scene. Then, when ResetScene() is called, it will reset all of the objects in the array to their starting positions and rotations. The respawn point variable isn't used inside this script.

There is also a SceneCompleted() function that can be called to remove the 'sceneRestore' instance from the PlayerDeathHandler script, so it will never be reset upon a player's death. This function can be called when a task is completed, to prevent the task from being reset if the player dies close to it.

PlayerDeathHandler.cs

Notes: *No longer fully works due to the switch in rendering pipelines. Should work just fine again if somebody replaces the PlayerVisionFade material, and any call to Shader.SetGlobalColor() to modify the variable contained within the new shader (if a new one is created, rather than updating the old one). If the new pipeline still allows variable manipulation, that is. *

This is a bit of a complicated script. The first thing to note is that you DO NOT need to populate the 'stateSavers' or 'respawnPoints' Lists in the inspector. They are public because of some weird internal error with Unity that prevented the AddStateToPlayer() function from working as intended.

The next thing to note is that the Update() function and the callDeathCycleDirectly bool can be removed when the script no longer needs to be tested. This isn't super important because we're only checking a bool every frame, but is still something to consider.

The main functionality lies in the StartDeathCycle() script, which will be called by an external trigger. The CallPlayerDeath script is an example of this. When called, the script will look at the list of SceneRestore respawnPoints and find the closest one to the player. It will then fade the screen to black using the shFadeOnDeathShader and CameraOverride; teleport the player to that respawn point, and call the reset function of the associated SceneRestore instance; and then finally fade the camera view back in.

Possible additions to this script would be raycasting downwards upon teleportation, to ensure that the player gets moved to the correct y coordinate. One could also add an extra 'lookAt' gameobject or float value, so that when the player respawns, they are made to look in a specific direction.

CameraOverride.cs

Notes: Used in the Player Death Handler. Overlays a material with a special shader (in this case shFadeOnDeath) onto the camera. Can be Enabled and Disabled by other scripts, so it doesn't interrupt the Camera when it is not being used.

shFadeOnDeath.shader

Notes: A shader I wrote using the default pipeline / GLSL shader code. Since we updated to the new render pipeline, it no longer works, and I didn't know how to fix it because I am unfamiliar with the new render pipeline, so needs to be re-implemented. Is used in Player Death Handler to fade the screen to black.

SimEvent.cs

Notes: 'Simulation Event' - This is a simple script that creates scriptable object events, and can be used with SimEventListener to create unique events that can 'Raise' and trigger every added listener.

This pair of scripts was made to give the option for unique, static events that can be easily modified. Scriptable Objects are very useful for this sort of thing.

SimEventListener.cs

Notes: A listener to a SimEvent. When the SimEvent is called, this will trigger a Unity Event (a custom response callback which shows up in the inspector, and allows you to target a specific function in this or another object).