# **File Structure:**

All the scripts are stored in the ./Assets/Scripts folder, most of the scripts are stored in the ./General folder, those are the scripts that we used in almost every level, such as the functionalities, the pause menu, and the physics involved in this game. We also have a folder dedicated to scripts for the AI behaviour, ./EnemyBehaviors and included subfolder for the all the scripts involved in each level.

Handling assets and asset packs using GitHub presents some challenges. One of the primary issues using Git involved large free Unity story asset packs causing minor pointer errors that caused near total collapse of our Git repo. We bypassed this risk by keeping all premade asset packs in an untracked folder (Assets/AssetPacks). Our application only uses some components from each of these premade asset packs, so for those specific files necessary to make our project portable, we dedicated a second folder where we copy those components and keep it tracked in Git (Assets/TrackedAssetPacks). This allows us to keep the premade assets in the project repo and minimize the chance for Git errors with large asset packs.

The Resources folder contains a file structure similar to the main Assets folder. The main reason for using the Resources folder is for runtime asset loading via the Resource.Load() method. This prevents us from solely relying on inspector assignments, with the assumption that the Resources file structure is unchanged. We kept the Resources folder as minimal as possible, only putting in files that were explicitly referenced in scripts. This was done to keep performance as optimal as we could, as a cluttered Resources folder causes searches to be more memory and performance intensive. We could have avoided the use of the Resources folder altogether, but we chose to include it so simple Prefab drag-and-drops into any scene work without any risk of missing components in inspector assignments.

# **Scripts:**

The general structure of our application relies on our modified version of the SteamVR prebuilt [CameraRig] prefab. We modified it by attaching our own scripts to each of the Controller GameObjects. These scripts allow us to listen in on controller input and keep track of different variables and actions. There are also scripts attached to the Camera(eye) GameObject in the [CameraRig] object to keep track of sight and collider interactions.

**FunctionController.cs**

This script is the main hub on the controller that directly listens to all controller touchpad inputs and controls the 5 different modes a controller can be in at any given time. Given a touchpad or trigger input, this script puts the controller into one of these modes, as well as rendering the icon on the controller in game. Once put into a mode, this script calls functions in the corresponding scripts attached to the controller. This scheme allows for modularity in development. Different team members can work on a different script for a different function and have the FunctionController script call its corresponding function as needed. The script also monitors the other controller to ensure functions from either the Left or Right controller don’t interfere with the other in situations where it could conflict.

**Rope.cs**

This script is dedicated to spawning climbable ropes in the game environment. It uses a raycast originating from the controller’s position and points towards the controller’s forward like a laser pointer. The raycast then checks to see if the point is either within range or climbable. If it is, on the corresponding touchpad input from FunctionController, this script spawns a node visible in the worldspace. The actual creation of ropes and management of nodes is handled in a world-scope script, “WorldRopeNodeTracker”. The use of this world-scope script is necessary to ensure both controllers can share the ability to create ropes at the same time.

**ControllerGrab.cs**

This script is designed to handle interaction with physics objects and climbing interactions. The script relies on the OnTriggerEnter() method built into Unity’s API’s in order to listen to trigger interactions on a controller. It pays particular attention to if the trigger is pulled and the type of object that is colliding with the controller at that moment (either a physics object or a climbable surface). On a valid collision and trigger pull, the colliding GameObject is stored as a reference (for physics object interaction) or the position of the controller is recorded(for climbing interactions). Methods from within this script are then called from the FunctionController script to handle those interactions as needed.

**ControllerRetract.cs**

This script works in a similar fashion to the Rope script. The only difference is in the type of GameObject it checks for (physics objects). On a valid check, this script turns off physics interactions for the object and Translates it towards the calling controller. The Translation stops when the object collides with the controller. This method of retracting the object was chosen over other, physics enabled methods due to game-breaking bugs that would prevent the object from ever reaching the controller. It causes the object to phase through other objects, which is a less than immersive interaction, but one we took over the errors.

**Fist.cs**

This script simply spawns a spherical collider around the calling controller that has special OnCollisionEnter() interactions. This collider must disable interactions with the pre-existing controller collider in order to function properly.

**Webshot.cs**

This script spawns a sphere primitive originating from in front of the controller. A special material, collision handling script, Rigidbody, and force is applied to the sphere and the Unity physics engine takes care of the trajectory and collisions. The interactions that apply upon collision of the webshot is handled in the script attached to the sphere upon instantiation.

**PauseMenu.cs**

This script handles the instantiation of the pause menu on input of the corresponding button on the controller. The pause menu is a floating plane that spawns in front of the player and can be interacted with via a laser pointer originating from the controller that called the pause menu. The actual creation of the pause menu is handled in the world-scoped script “PauseMenuWorld” that creates the pause menu and hands control of it to the controller that pressed the pause button. This script also listens to trigger inputs once the controller points to a valid spot on the pause menu. The pause menu also implements a check for enough empty space in front of the player. If a short raycast fails to hit any collider in front of the player, the pause menu will successfully spawn in the scene. Otherwise, the pause menu will fail to spawn and a fail-sound will play in game.

**HeaderColliderHandler.cs**

This script is attached to the player’s head and handles all collisions that the player can make with the world (damage collisions). Upon collision of the head with any collider in the world, damage is applied to the player. This prevents the player from phasing their head through walls, as well as handling damage interactions from enemy projectiles or hits. This script holds the HP attribute, beginning at 100 HP and taking damage from head collisions or enemy damage interactions. A red, transparent “filter” is applied to the camera in proportion to the amount of damage taken. At 0 HP, the red blur becomes opaque and the level resets, indicating player death.

**CameraIgnorePhysicsCollisions.cs**

There is another collider attached to the head of the [CameraRig] that handles Event Zone collisions, which are just colliders that cause things to happen in a level based on the player’s progress in the level. This less-than-optimally-named script simply disables Physics interactions with this collider and all other colliders using the Physics.IgnoreLayerCollision() built in Unity method. This collider and the colliders of Event Zones lie in a layer separate from the rest of the scene.

In addition to the scripts attached to the controllers, there are a number of world-scope scripts that must be implemented for use with the controller scripts. These scripts are attached to the WorldNodeTracker prefab and must be placed in the scene along with the modified [CameraRig] GameObject.

**PauseMenuWorld.cs**

This script controls the instantiation of the Pause menu, which was described in the PauseMenu script description above. This world-scope script is necessary to control which controller has control over the pause menu when spawned. Because both controllers can call the pause menu, this limitation is placed to prevent errors in spawning and despawning of the pause menu itself.

**WorldRopeNodeTracker.cs**

This script keeps track of all rope nodes present in a scene. The Rope.cs script adds its nodes to a list that is tracked in this script. Upon creation of 2 nodes, a climbable rope object is instantiated between them. This rope object is a primitive cylinder that has a special material, tag, automatic deletion coroutine, and unique transform applied to it upon instantiation. After a rope is spawned, the list is cleared and the script continues to listen for 2 nodes. Interestingly, the name of this script is the same as the name for the general world-scope GameObject prefab because it was the first world-scope script we wrote. The naming similarity in no way indicates level of importance in the application.

**EventUtil.cs**

This script serves as a global script where we store helper functions in. It is the most commonly called script in the whole project because of its utility. This script is also where an update method constantly monitors what object is immediately visible to the Camera’s head. This functions existence in this script could have been delegated to another, more relevant script, but we decided to put it here because of how frequently this script is used.

**WindowTextController.cs**

This script is attached to the Window prefab and can hold text information that the player can scroll through. For the window to have any image on it, it must have a unique material created specifically for the window, the material must be attached to the windowText child object, and a texture array must be assigned to the instance of the WindowTextController script via the UpdateArray(texture[] array) method call from elsewhere (usually an event controller script). Each window also has floating animations that can be called via the SetTrigger() method in its animator. The player can only use the grip buttons to cycle through messages if they are looking at the specific window (a Physics.SphereCast is used and updated in EventUtil.cs to determine what the player is looking at).

# **Setup:**

**General: (Upon creation of a new, empty scene)**

1. Delete Main Camera from the hierachy
2. Drag [CameraRig], [SteamVR], and WorldRopeNodeTracker into the scene from the Assets/Prefabs folder.

**Level Creation Requirements:**

* Climbable objects must have a Rigidbody, not use gravity, is kinematic, have a suitable collider, and be tagged as “Climbable”.
* Physics object (throwable) must have a Rigidbody, use gravity, not be kinematic, and have a suitable collider.
* Enemy prefabs can be drag and dropped into scene, but must have one of 3 Enemy AI scripts attached to it. Either Puncher.cs, Sentry.cs, or Hitman.cs
  + Puncher walks towards player and attempts melee attacks that do 25 damage to player
  + Sentry stands in place and shoots bullets at the player that do 10 damage
  + Hitman walks towards player and shoots a bullet every 3 seconds or so, doing 10 damage each
  + Each AI inherits from the BaseEnemy.cs script, with overridden methods determining each unique behavior
  + Each AI has a modifiable field of vision, set at a default value of a ~45 degree cone from their head. As long as the player is in this line of sight cone, the enemy will attempt attacks on the player.
* “Event Zones” that control the flow of the level can be created as an empty GameObject with only a collider of any shape, must be a trigger, be in layer “CameraZoneCollisions” (or Layer 10), and a script attached that will control actions OnTriggerEnter(). Physics interactions with these colliders will automatically disabled in the IgnoreCameraPhysicsCollisions.cs script.
* The ControllerGrab.cs script also looks out for grab interactions with objects that have a LevelBridge.cs script attached to it as a climbable object. In order to set level flow, the newLevel int attribute must be set in any Start() method as a public member.