Interaction System from The Lab

The Interaction System is a series of scripts, prefabs and other assets that was the basis of all the minigames and other scenes in The Lab.

The system was designed specifically to be lightweight and flexible so that it could support all the experiments that we were working on at the time. The system is fairly modular and you can choose to use as little or all of it in your projects.

Most of the included components were actually used in The Lab and have been tested fairly well. We have also included a few components that we didn't end up using in The Lab but you might still find useful.

In all of these cases the code was used mostly for simple things and has room for improvement in many areas.

Hopefully it can serve as a good starting point for your own experiments and help you in developing your projects.

Getting started

- Create a new scene
- Delete the “Main Camera” object
- Drag in the “Player” prefab from Core/Prefabs into your scene. This prefab sets up the main Player component and the hands. It also hooks into all the relevant SteamVR things needed.
- You should be able to see the scene in the headset now along with your controllers being tracked in the scene.
- Add the “Interactable” component to any object in the scene. All the other components on this object will then start receiving relevant messages from the player’s hands. (Look at Samples/Scripts/InteractableExample.cs for example usage)
- We have included a few commonly used interactable classes such as the Throwable. Adding this component to your object will allow it to be picked up and thrown by the player.

- To add teleporting to your scene drag in the Teleporting prefab from Teleport/Prefabs into your scene. This will set up all the teleport logic.
- Drag in some TeleportPoint prefabs from Teleport/Prefabs to add locations that the player can teleport to.
- You can also add the TeleportArea component to any object in your scene. This will allow the player to teleport anywhere along the collider for that object.

With these basic building blocks you can then go on and create some fairly complex objects. For some examples please take a look at the Interactions_Example scene in Samples/Scenes.

Sample scene

The sample scene **Interactions_Example** in the Samples/Scenes folder includes all of the major components and is a good place to familiarize yourself with the system.

The scene contains the following elements:

- **Player:** The Player prefab is at the core of the entire system. Most of the other components depend on the player to be present in the scene.
- **Teleporting:** The Teleporting prefab handles all the teleporting logic for the system.
- **InteractableExample:** This shows a very simple interactable that shows the basic aspects of receiving messages from the hands and reacting to them.
- **Throwables:** These show how to use the interaction system to create a slightly more complex object that can actually be used in a game.
- **UI & Hints:** This shows how hints are handled in the interaction system and how it can be used to interact with Unity UI widgets like buttons.
- **LinearDrive:** This is a slightly more complex interactable that combines a few different peices to create an animated object that can be controlled by simple interactions.
- **CircularDrive:** This shows how interactions can be constrained and mapped differently to cause more complex motions.
- **Longbow:** This is the actual Longbow used in The Lab. It is one of the more complex objects that we've created using this system and shows how simple peices can be combined to create an entire game mechanic.

Going over the different objects in the sample scene should give you a good idea of the breadth of the interaction system and how to combine its different parts to create complex game objects.

Documentation

This documentation will go over some of the basic components included in the interaction system.

The system is broken up into a few different parts:

Core:

- At the core of the interaction system are the **Player**, **Hand** and **Interactable** classes. The provided Player prefab sets up the player object and the SteamVR camera for the scene.
- The interaction system works by sending messages to any object that the hands interact with. These objects then react to the messages and can attach themselves to the hands if wanted.
- To make any object receive messages from the hands just add the **Interactable** component to that object. This object will then be considered when the hand does its hovering checks.
- We have also included a few commonly used interactables such as the **Throwable** or the **LinearDrive**.
- The Player prefab also creates an **InputModule** which allows the hands to mimic mouse events to easily work with Unity UI widgets.
- The interaction system also includes a fallback mode which allows for typical first-person camera controls using the keyboard and mouse. This also allows the mouse to act like one of the player's hands. This mode is particularly useful when not everyone on the team has access to VR headset.

Player

- The Player class acts like a singleton which means there should only be one Player object in the scene.
- The player itself doesn't do much except for keep track of the hands and the hmd.
- It can be accessed globally throughout the project and many aspects of the interaction system assume that the Player object always exists in the scene.
- It also keeps track of whether you are in VR mode or 2D fallback mode.
- Using the accessors through the Player class allows the other components to function similarly without knowing if the VR headset or mouse/keyboard is being used.
- The 2D fallback mode is useful but has its limitations. We mainly used this mode for testing out very simple interactions that only required 1 hand and the trigger button. It was mainly

useful during development when not everyone on the team had a VR headset on controllers with them at all times.

- The Player also includes a few useful properties:
    - **hmdTransform**: This will always return the transform of the current camera. This could be the VR headset or the 2D fallback camera.
    - **feetPositionGuess**: This guesses the position of the player's feet based on the where the hmd is. Since we don't actually know the position of their feet, this can be pretty inaccurate depending on how the player is standing.
    - **bodyDirectionGuess**: This is similar to the **feetPositionGuess** in that it can be inaccurate depending on how the player is standing.

- Note: The player class is set up to use icons to show the feet and hands in the editor scene view but due to the way Unity works these icons have to be located in a specific folder to work. These icons are provided under Core/Icons. Move them to a folder named "Gizmos" in the root of your projects Asset tree and they should work.

- The 2D fallback mode can be useful during testing but it's probably not something that you want to ship with your finished game. There are 2 ways to disable it:
    - Uncheck the "Allow Toggle To 2D" bool on the player object in your scene before making your build.
    - Add "HIDE_DEBUG_UI" to the list of Scripting Define Symbols in your project's PlayerSettings. This will only disable the 2D debug view in builds of your game while allowing you to keep using it in the editor.

Hand

- The Hand class does most of the heavy lifting for the interaction system.
- The Hand checks for objects (Interactables) that it is hovering over and sends them messages based on the current hover state.
- The hand can only hover on 1 object at a time and only 1 hand can hover on an object at the same time.
- Objects can be attached to and detached from the Hand. Only one object can be the object in focus of the hand but multiple objects can be attached to the hand at the same time.
- Once an object is detached from the hand then the previous object attached to the hand (if it is still attached) becomes the object in focus on the hand
- When nothing is attached to the hand it will always show the controller.

- The attached object can set **AttachmentFlags** that determine the behavior of the hand and the object once it has been attached.
- The hand can be locked from hovering over other objects or any object depending on the situation.

- These are the messages the hand sends to objects that it is interacting with:
  - **OnHandHoverBegin**: Sent when the hand first starts hovering over the object
  - **HandHoverUpdate**: Sent every frame that the hand is hovering over the object
  - **OnHandHoverEnd**: Sent when the hand stops hovering over the object
  - **OnAttachedToHand**: Sent when the object gets attached to the hand
  - **HandAttachedUpdate**: Sent every frame while the object is attached to the hand
  - **OnDetachedFromHand**: Sent when the object gets detached from the hand
  - **OnHandFocusLost**: Sent when an attached object loses focus because something else has been attached to the hand
  - **OnHandFocusAcquired**: Sent when an attached object gains focus because the previous focus object has been detached from the hand

- These are the messages that the hand sends to its child objects:
  - **OnHandInitialized**: Sent when the hand first gets initialized by associating itself with the device ID of a SteamVR tracked controller
  - **OnParentHandHoverBegin**: Sent when the hand starts hovering over something
  - **OnParentHandHoverEnd**: Sent when the hand stops hovering over something
  - **OnParentHandInputFocusAcquired**: Sent when the game window gains input focus
  - **OnParentHandInputFocusLost**: Sent when the game window loses input focus

- These members deal with attaching and detaching:
  - **AttachObject**: Attaches the object from the hand using the passed in AttachmentFlags
  - **DetachObject**: Detaches the object from the hand and optionally restores it to its original parent
  - **currentAttachedObject**: This returns the in-focus attached object on the hand, if any

- The Hand also has a few useful properties and functions that can be used to customize its behavior:
  - **OtherHand**: This is the other hand on the player. This can be useful for objects that need to interact with both hands such as the longbow.
  - **HoverSphereTransform** and Radius: This can be used to customize the hover range of the hand.
  - **HoverLayerMask**: This can be changed so that the hand only hovers over objects in certain layers.
  - **HoverUpdateInterval**: The hovering check can be done more or less frequently depending on the requirements of your game.

- **HoverLock/Unlock**: This is used to make the hand only hover over a certain object. Passing in null will make the hand not hover over anything while it is hover locked. This technique is used to make the hand not hover over objects while the teleport arc is active.
- **GetStandardInteractionButton/Up/Down**: These are used to check the state of the trigger on the controller. This is useful for simple objects that can also be used with the 2D fallback hand. In the 2D fallback case the left-click acts as the standard interaction button and the objects behave the same.
- **GuessCurrentHandType**: This uses some SteamVR functions to figure out which hand is leftmost and which is rightmost.
- **GetAttachmentTransform**: Objects can use "attachment transforms" on the hand to figure out how to snap on to the hand. These are just named children of the Hand object. The Player prefab contains "Attach_ControllerTip" as an example.

Interactable

- The **Interactable** class is more of identifier. It identifies to the **Hand** that this object is interactable.
- Any object with this component will receive the relevant messages from the **Hand**.

- Using just these 3 components you should be able to create many different and complex interactive objects.
- A few commonly used interactable system components that show how the system can combine these basic mechanics to create more complicated objects have been provided:

Throwable

- This is one of the most basic interactive objects.
- The player can pick up this object when a hand hovers over it and presses the interaction button (trigger).
- The object gets attached to the hand and is held there while the button is pressed.
- When the trigger is released then any velocity that was in the hand is given to thrown object.
- This lets you create basic objects that can be picked up and thrown.

LinearDrive

- This allows an object to be moved by the hand between a starting and ending position.
- The object's current position is used to set a **LinearMapping**.

CircularDrive

- This allows an object to move by the hand in a circular motion.
- The object's current position is used to set a **LinearMapping**.

LinearMapping

- This is a number that is set by a **LinearDrive** or **CircularDrive**.
- The mapping can be used to map simple hand interactions into more complex behaviors.
  - An example of this is string in the Longbow which uses a **LinearMapping** to map the pulling of the bow string to the longbow pull-back animation.
- The mapping is used by several other classes to interpolate their properties
  - **LinearAnimation**
  - **LinearAnimator**
  - **LinearBlendShape**
  - **LinearDisplacement**
  - **HapticRack**

VelocityEstimator

- This class is useful for estimating the velocity and acceleration of an object based on a change in its position.
- In most cases you would get more accurate results if you get the velocity and acceleration from the actual controller but sometimes that isn't possible such as when using the 2D fallback hand.

IgnoreHovering

- This can be added to an object or specific collider if you want it to be ignored by the hand when doing its hovering checks.

UIElement

- Adding this component to an existing will make it so that the hands can interact with it.
- This will generate mouse hover and click events based on hand interactions and send them through the Unity event system to work with existing UI widgets.
- In addition it will also generate an **OnHandClick** event which will also pass in the hand that clicked the element.

- Another big part of the interaction system is the concept of an **ItemPackage**

ItemPackage

- An **ItemPackage** is collection of objects that are meant to temporarily override the functionality of the hand.
  - An example of this is the **Longbow**. While the Longbow is attached to the hand it sort of takes over the base functionality of the hand.
- **ItemPackages** have the concept of being able to be picked up and put back where they were picked up from.
- Once picked up they remain attached to the hand until they are put back. No button needs to be held for them to remain attached to the hand. The hand still passes along messages like normal but these objects usually disable some of the base functionality of the hand, such as hovering while they are attached.
  - Other examples of an **ItemPackage** from The Lab would be the balloon tool or the Xortex drone controller. Both these objects take over the hand's base functionality while they are attached.
- An **ItemPackage** can be 1 or 2 handed.

ItemPackageSpawner

- This handles the logic for when to spawn and put away the **ItemPackage** and how to attach the items to the hand once spawned.
- It also handles showing the preview of the item or the outline of the item when it is picked up.

ItemPackageReference

- This component can be added to item to indicate that it is a part of an item package.

- There are a few other helper classes included as a core part of the interaction system.

PlaySound

- This class allows **AudioClips** to be played back with many more parameters
- It can take in many **AudioClips** and play back 1 at random each time
- It can also randomize how the clip is played back

SoundPlayOneShot

- This class is specifically for sounds that only play once and don't loop or need to be paused while playing

Util

- This is a class full of small utility functions that are used throughout the interaction system.

InteractableHoverEvents

- This class generates UnityEvents when it receives messages from the hand.

InteractableButtonEvents

- This class translates controller button input into UnityEvents.

ComplexThrowable

- This class attaches objects to the hand using a physics joint instead of simple parenting.
- This allows for more physics based interactions with the object once it is attached.
- Note: This class is a little experimental and since we didn't actually use it in The Lab it might not be feature complete and have bugs that would need to be fixed.

DistanceHaptics

- Triggers haptic pulses based on a distance between 2 transforms.

Player (Prefab)

- This is the single piece of the interaction system that combines all its basic parts.
- This prefab arranges the player and hands in a way to make them all accessible easily.
- It also contains all the setup for SteamVR and the 2D fallback system
- Most of the other components of the interaction system depend on the player and some of them assume that the player and hands are set up in this way.
- There should only be 1 of these in a scene.

BlankController (Prefab)

- This is used by the **Hand** when it has nothing else attached.
- The render model for the controller is loaded through SteamVR and all its parts are articulated.

Teleport

- The teleport system from The Lab supports teleporting to specific teleport points or a more general teleport area.
- The important classes are **Teleport**, **TeleportPoint** and **TeleportArea**.
- All the functionality is wrapped up in the Teleporting prefab in Teleport/Prefabs. This prefab includes all the logic for the teleport system to function.
- Add **TeleportPoints** or **TeleportAreas** to the scene to add spots where the player can teleport to.

Teleport

- This class handles most of the logic of teleporting.
- When the touchpad is pressed, the teleport pointer shows up. If the pointer is pointing at a valid spot when the touchpad is released then the player teleports.
    - You can also press 'T' on the keyboard while in 2D fallback mode to bring up the teleport pointer.
- There is a slight fade to black when the player teleports and then the game fades back in.
- This class keeps track of all the teleport markers in the scene and informs them to fade in/out depending on the state of the teleport pointer.
- In certain situations it can be useful to have a separate mesh for the floor the scene that is different from the teleport mesh. In these situations the teleport system will trace down from where it hit the teleport mesh and try to place the player on the floor mesh. The point of this is to try to match the visual floor in the scene with the physical floor in the player's play area.
- There are a few properties that will probably need to be tweaked:
    - **tracerLayerMask**: This is all the layers that the teleport pointer will try to hit
    - **floorFixupMask**: The layer that the floor is on.
    - **floorFixupMaximumTraceDistance**: The maximum distance to trace to try to look for the floor.
    - **ShowPlayAreaMarker**: This toggles whether to show the rectangle of the player's play area while teleporting. This can help in orienting the players in their physical space.
    - **arcDistance**: How far the teleport arc should go. Increasing this number will allow the player to teleport further in the scene. This value will probably need to be tweaked for each scene.

TeleportMarkerBase

- This is the base class for all the teleport markers.
- It contains methods that the **Teleport** class expects to be present in all the teleport markers.
- You can use this as your base class to create a new type of teleport marker.
- A teleport marker can be locked or unlocked. The player cannot teleport to locked markers.

TeleportArea

- This is a teleport area that is made up of a mesh.
- When teleporting onto these, the player will teleport exactly where they are pointing (plus the floor fixup)
- Add this component to any object with a collider and a mesh renderer to allow the player to teleport on it.

TeleportPoint

- This is a teleport point that the player can teleport to.
- When teleporting onto these, the player will teleport at the origin of the point regardless of where on the point they were pointing.
- These points can be named
- The points also have the ability to teleport players to new scenes. (This isn't fully functional since you will have to hook it up to your scene loading system.)

TeleportArc

- This draws the arc for the teleport pointer and does the physics trace for the teleport system.

AllowTeleportWhileAttachedToHand

- By default you can't teleport using a hand that has something attached to it. Adding this component to an attached object bypasses that rule.
- This is used by the **BlankController** and longbow **Arrow** objects so that the player can teleport even while they are attached to the hand.

IgnoreTeleportTrace

- Adding this to an object with a collider will allow the teleport trace to pass through it.
- A different way of dealing with this would be to put that object on a different layer that the **TeleportArc** doesn't check against.
- This is used on the longbow **Arrow** to allow the teleport trace to pass through the arrow tip.

Teleporting (Prefab)

- This prefab sets up the entire teleport system.
- Dragging this into your scene will give you the ability to bring up the teleport pointer in your game.
- All of the visuals and sounds of the teleport system can be changed by modifying the properties of this prefab.

TeleportPoint (Prefab)

- Add these to your scene to add locations that the player can teleport to.
- Note: The names of some of the objects in this scene are hard-coded and some of the code will need to be modified if you want to change the models.

Hints

- The hint system shows hints on the controllers.
- The hints are set up in a way where each button on the controller can be called out separately.
- There is also the ability to show text hints associated with each button.

ControllerButtonHints

- The hints are set up based on the render model of the controller.
- SteamVR provides a mapping from render model components to button IDs. This mapping is used to figure out what part of the controller corresponds to which button.
- Once a hint for a button is activated, that button will keep flashing on the controller model until the hint is turned off.
- Hints can be for buttons only or with an optional text hint associated with the button.
- There are a few static methods that are used to interface with the hint system:
    - **ShowButtonHint**: Flashes the specified button on the specified hand.
    - **HideButtonHint**: Stops flashing the specified button on the specified hand.
    - **HideAllButtonHints**: Stops flashing all the buttons on the specified hand.
    - **IsButtonHintActive**: Checked if a specified button is flashing on the specified hand.

    - **ShowTextHint**: Shows a text hint with the passed in string associated with the specified button on the specified hand.
    - **HideTextHint**: Hides the text hint for the specified hand on the specified button.
    - **HideAllTextHints**: Hides all the text hints that are currently active on the specified hand.
    - **GetActiveHintText**: Gets the active hint text for a specified button.

Longbow

- The Longbow is an example of a complex game mechanic created using the interaction system.
- The version included here is exactly the same one that we shipped in The Lab, including all the models, materials and sounds.
- The Longbow is built using the **ItemPackage** system. It is comprised of the **LongbowItemPackage** prefab which spawns the Longbow prefab in the main hand and the **ArrowHand** prefab in the other hand.
- The **ArrowHand** prefab then spawns a new arrow in the hand every time one is fired.

- All of the bow and arrow logic is present in the following scripts:

Longbow

- It handles the logic of how the bow controls in nocked and un-nocked modes
- It also keeps track of how far the bow string is pulled

ArrowHand

- Handles nocking and firing the arrow based on its position and the controller buttons
- Handles spawning an arrow in the hand when needed

Arrow

- The actual arrow that gets fired
- This script handles all the in-flight logic for the arrow including collision detection and deciding when to stick in to targets

ArrowheadRotation

- Rotates the arrowhead randomly every time a new arrow is spawned

SoundBowClick

- Plays the sounds of the bow string being pulled.

- The other scripts in the Longbow folder handle the logic for the longbow targets

ArcheryTarget

- This is the script for a generic archery target.
- It invokes a **UnityEvent** when hit by an arrow.

FireSource

- Indicates an object that can be set on fire. Once on fire this object can then spread fire when it comes in contact with another **FireSource**.

ExplosionWobble

- Used to make the weeble wobble.

Balloon

BalloonColliders

BalloonHapticBump

BalloonSpawner

- These scripts handle the logic for the balloons that spawn when the weeble is hit with an arrow.

Samples

- There are a few classes that were created specifically to show some examples in the sample scene.

ControllerHintsExample

- This class shows how to use the hint system.

InteractableExample

- This class shows a very simple example of receiving and responding to messages from the hand.