SteamVR Unity Toolkit - V2.0.0

A collection of useful scripts and prefabs for building SteamVR titles in Unity 5. Full source on Github - https://github.com/thestonefox/SteamVR_Unity_Toolkit

This Toolkit requires the SteamVR Plugin from the Unity Asset Store to be imported into your Unity project.

Quick Start

- Import the SteamVR Plugin from the Unity Asset Store
- Browse the `Examples` scenes for example usage of the scripts

FAQ/Troubleshooting

- How to create a new project using this toolkit along with the SteamVR Unity Plugin:
  - View answer video on YouTube
- Pointer beams/teleporting no longer works after a project build and running from that build:
  - View answer video on YouTube

Summary

This toolkit provides many common VR functionality within Unity3d such as (but not limited to):

- Controller button events with common aliases
- Controller world pointers (e.g. laser pointers)
- Player teleportation
- Grabbing/holding objects using the controllers
- Interacting with objects using the controllers
- Transforming game objects into interactive UI elements

The toolkit is heavily inspired and based upon the SteamVR Plugin for Unity3d Github Repo.

The reason this toolkit exists is because I found the SteamVR plugin to contain confusing to use or broken code and I decided to build a collection of scripts/assets that I would find useful when building for VR within Unity3d.

What's In The Box

This toolkit project is split into three main sections:

- Prefabs - `SteamVR_Unity_Toolkit/Prefabs/`
- Scripts - `SteamVR_Unity_Toolkit/Scripts/`
- Examples - `SteamVR_Unity_Toolkit/Examples/`

The `SteamVR_Unity_Toolkit` directory is where all of the relevant files are kept and this directory can be simply copied over to an existing project. The `Examples` directory contains useful scenes showing the `SteamVR_Unity_Toolkit` in action.

Prefabs

The available Prefabs are:

- `FramesPerSecondCanvas`
- `ObjectTooltip`
- `ControllerTooltips`
- `RadialMenu`

FramesPerSecondCanvas

This canvas adds a frames per second text element to the headset. To use the prefab it must be placed into the scene then the headset camera needs attaching to the canvas:

- Select `FramesPerSecondCanvas` object from the scene objects
- Find the `Canvas` component
- Set the `Render Camera` parameter to `Camera (eye)` which can be found in the `[CameraRig]` prefab.

There are a number of parameters that can be set on the Prefab. Expanding the `FramesPerSecondCanvas` object in the hierarchy view shows the child `FramesPerSecondText` obejct and clicking on that reveals additional paramters which can be set via the `FramesPerSecondViewer` script (which can be found in `SteamVR_Unity_Toolkit/Scripts/Helper/FramesPerSecondViewer`)

The following script parameters are available:

- **Display FPS:** Toggles whether the FPS text is visible.
- **Target FPS:** The frames per second deemed acceptable that is used as the benchmark to change the FPS text colour.
- **Font Size:** The size of the font the FPS is displayed in.
- **Position:** The position of the FPS text within the headset view.
- **Good Color:** The colour of the FPS text when the frames per second are within reasonable limits of the Target FPS.
- **Warn Color:** The colour of the FPS text when the frames per second are falling short of reasonable limits of the Target FPS.
- **Bad Color:** The colour of the FPS text when the frames per second are at an unreasonable level of the Target FPS.

An example of the `FramesPerSecondCanvas` Prefab can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/018_CameraRig_FramesPerSecondCounter` which displays the frames per second in the centre of the headset view. Pressing the trigger generates a new sphere and pressing the touchpad generates ten new spheres. Eventually when lots of spheres are present the FPS will drop and demonstrate the prefab.

ObjectTooltip

This adds a UI element into the World Space that can be used to provide additional information about an object by providing a piece of text with a line drawn to a destination point.

There are a number of parameters that can be set on the Prefab which are provided by the `SteamVR_Unity_Toolkit/Scripts/VRTK_ObjectTooltip` script which is applied to the prefab.

The following script parameters are available:

- **Display Text:** The text that is displayed on the tooltip.
- **Font Size:** The size of the text that is displayed.
- **Container Size:** The size of the tooltip container where `x = width` and `y = height`.
- **Draw Line From:** An optional transform of where to start drawing the line from. If one is not provided the the centre of the tooltip is used for the initial line position.
- **Draw Line To:** A transform of another object in the scene that a line will be drawn from the tooltip to, this helps denote what the tooltip is in relation to. If no transform is provided and the tooltip is a child of another object, then the parent object's transform will be used as this destination position.
- **Line Width:** The width of the line drawn between the tooltip and the destination transform.
- **Font Color:** The colour to use for the text on the tooltip.
- **Container Color:** The colour to use for the background container of the tooltip.
- **Line Color:** The colour to use for the line drawn between the tooltip and the destination transform.

An example of the `ObjectTooltip` Prefab can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/029_Controller_Tooltips` which displays two cubes that have an object tooltip added to them along with tooltips that have been added to the controllers.

ControllerTooltips

This adds a collection of Object Tooltips to the Controller that give information on what the main controller buttons may do. To add the prefab, it just needs to be added as a child of the relevant controller e.g. `[CameraRig]/Controller (right)` would add the controller tooltips to the right controller.

There are a number of parameters that can be set on the Prefab which are provided by the `SteamVR_Unity_Toolkit/Scripts/VRTK_ControllerTooltips` script which is applied to the prefab.

The following script parameters are available:

- **Trigger Text:** The text to display for the trigger button action.
- **Grip Text:** The text to display for the grip button action.
- **Touchpad Text:** The text to display for the touchpad action.
- **App Menu Text:** The text to display for the application menu button action.
- **Tip Background Color:** The colour to use for the tooltip background container.

- **Tip Text Color:** The colour to use for the text within the tooltip.
- **Tip Line Color:** The colour to use for the line between the tooltip and the relevant controller button.
- **Trigger:** The transform for the position of the trigger button on the controller (this is usually found in `Model/trigger/attach`.
- **Grip:** The transform for the position of the grip button on the controller (this is usually found in `Model/lgrip/attach`.
- **Touchpad:** The transform for the position of the touchpad button on the controller (this is usually found in `Model/trackpad/attach`.
- **App Menu:** The transform for the position of the app menu button on the controller (this is usually found in `Model/button/attach`.

If the transforms for the buttons are not provided, then the script will attempt to find the attach transforms on the default controller model in the `[CameraRig]` prefab.

An example of the `ControllerTooltips` Prefab can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/029_Controller_Tooltips` which displays two cubes that have an object tooltip added to them along with tooltips that have been added to the controllers.

RadialMenu

This adds a UI element into the world space that can be dropped into a Controller object and used to create and use Radial Menus from the touchpad.

There are a number of parameters that can be set on the Prefab which are provided by the `SteamVR_Unity_Toolkit/Scripts/Controls/2D/RadialMenu/RadialMenu.cs` script which is applied to the `Panel` child of the prefab.

The following script parameters are available:

- **Buttons:** Array of Buttons where you define the interactive buttons you want to be displayed as part of the radial menu. Each Button has the following properties:
  - **ButtonIcon:** Icon to use inside the button arc (should be circular).
  - **OnClick():** Methods to invoke when the button is clicked.
  - **OnHold():** Methods to invoke each frame while the button is held down.
- **Button Prefab:** The base for each button in the menu, by default set to a dynamic circle arc that will fill up a portion of the menu.
- **Button Thickness:** Percentage of the menu the buttons should fill, 1.0 is a pie slice, 0.1 is a thin ring.
- **Button Color:** The background color of the buttons, default is white.
- **Offset Distance:** The distance the buttons should move away from the center. This creates space between the individual buttons.
- **Offset Rotation:** The additional rotation of the Radial Menu.
- **Rotate Icons:** Whether button icons should rotate according to their arc or be vertical compared to the controller.

- **Icon Margin:** The margin in pixels that the icon should keep within the button.
- **Hide On Release:** Whether the buttons should be visible when not in use.
- **Menu Buttons:** The actual GameObjects that make up the radial menu.
- **Regenerate Buttons:** Button to force regeneration of the radial menu in the editor.

If the RadialMenu is placed inside a controller, it will automatically find a `VRTK_ControllerEvents` in its parent to use at the input. However, a `VRTK_ControllerEvents` can be defined explicitly by setting the `Events` parameter of the `Radial Menu Controller` script also attached to the prefab.

An example of the `RadialMenu` Prefab can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/030_Controls_RadialTouchpadMenu`, which displays a radial menu for each controller. The left controller uses the `Hide On Release` variable, so it will only be visible if the left touchpad is being touched.

Scripts

This directory contains all of the toolkit scripts that add VR functionality to Unity.

The current available scripts are:

Controller Actions (VRTK_ControllerActions)

The Controller Actions script provides helper methods to deal with common controller actions. The following public methods are available:

- **IsControllerVisible():** returns true is the controller model is visible, returns false if it is not visible.
- **ToggleControllerModel(bool on):** sets the visibility of the controller model to the given boolean state. If true is passed then the controller model is displayed, if false is passed then the controller model is hidden.
- **TriggerHapticPulse(int duration, ushort strength):** initiates the controller to begin vibrating for the given tick duration provided in the first parameter at a vibration intensity given as the strength parameter. The max strength that can be provided is 3999, any number higher than that will be capped.

An example of the `VRTK_ControllerActions` script can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/016_Controller_HapticRumble` which demonstrates the ability to hide a controller model and make the controller vibrate for a given length of time at a given intensity.

Controller Events (VRTK_ControllerEvents)

The Controller Events script is attached to a Controller object within the `[CameraRig]` prefab and provides event listeners for every button press on the controller (excluding the System Menu button as this cannot be overriden and is always used by Steam).

When a controller button is pressed, the script emits an event to denote that the button has been pressed which allows other scripts to listen for this event without needing to implement any

controller logic.

The script also has a public boolean pressed state for the buttons to allow the script to be queried by other scripts to check if a button is being held down.

When a controller button is released, the script also emits an event denoting that the button has been released.

The controller touchpad has two states, it can either be `touched` where the user simply presses their finger on the pressure sensitive pad or it can be `clicked` where the user presses down on the pad until it makes a clicking sound.

The Controller Events script deals with both touchpad touch and click events separately.

There are two button axis on the controller:

- Touchpad touch position, which has an x and y value depending on where the touchpad is currently being touched.
- Trigger button, which has an x value depending on how much the trigger button is being depressed.

There are two additional events emitted when either the Touchpad axis or the Trigger axis change their value which can be used to determine the change in either of the axis for finer control such as using the Touchpad to move a character, or knowing the pressure that the trigger is being pressed.

The Touchpad Axis is reported via the `TouchpadAxis` payload variable which is updated on any Controller Event.

The Trigger Axis is reported via the `buttonPressure` payload variable which is updated on any Controller Event. Any other button press will report a button pressure of 1 or 0 as all other buttons are digital (they are either clicked or not clicked) but because the Trigger is analog it will report a varying button pressure.

The amount of fidelity in the changes on the axis can be determined by the `axisFidelity` parameter on the script, which is defaulted to 1. Any number higher than 2 will probably give too sensitive results.

The event payload that is emitted contains:

- **controllerIndex:** The index of the controller that was used.
- **buttonPressure:** A float between 0f and 1f of the amount of. pressure being applied to the button pressed.
- **touchpadAxis:** A Vector2 of the position the touchpad is touched at.
- **touchpadAngle:** A float that shows the rotational position the touchpad is being touched at, 0 being top, 180 being bottom and all other angles accordingly.

There are also common action aliases that are emitted when controller buttons are pressed. These

action aliases can be mapped to a preferred controller button. The aliases are:

- **Toggle Pointer:** Common action of turning a laser pointer on/off
- **Toggle Grab:** Common action of grabbing game objects
- **Toggle Use:** Common action of using game objects
- **Toggle Menu:** Common action of bringing up an in-game menu

Each of the above aliases can have the preferred controller button mapped to their usage by selecting it from the drop down on the script parameters window.

When the set button is pressed it will emit the actual button event as well as an additional event that the alias is "On". When the set button is released it will emit the actual button event as well as an additional event that the alias button is "Off".

Listening for these alias events rather than the actual button events means it's easier to customise the controller buttons to the actions they should perform.

An example of the `VRTK_ControllerEvents` script can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/002_Controller_Events` and code examples of how the events are utilised and listened to can be viewed in the script `SteamVR_Unity_Toolkit/Examples/Resources/Scripts/VRTK_ControllerEvents_ListenerExample.cs`

Simple Laser Pointer (VRTK_SimplePointer)

The Simple Pointer emits a coloured beam from the end of the controller to simulate a laser beam. It can be useful for pointing to objects within a scene and it can also determine the object it is pointing at and the distance the object is from the controller the beam is being emitted from.

The laser beam is activated by default by pressing the `Touchpad` on the controller. The event it is listening for is the `AliasPointer` events so the pointer toggle button can be set by changing the `Pointer Toggle` button on the `VRTK_ControllerEvents` script parameters.

The Simple Pointer script can be attached to a Controller object within the `[CameraRig]` prefab and the Controller object also requires the `VRTK_ControllerEvents` script to be attached as it uses this for listening to the controller button events for enabling and disabling the beam. It is also possible to attach the Simple Pointer script to another object (like the `[CameraRig]/Camera (head)`) to enable other objects to project the beam. The controller parameter must be entered with the desired controller to toggle the beam if this is the case.

The following script parameters are available:

- **Enable Teleport:** If this is checked then the teleport flag is set to true in the Destination Set event so teleport scripts will know whether to action the new destination. This allows controller beams to be enabled on a controller but never trigger a teleport (if this option is unchecked).
- **Controller:** The controller that will be used to toggle the pointer. If the script is being applied onto

a controller then this parameter can be left blank as it will be auto populated by the controller the script is on at runtime.
- **Pointer Material:** The material to use on the rendered version of the pointer. If no material is selected then the default `WorldPointer` material will be used.
- **Pointer Hit Color:** The colour of the beam when it is colliding with a valid target. It can be set to a different colour for each controller.
- **Pointer Miss Color:** The colour of the beam when it is not hitting a valid target. It can be set to a different colour for each controller.
- **Show Play Area Cursor:** If this is enabled then the play area boundaries are displayed at the tip of the pointer beam in the current pointer colour.
- **Play Area Cursor Dimensions:** Determines the size of the play area cursor and collider. If the values are left as zero then the Play Area Cursor will be sized to the calibrated Play Area space.
- **Handle Play Area Cursor Collisions:** If this is ticked then if the play area cursor is colliding with any other object then the pointer colour will change to the `Pointer Miss Color` and the `WorldPointerDestinationSet` event will not be triggered, which will prevent teleporting into areas where the play area will collide.
- **Pointer Visibility:** Determines when the pointer beam should be displayed:
  - `On_When_Active` only shows the pointer beam when the Pointer button on the controller is pressed.
  - `Always On` ensures the pointer beam is always visible but pressing the Pointer button on the controller initiates the destination set event.
  - `Always Off` ensures the pointer beam is never visible but the destination point is still set and pressing the Pointer button on the controller still initiates the destination set event.
- **Activate Delay:** The time in seconds to delay the pointer beam being able to be active again. Useful for preventing constant teleportation.
- **Pointer Thickness:** The thickness and length of the beam can also be set on the script as well as the ability to toggle the sphere beam tip that is displayed at the end of the beam (to represent a cursor).
- **Pointer Length:** The distance the beam will project before stopping.
- **Show Pointer Tip:** Toggle whether the cursor is shown on the end of the pointer beam.
- **Layers To Ignore:** The layers to ignore when raycasting.

The Simple Pointer object extends the `VRTK_WorldPointer` abstract class and therefore emits the same events and payload.

An example of the `VRTK_SimplePointer` script can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/003_Controller_SimplePointer` and code examples of how the events are utilised and listened to can be viewed in the script `SteamVR_Unity_Toolkit/Examples/Resources/Scripts/VRTK_ControllerPointerEvents_ListenerExample.cs`

Bezier Curve Laser Pointer (VRTK_BezierPointer)

The Bezier Pointer emits a curved line (made out of game objects) from the end of the controller to

a point on a ground surface (at any height). It is more useful than the Simple Laser Pointer for traversing objects of various heights as the end point can be curved on top of objects that are not visible to the user.

The laser beam is activated by default by pressing the `Touchpad` on the controller. The event it is listening for is the `AliasPointer` events so the pointer toggle button can be set by changing the `Pointer Toggle` button on the `VRTK_ControllerEvents` script parameters.

The Bezier Pointer script can be attached to a Controller object within the `[CameraRig]` prefab and the Controller object also requires the `VRTK_ControllerEvents` script to be attached as it uses this for listening to the controller button events for enabling and disabling the beam. It is also possible to attach the Bezier Pointer script to another object (like the `[CameraRig]/Camera (head)`) to enable other objects to project the beam. The controller parameter must be entered with the desired controller to toggle the beam if this is the case.

The following script parameters are available:

- **Enable Teleport:** If this is checked then the teleport flag is set to true in the Destination Set event so teleport scripts will know whether to action the new destination. This allows controller beams to be enabled on a controller but never trigger a teleport (if this option is unchecked).
- **Controller:** The controller that will be used to toggle the pointer. If the script is being applied onto a controller then this parameter can be left blank as it will be auto populated by the controller the script is on at runtime.
- **Pointer Material:** The material to use on the rendered version of the pointer. If no material is selected then the default `WorldPointer` material will be used.
- **Pointer Hit Color:** The colour of the beam when it is colliding with a valid target. It can be set to a different colour for each controller.
- **Pointer Miss Color:** The colour of the beam when it is not hitting a valid target. It can be set to a different colour for each controller.
- **Show Play Area Cursor:** If this is enabled then the play area boundaries are displayed at the tip of the pointer beam in the current pointer colour.
- **Play Area Cursor Dimensions:** Determines the size of the play area cursor and collider. If the values are left as zero then the Play Area Cursor will be sized to the calibrated Play Area space.
- **Handle Play Area Cursor Collisions:** If this is ticked then if the play area cursor is colliding with any other object then the pointer colour will change to the `Pointer Miss Color` and the `WorldPointerDestinationSet` event will not be triggered, which will prevent teleporting into areas where the play area will collide.
- **Pointer Visibility:** Determines when the pointer beam should be displayed:
  - `On_When_Active` only shows the pointer beam when the Pointer button on the controller is pressed.
  - `Always On` ensures the pointer beam is always visible but pressing the Pointer button on the controller initiates the destination set event.
  - `Always Off` ensures the pointer beam is never visible but the destination point is still set and pressing the Pointer button on the controller still initiates the destination set event.

- **Activate Delay:** The time in seconds to delay the pointer beam being able to be active again. Useful for preventing constant teleportation.
- **Pointer Length:** The length of the projected forward pointer beam, this is basically the distance able to point from the controller potiion.
- **Pointer Density:** The number of items to render in the beam bezier curve. A high number here will most likely have a negative impact of game performance due to large number of rendered objects.
- **Show Pointer Cursor:** A cursor is displayed on the ground at the location the beam ends at, it is useful to see what height the beam end location is, however it can be turned off by toggling this.
- **Pointer Cursor Radius:** The size of the ground pointer cursor, This number also affects the size of the objects in the bezier curve beam. The larger the raduis, the larger the objects will be.
- **Beam Curve Offset:** The amount of height offset to apply to the projected beam to generate a smoother curve even when the beam is pointing straight.
- **Custom Pointer Tracer:** A custom Game Object can be applied here to use instead of the default sphere for the beam tracer. The custom Game Object will match the rotation of the controller.
- **Custom Pointer Cursor:** A custom Game Object can be applied here to use instead of the default flat cylinder for the pointer cursor.
- **Layers To Ignore:** The layers to ignore when raycasting.

The Bezier Pointer object extends the `VRTK_WorldPointer` abstract class and therefore emits the same events and payload.

An example of the `VRTK_BezierPointer` script can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/009_Controller_BezierPointer` which is used in conjunction with the Height Adjust Teleporter shows how it is possible to traverse different height objects using the curved pointer without needing to see the top of the object.

Another example can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/012_Controller_PointerWithAreaCollision` that shows how a Bezier Pointer with the Play Area Cursor and Collision Detection enabled can be used to traverse a game area but not allow teleporting into areas where the walls or other objects would fall into the play area space enabling the user to enter walls.

The bezier curve generation code is in another script located at `SteamVR_Unity_Toolkit/Scripts/Helper/CurveGenerator.cs` and was heavily inspired by the tutorial and code from Catlike Coding.

Unity UI Pointer (VRTK_UIPointer)

The UI Pointer provides a mechanism for interacting with Unity UI elements on a world canvas. The UI Pointer can be attached to any game object the same way in which a World Pointer can be and the UI Pointer also requires a controller to initiate the pointer activation and pointer click states.

The simplest way to use the UI Pointer is to attach the script to a game controller within the `[CameraRig]` along with a Simple Pointer as this provides visual feedback as to where the UI ray is

pointing.

The UI pointer is activated via the `Pointer` alias on the `Controller Events` and the UI pointer click state is triggered via the `Use` alias on the `Controller Events` to make things consistent with the standard World Pointers.

The following script parameters are available:

- **Controller:** The controller that will be used to toggle the pointer. If the script is being applied onto a controller then this parameter can be left blank as it will be auto populated by the controller the script is on at runtime.

An example of the `VRTK_UIPointer` script can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/034_Controls_InteractingWithUnityUI`. The scene uses the `VRTK_UIPointer` script on the right Controller to allow for the interaction with Unity UI elements using a Simple Pointer beam. The left Controller controls a Simple Pointer on the headset to demonstrate gaze interaction with Unity UI elements.

Basic Teleporter (VRTK_BasicTeleport)

The basic teleporter updates the `[CameraRig]` x/z position in the game world to the position of a World Pointer's tip location which is set via the `WorldPointerDestinationSet` event. The y position is never altered so the basic teleporter cannot be used to move up and down game objects as it only allows for travel across a flat plane.

The Basic Teleport script is attached to the `[CameraRig]` prefab and requires an implementation of the WorldPointer script to be attached to another game object (e.g. VRTK_SimplePointer attached to the Controller object).

The following script parameters are available:

- **Blink Transition Speed:** The fade blink speed can be changed on the basic teleport script to provide a customised teleport experience. Setting the speed to 0 will mean no fade blink effect is present. The fade is achieved via the `SteamVR_Fade.cs` script in the SteamVR Unity Plugin scripts.
- **Distance Blink Delay:** A range between 0 and 32 that determines how long the blink transition will stay blacked out depending on the distance being teleported. A value of 0 will not delay the teleport blink effect over any distance, a value of 32 will delay the teleport blink fade in even when the distance teleported is very close to the original position. This can be used to simulate time taking longer to pass the further a user teleports. A value of 16 provides a decent basis to simulate this to the user.
- **Headset Position Compensation:** If this is checked then the teleported location will be the position of the headset within the play area. If it is unchecked then the teleported location will always be the centre of the play area even if the headset position is not in the centre of the play area.
- **Ignore Target With Tag Or Class:** A string that specifies an object Tag or the name of a Script attached to an obejct and notifies the teleporter that the destination is to be ignored so the user

cannot teleport to that location. It also ensure the pointer colour is set to the miss colour.
- **Limit To Nav Mesh:** If this is checked then teleporting will be limited to the bounds of a baked NavMesh. If the pointer destination is outside the NavMesh then it will be ignored.

An example of the `VRTK_BasicTeleport` script can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/004_CameraRig_BasicTeleport`. The scene uses the `VRTK_SimplePointer` script on the Controllers to initiate a laser pointer by pressing the `Touchpad` on the controller and when the laser pointer is deactivated (release the `Touchpad`) then the user is teleported to the location of the laser pointer tip.

Height Adjustable Teleporter (VRTK_HeightAdjustTeleport)

The height adjust teleporter extends the basic teleporter and allows for the y position of the `[CameraRig]` to be altered based on whether the teleport location is on top of another object.

Like the basic teleporter the Height Adjust Teleport script is attached to the `[CameraRig]` prefab and requires a World Pointer to be available.

The following script parameters are available:

- **Blink Transition Speed:** The fade blink speed on teleport.
- **Distance Blink Delay:** A range between 0 and 32 that determines how long the blink transition will stay blacked out depending on the distance being teleported. A value of 0 will not delay the teleport blink effect over any distance, a value of 32 will delay the teleport blink fade in even when the distance teleported is very close to the original position. This can be used to simulate time taking longer to pass the further a user teleports. A value of 16 provides a decent basis to simulate this to the user.
- **Headset Position Compensation:** If this is checked then the teleported location will be the position of the headset within the play area. If it is unchecked then the teleported location will always be the centre of the play area even if the headset position is not in the centre of the play area.
- **Ignore Target With Tag Or Class:** A string that specifies an object Tag or the name of a Script attached to an obejct and notifies the teleporter that the destination is to be ignored so the user cannot teleport to that location. It also ensure the pointer colour is set to the miss colour.
- **Limit To Nav Mesh:** If this is checked then teleporting will be limited to the bounds of a baked NavMesh. If the pointer destination is outside the NavMesh then it will be ignored.
- **Play Space Falling:** Checks if the user steps off an object into a part of their play area that is not on the object then they are automatically teleported down to the nearest floor.

The `Play Space Falling` option also works in the opposite way that if the user's headset is above an object then the user is teleported automatically on top of that object, which is useful for simulating climbing stairs without needing to use the pointer beam location. If this option is turned off then the user can hover in mid air at the same y position of the object they are standing on.

An example of the `VRTK_HeightAdjustTeleport` script can be viewed in the scene

`SteamVR_Unity_Toolkit/Examples/007_CameraRig_HeightAdjustTeleport`. The scene has a collection of varying height objects that the user can either walk up and down or use the laser pointer to climb on top of them.

Another example can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/010_CameraRig_TerrainTeleporting` which shows how the teleportation of a user can also traverse terrain colliders.

Another example can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/020_CameraRig_MeshTeleporting` which shows how the teleportation of a user can also traverse mesh colliders.

Fading On Headset Collision (VRTK_HeadsetCollisionFade)

The purpose of the Headset Collision Fade is to detect when the user's VR headset collides with another game object and fades the screen to a solid colour. This is to deal with a user putting their head into a game object and seeing the inside of the object clipping, which is an undesired effect.

The reasoning behind this is if the user puts their head where it shouldn't be, then fading to a colour (e.g. black) will make the user realise they've done something wrong and they'll probably naturally step backwards.

If the headset is colliding then the teleport action is also disabled to prevent cheating by clipping through walls.

If using `Unity 5.3` or older then the Headset Collision Fade script is attached to the `Camera (head)` object within the `[CameraRig]` prefab.

If using `Unity 5.4` or newer then the Headset Collision Fade script is attached to the `Camera (eye)` object within the `[CameraRig]->Camera (head)` prefab.

The following script parameters are available:

- **Blink Transition Speed:** The fade blink speed on collision.
- **Fade Color:** The colour to fade the headset to on collision.
- **Ignore Target With Tag Or Class:** A string that specifies an object Tag or the name of a Script attached to an obejct and will prevent the object from fading the headset view on collision.

The following events are emitted:

- **HeadsetCollisionDetect:** Emitted when the user's headset collides with another game object.
- **HeadsetCollisionEnded:** Emitted when the user's headset stops colliding with a game object.

The event payload that is emitted contains:

- **collider:** The Collider of the game object the headset has collided with.
- **currentTransform:** The current Transform of the object that the Headset Collision Fade script is

attached to (Camera).

An example of the `VRTK_HeadsetCollisionFade` script can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/011_Camera_HeadSetCollisionFading`. The scene has collidable walls around the play area and if the user puts their head into any of the walls then the headset will fade to black.

Player Presence (VRTK_PlayerPresence)

The concept that the VR user has a physical in game presence which is accomplished by adding a collider and a rigidbody at the position the user is standing within their play area. This physical collider and rigidbody will prevent the user from ever being able to walk through walls or intersect other collidable objects. The height of the collider is determined by the height the user has the headset at, so if the user crouches then the collider shrinks with them, meaning it's possible to crouch and crawl under low ceilings.

The following script parameters are available:

- **Headset Y Offset:** The box collider which is created for the user is set at a height from the user's headset position. If the collider is required to be lower to allow for room between the play area collider and the headset then this offset value will shorten the height of the generated box collider.
- **Ignore Grabbed Collisions:** If this is checked then any items that are grabbed with the controller will not collide with the box collider and rigid body on the play area. This is very useful if the user is required to grab and wield objects because if the collider was active they would bounce off the play area collider.
- **Reset Position On Collision:** If this is checked then if the Headset Collision Fade script is present and a headset collision occurs, the Camera Rig is moved back to the last good known standing position. This deals with any collision issues if a user stands up whilst moving through a crouched area as instead of them being able to clip into objects they are transported back to a position where they are able to stand.

An example of the `VRTK_PlayerPresence` script can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/017_CameraRig_TouchpadWalking`. The scene has a collection of walls and slopes that can be traversed by the user with the touchpad but the user cannot pass through the objects as they are collidable and the rigidbody physics won't allow the intersection to occur.

Touchpad Movement (VRTK_TouchpadWalking)

The ability to move the play area around the game world by sliding a finger over the touchpad is achieved using this script. The Touchpad Walking script is applied to the `[CameraRig]` prefab and adds a rigidbody and a box collider to the user's position to prevent them from walking through other collidable game objects.

If the Headset Collision Fade script has been applied to the Camera prefab, then if a user attempts to collide with an object then their position is reset to the last good known position. This can

happen if the user is moving through a section where they need to crouch and then they stand up and collide with the ceiling. Rather than allow a user to do this and cause collision resolution issues it is better to just move them back to a valid location. This does break immersion but the user is doing something that isn't natural.

The following script parameters are available:

- **Left Controller:** If this is checked then the left controller touchpad will be enabled to move the play area. It can also be toggled at runtime.
- **Right Controller:** If this is checked then the right controller touchpad will be enabled to move the play area. It can also be toggled at runtime.
- **Max Walk Speed:** The maximum speed the play area will be moved when the touchpad is being touched at the extremes of the axis. If a lower part of the touchpad axis is touched (nearer the centre) then the walk speed is slower.
- **Deceleration:** The speed in which the play area slows down to a complete stop when the user is no longer touching the touchpad. This deceleration effect can ease any motion sickness that may be suffered.

An example of the `VRTK_TouchpadWalking` script can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/017_CameraRig_TouchpadWalking`. The scene has a collection of walls and slopes that can be traversed by the user with the touchpad. There is also an area that can only be traversed if the user is crouching. Standing up in this crouched area will cause the user to appear back at their last good known position.

VRTK_RoomExtender

This script allows the playArea to move with the user. The `[CameraRig]` is only moved when at the edge of a defined circle. Aims to create a virtually bigger play area. To use this add this script to the `[CameraRig]` prefab.

The following script parameters are available:

- **Additional Movement Enabled:** This is the a public variable to enable the additional movement. This can be used in other scripts to toggle the `[CameraRig]` movement.
- **Additional Movement Enabled On Button Press:** This configures the controls of the RoomExtender. If this is true you have to press the touchpad to enable it. If this is false you can disable it with pressing the touchpad.
- **Additional Movement Multiplier:** This is the factor by which movement at the edge of the circle is amplified. 0 is no movement of the `[CameraRig]`. Higher values simulate a bigger play area but may be too uncomfortable.
- **Head Zone Radius:** This is the size of the circle in which the playArea is not moved and everything is normal. If it is to low it becomes uncomfortable when crouching.
- **Debug Transform:** This transform visualises the circle around the user where the `[CameraRig]` is not moved. In the demo scene this is a cylinder at floor level. Remember to turn of collisions.

There is an additional script `VRTK_RoomExtender_PlayAreaGizmo` which can be attachted to the `[CameraRig]` to visualize the extended playArea.

An example of the `VRTK_RoomExtender` script can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/028_CameraRig_RoomExtender`. In the example scene the RoomExtender script is controlled by a VRTK_RoomExtender_Controller Example script located at both controllers. Pressing the `Touchpad` on the controller activates the Room Extender. The Additional Movement Multiplier is changed based on the touch distance to the center of the touchpad.

Interactable Object (VRTK_InteractableObject)

The Interactable Object script is attached to any game object that is required to be interacted with (e.g. via the controllers).

The following script parameters are available:

Touch Interactions

- **Highlight On Touch:** The object will only highlight when a controller touches it if this is checked.
- **Touch Highligt Color:** The colour to highlight the object when it is touched. This colour will override any globally set color (for instance on the `VRTK_InteractTouch` script).
- **Rumble On Touch:** The haptic feedback on the controller can be triggered upon touching the object, the `x` denotes the length of time, the `y` denotes the strength of the pulse. (x and y will be replaced in the future with a custom editor)
- **Allowed Touch Controllers:** Determines which controller can initiate a touch action. The options available are:
  - `Both` means both controllers will register a touch
  - `Left_Only` means only the left controller will register a touch
  - `Right_Only` means only the right controller will register a touch

Grab Interactions

- **Is Grabbable:** Determines if the object can be grabbed
- **Is Droppable:** Determines if the object can be dropped by the controller grab button being used. If this is unchecked then it's not possible to drop the item once it's picked up using the controller button. It is still possible for the item to be dropped if the Grab Attach Mechanic is a joint and too much force is applied to the object and the joint is broken. To prevent this it's better to use the Child Of Controller mechanic.
- **Is Swappable:** Determines if the object can be swapped between controllers when it is picked up. If it is unchecked then the object must be dropped before it can be picked up by the other controller.
- **Hold Button To Grab:** If this is checked then the grab button on the controller needs to be continually held down to keep grabbing. If this is unchecked the grab button toggles the grab action with one button press to grab and another to release.
- **Rumble On Grab:** The haptic feedback on the controller can be triggered upon grabbing the

object, the `x` denotes the length of time, the `y` denotes the strength of the pulse. (x and y will be replaced in the future with a custom editor)

- **Allowed Grab Controllers:** Determines which controller can initiate a grab action. The options available are:
  - `Both` means both controllers are allowed to grab
  - `Left_Only` means only the left controller is allowed to grab
  - `Right_Only` means only the right controller is allowed to grab
- **Precision_Snap:** If this is checked then when the controller grabs the object, it will grab it with precision and pick it up at the particular point on the object the controller is touching.
- **Right Snap Handle:** A Transform provided as an empty game object which must be the child of the item being grabbed and serves as an orientation point to rotate and position the grabbed item in relation to the right handed controller. If no Right Snap Handle is provided but a Left Snap Handle is provided, then the Left Snap Handle will be used in place. If no Snap Handle is provided then the object will be grabbed at it's central point.
- **Left Snap Handle:** A Transform provided as an empty game object which must be the child of the item being grabbed and serves as an orientation point to rotate and position the grabbed item in relation to the left handed controller. If no Left Snap Handle is provided but a Right Snap Handle is provided, then the Right Snap Handle will be used in place. If no Snap Handle is provided then the object will be grabbed at it's central point.

Grab Mechanics

- **Grab Attach Type:** This determines how the grabbed item will be attached to the controller when it is grabbed.
  - `Fixed Joint` attaches the object to the controller with a fixed joint meaning it tracks the position and rotation of the controller with perfect 1:1 tracking.
  - `Spring Joint` attaches the object to the controller with a spring joint meaing there is some flexibility between the item and the controller force moving the item. This works well when attempting to pull an item rather than snap the item directly to the controller. It creates the illusion that the item has resistence to move it.
  - `Track Object` doesn't attach the object to the controller via a joint, instead it ensures the object tracks the direction of the controller, which works well for items that are on hinged joints.
  - `Child Of Controller` simply makes the object a child of the controller grabbing so it naturally tracks the position of the controller motion.
- **Detach Threshold:** The force amount when to detach the object from the grabbed controller. If the controller tries to exert a force higher than this threshold on the object (from pulling it through another object or pushing it into another object) then the joint holding the object to the grabbing controller will break and the object will no longer be grabbed. This also works with Tracked Object grabbing but determines how far the controller is from the object before breaking the grab.
- **Spring Joint Strength:** The strength of the spring holding the object to the controller. A low number will mean the spring is very loose and the object will require more force to move it, a high number will mean a tight spring meaning less force is required to move it.

- **Spring Joint Damper:** The amount to damper the spring effect when using a Spring Joint grab mechanic. A higher number here will reduce the oscillation effect when moving jointed Interactable Objects.
- **Throw Multiplier:** An amount to multiply the velocity of the given object when it is thrown. This can also be used in conjunction with the Interact Grab Throw Multiplier to have certain objects be thrown even further than normal (or thrown a shorter distance if a number below 1 is entered).
- **On Grab Collision Delay:** The amount of time to delay collisions affecting the object when it is first grabbed. This is useful if a game object may get stuck inside another object when it is being grabbed.

Use Interactions

- **Is Usable:** Determines if the object can be used
- **Hold Button To Use:** If this is checked then the use button on the controller needs to be continually held down to keep using. If this is unchecked the the use button toggles the use action with one button press to start using and another to stop using.
- **Pointer Activates Use Action:** If this is checked then when a World Pointer beam (projected from the controller) hits the interactable object, if the object has `Hold Button To Use` unchecked then whilst the pointer is over the object it will run it's `Using` method. If `Hold Button To Use` is unchecked then the `Using` method will be run when the pointer is deactivated. The world pointer will not throw the `Destination Set` event if it is affecting an interactable object with this setting checked as this prevents unwanted teleporting from happening when using an object with a pointer.
- **Rumble On Use:** The haptic feedback on the controller can be triggered upon using the object, the `x` denotes the length of time, the `y` denotes the strength of the pulse. (x and y will be replaced in the future with a custom editor)
- **Allowed Use Controllers:** Determines which controller can initiate a use action. The options available are:
  - `Both` means both controllers are allowed to use
  - `Left_Only` means only the left controller is allowed to use
  - `Right_Only` means only the right controller is allowed to use

The following events are emitted:

- **InteractableObjectTouched:** Emitted when another object touches the current object.
- **InteractableObjectUntouched:** Emitted when the other object stops touching the current object.
- **InteractableObjectGrabbed:** Emitted when another object grabs the current object (e.g. a controller).
- **InteractableObjectUngrabbed:** Emitted when the other object stops grabbing the current object.
- **InteractableObjectUsed:** Emitted when another object uses the current object (e.g. a controller).
- **InteractableObjectUnused:** Emitted when the other object stops using the current object.

The event payload that is emitted contains:

- **interactingObject:** The object that is initiating the interaction (e.g. a controller)

The basis of this script is to provide a simple mechanism for identifying objects in the game world that can be grabbed or used but it is expected that this script is the base to be inherited into a script with richer functionality.

An example of the `VRTK_InteractableObject` can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/005_Controller_BasicObjectGrabbing`. The scene also uses the `VRTK_InteractTouch` and `VRTK_InteractGrab` scripts on the controllers to show how an interactable object can be grabbed and snapped to the controller and thrown around the game world.

Another example can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/013_Controller_UsingAndGrabbingMultipleObjects`. The scene shows mutltiple objects that can be grabbed by holding the buttons or grabbed by toggling the button click and also has objects that can have their Using state toggled to show how mutliple items can be turned on at the same time.

Touching Interactable Objects (VRTK_InteractTouch)

The Interact Touch script is attached to a Controller object within the `[CameraRig]` prefab.

The following script parameters are available:

- **Hide Controller On Touch**: Hides the controller model when a valid touch occurs
- **Hide Controller Delay:** The amount of seconds to wait before hiding the controller on touch.
- **Global Touch Highlight Color:** If the interactable object can be highlighted when it's touched but no local colour is set then this global colour is used.
- **Custom Rigidbody Object:** If a custom rigidbody and collider for the rigidbody are required, then a gameobject containing a rigidbody and collider can be passed into this parameter. If this is empty then the rigidbody and collider will be auto generated at runtime to match the HTC Vive default controller.

The following events are emitted:

- **ControllerTouchInteractableObject:** Emitted when a valid object is touched
- **ControllerUntouchInteractableObject:** Emitted when a valid object is no longer being touched

The event payload that is emitted contains:

- **controllerIndex:** The index of the controller doing the interaction
- **target:** The GameObject of the interactable object that is being interacted with by the controller

An example of the `VRTK_InteractTouch` can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/005_Controller/BasicObjectGrabbing`. The scene demonstrates the highlighting of objects that have the `VRTK_InteractableObject` script added to them to show the ability to highlight interactable objects when they are touched by the controllers.

Grabbing Interactable Objects (VRTK_InteractGrab)

The Interact Grab script is attached to a Controller object within the `[CameraRig]` prefab and the Controller object requires the `VRTK_ControllerEvents` script to be attached as it uses this for listening to the controller button events for grabbing and releasing interactable game objects. It listens for the `AliasGrabOn` and `AliasGrabOff` events to determine when an object should be grabbed and should be released.

The Controller object also requires the `VRTK_InteractTouch` script to be attached to it as this is used to determine when an interactable object is being touched. Only valid touched objects can be grabbed.

An object can be grabbed if the Controller touches a game object which contains the `VRTK_InteractableObject` script and has the flag `isGrabbable` set to `true`.

If a valid interactable object is grabbable then pressing the set `Grab` button on the Controller (default is `Trigger`) will grab and snap the object to the controller and will not release it until the `Grab` button is released.

When the Controller `Grab` button is released, if the interactable game object is grabbable then it will be propelled in the direction and at the velocity the controller was at, which can simulate object throwing.

The interactable objects require a collider to activate the trigger and a rigidbody to pick them up and move them around the game world.

The following script parameters are available:

- **Controller Attach Point:** The rigidbody point on the controller model to snap the grabbed object to (defaults to the tip)
- **Hide Controller On Grab:** Hides the controller model when a valid grab occurs
- **Hide Controller Delay:** The amount of seconds to wait before hiding the controller on grab.
- **Grab Precognition:** An amount of time between when the grab button is pressed to when the controller is touching something to grab it. For example, if an object is falling at a fast rate, then it is very hard to press the grab button in time to catch the object due to human reaction times. A higher number here will mean the grab button can be pressed before the controller touches the object and when the collision takes place, if the grab button is still being held down then the grab action will be successful.
- **Throw Multiplier:** An amount to multiply the velocity of any objects being thrown. This can be useful when scaling up the `[CameraRig]` to simulate being able to throw items further.
- **Create Rigid Body When Not Touching:** If this is checked and the controller is not touching an Interactable Object when the grab button is pressed then a rigid body is added to the controller to allow the controller to push other rigid body objects around.

The following events are emitted:

- **ControllerGrabInteractableObject:** Emitted when a valid object is grabbed
- **ControllerUngrabInteractableObject:** Emitted when a valid object is released from being grabbed

The event payload that is emitted contains:

- **controllerIndex:** The index of the controller doing the interaction
- **target:** The GameObject of the interactable object that is being interacted with by the controller

An example of the `VRTK_InteractGrab` can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/005_Controller/BasicObjectGrabbing`. The scene demonstrates the grabbing of interactable objects that have the `VRTK_InteractableObject` script attached to them. The objects can be picked up and thrown around.

More complex examples can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/013_Controller_UsingAndGrabbingMultipleObjects` which demonstrates that each controller can grab and use objects independently and objects can also be toggled to their use state simultaneously. The scene `SteamVR_Unity_Toolkit/Examples/014_Controller_SnappingObjectsOnGrab` demonstrates the different mechanisms for snapping a grabbed object to the controller.

Using Interactable Objects (VRTK_InteractUse)

The Interact Use script is attached to a Controller object within the `[CameraRig]` prefab and the Controller object requires the `VRTK_ControllerEvents` script to be attached as it uses this for listening to the controller button events for using and stop using interactable game objects. It listens for the `AliasUseOn` and `AliasUseOff` events to determine when an object should be used and should stop using.

The Controller object also requires the `VRTK_InteractTouch` script to be attached to it as this is used to determine when an interactable object is being touched. Only valid touched objects can be used.

An object can be used if the Controller touches a game object which contains the `VRTK_InteractableObject` script and has the flag `isUsable` set to `true`.

If a valid interactable object is usable then pressing the set `Use` button on the Controller (default is `Trigger`) will call the `StartUsing` method on the touched interactable object.

The following script parameters are available:

- **Hide Controller On Use:** Hides the controller model when a valid use action starts
- **Hide Controller Delay:** The amount of seconds to wait before hiding the controller on use.

The following events are emitted:

- **ControllerUseInteractableObject:** Emitted when a valid object starts being used
- **ControllerUnuseInteractableObject:** Emitted when a valid object stops being used

The event payload that is emitted contains:

- **controllerIndex:** The index of the controller doing the interaction
- **target:** The GameObject of the interactable object that is being interacted with by the controller

An example can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/006_Controller_UsingADoor`. Which simulates using a door object to open and close it. It also has a cube on the floor that can be grabbed to show how interactable objects can be usable or grabbable.

Another example can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/008_Controller_UsingAGrabbedObject` which shows that objects can be grabbed with one button and used with another (e.g. firing a gun).

Auto Grabbing Interactable Objects (VRTK_ObjectAutoGrab)

It is possible to automatically grab an Interactable Object to a specific controller by applying the Object Auto Grab script to the controller that the object should be grabbed by default.

The Object Auto Grab script is attached to a Controller object within the `[CameraRig]` prefab and the Controller object requires the `VRTK_InteractGrab` script to be attached.

The following script parameters are available:

- **Object To Grab:** A game object (either within the scene or a prefab) that will be grabbed by the controller on game start.
- **Clone Grabbed Object:** If this is checked then the Object To Grab will be cloned into a new object and attached to the controller leaving the existing object in the scene. This is required if the same object is to be grabbed to both controllers as a single object cannot be grabbed by different controllers at the same time. It is also required to clone a grabbed object if it is a prefab as it needs to exist within the scene to be grabbed.

An example can be viewed in the scene `SteamVR_Unity_Toolkit/Examples/026_Controller_ForceHoldObject`. Which automatically grabs a sword to each controller and also prevents the swords from being dropped so they are permanently attached to the user's controllers.

3D UI Controls (Controls/3D)

In order to interact with the world beyond grabbing and throwing, controls can be used to mimic real-life objects.

A number of controls are available which partially support auto-configuration. So can a slider for example detect its min and max points or a button the distance until a push event should be triggered. In the scene gizmos will be drawn that show the current settings. A yellow gizmo signals a valid configuration. A red one shows that you should either change the position of the object or switch to manual configuration mode.

All controls implement the abstract base class VRTK_Control and therefore have some common

functionality. This currently is the event `OnValueChanged`. Individual controls might emit additional events.

The controller should have the VRTK_Interact_Grab script attached with `Create Rigid Body` activated.

The following UI controls are available:

VRTK_Button

Attaching the script to a game object will allow you to interact with it as if it were a push button. The direction into which the button should be pushable can be freely set and auto-detection is supported. Since this is physics-based there needs to be empty space in the push direction so that the button can move.

The script will instantiate the required Rigidbody and ConstantForce components automatically in case they do not exist yet.

The following script parameters are available:

- **Direction:** The axis on which the button should move. All other axis will be frozen.
- **Activation Distance:** The local distance the button needs to be pushed until a push event is triggered.
- **Button Strength:** The amount of force needed to push the button down as well as the speed with which it will go back into its original position.

The following events are emitted:

- **OnPushed:** When the button is successfully pushed.

VRTK_Chest

Transforms a game object into a chest with a lid. The direction can be either x or z and can also be auto-detected with very high reliability.

The script will instantiate the required Rigidbody, Interactable and HingeJoing components automatically in case they do not exist yet. It will expect three distinct game objects: a body, a lid and a handle. These should be independent and not children of each other.

The following script parameters are available:

- **Direction:** The axis on which the chest should open. All other axis will be frozen.
- **Max:** The maximum opening angle of the chest.
- **Lid:** The game object for the lid.
- **Body:** The game object for the body.
- **Handle:** The game object for the handle.

VRTK_Knob

Attaching the script to a game object will allow you to interact with it as if it were a radial knob. The direction can be freely set.

The script will instantiate the required Rigidbody and Interactable components automatically in case they do not exist yet.

The following script parameters are available:

- **Direction:** The axis on which the knob should rotate. All other axis will be frozen.
- **Min:** The minimum value of the knob.
- **Max:** The maximum value of the knob.
- **Step Size:** The increments in which knob values can change.

VRTK_Lever

Attaching the script to a game object will allow you to interact with it as if it were a lever. The direction can be freely set.

The script will instantiate the required Rigidbody, Interactable and HingeJoing components automatically in case they do not exist yet. The joint is very tricky to setup automatically though and will only work in straight forward cases. If you experience issues create the HingeJoint component yourself and configure it as needed.

The following script parameters are available:

- **Direction:** The axis on which the lever should rotate. All other axis will be frozen.
- **Min:** The minimum value of the lever.
- **Max:** The maximum value of the lever.
- **Step Size:** The increments in which lever values can change.

VRTK_Slider

Attaching the script to a game object will allow you to interact with it as if it were a horizontal or vertical slider. The direction can be freely set and auto-detection is supported.

The script will instantiate the required Rigidbody and Interactable components automatically in case they do not exist yet.

The following script parameters are available:

- **Direction:** The axis on which the slider should move. All other axis will be frozen.
- **Min:** The minimum value of the slider.
- **Max:** The maximum value of the slider.
- **Step Size:** The increments in which slider values can change. The slider supports snapping.

2D UI Controls (Controls/2D)

RadialMenu

Attaching the script to a GameObject will allow you to create a dynamic Radial Menu with any number of buttons. The variables are documented in the prefabs section of this README, but a number of public methods are available for use. Interacting with buttons programmatically uses the angle of the desired button (0/360 being the top, 180 being the bottom).

- **RegenerateButtons():** Forces the regeneration of the UI buttons.
- **HoverButton(float menuAngle):** Calls the `Interact()` method with an event type of hover and an angle of menuAngle. This initiates the `pointerEnterHandler` action.
- **ClickButton(float menuAngle):** Calls the `Interact()` method with an event type of click and an angle of menuAngle. This initiates the `pointerDownHandler` action.
- **UnClickButton(float menuAngle):** Calls the `Interact()` method with an event type of stopClick and an angle of menuAngle. This initiates the `pointerUpHandler` action.
- **StopTouching():** Calls the `Interact()` method on the last touched button and clears the currently touched button ID. This initiates the `pointerExitHandler` action.
- **ShowMenu():** Tweens the Radial Menu to a scale of 1
- **HideMenu(bool force):** Tweens the RadialMenu to a scale of 0. If `force` is `false`, then it will only hide the menu if the `HideOnRelease` variable is `true`.

RadialMenuController

Attaching the script to a GameObject will add a RadialMenu component if it does not already exist If it is attached to a child of a controller, it will automatically find the required `VRTK_ControllerEvents` component in its parent. Otherwise one must be defined explicitly in the inspector.

This component will listen for controller touchpad events and call the relevant `RadialMenu` methods.

Abstract Classes (Abstractions/)

To allow for reusablity and object consistency, a collection of abstract classes are provided which can be used to extend into a concrete class providing consistent functionality across many different scripts without needing to duplicate code.

The current abstract classes are available:

VRTK_DestinationMarker

This abstract class provides the ability to emit events of destination markers within the game world. It can be useful for tagging locations for specific purposes such as teleporting.

It is utilised by the `VRTK_WorldPointer` for dealing with pointer events when the pointer cursor touches areas within the game world.

The following script parameters are available:

- **Enable Teleport:** If this is checked then the teleport flag is set to true in the Destination Set event so teleport scripts will know whether to action the new destination.

The following events are emitted:

- **DestinationMarkerEnter:** When a collision with another game object has occured.
- **DestinationMarkerExit:** When the collision with the other game object finishes.
- **DestinationMarkerSet:** When the destination marker is active in the scene to determine the last destination position (useful for selecting and teleporting).

The event payload that is emitted contains:

- **distance:** The distance between the origin and the collided destination.
- **target:** The Transform of the collided destination object.
- **destinationPosition:** The world position of the destination marker.
- **enableTeleport:** Whether the destination set event should trigger teleport
- **controllerIndex:** The optional index of the controller emitting the beam

VRTK_WorldPointer

This abstract class provides any game pointer the ability to know the the state of the implemented pointer. It extends the `VRTK_DestinationMarker` to allow for destination events to be emitted when the pointer cursor collides with objects.

The World Pointer also provides a play area cursor to be displayed for all cursors that utilise this class. The play area cursor is a representation of the current calibrated play area space and is useful for visualising the potential new play area space in the game world prior to teleporting. It can also handle collisions with objects on the new play area space and prevent teleporting if there are any collisions with objects at the potential new destination.

The play area collider does not work well with terrains as they are uneven and cause collisions regularly so it is recommended that handling play area collisions is not enabled when using terrains.

The following script parameters are available:

- **Enable Teleport:** If this is checked then the teleport flag is set to true in the Destination Set event so teleport scripts will know whether to action the new destination. This allows controller beams to be enabled on a controller but never trigger a teleport (if this option is unchecked).
- **Controller:** The controller that will be used to toggle the pointer. If the script is being applied onto a controller then this parameter can be left blank as it will be auto populated by the controller the script is on at runtime.
- **Pointer Material:** The material to use on the rendered version of the pointer. If no material is selected then the default `WorldPointer` material will be used.
- **Pointer Hit Color:** The colour of the beam when it is colliding with a valid target. It can be set to a different colour for each controller.
- **Pointer Miss Color:** The colour of the beam when it is not hitting a valid target. It can be set to a

different colour for each controller.
- **Show Play Area Cursor:** If this is enabled then the play area boundaries are displayed at the tip of the pointer beam in the current pointer colour.
- **Play Area Cursor Dimensions:** Determines the size of the play area cursor and collider. If the values are left as zero then the Play Area Cursor will be sized to the calibrated Play Area space.
- **Handle Play Area Cursor Collisions:** If this is ticked then if the play area cursor is colliding with any other object then the pointer colour will change to the `Pointer Miss Color` and the `WorldPointerDestinationSet` event will not be triggered, which will prevent teleporting into areas where the play area will collide.
- **Pointer Visibility:** Determines when the pointer beam should be displayed:
  - `On_When_Active` only shows the pointer beam when the Pointer button on the controller is pressed.
  - `Always On` ensures the pointer beam is always visible but pressing the Pointer button on the controller initiates the destination set event.
  - `Always Off` ensures the pointer beam is never visible but the destination point is still set and pressing the Pointer button on the controller still initiates the destination set event.
- **Activate Delay:** The time in seconds to delay the pointer beam being able to be active again. Useful for preventing constant teleportation.

Examples

This directory contains Unity3d scenes that demonstrate the scripts and prefabs being used in the game world to create desired functionality.

There is also a `/Resources/Scripts` directory within the `SteamVR_Unity_Toolkit/Examples` directory that contains helper scripts utilised by the example scenes to highlight certain functionality (such as event listeners). These example scripts are not required for real world usage.

The current examples are:

- **001_CameraRig_VR_PlayArea:** A simple scene showing the `[CameraRig]` prefab usage.
  - View Example Tour on YouTube
- **002_Controller_Events:** A simple scene displaying the events from the controller in the console window.
  - View Example Tour on YouTube
- **003_Controller_SimplePointer:** A scene with basic objects that can be pointed at with the laser beam from the controller activated by pressing the `Touchpad`. The pointer events are also displayed in the console window.
  - View Example Tour on YouTube
- **004_CameraRig_BasicTeleport:** A scene with basic objects that can be traversed using the controller laser beam to point at an object in the game world where the user is to be teleported to by pressing `Touchpad` on the controller. When the `Touchpad` is released, the user is teleported to the laser beam end location.
  - View Example Tour on YouTube
- **005_Controller_BasicObjectGrabbing:** A scene with a selection of objects that can be grabbed by

touching them with the controller and pressing the `Trigger` button down. Releasing the trigger button will propel the object in the direction and velocity of the grabbing controller. The scene also demonstrates simple highlighting of objects when the controller touches them. The interaction events are also displayed in the console window.
    - View Example Tour on YouTube
- **006_Controller_UsingADoor:** A scene with a door interactable object that is set to `usable` and when the door is used by pressing the controller `Trigger` button, the door swings open (or closes if it's already open).
    - View Example Tour on YouTube
- **007_CameraRig_HeightAdjustTeleport:** A scene with a selection of varying height objects that can be traversed using the controller laser beam to point at an object and if the laser beam is pointing on top of the object then the user is teleported to the top of the object. Also, it shows that if the user steps into a part of the play area that is not on the object then the user will fall to the nearest object. This also enables the user to climb objects just by standing over them as the floor detection is done from the position of the headset.
    - View Example Tour on YouTube
- **008_Controller_UsingAGrabbedObject:** A scene with interactable objects that can be grabbed (pressing the `Grip` controller button) and then used (pressing the `Trigger` controller button). There is a gun on a table that can be picked up and fired, or a strange box that when picked up and used the top spins.
    - View Example Tour on YouTube
- **009_Controller_BezierPointer:** A scene with a selection of varying height objects that can be traversed using the controller however, rather than just pointing a straight beam, the beam is curved (over a bezier curve) which allows climbing on top of items that the user cannot visibly see.
    - View Example Tour on YouTube
- **010_CameraRig_TerrainTeleporting:** A scene with a terrain object and a selection of varying height 3d objects that can be traversed using the controller laser beam pointer. It shows how the Height Adjust Teleporter can be used to climb up and down game objects as well as traversing terrains as well.
    - View Example Tour on YouTube
- **011_Camera_HeadSetCollisionFading:** A scene with three walls around the play area and if the user puts their head into any of the collidable walls then the headset fades to black to prevent seeing unwanted object clipping artifacts.
    - View Example Tour on YouTube
- **012_Controller_PointerWithAreaCollision:** A scene which demonstrates how to use a controller pointer to traverse a world but where the beam shows the projected play area space and if the space collides with any objects then the teleportation action is disabled. This means it's possible to create a level with areas where the user cannot teleport to because they would allow the user to clip into objects.
    - View Example Tour on YouTube
- **013_Controller_UsingAndGrabbingMultipleObjects:** A scene which demonstrates how interactable objects can be grabbed by holding down the grab button continuously or by pressing the grab button once to pick up and once again to release. The scene also shows that the use

button can have a hold down to keep using or a press use button once to start using and press again to stop using. This allows multiple objects to be put into their Using state at the same time as also demonstrated in this example scene.
    - View Example Tour on YouTube
- **014_Controller_SnappingObjectsOnGrab:** A scene with a selection of objects that demonstrate the different snap to controller mechanics. The two green guns, light saber and sword utilise a Snap Handle which uses an empty game object as a child of the interactable object as the orientation point at grab, so the rotation and position of the object matches that of the given `Snap Handle`. The red gun utilises a basic snap where no precision is required and no Snap Handles are provided which does not affect the object's rotation but positions the centre of the object to the snap point on the controller. The red/green gun utilises the `Precision Snap` which does not affect the rotation or position of the grabbed object and picks the object up at the point that the controller snap point is touching the object.
    - View Example Tour on YouTube
- **015_Controller_TouchpadAxisControl:** A scene with an R/C car that is controlled by using the Controller Touchpad. Moving a finger up and down on the Touchpad will cause the car to drive forward or backward. Moving a finger to the left or right of the `Touchpad` will cause the car to turn in that direction. Pressing the `Trigger` will cause the car to jump, this utilises the Trigger axis and the more the trigger is depressed, the higher the car will jump.
    - View Example Tour on YouTube
- **016_Controller_HapticRumble:** A scene with a collection of breakable boxes and a sword. The sword can be picked up and swung at the boxes. The controller rumbles at an appropriate vibration depending on how hard the sword hits the box. The box also breaks apart if it is hit hard enough by the sword.
    - View Example Tour on YouTube
- **017_CameraRig_TouchpadWalking:** A scene which demonstrates how to move around the game world using the touchpad by sliding a finger forward and backwards to move in that direction. Sliding a finger left and right across the touchpad strafes in that direction. The rotation is done via the user in game physically rotating their body in the place space and whichever way the headset is looking will be the way the user walks forward. Crouching is also possible as demonstrated in this scene and in conjunction with the Headset Collision Fade script it can detect unwanted collisions (e.g. if the user stands up whilst walking as crouched) and reset their position to the last good known position.
    - View Example Tour on YouTube
- **018_CameraRig_FramesPerSecondCounter:** A scene which displays the frames per second in the centre of the headset view. Pressing the trigger generates a new sphere and pressing the touchpad generates ten new spheres. Eventually when lots of spheres are present the FPS will drop and demonstrate the prefab.
    - View Example Tour on YouTube
- **019_Controller_InteractingWithPointer:** A scene which shows how the controller pointer beam can be used to toggle the use actions on interactable objects. Pressing the touchpad activates the beam and aiming it at objects will toggle their use state. It also demonstrates how a game menu could be created by using interactable objects snapped to a game object can trigger actions.

Pressing the Application Menu button displays a cube connected to the controller which has menu options. Pointing the beam with the other controller at the cube will select the menu options accordingly.
    - View Example Tour on YouTube
- **020_CameraRig_MeshTeleporting:** A scene with an object with a mesh collider to demonstrate how the Height Adjust Teleporter can be used to climb up and down onbjects with a mesh collider.
    - View Example Tour on YouTube
- **021_Controller_GrabbingObjectsWithJoints:** A scene with a collection of Interactable Objects that are attached to other objects with joints. The example shows that Interactable Objects can have different attach mechanics to determine the best way of connecting the object to the controller. Fixed Joint works well for holding objects like cubes as they track perfectly to the controller whereas a Spring Joint works well on the drawer to give it a natural slide when operating. Finally, the Track Object works well on the door to give a natural control over the swing of the door. There is also a Character Joint object that can be manipulated into different shapes by pulling each of the relevant sections.
    - View Example Tour on YouTube
- **022_Controller_CustomBezierPointer:** A scene that demonstrates how the Bezier Pointer can have complex objects passed to it to generate the tracer beam and the cursor of the pointer. In the scene, particle objects with rotations are used to demonstrate a different look to the bezier pointer beam.
    - View Example Tour on YouTube
- **023_Controller_ChildOfControllerOnGrab:** A scene that demonstrates the grab mechanic where the object being grabbed becomes a child of the controller doing the grabbing. This works well for objects that need absolute tracking of the controller and do not want to be disjointed under any circumstances. The object becomes an extension of the controller. The scene demonstrates this with a bow and arrow example, where the bow can be picked up and tracked to the controller, whilst the other controller is responsible for picking up arrows to fire in the bow.
    - View Example Tour on YouTube
- **024_CameraRig_ExcludeTeleportLocation:** A scene that shows how to exclude certain objects from being teleportable by either applying a named Tag to the object or by applying a Script of a certain name. In the scene, the yellow objects are excluded from teleport locations by having an `ExcludeTeleport` tag set on them and the black objects are excluded by having a script called `ExcludeTeleport` attached to them. The `ExcludeTeleport` script has no methods and is just used as a placeholder.
    - View Example Tour on YouTube
- **025_Controls_Overview:** A scene that showcases the existing interactive controls, different ways how they can be set up and how to react to events sent by them.
- **026_Controller_ForceHoldObject:** A scene that shows how to grab an object on game start and prevent the user from dropping that object. The scene auto grabs two swords to each of the controllers and it's not possible to drop either of the swords.
- **027_CameraRig_TeleportByModelVillage:** A scene that demonstrates how to teleport to different locations without needing a world pointer and using the Destination Events abstract class on objects that represent a mini map of the game world. Touching and using an object on the map

teleports the user to the specified location.

- **028_CameraRig_RoomExtender:** A scene that demonstates the concept of extending the physical room scale space by multiplying the physical steps taken in the chaperone bounds. A higher multiplier will mean the user can walk further in the play area and the walk multipler can be toggled by a button press.
- **029_Controller_Tooltips:** A scene that demonstrates adding tooltips to game objects and to the controllers using the prefabs `ObjectTooltip` and `ControllerTooltips`.
- **030_Controls_RadialTouchpadMenu:** A scene that demonstrates adding dynamic radial menus to controllers using the prefab `RadialMenu`.
- **031_CameraRig_HeadsetGazePointer:** A scene that demonstrates the ability to attach a pointer to the headset to allow for a gaze pointer for teleporting or other interactions supported by the World Pointers. The `Touchpad` on the right controller activates the gaze beam, where as the `Touchpad` on the left controller activates a beam projected from a drone in the sky as the World Pointers can be attached to any object.
- **032_Controller_CustomControllerModel:** A scene that demonstrates how to use custom models for the controllers instead of the default HTC Vive controllers. It uses two simple hands in place of the default controllers and shows simple state changes based on whether the grab button or use button are being pressed.
- **033_CameraRig_TeleportingInNavMesh:** A scene that demonstrates how a baked NavMesh can be used to define the regions that a user is allowed to teleport into.
- **034_Controls_InteractingWithUnityUI:** A scene that demonstrates how to interact with Unity UI elements. The scene uses the `VRTK_UIPointer` script on the right Controller to allow for the interaction with Unity UI elements using a Simple Pointer beam. The left Controller controls a Simple Pointer on the headset to demonstrate gaze interaction with Unity UI elements.

License