Pirate Cove* as an Example: How to Bring SteamVR into Unity

Who is the Target Audience of This Article?

This article is aimed at an existing Unity* developer who would like to incorporate SteamVR* into their scene. I am making the assumption that the reader already has an HTC Vive* that is set up on their computer and running correctly. If not, follow the instructions at the SteamVR site.

Why Did I Create the My Pirate Cove* VR Scene?

My focus at work changed and I needed to ramp up on virtual reality (VR) and working in the Unity environment. I wanted to figure out how to bring SteamVR into Unity, layout a scene, and enable teleporting so I could move around the scene.

This article is intended to talk to some points that I learned along the way as well as show you how I got VR working with my scene. I will not be talking much about laying out the scene and how I used Unity to get everything up and running; rather, the main focus of this article is to help someone get VR incorporated into their scene.

What Was I Trying to Create?

I was trying to create a virtual reality visual experience. Not a game per say, even though I was using Unity to create my experience. I created a visual experience that would simulate what a small, pirate-themed tropical island could look like. I chose something that was pleasing to me; after all, I live in the rain-infested Pacific Northwest. I wanted to experience something tropical.

What Tools Did I Use?

I used Unity* 5.6. From there I purchased a few assets from the Unity Asset Store. The assets I chose were themed around an old, tropical, pirate setting:

- Pirates Island*
- SteamVR
- Hessburg Navel Cutter*
- Aquas*

Along with a few other free odds and ends.

What Did I Know Going In to This Project?

I had some experience with Unity while working with our Intel® RealSense™ technology. Our SDK had an Intel RealSense Unity plugin, and I had written about the plugin as well as created a few training examples on it.

Up to this point I had never really tried to lay out a first-person type level in Unity, never worried about performance, frames per second (FPS), or anything like that. I had done some degree of scripting while

using Intel RealSense and other simple ramp up tools. However, I'd never had to tackle a large project or scene and any issues that could come with that.

What Was the End Goal of This Project?

What I had hoped for is that I would walk away from this exercise with a better understanding of Unity and incorporating VR into a scene. I wanted to see how difficult it was to get VR up and running. What could I expect once I got it working? Would performance be acceptable? Would I be able to feel steady, not woozy, due to potential lowered frame rates?

And have fun. I wanted to have fun learning all this, which is also why I chose to create a tropical island, pirate-themed scene. I personally have a fascination with the old Caribbean pirate days.

What Misconceptions Did I Have?

As mentioned, I did have some experience with Unity, but not a whole lot.

The first misconception I had was what gets rendered and when. What do I mean? For some reason I had assumed that if I have a terrain including, for example, a 3D model such as a huge cliff, that if I placed the cliff such that only 50 percent of the cliff was above the terrain, Unity would not try to render what was not visible. I somehow thought that that there was some kind of rendering algorithm that would prevent Unity from rendering anything under a terrain object. Apparently that is not the case. Unity still renders the entire cliff 3D model.

This same concept applied to two different 3D cliff models. For example, if I had two cliff game objects, I assumed that if I pushed one cliff into the other to give the illusion of one big cliff, any geometry or texture information that was no longer visible would not get rendered. Again, not the case.

Apparently, if it has geometry and textures, no matter if it's hidden inside something else, it will get rendered by Unity. This is something to take into consideration. I can't say this had a big impact on my work or that it caused me to go find a fix; rather, just in the normal process of ramping up on Unity, I discovered this.

Performance

This might be where I learned the most. Laying out a scene using Unity's terrain tools is pretty straightforward. Adding assets is also pretty straightforward. Before I get called out, I didn't say it was straightforward to do a GOOD job; I'm just saying that you can easily figure out how to lay things out. While I think my Pirates Cove scene is a thing of beauty, others would scoff, and rightfully so. But, it was my first time and I was not trying to create a first-person shooter level. This is a faux visual experience.

FPS: Having talked with people about VR I had learned that the target FPS for VR is 90. I had initially tried to use the Unity Stats window. After talking with others on the Unity forum, I found out that this is not the best tool for true FPS performance. I was referred to this script to use instead, FPS Display script. Apparently it's more accurate.

Occlusion culling: This was an interesting situation. I was trying to figure out a completely non-related issue and a co-worker came over to help me out. We got to talking about FPS and things you can do to help rendering. He introduced me to occlusion culling. He was showing me a manual way to do it, where

you define the sizes and shapes of the boxes. I must confess, I simply brought up Unity's Occlusion Culling window and allowed it to figure out the occlusions on its own. This seemed to help with performance.

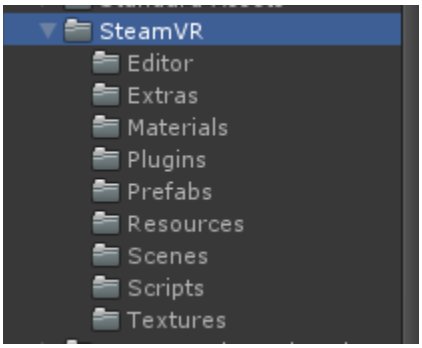
Vegetation: I didn't realize that adding grass to the terrain was going to have such a heavy impact. I had observed other scenes that seemed to have a lot of grass swaying in the wind. Thus, I went hog wild; dropped FPS to almost 0 and brought Unity to its knees. Rather than deal with this, I simply removed the grass and used a clover-looking texture that still made my scene look nice, yet without all the draw calls.

How I Got Vive* Working in My Scene

As mentioned at the top of the article, I'm making the assumption that the reader already has Vive set up and running on their workstation. This area is a condensed version of an existing article that I found, the HTC Vive Tutorial for Unity. I'm not planning on going into any detail on grabbing items with the controllers; for this article I will stick to teleporting. I did modify my version of teleporting, not so much because I think mine is better, but rather, by playing with it and discovering things on my own.

Before you can do any HTC Vive development, you must download and import the SteamVR plugin.

Once the plugin has been imported you will see the following file structure in your project:



In my project, I created a root folder in the hierarchy called VR. In there, I copied the [SteamVR] and [CameraRig] prefabs. You don't have to create a VR folder; I just like to keep my project semi-organized.

I did not need to do anything with the [SteamVR] plugin other than add it to my project hierarchy; instead, we will be looking at the [CameraRig].

I placed the [CameraRig] in my scene where I wanted it to initially start.

After placing the SteamVR [CamerRig] prefab, I had to delete the main camera; this is to avoid conflicts. At this point, I was able to start my scene and look around. I was not able to move, but from a stationary point I could look around and see the controllers in my hand. You can't go anywhere, but at least you can look around.

Getting Tracked Objects

Tracked objects are both the hand controllers as well the headset itself. For this code sample, I didn't worry about the headset; instead, I needed to get input from the hand controllers. This is necessary for tracking things like button clicks, and so on.

First, we must get an instance of the tracked object that the script is on. In this case it will be the controller; this is done in the Awake function.

```
void Awake( )
{
    _trackedController = GetComponent<SteamVR_TrackedObject>( );
}
```

Then, when you want to test for input from one of the two hand controllers, you can select the specific controller by using the following Get function. It uses the tracked object (the hand controller) that this script is attached to:

```
private SteamVR_Controller.Device HandController
{
    get
    {
        return SteamVR_Controller.Input( ( int )_trackedObj.index );
    }
}
```

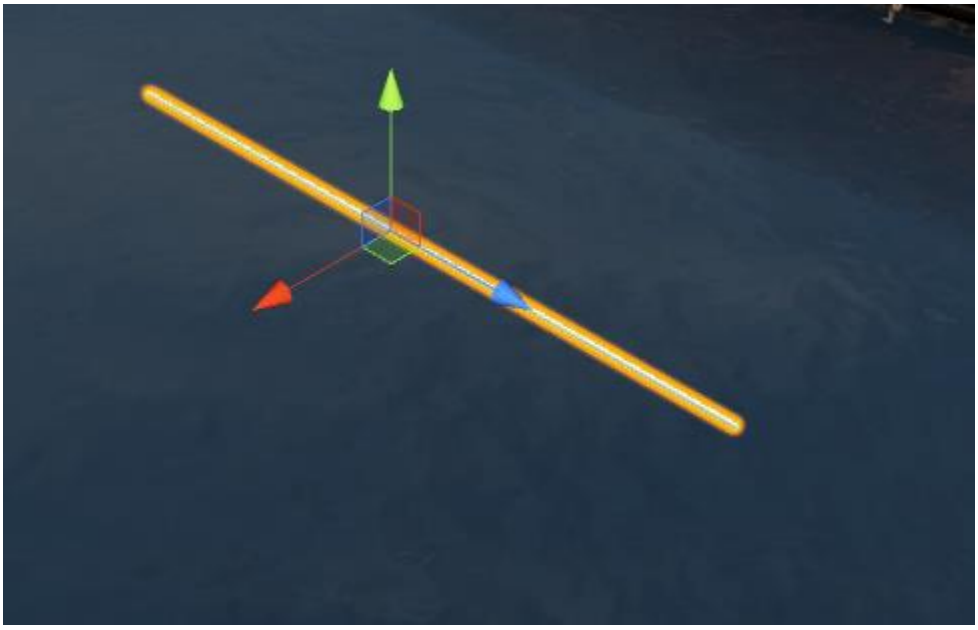
Creating a Teleport System

Now we want to add the ability to move around in the scene. To do this, I had to create a script that knew how to read the input from the hand controllers. I created a new Unity C# script and named it TeleportSystem.cs.

Not only do we need a script but we need a laser pointer, and in this specific case, a reticle. A reticle is not mandatory by any means but does add a little flair to the scene because the reticle can be used as a visual feedback tool for the user. I will just create a very simple circle with a skull image on it.

Create the Laser

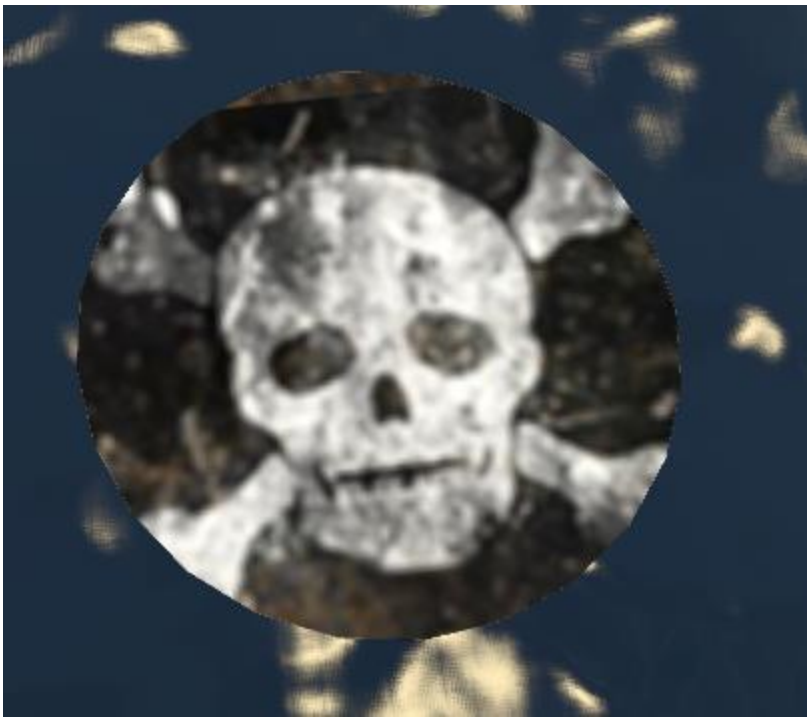
The laser was created by throwing a cube into the scene; high enough in the scene so that it didn't interfere with any of the other assets in that scene. From there I scaled it to x = 0.005, y = 0.005, and z = 1. This gives it a long, thin shape.

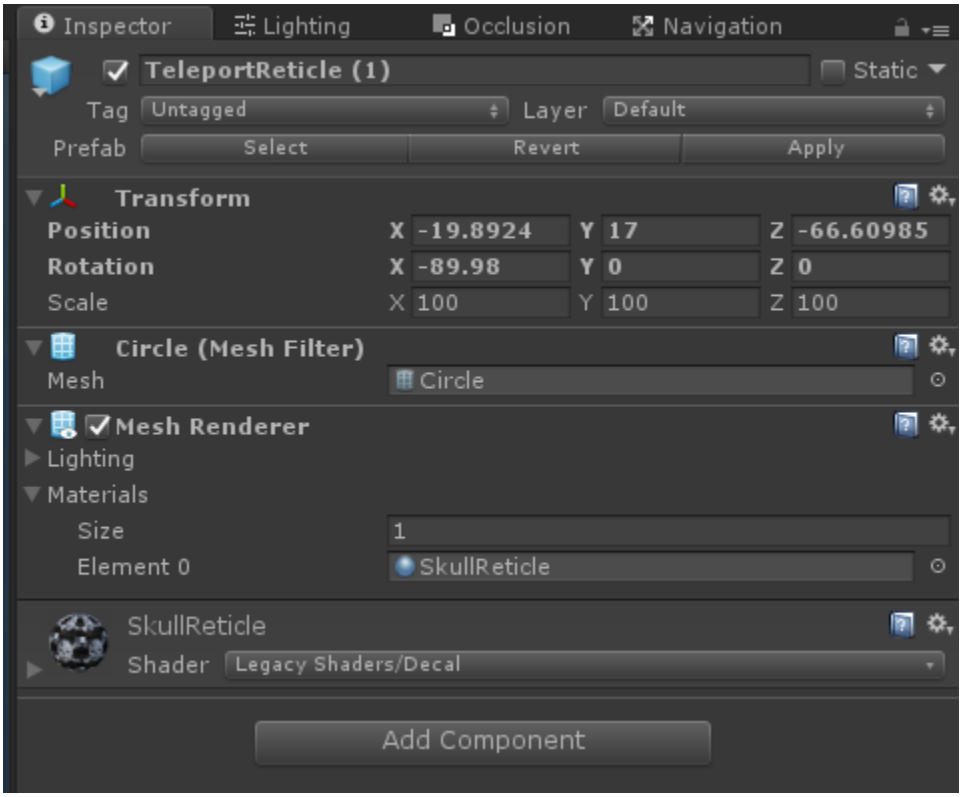


After the laser was created, I saved it as a prefab and removed the original cube from the scene because the cube was no longer needed.

Create the Reticle

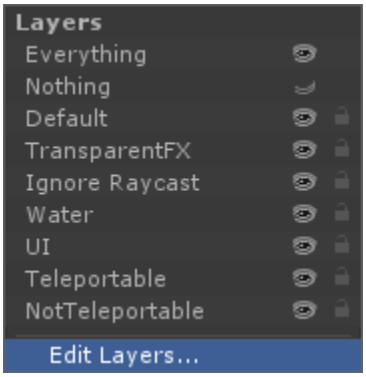
I wanted a customized reticle at the end of the laser; not required, but cool nonetheless. I created a prefab that is a circle mesh with a decal on it.




Inspector
Lighting
Occlusion
Navigation
TeleportReticle (1)
Static
Tag
Untagged
Layer
Default
Prefab
Select
Revert
Apply
Transform
Position
X -19.8924
Y 17
Z -66.60985
Rotation
X -89.98
Y 0
Z 0
Scale
X 100
Y 100
Z 100
Circle (Mesh Filter)
Mesh
Circle
Mesh Renderer
Lighting
Materials
Size
1
Element 0
SkullReticle
SkullReticle
Shader
Legacy Shaders/Decal
Add Component

Setting Up the Layers

This is an important step. You have to tell your laser/reticle what is and what is not teleportable. For example, you may not want to give the user the ability to teleport onto water, or you may not want to allow them to teleport onto the side of a cliff. You can restrict them to specific areas in your scene by using layers. I created two layers—Teleportable and NotTeleportable.



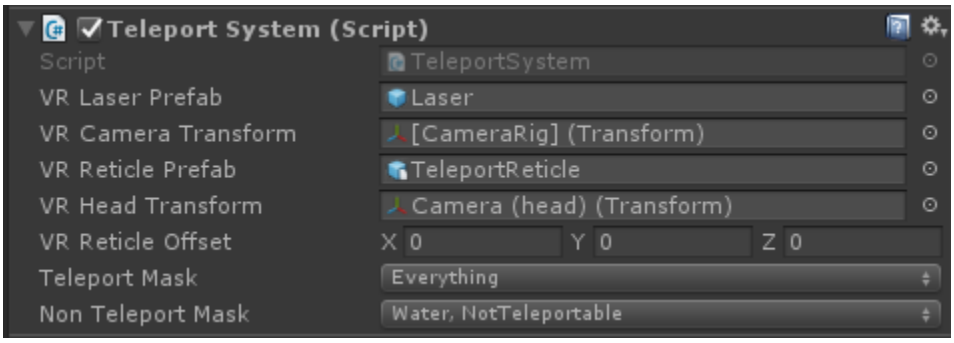
Things that are teleportable, like the terrain itself, the grass huts, and the stairs I would put on the Teleportable layer. Things like the cliffs or other items in the scene that I don't want a user to teleport to, I put on the NotTeleportable layer.

When defining my variables, I defined two-layer masks. One mask just had all layers in it. Then I had a non-teleportable mask that indicates layers are not supposed to be teleportable.

```
// Layer mask to filter the areas on which teleports are allowed
public LayerMask _teleportMask;

// Layer mask specifies which layers we can NOT teleport to.
public LayerMask _nonTeleportMask;
```

When defining the public layer masks, you will see them in the script. They contain drop-down lists that let you pick and choose which layers you do not want someone teleporting to.



Setting up the layers works in conjunction with the LayerMatchTest function.

```
/// <summary>
/// Checks to see if a GameObject is on a layer in a LayerMask.
/// </summary>
/// <param name="layers">Layers we don't want to teleport to</param>
/// <param name="objInQuestion">Object that the raytrace hit</param>
/// <returns>true if the provided GameObject's Layer matches one of the Layers in the
provided LayerMask.</returns>
private static bool LayerMatchTest( LayerMask layers, GameObject objInQuestion )
{
    return( ( 1 << objInQuestion.layer ) & layers ) != 0;
}
```

When LayerMatchTest() is called, I'm sending the layer mask that has the list of layers I don't want people teleporting to, and the game object that the HitTest detected. This test will see if that object is or is not in the non-teleportable layer list.

Updating Each Frame

```
void Update( )
{
    // If the touchpad is held down
    if ( HandController.GetPress( SteamVR_Controller.ButtonMask.Touchpad ) )
    {
        _doTeleport = false;

        // Shoot a ray from controller. If it hits something make it store the point
where it hit and show
        // the laser. Takes into account the layers which can be teleported onto
        if ( Physics.Raycast( _trackedController.transform.position, transform.forward,
out _hitPoint, 100, _teleportMask ) )
        {
            // Determine if we are pointing at something which is on an approved
teleport layer.
            // Notice that we are sending in layers we DON'T want to teleport to.
            _doTeleport = !LayerMatchTest( _nonTeleportMask,
_hitPoint.collider.gameObject );

            if( _doTeleport )
            {
                PointLaser( );
            }
            else
            {
                DisplayLaser( false );
            }
        }
    }
    else
    {
        // Hide _laser when player releases touchpad
        DisplayLaser( false );
    }
```

```
    if( HandController.GetPressUp( SteamVR_Controller.ButtonMask.Touchpad ) &&
_doTeleport )
    {
        TeleportToNewPosition();
        ResetTeleporting( );
    }
}
```

On each update, the code will test to see if the controller's touchpad button was pressed. If so, I'm getting a Raycast hit. Notice that I'm sending my teleport mask that has everything in it. I then do a layer match test on the hit point. By calling the LayerMatchTest function we determine whether it hit something that is or is not teleportable. Notice that I'm sending the list of layers that I do NOT want to teleport to. This returns a Boolean value that is then used to determine whether or not we can teleport.

If we can teleport, I then display the laser using the PointLaser function. In this function, I'm telling the laser prefab to look in the direction of the HitTest. Next, we stretch (scale) the laser prefab from the controller to the HitTest location. At the same time, I reposition the reticle at the end of the laser.

```
private void PointLaser( )
{
    DisplayLaser( true );

    // Position laser between controller and point where raycast hits. Use Lerp because
you can
    // give it two positions and the % it should travel. If you pass it .5f, which is
50%
    // you get the precise middle point.
    _laser.transform.position = Vector3.Lerp( _trackedController.transform.position,
_hitPoint.point, .5f );

    // Point the laser at position where raycast hits.
    _laser.transform.LookAt( _hitPoint.point );

    // Scale the laser so it fits perfectly between the two positions
    _laser.transform.localScale = new Vector3( _laser.transform.localScale.x,
                                               _laser.transform.localScale.y,
                                               _hitPoint.distance );

    _reticle.transform.position = _hitPoint.point + _VRReticleOffset;
}
```

If the HitTest is pointing to a non-teleportable layer, I ensure that the laser pointer is turned off via the DisplayLaser function.

At the end of the function, if both the touch pad is being pressed AND the shouldTeleport variable is true, I call the Teleport function to teleport the user to the new location.

```
private void TeleportToNewPosition( )
{
    // Calculate the difference between the positions of the camera's rig's center and
players head.
    Vector3 difference = _VRCameraTransform.position - _VRHeadTransform.position;

    // Reset the y-position for the above difference to 0, because the calculation
doesn’t consider the
    // vertical position of the player’s head
    difference.y = 0;

    _VRCameraTransform.position =  _hitPoint.point + difference;
}
```

In Closing

This is pretty much how I got my scene up and running. It involved a lot of discovering things on the Internet, reading other people’s posts, and a lot of trial and error. I hope that you have found this article useful, and I invite you to contact me.

For completeness, here is the full script:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


/// <summary>
/// Used to teleport the players location in the scene. Attach to SteamVR's
ControllerRig/Controller left and right
/// </summary>
public class TeleportSystem : MonoBehaviour
{
    // The controller itself
    private SteamVR_TrackedObject _trackedController;

    // SteamVR CameraRig transform
    public Transform _VRCameraTransform;

    // Reference to the laser prefab set in Inspecter
    public GameObject _VRLaserPrefab;

    // Ref to teleport reticle prefab set in Inspecter
    public GameObject _VRReticlePrefab;

    // Stores a reference to an instance of the laser
    private GameObject _laser;
```

```
    // Ref to instance of reticle
    private GameObject _reticle;

    // Ref to players head (the camera)
    public Transform _VRHeadTransform;

    // Reticle offset from the ground
    public Vector3 _VRReticleOffset;

    // Layer mask to filter the areas on which teleports are allowed
    public LayerMask _teleportMask;

    // Layer mask specifies which layers we can NOT teleport to.
    public LayerMask _nonTeleportMask;

    // True when a valid teleport location is found
    private bool _doTeleport;

    // Location where the user is pointing the hand held controller and releases the
button
    private RaycastHit _hitPoint;




    /// <summary>
    /// Gets the tracked object. Can be either a controller or the head mount.
    /// But because this script will be on a hand controller, don't have to worry about
    /// knowing if it's a head or hand controller, this will only get the hand
controller.
    /// </summary>
    void Awake( )
    {
        _trackedController = GetComponent<SteamVR_TrackedObject>( );
    }


    /// <summary>
    /// Initialize the two prefabs
    /// </summary>
    void Start( )
    {
        // Spawn prefabs, init the classes _hitPoint

        _laser      = Instantiate( _VRLaserPrefab );
        _reticle    = Instantiate( _VRReticlePrefab );

        _hitPoint   = new RaycastHit( );

    }


    /// <summary>
    /// Checks to see if player holding down touchpad button, if so, are the trying to
teleport to a legit location
    /// </summary>
    void Update( )
    {
```

```
        // If the touchpad is held down
        if ( HandController.GetPress( SteamVR_Controller.ButtonMask.Touchpad ) )
        {
            _doTeleport = false;

            // Shoot a ray from controller. If it hits something make it store the
point where it hit and show
            // the laser. Takes into account the layers which can be teleported onto
            if ( Physics.Raycast( _trackedController.transform.position,
transform.forward, out _hitPoint, 100, _teleportMask ) )
            {
                // Determine if we are pointing at something which is on an approved
teleport layer.
                // Notice that we are sending in layers we DON'T want to teleport to.
                _doTeleport = !LayerMatchTest( _nonTeleportMask,
_hitPoint.collider.gameObject );

                if( _doTeleport )
                    PointLaser( );
                else
                    DisplayLaser( false );
            }
        }
        else
        {
            // Hide _laser when player releases touchpad
            DisplayLaser( false );
        }
        if( HandController.GetPressUp( SteamVR_Controller.ButtonMask.Touchpad ) &&
_doTeleport )
        {
            TeleportToNewPosition( );
            DisplayLaser( false );
        }
    }


    /// <summary>
    /// Gets the specific hand contoller this script is attached to, left or right
controller
    /// </summary>
    private SteamVR_Controller.Device HandController
    {
        get
        {
            return SteamVR_Controller.Input( ( int )_trackedController.index );
        }
    }


    /// <summary>
    /// Checks to see if a GameObject is on a layer in a LayerMask.
    /// </summary>
    /// <param name="layers">Layers we don't want to teleport to</param>
    /// <param name="objInQuestion">Object that the raytrace hit</param>
    /// <returns>true if the provided GameObject's Layer matches one of the Layers in
the provided LayerMask.</returns>
    private static bool LayerMatchTest( LayerMask layers, GameObject objInQuestion )
```

```
    {
        return( ( 1 << objInQuestion.layer ) & layers ) != 0;
    }


    /// <summary>
    /// Displays the lazer and reticle
    /// </summary>
    /// <param name="showIt">Flag </param>
    private void DisplayLaser( bool showIt )
    {
        // Show _laser and reticle
        _laser.SetActive( showIt );
        _reticle.SetActive( showIt );
    }



    /// <summary>
    /// Displays the laser prefab, streteches it out as needed
    /// </summary>
    /// <param name="hit">Where the Raycast hit</param>
    private void PointLaser( )
    {
        DisplayLaser( true );

        // Position laser between controller and point where raycast hits. Use Lerp
because you can
        // give it two positions and the % it should travel. If you pass it .5f, which
is 50%
        // you get the precise middle point.
        _laser.transform.position = Vector3.Lerp(
_trackedController.transform.position, _hitPoint.point, .5f );

        // Point the laser at position where raycast hits.
        _laser.transform.LookAt( _hitPoint.point );

        // Scale the laser so it fits perfectly between the two positions
        _laser.transform.localScale = new Vector3( _laser.transform.localScale.x,
                                                   _laser.transform.localScale.y,
                                                   _hitPoint.distance );

        _reticle.transform.position = _hitPoint.point + _VRReticleOffset;
    }




    /// <summary>
    /// Calculates the difference between the cameraRig and head position. This ensures
that
    /// the head ends up at the teleport spot, not just the cameraRig.
    /// </summary>
    /// <returns></returns>
    private void TeleportToNewPosition( )
    {
        Vector3 difference = _VRCameraTransform.position - _VRHeadTransform.position;
```

```
        difference.y = 0;
        _VRCameraTransform.position =  _hitPoint.point + difference;
    }
}
```

About the Author

Rick Blacker works in the Intel® Software and Services Group. His main focus is on virtual reality with focus on Primer VR application development.

Notices