An Intel® RealSense™ Technology–Driven First-Person Controller in Unity* 3D

By Lynn Thompson

In this article, I use the hand tracking algorithm in the Intel® RealSense™ Software Development Kit (SDK) to control the position and LookAt of a Unity* 3D first-person controller (FPC). The scene’s main camera remains the standard used in a scene that has a default FPC. I then add a second and third camera in the lower left quarter and lower right quarter of the screen, respectively, to display a halo-type Unity 3D asset that reflects the motion of the user’s left and right hands, as interpreted by the Intel RealSense 3D camera. This interpretation is then interfaced with the default position and LookAt scripts controlling location and orientation of the FPC.

In this first article of a three-part series, I begin setting up the environment by installing Unity Pro and the Intel RealSense SDK, and then plugging the Intel RealSense 3D camera into the USB 3.0 port of a Lenovo Ultrabook™ device with Intel® Core™ i5 processor. To familiarize myself with this development environment, I explored the examples provided with the Intel RealSense SDK, double-clicking the Intel RealSense SDK 2014 Beta desktop icon to open Windows* File Explorer. The files reside in C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Intel® RealSense™ SDK 2014. In this directory, I right-clicked the Intel RealSense SDK Sample Browser file, and then clicked Run as Administrator, launching an interface to the demonstration applications, called Sample Browser.

In the left pane of the Sample Browser is the Unity Toolkit tab. This interface into Intel RealSense technology provides Unity Editor menus and configuration interfaces to the Intel RealSense technology elements by importing the Intel RealSense SDK’s asset package from C:\program files\Intel\RSSDK\framework\Unity\RSUnityToolkit.unitypackage. The result is a RealSense Unity Toolkit menu in the top menu bar of the Unity Editor. Although in this article I use the Unity Toolkit interface, the Sample Browser provides examples for implementing the Intel RealSense SDK with C#, C++, JavaScript*, and Unity without the Toolkit interface. For using Unity without the Toolkit interface see the SenseInput script in the CubeSense example. The C# script SenseInput demonstrates direct access to the Intel RealSense Technology SDK.

After I clicked the Unity Toolkit tab, a variety of examples appeared in the right pane of Sample Browser. To develop the FPC example using the Intel RealSense SDK for this article, I extensively referenced Sample 1: Translation, which uses a Grab gesture to initiate translation control over a sphere that knocks down other three-dimensional (3-D) geometry in the scene. I was initially confused by the lack of custom scripts attached to the Translating Sphere in the scene, but I discovered that the Translating Sphere is configured with a Translate Action, an Enable Behavior, and a Disable Behavior (all of which are available in the Unity Editor RealSense Unity Toolkit
menu). These components have intuitive menu options required for configuring gestures to activate their actions and behaviors.

# Configuring the Unity* 3D Scene

After the initial configuration and a bit of exploration in the Sample Browser, I create a Unity 3D scene with basic geometry and an FPC. In this scene, I use the keyboard to control the FPC’s position; I use a mouse to control the rotation and LookAt of the FPC.

To prepare the default FPC for Intel RealSense technology control, I add assets to relay gesture data to the scene. Under the scene’s floor, out of view of the main camera, I add two empty game objects to hold position vectors for the user’s left and right hand positions, respectively. I add a halo effect to each empty game object for visibility.

I then add an orthographic camera for each hand to view its respective halo’d position vector holder. Each halo-type empty game object is in the view of its respective orthographic camera, having depth = 1 (the scene’s main camera if depth = 0). The empty game object holding position for the left hand has a green halo; I place it in the lower left of screen by setting its affiliated camera’s Viewport Rect with values X = 0, W = 0.25, Y = 0, and H = 0.25. The empty game object holding position for the right hand has a red halo; I place it in the lower right of the screen by setting its affiliated camera Viewport Rect with values X = 0.75, W = 0.25, Y = 0, H = 0.25.

Finally, I add two more empty game objects to hold a state of inactive and active for each hand. I use these dummy assets in conjunction with the FPC MouseLook and FPSInputController scripts to reset and inactivate the hand tracking/FPC as needed when the user’s hands leave the Intel RealSense 3D camera field of view. The desired effect is for the FPC’s position, or LookAt, to become idle when the respective controlling hand is not in the camera’s field of view. Figure 1 shows the assets in Unity Editor.
An Intel® RealSense™ Technology–Driven First-Person Controller in Unity* 3D

January 2015

Figure 1. A basic Unity* 3D Editor first-person controller scene with assets for Intel® RealSense™ SDK

With the hand tracking and dummy assets in place, I can now assign them their respective behaviors. When these behaviors are configured, the default position and LookAt scripts assigned to the FPC will interpret and use them.

Intel® RealSense™ SDK Gesture Configuration

To configure hand tracking control using the Intel RealSense SDK, I first select the desired asset—in this case, the green LeftHandTracking asset. I then add action tracking from the RealSense Unity Toolkit menu (see Figure 2). I used the default settings with the exception of the Position and Rotation constraints. I selected the Z-axis check boxes for both of these elements to allow the user to translate the asset in the XY plane of the screen, not the Z-axis into the screen. I also add this hand tracking control to the RightHandTracking asset.
To allow users to enable and disable hand tracking, I then use the **RealSense Unity Toolkit** menu to add an Activate action and a Deactivate action to the `LeftHandTracking` and `RightHandTracking` assets. I kept the defaults for the Activate action, where I use the Grab gesture. I changed the gesture for the Deactivate action to FingersSpread (see Figure 3). The combination of these Activate and Deactivate actions with the modified FPC scripts result in users being able to stop motion by spreading his or her fingers.
Figure 3. Changing gestures associated with an Intel® RealSense™ SDK action

The Activate and Deactivate Actions must be tied to their respective dummy objects: LeftHandActive and RightHandActive. I do this by changing the Game Objects array size from 0 to 1, and then selecting the appropriate scene element (see Figure 4).
Figure 4. Configure the Intel® RealSense™ SDK Activate and Deactivate Actions to affect the desired scene assets

At this point in the example, I have configured the halo’d assets for hand tracking. I have also configured them with Deactivate and Activate actions to affect their respective dummy objects. These assets are now ready to be linked to the FPC position and LookAt scripts and to replace the keyboard and mouse inputs to the FPC.

First-Person Controller and Intel® RealSense™ SDK Integration

With the assets configured in Unity Editor, I complete the example by tying the left and right HandTracking and HandActive assets to the FPC FPSInputController and MouseLook scripts, respectively. The MouseLook script, shown in Listing 1, controls the FPC’s LookAt functionality. The default behavior of this script is to use the horizontal and vertical motion and position of the user’s mouse to manipulate the rotation of the scene’s main camera. The result is that the scene’s main camera “looks at” a 3-D point in the scene that correlates with the X and Y 2D positions of the user’s mouse. This script is configured with a dead band so as not to manipulate the FPC LookAt when near the initial state at the respective camera’s center of view.

Listing 1. MouseLook.cs

```csharp
using UnityEngine;
```
using System.Collections;

/// MouseLook rotates the transform based on the mouse delta.
/// Minimum and Maximum values can be used to constrain the possible rotation

/// To make an FPS style character:
/// - Create a capsule.
/// - Add the MouseLook script to the capsule.
/// - Set the mouse look to use LookX. (You want to only turn character but not tilt it)
/// - Add FPSInputController script to the capsule
///   -> A CharacterMotor and a CharacterController component will be automatically added.

/// - Create a camera. Make the camera a child of the capsule. Reset its transform.
/// - Add a MouseLook script to the camera.
///   -> Set the mouse look to use LookY. (You want the camera to tilt up and down like a
///      head. The character already turns.)

[AddComponentMenu("Camera-Control/Mouse Look")]
public class MouseLook : MonoBehaviour {

    public enum RotationAxes { MouseXAndY = 0, MouseX = 1, MouseY = 2 }
    public RotationAxes axes = RotationAxes.MouseXAndY;
    public float sensitivityX = 15F;
    public float sensitivityY = 15F;

    public float minimumX = -360F;
    public float maximumX = 360F;

    public float minimumY = -60F;
    public float maximumY = 60F;

    //float rotationY = 0.0F;

    GameObject leftHandTracking;
    GameObject leftHandActive;

    //The horizontal and vertical scale values below correlate with
    //the RealSense HandTracking box dimensions
    public float horzLeftScale = 10.0f;
    public float vertLeftScale = 10.0f;
    public Vector3 initLeftPos;
    public Vector3 runningLeftPos = new Vector3(0.0f,0.0f,0.0f);

    public float mouseX = 0.0f;
    public float mouseY = 0.0f;

    void Update ()
    {
        //The following block makes the First Person Controller update LookAt
        //only when RealSense Hand Tracking makes the LeftHandActive asset active.
        //The position of the Halo'd LeftHandTracking asset is only reset to the
        //position when RealSense has "lost track" of the left hand.
        //A Grab or "fist" gesture will activate the asset and controller while a
        //FingersSpread gesture will deactivate as configured in Unity Editor.
        if(leftHandActive.activeSelf ){
            runningLeftPos = leftHandTracking.transform.position;
        }
        else{
            leftHandTracking.transform.position = initLeftPos;
        }
    }
}
runningLeftPos = initLeftPos;
mouseX = 0.0f;
mouseY = 0.0f;

// The values of -0.25f and +0.25f below are to form a deadband
// at the center of the RealSense Box for the user to stop
// rotating (changing LookAt) the First Person Controller
if ((runningLeftPos.x - initLeftPos.x) / horzLeftScale > 0.25f &&
    mouseX <= maximumX) {
    mouseX += 0.5f;
}
if ((runningLeftPos.x - initLeftPos.x) / horzLeftScale < -0.25f &&
    mouseX >= minimumX) {
    mouseX -= 0.5f;
}
if ((runningLeftPos.y - initLeftPos.y) / vertLeftScale > 0.25f &&
    mouseY <= maximumY) {
    mouseY += 0.5f;
}
if ((runningLeftPos.y - initLeftPos.y) / vertLeftScale < -0.25f &&
    mouseY >= minimumY) {
    mouseY -= 0.5f;
}

transform.localEulerAngles = new Vector3 (-mouseY, mouseX, 0.0f);

void Start ()
{
    // Make the rigid body not change rotation
    if (GetComponent<Rigidbody>())
        GetComponent<Rigidbody>().freezeRotation = true;

    leftHandTracking = GameObject.Find ("LeftHandTracking");
    leftHandActive = GameObject.Find ("LeftHandActive");
    initLeftPos = leftHandTracking.transform.position;
}

The FPSInputController script, shown in Listing 2, controls the FPC’s position. This script also reads and scales the respective HandTracking asset’s position and uses it to manipulate the FPC position functionality. This script configures a dead band so as not to manipulate the FPC position when near the initial state at the respective camera’s center of view.

Listing 2. FPSInputController.js

private var motor : CharacterMotor;
var vertical : int;
var horizontal : int;
var rightHandTracking : GameObject;
var rightHandActive : GameObject;
var horzRightScale : float;
var vertRightScale : float;
var initRightPos : Vector3;
var runningRightPos : Vector3;

// Use this for initialization
function Awake () {
    motor = GetComponent(CharacterMotor);
    rightHandTracking = GameObject.Find("RightHandTracking");
    rightHandActive = GameObject.Find("RightHandActive");
    rightHandActive.SetActive(true);
    vertical = 0;
    horizontal = 0;

    // The horizontal and vertical scale values below correlate with
    // the RealSense HandTracking box dimensions
    horzRightScale = 10.0f;
    vertRightScale = 10.0f;
    initRightPos = rightHandTracking.transform.position;
    runningRightPos = rightHandTracking.transform.position;
}

// Update is called once per frame
function Update () {
    // The following block makes the First Person Controller update position
    // only when RealSense Hand Tracking makes the RightHandActive asset active.
    // The position of the Halo'd RightHandTracking asset is only reset to the
    // initial
    // position when RealSense has "lost track" of the right hand.
    // A Grab or "fist" gesture will activate the asset and controller while a
    // Fingers Spread gesture will deactivate as configured in Unity Editor.
    if(rightHandActive.activeSelf) {
        runningRightPos = rightHandTracking.transform.position;
    } else {
        rightHandTracking.transform.position = initRightPos;
        runningRightPos = initRightPos;
    }

    // The values of -0.25f and +0.25f below are to form a deadband
    // at the center of the RealSense Box for the user to stop
    // positioning the First Person Controller
    if ((runningRightPos.x - initRightPos.x) / horzRightScale > 0.25f) {
        horizontal = 1;
    }
    if ((runningRightPos.x - initRightPos.x) / horzRightScale < -0.25f) {
        horizontal = -1;
    }
    if ((runningRightPos.x - initRightPos.x) / horzRightScale > -0.25f &&
        (runningRightPos.x - initRightPos.x) / horzRightScale < 0.25f) {
        horizontal = 0;
    }
}
if ((runningRightPos.y - initRightPos.y) / vertRightScale > 0.25f) {
    vertical = 1;
}

if ((runningRightPos.y - initRightPos.y) / vertRightScale < -0.25f) {
    vertical = -1;
}

if ((runningRightPos.y - initRightPos.y) / horzRightScale > -0.25f &&
    (runningRightPos.y - initRightPos.y) / horzRightScale < 0.25f) {
    vertical = 0;
}

// Get the input vector from keyboard or analog stick
Input.GetAxis("Vertical");
var directionVector = new Vector3(horizontal, 0, vertical);

if (directionVector != Vector3.zero) {
    // Get the length of the direction vector and then normalize it
    // Dividing by the length is cheaper than normalizing when we already have the
    length anyway
    var directionLength = directionVector.magnitude;
    directionVector = directionVector / directionLength;

    // Make sure the length is no bigger than 1
    directionLength = Mathf.Min(1, directionLength);

    // Make the input vector more sensitive towards the extremes and less sensitive
    in the middle
    // This makes it easier to control slow speeds when using analog sticks
    directionLength = directionLength * directionLength;

    // Multiply the normalized direction vector by the modified length
    directionVector = directionVector * directionLength;
}

// Apply the direction to the CharacterMotor
motor.inputMoveDirection = transform.rotation * directionVector;
motor.inputJump = Input.GetButton("Jump");

// Require a character controller to be attached to the same game object
@script RequireComponent (CharacterMotor)
@script AddComponentMenu ("Character/FPS Input Controller")

Observations

With some practice, users can use their hands to manipulate the position and LookAt of the Unity FPC. In the example configured in this article, I wanted to have a FingersSpread gesture deactivate only the hand displaying the FingersSpread gesture. During runtime, the behavior exhibited of a FingersSpread gesture by either hand deactivated both hands. In addition, I wanted the FingersSpread Deactivate gesture to snap the HandTracking asset back to the center of its camera’s view. This behavior seemed to work for the left hand only. After some experimentation, this issue seems to be related to Ultrabook device performance. I encountered the behaviors I
described above when running the scene at the Ultrabook device’s maximum resolution of 3200x1800. The behavior was as expected when running the scene at a resolution of 1024x768.

To configure a fully functional and fun game, I would add discrete gestures (thumbs up, thumbs down, pinch, etc.) to augment the Translation Control of the FPC’s LookAt and position—for example, gestures to increase LookAt and position speed, projectile or weapons launch, heads-up display and graphical user interface (GUI) activation, camera zoom, and target selection. With Intel RealSense SDK technology you could develop a GUI to the Unity FPC that is much richer and more interesting than anything a keyboard and mouse can provide.

Some loss-of-control issues arose, but these may have been the result of the user (me) not having both hands in proper view of the camera. Available in the RealSense Unity Toolkit > Add to Scene menu is a Sense AR Object. You can use this object to display the view image that the Intel RealSense 3D camera obtains. The camera view displayed on the screen is a useful reference for users when positioning their hands and may help alleviate this loss of control. I use the Sense AR Object in the next article which provides an example of using Intel RealSense technology to implement Gesture Sequences. The Enable Behavior and Disable Behavior actions may also be useful for mitigating this behavior.

**Conclusion**

The RealSense Unity Toolkit menu provided in the Unity Editor is a useful interface to Intel RealSense technology when configuring Unity 3D assets for hand tracking. After I had configured everything, it wasn’t difficult to use these assets to manipulate the position and LookAt of a Unity 3D FPC. Although I had some control issues with this basic example, the toolkit does provide a means of mitigating them. With experience, a Unity 3D developer can effectively replace the traditional FPC inputs using hand tracking in the Intel RealSense SDK.

**About the Author**

Lynn Thompson is an IT professional with more than 20 years of experience in business and industrial computing environments. His earliest experience is using CAD to modify and create control system drawings during a control system upgrade at a power utility. During this time, Lynn received his B.S. degree in Electrical Engineering from the University of Nebraska, Lincoln. He went on to work as a systems administrator at an IT integrator during the dot com boom. This work focused primarily on operating system, database, and application administration on a wide variety of platforms. After the dot com bust, he worked on a range of projects as an IT consultant for companies in the garment, oil and gas, and defense industries. Now, Lynn has come full circle and works as an engineer at a power utility. Lynn has since earned a Masters of Engineering degree with a concentration in Engineering Management, also from the University of Nebraska, Lincoln.