When configuring gestures to control assets in a scene, it’s important to minimize the complexity of the gestures and the time they use to generate the desired response in a game or simulation. This reduction in complexity improves the flow and experience as users navigate and manipulate the geometry of the simulation or game. One means of accomplishing this is to configure the assets manipulated to handle multiple gesture combinations, eliminating the need for additional and more complex hand and voice gestures that may be confusing to new Intel® RealSense™ technology users.

In this second article of a three-part series, I configure a scene using Intel RealSense technology to manipulate the color and position of simple Unity* 3D assets—spheres, in this example. I followed these basic steps to configure the scene:

1. Configure the scene with a plane, two "rigid-body" spheres, and an empty asset called GestureReceive01.
2. Add Intel RealSense SDK’s Activate and Deactivate Actions to the empty asset. I include one pair of Activate and Deactivate Actions for each sphere. I also add a C# script to the empty asset to receive and process gestures as they occur.
3. Configure the first pair of Activate–Deactivate Actions with FingersSpread and Grab (Fist) gestures, respectively. Configure the second pair of Activate–Deactivate Actions with ThumbsDown and TwoFingerPinch gestures, respectively. These gestures are illustrated on pages 4 and 5 of the Intel RealSense SDK Hand Tracking Tutorial.
4. Code the C# script attached to the empty asset to manipulate the color and position of the spheres based on the order in which Intel RealSense SDK gestures are received.

This scene demonstrates Unity 3D with Intel RealSense SDK plug-ins running on the Windows* 8 operating system. You can customize these sequences for use in many combinations to achieve the interface and user experience you want. You can also create an interface that allows users to customize and personalize the effect of gesture sequences.

In a previous article demonstrating the use of a Unity First Person Controller manipulated by Intel RealSense Technology, I specify how to import the Intel RealSense SDK into the Unity Editor. This procedure is also detailed in the Intel RealSense Technology "Getting Started" document. The SDK is documented in full at the following URL.
Configuring the Unity* 3D Scene

I begin creating this scene by adding a floor plane and two elevated spheres. My intention is to activate and deactivate each sphere by using different hand gestures. Each activation and deactivation changes the color and resets the position of the spheres as described above. Following the creation of the basic geometry, I add a Sense AR Object, part of the Unity Toolkit provided in the Intel RealSense SDK (see Figure 1).

![Figure 1. Adding the Sense AR object, part of the Unity* toolkit provided in the Intel® RealSense™ SDK](image)

This object allows users to view the image stream (and the user gestures, as perceived by the camera) from the Intel RealSense 3D camera for augmented reality purposes. For the sake of convenience, I keep the scene’s default Main Camera and delete the Main Camera configured under the Sense AR Object hierarchy.

After creating the visible geometry and adding the Sense AR Object, I configure an empty asset named GestureReceive01 (see Figure 2) to hold the Intel RealSense SDK actions and the C# script to use these actions.
Figure 2. Deactivate and Activate Actions configured in GestureReceive01, with Grab and FingersSpread Gestures, respectively, for the left sphere, Sphere01.
I then add a Deactivate Action to GestureReceive01 for the left sphere (Sphere01). I use the Unity Editor Inspector in the right pane to set the Deactivate Action’s gesture to Grab. I then configure an Activate Action in GestureReceive01 for the left sphere. I use the Unity Editor Inspector to set the Activate Action’s gesture to FingersSpread.

I add a Deactivate Action to GestureReceive01 for the right sphere (Sphere02). I use the Unity Editor Inspector to set the Deactivate Action’s gesture to a TwoFingerPinch gesture. The last action I add to GestureReceive01 is an Activate Action for the right sphere. I use the Unity Editor Inspector to set the Activate Action’s gesture to a ThumbsDown gesture (see Figure 3).
Figure 3. Deactivate and Activate Actions configured in GestureReceive01, with TwoFingerPinch and ThumbsDown Gestures, respectively, for the right sphere, Sphere02.
Finally, I add a C# script named `GestureSequence01.cs` to the `GestureReceive01 Asset` to manipulate the spheres based on the Activate and Deactivate Actions and their affiliated gestures, configured above (see Figure 4).

![Image](image.png)

**Figure 4.** C# script called `GestureSequence01.cs` added to the `GestureReceive01 Asset` for sphere position and color manipulation.

The script begins by declaring `GameObject` variables for the spheres, `Vector3` variables to hold the initial position of the spheres, a `Boolean` variable to hold the state of a sequence toggle, and a `float` variable to hold the number of times a sequence of gestures has been performed. This variable is declared as a `float` because it’s used to set an RGB color component.

In the script’s `Start()` function, I initialize the sphere variables using `GameObject.Find`. The left sphere’s active state is set to `False`, and the right sphere’s active state is set to `True`. The `Vector3` variables are initialized with their respective sphere’s initial position. The color of the left sphere is initialized to red. I then initialize the right sphere’s color to green.

**FingersSpread Activate and Grab Deactivate**

The script’s `Update()` function is programmed to set the `sequenceToggle boolean` for the left sphere to `True` if the `sequenceToggle boolean` for the left sphere is `false` and the left sphere is activated by a FingersSpread gesture. If the `sequenceToggle boolean` for the left sphere is `true` and the left sphere is deactivated with a Grab gesture, the left sphere’s position is reinitialized, the left sphere’s color is transitioned toward green, and the left sphere’s `toggleCount variable` is incremented.
ThumbsDown Activate and TwoFingerPinch Deactivate

I use the ThumbsDown and TwoFingerPinch gestures to transition the color and reinitialize the position of the right sphere as I used the FingersSpread and Grab gestures for the left sphere. Where I use the FingersSpread and Grab gestures to transition the color of the left sphere from red to green over 10 gesture sequences, I use the ThumbsDown and TwoFingerPinch gestures to transition the color of the right sphere from green to blue over three gesture sequences. When Sphere02 is completely transitioned from green to blue, the left sphere is reinitialized to red. The complete C# script is provided in Listing 1.

Listing 1. GestureSequence01.cs

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

public class GestureSequence01 : MonoBehaviour {
    GameObject sphere01;
    Vector3 sphere01InitialPosition;
    bool sequenceToggle01;
    float toggleCount01;
    GameObject sphere02;
    Vector3 sphere02InitialPosition;
    bool sequenceToggle02;
    float toggleCount02;

    // Use this for initialization
    void Start () {
        // The Sphere01 Asset is configured for Activate Action on FingersSpread Gesture
        // and for Deactivate Action on Grab Gesture via the GestureReceive01 Asset in the
        // Unity Editor.
        sphere01 = GameObject.Find("Sphere01");
        sphere01.SetActive(false);
        sphere01InitialPosition = sphere01.transform.position;
        sphere01.renderer.material.color = new Color(1.0f, 0.0f, 0.0f);
        sequenceToggle01 = false;
        toggleCount01 = 0;

        // The Sphere02 Asset is configured for Activate Action on ThumbsDown Gesture
        // and for Deactivate Action on TwoFingerPinch Gesture via the GestureReceive01 Asset in the
        // Unity Editor.
        sphere02 = GameObject.Find("Sphere02");
        sphere02.SetActive(true);
        sphere02InitialPosition = sphere02.transform.position;
        sphere02.renderer.material.color = new Color(0.0f, 1.0f, 0.0f);
        sequenceToggle02 = false;
        toggleCount02 = 0;
    }

    // Update is called once per frame
    void Update () {
        // If Sphere01 is switched from Inactive to Active via the FingersSpread Gesture
        // then the toggle variable is set to true
        if (!sequenceToggle01 && !sphere01.activeSelf) {
            sequenceToggle01 = true;
        }

        // If Sphere01 is switched from Active to Inactive via the Grab Gesture
        // then the sphere is returned to its initial position. Sphere01 will
        // not become visible at the initial position until activated by the
        // FingersSpread Gesture. The toggle count is also incremented and the color of
        // Sphere01 is moved from red towards green.
        if (sequenceToggle01 && !sphere01.activeSelf) {
            sequenceToggle01 = false;
            sphere01.transform.position = sphere01InitialPosition;
            toggleCount01 += 1;
            if (toggleCount01 <= 10) {
                sphere01.renderer.material.color = new Color(toggleCount01 / 10, 1.0f, 0.0f);
            }
        }
    }
}
```

v3a DRAFT 16 December 2014
sphere01.renderer.material.color = new Color(1.0f - (toggleCount01/10.0f),toggleCount01/10.0f,0.0f);
}
}
//If Sphere02 is switched from Inactive to Active via the TwoFingerPinch Gesture
//then the toggle variable is set to true
if (!sequenceToggle02 && sphere02.activeSelf) {
    sequenceToggle02 = true;
}
//If Sphere02 if switched from Active to Inactive via the TwoFingerPinch Gesture
//then the sphere is returned to its initial position. Sphere02 will
//not become visible at the initial position until Activated by the
//ThumbDown Gesture. The toggle count is also incremented and the color of
//Sphere01 is moved from green towards blue. On the third toggle Sphere01's color
//is reset to red.
if (sequenceToggle02 && !sphere02.activeSelf) {
    sequenceToggle02 = false;
    sphere02.transform.position = sphere02InitialPosition;
    toggleCount02 += 1;
    if (toggleCount02 <= 10) {
        sphere02.renderer.material.color = new Color(0.0f,1.0f - (toggleCount01/10.0f),toggleCount01/10.0f);
    }
    if (toggleCount02 == 3) {
        sphere01.renderer.material.color = new Color(1.0f,0.0f,0.0f);
    }
}
}

Event- vs Time-Driven Gesture Sequences

The Intel RealSense SDK gesture sequences I configured in this article's example are based purely on events. I don't use the time a user takes to reach 10 sequences for the left sphere or three sequences for the right sphere in GestureSequences01.cs. A simple use for time would be to reset the gesture sequence count if too long a period of time has elapsed between gestures. A more complex implementation may use the Time.deltaTime function to change the way an asset is manipulated based on the time between Activate and Deactivate gestures—for example, configuring an Activate–Deactivate gesture sequence to initiate and configure the launch of a projectile. A short Time.deltaTime between gestures could initiate a low-mass, low-velocity projectile launch. A larger Time.deltaTime between the gestures could launch a proportionally higher-mass, higher-velocity projectile.

Observations

When running this code sample there were times when gestures seemed to be misinterpreted. On occasion, when I made a TwoFingerPinch gesture, it would be seen as a Grab gesture. I was able to correct this misinterpretation by exaggerating the gesture and slightly rocking my hand toward and away from the camera. This behavior did not appear in the examples developed exclusively in C++ and C# and included with the Intel RealSense software development kit.

With time, practice, and use of the Sense AR window, I was able to get the example to perform as desired. What I originally thought was an improvement through gesture exaggeration, as stated as
above, was actually proper hand orientation. The best example of proper hand orientation is the TwoFingerPinch gesture. I obtained consistent behavior when making a TwoFingerPinch gesture where the plane of the circle the thumb and index finger formed was parallel with the plane of the face of the Intel RealSense 3D camera. Another example is making a Grab [Fist] gesture where the line connecting the knuckles of the fist are parallel to the plane of the face of the Intel RealSense 3D camera.

Other Uses for Intel® RealSense™ SDK Gesture Sequences

In the article “An Intel® RealSense™ Technology–Driven First-Person Controller in Unity® 3D” I used the Intel RealSense SDK hand tracking functionality to manipulate the position and “look at” of a Unity 3D first-person controller (FPC). The activate/deactivate functionality I use for sphere manipulation in this article can be used to manipulate other FPC properties, including functions already built into the FPC, such as jump, slide, or speed. You can manipulate speed as a function of the toggleCount variables used in the C# script.

In addition to the built-in FPC properties, you can use gesture sequences to manipulate other typical FPC actions, including such things as starting and stopping the spawning of projectiles (firing a weapon) and warping to other areas of the scene. Just as the sphere’s position is reinitialized in this example, you can set the position of the FPC to a distant or otherwise unreachable vector3 in the scene.

Conclusion

The Unity 3D scene I configured in this article shows Intel RealSense technology running on Windows 8 Ultrabook™ devices as a viable platform for configuring and interpreting users’ hand gesture sequences. You can use gesture sequences to configure any number of user experiences based around manipulation of Unity 3D asset attributes.

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.
Notices
INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL’S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request. Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.

Intel, the Intel logo, Intel RealSense and Ultrabook are trademarks of Intel Corporation in the U.S. and/or other countries.

Copyright © 2015 Intel Corporation. All rights reserved.

*Other names and brands may be claimed as the property of others.