Implementing Multiple Touch Gestures Using Unity* 3D with TouchScript
By Lynn Thompson

This article provides an overview and example for the several TouchScript gestures (Press, Release, Long Press, Tap, Flick, Pan, Rotate, and Scale) available when developing touch-based Unity* 3D simulations and applications running on Ultrabook™ devices with the Windows* 8 operating system. TouchScript is available at no cost from the Unity 3D Asset Store.

The example used in this article starts with a preconfigured scene imported from Autodesk 3ds Max*. I then add geometry to the Unity 3D scene to construct graphical user interface (GUI) widgets that accept touch input from the user. The multiple gestures available via TouchScript will be implemented and customized such that adjustments to the widget can be made during runtime, allowing for a GUI widget that can provide a touch UI that is acceptable to a wider audience when running a Unity 3D-based application on Window 8.

Creating the Example

I first import into Unity 3D an Autodesk 3ds Max FBX* export that contains a few geometry primitives and a small patch of banyan and palm trees (see Figure 1). I add a first-person controller to the scene; then, I assign a box collider to the box primitive imported from Autodesk 3ds Max, which acts as the scene’s floor, to prevent the first-person controller from falling out of the scene.
Implementing Multiple Touch Gestures

Next, I add eight spheres (LeftLittleTouch, LeftRingTouch, LeftMiddleTouch, LeftIndexTouch, RightLittleTouch, RightRingTouch, RightMiddleTouch, and RightIndexTouch) as children of the main camera, which is a child of the first-person controller. I give these spheres a transform scale of $x = 0.15 \ y = 0.30 \ z = 0.15$ and position them in front of the main camera in a manner similar to fingertips on a flat surface. I add a point light above the modified spheres and make a child of the main camera to ensure illumination of the spheres. The layout of these modified spheres is shown in Figure 2.

Figure 1. *Unity* 3D editor with a scene imported from Autodesk 3ds Max*

5 December 2013
Implementing Multiple Touch Gestures

Figure 2. *Unity* 3D runtime with a first-person controller and modified spheres as children for the touch interface

This ends the base configuration of the example. From here, I add TouchScript gestures to the modified spheres and configure scripts to generate a desired touch response.

Adding Press and Release Gestures

The first-person controller from the initialization step of the example contains the JavaScript* file FPSInputController.js and the C# script MouseLook.cs. The FPSInputController.js script takes input from the keyboard; MouseLook.cs, obviously, takes input from the mouse. I modified these scripts to contain public variables that replace vertical and horizontal inputs into FPSInputController.js and to replace mouseX and mouseY inputs into the MouseLook.cs script.

This replacement is fairly straightforward in FPSInputController.js because the keyboard sending a 1, −1, or 0 to the script is replaced with a touch event that results in public variables being changed to a 1, −1, or 0. The touch objects, their respective scripts, and the values they send to script FPSInputController are provided in Table 1 and can be viewed in their entirety in the Unity 3D FirstPerson project accompanying this article.

Table 1. Touch Objects and Corresponding Scripts in FPSInputController.js

<table>
<thead>
<tr>
<th>Object or Asset</th>
<th>Script</th>
<th>Public Variable Manipulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LeftLittleTouch</td>
<td>MoveLeft.cs</td>
<td>horizontal = −1 onPress, 0 onRelease</td>
</tr>
</tbody>
</table>
Implementing Multiple Touch Gestures

Dec 2013

Object or Asset | Script | Public Variable Manipulation
--- | --- | ---
LeftRingTouch | MoveForward.cs | vertical = 1 onPress, 0 onRelease
LeftMiddleTouch | MoveRight.cs | horizontal = 1 onPress, 0 onRelease
LeftIndexTouch | MoveReverse.cs | vertical = -1 onPress, 0 onRelease

This method works for controller position because the information is discrete, as are the TouchScript onPress and onRelease functions. For rotation, an angle variable needs to be updated every frame. To accomplish this, I send a Boolean value to a Mouse Look.cs public variable, and the rotation angle is changed in the Mouse Look.cs Update function at a rate of 1 degree per frame accordingly. The touch objects, their respective scripts, and the values they send to the Mouse Look.cs script are provided in Table 2 and can be viewed in their entirety in the Unity 3D FirstPerson project accompanying this article.

Table 2. Touch Objects and Corresponding Scripts in Mouse Look.cs

<table>
<thead>
<tr>
<th>Object or Asset</th>
<th>Script</th>
<th>Public Variable Manipulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>RightLittleTouch</td>
<td>LookDown.cs</td>
<td>lookDown = true onPress, false onRelease</td>
</tr>
<tr>
<td>RightRingTouch</td>
<td>LookRight.cs</td>
<td>lookRight = true onPress, false onRelease</td>
</tr>
<tr>
<td>RightMiddleTouch</td>
<td>LookUp.cs</td>
<td>lookUp = true onPress, false onRelease</td>
</tr>
<tr>
<td>RightIndexTouch</td>
<td>LookLeft.cs</td>
<td>lookLeft = true onPress, false onRelease</td>
</tr>
</tbody>
</table>

These scripts allow touch interface for first-person shooter (FPS) position and rotation control, replacing keyboard and mouse input.

Using the LongPress Gesture

My original intent for this example was to have the LongPress Gesture make all the touch objects disappear after at least one object had been pressed for a certain amount of time. The touch objects would then all reappear after all touch objects had instigated a release gesture and had not been touched for a certain amount of time. When I tried implementing it this way, however, the behavior was not as I expected, possibly because the LongPress Gesture was used in conjunction with the standard Press and Release Gestures. As a workaround, I implemented this functionality by using the already-implemented Press and Release Gestures in combination with public variables and the delta time method in the system timer.

When initially setting up the Unity 3D scene, I configured a TopLevelGameObject asset to hold the TouchScript Touch Manager and the TouchScript Windows 7 Touch Input script. To facilitate the desired LongPress Gesture, I added a custom C# script named PublicVariables.cs to the TopLevelGameObject asset. I did this not only to hold public variables but also to perform actions based on the state of these variables.

To configure this disappear and reappear functionality, I configured each move and look script associated with its respective touch sphere to have access to the public variables in
PublicVariables.cs. PublicVariables.cs contains a Boolean variable for the state of each modified sphere’s move or look Press Gesture, being true when the modified sphere is pressed and false when it is released.

The PublicVariables.cs script uses the state of these variables to configure a single variable used to set the state of each modified sphere’s MeshRenderer. I configure the timer such that if any modified sphere or combination of modified spheres has been pressed for more than 10 seconds, the variable controlling the MeshRenderer state is set to False. If all of the spheres have been released for more than 2 seconds, the MeshRenderer state is set to True. Each move and look script has in its Update function a line of code to enable or disable its respective sphere’s MeshRenderer based on the state of this variable in PublicVariables.cs.

This code results in all of the modified spheres disappearing when any sphere or combination of spheres has been pressed for more than 10 consecutive seconds. The modified spheres then all reappear if all modified spheres have been released for more than 2 seconds. By enabling and disabling the modified spheres’ MeshRenderer, only the modified sphere’s visibility is affected, and it remains an asset in the scene and is able to process touch gestures. As such, the modified spheres are still used to manipulate the scene’s first-person controller. The user is required to intuitively know where the spheres are positioned and be able to use them while they are not being rendered to the screen. Examine the PublicVariables, Move, and Look scripts in the example provided to see the code in its entirety.

**The Tap Gesture**

To demonstrate the use of multiple gestures with one asset, I add the Tap Gesture to all four move spheres. The Tap Gesture is configured in all four of the left GUI widget’s modified spheres’ respective move scripts. The move scripts are then configured for access to the first-person controller’s Character Motor script. I configure the tap functions in each move script to manipulate the maximum speed variables in the Character Motor’s movement function.

The MoveForward script attached to the LeftRingTouch modified sphere is configured so that a Tap Gesture increases the maximum forward speed and maximum reverse speed by one. I configure the MoveReverse script attached to the LeftIndexTouch modified sphere for a Tap Gesture to decrease the maximum forward speed and maximum reverse speed by one. I configure the MoveLeft script attached to the LeftLittleTouch modified sphere for a Tap Gesture to increase the maximum sideways speed by one and the MoveRight script attached to the LeftMiddleTouch modified sphere for a Tap gesture to decrease the maximum sideways speed by one. The maximum speed variables are floating-point values and can be adjusted as desired.

When using the default settings with the Tap Gesture, the speeds change during the period when the user may want to press the modified sphere to instigate movement. In short, Press and Release Gestures are also considered Tap Gestures. To mitigate this behavior, I changed the Time Limit setting in the Will Recognize section of the Tap Gesture (see Figure 3) from Infinity to 0.25. The lower this setting, the sharper the tap action must be to instigate the Tap Gesture.
Implementing Multiple Touch Gestures

The modified sphere can be used to navigate the scene and adjust the speed at which the scene is navigated. A quirk of this method for navigating and adjusting speed is that when a Tap Gesture is used to adjust speed, the first-person controller is also moved in the direction associated with the modified sphere that was tapped. For example, tapping the LeftIndexTouch modified sphere to decrement the maximum forward speed and maximum reverse speed slightly moves the first-person controller, and subsequently the scene’s main camera, in reverse. In the accompanying Unity 3D project, I add GUI labels to display the maximum speed setting so that the labels can be visualized when tapping the modified spheres. You can remove this quirk by adding a GUI widget component that, when used, disables the Press and Release Gestures, allowing the user to tap the GUI widget component without moving the main scene’s camera. After the maximum forward speed and maximum reverse speed are set to the user’s preference, the new GUI widget component can be used again to enable the Press and Release Gestures.

When developing this portion of the example, I intended to add a Flick Gesture in combination with the Tap Gesture. The Tap Gesture was going to increase speed, and the Flick Gesture was intended to decrease speed. However, when adding both the Flick and the Tap Gestures, only the Tap Gesture was recognized. Both worked independently with the Press and Release Gestures, but the Flick Gesture was never recognized when used in conjunction with the Tap Gesture.

The Flick Gesture

To demonstrate the Flick Gesture, I add functionality to the modified spheres on the right side of the screen. The look scripts are attached to these spheres and control the rotation of the scene’s

Figure 3. Unity* 3D editor showing a modified Time Limit setting in a Tap Gesture

December 2013
main camera, which is a child of the first-person controller. I begin by adding a Flick Gesture to each sphere. I configure the Flick Gestures added to the RightTouchIndex and RightTouchRing modified spheres that control horizontal rotation with their touch direction as horizontal (see Figure 4). I configure the Flick Gestures added to the RightTouchMiddle and RightTouchLittle modified spheres that control vertical rotation with their touch direction as vertical. This may be useful when the modified spheres have disappeared after being pressed for 10 or more seconds and the touch interface does not respond to the user’s flick (as opposed to responding in an undesired manner). The user then knows that the touch interface—modified spheres need to be released, allows 2 seconds for the modified spheres to reappear, and then reengages the touch GUI widget.

Figure 4. Unity* 3D editor showing a modified Direction setting in a Flick Gesture

Each look script uses the public variables that exist in the Mouse Look script. When a modified sphere is flicked, the Mouse Look script instigates a rotation in the respective direction, but because there is no flick Release Gesture, the rotation continues indefinitely. To stop the rotation, the user must sharply press and release the modified sphere that was flicked. This action causes an additional degree of rotation from the Press Gesture but is followed by the Release Gesture, which sets the respective rotation public variable to False, stopping the rotation.

Like the Tap Gesture, the Flick Gesture now works in conjunction with the Press and Release Gestures. Users can still rotate the scene’s main camera by holding down the appropriate modified sphere, releasing it to stop the rotation. With the Flick Gesture implemented, users can also flick the desired modified sphere to instigate a continuous rotation that they can stop by pressing and releasing the same modified sphere.
The Remaining Gestures

To this point in the example, all of the gestures implemented enhance the user’s ability to directly navigate the scene. I use the remaining gestures (Rotate, Scale, and Pan) to allow the user to modify the touch targets’ (the modified spheres) layout for improved ergonomics.

Also, up to this point, all of the gestures are discrete in nature. An immediate action occurs when a Unity 3D asset is tapped, pressed, released, or flicked. This action may be the setting of a variable that results in a continuous action (the flick-instigated rotation), but the actions are discrete in nature. The Rotate, Scale, and Pan Gestures are continuous in nature. These gestures implement a delta method where the difference between the current state of the gesture and that of the previous frame is used in the script to manipulate a Unity 3D screen asset as desired.

The Rotate Gesture

I add the Rotate Gesture in the same way as previous gestures. I use the Add Component menu in the Inspector Panel to add the TouchScript gesture, and the script attached to the touch asset receiving the gesture is modified to react to the gesture. When implemented, the Rotate Gesture is instigated by a movement similar to using two fingers to rotate a coin on a flat surface. This action must occur within an area circumscribed by the Unity 3D asset receiving the gesture.

In this example, rotating the modified spheres results in the capsule shape becoming more of a sphere as the end of the modified sphere is brought into view. This behavior gives the user an alternate touch target interface, if desired. In this example, this functionality is of more use for the modified spheres on the right side of the screen. For the rotate widget on the right side of the screen, the user can flick the appropriate modified sphere for constant rotation up, down, left, or right. I configure the modified spheres controlling vertical rotation with vertical flicks. I configure the modified spheres controlling horizontal rotation with horizontal flicks. The modified spheres controlling horizontal rotation can now be rotated so that the longest dimension is horizontal, allowing for a more intuitive flicking action.

When rotating the modified spheres that are closest to the center of the screen, the modified spheres take on a more spherical appearance. The farther away from the center screen the modified sphere is when being rotated, the modified sphere takes on a more capsule-like appearance. This is an effect of the modified sphere’s distance from the scene’s main camera. It may be possible to mitigate this affect by adjusting the axes on which the modified sphere rotates. The following line of code does the work of rotating the modified sphere when the Rotate Gesture is active:

```csharp
targetRot = Quaternion.AngleAxis(gesture.LocalDeltaRotation, gesture.WorldTransformPlane.normal) * targetRot;
```

The second argument in the Quaternion.AngleAxis is the axis on which the modified sphere rotates. This argument is a Vector3 and can be changed as follows:
targetRot = Quaternion.AngleAxis(gesture.LocalDeltaRotation, new Vector3(1, 0, 0)) * targetRot;

By adjusting this Vector3 as a function of the modified sphere’s distance from the position relative to the scene’s main camera, I can remove the effect, resulting in the modified sphere’s appearance being more consistent and spherical across all the spheres.

**The Scale Gesture**

I add the Scale Gesture as an additional means of altering the modified sphere’s presentation. When rotated, the resulting circular touch target may not be large enough for the user’s preference. The user can employ the Scale Gesture to modify the size of the touch target.

The motion used to instigate a Scale Gesture is similar to the Pinch Gesture used on mobile devices. Two fingers are moved apart and brought together for a scale-down gesture. The fingers are together and moved apart to instigate a scale-up gesture. The code in the accompanying Unity 3D project scales the target asset uniformly. This is not required: You can code for scaling on any combination of the x, y, or z axes.

An additional feature that may help with user utilization of the GUI widgets is automatic scaling following the 10 seconds of constant use, resulting in the disappearance of the GUI widgets. By automatically multiplying a modified sphere’s transform.localScale by 1.1 whenever the modified sphere’s MeshRenderer has been disabled, the user automatically gets a larger touch target, which may reduce the user’s need to intermittently release the GUI widgets to confirm the modified sphere’s location on the touch screen.

**The Pan Gesture**

For the purposes of ergonomics, the Pan Gesture is probably the most useful gesture. It allows users to touch the objects to be manipulated and drag them anywhere on the screen. As the modified spheres are initially positioned, users may, depending on the Ultrabook device they are using, have wrists or forearms resting on the keyboard. With the Pan Gesture functionality implemented, users can drag the modified spheres to the sides of the screen, where there may be less chance of inadvertently touching the keyboard. For additional ergonomic optimization, users can touch all four modified spheres that affect the first-person controller and drag them at the same time to a place on the screen that allows them to rest their wrists and arms as desired.

The following two lines of code, taken from a Unity 3D example, do the work of moving the Unity 3D scene asset when the Pan Gesture is active:

```csharp
var local = new Vector3(transform.InverseTransformDirection(target.WorldDeltaPosition).x, transform.InverseTransformDirection(target.WorldDeltaPosition).y, 0);
targetPan += transform.parent.InverseTransformDirection(transform.TransformDirection(local));
```
Implementing Multiple Touch Gestures

Note that in the above code, the z component of the Vector3 is zero and that in the accompanying example, when the modified spheres are moved, or *panned*, they move only in the x–y plane. By modifying this Vector3, you can customize the interface a great deal. The first example that comes to mind is having a Pan Gesture result in a much faster Unity 3D asset motion on one axis than another.

In the “Everything” example provided with TouchScript, the following line of code limits the panning of the manipulated asset on the y-axis:

```csharp
if(transform.InverseTransformDirection(transform.parent.TransformDirection(targetPan - startPan)).y < 0) targetPan = startPan;
```

This line was commented out in the accompanying example but can easily be modified and implemented if you want to limit how far a user can move a GUI widget component from its original position.

**Resolving Issues During Development of the Example**

One issue I found during development was that the Rotate Gesture never seemed to be recognized when the Press, Release, and Tap Gestures were added. To work around this issue, I added a modified sphere to the GUI widget on the left side of the screen intended for use by the left thumb. I configured this modified sphere with a script (*ToggleReorganize.cs*) so that when a user taps the modified sphere, a Boolean variable is toggled in the PublicVariables script. All of the modified sphere’s scripts reference this variable and disable their Press, Release, Tap, or Flick Gesture when the toggle variable is True, resulting in a UI that requires the user to tap the left thumb button to modify the widget. The user must then tap this left thumb button again when finished modifying the widget to go back to navigating the scene.

During the process of implementing this functionality, I discovered that the right widget did not require this functionality for the widget to be modified. The user could rotate, pan, and scale the widget without tapping the left thumb modified sphere. I implemented the functionality anyway, forcing the user to tap the left thumb modified sphere in the left widget to alter the ergonomics of the right widget. I did this because the right widget became awkward to use when it was modified at the same time it was being used to navigate the scene.

**Looking Ahead**

In addition to the Unity 3D scene navigation control, users can customize the GUI widgets. They can rotate, scale, and move (pan) the components of the widget to suit their ergonomic needs. This functionality is valuable when developing applications that support multiple platforms, such as Ultrabook devices, touch laptops, and tablets. These platforms can be used in any number of environments, with users in a variety of physical positions. The more flexibility the user has to
adjust GUI widget configuration in these environments, the more pleasant the user’s experience will be.

The GUI widgets used in the accompanying example can and should be expanded to use additional GUI widget components designed for thumb use that can control assets in the game or simulation or control assets that are components of the GUI widgets. This expansion may include items in the simulation, such as weapons selection, weapons firing, camera zoom, light color, and jumping. To alter the GUI widget components, these thumb buttons can change the modified spheres to cubes or custom geometry. They can also be used to change the opacity of a material or color that GUI widget components use.

**Conclusion**

This article and the accompanying example show that using TouchScript with Unity 3D is a valid means of implementing a user-configurable GUI widget on Ultrabook devices running Windows 8. The GUI widgets implemented in the example provide a touch interface for the Unity 3D first-person controller. This interface can similarly be connected to the Unity 3D third-person controller or custom controllers simulating an overhead, driving, or flying environment.

When developing Unity 3D GUI widgets for Ultrabook devices running Windows 8, the desired result is for users to not revert back to the keyboard and mouse. All of the functionality that is typically associated with a legacy UI (a keyboard and mouse first-person controller) should be implemented in a production-grade touch interface. By taking this into consideration when implementing the TouchScript gestures described in this article and the accompanying example, you can greatly increase your prospects for obtaining a positive user response.

**Note:** The example provided with this article uses and references the examples provided with TouchScript as downloaded at no cost from the Unity 3D Asset Store.

**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. He has since earned a Masters of Engineering degree with a concentration in Engineering Management, also from the University of Nebraska, Lincoln.
Notice
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

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, and Ultrabook are trademarks of Intel Corporation in the U.S. and/or other countries. Copyright © 2013 Intel Corporation. All rights reserved.

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