"Penelope"

3rd Person Camera & Control Tutorial

for Unity iPhone

Part One:

Overview

Meet Penelope

Penelope is a teenage girl who works at her father's Emeracite mine after school each day. Emeracite is a volatile material which is used to power small villages that are off the main power grid of the large cities. The mining facility is quite old and the central power converter will occasionally blow its top, spewing all the converted Emeracite energy orbs into the air. Penelope's job is simple—she collects them all and deposits them back into the central power converter before they dissipate and lose their energy. Her father will only pay her for the amount of orbs she saves everyday.

At the beginning of the game, the central power converter will rumble and then spit out around 100 energy orbs. The orbs are randomly placed around the level to be collected within a set amount of time. Score is based upon how many energy orbs are deposited into the central power converter before the time runs out. A minimum number of Orbs must be collected before time is up.

The player's objective is to collect as many energy orbs as they can before the timer runs out and the orbs disappear. In order for the collected orbs to count towards their score/pay they must be deposited back into the central power converter.

To download a copy of Penelope from the AppStore, visit http://unity3d.com/iphone/penelope. An up-to-date version of this tutorial and the required project files can be downloaded at http://unity3d.com/support/resources/tutorials/penelope

What You Will Learn

As a demo project, Penelope is designed to provide users of Unity iPhone with the knowledge to address a number of real-world implementation challenges. Special attention has been given to setting up camera and player controls that take advantage of the device's unique input mechanisms and account for the hardware's constraints. In this demo project, you will find three unique third-person control configurations.

Two of the three, "Player Relative" and "Camera Relative", utilize simulated analog joysticks. The third, "Tap to Move", maps 2D touch input to the 3D environment, allowing the player to drag a finger and move Penelope to that point

Player Relative

This control configuration is designed to behave similar to third-person shooters. In it, the camera remains fixed behind the player. The left joystick controls player movement—the player strafes when the stick is moved left or right. The right stick rotates the player. Double tapping the right stick jumps.

Camera Relative

This configuration features camera and control behavior modeled on third-person platformers like Lerpz (The Unity 3rd person platformer tutorial). Like the Player Relative configuration, the

left stick controls player movement. In this configuration, the concept of "forward" is relative to the direction that the camera is facing, so the left stick also turns the player. The right stick is used to orbit the player and for jumping.

The distinction between these two setups is subtle, but each has a very unique feel. Be sure to play the completed project with both configurations several times before beginning to implement them in the second part of this tutorial.

Tap to Move

Tap to Move is a dramatic departure from the other two configurations. With this setup, the player touches the screen or drags their finger to move Penelope to that point. In addition to single-finger touches for player movement, several two-finger gestures such as pinching for camera zoom and twisting fingers to rotate the camera are supported. Since there are no joysticks to tap for jumping, Tap to Move features a jump button.

What You Should Already Know

This tutorial assumes that you are familiar with Unity's interface and know how to perform basic operations, such as positioning an asset in a scene, adding Components to a GameObject, and editing properties in the Inspector. If you are new to Unity or think that you might need to brush up on your Unity interface conventions, you should reference the section "Unity Basics" in the Getting Started section of your Unity documentation.

Building games for a device like the iPhone requires a different approach than targeting the home PC market. Because of this, you'll have to consider new things while developing your

games and plan for some of the feature differences in the iPhone version of Unity. This tutorial assumes that you have read "Getting Started with iPhone Development" in the Unity iPhone Manual and that you have already configured UnityRemote and installed it to your device.

Finally, the Penelope demo makes extensive use of scripting. A general knowledge of scripting is necessary to get the most out of this tutorial, and familiarity with scripting in Unity and Unity's implementation of JavaScript, which is used exclusively in this tutorial, is assumed.

How This Tutorial is Structured

This tutorial ships with two versions of the Penelope project. The first version, the Completed Project, is a fully completed and functional copy of the project that was built and submitted to the App Store. The second version of the project is designed for you to complete while following through this tutorial. Many of the assets used in the Completed Project are not included in the Tutorial Project. In the case of scripts, we will be writing these as we go. In other instances, we will be importing .unityPackage files which contain assets that will be required in order to complete the tutorial.

Tutorial Organization

This tutorial is divided into four main parts. The first part, where you are now, provides a general overview of Penelope. In the second part, Control Setup, we open the Tutorial Project and build our three control setup configuration. In The Emeracite Mine, part three, we start by exploring the main game level in the Completed Project and then switch to the Tutorial Project to implement our own Scene. The final part, Gameplay, ties everything together and adds functionality like orb collection and scoring to the Tutorial Project. When you finish the tutorial, you will have a fully working version of the Penelope Demo that you can compare with the Completed Project.

Elements Used in this Tutorial

Because this is a long tutorial containing a lot of information we have used several formatting and typographical conventions to make it easier to follow.

> Throughout this tutorial, sidebars and tips provide additional information to supplement the text. We use them to clarify, illuminate, and just plain digress. This content isn't necessarily essential to your understanding of this tutorial, but we've tried to make sure that it includes useful insight.

This tutorial contains a lot of scripts. Scripting appears in a `mono-space` font:

```
function Update(){
  DoSomething();
}
```

The scripts included in the tutorial include plenty of comments and are designed to be easy to follow. These comments are usually omitted in the code fragments in the tutorial to save space.

When there is an individual action for you to take, such as dragging a script onto a GameObject, we generally include the instruction in the text and describe why you need to take that action. In some cases, however, there are a sequence of actions that you will need to perform in Unity and no elaboration is required. In those cases, we will present the actions as a bulleted list.

Script names, assets, menu items or Inspector Properties are shown in **boldface** text. Conversely, a `monospace` font is used for script functions and event names.

Asset Overview

All your imported Assets appear in the Project View and they can be almost anything: a simple material or texture, audio files, or even a complete, Prefabricated GameObject (known as a "Prefab"). Unity does not attempt to force a particular way of organizing your project's assets. The Assets in the completed Penelope project have been organized into folders that represent asset types (e.g. "Audio" or "Materials"). This is a common way of organizing smaller projects.

If you don't already have it open, go ahead and open the Completed Project in Unity iPhone. In the Project View, you will see the folders that we have used.

Audio

This folder contains all of the project's audio assets. Files are prefixed with "mus" for music, "sfx" for sound effects and "vox" for player speech. Music files are compressed files that will be treated as iPhone background music. On the iPhone device, compressed audio is Apple Native. Sound effects and player speech files are .wav and .aif files that are treated as 16-bit uncompressed sound effects. You can have multiple uncompressed audio clips playing simultaneously on the device, but only one compressed audio clip at a time. Also, like meshes and textures, audio files are created outside of Unity.

Blob-Shadow

This folder contains a projector for Penelope's shadow. Projectors allow you to project an arbi-

trary Material on all objects that intersect its frustum. The projector uses a Material with a multiplicative shader to appear to cast a shadow onto the ground.

Control Setups

This folder contains a scene file for each of the three player control setups. Scenes contain the objects of your game. Think of each unique Scene file as a unique level. The Scenes in this folder are loaded additively into the game's main scene.

Fonts

This folder contains the true type font UTPenelope. Fonts are used with the GUIText Component. Unity automatically imports TrueType fonts and creates a font material and font texture.

Materials

This folder contains all of the Materials used in the project. Materials are used in conjunction with Mesh or Particle Renderers attached to a GameObject. They play an essential part in defining how your object is displayed. Materials include a reference to the Shader used to render the Mesh or Particles.

Objects

This folder contains imported Meshes used in the project. Meshes make up a large part of your 3D worlds. You don't build your meshes in Unity, but in another application.

Prefabs

This folder contains several Prefabs that are used throughout the project. A Prefab is a type of asset—a reusable GameObject stored in Project View. Prefabs can be inserted into any number of scenes, multiple times per scene. When you add a Prefab to a scene, you create an instance of it. All Prefab instances are linked to the original Prefab and are essentially clones of it. No matter how many instances exist in your project, when you make any changes to the source Prefab you will see the change applied to all instances.

Scripts

This folder contains all of the scripts used in this project. Scripting inside Unity consists of attaching custom script objects called behaviors to GameObjects.

Shaders

This folder contains Shaders used in this project. Shaders in Unity are used through Materials, which essentially combine shader code with parameters like textures. Shaders in Unity are written in a language called ShaderLab, which is similar to Microsoft's .FX files or NVIDIA's CgFX.

Textures

This folder contains textures that are used in this project. Textures bring your Meshes, Particles, and interfaces to life! They are image files that you lay over or wrap around your objects. They are usually created outside of Unity.

Scene Structure

Scenes contain the objects of your game. They can be used to create a main menu, individual

levels, and anything else. Think of each unique Scene file as a unique level. In each Scene, you will place your environments, obstacles, and decorations, essentially designing and building your game in pieces. There are four Scenes in this project.

The first Scene, **EmeraciteMine**, contains the game level. Both initial scripted camera movements and animation, and the actual gameplay happen in this Scene.

There are also three Scenes in the **Control Setups** folder. The Scenes are **CameraRelativeSetup**, **PlayerRelativeSetup**, **TapControlSetup**. Each Scene represents a different type of third-person control setup. These are additively loaded, depending on which control setup is selected in the menu. Unlike traditional Scene loading, when a Scene is additively loaded, objects from the current level are not automatically destroyed. Instead, objects from the new level are added to the current Scene. In the next part of this tutorial, we will actually construct these Scenes.

> It is important to note that the Scene structure used in this tutorial may not be ideal for an actual game. The various control setups have been separated only to clarify their implementation. In a real project, most of this tutorial would be accomplished in a single Scene.

Before moving on to the next part of this tutorial, close **Completed Project** and open **Tutorial Project**. This version of the Penelope demo contains many of the assets used in the final project, but has been designed to require you to build the actual Scenes as you work through the tutorial.

Part Two:

Control Setups

As we mentioned in the last part of this tutorial, our project contains three Scenes for our control setups. Each of these scenes will contain Penelope, her camera, and the objects necessary to control her. Each scene will also contain a plane so that we can test things. In part four of this tutorial, we will be additively loading these Scenes into **EmeraciteMine**. When we get there, we will create a script that deletes the plane when the Scene is loaded.

- To get started, create a new folder in the named Control Setups.
- In an empty scene add a **Plane** GameObject at the origin.
- Set the Scale of the plane's **Transform** to 10x10x10.

For our sample project, we've set the plane's Material to a lightmapped grass. This isn't critical for the game—the plane will never be seen—and you can add any material that you wish.

- Save the scene to the **Control Setups** folder. Name it **CameraRelativeSetup**.
- Save the scene again (**Save Scene As...**) and name it **PlayerRelativeSetup**.
- Save it again as **TapControlSetup**.

Setting Up Our Primitive Stand-In

Before we start working on our character and camera controls, we are going to need to create Penelope… or something like her. We are going to start creating our character control setups using a version of Penelope that has been roughed in using primitives.

There are a couple of real-world reasons to take such an approach. First, by working out gameplay mechanics before spending time on art, we can take advantage of Unity's rapid

development and testing cycles and get something working right away. Prototyping with primitives also lets us work through gameplay without relying on eye-candy as a crutch.

Create an empty GameObject called **StandIn** and rough out the shape of Penelope with primitives, adding them as children. Scale isn't important — we can scale the artwork when we put it inside our player object. Accuracy isn't necessarily important either. In many cases, you could use a capsule and a sphere—just enough to get a feel for size and orientation. Of course, you can also spend a few moments approximating Penelope's form if it helps you imagine things better. In this case, that might be best so that we can get a better understanding of how camera position will work.

When you are done, save StandIn as a Prefab and delete it from your Scene.

Joysticks

Two of our three control setups, **CameraRelativeSetup** and **PlayerRelativeSetup**, use virtual joysticks. Before we can implement those Scenes, we need to create a script to handle input for our joysticks and draw them to the screen.

> As of Unity iPhone version 1.5, a joystick script has shipped with Standard Assets. It was derived from this tutorial. By implementing it yourself, you should be able to gain a better understanding of how it works.

The Controls Object

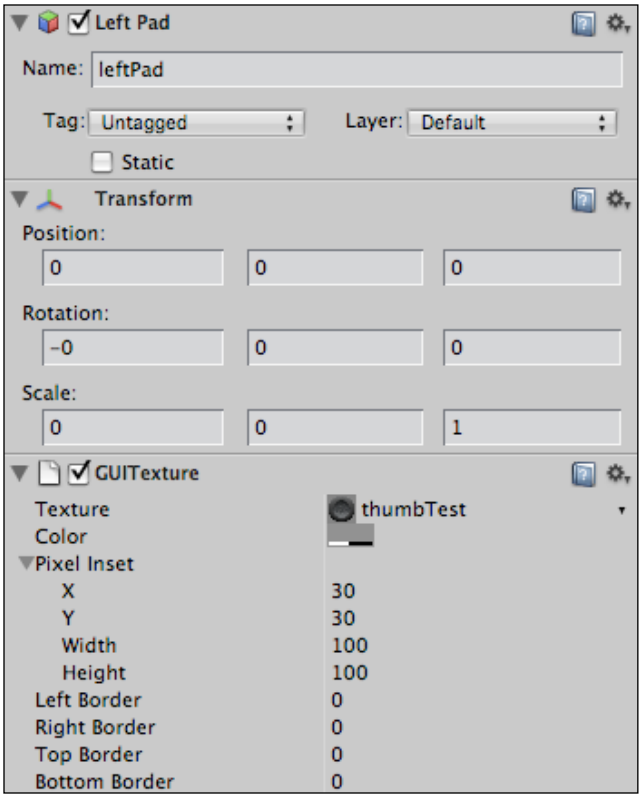
Begin by opening the **CameraRelativeSetup** Scene that you made at the beginning of this part of the tutorial. The Scene will currently have only the plane that you created and the default Main Camera.

Create a new GameObject named **Controls**. With **Controls** selected, check the **Inspector** to make sure that the **Transform** is centered at the origin with no Rotation and a Scale of 1x1x1.

Controls is a parent GameObject that we will use to hold our individual joysticks. Create two empty child GameObjects for **Controls** and name them **leftPad** and **rightPad**.

We will begin configuring **leftPad** by adding a **GUITexture** Component to it.

- For **Texture**, select **thumbTest**. This is an image that we created for the joysticks.
- For **Color**, choose a middle gray—127,127,127—with 45% opacity.
- For **Pixel Inset**, set **X** and **Y** to 30 and **Width** and **Height** to 100.
- All of the remaining values should remain set to their default values of 0.



Before moving on to **rightPad**, set the Scale of **leftPad's Transform** to 0,0,1.

- Add a **GUITexture** to **rightPad**.
- For **Texture**, select **thumbTest** again.
- Set **Color** to the same middle gray.
- For **Pixel Inset**, set **X** to 355 and **Y** to 30. **Width** and **Height** should both be 100.
- All of the remaining values should be set to 0.
- Adjust **rightPad's** scale to match that of **leftPad**.

If we run the Scene now, we will see both of our joysticks in the appropriate places. Neither of them are interactive at the moment, so we need to write a script to handle input for them.

Joystick.js

This script creates a moveable joystick that handles touch input, taps and phases. Dead zones can control where the joystick input gets picked up and can be normalized.

Begin by creating a new script named **Joystick.js** in your **Scripts** folder. Open it up and delete the default structure that Unity adds.

We will be adding this script to our **rightPad** and **leftPad** GameObjects, and it will rely on the existence of their **GUITexture** Components. Since a **GUITexture** is required, we want to add `@script RequireComponent( GUITexture )` to the top of our script. With that in place, we will cache our **GUITexture** Component for the script so that we can reference it later. Add a private variable named `gui` of type `GUITexture` to the script and create a `Start` function. In the `Start` function, set `gui` to `GetComponent( GUITexture )`.

When we use `GetComponent` to get a Component in the `Update` function of a script, Unity has to look that Component up one or more times per frame. To save time and cache the lookup, we declare a private variable at the top of our script and do the lookup only in the `Start` function.

```
private var gui : GUITexture;
function Start(){
  gui = GetComponent( GUITexture );
}
```

We have our **GUITexture**, so let's make it follow our finger around the screen. We will create an `Update` function so that we can check for touch events and update `gui.pixelInset` each frame, which is what places the GUI on screen and controls the width and height.

We need to iterate over all touches and check whether that touch's position falls within the bounds of the gui texture. To do that we use `iPhoneInput.touchCount` to give us the number of touches and `iPhoneInput.GetTouch(i)` to get a specific touch. If we find that a touch is within a gui texture's bounds, then we can set the gui to that new position.

```
function Update(){
  var count = iPhoneInput.touchCount;
  for (var i: int = 0; i < count; i++){
      var touch : iPhoneTouch = iPhoneInput.GetTouch(i);
      if (gui.HitTest( touch.position )){
```

```
            gui.pixelInset.x = touch.position.x;
            gui.pixelInset.y = touch.position.y;
        }
    }
}
```

Save your script and add it to **leftPad** so that we can test it out. When we launch UnityRemote on our device and press **Play** in the editor, we will notice an immediate problem that we need to address: it is incredibly hard to keep your finger on the **GUITexture**. As long as you manage to keep your finger touching the texture, you can drag it around, but it is quite challenging to do so. The cause of this problem is that we aren't offsetting our touch position by half of the height and width of `gui.pixelInset`. As a result, the corner of our joystick GUI is being set to `touch.position`.

To fix this, we initialize `defaultRect` to `gui.pixelInset`, so we know where the GUI texture was originally placed. This will become important for determining how much the thumb has moved away from the original center. We will also declare a private `Vector2` named `guiTouchOffset` and set `guiTouchOffset.x` to `defaultRect.width * 0.5` and `guiTouchOffset.y` to `defaultRect.height * 0.5`.

In the `Update` function, we get a new `Vector2` named `guiTouchPos` and set it to `touch.position - guiTouchOffset`, which accounts for the GUI texture starting in the top left. We can then use `guiTouchPos.x` and `guiTouchPos.y` to update our `gui.pixelInset` values.

```
private var gui : GUITexture;
private var defaultRect : Rect;
private var guiTouchOffset : Vector2;
function Start(){
  gui = GetComponent( GUITexture );
  // get where the gui texture was originally placed
  defaultRect = gui.pixelInset;
  // get our offset for center instead of corner
  guiTouchOffset.x = defaultRect.width * 0.5;
  guiTouchOffset.y = defaultRect.height * 0.5;
}

function Update(){
  var count = iPhoneInput.touchCount;
  // account for the offset in our calculations
  for (var i: int = 0; i < count; i++){
    var touch : iPhoneTouch = iPhoneInput.GetTouch(i);
    var guiTouchPos : Vector2 = touch.position - guiTouchOffset;
    if (gui.HitTest( touch.position )){
      gui.pixelInset.x = guiTouchPos.x;
      gui.pixelInset.y = guiTouchPos.y;
    }
  }
}
```

Save it and press **Play**. Now when we touch the joystick, we can drag it around the screen. We still have a lot to do if we want to behave like a real joystick, but this is a step in the right direction.

If we stop for a moment to think about how our joysticks should behave, we will realize that we want them to move as if they are attached to a fixed position on the screen. When we release them, we want them to snap back to that position.

To determine where they should snap to, we already have `defaultRect`, which stores this information. Basically, we just need to set `gui.pixelInset` to `defaultRect`. Let's create a function called `Reset` that does just that. We will potentially call this function from two places in our `Update` function.

First, we will check to see if `touch.phase` is `iPhoneTouchPhase.Ended` or if it is `iPhoneTouchPhase.Canceled`. If it is, the phase is over, so we should call `Reset`. But, we don't always know if we will get all of the proper phase events, so we will also check to see if `count` is zero. If it is, there are no fingers touching, so we call `Reset`.

```
private var gui : GUITexture;
private var defaultRect : Rect;
private var guiTouchOffset : Vector2;
function Start(){
  gui = GetComponent( GUITexture );
  defaultRect = gui.pixelInset;
```

```
  guiTouchOffset.x = defaultRect.width * 0.5;
  guiTouchOffset.y = defaultRect.height * 0.5;
}

// set our gui texture back to the original location
function Reset(){
  gui.pixelInset = defaultRect;
}

function Update(){
  var count  = iPhoneInput.touchCount;
  // no fingers are touching, so we reset the position
  if (count == 0)
      Reset();
  else {
    for (var i: int = 0; i < count; i++){
      var touch : iPhoneTouch = iPhoneInput.GetTouch(i);
      var guiTouchPos : Vector2 = touch.position - guiTouchOffset;
      if (gui.HitTest( touch.position )){
        gui.pixelInset.x = guiTouchPos.x;
        gui.pixelInset.y = guiTouchPos.y;
        // another check to see if fingers are touching
        if (touch.phase == iPhoneTouchPhase.Ended ||
            touch.phase == iPhoneTouchPhase.Canceled)
```

```
            Reset();
        }
      }
    }
}
```

That is great—the joystick snaps back in place when we release it. But we shouldn't be able to drag it all over the screen—we should clamp the **GUITexture** to limit the movement.

We need a boundary for clamping. Let's create a simple class for this.

```
class Boundary {
  var min : Vector2 = Vector2.zero;
  var max : Vector2 = Vector2.zero;
}
```

Declare a private variable named `guiBoundary` of type `Boundary`. In the `Start` function we will set the `min` and `max` values to the `defaultRect` offset by `guiTouchOffset`. In `Update`, we will clamp `guiTouchPos` between `guiBoundary.min` and `guiBoundary.max` when we set `gui.PixelInset`.

```
private var guiBoundary : Boundary = Boundary();
function Start() {
  // ...
```

```
  guiBoundary.min.x = defaultRect.x - guiTouchOffset.x;
  guiBoundary.max.x = defaultRect.x + guiTouchOffset.x;
  guiBoundary.min.y = defaultRect.y - guiTouchOffset.y;
  guiBoundary.max.y = defaultRect.y + guiTouchOffset.y;
}

function Update(){
  // ...
  gui.pixelInset.x = Mathf.Clamp( guiTouchPos.x,
                      guiBoundary.min.x, guiBoundary.max.x );
  gui.pixelInset.y = Mathf.Clamp( guiTouchPos.y,
                      guiBoundary.min.y, guiBoundary.max.y );
  // ...
}
```

Awesome, testing now shows that our joystick stays constrained to the area around where it started and snaps back to the original position when we release it. The whole thing is entirely cosmetic at this point, so let's make some quick adjustments to get some data out of it.

We want to declare a private `Vector2` named `guiCenter`. This represents the center of the GUI. We will cache it in the `Start` function since it doesn't change.

```
guiCenter.x = defaultRect.x + guiTouchOffset.x;
guiCenter.y = defaultRect.y + guiTouchOffset.y;
```

Next, declare a public `Vector2` named `position`. This variable will have a value between -1 and 1 based on the movement of the joystick. At the bottom of the `Update` function, add:

```
position.x = (  gui.pixelInset.x + guiTouchOffset.x - guiCenter.x ) / guiTouchOffset.x;
position.y = (  gui.pixelInset.y + guiTouchOffset.y - guiCenter.y ) / guiTouchOffset.y;
```

By looking at the position values in the Inspector, we can see that it is a little twitchy—the slightest tap starts sending lots of data. We are going to implement a deadzone in the middle of the stick to fix this. At the top of your script, declare a `Vector2` named `deadZone` to expose it to the Inspector. At the bottom of the `Update` function, we want to get the absolute values for `position`. If they are less than the values for `deadZone`, we want `position` to be 0.

```
public var deadZone : Vector2 = Vector2.zero;
function Update(){
  var absoluteX = Mathf.Abs( position.x );
  var absoluteY = Mathf.Abs( position.y );
  if ( absoluteX < deadZone.x ){
      position.x = 0;
  }
  if ( absoluteY < deadZone.y ){
      position.y = 0;
  }
}
```

In the Inspector, set the **Dead Zone** values to 0.5, 0.5.

Based on our description of how the control setups will behave, we know that a quick double-tap on the right stick should make the character jump. We don't have any behavior in place to handle that in our script, so let's fix that by adding a public variable to store our tap count.

```
public var tapCount : int;
```

We will need some kind of timer to ensure that taps happen within a fixed period of time. Let's create a private variable to store how much time there is left for a tap to occur. We will also need to set the time allowed between taps.

```
static private var tapTimeDelta : float = 0.3;
private var tapTimeWindow : float;
```

In our `Update` function, right after we have set `count` to `iPhoneInput.touchCount`, let's adjust the tap time window and set our `tapCount` to zero if the window has expired.

```
if ( tapTimeWindow > 0 )
  tapTimeWindow -= Time.deltaTime;
else
  tapCount = 0;
```

Inside our hit test, we will accumulate taps within the time window.

```
if ( tapTimeWindow > 0 )
  tapCount++;
else {
  tapCount = 1;
  tapTimeWindow = tapTimeDelta;
}
```

If we test it now and watch the **Tap Count** value in the Inspector, we will see that it climbs while we have our finger on the stick. This isn't exactly the behavior that we wanted, but it makes sense. What we really need is a two-part system that latches on to a finger if it is a new touch and then handles the movement of the GUI independent of that latching behavior.

For that, we are going to need a variable to store the id of the finger that was last used.

```
private var lastFingerId = -1;
```

We are also going to want to do some restructuring of our `Update` function. The function should start the same.

```
function Update(){
  var count = iPhoneInput.touchCount;
  if ( tapTimeWindow > 0 )
    tapTimeWindow -= Time.deltaTime;
  else
```

```
    tapCount = 0;
  if ( count == 0 )
      Reset();
  else {
      for(var i : int = 0;i < count; i++) {
        var touch : iPhoneTouch = iPhoneInput.GetTouch(i);
        var guiTouchPos : Vector2 = touch.position - guiTouchOffset;
```

When we get to our hit test, we want to make sure that this is a new touch. We can check to see if the `lastFingerId` is -1 or if it is just different than `touch.fingerId`. If it is, we will set `lastFingerId` to `touch.fingerId`.

```
      if  ( gui.HitTest( touch.position ) &&
          ( lastFingerId == -1 || lastFingerId != touch.fingerId ) ) {
        lastFingerId = touch.fingerId;
        if ( tapTimeWindow > 0 )
          tapCount++;
        else {
          tapCount = 1;
          tapTimeWindow = tapTimeDelta;
        }
      }
```

We can then use a second check—one that determines if `lastFingerId` is `touch.fingerId`—for

moving the joystick graphic and calling `Reset`.

```
      if ( lastFingerId == touch.fingerId ) {
        if ( touch.tapCount > tapCount )
          tapCount = touch.tapCount;
        gui.pixelInset.x =  Mathf.Clamp( guiTouchPos.x,
                            guiBoundary.min.x, guiBoundary.max.x );
        gui.pixelInset.y =  Mathf.Clamp( guiTouchPos.y,
                            guiBoundary.min.y, guiBoundary.max.y );
        if ( touch.phase == iPhoneTouchPhase.Ended ||
      touch.phase == iPhoneTouchPhase.Canceled )
          Reset();
      }
    }
  }
```

The `Update` function finishes the same way as before.

```
  position.x =  ( gui.pixelInset.x + guiTouchOffset.x - guiCenter.x ) / guiTouchOffset.x;
  position.y =  ( gui.pixelInset.y + guiTouchOffset.y - guiCenter.y ) / guiTouchOffset.y;
  var absoluteX = Mathf.Abs( position.x );
  var absoluteY = Mathf.Abs( position.y );
  if ( absoluteX < deadZone.x ) {
      position.x = 0;
```

```
  }
  if ( absoluteY < deadZone.y ) {
    position.y = 0;
  }
}
```

We can finish implementing this behavior by setting the `lastFingerId` back to -1 in the `Reset` function. **Joystick.js** is almost complete—there are only three things that we still want to do to it. First, we want multiple **Joystick** Components to behave nicely together. Specifically, now that we have the concept of "latching" a finger, we want one **Joystick** to release a finger if another latches it. The second thing that we still need to do is to write a quick function to let another script disable our joystick. Finally, we want to include the option to normalize our position output—rescaling it after taking the deadzone into account.

We'll start by creating a static collection of joysticks and a boolean flag to determine if we have populated that collection. At the top of our `Update` function, we will check to see if the flag is set. If it isn't, we will populate the collection and set the flag.

```
static private var joysticks : Joystick[];
static private var enumeratedJoysticks : boolean = false;

function Update(){
  if ( !enumeratedJoysticks ){
      joysticks = FindObjectsOfType( Joystick );
```

```
        enumeratedJoysticks = true;
    }
    // ...
}
```

Now we will create a function so that if another joystick has latched this finger, we know to release it.

```
function LatchedFinger( fingerId : int ){
    if ( lastFingerId == fingerId )
        Reset();
}
```

At the bottom of the latching conditional block in our Update function, we will tell other **Joystick** Components that we've latched a finger by sending `touch.fingerId` to their `LatchedFinger` functions.

```
for ( var j : Joystick in joysticks ){
    if ( j != this )
        j.LatchedFinger( touch.fingerId );
}
```

The second thing that we needed to do was to create a function to let another script disable our joystick. We'll call the function `Disable` and set `gameObject.active` and `enumerated-`

`Joysticks` to false in it.

```
function Disable(){
  gameObject.active = false;
  enumeratedJoysticks = false;
}
```

Finally, we are going to implement the option to normalize the position output from our joystick. This will be useful for camera movement.

Declare a public boolean named `normalize` so that it can be checked in the Inspector. By default, `normalize` can be set to false. At the bottom of our `Update` function, we currently check to see if the touch position is within the dead zone. We need to add code now to adjust the range of input with the dead zone in mind (i.e. when 'normalize' has been checked).

```
else if ( normalize ) {
  position.x =  Mathf.Sign( position.x ) * ( absoluteX - deadZone.x ) /
                ( 1 - deadZone.x );
}
```

We'll do the same for our `position.y` value using the `absoluteY` and `deadZone.y` values.

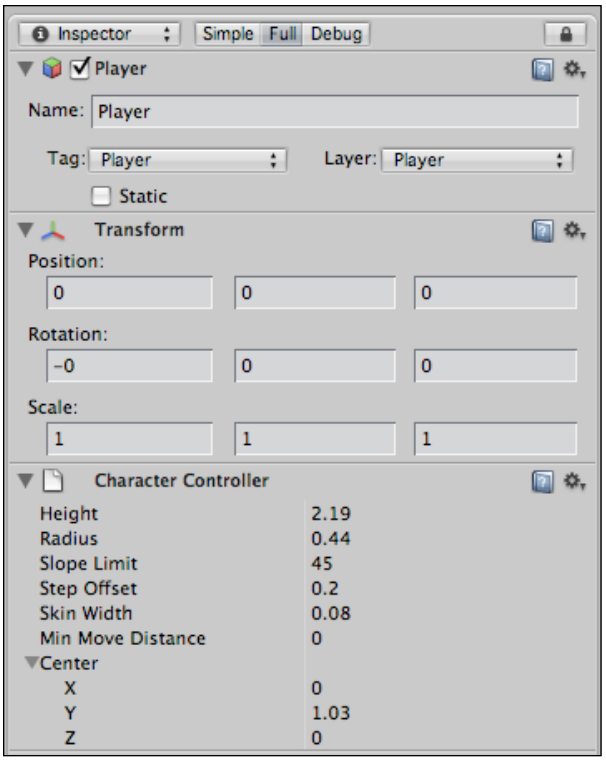
A completed copy of **Joystick.js** is included in the **Completed Project** and in the Script Appendix at the end of this tutorial. If your implementation is having problems, you should take the

time to reference it against this version.

Joystick.js needs to be added to **rightPad** as well. After you've dragged it on, set the **position** to 0,0, set the **Dead Zone** to 0, 0.5, and make sure that **Normalize** is checked.

Camera Relative Setup

Our **CameraRelativeSetup** Scene is already open, so let's continue with it. We are going to create a GameObject for Penelope.



- Create a new Empty GameObject (**GameObject > Create Empty**).
- Name the new object **Player**.
- Make sure that the **Player** GameObject's **Transform** has a **Position** at 0,0,0, no **Rotation**, and a **Scale** of 1x1x1.
- With **Player** selected, add a **Character Controller** Component (**Component > Physics > Character Controller**)
- In the **Character Controller**, set the **Height** to 2.19 and the **Radius** to 0.44.
- Set the **Slope Limit** to 45 and the **Step Offset** to 0.2.
- Set the **Skin Width** to 0.08 and the **Min Move Distance** to 0.
- Set the **Center** values to 0, 1.03, 0.

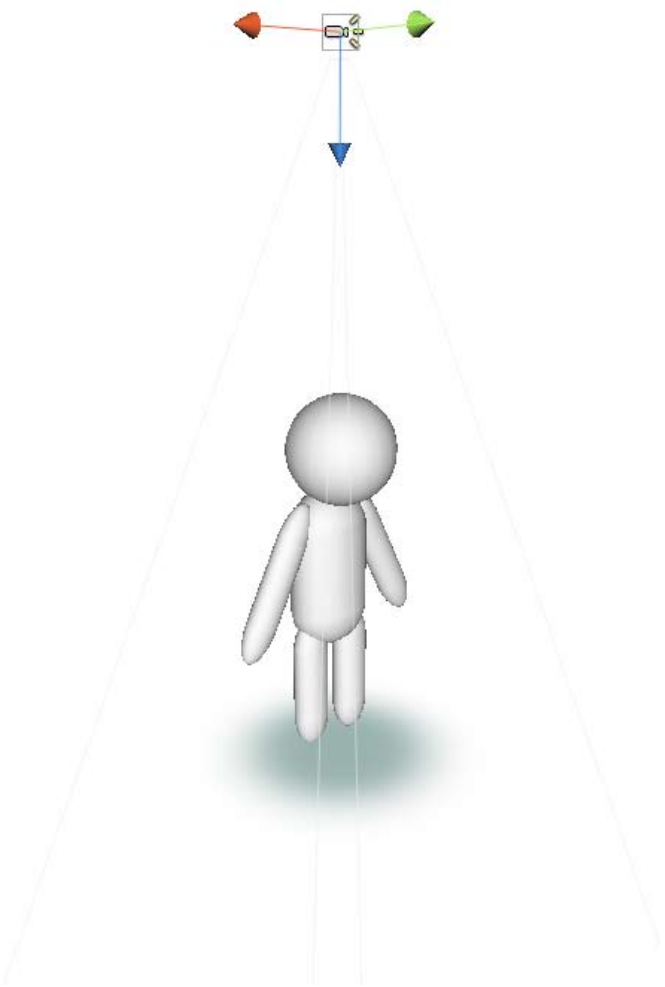
Add an **Audio Listener** Component to Player (**Component > Audio > AudioListener**)

Drag your **StandIn** Prefab into the **Hierarchy** as a child of **Player** and scale and position it so that it stands on the Plane and fits within the **Character Controller**'s Collider.

Set both the **Player's** layer and tag to "**Player**" and change children when prompted.

Create a new empty child GameObject for **Player** and name it **blob shadow projector**. Increase the y value of this GameObject's **Transform** position to 3.25 and rotate it 90 degrees around the X axis (so that the local Z axis points straight down).

- Add a **Projector** Component to **blob shadow projector** (**Component > Rendering > Projector**)
- Set the Near Clip Plane to 0.1 and the Far Clip Plane to 10.
- Set the Field of View to 30 and Aspect Ratio to 1.
- For the Material, select **shadow material**
- Set Ignore Layers to **Player** so that the shadow isn't projected onto Penelope.



Cameras

The idea for the camera system in this setup is that the GameObject that contains an actual Camera Component is the child of an offset GameObject that is, in turn, the child of a Component that follows Penelope around. It really isn't as complex as it sounds, so let's get it into our Scene.

Create a new GameObject called **CameraPivot**. Set the **Transform** to 0,0,0, make sure that there is no rotation and set the scale to 1x1x1. This will be the outermost parent for our camera setup.

We need a script to constrain the relative rotation of our **Transform**. Fortunately, we've included a generic one in the **Scripts** folder that will allow you to select the constraint axis in the editor and specify a min and max amount of rotation that is allowed from the default rotation. The script, **RotationConstraint.js**, is fairly straightforward and well documented, so we won't go into a lot of detail about its actual implementation. What it is doing, however, is grabbing the rotation from the **Transform** after all the `Updates` from other scripts have occurred. It then checks to see if any angles have fallen out of range and corrects them if they have.

- Drag **RotationConstraint.js** onto **CameraPivot**.
- Set the Axis to X.
- Set Min and Max to -15 and 15, respectively.

The second script that you need to add to **CameraPivot** is one that we will be using in many places, so you will be writing it here. This one will be called **FollowTransform.js**. We want our script to match the transform of the GameObject to which it is attached to a second object

selected by the user in the Inspector. We also want our user to be able to determine if the GameObject should match the forward vector of the selected object. To do this, we will need two public variables—a `Transform` named `targetTransform` and a `boolean` named `faceForward`. We will also need to get the Transform of the object that we attach the script to. Rather than getting this each frame, we will create a private variable named `thisTransform`. In the script's `Start` function, we will set `thisTransform` to transform, and will reference that in `Update`. For each frame we will set `thisTransform.position` to `targetTransform.position`. If `faceForward` is set, we will also set `thisTransform.forward` to `targetTransform.forward`.

If you were translating that description to JavaScript while we went, you should have something like this:

```
var targetTransform : Transform;
var faceForward : boolean = false;
private var thisTransform : Transform;

function Start() {
  thisTransform = transform;
}
function Update () {
  thisTransform.position = targetTransform.position;
  if ( faceForward )
    thisTransform.forward = targetTransform.forward;
}
```

Go ahead and save this file to your **Scripts** directory and drag it onto your **CameraPivot** object. **Follow Transform (Script)** will have appeared in your Inspector panel. Go ahead and set the **Target Transform** to **Player**.

Create a new empty GameObject named **CameraOffset** as a child of **CameraPivot**. Set the **Transform** to 0, 0.94, 0—about half the height of Penelope. There won't be any additional Components to attach to this GameObject.

Create a child for CameraOffset named **Camera**. **Camera** should have a **Transform** Position of 0, 3.69, -5.13, and should be rotated 35 degrees around the X axis. Add **Camera** and **GUILayer** Components (both from the **Component > Rendering** menu) to the GameObject. Once you've done this, you can go ahead and delete the default **Main Camera** GameObject from the Scene.

Our camera needs to handle objects that obstruct the view, so we have provided a simple script that uses a zoom value to zoom the camera in or out relative to a default position set in the editor. This script will let our camera snap to values when moving closer to the specified origin and smoothly seek when moving farther away. It also checks for any objects that obstruct the view of the camera and snaps to be in front of those objects. The script is called **ZoomCamera.js**. Drag it onto the **camera** GameObject and in the Inspector, set the Origin value in the Z**oom Camera (Script)** Component to **Player (Transform)**.

Finally, we want to add an **Audio Source** Component to **Camera**. Set the Audio Clip to **musplay**, and check both Play On Awake and Loop.

CameraRelativeControl.js

It is time for us to write the main script for this control setup—**CameraRelativeControl.js**. We didn't provide this one for you, so create a new JavaScript file named **CameraRelativeControl.js**, open it up, and clear the default script that Unity added.

We'll start by requiring that a **CharacterController** Component exists on the GameObject that **CameraRelativeControl.js** is attached to. We also need to declare three public and two private variables. For our public variables, we want references to a joystick script for movement, a camera's **Transform**, and ground speed exposed in the Inspector. For private variables, we want the GameObject's own Transform and the **CharacterController** for caching.

```
@script RequireComponent( CharacterController )
var moveJoystick : Joystick;
var cameraTransform : Transform;
var speed : float = 6;
private var thisTransform : Transform;
private var character : CharacterController;

function Start(){
  thisTransform = GetComponent( Transform );
  character = GetComponent( CharacterController );
}
```

In our `Update` function, we want to start by creating a variable for our movement direction in

the camera-space horizontal direction. By normalizing this vector, we can make sure that the magnitude in any direction is consistent.

```
function Update(){
  var movement =  cameraTransform.TransformDirection( Vector3(
                  moveJoystick.position.x, 0, moveJoystick.position.y ) );
  movement.y = 0;
  movement.Normalize();
}
```

Now, we want to multiply that by a value for speed, which we will get from the largest Component of the joystick position.

```
var absJoyPos =   Vector2( Mathf.Abs( moveJoystick.position.x ),
                  Mathf.Abs( moveJoystick.position.y ) );
movement *=       speed * ( ( absJoyPos.x > absJoyPos.y ) ? absJoyPos.x : absJoyPos.y );
```

Finally, we want to multiply `movement` by `Time.deltaTime` and call `character.Move(movement)` to actually move Penelope.

Save **CameraRelativeControl.js** and attach it to your **Player** GameObject. In the Inspector, set Move Joysick to **leftPad (Joystick)** and Camera Transform to **Camera (Transform)**. Testing the Scene by pressing **Play** will allow us to move Penelope around with the left joystick. Next, we need to adjust it so that she faces the direction that she is moving.

If you closed it when you saved, you need to open **CameraRelativeControl.js** back up and create a function called `FaceMovementDirection`. We are going to get the horizontal velocity by taking `character.velocity` with the Y value zeroed out. We will then check to see if the magnitude of that vector is greater than 0.1. If it is, we will set `thisTransform.forward` to the normalized horizontal velocity vector.

```
function FaceMovementDirection(){
  var horizontalVelocity : Vector3 = character.velocity;
  horizontalVelocity.y = 0;
  if ( horizontalVelocity.magnitude > 0.1 )
      thisTransform.forward = horizontalVelocity.normalized;
}
```

In `Update`, call `FaceMovementDirection` after `character.Move( movement )`. When you've made this change, save and **Play—**behavior for our left joystick is now complete.

Let's add a right joystick to handle camera rotation. Back in **CameraRelativeControl.js**, declare the right joystick below `moveJoystick`.

```
var rotateJoystick : Joystick;
```

We already have **Camera's Transform**, but we will also need **CameraPivot's Transform** since that is what we use for camera rotation. We also want to set the camera rotation speed on each axis.

```
var cameraPivot : Transform;
var rotationSpeed : Vector2 = Vector2( 50, 25 );
```

At the bottom of our `Update` function, we want to scale the joystick input with `rotationSpeed` and then rotate around the character. We'll be rotating in world space horizontally but in local space vertically.

```
var camRotation = rotateJoystick.position;
camRotation.x *= rotationSpeed.x;
camRotation.y *= rotationSpeed.y;
camRotation *= Time.deltaTime;
cameraPivot.Rotate( 0, camRotation.x, 0, Space.World );
cameraPivot.Rotate( camRotation.y, 0, 0 );
```

Save your script. In the Inspector, assign **rightPad (Joystick)** to Rotate Joystick and **CameraPivot (Transform)** to Camera Pivot. Hit Play to test — our left joystick now controls Penelope's movement and the right joystick orbits the camera.

It is time to make Penelope jump. When we created **Joystick.js**, we already set up the behavior for keeping a count of taps. Before we multiply movement by `Time.deltaTime`, we are going to check for a jump. We'll need two public variables for controlling jump speed and a private variable for continuing momentum while in the air.

```
var jumpSpeed : float = 16;
```

```
var inAirMultiplier : float = 0.25;
private var velocity : Vector3;
```

In our `Update` function, we'll add:

```
if ( character.isGrounded ){
  if ( rotateJoystick.tapCount == 2 ){
      velocity = character.velocity;
      velocity.y = jumpSpeed;
  }
} else {
  velocity.y += Physics.gravity.y * Time.deltaTime;
  movement.x *= inAirMultiplier;
  movement.z *= inAirMultiplier;
}
movement += velocity;
movement += Physics.gravity;
```

Before we move the character, we will check to see if `character.isGrounded` returns true, and if it does we will zero out our velocity vector. And with that, our script is pretty much done.

We need to do a little cleanup and then we can move on to our player relative setup. Create an `OnEndGame` function and call disable on both joysticks. Set `this.enabled` to false as well.

```
function OnEndGame() {
  moveJoystick.Disable();
  rotateJoystick.Disable();
  this.enabled = false;
}
```

The entire script is availalbe in both the **Completed Project** folder and the tutorial's Script Appendix for your reference.

Player Relative Setup

PlayerRelativeControl creates a control scheme similar to what might be found in a 3rd person, over-the-shoulder game found on consoles. The left stick is used to move the character, and the right stick is used to rotate the character. A quick double-tap on the right joystick will make the character jump.

Open the **PlayerRelativeSetup** Scene. The Scene Hierarchy will be almost identical to the **CameraRelativeSetup** Hierarchy.

Controls

The **Controls** GameObject is identical—only a few settings in the **Joystick** Components will change. Create your empty **Controls** GameObject and two empty child GameObjects for **Controls** and name them **leftPad** and **rightPad**.

There was some internal debate about whether we should make a Prefab out of the **Controls** GameObject to avoid this section. In the end, re-creating the GameObject is minimal hassle for experienced users and provides a good opportunity for new users to execute an already familiar task. As you use Unity, you will find that there are many similar workflow optimizations in the Editor.

We will begin configuring **leftPad** by adding a **GUITexture** Component to it.

- For Texture, select **thumbTest**. This is an image that we created for the joysticks.
- For Color, choose a middle gray—127,127,127—with 45% opacity.
- For Pixel Inset, set X and Y to 30 and Width and Height to 100.
- All of the remaining values should be set to 0.

Before moving on to rightPad, set the Scale of **leftPad's Transform** to 0,0,1.

Add a **GUITexture** to **rightPad**.

- For Texture, select **thumbTest** again.
- Set Color to the same middle gray.
- For Pixel Inset, set X to 355 and Y to 30. Width and Height should both be 100.
- All of the remaining values should be set to 0.

Adjust **rightPad's** scale to match that of **leftPad**.

- Drag **Joystick.js** onto **leftPad** and **rightPad**. For **leftPad's Joystick** Component, set Position to 0, 0, Dead Zone to 0.25, 0.25 and make sure that Normalized isn't checked.
- For **rightPad's** set Position to 0, 0, Dead Zone to 0, 0.75 and check Normalized.

Player and Camera

The **Player** GameObject in this Scene is a combination of the **CameraRelativeSetup** Scene's **Player** and **Camera** GameObjects.

- Create a new Empty GameObject (**GameObject > Create Empty**).
- Name the new object **Player**.
- Make sure that the **Player** GameObject's **Transform** has a **Position** at 0,0,0, no **Rotation**, and a **Scale** of 1x1x1.
- With **Player** selected, add a **Character Controller** Component (**Component > Physics > Character Controller**)
- In the **Character Controller**, set the Height to 2.19 and the Radius to 0.44.
- Set the **Slope Limit** to 45 and the **Step Offset** to 0.2.
- Set the **Skin Width** to 0.08 and the **Min Move Distance** to 0.
- Set the **Center** values to 0, 1.03, 0.
- Add an **Audio Listener** Component to **Player** (**Component > Audio > AudioListener**)

Drag your **StandIn** Prefab into the Hierarchy as a child of **Player** and scale and position it so that it stands on the **Plane** and fits within the **Character Controller's** Collider.

Set both the Player's **layer** and **tag** to "Player" and change children when prompted.

- Create a new empty child GameObject for **Player** and name it **blob shadow projector**
- Increase the y value of this GameObject's **Transform** position to 3.25 and rotate it 90 degrees around the X axis (so that the local Z axis points straight down)
- Add a **Projector** Component to **blob shadow projector** (**Component > Rendering > Projector**)
- Set the **Near Clip Plane** to 0.1 and the **Far Clip Plane** to 10.
- Set the **Field of View** to 30 and Aspect Ratio to 1.
- For the **Material**, select **shadow material**
- Set Ignore Layers to **Player** so that the shadow isn't projected onto Penelope.

Create a new child for the **Player** GameObject called **CameraPivot**. Set the **Transform** to 0,0,0, make sure that there is no rotation and set the scale to 1x1x1. This will be the outermost parent for our camera setup.

Create a new child for the **CameraPivot** GameObject called **CameraOffset**. Set the **Transform** Position to 0, 1, 0 and make sure that there is no Rotation. Drag **RotationConstraint.js** onto **CameraOffset**. Set the Axis to X and the Min and Max values to -20 and 20, respectively.

Create a child for **CameraOffset** named **Camera**. **Camera** should have a **Transform** Position of 0, 0.57, -1.75, and should have no rotation. Add **Camera** and **GUILayer** Components (both from the **Component > Rendering** menu) to the GameObject. Once you've done this, you can go ahead and delete the default **Camera** GameObject from the Scene.

Just like in the previous scene, our camera needs to handle objects that obstruct the view. Drag **ZoomCamera.js** onto the **Camera** GameObject and in the Inspector, set the Origin value

to **Player (Transform)**. Set Zoom Min to -5 and Zoom Max to 0.5.

Finally, we want to add an **Audio Source** Component to **Camera**. Set the Audio Clip to **mus-play**, and check both Play On Awake and Loop.

PlayerRelativeControl.js

PlayerRelativeControl.js is very similar to **CameraRelativeControl.js**. In fact, rather than walking you through creating it from scratch, we'll focus on the differences.

```
@script RequireComponent( CharacterController )
var moveJoystick : Joystick;
var rotateJoystick : Joystick;
var cameraPivot : Transform;
var forwardSpeed : float = 6;
var backwardSpeed : float = 3;
var sidestepSpeed : float = 4;
var jumpSpeed : float =  16;
var inAirMultiplier : float = 0.25;
var rotationSpeed : Vector2 = Vector2( 50, 25 );
private var thisTransform : Transform;
private var character : CharacterController;
private var animationController : AnimationController;
private var cameraVelocity : Vector3;
private var velocity : Vector3;
```

PlayerRelativeControl.js declares most of the same variables at the top. The difference is that a generic movement speed has been replaced by unique speeds for forward, backwards and sideways movement and that we have declared a variable for the camera's velocity.

```
function Start(){
  thisTransform = GetComponent( Transform );
  character = GetComponent( CharacterController );
}
function OnEndGame(){
  moveJoystick.Disable();
  rotateJoystick.Disable();
  this.enabled = false;
}
```

The `Start` and `OnEndGame` functions are identical to those found in **CameraRelativeControl.js**. There is, however, no `FaceMovementDirection` function in **PlayerRelativeControl.js**.

The `Update` function has a few more differences because we are accounting for differing camera behaviors and directional movement speeds. We begin by setting our movement variable. This is similar to the camera relative version, however it is relative to the **Player's Transform** instead of **Camera's**.

```
var movement =    thisTransform.TransformDirection( Vector3(
                  moveJoystick.position.x, 0, moveJoystick.position.y ) );
```

```
movement.y = 0;
movement.Normalize();
```

Next, we apply movement from the left joystick. This is a little more complicated than our previous version because we are taking multiple movement directions into account. Another difference is that we have introduced a variable named `cameraTarget` and we are offsetting it slightly depending on how we move.

```
var cameraTarget = Vector3.zero;
var absJoyPos = Vector2( Mathf.Abs( moveJoystick.position.x ),
                Mathf.Abs( moveJoystick.position.y ) );
if ( absJoyPos.y > absJoyPos.x ) {
  if ( moveJoystick.position.y > 0 )
      movement *= forwardSpeed * absJoyPos.y;
  else {
      movement *= backwardSpeed * absJoyPos.y;
      cameraTarget.z = moveJoystick.position.y * 0.75;
  }
} else {
  movement *= sidestepSpeed * absJoyPos.x;
  cameraTarget.x = -moveJoystick.position.x * 0.5;
}
```

After getting our `movement` and `cameraTarget` variables, we check for jumping, move the

character and remove any persistent velocity for landing. The only difference here is that we use our `cameraTarget` variable to move the camera back from the character when we jump.

```
if ( character.isGrounded ) {
  if ( rotateJoystick.tapCount == 2 ) {
      velocity = character.velocity;
      velocity.y = jumpSpeed;
  }
} else {
  velocity.y += Physics.gravity.y * Time.deltaTime;
  cameraTarget.z = -jumpSpeed * 0.25;
  movement.x *= inAirMultiplier;
  movement.z *= inAirMultiplier;
}
movement += velocity;
movement += Physics.gravity;
movement *= Time.deltaTime;
character.Move( movement );
if ( character.isGrounded )
  velocity = Vector3.zero;
```

Our next behavior is very different than the camera relative setup. We want to seek our camera towards a target position.

```
var pos = cameraPivot.localPosition;
pos.x = Mathf.SmoothDamp( pos.x, cameraTarget.x, cameraVelocity.x, 0.3 );
pos.z = Mathf.SmoothDamp( pos.z, cameraTarget.z, cameraVelocity.z, 0.5 );
cameraPivot.localPosition = pos;
```

Finally, we will rotate the character around the world Y axis using the X axis of the joystick. We'll then rotate the camera with Y axis input.

```
if ( character.isGrounded ){
  var camRotation = rotateJoystick.position;
  camRotation.x *= rotationSpeed.x;
  camRotation.y *= rotationSpeed.y;
  camRotation *= Time.deltaTime;
  thisTransform.Rotate( 0, camRotation.x, 0, Space.World );
  cameraPivot.Rotate( camRotation.y, 0, 0 );
}
```

Save your script and add it to **Player**. Set Move Joystick to **leftPad** and Rotate Joystick to **right-Pad**. Camera Pivot should be set to **CameraOffset**. The remaining default parameters can stay. When you are done, save the Scene.

A completed version of this script is included in the Script Appendix.

Tap Control Setup

TapControl handles the control scheme in which Penelope is driven by a single finger. When the player touches the screen, Penelope will move toward the finger. The player can also use two fingers to do pinching and twisting gestures for camera zooming and rotation. As dramatically different as this input scheme is from the others, the overall Scene hierarchy is remarkably similar. Before we begin, open **TapControlSetup**.

Player

The **Player** GameObject in this Scene is the same as in **CameraRelativeSetup**.

- Create a new Empty GameObject.
- Name the new object **Player**.
- Make sure that the **Player** GameObject's **Transform** has a Position at 0,0,0, no Rotation, and a Scale of 1x1x1.
- With **Player** selected, add a **Character Controller** Component.
- In the **Character Controller**, set the Height to 2.19 and the Radius to 0.44.
- Set the Slope Limit to 45 and the Step Offset to 0.2.
- Set the Skin Width to 0.08 and the Min Move Distance to 0.
- Set the Center values to 0, 1.03, 0.
- Add an **Audio Listener** Component to **Player**.

Drag your **StandIn** Prefab into the Hierarchy as a child of **Player** and scale and position it so that it stands on the Plane and fits within the **Character Controller**'s Collider.

Set both the **Player's** layer and tag to "Player" and change children when prompted.

Create a new empty child GameObject for **Player** and name it **blob shadow projector**. Increase the y value of this GameObject's **Transform** position to 3.25 and rotate it 90 degrees around the X axis (so that the local Z axis points straight down).

- Add a **Projector** Component to **blob shadow projector**
- Set the Near Clip Plane to 0.1 and the Far Clip Plane to 10.
- Set the Field of View to 30 and Aspect Ratio to 1.
- For the Material, select **shadow material**
- Set Ignore Layers to **Player** so that the shadow isn't projected onto Penelope.

Cameras

The camera system is very similar to the one found in **CameraRelativeSetup**. There are a few differences to be aware of.

Create a new GameObject called **CameraPivot**. Set the **Transform** Position to 0,0,0, make sure that there is no Rotation and set the Scale to 1x1x1. This will be the outermost parent for our camera setup.

In this Scene, **CameraPivot** does not have rotation constraint. It does, however follow the **Player's Transform**. Drag **FollowTransform.js** onto the GameObject. In the Inspector, set Target Transform to **Player**.

Create a new empty GameObject named **CameraOffset** as a child of **CameraPivot**. Set the **Transform** to 0, 0.79, 0—about half the height of Penelope and a little lower than in **CameraRelativeSetup**. There won't be any additional Components to attach to this GameObject.

Create a child for **CameraOffset** named **Camera**. **Camera** should have a **Transform** Position of 0, 8.43, -5, and should be rotated 60 degrees around the X axis. Add **Camera** and **GUILayer** Components to the GameObject. Once you've done this, you can go ahead and delete the default **Camera** GameObject from the Scene.

Drag **ZoomCamera.js** onto the **Camera** GameObject and set the Origin value in the to **Player**.

Finally, we want to add an **Audio Source** Component to **Camera**. Set the Audio Clip to **musplay**, and check both Play On Awake and Loop.

Jump Button

Since there are no joysticks in this control setup, jumping will require a unique button. Create an empty GameObject at world origin and add a **GUITexture** Component.

- Set the Texture to **jumpButton**
- Set the Color to a middle gray with 45% opacity
- Set the Pixel Inset's X and Y values to 410, 6
- Set the Pixel Inset's Width and Height to 64x64
- All other values should be 0.

TapControl.js

The primary scripts in the other two control setups were very similar. Because **TapControl.js** doesn't use joysticks it will be remarkably different.

Create a new script called **TapControl.js**. The heart of this script is a state machine that looks at our touch input and determines what we are doing. At first, we will build the state machine and use `Debug.Log` to output the state.

We'll start by creating an enumeration named `ControlState` that contains the various states that our state machine will need to track.

```
enum ControlState {
  WaitingForFirstTouch,
  WaitingForSecondTouch,
  MovingCharacter,
  WaitingForMovement,
  ZoomingCamera,
  RotatingCamera,
  WaitingForNoFingers
}
```

We will also need a few public and private variables for the state machine.

```
var minimumTimeUntilMove = 0.25;
```

```
var zoomEnabled : boolean = true;

var zoomEpsilon : float = 25;

var rotateEnabled : boolean = true;

var rotateEpsilon : float = 10;

private var state = ControlState.WaitingForFirstTouch;

private var fingerDown : int[] = new int[ 2 ];

private var fingerDownPosition : Vector2[] = new Vector2[ 2 ];

private var fingerDownFrame : int[] = new int[ 2 ];

private var firstTouchTime : float;
```

We need a function that lets us return to the origin state and reset fingers that we are watching.

```
function ResetControlState() {
  state = ControlState.WaitingForFirstTouch;
  fingerDown[ 0 ] = -1;
  fingerDown[ 1 ] = -1;
}
```

For now, our `Start` function will only include a call to `ResetControlState`.

```
function Start(){
  ResetControlState();
}
```

We've also gotten in the habit of including an `OnEndGame` function to stop accepting control changes when the game ends. We'll do that now so that we don't forget to add it later.

```
function OnEndGame(){
  this.enabled = false;
}
```

> UnityRemote inherently introduces latency into the touch input received because the data is being passed back over WiFi. Sometimes you will get an `iPhoneTouchPhase.Moved` event before you have even seen an `iPhoneTouchPhase.Began`. The `Update` function takes this into account to improve the feedback loop when using UnityRemote.

The `Update` function is where we will actually write our state machine. We'll begin by getting `iPhoneInput.touchCount`. If there are no touches, we'll call `ResetControlState`. The rest of our state machine is going to occur inside this conditional check's `else` block.

```
var touchCount : int = iPhoneInput.touchCount;
if ( touchCount == 0 ){
  ResetControlState();
} else {
  // everything else is going to go here
}
```

We are going to need variables in to store touches and to determine if touch events have occurred.

```
var i : int;
var touch : iPhoneTouch;
var touches = iPhoneInput.touches;
var touch0 : iPhoneTouch;
var touch1 : iPhoneTouch;
var gotTouch0 = false;
var gotTouch1 = false;
```

If we are waiting to see if the first finger is down, our state will be `ControlState.WaitingForFirstTouch`. We want to see if that is our current state and, if it is, we want to see if a touch has occurred so that we can update our state machine.

```
if ( state == ControlState.WaitingForFirstTouch ) {
  for ( i = 0; i < touchCount; i++ ) {
      touch = touches[ i ];
      if (  touch.phase != iPhoneTouchPhase.Ended &&
            touch.phase != iPhoneTouchPhase.Canceled ) {
        state = ControlState.WaitingForSecondTouch;
        firstTouchTime = Time.time;
        fingerDown[ 0 ] = touch.fingerId;
        fingerDownPosition[ 0 ] = touch.position;
        fingerDownFrame[ 0 ] = Time.frameCount;
        break;
    }
```

```
  }
}
```

Now we want to see if we are waiting for a second finger to touch down. Otherwise, we will register this as a character move.

```
if ( state == ControlState.WaitingForSecondTouch ) {
  for ( i = 0; i < touchCount; i++ ) {
    touch = touches[ i ];
    if ( touch.phase != iPhoneTouchPhase.Canceled ) {
      if ( touchCount >= 2 && touch.fingerId != fingerDown[ 0 ] ) {
        state = ControlState.WaitingForMovement;
        fingerDown[ 1 ] = touch.fingerId;
        fingerDownPosition[ 1 ] = touch.position;
        fingerDownFrame[ 1 ] = Time.frameCount;
        break;
      } else if ( touchCount == 1 ) {
        var deltaSinceDown = touch.position - fingerDownPosition[ 0 ];
        if ( touch.fingerId == fingerDown[ 0 ] &&
           ( Time.time > firstTouchTime + minimumTimeUntilMove ||
           touch.phase == iPhoneTouchPhase.Ended ) ) {
          state = ControlState.MovingCharacter;
          break;
      }
```

```
      }
    }
  }
}
```

In that block of script, we set state to `ControlState.WaitingForMovement` if the second finger touched down. Next, we will determine what kind of gesture is being made. This section of the script is, admittedly, a bit long.

```
if ( state == ControlState.WaitingForMovement ){
  for ( i = 0; i < touchCount; i++ ) {
    touch = touches[ i ];
    if ( touch.phase == iPhoneTouchPhase.Began ) {
      if  ( touch.fingerId == fingerDown[ 0 ] &&
          fingerDownFrame[ 0 ] == Time.frameCount ) {
        touch0 = touch;
        gotTouch0 = true;
      } else if ( touch.fingerId != fingerDown[ 0 ] && touch.fingerId != fingerDown[ 1 ] ) {
        fingerDown[ 1 ] = touch.fingerId;
        touch1 = touch;
        gotTouch1 = true;
      }
    }
    if (  touch.phase == iPhoneTouchPhase.Moved || touch.phase == iPhoneTouchPhase.Stationary
```

```
        || touch.phase == iPhoneTouchPhase.Ended ){
    if ( touch.fingerId == fingerDown[ 0 ] ) {
      touch0 = touch;
      gotTouch0 = true;
    } else if ( touch.fingerId == fingerDown[ 1 ] ) {
      touch1 = touch;
      gotTouch1 = true;
    }
  }
}

if ( gotTouch0 ) {
    if ( gotTouch1 ) {
      var originalVector = fingerDownPosition[ 1 ] - fingerDownPosition[ 0 ];
      var currentVector = touch1.position - touch0.position;
      var originalDir = originalVector / originalVector.magnitude;
      var currentDir = currentVector / currentVector.magnitude;
      var rotationCos : float = Vector2.Dot( originalDir, currentDir );
      if ( rotationCos < 1 ) {
        var rotationRad = Mathf.Acos( rotationCos );
      if ( rotationRad > rotateEpsilon * Mathf.Deg2Rad ) {
          state = ControlState.RotatingCamera;
        }
      }
```

```
        if ( state == ControlState.WaitingForMovement ) {
          var deltaDistance = originalVector.magnitude -
                              currentVector.magnitude;
          if ( Mathf.Abs( deltaDistance ) > zoomEpsilon ){
            state = ControlState.ZoomingCamera;
        }
      }
    }
  } else {
      state = ControlState.WaitingForNoFingers;
  }
}
```

At this point, let's add `Debug.Log(state)` to our script and attach it to the **Player** object. Press **Play** and take a note of your **Console** messages you will notice that it says WaitingForFirst-Touch. When you touch UnityRemote and hold your finger down, you will notice that it briefly says WaitingForSecondTouch and then MovingCharacter. Lifting your finger will return the state to WaitingForFirstTouch. If you touch two fingers to your device, the state will change to WaitingForMovement. Moving the fingers in opposite directions will update the state to ZoomingCamera and planting one finger and twisting the other around it will update the state to RotatingCamera.

It looks like our state machine is almost complete. We just need to add one more section that keeps feeding changes while we rotate or zoom the camera.

```
if ( state == ControlState.RotatingCamera ||
      state == ControlState.ZoomingCamera ) {
  for ( i = 0; i < touchCount; i++ ) {
    touch = touches[ i ];
    if (  touch.phase == iPhoneTouchPhase.Moved || touch.phase == iPhoneTouchPhase.Stationary
        || touch.phase == iPhoneTouchPhase.Ended ) {
      if ( touch.fingerId == fingerDown[ 0 ] ) {
        touch0 = touch;
        gotTouch0 = true;
      } else if ( touch.fingerId == fingerDown[ 1 ] ) {
        touch1 = touch;
        gotTouch1 = true;
      }
    }
  }
  if ( gotTouch0 ) {
    if ( gotTouch1 ) {
      // Call our Camera Control Function
    }
  } else {
      state = ControlState.WaitingForNoFingers;
  }
}
```

Note that we included a comment in that script for where we should call our camera control function. This function will actually take care of zooming or rotating the camera. Let's also add a comment to call our Character Control function at the end of the `Update` function... if the character needs to move, we will do it there.

Before we can implement either function, we need to expose some new public variables to the Inspector and create some new private variables for caching. Many of these will be familiar from the previous two character controllers.

```
var cameraObject : GameObject;
var cameraPivot : Transform;
var jumpButton : GUITexture;
var speed : float;
var jumpSpeed : float;
var inAirMultiplier : float = 0.25;
var minimumDistanceToMove = 1.0;
var zoomRate : float;
private var zoomCamera : ZoomCamera;
private var cam : Camera;
private var thisTransform : Transform;
private var character : CharacterController;
private var animationController : AnimationController;
private var targetLocation : Vector3;
private var moving : boolean = false;
```

```
private var rotationTarget : float;
private var rotationVelocity : float;
private var velocity : Vector3;
```

In our `Start` function, before we call `ResetControlState`, we are going to cache some Component lookups.

```
thisTransform = transform;
zoomCamera = cameraObject.GetComponent( ZoomCamera );
cam = cameraObject.camera;
character = GetComponent( CharacterController );
```

In the camera relative setup, we used a function called `FaceMovementDirection` to have Penelope face the direction that she was supposed to be going. We'll create an identical function here.

```
function FaceMovementDirection(){
  var horizontalVelocity : Vector3 = character.velocity;
  horizontalVelocity.y = 0;
  if( horizontalVelocity.magnitude > 0.1 )
    thisTransform.forward = horizontalVelocity.normalized;
}
```

Now we are going to write our function for controlling the character. Create a new function called `CharacterControl`. We want to begin this function by getting the `iPhoneInput`.

touchCount. We want to check to see if there is one finger down and if our state is ControlState.MovingCharacter, if that is the case, we want to get the touch event for that finger. Right after we get that touch, we want to check to see if Penelope is jumping. She will be jumping if jumpButton.HitTest(touch.position) and character.isGrounded both return true.

```
var count : int = iPhone.touchCount;
if ( count == 1 && state == ControlState.MovingCharacter ) {
  var touch : iPhoneTouch = iPhoneInput.GetTouch(0);
  if ( character.isGrounded && jumpButton.HitTest( touch.position ) ){
      // let's jump
  } else if (!jumpButton.HitTest( touch.position ) && touch.phase != iPhoneTouchPhase.Began){
      // we aren't jumping, so let's move
  }
}
```

To jump, we will set velocity to character.velocity and velocity.y to jumpSpeed. If we aren't jumping but we need to move, we will raycast to find out where we touched. If the distance to where we touched is greater than the minimumDistanceToMove, we will set the RaycastHit.point as our target location. We'll set our moving boolean to true.

```
var hit : RaycastHit;
if( Physics.Raycast(ray, hit) ) {
  var touchDist : float = (transform.position - hit.point).magnitude;
  if( touchDist > minimumDistanceToMove ) {
      targetLocation = hit.point;
```

```
  }
  moving = true;
}
```

Finally, we will calculate our `movement` variable, move the character, and call the `FaceMovementDirection` function.

```
var movement : Vector3 = Vector3.zero;
if( moving ){
  movement = targetLocation - thisTransform.position;
  movement.y=0;
  var dist : float = movement.magnitude;
  if( dist < 1 ){
      moving = false;
  } else {
      movement = movement.normalized * speed;
  }
}
if ( !character.isGrounded ){
  velocity.y += Physics.gravity.y * Time.deltaTime;
  movement.x *= inAirMultiplier;
  movement.z *= inAirMultiplier;
}
movement += velocity;
```

```
movement += Physics.gravity;
movement *= Time.deltaTime;
character.Move( movement );
if ( character.isGrounded )
  velocity = Vector3.zero;
FaceMovementDirection();
```

Replace the comment that you made about where to call the character control function with an actual call to `CharacterControl`. Save the script and, in the Inspector, set Camera Object to **Camera**, Camera Pivot to **CameraPivot**, and Jump Button to **JumpButton**. If you test the Scene by pressing **Play**, you will notice that Penelope now runs and jumps across the plane.

Now we need to write our camera control function. `CameraControl` will take two `iPhoneTouch` events, `touch0` and `touch1`. The first part of the function will deal with rotation if `rotateEnabled` is true and the state is `ControlState.RotatingCamera`. The second part of the function will deal with zooming if `zoomEnabled` is true and the state is `ControlState.ZoomingCamera`.

```
function CameraControl( touch0 : iPhoneTouch, touch1 : iPhoneTouch ){
  if ( rotateEnabled && state == ControlState.RotatingCamera ){
      // rotation stuff
  } else if ( zoomEnabled && state == ControlState.ZoomingCamera ){
      // zooming stuff
  }
}
```

For the rotation portion of the function, we want to get the rotation amount between last frame and the current frame. We will accumulate rotation change with our target rotation and wrap the rotation values to keep them between 0 and 360 degrees.

```
var currentVector : Vector2 = touch1.position - touch0.position;
var currentDir = currentVector / currentVector.magnitude;
var lastVector : Vector2 = ( touch1.position - touch1.deltaPosition ) - ( touch0.position -
touch0.deltaPosition );
var lastDir = lastVector / lastVector.magnitude;
var rotationCos : float = Vector2.Dot( currentDir, lastDir );
if ( rotationCos < 1 ) {
  var   currentVector3 : Vector3 =  Vector3( currentVector.x, currentVector.y );
  var lastVector3 : Vector3 = Vector3( lastVector.x, lastVector.y );
  var rotationDirection : float = Vector3.Cross( currentVector3, lastVector3 ).normalized.z;
  var rotationRad = Mathf.Acos( rotationCos );
  rotationTarget += rotationRad * Mathf.Rad2Deg * rotationDirection;
  if ( rotationTarget < 0 )
      rotationTarget += 360;
  else if ( rotationTarget >= 360 )
      rotationTarget -= 360;
}
```

For the zoom portion of the function, we will accumulate the pinch change with our target zoom.

```
var touchDistance = ( touch1.position - touch0.position ).magnitude;
var lastTouchDistance =   ((touch1.position - touch1.deltaPosition ) -
                          ( touch0.position - touch0.deltaPosition ) ).magnitude;
var deltaPinch = touchDistance - lastTouchDistance;
zoomCamera.zoom += deltaPinch * zoomRate * Time.deltaTime;
```

That takes care of our `CameraControl` function. Replace your comment about calling a camera control function with `CameraControl( touch0, touch1)`.

The last thing that we want to do is smoothly seek towards the target location in `LateUpdate`.

```
function LateUpdate() {
   cameraPivot.eulerAngles.y =  Mathf.SmoothDampAngle( cameraPivot.eulerAngles.y,
                                rotationTarget, rotationVelocity, 0.3 );
}
```

Save the script and test it out by pressing **Play**. For reference, **TapControl.js** is located in both the **Completed Project** and Script Appendix.

Introducing... The Real Penelope

It's time to get rid of our stand-in and get Penelope animated. Import **PenelopeArtwork.unityPackage**. This asset package contains the Penelope model, textures and materials. From the **Objects** folder, drag **penelopeFBX** onto the **Player** object in each of the control setup scenes. Make sure that the model's **Transform** Position is set to 0,0,0 and that it is Scaled 1x1x1. Once this is done, you can remove the StandIn object—we don't need it now that the real thing is in place.

We've replaced our stand-in with Penelope, but she isn't very lively—let's take care of that by writing some scripts to take care of her animation.

AnimationController.js

Create a new script called **AnimationController.js.** This script will be attached to **Player**. This script plays the appropriate animations for Penelope and controls the blending between them. It uses the character's movement direction to determine which animation should be played.

Before we start writing the script, let's look at how animation in Unity works. The actual animating of characters is done through Unity's scripting interface. Animators create separate animations, e.g. a walk cycle, run cycle, idle or shoot animation. At in point in time in your game, you need to be able to transition from the idle animation into the walk cycle and vice versa. Of course you don't want any sudden jumps in the motion. You want the animation to smoothly transition. This is where animation blending comes in. In Unity you can have an

arbitrary number of animations playing on the same character. All animations are blended or added together to create the final animation.

We are going to need to access Penelope's **Animation** Component, so in **AnimationController.js**, we will declare a variable of type `Animation` named `animationTarget` and expose it in the Inspector.

In our `Start` function, we are going to configure the `wrapModes` for each of our `AnimationStates`. An animation state's wrap mode determines how time beyond the playback range should be treated. We will begin by setting the wrap mode for the **Animation** Component to `wrapMode.Loop`. The default wrap mode for each animation state is initialized to the value set in the animation Component's wrap mode. Since we want most of our clips to loop by default, `wrapMode.Loop` is a good choice. We want a different wrap mode for several of our animation states. For those, we will explicitly set them to `WrapMode.ClampForever`. `WrapMode.ClampForever` plays back the animation. When it reaches the end, it will keep playing the last frame and never stop playing.

```
var animationTarget : Animation;
function Start(){
  animationTarget.wrapMode = WrapMode.Loop;
  animationTarget["jump"].wrapMode = WrapMode.ClampForever;
  animationTarget["jump-land"].wrapMode = WrapMode.ClampForever;
  animationTarget["run-land"].wrapMode = WrapMode.ClampForever;
  animationTarget["LOSE"].wrapMode = WrapMode.ClampForever;
}
```

We need to set some variables that we will use for controlling which animations to play. Create three public floats named `maxForwardSpeed`, `maxBackwardSpeed` and `maxSidestepSpeed`. Their values should be 6, 3 and 4, respectively. These variables will describe how fast Penelope can move in different directions.

You also need to create two private variables. The first will be a `boolean` named `jumping`. It will be used as a flag to check whether Penelope is jumping. We also want to set a minimum upward speed that she has to exceed in order to jump. Create another private variable named `minUpwardSpeed` and set the value to 2.

```
var maxForwardSpeed : float = 6;
var maxBackwardSpeed : float = 3;
var maxSidestepSpeed : float = 4;
private var jumping : boolean = false;
private var minUpwardSpeed = 2;
```

Before we get into the meat of the `Update` function, we want to cache some Components so that we don't have to look them up every frame. To do this, we will create two more private variables. We will set the values for these variables once in the `Start` function.

Create a variable named `character` of type `CharacterController` and a variable named `thisTransform` of type `Transform`. In `Start`, set `character` to `GetComponent(CharacterController)` and `thisTransform` to `transform`. Now we can reference these local variables instead of doing the Component lookups each frame.

```
var character : CharacterController;
var thisTransform : Transform;
function Start() {
  character = GetComponent( CharacterController );
  thisTransform = transform;
  // ...
}
```

The `Update` function is going to contain a lot of conditional checks to determine which animation to play. Before we start writing them, we need to get some information about Penelope's horizontal and vertical movement.

First, we are going to create a variable named `characterVelocity` and set it to the velocity from the **CharacterController** Component. We want to separate horizontal and vertical movement for our animation logic. For horizontal motion, we will use the `characterVelocity` vector, but will zero the Y value. A variable, `speed`, will be the magnitude of this vector. For Penelope's upwards motion, we will take the dot product of `transform`.up (remember that transform was cached as thisTransform) and the `characterVelocity`.

We also want to know if Penelope is jumping or not. This is the `jumping` boolean that we mentioned when setting `minUpwardSpeed`. Penelope is jumping if her `CharacterController`.`isGrounded` is false and her `upwardsMotion` is greater than `minUpwardSpeed`.

At this point, the `Update` function should look something like this:

```
function Update() {
  var characterVelocity = character.velocity;
  var horizontalVelocity : Vector3 = characterVelocity;
  horizontalVelocity.y = 0;
  var speed = horizontalVelocity.magnitude;
  var upwardsMotion = Vector3.Dot( thisTransform.up, characterVelocity );
    if ( !character.isGrounded && upwardsMotion > minUpwardSpeed )
    jumping = true;
}
```

We now have enough information available to us to start working out the logic behind our animation controller. We will be using a series of four conditional checks to determine broadly what Penelope is doing. The first set of conditions that we want to check for is whether the **run-land** animation is playing. If it is, and if Penelope is moving, we want to let the animation finish. If that animation isn't playing, the second thing we want to check is to see if the **jump-land** animation is playing. If it isn't, we want to check if our `jumping` boolean is true. If Penelope isn't jumping, we want to check to see if she is moving at all. Finally, if all else fails, we will play her idle animation. Sound complicated? A little JavaScript should clear it up:

```
if  ( animationTarget.IsPlaying( "run-land" ) &&
    animationTarget[ "run-land" ].normalizedTime < 1.0 &&
    speed > 0 ) {
  // ...
} else if ( animationTarget.IsPlaying( "jump-land" ) ) {
```

```
  // ...
} else if ( jumping ) {
  // ...
} else if ( speed > 0 ) {
  // ...
} else {
  // ...
}
```

If the first set of conditions were met, we literally don't want to do anything... just let the animation play and leave that block empty.

If **jump-land** is playing, we probably just want to let the animation finish, so we most likely don't want to do anything there either. We do, however, need to check to see if the animation is done playing so that we can go back to idle. To do this, we will check to see if the `normalizedTime` for the animation is greater than or equal to one.

If Penelope is jumping, we have a couple of checks to make. First, we want to see what `CharacterController.isGrounded` is set to. If it is false, we can play the jump animation. If it is true, however, we need to also check Penelope's `speed`. If she is moving, we will play the **run-land** animation. Otherwise, we will play the **jump-land** animation.

By far our most complex response is to the next block—if `speed` is greater than zero. We need to begin by separating Penelope's forward motion from her sideways motion. We will find this

in a similar way to how we found her upwards motion, but this time we will be using the dot product of the relevant transform directions and the `horizontalVelocity` vector instead of the overall velocity.

We are going to be doing a lot of interpolation between values here, so we are going to go ahead and declare a variable named `t` that we will use when we lerp.

```
} else if ( speed > 0 ) {
  var forwardMotion = Vector3.Dot( thisTransform.forward, horizontalVelocity );
  var sidewaysMotion = Vector3.Dot( thisTransform.right, horizontalVelocity );
  var t = 0.0;
}
```

We want to use the largest movement direction to determine which animations we will play. We will determine this by comparing the absolute values of `forwardMotion` and `sidewaysMotion`. If the primary movement is along the forward/backward axis, we will be playing the **run** or **runback** animations. We can determine which animation to play by checking to see if `forwardMotion` is greater than zero. In either case, we will need to adjust the speed of the animation to match how fast the character is moving. We will do this by taking the absolute value of `speed` divided by the maximum speed and clamping it between zero and the maximum speed where the maximum speed for forward and backward movement is `maxForwardSpeed` or `maxBackwardSpeed`, respectively. We will assign this value to `t` and will then interpolate 0.25 to 1 by `t` and assign that to the animation speed. If `forwardMotion` was greater than zero, we need to check to see if **run-land** or **idle** is playing. If either is playing, Penelope

has just landed a jump, and we want to play the **run** animation straight away. If not, we want to CrossFade to it. If Penelope is running backwards, no check has to be made. We will always CrossFade to **runback**.

```
if ( Mathf.Abs( forwardMotion ) > Mathf.Abs( sidewaysMotion ) ){
  if ( forwardMotion > 0 ){
    t = Mathf.Clamp( Mathf.Abs( speed / maxForwardSpeed ), 0, maxForwardSpeed );
    animationTarget[ "run" ].speed = Mathf.Lerp( 0.25, 1, t );
    if ( animationTarget.IsPlaying( "run-land" ) || animationTarget.IsPlaying( "idle" ) )
      animationTarget.Play( "run" );
    else
      animationTarget.CrossFade( "run" );
  } else {
    t = Mathf.Clamp( Mathf.Abs( speed / maxBackwardSpeed ), 0, maxBackwardSpeed );
    animationTarget[ "runback" ].speed = Mathf.Lerp( 0.25, 1, t );
    animationTarget.CrossFade( "runback" );
  }
}
```

Dealing with sideways motion is less complex. We are going to calculate our animation speed just like we did above, using `maxSidestepSpeed` in our calculation of `t` for both **runright** and **runleft** animations. To determine which animation we will crossfade, we will check to see if `sidewaysMotion` is greater than zero.

```
} else {
  t = Mathf.Clamp( Mathf.Abs( speed / maxSidestepSpeed ), 0, maxSidestepSpeed );
  if ( sidewaysMotion > 0 ) {
    animationTarget[ "runright" ].speed = Mathf.Lerp( 0.25, 1, t );
    animationTarget.CrossFade( "runright" );
  } else {
    animationTarget[ "runleft" ].speed = Mathf.Lerp( 0.25, 1, t );
    animationTarget.CrossFade( "runleft" );
  }
}
```

Add a CrossFade to **idle** if none of our four main conditions were met, and **AnimationController.js** is nearly done.

```
} else {
  animationTarget.CrossFade( "idle" );
}
```

Before you save the script and attach it to the **Player** object, we need to create an `OnEndGame` function and set `this.enabled` to false in it.

```
function OnEndGame() {
  this.enabled = false;
}
```

With the **AnimationController.js** on all three of our **Player** objects, we need to modify the control scripts to send speed information for animation syncing. In each of the three scripts (**CameraRelativeControl.js**, **PlayerRelativeControl.js** and **TapControl.js**), you need to declare a private variable.

```
private var animationController : AnimationController;
```

In the `Start` function of **CameraRelativeControl.js** and **TapControl.js**, add:

```
animationController = GetComponent( AnimationController );
animationController.maxForwardSpeed = speed;
```

In the `Start` function of **PlayerRelativeControl.js**, add:

```
animationController = GetComponent( AnimationController );
animationController.maxForwardSpeed = forwardSpeed;
animationController.maxBackwardSpeed = backwardSpeed;
animationController.maxSidestepSpeed = sidestepSpeed;
```

Penelope is now fully animated. You can reference completed versions of the scripts that you just wrote in the Script Appendix.

In our completed project file, you will notice a second animation script—**AnimationDebug.js.** We aren't going to go into detail about the script—it was used to debug animations while building the controls. If you have time, taking a look at how this script works might be helpful in your own projects.

What You Learned

In this Part of the tutorial, we focused on building three real-world third-person control setups. By the end, you had created these Scenes from scratch and were controlling a fully-animated character. The Scenes that you created can be adapted for a number of Unity iPhone projects and many of the skills that you developed can be used for desktop or web-based deployments as well.

Part Three:

Emeracite Mine

The control setup Scenes that you worked on in the last part of this tutorial will be additively loaded into a Scene containing the game level. This level, **EmeraciteMine** in the root of the **Completed Project**, also contains the introduction, control options, and initial animation.

What's in the Emeracite Mine?

To begin, let's open up the completed **EmeraciteMine** Scene in the **Completed Project** and take a look at the Hierarchy View to get a feel for how the Scene is structured.

allColliders — This object contains, as child objects, all the Colliders used in this Scene. With one exception, all Colliders consist of combinations of primitive (capsule, cube and sphere) Colliders. More details on how Colliders were set can be found in the next section.

Camera — This is our default camera. It has several scripts for animating it during the fly-through. We will walk you through this camera's setup later in this part of the tutorial.

cameraAnimation — This is an imported fbx animation from Maya. It contains a locator with a motion path. The camera will follow this object to create the fly-through effect.

ControlMenu — ControlMenu creates the menu from which the player can choose which control scheme to play. It makes use of Unity's GUILayout system to create buttons. The menu loads a background image so that the player can't see the transitions between the different Scenes which contain the control schemes. **ControlMenu** is also responsible for starting the launch intro animation and for activating the orb emitter. You will begin creating this object later in this part of the tutorial.

DepositArea — This is a Collider that triggers the orb deposit action. It is detailed in the next part of this tutorial.

depositoryActivationGraphics — This object contains visual effects related to the depositing of orbs.

LaunchIntro (disabled) **—** This object, and its children, are used for the animated "eruption" sequence. The object is enabled at the appropriate time by **ControlMenu**. Later in this part of this tutorial, we will detail the building of this object.

mainLevelFBX — This is our actual imported level artwork. The object also contains the level's skybox.

PlayerSpawn — PlayerSpawn's Transform Position is the location for character to spawn during additive level loading.

rocksGlow — This contains additional geometry used for the rocks in the mining carts. The material **Rocks Glow Mat** uses an Additive Culled particle shader.

sunLight — This object has a single directional light which is used to light the character.

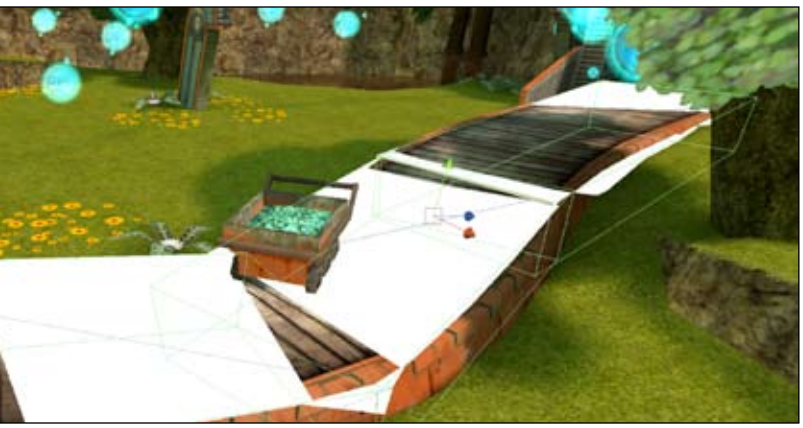
theOrbs — theOrbs contains behaviors relating to orb generation. The child GameObjects (named **orbLocation**) determine the location of potential orb spawn points throughout the level. We will actually be building this part of the Scene in this part of this tutorial.

waterController — This object contains a simple script (**WaterMovement.js**) for animating the

textures on two water meshes. There is more on how water was created in the next section.

Mine Level Design

As you can see, many of these objects rely on third-party modeling or texturing software (and a talented artist) to create their magic. Rather than simply providing a completed Scene to continue the tutorial with, we are going to pause for a moment to point out how we have addressed several typical artistic and creative challenges. Later in this part of the tutorial, we will begin creating the objects and writing the scripts necessary to add interaction to our Scene.

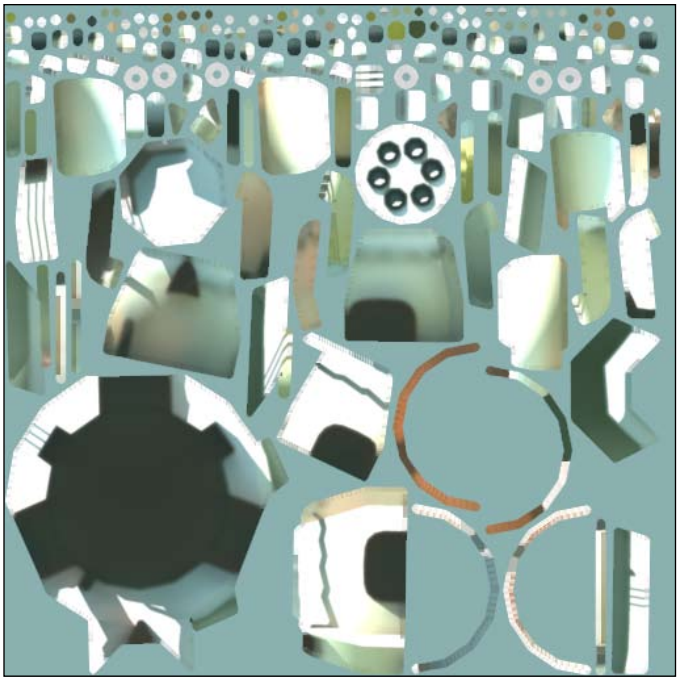


Colliders

allColliders is a parent to all Colliders in the Scene. With one exception, all Colliders consist of combinations of primitive (capsule, cube and sphere) Colliders. For the ground, we have used a **Mesh Collider** that uses a simplified version of the mesh used for rendering. In other areas, these primitive Colliders create a rough approximation of the level geometry. By selecting these Colliders in the **Hierarchy View**, you can see their highlighted forms in the Scene view. Turning on the **Mesh Renderers** for these objects allows us to clearly see how they have been placed to approximate, but not to exactly match, the level geometry.

The result of this slight mismatch is that our character, Penelope, will sometimes hover slightly above the ground or sink slightly into it. The effect will be too minimal for most players to notice, especially on the iPhone, but the performance gains are critical to the success of the project.

Textures and Lightmaps



By using lightmaps for most of our level artwork, we are able to minimize the number of in-game lights required for quality visuals. Lightmapped shaders use a secondary Lightmap texture that defines light that has been collected by the object. The Lightmap texture also uses the secondary UV map of the mesh for controlling the placement of light, which can differ from the primary UV map. These shaders allows you to use baked colored light, radiosity with color bleeding and other interesting effects.

Your game's draw calls are how many separate objects are rendered in total. This accumulates objects that are drawn multiple times as well, for example by each pass in a shader or by having multiple materials. Keeping draw calls to a minimum is important for developing a game on the iPhone. Ideally, we want our draw calls to be at or below 30. Fortunately, Unity automatically combines small meshes (less than 300 triangles) into single draw calls if those objects share the same material. To take advantage of this, it is important for objects to use a shared texture sheet.

Alpha Textures for Faking Geometry

The iPhone hardware requires you to carefully budget vertices. Drawing fewer polygons per object means that you can draw more objects. One technique for making low-polygon objects appear to have more detailed geometry is to use a material with a shader that supports alpha transparency. With a wireframe overlay on the level's vegetation, you can see how we are able to have objects use far fewer polygons than we could if we were attempting to model edge geometry.

Faking Effects

Creating visually convincing water can be a challenge with advanced rendering techniques, and it can be downright difficult without them. To create the water in this level, several techniques were employed. The reflection of level geometry was created by mirroring the geometry below the water's surface. Because the artwork was already lightmapped, this mirrored geometry has accurate light and shadow. The water surface uses a shiny transparent Material. A script attached to an object named **WaterController.js** animates the texture offset for the water. This creates the appearance of movement and churn created by the water that flows (also through texture offsets) from the pipes.

Your iPhone game may not have water in it, but coming up with creative solutions to similar visual challenges that you might have solved with expensive rendering techniques on a desktop is a critical skill for iPhone game development.

Before we move on, open your **Tutorial Project** in Unity iPhone and load the package **part3.unityPackage**. This file contains a base **EmeraciteMine** Scene and the assets needed to complete this part of the tutorial.

Flying Around

Open the **EmeraciteMine** Scene. You will notice that several of the objects that were present in the completed version of this Scene are not included here. We are going to create them.

The first thing that we need to do is create the camera animation that happens while the title screen displays. In your Scene, you will find an imported FBX named **cameraAnimation**. This file is a looping animation of an object, called **cameraLocator**, along a path. We are going to write two scripts and attach them to our **Camera**. The first script will allow us to adjust the speed of **cameraAnimation's** playback. The second will force the camera to follow **cameraLocator's** Transform Position around the level.

AnimationSpeed.js

Create a new JavaScript file named **AnimationSpeed.js** and open it in your text editor of choice. You can delete the functions automatically generated by Unity.

This script is going to be incredibly simple. We are going to set an animation state's speed

value to a user-controllable value in the Inspector. To do this, the user will need to be able to specify both the speed and the animation that will be effected, so we need to expose those values in the Inspector. We will name the variable for speed `speed`, and the target animation `animationTarget`. We want the `speed` variable of the animation state for the default animation clip in our target animation. This can be accessed through `Animation[Animation.clip.name].speed`. Create a `Start` function in your script and `animationTarget[ animationTarget.clip.name ].speed = speed;` Your finished file should look like this. Save it to your **Scripts** folder and drag it onto your **Camera** object.

```
var animationTarget : Animation;
var speed = 1.0;
function Start() {
  animationTarget[ animationTarget.clip.name ].speed = speed;
}
```

A Component, **Animation Speed (Script)**, was added in your **Camera** object's Inspector window. Set the `animationTarget` variable by clicking on the selector arrow to the right of Animation Target. Animations in the scene will be shown on a selectable list. You want to choose **cameraAnimation**. The default value that we set for speed was 1.0. For this project, change the value to 0.2 in the Inspector.

FollowTransform.js

Find **FollowTransform.js** in your **Scripts** directory and drag it onto your **Camera** object. This is the same script that we used before in the control setup Scenes. Follow Transform (Script)

will have appeared in your **Camera's** Inspector panel. Go ahead and set the Target Transform to **cameraLocator** (as you did with Animation Target in the previous script) and check the Face Forward box.

Let's test our scene by hitting the **Play** button. We will see that the camera slowly flies around the **EmeraciteMine** Scene.

Adding the Logo

Begin by creating an empty GameObject and naming it **ControlMenu**. This object will do a whole lot more in the near future when we write our control menu script. Right now, however, we are going to quickly add a **GUI-Texture** for the Penelope logo.

- Add a **GUITexture** Component.
- For the texture, select **PenelopeMenuFinalTGA** from the **Textures** folder.
- Set the Pixel Inset values. You can do the math yourself (based on the screen and texture sizes) or you can just set X to 112, Y to 96, Width to 256 and Height to 128 (if you believe us).
- Press **Play** to check it out—the logo now sits on top of the animated **Camera** fly-through

Control Menu

We are going to start writing our control menu script. In this part of the tutorial, we will focus on determining if a user clicks on the logo. If they do, we will initiate the **LaunchIntro** and

destroy the animated fly-through setup. In the next part of the tutorial, we will be expanding the script to add additional functionality.

Begin by creating a new JavaScript file, **ControlMenu.js**. Delete the automatically generated contents of the file and add it as a Component to the **ControlMenu** object—we want to test as we go.

The first thing that we are going to do is write an `Update` function that will check to see if the logo was touched. If it was, it will disable the **GUITexture**. We need to check to see if `iPhoneInput.touchCount` is greater than zero. If it is, we want to iterate over each touch. For each touch, we want to check to see that it just started (`touch.phase` is `iPhoneTouchPhase.Began`) and do a hit test with the **GUITexture**. If a touch passes that, we will disable the **GUITexture**.

```
function Update () {
  if (iPhoneInput.touchCount > 0 ) {
    for(var i : int = 0; i< iPhoneInput.touchCount;i++) {
      var touch : iPhoneTouch = iPhoneInput.GetTouch(i);
      if  ( touch.phase == iPhoneTouchPhase.Began && guiTexture.HitTest(touch.position)) {
        guiTexture.enabled = false;
      }
    }
  }
}
```

Test by pressing **Play**. With your iPhone running Unity Remote, you should be able to click on

the Penelope logo to get it to disappear. Now we need to destroy the objects related to the fly-through and activate the **LaunchIntro**. We'll start by destroying some objects. In **ControlMenu.js**, we are going to add a public variable called `destroyOnLoad` to hold the objects that we are getting rid of. In this function, we will iterate over all of the objects in `destroyOnLoad` and destroy them. As a final step, we will also destroy our **ControlMenu**.

```
var destroyOnLoad : Transform[];
function Update () {
  if (iPhoneInput.touchCount > 0 ){
    for(var i : int = 0; i< iPhoneInput.touchCount;i++) {
      var touch : iPhoneTouch = iPhoneInput.GetTouch(i);
      if  ( touch.phase == iPhoneTouchPhase.Began &&
          guiTexture.HitTest(touch.position)) {
        guiTexture.enabled = false;
        Launch();
      }
    }
  }
}

function Launch() {
  for ( var t in destroyOnLoad )
    Destroy( t.gameObject );
}
```

Before we can test, we need to add some objects to get destroyed. In the Inspector, set the Size of Destroy On Load to 2, and add **Camera (Transform)** and **cameraAnimation (Transform)** to Elements 0 and 1, respectively. Testing now will show that we do indeed destroy these elements when we click the logo. Now we need to create our **LaunchIntro** so that we can enable it to take **ControlMenu's** place.

We haven't created our **LaunchIntro** yet, but having an idea about how it will behave will help us write this script. First, everything related to the **LaunchIntro** will be the child of a single GameObject. Second, we will begin with all of the **LaunchIntro** objects all deactivated. Third, when we want to start the **LaunchIntro**, all we will have to do is activate all the objects. Knowing these things, we will expose a variable named `launchIntro` in the Inspector. We will also make sure that the object referenced in this variable is disabled when we start by creating a `Start` function and adding `launchIntro.SetActiveRecursively(false)` to it. In our `Launch` function, we will do the opposite—adding `launchIntro.SetActiveRecursively(true)` after destroying the objects in `destroyOnLoad`.

```
var destroyOnLoad : Transform[];
var launchIntro : GameObject;

function Start(){
  launchIntro.SetActiveRecursively( false );
}
```

```
function Launch(){
  for ( var t in destroyOnLoad )
    Destroy( t.gameObject );
  launchIntro.SetActiveRecursively( true );
  Destroy( GameObject);
}
```

The Launch Intro

ControlMenu is done for now, but we need to create our **LaunchIntro** object to add to it in the Inspector. **LaunchIntro** will do two main things—it will shake the camera and then launch a series of (fake) orbs. The events will be tied to audio clips in order to synch sound.

Begin by creating an empty GameObject and naming it **LaunchIntro**. Set the **Transform** for this object so that it sits in the middle of the somewhat spherical top portion of the central power converter (the level's main structure).

Create a second empty object. Name this one **CameraPivot** and set it as a child of the **LaunchIntro** object. Create a new camera GameObject named **Camera** and set it as a child of **CameraPivot**. Locally, this object should have o for all Position and Rotation Transform values in the Inspector. When you added the camera, your game view changed to show the view from this camera. This will help position **CameraPivot**. Move it above the scene and have it look down on the the central power converter. Where you place it will determine where the camera is during the **LaunchIntro** animation. You can reference this screenshot from our project for camera placement. It isn't important to match our camera location exactly (although

you can reference the values in the completed project, if you like).

When we shake **Camera**, we will do so by adjusting its transform values, but it will continue to inherit the overall transform of **CameraPivot**. Create one last object. This one should be a Plane named **CollisionPlane** that is set as a sibling to **CameraPivot**. Set the Scale values for the plane to 20 and set the Location to 0, -7, 0.

LaunchIntro.js

Create a new script named **LaunchIntro.js**. We will start by creating two variables to expose to the Inspector. These will be for the audio clips (**sfx-rumble** and **sfx-boom**). Name them `rumbleSound` and `boomSound`. We will also have two private vars so that we can cache our `Transform` and `AudioSource` at `Start`. We will call them `thisTransform` and `thisAudio`.

In the `Start` function, we will set `thisTransform` to `transform` and `thisAudio` to `audio`. We will then play the rumble sound by calling `thisAudio.PlayOneShot(rumbleSound, 1.0);` While that is playing, we want to shake the camera. We will create a new function called

CameraShake. In CameraShake, we will create a Vector3 with random values for X and Y. We will assign this to the localEulerAngles of thisTransform. Back in Start, we will add InvokeRepeating("CameraShake", 0, 0.05);

We want to play **boomSound** when the rumble is done. We will create a new function called Launch. In it, we will add thisAudio.PlayOneShot(boomSound, 1.0) and cancel the invoke for the shaking camera. Back in Start, we will invoke Launch after rumbleSound.length time has passed.

```
var rumbleSound : AudioClip;
var boomSound : AudioClip;
private var thisTransform : Transform;
private var thisAudio : AudioSource;

function Start()  {
  thisTransform = transform;
  thisAudio = audio;
  thisAudio.PlayOneShot( rumbleSound, 1.0 );
  InvokeRepeating( "CameraShake", 0, 0.05 );
  Invoke( "Launch", rumbleSound.length );
}

function CameraShake() {
  var eulerAngles = Vector3( Random.Range( 0, 5 ), Random.Range( 0, 5 ), 0 );
```

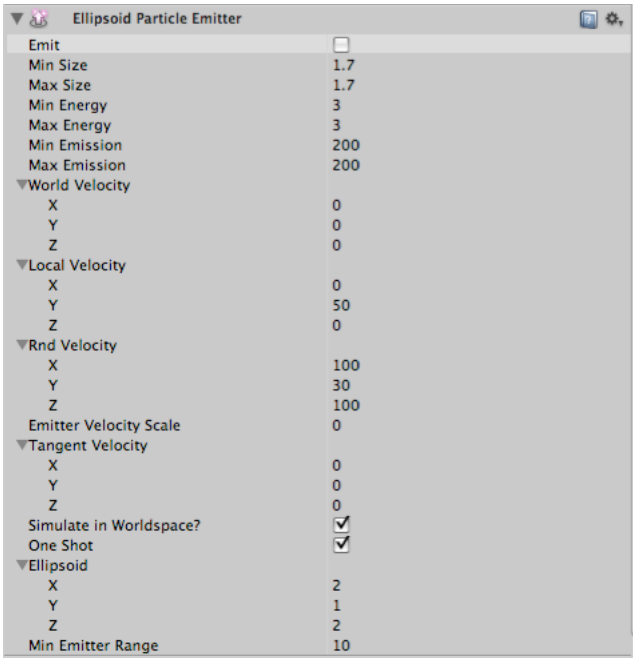
```
  thisTransform.localEulerAngles = eulerAngles;
}

function Launch() {
  thisAudio.PlayOneShot( boomSound, 1.0 );
  Invoke( "CancelInvoke", 0.5 );
}
```

Save your file and attach it to the **Camera** object that you created a few minutes ago. In the **Camera's** Inspector, set Rumble Sound to **sfx-rumble** and Boom Sound to **sfx-boom**.

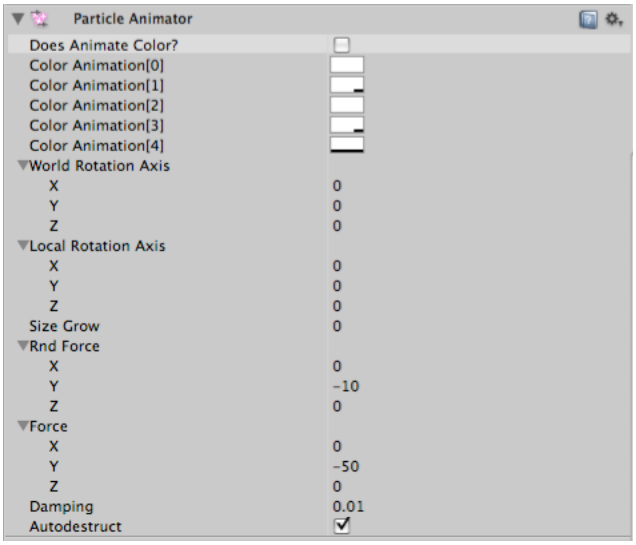
Before we test this out, in **LaunchIntro's** Inspector, uncheck the box to deactivate the object. When prompted, press **Deactivate Children**. If we play now, clicking on the logo will switch camera views to a bird's eye view over the central power converter. The camera will shake and we will hear a rumbling sound followed by a loud explosion. There aren't, however, any orbs launched during the explosion. We need to fix that.

We are going to create some fake orbs (just cosmetic, not the ones that are actually collected) to finish off this Scene. To do this, we will add some Particle-related Components to our **LaunchIntro** GameObject.

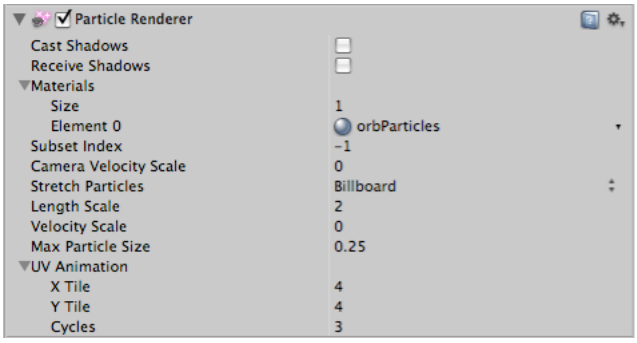
- Add an Ellipsoid Particle Emitter



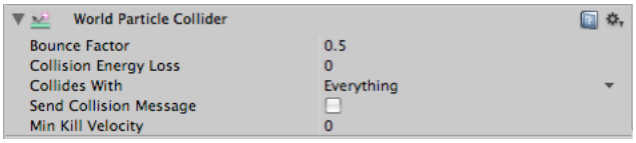
- Add a Particle Animator



- Add a Particle Renderer



- Add a World Particle Collider



We also need to make a few additions to our **LaunchIntro** script. We are going to expose a `ParticleEmitter` named `spawnParticleEmitter` in the Inspector. We will also modify our `Launch` function to set `spawnParticleEmitter.emit` to true.

```
var spawnParticleEmitter : ParticleEmitter;

var rumbleSound : AudioClip;

var boomSound : AudioClip;

private var thisTransform : Transform;

private var thisAudio : AudioSource;


function Start() {

  thisTransform = transform;

  thisAudio = audio;
```

```
  thisAudio.PlayOneShot( rumbleSound, 1.0 );
  InvokeRepeating( "CameraShake", 0, 0.05 );
  Invoke( "Launch", rumbleSound.length );
}

function CameraShake() {
  var eulerAngles = Vector3( Random.Range( 0, 5 ), Random.Range( 0, 5 ), 0 );
  thisTransform.localEulerAngles = eulerAngles;
}

function Launch() {
  spawnParticleEmitter.emit = true;
  thisAudio.PlayOneShot( boomSound, 1.0 );
  Invoke( "CancelInvoke", 0.5 );
}
```

Set Spawn Particle Emitter to **LaunchIntro (Ellipsoid Particle Emitter)** in the Inspector. If we test our game now, it finishes with an explosion of (fake) orbs.

What You Learned

In this part of the tutorial, we started with a tour of the EmeraciteMine Scene. We went over different techniques for creating quality artwork for game levels on the iPhone. After reviewing the Scene, you went on to build an automated introductory animation and a pre-scripted sequence based on sound effects. While these two final things aren't strictly necessary (we could have had a title/menu screen and just jumped straight into gameplay) they are a detail seldom seen on mobile games.

Part Four:

Gameplay

Bringing In Penelope

It is time for us to bring Penelope into the main Scene. We will start by adding a couple lines of script to each of our three control scripts. In the `Start` functions, add:

```
var spawn = GameObject.Find( "PlayerSpawn" );
if( spawn )
  thisTransform.position = spawn.transform.position;
```

This little addition will look for the **PlayerSpawn** GameObject and will move Penelope there if it exists. PlayerSpawn exists in our **EmeraciteMine** Scene.

DestroyInMainScene.js

DestroyInMainScene.js destroys objects that have this script attached when they are loaded additively into a larger scene. By default, the first loaded scene is given an index of 0, so we use that to distinguish whether this scene is being loaded by itself or additively to another scene.

Create a new script named **DestroyInMainScene.js** and add the following function:

```
function Start() {
  if ( Application.loadedLevel == 0 )
    Destroy( gameObject );
}
```

Save the script and attach it to the **Plane** GameObject in each of the control setup Scenes.

ControlMenu.js

We need to return to the **EmeraciteMine** Scene and open up **ControlMenu.js**. It is time to add the menu to it. At the top of our script, add the following:

```
var background : Texture2D;
var display = false;
var font : Font;
class ControllerScene {
  var label : String;
  var controlScene : String;
}
var controllers : ControllerScene[];
```

Change the `Update` function so that the primary conditional check goes from `if(iPhoneInput.touchCount > 0)` to `if(!display && selection == -1 && iPhoneInput.touchCount > 0)`.

Also in the `Update` function, we also want to change the part were we set `guiTexture.enabled` to false and called `Launch`. Now it should be:

```
display = true;
displayBackground = false;
guiTexture.enabled = false;
```

We want to create an `OnGUI` function to show the buttons for all the separate control configurations and load the appropriate controls when one is selected.

```
function OnGUI () {
  GUI.skin.font = font;
  if ( displayBackground )
    GUI.DrawTexture( Rect( 0, 0, Screen.width, Screen.height ),
                     background, ScaleMode.StretchToFill, false );
  if ( display ) {
    var hit : int = -1;
    var minHeight = 60;
    var areaWidth = 400;
    GUILayout.BeginArea ( Rect( ( Screen.width - areaWidth ) / 2,
                         ( Screen.height - minHeight ) / 2,
                         areaWidth, minHeight ) );
    GUILayout.BeginHorizontal();
    for(var i : int = 0; i< controllers.length; i++) {
      if  (GUILayout.Button( controllers[ i ].label,
           GUILayout.MinHeight( minHeight ))) {
        hit = i;
      }
    }
    if(hit >= 0) {
      selection = hit;
```

```
      guiTexture.enabled = false;
      display = false;
      displayBackground = false;
      ChangeControls();
    }
    GUILayout.EndHorizontal();
    GUILayout.EndArea();
  }
}
```

Note that we are calling `ChangeControls`. This function name makes a whole lot more sense than "`Launch`" now that we are actually changing the control setup. In addition to renaming the function, we can expand it a bit.

```
function ChangeControls() {
  for ( var t in destroyOnLoad )
    Destroy( t.gameObject );
  launchIntro.SetActiveRecursively( true );
  yield WaitUntilObjectDestroyed( launchIntro );
  displayBackground = true;
  Application.LoadLevelAdditive( controllers[ selection ].controlScene );
  Destroy( gameObject, 1 );
}
```

In that function, we called a co-routine called `WaitUntilObjectDestroyed` to hold execution while the **LaunchIntro** was there. We need to write that function.

```
function WaitUntilObjectDestroyed( o : Object ) {
  while ( o )
    yield WaitForFixedUpdate();
}
```

You need to update some values in the Inspector for the **ControlMenu** GameObject. Set Background to **Black**, Font to **UTPenelope** and create 3 items in Controllers. For the first one, set the label to "**Tap to Move**" and the Control Scene to **TapControlSetup**. For the second, use "**Camera Relative**" and **CameraRelativeSetup**. For the third, use "**Player Relative**" and **PlayerRelativeSetup**.

Test the Scene by hitting **Play**. You can now choose a control Scene and Penelope is additively loaded into the main level.

Collecting The Orbs

The orbs that we showed to the player in the **LaunchIntro** were fakes. By the time the camera cuts back into the real scene, we will have populated it with real collectible orbs.

Putting Orbs in the Scene

theOrbs is an object that sits, disabled, in our main Scene. Once the **LaunchIntro** object has been destroyed, **ControlMenu** sets **theOrbs** active recursively. **theOrbs** contains several parti-

cle-related Components (the orbs are rendered as particles) and a script called **PickupManager.js**. It also has a number of children which are empty GameObjects. The Transform Positions of these children are used as possible spawn locations for the orbs.

Beyond this script, which we will be writing soon, **theOrbs** is just a particle system and a LOT of hand-placed empty GameObjects. To speed up the process of manually placing all of these objects, we have provided a package for you to import. Import **theOrbs.unityPackage** and drag **theOrbs** Prefab (now located in the **Prefabs** folder) into your Scene. Set the position of the object to -3, 2.2, -1.8.

Notice that **theOrbs** has a disabled script named "**Orb Gizmo**" attached to it. Activate the script to see all of the orb locations—that is why we didn't have you manually place each of them. Before we move on, deactivate **theOrbs** GameObject in the Inspector.

We need **ControlMenu.js** to enable **theOrbs** when the time is right. Open the file back up. We first need to expose a variable to reference the GameObject in the Inspector. We will call it `orbEmitter` and it will be of type `GameObject`. We want to make sure that **theOrbs** is really disabled. In our `Start` function, add `orbEmitter.SetActiveRecursively( false );` Way down in our `ChangeControls` function, after we have set `displayBackground` to true, we need to add `orbEmitter.SetActiveRecursively( true );` That should be it. Save **ControlMenu.js** and, in the Inspector, add **theOrbs** to the newly exposed variable named Orb Emitter.

PickupManager.js

Create a script called **PickupManager.js** and drag it onto **theOrbs**. This script will handle positioning the orbs. It uses the children of its GameObject as the spawn locations, randomly selects a child, and then places a particle on top of it.

Since **PickupManager.js** deals directly with the particle Components in **theOrbs**, we want to start by making sure that itonly does anything if a particle emitter Component is present. Add `@script RequireComponent( ParticleEmitter )` to the top of the script.

Most of the scripts functionality will be contained in the `Start` function. The first thing that we want to do is create a variable to reference the object's **particleEmitter**. We will call this variable `emitter`. We want to call `ClearParticles` and `Emit` on `emitter`.

In our Scene, this will emit 100 particles. Now we need to iterate over these particles and assign them locations. The idea is that for each particle we will randomly select a child object and set the particle's position to match the child's. We will then remove the child from the potential pool of locations by moving it to another GameObject that we will destroy when we are done.

Let's set a variable called `myParticles` to `emitter.particles` so that we can easily set the positions of the individual particles. We will also need to create a new GameObject to store our particles to destroy. We will call it `toDestroy`. Now, we want to access `myParticles[i]` for values of `i` between 0 and `emitter.particleCount`. We will write a `for` loop. Inside our loop, we want to first check to make sure that there are child objects and break the loop if there

aren't any. Assuming that there are child objects, we want to randomly get one of them. We will set a variable named `child` to `transform.GetChild( Random.Range( 0, transform.childCount ) )`. We now want to set `myParticles[i].position` to `child.position` and `child.parent` to `toDestroy.transform`.

After our loop, we will call `Destroy` on `toDestroy` and set `emitter.particles` to `myParticles`. At this point, you can save the script and test to see if it is working. You will, however, quickly notice that the orbs don't do anything when you pass over them. We still need to modify the script to handle picking up particles.

```
@script RequireComponent( ParticleEmitter )
function Start(){
  var emitter = particleEmitter;
  emitter.ClearParticles();
  emitter.Emit();
  var myParticles = emitter.particles;
  var toDestroy = new GameObject( "ObjectsToDestroy" );
  for(var i : int; i < emitter.particleCount; i++){
    if ( transform.childCount <= 0 )
      break;
    var child = transform.GetChild( Random.Range( 0, transform.childCount ) );
    myParticles[i].position = child.position;
    child.parent = toDestroy.transform;
  }
```

```
  Destroy( toDestroy );
  emitter.particles = myParticles;
}
```

Back in **PickupManager.js**, we want to expose a variable named `ColliderPrefab` of the type `GameObject`. This is going to reference a Prefab that we secretly added when you imported **theOrbs.unityPackage**. Before the `for` loop in our `Start` function, we want to create a variable named `ColliderContainer` as a GameObject to store all of our particle Colliders.

In the for loop, after setting `child.parent` to `toDestroy.transform`, we want to instantiate a Prefab and set some variables on a script that is attached to it. Create a variable named `Prefab` of type `GameObject` and instantiate the `ColliderPrefab` at `myParticles[i].position`. We need to access the **ParticlePickup.js** script that is attached to this Prefab. We will detail the script in a moment, but for now, we want to just create a variable named `pickup` and get the **ParticlePickup** Component from `Prefab`. Set `pickup.emitter` to `emitter` and `pickup.index` to `i`. This will let the **ParticlePickup** script reference the particle and visually remove it when it is picked up.

```
@script RequireComponent( ParticleEmitter )
var ColliderPrefab : GameObject;
function Start(){
  var emitter = particleEmitter;
  emitter.ClearParticles();
  emitter.Emit();
```

```
  var location : Vector3;
  var myParticles = emitter.particles;
  var toDestroy = new GameObject( "ObjectsToDestroy" );
  var ColliderContainer = new GameObject( "ParticleColliders" );
  for(var i : int; i < emitter.particleCount; i++){
    if ( transform.childCount <= 0 )
      break;
    var child = transform.GetChild( Random.Range( 0, transform.childCount ) );
    myParticles[i].position = child.position;
    child.parent = toDestroy.transform;
    var Prefab : GameObject = Instantiate( ColliderPrefab,  myParticles[i].position,
                              Quaternion.identity );
    var pickup : ParticlePickup = Prefab.GetComponent( ParticlePickup );
    pickup.emitter = emitter;
    pickup.index = i;
    Prefab.transform.parent = ColliderContainer.transform;
  }
  Destroy( toDestroy );
  emitter.particles = myParticles;
}
```

ParticleCollider

Ok, so we cheated a bit and already included this Prefab so that **ParticleManager.js** would just work when it was done. It is really just a box Collider with a really simple script named **Parti-**

clePickup.js. The script is in your **Scripts** directory now—let's take a look at it.

We set the `emitter` and `index` values in **PickupManager.js**. There is another variable that is set at the top of the script—`collectedParticle`. This is a GameObject that references an additional Prefab that was included with **theOrbs.unityPackage**. It is just the little visual particle system that plays when an orb is collected. There are just two functions in **ParticlePickup**—`OnTriggerEnter` and `Collected`. `OnTriggerEnter` is automatically passed to the Collider that entered the box Collider. The Collider is going to be Penelope, and this gets the **ScoreKeeper** Component on Penelope and calls `Pickup` on it. Since there is no **ScoreKeeper** component yet, these lines are initially commented out. What **ScoreKeeper** Component? All in good time.

The `Collected` function is going to be called by **ScoreKeeper**. It instantiates our **collectedParticle** and then, using `index` and `emitter` variables, accesses the particle in `emitter` and scales it down so that it is no longer visible. The last thing that `Collected` does is destroy the GameObject so that collection can't happen again.

```
var emitter : ParticleEmitter;
var index : int;
var collectedParticle : GameObject;
function OnTriggerEnter(other : Collider){
  var sk : ScoreKeeper = other.GetComponent( ScoreKeeper );
  sk.Pickup( this );
}
```

```
function Collected(){
  Instantiate( collectedParticle, transform.position, Quaternion.identity );
  var particles : Particle[] = emitter.particles;
  particles[ index ].size = 0;
  emitter.particles = particles;
  Destroy( gameObject );
}
```

Depositing the Orbs

Collecting Orbs doesn't do Penelope any good if she doesn't have anywhere to deposit them. This game is, after all, about her getting paid by her father for each orb that she collects.

The Deposit Area

Create a Cube in your Scene. We will be turning the **Mesh Renderer** off, but having it on to begin with will help us place and scale deposit area.

Rename the cube to **DepositArea** and place it in the middle of the orb collection point on the central power converter. Once you have it in place, scale it to completely cover the deposit area. In the completed project,

DepositArea is located at -4.04, 2.31, 5.37 and scaled to 2.76 x 3.73 x 2.79. The GameObject is hand placed and hand scaled, so don't worry about exactly matching those measurements. In the Inspector, check the "Is Trigger" box in **DepositArea's Box Collider** and disable the **Mesh Renderer**.

Deposit Area Graphics

There are two sets of graphics that we need to add to the deposit area. The first is a child of **DepositArea**. This GameObject is an arrow that is created to help the player find out where to take orbs when they pick them up.

- Create a new empty GameObject named **ArrowOffset** as a child of **DepositArea**.
- Set the **Transform** Position to -0.44, -0.03, 0.
- Set the Rotation to 46 around the Z axis.
- Set the Scale to 0.5 x 0.5 x 0.5.

Your **Objects** folder contains a file named **arrow**. Drag this into the Hierarchy View as a child of **ArrowOffset**.

The next graphic that you need to add is for the particle systems relating to the orb deposit area. We've included a Prefab. Drag **depositoryActivationGraphics** into your Hierarchy. This object should sit at root level—it should be a sibling to **DepositArea**, not a child.

DepositTrigger.js

Create a new script named **DepositTrigger.js**. This script will be attached to **DepositArea**. When the player enters the **DepositArea**, this script will active the particle effects for the area and deposit carried items.

We need to start by exposing an array of **ParticleEmitter** objects for the particle systems and the root GameObject for the depository. We'll also create a private boolean to flag whether or not the **arrow** is being shown.

```
var emitters : ParticleEmitter[];
var depository : GameObject;
private var arrowShown = false;
```

In our `Start` function, we will disable everything by default and will call a `DeactivateDepository` function that we wil write in a moment.

```
function Start() {
  for ( var emitter in emitters )
    emitter.emit = false;
  DeactivateDepository();
  for ( var child : Transform in transform )
    child.gameObject.SetActiveRecursively( false );
}
```

We'll go ahead and write that `DeactivateDepository` function now. We'll also include an `ActivateDepository` function for good measure.

```
function DeactivateDepository() {
  depository.SendMessage( "FadeOut" );
}

function ActivateDepository() {
  if ( !arrowShown )
    gameObject.SetActiveRecursively( true );
  depository.SendMessage( "FadeIn" );
}
```

There isn't too much more to the script—we need to create our `OnTriggerEnter` and `OnTriggerExit` functions.

In `OnTriggerEnter`, we will activate the depository objects and emitters. Next, we will tell the player that they have entered the depository through a `SendMessage` call to the other Collider. Finally, we will destroy the **arrow** the designates the depository now that we know the player has found it.

```
function OnTriggerEnter ( other : Collider ) {
  ActivateDepository();
  for ( var emitter in emitters )
```

```
    emitter.emit = true;
  other.SendMessage( "Deposit" );
  if ( !arrowShown ) {
    for ( var child : Transform in transform )
      Destroy( child.gameObject );
    arrowShown = true;
  }
}
```

In `OnTriggerExit`, we will disable the depository when the player leaves.

```
function OnTriggerExit ( other : Collider ) {
  for ( var emitter in emitters )
    emitter.emit = false;
  DeactivateDepository();
}
```

Save your file and add **DepositTrigger.js** to **DepositArea**. In the Inspector, set the size of the Emitters array to 2 and include **depositoryContacts** and **depositoryVapors**. Set Depository to **depositoryGlowMeshes**.

Keeping Score

Everything is set for Penelope to be able to collect and deposit orbs—both the collection scripts and deposit scripts have been communicating with Penelope, we just need to make a

couple additions in the control setup Scenes. Go ahead and open **CameraRelativeSetup**.

Heads Up Display

Your **Prefabs** folder contains a Prefab named **HUD**. This is what will display Penelope's score. Drag it into your **Hierarchy View**. The **Transform** Position will need to be adjusted so that it is placed in the upper left of the screen. For our **Completed Project**, we've set it at 5.047592, 0.05676067, 8.718809.

Repeat this process for the other two control setup Scenes.

ScoreKeeper.js

Create a new script named **ScoreKeeper.js**. **Scorekeeper.js** keeps track of the player's score, both the deposited and carried points. It also manages the game timer to keep track of how long until the game ends. The script keeps references to the GUI elements which display the score and time so that those GUI elements can be updated whenever the values change. We will start by declaring some variables to store this information.

```
var carrying : int;
var carryLimit : int;
var deposited : int;
var winScore : int;
var gameLength : int;
var guiMessage : GameObject;
```

We will also need to expose some `GUIText` objects so that we can assign them in the editor.

```
var carryingGui : GUIText;
var depositedGui : GUIText;
var timerGui : GUIText;
```

We also want **ScoreKeeper.js** to have sound effects and voices for different events.

```
var collectSounds : AudioClip[];
var winSound : AudioClip;
var loseSound : AudioClip;
var pickupSound : AudioClip;
var depositSound : AudioClip;

private var timeSinceLastPlay : float;
private var timeLeft : float;
```

Our `Start` function will set things up at the beginning of the game.

```
public function Start() {
  timeLeft = gameLength;
  timeSinceLastPlay = Time.time;
  UpdateCarryingGui();
  UpdateDepositedGui();
  CheckTime();
```

```
}
```

Note that we called `UpdateCarryingGui` and `UpdateDepositedGui`. We want to write some basic functions that can be called to update the text for our various **HUD** objects.

```
function UpdateCarryingGui() {
  carryingGui.text = "Carrying: " + carrying + " of " + carryLimit;
}

function UpdateDepositedGui() {
  depositedGui.text = "Deposited: " + deposited + " of " + winScore;
}

function UpdateTimerGui() {
  timerGui.text = "Time: " + TimeRemaining();
}
```

The last line of our `Start` function called a function named `CheckTime`. `CheckTime` is a co-routine that controls the timer. Since we only need to check the timer once every second, doing this in an `Update` function would be wasteful.

```
private function CheckTime() {
  while ( timeLeft > 0 ) {
    UpdateTimerGui();
```

```
    yield WaitForSeconds(1);
    timeLeft -= 1;
  }
  UpdateTimerGui();
  EndGame();
}
```

We are going to include a utility function to play one shot audio at a specific position and at a specific volume. This will be used by several functions in this script.

```
function PlayAudioClip( clip : AudioClip, position : Vector3, volume : float ) {
  var go = new GameObject( "One shot audio" );
  go.transform.position = position;
  var source : AudioSource = go.AddComponent( AudioSource );
  source.rolloffFactor = 0;
  source.clip = clip;
  source.volume = volume;
  source.Play();
  Destroy( go, clip.length );
  return source;
}
```

Our next task is to write our `EndGame` function. This should handle sending messages to other Components when the game ends. It should also reload the intro level when the player touches the device.

```
private function EndGame() {
  var animationController : AnimationController = GetComponent( AnimationController );
  var Prefab : GameObject = Instantiate(guiMessage);
  var endMessage : GUIText = Prefab.GetComponent( GUIText );
  if(deposited >= winScore) {
    endMessage.text = "You win!";
    PlayAudioClip( winSound, Vector3.zero, 1.0 );
    animationController.animationTarget.Play( "WIN" );
  } else {
      endMessage.text = "Oh no...You lose!";
      PlayAudioClip( loseSound, Vector3.zero, 1.0 );
      animationController.animationTarget.Play( "LOSE" );
  }
  SendMessage( "OnEndGame" );
  while( true ) {
    yield WaitForFixedUpdate();
    if  ( iPhoneInput.touchCount > 0 && iPhoneInput.GetTouch( 0 ).phase ==
        iPhoneTouchPhase.Began )
      break;
  }
  Application.LoadLevel( 0 );
}
```

We need to write our `Pickup` and `Deposit` functions. If you recall, we referenced the `Pickup`

function when we were writing our particle pickup script. This function also sends a message to activate the depository once an orb has been picked up.

```
public function Pickup( pickup : ParticlePickup ) {
  if ( carrying < carryLimit ) {
    carrying++;
    UpdateCarryingGui();
    var minTimeBetweenPlays = 5;
    if  ( Random.value < 0.1 && Time.time > ( minTimeBetweenPlays + timeSinceLastPlay ) ){
      PlayAudioClip ( collectSounds[ Random.Range( 0, collectSounds.length )],
                    Vector3.zero, 0.25 );
      timeSinceLastPlay = Time.time;
    }
    pickup.Collected();
    PlayAudioClip( pickupSound, pickup.transform.position, 1.0 );
  } else {
    var warning : GameObject = Instantiate( guiMessage );
    warning.guiText.text = "You can't carry any more";
    Destroy(warning, 2);
  }
  if ( carrying >= carryLimit )
    pickup.emitter.SendMessage( "ActivateDepository" );
}
```

The `Deposit` function is simple. We update totals, update our GUI and play a sound.

```
public function Deposit() {
  deposited += carrying;
  carrying = 0;
  UpdateCarryingGui();
  UpdateDepositedGui();
  PlayAudioClip( depositSound, transform.position, 1.0 );
}
```

The last thing we need to do in this script is to create the `TimeRemaining` function called in `UpdateTimerGui`. This function gets the time left for gameplay and formats it nicely.

```
public function TimeRemaining() : String {
  var remaining : int = timeLeft;
  var val : String;
  if(remaining > 59)
    val+= remaining / 60 + ".";
  if(remaining >= 0) {
    var seconds : String = (remaining % 60).ToString();
    if(seconds.length  < 2)
      val += "0" + seconds;
    else
      val += seconds;
```

```
  }
  return val;
}
```

- Save the script and add it to your **Player** GameObject.
- In the Inspector, set Gui Message to **CenteredGUIText**.
- Set Carrying Gui to **CarryingDisplay (GUIText)**
- Set Deposited Gui to **DepositedDisplay (GUIText)**
- Set Timer Gui to **TimerDisplay (GUIText)**
- Set the length of Collect Sounds to 2 and add **vox-allright** and **vox-gotit**
- Set Win Sound to **vox-iamawesome**
- Set Lose Sound to **vox-thatsucks**
- Set Pickup Sound to **sfx-pickup**
- Set Deposit Sound to **sfx-deposit**

Carrying and Deposited should be set to 0 to begin with, but the Carry Limit, Win Score and Game Length are values that you will probably want to tweak. We set them differently for each control scheme in the **Completed Project.**

ScoreKeeper.js needs to be added to the **Player** objects in each of the control setup Scenes. The last thing that you need to do is uncomment the **ScoreKeeper** Component references in **ParticlePickup.js**.

What You Learned

In the final part of this tutorial, you learned how to bring your control setup Scenes into the primary game level. You also learned how to build a system for pseudo-randomly placed items and how to handle collection and scoring. In the four parts of this tutorial, you have created a game for the iPhone. Congratulations.

Suggested Improvements

This tutorial might be over, but your time with Penelope doesn't have to be. Here are some suggestions to get you started extending Penelope into a full-fledged game.

Enemies

Does the power of the Emeracite orbs lure strange and exotic beasts who would like to consume them (and Penelope!)? Adding enemies to the game would not only expand the gameplay, but it would give you the opportunity to implement some sort of AI.

Networked High-Score

Who collected the most orbs? A potential upgrade to Penelope could include a high score system. This would give you a chance to perfect submitting scores to a web-based database and would give your players a chance to gloat.

More Levels

One level? Really? Perhaps Penelope's orb collection skills place her in high demand at other mine locations. Or maybe someone has been stealing orbs (see Enemies, above) and she has to track

them down. In any case, to make a complete game out of this demo, more levels are in order.

Further Reading

The first place to look for more information is, as always, Unity's own documentation. There are also many tutorials (including video primers) on the Unity website:

http://unity3d.com/support/documentation/

Script Appendix

Joystick.js

```
@script RequireComponent( GUITexture )

class Boundary {

  var min : Vector2 = Vector2.zero;

  var max : Vector2 = Vector2.zero;

}


static private var joysticks : Joystick[];

static private var enumeratedJoysticks : boolean = false;

static private var tapTimeDelta : float = 0.3;

public var position : Vector2;

public var deadZone : Vector2 = Vector2.zero;

public var normalize : boolean = false;

public var tapCount : int;

private var lastFingerId = -1;

private var tapTimeWindow : float;

private var gui : GUITexture;

private var defaultRect : Rect;

private var guiBoundary : Boundary = Boundary();

private var guiTouchOffset : Vector2;

private var guiCenter : Vector2;
```

```
function Start() {
  gui = GetComponent( GUITexture );
  defaultRect = gui.pixelInset;
  guiTouchOffset.x = defaultRect.width * 0.5;
  guiTouchOffset.y = defaultRect.height * 0.5;
  guiCenter.x = defaultRect.x + guiTouchOffset.x;
  guiCenter.y = defaultRect.y + guiTouchOffset.y;
  guiBoundary.min.x = defaultRect.x - guiTouchOffset.x;
  guiBoundary.max.x = defaultRect.x + guiTouchOffset.x;
  guiBoundary.min.y = defaultRect.y - guiTouchOffset.y;
  guiBoundary.max.y = defaultRect.y + guiTouchOffset.y;
  enumeratedJoysticks = false;
}

function Disable() {
  gameObject.active = false;
  enumeratedJoysticks = false;
}

function Reset() {
  gui.pixelInset = defaultRect;
  lastFingerId = -1;
}
```

```
function LatchedFinger( fingerId : int ) {
  if ( lastFingerId == fingerId )
    Reset();
}

function Update() {
  if ( !enumeratedJoysticks ) {
    joysticks = FindObjectsOfType( Joystick );
    enumeratedJoysticks = true;
  }
  var count = iPhoneInput.touchCount;
  if ( tapTimeWindow > 0 )
    tapTimeWindow -= Time.deltaTime;
  else
    tapCount = 0;
  if ( count == 0 )
    Reset();
  else {
    for(var i : int = 0;i < count; i++) {
      var touch : iPhoneTouch = iPhoneInput.GetTouch(i);
      var guiTouchPos : Vector2 = touch.position - guiTouchOffset;
      if ( gui.HitTest( touch.position ) && ( lastFingerId == -1 ||
         lastFingerId != touch.fingerId ) ) {
        lastFingerId = touch.fingerId;
```

```
        if ( tapTimeWindow > 0 )
          tapCount++;
        else {
          tapCount = 1;
          tapTimeWindow = tapTimeDelta;
        }
        for ( var j : Joystick in joysticks ) {
          if ( j != this )
            j.LatchedFinger( touch.fingerId );
        }
      }
      if ( lastFingerId == touch.fingerId ) {
        if ( touch.tapCount > tapCount )
          tapCount = touch.tapCount;
        gui.pixelInset.x = Mathf.Clamp(guiTouchPos.x, guiBoundary.min.x, guiBoundary.max.x );
        gui.pixelInset.y = Mathf.Clamp(guiTouchPos.y, guiBoundary.min.y, guiBoundary.max.y );
        if ( touch.phase == iPhoneTouchPhase.Ended ||
           touch.phase == iPhoneTouchPhase.Canceled )
          Reset();
      }
    }
}
```

```
    position.x = ( gui.pixelInset.x + guiTouchOffset.x - guiCenter.x ) / guiTouchOffset.x;
    position.y = ( gui.pixelInset.y + guiTouchOffset.y - guiCenter.y ) / guiTouchOffset.y;
    var absoluteX = Mathf.Abs( position.x );
    var absoluteY = Mathf.Abs( position.y );
    if ( absoluteX < deadZone.x ) {
      position.x = 0;
    } else if ( normalize ) {
      position.x = Mathf.Sign( position.x ) * ( absoluteX - deadZone.x ) / ( 1 - deadZone.x );
    }
    if ( absoluteY < deadZone.y ) {
      position.y = 0;
    }
    else if ( normalize ) {
      position.y = Mathf.Sign( position.y ) * ( absoluteY - deadZone.y ) / ( 1 - deadZone.y );
    }
  }
```

PlayerRelativeControl.js

```
@script RequireComponent( CharacterController )

var moveJoystick : Joystick;

var rotateJoystick : Joystick;

var cameraPivot : Transform;

var forwardSpeed : float = 6;

var backwardSpeed : float = 3;

var sidestepSpeed : float = 4;

var jumpSpeed : float = 16;

var inAirMultiplier : float = 0.25;

var rotationSpeed : Vector2 = Vector2( 50, 25 );

private var thisTransform : Transform;

private var character : CharacterController;

private var animationController : AnimationController;

private var cameraVelocity : Vector3;

private var velocity : Vector3;


function Start() {

  thisTransform = GetComponent( Transform );

  character = GetComponent( CharacterController );

  animationController = GetComponent( AnimationController );

  animationController.maxForwardSpeed = forwardSpeed;

  animationController.maxBackwardSpeed = backwardSpeed;

  animationController.maxSidestepSpeed = sidestepSpeed;
```

```
  var spawn = GameObject.Find( "PlayerSpawn" );
  if ( spawn )
    thisTransform.position = spawn.transform.position;
}

function OnEndGame() {
  moveJoystick.Disable();
  rotateJoystick.Disable();
  this.enabled = false;
}

function Update() {
  var movement = thisTransform.TransformDirection(
                Vector3( moveJoystick.position.x, 0, moveJoystick.position.y ) );
  movement.y = 0;
  movement.Normalize();
  var cameraTarget = Vector3.zero;
  var absJoyPos = Vector2( Mathf.Abs( moveJoystick.position.x ),
                 Mathf.Abs( moveJoystick.position.y ) );
  if ( absJoyPos.y > absJoyPos.x ) {
    if ( moveJoystick.position.y > 0 )
      movement *= forwardSpeed * absJoyPos.y;
    else {
      movement *= backwardSpeed * absJoyPos.y;
```

```
    cameraTarget.z = moveJoystick.position.y * 0.75;
  }
} else {
  movement *= sidestepSpeed * absJoyPos.x;
  cameraTarget.x = -moveJoystick.position.x * 0.5;
}
if ( character.isGrounded ) {
  if ( rotateJoystick.tapCount == 2 ) {
    velocity = character.velocity;
    velocity.y = jumpSpeed;
  }
} else {
  velocity.y += Physics.gravity.y * Time.deltaTime;
  cameraTarget.z = -jumpSpeed * 0.25;
  movement.x *= inAirMultiplier;
  movement.z *= inAirMultiplier;
}
movement += velocity;
movement += Physics.gravity;
movement *= Time.deltaTime;
character.Move( movement );
if ( character.isGrounded )
  velocity = Vector3.zero;
var pos = cameraPivot.localPosition;
```

```
    pos.x = Mathf.SmoothDamp( pos.x, cameraTarget.x, cameraVelocity.x, 0.3 );
    pos.z = Mathf.SmoothDamp( pos.z, cameraTarget.z, cameraVelocity.z, 0.5 );
    cameraPivot.localPosition = pos;
    if ( character.isGrounded ) {
      var camRotation = rotateJoystick.position;
      camRotation.x *= rotationSpeed.x;
      camRotation.y *= rotationSpeed.y;
      camRotation *= Time.deltaTime;
      thisTransform.Rotate( 0, camRotation.x, 0, Space.World );
      cameraPivot.Rotate( camRotation.y, 0, 0 );
    }
  }
```

CameraRelativeControl.js

```
@script RequireComponent( CharacterController )

var moveJoystick : Joystick;

var rotateJoystick : Joystick;

var cameraPivot : Transform;

var cameraTransform : Transform;

var speed : float = 5;

var jumpSpeed : float = 16;

var inAirMultiplier : float = 0.25;

var rotationSpeed : Vector2 = Vector2( 50, 25 );

private var thisTransform : Transform;

private var character : CharacterController;

private var animationController : AnimationController;

private var velocity : Vector3;


function Start() {

  thisTransform = GetComponent( Transform );

  character = GetComponent( CharacterController );

  animationController = GetComponent( AnimationController );

  animationController.maxForwardSpeed = speed;

  var spawn = GameObject.Find( “PlayerSpawn” );

  if ( spawn )

    thisTransform.position = spawn.transform.position;

}
```

```
function FaceMovementDirection() {
  var horizontalVelocity : Vector3 = character.velocity;
  horizontalVelocity.y = 0;
  if ( horizontalVelocity.magnitude > 0.1 )
    thisTransform.forward = horizontalVelocity.normalized;
}

function OnEndGame() {
  moveJoystick.Disable();
  rotateJoystick.Disable();
  this.enabled = false;
}

function Update() {
  var movement = cameraTransform.TransformDirection(
                Vector3( moveJoystick.position.x, 0, moveJoystick.position.y ) );
  movement.y = 0;
  movement.Normalize();
  var absJoyPos = Vector2( Mathf.Abs( moveJoystick.position.x ),
                 Mathf.Abs( moveJoystick.position.y ) );
  movement *= speed * ( ( absJoyPos.x > absJoyPos.y ) ? absJoyPos.x : absJoyPos.y );
  if ( character.isGrounded ) {
    if ( rotateJoystick.tapCount == 2 ) {
      velocity = character.velocity;
```

```
        velocity.y = jumpSpeed;
      }
    } else  {
      velocity.y += Physics.gravity.y * Time.deltaTime;
      movement.x *= inAirMultiplier;
      movement.z *= inAirMultiplier;
    }
    movement += velocity;
    movement += Physics.gravity;
    movement *= Time.deltaTime;
    character.Move( movement );
    if ( character.isGrounded )
      velocity = Vector3.zero;
    FaceMovementDirection();
    var camRotation = rotateJoystick.position;
    camRotation.x *= rotationSpeed.x;
    camRotation.y *= rotationSpeed.y;
    camRotation *= Time.deltaTime;
    cameraPivot.Rotate( 0, camRotation.x, 0, Space.World );
    cameraPivot.Rotate( camRotation.y, 0, 0 );
  }
```

TapControl.js

```
enum ControlState {
  WaitingForFirstTouch,
  WaitingForSecondTouch,
  MovingCharacter,
  WaitingForMovement,
  ZoomingCamera,
  RotatingCamera,
  WaitingForNoFingers
}
var cameraObject : GameObject;
var cameraPivot : Transform;
var jumpButton : GUITexture;
var speed : float;
var jumpSpeed : float;
var inAirMultiplier : float = 0.25;
var minimumDistanceToMove = 1.0;
var minimumTimeUntilMove = 0.25;
var zoomEnabled : boolean;
var zoomEpsilon : float;
var zoomRate : float;
var rotateEnabled : boolean;
var rotateEpsilon : float = 1;
private var zoomCamera : ZoomCamera;
```

```
private var cam : Camera;
private var thisTransform : Transform;
private var character : CharacterController;
private var animationController : AnimationController;
private var targetLocation : Vector3;
private var moving : boolean = false;
private var rotationTarget : float;
private var rotationVelocity : float;
private var velocity : Vector3;
private var state = ControlState.WaitingForFirstTouch;
private var fingerDown : int[] = new int[ 2 ];
private var fingerDownPosition : Vector2[] = new Vector2[ 2 ];
private var fingerDownFrame : int[] = new int[ 2 ];
private var firstTouchTime : float;

function Start() {
  thisTransform = transform;
  zoomCamera = cameraObject.GetComponent( ZoomCamera );
  cam = cameraObject.camera;
  character = GetComponent( CharacterController );
  animationController = GetComponent( AnimationController );
  animationController.maxForwardSpeed = speed;
  ResetControlState();
  var spawn = GameObject.Find( "PlayerSpawn" );
```

```
  if( spawn )
    thisTransform.position = spawn.transform.position;
}

function OnEndGame() {
  this.enabled = false;
}

function FaceMovementDirection() {
  var horizontalVelocity : Vector3 = character.velocity;
  horizontalVelocity.y = 0;
  if( horizontalVelocity.magnitude > 0.1 )
    thisTransform.forward = horizontalVelocity.normalized;
}

function CameraControl( touch0 : iPhoneTouch, touch1 : iPhoneTouch ) {
  if( rotateEnabled && state == ControlState.RotatingCamera ) {
    var currentVector : Vector2 = touch1.position - touch0.position;
    var currentDir = currentVector / currentVector.magnitude;
    var lastVector : Vector2 = ( touch1.position - touch1.deltaPosition ) -
                               ( touch0.position - touch0.deltaPosition );
    var lastDir = lastVector / lastVector.magnitude;
    var rotationCos : float = Vector2.Dot( currentDir, lastDir );
    if ( rotationCos < 1 ) {
```

```
      var currentVector3 : Vector3 = Vector3( currentVector.x, currentVector.y );
      var lastVector3 : Vector3 = Vector3( lastVector.x, lastVector.y );
      var rotationDirection : float = Vector3.Cross( currentVector3,
                                                     lastVector3 ).normalized.z;
      var rotationRad = Mathf.Acos( rotationCos );
      rotationTarget += rotationRad * Mathf.Rad2Deg * rotationDirection;
      if ( rotationTarget < 0 )
        rotationTarget += 360;
      else if ( rotationTarget >= 360 )
        rotationTarget -= 360;
    }
  }
  else if( zoomEnabled && state == ControlState.ZoomingCamera ) {
    var touchDistance = ( touch1.position - touch0.position ).magnitude;
    var lastTouchDistance = ( ( touch1.position - touch1.deltaPosition ) -
                              ( touch0.position - touch0.deltaPosition ) ).magnitude;
    var deltaPinch = touchDistance - lastTouchDistance;
    zoomCamera.zoom += deltaPinch * zoomRate * Time.deltaTime;
  }
}

function CharacterControl() {
  var count : int = iPhoneInput.touchCount;
  if( count == 1 && state == ControlState.MovingCharacter ) {
```

```
    var touch : iPhoneTouch = iPhoneInput.GetTouch(0);
    if ( character.isGrounded && jumpButton.HitTest( touch.position ) ) {
      velocity = character.velocity;
      velocity.y = jumpSpeed;
    } else if ( !jumpButton.HitTest( touch.position ) &&
              touch.phase != iPhoneTouchPhase.Began ) {
      var ray = cam.ScreenPointToRay( Vector3( touch.position.x, touch.position.y ) );
      var hit : RaycastHit;
      if( Physics.Raycast(ray, hit) ) {
        var touchDist : float = (transform.position - hit.point).magnitude;
        if( touchDist > minimumDistanceToMove ) {
          targetLocation = hit.point;
        }
        moving = true;
      }
    }
  }

var movement : Vector3 = Vector3.zero;
if( moving ){
  movement = targetLocation - thisTransform.position;
  movement.y=0;
  var dist : float = movement.magnitude;
  if( dist < 1 ) {
```

```
      moving = false;
    } else {
      movement = movement.normalized * speed;
    }
  }
  if ( !character.isGrounded ){
    velocity.y += Physics.gravity.y * Time.deltaTime;
    movement.x *= inAirMultiplier;
    movement.z *= inAirMultiplier;
  }
  movement += velocity;
  movement += Physics.gravity;
  movement *= Time.deltaTime;
  character.Move( movement );
  if ( character.isGrounded )
    velocity = Vector3.zero;
  FaceMovementDirection();
}

function ResetControlState() {
  state = ControlState.WaitingForFirstTouch;
  fingerDown[ 0 ] = -1;
  fingerDown[ 1 ] = -1;
}
```

```
function Update () {
  var touchCount : int = iPhoneInput.touchCount;
  if ( touchCount == 0 ) {
    ResetControlState();
  } else {
    var i : int;
    var touch : iPhoneTouch;
    var touches = iPhoneInput.touches;
    var touch0 : iPhoneTouch;
    var touch1 : iPhoneTouch;
    var gotTouch0 = false;
    var gotTouch1 = false;
    if ( state == ControlState.WaitingForFirstTouch ) {
      for ( i = 0; i < touchCount; i++ ) {
        touch = touches[ i ];
        if ( touch.phase != iPhoneTouchPhase.Ended
           && touch.phase != iPhoneTouchPhase.Canceled ) {
          state = ControlState.WaitingForSecondTouch;
          firstTouchTime = Time.time;
          fingerDown[ 0 ] = touch.fingerId;
          fingerDownPosition[ 0 ] = touch.position;
          fingerDownFrame[ 0 ] = Time.frameCount;
          break;
        }
```

```
      }
    }
  if ( state == ControlState.WaitingForSecondTouch ) {
    for ( i = 0; i < touchCount; i++ ) {
      touch = touches[ i ];
      if ( touch.phase != iPhoneTouchPhase.Canceled ) {
        if ( touchCount >= 2 && touch.fingerId != fingerDown[ 0 ] ) {
          state = ControlState.WaitingForMovement;
          fingerDown[ 1 ] = touch.fingerId;
          fingerDownPosition[ 1 ] = touch.position;
          fingerDownFrame[ 1 ] = Time.frameCount;
          break;
        } else if ( touchCount == 1 ) {
          var deltaSinceDown = touch.position - fingerDownPosition[ 0 ];
          if ( touch.fingerId == fingerDown[ 0 ] &&
             ( Time.time > firstTouchTime + minimumTimeUntilMove
             || touch.phase == iPhoneTouchPhase.Ended ) ) {
            state = ControlState.MovingCharacter;
            break;
          }
        }
      }
    }
```

```
if ( state == ControlState.WaitingForMovement ){
  for ( i = 0; i < touchCount; i++ ) {
    touch = touches[ i ];
    if ( touch.phase == iPhoneTouchPhase.Began ) {
      if ( touch.fingerId == fingerDown[ 0 ] && fingerDownFrame[ 0 ] == Time.frameCount ) {
        touch0 = touch;
        gotTouch0 = true;
      } else if ( touch.fingerId != fingerDown[ 0 ] && touch.fingerId != fingerDown[ 1 ] ) {
        fingerDown[ 1 ] = touch.fingerId;
        touch1 = touch;
        gotTouch1 = true;
      }
    }
    if ( touch.phase == iPhoneTouchPhase.Moved
       || touch.phase == iPhoneTouchPhase.Stationary
       || touch.phase == iPhoneTouchPhase.Ended ){
      if ( touch.fingerId == fingerDown[ 0 ] ) {
        touch0 = touch;
        gotTouch0 = true;
      } else if ( touch.fingerId == fingerDown[ 1 ] ) {
        touch1 = touch;
        gotTouch1 = true;
      }
    }
```

```
}
if ( gotTouch0 ) {
  if ( gotTouch1 ) {
    var originalVector = fingerDownPosition[ 1 ] - fingerDownPosition[ 0 ];
    var currentVector = touch1.position - touch0.position;
    var originalDir = originalVector / originalVector.magnitude;
    var currentDir = currentVector / currentVector.magnitude;
    var rotationCos : float = Vector2.Dot( originalDir, currentDir );
    if ( rotationCos < 1 ) {
      var rotationRad = Mathf.Acos( rotationCos );
      if ( rotationRad > rotateEpsilon * Mathf.Deg2Rad ) {
        state = ControlState.RotatingCamera;
      }
    }
    if ( state == ControlState.WaitingForMovement ) {
      var deltaDistance = originalVector.magnitude - currentVector.magnitude;
      if ( Mathf.Abs( deltaDistance ) > zoomEpsilon ){
        state = ControlState.ZoomingCamera;
      }
    }
  }
} else {
  state = ControlState.WaitingForNoFingers;
}
```

```
}

if ( state == ControlState.RotatingCamera || state == ControlState.ZoomingCamera ) {
  for ( i = 0; i < touchCount; i++ ) {
    touch = touches[ i ];
    if ( touch.phase == iPhoneTouchPhase.Moved
       || touch.phase == iPhoneTouchPhase.Stationary
       || touch.phase == iPhoneTouchPhase.Ended ) {
      if ( touch.fingerId == fingerDown[ 0 ] ) {
        touch0 = touch;
        gotTouch0 = true;
      } else if ( touch.fingerId == fingerDown[ 1 ] ) {
        touch1 = touch;
        gotTouch1 = true;
      }
    }
  }
  if ( gotTouch0 ) {
    if ( gotTouch1 ) {
       CameraControl( touch0, touch1 );
    }
  } else {
    state = ControlState.WaitingForNoFingers;
  }
```

```
    }
  }
   CharacterControl();
}

function LateUpdate() {
   cameraPivot.eulerAngles.y = Mathf.SmoothDampAngle( cameraPivot.eulerAngles.y,
                               rotationTarget, rotationVelocity, 0.3 );
}
```

AnimationController.js

```
var animationTarget : Animation;

var maxForwardSpeed : float = 6;

var maxBackwardSpeed : float = 3;

var maxSidestepSpeed : float = 4;

private var character : CharacterController;

private var thisTransform : Transform;

private var jumping : boolean = false;

private var minUpwardSpeed = 2;


function Start() {

  character = GetComponent( CharacterController );

  thisTransform = transform;

  animationTarget.wrapMode = WrapMode.Loop;

  animationTarget[ “jump” ].wrapMode = WrapMode.ClampForever;

  animationTarget[ “jump-land” ].wrapMode = WrapMode.ClampForever;

  animationTarget[ “run-land” ].wrapMode = WrapMode.ClampForever;

  animationTarget[ “LOSE” ].wrapMode = WrapMode.ClampForever;

}


function OnEndGame() {

  this.enabled = false;

}
```

```
function Update() {
  var characterVelocity = character.velocity;
  var horizontalVelocity : Vector3 = characterVelocity;
  horizontalVelocity.y = 0;
  var speed = horizontalVelocity.magnitude;
  var upwardsMotion = Vector3.Dot( thisTransform.up, characterVelocity );
  if ( !character.isGrounded && upwardsMotion > minUpwardSpeed )
    jumping = true;
  if ( animationTarget.IsPlaying( “run-land” )
    && animationTarget[ “run-land” ].normalizedTime < 1.0
    && speed > 0 ) {
  } else if ( animationTarget.IsPlaying( “jump-land” ) ) {
    if ( animationTarget[ “jump-land” ].normalizedTime >= 1.0 )
      animationTarget.Play( “idle” );
  } else if ( jumping ) {
    if ( character.isGrounded ) {
      if ( speed > 0 )
        animationTarget.Play( “run-land” );
      else
        animationTarget.Play( “jump-land” );
      jumping = false;
    } else
      animationTarget.Play( “jump” );
  } else if ( speed > 0 ) {
```

```
var forwardMotion = Vector3.Dot( thisTransform.forward, horizontalVelocity );
var sidewaysMotion = Vector3.Dot( thisTransform.right, horizontalVelocity );
var t = 0.0;
if ( Mathf.Abs( forwardMotion ) > Mathf.Abs( sidewaysMotion ) ) {
  if ( forwardMotion > 0 ) {
    t = Mathf.Clamp( Mathf.Abs( speed / maxForwardSpeed ), 0, maxForwardSpeed );
    animationTarget[ "run" ].speed = Mathf.Lerp( 0.25, 1, t );
    if ( animationTarget.IsPlaying( "run-land" ) || animationTarget.IsPlaying( "idle" ) )
      animationTarget.Play( "run" );
    else
      animationTarget.CrossFade( "run" );
  } else {
    t = Mathf.Clamp( Mathf.Abs( speed / maxBackwardSpeed ), 0, maxBackwardSpeed );
    animationTarget[ "runback" ].speed = Mathf.Lerp( 0.25, 1, t );
    animationTarget.CrossFade( "runback" );
  }
} else {
  t = Mathf.Clamp( Mathf.Abs( speed / maxSidestepSpeed ), 0, maxSidestepSpeed );
  if ( sidewaysMotion > 0 ) {
    animationTarget[ "runright" ].speed = Mathf.Lerp( 0.25, 1, t );
    animationTarget.CrossFade( "runright" );
  } else {
    animationTarget[ "runleft" ].speed = Mathf.Lerp( 0.25, 1, t );
    animationTarget.CrossFade( "runleft" );
```

```
        }
      }
    } else
      animationTarget.CrossFade( “idle” );
}
```

ControlMenu.js

```
var background : Texture2D;

var display = false;

var font : Font;


class ControllerScene {

  var label : String;

  var controlScene : String;

}


var controllers : ControllerScene[];

var destroyOnLoad : Transform[];

var launchIntro : GameObject;

var orbEmitter : GameObject;

private var selection = -1;

private var displayBackground = false;


function Start() {

  launchIntro.SetActiveRecursively( false );

  orbEmitter.SetActiveRecursively( false );

}


function Update () {

  if ( !display && selection == -1 &&  iPhoneInput.touchCount > 0 ) {
```

```
    for(var i : int = 0; i< iPhoneInput.touchCount;i++) {
      var touch : iPhoneTouch = iPhoneInput.GetTouch(i);
      if(touch.phase == iPhoneTouchPhase.Began && guiTexture.HitTest(touch.position)) {
        display = true;
        displayBackground = false;
        guiTexture.enabled = false;
      }
    }
  }
}

function OnGUI () {
  GUI.skin.font = font;
  if ( displayBackground )
    GUI.DrawTexture( Rect( 0, 0, Screen.width, Screen.height ), background,
                     ScaleMode.StretchToFill, false );
  if ( display ) {
    var hit : int = -1;
    var minHeight = 60;
    var areaWidth = 400;
    GUILayout.BeginArea( Rect( ( Screen.width - areaWidth ) / 2,
                       ( Screen.height - minHeight ) / 2, areaWidth, minHeight ) );
    GUILayout.BeginHorizontal();
```

```
    for(var i : int = 0; i< controllers.length; i++) {
      if(GUILayout.Button( controllers[ i ].label, GUILayout.MinHeight( minHeight ))) {
        hit = i;
      }
    }
    if(hit >= 0) {
      selection = hit;
      guiTexture.enabled = false;
      display = false;
      displayBackground = false;
      ChangeControls();
    }
    GUILayout.EndHorizontal();
    GUILayout.EndArea();
  }
}

function WaitUntilObjectDestroyed( o : Object ) {
  while ( o )
    yield WaitForFixedUpdate();
}

function ChangeControls() {
  for ( var t in destroyOnLoad )
```

```
    Destroy( t.gameObject );
  launchIntro.SetActiveRecursively( true );
  yield WaitUntilObjectDestroyed( launchIntro );
  displayBackground = true;
  orbEmitter.SetActiveRecursively( true );
  Application.LoadLevelAdditive( controllers[ selection ].controlScene );
  Destroy( gameObject, 1 );
}
```