

3D Platformer Tutorial

Building a 3D Platform Game in Unity 2.0

Contents

1. Introduction	
What you will learn	5
Project Organization	6
Files	7
Typographical Conventions	7
Backgrounders & Tangents	7
Unity Conventions	8
GameObjects, Components, Assets & Prefabs	8
Custom Icons & Gizmos	9
Acknowledgments	10
2. First Steps	
Animating Lerpz	11
The Plot	11
Introducing Lerpz	12
Third Person Cameras.	13
Making Changes While Playing	15
Connections & Dependencies.	16
The Character Controller & the Third Person Controller script.	17
Animating Lerpz.	18
Skeletons & Armatures	18
Gizmos	20
The Jet-Pack	20
What is a Particle System?	21
Why no shadows?	25
Blob Shadows	27
Why not adjust the Near Clip Plane?	29
Scripting Concepts	31
Organization & Structure.	32
Scripts in a Visual Development Environment	33
Death & Rebirth	33
Respawn Points	35
3. Setting the Scene	
First Steps	39

Placing Props	39
Building Your Own Levels	39
Pretty Properties	43
Scripting the Collectable Items	43
Triggers	44
How much health?	44
Jump Pads	45
4. The GUI	
The User Interface	47
Unity 2.0's new GUI system	47
The In-game HUD	48
Resolution Independence.	52
The Start Menu	53
Setting the Scene	53
The Quit Button.	59
Game Over	59
5. Adversaries	
Antagonists & Conflict	65
The Laser Traps	65
The Line Renderer Component	68
The Laser Trap Script	68
Shock & Awe	69
The Robot Guards	70
Droppable Pickups & Physics	74
Spawning & Optimization	74
Alternative Optimizations	77
6. Audio & Finishing Touches	
Introduction	78
Audio	78
Adding Sound to Lerpz Escapes!	79
Cut Scenes	89
7. Optimizing	
Why Optimize?	103
Optimizing Rendering: Monitoring Frames Per Second.	103
Optimizing Rendering: The Two-Camera System.	105
8. End of the road.	
The Road Less Travelled	107
Suggested Improvements	107
Further Reading	108

Introduction

Unity is a powerful tool for game development, suitable for many game genres, from first-person shooters through to puzzle games.



With its myriad features, including height-mapped terrains, native networking support, complete physics integration and scripting, Unity can be daunting for newcomers, but mastering its many tools is infinitely rewarding.

This tutorial will walk you through the process of building a complete 3D platform game level with a third-person perspective view. This includes everything from player controls, collision detection, some advanced scripting, blob shadows, basic AI, adding a game HUD, cut-scenes and audio spot effects.

What you will learn

This tutorial focuses on the technical side of building a game in Unity, covering the following:

- Character Controllers
- Projectors
- Audio Listeners, Audio Sources & Audio Clips
- Multiple Cameras (and how to switch between them)
- UnityGUI scripting system
- Colliders
- Messages & events

- Lighting
- Particle systems
- Blob shadows
- Scripting (AI, state machines, player controls)

This tutorial will show how these features can be used together to create a game.

What you should already know

This tutorial makes extensive use of scripting so you should be familiar with at least one of the supported scripting languages: JavaScript, C# or Boo. (JavaScript is used for the scripts in this tutorial.)

It is also assumed 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.

Project Organization

Unity does not attempt to force a particular way of organizing your project's assets. You may prefer organizing your assets by asset type, with separate folders for, say, "Textures", "Models", "Sound effects" and so on. At Unity Technologies, we have found this works well for smaller projects. For more complex projects, our users generally recommend organizing assets by function, perhaps grouping them under folders such as "Player", "Enemies", "Props", "Scenery", and so on.

This tutorial's project was worked on by a number of team members and grew organically to reflect their different conventions and styles. In the interests of authenticity, we have decided to leave the project's organization as it was as this is more representative of a 'smaller' project's organization and structure.

Abstract GameObjects & Components

Unity's design places each scene's assets at the center of the development process. This makes for a very visual approach to game development, with most of the work involving dragging and dropping. This is ideal for the bulk of level design work, but not all assets can be displayed in this way. Some assets are abstract rather than visual objects, so they are either represented by abstract icons and wireframe gizmos -- e.g. Audio Sources and Lights -- or are not displayed at all within the Scene View. Scripts fall into this latter category.

Scripts define how assets and GameObjects in a Unity Scene interact with each other and this interactivity is at the core of all games. For this reason, it is usually a good plan to keep informative notes inside your scripts.

This tutorial will assume you can read the provided scripts and understand the many comments liberally sprinkled throughout them. However, when a particular scripting technique or concept is important, we will cover it in detail.

To help understand the scripts, we have included a list of all the scripts used in this tutorial and their key design features in Appendix A, along with diagrams showing which other scripts and GameObjects it interacts with.

Files

The most up-to-date files for this project can be downloaded from: [PROJECT URL]

There are two Scenes in this project: “TutorialSTART” and “TutorialEND”. The former is the starting point of the tutorial. The latter shows how the final Scene should look after completing the tutorial.

This tutorial assumes you already know basic Unity controls, such as positioning objects in a scene, so TutorialSTART already has the scenery and some props in place, but with no enemies, no player and no interactivity.

Typographical Conventions

This is a long tutorial containing a lot of information. To make it easier to follow, some simple conventions are used:

Backgrounders & Tangents




Text in boxes like these contains additional information that may help clarify the main text.

Scripting code will appear in boxes like that shown below:

```
// This is some script code.  
Function Update()  
{  
    DoSomething();  
}
```

NOTE *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 text to save space.*

Actions you need to perform within Unity are shown like this:

-  Click on this;
-  Then this;
-  Then click Play.

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, such as the `Update()` function in the script example above.

Unity Conventions

Unity is a unique development system. Most developers will be used to working in a code editor, spending 90% of their time editing code and even writing code to load up and use assets. Unity is different: It is asset-centric rather than code-centric, placing the focus on the assets in much the same way as a 3D modeling application. For this reason, it is worth understanding the key conventions and terminology unique to Unity development:

Projects

A game built in Unity will consist of a **Project**. This contains all your project's elements, such as models, scripts, levels, menus, etc. Usually, a single Project file will contain all the elements for your game. When you start Unity 2.0, the first thing it does is open a Project file. (If you have only just installed it, this will be the Project file containing the Islands Demo.)

Scenes

Each Project contains one or more documents called **Scenes**. A single Scene will contain a single game level, but major user-interface elements, such as game menus, game-over sequences or major cut-scenes may also live in their own Scene files. Complex games may even use entire Scenes just for initialization purposes. Thus all levels in a game will most likely be Scenes, but not every Scene will necessarily be a game level.

GameObjects, Components, Assets & Prefabs

Key to understanding Unity is the relationship between a **GameObject** and a **Component**.

GameObjects

A **GameObject** is the fundamental building block in Unity. A **GameObject** is a container with an associated *Transform*, which defines its position and orientation. A **GameObject** usually contains one or more **Components**. Together, these define an **Asset**.

GameObject Hierarchies

The real power of the **GameObject** is its ability to contain other **GameObjects**, acting much like a folder in OS X's Finder. This allows *hierarchical* organization of **GameObjects**, so that, say, a complex model or a complete lighting rig can be defined under a single parent **GameObject**. (In fact, most models will appear in Unity as a hierarchical of **GameObjects** because this reflects how they are defined in the modeling package.) A **GameObject** defined inside another **GameObject** is considered a *child GameObject*.

Components

Components are the elements you use to build an Asset.

A Component may represent visible entities, such as models, materials, terrain data or particle system. Other Component types are more abstract, such as Cameras and Lights, which do not have a physical model representing them; instead, you will see an icon and some wire-frame guidelines illustrating their key settings.

A Component is *a/ways* attached to a GameObject. It cannot live alone. Multiple Components can be attached to the same GameObject. Some Component types, such as scripts, can be added to the same GameObject multiple times; others, such as those used to define particle systems, are exclusive and can only appear once in any single GameObject. (I.e. if you want to define multiple particle systems, you will most likely need to use a hierarchy of GameObjects, each containing its own set of particle system Components.)

Assets

An **Asset** is a convenience: it is a GameObject with one or more Components and/or child GameObjects, the whole defining a single element of your game. It can also, in some cases, be just a Component: a script, a particular material, an audio clip, a light, and so on.

For example, a player character could be defined as a single asset, containing the model and its associated animations. It could also contain script Components, audio clips and any other Components it needs to function.

Custom Icons & Gizmos

You can tell Unity to display custom icons and other visual information for your Assets if you wish. We will see an example of this in [the next chapter](#).

Your project's Assets are shown in the Project Pane. When you drop one into your Scene, it appears in the Hierarchy Pane, which defines the content of the Scene. (A Scene is equivalent of the stage in a theater. It can be a level, a menu, a multiplayer game lobby – whatever you wish.) The Project Pane is retained across all Scenes in your Project.

Prefabs

A **Prefab** is an Asset which has been defined as a *template*. When you place a Prefab into your Scene, Unity places a link to the Prefab instead of a complete copy. This is called *instantiation*. Each link you make is referred to as an *instance* of the Prefab.

Why would you want to do this?

Simple: if you click on a Prefab in your Project Pane and tweak its settings, you will find that those changes are instantly reflected in all the instances in your Scene. This makes Prefabs ideal for complex entities, such as bullets, enemies and so on. If you

find your enemy isn't behaving correctly, you only need to adjust the script or settings in the original Prefab instead of editing each one in the Scene individually.

However, if you need to adjust a couple of settings in a specific *instance* of a Prefab, you may do this too: these changes will only affect that particular instance.

Prefabs are displayed in blue text in both the Project and Hierarchy Pane.

NOTE *A Prefab instance cannot have additional Components added to it as doing so will break the link to the original Prefab. Unity will warn you if you try and do this. Unity will, however, allow you to update the original Prefab with such changes after the link is broken.*

Acknowledgments

This tutorial could not have been produced without the following people:

David Helgason, Joachim Ante, Tom Higgins, Sam Kalman, Keli Hlodversson, Nicholas Francis, Aras Pranckevičius, Forest Johnson and, of course, Ethan Vosburgh who produced the beautiful assets for this tutorial.

First Steps

Every platform game has its star character who the player controls. Our star is *Lerpz*.



Animating Lerpz

In this chapter we will look at:

- Implementing third-person player and camera controls
- Controlling and blending animations
- Using particle systems to implement a jet-pack's thrusters
- Adding a blob-shadow to the player
- Maintaining the player's state
- Handling player health, death and re-birth.

Before we can begin, we need to know what this game is all about. In short, we need...

The Plot

Our hero is Lerpz: an alien visiting Robot World Version 2. This replaced Robot World Version 1, which suffered a particularly brutal segmentation fault and abruptly crashed into its sun many years ago.

Unfortunately, Lerpz has had some bad luck: his spaceship has been impounded by the corrupt local police. After looking high and low, Lerpz has found found his spaceship,

but how to get it back from Mr. Bigger's nastier, obsessive-compulsive cousin, Mr. Even Bigger?

Mr. Bigger loves nothing more than artistically arranging fuel canisters on his floating patio. He particularly admires how they glow when he places them on hover pads. (And, of course, they're cheaper than fitting normal garden lights.)

But there's something Mr. Bigger hasn't realized! Thanks to his penny-pinching ways, Lerpz knows that, if he collects all the fuel canisters, the power used to keep them hovering will overload the security system. This will shut down the impound lot's fence and free Lerpz's spaceship. Lerpz can then enter his spaceship, add the fuel from the cans and fly away to freedom.






All our hero has to do is collect enough fuel canisters and the impound's force field will automatically shut down. Lerpz can then get back into his space car and drive it away. Mr. Bigger's hired robot guards will try to stop Lerpz, but luckily, they're not particularly bright.

Now that that's out of the way, we can start fleshing out our hero.

Introducing Lerpz

 Open the project up and view the "TutorialSTART" Scene.

Our first step is to add Lerpz to our Scene:

-  Open the **Objects** folder in the Project Pane;
-  Drag the **Lerpz** asset into either the Scene View or the Hierarchy View;
-  Click on the new **Lerpz** entry in the Hierarchy and rename it to **Player**;
-  Keep the **Player** object selected, move the mouse over the Scene View and tap the **F** (focus) key to center the view on the Lerpz model.
-  Move Lerpz onto the raised platform with the Jump Pad, near the Jail.

If you click Play now, you should see Lerpz standing in the courtyard outside the jail. At this stage, Lerpz cannot be moved and the camera also needs to be linked to our player's character.

 Click the Play button again to stop the game.

We need to get Lerpz moving, but first, we need to step back a moment and take a look at our camera.

Third Person Cameras.

In a first-person shooter, the camera is the player's point of view, so there is no need to worry about making it follow another object around the scene. The player controls the camera object directly. First-person cameras are therefore relatively easy to implement.

However, a third-person viewpoint camera requires a camera that can follow the player around. This seems simple enough until you realize the camera also needs to avoid getting scenery between the player's character and the camera's viewpoint. This can be achieved using raycasting to check for unwanted objects between the camera and player avatar, but there are some special cases to consider. For example:


What happens if Lerpz is backed up against a solid wall? Should the camera move above and look down on the player? Should it move to the side?

What if an enemy gets between the camera and our player avatar?

How should the player controls work? Should they be relative to the camera's view? If so, this could get very confusing if the camera moves in an unexpected direction to avoid an obstacle.

A number of solutions for third-person cameras have been tried over the years. It's arguable that none have ever been 100% perfect. Some solutions faded out anything between them and their focus, making walls or enemies semi-transparent. Other options include cameras which follow the player around, but which will, if necessary, move through walls and buildings to keep the player's view consistent.

The project supplied with this tutorial includes a few different camera scripts, but for the purposes of this tutorial, we'll use **SpringFollowCamera**. You'll find it in the Project Pane inside the **Camera** sub-folder of the **Scripts** folder.

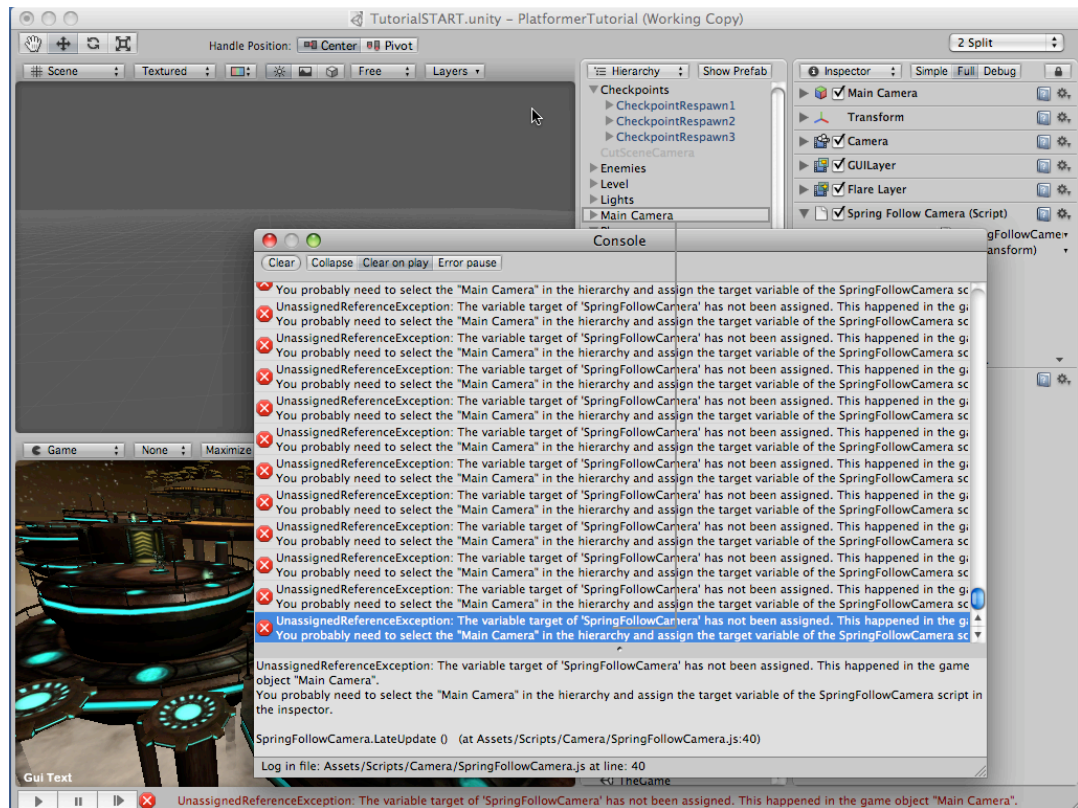
 Drag the **SpringFollowCamera** script from the Project Pane onto the Main Camera object in the Hierarchy Pane.

 Click Play.

You will get an error message. This appears just to the right of the Play, Pause and Step buttons at the bottom of Unity's window.

 Bring up the Debug Console (Shift+Cmd+C), if it is not already visible.

This displays any warnings, errors and other debugging information from your game. You will probably see a lot of copies of the error message repeated in the log. Highlight one and the pane below the log will show a bit more information about this error message, as shown in image 3.1.



"No target" error message.

TIP Whenever possible, the Debug Log window will show a line linking to the offending GameObject in the Hierarchy, or to the Project Pane if the fault is in a Prefab or Script.

The `UnassignedReferenceException` error type is one you will likely see very frequently. It sounds scary, but all it means is that a script variable has not been set. The Debug Log explains this too, so let's do as it suggests:

❏ Click on the Main Camera object in the Hierarchy Pane and look at the Spring Follow Camera (Script) Component's properties.

The Target property is set to None (Transform). This property is the target which the camera needs to point at. We need to set it.

- ❏ Stop the game if you haven't done so already.
- ❏ Click on the Main Camera object in the Hierarchy Pane.
- ❏ Look at the "Spring Follow Camera" script component in the Inspector and note the "Target" item.
- ❏ Drag our "Player" object from the Hierarchy Pane onto the Target setting to set it.

Making Changes While Playing

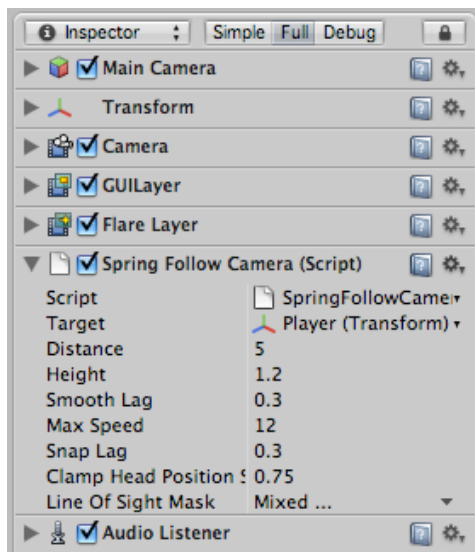
When you are playing the game, Unity will let you tweak the properties of the various game objects and components in the game. However, it will not save them! The moment you stop the game, any changes will be discarded!

If you want your changes to stick, **always stop the game first!**

If you click Play now, the camera still won't work. You will see a complaint from the camera's new script: it wants the target to have a "Third Person Controller" script attached to it. This is because a third-person camera is closely tied to the player controls: it needs to know what the player is doing so that it can react accordingly.

The final settings should look as shown in the image below. (Ignore the Line Of Sight Mask setting for now. We'll come to this later in this tutorial.)

Experiment with the numbers if you don't like the way the camera works; this is a subjective judgement and there is no single correct setting for something like this.



Spring Follow Camera script settings.

This is the first in a series of dependencies that we need to deal with.

- 🔗 Complete the connection between the camera and the player by dragging the **ThirdPersonController** script from the **Scripts->Player** folder in the Project Pane onto our **Player** GameObject (in the Hierarchy Pane).

The **Third Person Controller** script also has its own requirements and dependencies. The most important of these is the **Character Controller** component. Luckily, the script already tells Unity about this, so Unity will add this component for us.

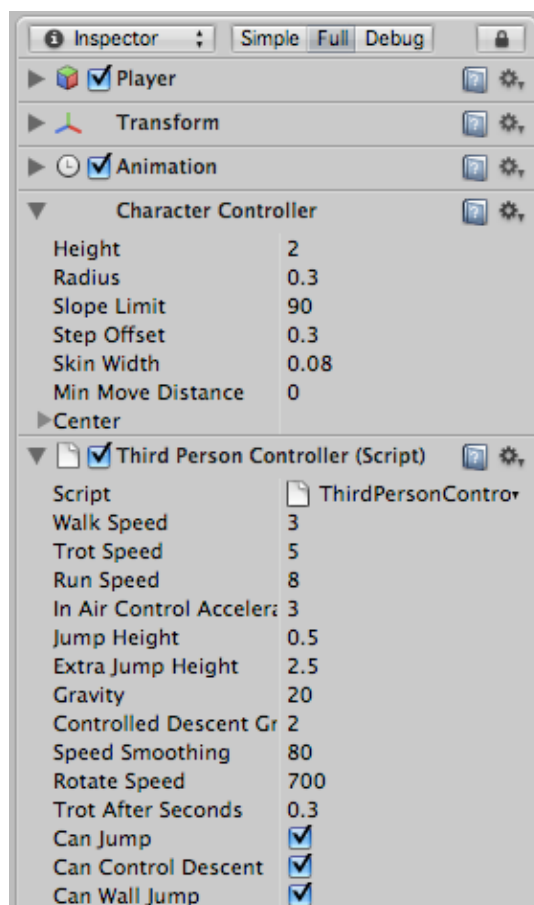
Connections & Dependencies.

Unity excels at showing visual assets, but these also have to be connected to each other to provide the interactivity we expect from a game. These connections are difficult to show visually.

These connections are known as dependencies, and it's what you get when one object requires a second object to function. That second object may, in turn, require yet more objects to work. The result is that your assets are tied to each other with myriad virtual bits of string – scripts – tying them all together to make a game.

Defining all these dependencies is a key element of game design.

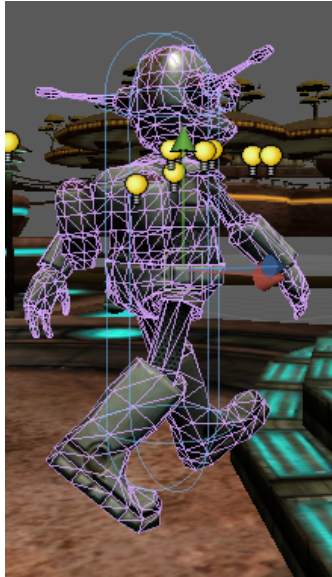
 Select the **Player** object and look in the Inspector. It should look similar to this:



Character Controller and Third Person Controller Script Components in place.

Our next step is to adjust the Character Controller. At the moment, the Capsule Collider it uses is located too far down in the Y axis, so Lerpz stands on thin air. (You can see the Collider's position in the Scene View: it's the long blue cylindrical wireframe shape.) We need to change the Center Y value. A little experimentation suggests a setting of 1.03 will align its lower end perfectly with Lerpz's feet.

🔍 Position the Capsule Collider as shown in the image below.



Adjusting the Character Controller's Capsule Collider.

If you click Play now, Lerpz should now move around when you use the control keys with his feet firmly on the ground.

The Character Controller & the Third Person Controller script.

The **Character Controller** simplifies movement for a player (and many non-player) character types. At its most basic level, it consists of a capsule collider tied to a basic movement system, allowing our character to move around, climb steps and slide up or down slopes. You can change the maximum step and slope sizes in the Inspector.

In most games, the player's avatar is capable of impossible physical feats such turning and stopping instantaneously, leaping improbable distances and other actions which would be difficult to model using traditional physics. The Character Controller therefore decouples our player avatar from the physics engine, providing basic movement controls.

The Character Controller is itself controlled by a Controller script. This is just an ordinary Unity script asset which talks to the Character Controller and extends its capabilities to meet the needs of the game. In our project, the **Third Person Controller** script performs this duty and adds the necessary support for our platform game. It reads the joystick, keyboard, mouse or other input device and acts upon it to control the player's

avatar. The Unity **Input Manager** (*Edit->Project Settings->Input Manager*) allows you to define how the input devices control the player.

The next step is to make Lerpz animate correctly and add the additional movements, such as jumping and punching...

Animating Lerpz.

At this point, Lerpz is just gliding across the scenery. This is because the Character Controller doesn't handle animation. It doesn't know anything about our player's model or which animation sequences apply to each movement. We need to connect Lerpz to his animation sequences and this is done with the **ThirdPersonAnimation** script.



Use the Component menu to add this script to the Player game object.

If you click Play now, you'll see Lerpz animating correctly.

So what's going on here? What does this script do? The answer lies in how Unity handles character animation data.

Character Animation.

Character animation sequences are created within a modeling package, like Cheeta3D, 3D Studio MAX, Maya or Blender. On importing into Unity, these sequences are automatically extracted and stored in an Animation component.

These animation sequences are defined on a virtual skeleton, which is used to animate the basic model. These skeletons define how the model's mesh -- the important data defining the visible surfaces of the model itself -- is modified and transformed by the engine to produce the required animation.

Skeletons & Armatures

If you are familiar with stop-motion or "claymation" animation techniques, you may be aware of their use of metal armatures. The animated models are built around these armatures. The virtual skeletons used in 3D models are directly equivalent to these and are rarely as complex as real skeletons.

The mesh component of such models is commonly referred to as a *skinned mesh*. The virtual skeleton provides the bones beneath the mesh and defines how it animates.

Animation blending.

Character animations are usually *blended* together to provide the necessary flexibility for a game. For instance, an animated walk cycle could be blended with a series of speech animations, the result being a character that is walking and talking at the same time.

Blending is also used to produce smooth transitions between animations, such as the transition between a walk cycle and a punch sequence.

We need to use a script to tell Unity when we need to switch animations, when animation blending is needed and how it should be done. This is where scripting comes in.

The Third Person Animation script.

The Lerpz model we're using was created for multiple projects and contains fifteen animation sequences. Only eleven are used in this tutorial. If you select the Player object in the Hierarchy pane and look at the Inspector, you will see all fifteen animation sequences listed within the Animation component, of which only the following are actually used in this tutorial:

- *Walk* – The normal walk cycle.
- *Run* – A running animation. (Hold the Option key while playing to run.)
- *Punch* – Played when attacking an enemy robot guard.
- *Jump* – Played when Lerpz leaps into the air.
- *Jump fall* – Played when Lerpz's leap reaches its apex and he starts to fall.
- *Idle* – A loop played when Lerpz is idle.
- *Wall jump* – A backflip animation played when Lerpz jumps off a wall.
- *Jet-pack Jump* – Played when Lerpz's jet-pack is slowing his fall.
- *Ledge fall* – Played when Lerpz steps off the edge of a platform.
- *Buttstomp* – Played when Lerpz has been struck by a robot guard.
- *Jump land* – Played when Lerpz lands after a jump or fall.

Most of these animations are dealt with by the **ThirdPersonPlayerAnimation** script, which checks the controls the player is using and reacts accordingly. Some animations are layered over others while others are simply queued up one after another. The script is mostly a set of message responder functions. The relevant messages are fired off by the **ThirdPersonController** script, which reads the input devices and updates the character's state accordingly.

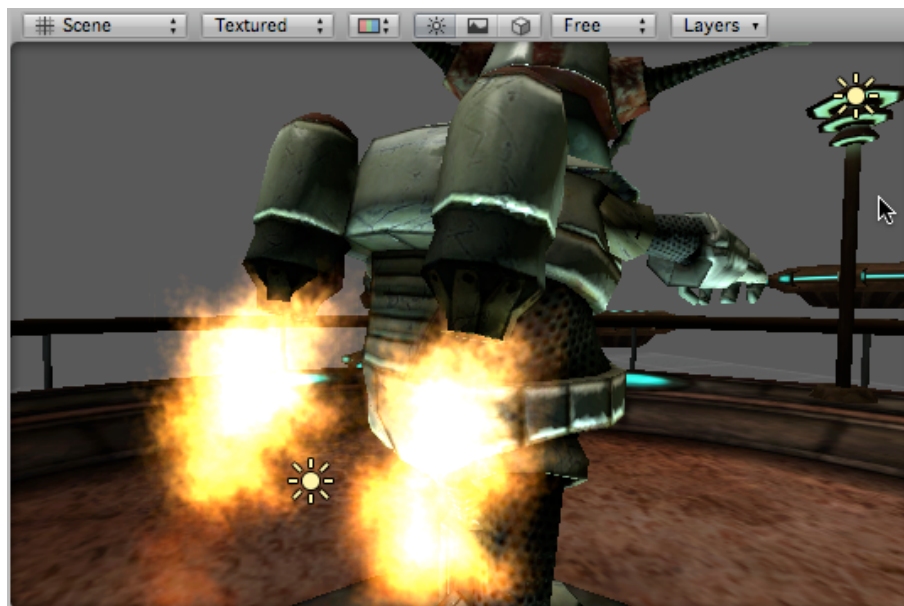
Lerpz's attacking move – his punch – is dealt with in a separate script, **ThirdPerson-CharacterAttack**. This may seem an arbitrary split, but it is not: most of the basic movements – walking, running, jumping, etc. – are pretty similar no matter what your

player's character looks like. However, attacking and defensive moves tend to be much more varied. In some platform games, the player character might have a gun; in another, he might perform a martial arts move. In this tutorial, Lerpz is a master of the popular Western martial art known as *Fisticuffs*, a name which translates to "hitting your opponent very hard with your fist". The animation is a simple punching animation.

Gizmos

ThirdPersonCharacterAttack also includes a useful testing feature: a gizmo which draws a sphere to represent the area affected by Lerpz's punching action. Gizmos are drawn inside one of two Gizmo-drawing message handling functions. In this example, the gizmo -- wireframe yellow sphere drawn at the punch position and displaying its area of effect -- is drawn in response to the `OnDrawGizmosSelected()` function. This function must be static and will be called by the Unity Editor itself. An alternative is `OnDrawGizmos()`, which is called by the Unity Editor every update cycle, regardless of whether the parent **GameObject** has been selected.

The Jet-Pack



Lerpz's Jet-pack in action.

At this point, our character is running and jumping around, but his jet-pack is not yet working. Lerpz uses the jet-pack to slow his rate of descent. The movement is already in place, but the jet-pack's jets don't animate at all. To make the jets work, we will need to add two **Particle Systems** and a **Point Light** component. The particle systems


will produce a flame-like effect, while the point light will give the illusion of the flames acting as a source of illumination.

TIP *Ideally we would have a point light source for each jet, but the jet exhausts are close enough to each other that we can get away with just the one. As lights are computationally expensive, this is a handy optimization.*

What is a Particle System?

Particle Systems emit dozens of particles -- usually flat 2D billboards or sprites -- into the 3D world. Each particle is emitted at a set speed and velocity, and lives for a certain time. Depending on the settings and billboard materials used, these particle systems can be used to simulate anything from fire, smoke and explosions to star-fields.

Adding the Particle Systems.

 Use the GameObject Menu to create an empty GameObject in the Hierarchy Pane.

 Rename this GameObject "Jet".


With the new GameObject selected, add:

 An Ellipsoid Particle Emitter

 A Particle Animator

 A World Particle Collider

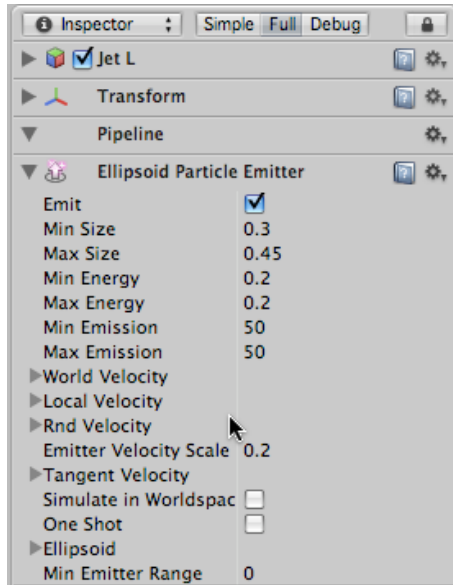
 A Particle Renderer

 Uncheck the "Enabled" checkbox in the Inspector for the Particle Renderer Component. This will disable it temporarily.

 Position the Jet directly below Lerpz's right-hand jet exhaust.

 Re-enable the Particle Renderer.

 Adjust the settings for the Ellipsoid Particle Emitter as shown below:



Ellipsoid Particle Emitter settings.

These settings result in a narrow stream of particles which we will use to simulate a jet of flame.

TIP *If the particles are not moving directly downwards, use Unity's rotation tools to rotate our object until the jet is moving in line with the jet-pack's exhaust.*

When we're done, the particle system will be attached to the Player's "torso" child object in its hierarchy. This will cause the jet to follow the player's movements. At this point however, we're primarily interested in getting it to look right, so don't worry too much about accurate placement.

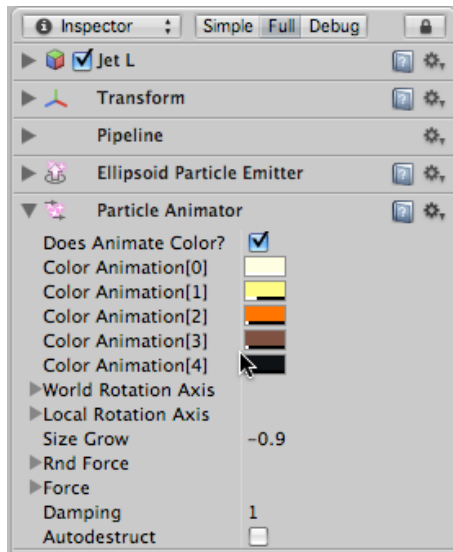
The **Min Size** and **Max Size** settings define the size range of the particles. The **Min Energy** and **Max Energy** settings define the minimum and maximum lifespan of the particles. Our particles will live for only a short time -- 0.2 seconds in this case -- before fading away.

We set the quantity of particles to emit to 50. This defines how many particles we want on screen at any one time, and thus defines the rate of particle emission. We have chosen to set the minima and maxima of value ranges to similar values or it'll look like the jet is sputtering rather than blasting away smoothly. The result should be a smooth flow of particles.

NOTE *We've disabled "Simulate in Worldspace" here. This helps give the impression that we have a hot, fast jet of gas rather than a much slower flame, even though the particles aren't moving all that quickly. By making the jet ignore Lerpz's twists and turns, it looks similar to the hot, steady flame from a blow-torch.*

Now we move on to the **Particle Animator**.

Set this Component's settings as shown:



Particle Animator settings.

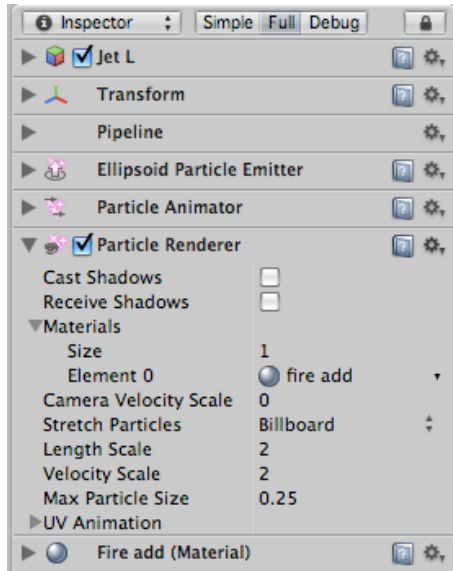
The **Particle Animator** will animate the particle colors as they age. The particles will start off white, darkening through yellow and orange as our virtual jet cools. Since we're going to be rendering a texture into each particle, the Particle Animator will be used to tint this particle, so the color animation will be subtle, but hopefully effective.

The color picker dialog which appears when you click on a color also offers an Alpha slider. This should be set to around 70% for Color Animation[0], decreasing to around 10% or so for the last color in the sequence. This will make the particle 'flames' fade as they cool.

Next is the **Particle Renderer**. This Component draws each particle, so it needs to be told how the particles will appear. It also defines the material to use to render each particle. We want a flame-like jet effect, so we shall use the "Fire add" material provided in the *Particles* folder.

TIP This asset can be found in the *Standard Assets* folder.

Set this component's values as follows:



Particle Renderer settings.

The Stretch Particles setting tells Unity whether the particles should be rendered stretched if they are moving at high speed. We want the particles to stretch a little according to their velocity. This adds a subtle visual cue and makes the small, round shapes we're using for this jet blend more into each other.

NOTE The **Cast Shadows** and **Receive Shadows** settings have no effect unless you use a custom shader. This is an advanced topic beyond the scope of this tutorial.

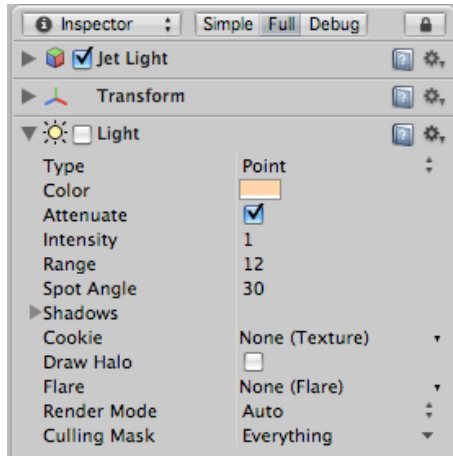
Adding the Light.

A Particle System just spits out images, but the resulting flame-like effect does not emit light. To complete the illusion, we need to add a **Point Light**. This will be switched on and off at the same time as the jets. The result will be a jet of flame which appears to light up its immediate surroundings. Remember, we will only use a single light, rather than one light per jet.

- ✎ Create a new **Point Light** GameObject.
- ✎ Name this "Jet Light" and position it between the two jets on Lerpz's back. This light will create an illusion that the jets are emitting light.

For this effect to work, we need a bright point light with a high intensity.

- ✎ Select the Light and adjust settings as shown:



Jet-pack Light settings.

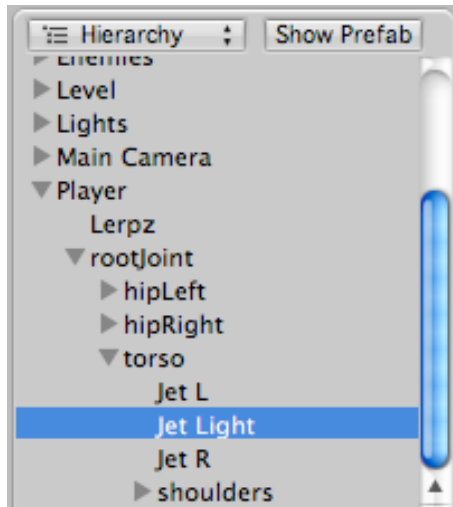
Why no shadows?

Shadows are computationally expensive for most hardware. It makes sense to avoid calculating them if we can avoid it, and this is one area where we can get away with it. The jets aren't very big, so they only need to light up Lerpz's back. The point light will also be reflected in nearby scenery, but it won't be bright enough to make the lack of shadows noticeable.

The next step is to update the Player prefab to include the jets and light object.

- ✎ Go to the Player object in the Hierarchy,
- ✎ Open it up until you find the "torso" child object.
- ✎ Drag the Jet object onto this object twice. This will create two "Jet" instances.
- ✎ Rename the two "Jet" instances to "Jet L" and "Jet R",
- ✎ Drop the Jet Light onto the same "torso" object.

You should now have an object hierarchy that looks something like this:



Jet-pack Hierarchy.

- ✎ Use Unity's manipulation tools to position each jet particle system under its respective jet outlet in Lerpz's model.
- ✎ Move the Jet Light to a point between the two jets.

When you've achieved this, Lerpz should now have two flaming jets gushing from his jet-pack as he moves around. We're almost done!

The final step is to make the jet-pack's jets and light activate only when he's jumping. This is achieved through scripting.

- ✎ Look through the Scripts tree in the Project Pane until you find the **JetPackParticleController** script.
- ✎ Drag this onto the top-most "Player" object in the Hierarchy Pane to add the script to our player character.

You should now find that the jet-pack works as expected. The script controls the two particle systems and the light, synchronizing them with Lerpz's movements and triggering all three elements together whenever the player presses the jump button to jump or to slow his descent.



Lerpz's Jet-pack in action.

Blob Shadows

Lerpz must be easy to identify at all times, so that players won't lose track of their avatars when the game gets visually busy. Most of this work is up to the artists and the game's designer, but there are some elements which have to be handled by Unity itself.

One of the most important of these is shadowing and lighting. To aid performance, these are often 'baked' into the textures, but this technique only works well on static objects, such as sets and fixed props. A character walking under a streetlight needs to react to that light in real time. The road beneath the character can have the lighting baked in, but the character cannot use this trick.

The solution is to position dynamic lights where the textures suggest it should be, but make the lights only affect moving objects. Such lights have already been placed in the scene for you. (Positioning these is often best left to the artists who create the scenery assets. They'll know where the lights should be.)

This leaves one final element: shadows.

In a 3D platform game, the shadow plays a key role in telling us where the character will land if he is jumping or falling. This means Lerpz should have a good, visible shadow, which is not the case at the moment.

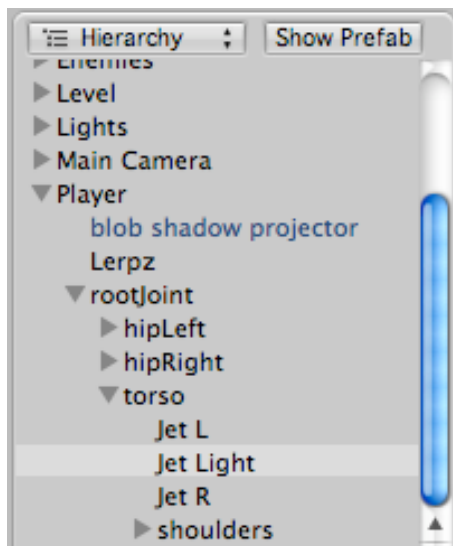
Shadows can be produced using lights, with the shadow computed and rendered in real time by the graphics engine. However, such shadows are expensive in terms of processing power. In addition, not all graphics cards can compute shadows quickly or effectively; older cards may not be able to do so at all.

For this reason, we will use a *Blob Shadow* for Lerpz.

Adding a Blob Shadow.

A Blob Shadow is a cheat. Instead of casting rays of light and checking if they hit anything, we simply project a dark image – in this case just a circular black blob – onto anything below our character. This is quicker and easier for the graphics card to do, so it should work well on all ranges of hardware.

Unity includes a Blob-Shadow prefab in its Standard Assets collection, so we shall use this rather than creating our own. This asset has already been imported and added to the project in the *Blob-Shadow* folder. Open this folder and click on the **blob shadow projector** Prefab and drag it onto our top-level character object – **Player** – in the Hierarchy Pane. This should add the Projector just below the top level in our Player object's hierarchy:



The Blob Shadow Projector Prefab in the Player's hierarchy.

Next, you will need to modify the blob shadow projector's Position and Rotation data so that it is directly above our character and pointing directly down at the ground.

- ✎ Select the *4-Split* layout.
- ✎ Set the **blob shadow projector's** **Rotation** values to *90*, *180* and *0* respectively.
- ✎ Now use the side and top views to move the projector directly over Lerpz' head. You might want to move it up or down a little until you're happy with the shadow's size.

Creating a new Layer.

At this point you will have noticed that the blob is also being projected onto Lerpz. We don't want this to happen. There are two options to get around this: move the

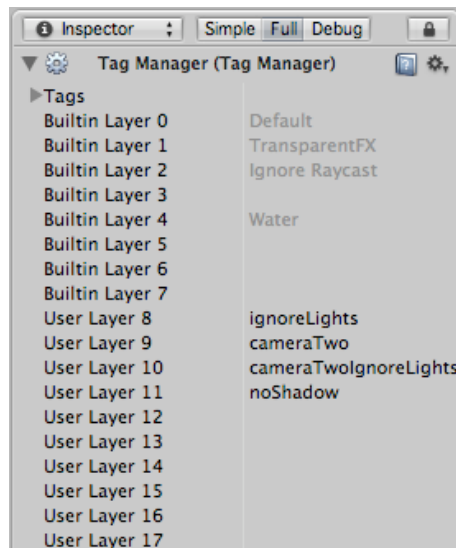
Near Clip Plane setting further away from the projector, or simply tell it not to project onto objects in specific Layers. We shall use the latter option.

Why not adjust the Near Clip Plane?

This technique might seem easiest at first glance, but the plane would need to be adjusted by scripting to take into account Lerpz's animations. His feet move further out when he jumps, then briefly move a little closer when he lands again. Since the shadow must always be projected onto the ground on which Lerpz stands, this means the Near Clip Plane cannot remain the same throughout these sequences.

- Open up the **Player** GameObject and then on the **Lerpz** object within. This selects the model data itself.
- Open the *Layer* drop-down in the Inspector.
- Choose *Add new layer...*
- Click on the first empty **User Layer** entry and name it **noShadow**.

You should now see something like this in your Inspector:

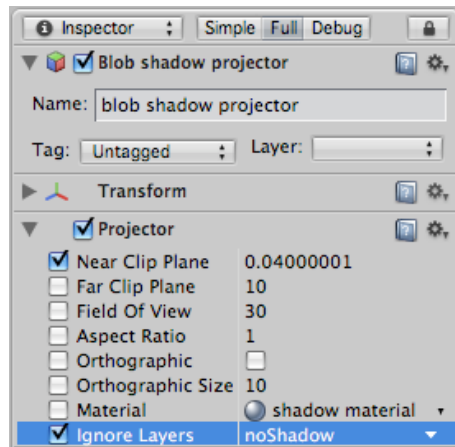


Adding a new Layer using the Tag Manager.

- Now click back on the **Lerpz** object in the Hierarchy Pane to bring up the usual Inspector settings.
- Click on the "Layer" drop-down and set it to the new layer name, **noShadow**.

Next we need to tell the **Blob Shadow Projector** not to project onto objects in this Layer.

- ✎ Bring up the blob shadow's properties in the Inspector and look at the **Ignore Layers** entry in the Projector Component.
- ✎ Use the drop-down menu to the right to select the **noShadow** Layer, as shown:



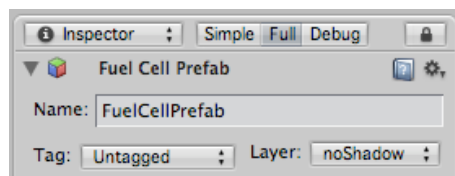
The Blob Shadow Projector settings.

If you now play the game and move around you should see the shadow behaving pretty much as expected.... except if you jump around near the collectable fuel cells. If you try this, you will see the item showing the blob shadow too.

We want collectable items to stand out at all times, it makes sense to tell the Blob Shadow Projector to avoid these too.

We'll be looking at these collectables in much more detail in the next chapter, but let's fix this problem now while we're here.

- ✎ First, stop the game.
- ✎ Now go to the Project Pane and locate the **FuelCellPrefab** and **healthLifePickUp-Prefab** objects. You'll find them inside the *Props* folder.
- ✎ Select the root object of each Prefab and set its Layer to "noShadow", as shown below:



Layer changed to "noShadow"

NOTE When making a change to a parent object, Unity will often ask if the change should also be applied to that object's children. Doing so can be dangerous if you haven't thought through all the ramifications.

In this case, we want all the child objects of the "Player" GameObject to be in the same "noShadow" layer, so when Unity asks, agree to propagate the changes.

Scripting Concepts

History is littered with surprisingly complex machines – known as automata – built by our ancestors for the purposes of entertainment. Some were very elaborate and could even perform simple plays using puppets. Others were interactive and changed their behavior according to user input. These machines were fundamentally the same: the designer created assets – puppets, props, etc. – and then designed machinery to make those assets behave as they desired.

The basic principle has remained unchanged over the years. Computers have merely turned physical machinery, built of steel and springs, into virtual machinery made from *scripts*.

Most scripts are centered on a concept popular in game development: the *Finite State Machine*. A Finite State Machine essentially defines a system of interacting states.

Computer games make very heavy use of this concept. A state can be almost anything, such as whether an object should be rendered at all, whether it should be subject to the laws of physics, be lit or cast a shadow, whether it can bounce, and so on. The Inspector Pane lets us change most of these states directly because these states are common to almost all games.

However, there is another type of state which is specific to the game itself. Unity does not know what the player's avatar is an alien, how much damage Lerpz can take or that Lerpz has a jet-pack. How can Unity be aware of the robot guards' required behavior or how they should interact with Lerpz?

This is where scripts come in. We use scripts to add the interaction and state management specific to our game.

Our game will need to keep track of a number of states. These include:

- The player's health;
- The number of fuel canisters the player has collected;
- Whether the player has collected enough fuel to unlock the force field;
- Whether the player has stepped on a jump pad;
- Whether the player has touched a collectable item;
- Whether the player has touched the spaceship;
- Whether the player has touched a respawn point;

- Whether the Game Over or Game Start screens should be shown;
- ...and more.

Many of these states require tests to be made against other objects' states to ensure they're up to date. Sometimes we even need intermediate states, to aid a transition. For example, collecting a fuel canister will force a check to be made to see if the player has enough to shut down the force field.

Organization & Structure.

In this tutorial the state machines for the player, the level and the enemies are handled by a bunch of scripts linked to various Game Objects. These scripts talk to each other, sending each other messages and calling each other's functions.

There are a number of ways we can set up these links:

- By adding a link exposed in the Inspector, which you drop the relevant object onto. This is ideal for general-purpose scripts which you intend to re-use in other projects.

This is the most efficient as the script merely plucks the data from the relevant variable and doesn't need to do any searching. However, it does assume you know in advance exactly which object or component you'll be linking to.

We use this option for the cut-scene cameras in **LevelStatus**. This gives us the flexibility to set up multiple cameras, one for the "level exit unlocked" cut-scene and another for the "level complete" sequence. In practice, we're only using two cameras in the game; one for the player and the "level complete" sequence, the other for the "unlocked" cut-scene. But the option is there to change this.

- Setting up a link within the script's Awake() function. This function is called on every script you write before the first Update() event is fired at the GameObject it is attached to. Setting up the link here allows you to cache the result for later use in an Update() function. Typically, you would set up a private variable with a link to another GameObject or component you need to access within your script. If you need to do a `GameObject.Find()` call to locate the relevant object, it is much better to do so at this stage as this particular function is quite slow.

This option is more suited to those situations where you don't need the flexibility of the first option, but don't want to have to perform a convoluted search for the object every game cycle. The solution is therefore to search for the object when the script is 'woken up', storing the results of the search for use in the update section.

For example, the **LevelStatus** script, which handles level state, caches links to a number of other objects, including the Player object. We know these won't change, so we may as well make the computer do this work for us.

- Setting up a link during the `Update()` function. This function is called at least once per game cycle, so it is best to avoid using slow function calls here. However, the `GameObject.Find()` and `GetComponent()` functions can be quite slow.

This option is used for those situations where the object you need could change at any time during the gameplay.

For example, which of the multiple Respawn points in this tutorial's Scene should the player be respawned at? This clearly changes while the game is running, so we need to handle this accordingly.

The problem with this is that it's slow, so it is best to design your game such that you don't need to do this often.

Scripts in a Visual Development Environment

Unity is an unusual tool in that its focus is on the visual assets rather than the links and connections between them. A large Unity project can have dozens of scripts of varying complexity dotted around the Hierarchy, so the design used for this tutorial uses some object-orientation techniques to alleviate this.

The script which deals with a particular part of the state machine -- e.g. player animation -- should also be the one which keeps track of the relevant state variables. This can make things a little complicated when a script needs to access a state variable stored in another script, which is why some scripts cache some values locally to make access to the information more quickly. This technique also occasionally results in chains of commands, where a function in one script merely calls a similar function in another script. The handling of the player's death and health is an example of this.

NOTE *As you get more experience with Unity, you will find other ways to handle states that may be better suited to your own games.*

Death & Rebirth

Platform game characters tend to lead risky lives and Lerpz is no exception. We need to ensure he loses a life if he falls off the level. We also need to make him re-appear at a safe spot on the level -- often called a "re-spawn point" -- so he can continue his quest.

Another point is that if Lerpz can fall off the landscape, it is also possible that the level's other residents could do so too, so these must also be dealt with appropriately.

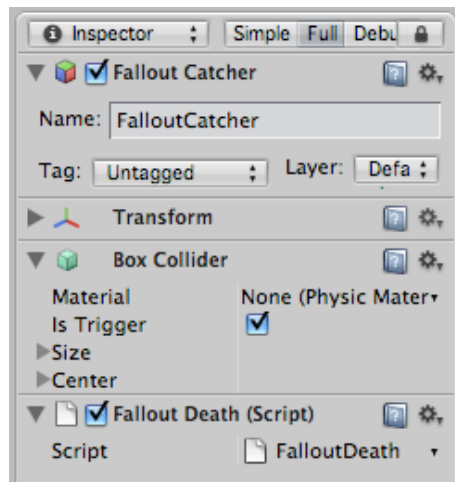
The best solution for this is to use a box collider to catch anything falling off the level. We'll make it very long and broad, so that if a player should try using the jet-pack

while jumping off, we'll still catch him.

However, Lerpz will first need somewhere to re-spawn. We'll come to the re-spawn points shortly. First, let's build the box collider:

- Create an empty GameObject.
- Add a Box Collider object to it.
- Add the "Fallout Death" script from the Component menu.

Use the Inspector to set the values as shown in the screenshot below:



Fallout Catcher settings.

If you play the game now, should find that Lerpz loses a life and then reappears at the default re-spawn point.

The Fallout Death script

This script has a number of interesting features:

- The code to handle the collider's trigger in `OnTriggerEnter()`. This function is called by Unity when the Box Collider is struck by another GameObject containing a Collider component, such as Lerpz or an enemy.
- If the player hits the box collider, the code simply calls the `FalloutDeath()` function in Lerpz's **ThirdPersonStatus** script.
- If another object with a collider object hits our GameObject, the function destroys it, removing it from the Scene.

NOTE There are three tests: one for the player, one for a simple **Rigidbody** object, and a third test to check if the object has a **CharacterController** Component. The second test looks for any props such as boxes or crates falling off the level.

We don't have such items in this level, but you can add one if you would like to experiment. The third test is used for enemies as these won't have ordinary physics attached.

In addition, we have:

- The utility function – `Reset()` – which ensures any required Components are also present,
- The `@Script` directive, which also adds the script directly to Unity's *Component* menu.

Respawn Points

When the player dies, we need somewhere safe for him to re-appear. In this tutorial, Lerpz will reappear at one of three re-spawn points. When Lerpz touches one of these points, it will become active and this will be where he reappears if he dies.



Lerpz standing on an active Respawn point.

The re-spawn points are instances of the **RespawnPrefab** prefab object. (You'll find it in the Project Pane's *Props* folder.)

This prefab is a model of a teleport base, coupled with three complete particle systems, a spotlight and some other odds and ends. Here's the basic structure:

- **RSBase** contains the model itself: a short cylindrical base with a glowing blue disc in the center.
- **RSSpotlight** is a spotlight object which shines a subtle blue light up from the surface of the model, giving the illusion that the blue texture is glowing.
- The remaining game objects are particle systems. The **Respawn** script attached to the parent **RespawnPrefab** object switches between these particle systems depending on the prefab's state.
 - If the respawn point is inactive, a small, subtle particle effect is shown looking like a bright blue mist. This is contained in **RsParticlesInactive**.
 - If the respawn point is active, a larger, more ostentatious effect is shown. This is contained in **RsParticlesActive**.

Only one respawn point can be active on the level at any one time. When the player touches the respawn point, a collider object (set as a trigger) detects this and triggers activation of the respawn point.

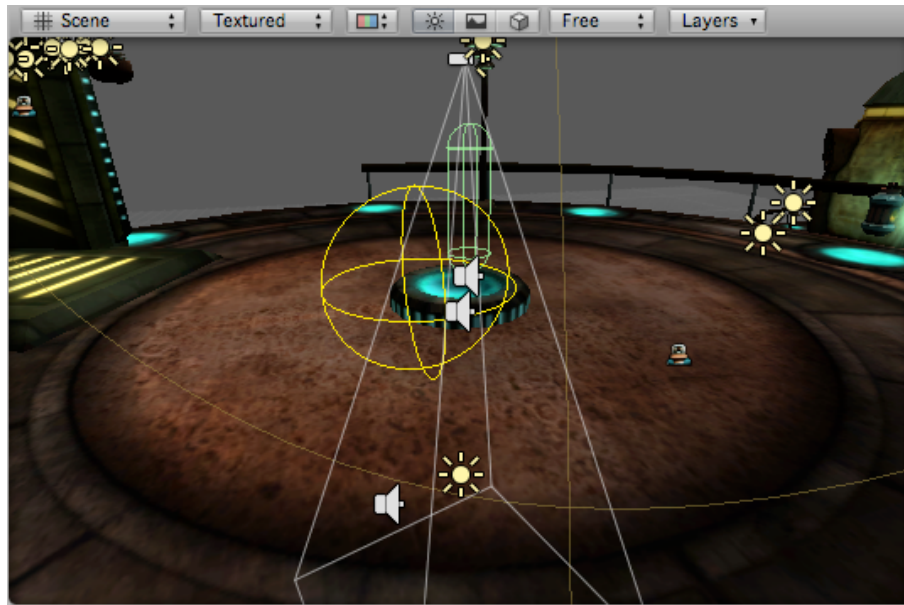
- The remaining three particle systems -- **RsParticlesRespawn1**, **RsParticlesRespawn2** and **RsParticlesRespawn3** -- are enabled together when the player is respawned at the respawn point. These are one-shot particle systems. The script lets these play, then restores the **RsParticlesActive** particle system once this one-shot sequence is completed.

The prefab contains a script, **Respawn**, which controls the state of the respawn point. However, in order for the game to know which specific respawn point the player needs to be returned to when he dies, we need to arrange the respawn points in a hierarchy under a master controller script. Let's do this now:

- 🔧 Drag the **RespawnPrefab** into the Scene View.
- 🔧 Position it as shown in the image on the next page.
- 🔧 Rename this instance **Respawn1**.
- 🔧 Repeat the above steps twice more.

I placed one beneath a fuel canister near the trees, and another close to the arena at the far end of the level, but feel free to place them wherever you wish. Feel free to add even more instances if you wish.

- 🔧 The next step is to create an empty GameObject.
- 🔧 Rename this **RespawnPoints**
- 🔧 Make all the respawn prefab instances children of **RespawnPoints**.



Positioning the first respawn point.

How it works.

When the Scene is loaded, Unity calls the `Start()` function in each instance of the Respawn script, where some useful variables are initialized and pointers to other elements cached.

The key mechanism is centered around this `static` variable:

```
static var currentRespawn : Respawn;
```

This defines a global variable named **currentRespawn**. The `static` keyword means it is shared across all instances of the script. This lets us keep track of which Respawn point is the current, active one. However, when the Scene begins, none of the Respawn points is activated, so we need to set a default one for our Scene. The Unity Inspector will not display static variable types at all, so the script defines an **Initial Respawn** property, which needs to be set for each instance. Drag the default Respawn point onto this. You'll need to repeat this for *all* Respawn points in the scene. (In the tutorial project's case, the default is set to **Respawn1**, which is located near the Jail and directly below the player's starting point.)

TIP You could also set this variable just once and apply the change to the Prefab, or even apply the change directly to the Prefab itself, if we had a lot of Respawn points.

When a respawn point is activated by the player triggering its collider, that point's **Respawn** script first deactivates the old Respawn point and then sets `currentRespawn` to point to itself. The `SetActive()` function takes care of firing off the relevant particle systems and sound effects.

The scripts also handle sound effects. These are played as one-shot samples, except for an Audio Source attached to each **Respawn Prefab**. This Component contains the "active" sound, which is a loop. The script simply enables or disables this sound as appropriate, either while playing a one-shot effect -- such as when the player is actually respawning or activating the respawn point itself -- or when the respawn has been deactivated.

NOTE *Unity makes it almost too easy to add sound effects. Whenever you plan to add such an asset, consider carefully how it will be used. For example, we haven't included a "respawn deactivated" sound because you'd never hear the sound being played; you're unlikely to position two respawn points within ear-shot of each other.*

If you were to convert the project into a multiplayer game, you might want to add such a sound and the necessary script code to handle it.

The script is not complex and you should find the script code easy enough to follow.

Setting the Scene

With our hero now mobile, the next step is to give him something to do...



First Steps

In this section we will look at building the game world where the action takes place. In movie terminology, this means building the set, placing the props and writing the scripting that lets our hero interact with them.

Our first step is to prepare the stage. The tutorial file already has the basic level mesh set up and populated with a number of collectable items. We will place a few more props and elements, but most have been done for you as placing all of these props would make for a very long, very boring tutorial!

Placing Props

🔗 Open the project up and view the “TutorialSTART” Scene.

As you can see, the basic set is provided for you, along with a number of props already in place.

Building Your Own Levels

The tutorial level was built by arranging scenery components in Maya and then importing the level into Unity. If you would like to experiment, the individual components can be found in the **Build Your Own!** folder in the Project Pane.

There are a large number of the fuel cell props alone and placing all of these would make for a very dull tutorial. If you have read the previous tutorials, you will already know how to do this anyway, so we will limit the placement to the "Health" pickups, the jump-pads and respawn points, among others.

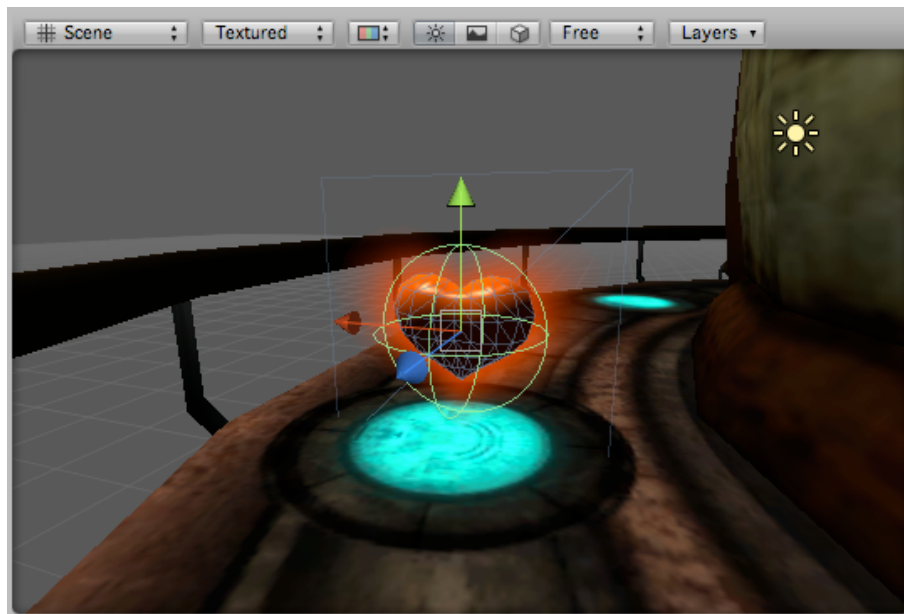
This tutorial is going to focus mainly on the interactivity. There is very little interactivity in place at this point. You can move around the scenery, but you cannot do much else and even the player animation has yet to be implemented. You cannot pick up the collectable items. You cannot use the jump pads. There is nothing in place to deal with what happens when you have collected all the items you need. There are no respawn points, nor a level-complete sequence.

In addition, there is no heads-up display showing our hero's health or collectables remaining. Nor is there any audio or music. So, plenty of work to be done! Let us begin...

Health Pickups

We begin with a simple task: adding some collectable health pickups. These are spinning, glowing hearts that add health to our player.

The pickups are already defined as Prefabs in the Project Pane. Look inside the "Props" folder and you will find the healthLifePickUpPrefab object ready to use.

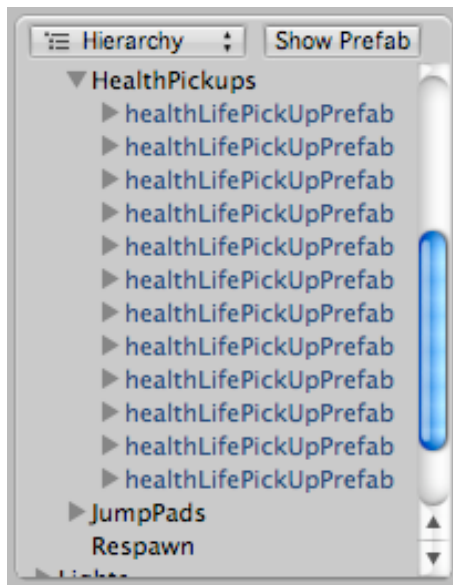


Placing a health pickup.

- 🖱️ Drag one onto the Scene View and use Unity's positioning tools to position it somewhere on the level.
- 🖱️ Repeat this process until you've placed about half a dozen of these around the map.

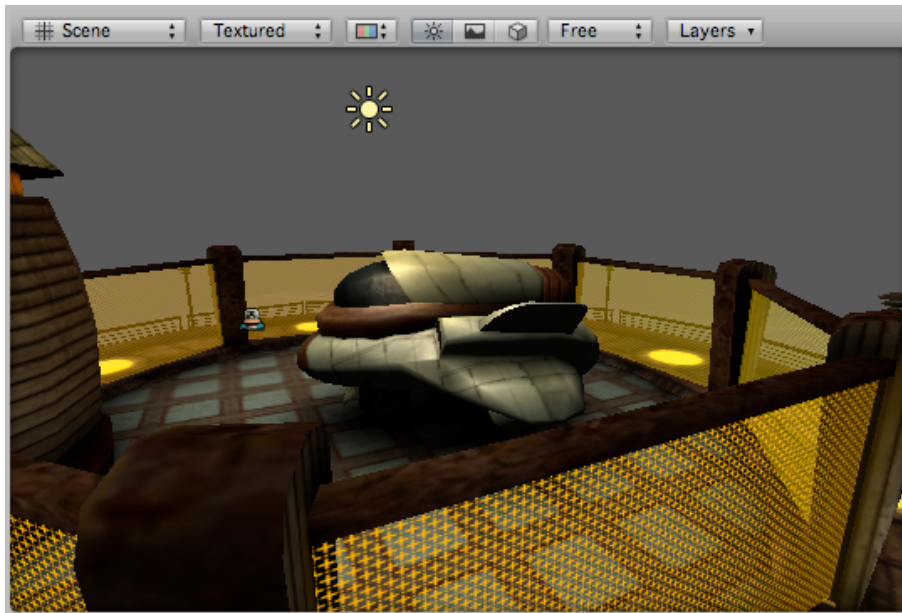
Where they're located is up to you, though they shouldn't be too easy to find or get to. Consider how the player might play the level and decide where the best places are for such pickups. It is best not to be too generous or the game will be too easy.

Finally, we should group these pickups into a folder of some sort to avoid having them clutter up the Hierarchy Pane. We can do this by creating an empty GameObject using the *GameObject->Create Empty* menu item. Rename this new object to Health Pickups and drag your health pickups into this, as shown.



Health pickups hierarchy.

The Force Field



The force field.

At the moment, the force field trapping our hero's spaceship doesn't animate: it's just a static mesh texture. The result is visually disappointing.

There are a number of ways to achieve a decent visual effect, but which to choose? Sometimes a simple solution is the best: we will just animate the texture's UV coordinates to give it the effect of a rippling force field.

The animation will be done using a short script which can be found in the Assets pane. It is named, "Fence Texture Offset" and looks like this:

```
var scrollSpeed = 8.0;

function Update()
{
    var offset = Time.time * scrollSpeed;
    renderer.material.mainTextureOffset = Vector2 (offset,offset);
}
```

The first line exposes a property we can edit directly in the Unity interface, **Scroll Speed**. The `Update()` function is called every game cycle by Unity. We use a short formula -- multiply the Scroll Speed value by the current time -- to define a texture offset.

Pretty Properties

When Unity's Inspector displays properties and variables, their names are adjusted to make them look nicer. Usually, this just means looking for the capital letters in the name and inserting a space before each one. In addition, Unity capitalizes the first letter of the name. Thus **scrollSpeed** is displayed as **Scroll Speed**.

When a texture is rendered, the texture itself is usually just an image. The **mainTextureOffset** property of a material tells Unity that it is to draw the texture's image offset by the specified number of pixels. This can be used to produce some very effective results without resorting to complex animation sequences.

Expand the **levelGeometryNew** GameObject to reveal the different elements of the level data. We need to animate the fence on the **impoundFence** object, so drop the **fenceTextureOffset** script onto this. Unity will warn that this will break the connection with the prefab. We want this, so click Continue.

We can now update the Prefab to include the animation by selecting the **levelGeometryNew** container, then choosing the Apply Changes to Prefab item from the GameObject menu. (This can take a few moments.) When the process is completed, the **levelGeometryNew** name should be blue again, as should all its children.




Scripting the Collectable Items

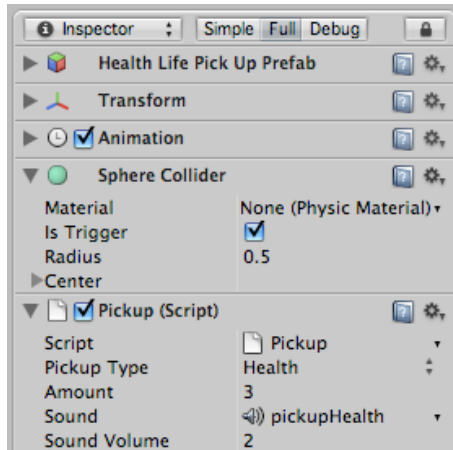
At the moment Lerpz does not pick up any of the items on the level. This is because Unity has not been told to let our hero do this. We need to add two elements to each collectable item:

- A Collider Component,
- Script code to handle the Collider and update player health, etc.

The collectible items in the Hierarchy Pane are all Prefab instances, which are displayed in blue. By editing the original Prefabs directly, we will automatically update all the items in the game.

The two prefabs our hero can collect are **fuelCellPrefab** and **healthPickUpPrefab**. These can be found inside the **Props** folder in the Project Pane.

-  Select the root **healthPickUpPrefab** object.
-  Use **Component->Physics-> Add Sphere Collider** to add a sphere collider to the Prefab. You should see it appear in the Inspector.
-  Finally, Set the **Is Trigger** checkbox.



The HealthPickUpPrefab in the Inspector.

Colliders have two uses: we can hit them with something else, or we can use them as Triggers.

Triggers

Triggers are invisible Components which, as their name implies, trigger an event. In Unity, a Trigger is simply a Collider with its **Is Trigger** property set. This means when something collides with the Trigger, it will act like a virtual switch instead of a physical entity.

Triggers will send one of three event messages when something sets them off: `OnTriggerEnter()`, `OnTriggerStay()` and `OnTriggerExit()`.

Trigger event messages are sent to any script attached to the trigger object, so now we need to add a suitable script to our health prefab:

- Go to the Components menu and choose the Pickup script from the Third Person Props sub-menu. This will add the Pickup script to our Prefab.
- Set the **Pickup Type** property in the Inspector to **Health** as shown in image 2.3.
- Finally, set the Amount property to 3 or so. This is the amount of health the pickup bestows on the player.

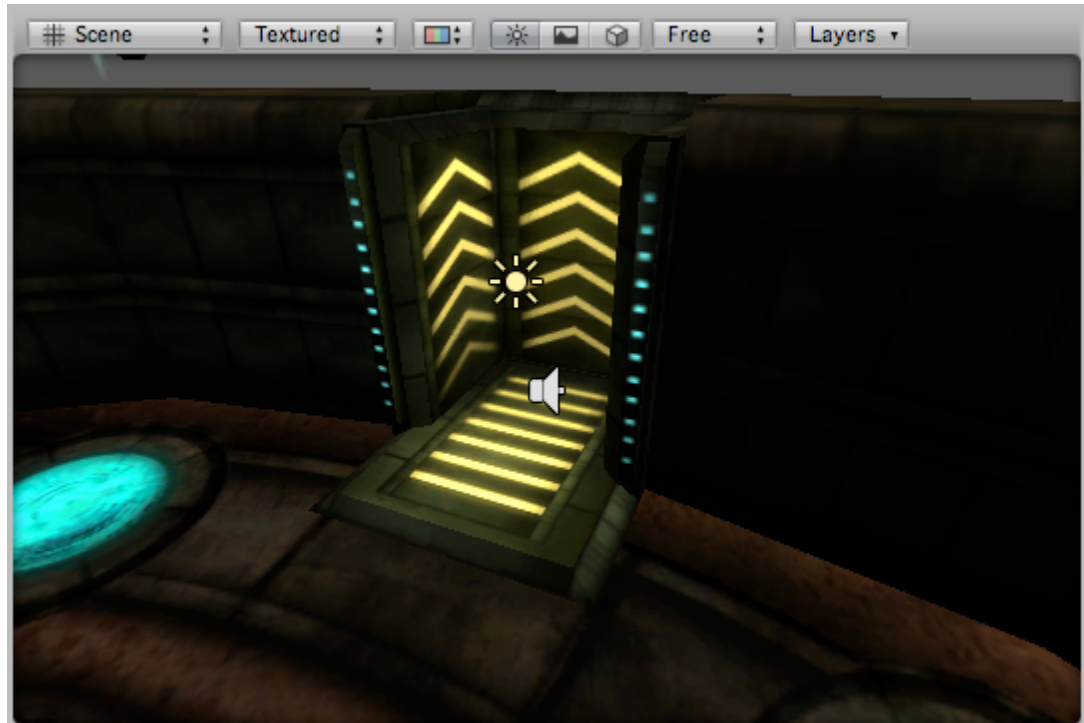
How much health?

The heads-up display, or 'HUD', which shows the player's current health level, lives, etc., can only handle a maximum health level of six. What happens if the player collects a health pickup when he already has a full health bar? This is a matter of taste, but I've chosen to make this trigger the addition of an extra life. The logic for this can be found in the player's state checking script, **ThirdPersonStatus**.

The fuel cell pickups are set in much the same way, with the only two differences:

- The **Pickup Type** setting should be **FuelCell**,
- The **Amount** value, which is the amount of fuel the pickup represents. (1 seems best.)

Jump Pads



A Jump Pad.

The Jump Pads are the bright yellow and black striped spaces in our level. These are supposed to boost Lerpz into the air. We shall use a collider with an attached script for this purpose.

First, create an empty GameObject and call it **Jump Pad Triggers**. We'll use this like a folder to keep our Jump Pad trigger objects together.

Now we'll build our Prefab:

- ❖ Create a new empty GameObject.
- ❖ Rename this object to **JumpPad Trigger 1**.
- ❖ Add a Collider object to it. (Either a Sphere Collider or Box Collider will work.)
- ❖ Set the Collider as a Trigger.

- Add the **JumpPad** script.
- Add the **jumpPad** sound effect. (This appears as an AudioSource automatically.)
- Disable the **Start on Wake** feature of the sound effect.

That's the object created. Now we need to turn it into a Prefab:

- Choose **Prefab** from the **Create** menu above the Project pane.
- Drag and drop our Jump Pad game object onto the new Prefab.
- Rename the Prefab to **JumpPad Trigger**.
- Delete the original GameObject from our Hierarchy pane.
- Drag a **JumpPad** Prefab into the scene and position it directly inside one of the Jump Pad locations. (There are six to place. I recommend using the 4 Split view layout to help with positioning.)
- Finally, hit Play and test the game to make sure all our new triggers work correctly.

NOTE *Scripts work similarly to Prefabs: we've just added a link to the **Pickup** script in the Props folder -- to our Prefab. Editing the one in our Project pane will also affect any copies in the Scene.*

Good organization is important if you want your workflow to be smooth and hassle free.

- Use instantiated Prefabs wherever possible.
- Try organizing by function instead of type.
- Use Empty GameObjects as containers.

You will be surprised at how many assets are needed for even a small-scale project.

The GUI

How many lives do we have remaining? What is Lerpz's health level? It's time for a GUI.



The User Interface

Games usually have Graphical User Interfaces (GUIs), such as menus, options screens and so on. Furthermore, games often have a GUI overlaid on top of the game itself. This could be as simple as a score displayed in a corner, or a more elaborate design involving icons, inventory displays and health status bars.

Unity 2.0 introduces a new GUI system to make it easy to build such GUIs for games and this is the system we shall use for Lerpz Escapes.

NOTE *The old system remains in place for backwards compatibility, but it is considered obsolete and will be removed in a future version of Unity.*

Unity 2.0's new GUI system

Previously, you would tell Unity to draw a button and Unity would fire off relevant messages to your script when the user hovered over the button, clicked the button on it, released the button, and so on.

The old system was based on the traditional Event-Driven GUI model, but Unity 2.0 introduces a brand new GUI system, known as an *Immediate Mode GUI*. If you are used to traditional GUI systems, the Immediate Mode GUI concept may come as a shock.

Here's an example:

```
function OnGUI()  
{  
    If (GUI.Button (Rect(50, 50, 100, 20), "Start Game") )  
        Application.LoadLevel("FirstLevel"); // load the level.  
}
```

`OnGUI()` is called at least twice every game cycle. In the first call, Unity builds the GUI and draws it. In this case, we get a simple button drawn at the coordinates specified, with "Start Game" displayed within.

The second call is when user input is processed. If the user clicks on the button, the `If(...)` conditional surrounding the button-drawing function returns true, so `Application.LoadLevel()` will be called.

Other GUI elements -- Labels, Groups, Check-boxes, etc. -- all work similarly, with the functions returning true / false, or user input as appropriate.

The obvious advantage here is that you don't need umpteen event handlers for a GUI. It's all contained in the one `OnGUI()` function.

Unity 2.0 provides two sets of Immediate Mode GUI functions: the basic GUI class as used in the example above, and a similar `GUILayout` class, which handles the layout of GUI elements for you to save time.

Further Information

More information on the new Unity GUI system can be found by following these links:

- <http://unity3d.com/support/documentation/Components/GUI%20Scripting%20Guide.html>
- <http://unity3d.com/support/documentation/ScriptReference/GUI.html>

The In-game HUD

Our game needs an in-game GUI to display the player's health, lives remaining and the number of fuel cells he needs to collect. The graphical elements are already included in our project file.

The GUI is handled within the **GameHUD** script, which uses the new GUI component to lay out the various elements. This script needs to be attached to a **Level** GameObject,

which is used to hold Scene-specific elements. (We could just as easily have added it to the Main Camera object or to its own 'GUI' GameObject; this is mainly a matter of personal taste rather than a key game design decision.)

➤ Create an empty GameObject.

➤ Name the object **Level**.

We will use this object to manage level-specific states and other scripts.

The GUI Skin object.

Unity 2.0's new GUI system includes support for *skinning*. This gives you full control over the look and feel of every GUI element. Building your own GUI skin content lets you change the shape of a button, its imagery, its font, its colors and do the same to every other GUI element, from text input boxes through to scroll bars and even whole windows.

As our in-game GUI will be based entirely around graphical images, we will build the game's HUD entirely using the `GUI.Label()` function. However, we do need to use a custom font for our HUD, in order to display the remaining fuel cans and lives.

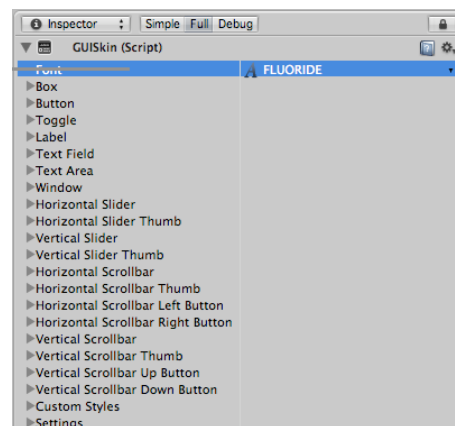
The **GUISkin** asset defines the 'look' of a Unity GUI, much as a CSS file defines the look of a website. The object is required if you need to change any default features. As we are changing the font, we need to include a **GUISkin** in our Scene.

➤ Use the **Assets** menu command to create a new GUI Skin object. This will appear in the Project Pane and contains the default Unity GUI skin data.

➤ Rename the new GUI Skin object to **LerpzTutorialSkin**.

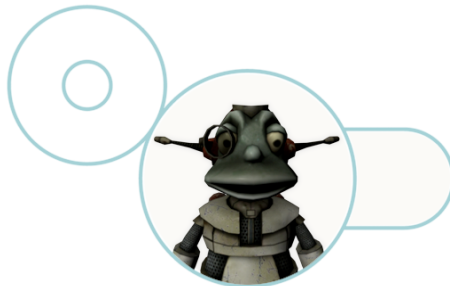
We are going to use a decorative font, named "Flouride", for our game. This is the only change we are making to the default skin.

➤ Drag the Flouride font object onto our new GUI Skin asset's "Font" entry:



GUI Skin, setting the font.

The GUI Skin object is not added to the Hierarchy Pane view; instead we reference the GUI Skin directly in our **GameHUD** script. Along with the GUI Skin, the **GameHUD** script also needs to be told which assets to use to build the GUI display. These include the **GUIHealthRing** asset.



The GUIHealthRing image

This image is used to display Lerpz's health information. The space to the right of Lerpz's image displays his remaining lives, while the circle to the left is used to show a pie chart of his remaining health. The pie chart is created by simply super-imposing the correct image from an array of six 2D textures, named **healthPie1** through **healthPie6**. The **healthPie5** image is shown below:



The healthPie5 image

NOTE These images include alpha channels to define transparency and translucency.

Using separate images lets us simply draw the image corresponding to Lerpz's actual health, instead of performing fancy calculations to rotate and draw segments programmatically.

The second major GUI element is for the Fuel Cell state, the main image for which is **GUIFuelCell**:



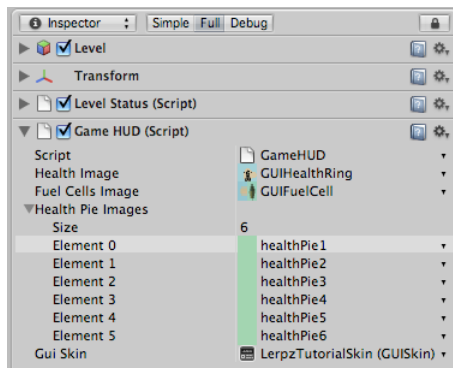
The GUIFuelCell image

This is displayed in the lower-right of the game screen and will show the fuel cells remaining to be collected before the level is unlocked.

- ✎ Add the **GameHUD** script to the **Level** GameObject.
- ✎ Select the Scene object and look at the Inspector and take a look at the Game HUD (Script) component entry.
- ✎ Add the GUIHealthRing and GUIFuelCell images to the GameHUD script.
- ✎ Open up the Health Pie Images entry.

Health Pie Images is an array. At the moment, Unity doesn't know how big it should be, so it has set it to zero. We have six health pie images to drop into this array, so we need to change this value.

- ✎ Click on the "o" next to Size. Change it to "6". You should now see six empty elements, named Element 0 through Element 5.
- ✎ Open the GUI assets folder in the Project Pane to reveal the health pie images. There are six of these, numbered 1 to 6.
- ✎ Computers count from zero, so **healthPie1** needs to go into Element 0. **healthPie2** needs to go into Element 1... and so on. Drag the images into their relevant slots.
- ✎ Finally add the **LerpzTutorialSkin** GUI Skin into the empty **Gui Skin** slot. The settings show now look like this:



GameHUD script settings.

If you run the game now, you should see the HUD appearing over the play area:



The in-game HUD

Resolution Independence.

One problem with the GUI is its size. The screenshot above is from a 24" iMac running at a resolution of 1920 x 1200.

Clearly we need to scale our HUD dynamically according to the current display size and resolution, so how do we achieve this?

Unity 2's new GUI system includes support for a transform matrix. This matrix is applied to all GUI elements prior to rendering, so they can be transformed, rotated or scaled -- in any combination -- dynamically.

The line below, from the GameHUD script, shows how:

```
GUI.matrix = Matrix4x4.TRS (Vector3.zero, Quaternion.identity, Vector3  
(Screen.width / 1920.0, Screen.height / 1200.0, 1));
```

If you go to the Game View, disable the "Maximize on Play" option and set the aspect ratio to 4:3, you will see that the GUI re-scales to fit.

If we wished, we could have our HUD spin around, flip upside down or zoom in from a distance. For game menus, high score screens and the like, this is a useful feature to have and we'll use this trick for our level-complete sequence.

The Start Menu

Every game needs a start menu. This is displayed when the game starts and lets the player change options, load a saved game and, most importantly, start playing the game. In this section, we will build a start menu from scratch.

NOTE *Splash screens, menus and the like are all just Unity Scenes, so a Scene is not always be a game level. We use scripts in one Scene to load and run other Scenes to link Scenes together.*

For the Start Menu, we will need:

- Two GUI text buttons: "Play" and "Quit".
- The name of the game. This will be rendered using a custom font.
- Some suitable music.
- A backdrop of some sort.

In other words, something like this:



The Start Menu.

Setting the Scene

The first step is to create a new, empty Scene.

🖱️ Type CMD+N to create one.

- 🔗 Name it **StartMenu**. Unity will automatically add a Camera to the Scene for us, but there is nothing for it to see at the moment.

Now we'll use the new GUI system to build a menu:

- 🔗 Go to the Project Pane and create a blank JavaScript file.
- 🔗 Rename it **StartMenuGUI** and open it in Unitron.

Before we get stuck in, we'll add a *Unity script directive*. Directives are commands which give Unity information or additional instructions about the script. These commands aren't part of Javascript as such, but aimed at Unity itself.

In this case, we want Unity to run our script inside the Editor, so that we can see the results immediately without having to stop and re-run the project each time:

```
// Make the script also execute in edit mode
@script ExecuteInEditMode()
```

We need a link to the **LerpzTutorialSkin** asset, so the first line of code will be this:

```
var gSkin : GUISkin;
```

We'll need a `Texture2D` object for the backdrop. (We'll drop our background image onto this in the Inspector.)

```
var backdrop : Texture2D; // our backdrop image goes in here.
```

We also want to display a "Loading..." message when the player clicks on the "Play" button, so we'll need a flag to handle this:

```
private var isLoading = false; // if true, we'll display the "Loading..." message.
```

Finally, we get to the `OnGUI` function itself:

```
function OnGUI()
{
    if (gSkin)
```

```

GUI.skin = gSkin;
else
    Debug.Log("StartMenuGUI: GUI Skin object missing!");

```

The code above checks if we have a link to a valid GUI Skin object. The `Debug.Log()` function spits out an error message if not. (It's a good habit to sanity-check any external links or data in this way as it makes debugging much easier.)

The Backdrop.

The backdrop image is a `GUI.Label` element set to use our background image as the element's background. It has no text and is always set to the size of our display, so it fills the screen.

```

var backgroundStyle : GUIStyle = new GUIStyle();
backgroundStyle.normal.background = backdrop;
GUI.Label ( Rect( (Screen.width - (Screen.height * 2)) * 0.75, 0, Screen.height * 2,
Screen.height), "", backgroundStyle);

```

First, we define a new `GUIStyle` object, which we'll use to override the default GUI Skin style. In this instance, we're just changing the "normal.background" style element to use our backdrop image.

The `GUI.Label()` function takes a **Rect** object. This rectangle's dimensions are derived from the display's dimensions, so that the image always fills the screen. The image's aspect ratio is also taken into account, so that the image is cropped and/or rescaled to fit without adding distortion.

Next we come to the title text:

```

GUI.Label ( Rect( (Screen.width/2)-197, 50, 400, 100), "Lerpz Escapes",
"mainMenuTitle");

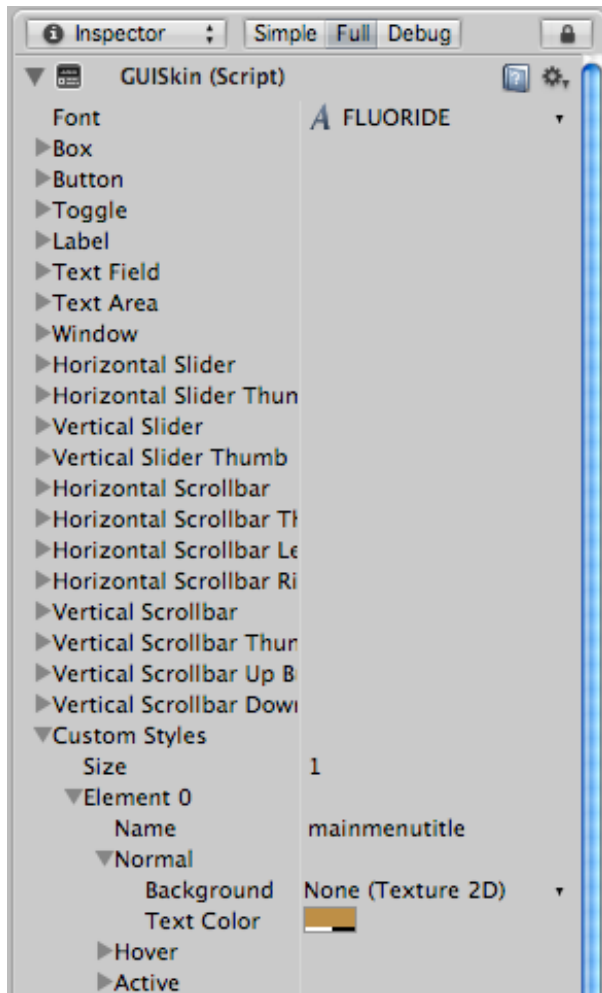
```

In this case, we're using the default GUI Skin's font for the text. However, note that we're using the "mainMenuTitle" style override for this. This is a custom GUI style which we'll define in **LerpzTutorialSkin** now:



Go to the Project Pane and click on **LerpzTutorialSkin** to bring up its details in the Inspector.

We will now add a Custom Style to our GUI Skin:



Defining a custom style in our GUISkin object.

- 🔧 Open up "Custom Styles", change "Size" to "1" and you'll see "Element 0" appear.
- 🔧 Open "Element 0" and set its elements as shown above. Specifically: set the Text Color to the orange-brown shown. (Don't worry about the elements not shown in the screenshot: they can be left as they are.)

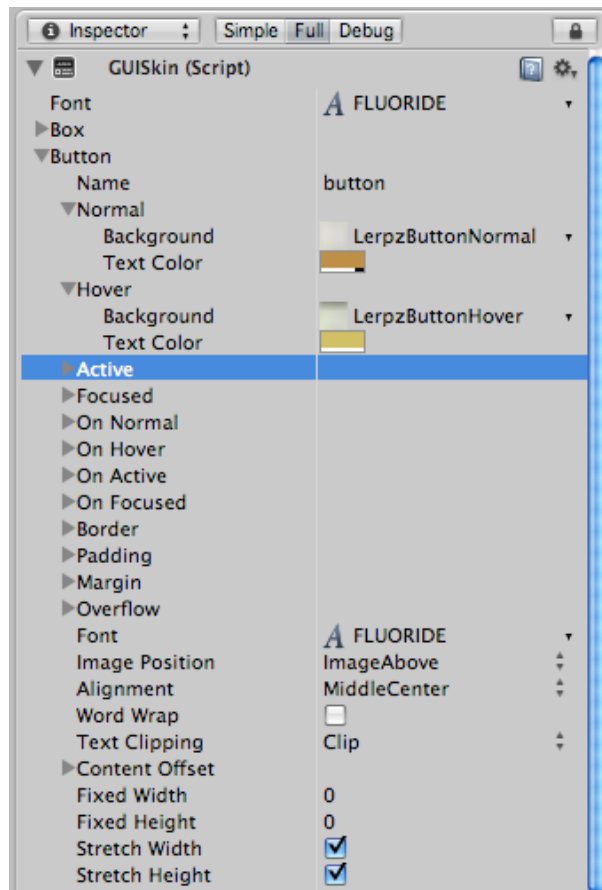
The Buttons.

We now need to modify the GUI Button properties of the **LerpzTutorialSkin** object to produce a more interesting button.

We're using the same GUI Skin object for this menu and for the in-game Heads-Up Display, or 'HUD'. For the HUD, only the default font needs to be changed. However, for the Start Menu buttons, we also need to have a graphical image behind the button text. The default button design doesn't fit the game's visual style, so we need to change it.

TIP If you want your GUI elements to react to *Hover*, *Focus* and *Active* events, you must set a background image too, even if the image is blank.

Click on the **LerpzTutorialSkin** asset in the Project Pane to bring up its details in the Inspector Pane. Change it to the settings shown below – ignore the other GUI element types; we won't be using them:



Setting the button images in the LerpzTutorialSkin GUISkin object.

Now let's add the "Play" button:

```
if (GUI.Button( Rect( (Screen.width/2)-70, Screen.height - 160, 140, 70),  
"Play"))  
{  
    isLoading = true;  
    Application.LoadLevel("TheGame"); // load the game level.  
}
```

The above is all it takes to render and handle the "Play" button. The code for both the rendering and the event handling is in the same place, making it easier to maintain. The `GUI.Button()` function takes a `Rect` object to define the button's position and size, followed by the text label.

If the user clicks on this button the button function returns true, so we can load the game level.

We set **isLoading** so that we know to show the "Loading..." text, then tell Unity to load the game level.

TIP *The new GUI system supports a number of alternative functions for drawing elements, allowing you to specify an image instead of text, or even an image, a label and a tooltip combined. See the documentation for more details.*

The "Quit" button is handled similarly, but with a test to ensure this is running as a standalone build (or in the Unity editor) added.

```
// We only display Quit button if standalone player or in editor:
var isWebPlayer = (Application.platform == RuntimePlatform.OSXWebPlayer ||
Application.platform == RuntimePlatform.WindowsWebPlayer);
if (!isWebPlayer)
{
    if (GUI.Button( Rect( (Screen.width/2)-70, Screen.height - 80, 140, 70), "Quit"))
        Application.Quit();// quit back to desktop. NOTE: Does nothing in Unity
editor!
}
```

As you can see, this is very similar to the "Play" button. We just draw it a little lower down and check if we need to draw it at all first.

The final step is the "Loading..." text, which needs to be displayed when the user selects the "Play" button. This is because it can take a few moments for our game Scene to load -- especially if it's being streamed over the Internet inside the web player.

This is where **isLoading** comes in:

```
if (isLoading)
    GUI.Label ( Rect( (Screen.width/2)-110, (Screen.height / 2) - 60, 400, 70),
"Loading...", "mainMenuTitle");
}
```

Again, we make use of the "mainMenuTitle" Custom GUI style so that the text style matches that of the title.


The Quit Button.


The "Quit" button will only work if you're running the game as a standalone app. If it's being played in a web-player, Dashboard widget or inside the Unity Editor itself, the "Quit" button makes no sense. The web-player cannot 'quit' as such; it's embedded in a web page. Nor is it a good idea for it to close the browser. You could argue that closing the page the web-player is running on might be an option, but this is best handled by the web page itself. Similarly, what could a "Quit" option do in a Dashboard widget? Closing the widget would delete it from the Dashboard.

So, we need to disable the Quit button and stop it displaying if the game isn't running as a standalone. The one exception is if the game is being run inside the Unity Editor itself. This is because we want to know if the button is being displayed in the right place and behaving itself, which is difficult if you can't see it.

Once the **StartMenu** Scene has loaded, it will call the function above automatically and set **isStandalone** accordingly. If it's set to true, we'll display the "Quit" button, otherwise we won't.

The final touch is to add some music.

 Go to the Project Pane, open up the Sounds folder and drag the **StartMenu** audio file onto the Main Camera object in the Hierarchy Pane. This will add an Audio Clip component.

 Click on the Main Camera object now and, in the Inspector, locate the new Audio Clip component. Tick the **Play On Awake** and **Loop** boxes.

TIP *The music was created using Apple Loops from Apple's Orchestral Jam Pack, arranged in Garageband. This is a handy tool for building place-holder tunes; you can use these to decide which musical style best fits your game.*

If you now play the scene, you should see something like this in the Game View, accompanied by a short, looping orchestral tune. So that's it! Our Start Menu is done.

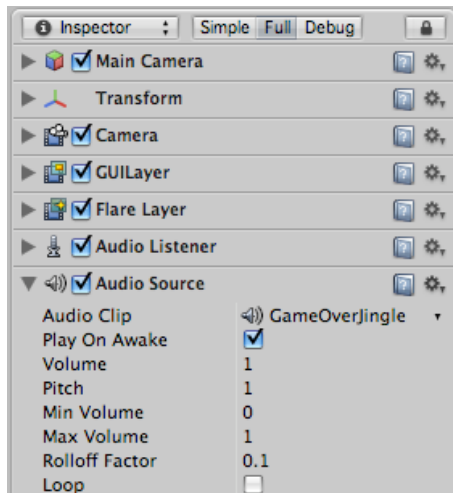
For your own projects, you will most likely want to add an Options Menu, perhaps a high score menu, a multiplayer lobby, etc. These can all be built using the Unity 2 GUI.

Game Over

The Game Over screen is shown when the player has either completed the game or has failed the challenge. Unlike the Start Menu, this Scene has no buttons or other visual user interaction: it just shows a "Game Over" message over a backdrop while playing a short jingle. Once the jingle is completed, or if the user clicks the mouse, we automatically load the "Start Menu" Scene.

 Firstly, create a new Scene and name it "GameOver"

- ✎ Drag and drop the **GameOverJingle** audio file onto the default **Main Camera** object and set it as shown:



The GameOverJingle settings.

We don't need to add anything else to the Scene using the Editor: the default camera alone will suffice.

The next step is to build our script:

- ✎ Create a new Javascript script asset and name it "GameOverGUI"
- ✎ Open it in the Unitron editor and add the code described below:

As with the Start Menu, we want to be able to see our GUI in the Unity editor even when the project isn't running, so add this:

```
@script ExecuteInEditMode()
```

For the Start Menu, we used the LerpzTutorialSkin GUI Skin asset. The GUI Skin defines a bunch of GUI Styles and lets us apply them to a GUI wholesale.

An alternative technique is to define individual GUI Style objects directly. We will do so for the Game Over script by defining three GUIStyle variables which we can then set in the Inspector, along with two variables defining the scaling of the text elements:

```
var background : GUIStyle;  
var gameOverText : GUIStyle;  
var gameOverShadow : GUIStyle;
```

We'll set these GUI Style objects in the Inspector shortly.

Next, we need to define the text scaling factor for our “Game Over” message as it will be rendered larger than the default font size.

We’re going to draw this message twice, in two different colors, to give a shadowed text effect, so we’ll define two scale variables:

```
var gameOverScale = 1.5;  
var gameOverShadowScale = 1.5;
```

At last we get to the `OnGUI()` function:

```
function OnGUI()  
{
```

First, the backdrop, rescaled similarly to that used in the Start Menu:

```
GUI.Label ( Rect( (Screen.width - (Screen.height * 2)) * 0.75, 0, Screen.height * 2,  
Screen.height), "", background);
```

The next task is to draw the shadowed version of the “Game Over” message. We need to scale the text up and render it centered on the screen. Luckily, we can use the GUI system’s built-in transform matrix to handle the scaling for us.

TIP *The GUI transform matrix can also be used to perform any arbitrary translations you wish: you can scale, rotate, flip and spin to your heart’s content.*

To ensure the text appears in a dark, shadow color, we pass the **gameOverShadow** GUI Style to the `GUI.Label` function.

```
GUI.matrix = Matrix4x4.TRS(Vector3(0, 0, 0), Quaternion.identity, Vector3.one *  
gameOverShadowScale);  
GUI.Label ( Rect( (Screen.width / (2 * gameOverShadowScale)) - 150,  
(Screen.height / (2 * gameOverShadowScale)) - 40, 300, 100), "Game Over",  
gameOverShadow);
```

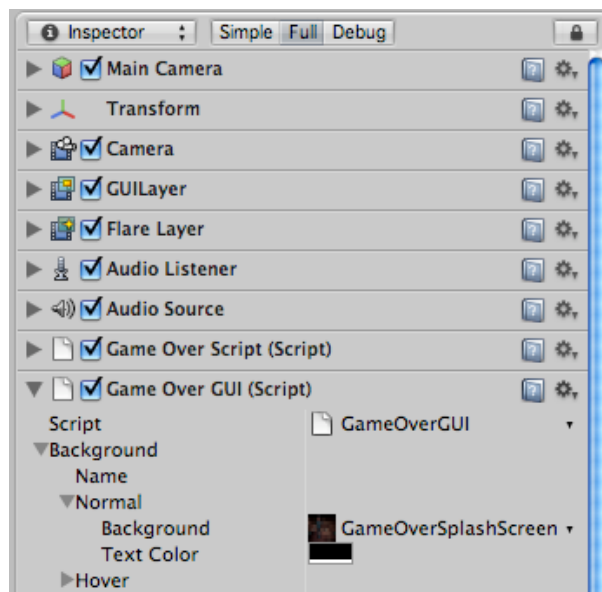
Finally, we draw the same text again, but in a lighter color. Unity’s GUI system will always render these elements in the order they appear in the code, so this text will appear on top of the shadow. Aside from using the `gameOverScale` scaling factor and the `gameOverText` GUI Style, there’s no other difference.

```

GUI.matrix = Matrix4x4.TRS(Vector3(0, 0, 0), Quaternion.identity, Vector3.one *
gameOverScale);
GUI.Label ( Rect( (Screen.width / (2 * gameOverScale)) - 150, (Screen.height / (2 *
gameOverScale)) - 40, 300, 100), "Game Over", gameOverText);
}

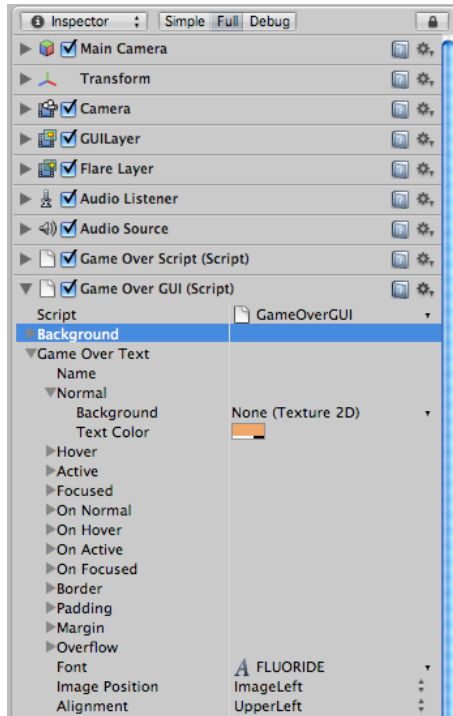
```

- ✎ Save the script and drop onto the Main Camera.
- ✎ Click the “Main Camera” object to bring it up in the Inspector. It’s time to set the variables...
- ✎ First, the Background GUI Style. This just needs the GameOverSplashScreen image dropped into the *Normal->Background* slot, as shown below:



The pertinent Background GUI Style settings.

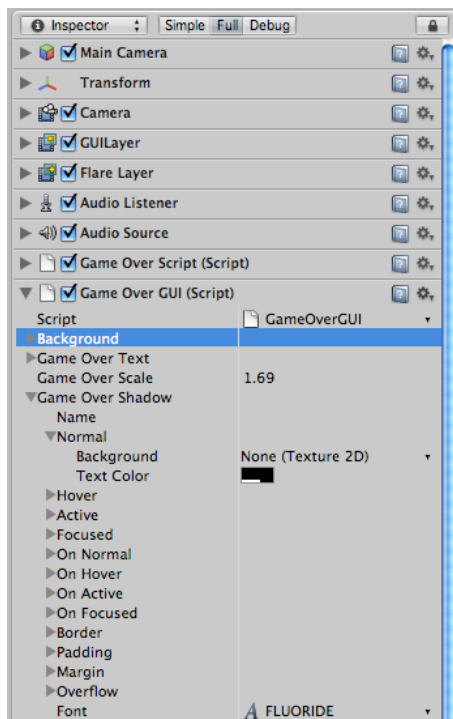
- ✎ Next, we’ll set the Game Over Text GUI Style as shown in the next image. (As usual, leave any other settings as they are.)



The pertinent Game Over Text GUI Style settings.

Next, set the Game Over Scale to 1.69.

Now for the Game Over Shadow GUI settings:



The pertinent Game Over Shadow GUI Style settings.

Finally, set the Game Over Shadow Scale to 1.61.

You should now see the GUI appear in the Game View, as shown below:



The Game Over screen.

The final touch is to add a second script to the Main Camera which checks if the music has finished playing and, if so, loads up the Start Menu Scene.

Create a new Javascript Script asset. Name it **GameOverScript**.

Open the script in Unitron and add the following code:

```
function Update ()
{
    if (!audio.isPlaying || Input.anyKeyDown)
        Application.LoadLevel("StartMenu");
}
```

This code checks if the audio has finished playing, or whether the player has pressed a key before loading the "StartMenu" Scene.

And that's it: GUIs done!

Adversaries

No game is complete without adversaries. In this chapter, we add enemies for Lerpz to fight.



Antagonists & Conflict

These two elements are key to any game, so we need something to keep Lerpz on his toes. The job of the game designer is to throw obstacles into the path of the player, but make them surmountable.

Lerpz faces two adversaries: robot guards and laser barriers.

The Laser Traps

The Laser Traps are located in the laser passages and will harm our player if he should touch the beam. The image below shows two of these. We will locate them in the short passageway structures on either side of the arena, at the far end of the level from the jail.



Lerpz faces two Laser Traps.

The lasers rise and fall. If the player tries to pass through one of these, he will lose some health.

Implementing the Laser Traps

Each laser trap is just a beam which rises and falls in a same vertical plane. If Lerpz (or an enemy) should happen to hit the beam, it'll cause damage; otherwise, they can pass harmlessly though.

The laser beam itself is produced by a **Line Renderer** component contained in its own **GameObject**. Its movement and the logic which controls it is entirely contained in the **LaserTrap** script. So let's build our first Laser Trap:

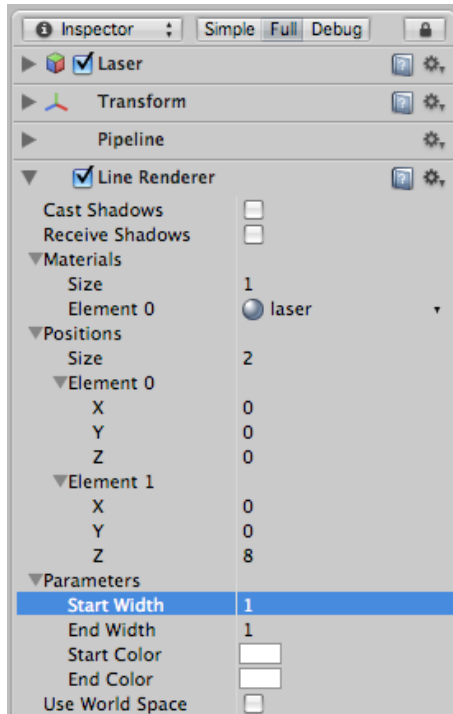
Create an empty **GameObject**.

Rename it to "Laser"

Add a **Line Renderer** Component (*Component->Miscellaneous->Line Renderer*).

Add the **LaserTrap** script.

Adjust the **Line Renderer** Component settings as shown:

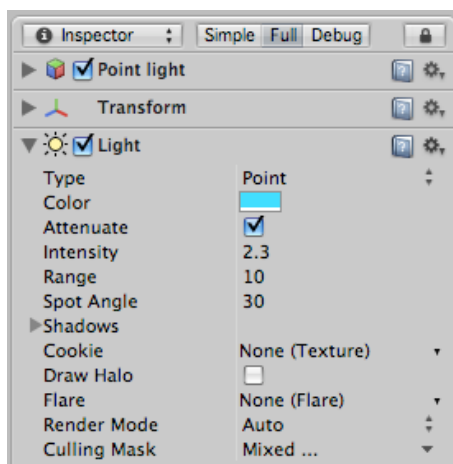


Line Renderer settings.

Position the resulting object in the laser tunnels. (These are the roofed corridor structures on either side of the arena, at the far end of the level from the Jail.)

Add a Point Light GameObject as a child to our Laser object.

Set the Point Light as shown:



Laser Trap Point Light settings.

The Point Light acts as the laser's light source and will rise and fall with the laser itself. This gives the illusion that the laser beam, drawn by the Line Renderer, is emitting light.

The Line Renderer Component

The Line Renderer, as its name suggests, draws lines in 3D space within our Scene. It contains an array which defines the series of points through which the line will be drawn. The line itself is drawn using the same rendering technique as the Trail Renderer component, making it ideal for lasers, lightning and so on.

The Laser Trap Script

The key to the Laser Trap is the **LaserTrap** script, so let's take a closer look...

Overview

The Laser Trap is a Line Renderer component which is moved up and down by the script. This same script also checks for collisions and triggers the requisite visual effects should one occur.

First, we define some basic properties for our laser trap:

- **height** defines the amplitude of the oscillation, dictating how far above and below its starting point the laser beam will travel;
- **speed** defines how fast the beam moves;
- **timingOffset** allows us to set each laser trap object to start at a different point in its oscillation cycle;
- **laserWidth** defines the width of the laser beam from end to end;
- **damage** defines how much damage the player will suffer if he runs into the beam.
- **hitEffect**, can be used to link to an arbitrary GameObject, which will be instantiated when something hits the laser trap. This is more flexible than hard-coding the effect in the Laser Trap object and makes it easier to re-use this asset in other projects.

The script defines two functions, so let's look at these in turn:

The `Start()` function initializes the Line Renderer component, storing its initial location (in the Y axis) for later, and setting the Line Renderer's second vertex to match the position defined by `laserWidth`. This lets us easily tweak the length of the beam to match the width of the corridor.

NOTE *We could just set the end coordinate of each line by hand in the Line Renderer's array, but handling this in the script gives us a bit more flexibility should we decide to make the laser's beam animate in-game.*

Now we come to the main `Update()` function. This is where the interesting stuff happens.

First, we have the laser beam movement to calculate. We just use the `Mathf.Sin()` function for this. We grab the current time using `Time.time`, (which returns the time, in seconds, since the game started), multiply it by our desired animation speed and add in the **timingOffset** value. This gives us our position along the sine curve. Finally, we modulate it by our desired **height** and use the result as an offset from our baseline.

The next step is to check for collisions. The script does this by casting a ray along the path of the beam and checking if it hits any `GameObject` which have `Collider` Components attached to them.

(The time-based test is there to allow time for a reaction. If we didn't do this, the player would lose a health point for every frame in which he's hit by the ray-cast.)

If we do hit something, we check if it's a player or enemy and, if so, send the "Apply-Damage" message to it. At the same time, we also instantiate the **hitEffect** `GameObject`, so that it can perform its magic. This is the **LaserHit** `GameObject`, which produces an energy burst effect:



Being hit by a laser is bad for your health.

If you play the tutorial, you should find that Lerpz loses a health point if he hits the laser beam.

Shock & Awe

At the moment, Lerpz doesn't react much to a hit from a laser. In the next part of this chapter, we'll look at the robot guards who send Lerpz flying when they hit him. Try adding the same action to the lasers.

The Robot Guards

The mobile antagonists are robot guards, placed strategically around the level. When Lerpz gets within range, these guards will home in on him and attempt to cause him harm.



A robot guard.

Lerpz's enemies in this game are robot guards. This particular model is now considered a 'classic', worthy of preservation. Collectors and enthusiasts compare them, unfavourably, with such other classic gems of yesteryear as the Ford Edsel, Apple Lisa and Sinclair C5.

This particular model is well-known for the way it was cunningly designed to be susceptible to a couple of well-aimed punches at the torso. When so incapacitated, the robots eject any items they are carrying, after which the robot will happily lie on the ground until the BIOS can reboot, although this would only take place when the proximity sensor asserted that there was nothing in the vicinity. These security robots are now used as garden ornaments.

From the above, we can determine that:

- Robots will have a fairly rudimentary AI.
- The robots can be attacked from the sides and from behind, but not from the front.
- The robots eject collectable items when knocked down.
- Robots will re-spawn when the player can't see them.

Seek & Destroy

Most game AI is primarily focused on modeling behavior rather than intelligence as such. Our robots deliberately display little actual intelligence, but merely react in predictable ways to the player's presence. This is not necessarily a bad thing: Many game players like this kind of behavior. It gives them a pattern to look for and thus makes it easier the learn how to defeat the robots.

The robot guards therefore have a very simple behavior pattern and this is reflected in the main AI script, **EnemyPoliceGuy**.

Idle – In this mode, the guards just stand there thinking robot thoughts, ticking away quietly and waiting for an intruder to come into range.

Threaten – If an intruder comes within a set distance of the guard, the guard comes out of standby and announces its intentions while spinning its baton in what was advertised in the sales brochure as a “threatening manner”.

Seek & Destroy – Once activated, the robot guards will home in on the intruder and try and attack him. If the robot guard gets within striking range of the intruder, the guard will attempt to hit it with its baton.

NOTE *One animation sequence is used for both modes and is named turnjump in the model and script.*




Struck – If the player strikes the robot guard, this animation is played. (“gothit” in the model and script.)

If the robot moves outside the robot's limited scanning range, it will revert back to “Idle” mode.

The above states are handled by the **EnemyPoliceGuy** script, which also deals with switching between the animation sequences. (There weren’t enough animation sequences to justify splitting the script into two as was done with the player.)

Adding the Robot Guards



We need a few of these robots on our level, so...

-  Open the Project Pane and locate the **CopperNew** Prefab inside the *Enemies* folder.
-  Find a suitable spot on the level and drop the Prefab onto the Scene, ensuring it is on the ground. (The Prefab includes a **Character Controller** Component. Make sure the lower end of the capsule collider is just touching the ground, or slightly above it.)
-  Add a few more robots to the level in this way.



Areas with fences or which are otherwise mainly enclosed are best as this makes it less likely that the robot will fall off the scenery.

If you play the game now, you will see the robots standing around playing their Idle animations. We now need to add some scripts to make these robots do something more interesting.

There are two scripts for the robots and we'll add them directly to the original Prefab:

-  Select the **CopperNew** Prefab to bring it up in the Inspector.
-  Drag the **EnemyDamage** and **EnemyPoliceGuy** scripts onto the **Character Controller** component in the Inspector.

If you now play the game, the robots still don't react to the player. This is because the **EnemyPoliceGuy** script component also requires a link to the Player object, so it knows what to attack.

-  Click on a **CopperNew** Prefab in the Project Pane to bring its details up in the Inspector.
-  Drag the **Player** GameObject onto the Target link in the robot's **Enemy Police Guy (Script)** component slot.

If you play the game now, the robots should react to you if you get too close to them.

Blue Sparks Of Death

If the player manages to beat the robot up and knock him down, a death sequence is initiated. We want the robot to fall over in a shower of sparks and lie there until the player moves out of range before resetting.

In addition, when the robot dies, we want it to spit out any collectables it's carrying.

Rather than adding yet more scripting, animation data and other elements to our current Prefab, we'll create a new one just for the death throes of our electric foe.

Divide & Conquer

When our robot dies, we need it to stop reacting to the player and suspend the scripts. This is not quite as simple as it seems: scripts tend to run independently of each other, so we would need to send out messages and maintain a dedicated state variable in each script which would need to be checked during every cycle.

It is much easier to simply swap out our robot Prefab for another one, built specifically for the purpose of falling over, all broken and sparkly, with some scripting to spit out up to two random pickups. And this is what we'll do.



A robot guard keels over

The image above shows this replacement Prefab in action.

The script which brings it into existence is **EnemyDamage**, so let's take a closer look at it...

When an *ApplyDamage* message is sent to the `GameObject`, the `ApplyDamage()` function is called. If the robot has suffered too many hits – in our example, the hit points are set to 3 – it calls the `Die()` function, which is where the fun begins.

The `Die()` function first marks our **Copper** `GameObject` for destruction. (The destruction is not performed until after the `Update()` function is done.)

Next, the replacement Prefab is instantiated and the robot's current position – including those of all child object from head through torso and arms – is copied over.

The sparkly explosion is instantiated next and made a child of our new dying robot `GameObject`.

The next step, now that we have our instance in place, is to give it a push away from the player so that Unity's physics engine can make it fall over and roll about.

Finally, we instantiate up to two pickups – each has a 50:50 chance of being either a health or fuel cell pickup – and launch them into the air in a random direction.

Droppable Pickups & Physics

The pickups which appear when a robot is knocked over are Prefabs derived from the originals we created earlier, but with added particle effects and a script, **DroppableMover**, which handles movement and collisions.

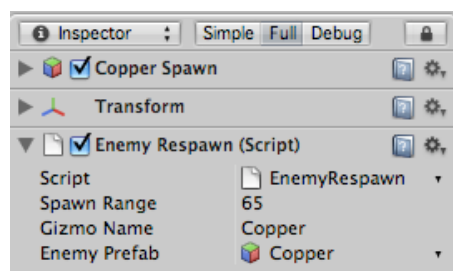
The script is needed because the collider is set as a trigger for the pickup and thus is ignored by the physics engine. The pickup is given an initial velocity and the script then uses raycasting to check if the pickup has hit anything. When it does, the script is simply disabled.

Spawning & Optimization

Spawning enemies when they are within a set distance of the player is an old trick dating right back to the earliest computer and video games; it removed the need to store each enemy's state. We can also use this trick to reduce the load on the processor. By simply deleting each robot if it's not visible to the player, we can avoid running scripts and AI unnecessarily.





Let's begin:

- ✎ Create an empty GameObject at the top level of the Hierarchy Pane.
- ✎ Rename it **CopperSpawn**.
- ✎ Drag the **EnemyRespawn** script onto the object. (This will automatically add the trigger collider Component for us.)
- ✎ Position our **CopperSpawn** object somewhere where you'd like to see a robot appear. (You should see a small robot icon displayed as a gizmo, as well as a sphere to show the spawn range. These are drawn by the **EnemyRespawn** script.)
- ✎ Adjust the object's settings as shown.



CopperSpawn settings.

We'll need a few of these Game Objects, so let's set up a parent GameObject and make our **CopperSpawn** object its children...

-  Create an empty GameObject at the top level of the Hierarchy Pane.
-  Rename this new object **Enemies**.
-  Make our **CopperSpawn** GameObject a child of **Enemies** by moving the former into the latter.
-  Now use copy and paste to duplicate our **CopperSpawn** object, ensuring they're kept under the **Enemies** object. Position these in the level as you see fit.

NOTE *One side-effect of this technique is that, if you have killed a robot and then moved out of range, it will reappear as good as new if you return to its location.*

How it works.

The **CopperSpawn** Game Objects contain a script which checks if the player has come within range and, if so, creates an instance of the **CopperNew** Prefab. When the player walks outside this range, our robot-spawning script will automatically remove the robot from the Scene.

The script which does this is **EnemyRespawn**. This script is heavily commented, but the two main functions are:

`Start()` – just caches a link to the player Game Object's transform as we'll need to mess with it later.

`Update()` – First checks if the player's in range and instantiates the robot if so. If not, and the player has just stepped out of range, the robot prefab is destroyed.

There are also two Gizmo functions used in the Editor.

EnemyRespawn also makes use of Unity's "Gizmos" feature. A Gizmo is usually a visual aid displayed only in the Scene View rather than in-game, such as, in this case, a sphere showing the spawn range. In this script, we have two types of gizmos: the first draws a robot icon, which lets us select the spawn object by clicking on its icon in the Scene with the mouse:

NOTE *The icon image is currently stored in the `/Applications/Unity/Unity.app/Contents/Resources` folder, although this may change in future versions.*



The OnDrawGizmos() function showing the robot icon.

The Gizmo code for the icon is shown below:

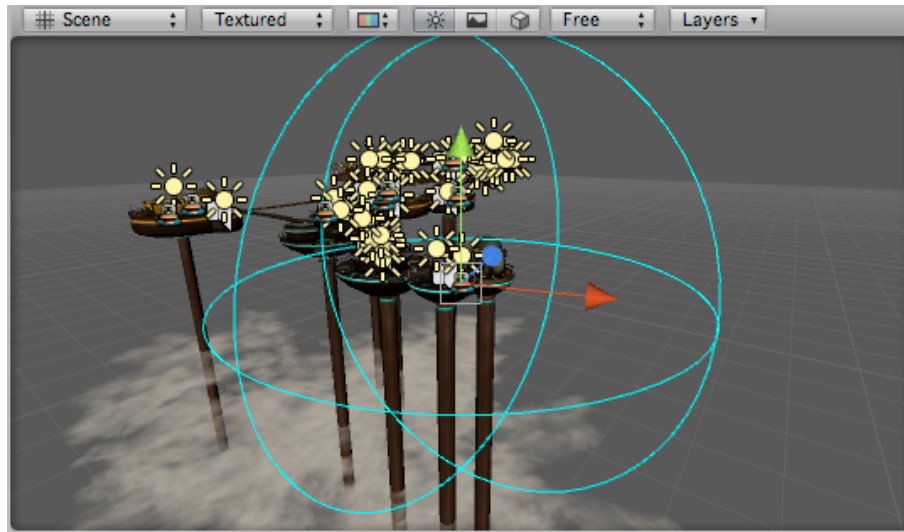
```
function OnDrawGizmos ()
{
    Gizmos.color = Color(1, 1, 1, 1);
    Gizmos.DrawIcon(transform.position, type + ".psd");
}
```

The `OnDrawGizmos ()` function is called every time the Unity editor GUI is updated or refreshed, so the icon will always be visible.

Conversely, the `OnDrawGizmosSelected ()` function is called by Unity's editor GUI only when the object is selected. As long as it is selected, it will be called every time the Unity editor GUI is updated or refreshed.

In this example, the function draws a sphere using **spawnRange** to define its radius, thus providing a visual display of the area within which the enemy robot will be instantiated; when the player moves outside this sphere, the robot will be automatically destroyed.

```
function OnDrawGizmosSelected ()
{
    Gizmos.color = Color(0, 1, 1);
    Gizmos.DrawWireSphere(transform.position, spawnRange);
}
```

The `OnDrawGizmosSelected()` function showing the sphere defined by the `Spawn Range` variable.

Alternative Optimizations

In addition to the above technique, Unity also offers the `OnBecameVisible()` and `OnBecameInvisible()` functions. However, unlike our respawning technique, the above functions are based on the camera's orientation and other settings rather than those of the player object. This means you will see `OnBecameInvisible()` called on an object just because the camera has turned away from it. This may not be what you require.

Another technique, even more optimal than our own, is to use Collider components as triggers instead of using script code to check for the player's location. Unity provides the `OnTriggerEnter()` and `OnTriggerExit()` functions for this purpose. However, this might not be feasible if you want your respawn scripts to be attached to an object that needs to use a collider for other purposes.

Audio & Finishing Touches

In this chapter, we add sound effects and two cut-scenes to our game.



Introduction

When you buy a CD or watch a movie, the sound you hear has invariably been through a number of stages, the last of which is known as mastering. Mastering is the art of listening to the audio as a whole and adjusting volume levels, filtering out frequencies and any number of other technical tricks to make the final mix sound as good as possible. Sometimes the process fixes obvious flaws in a musical number or soundtrack, such as an instrument being too loud or too harsh. At other times, the process is just used to ensure the soundtrack will sound good enough on a cheap TV loudspeaker without compromising the sound which would be heard from a high-end home cinema system.

Games are no different: once the rough gameplay is in place, it makes sense to spend some time fine-tuning it, tweaking and adjusting until it is as good as it can possibly be.

Audio

Finalizing audio and performing mastering duties is difficult with games. Games are interactive rather than passive, linear forms of entertainment. This means you need to consider how the individual audio assets in your project will interact and, if necessary, perform some preemptive mastering and mixing yourself. Essentially, mastering is a real-time task which needs to be handled within the game logic itself.

In an ideal world, every developer would have access to their own tame audio engineer, but this is the real world and many small-scale projects simply cannot afford this.

The most important consideration is ensuring each sound in your project sits well with all its fellows. This is a subjective process as some people like a lot of bass, while others prefer higher-frequency sounds in their soundscapes, so it's a good idea to use alpha and beta tester feedback to get a mix of objective views on your project's sound. (An audio monitoring setup is also useful, but these can be expensive and annoying to colleagues if you can't afford a dedicated room for the purpose.)

Ideally, you need to use audio from a single source, mixed by the same pair of ears, so you get a consistent sound. If – as in this tutorial's case – you must use a bunch of different audio sources, be prepared for a lot of tweaking of settings to get the sounds to sit just right in the game's overall mix. An over-loud sound effect will be very obvious to players and could become irritating if it is heard frequently. Play-test frequently and be prepared to budget some time for this process.

Sample Notes

Unity has a fairly simple interface for audio, but there are some important things to consider:

- Ensure your samples have similar levels. This makes volume levels and roll-off settings much more consistent. Normalization can help here, but you should also consider how your sounds will be mixed together. Too many noisy sound effects will confuse the player.
- Use mono samples if you need the sound to be positioned realistically within the 3D world.
- Compress samples using the Ogg Vorbis compression system. (Unity can do this for you.)
- If building for the web, you should always compress your sounds using Ogg Vorbis.
- Check the "Decompress on load" box for shorter, frequently-used sound effects.
- Only use stereo samples for long musical pieces which do not need to be positioned spatially within the scene. Such sample files will always be played back as they are, at the default Audio Listener volume.

Adding Sound to Lerpz Escapes!

We have already added a few spot effects to the game. In this chapter we will add some more audio to our game and complete the process.

In many game genres, sound effects can be obtained from real sources, or taken from a sound effects library. However, Lerpz Escapes! needs some sound effects which cannot be obtained so easily. Pointing a microphone at a passing spacecraft or convenient robot guard is not an option, so we will need to think creatively about these.

We will not add every possible sound effect to the game in this chapter; only enough to demonstrate Unity's audio features. Once you have read through this chapter, you will be able to add additional sound effects on your own.

The complete list of sound effects the game needs is:

The Player:

- Walking / Running sounds;
- Attacking sound;
- Getting hit sound;
- Dying sound;
- Jet-pack thrusters sound;

The Robot Guard:

- Idle sound;
- Attacking sound;
- Getting hit sound;
- Dying / Exploding sounds;

The Collectables:

- Fuel Cell collected sound;
- Health collected sound;

Ambience:

- (A long, looped sound sample to give a sense of ambience.)

Spaceship Impound Fence

- "Active" sound;
- "Shutdown" sound;

The Spaceship:

- A take-off sound to play with the cut-scene animation;

That's a minimum of fifteen sound effects. Some of these are simple foley sounds, such as the player's footsteps and the jet-pack thrusters. Some of the other samples are repurposed foley sounds. (The pickupFuel sound, for example, is just the sound of a golf ball being struck by a golf club.)

These sounds can be found in almost any sound effects library. Apple's own Garage-Band, Logic Studio 8 and Soundtrack / Soundtrack Pro software include such sounds and these libraries were the source for most of the sound effects in this tutorial. Many more such samples are available online.

For sample manipulation, we recommend a good sample-editing tool.

Ambient Sounds

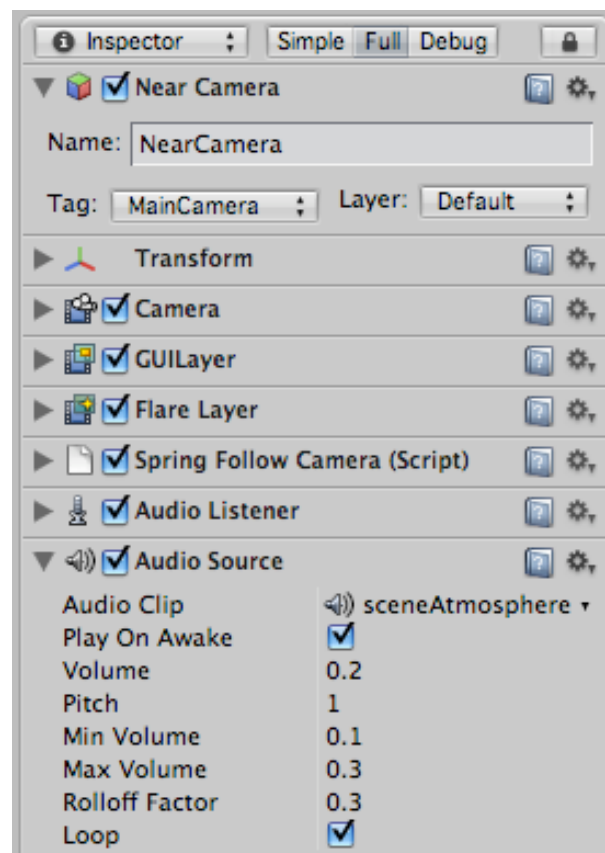
Our game is set in an environment way above the clouds with an impressive view of nearby planets. Such an exposed location needs a suitably out-of-this-world soundscape to improve the sense of immersion in the game.

The tutorial project includes a looped ambient sound sample named **sceneAtmosphere**. This is a stereo Ogg Vorbis sample created by throwing some old, mono BBC Radiophonic Workshop sound effects from the 1960s at GarageBand, and combining them into something that sounded suitably alien.

The sample is added to the Near Camera object in the Hierarchy Pane.

🔗 Drop the sample onto the Near Camera object. Unity will automatically create an Audio Source component.

🔗 Adjust its settings as shown below:



Adding the sceneAtmosphere sound to the Near Camera object.

The **Audio Source** component includes basic controls which allow us to tell Unity how to deal with the sound either through the Inspector or through scripting.

In this instance, we set the **Play On Awake** switch so the sound starts playing automatically. The volume level is deliberately set low as this is ambient, background sound and we don't want it to distract from the other sounds in the game.

Let's take a quick look at what the remaining settings mean:

The **Pitch** setting defines how fast the sample plays, with 1 being the normal speed. The algorithm used is a basic one, similar to a tape recorder, so there is no time-stretching involved; a setting of 2 here would play the sample at twice its normal speed, doubling its effective pitch.

TIP *The **Pitch** setting is useful for one-shot sound effects as you can adjust the value slightly with each playback to add variety to the sounds. It's ideal for short effects like gunshots, lasers and footsteps and saves having to create multiple samples.*

Next come settings which define the volume range of the sound. If we wanted the clip to be inaudible when we're a long way from its source, **Min Volume** would be set to zero.

As its name suggests, **Max Volume** is the maximum volume of the audio. We could stand right on top of the sound source and it wouldn't get louder than this value.

Next comes the **Rolloff Factor**, which determines how quickly the sound's volume changes relative to the listener's distance. A smaller value means the sound will be audible over a larger distance. This setting is key to ensuring realistic aural behavior in the game.

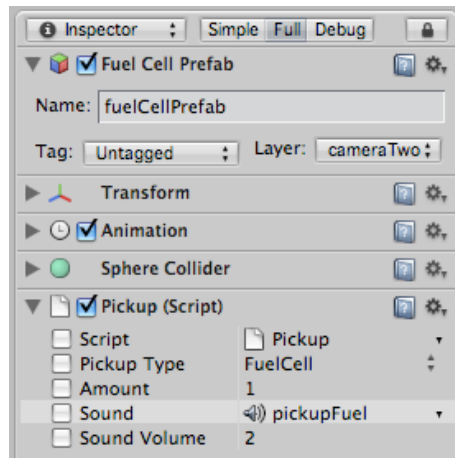
Finally we have the **Loop** setting. This is enabled because we want the impound fence's sound to keep on playing until we tell it to stop.

NOTE *It is important to ensure looped sounds are designed for clean looping or you will most likely hear an audible click or pop each time playback is restarted from the beginning of the sample. Most sample editors include a "find zero crossings" feature for this purpose.*

Collectables

The collectable items are the easiest to deal with. These sound effects are **pickupFuel** and **pickupHealth** for the fuel cells and health collectables respectively. Adding these effects is simplicity itself: the Pickup script already has support for audio; we just need to drop the relevant sound sample into the script's sound effect slot in each pickup type's Prefab.

The image below shows the **fuelCellPrefab** Inspector details for one of the instances in the level, with the **pickupFuel** sound effect added to the Pickup script's Sound variable:



Setting the *fuelCellPrefab*'s sound effect.

- Add the audio either by dropping the **pickupFuel** sound effect onto the **Sound** variable's slot, or by picking the sound effect directly from the list of available samples which you can bring up by clicking the small triangle to the right of the slot.
- Set the **Sound Volume** to 2, to make the sound effect stand out.
- Apply the same process for the Health pickup too, using the **pickupHealth** sample instead.

TIP You can add the sound effect directly to the Prefab to save time.

If you play the game now, each collectable item will now play the appropriate sound effect when you pick it up.

The Impound Fence

The force field that surrounds the spaceship should make a humming, fizzing noise while it is active. A suitable sound effect was created using GarageBand by combining some looped ambient textures together and exporting the resulting sound to a file.

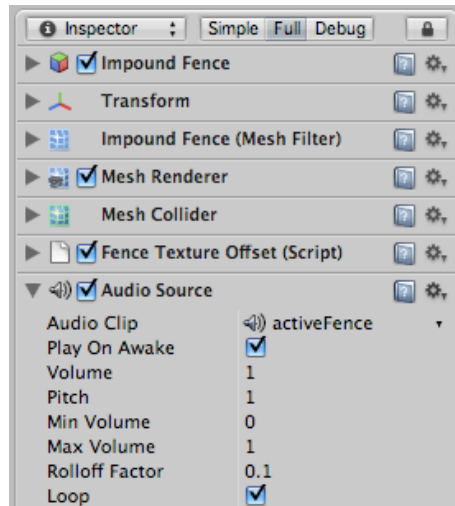
Apple's GarageBand only exports to the AAC (".M4A") sound format, so Audacity was used to convert this to the uncompressed, mono AIFF audio file used in the project.

The sample is named **activeFence**.

- Go to the Hierarchy Pane, open Scenery and find the **impoundFence** object.

Drop the **activeFence** sound directly onto this object. This will add an **Audio Source** component to the **impoundFence** object.

Finally, change the **Audio Source**'s settings to read as follows:



Impound Fence Audio Source settings.

The Player

Lerpz himself makes no sounds at the moment. Adding sound effects makes sense, but which ones?

The sound effects we will implement in this tutorial are:

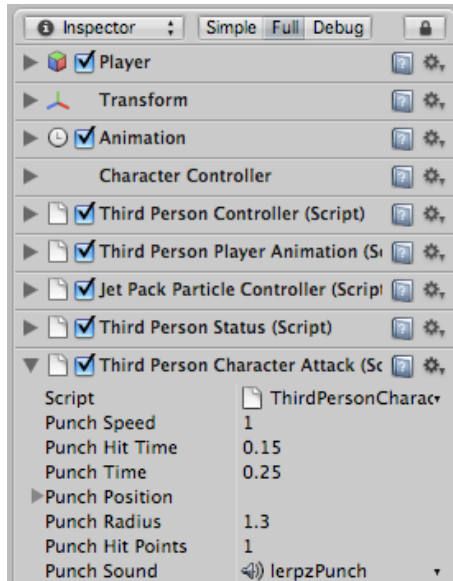
- A punch sound;
- A “struck” sound, played when Lerpz is hit by a robot;
- A sound effect for the jet-pack thrusters;
- A sound to play when the player dies and is re-spawned.

(The footstep effect is left as an exercise for the reader.)

These sounds will be played by our scripts.

Punching.

The punching movement and animation are handled by the **Third Person Character Attack** script. A property for the sound effect is exposed by the script in the Inspector. Set the **Punch Sound** property as shown:



Lerpz Punch sound effect.

The script plays this sound using the following code:

```
if (punchSound)
    audio.PlayOneShot(punchSound);
```

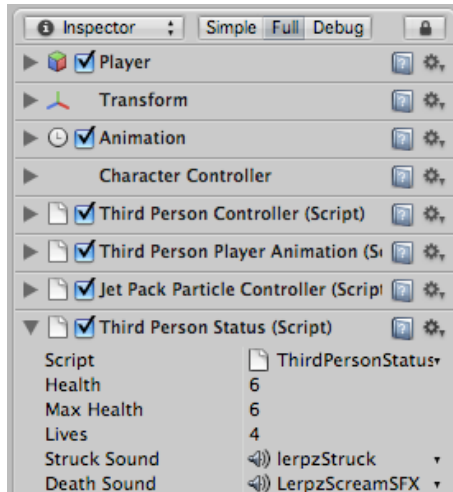
This first checks if a punch sound effect has been supplied to the script. If so, it uses the `PlayOneShot()` function to play the sound. This function creates a temporary `GameObject` with an `Audio Source` component, adds it to the `Scene` and plays it. When the sound effect is finished, the `GameObject` is removed from the scene.

NOTE While the game is running, you will see these one-shot sounds appearing briefly in the *Hierarchy Pane*. This is normal behavior.

Struck sound & Scream sound

The **Third Person Status** script handles two more sound effects: the one played when Lerpz is struck by an enemy, and the one played when Lerpz dies (just before he is re-spawned or the game ends). Both sound effects are handled the same way as the “lerpzPunch” sound effect we added earlier.

Add both the **LerpzStruck** and **LerpzScreamSFX** sounds as shown below:

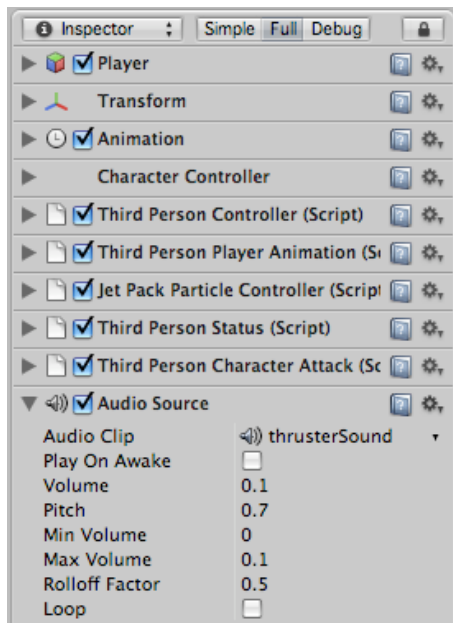


Adding the Struck Sound and Death Sound effects.

The Jet-pack

The jet-pack is a looped, rather than a one-shot, sound effect, so we'll add the sound effect to the Player GameObject directly as an Audio Source Component. The Jet Pack Particle Controller script will automatically create an empty Audio Source component, but we need to add the audio file to it:

- 🔗 Drag the **thrusterSound** audio file onto the Audio Source component inside the **Player** GameObject.
- 🔗 Ensure the Audio Source settings are as shown:



Jet-pack audio settings

There are two sections of script to handle this sound effect, both of which are in the **Jet Pack Particle Controller** script. The first section goes in the `Start()` function and initializes the Audio Source component:

```
var soundSource : AudioSource = GetComponent();
soundSource.loop = false;
soundSource.Stop();
```

The second chunk is further down in the same function:

```
if (isFlying)
{
    if (!soundSource.isPlaying)
    {
        soundSource.Play();
    }
}
else
{
    soundSource.Stop();
}
```

This code is self-explanatory. The need to test if the sound effect is already playing is because the `Play()` function will always start playing the sound effect from the beginning, regardless of whether it is already playing. This means we'd hear a stuttering sound as Unity would repeatedly restart the sound sample every time the `Play()` function is called.

The Robot Guards

These guys have a number of scripts, some of which also have audio samples as properties. In addition, each robot guard has an Audio Source component.

Unlike the **Player**, which uses its Audio Source component solely for the jet pack sound, the **EnemyPoliceGuy** script uses the Copper Prefab's Audio Source component for multiple looping sounds, switching between them as required. An example of the code used to achieve this is shown below:

```
if (idleSound)
{
    if (audio.clip != idleSound)
    {
        audio.Stop();
```

```
        audio.clip = idleSound;
        audio.loop = true;
        audio.Play();
    }
}
```

The above example can be seen at the top of the **EnemyPoliceGuy** script's `Idle()` function. The call to `audio.Stop()` is important here as swapping out a sample while it's being played can have unpredictable results.

To add the sound effects:

- 🔗 Select the **Copper** Prefab in the Project Pane to bring it up in the Inspector.
- 🔗 Drag the **CopperIdleLoop** sound effect into the Audio Source component.
- 🔗 Add the same audio file to the Idle Sound property in the **EnemyPoliceGuy** script component.
- 🔗 Add the **CopperActiveLoop** sound effect to the **Attack Sound** property in the same component.
- 🔗 Finally, add the **MetalHit** sound effect to the **Struck Sound** property of the **Enemy Damage** script component.

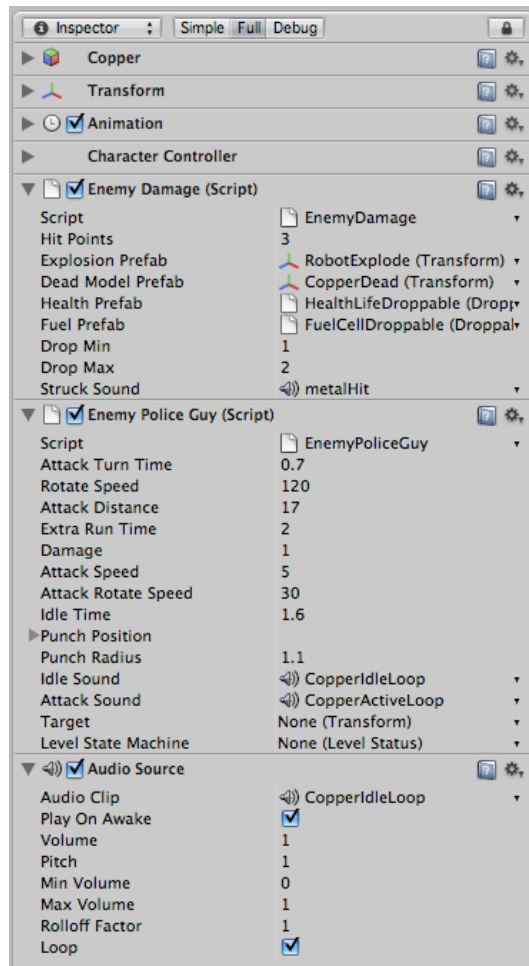
(The resulting view in the Inspector Pane is shown on the next page.)

The **MetalHit** sound is played in the `ApplyDamage()` function within the **Enemy-Damage** script. The code which does this is shown below:

```
if (audio && struckSound)
    audio.PlayOneShot(struckSound);
```

The `audio` variable is actually a shortcut variable created by Unity itself. It is the equivalent of a `GetComponent (AudioSource)` function call. The upshot is that the Audio Source component attached to the Copper Prefab is used to play the sound effect for us, saving us the need to instantiate a temporary audio source component for the purpose.

NOTE *The same trick is used for some of the Player sound effects.*



Copper Prefab Inspector settings after adding the audio files.

Cut Scenes

Cut-scenes provide a useful way to fill in the player on an event or story element which they need to know. Ideally, the need to cut away from the player should be kept to a minimum. For this reason, we have only two cut-scenes.

The first occurs when the player manages to collect all the required fuel cells in the level, thus unlocking the spaceship. This cut-scene appears using a picture-in-picture technique, so that the player can continue playing while the sequence plays out.

One extremely pragmatic reason for not taking over the whole screen with this cut-scene is that we'd otherwise have to freeze all the game element – robots, player controls, etc. – while the cut-scene plays because the player wouldn't be effectively blind until the scene completes.

NOTE *It is still possible to get at the spaceship if you climb onto the nearby crates, but the spaceship is still locked down at this point and won't take off. (The mesh collider isn't changed to a Trigger type until the impound fence is unlocked.)*

The second cut-scene occurs when the player touches the spaceship after the fence has been disabled. In this scene, which is played full-screen, we see the spaceship taking off and flying off to freedom and new adventures before switching to the Game Over sequence.

Let's look at the first scene in detail...

Unlocking the impound fence

We first came across the Impound Fence part of the level when we added some animation to it. Earlier in this chapter, we added a sound effect to it as well. Now we'll animate it.

🔗 Open the **LevelStatus** script. This is where all the cut-scenes are handled.

🔗 At the top of the script, add the following code:

```
var exitGateway: GameObject;  
var levelGoal: GameObject;  
  
var unlockedSound: AudioClip;  
var levelCompleteSound: AudioClip;  
  
var mainCamera: GameObject;  
var unlockedCamera: GameObject;  
var levelCompletedCamera: GameObject;
```

NOTE *The above code includes variables we'll need for the second cut-scene too.*

🔗 Next, add the following line of script code to the **Awake()** function:

```
levelGoal.GetComponent(MeshCollider).isTrigger = false;
```

That line is crucial. It stops the second cut-scene from triggering prematurely. With the **isTrigger** switch turned off, the spaceship becomes just another part of the scenery as long as the impound fence is still active.

Now for the fun part: the unlocking sequence itself.

✎ Add the following function to the **LevelStatus** script. (I'll explain it as we go.)

```
function UnlockLevelExit()
{
    mainCamera.GetComponent(AudioListener).enabled = false;
```

Unity supports just one – and *only* one – Audio Listener component in any one Scene. Usually this is attached to the main Camera, but we're using multiple cameras in our Scene, so we need to ensure we only have the one Audio Listener active at any one time. We want to hear our "fence disabled" sound effects, so we'll briefly enable the Audio Listener on our cut-scene's camera here.

Next, we need to activate the cut-scene camera and enable its Audio Listener component:

```
unlockedCamera.active = true;
unlockedCamera.GetComponent(AudioListener).enabled = true;
```

The Impound Fence has a looped sound effect attached to it. We need that sound to stop playing now:

```
exitGateway.GetComponent(AudioSource).Stop();
```

Now we can start playing our "fence disabled" sound effect:

```
if (unlockedSound)
{
    AudioSource.PlayClipAtPoint(unlockedSound,
    unlockedCamera.GetComponent(Transform).position, 2.0);
}
```

With our sound effect started, we can start the animation sequence. We'll do this *procedurally* by using script code to achieve the animation. The next few lines perform this sequence. The first line adds a delay before the sequence begins to give the appearance of the cut-scene time to register on the player's consciousness. I've left the remaining comments in place so you can follow the animation sequence itself:

```

yield WaitForSeconds(1);

exitGateway.active = false; // ... the fence goes down briefly...

yield WaitForSeconds(0.2); //... pause for a fraction of a second...
exitGateway.active = true; //... now the fence flashes back on again...
yield WaitForSeconds(0.2); //... another brief pause before...
exitGateway.active = false; //... the fence finally goes down forever!

```

We now have access to the ship! All we need to do now is make the spaceship's Mesh Collider component a Trigger rather than a normal collider:

```

levelGoal.GetComponent(MeshCollider).isTrigger = true;

```

Finally we pause a few more seconds so the player has time to see the results, before shutting off our cut-scene camera and restoring the Audio Listener component on our main camera:

```






yield WaitForSeconds(4); // give the player time to see the result.

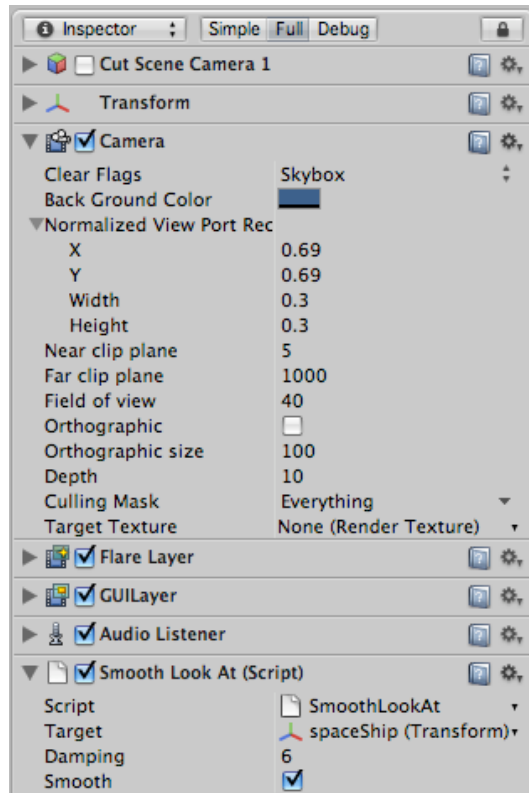
// swap the cameras back.
unlockedCamera.active = false; // this lets the main camera get the screen all to
itself.
unlockedCamera.GetComponent(AudioListener).enabled = false;

mainCamera.GetComponent(AudioListener).enabled = true;
}

```

Creating the cut-scene camera itself is our next task. This is just another Camera game object, exactly like the one we've been using in the game itself. Let's set it up:

-  Add a new Camera to the Scene.
-  Rename it CutSceneCamera1
-  Add the **SmoothLookAt** script to the Camera.
-  Drop a link to the **spaceShip** model onto the **SmoothLookAt** script so the script knows what the camera needs to point at.
-  Set the remaining properties as shown:



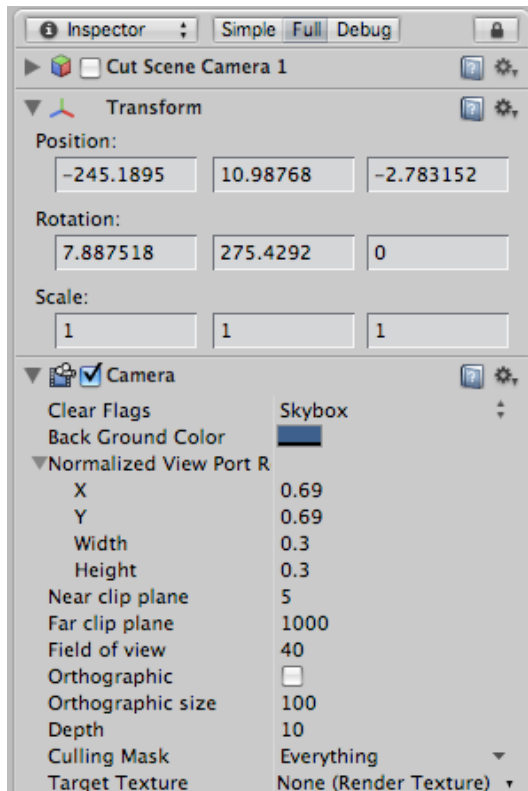
Cut Scene Camera 1 properties.

This camera should be set to look at the spaceship impound lot, clearly showing the fencing. The settings shown above should be a good approximation, but feel free to tweak it:



Positioning Cut Scene Camera 1.

Next, adjust the **CutSceneCamera1** Camera settings like so:

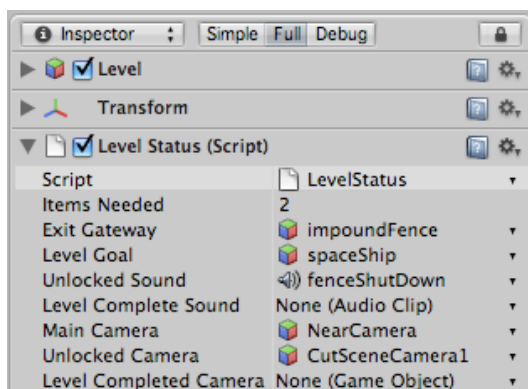


Cut Scene Camera 1 Camera Component settings.

Firstly, the camera needs to be disabled. It'll be enabled by our scripts, but we don't want it rendering anything until the script says so. Disabling the camera is simple: just uncheck the box next to **Cut Scene Camera 1**, right at the top of the Inspector.

Also, take care to set the **Normalized View Port Rect** settings as shown above. These define the camera's output's position on the screen so that it appears in the top right corner of the display. The camera's depth is also set to 10, which is higher than that of the **Near Camera** so the cut-scene will appear on top of the main game imagery.

We need to test the sequence, so set the **LevelStatus** script's properties as shown below:



Setting the Level Status script properties.

The **spaceShip** object is the spaceship model sitting inside the impound fence.

TIP I've temporarily set the **Items Needed** value to 2 in the screenshot above. This lets me test the cut-scene by collecting only two fuel cans instead of wasting time running around most of the level first. Remember to reset it to a higher number – say, 20 – when we're done.

If you play the game now, you should see the cut-scene appear as shown below.

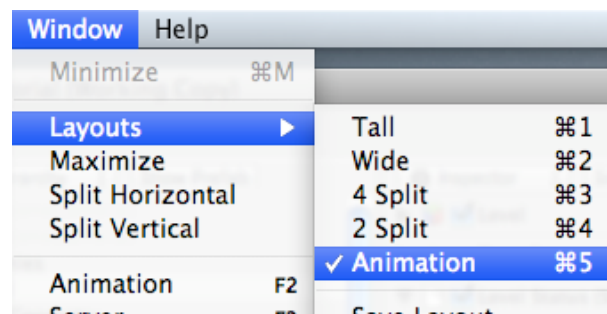


Our first cut-scene in action.

NOTE The frames-per-second counter visible in both the main view and the cut-scene insert is covered in the chapter on optimization.

The last cut-scene is a little more complex. We need the spaceship to take off and fly away. We could do this using scripting, but it's much easier to use Unity's built-in Animation features for this instead:

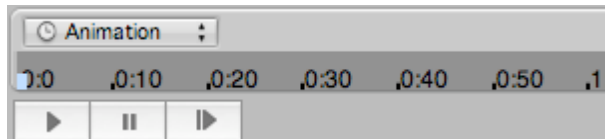
- 🔍 Click on the spaceship model inside the impound fence to select it. (Or click on it in the Hierarchy Pane.)
- 🔍 Switch to the Animation view:




Switching to the Animation Layout. Select from the Windows menu, or just use CMD+5

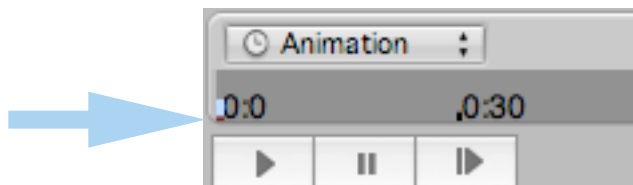
You should now see the Animation Pane across the bottom of the Unity window. This displays a long timeline, counting up from 0.0.

☞ Click on the 0.0 position on the timeline. A small blue highlight should appear:



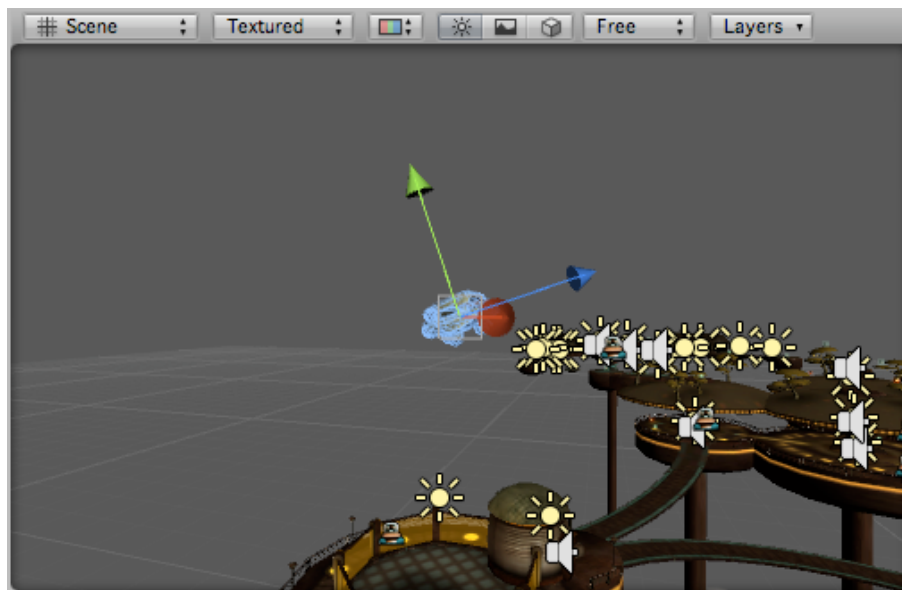
The 0.0 position, highlighted, on the timeline.

☞ Click on the Record button  to record the spaceship's current position as the first key-frame. A small red mark should appear in the timeline to mark the key-frame:



The first key-frame is recorded. (The red mark is just visible at the bottom of the pale blue highlight.)

☞ With the spaceship model still selected, move it up into the air above the platform and tilt its nose up a little too:



The spaceship's first maneuver: Straight up and angling towards freedom.

☞ Now move along the timeline to around the four-second mark. (4.0 on the timeline.)

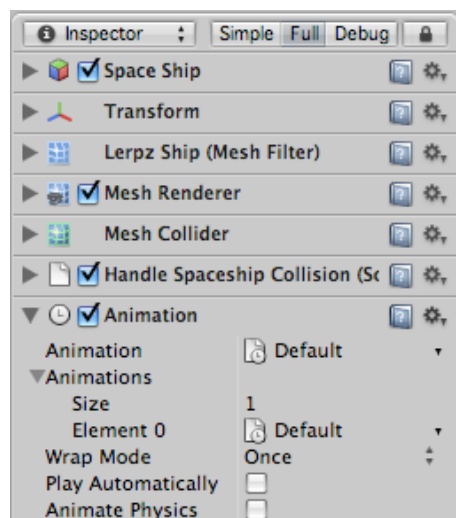
- ✎ Record the spaceship's new position as a keyframe by clicking the Record button again.
- ✎ Next, move the spaceship a long way away from the platforms by dragging the blue arrow. (This moves the spaceship forward along its local Z axis.) The spaceship's new position will be its final key-frame, so it should be barely visible from the impound lot.
- ✎ Move along the timeline again to around the sixteen-second mark. (16.0 on the timeline.) You may need to scroll the timeline along to get to this position: hold down the Option-drag the timeline to the left to bring it into view. (You could use the mouse's scroll-wheel to zoom in and out of the timeline if you prefer.)
- ✎ Record the spaceship's new position as a keyframe by clicking the Record button one last time.
- ✎ Finally, click back at the 0.0 position on the timeline.

NOTE *The final step above is important: if you don't do this and try playing the game again, the spaceship will appear at the location defined on the timeline where you left the timeline's blue cursor. As we'd just recorded the final keyframe of the animation, the spaceship would appear at that point! It is therefore crucial to always click back on the 0.0 mark of the timeline once you're done.*

If you click on the spaceShip object and look in the Inspector, you'll see a new Animation component has been added.

By default, Unity assumes you want animations to play automatically. We don't want this, so, with the game stopped:

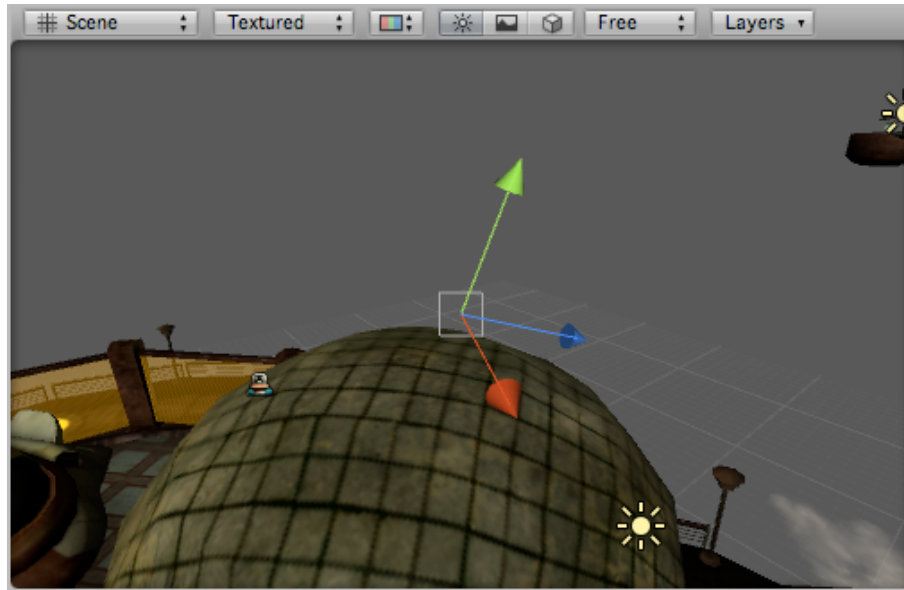
- ✎ Click on the spaceShip model to bring up its properties in the Inspector.
- ✎ Adjust the Animation component as shown:



The spaceShip object's animation settings.

The next step is to create our second cut-scene camera:

- ✎ Create a new **Camera** object.
- ✎ Rename it **CutSceneCamera2**.
- ✎ Position it on top of the impound lot's office building, as shown:

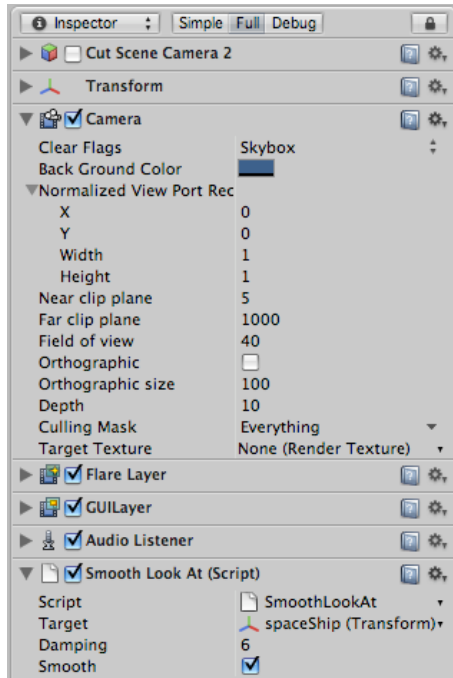


Positioning CutSceneCamera2.

Don't worry about the direction it's facing: we're only interested in its location. The script we'll attach to it will take care of the rest.

- ✎ Add the **SmoothLookAt** script to the Camera.
- ✎ Drop a link to the **spaceShip** model onto the **SmoothLookAt** script so the script knows what the camera needs to point at.

As with **CutSceneCamera1**, we also need to ensure this one is disabled by default, but the remaining settings are a little different, as you can see from the screenshot on the next page. The key difference is that this camera needs to take over the whole screen, rather than appearing in the corner, so the Normalized View Port Rect properties are set to take this into account.



Settings for CutSceneCamera2.

Now for the cut-scene itself. This is a little trickier than our first cut-scene because the messages used to trigger the scene need to be relayed along a chain:

The initial trigger happens when the Player touches the spaceShip model. (If the first cut-scene has played through, the spaceShip model is now acting as a trigger instead of a solid object.)

The spaceShip model therefore needs a script attached to it to deal with this trigger event:

- 🔗 Create a new JavaScript script asset.
- 🔗 Rename it **HandleSpaceshipCollision**.
- 🔗 Add the following code to it:

```
private var playerLink : ThirdPersonStatus;

function OnTriggerEnter (col : Collider)
{
    playerLink=col.GetComponent(ThirdPersonStatus);

    if (!playerLink)// not the player.
    {
        return;
    }
    else
```

```

    {
        playerLink.LevelCompleted();
    }
}

```

All the above code does is check if the player has touched the spaceship and, if so, call the `LevelCompleted()` function in the Player's **ThirdPersonStatus** script.

The **ThirdPersonStatus** script's `LevelCompleted()` function is even shorter and does something very similar:

```

function LevelCompleted()
{
    levelStateMachine.LevelCompleted();
}

```

levelStateMachine is a property which links to the **LevelStatus** script. **LevelStatus** is where the action is as the level completion animation is something only the level-related scripts should know about.



Add the `LevelCompleted()` function to **LevelStatus** now (I'll explain it as we go):

```

function LevelCompleted()
{

```

First, we have to do the same Audio Listener switchover as we did for our first cut-scene:

```

    mainCamera.GetComponent(AudioListener).enabled = false;
    levelCompletedCamera.active = true;
    levelCompletedCamera.GetComponent(AudioListener).enabled = true;

```

Next, we want to give the illusion that the player is inside the spaceship, so we'll hide him. Do do this, we'll send a "HidePlayer" message to the Player's **ThirdPersonController** script. The function disables the rendering of the player, so he becomes invisible:

```

    playerLink.GetComponent(ThirdPersonController).SendMessage("HidePlayer");

```

Just to be on the safe side, we'll also physically relocate the player to a position we

know should be safe. (The robots don't check if the player's visible or not and they're still operating!) In this instance, we'll just move the player 500 units straight up, which should be far enough away from any immediate danger.

```
playerLink.transform.position+=Vector3.up*500.0; // just move him 500 units
```

Next, we'll start the level completed sound effect. In this case, it's the sound of the spaceship taking off:

```
if (levelCompleteSound)
{
    AudioSource.PlayClipAtPoint(levelCompleteSound, levelGoal.transform.position,
2.0);
}
```

Now we start the timeline-based animation we recorded earlier and wait for it to finish:

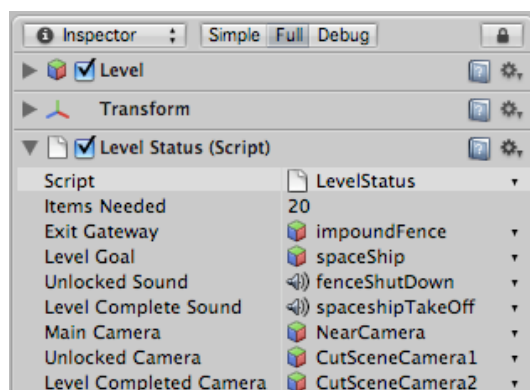
```
levelGoal.animation.Play();

yield WaitForSeconds (levelGoal.animation.clip.length);
```

And finally, we load the "Game Over" Scene:

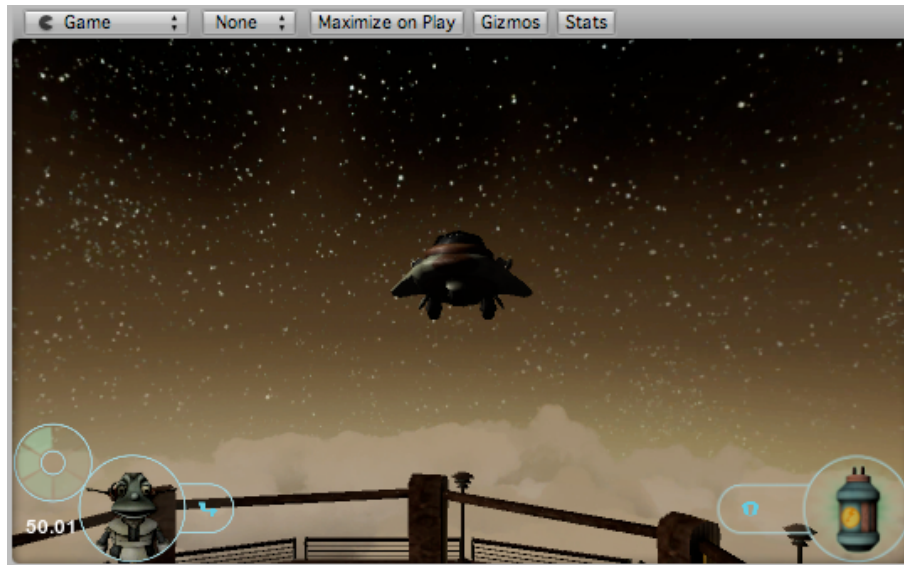
```
Application.LoadLevel("GameOver"); //...just show the Game Over
sequence.
}
```

The last step is to set the script's properties as shown:



The final settings for the Level Status script properties.

The result, when you collect all the fuel cans and jump into the spaceship, should look something like this:



Mission completed! Our hero takes off for new adventures.

The next chapter wraps things up with a look at optimization techniques.

Optimizing

Games may be required to run on anything from a G4 iBook to a current Intel-based Mac Pro stuffed with memory and a high-end graphics card or two. For our games to run on both, we need to optimize...



We're now at the wrapping-up stage. Optimizing is something done near the end of a project, once all the key elements are in place and nailed down. What, where and how you optimize your project very much depends on your project's design and content, so this chapter is primarily a discussion covering the more common types of optimization.

Why Optimize?

Unity projects are often targeted at older computers than, say, a mainstream, big-budget game. Puzzle games, casual titles and other project types may be required to run on anything from a G4 iBook with a mere 256Mb of RAM and an ancient graphics chipset, through to a current Intel-based Mac Pro stuffed to the faceplates with memory and a high-end graphics card.

For this reason, we need to consider optimizing our projects for final release. In our case, we've already optimized the enemy robots by making the pop into existence only when in range, so that's already covered. However, the rendering of the scene can be quite slow and this is worth looking at in more depth.

Optimizing Rendering: Monitoring Frames Per Second.

The best way to determine whether your game needs to be optimized is to find out the frame-rate -- the number of images being rendered each second -- and display this. The lower the number, the slower the game is running.

You may have noticed a number, or the letters "FPS", in some of the screenshots in this tutorial. This is because one important, but mercifully very short, script is running

which we'll look at now: a *Frames-per-second reporter* script. The script is named simply **FPS** and can be found in the *Scripts* -> *GUI* folder of the Project Pane.

The script is thoroughly documented, so I won't cover it in detail here, other than to add that it requires a **GUIText** Component. (This is a Unity 1.x GUI Component type as opposed to the new Unity 2.0 GUI.)

With this FPS counter, it is easier to get a good idea of where optimizations need to be made.

Making sense of the Stats display

New to Unity 2.0 is a "Stats" button above the Game View. If we enable this, we can get some additional metrics on our game, which can help determine whether there are issues of polygon-count or other object complexity to resolve.



The Stats panel in action.

These statistics are based on what the camera is rendering, so moving around the scene will change many of the statistics. The important elements are:

Draw Calls – the number of render passes. Each light in the scene will increase this number.

Tris – The number of triangles being drawn.

All 3D models are built up from triangles. The fewer the triangles, the faster they'll render. Round, curved objects will generally use more triangles than basic, straight-edged shapes like cubes and planes.

Verts – The number of vertices being sent to the graphics chip.

A vertex is a point in 3D space. The more vertices you can share across triangles, the more of said triangles you can render and thus the more complex your models can be.

Used Textures – the number of textures used to render what you see.

Materials can use one or more textures, depending on how the material is defined and the shader script it is using. The shader defines how the textures are combined, producing effects such as bump-mapping, glossy shine highlights, reflections and refraction.

TIP *Particle systems use two triangles and at least one texture for each particle. It is very easy to get carried away with these effects, but you should take care not to overdo it.*

Render Textures – number of cameras outputting to a texture rather than directly to the display.

Render textures are used to achieve a number of effects, such as a CCTV screen displaying another area of a level, or to produce reflection (e.g. a mirror), and water or glass refraction effects.

Optimizing Rendering: The Two-Camera System.

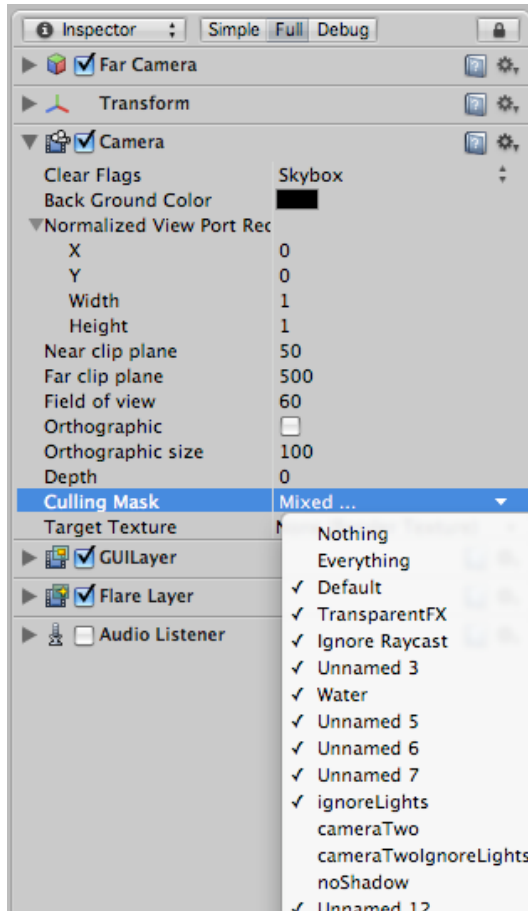
If you have played with the completed project, you may have noticed by now that there are, in fact, two main cameras: a Near Camera and a Far Camera. This two-camera system reduces the amount of rendering needed for each frame.

The **Near Camera** renders everything within a close range; in this case, from 0.4 to 50 units.

TIP *A unit can, in theory, be any arbitrary length you like, but most developers tend to stick with "1 unit = 1 meter" for the sake of sanity. An important tip is to make sure your artist(s) are aware of the scale you are using.*

The **Far Camera** renders from the 50 units mark through to around 500 units. However, it only renders a subset of the Scene data. The subset is defined by the tags attached to each object in the Scene. The **Near Camera** renders all elements regardless of their tag, but as you can see from the screenshot below, the **Far Camera** has been told to ignore those items with the "cameraTwo" or "cameraTwoIgnoreLights" tags. We'd also like it to ignore the fuel cans and health pickups. These have the "noShadow" tag, so that too is unchecked for this camera.

NOTE *Unchecking the "noShadow" tag also means it won't render the player either, which is fine.*



The Far Camera Culling Mask settings.

The **Far Camera** is also the camera where the Skybox is rendered. (See the **Clear Flags** setting in the shot above.) The **Near Camera**'s **Clear Flags** setting is "Depth Only", so that its content is superimposed over that of the **Far Camera**. The **Far Camera**'s content is rendered first.

This selection is defined by the Culling Mask property of the camera component. Ticked layers are rendered; unticked layers are not. You can define a Layer in the Inspector and assign one to any GameObject.

You can see this optimization in action with the robot guards and the collectable items. If you move towards one of these items, you will see the scenery around it is always rendered while the item itself appears relatively close to the player.

End of the road.

What we have learned.

Where to go next.



The Road Less Travelled

At the time of writing, this tutorial holds the dubious honor of being the longest ever produced for Unity. We have seen how to build a single level of a game based around the 3D Platformer genre, but even after all these pages, we have barely scratched the surface of what is possible with Unity 2.0, or even with this particular genre.

Our journey together is done: It's time for you to take the stabilizer wheels off the bicycle and continue alone, but before you go, here are some suggestions for what to try next...

Suggested Improvements

Lerpz Escapes has been left deliberately unfinished. We have a very basic Start Menu and a Game Over screen, but there's only the one game level and it is clearly intended to be the last one in the game. How could it be improved?

Fixing the deliberate mistakes

Yes, there are some minor issues with the game as it stands. These have been deliberately left in place to give you a chance to hone your skills. They are:

- If you kill a robot, but are respawned nearby before you have moved out of range, a new robot will appear, but the old one will remain.
- The Laser Traps don't kick the player away, so it is possible to lose all your health rather quickly.

Both can be resolved by applying what you have learned in this tutorial.

More levels

The Project Pane includes a “Build Your Own” folder containing all the individual assets used to build the level, so adding new levels should not be difficult.

You will need to use the `DontDestroyOnLoad()` function so that you can carry game state information between the levels such as the current score, lives remaining, etc.

More enemies

The game only has the one ambulatory enemy in the form of the robot guards. Why not add some more? This is a good way to ensure you have understood the animation and AI aspects. It will also help you get a firm grasp of building models and importing them into Unity.

Add scoring

Lerpz Escapes lacks a score system. Adding one is not difficult, but adding visual effects when the player does something worthy of increasing their score can be as challenging as you like. (And, of course, you’ll want to keep track of the score across Scenes.)

Add a networked high-score system

The Unity 2.0 brings solid networking support to the table. What better way to show off than uploading your high score to a central server so you can gloat? This is a good way to wrap your head around networking basics.

Add multiplayer support

Adding networked multiplayer game support is probably the trickiest thing you can do with any game. Naturally, Unity can help here too, but you will need to get down and dirty with scripting. This is a good, advanced-level improvement to add.

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/>

In addition, the Unify Wiki is an excellent source of user-contributed info:

<http://www.unifycommunity.com>

And finally, you can talk to experts and newcomers alike in our thriving forums, here:

<http://forum.unity3d.com/>