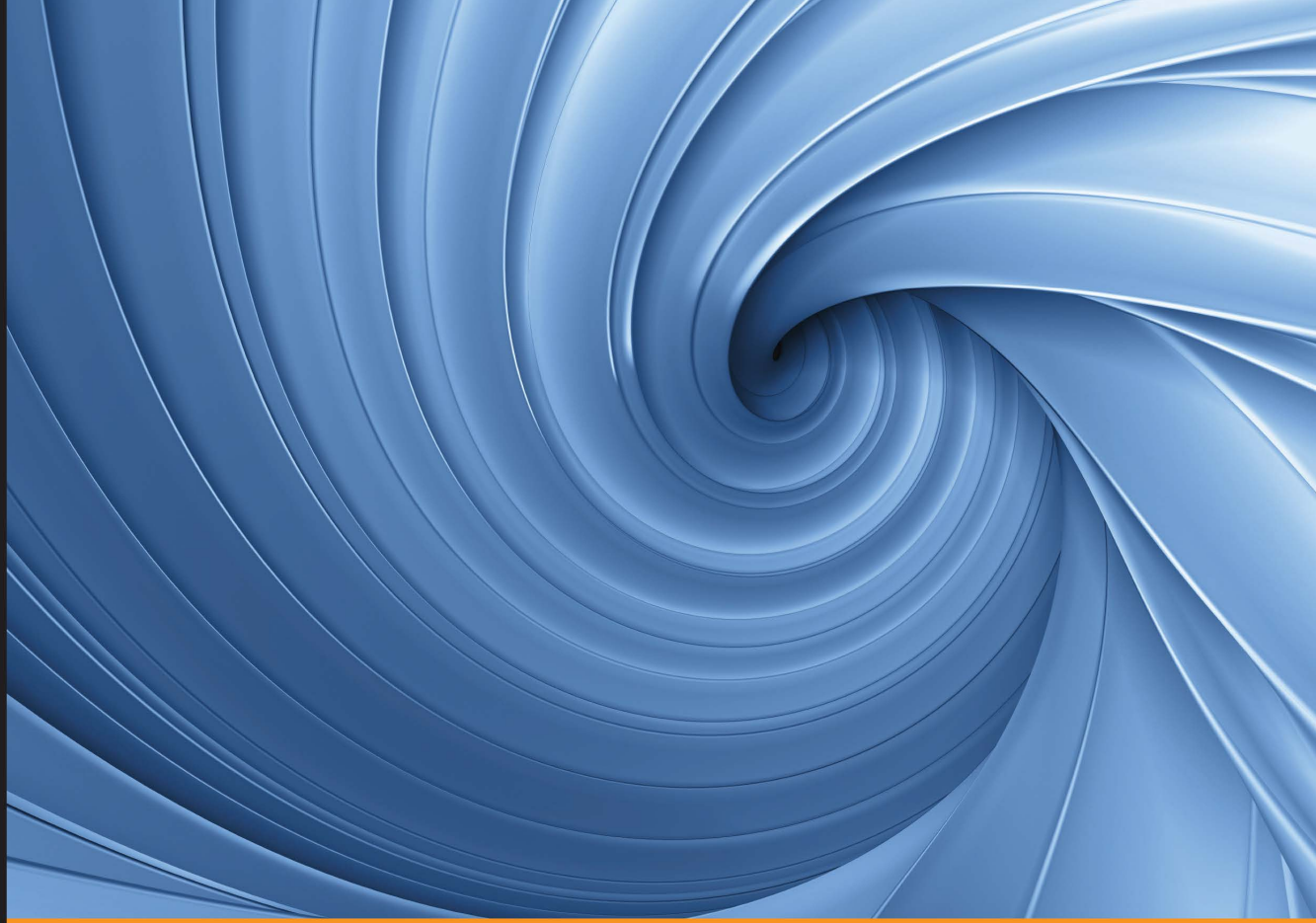


Community Experience Distilled

Mastering Unity Shaders and Effects

Harness the power of Unity 5 tools to write shaders and create stunning effects for next generation games

Jamie Dean

[PACKT] PUBLISHING

Mastering Unity Shaders and Effects

Harness the power of Unity 5 tools to write shaders and create stunning effects for next-generation games

Jamie Dean

BIRMINGHAM - MUMBAI

Mastering Unity Shaders and Effects

First published: September 2016

Production reference: 1210916

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.
ISBN 978-1-78355-367-9

www.packtpub.com

Credits

Author

Jamie Dean

Reviewer

Kenneth Lammers

Commissioning Editor

Akram Hussain

Acquisition Editor

Aaron Lazar

Content Development Editor

Deepti Thore

Technical Editor

Manthan Raja

Copy Editor

Vibha Shukla

Project Coordinator

Shweta H Birwatkar

Proofreader

Safis Editing

Indexer

Mariammal Chettiyar

Production Coordinator

Arvindkumar Gupta

About the Author

Jamie Dean is a game artist, instructor, and freelancer, with over seven years of teaching experience in higher education. He is currently focused on developing content for mobile games.

Jamie also wrote *Unity Character Animation with Mecanim*, *Packt Publishing*, in *2015*.

I would like to thank my family, Carey, Silas, and Rowan, for their love and encouragement, and all the people at Packt Publishing who helped get this project off the ground and keep the momentum. Thanks to Keith, Chip, and rest of my colleagues at Concrete Software Inc. Thanks for the great opportunities!

About the Reviewer

Kenny Lammers has worked over 16 years in the gaming industry as a character artist, technical artist, technical art director, and programmer. Throughout his career, he has worked on titles such as Call of Duty 3, Crackdown 2, Alan Wake, and Kinect Star Wars. He currently owns and operates Ozone Interactive, along with his business partner Noah Kaarbo. Together, they have worked with clients such as Amazon, E-line Media, IGT, Microsoft, and BioLucid.

Kenny has worked for Microsoft Games Studios, Activision, and Surreal, and has recently gone out on his own, operating CreativeTD and co-operating Ozone Interactive.

Kenny authored *Unity Shaders and Effects Cookbook*, *Packt Publishing*, and was very happy to be a part of the writing and updating of *Unity 5.x Shaders and Effects Cookbook*, *Packt Publishing*.

www.PacktPub.com

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Unity offers a range of tools to get your game project up and running. In version 5, the shaders got a significant upgrade, mostly being replaced with a set of Standard Shaders that can be easily set up to emulate the physical characteristics of real surfaces.

In this book, we will be exploring the options made possible with this new set of shaders and how they interact with lighting and effects in the game scene to produce high quality results.

It has been a long journey towards this level of realism in games. For a long time, the processing overhead of real-time lighting made it problematic. The burden of realism was placed squarely on the texture artist who had to bake static surface effects into the textures of characters and props.

Particularly with dynamic game objects such as characters, this made for some disappointing results—characters would move into darker or more brightly lit spaces with a disappointing level of light variation, exposing the simplicity and lack of realism in the game scene.

Advances with graphics, particularly the rapid improvement of graphics cards, have made real-time lighting techniques more cost effective.

This has resulted in higher expectations in games—more realistic surfaces that respond to dynamic lighting in a game scene.

In this book, readers will explore these shader and lighting concepts through the context of a typical sci-fi horror game. This will allow us to explore a variety of lighting and surface effects.

What this book covers

In this book, we will work through each of the major shader and effects concepts in different chapters. In each chapter, we will work through a different scene file:

`Chapter 1`, *Getting to Grips with Standard Shaders*, introduces the context and explores the capabilities and limitations of Unity's Standard Shaders as we set up the materials for our sci-fi horror-themed spacecraft bridge scene.

`Chapter 2`, *Creating Custom Shaders*, continues with spacecraft maintenance scene—creating custom shaders to improve the appearance of the game scene. We will create our first custom shader from scratch and then build on this code to create more complex effects for the helmet transparency and planet's atmosphere.

`Chapter 3`, *Working with Lighting and Light-Emitting Surfaces*, delves into the relationship between scene lighting and light-emitting surfaces, creating an animated holographic display for our spacecraft cockpit.

`Chapter 4`, *Animating Surfaces with Code and Shaders*, explores different techniques to implement animation in a shader workflow. We will demonstrate UV scrolling and iterating through texture arrays with C# to complete our animated holographic spacecraft display before showing more complex vertex animation in a custom shader.

`Chapter 5`, *Exploring Transparent Surfaces and Effects*, introduces different applications of transparency on our planet's surface scene and setting up various materials with existing and purpose-created shaders.

`Chapter 6`, *Working with Specular and Metallic Surfaces*, highlights the differences between Unity 5 Standard Shaders' specular and metallic workflows. We will create custom shaders for both types to create some special in-game effects.

`Chapter 7`, *Shaders for Organic Surfaces*, covers the creation of unique skin and hair shaders for a character in an interior scene. The first shader explores methods of representing subsurface scattering effects for partially translucent materials, such as our astronaut character's skin. We follow this up with creation of PBR-compatible hair and eye shaders.

`Chapter 8`, *Custom Particle Shaders – Smoke, Steam, and Fluids*, teaches how to create shaders to work with particle effects such as smoke, steam, and fire. As earlier, we will go beyond Unity's default shaders to create more advanced cinematic effects to work within our sci-fi horror context, this time a spacecraft corridor.

`Chapter 9`, *Optimizing Shaders for Mobile*, explores testing custom shaders on mobile platforms, such as Android and iOS.

What you need for this book

Understanding key 3D graphics concepts is necessary to complete the chapters in this book. In addition, you will need the following:

- Unity 5 installed on your machine (macOS or Windows). The free personal version of the software is sufficient for all of the project content in this book. This can be downloaded from the Unity webpage.
- A basic understanding of Unity script or C# will be helpful to complete the projects, though the code that is included is clearly explained.

Who this book is for

This book is intended for intermediate-level game developers that have experience with Unity and C# and are interested in taking the next step with shaders and special effects for next-generation games. A basic background in game development and art is assumed.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Give the project a suitable name, such as `Shaders and Effects`, and assign a location on your computer's hard drive from where it can easily be accessed."

A block of code is set as follows:

```
Properties
{
    _Color ("Color", Color) = (0,0,0,0)
    _EdgeColor ("Edge Color", Color) = (0,1,0,1)
    _Width ("Width", float) = 0.1
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "When prompted to open an existing project or create a new one, choose **Create New Project**."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/mastering-unity-shaders-and-effects`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `http://www.packtpub.com/sites/default/files/downloads/MasteringUnityShadersAndEffects_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Getting to Grips with Standard Shaders

In this chapter, we will explore Unity's **Standard Shaders**, creating a set of materials that we will apply to our first scene.

This new set of shaders was introduced in Unity's version 5 release. The Standard Shaders utilize physically-based rendering.

Before the Standard Shader arrangement, specialized shaders were required to create transparent, reflective, and metallic materials. This made including a lot of different surfaces in a single game scene problematic—each shader is essentially a small program that needs to be run, negatively impacting the performance.

The Standard Shaders offer a universal approach—they can be reflective, transparent, metallic, and all of these. Unused map channels are discarded at runtime, making for an efficient shader and reduced performance time.

Combined with Unity's versatile **Enlighten** lighting solution, Standard Shaders allow for more responsive, realistic surfaces in a game scene.

We will compare the Standard Shader metallic and specular workflows, and offer different opportunities to use both within our game context.

This chapter will cover the following topics:

- Discussing the project setup and importing a custom Asset Package
- Creating complex materials to cover a variety of surfaces
- Defining specular and metallic values using sliders and texture maps
- Utilizing material groups on imported models

- Making surfaces transparent
- Layering materials
- Discussing key differences between Standard (metallic) and standard specular workflows
- Discussing Skybox asset creation
- Discussing basic scene light adjustment

Throughout this book, we will develop an understanding of shaders and the effects within the context of a next generation science-fiction horror game.

In the game, a lone astronaut journeys in her spacecraft to a distant planet, **Ridley VI**, to make contact with a research team after communication with the team was mysteriously cut off.

This context will give us plenty of opportunities to explore the use of shaders and effects in a game development setting.

Let's get to work!

Creating the project

The project is where all the scenes, models, materials, shaders, textures, and other assets are kept together.

At this stage, we will create the project and import the project files that are required to follow the examples in this book:

1. Launch Unity.
2. When prompted to open an existing project or create a new one, choose **Create New Project**.
3. Give the project a suitable name, such as `Shaders and Effects`, and assign a location on your computer's hard drive from where it can easily be accessed.
4. Keep the default **3D** option.
5. Choose the `Effects` assets package.
6. Click on the **Create project** button.

Unity will take a moment to copy the appropriate files. When it is finished, you will have a new blank space in the **Scene** view.

The next step is to import the project files needed for the chapters.

Downloading the example code
Detailed steps to download the code bundle are mentioned in the Preface of this book. Please have a look.
The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Mastering-Unity-Shaders-And-Effects`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Importing the project files

The project files for this book have been saved as a Unity assets package that contains all the models and textures needed to follow the tasks in the chapters. Follow these steps to import the project files:

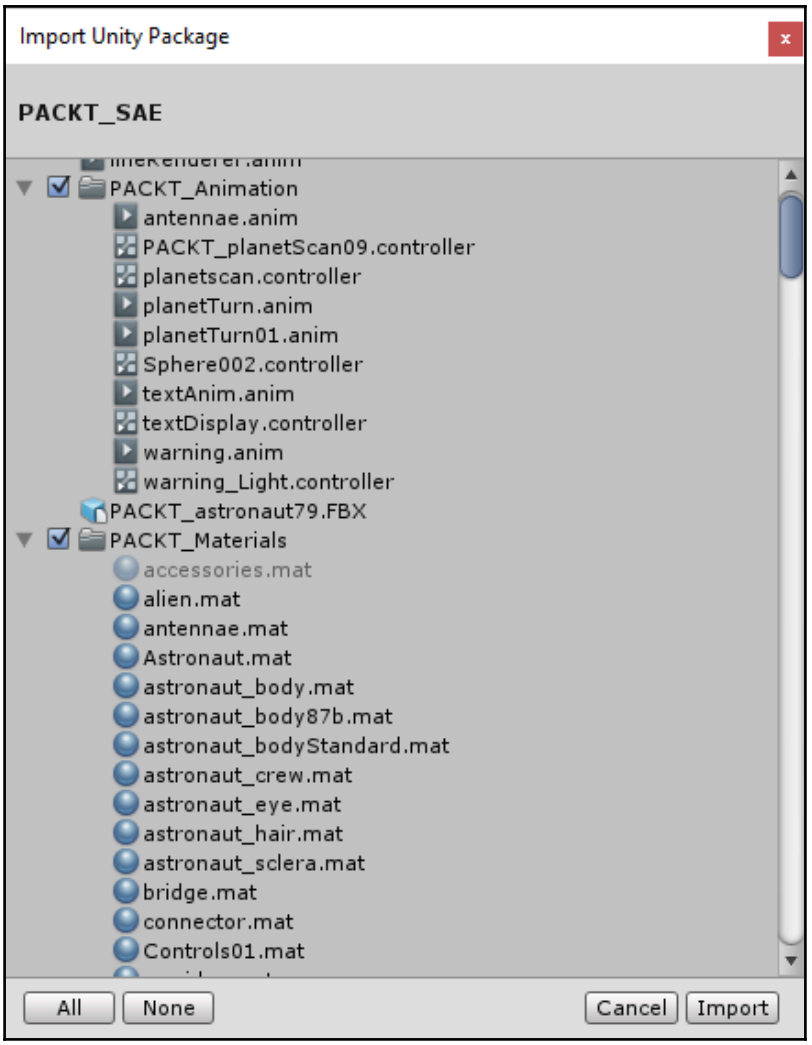
1. On the menu bar in the main Unity interface, click on the **Assets** tab.
2. Choose **Import Package** | **Custom Package…** from the list that appears.
3. In the window that appears, navigate to the location where you saved and unzipped the `PACKT_SAE` zip file.

   Within the folder, there will be a `PACKT_SAE.unitypackage` file.

4. Select it.

   Unity will take a moment or two to decompress the package.

The contents of the package will then be displayed as a list in the **Import Unity Package** dialogue:



Unity's package import

All assets will be selected by default.

5. Click on the **Import** button.

When the process is finished, there should be a number of project folders visible in the **Project** panel.

In the next step, we will load our first scene.

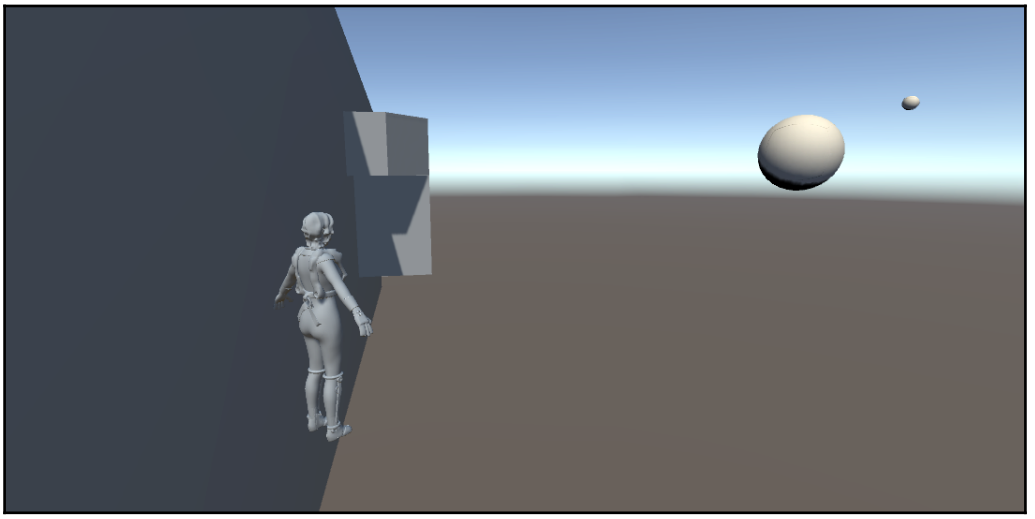
Loading and navigating the spacecraft maintenance scene

Our first scene represents a location near to the planet Ridley VI. Our astronaut has stopped to perform some essential repairs before exploring the planet.

Here, we will get comfortable working with Unity's Standard Shaders, setting up specialized materials with unique qualities for the various surfaces they will represent:

1. In the **Project** panel, locate the `PACKT_Scenes` folder and click on it once in order to view its contents in the **Assets** panel.
2. In the **Assets** panel, locate `Chapter1_Start`.
3. Double-click on the asset to load this scene in the Unity project.

The scene will become visible in the **Scene** view:



The initial state of the scene

The scene consists of the spacecraft, astronaut, and distant planet with its moon.

Currently, all the models have a basic, default material applied to them.

The scene is already lit with Unity's default scene lighting, so we will be able to compare the way different materials respond to light when we set up our Standard Shaders.

In the next step, we will start with the astronaut's material.

Creating the astronaut material

The astronaut model has a number of different surfaces—her face, glass helmet, and spacesuit, which has metal and fabric sections. Unity's Standard Shader can handle these differences, using sliders or texture values to determine the metallic and shiny surfaces.

We will start by creating the material asset:

1. In the **Project** panel, click on the `PACKT_Materials` folder to view its contents in the **Assets** panel.
2. Right-click on an empty area of the **Assets** panel and choose **Create** | **Material** from the drop-down list that appears.

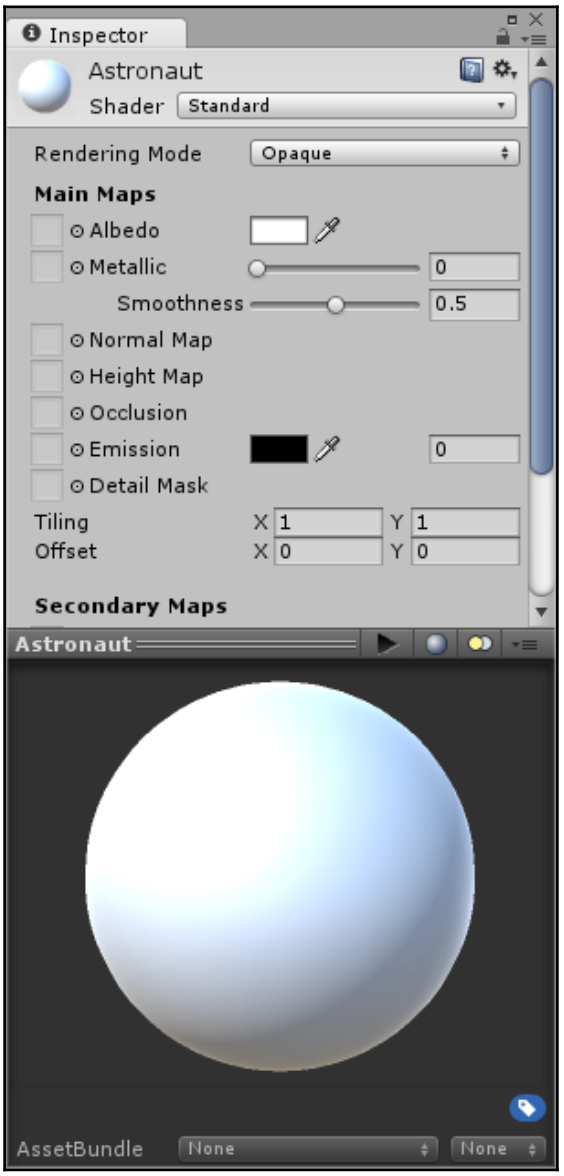
   The material asset will be created in the `PACKT_Materials` folder.

3. Rename it `Astronaut`.

   By default, in Unity 5, new materials are created using Standard Shader, which uses a metallic workflow.

   This is a versatile shader that can be used for many different surfaces.

4. Click on the `Astronaut` material in the **Assets** panel to view its properties in the **Inspector**:



Default material settings

In its blank state, the `Astronaut` material has a white **Albedo** color and some amount of **Smoothness** that will make the surface appear slightly shiny.

5. Click on the white rectangle next to the **Albedo** slot in the **Inspector** and choose a color other than white.

   Setting a temporary color will allow us to see where the material is used in the scene before we assign more texture maps to it.

   At the top of the center of the main Unity interface, switch to the **Scene** view by clicking the **Scene** tab.

   This will allow us to get a better view of the parts of the scene that we are working on.

6. Focus in on the astronaut character using the camera controls: *Alt* + LMB to rotate, *Alt* + MMB to pan, and *Alt* + RMB to zoom.
7. Drag the `Astronaut` material from the **Assets** panel onto the `Astronaut` model in the **Scene** view and release.

   The model's appearance should change, indicating that it now uses the `Astronaut` material:

The Astronaut material applied in the Scene view

Now, we will be able to see the changes that we make to the material on the model in the **Scene** view.

8. In the **Project** panel, click on the `PACKT_Textures` folder to view its contents in the **Assets** panel.
9. In the **Assets** panel, locate the `astronaut_suit` subfolder.

   This is where all the models textures are located.

10. Within the `astronaut_suit` subfolder, locate the `astronaut_albedo` texture.
11. Drag this to the **Albedo** slot in the **Inspector**.

    This texture contains the flat surface color information for the astronaut. This is a **TGA** file without an alpha channel.

12. Return the Albedo color to white by clicking on the rectangle and choosing pure white from the color picker:

The Astronaut mode with the albedo texture applied

The different parts of the model such as the face and fabric currently respond to the scene's lighting in the same way.

The **metallicity** and **smoothness** are uniform as these values are currently defined by value sliders in the `Astronaut` material.

13. In the **Assets** panel, locate `astronaut_metal` and drag this into the **Metallic** slot in the **Inspector**.

    The astronaut's appearance will change again in the **Scene** view.

In metallic maps, the **metallicity** or **metalness** value is defined in the red channel of an RBGA texture file. Surfaces that are more metallic have a higher red value. Other channels, such as green and blue, are ignored. Note that when we dragged the `astronaut_metal` file into the material, the smoothness slider disappeared.
Smoothness is defined in the alpha channel of the metallic texture. When this is not present, the material is assigned full smoothness.

14. In the **Assets** panel, locate `astronaut_normal` and drag this to the **Normal Map** slot in the **Inspector**.

A **Normal Map** overrides the surface normals of a model defining the direction in which it will reflect light. This technique is commonly used to fake high-resolution detail on a model's surface. The **Normal Map** slot uses an RGB map, with each of the channels used to define a direction of the normal's surface.

15. Locate `astronaut_suit_ao` and drag this to the **Occlusion** slot in the **Inspector**.

Occlusion or **ambient occlusion** is a lighting effect in which recessed parts of a model such as cavities and folds in clothing that never receive full light are made to appear darker. Used together with other maps, such as **Normal** and **Metallic,** this will help a complex model to have a more realistic appearance. In the Standard Shader, Occlusion uses a grayscale RGB map.

By default, the material uses full ambient occlusion when a map is applied. This may be a little strong for our current lighting set up, so we will reduce the value.

16. In the numerical field at the end of the **Occlusion** row in the **Inspector**, set the value to `0.5`.

    This will slightly reduce the ambient occlusion effect.

    In the **Scene** view, the astronaut should now appear to have a realistic set of surfaces:

The Astronaut's character with all texture maps applied to the main material

Parts of the character still use the default gray material, as they have been set up to use different material sets or consist of different submodels.

Character eyes are often separate models, so they can be animated separately from the main character mesh. In this case, our eyes share the same material as the main character model.

In the next step, we will assign the `Astronaut` material to the eye models:

1. In the **Hierarchy** view, click on the small arrow next to the `astronaut` game object.

   This will allow us to view any child game objects that the astronaut contains.

2. Drag the `Astronaut` material to the `eye_L` object, and then the `eye_R` object to apply it to them:

The Astronaut's character's eyes assigned with the main material

The astronaut has an additional material group assigned to her harness and metal equipment. This has its own texture set.

More complex models, such as the character, often use more than one material group.

This will allow different surfaces in a model to use different materials and respond differently to light.

Let's set up this material in the next step.

Creating a material for the astronaut's accessories

At this point, we need to create the astronaut's secondary material used in her harness and other equipment.

We will start by creating a new material in the same way:

1. In the **Project** panel, click to select the `PACKT_Materials` folder.

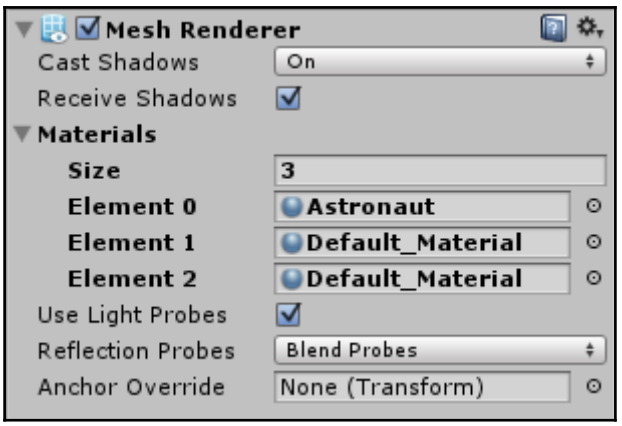
   The folders contents will appear in the **Assets** panel.

2. In an empty area, create a new material. Right-click and choose **Create** | **Material** from the drop-down list.
3. Rename the material `Astronaut_equipment`.
4. In the **Hierarchy** panel, select the `astronaut` game object.

   Its components will show up in the **Inspector**.

5. Scroll down until the astronaut game object's **Mesh Renderer** is visible in the **Inspector**.

   In the **Materials** section, the **Mesh Renderer** should currently be displaying an array of three material elements:



The astronaut game object's Mesh Renderer component visible in the Inspector

6. Drag the `Astronaut_equipment` material from the **Assets** panel to the **Element 1** position.

This will replace the gray default material in the array and display a different material on the astronaut's equipment in the **Scene** view.

Now that the material is assigned, let's add the correct texture maps.

7. Then, in the **Inspector**, expand the `Astronaut_equipment` material by clicking on the small arrow next to its name.

   The map inputs and sliders should now be visible.

8. In the **Project** panel, click on the `PACKT_Textures` folder.

   Its contents will appear in the **Assets** panel.

9. Click on the `astronaut_suit` subfolder again to view the character texture assets.
10. Locate the `equipment_albedo` texture asset.
11. Drag this into the **Albedo** slot in the **Inspector**.

    The astronaut's harness will change color in the Scene view, indicating that the texture map is being used in the material:

The astronaut's equipment displayed with an albedo texture assigned

12. Assign the **Metallic**, **Normal**, and **Occlusion** maps to the `Astronaut_equipment` material in the same way.
13. After assigning the **Occlusion** map, make sure to set the value to `0.5`, so that the equipment part of the model uses the same level as the character's spacesuit.

    We will assign an extra texture map to this material. The astronaut's harness has some lights, and we can use an **Emissive** texture to brighten these up.

14. Locate the `equipment_emissive` texture.
15. Drag this into the **Emission** slot in the `Astronaut_equipment` material in the **Inspector**.

    In the **Scene** view, the lights in the character's harness should now be lit up:

The astronaut's equipment is complete with metallic, normal, occlusion, and emissive maps

The Standard Shader's **Emission** slot takes an RGB map. This can then be filtered with an additional color in material and further modified with a value. This feature of Standard Shader will be covered in detail in `Chapter 3`, *Working with Light and Light-Emitting Surfaces*.

The third material slot in the array is used for the character's hair. This material uses the same workflow and arrangement of texture maps and has been set up already.

16. You can find the `Astronaut_hair` material in the `Final_Materials` subfolder in the `PACKT_Materials` folder.

You will also find final versions of the `Astronaut` and `Astronaut_equipment` materials in the `Final_Materials` subfolder for reference.

17. Select the `astronaut` game object in the **Hierarchy** view.
18. Drag the `Astronaut_hair` material into the **Element 2** slot in the material's array under the `astronaut` game object's **Mesh Renderer** component:

The astronaut's equipment complete with metallic, normal, and emissive maps

Before we move on, there is one more step. The astronaut's spacesuit has a transparent helmet, which we have hidden while we set up the main material. We will deal with this next.

Making objects transparent

Models can have different materials assigned to them as material groups during the modeling process.

We have already seen this with the astronaut's equipment, which uses a different material with an emissive quality.

In this case, the astronaut's transparent helmet has been laid out in its own UV shell and with its own material group, allowing us to assign a unique material to it in Unity.

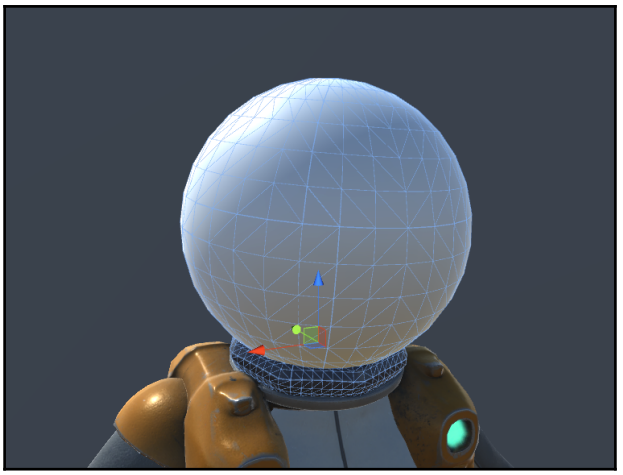
The helmet model is parented to the astronaut game object, and currently hidden in the scene.

We will make it visible in the next step, so we can see the changes that we make to its appearance:

1. In the **Hierarchy** panel, select the `helmet` game object.

   You may have to expand the astronaut game object's hierarchy by clicking on the small arrow next to its name to see its child game objects.

2. Activate the `helmet` game object by clicking on the checkbox next to its name in the **Inspector**:



The Astronaut's helmet game object

The helmet mesh contains two material groups, shown further down in the **Mesh Renderer** component as an array of material elements.

The `Astronaut_equipment` material is in the first position, which defines the non-transparent rim of the helmet.

The second position in the array will define the transparent part of the helmet. This currently uses the default gray material.

In this step, we will create the material, once again, using the Standard Shader:

1. In the **Project** panel, select the `PACKT_Materials` folder.
2. When its contents become visible in the **Assets** panel, create a new material by right-clicking on an empty space and choosing **Create** | **Material** from the drop-down list.
3. Rename the new material `Helmet`.
4. Drag the `Helmet` material to the **Element 1** slot in the `helmet` game object's **Mesh Renderer** component in the **Inspector** to assign it:

The new material assigned to the helmet game object

5. Click on `Helmet` in the **Assets** panel to show its parameters in the **Inspector**.

   We can leave most of the material's properties at their default values, but we will need to change the **Rendering Mode**.

6. Click on the **Rendering Mode** drop-down list and choose **Transparent**.
7. Click on the white rectangle next to **Albedo**, and when the color picker appears, set the **A** value to `92`.

The alpha channel's opacity is defined on a scale of 0 to `255`, the same as the individual RGB values used in the color.

In this case, we want the helmet to be visible, but transparent.

A texture map can also be used to define the transparent qualities of an object. We will cover this in detail later in `Chapter 5`, *Exploring Transparent Surfaces and Effects*.

In the **Scene** view, the astronaut's helmet should be slightly transparent.

Now that it is assigned to the astronaut, we will make one more adjustment to the `Helmet` material:

1. Click on the `Helmet` material once more in the **Assets** panel.
2. In the **Inspector**, drag the **Smoothness** slider until the value is `0.65`.

The helmet will appear slightly shiny in the **Scene** view:

The character's materials are completed

In the next step, we will create a material for the spacecraft using some more advanced Standard Shader features.

Creating the spacecraft material

When we set up the `Astronaut` material, we assigned a single albedo, metallic, and normal map to define these qualities for the complete model.

We have a slightly different challenge for the spacecraft. The spacecraft model takes up a relatively large space and is a backdrop for the character. To make sure that the surface has sufficient detail, we will add an additional material to the object's Mesh Renderer:

1. In the **Project** panel, click on the `PACKT_Materials` folder to view its contents in the **Inspector**.
2. In an empty area of the **Assets** panel, right-click and choose **Create** | **Material**.
3. Rename the new material `Spacecraft`.
4. Assign the `Spacecraft` material to the `spacecraft_surface` model in the **Scene** view by dragging it to the surface, in the same way that we did for the astronaut model.
5. In the **Project** panel, click on the `PACKT_Textures` folder to view its contents in the **Assets** panel.
6. Locate the `spacecraft_surface` subfolder and click on it to view its contents.
7. Locate `spacecraft_albedo`, `spacecraft_metal`, `spacecraft_ao`, and `spacecraft_normal` and drag them to the appropriate slots in the `Spacecraft` material in the **Inspector**.

These maps should convey the overall surface and volume of the spacecraft in the **Scene** view:

The spaceCraft_surface game object with its material assigned

Next, we will create another material that we will layer on top of the `Spacecraft` material that we have just assigned.

Creating the spacecraft's decal material

We will follow the same process to create the additional material:

1. In the **Project** panel, click on the `PACKT_Materials` folder to view its contents in the **Assets** panel.
2. Create a new material in an empty part of the **Assets** panel. Right-click and choose **Create** | **Material** from the drop-down list.
3. Rename the new material `Spacecraft_decal`.

   This time, we want the material to be transparent.

4. In the **Inspector**, set the material's **Rendering Mode** to `Transparent`.

5. In the **Project** panel, click on the `PACKT_Textures` folder to view this folder's contents in the **Assets** panel.
6. In the **Assets** panel, click on the `spacecraft_surface` subfolder to view its contents.
7. Locate the `spacecraft_decal` texture asset and drag it to the `Spacecraft_decal` material's **Albedo** slot.

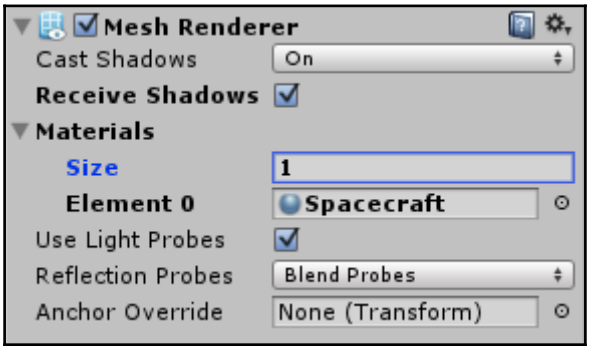
   This is a transparent texture with an alpha map.

   We will assign the new material to the spacecraft surface model in the next step.

8. In the **Hierarchy** panel, click on the `spaceCraft_surface` game object.

   This object's components will become visible in the **Inspector**.

9. In the **Mesh Renderer** component, locate the **Materials** array:



The spaceCraft_surface game object's default Mesh Renderer parameters

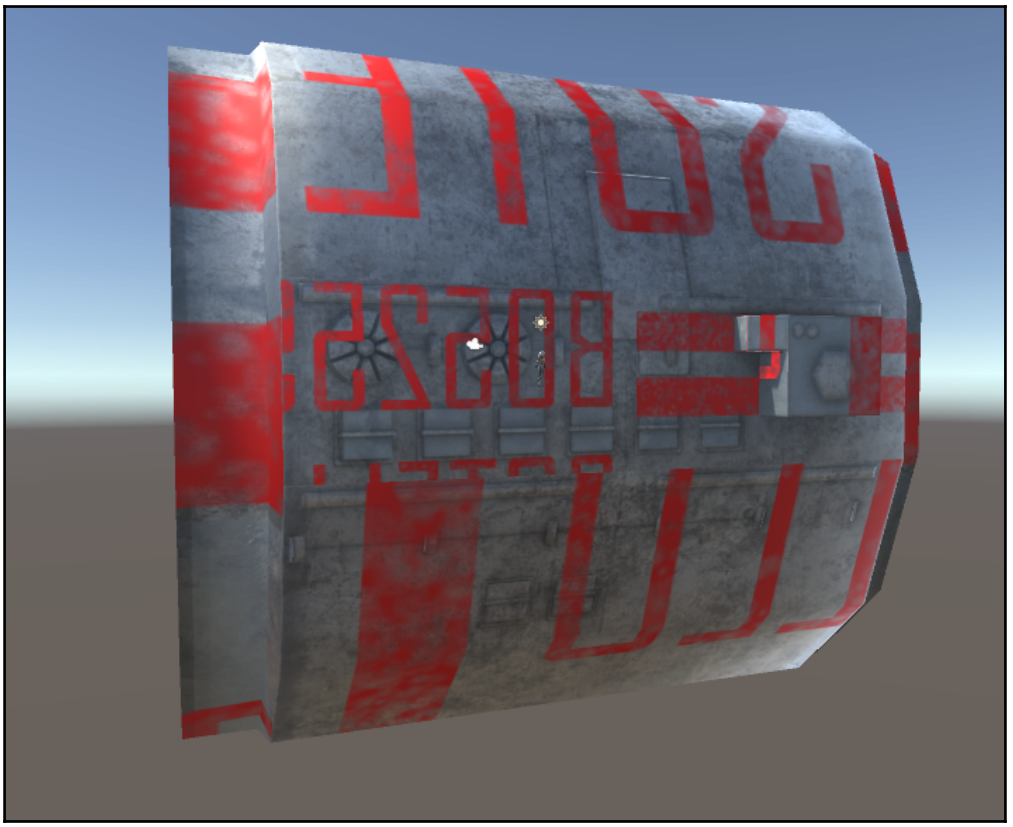
10. Increase the **Size** value to `2` by entering this number in the field.

    A new material slot will become available within the array.

11. Click on the `Spacecraft` material in the **Element 0** slot to open its location in the **Assets** panel.

12. Drag the `Spacecraft_decal` material into the second **Material** slot in the **Inspector**.

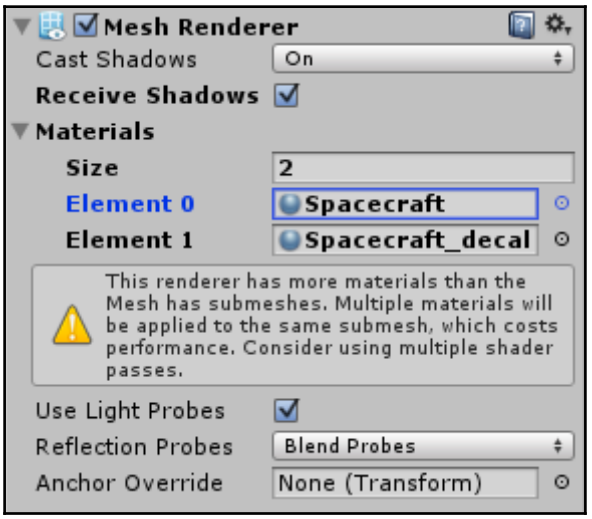
    In the **Scene** view, the decal texture will appear superimposed over the spacecraft's surface:



The spacecraft's surface with its second material added

When we assign additional materials to the meshes that do not have more than one material group, materials are layered over each other. This is only really apparent if the upper materials use the transparent, cutout, or fade Rendering Modes.

Note that a warning message has appeared beneath the **Materials** array in the **Inspector**:



The adjusted Materials array with the warning message

Combining materials in this way is not efficient for performance.

We will cover more efficient methods of blending textures in a custom shader in `Chapter 6`, *Working with Specular and Metallic Surfaces*.

Now that our second material is showing up, we need to adjust its UV parameters so that the additional detail fits nicely onto the spacecraft's surface:

1. In the **Assets** panel, click on the `Spacecraft_decal` material.

   The **Tiling** and **Offset** parameters for the Standard Shader are located directly beneath the main texture input parameters.

2. Set the material's **Tiling X** value to `-7` and **Tiling Y** value to `7`.

   This will reduce the size of the texture so that it is one-seventh of its full size.

The texture does not actually repeat as it has been clamped in its **Import Settings**.

3. Set the material's **Offset X** value to `3.03` and **Offset Y** value to `-0.34`.

   This should line up the decal correctly with the geometry and the original material:

The completely layered spacecraft material

In the next step, we will assign a material to the planet.

Creating the planet material

The distant planet of Ridley VI is an inhospitable place. Dangerous nitrogen storms rage over most of the planet's surface.

To convey this in our scene, we will set up a Standard Specular material.

Standard specular is a variant of Unity's Standard Shader that allows us to determine the specular quality of a surface, rather than defining how metallic it is.

In real terms, this gives us a little more freedom with our material. We can define the surface highlight size and strength and define a specular color different to the albedo and light color.

We will start by creating the material, as follows:

1. In the **Project** panel, click on the `PACKT_Materials` folder.

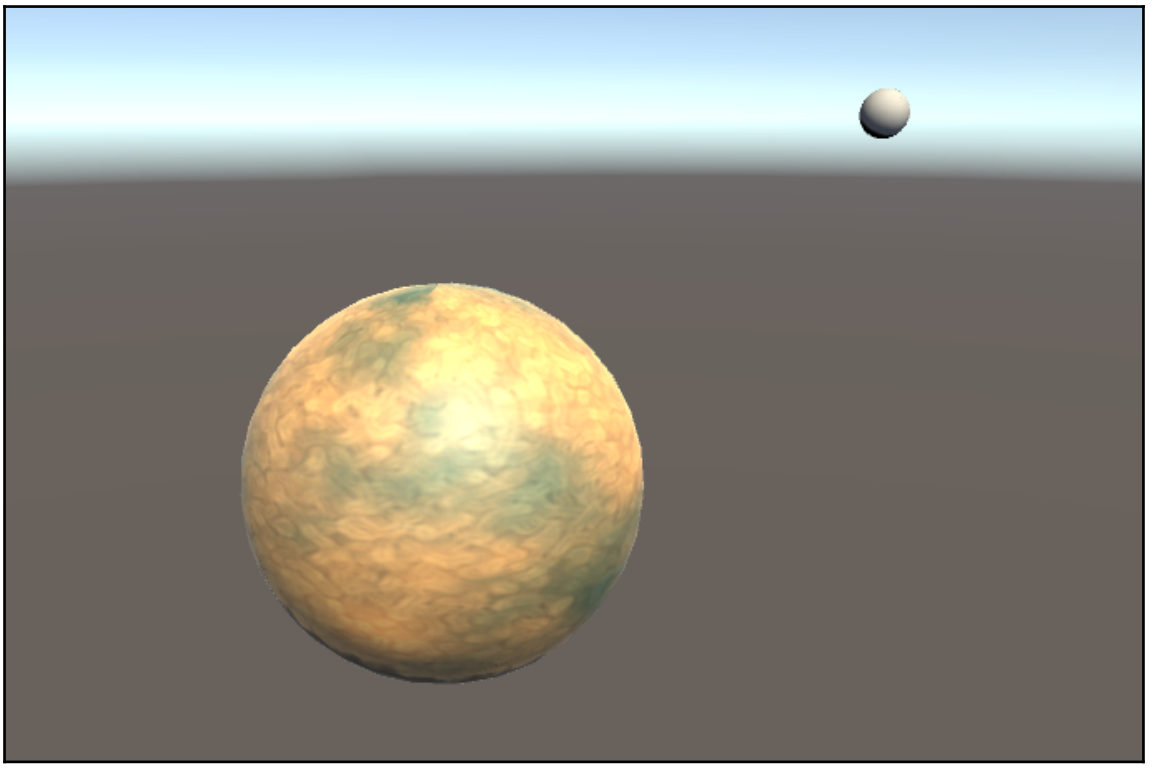
   Its contents will appear in the **Assets** panel.

2. In an empty area, create a new material by right-clicking and choosing **Create** | **Material** from the drop-down menu.
3. Name the material `Planet`.
4. In the **Assets** panel, click on `Planet` once in order to view its properties in the **Inspector**.
5. In the **Project** panel, click on the `PACKT_Textures` folder to view its contents in the **Assets** panel.
6. Locate `planet001` and drag this to the **Albedo** slot in the `Planet` material in the **Inspector**.

   This is the first of the sequence of images used to represent Ridley VI's swirling surface.

7. In the **Hierarchy** panel, double-click on the `ridleyVI` game object to select it and frame it in the **Scene** view.
8. Return to the `PACKT_Materials` folder by clicking on it in the **Project** panel.
9. Drag the `Planet` material to the planet in the **Scene** view.

The appearance of the planet will change when the material is applied:



Ridley VI with an albedo texture applied

There are few more adjustments that we need to make to get the surface look more like a planet.

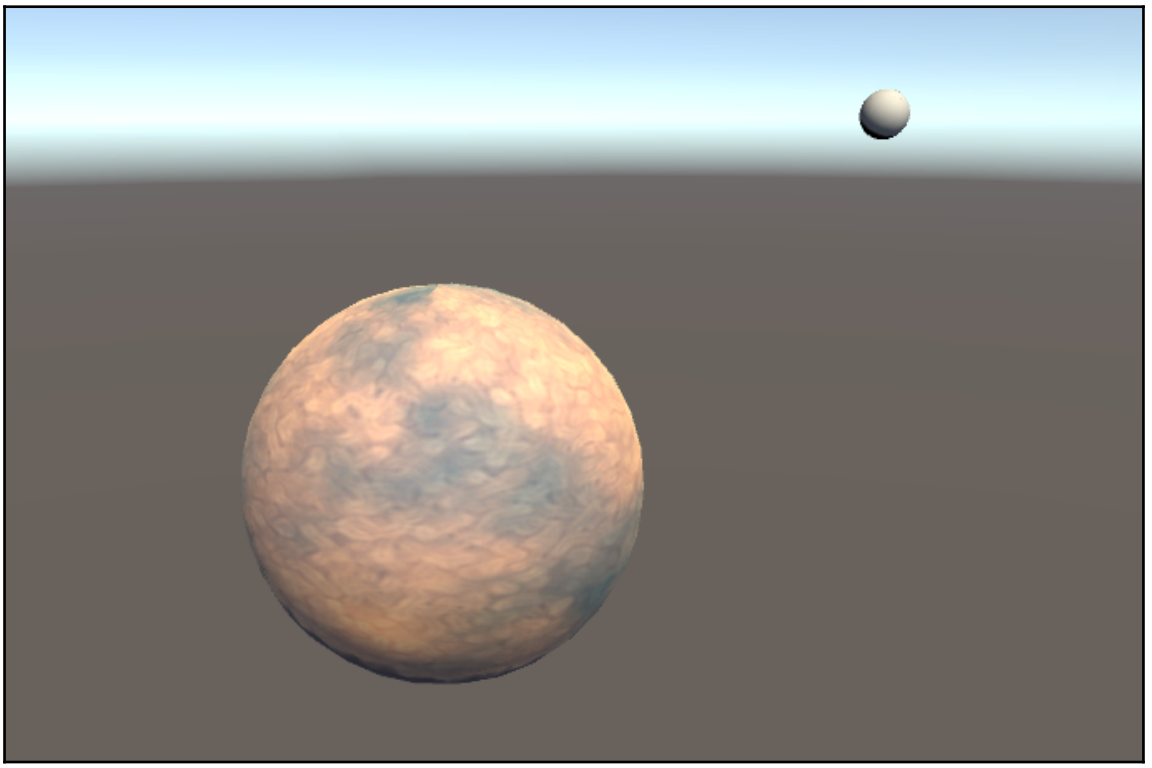
10. Click on the `Planet` material once again in the **Assets** panel.
11. In the **Inspector**, set the **Shader** type to **Standard (Specular setup)**.

    Similar to the Standard Shader, **Standard Specular** allows us to define a material's surface color (Albedo), reflectivity, and shininess (Smoothness and specular).

    The key difference is that it uses the full RGB to define a specular value, allowing us to assign a reflected color to the surface.

12. Click on the gray rectangle next to the **Specular** setting in the material and set the RGB values to `67`, `63`, and `92` to define a slightly blue specularity.
13. Drag the **Smoothness** slider to `0.1`.

    Decreasing the **Smoothness** value will increase the size of the highlight, giving the planet a more diffused appearance:



The planet's surface with a defined specular color and smoothness value

This will become more noticeable when we add the skybox and adjust the scene lighting.

In the next step, we will set up the scene's skybox.

Setting up the skybox

The scene currently uses Unity's default skybox, which defined the colors for the sky and ground.

This is more suited to a terrestrial landscape than a space scene, so we will create a new skybox to fit our scene, as follows:

1. In the **Project** panel, click on the `PACKT_Materials` folder to view its contents in the **Assets** panel.
2. In an empty area, right-click and choose **Create** | **Material** from the drop-down list.
3. Name the material `Starfield`.

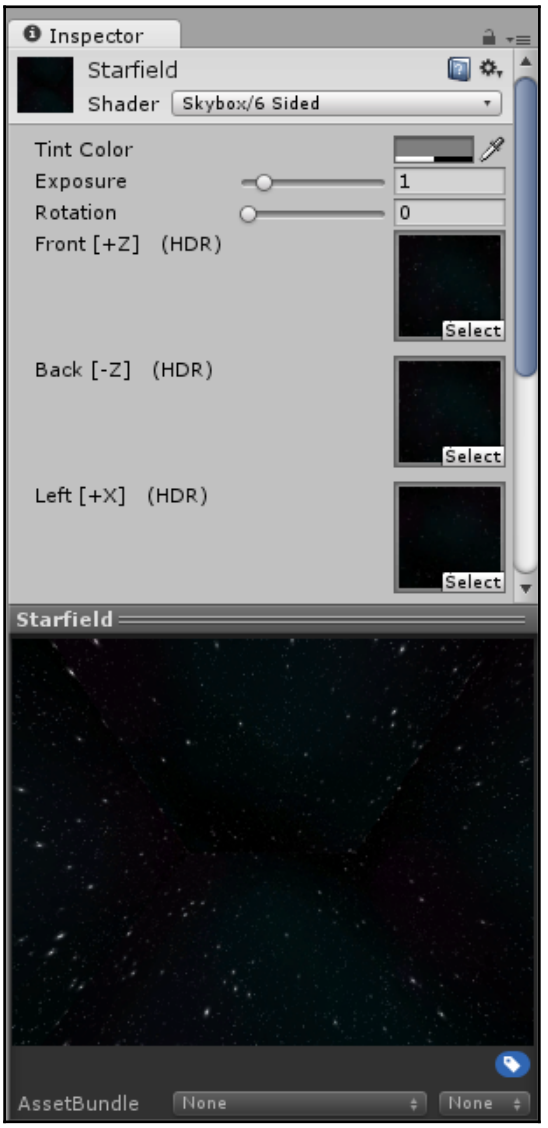
    Unity has a set of unique shaders used for skyboxes. We will choose an appropriate option in the next step.

4. In the **Inspector**, click on the **Shader** drop-down list and choose **Skybox** | **6 Sided**.

    The six-sided skybox shader has space for six individual texture maps, defining each of the directions.

5. In the **Project** panel, click on the `PACKT_Textures` folder to view its contents in the **Assets** panel.
6. In the **Assets** panel, locate `starfield_front` and drag this to the first slot in the `Starfield` material in the **Inspector**.
7. Repeat this for the five other `starfield` textures to assign each to the correct slot in the material.

A visualization of the assembled skybox will be visible in the preview panel at the bottom of the Inspector. Any misplaced textures will be distinguished by mismatched seams:



Skybox material assembly

In Unity 5, the skybox is defined in the **Lighting** panel, which is usually hidden by default.

8. To open it, click on **Window** | **Lighting** in the menu bar.

    The **Lighting** window will appear over the main Unity interface.

9. Dock the **Lighting** window next to the **Inspector**, or put it in some other convenient location that will still allow you to see the **Scene** view at the same time.
10. In the **Project** panel, click on the `PACKT_Materials` folder.
11. When its contents appear in the **Assets** panel, drag the `Starfield` material to the **Skybox** slot, replacing `Default-Skybox`.

    The starfield should now be visible in the scene:

The starfield skybox added to the scene

The light areas of the skybox contribute to the scene lighting in Unity's default lighting set up. We can adjust the exposure and rotation of the skybox textures at the material level in the **Inspector**.

In the next step, we will adjust the scene lighting and add some additional effects.

Adjusting the scene lighting and adding effects

For this scene, we can stick with the default`Directional Light`, which provides real-time shadows. We need to adjust its position and rotation so that it appears more like a sun behind the planet:

1. In the **Hierarchy** panel, select the `Directional Light` game object.

   The object's parameters will become visible in the **Inspector**.

2. In the **Transform** component at the top of the **Inspector**, set the **Position X** value to `10.8`, **Position Y** to `15.75`, and **Position Z** to `97`.

   This will position the light roughly behind the planet:

The repositioned scene light

Next, we will rotate the light to point towards the astronaut and her spacecraft.

3. Back in the **Inspector**, set the **Rotation X** value to `8.7`, **Rotation Y** value to `-174`, and **Rotation Z** value to `-153`.

   This will result in the light pointing towards the astronaut and illuminating one edge of the planet:

The rotated scene light

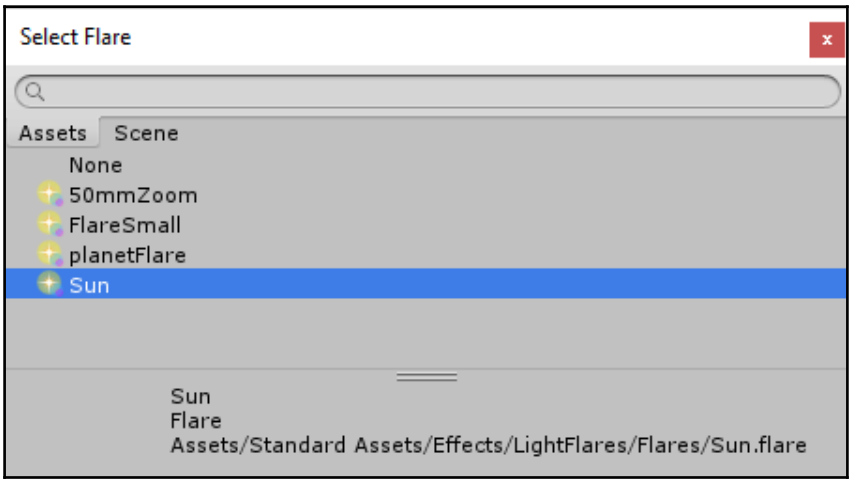
We can add some extra effects here to get our scene looking more cinematic.

Further down in the **Inspector**, check the **Draw Halo** button.

This creates a light area at the point where we have positioned the light, making it visible in the scene.

Click on the radio button next to **Flare**.

In the selection list that appears, choose **Sun** from the list of available flare effects:



Selection of Standard Asset flares

In the **Scene** view, the flare will appear to be centered on the `Directional Light` object:

Final scene lighting

The scene lighting is complete.

We will take a closer look at the lighting properties and effects in `Chapter 3`, *Working with Light and Light-Emitting Surfaces*.

Summary

In this chapter, we utilized different qualities of Unity's Standard Shader to achieve a variety of results in our space scene.

First, we demonstrated the basic functions of Standard Shader, applying appropriate texture maps to create materials for the astronaut character. We used the transparent option in another standard material to make the astronaut's helmet transparent.

For the spacecraft's material, we explored an additional layer technique in order to blend together different surface qualities.

We introduced the Standard specular workflow and used a specular material for the planet, defining colored specularity on the planet's surface.

After this, we created a custom skybox as the backdrop for our scene, adjusted scene lighting, and then added some appealing effects to the scene.

In the next chapter, we will take a step beyond the Standard Shader, adding animation and additional effects.

2
Creating Custom Shaders

In this chapter, we will continue with our spacecraft repair scene, adding our first custom shaders to increase the realism of the scene.

We will explore the interaction between material surfaces and our existing scene lights and how we can use the properties of Unity's physically-based shaders to improve the appearance of the game.

We will cover the following topics in this chapter:

- Creating a basic Unity shader from scratch
- Shader property types
- Testing a shader in our scene
- Incorporating Cg snippets
- Creating and using multi-pass shaders for transparent effects
- Creating a custom atmosphere shader

Opening the project

If you have not already done so, download and install the shaders and effects project ZIP from the Packt's website. This file contains the Unity asset package containing all the scene assets used throughout this book.

In `Chapter 1`, *Getting to Grips with Standard Shaders*, we set up **Standard Surface** materials for most of the surfaces in our space scene. If you skipped this stage and have not set up the Unity project, you may want to do this so that you can follow the examples in this chapter.

One surface that we did not create a new material for is the planet's moon. We will start by creating a very basic shader to use with the moon in our scene.

This will be a good opportunity to see a shader work as we create it.

Opening the scene

The scene file for this chapter contains the models and materials that we will be working on.

In the Unity project, load the `Chapter2_Start` scene. From the menu bar, select **File** | **Open**. In the dialog that appears, click to open the `PACKT_Scenes` folder and select `Chapter2_Start` from the list.

The Chapter2_Start scene initial state

We are starting at the same place we left in `Chapter 1`, *Getting to Grips with Standard Shaders*. The scene again shows the **Sotello** spacecraft as it stops for repairs on its way to the planet **Ridley VI**.

Creating our first custom shader

Unity shaders use **ShaderLab** for the main part of the shader and often Cg for the more complex functions. We will start with the main part of the shader.

We will write this shader from scratch, explaining the structure and syntax as we go:

1. In the **Project** panel, click on the `PACKT_Shaders` folder.

   The folders contents will become visible in the **Assets** panel.

The `Final_Shaders` subfolder currently contains the finalized code that you may use for reference.

2. Create a new shader by right-clicking on an empty area of the **Assets** panel.
3. Navigate to **Create** | **Shader** | **Standard Surface Shader**.
4. Rename the shader `Moon`.

   The name of the shader asset is different from its name that will appear in the shader list, we will define that in the code.

5. Double-click on `Moon` in the **Assets** panel to open it in **MonoDevelop**.

   The shader that we created contains the default code that we will delete. We will write this first shader from scratch to get a better handle on the syntax.

6. In MonoDevelop, select all the code (use the *Ctrl* + *A* hotkey combination or *command* + *A* if you are working on a Mac).
7. Delete the code *(Delete* or *command* + *Delete* on a Mac).

   We will start by defining the name of the shader that will appear in the shader list that can be selected in a material.

8. Add the following line of code:

```
Shader "PACKT/Moon" {
}
```

This defines the code as a shader and designates how it will be selected in the shader list within the project.

We can organize new shaders by adding the folder name before the name of the shader. If the folder name is not already defined by another shader, it will be created.

Here we designate the folder as `PACKT` to separate our created shaders from those provided in `Standard Assets`.

The actual name of the shader is defined here and can be different from the name of the shader file that we defined in the **Assets** panel, though this can make it more confusing when you want to go back and find a shader to make changes later on.

Next we will define the shader's properties.

Between the opening and closing curly brackets, add the following code:

```
Properties {
    _Color ("Color", Color) = (1,1,1,1)
}
```

The `Properties` block contains the raw materials of the shader, such as a defined color, diffuse texture, and a slider that can be tweaked in the material.
Properties are analogous to public variables in a regular **C#** or **UnityScript** code.

We are defining a property for `_Color`. The name of the property at the start of the line of code is usually prefixed with an underscore. This is the property name used in the main body of the shader.

The name that we put in quotation marks is the name of the property that will appear in the **Material Inspector**. Color properties are defined as a series of four decimal values: **R**, **G**, **B**, and **A**.

Here we are defining the color as full white, with the alpha at its maximum value. For our project, we can change this color here in the shader code or later when the shader is used in a material.

After the `Properties` block, add the following code:

```
SubShader {
    Pass
    {
        Color[_Color]
    }
}
```

`subShader` contains the main body of the shader and processes any input colors, textures, or other values.

All shaders include at least one `subShader`. Their primary function is to separate different shader definitions. This feature allows us to create a shader that will work on multiple devices.
When a shader is used in an application, the system will run the next best `subShader` if the first is incompatible with the GPU.
We will be looking more closely at shader compatibility in `Chapter 9`, *Optimizing Shaders for Mobile*.

`subShader` contains a single `Pass`, which is the minimum requirement for a shader such as this:

1. Add the following code inside the `Pass` brackets:

```
Color [_Color]
```

2. Save the shader.

This shader is written in **ShaderLab**, which is the standard language and syntax for Unity shaders. Later examples will also use **Cg**—a dialect of **C**. The **g** stands for graphics. This version is primarily used for shaders.

The finished code should appear as follows:

```
Shader "PACKT/Moon"
{
    Properties {
        _Color ("Color", Color) = (1,0.5,0,1)
    }

    SubShader {

        Pass
        {
            Color [_Color]
        }
    }
}
```

This is just about as simple as we can get with a shader that will work in Unity 5.

Be careful with opening/closing brackets, as missing one of these or putting it in the wrong place will prevent the shader from running.

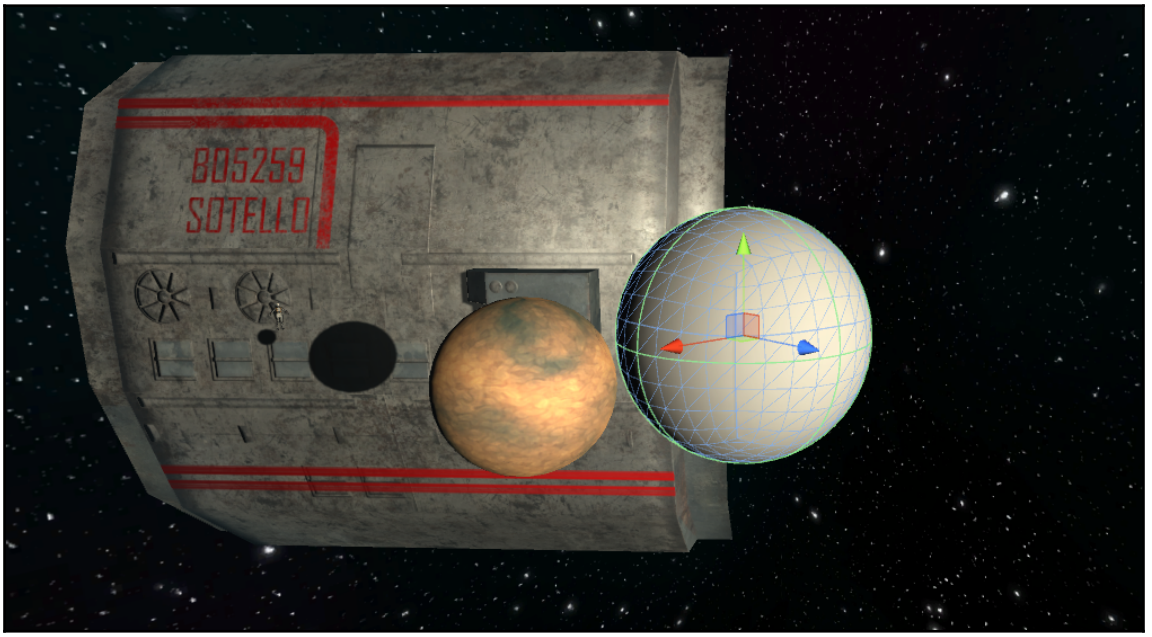
3. Minimize MonoDevelop and return to the main Unity interface.

In the next section, we will apply our newly created shader to a material and see how it looks in the scene.

Seeing the shader in action

At this point, we have created our first simple shader and want to see it on a surface in our scene. This stuff:

1. In the **Hierarchy** panel, double-click on the `smallMoon` game object to select it and zoom in on it in our scene:



The smallMoon game object in the scene

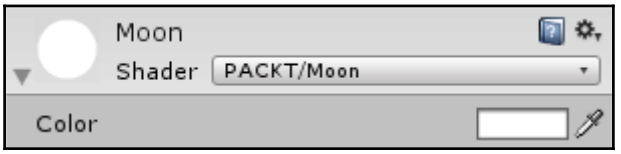
The `smallMoon` game object still uses Unity's default gray material. We need to create a unique material for it next.

2. In the **Project** panel, click on the `PACKT_Materials` folder to view its contents in the **Assets** panel.
3. Right-click on an empty area of the **Assets** panel and choose **Create** | **Material** from the drop-down list.
4. When the material asset icon appears in the **Assets** panel, rename it `Moon`.

The new material's properties will be visible in the **Inspector**.

5. Click on the **Shader** drop-down list at the top of the **Inspector**.
6. Choose **PACKT/Moon** from the list.

The material's properties will update in the **Inspector**:

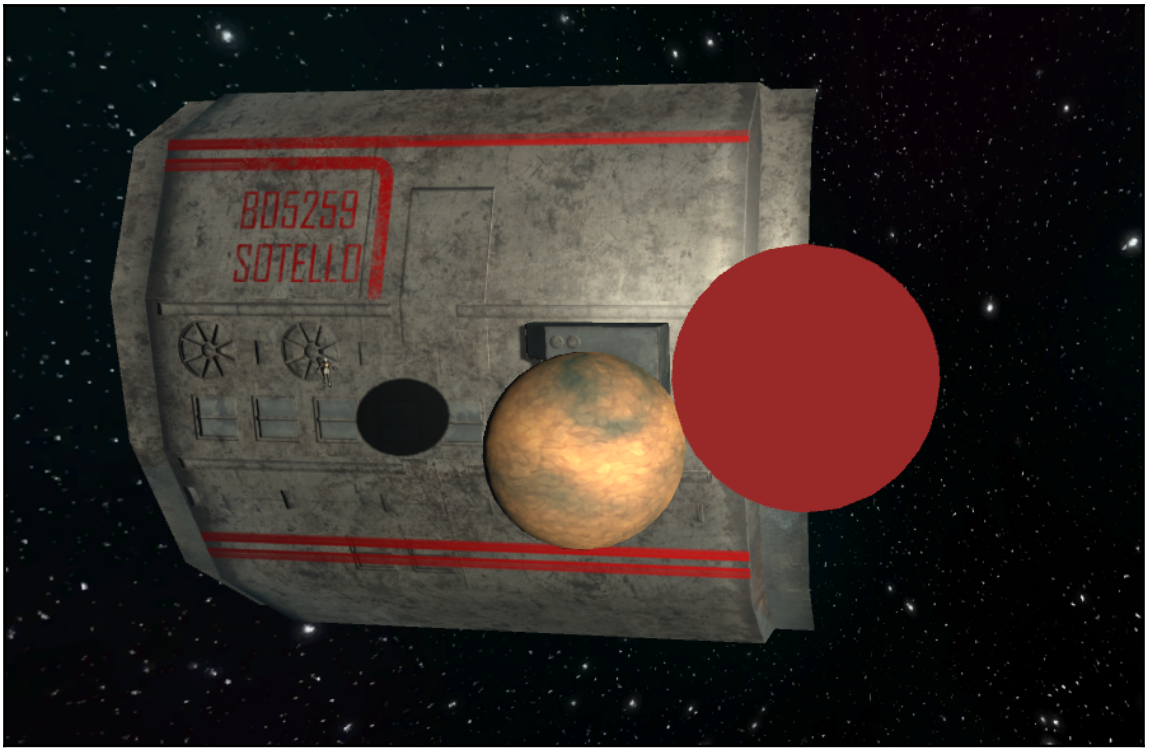


The Moon material's properties in the Inspector

The material has a single `Color` property.

7. Apply the material to the `smallMoon` game object by dragging it from the **Assets** panel onto the cube's surface.
8. In the **Inspector**, click within the **Color** swatch and define a deep red color.

9. In the **Scene** view, the moon will change color:



The Moon material with the custom shader and a color property applied

The result of the shader is a uniform colored surface that has no shadows or highlights.

This is a typical unlit shader that is very cheap on performance.

In the next step, we will add a texture to the shader.

Adding a texture to the moon shader

We can improve the complexity of the moon's surface by adding a texture map to the material. To allow this, we need to make some edits to the shader:

1. Return to MonoDevelop.

2. In the `Properties` block, near the top of the shader, add the following code:

```
_MainTex ("Albedo (RGB)", 2D) = "white" {}
```

   Here, we will define a property called `_MainTex`. This is the standard name used in Unity for a primary texture map, which is usually used for the Albedo component of a shader. The name in quotation marks is the label that will appear when we see the material in the **Inspector**.

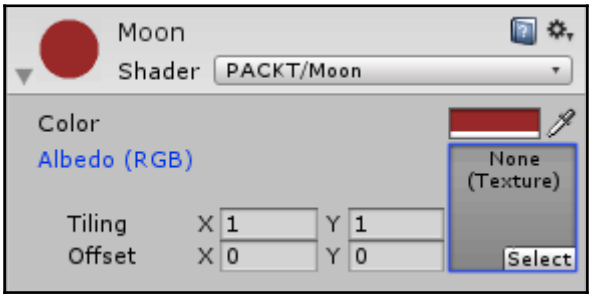
3. In the `Pass` block, add the following code after the `Color` line:

```
SetTexture [_MainTex] {
    Combine Primary * Texture
}
```

   Here, we will set the texture, calling it by its property name. We will then apply it, multiplying it over the original (or `Primary`) content.

4. Save the shader.
5. Return to the main Unity interface.

   If the shader compiles correctly, a new texture slot will have appeared in the material properties section in the **Inspector**:

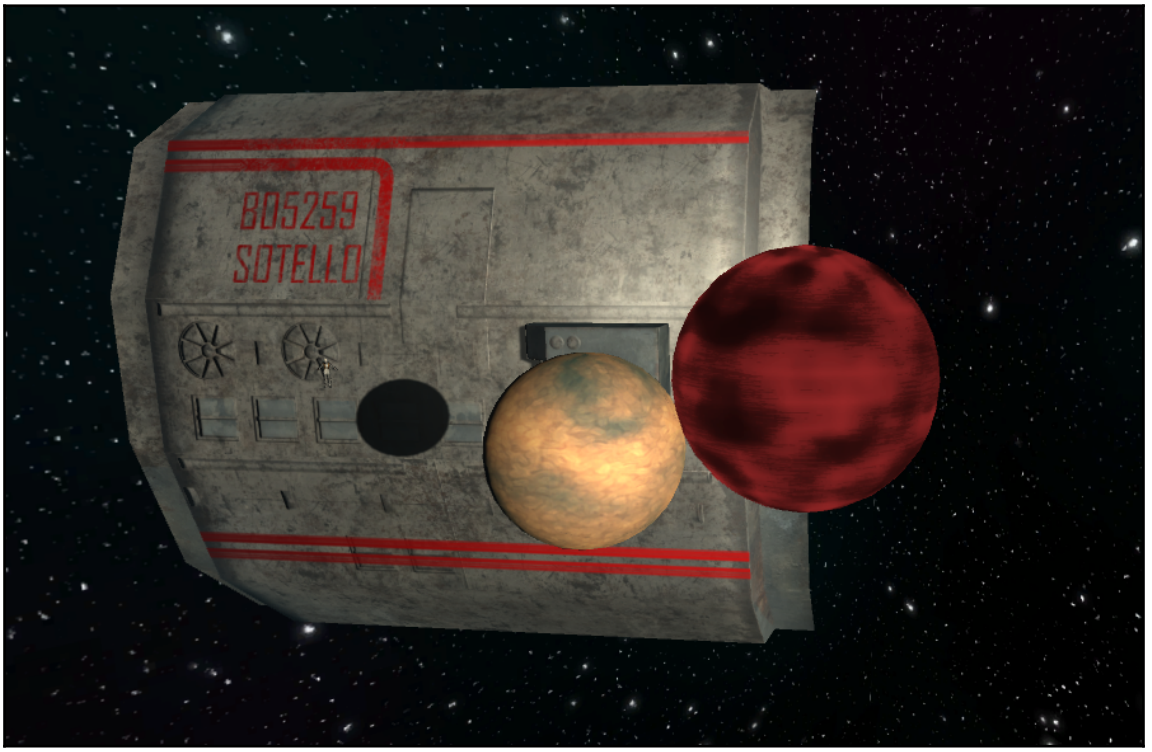


The Moon material with the new Albedo texture slot

   We can now apply the appropriate texture.

6. In the **Project** panel, click on the `PACKT_Textures` folder to open its contents in the **Assets** panel.
7. Locate the `moon_albedo` texture and drag this to the `Moon` material's **Albedo** slot in the **Inspector**.

8. In the **Scene** view, the `smallMoon` game object will update to show the texture applied:



The moon with its albedo texture applied

The bitmap texture blends with the color on the surface of the moon.

This improves the appearance of the moon, but it is still not responding to our scene lighting.

To get shadows on the surface, we need to add more code to the shader.

Making the moon shader compatible with the scene lighting

We need to make some changes to the shader code to allow our `Moon` material to respond well to the scene lighting:

1. Return to MonoDevelop.

   Most Unity shaders use Cg for their functions. In this case, this will replace our existing pass.

2. Select and delete the **Pass** block in the shader.
3. Add the following code in its place:

   ```
   CGPROGRAM
   ```

   This is the start of our Cg snippet, where we will define and process our properties.

4. Tap *Enter* a few times to create a little space before adding the following line:

   ```
   ENDCG
   ```

   This is the end of the Cg snippet. Our Cg code needs to be contained between these two tags.

   We will start adding more code between the Cg tags.

5. Add the following line directly after our opening `CGPROGRAM` tag:

   ```
   #pragma surface surf Lambert
   ```

   This line is called the **shader compilation directive**.

   A line such as this always follows the opening Cg tab. This defines the lighting model and the shader functions to be compiled when the shader runs.

   In this case, we are using the **Lambert** model, which allows shadows.

The Cg tags and the shader directive are colored magenta within MonoDevelop, distinguishing them from the rest of the code.

6. Next, add the following code:

```
struct Input {
    float2 uv_MainTex;
};
```

   The `Input` struct contains the UV data that will allow us to apply the texture to the model's surface correctly.

   The UV data is contained here as `float2`: a variable consisting of two float values.

7. Next, add the following code:

```
sampler2D _MainTex;
float4 _Color;
```

   Here, we are processing our properties into Cg variables that can be calculated in the shader. Firstly, our texture, `_MainTex`, and then `_Color`, which is a `float4` variable consisting of R, G, B, and A channel values.

   Finally, we will add the main part of the shader.

8. Add the following code:

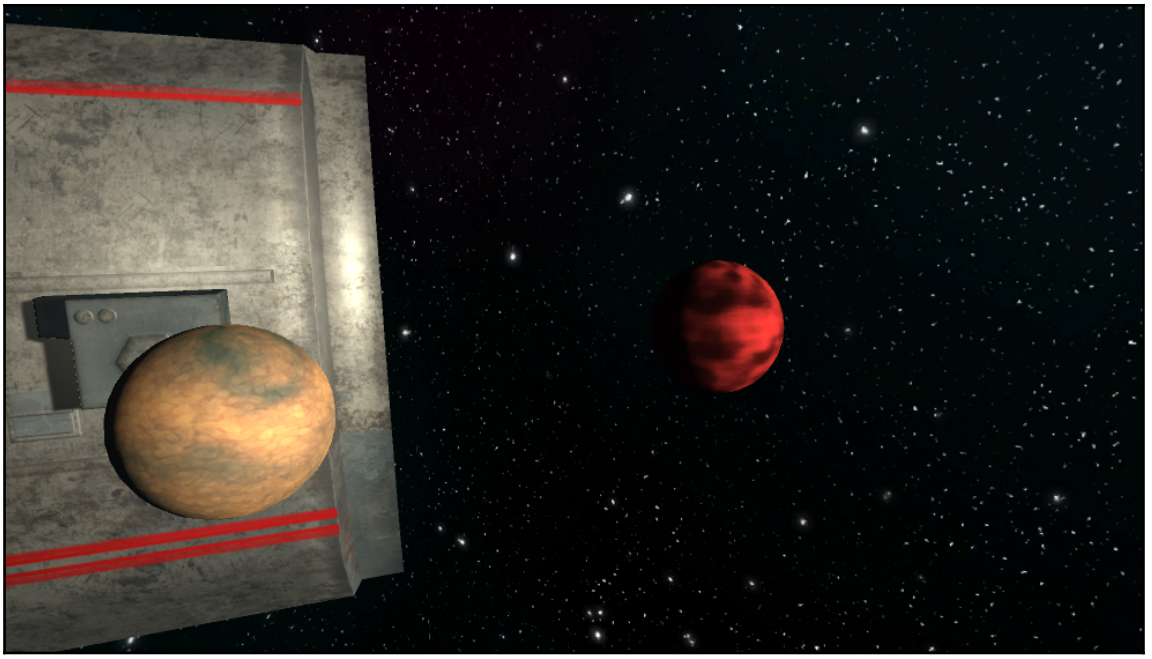
```
void surf (Input IN, inout SurfaceOutput o) {
    o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb * _Color.rgb;
}
```

   The `surf` function combines the input data and defines the shader output, denoted here with the `o` variable name.

   The only output our shader has to deal with is the `Albedo` texture, as the lighting is already defined by the lighting model that we specified.

   We will multiply the `_MainTex` `rgb` data by the `_Color` `rgb` data to get our output color.

9. Save the shader and return to the main Unity interface.



The moon with a Lambert shader applied

When the shader compiles, we should be able to see a variation in the lightness of the surface as the moon responds to the single scene light.

In the next step, we will take a look at how physically-based rendering can be used to create a more sophisticated shader that also works with the scene lighting.

We will do this by creating a custom shader for the astronaut character's transparent helmet.

Creating better transparency for the astronaut's helmet

When we set up the initial material for the character's helmet in `Chapter 1`, *Getting to Grips with Standard Shaders*, we used the transparency feature of the **Standard Shader** to render the helmet geometry partially transparent. The triangles that make up the helmet model are only facing outward, so there is no specularity on the inner surface where we would expect to see it.

This problem can be solved using a custom shader. We will build on the code structure we worked with in the previous section, making a more complex shader that is lit realistically by the scene lights.

Creating a custom transparent shader

We will start by creating a shader in the Unity project as we did in the previous section:

1. In the **Project** panel, click on the `PACKT_Shaders` folder.

   The folder's contents will become visible in the **Assets** panel.

The `Final_Shaders` subfolder contains the finalized shaders that you may use for reference.

2. Create a new shader by right-clicking on an empty area of the **Assets** panel.
3. Navigate to **Create** | **Shader** | **Standard Surface Shader**.
4. Rename the shader `Glass`.

As earlier, the name of the shader asset is different from its name that will appear in the shader list, we will define this in the code.

Next, let's use the shader in the astronaut's helmet material so that we can see how its appearance changes as we write the shader:

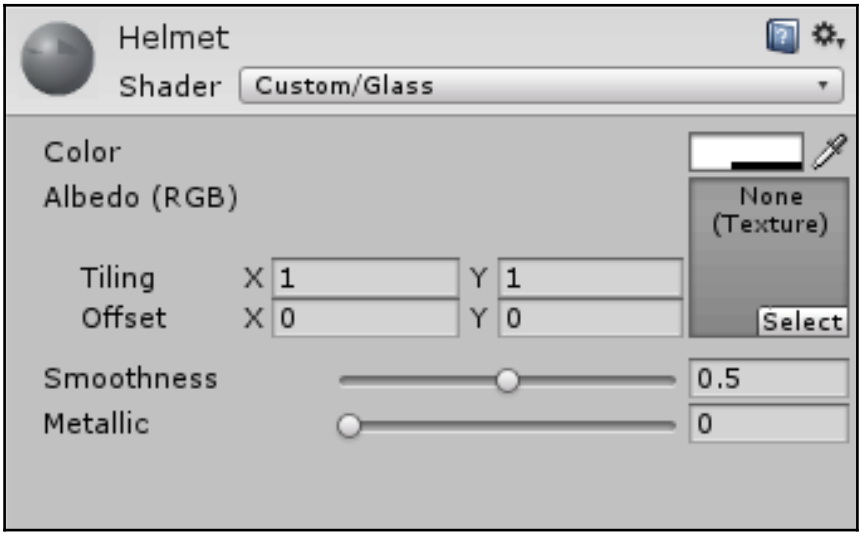
1. In the **Project** panel, click on the `PACKT_Materials` folder to view its contents in the **Assets** panel.

2. Click on the `Helmet` material.

   We assigned this material to the astronaut's helmet in `Chapter 1`, *Getting to Grips with Standard Shaders*.

   Any changes made to the material will be visible in the objects that they are applied to.

3. At the top of the **Inspector**, click on the **Shader** list and choose **Custom** | **Glass** from the drop-down list that appears:



In the **Scene** view, the astronaut's helmetshould turn opaque as the new shader's default code does not currently define the transparency.

We will fix this in the next section.

Editing the new glass shader

When it was created, the new shader was written to include the name of the shader asset, this was automatically prefixed with the word `Custom` to separate it from the existing shaders.

1. Double-click on the shader to open it in MonoDevelop.

   This time, we will go through the default code and explain the different parts as we update it.

2. Replace the first line of code with the following:

```
Shader "PACKT/Glass" {
```

   The curly bracket at the end of the line encloses the rest of the shader.

3. Save the code using the *Ctrl* + *S* shortcut (or *command* + *S* if you are working on a Mac).

   By default, we already have a shader that will respond to lighting, but the default shader code defines a surface that is opaque, rather than transparent. We need to edit the shader to allow the helmet to be transparent.

4. In `subShader`, locate the `Tags` line and replace the word `Opaque` with `Transparent`.

The `subShader` block defines different shader characteristics and will allow it to identify an alpha channel or other property that can be used to define the transparency for instance.

   A little further down is the **Cg** snippet, differentiated in MonoDevelop by its magenta color.

5. Within this, locate the shader compilation directive:

```
#pragma surface surf Standard fullforwardshadows
```

6. Add the word `alpha` at the end of the directive.

   The addition of the `alpha` keyword to the end of the directive allows the shader to compile the transparent component.

7. Save the shader.
8. Minimize MonoDevelop.

   In the main Unity scene, you may notice that the astronaut's helmet has now become transparent:

The astronaut's helmet with the custom transparent shader

The amount of transparency is defined by the alpha channel in the `Color` property that can be adjusted by clicking on the swatch in the **Inspector**.

9. Click on the **Color** swatch.
10. In the color picker that appears, set the alpha channel to a value of `128`.
11. Close the window by clicking on the **X** in the corner.
12. In the **Inspector**, set the **Smoothness** value to `0.9`.

    This will make the astronaut's helmet reflective.

13. Set the **Metallic** value to `0.188`.

    This will define a surface that is transparent and relatively shiny:

The adjusted transparent helmet material

If you look closely at the astronaut's helmet in the **Scene** view, you will notice that only the outside surface of the helmet appears to be shiny.

We can make some changes to our shader in order to make the inner surface visible, resulting in a more realistic appearance.

Creating the inner surface of the helmet

For performance reasons, Unity's shaders only render the front faces of a mesh by default. Back faces are automatically culled in Unity.

We can change this with a single line of code:

1. Back in MonoDevelop, locate the `Glass` shader's `subShader` block.
2. Add the following line just after the `LOD` line:

```
Cull Off
```

3. Save the script and return to the main Unity interface.

The inside of the Astronaut's helmet is now being rendered, but both sides are being rendered in the same place, resulting in some artifacts:

The glass shader with Cull Off

We can improve the appearance of this by rendering the front faces and back faces of the helmet separately in the shader.

Separating front and back faces

The appearance of thickness can be faked in our shader, making the astronaut's helmet seem more realistic in the scene.

We need to make some more changes to our glass material:

1. Maximize the MonoDevelop window.
2. Near the top of the shader code, add the following line after the other properties:

```
_Thickness ("Thickness", Range(-1,1)) = 0.5
```

   We will use this property to define how far the inner surface of the helmet is extruded.

3. Locate the `Cull Off` line that we recently added and replace it with the following:

```
Cull Back
```

4. Scroll down to the `ENDCG` line.

   This marks the end of the Cg snippet and will usually appear in magenta text.

5. After this, add the following code:

```
Cull Front
CGPROGRAM
#pragma surface surf Standard fullforwardshadows alpha vertex:vert
struct Input {
    float2 uv_MainTex;
};
float _Thickness;
void vert (inout appdata_full v) {
    v.vertex.xyz += v.normal * _Thickness;
}
sampler2D _MainTex;
half _Glossiness;
half _Metallic;
fixed4 _Color;

void surf (Input IN, inout SurfaceOutputStandard o) {
```

```
        fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
        o.Albedo = c.rgb;
        o.Metallic = _Metallic;
        o.Smoothness = _Glossiness;
        o.Alpha = c.a;
    }
    ENDCG
```

This is essentially an additional shader that we are using with the first and that will define the appearance of the back faces.

In the next line, we will cull the front faces so that there is no overlap between the two shader sections.

It is also possible to blend the shaders together by overlaying them. We will take a look at more complex shader functions later in the book.

Once again, we use a Cg snippet to add the property directive that defines the shader type. This directive is identical to the original shader directive with the addition of `vertex:vert` that allows us to access the vertices that the shader is applied to.

In the next line, we will declare the `_Thickness` float using the property name that we already defined on the top of the shader.

In the calculation that follows this, we will modify the position of the vertices, moving them out by the value specified in `_Thickness`.

The rest of the shader section is identical to the original section that we already defined. The result should be a glass surface with thickness that can be modified within the **Inspector**.

1. Save the script.
2. Minimize MonoDevelop.

3. In the **Inspector**, set the material's **Thickness** value to `-0.2`.

The helmet material with the inner surface applied

This technique works for thin pieces of glass, such as helmets, visors, and glasses, but it does not simulate refraction the way that would be necessary to define thicker glass and other denser transparent materials.

In the next section, we will use a variation of this technique to simulate the planet's atmosphere.

Improving the planet's atmosphere

In the first chapter, we added materials to the spacecraft repair scene using Unity's Standard Shaders. We also created a **Standard Specular** material for the planet Ridley VI to show its dramatic stormy environment.

At this point, we will write a custom shader to give it more of an atmospheric effect.

Rather than reflecting light in the same way as other solid objects, the layers of gas that surround many planets create a haze around them.

We will create something like this with a custom shader.

Creating the custom planet shader

We will start by creating another shader in the Unity project:

1. In the **Project** panel, click on the `PACKT_Shaders` folder.

   The folder's contents will become visible in the **Assets** panel.

   The folder currently contains the finalized code that you may use for reference.

2. Create a new shader by right-clicking on an empty area of the **Assets** panel.
3. Navigate to **Create** | **Shader** | **Standard Surface Shader**.
4. Rename the shader `Planet`.

As we demonstrated previously, the name of the shader asset is different from its name that will appear in the shader list, we will define this next in the code.

Applying the planet shader

We will continue by changing the shader's name and setting it up in a material in the scene so that we can view the changes that we make:

1. Double-click on the shader to open it in MonoDevelop.
2. Replace the first line of code with the following:

   ```
   Shader "PACKT/Planet_falloff" {
   ```

   This will make it easy to locate our shader when we select it in the material.

3. Save the code using the *Ctrl* + *S* shortcut (or *command* + *S* if you are working on a Mac).

4. Minimize MonoDevelop and return to the main Unity interface.
5. In the **Project** panel, select the `PACKT_Materials` folder to view its contents in the **Assets** panel.

Next, we will create a new material. This will allow us to switch between the original planet material and the new custom material to compare their appearances:

1. Right-click on an empty area of the **Assets** panel and choose **Create** | **Material**.
2. Rename the new material `Planet_falloff`.
3. Click on the new material to select it.

   When `Planet_falloff` properties appear in the **Inspector**, click on the drop-down list next to **Shader**.

4. Select **PACKT** | **Planet_falloff** to choose the shader that we just created.

   The new material has a color, single texture slot for the main texture, and sliders that define **Smoothness** and **Metallic** values.

5. Drag the `Planet_falloff` material to the `ridleyVI` game object in the **Scene** view to apply it.

You should see the planet change to the default gray color in the scene:

The planet material replaced in the scene

Now that we are using the new shader in the scene, we need to make some real changes to it.

Editing the planet shader

We will start by stripping out the elements that we do not need:

1. Maximize MonoDevelop.
2. In the `Properties` block of the shader, delete the `_Glossiness` and `_Metallic` property lines.
3. Scroll down until you see the variable definitions for these properties:

```
half _Glossiness;
half _Metallic;
```

4. Delete these lines.
5. Scroll down until you see the `surf` function that defines the shader output.
6. Delete the lines that define the `Glossiness` and `Metallic` output values.

Our shader does not need metallic or smoothness qualities. We will add new properties in the next section.

Adding new properties to the planet shader

The new properties that we add will allow us to define the depth, color, and opacity of the planet's atmosphere.

Add the following lines to the shader's `Properties` block:

```
_Thickness ("Thickness", Range(-1,1)) = 0.5
_AtmosColor (" Atmosphere Color", Color) = (1,1,1,1)
```

In the shader, `_Thickness` will define the depth of the atmosphere. The `_AtmosColor` property will be used to differentiate the atmosphere from the actual planet surface.

Like the helmet shader, the planet shader will use an additional pass to render the back faces.

In this case, we will render the back faces outside the original sphere to simulate the atmospheric effect.

Adding the atmosphere shader pass

We will add the new pass to the shader under the original shader code:

1. Scroll to end of the shader's Cg snippet:

```
ENDCG
```

2. Directly after this, add the following code:

```
Cull Front
CGPROGRAM
#pragma surface surf Standard fullforwardshadows alpha vertex:vert
struct Input {
    float2 uv_MainTex;
};
float _Thickness;
void vert (inout appdata_full v) {
    v.vertex.xyz += v.normal * _Thickness;
}
fixed4 _AtmosColor;
void surf (Input IN, inout SurfaceOutputStandard o) {
    o.Albedo = _AtmosColor.rgb;
    o.Alpha = _AtmosColor.a;
}
ENDCG
```

   Here we define the second pass. We will use a vertex function to extrude the geometry like we did with the helmet.

   This time, the shader does not use specular or smoothness qualities, but we will define a secondary color that is used to define the atmosphere.

   We will also use the alpha channel of this color to set the opacity of the atmosphere.

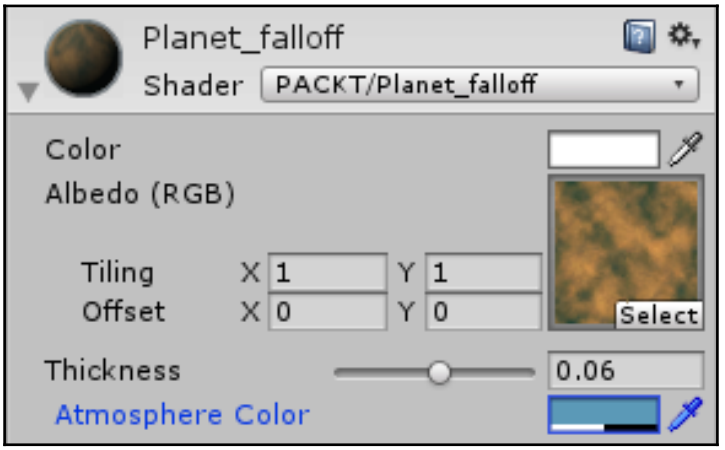
3. Save the script.
4. Minimize MonoDevelop.
5. Back in the main Unity interface, select the `ridleyVI` game object in the **Hierarchy** panel.

Next, we need to set the new material inputs in the **Inspector**.

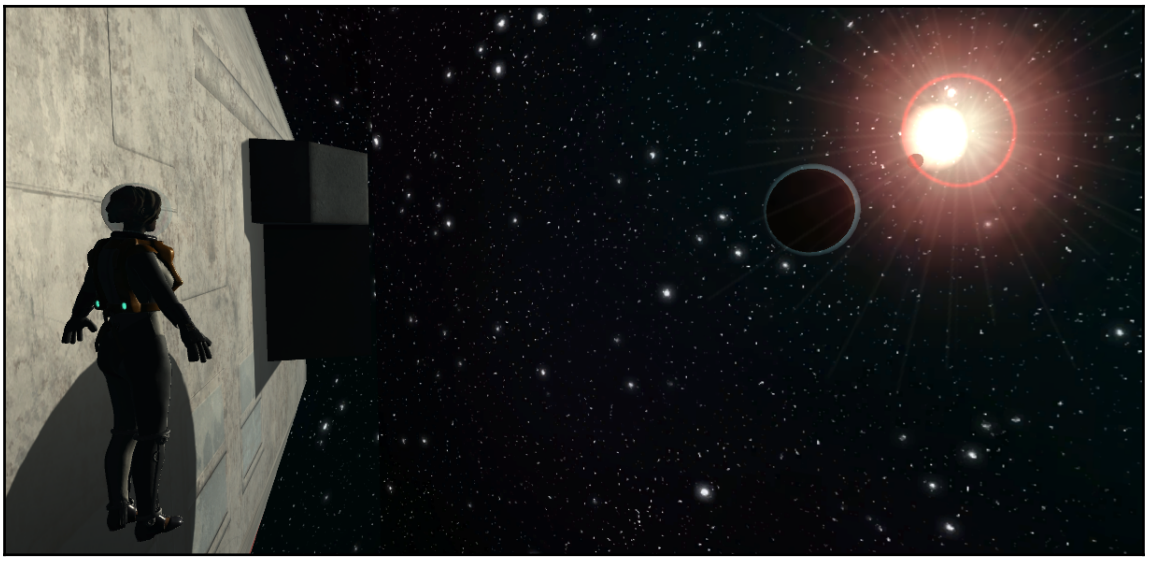
Setting the planet material inputs

The new shader defines a bitmap albedo texture as well as a number of numerical values in the planet material:

1. In the **Project** panel, click on the `PACKT_Textures` folder to view its contents in the **Assets** panel.
2. In the **Assets** panel, locate the `planet001` texture.
3. Drag this onto the **Albedo RGB** map slot in the **Inspector**.
4. Further down in the **Inspector**, set the **Thickness** value to `0.06` to define the extent of the atmosphere.
5. Click on the **Atmosphere Color** swatch and define a slightly bluish color.
6. Set the alpha channel to `128`, or 50% of its range, on the slider.
7. Exit the color selection window by clicking on the **X** in the right-hand top corner:



Viewing the planet in the **Scene** view at this point will allow you to see the atmosphere effect that we added with the shader:



The completed planet shader effect

The planet now has an atmospheric quality that can be further adjusted in the **Inspector**. We have control over the color, thickness, and opacity.

Summary

In this chapter, we began to create custom materials that work within Unity's physically-based shading system.

We started by creating a simple unlit material for the moon, exploring the basic syntax of the shader and demonstrating the interaction between shader, material, and mesh.

We added complexity by incorporating an albedo texture, before implementing a Lambert lighting model to allow the moon to respond to the scene lighting.

In the next example, we built on these concepts, working with passes and Cg snippets to create an improved transparent effect for the astronaut's helmet.

Finally, we created a more complex shader for the planet's atmosphere.

In `Chapter 3`, *Working with Lighting and Light-Emitting Surfaces*, we will demonstrate the impact of scene lights and emissive materials in a new scene and explore how these properties can be utilized in a new set of custom shaders.

3

Working with Lighting and Light-Emitting Surfaces

In this chapter, we will explore the interaction between Unity's lighting tools and various materials in our scene, that will allow us to create a visually interesting game environment.

The important topics covered in the chapter are as follows:

- Setting up lights
- Real-time versus baked-lighting solutions
- Emissive materials
- Real-time glow effects
- Effects of light on different surfaces such as metal, plastic, fabric, skin, eyes, and hair
- Creating a custom wireframe shader

At this point in our game, our astronaut is preparing to enter the atmosphere of the planet in her search for the missing research crew. The interior of the spacecraft is lit with glowing lights and displays.

This scene already has standard materials applied. We will demonstrate how some of the surfaces interact with light before we examine the specific qualities in our own shader.

We will start by looking at the existing scene lights.

Looking at the scene light setup

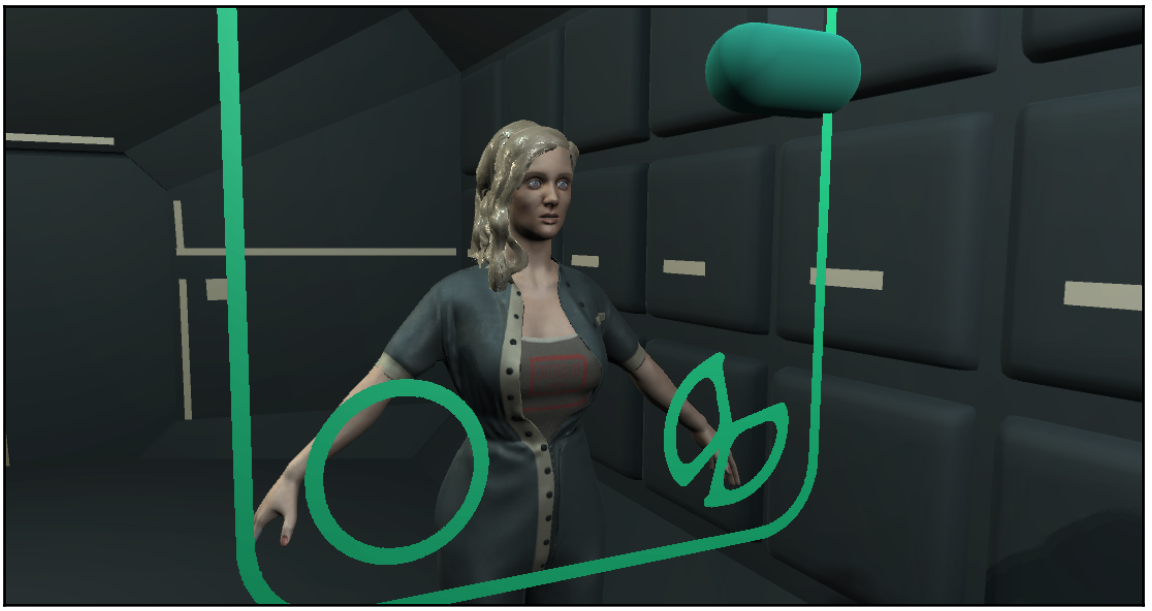
In Unity, we can light scene objects in real time, use lights to bake light maps, and even cast shadows in real time.

In this section, we will demonstrate Unity's lighting setup and how scene lighting affects the scene materials.

You will find the scene set up for this chapter in the Unity project that you have downloaded:

1. Open the scene using the menu bar in the main Unity interface by clicking on **File** and then **Open Scene**.
2. In the dialog that appears, choose the `Chapter3_Start` scene from the `PACKT_Scenes` folder.

The spacecraft bridge environment will become visible in the **Scene** view:



Initial state of the example scene

At this stage in our game, our astronaut is back inside the spacecraft after completing the repairs. She is preparing to travel to the planet's surface in order to make contact with the research team.

The scene represents the spacecraft's bridge, the control center where the astronaut can look at data and plot the ship's course.

Currently, the scene has just a few basic materials set up and a basic lighting setup.

Unity uses **Enlighten** for its lighting solutions. The lighting setup can be baked or real time, but each has their own limitations.

For a more optimized result, we should use **baked lighting**. This solution supports all light sources and emissive surfaces, but it does not accommodate dynamic elements such as a character moving through a scene or lights switching on and off.

Realtime lighting is calculated for every frame and is better for dynamic situations, but currently, it only has limited support for lights. At the time of writing, only **directional lights** can cast real-time shadows. We will be using this as we have animation and dynamic effects in our scene.

We need to make sure that we are using the correct solution:

1. If it is not already present in the Unity interface, open the **Lighting** panel by clicking on **Windows** and then on **Lighting**.

You may find it convenient to dock the **Lighting** window next to **Inspector**, so it does not disappear behind the main Unity window.

2. Scroll to the bottom of the **Lighting** panel and make sure that the **Auto** checkbox is checked.

   This will ensure that lightmaps are automatically baked when a change is made to the scene that affects the lighting.

3. Leave the other settings at their default values.

The lightmaps will now be calculated and you will briefly see a blue progress bar in the bottom-right corner of the interface.

There should be a noticeable difference in the scene:

The example scene with the lighting solution

When the lightmap is baked, there is deeper shadow in the space behind the astronaut character.

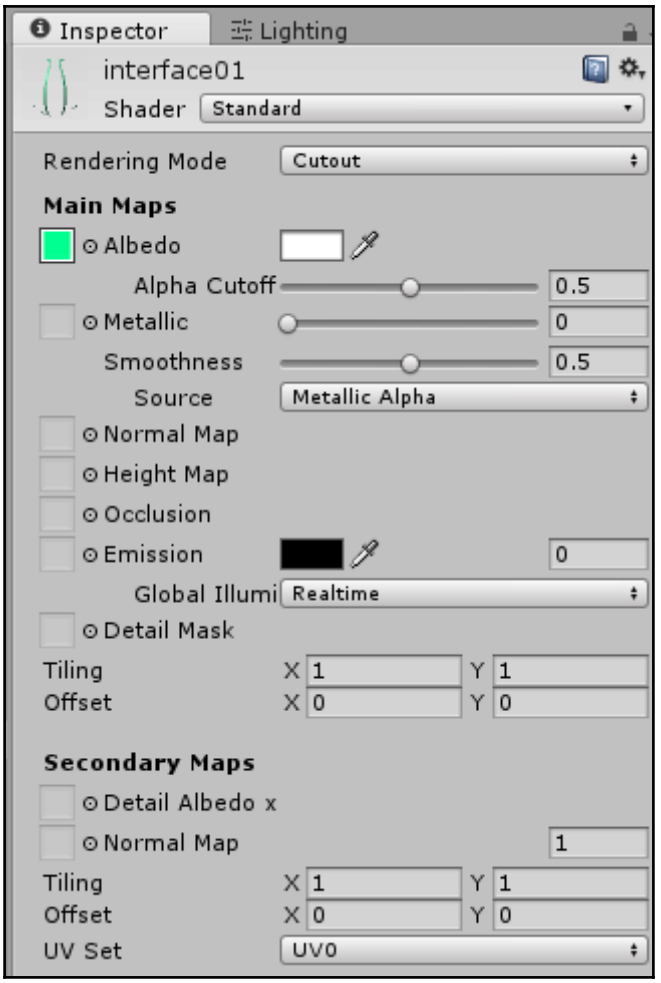
In the next step, we will add emissive properties to the instrument console to light it up.

Adding emissive properties to a material

Any **Standard Shader** material can emit light into a scene by adjusting some values in the Inspector panel.

We will start by defining the strength of the illumination:

1. In the **Project** panel, click on the `PACKT_Materials` folder to view its contents in the **Assets** panel.
2. In the **Assets** panel, locate the `interface01` material and click it once to display its properties in the **Inspector** panel:



The interface01 material parameters

The `interface01` material is a Standard Shader material set to the **Cutout** mode. It uses a bitmap **Albedo** texture with an alpha channel to define the transparency.

The emission value in the Standard Shader can be defined by a color value or a texture map in the **Inspector** panel.

3. In the **Inspector** panel, click on the rectangular color swatch next to **Emission**.
4. Choose a gray-blue color. The hex code `3A6359` should give you a suitable value.

In the **Scene** view, the control panel will become illuminated.

Let's add a similar illuminated material to another part of the geometry to lighten up the scene even further:

1. In the **Hierarchy** panel, locate the `lightCapsule` game object.
2. Drag the `glowLight` material from the **Assets** panel onto `lightCapsule` in the **Hierarchy** panel or **Scene** view to add it.

    Next, we can tone down the existing lighting a little to better see the emissive effect.

3. In the **Hierarchy** panel, select `front light`.

    This is a **point light** with adjusted **Range** and **Intensity** values to fit the scene.

4. In the **Inspector**, decrease the light's intensity to `2.0`.

    In the **Scene** view, the area in front of the astronaut should become a little darker as a result:

The example scene with adjusted light values

The **Emissive** texture map uses brightness to define the areas that are emissive. Full white equals maximum emission and black equals zero emission.

The control panel is lit up, but it does not currently cast light onto other objects in the scene, such as the astronaut, like a real light would do. To do this, we need to add an effect to the camera.

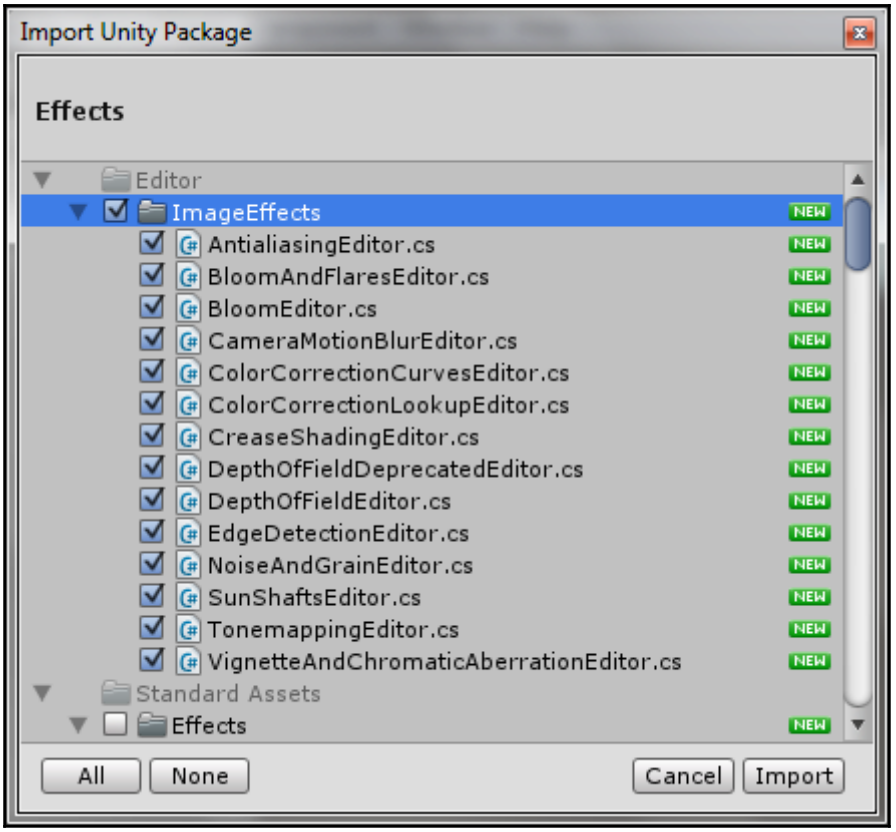
Adding the Bloom effect

Unity's native glow effect is called bloom, it will make all light sources (including emissive surfaces) in a scene glow, based on their light values.

Unity's image effects are included in a package that comes with the install. We can install packages (or selected contents) when we create a project or any time after:

1. In the menu bar at the top of the main Unity interface, navigate to **Assets** | **Import Package** | **Effects**.

2. In the **Import Unity Package** dialog that appears, check the **ImageEffects** folder:



The Import Unity Package window

3. Click on the **Import** button to add the effects scripts to the project.
4. In the **Hierarchy** panel, select the `Main Camera` game object.
5. In the **Inspector**, scroll to the bottom of the components and click on the **Add Component** button.

6. Navigate to **Image Effects** | **Bloom and Glow** | **Bloom**.

The default **Bloom** effect will be added to the `Main Camera`:

The Bloom effect added to our scene

The default values appear a little understated within our scene. We need to tweak some values in the effect to get it to look better.

7. In the **Bloom (Script)** component in the **Inspector**, set the **Intensity** value to `2.2`.
8. Set the **Threshold** value to `0.23`.
9. Set the **Blur Iterations** value to `1`.
10. Leave the **Sample Distance** value at its default value of `2.5`.
11. Leave the remaining parameters at their default values.

When the lighting solution updates, we should see some changes in the scene:

The Bloom effect with adjusted values

We can get a better result by adding a **Reflection Probe** to the scene. This will create a map of the light that bounces off reflective surfaces.

We can further improve the effect by adding a Reflection Probe to simulate reflected light in the scene. By default, Unity uses the skybox for reflections, and this is not always appropriate for interior scenes such as this one.

Adding a reflection probe

Reflection probes sample the reflected material values in an area of the scene.

Add the ReflectionProbe from the menu bar by clicking on **GameObject**, selecting **Light**, and then clicking on **Reflection Probe**.

As we have already made sure that the light solution is set to **Auto**, the reflection probe will bake automatically.

Give it a few minutes to complete. You will see a progress bar in the bottom-right corner of the main Unity interface.

When the lighting solution has updated, you will see some changes in the scene:

Effect of Reflection Probe on the Bloom effect in the example scene

The glow effect will appear softer on lit objects that are less shiny, such as the astronaut's hair and clothing, and more pronounced on glossy surfaces, such as the skin and some parts of the bridge interior.

Creating a wireframe emissive material for the planet-surface scanner

eow that we have demonstrated the use of emissive materials to create effects and light up our scene, we will go one step further and create a wireframe material to represent scanning the planet's surface. Rather than illuminating a part of the object's surface, we will be lighting up the object's geometric edges using this very specialized shader.

We will start by creating a new shader:

1. In the **Project** panel, click on the `PACKT_Shaders` folder.

   When the folder's contents appear in the **Assets** panel, create a new shader.

2. Right-click in an empty area of the panel, click on **Create**, and choose **Shader**.
3. Rename the new shader `Unlit_wire`.
4. In the **Assets** panel, double-click on `Unlit_wire` to open it in **MonoDevelop**.
5. When the shader opens in MonoDevelop, change the shader's directory by replacing the first line of code with the following:

```
Shader "PACKT/unlit_wireframe" {
```

6. Select the `Properties` block and replace it with the following:

```
Properties
{
    _Color ("Color", Color) = (0,0,0,0)
    _EdgeColor ("Edge Color", Color) = (0,1,0,1)
    _Width ("Width", float) = 0.1
}
```

   These are the three properties that our shader needs.

   The `_Color` property defines an RGBA value that the model will use. The `_EdgeColor` property will define the unlit color of the wire. Lastly, `_Width` will define the thickness of the outline.

7. In the `subShader` block, replace the `Tags` line with the following code:

```
Tags { "RenderType"="Transparent" "Queue"="Transparent" }
```

   This line of code will allow the shader to be transparent.

   The shader requires two passes to render the front and back faces of the geometry separately. We will create the first pass by defining it with tags and defining how it is blended.

8. Add the following code:

```
Pass
{
    Blend SrcAlpha OneMinusSrcAlpha
    Cull Front
    AlphaTest Greater 0.5
}
```

   The Blend mode defines how the pass is added to the rest of the shader. There are two factors included here—the **source texture** and the **destination texture**. When we are just blending alphas, we will define this by writing Alpha as the source and/or destination.

   The next line allows us to not render the front faces of the model. The front faces will be rendered in the second pass. We keep this order to render far away things first and avoid z-fighting issues.

   The last line allows us to use cutout transparency, which will define pixels as completely transparent or completely opaque. Using real, per pixel transparency here would be costly to performance, and our Bloom effect would negate the visual clarity anyway.

   The meat of the code uses a **Cg** snippet.

9. Add the following code within the brackets of the pass beneath the `Blend` line:

```
CGPROGRAM
#pragma vertex vert
#pragma fragment frag

uniform float4 _Color;
uniform float4 _EdgeColor;
uniform float _Width;
```

   Here we will select the Cg shader elements to use and define our properties as variables we can use in the shader.

10. Next, add the following code:

```
struct appdata
{
    float4 vertex : POSITION;
    float4 texcoord1 : TEXCOORD0;
    float4 color : COLOR;
};
```

   Here, we are defining local variables that we use in our shader, such as vertex positions, texture coordinates, and a color.

11. Then, add the following code:

```
struct v2f
{
    float4 pos : POSITION;
    float4 texcoord1 : TEXCOORD0;
    float4 color : COLOR;
};
```

   Here, we will define the variables used for the vertex to fragment calculation that allows us to make the geometry visible in the scene.

   Next, we need to define how this data is used in the shader.

12. Add the following code:

```
v2f vert (appdata v)
{
    v2f o;
    o.pos = mul( UNITY_MATRIX_MVP, v.vertex);
    o.texcoord1 = v.texcoord1;
    o.color = v.color;
```

```
    return o;
}
```

This block of code uses `appdata v` variable that we previously defined.

We will define the vertex to fragment the `o` variable (output), and then define the output color as the vertex color we have already defined using the UV coordinates.

Lastly, we will define the fragment part of the shader:

13. Add the following code:

```
fixed4 frag(v2f i) : COLOR
{
    fixed4 answer;

    float lx = step(_Width, i.texcoord1.x);
    float ly = step(_Width, i.texcoord1.y);
    float hx = step(i.texcoord1.x, 1.0 - _Width);
    float hy = step(i.texcoord1.y, 1.0 - _Width);

    answer = lerp(_EdgeColor, _Color, lx*ly*hx*hy);

    return answer;
}
ENDCG
```

Here, we are defining the final appearance of the geometry based on the properties that we defined at the top of the shader, and our vertex data that we got from the geometry.

Finally, we end the Cg snippet using `ENDCG`.

14. Delete the remaining default code within the `subShader`.
15. Save the shader and return to the main Unity interface.

In the next step, we will use the shader in our scene.

Viewing the wireframe emissive shader in the scene

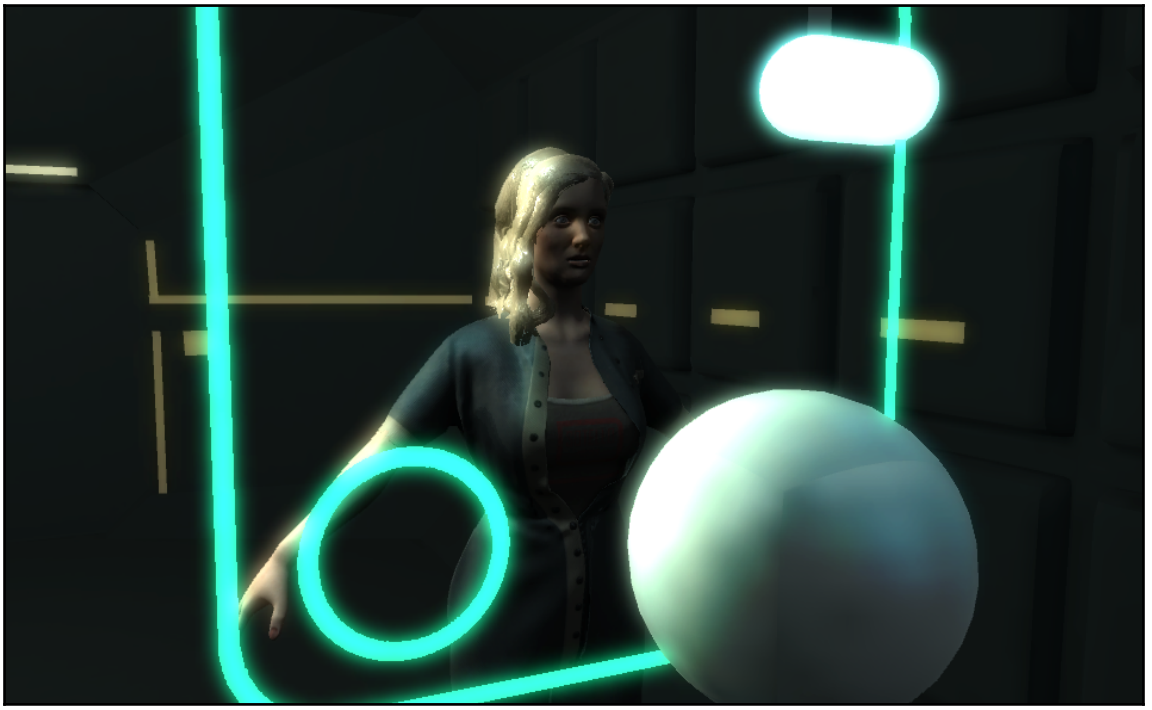
We already have the geometry and a material in the scene that will use this shader.

The object is a sphere positioned in front of the ship's control console. The `planetscan` object represents the ship's scan of the planet's surface as it begins its descent to the surface.

1. In the **Hierarchy** view, click on the `planetscan` object.

   Its properties will appear in the **Inspector**.

2. Uncheck the checkbox in the top left-hand corner of the **Inspector** to make the game object visible:



The planetscan game object visible in the scene

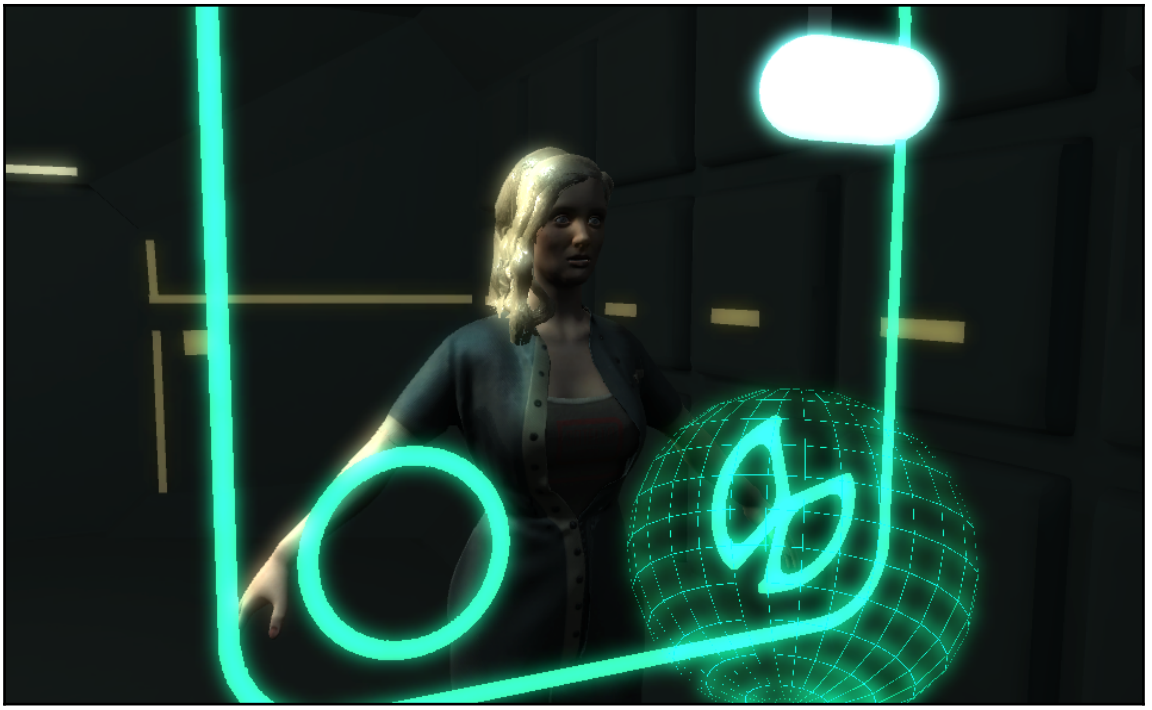
At the bottom of the **Inspector**, the object's current material is displayed. It is currently set to use a default Standard Shader material.

3. Click on the **Shader** dropdown and select `PACKT - wireframeEmissive`.

   The properties in the **Inspector** will update to show those contained in the shader that we created.

4. Click on the color swatch next to the **Color** property and set the alpha channel value to zero.
5. Click the **Edge Color** swatch and enter the code `12F7BFFF` into the hex code field at the bottom of the color picker.
6. Set the **Width** to a value of `0.01`.

As our material is already being used in the scene, we should be able to see these changes on the `planetscan` object instantly:



The planetscan object with the wireframe shader

The wireframe shader responds to the Bloom effect really well as it is unlit; it is not otherwise affected by the scene lighting.

Note that just the back faces of the sphere are being rendered here.

At this point, we can return to MonoDevelop and add the second pass to render the front face of the sphere.

Adding the wireframe shader's second pass

Using the additional pass will allow us to render both sides of the sphere wireframe in a clean way:

1. Maximize MonoDevelop.
2. Select the entire `Pass` in the shader.

If you select the opening bracket directly beneath the `Pass` tag, MonoDevelop will highlight the closing bracket allowing you to visualize the end of the block of code.

3. Copy the selected code by right-clicking and selecting **Copy** from the drop-down list or using the *Ctrl* + *C* hotkey combination (or *Command* + *C* if you are working on a Mac).
4. Paste the copied code directly after the closing bracket. Use *Ctrl* + *V* (*Command* + *V* on a Mac), or the **Paste** option from the right-click selection dropdown.

   The code for the second pass is almost the same, we just need to cull the back faces rather than the front.

5. Near the top of the pasted `Pass`, locate the following line of code:

```
Cull Front
```

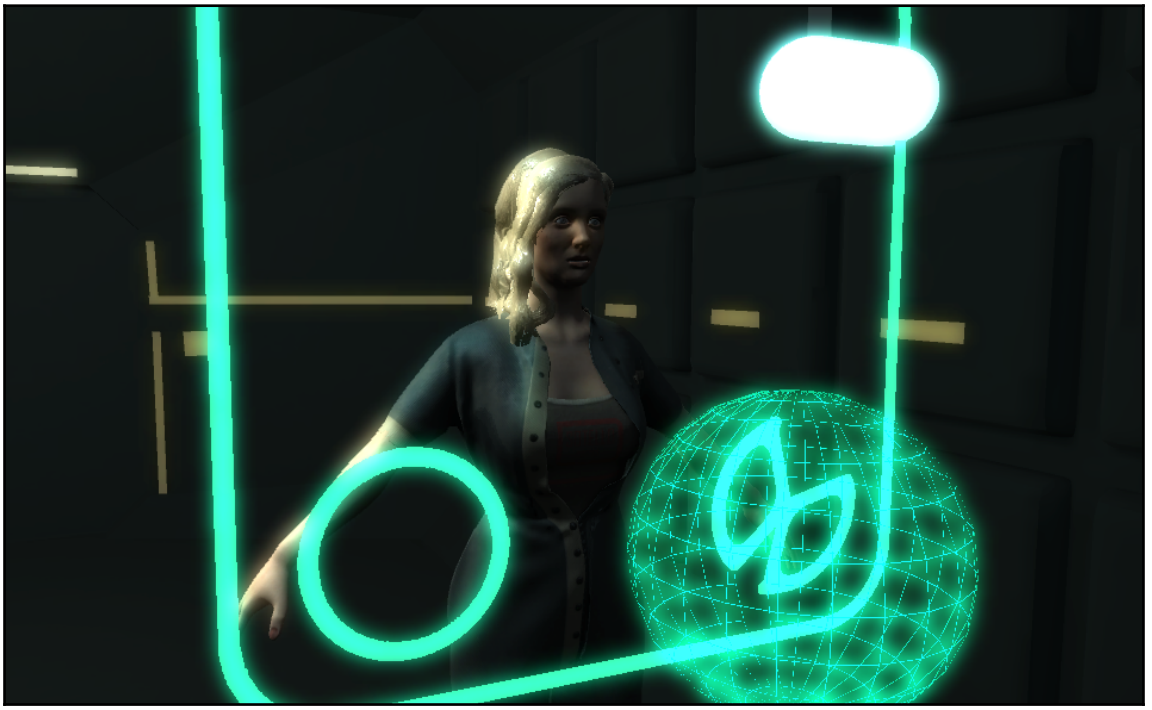
6. Replace it with the following line:

```
Cull Back
```

7. Save the shader.

Return to the main Unity interface.

The shader should update automatically, allowing you to view the complete `planetScan` sphere in the **Game** view:



The planetscan object now showing the additional shader pass

In the next step, we will add a projection cone to the sphere to complete this effect.

Completing the planet scan projection effect

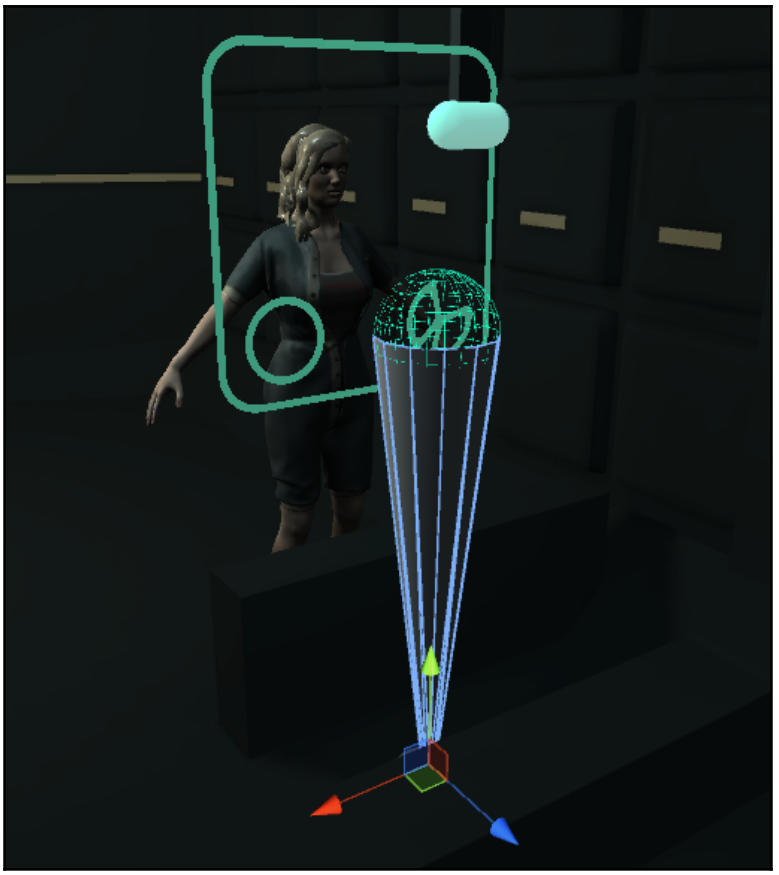
We can get the planet scan looking a little more like a projection by adding some more polish to the effect:

1. In the **Project** panel, click on the `PACKT_Models` folder.
2. The contents of the folder will appear in the **Assets** panel.

3. In the **Assets** panel, locate the `projectionCone` asset.

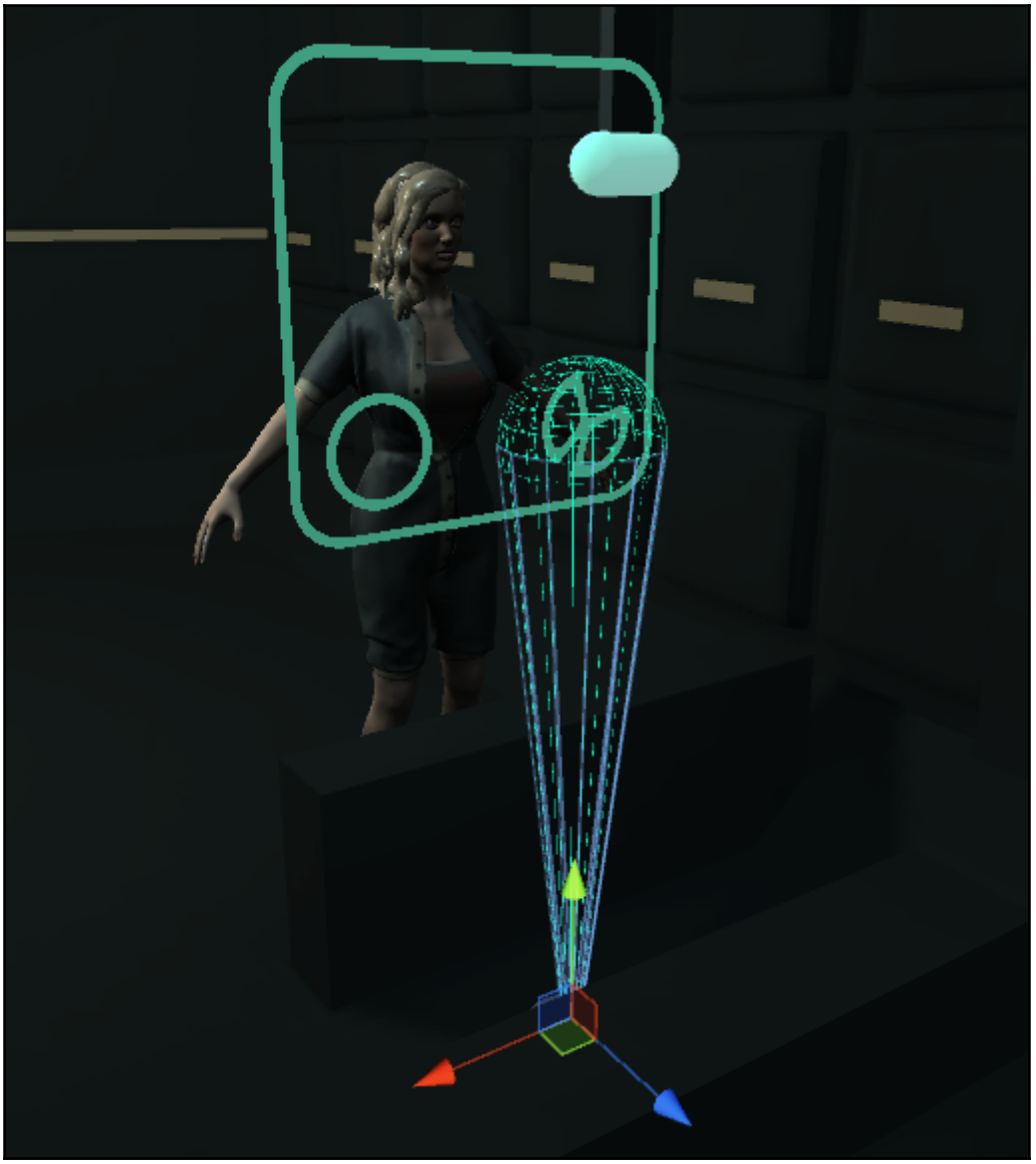
   This is a simple cone model created in an external three-dimensional application. A lot of the time, Unity's primitives work great for simple effects, but at the time of writing, Unity does not have a cone primitive.

4. Drag `projectionCone` from the **Assets** panel into the **Hierarchy** panel to add it to the scene.
5. In the **Scene** view, transform tools and the **Inspector** to rotate the cone so that it is completely inverted and positioned directly beneath the `planetScan` sphere:



The unhidden projectionCone object in its initial state

6. When it is positioned correctly, drag the `PlanetScan` material from the **Assets** panel onto the `projectionCone` game object:



The projectionCone object using the wireframe shader

We have completed the setup of the emissive surfaces in our scene.

At this point, we should have a more interesting scene. We will continue adding to this in the next chapter.

Summary

In this chapter, we introduced some basic features of scene lighting in Unity. We then explored the options for creating emissive materials with Unity's Standard Shader.

We also demonstrated how the Standard Shader can be used to emit light in the scene and what impact this has on a variety of surfaces.

Within the context of our spacecraft bridge scene, we created a custom wireframe shader to represent a projected scan visualization of the planet's surface. We completed the chapter by adding further effects to complete the look of the spacecraft bridge.

In the next chapter, we will demonstrate how these shader effects can be animated and controlled during gameplay with the use of a little external code and a closer look at the shaders.

4

Animating Surfaces with Code and Shaders

In this chapter, we will set up the animated effects for the spacecraft bridge display as the astronaut begins her descent to the planet in order to search for the missing research crew.

We will use the scene and materials that we previously set up in `Chapter 3`, *Working with Lighting and Light-Emitting Surfaces* to demonstrate how to bring an environment to life with Unity's animation tools and a little code.

We will explore the following topics in this chapter:

- Creating and animating dynamic scene lights
- Accessing shader properties with C#
- Interpolating between albedo/transparency textures at runtime
- Scrolling UV coordinates
- Creating looping animation sequences with Unity's Animation window

By the end of the chapter, we will have created a vibrant and exciting scene.

Let's get started!

Starting the scene

In the previous chapter, we set up some materials and lighting for the spacecraft bridge scene. We will be building on this scene next:

1. In the **Project** panel, double-click on the `PACKT_Scenes` folder to view its contents in the **Assets** panel.
2. In the **Assets** panel, double-click on the `Chapter4_Start` asset to load the scene.

   The scene will load. You should be able to see the spacecraft bridge interior in the **Scene** view:

The environment is lit as we left it at the end of `Chapter 3`, *Working with Lighting and Light-Emitting Surfaces*. In the next step, we will add and animate an additional light.

Creating a dynamic warning light effect

The scene is currently lit by the directional light and emissive materials in the display and `planetscan` objects. Next, we will add a point light to the scene:

1. Create a new light using the **GameObject** dropdown in the menu bar by navigating to **GameObject** | **Light** | **Point Light**.

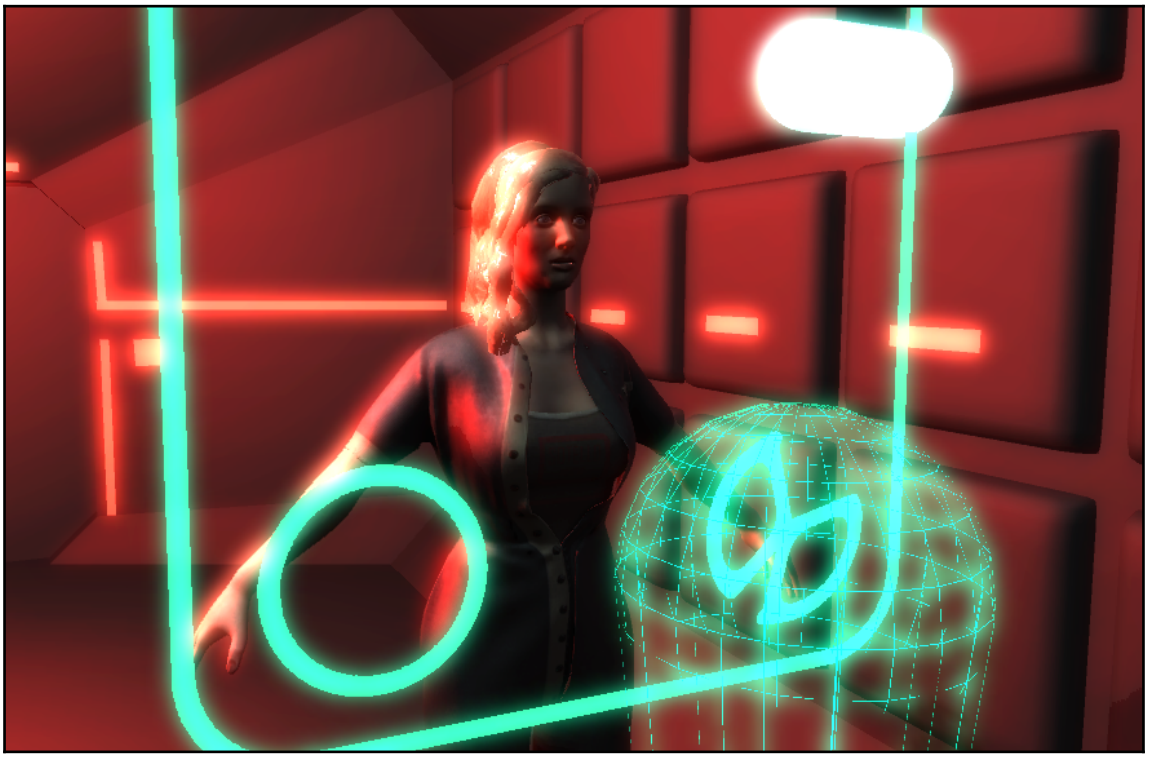
    The new light's components will become visible in the **Inspector**.

2. Rename the light `warning_light`.
3. Change the light's position by adjusting the values in the **Transform** parameters near the top of the **Inspector**. Set the **Position X** value to `0.5`, the **Position Y** value to `2.5`, and the **Position Z** value to 0.

    This will move the light to the top of the center of the spacecraft bridge scene, so it will illuminate the astronaut and her immediate environment.

4. In the **Light** component parameters, set the **Intensity** to `2.0`.
5. Set the **Range** to `15`.
6. Click the **Color** swatch and choose a bright red color using the color picker. The hex code `FF0000FF` will give you a full red value.
7. We will leave the **Baking** set to `Realtime` and the **Shadow Type** set to `No Shadows`.

At the time of writing, Unity does not support real-time shadow casting for point lights, and we will be updating the light at runtime, so we cannot bake the light with the others:



As you add the light and update its parameters, you should see the Enlighten progress bar in the lower-right corner of the Unity interface as the light maps are rebaked.

Next, we will animate the light.

Animating the light

In the spacecraft bridge scene, the warning light flashes to indicate danger during the spacecraft's entry of the planet's atmosphere. Unity's native animation workflow is ideal for simple effects like these:

1. Using the menu bar, open the **Animation** panel by clicking on **Window** and then selecting **Animation**.

   The **Animation** panel is used to create and edit animation clips in Unity.

   At the center of the panel, there is a message prompting us to create an animation clip and controller for the selected object.

2. Click on the **Create** button.

   A dialog will appear, allowing us to specify a location and name for the new animation.

3. Select the `PACKT_Animation` folder as the location.
4. Type `warning` as the animation name and click **Save**.

   Next, we need to select the properties of the light that we want to animate.

5. Click on the **Add Property** button.
6. In the properties list that appears, click on the small arrow next to **Light**.
7. Click on the **+** sign next to **Intensity** to add this property to the animation.

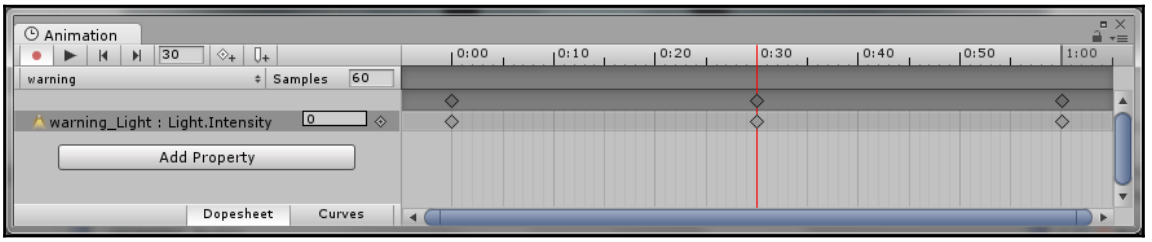
   Start and end keyframes are added to the animation timeline. Both are set to the current intensity value. We need to set an intermediate key to see some variation in the **Scene** view.

8. Drag the red line to the middle of the animation track to move the playhead.

9. Click on the diamond-shaped button near the left-hand top of the panel to add a keyframe at this point in the animation.

10. With the new keyframe still selected, locate the **Intensity** value next to the property name.

11. Set the value to 0:



Now that we have created the animation, we can preview it in the **Scene** by pressing the play button within the **Animation** panel.

12. Click on the play button.

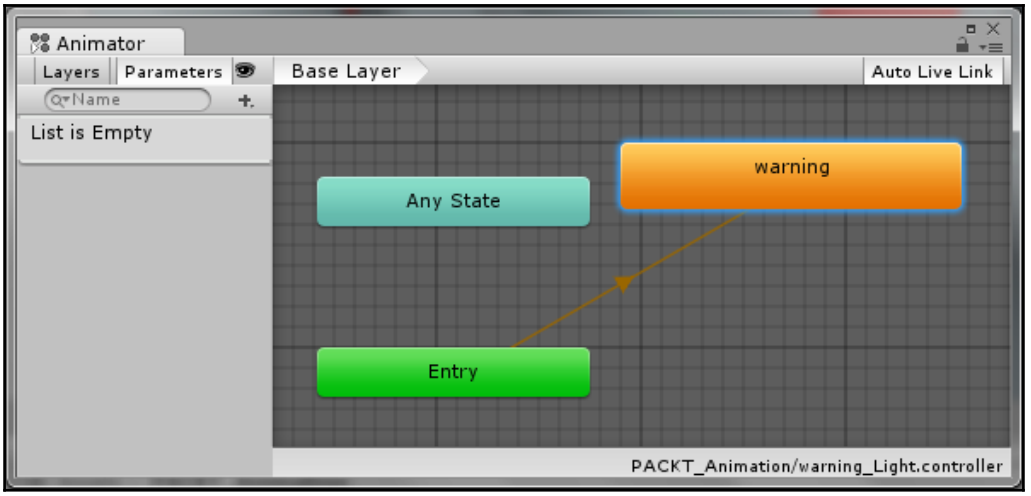
In the **Scene** view, the light should flash on and off.

13. Click the play button again to exit the play mode.
14. Close the **Animation** panel.

When we created the animation, a simple state machine was set up and added to the game object as an **Animator Controller**.

15. Open the **Animator** panel using the menu bar by clicking on **Window** and then selecting **Animator**.
16. In the **Hierarchy** panel, click on the `warning_light` game object again if it has been deselected.

17. The object's state machine will become visible in the **Animator** panel:



The warning state, containing the animation that we created, becomes active when the scene starts.

We can add more states to the controller and more complexity to the animation later in the chapter.

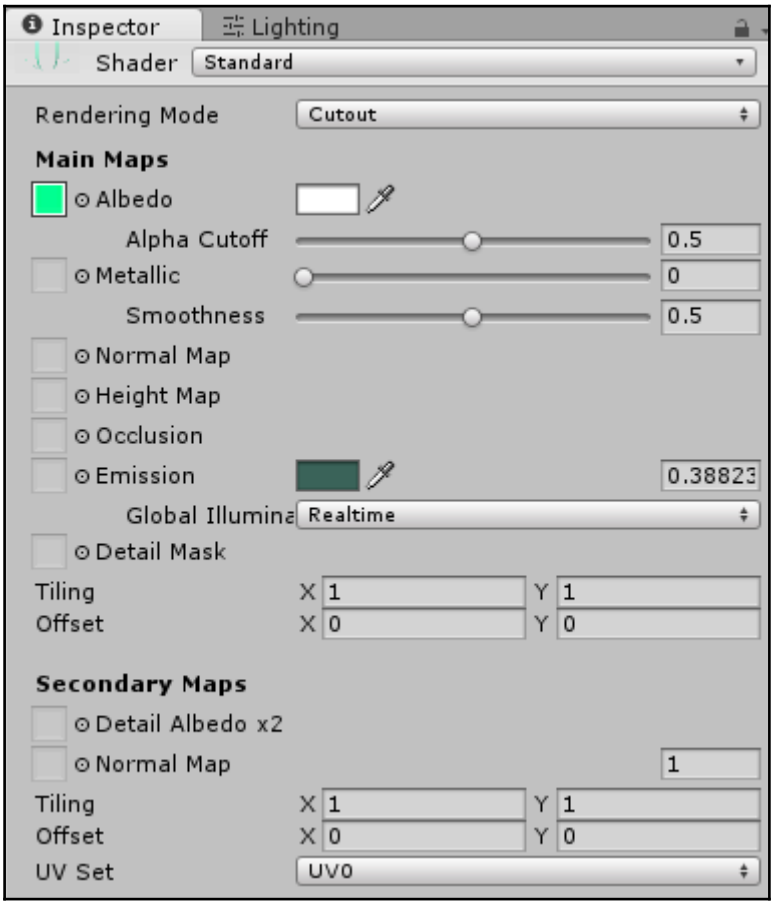
In the next section, we will use an external **C#** script to change shader properties within a material at runtime.

Animating the control panel illumination

The control panel shader was set up with an emissive material to simulate the glowing controls. In this step, we will animate the effect to simulate the conditional changes as the ship enters the atmosphere.

1. In the **Project** panel, click on the `PACKT_Materials` folder to view its contents in the **Assets** panel.

2. Click on the `interface01` material to view its properties in the **Inspector**:



The material uses the default Standard Shader set up with emissive properties.

We can change the texture used by the material using a little code, we just need to know the name of the property that uses the texture in the shader. The property names enclosed in quotation marks are exposed in the **Inspector** when the shader is used in a material. This is distinct from the property name used in the shader that we will need to know to change the texture that it uses.

When you are working with a shader that you have not written yourself, you can find the property name by selecting the shader.

3. In the **Hierarchy** panel, select the `interface_quad` game object.

   Its components will become visible in the **Inspector**.

4. Then scroll down to the object's **Material** component.
5. Click on the gear icon on the right-hand top corner of the component block.
6. Choose **Edit Shader** from the dropdown list.

   The shader source will be selected and the property names will become visible in the **Inspector**:

We can see that the shader property that handles the emissive map is named `_MainTex`, which is the standard name used for the albedo texture.

This is the property that we need to access in the script to animate the effect.

7. In the **Project** panel, click on the `PACKT_Scripts` folder.
8. When its contents appear in the **Assets** panel, create a new C# script by right-clicking and choosing **Create**. Then, select **C# Script**.
9. Rename the script `controlTwinkle`.
10. Double-click on the `controlTwinkle` asset to open it in MonoDevelop.
11. Directly after the script's opening brackets, add the following code:

```
public bool switched;
public float frequency = 0.2f;
public Texture2D origAlbedo;
public Texture2D newAlbedo;
```

Here, we are defining the variables that will be used in the control script. The `switched` variable is a **Boolean**: a simple on/off switch that we can use to switch between the two textures used in the material.
The `frequency` value is a **float**, or decimal number that will define how long the state lasts for. These first two variables do not strictly need to be public, but it is sometimes useful to see the Boolean go on and off and to tweak values at runtime in order to fine tune the script rather than hard coding everything.
The next two variables store the references to the textures that we will switch between during the runtime. These need to be public variables so that we can drag the textures into the scripts' slots to define them.

12. Next, we will replace the default definition of the `Start` function with the following code:

```
void Start ()
{
    InvokeRepeating ("Switch", frequency, frequency);
}
```

Here, we will use the `InvokeRepeating` method, which runs a separate function periodically based on a duration value and an interval value. We are setting these to be the same using our frequency variable that we just defined.

13. At the end of the script, but before the closing bracket, add the following code:

```
void Switch ()
{
    switched = !switched;
}
```

Here, we will define the `Switch` function that is run by the `InvokeRepeating` line in `Start`.

The content of the function is extremely simple. It reverses the state of the `switched` Boolean. If `switched` is true, it is set to false and vice versa.

Now that we got the switch working, we need to define what happens when the Boolean is true and when it is false.

14. Replace the default `Update` function with the following code:

```
void Update ()
{
    if(switched)
    {
        GetComponent<Renderer>().material.SetTexture(
          "_MainTex", newAlbedo);
    }
    else
    {
        GetComponent<Renderer>().material.SetTexture(
          "_MainTex", origAlbedo);
    }
}
```

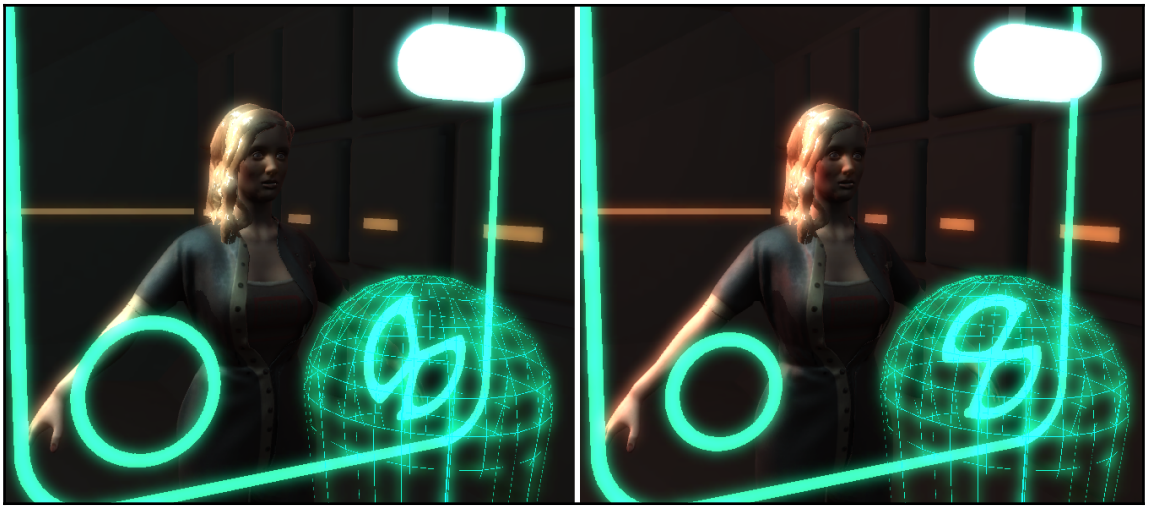
Here, we will define the consequences of the two states of the `switched` Boolean.

In the first case, we will access the object's **Renderer** using the `GetComponent` method. Within this, we will set the texture of `_EmissionMap` to `newEmit`, which are the first of the two textures defined in the script.

When `switched` is false, we set the texture to `origAlbedo` to use the second texture.

15. Save the script.
16. Minimize MonoDevelop and return to the main Unity interface.

17. In the **Hierarchy** panel, reselect the `interface_quad` object.
18. Drag the `controlTwinkle` script into the **Inspector** to add it to the object as a component.
19. In the **Project** panel, select the `PACKT_Textures` folder.
20. Locate `interface01` and drag it into the `Orig Albedo` texture slot in the **Inspector**.
21. Locate `interface02` and drag it into the `New Albedo` texture slot.
22. At the center of the top of the Unity interface, press the play button to see the effect:



During the play mode, the emissive texture in the `controlPanel` material quickly alternates between the two defined textures.

Next, we will animate the text readout display by accessing the shader's UV values.

Animating UV coordinates

In this scene, the illuminated display has been split up so that it can be animated. We want the text readout to scroll down in a constant loop. We can do this by accessing the shader's UV coordinates and animating them:

1. In the **Hierarchy** panel, select the `textDisplay` object.
2. When its properties appear in the **Inspector**, check the checkbox next to its name at the top of the panel to activate it.

   The game object uses a Standard Shader material with emissive values already set up.

   As we are animating the UV values here, we can use Unity's native animation tools rather than using a script.

3. In the **Project** panel, click on the `PACKT_Scripts` folder.

   The folder's contents will appear in the **Assets** panel.

4. In an empty area of the panel, create a new C# script by right-clicking.
5. Click on **Create** and then choose **C# Script** from the drop-down list.
6. Rename the script `textAnim`.
7. Double-click on the new script in the **Assets** panel to open it in MonoDevelop.

   We can keep this code simple; all we need to do is access the shader's vertical UV coordinate and increase it with time.

8. When the script opens in MonoDevelop, delete the default `Start` and `Update` functions.
9. Add the following lines inside the script's curly brackets:

```
Vector2 uvOffset = Vector2.zero;
public Vector2 uvAnimSpeed = new Vector2( 0.0f, 0.1f );
```

   Both the lines of code define new `Vector2` variables. We set the first to `zero`.

   By typing `public` in front of the second variable, we expose it in the Inspector so that it can be adjusted. We set the second value of `uvAnimSpeed` to `0.1` to start.

10. Under the variables, add the following code:

```
void LateUpdate ()
{
    uvOffset += (uvAnimSpeed * Time.deltaTime);
    GetComponent<Renderer>().material.SetTextureOffset(
      "_MainTex", uvOffset );
}
```

    The only function we use in our script is `LateUpdate`, which runs every frame. Within the function, we will increase the values of our `uvOffset` variable using `uvAnimSpeed` multiplied by time in seconds.

    We then use the `GetComponent` method to access the `Renderer` attached to the same object as the script. We set the texture offset of `_MainTex` using the `uvOffset` `Vector2` variable as the value.

11. Save the script.
12. Add the `textAnim` script to the game object by dragging it from the **Assets** panel onto `textDisplay` in the **Hierarchy** panel.
13. Press the play button at the center of the top of the main Unity interface to see the `textdisplay` scroll:

In the next step, we will animate the planet projection display.

Animating the planet projection display

In `Chapter 3`, *Working with Lighting and Light-Emitting Surfaces*, we set up the representation of the planet scan using a custom wireframe shader. In this section, we will animate the effect using the **Animation** panel and by modifying the shader.

We will start by animating the rotation of the planet sphere:

1. In the **Hierarchy** panel, select the `planetScan` game object.
2. Open the **Animation** panel by clicking on **Window** and then selecting **Animation**.
3. When the panel appears, click on the **Create** button to set up the components and create the animation file.
4. Name the animation `planetTurn` and designate the `PACKT_Animation` folder as its location.
5. In the **Animation** panel, click on the **Add Property** button.
6. Click on **Transform** and choose **Rotation** as the property.
7. Expand the animated property by clicking on the small hierarchy arrow next to the `planetScan` rotation on the left-hand side of the **Animation** panel.

   We only want the planet sphere to rotate on the y axis in a continual loop.

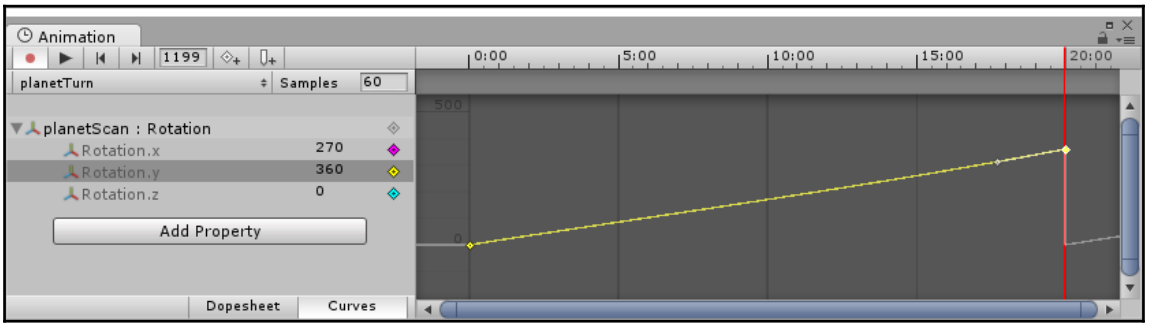
8. Click on the end keyframe in the **Rotation.Y** track.
9. In the value field, type the number `360`.

   This will ensure that the sphere rotates all the way around.

   If you press the play button in the **Animation** panel at this point, you will see the sphere spin in the **Scene** view. With our default timing, it spins a little too fast.

10. In the **Animation** panel, scroll back on the mouse wheel to zoom out.
11. When a larger area of animation timeline is visible, drag the end keyframe to position `20:00` on the timeline.

12. You may have to reset the rotation value to `360`:



13. Dock or close the **Animation** window.
14. Press the play button in the center of the top of the main Unity interface.

The `planetScan` sphere will rotate in a smooth loop.

The rotating sphere represents the planet, but we also need something to represent the spacecraft's destination and trajectory.

In the next step, we will create an animated graphic for the destination on the planet's surface.

Creating an animated hotspot on the planet

Now that the planet's display is animating, we will add a graphic to represent the ship's landing spot. We will do this using a sequence of bitmap images pasted onto a quad.

Creating the hotspot geometry

The geometry that we need for the destination hotspot is very simple, so we will create it within Unity:

1. Create a new quad game object in the scene using the **GameObject** tab in the menu bar by navigating to **GameObject** | **3D Object** | **Quad**.
2. When it appears in the **Hierarchy** panel, click on the quad and rename it `destination`.

3. Parent the new object to the sphere object by dragging it on top of `planetScan` in the **Hierarchy** panel.
4. In the **Inspector**, zero out the `destination` object's **TransformPosition** values to align it to the planet's center.

   Currently, the quad will be faced away from the view. We need to rotate it so that it can be seen.

5. In the **TransformRotationX** field in the **Inspector**, enter the value `-90` to rotate the quad.

   Now that the quad is visible, we need to scale it down so that it is appropriate for the size of the planet.

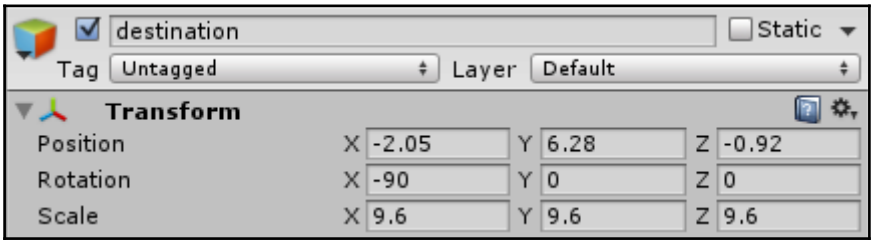
   In the **TransformScale** fields in the **Inspector**, enter the value `9.6` in each of the three fields to scale down the quad.

   Remember that as a child object, the scale visible here is relative to the parent object.

6. Use the move tool in the **Scene** view to reposition the quad so that it is close to the surface of the planet and a little below the equator.

   Try to avoid intersecting the planet object.

   You can use the values in the following illustration if you want to match the example values exactly:



At this point, the `destination` quad should appear behind the `planetScan` sphere in the scene:

Our next objective is to create the hotspot material.

Creating the hotspot material

We will assign a new material to the hotspot and give it similar values to the interface that we set up in the last chapter:

1. In the **Project** panel, click on the `PACKT_Materials` folder to view its contents in the **Assets** panel.
2. In an empty area, create a new material by right-clicking and choosing **Create** and then selecting **Material** from the drop-down list.
3. Rename the new material `hotSpot`.

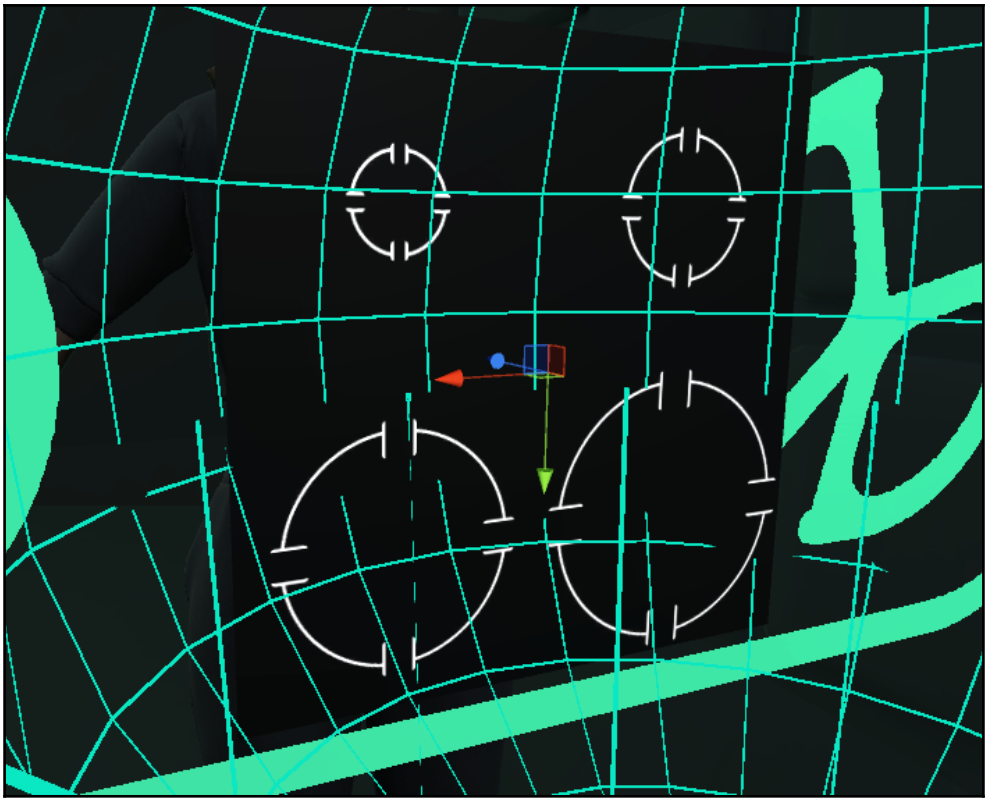
4. Assign it to the `destination` quad by dragging it onto the name of the game object in the **Hierarchy** panel.

    The hotspot texture is located in the texture's folder and needs to be assigned to our new material.

5. In the **Project** panel, click on the `PACKT_Textures` folder to view its contents in the **Assets** panel.
6. Locate the hotspot texture asset and drag it into the `hotSpot` material's **Albedo** slot near the top of the **Inspector**.

    The texture will appear on the quad.

7. Drag the same texture into the **Emission** field further down in the **Inspector**:



We want this graphic to emit light like the other interface elements.

There are four different sizes of the ring graphic and currently all four are displayed on the quad. We need to adjust the UV coordinates so that only one circle is displayed at a time.

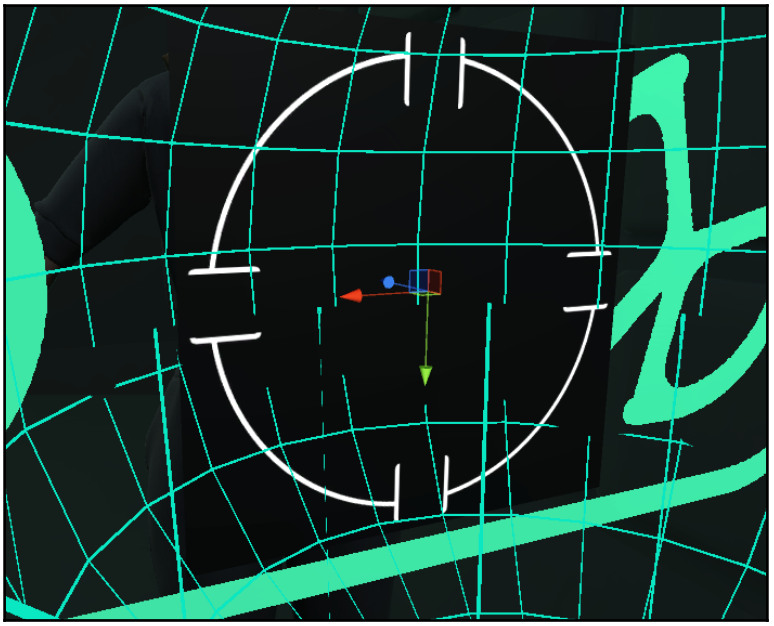
8. Enter the value `0.5` in both the **Tiling** fields, under the **Emission** input settings in the **Inspector**.

   The texture on the quad will change to only display a single circle. This will be the second smallest one, in the lower left of the texture.

   UV coordinates are measured from the lower left-hand corner of the texture space.

9. In the **OffsetY** field, enter the value `0.5`.

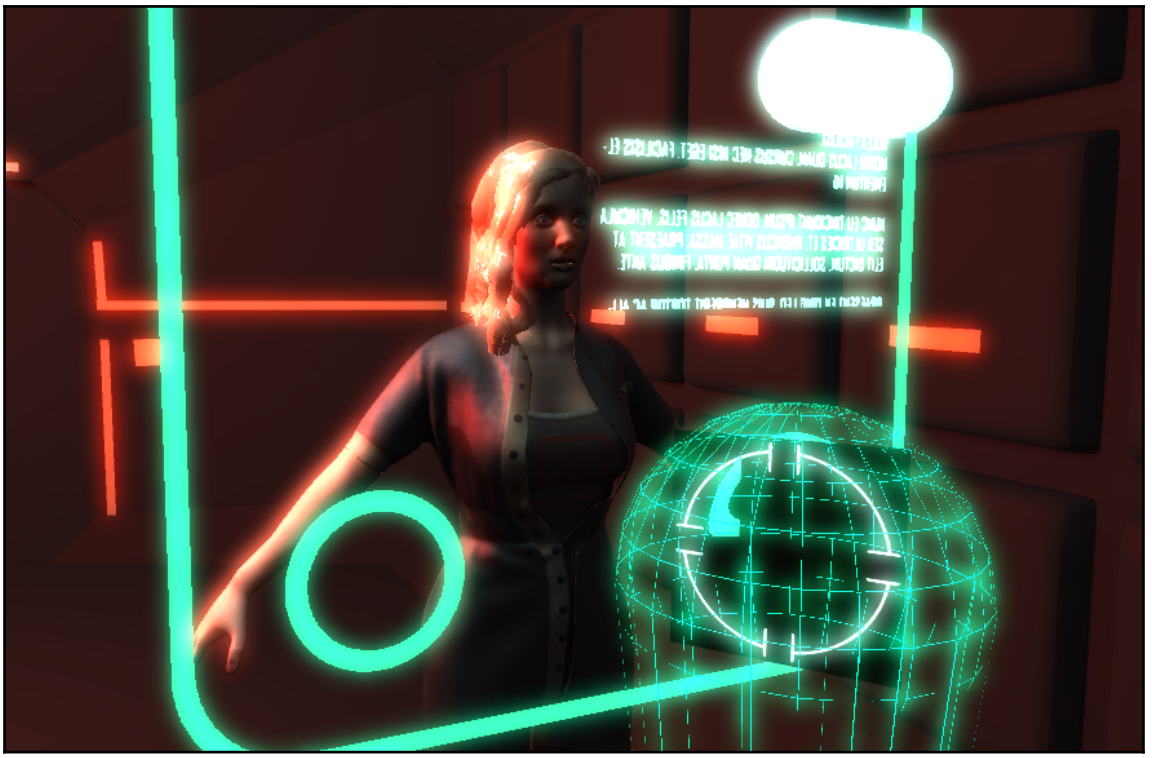
   This will display the largest circle, which is situated in the upper-left quadrant of the texture:



Visualizing the largest circle will make it easier to preview the object in the scene in order to make sure that it fits with the other graphic elements.

10. Switch to the **Game View** by clicking on the **Game** tab in the center of the top of the main Unity interface.

    The `destination` graphic should be clearly visible through the `planetScan` wireframe object:



    At the moment, the destination hotspot is opaque, and the dark areas of the quad become visible as the planet rotates. We need to change the material's **Rendering Mode** to make it invisible.

11. In the **Inspector**, set the `hotSpot` material's **Rendering Mode** to **Cutout**.

Now that the `hotSpot` material is set up, we can animate the effect with a little code.

Writing the hotspot animation script

For the destination hotspot graphic, we could animate using Unity's **Animation** window, but it is easier to tie this in with the other animated effects if we just write a short script to do this:

1. In the **Project** panel, click on the `PACKT_Scripts` folder to view its contents in the **Assets** panel.
2. In the **Assets** panel, create a new C# script asset in an empty part of the folder. Right-click and choose **Create** and then select **C# Script**.
3. When it is created, rename the asset `destHS`.
4. Double-click on `destHS` to open it in MonoDevelop.

   We want the script to access the material and offset and scale the UVs.

5. We will start by creating an array for the UV offsets. Add the following code just after the script's opening bracket:

   ```
   public Vector2 [] UVOffsets;
   ```

6. We will add an integer variable to keep track of the position in the array. Add the following line:

   ```
   public int currArrayPos;
   ```

7. Next, we will add a float variable that will define the duration of each stage of the animation. Add the following code:

   ```
   public float interval = 0.1f;
   ```

   The script will be accessing the destination game object's renderer in order to alter the material at runtime, so we need to include a variable to keep track of this.

8. Add the following code:

   ```
   public Renderer rend;
   ```

   We will set this variable immediately.

9. In the `Start` function, add the following code:

```
rend = GetComponent<Renderer>();
```

   Next, we need to call a custom function that will change the values of our array.

10. Add the following line:

```
InvokeRepeating("NextStage", 1, interval);
```

   The `InvokeRepeating` method calls a named function repetitively at set times. Here, we use our `interval` variable to define the value.

11. Add the new function at the end of the script. Make sure to include it before the script's closing bracket:

```
void NextStage()
{
    currArrayPos++;
    if(currArrayPos > UVOffsets.Length-1)
    {
        currArrayPos = 0;
    }
    rend.material.SetTextureOffset("_MainTex",
      UVOffsets[currArrayPos]);
}
```

   In the `NextStage` function, we will increase the `currArrayPos` integer. In the next line, we will check to see if the value has exceeded the number of positions in the array. When this is the case, we set it back to 0, looping it.

   After this has taken place, we will access the game object's renderer and set the texture offset using the current position in the array.

12. Save the script.
13. Minimize MonoDevelop.
14. Back in the main Unity interface, drag the `destHS` script asset onto the `destination` game object to add it as a component.

In the next step, we will set up the hotspot script variables and preview our animation.

Setting up the hotspot variables and finalizing the effect

Now that we have written the animation script and added it to the game object, we need to add values to the `Vector3` arrays in the **Inspector**:

1. In the **Hierarchy** panel, select the `destination` object; its components should become visible in the **Inspector**.
2. In the **Inspector**, scroll down to locate the **Dest HS (Script)** component block.
3. Click on the small arrow next to the **UV Offsets** variable to view its subcomponents.
4. Set the **Size** property to `4`.

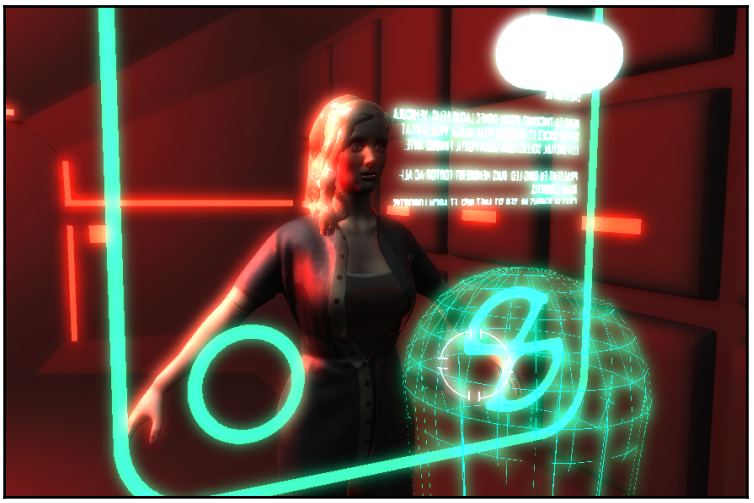
   This will create four sets of fields in the array.

5. Add the following values to the fields:

```
Element 0: 0.5, 0
Element 1: 0, 0
Element 2: 0.5, 0.5
Element 3: 0, 0.5
```

6. Preview this effect by clicking on the play button at the center of the top of the main Unity interface.

   You should see the destination hotspot change size:



We can make this element even more of a focal point in the scene by making it flash off between stages. We can do this by utilizing some of the dead space that is already existing in our texture.

7. Return to the script in MonoDevelop.
8. Add the following code under the other variables that we added:

```
public bool UVTileSwitch;
public float currTile;
```

Here, we add a Boolean variable named `UVTileSwitch`.

The UV tiling will only switch between two states, so it is unnecessary to use an array like we did for the UV offset.

We add the `currTile` float variable for the actual tiling offset. This is set to a default value of 0, where the variable is defined at the top of the script, but it will be changed to define an area of the texture map that is black.

9. Add the following code to the `Update` function:

```
if(UVTileSwitch)
{
    currTile = 0.5f;
}
else
{
    currTile = 0.25f;
}
```

Here, we tie the `currTile` float to the `UVTileSwitch` Boolean. Whenever `UVTileSwitch` returns `true`, we set the `currTile` value to `0.5`; whenever this is not the case, we set it to `0.25`.

10. Save the script.
11. Press the play button again to preview the effect.

The hotspot graphic should now flash off between its different size frames.

We have completed the changes to the Unity scene to make the spacecraft bridge more dynamic and interesting.

Summary

In this chapter, we explored various aspects of animation in our spacecraft bridge scene.

We started by creating a dynamic warning light effect using Unity's native animation tools.

We brought the control display to life by alternating the texture used for the main display albedo and opacity using a simple script.

We then animated the UV coordinates of the text display to make a text readout scroll at runtime using another short script.

We animated the planet scan projection by defining a curve in the **Animation** panel.

Finally, we created an animated graphic for the ship's destination on the planet using another kind of UV animation.

In the next chapter, we will explore the use of transparency to create responsive surfaces and scene effects on the surface of the planet.

5
Exploring Transparent Surfaces and Effects

In this chapter, we will use transparent shaders and atmospheric effects to present the volatile conditions of the planet Ridley VI from the surface.

At this point in our game, the astronaut has landed on the planet to locate the research station and investigate the disappearance of the team. She travels the short distance to the station in a planetary rover, negotiating the planet's turbulent nitrogen storms.

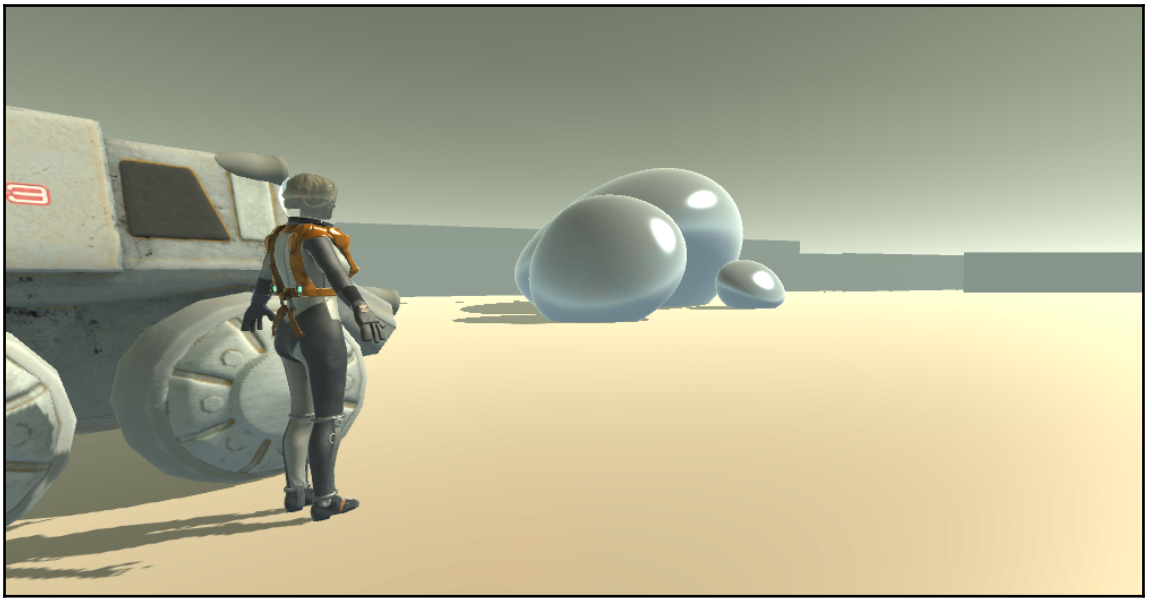
In this chapter, we will discuss the following topics:

- Exploring the difference between cutout, transparent, and fade **Rendering Modes**
- Understanding how to access material properties via code
- Animating the transparency of a material, making transparent images dissolve and coalesce with code
- Understanding the connection between shaders, material instances, and prefabs in the Unity project
- Implementing and adjusting Unity's fog effect in the scene
- Creating more realistic transparent materials with normal mapping
- Learning how to adjust a shader's **Render Queue** to allow Unity to render it in an appropriate order
- Adjusting the shadow-casting parameters of transparent objects

Starting the scene

The planet surface is represented by some basic primitives and other more complex models. In this section, we will be setting up environmental effects using transparent features available in the shaders:

1. In the **Project** panel, click on the `PACKT_Scenes` folder to view its contents in the **Assets** panel.
2. In the **Assets** panel, locate `Chapter5_Start` and double-click on it to load the scene:



Our starting point

The scene is currently set up with **Standard Shader** materials and default lighting. In the next step, we will add fog to make the environment more ambient.

Creating the dust cloud material

The surface of Ridley VI is made inhospitable by dangerous nitrogen storms. In our game scene, these are represented by dust cloud planes situated near to the surface.

Next, we need to set up the materials for these clouds, as follows:

1. In the **Project** panel, click on the `PACKT_Materials` folder to view its contents in the **Assets** panel.
2. In the **Assets** panel, right-click on an empty area and choose **Create** | **Material**.
3. Rename the material `DustCloud`.
4. In the **Hierarchy** panel, click to select the `dustCloud` game object.

   The object's properties will appear in the **Inspector**.

5. Drag the `DustCloud` material from the **Assets** panel to the **Materials** field in the **Mesh Renderer** property that is visible in the **Inspector**.

   Next, we will set the texture map for the material.

6. Reselect the `DustCloud` material by clicking on it in the **Assets** panel.
7. Lock the **Inspector** by clicking the small lock icon in the top right-hand corner of the panel.

Locking the Inspector allows you to maintain focus on assets while you are hooking up an associated asset in your project.

8. In the **Project** panel, click on the `PACKT_Textures` folder.
9. Locate the `strato` texture map and drag it to the `DustCloud` material's **albedo** texture slot in the **Inspector**.

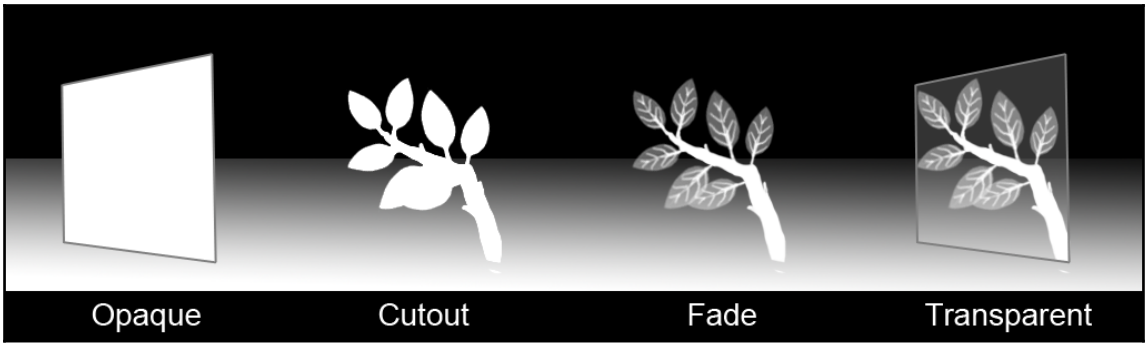
   The texture map contains four atlassed variations of the cloud effect. We need to adjust the amount of texture shown in the material.

10. In the **Inspector**, set the **Tiling Y** value to `0.25`.

This will ensure that only a quarter of the complete height of the texture will be used in the material.

The texture map also contains opacity data. To use this in our material, we need to adjust the **Rendering Mode**.

The Standard Shader's rendering mode allows us to specify the opaque nature of a surface.
Most often, scene objects are **Opaque**. Objects behind them are blocked by them and are not visible through their surface.
The next option, **Cutout**, is used for surfaces containing areas of complete opacity and complete transparency, such as leaves on a tree or a chain link fence. Opacity is basically on or off for each pixel in a texture.
Fade allows objects to have cutout areas where the pixels are completely transparent or partially transparent.
The **Transparent** option is suitable for truly transparent surfaces, such as windows, glass, and some kinds of plastic. When specular is used with a transparent material, it is applied over the entire surface, making it unsuitable for cutout effects.



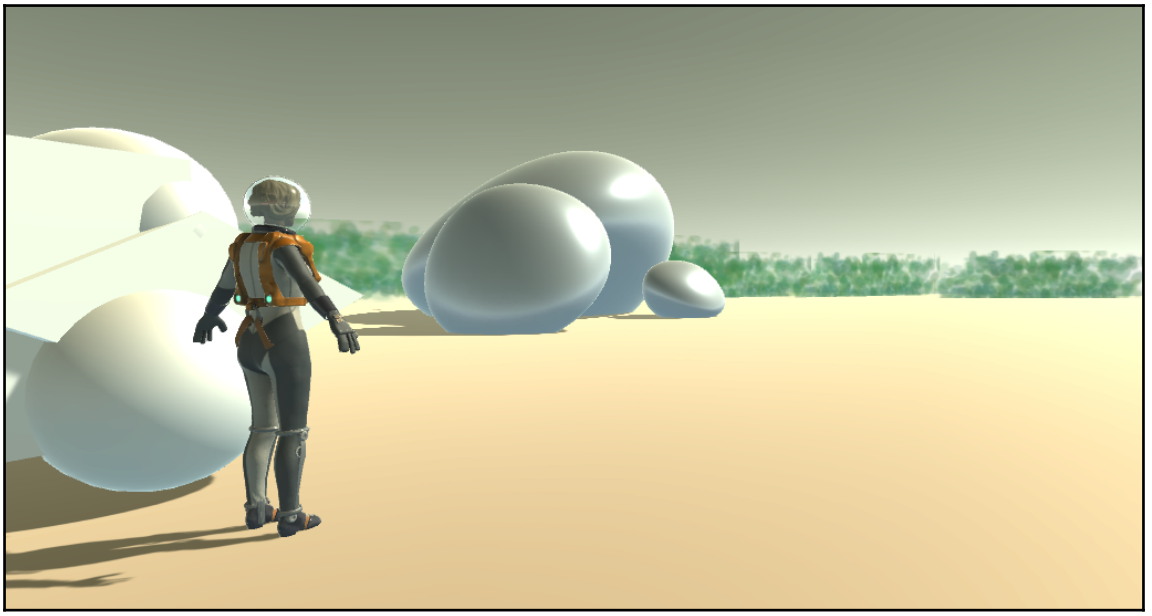
Comparison of Standard Shader transparency types

The **Fade** Rendering Mode is the best option for our `DustCloud` material as we want the cloud objects to be cutouts so that the edges of the quad where the material is applied to is not visible.

We want the surface to be partially transparent so that other `dustCloud` quads are visible behind them, blending the effect. We can make this change in the standard shader:

1. At the top of the material properties in the **Inspector**, click on the **Rendering Mode** drop-down menu and set it to `Fade`.

2. Add the `DustCloud` material to other `dustCloud` objects by dragging the material onto them in the **Hierarchy** panel.
   The dust clouds' appearance will change in the **Scene** and **Game** views:



Transparent DustCloud material applied

The dust clouds should now be visible with their opacity reading correctly, as shown in the preceding image.

In the next step, we will add some further environmental effects to the scene.

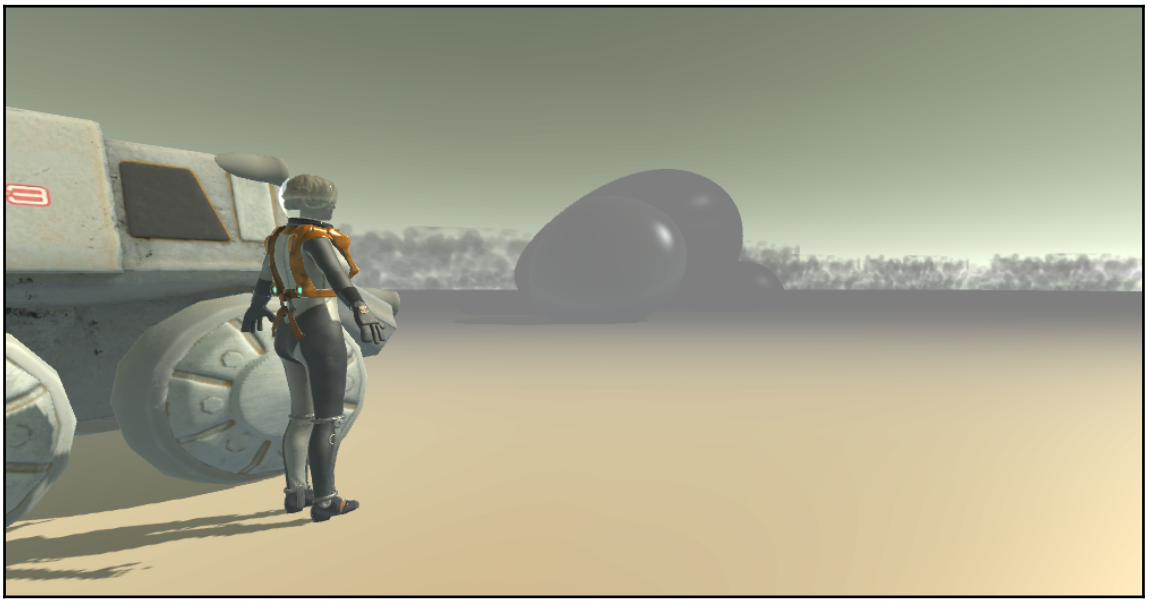
Adding fog to the scene

In this step, we will add fog to the scene. Fog can be set to fade out distant background elements to reduce the amount of scenery that needs to be rendered. It can be colored, allowing us to blend elements together and give our scene some depth:

1. If the **Lighting** tab is not already visible in the Unity project, activate it from the menu bar by going to **Windows** | **Lighting**.
2. Dock the **Lighting** panel if necessary.

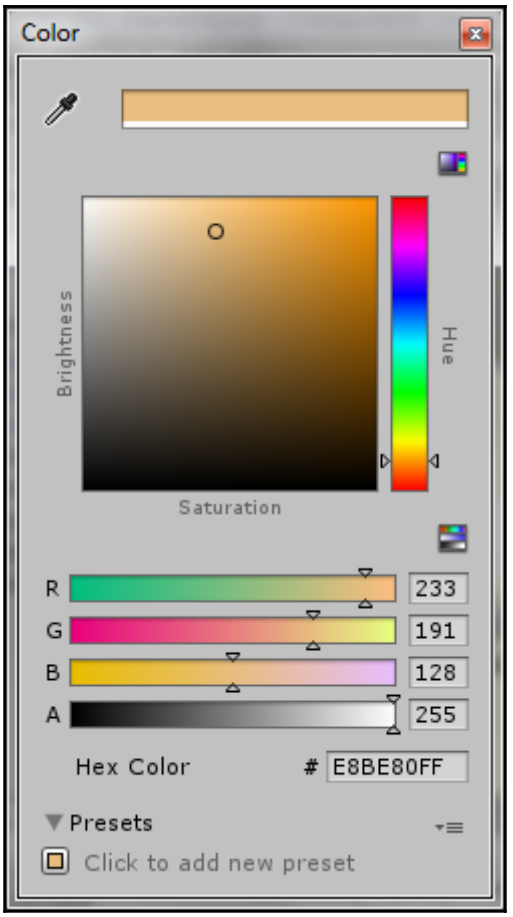
3. Scroll down to the bottom to locate the **Fog** properties group.
4. Check the checkbox next to **Fog** to enable it.

    You will see that fog is added to the environment in the **Scene** view as shown in the following image. The default values do not quite match what we need on the planet's surface environment:



Unity's default fog effect

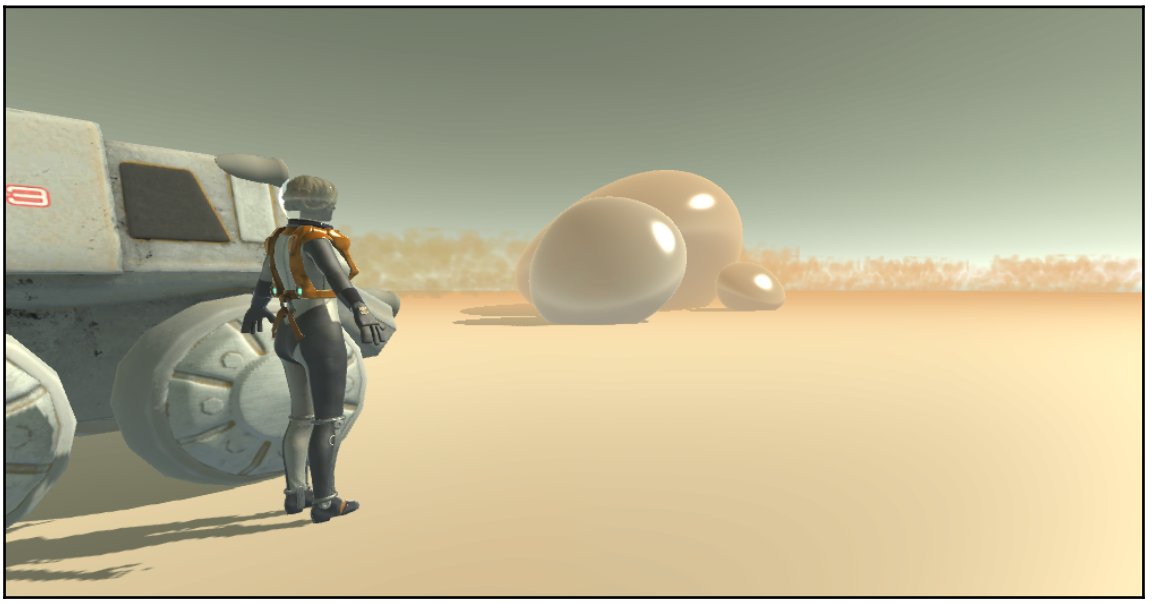
5. Click within the color swatch next to **Fog Color** to define the color value.
6. When the color picker appears over the main Unity interface, type the hex code `E8BE80FF` in the **Hex Color** field near the bottom, as shown in the following screenshot:



Fog effect color selection

This will define a yellow orange color that is appropriate for our planet's atmosphere.

7. Decrease the fog slightly by reducing the **Density** value to `0.05`.
   The fog will change in the **Game** view, showing the reduced density:



Adjusted fog blended with dust cloud transparencies

Our dust cloud objects are getting blended in with the fog as shown in the preceding image. They look nice at this point, but we want to animate their transparency.

We will now write a short script to do this.

Animating the dust cloud transparency

Now that we have applied a material to the dust clouds, we need to animate the effect by fading out the transparency and switching the texture atlas so that all four cloud variations are used.

We could animate this effect in the shader, but as the same shader is applied to all the dust clouds, the effect would be synchronous.

Materials are applied as instances to different objects in the scene, so if we manipulate the material rather than the shader, we can create a staggered effect for greater realism.

We can also animate certain material properties using the **Animation** window, but this would involve adding multiple components to each game object and it is also less time effective. There is more potential to automate this effect by writing some code:

1. In the **Project** panel, click on the `PACKT_Shaders` folder to view its contents in the **Assets** panel.
2. In an empty area of the **Assets** panel, create a new **C#** script asset by clicking right mouse button and choosing **Create | C# Script**.
3. Rename the script `cloudAnim`.
4. Double-click on `cloudAnim` to open it in MonoDevelop.

   Our first objective is to get the opacity of the objects to fade in and out.

   We will start by declaring some variables.

5. Add the following code directly after the script's opening curly bracket:

```
public float currOpacity;
public float fadeSpeed;
public bool ping;
public Renderer rend;
```

We will add a `currOpacity` float variable to keep track of the material reference's current state. This value will be defined later in the script.

We will also use a `float` value to define `fadeSpeed`, which will determine how quickly the dust clouds fade to completely transparent.

We want the cloud to fade in and out, so we will create a `ping` **Boolean** variable to allow us to determine whether the object is becoming more or less transparent.

Lastly, we will store a reference to the object's renderer with a `rend` variable.

In the `Start` state, we will initialize the dust cloud.

Add the following code inside the brackets of `void Start`:

```
rend = GetComponent<Renderer>();
currOpacity = Random.Range (0.05f, 0.95f);
```

The first line accesses the object's renderer component and stores it using the `rend` variable we just created.

In the next line, we will set the starting opacity to a random number. Set this a little above zero and a little under one respectively, so that the script does not get stuck in a loop.

We will put these elements together in the `Update` function.

Add the following code in the curly brackets of `void Update`:

```
Color currColor = new Color(1,1,1,currOpacity);
rend.material.SetColor("_Color", currColor);
if(ping)
{
    fadeSpeed = 0.35f;
}
else
{
    fadeSpeed = -0.35f;
}
currOpacity += Time.deltaTime * fadeSpeed;
if(currOpacity > 1f)
{
    ping = !ping;
}
else if (currOpacity < 0f)
{
    ping = !ping;
}
```

We will start the function by declaring a local variable to handle the `_Color` component of the shader. This is where the transparency is stored. We will define the color's RGBA values as `1`, `1`, `1`, and the value of `currOpacity` for the alpha. Putting this code in the `Update` function means that it will run every frame.

In the next line, we will use our rend variable to access the object's renderer and set the color using our updated `currColor`.

To increase the value, we will check whether the `ping` Boolean is `true` using an `if` statement. When this condition is `true`, we will set `fadeSpeed` to a positive value, `0.35`.

We will use the `else` statement to define the consequences that exist when `ping` is `false`, setting `fadeSpeed` to a negative value, `-0.35`.

The next line increases `currOpacity` by time in seconds, multiplied by `fadeSpeed`, increasing or decreasing the value.

Following this, we use an `if` statement to clamp the value range of the opacity by toggling the `ping` Boolean so that the `currOpacity` value ping pongs when it reaches its minimum and maximum.

We could optimize the last part of the script a little by putting two conditionals in the same `if` statement, but we will be adding a little more code to the last statement.

For now, we will test the effect in the scene:

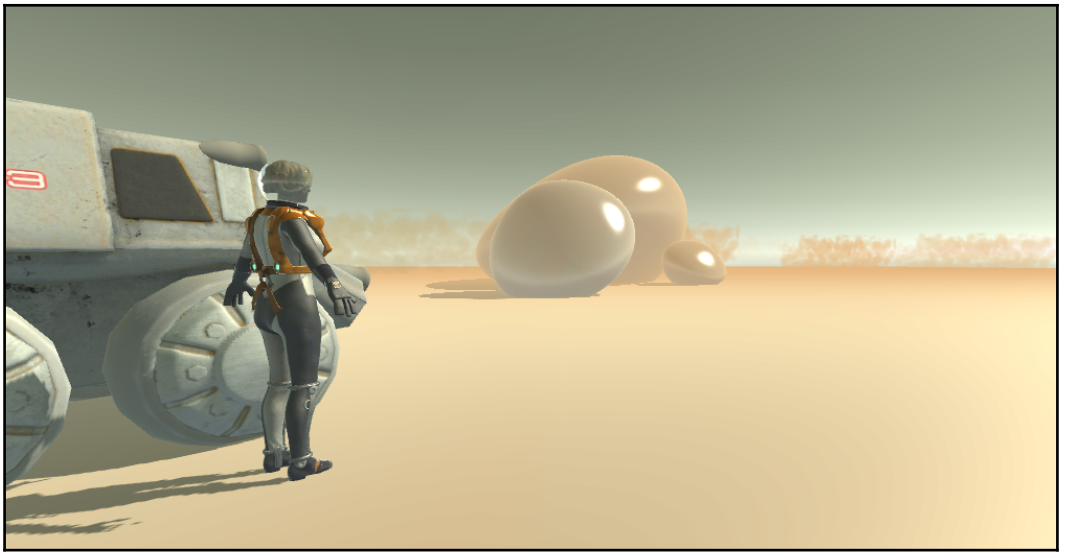
1. Save the script (*Ctrl* + *S*, or *command* + *S* if you are working on a Mac).
2. Minimize MonoDevelop.
3. In the **Hierarchy** panel, click to select the first `dustCloud` game object.

   Its parameters will appear in the **Inspector**.

4. Drag the `cloudAnim` script asset to the **Inspector** to add it to the game object.
5. At the top of the **Inspector**, click on the **Apply** button to save the prefab.

   The other `dustCloud` objects in the scene are instances, so changes made to one will be replicated.

6. At the top center of the main Unity interface, click on the **Play** button to see the dust clouds animate:



Fading dust cloud transparency

Each cloud slowly fades in and out, but the realism is marred by the recurrence of the same texture alpha.

We will fix this in the next step by offsetting the UV coordinates so that the cloud uses a different part of the texture atlas each time.

Switching the cloud texture atlas positions

The material's UV coordinates can be adjusted in the **Inspector**. We will add a few more lines of code to automate this process and get the clouds looking a little more organic:

1. Maximize MonoDevelop.
2. Add the following code at the bottom of the list of variables that we wrote earlier:

```
public float atlasPosition;
public float [] vValue = {0f, 0.25f, 0.5f, 0.75f};
```

   Here, we will add a new `atlasPosition` float variable to keep track of the material's vertical UV offset.

   Next, we add a `vValue` float array and assign values in increments of `0.25`. These represent the four positions of the cloud images in the texture atlas.

3. At the bottom of the `Start` function, add the following code:

```
atlasPosition = vValue[Random.Range(0, 3)];
rend.material.SetTextureOffset("_MainTex", new Vector2(0,
  atlasPosition));
```

Here, we define the value of `atlasPosition` as a number randomized from the `vValue` array. Potentially, this will cause each instance to start with a different image.

We access the game object's renderer that is already stored in our `rend` variable and use the `SetTextureOffset` method to set the starting offset for the `_MainTex` component.

We set the first value, which represents the horizontal (**U**) axis to zero and set the second value, representing the vertical (**V**) axis, to the value of `atlasPosition`.

1. Scroll down to the bottom of the script and locate `else if (currOpacity < 0f)` in the `Update` function.

2. Add the following line of code inside the statement's curly brackets:

```
ChangeVPos();
```

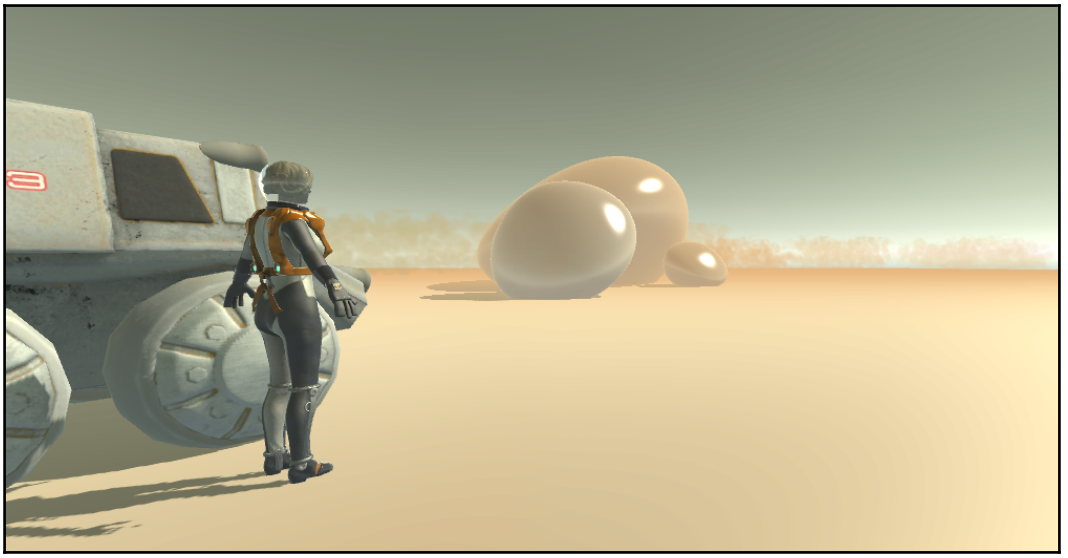
   This line will run a new custom function, where we will change the vertical UV offset. We will separate this from the other code as it does not run every frame, just when the cloud object reaches zero opacity.

3. Add the following code after the `Update` function's closing curly bracket:

```
void ChangeVPos()
{
    atlasPosition = vValue[Random.Range(0, 3)];
    rend.material.SetTextureOffset("_MainTex", new Vector2(0,
      atlasPosition));
}
```

   Here, we set `atlasPosition` using the same lines of code that we used to initialize the dust cloud. This will switch it to a different cloud image within the texture atlas.

4. Save the script.
5. Minimize MonoDevelop and preview the effect by pressing the **Play** button at the top center of the Unity interface:



Varied dust cloud transparency

The dust clouds fade in and out like before, but this time, they fade back in with a different cloud variation, creating a more believable environment, as shown in the preceding image.

The following is the final code of the `cloudAnim` script:

```
using UnityEngine;
using System.Collections;

public class cloudAnim : MonoBehaviour
{
    public float currOpacity;
    public float fadeSpeed;
    public bool ping; //to pingpong the transparency effect
    public Renderer rend;
    public float atlasPosition;
    public float [] vValue = {0f, 0.25f, 0.5f, 0.75f};

    void Start ()
    {
        rend = GetComponent<Renderer>();//set startOpacity
        currOpacity = Random.Range (0, 0.95f);
        atlasPosition = vValue[Random.Range(0, 3)];
        rend.material.SetTextureOffset("_MainTex", new Vector2(0,
          atlasPosition));
    }
    void Update ()
    {
        Color currColor = new Color(1,1,1,currOpacity);
        rend.material.SetColor("_Color", currColor);
        if(ping)
        {
            fadeSpeed = 0.35f;
        }
        else
        {
            fadeSpeed = -0.35f;
        }
        currOpacity += Time.deltaTime * fadeSpeed;
        if(currOpacity > 1f)
        {
            ping = !ping;
        }
        else if (currOpacity < 0f)
        {
            ping = !ping;
            ChangeVPos();
        }
    }
```

```
        void ChangeVPos()
        {
            atlasPosition = vValue[Random.Range(0, 3)];
            rend.material.SetTextureOffset("_MainTex", new Vector2(0,
              atlasPosition));
        }
    }
```

Back in `Chapter 2`, *Creating Custom Shaders*, we improved the appearance of the astronaut's helmet by creating a two-sided glass material. We can improve this by making a more detailed and responsive PBR-based glass material.

Creating a better transparent glass material

In `Chapter 2`, *Creating Custom Shaders*, we set the default shader code for the helmet shader to transparent, rather than opaque, and created an additional pass to separate the front and back surfaces and create more realistic specularity from inside the helmet rather than just the outside.

Let's focus on the astronaut in the scene to take a closer look at the helmet surface:

1. In the **Hierarchy** panel, locate the astronaut game object.
2. Click on the small arrow next to its name to expand its **Hierarchy**.
3. Double-click on the `helmet` game object to focus on it in the **Scene** view:

Initial astronaut helmet glass

Now that we have worked with some different transparent shader types, let's take this shader to the next level:

1. In the **Project** panel, click on the `PACKT_Shaders` folder to view its contents in the **Assets** panel.
2. Locate the `Glass` shader and double-click on it to open it in MonoDevelop.

   At the top of the shader is the `Tags` line of `subShader`:

```
Tags { "RenderType"="Opaque"}
```

   This line defines how Unity will treat the shader. The default value is `Opaque`, but we can also use `Transparent`, though this is used for shader replacement and some camera-depth functions, rather than actually affecting the material's appearance in the game.

3. Before the `Tags` closing curly bracket, add the following code:

```
"Queue" = "Transparent"
```

The **Render Queue** defines when the pixels associated with a shader will be rendered. By default, Unity will render all the pixels at the same time, but with transparent objects, this means that opaque objects behind them will cull them.

We typically want to render transparent pixels after opaque pixels so that they appear to be transparent, and that is what Unity does in this case.

Background: Pixels using shaders marked with this queue tag are rendered before all others. This is used for skyboxes and other environmental features.
Geometry: This is the default queue tag and is mainly used for solid objects.
AlphaTest: Objects with **Cutout** transparency use this tag.
Transparent: This is used for true transparent surfaces and some particle effects.
Overlay: This queue tag is used for some camera and lighting effects.

In addition to these built in **Render Queue** tags, we can also fine tune shaders by giving them additional numbers, such as `Geometry+1`.

Next, we will start by adding some tiny surface scratches on the helmet glass with a normal map. This will make the helmet reflection less perfect:

1. At the end of the **Properties** block, add the following line of code:

```
_BumpMap ("Normalmap", 2D) = "bump" {}
```

   This will handle the normal map input.

   We also need to set up the shader variable to use the map in Cg.

2. Locate the `sampler2D _MainTex` line.
3. Add the following line of code just under it:

```
sampler2D _BumpMap;
```

4. Scroll down to the `surf` function, still in the same pass, and add the following line:

```
o.Normal = UnpackNormal (tex2D(_BumpMap, IN.uv_MainTex));
```

   We can omit this code from the back face pass. It would look odd for identical scratches to be on the inner surface of the glass.

5. Save the shader.
6. Minimize MonoDevelop and return to the main Unity interface.
7. In the **Project** panel, click on the `PACKT_Textures` folder to view its contents in the **Assets** panel.
8. Locate the `astronaut_helmet_normal` texture asset.

9. Drag this to the **Normal Map** slot in the `Helmet` material that is now visible in the **Inspector**:

Normal mapped helmet effect

The helmet's normal map gives the appearance of a more realistic surface and reduces the uniformity of the reflection.

In the next section, we will create a more dramatic weather effect for the planet surface scene.

Setting up the whirlwind effect

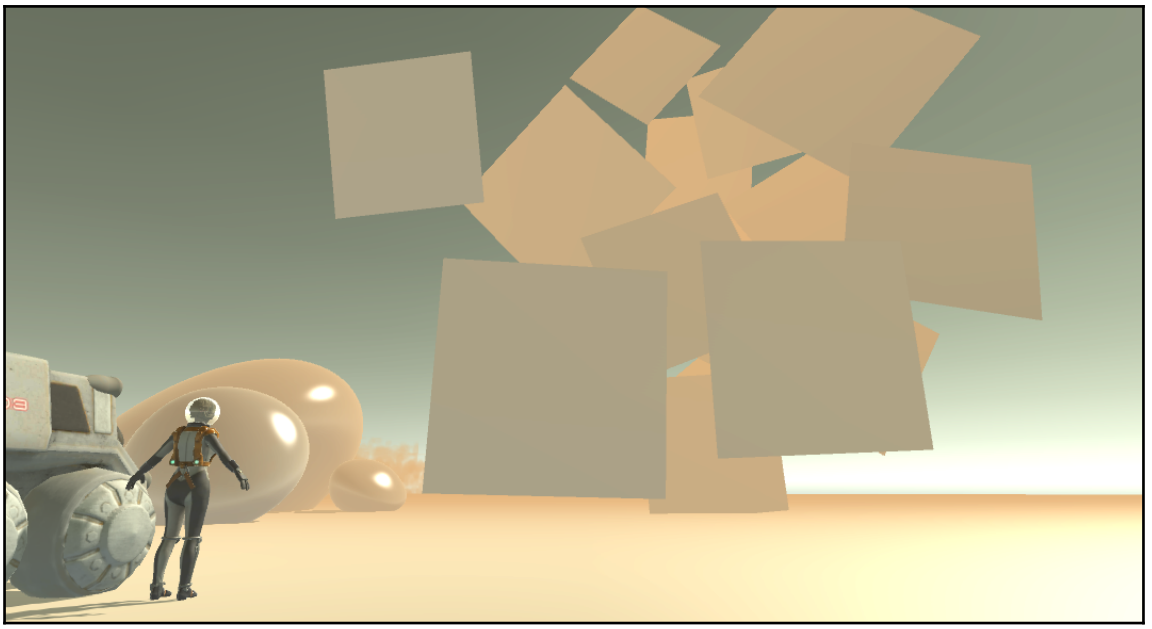
The scene contains a whirlwind effect that is animated with a simple script. In the next step, we will activate the `whirlwind` game object and preview it in the game view:

1. In the **Hierarchy** panel, select the `whirlwind` game object.

   This game object is currently inactive.

2. In the **Inspector**, check the checkbox next to its name.

    The object will appear in the scene:



The whirlwind game object in the scene

Now let's talk about the object hierarchy and how the material is set up.

Each of the planes in the whirlwind object is UV mapped with the same coordinates, ensuring that they show the texture in the same way.

Objects are rendered in a specific order. Opaque objects are rendered first, then alphatesting (cutout objects), and finally, transparent objects are rendered over the top. When it comes to transparent objects, it would be normal to render distant objects first, but we usually adjust this to render closer triangles so that they are not occluded by other triangles behind them.

Let's demonstrate this with a custom shader that will give the distant triangles in the whirlwind more opacity and a stronger tint color.

1. In the **Project** panel, click on the `PACKT_Shaders` folder to view its content in the **Assets** panel.
2. In an empty area of the **Assets** panel, create the new shader. Right-click and choose **Create** | **Shader** | **Standard Surface Shader** from the drop-down list.
3. When it is created in the **Assets** panel, rename the shader `Wind`.
4. Double-click on the new shader to open it in MonoDevelop.
5. We will start by amending the shader's name so that it is selectable from our main shader's folder.
6. Adjust the first line to read as follows:

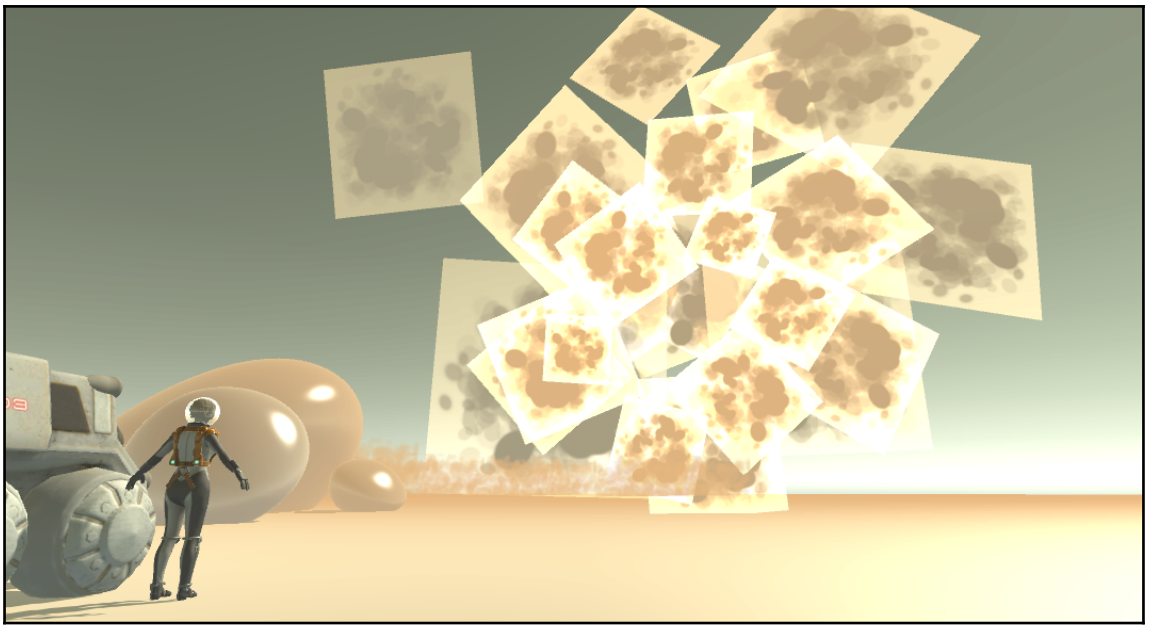
```
Shader "PACKT/Wind" {
```

Next, we will make the geometry transparent.

7. Locate the `RenderType` tag and change it to `Transparent`:

```
Tags { "RenderType"="Transparent"}
```

8. Locate the shader directive and add the `alpha` keyword at its end.
9. Save the shader.
10. Return to the main Unity interface.
11. Click the small arrow next to the `whirlwind` game object in the **Hierarchy** panel to expand it.
12. Select the `stormPlanes` subobject and scroll to the bottom of its components in the **Inspector** panel.
13. In the `PACKT_Textures` folder locate the storm texture asset.
14. Drag this into the albedo slot in the `storm` material. The quads that make up the model should now appear partially transparent in the **Scene** and **Game** views:

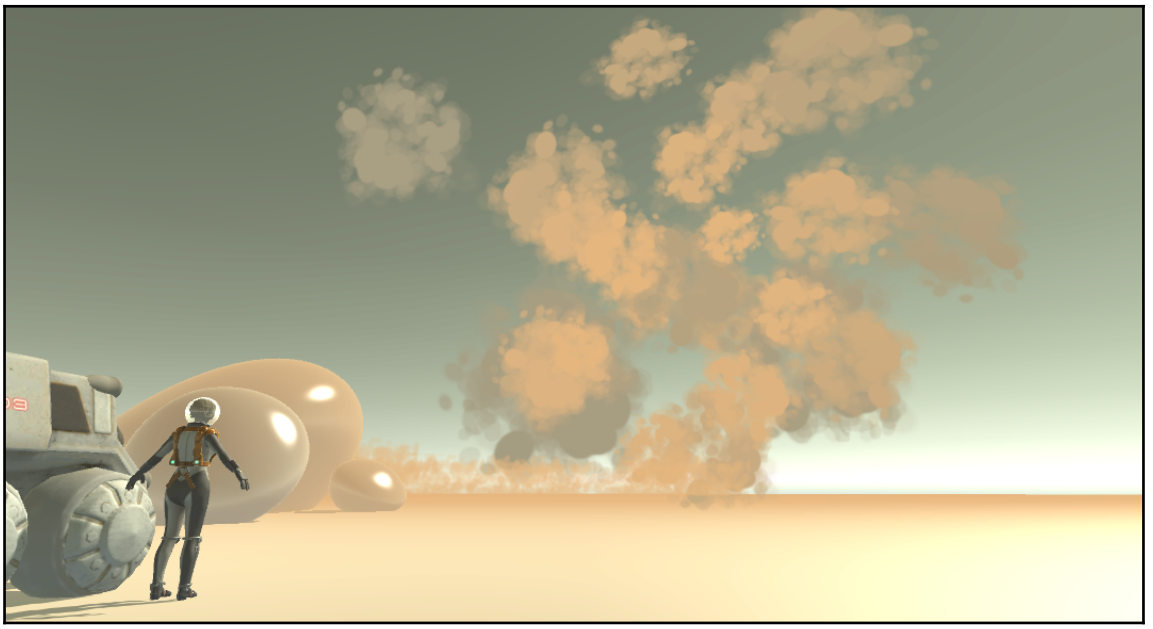


The whirlwind game object in the Game view

The planes in the whirlwind object have now become transparent, but the planes are currently lit. We can make an adjustment to the shader to fix this.

15. Back in MonoDevelop, replace the `alpha` keyword that we just added with `alpha:fade`.
16. Save the shader.

If you look at the **Game** view now, you should see that the whirlwind's appearance has changed again:



The whirlwind game object with the alpha:fade shader applied

The planes have retained their transparency, but now the edges of the quads are no longer visible. This is another useful built-in function.

Summary

In this chapter, we explored transparent material types and how they can be used with scene lighting and environmental effects to create appealing outdoor environments.

We took a closer look at material **Rendering Modes** and how transparent effects can be implemented in a scene.

We further explored real-time environmental effects by creating dust clouds that fade in and out using the atlassed textures.

We then set up an environmental fog effect using Unity's built-in tools.

We then created a more realistic transparent material for the astronaut's helmet by adding a normal map for the outer surface, simulating pits, and small scratches.

Our final objective was the creation of an animated weather effect that uses transparencies and a custom shader.

In the next chapter, we will compare the specialized specular and metallic workflows and demonstrate how these can be used with real-time physically-based shading to add variety and realism to a game scene.

6
Working with Specular and Metallic Surfaces

In this chapter, we will examine the key differences between metallic and specular workflows that are used in Unity's **Physically-Based Shading**.

We have used these shaders and variants of our own to create various materials in the previous chapters, but what is really going on beneath the surface? How are metallic and specular materials set up to respond differently to light? We will answer these questions as we work through the objectives in this chapter.

In this chapter, we will cover the following topics:

- Comparing metallic and specular workflows
- Discussing shader dependence on lighting and reflection data
- Adding new properties to a shader
- Adding and utilizing unique UV coordinates
- Accessing and modifying shader properties with scripts
- Modifying material instances at runtime
- Locating and utilizing Unity's built-in shader assets in `CGIncludes`

Let's take a look at how metallic and specular features can be added manually and what kind of surfaces we can achieve outside the Standard Shader's range.

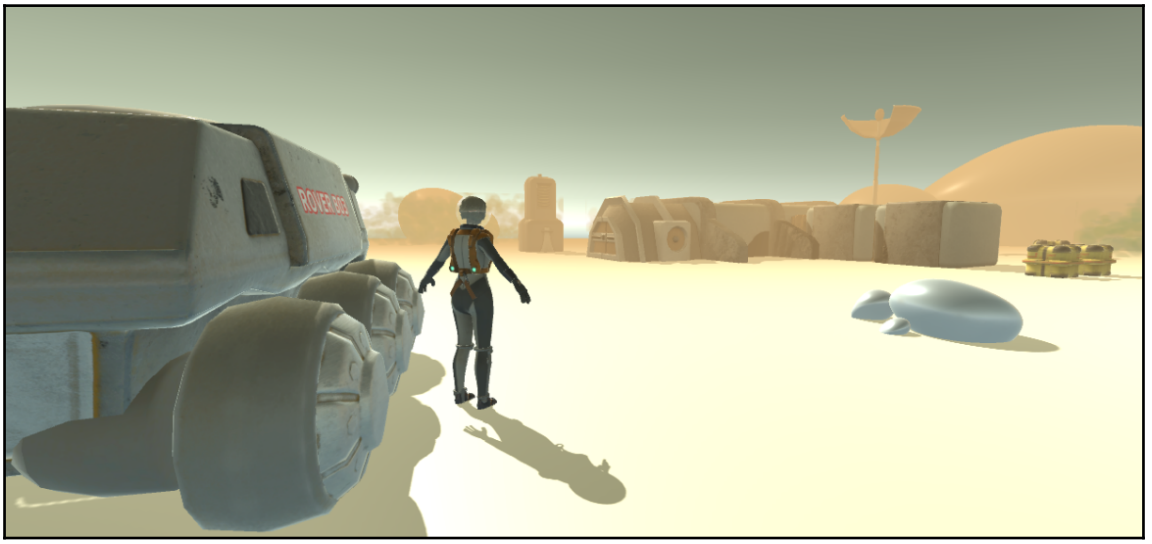
At this point in the game, our character reaches the damaged research station and prepares to enter. From outside, the research center appears to be intact.

Starting the scene

The first scene file of this chapter contains the game objects and lighting setup that we need to get started.

Using the menu bar at the top of the Unity interface, open the `Chapter6_Start` scene by navigating to **File** | **Open Scene** | **Chapter6_Start**.

The following scene shows the research station on the astronaut's arrival:



The starting scene's initial state

In the next step, we will take a closer look at the crate model's material setup in the scene.

Altering the crate's secondary material at runtime

The crate props in our scene use the standard material with an albedo, metallic, and normal map. As the crate props use the same material, the texture is the same on all the crates.

For this model, we want to create a slightly unique texture for each crate by applying a number to the albedo texture.

Let's start by testing the limits of the Standard Shader, firstly by applying the decal to the crate.

Applying a secondary albedo texture

Back in `Chapter 1`, *Getting to Grips with Standard Shaders*, we applied some painted detail to the spacecraft using a secondary map. We will start with the same technique here:

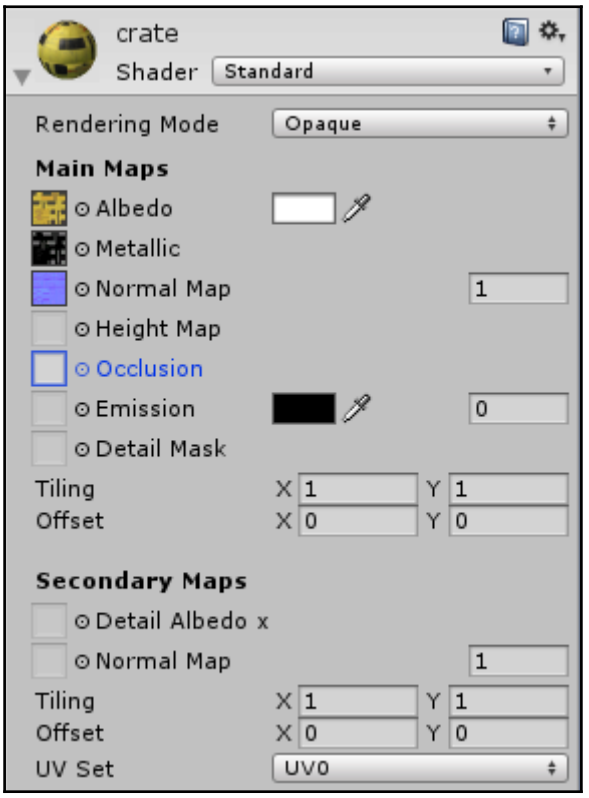
1. In the **Hierarchy** panel, double-click on `crate01` to zoom in on it in the **Scene** view:

The crate model with its initial material

   The crates have a standard material applied to them. The material currently uses an albedo, metallic, and normal map.

2. In the **Inspector**, scroll down until the material properties are visible.

3. Click on the small arrow next to the `crate` material's name to see how the maps are applied:



The crate material parameters in the Inspector

We need to assign the number decal next.

4. In the **Project** panel, click on the `PACKT_Textures` folder to view its contents in the **Assets** panel.

5. Double-click on the `researchStation_ext` subfolder to open it.
6. In the **Assets** panel, locate the `number3` asset and drag it to the crate material's **Detail Albedo x** map slot in the **Secondary Maps** group in the **Inspector**.

You will see the number added to the texture in the **Scene** view:

The crate material with detail texture applied

Currently, all the pixels of PNG are being added to the texture. We need to fix the transparency by applying the alpha mask to the secondary texture.

7. Drag the `number3` asset onto the **Detail Mask** slot in the **Inspector**.

The numbers are now blended a lot more nicely:

The crate's secondary texture with Detail Mask applied

The next step is to scale down and position the number so that it fits within the texture.

At this point, we may run into an issue with the Standard Shader. **Detail Mask** is set up to use the same UV coordinates as the material's base textures.

If we try to change the tiling or UV offset, all the maps are affected.

The best solution is to write another shader in which we have a separate set of coordinates.

Creating a custom decal shader for the crate

We will begin writing our new decal shader so that we can apply the crate number correctly:

1. In the **Project** panel, double-click on the `PACKT_Shaders` folder to view its content in the **Assets** panel.
2. Create a new shader asset by right-clicking on an empty part of the panel and going to **Create** | **Shader** | **Standard Surface Shader**.
3. Rename the new shader `Decal`.
4. Double-click on it to open it in MonoDevelop.
5. Change the shader name definition in the first line to the following:

```
Shader "PACKT/Decal" {
```

   This will ensure that the shader appears in the same list as the others that we have created.

6. Delete the **Properties** definitions for `_Glossiness` and `_Metallic` and replace them with the following:

```
_Metallic ("Metal (R), Smoothness (A)", 2D) = "white" {}
_DecalTex ("Decal Albedo (RGB)", 2D) = "white" {}
```

   In the default shader base, **Metallic** and **Glossiness** properties are defined by sliders and we want to use a map such as the Standard Shader.

   Smoothness uses the alpha channel of the Metallic texture map, so it does not need its own property.

   The property definition for the decal, `_DecalTex`, is identical to the main texture.

   After the start of the Cg snippet, you will see the `Sampler2D_MainTex` property.

7. Directly after this, add the following lines:

```
sampler2D _Metallic;
sampler2D _DecalTex;
```

These lines will allow us to use the texture maps in the shader.

Within the `input` struct, add the following code:

```
float2 uv_DecalTex;
```

We will add a float variable to handle the additional UV coordinates of the decal texture. It is not necessary to add these for the Metallic map, because it uses the same UV coordinates as `_MainTex`.

8. Delete the variable definitions for `_Glossiness` and `_Metallic`, as these are now handled by a texture map.
9. In the `surf` function that follows this, locate the following line of code:

```
fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
```

This line defines a variable named `c`, which combines the texture map with the tint color in the shader. It applies the texture using the UV coordinates. We can do something similar for the decal texture.

This is the method by which bitmap textures (which require UVs) are transferred into the `input` struct.

10. Add the following line of code directly after the `fixed4 c` line:

```
fixed4 d = tex2D (_DecalTex, IN.uv_DecalTex);
```

In this case, we just want to apply the texture with the appropriate UV coordinates. We will do the blending in a different line later on.

11. Next, add the following code:

```
fixed4 m = tex2D (_Metallic, IN.uv_MainTex);
```

The `_Metallic` map is applied using the `_MainTex` UV coordinates. We have no reason to offset it.

The next line in the shader defines the output albedo:

```
o.Albedo = c.rgb;
```

Let's add the decal texture to this.

12. Replace the `o.Albedo` line with the following code:

```
o.Albedo = lerp (c.rgb, d.rgb, d.a);
```

    Rather than adding the two textures together here, we will use the `lerp` operator to blend the two sets of RGB values. The result is a linear interpolation of `c.rgb` and `d.rgb` with the weight of `d.a`.

    This has the effect of subtracting the decal's alpha channel so that the void areas of the decal do not affect the original texture.

13. Finally, replace the output metallic and glossiness output with the following:

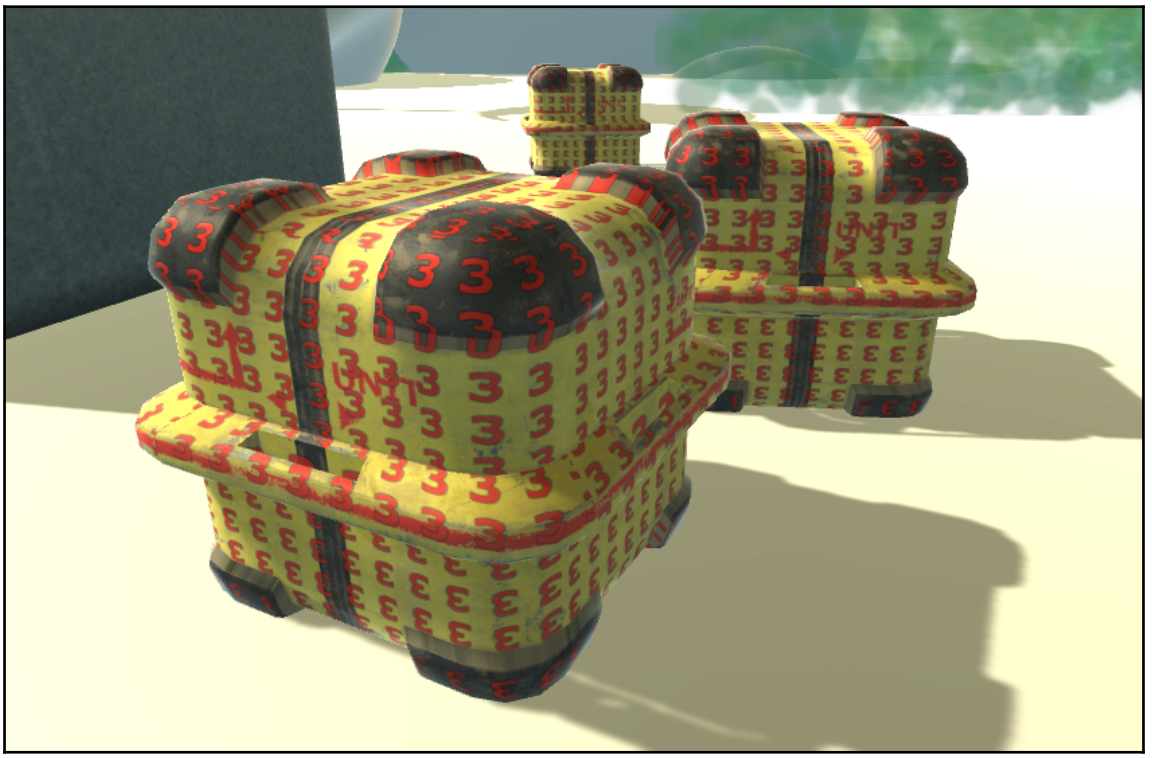
```
o.Metallic = m.r;
o.Smoothness = m.a;
```

    This will make them use RGB and channels of the `_Metallic` texture map respectively.

14. Save the shader.
15. Back in the main Unity interface, make sure that the `crate` material properties are still visible in the **Inspector**.
16. Click on the **Shader** selection dropdown at top of the **Material Properties** and select **PACKT | Decal**.
17. Drag the `number3` texture to the decal texture slot.

18. Increase the **Tiling X** and **Y** values to `19.3`.

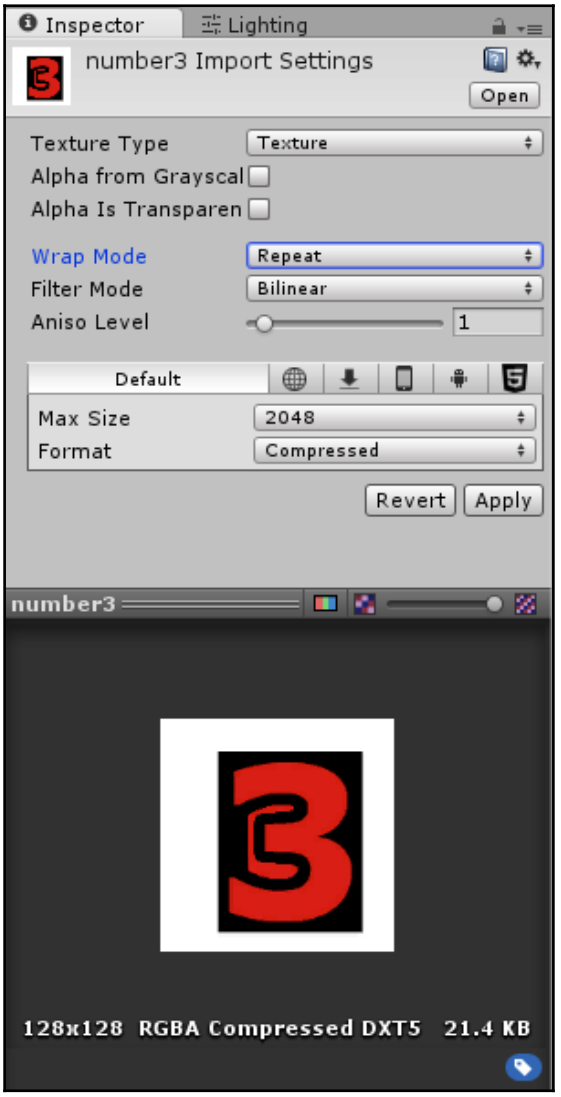
    The decal texture's UV coordinates can be manipulated separately from those of the main texture:



The crate model with the new decal shader

The `number3` decal is now tiling 19.3 times across and 19.3 times down our texture. Textures assets are set to tile by default, we need to adjust an import setting to get it to only show up once.

19. Click on the `number3` asset in the Assets panel to view the texture's import settings:



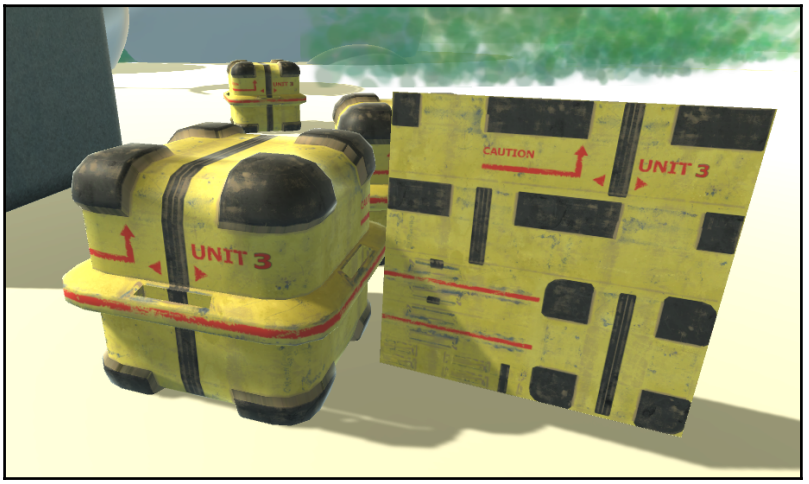
The number3 texture asset's default import settings

20. Set the **Wrap Mode** to `Clamp` and click on the **Apply** button to store the change.

    The number decal will disappear. Actually, it is being added to a part of the texture that we cannot see on screen.

    One way to make the placement of the decal easier is to show the material on a flat quad. This way, we can line up the decal with the other details in the texture without having to worry about the geometry.

21. Create a quad using the menu bar by navigating to **GameObject** | **3D Object** | **Quad**.
22. Move it so that it is visible in the **Scene** view.
23. Locate the crate material again by navigating to the `PACKT_Materials` folder.
24. Drag the `crate` material to the quad in the **Scene** view or **Hierarchy** panel.
25. Use the offset and scale controls to line up the `number3` decal on the main texture:



Completed decal setup with quad display

The final values used here are as follows:

Tiling X: `19.3`, **Tiling Y**: `19.3`

Offset X: `-16.4`, **Offset Y**: `-15.6`

In the next step, we will switch the number decal.

Switching the decal texture at runtime

Now that we have a decal added to the `crate` material, we can switch the actual number at runtime with a script:

1. In the **Project** panel, double-click on the `PACKT_Scripts` folder to view its contents in the **Assets** panel.
2. Create a new C# script in an empty part of the panel. Right-click and choose **Create | C# Script** from the drop-down list.
3. Rename the script `crateNumbers`.
4. Double-click on it to open it in MonoDevelop.

In the MonoDevelop window, we will start by creating some variables.

5. Directly inside the script's opening curly bracket, add the following code:

```
public GameObject [] crateObjects;
public Texture2D [] crateTextures;
public int arrayIndex;
```

We will create an array of game objects named `crateObjects`. This allows us to keep track of the crates that we want to apply the script to.

Next, we create a texture array named `crateTextures` to store the number textures.

Our last variable, `arrayIndex`, stores the position in these arrays. We will apply a different number to each crate, so we only need one index number:

1. In the `Start` function, add the following code:

```
foreach (GameObject crate in crateObjects)
{
    arrayIndex++;

crate.GetComponent<Renderer>().material.SetTexture("_DecalTex",
      crateTextures[arrayIndex]);
}
```

Here, we will add a `foreach` statement that will loop through all the game objects in the `crateObjects` array and run the same code on each object.

Firstly, we will access the selected object's renderer and set its `_DecalTex` to the `crateTexture` array's current position.

Then, we will increase `arrayIndex` so that the game object and texture are different next time the code is run.

2. Save the script.
3. Minimize MonoDevelop.
4. Back in the main Unity interface, select the `crate_CTRL` object in the **Hierarchy** panel.
5. Drag the `crateNumbers` script onto this object.

   It may take a moment for the public variables to pop up in the **Inspector** as the editor reads the script.

6. At the top right of the `Inspector`, click on the lock icon to maintain focus on the `crate_CTRL` object's properties.
7. In the **Hierarchy** panel, select all of the crate objects and drag them to the **Crate Objects** array within the **Crate Numbers (Script)** component in the **Inspector**.

   This will add the crates to the script's array.

8. In the **Project** panel, double-click on the `PACKT_Textures` folder to view its contents in the **Assets** panel.
9. Double-click on the `researchStation_ext` subfolder in the **Assets** panel.
10. Select `number1`, `2`, and `3` and drag these onto the **Crate Textures** array to populate the array.

The order that you select items in a multiple selection will affect the order in which they are assigned.

11. Press the play button to view the effect:

Crate material instances displaying unique decal numbers at runtime

When the game runs, the numbers on each of the crates are unique. This works because the material assigned to a mesh renderer at runtime is a unique instance of the material in the project.

In the next step, we will take a closer look at Unity's built-in light models.

Locating and modifying shader light models

When we have created shaders so far, we have relied on the default template that Unity provides and is easily accessible in the **Assets** panel.

Let's start here and identify the part of the shader that handles scene lighting.

We will start by creating a default shader:

1. In the **Project** panel, click on the `PACKT_Shaders` folder to view its contents in the **Assets** panel.
2. In an empty area of the **Assets** panel, create a new default shader. Right-click and navigate to **Create** | **Shader** | **Standard Surface Shader** from the drop-down list.
3. Rename the shader `TestLambert`.
4. Double-click on the new shader to open it in MonoDevelop.
5. In MonoDevelop, replace the first line with the following code:

```
Shader "PACKT/TestLambert" {
```

6. Locate the shader compile directives. This is the code within the Cg snippet that will be highlighted in magenta by default:

```
            // Physically based Standard lighting model, and enable shadows
on all light types
            #pragma surface surf Standard fullforwardshadows

            // Use shader model 3.0 target, to get nicer looking lighting
            #pragma target 3.0
```

   The lines of comment outline the function of the code, but the keywords actually refer to some external code that handles the lighting.

   The `Standard` keyword defines a set of inputs that the shader uses, which is also accessed from an external source.

   This code is contained in a Cg file called `UnityPBSLighting.cginc`, which is located in the following path within the Unity program file location on your machine:

   Editor | **Data** | **CGIncludes**.

7. Navigate to this folder and open this file in MonoDevelop by double-clicking on it.

8. When the file opens in MonoDevelop, scroll down until you see the following code:

```
struct SurfaceOutputStandard
{
    fixed3 Albedo; // base (diffuse or specular) color
    fixed3 Normal; // tangent space normal, if written
    half3 Emission;
    half Metallic; // 0=non-metal, 1=metal
    half Smoothness; // 0=rough, 1=smooth
    half Occlusion; // occlusion (default 1)
```

This data structure defines the output types that we have been using in the Standard Shaders. Variables such as `fixed3` are called packed arrays; `fixed3` is a collection of three fixed variables.

The variables `float`, `half`, and `fixed` are all actually float variables of different precision.

These are outlined really well in Unity's official documentation, available at `https://docs.unity3d.com/Manual/SL-DataTypesAndPrecision.html`.

To summarize:

- `float`: 32 bits, highest level of precision
- `half`: 16 bits with three decimal digits of precision
- `fixed`: 11 bits, lowest level of precision

These variables are used in the shader based on how they are used. Typically, higher level calculations, such as those involving world coordinates, are done at a higher precision.

We will discuss these factors in `Chapter 8`, *Optimizing Shaders for Mobile*.

Next in the code is the lighting model:

```
inline half4 LightingStandard (SurfaceOutputStandard s, half3 viewDir,
UnityGI gi)
{
    s.Normal = normalize(s.Normal);
    half oneMinusReflectivity;
    half3 specColor;
    s.Albedo = DiffuseAndSpecularFromMetallic (s.Albedo, s.Metallic,
/*out*/ specColor, /*out*/
      oneMinusReflectivity);
    // shader relies on pre-multiply alpha-blend (_SrcBlend = One,
_DstBlend = OneMinusSrcAlpha)
    // this is necessary to handle transparency in physically correct way -
```

```
only diffuse component gets
      affected by alpha
    half outputAlpha;
    s.Albedo = PreMultiplyAlpha (s.Albedo, s.Alpha, oneMinusReflectivity,
/*out*/ outputAlpha);
    half4 c = UNITY_BRDF_PBS (s.Albedo, specColor, oneMinusReflectivity,
s.Smoothness, s.Normal,
      viewDir, gi.light, gi.indirect);
    c.rgb += UNITY_BRDF_GI (s.Albedo, specColor, oneMinusReflectivity,
s.Smoothness, s.Normal,
      viewDir, s.Occlusion, gi);
    c.a = outputAlpha;
    return c;
}
```

This block of code defines how the shader reacts to the light, it includes global illumination, which in this case is calculated in `UnityGI`, another external Cg file. In the sixth and seventh lines of proper code, we see that the lighting calculation uses `UNITY_BRDF_PBS` and `UNITY_BRDF_GI`. These definitions are included in an external file and assembled here using the surface output components.

The acronym **BRDF** stands for **Bidirectional Reflectance Distribution Function**. This is the lighting function that processes the scene light direction and strength with the surface normal and view direction to correctly affect the surface.

In the next step, we will switch to a simpler lighting model that we can fit entirely in our shader.

Modifying the shader lighting model

To see the lighting model working directly in our shader, we need to start with the compile directive:

1. In MonoDevelop, return to the `TestLambert.shader` tab.
2. Locate the shader compile directive:

```
#pragma surface surf Standard fullforwardshadows
```

3. Replace this with the following:

```
#pragma surface surf Lambert
```

Lambert is one of the simpler and cheaper lighting models as it does not include surface highlights.

If we save the shader at this point, Unity will show a syntax error in the console panel.

We need to define properties and surface output that are compatible with the lighting model.

We will start with the **Properties** section:

1. In the `Properties` block, near the top of the shader, delete the following lines of code:

```
_Glossiness ("Smoothness", Range(0,1)) = 0.5
_Metallic ("Metallic", Range(0,1)) = 0.0
```

   We should also delete the variables that are created to accommodate these properties.

2. Select and delete the following lines of code:

```
half _Glossiness;
half _Metallic;
```

3. In the `surf` function that follows the variables, remove the `Standard` keyword from the opening line.

   Omitting this will tell Unity to use the built-in `SurfaceOutput` struct for this shader.

   We have a few more changes to make to the `surf` function.

4. Select and delete the following lines of code:

```
o.Metallic = _Metallic;
o.Smoothness = _Glossiness;
```

5. Save the shader.

At this point, there should be no compile errors in the main Unity interface.

The shader compiled because we used another built-in lighting model by using the Lambert keyword in our compile directive.

The code that defines this model is located in another `CGIncludes` file called `Lighting.cginc`.

The advantage of separating lighting models and shader inputs is that it will reduce the processing time when multiple shaders share the same code.

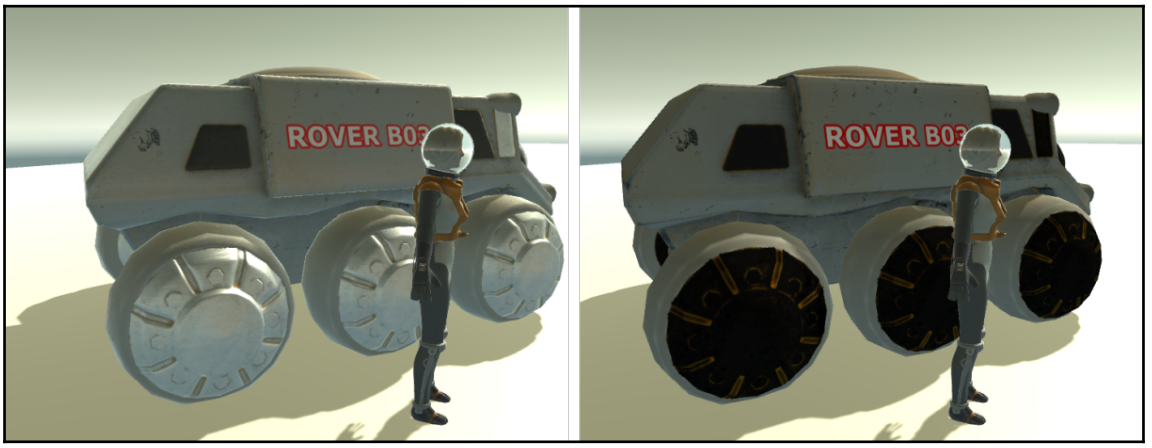
We can apply the shader to an existing scene material to see how it looks:

1. In the **Hierarchy** panel, select the rover game object.

   The object's components will appear in the **Inspector**.

2. Scroll down until the rover's **Mesh Renderer** component is visible.
3. Click on the small arrow next to the `Rover` material to expand it.
4. Switch to the new shader by clicking the **Shader** button and choosing **PACKT | TestLambert** from the drop-down list.

   You will see some major differences in the **Scene** view:



The rover model with Standard Specular material (left) and TestLambert material (right)

   The metallic parts of the rover have become dark as this part of the model is handled mainly by the specular and glossiness channels of the original shader, and these are not present in the `TestLambert` shader.

   In the next step, we will get all the external code into our shader so that we can make further changes.

5. Locate this file within the `CGIncludes` folder and open it in MonoDevelop.

6. First, locate the `SurfaceOutput` struct:

```
struct SurfaceOutput {
    fixed3 Albedo;
    fixed3 Normal;
    fixed3 Emission;
    half Specular;
    fixed Gloss;
    fixed Alpha;
};
```

   This set of variables is defined as those used by the shader, so we need to copy this code into ours.

7. Select and copy the block of code.
8. Return to the `TestLambert.shader` tab in MonoDevelop and paste the code right after the shader variables.

The `SurfaceOutput` struct actually contains a bunch of variables (`Normal`, `Emission`, `Specular`, and `Gloss`) that we do not need in this shader. They are included because the struct is used for multiple shaders within Unity's framework. Leaving the extra variables here does not significantly contribute to the processing at runtime.

If we save the shader at this point, we will get some conflicts. The shader is still pointing to the built-in `Lambert` lighting model and we have a `SurfaceOutput` struct with the same name as the struct in this file.

The solution is to give the lighting model a unique name.

9. Locate the shader compile directive:

```
#pragma surface surf Lambert
```

10. Replace it with the following line:

```
#pragma surface surf MyLambert
```

    We will still get compile errors at this point, as we are calling a lighting model that has not yet been defined.

11. Next, locate the `SurfaceOutput` struct and rename it `SurfaceOutputMyLambert`.

This will force the shader to use the internal surface output variables, rather than the built-in set, when it compiles.

12. After this struct, add the following code:

```
            inline fixed4 LightingMyLambert (SurfaceOutputMyLambert s,
fixed3 lightDir, half3 viewDir,
              fixed atten)
            {
                fixed diff = max (0, dot (s.Normal, lightDir));

                fixed4 c;
                c.rgb = s.Albedo * _LightColor0.rgb * diff;
                c.a = s.Alpha;
                return c;
            }
```

Here, we are defining a new lighting model with a unique name, `LightingMyLambert`.

The lighting model uses the output variables from the struct, as well as lighting, view direction, and attenuation variables that we get from Unity.

The `diff` variable is a low-precision float derived from the surface normal, multiplied by the light direction. The `max` operator takes the highest number between this dot product and the other provided value, 0.

The variable defines the surface color. This is produced by multiplying the albedo by the light color and our `diff` variable.

The `_LightColor0` variable is obtained from Unity. This is the first light in the light array, it is usually a directional light.

13. Scroll down to the `surf` function and amend the opening line to use the new output variables list:

```
            void surf (Input IN, inout SurfaceOutputMyLambert o) {
```

14. Save the shader and return to the main Unity interface.

Unless anything has been missed, the shader should compile as before. This time, it is not accessing the external `CGIncludes` files.

In the next section, we will create a more complex lighting model for our scene.

Adding specularity to our custom lighting model

The Lambert lightingmodel does not allow us to add specular highlights to our surfaces. Adding these highlights requires view directional data. We need to know where the surface is being viewed from, rather than just where the light is.

For this, we can use Unity's built-in view direction variable, `viewDir`.

Let's start by making a copy of the shader so that we can compare them:

1. In the **Assets** panel, select the `TestLambert` shader and duplicate it using the *Ctrl* + *D* hotkey combination (use *command* + *D* if you are working on a Mac).
2. Rename the new shader `TestSpec`.

   Our first step is to add the appropriate properties that will allow us to demonstrate the specular.

3. Double-click on the `TestSpec` shader to open it in MonoDevelop.
4. Add the following code to the `Properties` section:

   ```
   _Glossiness ("Smoothness", Range(0,1)) = 0.5
   _Specular ("Specular", Range(0,1)) = 0.0
   ```

   We will add `Range` properties for the `_Glossiness` and `_Specular` components to allow us to control the highlight size and strength with sliders within the **Inspector**.

5. Locate the shader compile directive (which is displayed in magenta in MonoDevelop) and replace it with the following:

   ```
   #pragma surface surf MySpecular
   ```

6. Add the following code at the bottom of the variables list, further down in the shader code:

   ```
   half _Glossiness;
   half _Specular;
   ```

7. Rename the `SurfaceOutputMyLambert` struct to point to the `MySpecular` data structure by rewriting the opening line as follows:

   ```
   struct SurfaceOutputMySpecular {
   ```

8. Locate the opening line of the lighting model and rewrite it to point to the renamed surface output struct, as follows:

```
            inline fixed4 LightingMySpecular (SurfaceOutputMySpecular,
fixed3 lightDir, half3 viewDir,
              fixed atten)
            {
```

   You will see that this line already contains a view direction variable called `viewDir`.

   In the next step, we will include it in the lighting calculation.

9. Replace the remaining part of the light model with the following code:

```
            half3 h = normalize (lightDir + viewDir);
            fixed diff = max (0, dot (s.Normal, lightDir));
            float nh = max (0, dot (s.Normal, h));
            float spec = pow (nh, s.Specular*128.0) * s.Gloss;

            fixed4 c;
            c.rgb = s.Albedo * _LightColor0.rgb * diff + _LightColor0.rgb *
spec;
            c.a = s.Alpha;

            return c;
            }
```

   Here, we will set up a few local variables to allow us to do some more complex calculations.

   The `h` variable is a combination of the light direction and view direction, clamped with the normalize function.

   The `difffixed` float takes the maximum number between 0 and the dot product of the surface normal and light direction.

   The `nh` float is equal to the maximum value of 0 and the dot product of the surface normal and `h`.

   The `spec` float returns the surface specular value increase to the value of `nh` multiplied by the surface gloss value.

   These values are assembled in the surface albedo, multiplied by the light color.

10. In the `surf` function, at the bottom of the shader, add the two necessary output definitions:

```
o.Gloss = _Glossiness;
o.Specular = _Specular;
```

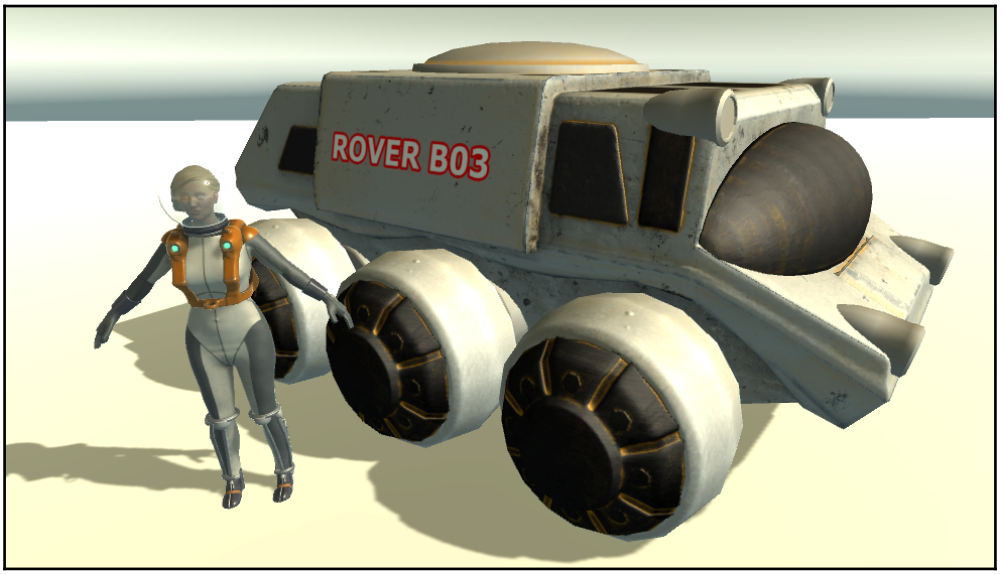
11. Save the shader.

    Let's apply the new specular shader to the rover vehicle to compare it to the Lambert lighting.

12. Select the `rover` game object by clicking on it in the **Hierarchy** panel.
13. When its components appear in the **Inspector**, scroll down to the **Mesh Renderer** component.
14. In the `Rover` material parameters section, click on the **Shader** button and choose **PACKT | TestSpec** from the drop-down list.

You will notice a slight change in the rover's appearance in the **Scene** view.

In the **Inspector**, the **Smoothness** slider will define the overall shininess of the surface and the **Specular** slider will define the size of the highlighted area on the surface:



The rover model with the custom specular lighting model applied

To define a varied surface with shiny and dull areas, we will need to incorporate texture maps. We will be covering this in detail in the next chapter when we explore more complex organic lighting effects.

Summary

In this chapter, we explored the differences between metallic and specular workflows. With the metallic shader workflow, we created a custom decal material for the scene's crate asset. We then demonstrated how the decal could be varied among different crate game objects in the scene at runtime with the help of a little coding.

With the specular workflow, we took a look at Unity's built-in lighting models and incorporated the necessary code in our own custom shader to utilize specular lighting effects.

In the next chapter, we will take a closer look at shader creation for organic surfaces, such as hair and skin.

7
Shaders for Organic Surfaces

In this chapter, we will build on concepts introduced in the previous chapters to create more advanced organic surfaces for the astronaut's character.

We will cover the following topics in this chapter:

- Creating a skin shader with a subsurface scattering effect
- Layering specular materials and manipulating UV coordinates to create a realistic and responsive eyeball setup
- Creating a custom anisotropic shader to create a hair shader

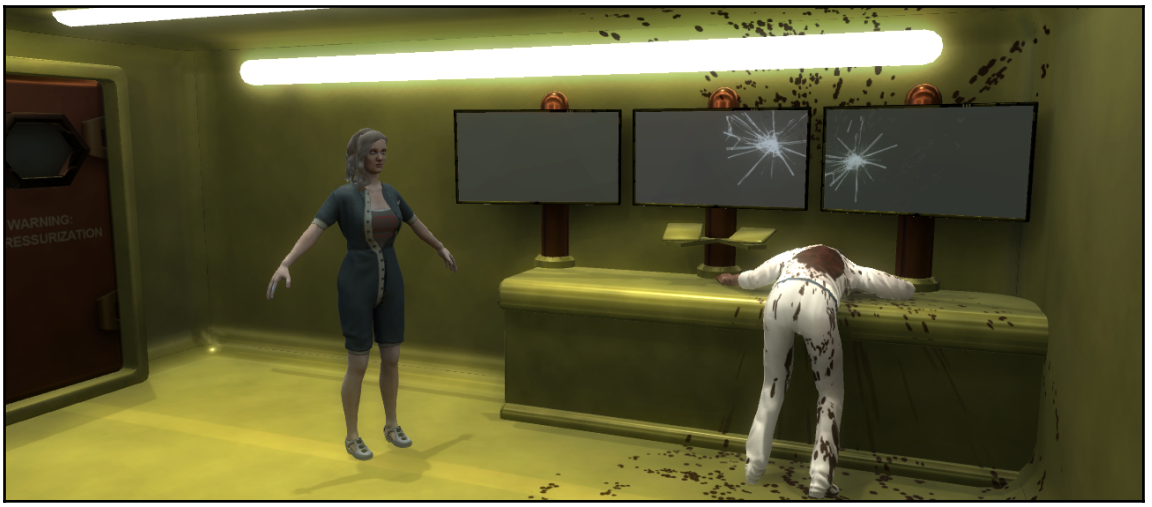
We will start by taking a look at the example scene for this chapter.

Start the scene

As we did earlier, the content needed for the chapter is located in the project files.

Load the `Chapter7_Start` scene using the menu bar by navigating to **File** | **Open Scene** | **Chapter7_Start**.

The scene depicts the interior of the research station:



Initial scene state—character with Standard Shader materials

The scene uses real-time lighting. It has a single directional light pointing downward and self-illuminating tubes that also light the interior.

At the moment, the astronaut's character uses default Standard Shader materials. In the next step, we will begin writing a custom shader to improve the realism of the organic surfaces.

First, let's talk a little about what we are trying to achieve.

Understanding the complexities of human skin

Our astronaut is currently set up to work with the standard shader and it does a pretty good job of defining most surfaces. For added realism, we will create a new shader that more closely approximates the physical properties of human skin.

Human skin is partially translucent, rather than reflecting light in the same way as a typical opaque material. Some light is scattered around in the underlying, or epidermal, layers. This often means that skin appears lighter in all but extremely low-light conditions.

In game graphics, we do not usually have the resources to recreate the underlying layers of skin in human characters, we have to try and accomplish the effect by varying the surface's response to light in our shader output.

In this case, we will set up a standard specular material and add a subtle emissive quality to the surface.

Creating the skin shader

Moving beyond the purely surface qualities of the model, we need to use a representation of the relative thickness of the model to properly define the subsurface scattering effect. We can calculate the thickness within a shader and then use it to define where the effect is visible.

We will start by creating a new shader:

1. In the **Project** panel, click on the `PACKT_Shaders` folder to view its contents in the **Assets** panel.
2. In an empty part of the **Assets** panel, create a new shader like we have done before. Right-click and choose **Create** | **Shader** | **Standard Surface Shader** from the drop-down list.
3. In the **Assets** panel, rename the shader `SSS_Skin`.
4. Double-click on the shader to open it in MonoDevelop.
5. In MonoDevelop, rename the shader as follows:

```
Shader "PACKT/SSS_Skin"
```

   Next, we will make some changes to the `Properties`.

6. In the `Properties` block, delete the `_MainTex`, `_Glossiness`, and `_Metallic` lines.

   We need to add a bump map property to correctly show the model's detail.

7. Add the following line:

```
_BumpMap ("Normal (Normal)", 2D) = "bump" {}
```

   We will also add a scale slider to control the range of the effect.

8. Add the following line:

```
_Scale ("Thickness Scale", Range(0,1)) = 0.0
```

   Within the Cg snippet, there is a variable for the `_MainTex` property that no longer exists, we will rename it to work with the bump map.

9. Locate the following line:

```
sampler2D _MainTex;
```

10. Rename it as follows:

```
sampler2D _BumpMap;
```

11. In the `Input` struct, delete the `uv_MainTex` variable and add the following lines:

```
float2 uv_BumpMap;
float3 worldPos; //added
float3 worldNormal;
INTERNAL_DATA
```

    The `uv_BumpMap` variable stores the UV information of the normal map. We get the `worldPos` and `worldNormal` values from Unity, defining the position and normal based on global data. These are marked as `INTERNAL_DATA`.

12. Delete the `_Glossiness` and `_Metallic` variables and add the following line:

```
half _Scale;
```

    This will allow us to use our scale slider.

    In the `surf` function, we need to clean up a few lines of code that we do not need any more.

13. Select and delete the code within the `surf` function near the bottom of the shader code.

    Next, we need to unpack our normal map as this will be used for the thickness.

14. Add the following line:

```
float3 n = UnpackNormal(tex2D(_BumpMap, IN.uv_BumpMap));
```

We will create a new variable for the normal so that it can be used in another calculation. We will specify the property name and the UV coordinates that we have previously added.

15. Add the next line of code:

```
float thickness = length (fwidth (WorldNormalVector (IN,
  n)))/length (fwidth (IN.worldPos)) * _Scale;
```

Here, we will create the variable thickness and calculate it using the `WorldNormal` and `WorldPos` variables that we get from Unity. We will multiply the result using our `_Scale` value as the factor.

We need to add a line of code to specify the output albedo.

16. Add the following line:

```
o.Albedo = _Color.rgb * thickness;
```

We will multiply the `Color` RGB values by our new thickness value. This will allow us to tint the thinnest parts of the model to see the effect more clearly.

17. Finally, add the output alpha line:

```
o.Alpha = _Color.a;
```

We set the alpha to be equal to the alpha channel value in the color used in the shader.

18. Save the shader and return to the main Unity interface.

Now that we have written the shader, let's use this in the astronaut's material and view it in our scene.

19. In the `PACKT_Materials` folder, locate and select the `Astronaut_crewsuit` material.
20. At the top of the **Inspector**, set the material's shader to **PACKT** | **SSS_Skin** from the drop-down list.

The character's appearance will change in the **Scene** and **Game** views.

21. Reduce the **Thickness Scale** value to `0.01`.

The calculated thickness visualized on the character model

Thinner parts of the mesh will be lighter and tinted by the color that you choose in the material. It is the thin areas of the skin where we want to introduce the subsurface scattering to simulate the diffusion on light beneath the surface.

In the next step, we will add some texture inputs to the shader and make the subsurface effect dependent on lighting.

Adding complexity to the skin shader

We have got a representation of the model's thickness, but we need to add the relevant input maps and the lighting calculation that will get the skin looking right.

Let's add the properties that we need first:

1. Back in MonoDevelop, add the following lines to the shader's `Properties` block:

```
_MainTex ("Albedo (RGB)", 2D) = "white" {}
_Metallic ("Metal (R), Smoothness (A)", 2D) = "white" {}
```

   We stripped these properties out of the shader previously, but we need the character's base albedo and metallic/smoothness inputs to approximate the surface color and level of shininess.

2. Next, add the following lines:

```
_SSS_Color ("Subsurface Color", Color) = (1.0, 1.0, 1.0, 1.0)
_Diffusion ("Subsurface Diffusion", Float) = 0.0
```

   We will add an additional color property, `_SSS_Color`, for the subsurface color, freeing `_Color` to be used to tint the whole material as usual.

   The diffusion property will be used to determine the depth of the light-scattering effect.

3. Within the Cg snippet, locate the `sampler2d _BumpMap` line and add the following code under it:

```
sampler2D _MainTex;
sampler2D _Metallic;
```

   Here, we set up the local variables to handle the texture input.

   When we added the bump map, we amended the UV coordinate variable in the `Input` struct to use the coordinates tied to `_BumpMap`. We will amend this next.

4. Locate the `float2 uv_BumpMap` line in the `Input` struct.
5. Replace it with the following:

```
float2 uv_MainTex;
```

   All our input maps will share the same UV coordinates.

6. After the closing bracket and semicolon of the `Input` struct, add the remaining variables:

```
fixed4 _SSS_Color;
float _Diffusion;
```

   We will use a `fixed4` type variable for the `_SSS_Color` property to handle all four channels of the color. The `_Diffusion` variable is set up as a float.

7. In the `surf` function, add the following lines:

```
fixed4 c = tex2D (_MainTex, IN.uv_MainTex);
fixed4 m = tex2D (_Metallic, IN.uv_MainTex);
```

   Both texture inputs use the same `_MainTex` UV coordinates.

   We need to use the same UV coordinates in the bump map.

8. In the `float3 n` line, replace the `uv_BumpMap` reference with `uv_MainTex`.
9. Adjust the `o.Albedo` line to include the main texture and subsurface color `rgb` values by replacing it with the following:

```
o.Albedo = c.rgb * _Color.rgb;
```

   We add the product of `SSS_Color` and `thickness` to the main texture `rgb`, essentially masking the color with the thickness.

10. Add the output lines for `Metallic` and `Smoothness` to use the relevant channels of the `_Metallic` texture map:

```
o.Metallic = m.r;
o.Smoothness = m.a;
Save the shader.
```

11. Return to the main Unity interface.

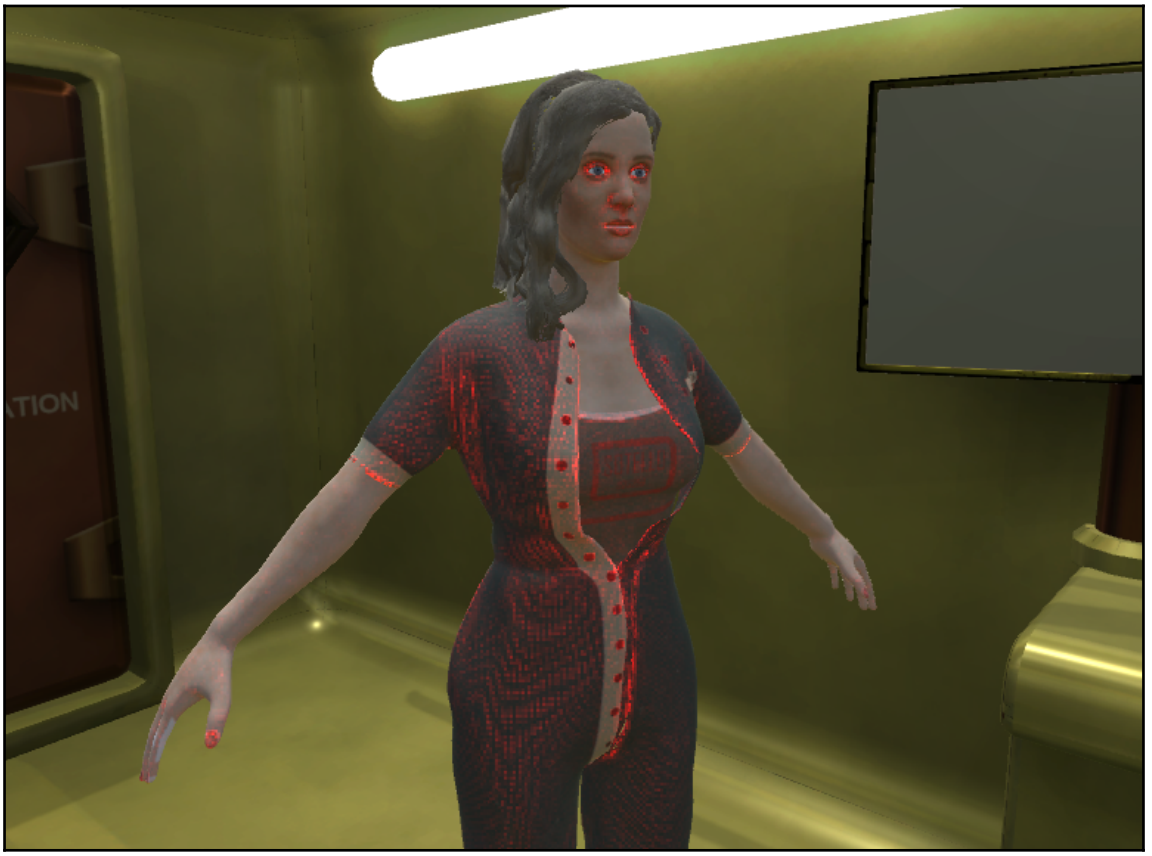
    When the shader has compiled, there should be two additional texture slots in the `SSS_Skin` material in the **Inspector**.

    The appropriate textures are located in the `astronaut_crewsuit` subfolder within the `PACKT_Textures` project folder.

12. Drag the `crewsuit_albedo` texture to the material's **Albedo** slot.
13. Next, drag the `crewsuit_metallic` texture to the material's **Metallic** slot.

14. Change the **Color** swatch at the top of the material back to white.
15. Set the **SSS_Color** swatch to a bright red color.

The results should appear as shown in the **Scene** view:



The calculated thickness added to the albedo with the subsurface color

At the moment, we have a nice glow effect on the thinner parts of the character, but for a better subsurface effect, we will integrate a custom lighting model.

Writing the custom lighting model

At this stage, we want to add better interaction with the scene lighting. To create the appearance of translucency, we want thinner areas of the character model to show more bounced light if they have light behind them.

This will be handled in a custom light model:

1. Back in MonoDevelop, add the following code after the `surf` function's closing bracket:

```
inline fixed4 LightingSkinSSS (SurfaceOutput s, fixed3 lightDir,
  fixed3 viewDir, fixed atten)
{

}
```

   Within the lighting model, we will grab the light direction, view direction, and attenuation ready to be used in our calculations.

   We need to further define the light direction and how this interacts with the surface normal.

2. Add the following code after the opening bracket:

```
half3 diffLightDir = lightDir + s.Normal * _Diffusion;
float transDot = max(0, dot(viewDir, -diffLightDir)) * _Scale;
fixed3 transLight = transDot * _SSS_Color.rgb * atten;
fixed3 transAlbedo = transLight * s.Albedo * _LightColor0.rgb;
```

Here, we will create a variable called `diffLightDir`, which is a combination of `lightDir` and the surface normal, multiplied by the diffusion.

The next variable, `transDot`, returns a float value that is the result of the highest value of `viewDir` multiplied by the inversed `diffLightDir` and zero. The result is multiplied by our `_Scale` value.

Following this, we will create a `fixed3` variable, `transLight`, which is the product of `transDot`, the defined subsurface color and the light's attenuation.

Lastly, we will define `transAlbedo` as the product of `transLight`, the surface albedo and the light color.

The next calculations will define the light's effect on the surface:

1. Add the following code:

```
half3 h = normalize (lightDir + viewDir);
fixed diff = max(0, dot (s.Normal, lightDir));
float nh = max(0, dot(s.Normal, h));
float spec = pow (nh, s.Specular * 128.0) * s.Gloss;
fixed3 diffAlbedo = (s.Albedo * (_LightColor0.rgb * diff) +
    (_LightColor0.rgb * spec)) * atten;
```

   Here, we have added standard Unity **Blinn-Phong** lighting to take care of the areas of the model where we do not see the subsurface effect.

   Finally, we will process the albedo to include the diffused and translucent light additives:

2. Add the following code:

```
fixed4 c;
c.rgb = diffAlbedo + transAlbedo;
c.a = _LightColor0.a * atten;
return c;
```

   At this point, our shader may compile correctly, but we still need to make it use the new lighting model and adjust a few incompatible output types.

3. Near the top of the Cg snippet, locate the shader compile directive:

```
#pragma surface surf Standard fullforwardshadows
```

4. Make it to point to our new lighting model by amending it as follows:

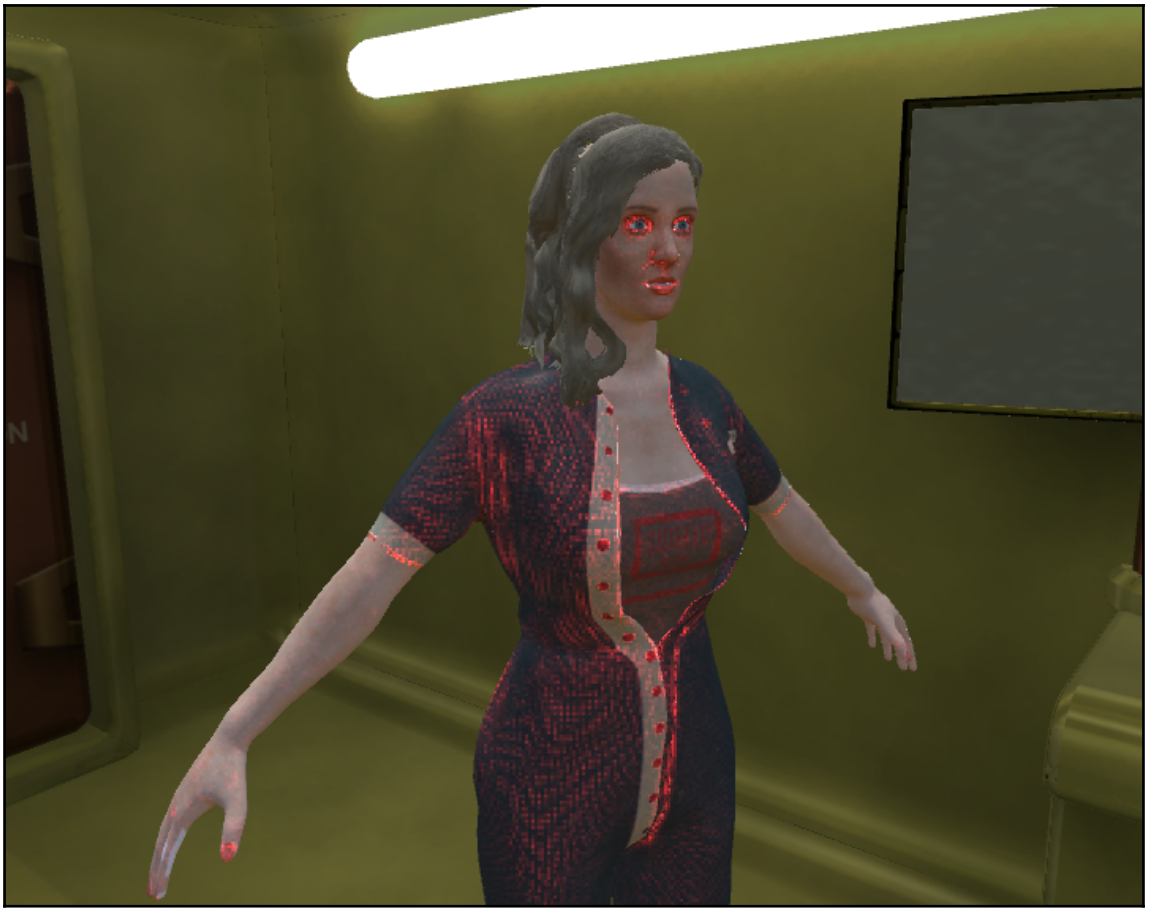
```
#pragma surface surf SkinSSS fullforwardshadows
```

5. Scroll down to the `surf` function and replace the reference to `SurfaceOutputStandard` with `SurfaceOutput`.
6. Remove the `Metallic` output line to comply with the default surface output list.
7. Rename the `o.Smoothness` line to the following:

```
o.Gloss = m.a * m.r;
```

8. Save the shader.

9. Return to the main Unity interface to see the result in the **Scene** and **Game** views.



The result of the custom lighting model

The result is more subtle, with the diffusion working correctly, the light will appear to be more scattered rather than appearing as a surface glow.

At the moment, the subsurface effect appears across the whole model as the astronaut's body has a single material.

We can limit this to just the skin areas by using a texture map instead of calculating the thickness in the shader. We already have a thickness map ready to go. We will add this next.

Adding the thickness map input to the shader

Using a texture map to define the thickness of the character for subsurface scattering makes it possible to apply the effect selectively to the model.

Let's start by taking a look at the map:

1. Select the character's mesh by clicking on `PACKT_astronaut_crewSuit` in the **Hierarchy** panel.

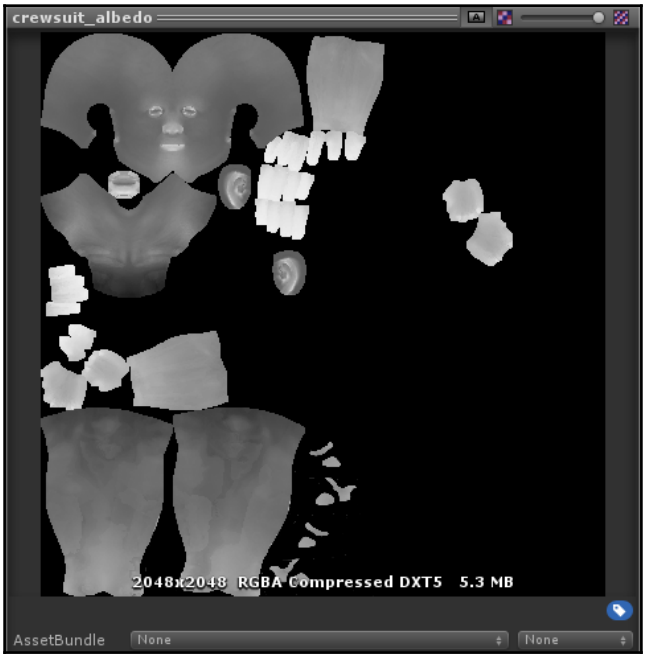
   Its components will appear in the **Inspector**.

2. In the material components at the bottom, click on the character's albedo map.

   The texture file will become highlighted in the **Assets** panel.

3. Click the source texture, `crewsuit_albedo`, to see its import settings.

4. Within the material preview, click on the small RGB icon to see the texture's alpha map:



The crewsuit_albedo texture's alpha channel

This thickness map has been generated in an external application. The thickness is only defined for the skin parts of the character, the other areas are left black and will not be affected.

We need to make a few changes to the shader code to use this map.

5. Return to MonoDevelop.
6. At the top of the shader, change the `_MainTex` property to read as follows:

```
_MainTex ("Albedo (RGB), Thickness (A)", 2D) = "white" {}
```

7. In the `Input` struct, delete the following lines of code:

```
float3 worldPos;
float3 worldNormal;
INTERNAL_DATA
```

8. In the lighting model, locate the `transLight` line and replace it with the following:

```
fixed3 transLight = transDot * s.Alpha * _SSS_Color.rgb * atten;
```

This will include the alpha channel in the translucency calculation.

9. Save the shader.

10. Return to the main Unity interface to see the results:

The completed skin shader

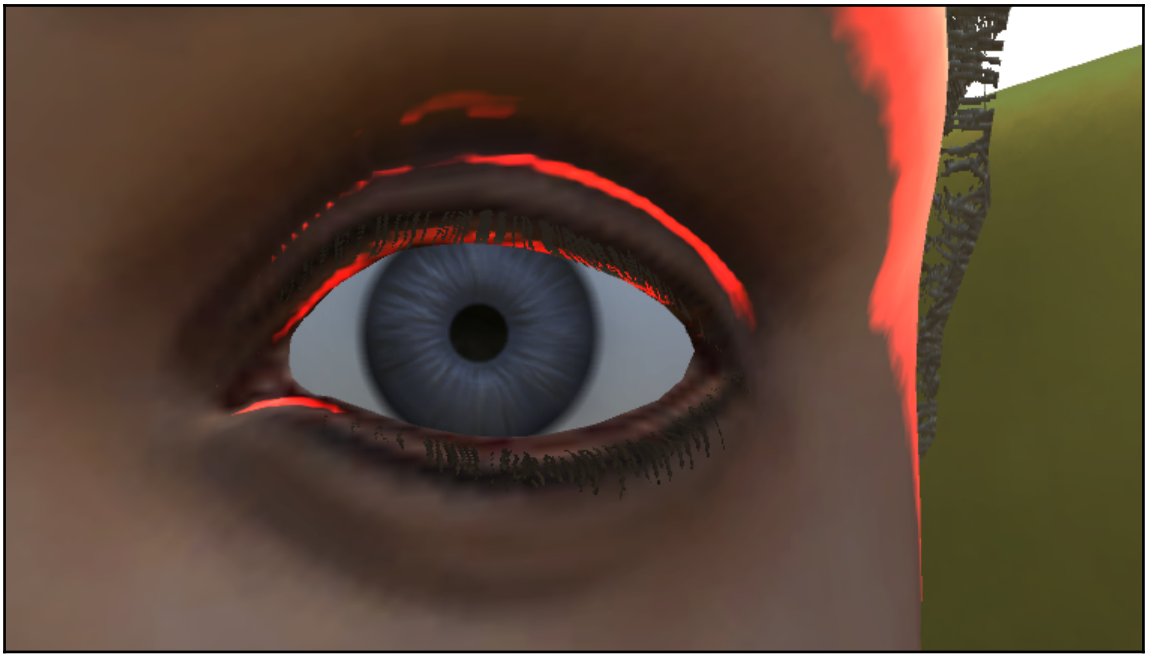
In the next section, we will create a custom eye shader.

Creating the eye material

The white part of the eye, the sclera, absorbs light and is reflective, so we need to treat it differently from the rest of the body:

1. In the **Hierarchy** panel, locate the `astronaut_eye_L` model.

2. Double-click on its name in the list to zoom in on the model in the **Scene** view:



The eye set up with a Standard Shader and initial textures

The surface looks quite dull with the default material settings.

3. In the **Inspector**, scroll down to the **Material** properties and set the shader type to `Standard Specular`.
4. Set the **Specular** color to a dark, slightly warm gray. The hex code `383434FF` will work pretty good.
5. Set the **Smoothness** value to `0.85`.

The eye should appear more shiny.

As the eyeball is a separate shape that does not deform, we can partially solve these problems with geometry by adding an additional shell to render the reflection.

In this case, we will use a transparent shiny material to render the sclera.

6. In the **Hierarchy** panel, click on the small arrow next to the `astronaut_eyeL` game model to expand its hierarchy.

   The `scleraL` object is a child of `astronaut_eyeL` and is currently deactivated.

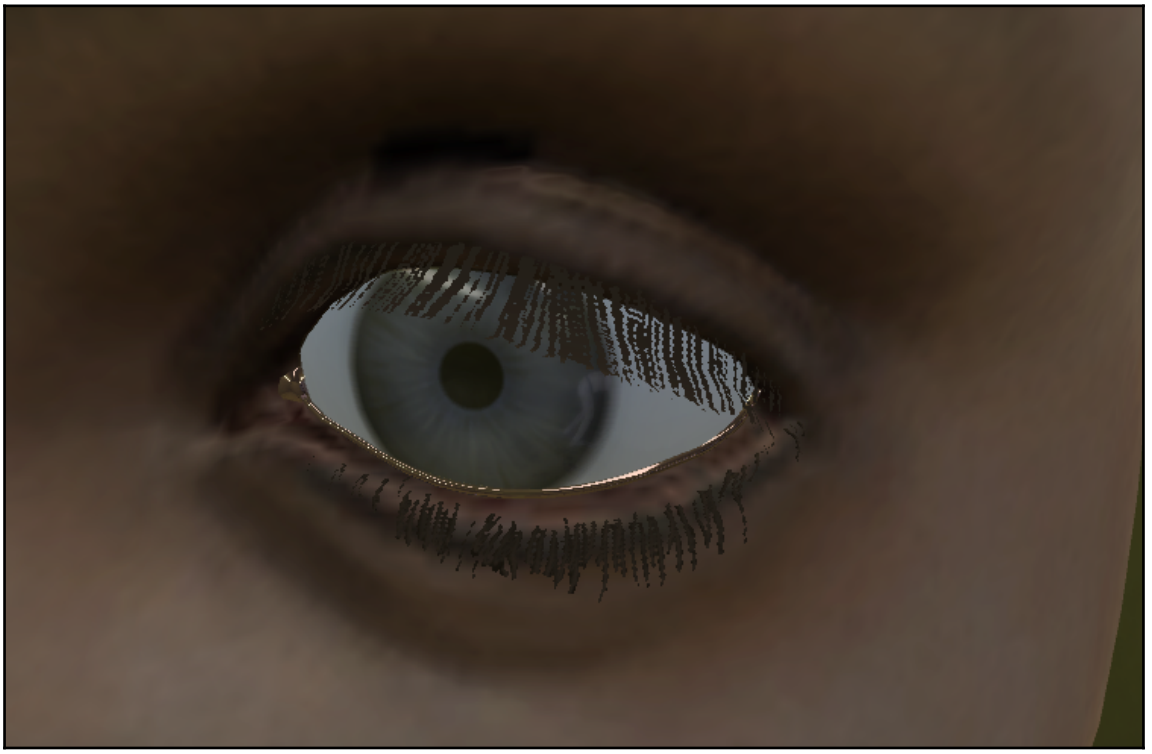
7. Click to select `scleraL` in the **Hierarchy** panel and activate it by clicking on the checkbox next to its name in the **Inspector**.

The activated sclera game object

The sclera's material has already been set up and needs to be assigned.

8. In the **Project** panel, click on the `PACKT_Materials` folder to view its contents in the **Assets** panel.

9. Locate the `astronaut_sclera` material and drag it to the `scleraL` game object in the **Hierarchy** panel.



The sclera with its material applied

The material gives the eye a shinier appearance. The geometry that covers the iris is slightly curved outward, creating a more realistic reflection.

The size of the pupil is fixed in the texture. We may want the pupil to change its size to adapt to lighter or darker conditions within our game.

Let's create a custom shader to get this working.

Creating the custom eye shader

Creating a custom shader for the eye will allow us to modify the scale of the pupil at runtime:

1. In the **Project** panel, click on the `PACKT_Shaders` folder to view its contents in the **Assets** panel.
2. Right-click on an empty space within the **Assets** panel and create a new shader by selecting **Create** | **Shader** | **Standard Surface Shader** from the drop-down list.
3. Rename the new shader `eye`.
4. Double-click on it to open it in MonoDevelop.
5. In MonoDevelop, rename the shader as follows:

```
Shader "PACKT/eye"
```

   Next, we will change the properties.

6. In the `Properties` block, delete the `_Metallic` line and add the following:

```
_Specular ("Specular", Range(0,1)) = 0.0
_BumpMap ("Normal Map", 2D) = "bump" {}
_PupilScale("Pupil Size", Range(0, 100)) = 1.0//adjust ranges
_UVRange("UV Range", Range(50, 100)) = 50
```

   Here, we will add properties for specular and normal map inputs. We will add range properties to allow us to control the scale of the pupil and range of UVs that are being affected.

7. In the shader compile directive, add the `Specular` keyword to `Standard` to allow the shader to use the specular workflow. The adjusted directive should read as follows:

```
#pragma surface surf StandardSpecular fullforwardshadows
```

8. Locate the `sampler2d _MainTex` line and add the following lines of code just beneath it:

```
sampler2D _BumpMap;
float _PupilScale;
float _UVRange;
```

We will add `sampler2d` so that the shader will interpret the normal texture map, and the `_PupilScale` and `_UVRange` floats so that we can use the sliders in our calculations.

9. Within the `surf` function, delete the `o.Metallic` line and replace it with the following:

```
o.Specular = _SpecColor.rgb;
```

Rather than using a texture map here, we will use the values provided by the defined color in our properties.

Now that we have adjusted the specular components, let's get the pupil scaling working correctly.

10. At the top of the `surf` function, add the following lines:

```
float2 pos = IN.uv_MainTex.xy-0.5;
float dist = 1-length(pos);
dist = pow(dist, _UVRange);
float2 disp = pos * clamp(dist * _PupilScale, 0.0, 1);
pos -= disp;
```

Here, we will create a new `float2` variable, `pos`, to handle the adjusted UV position.

The next `float` variable, `dist`, is defined as `1- length(pos)`.

We then modify dist using the `pow` function to bring `dist` up to the power of the `_UVRange` variable.

In the next line, we will add a new `float2` variable named `disp` and make it equal to `pos`, multiplied using the clamp function, which we define as `dist` multiplied by the `_PupilScale` variable, using the ranges `0.0` and `1`.

Lastly, we will make `pos` negatively affected by `disp`:

1. Locate the `fixed4 c` line and replace it with the following:

```
fixed4 c = tex2D (_MainTex, pos + 0.5);
```

Here we will use our modified pos UV variable in the two-dimensional texture definition. We will add `0.5` to both values to center the coordinates.

As we are also using a bump map in this shader, we need to modify the UV interpretation the same way.

2. Add the following line under the `o.Specular` line:

```
o.Normal = UnpackNormal ( tex2D (_BumpMap, pos + 0.5));
```

3. Save the shader.
4. Return to the main Unity interface.

5. Locate the `astronaut_eye_norm` texture in the `PACKT_Textures` folder and drag it to the **Normal Map** texture slot in the **Inspector**:

The final eye material

With the new shader, we can modify the size of the pupil by dragging the `Pupil Size` value. The current value of the **UV Range** constricts the effect and is clamped between `50` and `100`.

In the next section, we will create a custom hair material.

Creating the hair material

In most games, hair is rendered using opacity-mapped planes rather than rendering individual strands. This efficiently gives the impression of multiple strands without the performance cost of the thousands of triangles that would have to be used for each strand.

In the scene, take a look at how the astronaut's hair currently appears:

The astronaut's hair with a Standard Shader material

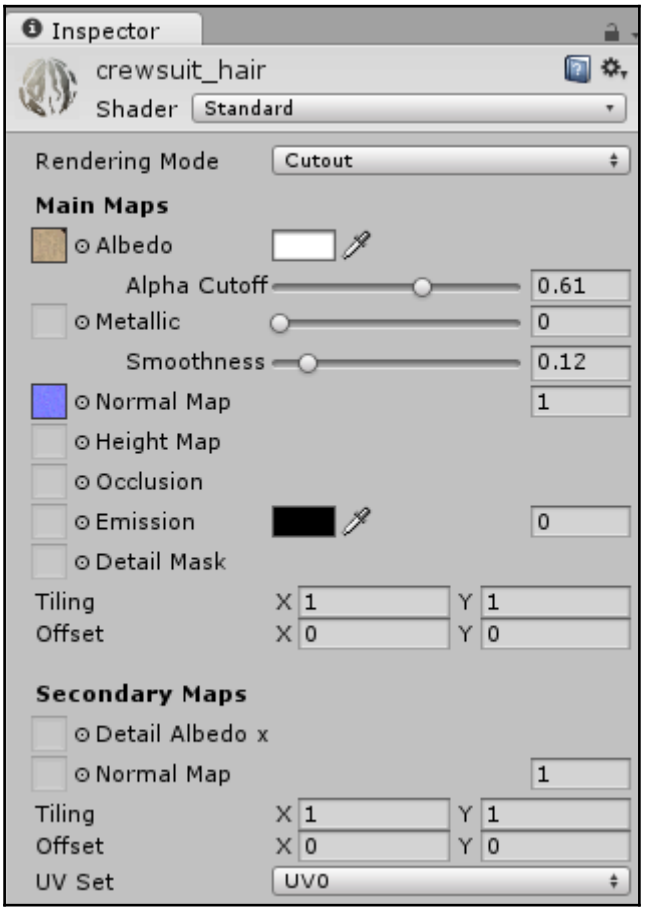
Right now, each lock of hair only renders in one direction.

We can improve this by rendering the back faces of the hair mesh and using a custom lighting model to produce more realistic lighting.

This time, we will use anisotropic lighting. This method emphasizes surface normals of objects viewed at glancing angles and is ideal for long hair.

Let's start by taking a look at the current material's input maps:

1. In the **Hierarchy** panel, select the `astronaut_crewsuit` game object.
2. If necessary, expand the game object's hierarchy to view its child game objects.
3. Click on the `hair` game object to select it.
4. Scroll to the bottom of the **Inspector** and click on the `crewsuit_hair` material to expand it and view its inputs:



The crewsuit_hair material set up with a Standard Shader

The hair material currently uses albedo and normal map inputs with a value applied for smoothness.

We will create the new custom shader next.

Creating the custom hair shader

We have established the texture maps that will be used as inputs and can start building our new shader:

1. In the **Project** panel, click on the `PACKT_Shaders` folder to view its contents in the **Assets** panel.
2. Right-click on an empty area of the **Assets** panel and navigate to **Create | Shader | Standard Surface Shader** from the drop-down selection list.
3. Rename the new shader `Hair`.
4. Double-click on it to open it in MonoDevelop.

    In MonoDevelop, we will start by setting the shader name and location.

5. Change the first line of the shader to the following:

```
Shader "PACKT/hair" {
```

6. Delete the `_Metallic` line from the `Properties` block.
7. Replace it with the following lines:

```
_SpecColor ("Specular Color", Color) = (1,1,1,1)
_Direction ("Anisotropic Direction)", 2D) = "bump" {}
_Anisotropic ("Anisotropy", Range(-1,1)) = -0.0
```

    We will add the `_SpecColor` property to control the color of the highlight. `_Direction` will use a normal map to define the highlighted details in the hair. The last property, `_Anisotropic`, will define the extent of the effect.

8. Further down in the shader, locate the **Level Of Detail** (**LOD**) line and add the following code under it:

```
Cull Off
```

    This will ensure that the back faces of the hair mesh are rendered.

9. Scroll down to the Cg snippet and replace the Shader compile directive with the following:

```
#pragma surface surf HairAnisotropic alpha
```

   We will be defining the custom lighting model further down in the shader.

10. Locate the following line:

```
sampler2D _MainTex;
```

   We need to add similar variables to handle the anisotropic direction map.

11. Add the following line directly after this:

```
sampler2D _Direction;
```

   Next, we need to add the variables to handle the extent of the anisotropic effect.

12. Add the following code:

```
float _Anisotropic;
```

   Next, we need to add the variables needed by our custom lighting function.

13. Add the following code:

```
struct SurfaceHairAnisotropicOutput
{
    fixed3 Albedo;
    fixed3 Normal;
    fixed3 Emission;
    fixed3 Direction;
    half Specular;
    fixed Gloss;
    fixed Alpha;
};
```

   The next step is to add the lighting function.

14. Add the following code:

```
inline fixed4 LightingHairAnisotropic (SurfaceHairAnisotropicOutput
  s, fixed3 lightDir, half3 viewDir, fixed atten)
{
    fixed3 lightView = normalize(lightDir + viewDir);
    float NdotL = saturate(dot(s.Normal, lightDir));
    fixed NdotD = dot(normalize(s.Normal + s.Direction),
      lightView);
    float anisotropy = max(0, sin(radians((NdotD * _Anisotropic) *
      180)));
    float spec = saturate(pow(anisotropy, s.Gloss * 128) *
      s.Specular);
    fixed4 c;
    c.rgb = ((s.Albedo * _LightColor0.rgb * NdotL) +
      (_LightColor0.rgb * _SpecColor.rgb * spec)) * atten;
    c.a = s.Alpha;
    return c;
}
```

Here, we will define the shader lighting, inputting the light direction, view direction, and attenuation from Unity.

We will create a `fixed3` float named `lightView`, which adds the light direction and view direction together.

In the next line, we will create a `float` named `NdotL` based on the product of the surface normal and the light direction, clamped to the 0, `1` range using the `saturate` function.

Next, we will create a float called `NdotD` equal to the product of the surface normal and surface direction—provided by the normal map input.

The next line creates an `anisotropy` float. This is defined as the maximum of the two specified values, zero and the factor of `NdotD` and `_Anisotropic`, multiplied by `180`, and processed with the `sin radians` method to translate the light angle degree into a number value that we can work with.

The next value defines the specular as the saturated factor of the `anisotropy` and surface gloss multiplied by `128`. The product of this is further multiplied by the surface specular.

We preprocess the albedo output by defining `c.rgb` as the result of the surface albedo multiplied by the light color, multiplied by `NdotL`. This value is added to the result of the light color multiplied by the specular color and processed specular level and the light's attenuation.

We will define the fourth channel of `c` as being equal to our surface alpha, preserving the main texture's transparency.

1. Scroll down to the `surf` function and replace its opening line with the following:

```
void surf (Input IN, inout SurfaceHairAnisotropicOutput o) {
```

   We will replace the standard output set with the custom variables that we defined in the `SurfaceHairAnisotropicOutput` struct.

2. Locate the fixed4 c line and replace it with the following:

```
fixed4 c = tex2D (_MainTex, IN.uv_MainTex);
```

   We remove the reference to the `_Color` property to prevent it from affecting the transparency that is defined in the alpha channel.

3. Amend the `o.Albedo` line as follows:

```
o.Albedo = c.rgb * _Color.rgb;
```

4. Add the following line of code:

```
o.Direction = UnpackNormal(tex2D(_Direction, IN.uv_MainTex));
```

   Here, we will unpack the normal, storing it in the `o.Direction` slot. It shares UV coordinates with the main texture.

5. Scroll down to locate the `o.Metallic` and `o.Smoothness` lines and delete them.
6. Add the following code:

```
o.Specular = _SpecColor.rgb;
o.Gloss = _Glossiness;
```

7. Save the shader and return to the main Unity interface.
8. In the **Inspector**, click on the Shader selection drop-down menu and choose the new shader by navigating to **PACKT** | **hair**.
9. In the **Project** panel, click on the `PACKT_Textures` folder to view its contents in the **Assets** panel.
10. Locate the `hair_normal` texture asset and drag it to the **Anisotropic Direction** texture slot in the hair material in the **Inspector**.

In the **Scene** and **Game** views, the resulting hair surface should have more pronounced highlights. The extent of the anisotropic effect can be adjusted by manipulating the **Anisotropy** slider in the **Inspector**.

The hair model with the finished shader

The resulting hair surface has a wider spread of highlight to simulate the light bouncing off individual strands rather than a solid mass of geometry.

Summary

In this chapter, we added more realism and emphasis to the organic surfaces in our scene.

For the character's skin, we added specular and gloss values and used calculated and texture map thickness to simulate a subsurface scattering effect.

In the eye texture, we varied specular and gloss to get better highlights for the outer eye area. We created a custom shader to manipulated the pupil scale.

For the hair, we created a custom anisotropic lighting effect to emphasize the strands of the hair using a normal map.

In the next chapter, we will look at how shaders can be created for eye catching particle effects in our game.

8

Custom Particle Shaders – Smoke, Steam, and Fluids

In this chapter, we will take a closer look at Unity's particle system and how it can be used to set up a number of effects.

At its basic level, a particle system is made up of billboard textures emitted in a specific configuration at runtime.

Unity, like most game engines, allows us to adjust a number of parameters to make the particles appear like water, snow, fire, and a lot of other different effects.

We will be integrating particles and other effects into the provided example scene.

We will be looking at the following topics in this chapter:

- Particle system creation
- Particle-specific shaders
- Steam and smoke particle effects with animated sprite sheets
- Custom fire shader creation
- Fluid effects without particle systems

Unlike many other shaders that we have built at this point, particle shaders are not expected to interact with scene lighting in the same way.

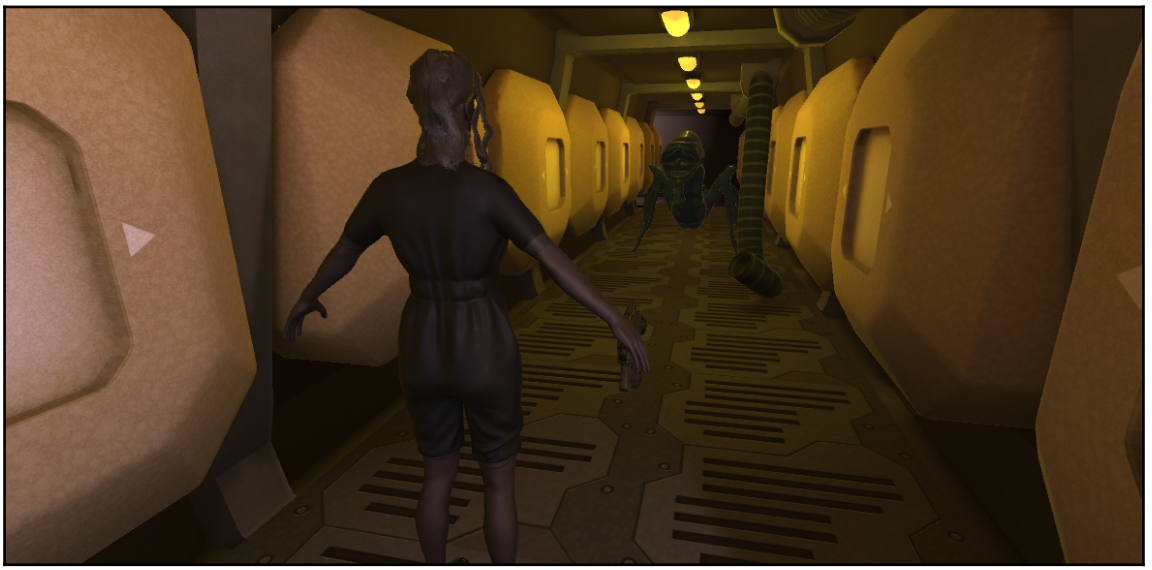
On screen, the effects often occur quickly, they tend to be simple, but need to react with other elements and each other in a specific way.

Starting the scene

As in previous chapters, our project assets are set up within a scene.

Open the `Chapter8_Start` scene.

This time, our game environment is a service corridor on the spacecraft where our astronaut has cornered the alien stowaway:



Initial scene state

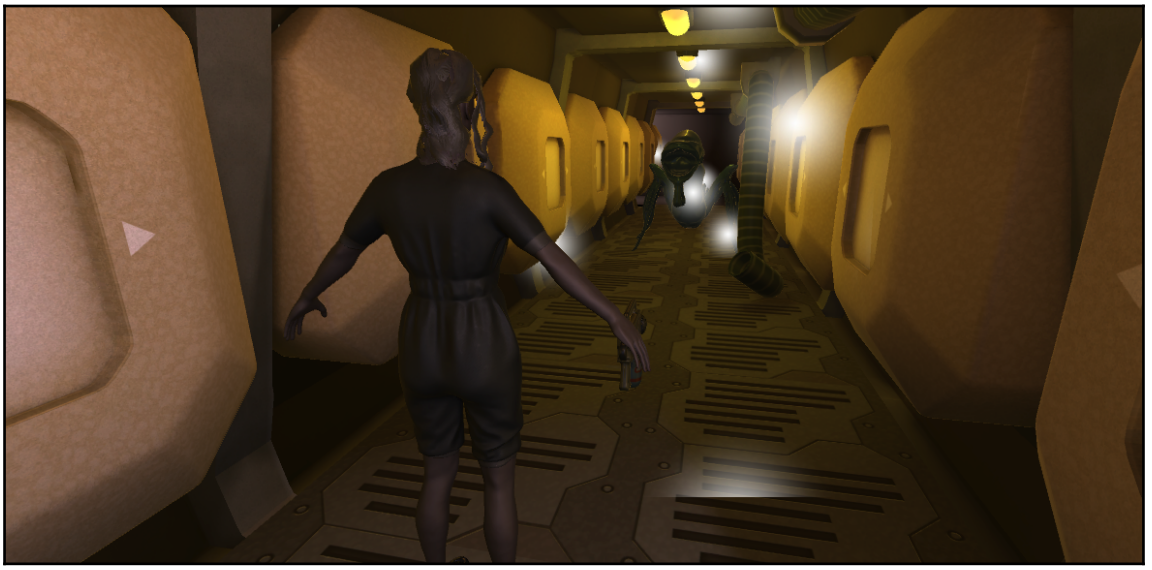
The astronaut is ready to deal with the alien menace with her handy flamethrower. We will start by setting up a particle system for this weapon.

Adding a particle system

We will start by creating a new particle system in the scene:

1. In the menu bar, navigate to **Game Object** | **Particle System** to add this to the scene.
2. Rename the particle system `flameShoot`.
3. In the **Hierarchy** panel, expand the `flamethrower` game object by clicking on the small arrow next to its name so its child, `muzzle`, becomes visible in the list.

4. Drag the new particle system onto the `muzzle` game object to parent it.
5. Click on `flameShoot` to view its components in the **Inspector**.
6. Zero out the **Transform Position** and **Rotation** values to center the particle system to `muzzle`.
7. Reset all three **Transform Scale** values to `1` so that the particle system does not inherit the parent's scale:



The default particle system scaled to the muzzle game object in the scene

Currently, the particle system does not look much like a stream of fire. We will make some adjustments to the system next.

Adjust the particle system parameters

The default particle system is quite different from the stream of fire that we need for the flamethrower effect. In this section, we will go through the list of parameters and make changes to the particle system's behavior:

1. In **Inspector**, scroll down slightly until the parameters of the **Particle System** component are visible.
2. Set the **Duration** value to `2.5`.

This will make a shorter, more continuous particle loop.

3. Leave the **Looping** checkbox checked and the **Prewarm** checkbox unchecked.

   Without **Looping**, our particle effect will just play once; and with **Prewarm**, it will be playing from the start without any input.

4. Leave **Start Delay** set to 0 as we want this effect to be immediate.
5. Click on the small arrow to the right of **Start Lifetime** and choose **Random Between Two Constants** from the drop-down list.
6. Enter the number `4` in the first field and `5` in the second field to define the amount of variation.

   This will create a little randomness in the duration of each particle.

7. Click on the small arrow to the right of **Start Speed** and again choose **Random Between Two Constants** from the drop-down list.
8. Enter the values `0.7` and `1.2` in the two fields respectively.
9. Set the **Start Size** value down to `0.4`.

   At this point, it should be much easier to see the particles in the scene.

10. Leave **3D Particle Rotation** unchecked.

    This checkbox will allow particles to rotate fully in three axes. Usually, we do not want to do this as their billboard geometry becomes quite apparent.

    Instead, we will modify **Start Rotation**.

11. Click on the small arrow to the right of **Start Rotation** and choose **Random Between Two Constants**.
12. Enter the values 0 and `360` in the two fields.

    We will leave the remaining parameters in this group at their default values and skip to the **Emission** group.

13. Set the **Rate** value to `12`.

14. In the **Shape** group, set **Angle** to 0 and **Radius** to `0.01`.

    The **Cone** emitter shape of the default particle system works well for the flamethrower effect, but here we will reduce the size of the emission area so that it is small enough to fit the nozzle of the flamethrower.

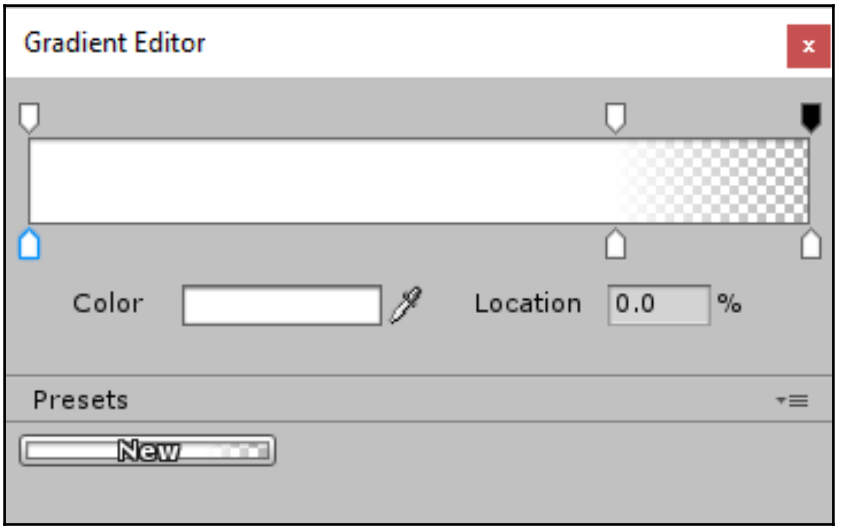
15. Click on the **Color over Lifetime** checkbox, clicking the parameter's name will expand its settings again.
16. Click on the gradient bar to edit the gradient.
17. In the **Gradient Editor**, create a new point about three-quarters of the way along the top of the gradient bar.
18. Click the marker and set **Location** to `75%`.

    By default, this point will have full opacity.

19. Click on the gradient bar, again three-quarters of the way along the bottom edge to create another marker.

    By default, the point should be set to white—RGB values all set to `255`.

20. Set **Location** to `75%`.
21. Click on the end marker along the top edge of the gradient bar and set its **Alpha** to zero by dragging the slider to the left:



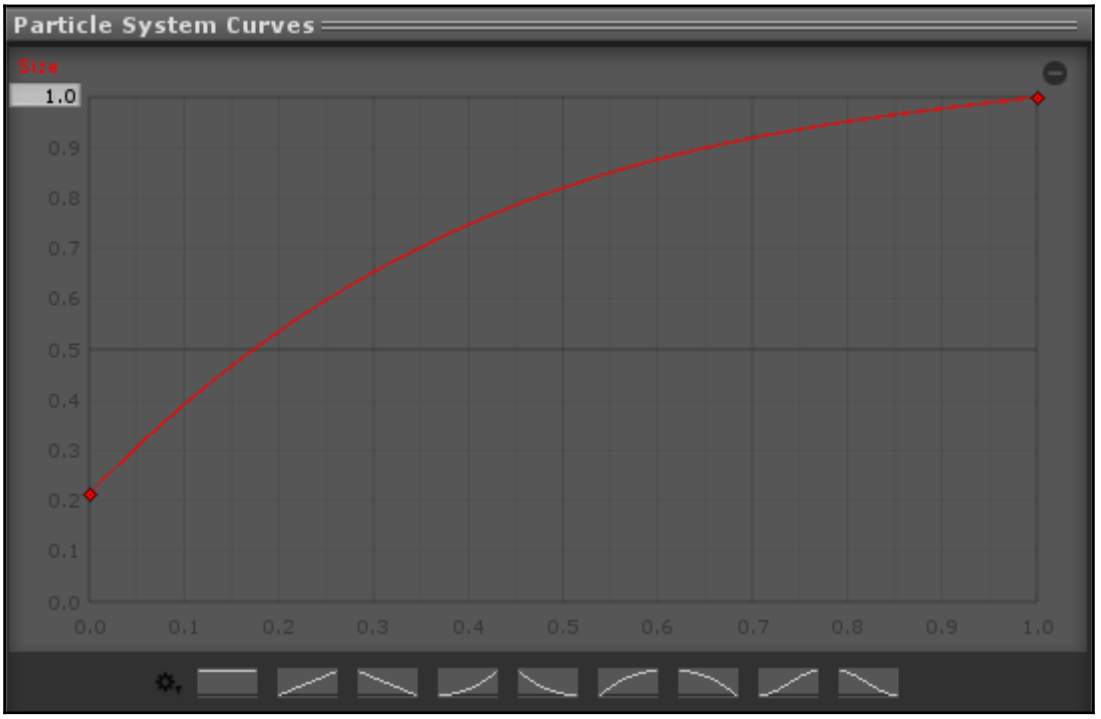
The adjusted Color over Lifetime gradient

Each particle will now fade out near the end of its life to create a more convincing jet of flame.

22. Click on the **Size over Lifetime** checkbox and then click on the collapsed bar to expand the additional parameters.
23. Click on the small arrow next to the value fields and choose **Curve** from the drop-down list.
24. A colored line now becomes visible in the field.

    We want our stream of flame particles to gradually increase in size after they are emitted. The line also becomes visible in the curve editor at the bottom of the **Inspector**.

25. Move the first key slightly down to around `0.2`.
26. Adjust the tangents of both of the keys so that the resulting curve approximates the curve in the following image:



The completed Size over Lifetime curve

Previewing the particle system in the **Scene** view should show the particles gradually becoming larger.

27. Check the **Rotation over Lifetime** checkbox.
28. Click on the parameter to expand its settings.
29. Leave **Angular Velocity** set to its default value of `45`.

    This will allow the particles to rotate as they move.

Currently, the particle system uses Unity's default particle material.

We can set up a unique material next.

Setting up a new material for the particle system

Setting up a new particle material will allow us to make a more convincing flamethrower effect:

1. In the **Project** panel, click on the `PACKT_Materials` folder to view its contents in the **Assets** panel.
2. In an empty area of the **Assets** panel, create a new material. Right-click and choose **Create** | **Material**.
3. Rename the new material `Flames`.
4. Click on the `Flames` material to view its properties in the **Inspector**.
5. In the **Shader** selection at the top of the **Inspector**, navigate to **Particles** | **Additive** from the drop-down list.

   The particle additive shader allows us to use a grayscale image to define the particle's shape and blend it with a tint color.

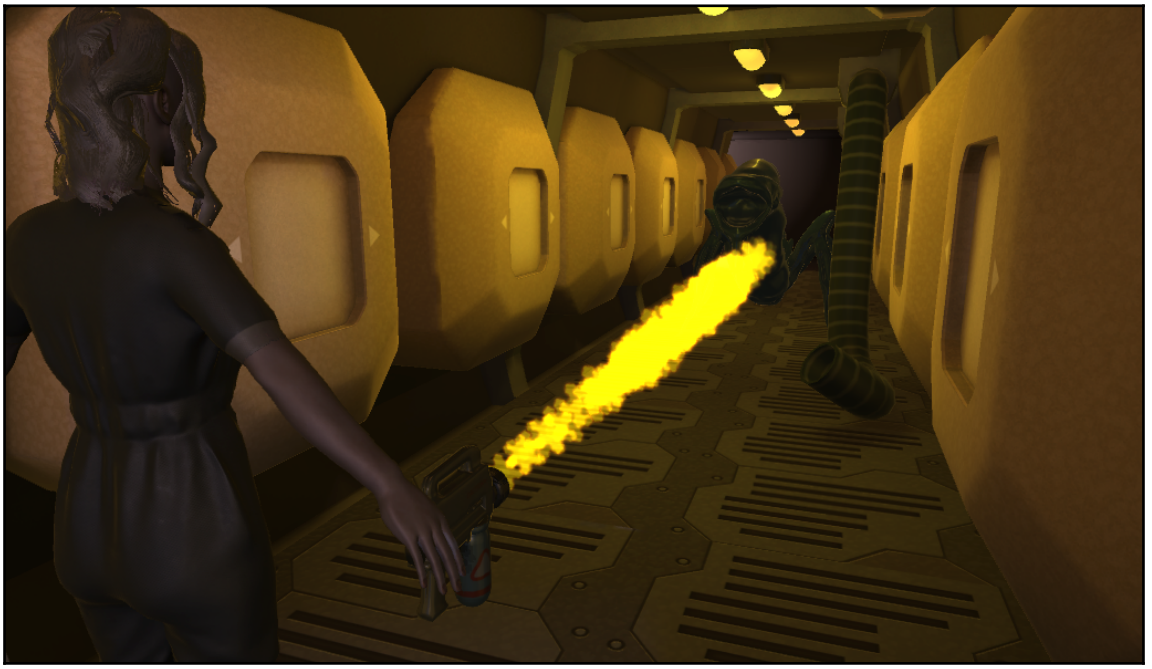
6. Drag the `Flames` material from the **Assets** panel onto the `Particle System` in the **Inspector** to use it.

   We will finish *adjust the material* properties now so that we can see the particle system in action.

7. Make sure that the `Flames` material is selected and click on the lock icon at the top of the **Inspector** to keep the **Inspector** focused on this item.
8. In the **Project** panel, click on the `PACKT_Textures` folder to view its contents in the **Assets** panel.

9. In the **Assets** panel, click on the `Particles` subfolder.
10. Locate the `pFlame` asset and drag it to the **Particle Texture** slot in the **Inspector**.
11. Click on the **Tint Color** swatch and choose a bright orange color.

    You can use the `FF800040` hex code to define a full orange with a lower alpha channel to see slightly more transparency in the flame particles, as shown in the following image:



The modified flamethrower particle effect

Creating the particle shader

In Unity particle systems use a unique kind of shader. We will create this next:

1. In the **Project** panel, click on the `PACKT_Shaders` folder to view its contents in the **Assets** panel.

2. In an empty area of the **Assets** panel, create a new shader. Right-click and choose **Shader** | **Unlit Shader**.

Unlit shaders do not respond to scene lighting. This makes them a great choice for particle effects that tend to use billboard textures rather than being actual geometry.

3. Rename the new shader `flameJet`.
4. Double-click on the shader to open it in **MonoDevelop**.
5. In MonoDevelop, change the first line of the shader to match the following:

```
Shader "PACKT/Unlit/flameJet"
```

Putting this shader in an `Unlit` subfolder will make it easy to select when it is used again in the project.

Let's apply the shader to the particle system and see how it looks straight out of the box:

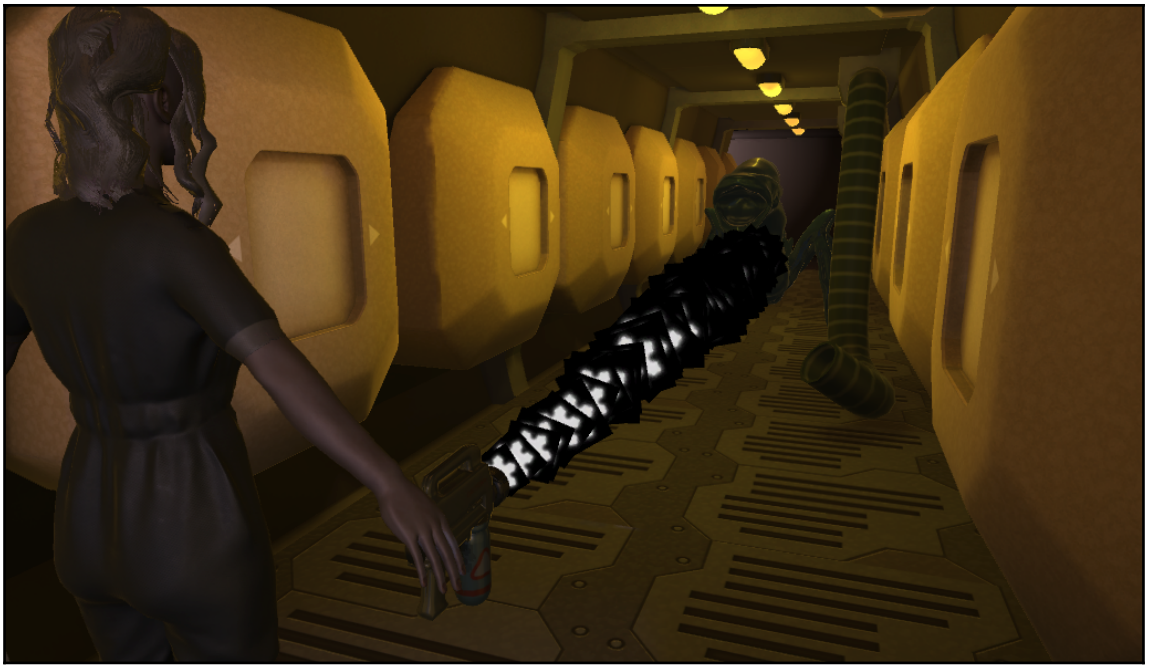
1. Minimize MonoDevelop and return to the main Unity interface.
2. In the **Project** panel, click on the `PACKT_Materials` folder.
3. When its contents appear in the **Assets** panel, click on the `Flames` material.

The material's properties will appear in the **Inspector**.

4. Click on the **Shader** button and navigate to **PACKT** | **Unlit** | **flameJet**.

The particle system's appearance will change in the **Scene** view.

5. Press the play button in the top center of the Unity interface:



The default unlit shader in the scene

By default, the unlit shader does not support transparency. It renders the entire quad with the black part of the texture that should be transparent.

Our next objective is to add transparency to the shader:

1. Return to MonoDevelop.
2. Within the `subShader`, replace the `Tag` line with the following code:

```
Tags { "RenderType"="Transparent" "Queue" = "Transparent"}
```

As we have done earlier, we will set `RenderType` to `Transparent` in order to allow the shader to remove transparent pixels. We will use the `Transparent` queue to make sure that Unity renders the pixels handled by this shader on top of opaque objects.

3. Delete the `LOD` line.

   Our particles will be rendered within a confined space, so we don't need to let Unity make a decision about whether to render them.

4. Add the following code:

```
Blend SrcAlpha One
ColorMask RGB
```

   The `Blend` line will blend the particles using the alpha channel in the texture. The `ColorMask` line is used to avoid sending the RGB data to the depth buffer, only the alpha channel is necessary.

5. Save the shader.
6. Return to the main Unity interface.
7. Press the play button.

   The particle system will start up. The edges of the particles will now be transparent, as shown in the following image:

Particles with transparent edges

Our flame particle is currently rendering as a white shape, but the edges of each quad appear to cull the particles behind. This is something we need to fix in the shader.

8. Add the following line of code:

```
ZWrite Off
```

   By default, geometry in the scene is rendered in a specific order. Objects rendered with a shader marked as transparent are rendered after opaque and cutout geometry, but they are also rendered in depth order in the scene.

   We can override this ordering by switching `ZWrite` off, rendering triangles together, and fixing the transparency issue.

9. Save the shader and check the particle simulation to see the improved effect.

In the next section, we will define a color.

Adding a color to the particle shader

Now that we have the transparency working, we need to add some code to the shader to define a color:

1. Return to MonoDevelop.
2. In the `Properties` block, add the following code:

```
_Color ("Color", Color) = (1,1,1,1)
```

3. Within the `appdata` struct, add the following line:

```
fixed4 color : COLOR;
```

   Here, we will define a low-precision float with four positions to support the standard RGBA channels.

   We need to add an identical variable in the `v2f` struct.

4. Locate the `v2f` struct and add the same line of code:

```
fixed4 color : COLOR;
```

   We will add an equivalent variable to the general Cg code to handle the `_Color` input. We will put it with the `_MainTex` variable that is included by default.

5. Locate the `_MainTex` definition. In the next line, add the following line:

```
fixed4 _Color;
```

6. Within the vertex to fragment the `v2f vert` structure, add the following line:

```
o.color = v.color;
```

   This will define the output color as those associated with the vertices now.

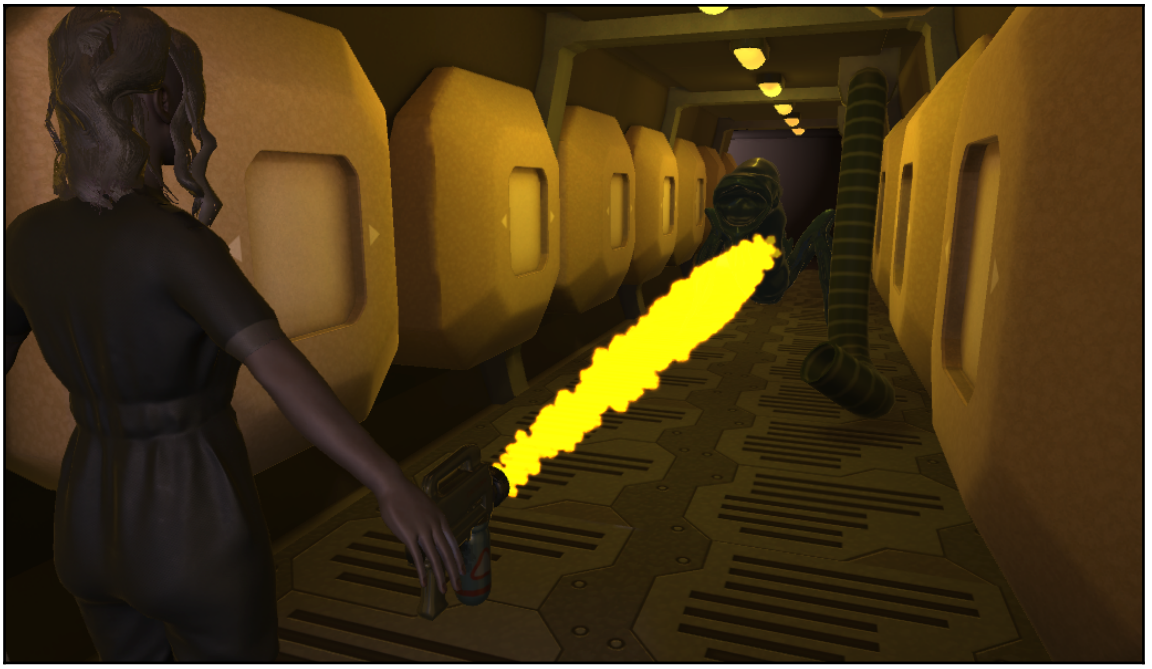
7. Finally, scroll down to the `fixed4 frag` definition at the bottom of the shader.
8. Replace the `fixed4 col` line with the following line:

```
fixed4 col = 2.0f * i.color * _Color * tex2D(_MainTex, i.uv);
```

   Here, we will add the defined `_Color` variable to the output color calculation by multiplying it with the `_MainTex` color.

9. Save the shader.
10. Return to the main Unity interface.
11. Notice that the `_Color` swatch has already been defined as the orange that we previously set. Unity remembers variables in shaders if they use standard naming conventions.

12. Press the play button to see the newly tinted shader:



Particle shader with color tint

Now that we have added transparency and color to the flame shader, let's add some additional particle effects to the scene.

We will start by creating some steam leaking from the broken pipes in the corridor.

Adding the steam particle effect

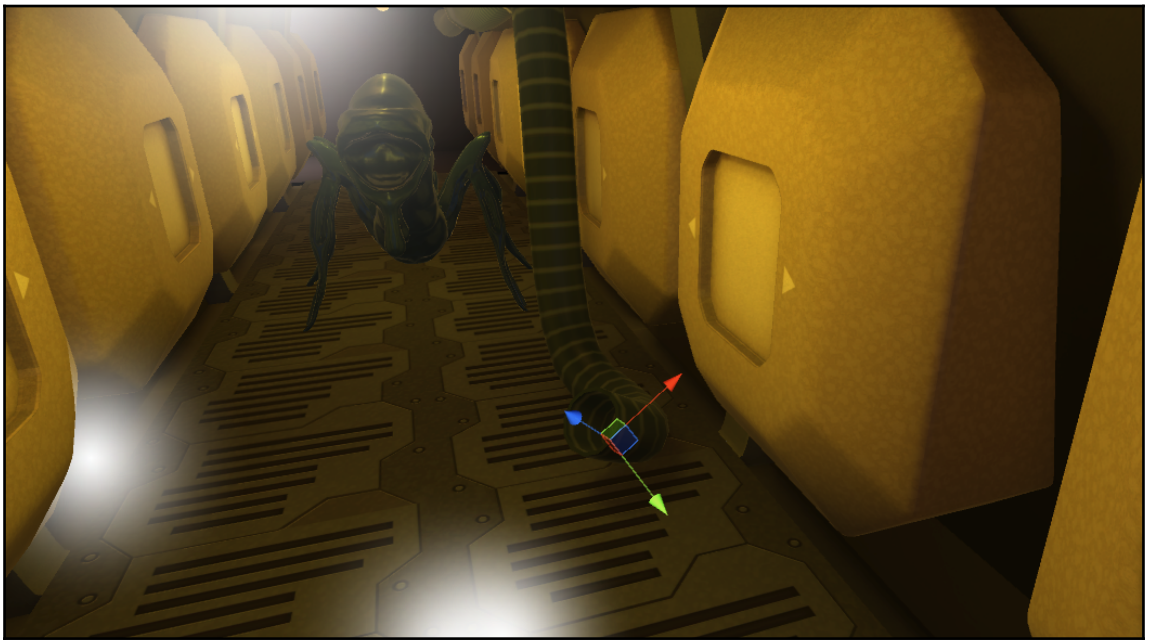
We want to add a steam particle effect in the corridor to add some ambience to the scene.

This time, we will use an animated sprite effect rather than a static particle texture.

We will start by adding the new particle system:

1. In the main Unity interface, create a particle system by navigating to **GameObject** | **Particle System**.
2. In the **Inspector**, rename the particle system `Particle_steam`.

3. Use the **Move** and **Rotate** tools to position the system near the end of the broken pipe:



Steam particle added to the scene

We need to scale down and slow down the particles to make them look more like a steam effect.

Adjusting the steam particle's parameters

We will make some adjustments to make the default particle seem more like steam coming out of the pipe:

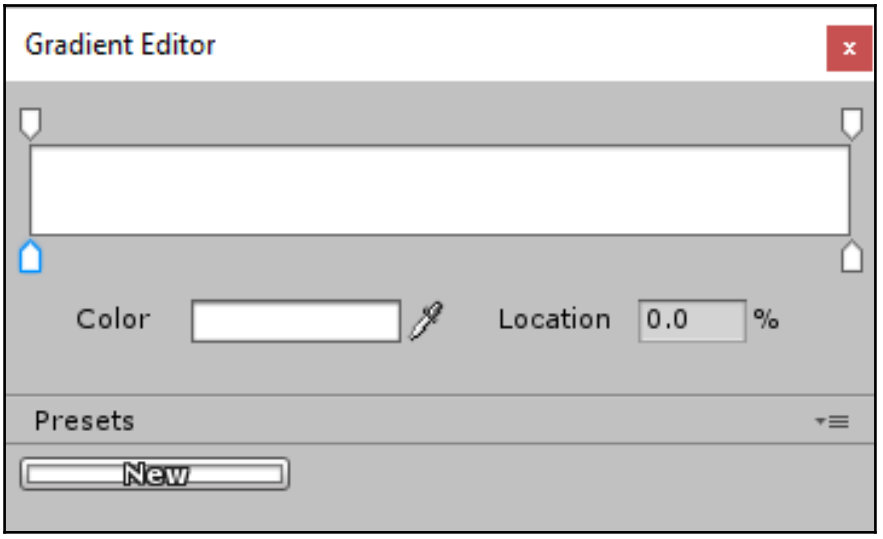
1. In the **Inspector**, set the particle system's **Start Speed** and **Start Size** to `0.25`.
2. Click the **Emission** tab and reduce **Rate** to `5`.

   This will slow down the rate at which the particles are emitted.

3. In the **Shape** group, reduce **Angle** to 10 and set **Radius** of the cone to `0.01`.

   This will cause the particles to be emitted from a smaller area.

4. Click on the **Color over Lifetime** tab and check the box to enable this function.
5. Click on the rectangular gradient bar to bring up the **Gradient Editor** in a floating window:



The default gradient bar

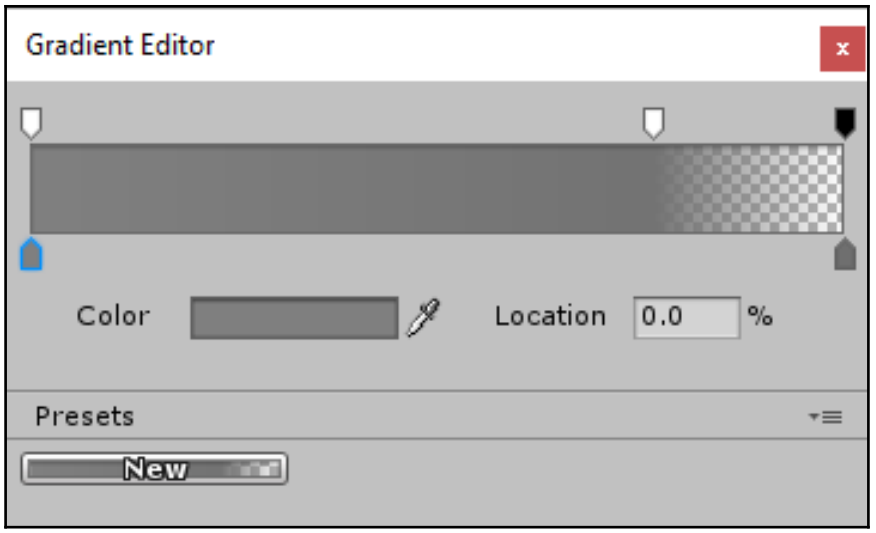
The default gradient just has a completely opaque white marker at either end. We want to adjust this to let the gradient fade out to soften its transition.

6. Click on the bottom-left marker and define a mid-gray color using the color picker that appears.
7. Enter the value `128` into all three of the RGB fields.
8. Define the bottom-right marker using the same color.
9. Along the top of the gradient bar, click to create another marker.

Markers along the top edge define opacity.

10. Use the fields to set **Location** of the new marker to `75.0`, and make sure that **Alpha** is set to `255`.
11. Click on the top-right marker and reduce the **Alpha** value to `0.0`.

    This will ensure that the particle completely fades out at the end of its lifetime:

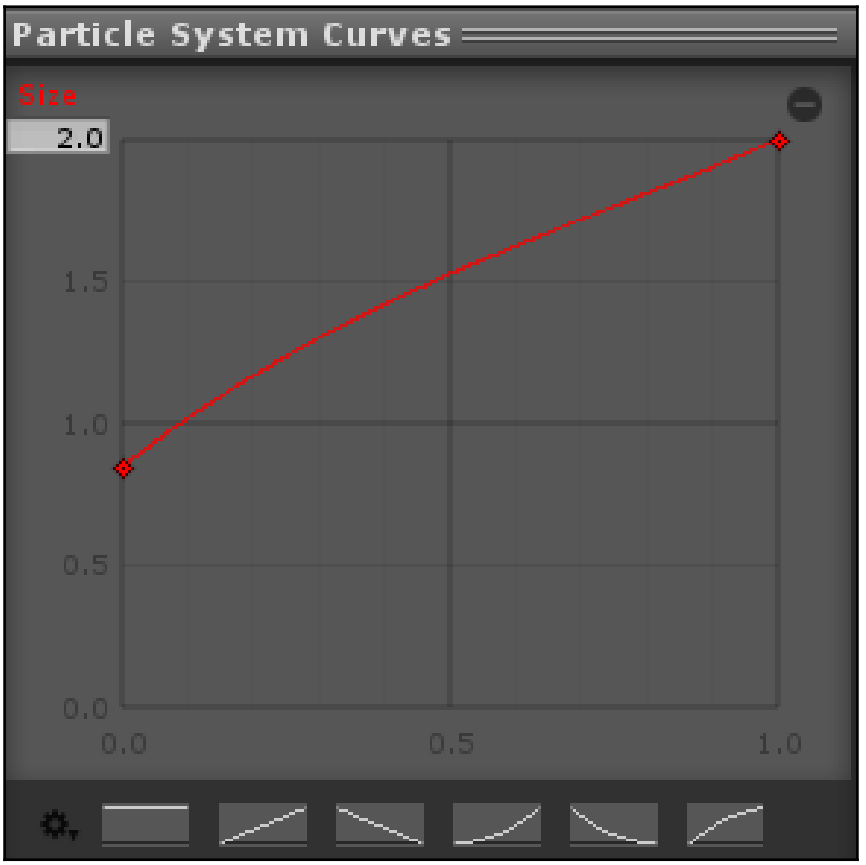


The adjusted color by lifetime gradient

    Next, we will adjust the particle size.

12. Click on the **Size over Lifetime** tab and check the box to enable this function.
13. Click on the curve bar to open the **Curve Editor**.
14. Define a curve that starts at just below `1.0`.

15. Click on the white field near the top of the curve and enter the value `2.0` to define the maximum size of the particle:



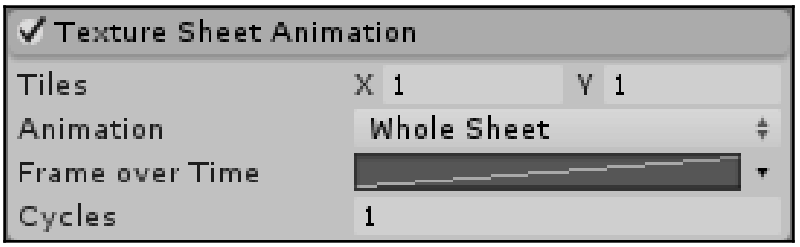
The adjusted Size over Lifetime curve

At this point, we will enable the texture sheet animation and complete the particle effect.

Completing the steam particle effect

Unlike the flamethrower effect, the steam particle is quite slow. We will get it to look more interesting using the particle system's built-in **Texture Sheet Animation** function:

1. Click on the **Texture Sheet Animation** tab and click on the checkbox to enable this function:



Texture Sheet Animation default values

   The fields in the **Tiles** group allow us to specify the amount of frames in the atlassed texture sheet.

   Our steam particle texture only has three horizontal frames.

2. Enter the value `3` in the **X** field.
3. Leave the **Y** field at its default value of `1`.

The **Frame over Time** curve gives us additional control over the timing of the frame transitions.

We will leave this at its default value.

The last parameter, **Cycles**, allows us to define how many times the animation sequence occurs over the particle's lifetime.

We will also leave this parameter at its default setting.

Our next step is to set up the material and add the texture sheet.

Setting up the steam particle material

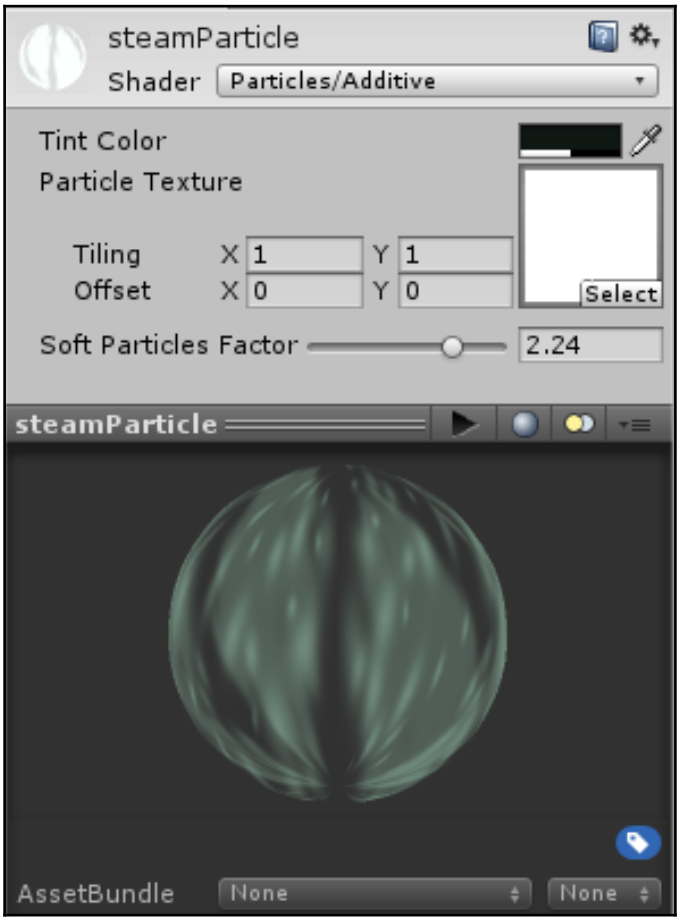
We need to create a unique material for the steam, as it uses its texture very differently:

1. In the **Project** panel, click on the `PACKT_Materials` folder to view its contents in the **Assets** panel.
2. In the **Assets** panel, right-click on an empty area and create a new material by choosing **Create** | **Material**.
3. Rename the new material `SteamParticle`.

    We will start with one of Unity's built-in shaders for this material.

4. Click on the **Shader** button at the top of the **Inspector** and choose **Particles** | **Additive** from the drop-down list.
5. Click on the **Tint Color** swatch and define a dull dark green color.
6. In the **Project** panel, click on the `PACKT_Textures` folder to view its contents in the **Inspector**.
7. Click on the **Particles** subfolder and drag `steamParticle` onto the **Main Texture** slot in the **Inspector**.

The content of the texture is white and the background is transparent, so the texture will look white in the preview slot:

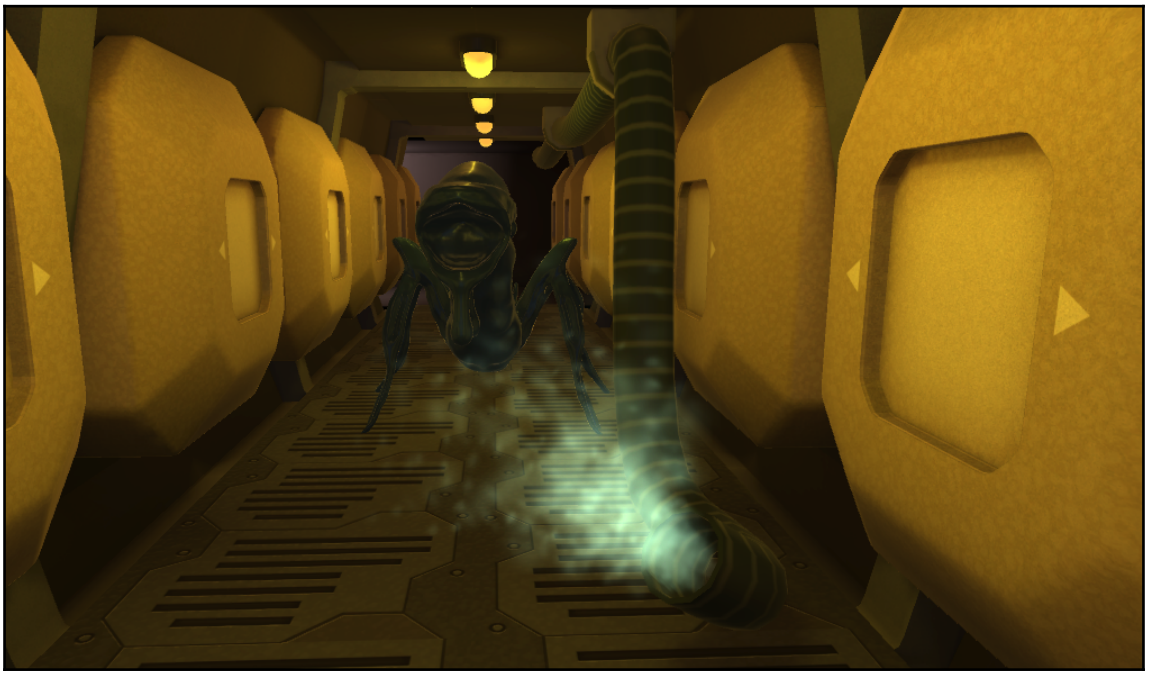


The completed SteamParticle material parameters

Let's add this material to our particle system and make the final adjustments:

1. In the **Hierarchy** panel, click on the `Particle_steam` game object to access its components in the **Inspector**.
2. In the **Inspector**, scroll to the bottom of the **Particle System** component and click on the **Renderer** tab.
3. Drag the `SteamParticle` material into the **Material** field in the **Renderer** group to replace the `default-particle` material.

Now that we have added the material, let's see what the particles look like in our scene:



The adjusted steam particles

While the particle system is selected, the particle system will simulate in the **Scene** and **Game** views.

The particle texture cycles between the three frames, but the effect is a little too uniform.

We will return to adjust one more value in the particle system:

1. In the **Inspector**, click on the **Rotation over Lifetime** tab within the **Particle System** component.
2. Check the box to enable this function.
3. Click on the small arrow next to the **Angular Velocity** value and choose **Random Between Two Constants** from the drop-down selection list.

   We want each of the particles to rotate independently to create a more organic looking effect.

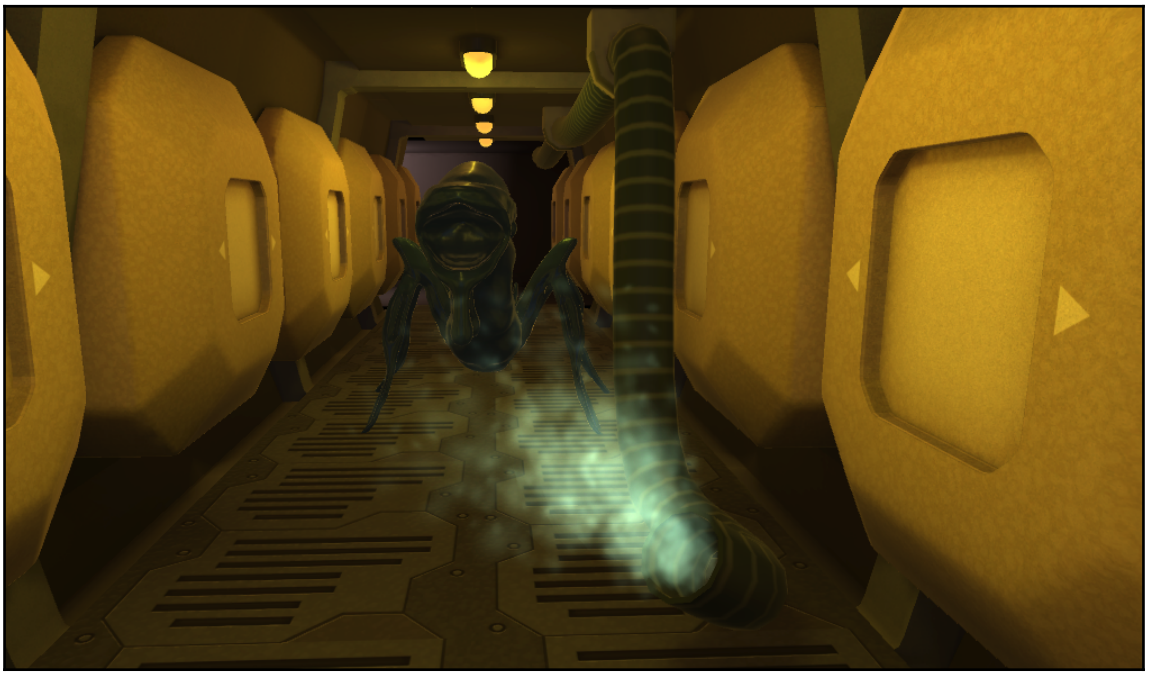
4. Leave the **Separate Axes** checkbox unchecked.

   We are still using billboard particles here that need to face the camera for the effect to be convincing.

5. Leave the **Angular Velocity** values at 0 and `45`.

   These default values should be adequate.

Take a look at the steam particles in the **Scene** view once more:



The completed steam particle system

Each particle now rotates randomly, creating a more realistic steam effect.

In the next section, we will create a different type of effect to enhance the alien creature.

Creating the slime-drip effect

In our game, something that makes the alien particularly fiendish is the slime dripping from it. This is a corrosive substance that damages the ship and may injure the astronaut.

As the dripping effect is slow, we want to use a mesh rather than a billboard particle. We will set this without a particle system to have greater control of the physics.

We will start by setting up the `drop` game object:

1. In the **Project** panel, click on the `PACKT_Models` folder to view its contents in the **Assets** panel.
2. Locate the `drop` model.
3. Drag this to the **Hierarchy** panel to add it to the scene.
4. Use the **Move** tool to position the `drop` game object close to the alien.
5. In the **Inspector**, zero out the **Rotation X** value in the **Transform** component so that the drop is oriented correctly.

The drop model added to the scene

At the moment, the drop uses Unity's default material. A suitable material has already been set up for this.

6. In the **Project** panel, click on the `PACKT_Materials` folder to view its contents in the **Assets** panel.

7. Locate the `droplet` material and drag it to the drop in the **Hierarchy** or **Scene** view.

The drop model will update in the **Scene** and **Game** views.

The droplet material uses a **Standard Specular** shader with a little transparency and high specular and smoothness levels to make it appear wet.

Next, we want the drips to affect the appearance of the flooring when they come in contact with it.

We can set this up next with a decal overlay technique.

Creating the floor damage effect

In this step, we will create an effect to simulate the damage to the environment when the alien's corrosive slime drips from it:

1. In the scene, create a new `quad` game object.

   We will prototype the effect with this geometry.

2. In the **Hierarchy** panel or the **Inspector**, rename the quad `burnQuad`.
3. Rotate the quad so that it is parallel to the floor by setting the **Rotation X** value to `90` in the **Transform** component in the **Inspector**.
4. Create a new material in the `PACKT_Materials` folder.
5. Name it `Corrosive`.
6. Drag the material to the quad to apply it.

   There is a transparent texture included in the project files that we will use to show the damage.

7. In the **Project** panel, click on the `PACKT_Textures` folder.

8. Locate the `burn` texture asset.
9. Drag this to the **Albedo** slot in the **Inspector**.

    The texture is transparent. We want to be able to show this as an overlay effect.

    Next, we will change the material's **Rendering Mode** to implement transparency.

10. In the **Inspector**, set the `Corrosive` material's **Rendering Mode** to **Cutout**.

In the **Scene** view, the edges of the quad will become transparent:

The quad added to the scene

With the **CutoutRendering Mode**, we can control the level of the alpha clipping with the **Alpha cutoff** slider in the **Inspector**.

With the **Alpha cutoff** value set to 0, the texture appears completely opaque. When we drag it all the way to `1`, the texture becomes completely transparent.

When the droplet hits the floor's surface, we want the burn overlay to grow and remain at the lowest value that does not show the edges of the quad.

We can handle this at runtime with a short C# script.

Writing the control script

Our control script will access the **Alpha** cutoff slider in the Standard Shader:

1. In the **Project** panel, click on the `PACKT_Scripts` folder to view its contents in the **Assets** panel.
2. In the **Assets** panel, create a new **C# Script** asset.
3. Rename the script `burnEffect` and double-click on it to open it in MonoDevelop.

   In MonoDevelop, we will start by defining some variables.

4. Add the following code directly inside the script's opening bracket:

```
public float cutoffValue = 0.95f;
public float cutoffMin = 0.5f;
public Transform burnPrefab;
public bool burning;
public float burnRate = 0.005f;
public Transform newBurnQuad;
```

The first float variable, `cutoffValue`, will be directly tied to the **Alpha cutoff** slider. We give it a value of `0.95` so that the effect starts a little quicker. There should be something visible as soon as the droplet hits the ground.

The next float, `cutoffMin`, is the lowest value that we want to use in the shader, under this, the transparent parts of the texture will become visible.

The next variable, `burnPrefab`, is used to store the quad. We will need to save this as a prefab in order to instantiate it at runtime.

We add a Boolean variable named `burning` to keep track of whether the effect is active or not. If we do not specify otherwise, the Boolean is inactive or `false` by default.

Next, we will add another float, `burnRate`, which will allow us to decelerate the effect, so that the corrosive appears to gradually lose its energy. We will set this to a very low value here, because it will be multiplied by time.

Finally, we will include a variable to store the instantiated object, `newBurnQuad`. This will allow us to access it again after it has been instanced.

We want something to happen when the droplet collides with the floor, so we will use Unity's built-in `OnCollisionEnter` function:

1. Add the following code beneath the `Update` function:

```
void OnCollisionEnter (Collision collision)
{
}
```

   This function is called whenever the game object that it is attached to collides with another collision object, such as the one attached to the floor in the scene.

   The function will give us some data about the collision, such as the point on the mesh that was collided with.

2. Add the following line inside the `OnCollisionEnter` function's brackets:

```
ContactPoint contact = collision.contacts[0];
```

   The function can store more than one contact point, but for this effect, we only care about the first one.

   We will store this contact point as a `Vector3` value, giving it a position in World Space.

3. Add the following line:

```
Vector3 pos = contact.point;
```

   To prevent further physics interactions with this game object, we can set it to ignore any more collisions.

4. Add the following line of code:

```
GetComponent<Rigidbody>().detectCollisions = false;
```

We also want the droplet to disappear when it hits the ground. For this, we will deactivate the **Mesh Renderer**.

5. Add the following line:

```
GetComponent<MeshRenderer>().enabled = false;
```

We could also use the `Destroy` method that would get rid of the droplet entirely, but this would also prevent anything further happening in this control script.

6. Next, add the following code:

```
newBurnQuad = Instantiate (burnPrefab, pos, Quaternion.Euler(90f,
  0f, 0f))as Transform;
```

This is where we will add the quad to the game scene.

We will instantiate the prefab that we defined as a variable. We set its position to the contact point that we defined earlier.

As we are using a quad, which is rotated to face along the z axis, we need to define a new Quaternion rotation to make it parallel to the ground.

Lastly, we set our Boolean:

1. Add the following line of code:

```
burning = true;
```

This is used as a flag to start the **Alpha cutoff** value adjustment in the `Update` function.

2. After the opening bracket of `Update`, add the following line:

```
if (burning)
{
}
```

We will use an `if` statement to check the status of our burning Boolean.

3. Within the `if` statement's brackets, add the following code:

```
Renderer burnRenderer = newBurnQuad.GetComponent <MeshRenderer> ();
burnRenderer.material.SetFloat ("_Cutoff", cutoffValue);
```

Here, we will create a local variable to access the quad's mesh renderer. This is how we will access its material. We will set the value of the `_Cutoff` float to equal `cutoffValue`.

As we have added this code to the `Update` function, it will run every frame, as long as the `burning` Boolean reads as `true`.

We will add a few more `if` statements to do our calculations.

4. Add the following line next within the `if` statement:

```
if (burnRate < 1f)
{
    burnRate += 0.001f * Time.deltaTime;
}
```

We add the conditional `burnRate < 1f` to make sure that the value is not already sufficiently high. We will then multiply it by a small value and `Time.deltaTime`. This is the time in seconds.

5. Add the next `if` statement:

```
if(cutoffValue >= cutoffMin)
{
    cutoffValue -= burnRate;
}
else
{
    cutoffValue = cutoffMin;
    burning = false;
}
```

Here, we will check to make sure that `cutoffValue` is higher than its lowest acceptable value. If this is the case, we will reduce it by the `burnRate` value.

We will add an `else` statement to take care of the other possibility, setting `cutoffValue` to `cutoffMin` and setting `burning` to `false`.

6. Save the script.
7. Return to the main Unity interface.

In the next section, we will make some changes to the `drop` game object.

Modifying the drop prefab

We now need to add the script to the `drop` game object:

1. In the **Project** panel, click on the `PACKT_Prefabs` folder to view its contents in the **Assets** panel.
2. Drag the `drop` game object from the **Hierarchy** panel into the **Assets** panel to make it into a prefab.
3. Click on the `PACKT_Scripts` folder again and drag the `burnEffect` script onto the `drop` to add it as a component.

   We also need to add **Rigidbody** and **Collider** components for the physics to work.

4. Use the menu bar to add a Rigidbody component by navigating to **Component** | **Physics** | **Rigidbody**.
5. Next, add the **Box Collider** component from the same drop-down list.

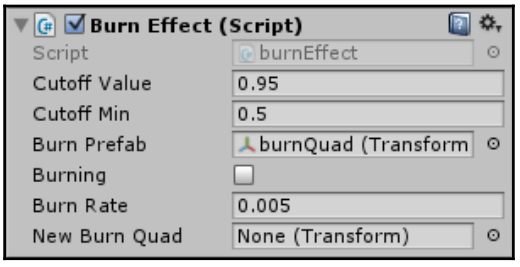
   We need to save the `burnQuad` game object as a prefab so that it can be added to the script component.

6. Drag the `burnQuad` object from the **Hierarchy** panel into the `PACKT_Prefabs` folder in the **Assets** panel.

   Next, we need to specify this as the prefab to be instantiated in the script.

   We have to do this at the prefab level for this change to be stored.

7. Select the `drop` prefab in the **Assets** panel.
8. Drag the `burnQuad` prefab from the **Assets** panel into the `burnPrefab` slot in the `droplet` game object's `burnEffect` script component in the **Inspector**:



The drop prefab's Burn Effect Script component

9. Scroll to the top of the **Inspector** and save the changes to the prefab by clicking on the **Apply** button.

   The `alien` game object in the scene already has a script set up to instantiate drops, we just need to add the new prefab to the script component.

10. In the **Hierarchy** panel, click on the `alien` game object.
11. Drag the drop prefab from the **Assets** panel onto the **Drop Prefab** slot in the `Alien Drops (Script)` component that is visible in the **Inspector**.
12. Delete the `drop` and `burnQuad` game objects from the scene.

    This will make it easier to see our new effects as they are produced.

13. Preview the scene by pressing the play button in the top center of the Unity interface:

The completed slime-drip effect

The drops are emitted randomly from two defined points in the `alien` game object hierarchy.

When the drops hit the floor's collider, they instantiate the `burnQuad` prefab, at which point, we see the animated cutoff effect.

Summary

In this chapter, we demonstrated the major differences between particle shaders and the other lit shaders that we have created in the previous chapters.

We also set up particle systems to represent different effects in our spacecraft scene.

Next, we wrote a custom unlit shader to further demonstrate the functionality of particles in Unity.

We set up an animated sprite effect for the steam particle in the scene to demonstrate this additional feature of the particle system.

In the final section, we created a more complex two-part effect using a little code to coordinate the slime droplets and corrosion in our spacecraft corridor environment. This effect offered an alternative to using the particle system in a scene.

In the next chapter, we will explore the possibilities of shaders that can work within the limits of mobile devices, such as phones and tablets.

9
Optimizing Shaders for Mobile

In this chapter, we will take a closer look at specialized shaders for mobile games. Typically, mobile devices do not have the performance capabilities of consoles and PCs, they also do not require the same level of texture detail, but by writing a few custom shaders, we can still get the game levels looking great.

In this chapter, we will cover the following topics:

- Outputting builds for iOS and Android platforms
- Looking at game performance in the **Profiler**
- Writing a simple mobile shader
- Creating an optimized character shader

Let's take a look at the scene and start with our next shader.

Starting the scene

The scene that we will be working with in this chapter is located in the `PACKT_Scenes` file in the project:

1. In the **Project** panel, click on the `PACKT_Scenes` folder to view its contents in the **Assets** panel.
2. Locate the `Chapter9_Start` Unity scene file.

3. Double-click on the file to open the scene in the editor:

Initial scene setup

In this scene, our astronaut fights the alien that wiped out the research team on the planet. The action takes place in the spacecraft's loading dock.

The scene uses standard shader materials.

In the next step, we will build the scene to test it on a device.

Building the scene for a device

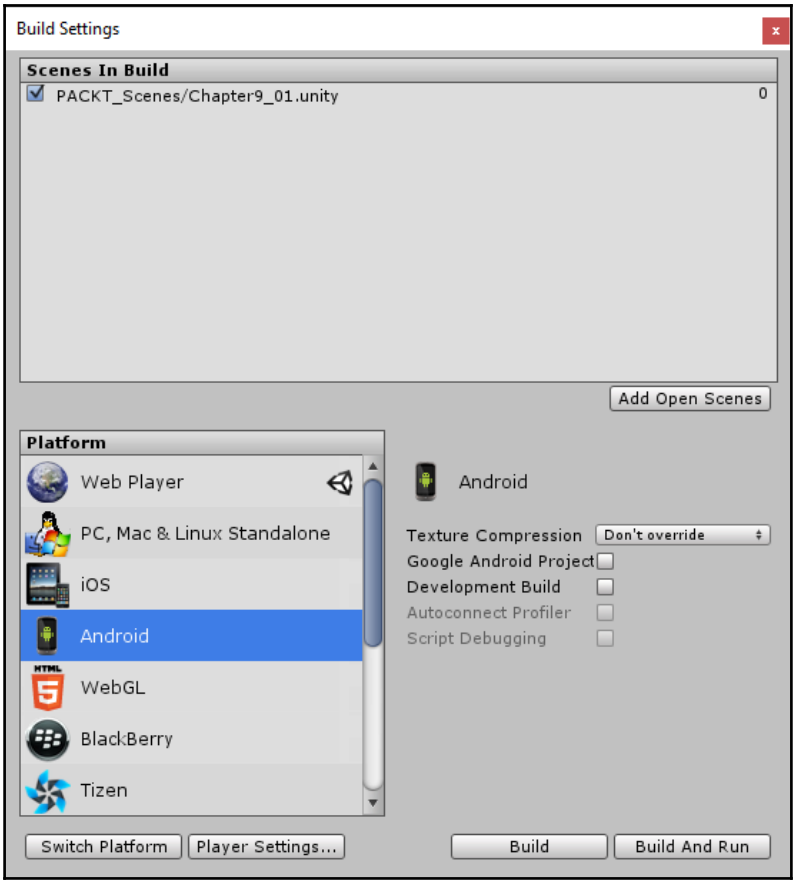
We will go through the steps to create a build for Android and iOS devices.

Building for Android

There are just a few steps to create the Android APK:

1. Make sure that **Android SDK** and **Java SDK** are installed and updated to the latest versions.
2. Open the **Build Settings** window by navigating to **File** | **Build Settings**.
3. Click on the **Add Open Scenes** button at the top half of the window.
4. `Chapter9_Start` should now appear in the list.
5. In the **Platform** list, click on **Android**.

It should be highlighted as follows:



Build Settings for Android device

6. Click on the **Switch Platform** button.
7. The device should be connected with a USB cable and have **USB Debugging** switched on in the **Advanced Settings** or **Developer Settings** on the device.
8. Click on the **Build** button to output the game build.

   After running for a short period, a dialog will appear asking you to specify the Android SDK root folder location.

9. Navigate to the Android SDK root folder and click on the **OK** button to continue the build.

   The build will finish and the designated build folder will pop up on your desktop.

10. Create a name for the APK file and click on **Save**.

    The build will be written to the connected device. It will automatically run on the device and look similar to the following illustration:

Android build screenshot

Building for iOS

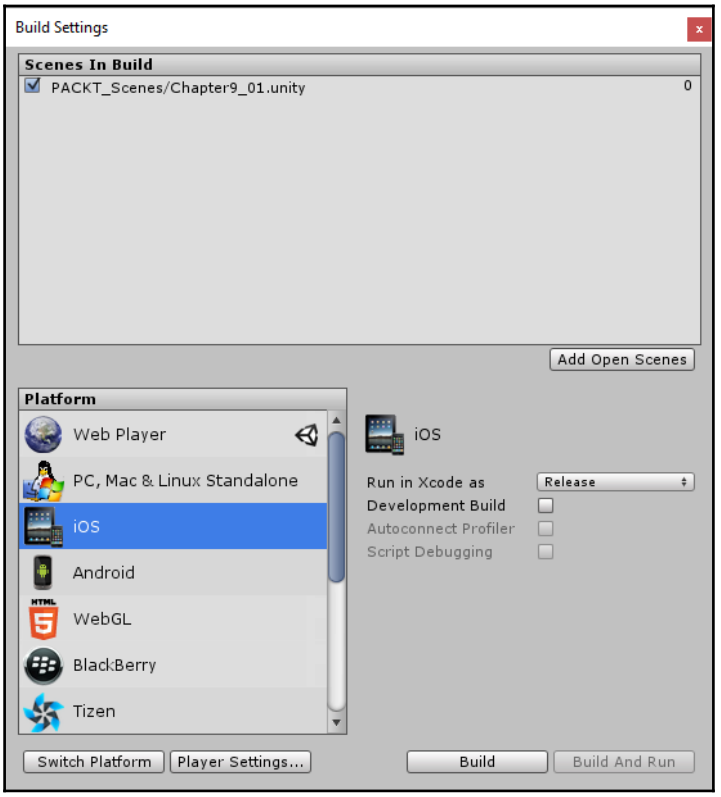
We have a few steps to follow to build for iOS:

1. Firstly, you need to set up your Apple developer account.

   This process is documented on Apple's developer portal.

2. Ensure that you have the most recent **XCode** version installed on your machine.
3. Connect your build device to your machine with a USB cable.
4. In Unity, open the **Build Settings** window to target the correct platform by navigating to **File** | **Build Settings**.
5. Choose **iOS** from the **Platform** list.

   The window should appear as follows:



iOS Build Settings dialog

6. Click on the **Player Settings…** button.

   The **Player Settings…** will appear in the **Inspector**.

7. Set the **Bundle Identifier** to the correct ID.

   It should begin with `com`, followed by your developer ID, and should terminate in the product name specified in your developer account.

8. Back in the **Build Settings** window, click on the **Build and Run** button to initiate the build.

   If the build compiles correctly, you should have a new application available on your test device. It will automatically run and look similar to the following illustration:

iOS build screenshot

We can usually see if the game is struggling on the device when we test it. When this happens, we can get descriptive information in **Profiler** about what is making the game lag.

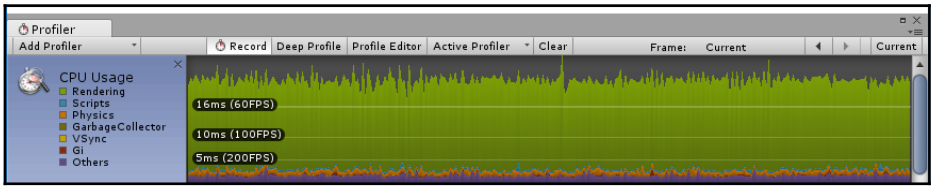
Viewing the frame rate in the Profiler

To view the Profiler information in the editor, we need to perform the following steps:

1. Make sure that the device and the build computer are connected to the same Wi-Fi network.
2. Check the **Development Build** checkbox in the **Build Settings** window.
3. Check the **Autoconnect Profiler** checkbox when this becomes active.
4. Click the **Build and Run** button.

When the build is complete, it should run on the connected device. The **Profiler** window will appear over the main Unity interface, displaying a graph of the game's performance per frame in milliseconds.

If the **Profiler** does not automatically display the connected device, you can select the device from the list using the **Active Profiler** tab in the **Profiler** window:



Initial Profiler graph for a connected Android device

There is a separate graph for the **CPU** and other processes.

These are further color-coded in different categories, such as **Rendering**, **Scripts**, and **Physics**, which affect the performance.

We can see that majority of the processing is taken up by the rendering, displayed here in green.

Typically, we want to keep the frame rate above 30 frames per second to keep the game playing smooth on the device.

If you move the character around the space, you will see noticeable spikes in the frame rate.

In the next section, we will attempt to improve the frame rate by replacing some of the more complex shaders with a reduced mobile shader.

Writing a simple mobile shader

Mobile shaders tend to be more efficient—ready to be run with fewer and smaller textures for a reduced screen size.

Unity comes with quite a few Mobile shaders ready to be used in a project. Unlike the Standard shaders, mobile shaders are not automatically optimized, so they need to include the exact inputs and features that you want to use in a material.

One shader that is missing is a **Mobile Diffuse Specular** shader. We can write this shader and compare it in our mobile build:

1. In the **Project** panel, select the `PACKT_Shaders` folder to view its contents in the **Assets** panel.
2. In an empty area of the **Assets** panel, create a new shader asset. Right-click and choose **Create** | **Shader** | **Standard Surface Shader**.

   A new shader will be created in the folder.

3. Rename the shader `m_diffuse_spec`.

   In this case, we will be deleting all the default code and writing the shader from scratch, but we need the shader shell set up in the **Assets** Unity project folder.

4. Double-click on the `m_diffuse_spec` asset to open the shader in **MonoDevelop**.
5. In MonoDevelop, select and delete all of the default code.
6. Start the shader with the following line of code:

```
Shader "PACKT/Mobile/diffuse_specular" {
```

   The first line defines the shader's name that will appear in the selection list of **Material**. Here, we will add in another subfolder so that the mobile shaders will be distinct from the other shaders that we have in the project.

7. Next, add the following code:

```
Properties {
    _MainTex ("Base (RGB) Gloss (A)", 2D) = "white" {}
    _Shininess ("Shininess", Range (0.03, 1)) = 0.078125
}
```

We will add our common `_MainTex` property to handle our albedo texture. Unlike in some of the other shaders we have written, we will omit the `_Color` property. The tint color is an additional line of code that will increase the performance cost of the shader.

The default code for the `_Color` variable also uses a `float4` variable, which is the highest precision float and largest data size. Omitting it will be a significant optimization to this shader.

In this case, the glossiness will be included as an alpha channel in the texture map, so we can define the specularity, or shininess, as a simple value.

Next, we will add the `SubShader` section.

8. Add the following code:

```
SubShader {
    Tags { "RenderType"="Opaque" }
    LOD 250
```

The `SubShader` encapsulates the rest of the code in the shader and can be used to separate the different versions that can run on low-end devices if necessary. For now, we will use simple tags. The surface that the shader will be used for is not transparent, so we will use the `Opaque` render type.

The `LOD` value defines the distance in world units that the surface handled by the shader will still be written for. Giving a shader a higher **Level of Detail** (**LOD**) value ensures that it will still be written.

In this case, our scene is a small enclosed space, so the LOD value in this shader will never have to be compared.

Next, we will add our CG snippet.

9. Add the following code:

```
CGPROGRAM
#pragma surface surf MobileBlinnPhong exclude_path:prepass
  nolightmap noforwardadd halfasview interpolateview
```

Within the CG part of the shader, we will include the shader compile directive that allows us to specify the kind of shader we are creating and the inputs to be read.

In this case, we are using the `MobileBlinnPhong` shader, which is defined in the `CGIncludes` file. We will set the shader to not use light maps; typically, textures used in fixed assets on mobiles incorporate lighting in their main texture.

The `noforwardadd` keyword restricts the scene lighting to a single directional light. Other lights in the scene can still affect the object if baked into light probes. The next keyword, `halfasview`, combines and normalizes the light direction and view direction, speeding up the lighting calculation.

There are some more good suggestions for optimizing shaders on the Unity website at:
`https://docs.unity3d.com/Manual/SL-ShaderPerformance.html`.

Next, we will define the shader's lighting interaction.

10. Add the following code:

```
inline fixed4 LightingMobileBlinnPhong (SurfaceOutput s, fixed3
  lightDir, fixed3 halfDir, fixed atten)
{
    fixed diff = max (0, dot (s.Normal, lightDir));
    fixed nh = max (0, dot (s.Normal, halfDir));
    fixed spec = pow (nh, s.Specular*128) * s.Gloss;
    fixed4 c;
    c.rgb = (s.Albedo * _LightColor0.rgb * diff + _LightColor0.rgb
      * spec) * atten;
    UNITY_OPAQUE_ALPHA(c.a);
    return c;
}
```

Here, we will create variables to keep track of the light direction and attenuation that we will pass to the shader in order to determine the amount of specularity that we should add to the shader output.

Using fixed float variables is a great optimization for mobile. Fixed floats are the lowest precision numbers and take less computing time than float or half types.

Next, we need to set up some variables to handle the texture map input.

11. Add the following code:

```
sampler2D _MainTex;
half _Shininess;
```

    We will add a `sampler2D` variable to process our `_MainTex` input and a `half` variable, four eight-bit channels, to handle the `_Shininess` slider input.

    Next, we need to set up a variable to handle the UV coordinates for `_MainTex`.

12. Add the following code:

```
struct Input {
    float2 uv_MainTex;
};
```

    The `uv_MainTex` float will be used in the `surf` output definition to apply our albedo texture.

    Our next step is to set up the `surf` function.

13. Add the following code:

```
void surf (Input IN, inout SurfaceOutput o) {
    fixed4 tex = tex2D(_MainTex, IN.uv_MainTex);
    o.Albedo = tex.rgb;
    o.Gloss = tex.a;
    o.Alpha = tex.a;
    o.Specular = _Shininess;
}
```

    We will define the surface output values as we have done before. This time, the `_MainTex` values do not have to be multiplied by `_Color`, as this has been omitted.

    In the `o.Albedo` line, we need to specify `tex.rgb` as the output because the texture has four channels and albedo can only use three.

    We will specify the `Gloss` and `Alpha` output as `tex.a`. Note that we have to specify an output for `Alpha` to satisfy the shader type input requirements, even though, as an opaque shader, the transparency is never implemented.

    We still need to close the CG snippet.

14. Add the following line:

```
ENDCG
```

    We will then need to add a closing curly bracket to terminate `SubShader`, otherwise the shader will fail to compile.

15. Add the closing curly bracket in the next line.

    Lastly, we can specify a suitable fallback shader that can be run on an older device that cannot handle this shader.

16. Add the following code:

```
FallBack "Mobile/VertexLit"
```

17. Add the closing bracket to end the shader.
18. Save the shader and minimize MonoDevelop.

It is time to view the shader in the **Scene** view and test it in our build.

Replacing shaders in scene materials

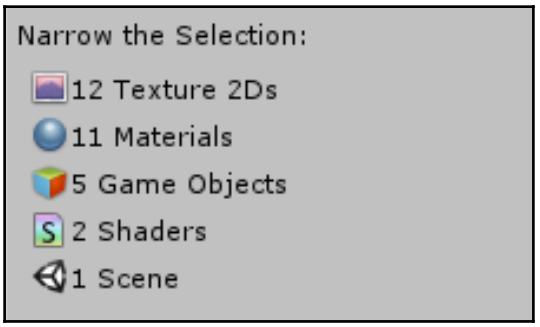
At this point, we will need to replace the scene shaders with our new mobile shader.

We can quickly select all of our scene materials with the **Select Dependencies** option in the **Assets** panel:

1. In the **Project** panel, click on the `PACKT_Scenes` folder to view its contents in the **Assets** panel.
2. In the **Assets** panel, locate the `Chapter9_Start`.
3. Right-click on the `Chapter9_Start` asset.
4. Choose **Select Dependencies** from the drop-down list that appears.

    All the assets that the scene uses will appear in the **Assets** panel.

In the **Inspector**, you will see a list of the assets sorted by type:



Selected scene materials

5. Click on the **Materials** list item to select all the Materials in the scene.

   There are a few materials that we don't want to replace with this shader.

6. Deselect the **Starfield** and **Helmet** materials in the **Assets** panel by *Ctrl* + clicking them (Use *command* + click if you are working on Mac).

   These materials require more complexity than our custom shader can handle, so we will leave this as they are.

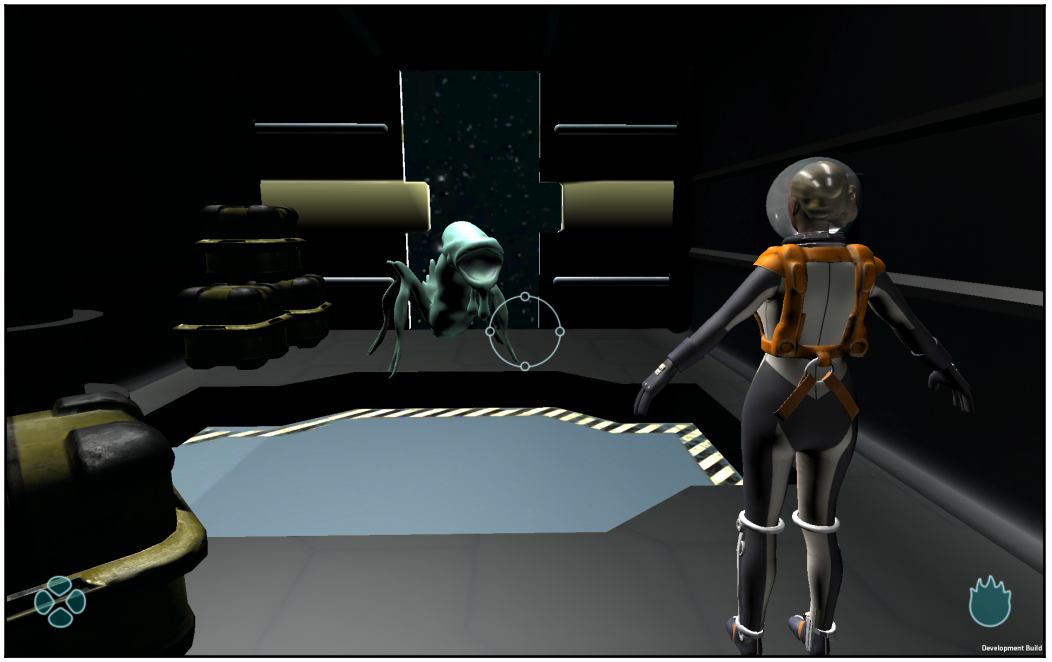
   The remaining scene materials are still selected and the sample spheres will be visible in the **Inspector**.

7. At the top of the **Inspector**, click on the **Shader** selection and choose **PACKT** | **Mobile** | **diffuse_specular** from the drop-down list.

   This will cause the surfaces in the scene to be overlit. We can compensate for this by reducing the strength of the main light.

8. In the **Hierarchy** panel select the **Directional Light**.
9. In the **Inspector**, reduce the **Intensity** value to `0.5`.

This will result in more even lighting in the **Game** and **Scene** views:



The scene is now using the custom mobile shader

10. Delete the previous build from your test device.
11. Open the **Build Settings** window by navigating to **File** | **Build Settings**.
12. Run a new build by clicking on the **Build** button.

Generally speaking, the fewer shaders you can get away with using in a mobile project, the better.

The shader we just implemented is very simple and we may want to make hero assets in our scene, such as the astronaut and alien intruder, look a bit more special by incorporating some of the physically-based rendering techniques that Unity 5 gives us to work with.

These elements are likely to receive more scrutiny in the game and should be lit with real-time lights as they will move at runtime. We will write a more advanced shader next.

Writing an advanced mobile shader

When we reduced the complexity of the shaders for our mobile build, we lost some of the surface in our astronaut and alien. In this step, we will build a compromise shader that we can use for both of these characters that will improve their appearance and not contribute significantly to the loss of frame rate on our mobile build.

Both the astronaut and alien should require specular and normal maps to look really good. Unlike the rest of the scenery, they are lit real-time, so we will build a compromise shader that uses some of the **Physically Based Shading** (**PBS**) options:

1. In the **Project** panel, click on the `PACKT_Shaders` folder to view its contents in the **Assets** panel.
2. In an empty area of the **Assets** panel, right click and choose **Create** | **New Shader** | **Standard Surface Shader** from the drop-down list.
3. When the new shader asset is created in the **Assets** panel, rename it `m_character`.
4. Double-click on the new shader to open it in MonoDevelop.
5. In MonoDevelop, replace the first line of the shader with the following code:

```
Shader "PACKT/Mobile/Character" {
```

   As with our last shader, we can make this one easier to find in our project by making it accessible in the `Mobile` subfolder in our **Shader** drop-down list.

6. In the `Properties` block, delete the `_Color` property line.

   The color tint is unnecessary here, as our albedo texture already handles the surface color.

7. Delete the properties for `_Glossiness` and `_Metallic`.
8. Add the following lines of code:

```
_Specular ("Specular", Range(0,1)) = 0.0
_BumpMap("Normal Map", 2D) = "bump" {}
```

   We are using a range slider to define our specular value here, so we do not need to add an additional texture map to the project.

   We will also define the `_BumpMap` property to handle the character's normal map.

9. Within the `CGProgram` snippet, amend the default shader directive to read as follows:

```
#pragma surface surf StandardSpecular fullforwardshadows
```

   Using the `StandardSpecular` directive will allow us to use the specular workflow. Without this, our specular input will not be recognized.

10. Under the `sampler2D MainTex` variable definition, add the following line:

```
sampler2D _BumpMap;
```

    This will handle the normal map texture.

11. Scroll down to locate the `half _Metallic`, `half _Glossiness`, and `float4 _Color` lines and delete them.
12. Add the following line:

```
half _Specular;
```

    This will allow us to use our `_Specular` slider input in the shader code.

13. Amend the `surf` function line to include the `StandardSpecular` directive as follows:

```
void surf (Input IN, inout SurfaceOutputStandardSpecular o) {
```

    As we have removed the `_Color` variable, we can also remove it from our albedo texture calculation.

14. Locate the `fixed4 c` line in the `surf` function and amend it to read as follows:

```
fixed4 c = tex2D (_MainTex, IN.uv_MainTex);
```

15. Locate the `o.Metallic` line and delete it.
16. Amend the `o.Smoothness` line to use the `_MainTex` alpha as follows:

```
o.Smoothness = c.a;
```

17. Add the following line to use the `_Specular` slider value as the `specular` output:

```
o.Specular = _Specular;
```

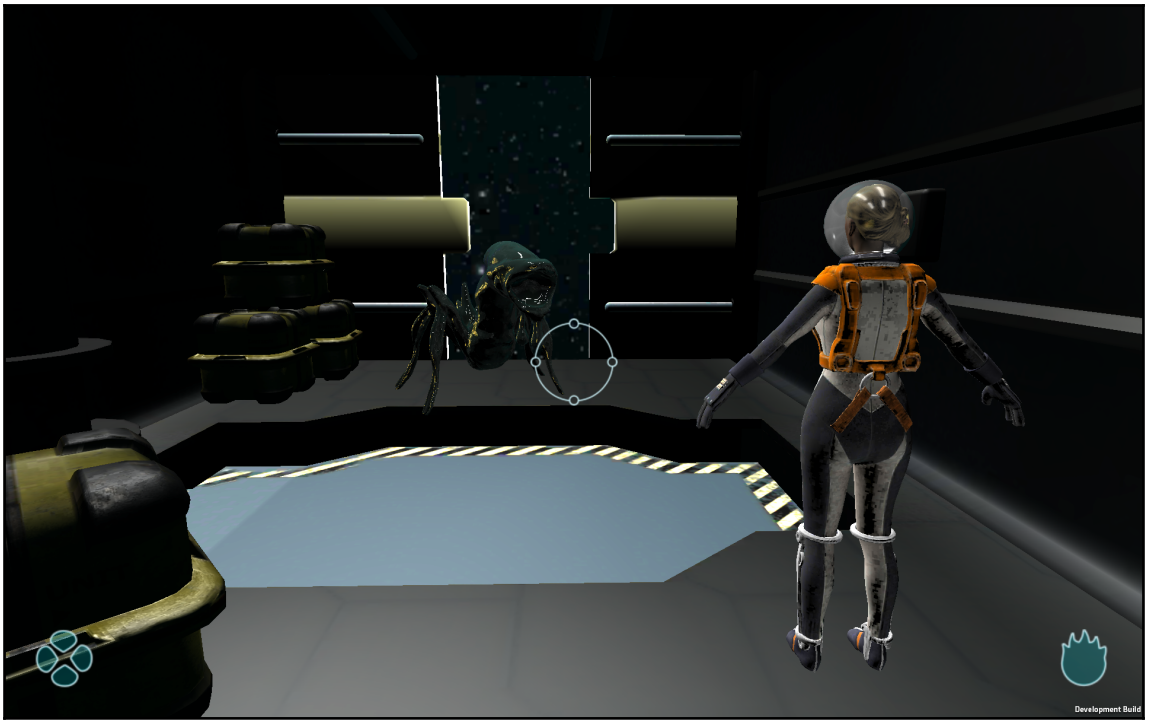
18. Finally, add the following code to unpack our normal map:

```
o.Normal = UnpackNormal (tex2D (_BumpMap, IN.uv_MainTex));
```

19. Save the shader.
20. Minimize MonoDevelop and return to the main Unity interface.
21. In the **Hierarchy** panel, select the `astronaut` game object.
22. In the **Inspector**, modify the three materials to use the new shader.
23. Select the **Alien** game object. Scroll to the bottom of the **Inspector** to locate its material.
24. Replace the material's shader with the `Mobile - Character` shader by selecting it from the dropdown list.

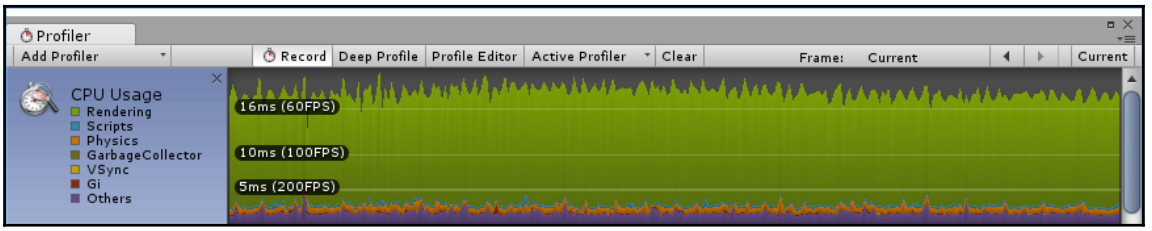
Open the **Build Settings** window again and rebuild the app on the device.

Their should be an improvement in the appearance of the alien and astronaut models on the device, as seen in the following illustration:



Android build with improved character shaders

If you check the **Profiler** while testing the game, you will see a noticeable increase in the frame rate:



The optimized build in the Profiler showing the connected Android device

The improvement in the performance will vary, depending on the device. Crucially, reducing the use of real-time lights, the number of shaders, and the complexity of the light calculations will make a major difference on mobile.

Summary

In this chapter, we explored some optimization techniques in builds for mobile devices.

We summarized the build process for iOS and Android and tested our initial scene.

We then wrote a custom mobile shader to demonstrate the effect on the frame rate.

We also wrote a slightly more advanced shader to handle the scene's most important elements: the astronaut's character and her alien adversary.

We completed our objectives for the chapter with a few more tips and tricks to get the mobile build running more smoothly.

Index

S

T

V

W