3
Maps and Materials

In this chapter, we will cover:

- Creating a reflective material
- Creating a review mirror (Pro only)
- Creating a self-illuminated material
- Creating specular texture maps
- Creating transparency texture maps
- Using cookie textures to simulate a cloudy outdoor
- Creating a color selection dialog
- Combining textures in real time through the GUI
- Highlighting materials at mouse over
- Fading the transparency of a material
- Playing videos inside a scene (Pro only)
- Using a video file as an intro or cut-scene (Pro only)
- Animated textures by looping through array of materials materials (e.g. simulated video)
- Disabling culling for a material

Introduction

Most materials are created with the help of texture maps. To create those maps, which are actually bitmap files (regardless of their file extension), it is necessary to use a image editor such as Adobe Photoshop (which is the industry standard and has its native format supported by Unity), GIMP, etc. In order to follow recipes on this chapter, it's strongly recommended that you have access to software like that.

When saving texture maps, specially the ones that have an Alpha channel, you might want to choose an adequate file format. PSD, Photoshop's native format, is practical for preserving the original artwork in many layers. TGA is also a good option for preserving the alpha channel -- although the image will be flattened.

Finally, before you start, it might a good idea to read Unity's documentation on textures. It can be found online at `http://unity3d.com/support/documentation/Manual/Textures.html.`

Creating a reflective material

Metal, and car paint and glossy plastic surfaces are just some very recurrent examples of materials that need reflectiveness. Luckily for us, Unity includes reflective shaders that, when properly configured, can help us to achieve that look we're after.

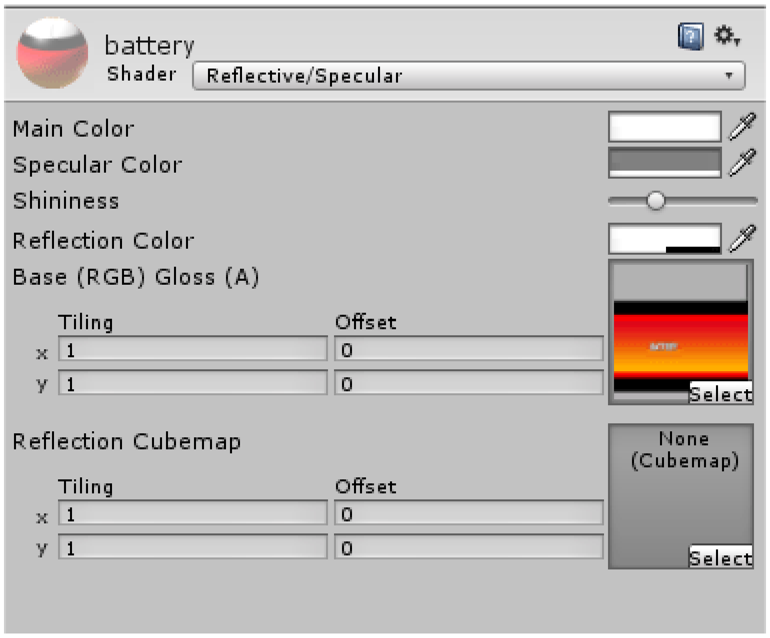
Getting ready

For this recipe, we will prepare two texture maps: the Base map and the Reflection Cubemap. The Base map will be a RGBA image, where the alpha channel will assign the reflection level from completely opaque (black) to completely reflective (white). The Reflection Cubemap can be made from either six or, as shown in this recipe, a single texture.

To help us focus on this recipe's main subject, which is setting up a reflective material, a Unity package containing a previously made scene, and also a .jpg file to be used as reflection map, are available inside the folder 03_01.

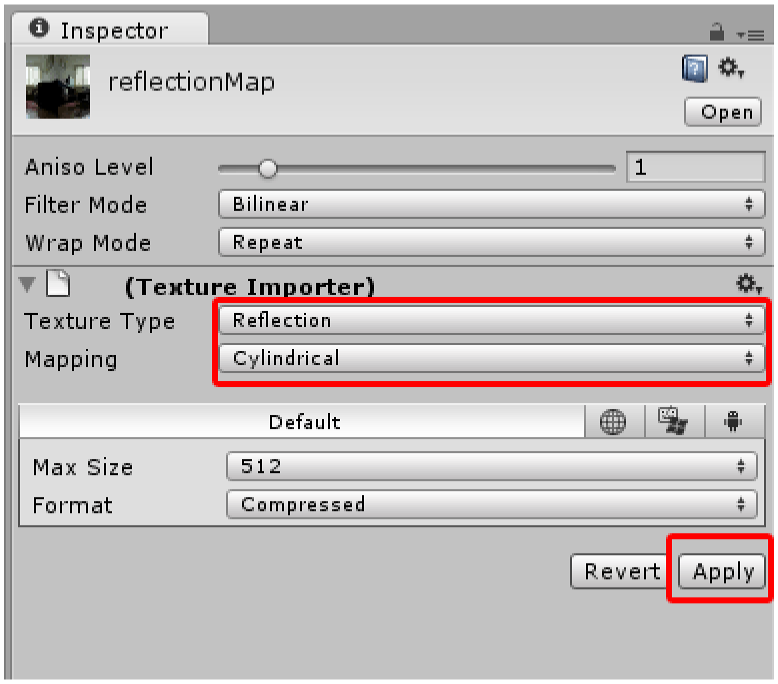
How to do it...

1 - Import the Unity Package *reflectiveMaterial.unitypackage*, inside the folder 03_01.
2 - Open the level *reflectiveMaterialLevel*. There will be a single 3d object (a battery) that rotates around its own axis when the level is played. We can also orbit the camera around it.
3 - Stop the level from running and, in the *Hierarchy* window, select the battery. The *Inspector* window will display its material. Use the drop-down menu to change the Shader from Diffuse to Reflective / Specular.



0423_03_01.png

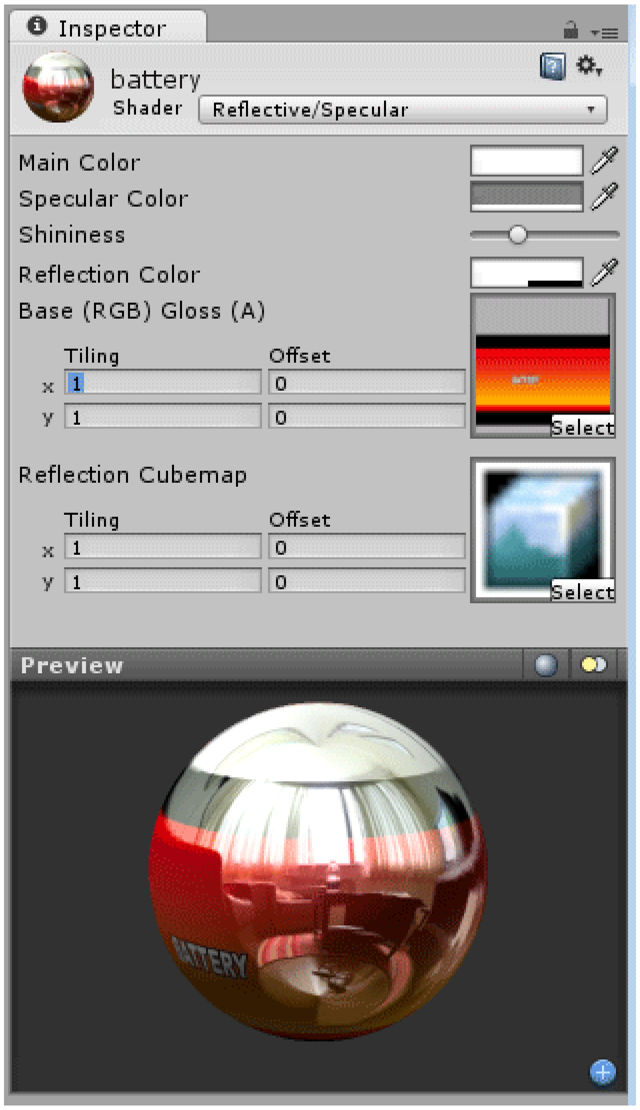
4 - Now we must build our Reflection Cubemap. In the Project window, select the *reflectionMap* texture. The Texture Importer settings will be displayed in the *Inspector* window. Using the Texture Type drop-down menu, choose *Reflection*. Then, change the Mapping option to *Cylindrical*. Finally, click *Apply* to apply the changes we've made.



Aniso Level 1
Filter Mode Bilinear
Wrap Mode Repeat
(Texture Importer)
Texture Type Reflection
Mapping Cylindrical
Max Size 512
Format Compressed

0423_03_02.png

5 - Select the *reflectionMap* asset in your *Project* window. It has changed from a 2D texture to a Cubemap (as you can see from its icon).
6 - Select the battery material and apply the *reflectionMap* into the Reflection Cubemap slot.


Inspector

battery

Shader: Reflective/Specular

Main Color

Specular Color

Shininess

Reflection Color

Base (RGB) Gloss (A)

	Tiling	Offset
x	1	0
y	1	0

Reflection Cubemap

	Tiling	Offset
x	1	0
y	1	0

0423_03_03.png

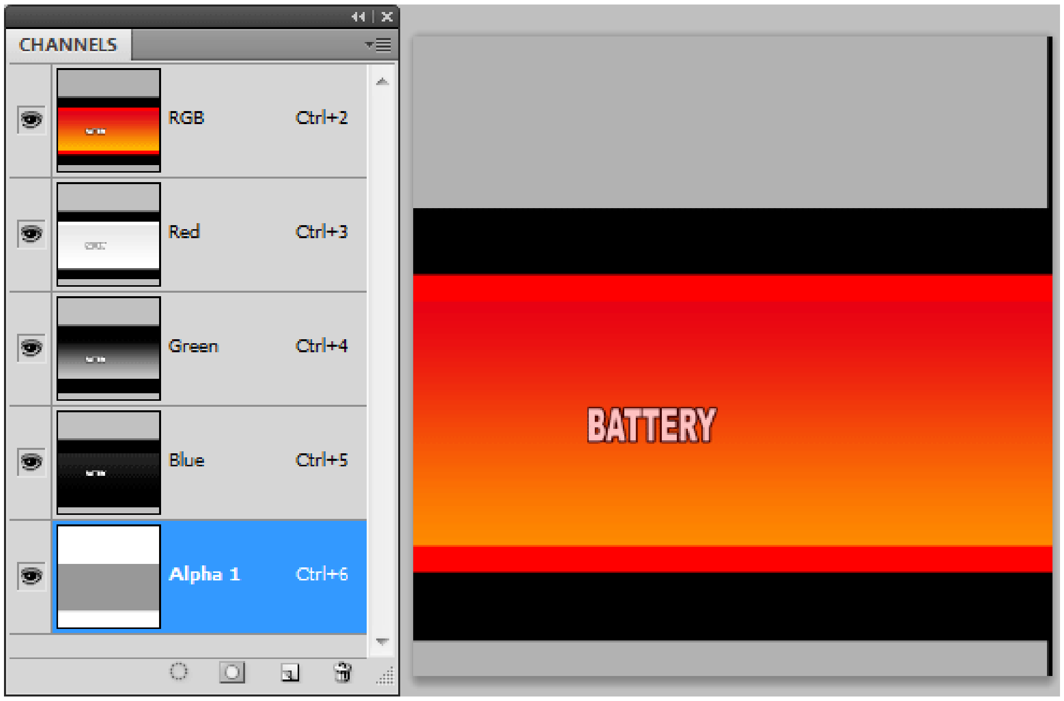
7 - Play the level. The reflection effect is visible, but it could be smoother in some areas of the texture.

8 - Open the base texture (named *batteryBase*) in your image editor.

9 - In your image editor, make all channels visible and select the Alpha channel window (in Photoshop this can be done through the menu *Window* > *Channels*).

10 - The Alpha channel is completely white. Paint a grey rectangle in the region

correspondent to the battery's label. In this example, we have used a grey color of Red, Green and Blue values of 152.



0423_03_04.png

11 - Save the file to make the changes and wait Unity to reload the image. Reflection should now be more discreet inside the label area.

How it works...

The base texture map's alpha channel sets the reflectiveness and, in this example, the specular level of the material according to each pixel's brightness. That means the Reflection Cubemap texture will be more visible where the alpha channel is white.

The Reflection Cubemap is actually made from six different images, which where generated from out original image file by the Texture Importer when we changed the Texture Type to *Reflection*. These six images are mapped as top, bottom, front, back, left and right.

There's more...

Reflection Cubemaps can be done in many ways and have different mapping properties.

Using six different texture maps

Instead of using a single texture map, you could also create an empty Reflection Cubemap in the *Project* window and assign six different image files illustrating top, bottom, front, back, left and right of your environment.

Playing with mapping options

The cylindrical mapping we used was the best choice for the battery model. Experiment other settings for different shapes.

Shader settings

Besides the Base and Reflection texture maps, you can also experiment with the other shader settings such as the Main, Specular and Reflection Colors (that can be used to tint the material) and also Shininess (which controls the specular dispersion). ,

Reflecting actual scene geometry

You could make a reflection map from the actual level geometry. In fact, it could either reflect your level in real time or bake a single frame into the Reflection Cubemap. Unity's documentation has all the instructions you need, including the complete script code, at
`http://unity3d.com/support/documentation/ScriptReference/Camera.RenderToCubemap.html`

See also

Creating a review mirror (Pro only)

Creating a review mirror (Pro only)

In this recipe, we will take advantage of Unity Pro ability to render camera views into textures to make a review mirror.

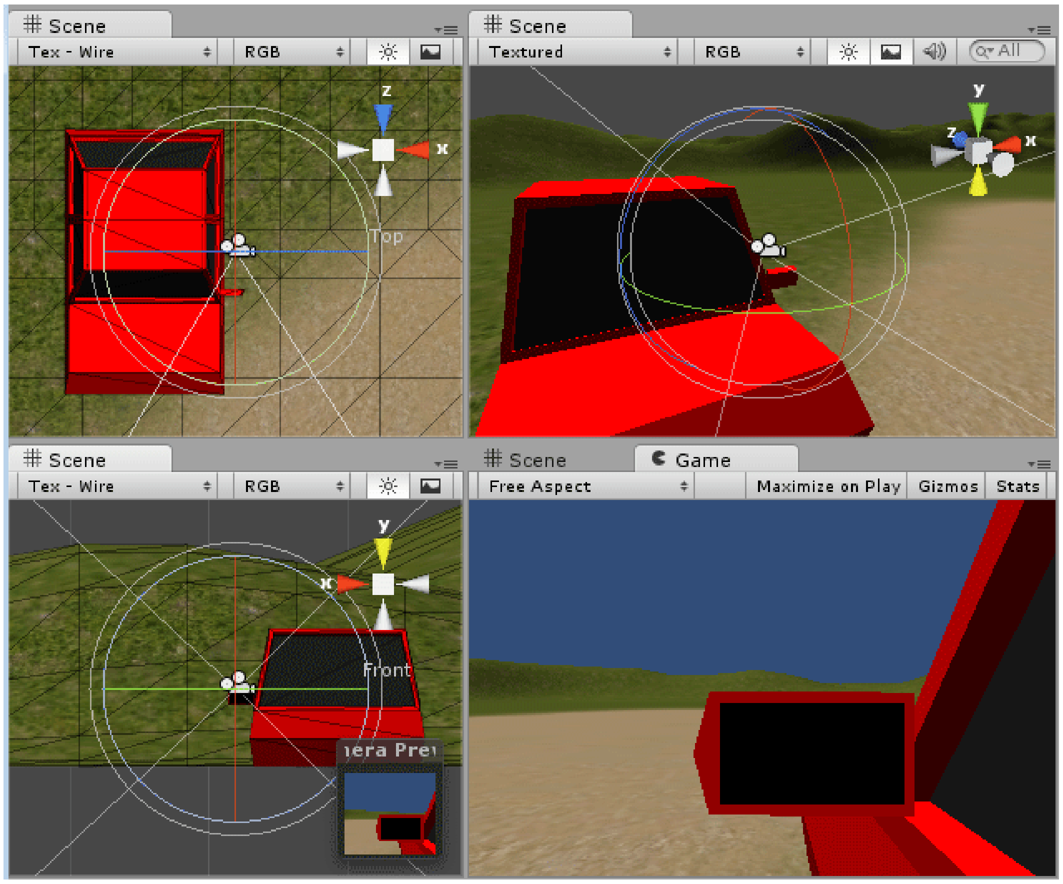
Getting ready

In order to follow this recipe, please use the example scene *mirrorPackage*, available in Unity Package format, located at the folder 03_02.

How to do it...

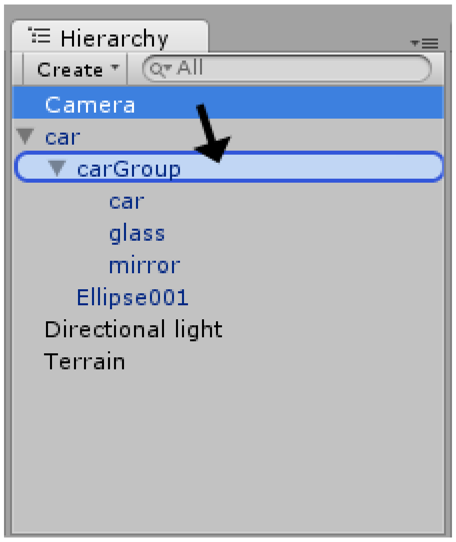
1 - Import the *mirrorPackage* and open the scene named *mirrorScene*.

2 - Using the transformation tools (Move and Rotate), place the main camera next to the left side of the car, as shown in the illustration below. Make sure the review mirror is framed.



0423_03_05.png

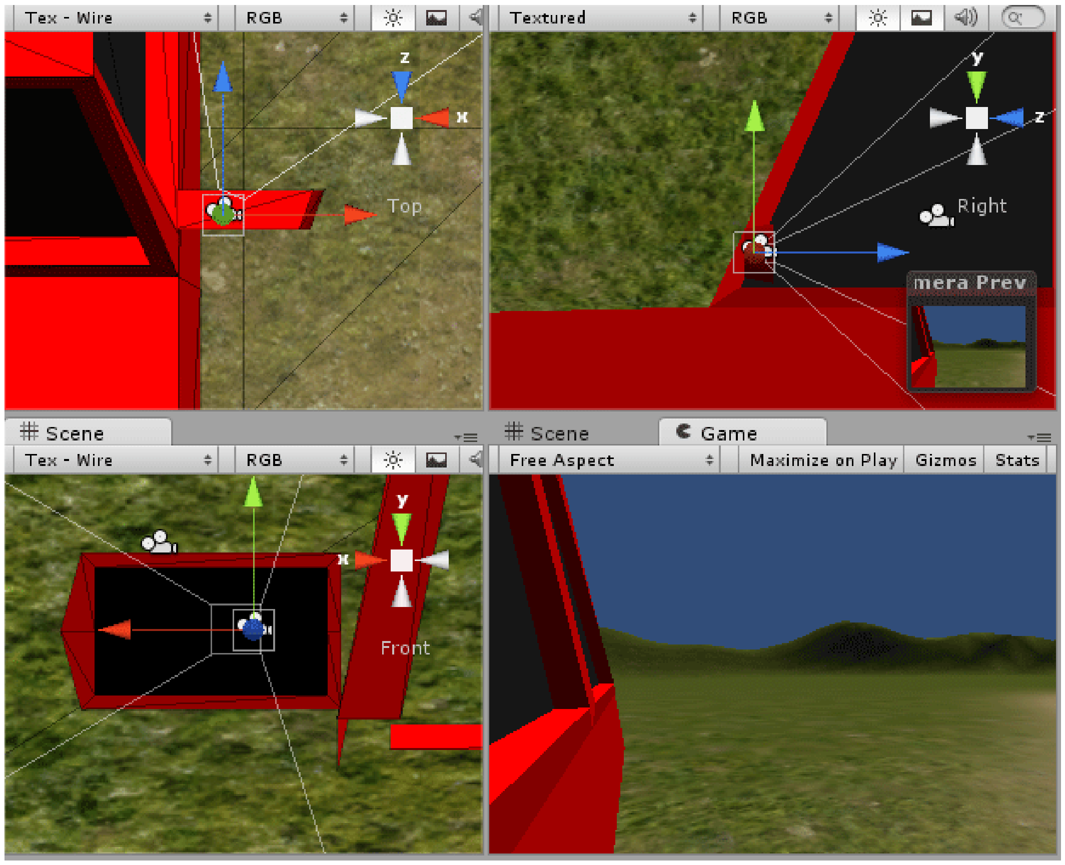
3 - In the *Hierarchy* window, expand the *car* object. Then, drag the *Camera* into the *carGroup* sub-object, making it a children of the *carGroup*.



0423_03_06.png

4 - In the *Hierarchy* window, access the *Create* dropdown menu to add a Camera to the scene. Rename it to *mirrorCam* and, in the *Inspector* window, disable its *Audio Listener*.

5 - Position and rotate the camera as if it is inside the review mirror object.



0423_03_07.png

6 - Expand the *car* object and then the *carGroup* sub-object. Make the *mirrorCam* a child of the *mirror* sub-object (within the *carGroup*).

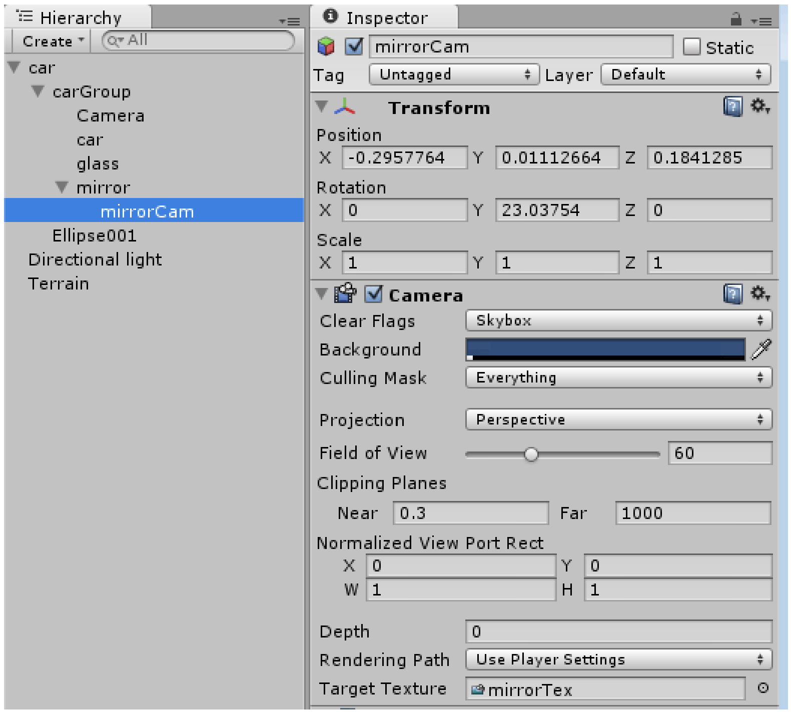
7 - Now, let's create a new material for the mirror. In the *Project* window, enter the *Create* dropdown menu and choose *Material*. Name it *mirrorMaterial*. Next, enter the *Create* dropdown menu one more time and create a Render Texture. Rename it *mirrorTex*.

8 - In the *Inspector* window, set the *mirrorText* size 1024 x 512.

9 - Select the *mirrorCam* object. In the Inspector window, change the Target Texture field to *mirrorTex*.

10 - Select the *mirrorMaterial* and change its Main Color texture to *mirrorTex*. This can be done by dragging the *mirrorTex* material from the *Project* window into the Main Color texture slot.

11 - In the *Hierarchy* window, select the *mirrorCam* sub-object. Then, apply the *mirrorMaterial* by dragging it from the *Project* window into the *Inspector* window (below its current material).



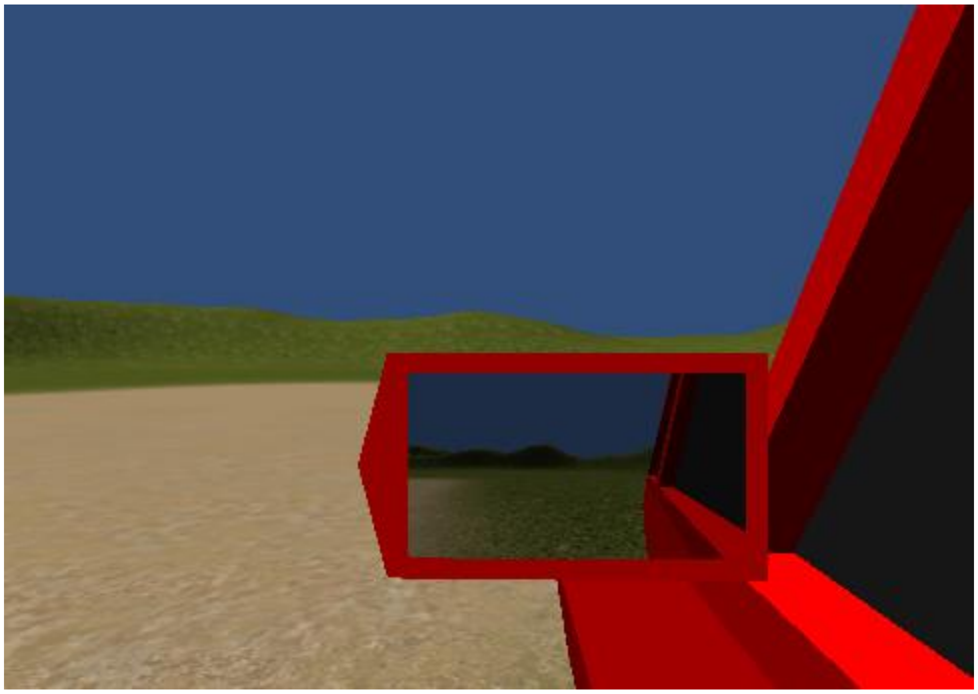
```
Hierarchy
Create ▾  All
▼ car
  ▼ carGroup
      Camera
      car
      glass
    ▼ mirror
        mirrorCam
    Ellipse001
  Directional light
  Terrain
```

```
Inspector
☑ mirrorCam                                   ☐ Static
Tag  Untagged                 Layer  Default
▼ Transform
Position
X -0.2957764     Y 0.01112664     Z 0.1841285
Rotation
X 0              Y 23.03754       Z 0
Scale
X 1              Y 1              Z 1
▼ ☑ Camera
Clear Flags          Skybox
Background
Culling Mask         Everything

Projection           Perspective
Field of View                         60
Clipping Planes
   Near  0.3                 Far  1000
Normalized View Port Rect
   X  0                      Y  0
   W  1                      H  1

Depth                0
Rendering Path       Use Player Settings
Target Texture       mirrorTex
```

0423_03_08.png

12 –Now, the map must be flipped horizontally (more precisely, mirrored). In the *Project* window, select the *mirrorTex* and change its Wrap Mode to *Repeat*.

13 - Select the *mirrorMaterial* and change its “x” Tiling value to -1.



0423_03_09.png

How it works...

As you have learned from the recipe, the mirror's camera view is actually being rendered into the render texture in real time.

There's more...

There is a lot of interesting things to be done with Render to Texture:

Adding Detail

Also, you could make your mirror material more interesting by changing its shader from *Diffuse* to *Decal* and adding a RGBA texture map to it, where the alpha channel would control the opaqueness of the Decal texture. That's a great way of adding a layer of dust or stain to the mirror.

You don't have to mirror it

You could easily adapt this recipe to make a TV or monitor screen texture. The only major change would be to not changing the 'x' tiling value to -1.

See also

Creating a reflective material

Creating a self-illuminated material

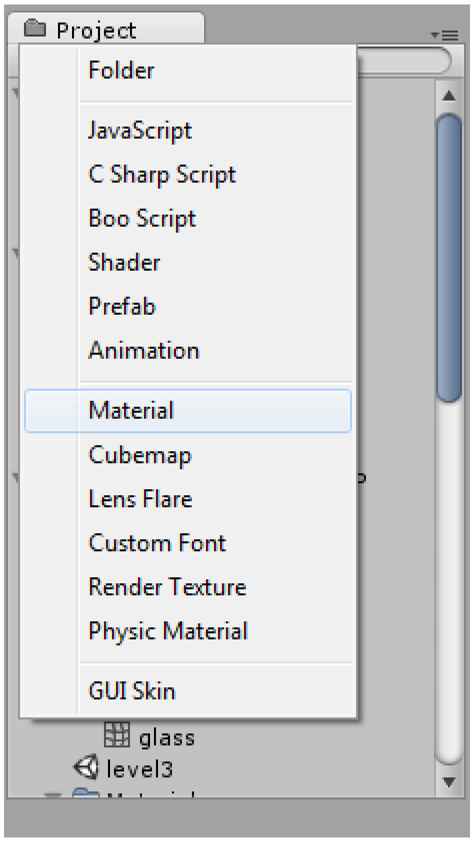
Self-illuminated materials can be used to simulate a variety of objects, from led mobile displays to futuristic Tron suits. In this recipe, we will learn how to configure this kind of material and its texture maps.

Getting ready

As we will create a LCD display in this example, please make sure you have a font in that style installed in your computer. If not, you can find several free lcd fonts in specialized websites such as `www.dafont.com`.

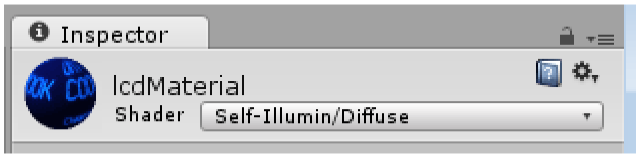
How to do it...

1 - Create a new material. The easiest way to do that is to access the *Project* window, click the *Create* drop-down menu and choose *Material*.



0423_03_10.png

2 - Rename your new material. For this example, let's call it *LCDMaterial*.
3 - Select your material. In the Inspector window, under the material's name, use the drop-down menu to change its shader *to Self-Illumin/Diffuse*.



lcdMaterial
Shader Self-Illumin/Diffuse

0423_03_11.png

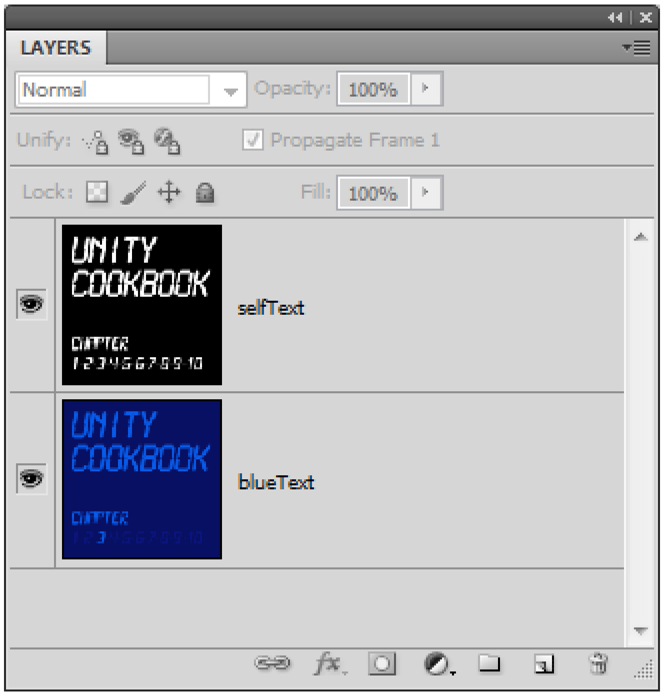
4 - Open your image editor (we'll use Adobe Photoshop to illustrate the next steps).
5 - Create a new image. Let's make it 256 pixel both wide and tall, at 72 dpi (RGB mode).
6 - Create a new layer and fill it with a dark blue color (for instance, R:8, G:16, B:99). Name it *blueBg*.

7 - Create a type layer and write your LCD text in light blue (R:8 G:90 B:231). Name it *blueText*.

8 - Duplicate the layers *blueBg* and *blueText*. Rename them to *selfBg* and *selfText*, respectively.

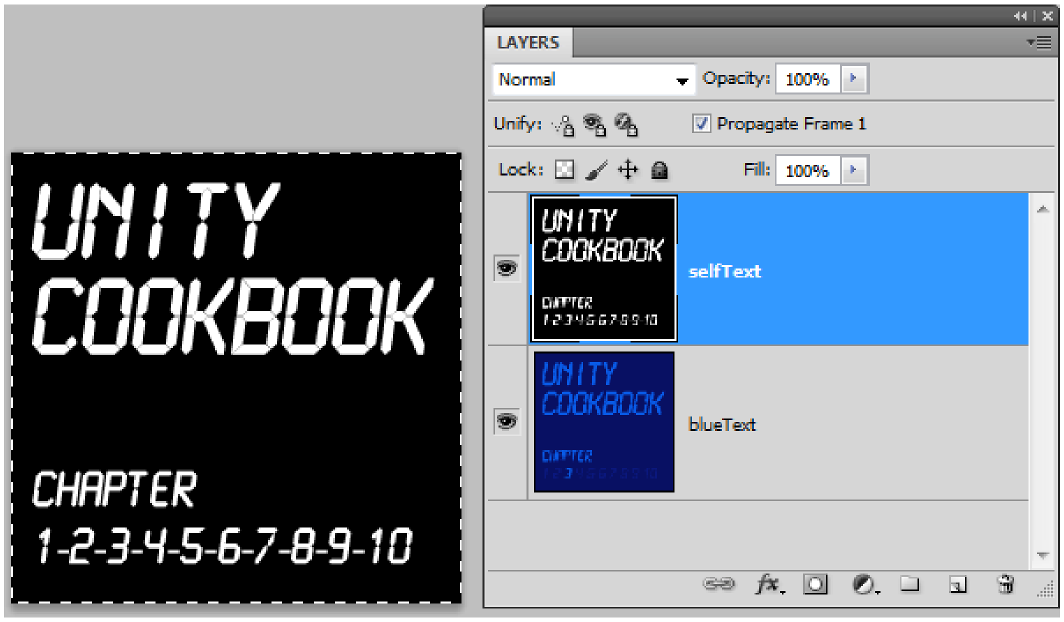
9 - Change the color of the *selfText* type to white. Then, fill the layer *selfBg* in black.

10 - Merge the layers *selfText* and *selfBg*. Then, merge the layers *blueText* and *blueBg*.



0423_03_12.png

11 - Make a selection of the entire black and white layer (Ctrl + A on Windows, Command + A on Mac OS). Then, copy it (using Ctrl + C or Command + C).

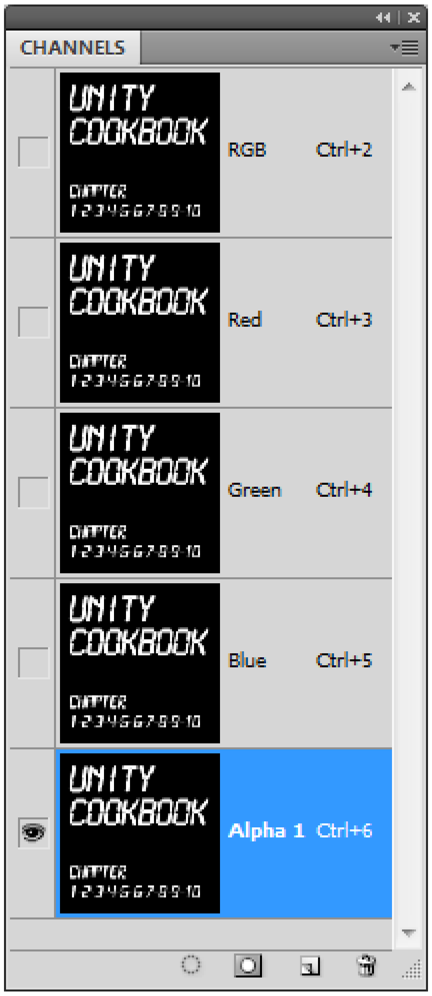


0423_03_13.png

12 - Open the *Channels* window window (in Photoshop this can be done through the menu *Window > Channels*).

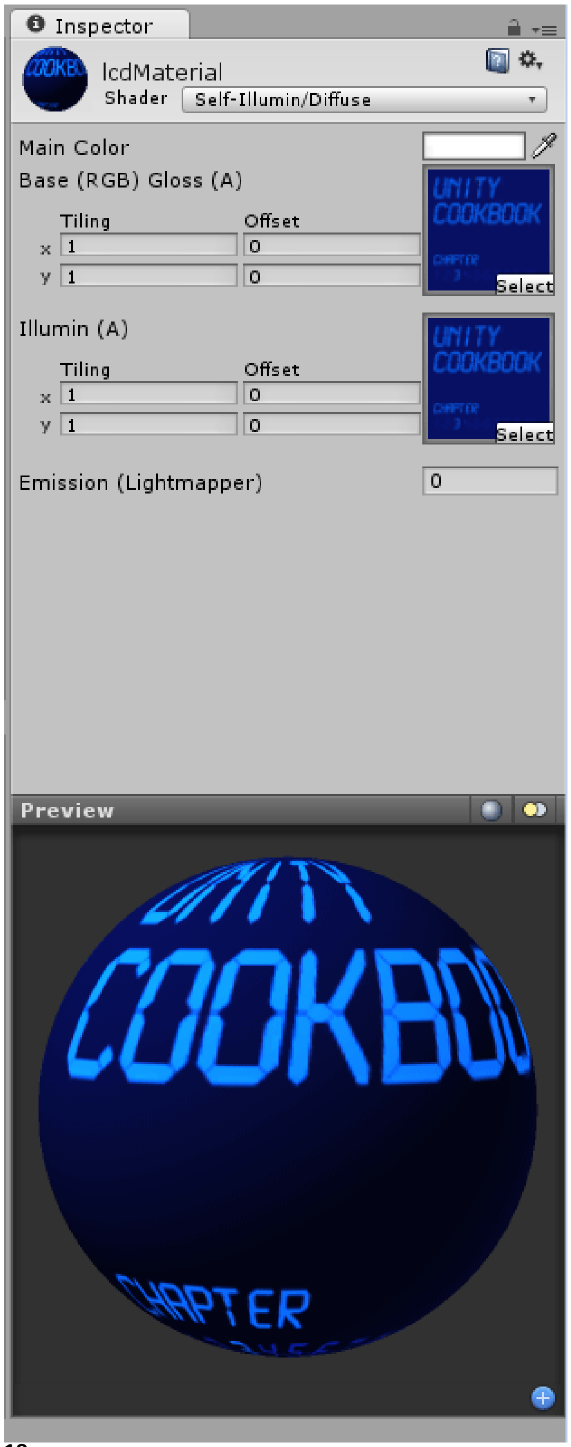
13 - There should be three channels: Red, Green and Blue. Create a new channel. That will be the Alpha channel that controls the level of self-illumination and gloss.

14 - Paste (Ctrl + V or Command + V) your black and white layer into the Alpha channel.



0423_03_14.png

15 - In the *Layers* window, hide or delete the merged black and white layer.
16 - Save your file (.PSD or .TGA formats are good options, as they keep the alpha channel).
17 - Inside the Unity Editor, import your image file access through the *Assets* menu, clicking on *Import New Asset...*
18 - In the *Project* window, choose the *LCDMaterial*. Then, select your image file as both *Base* and *Illumin* maps (by clicking on the *Select* button or dragging them from the *Project* window to the material slots). Your self-illuminated material is ready.


Inspector

lcdMaterial

Shader: Self-Illumin/Diffuse

Main Color

Base (RGB) Gloss (A)

	Tiling	Offset
x	1	0
y	1	0

Illumin (A)

	Tiling	Offset
x	1	0
y	1	0

Emission (Lightmapper): 0

0423_03_15.png

How it works...

Unity is able to read four channels of a texture map: R (red), G (green), B (blue) and A (alpha). Self-illuminated shaders use RGB channels as the base texture (often referred to as the “diffuse texture”), while using the Alpha to illuminate the material according to each pixel’s britghtness level.

There's more...

Reflection Cubemaps can be done in many ways and have different mapping properties.

Independent maps

Please note that you can use two different image files as the Base and Illumination maps. In this case, the Base texture’s alpha channel will be control the material’s gloss.

Emitting light over other objects

The *Emission (Lightmapper)* field in the Material options can be used to simulate the material’s light projection over other objects when baking a lightmap.

See also

Creating specular texture maps
Creating transparency texture maps

Creating specular texture maps

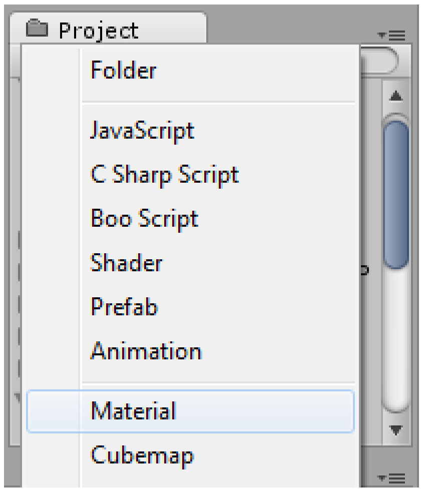
Some surfaces can have both glossy and matte areas. In order to achieve that effect, we can use specular maps.

Getting ready

For illustration purposes, we will create a rusted metal material to demonstrate how the specular property can be used to enhance realism. If you don’t have a base texture to develop into a specular material, please use the one included in the book’s content folder 03_04.

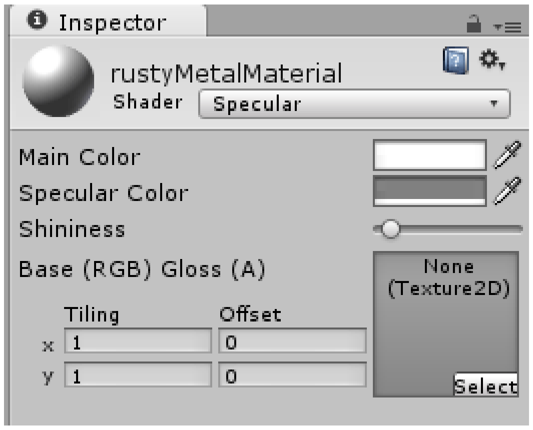
How to do it...

1 - Create a new material. The easiest way to do that is to access the *Project* window, click the *Create* drop-down menu and choose *Material*.



0423_03_16.png

2 - Rename your new material. For this example, let's call it *rustyMetalMaterial*
3 - Select your material. In the *Inspector* window, under the material's name, use the drop-down menu to change its shader to *Specular*.

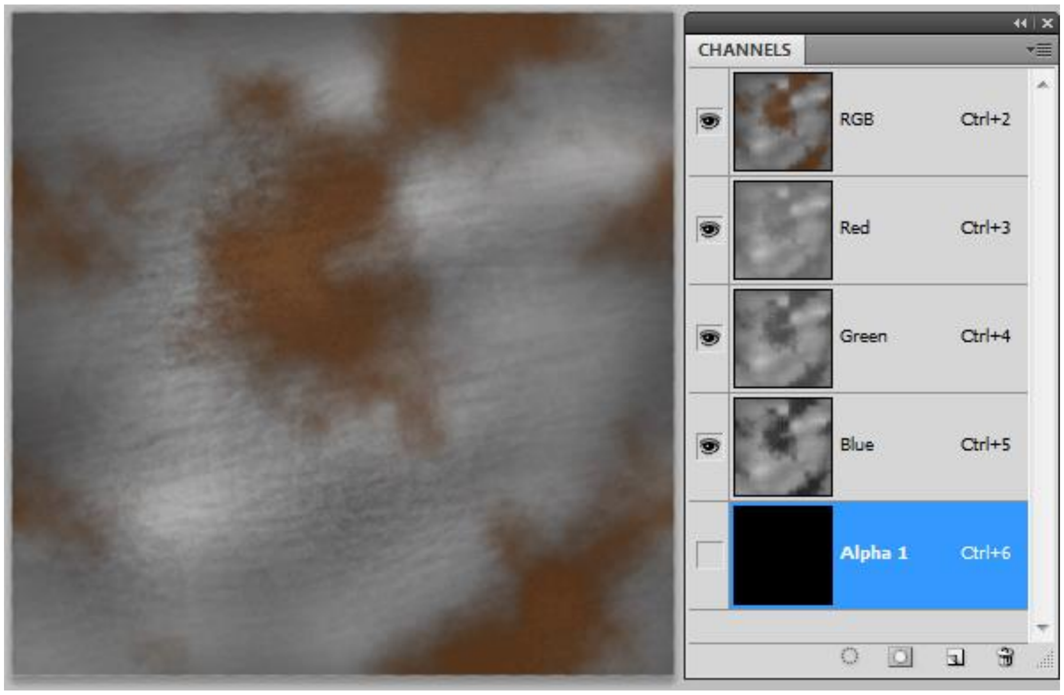


0423_03_17.png

4 - Open the base texture in your image editor (we'll use Adobe Photoshop to illustrate the next steps).

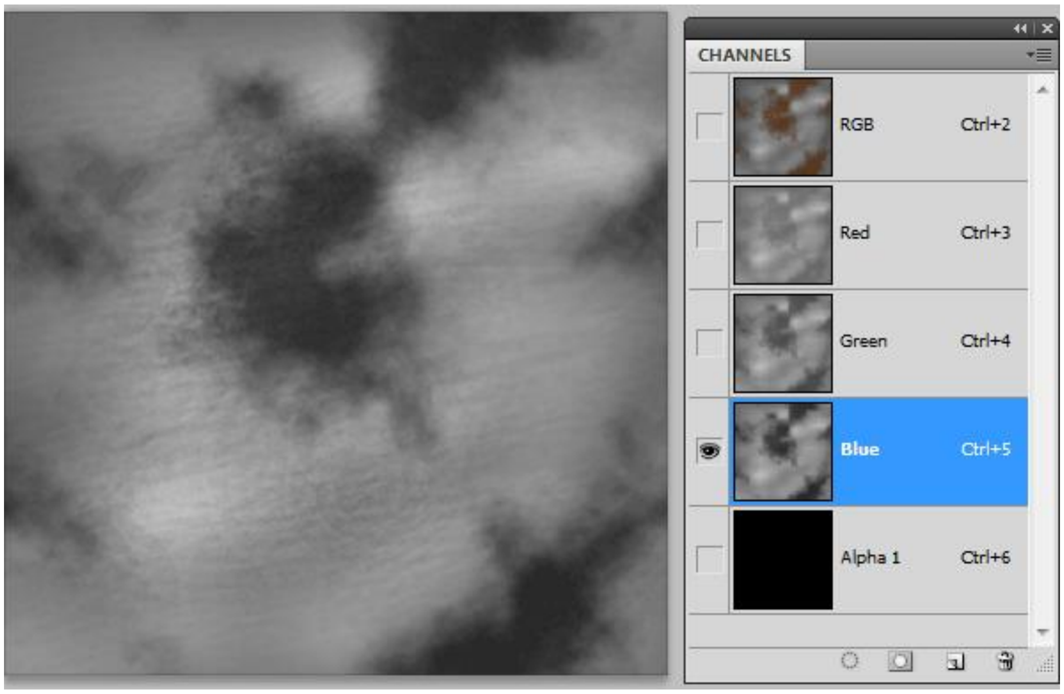
5 - Open the *Channels* window window (in Photoshop this can be done through the menu *Window > Channels*).

6 - There should be three channels: Red, Green and Blue. Create a new channel. That will be the Alpha channel that controls the gloss level of our specular material.



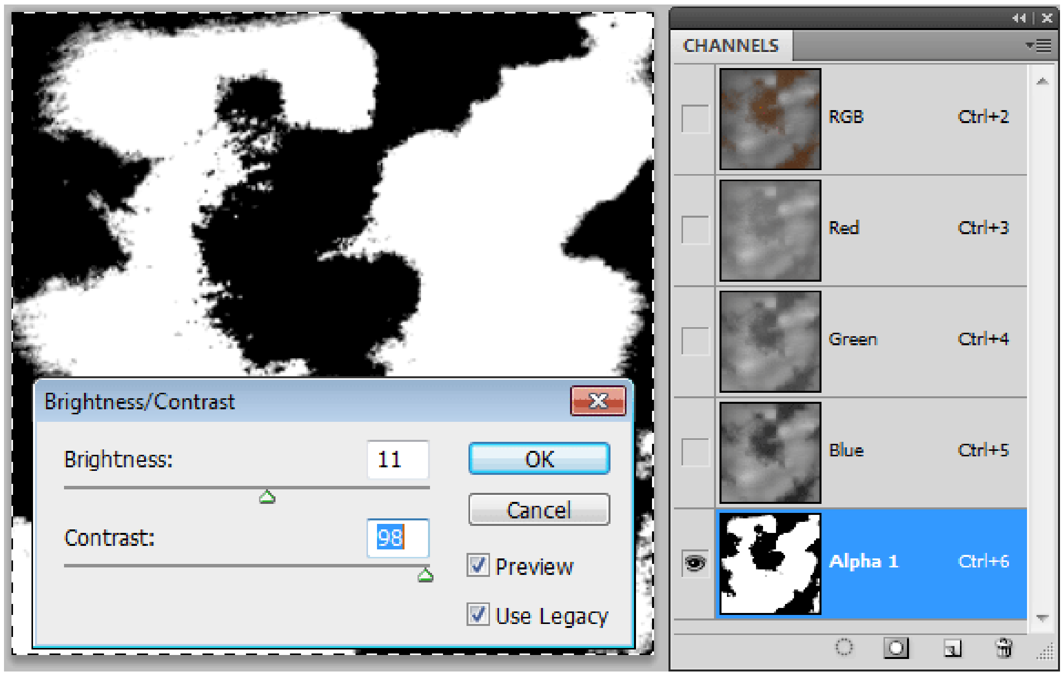
0423_03_18.png

7 - We want our Alpha channel to emphasize the contrast between rust and metal. Since *Blue* is the channel with higher level of contrast, we will select it on the *Channels* window and turn all the other channels invisible. The resulting image will be displayed in grayscale.



0423_03_19.png

8 - Select and copy all the image pixels (using Ctrl + A / Ctrl + C on Windows, or Command + A / Command C on Mac OS).
9 - In the *Channels* window, select the Alpha channel and paste the image you have copied.
10 - Now, intensify the contrast by accessing the main menu *option Image > Adjustments > Brightness/Contrast*.


Brightness/Contrast

Brightness: 11

Contrast: 98

[✓ Preview]

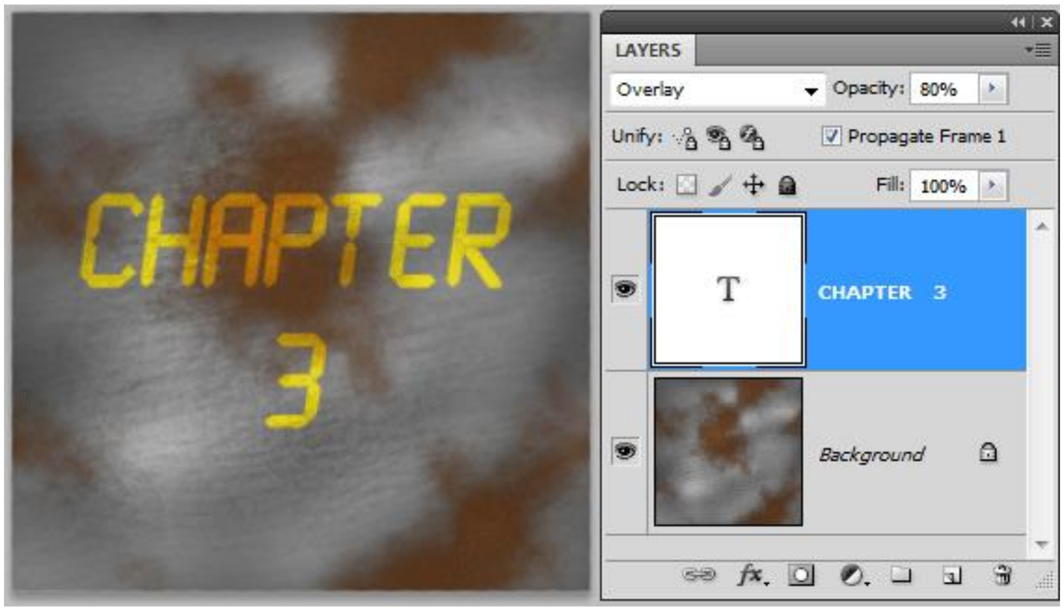
[✓ Use Legacy]

0423_03_20.png

11 - Still in the *Channels* window, bring the colors back by enabling all RGB channels.

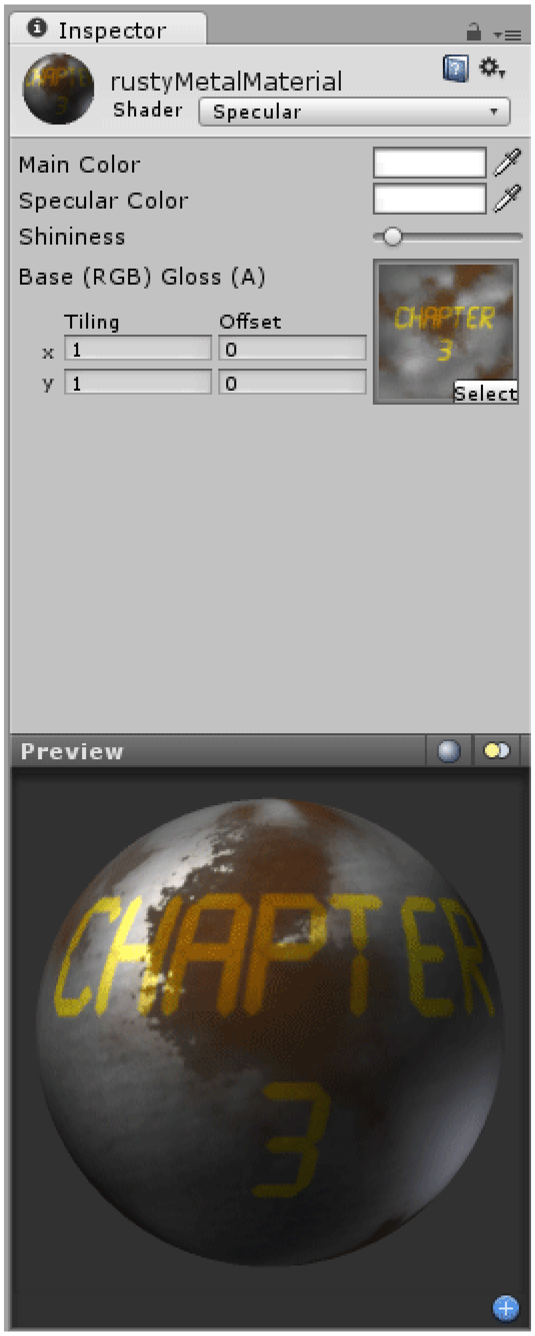
12 - In addition to the rusty metal plate, let's simulate a paint layer on the wall: create a type layer in yellow. You can type whatever you want.

13 - Change the type layer's blending mode to *Overlay* and its opacity to 80%.



0423_03_21.png

14 - Flatten your image (this is necessary, or Unity might discard the alpha channel you created and use the image layer transparent pixels as the alpha channel instead).
15 - Save your work as a new image file (.PSD or .TGA formats are good options, as they keep the alpha channel).
16 - Inside the Unity editor, import your image file access through the *Assets* menu, clicking on *Import New Asset...*
17 - In the *Project* window, choose the *rustyMetalMaterial*. Then, apply the texture you've made as Base map (by either clicking on the *Select* button or dragging it from the *Project* window to the material slots). Your specular mapped material is ready.



Inspector

rustyMetalMaterial

Shader: Specular

Main Color

Specular Color

Shininess

Base (RGB) Gloss (A)

	Tiling	Offset
x	1	0
y	1	0

0423_03_22.png

How it works...

Unity is able to read four channels of a texture map: R (red), G (green), B (blue) and A (alpha). Specular shaders use RGB channels as the base texture (often called "diffuse texture), while using the Alpha to set the specular brightness of the material according to each pixel's brightness level.

There's more...

The specular shader features some other parameters that are worth looking into: Specular Color and Shininess.

Specular Color

If you ever get tired of that white shining spot, you can add some variety by changing the Specular color.

Shininess

Shining spots don't look the same on bowling balls (where they are highly concentrated) as they do on brushed metal surfaces (where they spread a bit more). This concentration level is controlled by the Shininess parameter.

See also

Creating transparency texture maps

Creating transparency texture maps

If you want your player to partially see through an object, you'll need a transparent or semi-transparent material. Plastic film, cutouts, grids are some of the artifacts you can produce with transparent texture maps.

Getting ready

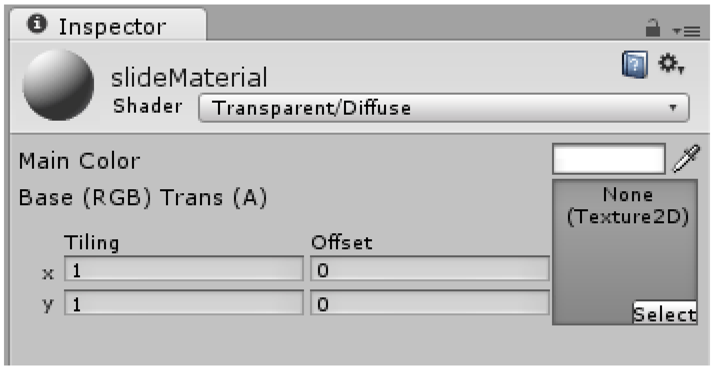
For this recipe, we will create a material simulating a slide film. The main reason behind this choice is that a material like that allow us to use full transparency to make rounded borders of the frame, and also use semi-transparency to illustrate the film itself. For the rest of this recipe, it is assumed you are capable of creating, using your image editor, an image like the one intended (basically a beveled round-cornered rectangle containing a picture within). If not, please feel free to use the one included inside the folder 03_05.

How to do it...

1 - Create a new material. The easiest way to do that is to access the *Project* window, click the *Create* drop-down menu and choose *Material*.

2 - Rename your new material. For this example, let's call it *slideMaterial*

3 - Select your material. In the *Inspector* window, under the material's name, use the drop-down menu to change its shader to *Transparent / Diffuse*.

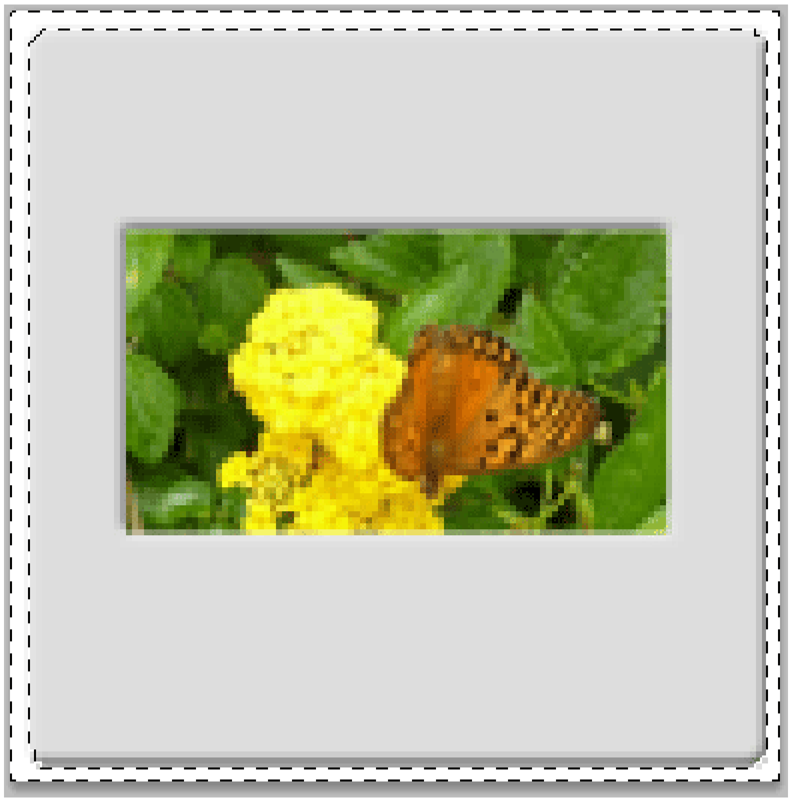


Shader Transparent/Diffuse
Main Color
Base (RGB) Trans (A) None (Texture2D)
Tiling x 1 y 1
Offset x 0 y 0

0423_03_23.png

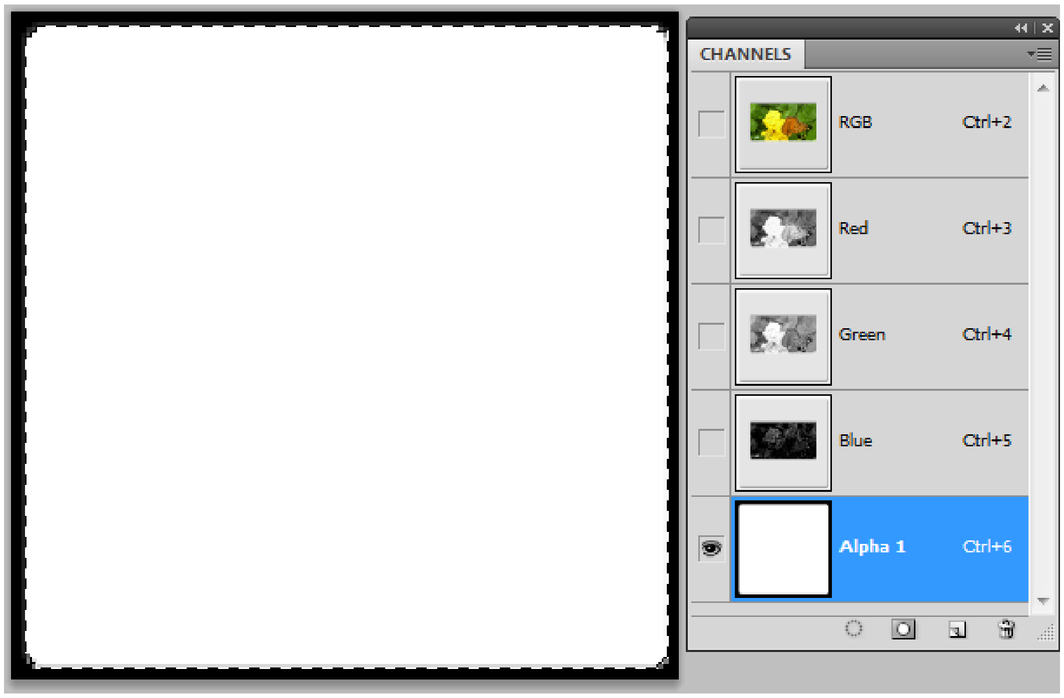
4 - Open the base texture in your image editor (we'll use Adobe Photoshop to illustrate the next steps).

5 - Select the empty pixels around the rounded rectangle (this can be done with the *Magic Wand* tool with Anti-alias turned on and a Tolerance level of 0).



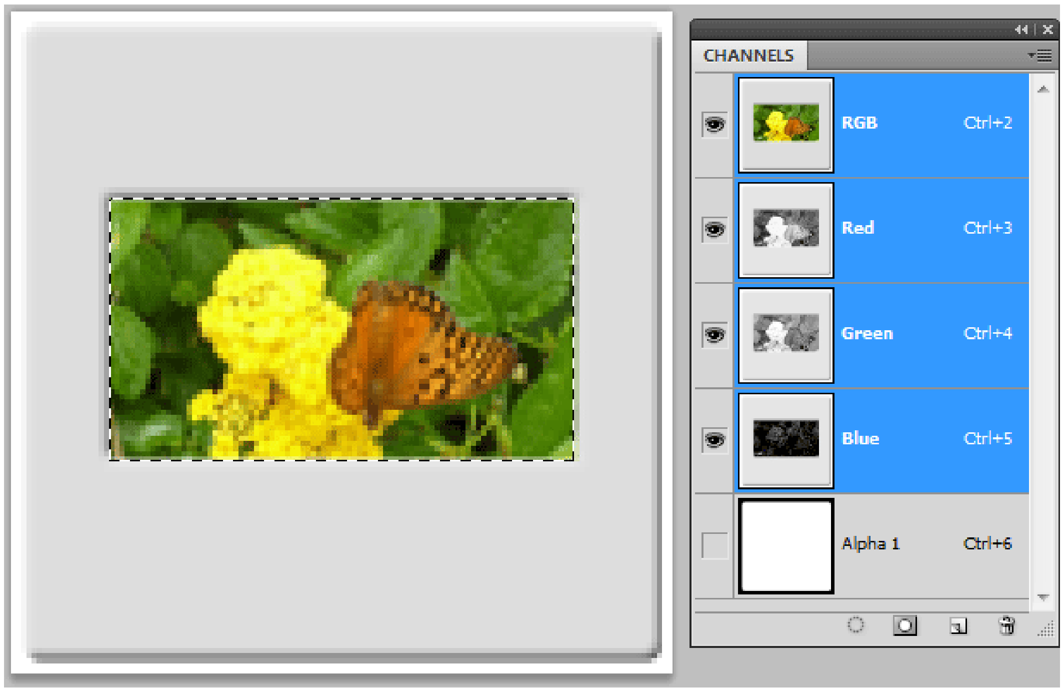
0423_03_24.png

6 - Open the *Channels* window window (in Photoshop this can be done through the menu *Window > Channels*).
7- There should be three channels: Red, Green and Blue. Create a new channel. That will be the Alpha channel that controls the level of transparency, where darker levels are more transparent than brighter levels.
8 - In the *Channels* window, select the Alpha channel and invert your selection (CTRL + SHIFT + I if using Windows, Command + Shift + I if using Mac OS).
9 - Using the *Paint Bucket* tool, paint the selection area white.



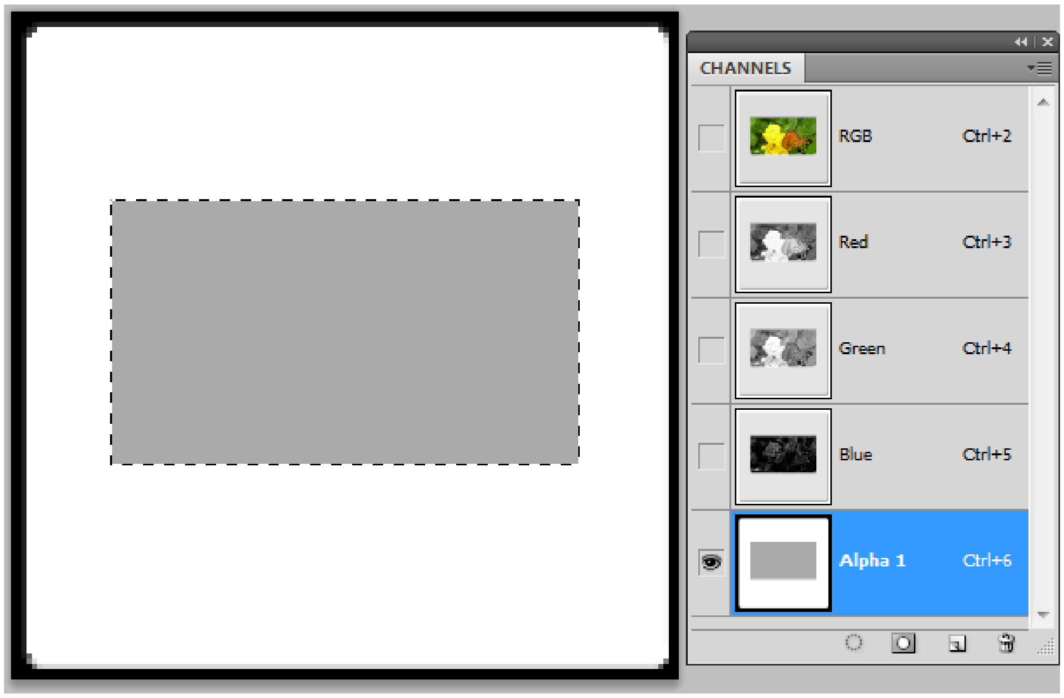
0423_03_25.png

10 - Reset your selection area (using CTRL + D / Command + D). Now, create a rectangular selection around the actual photo.



0423_03_26.png

11 - Go back to the Alpha channel and fill the selected rectangle light gray (R, G and B value around 170, for instance).



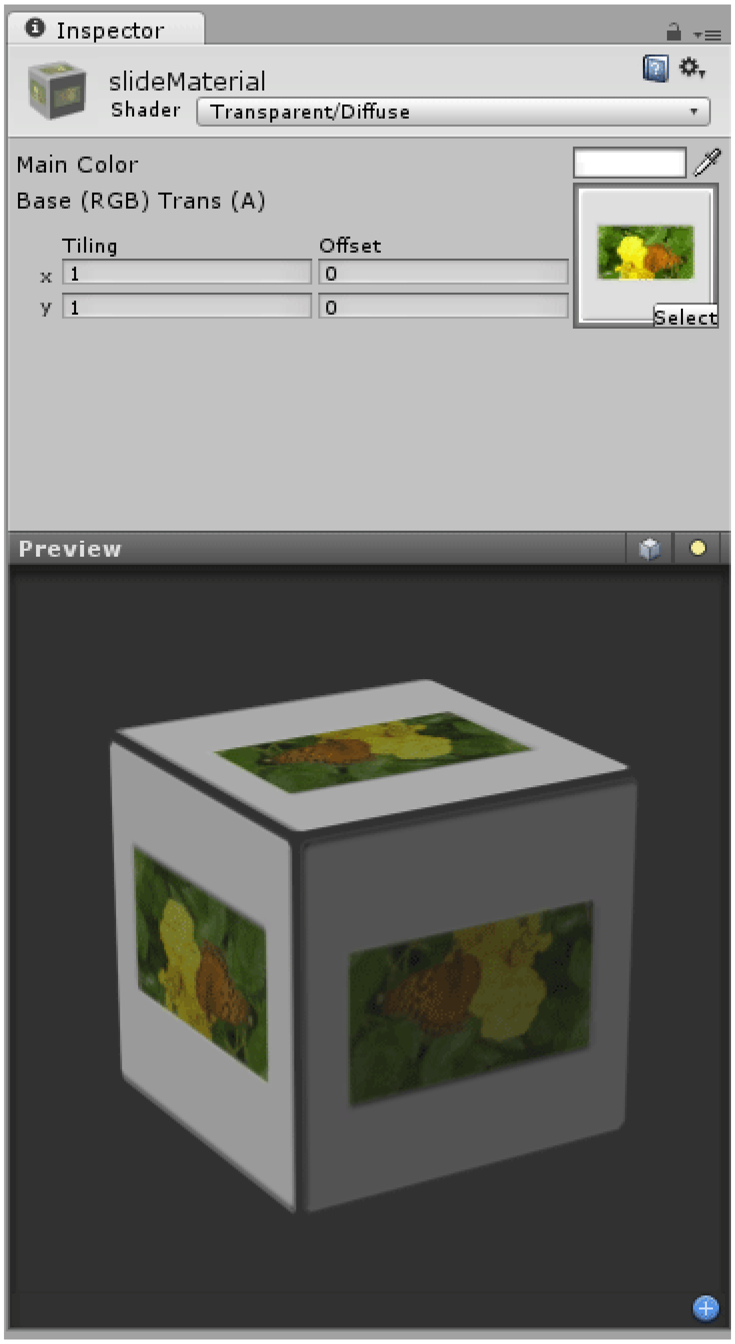
0423_03_27.png

12 - Flatten your image (this is necessary, or Unity might discard the alpha channel you created and use the image layer transparent pixels as the alpha channel instead).

13 - Save your file (.PSD or .TGA formats are good options, as they keep the alpha channel).

14 - Inside the Unity editor, import your image file access through the menu *Assets > Import New Asset...*

15 - In the *Project* window, choose the *slideMaterial*. Then, select the image file you created as Base map (by either clicking on the *Select* button or dragging them from the *Project* window to the material slots). Your transparent material is ready.



Inspector

slideMaterial
Shader Transparent/Diffuse

Main Color
Base (RGB) Trans (A)

	Tiling	Offset
x	1	0
y	1	0

0423_03_28.png

How it works...

Unity is able to read four channels of a texture map: R (red), G (green), B (blue) and A (alpha). Transparent shaders use RGB channels as the base texture (often called "diffuse texture"), while using the Alpha to set the transparency of the material according to each pixel's brightness level. Transparent shaders in the *Cutout* subgroup will not render semi-transparency, just allowing texture pixels to be invisible or fully opaque.

There's more...

Unity has a range of transparent shaders that can be used to achieve different results.

Cutouts

There is a subgroup of transparent shaders named *Cutout*. If you don't need semi-transparency in your material, it might be a good idea using a *Cutout* shader. They are faster and allow the object to cast and receive shadows.

Bumped Diffuse

To make an experiment, change the material shader to *Transparent/Bumped Diffuse*. Then, apply the image file *slideNormalTexture.png* (included in the folder 03_05) as a normal bump map.

See also

Creating a self-illuminated material
Creating specular texture maps
Solving transparent objects pop-ups
Disabling culling for a material

Using Cookie textures to simulate a cloudy outdoor

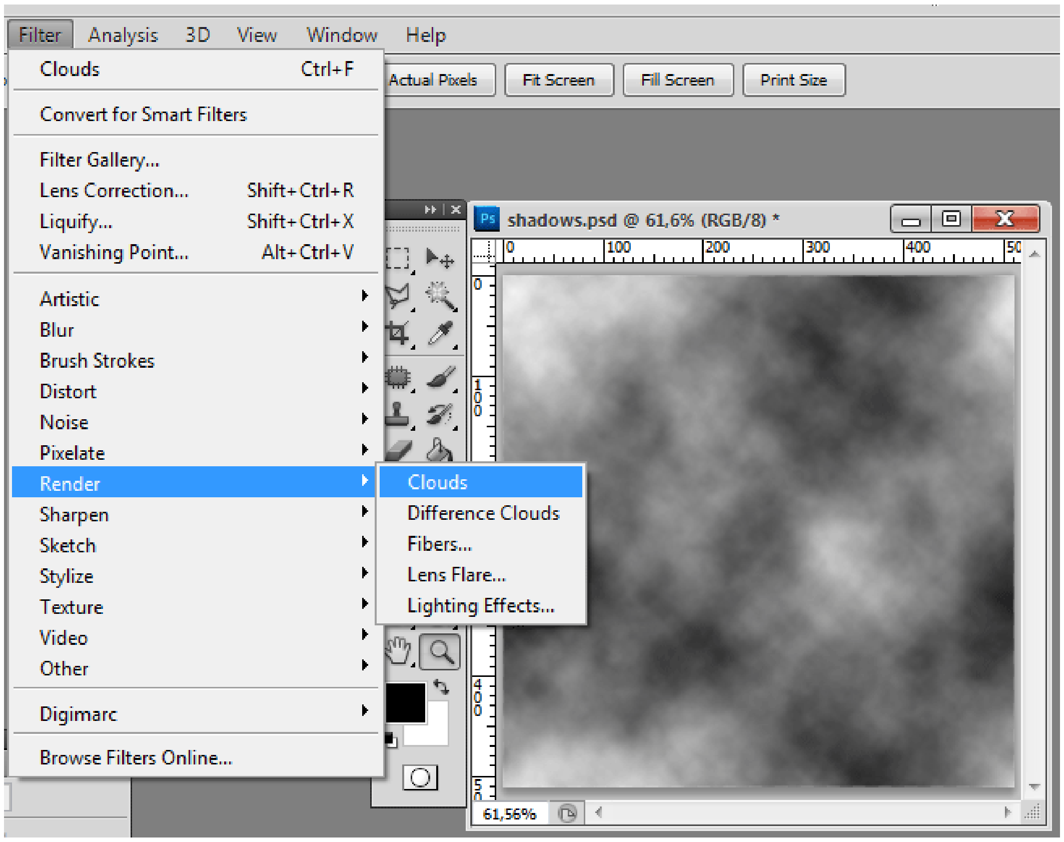
As it can be seen in many first person shooters and survival horror games, lights and shadows can add a great deal of complexity to a scene, helping immensely to create the right atmosphere to the game. In this recipe, we will create a cloudy outdoor environment lighting using cookie textures.

Getting ready

If you don't have access to a image editor or prefer to skip the texture map elaboration in order to focus on the implementation, please use the image file *cloudCookie.tga* provided in the folder 03_06.

How to do it...

1 - In your image editor, create a new 512 x 512 pixel image.
2 - Using black as foreground color and white as background color, apply the *Render Clouds* filter (in Photoshop, this is done through the menu *Filter > Render > Clouds*) (Note: image editors usually have a filter or command that renders clouds. If your image editor doesn't have such capability you can either paint it manually or use the image file *cloudCookie.tga* provided in the folder 03_06)



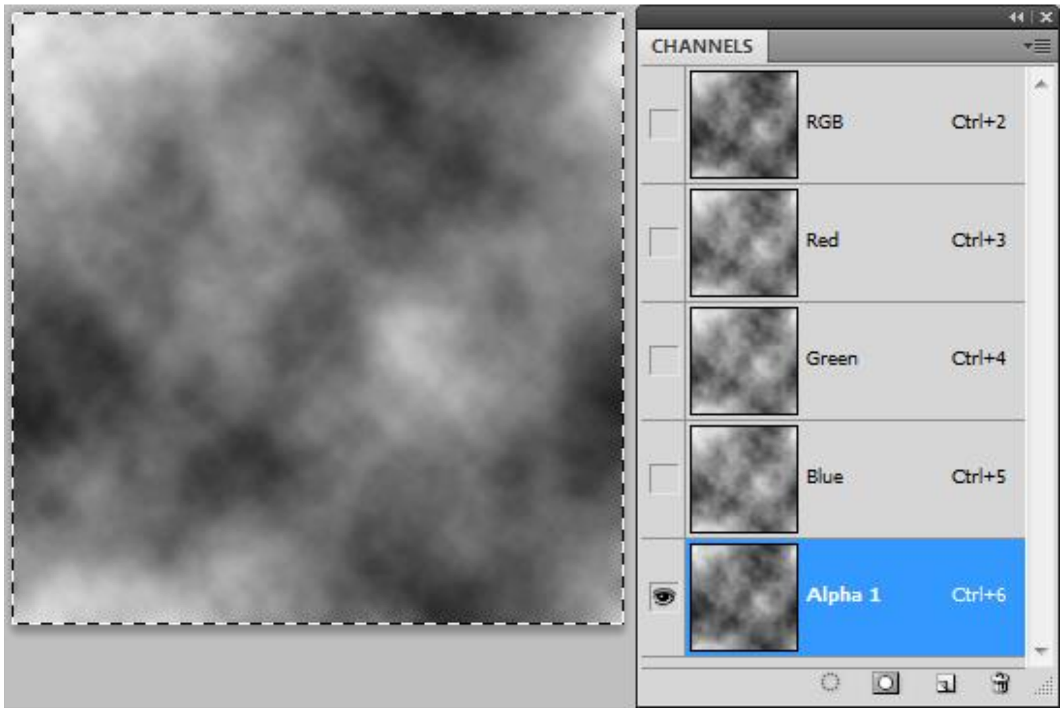
0423_03_29.png

3 - Select your entire image and copy it.

4 - Open the *Channels* window (in Photoshop this can be done through the menu *Window > Channels*).

5 - There should be three channels: Red, Green and Blue. Create a new channel. That will be the Alpha channel.

6 - In the *Channels* window, select the Alpha channel and paste your image into it.

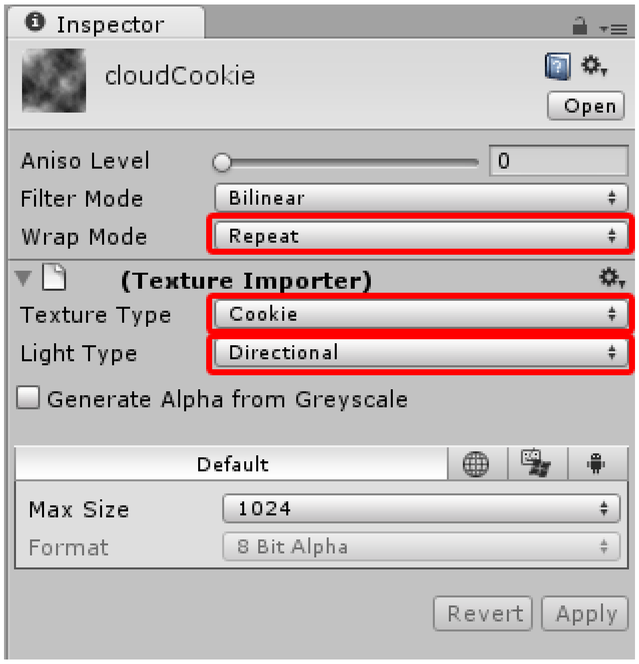


0423_03_30.png

7 - Save your image file as *cloudCookie.PSD* or TGA.

8 - Import your image file into Unity and select it in the *Project* window.

9 - In the *Inspector* window, change the *Wrap Mode* to *Repeat*, *Texture Type* to *Cookie* and *Light Type* to *Directional* (as shown in the image below).


Inspector

cloudCookie

Open

Aniso Level 0

Filter Mode Bilinear

Wrap Mode Repeat

(Texture Importer)

Texture Type Cookie

Light Type Directional

Generate Alpha from Greyscale

Default

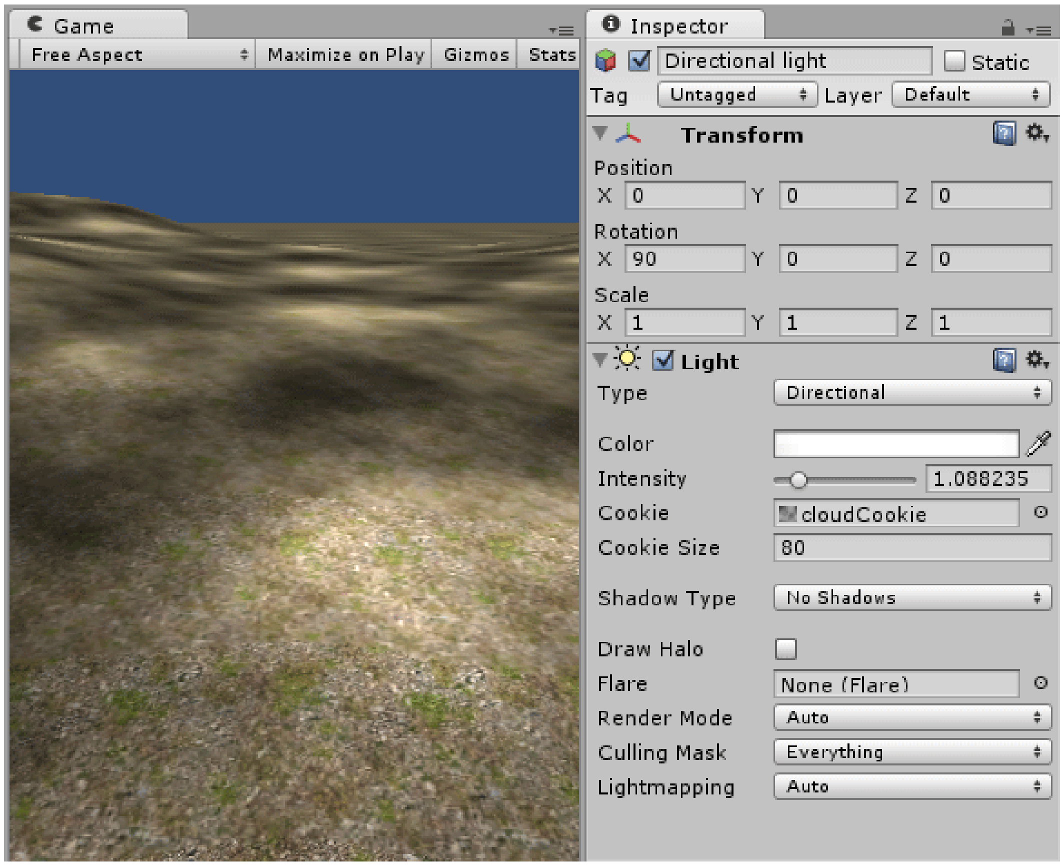
Max Size 1024

Format 8 Bit Alpha

Revert Apply

0423_03_31.png

10 - We will need a surface to actually see the lighting effect. You can either add a plane to your scene (via the menu *GameObject > Create Other... > Plane*) or create a Terrain (menu option *Terrain > Create Terrain*) and edit it, if so you wish.
11 - Let's add a light to our scene. Since we want to simulate sunlight, the best option is to create a *Directional Light*. You can do that through the dropdown menu named *Create* in the *Hierarchy* window.
12 - Using the *Transform* component of the *Inspector* window, reset the light position to 0, 0, 0 and its rotation to 90, 0, 0.
13 - In the *Cookie* field, select the image *cloudCookie* you have imported earlier. Change the field *Cookie Size* to 80, or a number that you feel as more appropriate to the scene's dimension. Please leave the *Shadow Type* as *No Shadows*



Directional light Static
Tag Untagged Layer Default

Transform
Position X 0 Y 0 Z 0
Rotation X 90 Y 0 Z 0
Scale X 1 Y 1 Z 1

Light
Type Directional
Color
Intensity 1.088235
Cookie cloudCookie
Cookie Size 80
Shadow Type No Shadows
Draw Halo
Flare None (Flare)
Render Mode Auto
Culling Mask Everything
Lightmapping Auto

0423_03_32.png

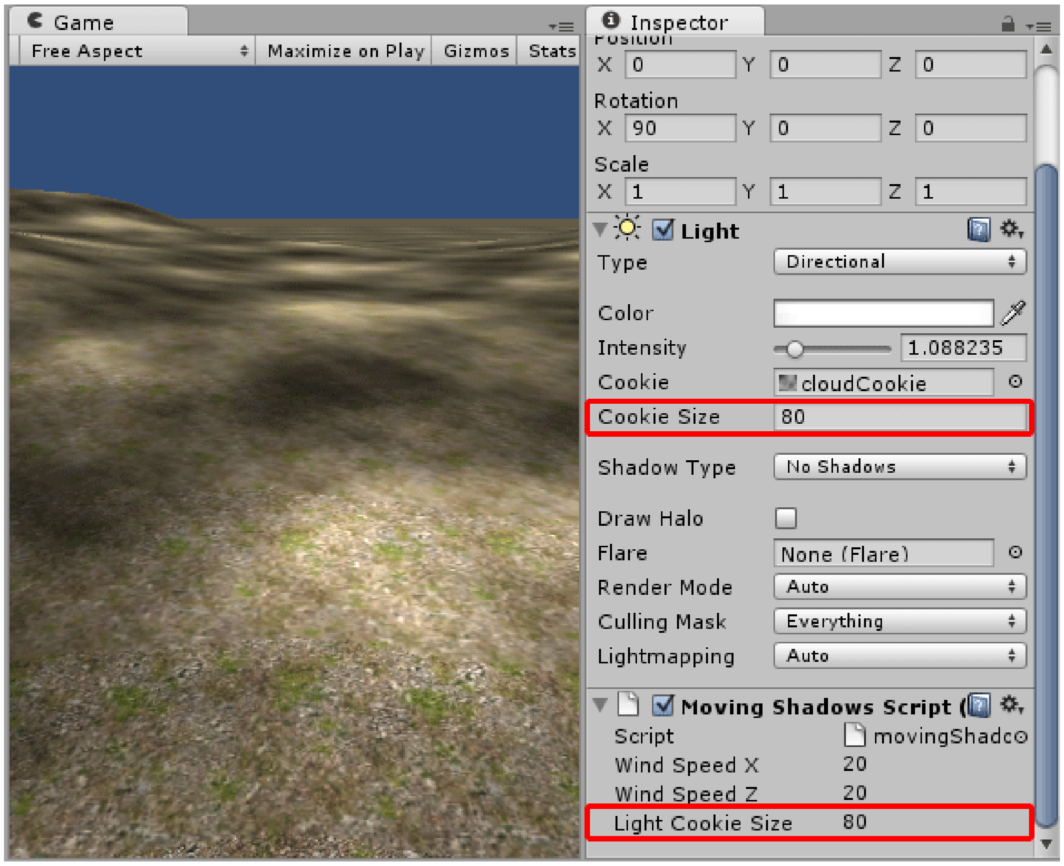
14 - Now, we need a script to translate our *Light* and, consequently, the *Cookie* projection. Using the dropdown menu in the *Project* window, create a new JavaScript and rename it to *movingShadowsScript*.

15 - Open your script and replace everything with the following code:

```
var windSpeedX : float;
var windSpeedZ : float;
var lightCookieSize : float; // make it equal to the light's
Cookie Size parameter in the inspector

private var initPos : Vector3;

function Start () {
   initPos = transform.position;

}

function Update () {
   if(Mathf.Abs(transform.position.x) >= Mathf.Abs(initPos.x)
+ lightCookieSize){

      transform.position.x = initPos.x;

   } else {

      transform.Translate(Time.deltaTime * windSpeedX, 0, 0,
 Space.World);

   }

   if(Mathf.Abs(transform.position.z) >= Mathf.Abs(initPos.z)
+ lightCookieSize){

      transform.position.z = initPos.z;

   } else {

      transform.Translate(0, 0, Time.deltaTime * windSpeedZ,
 Space.World);
   }

}
```

16 - Save your script and apply it to the *Directional Light*.

17 - Select the *Directional Light*. In the *Inspector* window, change the parameters *Wind Speed X* and *Wind Speed Z* to 20 (you can change these values as you wish). The parameter *Light Cookie Size* has to be changed to the exact amount of the light's *Cookie Size* (in our case, 80).



Position
X 0 Y 0 Z 0
Rotation
X 90 Y 0 Z 0
Scale
X 1 Y 1 Z 1

Light
Type: Directional
Color
Intensity: 1.088235
Cookie: cloudCookie
Cookie Size: 80
Shadow Type: No Shadows
Draw Halo
Flare: None (Flare)
Render Mode: Auto
Culling Mask: Everything
Lightmapping: Auto

Moving Shadows Script (
Script: movingShadc
Wind Speed X: 20
Wind Speed Z: 20
Light Cookie Size: 80

0423_03_33.png

18 - Play your scene. The clouds shadow should be moving.

How it works...

With our script, we are telling the *Directional Light* to move across the 'x' and 'z' axis, causing the *Light Cookie* texture to be displaced as well. Also, we reset the light object to its original position whenever it travels a distance equal to or greater than the *Light Cookie Size*. The light position must be reset to prevent it from travelling too far, causing problems in real time render and lighting. The *Light Cookie Size* parameter is used to ensure a smooth transition.

The reason we are not enabling shadows is because the light angle for the 'x' axis must be 90 degrees (or there will be a noticeable gap when the light resets to the original position). If you want dynamic shadows in your scene, please add a second Directional Light.

There's more...

In this recipe we have applied a cookie texture to a Directional Light. But what if we were using Spot or Point Lights?

Spot Light Cookies

Unity documentation has an excellent tutorial on how to make Spot Light cookies. This is great to simulate shadows coming from projectors, windows etc. You can check it out at `http://unity3d.com/support/documentation/Manual/HOWTO-LightCookie.html`

Point Light Cookies

If you want to use a cookie texture with a Point Light, you'll need to change the *Light Type* in the *Texture Importer* section of the *Inspector*.

See also

Creating a color selection dialog

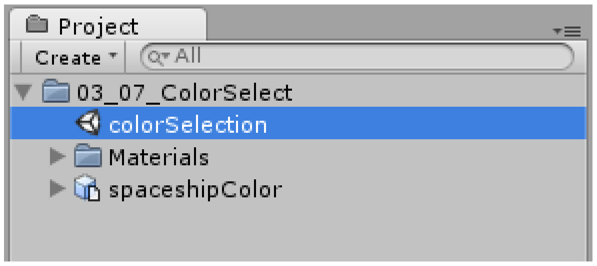
In-game customization and user-made content have been taking over games for a while. A very common feature is the ability of changing one's avatar color. In this recipe, we will create a dialog box that allows the player to change the object's color trough sliders that modify the Red, Green and Blue values of the material.

Getting ready

If you want to use our sample scene to follow this recipe, please import the package named *colorSelector*, available in the folder named 03_07.

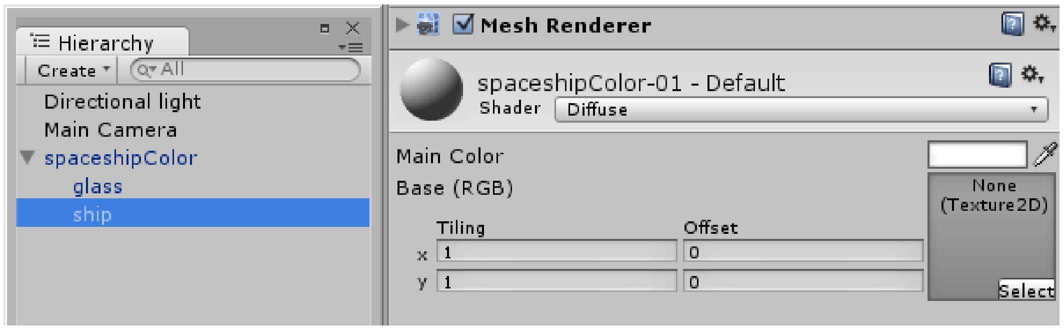
How to do it...

1 - Once the package has been imported, open the scene *colorSelection*.



0423_03_34.png

2 - Expand the *spaceshipColor* Game Object in the *Hierarchy* window and select its child named *ship*. This is the object that will have its material changed by our script.



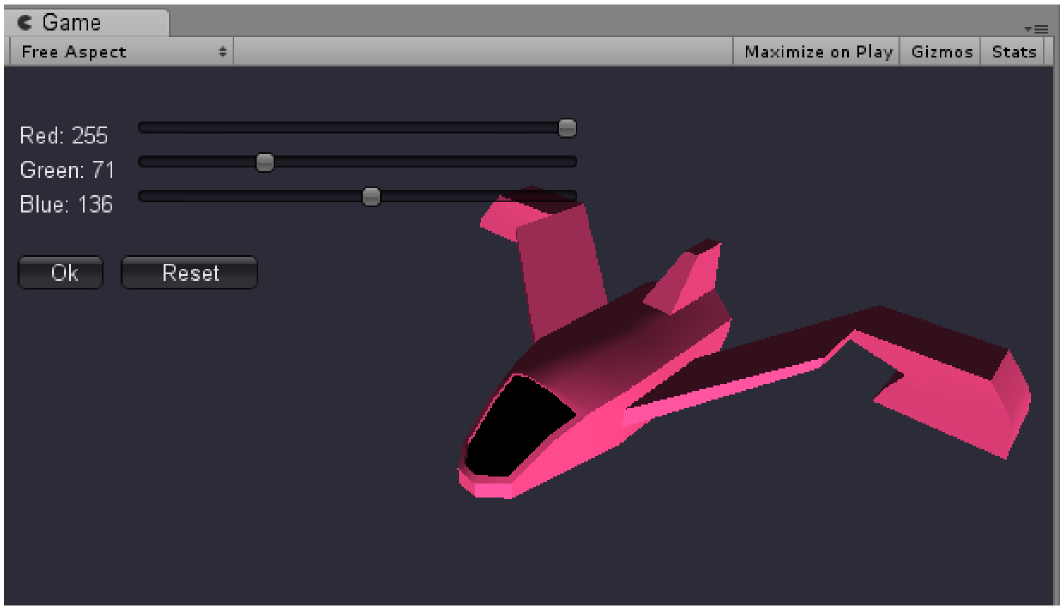
0423_03_35.png

3 - In the *Project* window, create a new Java Script. Rename it *ColorSelectorScript* and open it in your editor.

4 - Add the following code to the top of the script:

```
var redValue : float = 1.0;
var greenValue : float = 1.0;
var blueValue : float = 1.0;
var selectorOn : boolean = false;
private var redReset : float = 1.0;
private var greenReset : float = 1.0;
private var blueReset : float = 1.0;

function OnMouseUp(){

   selectorOn = true;

}

function OnGUI(){

   if(selectorOn){

      GUI.Label (Rect (10, 30, 90, 20), "Red: " +
Mathf.RoundToInt(redValue * 255));
      redValue = GUI.HorizontalSlider (Rect (80, 30, 256, 20),
redValue, 0.0, 1.0);

      GUI.Label (Rect (10, 50, 90, 20), "Green: " +
Mathf.RoundToInt(greenValue * 255));
      greenValue = GUI.HorizontalSlider (Rect (80, 50, 256,
20), greenValue, 0.0, 1.0);

      GUI.Label (Rect (10, 70, 90, 20), "Blue: " +
Mathf.RoundToInt(blueValue * 255));
      blueValue = GUI.HorizontalSlider (Rect (80, 70, 256,
20), blueValue, 0.0, 1.0);

      if (GUI.Button(Rect(10,110,50,20),"Ok")){

         selectorOn = false;
         redReset = redValue;
         greenReset = greenValue;
         blueReset = blueValue;

      }

      if (GUI.Button(Rect(70,110,80,20),"Reset")){

         redValue = redReset;
         greenValue = greenReset;
         blueValue = blueReset;

      }

      renderer.material.SetColor("_Color", Color(redValue,
greenValue, blueValue, 1));


   } else {

      GUI.Label (Rect (10, 10, 500, 20), "Click the spaceship
to change color");

   }

}
```

5 - Apply *ColorSelectorScript* to the *ship* Game Object (and not its parent object, *spaceshipColor*).
6 - Add a collider to the *ship* game object. Again, it is important that the collider is applied to *ship*, not its parent object. Adding a collider can be done by selecting the object and accessing the menu *Component > Physics > Mesh Collider*.
7 - Play your scene and test your color selector.



0423_03_36.png

How it works...

Besides assigning the sliders values to the material color, we are keeping track of the last used color, storing its values into three variables. This is necessary in case the player wants to go back to the previous color after messing up with the sliders.

Also, we multiply those values by 255 in the text label so the player can read the RGB values in a more traditional way.

There's more...

With just a few twists, you could use this recipe to change other properties of your object's material (such as its transparency or emissive color, for instance).

See also

Combining textures in real time through the GUI

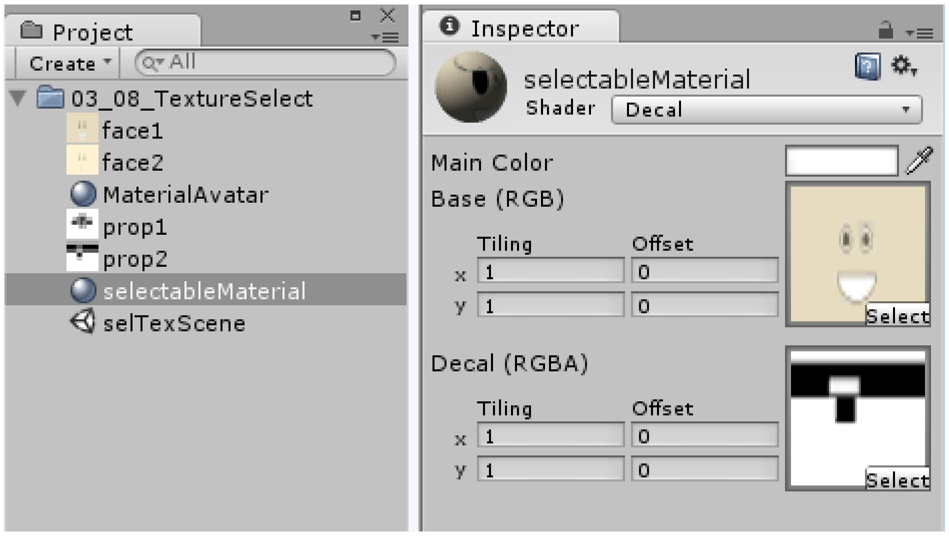
The customization of avatars usually include the selection the one or more textures that define its looks. In this recipe, we will implement a GUI that allows the player to create their avatar by combining two texture channels.

Getting ready

The 3d object and image files needed for this recipe are included in the package *selectTexture*, inside the folder 03_08.

How to do it...

1 - Import the Unity package named *selectTexture* into your Project.
2 - Open the level named *selTextScene*
3 - Let's create our base material: in the *Project* window, use the drop down menu to create a new material. Name it *selectableMaterial*
4 - Change the shader of the *selectableMaterial* to *Decal*. Then, apply the texture map named *face1* as the *Base* map and *prop1* as the *Decal* map.



selectableMaterial
Shader: Decal
Main Color
Base (RGB)
Tiling: x 1, y 1
Offset: x 0, y 0
Decal (RGBA)
Tiling: x 1, y 1
Offset: x 0, y 0

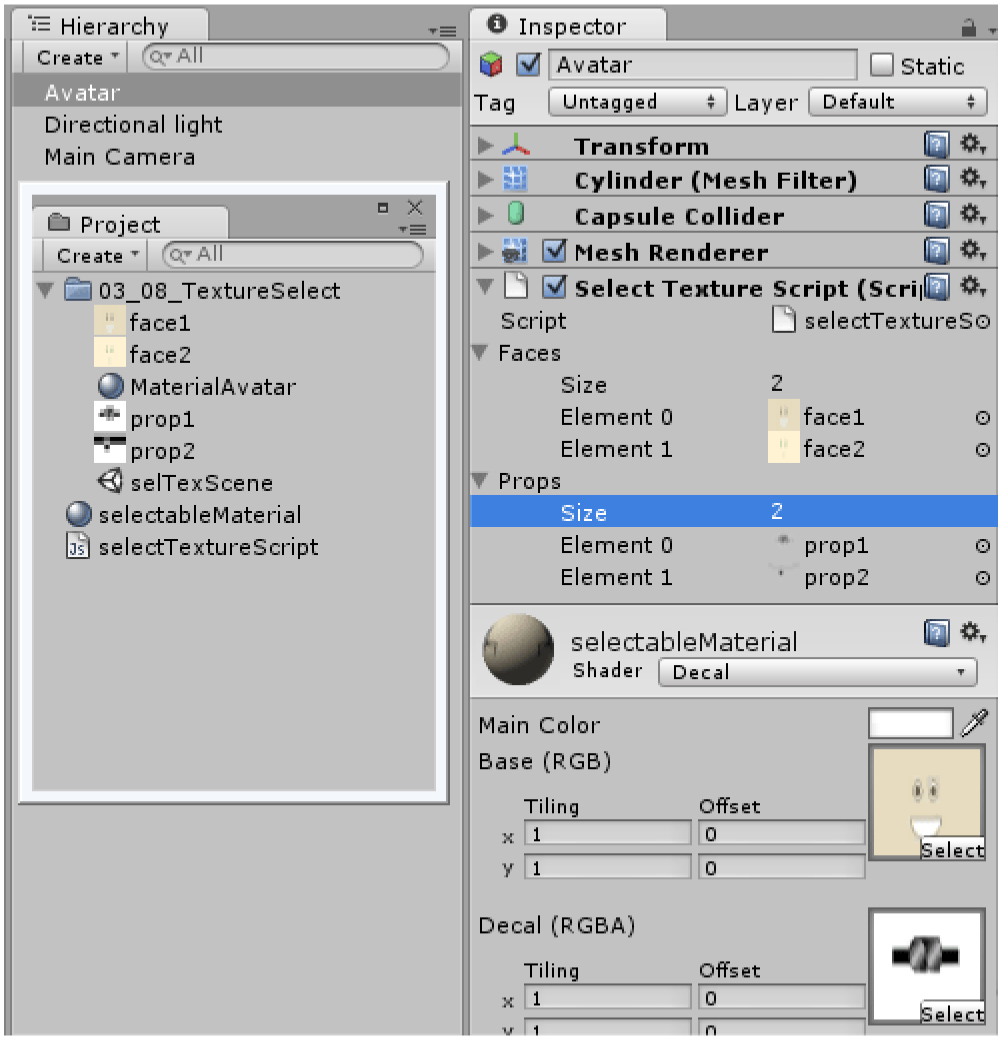
0423_03_37.png

5 - Apply the material to the game object *Avatar*. You can do this by dragging the material from the *Project* window into the Game Object's name in the *Hierarchy* window.
6 - In the *Project* window, create a new Java Script and rename it *selectTextureScript*.
7 - Open *selectTextureScript* in your script editor and add the following lines of code to the top of the script:

```
var faces : Texture2D[];
var props : Texture2D[];

function OnGUI(){

   for(var i=0; i<faces.length; i++){

      if (GUI.Button(Rect(0,i * 64,128,64),faces[i])){

            changeMaterial("faces",i);

      }
   }

   for(var j=0; j<props.length; j++){

      if (GUI.Button(Rect(128,j * 64,128,64),props[j])){

         changeMaterial("props",j);

      }
   }

}


function changeMaterial(category : String, index : int){

   if(category == "faces"){
      renderer.material.mainTexture = faces[index];
   }

   if(category == "props"){
      renderer.material.SetTexture("_DecalTex", props[index]);
   }

}
```

8 - Save your script and apply it to *Avatar* Game Object.
9 - In the *Inspector* window, change the *Size* value of both *Faces* and *Props* to "2".

10 - Change the *Element 0* and *1* of *Faces* to *face1* and *face2*. Do the same for the *Element 0* and *1* of *Props* (changing it to *prop1* and *prop2* instead).



Avatar
Directional light
Main Camera

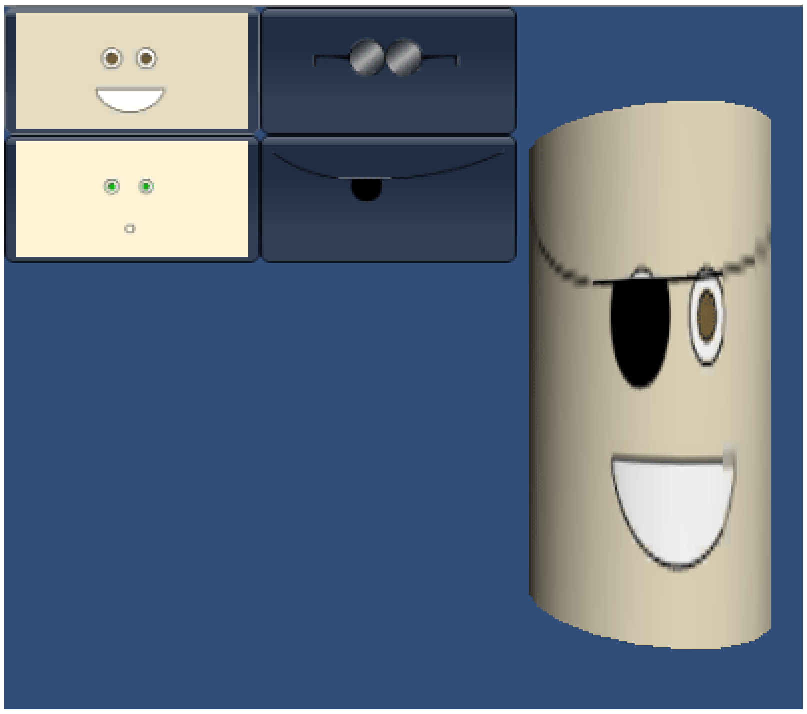
03_08_TextureSelect
face1
face2
MaterialAvatar
prop1
prop2
selTexScene
selectableMaterial
selectTextureScript

Avatar Static
Tag Untagged Layer Default
Transform
Cylinder (Mesh Filter)
Capsule Collider
Mesh Renderer
Select Texture Script (Scri
Script selectTextureS
Faces
Size 2
Element 0 face1
Element 1 face2
Props
Size 2
Element 0 prop1
Element 1 prop2

selectableMaterial
Shader Decal
Main Color
Base (RGB)
Tiling Offset
x 1 0
y 1 0
Decal (RGBA)
Tiling Offset
x 1 0
y 1 0

0423_03_38.png

11 - Play the scene. You will be able to select your texture combination by clicking on the appropriate buttons.



0423_03_39.png

How it works...

The script allows the user to create two collections of textures: one for the *Base* map (named *Faces*) and another one for the *Decal* (named *Props*). When the scene is played, the textures are displayed inside GUI buttons which can be used to change the texture in the *Avatar's* material by calling the function *changeMaterial*. This function will receive as parameters both the category (either *Face* or *Prop*) and index of the image, assigning the correspondent texture map to the appropriate texture channel.

There's more...

This recipe could be easily adapted to change other parameters of different material shaders. Check Unity's online documentation to learn about other texture names at http://unity3d.com/support/documentation/ScriptReference/Material.SetTexture.html . Also, you might want to learn more about shaders by exploring the built-in shaders source available at `http://unity3d.com/support/resources/assets/built-in-shaders`

See also

Creating a color selection dialog

Highlight materials at mouse over

Highlighting an object is a very effective way of letting players know they can interact with it. This is very useful in a number of game *genres* such as puzzles and point-and-click adventures, and it can also be applied to create 3D user interfaces.

Getting ready

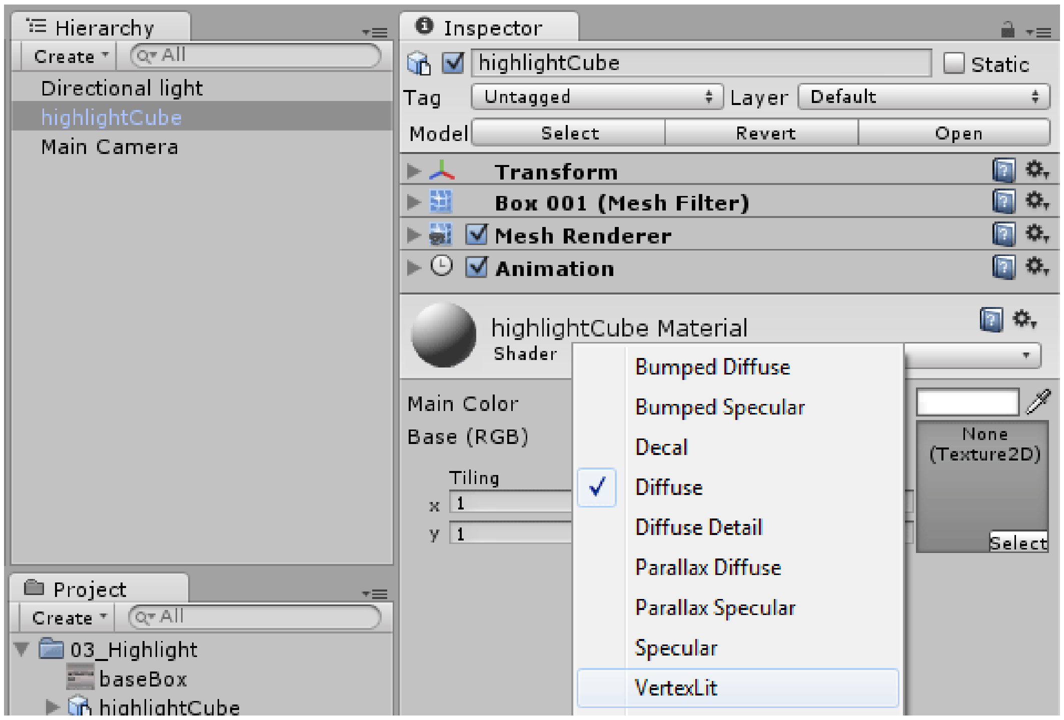
For this recipe, you'll need a 3D model and a 2D texture map. If you don't have them, please import the package *highlight.unitypackage*, available in the folder 03_09, into your project.

How to do it...

1 - Import the Unity Package and open the *highlightScene* scene.

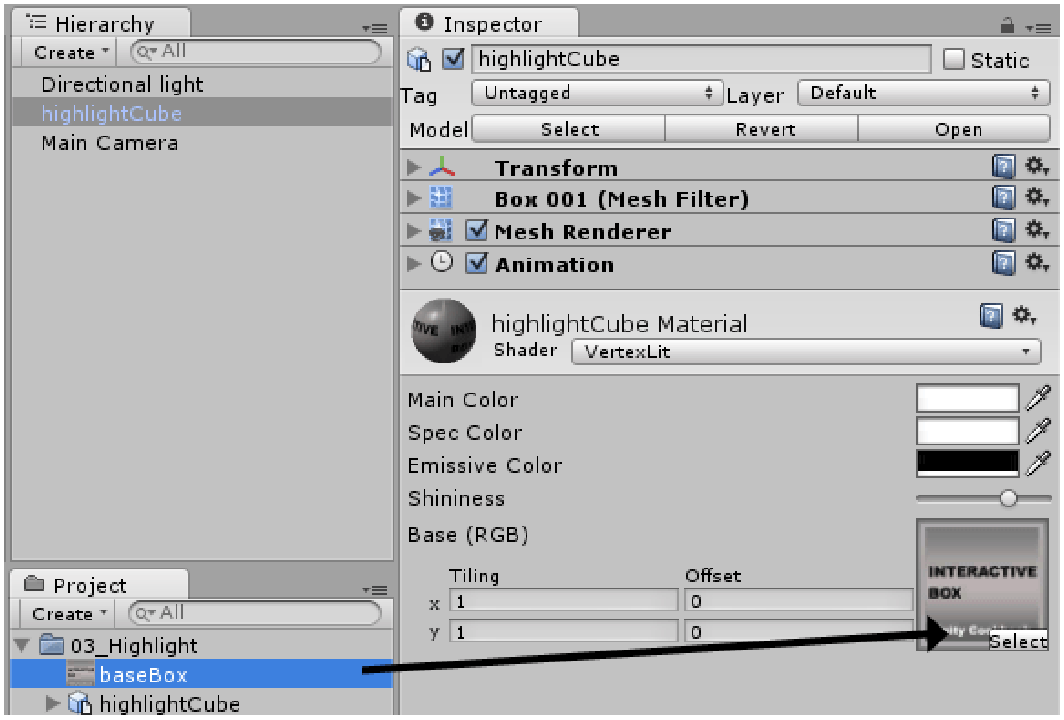
2 - In the *Hierarchy* window, select the 3D object to be highlighted (in our case, the object named *highlightCube*).

3 - The Inspector window should display the Game Object's material. Using the drop-down menu, change its shader from *Diffuse* to *VertexLit*.



0423_03_40.png

4 - Apply the texture *baseBox* to the *Base* texture of the material.



0423_03_41.png

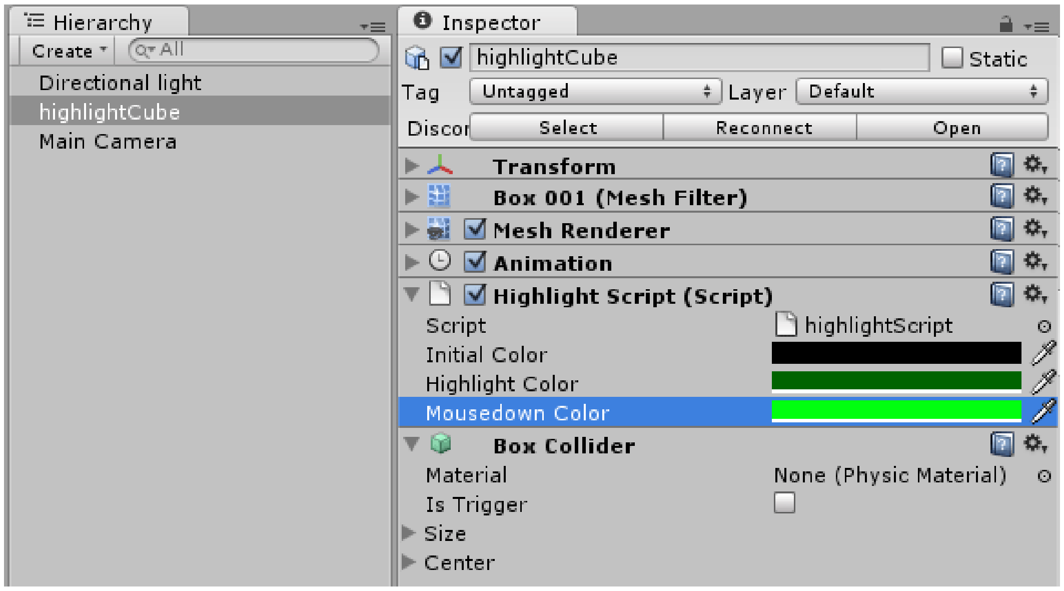
5 - Note that VertexLit shader has a property called *Emissive Color*, which should be black as default. If you want a preview of what is going to happen later in the recipe, change that to green (but be sure to change it back to black).

6 - We need to create a Script. In the *Project* window, click the *Create* drop down menu and choose JavaScript. Rename it *highlightScript* and open it in your editor.

7 - Add the following code to the top of your script:

```
var initialColor : Color;
var highlightColor : Color;
var mousedownColor : Color;
private var mouseon : boolean = false;

function OnMouseEnter () {

   mouseon = true;
   renderer.material.SetColor ("_Emission", highlightColor);

}

function OnMouseExit () {

   mouseon = false;
   renderer.material.SetColor ("_Emission", initialColor);
}
function OnMouseDown () {
   renderer.material.SetColor ("_Emission", mousedownColor);
}
function OnMouseUp () {
   if(mouseon){
   renderer.material.SetColor ("_Emission", highlightColor);
   } else {
   renderer.material.SetColor ("_Emission", initialColor);
   }
}
```

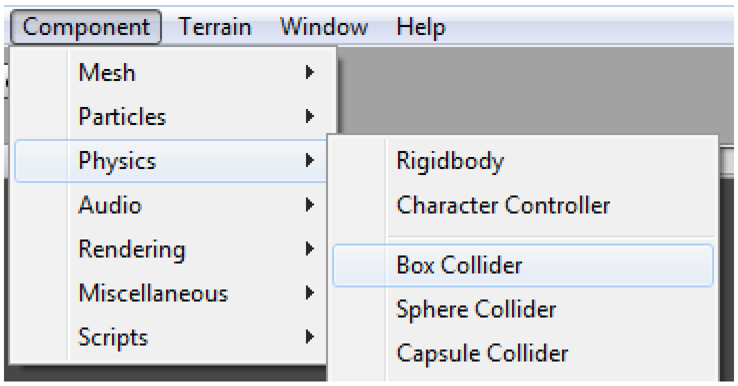
8 - Save your script and apply it to the *highlightCube* Game Object (in the *Hierarchy* window).
9 - Select the *highlightCube* and, in the *Inspector* window, change the *Highlight Color* to dark green (R: 0, G: 100, B: 0) and the *Mousedown Color* to light green (R:0, G: 255, B: 0).

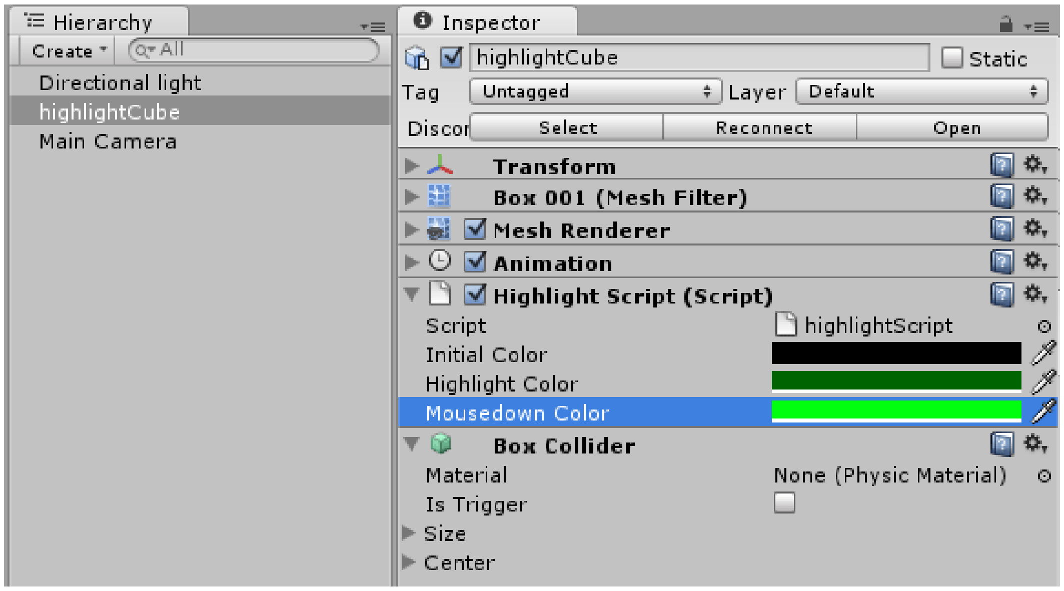


highlightCube	
Tag	Untagged
Layer	Default
Script	highlightScript
Initial Color	⬛
Highlight Color	🟩
Mousedown Color	🟩
Material	None (Physic Material)
Is Trigger	☐

0423_03_42.png

10 - Add a box collider to the highlightCube by accessing the menu Component > Physics > Box Collider.





highlightCube Static
Tag Untagged Layer Default
Transform
Box 001 (Mesh Filter)
Mesh Renderer
Animation
Highlight Script (Script)
Script highlightScript
Initial Color
Highlight Color
Mousedown Color
Box Collider
Material None (Physic Material)
Is Trigger
Size
Center

0423_03_43.png

11 - Test the scene. The box should be highlighted when the mouse is over it (and even more when clicked).

How it works...

The box collider detects the mouse pointer over the object, working as a trigger for the emissive color value change. The boolean variable *mouseon* is used to detect if the mouse button is released within or without the box collider, changing its color accordingly.

There's more...

You can achieve other interesting results using other shaders, but be sure to address the changes to their particular material properties.

Self-Illuminated shaders

Self Illuminated shaders will work if you replace `"_Emission"` by `“_Color”` in the script.

Transparent shaders

Transparent shaders can be an interesting option. Remember you can change the transparency of the material by changing the Alpha value of its main color (which should be addressed as `“_Color”` in the script).

See also

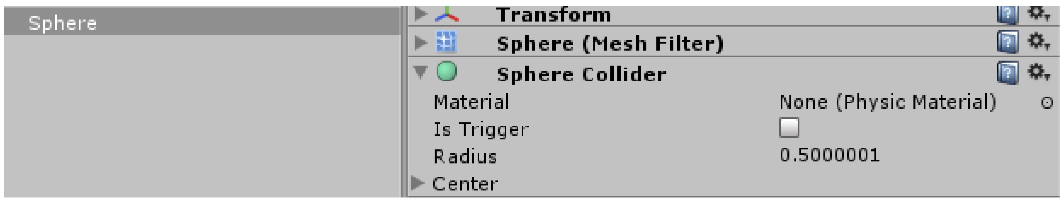
Fading the transparency of a material

In this recipe, we will create an object that, once clicked, fades out and disappear. However, the script will be flexible enough to allow us adjust the initial and final alpha values. Plus, we will have the option of making the object self-destructible when turning invisible.

How to do it...

1 - Add a sphere to your scene by accessing the menu *GameObject > Create Other... > Sphere*.

2 - Select the *Sphere* and make sure it has a collider (if you are using a custom 3d object, you might have to add a collider through the *Components > Physics* menu)



```
Material      None (Physic Material)
Is Trigger    ☐
Radius        0.5000001
```

0423_03_44.png

3 - Create a new material. The easiest way to do that is to access the *Project* window, click the *Create* drop-down menu and choose *Material*.

4 - Rename your new material. For this example, let’s call it *fadeMaterial*
5 - Select your material. In the *Inspector* window, under the material’s name, use the drop-down menu to change its shader to *Transparent / Diffuse*.
6 - Apply the *fadeMaterial* to *Sphere* by dragging it from the *Project* window into the *Sphere* Game Object name in the *Hierarchy* window.
7 – We need a Java Script. In the *Project* window, click the *Create* drop down menu and choose *JavaScript*. Rename it to *FadeMaterialScript* and open it in your editor.
8 - Replace your script with the code below:

```
var fadeFromStart : boolean = false;
var fadeDuration : float = 1;
var useMaterialAlpha : boolean = false; // Set it to "true" if
you want to use the alpha value of the material
var alphaStart : float = 1.0;
var alphaEnd : float = 0.0;
var destroyInvisibleObject : boolean = true;
private var fadeAlpha : boolean = false;
private var alphaDiff;
private var startTime;

function Start(){

   if(!useMaterialAlpha){

      renderer.material.color.a = alphaStart;

   } else {

      alphaStart = renderer.material.color.a;

   }

   alphaDiff = alphaEnd - alphaStart;

   if(fadeFromStart){

      FadeAlpha();

   }


}

function Update () {



   if(fadeAlpha){

      var elapsedTime = Time.time - startTime;
      if(elapsedTime <= fadeDuration){

         renderer.material.color.a = alphaStart +
((elapsedTime / fadeDuration) * alphaDiff);

      } else {

         renderer.material.color.a = alphaEnd;

         if(renderer.material.color.a <= 0){

            Destroy (gameObject);

         }
         fadeAlpha = false;

      }

   }

}

function FadeAlpha(){

   fadeAlpha = true;
   startTime = Time.time;

}

function OnMouseUp(){

   FadeAlpha();

}
```

9 - Save your script and apply it to the *Sphere* Game Object (in the *Hierarchy* window).
10 - Play your scene and click on the sphere to see it fade away and self-destruct

How it works...

Since the opaqueness of the material using a transparent shader is determined by the alpha value of its main color, all we need to do in order to fade it is changing that value over a given amount of time. This transformation is expressed, in our script, on the following line of code:

```
renderer.material.color.a = alphaStart + ((elapsedTime /
fadeDuration) * alphaDiff);
```

There's more...

You could call the *FadeAlpha* function in other circumstances (such as a *Rigidbody* collision, for instance). In fact, you could even call it from another Game Object's script by using the *GetComponent* command. The script would be something like:

```
GameObject.Find("Sphere").GetComponent("FadeMaterialScript").F
adeAlpha();
```

See also

Playing videos inside a scene (Pro Only)

TV sets, projectors, monitors... If you want complex animated materials in your level, you can play video files as texture maps. In this recipe, we will learn how to apply a video texture to a cube. We will also implement a simple control scheme that plays or pauses the video when the cube is clicked.

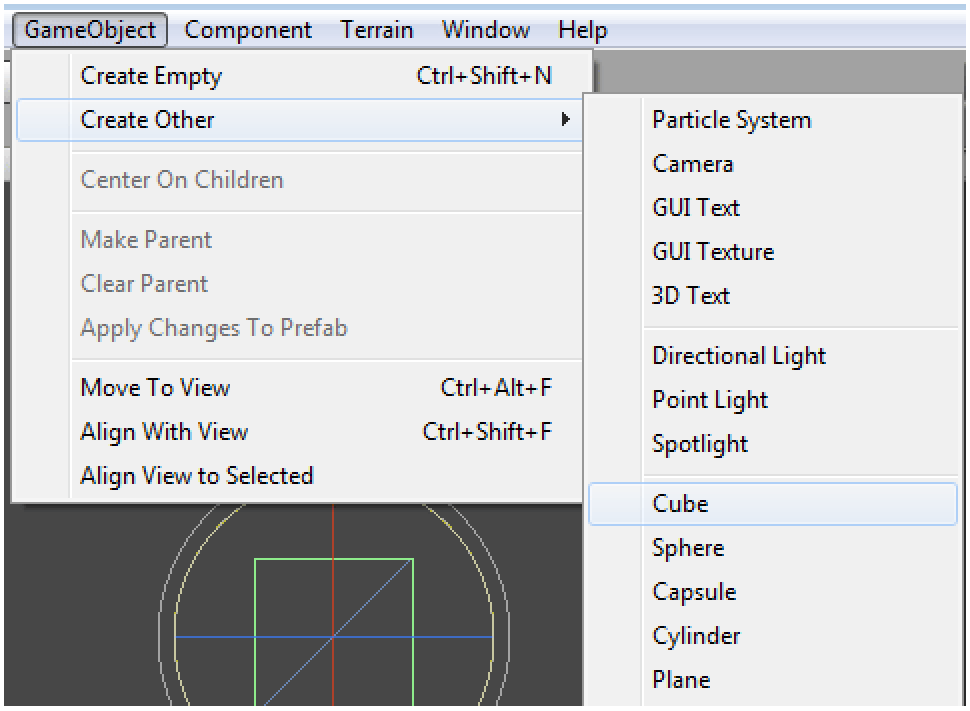
Getting ready

Unity imports video files through Apple Quicktime. If you don't have it installed in your machine, please download it at `http://www.apple.com/quicktime/download/`

Also, if you need a video file to follow this recipe, please use the *videoTexture.mov* included in the folder 03_11_12.

How to do it...

1 - Create a cube by accessing the menu Game Object > Create Other > Cube.



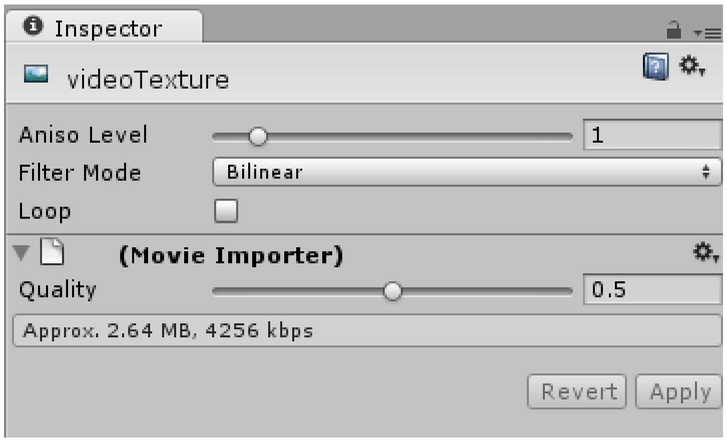
0423_03_45.png

2 - In the Project window, use the Create drop-down menu to create a new material. Rename it to videoMaterial.

3 - We need to import our video texture. Access the menu Assets > Import New Asset... and browse to your video file (it will import and convert any video file your Quicktime is capable of reading).

4 - Unity will convert your file to Ogg Theora format.

5 - Select your video texture and check its parameters in the Inspector window. Let's keep all default values and leave the Loop option unselected.

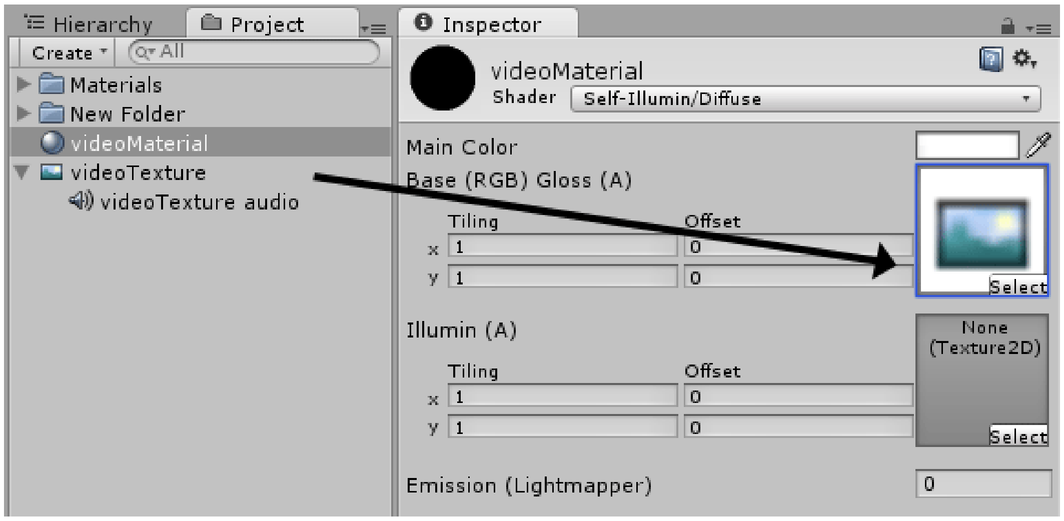


Aniso Level: 1
Filter Mode: Bilinear
Loop: ☐

(Movie Importer)
Quality: 0.5
Approx. 2.64 MB, 4256 kbps

0423_03_46.png

6 - In the *Project* window, select *videoMaterial*. In the *Inspector* window, change its Shader option *to Self-Illumin/Diffuse*.
7 - Apply the video texture on the *Main Color* of the *videoMaterial* by dragging it from the *Project* window into the appropriate slot.



Shader: Self-Illumin/Diffuse

Main Color
Base (RGB) Gloss (A)

	Tiling	Offset
x	1	0
y	1	0

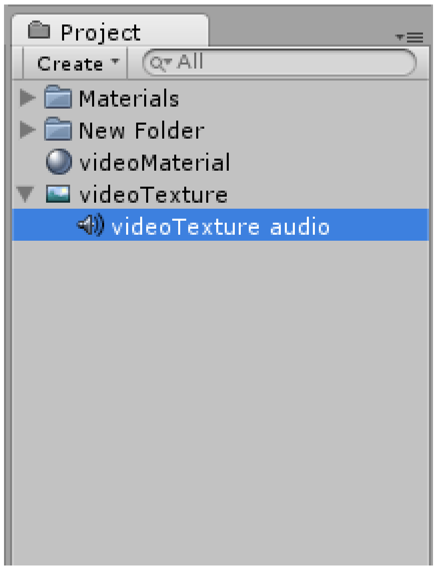
Illumin (A)
None (Texture2D)

	Tiling	Offset
x	1	0
y	1	0

Emission (Lightmapper): 0

0423_03_47.png

8 - Apply the *videoMaterial* to the cube you have created.
9 - Expand the video texture on the *Project* window to reveal its *Audio Clip*. Then, apply the audio clip to the cube (you can do it by dragging it from the *Project* window to the *Cube* in the *Hierarchy* window or a viewport).



0423_03_48.png

10 - Select the cube. Make sure there is a *Collider* component in the inspector. In case there isn't one, add it via the menu *Component > Physics > Box Collider*. Colliders are needed for mouse collision detection.
11 - Now we need to create Script. In the *Project* window, click the *Create* drop down menu and choose JavaScript. Rename it *playVideoTexture* and open it in your editor.
12 - Replace your script with the code below:

```
var loopVideo : boolean = true;
var startPlaying : boolean = true;
private var videoTexture : MovieTexture;

function Start () {

   videoTexture = renderer.material.mainTexture;
   videoTexture.loop = loopVideo;

   if(startPlaying){
      controlMovie();
   }

}

function OnMouseUp(){

   controlMovie();

}
```

```
function controlMovie(){

   if(videoTexture.isPlaying){

      videoTexture.Pause();

   } else {

      videoTexture.Play();
      audio.Play();

   }

}

@script RequireComponent (AudioSource)
```

13 - Save your script and apply it to the *Cube* (in the *Hierarchy* window).
14 - Test your scene. You should be able to see the movie being played in the cube face, and also pause / play it by clicking on it.

How it works...

By default, our script makes the movie texture play in loop mode. There is, however, a boolean variable than can be changed through the *Inspector* window, where it is represented by a check box. Likewise, there is a check box that can be used to prevent the movie from playing when the level starts.

There's more...

There are some other movie texture commands and parameters that can be played with. Don't forget to check out Unity's scripting guide at `http://docs.unity3d.com/Documentation/ScriptReference/MovieTexture.html.`

See also

Using a video file as an intro or cut-scene (Pro only)

Pre-rendered scenes are very common in games. They can be used to display flying logos, introductory movies or narrative cut scenes. In this recipe we will create am introductory video that, once finished, redirects us to the next level of the game.

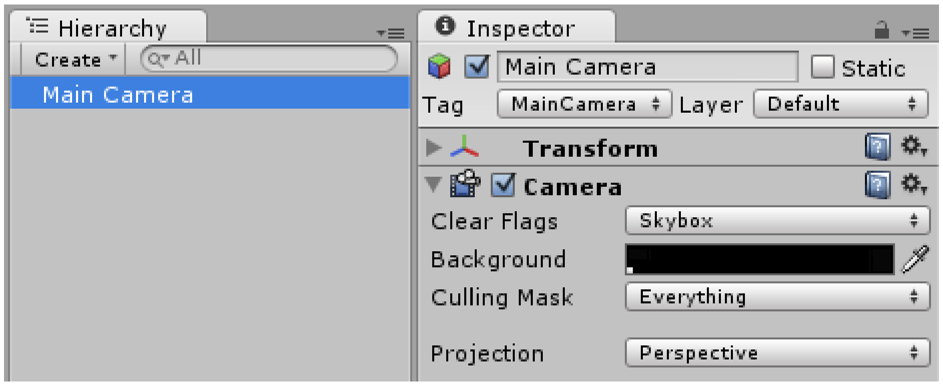
Getting ready

Unity imports video files through Apple Quicktime. If you don't have it installed in your machine, please download it at http://www.apple.com/quicktime/download/

Also, if you need a video file to follow this recipe, please use the *videoTexture.mov* included in the folder 03_11_12.

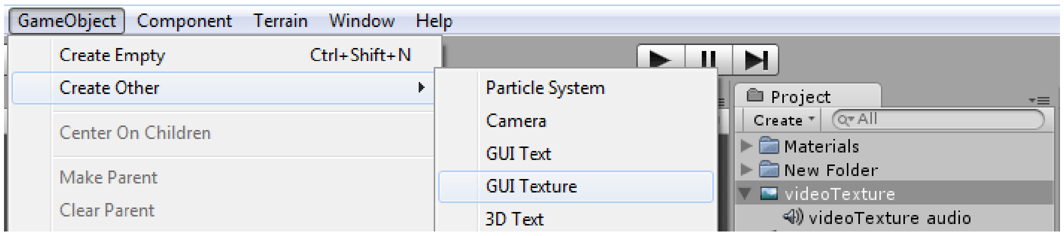
How to do it...

1 - In the *Hierarchy* window, select the *Main Camera* and change the background color to black.



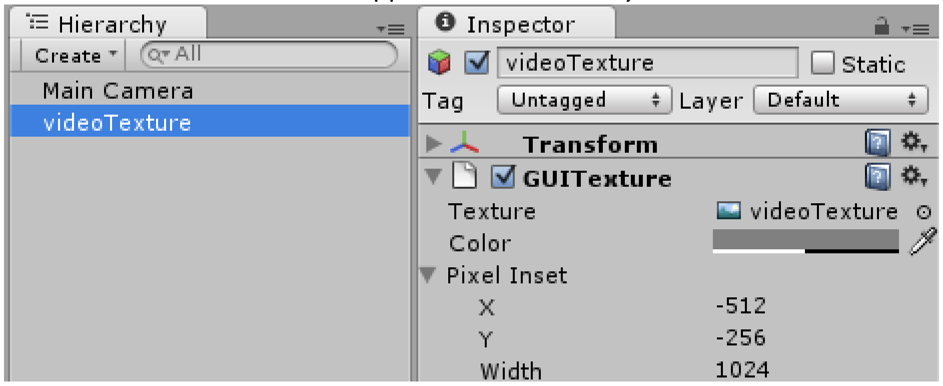
0423_03_49.png

2 - Access the menu *Assets* > *Import New Asset...* and browse to your video file. Unity will convert it to Ogg Theora format and add it to the *Project* window.
3 - In the *Project* window, select your video texture and check the parameters at the *Inspector* window. Make sure the *Loop* option is unselected.
4 - Keep the video texture selected in the *Project* window and make it a GUI Texture by accessing the menu *GameObject* > *Create Other...* > *GUI Texture*.



0423_03_50.png

5 - A new GUI Texture should appear in the *Hierarchy* window.

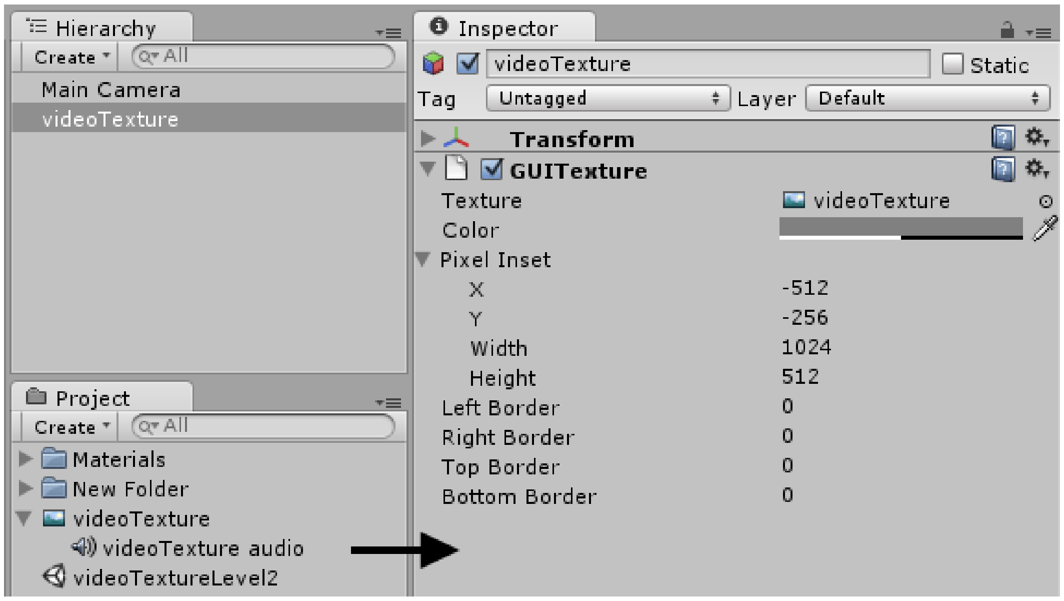


videoTexture
Static
Tag Untagged Layer Default

Texture videoTexture
Color
Pixel Inset
X -512
Y -256
Width 1024

0423_03_51.png

6 - We still need to add the movie's audio clip to our texture. Do this by dragging the audio clip within *videoTexture* (inside the Project window) to the GUI Texture in the *Hierarchy* window.



videoTexture
Static
Tag Untagged Layer Default

Texture videoTexture
Color
Pixel Inset
X -512
Y -256
Width 1024
Height 512
Left Border 0
Right Border 0
Top Border 0
Bottom Border 0

0423_03_52.png

7 – Now, let's create a script to control the movie's playback, size and position. In the *Project* window, click the *Create* drop down menu and choose JavaScript. Rename it *cutsceneScript* and open it in your editor.

8 - Replace your script with the code below:

```
var movieTexture : MovieTexture ;
var nextLevel : String;

function Start(){


   if(!movieTexture){

      movieTexture = guiTexture.texture;

   } else {

      guiTexture.texture = movieTexture;

   }


   var movieAspectRatio : float = movieTexture.width /
movieTexture.height;
   var screenAspectRatio : float = Screen.width /
Screen.height;
   var movieWidth : int;
   var movieHeight : int;

   if (movieAspectRatio >= screenAspectRatio){

      movieWidth = Screen.width;
      movieHeight = movieWidth / movieAspectRatio;

   } else {

   movieHeight = Screen.height;
   movieWidth = movieHeight * movieAspectRatio;

}

guiTexture.pixelInset = Rect (-(movieWidth * 0.5), -
(movieHeight * 0.5), movieWidth, movieHeight);

movieTexture.Play();
audio.Play();

}

function Update(){

if (movieTexture == false){
Application.LoadLevel (nextLevel);
}

}
```

9 - Save your script and apply it to the GUI Texture (in the *Hierarchy* window).

10 - Test your scene. The script will play the movie and, once the movie is over, attempt to load the level specified in the field *Next Level*.

How it works...

Once a Movie Texture has been assigned as a GUI Texture, all we need to do is create a script to adjust its size to the screen, play it and detect when it reaches the end (so it can load the next level). Please note this script should work properly with videos in portrait orientation, only.

There's more...

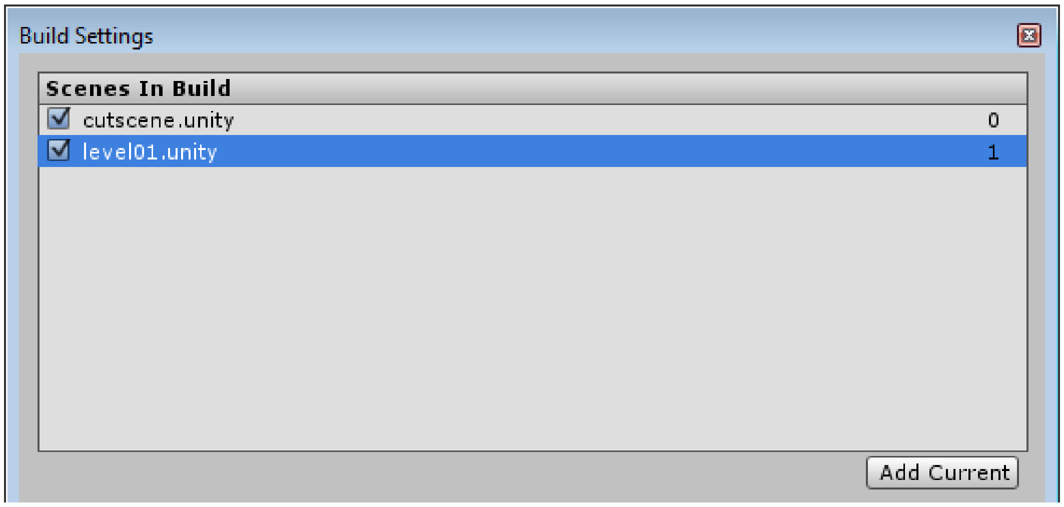
Some details you don't want to miss:

Play other video files

You could assign another Movie Texture to the GUI through the *Cutscene Script* component in the *Inspector* window.

Don't forget to add your levels to the Build Settings list!

Always remember to include levels to be loaded in the *Build Settings* window (through the menu *File > Build Settings...*)



Scenes In Build	
☑ cutscene.unity	0
☑ level01.unity	1

0423_03_53.png

See also

Animating textures by looping through array of materials (e.g. simulated video)

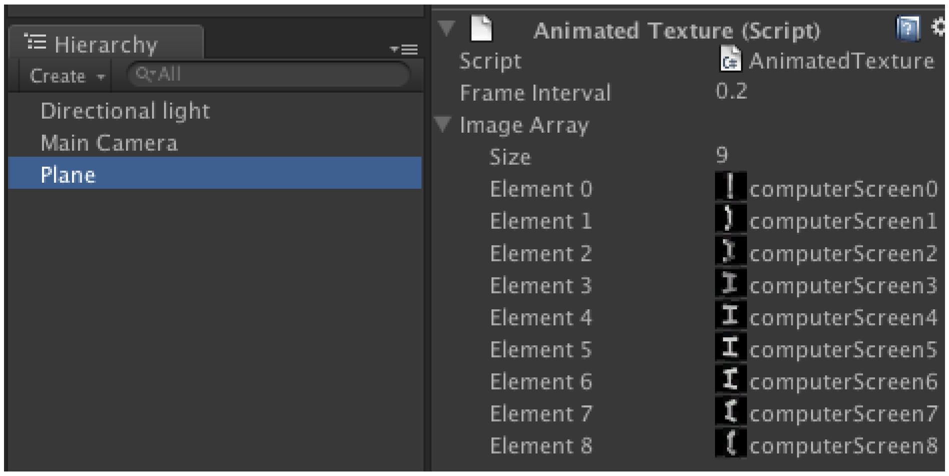
There are occasions when you want the material of an object to be animated, for example a simulated computer screen in a spaceship control room, or a clickable object acting as a button to which you wish to draw the player's attention. While Unity Pro offers one solution of an video that can play as a texture, often this is overkill (or not an option if you are using Unity free…). So another solution is to use code to change the texture of an object in real time…

Getting ready

You'll need a set of images that work as a sequence. It's usually a good idea to name them sequentially, e.g. computerScreen1.png, computerScreen2.png etc.

How to do it...

1. Start a new scene
2. Create a plane at (0,0,0), and set its rotation to (-90,0,0), so that it is squarely facing the camera
    - Position it at (0,0,0) and rotate it by (-90,0,0)
3. Create project folder *ScriptClasses* and in that folder create the C# script class *AnimatedTexture* (see Listing 1)
4. Add C# script class *AnimatedTexture* as a component of your plane scene object
5. Populate the public array with your sequence of images (you'll need to first set the *size* property of the array to the number of images)
6. To control the speed of the animation, in the Inspector choose an appropriate value for the *Frame Interval* public variable



Directional light
Main Camera
Plane

Animated Texture (Script)

Script	AnimatedTexture
Frame Interval	0.2
Image Array	
Size	9
Element 0	computerScreen0
Element 1	computerScreen1
Element 2	computerScreen2
Element 3	computerScreen3
Element 4	computerScreen4
Element 5	computerScreen5
Element 6	computerScreen6
Element 7	computerScreen7
Element 8	computerScreen8

0423_03_54.png

```
// file: AnimatedTexture.cs
using UnityEngine;
using System.Collections;

public class AnimatedTexture : MonoBehaviour
{
   public float frameInterval = 0.9f;
   public Texture2D[] imageArray;
   private int imageIndex = 0;

   private void Awake()
   {
      if (imageArray.Length < 1)
         Debug.LogError("no images in array!");
      else
         StartCoroutine( PlayAnimation() );
   }

   private IEnumerator PlayAnimation()
   {
      while( true )
      {
         ChangeImage();
         yield return new WaitForSeconds(changeInterval);

      }
   }

   private void ChangeImage()
   {
      imageIndex++;
```

```
        imageIndex = (imageIndex % imageArray.Length);
        Texture2D nextImage = imageArray[ imageIndex ];
        renderer.material.SetTexture("_MainTex", nextImage);
    }
}
```

How it works...

The array *imageArray* stores the sequence of images. The *Awake()* method checks there are some contents in the array, and starts the *PlayAnimation()* method as a coroutine.

Method *PlayAnimation()* runs an infinite loop, which calls the method to display the next image in the sequence, and then makes the *PlayAnimation()* wait for the number of seconds set in variable *frameInterval*.

Method *ChangeImage()* needs to find the next image in the sequence and make the material of the parent GameObject use that image. Instance variable *imageIndex* always stores the array index of the currently displaying image. By adding 1 to this index value, and then applying a modulus operation, the result is the next highest array index, or zero, if the last image has already been displayed. A local variable *nextImage* is set to store the next image to be displayed from the array, and the main texture of the material of the parent GameObject is set to this image, in the final statement of the method.

Materials rather than Texture2D images

If you prefer to work with materials rather than texture images, you could use the following code with an array of Materials named m*aterialArray*, and change the shared material of the render of the parent GameObject, as illustrate in the final statement of this method:

```
    private void ChangeMaterial()
    {
        materialIndex++;
        materialIndex = (materialIndex % materialArray.Length);
        Material nextMaterial = materialArray[ materialIndex ];
        renderer.sharedMaterial = nextMaterial;
    }
```

See also

Highlighting materials at mouse over

Disabling Culling for a material

When creating a transparent or semi-transparent object, we might want to see its internal faces. However, by default, Unity transparent shaders make them invisible. In this recipe, we will edit one of Unity's built-in transparent shaders in order to make those faces visible to the user.

Getting ready

In order to follow this recipe, you will need to download the source code to Unity's built-in Shaders at `http://unity3d.com/support/resources/assets/built-in-shaders`. You will also need a texture with a transparency channel. We have provided one, named *grid.tga*, in the folder 03_14.

How to do it...

1 – Open the archive containing the source code to Unity's built-in shaders and make a extract the file named *AlphaTest-BumpSpec.shader*, available within the folder *DefaultResources*, into your desktop.

2 – Rename the *AlphaTest-BumpSpec.shader* copy as *AlphaTest-DoubleSided.shader*. Then, open it in your code editor.

3 – Change the first line to:

```
   Shader "Transparent/Cutout/DoubleSided" {
```

4 – Add the following line of code where indicated.

```
SubShader {

Tags {"Queue"="AlphaTest" "IgnoreProjector"="True"
"RenderType"="TransparentCutout"}

   LOD 400
   // Line of code to be added:
   Cull Off
   // End of code to be added.
```

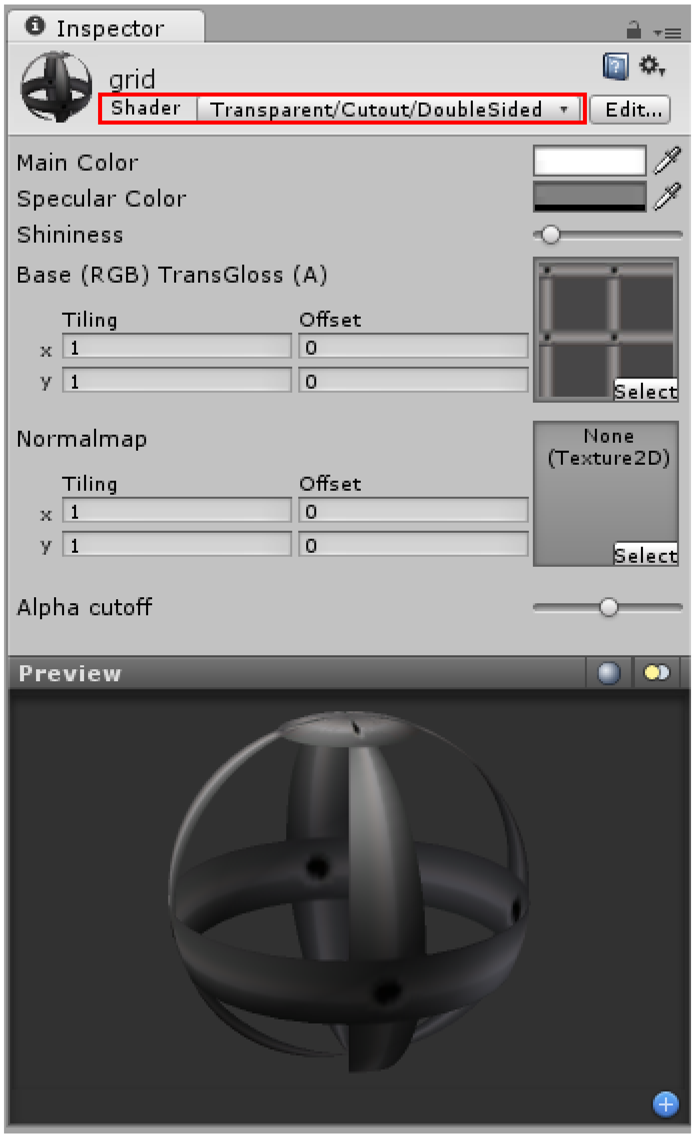
5 – Save the file and import it into your Unity Project.

6 – Import the texture *grid.tga*.

7 – Let's create a new material. Access the *Project* window, click on the *Create* drop-down menu and choose *Material*. Rename it *Grid*.

8 – Select the *Grid* material and, in the *Inspector* window, change its shader to *Transparent > Cutout > DoubleSided*.

9 – Apply the *grid* texture as the Base texture for the *Grid* material.



0423_03_55.png

10 – Your double-sided transparent material it's ready to use. It can even cast shadows!

How it works...

The `Cull Off` command makes Unity render both front and back faces of the object. Although it works fine for our *Cutout* shader, using it with other types of transparent shaders might cause lead to results.

There's more...

Shader documentation

Unity's documentation includes several articles on shader programming. You can check them out at `http://docs.unity3d.com/Documentation/Components/SL-Reference.html`

Another possible solution

You can get similar results by duplicating the faces of your 3d models, and then flipping normals of the new geometry.

See also

Creating transparency texture maps