OpenGL Shading Language Programming

Wikibooks.org

June 19, 2012

Contents

1. Introduction

1.0.1. About GLSL

GLSL (OpenGL Shading Language) is one of several commonly used shading languages for real-time rendering (other examples are Cg and HLSL). These shading languages are used to program shaders (i.e. more or less small programs) that are executed on a GPU (graphics processing unit), i.e. the processor of the graphics system of a computer – as opposed to the CPU (central processing unit) of a computer.

GPUs are massively parallel processors, which are extremely powerful. Most of today's real-time graphics in games and other interactive graphical applications would not be possible without GPUs. However, to take full advantage of the performance of GPUs, it is necessary to program them directly. This means that small programs (i.e. shaders) have to be written that can be executed by GPUs. The programming languages to write these shaders are shading languages. GLSL is one of them. In fact, it is the shading language of several 3D graphics APIs (application programming interfaces), namely OpenGL, OpenGL ES 2.x, and WebGL. Therefore, GLSL is commonly used in applications for desktop computers, mobile devices, and the web.

1.0.2. About this Wikibook

This wikibook was written with students in mind, who like neither programming nor mathematics. The basic motivation for this book is the observation that students are much more motivated to learn programming environments, programming languages and APIs if they are working on specific projects. Such projects are usually developed on specific platforms and therefore the approach of this book is to present GLSL within the game engine Unity.

Chapters 1 to 8 of the book consist of tutorials with working examples that produce certain effects. Note that these tutorials assume that you read them in the order in which they are presented, i.e. each tutorial will assume that you are familiar with

the concepts and techniques introduced by previous tutorials. If you are new to GLSL or Unity you should at least read through the tutorials in Chapter 1, “Basics”.

More details about the OpenGL pipeline and GLSL syntax in general are included in an “Appendix on the OpenGL Pipeline and GLSL Syntax”. Readers who are not familiar with OpenGL or GLSL might want to at least skim this part since a basic understanding of the OpenGL pipeline and GLSL syntax is very useful for understanding the tutorials.

1.0.3. About GLSL in Unity

GLSL programming in the game engine Unity is considerably easier than GLSL programming for an OpenGL, OpenGL ES, or WebGL application. Import of meshes and images (i.e. textures) is supported by a graphical user interface; mipmaps and normal maps can be computed automatically; the most common vertex attributes and uniforms are predefined; OpenGL states can be set by very simple commands; etc.

A free version of Unity can be downloaded for Windows and MacOS at UNITY'S DOWNLOAD PAGE[1]. All of the included tutorials work with the free version. Three points should be noted:

- First, **Windows users** have to use the command-line argument `-force-opengl` HTTP://UNITY3D.COM/SUPPORT/DOCUMENTATION/MANUAL/COMMAND%20LINE%20ARGU when starting Unity in order to be able to use GLSL shaders; for example, by changing the `Target` setting in the properties of the desktop icon to: `"C:\Program Files\Unity\Editor\Unity.exe" -force-opengl` . (On MacOS X, OpenGL and therefore GLSL is used by default.)
- Secondly, this book assumes that readers are somewhat familiar with Unity. If this is not the case, readers should consult the first three sections of Unity's User Guide HTTP://UNITY3D.COM/SUPPORT/DOCUMENTATION/MANUAL/USER%20GUIDE.HTML[3] (Unity Basics, Building Scenes, Asset Import and Creation).
- Furthermore, as of version 3.5, Unity supports a version of GLSL similar to version 1.0.x for OpenGL ES 2.0 (the specification is available at the “KHRONOS

1 HTTP://UNITY3D.COM/UNITY/DOWNLOAD/
2 HTTP://UNITY3D.COM/SUPPORT/DOCUMENTATION/MANUAL/COMMAND%20LINE%20ARGUMENTS.HTML
3 HTTP://UNITY3D.COM/SUPPORT/DOCUMENTATION/MANUAL/USER%20GUIDE.HTML

OpenGL ES API Registry"[4]); however, Unity's shader documentation http://unity3d.com/support/documentation/Components/SL-Reference.html[5] focuses on shaders written in Unity's own "surface shader" format and Cg/HLSL http://unity3d.com/support/documentation/Components/SL-ShaderPrograms.html[6]. There are only very few details documented that are specific to GLSL shaders http://unity3d.com/support/documentation/Components/SL-GLSLShaderPrograms.html[7]. Thus, this wikibook might also help to close some gaps in Unity's documentation. However, optimizations (see, for example, this blog[8]) are usually not discussed.

Martin Kraus, May 2012

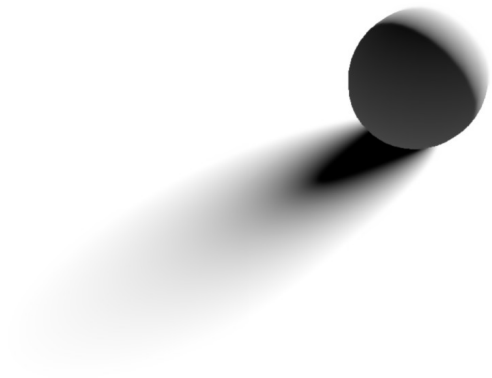

Figure 1

4 http://www.khronos.org/registry/gles/

5 http://unity3d.com/support/documentation/Components/SL-Reference.html

6 http://unity3d.com/support/documentation/Components/SL-ShaderPrograms.html

7 http://unity3d.com/support/documentation/Components/SL-GLSLShaderPrograms.html

8 http://aras-p.info/blog/2011/02/01/ios-shader-tricks-or-its-2001-all-over-a

Part I.

Basics

2. Minimal Shader

This tutorial covers the basic steps to create a minimal GLSL shader in Unity.

2.0.4. Starting Unity and Creating a New Project

After downloading and starting Unity (Windows users have to use the command-line argument `-force-opengl`), you might see an empty project. If not, you should create a new project by choosing **File > New Project...** from the menu. For this tutorial, you don't need to import any packages but some of the more advanced tutorials require the scripts and skyboxes packages. After creating a new project on Windows, Unity might start without OpenGL support; thus, Windows users should always quit Unity and restart it (with the command-line argument `-force-opengl`) after creating a new project. Then you can open the new project with **File > Open Project...** from the menu.

If you are not familiar with Unity's Scene View, Hierachy View, Project View and Inspector View, now would be a good time to read the first two (or three) sections ("Unity Basics" and "Building Scenes") of the UNITY USER GUIDE[1].

2.0.5. Creating a Shader

Creating a GLSL shader is not complicated: In the **Project View**, click on **Create** and choose **Shader**. A new file named "NewShader" should appear in the Project View. Double-click it to open it (or right-click and choose **Open**). An editor with the default shader in Cg should appear. Delete all the text and copy & paste the following shader into this file:

```
Shader "GLSL basic shader" { // defines the name of the shader
   SubShader { // Unity chooses the subshader that fits the GPU best
      Pass { // some shaders require multiple passes
         GLSLPROGRAM // here begins the part in Unity's GLSL
```

1 HTTP://UNITY3D.COM/SUPPORT/DOCUMENTATION/MANUAL/USER%20GUIDE.HTML

```
         #ifdef VERTEX // here begins the vertex shader

         void main() // all vertex shaders define a main() function
         {
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
               // this line transforms the predefined attribute
               // gl_Vertex of type vec4 with the predefined
               // uniform gl_ModelViewProjectionMatrix of type mat4
               // and stores the result in the predefined output
               // variable gl_Position of type vec4.
         }

         #endif // here ends the definition of the vertex shader


         #ifdef FRAGMENT // here begins the fragment shader

         void main() // all fragment shaders define a main() function
         {
            gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
               // this fragment shader just sets the output color
               // to opaque red (red = 1.0, green = 0.0, blue = 0.0,
               // alpha = 1.0)
         }

         #endif // here ends the definition of the fragment shader

         ENDGLSL // here ends the part in GLSL
      }
   }
}
```

Save the shader (by clicking the save icon or choosing **File > Save** from the editor's menu).

Congratulations, you have just created a shader in Unity. If you want, you can rename the shader file in the Project View by clicking the name, typing a new name, and pressing Return. (After renaming, reopen the shader in the editor to make sure that you are editing the correct file.)

Unfortunately, there isn't anything to see until the shader is attached to a material.

2.0.6. Creating a Material and Attaching a Shader

To create a material, go back to Unity and create a new material by clicking **Create** in the **Project View** and choosing **Material**. A new material called “New Material” should appear in the Project View. (You can rename it just like the shader.) If it isn't selected, select it by clicking. Details about the material appear now in the Inspector View. In order to set the shader to this material, you can either

- drag & drop the shader in the **Project View** over the material or
- select the material in the **Project View** and then in the **Inspector View** choose the shader (in this case "GLSL basic shader" as specified in the shader code above) from the drop-down list labeled **Shader**.

In either case, the Preview in the Inspector View of the material should now show a red sphere. If it doesn't and an error message is displayed at the bottom of the Unity window, you should reopen the shader and check in the editor whether the text is the same as given above. Windows users should make sure that OpenGL is supported by restarting Unity with the command-line argument `-force-opengl`.

2.0.7. Interactively Editing Shaders

This would be a good time to play with the shader; in particular, you can easily change the computed fragment color. Try neon green by opening the shader and replacing the fragment shader with this code:

```
#ifdef FRAGMENT

void main()
{
   gl_FragColor = vec4(0.6, 1.0, 0.0, 1.0);
      // red = 0.6, green = 1.0, blue = 0.0, alpha = 1.0
}

#endif
```

You have to save the code in the editor and activate the Unity window again to apply the new shader. If you select the material in the Project View, the sphere in the Inspector View should now be green. You could also try to modify the red, green, and blue components to find the warmest orange or the darkest blue. (Actually, there is a MOVIE[2] about finding the warmest orange and ANOTHER[3] about dark blue that is almost black.)

You could also play with the vertex shader, e.g. try this vertex shader:

```
#ifdef VERTEX

void main()
{
   gl_Position = gl_ModelViewProjectionMatrix
   * (vec4(1.0, 0.1, 1.0, 1.0) * gl_Vertex);
}
```

2 HTTP://EN.WIKIPEDIA.ORG/WIKI/SIAM%20SUNSET
3 HTTP://EN.WIKIPEDIA.ORG/WIKI/AZULOSCUROCASINEGRO

```
#endif
```

This flattens any input geometry by multiplying the *y* coordinate with 0.1. (This is a component-wise vector product; for more information on vectors and matrices in GLSL see the discussion in GLSL PROGRAMMING/VECTOR AND MATRIX OPERATIONS[4].)

In case the shader does not compile, Unity displays an error message at the bottom of the Unity window and displays the material as bright magenta. In order to see all error messages and warnings, you should select the shader in the **Project View** and read the messages in the **Inspector View**, which also include line numbers, which you can display in the text editor by choosing **View > Line Numbers** in the text editor menu. You could also open the Console View by choosing **Window > Console** from the menu, but this will not display all error messages and therefore the crucial error is often not reported.

2.0.8. Attaching a Material to a Game Object

We still have one important step to go: attaching the new material to a triangle mesh. To this end, create a sphere (which is one of the predefined game objects of Unity) by choosing **GameObject > Create Other > Sphere** from the menu. A sphere should appear in the Scene View and the label "Sphere" should appear in the Hierarchy View. (If it doesn't appear in the Scene View, click it in the Hierarchy View, move (without clicking) the mouse over the Scene View and press "f". The sphere should now appear centered in the Scene View.)

To attach the material to the new sphere, you can:

- drag & drop the material from the **Project View** over the sphere in the **Hierarchy View** or
- drag & drop the material from the **Project View** over the sphere in the **Scene View** or
- select the sphere in the **Hierarchy View**, locate the **Mesh Renderer** component in the **Inspector View** (and open it by clicking the title if it isn't open), open the **Materials** setting of the Mesh Renderer by clicking it. Change the "Default-Diffuse" material to the new material by clicking the dotted circle icon to the right of the material name and choosing the new material from the pop-up window.

In any case, the sphere in the Scene View should now have the same color as the preview in the Inspector View of the material. Changing the shader should (after

4 Chapter 44 on page 421

saving and switching to Unity) change the appearance of the sphere in the Scene View.

2.0.9. Saving Your Work in a Scene

There is one more thing: you should save you work in a "scene" (which often corresponds to a game level). Choose **File > Save Scene** (or **File > Save Scene As...**) and choose a file name in the "Assets" directory of your project. The scene file should then appear in the Project View and will be available the next time you open the project.

2.0.10. One More Note about Terminology

It might be good to clarify the terminology. In GLSL, a "shader" is either a vertex shader or a fragment shader. The combination of both is called a "program".

Unfortunately, Unity refers to this kind of program as a "shader", while in Unity a vertex shader is called a "vertex program" and a fragment shader is called a "fragment program".

To make the confusion perfect, I'm going to use Unity's word "shader" for a GLSL program, i.e. the combination of a vertex and a fragment shader. However, I will use the GLSL terms "vertex shader" and "fragment shader" instead of "vertex program" and "fragment program".

2.0.11. Summary

Congratulations, you have reached the end of this tutorial. A few of the things you have seen are:

- How to create a shader.
- How to define a GLSL vertex and fragment shader in Unity.
- How to create a material and attach a shader to the material.
- How to manipulate the ouput color `gl_FragColor` in the fragment shader.
- How to transform the input attribute `gl_Vertex` in the vertex shader.
- How to create a game object and attach a material to it.

Actually, this was quite a lot of stuff.

2.0.12. Further Reading

If you still want to know more

- about vertex and fragment shaders in general, you should read the description in GLSL PROGRAMMING/OPENGL ES 2.0 PIPELINE[5].
- about the vertex transformations such as `gl_ModelViewProjectionMatrix`, you should read GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[6].
- about handling vectors (e.g. the `vec4` type) and matrices in GLSL, you should read GLSL PROGRAMMING/VECTOR AND MATRIX OPERATIONS[7].
- about how to apply vertex transformations such as `gl_ModelViewProjectionMatrix`, you should read GLSL PROGRAMMING/APPLYING MATRIX TRANSFORMATIONS[8].
- about Unity's ShaderLab language for specifying shaders, you should read UNITY'S SHADERLAB REFERENCE[9].

5 Chapter 42 on page 399
6 Chapter 43 on page 407
7 Chapter 44 on page 421
8 Chapter 45 on page 429
9 HTTP://UNITY3D.COM/SUPPORT/DOCUMENTATION/COMPONENTS/SL-REFERENCE.HTML

3. RGB Cube

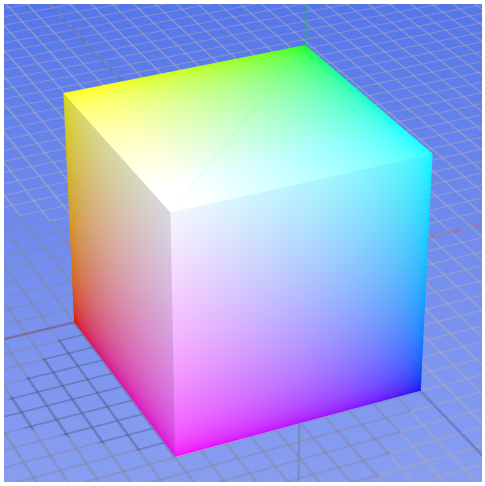

Figure 2 An RGB cube: the x, y, z coordinates are mapped to red, green, and blue color components.

This tutorial introduces **varying variables**. It is based on GLSL PROGRAMMING/UNITY/MINIMAL SHADER[1].

In this tutorial we will write a shader to render an RGB cube similar to the one shown below . The color of each point on the surface is determined by its coordinates; i.e., a point at position (x,y,z) has the color $(\text{red},\text{green},\text{blue}) = (x,y,z)$. For example, the point $(x,y,z) = (0,0,1)$ is mapped to the color $(\text{red},\text{green},\text{blue}) = (0,0,1)$, i.e. pure blue. (This is the blue corner in the lower right of the figure below .)

1 Chapter 2 on page 9

3.0.13. Preparations

Since we want to create an RGB cube, you first have to create a cube game object. As described in GLSL PROGRAMMING/UNITY/MINIMAL SHADER[2] for a sphere, you can create a cube game object by selecting **GameObject > Create Other > Cube** from the main menu. Continue with creating a material and a shader object and attaching the shader to the material and the material to the cube as described in GLSL PROGRAMMING/UNITY/MINIMAL SHADER[3].

3.0.14. The Shader Code

Here is the shader code, which you should copy & paste into your shader object:

```
Shader "GLSL shader for RGB cube" {
   SubShader {
      Pass {
         GLSLPROGRAM

         #ifdef VERTEX // here begins the vertex shader

         varying vec4 position;
            // this is a varying variable in the vertex shader

         void main()
         {
            position = gl_Vertex + vec4(0.5, 0.5, 0.5, 0.0);
               // Here the vertex shader writes output data
               // to the varying variable. We add 0.5 to the
               // x, y, and z coordinates, because the
               // coordinates of the cube are between -0.5 and
               // 0.5 but we need them between 0.0 and 1.0.
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif // here ends the vertex shader

         #ifdef FRAGMENT // here begins the fragment shader

         varying vec4 position;
            // this is a varying variable in the fragment shader

         void main()
         {
            gl_FragColor = position;
               // Here the fragment shader reads intput data
               // from the varying variable. The red, gree, blue,
               // and alpha component of the fragment color are
```

2 Chapter 2 on page 9
3 Chapter 2 on page 9

```
                // set to the values in the varying variable.
            }

            #endif // here ends the fragment shader

            ENDGLSL
        }
    }
}
```

If your cube is not colored correctly, check the console for error messages (by selecting **Window > Console** from the main menu), make sure you have saved the shader code, and check whether you have attached the shader object to the material object and the material object to the game object.

3.0.15. Varying Variables

The main task of our shader is to set the output fragment color (`gl_FragColor`) in the fragment shader to the position (`gl_Vertex`) that is available in the vertex shader. Actually, this is not quite true: the coordinates in `gl_Vertex` for Unity's default cube are between -0.5 and +0.5 while we would like to have color components between 0.0 and 1.0; thus, we need to add 0.5 to the x, y, and z component, which is done by this expression: `gl_Vertex + vec4(0.5, 0.5, 0.5, 0.0)`.

The main problem, however, is: how do we get any value from the vertex shader to the fragment shader? It turns out that the **only** way to do this is to use varying variables (or varyings for short). Output of the vertex shader can be written to a varying variable and then it can be read as input by the fragment shader. This is exactly what we need.

To specify a varying variable, it has to be defined with the modifier `varying` (before the type) in the vertex and the fragment shader outside of any function; in our example: `varying vec4 position;`. And here comes the most important rule about varying variables: **The type and name of a varying variable definition in the vertex shader has to match exactly the type and name of a varying variable definition in the fragment shader and vice versa.** This is required to avoid ambiguous cases where the GLSL compiler cannot figure out which varying variable of the vertex shader should be matched to which varying variable of the fragment shader.

3.0.16. A Neat Trick for Varying Variables in Unity

The requirement that the definitions of varying variables in the vertex and fragment shader match each other often results in errors, for example if a programmer changes a type or name of a varying variable in the vertex shader but forgets to change it in the fragment shader. Fortunately, there is a nice trick in Unity that avoids the problem. Consider the following shader:

```
Shader "GLSL shader for RGB cube" {
   SubShader {
      Pass {
         GLSLPROGRAM // here begin the vertex and the fragment shader

         varying vec4 position;
            // this line is part of the vertex and the fragment shader

         #ifdef VERTEX
            // here begins the part that is only in the vertex shader

         void main()
         {
            position = gl_Vertex + vec4(0.5, 0.5, 0.5, 0.0);
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif
            // here ends the part that is only in the vertex shader

         #ifdef FRAGMENT
            // here begins the part that is only in the fragment shader

         void main()
         {
            gl_FragColor = position;
         }

         #endif
            // here ends the part that is only in the fragment shader

         ENDGLSL // here end the vertex and the fragment shader
      }
   }
}
```

As the comments in this shader explain, the line `#ifdef VERTEX` doesn't actually mark the beginning of the vertex shader but the beginning of a part that is **only** in the vertex shader. Analogously, `#ifdef FRAGMENT` marks the beginning of a part that is only in the fragment shader. In fact, both shaders begin with the line `GLSLPROGRAM`. Therefore, any code between `GLSLPROGRAM` and the first `#ifdef` line will be shared by the vertex and the fragment shader. (If you are familiar with the C or C++ preprocessor, you might have guessed this already.)

This is perfect for definitions of varying variables because it means that we may type the definition only once and it will be put into the vertex and the fragment shader; thus, matching definitions are guaranteed! I.e. we have to type less and there is no way to produce compiler errors because of mismatches between the definitions of varying variables. (Of course, the cost is that we have to type all these `#ifdef` and `#end` lines.)

3.0.17. Variations of this Shader

The RGB cube represents the set of available colors (i.e. the gamut of the display). Thus, it can also be used show the effect of a color transformation. For example, a color to gray transformation would compute either the mean of the red, green, and blue color components, i.e. $(\text{red} + \text{green} + \text{blue})/3$, and then put this value in all three color components of the fragment color to obtain a gray value of the same intensity. Instead of the mean, the relative luminance could also be used, which is $0.21\ \text{red} + 0.72\ \text{green} + 0.07\ \text{blue}$. Of course, any other color transformation (changing saturation, contrast, hue, etc.) is also applicable.

Another variation of this shader could compute a CMY (cyan, magenta, yellow) cube: for position (x, y, z) you could subtract from a pure white an amount of red that is proportional to x in order to produce cyan. Furthermore, you would subtract an amount of green in proportion to the y component to produce magenta and also an amount of blue in proportion to z to produce yellow.

If you really want to get fancy, you could compute an HSV (hue, saturation, value) cylinder. For x and z coordinates between -0.5 and +0.5, you can get an angle H between 0 and 360° with `180.0+degrees(atan(z, x))` in GLSL and a distance S between 0 and 1 from the y axis with `2.0 * sqrt(x * x + z * z)`. The y coordinate for Unity's built-in cylinder is between -1 and 1 which can be translated to a value V between 0 and 1 by $(y + 1.0)/2.0$. The computation of RGB colors from HSV coordinates is described in the ARTICLE ON HSV IN WIKIPEDIA[4].

3.0.18. Interpolation of Varying Variables

The story about varying variables is not quite over yet. If you select the cube game object, you will see in the Scene View that it consists of only 12 triangles and 8 vertices. Thus, the vertex shader might be called only eight times and only eight

4 HTTP://EN.WIKIPEDIA.ORG/WIKI/HSL%20AND%20HSV%23FROM%20HSV

different outputs are written to the varying variable. However, there are many more colors on the cube. How did that happen?

The answer is implied by the name **varying** variables. They are called this way because they vary across a triangle. In fact, the vertex shader is only called for each vertex of each triangle. If the vertex shader writes different values to a varying variable for different vertices, the values are interpolated across the triangle. The fragment shader is then called for each pixel that is covered by the triangle and receives interpolated values of the varying variables. The details of this interpolation are described in GLSL PROGRAMMING/RASTERIZATION[5].

If you want to make sure that a fragment shader receives one exact, non-interpolated value by a vertex shader, you have to make sure that the vertex shader writes the same value to the varying variable for all vertices of a triangle.

3.0.19. Summary

And this is the end of this tutorial. Congratulations! Among other things, you have seen:

- What an RGB cube is.
- What varying variables are good for and how to define them.
- How to make sure that a varying variable has the same name and type in the vertex shader and the fragment shader.
- How the values written to a varying variable by the vertex shader are interpolated across a triangle before they are received by the fragment shader.

3.0.20. Further Reading

If you want to know more

- about the data flow in and out of vertex and fragment shaders, you should read the description in GLSL PROGRAMMING/OPENGL ES 2.0 PIPELINE[6].
- about vector and matrix operations (e.g. the expression `gl_Vertex + vec4(0.5, 0.5, 0.5, 0.0);`), you should read GLSL PROGRAMMING/VECTOR AND MATRIX OPERATIONS[7].

5 Chapter 46 on page 437
6 Chapter 42 on page 399
7 Chapter 44 on page 421

- about the interpolation of varying variables, you should read GLSL PROGRAMMING/RASTERIZATION[8].
- about Unity's official documentation of writing vertex shaders and fragment shaders in Unity's ShaderLab, you should read UNITY'S SHADERLAB REFERENCE ABOUT "GLSL SHADER PROGRAMS"[9].

8 Chapter 46 on page 437

9 HTTP://UNITY3D.COM/SUPPORT/DOCUMENTATION/COMPONENTS/SL-GLSLSHADERPROGRAMS.HTML

4. Debugging of Shaders

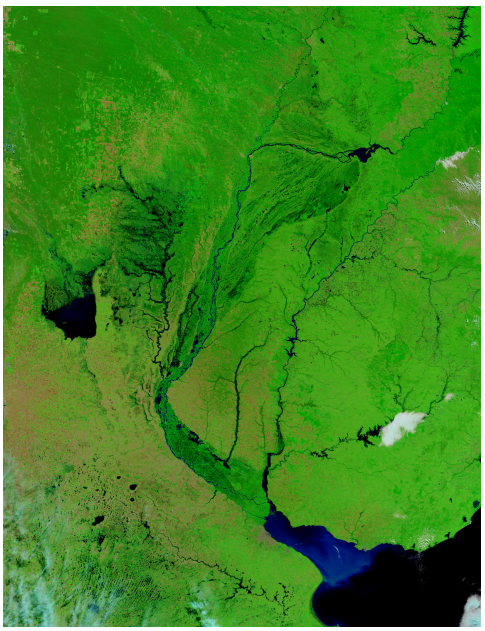

Figure 3 A false-color satellite image.

This tutorial introduces **attribute variables**. It is based on GLSL PROGRAMMING/UNITY/MINIMAL SHADER[1] and GLSL PROGRAMMING/UNITY/RGB CUBE[2].

This tutorial also introduces the main technique to debug shaders in Unity: false-color images, i.e. a value is visualized by setting one of the components of the fragment color to it. Then the intensity of that color component in the resulting image allows you to make conclusions about the value in the shader. This might appear to be a very primitive debugging technique because it is a very primitive debugging technique. Unfortunately, there is no alternative in Unity.

1 Chapter 2 on page 9
2 Chapter 3 on page 15

4.0.21. Where Does the Vertex Data Come from?

In GLSL PROGRAMMING/UNITY/RGB CUBE[3] you have seen how the fragment shader gets its data from the vertex shader by means of varying variables. The question here is: where does the vertex shader get its data from? Within Unity, the answer is that the Mesh Renderer component of a game object sends all the data of the mesh of the game object to OpenGL in each frame. (This is often called a "draw call". Note that each draw call has some performance overhead; thus, it is much more efficient to send one large mesh with one draw call to OpenGL than to send several smaller meshes with multiple draw calls.) This data usually consists of a long list of triangles, where each triangle is defined by three vertices and each vertex has certain attributes, including position. These attributes are made available in the vertex shader by means of attribute variables.

4.0.22. Built-in Attribute Variables and how to Visualize Them

In Unity, most of the standard attributes (position, color, surface normal, and texture coordinates) are built in, i.e. you need not (in fact must not) define them. The names of these built-in attributes are actually defined by the OpenGL "compability profile" because such built-in names are needed if you mix an OpenGL application that was written for the fixed-function pipeline with a (programmable) vertex shader. If you had to define them, the definitions (only in the vertex shader) would look like this:

```
attribute vec4 gl_Vertex; // position (in object coordinates,
   // i.e. local or model coordinates)
attribute vec4 gl_Color; // color (usually constant)
attribute vec3 gl_Normal; // surface normal vector
   // (in object coordinates; usually normalized to unit length)
attribute vec4 gl_MultiTexCoord0; //0th set of texture coordinates
   // (a.k.a. "UV"; between 0 and 1)
attribute vec4 gl_MultiTexCoord1; //1st set of texture coordinates
   // (a.k.a. "UV"; between 0 and 1)
...
```

There is only one attribute variable that is provided by Unity but has no standard name in OpenGL, namely the tangent vector, i.e. a vector that is orthogonal to the surface normal. You should define this variable yourself as an attribute variable of type `vec4` with the specific name `Tangent` as shown in the following shader:

```
Shader "GLSL shader with all built-in attributes" {
   SubShader {
      Pass {
```

3 Chapter 3 on page 15

```
         GLSLPROGRAM

         varying vec4 color;

         #ifdef VERTEX

         attribute vec4 Tangent; // this attribute is specific to Unity

         void main()
         {
            color = gl_MultiTexCoord0; // set the varying variable

            // other possibilities to play with:

            // color = gl_Vertex;
            // color = gl_Color;
            // color = vec4(gl_Normal, 1.0);
            // color = gl_MultiTexCoord0;
            // color = gl_MultiTexCoord1;
            // color = Tangent;

            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif

         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor = color; // set the output fragment color
         }

         #endif

         ENDGLSL
      }
   }
}
```

In GLSL PROGRAMMING/UNITY/RGB CUBE[4] we have already seen, how to visualize the `gl_Vertex` coordinates by setting the fragment color to those values. In this example, the fragment color is set to `gl_MultiTexCoord0` such that we can see what kind of texture coordinates Unity provides.

Note that only the first three components of `Tangent` represent the tangent direction. The scaling and the fourth component are set in a specific way, which is mainly useful for parallax mapping (see GLSL PROGRAMMING/UNITY/PROJECTION OF BUMPY SURFACES[5]).

4 Chapter 3 on page 15
5 Chapter 22 on page 193

4.0.23. How to Interpret False-Color Images

When trying to understand the information in a false-color image, it is important to focus on one color component only. For example, if the standard attribute `gl_MultiTexCoord0` for a sphere is written to the fragment color then the red component of the fragment visualizes the `x` coordinate of `gl_MultiTexCoord0`, i.e. it doesn't matter whether the output color is maximum pure red or maximum yellow or maximum magenta, in all cases the red component is 1. On the other hand, it also doesn't matter for the red component whether the color is blue or green or cyan of any intensity because the red component is 0 in all cases. If you have never learned to focus solely on one color components, this is probably quite challenging; therefore, you might consider to look only at one color component at a time. For example by using this line to set the varying in the vertex shader:

```
color = vec4(gl_MultiTexCoord0.x, 0.0, 0.0, 1.0);
```

This sets the red component of the varying variable to the `x` component of `gl_MultiTexCoord0` but sets the green and blue components to 0 (and the alpha or opacity component to 1 but that doesn't matter in this shader).

If you focus on the red component or visualize only the red component you should see that it increases from 0 to 1 as you go around the sphere and after 360° drops to 0 again. It actually behaves similar to a longitude coordinate on the surface of a planet. (In terms of spherical coordinates, it corresponds to the azimuth.)

If the `x` component of `gl_MultiTexCoord0` corresponds to the longitude, one would expect that the `y` component would correspond to the latitude (or the inclination in spherical coordinates). However, note that texture coordinates are always between 0 and 1; therefore, the value is 0 at the bottom (south pole) and 1 at the top (north pole). You can visualize the `y` component as green on its own with:

```
color = vec4(0.0, gl_MultiTexCoord0.y, 0.0, 1.0);
```

Texture coordinates are particularly nice to visualize because they are between 0 and 1 just like color components are. Almost as nice are coordinates of normalized vectors (i.e., vectors of length 1; for example, `gl_Normal` is usually normalized) because they are always between -1 and +1. To map this range to the range from 0 to 1, you add 1 to each component and divide all components by 2, e.g.:

```
color = vec4((gl_Normal + vec3(1.0, 1.0, 1.0)) / 2.0, 1.0);
```

Note that `gl_Normal` is a three-dimensional vector. Black corresponds then to the coordinate -1 and full intensity of one component to the coordinate +1.

If the value that you want to visualize is in another range than 0 to 1 or -1 to +1, you have to map it to the range from 0 to 1, which is the range of color components. If you don't know which values to expect, you just have to experiment. What helps here is that if you specify color components outside of the range 0 to 1, they are automatically clamped to this range. I.e., values less than 0 are set to 0 and values greater than 1 are set to 1. Thus, when the color component is 0 or 1 you know at least that the value is less or greater than what you assumed and then you can adapt the mapping iteratively until the color component is between 0 and 1.

4.0.24. Debugging Practice

In order to practice the debugging of shaders, this section includes some lines that produce black colors when the assignment to `color` in the vertex shader is replaced by each of them. Your task is to figure out for each line, why the result is black. To this end, you should try to visualize any value that you are not absolutely sure about and map the values less than 0 or greater than 1 to other ranges such that the values are visible and you have at least an idea in which range they are. Note that most of the functions and operators are documented in GLSL PROGRAMMING/VECTOR AND MATRIX OPERATIONS[6].

```
color = gl_MultiTexCoord0 - vec4(1.5, 2.3, 1.1, 0.0);

color = vec4(gl_MultiTexCoord0.z);

color = gl_MultiTexCoord0 / tan(0.0);
```

The following lines require some knowledge about the dot and cross product:

```
color = dot(gl_Normal, vec3(Tangent)) * gl_MultiTexCoord0;

color = dot(cross(gl_Normal, vec3(Tangent)), gl_Normal) *
   gl_MultiTexCoord0;

color = vec4(cross(gl_Normal, gl_Normal), 1.0);

color = vec4(cross(gl_Normal, vec3(gl_Vertex)), 1.0);
   // only for a sphere!
```

Do the functions `radians()` and `noise()` always return black? What's that good for?

6 Chapter 44 on page 421

```
color = radians(gl_MultiTexCoord0);

color = noise4(gl_MultiTexCoord0);
```

Consult the documentation in the "OpenGL ES Shading Language 1.0.17 Specification" available at the "KHRONOS OPENGL ES API REGISTRY"[7] to figure out what `radians()` is good for and what the problem with `noise4()` is.

4.0.25. Special Variables in the Fragment Shader

Attributes are specific to vertices, i.e., they usually have different values for different vertices. There are similar variables for fragment shaders, i.e., variables that have different values for each fragment. However, they are different from attributes because they are not specified by a mesh (i.e. a list of triangles). They are also different from varyings because they are not set explicitly by the vertex shader.

Specifically, a four-dimensional vector `gl_FragCoord` is available containing the screen (or: window) coordinates $(x, y, z, 1/w)$ of the fragment that is processed; see GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[8] for the description of the screen coordinate system.

Moreover, a boolean variable `gl_FrontFacing` is provided that specifies whether the front face or the back face of a triangle is being rendered. Front faces usually face the "outside" of a model and back faces face the "inside" of a model; however, there is no clear outside or inside if the model is not a closed surface. Usually, the surface normal vectors point in the direction of the front face, but this is not required. In fact, front faces and back faces are specified by the order of the vertex triangles: if the vertices appear in counter-clockwise order, the front face is visible; if they appear in clockwise order, the back face is visible. An application is shown in GLSL PROGRAMMING/UNITY/CUTAWAYS[9].

4.0.26. Summary

Congratulations, you have reached the end of this tutorial! We have seen:

- The list of built-in attributes in Unity: `gl_Vertex`, `gl_Color`, `gl_Normal`, `gl_MultiTexCoord0`, `gl_MultiTexCoord1`, and the special `Tangent`.

7 HTTP://WWW.KHRONOS.ORG/REGISTRY/GLES/
8 Chapter 43 on page 407
9 Chapter 6 on page 43

- How to visualize these attributes (or any other value) by setting components of the output fragment color.
- The two additional special variables that are available in fragment programs: `gl_FragCoord` and `gl_FrontFacing`.

4.0.27. Further Reading

If you still want to know more

- about the data flow in vertex and fragment shaders, you should read the description in GLSL PROGRAMMING/OPENGL ES 2.0 PIPELINE[10].
- about operations and functions for vectors, you should read GLSL PROGRAMMING/VECTOR AND MATRIX OPERATIONS[11].

10 Chapter 42 on page 399
11 Chapter 44 on page 421

5. Shading in World Space

Figure 4 Some chameleons are able to change their color according to the world around them.

This tutorial introduces **uniform variables**. It is based on GLSL PROGRAMMING/UNITY/MINIMAL SHADER[1], GLSL PROGRAMMING/UNITY/RGB CUBE[2], and GLSL PROGRAMMING/UNITY/DEBUGGING OF SHADERS[3].

In this tutorial we will look at a shader that changes the fragment color depending on its position in the world. The concept is not too complicated; however, there are extremely important applications, e.g. shading with lights and environment maps. We will also have a look at shaders in the real world; i.e., what is necessary to enable non-programmers to use your shaders?

1 Chapter 2 on page 9
2 Chapter 3 on page 15
3 Chapter 4 on page 23

5.0.28. Transforming from Object to World Space

As mentioned in GLSL PROGRAMMING/UNITY/DEBUGGING OF SHADERS[4], the attribute `gl_Vertex` specifies object coordinates, i.e. coordinates in the local object (or model) space of a mesh. The object space (or object coordinate system) is specific to each game object; however, all game objects are transformed into one common coordinate system — the world space.

If a game object is put directly into the world space, the object-to-world transformation is specified by the Transform component of the game object. To see it, select the object in the **Scene View** or the **Hierarchy View** and then find the Transform component in the **Inspector View**. There are parameters for "Position", "Rotation" and "Scale" in the Transform component, which specify how vertices are transformed from object coordinates to world coordinates. (If a game object is part of a group of objects, which is shown in the Hierarchy View by means of indentation, then the Transform component only specifies the transformation from object coordinates of a game object to the object coordinates of the parent. In this case, the actual object-to-world transformation is given by the combination of the transformation of a object with the transformations of its parent, grandparent, etc.) The transformations of vertices by translations, rotations and scalings, as well as the combination of transformations and their representation as 4×4 matrices are discussed in GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[5].

Back to our example: the transformation from object space to world space is put into a 4×4 matrix, which is also known as "model matrix" (since this transformation is also known as "model transformation"). This matrix is available in the uniform variable `_Object2World`, which is defined and used in the following shader:

```
Shader "GLSL shading in world space" {
   SubShader {
      Pass {
         GLSLPROGRAM

         uniform mat4 _Object2World;
            // definition of a Unity-specific uniform variable

         varying vec4 position_in_world_space;

         #ifdef VERTEX

         void main()
         {
            position_in_world_space = _Object2World * gl_Vertex;
```

4 Chapter 4 on page 23
5 Chapter 43 on page 407

```
            // transformation of gl_Vertex from object coordinates
            // to world coordinates;

         gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
      }

      #endif

      #ifdef FRAGMENT

      void main()
      {
         float dist = distance(position_in_world_space,
            vec4(0.0, 0.0, 0.0, 1.0));
            // computes the distance between the fragment position
            // and the origin (the 4th coordinate should always be
            // 1 for points).

         if (dist < 5.0)
         {
            gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0);
               // color near origin
         }
         else
         {
            gl_FragColor = vec4(0.3, 0.3, 0.3, 1.0);
               // color far from origin
         }
      }

      #endif

      ENDGLSL
   }
  }
}
```

Note that this shader makes sure that the definition of the uniform is included in both the vertex and the fragment shader (although this particular fragment shader doesn't need it). This is similar to the definition of varyings discussed in GLSL PROGRAMMING/UNITY/RGB CUBE[6].

Usually, an OpenGL application has to set the value of uniform variables; however, Unity takes care of always setting the correct value of predefined uniform variables such as `_Object2World`; thus, we don't have to worry about it.

This shader transforms the vertex position to world space and gives it to the fragment shader in a varying. For the fragment shader the varying variable contains the interpolated position of the fragment in world coordinates. Based on the distance of this position to the origin of the world coordinate system, one of two colors is set.

6 Chapter 3 on page 15

Thus, if you move an object with this shader around in the editor it will turn green near the origin of the world coordinate system. Farther away from the origin it will turn dark grey.

5.0.29. More Unity-Specific Uniforms

There are, in fact, several predefined uniform variables similar to `_Object2World`. Here is a short list (including `_Object2World`), which appears in the shader codes of several tutorials:

```
// The following built-in uniforms (except _LightColor0 and
// _LightMatrix0) are also defined in "UnityCG.glslinc",
// i.e. one could #include "UnityCG.glslinc"
uniform vec4 _Time, _SinTime, _CosTime; // time values from Unity
uniform vec4 _ProjectionParams;
   // x = 1 or -1 (-1 if projection is flipped)
   // y = near plane; z = far plane; w = 1/far plane
uniform vec4 _ScreenParams;
   // x = width; y = height; z = 1 + 1/width; w = 1 + 1/height
uniform vec4 unity_Scale; // w = 1/scale; see _World2Object
uniform vec3 _WorldSpaceCameraPos;
uniform mat4 _Object2World; // model matrix
uniform mat4 _World2Object; // inverse model matrix
   // (all but the bottom-right element have to be scaled
   // with unity_Scale.w if scaling is important)
uniform vec4 _LightPositionRange; // xyz = pos, w = 1/range
uniform vec4 _WorldSpaceLightPos0;
   // position or direction of light source
uniform vec4 _LightColor0; // color of light source
uniform mat4 _LightMatrix0; // matrix to light space
```

As the comments suggest, instead of defining all these uniforms (except `_LightColor0` and `_LightMatrix0`), you could also include the file `UnityCG.glslinc`. However, for some unknown reason `_LightColor0` and `_LightMatrix0` are not included in this file; thus, we have to define them separately:

```
#include "UnityCG.glslinc"
uniform vec4 _LightColor0;
uniform mat4 _LightMatrix0;
```

Unity does not always update all of these uniforms. In particular, `_WorldSpaceLightPos0`, `_LightColor0`, and `_LightMatrix0` are only set correctly for shader passes that are tagged appropriately, e.g. with `Tags`

`{"LightMode" = "ForwardBase"}` as the first line in the `Pass {...}` block; see also GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[7].

5.0.30. More OpenGL-Specific Uniforms

Another class of built-in uniforms are defined for the OpenGL compability profile, for example the `mat4` matrix `gl_ModelViewProjectionMatrix`, which is equivalent to the matrix product `gl_ProjectionMatrix * gl_ModelViewMatrix` of two other built-in uniforms. The corresponding transformations are described in detail in GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[8].

As you can see in the shader above, these uniforms don't have to be defined; they are always available in GLSL shaders in Unity. If you had to define them, the definitions would look like this:

```
uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ProjectionMatrix;
uniform mat4 gl_ModelViewProjectionMatrix;
uniform mat4 gl_TextureMatrix[gl_MaxTextureCoords];
uniform mat3 gl_NormalMatrix;
   // transpose of the inverse of gl_ModelViewMatrix
uniform mat4 gl_ModelViewMatrixInverse;
uniform mat4 gl_ProjectionMatrixInverse;
uniform mat4 gl_ModelViewProjectionMatrixInverse;
uniform mat4 gl_TextureMatrixInverse[gl_MaxTextureCoords];
uniform mat4 gl_ModelViewMatrixTranspose;
uniform mat4 gl_ProjectionMatrixTranspose;
uniform mat4 gl_ModelViewProjectionMatrixTranspose;
uniform mat4 gl_TextureMatrixTranspose[gl_MaxTextureCoords];
uniform mat4 gl_ModelViewMatrixInverseTranspose;
uniform mat4 gl_ProjectionMatrixInverseTranspose;
uniform mat4 gl_ModelViewProjectionMatrixInverseTranspose;
uniform mat4 gl_TextureMatrixInverseTranspose[gl_MaxTextureCoords];

struct gl_LightModelParameters { vec4 ambient; };
uniform gl_LightModelParameters gl_LightModel;
...
```

In fact, the compability profile of OpenGL defines even more uniforms; see Chapter 7 of the "OpenGL Shading Language 4.10.6 Specification" available at KHRONOS' OPENGL PAGE[9]. Unity supports many of them but not all.

7 Chapter 10 on page 77
8 Chapter 43 on page 407
9 HTTP://WWW.KHRONOS.ORG/OPENGL/

Some of these uniforms are arrays, e.g `gl_TextureMatrix`. In fact, an array of matrices `gl_TextureMatrix[0]`, `gl_TextureMatrix[1]`, ..., `gl_TextureMatrix[gl_MaxTextureCoords - 1]` is available, where `gl_MaxTextureCoords` is a built-in integer.

5.0.31. Computing the View Matrix

Traditionally, it is customary to do many computations in view space, which is just a rotated and translated version of world space (see GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[10] for the details). Therefore, OpenGL offers only the product of the model matrix $M_{object \to world}$ and the view matrix $M_{world \to view}$, i.e. the model-view matrix $M_{object \to view}$, which is available in the uniform `gl_ModelViewMatrix`. The view matrix is not available. Unity also doesn't provide it.

However, `_Object2World` is just the model matrix and `_World2Object` is the inverse model matrix. (Except that all but the bottom-right element have to be scaled by `untiy_Scale.w`.) Thus, we can easily compute the view matrix. The mathematics looks like this:

$$M_{object \to view} = M_{world \to view} M_{object \to world} \Rightarrow M_{world \to view} = M_{object \to view} M^{-1}_{object \to world}$$

In other words, the view matrix is the product of the model-view matrix and the inverse model matrix (which is `_World2Object * unity_Scale.w` except for the bottom-right element, which is 1). Assuming that we have defined the uniforms `_World2Object` and `unity_Scale`, we can compute the view matrix this way in GLSL:

```
mat4 modelMatrixInverse = _World2Object * unity_Scale.w;
modelMatrixInverse[3][3] = 1.0;
mat4 viewMatrix = gl_ModelViewMatrix * modelMatrixInverse;
```

5.0.32. User-Specified Uniforms: Shader Properties

There is one more important type of uniform variables: uniforms that can be set by the user. Actually, these are called shader properties in Unity. You can think of them as parameters of the shader. A shader without parameters is usually used only by its programmer because even the smallest necessary change requires some

10 Chapter 43 on page 407

programming. On the other hand, a shader using parameters with descriptive names can be used by other people, even non-programmers, e.g. CG artists. Imagine you are in a game development team and a CG artist asks you to adapt your shader for each of 100 design iterations. It should be obvious that a few parameters, which even a CG artist can play with, might save **you** a lot of time. Also, imagine you want to sell your shader: parameters will often dramatically increase the value of your shader.

Since the DESCRIPTION OF SHADER PROPERTIES[11] in Unity's ShaderLab reference is quite OK, here is only an example, how to use shader properties in our example. We first declare the properties and then define uniforms of the same names and corresponding types.

```
Shader "GLSL shading in world space" {
   Properties {
      _Point ("a point in world space", Vector) = (0., 0., 0., 1.0)
      _DistanceNear ("threshold distance", Float) = 5.0
      _ColorNear ("color near to point", Color) = (0.0, 1.0, 0.0, 1.0)
      _ColorFar ("color far from point", Color) = (0.3, 0.3, 0.3, 1.0)
   }

   SubShader {
      Pass {
         GLSLPROGRAM

         // uniforms corresponding to properties
         uniform vec4 _Point;
         uniform float _DistanceNear;
         uniform vec4 _ColorNear;
         uniform vec4 _ColorFar;

         #include "UnityCG.glslinc"
            // defines _Object2World and _World2Object

         varying vec4 position_in_world_space;

         #ifdef VERTEX

         void main()
         {
            mat4 modelMatrix = _Object2World;

            position_in_world_space = modelMatrix * gl_Vertex;

            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif
```

11 HTTP://UNITY3D.COM/SUPPORT/DOCUMENTATION/COMPONENTS/SL-PROPERTIES.HTML

```
         #ifdef FRAGMENT

         void main()
         {
            float dist= distance(position_in_world_space, _Point);

            if (dist < _DistanceNear)
            {
               gl_FragColor = _ColorNear;
            }
            else
            {
               gl_FragColor = _ColorFar;
            }
         }

         #endif

         ENDGLSL
      }
   }
}
```

With these parameters, a non-programmer can modify the effect of our shader. This is nice; however, the properties of the shader (and in fact uniforms in general) can also be set by scripts! For example, a JavaScript attached to the game object that is using the shader can set the properties with these lines:

```
renderer.sharedMaterial.SetVector("_Point",
   Vector4(1.0, 0.0, 0.0, 1.0));
renderer.sharedMaterial.SetFloat("_DistanceNear",
   10.0);
renderer.sharedMaterial.SetColor("_ColorNear",
   Color(1.0, 0.0, 0.0));
renderer.sharedMaterial.SetColor("_ColorFar",
   Color(1.0, 1.0, 1.0));
```

Use `sharedMaterial` if you want to change the parameters for all objects that use this material and just `material` if you want to change the parameters only for one object. With scripting you could, for example, set the `_Point` to the position of another object (i.e. the `position` of its Transform component). In this way, you can specify a point just by moving another object around in the editor. In order to write such a script, select **Create > JavaScript** in the **Project View** and copy & paste this code:

```
@script ExecuteInEditMode() // make sure to run in edit mode

var other : GameObject; // another user-specified object

function Update () // this function is called for every frame
{
   if (null != other) // has the user specified an object?
```

```
   {
      renderer.sharedMaterial.SetVector("_Point",
         other.transform.position); // set the shader property
         // _Point to the position of the other object
   }
}
```

Then, you should attach the script to the object with the shader and drag & drop another object to the `other` variable of the script in the **Inspector View**.

5.0.33. Summary

Congratulations, you made it! (In case you wonder: yes, I'm also talking to myself here. ;) We discussed:

- How to transform a vertex into world coordinates.
- The most important Unity-specific uniforms that are supported by Unity.
- The most important OpenGL-specific uniforms that are supported by Unity.
- How to make a shader more useful and valuable by adding shader properties.

5.0.34. Further Reading

If you want to know more

- about vector and matrix operations (e.g. the `distance()` function), you should read GLSL PROGRAMMING/VECTOR AND MATRIX OPERATIONS[12].
- about the standard vertex transformations, e.g. the model matrix and the view matrix, you should read GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[13].
- about the application of transformation matrices to points and directions, you should read GLSL PROGRAMMING/APPLYING MATRIX TRANSFORMATIONS[14].
- about the specification of shader properties, you should read Unity's documentation about "SHADERLAB SYNTAX: PROPERTIES"[15].

12 Chapter 44 on page 421
13 Chapter 43 on page 407
14 Chapter 45 on page 429
15 HTTP://UNITY3D.COM/SUPPORT/DOCUMENTATION/COMPONENTS/SL-PROPERTIES.HTML

Part II.

Transparent Surfaces

6. Cutaways

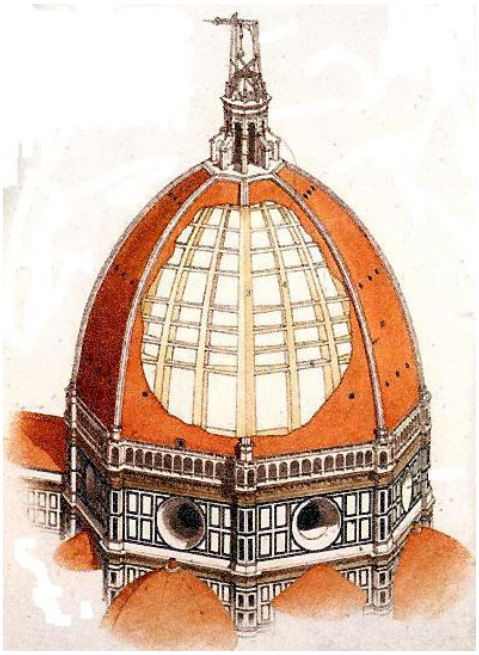

Figure 5 Cutaway drawing of the dome of the Florence cathedral by Filippo Brunelleschi, 1414-36.

This tutorial covers **discarding fragments**, determining whether the front face or back face is rendered, and **front-face and back-face culling**. This tutorial assumes that you are familiar with varying variables as discussed in GLSL PROGRAMMING/UNITY/RGB CUBE[1].

The main theme of this tutorial is to cut away triangles or fragments even though they are part of a mesh that is being rendered. The main two reasons are: we want to look through a triangle or fragment (as in the case of the roof in the drawing below

1 Chapter 3 on page 15

, which is only partly cut away) or we know that a triangle isn't visible anyways; thus, we can save some performance by not processing it. OpenGL supports these situations in several ways; we will discuss two of them.

6.0.35. Very Cheap Cutaways

The following shader is a very cheap way of cutting away parts of a mesh: all fragments are cut away that have a positive *y* coordinate in object coordinates (i.e. in the coordinate system in which it was modeled; see GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[2] for details about coordinate systems). Here is the code:

```
Shader "GLSL shader using discard" {
   SubShader {
      Pass {
         Cull Off // turn off triangle culling, alternatives are:
         // Cull Back (or nothing): cull only back faces
         // Cull Front : cull only front faces

         GLSLPROGRAM

         varying vec4 position_in_object_coordinates;

         #ifdef VERTEX

         void main()
         {
            position_in_object_coordinates= gl_Vertex;
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif

         #ifdef FRAGMENT

         void main()
         {
            if (position_in_object_coordinates.y > 0.0)
            {
               discard; // drop the fragment if y coordinate > 0
            }
            if (gl_FrontFacing) // are we looking at a front face?
            {
               gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0); // yes: green
            }
            else
            {
               gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0); // no: red
            }
         }
```

2 Chapter 43 on page 407

```
         #endif

         ENDGLSL
      }
   }
}
```

When you apply this shader to any of the default objects, the shader will cut away half of them. This is a very cheap way of producing hemispheres or open cylinders.

6.0.36. Discarding Fragments

Let's first focus on the `discard` instruction in the fragment shader. This instruction basically just discards the processed fragment. (This was called a fragment "kill" in earlier shading languages; I can understand that the fragments prefer the term "discard".) Depending on the hardware, this can be a quite expensive technique in the sense that rendering might perform considerably worse as soon as there is one shader that includes a `discard` instruction (regardless of how many fragments are actually discarded, just the presence of the instruction may result in the deactivation of some important optimizations). Therefore, you should avoid this instruction whenever possible but in particular when you run into performance problems.

One more note: the condition for the fragment `discard` includes only an object coordinate. The consequence is that you can rotate and move the object in any way and the cutaway part will always rotate and move with the object. You might want to check what cutting in world space looks like: change the vertex and fragment shader such that the world coordinate *y* is used in the condition for the fragment `discard`. Tip: see GLSL PROGRAMMING/UNITY/SHADING IN WORLD SPACE[3] for how to transform the vertex into world space.

6.0.37. Better Cutaways

If you are not(!) familiar with scripting in Unity, you might try the following idea to improve the shader: change it such that fragments are discarded if the *y* coordinate is greater than some threshold variable. Then introduce a shader property to allow the user to control this threshold. Tip: see GLSL PROGRAMMING/UNITY/SHADING IN WORLD SPACE[4] for a discussion of shader properties.

3 Chapter 5 on page 31
4 Chapter 5 on page 31

If you are familiar with scripting in Unity, you could try this idea: write a script for an object that takes a reference to another sphere object and assigns (using `renderer.sharedMaterial.SetMatrix()`) the inverse model matrix (`renderer.worldToLocalMatrix`) of that sphere object to a `mat4` uniform variable of the shader. In the shader, compute the position of the fragment in world coordinates and apply the inverse model matrix of the other sphere object to the fragment position. Now you have the position of the fragment in the local coordinate system of the other sphere object; here, it is easy to test whether the fragment is inside the sphere or not because in this coordinate system all spheres are centered around the origin with radius 0.5. Discard the fragment if it is inside the other sphere object. The resulting script and shader can cut away points from the surface of any object with the help of a cutting sphere that can be manipulated interactively in the editor like any other sphere.

6.0.38. Distinguishing between Front and Back Faces

A special boolean variable `gl_FrontFacing` is available in the fragment shader that specifies whether we are looking at the front face of a triangle. Usually, the front faces are facing the outside of a mesh and the back faces the inside. (Just as the surface normal vector usually points to the outside.) However, the actual way front and back faces are distinguished is the order of the vertices in a triangle: if the camera sees the vertices of a triangle in counter-clockwise order, it sees the front face. If it sees the vertices in clockwise order, it sees the back face.

Our fragment shader checks the variable `gl_FrontFacing` and assigns green to the output fragment color if `gl_FrontFacing` is `true` (i.e. the fragment is part of a front-facing triangle; i.e. it is facing the outside), and red if `gl_FrontFacing` is `false` (i.e. the fragment is part of a back-facing triangle; i.e. it is facing the inside). In fact, `gl_FrontFacing` allows you not only to render the two faces of a surfaces with different colors but with completely different styles.

Note that basing the definition of front and back faces on the order of vertices in a triangle can cause problems when vertices are mirrored, i.e. scaled with a negative factor. Unity tries to take care of these problems; thus, just specifying a negative scaling in the Transform component of the game object will usually not cause this problem. However, since Unity has no control over what we are doing in the vertex shader, we can still turn the inside out by multiplying one (or three) of the coordinates with -1, e.g. by assigning `gl_Position` this way in the vertex shader:

```
gl_Position = gl_ModelViewProjectionMatrix
* vec4(-gl_Vertex.x, gl_Vertex.y, gl_Vertex.z, 1.0);
```

This just multiplies the *x* coordinate by -1. For a sphere, you might think that nothing happens, but it actually turns front faces into back faces and vice versa; thus, now the inside is green and the outside is red. (By the way, this problem also affects the surface normal vector.) Thus, be careful with mirrors!

6.0.39. Culling of Front or Back Faces

Finally, the shader (more specifically the shader pass) includes the line `Cull Off`. This line has to come before `GLSLPROGRAM` because it is not in GLSL. In fact, it is the COMMAND OF UNITY'S SHADERLAB[5] to turn off any triangle culling. This is necessary because by default back faces are culled away as if the line `Cull Back` was specified. You can also specify the culling of front faces with `Cull Front`. The reason why culling of back-facing triangles is active by default, is that the inside of objects is usually invisible; thus, back-face culling can save quite some performance by avoiding to rasterize these triangles as explained next. Of course, we were able to see the inside with our shader because we have discarded some fragments; thus, we had to deactivate back-face culling.

How does culling work? Triangles and vertices are processed as usual. However, after the viewport transformation of the vertices to screen coordinates (see GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[6]) the graphics processor determines whether the vertices of a triangle appear in counter-clockwise order or in clockwise order on the screen. Based on this test, each triangle is considered a front-facing or a back-facing triangle. If it is front-facing and culling is activated for front-facing triangles, it will be discarded, i.e., the processing of it stops and it is not rasterized. Analogously, if it is back-facing and culling is activated for back-facing triangles. Otherwise, the triangle will be processed as usual.

6.0.40. Summary

Congratulations, you have worked through another tutorial. (If you have tried one of the assignments: good job! I didn't yet.) We have looked at:

5 HTTP://UNITY3D.COM/SUPPORT/DOCUMENTATION/COMPONENTS/SL-CULLANDDEPTH.HTML

6 Chapter 43 on page 407

- How to discard fragments.
- How to render front-facing and back-facing triangles in different colors.
- How to deactivate the default culling of back faces.
- How to activate the culling of front faces.

6.0.41. Further Reading

If you still want to know more

- about the vertex transformations such as the model transformation from object to world coordinates or the viewport transformation to screen coordinates, you should read GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[7].
- about how to define shader properties, you should read GLSL PROGRAMMING/UNITY/SHADING IN WORLD SPACE[8].
- about Unity's ShaderLab syntax for specifying culling, you should read CULLING & DEPTH TESTING[9].

7 Chapter 43 on page 407
8 Chapter 5 on page 31
9 HTTP://UNITY3D.COM/SUPPORT/DOCUMENTATION/COMPONENTS/SL-CULLANDDEPTH.HTML

7. Transparency

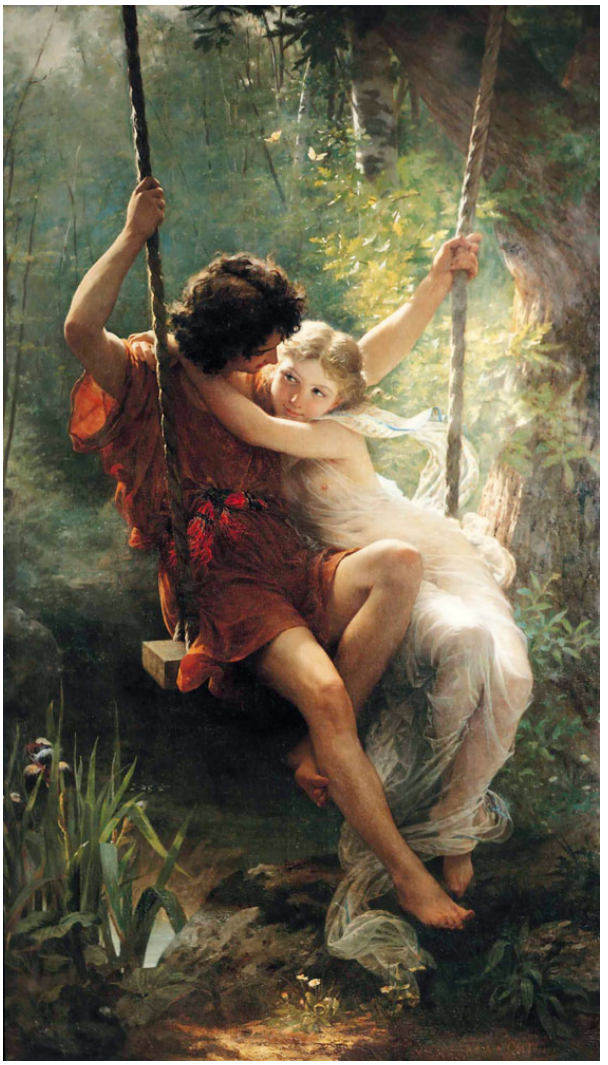

Figure 6 «Le Printemps» by Pierre Auguste Cot, 1873. Note the transparent clothing.

This tutorial covers **blending** of fragments (i.e. compositing them) using GLSL shaders in Unity. It assumes that you are familiar with the concept of front and back faces as discussed in GLSL PROGRAMMING/UNITY/CUTAWAYS[1].

More specifically, this tutorial is about **rendering transparent objects**, e.g. transparent glass, plastic, fabrics, etc. (More strictly speaking, these are actually semi-transparent objects because they don't need to be perfectly transparent.)

7.0.42. Blending

As mentioned in GLSL PROGRAMMING/OPENGL ES 2.0 PIPELINE[2], the fragment shader computes an RGBA color (i.e. red, green, blue, and alpha components in `gl_FragColor`) for each fragment (unless the fragment is discarded). The fragments are then processed as discussed in GLSL PROGRAMMING/PER-FRAGMENT OPERATIONS[3]. One of the operations is the blending stage, which combines the color of the fragment (as specified in `gl_FragColor`), which is called the "source color", with the color of the corresponding pixel that is already in the framebuffer, which is called the "destination color" (because the "destination" of the resulting blended color is the framebuffer).

Blending is a fixed-function stage, i.e. you can customize it but not program it. The way it is customized, is by specifying a **blend equation**. You can think of the blend equation as this definition of the resulting RGBA color:

```
vec4 result = SrcFactor * gl_FragColor + DstFactor *
pixel_color;
```

where `pixel_color` is the RGBA color that is currently in the framebuffer and `result` is the blended result, i.e. the output of the blending stage. `SrcFactor` and `DstFactor` are customizable RGBA colors (of type `vec4`) that are multiplied component-wise with the fragment color and the pixel color. The values of `SrcFactor` and `DstFactor` are specified in Unity's ShaderLab syntax with this line:

`Blend` {code for `SrcFactor`} {code for `DstFactor`}

1 Chapter 6 on page 43
2 Chapter 42 on page 399
3 Chapter 47 on page 443

The most common codes for the two factors are summarized in the following table (more codes are mentioned in UNITY'S SHADERLAB REFERENCE ABOUT BLENDING[4]):

Code	Resulting Factor (`SrcFactor` or `DstFactor`)
`One`	`vec4(1.0)`
`Zero`	`vec4(0.0)`
`SrcColor`	`gl_FragColor`
`SrcAlpha`	`vec4(gl_FragColor.a)`
`DstColor`	`pixel_color`
`DstAlpha`	`vec4(pixel_color.a)`
`OneMinusSrcColor`	`vec4(1.0) - gl_FragColor`
`OneMinusSrcAlpha`	`vec4(1.0 - gl_FragColor.a)`
`OneMinusDstColor`	`vec4(1.0) - pixel_color`
`OneMinusDstAlpha`	`vec4(1.0 - pixel_color.a)`

As discussed in GLSL PROGRAMMING/VECTOR AND MATRIX OPERATIONS[5], `vec4(1.0)` is just a short way of writing `vec4(1.0, 1.0, 1.0, 1.0)`. Also note that all components of all colors and factors in the blend equation are clamped between 0 and 1.

7.0.43. Specific Blending Equations

An example of specifying a blend equation in Unity is:

```
Blend One Zero
```

This sets `SrcFactor` to `vec4(1.0)` and `DstFactor` to `vec4(0.0)`. Thus the blend equation becomes:

```
vec4 result = vec4(1.0) * gl_FragColor + vec4(0.0) *
pixel_color;
```

which is actually the same as:

```
vec4 result = gl_FragColor;
```

4 HTTP://UNITY3D.COM/SUPPORT/DOCUMENTATION/COMPONENTS/SL-BLEND.HTML

5 Chapter 44 on page 421

I.e this blend equation just writes the fragment color to the framebuffer, which is in fact the default behavior if no blending is specified.

Another example:

```
Blend One One
```

corresponds to:

```
vec4 result = vec4(1.0) * gl_FragColor + vec4(1.0) *
pixel_color;
```

which just adds the fragment color to the color in the framebuffer. This is often used for particle systems when they represent fire or something else that is transparent and emits light.

Another specific example is called “alpha blending”:

```
Blend SrcAlpha OneMinusSrcAlpha
```

which corresponds to:

```
vec4 result = vec4(gl_FragColor.a) * gl_FragColor +
vec4(1.0 - gl_FragColor.a) * pixel_color;
```

This uses the alpha component of `gl_FragColor` as an **opacity**. I.e. the more opaque the fragment color is, the larger its opacity and therefore its alpha component, and thus the more of the fragment color is mixed in the the result and the less of the pixel color in the framebuffer. A perfectly opaque fragment color (i.e. with an alpha component of 1) will completely replace the pixel color.

This blend equation is sometimes referred to as an “over” operation, i.e. “`gl_FragColor` over `pixel_color`”, since it corresponds to putting a layer of the fragment color with a specific opacity on top of the pixel color.

There is an important variant of alpha blending: sometimes the fragment color has its alpha component already premultiplied to the color components. (You might think of it as a price that has VAT already included.) In this case, alpha should not be multiplied again (VAT should not be added again) and the correct blending is:

```
Blend One OneMinusSrcAlpha
```

which corresponds to:

```
vec4 result = vec4(1.0) * gl_FragColor + vec4(1.0 -
gl_FragColor.a) * pixel_color;
```

Due to the popularity of alpha blending, the alpha component of a color is often called opacity even if alpha blending is not employed. Moreover, note that in computer graphics a common formal definition of **transparency** is **1 – opacity**.

More examples of blend equations are given in UNITY'S SHADERLAB REFERENCE ABOUT BLENDING[6].

7.0.44. Shader Code

Here is a simple shader which uses a green color with opacity 0.3:

```
Shader "GLSL shader using blending" {
   SubShader {
      Tags { "Queue" = "Transparent" }
         // draw after all opaque geometry has been drawn
      Pass {
         ZWrite Off // don't write to depth buffer
            // in order not to occlude other objects

         Blend SrcAlpha OneMinusSrcAlpha // use alpha blending

         GLSLPROGRAM

         #ifdef VERTEX

         void main()
         {
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif

         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor = vec4(0.0, 1.0, 0.0, 0.3);
               // the fourth component (alpha) is important:
               // this is semitransparent green
         }

         #endif

         ENDGLSL
      }
   }
}
```

6 HTTP://UNITY3D.COM/SUPPORT/DOCUMENTATION/COMPONENTS/SL-BLEND.HTML

Apart from the blend equation, which has been discussed above, there are only two lines that need more explanation: `Tags { "Queue" = "Transparent" }` and `ZWrite Off`.

`ZWrite Off` deactivates writing to the depth buffer. As explained in GLSL PROGRAMMING/PER-FRAGMENT OPERATIONS[7], the depth buffer keeps the depth of the nearest fragment and discards any fragments that have a larger depth. In the case of a transparent fragment, however, this is not what we want since we can (at least potentially) see through a transparent fragment. Thus, transparent fragments should not occlude other fragments and therefore the writing to the depth buffer is deactivated. See also UNITY'S SHADERLAB REFERENCE ABOUT CULLING AND DEPTH TESTING[8].

The line `Tags { "Queue" = "Transparent" }` specifies that the meshes using this subshader are rendered after all the opaque meshes were rendered. The reason is partly because we deactivate writing to the depth buffer: one consequence is that transparent fragments can be occluded by opaque fragments even though the opaque fragments are farther away. In order to fix this problem, we first draw all opaque meshes (in Unity´s "opaque queue") before drawing all transparent meshes (in Unity's "transparent queue"). Whether or not a mesh is considered opaque or transparent depends on the tags of its subshader as specified with the line `Tags { "Queue" = "Transparent" }`. More details about subshader tags are described in UNITY'S SHADERLAB REFERENCE ABOUT SUBSHADER TAGS[9].

It should be mentioned that this strategy of rendering transparent meshes with deactivated writing to the depth buffer does not always solve all problems. It works perfectly if the order in which fragments are blended does not matter; for example, if the fragment color is just added to the pixel color in the framebuffer, the order in which fragments are blended is not important; see GLSL PROGRAMMING/UNITY/ORDER-INDEPENDENT TRANSPARENCY[10]. However, for other blending equations, e.g. alpha blending, the result will be different depending on the order in which fragments are blended. (If you look through almost opaque green glass at almost opaque red glass you will mainly see green, while you will mainly see red if you look through almost opaque red glass at almost opaque green glass. Similarly, blending almost opaque green color over almost opaque red color will be different from blending almost opaque red color over almost opaque

7 Chapter 47 on page 443

8 http://unity3d.com/support/documentation/Components/SL-CullAndDepth.html

9 http://unity3d.com/support/documentation/Components/SL-SubshaderTags.html

10 Chapter 8 on page 59

green color.) In order to avoid artifacts, it is therefore advisable to use additive blending or (premultiplied) alpha blending with small opacities (in which case the destination factor `DstFactor` is close to 1 and therefore alpha blending is close to additive blending).

7.0.45. Including Back Faces

The previous shader works well with other objects but it actually doesn't render the "inside" of the object. However, since we can see through the outside of a transparent object, we should also render the inside. As discussed in GLSL PROGRAMMING/UNITY/CUTAWAYS[11], the inside can be rendered by deactivating culling with `Cull Off`. However, if we just deactivate culling, we might get in trouble: as discussed above, it often matters in which order transparent fragments are rendered but without any culling, overlapping triangles from the inside and the outside might be rendered in a random order which can lead to annoying rendering artifacts. Thus, we would like to make sure that the inside (which is usually farther away) is rendered first before the outside is rendered. In Unity's ShaderLab this is achieved by specifying two passes, which are executed for the same mesh in the order in which they are defined:

```
Shader "GLSL shader using blending (including back faces)" {
   SubShader {
      Tags { "Queue" = "Transparent" }
         // draw after all opaque geometry has been drawn
      Pass {
         Cull Front // first pass renders only back faces
             // (the "inside")
         ZWrite Off // don't write to depth buffer
             // in order not to occlude other objects
         Blend SrcAlpha OneMinusSrcAlpha // use alpha blending

         GLSLPROGRAM

         #ifdef VERTEX

         void main()
         {
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif


         #ifdef FRAGMENT
```

11 Chapter 6 on page 43

```
         void main()
         {
            gl_FragColor = vec4(1.0, 0.0, 0.0, 0.3);
               // the fourth component (alpha) is important:
               // this is semitransparent green
         }

         #endif

         ENDGLSL
      }

      Pass {
         Cull Back // second pass renders only front faces
            // (the "outside")
         ZWrite Off // don't write to depth buffer
            // in order not to occlude other objects
         Blend SrcAlpha OneMinusSrcAlpha
            // standard blend equation "source over destination"

         GLSLPROGRAM

         #ifdef VERTEX

         void main()
         {
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif


         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor = vec4(0.0, 1.0, 0.0, 0.3);
               // fourth component (alpha) is important:
               // this is semitransparent green
         }

         #endif

         ENDGLSL
      }
   }
}
```

In this shader, the first pass uses front-face culling (with `Cull Front`) to render the back faces (the inside) first. After that the second pass uses back-face culling (with `Cull Back`) to render the front faces (the outside). This works perfect for convex meshes (closed meshes without dents; e.g. spheres or cubes) and is often a good approximation for other meshes.

7.0.46. Summary

Congratulations, you made it through this tutorial! One interesting thing about rendering transparent objects is that it isn't just about blending but also requires knowledge about culling and the depth buffer. Specifically, we have looked at:

- What blending is and how it is specified in Unity.
- How a scene with transparent and opaque objects is rendered and how objects are classified as transparent or opaque in Unity.
- How to render the inside and outside of a transparent object, in particular how to specify two passes in Unity.

7.0.47. Further Reading

If you still want to know more

- the OpenGL pipeline, you should read GLSL Programming/OpenGL ES 2.0 Pipeline[12].
- about per-fragment operations in the OpenGL pipeline (e.g. blending and the depth test), you should read GLSL Programming/Per-Fragment Operations[13].
- about front-face and back-face culling, you should read GLSL Programming/Unity/Cutaways[14].
- about how to specify culling and the depth buffer functionality in Unity, you should read Unity's ShaderLab reference about culling and depth testing[15].
- about how to specify blending in Unity, you should read Unity's ShaderLab reference about blending[16].
- about the render queues in Unity, you should read Unity's ShaderLab reference about subshader tags[17].

12 Chapter 42 on page 399
13 Chapter 47 on page 443
14 Chapter 6 on page 43
15 http://unity3d.com/support/documentation/Components/SL-CullAndDepth.html
16 http://unity3d.com/support/documentation/Components/SL-Blend.html
17 http://unity3d.com/support/documentation/Components/SL-SubshaderTags.html

8. Order-Independent Transparency

Figure 7 “Where Have You Bean” by flickr user Ombligotron. The typo in the title refers to the depicted sculpture “Cloud Gate” a.k.a. “The Bean”.

Figure 8 “84 – Father son” by Ben Newton. An example of double exposure.

This tutorial covers **order-independent blending**.

It continues the discussion in GLSL PROGRAMMING/UNITY/TRANSPARENCY[1] and solves some problems of standard transparency. If you haven't read that tutorial, you should read it first.

8.0.48. Order-Independent Blending

As noted in GLSL PROGRAMMING/UNITY/TRANSPARENCY[2], the result of blending often (in particular for standard alpha blending) depends on the order in which triangles are rendered and therefore results in rendering artifacts if the triangles are not sorted from back to front (which they usually aren't). The term “order-independent transparency” describes various techniques to avoid this problem. One of these techniques is order-independent blending, i.e. the use of a blend equation that does not depend on the order in which triangles are rasterized. There two basic possibilities: additive blending and multiplicative blending.

1 Chapter 7 on page 49
2 Chapter 7 on page 49

Additive Blending

The standard example for additive blending are double exposures as in the images in this section: colors are added such that it is impossible (or at least very hard) to say in which order the photos were taken. Additive blending can be characterized in terms of the blend equation introduced in GLSL PROGRAMMING/UNITY/TRANSPARENCY[3]:

```
vec4 result = SrcFactor * gl_FragColor + DstFactor *
pixel_color;
```

where `SrcFactor` and `DstFactor` are determined by a line in Unity's ShaderLab syntax:

`Blend` {code for `SrcFactor`} {code for `DstFactor`}

For additive blending, the code for `DstFactor` has to be `One` and the code for `SrcFactor` must not depend on the pixel color in the framebuffer; i.e., it can be `One`, `SrcColor`, `SrcAlpha`, `OneMinusSrcColor`, or `OneMinusSrcAlpha`.

An example is:

```
Shader "GLSL shader using additive blending" {
   SubShader {
      Tags { "Queue" = "Transparent" }
         // draw after all opaque geometry has been drawn
      Pass {
         Cull Off // draw front and back faces
         ZWrite Off // don't write to depth buffer
            // in order not to occlude other objects
         Blend SrcAlpha One // additive blending

         GLSLPROGRAM

         #ifdef VERTEX

         void main()
         {
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif


         #ifdef FRAGMENT

         void main()
```

3 Chapter 7 on page 49

```
         {
            gl_FragColor = vec4(1.0, 0.0, 0.0, 0.3);
         }

         #endif

         ENDGLSL
      }
   }
}
```

Multiplicative Blending

An example for multiplicative blending in photography is the use of multiple uniform grey filters: the order in which the filters are put onto a camera doesn't matter for the resulting attenuation of the image. In terms of the rasterization of triangles, the image corresponds to the contents of the framebuffer before the triangles are rasterized, while the filters correspond to the triangles.

When specifying mutiplicative blending in Unity with the line

`Blend` {code for `SrcFactor`} {code for `DstFactor`}

the code for `SrcFactor` has to be `Zero` and the code for `DstFactor` must depend on the fragment color; i.e., it can be `SrcColor`, `SrcAlpha`, `OneMinusSrcColor`, or `OneMinusSrcAlpha`. A typical example for attenuating the background with the opacity specified by the alpha component of fragments would use `OneMinusSrcAlpha` for the code for `DstFactor`:

```
Shader "GLSL shader using multiplicative blending" {
   SubShader {
      Tags { "Queue" = "Transparent" }
         // draw after all opaque geometry has been drawn
      Pass {
         Cull Off // draw front and back faces
         ZWrite Off // don't write to depth buffer
            // in order not to occlude other objects
         Blend Zero OneMinusSrcAlpha // multiplicative blending
            // for attenuation by the fragment's alpha

         GLSLPROGRAM

         #ifdef VERTEX

         void main()
         {
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif
```

```
         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor = vec4(1.0, 0.0, 0.0, 0.3);
               // only A (alpha) is used
         }

         #endif

         ENDGLSL
      }
   }
}
```

8.0.49. Complete Shader Code

Finally, it makes good sense to combine multiplicative blending for the attenuation of the background and additive blending for the addition of colors of the triangles in one shader by combining the two passes that were presented above. This can be considered an approximation to alpha blending for **small opacities**, i.e. **small values of alpha**, if one ignores attenuation of colors of the triangle mesh by itself.

```
Shader "GLSL shader using order-independent blending" {
   SubShader {
      Tags { "Queue" = "Transparent" }
         // draw after all opaque geometry has been drawn
      Pass {
         Cull Off // draw front and back faces
         ZWrite Off // don't write to depth buffer
            // in order not to occlude other objects
         Blend Zero OneMinusSrcAlpha // multiplicative blending
            // for attenuation by the fragment's alpha

         GLSLPROGRAM

         #ifdef VERTEX

         void main()
         {
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif


         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor = vec4(1.0, 0.0, 0.0, 0.3);
```

```
            // only A (alpha) is used
         }

         #endif

         ENDGLSL
      }

      Pass {
         Cull Off // draw front and back faces
         ZWrite Off // don't write to depth buffer
            // in order not to occlude other objects
         Blend SrcAlpha One // additive blending to add colors

         GLSLPROGRAM

         #ifdef VERTEX

         void main()
         {
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif


         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor = vec4(1.0, 0.0, 0.0, 0.3);
         }

         #endif

         ENDGLSL
      }
   }
}
```

Note that the order of the two passes is important: first the background is attenuated and then colors are added.

8.0.50. Summary

Congratulations, you have reached the end of this tutorial. We have looked at:

- What order-independent transparency and order-independent blending is.
- What the two most important kinds of order-independent blending are (additive and multiplicative).
- How to implement additive and multiplicative blending.

- How to combine two passes for additive and multiplicative blending for an order-independent approximation to alpha blending.

8.0.51. Further Reading

If you still want to know more

- about the shader code, you should read GLSL PROGRAMMING/UNITY/TRANSPARENCY[4].
- about another technique for order-independent transparency, namely depth peeling, you could read a technical report by Cass Everitt: "Interactive Order-Independent Transparency", which is available ONLINE[5].

4 Chapter 7 on page 49

5 HTTP://DEVELOPER.NVIDIA.COM/CONTENT/ORDER-INDEPENDENT-TRANSPARENCY

9. Silhouette Enhancement

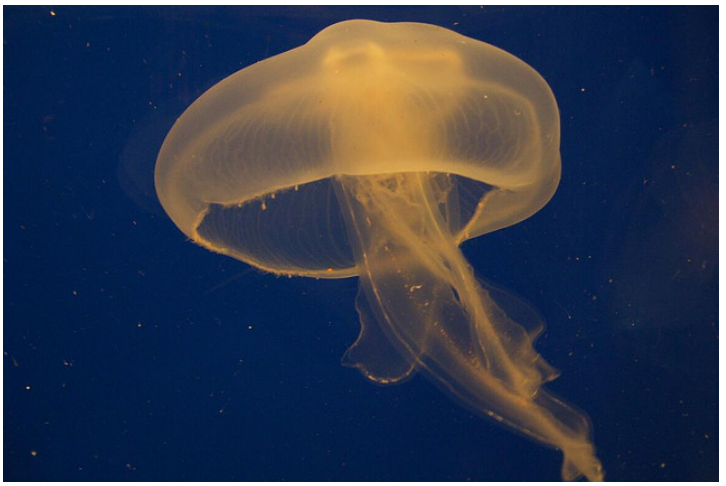

Figure 9 A semitransparent jellyfish. Note the increased opaqueness at the silhouettes.

This tutorial covers the **transformation of surface normal vectors**. It assumes that you are familiar with alpha blending as discussed in GLSL PROGRAMMING/UNITY/TRANSPARENCY[1] and with shader properties as discussed in GLSL PROGRAMMING/UNITY/SHADING IN WORLD SPACE[2].

The objective of this tutorial is to achieve an effect that is visible in the photo to the left: the silhouettes of semitransparent objects tend to be more opaque than the rest of the object. This adds to the impression of a three-dimensional shape even without lighting. It turns out that transformed normals are crucial to obtain this effect.

1 Chapter 7 on page 49
2 Chapter 5 on page 31

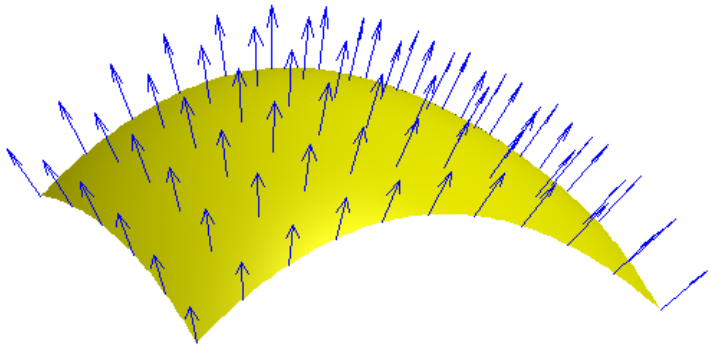

Figure 10 Surface normal vectors (for short: normals) on a surface patch.

9.0.52. Silhouettes of Smooth Surfaces

In the case of smooth surfaces, points on the surface at silhouettes are characterized by normal vectors that are parallel to the viewing plane and therefore orthogonal to the direction to the viewer. In the figure to the left, the blue normal vectors at the silhouette at the top of the figure are parallel to the viewing plane while the other normal vectors point more in the direction to the viewer (or camera). By calculating the direction to the viewer and the normal vector and testing whether they are (almost) orthogonal to each other, we can therefore test whether a point is (almost) on the silhouette.

More specifically, if $\mathbf{V}$ is the normalized (i.e. of length 1) direction to the viewer and $\mathbf{N}$ is the normalized surface normal vector, then the two vectors are orthogonal if the dot product is 0: $\mathbf{V} \cdot \mathbf{N} = 0$. In practice, this will be rarely the case. However, if the dot product $\mathbf{V} \cdot \mathbf{N}$ is close to 0, we can assume that the point is close to a silhouette.

9.0.53. Increasing the Opacity at Silhouettes

For our effect, we should therefore increase the opacity α if the dot product $\mathbf{V} \cdot \mathbf{N}$ is close to 0. There are various ways to increase the opacity for small dot products between the direction to the viewer and the normal vector. Here is one of them (which actually has a physical model behind it, which is described in Section 5.1 of

THIS PUBLICATION[3]) to compute the increased opacity α' from the regular opacity α of the material:

$$\alpha' = \min\left(1, \frac{\alpha}{|\mathbf{V}\cdot\mathbf{N}|}\right)$$

It always makes sense to check the extreme cases of an equation like this. Consider the case of a point close to the silhouette: $\mathbf{V}\cdot\mathbf{N} \approx 0$. In this case, the regular opacity α will be divided by a small, positive number. (Note that GLSL guarantees to handle the case of division by zero gracefully; thus, we don't have to worry about it.) Therefore, whatever α is, the ratio of α and a small positive number, will be larger. The min function will take care that the resulting opacity α' is never larger than 1.

On the other hand, for points far away from the silhouette we have $\mathbf{V}\cdot\mathbf{N} \approx 1$. In this case, $\alpha' \approx \min(1, \alpha) \approx \alpha$; i.e., the opacity of those points will not change much. This is exactly what we want. Thus, we have just checked that the equation is at least plausible.

9.0.54. Implementing an Equation in a Shader

In order to implement an equation like the one for α in a shader, the first question should be: Should it be implemented in the vertex shader or in the fragment shader? In some cases, the answer is clear because the implementation requires texture mapping, which is only available in the fragment shader (at least in Unity). In many cases, however, there is no general answer. Implementations in vertex shaders tend to be faster (because there are usually fewer vertices than fragments) but of lower image quality (because normal vectors and other vertex attributes can change abruptly between vertices). Thus, if you are most concerned about performance, an implementation in a vertex shader is probably a better choice. On the other hand, if you are most concerned about image quality, an implementation in a pixel shader might be a better choice. The same trade-off exists between per-vertex lighting (i.e. Gouraud shading, which is discussed in GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[4]) and per-fragment lighting (i.e. Phong shading, which is discussed in GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[5]).

3 HTTP://CITESEERX.IST.PSU.EDU/VIEWDOC/SUMMARY?DOI=10.1.1.125.1928

4 Chapter 11 on page 91

5 Chapter 13 on page 111

The next question is: in which coordinate system should the equation be implemented? (See GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[6] for a description of the standard coordinate systems.) Again, there is no general answer. However, an implementation in world coordinates is often a good choice in Unity because many uniform variables are specified in world coordinates. (In other environments implementations in view coordinates are very common.)

The final question before implementing an equation is: where do we get the parameters of the equation from? The regular opacity α is specified (within a RGBA color) by a shader property (see GLSL PROGRAMMING/UNITY/SHADING IN WORLD SPACE[7]). The normal vector `gl_Normal` is a standard vertex attribute (see GLSL PROGRAMMING/UNITY/DEBUGGING OF SHADERS[8]). The direction to the viewer can be computed in the vertex shader as the vector from the vertex position in world space to the camera position in world space `_WorldSpaceCameraPos`, which is provided by Unity.

Thus, we only have to transform the vertex position and the normal vector into world space before implementing the equation. The transformation matrix `_Object2World` from object space to world space and its inverse `_World2Object` are provided by Unity as discussed in GLSL PROGRAMMING/UNITY/SHADING IN WORLD SPACE[9]. The application of transformation matrices to points and normal vectors is discussed in detail in GLSL PROGRAMMING/APPLYING MATRIX TRANSFORMATIONS[10]. The basic result is that points and directions are transformed just by multiplying them with the transformation matrix, e.g.:

```
uniform mat4 _Object2World;
...
vec4 positionInWorldSpace = _Object2World * gl_Vertex;
vec3 viewDirection = _WorldSpaceCameraPos - vec3(positionInWorldSpace);
```

On the other hand **normal vectors are transformed by multiplying them with the transposed inverse transformation matrix**. Since Unity provides us with the inverse transformation matrix (which is `_World2Object * unity_Scale.w`), a better alternative is to multiply the normal vector **from the left** to the inverse matrix, which is equivalent to multiplying it from the right to the transposed inverse matrix as discussed in GLSL PROGRAMMING/APPLYING MATRIX TRANSFORMATIONS[11]:

6 Chapter 43 on page 407
7 Chapter 5 on page 31
8 Chapter 4 on page 23
9 Chapter 5 on page 31
10 Chapter 45 on page 429
11 Chapter 45 on page 429

```
uniform mat4 _World2Object; // the inverse of _Object2World
   // (after multiplication with unity_Scale.w)
uniform vec4 unity_Scale;
...
vec3 normalInWorldSpace = vec3(vec4(gl_Normal, 0.0) * _World2Object
   * unity_Scale.w); // corresponds to a multiplication of the
   // transposed inverse of _Object2World with gl_Normal
```

However, the multiplication with `unity_Scale.w` is unnecessary if the scaling doesn't matter; for example, if we normalize all transformed vectors.

Now we have all the pieces that we need to write the shader.

9.0.55. Shader Code

```
Shader "GLSL silhouette enhancement" {
   Properties {
      _Color ("Color", Color) = (1, 1, 1, 0.5)
         // user-specified RGBA color including opacity
   }
   SubShader {
      Tags { "Queue" = "Transparent" }
         // draw after all opaque geometry has been drawn
      Pass {
         ZWrite Off // don't occlude other objects
         Blend SrcAlpha OneMinusSrcAlpha // standard alpha blending

         GLSLPROGRAM

         uniform vec4 _Color; // define shader property for shaders

         // The following built-in uniforms are also defined in
         // "UnityCG.glslinc", which could be #included
         uniform vec3 _WorldSpaceCameraPos;
            // camera position in world space
         uniform mat4 _Object2World; // model matrix
         uniform mat4 _World2Object; // inverse model matrix
            // (apart from the factor unity_Scale.w)

         varying vec3 varyingNormalDirection;
            // normalized surface normal vector
         varying vec3 varyingViewDirection;
            // normalized view direction

         #ifdef VERTEX

         void main()
         {
            mat4 modelMatrix = _Object2World;
            mat4 modelMatrixInverse = _World2Object;
               // multiplication with unity_Scale.w is unnecessary
               // because we normalize transformed vectors
```

```
            varyingNormalDirection = normalize(
               vec3(vec4(gl_Normal, 0.0) * modelMatrixInverse));
            varyingViewDirection = normalize(_WorldSpaceCameraPos
               - vec3(modelMatrix * gl_Vertex));

            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif

         #ifdef FRAGMENT

         void main()
         {
            vec3 normalDirection = normalize(varyingNormalDirection);
            vec3 viewDirection = normalize(varyingViewDirection);

            float newOpacity = min(1.0, _Color.a
               / abs(dot(viewDirection, normalDirection)));
            gl_FragColor = vec4(vec3(_Color), newOpacity);
         }

         #endif

         ENDGLSL
      }
   }
}
```

The assignment to `newOpacity` is an almost literal translation of the equation

$$\alpha' = \min(1, \alpha / |\mathbf{V} \cdot \mathbf{N}|)$$

Note that we normalize the varyings `varyingNormalDirection` and `varyingViewDirection` in the vertex shader (because we want to interpolate between directions without putting more nor less weight on any of them) and at the begin of the fragment shader (because the interpolation can distort our normalization to a certain degree). However, in many cases the normalization of `varyingNormalDirection` in the vertex shader is not necessary. Similarly, the normalization of `varyingViewDirection` in the fragment shader is in most cases unnecessary.

9.0.56. More Artistic Control

While the described silhouette enhancement is based on a physical model, it lacks artistic control; i.e., a CG artist cannot easily create a thinner or thicker silhouette than the physical model suggests. To allow for more artistic control, you could introduce another (positive) floating-point number property and take the dot product $|\mathbf{V} \cdot \mathbf{N}|$ to the power of this number (using the built-in GLSL function `pow(float`

`x, float y)`) before using it in the equation above. This will allow CG artists to create thinner or thicker silhouettes independently of the opacity of the base color.

9.0.57. Summary

Congratulations, you have finished this tutorial. We have discussed:

- How to find silhouettes of smooth surfaces (using the dot product of the normal vector and the view direction).
- How to enhance the opacity at those silhouettes.
- How to implement equations in shaders.
- How to transform points and normal vectors from object space to world space (using the transposed inverse model matrix for normal vectors).
- How to compute the viewing direction (as the difference from the camera position to the vertex position).
- How to interpolate normalized directions (i.e. normalize twice: in the vertex shader and the fragment shader).
- How to provide more artistic control over the thickness of silhouettes .

9.0.58. Further Reading

If you still want to know more

- about object space and world space, you should read the description in GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[12].
- about how to apply transformation matrices to points, directions and normal vectors, you should read GLSL PROGRAMMING/APPLYING MATRIX TRANSFORMATIONS[13].
- about the basics of rendering transparent objects, you should read GLSL PROGRAMMING/UNITY/TRANSPARENCY[14].
- about uniform variables provided by Unity and shader properties, you should read GLSL PROGRAMMING/UNITY/SHADING IN WORLD SPACE[15].

12 Chapter 43 on page 407
13 Chapter 45 on page 429
14 Chapter 7 on page 49
15 Chapter 5 on page 31

- about the mathematics of silhouette enhancement, you could read Section 5.1 of the paper “Scale-Invariant Volume Rendering” by Martin Kraus, published at IEEE Visualization 2005, which is available ONLINE[16].

16 HTTP://CITESEERX.IST.PSU.EDU/VIEWDOC/SUMMARY?DOI=10.1.1.125.1928

Part III.

Basic Lighting

10. Diffuse Reflection

Figure 11 The light reflection from the surface of the moon is (in a good approximation) only diffuse.

This tutorial covers **per-vertex diffuse reflection**.

It's the first in a series of tutorials about basic lighting in Unity. In this tutorial, we start with diffuse reflection from a single directional light source and then include point light sources and multiple light sources (using multiple passes). Further tutorials cover extensions of this, in particular specular reflection, per-pixel lighting, and two-sided lighting.

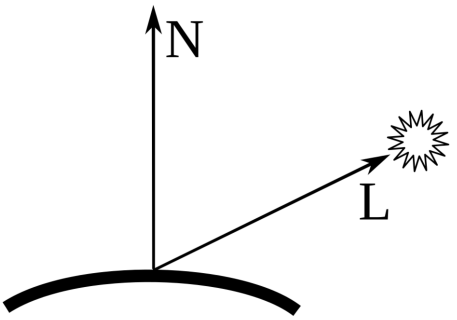

Figure 12 Diffuse reflection can be computed using the surface normal vector N and the light vector L, i.e. the vector to the light source.

10.0.59. Diffuse Reflection

The moon exhibits almost exclusively diffuse reflection (also called Lambertian reflection), i.e. light is reflected into all directions without specular highlights. Other examples of such materials are chalk and matte paper; in fact, any surface that appears dull and matte.

In the case of perfect diffuse reflection, the intensity of the observed reflected light depends on the cosine of the angle between the surface normal vector and the ray of the incoming light. As illustrated in the figure below , it is common to consider normalized vectors starting in the point of a surface, where the lighting should be computed: the normalized surface normal vector **N** is orthogonal to the surface and the normalized light direction **L** points to the light source.

For the observed diffuse reflected light I_{diffuse}, we need the cosine of the angle between the normalized surface normal vector **N** and the normalized direction to the light source **L**, which is the dot product **N**·**L** because the dot product **a**·**b** of any two vectors **a** and **b** is:

$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}|\,|\mathbf{b}| \cos \measuredangle(\mathbf{a}, \mathbf{b})$.

In the case of normalized vectors, the lengths |**a**| and |**b**| are both 1.

If the dot product **N**·**L** is negative, the light source is on the “wrong” side of the surface and we should set the reflection to 0. This can be achieved by using max(0,

$\mathbf{N}\cdot\mathbf{L}$), which makes sure that the value of the dot product is clamped to 0 for negative dot products. Furthermore, the reflected light depends on the intensity of the incoming light I_{incoming} and a material constant k_{diffuse} for the diffuse reflection: for a black surface, the material constant k_{diffuse} is 0, for a white surface it is 1. The equation for the diffuse reflected intensity is then:

$$I_{\text{diffuse}} = I_{\text{incoming}}\, k_{\text{diffuse}} \max(0, \mathbf{N}\cdot\mathbf{L})$$

For colored light, this equation applies to each color component (e.g. red, green, and blue). Thus, if the variables I_{diffuse}, I_{incoming}, and k_{diffuse} denote color vectors and the multiplications are performed component-wise (which they are for vectors in GLSL), this equation also applies to colored light. This is what we actually use in the shader code.

10.0.60. Shader Code for One Directional Light Source

If we have only one directional light source, the shader code for implementing the equation for I_{diffuse} is relatively small. In order to implement the equation, we follow the questions about implementing equations, which were discussed in GLSL PROGRAMMING/UNITY/SILHOUETTE ENHANCEMENT[1]:

- Should the equation be implemented in the vertex shader or the fragment shader? We try the vertex shader here. In GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[2], we will look at an implementation in the fragment shader.
- In which coordinate system should the equation be implemented? We try world space by default in Unity. (Which turns out to be a good choice here because Unity provides the light direction in world space.)
- Where do we get the parameters from? The answer to this is a bit longer:

We use a shader property (see GLSL PROGRAMMING/UNITY/SHADING IN WORLD SPACE[3]) to let the user specify the diffuse material color k_{diffuse}. We can get the direction to the light source in world space from the Unity-specific uniform `_WorldSpaceLightPos0` and the light color I_{incoming} from the Unity-specific uniform `_LightColor0`. As mentioned in GLSL PROGRAMMING/UNITY/SHADING IN WORLD SPACE[4], we have to tag the shader pass with `Tags {"LightMode" = "ForwardBase"}` to make sure that these uniforms have

1 Chapter 9 on page 67
2 Chapter 13 on page 111
3 Chapter 5 on page 31
4 Chapter 5 on page 31

the correct values. (Below we will discuss what this tag actually means.) We get the surface normal vector in object coordinates from the attribute `gl_Normal`. Since we implement the equation in world space, we have to convert the surface normal vector from object space to world space as discussed in GLSL PROGRAMMING/UNITY/SILHOUETTE ENHANCEMENT[5].

The shader code then looks like this:

```
Shader "GLSL per-vertex diffuse lighting" {
   Properties {
      _Color ("Diffuse Material Color", Color) = (1,1,1,1)
   }
   SubShader {
      Pass {
         Tags { "LightMode" = "ForwardBase" }
            // make sure that all uniforms are correctly set

         GLSLPROGRAM

         uniform vec4 _Color; // shader property specified by users

         // The following built-in uniforms (except _LightColor0)
         // are also defined in "UnityCG.glslinc",
         // i.e. one could #include "UnityCG.glslinc"
         uniform mat4 _Object2World; // model matrix
         uniform mat4 _World2Object; // inverse model matrix
         uniform vec4 _WorldSpaceLightPos0;
            // direction to or position of light source
         uniform vec4 _LightColor0;
            // color of light source (from "Lighting.cginc")

         varying vec4 color;
            // the diffuse lighting computed in the vertex shader

         #ifdef VERTEX

         void main()
         {
            mat4 modelMatrix = _Object2World;
            mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
               // is unnecessary because we normalize vectors

            vec3 normalDirection = normalize(
               vec3(vec4(gl_Normal, 0.0) * modelMatrixInverse));
            vec3 lightDirection = normalize(
               vec3(_WorldSpaceLightPos0));

            vec3 diffuseReflection = vec3(_LightColor0) * vec3(_Color)
               * max(0.0, dot(normalDirection, lightDirection));

            color = vec4(diffuseReflection, 1.0);
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```

5 Chapter 9 on page 67

```
         }

         #endif

         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor = color;
         }

         #endif

         ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Diffuse"
}
```

When you use this shader, make sure that there is only one light source in the scene, which has to be directional. If there is no light source, you can create a directional light source by selecting **Game Object > Create Other > Directional Light** from the main menu. Also, make sure that the "Forward Rendering Path" is active by selecting **Edit > Project Settings > Player** and then in the **Inspector View** the **Per-Platform Settings > Other Settings > Rendering > Rendering Path** should be set to **Forward**. (See below for more details about the "Forward Rendering Path".)

10.0.61. Fallback Shaders

The line `Fallback "Diffuse"` in the shader code defines a built-in fallback shader in case Unity doesn't find an appropriate subshader. For our example, Unity would use the fallback shader if it doesn't use the "forward rendering path" (see below) or if it couldn't compile the shader code. By choosing the specific name "_Color" for our shader property, we make sure that this built-in fallback shader can also access it. The source code of the built-in shaders is available at UNITY'S WEBSITE[6]. Inspection of this source code appears to be the only way to determine a suitable fallback shader and the names of the properties that it is using.

As mentioned, Unity will also use the fallback shader if there is a compile error in the shader code. In this case, the error is only be reported in the **Inspector View** of the shader; thus, it might be difficult to understand that the fallback shader is

6 HTTP://UNITY3D.COM/SUPPORT/RESOURCES/ASSETS/BUILT-IN-SHADERS

being used. Therefore, it is usually a good idea to comment the fallback instruction out during development of a shader but include it in the final version for better compatibility.

10.0.62. Shader Code for Multiple Directional (Pixel) Lights

So far, we have only considered a single light source. In order to handle multiple light sources, Unity chooses various techniques depending on the rendering and quality settings. In the tutorials here, we will only cover the "Forward Rendering Path". In order to choose it, select **Edit > Project Settings > Player** and then in the Inspector View set **Per-Platform Settings > Other Settings > Rendering > Rendering Path** to **Forward**. (Moreover, all cameras should be configured to use the player settings, which they are by default.)

In this tutorial we consider only Unity's so-called **pixel lights**. For the first pixel light (which always is a directional light), Unity calls the shader pass tagged with `Tags { "LightMode" = "ForwardBase" }` (as in our code above). For each additional pixel light, Unity calls the shader pass tagged with `Tags { "LightMode" = "ForwardAdd" }`. In order to make sure that all lights are rendered as pixel lights, you have to make sure that the quality settings allow for enough pixel lights: Select **Edit > Project Settings > Quality** and then increase the number labeled **Pixel Light Count** in any of the quality settings that you use. If there are more light sources in the scene than pixel light count allows for, Unity renders only the most important lights as pixel lights. Alternatively, you can set the **Render Mode** of all light sources to **Important** in order render them as pixel lights. (See GLSL PROGRAMMING/UNITY/MULTIPLE LIGHTS[7] for a discussion of the less important **vertex lights**.)

Our shader code so far is OK for the `ForwardBase` pass. For the `ForwardAdd` pass, we need to add the reflected light to the light that is already stored in the framebuffer. To this end, we just have to customize the blending to add the new fragment color (`gl_FragColor`) to the color in the framebuffer. As discussed in GLSL PROGRAMMING/UNITY/TRANSPARENCY[8], this is achieved by an additive blend equation, which is specified by this line:

```
Blend One One
```

Blending automatically clamps all results between 0 and 1; thus, we don't have to worry about colors or alpha values greater than 1.

7 Chapter 15 on page 127
8 Chapter 7 on page 49

All in all, our new shader for multiple directional lights becomes:

```
Shader "GLSL per-vertex diffuse lighting" {
   Properties {
      _Color ("Diffuse Material Color", Color) = (1,1,1,1)
   }
   SubShader {
      Pass {
         Tags { "LightMode" = "ForwardBase" }
            // pass for first light source

         GLSLPROGRAM

         uniform vec4 _Color; // shader property specified by users

         // The following built-in uniforms (except _LightColor0)
         // are also defined in "UnityCG.glslinc",
         // i.e. one could #include "UnityCG.glslinc"
         uniform mat4 _Object2World; // model matrix
         uniform mat4 _World2Object; // inverse model matrix
         uniform vec4 _WorldSpaceLightPos0;
            // direction to or position of light source
         uniform vec4 _LightColor0;
            // color of light source (from "Lighting.cginc")

         varying vec4 color;
            // the diffuse lighting computed in the vertex shader

         #ifdef VERTEX

         void main()
         {
            mat4 modelMatrix = _Object2World;
            mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
                // is unnecessary because we normalize vectors

            vec3 normalDirection = normalize(
               vec3(vec4(gl_Normal, 0.0) * modelMatrixInverse));
            vec3 lightDirection = normalize(
               vec3(_WorldSpaceLightPos0));

            vec3 diffuseReflection = vec3(_LightColor0) * vec3(_Color)
               * max(0.0, dot(normalDirection, lightDirection));

            color = vec4(diffuseReflection, 1.0);
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif

         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor = color;
         }
```

```
      #endif

      ENDGLSL
   }

   Pass {
      Tags { "LightMode" = "ForwardAdd" }
         // pass for additional light sources
      Blend One One // additive blending

      GLSLPROGRAM

      uniform vec4 _Color; // shader property specified by users

      // The following built-in uniforms (except _LightColor0)
      // are also defined in "UnityCG.glslinc",
      // i.e. one could #include "UnityCG.glslinc"
      uniform mat4 _Object2World; // model matrix
      uniform mat4 _World2Object; // inverse model matrix
      uniform vec4 _WorldSpaceLightPos0;
         // direction to or position of light source
      uniform vec4 _LightColor0;
         // color of light source (from "Lighting.cginc")

      varying vec4 color;
         // the diffuse lighting computed in the vertex shader

      #ifdef VERTEX

      void main()
      {
         mat4 modelMatrix = _Object2World;
         mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
            // is unnecessary because we normalize vectors

         vec3 normalDirection = normalize(
            vec3(vec4(gl_Normal, 0.0) * modelMatrixInverse));
         vec3 lightDirection = normalize(
            vec3(_WorldSpaceLightPos0));

         vec3 diffuseReflection = vec3(_LightColor0) * vec3(_Color)
            * max(0.0, dot(normalDirection, lightDirection));

         color = vec4(diffuseReflection, 1.0);
         gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
      }

      #endif

      #ifdef FRAGMENT

      void main()
      {
         gl_FragColor = color;
      }

      #endif
```

```
         ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Diffuse"
}
```

This appears to be a rather long shader; however, both passes are identical apart from the tag and the `Blend` setting in the `ForwardAdd` pass.

10.0.63. Changes for a Point Light Source

In the case of a directional light source `_WorldSpaceLightPos0` specifies the direction from where light is coming. In the case of a point light source (or a spot light source), however, `_WorldSpaceLightPos0` specifies the position of the light source in world space and we have to compute the direction to the light source as the difference vector from the position of the vertex in world space to the position of the light source. Since the 4th coordinate of a point is 1 and the 4th coordinate of a direction is 0, we can easily distinguish between the two cases:

```
vec3 lightDirection;

if (0.0 == _WorldSpaceLightPos0.w) // directional light?
{
   lightDirection = normalize(vec3(_WorldSpaceLightPos0));
}
else // point or spot light
{
   lightDirection = normalize(
      vec3(_WorldSpaceLightPos0 - modelMatrix * gl_Vertex));
}
```

While there is no attenuation of light for directional light sources, we should add some attenuation with distance to point and spot light source. As light spreads out from a point in three dimensions, it's covering ever larger virtual spheres at larger distances. Since the surface of these spheres increases quadratically with increasing radius and the total amount of light per sphere is the same, the amount of light per area decreases quadratically with increasing distance from the point light source. Thus, we should divide the intensity of the light source by the squared distance to the vertex.

Since a quadratic attenuation is rather rapid, we use a linear attenuation with distance, i.e. we divide the intensity by the distance instead of the squared distance. The code could be:

```
            vec3 lightDirection;
            float attenuation;

            if (0.0 == _WorldSpaceLightPos0.w) // directional light?
            {
               attenuation = 1.0; // no attenuation
               lightDirection = normalize(vec3(_WorldSpaceLightPos0));
            }
            else // point or spot light
            {
               vec3 vertexToLightSource = vec3(_WorldSpaceLightPos0
                  - modelMatrix * gl_Vertex);
               float distance = length(vertexToLightSource);
               attenuation = 1.0 / distance; // linear attenuation
               lightDirection = normalize(vertexToLightSource);
            }
```

The factor `attenuation` should then be multiplied with `_LightColor0` to compute the incoming light; see the shader code below. Note that spot light sources have additional features, which are beyond the scope of this tutorial.

Also note that this code is unlikely to give you the best performance because any `if` is usually quite costly. Since `_WorldSpaceLightPos0.w` is either 0 or 1, it is actually not too hard to rewrite the code to avoid the use of `if` and optimize a bit further:

```
            vec3 vertexToLightSource = vec3(_WorldSpaceLightPos0
               - modelMatrix * gl_Vertex * _WorldSpaceLightPos0.w);
            float one_over_distance =
               1.0 / length(vertexToLightSource);
            float attenuation =
               mix(1.0, one_over_distance, _WorldSpaceLightPos0.w);
            vec3 lightDirection =
               vertexToLightSource * one_over_distance;
```

However, we will use the version with `if` for clarity. (“Keep it simple, stupid!”)

The complete shader code for multiple directional and point lights is:

```
Shader "GLSL per-vertex diffuse lighting" {
   Properties {
      _Color ("Diffuse Material Color", Color) = (1,1,1,1)
   }
   SubShader {
      Pass {
         Tags { "LightMode" = "ForwardBase" }
            // pass for first light source

         GLSLPROGRAM

         uniform vec4 _Color; // shader property specified by users

         // The following built-in uniforms (except _LightColor0)
```

```
// are also defined in "UnityCG.glslinc",
// i.e. one could #include "UnityCG.glslinc"
uniform mat4 _Object2World; // model matrix
uniform mat4 _World2Object; // inverse model matrix
uniform vec4 _WorldSpaceLightPos0;
   // direction to or position of light source
uniform vec4 _LightColor0;
   // color of light source (from "Lighting.cginc")

varying vec4 color;
   // the diffuse lighting computed in the vertex shader

#ifdef VERTEX

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   vec3 normalDirection = normalize(
      vec3(vec4(gl_Normal, 0.0) * modelMatrixInverse));
   vec3 lightDirection;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
   }
   else // point or spot light
   {
      vec3 vertexToLightSource = vec3(_WorldSpaceLightPos0
         - modelMatrix * gl_Vertex);
      float distance = length(vertexToLightSource);
      attenuation = 1.0 / distance; // linear attenuation
      lightDirection = normalize(vertexToLightSource);
   }

   vec3 diffuseReflection =
      attenuation * vec3(_LightColor0) * vec3(_Color)
      * max(0.0, dot(normalDirection, lightDirection));

   color = vec4(diffuseReflection, 1.0);
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
   gl_FragColor = color;
}

#endif
```

```
      ENDGLSL
   }

   Pass {
      Tags { "LightMode" = "ForwardAdd" }
         // pass for additional light sources
      Blend One One // additive blending

      GLSLPROGRAM

      uniform vec4 _Color; // shader property specified by users

      // The following built-in uniforms (except _LightColor0)
      // are also defined in "UnityCG.glslinc",
      // i.e. one could #include "UnityCG.glslinc"
      uniform mat4 _Object2World; // model matrix
      uniform mat4 _World2Object; // inverse model matrix
      uniform vec4 _WorldSpaceLightPos0;
         // direction to or position of light source
      uniform vec4 _LightColor0;
         // color of light source (from "Lighting.cginc")

      varying vec4 color;
         // the diffuse lighting computed in the vertex shader

      #ifdef VERTEX

      void main()
      {
         mat4 modelMatrix = _Object2World;
         mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
            // is unnecessary because we normalize vectors

         vec3 normalDirection = normalize(
            vec3(vec4(gl_Normal, 0.0) * modelMatrixInverse));
         vec3 lightDirection;
         float attenuation;

         if (0.0 == _WorldSpaceLightPos0.w) // directional light?
         {
            attenuation = 1.0; // no attenuation
            lightDirection = normalize(vec3(_WorldSpaceLightPos0));
         }
         else // point or spot light
         {
            vec3 vertexToLightSource = vec3(_WorldSpaceLightPos0
               - modelMatrix * gl_Vertex);
            float distance = length(vertexToLightSource);
            attenuation = 1.0 / distance; // linear attenuation
            lightDirection = normalize(vertexToLightSource);
         }

         vec3 diffuseReflection =
            attenuation * vec3(_LightColor0) * vec3(_Color)
            * max(0.0, dot(normalDirection, lightDirection));

         color = vec4(diffuseReflection, 1.0);
         gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```

```
         }

         #endif

         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor = color;
         }

         #endif

         ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Diffuse"
}
```

Note that the light source in the `ForwardBase` pass always is a directional light; thus, the code for the first pass could actually be simplified. On the other hand, using the same GLSL code for both passes, makes it easier to copy & paste the code from one pass to the other in case we have to edit the shader code.

If there is a problem with the shader, remember to activate the "Forward Rendering Path" by selecting **Edit > Project Settings > Player** and then in the **Inspector View** set **Per-Platform Settings > Other Settings > Rendering > Rendering Path** to **Forward**.

10.0.64. Changes for a Spotlight

Unity implements spotlights with the help of cookie textures as described in GLSL PROGRAMMING/UNITY/COOKIES[9]; however, this is somewhat advanced. Here, we treat spotlights as if they were point lights.

10.0.65. Summary

Congratulations! You just learned how Unity's per-pixel lights work. This is essential for the following tutorials about more advanced lighting. We have also seen:

- What diffuse reflection is and how to describe it mathematically.

9 Chapter 23 on page 207

- How to implement diffuse reflection for a single directional light source in a shader.
- How to extend the shader for point light sources with an attenuation similar to the one used by Unity.
- How to further extend the shader to handle multiple per-pixel lights.

10.0.66. Further Reading

If you still want to know more

- about transforming normal vectors into world space, you should read GLSL PROGRAMMING/UNITY/SHADING IN WORLD SPACE[10].
- about uniform variables provided by Unity and shader properties, you should read GLSL PROGRAMMING/UNITY/SHADING IN WORLD SPACE[11].
- about (additive) blending, you should read GLSL PROGRAMMING/UNITY/-TRANSPARENCY[12].
- about pass tags in Unity (e.g. `ForwardBase` or `ForwardAdd`), you should read UNITY'S SHADERLAB REFERENCE ABOUT PASS TAGS[13].
- about how Unity processes light sources in general, you should read UNITY'S MANUAL ABOUT RENDERING PATHS[14].

10 Chapter 5 on page 31
11 Chapter 5 on page 31
12 Chapter 7 on page 49
13 HTTP://UNITY3D.COM/SUPPORT/DOCUMENTATION/COMPONENTS/SL-PASSTAGS.HTML
14 HTTP://UNITY3D.COM/SUPPORT/DOCUMENTATION/MANUAL/RENDERINGPATHS.HTML

11. Specular Highlights

Figure 13 “Apollo the Lute Player” (Badminton House version) by Michelangelo Merisi da Caravaggio, ca. 1596.

This tutorial covers **per-vertex lighting** (also known as **Gouraud shading**) using the **Phong reflection model**.

It extends the shader code in GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[1] by two additional terms: ambient lighting and specular reflection. Together, the three terms constitute the Phong reflection model. If you haven't read GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[2], this would be a very good opportunity to read it.

1 Chapter 10 on page 77
2 Chapter 10 on page 77

11.0.67. Ambient Light

Consider the painting by Caravaggio below . While large parts of the white shirt are in shadows, no part of it is completely black. Apparently there is always some light being reflected from walls and other objects to illuminate everything in the scene — at least to a certain degree. In the Phong reflection model, this effect is taken into account by ambient lighting, which depends on a general ambient light intensity $I_{\text{ambient light}}$ and the material color k_{diffuse} for diffuse reflection. In an equation for the intensity of ambient lighting I_{ambient}:

$$I_{\text{ambient}} = I_{\text{ambient light}}\, k_{\text{diffuse}}$$

Analogously to the equation for diffuse reflection in GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[3], this equation can also be interpreted as a vector equation for the red, green, and blue components of light.

In Unity, the ambient light is specified by choosing **Edit > Render Settings** from the main menu. In a GLSL shader in Unity, this color is always available as `gl_LightModel.ambient`, which is one of the pre-defined uniforms of the OpenGL compatibility profile mentioned in GLSL PROGRAMMING/UNITY/SHADING IN WORLD SPACE[4].

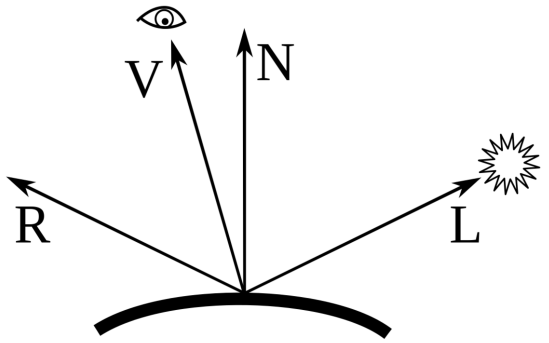

Figure 14 The computation of the specular reflection requires the surface normal vector N, the direction to the light source L, the reflected direction to the light source R, and the direction to the viewer V.

3 Chapter 10 on page 77
4 Chapter 5 on page 31

11.0.68. Specular Highlights

If you have a closer look at Caravaggio's painting, you will see several specular highlights: on the nose, on the hair, on the lips, on the lute, on the violin, on the bow, on the fruits, etc. The Phong reflection model includes a specular reflection term that can simulate such highlights on shiny surfaces; it even includes a parameter $n_{\text{shininess}}$ to specify a shininess of the material. The shininess specifies how small the highlights are: the shinier, the smaller the highlights.

A perfectly shiny surface will reflect light from the light source only in the geometrically reflected direction **R**. For less than perfectly shiny surfaces, light is reflected to directions around **R**: the smaller the shininess, the wider the spreading. Mathematically, the normalized reflected direction **R** is defined by:

$$\mathbf{R} = 2\mathbf{N}(\mathbf{N} \cdot \mathbf{L}) - \mathbf{L}$$

for a normalized surface normal vector **N** and a normalized direction to the light source **L**. In GLSL, the function `vec3 reflect(vec3 I, vec3 N)` (or `vec4 reflect(vec4 I, vec4 N)`) computes the same reflected vector but for the direction `I` from the light source to the point on the surface. Thus, we have to negate our direction **L** to use this function.

The specular reflection term computes the specular reflection in the direction of the viewer **V**. As discussed above, the intensity should be large if **V** is close to **R**, where "closeness" is parametrized by the shininess $n_{\text{shininess}}$. In the Phong reflection model, the cosine of the angle between **R** and **V** to the $n_{\text{shininess}}$-th power is used to generate highlights of different shininess. Similarly to the case of the DIFFUSE REFLECTION[5], we should clamp negative cosines to 0. Furthermore, the specular term requires a material color k_{specular} for the specular reflection, which is usually just white such that all highlights have the color of the incoming light I_{incoming}. For example, all highlights in Caravaggio's painting are white. The specular term of the Phong reflection model is then:

$$I_{\text{specular}} = I_{\text{incoming}}\, k_{\text{specular}} \max(0, \mathbf{R} \cdot \mathbf{V})^{n_{\text{shininess}}}$$

Analogously to the case of the DIFFUSE REFLECTION[6], the specular term should be ignored if the light source is on the "wrong" side of the surface; i.e., if the dot product **N**·**L** is negative.

5 HTTP://EN.WIKIBOOKS.ORG/WIKI/%2FUNITY%2FDIFFUSE%20REFLECTION
6 HTTP://EN.WIKIBOOKS.ORG/WIKI/%2FUNITY%2FDIFFUSE%20REFLECTION

11.0.69. Shader Code

The shader code for the ambient lighting is straightforward with a component-wise vector-vector product:

```
vec3 ambientLighting = vec3(gl_LightModel.ambient) * vec3(_Color);
```

For the implementation of the specular reflection, we require the direction to the viewer in world space, which we can compute as the difference between the camera position and the vertex position (both in world space). The camera position in world space is provided by Unity in the uniform `_WorldSpaceCameraPos`; the vertex position can be transformed to world space as discussed in GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[7]. The equation of the specular term in world space could then be implemented like this:

```
vec3 viewDirection = normalize(vec3(
   vec4(_WorldSpaceCameraPos, 1.0)
   - modelMatrix * gl_Vertex));

vec3 specularReflection;
if (dot(normalDirection, lightDirection) < 0.0)
   // light source on the wrong side?
{
   specularReflection = vec3(0.0, 0.0, 0.0);
      // no specular reflection
}
else // light source on the right side
{
   specularReflection = attenuation * vec3(_LightColor0)
      * vec3(_SpecColor) * pow(max(0.0, dot(
      reflect(-lightDirection, normalDirection),
      viewDirection)), _Shininess);
}
```

This code snippet uses the same variables as the shader code in GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[8] and additionally the user-specified properties `_SpecColor` and `_Shininess`. (The names were specifically chosen such that the fallback shader can access them; see the discussion in GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[9].) `pow(a, b)` computes a^b.

If the ambient lighting is added to the first pass (we only need it once) and the specular reflection is added to both passes of the full shader of GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[10], it looks like this:

7 Chapter 10 on page 77
8 Chapter 10 on page 77
9 Chapter 10 on page 77
10 Chapter 10 on page 77

```
Shader "GLSL per-vertex lighting" {
   Properties {
      _Color ("Diffuse Material Color", Color) = (1,1,1,1)
      _SpecColor ("Specular Material Color", Color) = (1,1,1,1)
      _Shininess ("Shininess", Float) = 10
   }
   SubShader {
      Pass {
         Tags { "LightMode" = "ForwardBase" }
            // pass for ambient light and first light source

         GLSLPROGRAM

         // User-specified properties
         uniform vec4 _Color;
         uniform vec4 _SpecColor;
         uniform float _Shininess;

         // The following built-in uniforms (except _LightColor0)
         // are also defined in "UnityCG.glslinc",
         // i.e. one could #include "UnityCG.glslinc"
         uniform vec3 _WorldSpaceCameraPos;
            // camera position in world space
         uniform mat4 _Object2World; // model matrix
         uniform mat4 _World2Object; // inverse model matrix
         uniform vec4 _WorldSpaceLightPos0;
            // direction to or position of light source
         uniform vec4 _LightColor0;
            // color of light source (from "Lighting.cginc")

         varying vec4 color;
            // the Phong lighting computed in the vertex shader

         #ifdef VERTEX

         void main()
         {
            mat4 modelMatrix = _Object2World;
            mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
               // is unnecessary because we normalize vectors

            vec3 normalDirection = normalize(vec3(
               vec4(gl_Normal, 0.0) * modelMatrixInverse));
            vec3 viewDirection = normalize(vec3(
               vec4(_WorldSpaceCameraPos, 1.0)
               - modelMatrix * gl_Vertex));
            vec3 lightDirection;
            float attenuation;

            if (0.0 == _WorldSpaceLightPos0.w) // directional light?
            {
               attenuation = 1.0; // no attenuation
               lightDirection = normalize(vec3(_WorldSpaceLightPos0));
            }
            else // point or spot light
            {
               vec3 vertexToLightSource = vec3(_WorldSpaceLightPos0
```

```
            - modelMatrix * gl_Vertex);
         float distance = length(vertexToLightSource);
         attenuation = 1.0 / distance; // linear attenuation
         lightDirection = normalize(vertexToLightSource);
      }

      vec3 ambientLighting =
         vec3(gl_LightModel.ambient) * vec3(_Color);

      vec3 diffuseReflection =
         attenuation * vec3(_LightColor0) * vec3(_Color)
         * max(0.0, dot(normalDirection, lightDirection));

      vec3 specularReflection;
      if (dot(normalDirection, lightDirection) < 0.0)
         // light source on the wrong side?
      {
         specularReflection = vec3(0.0, 0.0, 0.0);
            // no specular reflection
      }
      else // light source on the right side
      {
         specularReflection = attenuation * vec3(_LightColor0)
            * vec3(_SpecColor) * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _Shininess);
      }

      color = vec4(ambientLighting + diffuseReflection
         + specularReflection, 1.0);
      gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
   }

   #endif

   #ifdef FRAGMENT

   void main()
   {
      gl_FragColor = color;
   }

   #endif

   ENDGLSL
}

Pass {
   Tags { "LightMode" = "ForwardAdd" }
      // pass for additional light sources
   Blend One One // additive blending

   GLSLPROGRAM

   // User-specified properties
   uniform vec4 _Color;
   uniform vec4 _SpecColor;
   uniform float _Shininess;
```

```
// The following built-in uniforms (except _LightColor0)
// are also defined in "UnityCG.glslinc",
// i.e. one could #include "UnityCG.glslinc"
uniform vec3 _WorldSpaceCameraPos;
   // camera position in world space
uniform mat4 _Object2World; // model matrix
uniform mat4 _World2Object; // inverse model matrix
uniform vec4 _WorldSpaceLightPos0;
   // direction to or position of light source
uniform vec4 _LightColor0;
   // color of light source (from "Lighting.cginc")

varying vec4 color;
   // the diffuse lighting computed in the vertex shader

#ifdef VERTEX

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   vec3 normalDirection = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));
   vec3 viewDirection = normalize(vec3(
      vec4(_WorldSpaceCameraPos, 1.0)
      - modelMatrix * gl_Vertex));
   vec3 lightDirection;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
   }
   else // point or spot light
   {
      vec3 vertexToLightSource = vec3(_WorldSpaceLightPos0
         - modelMatrix * gl_Vertex);
      float distance = length(vertexToLightSource);
      attenuation = 1.0 / distance; // linear attenuation
      lightDirection = normalize(vertexToLightSource);
   }

   // vec3 ambientLighting =
   //    vec3(gl_LightModel.ambient) * vec3(_Color);

   vec3 diffuseReflection =
      attenuation * vec3(_LightColor0) * vec3(_Color)
      * max(0.0, dot(normalDirection, lightDirection));

   vec3 specularReflection;
   if (dot(normalDirection, lightDirection) < 0.0)
      // light source on the wrong side?
   {
      specularReflection = vec3(0.0, 0.0, 0.0);
```

```
                  // no specular reflection
            }
            else // light source on the right side
            {
               specularReflection = attenuation * vec3(_LightColor0)
                  * vec3(_SpecColor) * pow(max(0.0, dot(
                  reflect(-lightDirection, normalDirection),
                  viewDirection)), _Shininess);
            }

            color = vec4(diffuseReflection + specularReflection, 1.0);
               // no ambient lighting in this pass
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif

         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor = color;
         }

         #endif

         ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Specular"
}
```

11.0.70. Summary

Congratulation, you just learned how to implement the Phong reflection model. In particular, we have seen:

- What the ambient lighting in the Phong reflection model is.
- What the specular reflection term in the Phong reflection model is.
- How these terms can be implemented in GLSL in Unity.

11.0.71. Further Reading

If you still want to know more

- about the shader code, you should read GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[11].

11 Chapter 10 on page 77

12. Two-Sided Surfaces

Figure 15 An algebraic surface that includes an oloid in the center. The rendering uses different colors for the two sides of the surface.

This tutorial covers **two-sided per-vertex lighting**.

It's part of a series of tutorials about basic lighting in Unity. In this tutorial, we extend GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[1] to render two-sided surfaces. If you haven't read GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[2], this would be a very good time to read it.

1 Chapter 11 on page 91
2 Chapter 11 on page 91

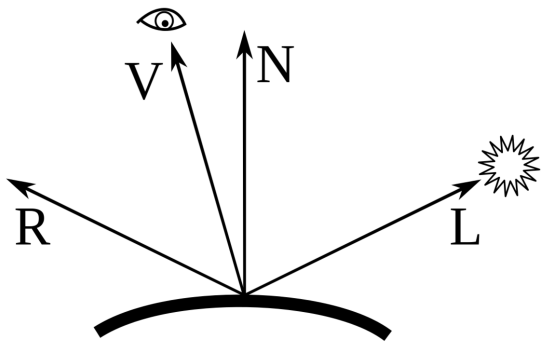

Figure 16 The surface normal vector N and the direction to the viewer V are usually on the same side of the surface but there are exceptions; thus, one shouldn't rely on it.

12.0.72. Two-Sided Lighting

As shown by the figure of the algebraic surface, it's sometimes useful to apply different colors to the two sides of a surface. In GLSL PROGRAMMING/UNITY/-CUTAWAYS[3], we have seen how a fragment shader can use the built-in variable `gl_FrontFacing` to determine whether a fragment is part of a front-facing or a back-facing triangle. Can a vertex shader also determine whether it is part of a front-facing or a back-facing triangle? The answer is a clear: **no!** One reason is that the same vertex can be part of a front-facing **and** a back-facing triangle at the same time; thus, whatever decision is made in the vertex shader, it is potentially wrong for some triangles. If you want a simple rule to remember: “Fragments are either front-facing or back-facing. Vertices are bi.”

Thus, two-sided per-vertex lighting has to let the fragment shader determine, whether the front or the back material color should be applied. For example, with this fragment shader:

```
#ifdef FRAGMENT

varying vec4 frontColor; // color for front face
varying vec4 backColor; // color for back face
```

3 Chapter 6 on page 43

```
        void main()
        {
           if (gl_FrontFacing) // is the fragment part of a front face?
           {
              gl_FragColor = frontColor;
           }
           else // fragment is part of a back face
           {
              gl_FragColor = backColor;
           }
        }

        #endif
```

On the other hand, this means that the vertex shader has to compute the surface lighting twice: for a front face and for a back face. Fortunately, this is usually still less work than computing the surface lighting for each fragment.

12.0.73. Shader Code

The shader code for two-sided per-vertex lighting is a straightforward extension of the code in GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[4]. It requires two sets of material parameters (front and back) and deactivates culling. The vertex shader computes two colors, one for front faces and one for back faces using the negated normal vector and the second set of material parameters. Then the fragment shader decides which one to apply.

```
Shader "GLSL two-sided per-vertex lighting" {
   Properties {
      _Color ("Front Material Diffuse Color", Color) = (1,1,1,1)
      _SpecColor ("Front Material Specular Color", Color) = (1,1,1,1)
      _Shininess ("Front Material Shininess", Float) = 10
      _BackColor ("Back Material Diffuse Color", Color) = (1,1,1,1)
      _BackSpecColor ("Back Material Specular Color", Color)
         = (1,1,1,1)
      _BackShininess ("Back Material Shininess", Float) = 10
   }
   SubShader {
      Pass {
         Tags { "LightMode" = "ForwardBase" }
            // pass for ambient light and first light source
         Cull Off // render front faces and back faces

         GLSLPROGRAM

         // User-specified properties
         uniform vec4 _Color;
         uniform vec4 _SpecColor;
```

4 Chapter 11 on page 91

```
uniform float _Shininess;
uniform vec4 _BackColor;
uniform vec4 _BackSpecColor;
uniform float _BackShininess;

// The following built-in uniforms (except _LightColor0)
// are also defined in "UnityCG.glslinc",
// i.e. one could #include "UnityCG.glslinc"
uniform vec3 _WorldSpaceCameraPos;
   // camera position in world space
uniform mat4 _Object2World; // model matrix
uniform mat4 _World2Object; // inverse model matrix
uniform vec4 _WorldSpaceLightPos0;
   // direction to or position of light source
uniform vec4 _LightColor0;
   // color of light source (from "Lighting.cginc")

varying vec4 frontColor;
   // lighting of front faces computed in the vertex shader
varying vec4 backColor;
   // lighting of back faces computed in the vertex shader

#ifdef VERTEX

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   vec3 normalDirection = normalize(vec3(vec4(gl_Normal, 0.0)
      * modelMatrixInverse));
   vec3 viewDirection = normalize(vec3(
       vec4(_WorldSpaceCameraPos, 1.0)
       - modelMatrix * gl_Vertex));
   vec3 lightDirection;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
   }
   else // point or spot light
   {
      vec3 vertexToLightSource = vec3(_WorldSpaceLightPos0
         - modelMatrix * gl_Vertex);
      float distance = length(vertexToLightSource);
      attenuation = 1.0 / distance; // linear attenuation
      lightDirection = normalize(vertexToLightSource);
   }

   // Computation of lighting for front faces

   vec3 ambientLighting =
      vec3(gl_LightModel.ambient) * vec3(_Color);

   vec3 diffuseReflection =
```

```
      attenuation * vec3(_LightColor0) * vec3(_Color)
      * max(0.0, dot(normalDirection, lightDirection));

   vec3 specularReflection;
   if (dot(normalDirection, lightDirection) < 0.0)
      // light source on the wrong side?
   {
      specularReflection = vec3(0.0, 0.0, 0.0);
         // no specular reflection
   }
   else // light source on the right side
   {
      specularReflection = attenuation * vec3(_LightColor0)
         * vec3(_SpecColor) * pow(max(0.0, dot(
         reflect(-lightDirection, normalDirection),
         viewDirection)), _Shininess);
   }

   frontColor = vec4(ambientLighting + diffuseReflection
      + specularReflection, 1.0);

   // Computation of lighting for back faces
   // (uses negative normalDirection and back material colors)

   vec3 backAmbientLighting =
      vec3(gl_LightModel.ambient) * vec3(_BackColor);

   vec3 backDiffuseReflection =
      attenuation * vec3(_LightColor0) * vec3(_BackColor)
      * max(0.0, dot(-normalDirection, lightDirection));
   vec3 backSpecularReflection;
   if (dot(-normalDirection, lightDirection) < 0.0)
      // light source on the wrong side?
   {
      backSpecularReflection = vec3(0.0, 0.0, 0.0);
         // no specular reflection
   }
   else // light source on the right side
   {
      backSpecularReflection =
         attenuation * vec3(_LightColor0)
         * vec3(_BackSpecColor) * pow(max(0.0, dot(
         reflect(-lightDirection, -normalDirection),
         viewDirection)), _BackShininess);
   }

   backColor = vec4(backAmbientLighting +
      backDiffuseReflection + backSpecularReflection, 1.0);

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
```

```
      if (gl_FrontFacing)
         // is the fragment part of a front face?
      {
         gl_FragColor = frontColor;
      }
      else // fragment is part of a back face
      {
         gl_FragColor = backColor;
      }
   }

   #endif

   ENDGLSL
}

Pass {
   Tags { "LightMode" = "ForwardAdd" }
      // pass for additional light sources
   Blend One One // additive blending
   Cull Off // render front faces and back faces

   GLSLPROGRAM

   // User-specified properties
   uniform vec4 _Color;
   uniform vec4 _SpecColor;
   uniform float _Shininess;
   uniform vec4 _BackColor;
   uniform vec4 _BackSpecColor;
   uniform float _BackShininess;

   // The following built-in uniforms (except _LightColor0)
   // are also defined in "UnityCG.glslinc",
   // i.e. one could #include "UnityCG.glslinc"
   uniform vec3 _WorldSpaceCameraPos;
      // camera position in world space
   uniform mat4 _Object2World; // model matrix
   uniform mat4 _World2Object; // inverse model matrix
   uniform vec4 _WorldSpaceLightPos0;
      // direction to or position of light source
   uniform vec4 _LightColor0;
      // color of light source (from "Lighting.cginc")

   varying vec4 frontColor;
      // lighting of front faces computed in the vertex shader
   varying vec4 backColor;
      // lighting of back faces computed in the vertex shader

   #ifdef VERTEX

   void main()
   {
      mat4 modelMatrix = _Object2World;
      mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
         // is unnecessary because we normalize vectors

      vec3 normalDirection = normalize(vec3(
```

```
   vec4(gl_Normal, 0.0) * modelMatrixInverse));
vec3 viewDirection = normalize(vec3(
   vec4(_WorldSpaceCameraPos, 1.0)
   - modelMatrix * gl_Vertex));
vec3 lightDirection;
float attenuation;

if (0.0 == _WorldSpaceLightPos0.w) // directional light?
{
   attenuation = 1.0; // no attenuation
   lightDirection = normalize(vec3(_WorldSpaceLightPos0));
}
else // point or spot light
{
   vec3 vertexToLightSource = vec3(_WorldSpaceLightPos0
      - modelMatrix * gl_Vertex);
   float distance = length(vertexToLightSource);
   attenuation = 1.0 / distance; // linear attenuation
   lightDirection = normalize(vertexToLightSource);
}

// Computation of lighting for front faces

vec3 diffuseReflection =
   attenuation * vec3(_LightColor0) * vec3(_Color)
   * max(0.0, dot(normalDirection, lightDirection));

vec3 specularReflection;
if (dot(normalDirection, lightDirection) < 0.0)
   // light source on the wrong side?
{
   specularReflection = vec3(0.0, 0.0, 0.0);
      // no specular reflection
}
else // light source on the right side
{
   specularReflection = attenuation * vec3(_LightColor0)
      * vec3(_SpecColor) * pow(max(0.0, dot(
      reflect(-lightDirection, normalDirection),
      viewDirection)), _Shininess);
}

frontColor = vec4(diffuseReflection
   + specularReflection, 1.0);

// Computation of lighting for back faces
// (uses negative normalDirection and back material colors)

vec3 backDiffuseReflection =
   attenuation * vec3(_LightColor0) * vec3(_BackColor)
   * max(0.0, dot(-normalDirection, lightDirection));
vec3 backSpecularReflection;
if (dot(-normalDirection, lightDirection) < 0.0)
   // light source on the wrong side?
{
   backSpecularReflection = vec3(0.0, 0.0, 0.0);
      // no specular reflection
}
```

```
            else // light source on the right side
            {
               backSpecularReflection =
                  attenuation * vec3(_LightColor0)
                  * vec3(_BackSpecColor) * pow(max(0.0, dot(
                  reflect(-lightDirection, -normalDirection),
                  viewDirection)), _BackShininess);
            }

            backColor = vec4(backDiffuseReflection
               + backSpecularReflection, 1.0);

            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif

         #ifdef FRAGMENT

         void main()
         {
            if (gl_FrontFacing)
               // is the fragment part of a front face?
            {
               gl_FragColor = frontColor;
            }
            else // fragment is part of a back face
            {
               gl_FragColor = backColor;
            }
         }

         #endif

         ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Specular"
}
```

Again, this code consists of two passes, where the second pass is the same as the first apart from the additive blending and the missing ambient color.

12.0.74. Summary

Congratulations, you made it to the end of this short tutorial with a long shader. We have seen:

- Why a vertex shader cannot distinguish between front-facing and back-facing vertices (because the same vertex might be part of a front-facing and a back-facing triangles).

- How to compute lighting for front faces and for back faces in the vertex shader.
- How to let the fragment shader decide which color to apply.

12.0.75. Further Reading

If you still want to know more

- about the shader version for single-sided surfaces, you should read GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[5].
- about front-facing and back-facing triangles in GLSL, you should read GLSL PROGRAMMING/UNITY/CUTAWAYS[6].

5 Chapter 11 on page 91
6 Chapter 6 on page 43

13. Smooth Specular Highlights

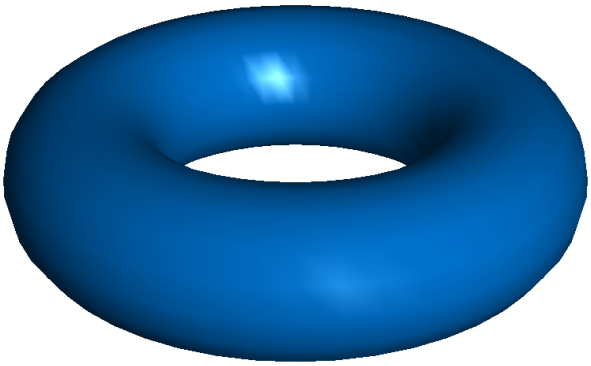

Figure 17 Rendering of a torus mesh with per-vertex lighting.

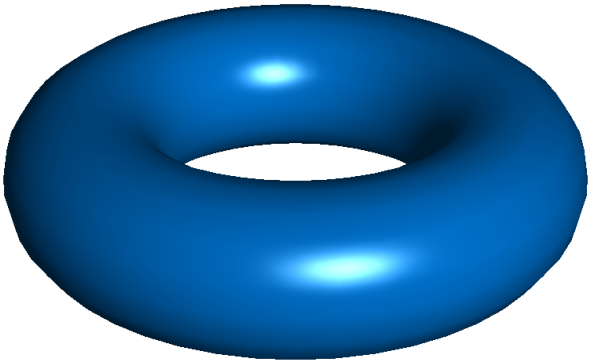

Figure 18 Rendering of a torus mesh with per-pixel lighting.

This tutorial covers **per-pixel lighting** (also known as **Phong shading**).

It is based on GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[1]. If you haven't read that tutorial yet, you should read it first. The main disadvantage of per-vertex lighting (i.e. of computing the surface lighting for each vertex and then interpolating the vertex colors) is the limited quality, in particular for specular highlights as demonstrated by the figure below . The remedy is per-pixel lighting which computes the lighting for each fragment based on an interpolated normal vector. While the resulting image quality is considerably higher, the performance costs are also significant.

13.0.76. Per-Pixel Lighting (Phong Shading)

Per-pixel lighting is also known as Phong shading (in contrast to per-vertex lighting, which is also known as Gouraud shading). This should not be confused with the Phong reflection model (also called Phong lighting), which computes the surface lighting by an ambient, a diffuse, and a specular term as discussed in GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[2].

The key idea of per-pixel lighting is easy to understand: normal vectors and positions are interpolated for each fragment and the lighting is computed in the fragment shader.

13.0.77. Shader Code

Apart from optimizations, implementing per-pixel lighting based on shader code for per-vertex lighting is straightforward: the lighting computation is moved from the vertex shader to the fragment shader and the vertex shader has to write the attributes required for the lighting computation to varyings. The fragment shader then uses these varyings to compute the lighting (instead of the attributes that the vertex shader used). That's about it.

In this tutorial, we adapt the shader code from GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[3] to per-pixel lighting. The result looks like this:

```
Shader "GLSL per-pixel lighting" {
   Properties {
      _Color ("Diffuse Material Color", Color) = (1,1,1,1)
      _SpecColor ("Specular Material Color", Color) = (1,1,1,1)
      _Shininess ("Shininess", Float) = 10
```

1 Chapter 11 on page 91
2 Chapter 11 on page 91
3 Chapter 11 on page 91

```
}
SubShader {
   Pass {
      Tags { "LightMode" = "ForwardBase" }
         // pass for ambient light and first light source

      GLSLPROGRAM

      // User-specified properties
      uniform vec4 _Color;
      uniform vec4 _SpecColor;
      uniform float _Shininess;

      // The following built-in uniforms (except _LightColor0)
      // are also defined in "UnityCG.glslinc",
      // i.e. one could #include "UnityCG.glslinc"
      uniform vec3 _WorldSpaceCameraPos;
         // camera position in world space
      uniform mat4 _Object2World; // model matrix
      uniform mat4 _World2Object; // inverse model matrix
      uniform vec4 _WorldSpaceLightPos0;
         // direction to or position of light source
      uniform vec4 _LightColor0;
         // color of light source (from "Lighting.cginc")

      varying vec4 position;
         // position of the vertex in world space
      varying vec3 varyingNormalDirection;
         // surface normal vector in world space

      #ifdef VERTEX

      void main()
      {
         mat4 modelMatrix = _Object2World;
         mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
            // is unnecessary because we normalize vectors

         position = modelMatrix * gl_Vertex;
         varyingNormalDirection = normalize(vec3(
            vec4(gl_Normal, 0.0) * modelMatrixInverse));

         gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
      }

      #endif

      #ifdef FRAGMENT

      void main()
      {
         vec3 normalDirection = normalize(varyingNormalDirection);

         vec3 viewDirection =
            normalize(_WorldSpaceCameraPos - vec3(position));
         vec3 lightDirection;
         float attenuation;
```

```
         if (0.0 == _WorldSpaceLightPos0.w) // directional light?
         {
            attenuation = 1.0; // no attenuation
            lightDirection = normalize(vec3(_WorldSpaceLightPos0));
         }
         else // point or spot light
         {
            vec3 vertexToLightSource =
               vec3(_WorldSpaceLightPos0 - position);
            float distance = length(vertexToLightSource);
            attenuation = 1.0 / distance; // linear attenuation
            lightDirection = normalize(vertexToLightSource);
         }

         vec3 ambientLighting =
            vec3(gl_LightModel.ambient) * vec3(_Color);

         vec3 diffuseReflection =
            attenuation * vec3(_LightColor0) * vec3(_Color)
            * max(0.0, dot(normalDirection, lightDirection));

         vec3 specularReflection;
         if (dot(normalDirection, lightDirection) < 0.0)
            // light source on the wrong side?
         {
            specularReflection = vec3(0.0, 0.0, 0.0);
               // no specular reflection
         }
         else // light source on the right side
         {
            specularReflection = attenuation * vec3(_LightColor0)
               * vec3(_SpecColor) * pow(max(0.0, dot(
               reflect(-lightDirection, normalDirection),
               viewDirection)), _Shininess);
         }

         gl_FragColor = vec4(ambientLighting + diffuseReflection
            + specularReflection, 1.0);
      }

      #endif

      ENDGLSL
   }

   Pass {
      Tags { "LightMode" = "ForwardAdd" }
         // pass for additional light sources
      Blend One One // additive blending

      GLSLPROGRAM

      // User-specified properties
      uniform vec4 _Color;
      uniform vec4 _SpecColor;
      uniform float _Shininess;

      // The following built-in uniforms (except _LightColor0)
```

```
// are also defined in "UnityCG.glslinc",
// i.e. one could #include "UnityCG.glslinc"
uniform vec3 _WorldSpaceCameraPos;
   // camera position in world space
uniform mat4 _Object2World; // model matrix
uniform mat4 _World2Object; // inverse model matrix
uniform vec4 _WorldSpaceLightPos0;
   // direction to or position of light source
uniform vec4 _LightColor0;
   // color of light source (from "Lighting.cginc")

varying vec4 position;
   // position of the vertex in world space
varying vec3 varyingNormalDirection;
   // surface normal vector in world space

#ifdef VERTEX

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   position = modelMatrix * gl_Vertex;
   varyingNormalDirection = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
   vec3 normalDirection = normalize(varyingNormalDirection);

   vec3 viewDirection =
      normalize(_WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
   }
   else // point or spot light
   {
      vec3 vertexToLightSource =
         vec3(_WorldSpaceLightPos0 - position);
      float distance = length(vertexToLightSource);
      attenuation = 1.0 / distance; // linear attenuation
      lightDirection = normalize(vertexToLightSource);
   }
```

```
            vec3 diffuseReflection =
               attenuation * vec3(_LightColor0) * vec3(_Color)
               * max(0.0, dot(normalDirection, lightDirection));

            vec3 specularReflection;
            if (dot(normalDirection, lightDirection) < 0.0)
               // light source on the wrong side?
            {
               specularReflection = vec3(0.0, 0.0, 0.0);
                  // no specular reflection
            }
            else // light source on the right side
            {
               specularReflection = attenuation * vec3(_LightColor0)
                  * vec3(_SpecColor) * pow(max(0.0, dot(
                  reflect(-lightDirection, normalDirection),
                  viewDirection)), _Shininess);
            }

            gl_FragColor =
               vec4(diffuseReflection + specularReflection, 1.0);
         }

         #endif

         ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Specular"
}
```

Note that the vertex shader writes a normalized vector to `varyingNormalDirection` in order to make sure that all directions are weighted equally in the interpolation. The fragment shader normalizes it again because the interpolated directions are no longer normalized.

13.0.78. Summary

Congratulations, now you know how per-pixel Phong lighting works. We have seen:

- Why the quality provided by per-vertex lighting is sometimes insufficient (in particular because of specular highlights).
- How per-pixel lighting works and how to implement it based on a shader for per-vertex lighting.

13.0.79. Further Reading

If you still want to know more

- about the shader version for per-vertex lighting, you should read GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[4].

4 Chapter 11 on page 91

14. Two-Sided Smooth Surfaces

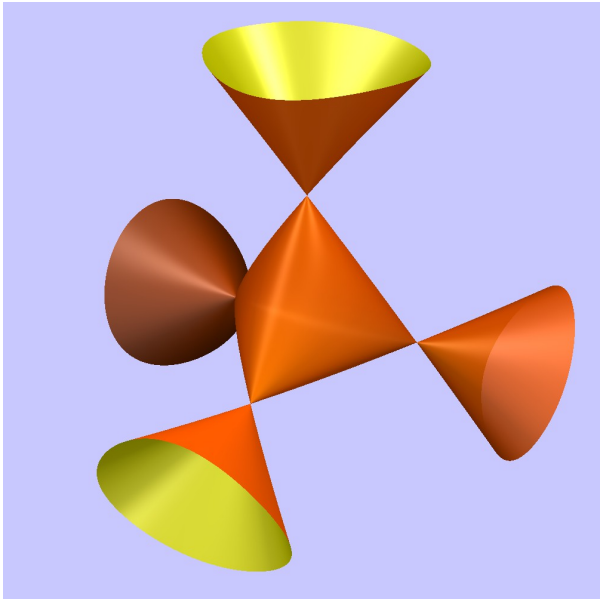

Figure 19 Rendering of Cayley's nodal cubic surface using different colors on the two sides of the surface.

This tutorial covers **two-sided per-pixel lighting** (i.e. **two-sided Phong shading**).

Here we combine the per-pixel lighting discussed in GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[1] with the two-sided lighting discussed in GLSL PROGRAMMING/UNITY/TWO-SIDED SURFACES[2].

1 Chapter 13 on page 111
2 Chapter 12 on page 101

14.0.80. Shader Coder

The required changes to the code of GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[3] are: new properties for the back material, deactivation of culling, new local variables for the material parameters in the fragment shader, which are set either to the front material parameters or the back material parameters according to `gl_FrontFacing`. Also, the surface normal vector is negated in case a back face is rendered. It's actually quite straightforward. The code looks like this:

```
Shader "GLSL two-sided per-pixel lighting" {
   Properties {
      _Color ("Front Material Diffuse Color", Color) = (1,1,1,1)
      _SpecColor ("Front Material Specular Color", Color) = (1,1,1,1)
      _Shininess ("Front Material Shininess", Float) = 10
      _BackColor ("Back Material Diffuse Color", Color) = (1,1,1,1)
      _BackSpecColor ("Back Material Specular Color", Color)
         = (1,1,1,1)
      _BackShininess ("Back Material Shininess", Float) = 10
   }
   SubShader {
      Pass {
         Tags { "LightMode" = "ForwardBase" }
            // pass for ambient light and first light source
         Cull Off

         GLSLPROGRAM

         // User-specified properties
         uniform vec4 _Color;
         uniform vec4 _SpecColor;
         uniform float _Shininess;
         uniform vec4 _BackColor;
         uniform vec4 _BackSpecColor;
         uniform float _BackShininess;

         // The following built-in uniforms (except _LightColor0)
         // are also defined in "UnityCG.glslinc",
         // i.e. one could #include "UnityCG.glslinc"
         uniform vec3 _WorldSpaceCameraPos;
            // camera position in world space
         uniform mat4 _Object2World; // model matrix
         uniform mat4 _World2Object; // inverse model matrix
         uniform vec4 _WorldSpaceLightPos0;
            // direction to or position of light source
         uniform vec4 _LightColor0;
            // color of light source (from "Lighting.cginc")

         varying vec4 position;
            // position of the vertex in world space
```

3 Chapter 13 on page 111

```
varying vec3 varyingNormalDirection;
   // surface normal vector in world space

#ifdef VERTEX

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   position = modelMatrix * gl_Vertex;
   varyingNormalDirection = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
   vec3 normalDirection = normalize(varyingNormalDirection);
   vec4 diffuseColor;
   vec4 specularColor;
   float shininess;

   if (gl_FrontFacing)
   {
      diffuseColor = _Color;
      specularColor = _SpecColor;
      shininess = _Shininess;
   }
   else
   {
      diffuseColor = _BackColor;
      specularColor = _BackSpecColor;
      shininess = _BackShininess;
      normalDirection = -normalDirection;
   }

   vec3 viewDirection = normalize(
      _WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
   }
   else // point or spot light
   {
      vec3 vertexToLightSource =
         vec3(_WorldSpaceLightPos0 - position);
      float distance = length(vertexToLightSource);
```

```
            attenuation = 1.0 / distance; // linear attenuation
            lightDirection = normalize(vertexToLightSource);
         }

         vec3 ambientLighting =
            vec3(gl_LightModel.ambient) * vec3(diffuseColor);

         vec3 diffuseReflection =
            attenuation * vec3(_LightColor0) * vec3(diffuseColor)
            * max(0.0, dot(normalDirection, lightDirection));

         vec3 specularReflection;
         if (dot(normalDirection, lightDirection) < 0.0)
            // light source on the wrong side?
         {
            specularReflection = vec3(0.0, 0.0, 0.0);
               // no specular reflection
         }
         else // light source on the right side
         {
            specularReflection = attenuation * vec3(_LightColor0)
               * vec3(specularColor) * pow(max(0.0, dot(
               reflect(-lightDirection, normalDirection),
               viewDirection)), shininess);
         }

         gl_FragColor = vec4(ambientLighting + diffuseReflection
            + specularReflection, 1.0);
      }

      #endif

      ENDGLSL
   }

   Pass {
      Tags { "LightMode" = "ForwardAdd" }
         // pass for additional light sources
      Blend One One // additive blending
      Cull Off

      GLSLPROGRAM

      // User-specified properties
      uniform vec4 _Color;
      uniform vec4 _SpecColor;
      uniform float _Shininess;
      uniform vec4 _BackColor;
      uniform vec4 _BackSpecColor;
      uniform float _BackShininess;

      // The following built-in uniforms (except _LightColor0)
      // are also defined in "UnityCG.glslinc",
      // i.e. one could #include "UnityCG.glslinc"
      uniform vec3 _WorldSpaceCameraPos;
         // camera position in world space
      uniform mat4 _Object2World; // model matrix
      uniform mat4 _World2Object; // inverse model matrix
```

```
uniform vec4 _WorldSpaceLightPos0;
   // direction to or position of light source
uniform vec4 _LightColor0;
   // color of light source (from "Lighting.cginc")

varying vec4 position;
   // position of the vertex in world space
varying vec3 varyingNormalDirection;
   // surface normal vector in world space

#ifdef VERTEX

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   position = modelMatrix * gl_Vertex;
   varyingNormalDirection = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
   vec3 normalDirection = normalize(varyingNormalDirection);
   vec4 diffuseColor;
   vec4 specularColor;
   float shininess;

   if (gl_FrontFacing)
   {
      diffuseColor = _Color;
      specularColor = _SpecColor;
      shininess = _Shininess;
   }
   else
   {
      diffuseColor = _BackColor;
      specularColor = _BackSpecColor;
      shininess = _BackShininess;
      normalDirection = -normalDirection;
   }

   vec3 viewDirection = normalize(
      _WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
```

```
            lightDirection = normalize(vec3(_WorldSpaceLightPos0));
         }
         else // point or spot light
         {
            vec3 vertexToLightSource =
               vec3(_WorldSpaceLightPos0 - position);
            float distance = length(vertexToLightSource);
            attenuation = 1.0 / distance; // linear attenuation
            lightDirection = normalize(vertexToLightSource);
         }

         vec3 diffuseReflection =
            attenuation * vec3(_LightColor0) * vec3(diffuseColor)
            * max(0.0, dot(normalDirection, lightDirection));

         vec3 specularReflection;
         if (dot(normalDirection, lightDirection) < 0.0)
            // light source on the wrong side?
         {
            specularReflection = vec3(0.0, 0.0, 0.0);
               // no specular reflection
         }
         else // light source on the right side
         {
            specularReflection = attenuation * vec3(_LightColor0)
               * vec3(specularColor) * pow(max(0.0, dot(
               reflect(-lightDirection, normalDirection),
               viewDirection)), shininess);
         }

         gl_FragColor =
            vec4(diffuseReflection + specularReflection, 1.0);
      }

      #endif

      ENDGLSL
    }
  }
  // The definition of a fallback shader should be commented out
  // during development:
  // Fallback "Specular"
}
```

As always, the only difference between the two passes is the lack of ambient lighting and the additive blending in the second pass.

14.0.81. Summary

Congratulations, you have reached the end of this short tutorial. We have seen:

- How two-sided surfaces can be rendered with per-pixel lighting.

14.0.82. Further Reading

If you still want to know more

- about the shader version for single-sided per-pixel lighting, you should read GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[4].
- about the shader version for two-sided per-vertex lighting, you should read GLSL PROGRAMMING/UNITY/TWO-SIDED SURFACES[5].

4 Chapter 13 on page 111
5 Chapter 12 on page 101

15. Multiple Lights

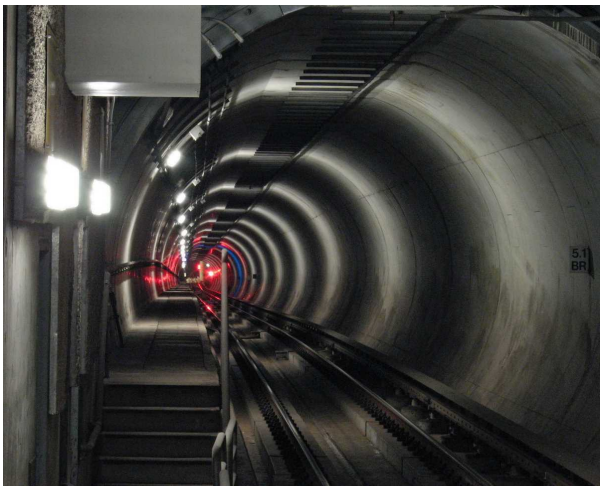

Figure 20 Multiple subway lights of limited range in a tunnel.

This tutorial covers **lighting by multiple light sources in one pass**. In particular, it covers Unity's so-called “vertex lights” in the `ForwardBase` pass.

This tutorial is an extension of GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[1]. If you haven't read that tutorial, you should read it first.

15.0.83. Multiple Lights in One Pass

As discussed in GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[2], Unity's forward rendering path uses separate passes for the most important light sources. These are called “pixel lights” because the built-in shaders render them with per-pixel lighting. All light sources with the **Render Mode** set to **Important** are

1 Chapter 13 on page 111
2 Chapter 10 on page 77

rendered as pixel lights. If the **Pixel Light Count** of the **Quality** project settings allows for more pixel lights, then some of the light sources with **Render Mode** set to **Auto** are also rendered as pixel lights. What happens to the other light sources? The built-in shaders of Unity render four additional lights as **vertex lights** in the `ForwardBase` pass. As the name indicates, the built-in shaders render these lights with per-vertex lighting. This is what this tutorial is about. (Further lights are approximated by spherical harmonic lighting, which is not covered here.)

Unfortunately, it is somewhat unclear how to access the four vertex lights (i.e. their positions and colors). Here is, what appears to work in Unity 3.4 on Windows and MacOS X:

```
// Built-in uniforms for "vertex lights"
uniform vec4 unity_LightColor[4];
   // array of the colors of the 4 light sources
uniform vec4 unity_4LightPosX0;
   // x coordinates of the 4 light sources in world space
uniform vec4 unity_4LightPosY0;
   // y coordinates of the 4 light sources in world space
uniform vec4 unity_4LightPosZ0;
   // z coordinates of the 4 light sources in world space
uniform vec4 unity_4LightAtten0;
   // scale factors for attenuation with squared distance
// uniform vec4 unity_LightPosition[4] is apparently not
// always correctly set in Unity 3.4
// uniform vec4 unity_LightAtten[4] is apparently not
// always correctly set in Unity 3.4
```

Depending on your platform and version of Unity you might have to use `unity_LightPosition[4]` instead of `unity_4LightPosX0`, `unity_4LightPosY0`, and `unity_4LightPosZ0`. Similarly, you might have to use `unity_LightAtten[4]` instead of `unity_4LightAtten0`. Note what's not available: neither any cookie texture nor the transformation to light space (and therefore neither the direction of spotlights). Also, no 4th component of the light positions is available; thus, it is unclear whether a vertex light is a directional light, a point light, or a spotlight.

Here, we follow Unity's built-in shaders and only compute the diffuse reflection by vertex light using per-vertex lighting. This can be computed with the following for-loop inside the vertex shader:

```
vertexLighting = vec3(0.0, 0.0, 0.0);
for (int index = 0; index < 4; index++)
{
   vec4 lightPosition = vec4(unity_4LightPosX0[index],
      unity_4LightPosY0[index],
      unity_4LightPosZ0[index], 1.0);

   vec3 vertexToLightSource =
```

```
                vec3(lightPosition - position);
            vec3 lightDirection = normalize(vertexToLightSource);
            float squaredDistance =
               dot(vertexToLightSource, vertexToLightSource);
            float attenuation = 1.0 / (1.0  +
                unity_4LightAtten0[index] * squaredDistance);
            vec3 diffuseReflection =
               attenuation * vec3(unity_LightColor[index])
               * vec3(_Color) * max(0.0,
               dot(varyingNormalDirection, lightDirection));

            vertexLighting = vertexLighting + diffuseReflection;
         }
```

The total diffuse lighting by all vertex lights is accumulated in `vertexLighting` by initializing it to black and then adding the diffuse reflection of each vertex light to the previous value of `vertexLighting` at the end of the for-loop. A for-loop should be familiar to any C/C++/Java/JavaScript programmer. Note that for-loops are sometimes severely limited; in particular the limits (here: 0 and 4) have to be constants in Unity, i.e. you cannot even use uniforms to determine the limits. (The technical reason is that the limits have to be known at compile time in order to "un-roll" the loop.)

This is more or less how vertex lights are computed in Unity's built-in shaders. However, remember that nothing would stop you from computing specular reflection or per-pixel lighting with these "vertex lights".

15.0.84. Complete Shader Code

In the context of the shader code from GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[3], the complete shader code is:

```
Shader "GLSL per-pixel lighting with vertex lights" {
   Properties {
      _Color ("Diffuse Material Color", Color) = (1,1,1,1)
      _SpecColor ("Specular Material Color", Color) = (1,1,1,1)
      _Shininess ("Shininess", Float) = 10
   }
   SubShader {
      Pass {
         Tags { "LightMode" = "ForwardBase" } // pass for
            // 4 vertex lights, ambient light & first pixel light

         GLSLPROGRAM
         #pragma multi_compile_fwdbase

         // User-specified properties
```

3 Chapter 13 on page 111

```
uniform vec4 _Color;
uniform vec4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (except _LightColor0)
// are also defined in "UnityCG.glslinc",
// i.e. one could #include "UnityCG.glslinc"
uniform vec3 _WorldSpaceCameraPos;
   // camera position in world space
uniform mat4 _Object2World; // model matrix
uniform mat4 _World2Object; // inverse model matrix
uniform vec4 _WorldSpaceLightPos0;
   // direction to or position of light source
uniform vec4 _LightColor0;
   // color of light source (from "Lighting.cginc")

// Built-in uniforms for "vertex lights"
uniform vec4 unity_LightColor[4];
uniform vec4 unity_4LightPosX0;
   // x coordinates of the 4 light sources in world space
uniform vec4 unity_4LightPosY0;
   // y coordinates of the 4 light sources in world space
uniform vec4 unity_4LightPosZ0;
   // z coordinates of the 4 light sources in world space
uniform vec4 unity_4LightAtten0;
   // scale factors for attenuation with squared distance

// Varyings
varying vec4 position;
   // position of the vertex (and fragment) in world space
varying vec3 varyingNormalDirection;
   // surface normal vector in world space
varying vec3 vertexLighting;

#ifdef VERTEX

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   position = modelMatrix * gl_Vertex;
   varyingNormalDirection = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

   // Diffuse reflection by four "vertex lights"
   vertexLighting = vec3(0.0, 0.0, 0.0);
   #ifdef VERTEXLIGHT_ON
   for (int index = 0; index < 4; index++)
   {
      vec4 lightPosition = vec4(unity_4LightPosX0[index],
         unity_4LightPosY0[index],
         unity_4LightPosZ0[index], 1.0);

      vec3 vertexToLightSource =
```

```
            vec3(lightPosition - position);
         vec3 lightDirection = normalize(vertexToLightSource);
         float squaredDistance =
            dot(vertexToLightSource, vertexToLightSource);
         float attenuation = 1.0 / (1.0 +
            unity_4LightAtten0[index] * squaredDistance);
         vec3 diffuseReflection =
            attenuation * vec3(unity_LightColor[index])
            * vec3(_Color) * max(0.0,
            dot(varyingNormalDirection, lightDirection));

         vertexLighting = vertexLighting + diffuseReflection;
      }
      #endif
   }

   #endif

   #ifdef FRAGMENT

   void main()
   {
      vec3 normalDirection = normalize(varyingNormalDirection);
      vec3 viewDirection =
         normalize(_WorldSpaceCameraPos - vec3(position));
      vec3 lightDirection;
      float attenuation;

      if (0.0 == _WorldSpaceLightPos0.w) // directional light?
      {
         attenuation = 1.0; // no attenuation
         lightDirection = normalize(vec3(_WorldSpaceLightPos0));
      }
      else // point or spot light
      {
         vec3 vertexToLightSource =
            vec3(_WorldSpaceLightPos0 - position);
         float distance = length(vertexToLightSource);
         attenuation = 1.0 / distance; // linear attenuation
         lightDirection = normalize(vertexToLightSource);
      }

      vec3 ambientLighting =
          vec3(gl_LightModel.ambient) * vec3(_Color);

      vec3 diffuseReflection =
         attenuation * vec3(_LightColor0) * vec3(_Color)
         * max(0.0, dot(normalDirection, lightDirection));

      vec3 specularReflection;
      if (dot(normalDirection, lightDirection) < 0.0)
         // light source on the wrong side?
      {
         specularReflection = vec3(0.0, 0.0, 0.0);
            // no specular reflection
      }
      else // light source on the right side
      {
```

```
            specularReflection = attenuation * vec3(_LightColor0)
               * vec3(_SpecColor) * pow(max(0.0, dot(
               reflect(-lightDirection, normalDirection),
               viewDirection)), _Shininess);
         }

         gl_FragColor = vec4(vertexLighting + ambientLighting
            + diffuseReflection + specularReflection, 1.0);
      }

      #endif

      ENDGLSL
   }

   Pass {
      Tags { "LightMode" = "ForwardAdd" }
         // pass for additional "pixel lights"
      Blend One One // additive blending

      GLSLPROGRAM

      // User-specified properties
      uniform vec4 _Color;
      uniform vec4 _SpecColor;
      uniform float _Shininess;

      // The following built-in uniforms (except _LightColor0)
      // are also defined in "UnityCG.glslinc",
      // i.e. one could #include "UnityCG.glslinc"
      uniform vec3 _WorldSpaceCameraPos;
         // camera position in world space
      uniform mat4 _Object2World; // model matrix
      uniform mat4 _World2Object; // inverse model matrix
      uniform vec4 _WorldSpaceLightPos0;
         // direction to or position of light source
      uniform vec4 _LightColor0;
         // color of light source (from "Lighting.cginc")

      // Varyings
      varying vec4 position;
         // position of the vertex (and fragment) in world space
      varying vec3 varyingNormalDirection;
         // surface normal vector in world space

      #ifdef VERTEX

      void main()
      {
         mat4 modelMatrix = _Object2World;
         mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
            // is unnecessary because we normalize vectors

         position = modelMatrix * gl_Vertex;
         varyingNormalDirection = normalize(vec3(
            vec4(gl_Normal, 0.0) * modelMatrixInverse));

         gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```

```
      }

      #endif

      #ifdef FRAGMENT

      void main()
      {
         vec3 normalDirection = normalize(varyingNormalDirection);

         vec3 viewDirection =
            normalize(_WorldSpaceCameraPos - vec3(position));
         vec3 lightDirection;
         float attenuation;

         if (0.0 == _WorldSpaceLightPos0.w) // directional light?
         {
            attenuation = 1.0; // no attenuation
            lightDirection = normalize(vec3(_WorldSpaceLightPos0));
         }
         else // point or spot light
         {
            vec3 vertexToLightSource =
               vec3(_WorldSpaceLightPos0 - position);
            float distance = length(vertexToLightSource);
            attenuation = 1.0 / distance; // linear attenuation
            lightDirection = normalize(vertexToLightSource);
         }

         vec3 diffuseReflection =
            attenuation * vec3(_LightColor0) * vec3(_Color)
            * max(0.0, dot(normalDirection, lightDirection));

         vec3 specularReflection;
         if (dot(normalDirection, lightDirection) < 0.0)
            // light source on the wrong side?
         {
            specularReflection = vec3(0.0, 0.0, 0.0);
               // no specular reflection
         }
         else // light source on the right side
         {
            specularReflection = attenuation * vec3(_LightColor0)
               * vec3(_SpecColor) * pow(max(0.0, dot(
               reflect(-lightDirection, normalDirection),
               viewDirection)), _Shininess);
         }

         gl_FragColor = vec4(diffuseReflection
            + specularReflection, 1.0);
      }

      #endif

      ENDGLSL
   }
}
// The definition of a fallback shader should be commented out
```

```
   // during development:
   // Fallback "Specular"
}
```

The use of `#pragma multi_compile_fwdbase` and `#ifdef VERTEXLIGHT_ON ... #endif` appears to be necessary to make sure that no vertex lighting is computed when Unity doesn't provide the data.

15.0.85. Summary

Congratulations, you have reached the end of this tutorial. We have seen:

- How Unity's vertex lights are specified.
- How a for-loop can be used in GLSL to compute the lighting of multiple lights in one pass.

15.0.86. Further Reading

If you still want to know more

- about other parts of the shader code, you should read GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[4].
- about Unity's forward rendering path and what is computed in the `ForwardBase` pass, you should read UNITY'S REFERENCE ABOUT FORWARD RENDERING[5].

4 Chapter 13 on page 111

5 HTTP://UNITY3D.COM/SUPPORT/DOCUMENTATION/COMPONENTS/RENDERTECH-FORWARDRENDERING.HTML

Part IV.

Basic Texturing

16. Textured Spheres

Figure 21 The Earth seen from Apollo 17. The shape of the Earth is close to a quite smooth sphere.

This tutorial introduces **texture mapping**.

It's the first in a series of tutorials about texturing in GLSL shaders in Unity. In this tutorial, we start with a single texture map on a sphere. More specifically, we map an image of the Earth's surface onto a sphere. Based on this, further tutorials cover topics such as lighting of textured surfaces, transparent textures, multitexturing, gloss mapping, etc.

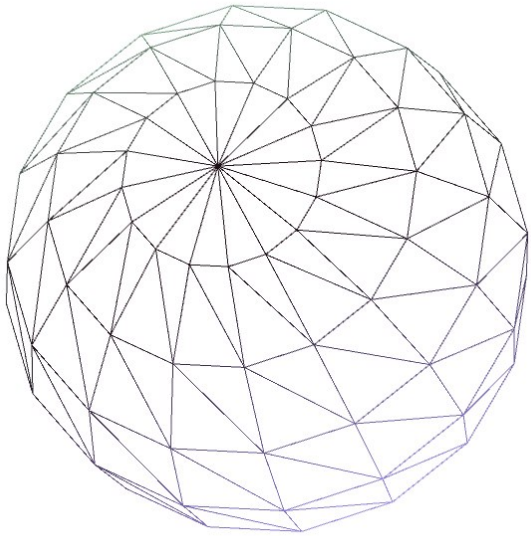

Figure 22 A triangle mesh approximating a sphere.

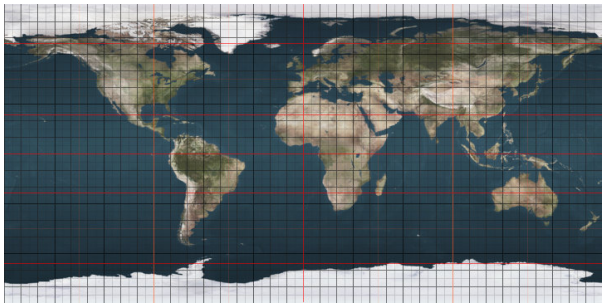

Figure 23 An image of the Earth's surface. The horizontal coordinate represents the longitude, the vertical coordinate the latitude.

16.0.87. Texture Mapping

The basic idea of “texture mapping” (or “texturing”) is to map an image (i.e. a “texture” or a “texture map”) onto a triangle mesh; in other words, to put a flat image onto the surface of a three-dimensional shape.

To this end, “texture coordinates” are defined, which simply specify the position in the texture (i.e. image). The horizontal coordinate is officially called `S` and the vertical coordinate `T`. However, it is very common to refer to them as `x` and `y`. In animation and modeling tools, texture coordinates are usually called `U` and `V`.

In order to map the texture image to a mesh, every vertex of the mesh is given a pair of texture coordinates. (This process (and the result) is sometimes called “UV mapping” since each vertex is mapped to a point in the UV-space.) Thus, every vertex is mapped to a point in the texture image. The texture coordinates of the vertices can then be interpolated for each point of any triangle between three vertices and thus every point of all triangles of the mesh can have a pair of (interpolated) texture coordinates. These texture coordinates map each point of the mesh to a specific position in the texture map and therefore to the color at this position. Thus, rendering a texture-mapped mesh consists of two steps for all visible points: interpolation of texture coordinates and a look-up of the color of the texture image at the position specified by the interpolated texture coordinates.

In OpenGL, any valid floating-point number is a valid texture coordinate. However, when the GPU is asked to look up a pixel (or “texel”) of a texture image (e.g. with the “texture2D” instruction described below), it will internally map the texture coordinates to the range between 0 and 1 in a way depending on the “Wrap Mode” that is specified when importing the texture: wrap mode “repeat” basically uses the fractional part of the texture coordinates to determine texture coordinates in the range between 0 and 1. On the other hand, wrap mode “clamp” clamps the texture coordinates to this range. These internal texture coordinates in the range between 0 and 1 are then used to determine the position in the texture image: $(0,0)$ specifies the lower, left corner of the texture image; $(1,0)$ the lower, right corner; $(0,1)$ the upper, left corner; etc.

16.0.88. Texturing a Sphere in Unity

To map the image of the Earth's surface onto a sphere in Unity, you first have to import the image into Unity. Click the IMAGE[1] until you get to a larger version and save it (usually with a right-click) to your computer (remember where you saved it). Then switch to Unity and choose **Assets > Import New Asset...** from the main menu. Choose the image file and click on **Import** in the file selector box. The imported texture image should appear in the **Project View**. By selecting it there,

1 HTTP://COMMONS.WIKIMEDIA.ORG/WIKI/FILE:EARTHMAP720X360_GRID.JPG

details about the way it is imported appear (and can be changed) in the **Inspector View**.

Now create a sphere, a material, and a shader, and attach the shader to the material and the material to the sphere as described in GLSL PROGRAMMING/UNITY/MINIMAL SHADER[2]. The shader code should be:

```
Shader "GLSL shader with single texture" {
   Properties {
      _MainTex ("Texture Image", 2D) = "white" {}
         // a 2D texture property that we call "_MainTex", which should
         // be labeled "Texture Image" in Unity's user interface.
         // By default we use the built-in texture "white"
         // (alternatives: "black", "gray" and "bump").
   }
   SubShader {
      Pass {
         GLSLPROGRAM

         uniform sampler2D _MainTex;
            // a uniform variable refering to the property above
            // (in fact, this is just a small integer specifying a
            // "texture unit", which has the texture image "bound"
            // to it; Unity takes care of this).

         varying vec4 textureCoordinates;
            // the texture coordinates at the vertices,
            // which are interpolated for each fragment

         #ifdef VERTEX

         void main()
         {
            textureCoordinates = gl_MultiTexCoord0;
               // Unity provides default longitude-latitude-like
               // texture coordinates at all vertices of a
               // sphere mesh as the attribute "gl_MultiTexCoord0".
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif

         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor =
               texture2D(_MainTex, vec2(textureCoordinates));
               // look up the color of the texture image specified by
               // the uniform "_MainTex" at the position specified by
               // "textureCoordinates.x" and "textureCoordinates.y"
               // and return it in "gl_FragColor"
         }
```

2 Chapter 2 on page 9

```
         #endif

         ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Unlit/Texture"
}
```

Note that the name `_MainTex` was chosen to make sure that the fallback shader `Unlit/Texture` can access it (see the discussion of fallback shaders in GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[3]).

The sphere should now be white. If it is grey, you should check whether the shader is attached to the material and the material is attached to the sphere. If the sphere is magenta, you should check the shader code. In particular, you should select the shader in the **Project View** and read the error message in the **Inspector View**.

If the sphere is white, select the sphere in the **Hierarchy View** or the **Scene View** and look at the information in the **Inspector View**. Your material should appear under **Mesh Renderer** and under it should be a label **Texture Image**. (Otherwise click on the material bar to make it appear.) The label "Texture Image" is the same that we specified for our shader property `_MainTex` in the shader code. There is an empty box to the right of this label. Either click on the small **Select** button in the box and select the imported texture image or drag & drop the texture image from the **Project View** to this empty box.

If everything went right, the texture image should now appear on the sphere. Congratulations!

16.0.89. How It Works

Since many techniques use texture mapping, it pays off very well to understand what is happening here. Therefore, let's review the shader code:

The vertices of Unity's sphere object come with attribute data in `gl_MultiTexCoord0` for each vertex, which specifies texture coordinates that are similar to longitude and latitude (but range from 0 to 1). This is analogous to the attribute `gl_Vertex`, which specifies a position in object space, except that `gl_MultiTexCoord0` specifies texture coordinates in the space of the texture image.

3 Chapter 10 on page 77

The vertex shader then writes the texture coordinates of each vertex to the varying variable `textureCoordinates`. For each fragment of a triangle (i.e. each covered pixel), the values of this varying at the three triangle vertices are interpolated (see the description in GLSL PROGRAMMING/RASTERIZATION[4]) and the interpolated texture coordinates are given to the fragment shader. The fragment shader then uses them to look up a color in the texture image specified by the uniform `_MainTex` at the interpolated position in texture space and returns this color in `gl_FragColor`, which is then written to the framebuffer and displayed on the screen.

It is crucial that you gain a good idea of these steps in order to understand the more complicated texture mapping techniques presented in other tutorials.

16.0.90. Repeating and Moving Textures

In Unity's interface for the shader above, you might have noticed the parameters **Tiling** and **Offset**, each with an **x** and a **y** component. In built-in shaders, these parameters allow you to repeat the texture (by shrinking the texture image in texture coordinate space) and move the texture image on the surface (by offsetting it in texture coordinate space). In order to be consistent with this behavior, another uniform has to be defined:

```
uniform vec4 _MainTex_ST;
   // tiling and offset parameters of property "_MainTex"
```

For each texture property, Unity offers such a `vec4` uniform with the ending "_ST". (Remember: "S" and "T" are the official names of the texture coordinates, which are usually called "U" and "V", or "x" and "y".) This uniform holds the **x** and **y** components of the **Tiling** parameter in `_MainTex_ST.x` and `_MainTex_ST.y`, while the **x** and **y** components of the **Offset** parameter are stored in `_MainTex_ST.w` and `_MainTex_ST.z`. The uniform should be used like this:

```
gl_FragColor = texture2D(_MainTex,
   _MainTex_ST.xy * textureCoordinates.xy
   + _MainTex_ST.zw);
```

This makes the shader behave like the built-in shaders. In the other tutorials, this feature is usually not implemented in order to keep the shader code a bit cleaner.

And just for completeness, here is the complete shader code with this feature:

4 Chapter 46 on page 437

```
Shader "GLSL shader with single texture" {
   Properties {
      _MainTex ("Texture Image", 2D) = "white" {}
   }
   SubShader {
      Pass {
         GLSLPROGRAM

         uniform sampler2D _MainTex;
         uniform vec4 _MainTex_ST;
            // tiling and offset parameters of property

         varying vec4 textureCoordinates;

         #ifdef VERTEX

         void main()
         {
            textureCoordinates = gl_MultiTexCoord0;
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif

         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor = texture2D(_MainTex,
               _MainTex_ST.xy * textureCoordinates.xy
               + _MainTex_ST.zw);
               // textureCoordinates are multiplied with the tiling
               // parameters and the offset parameters are added
         }

         #endif

         ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Unlit/Texture"
}
```

16.0.91. Summary

You have reached the end of one of the most important tutorials. We have looked at:

- How to import a texture image and how to attach it to a texture property of a shader.
- How a vertex shader and a fragment shader work together to map a texture image onto a mesh.

- How Unity's tiling and offset parameters for textures work and how to implement them.

16.0.92. Further Reading

If you want to know more

- about the data flow in and out of vertex shaders and fragment shaders (i.e. vertex attributes, varyings, etc.), you should read the description in GLSL PROGRAMMING/OPENGL ES 2.0 PIPELINE[5].
- about the interpolation of varying variables for the fragment shader, you should read the discussion in GLSL PROGRAMMING/RASTERIZATION[6].

5 Chapter 42 on page 399
6 Chapter 46 on page 437

17. Lighting Textured Surfaces

Figure 24 Earthrise as seen from Apollo 8.

This tutorial covers **per-vertex lighting of textured surfaces**.

It combines the shader code of GLSL PROGRAMMING/UNITY/TEXTURED SPHERES[1] and GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[2] to compute lighting with a diffuse material color determined by a texture. If you haven't read those sections, this would be a very good opportunity to read them.

1 Chapter 16 on page 137
2 Chapter 11 on page 91

17.0.93. Texturing and Diffuse Per-Vertex Lighting

In GLSL PROGRAMMING/UNITY/TEXTURED SPHERES[3], the texture color was used as output of the fragment shader. However, it is also possible to use the texture color as any of the parameters in lighting computations, in particular the material constant k_{diffuse} for diffuse reflection, which was introduced in GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[4]. It appears in the diffuse part of the Phong reflection model:

$$I_{\text{diffuse}} = I_{\text{incoming}}\, k_{\text{diffuse}} \max(0, \mathbf{N} \cdot \mathbf{L})$$

where this equation is used with different material constants for the three color components red, green, and blue. By using a texture to determine these material constants, they can vary over the surface.

17.0.94. Shader Code

In comparison to the per-vertex lighting in GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[5], the vertex shader here computes two varying colors: `diffuseColor` is multiplied with the texture color in the fragment shader and `specularColor` is just the specular term, which shouldn't be multiplied with the texture color. This makes perfect sense but for historically reasons (i.e. older graphics hardware that was less capable) this is sometimes referred to as "separate specular color"; in fact, Unity's ShaderLab has an OPTION CALLED "SEPARATE-SPECULAR"[6] to activate or deactivate it.

Note that a property `_Color` is included, which is multiplied (component-wise) to all parts of the `diffuseColor`; thus, it acts as a useful color filter to tint or shade the texture color. Moreover, a property with this name is required to make the fallback shader work (see also the discussion of fallback shaders in GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[7]).

```
Shader "GLSL per-vertex lighting with texture" {
   Properties {
      _MainTex ("Texture For Diffuse Material Color", 2D) = "white" {}
      _Color ("Overall Diffuse Color Filter", Color) = (1,1,1,1)
```

3 Chapter 16 on page 137
4 Chapter 10 on page 77
5 Chapter 11 on page 91
6 HTTP://UNITY3D.COM/SUPPORT/DOCUMENTATION/COMPONENTS/SL-MATERIAL.HTML
7 Chapter 10 on page 77

```
   _SpecColor ("Specular Material Color", Color) = (1,1,1,1)
   _Shininess ("Shininess", Float) = 10
}
SubShader {
   Pass {
      Tags { "LightMode" = "ForwardBase" }
         // pass for ambient light and first light source

      GLSLPROGRAM

      // User-specified properties
      uniform sampler2D _MainTex;
      uniform vec4 _Color;
      uniform vec4 _SpecColor;
      uniform float _Shininess;

      // The following built-in uniforms (except _LightColor0)
      // are also defined in "UnityCG.glslinc",
      // i.e. one could #include "UnityCG.glslinc"
      uniform vec3 _WorldSpaceCameraPos;
         // camera position in world space
      uniform mat4 _Object2World; // model matrix
      uniform mat4 _World2Object; // inverse model matrix
      uniform vec4 _WorldSpaceLightPos0;
         // direction to or position of light source
      uniform vec4 _LightColor0;
         // color of light source (from "Lighting.cginc")

      varying vec3 diffuseColor;
         // diffuse Phong lighting computed in the vertex shader
      varying vec3 specularColor;
         // specular Phong lighting computed in the vertex shader
      varying vec4 textureCoordinates;

      #ifdef VERTEX

      void main()
      {
         mat4 modelMatrix = _Object2World;
         mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
            // is unnecessary because we normalize vectors

         vec3 normalDirection = normalize(vec3(
            vec4(gl_Normal, 0.0) * modelMatrixInverse));
         vec3 viewDirection = normalize(vec3(
            vec4(_WorldSpaceCameraPos, 1.0)
            - modelMatrix * gl_Vertex));
         vec3 lightDirection;
         float attenuation;

         if (0.0 == _WorldSpaceLightPos0.w) // directional light?
         {
            attenuation = 1.0; // no attenuation
            lightDirection = normalize(vec3(_WorldSpaceLightPos0));
         }
         else // point or spot light
         {
            vec3 vertexToLightSource = vec3(_WorldSpaceLightPos0
```

```
            - modelMatrix * gl_Vertex);
         float distance = length(vertexToLightSource);
         attenuation = 1.0 / distance; // linear attenuation
         lightDirection = normalize(vertexToLightSource);
      }

      vec3 ambientLighting =
         vec3(gl_LightModel.ambient) * vec3(_Color);

      vec3 diffuseReflection =
         attenuation * vec3(_LightColor0) * vec3(_Color)
         * max(0.0, dot(normalDirection, lightDirection));

      vec3 specularReflection;
      if (dot(normalDirection, lightDirection) < 0.0)
         // light source on the wrong side?
      {
         specularReflection = vec3(0.0, 0.0, 0.0);
            // no specular reflection
      }
      else // light source on the right side
      {
         specularReflection = attenuation * vec3(_LightColor0)
            * vec3(_SpecColor) * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _Shininess);
      }

      diffuseColor = ambientLighting + diffuseReflection;
      specularColor = specularReflection;
      textureCoordinates = gl_MultiTexCoord0;
      gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
   }

   #endif

   #ifdef FRAGMENT

   void main()
   {
      gl_FragColor = vec4(diffuseColor
         * vec3(texture2D(_MainTex, vec2(textureCoordinates)))
         + specularColor, 1.0);
   }

   #endif

   ENDGLSL
}

Pass {
   Tags { "LightMode" = "ForwardAdd" }
      // pass for additional light sources
   Blend One One // additive blending

   GLSLPROGRAM

   // User-specified properties
```

```
uniform sampler2D _MainTex;
uniform vec4 _Color;
uniform vec4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (except _LightColor0)
// are also defined in "UnityCG.glslinc",
// i.e. one could #include "UnityCG.glslinc"
uniform vec3 _WorldSpaceCameraPos;
   // camera position in world space
uniform mat4 _Object2World; // model matrix
uniform mat4 _World2Object; // inverse model matrix
uniform vec4 _WorldSpaceLightPos0;
   // direction to or position of light source
uniform vec4 _LightColor0;
   // color of light source (from "Lighting.cginc")

varying vec3 diffuseColor;
   // diffuse Phong lighting computed in the vertex shader
varying vec3 specularColor;
   // specular Phong lighting computed in the vertex shader
varying vec4 textureCoordinates;

#ifdef VERTEX

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   vec3 normalDirection = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));
   vec3 viewDirection = normalize(vec3(
      vec4(_WorldSpaceCameraPos, 1.0)
      - modelMatrix * gl_Vertex));
   vec3 lightDirection;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
   }
   else // point or spot light
   {
      vec3 vertexToLightSource = vec3(_WorldSpaceLightPos0
         - modelMatrix * gl_Vertex);
      float distance = length(vertexToLightSource);
      attenuation = 1.0 / distance; // linear attenuation
      lightDirection = normalize(vertexToLightSource);
   }

   vec3 diffuseReflection =
      attenuation * vec3(_LightColor0) * vec3(_Color)
      * max(0.0, dot(normalDirection, lightDirection));

   vec3 specularReflection;
```

```
            if (dot(normalDirection, lightDirection) < 0.0)
               // light source on the wrong side?
            {
               specularReflection = vec3(0.0, 0.0, 0.0);
                  // no specular reflection
            }
            else // light source on the right side
            {
               specularReflection = attenuation * vec3(_LightColor0)
                  * vec3(_SpecColor) * pow(max(0.0, dot(
                  reflect(-lightDirection, normalDirection),
                  viewDirection)), _Shininess);
            }

            diffuseColor = diffuseReflection;
            specularColor = specularReflection;
            textureCoordinates = gl_MultiTexCoord0;
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif

         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor = vec4(diffuseColor
               * vec3(texture2D(_MainTex, vec2(textureCoordinates)))
               + specularColor, 1.0);
         }

         #endif

         ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Specular"
}
```

In order to assign a texture image to this shader, you should follow the steps discussed in GLSL PROGRAMMING/UNITY/TEXTURED SPHERES[8].

17.0.95. Summary

Congratulations, you have reached the end. We have looked at:

- How texturing and per-vertex lighting are usually combined.
- What a “separate specular color” is.

8 Chapter 16 on page 137

17.0.96. Further Reading

If you still want to know more

- about fallback shaders or the diffuse reflection term of the Phong reflection model, you should read GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[9].
- about per-vertex lighting or the rest of the Phong reflection model, i.e. the ambient and the specular term, you should read GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[10].
- about the basics of texturing, you should read GLSL PROGRAMMING/UNITY/TEXTURED SPHERES[11].

9 Chapter 10 on page 77
10 Chapter 11 on page 91
11 Chapter 16 on page 137

18. Glossy Textures

Figure 25 Sun set with a specular highlight in the Pacific Ocean as seen from the International Space Station (ISS).

This tutorial covers **per-pixel lighting of partially glossy, textured surfaces**.

It combines the shader code of GLSL PROGRAMMING/UNITY/TEXTURED SPHERES[1] and GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[2] to compute per-pixel lighting with a material color for diffuse reflection that is determined by the RGB components of a texture and an intensity of the specular reflection that is determined by the A component of the same texture. If you haven't read those sections, this would be a very good opportunity to read them.

18.0.97. Gloss Mapping

In GLSL PROGRAMMING/UNITY/LIGHTING TEXTURED SURFACES[3], the material constant for the diffuse reflection was determined by the RGB components of

1 Chapter 16 on page 137
2 Chapter 13 on page 111
3 Chapter 17 on page 145

a texture image. Here we extend this technique and determine the strength of the specular reflection by the A (alpha) component of the same texture image. Using only one texture offers a significant performance advantage, in particular because an RGBA texture lookup is under certain circumstances just as expensive as an RGB texture lookup.

If the "gloss" of a texture image (i.e. the strength of the specular reflection) is encoded in the A (alpha) component of an RGBA texture image, we can simply multiply the material constant for the specular reflection k_{specular} with the alpha component of the texture image. k_{specular} was introduced in GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[4] and appears in the specular reflection term of the Phong reflection model:

$$I_{\text{specular}} = I_{\text{incoming}}\, k_{\text{specular}} \max(0, \mathbf{R} \cdot \mathbf{V})^{n_{\text{shininess}}}$$

If multiplied with the alpha component of the texture image, this term reaches its maximum (i.e. the surface is glossy) where alpha is 1, and it is 0 (i.e. the surface is not glossy at all) where alpha is 0.

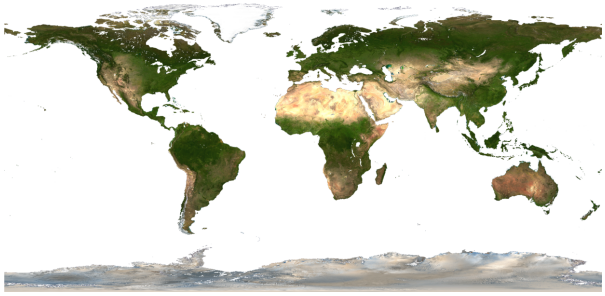

Figure 26 Map of the Earth with transparent water, i.e. the alpha component is 0 for water and 1 for land.

18.0.98. Shader Code for Per-Pixel Lighting

The shader code is a combination of the per-pixel lighting from GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[5] and the texturing from GLSL PROGRAMMING/UNITY/TEXTURED SPHERES[6]. Similarly to GLSL PROGRAM-

4 Chapter 11 on page 91
5 Chapter 13 on page 111
6 Chapter 16 on page 137

MING/UNITY/LIGHTING TEXTURED SURFACES[7], the RGB components of the texture color in `textureColor` is multiplied to the diffuse material color `_Color`.

In the particular texture image below , the alpha component is 0 for water and 1 for land. However, it should be the water that is glossy and the land that isn't. Thus, with this particular image, we should multiply the specular material color with `(1.0 - textureColor.a)`. On the other hand, usual gloss maps would require a multiplication with `textureColor.a`. (Note how easy it is to make this kind of changes to a shader program.)

```
Shader "GLSL per-pixel lighting with texture" {
   Properties {
      _MainTex ("RGBA Texture For Material Color", 2D) = "white" {}
      _Color ("Diffuse Material Color", Color) = (1,1,1,1)
      _SpecColor ("Specular Material Color", Color) = (1,1,1,1)
      _Shininess ("Shininess", Float) = 10
   }
   SubShader {
      Pass {
         Tags { "LightMode" = "ForwardBase" }
            // pass for ambient light and first light source

         GLSLPROGRAM

         // User-specified properties
         uniform sampler2D _MainTex;
         uniform vec4 _Color;
         uniform vec4 _SpecColor;
         uniform float _Shininess;

         // The following built-in uniforms (except _LightColor0)
         // are also defined in "UnityCG.glslinc",
         // i.e. one could #include "UnityCG.glslinc"
         uniform vec3 _WorldSpaceCameraPos;
            // camera position in world space
         uniform mat4 _Object2World; // model matrix
         uniform mat4 _World2Object; // inverse model matrix
         uniform vec4 _WorldSpaceLightPos0;
            // direction to or position of light source
         uniform vec4 _LightColor0;
            // color of light source (from "Lighting.cginc")

         varying vec4 position;
            // position of the vertex (and fragment) in world space
         varying vec3 varyingNormalDirection;
            // surface normal vector in world space
         varying vec4 textureCoordinates;

         #ifdef VERTEX
```

7 Chapter 17 on page 145

```
void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   position = modelMatrix * gl_Vertex;
   varyingNormalDirection = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
   textureCoordinates = gl_MultiTexCoord0;
}

#endif

#ifdef FRAGMENT

void main()
{
   vec3 normalDirection = normalize(varyingNormalDirection);

   vec3 viewDirection =
      normalize(_WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection;
   float attenuation;

   vec4 textureColor =
      texture2D(_MainTex, vec2(textureCoordinates));

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
   }
   else // point or spot light
   {
      vec3 vertexToLightSource =
         vec3(_WorldSpaceLightPos0 - position);
      float distance = length(vertexToLightSource);
      attenuation = 1.0 / distance; // linear attenuation
      lightDirection = normalize(vertexToLightSource);
   }

   vec3 ambientLighting = vec3(gl_LightModel.ambient)
      * vec3(_Color) * vec3(textureColor);

   vec3 diffuseReflection = attenuation * vec3(_LightColor0)
      * vec3(_Color) * vec3(textureColor)
      * max(0.0, dot(normalDirection, lightDirection));

   vec3 specularReflection;
   if (dot(normalDirection, lightDirection) < 0.0)
      // light source on the wrong side?
   {
      specularReflection = vec3(0.0, 0.0, 0.0);
         // no specular reflection
   }
```

```
      else // light source on the right side
      {
         specularReflection = attenuation * vec3(_LightColor0)
            * vec3(_SpecColor) * (1.0 - textureColor.a)
               // for usual gloss maps: "... * textureColor.a"
            * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _Shininess);
      }

      gl_FragColor = vec4(ambientLighting
         + diffuseReflection + specularReflection, 1.0);
   }

   #endif

   ENDGLSL
}

Pass {
   Tags { "LightMode" = "ForwardAdd" }
      // pass for additional light sources
   Blend One One // additive blending


   GLSLPROGRAM

   // User-specified properties
   uniform sampler2D _MainTex;
   uniform vec4 _Color;
   uniform vec4 _SpecColor;
   uniform float _Shininess;

   // The following built-in uniforms (except _LightColor0)
   // are also defined in "UnityCG.glslinc",
   // i.e. one could #include "UnityCG.glslinc"
   uniform vec3 _WorldSpaceCameraPos;
      // camera position in world space
   uniform mat4 _Object2World; // model matrix
   uniform mat4 _World2Object; // inverse model matrix
   uniform vec4 _WorldSpaceLightPos0;
      // direction to or position of light source
   uniform vec4 _LightColor0;
      // color of light source (from "Lighting.cginc")

   varying vec4 position;
      // position of the vertex (and fragment) in world space
   varying vec3 varyingNormalDirection;
      // surface normal vector in world space
   varying vec4 textureCoordinates;

   #ifdef VERTEX

   void main()
   {
      mat4 modelMatrix = _Object2World;
      mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
         // is unnecessary because we normalize vectors
```

```
   position = modelMatrix * gl_Vertex;
   varyingNormalDirection = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
   textureCoordinates = gl_MultiTexCoord0;
}

#endif

#ifdef FRAGMENT

void main()
{
   vec3 normalDirection = normalize(varyingNormalDirection);

   vec3 viewDirection =
      normalize(_WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection;
   float attenuation;

   vec4 textureColor =
      texture2D(_MainTex, vec2(textureCoordinates));

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
   }
   else // point or spot light
   {
      vec3 vertexToLightSource =
         vec3(_WorldSpaceLightPos0 - position);
      float distance = length(vertexToLightSource);
      attenuation = 1.0 / distance; // linear attenuation
      lightDirection = normalize(vertexToLightSource);
   }

   vec3 diffuseReflection = attenuation * vec3(_LightColor0)
      * vec3(_Color) * vec3(textureColor)
      * max(0.0, dot(normalDirection, lightDirection));

   vec3 specularReflection;
   if (dot(normalDirection, lightDirection) < 0.0)
      // light source on the wrong side?
   {
      specularReflection = vec3(0.0, 0.0, 0.0);
         // no specular reflection
   }
   else // light source on the right side
   {
      specularReflection = attenuation * vec3(_LightColor0)
         * vec3(_SpecColor) * (1.0 - textureColor.a)
            // for usual gloss maps: "... * textureColor.a"
         * pow(max(0.0, dot(
         reflect(-lightDirection, normalDirection),
         viewDirection)), _Shininess);
```

```
            }

            gl_FragColor =
               vec4(diffuseReflection + specularReflection, 1.0);
         }

         #endif

         ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Specular"
}
```

A useful modification of this shader for the particular texture image above, would be to set the diffuse material color to a dark blue where the alpha component is 0.

18.0.99. Shader Code for Per-Vertex Lighting

As discussed in GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[8], specular highlights are usually not rendered very well with per-vertex lighting. Sometimes, however, there is no choice because of performance limitations. In order to include gloss mapping in the shader code of GLSL PROGRAMMING/UNITY/LIGHTING TEXTURED SURFACES[9], the fragment shaders of both passes should be replaced with this code:

```
         #ifdef FRAGMENT

         void main()
         {
            vec4 textureColor =
               texture2D(_MainTex, vec2(textureCoordinates));
            gl_FragColor = vec4(diffuseColor * vec3(textureColor)
               + specularColor * (1.0 - textureColor.a), 1.0);
         }

         #endif
```

Note that a usual gloss map would require a multiplication with `textureColor.a` instead of `(1.0 - textureColor.a)`.

8 Chapter 13 on page 111
9 Chapter 17 on page 145

18.0.100. Summary

Congratulations! You finished an important tutorial about gloss mapping. We have looked at:

- What gloss mapping is.
- How to implement it for per-pixel lighting.
- How to implement it for per-vertex lighting.

18.0.101. Further Reading

If you still want to learn more

- about per-pixel lighting (without texturing), you should read GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[10].
- about texturing, you should read GLSL PROGRAMMING/UNITY/TEXTURED SPHERES[11].
- about per-vertex lighting with texturing, you should read GLSL PROGRAMMING/UNITY/LIGHTING TEXTURED SURFACES[12].

10 Chapter 13 on page 111
11 Chapter 16 on page 137
12 Chapter 17 on page 145

19. Transparent Textures

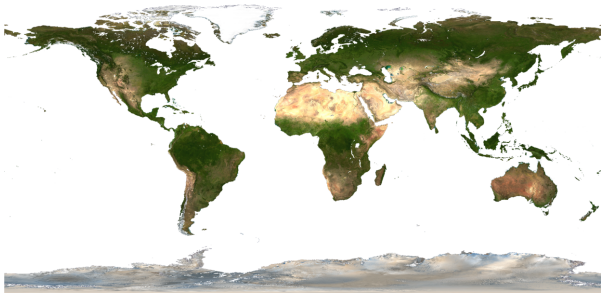

Figure 27 Map of the Earth with transparent water, i.e. the alpha component is 0 for water and 1 for land.

This tutorial covers various common uses of **alpha texture maps**, i.e. RGBA texture images with an A (alpha) component that specifies the opacity of texels.

It combines the shader code of GLSL PROGRAMMING/UNITY/TEXTURED SPHERES[1] with concepts that were introduced in GLSL PROGRAMMING/UNITY/CUTAWAYS[2] and GLSL PROGRAMMING/UNITY/TRANSPARENCY[3].

If you haven't read these tutorials, this would be a very good opportunity to read them.

19.0.102. Discarding Transparent Fragments

Let's start with discarding fragments as explained in GLSL PROGRAMMING/UNITY/CUTAWAYS[4]. Follow the steps described in GLSL PROGRAMMING/UNITY/-

1 Chapter 16 on page 137
2 Chapter 6 on page 43
3 Chapter 7 on page 49
4 Chapter 6 on page 43

TEXTURED SPHERES[5] and assign the image above to the material of a sphere with the following shader:

```
Shader "GLSL texturing with alpha discard" {
   Properties {
      _MainTex ("RGBA Texture Image", 2D) = "white" {}
      _Cutoff ("Alpha Cutoff", Float) = 0.5
   }
   SubShader {
      Pass {
         Cull Off // since the front is partially transparent,
            // we shouldn't cull the back

         GLSLPROGRAM

         uniform sampler2D _MainTex;
         uniform float _Cutoff;

         varying vec4 textureCoordinates;

         #ifdef VERTEX

         void main()
         {
            textureCoordinates = gl_MultiTexCoord0;
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif

         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor =
               texture2D(_MainTex, vec2(textureCoordinates));
            if (gl_FragColor.a < _Cutoff)
               // alpha value less than user-specified threshold?
            {
               discard; // yes: discard this fragment
            }
         }

         #endif

         ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Unlit/Transparent Cutout"
}
```

5 Chapter 16 on page 137

The fragment shader reads the RGBA texture and compares the alpha value against a user-specified threshold. If the alpha value is less than the threshold, the fragment is discarded and the surface appears transparent.

19.0.103. Alpha Testing

The same effect as described above can be implemented with an alpha test. The advantage of the alpha test is that it runs also on older hardware that doesn't support GLSL. Here is the code, which results in more or less the same result as the shader above:

```
Shader "GLSL texturing with alpha test" {
   Properties {
      _MainTex ("RGBA Texture Image", 2D) = "white" {}
      _Cutoff ("Alpha Cutoff", Float) = 0.5
   }
   SubShader {
      Pass {
         Cull Off // since the front is partially transparent,
            // we shouldn't cull the back
         AlphaTest Greater [_Cutoff] // specify alpha test:
            // fragment passes if alpha is greater than _Cutoff

         GLSLPROGRAM

         uniform sampler2D _MainTex;
         uniform float _Cutoff;

         varying vec4 textureCoordinates;

         #ifdef VERTEX

         void main()
         {
            textureCoordinates = gl_MultiTexCoord0;
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif

         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor =
               texture2D(_MainTex, vec2(textureCoordinates));
         }

         #endif

         ENDGLSL
      }
```

```
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Unlit/Transparent Cutout"
}
```

Here, no explicit `discard` instruction is necessary but the alpha test has to be configured to pass only those fragments with an alpha value of more than the `_Cutoff` property; otherwise they are discarded. More details about the alpha test are available in UNITY'S SHADERLAB DOCUMENTATION[6]. Note that the `discard` instruction is a lot more flexible and better supported by APIs such as OpenGL ES 2.0 than the alpha test.

19.0.104. Blending

The GLSL PROGRAMMING/UNITY/TRANSPARENCY[7] described how to render semitransparent objects with alpha blending. Combining this with an RGBA texture results in this code:

```
Shader "GLSL texturing with alpha blending" {
   Properties {
      _MainTex ("RGBA Texture Image", 2D) = "white" {}
   }
   SubShader {
      Tags {"Queue" = "Transparent"}

      Pass {
         Cull Front // first render the back faces
         ZWrite Off // don't write to depth buffer
            // in order not to occlude other objects
         Blend SrcAlpha OneMinusSrcAlpha
            // blend based on the fragment's alpha value

         GLSLPROGRAM

         uniform sampler2D _MainTex;

         varying vec4 textureCoordinates;

         #ifdef VERTEX

         void main()
         {
            textureCoordinates = gl_MultiTexCoord0;
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }
```

6 HTTP://UNITY3D.COM/SUPPORT/DOCUMENTATION/COMPONENTS/SL-ALPHATEST.HTML

7 Chapter 7 on page 49

```
         #endif

         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor =
               texture2D(_MainTex, vec2(textureCoordinates));
         }

         #endif

         ENDGLSL
      }

      Pass {
         Cull Back // now render the front faces
         ZWrite Off // don't write to depth buffer
            // in order not to occlude other objects
         Blend SrcAlpha OneMinusSrcAlpha
            // blend based on the fragment's alpha value

         GLSLPROGRAM

         uniform sampler2D _MainTex;

         varying vec4 textureCoordinates;

         #ifdef VERTEX

         void main()
         {
            textureCoordinates = gl_MultiTexCoord0;
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif

         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor =
               texture2D(_MainTex, vec2(textureCoordinates));
         }

         #endif

         ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Unlit/Transparent"
}
```

It should be mentioned that this particular texture image contains only alpha values of either 0 or 1. Thus, there are relatively few fragments that receive an alpha value in between 0 and 1 due to interpolation of the alpha values of neighboring texels. It is only for those fragments that the order of rendering is important. If one accepts potential rendering artifacts for these fragments, one might improve the performance of the shader by removing the subshader tag `"Queue" = "Transparent"` (thus, Unity doesn't have to sort the objects from back to front) and also removing the setting `ZWrite Off` (thus the depth test can work more efficiently).

Furthermore, all texels with an alpha value of 0 are black in this particular texture image. In fact, the colors in this texture image are "premultiplied" with their alpha value. (Such colors are also called "opacity-weighted.") Thus, for this particular image, we should actually specify the blend equation for premultiplied colors in order to avoid another multiplication of the colors with their alpha value in the blend equation. Therefore, an improvement of the shader (for this particular texture image) is to employ the following blend specification in both passes:

```
Blend One OneMinusSrcAlpha
```

Figure 28 Semitransparent globes are often used for logos and trailers.

19.0.105. Blending with Customized Colors

We should not end this tutorial without a somewhat more practical application of the presented techniques. Below is an image of a globe with semitransparent blue oceans, which I found on Wikimedia Commons. There is some lighting (or silhouette enhancement) going on, which I didn't try to reproduce. Instead, I only tried to reproduce the basic idea of semitransparent oceans with the following

shader, which ignores the RGB colors of the texture map and replaces them by specific colors based on the alpha value:

```
Shader "GLSL semitransparent colors based on alpha" {
   Properties {
      _MainTex ("RGBA Texture Image", 2D) = "white" {}
   }
   SubShader {
      Tags {"Queue" = "Transparent"}

      Pass {
         Cull Front // first render the back faces
         ZWrite Off // don't write to depth buffer
            // in order not to occlude other objects
         Blend SrcAlpha OneMinusSrcAlpha
            // blend based on the fragment's alpha value

         GLSLPROGRAM

         uniform sampler2D _MainTex;

         varying vec4 textureCoordinates;

         #ifdef VERTEX

         void main()
         {
            textureCoordinates = gl_MultiTexCoord0;
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif

         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor =
               texture2D(_MainTex, vec2(textureCoordinates));
            if (gl_FragColor.a > 0.5) // opaque back face?
            {
               gl_FragColor = vec4(0.0, 0.0, 0.2, 1.0);
                  // opaque dark blue
            }
            else // transparent back face?
            {
               gl_FragColor = vec4(0.0, 0.0, 1.0, 0.3);
                  // semitransparent dark blue
            }
         }

         #endif

         ENDGLSL
      }

      Pass {
```

```
         Cull Back // now render the front faces
         ZWrite Off // don't write to depth buffer
            // in order not to occlude other objects
         Blend SrcAlpha OneMinusSrcAlpha
            // blend based on the fragment's alpha value

         GLSLPROGRAM

         uniform sampler2D _MainTex;

         varying vec4 textureCoordinates;

         #ifdef VERTEX

         void main()
         {
            textureCoordinates = gl_MultiTexCoord0;
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif

         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor =
               texture2D(_MainTex, vec2(textureCoordinates));
            if (gl_FragColor.a > 0.5) // opaque front face?
            {
               gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0);
                  // opaque green
            }
            else // transparent front face
            {
               gl_FragColor = vec4(0.0, 0.0, 1.0, 0.3);
                  // semitransparent dark blue
            }
         }

         #endif

         ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Unlit/Transparent"
}
```

Of course, it would be interesting to add lighting and silhouette enhancement to this shader. One could also change the opaque, green color in order to take the texture color into account, e.g. with:

```
gl_FragColor = vec4(0.5 * gl_FragColor.r, 2.0 * gl_-
FragColor.g, 0.5 * gl_FragColor.b, 1.0);
```

which emphasizes the green component by multiplying it with 2 and dims the red and blue components by multiplying them with 0.5. However, this results in oversaturated green that is clamped to the maximum intensity. This can be avoided by halvening the difference of the green component to the maximum intensity 1. This difference is `1.0 - gl_FragColor.g`; half of it is `0.5 * (1.0 - gl_FragColor.g)` and the value corresponding to this reduced distance to the maximum intensity is: `1.0 - 0.5 * (1.0 - gl_FragColor.g)`. Thus, in order to avoid oversaturation of green, we could use (instead of the opaque green color):

```
gl_FragColor = vec4(0.5 * gl_FragColor.r, 1.0 - 0.5 *
(1.0 - gl_FragColor.g), 0.5 * gl_FragColor.b, 1.0);
```

In practice, one has to try various possibilites for such color transformations. To this end, the use of numeric shader properties (e.g. for the factors 0.5 in the line above) is particularly useful to interactively explore the possibilities.

19.0.106. Summary

Congratulations! You have reached the end of this rather long tutorial. We have looked at:

- How discarding fragments can be combined with alpha texture maps.
- How the alpha test can be used to achieve the same effect.
- How alpha texture maps can be used for blending.
- How alpha texture maps can be used to determine colors.

19.0.107. Further Reading

If you still want to know more

- about texturing, you should read GLSL PROGRAMMING/UNITY/TEXTURED SPHERES[8].
- about discarding fragments, you should read GLSL PROGRAMMING/UNITY/-CUTAWAYS[9].
- about the alpha test, you should read UNITY'S SHADERLAB DOCUMENTATION: ALPHA TESTING[10].

8 Chapter 16 on page 137
9 Chapter 6 on page 43
10 HTTP://UNITY3D.COM/SUPPORT/DOCUMENTATION/COMPONENTS/SL-ALPHATEST.HTML

- about blending, you should read GLSL PROGRAMMING/UNITY/TRANSPARENCY[11].

11 Chapter 7 on page 49

20. Layers of Textures

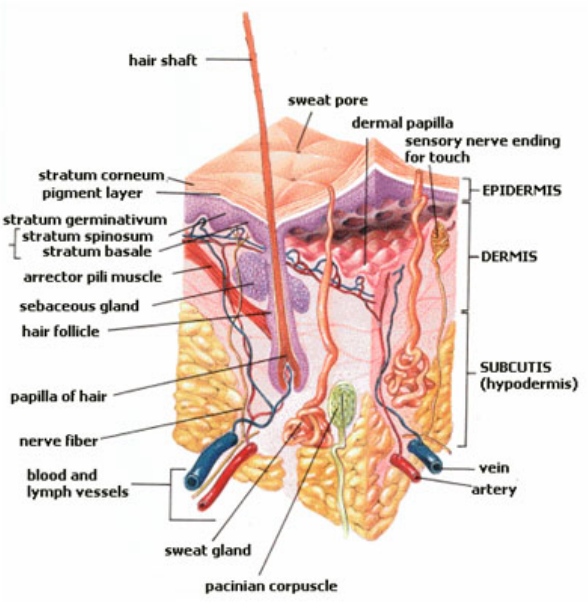

Figure 29 Layers of human skin.

This tutorial introduces **multitexturing**, i.e. the use of multiple texture images in a shader.

It extends the shader code of GLSL PROGRAMMING/UNITY/TEXTURED SPHERES[1] to multiple textures and shows a way of combining them. If you haven't read that tutorials, this would be a very good opportunity to read it.

1 Chapter 16 on page 137

20.0.108. Layers of Surfaces

Many real surfaces (e.g. the human skin illustrated in the image above) consist of several layers of different colors, transparencies, reflectivities, etc. If the topmost layer is opaque and doesn't transmit any light, this doesn't really matter for rendering the surface. However, in many cases the topmost layer is (semi)transparent and therefore an accurate rendering of the surface has to take multiple layers into account.

In fact, the specular reflection that is included in the Phong reflection model (see GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[2]) often corresponds to a transparent layer that reflects light: sweat on human skin, wax on fruits, transparent plastics with embedded pigment particles, etc. On the other hand, the diffuse reflection corresponds to the layer(s) below the topmost transparent layer.

Lighting such layered surfaces doesn't require a geometric model of the layers: they can be represented by a single, infinitely thin polygon mesh. However, the lighting computation has to compute different reflections for different layers and has to take the transmission of light between layers into account (both when light enters the layer and when it exits the layer). Examples of this approach are included in the “Dawn” demo by Nvidia (see Chapter 3 of the book “GPU Gems”, which is available ONLINE[3]) and the “Human Head” demo by Nvidia (see Chapter 14 of the book “GPU Gems 3”, which is also available ONLINE[4]).

A full description of these processes is beyond the scope of this tutorial. Suffice to say that layers are often associated with texture images to specify their characteristics. Here we just show how to use two textures and one particular way of combining them. The example is in fact not related to layers and therefore illustrates that multitexturing has more applications than layers of surfaces.

2 Chapter 11 on page 91

3 HTTP://DEVELOPER.NVIDIA.COM/NODE/108

4 HTTP://DEVELOPER.NVIDIA.COM/NODE/171

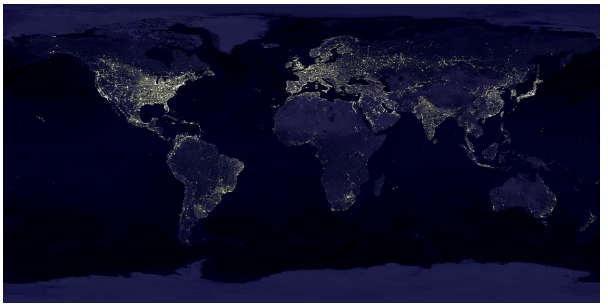

Figure 30 Map of the unlit Earth.

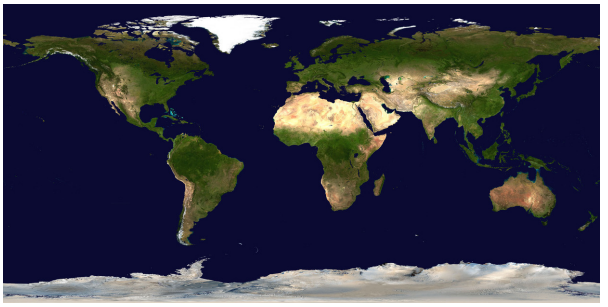

Figure 31 Map of the sunlit Earth.

20.0.109. Lit and Unlit Earth

Due to human activities, the unlit side of the Earth is not completely dark. Instead, artificial lights mark the position and extension of cities as shown in the image above . Therefore, diffuse lighting of the Earth should not just dim the texture image for the sunlit surface but actually blend it to the unlit texture image. Note that the sunlit Earth is far brighter than human-made lights on the unlit side; however, we reduce this contrast in order to show off the nighttime texture.

The shader code extends the code from GLSL PROGRAMMING/UNITY/TEXTURED SPHERES[5] to two texture images and uses the computation described in GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[6] for a single, directional light source:

$$I_{\text{diffuse}} = I_{\text{incoming}} k_{\text{diffuse}} \max(0, \mathbf{N} \cdot \mathbf{L})$$

5 Chapter 16 on page 137
6 Chapter 10 on page 77

According to this equation, the level of diffuse lighting `levelOfLighting` is max(0, **N**·**L**). We then blend the colors of the daytime texture and the nighttime texture based on `levelOfLighting`. This could be achieved by multiplying the daytime color with `levelOfLighting` and multiplying the nighttime color with `1.0 - levelOfLighting` before adding them to determine the fragment's color. Alternatively, the built-in GLSL function `mix` can be used (`mix(a, b, w) = b*w + a*(1.0-w)`), which is likely to be more efficient. Thus, the fragment shader could be:

```
#ifdef FRAGMENT

void main()
{
   nighttimeColor =
      texture2D(_MainTex, vec2(textureCoordinates));
   daytimeColor =
      texture2D(_DecalTex, vec2(textureCoordinates));
   gl_FragColor =
      mix(nighttimeColor, daytimeColor, levelOfLighting);
      // = daytimeColor * levelOfLighting
      // + nighttimeColor * (1.0 - levelOfLighting)
}

#endif
```

Note that this blending is very similar to the alpha blending that was discussed in GLSL PROGRAMMING/UNITY/TRANSPARENCY[7] except that we perform the blending inside a fragment shader and use `levelOfLighting` instead of the alpha component (i.e. the opacity) of the texture that should be blended "over" the other texture. In fact, if `_DecalTex` specified an alpha component (see GLSL PROGRAMMING/UNITY/TRANSPARENT TEXTURES[8]), we could use this alpha component to blend `_DecalTex` over `_MainTex`. This is actually what Unity's standard `Decal` shader does and it corresponds to a layer which is partially transparent on top of an opaque layer that is visible where the topmost layer is transparent.

20.0.110. Complete Shader Code

The names of the properties of the shader were chosen to agree with the property names of the fallback shader — in this case the `Decal` shader (note that the fallback `Decal` shade and the standard `Decal` shader appear to use the two textures in

7 Chapter 7 on page 49
8 Chapter 19 on page 161

opposite ways). Also, an additional property `_Color` is introduced and multiplied (component-wise) to the texture color of the nighttime texture in order to control its overall brightness. Furthermore, the color of the light source `_LightColor0` is multiplied (also component-wise) to the color of the daytime texture in order to take colored light sources into account.

```
Shader "GLSL multitexturing of Earth" {
   Properties {
      _DecalTex ("Daytime Earth", 2D) = "white" {}
      _MainTex ("Nighttime Earth", 2D) = "white" {}
      _Color ("Nighttime Color Filter", Color) = (1,1,1,1)
   }
   SubShader {
      Pass {
         Tags { "LightMode" = "ForwardBase" }
            // pass for the first, directional light

         GLSLPROGRAM

         uniform sampler2D _MainTex;
         uniform sampler2D _DecalTex;
         uniform vec4 _Color;

         // The following built-in uniforms (except _LightColor0)
         // are also defined in "UnityCG.glslinc",
         // i.e. one could #include "UnityCG.glslinc"
         uniform mat4 _Object2World; // model matrix
         uniform mat4 _World2Object; // inverse model matrix
         uniform vec4 _WorldSpaceLightPos0;
            // direction to or position of light source
         uniform vec4 _LightColor0;
            // color of light source (from "Lighting.cginc")

         varying float levelOfLighting;
            // level of diffuse lighting computed in vertex shader
         varying vec4 textureCoordinates;

         #ifdef VERTEX

         void main()
         {
            mat4 modelMatrix = _Object2World;
            mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
               // is unnecessary because we normalize vectors

            vec3 normalDirection = normalize(vec3(
               vec4(gl_Normal, 0.0) * modelMatrixInverse));
            vec3 lightDirection;

            if (0.0 == _WorldSpaceLightPos0.w) // directional light?
            {
               lightDirection = normalize(vec3(_WorldSpaceLightPos0));
            }
            else // point or spot light
            {
```

```
            lightDirection = vec3(0.0, 0.0, 0.0);
               // ignore other light sources
         }

         levelOfLighting =
            max(0.0, dot(normalDirection, lightDirection));
         textureCoordinates = gl_MultiTexCoord0;
         gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
      }

      #endif

      #ifdef FRAGMENT

      void main()
      {
         vec4 nighttimeColor = _Color
            * texture2D(_MainTex, vec2(textureCoordinates));
         vec4 daytimeColor = _LightColor0
            * texture2D(_DecalTex, vec2(textureCoordinates));
         gl_FragColor =
            mix(nighttimeColor, daytimeColor, levelOfLighting);
            // = daytimeColor * levelOfLighting
            // + nighttimeColor * (1.0 - levelOfLighting)
      }

      #endif

      ENDGLSL
   }
}
// The definition of a fallback shader should be commented out
// during development:
// Fallback "Decal"
}
```

When you run this shader, make sure that you have an activated directional light source in your scene.

20.0.111. Summary

Congratulations! You have reached the end of the last tutorial on basic texturing. We have looked at:

- How layers of surfaces can influence the appearance of materials (e.g. human skin, waxed fruits, plastics, etc.)
- How artificial lights on the unlit side can be taken into account when texturing a sphere representing the Earth.
- How to implement this technique in a shader.
- How this is related to blending an alpha texture over a second opaque texture.

20.0.112. Further Reading

If you still want to know more

- about basic texturing, you should read GLSL PROGRAMMING/UNITY/TEXTURED SPHERES[9].
- about diffuse reflection, you should read GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[10].
- about alpha textures, you should read GLSL PROGRAMMING/UNITY/TRANSPARENT TEXTURES[11].
- about advanced skin rendering, you could read Chapter 3 “Skin in the ‘Dawn’ Demo” by Curtis Beeson and Kevin Bjorke of the book “GPU Gems” by Randima Fernando (editor) published 2004 by Addison-Wesley, which is available ONLINE[12], and Chapter 14 “Advanced Techniques for Realistic Real-Time Skin Rendering” by Eugene d’Eon and David Luebke of the book “GPU Gems 3” by Hubert Nguyen (editor) published 2007 by Addison-Wesley, which is also available ONLINE[13].

9 Chapter 16 on page 137
10 Chapter 10 on page 77
11 Chapter 19 on page 161
12 HTTP://HTTP.DEVELOPER.NVIDIA.COM/GPUGEMS/GPUGEMS_CH03.HTML
13 HTTP://HTTP.DEVELOPER.NVIDIA.COM/GPUGEMS3/GPUGEMS3_CH14.HTML

Part V.

Textures in 3D

21. Lighting of Bumpy Surfaces

Figure 32 "The Incredulity of Saint Thomas" by Caravaggio, 1601-1603.

This tutorial covers **normal mapping**.

It's the first in a series of tutorials about texturing techniques that go beyond two-dimensional surfaces (or layers of surfaces). In this tutorial, we start with normal mapping, which is a very well established technique to fake the lighting of small bumps and dents — even on coarse polygon meshes. The code of this tutorial is based on GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[1] and GLSL PROGRAMMING/UNITY/TEXTURED SPHERES[2].

1 Chapter 13 on page 111
2 Chapter 16 on page 137

21.0.113. Perceiving Shapes Based on Lighting

The painting by Caravaggio depicted below is about the incredulity of Saint Thomas, who did not believe in Christ's resurrection until he put his finger in Christ's side. The furrowed brows of the apostles not only symbolize this incredulity but clearly convey it by means of a common facial expression. However, why do we know that their foreheads are actually furrowed instead of being painted with some light and dark lines? After all, this is just a flat painting. In fact, viewers intuitively make the assumption that these are furrowed instead of painted brows — even though the painting itself allows for both interpretations. The lesson is: bumps on smooth surfaces can often be convincingly conveyed by the lighting alone without any other cues (shadows, occlusions, parallax effects, stereo, etc.).

21.0.114. Normal Mapping

Normal mapping tries to convey bumps on smooth surfaces (i.e. coarse triangle meshes with interpolated normals) by changing the surface normal vectors according to some virtual bumps. When the lighting is computed with these modified normal vectors, viewers will often perceive the virtual bumps — even though a perfectly flat triangle has been rendered. The illusion can certainly break down (in particular at silhouettes) but in many cases it is very convincing.

More specifically, the normal vectors that represent the virtual bumps are first **encoded** in a texture image (i.e. a normal map). A fragment shader then looks up these vectors in the texture image and computes the lighting based on them. That's about it. The problem, of course, is the encoding of the normal vectors in a texture image. There are different possibilities and the fragment shader has to be adapted to the specific encoding that was used to generate the normal map.

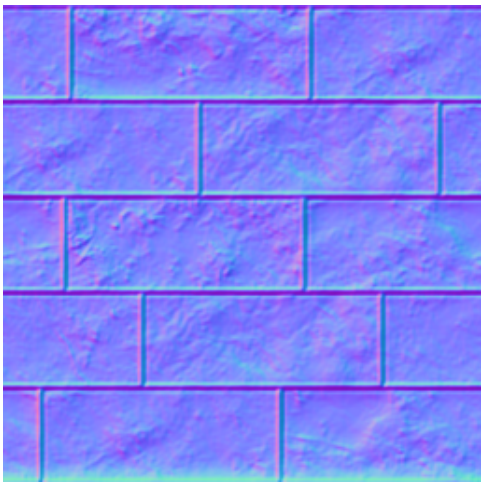

Figure 33 A typical example for the appearance of an encoded normal map.

21.0.115. Normal Mapping in Unity

The very good news is that you can easily create normal maps from gray-scale images with Unity: create a gray-scale image in your favorite paint program and use a specific gray for the regular height of the surface, lighter grays for bumps, and darker grays for dents. Make sure that the transitions between different grays are smooth, e.g. by blurring the image. When you import the image with **Assets > Import New Asset** change the **Texture Type** in the **Inspector View** to **Normal map** and check **Generate from greyscale**. After clicking **Apply**, the preview should show a bluish image with reddish and greenish edges. Alternatively to generating a normal map, the encoded normal map above can be imported (don't forget to uncheck the **Generate from greyscale** box).

The not so good news is that the fragment shader has to do some computations to decode the normals. First of all, the texture color is stored in a two-component texture image, i.e. there is only an alpha component A and one color component available. The color component can be accessed as the red, green, or blue component — in all cases the same value is returned. Here, we use the green component G since Unity also uses it. The two components, G and A, are stored as numbers between 0 and 1; however, they represent coordinates n_x and n_y between -1 and 1. The mapping is:

$$n_x = 2A - 1 \quad \text{and} \quad n_y = 2G - 1$$

From these two components, the third component n_z of the three-dimensional normal vector $\mathbf{n}= (n_x, n_y, n_z)$ can be calculated because of the normalization to unit length:

$$\sqrt{n_x^2 + n_y^2 + n_z^2} = 1 \quad \Rightarrow \quad n_z = \pm\sqrt{1 - n_x^2 - n_y^2}$$

Only the "+" solution is necessary if we choose the z axis along the axis of the smooth normal vector (interpolated from the normal vectors that were set in the vertex shader) since we aren't able to render surfaces with an inwards pointing normal vector anyways. The code snippet from the fragment shader could look like this:

```
vec4 encodedNormal = texture2D(_BumpMap,
   _BumpMap_ST.xy * textureCoordinates.xy
   + _BumpMap_ST.zw);
vec3 localCoords =
    vec3(2.0 * encodedNormal.ag - vec2(1.0), 0.0);
localCoords.z = sqrt(1.0 - dot(localCoords, localCoords));
   // approximation without sqrt:  localCoords.z =
   // 1.0 - 0.5 * dot(localCoords, localCoords);
```

The decoding for devices that use OpenGL ES is actually simpler since Unity doesn't use a two-component texture in this case. Thus, for mobile platforms the decoding becomes:

```
vec4 encodedNormal = texture2D(_BumpMap,
   _BumpMap_ST.xy * textureCoordinates.xy
   + _BumpMap_ST.zw);
vec3 localCoords = 2.0 * encodedNormal.rgb - vec3(1.0);
```

However, the rest of this tutorial (and also GLSL PROGRAMMING/UNITY/PROJECTION OF BUMPY SURFACES[3]) will cover only (desktop) OpenGL.

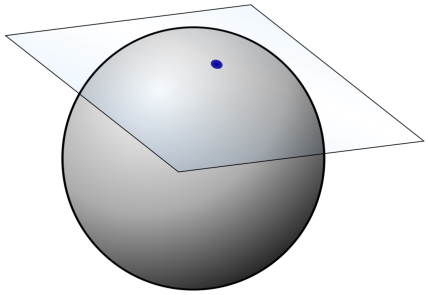

Figure 34 Tangent plane to a point on a sphere.

3 Chapter 22 on page 193

Unity uses a local surface coordinate systems for each point of the surface to specify normal vectors in the normal map. The z axis of this local coordinates system is given by the smooth, interpolated normal vector **N** in world space and the $x-y$ plane is a tangent plane to the surface as illustrated in the image below . Specifically, the x axis is specified by the tangent attribute **T** that Unity provides to vertices (see the discussion of attributes in GLSL PROGRAMMING/UNITY/DEBUGGING OF SHADERS[4]). Given the x and z axis, the y axis can be computed by a cross product in the vertex shader, e.g. $\mathbf{B} = \mathbf{N} \times \mathbf{T}$. (The letter **B** refers to the traditional name "binormal" for this vector.)

Note that the normal vector **N** is transformed with the transpose of the inverse model matrix from object space to world space (because it is orthogonal to a surface; see GLSL PROGRAMMING/APPLYING MATRIX TRANSFORMATIONS[5]) while the tangent vector **T** specifies a direction between points on a surface and is therefore transformed with the model matrix. The binormal vector **B** represents a third class of vectors which are transformed differently. (If you really want to know: the skew-symmetric matrix B corresponding to "$\mathbf{B}\times$" is transformed like a quadratic form.) Thus, the best choice is to first transform **N** and **T** to world space, and then to compute **B** in world space using the cross product of the transformed vectors.

With the normalized directions **T**, **B**, and **N** in world space, we can easily form a matrix that maps any normal vector **n** of the normal map from the local surface coordinate system to world space because the columns of such a matrix are just the vectors of the axes; thus, the 3×3 matrix for the mapping of **n** to world space is:

$$\mathrm{M}_{\mathrm{surface}\to\mathrm{world}} = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}$$

These calculations are performed by the vertex shader, for example this way:

```
varying vec4 position;
   // position of the vertex (and fragment) in world space
varying vec4 textureCoordinates;
varying mat3 localSurface2World; // mapping from
   // local surface coordinates to world coordinates

#ifdef VERTEX

attribute vec4 Tangent;

void main()
{
```

4 Chapter 4 on page 23
5 Chapter 45 on page 429

```
            mat4 modelMatrix = _Object2World;
            mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
               // is unnecessary because we normalize vectors

            localSurface2World[0] = normalize(vec3(
               modelMatrix * vec4(vec3(Tangent), 0.0)));
            localSurface2World[2] = normalize(vec3(
               vec4(gl_Normal, 0.0) * modelMatrixInverse));
            localSurface2World[1] = normalize(
               cross(localSurface2World[2], localSurface2World[0])
               * Tangent.w); // factor Tangent.w is specific to Unity

            position = modelMatrix * gl_Vertex;
            textureCoordinates = gl_MultiTexCoord0;
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif
```

The factor `Tangent.w` in the computation of `binormal` is specific to Unity, i.e. Unity provides tangent vectors and normal maps such that we have to do this multiplication.

In the fragment shader, we multiply the matrix in `localSurface2World` with **n**. For example, with this line:

```
            vec3 normalDirection =
               normalize(localSurface2World * localCoords);
```

With the new normal vector in world space, we can compute the lighting as in GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[6].

21.0.116. Complete Shader Code

This shader code simply integrates all the snippets and uses our standard two-pass approach for pixel lights.

```
Shader "GLSL normal mapping" {
   Properties {
      _BumpMap ("Normal Map", 2D) = "bump" {}
      _Color ("Diffuse Material Color", Color) = (1,1,1,1)
      _SpecColor ("Specular Material Color", Color) = (1,1,1,1)
      _Shininess ("Shininess", Float) = 10
   }
   SubShader {
      Pass {
         Tags { "LightMode" = "ForwardBase" }
            // pass for ambient light and first light source
```

6 Chapter 13 on page 111

```
GLSLPROGRAM

// User-specified properties
uniform sampler2D _BumpMap;
uniform vec4 _BumpMap_ST;
uniform vec4 _Color;
uniform vec4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (except _LightColor0)
// are also defined in "UnityCG.glslinc",
// i.e. one could #include "UnityCG.glslinc"
uniform vec3 _WorldSpaceCameraPos;
   // camera position in world space
uniform mat4 _Object2World; // model matrix
uniform mat4 _World2Object; // inverse model matrix
uniform vec4 _WorldSpaceLightPos0;
   // direction to or position of light source
uniform vec4 _LightColor0;
   // color of light source (from "Lighting.cginc")

varying vec4 position;
   // position of the vertex (and fragment) in world space
varying vec4 textureCoordinates;
varying mat3 localSurface2World; // mapping from local
   // surface coordinates to world coordinates

#ifdef VERTEX

attribute vec4 Tangent;

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   localSurface2World[0] = normalize(vec3(
      modelMatrix * vec4(vec3(Tangent), 0.0)));
   localSurface2World[2] = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));
   localSurface2World[1] = normalize(
      cross(localSurface2World[2], localSurface2World[0])
      * Tangent.w); // factor Tangent.w is specific to Unity

   position = modelMatrix * gl_Vertex;
   textureCoordinates = gl_MultiTexCoord0;
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
   // in principle we have to normalize the columns of
```

```
// "localSurface2World" again; however, the potential
// problems are small since we use this matrix only to
// compute "normalDirection", which we normalize anyways

vec4 encodedNormal = texture2D(_BumpMap,
   _BumpMap_ST.xy * textureCoordinates.xy
   + _BumpMap_ST.zw);
vec3 localCoords =
   vec3(2.0 * encodedNormal.ag - vec2(1.0), 0.0);
localCoords.z = sqrt(1.0 - dot(localCoords, localCoords));
   // approximation without sqrt: localCoords.z =
   // 1.0 - 0.5 * dot(localCoords, localCoords);
vec3 normalDirection =
   normalize(localSurface2World * localCoords);

vec3 viewDirection =
   normalize(_WorldSpaceCameraPos - vec3(position));
vec3 lightDirection;
float attenuation;

if (0.0 == _WorldSpaceLightPos0.w) // directional light?
{
   attenuation = 1.0; // no attenuation
   lightDirection = normalize(vec3(_WorldSpaceLightPos0));
}
else // point or spot light
{
   vec3 vertexToLightSource =
      vec3(_WorldSpaceLightPos0 - position);
   float distance = length(vertexToLightSource);
   attenuation = 1.0 / distance; // linear attenuation
   lightDirection = normalize(vertexToLightSource);
}

vec3 ambientLighting =
   vec3(gl_LightModel.ambient) * vec3(_Color);

vec3 diffuseReflection =
   attenuation * vec3(_LightColor0) * vec3(_Color)
   * max(0.0, dot(normalDirection, lightDirection));

vec3 specularReflection;
if (dot(normalDirection, lightDirection) < 0.0)
   // light source on the wrong side?
{
   specularReflection = vec3(0.0, 0.0, 0.0);
      // no specular reflection
}
else // light source on the right side
{
   specularReflection = attenuation * vec3(_LightColor0)
      * vec3(_SpecColor) * pow(max(0.0, dot(
      reflect(-lightDirection, normalDirection),
      viewDirection)), _Shininess);
}

gl_FragColor = vec4(ambientLighting
   + diffuseReflection + specularReflection, 1.0);
```

```
   }

   #endif

   ENDGLSL
}

Pass {
   Tags { "LightMode" = "ForwardAdd" }
      // pass for additional light sources
   Blend One One // additive blending

  GLSLPROGRAM

   // User-specified properties
   uniform sampler2D _BumpMap;
   uniform vec4 _BumpMap_ST;
   uniform vec4 _Color;
   uniform vec4 _SpecColor;
   uniform float _Shininess;

   // The following built-in uniforms (except _LightColor0)
   // are also defined in "UnityCG.glslinc",
   // i.e. one could #include "UnityCG.glslinc"
   uniform vec3 _WorldSpaceCameraPos;
      // camera position in world space
   uniform mat4 _Object2World; // model matrix
   uniform mat4 _World2Object; // inverse model matrix
   uniform vec4 _WorldSpaceLightPos0;
      // direction to or position of light source
   uniform vec4 _LightColor0;
      // color of light source (from "Lighting.cginc")

   varying vec4 position;
      // position of the vertex (and fragment) in world space
   varying vec4 textureCoordinates;
   varying mat3 localSurface2World; // mapping from
      // local surface coordinates to world coordinates

   #ifdef VERTEX

   attribute vec4 Tangent;

   void main()
   {
      mat4 modelMatrix = _Object2World;
      mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
         // is unnecessary because we normalize vectors

      localSurface2World[0] = normalize(vec3(
         modelMatrix * vec4(vec3(Tangent), 0.0)));
      localSurface2World[2] = normalize(vec3(
         vec4(gl_Normal, 0.0) * modelMatrixInverse));
      localSurface2World[1] = normalize(
         cross(localSurface2World[2], localSurface2World[0])
         * Tangent.w); // factor Tangent.w is specific to Unity

      position = modelMatrix * gl_Vertex;
```

```
   textureCoordinates = gl_MultiTexCoord0;
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
   // in principle we have to normalize the columns of
   // "localSurface2World" again; however, the potential
   // problems are small since we use this matrix only to
   // compute "normalDirection", which we normalize anyways

   vec4 encodedNormal = texture2D(_BumpMap,
      _BumpMap_ST.xy * textureCoordinates.xy
      + _BumpMap_ST.zw);
   vec3 localCoords =
      vec3(2.0 * encodedNormal.ag - vec2(1.0), 0.0);
   localCoords.z = sqrt(1.0 - dot(localCoords, localCoords));
      // approximation without sqrt: localCoords.z =
      // 1.0 - 0.5 * dot(localCoords, localCoords);
   vec3 normalDirection =
      normalize(localSurface2World * localCoords);

   vec3 viewDirection =
      normalize(_WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
   }
   else // point or spot light
   {
      vec3 vertexToLightSource =
         vec3(_WorldSpaceLightPos0 - position);
      float distance = length(vertexToLightSource);
      attenuation = 1.0 / distance; // linear attenuation
      lightDirection = normalize(vertexToLightSource);
   }

   vec3 diffuseReflection =
      attenuation * vec3(_LightColor0) * vec3(_Color)
      * max(0.0, dot(normalDirection, lightDirection));

   vec3 specularReflection;
   if (dot(normalDirection, lightDirection) < 0.0)
      // light source on the wrong side?
   {
      specularReflection = vec3(0.0, 0.0, 0.0);
         // no specular reflection
   }
   else // light source on the right side
   {
```

```
               specularReflection = attenuation * vec3(_LightColor0)
                  * vec3(_SpecColor) * pow(max(0.0, dot(
                  reflect(-lightDirection, normalDirection),
                  viewDirection)), _Shininess);
            }

            gl_FragColor =
               vec4(diffuseReflection + specularReflection, 1.0);
         }

         #endif

         ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Bumped Specular"
}
```

Note that we have used the tiling and offset uniform `_BumpMap_ST` as explained in the GLSL PROGRAMMING/UNITY/TEXTURED SPHERES[7] since this option is often particularly useful for bump maps.

21.0.117. Summary

Congratulations! You finished this tutorial! We have look at:

- How human perception of shapes often relies on lighting.
- What normal mapping is.
- How Unity encodes normal maps.
- How a fragment shader can decode Unity's normal maps and use them to per-pixel lighting.

21.0.118. Further Reading

If you still want to know more

- about texture mapping (including tiling and offseting), you should read GLSL PROGRAMMING/UNITY/TEXTURED SPHERES[8].
- about per-pixel lighting with the Phong reflection model, you should read GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[9].

7 Chapter 16 on page 137
8 Chapter 16 on page 137
9 Chapter 13 on page 111

- about transforming normal vectors, you should read GLSL PROGRAMMING/APPLYING MATRIX TRANSFORMATIONS[10].
- about normal mapping, you could read Mark J. Kilgard: "A Practical and Robust Bump-mapping Technique for Today's GPUs", GDC 2000: Advanced OpenGL Game Development, which is available ONLINE[11].

10 Chapter 45 on page 429

11 HTTP://CITESEER.IST.PSU.EDU/VIEWDOC/SUMMARY?DOI=10.1.1.18.537&RANK=2

22. Projection of Bumpy Surfaces

Figure 35 A dry-stone wall in England. Note how some stones stick out of the wall.

This tutorial covers (single-step) **parallax mapping**.

It extends and is based on GLSL PROGRAMMING/UNITY/LIGHTING OF BUMPY SURFACES[1].

22.0.119. Improving Normal Mapping

The normal mapping technique presented in GLSL PROGRAMMING/UNITY/-LIGHTING OF BUMPY SURFACES[2] only changes the lighting of a flat surface to create the illusion of bumps and dents. If one looks straight onto a surface (i.e. in the direction of the surface normal vector), this works very well. However,

1 Chapter 21 on page 181
2 Chapter 21 on page 181

if one looks onto a surface from some other angle (as in the image below), the bumps should also stick out of the surface while the dents should recede into the surface. Of course, this could be achieved by geometrically modeling bumps and dents; however, this would require to process many more vertices. On the other hand, single-step parallax mapping is a very efficient techniques similar to normal mapping, which doesn't require additional triangles but can still move virtual bumps by several pixels to make them stick out of a flat surface. However, the technique is limited to bumps and dents of small heights and requires some fine-tuning for best results.

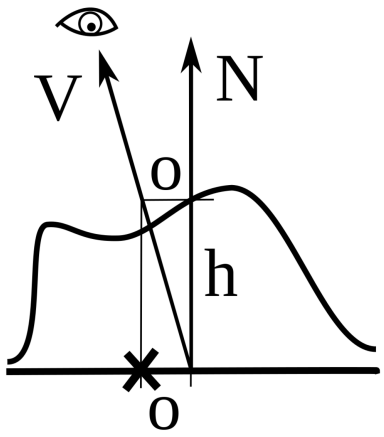

Figure 36 Vectors and distances in parallax mapping: view vector V, surface normal vector N, height of height map h, offset o to intersection of view ray with surface at height h.

22.0.120. Parallax Mapping Explained

Parallax mapping was proposed in 2001 by Tomomichi Kaneko et al. in their paper "Detailed shape representation with parallax mapping" (ICAT 2001). The basic idea is to offset the texture coordinates that are used for the texturing of the surface (in particular normal mapping). If this offset of texture coordinates is computed appropriately, it is possible to move parts of the texture (e.g. bumps) as if they were sticking out of the surface.

The illustration below shows the view vector **V** in the direction to the viewer and the surface normal vector **N** in the point of a surface that is rasterized in a fragment shader. Parallax mapping proceeds in 3 steps:

- Lookup of the height h at the rasterized point in a height map, which is depicted by the wavy line on top of the straight line at the bottom in the illustration.
- Computation of the intersection of the viewing ray in direction of **V** with a surface at height h parallel to the rendered surface. The distance o is the distance between the rasterized surface point moved by h in the direction of **N** and this intersection point. If these two points are projected onto the rendered surface, o is also the distance between the rasterized point and a new point on the surface (marked by a cross in the illustration). This new surface point is a better approximation to the point that is actually visible for the view ray in direction **V** if the surface was displaced by the height map.
- Transformation of the offset o into texture coordinate space in order to compute an offset of texture coordinates for all following texture lookups.

For the computation of o we require the height h of a height map at the rasterized point, which is implemented in the example by a texture lookup in the A component of the texture property `_ParallaxMap`, which should be a gray-scale image representing heights as discussed in GLSL PROGRAMMING/UNITY/LIGHTING OF BUMPY SURFACES[3]. We also require the view direction **V** in the local surface coordinate system formed by the normal vector (z axis), the tangent vector (x axis), and the binormal vector (y axis), which was also introduced GLSL PROGRAMMING/UNITY/LIGHTING OF BUMPY SURFACES[4]. To this end we compute a transformation from local surface coordinates to object space with:

$$\mathrm{M}_{\text{surface}\rightarrow\text{object}} = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}$$

where **T**, **B** and **N** are given in object coordinates. (In GLSL PROGRAMMING/UNITY/LIGHTING OF BUMPY SURFACES[5] we had a similar matrix but with vectors in world coordinates.)

We compute the view direction **V** in object space (as the difference between the rasterized position and the camera position transformed from world space to object space) and then we transform it to the local surface space with the matrix $\mathrm{M}_{\text{object}\rightarrow\text{surface}}$ which can be computed as:

3 Chapter 21 on page 181
4 Chapter 21 on page 181
5 Chapter 21 on page 181

$$\mathrm{M}_{\text{object}\to\text{surface}} = \mathrm{M}^{-1}_{\text{surface}\to\text{object}} = \mathrm{M}^{T}_{\text{surface}\to\text{object}}$$

This is possible because **T**, **B** and **N** are orthogonal and normalized. (Actually, the situation is a bit more complicated because we won't normalize these vectors but use their length for another transformation; see below.) Thus, in order to transform **V** from object space to the local surface space, we have to multiply it with the transposed matrix $(\mathrm{M}_{\text{surface}\to\text{object}})^T$. In GLSL, this is achieved by multiplying the vector from the left to the matrix $\mathrm{M}_{\text{surface}\to\text{object}}$.

Once we have the **V** in the local surface coordinate system with the z axis in the direction of the normal vector **N**, we can compute the offsets o_x (in x direction) and o_y (in y direction) by using similar triangles (compare with the illustration):

$\frac{o_x}{h} = \frac{V_x}{V_z}$ and $\frac{o_y}{h} = \frac{V_y}{V_z}$.

Thus:

$o_x = h\frac{V_x}{V_z}$ and $o_y = h\frac{V_y}{V_z}$.

Note that it is not necessary to normalize **V** because we use only ratios of its components, which are not affected by the normalization.

Finally, we have to transform o_x and o_y into texture space. This would be quite difficult if Unity wouldn't help us: the tangent attribute `Tangent` is actually appropriately scaled and has a fourth component `Tangent.w` for scaling the binormal vector such that the transformation of the view direction **V** scales V_x and V_y appropriately to have o_x and o_y in texture coordinate space without further computations.

22.0.121. Implementation

The implementation shares most of the code with GLSL PROGRAMMING/UNITY/-LIGHTING OF BUMPY SURFACES[6]. In particular, the same scaling of the binormal vector with the fourth component of the `Tangent` attribute is used in order to take the mapping of the offsets from local surface space to texture space into account:

```
vec3 binormal = cross(gl_Normal, vec3(Tangent)) * Tangent.w;
```

In the vertex shader, we have to add a varying for the view vector **V** in the local surface coordinate system (with the scaling of axes to take the mapping to texture space into account). This varying is called `viewDirInScaledSurfaceCoords`. It is computed by multiplying the view

6 Chapter 21 on page 181

vector in object coordinates (`viewDirInObjectCoords`) from the left to the matrix $M_{surface \to object}$ (`localSurface2ScaledObject`) as explained above:

```
vec3 viewDirInObjectCoords = vec3(
   modelMatrixInverse * vec4(_WorldSpaceCameraPos, 1.0)
   - gl_Vertex);
mat3 localSurface2ScaledObject =
   mat3(vec3(Tangent), binormal, gl_Normal);
   // vectors are orthogonal
viewDirInScaledSurfaceCoords =
   viewDirInObjectCoords * localSurface2ScaledObject;
   // we multiply with the transpose to multiply with
   // the "inverse" (apart from the scaling)
```

The rest of the vertex shader is the same as for normal mapping, see GLSL PROGRAMMING/UNITY/LIGHTING OF BUMPY SURFACES[7].

In the fragment shader, we first query the height map for the height of the rasterized point. This height is specified by the A component of the texture `_ParallaxMap`. The values between 0 and 1 are transformed to the range -`_Parallax`/2 to +`_Parallax` with a shader property `_Parallax` in order to offer some user control over the strength of the effect (and to be compatible with the fallback shader):

```
float height = _Parallax
    * (-0.5 + texture2D(_ParallaxMap, _ParallaxMap_ST.xy
    * textureCoordinates.xy + _ParallaxMap_ST.zw));
```

The offsets o_x and o_y are then computed as described above. However, we also clamp each offset between a user-specified interval -`_MaxTexCoordOffset` and `_MaxTexCoordOffset` in order to make sure that the offset stays in reasonable bounds. (If the height map consists of more or less flat plateaus of constant height with smooth transitions between these plateaus, `_MaxTexCoordOffset` should be smaller than the thickness of these transition regions; otherwise the sample point might be in a different plateau with a different height, which would mean that the approximation of the intersection point is arbitrarily bad.) The code is:

```
vec2 texCoordOffsets =
   clamp(height * viewDirInScaledSurfaceCoords.xy
   / viewDirInScaledSurfaceCoords.z,
   -_MaxTexCoordOffset, +_MaxTexCoordOffset);
```

In the following code, we have to apply the offsets to the texture coordinates in all texture lookups; i.e., we have to replace `vec2(textureCoordinates)` (or equivalentyl `textureCoordinates.xy`) by `(textureCoordinates.xy + texCoordOffsets)`, e.g.:

7 Chapter 21 on page 181

```
         vec2 texCoordOffsets =
            clamp(height * viewDirInScaledSurfaceCoords.xy
            / viewDirInScaledSurfaceCoords.z,
            -_MaxTexCoordOffset, +_MaxTexCoordOffset);
```

The rest of the fragment shader code is just as it was for GLSL PROGRAMMING/UNITY/LIGHTING OF BUMPY SURFACES[8].

22.0.122. Complete Shader Code

As discussed in the previous section, most of this code is taken from GLSL PROGRAMMING/UNITY/LIGHTING OF BUMPY SURFACES[9]. Note that if you want to use the code on a mobile device with OpenGL ES, make sure to change the decoding of the normal map as described in that tutorial.

The part about parallax mapping is actually only a few lines. Most of the names of the shader properties were chosen according to the fallback shader; the user interface labels are much more descriptive.

```
Shader "GLSL parallax mapping" {
   Properties {
      _BumpMap ("Normal Map", 2D) = "bump" {}
      _ParallaxMap ("Heightmap (in A)", 2D) = "black" {}
      _Parallax ("Max Height", Float) = 0.01
      _MaxTexCoordOffset ("Max Texture Coordinate Offset", Float) =
         0.01
      _Color ("Diffuse Material Color", Color) = (1,1,1,1)
      _SpecColor ("Specular Material Color", Color) = (1,1,1,1)
      _Shininess ("Shininess", Float) = 10
   }
   SubShader {
      Pass {
         Tags { "LightMode" = "ForwardBase" }
            // pass for ambient light and first light source

         GLSLPROGRAM

         // User-specified properties
         uniform sampler2D _BumpMap;
         uniform vec4 _BumpMap_ST;
         uniform sampler2D _ParallaxMap;
         uniform vec4 _ParallaxMap_ST;
         uniform float _Parallax;
         uniform float _MaxTexCoordOffset;
         uniform vec4 _Color;
         uniform vec4 _SpecColor;
```

8 Chapter 21 on page 181
9 Chapter 21 on page 181

```
uniform float _Shininess;

// The following built-in uniforms (except _LightColor0)
// are also defined in "UnityCG.glslinc",
// i.e. one could #include "UnityCG.glslinc"
uniform vec3 _WorldSpaceCameraPos;
   // camera position in world space
uniform mat4 _Object2World; // model matrix
uniform mat4 _World2Object; // inverse model matrix
uniform vec4 unity_Scale; // w = 1/uniform scale;
   // should be multiplied to _World2Object
uniform vec4 _WorldSpaceLightPos0;
   // direction to or position of light source
uniform vec4 _LightColor0;
   // color of light source (from "Lighting.cginc")

varying vec4 position;
   // position of the vertex (and fragment) in world space
varying vec4 textureCoordinates;
varying mat3 localSurface2World; // mapping from
   // local surface coordinates to world coordinates
varying vec3 viewDirInScaledSurfaceCoords;

#ifdef VERTEX

attribute vec4 Tangent;

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object * unity_Scale.w;

   localSurface2World[0] = normalize(vec3(
      modelMatrix * vec4(vec3(Tangent), 0.0)));
   localSurface2World[2] = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));
   localSurface2World[1] = normalize(
      cross(localSurface2World[2], localSurface2World[0])
      * Tangent.w);

   vec3 binormal =
      cross(gl_Normal, vec3(Tangent)) * Tangent.w;
      // appropriately scaled tangent and binormal
      // to map distances from object space to texture space

   vec3 viewDirInObjectCoords = vec3(modelMatrixInverse
      * vec4(_WorldSpaceCameraPos, 1.0) - gl_Vertex);
   mat3 localSurface2ScaledObject =
      mat3(vec3(Tangent), binormal, gl_Normal);
      // vectors are orthogonal
   viewDirInScaledSurfaceCoords =
      viewDirInObjectCoords * localSurface2ScaledObject;
      // we multiply with the transpose to multiply
      // with the "inverse" (apart from the scaling)

   position = modelMatrix * gl_Vertex;
   textureCoordinates = gl_MultiTexCoord0;
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```

```
}

#endif

#ifdef FRAGMENT

void main()
{
   // parallax mapping: compute height and
   // find offset in texture coordinates
   // for the intersection of the view ray
   // with the surface at this height

   float height =
      _Parallax * (-0.5 + texture2D(_ParallaxMap,
      _ParallaxMap_ST.xy * textureCoordinates.xy
      + _ParallaxMap_ST.zw));
   vec2 texCoordOffsets =
      clamp(height * viewDirInScaledSurfaceCoords.xy
      / viewDirInScaledSurfaceCoords.z,
      -_MaxTexCoordOffset, +_MaxTexCoordOffset);

   // normal mapping: lookup and decode normal from bump map

   // in principle we have to normalize the columns
   // of "localSurface2World" again; however, the potential
   // problems are small since we use this matrix only
   // to compute "normalDirection", which we normalize anyways
   vec4 encodedNormal = texture2D(_BumpMap,
      _BumpMap_ST.xy * (textureCoordinates.xy
      + texCoordOffsets) + _BumpMap_ST.zw);
   vec3 localCoords =
      vec3(2.0 * encodedNormal.ag - vec2(1.0), 0.0);
   localCoords.z = sqrt(1.0 - dot(localCoords, localCoords));
      // approximation without sqrt: localCoords.z =
      // 1.0 - 0.5 * dot(localCoords, localCoords);
   vec3 normalDirection =
      normalize(localSurface2World * localCoords);

   // per-pixel lighting using the Phong reflection model
   // (with linear attenuation for point and spot lights)

   vec3 viewDirection =
      normalize(_WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
   }
   else // point or spot light
   {
      vec3 vertexToLightSource =
         vec3(_WorldSpaceLightPos0 - position);
      float distance = length(vertexToLightSource);
      attenuation = 1.0 / distance; // linear attenuation
```

```
            lightDirection = normalize(vertexToLightSource);
         }

         vec3 ambientLighting =
            vec3(gl_LightModel.ambient) * vec3(_Color);

         vec3 diffuseReflection =
            attenuation * vec3(_LightColor0) * vec3(_Color)
            * max(0.0, dot(normalDirection, lightDirection));

         vec3 specularReflection;
         if (dot(normalDirection, lightDirection) < 0.0)
            // light source on the wrong side?
         {
            specularReflection = vec3(0.0, 0.0, 0.0);
               // no specular reflection
         }
         else // light source on the right side
         {
            specularReflection = attenuation * vec3(_LightColor0)
               * vec3(_SpecColor) * pow(max(0.0, dot(
               reflect(-lightDirection, normalDirection),
               viewDirection)), _Shininess);
         }

         gl_FragColor = vec4(ambientLighting + diffuseReflection
            + specularReflection, 1.0);
      }

      #endif

      ENDGLSL
   }

   Pass {
      Tags { "LightMode" = "ForwardAdd" }
         // pass for additional light sources
      Blend One One // additive blending

      GLSLPROGRAM

      // User-specified properties
      uniform sampler2D _BumpMap;
      uniform vec4 _BumpMap_ST;
      uniform sampler2D _ParallaxMap;
      uniform vec4 _ParallaxMap_ST;
      uniform float _Parallax;
      uniform float _MaxTexCoordOffset;
      uniform vec4 _Color;
      uniform vec4 _SpecColor;
      uniform float _Shininess;

      // The following built-in uniforms (except _LightColor0)
      // are also defined in "UnityCG.glslinc",
      // i.e. one could #include "UnityCG.glslinc"
      uniform vec3 _WorldSpaceCameraPos;
         // camera position in world space
      uniform mat4 _Object2World; // model matrix
```

```
uniform mat4 _World2Object; // inverse model matrix
uniform vec4 unity_Scale; // w = 1/uniform scale;
   // should be multiplied to _World2Object
uniform vec4 _WorldSpaceLightPos0;
   // direction to or position of light source
uniform vec4 _LightColor0;
   // color of light source (from "Lighting.cginc")

varying vec4 position;
   // position of the vertex (and fragment) in world space
varying vec4 textureCoordinates;
varying mat3 localSurface2World; // mapping
   // from local surface coordinates to world coordinates
varying vec3 viewDirInScaledSurfaceCoords;

#ifdef VERTEX

attribute vec4 Tangent;

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object * unity_Scale.w;

   localSurface2World[0] = normalize(vec3(
      modelMatrix * vec4(vec3(Tangent), 0.0)));
   localSurface2World[2] = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));
   localSurface2World[1] = normalize(
      cross(localSurface2World[2], localSurface2World[0])
      * Tangent.w);

   vec3 binormal =
      cross(gl_Normal, vec3(Tangent)) * Tangent.w;
      // appropriately scaled tangent and binormal
      // to map distances from object space to texture space

   vec3 viewDirInObjectCoords = vec3(modelMatrixInverse
      * vec4(_WorldSpaceCameraPos, 1.0) - gl_Vertex);
   mat3 localSurface2ScaledObject =
      mat3(vec3(Tangent), binormal, gl_Normal);
      // vectors are orthogonal
   viewDirInScaledSurfaceCoords =
      viewDirInObjectCoords * localSurface2ScaledObject;
      // we multiply with the transpose to multiply
      // with the "inverse" (apart from the scaling)

   position = modelMatrix * gl_Vertex;
   textureCoordinates = gl_MultiTexCoord0;
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
```

```
// parallax mapping: compute height and
// find offset in texture coordinates
// for the intersection of the view ray
// with the surface at this height

float height =
   _Parallax * (-0.5 + texture2D(_ParallaxMap,
   _ParallaxMap_ST.xy * textureCoordinates.xy
   + _ParallaxMap_ST.zw));
vec2 texCoordOffsets =
   clamp(height * viewDirInScaledSurfaceCoords.xy
   / viewDirInScaledSurfaceCoords.z,
   -_MaxTexCoordOffset, +_MaxTexCoordOffset);

// normal mapping: lookup and decode normal from bump map

// in principle we have to normalize the columns
// of "localSurface2World" again; however, the potential
// problems are small since we use this matrix only to
// compute "normalDirection", which we normalize anyways
vec4 encodedNormal = texture2D(_BumpMap,
   _BumpMap_ST.xy * (textureCoordinates.xy
   + texCoordOffsets) + _BumpMap_ST.zw);
vec3 localCoords =
   vec3(2.0 * encodedNormal.ag - vec2(1.0), 0.0);
localCoords.z = sqrt(1.0 - dot(localCoords, localCoords));
   // approximation without sqrt: localCoords.z =
   // 1.0 - 0.5 * dot(localCoords, localCoords);
vec3 normalDirection =
   normalize(localSurface2World * localCoords);

// per-pixel lighting using the Phong reflection model
// (with linear attenuation for point and spot lights)

vec3 viewDirection =
   normalize(_WorldSpaceCameraPos - vec3(position));
vec3 lightDirection;
float attenuation;

if (0.0 == _WorldSpaceLightPos0.w) // directional light?
{
   attenuation = 1.0; // no attenuation
   lightDirection = normalize(vec3(_WorldSpaceLightPos0));
}
else // point or spot light
{
   vec3 vertexToLightSource =
      vec3(_WorldSpaceLightPos0 - position);
   float distance = length(vertexToLightSource);
   attenuation = 1.0 / distance; // linear attenuation
   lightDirection = normalize(vertexToLightSource);
}

vec3 diffuseReflection =
   attenuation * vec3(_LightColor0) * vec3(_Color)
   * max(0.0, dot(normalDirection, lightDirection));

vec3 specularReflection;
```

```
            if (dot(normalDirection, lightDirection) < 0.0)
               // light source on the wrong side?
            {
               specularReflection = vec3(0.0, 0.0, 0.0);
                  // no specular reflection
            }
            else // light source on the right side
            {
               specularReflection = attenuation * vec3(_LightColor0)
                  * vec3(_SpecColor) * pow(max(0.0, dot(
                  reflect(-lightDirection, normalDirection),
                  viewDirection)), _Shininess);
            }

            gl_FragColor =
               vec4(diffuseReflection + specularReflection, 1.0);
         }

         #endif

         ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Parallax Specular"
}
```

22.0.123. Summary

Congratulations! If you actually understand the whole shader, you have come a long way. In fact, the shader includes lots of concepts (transformations between coordinate systems, application of the inverse of an orthogonal matrix by multiplying a vector from the left to it, the Phong reflection model, normal mapping, parallax mapping, ...). More specifically, we have seen:

- How parallax mapping improves upon normal mapping.
- How parallax mapping is described mathematically.
- How parallax mapping is implemented.

22.0.124. Further Reading

If you still want to know more

- about details of the shader code, you should read GLSL PROGRAMMING/UNITY/LIGHTING OF BUMPY SURFACES[10].

10 Chapter 21 on page 181

- about parallax mapping, you could read the original publication by Tomomichi Kaneko et al.: “Detailed shape representation with parallax mapping”, ICAT 2001, pages 205–208, which is available ONLINE[11].

11 HTTP://CITESEER.IST.PSU.EDU/VIEWDOC/SUMMARY?DOI=10.1.1.115.1050

23. Cookies

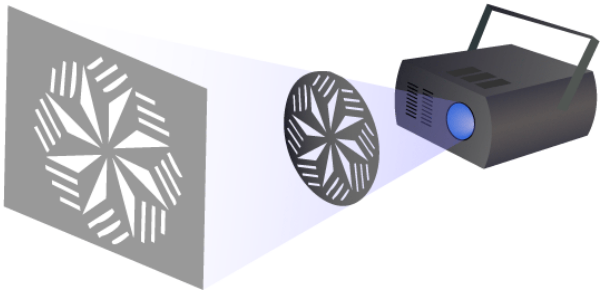

Figure 37 Illustration of a gobo in front of a light source.

Figure 38 A cookie in action: similar to a gobo but not as close to the light source.

This tutorial covers **projective texture mapping in light space**, which is useful to implement cookies for spotlights and directional light sources. (In fact, Unity uses a built-in cookie for any spotlight.)

The tutorial is based on the code of GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[1] and GLSL PROGRAMMING/UNITY/TRANSPARENT TEXTURES[2]. If you haven't read those tutorials yet, you should read them first.

1 Chapter 13 on page 111
2 Chapter 19 on page 161

23.0.125. Gobos and Cookies in Real Life

In real life, gobos are pieces of material (often metal) with holes that are placed in front of light sources to manipulate the shape of light beams or shadows. Cookies (or "cuculoris") serve a similar purpose but are placed at a larger distance from the light source as shown in the image below .

23.0.126. Unity's Cookies

In Unity, a **cookie** can be specified for each light source in the **Inspector View** when the light source is selected. This cookie is basically an alpha texture map (see GLSL PROGRAMMING/UNITY/TRANSPARENT TEXTURES[3]) that is placed in front of the light source and moves with it (therefore it is actually similar to a gobo). It lets light pass through where the alpha component of the texture image is 1 and blocks light where the alpha component is 0. Unity's cookies for spotlights and directional lights are just square, two-dimensional alpha texture maps. On the other hand, cookies for point lights are cube maps, which we will not cover here.

In order to implement a cookie, we have to extend the shader of any surface that should be affected by the cookie. (This is very different from how Unity's projectors work; see GLSL PROGRAMMING/UNITY/PROJECTORS[4].) Specifically, we have to attenuate the light of each light source according to its cookie in the lighting computation of a shader. Here, we use the per-pixel lighting described in GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[5]; however, the technique can be applied to any lighting computation.

In order to find the relevant position in the cookie texture, the position of the rasterized point of a surface is transformed into the coordinate system of the light source. This coordinate system is very similar to the clip coordinate system of a camera, which is described in GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[6]. In fact, the best way to think of the coordinate system of a light source is probably to think of the light source as a camera. The x and y light coordinates are then related to the screen coordinates of this hypothetical camera. Transforming a point from world coordinates to light coordinates is actually very easy because Unity provides the required 4×4 matrix as the uniform variable `_LightMatrix0`. (Otherwise

3 Chapter 19 on page 161
4 Chapter 25 on page 233
5 Chapter 13 on page 111
6 Chapter 43 on page 407

we would have to set up the matrix similar to the matrices for the viewing transformation and the projection, which are discussed in GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[7].)

For best efficiency, the transformation of the surface point from world space to light space should be performed by multiplying `_LightMatrix0` to the position in world space in the vertex shader, for example this way:

```
uniform mat4 _Object2World; // model matrix
uniform mat4 _World2Object; // inverse model matrix
uniform vec4 _WorldSpaceLightPos0;
   // direction to or position of light source
uniform vec4 _LightColor0;
   // color of light source (from Lighting.cginc)
uniform mat4 _LightMatrix0; // transformation
   // from world to light space (from Autolight.cginc)

varying vec4 position;
   // position of the vertex (and fragment) in world space
varying vec4 positionInLightSpace;
   // position of the vertex (and fragment) in light space
varying vec3 varyingNormalDirection;
   // surface normal vector in world space

#ifdef VERTEX

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   position = modelMatrix * gl_Vertex;
   positionInLightSpace = _LightMatrix0 * position;
   varyingNormalDirection = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif
```

Apart from the definitions of the uniform `_LightMatrix0` and the varying `positionInLightSpace` and the instruction to compute `positionInLightSpace`, this is the same vertex shader as in GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[8].

7 Chapter 43 on page 407
8 Chapter 13 on page 111

23.0.127. Cookies for Directional Light Sources

For the cookie of a directional light source, we can just use the *x* and *y* light coordinates in `positionInLightSpace` as texture coordinates for a lookup in the cookie texture `_LightTexture0`. This texture lookup should be performed in the fragment shader. Then the resulting alpha component should be multiplied to the computed lighting; for example:

```
// compute diffuseReflection and specularReflection

float cookieAttenuation = 1.0;
if (0.0 == _WorldSpaceLightPos0.w) // directional light?
{
   cookieAttenuation = texture2D(_LightTexture0,
      vec2(positionInLightSpace)).a;
}
// compute cookieAttenuation for spotlights here

gl_FragColor = vec4(cookieAttenuation
   * (diffuseReflection + specularReflection), 1.0);
```

Instead of `vec2(positionInLightSpace)` we could also use `positionInLightSpace.xy` to get a two-dimensional vector with the *x* and *y* coordinates in light space.

23.0.128. Cookies for Spotlights

For spotlights, the *x* and *y* light coordinates in `positionInLightSpace` have to be divided by the *w* light coordinate. This division is characteristic for projective texture mapping and corresponds to the perspective division for a camera, which is described in GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[9]. Unity defines the matrix `_LightMatrix0` such that we have to add 0.5 to both coordinates after the division:

```
cookieAttenuation = texture2D(_LightTexture0,
   vec2(positionInLightSpace) / positionInLightSpace.w
   + vec2(0.5)).a;
```

For some GPUs it might be more efficient to use the built-in function `texture2DProj`, which takes three texture coordinates in a `vec3` and divides the first two coordinates by the third coordinate before the texture lookup. A problem with this approach is that we have to add 0.5 **after** the division by `positionInLightSpace.w`; however, `texture2DProj` doesn't allow us

9 Chapter 43 on page 407

to add anything after the internal division by the third texture coordinate. The solution is to add `0.5 * positionInLightSpace.w` **before** the division by `positionInLightSpace.w`, which corresponds to adding 0.5 after the division:

```
vec3 textureCoords = vec3(vec2(positionInLightSpace)
   + vec2(0.5 * positionInLightSpace.w),
   positionInLightSpace.w);
cookieAttenuation =
   texture2DProj(_LightTexture0, textureCoords).a;
```

Note that the texture lookup for directional lights can also be implemented with `texture2DProj` by setting `textureCoords` to `vec3(vec2(positionInLightSpace), 1.0)`. This would allow us to use only one texture lookup for both directional lights and for spotlights, which is more efficient on some GPUs.

23.0.129. Complete Shader Code

For the complete shader code we use a simplified version of the `ForwardBase` pass of GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[10] since Unity only uses a directional light without cookie in the `ForwardBase` pass. All light sources with cookies are handled by the `ForwardAdd` pass. We ignore cookies for point lights, for which `_LightMatrix0[3][3]` is `1.0` (but we include them in the next section). Spotlights always have a cookie texture: if the user didn't specify one, Unity supplies a cookie texture to generate the shape of a spotlight; thus, it is OK to always apply the cookie. Directional lights don't always have a cookie; however, if there is only one directional light source without cookie then it has been processed in the `ForwardBase` pass. Thus, unless there are more than one directional light sources without cookies, we can assume that all directional light sources in the `ForwardAdd` pass have cookies. In this case, the complete shader code could be:

```
Shader "GLSL per-pixel lighting with cookies" {
   Properties {
      _Color ("Diffuse Material Color", Color) = (1,1,1,1)
      _SpecColor ("Specular Material Color", Color) = (1,1,1,1)
      _Shininess ("Shininess", Float) = 10
   }
   SubShader {
      Pass {
         Tags { "LightMode" = "ForwardBase" } // pass for ambient light
```

10 Chapter 13 on page 111

```
   // and first directional light source without cookie

GLSLPROGRAM

// User-specified properties
uniform vec4 _Color;
uniform vec4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (except _LightColor0)
// are also defined in "UnityCG.glslinc",
// i.e. one could #include "UnityCG.glslinc"
uniform vec3 _WorldSpaceCameraPos;
   // camera position in world space
uniform mat4 _Object2World; // model matrix
uniform mat4 _World2Object; // inverse model matrix
uniform vec4 _WorldSpaceLightPos0;
   // direction to or position of light source
uniform vec4 _LightColor0;
   // color of light source (from Lighting.cginc)

varying vec4 position;
   // position of the vertex (and fragment) in world space
varying vec3 varyingNormalDirection;
   // surface normal vector in world space

#ifdef VERTEX

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   position = modelMatrix * gl_Vertex;
   varyingNormalDirection = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
   vec3 normalDirection = normalize(varyingNormalDirection);

   vec3 viewDirection =
      normalize(_WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection =
      normalize(vec3(_WorldSpaceLightPos0));

   vec3 ambientLighting =
      vec3(gl_LightModel.ambient) * vec3(_Color);

   vec3 diffuseReflection = vec3(_LightColor0) * vec3(_Color)
```

```
            * max(0.0, dot(normalDirection, lightDirection));

         vec3 specularReflection;
         if (dot(normalDirection, lightDirection) < 0.0)
            // light source on the wrong side?
         {
            specularReflection = vec3(0.0, 0.0, 0.0);
               // no specular reflection
         }
         else // light source on the right side
         {
            specularReflection = vec3(_LightColor0)
               * vec3(_SpecColor) * pow(max(0.0, dot(
               reflect(-lightDirection, normalDirection),
               viewDirection)), _Shininess);
         }

         gl_FragColor = vec4(ambientLighting
            + diffuseReflection + specularReflection, 1.0);
      }

      #endif

      ENDGLSL
   }

   Pass {
      Tags { "LightMode" = "ForwardAdd" }
         // pass for additional light sources
      Blend One One // additive blending

      GLSLPROGRAM

      // User-specified properties
      uniform vec4 _Color;
      uniform vec4 _SpecColor;
      uniform float _Shininess;

      // The following built-in uniforms (except _LightColor0)
      // are also defined in "UnityCG.glslinc",
      // i.e. one could #include "UnityCG.glslinc"
      uniform vec3 _WorldSpaceCameraPos;
         // camera position in world space
      uniform mat4 _Object2World; // model matrix
      uniform mat4 _World2Object; // inverse model matrix
      uniform vec4 _WorldSpaceLightPos0;
         // direction to or position of light source
      uniform vec4 _LightColor0;
         // color of light source (from Lighting.cginc)

      uniform mat4 _LightMatrix0; // transformation
         // from world to light space (from Autolight.cginc)
      uniform sampler2D _LightTexture0;
         // cookie alpha texture map (from Autolight.cginc)

      varying vec4 position;
         // position of the vertex (and fragment) in world space
      varying vec4 positionInLightSpace;
```

```
   // position of the vertex (and fragment) in light space
varying vec3 varyingNormalDirection;
   // surface normal vector in world space

#ifdef VERTEX

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   position = modelMatrix * gl_Vertex;
   positionInLightSpace = _LightMatrix0 * position;
   varyingNormalDirection = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
   vec3 normalDirection = normalize(varyingNormalDirection);

   vec3 viewDirection =
      normalize(_WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
   }
   else // point or spot light
   {
      vec3 vertexToLightSource =
         vec3(_WorldSpaceLightPos0 - position);
      float distance = length(vertexToLightSource);
      attenuation = 1.0 / distance; // linear attenuation
      lightDirection = normalize(vertexToLightSource);
   }

   vec3 diffuseReflection =
      attenuation * vec3(_LightColor0) * vec3(_Color)
      * max(0.0, dot(normalDirection, lightDirection));

   vec3 specularReflection;
   if (dot(normalDirection, lightDirection) < 0.0)
      // light source on the wrong side?
   {
      specularReflection = vec3(0.0, 0.0, 0.0);
         // no specular reflection
   }
```

```
            else // light source on the right side
            {
               specularReflection = attenuation * vec3(_LightColor0)
                  * vec3(_SpecColor) * pow(max(0.0, dot(
                  reflect(-lightDirection, normalDirection),
                  viewDirection)), _Shininess);
            }

            float cookieAttenuation = 1.0;
            if (0.0 == _WorldSpaceLightPos0.w) // directional light?
            {
               cookieAttenuation = texture2D(_LightTexture0,
                  vec2(positionInLightSpace)).a;
            }
            else if (1.0 != _LightMatrix0[3][3])
               // spotlight (i.e. not a point light)?
            {
               cookieAttenuation = texture2D(_LightTexture0,
                  vec2(positionInLightSpace) / positionInLightSpace.w
                  + vec2(0.5)).a;
            }
            gl_FragColor = vec4(cookieAttenuation
               * (diffuseReflection + specularReflection), 1.0);
         }

         #endif

         ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Specular"
}
```

23.0.130. Shader Programs for Specific Light Sources

The previous shader code is limited to scenes with at most one directional light source without a cookie. Also, it doesn't take cookies of point light sources into account. Writing more general shader code requires different `ForwardAdd` passes for different light sources. (Remember that the light source in the `ForwardBase` pass is always a directional light source without cookie.) Fortunately, Unity offers a way to generate multiple shaders by using the following Unity-specific directive (right after `GLSLPROGRAM` in the `ForwardAdd` pass):

```
#pragma multi_compile_lightpass
```

With this instruction, Unity will compile the shader code for the `ForwardAdd` pass multiple times for different kinds of light sources. Each compilation is distinguished by the definition of one of the following symbols: `DIRECTIONAL`, `DIRECTIONAL_COOKIE`, `POINT`, `POINT_NOATT`, `POINT_COOKIE`, `SPOT`.

The shader code should check which symbol is defined (using the directives `#if defined ... #elif defined ... #endif`) and include appropriate instructions. For example:

```
Shader "GLSL per-pixel lighting with cookies" {
   Properties {
      _Color ("Diffuse Material Color", Color) = (1,1,1,1)
      _SpecColor ("Specular Material Color", Color) = (1,1,1,1)
      _Shininess ("Shininess", Float) = 10
   }
   SubShader {
      Pass {
         Tags { "LightMode" = "ForwardBase" } // pass for ambient light
            // and first directional light source without cookie

         GLSLPROGRAM

         // User-specified properties
         uniform vec4 _Color;
         uniform vec4 _SpecColor;
         uniform float _Shininess;

         // The following built-in uniforms (except _LightColor0)
         // are also defined in "UnityCG.glslinc",
         // i.e. one could #include "UnityCG.glslinc"
         uniform vec3 _WorldSpaceCameraPos;
            // camera position in world space
         uniform mat4 _Object2World; // model matrix
         uniform mat4 _World2Object; // inverse model matrix
         uniform vec4 _WorldSpaceLightPos0;
            // direction to or position of light source
         uniform vec4 _LightColor0;
            // color of light source (from Lighting.cginc)

         varying vec4 position;
            // position of the vertex (and fragment) in world space
         varying vec3 varyingNormalDirection;
            // surface normal vector in world space

         #ifdef VERTEX

         void main()
         {
            mat4 modelMatrix = _Object2World;
            mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
               // is unnecessary because we normalize vectors

            position = modelMatrix * gl_Vertex;
            varyingNormalDirection = normalize(vec3(
               vec4(gl_Normal, 0.0) * modelMatrixInverse));

            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif
```

```
      #ifdef FRAGMENT

      void main()
      {
         vec3 normalDirection = normalize(varyingNormalDirection);

         vec3 viewDirection =
            normalize(_WorldSpaceCameraPos - vec3(position));
         vec3 lightDirection =
            normalize(vec3(_WorldSpaceLightPos0));

         vec3 ambientLighting =
            vec3(gl_LightModel.ambient) * vec3(_Color);

         vec3 diffuseReflection = vec3(_LightColor0) * vec3(_Color)
            * max(0.0, dot(normalDirection, lightDirection));

         vec3 specularReflection;
         if (dot(normalDirection, lightDirection) < 0.0)
            // light source on the wrong side?
         {
            specularReflection = vec3(0.0, 0.0, 0.0);
               // no specular reflection
         }
         else // light source on the right side
         {
            specularReflection = vec3(_LightColor0)
               * vec3(_SpecColor) * pow(max(0.0, dot(
               reflect(-lightDirection, normalDirection),
               viewDirection)), _Shininess);
         }

         gl_FragColor = vec4(ambientLighting
            + diffuseReflection + specularReflection, 1.0);
      }

      #endif

      ENDGLSL
   }

   Pass {
      Tags { "LightMode" = "ForwardAdd" }
         // pass for additional light sources
      Blend One One // additive blending

      GLSLPROGRAM

      #pragma multi_compile_lightpass

      // User-specified properties
      uniform vec4 _Color;
      uniform vec4 _SpecColor;
      uniform float _Shininess;

      // The following built-in uniforms (except _LightColor0)
      // are also defined in "UnityCG.glslinc",
      // i.e. one could #include "UnityCG.glslinc"
```

```
uniform vec3 _WorldSpaceCameraPos;
   // camera position in world space
uniform mat4 _Object2World; // model matrix
uniform mat4 _World2Object; // inverse model matrix
uniform vec4 _WorldSpaceLightPos0;
   // direction to or position of light source
uniform vec4 _LightColor0;
   // color of light source (from Lighting.cginc)

uniform mat4 _LightMatrix0; // transformation
   // from world to light space (from Autolight.cginc)
#if defined DIRECTIONAL_COOKIE || defined SPOT
   uniform sampler2D _LightTexture0;
      // cookie alpha texture map (from Autolight.cginc)
#elif defined POINT_COOKIE
   uniform samplerCube _LightTexture0;
      // cookie alpha texture map (from Autolight.cginc)
#endif

varying vec4 position;
   // position of the vertex (and fragment) in world space
varying vec4 positionInLightSpace;
   // position of the vertex (and fragment) in light space
varying vec3 varyingNormalDirection;
   // surface normal vector in world space

#ifdef VERTEX

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   position = modelMatrix * gl_Vertex;
   positionInLightSpace = _LightMatrix0 * position;
   varyingNormalDirection = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
   vec3 normalDirection = normalize(varyingNormalDirection);

   vec3 viewDirection =
      normalize(_WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection;
   float attenuation = 1.0;
      // by default no attenuation with distance

   #if defined DIRECTIONAL || defined DIRECTIONAL_COOKIE
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
```

```
            #elif defined POINT_NOATT
               lightDirection =
                  normalize(vec3(_WorldSpaceLightPos0 - position));
            #elif defined POINT || defined POINT_COOKIE || defined SPOT
               vec3 vertexToLightSource =
                  vec3(_WorldSpaceLightPos0 - position);
               float distance = length(vertexToLightSource);
               attenuation = 1.0 / distance; // linear attenuation
               lightDirection = normalize(vertexToLightSource);
            #endif

            vec3 diffuseReflection =
               attenuation * vec3(_LightColor0) * vec3(_Color)
               * max(0.0, dot(normalDirection, lightDirection));

            vec3 specularReflection;
            if (dot(normalDirection, lightDirection) < 0.0)
               // light source on the wrong side?
            {
               specularReflection = vec3(0.0, 0.0, 0.0);
                  // no specular reflection
            }
            else // light source on the right side
            {
               specularReflection = attenuation * vec3(_LightColor0)
                  * vec3(_SpecColor) * pow(max(0.0, dot(
                  reflect(-lightDirection, normalDirection),
                  viewDirection)), _Shininess);
            }

            float cookieAttenuation = 1.0;
               // by default no cookie attenuation
            #if defined DIRECTIONAL_COOKIE
               cookieAttenuation = texture2D(_LightTexture0,
                  vec2(positionInLightSpace)).a;
            #elif defined POINT_COOKIE
               cookieAttenuation = textureCube(_LightTexture0,
                  vec3(positionInLightSpace)).a;
            #elif defined SPOT
               cookieAttenuation = texture2D(_LightTexture0,
                  vec2(positionInLightSpace)
                  / positionInLightSpace.w + vec2(0.5)).a;
            #endif
            gl_FragColor = vec4(cookieAttenuation *
               (diffuseReflection + specularReflection), 1.0);
         }

         #endif

         ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Specular"
}
```

Note that the cookie for a point light source is using a cube texture map. This kind of texture map is discussed in GLSL PROGRAMMING/UNITY/REFLECTING SURFACES[11].

23.0.131. Summary

Congratulations, you have learned the most important aspects of projective texture mapping. We have seen:

- How to implement cookies for directional light sources.
- How to implement spotlights (with and without user-specified cookies).
- How to implement different shaders for different light sources.

23.0.132. Further Reading

If you still want to know more

- about the shader version for lights without cookies, you should read GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[12].
- about texture mapping and in particular alpha texture maps, you should read GLSL PROGRAMMING/UNITY/TRANSPARENT TEXTURES[13].
- about projective texture mapping in fixed-function OpenGL, you could read NVIDIA's white paper “Projective Texture Mapping” by Cass Everitt (which is available ONLINE[14]).

11 Chapter 26 on page 243
12 Chapter 13 on page 111
13 Chapter 19 on page 161
14 HTTP://DEVELOPER.NVIDIA.COM/CONTENT/PROJECTIVE-TEXTURE-MAPPING

24. Light Attenuation

Figure 39 "The Rich Fool" by Rembrandt Harmenszoon van Rijn, 1627. Note the attenuation of the candlelight with the distance from the candle.

This tutorial covers **textures for light attenuation** or — more generally spoken — textures as lookup tables.

It is based on GLSL PROGRAMMING/UNITY/COOKIES[1]. If you haven't read that tutorial yet, you should read it first.

24.0.133. Texture Maps as Lookup Tables

One can think of a texture map as an approximation to a two-dimensional function that maps the texture coordinates to an RGBA color. If one of the two texture coordinates is kept fixed, the texture map can also represent a one-dimensional function. Thus, it is often possible to replace mathematical expressions that depend only on one or two variables by lookup tables in the form of texture maps. (The

1 Chapter 23 on page 207

limitation is that the resolution of the texture map is limited by the size of the texture image and therefore the accuracy of a texture lookup might be insufficient.)

The main advantage of using such a texture lookup is a potential gain of performance: a texture lookup doesn't depend on the complexity of the mathematical expression but only on the size of the texture image (to a certain degree: the smaller the texture image the more efficient the caching up to the point where the whole texture fits into the cache). However, there is an overhead of using a texture lookup; thus, replacing simple mathematical expressions — including built-in functions — is usually pointless.

Which mathematical expressions should be replaced by texture lookups? Unfortunately, there is no general answer because it depends on the specific GPU whether a specific lookup is faster than evaluating a specific mathematical expression. However, one should keep in mind that a texture map is less simple (since it requires code to compute the lookup table), less explicit (since the mathematical function is encoded in a lookup table), less consistent with other mathematical expressions, and has a wider scope (since the texture is available in the whole fragment shader). These are good reasons to avoid lookup tables. However, the gains in performance might outweigh these reasons. In that case, it is a good idea to include comments that document how to achieve the same effect without the lookup table.

24.0.134. Unity's Texture Lookup for Light Attenuation

Unity actually uses a lookup texture `_LightTextureB0` internally for the light attenuation of point lights and spotlights. (Note that in some cases, e.g. point lights without cookie textures, this lookup texture is set to `_LightTexture0` without `B`. This case is ignored here.) In GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[2], it was described how to implement linear attenuation: we compute an attenuation factor that includes one over the distance between the position of the light source in world space and the position of the rendered fragment in world space. In order to represent this distance, Unity uses the z coordinate in light space. Light space coordinates have been discussed in GLSL PROGRAMMING/UNITY/COOKIES[3]; here, it is only important that we can use the Unity-specific uniform matrix `_LightMatrix0` to transform a position from world space to light space. Analogously to the code in GLSL PROGRAMMING/UNITY/COOKIES[4], we store the position in light space in the varying variable `positionInLightSpace`.

2 Chapter 10 on page 77
3 Chapter 23 on page 207
4 Chapter 23 on page 207

We can then use the z coordinate of this varying to look up the attenuation factor in the alpha component of the texture `_LightTextureB0` in the fragment shader:

```
float distance = positionInLightSpace.z;
   // use z coordinate in light space as signed distance
attenuation =
   texture2D(_LightTextureB0, vec2(distance)).a;
   // texture lookup for attenuation
// alternative with linear attenuation:
//    float distance = length(vertexToLightSource);
//    attenuation = 1.0 / distance;
```

Using the texture lookup, we don't have to compute the length of a vector (which involves three squares and one square root) and we don't have to divide by this length. In fact, the actual attenuation function that is implemented in the lookup table is more complicated in order to avoid saturated colors at short distances. Thus, compared to a computation of this actual attenuation function, we save even more operations.

24.0.135. Complete Shader Code

The shader code is based on the code of GLSL PROGRAMMING/UNITY/COOKIES[5]. The `ForwardBase` pass was slightly simplified by assuming that the light source is always directional without attenuation. The vertex shader of the `ForwardAdd` pass is identical to the code in GLSL PROGRAMMING/UNITY/COOKIES[6] but the fragment shader includes the texture lookup for light attenuation, which is described above. However, the fragment shader lacks the cookie attenuation in order to focus on the attenuation with distance. It is straightforward (and a good exercise) to include the code for the cookie again.

```
Shader "GLSL light attenuation with texture lookup" {
   Properties {
      _Color ("Diffuse Material Color", Color) = (1,1,1,1)
      _SpecColor ("Specular Material Color", Color) = (1,1,1,1)
      _Shininess ("Shininess", Float) = 10
   }
   SubShader {
      Pass {
         Tags { "LightMode" = "ForwardBase" }
            // pass for ambient light and
            // first directional light source without attenuation

         GLSLPROGRAM
```

5 Chapter 23 on page 207
6 Chapter 23 on page 207

```
// User-specified properties
uniform vec4 _Color;
uniform vec4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (except _LightColor0)
// are also defined in "UnityCG.glslinc",
// i.e. one could #include "UnityCG.glslinc"
uniform vec3 _WorldSpaceCameraPos;
   // camera position in world space
uniform mat4 _Object2World; // model matrix
uniform mat4 _World2Object; // inverse model matrix
uniform vec4 _WorldSpaceLightPos0;
   // direction to or position of light source
uniform vec4 _LightColor0;
   // color of light source (from Lighting.cginc)

varying vec4 position;
   // position of the vertex (and fragment) in world space
varying vec3 varyingNormalDirection;
   // surface normal vector in world space

#ifdef VERTEX

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   position = modelMatrix * gl_Vertex;
   varyingNormalDirection = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
   vec3 normalDirection = normalize(varyingNormalDirection);

   vec3 viewDirection =
      normalize(_WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection =
      normalize(vec3(_WorldSpaceLightPos0));
      // we assume that the light source in ForwardBase pass
      // is a directional light source without attenuation

   vec3 ambientLighting =
      vec3(gl_LightModel.ambient) * vec3(_Color);

   vec3 diffuseReflection = vec3(_LightColor0) * vec3(_Color)
      * max(0.0, dot(normalDirection, lightDirection));
```

```
      vec3 specularReflection;
      if (dot(normalDirection, lightDirection) < 0.0)
         // light source on the wrong side?
      {
         specularReflection = vec3(0.0, 0.0, 0.0);
            // no specular reflection
      }
      else // light source on the right side
      {
         specularReflection = vec3(_LightColor0)
            * vec3(_SpecColor) * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _Shininess);
      }

      gl_FragColor = vec4(ambientLighting
         + diffuseReflection + specularReflection, 1.0);
   }

   #endif

   ENDGLSL
}

Pass {
   Tags { "LightMode" = "ForwardAdd" }
      // pass for additional light sources
   Blend One One // additive blending

   GLSLPROGRAM

   // User-specified properties
   uniform vec4 _Color;
   uniform vec4 _SpecColor;
   uniform float _Shininess;

   // The following built-in uniforms (except _LightColor0)
   // are also defined in "UnityCG.glslinc",
   // i.e. one could #include "UnityCG.glslinc"
   uniform vec3 _WorldSpaceCameraPos;
      // camera position in world space
   uniform mat4 _Object2World; // model matrix
   uniform mat4 _World2Object; // inverse model matrix
   uniform vec4 _WorldSpaceLightPos0;
      // direction to or position of light source
   uniform vec4 _LightColor0;
      // color of light source (from Lighting.cginc)

   uniform mat4 _LightMatrix0; // transformation
      // from world to light space (from Autolight.cginc)
   uniform sampler2D _LightTextureB0;
      // texture lookup (from Autolight.cginc)

   varying vec4 position;
      // position of the vertex (and fragment) in world space
   varying vec4 positionInLightSpace;
      // position of the vertex (and fragment) in light space
   varying vec3 varyingNormalDirection;
```

```
   // surface normal vector in world space

#ifdef VERTEX

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   position = modelMatrix * gl_Vertex;
   positionInLightSpace = _LightMatrix0 * position;
   varyingNormalDirection = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
   vec3 normalDirection = normalize(varyingNormalDirection);

   vec3 viewDirection =
      normalize(_WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
   }
   else // point or spot light
   {
      vec3 vertexToLightSource =
         vec3(_WorldSpaceLightPos0 - position);
      lightDirection = normalize(vertexToLightSource);

      float distance = positionInLightSpace.z;
         // use z coordinate in light space as signed distance
      attenuation =
         texture2D(_LightTextureB0, vec2(distance)).a;
         // texture lookup for attenuation
      // alternative with linear attenuation:
      //    float distance = length(vertexToLightSource);
      //    attenuation = 1.0 / distance;
   }

   vec3 diffuseReflection =
      attenuation * vec3(_LightColor0) * vec3(_Color)
      * max(0.0, dot(normalDirection, lightDirection));

   vec3 specularReflection;
   if (dot(normalDirection, lightDirection) < 0.0)
```

```
               // light source on the wrong side?
            {
               specularReflection = vec3(0.0, 0.0, 0.0);
                  // no specular reflection
            }
            else // light source on the right side
            {
               specularReflection = attenuation * vec3(_LightColor0)
                  * vec3(_SpecColor) * pow(max(0.0, dot(
                  reflect(-lightDirection, normalDirection),
                  viewDirection)), _Shininess);
            }

            gl_FragColor =
               vec4(diffuseReflection + specularReflection, 1.0);
         }

          #endif

          ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Specular"
}
```

If you compare the lighting computed by this shader with the lighting of a built-in shader, you will notice a difference in intensity by a factor of about 2 to 4. However, this is mainly due to additional constant factors in the built-in shaders. It is straightforward to introduce similar constant factors in the code above.

It should be noted that the z coordinate in light space is not equal to the distance from the light source; it's not even proportional to that distance. In fact, the meaning of the z coordinate depends on the matrix `_LightMatrix0`, which is an undocumented feature of Unity and can therefore change anytime. However, it is rather safe to assume that a value of 0 corresponds to very close positions and a value of 1 corresponds to farther positions.

Also note that point lights without cookie textures specify the attenuation lookup texture in `_LightTexture0` instead of `_LightTextureB0`; thus, the code above doesn't work for them.

24.0.136. Computing Lookup Textures

So far, we have used a lookup texture that is provided by Unity. If Unity wouldn't provide us with the texture in `_LightTextureB0`, we had to compute this texture ourselves. Here is some JavaScript code to compute a similar lookup texture. In order to use it, you have to change the name `_LightTextureB0` to

`_LookupTexture` in the shader code and attach the following JavaScript to any game object with the corresponding material:

```
@script ExecuteInEditMode()

public var upToDate : boolean = false;

function Start()
{
   upToDate = false;
}

function Update()
{
   if (!upToDate) // is lookup texture not up to date?
   {
      upToDate = true;
      var texture = new Texture2D(16, 16);
         // width = 16 texels, height = 16 texels
      texture.filterMode = FilterMode.Bilinear;
      texture.wrapMode = TextureWrapMode.Clamp;

      renderer.sharedMaterial.SetTexture("_LookupTexture", texture);
         // "_LookupTexture" has to correspond to the name
         // of the uniform sampler2D variable in the shader
      for (var j : int = 0; j < texture.height; j++)
      {
         for (var i : int = 0; i < texture.width; i++)
         {
            var x : float = (i + 0.5) / texture.width;
               // first texture coordinate
            var y : float = (j + 0.5) / texture.height;
               // second texture coordinate
            var color = Color(0.0, 0.0, 0.0, (1.0 - x) * (1.0 - x));
               // set RGBA of texels
            texture.SetPixel(i, j, color);
         }
      }
      texture.Apply(); // apply all the texture.SetPixel(...) commands
   }
}
```

In this code, `i` and `j` enumerate the texels of the texture image while `x` and `y` represent the corresponding texture coordinates. The function `(1.0-x)*(1.0-x)` for the alpha component of the texture image happens to produce similar results as compared to Unity's lookup texture.

Note that the lookup texture should not be computed in every frame. Rather it should be computed only when necessary. If a lookup texture depends on additional parameters, then the texture should only be recomputed if any parameter has been changed. This can be achieved by storing the parameter values for which a lookup texture has been computed and continuously checking whether any of the new

parameters are different from these stored values. If this is the case, the lookup texture has to be recomputed.

24.0.137. Summary

Congratulations, you have reached the end of this tutorial. We have seen:

- How to use the built-in texture `_LightTextureB0` as a lookup table for light attenuation.
- How to compute your own lookup textures in JavaScript.

24.0.138. Further Reading

If you still want to know more

- about light attenuation for light sources, you should read GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[7].
- about basic texture mapping, you should read GLSL PROGRAMMING/UNITY/TEXTURED SPHERES[8].
- about coordinates in light space, you should read GLSL PROGRAMMING/UNITY/COOKIES[9].
- about the SECS principles (simple, explicit, consistent, minimal scope), you could read Chapter 3 of David Straker's book "C Style: Standards and Guidelines", published by Prentice-Hall in 1991, which is available ONLINE[10].

7 Chapter 10 on page 77
8 Chapter 16 on page 137
9 Chapter 23 on page 207
10 HTTP://SYQUE.COM/CSTYLE/INDEX.HTM

25. Projectors

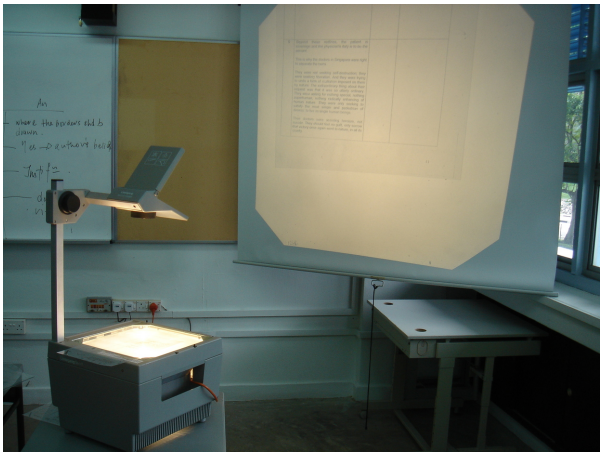

Figure 40 An overhead projector.

This tutorial covers **projective texture mapping for projectors**, which are particular rendering components of Unity.

It is based on GLSL PROGRAMMING/UNITY/COOKIES[1]. If you haven't read that tutorial yet, you should read it first.

25.0.139. Unity's Projectors

Unity's projectors are somewhat similar to spotlights. In fact, they can be used for similar applications. There is, however, an important technical difference: For spotlights, the shaders of all lit objects have to compute the lighting by the spotlight as discussed in GLSL PROGRAMMING/UNITY/COOKIES[2]. If the shader of an object ignores the spotlight, it just won't be lit by the spotlight. This is different for projectors: Each projector is associated with a material with a shader

1 Chapter 23 on page 207
2 Chapter 23 on page 207

that is applied to any object in the projector's range. Thus, an object's shader doesn't need to deal with the projector; instead, the projector applies its shader to all objects in its range as an additional render pass in order to achieve certain effects, e.g. adding the light of a projected image or attenuating the color of an object to fake a shadow. In fact, various effects can be achieved by using different blend equations of the projector's shader. (Blend equations are discussed in GLSL PROGRAMMING/UNITY/TRANSPARENCY[3].)

One might even consider projectors as the more "natural" way of implementing lights. However, the interaction between light and materials is usually specific to each material while the single shader of a projector cannot deal with all these differences. This limits the possibilities of projectors to three basic behaviors: adding light to an object, modulating an object's color, or both, adding light and modulating the object's color. We will look at adding light to an object and attenuating an object's color as an example of modulating them.

25.0.140. Projectors for Adding Light

In order to create a projector, choose **GameObject > Create Empty** from the main menu and then (with the new object still selected) **Component > Effects > Projector** from the main menu. You have now a projector that can be manipulated similarly to a spotlight. The settings of the projector in the **Inspector View** are discussed in UNITY'S REFERENCE MANUAL[4]. Here, the only important setting is the projector's **Material**, which will be applied to all objects in its range. Thus, we have to create another material and assign a suitable shader to it. This shader usually doesn't have access to the materials of the game objects, which it is applied to; therefore, it doesn't have access to their textures etc. Neither does it have access to any information about light sources. However, it has access to the attributes of the vertices of the game objects and its own shader properties.

A shader to add light to objects could be used to project any image onto other objects, similarly to an overhead projector or a movie projector. Thus, it should use a texture image similar to a cookie for spotlights (see GLSL PROGRAMMING/UNITY/COOKIES[5]) except that the RGB colors of the texture image should be added to allow for colored projections. We achieve this by setting the fragment color to the RGBA color of the texture image and using the blend equation

3 Chapter 7 on page 49

4 HTTP://UNITY3D.COM/SUPPORT/DOCUMENTATION/COMPONENTS/CLASS-PROJECTOR.HTML

5 Chapter 23 on page 207

`Blend One One`

which just adds this the fragment color to the color in the framebuffer. (Depending on the texture image, it might be better to use `Blend SrcAlpha One` in order to remove any colors with zero opacity.)

Another difference to the cookies of spotlights is that we should use the Unity-specific uniform matrix `_Projector` to transform positions from object space to projector space instead of the matrix `_LightMatrix0`. However, coordinates in projector space work very similar to coordinates in light space — except that the resulting *x* and *y* coordinates are in the correct range; thus, we don't have to bother with adding 0.5. Nonetheless, we have to perform the division by the *w* coordinates (as always for projective texture mapping); either by explicitly dividing *x* and *y* by *w* or by using `Texture2DProj`:

```
Shader "GLSL projector shader for adding light" {
   Properties {
      _ShadowTex ("Projected Image", 2D) = "white" {}
   }
   SubShader {
      Pass {
         Blend One One
            // add color of _ShadowTex to the color in the framebuffer

         GLSLPROGRAM

         // User-specified properties
         uniform sampler2D _ShadowTex;

         // Projector-specific uniforms
         uniform mat4 _Projector; // transformation matrix
            // from object space to projector space

         varying vec4 positionInProjSpace;
            // position of the vertex (and fragment) in projector space

         #ifdef VERTEX

         void main()
         {
            positionInProjSpace = _Projector * gl_Vertex;
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif

         #ifdef FRAGMENT

         void main()
         {
            if (positionInProjSpace.w > 0.0) // in front of projector?
            {
               gl_FragColor = texture2D(_ShadowTex ,
```

```
                  vec2(positionInProjSpace) / positionInProjSpace.w);
               // alternatively: gl_FragColor = texture2DProj(
               //    _ShadowTex, vec3(positionInProjSpace));
            }
            else // behind projector
            {
               gl_FragColor = vec4(0.0);
            }
         }

         #endif

         ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Projector/Light"
}
```

Notice that we have to test whether w is positive (i.e. the fragment is in front of the projector, not behind it). Without this test, the projector would also add light to objects behind it. Furthermore, the texture image has to be square and it is usually a good idea to use textures with wrap mode set to clamp.

Just in case you wondered: the shader property for the texture is called `_ShadowTex` in order to be compatible with the built-in shaders for projectors.

Figure 41 A cartoon character with a drop shadow.

25.0.141. Projectors for Modulating Colors

The basic steps of creating a projector for modulating colors are the same as above. The only difference is the shader code. The following example adds a drop

shadow by attenuating colors, in particular the floor's color. Note that in an actual application, the color of the shadow caster should not be attenuated. This can be achieved by assigning the shadow caster to a particular **Layer** (in the **Inspector View** of the game object) and specifying this layer under **Ignore Layers** in the **Inspector View** of the projector.

In order to give the shadow a certain shape, we use the alpha component of a texture image to determine how dark the shadow is. (Thus, we can use the cookie textures for lights in the standard assets.) In order to attenuate the color in the framebuffer, we should multiply it with 1 minus alpha (i.e. factor 0 for alpha equals 1). Therefore, the appropriate blend equation is:

`Blend Zero OneMinusSrcAlpha`

The `Zero` indicates that we don't add any light. Even if the shadow is too dark, no light should be added; instead, the alpha component should be reduced in the fragment shader, e.g. by multiplying it with a factor less than 1. For an independent modulation of the color components in the framebuffer, we would require `Blend Zero SrcColor` or `Blend Zero OneMinusSrcColor`.

The different blend equation is actually about the only change in the shader code compared to the version for adding light:

```
Shader "GLSL projector shader for drop shadows" {
   Properties {
      _ShadowTex ("Shadow Shape", 2D) = "white" {}
   }
   SubShader {
      Pass {
         Blend Zero OneMinusSrcAlpha // attenuate color in framebuffer
            // by 1 minus alpha of _ShadowTex

         GLSLPROGRAM

         // User-specified properties
         uniform sampler2D _ShadowTex;

         // Projector-specific uniforms
         uniform mat4 _Projector; // transformation matrix
            // from object space to projector space

         varying vec4 positionInProjSpace;
            // position of the vertex (and fragment) in projector space

         #ifdef VERTEX

         void main()
         {
            positionInProjSpace = _Projector * gl_Vertex;
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }
```

```
         #endif

         #ifdef FRAGMENT

         void main()
         {
            if (positionInProjSpace.w > 0.0) // in front of projector?
            {
               gl_FragColor = texture2D(_ShadowTex ,
                  vec2(positionInProjSpace) / positionInProjSpace.w);
               // alternatively: gl_FragColor = texture2DProj(
               //    _ShadowTex, vec3(positionInProjSpace));
            }
            else // behind projector
            {
               gl_FragColor = vec4(0.0);
            }
         }

         #endif

         ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Projector/Light"
}
```

25.0.142. Summary

Congratulations, this is the end of this tutorial. We have seen:

- How Unity's projectors work.
- How to implement a shader for a projector to add light to objects.
- How to implement a shader for a projector to attenuate objects' colors.

25.0.143. Further Reading

If you still want to know more

- about the light space (which is very similar to projector space), you should read GLSL PROGRAMMING/UNITY/COOKIES[6].
- about texture mapping and in particular alpha texture maps, you should read GLSL PROGRAMMING/UNITY/TRANSPARENT TEXTURES[7].

6 Chapter 23 on page 207
7 Chapter 19 on page 161

- about projective texture mapping in fixed-function OpenGL, you could read NVIDIA's white paper “Projective Texture Mapping” by Cass Everitt (which is available ONLINE[8]).
- about Unity's projectors, you should read UNITY'S DOCUMENTATION ABOUT PROJECTORS[9].

8 HTTP://DEVELOPER.NVIDIA.COM/CONTENT/PROJECTIVE-TEXTURE-MAPPING

9 HTTP://UNITY3D.COM/SUPPORT/DOCUMENTATION/COMPONENTS/CLASS-PROJECTOR.HTML

Part VI.

Environment Mapping

26. Reflecting Surfaces

Figure 42 An example of a reflecting surface: the "Cloud Gate" sculpture in Chicago.

This tutorial introduces **reflection mapping** (and **cube maps** to implement it).

It's the first in a small series of tutorials about environment mapping using cube maps in Unity. The tutorial is based on the per-pixel lighting described in GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[1] and on the concept of texture mapping, which was introduced in GLSL PROGRAMMING/UNITY/TEXTURED SPHERES[2].

1 Chapter 13 on page 111
2 Chapter 16 on page 137

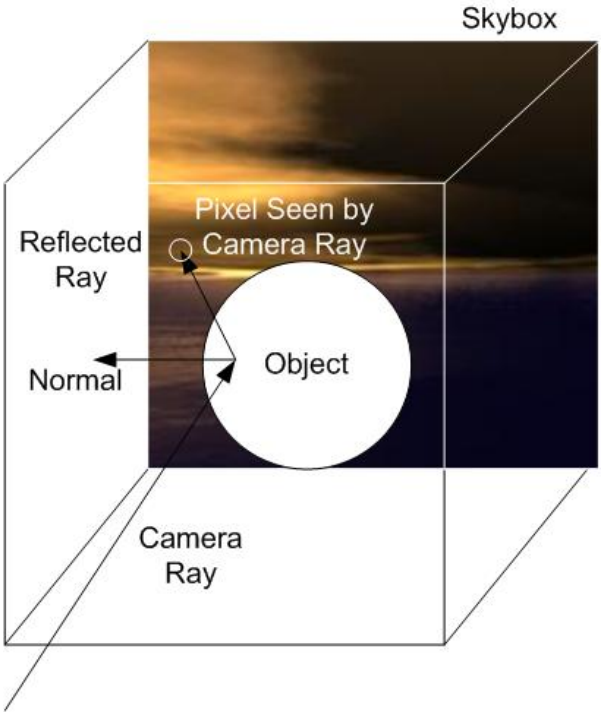

Figure 43 A skybox is a (infinitely) large box surrounding the whole scene. Here a reflected camera ray (i.e. view ray) hits one of the textured faces of the skybox.

26.0.144. Reflection Mapping with a Skybox

The illustration below depicts the concept of reflection mapping with a static skybox: a view ray is reflected at a point on the surface of an object and the reflected ray is intersected with the skybox to determine the color of the corresponding pixel. The skybox is just a large cube with textured faces surrounding the whole scene. It should be noted that skyboxes are usually static and don't include any dynamic objects of the scene. However, "skyboxes" for reflection mapping are often rendered to include the scene from a certain point of view. This is, however, beyond the scope of this tutorial.

Moreover, this tutorial covers only the computation of the reflection, it doesn't cover the rendering of the skybox, which is discussed in GLSL PROGRAMMING/UNI-

TY/SKYBOXES[3]. For the reflection of a skybox in an object, we have to render the object and reflect the rays from the camera to the surface points at the surface normal vectors. The mathematics of this reflection is the same as for the reflection of a light ray at a surface normal vector, which was discussed in GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[4].

Once we have the reflected ray, its intersection with a large skybox has to be computed. This computation actually becomes easier if the skybox is infinitely large: in that case the position of the surface point doesn't matter at all since its distance from the origin of the coordinate system is infinitely small compared to the size of the skybox; thus, only the direction of the reflected ray matters but not its position. Therefore, we can actually also think of a ray that starts in the center of a small skybox instead of a ray that starts somewhere in an infinitely large skybox. (If you are not familiar with this idea, you probably need a bit of time to accept it.) Depending on the direction of the reflected ray, it will intersect one of the six faces of the textured skybox. We could compute, which face is intersected and where the face is intersected and then do a texture lookup (see GLSL PROGRAMMING/UNITY/TEXTURED SPHERES[5]) in the texture image for the specific face. However, GLSL offers cube maps, which support exactly this kind of texture lookups in the six faces of a cube using a direction vector. Thus, all we need to do, is to provide a cube map for the environment as a shader property and use the `textureCube` instruction with the reflected direction to get the color at the corresponding position in the cube map.

26.0.145. Cube Maps

A cube map shader property called `_Cube` can be defined this way in a Unity shader:

```
Properties {
   _Cube ("Reflection Map", Cube) = "" {}
}
```

The corresponding uniform variable is defined this way in a GLSL shader:

```
uniform samplerCube _Cube;
```

3 Chapter 28 on page 253
4 Chapter 11 on page 91
5 Chapter 16 on page 137

To create a cube map, select **Create > Cubemap** in the **Project View**. Then you have to specify six texture images for the faces of the cube in the **Inspector View**. Examples for such textures can be found in **Standard Assets > Skyboxes > Textures**. Furthermore, you should check **MipMaps** in the **Inspector View** for the cube map; this should produce considerably better visual results for reflection mapping.

The vertex shader has to compute the view direction `viewDirection` and the normal direction `normalDirection` as discussed in GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[6]. To reflect the view direction in the fragment shader, we can use the GLSL function `reflect` as also discussed in GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[7]:

```
vec3 reflectedDirection =
   reflect(viewDirection, normalize(normalDirection));
```

And to perform the texture lookup in the cube map and store the resulting color in the fragment color, we simply use:

```
gl_FragColor = textureCube(_Cube, reflectedDirection);
```

That's about it.

26.0.146. Complete Shader Code

The shader code then becomes:

```
Shader "GLSL shader with reflection map" {
   Properties {
      _Cube("Reflection Map", Cube) = "" {}
   }
   SubShader {
      Pass {
         GLSLPROGRAM

         // User-specified uniforms
         uniform samplerCube _Cube;

         // The following built-in uniforms
         // are also defined in "UnityCG.glslinc",
         // i.e. one could #include "UnityCG.glslinc"
         uniform vec3 _WorldSpaceCameraPos;
            // camera position in world space
         uniform mat4 _Object2World; // model matrix
```

6 Chapter 11 on page 91
7 Chapter 11 on page 91

```
         uniform mat4 _World2Object; // inverse model matrix

         // Varyings
         varying vec3 normalDirection;
         varying vec3 viewDirection;

         #ifdef VERTEX

         void main()
         {
            mat4 modelMatrix = _Object2World;
            mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
               // is unnecessary because we normalize vectors

            normalDirection = normalize(vec3(
               vec4(gl_Normal, 0.0) * modelMatrixInverse));
            viewDirection = vec3(modelMatrix * gl_Vertex
               - vec4(_WorldSpaceCameraPos, 1.0));

            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif


         #ifdef FRAGMENT

         void main()
         {
            vec3 reflectedDirection =
               reflect(viewDirection, normalize(normalDirection));
            gl_FragColor = textureCube(_Cube, reflectedDirection);
         }

         #endif

         ENDGLSL
      }
   }
}
```

26.0.147. Summary

Congratulations! You have reached the end of the first tutorial on environment maps. We have seen:

- How to compute the reflection of a skybox in an object.
- How to generate cube maps in Unity and how to use them for reflection mapping.

26.0.148. Further Reading

If you still want to know more

- about the reflection of vectors, you should read GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[8].
- about cube maps in Unity, you should read UNITY'S DOCUMENTATION ABOUT CUBE MAPS[9].

8 Chapter 11 on page 91

9 HTTP://UNITY3D.COM/SUPPORT/DOCUMENTATION/COMPONENTS/CLASS-CUBEMAP.HTML

27. Curved Glass

Figure 44 Crystal balls are examples of curved, transparent surfaces.

This tutorial covers **refraction mapping** and its implementation with cube maps.

It is a variation of GLSL PROGRAMMING/UNITY/REFLECTING SURFACES[1], which should be read first.

27.0.149. Refraction Mapping

In GLSL PROGRAMMING/UNITY/REFLECTING SURFACES[2], we reflected view rays and then performed texture lookups in a cube map in the reflected direction.

1 Chapter 26 on page 243
2 Chapter 26 on page 243

Here, we refract view rays at a curved, transparent surface and then perform the lookups with the refracted direction. The effect will ignore the second refraction when the ray leaves the transparent object again; however, many people hardly notice the differences since such refractions are usually not part of our daily life.

Instead of the `reflect` function, we are using the `refract` function; thus, the fragment shader could be:

```
#ifdef FRAGMENT

void main()
{
   float refractiveIndex = 1.5;
   vec3 refractedDirection = refract(normalize(viewDirection),
      normalize(normalDirection), 1.0 / refractiveIndex);
   gl_FragColor = textureCube(_Cube, refractedDirection);
}

#endif
```

Note that `refract` takes a third argument, which is the refractive index of the outside medium (e.g. 1.0 for air) divided by the refractive index of the object (e.g. 1.5 for some kinds of glass). Also note that the first argument has to be normalized, which isn't necessary for `reflect`.

27.0.150. Complete Shader Code

With the adapted fragment shader, the complete shader code becomes:

```
Shader "GLSL shader with refraction mapping" {
   Properties {
      _Cube ("Environment Map", Cube) = "" {}
   }
   SubShader {
      Pass {
         GLSLPROGRAM

         // User-specified uniforms
         uniform samplerCube _Cube;

         // The following built-in uniforms
         // are also defined in "UnityCG.glslinc",
         // i.e. one could #include "UnityCG.glslinc"
         uniform vec3 _WorldSpaceCameraPos;
            // camera position in world space
         uniform mat4 _Object2World; // model matrix
         uniform mat4 _World2Object; // inverse model matrix

         // Varyings
         varying vec3 normalDirection;
```

```
         varying vec3 viewDirection;

         #ifdef VERTEX

         void main()
         {
            mat4 modelMatrix = _Object2World;
            mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
               // is unnecessary because we normalize vectors

            normalDirection = normalize(vec3(
               vec4(gl_Normal, 0.0) * modelMatrixInverse));
            viewDirection = vec3(modelMatrix * gl_Vertex
               - vec4(_WorldSpaceCameraPos, 1.0));

            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif


         #ifdef FRAGMENT

         void main()
         {
            float refractiveIndex = 1.5;
            vec3 refractedDirection = refract(normalize(viewDirection),
               normalize(normalDirection), 1.0 / refractiveIndex);
            gl_FragColor = textureCube(_Cube, refractedDirection);
         }

         #endif

         ENDGLSL
      }
   }
}
```

27.0.151. Summary

Congratulations. This is the end of another tutorial. We have seen:

- How to adapt reflection mapping to refraction mapping using the `refract` instruction.

27.0.152. Further Reading

If you still want to know more

- about reflection mapping and cube maps, you should read GLSL PROGRAMMING/UNITY/REFLECTING SURFACES[3].
- about the `refract` instruction, you could look it up in the "OpenGL ES Shading Language 1.0.17 Specification" available at the "KHRONOS OPENGL ES API REGISTRY"[4].

3 Chapter 26 on page 243

4 HTTP://WWW.KHRONOS.ORG/REGISTRY/GLES/

28. Skyboxes

Figure 45 View from a skyscraper. As long as the background is static and sufficiently far away, it is a good candidate for a skybox.

This tutorial covers the rendering of **environment maps as backgrounds** with the help of cube maps.

It is based on GLSL PROGRAMMING/UNITY/REFLECTING SURFACES[1]. If you haven't read that tutorial, this would be a very good time to read it.

28.0.153. Rendering a Skybox in the Background

As explained in GLSL PROGRAMMING/UNITY/REFLECTING SURFACES[2], a skybox can be thought of as an infinitely large, textured box that surrounds a scene. Sometimes, skyboxes (or skydomes) are implemented by sufficiently large textured models, which approximate an infinitely large box (or dome). However, GLSL

1 Chapter 26 on page 243
2 Chapter 26 on page 243

PROGRAMMING/UNITY/REFLECTING SURFACES[3] introduced the concept of a cube map, which actually represents an infinitely large box; thus, we don't need the approximation of a box or a dome of limited size. Instead, we can render any screen-filling model (it doesn't matter whether it is a box, a dome, or an apple tree as long as it covers the whole background), compute the view vector from the camera to the rasterized surface point in the vertex shader (as we did in GLSL PROGRAMMING/UNITY/REFLECTING SURFACES[4]) and then perform a lookup in the cube map with this view vector (instead of the reflected view vector in GLSL PROGRAMMING/UNITY/REFLECTING SURFACES[5]) in the fragment shader:

```
#ifdef FRAGMENT

void main()
{
   gl_FragColor = textureCube(_Cube, viewDirection);
}

#endif
```

For best performance we should, of course, render a model with only a few vertices and each pixel should be rasterized only once. Thus, rendering the inside of a cube that surrounds the camera (or the whole scene) is fine.

28.0.154. Complete Shader Code

The shader should be attached to a material, which should be attached to a cube that surrounds the camera. In the shader code, we deactivate writing to the depth buffer with `ZWrite Off` such that no objects are occluded by the skybox. (See the description of the depth test in GLSL PROGRAMMING/PER-FRAGMENT OPERATIONS[6].) Front-face culling is activated with `Cull Front` such that only the "inside" of the cube is rasterized. (See GLSL PROGRAMMING/UNITY/CUTAWAYS[7].) The line `Tags { "Queue" = "Background" }` instructs Unity to render this pass before other objects are rendered. This is not necessary but at least it conveys that this pass renders the background.

```
Shader "GLSL shader for skybox" {
   Properties {
      _Cube ("Environment Map", Cube) = "" {}
```

3 Chapter 26 on page 243
4 Chapter 26 on page 243
5 Chapter 26 on page 243
6 Chapter 47 on page 443
7 Chapter 6 on page 43

```
   }
   SubShader {
      Tags { "Queue" = "Background" }

      Pass {
         ZWrite Off
         Cull Front

         GLSLPROGRAM

         // User-specified uniform
         uniform samplerCube _Cube;

         // The following built-in uniforms
         // are also defined in "UnityCG.glslinc",
         // i.e. one could #include "UnityCG.glslinc"
         uniform vec3 _WorldSpaceCameraPos;
            // camera position in world space
         uniform mat4 _Object2World; // model matrix

         // Varying
         varying vec3 viewDirection;

         #ifdef VERTEX

         void main()
         {
            mat4 modelMatrix = _Object2World;

            viewDirection = vec3(modelMatrix * gl_Vertex
               - vec4(_WorldSpaceCameraPos, 1.0));

            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif


         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor = textureCube(_Cube, viewDirection);
         }

         #endif

         ENDGLSL
      }
   }
}
```

28.0.155. Summary

Congratulations, you have reached the end of another tutorial! We have seen:

- How to render skyboxes.

28.0.156. Further Reading

If you still want to know more

- about cube maps and reflections of skyboxes in objects, you should read GLSL PROGRAMMING/UNITY/REFLECTING SURFACES[8].
- about lighting that is consistent with a skybox, you should read GLSL PROGRAMMING/UNITY/MANY LIGHT SOURCES[9].

8 Chapter 26 on page 243
9 Chapter 29 on page 257

29. Many Light Sources

Figure 46 "Venus de Milo", a famous ancient Greek sculpture. Note the complex lighting environment.

This tutorial introduces **image-based lighting**, in particular **diffuse (irradiance) environment mapping** and its implementation with cube maps.

This tutorial is based on GLSL PROGRAMMING/UNITY/REFLECTING SURFACES[1]. If you haven't read that tutorial, this would be a very good time to read it.

1 Chapter 26 on page 243

29.0.157. Diffuse Lighting by Many Lights

Consider the lighting of the sculpture in the image below . There is natural light coming through the windows. Some of this light bounces off the floor, walls and visitors before reaching the sculpture. Additionally, there are artificial light sources, and their light is also shining directly and indirectly onto the sculpture. How many directional lights and point lights would be needed to simulate this kind of complex lighting environment convincingly? At least more than a handful (probably more than a dozen) and therefore the performance of the lighting computations is challenging.

This problem is addressed by image-based lighting. For static lighting environments that are described by an environment map, e.g. a cube map, image-based lighting allows us to compute the lighting by an arbitrary number of light sources with a single texture lookup in a cube map (see GLSL PROGRAMMING/UNITY/REFLECTING SURFACES[2] for a description of cube maps). How does it work?

In this section we focus on diffuse lighting. Assume that every texel (i.e. pixel) of a cube map acts as a directional light source. (Remember that cube maps are usually assumed to be infinitely large such that only directions matter, but positions don't.) The resulting lighting for a given surface normal direction can be computed as described in GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[3]. It's basically the cosine between the surface normal vector **N** and the vector to the light source **L**:

$$I_{\text{diffuse}} = I_{\text{incoming}}\, k_{\text{diffuse}} \max(0, \mathbf{N} \cdot \mathbf{L})$$

Since the texels are the light sources, **L** is just the direction from the center of the cube to the center of the texel in the cube map. A small cube map with 32×32 texels per face has already $32 \times 32 \times 6 = 6144$ texels. Adding the illumination by thousands of light sources is not going to work in real time. However, for a static cube map we can compute the diffuse illumination for all possible surface normal vectors **N** in advance and store them in a lookup table. When lighting a point on a surface with a specific surface normal vector, we can then just look up the diffuse illumination for the specific surface normal vector **N** in that precomputed lookup table.

Thus, for a specific surface normal vector **N** we add (i.e. integrate) the diffuse illumination by all texels of the cube map. We store the resulting diffuse illumination for this surface normal vector in a second cube map (the “diffuse irradiance

2 Chapter 26 on page 243
3 Chapter 10 on page 77

environment map" or "diffuse environment map" for short). This second cube map will act as a lookup table, where each direction (i.e. surface normal vector) is mapped to a color (i.e. diffuse illumination by potentially thousands of light sources). The fragment shader is therefore really simple (this one could use the vertex shader from GLSL PROGRAMMING/UNITY/REFLECTING SURFACES[4]):

```
#ifdef FRAGMENT

void main()
{
   gl_FragColor = textureCube(_Cube, normalDirection);
}

#endif
```

It is just a lookup of the precomputed diffuse illumination using the surface normal vector of the rasterized surface point. However, the precomputation of the diffuse environment map is somewhat more complicated as described in the next section.

29.0.158. Computation of Diffuse Environment Maps

This section presents some JavaScript code to illustrate the computation of cube maps for diffuse (irradiance) environment maps. In order to use it in Unity, choose **Create > JavaScript** in the **Project View**. Then open the script in Unity's text editor, copy the JavaScript code into it, and attach the script to the game object that has a material with the shader presented below. When a new cube map of sufficiently small dimensions is specified for the shader property `_OriginalCube` (which is labeled **Environment Map** in the shader user interface), the script will update the shader property `_Cube` (i.e. **Diffuse Environment Map** in the user interface) with a corresponding diffuse environment map. Note that the script only accepts cube maps of face dimensions 32×32 or smaller because the computation time tends to be very long for larger cube maps. Thus, when creating a cube map in Unity, make sure to choose a sufficiently small size.

The script includes only a handful of functions: `Awake()` initializes the variables; `Update()` takes care of communicating with the user and the material (i.e. reading and writing shader properties); `computeFilteredCubemap()` does the actual work of computing the diffuse environment map; and `getDirection()` is a small utility function for `computeFilteredCubemap()` to compute the direction associated with each texel of a cube map. Note that `computeFilteredCubemap()` not only integrates the diffuse illumination

4 Chapter 26 on page 243

but also avoids discontinuous seams between faces of the cube map by setting neighboring texels along the seams to the same averaged color.

JavaScript code: click to show/hide

```
@script ExecuteInEditMode()

private var originalCubemap : Cubemap; // a reference to the
   // environment map specified in the shader by the user
private var filteredCubemap : Cubemap; // the diffuse irradiance
   // environment map computed by this script

function Update()
{
   var originalTexture : Texture =
      renderer.sharedMaterial.GetTexture("_OriginalCube");
      // get the user-specified environment map

   if (originalTexture == null)
      // did the user specify "None" for the environment map?
   {
      if (originalCubemap != null)
      {
         originalCubemap = null;
         filteredCubemap = null;
         renderer.sharedMaterial.SetTexture("_Cube", null);
      }
      return;
   }
   else if (originalTexture == originalCubemap
       filteredCubemap != null
       null == renderer.sharedMaterial.GetTexture("_Cube"))
   {
      renderer.sharedMaterial.SetTexture("_Cube", filteredCubemap);
         // set the computed diffuse environment map in the shader
   }
   else if (originalTexture != originalCubemap
      || filteredCubemap
         != renderer.sharedMaterial.GetTexture("_Cube"))
```

```
      // has the user specified a cube map that is different of
      // what we had processed previously?
   {
      if (EditorUtility.DisplayDialog("Processing of Environment Map",
         "Do you want to process the cube map of face size "
         + originalTexture.width + "x" + originalTexture.width
         + "? (This will take some time.)",
         "OK", "Cancel"))
         // does the user really want to process this cube map?
      {
         originalCubemap = originalTexture;

         if (filteredCubemap
            != renderer.sharedMaterial.GetTexture("_Cube"))
         {
            if (null != renderer.sharedMaterial.GetTexture("_Cube"))
            {
               DestroyImmediate(renderer.sharedMaterial.GetTexture(
                  "_Cube")); // clean up
            }
         }
         if (null != filteredCubemap)
         {
            DestroyImmediate(filteredCubemap); // clean up
         }

         computeFilteredCubemap();
            // compute the diffuse environment map

         renderer.sharedMaterial.SetTexture("_Cube", filteredCubemap);
            // set the computed diffuse environment map in the shader
      }
      else // no cancel the processing and reset everything
      {
         originalCubemap = null;
         filteredCubemap = null;
         renderer.sharedMaterial.SetTexture("_OriginalCube", null);
         renderer.sharedMaterial.SetTexture("_Cube", null);
      }
```

```
    }
    return;
}

function computeFilteredCubemap()
    // This function computes a diffuse environment map in
    // "filteredCubemap" of the same dimensions as "originalCubemap"
    // by integrating - for each texel of "filteredCubemap" -
    // the diffuse illumination from all texels of "originalCubemap"
    // for the surface normal vector corresponding to the direction
    // of each texel of "filteredCubemap".
{
    filteredCubemap = Cubemap(originalCubemap.width,
        originalCubemap.format, true);
        // create the diffuse environment cube map

    var filteredSize : int = filteredCubemap.width;
    var originalSize : int = originalCubemap.width;

    // compute all texels of the diffuse environment
    // cube map by iterating over all of them
    for (var filteredFace : int = 0; filteredFace < 6; filteredFace++)
    {
        for (var filteredI : int = 0; filteredI < filteredSize; filteredI++)
        {
            for (var filteredJ : int = 0; filteredJ < filteredSize; filteredJ++)
            {
                var filteredDirection : Vector3 =
                    getDirection(filteredFace,
                    filteredI, filteredJ, filteredSize).normalized;
                var totalWeight : float = 0.0;
                var originalDirection : Vector3;
                var originalFaceDirection : Vector3;
                var weight : float;
                var filteredColor : Color = Color(0.0, 0.0, 0.0);

                // sum (i.e. integrate) the diffuse illumination
                // by all texels in the original environment map
                for (var originalFace : int = 0; originalFace < 6; originalFace++)
```

```
            {
               originalFaceDirection = getDirection(originalFace,
                  1, 1, 3).normalized; // the normal vector of the face

               for (var originalI : int = 0; originalI < originalSize;
originalI++)
               {
                  for (var originalJ : int = 0; originalJ < originalSize;
originalJ++)
                  {
                     originalDirection = getDirection(originalFace,
                        originalI, originalJ, originalSize);
                        // direction to the texel, i.e. light source
                     weight = 1.0 / originalDirection.sqrMagnitude;
                        // take smaller size of more distant texels
                        // into account
                     originalDirection = originalDirection.normalized;
                     weight = weight
                        * Vector3.Dot(originalFaceDirection,
                        originalDirection); // take tilt of texels
                        // compared to face into account
                     weight = weight * Mathf.Max(0.0,
                        Vector3.Dot(filteredDirection,
                        originalDirection));
                        // directional filter for diffuse illumination
                     totalWeight = totalWeight + weight;
                        // instead of analytically normalization,
                        // we just normalize to the potentially
                        // maximum illumination
                     filteredColor = filteredColor +
                        weight * originalCubemap.GetPixel(originalFace,
                        originalI, originalJ);
                        // add the illumination by this texel
                  }
               }
            }
            filteredCubemap.SetPixel(filteredFace, filteredI,
               filteredJ, filteredColor / totalWeight);
               // store the diffuse illumination of this texel
```

```
        }
    }
}

// Avoid seams between cube faces:
// average edge texels to the same color on both sides of the seam
// (except corner texels, see below)
var maxI : int = filteredCubemap.width - 1;
var average : Color;
for (var i : int = 1; i < maxI; i++)
{
    average = (filteredCubemap.GetPixel(0, i, 0)
        + filteredCubemap.GetPixel(2, maxI, maxI - i)) / 2.0;
    filteredCubemap.SetPixel(0, i, 0, average);
    filteredCubemap.SetPixel(2, maxI, maxI - i, average);
    average = (filteredCubemap.GetPixel(0, 0, i)
        + filteredCubemap.GetPixel(4, maxI, i)) / 2.0;
    filteredCubemap.SetPixel(0, 0, i, average);
    filteredCubemap.SetPixel(4, maxI, i, average);
    average = (filteredCubemap.GetPixel(0, i, maxI)
        + filteredCubemap.GetPixel(3, maxI, i)) / 2.0;
    filteredCubemap.SetPixel(0, i, maxI, average);
    filteredCubemap.SetPixel(3, maxI, i, average);
    average = (filteredCubemap.GetPixel(0, maxI, i)
        + filteredCubemap.GetPixel(5, 0, i)) / 2.0;
    filteredCubemap.SetPixel(0, maxI, i, average);
    filteredCubemap.SetPixel(5, 0, i, average);

    average = (filteredCubemap.GetPixel(1, i, 0)
        + filteredCubemap.GetPixel(2, 0, i)) / 2.0;
    filteredCubemap.SetPixel(1, i, 0, average);
    filteredCubemap.SetPixel(2, 0, i, average);
    average = (filteredCubemap.GetPixel(1, 0, i)
        + filteredCubemap.GetPixel(5, maxI, i)) / 2.0;
    filteredCubemap.SetPixel(1, 0, i, average);
    filteredCubemap.SetPixel(5, maxI, i, average);
    average = (filteredCubemap.GetPixel(1, i, maxI)
        + filteredCubemap.GetPixel(3, 0, maxI - i)) / 2.0;
    filteredCubemap.SetPixel(1, i, maxI, average);
```

```
      filteredCubemap.SetPixel(3, 0, maxI - i, average);
      average = (filteredCubemap.GetPixel(1, maxI, i)
         + filteredCubemap.GetPixel(4, 0, i)) / 2.0;
      filteredCubemap.SetPixel(1, maxI, i, average);
      filteredCubemap.SetPixel(4, 0, i, average);

      average = (filteredCubemap.GetPixel(2, i, 0)
         + filteredCubemap.GetPixel(5, maxI - i, 0)) / 2.0;
      filteredCubemap.SetPixel(2, i, 0, average);
      filteredCubemap.SetPixel(5, maxI - i, 0, average);
      average = (filteredCubemap.GetPixel(2, i, maxI)
         + filteredCubemap.GetPixel(4, i, 0)) / 2.0;
      filteredCubemap.SetPixel(2, i, maxI, average);
      filteredCubemap.SetPixel(4, i, 0, average);
      average = (filteredCubemap.GetPixel(3, i, 0)
         + filteredCubemap.GetPixel(4, i, maxI)) / 2.0;
      filteredCubemap.SetPixel(3, i, 0, average);
      filteredCubemap.SetPixel(4, i, maxI, average);
      average = (filteredCubemap.GetPixel(3, i, maxI)
         + filteredCubemap.GetPixel(5, maxI - i, maxI)) / 2.0;
      filteredCubemap.SetPixel(3, i, maxI, average);
      filteredCubemap.SetPixel(5, maxI - i, maxI, average);

   }

   // Avoid seams between cube faces: average corner texels
   // to the same color on all three faces meeting in one corner
   average = (filteredCubemap.GetPixel(0, 0, 0)
      + filteredCubemap.GetPixel(2, maxI, maxI)
      + filteredCubemap.GetPixel(4, maxI, 0)) / 3.0;
   filteredCubemap.SetPixel(0, 0, 0, average);
   filteredCubemap.SetPixel(2, maxI, maxI, average);
   filteredCubemap.SetPixel(4, maxI, 0, average);
   average = (filteredCubemap.GetPixel(0, maxI, 0)
      + filteredCubemap.GetPixel(2, maxI, 0)
      + filteredCubemap.GetPixel(5, 0, 0)) / 3.0;
   filteredCubemap.SetPixel(0, maxI, 0, average);
   filteredCubemap.SetPixel(2, maxI, 0, average);
   filteredCubemap.SetPixel(5, 0, 0, average);
```

```
    average = (filteredCubemap.GetPixel(0, 0, maxI)
       + filteredCubemap.GetPixel(3, maxI, 0)
       + filteredCubemap.GetPixel(4, maxI, maxI)) / 3.0;
    filteredCubemap.SetPixel(0, 0, maxI, average);
    filteredCubemap.SetPixel(3, maxI, 0, average);
    filteredCubemap.SetPixel(4, maxI, maxI, average);
    average = (filteredCubemap.GetPixel(0, maxI, maxI)
       + filteredCubemap.GetPixel(3, maxI, maxI)
       + filteredCubemap.GetPixel(5, 0, maxI)) / 3.0;
    filteredCubemap.SetPixel(0, maxI, maxI, average);
    filteredCubemap.SetPixel(3, maxI, maxI, average);
    filteredCubemap.SetPixel(5, 0, maxI, average);
    average = (filteredCubemap.GetPixel(1, 0, 0)
       + filteredCubemap.GetPixel(2, 0, 0)
       + filteredCubemap.GetPixel(5, maxI, 0)) / 3.0;
    filteredCubemap.SetPixel(1, 0, 0, average);
    filteredCubemap.SetPixel(2, 0, 0, average);
    filteredCubemap.SetPixel(5, maxI, 0, average);
    average = (filteredCubemap.GetPixel(1, maxI, 0)
       + filteredCubemap.GetPixel(2, 0, maxI)
       + filteredCubemap.GetPixel(4, 0, 0)) / 3.0;
    filteredCubemap.SetPixel(1, maxI, 0, average);
    filteredCubemap.SetPixel(2, 0, maxI, average);
    filteredCubemap.SetPixel(4, 0, 0, average);
    average = (filteredCubemap.GetPixel(1, 0, maxI)
       + filteredCubemap.GetPixel(3, 0, maxI)
       + filteredCubemap.GetPixel(5, maxI, maxI)) / 3.0;
    filteredCubemap.SetPixel(1, 0, maxI, average);
    filteredCubemap.SetPixel(3, 0, maxI, average);
    filteredCubemap.SetPixel(5, maxI, maxI, average);
    average = (filteredCubemap.GetPixel(1, maxI, maxI)
       + filteredCubemap.GetPixel(3, 0, 0)
       + filteredCubemap.GetPixel(4, 0, maxI)) / 3.0;
    filteredCubemap.SetPixel(1, maxI, maxI, average);
    filteredCubemap.SetPixel(3, 0, 0, average);
    filteredCubemap.SetPixel(4, 0, maxI, average);

    filteredCubemap.Apply();
       // apply all the texture.SetPixel(...) commands
```

```
}

function getDirection(face : int, i : int, j : int, size : int)
   : Vector3
   // This function computes the direction that is
   // associated with a texel of a cube map
{
   var direction : Vector3;

   if (face == 0)
   {
      direction = Vector3(0.5,
         -((j + 0.5) / size - 0.5), -((i + 0.5) / size - 0.5));
   }
   else if (face == 1)
   {
      direction = Vector3(-0.5,
         -((j + 0.5) / size - 0.5), ((i + 0.5) / size - 0.5));
   }
   else if (face == 2)
   {
      direction = Vector3(((i + 0.5) / size - 0.5),
         0.5, ((j + 0.5) / size - 0.5));
   }
   else if (face == 3)
   {
      direction = Vector3(((i + 0.5) / size - 0.5),
         -0.5, -((j + 0.5) / size - 0.5));
   }
   else if (face == 4)
   {
      direction = Vector3(((i + 0.5) / size - 0.5),
         -((j + 0.5) / size - 0.5), 0.5);
   }
   else if (face == 5)
   {
      direction = Vector3(-((i + 0.5) / size - 0.5),
         -((j + 0.5) / size - 0.5), -0.5);
   }
```

```
   return direction;
}
```

As an alternative to the JavaScript code above, you can also use the following C# code.

C# code: click to show/hide

```
using UnityEngine;
using UnityEditor;
using System.Collections;

[ExecuteInEditMode]
public class ComputeDiffuseEnvironmentMap : MonoBehaviour
{
    public Cubemap originalCubeMap;
       // environment map specified in the shader by the user
    //[System.Serializable]
       // avoid being deleted by the garbage collector,
       // and thus leaking
    private Cubemap filteredCubeMap;
       // the computed diffuse irradience environment map

    private void Update()
    {
        Cubemap originalTexture = null;
        try
        {
            originalTexture = renderer.sharedMaterial.GetTexture(
               "_OriginalCube") as Cubemap;
        }
        catch (System.Exception)
        {
            Debug.LogError("'_OriginalCube' not found on shader. "
            + "Are you using the wrong shader?");
            return;
```

```
        }

        if (originalTexture == null)
           // did the user set "none" for the map?
        {
            if (originalCubeMap != null)
            {
                renderer.sharedMaterial.SetTexture("_Cube", null);
                originalCubeMap = null;
                filteredCubeMap = null;
                return;
            }
        }
        else if (originalTexture == originalCubeMap
            filteredCubeMap != null
            renderer.sharedMaterial.GetTexture("_Cube") == null)
        {
            renderer.sharedMaterial.SetTexture("_Cube",
               filteredCubeMap); // set the computed
               // diffuse environment map in the shader
        }
        else if (originalTexture != originalCubeMap
           || filteredCubeMap
           != renderer.sharedMaterial.GetTexture("_Cube"))
        {
            if (EditorUtility.DisplayDialog(
                "Processing of Environment Map",
                "Do you want to process the cube map of face size "
                + originalTexture.width + "x" + originalTexture.width
                + "? (This will take some time.)",
                "OK", "Cancel"))
            {
                if (filteredCubeMap
                   != renderer.sharedMaterial.GetTexture("_Cube"))
                {
                    if (renderer.sharedMaterial.GetTexture("_Cube")
                       != null)
                    {
                        DestroyImmediate(
```

```
                        renderer.sharedMaterial.GetTexture(
                        "_Cube")); // clean up
                }
            }
            if (filteredCubeMap != null)
            {
                DestroyImmediate(filteredCubeMap); // clean up
            }
            originalCubeMap = originalTexture;
            filteredCubeMap = computeFilteredCubeMap();
               //computes the diffuse environment map
            renderer.sharedMaterial.SetTexture("_Cube",
               filteredCubeMap); // set the computed
               // diffuse environment map in the shader
            return;
        }
        else
        {
            originalCubeMap = null;
            filteredCubeMap = null;
            renderer.sharedMaterial.SetTexture("_Cube", null);
            renderer.sharedMaterial.SetTexture(
               "_OriginalCube", null);
        }
    }
}

// This function computes a diffuse environment map in
// "filteredCubemap" of the same dimensions as "originalCubemap"
// by integrating - for each texel of "filteredCubemap" -
// the diffuse illumination from all texels of "originalCubemap"
// for the surface normal vector corresponding to the direction
// of each texel of "filteredCubemap".
private Cubemap computeFilteredCubeMap()
{
    Cubemap filteredCubeMap = new Cubemap(originalCubeMap.width,
       originalCubeMap.format, true);

    int filteredSize = filteredCubeMap.width;
```

```
        int originalSize = originalCubeMap.width;

        // Compute all texels of the diffuse environment cube map
        // by itterating over all of them
        for (int filteredFace = 0; filteredFace < 6; filteredFace++)
           // the six sides of the cube
        {
            for (int filteredI = 0; filteredI < filteredSize; filteredI++)
            {
                for (int filteredJ = 0; filteredJ < filteredSize; filteredJ++)
                {
                    Vector3 filteredDirection =
                       getDirection(filteredFace,
                       filteredI, filteredJ, filteredSize).normalized;
                    float totalWeight = 0.0f;
                    Vector3 originalDirection;
                    Vector3 originalFaceDirection;
                    float weight;
                    Color filteredColor = new Color(0.0f, 0.0f, 0.0f);

                    // sum (i.e. integrate) the diffuse illumination
                    // by all texels in the original environment map
                    for (int originalFace = 0; originalFace < 6; originalFace++)
                    {
                        originalFaceDirection = getDirection(
                           originalFace, 1, 1, 3).normalized;
                           //the normal vector of the face

                        for (int originalI = 0; originalI < originalSize;
originalI++)
                        {
                            for (int originalJ = 0; originalJ < originalSize;
originalJ++)
                            {
                                originalDirection = getDirection(
                                   originalFace, originalI,
                                   originalJ, originalSize);
                                   // direction to the texel
                                   // (i.e. light source)
```

```
                                weight = 1.0f
                                   / originalDirection.sqrMagnitude;
                                   // take smaller size of more
                                   // distant texels into account
                                originalDirection =
                                   originalDirection.normalized;
                                weight = weight * Vector3.Dot(
                                   originalFaceDirection,
                                   originalDirection);
                                   // take tilt of texel compared
                                   // to face into account
                                weight = weight * Mathf.Max(0.0f,
                                   Vector3.Dot(filteredDirection,
                                   originalDirection));
                                   // directional filter
                                   // for diffuse illumination
                                totalWeight = totalWeight + weight;
                                   // instead of analytically
                                   // normalization, we just normalize
                                   // to the potential max illumination
                                filteredColor = filteredColor + weight
                                   * originalCubeMap.GetPixel(
                                   (CubemapFace)originalFace,
                                   originalI, originalJ); // add the
                                   // illumination by this texel
                            }
                        }
                    }
                    filteredCubeMap.SetPixel(
                       (CubemapFace)filteredFace, filteredI,
                       filteredJ, filteredColor / totalWeight);
                       // store the diffuse illumination of this texel
                }
            }
        }

        // Avoid seams between cube faces: average edge texels
        // to the same color on each side of the seam
        int maxI = filteredCubeMap.width - 1;
```

```
for (int i = 0; i < maxI; i++)
{
    setFaceAverage(ref filteredCubeMap,
       0, i, 0, 2, maxI, maxI - i);
    setFaceAverage(ref filteredCubeMap,
       0, 0, i, 4, maxI, i);
    setFaceAverage(ref filteredCubeMap,
       0, i, maxI, 3, maxI, i);
    setFaceAverage(ref filteredCubeMap,
       0, maxI, i, 5, 0, i);

    setFaceAverage(ref filteredCubeMap,
       1, i, 0, 2, 0, i);
    setFaceAverage(ref filteredCubeMap,
       1, 0, i, 5, maxI, i);
    setFaceAverage(ref filteredCubeMap,
       1, i, maxI, 3, 0, maxI - i);
    setFaceAverage(ref filteredCubeMap,
       1, maxI, i, 4, 0, i);

    setFaceAverage(ref filteredCubeMap,
       2, i, 0, 5, maxI - i, 0);
    setFaceAverage(ref filteredCubeMap,
       2, i, maxI, 4, i, 0);
    setFaceAverage(ref filteredCubeMap,
       3, i, 0, 4, i, maxI);
    setFaceAverage(ref filteredCubeMap,
       3, i, maxI, 5, maxI - i, maxI);
}

// Avoid seams between cube faces:
// average corner texels to the same color
// on all three faces meeting in one corner
setCornerAverage(ref filteredCubeMap,
   0, 0, 0, 2, maxI, maxI, 4, maxI, 0);
setCornerAverage(ref filteredCubeMap,
   0, maxI, 0, 2, maxI, 0, 5, 0, 0);
setCornerAverage(ref filteredCubeMap,
   0, 0, maxI, 3, maxI, 0, 4, maxI, maxI);
```

```
        setCornerAverage(ref filteredCubeMap,
           0, maxI, maxI, 3, maxI, maxI, 5, 0, maxI);
        setCornerAverage(ref filteredCubeMap,
           1, 0, 0, 2, 0, 0, 5, maxI, 0);
        setCornerAverage(ref filteredCubeMap,
           1, maxI, 0, 2, 0, maxI, 4, 0, 0);
        setCornerAverage(ref filteredCubeMap,
           1, 0, maxI, 3, 0, maxI, 5, maxI, maxI);
        setCornerAverage(ref filteredCubeMap,
           1, maxI, maxI, 3, 0, 0, 4, 0, maxI);

        filteredCubeMap.Apply(); //apply all SetPixel(..) commands

        return filteredCubeMap;
    }

    private void setFaceAverage(ref Cubemap filteredCubeMap,
       int a, int b, int c, int d, int e, int f)
    {
        Color average =
           (filteredCubeMap.GetPixel((CubemapFace)a, b, c)
           + filteredCubeMap.GetPixel((CubemapFace)d, e, f)) / 2.0f;
        filteredCubeMap.SetPixel((CubemapFace)a, b, c, average);
        filteredCubeMap.SetPixel((CubemapFace)d, e, f, average);
    }

    private void setCornerAverage(ref Cubemap filteredCubeMap,
        int a, int b, int c, int d, int e, int f, int g, int h, int i)
    {
        Color average =
           (filteredCubeMap.GetPixel((CubemapFace)a, b, c)
           + filteredCubeMap.GetPixel((CubemapFace)d, e, f)
           + filteredCubeMap.GetPixel((CubemapFace)g, h, i)) / 3.0f;
        filteredCubeMap.SetPixel((CubemapFace)a, b, c, average);
        filteredCubeMap.SetPixel((CubemapFace)d, e, f, average);
        filteredCubeMap.SetPixel((CubemapFace)g, h, i, average);
    }

    private Vector3 getDirection(int face, int i, int j, int size)
```

```
    {
        switch (face)
        {
            case 0:
                return new Vector3(0.5f,
                   -((j + 0.5f) / size - 0.5f),
                   -((i + 0.5f) / size - 0.5f));
            case 1:
                return new Vector3(-0.5f,
                   -((j + 0.5f) / size - 0.5f),
                   ((i + 0.5f) / size - 0.5f));
            case 2:
                return new Vector3(((i + 0.5f) / size - 0.5f),
                   0.5f, ((j + 0.5f) / size - 0.5f));
            case 3:
                return new Vector3(((i + 0.5f) / size - 0.5f),
                   -0.5f, -((j + 0.5f) / size - 0.5f));
            case 4:
                return new Vector3(((i + 0.5f) / size - 0.5f),
                   -((j + 0.5f) / size - 0.5f), 0.5f);
            case 5:
                return new Vector3(-((i + 0.5f) / size - 0.5f),
                   -((j + 0.5f) / size - 0.5f), -0.5f);
            default:
                return Vector3.zero;
        }
    }
}
```

29.0.159. Complete Shader Code

As promised, the actual shader code is very short; the vertex shader is a reduced version of the vertex shader of GLSL PROGRAMMING/UNITY/REFLECTING SURFACES[5]:

5 Chapter 26 on page 243

```
Shader "GLSL shader with image-based diffuse lighting" {
   Properties {
      _OriginalCube ("Environment Map", Cube) = "" {}
      _Cube ("Diffuse Environment Map", Cube) = "" {}
   }
   SubShader {
      Pass {
         GLSLPROGRAM

         // Uniform specified by the user or by a script
         uniform samplerCube _Cube; // the diffuse environment map

         // The following built-in uniforms
         // are also defined in "UnityCG.glslinc",
         // i.e. one could #include "UnityCG.glslinc"
         uniform mat4 _World2Object; // inverse model matrix

         // Varyings
         varying vec3 normalDirection;

         #ifdef VERTEX

         void main()
         {
            mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
               // is unnecessary because we normalize vectors

            normalDirection = normalize(vec3(
               vec4(gl_Normal, 0.0) * modelMatrixInverse));

            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif

         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor = textureCube(_Cube, normalDirection);
         }

         #endif

         ENDGLSL
      }
   }
}
```

29.0.160. Changes for Specular (i.e. Glossy) Reflection

The shader and script above are sufficient to compute diffuse illumination by a large number of static, directional light sources. But what about the specular illumination discussed in GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[6], i.e.:

$$I_{\text{specular}} = I_{\text{incoming}}\, k_{\text{specular}} \max(0, \mathbf{R} \cdot \mathbf{V})^{n_{\text{shininess}}}$$

First, we have to rewrite this equation such that it depends only on the direction to the light source $\mathbf{L}$ and the reflected view vector $\mathbf{R}_{\text{view}}$:

$$I_{\text{specular}} = I_{\text{incoming}}\, k_{\text{specular}} \max(0, \mathbf{R}_{\text{view}} \cdot \mathbf{L})^{n_{\text{shininess}}}$$

With this equation, we can compute a lookup table (i.e. a cube map) that contains the specular illumination by many light sources for any reflected view vector $\mathbf{R}_{\text{view}}$. In order to look up the specular illumination with such a table, we just need to compute the reflected view vector and perform a texture lookup in a cube map. In fact, this is exactly what the shader code of GLSL PROGRAMMING/UNITY/REFLECTING SURFACES[7] does. Thus, we actually only need to compute the lookup table.

It turns out that the JavaScript code presented above can be easily adapted to compute such a lookup table. All we have to do is to change the line

```
weight = weight * Mathf.Max(0.0,
   Vector3.Dot(filteredDirection, originalDirection));
   // directional filter for diffuse illumination
```

to

```
weight = weight * Mathf.Pow(Mathf.Max(0.0,
   Vector3.Dot(filteredDirection, originalDirection)), 50.0);
   // directional filter for specular illumination
```

where `50.0` should be replaced by a variable for $n_{\text{shininess}}$. This allows us to compute lookup tables for any specific shininess. (The same cube map could be used for varying values of the shininess if the mipmap-level was specified explicitly using the `textureCubeLod` instruction in the shader; however, this technique is beyond the scope of this tutorial.)

6 Chapter 11 on page 91
7 Chapter 26 on page 243

29.0.161. Summary

Congratulations, you have reached the end of a rather advanced tutorial! We have seen:

- What image-based rendering is about.
- How to compute and use a cube map to implement a diffuse environment map.
- How to adapt the code for specular reflection.

29.0.162. Further Reading

If you still want to know more

- about cube maps, you should read GLSL PROGRAMMING/UNITY/REFLECTING SURFACES[8].
- about (dynamic) diffuse environment maps, you could read Chapter 10, "Real-Time Computation of Dynamic Irradiance Environment Maps" by Gary King of the book "GPU Gems 2" by Matt Pharr (editor) published 2005 by Addison-Wesley, which is available ONLINE[9].

8 Chapter 26 on page 243

9 HTTP://HTTP.DEVELOPER.NVIDIA.COM/GPUGEMS2/GPUGEMS2_CHAPTER10.HTML

Part VII.

Variations on Lighting

30. Brushed Metal

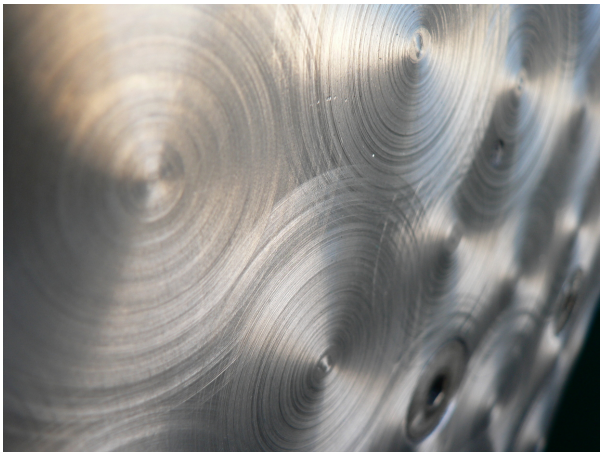

Figure 47 Brushed aluminium. Note the form of the specular highlights, which is far from being round.

This tutorial covers **anisotropic specular highlights**.

It is one of several tutorials about lighting that go beyond the Phong reflection model. However, it is based on lighting with the Phong reflection model as described in GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[1] (for per-vertex lighting) and GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[2] (for per-pixel lighting). If you haven't read those tutorials yet, you should read them first.

While the Phong reflection model is reasonably good for paper, plastics, and some other materials with isotropic reflection (i.e. round highlights), this tutorial looks specifically at materials with anisotropic reflection (i.e. non-round highlights), for example brushed aluminium as in the photo below .

1 Chapter 11 on page 91
2 Chapter 13 on page 111

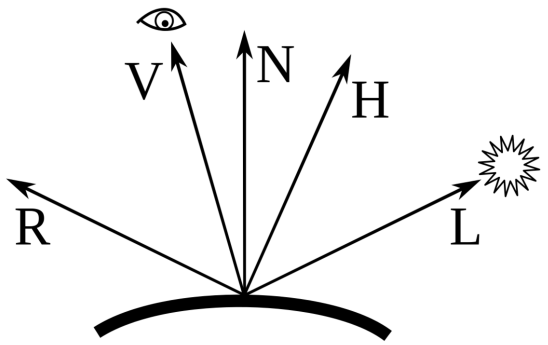

Figure 48 In addition to most of the vectors used by the Phong reflection model, we require the normalized halfway vector H, which is the direction exactly between the direction to the viewer V and the direction to the light source L.

30.0.163. Ward's Model of Anisotropic Reflection

Gregory Ward published a suitable model of anisotropic reflection in his work "Measuring and Modeling Anisotropic Reflection", Computer Graphics (SIGGRAPH '92 Proceedings), pp. 265–272, July 1992. (A copy of the paper is available ONLINE[3].) This model describes the reflection in terms of a BRDF (bidirectional reflectance distribution function), which is a four-dimensional function that describes how a light ray from any direction is reflected into any other direction. His BRDF model consists of two terms: a diffuse reflectance term, which is ρ_d/π, and a more complicated specular reflectance term.

Let's have a look at the diffuse term ρ_d/π first: π is just a constant (about 3.14159) and ρ_d specifies the diffuse reflectance. For colored light, a reflectance for each wave length is necessary in principle, but usually one just specifies a reflectance for each of the three color components: red, green, and blue. If we include the constant π, ρ_d/π just represents the diffuse material color k_{diffuse}, which we have first seen in GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[4] but which also appears in the Phong reflection model (see GLSL PROGRAMMING/UNITY/SPECULAR

3 HTTP://RADSITE.LBL.GOV/RADIANCE/PAPERS/SG92/PAPER.HTML
4 Chapter 10 on page 77

HIGHLIGHTS[5]). You might wonder why the factor max(0, **L**·**N**) doesn't appear in the BRDF. The answer is that the BRDF is defined in such a way that this factor is not included in it (because it isn't really a property of the material) but it should be multiplied with the BRDF when doing any lighting computation.

Thus, in order to implement a given BRDF for opaque materials, we have to multiply all terms of the BRDF with max(0, **L**·**N**) and – unless we want to implement physically correct lighting – we can replace any constant factors by user-specified colors, which usually are easier to control than physical quantities.

For the specular term of his BRDF model, Ward presents an approximation in equation 5b of his paper. I adapted it slightly such that it uses the normalized surface normal vector **N**, the normalized direction to the viewer **V**, the normalized direction to the light source **L**, and the hormalized halfway vector **H** which is (**V** + **L**) / |**V** + **L**|. Using these vectors, Ward's approximation for the specular term becomes:

$$\rho_s \frac{1}{\sqrt{(\mathbf{L}\cdot\mathbf{N})(\mathbf{V}\cdot\mathbf{N})}} \cdot \frac{1}{4\pi\alpha_x\alpha_y} \exp\left(-2\frac{((\mathbf{H}\cdot\mathbf{T})/\alpha_x)^2+((\mathbf{H}\cdot\mathbf{B})/\alpha_y)^2}{1+\mathbf{H}\cdot\mathbf{N}}\right)$$

Here, ρ_s is the specular reflectance, which describes the color and intensity of the specular highlights; α_x and α_y are material constants that describe the shape and size of the highlights. Since all these variables are material constants, we can combine them in one constant k_{specular}. Thus, we get a slightly shorter version:

$$k_{\text{specular}} \frac{1}{\sqrt{(\mathbf{L}\cdot\mathbf{N})(\mathbf{V}\cdot\mathbf{N})}} \exp\left(-2\frac{((\mathbf{H}\cdot\mathbf{T})/\alpha_x)^2+((\mathbf{H}\cdot\mathbf{B})/\alpha_y)^2}{1+\mathbf{H}\cdot\mathbf{N}}\right)$$

Remember that we still have to multiply this BRDF term with **L**·**N** when implementing it in a shader and set it to 0 if **L**·**N** is less than 0. Furthermore, it should also be 0 if **V**·**N** is less than 0, i.e., if we are looking at the surface from the "wrong" side.

There are two vectors that haven't been described yet: **T** and **B**. **T** is the brush direction on the surface and **B** is orthogonal to **T** but also on the surface. Unity provides us with a tangent vector on the surface as a vertex attribute (see GLSL PROGRAMMING/UNITY/DEBUGGING OF SHADERS[6]), which we will use as the vector **T**. Computing the cross product of **N** and **T** generates a vector **B**, which is orthogonal to **N** and **T**, as it should be.

5 Chapter 11 on page 91
6 Chapter 4 on page 23

30.0.164. Implementation of Ward's BRDF Model

We base our implementation on the shader for per-pixel lighting in GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[7]. We need another varying variable `tangentDirection` for the tangent vector **T** (i.e. the brush direction) and we compute two more directions: `halfwayDirection` for the halfway vector **H** and `binormalDirection` for the binormal vector **B**. The properties are `_Color` for k_{diffuse}, `_SpecColor` for k_{specular}, `_AlphaX` for α_x, and `_AlphaY` for α_y.

The fragment shader is then very similar to the version in GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[8] except that it normalizes `tangentDirection`, computes `halfwayDirection` and `binormalDirection`, and implements a different equation for the specular part. Furthermore, this shader computs the dot product **L**·**N** only once and stores it in `dotLN` such that it can be reused without having to recompute it. It looks like this:

```
#ifdef FRAGMENT

 void main()
 {
    vec3 normalDirection = normalize(varyingNormalDirection);
    vec3 tangentDirection = normalize(varyingTangentDirection);

    vec3 viewDirection =
       normalize(_WorldSpaceCameraPos - vec3(position));
    vec3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
       attenuation = 1.0; // no attenuation
       lightDirection = normalize(vec3(_WorldSpaceLightPos0));
    }
    else // point or spot light
    {
       vec3 vertexToLightSource =
          vec3(_WorldSpaceLightPos0 - position);
       float distance = length(vertexToLightSource);
       attenuation = 1.0 / distance; // linear attenuation
       lightDirection = normalize(vertexToLightSource);
    }

    vec3 halfwayVector =
       normalize(lightDirection + viewDirection);
    vec3 binormalDirection =
```

7 Chapter 13 on page 111
8 Chapter 13 on page 111

```
            cross(normalDirection, tangentDirection);
         float dotLN = dot(lightDirection, normalDirection);
            // compute this dot product only once

         vec3 ambientLighting = vec3(gl_LightModel.ambient)
            * vec3(_Color);

         vec3 diffuseReflection = attenuation * vec3(_LightColor0)
            * vec3(_Color) * max(0.0, dotLN);

         vec3 specularReflection;
         if (dotLN < 0.0) // light source on the wrong side?
         {
            specularReflection = vec3(0.0, 0.0, 0.0);
               // no specular reflection
         }
         else // light source on the right side
         {
            float dotHN = dot(halfwayVector, normalDirection);
            float dotVN = dot(viewDirection, normalDirection);
            float dotHTAlphaX =
              dot(halfwayVector, tangentDirection) / _AlphaX;
            float dotHBAlphaY = dot(halfwayVector,
              binormalDirection) / _AlphaY;

            specularReflection = attenuation * vec3(_SpecColor)
               * sqrt(max(0.0, dotLN / dotVN))
               * exp(-2.0 * (dotHTAlphaX * dotHTAlphaX
               + dotHBAlphaY * dotHBAlphaY) / (1.0 + dotHN));
         }

         gl_FragColor = vec4(ambientLighting
            + diffuseReflection + specularReflection, 1.0);
      }

      #endif
```

Note the term `sqrt(max(0, dotLN / dotVN))` which resulted from $\frac{1}{\sqrt{(\mathbf{L}\cdot\mathbf{N})(\mathbf{V}\cdot\mathbf{N})}}$ multiplied with $(\mathbf{L}\cdot\mathbf{N})$ and making sure that everything is greater than 0.

30.0.165. Complete Shader Code

The complete shader code just defines the appropriate properties and the tangent attribute. Also, it requires a second pass with additive blending but without ambient lighting for additional light sources.

```
Shader "GLSL anisotropic per-pixel lighting" {
   Properties {
      _Color ("Diffuse Material Color", Color) = (1,1,1,1)
      _SpecColor ("Specular Material Color", Color) = (1,1,1,1)
      _AlphaX ("Roughness in Brush Direction", Float) = 1.0
```

```
      _AlphaY ("Roughness orthogonal to Brush Direction", Float) = 1.0
   }
   SubShader {
      Pass {
         Tags { "LightMode" = "ForwardBase" }
            // pass for ambient light and first light source

         GLSLPROGRAM

         // User-specified properties
         uniform vec4 _Color;
         uniform vec4 _SpecColor;
         uniform float _AlphaX;
         uniform float _AlphaY;

         // The following built-in uniforms (except _LightColor0)
         // are also defined in "UnityCG.glslinc",
         // i.e. one could #include "UnityCG.glslinc"
         uniform vec3 _WorldSpaceCameraPos;
            // camera position in world space
         uniform mat4 _Object2World; // model matrix
         uniform mat4 _World2Object; // inverse model matrix
         uniform vec4 _WorldSpaceLightPos0;
            // direction to or position of light source
         uniform vec4 _LightColor0;
            // color of light source (from "Lighting.cginc")

         varying vec4 position;
            // position of the vertex (and fragment) in world space
         varying vec3 varyingNormalDirection;
            // surface normal vector in world space
         varying vec3 varyingTangentDirection;
            // brush direction in world space

         #ifdef VERTEX

         attribute vec4 Tangent; // tangent vector provided
            // by Unity (used as brush direction)

         void main()
         {
            mat4 modelMatrix = _Object2World;
            mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
               // is unnecessary because we normalize vectors

            position = modelMatrix * gl_Vertex;
            varyingNormalDirection = normalize(vec3(
               vec4(gl_Normal, 0.0) * modelMatrixInverse));
            varyingTangentDirection = normalize(vec3(
               modelMatrix * vec4(vec3(Tangent), 0.0)));

            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif

         #ifdef FRAGMENT
```

```
void main()
{
   vec3 normalDirection = normalize(varyingNormalDirection);
   vec3 tangentDirection = normalize(varyingTangentDirection);

   vec3 viewDirection =
      normalize(_WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
   }
   else // point or spot light
   {
      vec3 vertexToLightSource =
         vec3(_WorldSpaceLightPos0 - position);
      float distance = length(vertexToLightSource);
      attenuation = 1.0 / distance; // linear attenuation
      lightDirection = normalize(vertexToLightSource);
   }

   vec3 halfwayVector =
      normalize(lightDirection + viewDirection);
   vec3 binormalDirection =
      cross(normalDirection, tangentDirection);
   float dotLN = dot(lightDirection, normalDirection);
      // compute this dot product only once

   vec3 ambientLighting =
      vec3(gl_LightModel.ambient) * vec3(_Color);

   vec3 diffuseReflection = attenuation * vec3(_LightColor0)
      * vec3(_Color) * max(0.0, dotLN);

   vec3 specularReflection;
   if (dotLN < 0.0) // light source on the wrong side?
   {
      specularReflection = vec3(0.0, 0.0, 0.0);
         // no specular reflection
   }
   else // light source on the right side
   {
      float dotHN = dot(halfwayVector, normalDirection);
      float dotVN = dot(viewDirection, normalDirection);
      float dotHTAlphaX =
         dot(halfwayVector, tangentDirection) / _AlphaX;
      float dotHBAlphaY =
         dot(halfwayVector, binormalDirection) / _AlphaY;

      specularReflection = attenuation * vec3(_SpecColor)
         * sqrt(max(0.0, dotLN / dotVN))
         * exp(-2.0 * (dotHTAlphaX * dotHTAlphaX
         + dotHBAlphaY * dotHBAlphaY) / (1.0 + dotHN));
   }
```

```
      gl_FragColor = vec4(ambientLighting
         + diffuseReflection + specularReflection, 1.0);
   }

   #endif

   ENDGLSL
}

Pass {
   Tags { "LightMode" = "ForwardAdd" }
      // pass for additional light sources
   Blend One One // additive blending


   GLSLPROGRAM

   // User-specified properties
   uniform vec4 _Color;
   uniform vec4 _SpecColor;
   uniform float _AlphaX;
   uniform float _AlphaY;

   // The following built-in uniforms (except _LightColor0)
   // are also defined in "UnityCG.glslinc",
   // i.e. one could #include "UnityCG.glslinc"
   uniform vec3 _WorldSpaceCameraPos;
      // camera position in world space
   uniform mat4 _Object2World; // model matrix
   uniform mat4 _World2Object; // inverse model matrix
   uniform vec4 _WorldSpaceLightPos0;
      // direction to or position of light source
   uniform vec4 _LightColor0;
      // color of light source (from "Lighting.cginc")

   varying vec4 position;
      // position of the vertex (and fragment) in world space
   varying vec3 varyingNormalDirection;
      // surface normal vector in world space
   varying vec3 varyingTangentDirection;
      // brush direction in world space

   #ifdef VERTEX

   attribute vec4 Tangent; // tangent vector provided
      // by Unity (used as brush direction)

   void main()
   {
      mat4 modelMatrix = _Object2World;
      mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
         // is unnecessary because we normalize vectors

      position = modelMatrix * gl_Vertex;
      varyingNormalDirection = normalize(vec3(
         vec4(gl_Normal, 0.0) * modelMatrixInverse));
      varyingTangentDirection = normalize(vec3(
         modelMatrix * vec4(vec3(Tangent), 0.0)));
```

```
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
   vec3 normalDirection = normalize(varyingNormalDirection);
   vec3 tangentDirection = normalize(varyingTangentDirection);

   vec3 viewDirection =
      normalize(_WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
   }
   else // point or spot light
   {
      vec3 vertexToLightSource =
         vec3(_WorldSpaceLightPos0 - position);
      float distance = length(vertexToLightSource);
      attenuation = 1.0 / distance; // linear attenuation
      lightDirection = normalize(vertexToLightSource);
   }

   vec3 halfwayVector =
      normalize(lightDirection + viewDirection);
   vec3 binormalDirection =
      cross(normalDirection, tangentDirection);
   float dotLN = dot(lightDirection, normalDirection);
      // compute this dot product only once

   vec3 diffuseReflection = attenuation * vec3(_LightColor0)
      * vec3(_Color) * max(0.0, dotLN);

   vec3 specularReflection;
   if (dotLN < 0.0) // light source on the wrong side?
   {
      specularReflection = vec3(0.0, 0.0, 0.0);
         // no specular reflection
   }
   else // light source on the right side
   {
      float dotHN = dot(halfwayVector, normalDirection);
      float dotVN = dot(viewDirection, normalDirection);
      float dotHTAlphaX =
         dot(halfwayVector, tangentDirection) / _AlphaX;
      float dotHBAlphaY =
         dot(halfwayVector, binormalDirection) / _AlphaY;

      specularReflection = attenuation * vec3(_SpecColor)
```

```
                  * sqrt(max(0.0, dotLN / dotVN))
                  * exp(-2.0 * (dotHTAlphaX * dotHTAlphaX
                  + dotHBAlphaY * dotHBAlphaY) / (1.0 + dotHN));
            }

            gl_FragColor =
               vec4(diffuseReflection + specularReflection, 1.0);
         }

         #endif

         ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Specular"
}
```

30.0.166. Summary

Congratulations, you finished a rather advanced tutorial! We have seen:

- What a BRDF (bidirectional reflectance distribution function) is.
- What Ward's BRDF model for anisotropic reflection is.
- How to implement Ward's BRDF model.

30.0.167. Further Reading

If you still want to know more

- about lighting with the Phong reflection model, you should read GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[9].
- about per-pixel lighting (i.e. Phong shading), you should read GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[10].
- about Ward's BRDF model, you should read his article “Measuring and Modeling Anisotropic Reflection”, Computer Graphics (SIGGRAPH ’92 Proceedings), pp. 265–272, July 1992. (A copy of the paper is available ONLINE[11].) or you could read Section 14.3 of the book “OpenGL Shading Language” (3rd edition) by Randi Rost and others, published 2009 by Addison-Wesley, or Section 8 in the Lighting chapter of the book “Programming Vertex, Geometry, and Pixel Shaders”

9 Chapter 11 on page 91
10 Chapter 13 on page 111
11 HTTP://RADSITE.LBL.GOV/RADIANCE/PAPERS/SG92/PAPER.HTML

(2nd edition, 2008) by Wolfgang Engel, Jack Hoxley, Ralf Kornmann, Niko Suni, and Jason Zink (which is available ONLINE[12].)

12 HTTP://WIKI.GAMEDEV.NET/INDEX.PHP/D3DBOOK:TABLE_OF_CONTENTS

31. Specular Highlights at Silhouettes

Figure 49 Photo of pedestrians in Lisbon. Note the bright silhouettes due to the backlight.

This tutorial covers the **Fresnel factor for specular highlights**.

It is one of several tutorials about lighting that go beyond the Phong reflection model. However, it is based on lighting with the Phong reflection model as described in GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[1] (for per-vertex lighting) and GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[2] (for per-pixel lighting). If you haven't read those tutorials yet, you should read them first.

Many materials (e.g. matte paper) show strong specular reflection when light grazes the surface; i.e., when backlight is reflected from the opposite direction to the viewer as in the photo below . The Fresnel factor explains this strong reflection for some materials. Of course, there are also other reasons for bright silhouettes, e.g. translucent hair or fabrics (see GLSL PROGRAMMING/UNITY/TRANSLUCENT SURFACES[3]).

1 Chapter 11 on page 91
2 Chapter 13 on page 111
3 Chapter 33 on page 309

Interestingly, the effect is often hardly visible because it is most likely when the background of the silhouette is very bright. In this case, however, a bright silhouette will just blend into the background and thus become hardly noticeable.

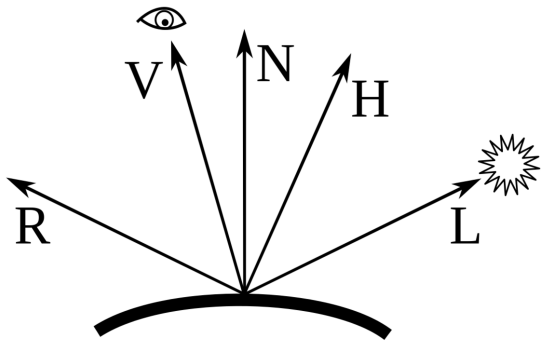

Figure 50 In addition to most of the vectors used by the Phong reflection model, we require the normalized halfway vector H, which is the direction exactly between the direction to the viewer V and the direction to the light source L.

31.0.168. Schlick's Approximation of the Fresnel Factor

The Fresnel factor F_λ describes the specular reflectance of nonconducting materials for unpolarized light of wavelength λ. Schlick's approximation is:

$$F_\lambda = f_\lambda + (1 - f_\lambda)(1 - \mathbf{H} \cdot \mathbf{V})^5$$

where **V** is the normalized direction to the viewer and **H** is the normalized halfway vector: **H** = (**V** + **L**) / |**V** + **L**| with **L** being the normalized direction to the light source. f_λ is the reflectance for **H**·**V** = 1, i.e. when the direction to the light source, the direction to the viewer, and the halfway vector are all identical. One the other hand, F_λ becomes 1 for **H**·**V** = 0, i.e. when the halfway vector is orthogonal to the direction to the viewer, which means that the direction to the light source is opposite to the direction to the viewer (i.e. the case of a grazing light reflection). In fact, F_λ is independent of the wavelength in this case and the material behaves just like a perfect mirror.

Using the built-in GLSL function `mix(x,y,w) = x*(1-w) + y*w` we can rewrite Schlick's approximation as:

$F_\lambda = f_\lambda + (1 - f_\lambda)(1 - \mathbf{H} \cdot \mathbf{V})^5 = f_\lambda \left(1 - (1 - \mathbf{H} \cdot \mathbf{V})^5\right) + (1 - \mathbf{H} \cdot \mathbf{V})^5 = \text{mix}\left(f_\lambda, 1, (1 - \mathbf{H} \cdot \mathbf{V})^5\right)$

which might be slightly more efficient, at least on some GPUs. We will take the dependency on the wavelength into account by allowing for different values of f_λ for each color component; i.e. we consider it an RGB vector. In fact, we identify it with the constant material color k_{specular} from GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[4]. In other words, the Fresnel factor adds a dependency of the material color k_{specular} on the angle between the direction to the viewer and the halfway vector. Thus, we replace the constant material color k_{specular} with Schlick's approximation (using $f_\lambda = k_{\text{specular}}$) in any calculation of the specular reflection.

For example, our equation for the specular term in the Phong reflection model was (see GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[5]):

$I_{\text{specular}} = I_{\text{incoming}} k_{\text{specular}} \max(0, \mathbf{R} \cdot \mathbf{V})^{n_{\text{shininess}}}$

Replacing k_{specular} by Schlick's approximation for the Fresnel factor with $f_\lambda = k_{\text{specular}}$ yields:

$I_{\text{specular}} = I_{\text{incoming}} \text{mix}\left(k_{\text{specular}}, 1, (1 - \mathbf{H} \cdot \mathbf{V})^5\right) \max(0, \mathbf{R} \cdot \mathbf{V})^{n_{\text{shininess}}}$

31.0.169. Implementation

The implementation is based on the shader code from GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[6]. It just computes the halfway vector and includes the approximation of the Fresnel factor:

```
vec3 specularReflection;
if (dot(normalDirection, lightDirection) < 0.0)
   // light source on the wrong side?
{
   specularReflection = vec3(0.0, 0.0, 0.0);
      // no specular reflection
}
else // light source on the right side
{
   vec3 halfwayDirection =
      normalize(lightDirection + viewDirection);
   float w = pow(1.0 - max(0.0,
      dot(halfwayDirection, viewDirection)), 5.0);
   specularReflection = attenuation * vec3(_LightColor0)
      * mix(vec3(_SpecColor), vec3(1.0), w)
```

4 Chapter 11 on page 91
5 Chapter 11 on page 91
6 Chapter 13 on page 111

```
            * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _Shininess);
        }
```

31.0.170. Artistic Control

A useful modification of the implementation above is to replace the power `5.0` by a user-specified shader property. This would give CG artists the option to exaggerate or attenuate the effect of the Fresnel factor depending on their artistic needs.

31.0.171. Consequences for Semitransparent Surfaces

Apart from influencing specular highlights, a Fresnel factor should also influence the opacity α of semitransparent surfaces. In fact, the Fresnel factor describes how a surface becomes more reflective for grazing light rays, which implies that less light is absorbed, refracted, or transmitted, i.e. the transparency T decreases and therefore the opacity $\alpha = 1 - T$ increases. To this end, a Fresnel factor could be computed with the surface normal vector $\mathbf{N}$ instead of the halfway vector $\mathbf{H}$ and the opacity of a semitransparent surface could increase from a user-specified value α_0 (for viewing in the direction of the surface normal) to 1 (independently of the wavelength) with

$$\alpha_{\text{Fresnel}} = \alpha_0 + (1 - \alpha_0)(1 - \mathbf{N} \cdot \mathbf{V})^5.$$

In GLSL PROGRAMMING/UNITY/SILHOUETTE ENHANCEMENT[7] the opacity was considered to result from an attenuation of light as it passes through a layer of semitransparent material. This opacity should be combined with the opacity due to increased reflectivity in the following way: the total opacity α_{total} is 1 minus the total transparency T_{total} which is the product of the transparency due to attenuation $T_{\text{attenuation}}$ (which is 1 minus $\alpha_{\text{attenuation}}$) and the transparency due to the Fresnel factor T_{Fresnel} (which is 1 minus α_{Fresnel}), i.e.:

$$\alpha_{\text{total}} = 1 - T_{\text{total}} \quad = 1 - T_{\text{attenuation}} T_{\text{Fresnel}} \quad = 1 - (1 - \alpha_{\text{attenuation}})(1 - \alpha_{\text{Fresnel}})$$

α_{Fresnel} is the opacity as computed above while $\alpha_{\text{attenuation}}$ is the opacity as computed in GLSL PROGRAMMING/UNITY/SILHOUETTE ENHANCEMENT[8]. For the view direction parallel to the surface normal vector, α_{total} and α_0 could be specified by the user. Then the equation fixes $\alpha_{\text{attenuation}}$ for the normal direction and, in fact, it

7 Chapter 9 on page 67
8 Chapter 9 on page 67

fixes all constants and therefore α_{total} can be computed for all view directions. Note that neither the diffuse reflection nor the specular reflection should be multiplied with the opacity α_{total} since the specular reflection is already multiplied with the Fresnel factor and the diffuse reflection should only be multiplied with the opacity due to attenuation $\alpha_{\text{attenuation}}$.

31.0.172. Complete Shader Code

Putting the code snippet from above in the complete shader from GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[9] results in this shader:

```
Shader "GLSL Fresnel highlights" {
   Properties {
      _Color ("Diffuse Material Color", Color) = (1,1,1,1)
      _SpecColor ("Specular Material Color", Color) = (1,1,1,1)
      _Shininess ("Shininess", Float) = 10
   }
   SubShader {
      Pass {
         Tags { "LightMode" = "ForwardBase" }
            // pass for ambient light and first light source

         GLSLPROGRAM

         // User-specified properties
         uniform vec4 _Color;
         uniform vec4 _SpecColor;
         uniform float _Shininess;

         // The following built-in uniforms (except _LightColor0)
         // are also defined in "UnityCG.glslinc",
         // i.e. one could #include "UnityCG.glslinc"
         uniform vec3 _WorldSpaceCameraPos;
            // camera position in world space
         uniform mat4 _Object2World; // model matrix
         uniform mat4 _World2Object; // inverse model matrix
         uniform vec4 _WorldSpaceLightPos0;
            // direction to or position of light source
         uniform vec4 _LightColor0;
            // color of light source (from "Lighting.cginc")

         varying vec4 position;
            // position of the vertex (and fragment) in world space
         varying vec3 varyingNormalDirection;
            // surface normal vector in world space

         #ifdef VERTEX

         void main()
```

9 Chapter 13 on page 111

```
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   position = modelMatrix * gl_Vertex;
   varyingNormalDirection = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
   vec3 normalDirection = normalize(varyingNormalDirection);

   vec3 viewDirection =
      normalize(_WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
   }
   else // point or spot light
   {
      vec3 vertexToLightSource =
         vec3(_WorldSpaceLightPos0 - position);
      float distance = length(vertexToLightSource);
      attenuation = 1.0 / distance; // linear attenuation
      lightDirection = normalize(vertexToLightSource);
   }

   vec3 ambientLighting =
      vec3(gl_LightModel.ambient) * vec3(_Color);

   vec3 diffuseReflection =
      attenuation * vec3(_LightColor0) * vec3(_Color)
      * max(0.0, dot(normalDirection, lightDirection));

   vec3 specularReflection;
   if (dot(normalDirection, lightDirection) < 0.0)
      // light source on the wrong side?
   {
      specularReflection = vec3(0.0, 0.0, 0.0);
         // no specular reflection
   }
   else // light source on the right side
   {
      vec3 halfwayDirection =
         normalize(lightDirection + viewDirection);
      float w = pow(1.0 - max(0.0,
```

```
            dot(halfwayDirection, viewDirection)), 5.0);
         specularReflection = attenuation * vec3(_LightColor0)
            * mix(vec3(_SpecColor), vec3(1.0), w)
            * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _Shininess);
      }

      gl_FragColor = vec4(ambientLighting
         + diffuseReflection + specularReflection, 1.0);
   }

   #endif

   ENDGLSL
}

Pass {
   Tags { "LightMode" = "ForwardAdd" }
      // pass for additional light sources
   Blend One One // additive blending

   GLSLPROGRAM

   // User-specified properties
   uniform vec4 _Color;
   uniform vec4 _SpecColor;
   uniform float _Shininess;

   // The following built-in uniforms (except _LightColor0)
   // are also defined in "UnityCG.glslinc",
   // i.e. one could #include "UnityCG.glslinc"
   uniform vec3 _WorldSpaceCameraPos;
      // camera position in world space
   uniform mat4 _Object2World; // model matrix
   uniform mat4 _World2Object; // inverse model matrix
   uniform vec4 _WorldSpaceLightPos0;
      // direction to or position of light source
   uniform vec4 _LightColor0;
      // color of light source (from "Lighting.cginc")

   varying vec4 position;
      // position of the vertex (and fragment) in world space
   varying vec3 varyingNormalDirection;
      // surface normal vector in world space

   #ifdef VERTEX

   void main()
   {
      mat4 modelMatrix = _Object2World;
      mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
         // is unnecessary because we normalize vectors

      position = modelMatrix * gl_Vertex;
      varyingNormalDirection = normalize(vec3(
         vec4(gl_Normal, 0.0) * modelMatrixInverse));
```

```
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
   vec3 normalDirection = normalize(varyingNormalDirection);

   vec3 viewDirection =
      normalize(_WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
   }
   else // point or spot light
   {
      vec3 vertexToLightSource =
         vec3(_WorldSpaceLightPos0 - position);
      float distance = length(vertexToLightSource);
      attenuation = 1.0 / distance; // linear attenuation
      lightDirection = normalize(vertexToLightSource);
   }

   vec3 diffuseReflection =
      attenuation * vec3(_LightColor0) * vec3(_Color)
      * max(0.0, dot(normalDirection, lightDirection));

   vec3 specularReflection;
   if (dot(normalDirection, lightDirection) < 0.0)
      // light source on the wrong side?
   {
      specularReflection = vec3(0.0, 0.0, 0.0);
         // no specular reflection
   }
   else // light source on the right side
   {
      vec3 halfwayDirection =
         normalize(lightDirection + viewDirection);
      float w = pow(1.0 - max(0.0,
         dot(halfwayDirection, viewDirection)), 5.0);
      specularReflection = attenuation * vec3(_LightColor0)
         * mix(vec3(_SpecColor), vec3(1.0), w)
         * pow(max(0.0, dot(
         reflect(-lightDirection, normalDirection),
         viewDirection)), _Shininess);
   }

   gl_FragColor =
      vec4(diffuseReflection + specularReflection, 1.0);
}
```

```
         #endif

         ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Specular"
}
```

31.0.173. Summary

Congratulations, you finished one of the somewhat advanced tutorials! We have seen:

- What the Fresnel factor is.
- What Schlick's approximation to the Fresnel factor is.
- How to implement Schlick's approximation for specular highlights.
- How to add more artistic control to the implementation.
- How to use the Fresnel factor for semitransparent surfaces.

31.0.174. Further Reading

If you still want to know more

- about lighting with the Phong reflection model, you should read GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[10].
- about per-pixel lighting (i.e. Phong shading), you should read GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[11].
- about Schlick's approximation, you should read his article “An inexpensive BRDF model for physically-based rendering” by Christophe Schlick, Computer Graphics Forum, 13(3):233—246, 1994. or you could read Section 14.1 of the book “OpenGL Shading Language” (3rd edition) by Randi Rost and others, published 2009 by Addison-Wesley, or Section 5 in the Lighting chapter of the book “Programming Vertex, Geometry, and Pixel Shaders” (2nd edition, 2008) by Wolfgang Engel, Jack Hoxley, Ralf Kornmann, Niko Suni, and Jason Zink (which is available ONLINE[12].)

10 Chapter 11 on page 91
11 Chapter 13 on page 111
12 HTTP://WIKI.GAMEDEV.NET/INDEX.PHP/D3DBOOK:TABLE_OF_CONTENTS

32. Diffuse Reflection of Skylight

Figure 51 A spherical building illuminated by an overcast sky from above and a green water basin from below. Note the different colors of the illumination of the building.

This tutorial covers **hemisphere lighting**.

It is based on diffuse per-vertex lighting as described in GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[1]. If you haven't read that tutorial yet, you should read it first.

Hemisphere lighting basically computes diffuse illumination with a huge light source that covers a whole hemisphere around the scene, for example the sky. It often includes the illumination with another hemisphere from below using a different color since the calculation is almost for free. In the photo below , the spherical building is illuminated by a overcast sky. However, there is also illumination by the green water basin around it, which results in a noticeable greenish illumination of the lower half of the building.

1 Chapter 10 on page 77

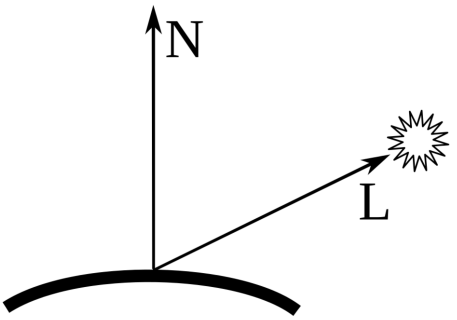

Figure 52 Diffuse reflection can be calculated with the surface normal vector N and the direction to the light source L.

32.0.175. Hemisphere Lighting

If we assume that each point (in direction **L**) of a hemisphere around a point on the surface acts as a light source, then we should integrate the diffuse illumination (given by max(0, **L**·**N**) as discussed in GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[2]) from all the points on the hemisphere by an integral. Let's call the normalized direction of the rotation axis of the hemisphere **U** (for "up"). If the surface normal **N** points in the direction of **U**, we have full illumination with a color specified by the user. If there is an angle γ between them (i.e. $\cos(\gamma) = \mathbf{U}\cdot\mathbf{N}$), only a spherical wedge (SEE THE WIKIPEDIA ARTICLE)[3] of the hemisphere illuminates the surface point. The fraction w of this illumination in comparison to the full illumination is:

$$w = \tfrac{1}{2}(1+\cos(\gamma)) \quad = \tfrac{1}{2}(1+\mathbf{U}\cdot\mathbf{N})$$

Thus, we can compute the incoming light as w times the user-specified color of the full illumination by the hemisphere. The hemisphere in the opposite direction will illuminate the surface point with $1-w$ times another color (which might be black if we don't need it). The next section explains how to derive this equation for w.

2 Chapter 10 on page 77

3 HTTP://EN.WIKIPEDIA.ORG/WIKI/SPHERICAL%20WEDGE

32.0.176. Derivation of the Equation

For anyone interested (and because I didn't find it on the web) here is a derivation of the equation for w. We integrate the illumination over the hemisphere at distance 1 in a spherical coordinate system attached to the surface point with the direction of **N** in the direction of the y axis. If **N** and **U** point in the same direction, the integral is (apart from a constant color specified by the user):

$$\int_0^\pi \mathrm{d}\phi \int_0^\pi \mathrm{d}\theta \sin(\theta)\, \mathbf{L}\cdot\mathbf{N} = \int_0^\pi \mathrm{d}\phi \int_0^\pi \mathrm{d}\theta \sin(\theta) \begin{pmatrix} x \\ y \\ z \end{pmatrix} \cdot \mathbf{N}$$

The term sin(**θ**) is the Jacobian determinant for our integration on the surface of a sphere of radius 1, (x,y,z) is (cos(**φ**)sin(**θ**), sin(**φ**)sin(**θ**), cos(**θ**)), and **N** = (0,1,0). Thus, the integral becomes:

$$\int_0^\pi \mathrm{d}\phi \int_0^\pi \mathrm{d}\theta (\sin(\theta))^2 \sin(\phi) = \pi$$

The constant π will be included in the user-defined color of the maximum illumination. If there is an angle γ with cos(**γ**) = **U**·**N** between **N** and **U**, then the integration is only over a spherical wedge (from γ to π):

$$w = \tfrac{1}{\pi} \int_\gamma^\pi \mathrm{d}\phi \int_0^\pi \mathrm{d}\theta (\sin(\theta))^2 \sin(\phi) \;\; = \tfrac{1}{2}(1+\cos(\gamma)) \;\; = \tfrac{1}{2}(1+\mathbf{U}\cdot\mathbf{N})$$

32.0.177. Shader Code

The implementation is based on the code from GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[4]. In a more elaborated implementation, the contributions of other light sources would also be included, for example using the Phong reflection model as discussed in GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[5]. In that case, hemisphere lighting would be included in the same way as ambient lighting.

Here, however, the only illumination is due to hemisphere lighting. The equation for w is:

$$w = \tfrac{1}{2}(1+\mathbf{U}\cdot\mathbf{N})$$

We implement it in world space, i.e. we have to transform the surface normal vector **N** to world space (see GLSL PROGRAMMING/UNITY/SHADING IN WORLD

4 Chapter 10 on page 77
5 Chapter 11 on page 91

SPACE[6]), while **U** is specified in world space by the user. We normalize the vectors and compute w before using w and $1 - w$ to compute the illumination based on the user-specified colors. Actually, it is pretty straightforward.

```
Shader "GLSL per-vertex hemisphere lighting" {
   Properties {
      _Color ("Diffuse Material Color", Color) = (1,1,1,1)
      _UpperHemisphereColor ("Upper Hemisphere Color", Color)
         = (1,1,1,1)
      _LowerHemisphereColor ("Lower Hemisphere Color", Color)
         = (1,1,1,1)
      _UpVector ("Up Vector", Vector) = (0,1,0,0)
   }
   SubShader {
      Pass {
         GLSLPROGRAM

         // shader properties specified by users
         uniform vec4 _Color;
         uniform vec4 _UpperHemisphereColor;
         uniform vec4 _LowerHemisphereColor;
         uniform vec4 _UpVector;

         // The following built-in uniforms
         // are also defined in "UnityCG.glslinc",
         // i.e. one could #include "UnityCG.glslinc"
         uniform mat4 _Object2World; // model matrix
         uniform mat4 _World2Object; // inverse model matrix

         varying vec4 color;
            // the hemisphere lighting computed in the vertex shader

         #ifdef VERTEX

         void main()
         {
            mat4 modelMatrix = _Object2World;
            mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
               // is unnecessary because we normalize vectors

            vec3 normalDirection = normalize(vec3(
               vec4(gl_Normal, 0.0) * modelMatrixInverse));
            vec3 upDirection = normalize(_UpVector);

            float w = 0.5 * (1.0 + dot(upDirection, normalDirection));
            color = (w * _UpperHemisphereColor
               + (1.0 - w) * _LowerHemisphereColor) * _Color;

            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif
```

6 Chapter 5 on page 31

```
         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor = color;
         }

         #endif

         ENDGLSL
      }
   }
}
```

32.0.178. Summary

Congratulations, you have finished another tutorial! We have seen:

- What hemisphere lighting is.
- What the equation for hemisphere lighting is.
- How to implement hemisphere lighting.

32.0.179. Further Reading

If you still want to know more

- about lighting with the diffuse reflection, you should read GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[7].
- about hemisphere lighting, you could read Section 12.1 of the book "OpenGL Shading Language" (3rd edition) by Randi Rost et al., published 2009 by Addison-Wesley.

7 Chapter 10 on page 77

33. Translucent Surfaces

Figure 53 Leaves lit from both sides: note that the missing specular reflection results in a more saturated green of the backlit leaves.

This tutorial covers **translucent surfaces**.

It is one of several tutorials about lighting that go beyond the Phong reflection model. However, it is based on per-pixel lighting with the Phong reflection model as described in GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[1]. If you haven't read that tutorial yet, you should read it first.

The Phong reflection model doesn't take translucency into account, i.e. the possibility that light is transmitted through a material. This tutorial is about translucent surfaces, i.e. surfaces that allow light to transmit from one face to the other, e.g. paper, clothes, plastic films, or leaves.

1 Chapter 13 on page 111

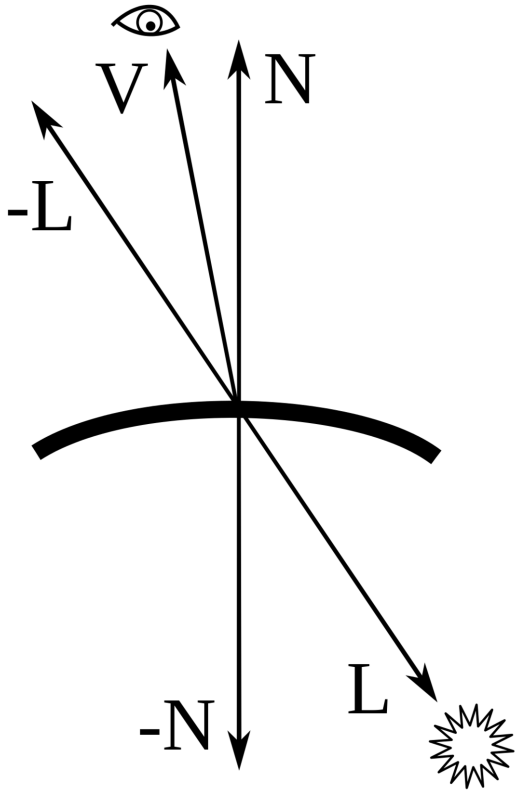

Figure 54 For translucent illumination, the vector V to the viewer and the vector L to the light source are on opposite sides.

33.0.180. Diffuse Translucency

We will distinguish between two kinds of light transmission: diffuse translucency and forward-scattered translucency, which correspond to the diffuse and specular terms in the Phong reflection model. Diffuse translucency is a diffuse transmission of light analogously to the diffuse reflection term in the Phong reflection model (see GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[2]): it only depends on

2 Chapter 10 on page 77

the dot product of the surface normal vector and the direction to the light source — except that we use the negative surface normal vector since the light source is on the backside, thus the equation for the diffuse translucent illumination is:

$$I_{\text{diffuse trans.}} = I_{\text{incoming}} k_{\text{diffuse trans.}} \max(0, \mathbf{L} \cdot (-\mathbf{N}))$$

This is the most common illumination for many translucent surfaces, e.g. paper and leaves.

33.0.181. Forward-Scattered Translucency

Some translucent surfaces (e.g. plastic films) are almost transparent and allow light to shine through the surface almost directly but with some forward scattering; i.e., one can see light sources through the surface but the image is somewhat blurred. This is similar to the specular term of the Phong reflection model (see GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[3] for the equation) except that we replace the reflected light direction **R** by the negative light direction -**L** and the exponent $n_{\text{shininess}}$ corresponds now to the sharpness of the forward-scattered light:

$$I_{\text{forward trans.}} = I_{\text{incoming}} k_{\text{forward trans.}} \max(0, -\mathbf{L} \cdot \mathbf{V})^{n_{\text{sharpness}}}$$

Of course, this model of forward-scattered translucency is not accurate at all but it allows us to fake the effect and tweak the parameters.

33.0.182. Implementation

The following implementation is based on GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[4], which presents per-pixel lighting with the Phong reflection model. The implementation allows for rendering backfaces and flips the surface normal vector in this case. A more elaborated version could also use different colors for the frontface and the backface (see GLSL PROGRAMMING/UNITY/TWO-SIDED SMOOTH SURFACES[5]). In addition to the terms of the Phong reflection model, we also compute illumination by diffuse translucency and forward-scattered translucency. Here is the part that is specific for the fragment shader:

```
#ifdef FRAGMENT
```

3 Chapter 11 on page 91
4 Chapter 13 on page 111
5 Chapter 14 on page 119

```
void main()
{
   vec3 normalDirection = normalize(varyingNormalDirection);
   if (!gl_FrontFacing) // do we look at the backface?
   {
      normalDirection = -normalDirection; // flip normal
   }

   vec3 viewDirection =
      normalize(_WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
   }
   else // point or spot light
   {
      vec3 vertexToLightSource =
         vec3(_WorldSpaceLightPos0 - position);
      float distance = length(vertexToLightSource);
      attenuation = 1.0 / distance; // linear attenuation
      lightDirection = normalize(vertexToLightSource);
   }

   // Computation of the Phong reflection model:

   vec3 ambientLighting =
      vec3(gl_LightModel.ambient) * vec3(_Color);

   vec3 diffuseReflection =
      attenuation * vec3(_LightColor0) * vec3(_Color)
      * max(0.0, dot(normalDirection, lightDirection));

   vec3 specularReflection;
   if (dot(normalDirection, lightDirection) < 0.0)
      // light source on the wrong side?
   {
      specularReflection = vec3(0.0, 0.0, 0.0);
         // no specular reflection
   }
   else // light source on the right side
   {
      specularReflection = attenuation * vec3(_LightColor0)
         * vec3(_SpecColor) * pow(max(0.0, dot(
         reflect(-lightDirection, normalDirection),
         viewDirection)), _Shininess);
   }

   // Computation of the translucent illumination:

   vec3 diffuseTranslucency = attenuation * vec3(_LightColor0)
      * vec3(_DiffuseTranslucentColor)
      * max(0.0, dot(lightDirection, -normalDirection));

   vec3 forwardTranslucency;
```

```
            if (dot(normalDirection, lightDirection) > 0.0)
               // light source on the wrong side?
            {
               forwardTranslucency = vec3(0.0, 0.0, 0.0);
                  // no forward-scattered translucency
            }
            else // light source on the right side
            {
               forwardTranslucency = attenuation * vec3(_LightColor0)
                  * vec3(_ForwardTranslucentColor) * pow(max(0.0,
                  dot(-lightDirection, viewDirection)), _Sharpness);
            }

            // Computation of the complete illumination:

            gl_FragColor = vec4(ambientLighting
               + diffuseReflection + specularReflection
               + diffuseTranslucency + forwardTranslucency, 1.0);
         }

         #endif
```

33.0.183. Complete Shader Code

The complete shader code defines the shader properties for the material constants and adds another pass for additional light sources with additive blending but without the ambient lighting:

```
Shader "GLSL translucent surfaces" {
   Properties {
      _Color ("Diffuse Material Color", Color) = (1,1,1,1)
      _SpecColor ("Specular Material Color", Color) = (1,1,1,1)
      _Shininess ("Shininess", Float) = 10
      _DiffuseTranslucentColor ("Diffuse Translucent Color", Color)
         = (1,1,1,1)
      _ForwardTranslucentColor ("Forward Translucent Color", Color)
         = (1,1,1,1)
      _Sharpness ("Sharpness", Float) = 10
   }
   SubShader {
      Pass {
         Tags { "LightMode" = "ForwardBase" }
            // pass for ambient light and first light source
         Cull Off // show frontfaces and backfaces

         GLSLPROGRAM

         // User-specified properties
         uniform vec4 _Color;
         uniform vec4 _SpecColor;
         uniform float _Shininess;
         uniform vec4 _DiffuseTranslucentColor;
         uniform vec4 _ForwardTranslucentColor;
```

```
uniform float _Sharpness;

// The following built-in uniforms (except _LightColor0)
// are also defined in "UnityCG.glslinc",
// i.e. one could #include "UnityCG.glslinc"
uniform vec3 _WorldSpaceCameraPos;
   // camera position in world space
uniform mat4 _Object2World; // model matrix
uniform mat4 _World2Object; // inverse model matrix
uniform vec4 _WorldSpaceLightPos0;
   // direction to or position of light source
uniform vec4 _LightColor0;
   // color of light source (from "Lighting.cginc")

varying vec4 position;
   // position of the vertex (and fragment) in world space
varying vec3 varyingNormalDirection;
   // surface normal vector in world space

#ifdef VERTEX

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   position = modelMatrix * gl_Vertex;
   varyingNormalDirection = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
   vec3 normalDirection = normalize(varyingNormalDirection);
   if (!gl_FrontFacing) // do we look at the backface?
   {
      normalDirection = -normalDirection; // flip normal
   }

   vec3 viewDirection =
      normalize(_WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
   }
   else // point or spot light
   {
```

```
            vec3 vertexToLightSource =
               vec3(_WorldSpaceLightPos0 - position);
            float distance = length(vertexToLightSource);
            attenuation = 1.0 / distance; // linear attenuation
            lightDirection = normalize(vertexToLightSource);
         }

         vec3 ambientLighting =
            vec3(gl_LightModel.ambient) * vec3(_Color);

         vec3 diffuseReflection =
            attenuation * vec3(_LightColor0) * vec3(_Color)
            * max(0.0, dot(normalDirection, lightDirection));

         vec3 specularReflection;
         if (dot(normalDirection, lightDirection) < 0.0)
            // light source on the wrong side?
         {
            specularReflection = vec3(0.0, 0.0, 0.0);
               // no specular reflection
         }
         else // light source on the right side
         {
            specularReflection = attenuation * vec3(_LightColor0)
               * vec3(_SpecColor) * pow(max(0.0, dot(
               reflect(-lightDirection, normalDirection),
               viewDirection)), _Shininess);
         }

         vec3 diffuseTranslucency = attenuation * vec3(_LightColor0)
            * vec3(_DiffuseTranslucentColor)
            * max(0.0, dot(lightDirection, -normalDirection));

         vec3 forwardTranslucency;
         if (dot(normalDirection, lightDirection) > 0.0)
            // light source on the wrong side?
         {
            forwardTranslucency = vec3(0.0, 0.0, 0.0);
               // no forward-scattered translucency
         }
         else // light source on the right side
         {
            forwardTranslucency = attenuation * vec3(_LightColor0)
               * vec3(_ForwardTranslucentColor) * pow(max(0.0,
               dot(-lightDirection, viewDirection)), _Sharpness);
         }

         gl_FragColor = vec4(ambientLighting
            + diffuseReflection + specularReflection
            + diffuseTranslucency + forwardTranslucency, 1.0);
      }

      #endif

      ENDGLSL
   }

   Pass {
```

```
      Tags { "LightMode" = "ForwardAdd" }
         // pass for additional light sources
      Cull Off
      Blend One One // additive blending

      GLSLPROGRAM

      // User-specified properties
      uniform vec4 _Color;
      uniform vec4 _SpecColor;
      uniform float _Shininess;
      uniform vec4 _DiffuseTranslucentColor;
      uniform vec4 _ForwardTranslucentColor;
      uniform float _Sharpness;

      // The following built-in uniforms (except _LightColor0)
      // are also defined in "UnityCG.glslinc",
      // i.e. one could #include "UnityCG.glslinc"
      uniform vec3 _WorldSpaceCameraPos;
         // camera position in world space
      uniform mat4 _Object2World; // model matrix
      uniform mat4 _World2Object; // inverse model matrix
      uniform vec4 _WorldSpaceLightPos0;
         // direction to or position of light source
      uniform vec4 _LightColor0;
         // color of light source (from "Lighting.cginc")

      varying vec4 position;
         // position of the vertex (and fragment) in world space
      varying vec3 varyingNormalDirection;
         // surface normal vector in world space

      #ifdef VERTEX

      void main()
      {
         mat4 modelMatrix = _Object2World;
         mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
            // is unnecessary because we normalize vectors

         position = modelMatrix * gl_Vertex;
         varyingNormalDirection = normalize(vec3(
            vec4(gl_Normal, 0.0) * modelMatrixInverse));

         gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
      }

      #endif

      #ifdef FRAGMENT

      void main()
      {
         vec3 normalDirection = normalize(varyingNormalDirection);
         if (!gl_FrontFacing) // do we look at the backface?
         {
            normalDirection = -normalDirection; // flip normal
         }
```

```
vec3 viewDirection =
   normalize(_WorldSpaceCameraPos - vec3(position));
vec3 lightDirection;
float attenuation;

if (0.0 == _WorldSpaceLightPos0.w) // directional light?
{
   attenuation = 1.0; // no attenuation
   lightDirection = normalize(vec3(_WorldSpaceLightPos0));
}
else // point or spot light
{
   vec3 vertexToLightSource =
      vec3(_WorldSpaceLightPos0 - position);
   float distance = length(vertexToLightSource);
   attenuation = 1.0 / distance; // linear attenuation
   lightDirection = normalize(vertexToLightSource);
}

vec3 diffuseReflection =
   attenuation * vec3(_LightColor0) * vec3(_Color)
   * max(0.0, dot(normalDirection, lightDirection));

vec3 specularReflection;
if (dot(normalDirection, lightDirection) < 0.0)
   // light source on the wrong side?
{
   specularReflection = vec3(0.0, 0.0, 0.0);
      // no specular reflection
}
else // light source on the right side
{
   specularReflection = attenuation * vec3(_LightColor0)
      * vec3(_SpecColor) * pow(max(0.0, dot(
      reflect(-lightDirection, normalDirection),
      viewDirection)), _Shininess);
}

vec3 diffuseTranslucency = attenuation * vec3(_LightColor0)
   * vec3(_DiffuseTranslucentColor)
   * max(0.0, dot(lightDirection, -normalDirection));

vec3 forwardTranslucency;
if (dot(normalDirection, lightDirection) > 0.0)
   // light source on the wrong side?
{
   forwardTranslucency = vec3(0.0, 0.0, 0.0);
      // no forward-scattered translucency
}
else // light source on the right side
{
   forwardTranslucency = attenuation * vec3(_LightColor0)
      * vec3(_ForwardTranslucentColor) * pow(max(0.0,
      dot(-lightDirection, viewDirection)), _Sharpness);
}

gl_FragColor = vec4(diffuseReflection + specularReflection
```

```
                + diffuseTranslucency + forwardTranslucency, 1.0);
            }

            #endif

            ENDGLSL
        }
    }
    // The definition of a fallback shader should be commented out
    // during development:
    // Fallback "Specular"
}
```

33.0.184. Summary

Congratulations! You finished this tutorial on translucent surfaces, which are very common but cannot be modeled by the Phong reflection model. We have covered:

- What translucent surfaces are.
- Which forms of translucency are most common (diffuse translucency and forward-scattered translucency).
- How to implement diffuse and forward-scattered translucency.

33.0.185. Further Reading

If you still want to know more

- about the diffuse term of the Phong reflection model, you should read GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[6].
- about the ambient or the specular term of the Phong reflection model, you should read GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[7].
- about per-pixel lighting with the Phong reflection model, you should read GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[8].
- about per-pixel lighting of two-sided surfaces, you should read GLSL PROGRAMMING/UNITY/TWO-SIDED SMOOTH SURFACES[9].

6 Chapter 10 on page 77
7 Chapter 11 on page 91
8 Chapter 13 on page 111
9 Chapter 14 on page 119

34. Translucent Bodies

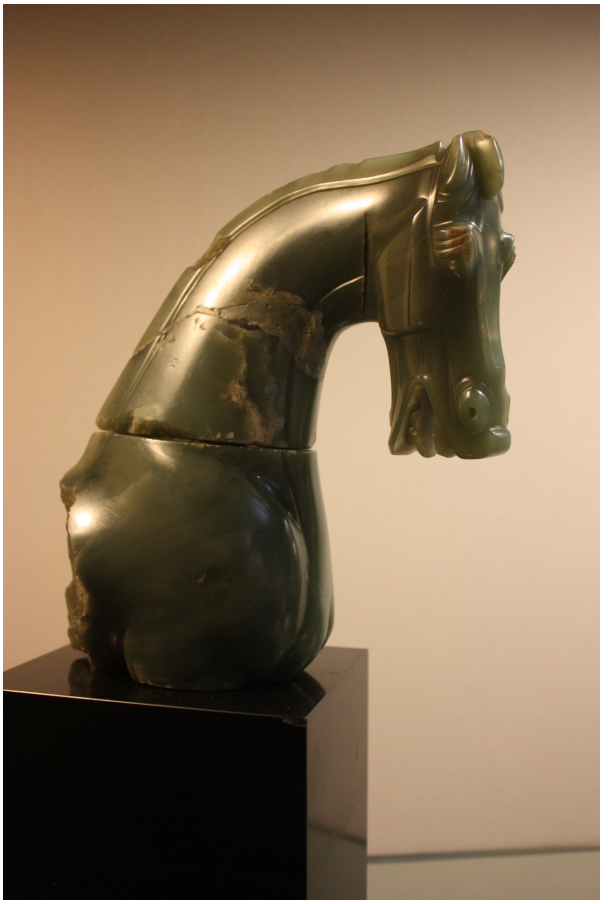

Figure 55 Chinese jade figure (Han dynasty, 206 BC - AD 220). Note the almost wax-like illumination around the nostrils of the horse.

This tutorial covers **translucent bodies**.

It is one of several tutorials about lighting that go beyond the Phong reflection model. However, it is based on per-pixel lighting with the Phong reflection model as

described in GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[1]. If you haven't read that tutorial yet, you should read it first.

The Phong reflection model doesn't take translucency into account, i.e. the possibility that light is transmitted through a material. While GLSL PROGRAMMING/UNITY/TRANSLUCENT SURFACES[2] handled translucent surfaces, this tutorial handles the case of three-dimensional bodies instead of thin surfaces. Examples of translucent materials are wax, jade, marble, skin, etc.

Figure 56 Wax idols. Note the reduced contrast of diffuse lighting.

34.0.186. Waxiness

Unfortunately, the light transport in translucent bodies (i.e. subsurface scattering) is quite challenging in a real-time game engine. Rendering a depth map from the point of view of the light source would help, but since this tutorial is restricted to the free version of Unity, this approach is out of the question. Therefore, we will fake some of the effects of subsurface scattering.

The first effect will be called “waxiness” and describes the smooth, lustrous appearance of wax which lacks the hard contrasts that diffuse reflection can provide. Ideally, we would like to smooth the surface normals before we compute the diffuse reflection (but not the specular reflection) and, in fact, this is possible if a normal

1 Chapter 13 on page 111
2 Chapter 33 on page 309

map is used. Here, however, we take another approach. In order to soften the hard contrasts of diffuse reflection, which is caused by the term max(0, **N**·**L**) (see GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[3]), we reduce the influence of this term as the waxiness w increases from 0 to 1. More specifically, we multiply the term max(0, **N**·**L**) with $1 - w$. However, this will not only reduce the contrast but also the overall brightness of the illumination. To avoid this, we add the waxiness w to fake the additional light due to subsurface scattering, which is stronger the "waxier" a material is.

Thus, instead of this equation for diffuse reflection:

$$I_{\text{diffuse}} = I_{\text{incoming}}\, k_{\text{diffuse}} \max(0, \mathbf{N} \cdot \mathbf{L})$$

we get:

$$I_{\text{diffuse}} = I_{\text{incoming}}\, k_{\text{diffuse}} \left(w + (1 - w) \max(0, \mathbf{N} \cdot \mathbf{L})\right)$$

with the waxiness w between 0 (i.e. regular diffuse reflection) and 1 (i.e. no dependency on **N**·**L**).

This approach is easy to implement, easy to compute for the GPU, easy to control, and it does resemble the appearance of wax and jade, in particular if combined with specular highlights with a high shininess.

Figure 57 Chessmen in backlight. Note the translucency of the white chessmen.

34.0.187. Transmittance of Backlight

The second effect that we are going to fake is backlight that passes through a body and exits at the visible front of the body. This effect is the stronger, the smaller the distance between the back and the front, i.e. in particular at silhouettes, where the

3 Chapter 10 on page 77

distance between the back and the front actually becomes zero. We could, therefore, use the techniques discussed in GLSL PROGRAMMING/UNITY/SILHOUETTE ENHANCEMENT[4] to generate more illumination at the silhouettes. However, the effect becomes somewhat more convincing if we take the actual diffuse illumination at the back of a closed mesh into account. To this end, we proceed as follows:

- We render only back faces and compute the diffuse reflection weighted with a factor that describes how close the point (on the back) is to a silhouette. We mark the pixels with an opacity of 0. (Usually, pixels in the framebuffer have opacity 1. The technique of marking pixels by setting their opacity to 0 is also used and explained in more detail in GLSL PROGRAMMING/UNITY/MIRRORS[5].)
- We render only front faces (in black) and set the color of all pixels that have opacity 1 to black (i.e. all pixels that we haven't rasterized in the first step). This is necessary in case another object intersects with the mesh.
- We render front faces again with the illumination from the front and add the color in the framebuffer multiplied with a factor that describes how close the point (on the front) is to a silhouette.

In the first and third step, we use the silhouette factor 1 - |**N**·**L**|, which is 1 at a silhouette and 0 if the viewer looks straight onto the surface. (An exponent for the dot product could be introduced to allow for more artistic control.) Thus, all the calculations are actually rather straightforward. The complicated part is the blending.

34.0.188. Implementation

The implementation relies heavily on blending, which is discussed in GLSL PROGRAMMING/UNITY/TRANSPARENCY[6]. In addition to three passes corresponding to the steps mentioned above, we also need two more additional passes for additional light sources on the back and the front. With so many passes, it makes sense to get a clear idea of what the render passes are supposed to do. To this end, a skeleton of the shader without the GLSL code is very helpful:

```
Shader "GLSL translucent bodies" {
   Properties {
      _Color ("Diffuse Color", Color) = (1,1,1,1)
      _Waxiness ("Waxiness", Range(0,1)) = 0
      _SpecColor ("Specular Color", Color) = (1,1,1,1)
      _Shininess ("Shininess", Float) = 10
```

4 Chapter 9 on page 67
5 Chapter 41 on page 385
6 Chapter 7 on page 49

```
   _TranslucentColor ("Translucent Color", Color) = (0,0,0,1)
}
SubShader {
   Pass {
      Tags { "LightMode" = "ForwardBase" } // pass for
         // ambient light and first light source on back faces
      Cull Front // render back faces only
      Blend One Zero // mark rasterized pixels in framebuffer
         // with alpha = 0 (usually they should have alpha = 1)

      GLSLPROGRAM
      [...]
      ENDGLSL
   }

   Pass {
      Tags { "LightMode" = "ForwardAdd" }
         // pass for additional light sources on back faces
      Cull Front // render back faces only
      Blend One One // additive blending

      GLSLPROGRAM
      [...]
      ENDGLSL
   }

   Pass {
      Tags { "LightMode" = "ForwardBase" } // pass for
         // setting pixels that were not rasterized to black
      Cull Back // render front faces only (default behavior)
      Blend Zero OneMinusDstAlpha // set colors of pixels
         // with alpha = 1 to black by multiplying with 1-alpha

      GLSLPROGRAM
      #ifdef VERTEX
      void main() {
         gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
      }
      #endif
      #ifdef FRAGMENT
      void main() { gl_FragColor = vec4(0.0); }
      #endif
      ENDGLSL
   }

   Pass {
      Tags { "LightMode" = "ForwardBase" } // pass for
         // ambient light and first light source on front faces
      Cull Back // render front faces only
      Blend One SrcAlpha // multiply color in framebuffer
         // with silhouetteness in fragment's alpha and add colors

      GLSLPROGRAM
      [...]
      ENDGLSL
   }

   Pass {
```

```
         Tags { "LightMode" = "ForwardAdd" }
            // pass for additional light sources on front faces
         Cull Back // render front faces only
         Blend One One // additive blending

         GLSLPROGRAM
         [...]
         ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Specular"
}
```

This skeleton is already quite long; however, it gives a good idea of how the overall shader is organized.

34.0.189. Complete Shader Code

In the following complete shader code, note that the property `_TranslucentColor` instead of `_Color` is used in the computation of the diffuse and ambient part on the back faces. Also note how the "silhouetteness" is computed on the back faces and the front faces; however, it is directly multiplied to the fragment color of the back faces and then indirectly through the alpha component of the fragment color and blending of this alpha with the destination color (the color of pixels in the framebuffer). Finally, the "waxiness" is only used for the diffuse reflection on the front faces.

```
Shader "GLSL translucent bodies" {
   Properties {
      _Color ("Diffuse Color", Color) = (1,1,1,1)
      _Waxiness ("Waxiness", Range(0,1)) = 0
      _SpecColor ("Specular Color", Color) = (1,1,1,1)
      _Shininess ("Shininess", Float) = 10
      _TranslucentColor ("Translucent Color", Color) = (0,0,0,1)
   }
   SubShader {
      Pass {
         Tags { "LightMode" = "ForwardBase" } // pass for
            // ambient light and first light source on back faces
         Cull Front // render back faces only
         Blend One Zero // mark rasterized pixels in framebuffer
            // with alpha = 0 (usually they should have alpha = 1)

         GLSLPROGRAM

         // User-specified properties
         uniform vec4 _Color;
         uniform float _Waxiness;
         uniform vec4 _SpecColor;
```

```
uniform float _Shininess;
uniform vec4 _TranslucentColor;

// The following built-in uniforms (except _LightColor0)
// are also defined in "UnityCG.glslinc",
// i.e. one could #include "UnityCG.glslinc"
uniform vec3 _WorldSpaceCameraPos;
   // camera position in world space
uniform mat4 _Object2World; // model matrix
uniform mat4 _World2Object; // inverse model matrix
uniform vec4 _WorldSpaceLightPos0;
   // direction to or position of light source
uniform vec4 _LightColor0;
   // color of light source (from "Lighting.cginc")

varying vec4 position;
   // position of the vertex (and fragment) in world space
varying vec3 varyingNormalDirection;
   // surface normal vector in world space

#ifdef VERTEX

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   position = modelMatrix * gl_Vertex;
   varyingNormalDirection = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
   vec3 normalDirection = normalize(varyingNormalDirection);

   vec3 viewDirection =
      normalize(_WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
   }
   else // point or spot light
   {
      vec3 vertexToLightSource =
         vec3(_WorldSpaceLightPos0 - position);
      float distance = length(vertexToLightSource);
```

```
            attenuation = 1.0 / distance; // linear attenuation
            lightDirection = normalize(vertexToLightSource);
         }

         vec3 ambientLighting =
            vec3(gl_LightModel.ambient) * vec3(_TranslucentColor);

         vec3 diffuseReflection = attenuation
            * vec3(_LightColor0) * vec3(_TranslucentColor)
            * max(0.0, dot(normalDirection, lightDirection));

         float silhouetteness =
            1.0 - abs(dot(viewDirection, normalDirection));

         gl_FragColor = vec4(silhouetteness
            * (ambientLighting + diffuseReflection), 0.0);
      }

      #endif

      ENDGLSL
   }

   Pass {
      Tags { "LightMode" = "ForwardAdd" }
         // pass for additional light sources on back faces
      Cull Front // render back faces only
      Blend One One // additive blending

      GLSLPROGRAM

      // User-specified properties
      uniform vec4 _Color;
      uniform float _Waxiness;
      uniform vec4 _SpecColor;
      uniform float _Shininess;
      uniform vec4 _TranslucentColor;

      // The following built-in uniforms (except _LightColor0)
      // are also defined in "UnityCG.glslinc",
      // i.e. one could #include "UnityCG.glslinc"
      uniform vec3 _WorldSpaceCameraPos;
         // camera position in world space
      uniform mat4 _Object2World; // model matrix
      uniform mat4 _World2Object; // inverse model matrix
      uniform vec4 _WorldSpaceLightPos0;
         // direction to or position of light source
      uniform vec4 _LightColor0;
         // color of light source (from "Lighting.cginc")

      varying vec4 position;
         // position of the vertex (and fragment) in world space
      varying vec3 varyingNormalDirection;
         // surface normal vector in world space

      #ifdef VERTEX

      void main()
```

```
      {
         mat4 modelMatrix = _Object2World;
         mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
            // is unnecessary because we normalize vectors

         position = modelMatrix * gl_Vertex;
         varyingNormalDirection = normalize(vec3(
            vec4(gl_Normal, 0.0) * modelMatrixInverse));

         gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
      }

      #endif

      #ifdef FRAGMENT

      void main()
      {
         vec3 normalDirection = normalize(varyingNormalDirection);

         vec3 viewDirection =
            normalize(_WorldSpaceCameraPos - vec3(position));
         vec3 lightDirection;
         float attenuation;

         if (0.0 == _WorldSpaceLightPos0.w) // directional light?
         {
            attenuation = 1.0; // no attenuation
            lightDirection = normalize(vec3(_WorldSpaceLightPos0));
         }
         else // point or spot light
         {
            vec3 vertexToLightSource =
               vec3(_WorldSpaceLightPos0 - position);
            float distance = length(vertexToLightSource);
            attenuation = 1.0 / distance; // linear attenuation
            lightDirection = normalize(vertexToLightSource);
         }

         vec3 diffuseReflection = attenuation
            * vec3(_LightColor0) * vec3(_TranslucentColor)
            * max(0.0, dot(normalDirection, lightDirection));

         float silhouetteness =
            1.0 - abs(dot(viewDirection, normalDirection));

         gl_FragColor =
            vec4(silhouetteness * diffuseReflection, 0.0);
      }

      #endif

      ENDGLSL
   }

   Pass {
      Tags { "LightMode" = "ForwardBase" } // pass for
         // setting pixels that were not rasterized to black
```

```
      Cull Back // render front faces only (default behavior)
      Blend Zero OneMinusDstAlpha // set colors of pixels
         // with alpha = 1 to black by multiplying with 1-alpha

      GLSLPROGRAM

      #ifdef VERTEX

      void main()
      {
         gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
      }

      #endif

      #ifdef FRAGMENT

      void main()
      {
         gl_FragColor = vec4(0.0);
      }

      #endif

      ENDGLSL
   }

   Pass {
      Tags { "LightMode" = "ForwardBase" } // pass for
         // ambient light and first light source on front faces
      Cull Back // render front faces only
      Blend One SrcAlpha // multiply color in framebuffer
         // with silhouetteness in fragment's alpha and add colors

      GLSLPROGRAM

      // User-specified properties
      uniform vec4 _Color;
      uniform float _Waxiness;
      uniform vec4 _SpecColor;
      uniform float _Shininess;
      uniform vec4 _TranslucentColor;

      // The following built-in uniforms (except _LightColor0)
      // are also defined in "UnityCG.glslinc",
      // i.e. one could #include "UnityCG.glslinc"
      uniform vec3 _WorldSpaceCameraPos;
         // camera position in world space
      uniform mat4 _Object2World; // model matrix
      uniform mat4 _World2Object; // inverse model matrix
      uniform vec4 _WorldSpaceLightPos0;
        // direction to or position of light source
      uniform vec4 _LightColor0;
        // color of light source (from "Lighting.cginc")

      varying vec4 position;
         // position of the vertex (and fragment) in world space
      varying vec3 varyingNormalDirection;
```

```
   // surface normal vector in world space

#ifdef VERTEX

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   position = modelMatrix * gl_Vertex;
   varyingNormalDirection = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
   vec3 normalDirection = normalize(varyingNormalDirection);

   vec3 viewDirection =
      normalize(_WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
   }
   else // point or spot light
   {
      vec3 vertexToLightSource =
         vec3(_WorldSpaceLightPos0 - position);
      float distance = length(vertexToLightSource);
      attenuation = 1.0 / distance; // linear attenuation
      lightDirection = normalize(vertexToLightSource);
   }

   vec3 ambientLighting =
      vec3(gl_LightModel.ambient) * vec3(_Color);

   vec3 diffuseReflection =
      attenuation * vec3(_LightColor0) * vec3(_Color)
      * (_Waxiness + (1.0 - _Waxiness)
      * max(0.0, dot(normalDirection, lightDirection)));

   vec3 specularReflection;
   if (dot(normalDirection, lightDirection) < 0.0)
      // light source on the wrong side?
   {
      specularReflection = vec3(0.0, 0.0, 0.0);
         // no specular reflection
```

```
         }
         else // light source on the right side
         {
            specularReflection = attenuation * vec3(_LightColor0)
               * vec3(_SpecColor) * pow(max(0.0, dot(
               reflect(-lightDirection, normalDirection),
               viewDirection)), _Shininess);
         }

         float silhouetteness =
            1.0 - abs(dot(viewDirection, normalDirection));

         gl_FragColor = vec4(ambientLighting + diffuseReflection
            + specularReflection, silhouetteness);
      }

      #endif

      ENDGLSL
   }

   Pass {
      Tags { "LightMode" = "ForwardAdd" }
         // pass for additional light sources on front faces
      Cull Back // render front faces only
      Blend One One // additive blending

      GLSLPROGRAM

      // User-specified properties
      uniform vec4 _Color;
      uniform float _Waxiness;
      uniform vec4 _SpecColor;
      uniform float _Shininess;
      uniform vec4 _TranslucentColor;

      // The following built-in uniforms (except _LightColor0)
      // are also defined in "UnityCG.glslinc",
      // i.e. one could #include "UnityCG.glslinc"
      uniform vec3 _WorldSpaceCameraPos;
         // camera position in world space
      uniform mat4 _Object2World; // model matrix
      uniform mat4 _World2Object; // inverse model matrix
      uniform vec4 _WorldSpaceLightPos0;
         // direction to or position of light source
      uniform vec4 _LightColor0;
         // color of light source (from "Lighting.cginc")

      varying vec4 position;
         // position of the vertex (and fragment) in world space
      varying vec3 varyingNormalDirection;
         // surface normal vector in world space

      #ifdef VERTEX

      void main()
      {
         mat4 modelMatrix = _Object2World;
```

```
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   position = modelMatrix * gl_Vertex;
   varyingNormalDirection = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
   vec3 normalDirection = normalize(varyingNormalDirection);

   vec3 viewDirection =
      normalize(_WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
   }
   else // point or spot light
   {
      vec3 vertexToLightSource =
         vec3(_WorldSpaceLightPos0 - position);
      float distance = length(vertexToLightSource);
      attenuation = 1.0 / distance; // linear attenuation
      lightDirection = normalize(vertexToLightSource);
   }

   vec3 diffuseReflection =
      attenuation * vec3(_LightColor0) * vec3(_Color)
      * (_Waxiness + (1.0 - _Waxiness)
      * max(0.0, dot(normalDirection, lightDirection)));

   vec3 specularReflection;
   if (dot(normalDirection, lightDirection) < 0.0)
      // light source on the wrong side?
   {
      specularReflection = vec3(0.0, 0.0, 0.0);
         // no specular reflection
   }
   else // light source on the right side
   {
      specularReflection = attenuation * vec3(_LightColor0)
         * vec3(_SpecColor) * pow(max(0.0, dot(
         reflect(-lightDirection, normalDirection),
         viewDirection)), _Shininess);
   }

   gl_FragColor =
```

```
                vec4(diffuseReflection + specularReflection, 1.0);
            }

            #endif

            ENDGLSL
        }
    }
    // The definition of a fallback shader should be commented out
    // during development:
    // Fallback "Specular"
}
```

34.0.190. Summary

Congratulations! You finished this tutorial on translucent bodies, which was mainly about:

- How to fake the appearance of wax.
- How to fake the appearance of silhouettes of translucent materials lit by backlight.
- How to implement these techniques.

34.0.191. Further Reading

If you still want to know more

- about translucent surfaces, you should read GLSL PROGRAMMING/UNITY/TRANSLUCENT SURFACES[7].
- about the diffuse term of the Phong reflection model, you should read GLSL PROGRAMMING/UNITY/DIFFUSE REFLECTION[8].
- about the ambient or the specular term of the Phong reflection model, you should read GLSL PROGRAMMING/UNITY/SPECULAR HIGHLIGHTS[9].
- about per-pixel lighting with the Phong reflection model, you should read GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[10].
- about basic real-time techniques for subsurface scattering, you could read Chapter 16, “Real-Time Approximations to Subsurface Scattering” by Simon Green of the book “GPU Gems” by Randima Fernando (editor) published 2004 by Addison-Wesley, which is available ONLINE[11].

7 Chapter 33 on page 309
8 Chapter 10 on page 77
9 Chapter 11 on page 91
10 Chapter 13 on page 111
11 HTTP://HTTP.DEVELOPER.NVIDIA.COM/GPUGEMS/GPUGEMS_CH16.HTML

35. Soft Shadows of Spheres

Figure 58 Shadows are not only important to understand the geometry of a scene (e.g. the distances between objects); they can also be quite beautiful.

This tutorial covers **soft shadows of spheres**.

It is one of several tutorials about lighting that go beyond the Phong reflection model, which is a local illumination model and therefore doesn't take shadows into account. The presented technique renders the soft shadow of a single sphere on any mesh and is somewhat related to a technique that was proposed by Orion Sky Lawlor (see the "FURTHER READING" SECTION[1]). The shader can be extended to render the shadows of a small number of spheres at the cost of rendering performance; however, it cannot easily be applied to any other kind of shadow caster. Potential applications are computer ball games (where the ball is often the only object that requires a soft shadow and the only object that should cast a dynamic shadow on all other objects), computer games with a spherical main character (e.g. "Marble Madness"), visualizations that consist only of spheres (e.g. planetary visualizations,

1 Chapter 35.0.197 on page 344

ball models of small nuclei, atoms, or molecules, etc.), or test scenes that can be populated with spheres and benefit from soft shadows.

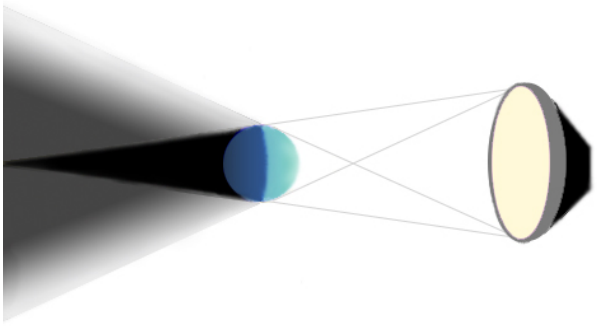

Figure 59 Umbra (black) and penumbra (gray) are the main parts of soft shadows.

Figure 60 "The crowning with thorns" by Caravaggio (ca. 1602). Note the shadow line in the upper left corner, which becomes softer with increasing distance from the shadow casting wall.

35.0.192. Soft Shadows

While directional light sources and point light sources produce hard shadows, any area light source generates a soft shadow. This is also true for all real light sources,

in particular the sun and any light bulb or lamp. From some points behind the shadow caster, no part of the light source is visible and the shadow is uniformly dark: this is the umbra. From other points, more or less of the light source is visible and the shadow is therefore less or more complete: this is the penumbra. Finally, there are points from where the whole area of the light source is visible: these points are outside of the shadow.

In many cases, the softness of a shadow depends mainly on the distance between the shadow caster and the shadow receiver: the larger the distance, the softer the shadow. This is a well known effect in art; see for example the painting by Caravaggio below .

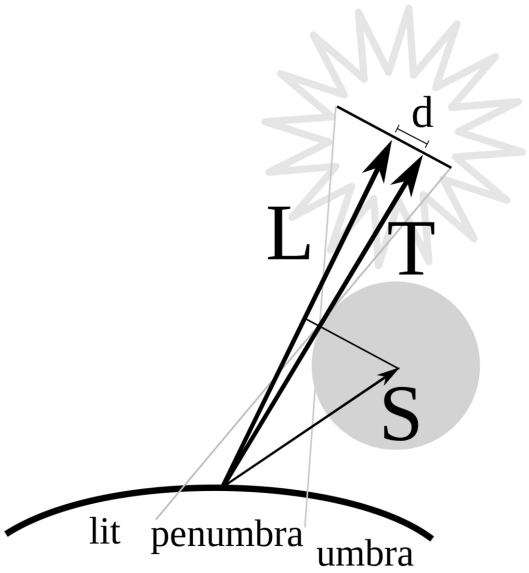

Figure 61 Vectors for the computation of soft shadows: vector L to the light source, vector S to the center of the sphere, tangent vector T, and distance d of the tangent from the center of the light source.

35.0.193. Computation

We are going to approximately compute the shadow of a point on a surface when a sphere of radius r_{sphere} at **S** (relative to the surface point) is occluding a spherical

light source of radius r_{light} at $\mathbf{L}$ (again relative to the surface point); see the figure below .

To this end, we consider a tangent in direction $\mathbf{T}$ to the sphere and passing through the surface point. Furthermore, this tangent is chosen to be in the plane spanned by $\mathbf{L}$ and $\mathbf{S}$, i.e. parallel to the view plane of the figure below . The crucial observation is that the minimum distance d of the center of the light source and this tangent line is directly related to the amount of shadowing of the surface point because it determines how large the area of the light source is that is visible from the surface point. More precisely spoken, we require a signed distance (positive if the tangent is on the same side of $\mathbf{L}$ as the sphere, negative otherwise) to determine whether the surface point is in the umbra ($d < -r_{\text{light}}$), in the penumbra ($-r_{\text{light}} < d < r_{\text{light}}$), or outside of the shadow ($r_{\text{light}} < d$).

For the computation of d, we consider the angles between $\mathbf{L}$ and $\mathbf{S}$ and between $\mathbf{T}$ and $\mathbf{S}$. The difference between these two angles is the angle between $\mathbf{L}$ and $\mathbf{T}$, which is related to d by:

$\measuredangle(\mathbf{L},\mathbf{T}) \approx \sin\measuredangle(\mathbf{L},\mathbf{T}) = \frac{d}{|\mathbf{L}|}$.

Thus, so far we have:

$d \approx |\mathbf{L}|\measuredangle(\mathbf{L},\mathbf{T}) \quad = |\mathbf{L}|\left(\measuredangle(\mathbf{L},\mathbf{S}) - \measuredangle(\mathbf{T},\mathbf{S})\right)$

We can compute the angle between $\mathbf{T}$ and $\mathbf{S}$ using

$\sin\measuredangle(\mathbf{T},\mathbf{S}) = \frac{r_{\text{sphere}}}{|\mathbf{S}|}$.

Thus:

$\measuredangle(\mathbf{T},\mathbf{S}) = \arcsin\frac{r_{\text{sphere}}}{|\mathbf{S}|}$.

For the angle between $\mathbf{L}$ and $\mathbf{S}$ we use a feature of the cross product:

$|\mathbf{a}\times\mathbf{b}| = |\mathbf{a}|\ |\mathbf{b}|\ \sin\measuredangle(\mathbf{a},\mathbf{b})$.

Therefore:

$\measuredangle(\mathbf{L},\mathbf{S}) = \arcsin\frac{|\mathbf{L}\times\mathbf{S}|}{|\mathbf{L}||\mathbf{S}|}$.

All in all we have:

$d \approx |\mathbf{L}|\left(\arcsin\frac{|\mathbf{L}\times\mathbf{S}|}{|\mathbf{L}||\mathbf{S}|} - \arcsin\frac{r_{\text{sphere}}}{|\mathbf{S}|}\right)$

The approximation we did so far, doesn't matter much; more importantly it doesn't produce rendering artifacts. If performance is an issue one could go further and use $\arcsin(x) \approx x$; i.e., one could use:

$$d \approx |\mathbf{L}| \left(\frac{|\mathbf{L} \times \mathbf{S}|}{|\mathbf{L}||\mathbf{S}|} - \frac{r_{\text{sphere}}}{|\mathbf{S}|} \right)$$

This avoids all trigonometric functions; however, it does introduce rendering artifacts (in particular if a specular highlight is in the penumbra that is facing the light source). Whether these rendering artifacts are worth the gains in performance has to be decided for each case.

Next we look at how to compute the level of shadowing w based on d. As d decreases from r_{light} to $-r_{\text{light}}$, w should increase from 0 to 1. In other words, we want a smooth step from 0 to 1 between values -1 and 1 of $-d/r_{\text{light}}$. Probably the most efficient way to achieve this is to use the Hermite interpolation offered by the built-in GLSL function `smoothstep(a,b,x) = t*t*(3-2*t)` with `t=clamp((x-a)/(b-a),0,1)`:

$$w = \text{smoothstep}\left(-1, 1, \frac{-d}{r_{\text{light}}}\right)$$

While this isn't a particular good approximation of a physically-based relation between w and d, it still gets the essential features right.

Furthermore, w should be 0 if the light direction $\mathbf{L}$ is in the opposite direction of $\mathbf{S}$; i.e., if their dot product is negative. This condition turns out to be a bit tricky since it leads to a noticeable discontinuity on the plane where $\mathbf{L}$ and $\mathbf{S}$ are orthogonal. To soften this discontinuity, we can again use `smoothstep` to compute an improved value w':

$$w' = w\,\text{smoothstep}\left(0.0, 0.2, \frac{\mathbf{L} \cdot \mathbf{S}}{|\mathbf{L}||\mathbf{S}|}\right)$$

Additionally, we have to set w' to 0 if a point light source is closer to the surface point than the occluding sphere. This is also somewhat tricky because the spherical light source can intersect the shadow-casting sphere. One solution that avoids too obvious artifacts (but fails to deal with the full intersection problem) is:

$$w'' = w'\,\text{smoothstep}\left(0, r_{\text{sphere}}, |\mathbf{L}| - |\mathbf{S}|\right)$$

In the case of a directional light source we just set $w'' = w'$. Then the term $(1 - w'')$, which specifies the level of unshadowed lighting, should be multiplied to any illumination by the light source. (Thus, ambient light shouldn't be multiplied with this factor.) If the shadows of multiple shadow casters are computed, the terms $(1 - w'')$ for all shadow casters have to be combined for each light source. The common way is to multiply them although this can be inaccurate (in particular if the umbras overlap).

35.0.194. Implementation

The implementation computes the length of the `lightDirection` and `sphereDirection` vectors and then proceeds with the normalized vectors. This way, the lengths of these vectors have to be computed only once and we even avoid some divisions because we can use normalized vectors. Here is the crucial part of the fragment shader:

```
// computation of level of shadowing w
vec3 sphereDirection = vec3(_SpherePosition - position);
float sphereDistance = length(sphereDirection);
sphereDirection = sphereDirection / sphereDistance;
float d = lightDistance
   * (asin(min(1.0,
   length(cross(lightDirection, sphereDirection))))
   - asin(min(1.0, _SphereRadius / sphereDistance)));
float w = smoothstep(-1.0, 1.0, -d / _LightSourceRadius);
w = w * smoothstep(0.0, 0.2,
   dot(lightDirection, sphereDirection));
if (0.0 != _WorldSpaceLightPos0.w) // point light source?
{
   w = w * smoothstep(0.0, _SphereRadius,
      lightDistance - sphereDistance);
}
```

The use of `asin(min(1.0, ...))` makes sure that the argument of `asin` is in the allowed range.

35.0.195. Complete Shader Code

The complete source code defines properties for the shadow-casting sphere and the light source radius. All values are expected to be in world coordinates. For directional light sources, the light source radius should be given in radians (1 rad = 180° / π). The best way to set the position and radius of the shadow-casting sphere is a short script that should be attached to all shadow-receiving objects that use the shader, for example:

```
@script ExecuteInEditMode()

var occluder : GameObject;

function Update () {
   if (null != occluder) {
      renderer.sharedMaterial.SetVector("_SpherePosition",
         occluder.transform.position);
      renderer.sharedMaterial.SetFloat("_SphereRadius",
         occluder.transform.localScale.x / 2.0);
```

```
   }
}
```

This script has a public variable `occluder` that should be set to the shadow-casting sphere. Then it sets the properties `_SpherePostion` and `_SphereRadius` of the following shader (which should be attached to the same shadow-receiving object as the script).

```
Shader "GLSL shadow of sphere" {
   Properties {
      _Color ("Diffuse Material Color", Color) = (1,1,1,1)
      _SpecColor ("Specular Material Color", Color) = (1,1,1,1)
      _Shininess ("Shininess", Float) = 10
      _SpherePosition ("Sphere Position", Vector) = (0,0,0,1)
      _SphereRadius ("Sphere Radius", Float) = 1
      _LightSourceRadius ("Light Source Radius", Float) = 0.005
   }
   SubShader {
      Pass {
         Tags { "LightMode" = "ForwardBase" }
            // pass for ambient light and first light source

         GLSLPROGRAM

         // User-specified properties
         uniform vec4 _Color;
         uniform vec4 _SpecColor;
         uniform float _Shininess;
         uniform vec4 _SpherePosition;
            // center of shadow-casting sphere in world coordinates
         uniform float _SphereRadius;
            // radius of shadow-casting sphere
         uniform float _LightSourceRadius;
            // in radians for directional light sources

         // The following built-in uniforms (except _LightColor0)
         // are also defined in "UnityCG.glslinc",
         // i.e. one could #include "UnityCG.glslinc"
         uniform vec3 _WorldSpaceCameraPos;
            // camera position in world space
         uniform mat4 _Object2World; // model matrix
         uniform mat4 _World2Object; // inverse model matrix
         uniform vec4 _WorldSpaceLightPos0;
            // direction to or position of light source
         uniform vec4 _LightColor0;
            // color of light source (from "Lighting.cginc")

         varying vec4 position;
            // position of the vertex (and fragment) in world space
         varying vec3 varyingNormalDirection;
            // surface normal vector in world space

         #ifdef VERTEX

         void main()
         {
```

```
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   position = modelMatrix * gl_Vertex;
   varyingNormalDirection = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
   vec3 normalDirection = normalize(varyingNormalDirection);

   vec3 viewDirection =
      normalize(_WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection;
   float lightDistance;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
      lightDistance = 1.0;
   }
   else // point or spot light
   {
      lightDirection = vec3(_WorldSpaceLightPos0 - position);
      lightDistance = length(lightDirection);
      attenuation = 1.0 / lightDistance; // linear attenuation
      lightDirection = lightDirection / lightDistance;
   }

   // computation of level of shadowing w
   vec3 sphereDirection = vec3(_SpherePosition - position);
   float sphereDistance = length(sphereDirection);
   sphereDirection = sphereDirection / sphereDistance;
   float d = lightDistance
      * (asin(min(1.0,
      length(cross(lightDirection, sphereDirection))))
      - asin(min(1.0, _SphereRadius / sphereDistance)));
   float w = smoothstep(-1.0, 1.0, -d / _LightSourceRadius);
   w = w * smoothstep(0.0, 0.2,
      dot(lightDirection, sphereDirection));
   if (0.0 != _WorldSpaceLightPos0.w) // point light source?
   {
      w = w * smoothstep(0.0, _SphereRadius,
         lightDistance - sphereDistance);
   }

   vec3 ambientLighting =
      vec3(gl_LightModel.ambient) * vec3(_Color);
```

```
      vec3 diffuseReflection =
         attenuation * vec3(_LightColor0) * vec3(_Color)
         * max(0.0, dot(normalDirection, lightDirection));

      vec3 specularReflection;
      if (dot(normalDirection, lightDirection) < 0.0)
         // light source on the wrong side?
      {
         specularReflection = vec3(0.0, 0.0, 0.0);
            // no specular reflection
      }
      else // light source on the right side
      {
         specularReflection = attenuation * vec3(_LightColor0)
            * vec3(_SpecColor) * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _Shininess);
      }

      gl_FragColor = vec4(ambientLighting
         + (1.0 - w) * (diffuseReflection + specularReflection),
         1.0);
   }

   #endif

   ENDGLSL
}

Pass {
   Tags { "LightMode" = "ForwardAdd" }
      // pass for additional light sources
   Blend One One // additive blending

   GLSLPROGRAM

   // User-specified properties
   uniform vec4 _Color;
   uniform vec4 _SpecColor;
   uniform float _Shininess;
   uniform vec4 _SpherePosition;
      // center of shadow-casting sphere in world coordinates
   uniform float _SphereRadius;
      // radius of shadow-casting sphere
   uniform float _LightSourceRadius;
      // in radians for directional light sources

   // The following built-in uniforms (except _LightColor0)
   // are also defined in "UnityCG.glslinc",
   // i.e. one could #include "UnityCG.glslinc"
   uniform vec3 _WorldSpaceCameraPos;
      // camera position in world space
   uniform mat4 _Object2World; // model matrix
   uniform mat4 _World2Object; // inverse model matrix
   uniform vec4 _WorldSpaceLightPos0;
      // direction to or position of light source
   uniform vec4 _LightColor0;
```

```
   // color of light source (from "Lighting.cginc")

varying vec4 position;
   // position of the vertex (and fragment) in world space
varying vec3 varyingNormalDirection;
   // surface normal vector in world space

#ifdef VERTEX

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   position = modelMatrix * gl_Vertex;
   varyingNormalDirection = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
   vec3 normalDirection = normalize(varyingNormalDirection);

   vec3 viewDirection =
      normalize(_WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection;
   float lightDistance;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
      lightDistance = 1.0;
   }
   else // point or spot light
   {
      lightDirection = vec3(_WorldSpaceLightPos0 - position);
      lightDistance = length(lightDirection);
      attenuation = 1.0 / lightDistance; // linear attenuation
      lightDirection = lightDirection / lightDistance;
   }

   // computation of level of shadowing w
   vec3 sphereDirection = vec3(_SpherePosition - position);
   float sphereDistance = length(sphereDirection);
   sphereDirection = sphereDirection / sphereDistance;
   float d = lightDistance
      * (asin(min(1.0,
      length(cross(lightDirection, sphereDirection))))
      - asin(min(1.0, _SphereRadius / sphereDistance)));
```

```
            float w = smoothstep(-1.0, 1.0, -d / _LightSourceRadius);
            w = w * smoothstep(0.0, 0.2,
               dot(lightDirection, sphereDirection));
            if (0.0 != _WorldSpaceLightPos0.w) // point light source?
            {
               w = w * smoothstep(0.0, _SphereRadius,
                  lightDistance - sphereDistance);
            }

            vec3 diffuseReflection =
               attenuation * vec3(_LightColor0) * vec3(_Color)
               * max(0.0, dot(normalDirection, lightDirection));

            vec3 specularReflection;
            if (dot(normalDirection, lightDirection) < 0.0)
               // light source on the wrong side?
            {
               specularReflection = vec3(0.0, 0.0, 0.0);
                  // no specular reflection
            }
            else // light source on the right side
            {
               specularReflection = attenuation * vec3(_LightColor0)
                  * vec3(_SpecColor) * pow(max(0.0, dot(
                  reflect(-lightDirection, normalDirection),
                  viewDirection)), _Shininess);
            }

            gl_FragColor = vec4((1.0 - w) * (diffuseReflection
               + specularReflection), 1.0);
         }

         #endif

         ENDGLSL
      }
   }
   // The definition of a fallback shader should be commented out
   // during development:
   // Fallback "Specular"
}
```

35.0.196. Summary

Congratulations! I hope you succeeded to render some nice soft shadows. We have looked at:

- What soft shadows are and what the penumbra and umbra is.
- How to compute soft shadows of spheres.
- How to implement the computation, including a script in JavaScript that sets some properties based on another `GameObject`.

35.0.197. Further Reading

If you still want to know more

- about the rest of the shader code, you should read GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[2].
- about computations of soft shadows, you should read a publication by Orion Sky Lawlor: "Interpolation-Friendly Soft Shadow Maps" in Proceedings of Computer Graphics and Virtual Reality '06, pages 111–117. A preprint is available ONLINE[3].

2 Chapter 13 on page 111

3 HTTP://WWW.CS.UAF.EDU/~OLAWLOR/PAPERS/2006/SHADOW/LAWLOR_SHADOW_2006.PDF

36. Toon Shading

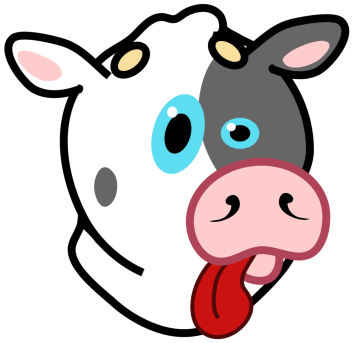

Figure 62 A cow smiley. All images in this section are by Mariana Ruiz Villarreal (A.K.A. LADYOFHATS)[a].

a HTTP://EN.COMMONS.ORG/WIKI/USER%3ALADYOFHATS

This tutorial covers **toon shading** (also known as **cel shading**) as an example of **non-photorealistic rendering** techniques.

It is one of several tutorials about lighting that go beyond the Phong reflection model. However, it is based on per-pixel lighting with the Phong reflection model as described in GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[1]. If you haven't read that tutorial yet, you should read it first.

Non-photorealistic rendering is a very broad term in computer graphics that covers all rendering techniques and visual styles that are obviously and deliberately different from the appearance of photographs of physical objects. Examples include hatching, outlining, distortions of linear perspective, coarse dithering, coarse color quantization, etc.

1 Chapter 13 on page 111

Toon shading (or **cel shading**) is any subset of non-photorealistic rendering techniques that is used to achieve a cartoonish or hand-drawn appearance of three-dimensional models.

Figure 63 A goat smiley.

36.0.198. Shaders for a Specific Visual Style

John Lasseter from Pixar once said in an interview: “Art challenges technology, and technology inspires the art.” Many visual styles and drawing techniques that are traditionally used to depict three-dimensional objects are in fact very difficult to implement in shaders. However, there is no fundamental reason not to try it.

When implementing one or more shaders for any specific visual style, one should first determine which features of the style have to be implemented. This is mainly a task of precise analysis of examples of the visual style. Without such examples, it is usually unlikely that the characteristic features of a style can be determined. Even artists who master a certain style are unlikely to be able to describe these features appropriately; for example, because they are no longer aware of certain features or might consider some characteristic features as unnecessary imperfections that are not worth mentioning.

For each of the features it should then be determined whether and how accurately to implement them. Some features are rather easy to implement, others are very difficult to implement by a programmer or to compute by a GPU. Therefore, a discussion between shader programmers and (technical) artists in the spirit of John Lasseter's quote above is often extremely worthwhile to decide which features to include and how accurately to reproduce them.

Figure 64 A bull smiley.

36.0.199. Stylized Specular Highlights

In comparison to the Phong reflection model that was implemented in GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[2], the specular highlights in the images in this section are plainly white without any addition of other colors. Furthermore, they have a very sharp boundary.

We can implement this kind of stylized specular highlights by computing the specular reflection term of the Phong shading model and setting the fragment color to the specular reflection color times the (unattenuated) color of the light source if the specular reflection term is greater than a certain threshold, e.g. half the maximum intensity.

But what if there shouldn't been any highlights? Usually, the user would specify a black specular reflection color for this case; however, with our method this results in black highlights. One way to solve this problem is to take the opacity of the specular reflection color into account and "blend" the color of the highlight over other colors by compositing them based on the opacity of the specular color. Alpha blending as a PER-FRAGMENT OPERATION[3] was described in GLSL PROGRAMMING/UNITY/TRANSPARENCY[4]. However, if all colors are known in a fragment shader, it can also be computed within a fragment shader.

In the following code snippet, `fragmentColor` is assumed to have already a color assigned, e.g. based on diffuse illumination. The specular color `_SpecColor`

2 Chapter 13 on page 111
3 HTTP://EN.WIKIBOOKS.ORG/WIKI/%2FPER-FRAGMENT%20OPERATIONS
4 Chapter 7 on page 49

times the light source color `_LightColor0` is then blended over `fragment_Color` based on the opacity of the specular color `_SpecColor.a`:

```
if (dot(normalDirection, lightDirection) > 0.0
   // light source on the right side?
   && attenuation *  pow(max(0.0, dot(
   reflect(-lightDirection, normalDirection),
   viewDirection)), _Shininess) > 0.5)
   // more than half highlight intensity?
{
   fragmentColor = _SpecColor.a
      * vec3(_LightColor0) * vec3(_SpecColor)
      + (1.0 - _SpecColor.a) * fragmentColor;
}
```

Is this sufficient? If you look closely at the eyes of the bull in the image above , you will see two pairs of specular highlights, i.e. there is more than one light source that causes specular highlights. In most tutorials, we have taken additional light sources into account by a second render pass with additive blending. However, if the color of specular highlights should not be added to other colors then additive blending should not be used. Instead, alpha blending with a (usually) opaque color for the specular highlights and transparent fragments for other fragments would be a feasible solution. (See GLSL PROGRAMMING/UNITY/TRANSPARENCY[5] for a description of alpha blending.)

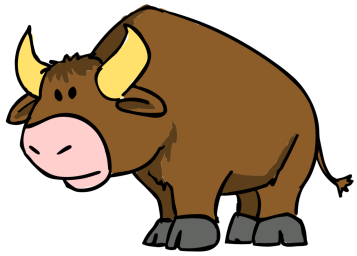

Figure 65 A cartoonish bull.

36.0.200. Stylized Diffuse Illumination

The diffuse illumination in the image of the bull below consists of just two colors: a light brown for lit fur and a dark brown for unlit fur. The color of other parts of the bull is independent of the lighting.

5 Chapter 7 on page 49

One way to implement this, is to use the full diffuse reflection color whenever the diffuse reflection term of the Phong reflection model reaches a certain threshold, e.g. greater than 0, and a second color otherwise. For the fur of the bull, these two colors would be different, for the other parts, they would be the same such that there is no visual difference between lit and unlit areas. An implementation for a threshold `_DiffuseThreshold` to switch from the darker color `_UnlitColor` to the lighter color `_Color` (multiplied with the light source color `_LightColor0`) could look like this:

```
vec3 fragmentColor = vec3(_UnlitColor);

if (attenuation
   * max(0.0, dot(normalDirection, lightDirection))
   >= _DiffuseThreshold)
{
   fragmentColor = vec3(_LightColor0) * vec3(_Color);
}
```

Is this all there is to say about the stylized diffuse illumination in the image below ? A really close look reveals that there is a light, irregular line between the dark brown and the light brown. In fact, the situation is even more complicated and the dark brown sometimes doesn't cover all areas that would be covered by the technique described above, and sometimes it covers more than that and even goes beyond the black outline. This adds rich detail to the visual style and creates a hand-drawn appearance. On the other hand, it is very difficult to reproduce convincingly in a shader.

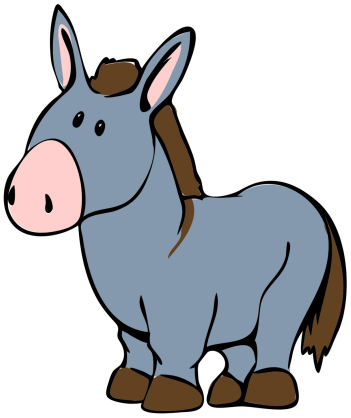

Figure 66 A cartoonish donkey.

36.0.201. Outlines

One of the characteristic features of many toon shaders are outlines in a specific color along the silhouettes of the model (usually black, but also other colors, see the cow above for an example).

There are various techniques to achieve this effect in a shader. Unity 3.3 is shipped with a toon shader in the standard assets that renders these outlines by rendering the back faces of an enlarged model in the color of the outlines (enlarged by moving the vertex positions in the direction of the surface normal vectors) and then rendering the front faces on top of them. Here we use another technique based on GLSL PROGRAMMING/UNITY/SILHOUETTE ENHANCEMENT[6]: if a fragment is determined to be close enough to a silhouette, it is set to the color of the outline. This works only for smooth surfaces, and it will generate outlines of varying thickness (which is a plus or a minus depending on the visual style). However, at least the overall thickness of the outlines should be controllable by a shader property.

Are we done yet? If you have a close look at the donkey, you will see that the outlines at its belly and in the ears are considerably thicker than other outlines. This conveys unlit areas; however, the change in thickness is continuous. One way to simulate this effect would be to let the user specify two overall outline thicknesses: one for fully lit areas and one for unlit areas (according to the diffuse reflection term of the Phong reflection model). In between these extremes, the thickness parameter could be interpolated (again according to the diffuse reflection term). This, however, makes the outlines dependent on a specific light source; therefore, the shader below renders outlines and diffuse illumination only for the first light source, which should usually be the most important one. All other light sources only render specular highlights.

The following implementation uses the `mix` instruction to interpolate between the `_UnlitOutlineThickness` (if the dot product of the diffuse reflection term is less or equal 0) and `_LitOutlineThickness` (if the dot product is 1). This interpolated value is then used as a threshold to determine whether a point is close enough to the silhouette. If it is, the fragment color is set to the color of the outline `_OutlineColor`:

```
if (dot(viewDirection, normalDirection)
   < mix(_UnlitOutlineThickness, _LitOutlineThickness,
   max(0.0, dot(normalDirection, lightDirection))))
{
```

6 Chapter 9 on page 67

```
            fragmentColor =
               vec3(_LightColor0) * vec3(_OutlineColor);
         }
```

36.0.202. Complete Shader Code

It should be clear by now that even the few images above pose some really difficult challenges for a faithful implementation. Thus, the shader below only implements a few characteristics as described above and ignores many others. Note that the different color contributions (diffuse illumination, outlines, highlights) are given different priorities according to which should occlude which. You could also think of these priorities as different layers that are put on top of each other.

```
Shader "GLSL shader for toon shading" {
   Properties {
      _Color ("Diffuse Color", Color) = (1,1,1,1)
      _UnlitColor ("Unlit Diffuse Color", Color) = (0.5,0.5,0.5,1)
      _DiffuseThreshold ("Threshold for Diffuse Colors", Range(0,1))
         = 0.1
      _OutlineColor ("Outline Color", Color) = (0,0,0,1)
      _LitOutlineThickness ("Lit Outline Thickness", Range(0,1)) = 0.1
      _UnlitOutlineThickness ("Unlit Outline Thickness", Range(0,1))
         = 0.4
      _SpecColor ("Specular Color", Color) = (1,1,1,1)
      _Shininess ("Shininess", Float) = 10
   }
   SubShader {
      Pass {
         Tags { "LightMode" = "ForwardBase" }
            // pass for ambient light and first light source

         GLSLPROGRAM

         // User-specified properties
         uniform vec4 _Color;
         uniform vec4 _UnlitColor;
         uniform float _DiffuseThreshold;
         uniform vec4 _OutlineColor;
         uniform float _LitOutlineThickness;
         uniform float _UnlitOutlineThickness;
         uniform vec4 _SpecColor;
         uniform float _Shininess;

         // The following built-in uniforms (except _LightColor0)
         // are also defined in "UnityCG.glslinc",
         // i.e. one could #include "UnityCG.glslinc"
         uniform vec3 _WorldSpaceCameraPos;
            // camera position in world space
         uniform mat4 _Object2World; // model matrix
         uniform mat4 _World2Object; // inverse model matrix
         uniform vec4 _WorldSpaceLightPos0;
            // direction to or position of light source
```

```
uniform vec4 _LightColor0;
   // color of light source (from "Lighting.cginc")

varying vec4 position;
   // position of the vertex (and fragment) in world space
varying vec3 varyingNormalDirection;
   // surface normal vector in world space

#ifdef VERTEX

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   position = modelMatrix * gl_Vertex;
   varyingNormalDirection = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
   vec3 normalDirection = normalize(varyingNormalDirection);

   vec3 viewDirection =
      normalize(_WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
   }
   else // point or spot light
   {
      vec3 vertexToLightSource =
         vec3(_WorldSpaceLightPos0 - position);
      float distance = length(vertexToLightSource);
      attenuation = 1.0 / distance; // linear attenuation
      lightDirection = normalize(vertexToLightSource);
   }

   // default: unlit
   vec3 fragmentColor = vec3(_UnlitColor);

   // low priority: diffuse illumination
   if (attenuation
      * max(0.0, dot(normalDirection, lightDirection))
      >= _DiffuseThreshold)
   {
```

```
            fragmentColor = vec3(_LightColor0) * vec3(_Color);
         }

         // higher priority: outline
         if (dot(viewDirection, normalDirection)
            < mix(_UnlitOutlineThickness, _LitOutlineThickness,
            max(0.0, dot(normalDirection, lightDirection))))
         {
            fragmentColor =
               vec3(_LightColor0) * vec3(_OutlineColor);
         }

         // highest priority: highlights
         if (dot(normalDirection, lightDirection) > 0.0
            // light source on the right side?
            && attenuation *  pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _Shininess) > 0.5)
            // more than half highlight intensity?
         {
            fragmentColor = _SpecColor.a
               * vec3(_LightColor0) * vec3(_SpecColor)
               + (1.0 - _SpecColor.a) * fragmentColor;
         }

         gl_FragColor = vec4(fragmentColor, 1.0);
      }

      #endif

      ENDGLSL
   }

   Pass {
      Tags { "LightMode" = "ForwardAdd" }
         // pass for additional light sources
      Blend SrcAlpha OneMinusSrcAlpha
         // blend specular highlights over framebuffer

      GLSLPROGRAM

      // User-specified properties
      uniform vec4 _Color;
      uniform vec4 _UnlitColor;
      uniform float _DiffuseThreshold;
      uniform vec4 _OutlineColor;
      uniform float _LitOutlineThickness;
      uniform float _UnlitOutlineThickness;
      uniform vec4 _SpecColor;
      uniform float _Shininess;

      // The following built-in uniforms (except _LightColor0)
      // are also defined in "UnityCG.glslinc",
      // i.e. one could #include "UnityCG.glslinc"
      uniform vec3 _WorldSpaceCameraPos;
         // camera position in world space
      uniform mat4 _Object2World; // model matrix
      uniform mat4 _World2Object; // inverse model matrix
```

```
uniform vec4 _WorldSpaceLightPos0;
   // direction to or position of light source
uniform vec4 _LightColor0;
   // color of light source (from "Lighting.cginc")

varying vec4 position;
   // position of the vertex (and fragment) in world space
varying vec3 varyingNormalDirection;
   // surface normal vector in world space

#ifdef VERTEX

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object; // unity_Scale.w
      // is unnecessary because we normalize vectors

   position = modelMatrix * gl_Vertex;
   varyingNormalDirection = normalize(vec3(
      vec4(gl_Normal, 0.0) * modelMatrixInverse));

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

#endif

#ifdef FRAGMENT

void main()
{
   vec3 normalDirection = normalize(varyingNormalDirection);

   vec3 viewDirection =
      normalize(_WorldSpaceCameraPos - vec3(position));
   vec3 lightDirection;
   float attenuation;

   if (0.0 == _WorldSpaceLightPos0.w) // directional light?
   {
      attenuation = 1.0; // no attenuation
      lightDirection = normalize(vec3(_WorldSpaceLightPos0));
   }
   else // point or spot light
   {
      vec3 vertexToLightSource =
         vec3(_WorldSpaceLightPos0 - position);
      float distance = length(vertexToLightSource);
      attenuation = 1.0 / distance; // linear attenuation
      lightDirection = normalize(vertexToLightSource);
   }

   vec4 fragmentColor = vec4(0.0, 0.0, 0.0, 0.0);
   if (dot(normalDirection, lightDirection) > 0.0
      // light source on the right side?
      && attenuation *  pow(max(0.0, dot(
      reflect(-lightDirection, normalDirection),
      viewDirection)), _Shininess) > 0.5)
```

```
            // more than half highlight intensity?
         {
            fragmentColor =
               vec4(_LightColor0.rgb, 1.0) * _SpecColor;
         }

         gl_FragColor = fragmentColor;
      }

      #endif

      ENDGLSL
   }
  }
  // The definition of a fallback shader should be commented out
  // during development:
  // Fallback "Specular"
}
```

One problem with this shader are the hard edges between colors, which often result in noticeable aliasing, in particular at the outlines. This could be alleviated by using the `smoothstep` function to provide a smoother transition.

36.0.203. Summary

Congratulations, you have reached the end of this tutorial. We have seen:

- What toon shading, cel shading, and non-photorealistic rendering are.
- How some of the non-photorealistic rendering techniques are used in toon shading.
- How to implement these techniques in a shader.

36.0.204. Further Reading

If you still want to know more

- about the Phong reflection model and the per-pixel lighting, you should read GLSL PROGRAMMING/UNITY/SMOOTH SPECULAR HIGHLIGHTS[7].
- about the computation of silhouettes, you should read GLSL PROGRAMMING/UNITY/SILHOUETTE ENHANCEMENT[8].
- about blending, you should read GLSL PROGRAMMING/UNITY/TRANSPARENCY[9].

7 Chapter 13 on page 111
8 Chapter 9 on page 67
9 Chapter 7 on page 49

- about non-photorealistic rendering techniques, you could read Chapter 18 of the book “OpenGL Shading Language” (3rd edition) by Randi Rost et al., published 2009 by Addison-Wesley.

Part VIII.

Non-Standard Vertex Transformations

37. Screen Overlays

Figure 67 Title screen of a movie from 1934.

This tutorial covers **screen overlays**, which are also known as "GUI Textures" in Unity.

It is the first tutorial of a series of tutorials on non-standard vertex transformations, which deviate from the standard vertex transformations that are described in GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[1]. This particular tutorial uses texturing as described in GLSL PROGRAMMING/UNITY/TEXTURED SPHERES[2] and blending as described in GLSL PROGRAMMING/UNITY/TRANSPARENCY[3].

37.0.205. Unity's GUI Textures

There are many applications for screen overlays (i.e. GUI textures in Unity's terminology), e.g. titles as in the image below , but also other GUI (graphical user

1 Chapter 43 on page 407
2 Chapter 16 on page 137
3 Chapter 7 on page 49

interface) elements such as buttons or status information. The common feature of these elements is that they should always appear on top of the scene and never be occluded by any other objects. Neither should these elements be affected by any of the camera movements. Thus, the vertex transformation should go directly from object space to screen space. Unity's GUI textures allow us to render this kind of elements by rendering a texture image at a specified position on the screen. This tutorial tries to reproduce the functionality of GUI textures with the help of shaders. Usually, you would still use GUI textures instead of such a shader; however, the shader allows for a lot more flexibility since you can adapt it in any way you want while GUI textures only offer a limited set of possibilities. (For example, you could change the shader such that the GPU spends less time on rasterizing the triangles that are occluded by an opaque GUI texture.)

37.0.206. Simulating GUI Textures with a GLSL Shader

The position of Unity's GUI textures is specified by an `X` and a `Y` coordinate of the lower, left corner of the rendered rectangle in pixels with $(0,0)$ at the center of the screen and a `Width` and `Height` of the rendered rectangle in pixels. To simulate GUI textures, we use similar shader properties:

```
Properties {
   _MainTex ("Texture", Rect) = "white" {}
   _Color ("Color", Color) = (1.0, 1.0, 1.0, 1.0)
   _X ("X", Float) = 0.0
   _Y ("Y", Float) = 0.0
   _Width ("Width", Float) = 128
   _Height ("Height", Float) = 128
}
```

and the corresponding uniforms

```
uniform sampler2D _MainTex;
uniform vec4 _Color;
uniform float _X;
uniform float _Y;
uniform float _Width;
uniform float _Height;
```

For the actual object, we could use a mesh that consists of just two triangles to form a rectangle. However, we can also just use the default cube object since back-face culling (and culling of triangles that are degenerated to edges) allows us to make sure that only two triangles of the cube are rasterized. The corners of the default cube object have coordinates -0.5 and $+0.5$ in object space, i.e., the lower, left corner of the rectangle is at $(-0.5,-0.5)$ and the upper, right corner is at $(+0.5,+0.5)$. To transform these coordinates to the user-specified coordinates in

screen space, we first transform them to raster positions in pixels where $(0,0)$ is at the lower, left corner of the screen:

```
uniform vec4 _ScreenParams; // x = width; y = height;
   // z = 1 + 1.0/width; w = 1 + 1.0/height
...
#ifdef VERTEX

void main()
{
    vec2 rasterPosition = vec2(
       _X + _ScreenParams.x / 2.0
       + _Width * (gl_Vertex.x + 0.5),
       _Y + _ScreenParams.y / 2.0
       + _Height * (gl_Vertex.y + 0.5));
    ...
```

This transformation transforms the lower, left corner of the front face of our cube from $(-0.5,-0.5)$ in object space to the raster position `vec2(_X + _ScreenParams.x / 2.0, _Y + _ScreenParams.y / 2.0)`, where `_ScreenParams.x` is the screen width in pixels and `_ScreenParams.y` is the height in pixels. The upper, right corner is transformed from $(+0.5,+0.5)$ to `vec2(_X + _ScreenParams.x / 2.0 + _Width, _Y + _ScreenParams.y / 2.0 + _Height)`. Raster positions are convenient and, in fact, they are often used in OpenGL; however, they are not quite what we need here.

The output of the vertex shader in `gl_Position` is in the so-called "clip space" as discussed in GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[4]. The GPU transforms these coordinates to normalized device coordinates between -1 and 1 by dividing them by the fourth coordinate `gl_Position.w` in the perspective division. If we set this fourth coordinate to 1, this division doesn't change anything; thus, we can think of the first three coordinates of `gl_Position` as coordinates in normalized device coordinates, where $(-1,-1,-1)$ specifies the lower, left corner of the screen on the near plane and $(1,1,-1)$ specifies the upper, right corner on the near plane. (We should use the near plane to make sure that the rectangle is in front of everything else.) In order to specify any screen position in `gl_Position`, we have to specify it in this coordinate system. Fortunately, transforming a raster position to normalized device coordinates is not too difficult:

```
gl_Position = vec4(
   2.0 * rasterPosition.x / _ScreenParams.x - 1.0,
   2.0 * rasterPosition.y / _ScreenParams.y - 1.0,
   -1.0, // near plane
```

4 Chapter 43 on page 407

```
            1.0 // all coordinates are divided by this coordinate
            );
```

As you can easily check, this transforms the raster position `vec2(0,0)` to normalized device coordinates $(-1.0, -1.0)$ and the raster position `vec2(_ScreenParams.x, _ScreenParams.y)` to $(1.0, 1.0)$, which is exactly what we need.

This is all we need for the vertex transformation from object space to screen space. However, we still need to compute appropriate texture coordinates in order to look up the texture image at the correct position. Texture coordinates should be between 0.0 and 1.0, which is actually easy to compute from the vertex coordinates in object space between -0.5 and $+0.5$:

```
textureCoords =
   vec4(gl_Vertex.x + 0.5, gl_Vertex.y + 0.5, 0.0, 0.0);
   // for a cube, gl_Vertex.x and gl_Vertex.y
   // are -0.5 or 0.5
```

With the varying variable `textureCoords`, we can then use a simple fragment program to look up the color in the texture image and modulate it with the user-specified color `_Color`:

```
#ifdef FRAGMENT

void main()
{
   gl_FragColor =
      _Color * texture2D (_MainTex, vec2(textureCoords));
}

#endif
```

That's it.

37.0.207. Complete Shader Code

If we put all the pieces together, we get the following shader, which uses the `Overlay` queue to render the object after everything else, and uses alpha blending (see GLSL PROGRAMMING/UNITY/TRANSPARENCY[5]) to allow for transparent textures. It also deactivates the depth test to make sure that the texture is never occluded:

```
Shader "GLSL shader for screen overlays" {
```

5 Chapter 7 on page 49

```
Properties {
   _MainTex ("Texture", Rect) = "white" {}
   _Color ("Color", Color) = (1.0, 1.0, 1.0, 1.0)
   _X ("X", Float) = 0.0
   _Y ("Y", Float) = 0.0
   _Width ("Width", Float) = 128
   _Height ("Height", Float) = 128
}
SubShader {
   Tags { "Queue" = "Overlay" } // render after everything else

   Pass {
      Blend SrcAlpha OneMinusSrcAlpha // use alpha blending
      ZTest Always // deactivate depth test

      GLSLPROGRAM

      // User-specified uniforms
      uniform sampler2D _MainTex;
      uniform vec4 _Color;
      uniform float _X;
      uniform float _Y;
      uniform float _Width;
      uniform float _Height;

      // The following built-in uniforms
      // are also defined in "UnityCG.glslinc",
      // i.e. one could #include "UnityCG.glslinc"
      uniform vec4 _ScreenParams; // x = width; y = height;
         // z = 1 + 1.0/width; w = 1 + 1.0/height

      // Varyings
      varying vec4 textureCoords;

      #ifdef VERTEX


      void main()
      {
         vec2 rasterPosition = vec2(
            _X + _ScreenParams.x / 2.0
            + _Width * (gl_Vertex.x + 0.5),
            _Y + _ScreenParams.y / 2.0
            + _Height * (gl_Vertex.y + 0.5));
         gl_Position = vec4(
            2.0 * rasterPosition.x / _ScreenParams.x - 1.0,
            2.0 * rasterPosition.y / _ScreenParams.y - 1.0,
            -1.0, // near plane is -1.0
            1.0);

         textureCoords =
            vec4(gl_Vertex.x + 0.5, gl_Vertex.y + 0.5, 0.0, 0.0);
            // for a cube, gl_Vertex.x and gl_Vertex.y
            // are -0.5 or 0.5
      }

      #endif
```

```
            #ifdef FRAGMENT

            void main()
            {
               gl_FragColor =
                  _Color * texture2D (_MainTex, vec2(textureCoords));
            }

            #endif

            ENDGLSL
         }
      }
}
```

When you use this shader for a cube object, the texture image can appear and disappear depending on the orientation of the camera. This is due to clipping by Unity, which doesn't render objects that are completely outside of the region of the scene that is visible in the camera (the view frustum). This clipping is based on the conventional transformation of game objects, which doesn't make sense for our shader. In order to deactivate this clipping, we can simply make the cube object a child of the camera (by dragging it over the camera in the **Hierarchy View**). If the cube object is then placed in front of the camera, it will always stay in the same relative position, and thus it won't be clipped by Unity. (At least not in the game view.)

37.0.208. Changes for Opaque Screen Overlays

Many changes to the shader are conceivable, e.g. a different blend mode or a different depth to have a few objects of the 3D scene in front of the overlay. Here we will only look at opaque overlays.

An opaque screen overlay will occlude triangles of the scene. If the GPU was aware of this occlusion, it wouldn't have to rasterize these occluded triangles (e.g. by using deferred rendering or early depth tests). In order to make sure that the GPU has any chance to apply these optimizations, we have to render the screen overlay first, by setting

```
Tags { "Queue" = "Background" }
```

Also, we should avoid blending by removing the `Blend` instruction. With these changes, opaque screen overlays are likely to improve performance instead of costing rasterization performance.

37.0.209. Summary

Congratulation, you have reached the end of another tutorial. We have seen:

- How to simulate GUI textures with a GLSL shader.
- How to modify the shader for opaque screen overlays.

37.0.210. Further Reading

If you still want to know more

- about texturing, you should read GLSL PROGRAMMING/UNITY/TEXTURED SPHERES[6].
- about blending, you should read GLSL PROGRAMMING/UNITY/TRANSPARENCY[7].
- about object space, screen space, clip space, normalized device coordinates, perspective division, etc., you should read the description of the standard vertex transformations in GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[8].

6 Chapter 16 on page 137
7 Chapter 7 on page 49
8 Chapter 43 on page 407

38. Billboards

Figure 68 Billboards along a highway. Note the orientation of the billboards for best visibility.

This tutorial introduces **billboards**.

It is based on GLSL PROGRAMMING/UNITY/TEXTURED SPHERES[1] and the discussion in GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[2].

38.0.211. Billboards

In computer graphics, billboards are textured rectangles that are transformed such that they always appear parallel to the view plane. Thus, they are similar to billboards along highways in that they are rotated for best visibility. However, they are different from highway billboards since they are dynamically rotated to always offer best visibility.

The main use of billboards is to replace complex three-dimensional models (e.g. grass, bushes, or even trees) by two-dimensional images. In fact, Unity also uses billboards to render grass. Moreover, billboards are often used to render

1 Chapter 16 on page 137
2 Chapter 43 on page 407

two-dimensional sprites. In both applications, it is crucial that the billboard is always aligned parallel to the view plane in order to keep up the illusion of a three-dimensional shape although only a two-dimensional image is rendered.

38.0.212. Vertex Transformation for Billboards

Similarly to GLSL PROGRAMMING/UNITY/SKYBOXES[3], we are going to use the default cube object to render a billboard. The basic idea is to transform only the origin $(0,0,0,1)$ of the object space to view space with the standard transformation `gl_ModelViewMatrix`. (In homogeneous coordinates all points have a 1 as fourth coordinate; see the discussion in GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[4].) View space is just a rotated version of world space with the *xy* plane parallel to the view plane as discussed in GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[5]. Thus, this is the correct space to construct an appropriately rotated billboard. We add the *x* and *y* object coordinates (`gl_Vertex.x` and `gl_Vertex.y`) to the transformed origin in view coordinates and then transform the result with the projection matrix `gl_ProjectionMatrix`:

```
gl_Position = gl_ProjectionMatrix * (gl_-
ModelViewMatrix * vec4(0.0, 0.0, 0.0, 1.0) + vec4(gl_-
Vertex.x, gl_Vertex.y, 0.0, 0.0));
```

Apart from this, we only have to compute texture coordinates, which is done the same way as in GLSL PROGRAMMING/UNITY/SKYBOXES[6]:

```
textureCoords = vec4(gl_Vertex.x + 0.5, gl_Vertex.y +
0.5, 0.0, 0.0);
```

Then the fragment shader just looks up the color at the interpolated texture coordinates.

38.0.213. Complete Shader Code

The complete shader code for the standard cube object is now:

```
Shader "GLSL shader for billboards" {
   Properties {
```

3 Chapter 28 on page 253
4 Chapter 43 on page 407
5 Chapter 43 on page 407
6 Chapter 28 on page 253

```
      _MainTex ("Texture Image", 2D) = "white" {}
   }
   SubShader {
      Pass {
         GLSLPROGRAM

         // User-specified uniforms
         uniform sampler2D _MainTex;

         // Varyings
         varying vec4 textureCoords;

         #ifdef VERTEX

         void main()
         {
            gl_Position = gl_ProjectionMatrix
               * (gl_ModelViewMatrix * vec4(0.0, 0.0, 0.0, 1.0)
               + vec4(gl_Vertex.x, gl_Vertex.y, 0.0, 0.0));

            textureCoords =
               vec4(gl_Vertex.x + 0.5, gl_Vertex.y + 0.5, 0.0, 0.0);
         }

         #endif

         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor = texture2D(_MainTex, vec2(textureCoords));
         }

         #endif

         ENDGLSL
      }
   }
}
```

Note that we never apply the model matrix to the object coordinates because we don't want to rotate them. However, this means that we also cannot scale them. Nonetheless, if you specify scale factors for the cube in Unity, you will notice that the billboard is actually scaled. The reason is that Unity performs the scaling on the object coordinates before they are sent to the vertex shader (unless all three scale factors are positive and equal, then the scaling is specified by `1.0 / unity_Scale.w`). Thus, in order to scale the billboard you can either use the scaling by Unity (with different scale factors for x and y) or you can introduce additional shader properties for scaling the object coordinates in the vertex shader.

38.0.214. Summary

Congratulations, you made it to the end of this tutorial. We have seen:

- How to transform and texture a cube in order to render a view-aligned billboard.

38.0.215. Further Reading

If you still want to learn more

- about object space, world space, view space, `gl_ModelViewMatrix` and `gl_ProjectionMatrix`, you should read the description in GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[7].
- about texturing, you should read GLSL PROGRAMMING/UNITY/TEXTURED SPHERES[8].

7 Chapter 43 on page 407
8 Chapter 16 on page 137

39. Nonlinear Deformations

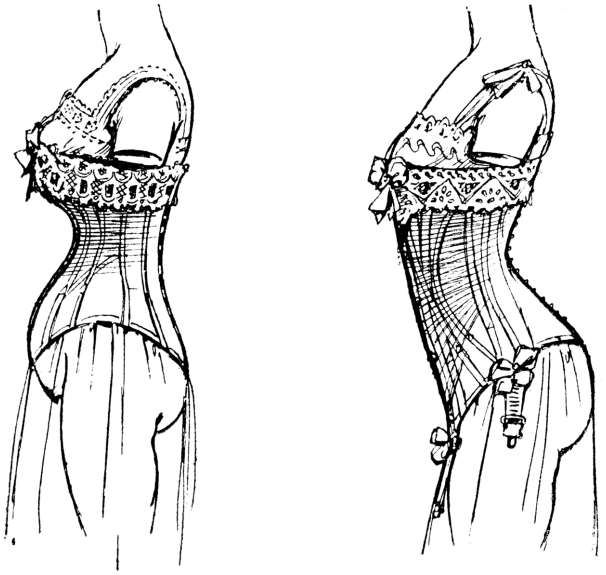

Figure 69 Illustration of deformations by corsets. These are examples of nonlinear deformations that cannot be modeled by a linear transformation.

This tutorial introduces **vertex blending** as an example of a nonlinear deformation. The main application is actually the rendering of skinned meshes.

While this tutorial is not based on any other specific tutorial, a good understanding of GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[1] is very useful.

39.0.216. Blending between Two Model Transformations

Most deformations of meshes cannot be modeled by the affine transformations with 4×4 matrices that are discussed in GLSL PROGRAMMING/VERTEX TRANSFOR-

1 Chapter 43 on page 407

MATIONS[2]. The deformation of bodies by tight corsets is just one example. A more important example in computer graphics is the deformation of meshes when joints are bent, e.g. elbows or knees.

This tutorial introduces vertex blending to implement some of these deformations. The basic idea is to apply multiple model transformations in the vertex shader (in this tutorial we use only two model transformations) and then blend the transformed vertices, i.e. compute a weighted average of them with weights that have to be specified for each vertex. For example, the deformation of the skin near a joint of a skeleton is mainly influenced by the position and orientation of the two (rigid) bones meeting in the joint. Thus, the positions and orientations of the two bones define two affine transformations. Different points on the skin are influenced differently by the two bones: points at the joint might be influenced equally by the two bones while points farther from the joint around one bone are more strongly influenced by that bone than the other. These different strengths of the influence of the two bones can be implemented by using different weights in the weighted average of the two transformations.

For the purpose of this tutorial, we use two uniform transformations `mat4 _Trafo0` and `mat4 _Trafo1`, which are specified by the user. To this end a small JavaScript (which should be attached to the mesh that should be deformed) allows to specify two other game objects and copies their model transformations to the uniforms of the shader:

```
@script ExecuteInEditMode()

public var bone0 : GameObject;
public var bone1 : GameObject;

function Update ()
{
   if (null != bone0)
   {
      renderer.sharedMaterial.SetMatrix("_Trafo0",
         bone0.renderer.localToWorldMatrix);
   }
   if (null != bone1)
   {
      renderer.sharedMaterial.SetMatrix("_Trafo1",
         bone1.renderer.localToWorldMatrix);
   }
   if (null != bone0 && null != bone1)
   {
      transform.position = 0.5 * (bone0.transform.position
         + bone1.transform.position);
      transform.rotation = bone0.transform.rotation;
```

2 Chapter 43 on page 407

```
    }
}
```

The two other game objects could be anything — I like cubes with one of the built-in semitransparent shaders such that their position and orientation is visible but they don't occlude the deformed mesh.

In this tutorial, the weight for the blending with the transformation `_Trafo0` is set to `gl_Vertex.z + 0.5`:

```
float weight0 = gl_Vertex.z + 0.5;
```

and the other weight is `1.0 - weight0`. Thus, the part with positive `gl_Vertex.z` coordinates is influenced more by `_Trafo0` and the other part is influenced more by `_Trafo1`. In general, the weights are application dependent and the user should be allowed to specify weights for each vertex.

The application of the two transformations and the weighted average can be written this way:

```
vec4 blendedVertex = weight0 * (_Trafo0 * gl_Vertex) +
(1.0 - weight0) * (_Trafo1 * gl_Vertex);
```

Then the blended vertex has to be multiplied with the view matrix and the projection matrix. The view transformation is not available directly but it can be computed by multiplying the model-view matrix (which is the product of the view matrix and the model matrix) with the inverse model matrix (which is available as `_World2Object` times `unity_Scale.w` except for the bottom-right element, which is 1):

```
mat4 modelMatrixInverse = _World2Object * unity_Scale.w;
modelMatrixInverse[3][3] = 1.0;
mat4 viewMatrix = gl_ModelViewMatrix * modelMatrixInverse;

gl_Position = gl_ProjectionMatrix * viewMatrix * blendedVertex;
```

In order to illustrate the different weights, we visualize `weight0` by the red component and `1.0 - weight0` by the green component of a color (which is set in the fragment shader):

```
color = vec4(weight0, 1.0 - weight0, 0.0, 1.0);
```

For an actual application, we could also transform the normal vector by the two corresponding transposed inverse model transformations and perform per-pixel lighting in the fragment shader.

39.0.217. Complete Shader Code

All in all, the shader code looks like this:

```
Shader "GLSL shader for vertex blending" {
   SubShader {
      Pass {
         GLSLPROGRAM

         // Uniforms set by a script
         uniform mat4 _Trafo0; // model transformation of bone0
         uniform mat4 _Trafo1; // model transformation of bone1

         // The following built-in uniforms
         // are also defined in "UnityCG.glslinc",
         // i.e. one could #include "UnityCG.glslinc"
         uniform mat4 _Object2World; // model matrix
         uniform vec4 unity_Scale; // w = 1/scale
         uniform mat4 _World2Object; // inverse model matrix
            // all but the bottom-right element should be
            // multiplied with unity_Scale.w

         // Varyings
         varying vec4 color;

         #ifdef VERTEX

         void main()
         {
            float weight0 = gl_Vertex.z + 0.5; // depends on the mesh

            vec4 blendedVertex = weight0 * (_Trafo0 * gl_Vertex)
               + (1.0 - weight0) * (_Trafo1 * gl_Vertex);

            mat4 modelMatrixInverse = _World2Object * unity_Scale.w;
            modelMatrixInverse[3][3] = 1.0;
            mat4 viewMatrix = gl_ModelViewMatrix * modelMatrixInverse;

            gl_Position =
               gl_ProjectionMatrix * viewMatrix * blendedVertex;

            color = vec4(weight0, 1.0 - weight0, 0.0, 1.0);
               // visualize weight0 as red and weight1 as green
         }

         #endif


         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor = color;
         }

         #endif
```

```
            ENDGLSL
        }
    }
}
```

This is, of course, only an illustration of the concept but it can already be used for some interesting nonlinear deformations such as twists around the z axis.

For skinned meshes in skeletal animation, many more bones (i.e. model transformations) are necessary and each vertex has to specify which bone (using, for example, an index) contributes with which weight to the weighted average. However, Unity computes the blending of vertices in software; thus, this topic is less relevant for Unity programmers.

39.0.218. Summary

Congratulations, you have reached the end of another tutorial. We have seen:

- How to blend vertices that are transformed by two model matrices.
- How this technique can be used for nonlinear transformations and skinned meshes.

39.0.219. Further Reading

If you still want to learn more

- about the model transformation, the view transformation, and the projection, you should read the description in GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[3].
- about vertex skinning, you could read the section about vertex skinning in Chapter 8 of the "OpenGL ES 2.0 Programming Guide" by Aaftab Munshi, Dan Ginsburg, and Dave Shreiner, published 2009 by Addison-Wesley.

3 Chapter 43 on page 407

40. Shadows on Planes

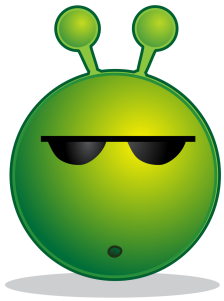

Figure 70 A cartoon character with a drop shadow.

This tutorial covers the **projection of shadows onto planes**.

It is not based on any particular tutorial; however, some understanding of GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[1] is useful.

40.0.220. Projecting Hard Shadows onto Planes

Computing realistic shadows in real time is difficult. However, there are certain cases that are a lot easier. Projecting a hard shadow (i.e. a shadow without penumbra; see GLSL PROGRAMMING/UNITY/SOFT SHADOWS OF SPHERES[2]) onto a plane is one of these cases. The idea is to render the shadow by rendering the shadow-casting object in the color of the shadow with the vertices projected just above the shadow-receiving plane.

1 Chapter 43 on page 407
2 Chapter 35 on page 333

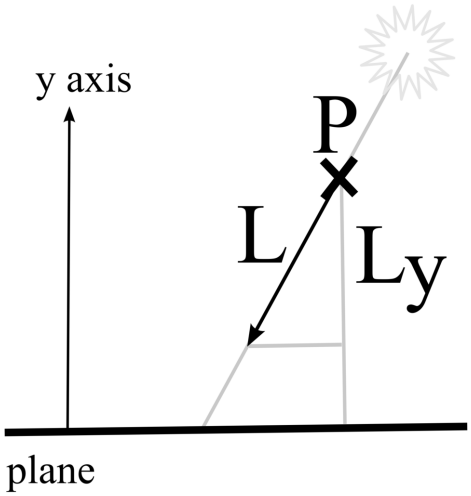

Figure 71 Illustration of the projection of a point **P** in direction **L** onto a plane in the coordinate system of the plane.

40.0.221. Projecting an Object onto a Plane

In order to render the projected shadow, we have to project the object onto a plane. In order to specify the plane, we will use the local coordinate system of the default plane game object. Thus, we can easily modify the position and orientation of the plane by editing the plane object. In the coordinate system of that game object, the actual plane is just the $y = 0$ plane, which is spanned by the x and z axes.

Projecting an object in a vertex shader means to project each vertex. This could be done with a projection matrix similar to the one discussed in GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[3]. However, those matrices are somewhat difficult to compute and debug. Therefore, we will take another approach and compute the projection with a bit of vector arithmetics. The illustration to the left shows the projection of a point **P** in the direction of light **L** onto a shadow-receiving plane. (Note that the vector **L** is in the opposite direction than the light vectors that are usually employed in lighting computations.) In order to move the point **P**

3 Chapter 43 on page 407

to the plane, we add a scaled version of **L**. The scaling factor turns out to be the distance of **P** to the plane divided by the length of **L** in the direction of the normal vector of the plane (because of similar triangles as indicated by the gray lines). In the coordinate system of the plane, where the normal vector is just the *y* axis, we can also use the ratio of the *y* coordinate of the point **P** divided by the negated *y* coordinate of the vector **L**.

Thus, the vertex shader could look like this:

```
GLSLPROGRAM

// User-specified uniforms
uniform mat4 _World2Receiver; // transformation from
   // world coordinates to the coordinate system of the plane

// The following built-in uniforms
// are also defined in "UnityCG.glslinc",
// i.e. one could #include "UnityCG.glslinc"
uniform mat4 _Object2World; // model matrix
uniform mat4 _World2Object; // inverse model matrix
uniform vec4 unity_Scale; // w = 1/uniform scale;
   // should be multiplied to _World2Object
uniform vec4 _WorldSpaceLightPos0;
   // position or direction of light source

#ifdef VERTEX

void main()
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object * unity_Scale.w;
   modelMatrixInverse[3][3] = 1.0;
   mat4 viewMatrix = gl_ModelViewMatrix * modelMatrixInverse;

   vec4 lightDirection;
   if (0.0 != _WorldSpaceLightPos0.w) // point or spot light?
   {
      lightDirection = normalize(
         modelMatrix * gl_Vertex - _WorldSpaceLightPos0);
   }
   else // directional light
   {
      lightDirection = -normalize(_WorldSpaceLightPos0);
   }

   vec4 vertexInWorldSpace = modelMatrix * gl_Vertex;
   float distanceOfVertex =
      (_World2Receiver * vertexInWorldSpace).y;
      // = height over plane
   float lengthOfLightDirectionInY =
      (_World2Receiver * lightDirection).y;
      // = length in y direction

   lightDirection = lightDirection
      * (distanceOfVertex / (-lengthOfLightDirectionInY));
```

```
            gl_Position = gl_ProjectionMatrix * (viewMatrix
               * (vertexInWorldSpace + lightDirection));
         }

         #endif
         ...
```

The uniform `_World2Receiver` is best set with the help of a small script that should be attached to the shadow-casting object:

```
@script ExecuteInEditMode()

public var plane : GameObject;

function Update ()
{
   if (null != plane)
   {
      renderer.sharedMaterial.SetMatrix("_World2Receiver",
         plane.renderer.worldToLocalMatrix);
   }
}
```

The script requires the user to specify the shadow-receiving plane object and sets the uniform `_World2Receiver` accordingly.

40.0.222. Complete Shader Code

For the complete shader code we improve the performance by noting that the y coordinate of a matrix-vector product is just the dot product of the second row (i.e. the first when starting with 0) of the matrix and the vector. Furthermore, we improve the robustness by not moving the vertex when it is below the plane, neither when the light is directed upwards. Additionally, we try to make sure that the shadow is on top of the plane with this instruction:

```
Offset -1.0, -2.0
```

This reduces the depth of the rasterized triangles a bit such that they always occlude other triangles of approximately the same depth.

The first pass of the shader renders the shadow-casting object while the second pass renders the projected shadow. In an actual application, the first pass could be replaced by one or more passes to compute the lighting of the shadow-casting object.

```
Shader "GLSL planar shadow" {
   Properties {
```

```
   _Color ("Object's Color", Color) = (0,1,0,1)
   _ShadowColor ("Shadow's Color", Color) = (0,0,0,1)
}
SubShader {
   Pass {
      Tags { "LightMode" = "ForwardBase" } // rendering of object

      GLSLPROGRAM

      // User-specified properties
      uniform vec4 _Color;

      #ifdef VERTEX

      void main()
      {
         gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
      }

      #endif

      #ifdef FRAGMENT

      void main()
      {
         gl_FragColor = _Color;
      }

      #endif

      ENDGLSL
   }

   Pass {
      Tags { "LightMode" = "ForwardBase" }
         // rendering of projected shadow
      Offset -1.0, -2.0
         // make sure shadow polygons are on top of shadow receiver

      GLSLPROGRAM

      // User-specified uniforms
      uniform vec4 _ShadowColor;
      uniform mat4 _World2Receiver; // set by script

      // The following built-in uniforms )
      // are also defined in "UnityCG.glslinc",
      // i.e. one could #include "UnityCG.glslinc"
      uniform mat4 _Object2World; // model matrix
      uniform mat4 _World2Object; // inverse model matrix
      uniform vec4 unity_Scale; // w = 1/uniform scale;
         // should be multiplied to _World2Object
      uniform vec4 _WorldSpaceLightPos0;
         // position or direction of light source

      #ifdef VERTEX

      void main()
```

```
{
   mat4 modelMatrix = _Object2World;
   mat4 modelMatrixInverse = _World2Object * unity_Scale.w;
   modelMatrixInverse[3][3] = 1.0;
   mat4 viewMatrix = gl_ModelViewMatrix * modelMatrixInverse;

   vec4 lightDirection;
   if (0.0 != _WorldSpaceLightPos0.w)
   {
      // point or spot light
      lightDirection = normalize(
         modelMatrix * gl_Vertex - _WorldSpaceLightPos0);
   }
   else
   {
      // directional light
      lightDirection = -normalize(_WorldSpaceLightPos0);
   }

   vec4 vertexInWorldSpace = modelMatrix * gl_Vertex;
   vec4 world2ReceiverRow1 =
      vec4(_World2Receiver[0][1], _World2Receiver[1][1],
      _World2Receiver[2][1], _World2Receiver[3][1]);
   float distanceOfVertex =
      dot(world2ReceiverRow1, vertexInWorldSpace);
      // = (_World2Receiver * vertexInWorldSpace).y
      // = height over plane
   float lengthOfLightDirectionInY =
      dot(world2ReceiverRow1, lightDirection);
      // = (_World2Receiver * lightDirection).y
      // = length in y direction

   if (distanceOfVertex > 0.0 && lengthOfLightDirectionInY < 0.0)
   {
      lightDirection = lightDirection
         * (distanceOfVertex / (-lengthOfLightDirectionInY));
   }
   else
   {
      lightDirection = vec4(0.0, 0.0, 0.0, 0.0);
         // don't move vertex
   }

   gl_Position = gl_ProjectionMatrix * (viewMatrix
      * (vertexInWorldSpace + lightDirection));
}

#endif

#ifdef FRAGMENT

void main()
{
   gl_FragColor = _ShadowColor;
}

#endif
```

```
            ENDGLSL
        }
    }
}
```

40.0.223. Further Improvements of the Fragment Shader

There are a couple of things that could be improved, in particular in the fragment shader:

- Fragments of the shadow that are outside of the rectangular plane object could be removed with the `discard` instruction, which was discussed in GLSL PROGRAMMING/UNITY/CUTAWAYS[4].
- If the plane is textured, this texturing could be integrated by using only local vertex coordinates for the texture lookup (also in the shader of the plane object) and specifying the texture of the plane as a shader property of the shadow-casting object.
- Soft shadows could be faked by computing the lighting of the plane in this shader and attenuating it depending on the angle of the surface normal vector of the shadow-casting object to the light direction similar to the approach in GLSL PROGRAMMING/UNITY/SILHOUETTE ENHANCEMENT[5].

40.0.224. Summary

Congratulations, this is the end of this tutorial. We have seen:

- How to project a vertex in the direction of light onto a plane.
- How to implement this technique to project a shadow onto a plane.

40.0.225. Further Reading

If you still want to learn more

- about the model transformation, the view transformation, and the projection, you should read the description in GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[6].

4 Chapter 6 on page 43
5 Chapter 9 on page 67
6 Chapter 43 on page 407

- about setting up a projection matrix to project the shadow, you could read Section 9.4.1 of the SIGGRAPH '97 Course "Programming with OpenGL: Advanced Rendering" organized by Tom McReynolds, which is available ONLINE[7].

7 HTTP://WWW.OPENGL.ORG/RESOURCES/CODE/SAMPLES/ADVANCED/ADVANCED97/NOTES/NODE100.HTML

41. Mirrors

Figure 72 “Toilet of Venus”, ca. 1644-48 by Diego Rodríguez de Silva y Velázquez.

This tutorial covers the rendering of virtual images of objects in **plane mirrors**.

It is based on blending as described in GLSL PROGRAMMING/UNITY/TRANSPARENCY[1] and requires some understanding of GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[2].

41.0.226. Virtual Images in Plane Mirrors

The image we see in a plane mirror is called a “virtual image” because it is the same as the image of the real scene except that all positions are mirrored at the plane of the mirror; thus, we don't see the real scene but a “virtual” image of it.

1 Chapter 7 on page 49
2 Chapter 43 on page 407

This transformation of a real object to a virtual object can be computed by transforming each position from world space into the local coordinate system of the mirror; negating the y coordinate (assuming that the mirror plane is spanned by the x and z axes); and transforming the resulting position back to world space. This suggests a very straightforward approach to rendering virtual images of game objects by using another shader pass with a vertex shader that mirrors every vertex and normal vector, and a fragment shader that mirrors the position of light sources before computing the shading. (In fact, the light source at the original positions might also be taken into account because they represent light that is reflected by the mirror before reaching the real object.) There isn't anything wrong with this approach except that it is very limited: no other objects may be behind the mirror plane (not even partially) and the space behind the mirror plane must only be visible through the mirror. This is fine for mirrors on walls of a box that contains the whole scene if all the geometry outside the box can be removed. However, it doesn't work for mirrors with objects behind it (as in the painting by Velaquez) nor for semitransparent mirrors, for example glass windows.

41.0.227. Placing the Virtual Objects

It turns out that implementing a more general solution is not straightforward in the free version of Unity because neither rendering to textures (which would allow us to render the scene from a virtual camera position behind the mirror) nor stencil buffers (which would allow us to restrict the rendering to the region of the mirror) are available in the free version of Unity.

I came up with the following solution: First, every game object that might appear in the mirror has to have a virtual “Doppelgänger”, i.e. a copy that follows all the movements of the real game object but with positions mirrored at the mirror plane. Each of these virtual objects needs a script that sets its position and orientation according to the corresponding real object and the mirror plane, which are specified by public variables:

```
@script ExecuteInEditMode()

var objectBeforeMirror : GameObject;
var mirrorPlane : GameObject;

function Update ()
{
   if (null != mirrorPlane)
   {
      renderer.sharedMaterial.SetMatrix("_WorldToMirror",
         mirrorPlane.renderer.worldToLocalMatrix);
      if (null != objectBeforeMirror)
```

```
        {
            transform.position = objectBeforeMirror.transform.position;
            transform.rotation = objectBeforeMirror.transform.rotation;
            transform.localScale =
                -objectBeforeMirror.transform.localScale;
            transform.RotateAround(objectBeforeMirror.transform.position,
                mirrorPlane.transform.TransformDirection(
                Vector3(0.0, 1.0, 0.0)), 180.0);

            var positionInMirrorSpace : Vector3 =
                mirrorPlane.transform.InverseTransformPoint(
                objectBeforeMirror.transform.position);
            positionInMirrorSpace.y = -positionInMirrorSpace.y;
            transform.position = mirrorPlane.transform.TransformPoint(
                positionInMirrorSpace);
        }
    }
}
```

The origin of the local coordinate system (`objectBeforeMirror.transform.position`) is transformed as described above; i.e., it's transformed to the local coordinate system of the mirror with `mirrorPlane.transform.InverseTransformPoint()`, then the y coordinate is reflected, and then it is transformed back to world space with `mirrorPlane.transform.TransformPoint()`. However, the orientation is a bit difficult to specify in JavaScript: we have to reflect all coordinates (`transform.localScale = -objectBeforeMirror.transform.localScale`) and rotate the virtual object by 180° around the surface normal vector of the mirror (`Vector3(0.0, 1.0, 0.0)` transformed to world coordinates. This does the trick because a rotation around 180° corresponds to the reflection of two axes orthogonal to the rotation axis. Thus, this rotation undoes the previous reflection for two axes and we are left with the one reflection in the direction of the rotation axis, which was chosen to be the normal of the mirror.

Of course, the virtual objects should always follow the real object, i.e. they shouldn't collide with other objects nor be influenced by physics in any other way. Using this script on all virtual objects is already sufficient for the case mentioned above: no real objects behind the mirror plane and no other way to see the space behind the mirror plane except through the mirror. In other cases we have to render the mirror in order to occlude the real objects behind it.

41.0.228. Rendering the Mirror

Now things become a bit tricky. Let's list what we want to achieve:

- Real objects behind the mirror should be occluded by the mirror.
- The mirror should be occluded by the virtual objects (which are actually behind it).
- Real objects in front of the mirror should occlude the mirror and any virtual objects.
- Virtual objects should only be visible in the mirror, not outside of it.

If we could restrict rendering to an arbitrary part of the screen (e.g. with a stencil buffer), this would be easy: render all geometry including an opaque mirror; then restrict the rendering to the visible parts of the mirror (i.e. not the parts that are occluded by other real objects); clear the depth buffer in these visible parts of the mirror; and render all virtual objects. It's straightforward if we had a stencil buffer.

Since we don't have a stencil buffer, we use the alpha component (a.k.a. opacity or A component) of the framebuffer as a substitute (similar to the technique used in GLSL PROGRAMMING/UNITY/TRANSLUCENT BODIES[3]). In the first pass of the shader for the mirror, all pixels in the visible part of the mirror (i.e. the part that is not occluded by real objects in front of it) will be marked by an alpha component of 0, while pixels in the rest of the screen should have an alpha component of 1. The first problem is that we have to make sure that the rest of the screen has an alpha component of 1, i.e. all background shaders and object shaders should set alpha to 1. For example, Unity's skyboxes don't set alpha to 1; thus, we have to modify and replace all those shaders that don't set alpha to 1. Let's assume that we can do that. Then the first pass of the shader for the mirror is:

```
// 1st pass: mark mirror with alpha = 0
Pass {
   GLSLPROGRAM

   #ifdef VERTEX

   void main()
   {
      gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
   }

   #endif

   #ifdef FRAGMENT

   void main()
   {
      gl_FragColor = vec4(1.0, 0.0, 0.0, 0.0);
         // this color should never be visible,
         // only alpha is important
```

3 Chapter 34 on page 319

```
        }

        #endif

        ENDGLSL
    }
```

How does this help us to limit the rendering to the pixels with alpha equal to 0? It doesn't. However, it does help us to restrict any changes of colors in the framebuffer by using a clever blend equation (see GLSL PROGRAMMING/UNITY/TRANSPARENCY[4]):

```
Blend OneMinusDstAlpha DstAlpha
```

We can think of the blend equation as:

```
vec4 result = vec4(1.0 - pixel_color.a) * gl_FragColor
+ vec4(pixel_color.a) * pixel_color;
```

where `pixel_color` is the color of a pixel in the framebuffer. Let's see what the expression is for `pixel_color.a` equal to 1 (i.e. outside of the visible part of the mirror):

```
vec4(1.0 - 1.0) * gl_FragColor + vec4(1.0) * pixel_-
color == pixel_color
```

Thus, if `pixel_color.a` is equal to 1, the blending equation makes sure that we don't change the pixel color in the framebuffer. What happens if `pixel_color.a` is equal to 0 (i.e. inside the visible part of the mirror)?

```
vec4(1.0 - 0.0) * gl_FragColor + vec4(0.0) * pixel_-
color == gl_FragColor
```

In this case, the pixel color of the framebuffer will be set to the fragment color that was set in the fragment shader. Thus, using this blend equation, our fragment shader will only change the color of pixels with an alpha component of 0. Note that the alpha component in `gl_FragColor` should also be 0 such that the pixels are still marked as part of the visible region of the mirror.

That was the first pass. The second pass has to clear the depth buffer before we start to render the virtual objects such that we can use the normal depth test to compute occlusions (see GLSL PROGRAMMING/PER-FRAGMENT OPERATIONS[5]). Actually, it doesn't matter whether we clear the depth buffer only for the pixels in the visible part of the mirror or for all pixels of the screen because we won't change

4 Chapter 7 on page 49
5 Chapter 47 on page 443

the colors of any pixels with alpha equal to 1 anyways. In fact, this is very fortunate because (without stencil test) we cannot limit the clearing of the depth buffer to the visible part of the mirror. Instead, we clear the depth buffer for the whole mirror by transforming the vertices to the far clipping plane, i.e. the maximum depth.

As explained in GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[6], the output of the vertex shader in `gl_Position` is divided automatically by the fourth coordinate `gl_Position.w` to compute normalized device coordinates between -1 and +1. In fact, a z coordinate of +1 represents the maximum depth; thus, this is what we are aiming for. However, because of that automatic (perspective) division by `gl_Position.w`, we have to set `gl_Position.z` to `gl_Position.w` in order to get a normalized device coordinate of +1. Here is the second pass of the mirror shader:

```
// 2nd pass: set depth to far plane such that
// we can use the normal depth test for the reflected geometry
Pass {
   ZTest Always
   Blend OneMinusDstAlpha DstAlpha

   GLSLPROGRAM

   uniform vec4 _Color;
      // user-specified background color in the mirror

   #ifdef VERTEX

   void main()
   {
      gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
      gl_Position.z = gl_Position.w;
         // the perspective division will divide gl_Position.z
         // by gl_Position.w; thus, the depth is 1.0,
         // which represents the far clipping plane
   }

   #endif

   #ifdef FRAGMENT

   void main()
   {
      gl_FragColor = vec4(_Color.rgb, 0.0);
         // set alpha to 0.0 and
         // the color to the user-specified background color
   }

   #endif
```

6 Chapter 43 on page 407

```
         ENDGLSL
      }
```

The `ZTest` is set to `Always` in order to deactivate it. This is necessary because our vertices are actually behind the mirror (in order to reset the depth buffer); thus, the fragments would fail a normal depth test. We use the blend equation which was discussed above to set the user-specified background color of the mirror. (If there is a skybox in your scene, you would have to compute the mirrored view direction and look up the environment map here; see GLSL PROGRAMMING/UNITY/SKYBOXES[7].)

This is the shader for the mirror. Here is the complete shader code, which uses `"Transparent+10"` to make sure that it is rendered after all real objects (including transparent objects) have been rendered:

```
Shader "GLSL shader for mirrors" {
   Properties {
      _Color ("Mirrors's Color", Color) = (1, 1, 1, 1)
   }
   SubShader {
      Tags { "Queue" = "Transparent+10" }
         // draw after all other geometry has been drawn
         // because we mess with the depth buffer

      // 1st pass: mark mirror with alpha = 0
      Pass {
         GLSLPROGRAM

         #ifdef VERTEX

         void main()
         {
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
         }

         #endif

         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor = vec4(1.0, 0.0, 0.0, 0.0);
               // this color should never be visible,
               // only alpha is important
         }

         #endif

         ENDGLSL
      }
```

7 Chapter 28 on page 253

```
      // 2nd pass: set depth to far plane such that
      // we can use the normal depth test for the reflected geometry
      Pass {
         ZTest Always
         Blend OneMinusDstAlpha DstAlpha

         GLSLPROGRAM

         uniform vec4 _Color;
            // user-specified background color in the mirror

         #ifdef VERTEX

         void main()
         {
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
            gl_Position.z = gl_Position.w;
               // the perspective division will divide gl_Position.z
               // by gl_Position.w; thus, the depth is 1.0,
               // which represents the far clipping plane
         }

         #endif

         #ifdef FRAGMENT

         void main()
         {
            gl_FragColor = vec4(_Color.rgb, 0.0);
               // set alpha to 0.0 and
               // the color to the user-specified background color
         }

         #endif

         ENDGLSL
      }
   }
}
```

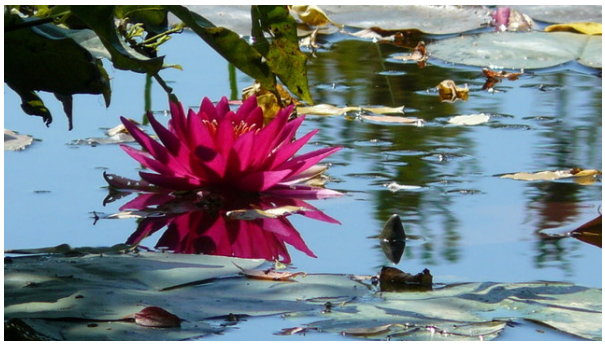

Figure 73 A water lily in Sheffield Park.

41.0.229. Rendering the Virtual Objects

Once we have cleared the depth buffer and marked the visible part of the mirror by setting the alpha component to 0, we can use the blend equation

```
Blend OneMinusDstAlpha DstAlpha
```

to render the virtual objects. Can't we? There is another situation in which we shouldn't render virtual objects and that's when they come out of the mirror! This can actually happen when real objects move into the reflecting surface. Water lilies and swimming objects are examples. We can avoid the rasterization of fragments of virtual objects that are outside the mirror by discarding them with the `discard` instruction (see GLSL PROGRAMMING/UNITY/CUTAWAYS[8]) if their *y* coordinate in the local coordinate system of the mirror is positive. To this end, the vertex shader has to compute the vertex position in the local coordinate system of the mirror and therefore the shader requires the corresponding transformation matrix, which we have fortunately set in the script above. The complete shader code for the virtual objects is then:

```
Shader "GLSL shader for virtual objects in mirrors" {
   Properties {
      _Color ("Virtual Object's Color", Color) = (1, 1, 1, 1)
   }
   SubShader {
      Tags { "Queue" = "Transparent+20" }
         // render after mirror has been rendered

      Pass {
         Blend OneMinusDstAlpha DstAlpha
            // when the framebuffer has alpha = 1, keep its color
            // only write color where the framebuffer has alpha = 0

         GLSLPROGRAM

         // User-specified uniforms
         uniform vec4 _Color;
         uniform mat4 _WorldToMirror; // set by a script

         // The following built-in uniforms
         // are also defined in "UnityCG.glslinc",
         // i.e. one could #include "UnityCG.glslinc"
         uniform mat4 _Object2World; // model matrix

         // Varying
         varying vec4 positionInMirror;

         #ifdef VERTEX
```

8 Chapter 6 on page 43

```
         void main()
         {
            positionInMirror =
               _WorldToMirror * (_Object2World * gl_Vertex);
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;;
         }

         #endif

         #ifdef FRAGMENT

         void main()
         {
            if (positionInMirror.y > 0.0)
               // reflection comes out of mirror?
            {
               discard; // don't rasterize it
            }
            gl_FragColor = vec4(_Color.rgb, 0.0); // set alpha to 0.0
         }

         #endif

         ENDGLSL
      }
   }
}
```

Note that the line

```
Tags { "Queue" = "Transparent+20" }
```

makes sure that the virtual objects are rendered after the mirror, which uses `"Transparent+10"`. In this shader, the virtual objects are rasterized with a uniform, user-specified color in order to keep the shader as short as possible. In a complete solution, the shader would compute the lighting and texturing with the mirrored normal vector and mirrored positions of light sources. However, this is straightforward and very much dependent on the particular shaders that are employed for the real objects.

41.0.230. Limitations

There are several limitations of this approach which we haven't addressed. For example:

- multiple mirror planes (virtual objects of one mirror might appear in another mirror)
- multiple reflections in mirrors
- semitransparent virtual objects
- semitransparent mirrors

- reflection of light in mirrors
- uneven mirrors (e.g. with a normal map)
- uneven mirrors in the free version of Unity
- etc.

41.0.231. Summary

Congratulations! Well done. Two of the things we have looked at:

- How to render mirrors with a stencil buffer.
- How to render mirrors without a stencil buffer.

41.0.232. Further Reading

If you still want to know more

- about using the stencil buffer to render mirrors, you could read Section 9.3.1 of the SIGGRAPH '97 Course "Programming with OpenGL: Advanced Rendering" organized by Tom McReynolds, which is available ONLINE[9].

9 HTTP://WWW.OPENGL.ORG/RESOURCES/CODE/SAMPLES/ADVANCED/ADVANCED97/NOTES/NODE90.HTML

Part IX.

Appendix on the OpenGL Pipeline and GLSL Syntax

42. OpenGL ES 2.0 Pipeline

The OpenGL ES 2.0 pipeline is important for GLSL shaders in OpenGL ES 2.0 and WebGL. It is also very similar to the OpenGL 2.0 pipeline without many of the features that were deprecated in newer versions of OpenGL. Therefore, the OpenGL ES 2.0 pipeline is not only highly relevant for programmers of mobile graphics using OpenGL ES 2.0 and web-based 3D graphics using WebGL, but also a very good starting point to learn about desktop-based 3D graphics using OpenGL, including 3D graphics in game engines such as Blender, Unity and Torque 3D.

42.0.233. Parallelism in the OpenGL Pipeline

GPUs are highly parallel processors. This is the main reason for their performance. In fact, they implement two kinds of parallelism: vertical and horizontal parallelism:

Figure 74

- **Vertical parallelism** describes parallel processing at different **stages of a pipeline**. This concept was also crucial in the development of the assembly

line at Ford Motor Company: many workers can work in parallel on rather simple tasks. This made mass production (and therefore mass consumption) possible. In the context of processing units in GPUs, the simple tasks correspond to less complex processing units, which save costs and power consumption.

Figure 75

- **Horizontal parallelism** describes the possibility to process work in **multiple pipelines**. This allows for even more parallelism than the vertical parallelism in a single pipeline. Again, the concept was also employed at Ford Motor Company and in many other industries. In the context of GPUs, horizontal parallelism of the graphics pipeline was an important feature to achieve the performance of modern GPUs.

The following diagram shows an illustration of vertical parallelism (processing in stages represented by boxes) and horizontal parallelism (multiple processing units for each stage represented by multiple arrows between boxes).

Vertex Data	e.g. triangle meshes provided by 3D modeling tools
↓↓↓↓↓↓↓	many vertices are processed in parallel

Vertex Shader	a small program in GLSL is applied to each vertex
↓↓↓↓↓↓↓	
Primitive Assembly	setup of primitives, e.g. triangles, lines, and points
↓↓↓↓↓↓↓	many primitives are processed in parallel
Rasterization	interpolation of data for all pixels covered by the primitive (e.g. triangle)
↓↓↓↓↓↓↓	many fragments (corresponding to pixels) are processed in parallel
Fragment Shader	a small program in GLSL is applied to each fragment (i.e. covered pixel)
↓↓↓↓↓↓↓	
Per-Fragment Operations	configurable operations on each fragment (i.e. covered pixel)

↓↓↓↓↓↓↓	results of many fragments are written in parallel to the framebuffer
Framebuffer	array of pixels in which the computed fragment colors are stored

In the following diagrams, there is only one arrow between any two stages. However, it should be understood that GPUs usually implement the graphics pipeline with massive horizontal parallelism. Only software implementations of OpenGL, e.g. Mesa 3D (SEE THE WIKIPEDIA ENTRY)[1], usually implement a single pipeline.

42.0.234. Programmable and Fixed-Function Stages

The pipelines of OpenGL ES 1.x and core OpenGL 1.x are configurable fixed-function pipelines, i.e. there is no possibility to include programs in these pipelines. In OpenGL (ES) 2.0 two stages (the vertex shader and the fragment shader stage) of the pipeline are programmable, i.e. small programs (shaders) written in GLSL are applied in these stages. In the following diagram, programmable stages are represented by green boxes, fixed-function stages are represented by gray boxes, and data is represented by blue boxes.

Vertex Data	e.g. triangle meshes provided by 3D modeling tools
↓	
Vertex Shader	a small program in GLSL is applied to each vertex

1 HTTP://EN.WIKIPEDIA.ORG/WIKI/MESA_3D_%28OPENGL%29

↓	
Primitive Assembly	setup of primitives, e.g. triangles, lines, and points
↓	
Rasterization	interpolation of data (e.g. color) for all pixels covered by the primitive
↓	
Fragment Shader	a small program in GLSL is applied to each fragment (i.e. covered pixel)
↓	
Per-Fragment Operations	configurable operations on each fragment (i.e. covered pixel)
↓	
Framebuffer	array of pixels in which the computed fragment colors are stored

The vertex shader and fragment shader stages are discussed in more detail in the platform-specific tutorials. The rasterization stage is discussed in GLSL

PROGRAMMING/RASTERIZATION[2] and the per-fragment operations in GLSL PROGRAMMING/PER-FRAGMENT OPERATIONS[3].

The primitive assembly stage mainly consists of clipping primitives to the view frustum (the part of space that is visible on the screen) and optional culling of front-facing and/or back-facing primitives. These possibilities are discussed in more detail in the platform-specific tutorials.

42.0.235. Data Flow

In order to program GLSL vertex and fragment shaders, it is important to understand the input and ouput of each shader. To this end, it is also useful to understand how data is communicated between all stages of the OpenGL pipeline. This is illustrated in the next diagram:

Vertex Data		
	↓	attributes (usually position, color, normal vector, and texture coordinates)
Vertex Shader	←	uniforms (constants); (optionally texture data but not in most OpenGL ES 2.0 implementations)
	↓	varyings (at vertices), vertex position, and point size
Primitive Assembly		
	↓	varyings (at vertices), vertex position, and point size
Rasterization		
	↓	varyings (now interpolated at pixels), fragment coordinates, point coordinates, front-facing flag

2 Chapter 46 on page 437
3 Chapter 47 on page 443

Fragment Shader	←	uniforms (constants) and texture data (images)
↓	fragment color and fragment depth	
Per-Fragment Operations		
↓	fragment color and fragment depth	
Framebuffer		

Attributes (or vertex attributes, or attribute variables) are defined based on the vertex data. The vertex position in an attribute is in object coordinates, i.e. this is the position as specified in a 3D modeling tool.

Uniforms (or uniform variables) have the same value for all vertex shaders and all fragment shaders that are executed when rendering a specific primitive (e.g. a triangle). However, they can be changed for other primitives. Typically, vertex transformations, specifications of light sources and materials, etc. are specified as uniforms.

Varyings (or varying variables) have to be consistently defined by the vertex shader and the fragment shader (i.e. the vertex shader has to define the same varying variables as the fragment shader). Typically, varyings are defined for colors, normal vectors, and/or texture coordinates.

Texture data include a uniform sampler, which specifies the texture sampling unit, which in turn specifies the texture image from which colors are fetched.

Other data is described in the tutorials for specific platforms.

42.0.236. Further Reading

The OpenGL ES 2.0 pipeline is defined in full detail in the "OpenGL ES 2.0.x Specification" and the "OpenGL ES Shading Language 1.0.x Specification" available at the "KHRONOS OPENGL ES API REGISTRY"[4].

A more accessible description of the OpenGL ES 2.0 pipeline is given in Chapter 1 of the book "OpenGL ES 2.0 Programming Guide" by Aaftab Munshi, Dan Ginsburg and Dave Shreiner published by Addison-Wesley (see ITS WEB SITE[5]).

4 HTTP://WWW.KHRONOS.ORG/REGISTRY/GLES/

5 HTTP://WWW.OPENGLES-BOOK.COM/

43. Vertex Transformations

One of the most important tasks of the vertex shader and the following stages in the OpenGL (ES) 2.0 pipeline[1] is the transformation of vertices of primitives (e.g. triangles) from the original coordinates (e.g. those specified in a 3D modeling tool) to screen coordinates. While programmable vertex shaders allow for many ways of transforming vertices, some transformations are performed in the fixed-function stages after the vertex shader. When programming a vertex shader, it is therefore particularly important to understand which transformations have to be performed in the vertex shader. These transformations are usually specified as uniform variables and applied to the incoming vertex positions and normal vectors by means of matrix-vector multiplications. While this is straightforward for points and directions, it is less straightforward for normal vectors as discussed in GLSL Programming/Applying Matrix Transformations[2].

Here, we will first present an overview of the coordinate systems and the transformations between them and then discuss individual transformations.

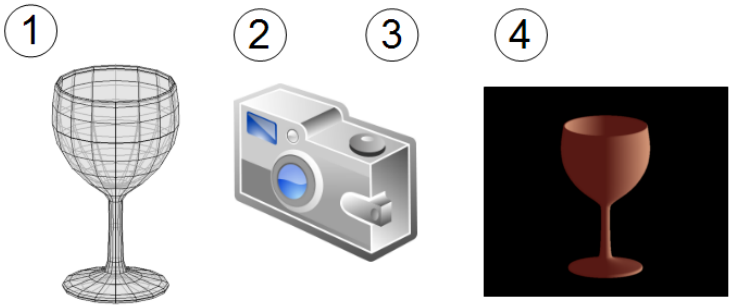

Figure 76

1 http://en.wikibooks.org/wiki/%2FOpenGL%20ES%202.0%20Pipeline
2 Chapter 45 on page 429

43.0.237. Overview: The Camera Analogy

It is useful to think of the whole process of transforming vertices in terms of a camera analogy as illustrated below . The steps and the corresponding vertex transformations are:

1. positioning the model — modeling transformation
2. positioning the camera — viewing transformation
3. adjusting the zoom — projection transformation
4. cropping the image — viewport transformation

The first three transformations are applied in the vertex shader. Then the perspective division (which might be considered part of the projection transformation) is automatically applied in the fixed-function stage after the vertex shader. The viewport transformation is also applied automatically in this fixed-function stage. While the transformations in the fixed-function stages cannot be modified, the other transformations can be replaced by other kinds of transformations than described here. It is, however, useful to know the conventional transformations since they allow to make best use of clipping and perspectively correct interpolation of varying variables.

The following overview shows the sequence of vertex transformations between various coordinate systems and includes the matrices that represent the transformations:

object/model coordinates		input to the vertex shader, i.e. position in attributes
↓	**modeling transformation**: model matrix $M_{object \to world}$	
world coordinates		
↓	**viewing transformation**: view matrix $M_{world \to view}$	
view/eye coordinates		
↓	**projection transformation**: projection matrix $M_{projection}$	
clip coordinates		output of the vertex shader, i.e. `gl_Position`
↓	**perspective division** (by `gl_Position.w`)	

normalized device coordinates		
↓	**viewport transformation**	
screen/window coordinates		`gl_FragCoord` in the fragment shader

Note that the modeling, viewing and projection transformation are applied in the vertex shader. The perspective division and the viewport transformation is applied in the fixed-function stage after the vertex shader. The next sections discuss all these transformations in detail.

43.0.238. Modeling Transformation

The modeling transformation specifies the transformation from object coordinates (also called model coordinates or local coordinates) to a common world coordinate system. Object coordinates are usually specific to each object or model and are often specified in 3D modeling tools. On the other hand, world coordinates are a common coordinate system for all objects of a scene, including light sources, 3D audio sources, etc. Since different objects have different object coordinate systems, the modeling transformations are also different; i.e., a different modeling transformation has to be applied to each object.

Structure of the Model Matrix

The modeling transformation can be represented by a 4×4 matrix, which we denote as the model matrix $M_{object \to world}$. Its structure is:

$$M_{object \to world} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & t_1 \\ a_{2,1} & a_{2,2} & a_{2,3} & t_2 \\ a_{3,1} & a_{3,2} & a_{3,3} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{with A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \quad \text{and } \mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}$$

A is a 3×3 matrix, which represents a linear transformation in 3D space. This includes any combination of rotations, scalings, and other less common linear transformations. **t** is a 3D vector, which represents a translation (i.e. displacement) in 3D space. $M_{object \to world}$ combines A and **t** in one handy 4×4 matrix. Mathematically

spoken, the model matrix represents an affine transformation: a linear transformation together with a translation. In order to make this work, all three-dimensional points are represented by four-dimensional vectors with the fourth coordinate equal to 1:

$$P = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{bmatrix}$$

When we multiply the matrix to such a point P, the combination of the three-dimensional linear transformation and the translation shows up in the result:

$$\mathrm{M}_{\text{object}\to\text{world}} P = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & t_1 \\ a_{2,1} & a_{2,2} & a_{2,3} & t_2 \\ a_{3,1} & a_{3,2} & a_{3,3} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{bmatrix} = \begin{bmatrix} a_{1,1}p_1 + a_{1,2}p_2 + a_{1,3}p_3 + t_1 \\ a_{2,1}p_1 + a_{2,2}p_2 + a_{2,3}p_3 + t_2 \\ a_{3,1}p_1 + a_{3,2}p_2 + a_{3,3}p_3 + t_3 \\ 1 \end{bmatrix}$$

Apart from the fourth coordinate (which is 1 as it should be for a point), the result is equal to

$$\mathrm{A} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}$$

Accessing the Model Matrix in a Vertex Shader

The model matrix $\mathrm{M}_{\text{object}\to\text{world}}$ can be defined as a uniform variable such that it is available in a vertex shader. However, it is usually combined with the matrix of the viewing transformation to form the modelview matrix, which is then set as a uniform variable. In some versions of OpenGL (ES), a built-in uniform variable `gl_ModelViewMatrix` is available in the vertex shader. (See also GLSL PROGRAMMING/APPLYING MATRIX TRANSFORMATIONS[3].)

Computing the Model Matrix

Strictly speaking, GLSL programmers don't have to worry about the computation of the model matrix since it is provided to the vertex shader in the form of a uniform variable. In fact, render engines, scene graphs, and game engines will usually provide the model matrix; thus, the programmer of a vertex shader doesn't have to

3 Chapter 45 on page 429

worry about computing the model matrix. However, when developing applications in modern versions of OpenGL and OpenGL ES or in WebGL, the model matrix has to be computed. (OpenGL before version 3.2, the compability profiles of newer versions of OpenGL, and OpenGL ES 1.x provide functions to compute the model matrix.)

The model matrix is usually computed by combining 4×4 matrices of elementary transformations of objects, in particular translations, rotations, and scalings. Specifically, in the case of a hierarchical scene graph, the transformations of all parent groups (parent, grandparent etc.) of an object are combined to form the model matrix. Let's look at the most important elementary transformations and their matrices.

The 4×4 matrix representing the translation by a vector $\mathbf{t} = (t_1, t_2, t_3)$ is:

$$\mathrm{M}_{\text{translation}} = \begin{bmatrix} 1 & 0 & 0 & t_1 \\ 0 & 1 & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The 4×4 matrix representing the scaling by a factor s_x along the x axis, s_y along the y axis, and s_z along the z axis is:

$$\mathrm{M}_{\text{scaling}} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The 4×4 matrix representing the rotation by an angle α about a normalized axis (x, y, z) is:

$$\mathrm{M}_{\text{rotation}} = \begin{bmatrix} (1-\cos\alpha)xx + \cos\alpha & (1-\cos\alpha)xy - z\sin\alpha & (1-\cos\alpha)zx + y\sin\alpha & 0 \\ (1-\cos\alpha)xy + z\sin\alpha & (1-\cos\alpha)yy + \cos\alpha & (1-\cos\alpha)yz - x\sin\alpha & 0 \\ (1-\cos\alpha)zx - y\sin\alpha & (1-\cos\alpha)yz + x\sin\alpha & (1-\cos\alpha)zz + \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Special cases for rotations about particular axes can be easily derived. These are necessary, for example, to implement rotations for Euler angles. There are, however, multiple conventions for Euler angles, which won't be discussed here.

A normalized quaternion (w_q, x_q, y_q, z_q) corresponds to a rotation by the angle $2\arccos(w_q)$. The direction of the rotation axis can be determined by normalizing the 3D vector (x_q, y_q, z_q).

Further elementary transformations exist, but are of less interest for the computation of the model matrix. The 4×4 matrices of these or other transformations are

combined by matrix products. Suppose the matrices M_1, M_2, and M_3 are applied to an object in this particular order. (M_1 might represent the transformation from object coordinates to the coordinate system of the parent group; M_2 the transformation from the parent group to the grandparent group; and M_3 the transformation from the grandparent group to world coordinates.) Then the combined matrix product is:

$M_{combined} = M_3 M_2 M_1$

Note that the order of the matrix factors is important. Also note that this matrix product should be read from the right (where vectors are multiplied) to the left, i.e. M_1 is applied first while M_3 is applied last.

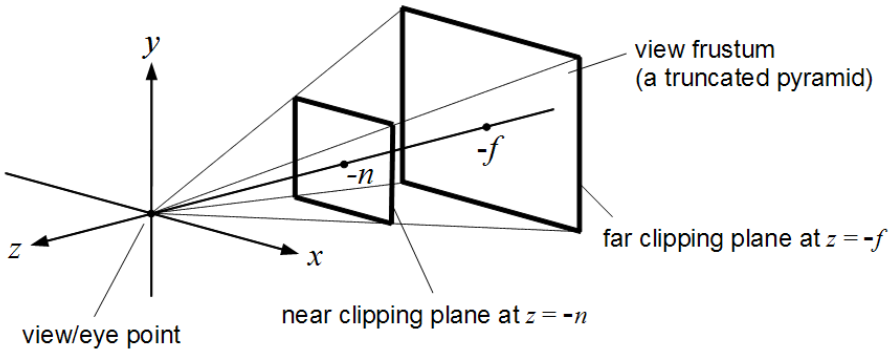

Figure 77

43.0.239. Viewing Transformation

The viewing transformation corresponds to placing and orienting the camera (or the eye of an observer). However, the best way to think of the viewing transformation is that it transforms the world coordinates into the view coordinate system (also: eye coordinate system) of a camera that is placed at the origin of the coordinate system, points to the **negative** z axis and is put on the xz plane, i.e. the up-direction is given by the positive y axis.

Accessing the View Matrix in a Vertex Shader

Similarly to the modeling transformation, the viewing transformation is represented by a 4×4 matrix, which is called view matrix $M_{world\to view}$. It can be defined as a uniform variable for the vertex shader; however, it is usually combined with the

model matrix $M_{object \to world}$ to form the modelview matrix $M_{object \to view}$. (In some versions of OpenGL (ES), a built-in uniform variable `gl_ModelViewMatrix` is available in the vertex shader.) Since the model matrix is applied first, the correct combination is:

$$M_{object \to view} = M_{world \to view} M_{object \to world}$$

(See also GLSL PROGRAMMING/APPLYING MATRIX TRANSFORMATIONS[4].)

Computing the View Matrix

Analogously to the model matrix, GLSL programmers don't have to worry about the computation of the view matrix since it is provided to the vertex shader in the form of a uniform variable. However, when developing applications in modern versions of OpenGL and OpenGL ES or in WebGL, it is necessary to compute the view matrix. (In older versions of OpenGL this is usually achieved by a utility function called `gluLookAt`.)

Here, we briefly summarize how the view matrix $M_{world \to view}$ can be computed from the position $\mathbf{t}$ of the camera, the view direction $\mathbf{d}$, and a world-up vector $\mathbf{k}$ (all in world coordinates). The steps are straightforward:

1. Compute (in world coordinates) the direction $\mathbf{z}$ of the z axis of the view coordinate system as the negative normalized $\mathbf{d}$ vector:

$$\mathbf{z} = -\frac{\mathbf{d}}{|\mathbf{d}|}$$

2. Compute (again in world coordinates) the direction $\mathbf{x}$ of the x axis of the view coordinate system by:

$$\mathbf{x} = \frac{\mathbf{d} \times \mathbf{k}}{|\mathbf{d} \times \mathbf{k}|}$$

3. Compute (still in world coordinates) the direction $\mathbf{y}$ of the y axis of the view coordinate system:

$$\mathbf{y} = \mathbf{z} \times \mathbf{x}$$

Using $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$, and $\mathbf{t}$, the inverse view matrix $M_{view \to world}$ can be easily determined because this matrix maps the origin (0,0,0) to $\mathbf{t}$ and the unit vectors (1,0,0), (0,1,0) and (0,0,1) to $\mathbf{x}$, $\mathbf{y}$,, $\mathbf{z}$. Thus, the latter vectors have to be in the columns of the matrix $M_{view \to world}$:

4 Chapter 45 on page 429

$$\mathrm{M}_{\text{view}\to\text{world}} = \begin{bmatrix} x_1 & y_1 & z_1 & t_1 \\ x_2 & y_2 & z_2 & t_2 \\ x_3 & y_3 & z_3 & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

However, we require the matrix $\mathrm{M}_{\text{world}\to\text{view}}$; thus, we have to compute the inverse of the matrix $\mathrm{M}_{\text{view}\to\text{world}}$. Note that the matrix $\mathrm{M}_{\text{view}\to\text{world}}$ has the form

$$\mathrm{M}_{\text{view}\to\text{world}} = \begin{bmatrix} \mathrm{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

with a 3×3 matrix R and a 3D vector $\mathbf{t}$. The inverse of such a matrix is:

$$\mathrm{M}^{-1}_{\text{view}\to\text{world}} = \mathrm{M}_{\text{world}\to\text{view}} = \begin{bmatrix} \mathrm{R}^{-1} & -\mathrm{R}^{-1}\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

Since in this particular case the matrix R is orthogonal (because its column vectors are normalized and orthogonal to each other), the inverse of R is just the transpose, i.e. the fourth step is to compute:

$$\mathrm{M}_{\text{world}\to\text{view}} = \begin{bmatrix} \mathrm{R}^T & -\mathrm{R}^T\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \quad \text{with } \mathrm{R} = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{bmatrix}$$

While the derivation of this result required some knowledge of linear algebra, the resulting computation only requires basic vector and matrix operations and can be easily programmed in any common programming language.

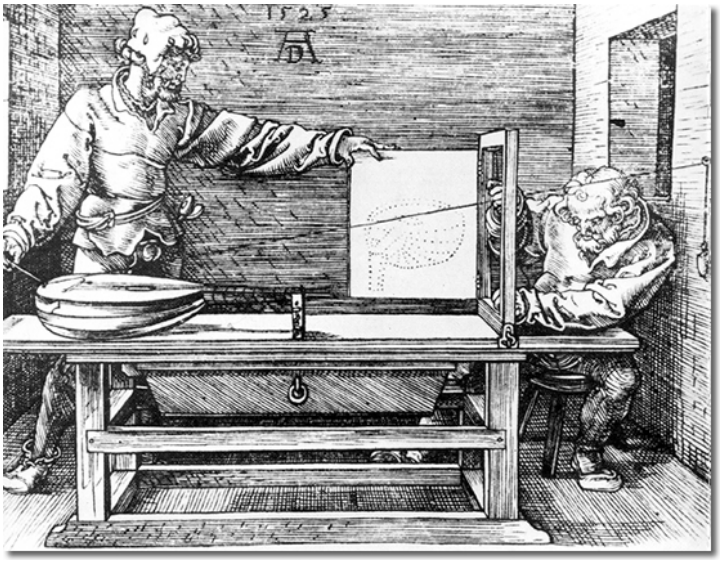

Figure 78

43.0.240. Projection Transformation and Perspective Division

First of all, the projection transformations determine the kind of projection, e.g. perspective or orthographic. Perspective projection corresponds to linear perspective with foreshortening, while orthographic projection is an orthogonal projection without foreshortening. The foreshortening is actually accomplished by the perspective division; however, all the parameters controlling the perspective projection are set in the projection transformation.

Technically spoken, the projection transformation transforms view coordinates to clip coordinates. (All parts of primitives that are outside the visible part of the scene are clipped away in clip coordinates.) It should be the last transformation that is applied to a vertex in a vertex shader before the vertex is returned in `gl_Position`. These clip coordinates are then transformed to normalized device coordinates by the **perspective division**, which is just a division of all coordinates by the fourth coordinate. (Normalized device coordinates are called this way because their values are between -1 and +1 for all points in the visible part of the scene.)

Accessing the Projection Matrix in a Vertex Shader

Similarly to the modeling transformation and the viewing transformation, the projection transformation is represented by a 4×4 matrix, which is called projection matrix $M_{projection}$. It is usually defined as a uniform variable for the vertex shader. (In some versions of OpenGL (ES), a built-in uniform variable `gl_Projection` is available in the vertex shader; see also GLSL PROGRAMMING/APPLYING MATRIX TRANSFORMATIONS[5].)

Computing the Projection Matrix

Analogously to the modelview matrix, GLSL programmers don't have to worry about the computation of the projection matrix. However, when developing applications in modern versions of OpenGL and OpenGL ES or in WebGL, it is necessary to compute the projection matrix. In older versions of OpenGL this is usually achieved with the functions `gluPerspective`, `glFrustum`, or `glOrtho`.

Here, we present the projection matrices for three cases:

5 Chapter 45 on page 429

- standard perspective projection (corresponds to `gluPerspective`)
- oblique perspective projection (corresponds to `glFrustum`)
- orthographic projection (corresponds to `glOrtho`)

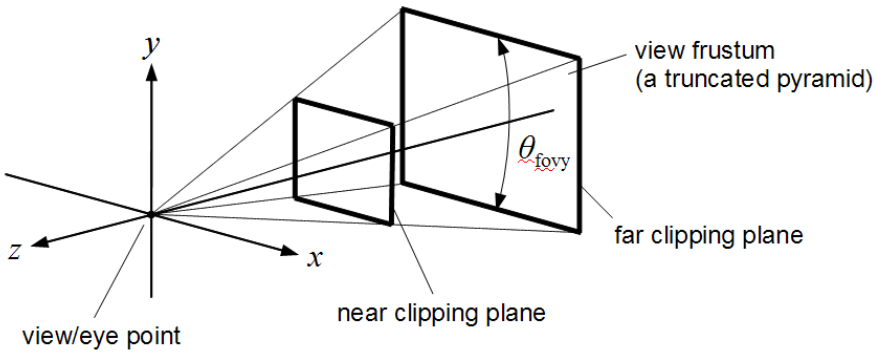

Figure 79

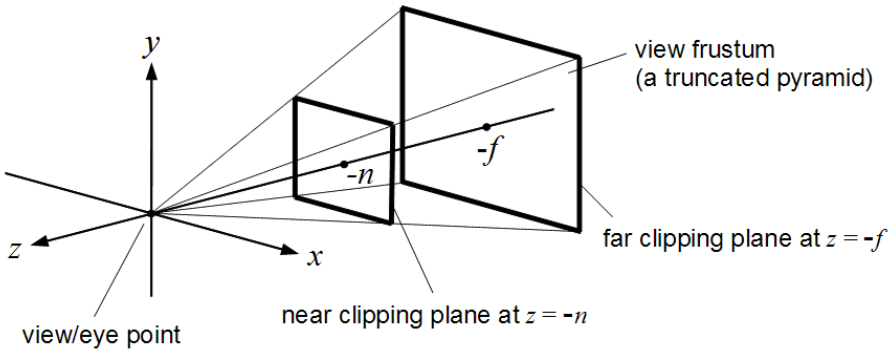

Figure 80

The **standard perspective projection** is characterized by

- an angle θ_{fovy} that specifies the field of view in y direction as illustrated in the figure below ,
- the distance n to the near clipping plane and the distance f to the far clipping plane as illustrated in the next figure,
- the aspect ratio a of the width to the height of a centered rectangle on the near clipping plane.

Together with the view point and the clipping planes, this centered rectangle defines the view frustum, i.e. the region of the 3D space that is visible for the specific projection transformation. All primitives and all parts of primitives that are outside of the view frustum are clipped away. The near and front clipping planes are necessary because depth values are stored with a finite precision; thus, it is not possible to cover an infinitely large view frustum.

With the parameters θ_{fovy}, a, n, and f, the projection matrix $\mathrm{M}_{\text{projection}}$ for the perspective projection is:

$$\mathrm{M}_{\text{projection}} = \begin{bmatrix} \frac{d}{a} & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad \text{with } d = \frac{1}{\tan(\theta_{\text{fovy}}/2)}$$

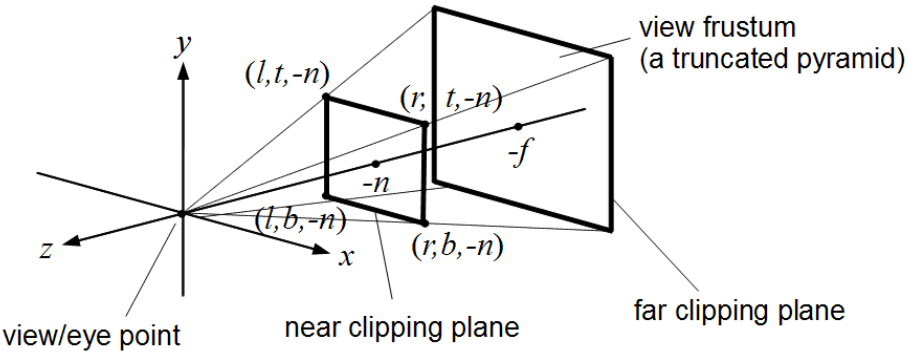

Figure 81

The **oblique perspective projection** is characterized by

- the same distances n and f to the clipping planes as in the case of the standard perspective projection,
- coordinates r (right), l (left), t (top), and b (bottom) as illustrated in the corresponding figure. These coordinates determine the position of the front rectangle of the view frustum; thus, more view frustums (e.g. off-center) can be specified than with the aspect ratio a and the field-of-view angle θ_{fovy}.

Given the parameters n, f, r, l, t, and b, the projection matrix $\mathrm{M}_{\text{projection}}$ for the oblique perspective projection is:

$$\mathrm{M}_{\text{projection}} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

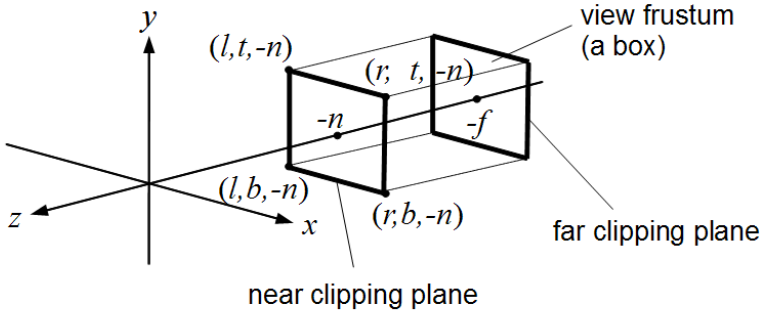

Figure 82

An **orthographic projection** without foreshortening is illustrated in the figure below . The parameters are the same as in the case of the oblique perspective projection; however, the view frustum (more precisely, the view volume) is now simply a box instead of a truncated pyramid.

With the parameters n, f, r, l, t, and b, the projection matrix $\mathrm{M}_{\text{projection}}$ for the orthographic projection is:

$$\mathrm{M}_{\text{projection}} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

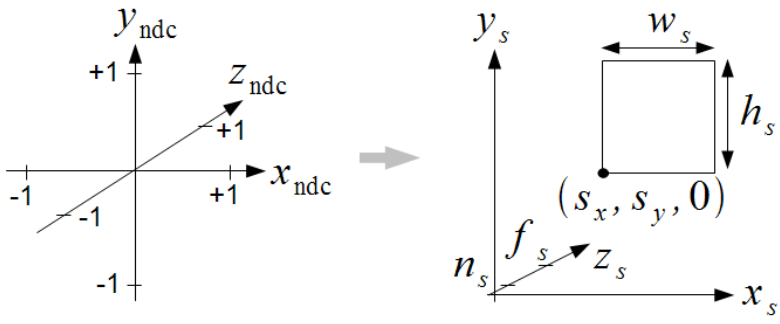

Figure 83

43.0.241. Viewport Transformation

The projection transformation maps view coordinates to clip coordinates, which are then mapped to normalized device coordinates by the perspective division by the fourth component of the clip coordinates. In normalized device coordinates (ndc), the view volume is always a box centered around the origin with the coordinates inside the box between -1 and +1. This box is then mapped to screen coordinates (also called window coordinates) by the viewport transformation as illustrated in the corresponding figure. The parameters for this mapping are the coordinates s_x and s_y of the lower, left corner of the viewport (the rectangle of the screen that is rendered) and its width w_s and height h_s, as well as the depths n_s and f_s of the front and near clipping planes. (These depths are between 0 and 1). In OpenGL and OpenGL ES, these parameters are set with two functions:

`glViewport(GLint` s_x`, GLint` s_y`, GLsizei` w_s`, GLsizei` h_s`);`

`glDepthRangef(GLclampf` n_s`, GLclampf` f_s`);`

The matrix of the viewport transformation isn't very important since it is applied automatically in a fixed-function stage. However, here it is for the sake of completeness:

$$\begin{bmatrix} \frac{w_s}{2} & 0 & 0 & s_x + \frac{w_s}{2} \\ 0 & \frac{h_s}{2} & 0 & s_y + \frac{h_s}{2} \\ 0 & 0 & \frac{f_s - n_s}{2} & \frac{n_s + f_s}{2} \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

43.0.242. Further Reading

The conventional vertex transformations described here are defined in full detail in Section 2.12 of the "OpenGL 4.1 Compatibility Profile Specification" available at the KHRONOS OPENGL WEB SITE[6].

A more accessible description of the vertex transformations is given in Chapter 3 (on viewing) of the book "OpenGL Programming Guide" by Dave Shreiner published by Addison-Wesley. (An older edition is available ONLINE[7]).

6 HTTP://WWW.KHRONOS.ORG/OPENGL/

7 HTTP://WWW.GLPROGRAMMING.COM/RED/CHAPTER03.HTML

44. Vector and Matrix Operations

The syntax of GLSL is very similar to C (and therefore C++ and Java); however, there are built-in data types and functions for floating-point vectors and matrices, which are specific to GLSL. These are discussed here. A full description of GLSL can be found in the literature in the "FURTHER READING"[1] section.

44.0.243. Data Types

In GLSL, the types `vec2`, `vec3`, and `vec4` represent 2D, 3D, and 4D floating-point vectors. (There are also types for integer and boolean vectors, which are not discussed here.) Vector variables are defined as you would expect if C, C++ or Java had these types:

```
vec2 a2DVector;
vec3 three_dimensional_vector;
vec4 vector4;
```

The data types for floating-point 2×2, 3×3, and 4×4 matrices are: `mat2`, `mat3`, and `mat4`:

```
mat2 m2x2;
mat3 linear_mapping;
mat4 trafo;
```

44.0.244. Constructors

Vectors can be initialized and converted by constructors of the same name as the data type:

```
vec2 a = vec2(1.0, 2.0);
vec3 b = vec3(-1., 0., 0.);
vec4 c = vec4(0.0, 0.0, 0.0, 1.0);
```

Note that some GLSL compilers will complain if integers are used to initialize floating-point vectors; thus, it is good practice to always include the decimal point.

1 Chapter 44.0.248 on page 427

One can also use one floating-point number in the constructor to set all components to the same value:

```
vec4 a = vec4(0.0); // = vec4(0.0, 0.0, 0.0, 0.0)
```

Casting a higher-dimensional vector to a lower-dimensional vector is also achieved with these constructors:

```
vec4 a = vec4(-1.0, 2.5, 4.0, 1.0);
vec3 b = vec3(a); // = vec3(-1.0, 2.5, 4.0)
vec2 c = vec2(b); // = vec2(-1.0, 2.5)
```

Casting a lower-dimensional vector to a higher-dimensional vector is achieved by supplying these constructors with the correct number of components:

```
vec2 a = vec2(0.1, 0.2);
vec3 b = vec3(0.0, a); // = vec3(0.0, 0.1, 0.2)
vec4 c = vec4(b, 1.0); // = vec4(0.0, 0.1, 0.2, 1.0)
```

Similarly, matrices can be initialized and constructed. Note that the values specified in a matrix constructor are consumed to fill the first column, then the second column, etc.:

```
mat3 m = mat3(
   1.1, 2.1, 3.1, // first column (not row!)
   1.2, 2.2, 3.2, // second column
   1.3, 2.3, 2.3  // third column
);
mat3 id = mat3(1.0); // puts 1.0 on the diagonal
                     // all other components are 0.0
vec3 column0 = vec3(0.0, 1.0, 0.0);
vec3 column1 = vec3(1.0, 0.0, 0.0);
vec3 column2 = vec3(0.0, 0.0, 1.0);
mat3 n = mat3(column0, column1, column2); // sets columns of matrix n
```

If a larger matrix is constructed from a smaller matrix, the additional rows and columns are set to the values they would have in an identity matrix:

```
mat2 m2x2 = mat2(
   1.1, 2.1,
   1.2, 2.2
);
mat3 m3x3 = mat3(m2x2); // = mat3(
   // 1.1, 2.1, 0.0,
   // 1.2, 2.2, 0.0,
   // 0.0, 0.0, 1.0)
mat2 mm2x2 = mat2(m3x3); // = m2x2
```

If a smaller matrix is constructed from a larger matrix, the top, left submatrix of the larger matrix is chosen, e.g. in the last line of the previous example.

44.0.245. Components

Components of vectors are accessed by array indexing with the `[]`-operator (indexing starts with 0) or with the `.`-operator and the element names `x, y, z, w` or `r, g, b, a` or `s, t, p, q`:

```
vec4 v = vec4(1.1, 2.2, 3.3, 4.4);
float a = v[3]; // = 4.4
float b = v.w; // = 4.4
float c = v.a; // = 4.4
float d = v.q; // = 4.4
```

It is also possible to construct new vectors by extending the `.`-notation:

```
vec4 v = vec4(1.1, 2.2, 3.3, 4.4);
vec3 a = v.xyz; // = vec3(1.1, 2.2, 3.3)
vec3 b = v.bgr; // = vec3(3.3, 2.2, 1.1)
vec2 c = v.tt; // = vec2(2.2, 2.2)
```

Matrices are considered to consist of column vectors, which are accessed by array indexing with the `[]`-operator. Elements of the resulting (column) vector can be accessed as discussed above:

```
mat3 m = mat3(
   1.1, 2.1, 3.1, // first column
   1.2, 2.2, 3.2, // second column
   1.3, 2.3, 3.3  // third column
);
vec3 column2 = m[2]; // = vec3(1.3, 2.3, 3.3)
vec3 m20 = m[2][0]; // = 1.3
vec3 m21 = m[2].y; // = 2.3
```

44.0.246. Operators

If the binary operators `*, /, +, -, =, *=, /=, +=, -=` are used between vectors of the same type, they just work component-wise:

```
vec3 a = vec3(1.0, 2.0, 3.0);
vec3 b = vec3(0.1, 0.2, 0.3);
vec3 c = a + b; // = vec3(1.1., 2.2, 3.3)
vec3 d = a * b; // = vec3(0.1, 0.4, 0.9)
```

Note in particular that `a * b` represents a component-wise product of two vectors, which is not often seen in linear algebra.

For matrices, these operators also work component-wise, **except** for the `*`-operator, which represents a matrix-matrix product, e.g.:

$$\mathrm{AB} = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} = \begin{bmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} & a_{1,1}b_{1,2} + a_{1,2}b_{2,2} \\ a_{2,1}b_{1,1} + a_{2,2}b_{2,1} & a_{2,1}b_{1,2} + a_{2,2}b_{2,2} \end{bmatrix}$$

And in GLSL:

```
mat2 a = mat2(1., 2.,  3., 4.);
mat2 b = mat2(10., 20.,  30., 40.);
mat2 c = a * b; // = mat2(
   // 1. * 10. + 3. * 20., 2. * 10. + 4. * 20.,
   // 1. * 30. + 3. * 40., 2. * 30. + 4. * 40.)
```

For a component-wise matrix product, the built-in function `matrixCompMult` is provided.

The $\star$-operator can also be used to multiply a floating-point value (i.e. a scalar) to all components of a vector or matrix (from left or right):

```
vec3 a = vec3(1.0, 2.0, 3.0);
mat3 m = mat3(1.0);
float s = 10.0;
vec3 b = s * a; // vec3(10.0, 20.0, 30.0)
vec3 c = a * s; // vec3(10.0, 20.0, 30.0)
mat3 m2 = s * m; // = mat3(10.0)
mat3 m3 = m * s; // = mat3(10.0)
```

Furthermore, the $\star$-operator can be used for matrix-vector products of the corresponding dimension, e.g.:

$$\mathrm{M}\mathbf{v} = \begin{bmatrix} m_{1,1} & m_{1,2} \\ m_{2,1} & m_{2,2} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} m_{1,1}v_1 + m_{1,2}v_2 \\ m_{2,1}v_1 + m_{2,2}v_2 \end{bmatrix}$$

And in GLSL:

```
vec2 v = vec2(10., 20.);
mat2 m = mat2(1., 2.,  3., 4.);
vec2 w = m * v; // = vec2(1. * 10. + 3. * 20., 2. * 10. + 4. * 20.)
```

Note that the vector has to be multiplied to the matrix from the right.

If a vector is multiplied to a matrix **from the left**, the result corresponds to multiplying a row vector from the left to the matrix. This corresponds to multiplying a column vector to the **transposed** matrix from the right:

$$\mathbf{v}^T\mathrm{M} = \left(\mathrm{M}^T\mathbf{v}\right)^T$$

In components:

$$\mathbf{v}^T\mathrm{M} = \begin{bmatrix} v_1 & v_2 \end{bmatrix} \begin{bmatrix} m_{1,1} & m_{1,2} \\ m_{2,1} & m_{2,2} \end{bmatrix} = \begin{bmatrix} v_1m_{1,1} + v_2m_{2,1} & v_1m_{1,2} + v_2m_{2,2} \end{bmatrix}$$
$$= \left(\begin{bmatrix} m_{1,1} & m_{2,1} \\ m_{1,2} & m_{2,2} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \right)^T = \left(\mathrm{M}^T\mathbf{v}\right)^T$$

Thus, multiplying a vector from the left to a matrix corresponds to multiplying it from the right to the transposed matrix:

```
vec2 v = vec2(10., 20.);
mat2 m = mat2(1., 2.,  3., 4.);
vec2 w = v * m; // = vec2(1. * 10. + 2. * 20., 3. * 10. + 4. * 20.)
```

Since there is no built-in function to compute a transposed matrix, this technique is extremely useful: whenever a vector should be multiplied with a transposed matrix, one can just multiply it from the left to the original matrix. Several applications of this technique are described in GLSL PROGRAMMING/UNITY/APPLYING MATRIX[2].

44.0.247. Built-In Vector and Matrix Functions

Component-Wise Functions

As mentioned, the function

```
TYPE matrixCompMult(TYPE a, TYPE b) // component-wise matrix product
```

computes a component-wise product for the matrix types `mat2`, `mat3` and `mat4`, which are denoted as `TYPE`.

The following functions work component-wise for variables of type `float`, `vec2`, `vec3` and `vec4`, which are denoted as `TYPE`:

```
TYPE min(TYPE a, TYPE b) // returns a if a < b, b otherwise
TYPE min(TYPE a, float b) // returns a if a < b, b otherwise
TYPE max(TYPE a, TYPE b) // returns a if a > b, b otherwise
TYPE max(TYPE a, float b) // returns a if a > b, b otherwise
TYPE clamp(TYPE a, TYPE minVal, TYPE maxVal)
   // = min(max(x, minVal), maxVal)
TYPE clamp(TYPE a, float minVal, float maxVal)
   // = min(max(x, minVal), maxVal)
TYPE mix(TYPE a, TYPE b, TYPE wb) // = a * (TYPE(1.0) - wb) + b * wb
TYPE mix(TYPE a, TYPE b, float wb) // = a * (TYPE(1.0) - wb) + b * wb
```

There are more built-in functions, which also work component-wise but are less useful for vectors, e.g., `abs`, `sign`, `floor`, `ceil`, `fract`, `mod`, `step`, `smoothstep`, `sqrt`, `inversesqrt`, `pow`, `exp`, `exp2`, `log`, `log2`, `radians` (converts degrees to radians), `degrees` (converts radians to degrees), `sin`, `cos`, `tan`, `asin`, `acos`, `atan` (with one argument and with two arguments for signed numerator and signed denominator), `lessThan`, `lessThanEqual`, `greaterThan`, `greaterThanEqual`, `equal`, `notEqual`, and `not`.

2 HTTP://EN.WIKIBOOKS.ORG/WIKI/GLSL%20PROGRAMMING%2FUNITY%2FAPPLYING%20MATRIX

Geometric Functions

The following functions are particular useful for vector operations. `TYPE` is any of: `float, vec2, vec3` and `vec4` (only one of them per line).

```
vec3 cross(vec3 a, vec3 b) // = vec3(a[1] * b[2] - a[2] * b[1],
   // a[2] * b[0] - a[0] * b[2],
   // a[0] * b[1] - a[1] * b[0])
float dot(TYPE a, TYPE b) // = a[0] * b[0] + a[1] * b[1] + ...
float length(TYPE a) // = sqrt(dot(a, a))
float distance(TYPE a, TYPE b) // = length(a - b)
TYPE normalize(TYPE a) // = a / length(a)
TYPE faceforward(TYPE n, TYPE i, TYPE nRef)
   // returns n if dot(nRef, i) < 0, -n otherwise
TYPE reflect(TYPE i, TYPE n) // = i - 2. * dot(n, i) * n
   // this computes the reflection of vector 'i'
   // at a plane of normalized(!) normal vector 'n'
```

Functions for Physics

The function

```
TYPE refract(TYPE i, TYPE n, float r)
```

computes the direction of a refracted ray if `i` specifies the normalized(!) direction of the incoming ray and `n` specifies the normalized(!) normal vector of the interface of two optical media (e.g. air and water). The vector `n` should point to the side from where `i` is coming, i.e. the dot product of `n` and `i` should be negative. The floating-point number `r` is the ratio of the refractive index of the medium from where the ray comes to the refractive index of the medium on the other side of the surface. Thus, if a ray comes from air (refractive index about 1.0) and hits the surface of water (refractive index 1.33), then the ratio `r` is 1.0 / 1.33 = 0.75. The computation of the function is:

```
float d = 1.0 - r * r * (1.0 - dot(n, i) * dot(n, i));
if (d < 0.0) return TYPE(0.0); // total internal reflection
return r * i - (r * dot(n, i) + sqrt(d)) * n;
```

As the code shows, the function returns a vector of length 0 in the case of total internal reflection (SEE THE ENTRY IN WIKIPEDIA)[3], i.e. if the ray does not pass the interface between the two materials.

3 HTTP://EN.WIKIPEDIA.ORG/WIKI/TOTAL%20INTERNAL%20REFLECTION

44.0.248. Further Reading

All details of GLSL for OpenGL are specified in the "OpenGL Shading Language 4.10.6 Specification" available at the KHRONOS OPENGL WEB SITE[4].

More accessible descriptions of the OpenGL shading language for OpenGL are given in recent editions of many books about OpenGL, e.g. in Chapter 15 of the "OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1" (7th ed.) by Dave Shreiner published 2009 by Addison-Wesley.

All details of GLSL for OpenGL ES 2.0 are specified in the "OpenGL ES Shading Language 1.0.17 Specification" available at the "KHRONOS OPENGL ES API REGISTRY"[5].

A more accessible description of the OpenGL ES shading language is given in Chapter 5 and Appendix B of the book "OpenGL ES 2.0 Programming Guide" by Aaftab Munshi, Dan Ginsburg and Dave Shreiner published by Addison-Wesley (see ITS WEB SITE[6]).

4 HTTP://WWW.KHRONOS.ORG/OPENGL/
5 HTTP://WWW.KHRONOS.ORG/REGISTRY/GLES/
6 HTTP://WWW.OPENGLES-BOOK.COM/

45. Applying Matrix Transformations

Applying the conventional VERTEX TRANSFORMATIONS[1] or any other transformations that are represented by matrices in shaders is usually accomplished by specifying the corresponding matrix in a uniform variable of the shader and then multiplying the matrix with a vector. There are, however, some differences in the details. Here, we discuss the transformation of points (i.e. 4D vectors with a 4th coordinates equal to 1), the transformation of directions (i.e. vectors in the strict sense: 3D vectors or 4D vectors with a 4th coordinate equal to 0), and the transformation of surface normal vectors (i.e. vectors that specify a direction that is orthogonal to a plane).

This section assumes some knowledge of the syntax of GLSL as described in GLSL PROGRAMMING/VECTOR AND MATRIX OPERATIONS[2].

45.0.249. Transforming Points

For points, transformations usually are represented by a 4×4 matrices since they might include a translation by a 3D vector $\mathbf{t}$ in the 4th column:

$$\mathrm{M} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & t_1 \\ a_{2,1} & a_{2,2} & a_{2,3} & t_2 \\ a_{3,1} & a_{3,2} & a_{3,3} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{with A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \quad \text{and } \mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}$$

(Projection matrices will also include additional values unequal to 0 in the last row.)

Three-dimensional points are represented by four-dimensional vectors with the 4th coordinate equal to 1:

$$P = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{bmatrix}$$

1 HTTP://EN.WIKIBOOKS.ORG/WIKI/%2FVERTEX%20TRANSFORMATIONS

2 Chapter 44 on page 421

In order to apply the transformation, the matrix M is multiplied to the vector P:

$$\mathrm{M}P = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & t_1 \\ a_{2,1} & a_{2,2} & a_{2,3} & t_2 \\ a_{3,1} & a_{3,2} & a_{3,3} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{bmatrix} = \begin{bmatrix} a_{1,1}p_1 + a_{1,2}p_2 + a_{1,3}p_3 + t_1 \\ a_{2,1}p_1 + a_{2,2}p_2 + a_{2,3}p_3 + t_2 \\ a_{3,1}p_1 + a_{3,2}p_2 + a_{3,3}p_3 + t_3 \\ 1 \end{bmatrix}$$

The GLSL code to apply a 4×4 matrix to a point represented by a 4D vector is straightforward:

```
mat4 matrix;
vec4 point;
vec4 transformed_point = matrix * point;
```

45.0.250. Transforming Directions

Directions in three dimensions are represented either by a 3D vector or by a 4D vector with 0 as the fourth coordinate. (One can think of them as points at infinity; similar to a point at the horizon of which we cannot tell the position in space but only the direction in which to find it.)

In the case of a 3D vector, we can either transform it by multiplying it with a 3×3 matrix:

```
mat3 matrix;
vec3 direction;
vec3 transformed_direction = matrix * direction;
```

or with a 4×4 matrix, if we convert the 3D vector to a 4D vector with a 4th coordinate equal to 0:

```
mat4 matrix;
vec3 direction;
vec3 transformed_direction = vec3(matrix * vec4(direction, 0.0));
```

Alternatively, the 4×4 matrix can also be converted to a 3×3 matrix.

On the other hand, a 4D vector can be multiplied directly with a 4×4 matrix. It can also be converted to a 3D vector in order to multiply it with a 3×3 matrix:

```
mat3 matrix;
vec4 direction; // 4th component is 0
vec4 transformed_direction = vec4(matrix * vec3(direction), 0.0);
```

45.0.251. Transforming Normal Vectors

Similarly to directions, surface normal vectors (or “normal vectors” for short) are represented by 3D vectors or 4D vectors with 0 as the 4th component. However, they

transform differently. (The mathematical reason is that they represent something that is called a covector, covariant vector, one-form, or linear functional.)

To understand the transformation of normal vectors, consider the main feature of a surface normal vector: it is orthogonal to a surface. Of course, this feature should still be true under transformations, i.e. the transformed normal vector should be orthogonal to the transformed surface. If the surface is being represented locally by a tangent vector, this feature requires that a transformed normal vector is orthogonal to a transformed direction vector if the original normal vector is orthogonal to the original direction vector.

Mathematically spoken, a normal vector $\mathbf{n}$ is orthogonal to a direction vector $\mathbf{v}$ if their dot product is 0. It turns out that if $\mathbf{v}$ is transformed by a 3×3 matrix A, the normal vector has to be transformed by the **transposed inverse** of A: $(\mathrm{A}^{-1})^T$. We can easily test this by checking the dot product of the transformed normal vector $(\mathrm{A}^{-1})^T\mathbf{n}$ and the transformed direction vector $\mathrm{A}\mathbf{v}$:

$$\left(\mathrm{A}^{-1}\right)^T\mathbf{n}\cdot\mathrm{A}\mathbf{v} = \left(\left(\mathrm{A}^{-1}\right)^T\mathbf{n}\right)^T\mathrm{A}\mathbf{v} = \left(\mathbf{n}^T\left(\left(\mathrm{A}^{-1}\right)^T\right)^T\right)\mathrm{A}\mathbf{v} = \left(\mathbf{n}^T\mathrm{A}^{-1}\right)\mathrm{A}\mathbf{v} = \mathbf{n}^T\mathrm{A}^{-1}\mathrm{A}\mathbf{v} = \mathbf{n}^T\mathbf{v} = \mathbf{n}\cdot\mathbf{v}$$

In the first step we have used $\mathbf{a}\cdot\mathbf{b} = \mathbf{a}^T\mathbf{b}$, then $(\mathrm{M}\,\mathbf{a})^T = \mathbf{a}^T\,\mathrm{M}^T$, then $(\mathrm{M}^T)^T = \mathrm{M}$, then $\mathrm{M}^{-1}\mathrm{M} = \mathrm{Id}$ (i.e. the identity matrix).

The calculation shows that the dot product of the transformed vectors is in fact the same as the dot product of the original vectors; thus, the transformed vectors are orthogonal if and only if the original vectors are orthogonal. Just the way it should be.

Thus, in order to transform normal vectors in GLSL, the **transposed inverse matrix** is often specified as a uniform variable (together with the original matrix for the transformation of directions and points) and applied as any other transformation:

```
mat3 matrix_inverse_transpose;
vec3 normal;
vec3 transformed_normal = matrix_inverse_transpose * normal;
```

In the case of a 4×4 matrix, the normal vector can be cast to a 4D vector by appending 0:

```
mat4 matrix_inverse_transpose;
vec3 normal;
vec3 transformed_normal = vec3(matrix_inverse_transpose * vec4(normal, 0.0));
```

Alternatively, the matrix can be cast to a 3×3 matrix.

If the inverse matrix is known, the normal vector can be multiplied from the left to apply the transposed inverse matrix. In general, multiplying a transposed matrix

with a vector can be easily expressed by putting the vector to the left of the matrix. The reason is that a vector-matrix product makes only sense for row vectors (i.e. transposed column vectors) and corresponds to a matrix-vector product of the transposed matrix with the corresponding column vector:

$$\mathbf{p}^T \mathrm{A} = \left(\mathrm{A}^T \mathbf{p}\right)^T$$

Since GLSL makes no distinction between column and row vectors, the result is just a vector.

Thus, in order to multiply a normal vector with the transposed inverse matrix, we can multiply it from the left to the inverse matrix:

```
mat3 matrix_inverse;
vec3 normal;
vec3 transformed_normal = normal * matrix_inverse;
```

In the case of multiplying a 4×4 matrix to a 4D normal vector (from the left or the right), it should be made sure that the 4th component of the resulting vector is 0. In fact, in several cases it is necessary to discard the computed 4th component (for example by casting the result to a 3D vector):

```
mat4 matrix_inverse;
vec4 normal;
vec4 transformed_normal = vec4(vec3(normal * matrix_inverse), 0.0);
```

Note that any normalization of the normal vector to unit length is not preserved by this transformation. Thus, normal vectors are often normalized to unit length after the transformation (e.g. with the built-in GLSL function `normalize`).

45.0.252. Transforming Normal Vectors with an Orthogonal Matrix

A special case arises when the transformation matrix A is orthogonal. In this case, the inverse of A is the transposed matrix; thus, the transposed of the inverse of A is the twice transposed matrix, which is the original matrix, i.e. for a orthogonal matrix A:

$$\left(\mathrm{A}^{-1}\right)^T = \left(\mathrm{A}^T\right)^T = \mathrm{A}$$

Thus, in the case of **orthogonal** matrices, normal vectors are transformed with the same matrix as directions and points:

```
mat3 matrix; // orthogonal matrix
vec3 normal;
vec3 transformed_normal = matrix * normal;
```

45.0.253. Transforming Points with the Inverse Matrix

Sometimes it is necessary to apply the inverse transformation. In most cases, the best solution is to define another uniform variable for the inverse matrix and set the inverse matrix in the main application. The shader can then apply the inverse matrix like any other matrix. This is by far more efficient than computing the inverse in the shader.

There is, however, a special case: If the matrix M is of the form presented above (i.e. the 4th row is (0,0,0,1)):

$$\mathrm{M} = \begin{bmatrix} \mathrm{A} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

with an orthogonal 3×3 matrix A (i.e. the row (or column) vectors of A are normalized and orthogonal to each other; for example, this is usually the case for the view transformation, see GLSL PROGRAMMING/VERTEX TRANSFORMATIONS[3]), then the inverse matrix is given by (because $\mathrm{A}^{-1} = \mathrm{A}^T$ for an orthogonal matrix A):

$$\mathrm{M}^{-1} = \begin{bmatrix} \mathrm{A}^{-1} & -\mathrm{A}^{-1}\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} \mathrm{A}^T & -\mathrm{A}^T\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

For the multiplication with a point P that is represented by the 4D vector $(p_x, p_y, p_z, 1)$ with the 3D vector $\mathbf{p} = (p_x, p_y, p_z)$ we get:

$$\mathrm{M}^{-1}P = \begin{bmatrix} \mathrm{A}^T & -\mathrm{A}^T\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathrm{A}^T\mathbf{p} - \mathrm{A}^T\mathbf{t} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathrm{A}^T(\mathbf{p} - \mathbf{t}) \\ 1 \end{bmatrix}$$

Note that the vector **t** is just the 4th column of the matrix M, which can be conveniently accessed in GLSL:

```
mat4 matrix;
vec4 last_column = matrix[3]; // indices start with 0 in GLSL
```

As mentioned above, multiplying a transposed matrix with a vector can be easily expressed by putting the vector to the left of the matrix because a vector-matrix product makes only sense for row vectors (i.e. transposed column vectors) and corresponds to a matrix-vector product of the transposed matrix with the corresponding column vector:

$$\mathbf{p}^T\mathrm{A} = \left(\mathrm{A}^T\mathbf{p}\right)^T$$

Using these features of GLSL, the term A^T(**p** - **t**) is easily and efficiently implemented as follows (note that the 4th component of the result has to be set to 1 separately):

3 Chapter 43 on page 407

```
mat4 matrix; // upper, left 3x3 matrix is orthogonal;
   // 4th row is (0,0,0,1)
vec4 point; // 4th component is 1
vec4 point_transformed_with_inverse = vec4(vec3((point - matrix[3]) * matrix),
 1.0);
```

45.0.254. Transforming Directions with the Inverse Matrix

As in the case of points, the best way to transform a direction with the inverse matrix is usually to compute the inverse matrix in the main application and communicate it to a shader via another uniform variable.

The exception is an orthogonal 3×3 matrix A (i.e. all rows (or columns) are normalized and orthogonal to each other) or a 4×4 matrix M of the form

$$\mathrm{M} = \begin{bmatrix} \mathrm{A} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

where A is an orthogonal 3×3 matrix. In these cases, the inverse matrix A^{-1} is equal to the transposed matrix A^T.

As discussed above, the best way in GLSL to multiply a vector with the transposed matrix is to multiply it from the left to the original matrix because this is interpreted as a product of a row vector with the original matrix, which corresponds to the product of the transposed matrix with the column vector:

$$\mathbf{p}^T\mathrm{A} = \left(\mathrm{A}^T\mathbf{p}\right)^T$$

Thus, the transformation with the transposed matrix (i.e. the inverse in case of a orthogonal matrix) is written as:

```
mat4 matrix; // upper, left 3x3 matrix is orthogonal
vec4 direction; // 4th component is 0
vec4 direction_transformed_with_inverse = vec4(vec3(direction * matrix), 0.0);
```

Note that the 4th component of the result has to be set to 0 separately since the 4th component of `direction * matrix` is not meaningful for the transformation of directions. (It is, however, meaningful for the transformation of plane equations, which are not discussed here.)

The versions for 3x3 matrices and 3D vectors only require different cast operations between 3D and 4D vectors.

45.0.255. Transforming Normal Vectors with the Inverse Transformation

Suppose the inverse matrix M^{-1} is available, but the transformation corresponding to M is required. Moreover, we want to apply this transformation to a normal vector. In this case, we can just apply the transpose of the inverse by multiplying the normal vector from the left to the inverse matrix (as discussed above):

```
mat4 matrix_inverse;
vec3 normal;
vec3 transformed_normal = vec3(vec4(normal, 0.0) * matrix_inverse);
```

(or by casting the matrix).

45.0.256. Built-In Matrix Transformations

Some frameworks (in particular the OpenGL compatibility profile but neither the OpenGL core profile nor OpenGL ES 2.x) provide several built-in uniforms to access certain vertex transformations in GLSL shaders. They should not be declared, but here are the declarations to specify their types:

```
uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ProjectionMatrix;
uniform mat4 gl_ModelViewProjectionMatrix;
uniform mat4 gl_TextureMatrix[gl_MaxTextureCoords];
uniform mat3 gl_NormalMatrix; // transpose of the inverse of the
   // upper left 3x3 matrix of gl_ModelViewMatrix
uniform mat4 gl_ModelViewMatrixInverse;
uniform mat4 gl_ProjectionMatrixInverse;
uniform mat4 gl_ModelViewProjectionMatrixInverse;
uniform mat4 gl_TextureMatrixInverse[gl_MaxTextureCoords];
uniform mat4 gl_ModelViewMatrixTranspose;
uniform mat4 gl_ProjectionMatrixTranspose;
uniform mat4 gl_ModelViewProjectionMatrixTranspose;
uniform mat4 gl_TextureMatrixTranspose[gl_MaxTextureCoords];
uniform mat4 gl_ModelViewMatrixInverseTranspose;
uniform mat4 gl_ProjectionMatrixInverseTranspose;
uniform mat4 gl_ModelViewProjectionMatrixInverseTranspose;
uniform mat4 gl_TextureMatrixInverseTranspose[gl_MaxTextureCoords];
```

45.0.257. Further Reading

The transformation of normal vectors is described in Section 2.12.2 of the "OpenGL 4.1 Compatibility Profile Specification" available at the KHRONOS OPENGL WEB SITE[4].

4 HTTP://WWW.KHRONOS.ORG/OPENGL/

A more accessible description of the transformation of normal vectors is given in Appendix E of the free HTML version of the "OpenGL Programming Guide" available ONLINE[5].

The built-in uniforms of matrix transformations are described in Section 7.4.1 of the "OpenGL Shading Language 4.10.6 Specification" available at the KHRONOS OPENGL WEB SITE[6].

5 HTTP://WWW.GLPROGRAMMING.COM/RED/APPENDIXE.HTML
6 HTTP://WWW.KHRONOS.ORG/OPENGL/

46. Rasterization

Rasterization is the stage of the OPENGL (ES) 2.0 PIPELINE[1] that determines the pixels covered by a primitive (e.g. a triangle) and interpolates the output variables of the vertex shader (i.e. varying variables and depth) for each covered pixel. These interpolated varying variables and the interpolated depth are then given to the fragment shader. (In a wider sense, the term "rasterization" also includes the execution of the fragment shader and the PER-FRAGMENT OPERATIONS[2].)

Usually, it is not necessary for GLSL programmers to know more about the rasterization stage than described in the previous paragraph. However, it is useful to know some details in order to understand features such as perspectively correct interpolation and the role of the fourth component of the vertex position that is computed by the vertex shader. For some advanced algorithms in computer graphics it is also necessary to know some details of the rasterization process.

The main two parts of the rasterization are:

- Determining the pixels that are covered by a primitive (e.g. a triangle)
- Linear interpolation of varying variables and depth for all covered pixels

1 HTTP://EN.WIKIBOOKS.ORG/WIKI/%2FOPENGL%20ES%202.0%20PIPELINE
2 HTTP://EN.WIKIBOOKS.ORG/WIKI/%2FPER-FRAGMENT%20OPERATIONS

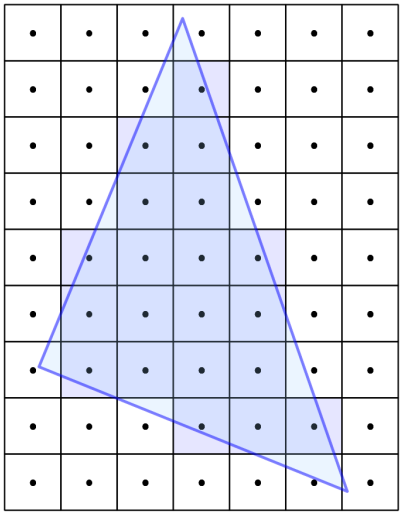

Figure 84

46.0.258. Determining Covered Pixels

In OpenGL (ES), a pixel of the framebuffer is defined as being covered by a primitive if the center of the pixel is covered by the primitive as illustrated in the diagram below .

There are certain rules for cases when the center of a pixel is exactly on the boundary of a primitive. These rules make sure that two adjacent triangles (i.e. triangles that share an edge) never share any pixels (unless they actually overlap) and never miss any pixels along the edge; i.e. each pixel along the edge between two adjacent triangles is covered by either triangle but not by both. This is important to avoid holes and (in case of semitransparent triangles) multiple rasterizations of the same pixel. The rules are, however, specific to implementations of GPUs; thus, they won't be discussed here.

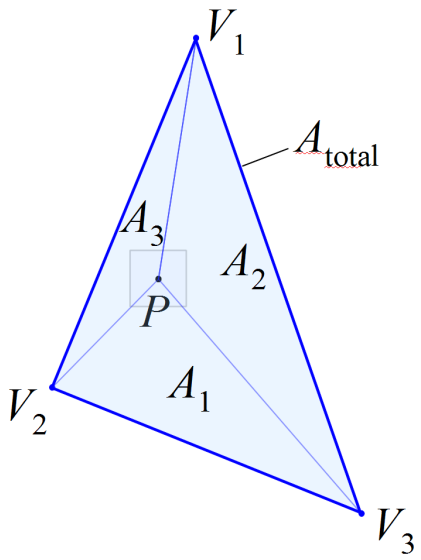

Figure 85

46.0.259. Linear Interpolation of Varying Variables

Once all the covered pixels are determined, varying variables are interpolated for each pixel. For simplicity, we will only discuss the case of a triangle. (A line works like a triangle with two vertices at the same position.)

For each triangle, the vertex shader computes the positions of the three vertices. In the diagram below , these positions are labeled as V_1, V_2, and V_3. The vertex shader also computes values of varying variables at each vertex. We denote one of them as f_1, f_2, and f_3. Note that these values refer to the same varying variable computed at different vertices. The position of the center of the pixel for which we want to interpolate the varying variables is labeled by P in the diagram.

We want to compute a new interpolated value f_P at the pixel center P from the values f_1, f_2, and f_3 at the three vertices. There are several methods to do this. One is using barycentric coordinates α_1, α_2, and α_3, which are computed this way:

$$\begin{aligned}
\alpha_1 &= \frac{A_1}{A_{\text{total}}} = \frac{\text{area of triangle } PV_2V_3}{\text{area of triangle } V_1V_2V_3} \\
\alpha_2 &= \frac{A_2}{A_{\text{total}}} = \frac{\text{area of triangle } V_1PV_3}{\text{area of triangle } V_1V_2V_3} \\
\alpha_3 &= \frac{A_3}{A_{\text{total}}} = \frac{\text{area of triangle } V_1V_2P}{\text{area of triangle } V_1V_2V_3}
\end{aligned}$$

The triangle areas A_1, A_2, A_3, and A_{total} are also shown in the diagram. In three dimensions (or two dimensions with an additional third dimension) the area of a triangle between three points Q, R, S, can be computed as one half of the length of a cross product:

$$\text{area of triangle } QRS \quad = \quad \tfrac{1}{2}|\overrightarrow{QR} \times \overrightarrow{QS}|$$

With the barycentric coordinates α_1, α_2, and α_3, the interpolation of f_P at P from the values f_1, f_2, and f_3 at the three vertices is easy:

$$f_P \quad = \quad \alpha_1 f_1 + \alpha_2 f_2 + \alpha_3 f_3$$

This way, all varying variables can be linearly interpolated for all covered pixels.

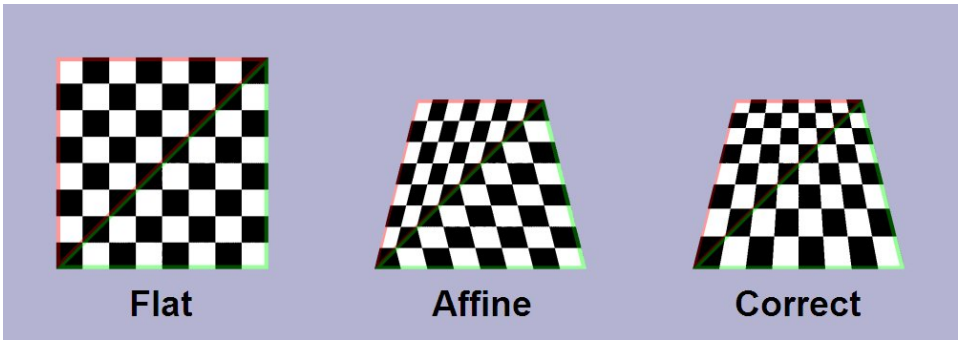

Figure 86

46.0.260. Perspectively Correct Interpolation of Varying Variables

The interpolation described in the previous section can result in certain distortions if applied to scenes that use perspective projection. For the perspectively correct interpolation the distance to the view point is placed in the fourth component of the three vertex positions (w_1, w_2, and w_3) and the following equation is used for the interpolation:

$$f_P \quad = \quad \frac{\alpha_1 f_1/w_1 + \alpha_2 f_2/w_2 + \alpha_3 f_3/w_3}{\alpha_1/w_1 + \alpha_2/w_2 + \alpha_3/w_3}$$

Thus, the fourth component of the position of the vertices is important for perspectively correct interpolation of varying variables. Therefore, it is also important that the perspective division (which sets this fourth component to 1) is not performed in the vertex shader, otherwise the interpolation will be incorrect in the case of perspective projections. (Moreover, clipping fails in some cases.)

It should be noted that actual OpenGL implementations are unlikely to implement exactly the same procedure because there are more efficient techniques. However, all perspectively correct linear interpolation methods result in the same interpolated values.

46.0.261. Further Reading

All details about the rasterization of OpenGL ES are defined in full detail in Chapter 3 of the "OpenGL ES 2.0.x Specification" available at the "KHRONOS OPENGL ES API REGISTRY"[3].

3 HTTP://WWW.KHRONOS.ORG/REGISTRY/GLES/

47. Per-Fragment Operations

The per-fragment operations are part of the OPENGL (ES) 2.0 PIPELINE[1] and consist of a series of tests and operations that can modify the fragment color generated by the fragment shader before it is written to the corresponding pixel in the framebuffer. If any of the tests fails, the fragment will be discarded. (An exception to this rule is the stencil test, which can be configured to change the stencil value of the pixel even if the fragment failed the stencil and/or depth test.)

Overview

In the overview of the per-fragment operations, blue boxes represent data and gray boxes represent configurable per-fragment operations.

Fragment Data (Color and Depth)	output of the fragment shader
↓	
Pixel Ownership Test	not customizable by applications
↓	
Scissor Test	fails if fragment is outside of a specified (scissor) rectangle
↓	

1 HTTP://EN.WIKIBOOKS.ORG/WIKI/%2FOPENGL%20ES%202.0%20PIPELINE

Alpha Test

only in OpenGL, not in OpenGL ES 2.x; outcome depends on alpha value (opacity) of the fragment, a reference alpha value and the configured alpha test

↓

Stencil Test

outcome depends on value in stencil buffer at corresponding pixel and the configured stencil test

↓

Depth Test

outcome depends on depth of fragment, depth in depth buffer at the corresponding pixel and the configured depth test

↓

Blending	outcome depends on fragment color (including its opacity), color in framebuffer at corresponding pixel and configured blend equation
↓	
Dithering	usually not customizable by applications
↓	
Logical Operations	only in OpenGL, not in OpenGL ES 2.x; outcome depends on color of fragment, color in framebuffer at corresponding pixel and configured logical (bitwise) operation
↓	
Framebuffer (Color, Depth and Stencil Value)	writing to the framebuffer can be disabled for individual channels

Specific Per-Fragment Operations

Some of the per-fragment operations are particularly important because they have established applications:

- The **depth test** is used to render opaque primitives (e.g. triangles) with correct occlusions. This is done by comparing the depth of a fragment to the depth of the frontmost previously rendered fragment, which is stored in the depth buffer. If the fragment is farther away, then the depth test fails and the fragment is discarded. Otherwise the fragment is the new frontmost fragment and its depth is stored in the depth buffer.
- **Blending** is used to render semitransparent primitives (glass, fire, flares, etc.) by blending (i.e. compositing) the color of the fragment with the color that is already stored in the framebuffer. This is usually combined with disabling writing to the depth buffer in order to avoid that semitransparent primitives occlude other primitives.
- The **stencil test** is often used to restrict rendering to certain regions of the screen, e.g. a mirror, a window or a more general "portal" in the 3D scene. It also has more advanced uses for some algorithms, e.g. shadow volumes.

More details about specific per-fragment operations can be found in the platform-specific tutorials because it depends on the platform, how the operations are configured.

Note on the Order of Per-Fragment Operations

While the specification of OpenGL imposes a certain order of the per-fragment operations, GPUs can change the order of these operations as long as this doesn't change the result. In fact, GPUs will perform many of the tests even before the fragment shader is executed whenever this is possible. In this case the fragment shader is not executed for a fragment that has failed one of the test. (One example is the so-called "early depth test.") This can result in considerable performance gains.

Further Reading

The per-fragment operations of OpenGL ES 2.0 are defined in full detail in Chapter 4 of the "OpenGL ES 2.0.x Specification" available at the "KHRONOS OPENGL ES API REGISTRY"[2].

2 HTTP://WWW.KHRONOS.ORG/REGISTRY/GLES/

A more accessible description is given in Chapter 11 of the book “OpenGL ES 2.0 Programming Guide” by Aaftab Munshi, Dan Ginsburg and Dave Shreiner published by Addison-Wesley (see ITS WEB SITE[3]).

3 HTTP://WWW.OPENGLES-BOOK.COM/

48. Contributors

Edits	User
1	ADRIGNOLA[1]
1	BEUC[2]
1	COMMONSDELINKER[3]
4	DIRK HÜNNIGER[4]
1	HETHRIRBOT[5]
2	JESSYUV[6]
665	MARTIN KRAUS[7]
1	PANIC2K4[8]
1	QUBOT[9]

1 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:ADRIGNOLA
2 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:BEUC
3 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:COMMONSDELINKER
4 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:DIRK_H%C3%BCNNIGER
5 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:HETHRIRBOT
6 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:JESSYUV
7 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:MARTIN_KRAUS
8 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:PANIC2K4
9 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:QUBOT

List of Figures

- GFDL: Gnu Free Documentation License. HTTP://WWW.GNU.ORG/LICENSES/FDL.HTML
- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. HTTP://CREATIVECOMMONS.ORG/LICENSES/BY-SA/3.0/
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. HTTP://CREATIVECOMMONS.ORG/LICENSES/BY-SA/2.5/
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. HTTP://CREATIVECOMMONS.ORG/LICENSES/BY-SA/2.0/
- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. HTTP://CREATIVECOMMONS.ORG/LICENSES/BY-SA/1.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. HTTP://CREATIVECOMMONS.ORG/LICENSES/BY/2.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. HTTP://CREATIVECOMMONS.ORG/LICENSES/BY/2.0/DEED.EN
- cc-by-2.5: Creative Commons Attribution 2.5 License. HTTP://CREATIVECOMMONS.ORG/LICENSES/BY/2.5/DEED.EN
- cc-by-3.0: Creative Commons Attribution 3.0 License. HTTP://CREATIVECOMMONS.ORG/LICENSES/BY/3.0/DEED.EN
- GPL: GNU General Public License. HTTP://WWW.GNU.ORG/LICENSES/GPL-2.0.TXT
- LGPL: GNU Lesser General Public License. HTTP://WWW.GNU.ORG/LICENSES/LGPL.HTML
- PD: This image is in the public domain.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.

- EURO: This is the common (reverse) face of a euro coin. The copyright on the design of the common face of the euro coins belongs to the European Commission. Authorised is reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.
- LFK: Lizenz Freie Kunst. HTTP://ARTLIBRE.ORG/LICENCE/LAL/DE
- CFR: Copyright free use.
- EPL: Eclipse Public License. HTTP://WWW.ECLIPSE.ORG/ORG/DOCUMENTS/EPL-V10.PHP

Copies of the GPL, the LGPL as well as a GFDL are included in chapter LICENSES[10]. Please note that images in the public domain do not require attribution. You may click on the image numbers in the following table to open the webpage of the images in your webbrower.

10 Chapter 49 on page 459

1	MARTIN KRAUS[11]	cc-by-sa-3.0
2	SHARKD[12]	GFDL
3	NASA[13]	PD
4	Hans Bernhard (SCHNOBBY[14])	GFDL
5		PD
6		PD
7	Flickr user Ombligotron	cc-by-2.0
8	Ben Newton	cc-by-2.0
9	ROBERTO ARIAS[15]	cc-by-2.0
10	Original uploader was OLEG ALEXANDROV[16] at EN.WIKIPEDIA[17]	PD
11	**阿爾特斯**[18]	GFDL
12	MARTIN KRAUS[19]	PD
13		PD
14	MARTIN KRAUS[20]	PD
15	XARIO[21]	GFDL
16	MARTIN KRAUS[22]	PD
17	Maarten Everts	PD
18	Maarten Everts	PD
19	SALIX ALBA[23]	GFDL
20	GOBAGOO[24]	cc-by-sa-3.0
21	NASA. Photo taken by either HARRISON SCHMITT[25] or RON EVANS[26] (of the APOLLO 17[27] crew).	PD
22	MAXDZ8[28]	PD
23	based on map by jimht at shaw dot ca	PD

11 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AMARTIN%20KRAUS
12 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3ASHARKD
13 HTTP://EN.WIKIBOOKS.ORG/WIKI/NASA
14 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3ASCHNOBBY
15 HTTP://WWW.FLICKR.COM/PHOTOS/ROBERTO8080/
16 HTTP://EN.WIKIBOOKS.ORG/WIKI/%3AEN%3AUSER%3AOLEG%20ALEXANDROV
17 HTTP://EN.WIKIPEDIA.ORG
18 HTTP://EN.WIKIBOOKS.ORG/WIKI/%3AZH%3AUSER%3A%96%3F%72%3E%72%79%65%AF
19 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AMARTIN%20KRAUS
20 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AMARTIN%20KRAUS
21 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AXARIO
22 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AMARTIN%20KRAUS
23 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3ASALIX%20ALBA
24 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AGOBAGOO
25 HTTP://EN.WIKIBOOKS.ORG/WIKI/%3AEN%3AHARRISON%20SCHMITT
26 HTTP://EN.WIKIBOOKS.ORG/WIKI/%3AEN%3ARONALD%20EVANS
27 HTTP://EN.WIKIBOOKS.ORG/WIKI/%3AEN%3AAPOLLO%2017
28 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AMAXDZ8

24	APOLLO 8[29] crewmember BILL ANDERS[30]	PD
25	NASA	PD
26	• LAND_SHALLOW_TOPO_2048.JPG[31]: NASA • derivative work: MARTIN[32] (TALK[33])	PD
27	• LAND_SHALLOW_TOPO_2048.JPG[34]: NASA • derivative work: MARTIN[35] (TALK[36])	PD
28		PD
29	US-Gov	PD
30	Original uploader was DE.MOLAI[37] at IT.WIKIPEDIA[38]. Later version(s) were uploaded by BELFADOR[39] at IT.WIKIPEDIA[40].	PD
31	NASA	PD
32	MICHELANGELO MERISI DA CARAVAGGIO[41]	PD
33	DwX	PD
34	Original uploader was ALEXWRIGHT[42] at EN.WIKIPEDIA[43] Later version(s) were uploaded by BENFRANTZDALE[44] at EN.WIKIPEDIA[45].	PD
35	ALETHE[46]	GFDL
36	MARTIN KRAUS[47]	PD

29 HTTP://EN.WIKIBOOKS.ORG/WIKI/APOLLO%208
30 HTTP://EN.WIKIPEDIA.ORG/WIKI/%20WILLIAM%20ANDERS
31 HTTP://EN.WIKIBOOKS.ORG/WIKI/%3AFILE%3ALAND_SHALLOW_TOPO_2048.JPG
32 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AMARTIN%20KRAUS
33 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%20TALK%3AMARTIN%20KRAUS
34 HTTP://EN.WIKIBOOKS.ORG/WIKI/%3AFILE%3ALAND_SHALLOW_TOPO_2048.JPG
35 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AMARTIN%20KRAUS
36 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%20TALK%3AMARTIN%20KRAUS
37 HTTP://EN.WIKIBOOKS.ORG/WIKI/%3AIT%3AUSER%3ADE.MOLAI
38 HTTP://IT.WIKIPEDIA.ORG
39 HTTP://EN.WIKIBOOKS.ORG/WIKI/%3AIT%3AUSER%3ABELFADOR
40 HTTP://IT.WIKIPEDIA.ORG
41 HTTP://EN.WIKIBOOKS.ORG/WIKI/MICHELANGELO%20MERISI%20DA%20CARAVAGGIO
42 HTTP://EN.WIKIBOOKS.ORG/WIKI/%3AEN%3AUSER%3AALEXWRIGHT
43 HTTP://EN.WIKIPEDIA.ORG
44 HTTP://EN.WIKIBOOKS.ORG/WIKI/%3AEN%3AUSER%3ABENFRANTZDALE
45 HTTP://EN.WIKIPEDIA.ORG
46 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AALETHE
47 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AMARTIN%20KRAUS

37		PD
38	Sean Devine at http://picasaweb.google.com/seanmdevine	cc-by-sa-3.0
39	= {{int:license	PD
40	MAILER_DIABLO[48]	GFDL
41	LADYOFHATS[49]	PD
42	Elaine Sosa Labalme	cc-by-sa-3.0
43	User TOPHERTG[50] on EN.WIKIPEDIA[51]	GFDL
44	E.Zimbres and Tom Epaminondas Mineral Collectors	cc-by-sa-2.5
45	D'ARCY NORMAN[52] from Calgary, Canada	cc-by-2.0
46	Ricardo André Frantz (USER:TETRAKTYS[53])	GFDL
47	Rick Kimpel	cc-by-sa-2.5
48	MARTIN KRAUS[54]	PD
49	Filipe Dâmaso Saraiva	cc-by-sa-2.0
50	MARTIN KRAUS[55]	PD
51	NEITRAM[56]	GFDL
52	MARTIN KRAUS[57]	PD
53	aphid dew	cc-by-2.0
54	MARTIN KRAUS[58]	PD
55	Wikipedia Loves Art participant " VA_VA_VAL[59]"	cc-by-sa-2.5
56	KENSPLANET[60]	PD
57	Jeff Meyer	cc-by-2.0
58	Luis Argerich	cc-by-2.0
59	Lars Hellvig	PD
60	ALBERTO FERNANDEZ FERNANDEZ[61]	PD
61	MARTIN KRAUS[62]	PD
62	LADYOFHATS[63]	PD

48 HTTP://EN.WIKIBOOKS.ORG/WIKI/%3AEN%3AUSER%3AMAILER%20DIABLO
49 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3ALADYOFHATS
50 HTTP://EN.WIKIBOOKS.ORG/WIKI/%3AEN%3AUSER%3ATOPHERTG
51 HTTP://EN.WIKIPEDIA.ORG
52 HTTP://WWW.FLICKR.COM/PEOPLE/51035644987@N01
53 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3ATETRAKTYS
54 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AMARTIN%20KRAUS
55 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AMARTIN%20KRAUS
56 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3ANEITRAM
57 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AMARTIN%20KRAUS
58 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AMARTIN%20KRAUS
59 HTTP://WWW.FLICKR.COM/GROUPS/WIKIPEDIA_LOVES_ART/POOL/TAGS/VA_VA_VAL/
60 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AKENSPLANET
61 /WIKI/USER:AFERNAND74
62 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AMARTIN%20KRAUS
63 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3ALADYOFHATS

63	LadyofHats[64]	PD
64	LadyofHats[65]	PD
65	LadyofHats[66]	PD
66	LadyofHats[67]	PD
67		PD
68	w:User:Ross Uber[68]	cc-by-sa-2.5
69	Georges Hébert	PD
70	LadyofHats[69]	PD
71	Martin Kraus[70]	PD
72		PD
73	Peter Trimming[71]	cc-by-sa-2.0
74		PD
75	USAAF	PD
76	• Camera_icon.svg[72]: Everaldo Coelho and YellowIcon • derivative work: Martin Kraus	LGPL
77	Martin Kraus[73]	cc-by-sa-3.0
78		PD
79	Martin Kraus[74]	cc-by-sa-3.0
80	Martin Kraus[75]	cc-by-sa-3.0
81	Martin Kraus[76]	cc-by-sa-3.0
82	Martin Kraus[77]	cc-by-sa-3.0
83	Martin Kraus[78]	cc-by-sa-3.0
84	Martin Kraus[79]	cc-by-sa-3.0
85	Martin Kraus[80]	cc-by-sa-3.0

64 http://en.wikibooks.org/wiki/User%3ALadyofHats
65 http://en.wikibooks.org/wiki/User%3ALadyofHats
66 http://en.wikibooks.org/wiki/User%3ALadyofHats
67 http://en.wikibooks.org/wiki/User%3ALadyofHats
68 http://en.wikipedia.org/wiki/User%3ARoss%20Uber
69 http://en.wikibooks.org/wiki/user%3ALadyofHats
70 http://en.wikibooks.org/wiki/User%3AMartin%20Kraus
71 http://www.geograph.org.uk/profile/34298
72 http://en.wikibooks.org/wiki/%3AFile%3ACamera_icon.svg
73 http://en.wikibooks.org/wiki/User%3AMartin%20Kraus
74 http://en.wikibooks.org/wiki/User%3AMartin%20Kraus
75 http://en.wikibooks.org/wiki/User%3AMartin%20Kraus
76 http://en.wikibooks.org/wiki/User%3AMartin%20Kraus
77 http://en.wikibooks.org/wiki/User%3AMartin%20Kraus
78 http://en.wikibooks.org/wiki/User%3AMartin%20Kraus
79 http://en.wikibooks.org/wiki/User%3AMartin%20Kraus
80 http://en.wikibooks.org/wiki/User%3AMartin%20Kraus

86	RAINWARRIOR[81]	PD

81 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3ARAINWARRIOR

49. Licenses

49.1. GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow. TERMS AND CONDITIONS 0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion. 1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work. 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary. 3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures. 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee. 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

* a) The work must carry prominent notices stating that you modified it, and giving a relevant date. * b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices". * c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it. * d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate. 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

* a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange. * b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge. * c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b. * d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements. * e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying. 7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

* a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or * b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or * c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or * d) Limiting the use for publicity purposes of names of licensors or authors of the material; or * e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or * f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way. 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate

your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10. 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so. 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party’s predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it. 11. Patents.

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor’s “contributor version”.

A contributor’s “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor’s essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a “patent license” is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To “grant” such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient’s use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law. 12. No Surrender of Others’ Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program. 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such. 14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version. 15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

<one line to give the program’s name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

<program> Copyright (C) <year> <name of author> This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’. This is free software, and you are welcome to redistribute it under certain conditions; type ‘show c’ for details.

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, your program’s commands might be different; for a GUI interface, you would use an “about box”.

You should also get your employer (if you work as a programmer) or school, if any, to sign a “copyright disclaimer” for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

49.2. GNU Free Documentation License

Version 1.3, 3 November 2008

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies. 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

* A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. * B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement. * C. State on the Title page the name of the publisher of the Modified Version, as the publisher. * D. Preserve all the copyright notices of the Document. * E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. * F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. * G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. * H. Include an unaltered copy of this License. * I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. * J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. * K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. * L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. * M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version. * N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section. * O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements". 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document. 11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with … Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

49.3. GNU Lesser General Public License

GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below. 0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

"The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work. 1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL. 2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

* a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or * b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

* a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

* a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the Combined Work with a copy of the GNU GPL and this license document. * c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document. * d) Do one of the following: o 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source. o 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version. * e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

* a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License. * b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.