

Cg Programming in Unity

An Introduction to Real-Time 3D Graphics

Contents

Articles

Introduction	1
1 Basics	3
1.1 Minimal Shader	3
1.2 RGB Cube	7
1.3 Debugging of Shaders	11
1.4 Shading in World Space	17
2 Transparent Surfaces	24
2.1 Cutaways	24
2.2 Transparency	29
2.3 Order-Independent Transparency	35
2.4 Silhouette Enhancement	40
3 Basic Lighting	45
3.1 Diffuse Reflection	45
3.2 Specular Highlights	58
3.3 Two-Sided Surfaces	65
3.4 Smooth Specular Highlights	76
3.5 Two-Sided Smooth Surfaces	82
3.6 Multiple Lights	93
4 Basic Texturing	101
4.1 Textured Spheres	101
4.2 Lighting Textured Surfaces	107
4.3 Glossy Textures	113
4.4 Transparent Textures	120
4.5 Layers of Textures	129
5 Textures in 3D	134
5.1 Lighting of Bumpy Surfaces	134
5.2 Projection of Bumpy Surfaces	144
5.3 Cookies	156
5.4 Light Attenuation	171
5.5 Projectors	179

6 Environment Mapping	185
6.1 Reflecting Surfaces	185
6.2 Curved Glass	189
6.3 Skyboxes	191
6.4 Many Light Sources	195
7 Variations on Lighting	212
7.1 Brushed Metal	212
7.2 Specular Highlights at Silhouettes	222
7.3 Diffuse Reflection of Skylight	230
7.4 Translucent Surfaces	234
7.5 Translucent Bodies	243
7.6 Soft Shadows of Spheres	257
7.7 Toon Shading	268
8 Non-Standard Vertex Transformations	278
8.1 Screen Overlays	278
8.2 Billboards	283
8.3 Nonlinear Deformations	286
8.4 Shadows on Planes	290
8.5 Mirrors	296
9 Graphics without Shaders	305
9.1 Rotations	305
9.2 Projection for Virtual Reality	312
9.3 Bézier Curves	318
9.4 Hermite Curves	324
Appendix on the Programmable Graphics Pipeline and Cg Syntax	331
A.1 Programmable Graphics Pipeline	331
A.2 Vertex Transformations	335
A.3 Vector and Matrix Operations	343
A.4 Applying Matrix Transformations	348
A.5 Rasterization	353
A.6 Per-Fragment Operations	356
References	
Article Sources and Contributors	358

Article Licenses

Introduction

About Cg

Nvidia's programming language Cg (C for graphics) is one of several commonly used shading languages for real-time rendering (other examples are Direct3D's HLSL and OpenGL's GLSL). 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. Cg is one of them. In fact, it was one of the first high-level shading languages for GPUs and is implemented for several 3D graphics APIs (application programming interfaces), most importantly OpenGL and Direct3D. Today, the main reason for its popularity is its similarity to HLSL, which is the shading language of Microsoft's Direct3D. In practice, there is usually no difference between Cg and HLSL shaders.

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 Cg within the game engine Unity.

Chapters 1 to 9 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 Cg or Unity you should at least read through the tutorials in Chapter 1, "Basics".

More details about the programmable graphics pipeline and Cg syntax in general are included in an "Appendix on the Programmable Graphics Pipeline and Cg Syntax". Readers who are not familiar with GPUs or Cg might want to at least skim this part since a basic understanding of this pipeline and Cg syntax is very useful for understanding the tutorials.

About Cg in Unity

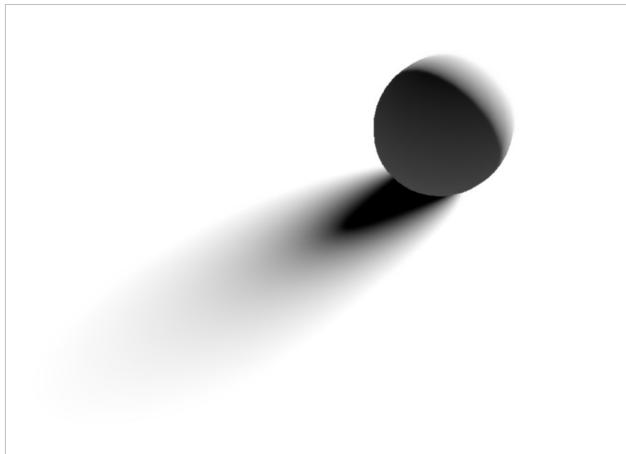
Cg programming in the game engine Unity is considerably easier than Cg programming for an OpenGL or Direct3D application. Import of meshes and images (i.e. textures) is supported by a graphical user interface; mipmaps and normal maps can be computed automatically; the most common vertex attributes and uniforms are predefined; OpenGL and Direct3D 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, the tutorials assume 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 [2] (Unity Basics, Building Scenes, Asset Import and Creation).
- Second, Unity doesn't distinguish between Cg (the shading language by Nvidia) and HLSL (the shading language in Direct3D) since the two languages are very similar; thus, most of these tutorials also apply to HLSL.
- Furthermore, Cg is documented by Nvidia's Cg Tutorial ^[3] and Nvidia's Cg Language Specification ^[4]. However, these descriptions are missing the details specific to Unity. On the other hand, Unity's shader documentation [5]

focuses on shaders written in Unity's own "surface shader" format, while the documentation of shaders in Cg/HLSL is very limited [6]. Thus, learning Cg programming in Unity without prior knowledge of Cg can be rather difficult. This wikibook tries to close this gap by providing an introduction to Cg programming in Unity without requiring prior knowledge of Cg.

Martin Kraus, March 2013



References

- [1] <http://unity3d.com/unity/download/>
- [2] <http://unity3d.com/support/documentation/Manual/User%20Guide.html>
- [3] http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html
- [4] http://http.developer.nvidia.com/Cg/Cg_language.html
- [5] <http://unity3d.com/support/documentation/Components/SL-Reference.html>
- [6] <http://unity3d.com/support/documentation/Components/SL-ShaderPrograms.html>

1 Basics

1.1 Minimal Shader

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

Starting Unity and Creating a New Project

After downloading and starting Unity, 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.

If you are not familiar with Unity's Scene View, Hierarchy 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 ^[2].

Creating a Shader

Creating a Cg 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 "Cg basic shader" { // defines the name of the shader
    SubShader { // Unity chooses the subshader that fits the GPU best
        Pass { // some shaders require multiple passes
            CGPROGRAM // here begins the part in Unity's Cg

                #pragma vertex vert
                // this specifies the vert function as the vertex shader
                #pragma fragment frag
                // this specifies the frag function as the fragment shader

                float4 vert(float4 vertexPos : POSITION) : SV_POSITION
                    // vertex shader
                {
                    return mul(UNITY_MATRIX_MVP, vertexPos);
                    // this line transforms the vertex input parameter
                    // vertexPos with the built-in matrix UNITY_MATRIX_MVP
                    // and returns it as a nameless vertex output parameter
                }

                float4 frag(void) : COLOR // fragment shader
                {
                    return float4(1.0, 0.0, 0.0, 1.0);
                    // this fragment shader returns a nameless fragment
                    // output parameter (with semantic COLOR) that is set to
                    // opaque red (red = 1, green = 0, blue = 0, alpha = 1)
            }
        }
    }
}
```

```

    }

    ENDCG // here ends the part in Cg
}

}

}

```

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.

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 “Cg 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.

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 in the `frag` function with this code:

```

float4 frag(void) : COLOR // fragment shader
{
    return float4(0.6, 1.0, 0.0, 1.0);
    // (red = 0.6, green = 1.0, blue = 0.0, alpha = 1.0)
}

```

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 about finding the warmest orange and another about dark blue that is almost black.)

You could also play with the vertex shader in the function `vert`, e.g. try this vertex shader:

```

float4 vert(float4 vertexPos : POSITION) : SV_POSITION
    // vertex shader
{
    return mul(UNITY_MATRIX_MVP,
        float4(1.0, 0.1, 1.0, 1.0) * vertexPos);
}

```

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 Cg see the discussion in Section “Vector and Matrix Operations”.)

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.

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

Saving Your Work in a Scene

There is one more thing: you should save your 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.

One More Note about Terminology

It might be good to clarify the terminology. In some APIs, a “shader” is either a vertex shader or a fragment shader. The combination of both is called a “program”. In other APIs, it's just the other way around: a “program” is either a vertex program or a fragment program, and the combination of both is a “shader”. Unfortunately, Unity's documentation mixes both conventions. To keep things simple, we try to avoid the term “program” here and use the term “shader” to denote the combination of a vertex shader and a fragment shader.

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 Cg vertex and fragment shader in Unity.
- How to create a material and attach a shader to the material.
- How to manipulate the fragment output parameter with the semantic `COLOR` in the fragment shader.
- How to transform the vertex input parameter with the semantic `POSITION` in the vertex shader.
- How to create a game object and attach a material to it.

Actually, this was quite a lot of stuff.

Further Reading

If you still want to know more

- about vertex and fragment shaders in general, you should read the description in Section “Programmable Graphics Pipeline”.
- about the vertex transformations such as `UNITY_MATRIX_MVP`, which contains a product of the model-view matrix and the projection matrix, you should read Section “Vertex Transformations”.
- about handling vectors (e.g. the `float4` type) and matrices in Cg, you should read Section “Vector and Matrix Operations”.
- about how to apply vertex transformations such as `UNITY_MATRIX_MVP`, you should read Section “Applying Matrix Transformations”.
- about Unity's ShaderLab language for specifying shaders, you should read Unity's ShaderLab reference [5].

page traffic for 90 days ^[1]

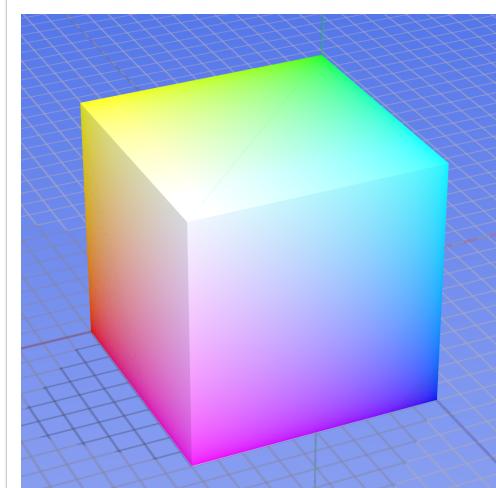
< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Minimal_Shader

1.2 RGB Cube



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

This tutorial discusses vertex output parameters and fragment input parameters. It is based on Section “Minimal Shader”.

In this tutorial we will write a shader to render an RGB cube similar to the one shown to the left. 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 to the left.)

Preparations

Since we want to create an RGB cube, you first have to create a cube game object. As described in Section “Minimal Shader” 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 Section “Minimal Shader”.

The Shader Code

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

```
Shader "Cg shader for RGB cube" {
    SubShader {
        Pass {
            CGPROGRAM

                #pragma vertex vert // vert function is the vertex shader
                #pragma fragment frag // frag function is the fragment shader

                // for multiple vertex output parameters an output structure
                // is defined:
                struct vertexOutput {
                    float4 pos : SV_POSITION;
                    float4 col : TEXCOORD0;
                };

                vertexOutput vert(float4 vertexPos : POSITION)
                // vertex shader
                {
                    vertexOutput output; // we don't need to type 'struct' here

                    output.pos = mul(UNITY_MATRIX_MVP, vertexPos);
                    output.col = vertexPos + float4(0.5, 0.5, 0.5, 0.0);
                    // Here the vertex shader writes output data
                }
            ENDCG
        }
    }
}
```

```

        // to the output structure. 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.
    return output;
}

float4 frag(vertexOutput input) : COLOR // fragment shader
{
    return input.col;
    // Here the fragment shader returns the "col" input
    // parameter with semantic TEXCOORD0 as nameless
    // output parameter with semantic COLOR.
}

ENDCG
}
}
}

```

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.

Communication between Vertex and Fragment Shaders

The main task of our shader is to set the fragment output color (i.e. the fragment output parameter with semantic COLOR) in the fragment shader to the vertex position that is available in the vertex shader. Actually, this is not quite true: the coordinates in the vertex input parameter with semantic POSITION 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: `vertexPos + float4(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 pairs of vertex output parameters and fragment input parameters **with the same semantics** (TEXCOORD0 in this case). In fact, it is only the semantics that are used to determine which vertex output parameters correspond to which fragment input parameters. Instead of the semantic TEXCOORD0 we could also use another semantic, e.g. COLOR, it doesn't really matter here, except that parameters with the semantic COLOR are often clamped to values between 0 and 1 (which would be OK in this case). It is, however, common to use the semantics TEXCOORD0, TEXCOORD1, TEXCOORD2, etc. for all kinds of parameters.

The next problem is to specify multiple vertex output parameters. Since the return instruction can only return one value, it is common to define a structure for all the required vertex output parameters. Here, this structure is called `vertexOutput`:

```

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 col : TEXCOORD0;
};

```

By using this structure as an argument of the fragment shader function, we make sure that the semantics match. Note that in Cg (in contrast to C), we don't have to write `struct vertexOutput` when defining variables of this type

but we can just use the name `vertexOutput` (without struct) for the same type.

The `out` Qualifier

An alternative to the use of an output structure would be to use arguments of the vertex shader function with the `out` qualifier, e.g.:

```
Shader "Cg shader for RGB cube" {
    SubShader {
        Pass {
            CGPROGRAM

                #pragma vertex vert // vert function is the vertex shader
                #pragma fragment frag // frag function is the fragment shader

                void vert(float4 vertexPos : POSITION,
                        out float4 pos : SV_POSITION,
                        out float4 col : TEXCOORD0)
                {
                    pos = mul(UNITY_MATRIX_MVP, vertexPos);
                    col = vertexPos + float4(0.5, 0.5, 0.5, 0.0);
                    return;
                }

                float4 frag(float4 pos : SV_POSITION,
                            float4 col : TEXCOORD0) : COLOR
                {
                    return col;
                }
            ENDCG
        }
    }
}
```

However, the use of an output structure is more common in practice and it makes sure that vertex output parameters and fragment input parameters have matching semantics.

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 to 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 $180.0 + \text{degrees}(\text{atan2}(z, x))$ in Cg 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.

Interpolation of Vertex Output Parameters

The story about vertex output parameters and fragment input parameters 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 different outputs are written to the vertex output parameters. However, there are many more colors on the cube. How did that happen?

In fact, the vertex shader is only called for each vertex of each triangle. However, the different values of the vertex output parameters for the different vertices 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 vertex output parameters as fragment input parameters. The details of this interpolation are described in Section "Rasterization".

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 vertex output parameters for all vertices of a triangle.

Summary

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

- What an RGB cube is.
- What an output structure is and how to define it.
- How output structures are used to make sure that vertex output parameters have the same semantics as fragment input parameters.
- How the values written to vertex output parameters are interpolated across a triangle before they are received as input parameters by the fragment shader.

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 Section "Programmable Graphics Pipeline".
- about vector and matrix operations (e.g. the expression `vertexPos + float4(0.5, 0.5, 0.5, 0.0)`), you should read Section "Vector and Matrix Operations".
- about the interpolation of vertex output parameters, you should read Section "Rasterization".
- about Unity's official documentation of writing vertex shaders and fragment shaders in Unity's ShaderLab, you should read Unity's ShaderLab reference about "Writing vertex and fragment shaders"^[1].

page traffic for 90 days^[2]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

- [1] <http://docs.unity3d.com/Documentation/Components/SL-ShaderPrograms.html>
- [2] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/RGB_Cube

1.3 Debugging of Shaders



A false-color satellite image.

This tutorial discusses vertex input parameters. It is based on Section “Minimal Shader” and Section “RGB Cube”.

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.

Where Does the Vertex Data Come from?

In Section “RGB Cube” you have seen how the fragment shader gets its data from the vertex shader by means of an output structure of vertex output parameters. 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 vertex input parameters. The mapping of different attributes to different vertex input parameters is usually achieved in Cg by means of semantics, i.e. each vertex input parameter has to specify a certain semantic, e.g. POSITION, NORMAL, TEXCOORD0, TEXCOORD1, TANGENT, COLOR, etc. However, in Unity’s particular implementation of Cg, the built-in vertex input parameters have to have specific names as discussed next.

Built-in Vertex Input Parameters and how to Visualize Them

In Unity, the built-in vertex input parameters (position, surface normal, two sets of texture coordinates, tangent vector, and vertex color) not only have to have certain semantics but also certain names and types. Furthermore, they should be included in a single structure of input vertex parameters, e.g.:

```
struct vertexInput {
    float4 vertex : POSITION; // position (in object coordinates,
    // i.e. local or model coordinates)
    float4 tangent : TANGENT;
    // vector orthogonal to the surface normal
    float3 normal : NORMAL; // surface normal vector (in object
    // coordinates; usually normalized to unit length)
```

```
float4 texcoord : TEXCOORD0; // 0th set of texture
    // coordinates (a.k.a. "UV"; between 0 and 1)
float4 texcoord1 : TEXCOORD1; // 1st set of texture
    // coordinates (a.k.a. "UV"; between 0 and 1)
fixed4 color : COLOR; // color (usually constant)
};
```

This structure could be used this way:

```
Shader "Cg shader with all built-in vertex input parameters" {
    SubShader {
        Pass {
            CGPROGRAM

#pragma vertex vert
#pragma fragment frag

            struct vertexInput {
                float4 vertex : POSITION;
                float4 tangent : TANGENT;
                float3 normal : NORMAL;
                float4 texcoord : TEXCOORD0;
                float4 texcoord1 : TEXCOORD1;
                fixed4 color : COLOR;
            };
            struct vertexOutput {
                float4 pos : SV_POSITION;
                float4 col : TEXCOORD0;
            };

            vertexOutput vert(vertexInput input)
            {
                vertexOutput output;

                output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
                output.col = input.texcoord; // set the output color

                // other possibilities to play with:

                // output.col = input.vertex;
                // output.col = input.tangent;
                // output.col = float4(input.normal, 1.0);
                // output.col = input.texcoord;
                // output.col = input.texcoord1;
                // output.col = input.color;

                return output;
            }
        }
    }
}
```

```

        float4 frag(vertexOutput input) : COLOR
    {
        return input.col;
    }

    ENDCG
}
}

}

```

In Section “RGB Cube” we have already seen, how to visualize the vertex coordinates by setting the fragment color to those values. In this example, the fragment color is set to the texture coordinates 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 Section “Projection of Bumpy Surfaces”).

Pre-Defined Input Structures

Usually, you can achieve a higher performance by only specifying the vertex input parameters that you actually need, e.g. position, normal, and one set of texture coordinates; sometimes also the tangent vector. Unity provides the pre-defined input structures `appdata_base`, `appdata_tan`, and `appdata_full` for the most common cases. These are defined in the file `UnityCG.cginc` (in the directory `Unity > Editor > Data > CGIncludes`):

```

struct appdata_base {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
};

struct appdata_tan {
    float4 vertex : POSITION;
    float4 tangent : TANGENT;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
};

struct appdata_full {
    float4 vertex : POSITION;
    float4 tangent : TANGENT;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
    float4 texcoord1 : TEXCOORD1;
    fixed4 color : COLOR;
    // and additional texture coordinates only on XBOX360)
};

```

The file `UnityCG.cginc` is included with the line `#include "UnityCG.cginc"`. Thus, the shader above could be rewritten this way:

```

Shader "Cg shader with all built-in vertex input parameters" {
    SubShader {
        Pass {
            CGPROGRAM

                #pragma vertex vert
                #pragma fragment frag
                #include "UnityCG.cginc"

                struct vertexOutput {
                    float4 pos : SV_POSITION;
                    float4 col : TEXCOORD0;
                };

                vertexOutput vert(appdata_full input)
                {
                    vertexOutput output;

                    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
                    output.col = input.texcoord;

                    return output;
                }

                float4 frag(vertexOutput input) : COLOR
                {
                    return input.col;
                }
            ENDCG
        }
    }
}

```

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 input vertex parameter `texcoord` with semantic `TEXCOORD0` for a sphere is written to the fragment color then the red component of the fragment visualizes the `x` coordinate of `texcoord`, 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 component, 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 output parameter in the vertex shader:

```
output.col = float4(input.texcoord.x, 0.0, 0.0, 1.0);
```

This sets the red component of the output parameter to the `x` component of `texcoord` 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 `texcoord` 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:

```
output.col = float4(0.0, input.texcoord.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, the `normal` input parameter 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.:

```
output.col = float4(
    (input.normal + float3(1.0, 1.0, 1.0)) / 2.0, 1.0);
```

Note that `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.

Debugging Practice

In order to practice the debugging of shaders, this section includes some lines that produce black colors when the assignment to `col` 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 Section “Vector and Matrix Operations”.

```
output.col = input.texcoord - float4(1.5, 2.3, 1.1, 0.0);
output.col = float4(input.texcoord.z);
output.col = input.texcoord / tan(0.0);
```

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

```
output.col = dot(input.normal, float3(input.tangent)) *
    input.texcoord;
output.col = dot(cross(input.normal, float3(input.tangent)),
    input.normal) * input.texcoord;
output.col = float4(cross(input.normal, input.normal), 1.0);
```

```
output.col = float4(cross(input.normal,
    float3(input.vertex)), 1.0);
// only for a sphere!
```

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

```
output.col = radians(input.texcoord);
output.col = noise(input.texcoord);
```

Summary

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

- The list of built-in vertex input parameters in Unity.
- How to visualize these parameters (or any other value) by setting components of the fragment output color.

Further Reading

If you still want to know more

- about the data flow in vertex and fragment shaders, you should read the description in Section “Programmable Graphics Pipeline”.
- about operations and functions for vectors, you should read Section “Vector and Matrix Operations”.

page traffic for 90 days ^[1]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Debugging_of_Shaders

1.4 Shading in World Space



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

This tutorial introduces **uniform parameters**. It is based on Section “Minimal Shader”, Section “RGB Cube”, and Section “Debugging of Shaders”.

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?

Transforming from Object to World Space

As mentioned in Section “Debugging of Shaders”, the vertex input parameter with semantic `POSITION` 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 Section “Vertex Transformations”.

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 parameter `_Object2World`, which is defined and used in the following shader:

```
Shader "Cg shading in world space"
{
    SubShader {
        Pass {
            CGPROGRAM
                #pragma vertex vert
                #pragma fragment frag

                uniform float4x4 _Object2World;
                // definition of a Unity-specific uniform parameter

                struct vertexInput {
                    float4 vertex : POSITION;
                };
            ENDCG
        }
    }
}
```

```

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 position_in_world_space : TEXCOORD0;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    output.position_in_world_space =
        mul(_Object2World, input.vertex);
    // transformation of input.vertex from object
    // coordinates to world coordinates;
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float dist = distance(input.position_in_world_space,
        float4(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)
    {
        return float4(0.0, 1.0, 0.0, 1.0);
        // color near origin
    }
    else
    {
        return float4(0.3, 0.3, 0.3, 1.0);
        // color far from origin
    }
}

ENDCG
}
}
}

```

Usually, the application has to set the value of uniform parameters; however, Unity takes care of always setting the correct value of predefined uniform parameters 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 the output structure. For the fragment shader, the parameter in the output structure 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. 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.

More Unity-Specific Uniforms

There are several built-in uniform parameters, which you have to define before you can use them, 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)
// are also defined in "UnityCG.cginc",
// i.e. one could #include "UnityCG.cginc"
uniform float4 _Time, _SinTime, _CosTime; // time values
uniform float4 _ProjectionParams;
    // x = 1 or -1 (-1 if projection is flipped)
    // y = near plane; z = far plane; w = 1/far plane
uniform float4 _ScreenParams;
    // x = width; y = height; z = 1 + 1/width; w = 1 + 1/height
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _LightPositionRange; // xyz = pos, w = 1/range
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0; // color of light source
```

As the comments suggest, instead of defining all these uniforms (except `_LightColor0`), you could also include the file `UnityCG.cginc` which includes the definitions. However, for some unknown reason `_LightColor0` is not included in this file; thus, we have to define it separately (if needed):

```
#include "UnityCG.cginc"
uniform float4 _LightColor0;
```

Unity does not always update all of these uniforms. In particular, `_WorldSpaceLightPos0` and `_LightColor0` 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 Section “Diffuse Reflection”.

For the official list of Unity's built-in uniforms, see the section “ShaderLab builtin values”^[1] in Unity's reference manual.

Pre-Defined Unity-Specific Uniforms

There is another class of built-in uniforms that don't have to be defined in shaders but are always available, for example the `float4x4 matrix UNITY_MATRIX_MVP`, which is equivalent to the matrix product `mul(UNITY_MATRIX_P, UNITY_MATRIX_MV)` of two other built-in uniforms. The corresponding transformations (projection and model-view) are described in detail in Section “Vertex Transformations”.

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

```
uniform float4x4 UNITY_MATRIX_MVP; // model view projection matrix
uniform float4x4 UNITY_MATRIX_MV; // model view matrix
uniform float4x4 UNITY_MATRIX_P; // projection matrix
uniform float4x4 UNITY_MATRIX_T_MV;
    // transpose of model view matrix
uniform float4x4 UNITY_MATRIX_IT_MV;
    // transpose of the inverse model view matrix
uniform float4x4 UNITY_MATRIX_TEXTURE0; // texture matrix
uniform float4x4 UNITY_MATRIX_TEXTURE1; // texture matrix
uniform float4x4 UNITY_MATRIX_TEXTURE2; // texture matrix
uniform float4x4 UNITY_MATRIX_TEXTURE3; // texture matrix
uniform float4 UNITY_LIGHTMODEL_AMBIENT; // ambient color
```

This list also given in section “Built-in state variables in shader programs”^[2] in Unity's reference manual.

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 Section “Vertex Transformations” for the details). Therefore, Unity offers only the product of the model matrix $M_{\text{object} \rightarrow \text{world}}$ and the view matrix $M_{\text{world} \rightarrow \text{view}}$, i.e. the model-view matrix $M_{\text{object} \rightarrow \text{view}}$, which is available in the uniform `UNITY_MATRIX_MV`. The view matrix itself is not available.

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 `unity_Scale.w`.) Thus, we can easily compute the view matrix. The mathematics looks like this:

$$M_{\text{object} \rightarrow \text{view}} = M_{\text{world} \rightarrow \text{view}} M_{\text{object} \rightarrow \text{world}} \Rightarrow M_{\text{world} \rightarrow \text{view}} = M_{\text{object} \rightarrow \text{view}} M_{\text{object} \rightarrow \text{world}}^{-1}$$

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 Cg:

```
float4x4 modelMatrixInverse = _World2Object * unity_Scale.w;
modelMatrixInverse[3][3] = 1.0;
float4x4 viewMatrix = mul(UNITY_MATRIX_MV, modelMatrixInverse);
```

User-Specified Uniforms: Shader Properties

There is one more important type of uniform parameters: uniforms that can be set by the user. Actually, these are called shader properties in Unity. You can think of them as user-specified uniform parameters of the shader. A shader without parameters is usually used only by its programmer because even the smallest necessary change requires some 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 ^[3] 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 "Cg 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 {
            CGPROGRAM

                #pragma vertex vert
                #pragma fragment frag

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

                // uniforms corresponding to properties
                uniform float4 _Point;
                uniform float _DistanceNear;
                uniform float4 _ColorNear;
                uniform float4 _ColorFar;

                struct vertexInput {
                    float4 vertex : POSITION;
                };
                struct vertexOutput {
                    float4 pos : SV_POSITION;
                    float4 position_in_world_space : TEXCOORD0;
                };

                vertexOutput vert(vertexInput input)
                {
                    vertexOutput output;

                    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
                    output.position_in_world_space =
                        mul(_Object2World, input.vertex);
                    return output;
                }
            ENDCG
        }
    }
}
```

```

        float4 frag(vertexOutput input) : COLOR
    {
        float dist = distance(input.position_in_world_space,
        _Point);
        // computes the distance between the fragment position
        // and the position _Point.

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

    ENDCG
}
}
}

```

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.sharedMaterialSetColor("_ColorNear",
    Color(1.0, 0.0, 0.0));
renderer.sharedMaterialSetColor("_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**.

Summary

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

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

Further Reading

If you want to know more

- about vector and matrix operations (e.g. the `distance()` function), you should read Section “Vector and Matrix Operations”.
- about the standard vertex transformations, e.g. the model matrix and the view matrix, you should read Section “Vertex Transformations”.
- about the application of transformation matrices to points and directions, you should read Section “Applying Matrix Transformations”.
- about Unity's built-in uniform parameters, you should read Unity's documentation about “ShaderLab builtin values”^[1].
- about the specification of shader properties, you should read Unity's documentation about “ShaderLab syntax: Properties”^[3].

page traffic for 90 days^[4]

< Cg Programming/Unity

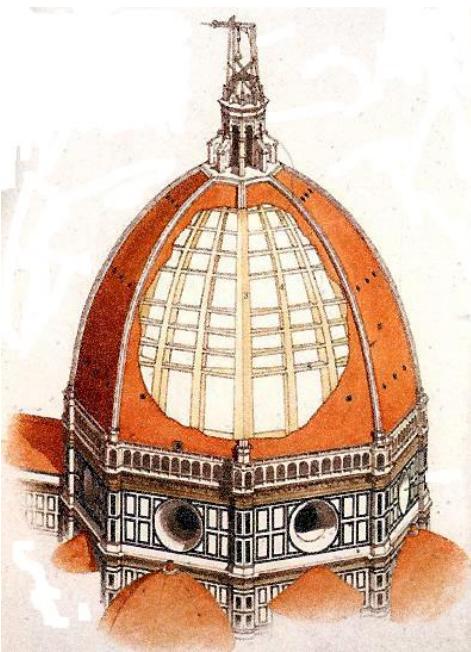
Unless stated otherwise, all example source code on this page is granted to the public domain.

References

- [1] <http://docs.unity3d.com/Documentation/Components/SL-BuiltinValues.html>
- [2] <http://docs.unity3d.com/Documentation/Components/SL-BuiltinStateInPrograms.html>
- [3] <http://unity3d.com/support/documentation/Components/SL-Properties.html>
- [4] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Shading_in_World_Space

2 Transparent Surfaces

2.1 Cutaways



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

This tutorial covers **discarding fragments** and **front-face and back-face culling**. This tutorial assumes that you are familiar with vertex output parameters as discussed in Section “RGB Cube”.

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 to the left, 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. GPUs support these situations in several ways; we will discuss two of them.

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 Section “Vertex Transformations” for details about coordinate systems). Here is the code:

```
Shader "Cg 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

            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            struct vertexInput {
                float4 vertex : POSITION;
            };
            struct vertexOutput {
                float4 pos : SV_POSITION;
                float4 posInObjectCoords : TEXCOORD0;
            };
        }
    }
}
```

```

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    output.posInObjectCoords = input.vertex;

    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    if (input.posInObjectCoords.y > 0.0)
    {
        discard; // drop the fragment if y coordinate > 0
    }
    return float4(0.0, 1.0, 0.0, 1.0); // green
}

ENDCG
}
}
}

```

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.

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 Section "Shading in World Space" for how to transform the vertex into world space.

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 Section “Shading in World Space” for a discussion of shader properties.

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 `float4x4` uniform parameter 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.

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 `CGPROGRAM` because it is not in Cg. In fact, it is a command of Unity's ShaderLab^[1] 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 should 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 Section “Vertex Transformations”) 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.

What can we use culling for? One application is to use a different shader for the front faces than for the back faces, i.e. for the outside and the inside of an object. The following shader uses two passes. In the first pass, only front faces are culled and the remaining faces are rendered red (if the fragments are not discarded). The second pass culs only back faces and renders the remaining faces in green.

```
Shader "Cg shader with two passes using discard" {
    SubShader {
        // first pass (is executed before the second pass)
        Pass {
            Cull Front // cull only front faces
            CGPROGRAM
                #pragma vertex vert
                #pragma fragment frag

                struct vertexInput {
```

```
        float4 vertex : POSITION;
    };

    struct vertexOutput {
        float4 pos : SV_POSITION;
        float4 posInObjectCoords : TEXCOORD0;
    };

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    output.posInObjectCoords = input.vertex;

    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    if (input.posInObjectCoords.y > 0.0)
    {
        discard; // drop the fragment if y coordinate > 0
    }
    return float4(1.0, 0.0, 0.0, 1.0); // red
}

ENDCG
}

// second pass (is executed after the first pass)
Pass {
    Cull Back // cull only back faces

    CGPROGRAM

#pragma vertex vert
#pragma fragment frag

struct vertexInput {
    float4 vertex : POSITION;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posInObjectCoords : TEXCOORD0;
};

vertexOutput vert(vertexInput input)
```

```

    {
        vertexOutput output;

        output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
        output.posInObjectCoords = input.vertex;

        return output;
    }

    float4 frag(vertexOutput input) : COLOR
    {
        if (input.posInObjectCoords.y > 0.0)
        {
            discard; // drop the fragment if y coordinate > 0
        }
        return float4(0.0, 1.0, 0.0, 1.0); // green
    }

    ENDCG
}
}
}

```

Remember that only one subshader of a Unity shader is executed (depending on which subshader fits the capabilities of the GPU best) but all passes of that subshader are executed.

In principle, there are also other ways to distinguish front and back faces in Cg (in particular using fragment input parameters with semantics FACE, VFACE or SV_IsFrontFace depending on the API); however, those don't appear to work well in Unity.

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:

- How to discard fragments.
- How to specify the culling of front and back faces.
- How to use culling and two passes in order to use different shaders for the inside and the outside of a mesh.

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 Section "Vertex Transformations".
- about how to define shader properties, you should read Section "Shading in World Space".
- about Unity's ShaderLab syntax for specifying culling, you should read Culling & Depth Testing ^[1].

page traffic for 90 days ^[2]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

- [1] <http://unity3d.com/support/documentation/Components/SL-CullAndDepth.html>
- [2] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Cutaways

2.2 Transparency

This tutorial covers **blending** of fragments (i.e. compositing them) using Cg shaders in Unity. It assumes that you are familiar with the concept of front and back faces as discussed in Section “Cutaways”.

More specifically, this tutorial is about **rendering transparent objects**, e.g. transparent glass, plastic, fabrics, etc. (More strictly speaking, these are actually semitransparent objects because they don't need to be perfectly transparent.) Transparent objects allow us to see through them; thus, their color “blends” with the color of whatever is behind them.

Blending

As mentioned in Section “Programmable Graphics Pipeline”, the fragment shader computes an RGBA color (i.e. red, green, blue, and alpha components in the fragment output parameter with semantic COLOR) for each fragment (unless the fragment is discarded). The fragments are then processed as discussed in Section “Per-Fragment Operations”. One of the operations is the blending stage, which combines the color of the fragment (as specified in the fragment output parameter), 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 configure it but not program it. The way it is configured, is by specifying a **blend equation**. You can think of the blend equation as this definition of the resulting RGBA color:

```
float4 result = SrcFactor * fragment_output + DstFactor * pixel_color;
```

where `fragment_output` is the RGBA color computed by the fragment shader and `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 configurable RGBA colors (of type `float4`) that are multiplied component-wise with the fragment output 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}
```

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^[1]):



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

Code	Resulting Factor (<code>SrcFactor</code> or <code>DstFactor</code>)
One	<code>float4(1.0)</code>
Zero	<code>float4(0.0)</code>
SrcColor	<code>fragment_output</code>
SrcAlpha	<code>float4(fragment_output.a)</code>
DstColor	<code>pixel_color</code>
DstAlpha	<code>float4(pixel_color.a)</code>
OneMinusSrcColor	<code>float4(1.0) - fragment_output</code>
OneMinusSrcAlpha	<code>float4(1.0 - fragment_output.a)</code>
OneMinusDstColor	<code>float4(1.0) - pixel_color</code>
OneMinusDstAlpha	<code>float4(1.0 - pixel_color.a)</code>

As discussed in Section “Vector and Matrix Operations”, `float4(1.0)` is just a short way of writing `float4(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.

Alpha Blending

One specific example for a blend equation is called “alpha blending”. In Unity, it is specified this way:

```
Blend SrcAlpha OneMinusSrcAlpha
```

which corresponds to:

```
float4 result = float4(fragment_output.a) * fragment_output + float4(1.0 - fragment_output.a) * pixel_color;
```

This uses the alpha component of `fragment_output` as an **opacity**. I.e. the more opaque the fragment output color is, the larger its opacity and therefore its alpha component, and thus the more of the fragment output color is mixed in the result and the less of the pixel color in the framebuffer. A perfectly opaque fragment output 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. “`fragment_output` over `pixel_color`”, since it corresponds to putting a layer of the fragment output color with a specific opacity on top of the pixel color. (Think of a layer of colored glass or colored semitransparent plastic on top of something of another 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**.

Premultiplied Alpha Blending

There is an important variant of alpha blending: sometimes the fragment output 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:

```
float4 result = float4(1.0) * fragment_output + float4(1.0) - fragment_output.a * pixel_color;
```

Additive Blending

Another example for a blending equation is:

Blend One One

This corresponds to:

```
float4 result = float4(1.0) * fragment_output + float4(1.0) * pixel_color;
```

which just adds the fragment output color to the color in the framebuffer. Note that the alpha component is not used at all; nonetheless, this blending equation is very useful for many kinds of transparent effects; for example, it is often used for particle systems when they represent fire or something else that is transparent and emits light. Additive blending is discussed in more detail in Section “Order-Independent Transparency”.

More examples of blend equations are given in Unity's ShaderLab reference about blending ^[1].

Shader Code

Here is a simple shader which uses alpha blending to render a green color with opacity 0.3:

```
Shader "Cg 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

            CGPROGRAM
                #pragma vertex vert
                #pragma fragment frag

                float4 vert(float4 vertexPos : POSITION) : SV_POSITION
                {
                    return mul(UNITY_MATRIX_MVP, vertexPos);
                }

                float4 frag(void) : COLOR
                {
```

```

        return float4(0.0, 1.0, 0.0, 0.3);
        // the fourth component (alpha) is important:
        // this is semitransparent green
    }

    ENDCG
}
}
}

```

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 Section “Per-Fragment Operations”, 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 [1].

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 [2].

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 Section “Order-Independent Transparency”. 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 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).

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 Section “Cutaways”, 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 "Cg shader using blending" {
    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

            CGPROGRAM
                #pragma vertex vert
                #pragma fragment frag

                float4 vert(float4 vertexPos : POSITION) : SV_POSITION
                {
                    return mul(UNITY_MATRIX_MVP, vertexPos);
                }

                float4 frag(void) : COLOR
                {
                    return float4(1.0, 0.0, 0.0, 0.3);
                    // the fourth component (alpha) is important:
                    // this is semitransparent red
                }
            ENDCG
        }
    }

    Pass {
        Cull Back // first 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 // use alpha blending

        CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            float4 vert(float4 vertexPos : POSITION) : SV_POSITION
            {
                return mul(UNITY_MATRIX_MVP, vertexPos);
            }
        
```

```
float4 frag(void) : COLOR
{
    return float4(0.0, 1.0, 0.0, 0.3);
    // the fourth component (alpha) is important:
    // this is semitransparent green
}

ENDCG
}
}

}
```

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.

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.

Further Reading

If you still want to know more

- the programmable graphics pipeline, you should read Section “Programmable Graphics Pipeline”.
- about per-fragment operations in the pipeline (e.g. blending and the depth test), you should read Section “Per-Fragment Operations”.
- about front-face and back-face culling, you should read Section “Cutaways”.
- about how to specify culling and the depth buffer functionality in Unity, you should read Unity's ShaderLab reference about culling and depth testing ^[1].
- about how to specify blending in Unity, you should read Unity's ShaderLab reference about blending ^[1].
- about the render queues in Unity, you should read Unity's ShaderLab reference about subshader tags ^[2].

page traffic for 90 days ^[3]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

- [1] <http://unity3d.com/support/documentation/Components/SL-Blend.html>
- [2] <http://unity3d.com/support/documentation/Components/SL-SubshaderTags.html>
- [3] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Transparency

2.3 Order-Independent Transparency



"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".

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

It continues the discussion in Section “Transparency” and solves some problems of standard transparency. If you haven't read that tutorial, you should read it first.

Order-Independent Blending

As noted in Section “Transparency”, 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.

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 Section “Transparency”:

```
float4 result = SrcFactor * fragment_output + 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 "Cg 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
        }
    }
}

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

float4 vert(float4 vertexPos : POSITION) : SV_POSITION
{
    return mul(UNITY_MATRIX_MVP, vertexPos);
}
```



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

```
        }
```

```
        float4 frag(void) : COLOR
```

```
{
```

```
        return float4(1.0, 0.0, 0.0, 0.3);
```

```
}
```

```
ENDCG
```

```
}
```

```
}
```

```
}
```

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 multiplicative 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 "Cg 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 SrcAlpha // multiplicative blending
            // for attenuation by the fragment's alpha
        }
    }
}

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

float4 vert(float4 vertexPos : POSITION) : SV_POSITION
{
    return mul(UNITY_MATRIX_MVP, vertexPos);
}

float4 frag(void) : COLOR
{
```

```

        return float4(1.0, 0.0, 0.0, 0.3);
    }

    ENDCG
}
}
}
```

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 "Cg 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

            CGPROGRAM

#pragma vertex vert
#pragma fragment frag

float4 vert(float4 vertexPos : POSITION) : SV_POSITION
{
    return mul(UNITY_MATRIX_MVP, vertexPos);
}

float4 frag(void) : COLOR
{
    return float4(1.0, 0.0, 0.0, 0.3);
}

ENDCG
}

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
}
```

```
CGPROGRAM

#pragma vertex vert
#pragma fragment frag

float4 vert(float4 vertexPos : POSITION) : SV_POSITION
{
    return mul(UNITY_MATRIX_MVP, vertexPos);
}

float4 frag(void) : COLOR
{
    return float4(1.0, 0.0, 0.0, 0.3);
}

ENDCG
}

}
```

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

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.

Further Reading

If you still want to know more

- about the shader code, you should read Section “Transparency”.
- 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 ^[1].

page traffic for 90 days ^[2]

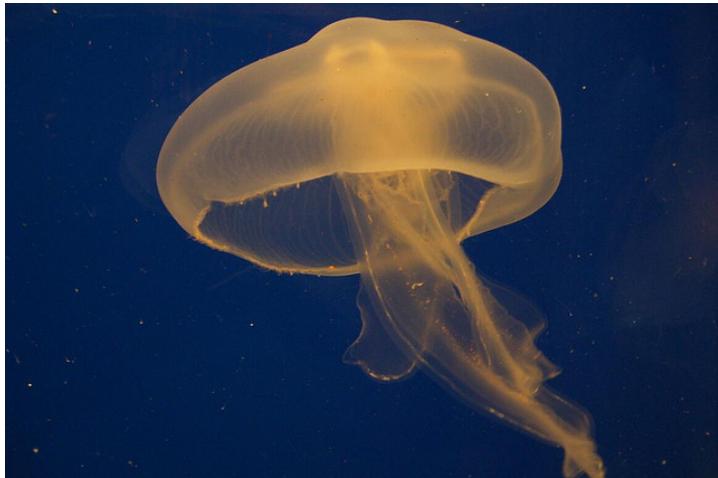
< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

- [1] <http://developer.nvidia.com/content/order-independent-transparency>
- [2] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Order-Independent_Transparency

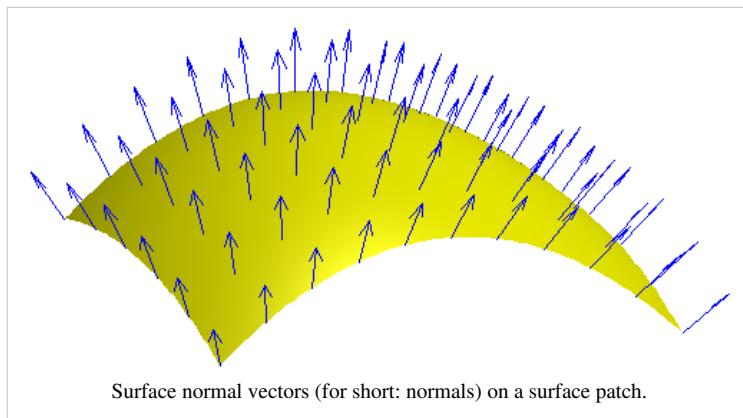
2.4 Silhouette Enhancement



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 Section “Transparency” and with shader properties as discussed in Section “Shading in World Space”.

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.



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 rarely be 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.

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 [1]) 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 GPUs usually 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.

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 often only available in the fragment shader. 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 Section “Specular Highlights”) and per-fragment lighting (i.e. Phong shading, which is discussed in Section “Smooth Specular Highlights”).

The next question is: in which coordinate system should the equation be implemented? (See Section “Vertex Transformations” 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 Section “Shading in World Space”). The normal vector `gl_Normal` is a standard vertex input parameter (see Section “Debugging of Shaders”). 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 Section “Shading in World Space”. The application of transformation matrices to points and normal vectors is discussed in detail in Section “Applying Matrix Transformations”. The basic result is that points and directions are transformed just by multiplying them with the transformation matrix, e.g. with `modelMatrix` set to `_Object2World`:

```
output.viewDir = normalize(_WorldSpaceCameraPos
    - float3(mul(modelMatrix, input.vertex)));
```

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` apart from the bottom-right element), 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 Section “Applying Matrix Transformations”:

```
output.normal = normalize(float3(
    mul(float4(input.normal, 0.0), modelMatrixInverse)));
```

Note that the incorrect bottom-right matrix element is no problem because it is always multiplied with 0. Moreover, the multiplication with `unity_Scale.w` is not necessary since the scaling doesn't matter because we normalize the vector.

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

Shader Code

```
Shader "Cg 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

            CGPROGRAM

                #pragma vertex vert
                #pragma fragment frag

                uniform float4 _Color; // define shader property for shaders

                // The following built-in uniforms are also defined in
                // "UnityCG.cginc", which could be #included
                uniform float4 unity_Scale; // w = 1/scale; see _World2Object
                uniform float3 _WorldSpaceCameraPos;
                uniform float4x4 _Object2World; // model matrix
                uniform float4x4 _World2Object; // inverse model matrix
                    // (all but the bottom-right element have to be scaled
                    // with unity_Scale.w if scaling is important)

                struct vertexInput {
                    float4 vertex : POSITION;
                    float3 normal : NORMAL;
                };
                struct vertexOutput {
```

```

        float4 pos : SV_POSITION;
        float3 normal : TEXCOORD;
        float3 viewDir : TEXCOORD1;
    };

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
    // multiplication with unity_Scale.w is unnecessary
    // because we normalize transformed vectors

    output.normal = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.viewDir = normalize(_WorldSpaceCameraPos
        - float3(mul(modelMatrix, input.vertex)));

    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normal);
    float3 viewDirection = normalize(input.viewDir);

    float newOpacity = min(1.0, _Color.a
        / abs(dot(viewDirection, normalDirection)));
    return float4(float3(_Color), newOpacity);
}

ENDCG
}
}
}

```

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 vertex output parameters `output.normal` and `output.viewDir` 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 `output.normal` in the vertex shader is not necessary. Similarly, the normalization of `output.viewDir` in the fragment shader is in most cases unnecessary.

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 $|V \cdot N|$ to the power of this number (using the built-in Cg 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.

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 .

Further Reading

If you still want to know more

- about object space and world space, you should read the description in Section “Vertex Transformations”.
- about how to apply transformation matrices to points, directions and normal vectors, you should read Section “Applying Matrix Transformations”.
- about the basics of rendering transparent objects, you should read Section “Transparency”.
- about uniform variables provided by Unity and shader properties, you should read Section “Shading in World Space”.
- 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^[1].

page traffic for 90 days^[2]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

- [1] <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.125.1928>
[2] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Silhouette_Enhancement

3 Basic Lighting

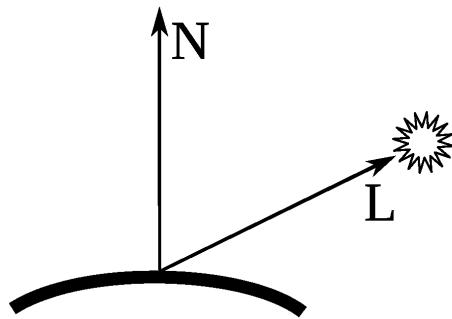
3.1 Diffuse Reflection



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.



Diffuse reflection can be computed using the surface normal vector \mathbf{N} and the light vector \mathbf{L} , i.e. the vector to the light source.

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 to the left, 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 \mathbf{N} is orthogonal to the surface and the normalized light direction \mathbf{L} points to the light source.

vector \mathbf{N} is orthogonal to the surface and the normalized light direction \mathbf{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 \mathbf{N} and the normalized direction to the light source \mathbf{L} , which is the dot product $\mathbf{N} \cdot \mathbf{L}$ because the dot product $\mathbf{a} \cdot \mathbf{b}$ of any two vectors \mathbf{a} and \mathbf{b} is:

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

In the case of normalized vectors, the lengths $|\mathbf{a}|$ and $|\mathbf{b}|$ are both 1.

If the dot product $\mathbf{N} \cdot \mathbf{L}$ 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 Cg), this equation also applies to colored light. This is what we actually use in the shader code.

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 Section “Silhouette Enhancement”:

- Should the equation be implemented in the vertex shader or the fragment shader? We try the vertex shader here. In Section “Smooth Specular Highlights”, 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 Section “Shading in World Space”) 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 Section “Shading in World Space”, we have to tag the shader pass with `Tags { "LightMode" = "ForwardBase" }` to make sure that these uniforms have the correct values. (Below we will discuss what this tag actually means.) We get the surface normal vector in object coordinates from the vertex input parameter with semantic `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 Section “Silhouette Enhancement”.

The shader code then looks like this:

```
Shader "Cg 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
        }
    }
    CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag

        uniform float4 _Color; // define shader property for shaders

        // The following built-in uniforms (apart from _LightColor0)
        // are defined in "UnityCG.cginc", which could be #included
        uniform float4 unity_Scale; // w = 1/scale; see _World2Object
    
```

```
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 col : COLOR;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
        // multiplication with unity_Scale.w is unnecessary
        // because we normalize transformed vectors

    float3 normalDirection = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    float3 lightDirection = normalize(
        float3(_WorldSpaceLightPos0));

    float3 diffuseReflection =
        float3(_LightColor0) * float3(_Color)
        * max(0.0, dot(normalDirection, lightDirection));

    output.col = float4(diffuseReflection, 1.0);
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    return input.col;
}
```

```

        ENDCG
    }
}

// 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”.)

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 ^[1]. 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 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.

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 to render them as pixel lights. (See Section “Multiple Lights” 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 configure the blending to add the new fragment output color to the color in the framebuffer. As discussed in Section “Transparency”, 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.

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

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

            CGPROGRAM

#pragma vertex vert
#pragma fragment frag

uniform float4 _Color; // define shader property for shaders

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 col : COLOR;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;
```

```

        float4x4 modelMatrix = _Object2World;
        float4x4 modelMatrixInverse = _World2Object;
        // multiplication with unity_Scale.w is unnecessary
        // because we normalize transformed vectors

        float3 normalDirection = normalize(float3(
            mul(float4(input.normal, 0.0), modelMatrixInverse)));
        float3 lightDirection = normalize(
            float3(_WorldSpaceLightPos0));

        float3 diffuseReflection =
            float3(_LightColor0) * float3(_Color)
            * max(0.0, dot(normalDirection, lightDirection));

        output.col = float4(diffuseReflection, 1.0);
        output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
        return output;
    }

    float4 frag(vertexOutput input) : COLOR
    {
        return input.col;
    }

    ENDCG
}

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

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

uniform float4 _Color; // define shader property for shaders

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)

```

```
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 col : COLOR;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
        // multiplication with unity_Scale.w is unnecessary
        // because we normalize transformed vectors

    float3 normalDirection = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    float3 lightDirection = normalize(
        float3(_WorldSpaceLightPos0));

    float3 diffuseReflection =
        float3(_LightColor0) * float3(_Color)
        * max(0.0, dot(normalDirection, lightDirection));

    output.col = float4(diffuseReflection, 1.0);
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    return input.col;
}

ENDCG
}
```

// 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.

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:

```
float3 lightDirection;

if (0.0 == _WorldSpaceLightPos0.w) // directional light?
{
    lightDirection = normalize(float3(_WorldSpaceLightPos0));
}
else // point or spot light
{
    lightDirection = normalize(float3(_WorldSpaceLightPos0
        - mul(modelMatrix, input.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:

```
float3 lightDirection;
float attenuation;

if (0.0 == _WorldSpaceLightPos0.w) // directional light?
{
    attenuation = 1.0; // no attenuation
    lightDirection = normalize(float3(_WorldSpaceLightPos0));
}
else // point or spot light
{
    float3 vertexToLightSource = float3(_WorldSpaceLightPos0
        - mul(modelMatrix, input.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:

```
float3 vertexToLightSource =
    float3 (_WorldSpaceLightPos0 - mul(modelMatrix,
        input.vertex * _WorldSpaceLightPos0.w);
float one_over_distance =
    1.0 / length(vertexToLightSource);
float attenuation =
    mix(1.0, one_over_distance, _WorldSpaceLightPos0.w);
float3 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 "Cg per-vertex diffuse lighting" {
Properties {
    _Color ("Diffuse Material Color", Color) = (1,1,1,1)
}
SubShader {
    Pass {
        Tags { "LightMode" = "ForwardBase" }
        // pass for first light source

        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag

        uniform float4 _Color; // define shader property for shaders

        // The following built-in uniforms (apart from _LightColor0)
        // are defined in "UnityCG.cginc", which could be #included
        uniform float4 unity_Scale; // w = 1/scale; see _World2Object
        uniform float3 _WorldSpaceCameraPos;
        uniform float4x4 _Object2World; // model matrix
        uniform float4x4 _World2Object; // inverse model matrix
        // (all but the bottom-right element have to be scaled
        // with unity_Scale.w if scaling is important)
        uniform float4 _WorldSpaceLightPos0;
        // position or direction of light source
        uniform float4 _LightColor0;
```

```
// color of light source (from "Lighting.cginc")  
  
struct vertexInput {  
    float4 vertex : POSITION;  
    float3 normal : NORMAL;  
};  
struct vertexOutput {  
    float4 pos : SV_POSITION;  
    float4 col : COLOR;  
};  
  
vertexOutput vert(vertexInput input)  
{  
    vertexOutput output;  
  
    float4x4 modelMatrix = _Object2World;  
    float4x4 modelMatrixInverse = _World2Object;  
    // multiplication with unity_Scale.w is unnecessary  
    // because we normalize transformed vectors  
  
    float3 normalDirection = normalize(float3(  
        mul(float4(input.normal, 0.0), modelMatrixInverse)));  
    float3 lightDirection;  
    float attenuation;  
  
    if (0.0 == _WorldSpaceLightPos0.w) // directional light?  
    {  
        attenuation = 1.0; // no attenuation  
        lightDirection =  
            normalize(float3(_WorldSpaceLightPos0));  
    }  
    else // point or spot light  
    {  
        float3 vertexToLightSource = float3(_WorldSpaceLightPos0  
            - mul(modelMatrix, input.vertex));  
        float distance = length(vertexToLightSource);  
        attenuation = 1.0 / distance; // linear attenuation  
        lightDirection = normalize(vertexToLightSource);  
    }  
  
    float3 diffuseReflection =  
        attenuation * float3(_LightColor0) * float3(_Color)  
        * max(0.0, dot(normalDirection, lightDirection));  
  
    output.col = float4(diffuseReflection, 1.0);  
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);  
    return output;  
}
```

```
}

float4 frag(vertexOutput input) : COLOR
{
    return input.col;
}

ENDCG
}

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

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

uniform float4 _Color; // define shader property for shaders

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 col : COLOR;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;
```

```
float4x4 modelMatrix = _Object2World;
float4x4 modelMatrixInverse = _World2Object;
// multiplication with unity_Scale.w is unnecessary
// because we normalize transformed vectors

float3 normalDirection = normalize(float3(
    mul(float4(input.normal, 0.0), modelMatrixInverse)));
float3 lightDirection;
float attenuation;

if (0.0 == _WorldSpaceLightPos0.w) // directional light?
{
    attenuation = 1.0; // no attenuation
    lightDirection =
        normalize(float3(_WorldSpaceLightPos0));
}
else // point or spot light
{
    float3 vertexToLightSource = float3(_WorldSpaceLightPos0
        - mul(modelMatrix, input.vertex));
    float distance = length(vertexToLightSource);
    attenuation = 1.0 / distance; // linear attenuation
    lightDirection = normalize(vertexToLightSource);
}

float3 diffuseReflection =
    attenuation * float3(_LightColor0) * float3(_Color)
    * max(0.0, dot(normalDirection, lightDirection));

output.col = float4(diffuseReflection, 1.0);
output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
return output;
}

float4 frag(vertexOutput input) : COLOR
{
    return input.col;
}

ENDCG
}
}

// 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 Cg 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**.

Changes for a Spotlight

Unity implements spotlights with the help of cookie textures as described in Section “Cookies”; however, this is somewhat advanced. Here, we treat spotlights as if they were point lights.

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.
- How to implement diffuse reflection for a single directional light source in a shader.
- How to extend the shader for point light sources with linear attenuation.
- How to further extend the shader to handle multiple per-pixel lights.

Further Reading

If you still want to know more

- about transforming normal vectors into world space, you should read Section “Shading in World Space”.
- about uniform variables provided by Unity and shader properties, you should read Section “Shading in World Space”.
- about (additive) blending, you should read Section “Transparency”.
- about pass tags in Unity (e.g. `ForwardBase` or `ForwardAdd`), you should read Unity's ShaderLab reference about pass tags ^[2].
- about how Unity processes light sources in general, you should read Unity's manual about rendering paths ^[3].
page traffic for 90 days ^[4]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

- [1] <http://unity3d.com/support/resources/assets/built-in-shaders>
- [2] <http://unity3d.com/support/documentation/Components/SL-PassTags.html>
- [3] <http://unity3d.com/support/documentation/Manual/RenderingPaths.html>
- [4] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Diffuse_Reflection

3.2 Specular Highlights



"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 Section "Diffuse Reflection" by two additional terms: ambient lighting and specular reflection. Together, the three terms constitute the Phong reflection model. If you haven't read Section "Diffuse Reflection", this would be a very good opportunity to read it.

Ambient Light

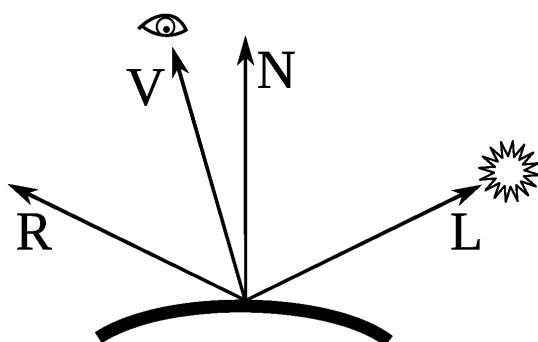
Consider the painting by Caravaggio to the left. While large parts of the white shirt are in shadows, no part of it is completely black. Apparently there is always some light being reflected from walls and other objects to

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 Section "Diffuse Reflection", 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 Cg shader in Unity, this color is always available as `UNITY_LIGHTMODEL_AMBIENT`, which is one of the pre-defined uniforms mentioned in Section "Shading in World Space".



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 .

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 \mathbf{N} and a normalized direction to the light source \mathbf{L} . In Cg, the function `float3 reflect(float3 I, float3 N)` (or `float4 reflect(float4 I, float4 N)`) computes the same reflected vector but for the direction \mathbf{I} from the light source to the point on the surface. Thus, we have to negate our direction \mathbf{L} to use this function.

The specular reflection term computes the specular reflection in the direction of the viewer \mathbf{V} . As discussed above, the intensity should be large if \mathbf{V} is close to \mathbf{R} , where “closeness” is parametrized by the shininess $n_{\text{shininess}}$. In the Phong reflection model, the cosine of the angle between \mathbf{R} and \mathbf{V} to the $n_{\text{shininess}}$ -th power is used to generate highlights of different shininess. Similarly to the case of the diffuse reflection, 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, the specular term should be ignored if the light source is on the “wrong” side of the surface; i.e., if the dot product $\mathbf{N} \cdot \mathbf{L}$ is negative.

Shader Code

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

```
float3 ambientLighting =
    float3(UNITY_LIGHTMODEL_AMBIENT) * float3(_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 Section “Diffuse Reflection”. The equation of the specular term in world space could then be implemented like this:

```
float3 viewDirection = normalize(float3(
    float4(_WorldSpaceCameraPos, 1.0)
    - mul(modelMatrix, input.vertex)));
    
float3 specularReflection;
if (dot(normalDirection, lightDirection) < 0.0)
    // light source on the wrong side?
{
    specularReflection = float3(0.0, 0.0, 0.0);
    // no specular reflection
}
else // light source on the right side
{
    specularReflection = attenuation * float3(_LightColor0)
        * float3(_SpecColor) * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _Shininess);
}
```

This code snippet uses the same variables as the shader code in Section “Diffuse Reflection” 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 Section “Diffuse Reflection”.) $\text{pow}(a, b)$ 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 Section “Diffuse Reflection”, it looks like this:

```
Shader "Cg 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

            CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 col : COLOR;
};

vertexOutput vert(vertexInput input)
```

```
{\n    vertexOutput output;\n\n    float4x4 modelMatrix = _Object2World;\n    float4x4 modelMatrixInverse = _World2Object;\n        // multiplication with unity_Scale.w is unnecessary\n        // because we normalize transformed vectors\n\n    float3 normalDirection = normalize(float3(\n        mul(float4(input.normal, 0.0), modelMatrixInverse)));\n    float3 viewDirection = normalize(float3(\n        float4(_WorldSpaceCameraPos, 1.0)\n        - mul(modelMatrix, input.vertex)));\n    float3 lightDirection;\n    float attenuation;\n\n    if (0.0 == _WorldSpaceLightPos0.w) // directional light?\n    {\n        attenuation = 1.0; // no attenuation\n        lightDirection =\n            normalize(float3(_WorldSpaceLightPos0));\n    }\n    else // point or spot light\n    {\n        float3 vertexToLightSource = float3(_WorldSpaceLightPos0\n            - mul(modelMatrix, input.vertex));\n        float distance = length(vertexToLightSource);\n        attenuation = 1.0 / distance; // linear attenuation\n        lightDirection = normalize(vertexToLightSource);\n    }\n\n    float3 ambientLighting =\n        float3(UNITY_LIGHTMODEL_AMBIENT) * float3(_Color);\n\n    float3 diffuseReflection =\n        attenuation * float3(_LightColor0) * float3(_Color)\n        * max(0.0, dot(normalDirection, lightDirection));\n\n    float3 specularReflection;\n    if (dot(normalDirection, lightDirection) < 0.0)\n        // light source on the wrong side?\n    {\n        specularReflection = float3(0.0, 0.0, 0.0);\n        // no specular reflection\n    }\n    else // light source on the right side\n    {
```

```
        specularReflection = attenuation * float3(_LightColor0)
            * float3(_SpecColor) * pow(max(0.0, dot(
                reflect(-lightDirection, normalDirection),
                viewDirection)), _Shininess);
    }

    output.col = float4(ambientLighting + diffuseReflection
        + specularReflection, 1.0);
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    return input.col;
}

ENDCG
}

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

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")
```

```
struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 col : COLOR;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
    // multiplication with unity_Scale.w is unnecessary
    // because we normalize transformed vectors

    float3 normalDirection = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    float3 viewDirection = normalize(float3(
        float4(_WorldSpaceCameraPos, 1.0)
        - mul(modelMatrix, input.vertex)));
    float3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else // point or spot light
    {
        float3 vertexToLightSource = float3(_WorldSpaceLightPos0
            - mul(modelMatrix, input.vertex));
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
        lightDirection = normalize(vertexToLightSource);
    }

    float3 diffuseReflection =
        attenuation * float3(_LightColor0) * float3(_Color)
        * max(0.0, dot(normalDirection, lightDirection));

    float3 specularReflection;
```

```

        if (dot(normalDirection, lightDirection) < 0.0)
            // light source on the wrong side?
        {
            specularReflection = float3(0.0, 0.0, 0.0);
            // no specular reflection
        }
        else // light source on the right side
        {
            specularReflection = attenuation * float3(_LightColor0)
                * float3(_SpecColor) * pow(max(0.0, dot(
                    reflect(-lightDirection, normalDirection),
                    viewDirection)), _Shininess);
        }

        output.col = float4(diffuseReflection
            + specularReflection, 1.0);
            // no ambient contribution in this pass
        output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
        return output;
    }

    float4 frag(vertexOutput input) : COLOR
    {
        return input.col;
    }

    ENDCG
}
}

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

```

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 Cg in Unity.

Further Reading

If you still want to know more

- about the shader code, you should read Section “Diffuse Reflection”.

page traffic for 90 days ^[1]

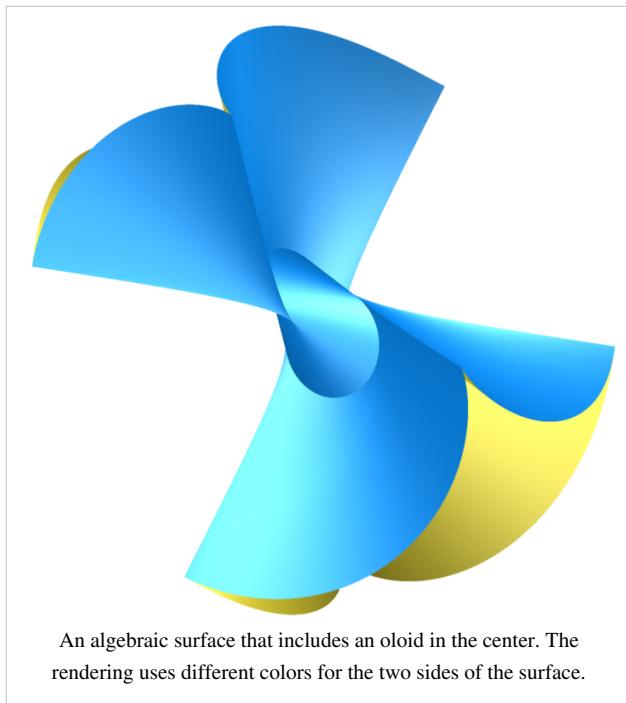
< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Specular_Highlights

3.3 Two-Sided Surfaces



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 Section "Specular Highlights" to render two-sided surfaces. If you haven't read Section "Specular Highlights", this would be a very good time to read it.

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 Section "Cutaways", we have seen how two passes with front face culling and back face culling can be used to apply different shaders to the two sides of a mesh. We will apply the same strategy here.

As mentioned in Section "Cutaways", an alternative approach in Cg is to use a fragment input parameter with semantic FACE, VFACE, or

`SV_IsFrontFacing` (depending on the API) to distinguish between the two sides, however, this doesn't seem to work in Unity.

Shader Code

The shader code for two-sided per-vertex lighting is a straightforward extension of the code in Section "Specular Highlights". It requires two sets of material parameters (front and back) and duplicates all passes — one copy with front-face culling and the other with back-face culling. The shaders of the two copies are identical except that the shader for the back faces uses the negated surface normal vector and the properties for the back material.

```
Shader "Cg 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 Back // render only front faces

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;
uniform float4 _BackColor;
uniform float4 _BackSpecColor;
uniform float _BackShininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 col : COLOR;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
        // multiplication with unity_Scale.w is unnecessary
        // because we normalize transformed vectors

    float3 normalDirection = normalize(float3(
```

```
mul(float4(input.normal, 0.0), modelMatrixInverse));  
float3 viewDirection = normalize(float3(  
    float4(_WorldSpaceCameraPos, 1.0)  
    - mul(modelMatrix, input.vertex)));  
float3 lightDirection;  
float attenuation;  
  
if (0.0 == _WorldSpaceLightPos0.w) // directional light?  
{  
    attenuation = 1.0; // no attenuation  
    lightDirection =  
        normalize(float3(_WorldSpaceLightPos0));  
}  
else // point or spot light  
{  
    float3 vertexToLightSource = float3(_WorldSpaceLightPos0  
        - mul(modelMatrix, input.vertex));  
    float distance = length(vertexToLightSource);  
    attenuation = 1.0 / distance; // linear attenuation  
    lightDirection = normalize(vertexToLightSource);  
}  
  
float3 ambientLighting =  
    float3(UNITY_LIGHTMODEL_AMBIENT) * float3(_Color);  
  
float3 diffuseReflection =  
    attenuation * float3(_LightColor0) * float3(_Color)  
    * max(0.0, dot(normalDirection, lightDirection));  
  
float3 specularReflection;  
if (dot(normalDirection, lightDirection) < 0.0)  
    // light source on the wrong side?  
{  
    specularReflection = float3(0.0, 0.0, 0.0);  
    // no specular reflection  
}  
else // light source on the right side  
{  
    specularReflection = attenuation * float3(_LightColor0)  
        * float3(_SpecColor) * pow(max(0.0, dot(  
            reflect(-lightDirection, normalDirection),  
            viewDirection)), _Shininess);  
}  
  
output.col = float4(ambientLighting + diffuseReflection  
    + specularReflection, 1.0);  
output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
```

```
        return output;
    }

    float4 frag(vertexOutput input) : COLOR
    {
        return input.col;
    }

    ENDCG
}

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

    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag

    // User-specified properties
    uniform float4 _Color;
    uniform float4 _SpecColor;
    uniform float _Shininess;
    uniform float4 _BackColor;
    uniform float4 _BackSpecColor;
    uniform float _BackShininess;

    // The following built-in uniforms (apart from _LightColor0)
    // are defined in "UnityCG.cginc", which could be #included
    uniform float4 unity_Scale; // w = 1/scale; see _World2Object
    uniform float3 _WorldSpaceCameraPos;
    uniform float4x4 _Object2World; // model matrix
    uniform float4x4 _World2Object; // inverse model matrix
        // (all but the bottom-right element have to be scaled
        // with unity_Scale.w if scaling is important)
    uniform float4 _WorldSpaceLightPos0;
        // position or direction of light source
    uniform float4 _LightColor0;
        // color of light source (from "Lighting.cginc")

    struct vertexInput {
        float4 vertex : POSITION;
        float3 normal : NORMAL;
    };
}
```

```
struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 col : COLOR;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
    // multiplication with unity_Scale.w is unnecessary
    // because we normalize transformed vectors

    float3 normalDirection = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    float3 viewDirection = normalize(float3(
        float4(_WorldSpaceCameraPos, 1.0)
        - mul(modelMatrix, input.vertex)));
    float3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else // point or spot light
    {
        float3 vertexToLightSource = float3(_WorldSpaceLightPos0
            - mul(modelMatrix, input.vertex));
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
        lightDirection = normalize(vertexToLightSource);
    }

    float3 diffuseReflection =
        attenuation * float3(_LightColor0) * float3(_Color)
        * max(0.0, dot(normalDirection, lightDirection));

    float3 specularReflection;
    if (dot(normalDirection, lightDirection) < 0.0)
        // light source on the wrong side?
    {
        specularReflection = float3(0.0, 0.0, 0.0);
        // no specular reflection
    }
}
```

```
        }

    else // light source on the right side
    {
        specularReflection = attenuation * float3(_LightColor0)
            * float3(_SpecColor) * pow(max(0.0, dot(
                reflect(-lightDirection, normalDirection),
                viewDirection)), _Shininess);
    }

    output.col = float4(diffuseReflection
        + specularReflection, 1.0);
    // no ambient contribution in this pass
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    return input.col;
}

ENDCG
}

Pass {
    Tags { "LightMode" = "ForwardBase" }
    // pass for ambient light and first light source
    Cull Front// render only back faces

    CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;
uniform float4 _BackColor;
uniform float4 _BackSpecColor;
uniform float _BackShininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
```

```
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 col : COLOR;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
        // multiplication with unity_Scale.w is unnecessary
        // because we normalize transformed vectors

    float3 normalDirection = normalize(float3(
        mul(float4(-input.normal, 0.0), modelMatrixInverse)));
        // negate input.normal for the back faces
    float3 viewDirection = normalize(float3(
        float4(_WorldSpaceCameraPos, 1.0)
        - mul(modelMatrix, input.vertex)));
    float3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else // point or spot light
    {
        float3 vertexToLightSource = float3(_WorldSpaceLightPos0
            - mul(modelMatrix, input.vertex));
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
```

```
        lightDirection = normalize(vertexToLightSource);
    }

    float3 ambientLighting =
        float3(UNITY_LIGHTMODEL_AMBIENT) * float3(_BackColor);

    float3 diffuseReflection =
        attenuation * float3(_LightColor0) * float3(_BackColor)
        * max(0.0, dot(normalDirection, lightDirection));

    float3 specularReflection;
    if (dot(normalDirection, lightDirection) < 0.0)
        // light source on the wrong side?
    {
        specularReflection = float3(0.0, 0.0, 0.0);
        // no specular reflection
    }
    else // light source on the right side
    {
        specularReflection = attenuation * float3(_LightColor0)
            * float3(_BackSpecColor) * pow(max(0.0, dot(
                reflect(-lightDirection, normalDirection),
                viewDirection)), _BackShininess);
    }

    output.col = float4(ambientLighting + diffuseReflection
        + specularReflection, 1.0);
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    return input.col;
}

ENDCG
}

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

    CGPROGRAM
```

```
#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;
uniform float4 _BackColor;
uniform float4 _BackSpecColor;
uniform float _BackShininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 col : COLOR;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
        // multiplication with unity_Scale.w is unnecessary
        // because we normalize transformed vectors

    float3 normalDirection = normalize(float3(
        mul(float4(-input.normal, 0.0), modelMatrixInverse)));
        // negate input.normal for the back faces
    float3 viewDirection = normalize(float3(
        float4(_WorldSpaceCameraPos, 1.0)
```

```
        - mul(modelMatrix, input.vertex));
float3 lightDirection;
float attenuation;

if (0.0 == _WorldSpaceLightPos0.w) // directional light?
{
    attenuation = 1.0; // no attenuation
    lightDirection =
        normalize(float3(_WorldSpaceLightPos0));
}
else // point or spot light
{
    float3 vertexToLightSource = float3(_WorldSpaceLightPos0
        - mul(modelMatrix, input.vertex));
    float distance = length(vertexToLightSource);
    attenuation = 1.0 / distance; // linear attenuation
    lightDirection = normalize(vertexToLightSource);
}

float3 diffuseReflection =
    attenuation * float3(_LightColor0) * float3(_BackColor)
    * max(0.0, dot(normalDirection, lightDirection));

float3 specularReflection;
if (dot(normalDirection, lightDirection) < 0.0)
    // light source on the wrong side?
{
    specularReflection = float3(0.0, 0.0, 0.0);
    // no specular reflection
}
else // light source on the right side
{
    specularReflection = attenuation * float3(_LightColor0)
        * float3(_BackSpecColor) * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _BackShininess);
}

output.col = float4(diffuseReflection
    + specularReflection, 1.0);
// no ambient contribution in this pass
output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
return output;
}

float4 frag(vertexOutput input) : COLOR
{
```

```
        return input.col;
    }

    ENDCG
}

}

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

This code consists of four passes, where the first pair of passes render the front faces, and the second pair of passes renders the back faces using the negated normal vector and the back material properties. The second pass of each pair is the same as the first apart from the additive blending and the missing ambient color.

Summary

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

- How to use front-face culling and back-face culling to apply different shaders on the two sides of a mesh.
- How to change the Phong lighting computation for back-facing triangles.

Further Reading

If you still want to know more

- about the shader version for single-sided surfaces, you should read Section “Specular Highlights”.
- about front-facing and back-facing triangles in Cg, you should read Section “Cutaways”.

page traffic for 90 days ^[1]

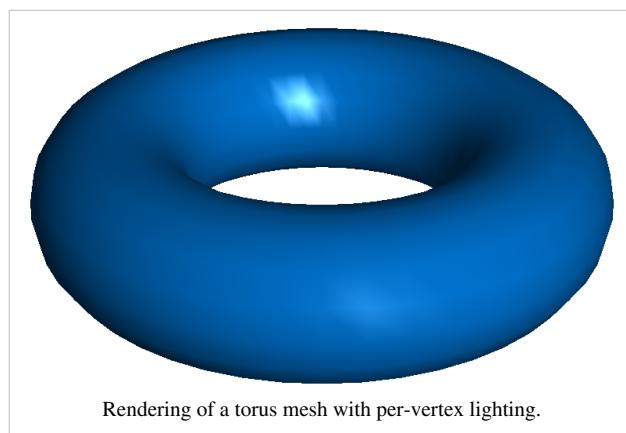
< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

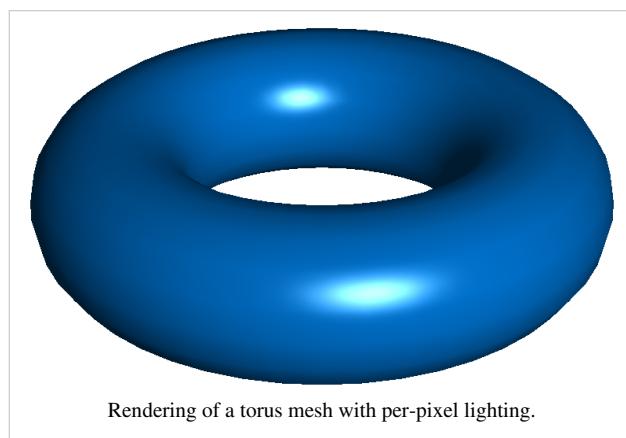
[1] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Two-Sided_Surfaces

3.4 Smooth Specular Highlights



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

It is based on Section “Specular Highlights”. 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 to the left. 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.



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 Section “Specular Highlights”.

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.

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 vertex input parameters required for the lighting computation to the vertex output parameters. The fragment shader then uses these parameters to compute the lighting. That's about it.

In this tutorial, we adapt the shader code from Section “Specular Highlights” to per-pixel lighting. The result looks like this:

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

```
CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
// (all but the bottom-right element have to be scaled
// with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
// position or direction of light source
uniform float4 _LightColor0;
// color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    float3 normalDir : TEXCOORD1;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
// multiplication with unity_Scale.w is unnecessary
// because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
}
```

```
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);

    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));
    float3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else // point or spot light
    {
        float3 vertexToLightSource =
            float3(_WorldSpaceLightPos0 - input.posWorld);
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
        lightDirection = normalize(vertexToLightSource);
    }

    float3 ambientLighting =
        float3(UNITY_LIGHTMODEL_AMBIENT) * float3(_Color);

    float3 diffuseReflection =
        attenuation * float3(_LightColor0) * float3(_Color)
        * max(0.0, dot(normalDirection, lightDirection));

    float3 specularReflection;
    if (dot(normalDirection, lightDirection) < 0.0)
        // light source on the wrong side?
    {
        specularReflection = float3(0.0, 0.0, 0.0);
        // no specular reflection
    }
    else // light source on the right side
    {
        specularReflection = attenuation * float3(_LightColor0)
            * float3(_SpecColor) * pow(max(0.0, dot(
                reflect(-lightDirection, normalDirection),
                viewDirection)), _Shininess);
    }
}
```

```
    }

    return float4(ambientLighting + diffuseReflection
        + specularReflection, 1.0);
}

ENDCG
}

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

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    float3 normalDir : TEXCOORD1;
};
```

```
vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
    // multiplication with unity_Scale.w is unnecessary
    // because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);

    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));
    float3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else // point or spot light
    {
        float3 vertexToLightSource =
            float3(_WorldSpaceLightPos0 - input.posWorld);
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
        lightDirection = normalize(vertexToLightSource);
    }

    float3 diffuseReflection =
        attenuation * float3(_LightColor0) * float3(_Color)
        * max(0.0, dot(normalDirection, lightDirection));

    float3 specularReflection;
    if (dot(normalDirection, lightDirection) < 0.0)
        // light source on the wrong side?
```

```

    {
        specularReflection = float3(0.0, 0.0, 0.0);
        // no specular reflection
    }
    else // light source on the right side
    {
        specularReflection = attenuation * float3(_LightColor0)
            * float3(_SpecColor) * pow(max(0.0, dot(
                reflect(-lightDirection, normalDirection),
                viewDirection)), _Shininess);
    }

    return float4(diffuseReflection
        + specularReflection, 1.0);
    // no ambient lighting in this pass
}

ENDCG
}
}

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

```

Note that the vertex shader writes a normalized vector to `output.normalDir` 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.

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.

Further Reading

If you still want to know more

- about the shader version for per-vertex lighting, you should read Section “Specular Highlights”.

page traffic for 90 days^[1]

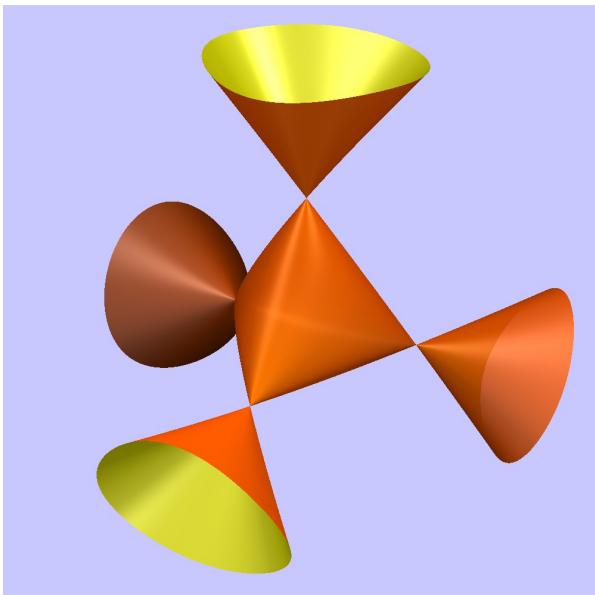
< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Smooth_Specular_Highlights

3.5 Two-Sided Smooth Surfaces



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 Section “Smooth Specular Highlights” with the two-sided lighting discussed in Section “Two-Sided Surfaces”.

Shader Coder

The required changes to the code of Section “Smooth Specular Highlights” are: new properties for the back material, duplication of the passes with front-face culling for one copy and back-face culling for the other copy, and negation of the surface normal vector for the rendering of the back faces. It’s actually quite straightforward. The code looks like this:

```
Shader "Cg two-sided 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
    _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 Back // render only front faces
        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        // User-specified properties
    }
}
```

```
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;
uniform float4 _BackColor;
uniform float4 _BackSpecColor;
uniform float _BackShininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
// (all but the bottom-right element have to be scaled
// with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
// position or direction of light source
uniform float4 _LightColor0;
// color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    float3 normalDir : TEXCOORD1;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
// multiplication with unity_Scale.w is unnecessary
// because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
```

```
{\n    float3 normalDirection = normalize(input.normalDir);\n\n    float3 viewDirection = normalize(\n        _WorldSpaceCameraPos - float3(input.posWorld));\n    float3 lightDirection;\n    float attenuation;\n\n    if (0.0 == _WorldSpaceLightPos0.w) // directional light?\n    {\n        attenuation = 1.0; // no attenuation\n        lightDirection =\n            normalize(float3(_WorldSpaceLightPos0));\n    }\n    else // point or spot light\n    {\n        float3 vertexToLightSource =\n            float3(_WorldSpaceLightPos0 - input.posWorld);\n        float distance = length(vertexToLightSource);\n        attenuation = 1.0 / distance; // linear attenuation\n        lightDirection = normalize(vertexToLightSource);\n    }\n\n    float3 ambientLighting =\n        float3(UNITY_LIGHTMODEL_AMBIENT) * float3(_Color);\n\n    float3 diffuseReflection =\n        attenuation * float3(_LightColor0) * float3(_Color)\n        * max(0.0, dot(normalDirection, lightDirection));\n\n    float3 specularReflection;\n    if (dot(normalDirection, lightDirection) < 0.0)\n        // light source on the wrong side?\n    {\n        specularReflection = float3(0.0, 0.0, 0.0);\n        // no specular reflection\n    }\n    else // light source on the right side\n    {\n        specularReflection = attenuation * float3(_LightColor0)\n            * float3(_SpecColor) * pow(max(0.0, dot(\n                reflect(-lightDirection, normalDirection),\n                viewDirection)), _Shininess);\n    }\n\n    return float4(ambientLighting + diffuseReflection\n        + specularReflection, 1.0);
```

```
}

ENDCG

}

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

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;
uniform float4 _BackColor;
uniform float4 _BackSpecColor;
uniform float _BackShininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    float3 normalDir : TEXCOORD1;
};
```

```
vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
    // multiplication with unity_Scale.w is unnecessary
    // because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);

    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));
    float3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else // point or spot light
    {
        float3 vertexToLightSource =
            float3(_WorldSpaceLightPos0 - input.posWorld);
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
        lightDirection = normalize(vertexToLightSource);
    }

    float3 diffuseReflection =
        attenuation * float3(_LightColor0) * float3(_Color)
        * max(0.0, dot(normalDirection, lightDirection));

    float3 specularReflection;
    if (dot(normalDirection, lightDirection) < 0.0)
        // light source on the wrong side?
```

```
{  
    specularReflection = float3(0.0, 0.0, 0.0);  
    // no specular reflection  
}  
else // light source on the right side  
{  
    specularReflection = attenuation * float3(_LightColor0)  
        * float3(_SpecColor) * pow(max(0.0, dot(  
            reflect(-lightDirection, normalDirection),  
            viewDirection)), _Shininess);  
}  
  
return float4(diffuseReflection  
    + specularReflection, 1.0);  
// no ambient lighting in this pass  
}  
  
ENDCG  
}  
  
Pass {  
    Tags { "LightMode" = "ForwardBase" }  
    // pass for ambient light and first light source  
    Cull Front // render only back faces  
  
    CGPROGRAM  
  
#pragma vertex vert  
#pragma fragment frag  
  
// User-specified properties  
uniform float4 _Color;  
uniform float4 _SpecColor;  
uniform float _Shininess;  
uniform float4 _BackColor;  
uniform float4 _BackSpecColor;  
uniform float _BackShininess;  
  
// The following built-in uniforms (apart from _LightColor0)  
// are defined in "UnityCG.cginc", which could be #included  
uniform float4 unity_Scale; // w = 1/scale; see _World2Object  
uniform float3 _WorldSpaceCameraPos;  
uniform float4x4 _Object2World; // model matrix  
uniform float4x4 _World2Object; // inverse model matrix  
    // (all but the bottom-right element have to be scaled  
    // with unity_Scale.w if scaling is important)  
uniform float4 _WorldSpaceLightPos0;
```

```
// position or direction of light source
uniform float4 _LightColor0;
// color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    float3 normalDir : TEXCOORD1;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
    // multiplication with unity_Scale.w is unnecessary
    // because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.normalDir = normalize(float3(
        mul(float4(-input.normal, 0.0), modelMatrixInverse)));
    // negate input.normal for the back faces
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);

    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));
    float3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else // point or spot light
```

```

    {

        float3 vertexToLightSource =
            float3(_WorldSpaceLightPos0 - input.posWorld);
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
        lightDirection = normalize(vertexToLightSource);
    }

    float3 ambientLighting =
        float3(UNITY_LIGHTMODEL_AMBIENT) * float3(_BackColor);

    float3 diffuseReflection =
        attenuation * float3(_LightColor0) * float3(_BackColor)
        * max(0.0, dot(normalDirection, lightDirection));

    float3 specularReflection;
    if (dot(normalDirection, lightDirection) < 0.0)
        // light source on the wrong side?
    {
        specularReflection = float3(0.0, 0.0, 0.0);
        // no specular reflection
    }
    else // light source on the right side
    {
        specularReflection = attenuation * float3(_LightColor0)
            * float3(_BackSpecColor) * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _BackShininess);
    }

    return float4(ambientLighting + diffuseReflection
        + specularReflection, 1.0);
}

ENDCG
}

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

    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag
}

```

```
// User-specified properties
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;
uniform float4 _BackColor;
uniform float4 _BackSpecColor;
uniform float _BackShininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
// (all but the bottom-right element have to be scaled
// with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
// position or direction of light source
uniform float4 _LightColor0;
// color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    float3 normalDir : TEXCOORD1;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
    // multiplication with unity_Scale.w is unnecessary
    // because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.normalDir = normalize(float3(
        mul(float4(-input.normal, 0.0), modelMatrixInverse)));
    // negate input.normal for the back faces
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}
```

```
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);

    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));
    float3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else // point or spot light
    {
        float3 vertexToLightSource =
            float3(_WorldSpaceLightPos0 - input.posWorld);
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
        lightDirection = normalize(vertexToLightSource);
    }

    float3 diffuseReflection =
        attenuation * float3(_LightColor0) * float3(_BackColor)
        * max(0.0, dot(normalDirection, lightDirection));

    float3 specularReflection;
    if (dot(normalDirection, lightDirection) < 0.0)
        // light source on the wrong side?
    {
        specularReflection = float3(0.0, 0.0, 0.0);
        // no specular reflection
    }
    else // light source on the right side
    {
        specularReflection = attenuation * float3(_LightColor0)
            * float3(_BackColor) * pow(max(0.0, dot(
                reflect(-lightDirection, normalDirection),
                viewDirection)), _BackShininess);
    }

    return float4(diffuseReflection
        + specularReflection, 1.0);
}
```

```
// no ambient lighting in this pass
}

ENDCG
}

}

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

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.

Further Reading

If you still want to know more

- about the shader version for single-sided per-pixel lighting, you should read Section “Smooth Specular Highlights”.
- about the shader version for two-sided per-vertex lighting, you should read Section “Two-Sided Surfaces”.

page traffic for 90 days ^[1]

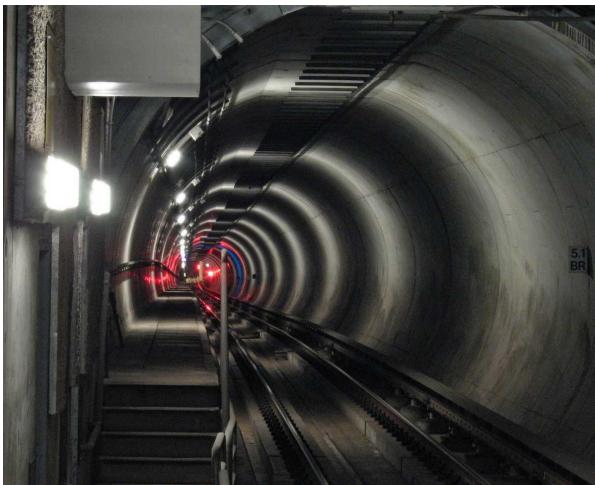
< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Two-Sided_Smooth_Surfaces

3.6 Multiple Lights



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 Section "Smooth Specular Highlights". If you haven't read that tutorial, you should read it first.

Multiple Lights in One Pass

As discussed in Section "Diffuse Reflection", 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 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 float4 unity_LightColor[4];
// array of the colors of the 4 light sources
uniform float4 unity_4LightPosX0;
// x coordinates of the 4 light sources in world space
uniform float4 unity_4LightPosY0;
// y coordinates of the 4 light sources in world space
uniform float4 unity_4LightPosZ0;
// z coordinates of the 4 light sources in world space
uniform float4 unity_4LightAtten0;
// scale factors for attenuation with squared distance
// uniform float4 unity_LightPosition[4] is apparently not
// always correctly set in Unity 3.4
// uniform float4 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 lights using per-vertex lighting. This can be computed with the following for-loop inside the vertex shader:

```

vertexLighting = float3(0.0, 0.0, 0.0);
for (int index = 0; index < 4; index++)
{
    float4 lightPosition = float4(unity_4LightPosX0[index],
        unity_4LightPosY0[index],
        unity_4LightPosZ0[index], 1.0);

    float3 vertexToLightSource =
        float3(lightPosition - position);
    float3 lightDirection = normalize(vertexToLightSource);
    float squaredDistance =
        dot(vertexToLightSource, vertexToLightSource);
    float attenuation = 1.0 / (1.0 +
        unity_4LightAtten0[index] * squaredDistance);
    float3 diffuseReflection =
        attenuation * float3(unity_LightColor[index])
        * float3(_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”.

Complete Shader Code

In the context of the shader code from Section “Smooth Specular Highlights”, the complete shader code is:

```

Shader "Cg 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
    }
}

```

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

// User-specified properties
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (except _LightColor0)
// are also defined in "UnityCG.cginc",
// i.e. one could #include "UnityCG.cginc"
uniform float3 _WorldSpaceCameraPos;
// camera position in world space
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
uniform float4 _WorldSpaceLightPos0;
// direction to or position of light source
uniform float4 _LightColor0;
// color of light source (from "Lighting.cginc")

// Built-in uniforms for "vertex lights"
uniform float4 unity_LightColor[4];
uniform float4 unity_4LightPosX0;
// x coordinates of the 4 light sources in world space
uniform float4 unity_4LightPosY0;
// y coordinates of the 4 light sources in world space
uniform float4 unity_4LightPosZ0;
// z coordinates of the 4 light sources in world space
uniform float4 unity_4LightAtten0;
// scale factors for attenuation with squared distance

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    float3 normalDir : TEXCOORD1;
    float3 vertexLighting : TEXCOORD2;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;
```

```
float4x4 modelMatrix = _Object2World;
float4x4 modelMatrixInverse = _World2Object;
// unity_Scale.w is unnecessary here

output.posWorld = mul(modelMatrix, input.vertex);
output.normalDir = normalize(float3(
    mul(float4(input.normal, 0.0), modelMatrixInverse)));
output.pos = mul(UNITY_MATRIX_MVP, input.vertex);

// Diffuse reflection by four "vertex lights"
output.vertexLighting = float3(0.0, 0.0, 0.0);
#ifndef VERTEXLIGHT_ON
for (int index = 0; index < 4; index++)
{
    float4 lightPosition = float4(unity_4LightPosX0[index],
        unity_4LightPosY0[index],
        unity_4LightPosZ0[index], 1.0);

    float3 vertexToLightSource =
        float3(lightPosition - output.posWorld);
    float3 lightDirection = normalize(vertexToLightSource);
    float squaredDistance =
        dot(vertexToLightSource, vertexToLightSource);
    float attenuation = 1.0 / (1.0 +
        unity_4LightAtten0[index] * squaredDistance);
    float3 diffuseReflection =
        attenuation * float3(unity_LightColor[index])
        * float3(_Color) * max(0.0,
        dot(output.normalDir, lightDirection));

    output.vertexLighting =
        output.vertexLighting + diffuseReflection;
}
#endif
return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);
    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));
    float3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
```

```
{  
    attenuation = 1.0; // no attenuation  
    lightDirection =  
        normalize(float3(_WorldSpaceLightPos0));  
}  
else // point or spot light  
{  
    float3 vertexToLightSource =  
        float3(_WorldSpaceLightPos0 - input.posWorld);  
    float distance = length(vertexToLightSource);  
    attenuation = 1.0 / distance; // linear attenuation  
    lightDirection = normalize(vertexToLightSource);  
}  
  
float3 ambientLighting =  
    float3(UNITY_LIGHTMODEL_AMBIENT) * float3(_Color);  
  
float3 diffuseReflection =  
    attenuation * float3(_LightColor0) * float3(_Color)  
    * max(0.0, dot(normalDirection, lightDirection));  
  
float3 specularReflection;  
if (dot(normalDirection, lightDirection) < 0.0)  
    // light source on the wrong side?  
{  
    specularReflection = float3(0.0, 0.0, 0.0);  
    // no specular reflection  
}  
else // light source on the right side  
{  
    specularReflection = attenuation * float3(_LightColor0)  
        * float3(_SpecColor) * pow(max(0.0, dot(  
            reflect(-lightDirection, normalDirection),  
            viewDirection)), _Shininess);  
}  
  
return float4(input.vertexLighting + ambientLighting  
    + diffuseReflection + specularReflection, 1.0);  
}  
ENDCG  
}  
  
Pass {  
    Tags { "LightMode" = "ForwardAdd" }  
    // pass for additional light sources  
    Blend One One // additive blending
```

```
CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
// (all but the bottom-right element have to be scaled
// with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
// position or direction of light source
uniform float4 _LightColor0;
// color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    float3 normalDir : TEXCOORD1;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
// multiplication with unity_Scale.w is unnecessary
// because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}
```

```
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);

    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));
    float3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else // point or spot light
    {
        float3 vertexToLightSource =
            float3(_WorldSpaceLightPos0 - input.posWorld);
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
        lightDirection = normalize(vertexToLightSource);
    }

    float3 diffuseReflection =
        attenuation * float3(_LightColor0) * float3(_Color)
        * max(0.0, dot(normalDirection, lightDirection));

    float3 specularReflection;
    if (dot(normalDirection, lightDirection) < 0.0)
        // light source on the wrong side?
    {
        specularReflection = float3(0.0, 0.0, 0.0);
        // no specular reflection
    }
    else // light source on the right side
    {
        specularReflection = attenuation * float3(_LightColor0)
            * float3(_SpecColor) * pow(max(0.0, dot(
                reflect(-lightDirection, normalDirection),
                viewDirection)), _Shininess);
    }

    return float4(diffuseReflection
        + specularReflection, 1.0);
}
```

```
// no ambient lighting in this pass
}

ENDCG
}

}

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

The use of `#pragma multi_compile_fowndbase` 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.

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 Cg to compute the lighting of multiple lights in one pass.

Further Reading

If you still want to know more

- about other parts of the shader code, you should read Section “Smooth Specular Highlights”.
- about Unity's forward rendering path and what is computed in the `ForwardBase` pass, you should read Unity's reference about forward rendering ^[1].

page traffic for 90 days ^[2]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

- [1] <http://unity3d.com/support/documentation/Components/RenderTech-ForwardRendering.html>
- [2] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Multiple_Lights

4 Basic Texturing

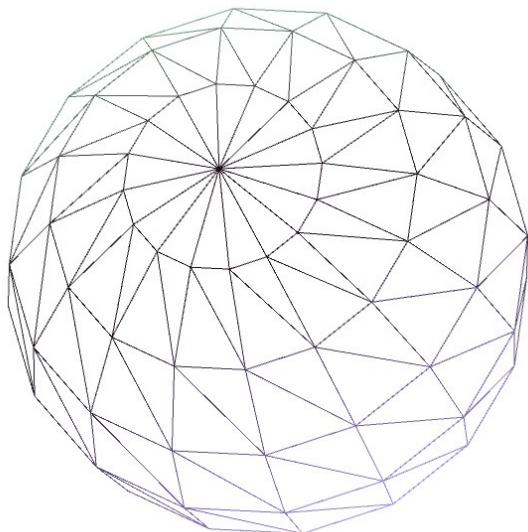
4.1 Textured Spheres



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 Cg 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.



A triangle mesh approximating a sphere.

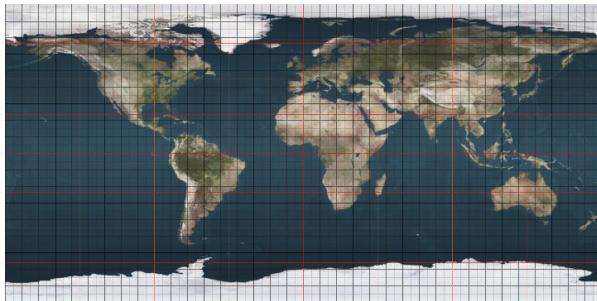
Texture Mapping

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



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

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.

Usually, 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 “`tex2D`” 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.

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, 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 Section “Minimal Shader”. The shader code should be:

```
Shader "Cg 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 {
            CGPROGRAM
                #pragma vertex vert
                #pragma fragment frag

                uniform sampler2D _MainTex;
                    // a uniform variable referring 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).  
  
struct vertexInput {  
    float4 vertex : POSITION;  
    float4 texcoord : TEXCOORD0;  
};  
struct vertexOutput {  
    float4 pos : SV_POSITION;  
    float4 tex : TEXCOORD0;  
};  
  
vertexOutput vert(vertexInput input)  
{  
    vertexOutput output;  
  
    output.tex = input.texcoord;  
    // Unity provides default longitude-latitude-like  
    // texture coordinates at all vertices of a  
    // sphere mesh as the input parameter  
    // "input.texcoord" with semantic "TEXCOORD0".  
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);  
    return output;  
}  
float4 frag(vertexOutput input) : COLOR  
{  
    return tex2D(_MainTex, float2(input.tex));  
    // look up the color of the texture image specified by  
    // the uniform "_MainTex" at the position specified by  
    // "input.tex.x" and "input.tex.y" and return it  
}  
  
ENDCG  
}  
}  
// 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 Section “Diffuse Reflection”).

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!

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 texture coordinates for each vertex in the vertex input parameter `texcoord` with semantic `TEXCOORD0`. These coordinates are similar to longitude and latitude (but range from 0 to 1). This is analogous to the vertex input parameter `vertex` with semantic `POSITION`, which specifies a position in object space, except that `texcoord` specifies texture coordinates in the space of the texture image.

The vertex shader then writes the texture coordinates of each vertex to the vertex output parameter `output.tex`. For each fragment of a triangle (i.e. each covered pixel), the values of this output parameter at the three triangle vertices are interpolated (see the description in Section “Rasterization”) and the interpolated texture coordinates are given to the fragment shader as input parameters. 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 as fragment output parameter, 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.

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 float4 _MainTex_ST;
    // tiling and offset parameters of property "_MainTex"
```

For each texture property, Unity offers such a `float4` 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:

```
return tex2D(_MainTex,
    _MainTex_ST.xy * input.tex.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:

```
Shader "Cg 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 {
        CGPROGRAM

#pragma vertex vert
#pragma fragment frag

uniform sampler2D _MainTex;
uniform float4 _MainTex_ST;
    // tiling and offset parameters of property

struct vertexInput {
    float4 vertex : POSITION;
    float4 texcoord : TEXCOORD0;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 tex : TEXCOORD0;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    output.tex = input.texcoord;
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    return tex2D(_MainTex,
        _MainTex_ST.xy * input.tex.xy + _MainTex_ST.zw);
    // texture coordinates are multiplied with the tiling
    // parameters and the offset parameters are added
}

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

```
// during development:  
// Fallback "Unlit/Texture"  
}
```

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.

Further Reading

If you want to know more

- about the data flow in and out of vertex shaders and fragment shaders (i.e. vertex input and output parameters, etc.), you should read the description in Section “Programmable Graphics Pipeline”.
- about the interpolation of vertex output parameters for the fragment shader, you should read the discussion in Section “Rasterization”.

page traffic for 90 days ^[2]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

- [1] http://commons.wikimedia.org/wiki/File:Earthmap720x360_grid.jpg
- [2] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Textured_Spheres

4.2 Lighting Textured Surfaces



Earthrise as seen from Apollo 8.

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

It combines the shader code of Section “Textured Spheres” and Section “Specular Highlights” 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.

Texturing and Diffuse Per-Vertex Lighting

In Section “Textured Spheres”, 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 Section “Diffuse Reflection”. 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.

Shader Code

In comparison to the per-vertex lighting in Section “Specular Highlights”, the vertex shader here computes two additional output colors: `diffuseColor` and `specularColor`, which use the semantics `TEXCOORD1` and `TEXCOORD2`.

The parameter `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 “SeparateSpecular” [1] 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 Section “Diffuse Reflection”).

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

    CGPROGRAM

        #pragma vertex vert
        #pragma fragment frag

        // User-specified properties
        uniform sampler2D _MainTex;
        uniform float4 _Color;
        uniform float4 _SpecColor;
        uniform float _Shininess;

        // The following built-in uniforms (apart from _LightColor0)
        // are defined in "UnityCG.cginc", which could be #included
        uniform float4 unity_Scale; // w = 1/scale; see _World2Object
        uniform float3 _WorldSpaceCameraPos;
        uniform float4x4 _Object2World; // model matrix
        uniform float4x4 _World2Object; // inverse model matrix
            // (all but the bottom-right element have to be scaled
            // with unity_Scale.w if scaling is important)
        uniform float4 _WorldSpaceLightPos0;
            // position or direction of light source
        uniform float4 _LightColor0;
            // color of light source (from "Lighting.cginc")

        struct vertexInput {
            float4 vertex : POSITION;
            float3 normal : NORMAL;
            float4 texcoord : TEXCOORD0;
        };
        struct vertexOutput {
            float4 pos : SV_POSITION;
            float4 tex : TEXCOORD0;
            float3 diffuseColor : TEXCOORD1;
            float3 specularColor : TEXCOORD2;
        };

        vertexOutput vert(vertexInput input)
        {
            vertexOutput output;

            float4x4 modelMatrix = _Object2World;
            float4x4 modelMatrixInverse = _World2Object;
                // multiplication with unity_Scale.w is unnecessary
```

```
// because we normalize transformed vectors

float3 normalDirection = normalize(float3(
    mul(float4(input.normal, 0.0), modelMatrixInverse)));
float3 viewDirection = normalize(float3(
    float4(_WorldSpaceCameraPos, 1.0)
    - mul(modelMatrix, input.vertex)));
float3 lightDirection;
float attenuation;

if (0.0 == _WorldSpaceLightPos0.w) // directional light?
{
    attenuation = 1.0; // no attenuation
    lightDirection =
        normalize(float3(_WorldSpaceLightPos0));
}
else // point or spot light
{
    float3 vertexToLightSource = float3(_WorldSpaceLightPos0
        - mul(modelMatrix, input.vertex));
    float distance = length(vertexToLightSource);
    attenuation = 1.0 / distance; // linear attenuation
    lightDirection = normalize(vertexToLightSource);
}

float3 ambientLighting =
    float3(UNITY_LIGHTMODEL_AMBIENT) * float3(_Color);

float3 diffuseReflection =
    attenuation * float3(_LightColor0) * float3(_Color)
    * max(0.0, dot(normalDirection, lightDirection));

float3 specularReflection;
if (dot(normalDirection, lightDirection) < 0.0)
    // light source on the wrong side?
{
    specularReflection = float3(0.0, 0.0, 0.0);
    // no specular reflection
}
else // light source on the right side
{
    specularReflection = attenuation * float3(_LightColor0)
        * float3(_SpecColor) * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _Shininess);
}
```

```
        output.diffuseColor = ambientLighting + diffuseReflection;
        output.specularColor = specularReflection;
        output.tex = input.texcoord;
        output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
        return output;
    }

    float4 frag(vertexOutput input) : COLOR
    {
        return float4(input.specularColor +
            input.diffuseColor * tex2D(_MainTex, float2(input.tex)),
            1.0);
    }

    ENDCG
}

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

    CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform sampler2D _MainTex;
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
```

```
float4 vertex : POSITION;
float3 normal : NORMAL;
float4 texcoord : TEXCOORD0;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 tex : TEXCOORD0;
    float3 diffuseColor : TEXCOORD1;
    float3 specularColor : TEXCOORD2;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
    // multiplication with unity_Scale.w is unnecessary
    // because we normalize transformed vectors

    float3 normalDirection = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    float3 viewDirection = normalize(float3(
        float4(_WorldSpaceCameraPos, 1.0)
        - mul(modelMatrix, input.vertex)));
    float3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else // point or spot light
    {
        float3 vertexToLightSource = float3(_WorldSpaceLightPos0
            - mul(modelMatrix, input.vertex));
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
        lightDirection = normalize(vertexToLightSource);
    }

    float3 diffuseReflection =
        attenuation * float3(_LightColor0) * float3(_Color)
        * max(0.0, dot(normalDirection, lightDirection));
}
```

```
float3 specularReflection;
if (dot(normalDirection, lightDirection) < 0.0)
    // light source on the wrong side?
{
    specularReflection = float3(0.0, 0.0, 0.0);
    // no specular reflection
}
else // light source on the right side
{
    specularReflection = attenuation * float3(_LightColor0)
        * float3(_SpecColor) * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _Shininess);
}

output.diffuseColor = diffuseReflection; // no ambient
output.specularColor = specularReflection;
output.tex = input.texcoord;
output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
return output;
}

float4 frag(vertexOutput input) : COLOR
{
    return float4(input.specularColor +
        input.diffuseColor * tex2D(_MainTex, float2(input.tex)),
        1.0);
}

ENDCG
}
// 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 Section “Textured Spheres”.

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.

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 Section “Diffuse Reflection”.
- about per-vertex lighting or the rest of the Phong reflection model, i.e. the ambient and the specular term, you should read Section “Specular Highlights”.
- about the basics of texturing, you should read Section “Textured Spheres”.

page traffic for 90 days ^[2]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] <http://unity3d.com/support/documentation/Components/SL-Material.html>

[2] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Lighting_Textured_Surfaces

4.3 Glossy Textures



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 Section “Textured Spheres” and Section “Smooth Specular Highlights” 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.

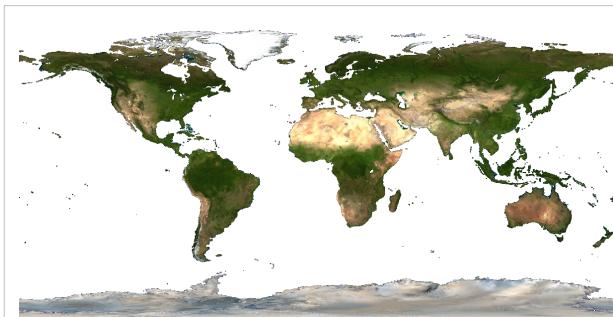
Gloss Mapping

In Section “Lighting Textured Surfaces”, the material constant for the diffuse reflection was determined by the RGB components of 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 Section “Specular Highlights” 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.



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

Shader Code for Per-Pixel Lighting

The shader code is a combination of the per-pixel lighting from Section “Smooth Specular Highlights” and the texturing from Section “Textured Spheres”. Similarly to Section “Lighting Textured Surfaces”, the RGB components of the texture color in `textureColor` is multiplied to the diffuse material color `_Color`.

In the particular texture image to the left, 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 $(1.0 - \text{textureColor.a})$. 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 "Cg per-pixel lighting with texture" {
    Properties {
        _MainTex ("RGB 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
            CGPROGRAM
                #pragma vertex vert
                #pragma fragment frag

                // User-specified properties
                uniform sampler2D _MainTex;
                uniform float4 _Color;
                uniform float4 _SpecColor;
                uniform float _Shininess;

                // The following built-in uniforms (apart from _LightColor0)
                // are defined in "UnityCG.cginc", which could be #included
                uniform float4 unity_Scale; // w = 1/scale; see _World2Object
                uniform float3 _WorldSpaceCameraPos;
                uniform float4x4 _Object2World; // model matrix
                uniform float4x4 _World2Object; // inverse model matrix
            ENDCG
        }
    }
}
```

```
// (all but the bottom-right element have to be scaled
// with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
// position or direction of light source
uniform float4 _LightColor0;
// color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    float3 normalDir : TEXCOORD1;
    float4 tex : TEXCOORD2;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
    // multiplication with unity_Scale.w is unnecessary
    // because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.tex = input.texcoord;
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);

    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));
    float3 lightDirection;
    float attenuation;

    float4 textureColor = tex2D(_MainTex, float2(input.tex));
}
```

```
    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else // point or spot light
    {
        float3 vertexToLightSource =
            float3(_WorldSpaceLightPos0 - input.posWorld);
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
        lightDirection = normalize(vertexToLightSource);
    }

    float3 ambientLighting = float3(textureColor) *
        float3(UNITY_LIGHTMODEL_AMBIENT) * float3(_Color);

    float3 diffuseReflection = float3(textureColor) *
        attenuation * float3(_LightColor0) * float3(_Color)
        * max(0.0, dot(normalDirection, lightDirection));

    float3 specularReflection;
    if (dot(normalDirection, lightDirection) < 0.0)
        // light source on the wrong side?
    {
        specularReflection = float3(0.0, 0.0, 0.0);
        // no specular reflection
    }
    else // light source on the right side
    {
        specularReflection = attenuation * float3(_LightColor0)
            * float3(_SpecColor) * (1.0 - textureColor.a)
            // for usual gloss maps: "... * textureColor.a"
            * pow(max(0.0, dot(
                reflect(-lightDirection, normalDirection),
                viewDirection)), _Shininess);
    }

    return float4(ambientLighting + diffuseReflection
        + specularReflection, 1.0);
}

ENDCG
}

Pass {
```

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

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform sampler2D _MainTex;
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    float3 normalDir : TEXCOORD1;
    float4 tex : TEXCOORD2;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
        // multiplication with unity_Scale.w is unnecessary
```

```
// because we normalize transformed vectors

output.posWorld = mul(modelMatrix, input.vertex);
output.normalDir = normalize(float3(
    mul(float4(input.normal, 0.0), modelMatrixInverse)));
output.tex = input.texcoord;
output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);

    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));
    float3 lightDirection;
    float attenuation;

    float4 textureColor = tex2D(_MainTex, float2(input.tex));

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else // point or spot light
    {
        float3 vertexToLightSource =
            float3(_WorldSpaceLightPos0 - input.posWorld);
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
        lightDirection = normalize(vertexToLightSource);
    }

    float3 diffuseReflection = float3(textureColor) *
        attenuation * float3(_LightColor0) * float3(_Color)
        * max(0.0, dot(normalDirection, lightDirection));

    float3 specularReflection;
    if (dot(normalDirection, lightDirection) < 0.0)
        // light source on the wrong side?
    {
        specularReflection = float3(0.0, 0.0, 0.0);
        // no specular reflection
    }
}
```

```

else // light source on the right side
{
    specularReflection = attenuation * float3(_LightColor0)
        * float3(_SpecColor) * (1.0 - textureColor.a)
        // for usual gloss maps: "... * textureColor.a"
        * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _Shininess);
}

return float4(diffuseReflection
    + specularReflection, 1.0);
// no ambient lighting in this pass
}

ENDCG
}
}

// 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.

Shader Code for Per-Vertex Lighting

As discussed in Section “Smooth Specular Highlights”, 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 Section “Lighting Textured Surfaces”, the fragment shaders of both passes should be replaced with this code:

```

float4 frag(vertexOutput input) : COLOR
{
    float4 textureColor = tex2D(_MainTex, float2(input.tex));
    return float4(input.specularColor * (1.0 - textureColor.a) +
        input.diffuseColor * float3(textureColor), 1.0);
}

```

Note that a usual gloss map would require a multiplication with `textureColor.a` instead of `(1.0 - textureColor.a)`.

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.

Further Reading

If you still want to learn more

- about per-pixel lighting (without texturing), you should read Section “Smooth Specular Highlights”.
- about texturing, you should read Section “Textured Spheres”.
- about per-vertex lighting with texturing, you should read Section “Lighting Textured Surfaces”.

page traffic for 90 days ^[1]

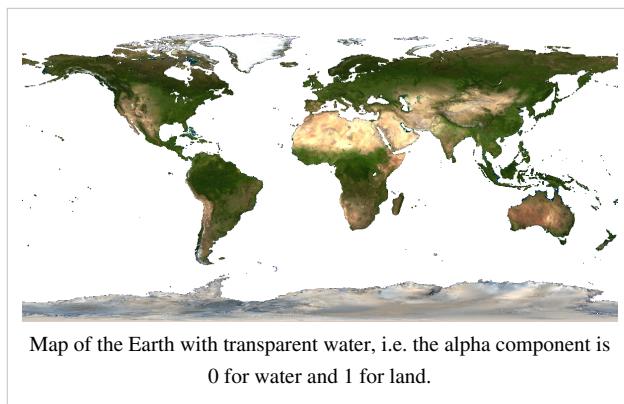
< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Glossy_Textures

4.4 Transparent Textures



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 Section “Textured Spheres” with concepts that were introduced in Section “Cutaways” and Section “Transparency”.

If you haven't read these tutorials, this would be a very good opportunity to read them.

Discarding Transparent Fragments

Let's start with discarding fragments as explained in Section “Cutaways”. Follow the steps described in Section “Textured Spheres” and assign the image to the left to the material of a sphere with the following shader:

```
Shader "Cg 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

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

uniform sampler2D _MainTex;
uniform float _Cutoff;

struct vertexInput {
    float4 vertex : POSITION;
    float4 texcoord : TEXCOORD0;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 tex : TEXCOORD0;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    output.tex = input.texcoord;
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float4 textureColor = tex2D(_MainTex, float2(input.tex));
    if (textureColor.a < _Cutoff)
        // alpha value less than user-specified threshold?
    {
        discard; // yes: discard this fragment
    }
    return textureColor;
}

ENDCG
}

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

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.

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 Cg. Here is the code, which results in more or less the same result as the shader above:

```
Shader "Cg texturing with alpha test" {
    Properties {
        _MainTex ("RGB 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
        }
    }
}

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

uniform sampler2D _MainTex;
uniform float _Cutoff;

struct vertexInput {
    float4 vertex : POSITION;
    float4 texcoord : TEXCOORD0;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 tex : TEXCOORD0;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    output.tex = input.texcoord;
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
```

```
        return tex2D(_MainTex, float2(input.tex));
    }

    ENDCG
}

// 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^[1].

Note that the alpha test and the `discard` instruction are rather slow on some platforms, in particular on mobile devices. Thus, blending is often a more efficient alternative.

Blending

The Section “Transparency” described how to render semitransparent objects with alpha blending. Combining this with an RGBA texture results in this code:

```
Shader "Cg texturing with alpha blending" {
    Properties {
        _MainTex ("RGB 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

        CGPROGRAM

        #pragma vertex vert
        #pragma fragment frag

        uniform sampler2D _MainTex;
        uniform float _Cutoff;

        struct vertexInput {
            float4 vertex : POSITION;
            float4 texcoord : TEXCOORD0;
        };
    }
}
```

```
struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 tex : TEXCOORD0;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    output.tex = input.texcoord;
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    return tex2D(_MainTex, float2(input.tex));
}

ENDCG
}

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
}

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

uniform sampler2D _MainTex;
uniform float _Cutoff;

struct vertexInput {
    float4 vertex : POSITION;
    float4 texcoord : TEXCOORD0;
};
struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 tex : TEXCOORD0;
};

vertexOutput vert(vertexInput input)
```

```

    {
        vertexOutput output;

        output.tex = input.texcoord;
        output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
        return output;
    }

    float4 frag(vertexOutput input) : COLOR
    {
        return tex2D(_MainTex, float2(input.tex));
    }

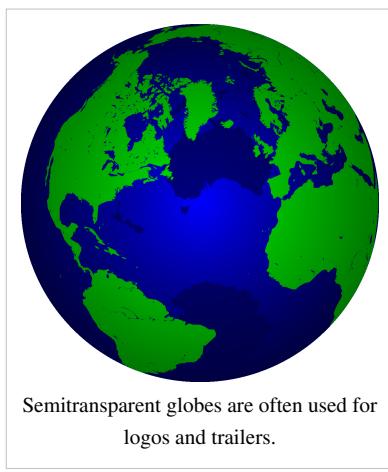
    ENDCG
}
}

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

```

Note that 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



Blending with Customized Colors

We should not end this tutorial without a somewhat more practical application of the presented techniques. To the left 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 "Cg 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

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

uniform sampler2D _MainTex;
uniform float _Cutoff;

struct vertexInput {
    float4 vertex : POSITION;
    float4 texcoord : TEXCOORD0;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 tex : TEXCOORD0;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    output.tex = input.texcoord;
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float4 color = tex2D(_MainTex, float2(input.tex));
    if (color.a > 0.5) // opaque back face?
    {
        color = float4(0.0, 0.0, 0.2, 1.0);
        // opaque dark blue
    }
    else // transparent back face?
    {
        color = float4(0.0, 0.0, 1.0, 0.3);
        // semitransparent dark blue
    }
}
```

```
        return color;
    }

    ENDCG
}

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

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

uniform sampler2D _MainTex;
uniform float _Cutoff;

struct vertexInput {
    float4 vertex : POSITION;
    float4 texcoord : TEXCOORD0;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 tex : TEXCOORD0;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    output.tex = input.texcoord;
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float4 color = tex2D(_MainTex, float2(input.tex));
    if (color.a > 0.5) // opaque front face?
    {
        color = float4(0.0, 1.0, 0.0, 1.0);
        // opaque green
    }
}
```

```

        else // transparent front face
    {
        color = float4(0.0, 0.0, 1.0, 0.3);
        // semitransparent dark blue
    }
    return color;
}

ENDCG
}
}

// 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:

```
color = float4(0.5 * color.r, 2.0 * color.g, 0.5 * color.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 halving the difference of the green component to the maximum intensity 1. This difference is $1.0 - \text{color.g}$; half of it is $0.5 * (1.0 - \text{color.g})$ and the value corresponding to this reduced distance to the maximum intensity is: $1.0 - 0.5 * (1.0 - \text{color.g})$. Thus, in order to avoid oversaturation of green, we could use (instead of the opaque green color):

```
color = float4(0.5 * color.r, 1.0 - 0.5 * (1.0 - color.g), 0.5 * color.b,
1.0);
```

In practice, one has to try various possibilities 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.

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.

Further Reading

If you still want to know more

- about texturing, you should read Section “Textured Spheres”.
- about discarding fragments, you should read Section “Cutaways”.
- about the alpha test, you should read Unity’s ShaderLab documentation: Alpha testing ^[1].
- about blending, you should read Section “Transparency”.

page traffic for 90 days ^[2]

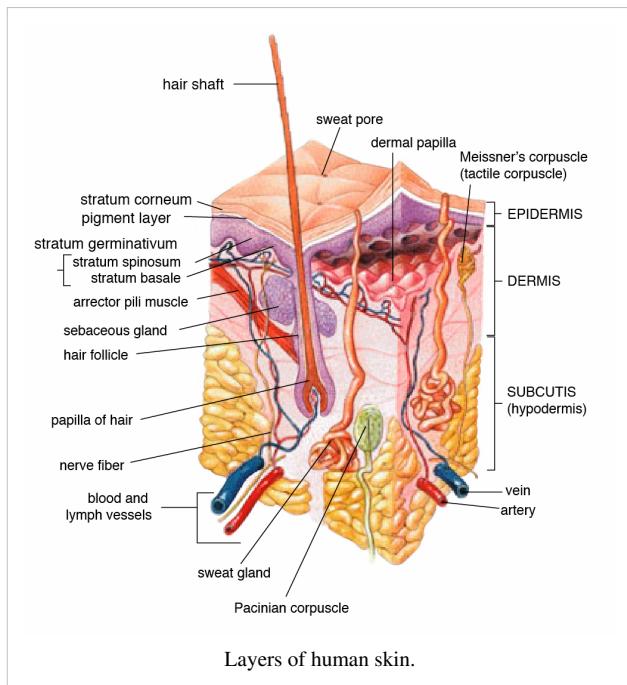
< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

- [1] <http://unity3d.com/support/documentation/Components/SL-AlphaTest.html>
[2] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Transparent_Textures

4.5 Layers of Textures



This tutorial introduces **multitexturing**, i.e. the use of multiple texture images in a shader.

It extends the shader code of Section “Textured Spheres” to multiple textures and shows a way of combining them. If you haven’t read that tutorial, this would be a very good opportunity to read it.

Layers of Surfaces

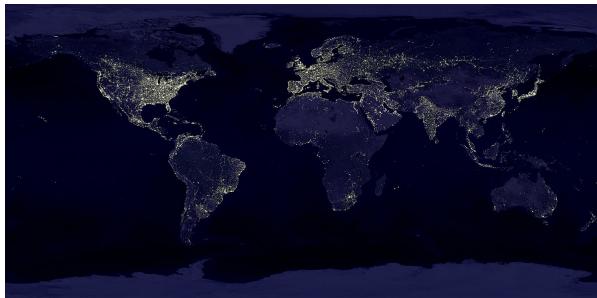
Many real surfaces (e.g. the human skin illustrated in the image to the left) consist of several layers of different colors, transparencies, reflectivities, etc. If the topmost layer is opaque and doesn’t transmit any light, this doesn’t really matter for rendering the surface. However, in many cases 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

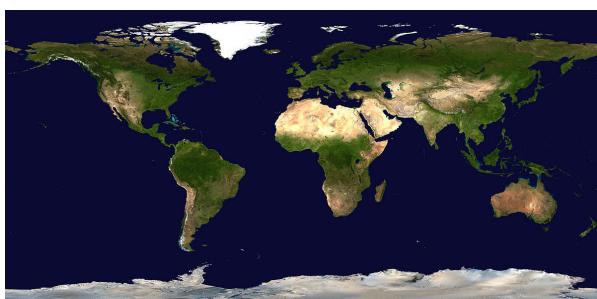
Phong reflection model (see Section “Specular Highlights”) often corresponds to a transparent layer that reflects light: sweat on human skin, wax on fruits, transparent plastics with embedded pigment particles, etc. On the other hand, the diffuse reflection corresponds to the layer(s) below the topmost 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 ^[1]) and the “Human Head” demo by Nvidia (see Chapter 14 of the book “GPU Gems 3”, which is also available online ^[2]).

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.



Map of the unlit Earth.



Map of the sunlit Earth.

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 to the left. 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 Section “Textured Spheres” to two texture images and uses the computation described in Section “Diffuse Reflection” for a single, directional light source:

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

According to this equation, the level of diffuse lighting `levelOfLighting` is $\max(0, \mathbf{N} \cdot \mathbf{L})$. 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 $1.0 - \text{levelOfLighting}$ before adding them to determine the fragment's color. Alternatively, the built-in Cg function `lerp` can be used ($\text{lerp}(a, b, w) = b * w + a * (1.0 - w)$), which is likely to be more efficient. Thus, the fragment shader could be:

```
float4 frag(vertexOutput input) : COLOR
{
    float4 nighttimeColor =
        tex2D(_MainTex, float2(input.tex));
    float4 daytimeColor =
        tex2D(_DecalTex, float2(input.tex));
    return lerp(nighttimeColor, daytimeColor,
                input.levelOfLighting);
    // = daytimeColor * levelOfLighting
    // + nighttimeColor * (1.0 - levelOfLighting)
}
```

Note that this blending is very similar to the alpha blending that was discussed in Section “Transparency” 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 Section “Transparent Textures”), 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.

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 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 "Cg 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
            CGPROGRAM
                #pragma vertex vert
                #pragma fragment frag

                uniform sampler2D _MainTex;
                uniform sampler2D _DecalTex;
                uniform float4 _Color;

                // The following built-in uniforms (apart from _LightColor0)
                // are defined in "UnityCG.cginc", which could be #included
                uniform float4 unity_Scale; // w = 1/scale; see _World2Object
                uniform float3 _WorldSpaceCameraPos;
                uniform float4x4 _Object2World; // model matrix
                uniform float4x4 _World2Object; // inverse model matrix
                    // (all but the bottom-right element have to be scaled
                    // with unity_Scale.w if scaling is important)
                uniform float4 _WorldSpaceLightPos0;
                    // position or direction of light source
                uniform float4 _LightColor0;
                    // color of light source (from "Lighting.cginc")

                struct vertexInput {
                    float4 vertex : POSITION;
                    float3 normal : NORMAL;
                    float4 texcoord : TEXCOORD0;
                };
                struct vertexOutput {
                    float4 pos : SV_POSITION;
```

```
        float4 tex : TEXCOORD0;
        float levelOfLighting : TEXCOORD1;
        // level of diffuse lighting computed in vertex shader
    };

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
    // multiplication with unity_Scale.w is unnecessary
    // because we normalize transformed vectors

    float3 normalDirection = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    float3 lightDirection = normalize(
        float3(_WorldSpaceLightPos0));

    output.levelOfLighting =
        max(0.0, dot(normalDirection, lightDirection));
    output.tex = input.texcoord;
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float4 nighttimeColor =
        tex2D(_MainTex, float2(input.tex));
    float4 daytimeColor =
        tex2D(_DecalTex, float2(input.tex));
    return lerp(nighttimeColor, daytimeColor,
        input.levelOfLighting);
    // = daytimeColor * levelOfLighting
    // + nighttimeColor * (1.0 - levelOfLighting)
}

ENDCG
}
}

// 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.

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.

Further Reading

If you still want to know more

- about basic texturing, you should read Section “Textured Spheres”.
- about diffuse reflection, you should read Section “Diffuse Reflection”.
- about alpha textures, you should read Section “Transparent Textures”.
- 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 ^[1], 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 ^[2].

page traffic for 90 days ^[3]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

- [1] http://http.developer.nvidia.com/GPUGems/gpugems_ch03.html
- [2] http://http.developer.nvidia.com/GPUGems3/gpugems3_ch14.html
- [3] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Layers_of_Textures

5 Textures in 3D

5.1 Lighting of Bumpy Surfaces



"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 Section "Smooth Specular Highlights" and Section "Textured Spheres".

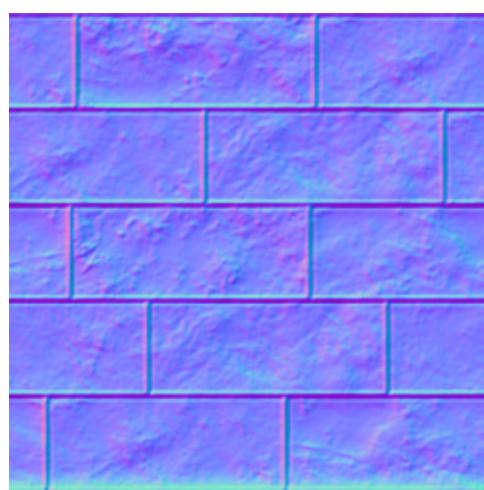
Perceiving Shapes Based on Lighting

The painting by Caravaggio depicted to the left 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.).

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.



A typical example for the appearance of an encoded normal map.

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 **Create from Grayscale**. 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 to the left can be imported (don't forget to uncheck the **Create from Grayscale** 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:

$$\sqrt{n_x^2 + n_y^2 + n_z^2} = 1 \quad \Rightarrow \quad n_z = \pm\sqrt{1 - n_x^2 - n_y^2}$$

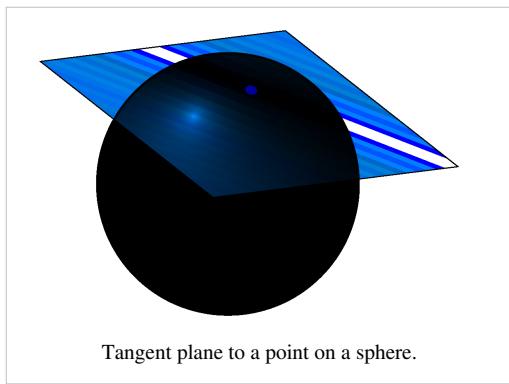
Only the “+” solution is necessary if we choose the z axis along the axis of the smooth normal vector (interpolated from the normal vectors that were set in the vertex shader) since we aren't able to render surfaces with an inwards pointing normal vector anyways. The code snippet from the fragment shader could look like this:

```
float4 encodedNormal = tex2D(_BumpMap,
    _BumpMap_ST.xy * input.tex.xy + _BumpMap_ST.zw);
float3 localCoords =
    float3(2.0 * encodedNormal.ag - float2(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:

```
float4 encodedNormal = tex2D(_BumpMap,
    _BumpMap_ST.xy * input.tex.xy + _BumpMap_ST.zw);
float3 localCoords = 2.0 * encodedNormal.rgb - float3(1.0);
```

However, the rest of this tutorial (and also Section “Projection of Bumpy Surfaces”) will cover only desktop platforms.



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 \mathbf{N} in world space and the $x - y$ plane is a tangent plane to the surface as illustrated in the image to the left. Specifically, the x axis is specified by the tangent parameter \mathbf{T} that Unity provides to vertices (see the discussion of vertex input parameters in Section “Debugging of Shaders”). 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 \mathbf{B} refers to the traditional name “binormal” for this vector.)

Note that the normal vector \mathbf{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 Section “Applying Matrix Transformations”) while the tangent vector \mathbf{T} specifies a direction between points on a surface and is therefore transformed with the model matrix. The binormal vector \mathbf{B} represents a third class of vectors which are transformed differently. (If you really want to know: the skew-symmetric matrix \mathbf{B} corresponding to “ $\mathbf{B}\mathbf{x}$ ” is transformed like a quadratic form.) Thus, the best choice is to first transform \mathbf{N} and \mathbf{T} to world space, and then to compute \mathbf{B} in world space using the cross product of the transformed vectors.

These computations are performed by the vertex shader, for example this way:

```

struct vertexInput {
    float4 vertex : POSITION;
    float4 texcoord : TEXCOORD0;
    float3 normal : NORMAL;
    float4 tangent : TANGENT;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    // position of the vertex (and fragment) in world space
    float4 tex : TEXCOORD1;
    float3 tangentWorld : TEXCOORD2;
    float3 normalWorld : TEXCOORD3;
    float3 binormalWorld : TEXCOORD4;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
    // unity_Scale.w is unnecessary

    output.tangentWorld = normalize(float3(
        mul(modelMatrix, float4(float3(input.tangent), 0.0))));
    output.normalWorld = normalize(

```

```

        mul(float4(input.normal, 0.0), modelMatrixInverse));
output.binormalWorld = normalize(
    cross(output.normalWorld, output.tangentWorld)
    * input.tangent.w); // tangent.w is specific to Unity

output.posWorld = mul(modelMatrix, input.vertex);
output.tex = input.texcoord;
output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
return output;
}

```

The factor `input.tangent.w` in the computation of `binormalWorld` is specific to Unity, i.e. Unity provides tangent vectors and normal maps such that we have to do this multiplication.

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:

$$M_{\text{surface} \rightarrow \text{world}} = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}$$

In Cg, it is actually easier to construct the transposed matrix since matrices are constructed row by row

$$M_{\text{surface} \rightarrow \text{world}}^T = \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{bmatrix}$$

The construction is done in the fragment shader, e.g.:

```

float3x3 local2WorldTranspose = float3x3(input.tangentWorld,
    input.binormalWorld, input.normalWorld);

```

We want to transform **n** with the transpose of `local2WorldTranspose` (i.e. the not transposed original matrix); therefore, we multiply **n** from the left with the matrix. For example, with this line:

```

float3 normalDirection =
    normalize(mul(localCoords, local2WorldTranspose));

```

With the new normal vector in world space, we can compute the lighting as in Section “Smooth Specular Highlights”.

Complete Shader Code

This shader code simply integrates all the snippets and uses our standard two-pass approach for pixel lights.

```

Shader "Cg 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

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform sampler2D _BumpMap;
uniform float4 _BumpMap_ST;
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float4 texcoord : TEXCOORD0;
    float3 normal : NORMAL;
    float4 tangent : TANGENT;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
        // position of the vertex (and fragment) in world space
    float4 tex : TEXCOORD1;
    float3 tangentWorld : TEXCOORD2;
    float3 normalWorld : TEXCOORD3;
    float3 binormalWorld : TEXCOORD4;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;
```

```
float4x4 modelMatrix = _Object2World;
float4x4 modelMatrixInverse = _World2Object;
// unity_Scale.w is unnecessary

output.tangentWorld = normalize(float3(
    mul(modelMatrix, float4(float3(input.tangent), 0.0)))); 
output.normalWorld = normalize(
    mul(float4(input.normal, 0.0), modelMatrixInverse));
output.binormalWorld = normalize(
    cross(output.normalWorld, output.tangentWorld)
    * input.tangent.w); // tangent.w is specific to Unity

output.posWorld = mul(modelMatrix, input.vertex);
output.tex = input.texcoord;
output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
return output;
}

float4 frag(vertexOutput input) : COLOR
{
    // in principle we have to normalize tangentWorld,
    // binormalWorld, and normalWorld again; however, the
    // potential problems are small since we use this
    // matrix only to compute "normalDirection",
    // which we normalize anyways

    float4 encodedNormal = tex2D(_BumpMap,
        _BumpMap_ST.xy * input.tex.xy + _BumpMap_ST.zw);
    float3 localCoords =
        float3(2.0 * encodedNormal.ag - float2(1.0), 0.0);
    localCoords.z = sqrt(1.0 - dot(localCoords, localCoords));
    // approximation without sqrt: localCoords.z =
    // 1.0 - 0.5 * dot(localCoords, localCoords);

    float3x3 local2WorldTranspose = float3x3(
        input.tangentWorld,
        input.binormalWorld,
        input.normalWorld);
    float3 normalDirection =
        normalize(mul(localCoords, local2WorldTranspose));

    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));
    float3 lightDirection;
    float attenuation;
```

```
    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else // point or spot light
    {
        float3 vertexToLightSource =
            float3(_WorldSpaceLightPos0 - input.posWorld);
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
        lightDirection = normalize(vertexToLightSource);
    }

    float3 ambientLighting =
        float3(UNITY_LIGHTMODEL_AMBIENT) * float3(_Color);

    float3 diffuseReflection =
        attenuation * float3(_LightColor0) * float3(_Color)
        * max(0.0, dot(normalDirection, lightDirection));

    float3 specularReflection;
    if (dot(normalDirection, lightDirection) < 0.0)
        // light source on the wrong side?
    {
        specularReflection = float3(0.0, 0.0, 0.0);
        // no specular reflection
    }
    else // light source on the right side
    {
        specularReflection = attenuation * float3(_LightColor0)
            * float3(_SpecColor) * pow(max(0.0, dot(
                reflect(-lightDirection, normalDirection),
                viewDirection)), _Shininess);
    }

    return float4(ambientLighting + diffuseReflection
        + specularReflection, 1.0);
}

ENDCG
}

Pass {
    Tags { "LightMode" = "ForwardAdd" }
    // pass for additional light sources
```

```
Blend One One // additive blending

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform sampler2D _BumpMap;
uniform float4 _BumpMap_ST;
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float4 texcoord : TEXCOORD0;
    float3 normal : NORMAL;
    float4 tangent : TANGENT;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
        // position of the vertex (and fragment) in world space
    float4 tex : TEXCOORD1;
    float3 tangentWorld : TEXCOORD2;
    float3 normalWorld : TEXCOORD3;
    float3 binormalWorld : TEXCOORD4;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;
```

```
float4x4 modelMatrix = _Object2World;
float4x4 modelMatrixInverse = _World2Object;
// unity_Scale.w is unnecessary

output.tangentWorld = normalize(float3(
    mul(modelMatrix, float4(float3(input.tangent), 0.0)))); 
output.normalWorld = normalize(
    mul(float4(input.normal, 0.0), modelMatrixInverse));
output.binormalWorld = normalize(
    cross(output.normalWorld, output.tangentWorld)
    * input.tangent.w); // tangent.w is specific to Unity

output.posWorld = mul(modelMatrix, input.vertex);
output.tex = input.texcoord;
output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
return output;
}

float4 frag(vertexOutput input) : COLOR
{
    // in principle we have to normalize tangentWorld,
    // binormalWorld, and normalWorld again; however, the
    // potential problems are small since we use this
    // matrix only to compute "normalDirection",
    // which we normalize anyways

    float4 encodedNormal = tex2D(_BumpMap,
        _BumpMap_ST.xy * input.tex.xy + _BumpMap_ST.zw);
    float3 localCoords =
        float3(2.0 * encodedNormal.ag - float2(1.0), 0.0);
    localCoords.z = sqrt(1.0 - dot(localCoords, localCoords));
    // approximation without sqrt: localCoords.z =
    // 1.0 - 0.5 * dot(localCoords, localCoords);

    float3x3 local2WorldTranspose = float3x3(
        input.tangentWorld,
        input.binormalWorld,
        input.normalWorld);
    float3 normalDirection =
        normalize(mul(localCoords, local2WorldTranspose));

    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));
    float3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
```

```

    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else // point or spot light
    {
        float3 vertexToLightSource =
            float3(_WorldSpaceLightPos0 - input.posWorld);
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
        lightDirection = normalize(vertexToLightSource);
    }

    float3 diffuseReflection =
        attenuation * float3(_LightColor0) * float3(_Color)
        * max(0.0, dot(normalDirection, lightDirection));

    float3 specularReflection;
    if (dot(normalDirection, lightDirection) < 0.0)
        // light source on the wrong side?
    {
        specularReflection = float3(0.0, 0.0, 0.0);
        // no specular reflection
    }
    else // light source on the right side
    {
        specularReflection = attenuation * float3(_LightColor0)
            * float3(_SpecColor) * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _Shininess);
    }

    return float4(diffuseReflection + specularReflection, 1.0);
}

ENDCG
}

}
}

```

Note that we have used the tiling and offset uniform `_BumpMap_ST` as explained in the Section “Textured Spheres” since this option is often particularly useful for bump maps.

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.

Further Reading

If you still want to know more

- about texture mapping (including tiling and offseting), you should read Section "Textured Spheres".
- about per-pixel lighting with the Phong reflection model, you should read Section "Smooth Specular Highlights".
- about transforming normal vectors, you should read Section "Applying Matrix Transformations".
- 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^[1].

page traffic for 90 days^[2]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] <http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.537&rank=2>

[2] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Lighting_of_Bumpy_Surfaces

5.2 Projection of Bumpy Surfaces



A dry-stone wall in England. Note how some stones stick out of the wall.

This tutorial covers (single-step) **parallax mapping**.

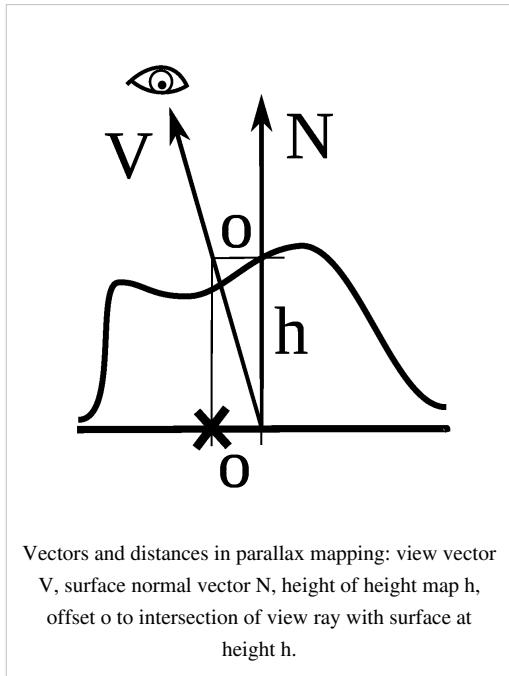
It extends and is based on Section "Lighting of Bumpy Surfaces".

Improving Normal Mapping

The normal mapping technique presented in Section "Lighting of Bumpy Surfaces" 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, if one looks onto a surface from some other angle (as in the image to the left), 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.



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 to the left shows the view vector \mathbf{V} in the direction to the viewer and the surface normal vector \mathbf{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 \mathbf{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 \mathbf{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 \mathbf{V} if the surface was displaced by the height map.

of \mathbf{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 \mathbf{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 \mathbf{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 Section “Lighting of Bumpy Surfaces”. We also require the view direction \mathbf{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 Section “Lighting of Bumpy Surfaces”. To this end we compute a transformation from local surface coordinates to object space with:

$$\mathbf{M}_{\text{surface} \rightarrow \text{object}} = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}$$

where \mathbf{T} , \mathbf{B} and \mathbf{N} are given in object coordinates. (In Section “Lighting of Bumpy Surfaces” we had a similar matrix but with vectors in world coordinates.)

We compute the view direction \mathbf{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 $\mathbf{M}_{\text{object} \rightarrow \text{surface}}$ which can be computed as:

$$\mathbf{M}_{\text{object} \rightarrow \text{surface}} = \mathbf{M}_{\text{surface} \rightarrow \text{object}}^{-1} = \mathbf{M}_{\text{surface} \rightarrow \text{object}}^T$$

This is possible because \mathbf{T} , \mathbf{B} and \mathbf{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 \mathbf{V} from object space to the local surface space, we have to multiply it with the transposed matrix

$(M_{\text{surface} \rightarrow \text{object}})^T$. This is actually good, because in Cg it is easier to construct the transposed matrix as \mathbf{T} , \mathbf{B} and \mathbf{N} are the row vectors of the transposed matrix.

Once we have the \mathbf{V} in the local surface coordinate system with the z axis in the direction of the normal vector \mathbf{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):

$$\frac{o_x}{h} = \frac{V_x}{V_z} \quad \text{and} \quad \frac{o_y}{h} = \frac{V_y}{V_z}.$$

Thus:

$$o_x = h \frac{V_x}{V_z} \quad \text{and} \quad o_y = h \frac{V_y}{V_z}.$$

Note that it is not necessary to normalize \mathbf{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 \mathbf{V} scales V_x and V_y appropriately to have o_x and o_y in texture coordinate space without further computations.

Implementation

The implementation shares most of the code with Section "Lighting of Bumpy Surfaces". 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:

```
float3 binormal = cross(input.normal, float3(input.tangent))
    * input.tangent.w;
```

We have to add an output parameter for the view vector \mathbf{V} in the local surface coordinate system (with the scaling of axes to take the mapping to texture space into account). This parameter is called `viewDirInScaledSurfaceCoords`. It is computed by transforming the view vector in object coordinates (`viewDirInObjectCoords`) with the matrix $M_{\text{surface} \rightarrow \text{object}}^T$ (`localSurface2ScaledObjectT`) as explained above:

```
float3 viewDirInObjectCoords = float3(mul(
    modelMatrixInverse, float4(_WorldSpaceCameraPos, 1.0))
    - input.vertex);
float3x3 localSurface2ScaledObjectT =
    float3x3(float3(input.tangent), binormal, input.normal);
// vectors are orthogonal
output.viewDirInScaledSurfaceCoords =
    mul(localSurface2ScaledObjectT, viewDirInObjectCoords);
// 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 Section "Lighting of Bumpy Surfaces" except that the view direction in world coordinates is computed in the vertex shader instead of the fragment shader, which is necessary to keep the number of arithmetic operations in the fragment shader small enough for some GPUs.

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 + tex2D(_ParallaxMap, _ParallaxMap_ST.xy
    * input.tex.xy + _ParallaxMap_ST.zw).x);
```

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:

```
float2 texCoordOffsets =
    clamp(height * input.viewDirInScaledSurfaceCoords.xy
    / input.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 `float2(input.tex)` (or equivalently `input.tex.xy`) by `(input.tex.xy + texCoordOffsets)`, e.g.:

```
float4 encodedNormal = tex2D(_BumpMap,
    _BumpMap_ST.xy * (input.tex.xy + texCoordOffsets)
    + _BumpMap_ST.zw);
```

The rest of the fragment shader code is just as it was for Section “Lighting of Bumpy Surfaces”.

Complete Shader Code

As discussed in the previous section, most of this code is taken from Section “Lighting of Bumpy Surfaces”. 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 "Cg 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
```

CGPROGRAM

```
#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform sampler2D _BumpMap;
uniform float4 _BumpMap_ST;
uniform sampler2D _ParallaxMap;
uniform float4 _ParallaxMap_ST;
uniform float _Parallax;
uniform float _MaxTexCoordOffset;
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float4 texcoord : TEXCOORD0;
    float3 normal : NORMAL;
    float4 tangent : TANGENT;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
        // position of the vertex (and fragment) in world space
    float4 tex : TEXCOORD1;
    float3 tangentWorld : TEXCOORD2;
    float3 normalWorld : TEXCOORD3;
    float3 binormalWorld : TEXCOORD4;
    float3 viewDirWorld : TEXCOORD5;
    float3 viewDirInScaledSurfaceCoords : TEXCOORD6;
};
```

```
vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
    // unity_Scale.w is unnecessary

    output.tangentWorld = normalize(float3(
        mul(modelMatrix, float4(float3(input.tangent), 0.0))));
    output.normalWorld = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.binormalWorld = normalize(
        cross(output.normalWorld, output.tangentWorld)
        * input.tangent.w); // tangent.w is specific to Unity

    float3 binormal = cross(input.normal, float3(input.tangent))
        * input.tangent.w;
    // appropriately scaled tangent and binormal
    // to map distances from object space to texture space

    float3 viewDirInObjectCoords = float3(mul(
        modelMatrixInverse, float4(_WorldSpaceCameraPos, 1.0))
        - input.vertex);
    float3x3 localSurface2ScaledObjectT =
        float3x3(float3(input.tangent), binormal, input.normal);
    // vectors are orthogonal
    output.viewDirInScaledSurfaceCoords =
        mul(localSurface2ScaledObjectT, viewDirInObjectCoords);
    // we multiply with the transpose to multiply with
    // the "inverse" (apart from the scaling)

    output.posWorld = mul(modelMatrix, input.vertex);
    output.viewDirWorld = normalize(
        _WorldSpaceCameraPos - float3(output.posWorld));
    output.tex = input.texcoord;
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    // 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 + tex2D(_ParallaxMap, _ParallaxMap_ST.xy
    * input.tex.xy + _ParallaxMap_ST.zw).x);

float2 texCoordOffsets =
    clamp(height * input.viewDirInScaledSurfaceCoords.xy
    / input.viewDirInScaledSurfaceCoords.z,
    -_MaxTexCoordOffset, +_MaxTexCoordOffset);

// normal mapping: lookup and decode normal from bump map

// in principle we have to normalize tangentWorld,
// binormalWorld, and normalWorld again; however, the
// potential problems are small since we use this
// matrix only to compute "normalDirection",
// which we normalize anyways

float4 encodedNormal = tex2D(_BumpMap,
    _BumpMap_ST.xy * (input.tex.xy + texCoordOffsets)
    + _BumpMap_ST.zw);
float3 localCoords =
    float3(2.0 * encodedNormal.ag - float2(1.0), 0.0);
localCoords.z = sqrt(1.0 - dot(localCoords, localCoords));
// approximation without sqrt: localCoords.z =
// 1.0 - 0.5 * dot(localCoords, localCoords);

float3x3 local2WorldTranspose = float3x3(
    input.tangentWorld,
    input.binormalWorld,
    input.normalWorld);
float3 normalDirection =
    normalize(mul(localCoords, local2WorldTranspose));

// per-pixel lighting using the Phong reflection model
// (with linear attenuation for point and spot lights)

float3 lightDirection;
float attenuation;

if (0.0 == _WorldSpaceLightPos0.w) // directional light?
{
    attenuation = 1.0; // no attenuation
    lightDirection =
        normalize(float3(_WorldSpaceLightPos0));
}
else // point or spot light
```

```
{  
    float3 vertexToLightSource =  
        float3(_WorldSpaceLightPos0 - input.posWorld);  
    float distance = length(vertexToLightSource);  
    attenuation = 1.0 / distance; // linear attenuation  
    lightDirection = normalize(vertexToLightSource);  
}  
  
float3 ambientLighting =  
    float3(UNITY_LIGHTMODEL_AMBIENT) * float3(_Color);  
  
float3 diffuseReflection =  
    attenuation * float3(_LightColor0) * float3(_Color)  
    * max(0.0, dot(normalDirection, lightDirection));  
  
float3 specularReflection;  
if (dot(normalDirection, lightDirection) < 0.0)  
    // light source on the wrong side?  
{  
    specularReflection = float3(0.0, 0.0, 0.0);  
    // no specular reflection  
}  
else // light source on the right side  
{  
    specularReflection = attenuation * float3(_LightColor0)  
        * float3(_SpecColor) * pow(max(0.0, dot(  
            reflect(-lightDirection, normalDirection),  
            input.viewDirWorld)), _Shininess);  
}  
  
return float4(ambientLighting + diffuseReflection  
    + specularReflection, 1.0);  
}  
  
ENDCG  
}  
  
Pass {  
    Tags { "LightMode" = "ForwardAdd" }  
    // pass for additional light sources  
    Blend One One // additive blending  
  
    CGPROGRAM  
  
#pragma vertex vert  
#pragma fragment frag
```

```
// User-specified properties
uniform sampler2D _BumpMap;
uniform float4 _BumpMap_ST;
uniform sampler2D _ParallaxMap;
uniform float4 _ParallaxMap_ST;
uniform float _Parallax;
uniform float _MaxTexCoordOffset;
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float4 texcoord : TEXCOORD0;
    float3 normal : NORMAL;
    float4 tangent : TANGENT;
};
struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
        // position of the vertex (and fragment) in world space
    float4 tex : TEXCOORD1;
    float3 tangentWorld : TEXCOORD2;
    float3 normalWorld : TEXCOORD3;
    float3 binormalWorld : TEXCOORD4;
    float3 viewDirWorld : TEXCOORD5;
    float3 viewDirInScaledSurfaceCoords : TEXCOORD6;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;
```

```
float4x4 modelMatrix = _Object2World;
float4x4 modelMatrixInverse = _World2Object;
// unity_Scale.w is unnecessary

output.tangentWorld = normalize(float3(
    mul(modelMatrix, float4(float3(input.tangent), 0.0)));
output.normalWorld = normalize(float3(
    mul(float4(input.normal, 0.0), modelMatrixInverse)));
output.binormalWorld = normalize(
    cross(output.normalWorld, output.tangentWorld)
    * input.tangent.w); // tangent.w is specific to Unity

float3 binormal = cross(input.normal, float3(input.tangent))
    * input.tangent.w;
// appropriately scaled tangent and binormal
// to map distances from object space to texture space

float3 viewDirInObjectCoords = float3(mul(
    modelMatrixInverse, float4(_WorldSpaceCameraPos, 1.0))
    - input.vertex);
float3x3 localSurface2ScaledObjectT =
    float3x3(float3(input.tangent), binormal, input.normal);
// vectors are orthogonal
output.viewDirInScaledSurfaceCoords =
    mul(localSurface2ScaledObjectT, viewDirInObjectCoords);
// we multiply with the transpose to multiply with
// the "inverse" (apart from the scaling)

output.posWorld = mul(modelMatrix, input.vertex);
output.viewDirWorld = normalize(
    _WorldSpaceCameraPos - float3(output.posWorld));
output.tex = input.texcoord;
output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
return output;
}

float4 frag(vertexOutput input) : COLOR
{
    // 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 + tex2D(_ParallaxMap, _ParallaxMap_ST.xy
        * input.tex.xy + _ParallaxMap_ST.zw).x);
}
```

```
float2 texCoordOffsets =
    clamp(height * input.viewDirInScaledSurfaceCoords.xy
        / input.viewDirInScaledSurfaceCoords.z,
        -_MaxTexCoordOffset, +_MaxTexCoordOffset);

// normal mapping: lookup and decode normal from bump map

// in principle we have to normalize tangentWorld,
// binormalWorld, and normalWorld again; however, the
// potential problems are small since we use this
// matrix only to compute "normalDirection",
// which we normalize anyways

float4 encodedNormal = tex2D(_BumpMap,
    _BumpMap_ST.xy * (input.tex.xy + texCoordOffsets)
    + _BumpMap_ST.zw);
float3 localCoords =
    float3(2.0 * encodedNormal.ag - float2(1.0), 0.0);
localCoords.z = sqrt(1.0 - dot(localCoords, localCoords));
// approximation without sqrt: localCoords.z =
// 1.0 - 0.5 * dot(localCoords, localCoords);

float3x3 local2WorldTranspose = float3x3(
    input.tangentWorld,
    input.binormalWorld,
    input.normalWorld);
float3 normalDirection =
    normalize(mul(localCoords, local2WorldTranspose));

// per-pixel lighting using the Phong reflection model
// (with linear attenuation for point and spot lights)

float3 lightDirection;
float attenuation;

if (0.0 == _WorldSpaceLightPos0.w) // directional light?
{
    attenuation = 1.0; // no attenuation
    lightDirection =
        normalize(float3(_WorldSpaceLightPos0));
}
else // point or spot light
{
    float3 vertexToLightSource =
        float3(_WorldSpaceLightPos0 - input.posWorld);
    float distance = length(vertexToLightSource);
    attenuation = 1.0 / distance; // linear attenuation
```

```

        lightDirection = normalize(vertexToLightSource);
    }

    float3 diffuseReflection =
        attenuation * float3(_LightColor0) * float3(_Color)
        * max(0.0, dot(normalDirection, lightDirection));

    float3 specularReflection;
    if (dot(normalDirection, lightDirection) < 0.0)
        // light source on the wrong side?
    {
        specularReflection = float3(0.0, 0.0, 0.0);
        // no specular reflection
    }
    else // light source on the right side
    {
        specularReflection = attenuation * float3(_LightColor0)
            * float3(_SpecColor) * pow(max(0.0, dot(
                reflect(-lightDirection, normalDirection),
                input.viewDirWorld)), _Shininess);
    }

    return float4(diffuseReflection
        + specularReflection, 1.0);
}

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

```

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, 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.

Further Reading

If you still want to know more

- about details of the shader code, you should read Section “Lighting of Bumpy Surfaces”.
- 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^[1].

page traffic for 90 days^[2]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] <http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.115.1050>

[2] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Projection_of_Bumpy_Surfaces

5.3 Cookies

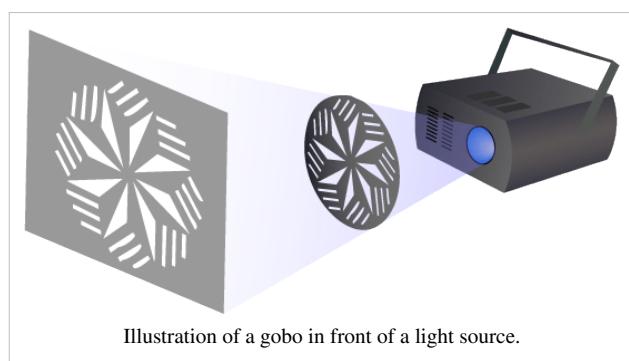


Illustration of a gobo in front of a light source.

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 Section “Smooth Specular Highlights” and Section “Transparent Textures”. If you haven’t read those tutorials yet, you should read them first.

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 “cucoloris”) serve a similar purpose but are placed at a larger distance from the light source as shown in the image to the left.

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 Section “Transparent Textures”) 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.



A cookie in action: similar to a gobo but not as close to the light source.

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 Section "Projectors".) 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 Section "Smooth Specular Highlights"; 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 Section "Vertex Transformations". 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 we would have to set up the matrix similar to the matrices for the viewing transformation and the projection, which are discussed in Section "Vertex Transformations".)

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 float4x4 _LightMatrix0; // transformation
// from world to light space (from Autolight.cginc)
...

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};
struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    // position of the vertex (and fragment) in world space
}
```

```

        float4 posLight : TEXCOORD1;
        // position of the vertex (and fragment) in light space
        float3 normalDir : TEXCOORD2;
        // surface normal vector in world space
    };

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
    // multiplication with unity_Scale.w is unnecessary
    // because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.posLight = mul(_LightMatrix0, output.posWorld);
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

```

Apart from the definitions of the uniform `_LightMatrix0` and the new output parameter `posLight` and the instruction to compute `posLight`, this is the same vertex shader as in Section “Smooth Specular Highlights”.

Cookies for Directional Light Sources

For the cookie of a directional light source, we can just use the *x* and *y* light coordinates in `posLight` 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 = tex2D(_LightTexture0,
        float2(input.posLight)).a;
}
// compute cookieAttenuation for spotlights here

return float4(cookieAttenuation
    * (diffuseReflection + specularReflection), 1.0);

```

Instead of `float2(input.posLight)` we could also use `input.posLight.xy` to get a two-dimensional vector with the *x* and *y* coordinates in light space.

Cookies for Spotlights

For spotlights, the x and y light coordinates in `posLight` 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 Section “Vertex Transformations”. Unity defines the matrix `_LightMatrix0` such that we have to add `0.5` to both coordinates after the division:

```
cookieAttenuation = tex2D(_LightTexture0,
    float2(input.posLight) / input.posLight.w
    + float2(0.5)).a;
```

For some GPUs it might be more efficient to use the built-in function `tex2Dproj`, which takes three texture coordinates in a `float3` 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 `posLight.w`; however, `tex2Dproj` doesn't allow us to add anything after the internal division by the third texture coordinate. The solution is to add `0.5 * input.posLight.w` **before** the division by `posLight.w`, which corresponds to adding `0.5` after the division:

```
float3 textureCoords = float3(float2(input.posLight)
    + float2(0.5 * input.posLight.w), input.posLight.w);
cookieAttenuation =
    tex2Dproj(_LightTexture0, textureCoords).a;
```

Note that the texture lookup for directional lights can also be implemented with `tex2Dproj` by setting `textureCoords` to `float3(float2(input.posLight), 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.

Complete Shader Code

For the complete shader code we use a simplified version of the `ForwardBase` pass of Section “Smooth Specular Highlights” 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 "Cg 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
    }
}
```

CGPROGRAM

```
#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    float3 normalDir : TEXCOORD1;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
        // multiplication with unity_Scale.w is unnecessary
        // because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}
```

```
float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);

    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));
    float3 lightDirection =
        normalize(float3(_WorldSpaceLightPos0));

    float3 ambientLighting =
        float3(UNITY_LIGHTMODEL_AMBIENT) * float3(_Color);

    float3 diffuseReflection =
        float3(_LightColor0) * float3(_Color)
        * max(0.0, dot(normalDirection, lightDirection));

    float3 specularReflection;
    if (dot(normalDirection, lightDirection) < 0.0)
        // light source on the wrong side?
    {
        specularReflection = float3(0.0, 0.0, 0.0);
        // no specular reflection
    }
    else // light source on the right side
    {
        specularReflection = float3(_LightColor0)
            * float3(_SpecColor) * pow(max(0.0, dot(
                reflect(-lightDirection, normalDirection),
                viewDirection)), _Shininess);
    }

    return float4(ambientLighting + diffuseReflection
        + specularReflection, 1.0);
}

ENDCG
}

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

    CGPROGRAM
        #pragma vertex vert
```

```
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

uniform float4x4 _LightMatrix0; // transformation
    // from world to light space (from Autolight.cginc)
uniform sampler2D _LightTexture0;
    // cookie alpha texture map (from Autolight.cginc)

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
        // position of the vertex (and fragment) in world space
    float4 posLight : TEXCOORD1;
        // position of the vertex (and fragment) in light space
    float3 normalDir : TEXCOORD2;
        // surface normal vector in world space
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
        // multiplication with unity_Scale.w is unnecessary
        // because we normalize transformed vectors
```

```
        output.posWorld = mul(modelMatrix, input.vertex);
        output.posLight = mul(_LightMatrix0, output.posWorld);
        output.normalDir = normalize(float3(
            mul(float4(input.normal, 0.0), modelMatrixInverse)));
        output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
        return output;
    }

    float4 frag(vertexOutput input) : COLOR
    {
        float3 normalDirection = normalize(input.normalDir);

        float3 viewDirection = normalize(
            _WorldSpaceCameraPos - float3(input.posWorld));
        float3 lightDirection;
        float attenuation;

        if (0.0 == _WorldSpaceLightPos0.w) // directional light?
        {
            attenuation = 1.0; // no attenuation
            lightDirection =
                normalize(float3(_WorldSpaceLightPos0));
        }
        else // point or spot light
        {
            float3 vertexToLightSource =
                float3(_WorldSpaceLightPos0 - input.posWorld);
            float distance = length(vertexToLightSource);
            attenuation = 1.0 / distance; // linear attenuation
            lightDirection = normalize(vertexToLightSource);
        }

        float3 diffuseReflection =
            attenuation * float3(_LightColor0) * float3(_Color)
            * max(0.0, dot(normalDirection, lightDirection));

        float3 specularReflection;
        if (dot(normalDirection, lightDirection) < 0.0)
            // light source on the wrong side?
        {
            specularReflection = float3(0.0, 0.0, 0.0);
            // no specular reflection
        }
        else // light source on the right side
        {
            specularReflection = attenuation * float3(_LightColor0)
```

```

        * float3(_SpecColor) * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _Shininess);
    }

    float cookieAttenuation = 1.0;
    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        cookieAttenuation = tex2D(_LightTexture0,
            float2(input.posLight)).a;
    }
    else if (1.0 != _LightMatrix0[3][3])
        // spotlight (i.e. not a point light)?
    {
        cookieAttenuation = tex2D(_LightTexture0,
            float2(input.posLight) / input.posLight.w
            + float2(0.5)).a;
    }

    return float4(cookieAttenuation
        * (diffuseReflection + specularReflection), 1.0);
}

ENDCG
}
}

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

```

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 `CGPROGRAM` 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 "Cg 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
    }

    CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag

        // User-specified properties
        uniform float4 _Color;
        uniform float4 _SpecColor;
        uniform float _Shininess;

        // The following built-in uniforms (apart from _LightColor0)
        // are defined in "UnityCG.cginc", which could be #included
        uniform float4 unity_Scale; // w = 1/scale; see _World2Object
        uniform float3 _WorldSpaceCameraPos;
        uniform float4x4 _Object2World; // model matrix
        uniform float4x4 _World2Object; // inverse model matrix
        // (all but the bottom-right element have to be scaled
        // with unity_Scale.w if scaling is important)
        uniform float4 _WorldSpaceLightPos0;
        // position or direction of light source
        uniform float4 _LightColor0;
        // color of light source (from "Lighting.cginc")

        struct vertexInput {
            float4 vertex : POSITION;
            float3 normal : NORMAL;
        };
        struct vertexOutput {
            float4 pos : SV_POSITION;
            float4 posWorld : TEXCOORD0;
            float3 normalDir : TEXCOORD1;
        };

        vertexOutput vert(vertexInput input)
        {
            vertexOutput output;

            float4x4 modelMatrix = _Object2World;
            float4x4 modelMatrixInverse = _World2Object;
```

```
// multiplication with unity_Scale.w is unnecessary
// because we normalize transformed vectors

output.posWorld = mul(modelMatrix, input.vertex);
output.normalDir = normalize(float3(
    mul(float4(input.normal, 0.0), modelMatrixInverse)));
output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);

    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));
    float3 lightDirection =
        normalize(float3(_WorldSpaceLightPos0));

    float3 ambientLighting =
        float3(UNITY_LIGHTMODEL_AMBIENT) * float3(_Color);

    float3 diffuseReflection =
        float3(_LightColor0) * float3(_Color)
        * max(0.0, dot(normalDirection, lightDirection));

    float3 specularReflection;
    if (dot(normalDirection, lightDirection) < 0.0)
        // light source on the wrong side?
    {
        specularReflection = float3(0.0, 0.0, 0.0);
        // no specular reflection
    }
    else // light source on the right side
    {
        specularReflection = float3(_LightColor0)
            * float3(_SpecColor) * pow(max(0.0, dot(
                reflect(-lightDirection, normalDirection),
                viewDirection)), _Shininess);
    }

    return float4(ambientLighting + diffuseReflection
        + specularReflection, 1.0);
}

ENDCG
}
```

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

CGPROGRAM

#pragma multi_compile_lightpass

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

uniform float4x4 _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

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};
struct vertexOutput {
    float4 pos : SV_POSITION;
```

```
float4 posWorld : TEXCOORD0;
    // position of the vertex (and fragment) in world space
float4 posLight : TEXCOORD1;
    // position of the vertex (and fragment) in light space
float3 normalDir : TEXCOORD2;
    // surface normal vector in world space
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
        // multiplication with unity_Scale.w is unnecessary
        // because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.posLight = mul(_LightMatrix0, output.posWorld);
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);

    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));
    float3 lightDirection;
    float attenuation = 1.0;
        // by default no attenuation with distance

#if defined(DIRECTIONAL) || defined(DIRECTIONAL_COOKIE)
    lightDirection =
        normalize(float3(_WorldSpaceLightPos0));
#elif defined(POINT_NOATT)
    lightDirection = normalize(
        float3(_WorldSpaceLightPos0 - input.posWorld));
#elif defined(POINT) || defined(POINT_COOKIE) || defined(SPOT)
    float3 vertexToLightSource =
        float3(_WorldSpaceLightPos0 - input.posWorld);
    float distance = length(vertexToLightSource);
    attenuation = 1.0 / distance; // linear attenuation
    lightDirection = normalize(vertexToLightSource);

```

```
#endif

float3 diffuseReflection =
    attenuation * float3(_LightColor0) * float3(_Color)
    * max(0.0, dot(normalDirection, lightDirection));

float3 specularReflection;
if (dot(normalDirection, lightDirection) < 0.0)
    // light source on the wrong side?
{
    specularReflection = float3(0.0, 0.0, 0.0);
    // no specular reflection
}
else // light source on the right side
{
    specularReflection = attenuation * float3(_LightColor0)
        * float3(_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 = tex2D(_LightTexture0,
        float2(input.posLight)).a;
#elif defined (POINT_COOKIE)
    cookieAttenuation = texCUBE(_LightTexture0,
        float3(input.posLight)).a;
#elif defined (SPOT)
    cookieAttenuation = tex2D(_LightTexture0,
        float2(input.posLight)
        / input.posLight.w + float2(0.5)).a;
#endif

return float4(cookieAttenuation
    * (diffuseReflection + specularReflection), 1.0);
}

ENDCG
}
}

// 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 Section “Reflecting Surfaces”.

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.

Further Reading

If you still want to know more

- about the shader version for lights without cookies, you should read Section “Smooth Specular Highlights”.
- about texture mapping and in particular alpha texture maps, you should read Section “Transparent Textures”.
- 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^[1]).

page traffic for 90 days^[2]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] <http://developer.nvidia.com/content/projective-texture-mapping>

[2] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Cookies

5.4 Light Attenuation



"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 Section "Cookies". If you haven't read that tutorial yet, you should read it first.

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 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.

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; thus, you should use spot lights to test the code.) In Section "Diffuse Reflection", 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 Section "Cookies"; 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 Section "Cookies", we store the position in light space in the vertex output parameter `posLight`. We can then use the `z` coordinate of this parameter to look up the attenuation factor in the alpha component of the texture `_LightTextureB0` in the fragment shader:

```

        float distance = input.posLight.z;
        // use z coordinate in light space as signed distance
attenuation =
    tex2D(_LightTextureB0, float2(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.

Complete Shader Code

The shader code is based on the code of Section “Cookies”. 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 Section “Cookies” 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 "Cg 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
}

CGPROGRAM
#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix

```

```
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    float3 normalDir : TEXCOORD1;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
        // multiplication with unity_Scale.w is unnecessary
        // because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);

    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));
    float3 lightDirection =
        normalize(float3(_WorldSpaceLightPos0));

    float3 ambientLighting =
        float3(UNITY_LIGHTMODEL_AMBIENT) * float3(_Color);

    float3 diffuseReflection =
```

```
        float3(_LightColor0) * float3(_Color)
        * max(0.0, dot(normalDirection, lightDirection));

        float3 specularReflection;
        if (dot(normalDirection, lightDirection) < 0.0)
            // light source on the wrong side?
        {
            specularReflection = float3(0.0, 0.0, 0.0);
            // no specular reflection
        }
        else // light source on the right side
        {
            specularReflection = float3(_LightColor0)
                * float3(_SpecColor) * pow(max(0.0, dot(
                    reflect(-lightDirection, normalDirection),
                    viewDirection)), _Shininess);
        }

        return float4(ambientLighting + diffuseReflection
            + specularReflection, 1.0);
    }

    ENDCG
}

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

    CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
// (all but the bottom-right element have to be scaled
```

```
// with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

uniform float4x4 _LightMatrix0; // transformation
    // from world to light space (from Autolight.cginc)
uniform sampler2D _LightTextureB0;
    // cookie alpha texture map (from Autolight.cginc)

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
        // position of the vertex (and fragment) in world space
    float4 posLight : TEXCOORD1;
        // position of the vertex (and fragment) in light space
    float3 normalDir : TEXCOORD2;
        // surface normal vector in world space
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
        // multiplication with unity_Scale.w is unnecessary
        // because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.posLight = mul(_LightMatrix0, output.posWorld);
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);

    float3 viewDirection = normalize(
```

```
_WorldSpaceCameraPos = float3(input.posWorld));
float3 lightDirection;
float attenuation;

if (0.0 == _WorldSpaceLightPos0.w) // directional light?
{
    attenuation = 1.0; // no attenuation
    lightDirection =
        normalize(float3(_WorldSpaceLightPos0));
}
else // point or spot light
{
    float3 vertexToLightSource =
        float3(_WorldSpaceLightPos0 - input.posWorld);
    lightDirection = normalize(vertexToLightSource);

    float distance = input.posLight.z;
    // use z coordinate in light space as signed distance
    attenuation =
        tex2D(_LightTextureB0, float2(distance)).a;
    // texture lookup for attenuation
    // alternative with linear attenuation:
    //     float distance = length(vertexToLightSource);
    //     attenuation = 1.0 / distance;
}

float3 diffuseReflection =
    attenuation * float3(_LightColor0) * float3(_Color)
    * max(0.0, dot(normalDirection, lightDirection));

float3 specularReflection;
if (dot(normalDirection, lightDirection) < 0.0)
    // light source on the wrong side?
{
    specularReflection = float3(0.0, 0.0, 0.0);
    // no specular reflection
}
else // light source on the right side
{
    specularReflection = attenuation * float3(_LightColor0)
        * float3(_SpecColor) * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _Shininess);
}

return float4(diffuseReflection + specularReflection, 1.0);
```

```
        ENDCG
    }
}

// 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. Moreover, the code doesn't check the sign of the *z* coordinate, which is fine for spot lights but results in a lack of attenuation on one side of point light sources.

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.

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.

Further Reading

If you still want to know more

- about light attenuation for light sources, you should read Section “Diffuse Reflection”.
- about basic texture mapping, you should read Section “Textured Spheres”.
- about coordinates in light space, you should read Section “Cookies”.
- 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 [1].

page traffic for 90 days [2]

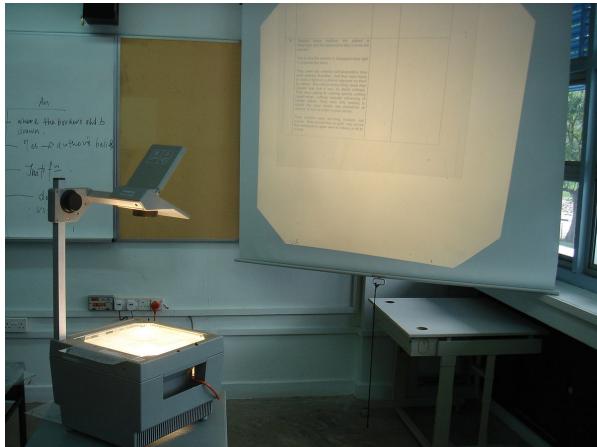
< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

- [1] <http://syque.com/cstyle/index.htm>
- [2] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Light_Attenuation

5.5 Projectors



An overhead projector.

This tutorial covers **projective texture mapping for projectors**, which are particular rendering components of Unity.

It is based on Section “Cookies”. If you haven't read that tutorial yet, you should read it first.

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 Section “Cookies”. 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 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 Section “Transparency”.)

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 colors as an example of modulating them.

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^[1]. 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 Section “Cookies”) 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

Blend One One

which just adds 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 `tex2Dproj`:

```
Shader "Cg 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

            CGPROGRAM
                #pragma vertex vert
                #pragma fragment frag

                // User-specified properties
                uniform sampler2D _ShadowTex;

                // Projector-specific uniforms
                uniform float4x4 _Projector; // transformation matrix
                    // from object space to projector space

                struct vertexInput {
                    float4 vertex : POSITION;
                    float3 normal : NORMAL;
                };
                struct vertexOutput {
                    float4 pos : SV_POSITION;
                    float4 posProj : TEXCOORD0;
                        // position in projector space
                };

                vertexOutput vert(vertexInput input)
                {
                    vertexOutput output;

                    output.posProj = mul(_Projector, input.vertex);
                    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
                }
            ENDCG
        }
    }
}
```

```

        return output;
    }

    float4 frag(vertexOutput input) : COLOR
    {
        if (input.posProj.w > 0.0) // in front of projector?
        {
            return tex2D(_ShadowTex ,
                float2(input.posProj) / input.posProj.w);
            // alternatively: return = tex2Dproj(
            //     _ShadowTex, float3(input.posProj));
        }
        else // behind projector
        {
            return float4(0.0);
        }
    }

    ENDCG
}
}

// 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.



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 "Cg projector shader for drop shadows" {
    Properties {
        _ShadowTex ("Projected Image", 2D) = "white" {}
    }
    SubShader {
        Pass {
            Blend Zero OneMinusSrcAlpha // attenuate color in framebuffer
            // by 1 minus alpha of _ShadowTex
            CGPROGRAM

                #pragma vertex vert
                #pragma fragment frag

                // User-specified properties
                uniform sampler2D _ShadowTex;

                // Projector-specific uniforms
                uniform float4x4 _Projector; // transformation matrix
                // from object space to projector space

                struct vertexInput {
                    float4 vertex : POSITION;
                    float3 normal : NORMAL;
                };
                struct vertexOutput {
                    float4 pos : SV_POSITION;
                    float4 posProj : TEXCOORD0;
                    // position in projector space
                };

                vertexOutput vert(vertexInput input)
                {
                    vertexOutput output;

                    output.posProj = mul(_Projector, input.vertex);
                    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
                    return output;
                }

                float4 frag(vertexOutput input) : COLOR
            ENDCG
        }
    }
}
```

```
{  
    if (input.posProj.w > 0.0) // in front of projector?  
    {  
        return tex2D(_ShadowTex ,  
                    float2(input.posProj) / input.posProj.w);  
        // alternatively: return = tex2Dproj(  
        //      _ShadowTex, float3(input.posProj));  
    }  
    else // behind projector  
    {  
        return float4(0.0);  
    }  
}  
  
ENDCG  
}  
}  
// The definition of a fallback shader should be commented out  
// during development:  
// Fallback "Projector/Light"  
}
```

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.

Further Reading

If you still want to know more

- about the light space (which is very similar to projector space), you should read Section “Cookies”.
- about texture mapping and in particular alpha texture maps, you should read Section “Transparent Textures”.
- 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 ^[1]).
- about Unity's projectors, you should read Unity's documentation about projectors ^[1].

page traffic for 90 days ^[2]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

- [1] <http://unity3d.com/support/documentation/Components/class-Projector.html>
- [2] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Projectors

6 Environment Mapping

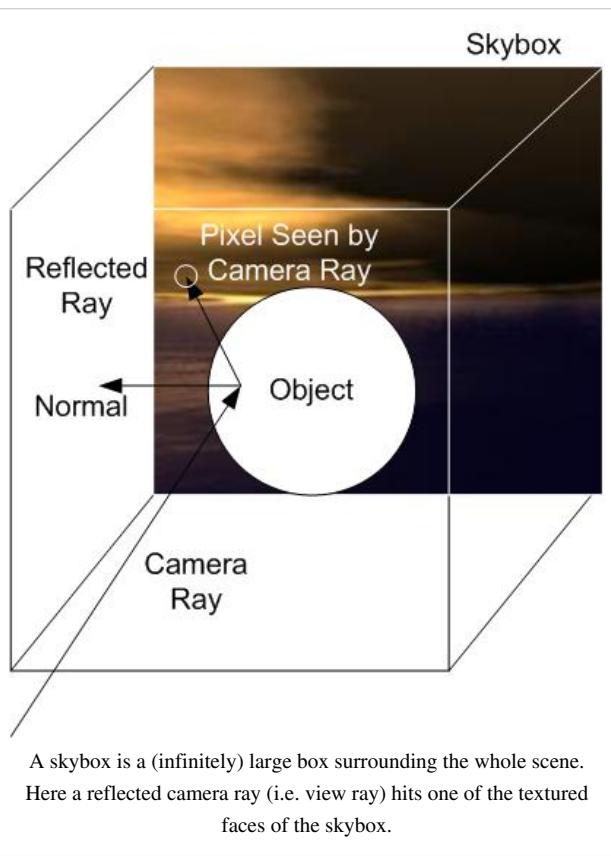
6.1 Reflecting Surfaces



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 Section “Smooth Specular Highlights” and on the concept of texture mapping, which was introduced in Section “Textured Spheres”.



Reflection Mapping with a Skybox

The illustration to the left 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 Section “Skyboxes”. 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 Section “Specular Highlights”.

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 Section "Textured Spheres") in the texture image for the specific face. However, Cg 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 `texCUBE` instruction with the reflected direction to get the color at the corresponding position in the cube map.

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 Cg shader:

```
uniform samplerCUBE _Cube;
```

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 `viewDir` and the normal direction `normalDir` as discussed in Section "Specular Highlights". To reflect the view direction in the fragment shader, we can use the Cg function `reflect` as also discussed in Section "Specular Highlights":

```
float3 reflectedDir =
    reflect(input.viewDir, normalize(input.normalDir));
```

And to perform the texture lookup in the cube map and store the resulting color in the fragment color, we simply use:

```
return texCUBE(_Cube, reflectedDir);
```

That's about it.

Complete Shader Code

The shader code then becomes:

```
Shader "Cg shader with reflection map" {
Properties {
    _Cube ("Reflection Map", Cube) = " "
}
SubShader {
    Pass {
```

```
CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified uniforms
uniform samplerCUBE _Cube;

// The following built-in uniforms are also
// defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float3 normalDir : TEXCOORD0;
    float3 viewDir : TEXCOORD1;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
    // multiplication with unity_Scale.w is unnecessary
    // because we normalize transformed vectors

    output.viewDir = float3(mul(modelMatrix, input.vertex)
        - float4(_WorldSpaceCameraPos, 1.0));
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 reflectedDir =
        reflect(input.viewDir, normalize(input.normalDir));
    return texCUBE(_Cube, reflectedDir);
```

```
    }

    ENDCG
}

}

}
```

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.

Further Reading

If you still want to know more

- about the reflection of vectors, you should read Section “Specular Highlights”.
- about cube maps in Unity, you should read Unity's documentation about cube maps ^[1].

page traffic for 90 days ^[2]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] <http://unity3d.com/support/documentation/Components/class-Cubemap.html>

[2] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Reflecting_Surfaces

6.2 Curved Glass



Crystal balls are examples of curved, transparent surfaces.

This tutorial covers **refraction mapping** and its implementation with cube maps.

It is a variation of Section “Reflecting Surfaces”, which should be read first.

Refraction Mapping

In Section “Reflecting Surfaces”, we reflected view rays and then performed texture lookups in a cube map in the reflected direction. 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:

```
float4 frag(vertexOutput input) : COLOR
{
    float refractiveIndex = 1.5;
    float3 refractedDir = refract(normalize(input.viewDir),
        normalize(input.normalDir), 1.0 / refractiveIndex);
    return texCUBE(_Cube, refractedDir);
}
```

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`.

Complete Shader Code

With the adapted fragment shader, the complete shader code becomes:

```
Shader "Cg shader with refraction mapping"
{
    Properties {
        _Cube("Reflection Map", Cube) = ""
    }
    SubShader {
        Pass {
            CGPROGRAM

                #pragma vertex vert
                #pragma fragment frag

                // User-specified uniforms
            ENDCG
        }
    }
}
```

```
uniform samplerCUBE _Cube;

// The following built-in uniforms are also
// defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float3 normalDir : TEXCOORD0;
    float3 viewDir : TEXCOORD1;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
    // multiplication with unity_Scale.w is unnecessary
    // because we normalize transformed vectors

    output.viewDir = float3(mul(modelMatrix, input.vertex)
        - float4(_WorldSpaceCameraPos, 1.0));
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float refractiveIndex = 1.5;
    float3 refractedDir = refract(normalize(input.viewDir),
        normalize(input.normalDir), 1.0 / refractiveIndex);
    return texCUBE(_Cube, refractedDir);
}

ENDCG
}
```

```
}
```

Summary

Congratulations. This is the end of another tutorial. We have seen:

- How to adapt reflection mapping to refraction mapping using the `refract` instruction.

Further Reading

If you still want to know more

- about reflection mapping and cube maps, you should read Section “Reflecting Surfaces”.
- about the `refract` instruction, you could look it up in appendix E of Nivida's Cg Tutorial [1].

page traffic for 90 days ^[2]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] http://http.developer.nvidia.com/CgTutorial/cg_tutorial_appendix_e.html

[2] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Curved_Glass

6.3 Skyboxes



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 Section “Reflecting Surfaces”. If you haven't read that tutorial, this would be a very good time to read it.

Rendering a Skybox in the Background

As explained in Section “Reflecting Surfaces”, 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, Section “Reflecting Surfaces” 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 Section “Reflecting Surfaces”) and then perform a lookup in the cube map with this view vector (instead of the reflected view vector in Section “Reflecting Surfaces”) in the fragment shader:

```
float4 frag(vertexOutput input) : COLOR
{
    return texCUBE(_Cube, input.viewDir);
}
```

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.

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 Section “Per-Fragment Operations”.) Front-face culling is activated with `Cull Front` such that only the “inside” of the cube is rasterized. (See Section “Cutaways”.) The line `Tags { "Queue" = "Background" }` instructs Unity to render this pass before other objects are rendered.

```
Shader "Cg shader for skybox" {
    Properties {
        _Cube ("Environment Map", Cube) = "" {}
    }
    SubShader {
        Tags { "Queue" = "Background" }

        Pass {
            ZWrite Off
            Cull Front

            CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified uniforms
uniform samplerCUBE _Cube;

// The following built-in uniforms are also
// defined in "UnityCG.cginc", which could be #included
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix

struct vertexInput {
    float4 vertex : POSITION;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float3 viewDir : TEXCOORD1;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    output.viewDir = float3(mul(modelMatrix, input.vertex)
```

```
        - float4 (_WorldSpaceCameraPos, 1.0));
    output.pos = mul (UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    return texCUBE (_Cube, input.viewDir);
}

ENDCG
}
}
```

Shader Code for Unity's Skybox System

The shader above illustrates how to render a skybox by rendering a cube around the camera with a specific shader. This is a very general approach. Unity, however, has its own skybox system that doesn't require any game object: you just specify the material with the skybox shader in the main menu **Edit > Render Settings > Skybox Material** and Unity takes care of the rest. Unfortunately, we cannot use our shader for this system since we have to do the lookup in the cube texture at the position that is specified by the vertex texture coordinates. This is actually easier than computing the view direction. Here is the code:

```
Shader "Cg shader for Unity-specific skybox" {
Properties {
    _Cube ("Environment Map", Cube) = "white" {}
}

SubShader {
    Tags { "Queue"="Background" }

    Pass {
        ZWrite Off
        Cull Off

        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag

        // User-specified uniforms
        samplerCUBE _Cube;

        struct vertexInput {
            float4 vertex : POSITION;
            float3 texcoord : TEXCOORD0;
        };

        struct vertexOutput {
```

```
float4 vertex : SV_POSITION;
float3 texcoord : TEXCOORD0;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;
    output.vertex = mul(UNITY_MATRIX_MVP, input.vertex);
    output.texcoord = input.texcoord;
    return output;
}

fixed4 frag (vertexOutput input) : COLOR
{
    return texCUBE (_Cube, input.texcoord);
}
ENDCG
}
}
```

As mentioned above, you should create a material with this shader and drag the material to **Edit > Render Settings > Skybox Material**. There is no need to attach the material to any game object.

Summary

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

- How to render skyboxes in general.
- How to render skyboxes in Unity without a game object.

Further Reading

If you still want to know more

- about cube maps and reflections of skyboxes in objects, you should read Section “Reflecting Surfaces”.
- about lighting that is consistent with a skybox, you should read Section “Many Light Sources”.

page traffic for 90 days ^[1]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Skyboxes

6.4 Many Light Sources



"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 Section "Reflecting Surfaces". If you haven't read that tutorial, this would be a very good time to read it.

Diffuse Lighting by Many Lights

Consider the lighting of the sculpture in the image to the left. 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 Section "Reflecting Surfaces" 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 Section "Diffuse Reflection". It's basically the cosine between the surface normal vector \mathbf{N} and the vector to the light source \mathbf{L} :

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

Since the texels are the light sources, \mathbf{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 \mathbf{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 \mathbf{N} in that precomputed lookup table.

Thus, for a specific surface normal vector \mathbf{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 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 Section "Reflecting Surfaces"):

```
float4 frag(vertexOutput input) : COLOR
{

```

```

        return texCUBE(_Cube, input.normalDir);
    }
}

```

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.

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 you should use small 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 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

```

<font size="9.00">
@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;  
}  
</font>
```

As an alternative to the JavaScript code above, you can also use the following C# code.

C# code: click to show/hide

```
<font size="9.00">  
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 irradiance 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 iterating 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;
}
}
}
</font>
```

Complete Shader Code

As promised, the actual shader code is very short; the vertex shader is a reduced version of the vertex shader of Section “Reflecting Surfaces”:

```

Shader "Cg shader with image-based diffuse lighting" {
Properties {
    _OriginalCube ("Environment Map", Cube) = "" {}
    _Cube ("Diffuse Environment Map", Cube) = "" {}
}
SubShader {
    Pass {
        CGPROGRAM

        #pragma vertex vert
        #pragma fragment frag

        // User-specified uniforms
        uniform samplerCUBE _Cube;

        // The following built-in uniform is also
        // defined in "UnityCG.cginc", which could be #included
        uniform float4x4 _World2Object; // inverse model matrix

        struct vertexInput {
            float4 vertex : POSITION;
            float3 normal : NORMAL;
        };
        struct vertexOutput {
```

```

        float4 pos : SV_POSITION;
        float3 normalDir : TEXCOORD0;
    };

    vertexOutput vert(vertexInput input)
    {
        vertexOutput output;

        float4x4 modelMatrixInverse = _World2Object;
        // multiplication with unity_Scale.w is unnecessary
        // because we normalize transformed vectors

        output.normalDir = normalize(float3(
            mul(float4(input.normal, 0.0), modelMatrixInverse)));
        output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
        return output;
    }

    float4 frag(vertexOutput input) : COLOR
    {
        return texCUBE(_Cube, input.normalDir);
    }
}

ENDCG
}
}
}

```

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 Section “Specular Highlights”, 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}_{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}_{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 Section “Reflecting Surfaces” 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.)

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.

Further Reading

If you still want to know more

- about cube maps, you should read Section “Reflecting Surfaces”.
- 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^[1].

page traffic for 90 days^[2]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter10.html

[2] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Many_Light_Sources

7 Variations on Lighting

7.1 Brushed Metal

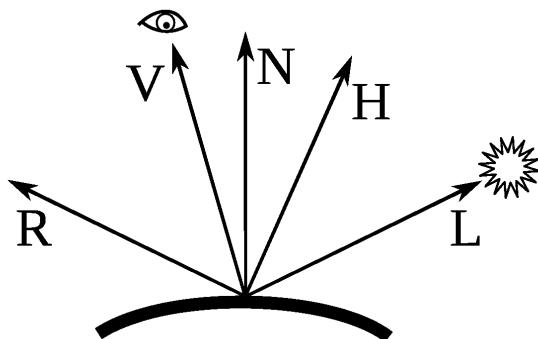


Brushed aluminium. Note the form of the specular highlights, which is far from being round.

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 Section “Specular Highlights” (for per-vertex lighting) and Section “Smooth Specular Highlights” (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 to the left.



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 .

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^[1].) 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. In principle, a reflectance for each wavelength is necessary; however, usually one reflectance for each of the three color components (red, green, and blue) is specified. If we include the constant π , ρ_d/π just represents the diffuse material color k_{diffuse} , which we have first seen in Section “Diffuse Reflection” but which also appears in the Phong reflection model (see Section “Specular Highlights”). You might wonder why the factor $\max(0, \mathbf{L} \cdot \mathbf{N})$ 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 $\max(0, \mathbf{L} \cdot \mathbf{N})$ 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 \mathbf{N} , the normalized direction to the viewer \mathbf{V} , the normalized direction to the light source \mathbf{L} , and the normalized halfway vector \mathbf{H} which is $(\mathbf{V} + \mathbf{L}) / |\mathbf{V} + \mathbf{L}|$. 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 $\mathbf{L} \cdot \mathbf{N}$ when implementing it in a shader and set it to 0 if $\mathbf{L} \cdot \mathbf{N}$ is less than 0. Furthermore, it should also be 0 if $\mathbf{V} \cdot \mathbf{N}$ 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: \mathbf{T} and \mathbf{B} . \mathbf{T} is the brush direction on the surface and \mathbf{B} is orthogonal to \mathbf{T} but also on the surface. Unity provides us with a tangent vector on the surface as a vertex attribute (see Section “Debugging of Shaders”), which we will use as the vector \mathbf{T} . Computing the cross product of \mathbf{N} and \mathbf{T} generates a vector \mathbf{B} , which is orthogonal to \mathbf{N} and \mathbf{T} , as it should be.

Implementation of Ward's BRDF Model

We base our implementation on the shader for per-pixel lighting in Section “Smooth Specular Highlights”. We need another vertex output parameter `tangentDir` for the tangent vector \mathbf{T} (i.e. the brush direction) and we also compute `viewDir` in the vertex shader to save some instructions in the fragment shader. In the fragment shader, we compute two more directions: `halfwayVector` for the halfway vector \mathbf{H} and `binormalDirection` for the binormal vector \mathbf{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 Section “Smooth Specular Highlights” except that it computes `halfwayVector` and `binormalDirection`, and implements a different equation for the specular part. Furthermore, this shader computes the dot product $\mathbf{L} \cdot \mathbf{N}$ only once and stores it in `dotLN` such that it can be reused without having to recompute it. It looks like this:

```
float4 frag(vertexOutput input) : COLOR
{
    float3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
}
```

```
    else // point or spot light
    {
        float3 vertexToLightSource =
            float3(_WorldSpaceLightPos0 - input.posWorld);
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
        lightDirection = normalize(vertexToLightSource);
    }

    float3 halfwayVector =
        normalize(lightDirection + input.viewDir);
    float3 binormalDirection =
        cross(input.normalDir, input.tangentDir);
    float dotLN = dot(lightDirection, input.normalDir);
    // compute this dot product only once

    float3 ambientLighting =
        float3(UNITY_LIGHTMODEL_AMBIENT) * float3(_Color);

    float3 diffuseReflection =
        attenuation * float3(_LightColor0) * float3(_Color)
        * max(0.0, dotLN);

    float3 specularReflection;
    if (dotLN < 0.0) // light source on the wrong side?
    {
        specularReflection = float3(0.0, 0.0, 0.0);
        // no specular reflection
    }
    else // light source on the right side
    {
        float dotHN = dot(halfwayVector, input.normalDir);
        float dotVN = dot(input.viewDir, input.normalDir);
        float dotHTAlphaX =
            dot(halfwayVector, input.tangentDir) / _AlphaX;
        float dotHBAAlphaY = dot(halfwayVector,
            binormalDirection) / _AlphaY;

        specularReflection = attenuation * float3(_SpecColor)
            * sqrt(max(0.0, dotLN / dotVN))
            * exp(-2.0 * (dotHTAlphaX * dotHTAlphaX
            + dotHBAAlphaY * dotHBAAlphaY) / (1.0 + dotHN));
    }

    return float4(ambientLighting + diffuseReflection
        + specularReflection, 1.0);
}
```

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 $(\mathbf{L} \cdot \mathbf{N})$. This makes sure that everything is greater than 0.

Complete Shader Code

The complete shader code just defines the appropriate properties and adds another vertex input parameter for the tangent. Also, it requires a second pass with additive blending but without ambient lighting for additional light sources.

```
Shader "Cg 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

            CGPROGRAM
                #pragma vertex vert
                #pragma fragment frag

                // User-specified properties
                uniform float4 _Color;
                uniform float4 _SpecColor;
                uniform float _AlphaX;
                uniform float _AlphaY;

                // The following built-in uniforms (apart from _LightColor0)
                // are defined in "UnityCG.cginc", which could be #included
                uniform float4 unity_Scale; // w = 1/scale; see _World2Object
                uniform float3 _WorldSpaceCameraPos;
                uniform float4x4 _Object2World; // model matrix
                uniform float4x4 _World2Object; // inverse model matrix
                // (all but the bottom-right element have to be scaled
                // with unity_Scale.w if scaling is important)
                uniform float4 _WorldSpaceLightPos0;
                // position or direction of light source
                uniform float4 _LightColor0;
                // color of light source (from "Lighting.cginc")

                struct vertexInput {
                    float4 vertex : POSITION;
```

```
float3 normal : NORMAL;
float4 tangent : TANGENT;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
        // position of the vertex (and fragment) in world space
    float3 viewDir : TEXCOORD1;
        // view direction in world space
    float3 normalDir : TEXCOORD2;
        // surface normal vector in world space
    float3 tangentDir : TEXCOORD3;
        // brush direction in world space
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
        // multiplication with unity_Scale.w is unnecessary
        // because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.viewDir = normalize(_WorldSpaceCameraPos
        - float3(output.posWorld));
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.tangentDir = normalize(float3(
        mul(modelMatrix, float4(float3(input.tangent), 0.0))));
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else // point or spot light
```

```
{  
    float3 vertexToLightSource =  
        float3(_WorldSpaceLightPos0 - input.posWorld);  
    float distance = length(vertexToLightSource);  
    attenuation = 1.0 / distance; // linear attenuation  
    lightDirection = normalize(vertexToLightSource);  
}  
  
float3 halfwayVector =  
    normalize(lightDirection + input.viewDir);  
float3 binormalDirection =  
    cross(input.normalDir, input.tangentDir);  
float dotLN = dot(lightDirection, input.normalDir);  
    // compute this dot product only once  
  
float3 ambientLighting =  
    float3(UNITY_LIGHTMODEL_AMBIENT) * float3(_Color);  
  
float3 diffuseReflection =  
    attenuation * float3(_LightColor0) * float3(_Color)  
    * max(0.0, dotLN);  
  
float3 specularReflection;  
if (dotLN < 0.0) // light source on the wrong side?  
{  
    specularReflection = float3(0.0, 0.0, 0.0);  
    // no specular reflection  
}  
else // light source on the right side  
{  
    float dotHN = dot(halfwayVector, input.normalDir);  
    float dotVN = dot(input.viewDir, input.normalDir);  
    float dotHTAlphaX =  
        dot(halfwayVector, input.tangentDir) / _AlphaX;  
    float dotHBAAlphaY = dot(halfwayVector,  
        binormalDirection) / _AlphaY;  
  
    specularReflection = attenuation * float3(_SpecColor)  
    * sqrt(max(0.0, dotLN / dotVN))  
    * exp(-2.0 * (dotHTAlphaX * dotHTAlphaX  
    + dotHBAAlphaY * dotHBAAlphaY) / (1.0 + dotHN));  
}  
  
return float4(ambientLighting + diffuseReflection  
    + specularReflection, 1.0);  
}
```

```
ENDCG
}

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

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _AlphaX;
uniform float _AlphaY;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
// (all but the bottom-right element have to be scaled
// with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
// position or direction of light source
uniform float4 _LightColor0;
// color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
    float4 tangent : TANGENT;
};
struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
        // position of the vertex (and fragment) in world space
    float3 viewDir : TEXCOORD1;
        // view direction in world space
    float3 normalDir : TEXCOORD2;
        // surface normal vector in world space
    float3 tangentDir : TEXCOORD3;
        // brush direction in world space
```

```
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
    // multiplication with unity_Scale.w is unnecessary
    // because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.viewDir = normalize(_WorldSpaceCameraPos
        - float3(output.posWorld));
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.tangentDir = normalize(float3(
        mul(modelMatrix, float4(float3(input.tangent), 0.0))));
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else // point or spot light
    {
        float3 vertexToLightSource =
            float3(_WorldSpaceLightPos0 - input.posWorld);
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
        lightDirection = normalize(vertexToLightSource);
    }

    float3 halfwayVector =
        normalize(lightDirection + input.viewDir);
    float3 binormalDirection =
        cross(input.normalDir, input.tangentDir);
    float dotLN = dot(lightDirection, input.normalDir);
```

```
// compute this dot product only once

float3 diffuseReflection =
    attenuation * float3(_LightColor0) * float3(_Color)
    * max(0.0, dotLN);

float3 specularReflection;
if (dotLN < 0.0) // light source on the wrong side?
{
    specularReflection = float3(0.0, 0.0, 0.0);
    // no specular reflection
}
else // light source on the right side
{
    float dotHN = dot(halfwayVector, input.normalDir);
    float dotVN = dot(input.viewDir, input.normalDir);
    float dotHTAlphaX =
        dot(halfwayVector, input.tangentDir) / _AlphaX;
    float dotHBAlphaY = dot(halfwayVector,
        binormalDirection) / _AlphaY;

    specularReflection = attenuation * float3(_SpecColor)
        * sqrt(max(0.0, dotLN / dotVN))
        * exp(-2.0 * (dotHTAlphaX * dotHTAlphaX
            + dotHBAlphaY * dotHBAlphaY) / (1.0 + dotHN));
}

return float4(diffuseReflection
    + specularReflection, 1.0);
}

ENDCG
}
}

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

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.

Further Reading

If you still want to know more

- about lighting with the Phong reflection model, you should read Section "Specular Highlights".
- about per-pixel lighting (i.e. Phong shading), you should read Section "Smooth Specular Highlights".
- 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^[1].) 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" (2nd edition, 2008) by Wolfgang Engel, Jack Hoxley, Ralf Kornmann, Niko Suni, and Jason Zink (which is available online^[2].)

page traffic for 90 days^[3]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

- [1] <http://radsite.lbl.gov/radiance/papers/sg92/paper.html>
- [2] http://wiki.gamedev.net/index.php/D3DBook:Table_Of_Contents
- [3] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Brushed_Metal

7.2 Specular Highlights at Silhouettes



Photo of pedestrians in Lisbon. Note the bright silhouettes due to the backlight.

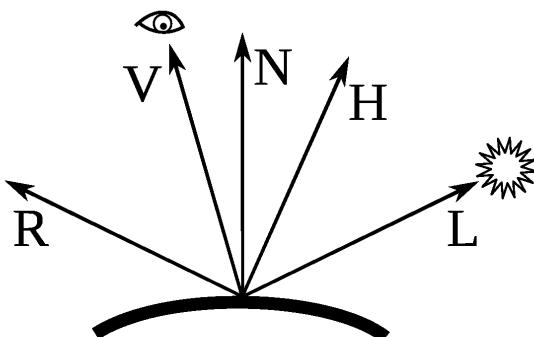
This tutorial covers the **Fresnel factor for specular highlights**.

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 Section “Specular Highlights” (for per-vertex lighting) and Section “Smooth Specular Highlights” (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 to the left. 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 Section “Translucent Surfaces”).

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.



In addition to most of the vectors used by the Phong reflection model, we require the normalized halfway vector \mathbf{H} , which is the direction exactly between the direction to the viewer \mathbf{V} and the direction to the light source \mathbf{L} .

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 \mathbf{V} is the normalized direction to the viewer and \mathbf{H} is the normalized halfway vector: $\mathbf{H} = (\mathbf{V} + \mathbf{L}) / |\mathbf{V} + \mathbf{L}|$ with \mathbf{L} being the normalized direction to the light source. f_λ is the reflectance for $\mathbf{H} \cdot \mathbf{V} = 1$, i.e. when the direction to the light source, the direction to the viewer, and the halfway vector are all identical. On the other hand, F_λ becomes 1 for $\mathbf{H} \cdot \mathbf{V} = 0$, 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 Cg function $\text{lerp}(x, y, w) = x * (1-w) + y * w$ we can rewrite Schlick's approximation as:

$$\begin{aligned} F_\lambda &= f_\lambda + (1 - f_\lambda)(1 - \mathbf{H} \cdot \mathbf{V})^5 \\ &= \text{lerp}(f_\lambda, 1, (1 - \mathbf{H} \cdot \mathbf{V})^5) \end{aligned}$$

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 Section “Specular Highlights”. 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 of the specular reflection.

For example, our equation for the specular term in the Phong reflection model was (see Section “Specular Highlights”):

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

Replacing k_{specular} by Schlick's approximation for the Fresnel factor with $f_\lambda = k_{\text{specular}}$ yields:

$$I_{\text{specular}} = I_{\text{incoming}} \text{lerp}(k_{\text{specular}}, 1, (1 - \mathbf{H} \cdot \mathbf{V})^5) \max(0, \mathbf{R} \cdot \mathbf{V})^{n_{\text{shininess}}}$$

Implementation

The implementation is based on the shader code from Section “Smooth Specular Highlights”. It just computes the halfway vector and includes the approximation of the Fresnel factor:

```
float3 specularReflection;
if (dot(normalDirection, lightDirection) < 0.0)
    // light source on the wrong side?
{
    specularReflection = float3(0.0, 0.0, 0.0);
    // no specular reflection
}
else // light source on the right side
{
    float3 halfwayDirection =
        normalize(lightDirection + viewDirection);
    float w = pow(1.0 - max(0.0,
        dot(halfwayDirection, viewDirection)), 5.0);
    specularReflection = attenuation * float3(_LightColor0)
        * lerp(float3(_SpecColor), float3(1.0), w)
        * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _Shininess);
}
```

Complete Shader Code

Putting the code snippet from above in the complete shader from Section “Smooth Specular Highlights” results in this shader:

```
Shader "Cg 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
    }
}
```

```
CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
// (all but the bottom-right element have to be scaled
// with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
// position or direction of light source
uniform float4 _LightColor0;
// color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    float3 normalDir : TEXCOORD1;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
// multiplication with unity_Scale.w is unnecessary
// because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
}
```

```
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);
    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));
    float3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else // point or spot light
    {
        float3 vertexToLightSource =
            float3(_WorldSpaceLightPos0 - input.posWorld);
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
        lightDirection = normalize(vertexToLightSource);
    }

    float3 ambientLighting =
        float3(UNITY_LIGHTMODEL_AMBIENT) * float3(_Color);

    float3 diffuseReflection =
        attenuation * float3(_LightColor0) * float3(_Color)
        * max(0.0, dot(normalDirection, lightDirection));

    float3 specularReflection;
    if (dot(normalDirection, lightDirection) < 0.0)
        // light source on the wrong side?
    {
        specularReflection = float3(0.0, 0.0, 0.0);
        // no specular reflection
    }
    else // light source on the right side
    {
        float3 halfwayDirection =
            normalize(lightDirection + viewDirection);
        float w = pow(1.0 - max(0.0,
            dot(halfwayDirection, viewDirection)), 5.0);
        specularReflection = attenuation * float3(_LightColor0)
```

```
        * lerp(float3(_SpecColor), float3(1.0), w)
        * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _Shininess);
    }

    return float4(ambientLighting
        + diffuseReflection + specularReflection, 1.0);
}

ENDCG
}

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

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
// (all but the bottom-right element have to be scaled
// with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
// position or direction of light source
uniform float4 _LightColor0;
// color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};
struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
```

```
float3 normalDir : TEXCOORD1;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
    // multiplication with unity_Scale.w is unnecessary
    // because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);
    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));
    float3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else // point or spot light
    {
        float3 vertexToLightSource =
            float3(_WorldSpaceLightPos0 - input.posWorld);
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
        lightDirection = normalize(vertexToLightSource);
    }

    float3 diffuseReflection =
        attenuation * float3(_LightColor0) * float3(_Color)
        * max(0.0, dot(normalDirection, lightDirection));

    float3 specularReflection;
```

```

        if (dot(normalDirection, lightDirection) < 0.0)
            // light source on the wrong side?
        {
            specularReflection = float3(0.0, 0.0, 0.0);
            // no specular reflection
        }
        else // light source on the right side
        {
            float3 halfwayDirection =
                normalize(lightDirection + viewDirection);
            float w = pow(1.0 - max(0.0,
                dot(halfwayDirection, viewDirection)), 5.0);
            specularReflection = attenuation * float3(_LightColor0)
                * lerp(float3(_SpecColor), float3(1.0), w)
                * pow(max(0.0, dot(
                    reflect(-lightDirection, normalDirection),
                    viewDirection)), _Shininess);
        }

        return float4(diffuseReflection
            + specularReflection, 1.0);
    }

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

```

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.

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 Section “Silhouette Enhancement” 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}), i.e.:

$$\alpha_{\text{total}} = 1 - T_{\text{total}} = 1 - T_{\text{attenuation}} T_{\text{Fresnel}} = 1 - (1 - \alpha_{\text{attenuation}})(1 - \alpha_{\text{Fresnel}})$$

α_{Fresnel} is the opacity as computed above while $\alpha_{\text{attenuation}}$ is the opacity as computed in Section “Silhouette Enhancement”. 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 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 $\alpha_{\text{attenuation}}$.

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.

Further Reading

If you still want to know more

- about lighting with the Phong reflection model, you should read Section “Specular Highlights”.
- about per-pixel lighting (i.e. Phong shading), you should read Section “Smooth Specular Highlights”.
- 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 ^[2].)

page traffic for 90 days ^[1]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Specular_Highlights_at_Silhouettes

7.3 Diffuse Reflection of Skylight

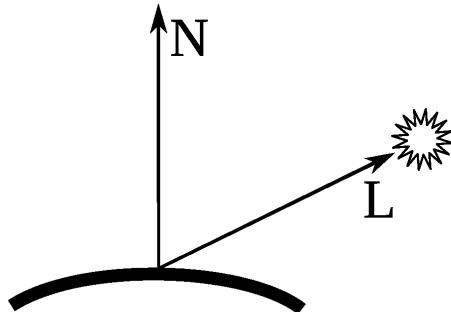


A spherical building illuminated by an overcast sky from above and a green water basin from below. Note the different colors of the illumination of the building.

This tutorial covers **hemisphere lighting**.

It is based on diffuse per-vertex lighting as described in Section “Diffuse Reflection”. 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 to the left, 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.



Diffuse reflection can be calculated with the surface normal vector \mathbf{N} and the direction to the light source \mathbf{L} .

Hemisphere Lighting

If we assume that each point (in direction \mathbf{L}) of a hemisphere around a point on the surface acts as a light source, then we should integrate the diffuse illumination (given by $\max(0, \mathbf{L} \cdot \mathbf{N})$) as discussed in Section “Diffuse Reflection”) from all the points on the hemisphere by an integral. Let’s call the normalized direction of the rotation axis of the hemisphere \mathbf{U} (for “up”). If the surface normal \mathbf{N} points in the direction of \mathbf{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) of the hemisphere illuminates the surface point.

The fraction w of this illumination in comparison to the full illumination is:

$$w = \frac{1}{2}(1 + \cos(\gamma)) = \frac{1}{2}(1 + \mathbf{U} \cdot \mathbf{N})$$

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 .

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 \mathbf{N} in the direction of the y axis. If \mathbf{N} and \mathbf{U} point in the same direction, the integral is (apart from a constant color specified by the user):

$$\int_0^\pi d\phi \int_0^\pi d\theta \sin(\theta) \mathbf{L} \cdot \mathbf{N} = \int_0^\pi d\phi \int_0^\pi d\theta \sin(\theta) \begin{pmatrix} x \\ y \\ z \end{pmatrix} \cdot \mathbf{N}$$

The term $\sin(\theta)$ is the Jacobian determinant for our integration on the surface of a sphere of radius 1, (x, y, z) is $(\cos(\varphi)\sin(\theta), \sin(\varphi)\sin(\theta), \cos(\theta))$, and $\mathbf{N} = (0, 1, 0)$. Thus, the integral becomes:

$$\int_0^\pi d\phi \int_0^\pi d\theta (\sin(\theta))^2 \sin(\phi) = \pi$$

The constant π will be included in the user-defined color of the maximum illumination. If there is an angle γ with $\cos(\gamma) = \mathbf{U} \cdot \mathbf{N}$ between \mathbf{N} and \mathbf{U} , then the integration is only over a spherical wedge (from γ to π):

$$w = \frac{1}{\pi} \int_\gamma^\pi d\phi \int_0^\pi d\theta (\sin(\theta))^2 \sin(\phi) = \frac{1}{2}(1 + \cos(\gamma)) = \frac{1}{2}(1 + \mathbf{U} \cdot \mathbf{N})$$

Shader Code

The implementation is based on the code from Section "Diffuse Reflection". 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 Section "Specular Highlights". 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:

$$w = \frac{1}{2}(1 + \mathbf{U} \cdot \mathbf{N})$$

We implement it in world space, i.e. we have to transform the surface normal vector \mathbf{N} to world space (see Section "Shading in World Space"), while \mathbf{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 "Cg 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 {
            CGPROGRAM
                #pragma vertex vert
                #pragma fragment frag
            ENDCG
        }
    }
}
```

```
// shader properties specified by users
uniform float4 _Color;
uniform float4 _UpperHemisphereColor;
uniform float4 _LowerHemisphereColor;
uniform float4 _UpVector;

// The following built-in uniforms
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 col : COLOR;
    // the hemisphere lighting computed in the vertex shader
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
        // multiplication with unity_Scale.w is unnecessary
        // because we normalize transformed vectors

    float3 normalDirection = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    float3 upDirection = normalize(_UpVector);

    float w = 0.5 * (1.0 + dot(upDirection, normalDirection));
    output.col = (w * _UpperHemisphereColor
        + (1.0 - w) * _LowerHemisphereColor) * _Color;

    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
```

```
{  
    return input.col;  
}  
  
ENDCG  
}  
}  
}
```

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.

Further Reading

If you still want to know more

- about lighting with the diffuse reflection, you should read Section “Diffuse Reflection”.
- 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.

page traffic for 90 days [1]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Diffuse_Reflection_of_Skylight

7.4 Translucent Surfaces

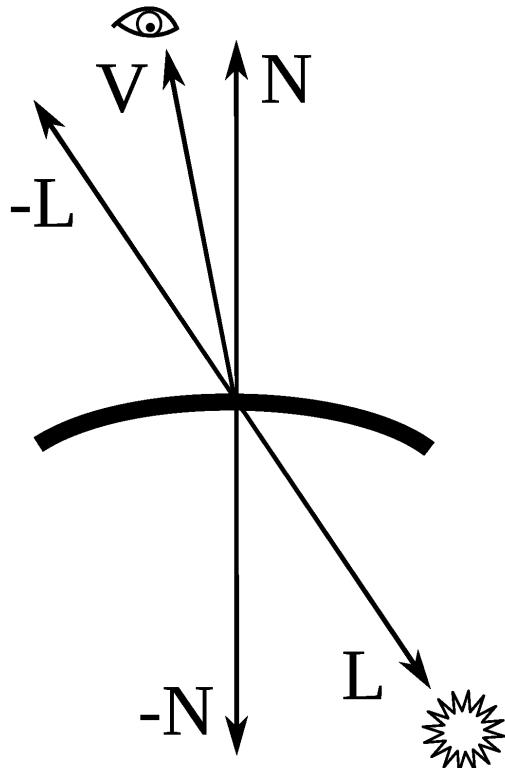


Leaves lit from both sides: note that the missing specular reflection results in a more saturated green of the backlit leaves.

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 Section “Smooth Specular Highlights”. 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.



For translucent illumination, the vector V to the viewer and the vector L to the light source are on opposite sides.

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 Section “Diffuse Reflection”): it only depends on 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.

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 Section

“Specular Highlights” for the equation) except that we replace the reflected light direction \mathbf{R} by the negative light direction $-\mathbf{L}$ 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.

Implementation

The following implementation is based on Section “Smooth Specular Highlights”, 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 using the built-in Cg function `faceforward(n, v, ng)` which returns `n` if `dot(v, ng) < 0` and `-n` otherwise. This method often fails at silhouettes, which results in incorrect lighting for some pixels. An improved version would use different passes and colors for the frontfaces and the backfaces as in Section “Two-Sided Smooth Surfaces”.

In addition to the terms of the Phong reflection model, we also compute illumination by diffuse translucency and forward-scattered translucency with this code:

```
float3 diffuseTranslucency =
    attenuation * float3(_LightColor0)
    * float3(_DiffuseTranslucentColor)
    * max(0.0, dot(lightDirection, -normalDirection));

float3 forwardTranslucency;
if (dot(normalDirection, lightDirection) > 0.0)
    // light source on the wrong side?
{
    forwardTranslucency = float3(0.0, 0.0, 0.0);
    // no forward-scattered translucency
}
else // light source on the right side
{
    forwardTranslucency = attenuation * float3(_LightColor0)
        * float3(_ForwardTranslucentColor) * pow(max(0.0,
            dot(-lightDirection, viewDirection)), _Sharpness);
}
```

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 "Cg 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

        CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;
uniform float4 _DiffuseTranslucentColor;
uniform float4 _ForwardTranslucentColor;
uniform float _Sharpness;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    float3 normalDir : TEXCOORD1;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
```

```
float4x4 modelMatrixInverse = _World2Object;
    // multiplication with unity_Scale.w is unnecessary
    // because we normalize transformed vectors

output.posWorld = mul(modelMatrix, input.vertex);
output.normalDir = normalize(float3(
    mul(float4(input.normal, 0.0), modelMatrixInverse)));
output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);
    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));

    normalDirection = faceforward(normalDirection,
        -viewDirection, normalDirection);
    // flip normal if dot(-viewDirection, normalDirection)>0

    float3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else // point or spot light
    {
        float3 vertexToLightSource =
            float3(_WorldSpaceLightPos0 - input.posWorld);
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
        lightDirection = normalize(vertexToLightSource);
    }

    // Computation of the Phong reflection model:

    float3 ambientLighting =
        float3(UNITY_LIGHTMODEL_AMBIENT) * float3(_Color);

    float3 diffuseReflection =
        attenuation * float3(_LightColor0) * float3(_Color)
        * max(0.0, dot(normalDirection, lightDirection));
```

```
float3 specularReflection;
if (dot(normalDirection, lightDirection) < 0.0)
    // light source on the wrong side?
{
    specularReflection = float3(0.0, 0.0, 0.0);
    // no specular reflection
}
else // light source on the right side
{
    specularReflection = attenuation * float3(_LightColor0)
        * float3(_SpecColor) * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _Shininess);
}

// Computation of the translucent illumination:

float3 diffuseTranslucency =
    attenuation * float3(_LightColor0)
    * float3(_DiffuseTranslucentColor)
    * max(0.0, dot(lightDirection, -normalDirection));

float3 forwardTranslucency;
if (dot(normalDirection, lightDirection) > 0.0)
    // light source on the wrong side?
{
    forwardTranslucency = float3(0.0, 0.0, 0.0);
    // no forward-scattered translucency
}
else // light source on the right side
{
    forwardTranslucency = attenuation * float3(_LightColor0)
        * float3(_ForwardTranslucentColor) * pow(max(0.0,
            dot(-lightDirection, viewDirection)), _Sharpness);
}

// Computation of the complete illumination:

return float4(ambientLighting
    + diffuseReflection + specularReflection
    + diffuseTranslucency + forwardTranslucency, 1.0);
}

ENDCG
}
```

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

    CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;
uniform float4 _DiffuseTranslucentColor;
uniform float4 _ForwardTranslucentColor;
uniform float _Sharpness;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    float3 normalDir : TEXCOORD1;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
```

```
float4x4 modelMatrixInverse = _World2Object;
    // multiplication with unity_Scale.w is unnecessary
    // because we normalize transformed vectors

output.posWorld = mul(modelMatrix, input.vertex);
output.normalDir = normalize(float3(
    mul(float4(input.normal, 0.0), modelMatrixInverse)));
output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);
    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));

    normalDirection = faceforward(normalDirection,
        -viewDirection, normalDirection);
    // flip normal if dot(-viewDirection, normalDirection)>0

    float3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else // point or spot light
    {
        float3 vertexToLightSource =
            float3(_WorldSpaceLightPos0 - input.posWorld);
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
        lightDirection = normalize(vertexToLightSource);
    }

    // Computation of the Phong reflection model:

    float3 ambientLighting =
        float3(UNITY_LIGHTMODEL_AMBIENT) * float3(_Color);

    float3 diffuseReflection =
        attenuation * float3(_LightColor0) * float3(_Color)
        * max(0.0, dot(normalDirection, lightDirection));
```

```
float3 specularReflection;
if (dot(normalDirection, lightDirection) < 0.0)
    // light source on the wrong side?
{
    specularReflection = float3(0.0, 0.0, 0.0);
    // no specular reflection
}
else // light source on the right side
{
    specularReflection = attenuation * float3(_LightColor0)
        * float3(_SpecColor) * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _Shininess);
}

// Computation of the translucent illumination:

float3 diffuseTranslucency =
    attenuation * float3(_LightColor0)
    * float3(_DiffuseTranslucentColor)
    * max(0.0, dot(lightDirection, -normalDirection));

float3 forwardTranslucency;
if (dot(normalDirection, lightDirection) > 0.0)
    // light source on the wrong side?
{
    forwardTranslucency = float3(0.0, 0.0, 0.0);
    // no forward-scattered translucency
}
else // light source on the right side
{
    forwardTranslucency = attenuation * float3(_LightColor0)
        * float3(_ForwardTranslucentColor) * pow(max(0.0,
            dot(-lightDirection, viewDirection)), _Sharpness);
}

// Computation of the complete illumination:

return float4(ambientLighting
    + diffuseReflection + specularReflection
    + diffuseTranslucency + forwardTranslucency, 1.0);
}

ENDCG
}
```

```
}
```

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.

Further Reading

If you still want to know more

- about the diffuse term of the Phong reflection model, you should read Section “Diffuse Reflection”.
- about the ambient or the specular term of the Phong reflection model, you should read Section “Specular Highlights”.
- about per-pixel lighting with the Phong reflection model, you should read Section “Smooth Specular Highlights”.
- about per-pixel lighting of two-sided surfaces, you should read Section “Two-Sided Smooth Surfaces”.

page traffic for 90 days ^[1]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Translucent_Surfaces

7.5 Translucent Bodies



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 Section “Smooth Specular Highlights”. 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 Section “Translucent Surfaces” 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.



Wax idols. Note the reduced contrast of diffuse lighting.

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 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 $\max(0, \mathbf{N} \cdot \mathbf{L})$ (see Section “Diffuse Reflection”), we reduce the influence of this term as the waxiness w increases from 0 to 1. More specifically, we multiply the term $\max(0, \mathbf{N} \cdot \mathbf{L})$ 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:

$$I_{\text{diffuse}} = I_{\text{incoming}} k_{\text{diffuse}} (w + (1 - w) \max(0, \mathbf{N} \cdot \mathbf{L}))$$

with the waxiness w between 0 (i.e. regular diffuse reflection) and 1 (i.e. no dependency on $\mathbf{N} \cdot \mathbf{L}$).

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.



Chessmen in backlight. Note the translucency of the white chessmen.

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 distance between the back and the front actually becomes zero. We could, therefore, use the techniques discussed in Section “Silhouette Enhancement” 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 Section “Mirrors”.)
- 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 $1 - |\mathbf{N} \cdot \mathbf{L}|$, 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.

Implementation

The implementation relies heavily on blending, which is discussed in Section “Transparency”. 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 Cg code is very helpful:

```
Shader "Cg 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)

        CGPROGRAM
        [...]
        ENDCG
    }

    Pass {
        Tags { "LightMode" = "ForwardAdd" }
            // pass for additional light sources on back faces
        Cull Front // render back faces only
        Blend One One // additive blending

        CGPROGRAM
        [...]
        ENDCG
    }

    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

        CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            float4 vert(float4 vertexPos : POSITION) : SV_POSITION
            {
                return mul(UNITY_MATRIX_MVP, vertexPos);
            }

            float4 frag(void) : COLOR
            {
                return float4(0.0, 0.0, 0.0, 0.0);
            }
        
```

```

        ENDCG
    }

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 silhouettleness in fragment's alpha and add colors

    CGPROGRAM
    [...]
    ENDCG
}

Pass {
    Tags { "LightMode" = "ForwardAdd" }
        // pass for additional light sources on front faces
    Cull Back // render front faces only
    Blend One One // additive blending

    CGPROGRAM
    [...]
    ENDCG
}
}

// 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.

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 “silhouettleness” is computed on the back faces as well as on the front faces; however, it is directly multiplied only to the fragment color of the back faces. On the front faces, it is only indirectly multiplied 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 "Cg 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)

        CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float _Waxiness;
uniform float4 _SpecColor;
uniform float _Shininess;
uniform float4 _TranslucentColor;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    float3 normalDir : TEXCOORD1;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;
```

```
float4x4 modelMatrix = _Object2World;
float4x4 modelMatrixInverse = _World2Object;
// multiplication with unity_Scale.w is unnecessary
// because we normalize transformed vectors

output.posWorld = mul(modelMatrix, input.vertex);
output.normalDir = normalize(float3(
    mul(float4(input.normal, 0.0), modelMatrixInverse)));
output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);

    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));
    float3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else // point or spot light
    {
        float3 vertexToLightSource =
            float3(_WorldSpaceLightPos0 - input.posWorld);
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
        lightDirection = normalize(vertexToLightSource);
    }

    float3 ambientLighting = float3(_TranslucentColor)
        * float3(UNITY_LIGHTMODEL_AMBIENT);

    float3 diffuseReflection = float3(_TranslucentColor)
        * attenuation * float3(_LightColor0)
        * max(0.0, dot(normalDirection, lightDirection));

    float silhouetteness =
        1.0 - abs(dot(viewDirection, normalDirection));
}
```

```
        return float4(silhouetteness
                      * (ambientLighting + diffuseReflection), 0.0);
    }

    ENDCG
}

Pass {
    Tags { "LightMode" = "ForwardAdd" }
    // pass for additional light sources on back faces
    Cull Front // render back faces only
    Blend One One // additive blending

    CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float _Waxiness;
uniform float4 _SpecColor;
uniform float _Shininess;
uniform float4 _TranslucentColor;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    float3 normalDir : TEXCOORD1;
```

```
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
    // multiplication with unity_Scale.w is unnecessary
    // because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);

    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));
    float3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else // point or spot light
    {
        float3 vertexToLightSource =
            float3(_WorldSpaceLightPos0 - input.posWorld);
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
        lightDirection = normalize(vertexToLightSource);
    }

    float3 diffuseReflection = float3(_TranslucentColor)
        * attenuation * float3(_LightColor0)
        * max(0.0, dot(normalDirection, lightDirection));
```

```
    float silhouetteness =
        1.0 - abs(dot(viewDirection, normalDirection));

    return float4(silhouetteness * diffuseReflection, 0.0);
}

ENDCG
}

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

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

float4 vert(float4 vertexPos : POSITION) : SV_POSITION
{
    return mul(UNITY_MATRIX_MVP, vertexPos);
}

float4 frag(void) : COLOR
{
    return float4(0.0, 0.0, 0.0, 0.0);
}
ENDCG
}

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

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
```

```
uniform float _Waxiness;
uniform float4 _SpecColor;
uniform float _Shininess;
uniform float4 _TranslucentColor;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
// (all but the bottom-right element have to be scaled
// with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
// position or direction of light source
uniform float4 _LightColor0;
// color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    float3 normalDir : TEXCOORD1;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
// multiplication with unity_Scale.w is unnecessary
// because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);
```

```
float3 viewDirection = normalize(
    _WorldSpaceCameraPos - float3(input.posWorld));
float3 lightDirection;
float attenuation;

if (0.0 == _WorldSpaceLightPos0.w) // directional light?
{
    attenuation = 1.0; // no attenuation
    lightDirection =
        normalize(float3(_WorldSpaceLightPos0));
}
else // point or spot light
{
    float3 vertexToLightSource =
        float3(_WorldSpaceLightPos0 - input.posWorld);
    float distance = length(vertexToLightSource);
    attenuation = 1.0 / distance; // linear attenuation
    lightDirection = normalize(vertexToLightSource);
}

float3 ambientLighting =
    float3(UNITY_LIGHTMODEL_AMBIENT) * float3(_Color);

float3 diffuseReflection =
    attenuation * float3(_LightColor0) * float3(_Color)
    * (_Waxiness + (1.0 - _Waxiness))
    * max(0.0, dot(normalDirection, lightDirection));

float3 specularReflection;
if (dot(normalDirection, lightDirection) < 0.0)
    // light source on the wrong side?
{
    specularReflection = float3(0.0, 0.0, 0.0);
    // no specular reflection
}
else // light source on the right side
{
    specularReflection = attenuation * float3(_LightColor0)
        * float3(_SpecColor) * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _Shininess);
}

float silhouetteness =
    1.0 - abs(dot(viewDirection, normalDirection));
```

```
        return float4(ambientLighting + diffuseReflection
            + specularReflection, silhouetteness);
    }

    ENDCG
}

Pass {
    Tags { "LightMode" = "ForwardAdd" }
        // pass for additional light sources on front faces
    Cull Back // render front faces only
    Blend One One // additive blending

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float _Waxiness;
uniform float4 _SpecColor;
uniform float _Shininess;
uniform float4 _TranslucentColor;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    float3 normalDir : TEXCOORD1;
};
}
```

```
vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
    // multiplication with unity_Scale.w is unnecessary
    // because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);

    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));
    float3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else // point or spot light
    {
        float3 vertexToLightSource =
            float3(_WorldSpaceLightPos0 - input.posWorld);
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
        lightDirection = normalize(vertexToLightSource);
    }

    float3 diffuseReflection =
        attenuation * float3(_LightColor0) * float3(_Color)
        * (_Waxiness + (1.0 - _Waxiness)
        * max(0.0, dot(normalDirection, lightDirection)));

    float3 specularReflection;
```

```

if (dot (normalDirection, lightDirection) < 0.0)
    // light source on the wrong side?
{
    specularReflection = float3(0.0, 0.0, 0.0);
    // no specular reflection
}
else // light source on the right side
{
    specularReflection = attenuation * float3(_LightColor0)
        * float3(_SpecColor) * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _Shininess);
}

float silhouettleness =
    1.0 - abs(dot(viewDirection, normalDirection));

return float4(diffuseReflection
    + specularReflection, silhouettleness);
}

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

```

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.

Further Reading

If you still want to know more

- about translucent surfaces, you should read Section “Translucent Surfaces”.
- about the diffuse term of the Phong reflection model, you should read Section “Diffuse Reflection”.
- about the ambient or the specular term of the Phong reflection model, you should read Section “Specular Highlights”.
- about per-pixel lighting with the Phong reflection model, you should read Section “Smooth Specular Highlights”.
- 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 ^[1].

page traffic for 90 days ^[2]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

- [1] http://http.developer.nvidia.com/GPUGems/gpugems_ch16.html
- [2] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Translucent_Bodies

7.6 Soft Shadows of Spheres

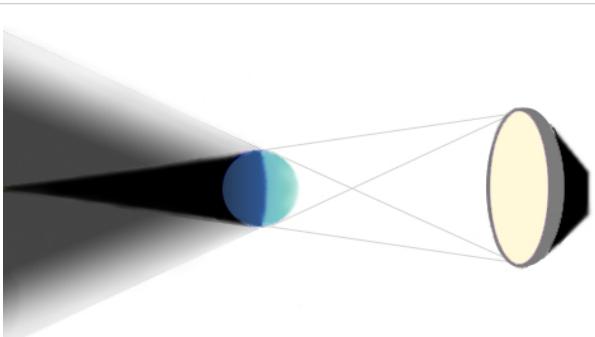


Shadows are not only important to understand the geometry of a scene (e.g. the distances between objects); they can also be quite beautiful.

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). 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, ball models of small nuclei, atoms, or molecules, etc.), or test scenes that can be populated with spheres and benefit from soft shadows.



Umbra (black) and penumbra (gray) are the main parts of soft shadows.

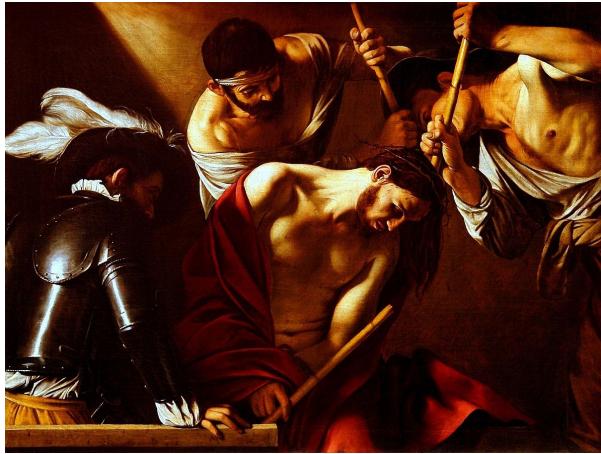
Soft Shadows

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

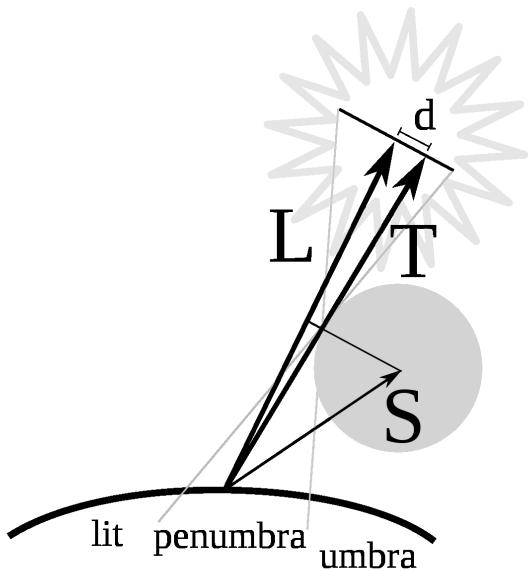
shadow.

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 to the right.



"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.



Vectors for the computation of soft shadows: vector \mathbf{L} to the light source, vector \mathbf{S} to the center of the sphere, tangent vector \mathbf{T} , and distance d of the tangent from the center of the light source.

$d < -r_{\text{light}}$), 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:

$$\angle(\mathbf{L}, \mathbf{T}) \approx \sin \angle(\mathbf{L}, \mathbf{T}) = \frac{d}{|\mathbf{L}|}.$$

Thus, so far we have:

$$d \approx |\mathbf{L}| \angle(\mathbf{L}, \mathbf{T}) = |\mathbf{L}| (\angle(\mathbf{L}, \mathbf{S}) - \angle(\mathbf{T}, \mathbf{S}))$$

We can compute the angle between \mathbf{T} and \mathbf{S} using

Computation

We are going to approximately compute the shadow of a point on a surface when a sphere of radius r_{sphere} at \mathbf{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 to the left.

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 to the left. 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 (

$$\sin \angle(\mathbf{T}, \mathbf{S}) = \frac{r_{\text{sphere}}}{|\mathbf{S}|}.$$

Thus:

$$\angle(\mathbf{T}, \mathbf{S}) = \arcsin \frac{r_{\text{sphere}}}{|\mathbf{S}|}.$$

For the angle between \mathbf{L} and \mathbf{S} we use a feature of the cross product:

$$|\mathbf{a} \times \mathbf{b}| = |\mathbf{a}| |\mathbf{b}| \sin \angle(\mathbf{a}, \mathbf{b}).$$

Therefore:

$$\angle(\mathbf{L}, \mathbf{S}) = \arcsin \frac{|\mathbf{L} \times \mathbf{S}|}{|\mathbf{L}| |\mathbf{S}|}.$$

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 Cg function `smoothstep(a, b, x) = t*t*(3-2*t)` with $t=\text{clamp}((x-a)/(b-a), 0, 1)$:

$$w = \text{smoothstep}\left(-1, 1, \frac{-d}{r_{\text{light}}}\right)$$

While this isn't a particularly 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:

$$w'' = w' \text{smoothstep}(0, r_{\text{sphere}}, |\mathbf{L}| - |\mathbf{S}|)$$

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).

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
float3 sphereDirection =
    float3(_SpherePosition - input.posWorld);
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.

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 ($1 \text{ rad} = 180^\circ / \pi$). 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 `_SpherePosition` and `_SphereRadius` of the following shader (which should be attached to the same shadow-receiving object as the script).

The fragment shader is quite long and in fact we have to use the line `#pragma target 3.0` to ignore some restrictions of older GPUs as documented in the Unity reference^[1].

```
Shader "Cg 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

            CGPROGRAM

#pragma vertex vert
#pragma fragment frag

#pragma target 3.0

// User-specified properties
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;
uniform float4 _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 (apart from _LightColor0)
        // are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
        // (all but the bottom-right element have to be scaled
        // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
        // position or direction of light source
uniform float4 _LightColor0;
        // color of light source (from "Lighting.cginc")
```

```
struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    float3 normalDir : TEXCOORD1;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
    // multiplication with unity_Scale.w is unnecessary
    // because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);

    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));
    float3 lightDirection;
    float lightDistance;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
        lightDistance = 1.0;
    }
    else // point or spot light
    {
        lightDirection = float3(
            _WorldSpaceLightPos0 - input.posWorld);
```

```
    lightDistance = length(lightDirection);
    attenuation = 1.0 / lightDistance; // linear attenuation
    lightDirection = lightDirection / lightDistance;
}

// computation of level of shadowing w
float3 sphereDirection = float3(
    _SpherePosition - input.posWorld);
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);
}

float3 ambientLighting =
    float3(UNITY_LIGHTMODEL_AMBIENT) * float3(_Color);

float3 diffuseReflection =
    attenuation * float3(_LightColor0) * float3(_Color)
    * max(0.0, dot(normalDirection, lightDirection));

float3 specularReflection;
if (dot(normalDirection, lightDirection) < 0.0)
    // light source on the wrong side?
{
    specularReflection = float3(0.0, 0.0, 0.0);
    // no specular reflection
}
else // light source on the right side
{
    specularReflection = attenuation * float3(_LightColor0)
        * float3(_SpecColor) * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _Shininess);
}

return float4(ambientLighting
    + (1.0 - w) * (diffuseReflection + specularReflection),
```

```
    1.0);
}

ENDCG
}

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

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

#pragma target 3.0

// User-specified properties
uniform float4 _Color;
uniform float4 _SpecColor;
uniform float _Shininess;
uniform float4 _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 (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};
struct vertexOutput {
```

```
float4 pos : SV_POSITION;
float4 posWorld : TEXCOORD0;
float3 normalDir : TEXCOORD1;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
    // multiplication with unity_Scale.w is unnecessary
    // because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);

    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));
    float3 lightDirection;
    float lightDistance;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
        lightDistance = 1.0;
    }
    else // point or spot light
    {
        lightDirection = float3(
            _WorldSpaceLightPos0 - input.posWorld);
        lightDistance = length(lightDirection);
        attenuation = 1.0 / lightDistance; // linear attenuation
        lightDirection = lightDirection / lightDistance;
    }
}
```

```
// computation of level of shadowing w
float3 sphereDirection = float3(
    _SpherePosition - input.posWorld);
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);
}

float3 diffuseReflection =
    attenuation * float3(_LightColor0) * float3(_Color)
    * max(0.0, dot(normalDirection, lightDirection));

float3 specularReflection;
if (dot(normalDirection, lightDirection) < 0.0)
    // light source on the wrong side?
{
    specularReflection = float3(0.0, 0.0, 0.0);
    // no specular reflection
}
else // light source on the right side
{
    specularReflection = attenuation * float3(_LightColor0)
        * float3(_SpecColor) * pow(max(0.0, dot(
            reflect(-lightDirection, normalDirection),
            viewDirection)), _Shininess);
}

return float4((1.0 - w) * (diffuseReflection
    + specularReflection), 1.0);
}

ENDCG
}
}

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

```
}
```

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.

Further Reading

If you still want to know more

- about the rest of the shader code, you should read Section “Smooth Specular Highlights”.
- 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 [1].

page traffic for 90 days [2]

< Cg Programming/Unity

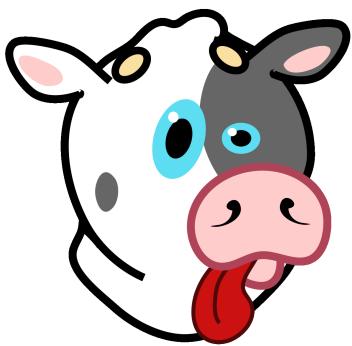
Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] http://www.cs.uaf.edu/~olawlor/papers/2006/shadow/lawlor_shadow_2006.pdf

[2] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Soft_Shadows_of_Spheres

7.7 Toon Shading



A cow smiley. All images in this section are by Mariana Ruiz Villarreal (a.k.a. LadyofHats).

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 Section “Smooth Specular Highlights”. 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.

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.



A goat smiley.

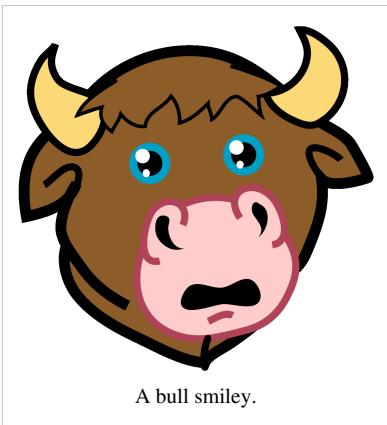
Shaders for a Specific Visual Style

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.



Stylized Specular Highlights

In comparison to the Phong reflection model that was implemented in Section “Smooth Specular Highlights”, 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.

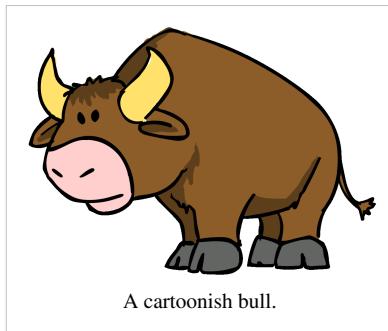
We can implement this kind of stylized specular highlights by computing the specular reflection term of the Phong shading model and setting the fragment color to the specular reflection color times the (unattenuated) color of the light source if the specular reflection term is greater than a certain 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 was described in Section “Transparency”. 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` 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
    * float3(_LightColor0) * float3(_SpecColor)
    + (1.0 - _SpecColor.a) * fragmentColor;
}
```

Is this sufficient? If you look closely at the eyes of the bull in the image to the left, 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 Section “Transparency” for a description of alpha blending.)



Stylized Diffuse Illumination

The diffuse illumination in the image of the bull to the left 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.

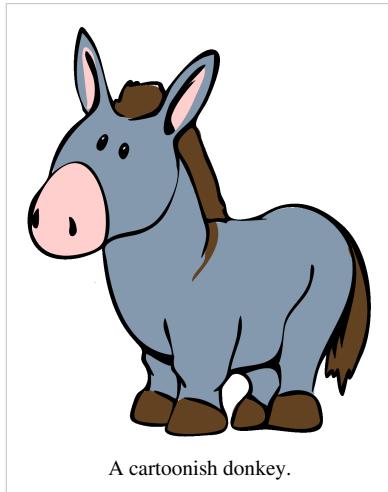
One way to implement this, is to use the full diffuse reflection color whenever the diffuse reflection term of the Phong reflection model reaches a certain threshold, e.g. greater than 0, and a second color otherwise. For the fur of the bull, these two colors would be different; for the other parts, 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:

```
float3 fragmentColor = float3(_UnlitColor);

if (attenuation
    * max(0.0, dot(normalDirection, lightDirection))
    >= _DiffuseThreshold)
{
    fragmentColor = float3(_LightColor0) * float3(_Color);
}
```

Is this all there is to say about the stylized diffuse illumination in the image to the left? 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 this convincingly in a shader.



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 Section "Silhouette Enhancement": 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). For a linear interpolation from a value `a` to another value `b` with a parameter `x` between 0 and 1, Cg offers the built-in function `lerp(a, b, x)`. The 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)
    < lerp(_UnlitOutlineThickness, _LitOutlineThickness,
          max(0.0, dot(normalDirection, lightDirection))))
{
    fragmentColor =
        float3(_LightColor0) * float3(_OutlineColor);
}
```

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 "Cg 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

        CGPROGRAM
        #pragma vertex vert
```

```
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float4 _UnlitColor;
uniform float _DiffuseThreshold;
uniform float4 _OutlineColor;
uniform float _LitOutlineThickness;
uniform float _UnlitOutlineThickness;
uniform float4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
// (all but the bottom-right element have to be scaled
// with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
// position or direction of light source
uniform float4 _LightColor0;
// color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    float3 normalDir : TEXCOORD1;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
// multiplication with unity_Scale.w is unnecessary
// because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
}
```

```
        output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
        return output;
    }

    float4 frag(vertexOutput input) : COLOR
    {
        float3 normalDirection = normalize(input.normalDir);

        float3 viewDirection = normalize(
            _WorldSpaceCameraPos - float3(input.posWorld));
        float3 lightDirection;
        float attenuation;

        if (0.0 == _WorldSpaceLightPos0.w) // directional light?
        {
            attenuation = 1.0; // no attenuation
            lightDirection =
                normalize(float3(_WorldSpaceLightPos0));
        }
        else // point or spot light
        {
            float3 vertexToLightSource =
                float3(_WorldSpaceLightPos0 - input.posWorld);
            float distance = length(vertexToLightSource);
            attenuation = 1.0 / distance; // linear attenuation
            lightDirection = normalize(vertexToLightSource);
        }

        // default: unlit
        float3 fragmentColor = float3(_UnlitColor);

        // low priority: diffuse illumination
        if (attenuation
            * max(0.0, dot(normalDirection, lightDirection))
            >= _DiffuseThreshold)
        {
            fragmentColor = float3(_LightColor0) * float3(_Color);
        }

        // higher priority: outline
        if (dot(viewDirection, normalDirection)
            < lerp(_UnlitOutlineThickness, _LitOutlineThickness,
            max(0.0, dot(normalDirection, lightDirection))))
        {
            fragmentColor =
                float3(_LightColor0) * float3(_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
    * float3(_LightColor0) * float3(_SpecColor)
    + (1.0 - _SpecColor.a) * fragmentColor;
}

return float4(fragmentColor, 1.0);
}

ENDCG
}

Pass {
    Tags { "LightMode" = "ForwardAdd" }
    // pass for additional light sources
    Blend SrcAlpha OneMinusSrcAlpha
    // blend specular highlights over framebuffer

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified properties
uniform float4 _Color;
uniform float4 _UnlitColor;
uniform float _DiffuseThreshold;
uniform float4 _OutlineColor;
uniform float _LitOutlineThickness;
uniform float _UnlitOutlineThickness;
uniform float4 _SpecColor;
uniform float _Shininess;

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
```

```
// (all but the bottom-right element have to be scaled
// with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source
uniform float4 _LightColor0;
    // color of light source (from "Lighting.cginc")

struct vertexInput {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 posWorld : TEXCOORD0;
    float3 normalDir : TEXCOORD1;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse = _World2Object;
        // multiplication with unity_Scale.w is unnecessary
        // because we normalize transformed vectors

    output.posWorld = mul(modelMatrix, input.vertex);
    output.normalDir = normalize(float3(
        mul(float4(input.normal, 0.0), modelMatrixInverse)));
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}

float4 frag(vertexOutput input) : COLOR
{
    float3 normalDirection = normalize(input.normalDir);

    float3 viewDirection = normalize(
        _WorldSpaceCameraPos - float3(input.posWorld));
    float3 lightDirection;
    float attenuation;

    if (0.0 == _WorldSpaceLightPos0.w) // directional light?
    {
        attenuation = 1.0; // no attenuation
        lightDirection =
            normalize(float3(_WorldSpaceLightPos0));
    }
    else
        attenuation = 1.0 / dot(-input.normalDir,
            float3(_WorldSpaceLightPos0));
        lightDirection =
            normalize(_WorldSpaceLightPos0 -
                float3(input.posWorld));
}
```

```
    }

    else // point or spot light
    {
        float3 vertexToLightSource =
            float3(_WorldSpaceLightPos0 - input.posWorld);
        float distance = length(vertexToLightSource);
        attenuation = 1.0 / distance; // linear attenuation
        lightDirection = normalize(vertexToLightSource);
    }

    float4 fragmentColor = float4(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 =
            float4(_LightColor0.rgb, 1.0) * _SpecColor;
    }

    return fragmentColor;
}

ENDCG
}
}

// 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.

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.

Further Reading

If you still want to know more

- about the Phong reflection model and the per-pixel lighting, you should read Section “Smooth Specular Highlights”.
- about the computation of silhouettes, you should read Section “Silhouette Enhancement”.
- about blending, you should read Section “Transparency”.
- 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.

page traffic for 90 days [1]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Toon_Shading

8 Non-Standard Vertex Transformations

8.1 Screen Overlays



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 Section “Vertex Transformations”. This particular tutorial uses texturing as described in Section “Textured Spheres” and blending as described in Section “Transparency”.

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 to the left, but also other GUI (graphical user 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.)

Simulating GUI Textures with a Cg 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 float4 _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 float4 _ScreenParams; // x = width; y = height;
// z = 1 + 1.0/width; w = 1 + 1.0/height
...
vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float2 rasterPosition = float2(
        _X + _ScreenParams.x / 2.0
        + _Width * (input.vertex.x + 0.5),
        _Y + _ScreenParams.y / 2.0
        + _Height * (input.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 $\text{float2}(_X + _ScreenParams.x / 2.0, _Y + _ScreenParams.y / 2.0)$, 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 $\text{float2}(_X + _ScreenParams.x / 2.0 + _Width, _Y + _ScreenParams.y / 2.0 + _Height)$. Raster positions are convenient and, in fact, they are often used in OpenGL; however, they are not quite what we need here.

The output parameter of the vertex shader is in the so-called “clip space” as discussed in Section “Vertex Transformations”. The GPU transforms these coordinates to normalized device coordinates between -1 and 1 by dividing them by the fourth coordinate 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 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 as vertex output parameter, we have to specify it in this coordinate system. Fortunately, transforming a raster position to normalized device coordinates is not too difficult:

```
output.pos = float4(
    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);
```

As you can easily check, this transforms the raster position `float2(0, 0)` to normalized device coordinates $(-1.0, -1.0)$ and the raster position `float2(_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 $+0.5$:

```
output.tex = float4(input.vertex.x + 0.5,
                     input.vertex.y + 0.5, 0.0, 0.0);
// for a cube, vertex.x and vertex.y
// are -0.5 or 0.5
```

With the vertex output parameter `tex`, 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`:

```
float4 frag(vertexOutput input) : COLOR
{
    return _Color
    * tex2D(_MainTex, float2(input.tex));
}
```

That's it.

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 Section “Transparency”) to allow for transparent textures. It also deactivates the depth test to make sure that the texture is never occluded:

```
Shader "Cg shader for screen overlays"
{
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

        CGPROGRAM
        #pragma vertex vert
```

```
#pragma fragment frag

// User-specified uniforms
uniform sampler2D _MainTex;
uniform float4 _Color;
uniform float _X;
uniform float _Y;
uniform float _Width;
uniform float _Height;

// The following built-in uniforms are also defined
// in "UnityCG.cginc", which could be #included
uniform float4 _ScreenParams; // x = width; y = height;
// z = 1 + 1.0/width; w = 1 + 1.0/height

struct vertexInput {
    float4 vertex : POSITION;
    float4 texcoord : TEXCOORD0;
};

struct vertexOutput {
    float4 pos : SV_POSITION;
    float4 tex : TEXCOORD0;
};

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    float2 rasterPosition = float2(
        _X + _ScreenParams.x / 2.0
        + _Width * (input.vertex.x + 0.5),
        _Y + _ScreenParams.y / 2.0
        + _Height * (input.vertex.y + 0.5));
    output.pos = float4(
        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);

    output.tex = float4(input.vertex.x + 0.5,
        input.vertex.y + 0.5, 0.0, 0.0);
// for a cube, vertex.x and vertex.y
// are -0.5 or 0.5
    return output;
}
```

```
float4 frag(vertexOutput input) : COLOR
{
    return _Color
    * tex2D(_MainTex, float2(input.tex));
}

ENDCG
}
}

}
```

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.)

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.

Summary

Congratulation, you have reached the end of another tutorial. We have seen:

- How to simulate GUI textures with a Cg shader.
- How to modify the shader for opaque screen overlays.

Further Reading

If you still want to know more

- about texturing, you should read Section “Textured Spheres”.
- about blending, you should read Section “Transparency”.
- about object space, screen space, clip space, normalized device coordinates, perspective division, etc., you should read the description of the standard vertex transformations in Section “Vertex Transformations”.

page traffic for 90 days ^[1]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Screen_Overlays

8.2 Billboards



Billboards along a highway. Note the orientation of the billboards for best visibility.

This tutorial introduces **billboards**.

It is based on Section “Textured Spheres” and the discussion in Section “Vertex Transformations”.

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 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.

Vertex Transformation for Billboards

Similarly to Section “Skyboxes”, 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 model-view transformation `UNITY_MATRIX_MV`. (In homogeneous coordinates all points have a 1 as fourth coordinate; see the discussion in Section “Vertex Transformations”.) View space is just a rotated version of world space with the xy plane parallel to the view plane as discussed in Section “Vertex Transformations”. Thus, this is the correct space to construct an appropriately rotated billboard. We add the x and y object coordinates (`vertex.x` and `vertex.y`) to the transformed origin in view coordinates and then transform the result with the projection matrix `UNITY_MATRIX_P`:

```
output.pos = mul(UNITY_MATRIX_P,
    mul(UNITY_MATRIX_MV, float4(0.0, 0.0, 0.0, 1.0))
    + float4(input.vertex.x, input.vertex.y, 0.0, 0.0));
```

Apart from this, we only have to compute texture coordinates, which is done the same way as in Section “Screen Overlays”:

```
output.tex = float4(input.vertex.x + 0.5,
    input.vertex.y + 0.5, 0.0, 0.0);
```

Then the fragment shader just looks up the color at the interpolated texture coordinates.

Complete Shader Code

The complete shader code for the standard cube object is now:

```
Shader "Cg shader for billboards" {
    Properties {
        _MainTex ("Texture Image", 2D) = "white" {}
    }
    SubShader {
        Pass {
            CGPROGRAM

                #pragma vertex vert
                #pragma fragment frag

                // User-specified uniforms
                uniform sampler2D _MainTex;

                struct vertexInput {
                    float4 vertex : POSITION;
                };
                struct vertexOutput {
                    float4 pos : SV_POSITION;
                    float4 tex : TEXCOORD0;
                };

                vertexOutput vert(vertexInput input)
                {
                    vertexOutput output;

                    output.pos = mul(UNITY_MATRIX_P,
                        mul(UNITY_MATRIX_MV, float4(0.0, 0.0, 0.0, 1.0))
                        + float4(input.vertex.x, input.vertex.y, 0.0, 0.0));

                    output.tex = float4(input.vertex.x + 0.5,
                        input.vertex.y + 0.5, 0.0, 0.0);

                    return output;
                }

                float4 frag(vertexOutput input) : COLOR
                {
                    return tex2D(_MainTex, float2(input.tex));
                }

            ENDCG
        }
    }
}
```

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.

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.

Further Reading

If you still want to learn more

- about object space, world space, view space, the model view matrix and the projection matrix, you should read the description in Section “Vertex Transformations”.
- about texturing, you should read Section “Textured Spheres”.

page traffic for 90 days ^[1]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Billboards

8.3 Nonlinear Deformations

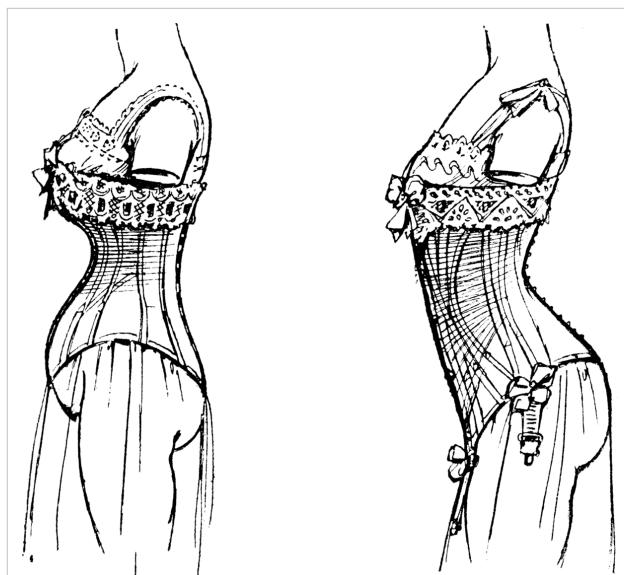


Illustration of deformations by corsets. These are examples of nonlinear deformations that cannot be modeled by a linear transformation.

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 Section “Vertex Transformations” is very useful.

Blending between Two Model Transformations

Most deformations of meshes cannot be modeled by the affine transformations with 4×4 matrices that are discussed in Section “Vertex Transformations”. 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 `float4x4 _Trafo0` and `float4x4 _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 us 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;
}
}

```

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 `input.vertex.z + 0.5`:

```
float weight0 = input.vertex.z + 0.5;
```

and the other weight is $1.0 - \text{weight0}$. Thus, the part with positive `input.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:

```

float4 blendedVertex =
    weight0 * mul(_Trafo0, input.vertex)
    + (1.0 - weight0) * mul(_Trafo1, input.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):

```

float4x4 modelMatrixInverse =
    _World2Object * unity_Scale.w;
modelMatrixInverse[3][3] = 1.0;
float4x4 viewMatrix =
    mul(UNITY_MATRIX_MV, modelMatrixInverse);
output.pos =
    mul(UNITY_MATRIX_P, mul(viewMatrix, blendedVertex));

```

In order to illustrate the different weights, we visualize `weight0` by the red component and $1.0 - \text{weight0}$ by the green component of a color (which is set in the fragment shader):

```
output.col = float4(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.

Complete Shader Code

All in all, the shader code looks like this:

```
Shader "Cg shader for vertex blending" {
    SubShader {
        Pass {
            CGPROGRAM

                #pragma vertex vert
                #pragma fragment frag

                // Uniforms set by a script
                uniform float4x4 _Trafo0; // model transformation of bone0
                uniform float4x4 _Trafol; // model transformation of bone1

                // The following built-in uniforms (apart from _LightColor0)
                // are defined in "UnityCG.cginc", which could be #included
                uniform float4 unity_Scale; // w = 1/scale; see _World2Object
                uniform float3 _WorldSpaceCameraPos;
                uniform float4x4 _Object2World; // model matrix
                uniform float4x4 _World2Object; // inverse model matrix
                // (all but the bottom-right element have to be scaled
                // with unity_Scale.w if scaling is important)

                struct vertexInput {
                    float4 vertex : POSITION;
                };
                struct vertexOutput {
                    float4 pos : SV_POSITION;
                    float4 col : COLOR;
                };

                vertexOutput vert(vertexInput input)
                {
                    vertexOutput output;

                    float weight0 = input.vertex.z + 0.5;
                    // depends on the mesh
                    float4 blendedVertex =
                        weight0 * mul(_Trafo0, input.vertex)
                        + (1.0 - weight0) * mul(_Trafol, input.vertex);

                    float4x4 modelMatrixInverse =
                        _World2Object * unity_Scale.w;
                    modelMatrixInverse[3][3] = 1.0;
                    float4x4 viewMatrix =
                        mul(UNITY_MATRIX_MV, modelMatrixInverse);
                    output.pos =
```

```
mul(UNITY_MATRIX_P, mul(viewMatrix, blendedVertex));  
  
output.col = float4(weight0, 1.0 - weight0, 0.0, 1.0);  
    // visualize weight0 as red and weight1 as green  
return output;  
}  
  
float4 frag(vertexOutput input) : COLOR  
{  
    return input.col;  
}  
  
ENDCG  
}  
}  
}
```

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.

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.

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 Section “Vertex Transformations”.
- 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.

page traffic for 90 days ^[1]

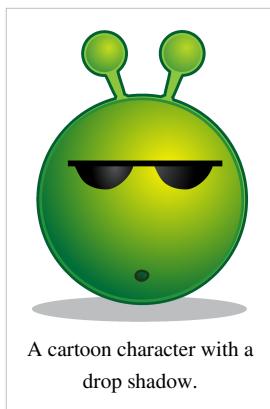
< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Nonlinear_Deformations

8.4 Shadows on Planes



This tutorial covers the **projection of shadows onto planes**.

It is not based on any particular tutorial; however, some understanding of Section “Vertex Transformations” is useful.

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 Section “Soft Shadows of Spheres”) 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.

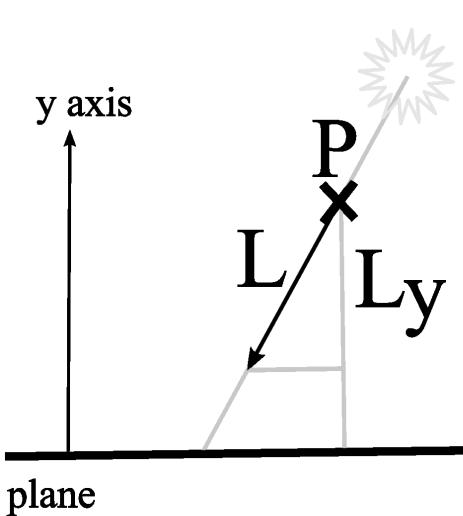


Illustration of the projection of a point **P** in direction **L** onto a plane in the coordinate system of the plane.

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 Section “Vertex Transformations”. 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** 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:

```
uniform float4x4 _World2Receiver; // transformation from
// world coordinates to the coordinate system of the plane

[...]

float4 vert(float4 vertexPos : POSITION) : SV_POSITION
```

```

{
    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse =
        _World2Object * unity_Scale.w;
    modelMatrixInverse[3][3] = 1.0;
    float4x4 viewMatrix =
        mul(UNITY_MATRIX_MV, modelMatrixInverse);

    float4 lightDirection;
    if (0.0 != _WorldSpaceLightPos0.w)
    {
        // point or spot light
        lightDirection = normalize(
            mul(modelMatrix, vertexPos - _WorldSpaceLightPos0));
    }
    else
    {
        // directional light
        lightDirection = -normalize(_WorldSpaceLightPos0);
    }

    float4 vertexInWorldSpace = mul(modelMatrix, vertexPos);
    float4 world2ReceiverRow1 =
        float4(_World2Receiver[0][1], _World2Receiver[1][1],
        _World2Receiver[2][1], _World2Receiver[3][1]);
    float distanceOfVertex =
        mul(_World2Receiver, vertexInWorldSpace).y
        // = height over plane
    float lengthOfLightDirectionInY =
        mul(_World2Receiver, lightDirection).y
        // = length in y direction

    lightDirection = lightDirection
        * (distanceOfVertex / (-lengthOfLightDirectionInY));

    return mul(UNITY_MATRIX_P, mul(viewMatrix,
        vertexInWorldSpace + lightDirection));
}

```

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.

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 "Cg 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

            CGPROGRAM

                #pragma vertex vert
                #pragma fragment frag

                // User-specified properties
                uniform float4 _Color;

                float4 vert(float4 vertexPos : POSITION) : SV_POSITION
                {
                    return mul(UNITY_MATRIX_MVP, vertexPos);
                }

                float4 frag(void) : COLOR
                {
                    return _Color;
                }
            ENDCG
        }
    }
}
```

```
}

ENDCG

}

Pass {
    Tags { "LightMode" = "ForwardBase" }
        // rendering of projected shadow
    Offset -1.0, -2.0
        // make sure shadow polygons are on top of shadow receiver

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified uniforms
uniform float4 _ShadowColor;
uniform float4x4 _World2Receiver; // transformation from
    // world coordinates to the coordinate system of the plane

// The following built-in uniforms (apart from _LightColor0)
// are defined in "UnityCG.cginc", which could be #included
uniform float4 unity_Scale; // w = 1/scale; see _World2Object
uniform float3 _WorldSpaceCameraPos;
uniform float4x4 _Object2World; // model matrix
uniform float4x4 _World2Object; // inverse model matrix
    // (all but the bottom-right element have to be scaled
    // with unity_Scale.w if scaling is important)
uniform float4 _WorldSpaceLightPos0;
    // position or direction of light source

float4 vert(float4 vertexPos : POSITION) : SV_POSITION
{
    float4x4 modelMatrix = _Object2World;
    float4x4 modelMatrixInverse =
        _World2Object * unity_Scale.w;
    modelMatrixInverse[3][3] = 1.0;
    float4x4 viewMatrix =
        mul(UNITY_MATRIX_MV, modelMatrixInverse);

    float4 lightDirection;
    if (0.0 != _WorldSpaceLightPos0.w)
    {
        // point or spot light
        lightDirection = normalize(
            mul(modelMatrix, vertexPos - _WorldSpaceLightPos0));
    }
}
```

```
        }

    else
    {
        // directional light
        lightDirection = -normalize(_WorldSpaceLightPos0);
    }

    float4 vertexInWorldSpace = mul(modelMatrix, vertexPos);
    float4 world2ReceiverRow1 =
        float4(_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 = float4(0.0, 0.0, 0.0, 0.0);
        // don't move vertex
    }

    return mul(UNITY_MATRIX_P, mul(viewMatrix,
        vertexInWorldSpace + lightDirection));
}

float4 frag(void) : COLOR
{
    return _ShadowColor;
}

ENDCG
}
}
```

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 Section “Cutaways”.
- 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 Section “Silhouette Enhancement”.

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.

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 Section “Vertex Transformations”.
- about setting up a projection matrix to project the shadow, you could read Section 9.4.1 of the SIGGRAPH '98 Course “Advanced Graphics Programming Techniques Using OpenGL” organized by Tom McReynolds, which is available online^[1].

page traffic for 90 days^[2]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

- [1] <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.165.9026&rep=rep1&type=pdf>
- [2] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Shadows_on_Planes

8.5 Mirrors



"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 Section “Transparency” and requires some understanding of Section “Vertex Transformations”.

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.

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 Velázquez) nor for semitransparent mirrors, for example glass windows.

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.

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 Section "Translucent Bodies"). 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 {
    CGPROGRAM

        #pragma vertex vert
        #pragma fragment frag

        float4 vert(float4 vertexPos : POSITION) : SV_POSITION
        {
            return mul(UNITY_MATRIX_MVP, vertexPos);
        }

        float4 frag(void) : COLOR
        {
            return float4(1.0, 0.0, 0.0, 0.0);
            // this color should never be visible,
            // only alpha is important
        }
    ENDCG
}
```

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 Section "Transparency"):

Blend OneMinusDstAlpha DstAlpha

We can think of the blend equation as:

```
float4 result = float4(1.0 - pixel_color.a) * fragment_output +
float4(pixel_color.a) * pixel_color;
```

where `pixel_color` is the color of a pixel in the framebuffer and `fragment_output` is the color output parameter of the fragment shader. Let's see what the expression is for `pixel_color.a` equal to 1 (i.e. outside of the visible part of the mirror):

```
float4(1.0 - 1.0) * fragment_output + float4(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)?

```
float4(1.0 - 0.0) * fragment_output + float4(0.0) * pixel_color == fragment_output
```

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 fragment output color 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 Section "Per-Fragment Operations"). 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 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 Section "Vertex Transformations", the output position of the vertex shader in is divided automatically by its fourth coordinate `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, we have to set the `z` coordinate to the `w` coordinate 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

    CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag

        uniform float4 _Color;
        // user-specified background color in the mirror

        float4 vert(float4 vertexPos : POSITION) : SV_POSITION
        {
            float4 pos = mul(UNITY_MATRIX_MVP, vertexPos);
            pos.z = pos.w;
            // the perspective division will divide pos.z
            // by pos.w; thus, the depth is 1.0,
            // which represents the far clipping plane
            return pos;
        }
    } // end CGPROGRAM
}
```

```

    }

    float4 frag(void) : COLOR
    {
        return float4(_Color.rgb, 0.0);
        // set alpha to 0.0 and
        // the color to the user-specified background color
    }
    ENDCG
}

```

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 Section “Skyboxes”.)

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 "Cg 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 {
        CGPROGRAM

        #pragma vertex vert
        #pragma fragment frag

        float4 vert(float4 vertexPos : POSITION) : SV_POSITION
        {
            return mul(UNITY_MATRIX_MVP, vertexPos);
        }

        float4 frag(void) : COLOR
        {
            return float4(1.0, 0.0, 0.0, 0.0);
            // this color should never be visible,
            // only alpha is important
        }
    ENDCG
}

```

```
// 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

    CGPROGRAM

#pragma vertex vert
#pragma fragment frag

uniform float4 _Color;
    // user-specified background color in the mirror

float4 vert(float4 vertexPos : POSITION) : SV_POSITION
{
    float4 pos = mul(UNITY_MATRIX_MVP, vertexPos);
    pos.z = pos.w;
        // the perspective division will divide pos.z
        // by pos.w; thus, the depth is 1.0,
        // which represents the far clipping plane
    return pos;
}

float4 frag(void) : COLOR
{
    return float4(_Color.rgb, 0.0);
    // set alpha to 0.0 and
    // the color to the user-specified background color
}
ENDCG
}
}
```



A water lily in Sheffield Park.

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 Section "Cutaways") 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 "Cg 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
    }
}

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

// User-specified uniforms
uniform float4 _Color;
uniform float4x4 _WorldToMirror; // set by a script

// The following built-in uniform is also defined
// in "UnityCG.cginc", which could be #included
uniform float4x4 _Object2World; // model matrix

struct vertexInput {
    float4 vertex : POSITION;
};
struct vertexOutput {
```

```

        float4 pos : SV_POSITION;
        float4 posInMirror : TEXCOORD0;
    };

    vertexOutput vert(vertexInput input)
    {
        vertexOutput output;

        output.posInMirror = mul(_WorldToMirror,
            mul(_Object2World, input.vertex));
        output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
        return output;
    }

    float4 frag(vertexOutput input) : COLOR
    {
        if (input.posInMirror.y > 0.0)
            // reflection comes out of mirror?
        {
            discard; // don't rasterize it
        }
        return float4(_Color.rgb, 0.0); // set alpha to 0.0
    }

    ENDCG
}
}
}

```

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.

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.

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.

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 '98 Course "Advanced Graphics Programming Techniques Using OpenGL" organized by Tom McReynolds, which is available online^[1].

page traffic for 90 days^[1]

< Cg Programming/Unity

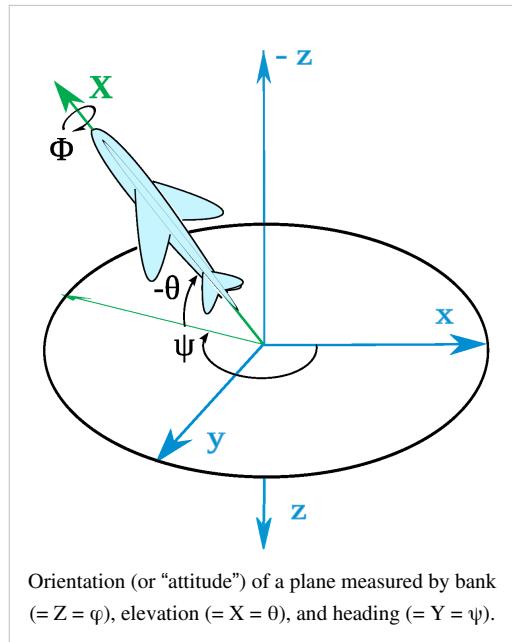
Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Mirrors

9 Graphics without Shaders

9.1 Rotations



This tutorial discusses different representations of local rotations in Unity. It is based on Section “Vertex Transformations”.

This is the first of a few tutorials that demonstrate useful functionality of Unity that is not directly associated with shaders. Therefore, the presented code is in JavaScript.

Unity's Euler Angles

If you create a GameObject and select it, the **Inspector** will show three values X, Y, Z under **Transform > Rotation**. These are three Euler angles measured in degrees. (More precisely spoken, they are one possible set of three Tait-Bryan angles, or nautical angles or Cardan angles, since they describe rotations about three different axes while “proper Euler angles” would describe a set of three angles that describe three rotations where two rotations are about the same axis.)

In JavaScript you can access these three angles as the `Vector3` variable `Transform.localEulerAngles`^[1]. The documentation states that the angles represent — in this order — a rotation by Z degrees about the z axis, X degrees about the x axis, and Y degrees about the y axis. More precisely spoken, these are rotations about the fixed(!) axes of the parent object (or the world axes if there is no parent object). Since these rotations use fixed axes, they are also called “extrinsic rotations”.

These three angles can describe any rotation of an object in three dimensions. In the case of `Transform.localEulerAngles` they actually describe the orientation of an object relative to the parent's coordinate system (or the world's coordinate system if there is no parent). Rotation in the sense of a rotating motion is discussed below.

In aviation, Euler angles are used when the orientation of a plane is specified relatively to the fixed axes of the ground (e.g. a tower of an airport) as illustrated in the figure above. In this case, they are called “bank” (corresponding to Z), “elevation” (corresponding to X), and “heading” (corresponding to Y). The following JavaScript code can be attached to an object to set the Euler angles in terms of these names. (In the **Project** view select **Create > Javascript**, double-click it to open it, copy & paste the code from below into the script, and drag the script from the **Project** view over the game object in the **Hierarchy** view, then select the game object and find the public variables of the script in the **Inspector**.)

```
@script ExecuteInEditMode()
#pragma strict

public var bankZ : float;
public var elevationX : float;
public var headingY : float;
```

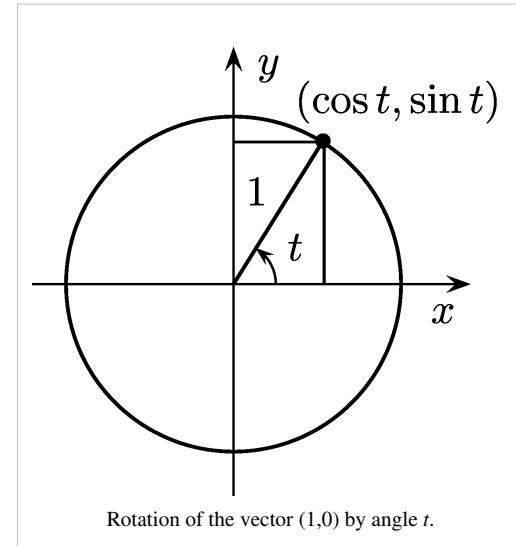
```
function Update()
{
    transform.localEulerAngles =
        Vector3(elevationX, headingY, bankZ);
}
```

Interact with the three variables in the **Inspector** to get familiar with their meaning. You could start with all three angles set to 0. Then change bank, elevation, and heading (in this order) and observe how the object rotates about the parent's (or world's) z axis (the blue axis in Unity) when changing bank, about the parent's x axis (red) when changing elevation, and about the parent's y axis (green) when changing heading.

Computing the Rotation Matrix

In order to establish the connection with the elementary model matrices in Section “Vertex Transformations”, this subsection presents how to compute a rotation matrix from the three angles X, Y, Z (or θ , ψ , and φ).

The 4×4 rotation matrix $R_z(\varphi)$ for a rotation by an angle φ ($= Z$) about the z axis is:



$$R_z(\varphi) = \begin{bmatrix} \cos \varphi & -\sin \varphi & 0 & 0 \\ \sin \varphi & \cos \varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The first two coordinates of the leftmost column can be understood as the rotation of the vector $(1,0)$ by an angle φ , which results in the rotated vector $(\cos \varphi, \sin \varphi)$. (The illustration uses the angle t ; thus, the result is $(\cos t, \sin t)$.) Similarly, the first two components of the next column can be understood as the rotation of the vector $(0,1)$ by the angle φ with the result $(-\sin \varphi, \cos \varphi)$.

Analogously, the 4×4 rotation matrix $R_x(\theta)$ for a rotation by an angle θ ($= X$) about the x axis is:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

And the 4×4 rotation matrix $R_y(\psi)$ for a rotation by an angle ψ ($= Y$) about the y axis is:

$$R_y(\psi) = \begin{bmatrix} \cos \psi & 0 & \sin \psi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \psi & 0 & \cos \psi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Note that the sign of the sine terms depend on some conventions. Here we use the conventions of Unity.

These three matrices have to be combined in one matrix product to form the total rotation. Since (column) vectors are multiplied from the right to transformation matrices, the first rotation $R_z(\varphi)$ has to be in the rightmost place and the last rotation $R_y(\psi)$ has to be in the leftmost place. Thus, the correctly ordered matrix product is:

$$M_{\text{rotation}}(\varphi, \theta, \psi) = R_y(\psi)R_x(\theta)R_z(\varphi)$$

The matrix $M_{\text{rotation}}(\varphi, \theta, \psi)$ can then be multiplied with other elementary transformations matrices to form the model matrix as discussed in Section “Vertex Transformations”.

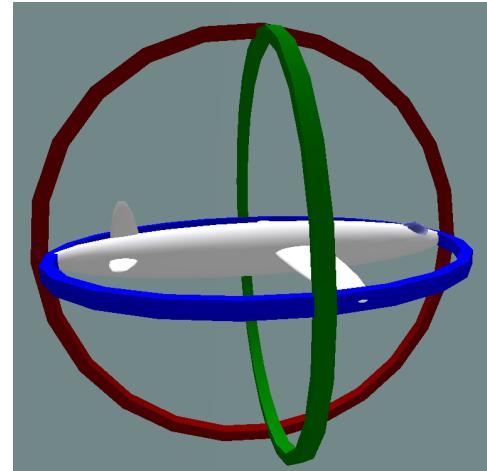
In Unity, you could compute a 4×4 matrix m for the Euler angles X, Y, Z in this way (quaternions are discussed in more detail below):

```
var m : Matrix4x4 = Matrix4x4.TRS(Vector3(0, 0, 0),
    Quaternion.Euler(X, Y, Z), Vector3(1, 1, 1));
```

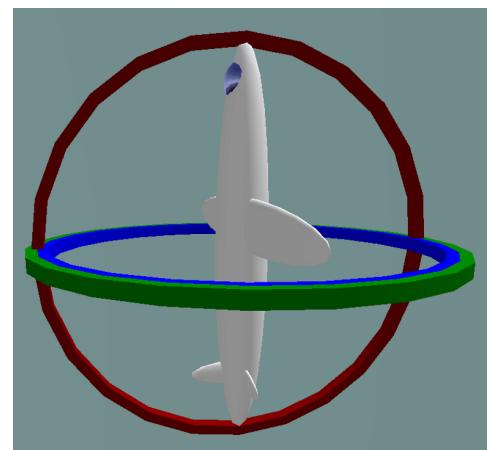
Gimbal Lock

One interesting feature of Euler angles (and one reason why Unity and most other graphics applications don't use them internally) is that they can result in a situation that is called “gimbal lock”. In this situation, two of the rotation axes are parallel and, therefore, only two different rotation axes are used. In other words, one degree of freedom is lost. This happens with Unity's Euler angles if the 2nd rotation (elevation) is $X = \pm 90^\circ$. In this case, the z axis is rotated onto the y axis; thus, the (rotated) first rotation axis is the same as the (fixed) third rotation axis.

The situation is easily constructed with the script from above if you set the elevation angle to 90° or -90° .



No gimbal lock: the axes of the three gimbals are different.



Gimbal lock: the axes of two gimbals are parallel.

Moving Rotation Axes

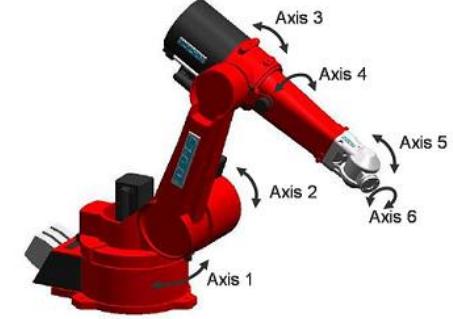
In some cases, for example robotics, it is more interesting to work with moving rotation axes (i.e. intrinsic rotations) instead of the fixed rotation axes (i.e. extrinsic rotations) discussed so far. Usually, the rotation relative to the base of the robot is referred to as “yaw”, the up-down rotation of the arm as “pitch”, and the next rotation as “roll”. Since the joints are rotated by each rotation, a description of rotations with moving rotation axes is required.

Fortunately, there is a simple equivalence: the extrinsic rotations specified by Z, X, and Y correspond to the following three intrinsic rotations (in this order): a rotation about the y axis by Y (“yaw”), a rotation about the rotated x axis by X (“pitch”), and a rotation about the rotated z axis by Z (“roll”). In other words, the rotation angles for moving rotation axes are the same as for fixed rotation axes if they are applied in reverse order (Y, X, Z instead of Z, X, Y). For most people, this equivalence is not intuitive at all; however, you might gain a better understanding by playing with the following script that lets you specify yaw, pitch, and roll.

```
@script ExecuteInEditMode()
#pragma strict

public var yawY : float;
public var pitchX : float;
public var rollZ : float;

function Update()
{
    transform.localEulerAngles =
        Vector3(pitchX, yawY, rollZ);
}
```

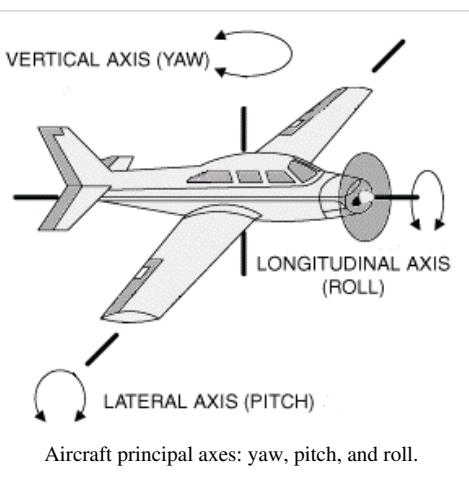


RV – Vertical articulating robot

An articulated robot: if the rotations about Axis 3, 5, and 6 were deactivated, the rotations about Axis 1, 2, and 4 would be referred to as yaw, pitch, and roll.

Roll, Pitch, and Yaw in Aviation

Roll, pitch, and yaw are also used to describe the orientation of vehicles; see the Wikipedia article on “Axes Conventions” [2]. Moreover, the principal axes of an aircraft are also called roll (z axis in Unity), pitch (x axis in Unity), and yaw (y axis in Unity). These axes are always fixed to the aircraft; thus, rolling, pitching, and yawing always refers to rotations about these aircraft principal axes regardless of the orientation of the aircraft. The mathematics of these rotations is therefore different from the Euler angles of the same name. In fact, the rotation about aircraft principal axes is more similar to the rotation about arbitrary axes discussed next.



Aircraft principal axes: yaw, pitch, and roll.

Rotation about an Axis by an Angle

As mentioned in Section “Vertex Transformations”, the rotation matrix for a rotation about a normalized axis (x, y, z) by an angle α is:

$$M_{\text{rotation}}(x, y, z, \alpha) = \begin{bmatrix} (1 - \cos \alpha)x^2 + \cos \alpha & (1 - \cos \alpha)xy - z \sin \alpha & (1 - \cos \alpha)xz + y \sin \alpha & 0 \\ (1 - \cos \alpha)xy + z \sin \alpha & (1 - \cos \alpha)y^2 + \cos \alpha & (1 - \cos \alpha)yz - x \sin \alpha & 0 \\ (1 - \cos \alpha)xz - y \sin \alpha & (1 - \cos \alpha)yz + x \sin \alpha & (1 - \cos \alpha)z^2 + \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Fortunately, this matrix is almost never needed when working with Unity. Instead, the object's local “rotation” can be set to a rotation about axis by angle in degrees in this way:

```
@script ExecuteInEditMode()
#pragma strict

public var angle : float;
public var axis : Vector3;

function Update()
{
    transform.localRotation =
        Quaternion.AngleAxis(angle, axis);
}
```

Actually, this sets the orientation of the object relative to its parent (or to the world if there is no parent). In order to rotate about a given axis at a given angular velocity in degrees per second, the function `Transform.Rotate` can be used:

```
#pragma strict

public var degreesPerSecond : float;
// angular speed
public var axis : Vector3;

function Update()
{
    transform.Rotate(axis,
        degreesPerSecond * Time.deltaTime,
        Space.Self);
}
```

`degreesPerSecond` is an angular speed, which specifies how fast the object is rotating. However, `Transform.Rotate` requires the rotation angle in degrees. To compute this angle, the angular speed has to be multiplied with the time (in seconds) that has passed since the last call to `Update`, which is `Time.deltaTime`.

Furthermore, `Transform.Rotate` takes a third argument that specifies the coordinate system in which the rotation axis is specified: either `Space.Self` for the object's local coordinate system or `Space.World` for the world coordinate system. With `Space.Self` we can easily simulate rolling, pitching, and yawing by specifying the axes $(1,0,0)$ for pitching, $(0,1,0)$ for yawing, and $(0,0,1)$ for rolling.

Quaternions

As you might have noticed in the code above, Unity is using quaternions (actually normalized quaternions) to represent rotations. For example, the variable `Transform.localRotation` is of type `Quaternion`.

In some sense, quaternions are simply four-dimensional vectors with some special functions. Normalized quaternions of length 1 correspond to rotations and are easy to construct when you know the normalized rotation axis (x, y, z) and the rotation angle α . The corresponding quaternion q is just:

$$q = (x_q, y_q, z_q, w_q) = (x \sin(\alpha/2), y \sin(\alpha/2), z \sin(\alpha/2), \cos(\alpha/2))$$

In Unity, however, you can just use the constructor `Quaternion.AngleAxis(alpha, Vector3(x, y, z))` with alpha in degrees to construct the normalized quaternion. (See the previous subsection for an example.)

Conversely, a normalized quaternion q with components (x_q, y_q, z_q, w_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) . In Unity, you can use the function `Quaternion.ToAngleAxis` to compute the corresponding axis and angle. As normalized quaternions correspond to rotations, they also correspond to rotation matrices. In fact, the product of two normalized quaternions corresponds to the product of the corresponding rotation matrices in the same order. Computing the product of two quaternions is actually a bit tricky but in Unity you can just use the `*` operator. In the case of products of quaternions (and also inverse quaternions), it is therefore useful to think of a quaternion as “something like a rotation matrix” instead of a four-dimensional vector.

For example, the code for the rotation at a given angular speed about an axis in the object's local coordinate system would require to multiply — in this order — the previous rotation matrix (which specifies the previous orientation) with the new (incremental) rotation matrix. The quaternion for the previous orientation is `transform.localRotation` and the new incremental quaternion is called `q` in this code:

```
#pragma strict

public var degreesPerSecond : float;
// angular speed
public var axis : Vector3;

function Update()
{
    var q : Quaternion = Quaternion.AngleAxis(
        degreesPerSecond * Time.deltaTime, axis);
    transform.localRotation =
        transform.localRotation * q;
}
```

If the quaternion product in the last line is changed to

```
transform.localRotation =
    q * transform.localRotation;
```

it corresponds to applying the incremental rotation in the parent's coordinate system (or the world's coordinate system if there is no parent) just like one would expect for applying a rotation matrix after transforming the vertices into the parent's coordinate system. (Remember that matrix products should be read from right to left because column vectors are multiplied from the right.)

At least in computer graphics, (normalized) quaternions are therefore rather harmless and never more complicated than rotation matrices or axis-angle representations (depending on the context). Their specific advantages are that

they show no gimbal lock (as opposed to Euler angles), they can be easily combined by multiplication (as opposed to Euler angles and the angle-axis representation of rotations), and they are easy to normalize (as opposed to the orthogonalization of rotation matrices). Therefore, most graphics applications use quaternions internally to represent rotations — even if Euler angles are used in the user interface.

Summary

In this rather theoretical tutorial, we have looked at:

- Unity's Euler angles for extrinsic rotations (i.e., elevation, heading, and bank).
- Unity's Euler angles for intrinsic rotations (i.e., pitch, yaw, and roll).
- The axis-angle representation of rotations.
- The quaternion representation of rotations.

Further reading

If you want to know more

- about the model matrix, you should read the description in Section “Vertex Transformations”.
- about Euler angles, you could read the Wikipedia article about Euler angles ^[3].
- about the use of Euler angles for specifying the orientation of vehicles, you could read the Wikipedia article about axes conventions ^[4].
- about related classes in Unity, you should read the official documentation of `Transform` ^[5], `Quaternion` ^[6], and `Matrix4x4` ^[7].

page traffic for 90 days ^[8]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

- [1] <http://docs.unity3d.com/Documentation/ScriptReference/Transform-localEulerAngles.html>
- [2] http://en.wikipedia.org/wiki/Axes_conventions#Frames_mounted_on_vehicles
- [3] http://en.wikipedia.org/wiki/Euler_angles
- [4] http://en.wikipedia.org/wiki/Axes_conventions
- [5] <http://docs.unity3d.com/Documentation/ScriptReference/Transform.html>
- [6] <http://docs.unity3d.com/Documentation/ScriptReference/Quaternion.html>
- [7] <http://docs.unity3d.com/Documentation/ScriptReference/Matrix4x4.html>
- [8] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Rotations

9.2 Projection for Virtual Reality



User in a CAVE. The user's head position is tracked and the graphics on the walls are computed for this tracked position.

This tutorial discusses off-axis perspective projection in Unity. It is based on Section “Vertex Transformations”. No shader programming is required since only the view matrix and the projection matrix are changed, which is implemented in JavaScript.

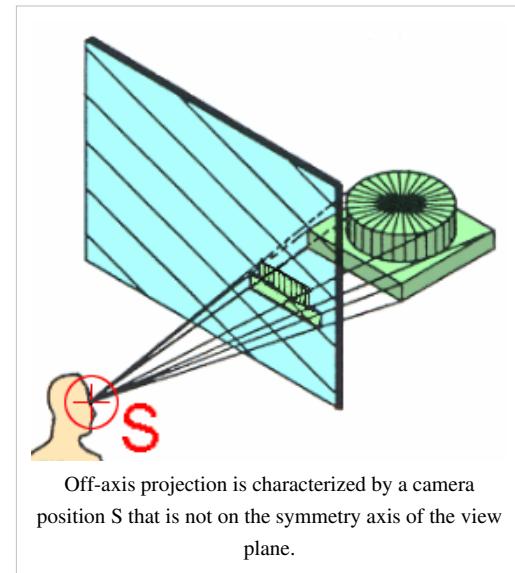
The main application of off-axis perspective projection are virtual reality environments such as the CAVE shown in the photo. Usually, the user's head position is tracked and the perspective projection for each display is computed for a camera at the tracked position such that the user experiences the illusion of looking through a window into a three-dimensional world instead of looking at a flat display.

Off-Axis vs. On-Axis Perspective Projection

On-axis projection refers to camera positions that are on the symmetry axis of the view plane, i.e. the axis through the center of the view plane and orthogonal to it. This case is discussed in Section “Vertex Transformations”.

In virtual reality environments, however, the virtual camera often follows the tracked position of the user's head in order to create parallax effects and thus a more compelling illusion of a three-dimensional world. Since the tracked head position is not limited to the symmetry axis of the view plane, on-axis projection is not sufficient for most virtual reality environments.

Off-axis perspective projection addresses this issue by allowing for arbitrary camera positions. While some low-level graphics APIs (e.g. older versions of OpenGL) supported off-axis projection, they had much better support for on-axis projection since this was the more common case. Similarly, many high-level tools (e.g. Unity) support off-axis projection but provide much better support for on-axis projection, i.e. you can specify any on-axis projection with some mouse clicks but you need to write a script to implement off-axis projection.



Computing Off-Axis Perspective Projection

Off-axis perspective projection requires a different view matrix and a different projection matrix than on-axis perspective projection. For the computation of the on-axis view matrix, a specified view direction is rotated onto the z axis as described in Section “Vertex Transformations”. The only difference for an off-axis view matrix is that this “view direction” is computed as the orthogonal direction to the specified view plane, i.e. the surface normal vector of the view plane.

The off-axis projection matrix has to be changed since the edges of the view plane are no longer symmetric around the intersection point with the (technical) “view direction.” Thus, the four distances to the edges have to be computed and put into a suitable projection matrix. For details, see the description by Robert Kooima in his publication

“Generalized Perspective Projection”^[1]. The next section presents an implementation of this technique in Unity.

Camera Script

The following script is based on the code in Robert Kooima's publication. There are very few implementation differences. One is that, in Unity, the view plane is more easily specified as a built-in Plane object, which has corners at $(\pm 5, 0, \pm 5)$ in object coordinates. Furthermore, the original code was written for a right-handed coordinate system while Unity uses a left-handed coordinate system; thus, the result of the cross product has to be multiplied with -1. Another difference is that the rotation of the camera's GameObject and the parameter `fieldOfView` are used by Unity for view frustum culling; therefore, the script should set those values to appropriate values. (These values have no meaning for the computation of the matrices.) Unfortunately, this might cause problems if other scripts (namely the script that sets the tracked head position) are also setting the camera rotation. Therefore, this estimation can be deactivated with the variable `estimateViewFrustum` (at the risk of incorrect view frustum culling by Unity).

```
// This script should be attached to a Camera object
// in Unity. Once a Plane object is specified as the
// "projectionScreen", the script computes a suitable
// view and projection matrix for the camera.
// The code is based on Robert Kooima's publication
// "Generalized Perspective Projection," 2009,
// http://csc.lsu.edu/~kooima/pdfs/gen-perspective.pdf
#pragma strict

public var projectionScreen : GameObject;
public var estimateViewFrustum : boolean = true;

function LateUpdate() {
    if (null != projectionScreen)
    {
        var pa : Vector3 =
            projectionScreen.transform.TransformPoint(
            Vector3(-5.0, 0.0, -5.0));
        // lower left corner in world coordinates
        var pb : Vector3 =
            projectionScreen.transform.TransformPoint(
            Vector3(5.0, 0.0, -5.0));
        // lower right corner
        var pc : Vector3 =
            projectionScreen.transform.TransformPoint(
            Vector3(-5.0, 0.0, 5.0));
        // upper left corner
        var pe : Vector3 = transform.position;
        // eye position
        var n : float = camera.nearClipPlane;
        // distance of near clipping plane
        var f : float = camera.farClipPlane;
        // distance of far clipping plane

        var va : Vector3; // from pe to pa
```

```
var vb : Vector3; // from pe to pb
var vc : Vector3; // from pe to pc
var vr : Vector3; // right axis of screen
var vu : Vector3; // up axis of screen
var vn : Vector3; // normal vector of screen

var l : float; // distance to left screen edge
var r : float; // distance to right screen edge
var b : float; // distance to bottom screen edge
var t : float; // distance to top screen edge
var d : float; // distance from eye to screen

vr = pb - pa;
vu = pc - pa;
vr.Normalize();
vu.Normalize();
vn = -Vector3.Cross(vr, vu);
// we need the minus sign because Unity
// uses a left-handed coordinate system
vn.Normalize();

va = pa - pe;
vb = pb - pe;
vc = pc - pe;

d = -Vector3.Dot(va, vn);
l = Vector3.Dot(vr, va) * n / d;
r = Vector3.Dot(vr, vb) * n / d;
b = Vector3.Dot(vu, va) * n / d;
t = Vector3.Dot(vu, vc) * n / d;

var p : Matrix4x4; // projection matrix
p[0,0] = 2.0*n/(r-l);
p[0,1] = 0.0;
p[0,2] = (r+l)/(r-l);
p[0,3] = 0.0;

p[1,0] = 0.0;
p[1,1] = 2.0*n/(t-b);
p[1,2] = (t+b)/(t-b);
p[1,3] = 0.0;

p[2,0] = 0.0;
p[2,1] = 0.0;
p[2,2] = (f+n)/(n-f);
p[2,3] = 2.0*f*n/(n-f);
```

```
p[3,0] = 0.0;  
p[3,1] = 0.0;  
p[3,2] = -1.0;  
p[3,3] = 0.0;  
  
var rm : Matrix4x4; // rotation matrix;  
rm[0,0] = vr.x;  
rm[0,1] = vr.y;  
rm[0,2] = vr.z;  
rm[0,3] = 0.0;  
  
rm[1,0] = vu.x;  
rm[1,1] = vu.y;  
rm[1,2] = vu.z;  
rm[1,3] = 0.0;  
  
rm[2,0] = vn.x;  
rm[2,1] = vn.y;  
rm[2,2] = vn.z;  
rm[2,3] = 0.0;  
  
rm[3,0] = 0.0;  
rm[3,1] = 0.0;  
rm[3,2] = 0.0;  
rm[3,3] = 1.0;  
  
var tm : Matrix4x4; // translation matrix;  
tm[0,0] = 1.0;  
tm[0,1] = 0.0;  
tm[0,2] = 0.0;  
tm[0,3] = -pe.x;  
  
tm[1,0] = 0.0;  
tm[1,1] = 1.0;  
tm[1,2] = 0.0;  
tm[1,3] = -pe.y;  
  
tm[2,0] = 0.0;  
tm[2,1] = 0.0;  
tm[2,2] = 1.0;  
tm[2,3] = -pe.z;  
  
tm[3,0] = 0.0;  
tm[3,1] = 0.0;  
tm[3,2] = 0.0;  
tm[3,3] = 1.0;
```

```

// set matrices
camera.projectionMatrix = p * rm * tm;
camera.worldToCameraMatrix = Matrix4x4.identity;
// we put everything into the projection matrix:
// because our "viewing matrix" might look at a
// point that is off the screen.

if (estimateViewFrustum)
{
    // rotate camera to screen for culling to work
    var q : Quaternion;
    q.SetLookRotation((0.5 * (pb + pc) - pe), vu);
        // look at center of screen
    camera.transform.rotation = q;

    // set fieldOfView to a conservative estimate
    // to make frustum tall enough
    if (camera.aspect >= 1.0)
    {
        camera.fieldOfView = Mathf.Rad2Deg *
            Mathf.Atan((vu.magnitude + vr.magnitude)
            / va.magnitude);
    }
    else
    {
        // take the camera aspect into account to
        // make the frustum wide enough
        camera.fieldOfView =
            Mathf.Rad2Deg / camera.aspect *
            Mathf.Atan((vu.magnitude + vr.magnitude)
            / va.magnitude);
    }
}
}
}

```

To use this script, choose **Create > Javascript** in the **Project** view, double-click the new script to edit it, and copy & paste the code from above into it. Then attach the script to your main camera (drag it from the **Project** view over the camera object in the **Hierarchy** view). Furthermore, create a Plane object (**GameObject > Create Other > Plane** in the main menu) and place it into the virtual scene to define the view plane. Deactivate the **Mesh Renderer** of the Plane in the **Inspector** to make it invisible (it is only a placeholder), but make sure that the front face of the plane is always facing the camera. Select the camera object and drag the plane object to **Projection Screen** in the **Inspector**. The script will be active when the game is started. Add the line

```
@script ExecuteInEditMode()
```

at the top of the code to make the script also run in the editor.

There are a couple of limitations of this implementation in Unity:

- To keep the script simple, it assumes that the front face of the plane (the face that is visible when backface culling is active) is facing the camera.
- The camera preview in the Scene view doesn't take the changed projection matrix into account and is therefore no longer useful.
- The built-in skybox system also doesn't take the changed projection matrix into account and is therefore also no longer useful. (See Section "Skyboxes" for an alternative way of implementing skyboxes.)
- There are probably further parts of Unity that ignore the new projection matrix and that therefore are unusable in combination with this script.

Summary

In this tutorial, we have looked at:

- uses of off-axis perspective projection and differences to on-axis perspective projection
- the computation of view and projection matrices for off-axis perspective projection
- an implementation of this computation and its limitations in Unity

Further Reading

If you want to know more

- about the on-axis view matrix and the on-axis projection matrix, you should read the description in Section "Vertex Transformations".
- about the implemented algorithm, you should read Robert Kooima's publication "Generalized Perspective Projection" [1].

page traffic for 90 days [2]

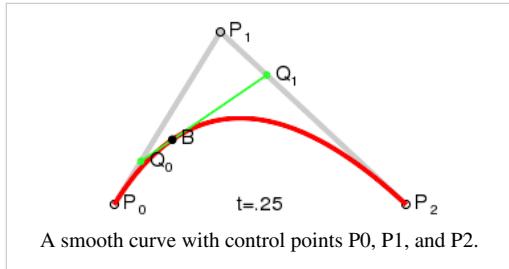
< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

- [1] <http://csc.lsu.edu/~kooima/pdfs/gen-perspective.pdf>
- [2] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Projection_for_Virtual_Reality

9.3 Bézier Curves



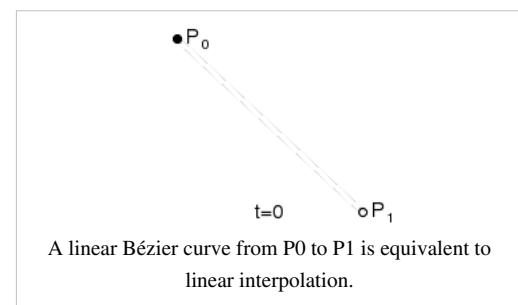
This tutorial discusses one way to render quadratic Bézier curves and splines in Unity. No shader programming is required since all the code is implemented in JavaScript.

There are many applications of smooth curves in computer graphics, in particular in animation and modeling. In a 3D game engine, one would usually use a tube-like mesh and deform it with vertex blending as discussed in Section “Nonlinear Deformations” instead of rendering curves; however, rendering curved lines instead of deformed meshes can offer a substantial performance advantage.

Linear Bézier Curves

The simplest Bézier curve is a linear Bézier curve $B(t)$ for t from 0 to 1 between two points P_0 and P_1 , which happens to be the same as linear interpolation between the two points:

$$B(t) = (1 - t)P_0 + tP_1$$



You might be fancy and call $1 - t$ and t the Bernstein basis polynomials of degree 1, but it really is just linear interpolation.

Quadratic Bézier Curves

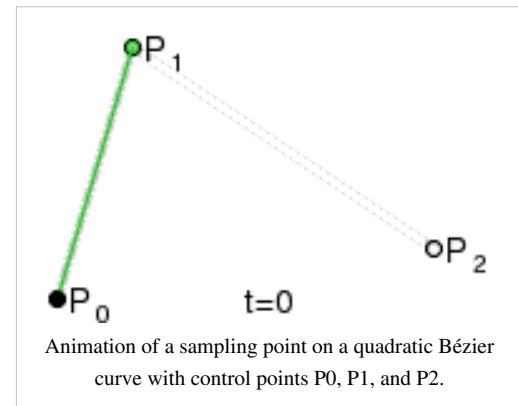
A more interesting Bézier curve is a quadratic Bézier curve $B(t)$ for t from 0 to 1 between two points P_0 and P_2 but influenced by a third point P_1 in the middle. The definition is:

$$B(t) = (1 - t)^2 P_0 + 2(1 - t)t P_1 + t^2 P_2$$

This defines a smooth curve $B(t)$ with $t \in [0, 1]$ that starts (for $t = 0$) from position P_0 in the direction to P_1 but then bends to P_2 (and reaches it for $t = 1$).

In practice, one usually samples the interval from 0 to 1 at sufficiently many points, e.g.

$B(0), B(0.05), B(0.1), B(0.15), B(0.2), \dots, B(1)$ and then renders straight lines between these sample points.



Curve Script

To implement such a curve in Unity, we can use the Unity component `LineRenderer`^[1]. Apart from setting some parameters, one should set the number of sample points on the curve with the function `SetVertexCount`. Then the sample points have to be computed and set with the function `SetPosition`. This is can be implemented this way:

```
var t : float;
var position : Vector3;
for(var i : int = 0; i < numberOfPoints; i++)
{
    t = i / (numberOfPoints - 1.0);
    position = (1.0 - t) * (1.0 - t) * p0
        + 2.0 * (1.0 - t) * t * p1
        + t * t * p2;
    lineRenderer.SetPosition(i, position);
}
```

Here we use an index `i` from 0 to `numberOfPoints-1` to count the sample points. From this index `i` a parameter `t` from 0 to 1 is computed. The next line computes $B(t)$, which is then set with the function `SetPosition`.

The rest of the code just sets up the `LineRenderer` component and defines public variables that can be used to define the control points and some rendering features of the curve.

```
@script ExecuteInEditMode()
#pragma strict

public var start : GameObject;
public var middle : GameObject;
public var end : GameObject;

public var color : Color = Color.white;
public var width : float = 0.2;
public var numberOfPoints : int = 20;

function Start()
{
    // initialize line renderer component
    var lineRenderer : LineRenderer =
        GetComponent(LineRenderer);
    if (null == lineRenderer)
    {
        gameObject.AddComponent(LineRenderer);
    }
    lineRenderer = GetComponent(LineRenderer);
    lineRenderer.useWorldSpace = true;
    lineRenderer.material = new Material(
        Shader.Find("Particles/Additive"));
}
```

```
function Update()
{
    // check parameters and components
    var lineRenderer : LineRenderer =
        GetComponent(LineRenderer);
    if (null == lineRenderer || null == start
        || null == middle || null == end)
    {
        return; // no points specified
    }

    // update line renderer
    lineRenderer.SetColors(color, color);
    lineRenderer.SetWidth(width, width);
    if (numberOfPoints > 0)
    {
        lineRenderer.SetVertexCount(numberOfPoints);
    }

    // set points of quadratic Bezier curve
    var p0 : Vector3 = start.transform.position;
    var p1 : Vector3 = middle.transform.position;
    var p2 : Vector3 = end.transform.position;
    var t : float;
    var position : Vector3;
    for(var i : int = 0; i < numberOfPoints; i++)
    {
        t = i / (numberOfPoints - 1.0);
        position = (1.0 - t) * (1.0 - t) * p0
            + 2.0 * (1.0 - t) * t * p1
            + t * t * p2;
        lineRenderer.SetPosition(i, position);
    }
}
```

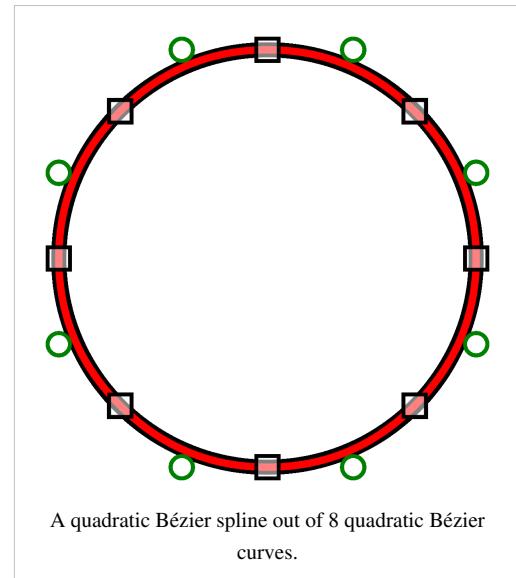
To use this script **create a Javascript** in the **Project** view, double-click it, copy & paste the code above, save it, create a new empty game object (in the main menu: **GameObject > Create Empty**) and attach the script (drag the script from the **Project** view over the empty game object in the **Hierarchy**).

Then create three more empty game objects (or any other game objects) with different(!) positions that will serve as control points. Select the game object with the script and drag the other game objects into the slots **Start**, **Middle**, and **End** in the **Inspector**. This should render a curve from the game object specified as “Start” to the game object specified as “End” bending towards “Middle”.

Quadratic Bézier Splines

A quadratic Bézier Spline is just a continuous, smooth curve that consist of segments that are quadratic Bézier curves. If the control points of the curves were chosen arbitrarily, the spline would neither be continuous nor smooth; thus, the control points have to be chosen in particular ways.

One common way is to use a certain set of user-specified control points (green circles in the figure) as the P_1 control points of the segments and to choose the center positions between two adjacent user-specified control points as the P_0 and P_2 control points (black rectangles in the figure). This actually guarantees that the spline is smooth (also in the mathematical sense that the tangent vector is continuous).



Spline Script

The following script implements this idea. For the j-th segment, it computes P_0 as the average of the j-th and (j+1)-th user-specified control points, P_1 is set to the (j+1)-th user-specified control point, and P_2 is the average of the (j+1)-th and (j+2)-th user-specified control points:

```
p0 = 0.5 * (controlPoints[j].transform.position
    + controlPoints[j + 1].transform.position);
p1 = controlPoints[j + 1].transform.position;
p2 = 0.5 * (controlPoints[j + 1].transform.position
    + controlPoints[j + 2].transform.position);
```

Each individual segment is then just computed as a quadratic Bézier curve. The only adjustment is that all but the last segment should not reach P_2 . If they did, the first sample position of the next segment would be at the same position which would be visible in the rendering. The complete script is:

```
@script ExecuteInEditMode()
#pragma strict

public var controlPoints : GameObject[] = new GameObject[3];
public var color : Color = Color.white;
public var width : float = 0.2;
public var numberOfPoints : int = 20;

function Start()
{
    // initialize line renderer component
    var lineRenderer : LineRenderer =
        GetComponent(LineRenderer);
    if (null == lineRenderer)
    {
        gameObject.AddComponent(LineRenderer);
```

```
        }

        lineRenderer = GetComponent(LineRenderer);
        lineRenderer.useWorldSpace = true;
        lineRenderer.material = new Material(
            Shader.Find("Particles/Additive"));
    }

function Update()
{
    // check parameters and components
    var lineRenderer : LineRenderer =
        GetComponent(LineRenderer);
    if (null == lineRenderer || controlPoints == null
        || controlPoints.length < 3)
    {
        return; // not enough points specified
    }

    // update line renderer
    lineRenderer.SetColors(color, color);
    lineRenderer.SetWidth(width, width);
    if (numberOfPoints < 2)
    {
        numberOfPoints = 2;
    }
    lineRenderer.SetVertexCount(numberOfPoints *
        (controlPoints.length - 2));

    // loop over segments of spline
    var p0 : Vector3;
    var p1 : Vector3;
    var p2 : Vector3;
    for (var j : int = 0; j < controlPoints.length - 2; j++)
    {
        // check control points
        if (controlPoints[j] == null ||
            controlPoints[j + 1] == null ||
            controlPoints[j + 2] == null)
        {
            return;
        }
        // determine control points of segment
        p0 = 0.5 * (controlPoints[j].transform.position
            + controlPoints[j + 1].transform.position);
        p1 = controlPoints[j + 1].transform.position;
        p2 = 0.5 * (controlPoints[j + 1].transform.position
            + controlPoints[j + 2].transform.position);
    }
}
```

```
// set points of quadratic Bezier curve
var position : Vector3;
var t : float;
var pointStep : float = 1.0 / numberOfPoints;
if (j == controlPoints.length - 3)
{
    pointStep = 1.0 / (numberOfPoints - 1.0);
    // last point of last segment should reach p2
}
for(var i : int = 0; i < numberOfPoints; i++)
{
    t = i * pointStep;
    position = (1.0 - t) * (1.0 - t) * p0
        + 2.0 * (1.0 - t) * t * p1
        + t * t * p2;
    lineRenderer.SetPosition(i + j * numberOfPoints,
        position);
}
}
```

The script works in the same way as the script for Bézier curves except that the user can specify an arbitrary number of control points. For closed splines, the last two user-specified control points should be the same as the first two control points. For open splines that actually reach the end points, the first and last control point should be specified twice.

Summary

In this tutorial, we have looked at:

- the definition of linear and quadratic Bézier curves and quadratic Bézier splines
- implementations of quadratic Bézier curves and quadratic Bézier splines with Unity's LineRenderer component.

Further Reading

If you want to know more

- about Bézier curves (and Bézier splines), the Wikipedia article on “Bézier curve”^[2] provides a good starting point.
- about Unity's LineRenderer, you should read Unity's documentation of the class LineRenderer^[1].

page traffic for 90 days^[3]

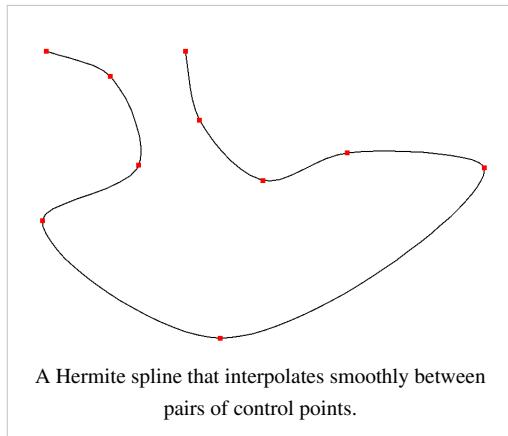
< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

- [1] <http://docs.unity3d.com/Documentation/ScriptReference/LineRenderer.html>
- [2] http://en.wikipedia.org/wiki/B%C3%A9zier_curve
- [3] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/B%C3%A9zier_Curves

9.4 Hermite Curves



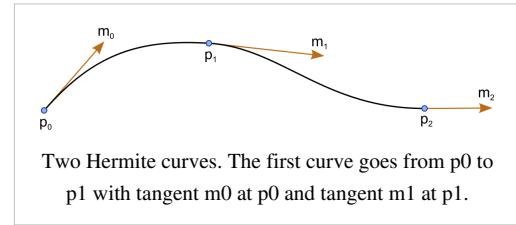
This tutorial discusses Hermite curves (more exactly: cubic Hermite curves) and Catmull-Rom splines in Unity. The latter are a special kind of cubic Hermite splines. No shader programming is required since all the code is implemented in JavaScript.

Some splines (for example the quadratic Bézier spline discussed in Section “Bézier Curves”) don't go through all control points, i.e. they don't interpolate between them. Hermite splines, on the other hand, can be defined such that they go through all control points. This is a useful feature in many applications, e.g. in animation where it is often important to set specific values for particular key frames and let tools smoothly interpolate further values for inbetweens.

Hermite Curves

A single cubic Hermite curve $H(t)$ for t from 0 to 1 is defined by a start point P_0 with tangent M_0 and an end point P_1 with tangent M_1 :

$$H(t) = (2t^3 - 3t^2 + 1)P_0 + (t^3 - 2t^2 + t)M_0 + (-2t^3 + 3t^2)P_1 + (t^3 - t^2)M_1$$



The curve starts (for $t = 0$) at P_0 in direction of M_0 and then changes course to direction M_1 and reaches P_1 for $t = 1$. As the figure illustrates, two Hermite curves can be smoothly attached to each other by choosing the same tangent vector for the corresponding end points.

Curve Script

To implement such a curve in Unity, we can use the Unity component `LineRenderer`^[1]. Apart from setting some parameters, one should set the number of sample points on the curve with the function `SetVertexCount`. Then the sample points have to be computed and set with the function `SetPosition`. This is can be implemented this way:

```
var t : float;
var position : Vector3;
for(var i : int = 0; i < numberOfPoints; i++)
{
    t = i / (numberOfPoints - 1.0);
    position = (2.0*t*t*t - 3.0*t*t + 1.0) * p0
        + (t*t*t - 2.0*t*t + t) * m0
        + (-2.0*t*t*t + 3.0*t*t) * p1
        + (t*t*t - t*t) * m1;
```

```
    lineRenderer.SetPosition(i, position);
}
```

Here we use an index `i` from 0 to `numberOfPoints-1` to count the sample points. From this index `i` a parameter `t` from 0 to 1 is computed. The next line computes $H(t)$, which is then set with the function `SetPosition`.

The rest of the code just sets up the `LineRenderer` component and defines public variables that can be used to define the control points and some rendering features of the curve.

```
@script ExecuteInEditMode()
#pragma strict

public var start : GameObject;
public var startTangentPoint : GameObject;
public var end : GameObject;
public var endTangentPoint : GameObject;

public var color : Color = Color.white;
public var width : float = 0.2;
public var numberOfPoints : int = 20;

function Start()
{
    // initialize line renderer component
    var lineRenderer : LineRenderer =
        GetComponent(LineRenderer);
    if (null == lineRenderer)
    {
        gameObject.AddComponent(LineRenderer);
    }
    lineRenderer = GetComponent(LineRenderer);
    lineRenderer.useWorldSpace = true;
    lineRenderer.material = new Material(
        Shader.Find("Particles/Additive"));
}

function Update()
{
    // check parameters and components
    var lineRenderer : LineRenderer =
        GetComponent(LineRenderer);
    if (null == lineRenderer
        || null == start || null == startTangentPoint
        || null == end || null == endTangentPoint)
    {
        return; // no points specified
    }
}
```

```
// update line renderer
lineRenderer.SetColors(color, color);
lineRenderer.SetWidth(width, width);
if (numberOfPoints > 0)
{
    lineRenderer.SetVertexCount(numberOfPoints);
}

// set points of Hermite curve
var p0 : Vector3 = start.transform.position;
var p1 : Vector3 = end.transform.position;
var m0 : Vector3 = startTangentPoint.transform.position
    - start.transform.position;
var m1 : Vector3 = endTangentPoint.transform.position
    - end.transform.position;
var t : float;
var position : Vector3;
for(var i : int = 0; i < numberOfPoints; i++)
{
    t = i / (numberOfPoints - 1.0);
    position = (2.0*t*t*t - 3.0*t*t + 1.0) * p0
        + (t*t*t - 2.0*t*t + t) * m0
        + (-2.0*t*t*t + 3.0*t*t) * p1
        + (t*t*t - t*t) * m1;
    lineRenderer.SetPosition(i, position);
}
}
```

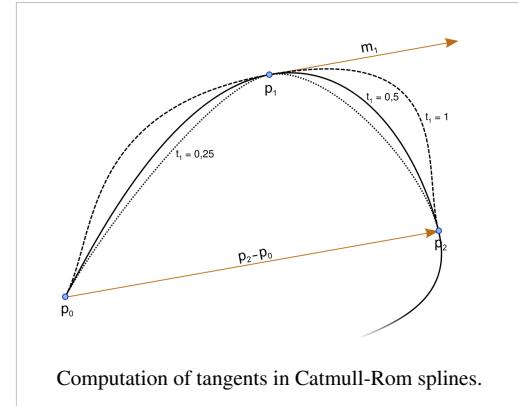
To use this script **create a Javascript** in the **Project** view, double-click it, copy & paste the code above, save it, create a new empty game object (in the main menu: **GameObject > Create Empty**) and attach the script (drag the script from the **Project** view over the empty game object in the **Hierarchy**).

Then create four more empty game objects (or any other game objects) with different(!) positions that will serve as control points. Select the game object with the script and drag the other game objects into the slots **Start**, **StartTangentPoint** (for the end point of a tangent starting in the start point), **End**, and **EndTangentPoint** in the **Inspector**. This should render a Hermite curve from the game object specified as “Start” to the game object specified as “End”.

Catmull-Rom Splines

A cubic Hermite spline consists of a continuous, smooth sequence of cubic Hermite curves. In order to guarantee smoothness, the tangent at the end point of one Hermite curve is the same as the tangent of the start point of the next Hermite curve. In some cases, users provide these tangents (one for each control point), in other cases, however, appropriate tangents have to be computed.

One specific way of computing a tangent vector m_k for the k-th control point p_k is this:



$$m_k = \frac{p_{k+1} - p_{k-1}}{2}$$

and $m_k = p_{k+1} - p_k$ for the first point and $m_k = p_k - p_{k-1}$ for the last point. The resulting cubic Hermite spline is called Catmull-Rom spline.

Spline Script

The following script implements this idea. For the j-th segment, it computes P_0 as the j-th control point p_j , P_1 is set to p_{j+1} , M_0 is set to $(p_{j+1} - p_{j-1})/2$ (unless it is the tangent of the first control point, in which case it is set to $p_{j+1} - p_j$) and M_1 is set to $(p_{j+2} - p_j)/2$ (unless it is the tangent of the last control point, then it is set to $p_{j+1} - p_j$).

```

p0 = controlPoints[j].transform.position;
p1 = controlPoints[j + 1].transform.position;
if (j > 0)
{
    m0 = 0.5 * (controlPoints[j + 1].transform.position
        - controlPoints[j - 1].transform.position);
}
else
{
    m0 = controlPoints[j + 1].transform.position
        - controlPoints[j].transform.position;
}
if (j < controlPoints.length - 2)
{
    m1 = 0.5 * (controlPoints[j + 2].transform.position
        - controlPoints[j].transform.position);
}
else
{
    m1 = controlPoints[j + 1].transform.position
        - controlPoints[j].transform.position;
}

```

Each individual segment is then just computed as a cubic Hermite curve. The only adjustment is that all but the last segment should not reach P_1 . If they did, the first sample position of the next segment would be at the same

position which would be visible in the rendering. The complete script is:

```
@script ExecuteInEditMode()
#pragma strict

public var controlPoints : GameObject[] = new GameObject[3];
public var color : Color = Color.white;
public var width : float = 0.2;
public var numberOfPoints : int = 20;

function Start()
{
    // initialize line renderer component
    var lineRenderer : LineRenderer =
        GetComponent(LineRenderer);
    if (null == lineRenderer)
    {
        gameObject.AddComponent(LineRenderer);
    }
    lineRenderer = GetComponent(LineRenderer);
    lineRenderer.useWorldSpace = true;
    lineRenderer.material = new Material(
        Shader.Find("Particles/Additive"));
}

function Update()
{
    // check parameters and components
    var lineRenderer : LineRenderer =
        GetComponent(LineRenderer);
    if (null == lineRenderer || controlPoints == null
        || controlPoints.length < 2)
    {
        return; // not enough points specified
    }

    // update line renderer
    lineRenderer.SetColors(color, color);
    lineRenderer.SetWidth(width, width);
    if (numberOfPoints < 2)
    {
        numberOfPoints = 2;
    }
    lineRenderer.SetVertexCount(numberOfPoints *
        (controlPoints.length - 1));

    // loop over segments of spline
    var p0 : Vector3;
```

```
var p1 : Vector3;
var m0 : Vector3;
var m1 : Vector3;
for (var j : int = 0; j < controlPoints.length - 1; j++)
{
    // check control points
    if (controlPoints[j] == null ||
        controlPoints[j + 1] == null ||
        (j > 0 && controlPoints[j - 1] == null) ||
        (j < controlPoints.length - 2 &&
         controlPoints[j + 2] == null))
    {
        return;
    }
    // determine control points of segment
    p0 = controlPoints[j].transform.position;
    p1 = controlPoints[j + 1].transform.position;
    if (j > 0)
    {
        m0 = 0.5 * (controlPoints[j + 1].transform.position
                    - controlPoints[j - 1].transform.position);
    }
    else
    {
        m0 = controlPoints[j + 1].transform.position
            - controlPoints[j].transform.position;
    }
    if (j < controlPoints.length - 2)
    {
        m1 = 0.5 * (controlPoints[j + 2].transform.position
                    - controlPoints[j].transform.position);
    }
    else
    {
        m1 = controlPoints[j + 1].transform.position
            - controlPoints[j].transform.position;
    }

    // set points of Hermite curve
    var position : Vector3;
    var t : float;
    var pointStep : float = 1.0 / numberOfPoints;
    if (j == controlPoints.length - 2)
    {
        pointStep = 1.0 / (numberOfPoints - 1.0);
        // last point of last segment should reach p1
    }
}
```

```
for(var i : int = 0; i < numberOfPoints; i++)
{
    t = i * pointStep;
    position = (2.0*t*t*t - 3.0*t*t + 1.0) * p0
        + (t*t*t - 2.0*t*t + t) * m0
        + (-2.0*t*t*t + 3.0*t*t) * p1
        + (t*t*t - t*t) * m1;
    lineRenderer.SetPosition(i + j * numberOfPoints,
        position);
}
}
```

The script works in the same way as the script for Hermite curves except that the user can specify an arbitrary number of control points and doesn't have to specify tangent points.

Summary

In this tutorial, we have looked at:

- the definition of cubic Hermite curves and Catmull-Rom splines
- implementations of cubic Hermite curves and Catmull-Rom splines with Unity's LineRenderer component.

Further Reading

If you want to know more

- about Hermite splines, the Wikipedia article on “cubic Hermite spline”^[1] provides a good starting point.
- about Unity's LineRenderer, you should read Unity's documentation of the class LineRenderer^[1].

page traffic for 90 days^[2]

< Cg Programming/Unity

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

- [1] http://en.wikipedia.org/wiki/Cubic_Hermite_spline
- [2] http://stats.grok.se/en.b/latest90/Cg_Programming/Unity/Hermite_Curves

Appendix on the Programmable Graphics Pipeline and Cg Syntax

A.1 Programmable Graphics Pipeline

The programmable graphics pipeline presented here is very similar to the OpenGL (ES) 2.0 pipeline, the WebGL pipeline, and the Direct3D 8.0 pipeline. As such it is the lowest common denominator of programmable graphics pipelines of the majority of today's desktop PCs and mobile devices.

Parallelism in Graphics Pipelines

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:

- **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.



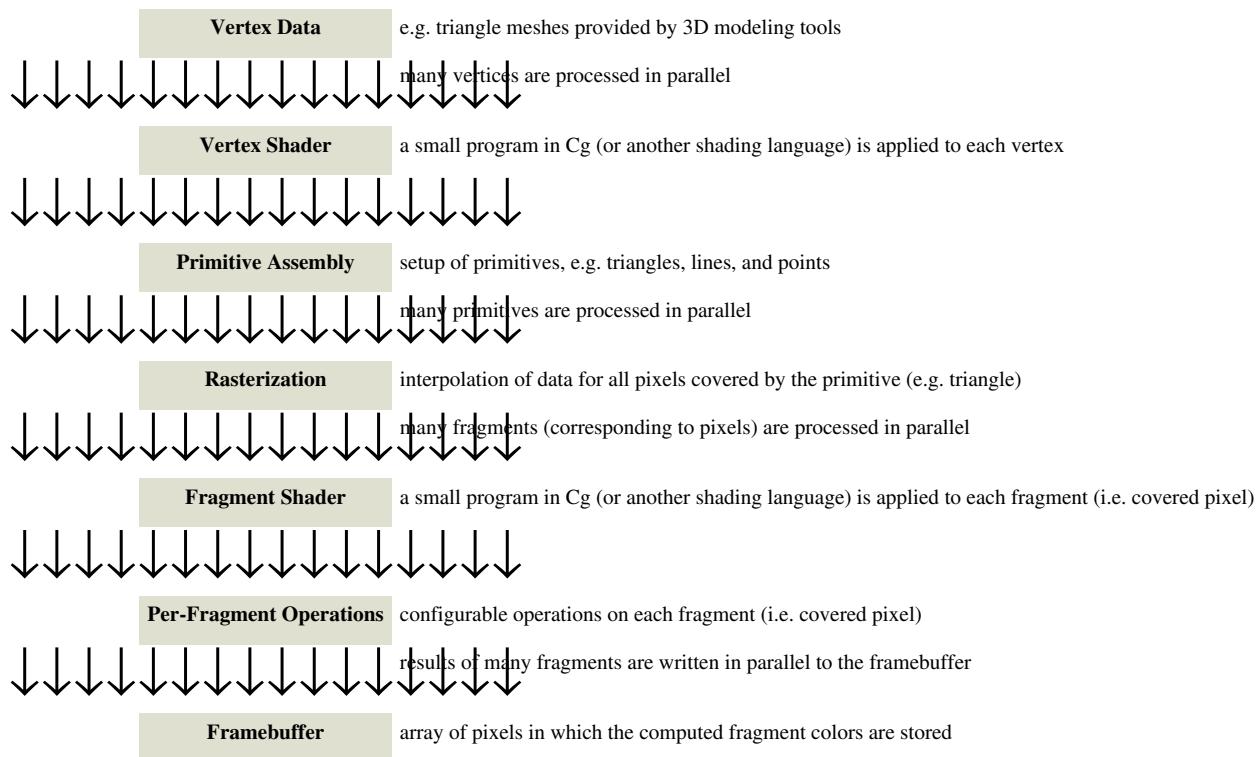
Ford assembly line, 1913.

- **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).



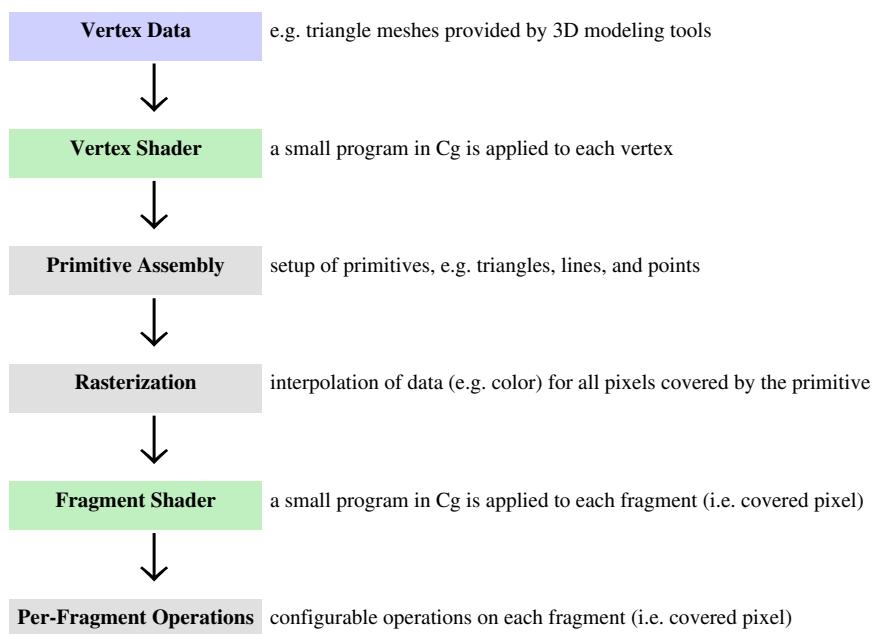
Assembly plant of the Bell Aircraft Corporation with multiple parallel assembly lines, ca. 1944.



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 the graphics pipeline, e.g. Mesa 3D (see the Wikipedia entry), usually implement a single pipeline.

Programmable and Fixed-Function Stages

The pipelines of OpenGL ES 1.x, core OpenGL 1.x, and Direct3D 7.x are configurable fixed-function pipelines, i.e. there is no possibility to include programs in these pipelines. In OpenGL (ES) 2.0, WebGL, and Direct3D 8.0, two stages (the vertex shader and the fragment shader stage) of the pipeline are programmable, i.e. small programs (shaders) written in Cg (or another shading language) 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.



**Framebuffer**

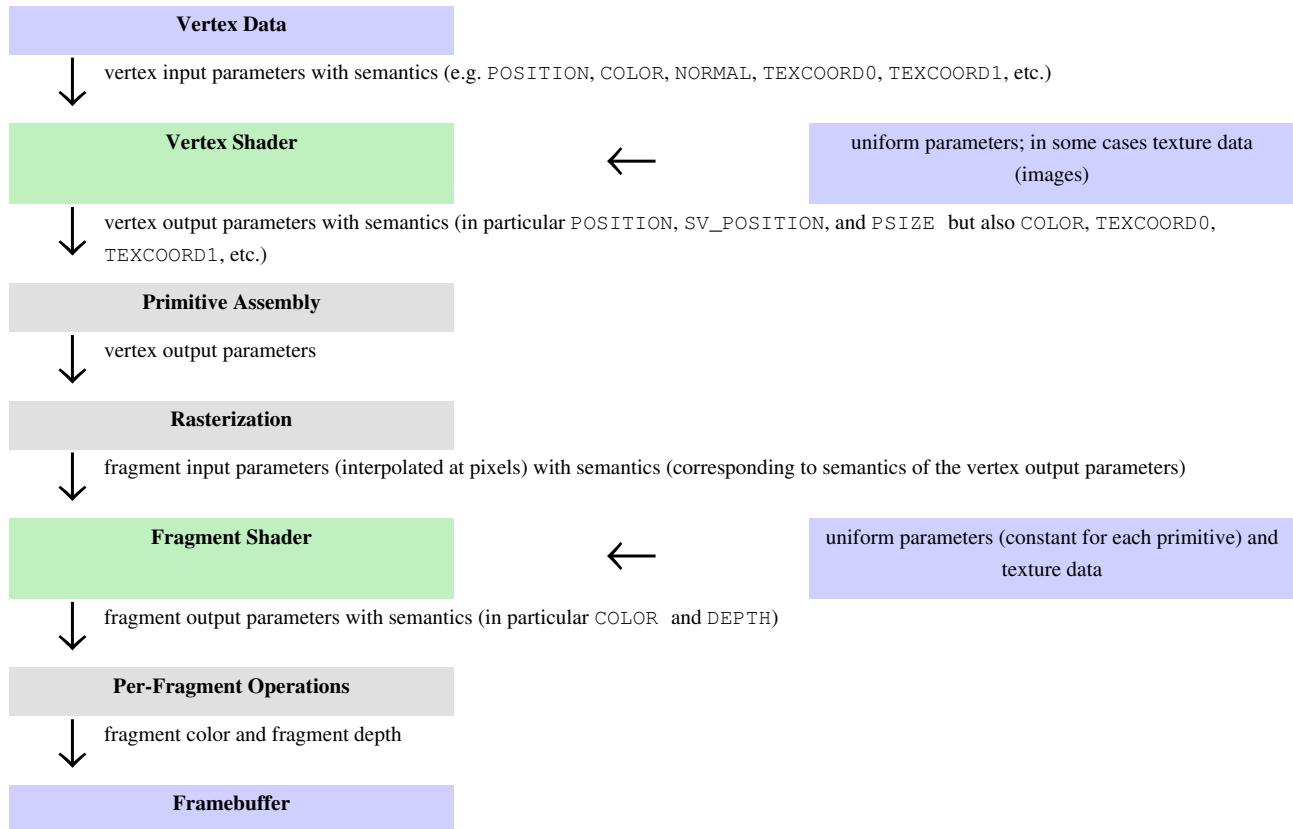
array of pixels in which the computed fragment colors are stored

The vertex shader and fragment shader stages are discussed in more detail in the platform-specific tutorials. The rasterization stage is discussed in Section “Rasterization” and the per-fragment operations in Section “Per-Fragment Operations”.

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.

Data Flow

In order to program Cg vertex and fragment shaders, it is important to understand the input and output of each shader. To this end, it is also useful to understand how data is communicated between all stages of the pipeline. This is illustrated in the next diagram:



Vertex input parameters are defined based on the vertex data. For each vertex input parameter a **semantic** has to be defined, which specifies how the parameter relates to data in the fixed-function pipeline. Examples of semantics are POSITION, COLOR, NORMAL, TEXCOORD0, TEXCOORD1, etc. This makes it possible to use Cg programs even with APIs that were originally designed for a fixed-function pipeline. For example, the vertex input parameter for vertex positions should use the POSITION semantic such that all APIs can provide the appropriate data for this input parameter. Note that the vertex position is in object coordinates, i.e. this is the position as specified in a 3D modeling tool.

Uniform parameters (or uniforms) 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. Usually, they have the same value for a large set of primitives that make up a mesh. Typically, vertex

transformations, specifications of light sources and materials, etc. are specified as uniforms.

Vertex output parameters are computed by the vertex shader, i.e. there is one set of values of these parameters for each vertex. A semantic has to be specified for each parameter, e.g. POSITION, SV_POSITION, COLOR, TEXCOORD0, TEXCOORD1, etc. Usually, there has to be an output parameter with the semantic POSITION or SV_POSITION, which determines where a primitive is rendered on the screen (“SV” stands for “system value” and can have a special meaning). The size of point primitives can be specified by an output parameter with the semantic PSIZE. Other parameters are interpolated (see Section “Rasterization”) for each pixel covered by a primitive.

Fragment input parameters are interpolated from the vertex output parameters for each pixel covered by a primitive. Similar to vertex output parameters, a semantic has to be specified for each fragment input parameter. These semantics are used to match vertex output parameters with fragment input parameters. Therefore, the names of corresponding parameters in the vertex shader and fragment shader can be different as long as the semantics are the same.

Fragment output parameters are computed by fragment shaders. A semantic has to be specified, which determines how the value is used in the following fixed-function pipeline. Most fragment shaders specify an output parameter with the semantic COLOR. The fragment depth is computed implicitly even if no output parameter with the semantic DEPTH is specified.

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.

Further Reading

The model of the programmable graphics pipeline used by Cg is described in the first chapter of Nvidia's Cg Tutorial [3].

page traffic for 90 days ^[1]

< Cg Programming

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] http://stats.grok.se/en.b/latest90/Cg_Programming/Programmable_Graphics_Pipeline

A.2 Vertex Transformations

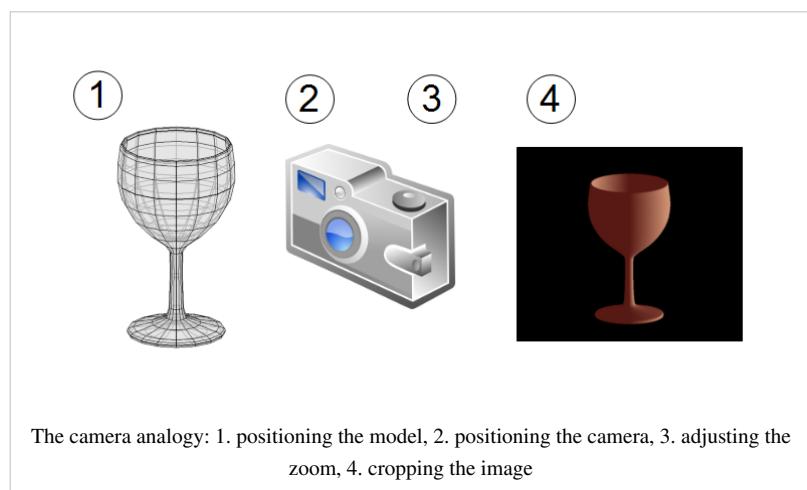
One of the most important tasks of the vertex shader and the following stages in a programmable graphics pipeline is the transformation of vertices of primitives (e.g. triangles) from the original coordinates (e.g. those specified in a 3D modeling tool) to screen coordinates. While programmable vertex shaders allow for many ways of transforming vertices, some transformations are usually 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 parameters and applied to the input 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 Section “[Applying Matrix Transformations](#)”.

Here, we will first present an overview of the coordinate systems and the transformations between them and then discuss individual transformations.

Overview: The Camera Analogy

It is useful to think of the whole process of transforming vertices in terms of a camera analogy as illustrated to the right. The steps and the corresponding vertex transformations are:

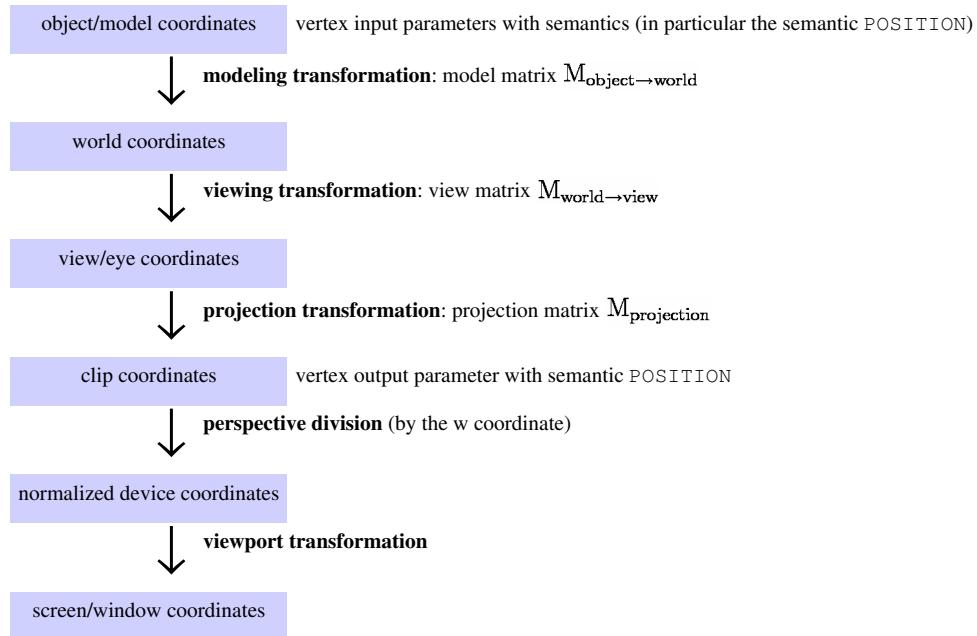
1. positioning the model — modeling transformation
2. positioning the camera — viewing transformation
3. adjusting the zoom — projection transformation
4. cropping the image — viewport transformation



The camera analogy: 1. positioning the model, 2. positioning the camera, 3. adjusting the zoom, 4. cropping the image

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:



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.

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 $M_{\text{object} \rightarrow \text{world}}$. Its structure is:

$$M_{\text{object} \rightarrow \text{world}} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & t_1 \\ a_{2,1} & a_{2,2} & a_{2,3} & t_2 \\ a_{3,1} & a_{3,2} & a_{3,3} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{with } A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \quad \text{and } \mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}$$

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. \mathbf{t} is a 3D vector, which represents a translation (i.e. displacement) in 3D space. $M_{\text{object} \rightarrow \text{world}}$ combines A and \mathbf{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:

$$M_{\text{object} \rightarrow \text{world}} P = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & t_1 \\ a_{2,1} & a_{2,2} & a_{2,3} & t_2 \\ a_{3,1} & a_{3,2} & a_{3,3} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{bmatrix} = \begin{bmatrix} a_{1,1}p_1 + a_{1,2}p_2 + a_{1,3}p_3 + t_1 \\ a_{2,1}p_1 + a_{2,2}p_2 + a_{2,3}p_3 + t_2 \\ a_{3,1}p_1 + a_{3,2}p_2 + a_{3,3}p_3 + t_3 \\ 1 \end{bmatrix}$$

Apart from the fourth coordinate (which is 1 as it should be for a point), the result is equal to

$$A \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}$$

Accessing the Model Matrix in a Vertex Shader

The model matrix $M_{\text{object} \rightarrow \text{world}}$ can be defined as a uniform parameter 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 parameter. In some APIs, the matrix is available as a built-in uniform parameter. (See also Section “Applying Matrix Transformations”.)

Computing the Model Matrix

Strictly speaking, Cg 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 parameter. 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 worry about computing the model matrix. In some cases, however, the model matrix has to be computed when developing graphics application.

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 $t = (t_1, t_2, t_3)$ is:

$$M_{\text{translation}} = \begin{bmatrix} 1 & 0 & 0 & t_1 \\ 0 & 1 & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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:

$$M_{\text{scaling}} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The 4×4 matrix representing the rotation by an angle α about a normalized axis (x, y, z) is:

$$M_{\text{rotation}} = \begin{bmatrix} (1 - \cos \alpha)x^2 + \cos \alpha & (1 - \cos \alpha)x y - z \sin \alpha & (1 - \cos \alpha)z x + y \sin \alpha & 0 \\ (1 - \cos \alpha)x y + z \sin \alpha & (1 - \cos \alpha)y^2 + \cos \alpha & (1 - \cos \alpha)y z - x \sin \alpha & 0 \\ (1 - \cos \alpha)z x - y \sin \alpha & (1 - \cos \alpha)y z + x \sin \alpha & (1 - \cos \alpha)z^2 + \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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 (x_q, y_q, z_q, w_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:

$$M_{\text{combined}} = M_3 M_2 M_1$$

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.

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 (by convention) to the **negative z** axis in OpenGL and to the **positive z** axis in Direct3D, and is put on the xz plane, i.e. the up-direction is given by the positive y axis.

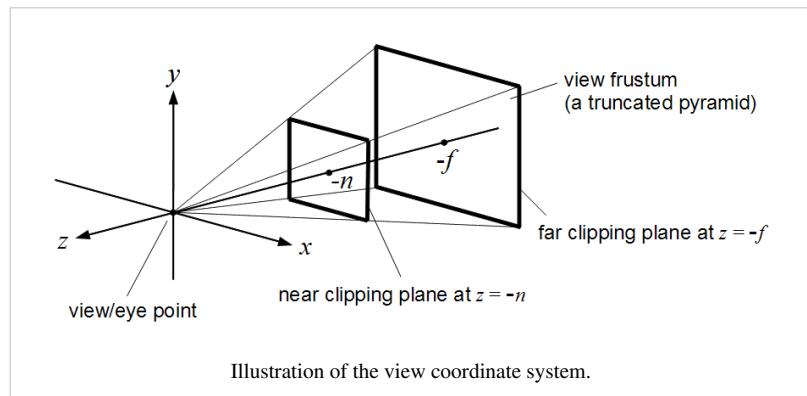


Illustration of the view coordinate system.

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 $M_{\text{world} \rightarrow \text{view}}$. It can be defined as a uniform parameter for the vertex shader; however, it is usually combined with the model matrix $M_{\text{object} \rightarrow \text{world}}$ to form the modelview matrix $M_{\text{object} \rightarrow \text{view}}$. Since the model matrix is applied first, the correct combination is:

$$M_{\text{object} \rightarrow \text{view}} = M_{\text{world} \rightarrow \text{view}} M_{\text{object} \rightarrow \text{world}}$$

(See also Section “Applying Matrix Transformations”.)

Computing the View Matrix

Analogously to the model matrix, Cg 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 parameter. However, when developing graphics applications, it is sometimes necessary to compute the view matrix.

Here, we briefly summarize how the view matrix $M_{\text{world} \rightarrow \text{view}}$ 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). Here we limit us to the case of the right-handed coordinate system of OpenGL where the camera points to the **negative z** axis. (There are some sign changes for Direct3D.) 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 $M_{\text{view} \rightarrow \text{world}}$ 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 $M_{\text{view} \rightarrow \text{world}}$:

$$M_{\text{view} \rightarrow \text{world}} = \begin{bmatrix} x_1 & y_1 & z_1 & t_1 \\ x_2 & y_2 & z_2 & t_2 \\ x_3 & y_3 & z_3 & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

However, we require the matrix $M_{\text{world} \rightarrow \text{view}}$; thus, we have to compute the inverse of the matrix $M_{\text{view} \rightarrow \text{world}}$.

Note that the matrix $M_{\text{viewworld}}$ has the form

$$M_{\text{view} \rightarrow \text{world}} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

with a 3×3 matrix \mathbf{R} and a 3D vector \mathbf{t} . The inverse of such a matrix is:

$$M_{\text{view} \rightarrow \text{world}}^{-1} = M_{\text{world} \rightarrow \text{view}} = \begin{bmatrix} \mathbf{R}^{-1} & -\mathbf{R}^{-1}\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

Since in this particular case the matrix \mathbf{R} is orthogonal (because its column vectors are normalized and orthogonal to each other), the inverse of \mathbf{R} is just the transpose, i.e. the fourth step is to compute:

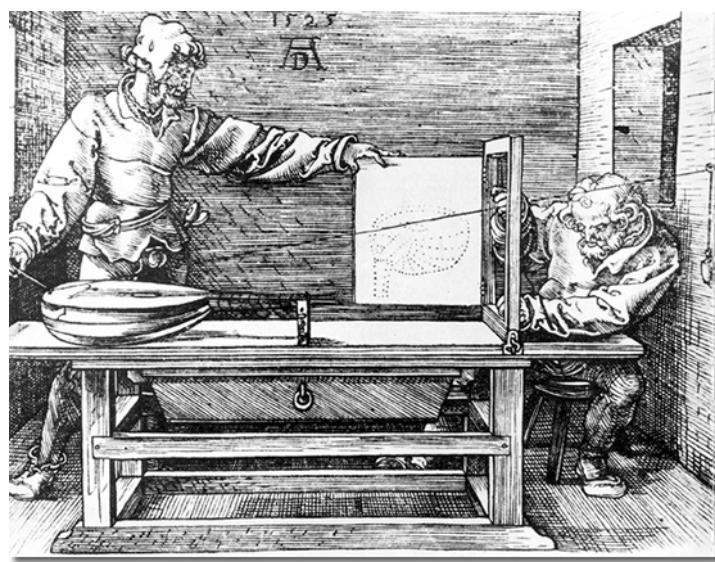
$$M_{\text{world} \rightarrow \text{view}} = \begin{bmatrix} \mathbf{R}^T & -\mathbf{R}^T\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \quad \text{with } \mathbf{R} = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{bmatrix}$$

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.

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 the output parameter with the semantic `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.)



Perspective drawing in the Renaissance: "Man drawing a lute" by Albrecht Dürer, 1525

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 $M_{\text{projection}}$. It is usually defined as a uniform parameter for the vertex shader.

Computing the Projection Matrix

Analogously to the modelview matrix, Cg programmers don't have to worry about the computation of the projection matrix. However, when developing applications, it is sometimes necessary to compute the projection matrix.

Here, we present the projection matrices for three cases (all for the OpenGL convention with a camera pointing to the negative z axis in view coordinates):

- standard perspective projection (corresponds to the OpenGL 2.x function `gluPerspective`)
- oblique perspective projection (corresponds to the OpenGL 2.x function `glFrustum`)
- orthographic projection (corresponds to the OpenGL 2.x function `glOrtho`)

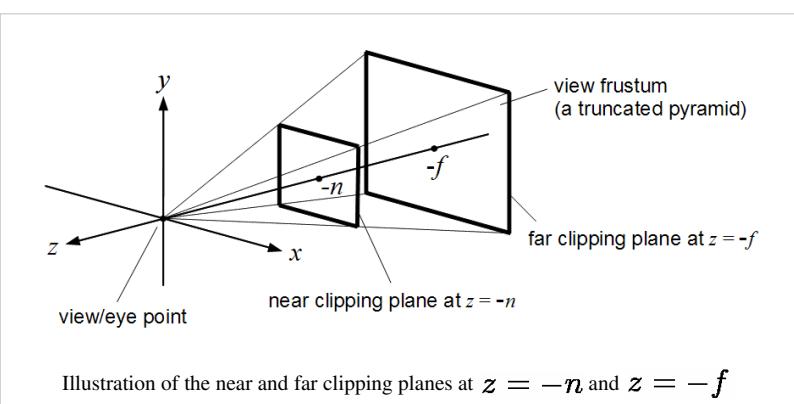
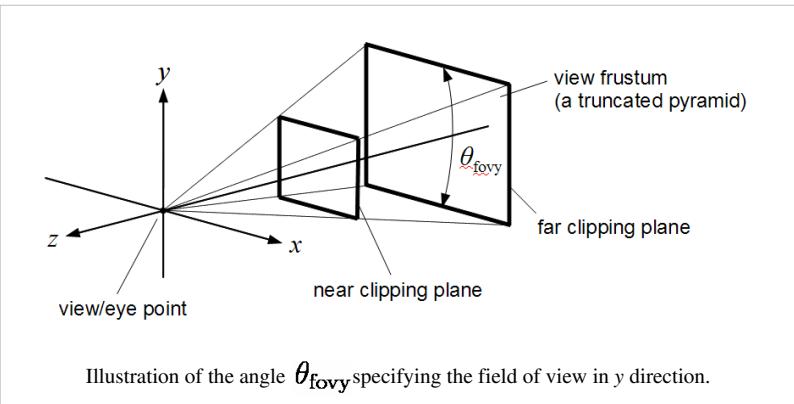
The **standard perspective projection** is characterized by

- an angle θ_{fovy} that specifies the field of view in y direction as illustrated in the figure to the right,
- 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.

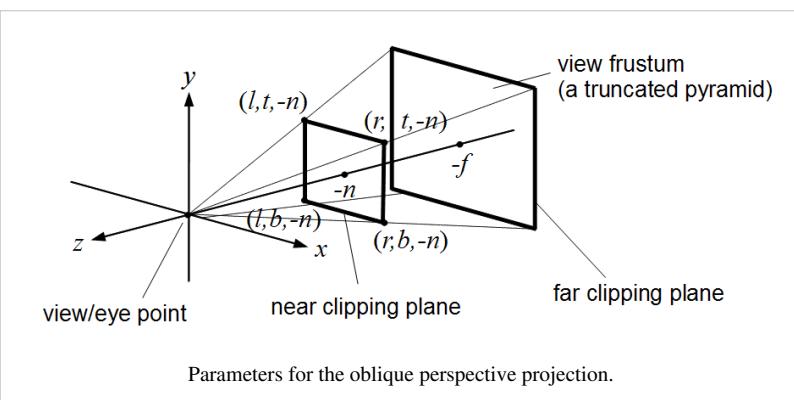
With the parameters θ_{fovy} , a , n , and f , the projection matrix $M_{\text{projection}}$ for the perspective projection is:

$$M_{\text{projection}} = \begin{bmatrix} \frac{d}{a} & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$



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} .



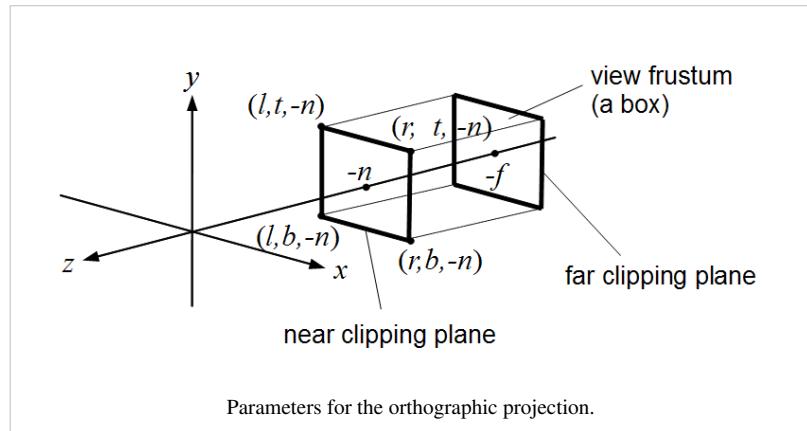
Given the parameters n , f , r , l , t , and b , the projection matrix $M_{\text{projection}}$ for the oblique perspective projection is:

$$M_{\text{projection}} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

An **orthographic projection** without foreshortening is illustrated in the figure to the right. 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 $M_{\text{projection}}$ for the orthographic projection is:

$$M_{\text{projection}} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Viewport Transformation

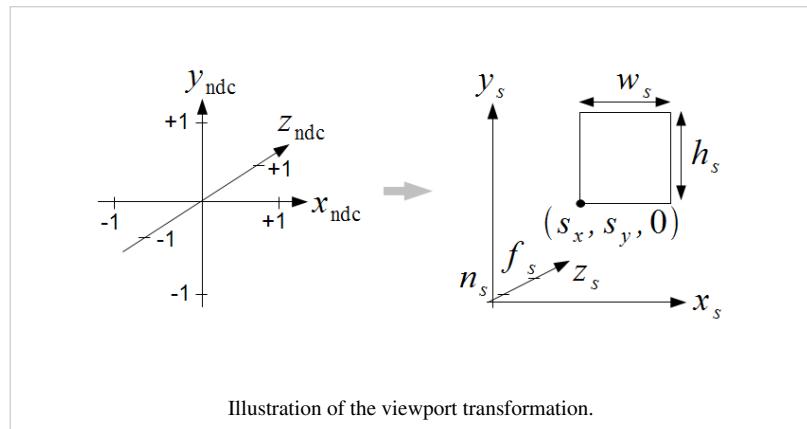
The projection transformation maps view coordinates to clip coordinates, which are then mapped to normalized device coordinates by the perspective division by the fourth component of the clip coordinates. In normalized device coordinates (ndc), the view volume is always a box centered around the origin 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:

$$\begin{bmatrix} \frac{w_s}{2} & 0 & 0 & s_x + \frac{w_s}{2} \\ 0 & \frac{h_s}{2} & 0 & s_y + \frac{h_s}{2} \\ 0 & 0 & \frac{f_s - n_s}{2} & \frac{n_s + f_s}{2} \\ 0 & 0 & 0 & 0 \end{bmatrix}$$



Further Reading

The conventional vertex transformations are also described in less detail in Chapter 4 of Nvidia's Cg Tutorial^[1].

The conventional OpenGL transformations are described in full detail in Section 2.12 of the "OpenGL 4.1 Compatibility Profile Specification" available at the Khronos OpenGL web site^[2]. 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^[3]).

page traffic for 90 days^[4]

< Cg Programming

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

- [1] http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter04.html
- [2] <http://www.khronos.org/opengl/>
- [3] <http://www.opengl.org/red/chapter03.html>
- [4] http://stats.grok.se/en.b/latest90/Cg_Programming/Vertex_Transformations

A.3 Vector and Matrix Operations

The syntax of Cg 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 Cg. These are discussed here. A full description of Cg can be found in Nvidia's Cg Tutorial^[3] and in Nvidia's Cg Language Specification^[4].

Floating-Point Data Types with Full Precision

In Cg, the types `float2`, `float3`, and `float4` represent 2D, 3D, and 4D floating-point vectors. Vector variables are defined as you would expect if C, C++ or Java had these types:

```
float2 a2DVector;
float3 three_dimensional_vector;
float4 vector4;
```

The data types for floating-point 2×2, 3×3, and 4×4 matrices are: `float2x2`, `float3x3`, and `float4x4`:

```
float2x2 m2x2;
float3x3 linear_mapping;
float4x4 trafo;
```

There are also types for matrices with different numbers of rows and columns, e.g. 3 rows and 4 columns: `float3x4`.

Data Types with Limited Precision

Apart from `float` data types, there are additional `half` types (`half`, `half2`, `half3`, `half4`, `half2x2`, `half3x3`, `half4x4`, etc.), and `fixed` types (`fixed`, `fixed2`, `fixed3`, `fixed4`, etc.), which represent numbers (i.e. scalars), vectors, and matrices of limited precision and range. Usually they provide better performance; thus, for best performance you should use one of the `half` types instead of the corresponding `float` types (in particular if the data represents geometric positions or texture coordinates) or even one of the `fixed` types (usually

if the data represents colors) if the limited precision and range does not result in rendering artifacts.

Here, the code will always use the `float` types to avoid any problems related to limited precision and to keep it as simple as possible. There are also types for integer and boolean vectors, which will not be discussed here.

Constructors

Vectors can be initialized and converted by constructors of the same name as the data type:

```
float2 a = float2(1.0, 2.0);
float3 b = float3(-1.0, 0.0, 0.0);
float4 c = float4(0.0, 0.0, 0.0, 1.0);
```

Note that some Cg compilers might complain if integers are used to initialize floating-point vectors; thus, it is good practice to always include the decimal point.

One can also use one floating-point number in the constructor to set all components to the same value:

```
float4 a = float4(0.0); // = float4(0.0, 0.0, 0.0, 0.0)
```

Casting a higher-dimensional vector to a lower-dimensional vector is also achieved with these constructors:

```
float4 a = float4(-1.0, 2.5, 4.0, 1.0);
float3 b = float3(a); // = float3(-1.0, 2.5, 4.0)
float2 c = float2(b); // = float2(-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:

```
float2 a = float2(0.1, 0.2);
float3 b = float3(0.0, a); // = float3(0.0, 0.1, 0.2)
float4 c = float4(b, 1.0); // = float4(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 row, then the second row, etc.:

```
float3x3 m = float3x3(
    1.1, 1.2, 1.3, // first row (not column as in GLSL!)
    2.1, 2.2, 2.3, // second row
    3.1, 3.2, 3.3 // third row
);
float3 row0 = float3(0.0, 1.0, 0.0);
float3 row1 = float3(1.0, 0.0, 0.0);
float3 row2 = float3(0.0, 0.0, 1.0);
float3x3 n = float3x3(row0, row1, row2); // sets rows of matrix n
```

If a smaller matrix is constructed from a larger matrix, the top, left submatrix of the larger matrix is chosen:

```
float3x3 m = float3x3(
    1.1, 1.2, 1.3,
    2.1, 2.2, 2.3,
    3.1, 3.2, 3.3
);
float2x2 n = float2x2(m); // = float2x2(1.1, 1.2, 2.1, 2.2)
```

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`:

```
float4 v = float4(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:

```
float4 v = float4(1.1, 2.2, 3.3, 4.4);
float3 a = v.xyz; // = float3(1.1, 2.2, 3.3)
float3 b = v.bgr; // = float3(3.3, 2.2, 1.1)
float2 c = v.tt; // = float2(2.2, 2.2)
```

Matrices are considered to consist of row vectors, which are accessed by array indexing with the []-operator. Elements of the resulting (row) vector can be accessed as discussed above:

```
float3x3 m = float3x3(
    1.1, 1.2, 1.3, // first row
    2.1, 2.2, 2.3, // second row
    3.1, 3.2, 3.3 // third row
);
float3 row2 = m[2]; // = float3(3.1, 3.2, 3.3)
float m20 = m[2][0]; // = 3.1
float m21 = m[2].y; // = 3.2
```

Operators

If the binary operators `*`, `/`, `+`, `-`, `=`, `*=`, `/=`, `+=`, `-=` are used between vectors of the same type, they just work component-wise:

```
float3 a = float3(1.0, 2.0, 3.0);
float3 b = float3(0.1, 0.2, 0.3);
float3 c = a + b; // = float3(1.1, 2.2, 3.3)
float3 d = a * b; // = float3(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. Again, a component-wise product of two matrices is not often seen in linear algebra.

For the usual matrix-vector product or matrix-matrix product, see the built-in `mul` function below. For the dot product and cross product between vectors, see the built-in functions `dot` and `cross` below.

The `*`-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):

```
float3 a = float3(1.0, 2.0, 3.0);
float2x2 m = float2x2(1.0, 0.0, 0.0, 1.0);
float s = 10.0;
float3 b = s * a; // float3(10.0, 20.0, 30.0)
```

```
float3 c = a * s; // float3(10.0, 20.0, 30.0)
float2x2 m2 = s * m; // = float2x2(10.0, 0.0, 0.0, 10.0)
float2x2 m3 = m * s; // = float2x2(10.0, 0.0, 0.0, 10.0)
```

Built-In Vector and Matrix Functions

Component-Wise Functions

The following functions work component-wise for variables of type `float`, `float2`, `float3`, `float4`, `half`, `half2`, `half3`, `half4`, `fixed`, `fixed2`, `fixed3`, and `fixed4`, which are denoted as `TYPE`:

```
TYPE min(TYPE a, TYPE b) // returns a if a < b, b otherwise
TYPE max(TYPE a, TYPE b) // returns a if a > b, b otherwise
TYPE clamp(TYPE a, TYPE minValue, TYPE maxValue)
    // = min(max(x, minValue), maxValue)
TYPE lerp(TYPE a, TYPE b, TYPE wb) // = a * (TYPE(1.0) - wb) + b * wb
TYPE lerp(TYPE a, TYPE b, float wb) // = a * TYPE(1.0 - wb) + b *
TYPE(wb)
```

There are more built-in functions, which also work component-wise but are less useful for vectors, e.g., `abs`, `sign`, `floor`, `ceil`, `round`, `frac`, `fmod`, `step`, `smoothstep`, `sqrt`, `pow`, `exp`, `exp2`, `log`, `log2`, `radians` (converts degrees to radians), `degrees` (converts radians to degrees), `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`.

Matrix Functions

For the usual matrix-matrix product in linear algebra, the `mul` function should be used. It computes components of the resulting matrix by multiplying a row (of the first matrix) with a column (of the second matrix), e.g.:

$$AB = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} = \begin{bmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} & a_{1,1}b_{1,2} + a_{1,2}b_{2,2} \\ a_{2,1}b_{1,1} + a_{2,2}b_{2,1} & a_{2,1}b_{1,2} + a_{2,2}b_{2,2} \end{bmatrix}$$

And in Cg:

```
float2x2 a = float2x2(1., 2., 3., 4.);
float2x2 b = float2x2(10., 20., 30., 40.);
float2x2 c = mul(a, b); // = float2x2(
    // 1. * 10. + 2. * 30., 1. * 20. + 2. * 40.,
    // 3. * 10. + 4. * 30., 3. * 20. + 4. * 40.)
```

Furthermore, the `mul` function can be used for matrix-vector products of the corresponding dimension, e.g.:

$$Mv = \begin{bmatrix} m_{1,1} & m_{1,2} \\ m_{2,1} & m_{2,2} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} m_{1,1}v_1 + m_{1,2}v_2 \\ m_{2,1}v_1 + m_{2,2}v_2 \end{bmatrix}$$

And in Cg:

```
float2 v = float2(10., 20.);
float2x2 m = float2x2(1., 2., 3., 4.);
float2 w = mul(m, v); // = float2x2(1. * 10. + 2. * 20., 3. * 10. + 4. * 20.)
```

Note that the vector has to be the second argument to be multiplied to the matrix from the right.

If a vector is specified as first argument, it is multiplied to a matrix **from the left**:

```
float2 v = float2(10., 20.);
float2x2 m = float2x2(1., 2., 3., 4.);
float2 w = mul(v, m); // = float(10. * 1. + 20. * 3., 10. * 2. + 20. *
4.)
```

Multiplying a row vector from the left to a matrix corresponds to multiplying a column vector to the **transposed** matrix from the right:

$$\mathbf{v}^T \mathbf{M} = (\mathbf{M}^T \mathbf{v})^T$$

In components:

$$\begin{aligned} \mathbf{v}^T \mathbf{M} &= [v_1 \quad v_2] \begin{bmatrix} m_{1,1} & m_{1,2} \\ m_{2,1} & m_{2,2} \end{bmatrix} = [v_1 m_{1,1} + v_2 m_{2,1} \quad v_1 m_{1,2} + v_2 m_{2,2}] \\ &= \left(\begin{bmatrix} m_{1,1} & m_{2,1} \\ m_{1,2} & m_{2,2} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \right)^T = (\mathbf{M}^T \mathbf{v})^T \end{aligned}$$

Thus, multiplying a vector from the left to a matrix corresponds to multiplying it from the right to the transposed matrix without explicitly computing the transposed matrix.

However, there is also a function `transpose` to compute the transposed matrix:

```
float2x2 m = float2x2(1., 2., 3., 4.);
float2x2 n = transpose(m); // = float2x2(1., 3., 2., 4)
```

Geometric Functions

The following functions are particular useful for vector operations. `TYPE` is any of: `float`, `float2`, `float3`, `float4`, `half`, `half2`, `half3`, `half4`, `fixed`, `fixed2`, `fixed3`, and `fixed4` (only one of them per line).

```
float3 cross(float3 a, float3 b)
    // = float3(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), i.e. if the ray does not pass the interface between the two materials.

Further Reading

A full description of Cg can be found in Nvidia's Cg Tutorial ^[3] (including all standard library functions in Appendix E ^[1]) and in Nvidia's Cg Language Specification ^[4].

page traffic for 90 days ^[1]

< Cg Programming

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] http://stats.grok.se/en.b/latest90/Cg_Programming/Vector_and_Matrix_Operations

A.4 Applying Matrix Transformations

Applying the conventional vertex transformations (see Section "Vertex Transformations") 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 coordinate 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 Cg as described in Section "Vector and Matrix Operations".

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:

$$\mathbf{M} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & t_1 \\ a_{2,1} & a_{2,2} & a_{2,3} & t_2 \\ a_{3,1} & a_{3,2} & a_{3,3} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{with } \mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \quad \text{and } \mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}$$

(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:

$$\mathbf{P} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{bmatrix}$$

In order to apply the transformation, the matrix \mathbf{M} is multiplied to the vector \mathbf{P} :

$$M P = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & t_1 \\ a_{2,1} & a_{2,2} & a_{2,3} & t_2 \\ a_{3,1} & a_{3,2} & a_{3,3} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{bmatrix} = \begin{bmatrix} a_{1,1}p_1 + a_{1,2}p_2 + a_{1,3}p_3 + t_1 \\ a_{2,1}p_1 + a_{2,2}p_2 + a_{2,3}p_3 + t_2 \\ a_{3,1}p_1 + a_{3,2}p_2 + a_{3,3}p_3 + t_3 \\ 1 \end{bmatrix}$$

The Cg code to apply a 4×4 matrix to a point represented by a 4D vector is straightforward:

```
float4x4 mymatrix;
float4 point;
float4 transformed_point = mul(mymatrix, point);
```

If a transposed matrix is given, one could apply the function `transpose` to compute the required matrix. However, it is more common to multiply the vector from the left to the transposed matrix as mentioned in Section “Vector and Matrix Operations”:

```
float4x4 matrix_transpose;
float4 point;
float4 transformed_point = mul(point, matrix_transpose);
```

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:

```
float3x3 mymatrix;
float3 direction;
float3 transformed_direction = mul(mymatrix, direction);
```

or with a 4×4 matrix, if we convert the 3D vector to a 4D vector with a 4th coordinate equal to 0:

```
float4x4 mymatrix;
float3 direction;
float4 transformed_direction = float4(mul(mymatrix, float4(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:

```
float3x3 mymatrix;
float4 direction; // 4th component is 0
float4 transformed_direction = float4(mul(mymatrix, float3(direction)),
0.0);
```

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 \mathbf{A} , the normal vector has to be transformed by the **transposed inverse** of \mathbf{A} : $(\mathbf{A}^{-1})^T$. We can easily test this by checking the dot product of the transformed normal vector $(\mathbf{A}^{-1})^T \mathbf{n}$ and the transformed direction vector $\mathbf{A} \mathbf{v}$:

$$(\mathbf{A}^{-1})^T \mathbf{n} \cdot \mathbf{A} \mathbf{v} = \left((\mathbf{A}^{-1})^T \mathbf{n} \right)^T \mathbf{A} \mathbf{v} = \left(\mathbf{n}^T \left((\mathbf{A}^{-1})^T \right)^T \right) \mathbf{A} \mathbf{v} = (\mathbf{n}^T \mathbf{A}^{-1}) \mathbf{A} \mathbf{v} =$$

$$\mathbf{n}^T \mathbf{A}^{-1} \mathbf{A} \mathbf{v} = \mathbf{n}^T \mathbf{v} = \mathbf{n} \cdot \mathbf{v}$$

In the first step we have used $\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T \mathbf{b}$, then $(\mathbf{M}\mathbf{a})^T = \mathbf{a}^T \mathbf{M}^T$, then $(\mathbf{M}^T)^T = \mathbf{M}$, then $\mathbf{M}^{-1} \mathbf{M} = \text{Id}$ (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 Cg, 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:

```
float3x3 matrix_inverse_transpose;
float3 normal;
float3 transformed_normal = mul(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:

```
float4x4 matrix_inverse_transpose;
float3 normal;
float3 transformed_normal = float3(mul(matrix_inverse_transpose,
float4(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 as mentioned in Section “Vector and Matrix Operations”. Thus, in order to multiply a normal vector with the transposed inverse matrix, we can multiply it from the left to the inverse matrix:

```
float3x3 matrix_inverse;
float3 normal;
float3 transformed_normal = mul(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):

```

float4x4 matrix_inverse;
float4 normal;
float4 transformed_normal = float4(float3(mul(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 Cg function `normalize`).

Transforming Normal Vectors with an Orthogonal Matrix

A special case arises when the transformation matrix \mathbf{A} is orthogonal. In this case, the inverse of \mathbf{A} is the transposed matrix; thus, the transposed of the inverse of \mathbf{A} is the twice transposed matrix, which is the original matrix, i.e. for a orthogonal matrix \mathbf{A} :

$$(\mathbf{A}^{-1})^T = (\mathbf{A}^T)^T = \mathbf{A}$$

Thus, in the case of **orthogonal** matrices, normal vectors are transformed with the same matrix as directions and points:

```

float3x3 mymatrix; // orthogonal matrix
float3 normal;
float3 transformed_normal = mul(mymatrix, normal);

```

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 \mathbf{M} is of the form presented above (i.e. the 4th row is $(0,0,0,1)$):

$$\mathbf{M} = \begin{bmatrix} \mathbf{A} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

with an orthogonal 3×3 matrix \mathbf{A} (i.e. the row (or column) vectors of \mathbf{A} are normalized and orthogonal to each other; for example, this is usually the case for the view transformation, see Section “Vertex Transformations”), then the inverse matrix is given by (because $\mathbf{A}^{-1} = \mathbf{A}^T$ for an orthogonal matrix \mathbf{A}):

$$\mathbf{M}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} & -\mathbf{A}^{-1}\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{A}^T & -\mathbf{A}^T\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

For the multiplication with a point \mathbf{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:

$$\mathbf{M}^{-1}\mathbf{P} = \begin{bmatrix} \mathbf{A}^T & -\mathbf{A}^T\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{A}^T\mathbf{p} - \mathbf{A}^T\mathbf{t} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{A}^T(\mathbf{p} - \mathbf{t}) \\ 1 \end{bmatrix}$$

Note that the vector \mathbf{t} is just the 4th column of the matrix \mathbf{M} , which could be accessed this way::

```

float4x4 m;
float4 last_column = float4(m[0][3], m[1][3], m[2][3], m[3][3]);

```

As mentioned above, multiplying a transposed matrix with a vector can be easily expressed by multiplying the vector from the left to the matrix and corresponds to a matrix-vector product of the transposed matrix with the corresponding column vector:

$$\mathbf{p}^T \mathbf{A} = (\mathbf{A}^T \mathbf{p})^T$$

Using these features of Cg, 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):

```
float4x4 m; // upper, left 3x3 matrix is orthogonal;
// 4th row is (0, 0, 0, 1)
float4 point; // 4th component is 1
float4 last_column = float4(m[0][3], m[1][3], m[2][3], m[3][3]);
float4 point_transformed_with_inverse = float4(float3(mul(point -
last_column, m)), 1.0);
```

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

$$M = \begin{bmatrix} A & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

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 Cg 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:

$$\mathbf{p}^T A = (A^T \mathbf{p})^T$$

Thus, the transformation with the transposed matrix (i.e. the inverse in case of a orthogonal matrix) is written as:

```
float4x4 mymatrix; // upper, left 3x3 matrix is orthogonal
float4 direction; // 4th component is 0
float4 direction_transformed_with_inverse =
float4(float3(mul(direction, mymatrix)), 0.0);
```

Note that the 4th component of the result has to be set to 0 separately since the 4th component of $\text{mul}(\text{direction}, \text{mymatrix})$ 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 3×3 matrices and $3D$ vectors only require different cast operations between $3D$ and $4D$ vectors.

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):

```
float4x4 matrix_inverse;
float3 normal;
float3 transformed_normal = float3(mul(float4(normal, 0.0),
matrix_inverse));
```

(or by casting the matrix).

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 [2].

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 [1].

page traffic for 90 days [2]

< Cg Programming

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] <http://www.glprogramming.com/red/appendixe.html>

[2] http://stats.grok.se/en.b/latest90/Cg_Programming/Applying_Matrix_Transformations

A.5 Rasterization

Rasterization is the stage of the graphics pipeline that determines the pixels covered by a primitive (e.g. a triangle) and interpolates the output parameters of the vertex shader (in particular depth) for each covered pixel. The interpolated output parameters 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.)

Usually, it is not necessary for Cg 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 perspective-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.

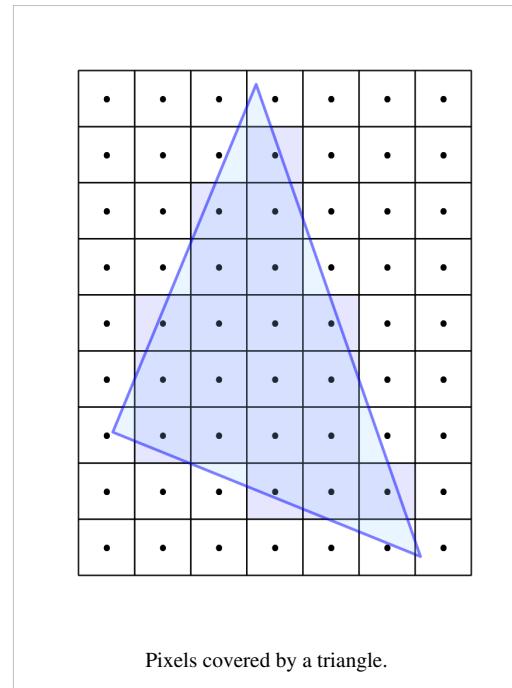
The main two parts of the rasterization are:

- Determining the pixels that are covered by a primitive (e.g. a triangle)
- Linear interpolation of output parameters of the vertex shader and depth for all covered pixels

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 to the right.

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. Moreover, other APIs than OpenGL may specify different rules. Thus, they won't be discussed here.



Linear Interpolation of Parameters

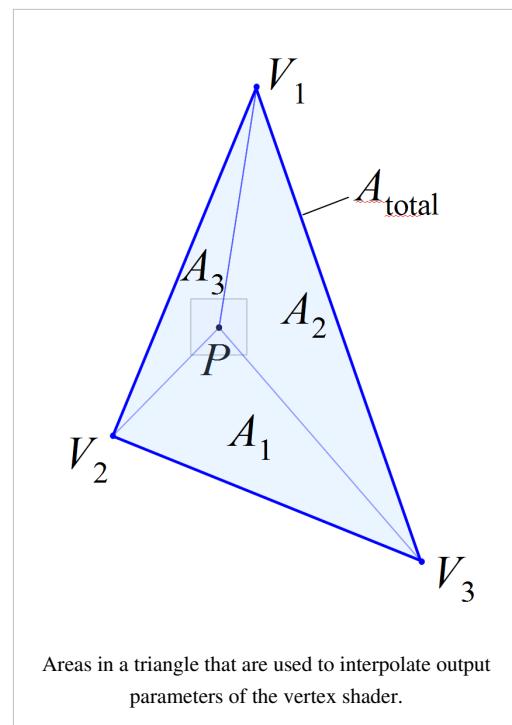
Once all the covered pixels are determined, the output parameters of the vertex shader are interpolated for each pixel. For simplicity, we will only discuss the case of a triangle. (A line works like a triangle with two vertices at the same position.)

For each triangle, the vertex shader computes the positions of the three vertices. In the diagram to the right, these positions are labeled as V_1 , V_2 , and V_3 . The vertex shader also computes values of output parameters at each vertex. We denote one of them as f_1 , f_2 , and f_3 . Note that these values refer to the same output parameter computed at different vertices. The position of the center of the pixel for which we want to interpolate the output parameter 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:

$$\text{area of triangle } QRS = \frac{1}{2} |\vec{QR} \times \vec{QS}|$$

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:

$$f_P = \alpha_1 f_1 + \alpha_2 f_2 + \alpha_3 f_3$$

This way, all output parameters can be linearly interpolated for all covered pixels.

Perspectively Correct Interpolation of Parameters

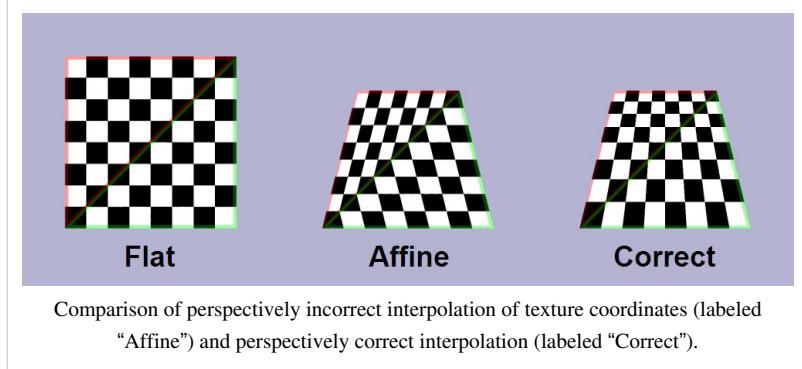
The interpolation described in the previous section can result in certain distortions if applied to scenes that use perspective projection. For the perspectively correct interpolation the distance to the view point is placed in the fourth component of the three vertex positions (w_1 , w_2 , and w_3)

and the following equation is used for the interpolation:

$$f_P = \frac{\alpha_1 f_1 / w_1 + \alpha_2 f_2 / w_2 + \alpha_3 f_3 / w_3}{\alpha_1 / w_1 + \alpha_2 / w_2 + \alpha_3 / w_3}$$

Thus, the fourth component of the position of the vertices is important for perspectively correct interpolation of output parameters. 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 implementations of graphics pipelines 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.



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”^[1].

page traffic for 90 days^[2]

< Cg Programming

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] <http://www.khronos.org/registry/gles/>

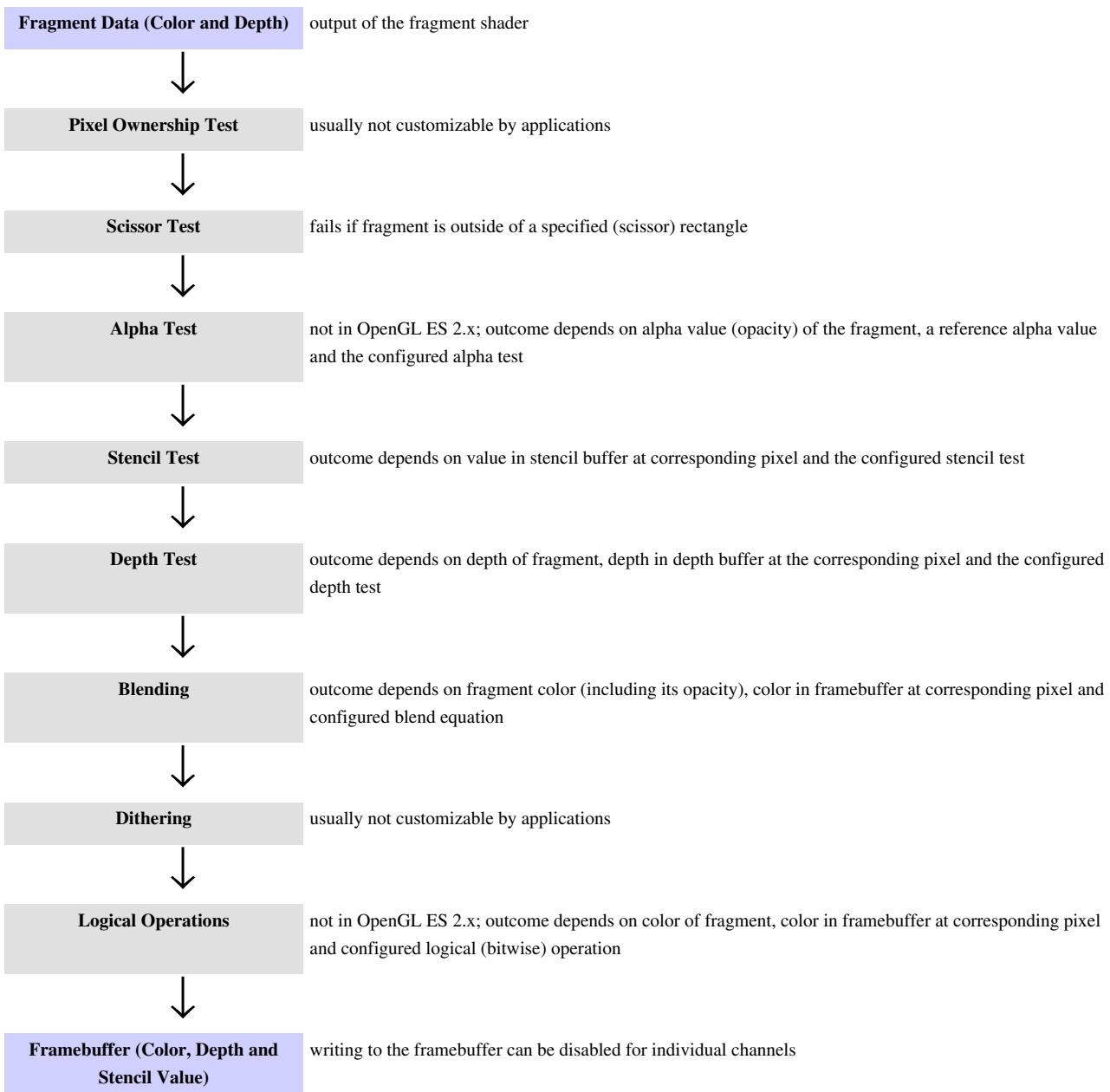
[2] http://stats.grok.se/en.b/latest90/Cg_Programming/Rasterization

A.6 Per-Fragment Operations

The per-fragment operations are part of graphics pipelines and usually 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.



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 specifications of APIs such as OpenGL and Direct3D impose 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.

Programmability of Per-Fragment Operations

Per-fragment operations are usually configurable but not programmable, i.e. the various tests can be configured in certain ways but you cannot write a program to compute the tests. However, this is changing. For example, OpenGL ES 2.0 offers no alpha test because the same functionality can be achieved by a conditional discard operation in a fragment shader, which is effectively a programmable alpha test.

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”^[1].

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^[1]).

page traffic for 90 days^[2]

< Cg Programming

Unless stated otherwise, all example source code on this page is granted to the public domain.

References

[1] <http://www.opengles-book.com/>

[2] http://stats.grok.se/en.b/latest90/Cg_Programming/Per-Fragment_Operations

Article Sources and Contributors

Introduction *Source:* <http://en.wikibooks.org/w/index.php?oldid=2505585> *Contributors:* Martin Kraus

1.1 Minimal Shader *Source:* <http://en.wikibooks.org/w/index.php?oldid=2400619> *Contributors:* Martin Kraus

1.2 RGB Cube *Source:* <http://en.wikibooks.org/w/index.php?oldid=2437933> *Contributors:* Benregn, Martin Kraus

1.3 Debugging of Shaders *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395577> *Contributors:* Martin Kraus

1.4 Shading in World Space *Source:* <http://en.wikibooks.org/w/index.php?oldid=2419963> *Contributors:* Martin Kraus

2.1 Cutaways *Source:* <http://en.wikibooks.org/w/index.php?oldid=2415316> *Contributors:* Martin Kraus, 1 anonymous edits

2.2 Transparency *Source:* <http://en.wikibooks.org/w/index.php?oldid=2388634> *Contributors:* Martin Kraus

2.3 Order-Independent Transparency *Source:* <http://en.wikibooks.org/w/index.php?oldid=2388721> *Contributors:* Martin Kraus

2.4 Silhouette Enhancement *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395601> *Contributors:* Martin Kraus

3.1 Diffuse Reflection *Source:* <http://en.wikibooks.org/w/index.php?oldid=2414248> *Contributors:* Martin Kraus

3.2 Specular Highlights *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395599> *Contributors:* Martin Kraus

3.3 Two-Sided Surfaces *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395597> *Contributors:* Martin Kraus

3.4 Smooth Specular Highlights *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395596> *Contributors:* Martin Kraus

3.5 Two-Sided Smooth Surfaces *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395595> *Contributors:* Martin Kraus

3.6 Multiple Lights *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395603> *Contributors:* Martin Kraus

4.1 Textured Spheres *Source:* <http://en.wikibooks.org/w/index.php?oldid=2393314> *Contributors:* Martin Kraus

4.2 Lighting Textured Surfaces *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395615> *Contributors:* Martin Kraus

4.3 Glossy Textures *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395616> *Contributors:* Martin Kraus

4.4 Transparent Textures *Source:* <http://en.wikibooks.org/w/index.php?oldid=2393439> *Contributors:* Martin Kraus

4.5 Layers of Textures *Source:* <http://en.wikibooks.org/w/index.php?oldid=2464185> *Contributors:* Martin Kraus, 2 anonymous edits

5.1 Lighting of Bumpy Surfaces *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395639> *Contributors:* Martin Kraus

5.2 Projection of Bumpy Surfaces *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395642> *Contributors:* Martin Kraus

5.3 Cookies *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395646> *Contributors:* Martin Kraus

5.4 Light Attenuation *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395649> *Contributors:* Martin Kraus

5.5 Projectors *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395650> *Contributors:* Martin Kraus

6.1 Reflecting Surfaces *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395661> *Contributors:* Martin Kraus

6.2 Curved Glass *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395662> *Contributors:* Martin Kraus

6.3 Skyboxes *Source:* <http://en.wikibooks.org/w/index.php?oldid=2444021> *Contributors:* Martin Kraus

6.4 Many Light Sources *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395665> *Contributors:* Martin Kraus

7.1 Brushed Metal *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395672> *Contributors:* Martin Kraus

7.2 Specular Highlights at Silhouettes *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395673> *Contributors:* Martin Kraus

7.3 Diffuse Reflection of Skylight *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395675> *Contributors:* Martin Kraus

7.4 Translucent Surfaces *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395677> *Contributors:* Martin Kraus

7.5 Translucent Bodies *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395678> *Contributors:* Martin Kraus

7.6 Soft Shadows of Spheres *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395679> *Contributors:* Martin Kraus

7.7 Toon Shading *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395680> *Contributors:* Martin Kraus

8.1 Screen Overlays *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395147> *Contributors:* Martin Kraus

8.2 Billboards *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395163> *Contributors:* Martin Kraus

8.3 Nonlinear Deformations *Source:* <http://en.wikibooks.org/w/index.php?oldid=2395681> *Contributors:* Martin Kraus

8.4 Shadows on Planes *Source:* <http://en.wikibooks.org/w/index.php?oldid=2417718> *Contributors:* Martin Kraus

8.5 Mirrors *Source:* <http://en.wikibooks.org/w/index.php?oldid=2417717> *Contributors:* Martin Kraus

9.1 Rotations *Source:* <http://en.wikibooks.org/w/index.php?oldid=2430671> *Contributors:* Martin Kraus, QuiteUnusual

9.2 Projection for Virtual Reality *Source:* <http://en.wikibooks.org/w/index.php?oldid=2511055> *Contributors:* Martin Kraus

9.3 Bézier Curves *Source:* <http://en.wikibooks.org/w/index.php?oldid=2518243> *Contributors:* Martin Kraus, QuiteUnusual

9.4 Hermite Curves *Source:* <http://en.wikibooks.org/w/index.php?oldid=2430681> *Contributors:* Martin Kraus, QuiteUnusual

A.1 Programmable Graphics Pipeline *Source:* <http://en.wikibooks.org/w/index.php?oldid=2419923> *Contributors:* Martin Kraus, 1 anonymous edits

A.2 Vertex Transformations *Source:* <http://en.wikibooks.org/w/index.php?oldid=2422680> *Contributors:* Martin Kraus

A.3 Vector and Matrix Operations *Source:* <http://en.wikibooks.org/w/index.php?oldid=2505344> *Contributors:* Martin Kraus, 2 anonymous edits

A.4 Applying Matrix Transformations *Source:* <http://en.wikibooks.org/w/index.php?oldid=2420040> *Contributors:* Martin Kraus

A.5 Rasterization *Source:* <http://en.wikibooks.org/w/index.php?oldid=2387553> *Contributors:* Martin Kraus

A.6 Per-Fragment Operations *Source:* <http://en.wikibooks.org/w/index.php?oldid=2387560> *Contributors:* Martin Kraus

Image Sources, Licenses and Contributors

Image:Sphere with soft shadow.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Sphere_with_soft_shadow.jpg *License:* Creative Commons Attribution-Sharealike 3.0
Contributors: User:Martin Kraus

File:RGB color solid cube.png *Source:* http://en.wikibooks.org/w/index.php?title=File:RGB_color_solid_cube.png *License:* Creative Commons Attribution-Share Alike *Contributors:* SharkD

File:Inundaciones en Argentina Argentina.A2003128.1420.721.250m.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Inundaciones_en_Argentina_Argentina.A2003128.1420.721.250m.jpg *License:* Public Domain *Contributors:* NASA

File:Chamäleon1.jpg *Source:* <http://en.wikibooks.org/w/index.php?title=File:Chamäleon1.jpg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Hans Bernhard (Schnobby)

File:Filippo Brunelleschi, cutaway of the Dome of Florence Cathedral (Santa Maria del Fiore).JPG *Source:* [http://en.wikibooks.org/w/index.php?title=File:Filippo_Brunelleschi,_cutaway_of_the_Dome_of_Florence_Cathedral_\(Santa_Maria_del_Fiore\).JPG](http://en.wikibooks.org/w/index.php?title=File:Filippo_Brunelleschi,_cutaway_of_the_Dome_of_Florence_Cathedral_(Santa_Maria_del_Fiore).JPG) *License:* Public Domain *Contributors:* Andreagrossmann, Balbo, Bouncey21, Edward, G.dallorto, Mac9, Sailko, TomAlt

File:1873 Pierre Auguste Cot - Spring.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:1873_Pierre_Auguste_Cot_-_Spring.jpg *License:* Public Domain *Contributors:* AnonMoos, BlackIceNRW, Cathy Richards, EDUCA33E, Gustav VH, Heflux, Hsarrasin, Infrogmation, Irish Pearl, Jean-Frédéric, Juanpdp, Martin H., Mattes, Mogelzahn, Pierpao, TwoWings, Wst, Zeugma fr, 4 anonymous edits

File:Where Have You Bean.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Where_Have_You_Bean.jpg *License:* Creative Commons Attribution 2.0 *Contributors:* Courcelles, Dirk Hünniger, FlickreviewR, Kmrt, Martin Kraus, Nevit, Shakko

File:84 - Father son.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:84_-_Father_son.jpg *License:* Creative Commons Attribution 2.0 *Contributors:* FlickreviewR, Martin Kraus, Wknight94

File:Jellyfish in pale orange with blue background.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Jellyfish_in_pale_orange_with_blue_background.jpg *License:* Creative Commons Attribution 2.0 *Contributors:* Cwbm (commons), FlickreviewR, Leoboudv, Linnea, Liné1, Martin Kraus

File:Surface normal.png *Source:* http://en.wikibooks.org/w/index.php?title=File:Surface_normal.png *License:* Public Domain *Contributors:* Original uploader was Oleg Alexandrov at en.wikipedia

File:The phase 8 day of the moon .jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:The_phase_8_day_of_the_moon_.jpg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* 阿爾特斯

File:Diffuse_Reflection_Vectors.svg *Source:* http://en.wikibooks.org/w/index.php?title=File:Diffuse_Reflection_Vectors.svg *License:* Creative Commons Zero *Contributors:* User:Martin Kraus

File:Oloid Surface3.png *Source:* http://en.wikibooks.org/w/index.php?title=File:Oloid_Surface3.png *License:* Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contributors:* Xario

File:Gouraudshading00.png *Source:* <http://en.wikibooks.org/w/index.php?title=File:Gouraudshading00.png> *License:* Public Domain *Contributors:* Maarten Everts

File:Phongshading00.png *Source:* <http://en.wikibooks.org/w/index.php?title=File:Phongshading00.png> *License:* Public Domain *Contributors:* Maarten Everts

File:CayleyCubic.png *Source:* <http://en.wikibooks.org/w/index.php?title=File:CayleyCubic.png> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Salix alba

File:Subway lights in tunnel.JPG *Source:* http://en.wikibooks.org/w/index.php?title=File:Subway_lights_in_tunnel.JPG *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Gobagoo

File:The Blue Marble 4463x4163.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:The_Blue_Marble_4463x4163.jpg *License:* Public Domain *Contributors:* NASA. Photo taken by either Harrison Schmitt or Ron Evans (of the Apollo 17 crew).

File:WireSphereLowTass.MaxDZ8.jpg *Source:* <http://en.wikibooks.org/w/index.php?title=File:WireSphereLowTass.MaxDZ8.jpg> *License:* Public Domain *Contributors:* MaxDZ

File:Earthmap720x360 grid.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Earthmap720x360_grid.jpg *License:* Copyrighted free use *Contributors:* based on map by jimht at shaw dot ca

File:NASA-Apollo8-Dec24-Earthrise.jpg *Source:* <http://en.wikibooks.org/w/index.php?title=File:NASA-Apollo8-Dec24-Earthrise.jpg> *License:* Public Domain *Contributors:* Apollo 8 crewmember Bill Anders

File:Iss007e10807 darker.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Iss007e10807_darker.jpg *License:* Public Domain *Contributors:* NASA

File:Land shallow topo alpha 2048.png *Source:* http://en.wikibooks.org/w/index.php?title=File:Land_shallow_topo_alpha_2048.png *License:* Public Domain *Contributors:* Land_shallow_topo_2048.jpg: NASA derivative work: Martin (talk)

File:Globe.svg *Source:* <http://en.wikibooks.org/w/index.php?title=File:Globe.svg> *License:* Public Domain *Contributors:* Abdull, AnonMoos, Cathy Richards, Cbrown1023, Denniss, Finnrind, Fred the Oyster, Geo Swan, Jahoe, Jujoce, Rursus, Shakko, Tony Wills, Valanne, Vonvon, Wereon, Wknight94, Wylove, 13 anonymous edits

File:Skin.png *Source:* <http://en.wikibooks.org/w/index.php?title=File:Skin.png> *License:* Public Domain *Contributors:* US-Gov

File:Earth lights lrg.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Earth_lights_lrg.jpg *License:* Public Domain *Contributors:* Original uploader was DE.MOLAI at it.wikipedia. Later version(s) were uploaded by Belfador at it.wikipedia.

File:Land shallow topo 2048.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Land_shallow_topo_2048.jpg *License:* Public Domain *Contributors:* Carbenium, Haham hanuka, 1 anonymous edits

File:Le Caravage - L'incrédulité de Saint Thomas.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Le_Caravage_-_L'incrédulité_de_Saint_Thomas.jpg *License:* Public Domain *Contributors:* Michelangelo Merisi da Caravaggio

File:IntP Brick NormalMap.png *Source:* http://en.wikibooks.org/w/index.php?title=File:IntP_Brick_NormalMap.png *License:* Copyrighted free use *Contributors:* DwX

File:Image Tangent-plane.svg *Source:* http://en.wikibooks.org/w/index.php?title=File:Image_Tangent-plane.svg *License:* Public Domain *Contributors:* Original uploader was Alexwright at en.wikipedia. Later version(s) were uploaded by BenFrantzDale at en.wikipedia.

File:Scar Top road, cp.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Scar_Top_road,_cp.jpg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Alethe

File:Parallax Vectors.svg *Source:* http://en.wikibooks.org/w/index.php?title=File:Parallax_Vectors.svg *License:* Creative Commons Zero *Contributors:* User:Martin Kraus

File:Gobo projected illustration.png *Source:* http://en.wikibooks.org/w/index.php?title=File:Gobo_projected_illustration.png *License:* Copyrighted free use *Contributors:* ArtMechanic, FSII, Pieter Kuiper, WikipediaMaster, 1 anonymous edits

File:NationalTreasureFilming.jpg *Source:* <http://en.wikibooks.org/w/index.php?title=File:NationalTreasureFilming.jpg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Sean Devine at <http://picassaweb.google.com/seanmdevine>

File:Rembrandt - Der reiche Narr.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Rembrandt_-_Der_reiche_Narr.jpg *License:* Public Domain *Contributors:* Auntof6, Mattes, Mjrmgt, Vincent Steenberg, Wst

File:OHP-sch.JPG *Source:* <http://en.wikibooks.org/w/index.php?title=File:OHP-sch.JPG> *License:* GNU Free Documentation License *Contributors:* mailer_diablo

File:Smiley green alien flustered.svg *Source:* http://en.wikibooks.org/w/index.php?title=File:Smiley_green_alien_flustered.svg *License:* Public Domain *Contributors:* LadyofHats

File:ParkGrillIceSkating.jpg *Source:* <http://en.wikibooks.org/w/index.php?title=File:ParkGrillIceSkating.jpg> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Elaine Sosa Labalme

File:Cube mapped reflection example.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Cube_mapped_reflection_example.jpg *License:* GNU Free Documentation License *Contributors:* User TopherTG on en.wikipedia

File:Quartz ball.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Quartz_ball.jpg *License:* unknown *Contributors:* E.Zimbres and Tom Epaminondas Mineral Collectors

File:A leaning child's view through a skyscraper's window and glass floor.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:A_leaning_child's_view_through_a_skyscraper's_window_and_glass_floor.jpg *License:* Creative Commons Attribution 2.0 *Contributors:* D'Arcy Norman from Calgary, Canada

File:0033 Louvre Venus de Milo.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:0033_Louvre_Venus_de_Milo.jpg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Ricardo André Frantz (User:Tetraktyz)

File:Brushed aluminium.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Brushed_aluminium.jpg *License:* Creative Commons Attribution-Sharealike 2.5 *Contributors:* Martin Kraus

File:Blinn_Vectors.svg *Source:* http://en.wikibooks.org/w/index.php?title=File:Blinn_Vectors.svg *License:* Creative Commons Zero *Contributors:* User:Martin Kraus

File:Pedestrians in Lisbon with light silhouettes due to backlight.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Pedestrians_in_Lisbon_with_light_silhouettes_due_to_backlight.jpg *License:* Creative Commons Attribution-Sharealike 2.0 *Contributors:* FlickreviewR, Martin Kraus, Tonton Bernardo

File:Mercateum.jpg *Source:* <http://en.wikibooks.org/w/index.php?title=File:Mercateum.jpg> *License:* unknown *Contributors:* Neitram

File:Sunlit leaves.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Sunlit_leaves.jpg *License:* Creative Commons Attribution 2.0 *Contributors:* Cirt, FlickreviewR, Martin Kraus, Pierpao

File:Translucency Vectors.svg *Source:* http://en.wikibooks.org/w/index.php?title=File:Translucency_Vectors.svg *License:* Creative Commons Zero *Contributors:* User:Martin Kraus

File:WLA vanda Head and Partial Torso of a Horse jade Han.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:WLA_vanda_Head_and_Partial_Torso_of_a_Horse_jade_Han.jpg *License:* Creative Commons Attribution-Sharealike 2.5 *Contributors:* Wikipedia Loves Art participant "va_va_val"

File:Wax idols at Bandra Fair.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Wax_idols_at_Bandra_Fair.jpg *License:* Public Domain *Contributors:* Kensplanet

File:Chessmen in backlight.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Chessmen_in_backlight.jpg *License:* Creative Commons Attribution 2.0 *Contributors:* Cbigorgne, Cirt, FlickreviewR, Martin Kraus, Pierpao, Shakko, Tony Wills

File:Backlit sphere with shadow.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Backlit_sphere_with_shadow.jpg *License:* Creative Commons Attribution 2.0 *Contributors:* FlickreviewR, Martin Kraus, Tony Wills

File:Skugga.jpg *Source:* <http://en.wikibooks.org/w/index.php?title=File:Skugga.jpg> *License:* Public Domain *Contributors:* Lars Hellvig

File:Michelangelo Caravaggio 072.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Michelangelo_Caravaggio_072.jpg *License:* Public Domain *Contributors:* User:Afernand74

File:Shadow of Sphere.svg *Source:* http://en.wikibooks.org/w/index.php?title=File:Shadow_of_Sphere.svg *License:* Creative Commons Zero *Contributors:* User:Martin Kraus

File:Cow icon 05.svg *Source:* http://en.wikibooks.org/w/index.php?title=File:Cow_icon_05.svg *License:* Public Domain *Contributors:* LadyofHats

File:Goat_icon_05.svg *Source:* http://en.wikibooks.org/w/index.php?title=File:Goat_icon_05.svg *License:* Public Domain *Contributors:* LadyofHats

File:Bull_icon_05.svg *Source:* http://en.wikibooks.org/w/index.php?title=File:Bull_icon_05.svg *License:* Public Domain *Contributors:* LadyofHats

File:Bull cartoon 04.svg *Source:* http://en.wikibooks.org/w/index.php?title=File:Bull_cartoon_04.svg *License:* Public Domain *Contributors:* LadyofHats

File:Donkey cartoon 04.svg *Source:* http://en.wikibooks.org/w/index.php?title=File:Donkey_cartoon_04.svg *License:* Public Domain *Contributors:* LadyofHats

File:Lost Stratosphere title.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Lost_Stratosphere_title.jpg *License:* Public Domain *Contributors:* SusanLesch

File:Fastfood.jpg *Source:* <http://en.wikibooks.org/w/index.php?title=File:Fastfood.jpg> *License:* Creative Commons Attribution-Sharealike 2.5 *Contributors:* Apathetic duck, Gödeke, Mjrmhg, Xnatedawgx, 4 anonymous edits

File:Deformation de la ligne de l'abdomen et des reins par le corset.png *Source:* http://en.wikibooks.org/w/index.php?title=File:Deformation_de_la_ligne_de_labdomen_et_des_reins_par_le_corset.png *License:* Public Domain *Contributors:* Georges Hébert

File:Smiley green alien huh.svg *Source:* http://en.wikibooks.org/w/index.php?title=File:Smiley_green_alien_huh.svg *License:* Public Domain *Contributors:* LadyofHats

File:Projected Shadow.png *Source:* http://en.wikibooks.org/w/index.php?title=File:Projected_Shadow.png *License:* Creative Commons Zero *Contributors:* User:Martin Kraus

File:Diego Velázquez, Venus at Her Mirror (The Rokeby Venus).jpg *Source:* [http://en.wikibooks.org/w/index.php?title=File:Diego_Velázquez,_Venus_at_Her_Mirror_\(The_Rokeby_Venus\).jpg](http://en.wikibooks.org/w/index.php?title=File:Diego_Velázquez,_Venus_at_Her_Mirror_(The_Rokeby_Venus).jpg) *License:* Public Domain *Contributors:* AxelBoldt, Balbo, Butko, Dcoetzee, FischX, Flominator, Goldfritha, Kilom691, Leppus, Luestling, Lycaon, Mattes, Neddyseagoon, Othertree, Raymond, Shakko, Warburg, Xenophon, 10 anonymous edits

File:Water Lily, Sheffield Park - geograph.org.uk - 1191227.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Water_Lily,_Sheffield_Park_-_geograph.org.uk_-_1191227.jpg *License:* Creative Commons Attribution-Share Alike 2.0 Generic *Contributors:* -

File:Plane.svg *Source:* <http://en.wikibooks.org/w/index.php?title=File:Plane.svg> *License:* Creative Commons Attribution 3.0 *Contributors:* Original uploader was Juansempera at en.wikipedia.

File:Unit circle2.svg *Source:* http://en.wikibooks.org/w/index.php?title=File:Unit_circle2.svg *License:* GNU Free Documentation License *Contributors:* User:Pyramide

File:No gimbal lock.png *Source:* http://en.wikibooks.org/w/index.php?title=File>No_gimbal_lock.png *License:* GNU Free Documentation License *Contributors:* MathsPoetry

File:Gimbal lock.png *Source:* http://en.wikibooks.org/w/index.php?title=File:Gimbal_lock.png *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* MathsPoetry

File:Articulating robot.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Articulating_robot.jpg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Abhishek santosh sharma

File:Axis.png *Source:* <http://en.wikibooks.org/w/index.php?title=File:Axis.png> *License:* GNU General Public License *Contributors:* User:Wessmann.clp

File:CAVE Crayoland.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:CAVE_Crayoland.jpg *License:* Public Domain *Contributors:* User:Davepage

File:Picture_plane_projection.png *Source:* http://en.wikibooks.org/w/index.php?title=File:Picture_plane_projection.png *License:* GNU Free Documentation License *Contributors:* Original : Pat Kelso (talk) Additions:Wapcaplet (talk). Original uploader was Wapcaplet at en.wikipedia

File:Bezier 2 big.png *Source:* http://en.wikibooks.org/w/index.php?title=File:Bezier_2_big.png *License:* Public Domain *Contributors:* Phil Tregoning

File:Bezier 1 big.gif *Source:* http://en.wikibooks.org/w/index.php?title=File:Bezier_1_big.gif *License:* Public Domain *Contributors:* Phil Tregoning

File:Bezier quadratic anim.gif *Source:* http://en.wikibooks.org/w/index.php?title=File:Bezier_quadratic_anim.gif *License:* Public Domain *Contributors:* Philip Tregoning

File:Circle and quadratic bezier.svg *Source:* http://en.wikibooks.org/w/index.php?title=File:Circle_and_quadratic_bezier.svg *License:* Public Domain *Contributors:* Damian Yerrick

File:Finite difference spline example.png *Source:* http://en.wikibooks.org/w/index.php?title=File:Finite_difference_spline_example.png *License:* Creative Commons Zero *Contributors:* Tomruen

File:Hermite spline 2-segments.svg *Source:* http://en.wikibooks.org/w/index.php?title=File:Hermite_spline_2-segments.svg *License:* Public Domain *Contributors:* Niabot

File:Catmull-rom-tangent.svg *Source:* <http://en.wikibooks.org/w/index.php?title=File:Catmull-rom-tangent.svg> *License:* Public Domain *Contributors:* Niabot

File:Ford assembly line - 1913.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Ford_assembly_line_-_1913.jpg *License:* Public Domain *Contributors:* Kozuch, Mdd, Vini 175, 1 anonymous edits

File:Airacobra P39 Assembly LOC 02902u.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Airacobra_P39_Assembly_LOC_02902u.jpg *License:* Public Domain *Contributors:* USAAF

File:Camera_analogy.png *Source:* http://en.wikibooks.org/w/index.php?title=File:Camera_analogy.png *License:* GNU Lesser General Public License *Contributors:* Camera_icon.svg: Everaldo Coelho and YellowIcon derivative work: Martin Kraus

File:Perspective view frustum.png *Source:* http://en.wikibooks.org/w/index.php?title=File:Perspective_view_frustum.png *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Martin Kraus

File:358durer.jpg *Source:* <http://en.wikibooks.org/w/index.php?title=File:358durer.jpg> *License:* Public Domain *Contributors:* Anamorphosis, Aristeas, Xenophon, 4 anonymous edits

File:Field of view angle in view frustum.png *Source:* http://en.wikibooks.org/w/index.php?title=File:Field_of_view_angle_in_view_frustum.png *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Martin Kraus

File:Oblique perspective view frustum.png *Source:* http://en.wikibooks.org/w/index.php?title=File:Oblique_perspective_view_frustum.png *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Martin Kraus

File:Orthographic view frustum.png *Source:* http://en.wikibooks.org/w/index.php?title=File:Orthographic_view_frustum.png *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Martin Kraus

File:Viewport transformation.png *Source:* http://en.wikibooks.org/w/index.php?title=File:Viewport_transformation.png *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Martin Kraus

File:Pixels covered by a triangle.png *Source:* http://en.wikibooks.org/w/index.php?title=File:Pixels_covered_by_a_triangle.png *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Martin Kraus

File:Linear interpolation in triangle.png *Source:* http://en.wikibooks.org/w/index.php?title=File:Linear_interpolation_in_triangle.png *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Martin Kraus

File:Perspective correct texture mapping.jpg *Source:* http://en.wikibooks.org/w/index.php?title=File:Perspective_correct_texture_mapping.jpg *License:* Public Domain *Contributors:* Rainwarrior

License

Creative Commons Attribution-Share Alike 3.0 Unported
[//creativecommons.org/licenses/by-sa/3.0/](http://creativecommons.org/licenses/by-sa/3.0/)