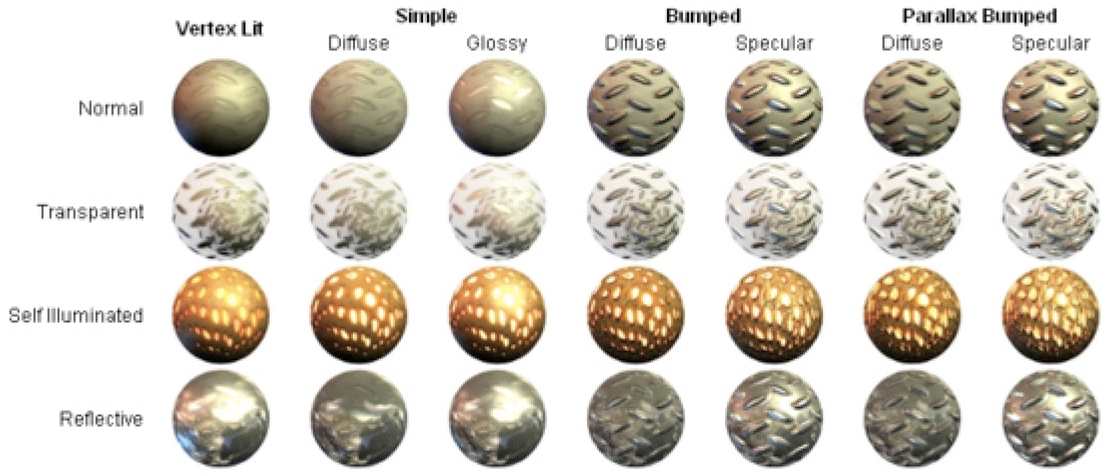
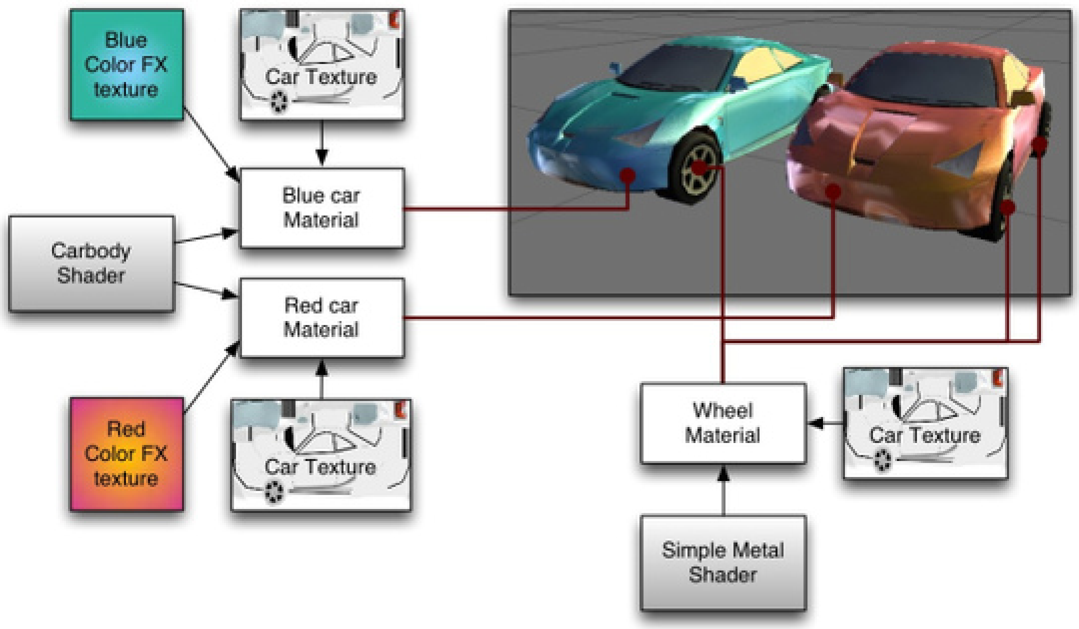
Graphics shaders

Mike Hergaarden
January 2011, VU Amsterdam





[Source: http://unity3d.com/support/documentation/Manual/Materials.html]

Abstract

Shaders are the key to finalizing any image rendering, be it computer games, images or movies. Shaders have greatly improved the output of computer generated media; from photo-realistic computer generated humans to more vivid cartoon movies. Furthermore shaders paved the way to general-purpose computation on GPU (GPGPU). This paper introduces shaders by discussing the theory and practice.

Introduction

A shader is a piece of code that is executed on the Graphics Processing Unit (GPU), usually found on a graphics card, to manipulate an image before it is drawn to the screen. Shaders allow for various kinds of rendering effect, ranging from adding an X-Ray view to adding cartoony outlines to rendering output.

The history of shaders starts at LucasFilm in the early 1980's. LucasFilm hired graphics programmers to computerize the special effects industry [1]. This proved a success for the film/ rendering industry, especially at Pixars Toy Story movie launch in 1995. RenderMan introduced the notion of Shaders;

> *"The Renderman Shading Language allows material definitions of surfaces to be described in not only a simple manner, but also highly complex and custom manner using a C like language. Using this method as opposed to a pre-defined set of materials allows for complex procedural textures, new shading models and programmable lighting. Another thing that sets the renderers based on the RISpec apart from many other renderers, is the ability to output arbitrary variables as an image—surface normals, separate lighting passes and pretty much anything else can be output from the renderer in one pass."* [1]

The term shader was first only used to refer to "pixel shaders", but soon enough new uses of shaders such as vertex and geometry shaders were introduced, making the term shaders more general. Shaders have forced some important movements in the video(card) hardware which will be further examined in the following sections.

From the fixed-function pipeline to the programmable pipeline

Before exploring the latest shader types and possibilities it is important to provide some more background on shaders. To this end we'll look at the graphics pipeline process to put shaders into perspective.

In the 1980s developing computer graphics was a pain; every hardware needed it's own custom software. To this end OpenGL and DirectX were introduced in 1992 respectively 1995. OpenGL and DirectX provide an easy to use graphics API to provide (hardware) abstraction. In 1995 the first consumer videocard was introduced. The graphic APIs in combination with the new hardware made real time (3D) graphics and games finally possible. There was no support for shaders yet. At this time the graphics APIs and video hardware only offered the fixed-function pipeline (FFP) as the one and only graphics processing pipeline. The FFP allowed for various customization of the rendering process. However the possibilities are predefined and therefore limited. The FFP did not allow for custom algorithms. Shader a-like effects are possible using the FFP, but only within the predefined constraints. Examples are setting fog color, lightning etc. For specific image adjustments, such as we want to do using shaders, custom access to the rendering process is required.

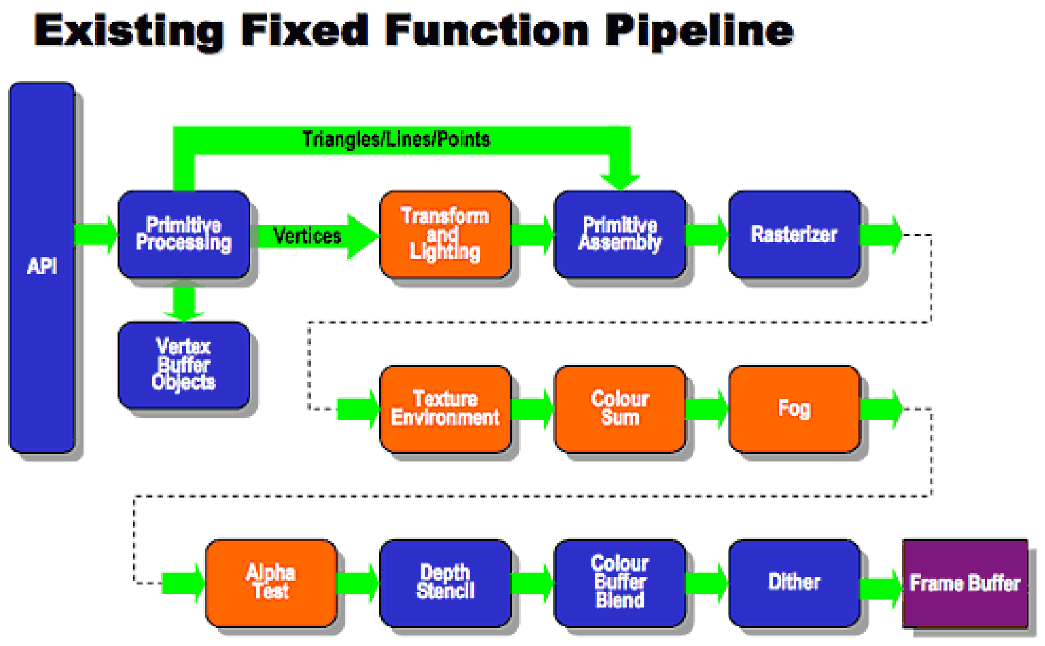


Figure 1: FFP in OpenGL ES 2.0 (http://www.khronos.org/opengles/2_X/)

In 2001 the GeForce3/Radeon 8500 introduced the first limited programmability in the graphics pipeline and the Radeon 9700/GeForce FX brought full programmability in 2002 [2]. This is called the programmable pipeline (PP), see figure 2. The orange boxes in figure 1 and 2 clearly show the difference between the FFP and the PP. The previously fixed-functions are replaced with general phases in which you can modify the data however you want; The PP allows you to do whatever you can program. You're in total control of every vertex and pixel on the screen and by now also geometry. However, with more power comes more responsibility. The PP required a lot more work as well. Shader languages, which we'll discuss in a later section, help to address this problem.

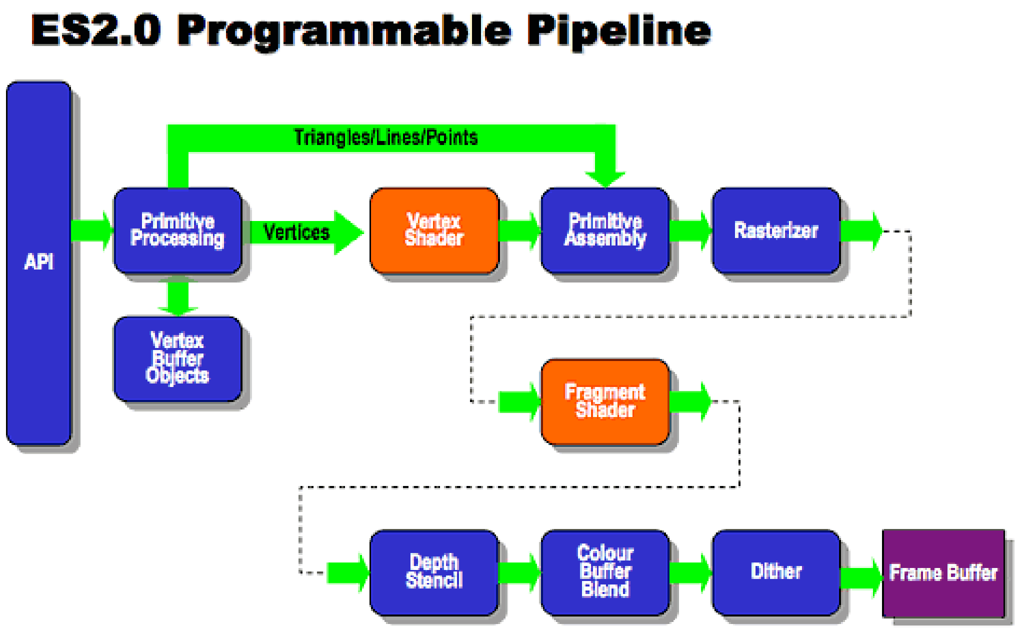


Figure 2: PP in OpenGL ES 2.0 (http://www.khronos.org/opengles/2_X/)

The evolution of the GPU: GPGPU

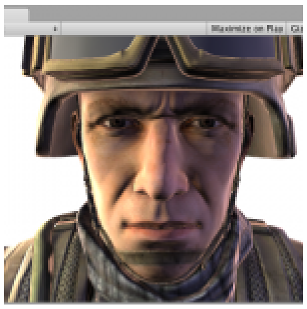
The videocard has evolved to it's current state thanks to 3D graphics. By making use of the video card 3D and 2D graphics are hardware accelerated.

The GPU has reached a point where it so powerful that it's a pity to use it's power just for graphics; Nowadays the GPU is the most powerful computational hardware in a computer.

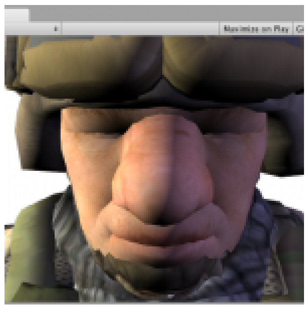
Initiatives to use the GPU for other computations have risen quickly. This GPU utilization is called the General-Purpose computation on GPU (GPGPU)[15]. The key features of GPGPU are the high flexibility, programmability, parallel nature, high-performance, high data throughput. High level programming languages such as C provide interfaces to the GPU allowing any application to make use of the GPU. The performance gained using the GPU is of a bigger magnitude than CPU optimizations. An interesting GPGPU application is the Folding@Home project [16]. This project allows anyone to donate idle time of their computer to the project which studies medical subjects. Playstation 3 owners are able to easily donate CPU and GPU computational efforts as the program is built in the Playstations main menu and supported by Sony.

Types of shaders

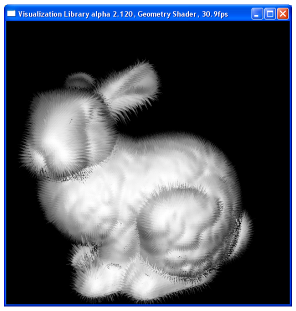
Originally shaders only performed pixel operations; operations are done on every pixel. this is what we now call pixel shaders. Since the introduction of other shaders the term shader is now a more general term to describe any of the following three shaders. Examples of shaders are shown in the shader techniques section.



Pixel shaders(DirectX) / Fragment shaders(OpenGL)
Calculate the color of individual pixels. The image on the left has various applied effects such as rim lightning and normal mapping [6]. Note that DirectX calls this pixel shaders and OpenGL calls this fragment shaders.



Vertex shaders
Modifies a vertices position, color and texture coordinates. To the left an example is shown where the vertices are moved along their normals [6]. Note that vertices can be (re)moved but new vertices cannot be added, that would fall under the geometry shader.



Geometry shaders
The geometry shader is the newest of the three shader types. It can modify vertices. I.e. procedurally generated geometry. An example is shown on the left, where the bunny has polygonal fur generated by the geometry shader [7]. Since this shader is only supported for a few years now it’s (possible) utilization is the most interesting. Even more so than the other two shader types it allows for procedural/dynamic content generation. One use that spring to mind immediately is dynamic grass/hair/fur.

Unified shader model

While the previously mentioned shader types are separate entities in OpenGL and DirectX, a “quite recent” (2006) change by nVidia introduces the Unified shader model hereby merging all shader types into one [8][9]. This unified shader model consists of two parts: A unified shading architecture and a unified shader model.

In DirectX the Unified shader model is called “Shader model 4.0”, in OpenGL it’s simply “Unified shader model” . The instructions for all three types of shaders are now merged into one instruction set for all shaders. To this end, the hardware had to undergo some changes as well. The hardware used to have separate units for vertex processing, pixel shading etc. To accommodate for the unified shader model a single shader core and processors were developed. Any of the independent processors can handle any type of shading operation [14]. This hardware change allows for more flexible workload distribution as it’s perfectly fine to dedicate all processors to pixel shading when no vertex shading is required.

Shader languages

Originally shaders were written in assembly, but as with any other programming fields it became clear that a higher level language was required. When the programmable pipeline was introduced one could finally program shaders in assembly. However, shaders were hard to write and read.

Assembly shader code:

```
vs_2_0
dcl_position v0
dcl_texcoord v1
dp4 oPos.x, v0, c0
```

To make shaders easier to understand and debug higher level programming languages were developed. These shader languages compile to assembly shader code. The three major shader languages that have been developed are:

HLSL (DirectX):

- Outputs a DirectX shader program.
- Latest version as of January 2011: Shader model 5 (http://msdn.microsoft.com/directx)
- Syntax similar to Cg

GLSL (OpenGL):

- Outputs a OpenGL shader program.
- Latest version as of January 2011: OpenGL 4.1 (http://www.opengl.org/registry/#apispecs)

Cg (nVidia):

- Outputs a DirectX or OpenGL shader program
- Latest version as of January 2011: Cg 3.0 (http://developer.nvidia.com/page/cg_main.html)
- Syntax similar to HLSL

The Cg and HLSL syntax is very similar since nVidia and Microsoft co-developed these languages. The main difference between Cg and HLSL is that Cg has extra function to support GLSL. None of these language is ultimately the best as it depends on the targeted platform(s) and the specific features/performance you need. Cg seems to be the most appealing language since it supports both the major paltforms. Cg is the shader language used in the Unity game engine, see Appendix 2 and 3 for more details on this subject.

For a simple demonstration of a shader we look at an Cg shader, or actually, we will look at an CgFX "effect". This needs explanation first. Cg shaders are not perfectly transferable between applications as application specific details are always embedded in Cg shaders by requirement. CgFX solves this. A CgFX "effect" is everything required, as a bundle, to render a desired shader/rendering effect so that it can be used independently from the 3D application that finally uses the effect [17]. DirectX has it's HLSL equivalent namely Directx Fx's.

Below is the code for a simple shader that colors a vertice green. As you can figure from the name and the example source code, Cg code is very similar to the C language. The same counts for the other two shader languages. However, the shading languages have their own data types and structure.

```
struct VertIn {
   float4 pos   : POSITION;
   float4 color : COLOR0;
};

VertOut main(VertIn IN, uniform float4x4 modelViewProj) {
   VertOut OUT;
   OUT.pos     = mul(modelViewProj, IN.pos); // calculate output coords
   OUT.color   = IN.color; // copy input color to output
   OUT.color.z = 1.0f; // blue component of color = 1.0f
   return OUT;
}
```

To start with shaders, AMDs RenderMonkey and nVidias FX composer are essential tools; they allow you to quickly and easily test and compile shaders. Which one is the best to use is a matter of personal taste. The tools can be downloaded here:

RenderMonkey: http://developer.nvidia.com/object/fx_composer_home.html
FX Composer: http://developer.amd.com/archive/gpu/rendermonkey/pages/default.aspx

Shader techniques

To visualize the application of shaders this section lists various shading techniques. Ranging from per-object shader to full screen shaders.

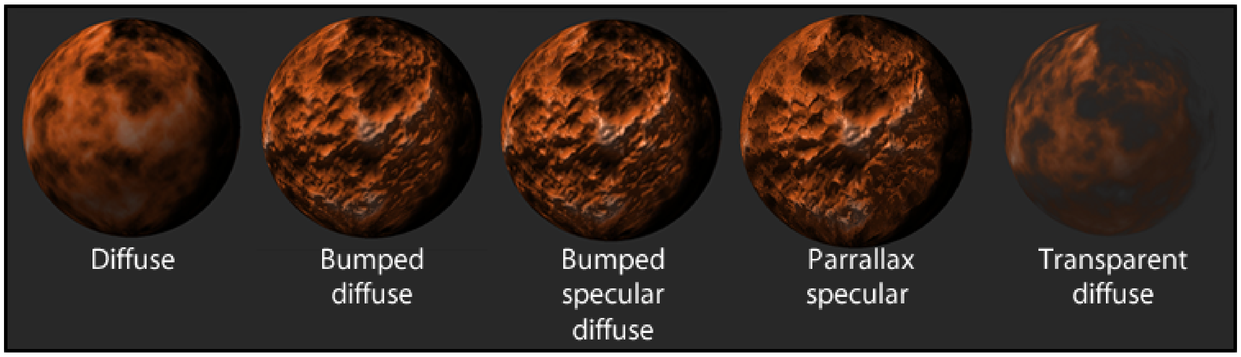


Figure 3: common shaders

The most common shaders are shown in the image above. In nowadays 3D games usually all of these shaders used. The aim of these shaders is to add lightning effects or to add a level of detail to objects without increasing the geometry's level of detail (which is more costly to render). For code examples of the above five shaders see Appendix C.

Diffuse: Calculate color based on lightning.
Bumped diffuse (/normal mapping): Fakes the lightning of bumps without adding any geometry.
Specular: Defines the reflection/"Shininess" of an object.
Parrallax specular: Creating displacement; an object looks different from different angles and distances.
Transparant: Transparency calculations based on color/alpha level.

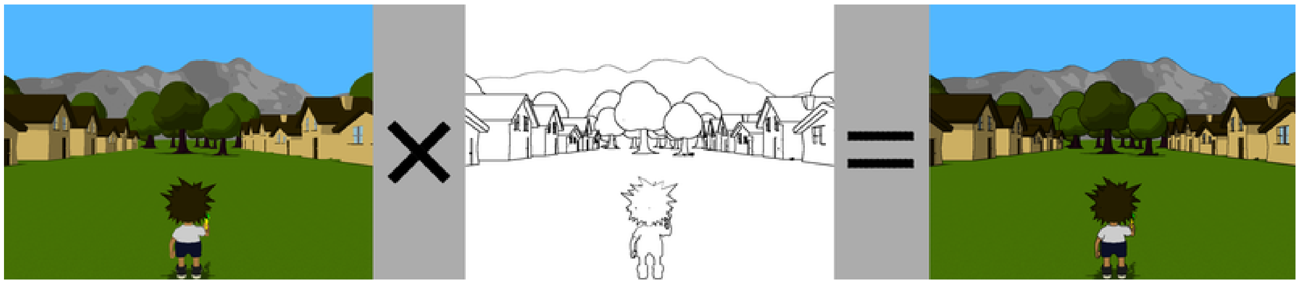


Figure 4: Cel shading (http://en.wikipedia.org/wiki/Cel-shaded_animation)

Next to the shaders that are applied per-object, there are shaders that are applied for the entire screen. The most common full screen shaders are DoF, SSAO, Bloom and toon shading.

Depth of Field (DoF): Only a section of the screen is sharp, creating focus on a target.
Bloom/glow: Adds a lot of glow around edges of a bright object/light source.
Motion blur: By blending several frames the notion of motion is added.
Toon shading/Cel shading: Adds a black outline to edges for a cartoon effect.
Screen space ambient occlusion (SSAO): Ambient occlusion makes reflections more realistic.

Alternatives

To put shaders in context it is useful to define their boundaries and alternatives. To recap, shaders are programs that manipulate rendering output and are run on the GPU. Alternatives to shaders would be other methods of modifying rendering (output). The closests alternatives are the fixed-function pipeline and rendering manipulating software. One example of the last option is PixelBender[18].

As the fixed function pipeline is the predecessor of shaders it is the closest alternative for shaders. However, shaders were introduced to address issues (read limitations) with the fixed function pipeline, this alone indicates that the fixed function pipeline is no true alternative.

Different alternatives would be other programs that manipulate rendering output. However, for these programs it would make no sense not to use the GPU and it's programmable pipeline, which is what would roughly define a shader. Simply put; there's no room for alternatives. Any 'alternative' would provide a more limited version of rendering manipulation.

One example is PixelBender by Adobe [18]. PixelBender allows image processing in a hardware-independent matter. However, PixelBender does allow hardware acceleration if a supported GPU is present. At the moment of writing the GPU support is not great yet. It is lacking accuracy and can create rendering artifacts [20]. A product like PixelBender will never replace shaders, but is still very useful for the flash platform. It is more likely that PixelBender would support making use of real shader function in the future.

Conclusion

The very basics of shaders can be easily summarized: The programmable pipeline allowed shaders to fully change rendering output instead of the limited options the fixed-function pipeline had to offer. There are three types of shaders; Vertex, Pixel and Geometry shaders. These three shaders have been unified in the Unified shader model. To program shaders three main languages are available; HLSL, GLSL and Cg.

Shader technology has matured the last few years, both the software and hardware. However this does *not* mean we'll see less new developments from now on. The latest shader models are not yet utilized broadly because hardware adoption is slow.
Shaders are an essential piece of the rendering process to finish an image/movie or game. In games shaders even allow for new gameplay, think of X-rays or dynamic content via geometry shading. Related fields that undergo a lot of movement are parallel (shader) processing and moving processes from the CPU to the GPU (GPGPU). As hardware keeps improving we will only see more and more uses of shaders appearing, especially on mobile devices where the hardware support is still limited.

References

[1] RenderMan shader history, 2006.
http://wiki.cgsociety.org/index.php/Renderman
Since RenderMan defined the notion of shading, this article explains the history of shaders.

[2] Understanding the graphics pipeline, Suresh Venkatasubramanian, 2005.
http://www.cis.upenn.edu/~suvenkat/700/lectures/2/Lecture2.ppt
A self explanatory presentation about the evolution of the graphics pipeline. The Fixed-function pipeline is explained very well

[3] Introduction to Shaders
http://gamedevelopment.com/blogs/AtulSharma/20090318/937/Introduction_to_Shaders.php
A very good resource for the simplest introduction to shaders.

[4] How things work: How GPUs work. David Luebke, NVIDIA Research, Greg Humphreys, University of Virginia, 2007.
http://www.cs.virginia.edu/~gfx/papers/pdfs/59_HowThingsWork.pdf
Detailed article about the GPU.

[5] A Crash Course in HLSL, Matt Christian
www.insidegamer.org/presentations/CrashCourseInHLSL.ppt
Presentation about HLSL, with a brief general shading introduction.

[6] Unity 3 technology – Surface Shaders, Aras Pranckevičius, 2010.
http://blogs.unity3d.com/2010/07/17/unity-3-technology-surface-shaders/
Technical note about surface shaders implementation in Unity.

[7] Visualization Library; A lightweight C++ OpenGL middleware for 2D/3D graphics, Michele Bosi.
http://www.visualizationlibrary.com/jetcms/node/4
Visualization library, featuring shader visualisations in the image gallery.

[8] Common-Shader Core (DirectX HLSL), Microsoft.
http://msdn.microsoft.com/en-us/library/bb509580(VS.85).aspx

Official DirectX HLSL documentation.

[9] GeForce 8800 GTX: 3D Architecture Overview, ExtremeTech, 2006.
http://www.extremetech.com/article2/0,1697,2053309,00.asp
Highlighting nVidia's new Unified shader model GPU Architecture

[10] Shaders for Game Programmers and Artists - Sebastien St-Laurent, 2004.
Great shader introduction book

[11] Shaders, Bonzai Engine, 2010.
http://jerome.jouvie.free.fr/shaders/Shaders.php
Various, shaders visualisations and accompanying shader tutorial.

[12] Introduction to Shaders, Marco Benvegnù, 2005.
http://www.benve.org/Download/Introduction%20to%20Shaders.pdf
Detailed presentation about shaders.

[13] Exploiting the Shader Model 4.0 Architecture, Suryakant Patidar, Shiben Bhattacharjeey,
Jag Mohan SinghzP. J. Narayanan, 2006.
http://researchweb.iiit.ac.in/~skp/papers/gfxSM4.pdf
Advanced paper researching new features of the 4.0 shader model.

[14] GeForce 8 introduction FAQ, nVidia, 2006.
http://www.nvidia.com/object/geforce_8600_8500_faq.html
This FAQ explains the new unified shader core.

[15] GPGPU.org
http://gpgpu.org
Catalogs the current and historical use of GPUs for general-purpose computation and provides a central resource for GPGPU software developers

[16] Folding@Home, Pande lab Stanford University
http://folding.stanford.edu/
Distributed computing project, also makes use of GPGPU.

[17] Cg Runtime Tutorial with CgFX, nVidia, 2006.
http://developer.download.nvidia.com/cg/Cg_3.0/CgFX_bumpdemo_Tutorial.pdf
Bumpmap implementation tutorial with CgFX

[18] PixelBender, Adobe.
http://www.adobe.com/devnet/pixelbender.html

An alternative shading implementation for the flash platform

[19] PixelBender rendering performance, Adobe, 2010.
http://help.adobe.com/en_US/as3/mobile/WS948100b6829bd5a6-54120f1012593d8f030-8000.html
PixelBender documentation, clarifying the GPU rendering performance and usage.

Appendix A: Shading tools

RenderMonkey
by ATI
http://developer.amd.com/archive/gpu/rendermonkey/pages/default.aspx

Cg Toolkit
by nVidia
http://developer.nvidia.com/object/cg_toolkit.html

FX Composer
by Microsoft & nVidia
http://developer.nvidia.com/object/fx_composer_home.html

Appendix B: Shaders in Unity

This appendix lists some pointers on how to get started with shaders in Unity (http://www.Unity3d.com). Unity has developed it's own shader language which they call "ShaderLab". They made their own wrapper to allow seamless integration in Unitys (i.e. using the built in materials). For the "real" shader operations ShaderLab includes Cg/HLSL code. GLSL is also supported but since this will not compile for Cg/HLSL it is not advised to be used unless you know you're only targetting GLSL platforms.

Hereby an example of one of the simplest Unity shaders. It makes an object entirely white and enables lambert lightning.

```
Shader "Example/Diffuse Simple" {
   SubShader {
     Tags { "RenderType" = "Opaque" }
     CGPROGRAM
     #pragma surface surf Lambert
     struct Input {
         float4 color : COLOR;
     };
     void surf (Input IN, inout SurfaceOutput o) {
         o.Albedo = 1;
     }
     ENDCG
   }
   Fallback "Diffuse"
 }
```

[http://unity3d.com/support/documentation/Components/SL-SurfaceShaderExamples.html]

Video tutorials:
http://unity3d.com/support/resources/unite-presentations/shader-programming-course
http://unity3d.com/support/resources/unite-presentations/shader-tricks-in-game-production

Shader examples:
http://unity3d.com/support/resources/example-projects/shader-replacement

Documentation:
http://unity3d.com/support/documentation/Manual/Shaders.html
http://unity3d.com/support/documentation/ScriptReference/Shader.html
http://unity3d.com/support/documentation/Components/SL-Reference.html

Built-in shader descriptions:
http://unity3d.com/support/documentation/Components/Built-in%20Shader%20Guide.html

About Unity rendering:
http://unity3d.com/support/documentation/Components/Rendering-Tech.html

Appendix C: Default Unity shaders

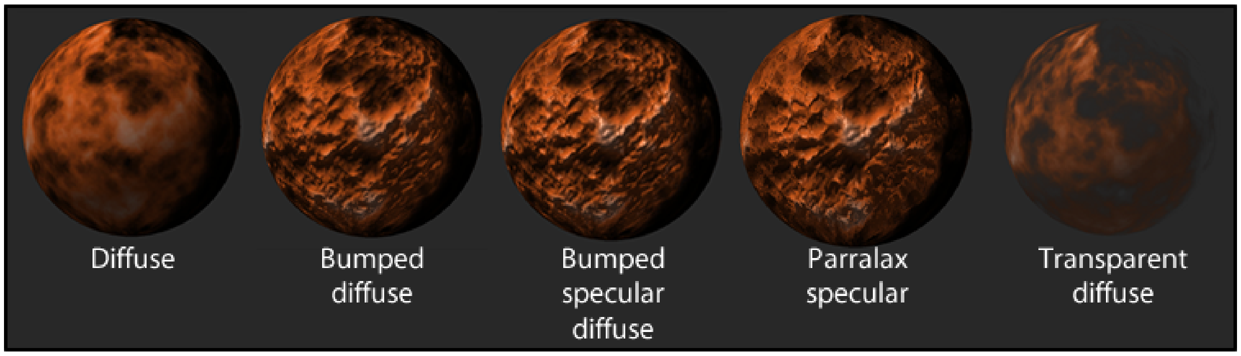


Figure 3: common shaders

This appendix lists the source code of the most used Unity shaders: Diffuse, Bumped diffuse, Bumped specular diffuse, Parrallax specular and the transparent shader. The shaders are listed in this specific order as the first four are extensions of their preprocessors. The changes in relation to the previous shader are highlighted in red. While Unity uses their own custom “ShaderLab” language, most of the code is Cg code which is included in ShaderLab.

The source code of these shaders is provided by Unity and can be found on:
http://unity3d.com/support/resources/assets/built-in-shaders

Diffuse:

```
Shader "Diffuse" {
Properties {
	_Color ("Main Color", Color) = (1,1,1,1)
	_MainTex ("Base (RGB)", 2D) = "white" {}
}
SubShader {
	Tags { "RenderType"="Opaque" }
	LOD 200

CGPROGRAM
#pragma surface surf Lambert

sampler2D _MainTex;
float4 _Color;

struct Input {
	float2 uv_MainTex;
};

void surf (Input IN, inout SurfaceOutput o) {
	half4 c = tex2D(_MainTex, IN.uv_MainTex) * _Color;
	o.Albedo = c.rgb;
	o.Alpha = c.a;
}
ENDCG
}

Fallback "VertexLit"
```

Bumped diffuse

```
Shader "Bumped Diffuse" {
Properties {
            _Color ("Main Color", Color) = (1,1,1,1)
            _MainTex ("Base (RGB)", 2D) = "white" {}
            _BumpMap ("Normalmap", 2D) = "bump" {}
}

SubShader {
            Tags { "RenderType"="Opaque" }
            LOD 300

CGPROGRAM
#pragma surface surf Lambert

sampler2D _MainTex;
sampler2D _BumpMap;
float4 _Color;

struct Input {
            float2 uv_MainTex;
            float2 uv_BumpMap;
};

void surf (Input IN, inout SurfaceOutput o) {
            half4 c = tex2D(_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;
            o.Alpha = c.a;
            o.Normal = UnpackNormal(tex2D(_BumpMap, IN.uv_BumpMap));
}
ENDCG
}

FallBack "Diffuse"
}
```

Bumped specular

```
Shader "Bumped Specular" {
Properties {
            _Color ("Main Color", Color) = (1,1,1,1)
            _SpecColor ("Specular Color", Color) = (0.5, 0.5, 0.5, 1)
            _Shininess ("Shininess", Range (0.03, 1)) = 0.078125
            _MainTex ("Base (RGB) Gloss (A)", 2D) = "white" {}
            _BumpMap ("Normalmap", 2D) = "bump" {}
}
SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 400

CGPROGRAM
#pragma surface surf BlinnPhong

sampler2D _MainTex;
sampler2D _BumpMap;
float4 _Color;
float _Shininess;

struct Input {
        float2 uv_MainTex;
        float2 uv_BumpMap;
};

void surf (Input IN, inout SurfaceOutput o) {
        half4 tex = tex2D(_MainTex, IN.uv_MainTex);
        o.Albedo = tex.rgb * _Color.rgb;
        o.Gloss = tex.a;
        o.Alpha = tex.a * _Color.a;
        o.Specular = _Shininess;
        o.Normal = UnpackNormal(tex2D(_BumpMap, IN.uv_BumpMap));
}
ENDCG
}

FallBack "Specular"
}
```

(Bumped) Parrallax Specular

```
Shader "Parallax Specular" {
Properties {
        _Color ("Main Color", Color) = (1,1,1,1)
        _SpecColor ("Specular Color", Color) = (0.5, 0.5, 0.5, 1)
        _Shininess ("Shininess", Range (0.01, 1)) = 0.078125
        _Parallax ("Height", Range (0.005, 0.08)) = 0.02
        _MainTex ("Base (RGB) Gloss (A)", 2D) = "white" {}
        _BumpMap ("Normalmap", 2D) = "bump" {}
        _ParallaxMap ("Heightmap (A)", 2D) = "black" {}
}
SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 600

CGPROGRAM
#pragma surface surf BlinnPhong

sampler2D _MainTex;
sampler2D _BumpMap;
sampler2D _ParallaxMap;
float4 _Color;
float _Shininess;
float _Parallax;

struct Input {
	float2 uv_MainTex;
	float2 uv_BumpMap;
	float3 viewDir;
};

void surf (Input IN, inout SurfaceOutput o) {
	half h = tex2D (_ParallaxMap, IN.uv_BumpMap).w;
	float2 offset = ParallaxOffset (h, _Parallax, IN.viewDir);
	IN.uv_MainTex += offset;
	IN.uv_BumpMap += offset;

	half4 tex = tex2D(_MainTex, IN.uv_MainTex);
	o.Albedo = tex.rgb * _Color.rgb;
	o.Gloss = tex.a;
	o.Alpha = tex.a * _Color.a;
	o.Specular = _Shininess;
	o.Normal = UnpackNormal(tex2D(_BumpMap, IN.uv_BumpMap));
}
ENDCG
}

FallBack "Bumped Specular"
}
```

Last but not least is the transparency shader. It makes no sense to compare it to the Parrallax specular shader as that shader is far more complicated, instead, this shader has been compared against the diffuse shader.

Transparent Diffuse:

```
Shader "Transparent/Diffuse" {
Properties {
	_Color ("Main Color", Color) = (1,1,1,1)
	_MainTex ("Base (RGB) Trans (A)", 2D) = "white" {}
}

SubShader {
	Tags {"Queue"="Transparent" "IgnoreProjector"="True" "RenderType"="Transparent"}
	LOD 200

CGPROGRAM
#pragma surface surf Lambert alpha

sampler2D _MainTex;
float4 _Color;

struct Input {
	float2 uv_MainTex;
};
void surf (Input IN, inout SurfaceOutput o) {
	half4 c = tex2D(_MainTex, IN.uv_MainTex) * _Color;
	o.Albedo = c.rgb;
	o.Alpha = c.a;
}
ENDCG
}

Fallback "Transparent/VertexLit"
}
```