MAGIC
and how to do it yourself

Chapter 1: Introduction

1.1 Shaders, the reason to read your way through this

When talking about shaders, people usually think of a couple of exaggerated special effects like specularity, parallax occlusion mapping, bloom and blur effects and believe that they can do most things better without shaders.
But in reality, shaders are already a needed part of the rendering pipeline. They are basicly small programs, usually one executed for each vertex of the rendered mesh (a vertex shader program) and one for each „visible“ pixel/fragment (the pixel shader program or also fragment shader program) of the rendered mesh. These shader programs are designed to be executed very efficiently in parallel on the graphics card. The main task of the vertex shader is to project the meshs vertex positions from their local coordinate system into the coordinate system of your monitor applying scaling, rotation and translation of the object and the camera and the final projection to 2D. The pixel shader has to determine the drawn pixels color which is usually looked up from a texture using the

vertices texture coordinates which are automatically interpolated between the pixels. To the texture color one usually applies some shading calculated in the vertex or pixelshader, or both.
As there are usually more drawn pixels than vertices, the vertex shader can usually be more complex than the pixel shader.
Before there were shaders, there was the fixed function pipeline. It did exactly the things I described above (often referred to as hardware transformation and lighting (T&L)) and offered an interface to manipulate it a little bit. This fixed function pipeline was part of the graphics hardware for a couple of years and is now mostly replaced completely by shaders. The hardware with a real fixed function pipeline is mostly the one of mobile phones like the iPhone 3G. The iPhone 3GS and later however already have a full shader based rendering as well as several other current mobile phones. NVIDIAs upcoming Tesla solution which will probably be part of many future mobiles is also completely shader based.
All hardware without a fixed function pipeline emulates it using shaders!
This means that the correct shader has to be choosen again and again, which of course means a small performance loss compared to directly assigning the correct selfwritten shaders. Directly assigning the correct shaders also allows to use simplified shading calculations on some objects and some more complex effects on others, which can help you to make your scenes look better and run at a higher speed, than without shaders.
But you always have to keep in mind that more advanced shaders are of course slower than simple ones, which means that in the best case you don´t have few quite complex, general shaders, but instead many different shaders for each slightly different situation and while this probably shows very well how flexible shaders are, it also shows their big disadvantage compared to the fixed function pipeline. Writing many different shaders means a lot of extra work compared to just enable lighting, placing up to eight lights and then render everything, which is how it basicly works with the FFP.
However, when looking on current and future devices, there is no way around shaders and upcoming graphics APIs probably won´t even support the fixed function pipeline anymore, like OpenGL ES 2.0 for example already does.

1.2 Object and Post Processing shaders

I tend to differentiate between object and post processing shaders. Technically it is both the same, but when using and writing them they just feel different. A object shader is a shader directly used to render an object as a part of the scene. A full screen post processing shader in contrast is a shader applied to a screen aligned quad which processes an image of the previously rendered scene. This means that one has to render the scene to a texture and not to the screen.

1.3 Shader languages

In the very beginning of hardware supporting shaders, developers had to write their shaders in assembler, but soon the High Level Shading Language (HLSL) was developed for DirectX and the OpenGL Shading Language (GLSL) for OpenGL. A bit later, NVIDIA introduced C for Graphics (CG), which is basically the same as HLSL, but compiles to DirectX as well as to OpenGL shader code, and works with basically any graphics card, not only NVIDIA ones, as one may expect.
I personally prefer the naming conventions of HLSL/CG, which is also less strict than GLSL if it comes to typecasting (converting from int to float for example, as GLSL code

won´t compile if one assigns 1 to a float, while 1.0 works fine of course, and both usually works with HLSL/CG).
In the end they are all based on the C syntax with some minor differences in naming and setting up registers to pass variables to from the vertex to the pixel shader.
Because of this, I will provide all examples in HLSL/CG and may add a chapter on how to convert your HLSL/CG code to GLSL, which seems to have a bright future on mobile devices

1.4 Coordinate systems and transformations

For writing shaders, you need to understand some basics about different coordinate systems and on how to transform between them. The coordinate system of most vertices pushed into the rendering pipeline will usually be the one you know from within your model editor, which is sometimes rotated or mirrored on export to fit your engines defaults. In this coordinate system, each vertex has a three dimensional position and the center is usually within or close to your mesh. This coordinate system is usually called object space or local space. This mesh is then placed within your level with a position, a rotation and a scale relative to your levels origin. This is often refered to as world space or global space. Now within your level, you usually also place a camera/viewer. When looking through the camera, you will see the objects in your world space from the cameras point of view, the so called view space or camera space. The origin of the view space is the cameras center. As you usually watch your level through the camera on your flat monitor, there is also the projection space, which is a two dimensional image created by projecting your three dimensional camera space to a a plane with the monitors center as origin. The projection also applys the camera attributes like the field of view and the aspect ratio.
As the vertex shader receives the vertices in object space and should output them in projection space, one has to transform the vertex position to world space, to view space and then to projection space. These transformations are usually done by multiplying the vertex positions with matrices provided by the engine. It is also possible to invert such a matrix to transform the other way around with it. The order of a matrices values in OpenGL is usually a bit different than the one used in DirectX, which means that one usually have to multiply the other way around in OpenGL than in DirectX. The order of the multiplication is important as a matrix multiplication isn´t commutative.
This means that A*B is not B*A if A and B are matrices (a vector is a matrix with just one row/column). In some cases A*B is the same as B*Ainv, where Ainv is the inverted A matrix and A*B is always B*Atrans, where Atrans is the transposed A matrix, which means that rows and columns are swapped which is also the difference between OpenGL and DirectX matrices.
When multiplying several matrices and finally transforming a vector with the result, the transformations of each matrix is applied in the order of multiplication to the vector. This matrix is usually provided by the engine as WorldViewProjectionMatrix or something similar and allows to simplify the transformation formula within a vertex shader.
In the end, this is no hard maths or something, but the math behind a matrix multiplication tends to confuse, especially when working with both, DirectX and OpenGL. And the most important thing to keep in mind is to check the order of multiplication if something behaves really strange.
There are of course some more coordinate systems one may needs when using tangent space normalmaps or when doing animations on the GPU.

1.5 Shading

Shading or lighting, is the process of determining the brightness and tone of a drawn pixel which is then multiplied and/or added to a pixels initial color taken from for example a texture.
The shading usually consists of ambient lighting, diffuse lighting and specular lighting. Ambient light is the general brightness caused by light being reflected and scattered through the whole scene kinda randomly. The ambient light is usually just a constant factor. The diffuse light is the light directly illuminating the surface, which is then scattered into all directions. The diffuse lighting is usually calculated based on Lambert´s cosine law, which basicly says that the light intensity observed from a completely diffuse surface is proportional to the cosine of the angle betwen the light direction vector and the surface normal vector. As the cosine of that angle is the same as the dot product of the two vectors, this can be calculated really easily with good performance.
The specular light is the light which is directly reflected into the viewers eyes by the surface. This would be the only lighting technique for a perfect mirror and is the most complex part of the usual shading and usually based on a formula developed by Bui Tuong Phong which was later improved by Jim Blinn and is now known as the Blinn-Phong reflection model. While it is not based on physics in any ways, it produces some very decent results. It uses the vector pointing to a point between the light and the viewer, which would also be the ideal normal for full reflection, usually called half vector, which is then compared with the surface normal using the dot product.

1.6 Fog, Multitexturing and other extensions

Most hardware offers more than just the very basic transformation and lighting. Fog is for example a very useful effect, which can usually be activated using the fixed function pipeline, based on a start and end value and a color. The color will completely cover all parts further away than the end value and won´t effect the parts closer than the start value. In between it is blended over the color without fog and one can often choose if it is meant to be linear or quadratic. On most hardware and up to shadermodel 2 when working with DirectX, it is possible to calculate a fog factor within the vertex shader and pass it to some special fog register. Everything else will than be handled automatically. A basic formula for a linear fog factor is fog = (camdist-fogstart)/(fogend-fogstart).
Another great thing can be multitexturing. This means that one pixel can be influenced by more than one texture in one draw call and offers a fast possibility to realize effects like for example detailmapping and blending different textures together based on the textures alpha channel.
It also makes the basis of bump and environment mapping which is supported on some hardware through the FFP as well.
While multitexturing is not a must for shader hardware, it is usually possible to use at least four textures per pixel and often a lot more.

1.7 Renderstates

Before rendering, it is possible to setup some behavior not influenceable from within the shader, the so called renderstates. These renderstates define, which colors of a pixel are meant to be drawn, if a pixel is meant to draw into the depth buffer and if it is meant to be drawn if covered by another pixel or if there is another pixel at exactly the same position and so on. It is also possible to activate and deactivate backface culling (clipping of faces not facing the camera) and to specify how to blend the rendered pixel over the pixels drawn before at the same position. Good knowledge of the use of renderstates can be very helpful if it comes to the rendering of translucent objects.

1.8 Passes

To render an object in several passes means that it is rendered several times. The geometry and texture data is usually exactly the same, but the shader and render states are usually different. This allows to create quite complex effects on hardware that is too restricted to do the effect in just one shader. It is also an easy possibility for fur and outline effects.

1.9 Uniforms, attributes, varyings and registers

In OpenGL and DirectX, there are uniforms which are variables passed from the engine to the shader, which are uniform for all vertices and pixels they were set for before rendering, usually per object. Important uniforms which are often set up per default by the engine are the transformation matrices needed for transforming the vertex positions and the light and material properties as well as often the camera position and information about the fog. Other important uniforms are the textures.
Attributes are variables which are set per vertex. Important attributes are the vertex position, the vertex normal and the vertices texture coordinates. There can be some more, but these are the ones defined by the default vertex format in nearly every engine. In OpenGL the attributes can have any name, but there are some predefined ones in most OpenGL versions.
Varyings are variables passed from the vertex shader to the pixel shader. They are set up within the shader and their values are linearly interpolated for each pixel between the vertices.
Attributes and varyings are mainly concepts of OpenGL and are replaced in DirectX with the concept of registers. In DirectX it is possible to assign variables to registers, with predefined names. The register names for the attributes mentioned above are POSITION, NORMAL0 and TEXCOORD0. The variable assigned to such a register will access the value from within the register and write to it. The registers used for varyings are usually COLORn and TEXCOORDn, where n is a number between 0 and a limit by the shadermodel.
The vertex shader always has to output a position. The register it has to be passed to is called POSITION. In OpenGL, it has to be assigned to a predefined variable called gl_Position.
The pixel shader has to output one or more colors. The registers for this are COLORn and the OpenGL variables are gl_Colorn.