The democratization of real-time graphics

The basics and ideas of pixel shaders from a programmer's point of view

Joost van Dongen

24th April 2006

Website: www.oogst3d.net
E-mail: tsgoo@hotmail.com

Abstract

Vertex and pixel shaders have broadly expanded the possibilities of real-time graphics. This article explains how they work and shows some examples of what can be achieved using these programmable hardware shaders.

This article was written for the course "Advanced Graphics", supervised by dr. R.W. van Oostrum in 2006, for the master Game & Media Technology at the Utrecht University.

Contents

1 Introduction

In the last decade, real-time computer graphics has made a huge leap forward. From the first 3DFX hardware to today's newest cards from Nvidia and ATi has not only brought forward immense brute triangle and pixel processing power, but has also yielded something much more exciting: programmability. The first graphics hardware simply rendered triangles to the screen, but today's GPU's allow almost full control over how this is done. The programmer can send special shader programs to run on the graphics card and calculate the positions of vertices and colours of pixels. Although some strong constraints are still attached to these shader-programs, an inventive programmer can achieve many effects that were formerly unthinkable in real-time. These innovations not only make it possible to create more complex and realistic shading, but also allow to create original and artistic effects that were unthinkable in the past. Therefore, programmable hardware shaders can truly be called the democratization of real-time graphics.

This article starts with a brief explanation of how graphics cards worked in the past and how they evolved to their current state. After this comes the core of the subject: how do these programmable shaders work? This is followed by one detailed example of a vertex and pixel shader and some shorter examples that illustrate the wide range of possibilities. After all this pleasant information comes the bad part: shader programmers still face some heavy limitations due to the different functionality on various graphics cards and due to the nature of shaders.

This article is an introduction into the world of programmable shaders and does not intend to explain exactly how to write a shader. In the reference-section some very practical tutorials are listed where the reader can bring the knowledge learned here into practice.

2 The evolution of real-time graphics

To understand the benefits of programmable shaders, it is necessary to first understand how real-time graphics were without them. Therefore in this paragraph it is shown how graphics hardware came from where it started to where it is now.

Traditionally, real-time graphics where rendered by the CPU. This made them quite the same as non real-time graphics, only differing in the speed with which the frames were rendered. Because the most important part of calculating real-time images was the drawing of textured polygons to the screen, specialised hardware was developed to do only this. The first of such specialised rendering hardware came from companies like Silicon Graphics and Evans & Sutherland [1], and was very expensive. With cards such as the famous 3DFX Voodoo, this kind of hardware reached the games and the consumer marked in the mid-1990's.

The reason such cards can work much faster than a normal CPU without becoming very expensive, is that they are so specialised. Originally, they could only light vertices and interpolate these values over the triangles, so only simple lighting could be performed. Furthermore, the user had little control over the lighting-algorithms and could only create the few looks that were hardwired in the GPU. These cards were incredibly limited and only did what the hardware manufacturer had built in, but because they did this so fast, they became very popular.

In the years after this, the hardware became more sophisticated. New options were added with every new graphics card to allow the programmer more customization of how the triangles would look. This was never real freedom, though, for only a combination of the standard settings the hardware manufacturer provided was possible. One of the most important new options that came with the graphics cards after 1999, of which the Nvidia GeForce 256 is probably the most well-known, was that they could do vertex transformations. Until then all vertices were processed at the CPU and then sent to the GPU, but now the GPU could do much of that processing itself.

While real-time graphics was living in its very limited universe, pre-rendered graphics roamed around freely. A programmer could write his own renderer and give it any option or technique he could imagine, for the CPU is not limited to specific algorithms. Already in the late 1980's Pixar had created the so-called RenderMan Shader Language. In this language a programmer could write his own so-called "shaders", little programs that calculate colour values. This allowed for a lot of freedom within the bounds of Pixar's own RenderMan rendering system. The contrast could not be bigger: while non real-time renderers allowed the programmer to make anything he wanted, real-time graphics only allowed a limited set of very specific options.

By 2001, languages like the RenderMan Shader Language inspired a similar, although much more limited, technique on GPU's. With the release of graphics cards like the ATi Radeon 8500 and Nvidia Geforce 3, cards allowed the programmer to write small programs that can run on the GPU and process the vertices and pixels. Now much more could be done in real-time rendering than in the past, because not just a combination of the standard options the hardware manufacturer had created was possible, but new techniques could be programmed.

These first versions of small shader programs were very limited, both in size and in options. In the years since 2001 new graphics cards have been released that allow larger and more complex programs to run on the GPU. They are still not as flexible as pre-rendered graphics, but real-time graphics has made a great leap forward and can now render much more advanced and customized effects, while keeping its interactive framerates.

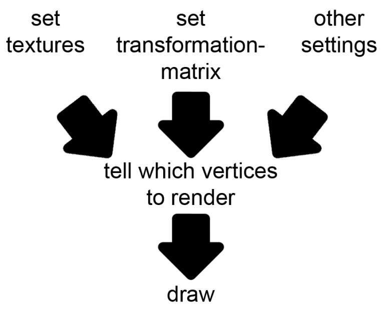

Figure 1: *The steps to render an object.*

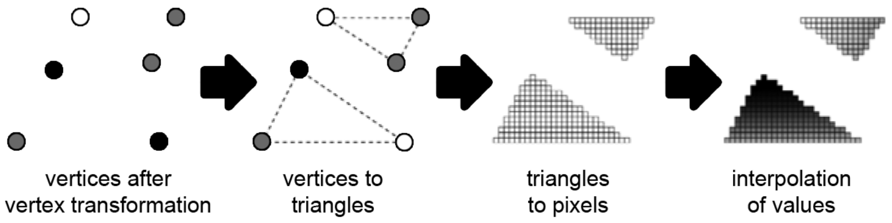

Figure 2: *Triangle rasterization.*

3 Rendering triangles

To understand the way programmable shaders operate, it is necessary to understand the basics of how objects are rendered. First, lets take a look at figure 1. To let the GPU render an object, it first needs to know which textures are applied to it, what its transformation matrix is and what rendering settings are used. Examples of some rendering settings are whether to render with transparency and whether to render triangles that do not face the camera. Then the GPU is told which vertices to render with these settings and finally to actually start rendering. When in the next paragraph of this article programmable shaders are added, they are sent to the GPU in the same stage as the setting of the transformation matrix.

During the rendering of a triangle, the hardware performs a number of steps [1]. This is illustrated in figure 2. The first step is to transform the vertices from object space to view space by multiplying by the transformation matrix. The triangles are then assembled from the vertices. The next step is to rasterize the triangles: they are turned into actual pixels that can fill the triangle on the screen. Finally, the values from the vertices are interpolated and combined with textures to get the different colours for each specific pixel. The values that are interpolated, are data stored with the vertices or calculated by the GPU. The most important example of these is

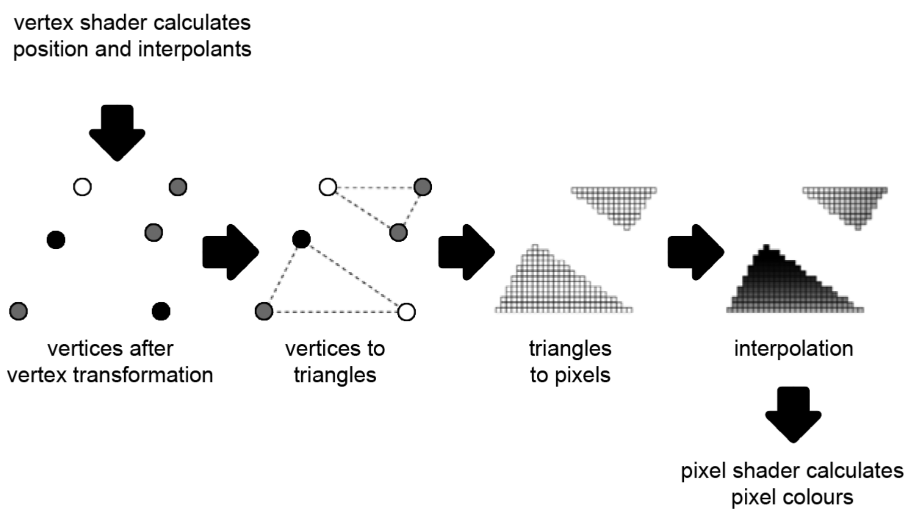

Figure 3: *Triangle rasterization with shaders.*

the lighting: the hardware calculates the lighting at the vertices and then simply interpolates these values over the pixels, so lighting is not calculated per pixel, but per vertex.

The final stage is to mix the calculated pixel colours with the entire image that is being rendered. In most cases, this means checking with the z-buffer to see whether the pixel is not hidden behind another object. If this test succeeds, the newly calculated colour overrides the old colour at that pixel. More advanced effects may be used here, though, for instance to create transparent effects.

4 How programmable shaders work

Now that we know why we want the freedom of programmable shaders and how triangles are rendered, how do shaders work? The basic idea is that there are two small programs. One is the vertex shader, the other is the pixel shader, also know as the fragment program. For every object that the programmer sends to the GPU to be rendered, he can define a vertex shader and a pixel shader and send these shaders to the GPU. Now for every vertex in the object, the GPU calls the vertex shader to do its work. The vertex shader positions the vertex in view space and pre-calculates the values the pixel shader needs. After this the triangles of the object are rasterized. For each pixel, the pixel shader is called to calculate its colour. The vertex shader can actually be seen as operating before the parts of figure 2, while the pixel shader operates after them. This is shown in figure 3.

To make interesting calculations, the shaders need information. What

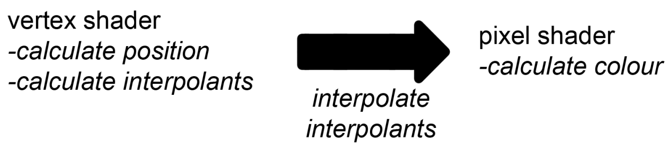

Figure 4: *The flow from vertex shader to pixel shader.*

information can a vertex shader get from where? There are two kinds of variables that a shader can use: uniform variables and specific variables. Uniform variables are the same for the whole object. In the vertex shader, examples of this are the projection matrix of the object and maybe some settings specific for this object. The other type of information the vertex shader gets is specific for the vertex that is being processed. This is usually the position of the vertex and some extra information, like its normal, its texture coordinates and its vertex colours. The most common use of this information if to move the vertex position, which is in object space, to view space by multiplying it by the projection matrix for the whole object.

The variables a pixel shader can get are divided into uniform and specific variables as well. The uniform information has about the same properties as it has at the vertex shader: it contains things specific for the whole object. This time however, this will typically be a number of textures and maybe some material properties like the strength of the specular lighting. The specific variables the pixel shader gets are called interpolants and come directly from the vertex shader. At each vertex, the vertex shader calculates some values, as defined by the programmer. The position in view space was already mentioned, but it may output other information as well, like the vertex normal and the texture coordinates. Each triangle has three vertices, so at every vertex different interpolants may have been calculated. Now for the pixels in the triangle, these interpolants are interpolated to get specific values for every pixel.

Both shaders can perform calculations with the variables they receive. A simple example is that the vertex shader may calculate a vector from a light to its position and pass this on to the pixel shader, together with the normal and the texture coordinates of the vertex. Now the pixel shader uses the interpolated version of the normal and the light-vector to calculate a dot-product between them and thus the amount of diffuse lighting. Then the colour in the texture at the specific pixel is looked up, using the interpolated texture coordinates from the vertices and the uniform texture that was given to the pixel shader. Finally this colour is multiplied by the diffuse lighting and the resulting value is outputted to the screen. This simple example shows that interpolants can represent anything the programmer wants, including vectors and texture coordinates.

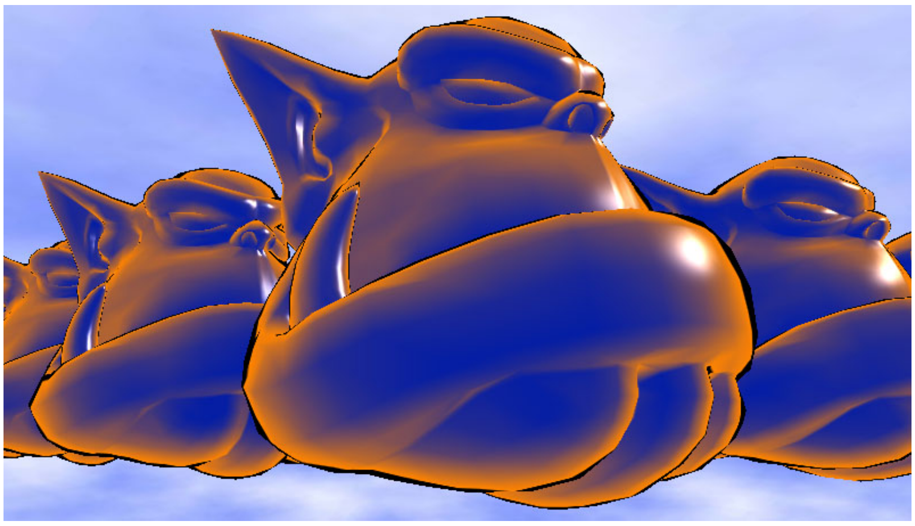

Figure 5: *An image rendered with the falloff and specular shaders. The model that was used comes with the Ogre-engine.[2]*

5 A detailed example: falloff and specular lighting

The best way to really understand how shaders work, is to look at a practical example. The following example may seem a bit exotic at first, but it illustrates all the major concepts of writing shaders. Also, writing shaders is all about freedom, so creating non-standard effects is one of the reasons programmable shaders are so exiting.

Shaders are written in a specific language. Of these languages more will be told later on in this article. For now it suffices to know that this example is given in Cg, a language much like C.

The shader program that is being discussed here calculates a falloff colour, which changes towards the edges of an object, and specular lighting. In figure 5, a screenshot can be seen of how this looks on a simple model.

Before diving into the code, let's see how these colours are calculated. First the falloff. The idea of falloff is simply to choose the colour depending on the angle between the surface-normal and the eye-vector. This is illustrated in figure 6. Artists can define the look of the objects by creating the 1D-texture in which the colour is stored for each angle. In figure 6, an example is seen of a gradient from blue to orange, where blue is the colour that is choosen for triangles that face the camera. Because the left part of the texture is made black, the objects are rendered with a black outline.

The other part of this shader is blinn-speculars. These are calculated by first calculating the so-called half-vector, which is the average of the eye-vector and the light-vector, as is shown in figure 7. Then the cosine of the

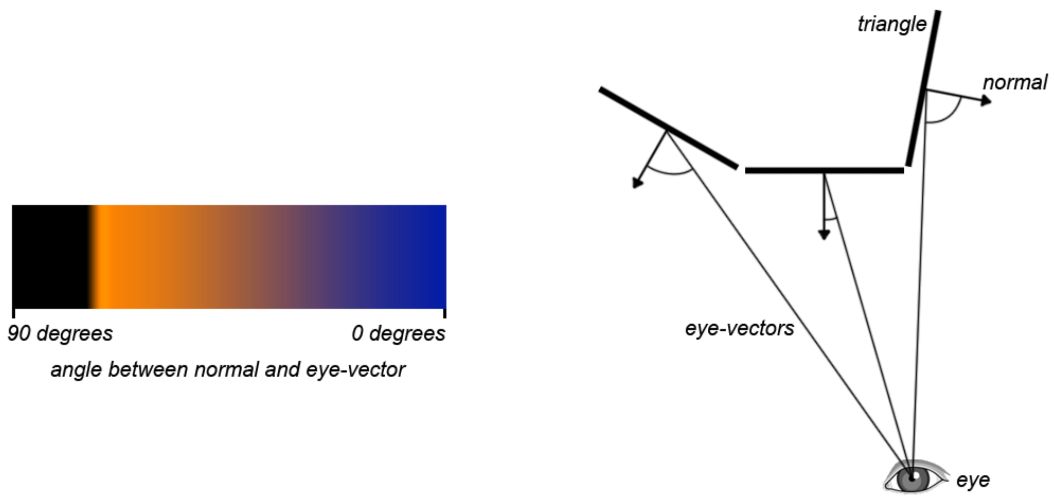

Figure 6: *On the left the 1D-texture that defines the colour. On the right the angles between normals and eye-vectors.*

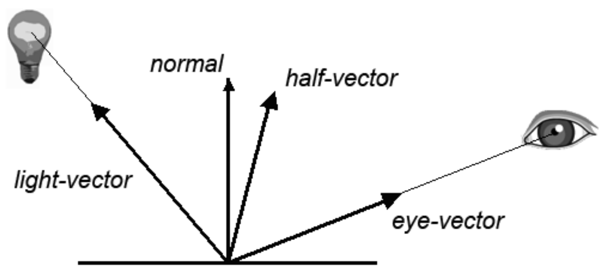

Figure 7: *The half-vector is the average of the light-vector and the eye-vector.*

angle between the half-vector and the surface normal is calculated. To get a more sharp specular, as in the picture above, this angle is taken to a power. In figure 5, this power was 100.

Below is the actual code for the combination of a vertex shader and a pixel shader that calculates the falloff and specular and generated the render in figure 5. The vertex shader:

```
void vertex_shader(float4 inPosition : POSITION,
                   float3 inNormal   : NORMAL,

                   out float4 outPosition   : POSITION,
                   out float  outFalloffU   : TEXCOORD0,
                   out float3 outNormal     : TEXCOORD1,
7.                     out float3 outHalfVector : TEXCOORD2,
8.
9.                     uniform float3   lightPosition,
10.                    uniform float3   eyePosition,
11.                    uniform float4x4 transformationMatrix
12.                   )
13. {
14.     outPosition = mul(transformationMatrix, inPosition);
15.     float3 eyeVector = normalize(eyePosition - inPosition.xyz);
16.     outFalloffU = dot(inNormal, eyeVector);
17.     float3 lightDir = normalize(lightPosition - inPosition.xyz);
18.     outHalfVector = normalize(lightDir + eyeVector);
19.     outNormal = inNormal;
20. }
```

And the pixel shader:

```
21. void pixel_shader(float  inFalloffU   : TEXCOORD0,
22.                   float3 inNormal     : TEXCOORD1,
23.                   float3 inHalfVector : TEXCOORD2,
24.
25.                   out float4 outColour : COLOR,
26.
27.                   uniform sampler1D falloffRamp
28.                  )
29. {
30.     float4 falloffColour = tex1D(falloffRamp, inFalloffU);
31.     inNormal = normalize(inNormal);
32.     float specular = dot(inHalfVector, inNormal);
33.     specular = pow(specular, 100);
34.     outColour = falloffColour + specular;
35. }
```

So, how exactly does this work? Lets start with the simple observation that two functions are defined here: a vertex shader and a pixel shader. Although both functions specify that they return void (nothing), they do return something. This is specified in lines 4 to 7 for the vertex shader and line 25 for the pixel shader. The pixel shader returns only the colour of the pixel, *outColour*. The vertex shader returns not only its position in view space, *outPosition*, but also its normal, the half-vector for the specular lighting, and the 1D texture coordinate to look up the falloff colour.

In lines 1 and 2, the vertex shader receives the parameters that are specific for this vertex: its position and normal, both in object space. In lines 9 to 11 it receives the uniform parameters that are constant for the entire object. These are the position of the light and the eye, both in object space, and the matrix that transforms the object from object space to view space.

In lines 14 to 19 the vertex shader actually performs its work. In line 14 it transforms the position of the vertex from object space to view space by multiplying by the transformation matrix. In line 15 the vector from the eye to the light is calculated and in line 16 the dot-product of this vector with the normal is calculated. It is not necessary to transform the normal from object space to world space, because the positions of the eye and the light are in object space. In line 17 the vector from the light to the position of the vertex is calculated and this is used in line 18 to calculate the half-vector. In line 19, the normal is simply copied from the input to the output. Thus ends the vertex shader.

These values are now used by the pixel shader to calculate the colour of the pixel. The pixel shader receives a so-called *sampler1D*, which is a 1D-texture that is uniform for the entire object. It also receives the interpolants that the vertex shader outputted. Note that the vertex shader also calculated the variable outPosition, but as we do not need it here, is it ignored in the pixel shader. The output of the vertex shader is linked to the input of the pixel shader using keywords such as *TEXCOORD0*.

Lines 30 to 34 are the body of the pixel shader. In line 30 the function *tex1D* is used to look up the colour in the texture at the texture coordinate *inFalloffU*. This shows how the angle between the eye-vector and the normal that was calculated in the vertex shader, is now used as a texture coordinate in the pixel shader. In line 31 the normal is re-normalized. This is necessary, because the normal was interpolated between the vertices and is not necessarily of length 1 anymore. In line 32 the cosine of the angle between the normal and the half-vector is calculated using a dot-product. In line 33, the resulting specular-value is taken to a power of 100 to get a sharp specular. Finally, the specular-colour and the falloff-colour are added and returned as the colour for this pixel.

Now that it is clear how these two shaders work, there are some important things to note. The first is that in the pixel shader, the half-vector is not normalized, while the normal is. This is actually not correct: the half-vector was interpolated just like the normal and should have been normalized to give it length 1 again. However, we are dealing with real-time graphics here, so any optimisation that does not visibly influence the final image, is allowed. To verify that the specular looks smooth without this extra normalization, just take a look at figure 5 again. Programmable shaders significantly decrease the framerate, so where possible, shader programmers try the optimise them and this is a nice example of the kind of optimizations that are often possible. When the size of the triangles becomes very large in proportion to the distance of the camera and the light, this normalization may look bad, so care should be taken with this kind of optimization.

The second thing to note is that the normal and half-vector are stored in variables with the keywords *TEXCOORD1* and *TEXCOORD2*, even though they are not texture coordinates. This is done because the number of keywords available for interpolants is limited and there is no *VECTOR*-keyword to identify them. This illustrates an important aspect of writing shaders: in general the options of the hardware are limited and optimisation if very important, so code is often not very elegant and may look like a hack. This is not a huge problem, though, because most shader-programs are not very complicated.

6 Some short examples of the wide range of possibilities

In the previous paragraph, an example of how vertex shaders and pixel shaders work has been discussed. To give some insight into the possibilities of shaders, some examples of what shaders can do are now discussed. These examples where selected to show the broad range of possibilities shaders offer.

Animated vertices

The first example is animated vertices. This is something that is purely done in the vertex shader. In the example-shader in the previous paragraph, the vertex shader simply transformed the vertex from object space to view space and left it at that. More interesting effects can be achieved by animating the vertices in the vertex shader. This can be done by passing the time since the program started as a uniform parameter to the vertex shader and moving the vertices around depending on this time.

Some very interesting effects can be created this way. One possibility is to create ocean-waves, or to animate grass that moves in the wind [2]. A screenshot of such grass is shown in figure 8. The grass is modelled using planes that are perpendicular to the ground. On these planes, a transparent image of grass is used as the texture. Now the upper vertices of the planes are moved left and right by a sine-function that takes the time as its input, giving the impression of the grass moving in the wind. This is a very simple example of how animations can be made using vertex shaders.

It might be objected that this is not really a new feature introduced by vertex shaders: animating vertices could already be done by editing the vertices in memory using the CPU. However, this is so much slower than by the use of vertex shaders, that the latter has opened up a world of new possibilities.

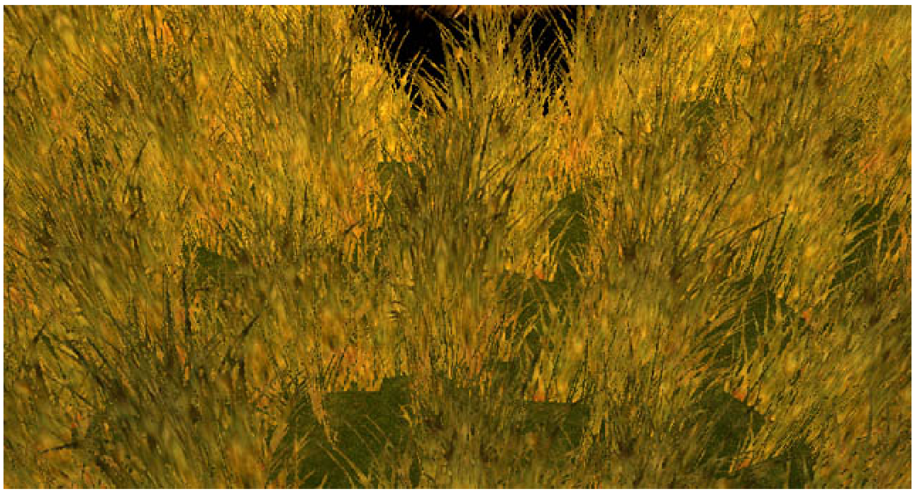

Figure 8: *Animated grass made using textured planes.[2]*

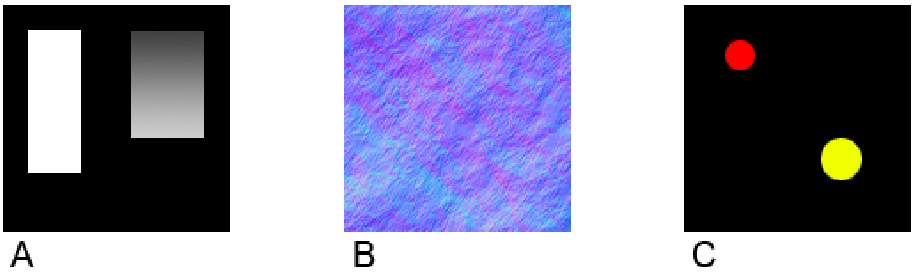

Figure 9: *From left to right: a specular map, a normal map and a self-illumination map*

Specular-mapping, normal-mapping, and everything-mapping

Traditionally, textures could only be used for some specific effects the GPU offered. This usually meant storing the colour and the transparency in a texture and nothing more. With programmable shaders, the data in textures can be interpreted as anything the programmer wants. A simple example of this is specular mapping. An extra greyscale texture is added to an object and the pixel shader uses the value in this texture to compute the strength of the specular highlights. This way an artist can mark parts of the object that are very shiny with white in the specular map, and parts of the object that are very dull with dark grey. An example of a specular map can be seen in figure 9 part A. This is simply a greyscale-texture and the values can be interpreted as anything we want. Here they are interpreted as the strength of the specular highlights.

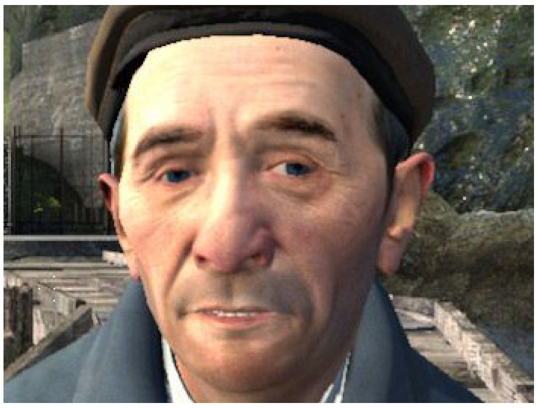

Figure 10: *Normal mapping in action in Half-life 2.[3]*

Another example of storing data is normal mapping. Here the values in the texture for the red, green and blue colour are interpreted not as colours, but as 3D-vectors. RGB is simply interpreted as XYZ. In standard lighting techniques, the normals at the vertices are interpolated and used to calculate lighting, as was done in the example-shader in the previous paragraph. With normal mapping, not the vertex normal is used, but the normal that is defined in the texture. This way the lighting of highly detailed surfaces can be simulated on surfaces with only a few polygons. An example of this can be seen in figure 10, where the wrinkles in the man's skin are not in the mesh, but in the normal map. An example of what a normal map may look like, is visible in figure 9 part B, where the details of a rocky surface are put in a normal map.

Normal mapping and specular mapping are only two of the most-used examples of storing data in textures. Another example is storing the self-illumination (also known as self-emmitance) in a texture, as can be seen in figure 9 part C. Here two lights are on the surface, a red and a yellow one, and these parts will be lit regardless of whether any other lights illuminate them. In fact, anything can be stored in a texture, so any property of a material can be varied over a surface. Thus, this technique could also be called "everything-mapping".

Edge detection as a post-effect

Another interesting use of pixel shaders is post effects. By first rendering the scene to a texture and then rendering a rectangle with the texture and a shader on it to the screen, the shader can apply special effects to the image the user sees. An example of this is an edge detection filter. This filter looks at the differences in brightness of neighbouring pixels and tries to mark where an edge is in the image. In figure 11 the left image is the

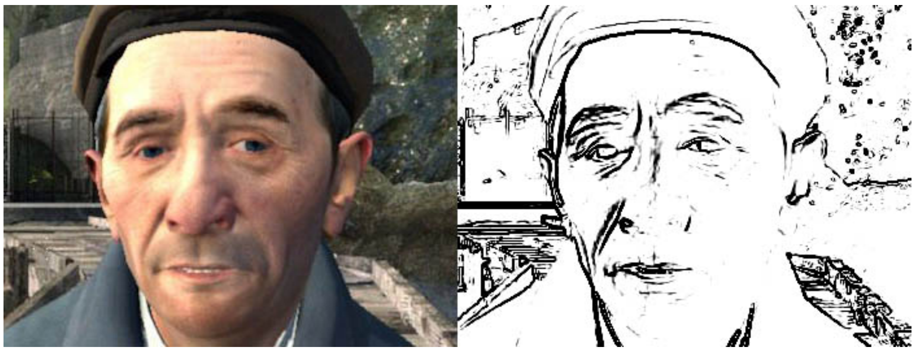

Figure 11: *To the left the original image, to the right the same image with edge detection. The image is a screenshot from Half-life 2. [3]*

original image and the right image is the same image with edge detection.

Many other interesting effects can be achieved with post effect shaders, like glowing objects and motion blur.

The Nvidia blood-shader

The standard use of shaders is for graphics, but they can be used for many other effects as well. An example of this is the Nvidia blood-shader [4], which uses a shader to simulate the motion of a fluid (blood) over a surface. Although a visualisation of the blood is part of this technique, the essence is doing physics simulation in a pixel shader.

In figure 12, a screenshot of the blood-shader can be seen. The blood flows down and leaves a trace where it has been. The pixel shader works as follows: an image is made of where the blood is at the beginning. During a frame this image is used as a texture and rendered to another texture, while the pixel shader does the physics simulation. In the next frame, the resulting texture from the previous frame is used as the input for another simulation step. So in frame 1 image A is rendered to image B and in frame 2 image B is rendered to frame A. Then frame A is rendered to frame B again, and so forth.

The actual simulation step is done by passing a gravity-vector to the pixel shader and a normal map that defines the details of the surface. For every pixel, the shader looks at its surroundings to see whether there is any blood in the vicinity. If there is, the orientation of the surface and the gravity are used to calculate whether the blood reaches the current pixel.

Each frame, the final step is to put the texture with the blood on it on the actual object and render this to the screen.

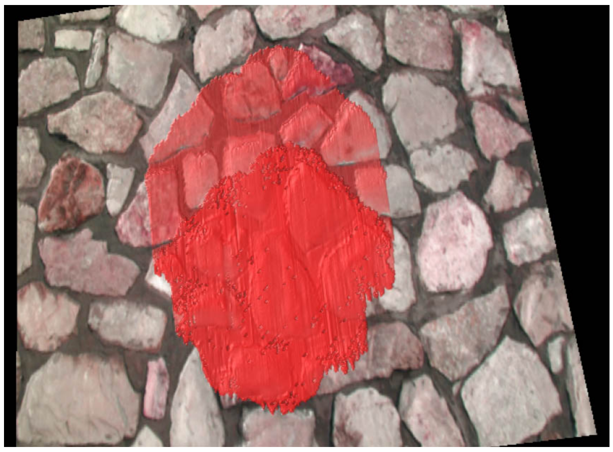

Figure 12: *A screenshot of Nvidia's blood shader in action. [4]*

7 Shader languages

The example shader that has been shown in paragraph 5 was written in the shader-language Cg. This is not the only language available for writing shaders. As the possibilities of programmable shaders are limited mostly by hardware and not by software, other languages offer roughly the same functionality and differ only in syntax. Usually for one project one language is chosen, although this is not necessary. Which language is best to use, is different for each project. These are the available choices:

- Originally, **assembly** was the only available language. Like other assembly-languages, this means ultimate control, but reading and writing shaders is very hard. It is not often that assembly is still the best choice today, except when very exact control over the operation of a shader is required.
- **Cg** is for OpenGL and DirectX and was developed by Nvidia. It is very similar to C, which is where Cg's name comes from ("C for Graphics"). Although one might not expect it, Cg is not limited in any way to Nvidia graphics hardware and will run on ATi-cards just as well.
- **HLSL** was developed by Microsoft and will run only on DirectX. It is very much like Cg. Both Cg and HLSL started as the same project by Microsoft and Nvidia together, which explains the similarities between the two [1].
- **GLSlang** is only for OpenGL. It is quite similar to Cg and HLSL.

8 Shader versions and limitations

In the preceding paragraphs the great possibilities of programmable shaders have been shown. Unfortunately, the world is not as beautiful as it may seem to be at this point. As the shaders discussed in the article are designed for use in real-time graphics, they cannot be made arbitrarily complex. Especially on older hardware, the number and type of instructions that are possible is heavily limited. These restrictions often require some inventive programming and ugly code to achieve interesting effects. For different generations of hardware, different versions of pixel and vertex shaders are available. What is possible in which version, is listed in the two tables below[5][6]. Hardware that supports a specific version always supports all previous versions as well.

Pixel shader version	Hardware	Limitations	Special
ps 1.1	Geforce 4, Radeon 8xxx	8 instructions + 4 textures reads	Hardly any standard functions are available. For example no square root or sine.
ps 2.0	Geforce FX, Radeon 9xxx	64 instructions + 32 textures reads	It is now possible to calculate the UVW-position on which to do a texture-read from the result of another texture read. This is called "dependent read".
ps 3.0	Geforce 6, Radeon X1xxx	∞ instructions	Controlling program flow using if/then, for-loops and while-loops is now possible

Vertex shader version	Hardware	Limitations	Special
vs 1.1	Geforce 4, Radeon 8xxx	128 instructions	
vs 2.0	Geforce FX, Radeon 9xxx	256 instructions	
vs 3.0	Geforce 6, Radeon X1xxx	∞ instructions	Controlling program flow using if/then, for-loops and while-loops is now possible. Also, textures can now be read from the vertex shader.

These are not all existing versions and not all restrictions and possibilities are shown here. Ps 3.0 and vs 3.0 were at the time this article was written the most recent versions.

To get the most out of each type of graphics hardware, it is common practice to write different versions of the same shader for different GPU's. This way ps 3.0-hardware can get the most fancy graphics, while ps 1.1-hardware can render more simple effects.

The limitations on shaders are fading fast with every new generation of graphics hardware. As new GPU's become more commonplace, it will be less important to cope with the heavy restrictions of older shader-versions, especially the very limited ps1.1.

9 Conclusion

This article has shown how programmable hardware shaders have increased the possibilities of real-time graphics. Today programmers can achieve complex effects without being limited strongly by what options hardware manufacturers have put in the GPU, thus bringing democracy to real-time graphics. Any programmer who has basic knowledge of graphics math can write vertex and pixel shaders. Most shading languages are very much like normal programming languages and many tutorials are available on the internet and in specialised bookstores. Some useful tutorials and manuals are given in the references-section below [1][5][6]. A good way to learn more about writing shaders is by looking at existing shaders. Some places where lots of existing shaders can be found are listed in the references-section as well[7][8].

Although writing shaders is not very hard, it requires creativity to design new and interesting effects. However, no matter how creative a programmer is, the restrictions of raw processing power remain: a large scene with complex shaders lowers the framerate and real-time graphics requires a minimum framerate. Implementing shaders that are aesthetically pleasing without burdening the GPU too much is an art by its own right.

References

[1] R. Fernando and M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Professional, 2003. ISBN 0321194969.

[2] Ogre-team. The ogre3d-demos (graphics demo's). `http://www.ogre3d.org`, 2004.

[3] Valve. Half-life 2 (game), 2003.

[4] K. Myers. Bloodshader - real-time adaptive animation). `http://developer.nvidia.com`, 2004.

[5] Nvidia. The cg-manual - cg toolkit user's manual - a developer's guide to programmable graphics - release 1.2. `http://developer.nvidia.com`, 2004.

[6] Microsoft. Hlsl shaders - msdn direct 3d programming guide. `http://msdn.microsoft.com/library`, 2006.

[7] Nvidia. Fx-composer (software tool). `http://developer.nvidia.com`.

[8] ATi. Rendermonkey (software tool). `http://www.ati.com/developer/rendermonkey`.

The image on the cover is part of a screenshot of the game Far Cry Instincts Predator for the XBOX 360, released in 2006, developed by Crytek Studios and published by Ubisoft.