WHAT ARE THE SHADERS AND HOW THEY WORK?

Norbert Zsolt Zentai – Sándor Kaczur

1. ABSTRACT

This project has started as a simple learning urge, and developed itself to a modern shader based rendering engine. The rendering engine called NZEngine (developed by Norbert Zsolt Zentai) was presented at Dennis Gabor College's Students' Scientific Association in 2009 [1], and it won the first prize. Due to our teacher's recommendation, three students together made a presentation showing how exactly rendering works, and they were able to go to Finland for an ERASMUS Life Long Learning Intensive Programme in spring 2012 [2]. Meanwhile, the rendering engine has been enhanced. This article is the important part of the bachelor thesis of Norbert Zsolt Zentai, which has already been made [3].

2. BEFORE SHADERS

Before shaders existed, video cards had hard wired algorithms which the programmer could use along with some built-in parameters that could be altered. The programmer could not create new algorithms that could not be built only on top of existing ones and had no way of creating something that the video card manufacturer has not thought about. The hard wired concept was not in use because everyone thought that this would be enough, but since video cards were not fast enough to be able to run small programs [4]. As the need grew and the video cards became faster and faster the manufacturer started to implement popular effects like normal mapping, and they added custom extensions to OpenGL. The OpenGL is an open standard which the programmer could use [4]. Manufacturers quickly realized that this could lead to danger as they implement different effects using different approaches. The fragmentation would have been inevitable.

3. THE SHADERS

The solution was not to implement those effects but rather create something even lower level which can be used to implement already existing hard-wired algorithms. If this goal can be met, the programmer should be able to use it to create more complex algorithms. The solution is the shader. The shader is a small program which the video card can run at different pipeline stages. This program is compiled by the driver of video card and then is uploaded to the video card's shader units [5]. Those shader units are composed of program storage and executer units to which the video card's pipeline could write and read from.

The first shader was not written using OpenGL's Shading Language GLSL or Direct3D's HLSL. Video cards supported shader running before these were created. Shaders were written using shader assembly which is an assembly styled language with a different instruction set specifically for vector math. This was a huge step from the previous hard wired pipeline, but because of assembly's very low level nature, it was very hard to be written. To solve this, GLSL and HLSL were born [5].

4. THE TYPES OF SHADERS

The two most common types of shader are the vertex shader and the fragment shader.

4.1 The vertex shader

Vertex shader runs at every vertex. These vertices are essentially control points from which the vertex shader calculates the output for the fragment shader. The main task for the vertex shader is to calculate the vertex position in screen space. This shader has two types of input. The first one is called ***uniform.*** Uniforms are constant values during the shader run but can be modified between these runs. A typical uniform is the light's position to the body that we want to render or the body's model view matrix which is basically the body's position, the projection matrix which is used to project the 3 dimensional point into the 2 dimensional screen. These properties are not changing while we render the body. It would look really weird how for one vertex the body is facing north, yet for the next one it's facing west [9]. The second type of input is called ***attribute.*** Attribute is similar to the uniform in that this also cannot be altered during the shader run, but these supplies

different values [3]. Of course, these values are not random ones; these are typically the vertex's position in 3 dimensional space or the vertex's normal vector, color texture coordinate. The output from the vertex shader's calculation is called varying.

4.2 The fragment shader

The second type of shader is the ***fragment shader.*** The fragment shader runs after three vertex calculations is ready. Three vertex calculations will produce three vertex positions thus creating a triangle. From this triangle, the video card allocates shader units for fragment calculation for each generated pixel. The fragment shader's task is to calculate the color of that pixel. The fragment shader has access to uniform values just as the vertex shader, and it also has access to the vertex shader's outputs, the varying. If the vertex shader calculates the varying, three of them are needed for the fragment shader stage which one of the varying version should be used for the fragment shader? The answer is it should be weighted for the specific pixel. If the pixel is closer to one of the vertex, it is a value should be more dominant than the two others. The varying is a linear-interpolated value for the fragment shader. For example [7], vertex "A" produces an output of [1;0;0], a vector with three components, the vertex "B" produces [0;1;0] and "C" [0;0;1]. The fragment shader for the pixel right on top of "A" position receives [1;0;0]. A fragment shader for the "A"-"B" edge receives [0,5;0,5;0]. The only output of the fragment shader is the pixel's colour [6]. Because it runs for each pixel, the optimization of this is critical. Everything which can be calculated in the vertex shader should be moved there. You can even sacrifice some graphical fidelity in exchange for much better performance by calculating light values in the vertex shader than using interpolation in the fragment shader.

4.3 The problems and solutions

This control point approach is used because the video card is a parallel processor capable of running literally hundreds of threads simultaneously. This makes a huge synchronization issue, where you must know when the shader unit is finished and accepts new inputs when is the bus free for shaders to get their inputs. This control point approach solves this problem, the synchronization is much easier when you know that every shader unit runs the same program in the same clock. Also, you can calculate the necessary amount of tick for the shader – so you know – that if you start the calculation after the calculated amount of tick, every unit will be ready; you do not need to query every single one of them, also you will know what every shader would need as an input for the specific tick. But this does create a new problem which is the branching. What if your shader uses an ***if*** statement? If one branch requires 10 ticks, the other requires 12, then you bring up the same problem again, that you cannot be sure when all of them are finished, not to talk about the input feeding. If one branch requires the value from one of the texture the other from the other requires a single number, you cannot prepare the texture lookup, the main control unit would need to read the shader's state on which one does it need, the other shader would be processing the other branch, but since it did not get the value, it would use garbage which it found by the same accumulator to which the input would have been wrote, it would create chaos. But we need branching, there are situations in which branching is a must, we cannot use a different math for it. For this, the shader unit calculates every branch and then, it masks out the one needed one [3]. This of course slows the shader down, so it is generally a good idea not to use any branching. If you know, that for example you do not need the specific branch for the current rendering, you should compile a new shader which does not include it. For example if you do not need light calculation for that frame, do not include it in the shader, create a new one without it, and if you need one for the next frame, just switch out the shader. It is recommended to use multiple shaders than use branching to turn effects on or off.

There is also another problem. Where you do not run every shader, because there is not enough pixel to calculate. You cannot start the calculation of a new triangle, because its pixels could overlap the ones under calculation. The solution for this problem isn't to use small triangles, but rather lower the quality of the body's geometry if it is far. The video card manufacturers came up with another solution which is called tile-based deferred rendering [5]. The video card's driver buffers every command, slices up the screen into tiles, and then pushes the commands to the video card. The video card starts the calculation for each tile, and if a fragment shader frees up, the main control unit send that shader processor to render another tile. Than, it is impossible to overlap other renderings, as the tiles do not overlap. The driver can also prioritize each tile, so if statistics of one tile requires more time, the driver allocates more shader units to that tile. By buffering the commands, if the driver detects, that the position for

each vertex is easy to calculate, like a standard 3D projection, it pre-calculates those and it drops calculation for the pixels which would be overridden by another calculation.

5. A SIMPLE EXAMPLE

Let us look at a basic shader. This shader will fill the geometry with a single color. The shader accepts 2 dimensional vectors as the vertex's position and it will not transform them [8].

```
attribute vec3 position;
void main() {
    gl_Position = vec4(position.x,
                       position.y,
                       0.0,
                       1.0);
}
```

Figure 1.
A basic 2D vertex shader

We need an attribute which is the position of the given vertex. Because this shader is a 2 dimensional one, the position is given in a vector with two components. The next part is to write the main block. This is the function the video card invokes when it runs the shader. Here, the shader has a magic variable called gl_Position, and its type is a vector with 4 components (Fig. 1). As this shader is not interested in doing 3D projection it just transforms the position into a vector with 4 components. The fourth component of that vector is 1.0, this is used for projection, and it is best to set this to 1.0 if we do not want to use it. See how has been written 0.0 and 1.0 even when in theory I could have written 0 and 1. GLSL is a very strictly typed language where 1 is strictly an integer and 1.0 is strictly a floating point number, and vec4 only accepts floating point numbers.

So, we have the vertex shader; let us take a look at the fragment shader (Fig. 2).

```
uniform vec4 color;
void main() {
    gl_FragColor = color;
}
```

Figure 2.
A basic colour fill fragment shader

Here we get our color as an input. We could just use a literal but that would make our shader too specific and it would not be reusable. The color is a vector with 4 components, as we can pass the red, green, blue and the alpha channel which we can use for blending like a basic transparency. In the main function, GLSL has another magic variable which is the gl_FragColor. This is a vector with 4 components and this will be used as the final color of the given pixel. As we don't want to modify the input color in any way, we can just pass that as the final color.

Now let us look at a bit more complicated shader. This shader is going to be a 3D shader, and it will be able to texture the geometry. The vertex shader will look like as follows (Fig. 3).

```
uniform mat4 modelviewMatrix;
uniform mat4 projectionMatrix;
attribute vec3 position;
attribute vec2 textureCoordinate;
varying vec2 texCoord;
void main() {
    texCoord = textureCoordinate;
    gl_Position = projectionMatrix *
                  modelviewMatrix *
                  vec4(position, 1.0);
}
```

Figure 3.
A 3D vertex shader with texture support

For 3 dimensional projection, we need to know the body's position, rotation, scaling, skewing. For this we use matrices, and the matrix that represents the body's state relative to the camera's position is called model view. This is a 4x4 matrix. For the projection we use a matrix that will distort the image from being orthogonal into having a perspective effect. This is also a 4x4 matrix. Then we need the vertex's position is 3 dimension and the texture coordinate which is a two dimensional vector. Texturing is done by stretching the image. The stretching is done by assigning a texture coordinate for the given vertex. This value describes what point of the texture should be at the given vertex. Because we use shaders, we must somehow tell the fragment shader what's exactly its texture coordinate. We need to interpolate the triangles texture coordinate for the generated triangle. Fortunately, varying provides exactly that. We simply assign the vertex's texture coordinate to this varying and let the video card interpolate this for the fragment shader. To calculate the vertex's position in screen space, thus projecting it, we use linear algebra. Because we would multiply a mat4 with a vec3, the shader compiler would throw an error telling us, that there is no such operator that can take a mat4 on its left and a vec3 on its right. The solution is to create a vec4 out by just calling vec4, giving the vec3 position as the first argument and a 1.0 as its second.

Vec4 has an overloaded method that interprets this as having the first three values from the vec3 and the fourth from the second argument.

```
uniform sampler2D texture;
uniform vec4 color;
varying vec2 texCoord;
void main() {
    vec4 texColor = texture2D(texture,
                              texCoord);
    gl_FragColor = texColor * color;
}
```

Figure 4.
A fragment shader with texture and tinting support

At first, we have to know, what's the texture we need to read from it. This is done via giving the sampler unit where we did bind our texture. This is basically the id of the sampler unit. Next, for some flexibility, we take a color which we can use for tinting the texture, giving it transparency. Finally, we take the texCoord from the vertex shader. In the main function, we query the texture's color at the given coordinate by calling texture2D, giving it the sampler id and the query coordinate. This function returns a vec4, and we store it in our texColor variable. Then we just assign the texColor multiplied by the uniform color to the gl_FragColor (Fig. 4).

For performance reasons, it is the best way not to modify any variable which is used for texture lookups in the fragment shader, as it would kill the video card's texture color pre-fetch, and would also disable the video card's automatic mipmap level calculation.

6. SUMMARY

Nowadays, shaders are present almost in every game. All modern desktop video cards, and even some mobile video cards support it. Old pipeline does not have any advantage, because modern video cards do not include them, but rather emulate it by dynamically creating new shaders for the specific render state. If you would like to learn more about shaders, we recommend using some OpenGL shader builder where you can leave out the OpenGL setup code.

Shaders are not used only for games, they can be quiet useful in big 3D applications as well. The principle is pretty much the same, only the hardware which runs it is different in being a CPU, not a GPU for greater accuracy and control. Shader logic can also be used in science for complex field calculations. Because of it being also based on multithreading, these calculations can happen across servers with extremely powerful GPUs creating the power of a supercomputer. Some graphics cards are even primarily designed for such scientific usage.

7. REFERENCES

[1] GDF honlap, TDK szekció: http://www.gdf.hu/tudomanyos-elet/tudomanyos-diakkor-tdk/tdk-konferenciak/2010, 2013.04.05.

[2] ERASMUS Life Long Learning Intensive Programme, Virrat, Finland: http://ec.europa.eu/education/erasmus/ip_en.htm, 2013.04.05. http://www.tamk.fi/digisomemarit, 2013.04.05.

[3] Zentai, N. Zs. (2012): Multiplatformos 3D Motor – NZEngine, szakdolgozat, GDF, mérnök-informatikus szak, http://ilias.gdf.hu/ilias.php?ref_id=41919&obj_id=38143&from_page=38143&cmd=downloadFile&cmdClass=illmpresentationgui&cmdNode=8n&baseClass=ilLMPresentationGUI&file_id=il__file_92173, 2013.04.05.

[4] Budai, A., Vári Kakas, I. (2007): Számítógépes grafika, Inok Kft., Budapest, ISBN 963 9625 32 7

[5] Szirmay-Kalos, L., Antal, Gy., Csonka, F. (2003): Háromdimenziós grafika, animáció és játékfejlesztés, ComputerBooks, Budapest, ISBN 9636183031

[6] Zentai, N. Zs., Ágnecz, G., Takács, R., Kaczur, S. (2012): Hardveresen gyorsított 3D/2D renderelés. Konferencia cikk, Multimédia az oktatásban konferencia, 2012

[7] Kopácsi, S., Kaczur, S. (2008): Practical application of coordinate and dot transformations. A GAMF Közleményei, Kecskemét, XXIII. évf., HU ISSN 1587-4400, p.121-126

[8] Hajós, Gy.: Bevezetés a geometriába, Nemzeti Tankönyvkiadó, Budapest, 1971, ISBN 963 18 6771 4

[9] http://developer.apple.com/library/ios/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/OpenGLES_ProgrammingGuide.pdf → Avoid Misaligned Vertex Data, 2013.04.05.