Writing Shaders - An Introduction

Notes for a Msc Course in Computer Graphics
University of Minho
António Ramires
2012-10-25

Introduction

Once buffers have been created and the pipeline is built it is time to add the shaders. In here we're going to start with the minimal shader, a sort of "hello world" for GLSL, and add features incrementally until we arrive at a customisable toon shader.

Before we start writing shaders, it is important to understand what is going on on each step of the pipeline. Let's assume a simple pipeline consisting of only two programmable stages: the vertex and fragment stages.

Let's consider that we have two buffers to feed the pipeline: a buffer with positions, and another with colors. The output will be a coloured image. Hence we can have an initial representation of our pipeline as a black box with two inputs and an output:

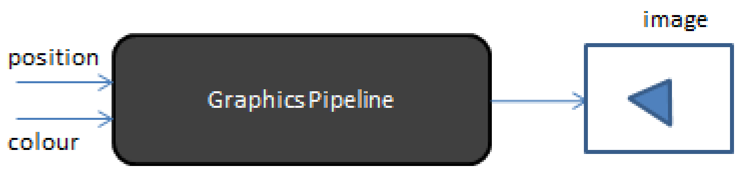


Figure 1. The pipeline seen as a black box

Inside the pipeline, the vertices are going to be processed, primitives will be built, fragments computed and coloured, until, finally, an image is produced. In more detail, we can consider the scheme in Figure 2 as a simplified depiction of the components inside the black box.

Initially, the vertices are fed, one by one, to the vertex shader. This shader processes each vertex individually, and outputs the transformed vertices. When primitive connectivity information becomes available, the primitives are assembled, and sent to the rasterisation phase. This is where the primitive is "split" into pixels, and for each pixel its attributes are computed by interpolation. The set of pixel attributes, including its position on screen, is called a fragment. Fragments are then processed, one by one, by the fragment shader, which outputs pixels to a final stage where blending and other operations take place, and the final pixel is produced.

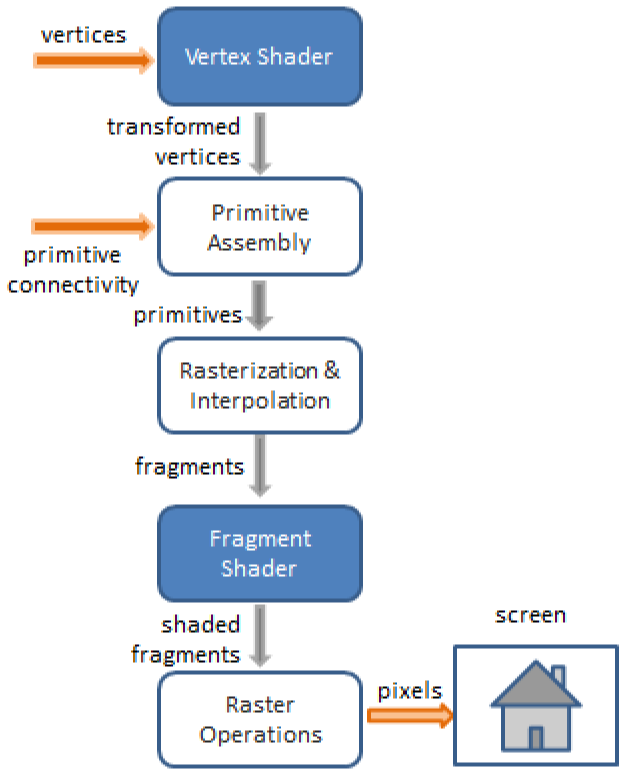


Figure 2. Simplified pipeline diagram

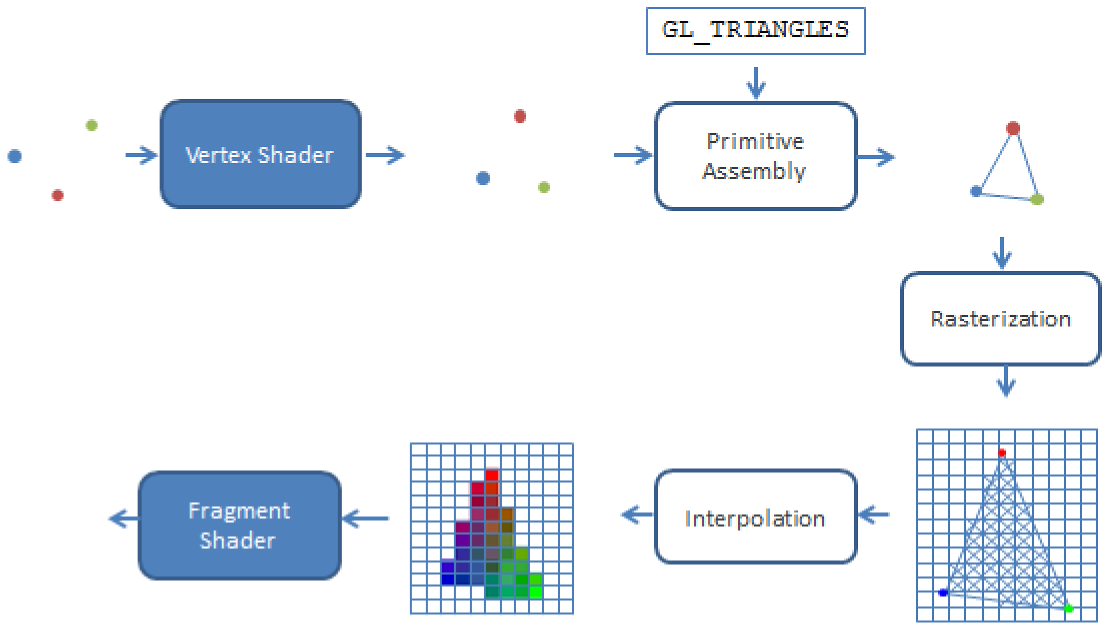


Figure 3. Data centred diagram

Another perspective is presented in Figure 3, a data centred diagram, that shows what happens in each stage.

Initially we have vertices, then primitives, and finally fragments. As shown in the diagram, after the primitive is assembled, the rasterization process generates a set of fragments. Each fragment has an immutable position in the final image. The interpolation phase is where all the attributes are going to be computed for each fragment. The interpolated attributes are then fed to the fragment shader.

The Minimal Shader

A shader should always begin with the GLSL version for which it was written. This is accomplished with pragma `#version`. As of OpenGL version 3.3, the GLSL version numbers match the GL version (multiplied by 100).

The vertex shader must receive at least the position of the vertices, each a vector of 4 floats, so we'll consider a single input attribute. The header part of our shader can be:

```
#version 150

in vec4 position;
```

Every shader unit we write must have a `main` function, similarly to the C programming language, but in GLSL there are no params and no return value. As in C, the main function may call other user defined functions. The main function has access to the inputs of the shader, in this case, for each vertex we can access its position as stored in the buffer. The simplest shader we can write is just a pass through.

The position attribute plays a particular role, since the rasterization and interpolation processes are based on it, as seen in the diagram from Figure 3. Hence, it is required that GLSL knows which attribute to use as position. This is achieved when we compute the outputs of the vertex shader, writing to a particular variable: `gl_Position`.

Hence, our main function could be as simple as:

```
void main() {
      gl_Position = position;
}
```

Moving on to the fragment shader. This shader receives a fragment, and it must output a color. The only output of the vertex shader is the position, which is a GLSL defined variable. Hence there is no need to declare any input. The shader has access to the input location of the pixel, plus its depth. However, it can not change the input location, only its depth.

The fragment shader needs to declare an output variable for the color, for instance `outColor`, and assign it a value.

The code for the minimal fragment shader will assign a constant value to this output. Here is an example of such a shader:

```
#version 150

out vec4 outColor;

void main() {
      outColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```

Adding Color - Defining Attributes

Next we’re going to add another attribute to our shader, the color. This is going to be specified for each vertex. In the OpenGL setup we now need two buffers: one with the positions and another with the colors.

The vertex shader is going to be a simple pass through for both attributes. The novelty in this example is the color attribute. Whereas for the position, OpenGL has a predefined variable, `gl_Position`, for the other attributes there is no such thing. Hence we must declare an input and the corresponding output in the vertex shader.

Since the output of the vertex shader is going to be the input of the fragment shader, it makes no sense to add a suffix, or prefix, “out” to the variable. A good strategy to keep things clear is to add the initial of the shader to the output variable. So our color output can be named `colorV`.

```
#version 150
// input attributes
in vec4 color;
in vec4 position;
// output from the vertex shader
out vec4 colorV;

void main() {
      colorV = color;
      gl_Position = position;
}
```

The fragment shader will receive the fragments with the interpolated colors. So we must declare an input variable with the same name as the output from the vertex shader. This is the most simple way for GLSL to determine which variables to pair.

The code for the fragment shader can be as simple as:

```
#version 150

in vec4 colorV;

out vec4 outColor;

void main() {
      outColor = colorV;
}
```

Setting Color as a constant per mesh - Uniform Variables

Setting the color per vertex implies providing an array with as many colors as vertices to the pipeline. In general we tend to have one color per mesh, or draw call, hence the color would behave as a constant for each mesh. GLSL allows us to define per-draw-call constants, with the qualifier `uniform`.

As the color is only required in the fragment shader we simplify the vertex shader as follows:

```
#version 150

in vec4 position;

void main() {
      gl_Position = position;
}
```

which is basically the first shader we started with.

In the fragment shader the color is not a fragment input anymore, it is now a uniform variable.

```
#version 150

uniform vec4 color;

out vec4 outColor;

void main() {
      outColor = color;
}
```

Adding Geometric Transformations - Uniform Variables

So far these shaders would draw the triangles considering that the input coordinates were already in clip space. This is because the shaders don't take into account the camera, regarding both perspective and viewpoint, as well as possible geometric transformations the object may suffer.

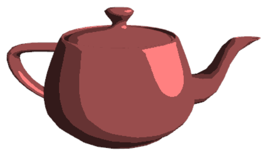
Commonly, we create a composite matrix with all these operations included, called the Projection View Model matrix. This matrix is constant per draw call, hence it is a uniform variable. The transformations in the matrix will be applied to all vertices, hence the vertex shader needs to be updated. The new vertex shader is:

```
#version 150

in vec4 position;

// pvm is a 4x4 matrix
uniform mat4 pvm;

void main() {
      gl_Position = pvm * position;
}
```

The previous fragment shader can be coupled with this fragment shader to produce exactly what fixed function OpenGL would provide with no lighting.

A Simple Toon Shader

A toon character is shaded with only a small set of tones. As opposed to real life lighting, there is no continuum of tones. In order to determine the tone to use we first compute the intensity which would be reflected by the surface according to Lambert's law for diffuse lighting.



Lambert's law takes in consideration the surface normal and the direction from the surface to the light. The intensity is proportional to the cosine of the angle between these two vectors.

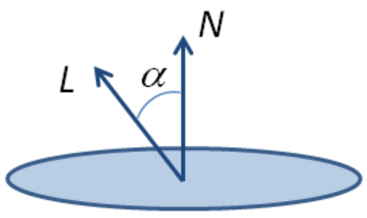


Figure 4 - Lambert's law

The equation to compute the reflected intensity I is:

$$I = K_d \times K_L \times cos(\alpha)$$

where K_d is the material diffuse coefficient, K_L is the light intensity, and α is the angle between the normal vector and the light's direction.

The dot product between two vectors can be written as a function of the cosine of the angle between them

$$dot(A, B) = cos(A, B) \times |A| \times |B|$$

When the magnitude of both vectors, A and B, is equal to one, i.e. when both vectors are normalised, the dot product is equal to the cosine of the angle between them, providing a very efficient way of incorporating Lambert's law in our shaders.

A simple algorithm to create toon lighting is as follows:

```
compute intensity I
if ( I > threshold1)
      color = vec4(1.0);
else if ( I > threshold2)
      color = vec4(0.7);
…
```

We can define as many tones as desired using the above algorithm, but commonly only three or four are used.

In the first version of our implementation, we are going to concentrate the computation on the vertex shader.

As attributes for each vertex we need the position and normal vector. We also require the transformation matrix from the previous example.

To compute the cosine between the two vectors, the normal and the light direction, these must be in the same space. The light direction can be specified in world space or camera space. Specifying the light in world space is more intuitive and does not depend on the camera position, hence, it would seem a good option. However, as we'll see in later examples, we will need the light direction in camera space, hence we're going to use camera space as our default for the light direction. This leaves us with two options:

1. Going for the more intuitive approach, we set the light direction in world space and transform it to camera space in the shader;
2. We transform the light direction from world space to camera space in the application and provide the light direction in this later space to the shader.

The second option makes more sense, performance wise, since it doesn't require the shader to perform the same calculation, transforming the light direction to camera space, for every vertex.

Since the normal vector must geometrically transformed from local space to camera space. The view-model matrix can be used to transform points and directions from local space to camera space, however, this matrix does not guarantee the preservation of orthogonality. As we'll see later we'll need a new matrix to transform normal vectors: the normal matrix.

Although the normal matrix guarantees a correct direction for the post transformed normal vector, the same can't be said about its magnitude. Assuming that the normal vector provided by the application is unit length, the transformed vector is not guaranteed to keep its magnitude, so normalization of the post transformed normal vector is required in the general case.

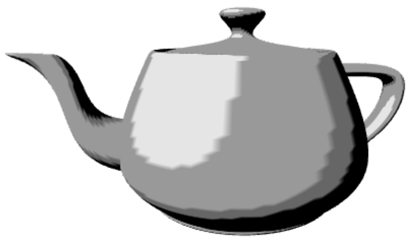
Now, lets go back to our shader. Another required uniform variable is the light direction, which specifies a vector pointing to the light in world space. As outputs from the vertex shader, besides the implicit `gl_Position`, we will define a single variable for the color.

A possible implementation for a simple toon shader is as follows. First the vertex shader:

```
#version 150

in vec4 position;
in vec3 normal;

out vec4 colorV;

uniform mat4 pvm;
uniform mat3 normalMat;
uniform vec3 lightDir;

void main() {
      // normalise both vectors
      vec3 n = normalize(normalMat * normal);

      // compute the cosine using the dot operation
      float intensity = dot(n, lightDir);

      // compute the color based on the intensity
      if (intensity > 0.9)
            colorV = vec4(0.9);
      else if (intensity > 0.5)
            colorV = vec4(0.6);
      else if (intensity > 0.3)
            colorV = vec4(0.4);
      else
            colorV = vec4(0.0);
      gl_Position = pvm * position;
}
```

The fragment shader is the same as in the previous example, it receives the interpolated colors for each fragment and outputs them.

The result we get with this pair of shaders is similar to the following figure:



As we can see the result is far from perfect. This is what we get when we compute colors per vertex, using the Gouraud lighting model.

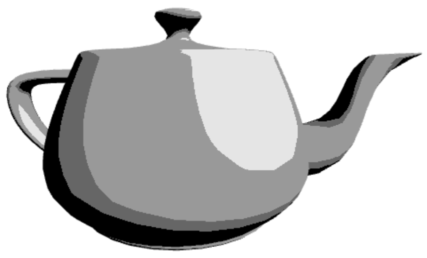
To try to fix this we are going to move, gradually, the computations to the fragment shader. We start by moving the color computation to the fragment shader, i.e. the vertex shader will output the intensity instead of the color. The new vertex shader is as follows:

```
#version 150

in vec4 position;
in vec3 normal;

out float intensityV;

uniform mat4 pvm;
uniform mat3 normalMat;
uniform vec3 lightDir;

void main() {
      // transform and normalise both vectors
      vec3 n = normalize(normalMat * normal);

      // compute the intensity using the dot operation
      intensityV = dot(n, lightDir);

      gl_Position = pvm * position;
}
```

The fragment shader receives the interpolated intensities for each fragment and computes the color:

```
#version 150

in float intensityV;

out vec4 outputF;

void main() {
      vec4 color;
      // compute the color based on the intensity
      if (intensityV > 0.9)
            color = vec4(0.9);
      else if (intensityV > 0.5)
            color = vec4(0.6);
      else if (intensityV > 0.3)
            color = vec4(0.4);
      else
            color = vec4(0.0);

      outputF = color;
}
```

And the result is as shown in the next figure. As can be seen it is far better. In this latter example we are interpolating the intensity instead of the discretized color.



What would happen if we did move the intensity computation to the fragment shader as well? That's what our next version of the toon shader proposes. The vertex shader is as follows:

```
#version 150

in vec4 position;
in vec3 normal;

out vec3 normalV;
```

```
uniform mat4 pvm;
uniform mat3 normalMat;

void main() {
	// transform and normalise normal
	normalV = normalize(normalMat * normal);

	gl_Position = pvm * position;
}
```

As can be seen the vertex shader is getting simpler. The fragment shader on the other hand gets a little bit more code:

```
#version 150

in vec3 normalV;

out vec4 outputF;

uniform mat3 normalViewMat;
uniform vec3 lightDir;

void main() {

	float intensity = dot(normalV,lightDir);

	vec4 color;
	// compute the color based on the intensity
	if (intensity > 0.9)
		color = vec4(0.9);
	else if (intensity > 0.5)
		color = vec4(0.6);
	else if (intensity > 0.3)
		color = vec4(0.4);
	else
		color = vec4(0.0);

	outputF = color;
}
```

The result from this pair of shaders is the same as before! This is because this pair of shaders performs exactly the same computations as the previous pair.

However, when we look carefully at this shader we discover that there is something missing. The intensity is not being properly computed, since there is no guarantee that the interpolated normal vector is unit length. In fact, in the general case it won't be unit length.

The only particular case where it will be unit length is when all normals for a face are identical. In the general case, where each vertex has a different normal, and considering points inside the face, the normal will have a magnitude smaller than one.

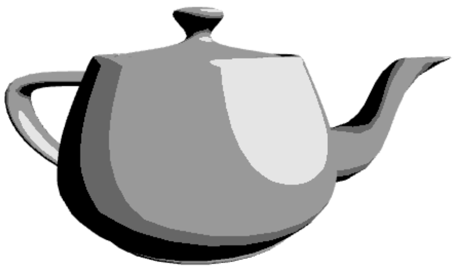
So what we need to do is to normalize the normal before we compute the intensity. Instead of writing

```
    float intensity = dot(normalV,lightDir);
```

we should write

```
    float intensity = dot(normalize(normalV),lightDir);
```

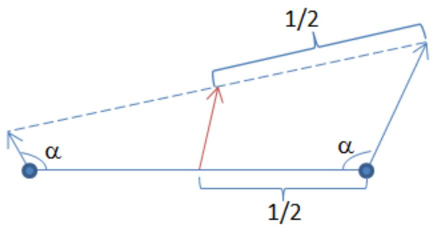
And the result we get is as follows. As can be seen, the curvature of the highlights is smoother in this image. This is the Phong Lighting model, where lighting is computed per fragment, with interpolated normals.



Normalisation Issues

As presented in the last pair of shaders, the normal vector is normalised twice: once in the vertex shader, and again the interpolated vector is normalised in the fragment shader. The question is do we really need to perform this normalisation twice? Are there any situations where we can get away with just one or even no normalisations at all?

First lets consider the normalisation in the vertex shader. The output vectors from the vertex shader are going to be interpolated. So what happens if they are not normalised?

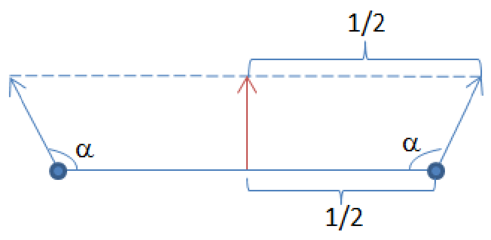


In the above figure, the interpolated vector should have a vertical direction. As can be seen, the largest vector has more influence on the direction of the interpolated vector. Normalising is required for the interpolated vector to have the correct direction.

So when can we avoid the normalisation in the vertex shader? When the transformed normal is unit length.

We can guarantee that the normal attribute, the input of the vertex shader, is unit length. So the question now becomes: when does the transformation with the normal matrix preserves the length of the original normal vector? The answer is: when the normal matrix is orthonormal, or to put it in more simpler terms, when all the operations we perform in the modeling and viewing matrices are translations and rotations, i.e. no scales.

Next we focus on the fragment shader. What are the properties of the interpolated normal? We have already seen that as long as the normals per vertex are unit length, the direction of the interpolated normal is correct. But what about its magnitude?



As can be seen from the figure, the magnitude of the interpolated vector is smaller than the original vectors, both of which are unit length. The only situation where the interpolated normal will be unit length is when all normals have the same direction.

Note that, if we had decided to transform the light direction in the vertex shader and pass the interpolated vector to the fragment shader, the same reasoning does not apply. This is because the light direction is constant for all vertices, hence the interpolation would always provide the same vector.