Elmar Eisemann

Shaders in OpenGL

Practical Exercise

The modern graphics pipeline

Recent versions of OpenGL offer a lot more flexibility to the programmer to control rendering and effects like lighting themselves. They no longer include built-in commands like `glLight` and `glTranslate`, instead you have to implement the matrix math and lighting calculations yourself. This adds a little bit of extra work for you, but it allows you to implement much more interesting effects.

Modern OpenGL requires you to write *shaders*, which are small programs that run on the graphics card. The first type of shader is a *vertex shader*. This is a program that takes the 3D coordinates of your model as input and transforms them into a 2D position on the screen as output. One of the simplest vertex shaders looks like this:

```
#version 430

layout(location = 0) uniform mat4 mvp;

layout(location = 0) in vec3 pos;

void main() {
    gl_Position = mvp * vec4(pos, 1.0);
}
```

As you can see, shaders are written in a language that looks like C++. The vertex shader program will be executed for every point in every triangle of the things you render. This sample code takes the 3D position of the point and multiplies it with the combined model/view/projection matrix to transform it into a position on the screen. The `gl_Position` variable is a special output variable in OpenGL shaders that the program has to write its screen position to. To calculate this position, you can use several types of variables in your shader. Data that is the same across all of your points, like the perspective matrix, should be passed as a global variable, known as a `uniform` in OpenGL shaders. You can set these values from your C++ program. Because the vertex shader runs for every point in your model i.e. every vertex, it also receives per-vertex data like the position. These per-vertex values are passed as `in` variables.

After points in your model have been transformed into screen coordinates by the vertex shader, the graphics card will combine them into points, lines or triangles. This process is known as *rasterization* and determines which pixels on the screen every triangle, for example, will cover.

To determine the color of each pixel in a triangle, a second type of shader is run called the *fragment shader*. This is a shader program that runs for every pixel inside shapes that are drawn and should output the final color that you see on the screen. A simple fragment shader looks like the following code.

```
#version 430

layout(location = 0) out vec4 color;

void main() {
    color = vec4(1.0, 0.0, 0.0, 1.0);
}
```

This fragment shader will color every pixel of the shapes you draw red. Just like the vertex shader, you have to write your final result to a variable. In the case of fragment shaders, that is a special `out` variable. The reason that this is a variable that you have to define yourself instead of there just be a `gl_Color` is that it is possible to output multiple colors from a fragment shader, but this is advanced functionality.

So, to summarize what we've learned so far:

1. You specify per-point data for your model, like positions, normals and texture coordinates.
2. The vertex shader is run for each point with these *attributes* as input and it will output an on-screen position for all of them.
3. The graphics card takes the on-screen points and combines them into shapes. For triangles, for example, it will create groups of 3. It will determine which pixels on the screen are covered by that triangle and then it will run the fragment shader for each of them to determine the color to put on the screen.

Specifying the points of your model, known as *vertices*, is also different from older versions of OpenGL. You can no longer use commands like `glBegin` and `glVertex`. You have to allocate memory on the graphics card and store your vertex data in it. Memory on the graphics card that holds your model's vertex data is known as a `Vertex Buffer Object` in OpenGL. For comparison, let's say we want to draw the following triangle in modern OpenGL:

```
glBegin(GL_TRIANGLES);
    glColor3f(1.0f, 0.0f, 0.0f);
    glVertex3f(0.0f, 0.0f, 0.0f);

    glColor3f(0.0f, 1.0f, 0.0f);
    glVertex3f(1.0f, 0.0f, 0.0f);

    glColor3f(0.0f, 0.0f, 1.0f);
    glVertex3f(1.0f, 1.0f, 1.0f);
glEnd();
```

For modern OpenGL, you would first define a C++ type to hold your vertex data:

```
struct vertex {
    vec3 color;
    vec3 pos;
};
```

The `glm::vec3` type is a vector with 3 float coordinates. Next, you need to build an array of your vertex data:

```
std::vector vertices = {
    Vertex { vec3(1.0f, 0.0f, 0.0f), vec3(0.0f, 0.0f, 0.0f) },
    Vertex { vec3(0.0f, 1.0f, 0.0f), vec3(1.0f, 0.0f, 0.0f) },
    Vertex { vec3(0.0f, 0.0f, 1.0f), vec3(1.0f, 1.0f, 0.0f) },
};
```

You can then create a Vertex Buffer Object (VBO) and put your vertex data in graphics card memory.

```
GLuint vbo;
glCreateBuffers(1, &vbo);
glNamedBufferStorage(
    vbo,
    vertices.size() * sizeof(Vertex),
    vertices.data(),
    0
);
```

Most triangles share their points (vertices) with neighboring triangles. Creating a duplicate vertex for each triangle will increases (GPU) memory usage. So instead we use an index buffer. The index buffer stores three integers for each triangle, which act as indices into the vertex buffer. We create an Element Buffer Object to store these indices.

```
GLuint ibo;
glCreateBuffers(1, &ibo);
glNamedBufferStorage(
        ibo,
        indices.size() * sizeof(unsinged int),
        indices.data(),
        0
);
```

We now need to write a vertex shader that takes per-vertex colors and positions as input. Such a program would look like this:

```
#version 430

layout(location = 0) uniform mat4 mvp;

layout(location = 0) in vec3 pos;
layout(location = 1) in vec3 color;

out vec3 fragColor;

void main() {
    gl_Position = mvp * vec4(pos, 1.0);
    fragColor = color;
}
```

Note that we've added a new `in` variable to also take the colors as input. However, the vertex shader cannot assign colors to triangles by itself, because it only exists to output screen coordinates, not colored pixels. So we have to pass the color through to the fragment shader to actually color the triangles. That's what the `out` variable is for. This vertex shader will not just output the on-screen position of each vertex, but also a color per-vertex. These color values are then interpolated across the area of the triangle and the interpolated values are fed to the fragment shader. The fragment shader can use the color from the vertex shader like this:

```
#version 430

layout(location = 0) out vec4 color;

in vec3 fragColor;

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

There's still one piece missing from the puzzle. With the old commands, like `glColor3f`, OpenGL would know that the vector you specify is color data and `glVertex3f` would tell OpenGL that the vector is a 3D position. However, the vertex shaders in modern OpenGL can take any type of data and your vertex structs can have any type of layout and variable names. That means that the array of vertices just looks like a heap of bytes to OpenGL right now and it has no idea how to pass them to the vertex shader inputs.

To solve that, you have to define a mapping between the values in your vertex array and the inputs of your shader. This mapping does not use the names of the vertex shader `in` variables directly, but rather their indices. These indices are specified in the vertex shaders above using the `layout(location = some_index)` syntax. For each of these inputs, you have to specify:

- Which Vertex Buffer Object to load the vertices from
- At which byte offset in the buffer the data starts
- How much space in bytes there is between the data of two vertices

This is quite a lot of data, and you wouldn't want to specify this again every time you want to draw something. That's why OpenGL has a special *Vertex Array Object* that can store these mappings for you. Don't mind the misleading name, it stores mappings and not vertex arrays. To link the aforementioned vertex array with positions and colors to the vertex shader, you would use the following code:

```
GLuint vao;
glCreateVertexArrays(1, &vao);

// The VAO mapping also stores which Element Buffer Object to
    use to index into the vertex array.
glVertexArrayElementBuffer(vao, ibo);
```

```
glVertexArrayVertexBuffer(
        vao, // Vertex Array Object
        0, // Vertex shader input index
        vbo, // Vertex Buffer Object
        offset(Vertex, pos), // Offset of the position in the
            Vertex struct
        sizeof(Vertex) // Space between the vertices
);
glVertexArrayVertexBuffer(
        vao, // Vertex Array Object
        1, // Vertex shader input index
        vbo, // Vertex Buffer Object
        offset(Vertex, normal), // Offset of the normal in the
            Vertex struct
        sizeof(Vertex) // Space between the vertices
);
glEnableVertexArrayAttrib(vao, 0);
glEnableVertexArrayAttrib(vao, 1);
```

We first tell the Vertex Array Object which element buffer to use. Secondly, we specify the Vertex Buffer Object, the index of the shader input and how it is laid out in memory. Shader inputs are disabled by default, so they need to be enabled using `glEnableVertexArrayAttrib` by their index.

You now know how per-vertex data is provided to shaders to draw geometry on the screen. However, you also need to provide some data for your global (`uniform`) shader variables. This is also done by index:

```
const glm::mat4 view = trackball.worldToCameraMatrix();
const glm::mat4 proj = glm::perspective(fieldOfView,
    aspectRatio, clipNear, clipFar);
const glm::mat4 model = glm::mat4(1.0f); // Identity matrix

const glm::mat4 mvp = proj * view * model;

glUniformMatrix4fv(
    0, // Shader uniform index
    1, // Matrix count
    GL_FALSE, // Transpose of matrix?
    glm::value_ptr(mvp) // Pointer to 4x4 floats
);
```

Shader uniforms can be set directly with functions like `glUniformMatrix4fv`, you don't have to explicitly put them in graphics card memory like vertex data. In this code snippet we use the GLM library to set up a view, projection and model transformation. These matrices are multiplied to create a combined model/view/projection matrix that is then uploaded to the shader uniform variable called `mvp` with `location = 0` (see the vertex shader code above).

Reminder: Illumination models

In the previous introduction we assumed that we outpout the colors per vertex in the vertex shader and interpolate those values in the fragment shader. However, we will get higher quality results if we compute the illumination per pixel in the fragment shader.

For the following exercises you will be expected to write fragment shaders which compute the color of a pixel in a triangle. Please have a look at the provided vertex shader and debug fragment shader which are located in the shaders folder. This fragment shader takes the world space normal that is passed by the vertex shader and reinterprets it as a RGB value.

To place the light at the current camera position press l. To add a second light, use L. For this exercise, we will always assume to have white light sources.

Lambertian Model

First, we will define a simple diffuse surface. The main part of the formula is $Kddot(N, L)$, where L is the direction of the point to the light and N its normal. Use the Kd value that is stored in `shadingData`.

Also compare: `http://en.wikipedia.org/wiki/Lambertian_reflectance`

Implement a Lambertian Model in the main function of *shaders/lambert_frag.glsl*.
Use uniforms to pass variables to your fragment shader. Modify the main rendering loop where it says `// === SET YOUR LAMBERT UNIFORMS HERE ===` to pass the CPU values to the uniforms that you've defined in your shader. You can set the uniform values using `glUniform`:

```
glUniform1i(uniformIndex, intValue);
glUniform1f(uniformIndex, floatValue);
glUniform3fv(uniformIndex, 1, glm::value_ptr(glmVector));
```

Here the uniform index should match the `layout(location = ...)` that you set in your shader. **Do not use uniform index 0!** That slot is already used by the vertex shader to pass the model/view/projection matrix.

Phong Model

Implement the Phong model in *shaders/phong_frag.glsl* which introduces view-dependent effects (specularities).

Its main formula looks like this : $k_s dot(R, V)^s$, where R is the reflection vector (the reflected vector from the light at the plane defined by the surface's normal) and V is the vector from the surface point towards the camera position. You can make use of the build-in glsl function `reflect`. Be aware that it assumes that the light vector is pointing towards the surface. Use the k_s value that is stored in `shadingData`. The camera position is stored in the `cameraPos` variable. Modify *main.cpp* to pass the required uniform variables (the comments will indicate where to put your code).

Blinn-Phong Model

Implement the Blinn-Phong model which also takes view-dependent effects (specularities) into account in *shaders/blinn_phong_frag.glsl.* Its main formula looks like this: $dot(H, N)^s$, where H is the unit vector exactly between the view direction V and the direction towards the light. s is the specular exponent (in the code called shininess) and influences the size of the highlight.

More can be found here.

What impact does s have?

If, for an even exponent, a light behind the model illuminates it, you might have to add a fix!

Diffuse Toon shading

Toon shading give the impression of looking at a comic drawing. In our case, all you need to do is quantize the diffuse (Lambert) illumination. This means that there are `shadingData.toonDiscretize` uniform intervals between 0 and k_d. Figure 1 illustrates a possible result.

We can also apply toon shading to specular highlights. Compute the Blinn-Phong specular highlights, if it has a value larger than or equal to `shadingData.toonSpecularThreshold` then make the pixel white, otherwise discard it or return black.

Implement the diffuse toon-shading shader in *shaders/toon_diffuse_frag.glsl* and the specular toon shader in *shaders/toon_specular_frag.glsl.*

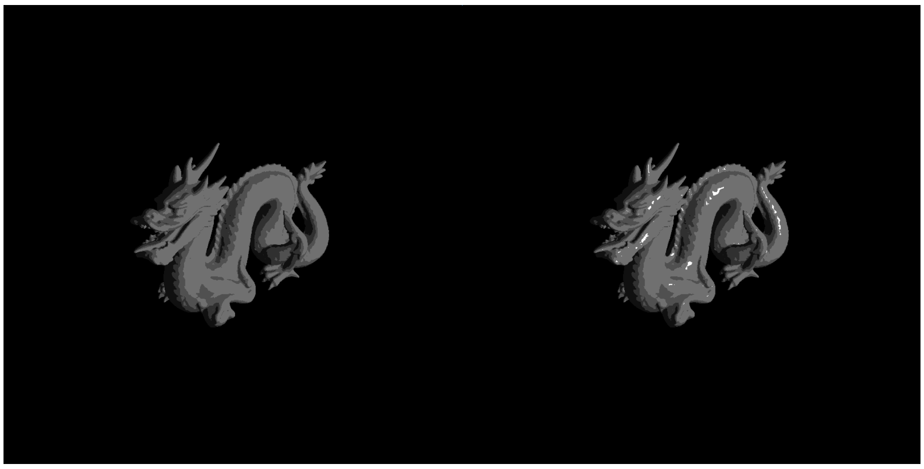
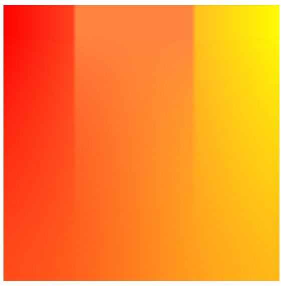


Figure 1: Toon shading (left without, right with specularities)

X-Toon shading

The downside of toon shading is that it is hard to control the effect as an artist. For example, it is not possible to easily color shadows differently from lit areas. X-Toon shading is an extended version of toon shading that tackles this problem by using a texture to lookup the lighting instead of doing math inside the shader itself:



Instead of doing math on the brightness value in the shader, it is used to do a color lookup in this texture. You'll already recognize three distinct shades in the textures, similar to our previous toon shader.

As you may have already noticed, textures in OpenGL are created similarly to Vertex Buffer Objects. You create a Texture Object using `glCreateTextures`, allocate it using `glTextureStorage2D` and then upload data to it with `glTextureSubImage2D`. In OpenGL you cannot just sample an arbitrary texture in your shader, you will have to bind it to a certain slot like `GL_TEXTURE0` or `GL_TEXTURE7`. This is done by first making a slot active with `glActiveTexture` and then binding a texture with `glBindTexture`. You can then access this texture in a shader by using a special `uniform` variable:

```
layout(location = 2) uniform sampler2D texToon;
```

The `sampler2D` type specifies a sampler object through which you can access a texture. This sampler is assigned a specific texture slot index with the `glUniform1i` call. The behavior of texture sampling is set with the `glTextureParameteri` call. For example, you can configure:

- What should happen when accessing the texture beyond the $[0, 1]$ range?
- How should a color between pixels be calculated?
- Should the texture have a border - and if so - what color?

Sampling a texture beyond the $[0, 1]$ range is useful if you want to repeat a texture, for example, to fill a landscape with grass. Interpolating values between pixels will make a texture look smoother, although this may not be the intended effect if you want to go for an art style like Minecraft's, for example.

The program you were given already loads the sample X-Toon map you saw before into a texture `texToon`. Note that this is a 2D image, but we will only use the first row for now. You can access a texture in the shader using the `texture` function like so:

```
outColor = texture(texToon, vec2(0.0, 0.0));
```

The `texture` function takes the sampler variable and a texture coordinate as parameters. Texture coordinates are almost always in the $[0, 1]$ range instead of pixels coordinates like $[0, 512)$. The code above will sample the red color in the left corner of the texture, whereas the following code will sample the yellow right corner:

```
outColor = texture(texToon, vec2(1.0, 0.0));
```

Now try using the brightness of the original Lambert + Blinn-Phong shading without toon effects to sample the texture. You should only vary the horizontal coordinate and keep the vertical coordinate `0.0`. That should result in something like this: X-Toon shading uses the vertical axis in the texture to add

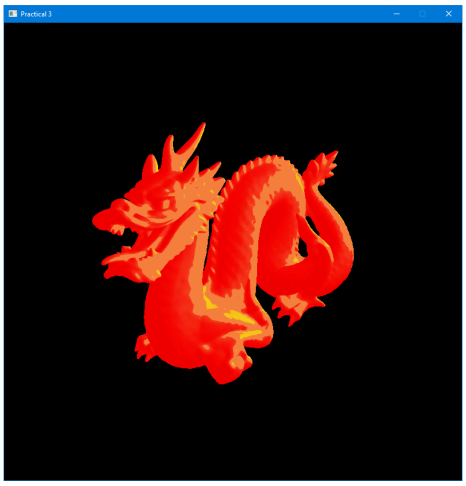


Figure 2: Basic X-Toon shading.

details based on the distance between the viewer and the model. First try using the world-space distance between the viewer and the fragment to vary the brightness like below. You may need to offset the distance and scale it to make it look this way, otherwise most of the distance values are outside the $[0, 1]$ color range.

Now use this distance for the vertical coordinate in the texture lookup and you will get different styles for closer fragment and farther fragments like in figure 3.

Notice that the head of the dragon, which is close to the viewer, has very discrete shades of color whereas the farther areas like the tail have smoother shading. This is now directly controlled by the X-Toon texture map. Try using Paint to edit the texture map to create different styles. Below are some examples you can use. Also try varying the light position again to see what your effect looks like from every angle!

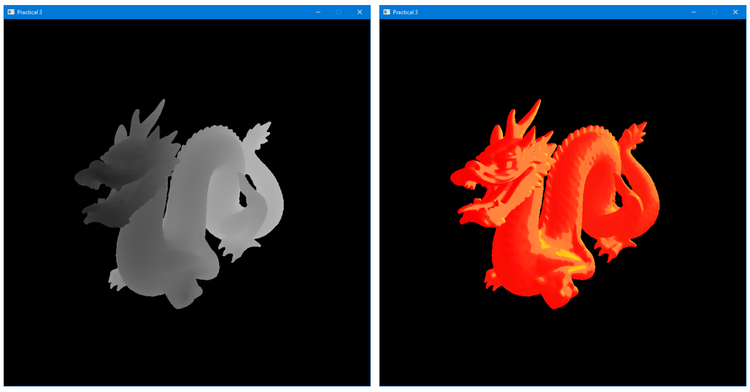


Figure 3: Left: distance from the camera. Right: full X-Toon shading.

And there was light!

Now, we will focus on the control of the illumination. For an artist, it can be very difficult to control the light positions (just look at the credits of any modern movie, you have hundreds of people working on the light placements!). Our goal is to simplify this work and provide the artist with a tool to efficiently place lights in an indirect manner.

We will use the function *userInteraction* in the following. This function receives, where the user has "clicked" (as the mouse button is used for the navigation, pressing *space* will launch this function instead of a click). This interaction will trigger the function *userInteraction....* We will use these functions to define a new light position, based on different criteria. To switch the interaction mode (hence, the userInteraction function that is called), use the "M" key. Please also read the indications in the source code.

Placing the light with the mouse

The function userInteractionSphere will be used to: Place the light on a sphere of radius 1.5, centered at (0,0,0). Pay attention to choosing the intersection point closest to the observer - which can even be behind the observer!

Placing the light according to shading

This time, the new light position should be chosen such that the light produces a exactly a lambertian shading of zero at the clicked location. (Use the mode Toon shading to verify your solution). There are several possibilities to achieve this goal! The implicit condition is that $dot(L, N)$ should be zero, which leaves some degrees of freedom. **Come up with a solution that you find appropriate and test if the light behaves "intuitively".**

Placing the light based on specularities

Placing a specularity is not easy because its position depends on the view AND the light. **Find a solution to make sure that the specularity is centered at the clicked location.**

Multiple light sources

Use the combination *shift+L* to add lights in the scene. To restart from scratch with all your light sources, press *shift+N*.

Controlling the parameters

We would like to add colored sources, which will be possible by by modifying `lights[xxx].color`. It will contain the RGB triplet for each light source, which represents its color.

Complete the function *keyboard* and add the keys for R, G, B, r, g, b. Capital letters should increment, small letters reduce the corresponding values the color by a small constant, e.g., 0.1. The affected light should be the current, which is stored in a global variable `SelectedLight` and can be changed by the + and - keys. The selected light can be determined on the screen by its border color.

Change your fragment shaders to take the light color into account.