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. One of which is X-Toon shading, which you'll implement in this exercise. This shading allows you to do interesting toon shading effects based on depth and Blinn-Phong lighting, as seen in the image below.

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<vertex> 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;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(
    GL_ARRAY_BUFFER,
    vertices.size() * sizeof(vertex),
    vertices.data(),
    GL_STATIC_DRAW
);
```

Note that many functions in OpenGL don't directly take the value they operate on as argument. Instead of having `glBufferData(vbo, ...)`, you often have to call a function to bind an object as active object and then any subsequent function calls will do something with that object until you bind another one.

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 data from
- At which byte offset in the buffer the data starts
- How much space in bytes there is between the data of two vertices
- What data type the data has, e.g. float
- How many components there are, e.g. 3 for a 3D vector or 1 for a single float

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;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

glBindBuffer(GL_ARRAY_BUFFER, vbo);
glVertexAttribPointer(
    0, // Vertex shader input index
    3, // Number of components (vec3)
    GL_FLOAT, // Type of data
    GL_FALSE, // Not normalized
    sizeof(Vertex), // Space between vertices
    // Offset of position in Vertex struct
    reinterpret_cast<void*>(offsetof(Vertex, pos))
);
glEnableVertexAttribArray(0);

glBindBuffer(GL_ARRAY_BUFFER, vbo);
glVertexAttribPointer(
    1, // Vertex shader input index
    3, // Number of components (vec3)
    GL_FLOAT, // Type of data
    GL_FALSE, // Not normalized
    sizeof(Vertex), // Space between vertices
    // Offset of position in Vertex struct
    reinterpret_cast<void*>(offsetof(Vertex, color))
);
glEnableVertexAttribArray(1);
```

We first specify the Vertex Buffer Object to take the data from, in this case it's the same for both *vertex attributes* (position and color). Secondly, we specify the index of the shader input, the type of data and how it is laid out in memory. These mappings of vertex buffer data to shader input indices is stored in the Vertex Array Object. Remember that OpenGL functions often have implicit arguments due to global state, so in this case `glVertexAttribPointer` has the current VBO (Vertex Buffer Object) and VAO (Vertex Array Object) as implicit arguments that it works with, simply because they are the currently bound objects. Shader inputs are disabled by default, so they need to be enabled using `glEnableVertexAttribArray` 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:

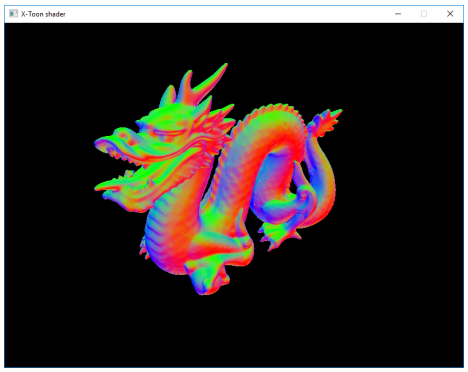
```
glm::mat4 view = glm::lookAt(viewPos, targetPos, upVector);
glm::mat4 proj = glm::perspective(fieldOfView, aspectRatio,
    clipNear, clipFar);
glm::mat4 model = glm::mat4(); // Identity matrix
```

```
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 just like you would have done in old OpenGL with functions like `gluLookAt`. 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).

You should now take a look at the code in *main.cpp*, *shader.vert* and *shader.frag* to see how all of this code comes together to render a model of a dragon on the screen. Open the project in the *Visual Studio* folder to run the code, after which you should see this:



Use the comments in the code and the information above to understand what is happening. Some things you can try are listed below. If the program immediately exits when it starts up, then you probably have an error in your shader. Run your program with *Ctrl + F5* to keep the command prompt open to see the error.

- Use the position as color in the fragment shader instead of normal
- Use the `time` variable and a function like `sin` to animate the color of the dragon through time
- Copy the `time` uniform declaration from the fragment shader to the vertex shader and use it to animate the movement of the dragon, for example with `gl_Position = mvp * vec4(pos + vec3(0.0, 0.1, 0.0) * sin(time), 1.0);`. If you proceed with this, then make sure that `fragPos` is also updated with the new positions.

You can also try adding your own field to `Vertex` and use it as input for the vertex shader with the following steps:

- Add field to `Vertex`
- Set the value(s) of the field in the OBJ loading code where `vertex.pos` and `vertex.normal` are also set
- Add a `glVertexAttribPointer` and `glEnableVertexAttribArray` call for index `2` with the right type, size and offset in the vertex buffer
- Add an input in the vertex shader with the right index and type

You can find a nice reference of functions that are available in the shader code here: `http://www.shaderific.com/glsl-functions/`.

Exercise 1: Per-pixel Blinn-Phong

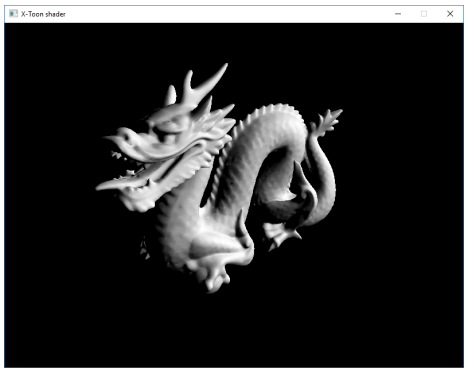
You should already be familiar with Blinn-Phong shading from the previous exercises, however we will now implement this shading per-pixel on the graphics card rather than per-vertex on the CPU. This will allow for quick real-time lighting updates and higher quality results, because each pixel on-screen will be shaded independently. To implement this, we will only be working in the fragment shader.

It already has almost all of the data that you need available: `viewPos`, `fragPos` and `fragNormal`. The only thing that is missing is the light position. We will define a static lighting position for now:

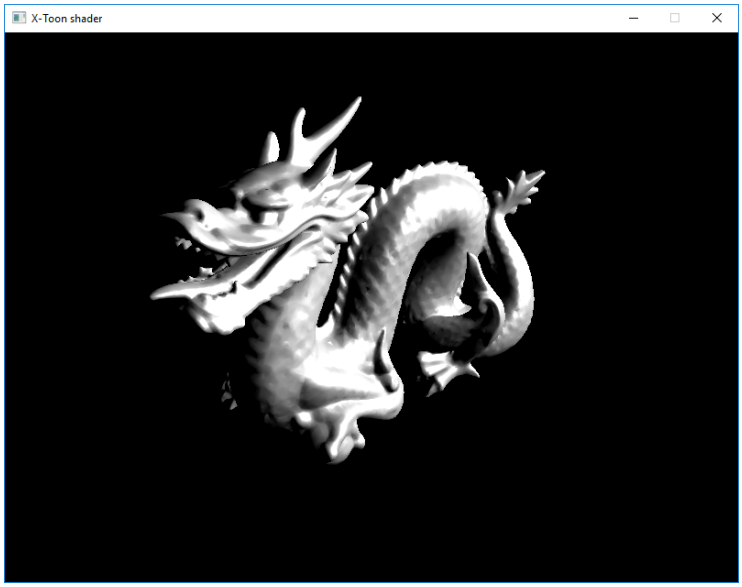
```
const vec3 lightPos = vec3(-1.0, 1.0, 1.0);
```

You can put this line of code inside or outside the `main` function, but I prefer to put it outside because it's bound to be replaced by a `uniform` at some point.

You should now be able to write the code to shade the dragon with diffuse lighting using the formula that you used in the previous OpenGL exercises. You can use the `dot` function for dot products and the `clamp` function to keep a value in the $[0, 1]$ range. The diffuse lighting should look like this:



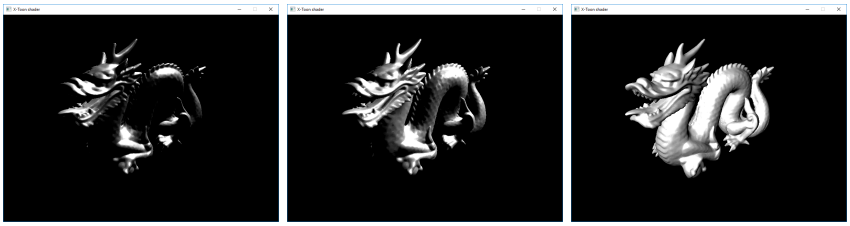
After this, add specular lighting and combine it with the diffuse lighting to complete Blinn-Phong shading. It is good to first define values like `viewDir` and `lightDir`. You should use a specular exponent around `16`. Your end result should now have diffuse lighting along with specular highlights:



To make sure that the diffuse lighting and highlights are correct, you should make the light source move by changing this line:

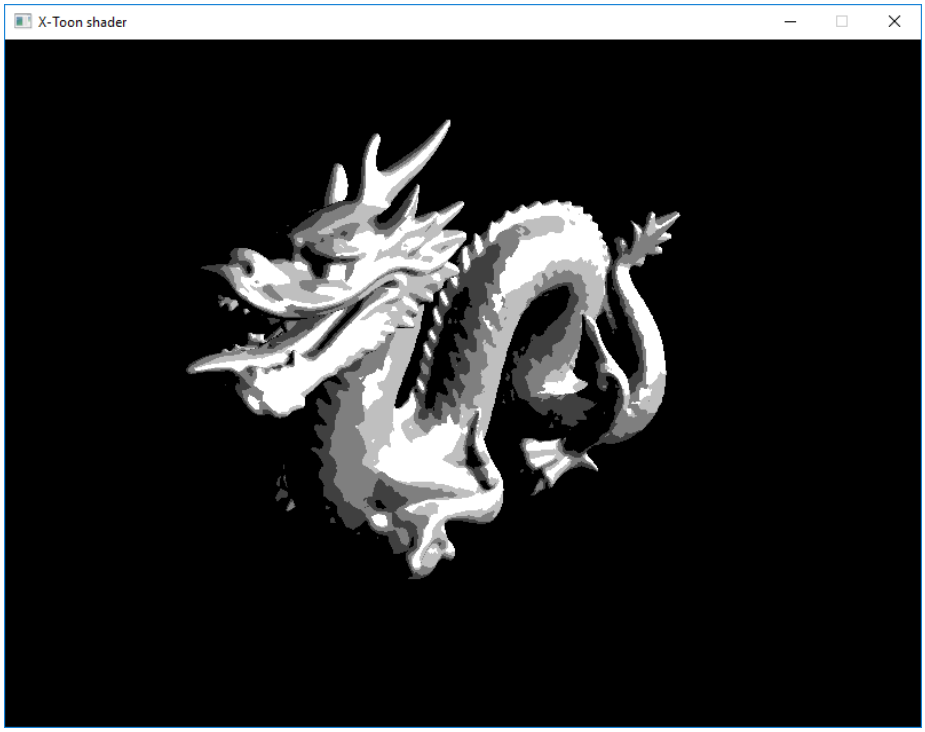
```
const vec3 lightPos = vec3(sin(time), 1.0, cos(time));
```

You should then see the lighting change:



Exercise 2: Toon shading

Toon shading or cel shading is a shading technique that changes lighting to look cartoony. Your Blinn-Phong implementation results in nice gradients from dark to bright regions, but to get a flat cartoony look, you need to turn those gradients into discrete bands of color. You can do this by mapping your lighting from the entire $[0, 1]$ range to a few discrete values, like $0.0, 0.25, 0.50, 0.75, 1.0$. You should also threshold your specular highlights, so that they have value 1.0 above a value like 0.2 and otherwise they're 0.0. Try to extend your fragment shader to perform this discretization. With 5 levels, you should get something like this:



Instead of smooth gradients of lighting, you will now have flat looking color bands that stylize the image. This will look especially nice if you have the light rotate like before.

Exercise 3: 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 `glGenTextures`, bind it and then upload data to it with `glTexImage2D`. 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, similarly to how you assign a matrix value to a `mat4`. You can see this in the render loop of `main.cpp`. The behavior of texture sampling is set with the `glTexParameteri` 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.

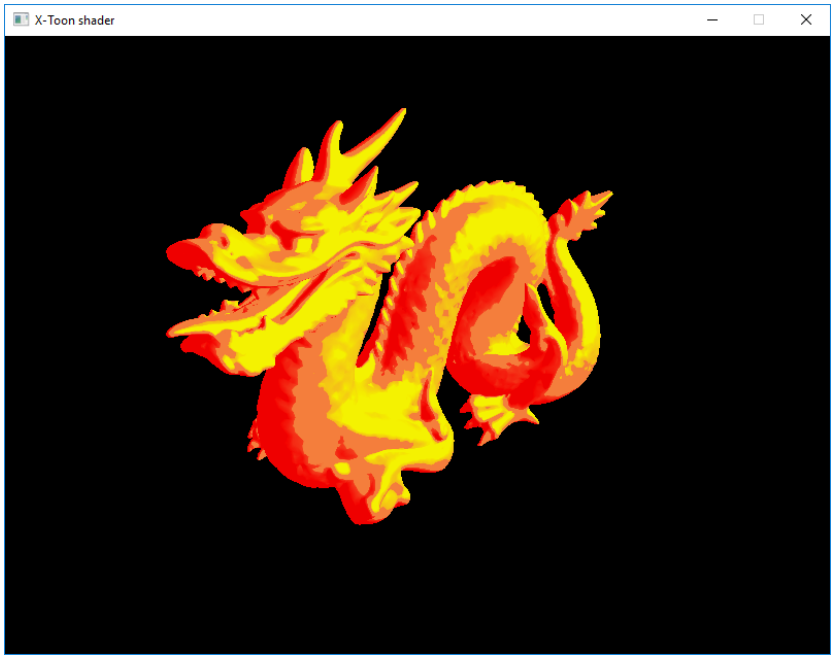
The program you were given already loads the sample X-Toon map you saw before into a texture that you can access from the fragment shader using the `texToon` variable. Note that this is a 2D image, but we will only use the first row for now. You can access a texture in the shader using the `texture`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

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

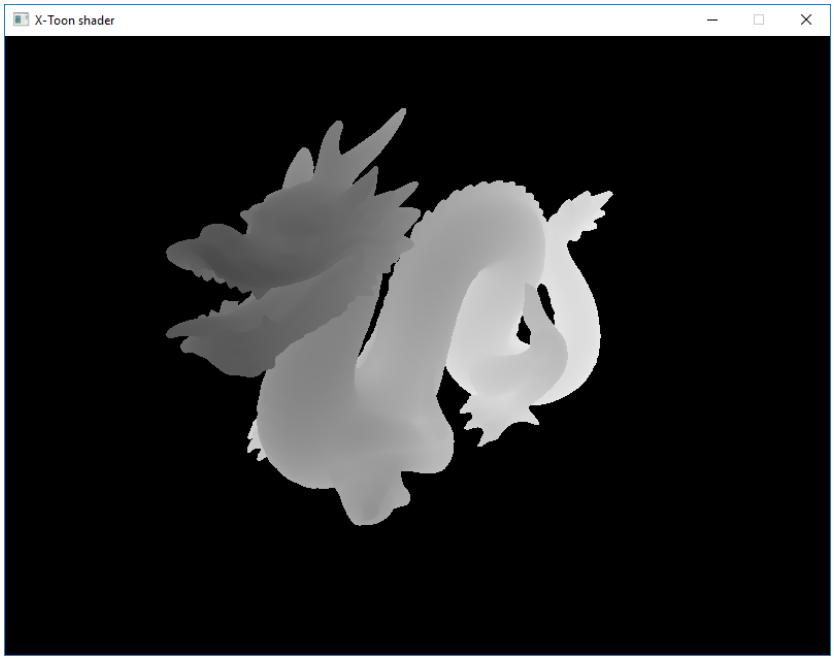
The `texture` function takes the sampler variable and a texture coordinate as parameters. Texture coordinates are almost always in the $[0, 1]$ range instead of pixels coordinates like $[0, 512)$. The code above will sample the red color in the left corner of the texture, whereas the following code will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

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

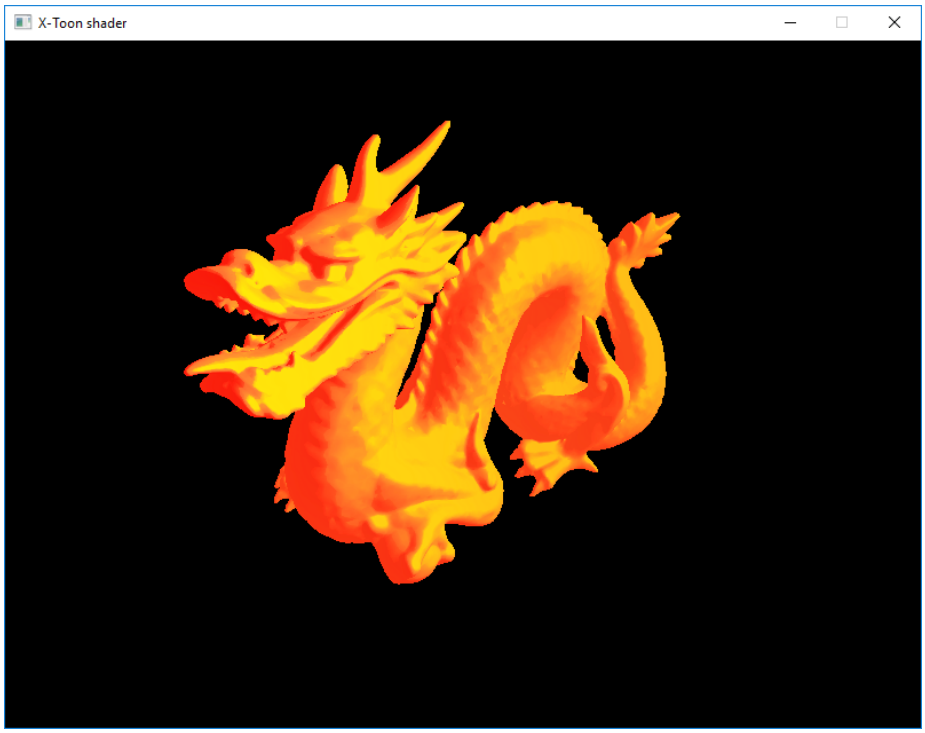
Now try using the brightness of the original Blinn-Phong shading without toon effects to sample the texture. You should only vary the horizontal coordinate and keep the vertical coordinate `0.0`. That should result in something like this:



X-Toon shading uses the vertical axis in the texture to add 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 this:



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! By doing these exercises, you've seen how

flexible shader code in modern OpenGL can be. Rendering lighting effects like this in real-time would have been infeasible with the older API, and this is just the beginning! You can also try deforming the dragon in the vertex shader like this:

```
vec3 finalPos = pos + vec3(0.0, sin(pos.y * 10.0 + time * 2.0)
    * 0.007, 0.0);
gl_Position = mvp * vec4(finalPos, 1.0);
fragPos = finalPos;
```

Have fun!