IM420 Advanced Computer Graphics – WS 2020
Assignment 3 - Extended Toon Shader (Xtoon Shader)

Sahabaj Barbhuiya, Faezeh Asgharkermani

25th November, 2020

1.1 Task

This assignment report and source codes aims to explain the implementation of *Extended Toon (Xtoon)* shader. This assignment is based on the lectures and project template of Gooch shader *06b-GoochOBJModel* where different lighting models such as *Ambient, Diffuse,* and *Specular* have been used to implement the Gooch shader. Textures are loaded and also implemented on the model. Four different types of textures are created using Photoshop tool which have been implemented in this assignment which gives the Buddha model a very distinctive shading effect.

1.2 Introduction

Non-photorealistic rendering (NPR) is a field of computer graphics inspired by artistic styles such as painting, drawing, technical illustrations, and cartoons. NPR representation of images and models represents details in the changed areas such as shape, color, structure, shading and light. Toon shading (Cel-shading) is an examples of NPR styles which have been used in short films and video games. Example of toon shader implemented on a model and is shown in the Figure.1.1.

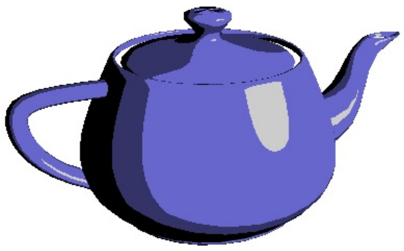

Figure 1.1: Example of a toon shader on a teapot model

1.3 Approach to the solution

1.3.1 Application Setup (main.cpp)

Some changes were made in the main.cpp file was the creation and initialization of uniforms for the fragment shader, such as cameraPos, and lightPos (*lightPos* alters the light position for the model object) in the main.cpp, these uniforms are defined as follows:

```
    vec3 cameraPos = vec3(0.0f, 0.0f, 0.75f);
    ...
    vec3 lightPos = vec3(lightX, lightY, lightZ);
```

Furthermore, to utilize the *Shader* and *Model* class, we have the following code:

```
Shader myShader("modelShader.vert", "modelShader.frag");
Model myModel("objects/buddha2/buddhaNoMaterial.obj");
```

For accessing the strings and functions from the *Shader* and *Model* class the following code has been implemented in the main.cpp:

```
myShader.setMat4("projection", projection);
myShader.setMat4("view", view);
myShader.setMat4("model", model);

myShader.setVec3("cameraPos", cameraPos);
myShader.setVec3("lightPos", lightPos);
myShader.setInt("shaderType", shaderType);
myShader.setFloat("weightTexture", weightTexture);

myModel.Draw(myShader);
```

1.3.2 Textures

To load the textures, we have defined an unsigned integer *"texture"* and integer type *"textureType"* which will later be used in main loop of our application file to switch between different textures.

```
    unsigned int texture;
    int textureType = 1;
```

"TextureFromFile" function can be used to load the textures, which is called from the ***model.h*** header file. At the starting phase of the assignment, we used sample textures provided in the lecture slides to test if the application file loads the texture. Later, we used Photoshop tool to create four different textures which will be implemented in our model. Nested if-statement can be used in the main loop to select different textures for our model.

```
    //New textures
    if (textureType == 1) {
      texture = TextureFromFile("linearGrad.png", "textures");
    }
    if (textureType == 2)
    {
      texture = TextureFromFile("radialGrad.png", "textures");
    }
    if (textureType == 3)
    {
      texture = TextureFromFile("angularGrad.png", "textures");
    }
    if (textureType == 4)
    {
      texture = TextureFromFile("linearGradBW.png", "textures");
    }
```

Moreover we use nested if-statements to switch between these textures using different keys. The following code snippet shows the keys used to switch between the textures.

```
    //switch between textures
  if (glfwGetKey(window, GLFW_KEY_1) == GLFW_PRESS)
    textureType = 1;
  if (glfwGetKey(window, GLFW_KEY_2) == GLFW_PRESS)
    textureType = 2;
  if (glfwGetKey(window, GLFW_KEY_3) == GLFW_PRESS)
    textureType = 3;
  if (glfwGetKey(window, GLFW_KEY_4) == GLFW_PRESS)
    textureType = 4;
```

1.4 Shader Programs

1.4.1 Vertex Shader

We define outgoing vectors *normal* and *fragPos* which will be used in the fragment shader to calculate and simulate the different lighting models for our Xtoon shader. We also define transformations matrices as uniforms (*model, view, projection*) which will be used to compute our *gl_position*. The full vertex shader is given below:

```
    #version 330 core
    layout (location = 0) in vec3 aPos;
    layout (location = 1) in vec3 aNormal;
    layout (location = 2) in vec2 aTexCoords;

    out vec2 texCoord;
    out vec3 normal;
    out vec3 fragPos;

    uniform mat4 model;
    uniform mat4 view;
    uniform mat4 projection;

    void main()
    {
    texCoord = aTexCoords;
    normal = mat3(transpose(inverse(model))) * aNormal;
    fragPos = vec3(model * vec4(aPos, 1.0));
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    }
```

1.4.2 Fragment Shader

For the implementation of the Xtoon shader, most of the work is done on the fragment shader. We define the incoming vectors coming from the vertex shader which will be later in the fragment shader, which are defined as follows:

```
in vec3 normal;
in vec2 texCoord;
in vec3 fragPos;
```

We define the uniform variables in our fragment shader with their respective values which is passed by the main.cpp. The uniform variables for our Xtoon fragment shader are given as:

```
uniform sampler2D texture_toon;
uniform vec3 cameraPos;
uniform vec3 lightPos;
```

Here, *sampler2D* is a uniform which represents a single texture of a particular texture type, and in our case we are using a 2D texture. *texture_toon* is the texture which will be applied to the model.

We also define light color (default light color as white) and the shading color. The shading color will be later used to hold the different lighting model values (combination of ambient, diffuse, and specular lighting).

```
vec3 lightColor = vec3(1.0, 1.0, 1.0);
vec3 shadingColor;
```

Let's discuss the different lighting models.

- **Ambient**: Ambient lighting is the basic lighting, and to simulate the ambient lighting , we use ambient lighting constant and multiply it with predefined light color.

  ```
  //Ambient
  float ambientFactor = 0.1; //ambient constant
  vec3 ambient = ambientFactor * lightColor; //multiply constant with the light
   color.
  ```

- **Diffuse**: Diffuse lighting gives a significant impact on the object. Diffuse lighting gives the model more brightness the closer the model is to the light source. To calculate the diffuse lighting we need to calculate the normal vector and the light direction.

  ```
  vec3 nnormal = normalize(normal); //normalize the incoming normal
  ...
  vec3 lightDir = normalize(lightPos - fragPos); //calculate light direction vector
  float diffFactor = max(dot(normal, lightDir), 0.0); //define a float value
  vec3 diffuse = diffFactor * lightColor;
  ```

  We use the max fucntion to remove the negative values(or to remove the darkness). We finally calculate the diffuse by multiplying diffFactor and lightColor.

- **Specular**: Specular lighting is based on the reflective properties of the object. Similar to the diffuse lighting . specular lighting is based on the light direction and normal vectors but it is also based on the view direction.

```
float specularStrength = 1.0; //define constant
vec3 viewDir = normalize(cameraPos - fragPos); //calculate view direction
vec3 reflectDir = normalize(reflect(nnormal, -lightDir));
float specFactor = pow(max(dot(viewDir, reflectDir), 0.0), 32);
vec3 specular = specularStrength * specFactor * lightColor; //calculating the
 specular component
```

So far, we have calculated the ambient, diffuse, and specular components. Finally, we can start implementing our Xtoon shader. We use previously defined shadingColor vector to store the ambient, diffuse, and specular component value which we will use in our Xtoon shading effect.

We define two float values f1 and f2.

```
float f1 = max(dot(nnormal, lightDir), 0.005);
float f2 = max(dot(nnormal, viewDir), 0.005);
```

Here, f1 is the diffuse light influence calculated in the diffuse component and f2 is the view angle influence calculated in the specular component shown in the Figure.1.2.

We also define a vector *xToonColor*, which will use the texture function (retrives texture elements from a texture), and in our case *toon_texture* is our texture to be implemented on the model. The final result is obtained by multiplying the shadingColor by lightColor and the *FragColor* is the previously calculated *xToonColor*.

```
vec4 xToonColor = texture(texture_toon, vec2(f1, f2));
vec3 result = shadingColor.rgb * vec3(1.0, 1.0, 1.0);
FragColor =  xToonColor;
```

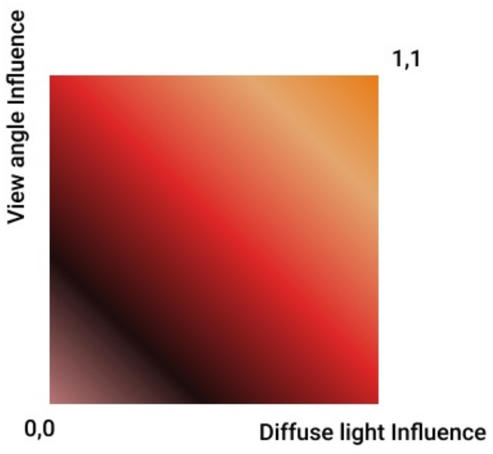

Figure 1.2: A new texture implemented based on Orientation-based attribute mapping

1.5 Summary

In this assignment a Xtoon shader was implemented using OpenGL and GLSL into a Buddha model object. First, we created some textures using the Adobe Photoshop tool. The ambient, diffuse and specular lighting models were also used to complete this assignment. The most challenging part of this assignment was to find a way to load the textures and implement the textures into our model. To overcome that problem we have used the *shader* and *model* header files.

We also implemented four different textures in our Buddha model object. Users are able to switch between the textures using keys **1, 2, 3,** and **4**. The textures used are shown in the Figure.1.3 and the results are shown in the Figure.1.4.

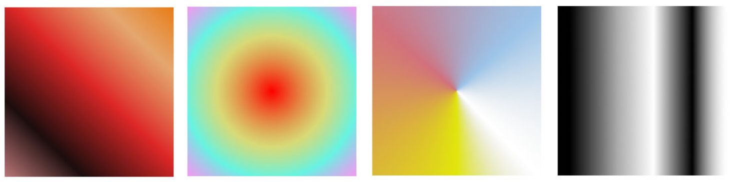

Figure 1.3: Textures used to implement the Xtoon shader

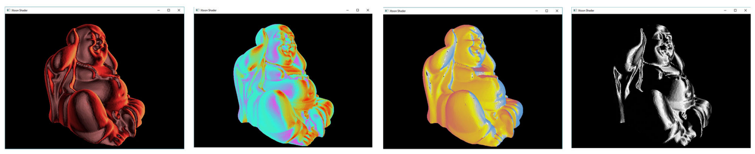

Figure 1.4: Outputs using the new textures