IM420 Advanced Computer Graphics – WS 2019/20
Final Project Report

Sahabaj Barbhuiya

6th May, 2020

1.1 THE PROJECT TASK

This project report and source codes aims to explain the implementation of *Gooch* shader and *Toon* shader. This project report consists of four essential components: code snippets from application file (main.cpp) which initializes the shaders, source code from the vertex shaders, source code from fragment shaders, and shader class. Furthermore, this project have been implemented in three phases. In the first phase, gooch shader is implemented on a model, in the second phase a toon shader is implemented, in the third phase a both shaders were implemented on a model object where user can switch between the two shaders. https://learnopengl.com/, Orange Book-OpenGL Shading Language 2nd Edition, and https://www.khronos.org/opengl/ have been used as reference to implement the project.

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. Since photorealistic representations are very detailed and complex to process quickly, NPR representations can be advantageous in the technical field as well as in the medical field. Toon shading (Cel-shading) and Gooch shading (developed by Bruce and Amy Gooch) also known as "cool-to-warm" shading are two examples of NPR styles which have been used in short films, video games and other software. Example of toon shader and gooch shader with model are shown in the Figure 1.1 and Figure 1.2.

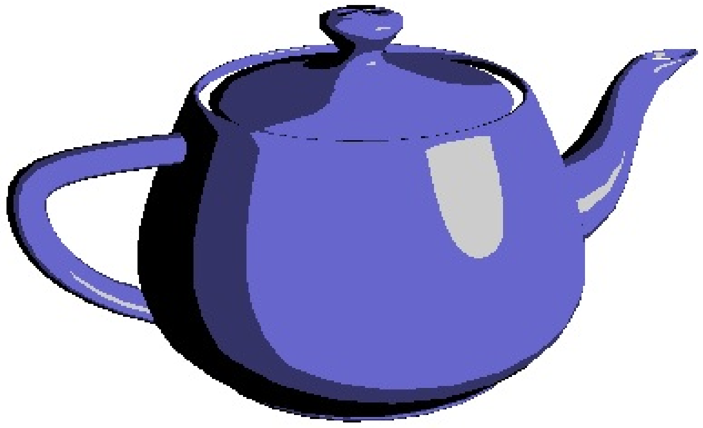

Figure 1.1: Toon Shader

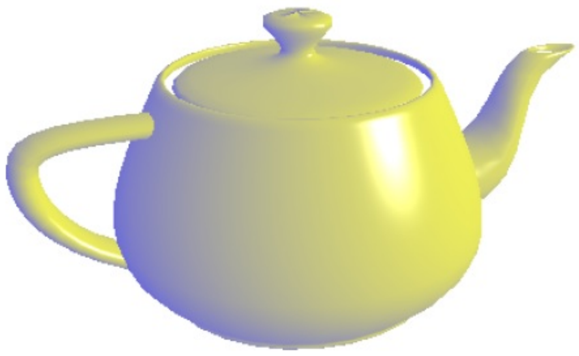

Figure 1.2: Gooch Shader

1.3 PROJECT IMPLEMENTATION

1.3.1 Base project

In the initial phase of the project, *07a-DiffuseLightingCube* was chosen as a base project for implementation and to test the gooch and toon shaders, where different simulated lighting has been applied to a rotating cube. Moreover, the toon shading part was included from the *Lecture-07-Shading*. In the later phase of the project, *08b-CubeMapOBJ* was chosen to implement the shaders on a model where various header files such as *shader_m.h*, *model.h* etc have been used to shorten the overall code and lessen the code complexity.

1.3.2 Normalization and Layout Qualifiers

In the initial phases of implementation of the project, the problem of normalization occurred in the gooch shader as well in the toon shader. OpenGL assumes that if we pass any normals, it is already of a unit length. Moreover, if the normals are not of unit length, we might have strange lighting results. To fix the issue of normalization of the vectors we have to use *"normalize"* function, which calculates the unit vector in the same direction as the original vector. Hence, in the shaders we have to normalize *nnormal*(normalised normal), *lightDir*(light direction), *ReflectVec*(reflection vector), and *ViewVec* (viewing vector) etc.

Moreover, we have to use layout qualifiers for example *layout (location = 0)* etc. in the gooch and toon vertex shaders. Vertex shaders inputs can specify the attribute index that the particular input uses. Using the layout location qualifiers, we can surpass the use of *glBindAttribLocation* entirely. We have used layout qualifiers in the gooch vertex shader which is given as:

```
layout (location = 0) in vec4 inVertex;
layout (location = 1) in vec3 inNormal;
```

and similarly for the toon vertex shader as:

```
layout (location = 0) in vec3 inVertex;
layout (location = 1) in vec3 inNormal;
```

1.3.3 Shader Class (shader_m.h)

A shader class was utilised to concatenate shader compilation process, taking the vertex and fragment strings, which link, compile and check for errors. Further in the class are functions used to simplify usage of *glUniform* functions in order to speed up coding time, they allow simple allocation of primitive data types, vectors and matrices.

The *shader.Set* functions were used to input parameters into the shader, such as *SurfaceColor*, *WarmColor*, *CoolColor*, *lightPos* etc. Next, *myModel.Draw(Shader)* function uses the shader to render each vertex in the mesh is provided. The main loop renders as gooch or toon, depending on an integer value, variable with the T and G keys, changing the appearance of the Buddha. Some example of the gooch shader attributes using *shader.Set* function linked with *(shader_m.h)* class are given below.

```
    goochShader.setVec3("SurfaceColor", SurfaceColor);
    goochShader.setVec3("WarmColor", WarmColor);
    goochShader.setVec3("CoolColor", CoolColor);
```

Similarly, an example of *.Draw* function is given below:

```
myModel.Draw(goochShader);
```

1.3.4 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 *SurfaceColor, WarmColor*, and *CoolColor* etc. in the main.cpp, these uniforms are defined as follows:

```
vec3 SurfaceColor = vec3(0.0, 0.0, 0.0);
vec3 WarmColor = vec3(0.5, 0.5, 0.0);
vec3 CoolColor = vec3(0.0, 0.0, 0.8);
```

These values can be altered within the main.cpp for different results. This allows user flexibility so that the user do not have to make changes in the fragment shaders. Moreover, *lightPos* is also defined in the main.cpp. This value is alters the light position for the model object. Users do not have to alter the value of *lightPos* in gooch and toon fragment shaders. The value for lightPos in the main.cpp is given below:

```
vec3 lightPos = vec3(1.0f, 1.0f, 2.0f);
```

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

```
Shader goochShader("goochShader.vert", "goochShader.frag");
Shader toonShader("toonShader.vert","toonShader.frag");
Model myModel("objects/happy-buddha/happy-buddha.obj");
```

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

```
    //gooch shader
    goochShader.setMat4("projection", projection); //setMat4 is used for projection matrix
    goochShader.setMat4("view", view);
    goochShader.setMat4("model", model);
    goochShader.setVec3("SurfaceColor", SurfaceColor); //setvec3 is used for SurfaceColor, a 3-component vector
    goochShader.setVec3("WarmColor", WarmColor);
    goochShader.setVec3("CoolColor", CoolColor);
    goochShader.setFloat("DiffuseWarm", DiffuseWarm); //setFloat is used for DiffuseWarm, a float value
    goochShader.setFloat("DiffuseCool", DiffuseCool);
    goochShader.setVec3("lightPos", lightPos);

    myModel.Draw(goochShader);
```

To switch between the shaders, if /else statement is used and an integer int currentShader = 0 has been defined in the main.cpp . The current shader is the gooch shader. The code for if/else is as follows:

```
    if (currentShader == 0)
    {
      myModel.Draw(goochShader);
      goochShader.use();
    }
    else
    {
      myModel.Draw(toonShader);
```

Finally, another if statement has been used for the key events to switch between the gooch and toon shaders. The code for key event is as follows:

```
  if (glfwGetKey(window, GLFW_KEY_G) == GLFW_PRESS)
  {
    currentShader = 0;

  } if (glfwGetKey(window, GLFW_KEY_T) == GLFW_PRESS)
  {
    currentShader = 1;
  }
```

1.4 SHADERS

Shaders are programs that rest in the GPU transforming inputs and outputs. Shaders programs uses *Qualifiers* such as *in, out, uniform.* The typical structure of a shader program is given below:

```
#version version_number
in type in_variable_name;
in type in_variable_name;
out type out_variable_name;
uniform type uniform_name;
void main()
{
// process input(s) and do some weird graphics stuff
...
// output processed stuff to output variable
out_variable_name = some_stuff_we_processed;
}
```

We will discuss various shaders used to implement the project in-detail in the next sections of the report.

1.4.1 Gooch Vertex Shader

For the gooch vertex shader using the base project *08b-CubeMapOBJ*, we use the full potential of *shader_m.h* and *model.h* header files. Shader class and model class provides some of the crucial attributes for our vertex shaders. *inVertex* and *inNormal* are the two important array which used at the beginning of our vertex shader which are given below:

```
layout (location = 0) in vec4 inVertex;
layout (location = 1) in vec3 inNormal;
```

The explanation for using *inVertex* and *inNormal* is as follows: *mVertices* is the vertex position array present in *mesh.h* header file. In our case this we call it as *inVertex. mNormals* is the vertex normals is an array which contains the normalized vectors also present in the mesh.h header file, and in our case we call it *inNormal.* But, a mesh consisting of points and lines only may not have been normalized. Hence, we have to normalize our *inNormal.* This is done in the vertex shader. We can define the *inVertex* and *inNormal* directly to our vertex shader files. The same concept has also been applied to toon vertex shader.

We also declare transformations matrices as uniforms which will be used to compute our *gl_position.* We also declare a *uniform vec3 lightPos* in the vertex shader and its values has be given in the main.cpp. We also declare outgoing *float NdotL, vec3 ReflectVec* and *vec3 viewVec* in our vertex shader. Let's discuss the main of our vertex shader.

We define a normal which is transpose of inverse of our model matrix multiplied by *inNormal.* Now we declare a *vec3 nnormal* to normalize previously computed normal. We also normalize our light direction *lightDir*, which is computed by subtracting *fragPos* from *lightPos.* We also calculate the reflection vector *ReflectVec* by using normalizing function *normalize*, reflection function *reflect* of the previously computed light direction *lightDir* and normalized normal *nnormal.* The main function of our vertex shader is given below:

```
void main()
{
  normal = mat3(transpose(inverse(model))) * inNormal;
  vec3 fragPos = vec3(model * inVertex);
  vec3 nnormal = normalize(normal); //nnormal is the normalized normal
  vec3 lightDir = normalize(lightPos - fragPos);
  ReflectVec = normalize(reflect(-lightDir, nnormal));
  ViewVec = normalize(-fragPos);
```

Finally, we can also calculate NdotL, the weighting factor in our vertex shader which will be passed to our fragment shader. float NdotL can be given as:

```
NdotL = dot(nnormal, lightDir) * 0.5 + 0.5;
```

The complete gooch vertex shader is given below:

```
#version 330 core
layout (location = 0) in vec4 inVertex;
layout (location = 1) in vec3 inNormal;

out vec3 normal;
out vec3 fragPos;

//declaring transformation matrices as uniforms
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

uniform vec3 lightPos; // defined in main

out float NdotL;
out vec3 ReflectVec;
out vec3 ViewVec;

void main()
{
  normal = mat3(transpose(inverse(model))) * inNormal;
  vec3 fragPos = vec3(model * inVertex);
  vec3 nnormal = normalize(normal); //nnormal is the normalized normal
  vec3 lightDir = normalize(lightPos - fragPos);
  ReflectVec = normalize(reflect(-lightDir, nnormal));
  ViewVec = normalize(-fragPos);

  NdotL = dot(nnormal, lightDir) * 0.5 + 0.5;

  gl_Position = projection * view * model * inVertex;
}
```

1.4.2 Gooch Fragment Shader

For the implementation of gooch effect most of the work is done on the fragment shader. 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 gooch fragment shader are given as:

```
uniform vec3 SurfaceColor;
uniform vec3 WarmColor;
uniform vec3 CoolColor;
uniform float DiffuseWarm;
uniform float DiffuseCool;
```

To demonstrate technical illustrations, there is reduction of excessive details and emphasizing important features. Standard lighting models are not suitable for such demonstration. *Amy Gooch* and *Bruce Gooch* developed a lighting model, which overcomes the problems which other lighting models cannot solve.

Details such as shadows and reflections are removed, instead a warm and cool color are defined to show the shape and curvature of the surface or a model. These are weighted based depending on the primary color of the object. The regions facing away from the light are blue (cool hue) and the regions facing the light is yellow (warm hue). Hence, gooch shading technique is also known as *"cool-to-warm"* shading. These cool and warm shades obtained then diffuses reflection in term of **L.N** and then interpolated over an object. The gooch lighting formula can be given as follows:

$$k_{\text{cool}} = k_{\text{blue}} + \alpha * k_{\text{diffuse}}$$

$$k_{\text{warm}} = k_{\text{yellow}} + \beta * k_{\text{diffuse}}$$

$$k_{\text{final}} = \left(\frac{1+N \bullet L}{2}\right) * k_{\text{cool}} + \left(1 - \frac{1+N \bullet L}{2}\right) * k_{\text{warm}}$$

Where,
k_{blue}– cool shade
k_{yellow}– warm shade
k_{diffuse}– primary color of the object
α– Weight offset against the primary color and cold tone
β– Weight offset against the primary color and warm tone

The code to compute *kcool, kwarm* and *kfinal* are given below:

```
vec3 kcool = min(CoolColor + DiffuseCool * SurfaceColor, 1.0);
vec3 kwarm = min(WarmColor + DiffuseWarm * SurfaceColor, 1.0);

vec3 kfinal = mix(kcool, kwarm, NdotL);
```

As we can recall that, *NdotL* has already been calculated in the vertex shader and passed onto the fragment shader for its use in the final color. Now, we use specular lighting which is based on the light's direction, object's normal vectors and also the view direction. The higher the correspondence of the reflected light beam and view vector the greater the effect of the highlights.

As we have already computed and normalized reflection vector *ReflectVec* and viewing vector *ViewVec*, they can be assigned as *nreflect* and *nview* respectively to compute the specular lighting. Now, to compute *float spec*, we perform the **dot** product of *nreflect* and *nview* vectors and also take the maximum value using the in-built max function. The code for the above mentioned process is given below:

```
vec3 nreflect = normalize(ReflectVec); //nreflect is normalised ReflectVec
vec3 nview = normalize(ViewVec); //nview is normalised ViewVec
```

```

float spec = max(dot(nreflect, nview), 0.0); //spec is specular
```

The complete gooch fragment shader is given below:

```
#version 330 core

in vec3 fragPos;
in vec3 normal;

uniform vec3 SurfaceColor;
uniform vec3 WarmColor;
uniform vec3 CoolColor;
uniform float DiffuseWarm;
uniform float DiffuseCool;

in float NdotL; //the weighting factor is NdotL, the lighting value

//variables passed by the vertex shader
in vec3 ReflectVec;
in vec3 ViewVec;

out vec4 fragColor;

void main() {
  //combination of CoolColor, WarmColor, and surface color
  vec3 kcool = min(CoolColor + DiffuseCool * SurfaceColor, 1.0);
  vec3 kwarm = min(WarmColor + DiffuseWarm * SurfaceColor, 1.0);

  //interpolation by dot product of normal and light vector
  vec3 kfinal = mix(kcool, kwarm, NdotL);

  //specular(spec) highlighted by reflected lightVec and viewVec
  vec3 nreflect = normalize(ReflectVec);
  vec3 nview = normalize(ViewVec);

  float spec = max(dot(nreflect, nview), 0.0);
  spec = pow(spec, 32.0);

  fragColor = vec4(min(kfinal + spec, 1.0), 1.0);
}
```

The results for the gooch shader on a Buddha model is shown are the Figure.1.3 and Figure.1.4.

1.4.3 Toon Vertex Shader

Similar to the gooch vertex shader, we use *08b-CubeMapOBJ* as a base project for the toon shader as well. We have already discussed about the *inVertex* and *inNormal* in-detail in the gooch vertex shader. The only minor difference between the gooch and toon vertex shader is the calculation of *fragPos* which is obtained by multiplying the model matrix with *inVertex*, and the *gl_Position* is calculated by projection matrix, view matrix with a 4 coordinate vector having the value of previously calculated 3 coordinate vector *fragPos* value. The complete toon vertex shader is given below:

```
#version 330 core
```

```
layout (location = 0) in vec3 inVertex;
layout (location = 1) in vec3 inNormal;

out vec3 normal;
out vec3 fragPos;

//declaring transformation matrices as uniforms
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
    normal = mat3(transpose(inverse(model))) * inNormal;

    fragPos = vec3(model * vec4(inVertex, 1.0));
    //Multipy transformation matrices with vertex coordinates
    //Note that we read the multiplication from right to left
    gl_Position = projection * view * vec4(fragPos, 1.0);
}
```

1.4.4 Toon Fragment Shader

To implement the toon shader, we have to work mostly with the fragment shader. To demonstrate a working 3D toon effect, we use tones. In the fragment shader, tones are achieved on the angles. This angle is the cosine of the angle between light direction and normal to the surface. For the tones to work properly a light source of different types are also required. To implement the toon effect we have to use *Ambient* and *Diffuse* lighting. We also have to normalize the light sources. The following have been used to implement the toon fragment shader:

- *normal*: incoming normal vector from the vertex shader
- *lightDir*: light direction which is used for angle calculation
- *intensity*: intensity variable with cosine of the angle values, which can be altered in the fragment shader for different effects
- *lightColor*: a vector for light color, white light in our case
- *ambient* and *diffuse* light models

The fragment shader receives the normal coming from the vertex shader. This normal has to be normalized. The light direction *lightDir* also has to be normalized before using it to calculate the intensity. We are using the diffuse lighting, which requires a normalized normal (*nnormal*) vector and light direction. The following code snippet shows the mentioned calculations.

```
// light
  vec3 lightColor = vec3(1.0, 1.0, 1.0);

  // ambient
  float ambientFactor = 0.1;
  vec3 ambient = ambientFactor * lightColor;

  // diffuse
  vec3 nnormal = normalize(normal); // incoming normal vector normalized for the diffuse
      light, nnnormal will be our normalized normal
  vec3 lightDir = normalize(lightPos - fragPos); // normalizing the lightDir aswell
  float diffFactor = max(dot(nnormal, lightDir), 0.0);
  vec3 diffuse = diffFactor * lightColor;
```

So far we have normalized *lightDir* and calculated its value by subtracting *fragPos* coming from the vertex shader from the light position (*lightPos*) which is a uniform defined in the fragment shader. The value of *lightDir* will be later used to calculate the intensity based on the normalized *lightDir* and *nnormal*. intensity calculated is the cosine between normalized *lightDir* and *nnormal* which can be given as follows:

```
float intensity = max(dot(nnormal, lightDir), 0.0); // intensity now uses proper
    normalized normal vector and lightDir
```

As we have calculated the *intensity*, we can define different intensity values which provides different intensity levels for our toon effect. This *intensity* variable will be used for color intensity in our fragment shader as we know that fragment shader is accountable for the colors. Finally, we define a color in our fragment shader based on the intensity calculated which is shown in the following code snippet:

```
float intensity = max(dot(nnormal, lightDir), 0.0); // intensity now uses proper
    normalized normal vector and lightDir
  vec3 toonColor;

  if (intensity > 0.95) // cosine greater than 0.95 will result  in
    toonColor = vec3(1.0, 1.0, 1.0);
  else if (intensity > 0.75) toonColor = vec3(0.8, 0.8, 0.8);
  else if (intensity > 0.50) toonColor = vec3(0.6, 0.6, 0.6);
  else if (intensity > 0.25) toonColor = vec3(0.4, 0.4, 0.4);
  else                  toonColor = vec3(0.2, 0.2, 0.2);

  vec3 resultColor = toonColor * vec3(1.0, 0.0, 0.0);
```

The complete toon fragment shader has been given below:

```
#version 330 core

uniform vec3 lightPos;

in vec3 fragPos;
in vec3 normal;

out vec4 fragColor;

void main() {

  // light
  vec3 lightColor = vec3(1.0, 1.0, 1.0);

  // ambient
  float ambientFactor = 0.1;
  vec3 ambient = ambientFactor * lightColor;

  // diffuse
  vec3 nnormal = normalize(normal);
  vec3 lightDir = normalize(lightPos - fragPos);
  float diffFactor = max(dot(nnormal, lightDir), 0.0);
  vec3 diffuse = diffFactor * lightColor;

  vec3 result = (ambient + diffuse) * vec3(1.0, 1.0, 1.0);

  float intensity = max(dot(nnormal, lightDir), 0.0);
  vec3 toonColor;

  if (intensity > 0.95)
    toonColor = vec3(1.0, 1.0, 1.0);
  else if (intensity > 0.75) toonColor = vec3(0.8, 0.8, 0.8);
  else if (intensity > 0.50) toonColor = vec3(0.6, 0.6, 0.6);
  else if (intensity > 0.25) toonColor = vec3(0.4, 0.4, 0.4);
  else             toonColor = vec3(0.2, 0.2, 0.2);

  vec3 resultColor = toonColor * vec3(1.0, 0.0, 0.0);

  fragColor = vec4(resultColor, 1.0);
}
```

The results for the toon shader on a Buddha model is shown are the Figure.1.5 and Figure.1.6. The *resultColor* is the color calculated by the different color values multiplied by a vec3(1.0, 0.0, 0.0). This vec3(1.0, 0.0, 0.0) values can be changed to produce different colors in our toon effect such as vec3(0.0, 1.0, 0.0) will result in green effect which shown in the Figure.1.7 and Figure.1.8.

1.5 Summary and Conclusion

In this project a Gooch shader and a toon shader was implemented using OpenGL and GLSL into a Buddha model object. Source codes *(07a-DiffuseLightingCube* and *08b-CubeMapOBJ)* from the lectures was used as base for the project. Some changes have been made such as adding gooch attributes etc. were added in the main.cpp.

A user can change the values of *SurfaceColor* (default is set to white), *WarmColor*, and *CoolColor* values to change the appearance of the gooch model object to appear more/less yellowish and more/less bluish respectively. Changing the value of *lightPos* in the main.cpp will result in different gooch and toon effect on the model object.

Moreover, users are able to switch between the shaders using keys. The default shader in use is the gooch shader. Pressing key T will let the user to switch to toon shader, and pressing key G will switch to gooch shader again. Some improvements can be made such as edge detection to detect the silhouette in the model. Raskar and Cohen silhouette creation method can be used to draw the silhouettes.

1.6 RESULTS

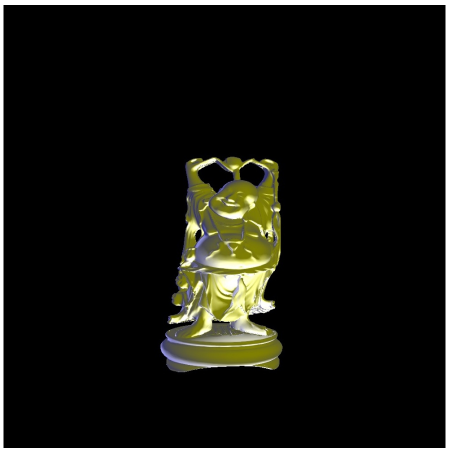

Figure 1.3: Gooch Shader front

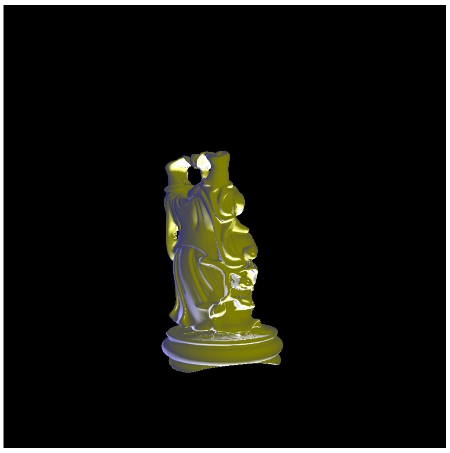

Figure 1.4: Gooch Shader back

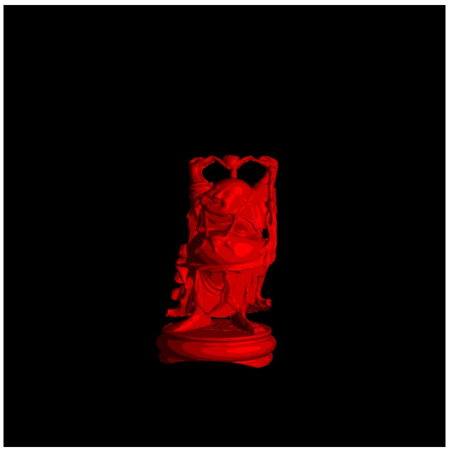

Figure 1.5: Toon Shader front

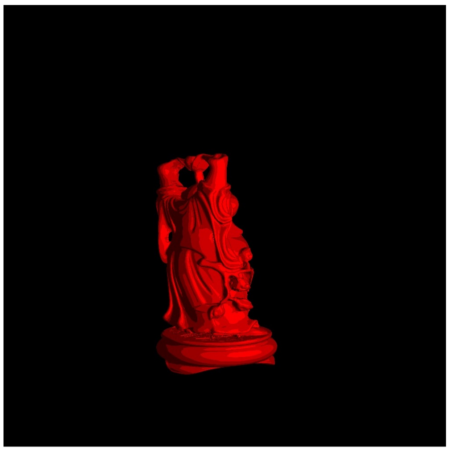

Figure 1.6: Toon Shader back

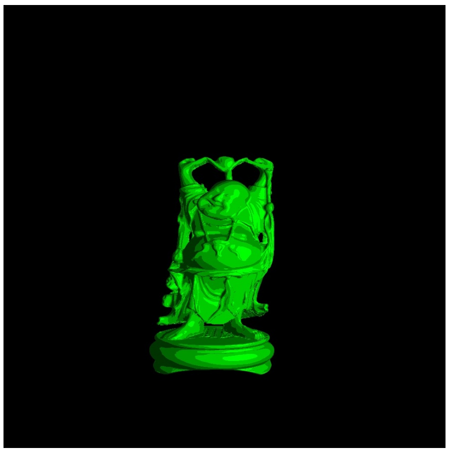

Figure 1.7: Toon Shader green front

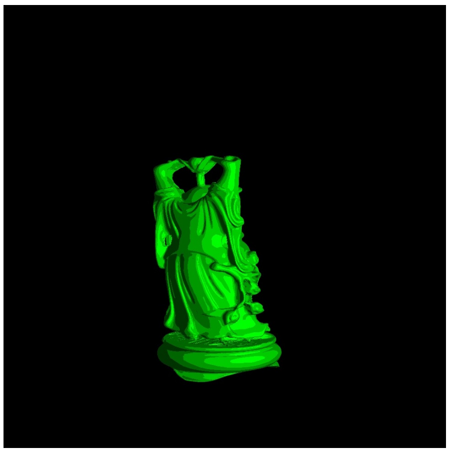

Figure 1.8: Toon Shader green back