# 实验报告3

姓名：蒋卓洋

学号：59119125

### 实现

1. 函数位置：rasterizer.cpp

·函数实现：

1. rasterize\_triangle()
2. //Screen space rasterization
3. void rst::rasterizer::rasterize\_triangle(const Triangle& t, const std::array<Eigen::Vector3f, 3>& view\_pos)
4. {
5. // TODO: From your HW3, get the triangle rasterization code.
6. // TODO: Inside your rasterization loop:
7. //    \* v[i].w() is the vertex view space depth value z.
8. //    \* Z is interpolated view space depth for the current pixel
9. //    \* zp is depth between zNear and zFar, used for z-buffer
10. // float Z = 1.0 / (alpha / v[0].w() + beta / v[1].w() + gamma / v[2].w());
11. // float zp = alpha \* v[0].z() / v[0].w() + beta \* v[1].z() / v[1].w() + gamma \* v[2].z() / v[2].w();
12. // zp \*= Z;
13. // TODO: Interpolate the attributes:
14. // auto interpolated\_color
15. // auto interpolated\_normal
16. // auto interpolated\_texcoords
17. // auto interpolated\_shadingcoords
18. // Use: fragment\_shader\_payload payload( interpolated\_color, interpolated\_normal.normalized(), interpolated\_texcoords, texture ? &\*texture : nullptr);
19. // Use: payload.view\_pos = interpolated\_shadingcoords;
20. // Use: Instead of passing the triangle's color directly to the frame buffer, pass the color to the shaders first to get the final color;
21. // Use: auto pixel\_color = fragment\_shader(payload);
22. //////////////////Solution////////////////////
23. ////Name:JiangZhuoyang
24. ////StudentID:58119125
25. ////FinishDate:21/10/23
26. //1.COnstruct the bounding box with for value, the value is difined by the 4 extremums in to directions
27. //(1)Get bound
28. auto v = t.toVector4();
29. float bound\_L = std::min(v[0][0], std::min(v[1][0],v[2][0]));//Left bound: bounded by the minimum of x-coordinate of three points of triangle
30. float bound\_R = std::max(v[0][0], std::max(v[1][0],v[2][0]));//Right bound: bounded by the maximum of x-coordinate of three points of triangle
31. float bound\_T = std::min(v[0][1], std::min(v[1][1],v[2][1]));//Top bound: bounded by the minimum of y-coordinate of three points of triangle
32. float bound\_B = std::max(v[0][1], std::max(v[1][1],v[2][1]));//Bottom bound: bounded by the maximum of y-coordinate of three points of triangle
33. //(2)Nomalize to integer for iteration
34. bound\_L = (int)std::floor(bound\_L);//round down the left bound
35. bound\_R = (int)std::ceil(bound\_R); //round up the right bound
36. bound\_T = (int)std::floor(bound\_T);//round down the top bound
37. bound\_B = (int)std::ceil(bound\_B);  //round up the bottom bound
38. //2.Iterate through the pixel in the bound box and find if the current pixel is inside the triangle
39. for(int x = bound\_L; x <= bound\_R; x++){
40. for(int y = bound\_T; y <= bound\_B; y++){
41. //(1)Judge if the current pixel is inside the triangle
43. if(insideTriangle(x+0.5, y+0.5, t.v)){
44. //(2)Depth interpolate:
45. //A.define the min depth, innitialize it with infinite.
46. float depth\_min = FLT\_MAX;
47. //B.calculate the Barycentric coordinate weight
48. auto tuple = computeBarycentric2D(x, y, t.v);
49. float alpha, beta, gamma;
50. std::tie(alpha, beta, gamma) = tuple; // Debug the given method
51. //C.interpolate the depth with Barycentric coordinate weight:
52. //a.normalization: satisfies "alpha + beta + gamma = 1"???????????????????????
53. float w\_reciprocal = 1.0/(alpha / v[0].w() + beta / v[1].w() + gamma / v[2].w());
54. //b.
55. float z\_interpolated = alpha \* v[0].z() / v[0].w() + beta \* v[1].z() / v[1].w() + gamma \* v[2].z() / v[2].w();
56. z\_interpolated \*= w\_reciprocal;
57. //D.get the min depth
58. depth\_min = std::min(depth\_min,z\_interpolated);
59. //(3)Interpolate the attributes under the premise of depth
60. if(depth\_min < depth\_buf[get\_index(x,y)]){//the current point is more shallow
61. //A.Renew the z-buffer with current point
62. depth\_buf[get\_index(x,y)] = depth\_min;
63. //B.INTERPOLATE!!
64. //a.color
65. auto interpolated\_color = alpha \* t.color[0] + beta \* t.color[1] + gamma \* t.color[2];
66. //b.normal
67. auto interpolated\_normal = alpha \* t.normal[0] + beta \* t.normal[1] + gamma \* t.normal[2];
68. //c.texture
69. auto interpolated\_texcoords = alpha \* t.tex\_coords[0] + beta \* t.tex\_coords[1] + gamma \* t.tex\_coords[2];
70. //d.shadingcoords
71. auto interpolated\_shadingcoords = alpha \* view\_pos[0] + beta \* view\_pos[1] + gamma \* view\_pos[2];
72. //C.Interpolate Result transportation
73. fragment\_shader\_payload payload( interpolated\_color, interpolated\_normal.normalized(), interpolated\_texcoords, texture ? &\*texture : nullptr);
74. payload.view\_pos = interpolated\_shadingcoords;
75. auto pixel\_color = fragment\_shader(payload);
76. //D.Instead of passing the triangle's color directly to the frame buffer, pass the color to the shaders first to get the final color;
77. set\_pixel(Vector2i(x,y), pixel\_color);
78. }
80. }
81. }
82. }
83. }
84. 函数位置：main.cpp

·函数实现：

1. get\_projection\_matrix()
2. Eigen::Matrix4f get\_projection\_matrix(float eye\_fov, float aspect\_ratio, float zNear, float zFar)
3. {
4. // Students will implement this function
5. Eigen::Matrix4f projection = Eigen::Matrix4f::Identity();
6. // TODO: Implement this function
7. // Create the projection matrix for the given parameters.
8. // Then return it.
9. //////////////////Solution////////////////////
10. ////Name:JiangZhuoyang
11. ////StudentID:58119125
12. ////FinishDate:21/9/30
13. //1.Definition
14. Eigen::Matrix4f perspective = Eigen::Matrix4f::Identity();//perspective projection
15. Eigen::Matrix4f persp\_to\_ortho = Eigen::Matrix4f::Identity();//turn the perspective projection to orthographic projection
16. Eigen::Matrix4f orthographic = Eigen::Matrix4f::Identity();//orthographic projection
17. //2.Construction
18. //2.1.P -> O
19. persp\_to\_ortho << zNear,0,0,0,
20. 0,zNear,0,0,
21. 0,0,zNear+zFar,-zNear\*zFar,
22. 0,0,1,0;
23. //2.2.Orthographic
24. //(1)Get edges: implicitly, do the first translation with the use of eye fov and aspect ration directly.
25. float yTop = -zNear \* tan( (eye\_fov/2) \* MY\_PI / 180 );
26. float yBottom = (-1) \* yTop;
27. float xRight = yTop \* aspect\_ratio;
28. float xLeft = (-1) \* xRight;
29. //(2)Orthographic translate:
30. Eigen::Matrix4f ortho\_trans = Eigen::Matrix4f::Identity();
31. ortho\_trans << 1, 0, 0, -(xRight + xLeft) / 2, //(1,0,0,0)
32. 0, 1, 0, -(yTop + yBottom) / 2, //(0,1,0,0)
33. 0, 0, 1, -(zNear + zFar) / 2,
34. 0, 0, 0, 1;
35. //(3)Orthographic scale:
36. Eigen::Matrix4f ortho\_scale = Eigen::Matrix4f::Identity();
37. ortho\_scale << 2/(xRight - xLeft), 0, 0, 0, //(1,0,0,0)
38. 0, 2/(yTop - yBottom), 0, 0, //(0,1,0,0)
39. 0, 0, 2/(zNear - zFar), 0,
40. 0, 0, 0, 1;
41. //(3)Orthographic Matrix:
42. orthographic = ortho\_scale \* ortho\_trans;
43. //2.3.Perspective:
44. perspective = orthographic \* persp\_to\_ortho;
45. //3.projection:
46. projection = perspective;
47. /////////////////////////////////////////////
48. return projection;
49. }
50. texture\_fragment\_shader()
51. Eigen::Vector3f texture\_fragment\_shader(const fragment\_shader\_payload& payload)
52. {
53. Eigen::Vector3f return\_color = {0, 0, 0};
54. if (payload.texture)
55. {
56. // TODO: Get the texture value at the texture coordinates of the current fragment
57. return\_color = payload.texture->getColor(payload.tex\_coords.x(),payload.tex\_coords.y());
58. }
59. Eigen::Vector3f texture\_color;
60. texture\_color << return\_color.x(), return\_color.y(), return\_color.z();
61. Eigen::Vector3f ka = Eigen::Vector3f(0.005, 0.005, 0.005);
62. Eigen::Vector3f kd = texture\_color / 255.f;
63. Eigen::Vector3f ks = Eigen::Vector3f(0.7937, 0.7937, 0.7937);
64. auto l1 = light{{20, 20, 20}, {500, 500, 500}};
65. auto l2 = light{{-20, 20, 0}, {500, 500, 500}};
66. std::vector<light> lights = {l1, l2};
67. Eigen::Vector3f amb\_light\_intensity{10, 10, 10};
68. Eigen::Vector3f eye\_pos{0, 0, 10};
69. float p = 150;//fir specular light
70. Eigen::Vector3f color = texture\_color;
71. Eigen::Vector3f point = payload.view\_pos;
72. Eigen::Vector3f normal = payload.normal;
73. Eigen::Vector3f result\_color = {0, 0, 0};
74. for (auto& light : lights)
75. {
76. // TODO: For each light source in the code, calculate what the \*ambient\*, \*diffuse\*, and \*specular\*
77. // components are. Then, accumulate that result on the \*result\_color\* object.
78. //////////////////Solution////////////////////
79. ////Name:JiangZhuoyang
80. ////StudentID:58119125
81. ////FinishDate:21/10/22
82. //1.Get all the vectors we need to use
83. //(1).Light direction
84. Eigen::Vector3f light\_dir = light.position - point;
85. //(2).Viewer direction
86. Eigen::Vector3f viewer\_dir = eye\_pos - point;
87. //(3).Surface normal:has had
88. //(4).Half vector
89. Eigen::Vector3f h = (light\_dir + viewer\_dir).normalized();
90. //2.Difine light distance to represent the light fallof
91. float r\_2 = light\_dir.dot(light\_dir);
92. //3.Calculate all 3 kings of light
93. //(1).Ambient:
94. Eigen::Vector3f La = ka.cwiseProduct(amb\_light\_intensity);
95. //(2).Diffuse:
96. Eigen::Vector3f Ld = kd.cwiseProduct(light.intensity / r\_2) \* std::max(0.0f, normal.normalized().dot(light\_dir.normalized()));
97. //(3).Specular:
98. Eigen::Vector3f Ls = ks.cwiseProduct(light.intensity / r\_2) \* std::pow(std::max(0.0f, normal.normalized().dot(h)) , p);
99. //4.get result of Blinn-Phong Model
100. result\_color += (La + Ld + Ls);
101. //////////////////////////////////////////////
102. }
103. return result\_color \* 255.f;
104. }
105. phong\_fragment\_shader()
106. Eigen::Vector3f phong\_fragment\_shader(const fragment\_shader\_payload& payload)
107. {
108. //Three coefficient
109. Eigen::Vector3f ka = Eigen::Vector3f(0.005, 0.005, 0.005);
110. Eigen::Vector3f kd = payload.color;
111. Eigen::Vector3f ks = Eigen::Vector3f(0.7937, 0.7937, 0.7937);
112. auto l1 = light{{20, 20, 20}, {500, 500, 500}};
113. auto l2 = light{{-20, 20, 0}, {500, 500, 500}};
114. std::vector<light> lights = {l1, l2};
115. Eigen::Vector3f amb\_light\_intensity{10, 10, 10};
116. Eigen::Vector3f eye\_pos{0, 0, 10};
117. float p = 150;
118. Eigen::Vector3f color = payload.color;
119. Eigen::Vector3f point = payload.view\_pos;
120. Eigen::Vector3f normal = payload.normal;
121. Eigen::Vector3f result\_color = {0, 0, 0};
122. for (auto& light : lights)
123. {
124. // TODO: For each light source in the code, calculate what the \*ambient\*, \*diffuse\*, and \*specular\*
125. // components are. Then, accumulate that result on the \*result\_color\* object.
126. //////////////////Solution////////////////////
127. ////Name:JiangZhuoyang
128. ////StudentID:58119125
129. ////FinishDate:21/10/22
130. //1.Get all the vectors we need to use
131. //(1).Light direction
132. Eigen::Vector3f light\_dir = (light.position - point).normalized();
133. //(2).Viewer direction
134. Eigen::Vector3f viewer\_dir = (eye\_pos - point).normalized();
135. //(3).Surface normal:has had
136. //(4).Half vector
137. Eigen::Vector3f h = (light\_dir+viewer\_dir).normalized();
138. //2.Difine light distance to represent the light fallof
139. float r\_2 = (light.position - point).dot(light.position - point);
140. //3.Calculate all 3 kings of light
141. //(1).Ambient:
142. Eigen::Vector3f La = ka.cwiseProduct(amb\_light\_intensity);
143. //(2).Diffuse:
144. Eigen::Vector3f Ld = kd.cwiseProduct((light.intensity / r\_2) \* std::max(0.0f, normal.dot(light\_dir)));
145. //(3).Specular:
146. Eigen::Vector3f Ls = ks.cwiseProduct((light.intensity / r\_2) \* std::pow(std::max(0.0f, normal.dot(h)) , p));
147. //4.get result of Blinn-Phong Model
148. result\_color += (La + Ld + Ls);
149. //////////////////////////////////////////////
150. }
151. return result\_color \* 255.f;
152. }
153. bump\_fragment\_shader()
154. Eigen::Vector3f bump\_fragment\_shader(const fragment\_shader\_payload& payload)
155. {
157. Eigen::Vector3f ka = Eigen::Vector3f(0.005, 0.005, 0.005);
158. Eigen::Vector3f kd = payload.color;
159. Eigen::Vector3f ks = Eigen::Vector3f(0.7937, 0.7937, 0.7937);
160. auto l1 = light{{20, 20, 20}, {500, 500, 500}};
161. auto l2 = light{{-20, 20, 0}, {500, 500, 500}};
162. std::vector<light> lights = {l1, l2};
163. Eigen::Vector3f amb\_light\_intensity{10, 10, 10};
164. Eigen::Vector3f eye\_pos{0, 0, 10};
165. float p = 150;
166. Eigen::Vector3f color = payload.color;
167. Eigen::Vector3f point = payload.view\_pos;
168. Eigen::Vector3f normal = payload.normal;
169. float kh = 0.2, kn = 0.1;
170. // TODO: Implement bump mapping here
171. // Let n = normal = (x, y, z)
172. // Vector t = (x\*y/sqrt(x\*x+z\*z),sqrt(x\*x+z\*z),z\*y/sqrt(x\*x+z\*z))
173. // Vector b = n cross product t
174. // Matrix TBN = [t b n]
175. // dU = kh \* kn \* (h(u+1/w,v)-h(u,v))
176. // dV = kh \* kn \* (h(u,v+1/h)-h(u,v))
177. // Vector ln = (-dU, -dV, 1)
178. // Normal n = normalize(TBN \* ln)
179. //////////////////Solution////////////////////
180. ////Name:JiangZhuoyang
181. ////StudentID:58119125
182. ////FinishDate:21/10/23
183. //Follow the cue
184. float x = normal.x(), y = normal.y(), z = normal.z();
185. Eigen::Vector3f t = Vector3f( x\*y / std::sqrt(x\*x + y\*y), std::sqrt(x\*x + z\*z), y\*z / std::sqrt(x\*x + z\*z) );
186. Eigen::Vector3f b = normal.cross(t);
187. Eigen::Matrix3f TBN;
188. TBN << t,b,normal;
189. float u = payload.tex\_coords.x();
190. float v = payload.tex\_coords.y();
191. float w = payload.texture->width;
192. float h = payload.texture->height;
193. float dU = kh \* kn \*( payload.texture->getColor( (u+1.0f)/w, v ).norm() - payload.texture->getColor(u,v).norm() );
194. float dV = kh \* kn \*( payload.texture->getColor( u, (v+1.0f)/h ).norm() - payload.texture->getColor(u,v).norm() );
195. Eigen::Vector3f Ln{-dU,-dV,1};
196. normal = (TBN \*Ln).normalized();
197. Eigen::Vector3f result\_color = {0, 0, 0};
198. result\_color = normal;
199. return result\_color \* 255.f;
200. }

### **结果**

**实验结果如下：**

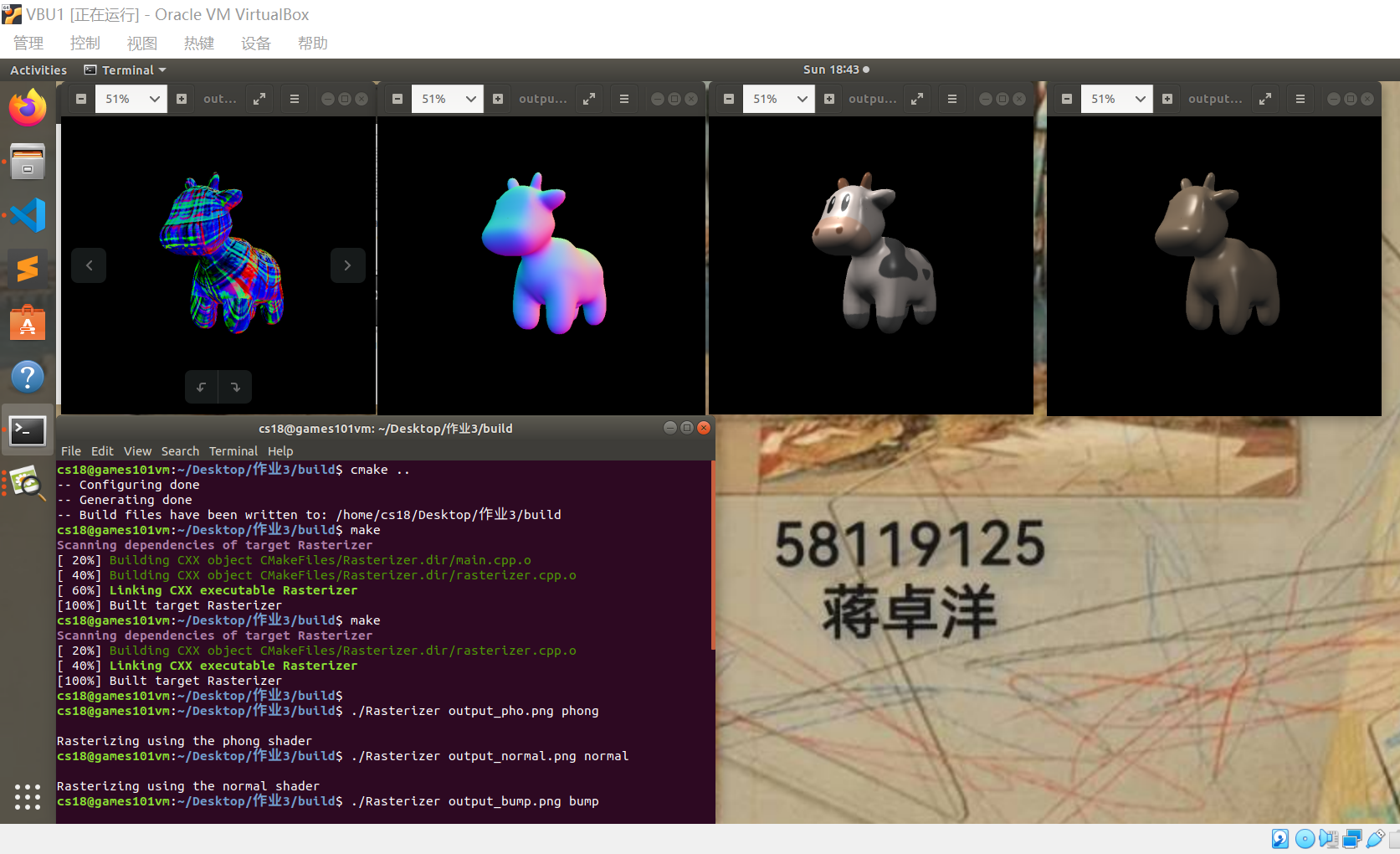
****

图1.作业编译三结果



图2.作业三结果--phong



图3.作业三结果--bump



图4.作业三结果--normal



图5.作业三结果--texture