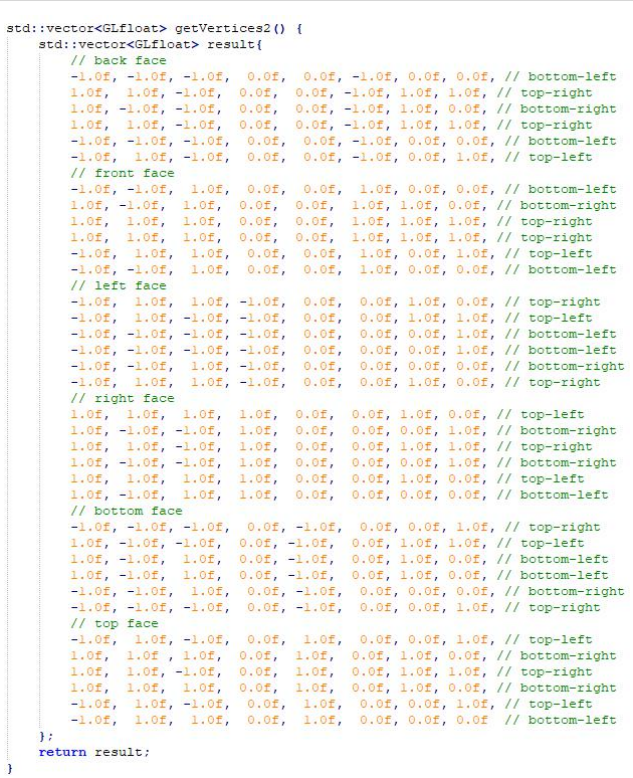
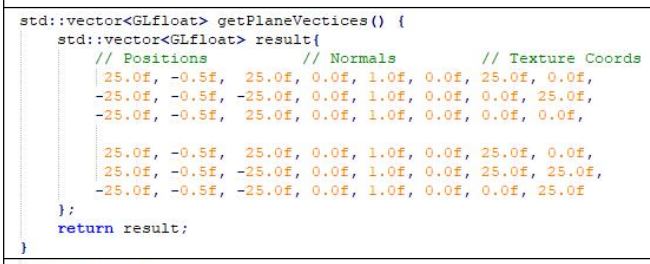
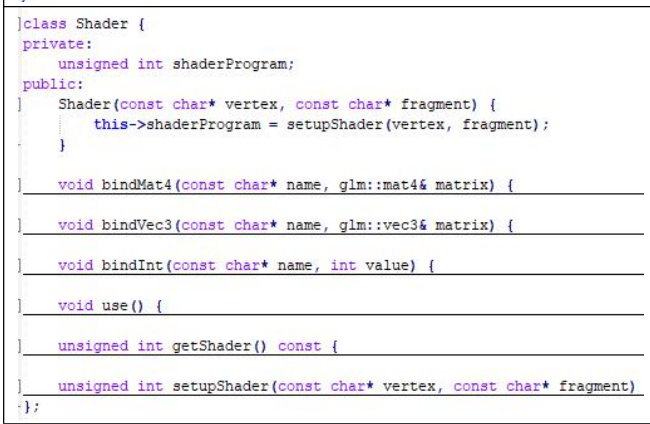
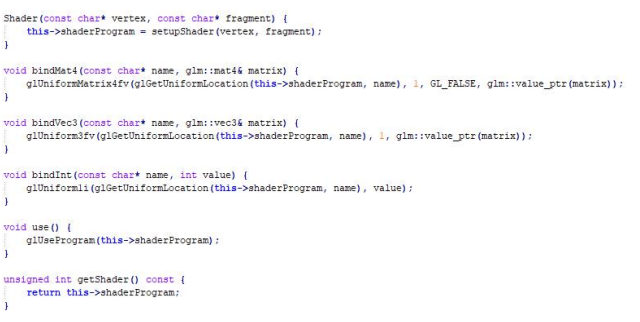
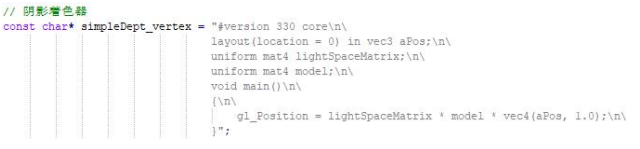
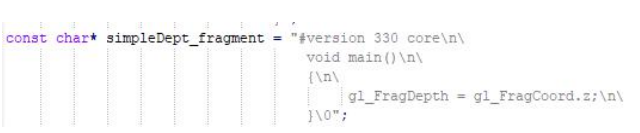
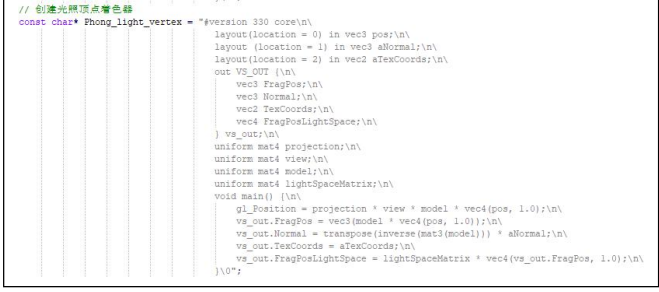
计算机图形学实验报告

第一部分：实现思路和代码分析

代码截图	代码说明
	将前面使用的立方体坐标代码稍作修改，将立方体中的每个顶点的坐标向量改为由位置坐标，法向量，纹理坐标组成的八维向量。
	平面的顶点数组，将平面划分成两个三角形，共有六个顶点。每个顶点的同样使用一个八维坐标向量表示，与立方体顶点的表示方法相同。
	将着色器 Shader 的配置独立成一个类，使主体代码更加简洁。这个类提供了着色器的初始化操作，同时支持不同类型的外部变量的绑定（数字，矩阵和向量）。

	着色器 Shader 类的功能函数具体实现。这部分的代码主要实现了着色器代码的外部变量的绑定以及着色器程序的调用。
	着色器 Shader 类的功能函数具体实现。这部分与前几次作业中的着色器生成程序一致，首先编译顶点着色器，然后编译片段着色器，最后创建着色器程序。
	生成深度贴图所需的顶点着色器。这部分主要是将顶点坐标变换到光空间中。
	生成深度贴图所需的片段着色器。由于深度贴图没有颜色缓冲，因此最后的片元不需要任何处理，因此使用一个空的片段着色器即可。
	渲染阴影的顶点着色器。该顶点着色器使用同一个光空间转换矩阵，把世界空间顶点位置转换为光空间。顶点着色器传递一个普通的经变换的世界空间顶点位置和一个光空间的顶点位置给像素着色器。

<table>
<tr>
<td>

```
// 创建光照片段着色器
const char* Phong_light_fragment = "#version 330 core\n\
                                   out vec4 FragColor;\n\
                                   in VS_OUT{\n\
                                       vec3 FragPos;\n\
                                       vec3 Normal;\n\
                                       vec2 TexCoords;\n\
                                       vec4 FragPosLightSpace;\n\
                                   } fs_in;\n\
                                   uniform sampler2D diffuseTexture;\n\
                                   uniform sampler2D shadowMap;\n\
                                   uniform vec3 lightPos;\n\
                                   uniform vec3 viewPos;\n\
                                   float ShadowCalculation(vec4 fragPosLightSpace)\n\
                                   {\n\
                                       // perform perspective divide\n\
                                       vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;\n\
                                       // transform to [0,1] range\n\
                                       projCoords = projCoords * 0.5 + 0.5;\n\
                                       // get closest depth value from light's perspective (using [0,1] range fragPosLight as coords)\n\
                                       float closestDepth = texture(shadowMap, projCoords.xy).r;\n\
                                       // get depth of current fragment from light's perspective\n\
                                       float currentDepth = projCoords.z;\n\
                                       // calculate bias (based on depth map resolution and slope)\n\
                                       vec3 normal = normalize(fs_in.Normal);\n\
                                       vec3 lightDir = normalize(lightPos - fs_in.FragPos);\n\
                                       float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);\n\
                                       // check whether current frag pos is in shadow\n\
                                       // float shadow = currentDepth - bias > closestDepth  ? 1.0 : 0.0;\n\
                                       // PCF\n\
                                       float shadow = 0.0; \n\
                                       vec2 texelSize = 1.0 / textureSize(shadowMap, 0);\n\
                                       for (int x = -1; x <= 1; ++x)\n\
                                       {\n\
                                           for (int y = -1; y <= 1; ++y)\n\
                                           {\n\
                                               float pcfDepth = texture(shadowMap, projCoords.xy + vec2(x, y) * texelSize).r;\n\
                                               shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;\n\
                                           }\n\
                                       }\n\
                                       shadow /= 9.0;\n\
                                       // keep the shadow at 0.0 when outside the far_plane region of the light's frustum.\n\
                                       if (projCoords.z > 1.0)\n\
                                           shadow = 0.0;\n\
                                       return shadow;\n\
                                   }\n\
                                   \n\
                                   void main()\n\
                                   {\n\
                                       vec3 color = texture(diffuseTexture, fs_in.TexCoords).rgb;\n\
                                       vec3 normal = normalize(fs_in.Normal);\n\
                                       vec3 lightColor = vec3(0.3);\n\
                                       // ambient\n\
                                       vec3 ambient = 0.3 * color; \n\
                                       // diffuse\n\
                                       vec3 lightDir = normalize(lightPos - fs_in.FragPos);\n\
                                       float diff = max(dot(lightDir, normal), 0.0);\n\
                                       vec3 diffuse = diff * lightColor;\n\
                                       // specular\n\
                                       vec3 viewDir = normalize(viewPos - fs_in.FragPos);\n\
                                       vec3 reflectDir = reflect(-lightDir, normal);\n\
                                       float spec = 0.0;\n\
                                       vec3 halfwayDir = normalize(lightDir + viewDir);\n\
                                       spec = pow(max(dot(normal, halfwayDir), 0.0), 64.0);\n\
                                       vec3 specular = spec * lightColor;\n\
                                       // calculate shadow\n\
                                       float shadow = ShadowCalculation(fs_in.FragPosLightSpace); \n\
                                       shadow = min(shadow, 0.75); \n\
                                       vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) * color; \n\
                                       FragColor = vec4(lighting, 1.0); \n\
                                   };";
```

</td>
<td>渲染阴影的顶点着色器。本次作业的顶点着色器使用 Blinn-Phong 光照模型渲染场景。接着计算出一个 shadow 值，当片元在阴影中时 shadow 值为 1.0，在阴影外是 0.0。计算过程大致如下：把光空间片元位置转换为裁切空间的标准化设备坐标；将坐标向量都需要变换到[0,1]范围；得到光的位置视野下最近的深度；得到片元的当前深度；判断片元是否在阴影中。这部分还包括了一些阴影优化的代码。</td>
</tr>
<tr>
<td>

```
// 开启深度测试
glEnable(GL_DEPTH_TEST);
// 着色器
Shader normalShader(Phong_light_vertex, Phong_light_fragment);
Shader simpleDepthShader(simpleDept_vertex, simpleDept_fragment);

// 贴图
unsigned int woodTexture = loadTexture("./wood.png");
```

</td>
<td>开启深度测试，以及着色器和纹理贴图的初始化。</td>
</tr>
<tr>
<td>

```
// Setup plane VAO
GLuint planeVBO;
glGenVertexArrays(1, &planeVAO);
glGenBuffers(1, &planeVBO);
glBindVertexArray(planeVAO);
glBindBuffer(GL_ARRAY_BUFFER, planeVBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) * total_p_v, planeVertices, GL_STATIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (GLvoid*)(3 * sizeof(GLfloat)));
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (GLvoid*)(6 * sizeof(GLfloat)));
glBindVertexArray(0);
```

</td>
<td>平面顶点的初始化。与立方体创建方式相同。</td>
</tr>
<tr>
<td>

```
// Configure depth map FBO
const GLuint SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
GLuint depthMapFBO;
glGenFramebuffers(1, &depthMapFBO);
// - Create depth texture
GLuint depthMap;
glGenTextures(1, &depthMap);
glBindTexture(GL_TEXTURE_2D, depthMap);

glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depthMap, 0);
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

</td>
<td>为渲染的深度贴图创建一个帧缓冲对象。然后，创建一个 2D 纹理，提供给帧缓冲的深度缓冲使用。接着，把生成的深度纹理作为帧缓冲的深度缓冲。最后设置相关渲染参数，为生成深度贴图做准备。</td>
</tr>
<tr>
<td>

```
// - now render scene from light's point of view
simpleDepthShader.use();
simpleDepthShader.bindMat4("lightSpaceMatrix", lightSpaceMatrix);

glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
    glClear(GL_DEPTH_BUFFER_BIT);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, woodTexture);
    glCullFace(GL_FRONT);
    renderScene(simpleDepthShader);
    glCullFace(GL_BACK); // 不要忘记设回原先的culling face
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

</td>
<td>渲染深度贴图。使用深度着色器进行场景的渲染。这里需要初始化视窗的大小，因为阴影部分一般比原有的视窗大小要更大。然后设置纹理属性，将深度贴图渲染出来。</td>
</tr>
</table>

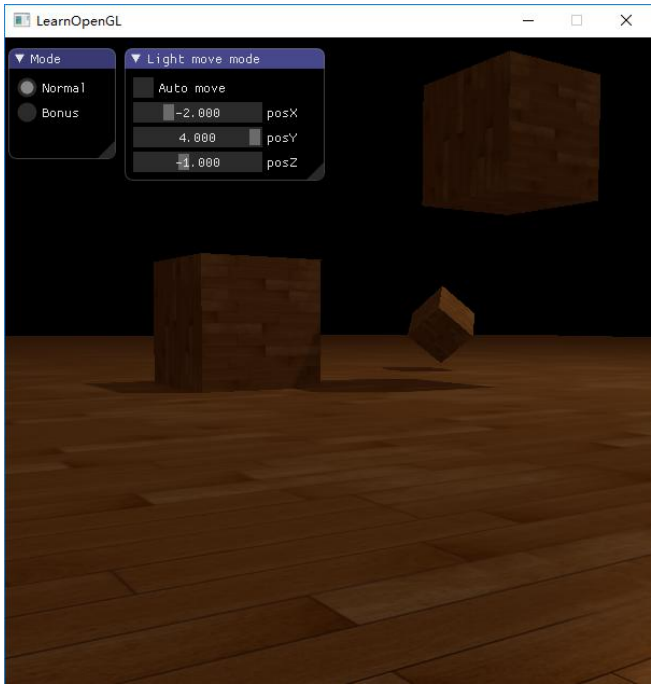
`` // 2. render scene as normal using the generated depth/shadow map // -- glViewport(0, 0, WINDOW_WIDTH, WINDOW_HEIGHT); glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); normalShader.use(); glm::mat4 projection = glm::perspective(glm::radians(camera.getZoom()), (float)WINDOW_WIDTH / (float)WINDOW_HEIGHT, 0.1f, 100.0f); glm::mat4 view = camera.getView(); glm::vec3 camPos = camera.getPosition(); normalShader.bindMat4("projection", projection); normalShader.bindMat4("view", view); // set light uniforms normalShader.bindVec3("viewPos", camPos); normalShader.bindVec3("lightPos", lightPos); normalShader.bindMat4("lightSpaceMatrix", lightSpaceMatrix); glActiveTexture(GL_TEXTURE0); glBindTexture(GL_TEXTURE_2D, woodTexture); glActiveTexture(GL_TEXTURE1); glBindTexture(GL_TEXTURE_2D, depthMap); renderScene(normalShader); ```	渲染一般场景贴图。使用常规着色器进行场景的渲染。这里首先要初始化视窗（viewport）的参数以适应阴影贴图的尺寸，否则将可能导致阴影部分显示不完整。
`` void renderScene(Shader &shader) { // floor glm::mat4 model = glm::mat4(1.0f); shader.bindMat4("model", model); glBindVertexArray(planeVAO); glDrawArrays(GL_TRIANGLES, 0, 6); // cubes model = glm::mat4(1.0f); model = glm::translate(model, glm::vec3(0.0f, 1.5f, 0.0)); model = glm::scale(model, glm::vec3(0.5f)); shader.bindMat4("model", model); renderCube(); model = glm::mat4(1.0f); model = glm::translate(model, glm::vec3(2.0f, 0.0f, 1.0)); model = glm::scale(model, glm::vec3(0.5f)); shader.bindMat4("model", model); renderCube(); model = glm::mat4(1.0f); model = glm::translate(model, glm::vec3(-1.0f, 0.0f, 2.0)); model = glm::rotate(model, glm::radians(60.0f), glm::normalize(glm::vec3(1.0, 0.0, 1.0))); model = glm::scale(model, glm::vec3(0.25)); shader.bindMat4("model", model); renderCube(); } ```	对待渲染的平面和立方体进行变换操作：平面不进行变换操作，三个立方体分别进行平移，旋转，缩放等变换操作。
`` unsigned int loadTexture(char const * path) { unsigned int textureID; glGenTextures(1, &textureID); int width, height, nrComponents; unsigned char *data = stbi_load(path, &width, &height, &nrComponents, 0); if (data) { GLenum format; if (nrComponents == 1) format = GL_RED; else if (nrComponents == 3) format = GL_RGB; else if (nrComponents == 4) format = GL_RGBA; glBindTexture(GL_TEXTURE_2D, textureID); glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format, GL_UNSIGNED_BYTE, data); glGenerateMipmap(GL_TEXTURE_2D); glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, format == GL_RGBA ? GL_CLAMP_TO_EDGE : GL_REPEAT); glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, format == GL_RGBA ? GL_CLAMP_TO_EDGE : GL_REPEAT); glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR); glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR); stbi_image_free(data); } else { std::cout << "Texture failed to load at path: " << path << std::endl; stbi_image_free(data); } return textureID; } ```	读取贴图并生成纹理。

第二部分：实验截图分析

（1）基本要求

A. 实现方向光源的 Shadowing Mapping：要求场景中至少有一个 object 和一块平面(用于显示 shadow)；光源的投影方式任选其一即可；解释 Shadowing Mapping 算法。

1) 效果图：



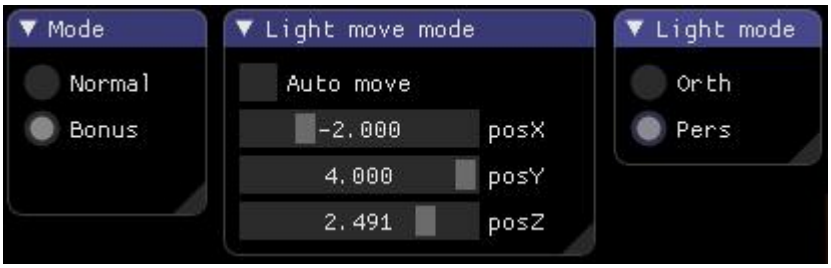
2) 解释 Shadowing Mapping 算法：

Shadow mapping 算法的思路是以光的位置为视角进行渲染，在此视角下能直接看到的平面都被认为是可以被光照亮的平面，不可见的平面被认为是在阴影之中。我们可以把光线理解成射线，光照射在空间内的一个点，相当于于射线第一次穿过那个点。然后用这个最近点和射线上其他点进行对比。然后看看这个点是否比射线上的最近点更远，如果是的话，这个点就在阴影中。这种方式运算速度太慢，在实际操作中，通常使用深度缓冲进行计算。具体方法如下：从光源的透视图来渲染场景，并把深度值的结果储存到纹理中。通过这种方式，就能对光源的透视图所见的最近的深度值进行采样。最终会显示从光源的透视图下见到的第一个片元了。而储存在纹理中的所有这些深度值也被称为深度贴图或阴影贴图。具体实现代码看第一部分的代码分析。

B. 修改 GUI

1) 一般情况

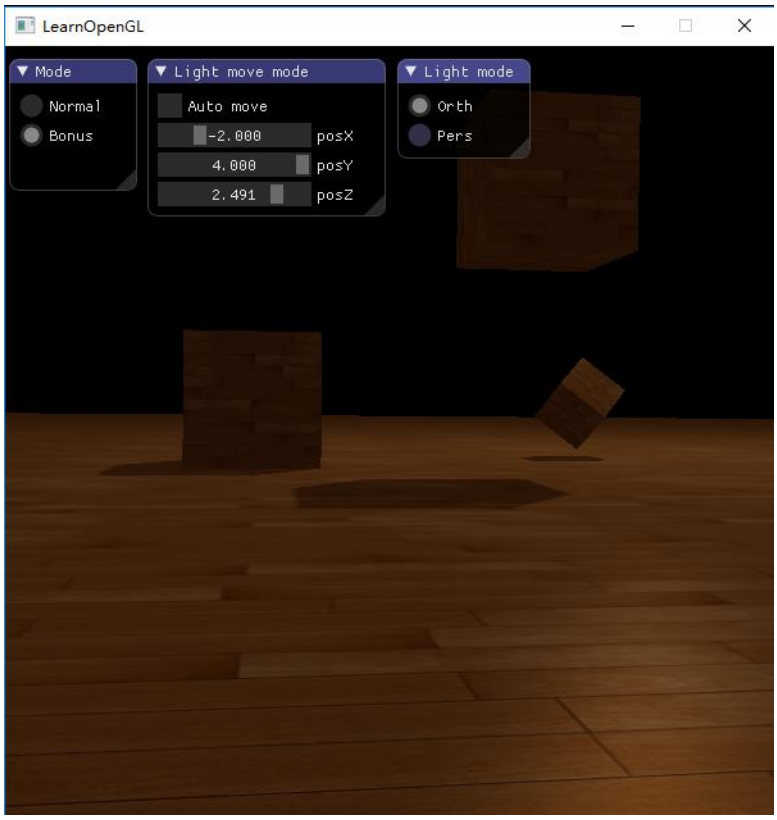
2) Bonus



（2）Bonus

A. 实现光源在正交/透视两种投影下的 Shadowing Mapping

1) 正交投影



2) 透视投影

B. 优化 Shadowing Mapping

1) 优化阴影失真

实现代码：

```
// calculate bias (based on depth map resolution and slope)\n\
vec3 normal = normalize(fs_in.Normal);\n\
vec3 lightDir = normalize(lightPos - fs_in.FragPos);\n\
float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);\n\
// check whether current frag pos is in shadow\n\
// float shadow = currentDepth - bias > closestDepth  ? 1.0 : 0.0;\n\
```

代码说明：使用阴影偏移技术，对表面的深度应用一个偏移量，使得所有采样点都获得了比表面深度更小的深度值，这样整个表面就正确地被照亮，没有任何阴影。

2) 优化悬浮失真

实现代码：

```
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
    glClear(GL_DEPTH_BUFFER_BIT);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, woodTexture);
    glCullFace(GL_FRONT);
    renderScene(simpleDepthShader);
    glCullFace(GL_BACK); // 不要忘记设回原先的culling face
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

代码说明：使用阴影偏移技术后，有可能导致因阴影相对实际物体位置的偏移过大而出现的失真。可以使用当渲染深度贴图时候使用正面剔除这一技巧进行解决。

3) 优化采样过多

实现代码：

```
// keep the shadow at 0.0 when outside the far_plane region of the light's frustum.\n\
if (projCoords.z > 1.0)\n\
    shadow = 0.0;\n\
```

代码说明：在实现 shadow mapping 的过程中，光的视锥不可见的区域一律被认为是处于阴影中，不管它真的处于阴影之中。因为超出光的视锥的投影坐标比 1.0 大，这样采样的深度纹理就会超出他默认的 0 到 1 的范围。解决方法是只要投影向量的 z 坐标大于 1，我们就把 shadow 的值强制设为 0。

4) 阴影锯齿化消除：PCF 技术

实现代码：

```
// PCF\n\
float shadow = 0.0; \n\
vec2 texelSize = 1.0 / textureSize(shadowMap, 0);\n\
for (int x = -1; x <= 1; ++x)\n\
{\n\
    for (int y = -1; y <= 1; ++y)\n\
    {\n\
        float pcfDepth = texture(shadowMap, projCoords.xy + vec2(x, y) * texelSize).r;\n\
        shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;\n\
    }\n\
}\n\
shadow /= 9.0;\n\
```

代码说明：从深度贴图中多次采样，每一次采样的纹理坐标都稍有不同。每个独立的样本可能在也可能不再阴影中。所有的次生结果接着结合在一起，进行平均化，从而得到柔和阴影。