Project report for CG 100433 course

Project Title

基于OpenGL的同济大学四平校区的三维建模与浏览

Team member

学号	姓名
1752204	李宛霖（组长）
1852839	李培然
1853702	刘吉宇
1852410	刘卓奇
1856005	吴伟登
1854127	周家旋

Abstract

Motivation

经过一段时间的线上学习，我们更加怀念校园生活。

The Goal of the project

希望能对同济大学四平校区主要建筑和路面实现建模、贴图导入、渲染以及可自由移动的摄像机。

Involved CG techniques

模型加载、纹理、坐标变换、正交投影、正视投影、光照、阴影等。

Motivation

在疫情隔离在家的一学期之后，我们感受到了在学校——同济大学学习与生活的弥足珍贵。同时，又由于本学期学习了计算机图形学的知识，于是我们便想要在计算机中运用计算机图形学的知识去对学校的部分建筑场景等进行建模复现，并且自由浏览。

The Goal of the project

1. 模型：
    1. 完成对四平校区主要建筑的建模。
    2. 完成对四平校区主要路面的建模。
    3. 完成对四平校区主要建筑的纹理实现。
2. 渲染：
    1. 整合所有建筑及道路的模型。
    2. 完成对建筑的光照和阴影等的渲染。
    3. 完成多自由度可移动的第三人称视角的摄像机的实现。

（注：我们本来打算对嘉定校区进行建模渲染，但是由于嘉定校区在网上的数据信息过少，且寻找第三方建模花费过高（几千元），于是我们选择了与建筑学院的同学合作，利用他们已有的小部分同济大学四平校区的模型，在其基础上继续建模，于是我们的目标从嘉定校区换为了四平校区）

The Scope of the project

1. 不对一部分建筑进行建模。
2. 不做精细的建筑室内的场景，只对室外的场景进行建模与渲染。
3. 不对室内外溢的光源和路灯光源进行实现，只实现室外自然光。
4. 原则上假设路面上没有车辆和行人（项目富余时间足够的话会为路面增加车辆和行人）。
5. 对于建筑建模精度会略低，位置关系只能尽量还原。
6. 不考虑自然天气变化。

Related CG techniques

1. 与建筑等物体的建模、添加纹理或着色相关的技术：
    1. 建模：
        1. 图元的输出（primitive）。
        2. 中点圆算法（Midpoint rasterizer）。
    2. 添加纹理：
        1. 纹理映射。
        2. 凹凸映射（Bump-mapping）。
    3. 着色：

       RGB颜色模型。
2. 导入模型：
    1. 半边数据结构（Half-edge data structure）。
3. 整合模型：

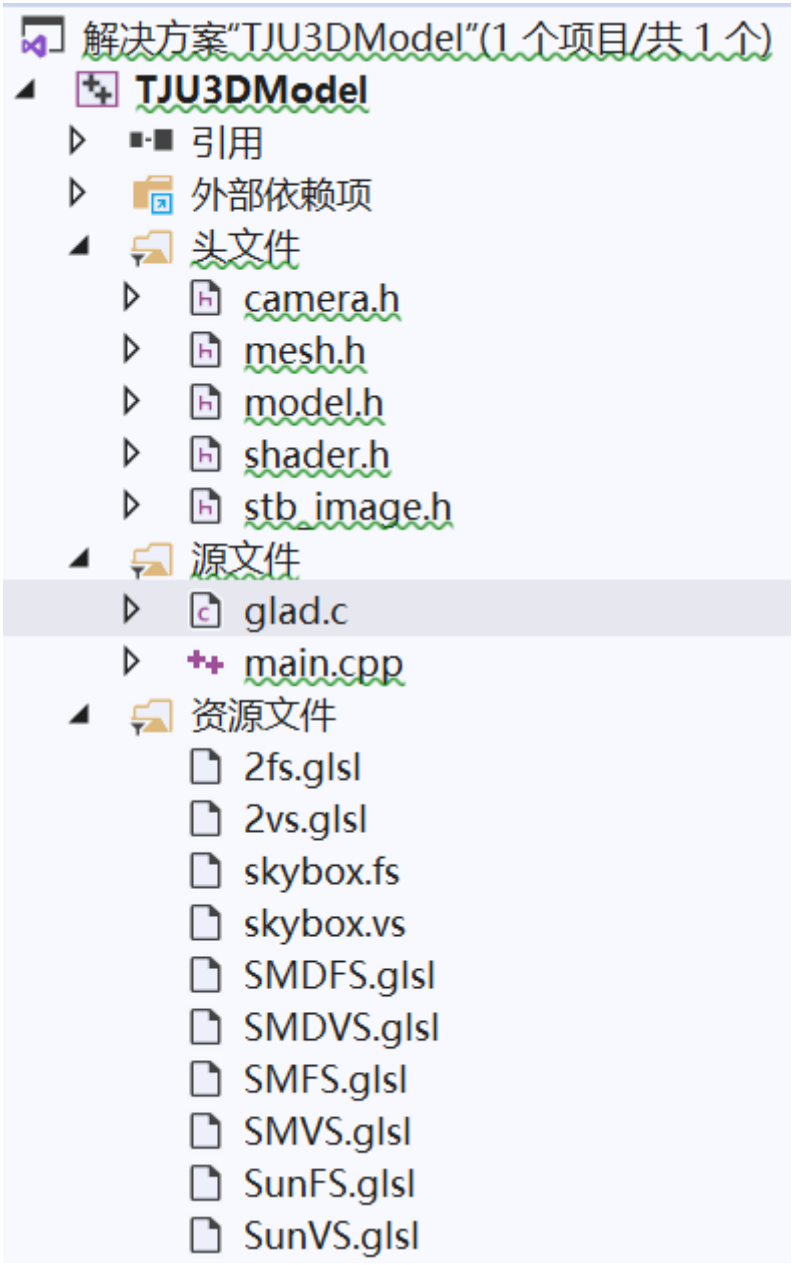
   三维矩阵变换（3D matrix transformation）。
4. 摄像机的实现：
    1. 正视投影（orthographic projection）。
    2. 透视投影（perspective projection）。
5. 渲染：
    1. Blinn-Phong模型。
    2. Phong表面绘制（Phong surface rendering）。
    3. 阴影映射（Shadow Mapping）：
        1. 深度贴图（depth map）。
        2. 阴影偏移（shadow bias）。
        3. 正面剔除（front face culling）。
        4. PCF（percentage-closer filtering）。
    4. 可见面判别：

       深度测试（depth test）：深度缓存算法（depth buffer method，z-buffer method）。
    5. Gamma校正（Gamma Correction）。
    6. 抗锯齿（anti-aliasing）。

Project contents

1. 实现了对四平校区主要建筑场景的建模。
2. 实现了一个多自由度可移动的第三人称视角的摄像机，可以自由浏览校园景色。
3. 实现了对四平校区主要建筑场景的渲染。

项目文件结构如下：



Implementation

项目文档见文件夹中的TJU3DModel.chm，它是由doxygen+graphviz生成的，内含各文件、类、函数、变量的成员和调用关系等。

我们的实现主要分为以下三个方面：

9.1 建模与贴图

9.1.1 原定计划的更改

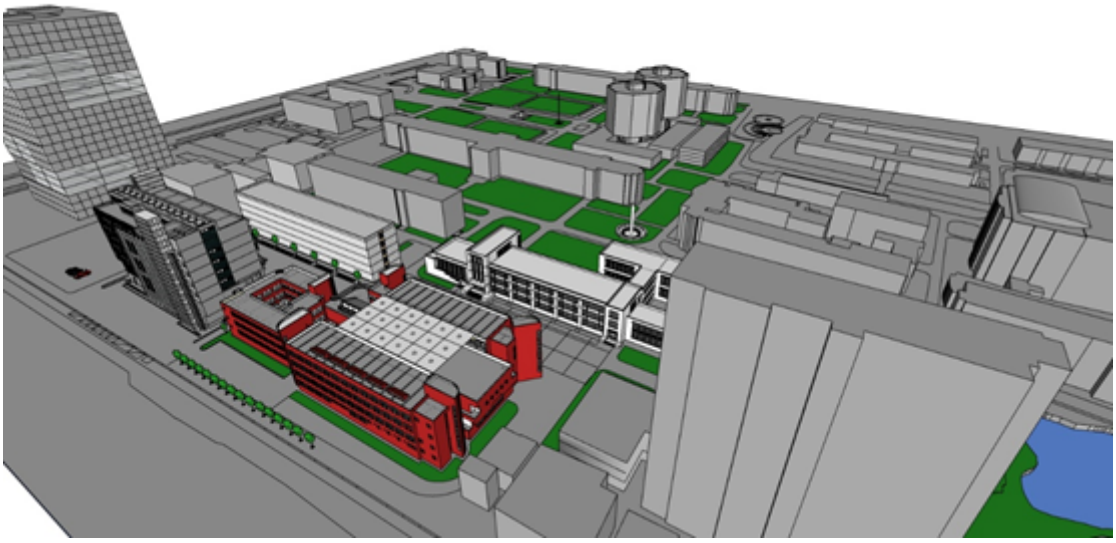
在项目计划报告中，原定是对同济大学嘉定校区的主要区域进行建模，但是由于计划区域范围较大，范围内的建筑、绿地等相关数据难以通过各种途径进行获取，初期建模过程仅停留在对仰望星空及友园7、8、9、10号楼的建模阶段。

随着课程的深入，项目需要进一步得到推进，但建筑群的数据无法获取的问题致使模型比例无法得到统一。此外，原定的建模软件3Ds MAX虽然功能强大，但要在短短几周内掌握其功能且建立完善且精美的模型是极具挑战性的，所以我们在之后尝试过使用Blender和寻找建模工作室帮助等方式。但最终由于Blender插件环境配置和工作室费用较昂贵等原因放弃以上途径。

最终，我们在赵君峤老师的建议以及建筑与城市规划学院同学的帮助下，决定基于SketchUp平台，对同济大学四平路校区的主要区域进行建模。SketchUp平台功能强大且易于初学者上手，模型的各个参数简洁明了，易于多位合作者间工程交接，非常适合我们的建模项目。确定好建模平台后，我们联系了建筑与城市规划学院的同学，他们向我们提供了基于SketchUp的同济大学建筑与城市规划学院建筑群的模型，并在建筑绿地等比例和尺寸方面向我们提供了相关建议，我们在城市规划学院所在区域的基础上向外延展，对同济大学四平路校区的校前区部分进行了建模。

9.1.2 模型的创建

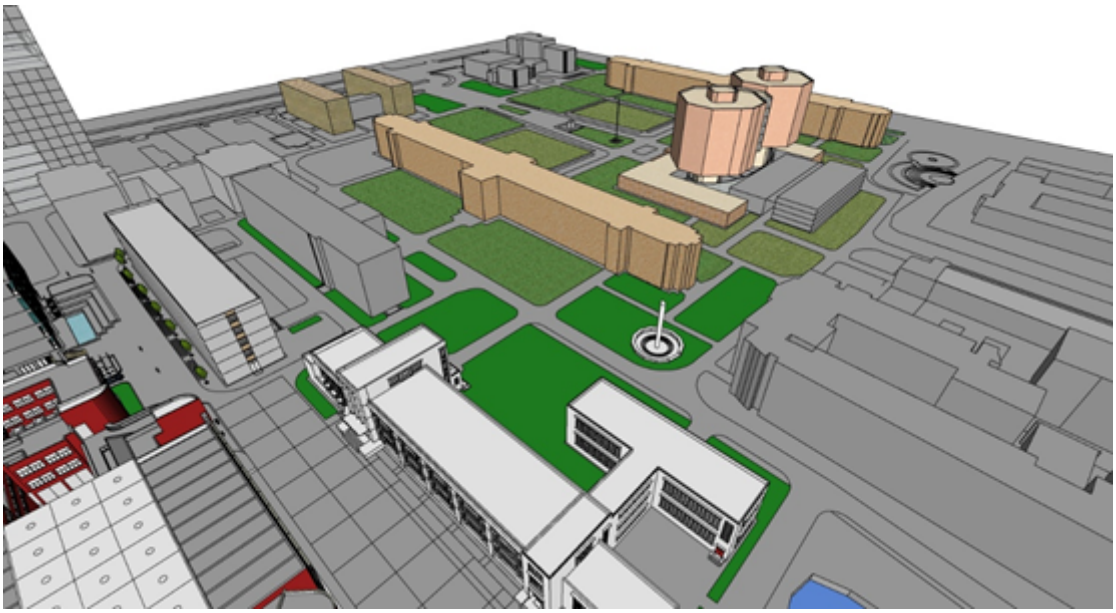
我们团队重新对建模区域进行了圈定和分配，新选定的范围是衷和楼-三好坞-大学生活动中心-南楼-中法中心等近似正方形的区域中。建筑及其他对象的建模和纹理贴图工作主要是由刘卓奇、周家旋和李宛霖负责。建筑物具有的形状都是基本的几何图形，可以直接在SketchUp中添加顶点、边和面，并对它们的位置等相关属性进行调整，以获得物理模型上建筑物的特征。



在提供的模型基础上向外延展对同济大学四平路校区的校前区部分建模初期

9.1.3 模型的纹理优化

在对建筑的外部形状进行建模之后，我们通过添加纹理贴图对模型进行了进一步的优化。纹理贴图的素材来源主要来自于SketchUp的纹理素材库以及网络上专业纹理网站上的纹理素材。建筑之外的其他对象（如绿地、汽车等）可以从网络直接导入相关素材。最后，各个模型会根据它们的实际位置，在三维地图中将它们组合在一起。



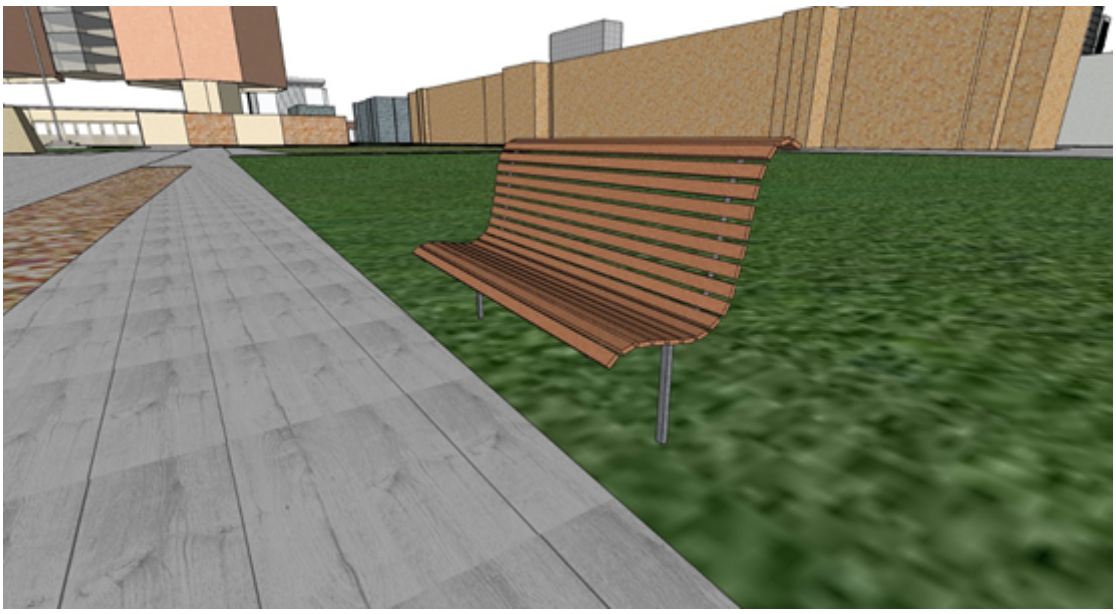
模型的贴图阶段（一）

模型的贴图阶段（二）

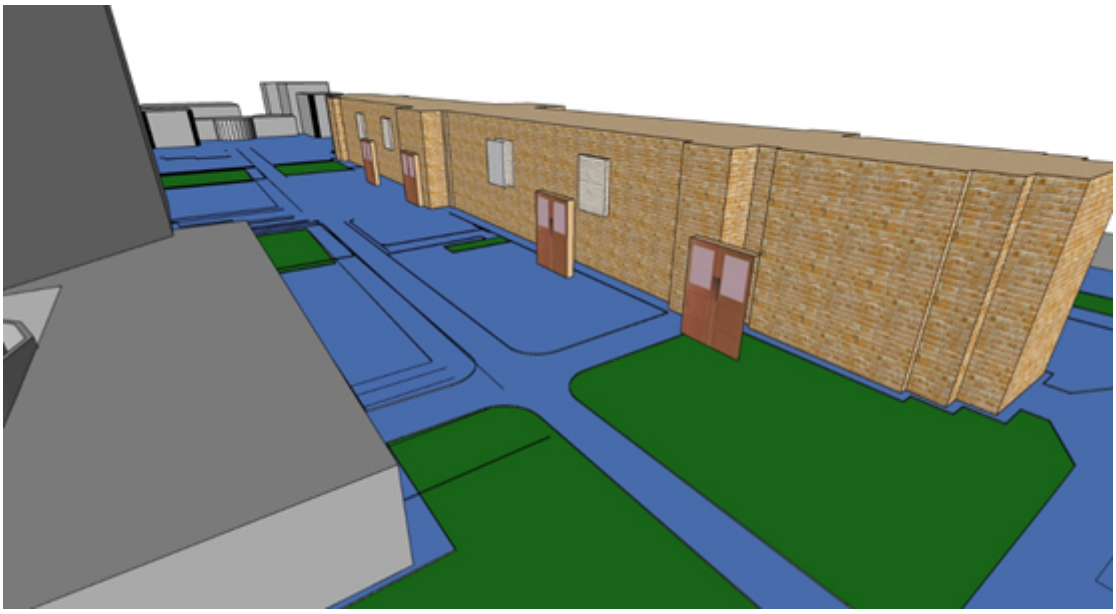
9.1.4 模型的特征优化尝试

在建模阶段的后期，我们试图对模型进行进一步的优化，具体的想法是将模型与纹理贴图进一步融合，为建筑添加门窗及其他细节处理，但由于模型范围较大，“牵一发而动全身”的问题持续出现，改动某一部分就会影响到其他部分模型的完整性，且遇到了例如Z-Fighting的渲染问题。基于时间和进度的考虑，这一部分的探究最终没有在成果中体现。

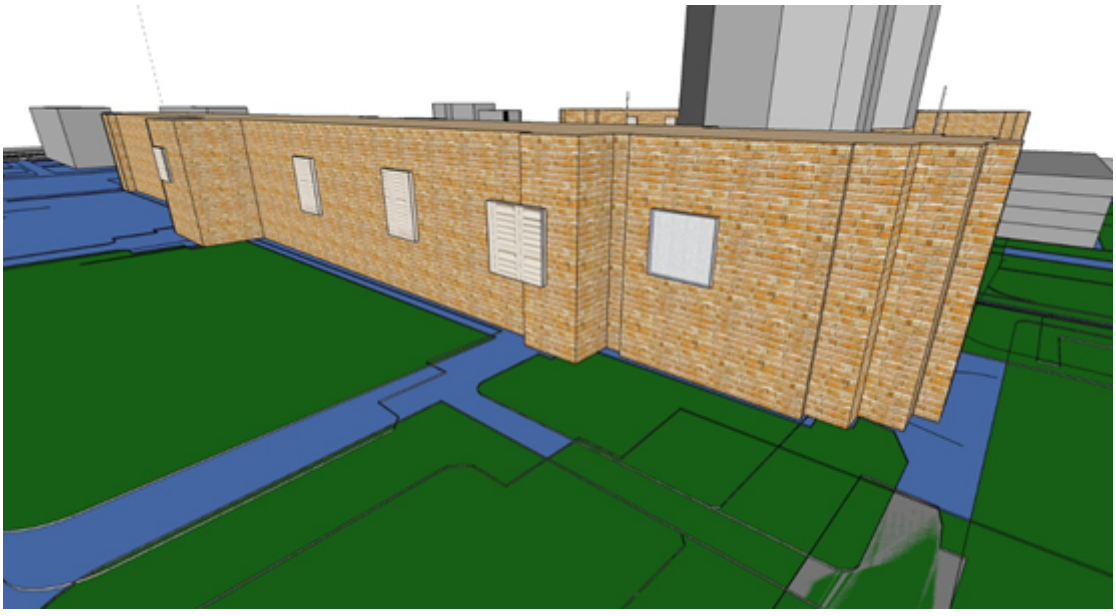
模型的特征优化尝试（图书馆大门贴图）



模型的特征优化尝试（校前区休闲长椅）



模型的特征优化尝试（南北楼大门）



模型的特征优化尝试（南北楼窗户）

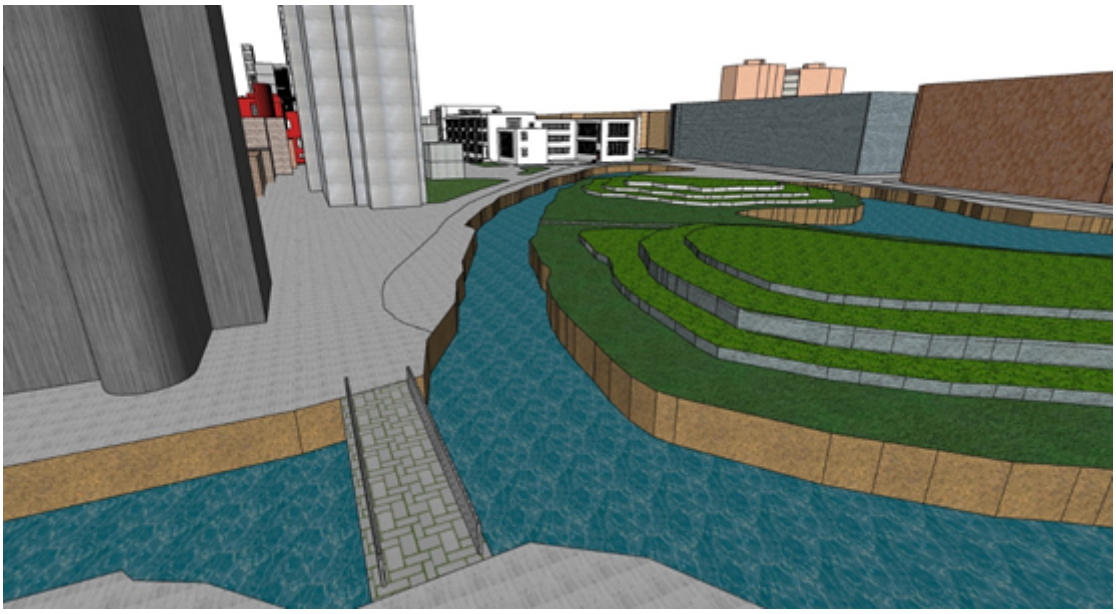
9.1.5 模型的导出

最后使用SketchUp进行模型导出。所导出的模型文件将自动包含所有生成的顶点、它们的坐标、纹理坐标和其他辅助信息。之后就可以将地图模型导入OPENGL中进行后续处理。

模型全景

模型局部（一）

模型局部（二）



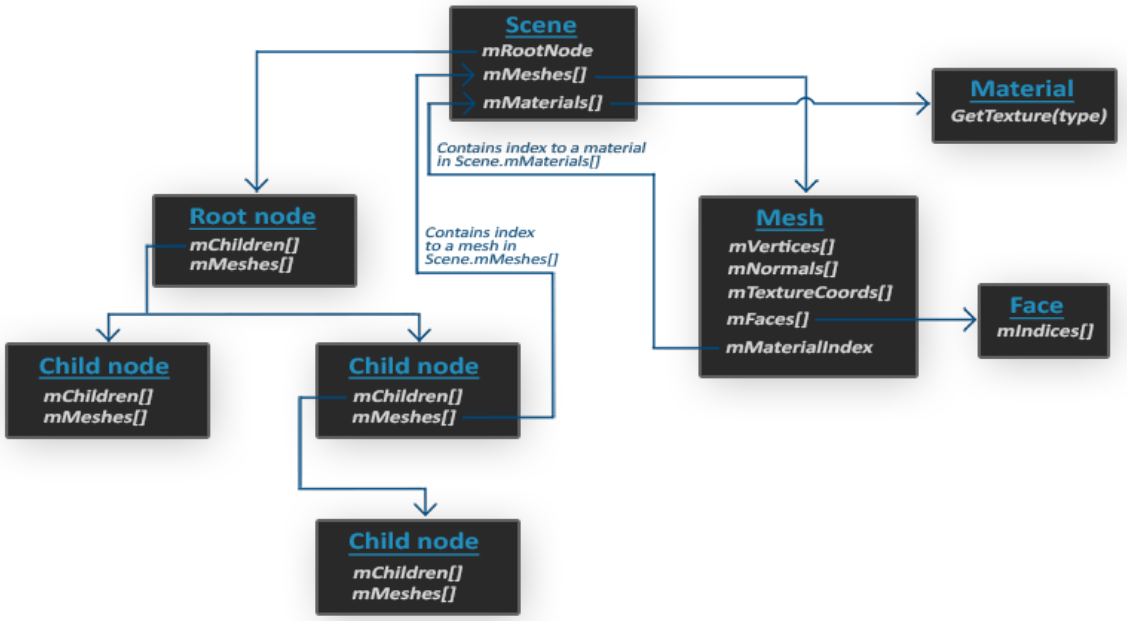
模型局部（三）

9.2 模型导入与天空盒

9.2.1 模型导入

在OpenGL中手动构建模型是一个非常抽象且不简单的任务，因此想要构建复杂多彩的世界，我们通常会在其他可视化建模软件中完成模型的构建、贴图等工作并将其保存为一个合适的格式，再使用一些额外的代码将这些建立好的模型数据处理成为能被OpenGL识别处理的数据，将其导入至OpenGL的世界并显示。我们使用的模型导入库是Assimp库，它能够帮助我们解析obj文件的文件内容，之后我们就能够从Assimp的数据结构中提取我们所需的数据了。

使用Assimp导入一个模型的时候，它通常会将整个模型加载进一个Scene中，它会包含导入模型的所有数据，Assimp的数据结构模型可见下图：



reference [learnopengl.com]

- 场景的Root node（根节点）可能包含子节点（和其它的节点一样），它会有一系列指向场景对象中mMeshes数组中储存的网格数据的索引。Scene下的mMeshes数组储存了真正的Mesh对象，节点中的mMeshes数组保存的只是场景中网格数组的索引。
- 一个Mesh对象本身包含了渲染所需要的所有相关数据，像是顶点位置、法向量、纹理坐标、面(Face)和物体的材质。
- 一个Mesh包含了多个面。Face代表的是物体的渲染图元(Primitive)（三角形、方形、点）。一个面包含了组成图元的顶点的索引。由于顶点和索引是分开的，使用一个索引缓冲来渲染是非常简单的。
- 最后，一个Mesh也包含了一个Material对象，它包含了一些函数能让我们获取物体的材质属性，比如说颜色和纹理贴图（比如漫反射和镜面光贴图）。

所以我们需要做的是就是将物体加载到Scene对象中，遍历节点获取对应的Mesh对象，并处理每个Mesh对象来获取顶点数据，索引以及其材质属性。

```
void loadModel(string const& path)
    {
        // read file via ASSIMP
        Assimp::Importer importer;
        const aiScene* scene = importer.ReadFile(path, aiProcess_Triangulate | aiProcess_GenSmoothNormals |
                                                       aiProcess_FlipUVs | aiProcess_CalcTangentSpace);

        // check for errors
        if (!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode) // if is Not Zero
        {
            cout << "ERROR::ASSIMP:: " << importer.GetErrorString() << endl;
            return;
        }
        // retrieve the directory path of the filepath
        directory = path.substr(0, path.find_last_of('/'));
        // process ASSIMP's root node recursively
        processNode(scene->mRootNode, scene);
    }
void processNode(aiNode* node, const aiScene* scene)
    {
        // process each mesh located at the current node
        for (unsigned int i = 0; i < node->mNumMeshes; i++)
        {

            aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
            meshes.push_back(processMesh(mesh, scene));
        }

        for (unsigned int i = 0; i < node->mNumChildren; i++)
        {
            processNode(node->mChildren[i], scene);
        }

    }
```

在获取到一大堆的网格数据后，在将它们用于渲染之前，需要配置好正确的缓冲，也就是通过顶点属性指针定义好定点着色器的布局：

```
        glEnableVertexAttribArray(0);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);
        // vertex normals
        glEnableVertexAttribArray(1);
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, Normal));
        // vertex texture coords
        glEnableVertexAttribArray(2);
        glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, TexCoords));
        // vertex tangent
        glEnableVertexAttribArray(3);
        glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, Tangent));
        // vertex bitangent
        glEnableVertexAttribArray(4);
        glVertexAttribPointer(4, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, Bitangent));
```

为了成功在OpenGL中绘制出图形，我们需要为Mesh类定义最后一个函数，Draw函数，在本模型中，因为使用sktchup导出的模型缺失了法线等光照计算的参数，因此我们在shader中固定了物体在光照下的反射系数，简化了渲染步骤，仅使用了以下代码进行图像的绘制：

```
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
glBindVertexArray(0);
```

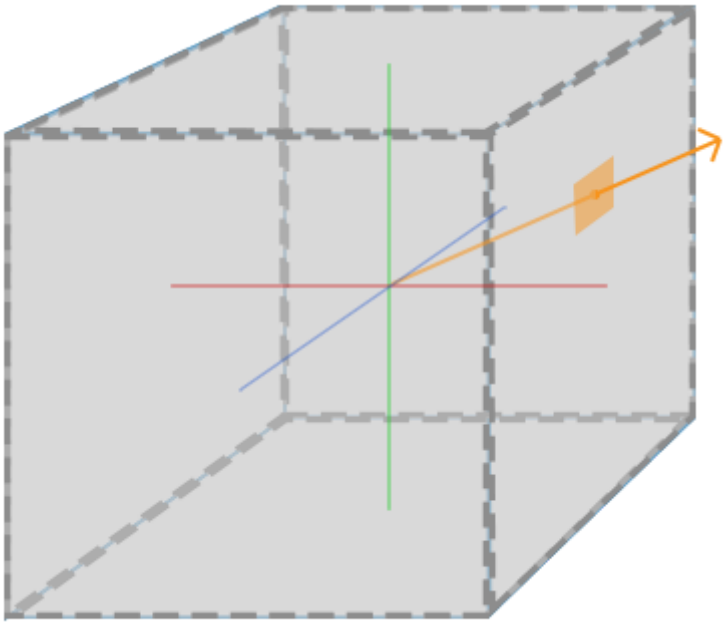
至此，我们就可以在OpenGL中绘制出我们已经在其他建模软件中建立好的模型。值得注意的是，在OpenGL中，为了让我们的图形看起来真实，一般都会采用深度测试，而对于较为大型的模型而言，其在建模软件中的尺寸可能会巨大无比，将其导入进OpenGL后，在较远的部分可能会引起Z-Fighting，从而导致闪烁现象，要想解决这一问题，要么在绘制前对模型进行缩放，要么需要调整透视空间的平截头体的大小，避免远处的模型产生闪烁现象。

还有一点值得注意的是，加载模型时难免需要贴图，加载外部贴图时使用的库是`stb_image.h`，不知道为什么这个库是使用C语言写的，对中文的支持非常差，因此在使用时需要注意路径不能存在中文（甚至空格），除此之外，这个库对纹理的加载时常会出现一些莫名奇妙的无法载入的错误，修复这些错误也花费了许多时间。

9.2.2 天空盒

为了让整个背景不那么单调空洞，我们需将给整个模型放进一个"世界"中，这个世界就是天空盒。

在OpenGL中，我们使用立方体贴图的方式，简单来说，立方体贴图就是一个包含了6个2D纹理的纹理，每个2D纹理都组成了立方体的一个面：一个有纹理的立方体。你可能会奇怪，这样一个立方体有什么用途呢？为什么要把6张纹理合并到一张纹理中，而不是直接使用6个单独的纹理呢？立方体贴图有一个非常有用的特性，它可以通过一个方向向量来进行索引/采样。假设我们有一个1x1x1的单位立方体，方向向量的原点位于它的中心。使用一个橘黄色的方向向量来从立方体贴图上采样一个纹理值会像是这样：



reference learnopengl.com

我们可以使用如下的代码来加载一个天空盒，通过这样的方式，我们给这个立方体贴图的每一个面都生成了纹理

```
unsigned int loadCubemap(std::vector<std::string> faces)
{
    unsigned int textureID;
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);

    int width, height, nrComponents;
    for (unsigned int i = 0; i < faces.size(); i++)
    {
        unsigned char* data = stbi_load(faces[i].c_str(), &width, &height, &nrComponents, 0);
        if (data)
        {
            glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
            stbi_image_free(data);
```

```
        }
        else
        {
            std::cout << "Cubemap texture failed to load at path: " << faces[i] << std::endl;
            stbi_image_free(data);
        }
    }
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
    return textureID;
}
```

天空盒的本质是一个立方体贴图，这也就决定了它是绘制在一个立方体上的，与其他物体一样，我们需要VAO，VBO和一组新的顶点来对其进行绘制，用于贴图3D立方体的立方体贴图可以使用立方体的位置作为纹理坐标来采样。当立方体处于原点(0, 0, 0)时，它的每一个位置向量都是从原点出发的方向向量。这个方向向量正是获取立方体上特定位置的纹理值所需要的。正是因为这个，我们只需要提供位置向量而不用纹理坐标了。

绘制天空盒时，需要注意的是，我们希望在这个虚拟的"世界"中，玩家永远走不到边界，并且天空盒应当是作为这个世界的"边界"存在的，没有任何东西可以触及，因此我们在绘制天空盒之前应当修改深度函数，将其从默认的GL_LESS改为GL_LEQUAL，这样深度缓冲就会被填上天空盒的1.0（即标准坐标系下的最远坐标），这样子天空盒就会永远被绘制在其它物体的背后了。使用的代码如下：

```
        glDepthFunc(GL_LEQUAL);
        // change depth function so depth test passes when values are equal to depth buffer's content
        skyboxShader.use();
        view = glm::mat4(glm::mat3(camera.GetViewMatrix())); // remove translation from the view matrix
        skyboxShader.setMat4("view", view);
        skyboxShader.setMat4("projection", projection);
        // skybox cube
        glBindVertexArray(skyboxVAO);
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_CUBE_MAP, cubemapTexture);
        glDrawArrays(GL_TRIANGLES, 0, 36);
        glBindVertexArray(0);
        glDepthFunc(GL_LESS);
```

最后，天空盒的效果如下：

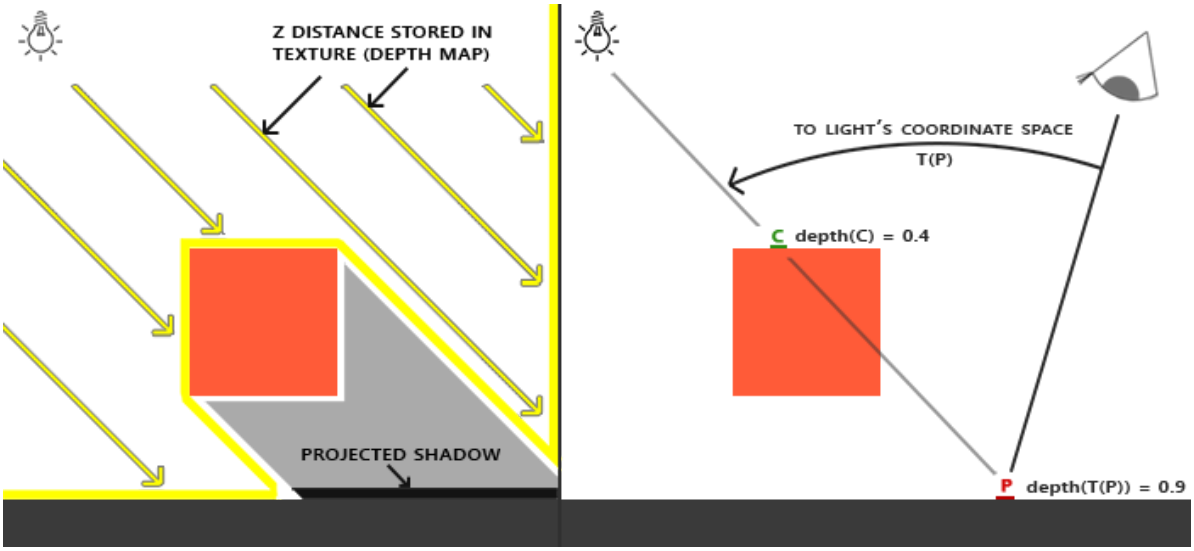
9.3 渲染

9.3.1 阴影

9.3.1.1 阴影贴图

由于只考虑太阳，太阳离建筑无限远，可看作平行光，所以我们采用深度贴图（depth map）来实现。

我们通过使用光的透视图进行场景渲染，把深度值（即深度贴图）存到纹理中。这样，我们可以对光的透视图所见的最近的深度值进行采样。最终，深度值就会显示从光源的透视图下见到的第一个片段了，效果示意图如下。



在c++中的渲染阶段的代码框架如下：

```
// render depth map
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
ConfigureShaderAndMatrices();
RenderScene();
glBindFramebuffer(GL_FRAMEBUFFER, 0);
// render the scene with depth map
glViewport(0, 0, SCR_WIDTH, SCR_HEIGHT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
ConfigureShaderAndMatrices();
glBindTexture(GL_TEXTURE_2D, depthMap);
RenderScene();
```

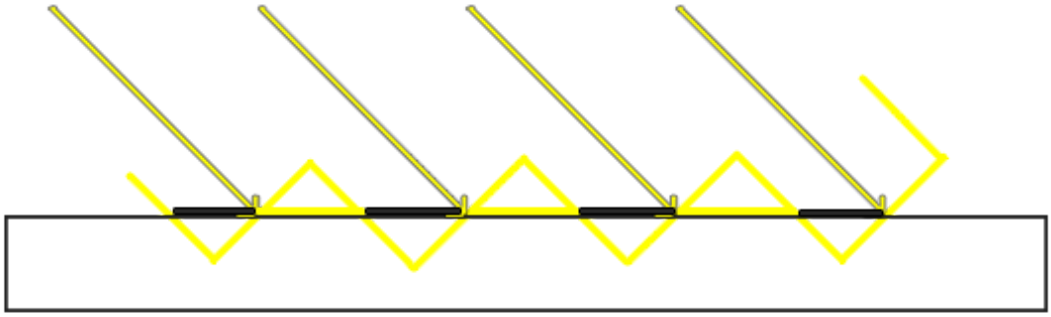
对应的shader的glsl代码如下：

```
float ShadowCalculation(vec4 fragPosLightSpace)
{
    // perform perspective divide
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    // transform to [0,1] range
    projCoords = projCoords * 0.5 + 0.5;
    // get closest depth value from light's perspective (using [0,1] range fragPosLight as coords)
    float closestDepth = texture(shadowMap, projCoords.xy).r;
    // get depth of current fragment from light's perspective
    float currentDepth = projCoords.z;
    // calculate bias (based on depth map resolution and slope)
    vec3 normal = normalize(fs_in.Normal);
    vec3 lightDir = normalize(lightPos - fs_in.FragPos);
    float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);
    // PCF
    float shadow = 0.0;
    vec2 texelSize = 1.0 / textureSize(shadowMap, 0);
    for (int x = -1; x <= 1; ++x)
    {
        for (int y = -1; y <= 1; ++y)
        {
            float pcfDepth = texture(shadowMap, projCoords.xy + vec2(x, y) * texelSize).r;
            shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;
        }
    }
    shadow /= 9.0;

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

    return shadow;
}
```

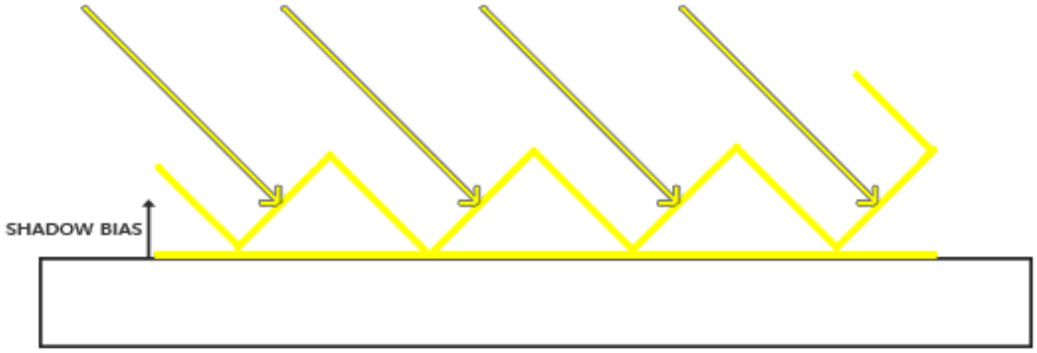
上述代码中还有一些对阴影的优化，具体如下：

9.3.1.2 阴影偏移

阴影贴图受制于分辨率，在光源比较远时，多个片段可能从深度贴图的同一个值中采样。下图每个斜坡代表深度贴图一个单独的纹理像素。可以看到，多个片段从同一个深度值采样。当光以一个角度朝向表面的时候，深度贴图也是从一个角度渲染。多个片段会从同一个斜坡的深度纹理像素中采样，有的在地面上，有的在地面下，因此我们得到的阴影有了差异。所以，有些片段被认为在阴影内，有些不在，由此渲染出很大一块交替黑线，即阴影失真(Shadow Acne)。



我们使用了一个叫阴影偏移（shadow bias）的技巧解决此问题。我们对深度贴图加一个偏移量，这样片段就不会被错误地认为在表面之下了，示意图如下：

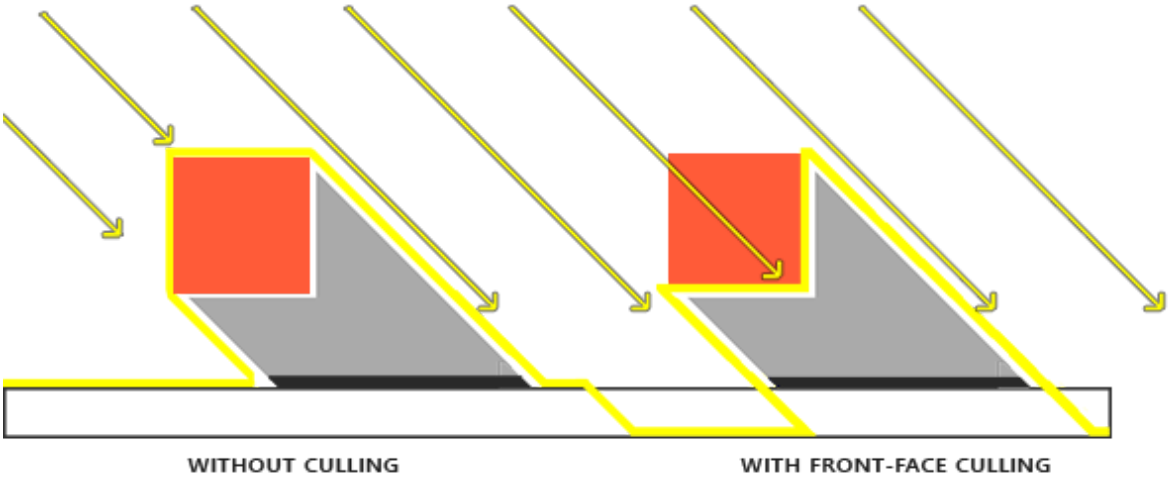


具体的glsl代码实现如下：

```
// calculate bias (based on depth map resolution and slope)
vec3 normal = normalize(fs_in.Normal);
vec3 lightDir = normalize(lightPos - fs_in.FragPos);
float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);
shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;
```

9.3.1.3 正面剔除

当阴影偏移值过大时，可以明显地看出阴影相对实际物体位置的偏移，此时发生了悬浮(Peter Panning)。我们可以使用正面剔除（front face culling）解决大部分的Peter panning问题。因为我们只需要深度贴图的深度值，对于实体物体无论用它们的正面还是背面都没问题。



使用opengl自带的函数即可实现，具体代码如下：

```
glCullFace(GL_FRONT);
RenderScene();
glCullFace(GL_BACK);
```

9.3.1.4 PCF

放大看阴影时，可以发现其对分辨率依赖严重，因为深度贴图有固定的分辨率，多个片段对应于一个纹理像素。这样会造成多个片段从深度贴图的同一个深度值采样，它们得到的是同一个阴影，所以会有锯齿。

解决方案之一叫做PCF（percentage-closer filtering）。它的核心是从深度贴图中多次采样，每一次采样的纹理坐标都不同。每次采样可能在或不在阴影中。所有的结果求平均数便得到了柔和阴影，具体的glsl代码如下：

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

测试模型的渲染阴影效果如下图所示：

9.3.2 Blinn-Phong模型

Blinn-Phong模型是Jim Blinn对Phong反射模型改进的版本。

Blinn-Phong反射模型用 $\vec{r} \cdot \vec{v}$ 代替了Phong反射模型中的，$\vec{n} \cdot \vec{h}$。

$\vec{h}$是观察者和光源向量之间的半程单位向量，可以根据以下公式计算：

$$\vec{h} = \frac{\vec{\ell} + \vec{v}}{\left\|\vec{\ell} + \vec{v}\right\|}$$

最后加上阴影的Blinn-Phong模型的glsl代码如下：

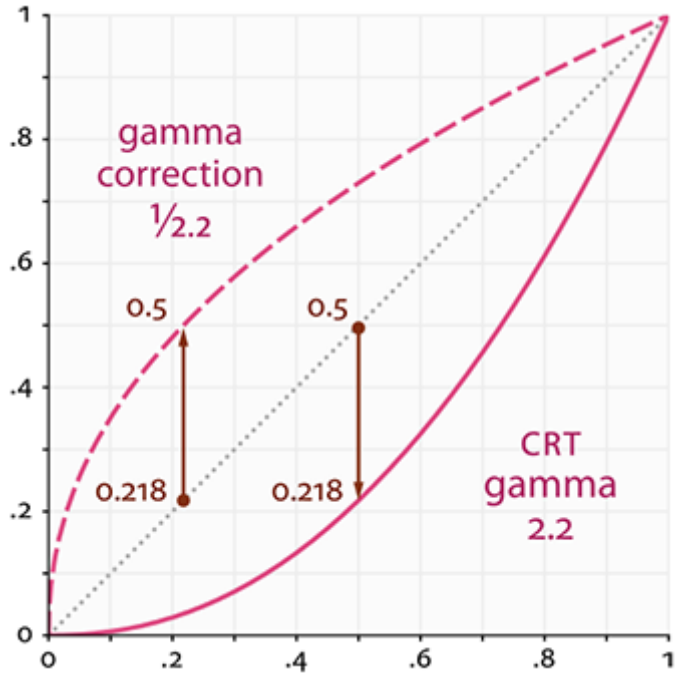
```
void main()
{
    vec3 color = texture(diffuseTexture, fs_in.TexCoords).rgb;
    vec3 normal = normalize(fs_in.Normal);
    vec3 lightColor = vec3(0.5);
    //vec3 lightColor = vec3(0.3);
    // ambient
    vec3 ambient = 0.3 * color;
    // diffuse
    vec3 lightDir = normalize(lightPos - fs_in.FragPos);
    float diff = max(dot(lightDir, normal), 0.0);
    vec3 diffuse = diff * lightColor;
    // specular
    vec3 viewDir = normalize(viewPos - fs_in.FragPos);
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = 0.0;
    vec3 halfwayDir = normalize(lightDir + viewDir);
    spec = pow(max(dot(normal, halfwayDir), 0.0), 64.0);
    vec3 specular = spec * lightColor;
    // calculate shadow
    //float shadow = 0.0;
    float shadow = ShadowCalculation(fs_in.FragPosLightSpace);
    vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) *
color;

    FragColor = vec4(lighting, 1.0);
```

```
26 }
```

9.3.3 Gamma校正

Gamma也叫灰度系数，每种显示设备都有自己的Gamma值，都不相同，有一个公式：设备输出亮度 = 电压的Gamma次幂，任何设备Gamma基本上都不会等于1，此时为理想的线性关系。对于阴极射线管显示器（CRT），它的Gamma通常为2.2。

一个电压与设备输出亮度的关系图如下：



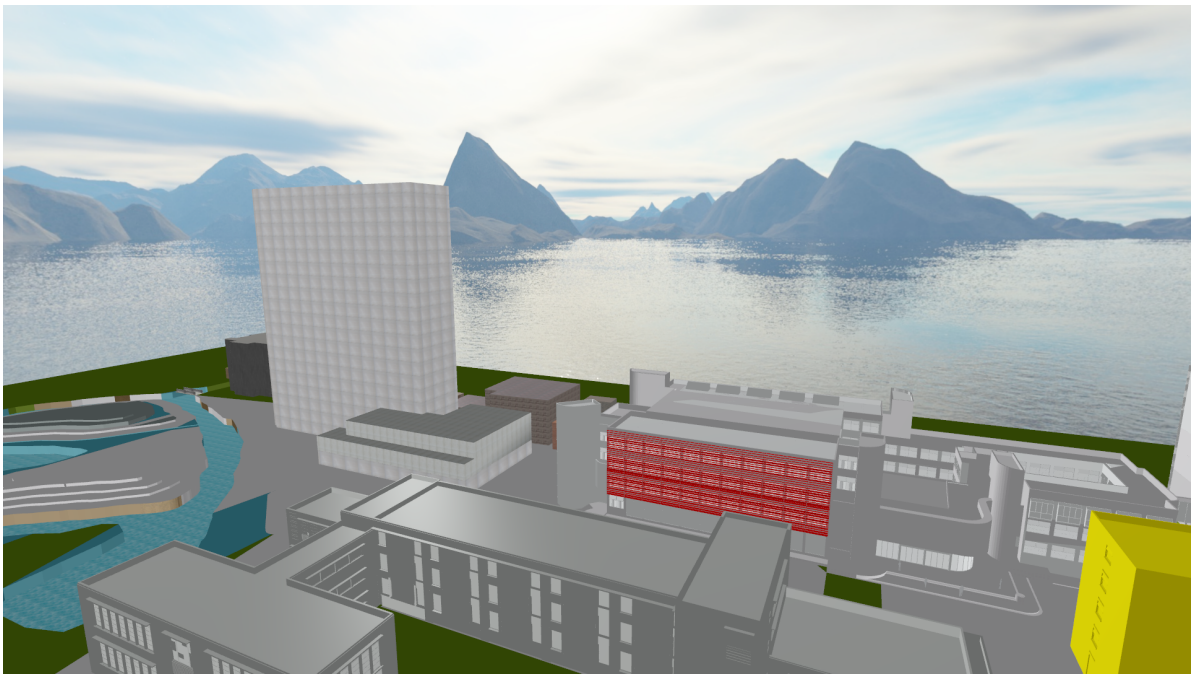
由上如图可知，我们并不能得到真正的亮度，下图便是未经过Gamma校正的图像：

为了使输出的亮度正常，Gamma校正(Gamma Correction)的思路是在最终的颜色输出上应用监视器Gamma的倒数。

我们使用了opengl自带的方法，来使用Gamma校正，具体代码如下：

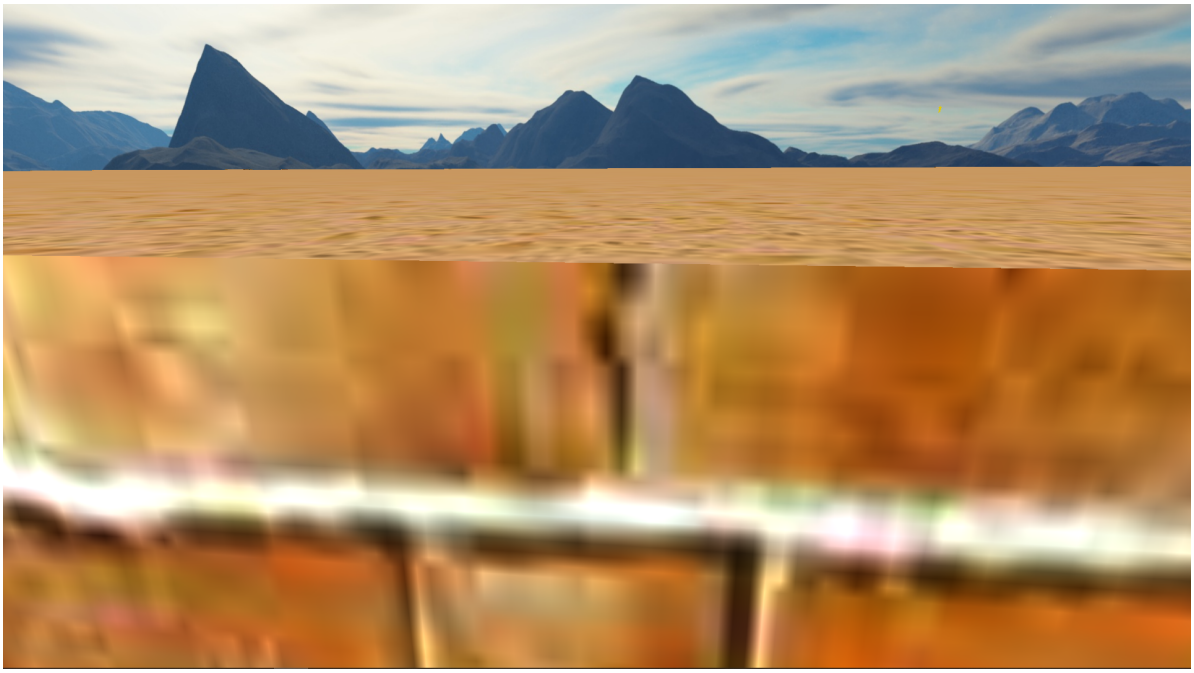
```
1 glEnable(GL_FRAMEBUFFER_SRGB);
```

开启后的效果如下图所示：

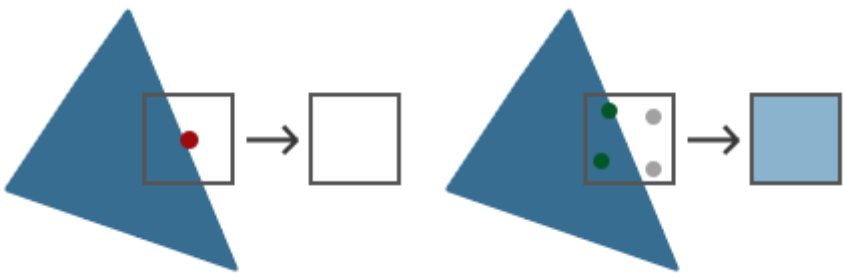


9.3.4 抗锯齿

当靠近我们的一个建筑时，仔细观察可以发现它边缘有锯齿，如下图所示：



我们的解决方法是使用多重采样抗锯齿(Multisample Anti-aliasing, MSAA)。我们使用特定图案排列的4个子采样点(Subsample)来采样，并使用这些子采样点来决定像素的遮盖度，从而达到抗锯齿的目的，如下图所示：



具体的实现我们采用了opengl自带的MSAA，具体代码如下：

```
glfwWindowHint(GLFW_SAMPLES, 4);
// 创建窗口、绑定上下文等
glEnable(GL_MULTISAMPLE);
```

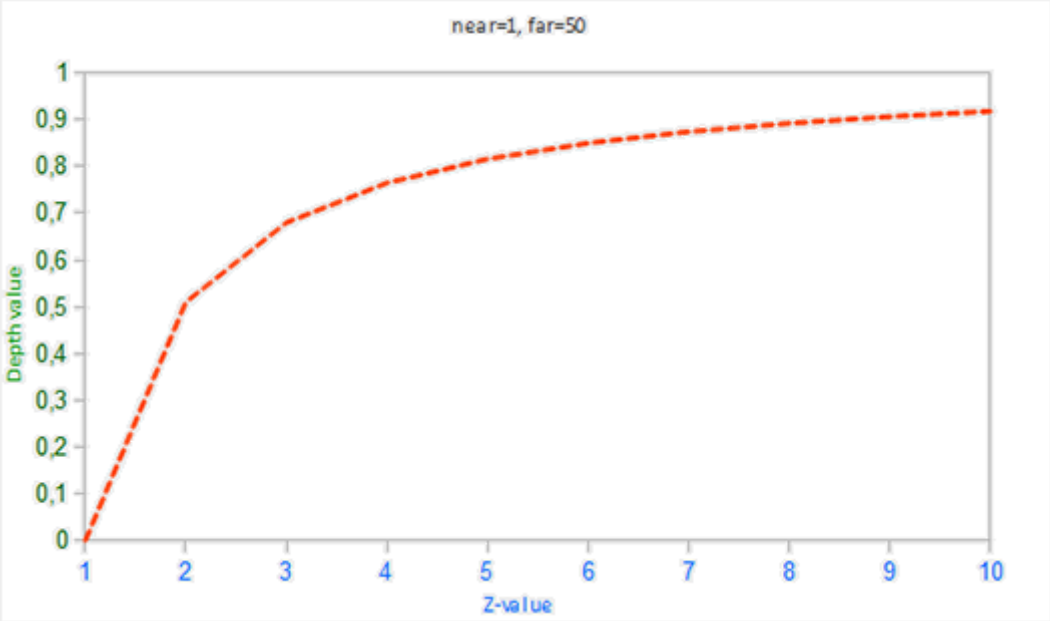
开启抗锯齿后的效果如下图所示：

9.3.5 深度测试与解决深度冲突

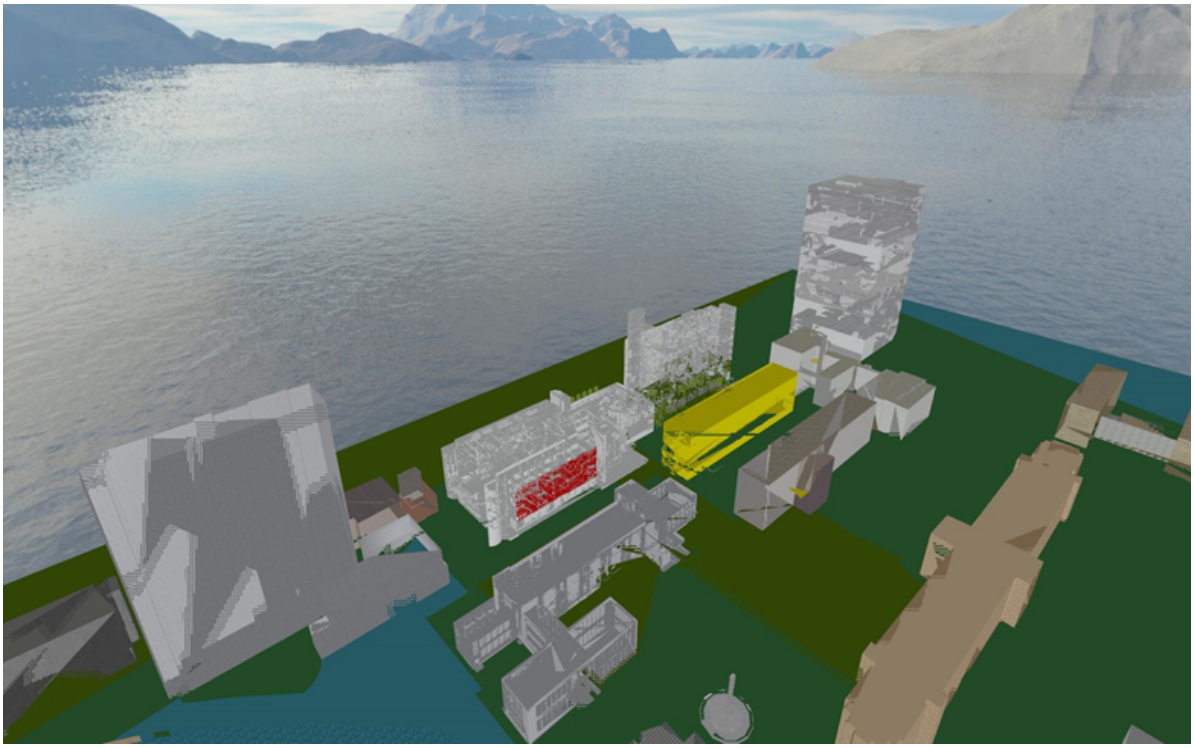
由于实际中的深度缓冲方程如下：

$$F_{depth} = \frac{\frac{1}{z} - \frac{1}{near}}{\frac{1}{far} - \frac{1}{near}}$$

导致了有下图的z与深度值的关系：



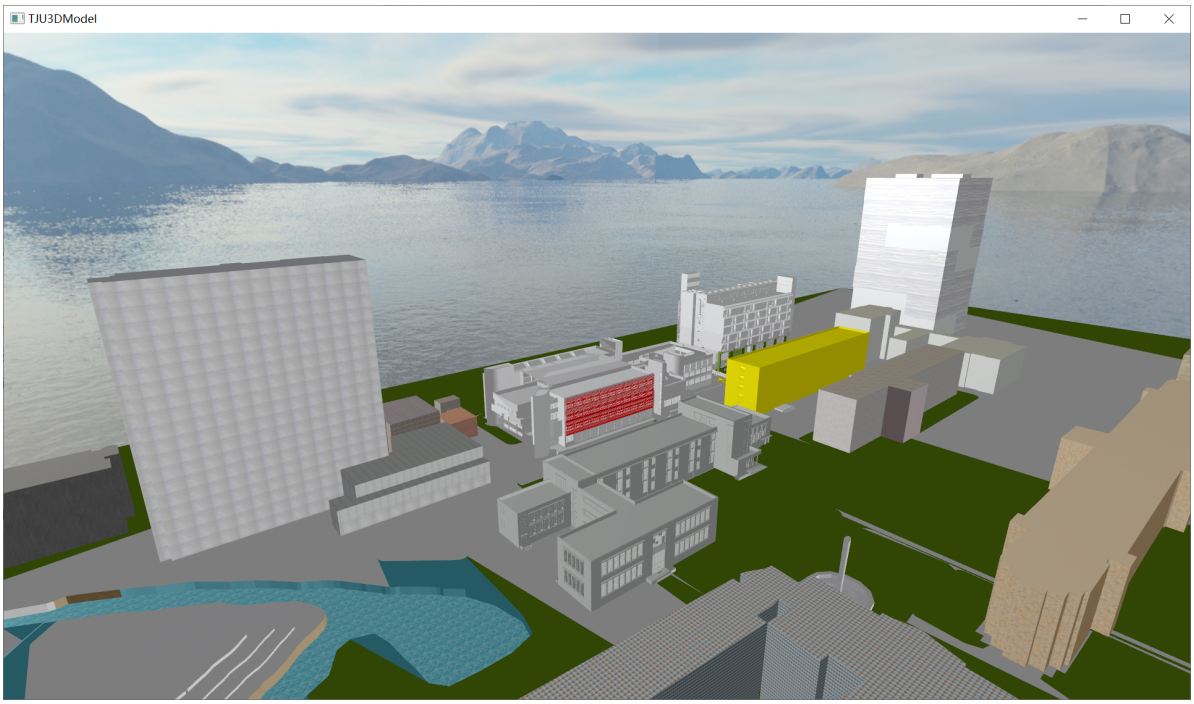
显而易见，它们二者是非线性的，且随着z的增大，深度值的精度显著降低，深度缓冲没有足够的精度来决定附近的形状哪个在前面。结果就是附近的形状不断地在切换前后顺序，会产生奇怪的花纹。这个现象叫做深度冲突(Z-fighting)，如下图所示：



对于我们的模型来说，这个问题是因为模型过大。所以我们的解决方法是将透视投影矩阵的near值从原来的0改成100，具体代码如下：

```
glm::mat4 projection = glm::perspective(camera.Zoom, (float)SCR_WIDTH / (float)SCR_HEIGHT, 100.0f, 10000000.0f);
```

改完后的效果如下：



Results

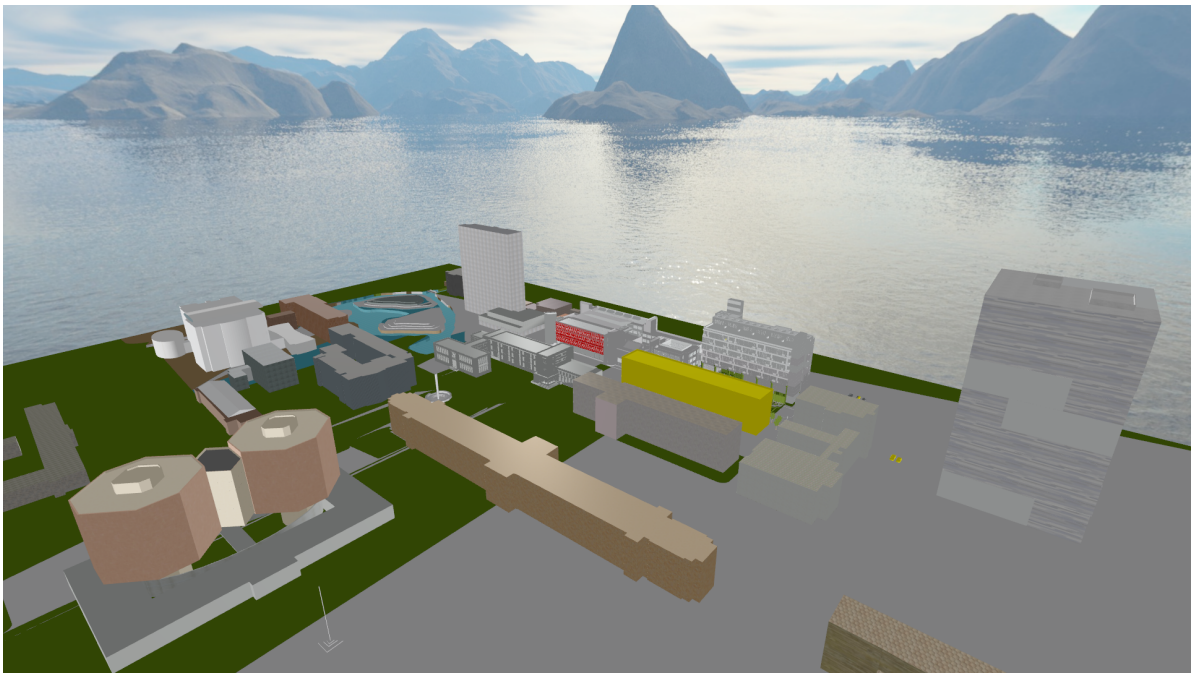
由于一些未解决的bug，我们的项目还存在着以下问题：

地面的贴图导入失败，变成大块的颜色；

阴影效果消失（但是在测试模型上存在）。

根据其它组同学在期末报告后的猜测，可能是由于模型没导出法线造成的，，因为我们用sketchup没有这个选项。于是我们将sketchup导出的obj模型导入3dsmax中，以期导出法线。但是此时3dsmax又无法读取贴图，重新贴图对组内贴图的同学来说工作量巨大，所以我们暂且试着导出无贴图但是有法线的模型，除了地面仍因为未知原因有bug无阴影外，剩下的建筑模型有了阴影。（但是我们展示的版本仍然是期末汇报的版本，之后在假期我们会继续修复）。

效果图如下（视频在文件夹中）：



Roles in group

学号	姓名	完成内容	贡献百分比
1752204	李宛霖（组长）	部分建筑场景的建模和模型导入	20%
1852839	李培然	模型导入整合及天空盒；期末报告PPT的制作	20%
1853702	刘吉宇	模型渲染及代码整合；proposal的编写	20%
1852410	刘卓奇	部分建筑场景的建模和贴图；proposal的PPT和中期报告PPT的制作	20%
1856005	吴伟登		0%
1854127	周家旋	部分建筑场景的建模和贴图	20%

期末报告由所有人共同完成。

References

[1] https://learnopengl.com/

[2] https://www.autodesk.com/products/3ds-max/overview

[3] http://cs1.tongji.edu.cn/courses/CS100433/

[4] Hearn D, Baker M P, Carithers W R. Computer graphics with OpenGL[M]. Upper Saddle River, NJ: Pearson Prentice Hall, 2014.

[5] https://developer.nvidia.com/gpugems/gpugems/part-ii-lighting-and-shadows/chapter-11-shadow-map-antialiasing

[6] https://developer.nvidia.com/gpugems/gpugems/part-ii-lighting-and-shadows/chapter-14-perspective-shadow-maps-care-and-feeding

[7] https://blog.sketchup.com/

[8] Kessenich J, Sellers G, Shreiner D. OpenGL programming guide: the official guide to learning OpenGL, Version 4.5 with SPIR-V[M]. Addison-Wesley Professional, 2016.

[9] http://ogldev.atspace.co.uk/

[10] https://www.khronos.org/opengl/wiki/