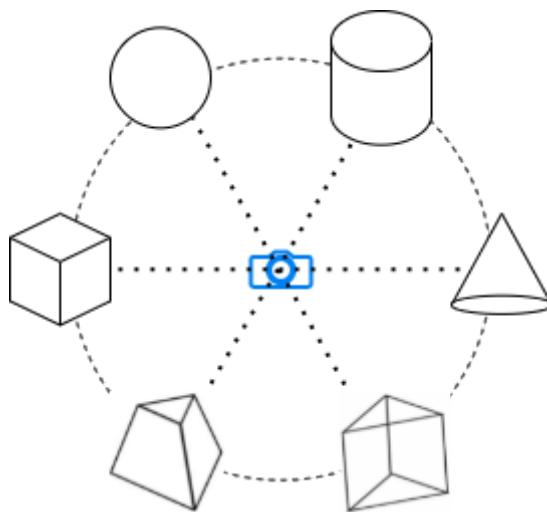


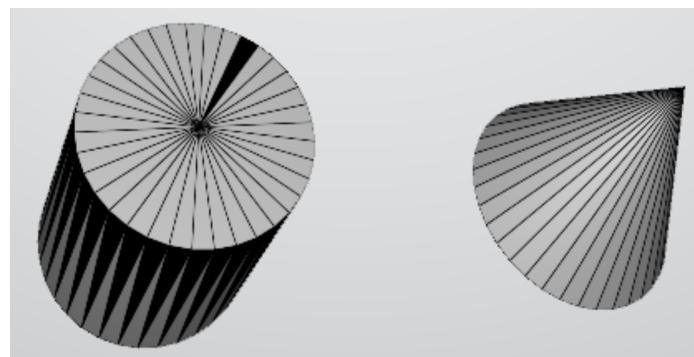
CG大程：简单室内场景三维建模及真实感绘制

一、基本靠求

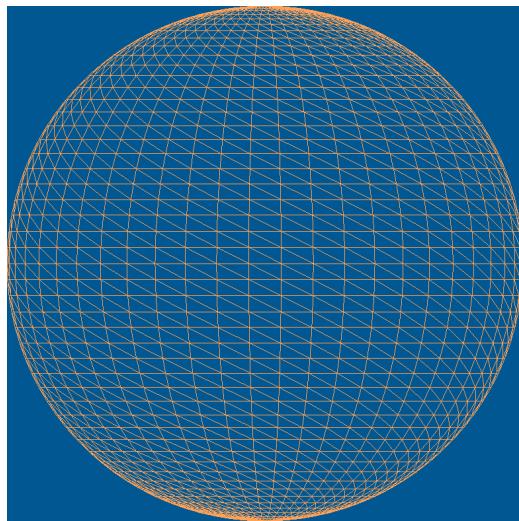
1. 基本体素的建模表达



实验中实现了立方体、球、圆柱、圆锥、三棱柱、三棱台六种基本体素的建模表达。`six_basic.cpp` 中分别定义了对应的类，核心是计算各个体素的顶点和索引。含有曲面的球、圆柱和圆锥绘制难度更大一些，需要将曲面离散化成三角形。对于圆柱，实验中沿着上下表面边缘采样离散点，利用这些点组合成三角形；圆锥的处理也是同样的思路：



绘制球的难度最大，实验中按照经纬度的方式拆解计算点：



```
// vertices
for (int y = 0; y <= _segments; y++)
{
    for (int x = 0; x <= _segments; x++)
    {
        // vertices calculated under the polar coordinate
        float xSegment = (float)x / (float)_segments;
        float ySegment = (float)y / (float)_segments;
        float xPos = std::cos(xSegment * 2.0f * MY_PI) * std::sin(ySegment * MY_PI) * _radius;
        float yPos = std::cos(ySegment * MY_PI) * _radius;
        float zPos = std::sin(xSegment * 2.0f * MY_PI) * std::sin(ySegment * MY_PI) * _radius;
        _vertices.push_back(xPos);
        _vertices.push_back(yPos);
        _vertices.push_back(zPos);
    }
}

// indices
for (int i = 0; i < _segments; i++)
{
    for (int j = 0; j < _segments; j++)
    {
        _indices.push_back(i * (_segments + 1) + j);
        _indices.push_back((i + 1) * (_segments + 1) + j);
        _indices.push_back((i + 1) * (_segments + 1) + j + 1);
        _indices.push_back(i * (_segments + 1) + j);
        _indices.push_back((i + 1) * (_segments + 1) + j + 1);
        _indices.push_back(i * (_segments + 1) + j + 1);
    }
}
```

2. OBJ三维网格导入导出

实验以LOFT公寓模型为主体，模型的信息保存在 `.obj` 和 `.mtl` 文件中，由 `Model` 类负责导入和导出。

obj 文件是3D模型文件格式，它是一种文本文件，可以直接用写字板打开进行查看和编辑，以下是实验中使用的 `Bedroom.obj` 文件的部分内容：

```
# Blender v3.1.2 OBJ File: 'Bedroom_color.blend'
# www.blender.org
mtllib Bedroom.mtl
o wall.001_Cube.002
v -2.490000 -0.006239 -2.015691
v -2.490000 0.003761 -2.005691
...
vt 0.624375 0.999500
vt 0.375625 0.999500
...
vn 0.0000 0.0000 1.0000
vn 1.0000 0.0000 0.0000
...
usemtl wall
s off
f 4/1/1 2/2/1 32/3/1 31/4/1 25/5/1
f 32/3/1 2/2/1 15/6/1 17/7/1 4/1/1 25/5/1 26/8/1 29/9/1 27/10/1 28/11/1 30/12/1
...
o Floor.001_Cube.005
...
```

`v`、`vt` 和 `vn` 分别代表几何顶点、纹理坐标和顶点法线，`f` 表示面，`f v1/vt1/vn1 v2/vt2/vn2` 依次用 / 分隔开顶点索引、纹理坐标索引和顶点法线索索引。`o` 声明用于标识在一个文件中的不同对象单元。

`.mtl` 文件（Material Library File）是材质库文件，与 `.obj` 文件配合，把纹理颜色渲染到 `.obj` 模型上。在 `.obj` 中通过 `mtllib` 来引用材质库；`usemtl` 则用于指定材质，例如 `Bedroom.mtl` 中这么定义 `wall`：

```
newmtl wall
Ns 250.000000
Ka 1.000000 1.000000 1.000000
Kd 1.000000 0.641031 0.403095
Ks 0.500000 0.500000 0.500000
Ke 0.000000 0.000000 0.000000
Ni 1.450000
d 1.000000
illum 2
```

`Ns` 和 `Ni` 分别指定材质的反射指数和折射值，`Ka`、`Kd`、`Ks` 和 `Ke` 分别表示环境反射、漫反射、镜面反射和放射光，`d` 指明渐隐指数，1 代表完全不透明，`illum` 是照明度。

实验中模仿 `tiny obj loader` 实现了简单的模型导入和导出功能。核心思想是按行处理 `.obj` 文件，根据不同的属性分别读取数据。需要额外注意的是几种特殊属性：

- `mtllib` 导入材质库

实验中简化了材质属性，利用 `material_t` 结构体保存：

```

typedef struct {
    std::string name;
    float ka[3];
    float kd[3];
    float ks[3];
    float ke[3];
    float ns;
    float ni;
    float d;
    int illum;
    std::map<std::string, std::string> unknown_parameter;
} material_t;

```

.mtl 文件中以 newmtl 为分隔，依次读取材质保存在向量中。

- 定义对象

同一个 .obj 文件中可以定义不同的对象，它们有不同的材质，因此新定义了 shape_t 结构体逐一保存。

```

typedef struct {
    std::string name;
    mesh_t mesh;
} shape_t;

```

不同对象共享顶点数据，因此只需额外保存索引和材质：

```

typedef struct {
    std::vector<index_t> indices;
    std::vector<unsigned char> num_face_vertices; // The number of
vertices per
                                                // face. 3 = polygon,
4 = quad,                                     // ... Up to 255.
                                                // per-face material
    std::vector<int> material_ids;               ID
} mesh_t;

```

实验中将多边形表面都三角化了，因此 num_face_vertices 值为 3。

- usemtl 指定材质

此关键字之后，直到下一个关键字之前，对象都采用指定的材质，因此也需要更新储存当前对象数据的 shape 变量：

```

if (newMaterialId != material) {
    if (!faceGroup.empty())
    {
        // Flatten vertices and indices
        for (size_t i = 0; i < faceGroup.size(); i++) {
            const std::vector<vertex_index>& face =
faceGroup[i];

            vertex_index i0 = face[0];
            vertex_index i1(-1);

```

```

        vertex_index i2 = face[1];

        size_t npolys = face.size();

        // Polygon -> triangle fan conversion
        for (size_t k = 2; k < npolys; k++) {
            i1 = i2;
            i2 = face[k];

            index_t idx0, idx1, idx2;
            idx0.vertex_index = i0.v_idx;
            idx0.normal_index = i0.vn_idx;
            idx0.texcoord_index = i0.vt_idx;
            idx1.vertex_index = i1.v_idx;
            idx1.normal_index = i1.vn_idx;
            idx1.texcoord_index = i1.vt_idx;
            idx2.vertex_index = i2.v_idx;
            idx2.normal_index = i2.vn_idx;
            idx2.texcoord_index = i2.vt_idx;

            shape.mesh.indices.push_back(idx0);
            shape.mesh.indices.push_back(idx1);
            shape.mesh.indices.push_back(idx2);

            shape.mesh.num_face_vertices.push_back(3);
            shape.mesh.material_ids.push_back(material);
        }
    }
    shape.name = name;
    shapes->push_back(shape);
}
shape = shape_t();
faceGroup.clear();
material = newMaterialId;
}

```

实验中可以导出6个基本体素的 .obj 文件。SaveObj 被定义为 BaseGeo 的成员函数，但由于希望导出的是在世界坐标下的位置关系，因此需要左乘 model 变换矩阵，将局部坐标转变为世界坐标。此外，三角形面片的顶点索引也需要累加计算，才能与集合了6个几何体的顶点数据相对应。

3. 基本材质和纹理

前面提到，在 .mtl 文件中保存着对象的材质信息，它们被保存在 Model 类的成员变量 std::vector<material_t> _materials 中。实验中在每个顶点的缓冲区中额外保存了材质编号的信息：

```

glVertexAttribIPointer(3, 1, GL_INT, sizeof(VertexMaterial),
(void*)offsetof(VertexMaterial, material_id));
 glEnableVertexAttribArray(3);

```

这样每次渲染时，顶点着色器输出材质编号：

```
"layout(location = 3) in int aMaterial;\n"
...
"flat out int material_id;\n"
..."
```

交由片段着色器读取具体材质：

```
...
"flat in int material_id;\n"
...
"// material data structure declaration\n"
"struct Material {\n"
    "vec3 ka;\n"
    "vec3 kd;\n"
    "vec3 ks;\n"
    "float ns;\n"
};\n"
...
"uniform Material materials[20];\n"
```

纹理与 `project6` 相关，借助 `Texture2D` 类可以方便地实现纹理的显示和切换。

4. 基本几何变换

借助 `project2` 的思路，可以利用模型变换矩阵对第1部分的6个基本体素进行旋转、平移、缩放操作。此外，除自旋外，实验中还实现了绕相机的公转，只需要先通过 `view` 矩阵变换到相机坐标系，再左乘旋转矩阵就可以：

```
"    gl_Position = projection * rotation * view * model * vec4(aPosition,
1.0f);\n"
```

5. 基本光照模型

实验中实现了 `project5` 中的冯氏光照模型。此外，为了实时阴影的效果，实验允许调整光源的位置。

6. 场景漫游

`project3` 已经实现了相机位置移动和视角移动的功能，其中水平方向的视角移动即为 `pan`。本实验中增加了 `zoom in / out` 的观察功能。

使用鼠标的滚轮来缩放场景。视野 (`fov`) 定义了我们可以看到场景中多大的范围，当视野变小时，场景投影出来的空间就会减小，产生放大 (`zoom in`) 了的感觉。因此只需根据滚轮竖直滚动的大小调整视野：

```

if (_input.mouse.scroll.yoffset) {
    // zoom in / out
    if (_camera->fovy >= glm::radians(1.0f) && _camera->fovy <=
glm::radians(90.0f))
        _camera->fovy -= _input.mouse.scroll.yoffset * cameraZoomRate;
    if (_camera->fovy <= glm::radians(1.0f))
        _camera->fovy = glm::radians(1.0f);
    if (_camera->fovy >= glm::radians(90.0f))
        _camera->fovy = glm::radians(90.0f);
}

```

7. 屏幕截取并保存

本实验提供截取屏幕并保存成24位、无压缩的 `BMP` 文件的功能。可以利用库函数 `glReadPixels` 从缓冲区读取数据到内存，即当前显示的帧：

```

glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
glReadPixels(0, 0, width, height, GL_BGR, GL_UNSIGNED_BYTE, pixel_data);

```

注意到通过 `glPixelStorei` 修改像素保存时的对齐方式为4字节，这是因为 `BMP` 文件采用4字节对齐机制。此外，在分配内存时也需修正内存空间大小，不能直接使用图像的宽度乘以高度：

```

int i = width * BYTES_PER_PIXEL;
while (i % 4 != 0) ++i;
int pixel_data_length = i * height;

```

Windows 所使用的 `BMP` 文件，有一个54字节的文件头，包括了文件格式标识、颜色数、图象大小、压缩方式等信息。实验中从正确的 `BMP` 文件中读取文件头，并修改其中的宽度和高度信息，就可以得到新的文件头了：

```

// copy the bmp header
GLubyte BMP_header[BMP_HEADER_LENGTH];
fread(BMP_header, sizeof(BMP_header), 1, dummy);
fwrite(BMP_header, sizeof(BMP_header), 1, output);

// change the resolution
fseek(output, 0x0012, SEEK_SET);
fwrite(&width, sizeof(width), 1, output);
fwrite(&height, sizeof(height), 1, output);

```

二、额外要求

1. NURBS 曲线

`NURBS` (non-uniform rational B-spline) 是指非均匀有理B样条，非均匀是指节点矢量可任意分布，即一个控制顶点的影响力范围能够改变。

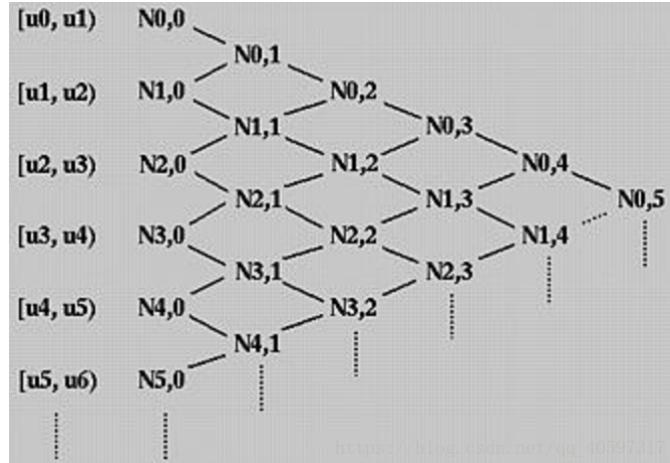
B样条曲线方程可表示为：

$$p(u) = \sum_{i=0}^n d_i N_{i,k}(u)$$

其中 d_i 为控制顶点, $N_{i,k}$ 为 k 次规范B样条基函数。基函数是由一个称为节点矢量的非递减参数序列 $U : u_0 \leq u_1 \leq \dots \leq u_{n+k+1}$ 所决定的 k 次分段多项式。B样条的基函数通常采用 Cox-deBoor 递推公式:

$$N_{i,0}(u) = \begin{cases} 1 & \text{if } u_i \leq u \leq u_{i+1} \\ 0 & \text{others} \end{cases}$$

$$N_{i,k}(u) = \frac{u - u_i}{u_{i+k} - u_i} N_{i,k-1}(u) + \frac{u_{i+k+1} - u}{u_{i+k+1} - u_{i+1}} N_{i+1,k-1}(u)$$



注意控制点和节点之间并不是一一对应的关系:

```
int delta = 0;

// determine the delta needed for the specific value of u
while (u >= _knots[delta + 1]) { delta++; }

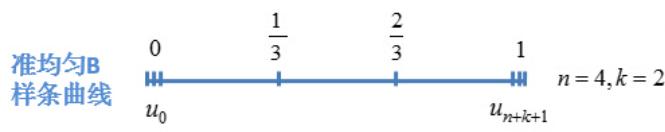
// get all the control points that are relevant to our u value
for (int i = 0; i <= _order - 1; i++)
{
    c.push_back(_controlPoints[delta - i] * _weights[delta - i]);
    w.push_back(_weights[delta - i]);
}
```

[liuwei792966953/nurbs-in-glfw \(github.com\)](https://github.com/liuwei792966953/nurbs-in-glfw) 提供了一种非递归的计算方式:

```
for (int r = _order; r >= 2; r--)
{
    int i = delta;
    for (int s = 0; s < r - 1; s++)
    {
        float denominator = _knots[i + r - 1] - _knots[i],
              omega = 0.f;
        if (denominator > FLOAT_ERR)
            omega = (u - _knots[i]) / denominator;
        c[s] = omega * c[s] + (1 - omega) * c[s + 1];
        w[s] = omega * w[s] + (1 - omega) * w[s + 1];
        i--;
    }
}
```

```
}
```

可见，不同类型的B样条曲线区别在于节点矢量的取值不同。实验中实现了准均匀B样条曲线：节点矢量中两端节点具有重复度 $k+1$ ，即 $u_0 = u_1 = \dots = u_k$, $u_{n+1} = u_{n+2} = \dots = u_{n+k+1}$, 所有内节点均匀分布，重复度为 1：

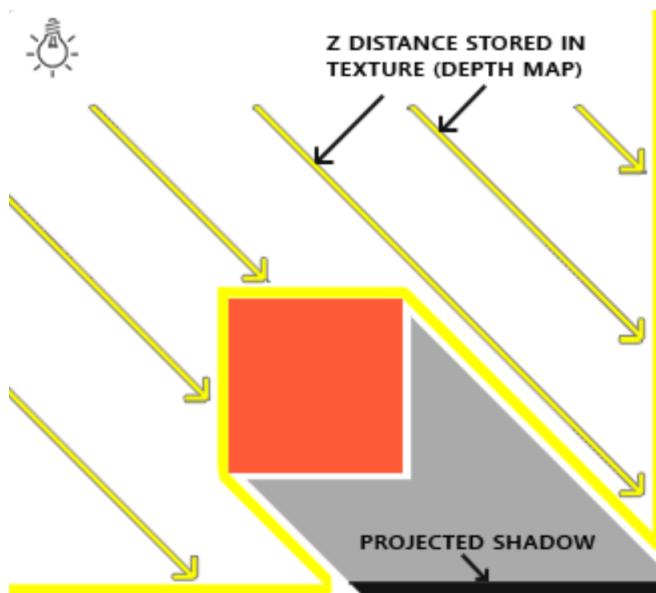


```
unsigned int minNumOfKnots = _controlPoints.size() + _order;
float uStep = (1.f / (_controlPoints.size() - _order + 1));
for (int i = 0; i < minNumOfKnots; ++i) {
    if (i < _order)
        _knots.push_back(0.f);
    else if (i > _controlPoints.size())
        _knots.push_back(1.f);
    else
        _knots.push_back(_knots[i - 1] + uStep);
}
```

2. 阴影贴图

阴影映射 (Shadow Mapping) 的基本思想是**以光的位置**为视角进行渲染，能看到的东西都将被点亮，看不见的就是在阴影之中。主要分为两步，首先渲染深度贴图，然后像往常一样渲染场景，使用生成的深度贴图来计算片段是否在阴影之中。

• 深度贴图



我们希望得到光线第一次击中的那个物体，然后用这个最近点和射线上其他点进行对比。如果光线上的其他点比最近点更远，这个点就在阴影中。借助深度测试的思想，我们从光源的视角来渲染场景，这时深度缓冲中的值就对应光源能照亮的第一个片段。

和 `bonus1` 的想法类似，我们利用帧缓冲保存深度贴图。注意到从光的透视图下渲染场景时，我们只需要深度信息，所以通过将 `glDrawBuffer` 和 `glReadBuffer` 设置为 `GL_NONE` 来禁用颜色缓冲。

```
_depthMapFbo.reset(new Framebuffer);
_depthMap.reset(new Texture2D(GL_DEPTH_COMPONENT, SHADOW_WIDTH,
SHADOW_HEIGHT, GL_DEPTH_COMPONENT, GL_FLOAT));
_depthMapFbo->bind();
_depthMapFbo->attachTexture(*_depthMap, GL_DEPTH_ATTACHMENT);
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
_depthMapFbo->unbind();
```

第一次渲染用于生成深度贴图，注意需要调用 `glviewport`，因为阴影贴图和原来的场景有不同的分辨率：

```
// the 1st pass: generate depth map
glviewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
_depthMapFbo->bind();
glclear(GL_DEPTH_BUFFER_BIT);
 glEnable(GL_DEPTH_TEST);

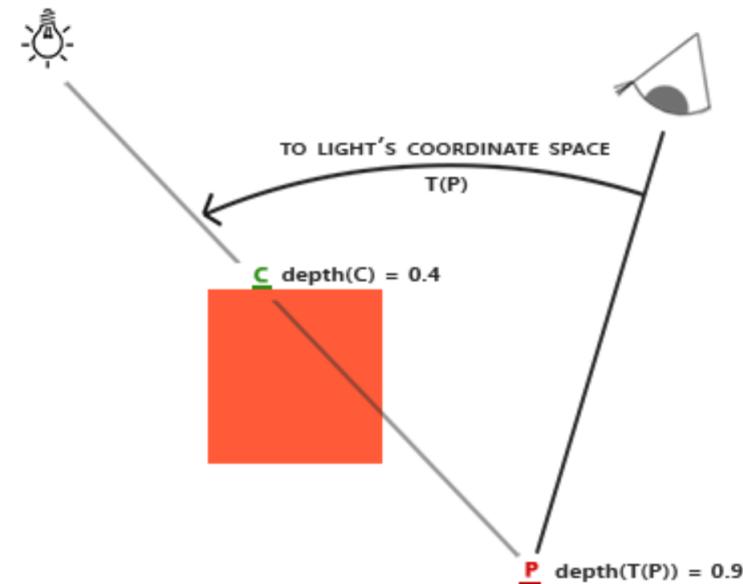
_depthMapShader->use();
_depthMapShader->setUniformMat4("projection", _lightProjection);
_depthMapShader->setUniformMat4("view", _lightview);

_loft->draw();
```

第二次渲染使用深度贴图正常渲染场景：

```
// the 2nd pass: apply depth map
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glviewport(0, 0, _windowwidth, _windowHeight);
...
_depthMap->bind(1);
_loft_shader->setUniformInt("shadowMap", 1);
...
```

• 光源空间的变换



在第一次渲染之前，还需要设置来自光源的视图和投影矩阵，它们将世界坐标转变为到光源的可见坐标。因为实验中使用的是平行光，可将投影矩阵使用正交投影矩阵，并使用 `glm::lookAt` 函数设置视图矩阵，从光源的位置看向光线方向：

```
_lightProjection = glm::ortho(-4.0f * aspect, 4.0f * aspect, -4.0f,
    4.0f, zNear, zFar);
_lightView = glm::lookAt(_directionalLight->transform.position,
    glm::vec3(0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
```

• 渲染至深度贴图

由于我们只关心深度值，只需一个简单的着色器，它负责把顶点变换到光空间，其他什么也不做：

```
// shader for depth mapping
const char* shadow_vs =
"#version 330 core\n"
"layout(location = 0) in vec3 aPosition;\n"

"uniform mat4 projection;\n"
"uniform mat4 view;\n"
"uniform mat4 model;\n"

"void main() {\n"
"    gl_Position = projection * view * model * vec4(aPosition,
1.0f);\n"
"}\n";

const char* shadow_fs =
"#version 330 core\n"
"void main() {\n"
"    // gl_FragDepth = gl_FragCoord.z;\n"
"}\n";
```

• 渲染阴影

正确生成深度贴图后就可以开始生成阴影了。在顶点着色器中进行光空间的变换：

```
...
"out vec3 fPosition;\n"
"out vec4 fPositionLightSpace;\n"
...
"uniform mat4 lightSpaceMatrix;\n"
...
"    fPosition = vec3(model * vec4(aPosition, 1.0f));\n"
"    fPositionLightSpace = lightSpaceMatrix * vec4(fPosition,
1.0f);\n"
```

在片段着色器中，我们利用 `shadowCalculation` 函数检验一个片段是否在阴影之中。当我们在顶点着色器输出一个裁切空间顶点位置到 `gl_Position` 时，OpenGL 自动进行一个透视线除法，将裁切空间坐标的范围 `-w` 到 `w` 转为 `-1` 到 `1`，因此我们需要对光空间片段位置手动做透视线除法，即将 `x, y, z` 元素除以向量的 `w` 元素来实现。此外，来自深度贴图的深度在 `0` 到 `1` 的范围，所以也需要将投影坐标变换到相应的范围。此时投影向量的 `z` 坐标就是来自光的透视线角的片段的深度 `currentDepth`，而光的位置视野下最近的深度 `closestDepth` 则可通过阴影纹理获得。若 `currentDepth` 高于 `closestDepth`，则片段在阴影中。

```
"float shadowCalculation(vec4 fragPosLightSpace) {\n"
"    // perspective division"
"    vec3 projCoords = fragPosLightSpace.xyz /
fragPosLightSpace.w;\n"
"    // from [-1,1] to [0,1]"
"    projCoords = projCoords * 0.5 + 0.5;\n"
"    float closestDepth = texture(shadowMap, projCoords.xy).r; "
"    float currentDepth = projCoords.z;\n"
"    float shadow = currentDepth > closestDepth ? 0.9 : 0.0;\n"
"    return shadow;\n"
"}\n"
```

片段着色器需要两个额外的输入，一个是光空间的片段位置和第一个渲染阶段得到的深度贴图：

```
"in vec4 fPositionLightSpace;\n"
...
"uniform sampler2D shadowMap;\n"
...
"    float shadow = shadowCalculation(fPositionLightSpace);\n"
"    vec4 coef = vec4(ambient + (1.0 - shadow) * (diffuse +
specular), 1.0f);\n"
```

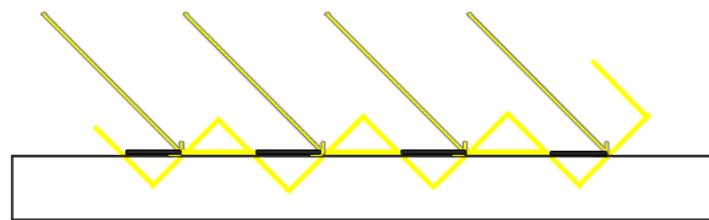
由于散射，阴影不会是全黑的，我们保留 `ambient` 分量。

• 阴影失真

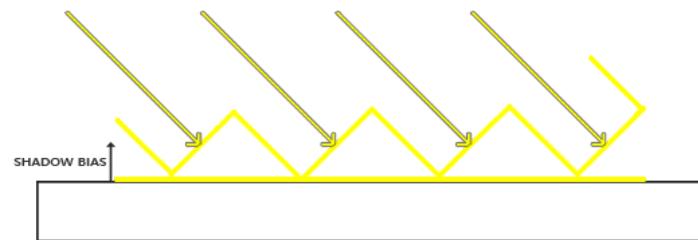
然而前面得到的场景明显有不对的地方：



这些明显的线条样式叫做阴影失真（shadow acne）。因为阴影贴图受限于分辨率，在距离光源比较远的情况下，多个片段可能从深度贴图的同一个值中去采样：



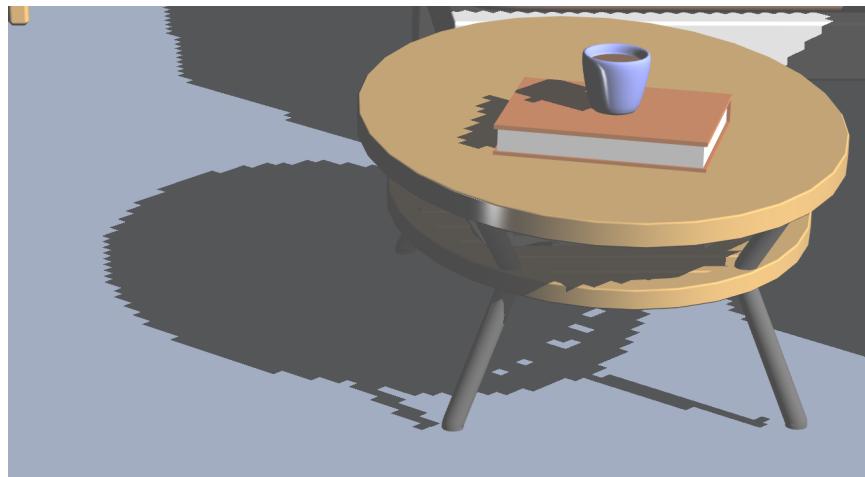
可以用阴影偏移（shadow bias）的技巧来解决这个问题，对表面的深度（或深度贴图）应用一个偏移量，这样片段就不会被错误地认为在表面之下了：



```
"    float bias = 0.000005;\n"
"    float shadow = currentDepth - bias > closestDepth ? 0.9 :
0.0;\n"
```

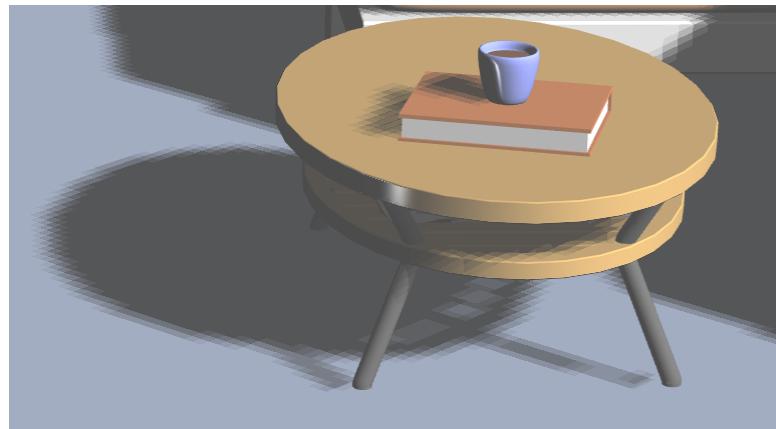
• PCF

然而深度贴图有一个固定的分辨率，多个片段对应于一个纹理像素。结果就是多个片段会从深度贴图的同一个深度值进行采样，这几个片段便得到的是同一个阴影，这就会产生锯齿边：



一种解决方案叫做 PCF (percentage-closer filtering)，这是一种多个不同过滤方式的组合，它产生柔和阴影，使它们出现更少的锯齿块和硬边。核心思想是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，我们就得到了柔和阴影。实验中简单的从纹理像素四周对深度贴图采样，然后把结果平均起来：

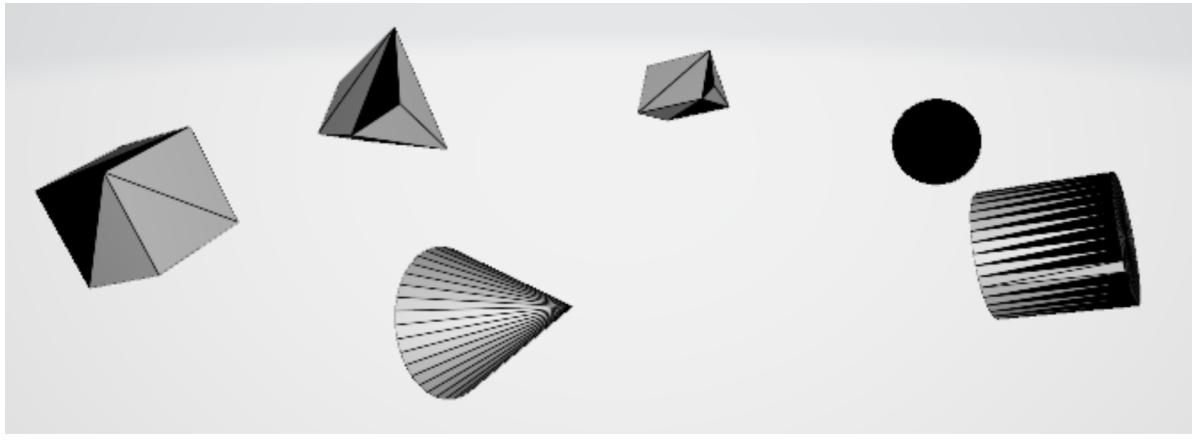
```
"    float shadow = 0.0;\n"
"    vec2 texelSize = 1.0 / textureSize(shadowMap, 0);\n"
"    for(int x = -1; x <= 1; ++x) {\n"
"        for(int y = -1; y <= 1; ++y) {\n"
"            float pcfDepth = texture(shadowMap, projCoords.xy +
vec2(x, y) * texelSize).r;\n"
"            shadow += currentDepth - bias > pcfDepth ? 0.9 : 0.0;\n"
"        }\n"
"    }\n"
"    shadow /= 9.0;\n"
```



三、实验结果

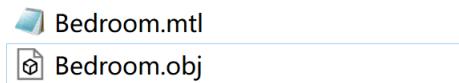
1. 基本体素

实验中绘制了6个基本体素，以下是导出 .obj 文件的结果：



2. OBJ网格

实验导入带有材质库 .mtl 的 .obj 文件：



3. 材质、纹理

材质库包含有对象的材质信息：

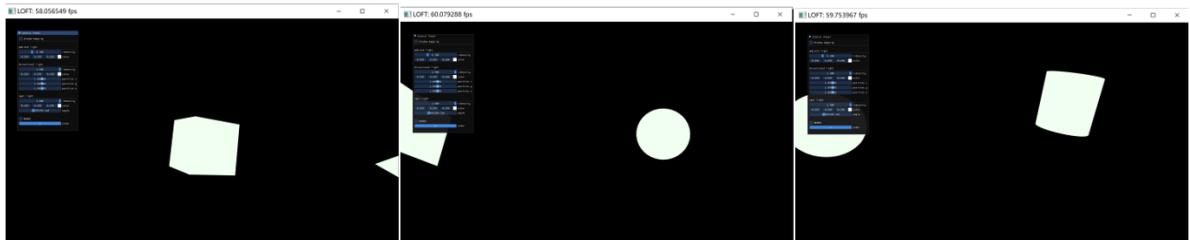
```
newmtl Black
Ns 250.000000
Ka 1.000000 1.000000 1.000000
Kd 0.089840 0.089840 0.089840
Ks 0.500000 0.500000 0.500000
Ke 0.000000 0.000000 0.000000
Ni 1.450000
d 1.000000
illum 2
```

实验中还可以改变壁画的纹理：

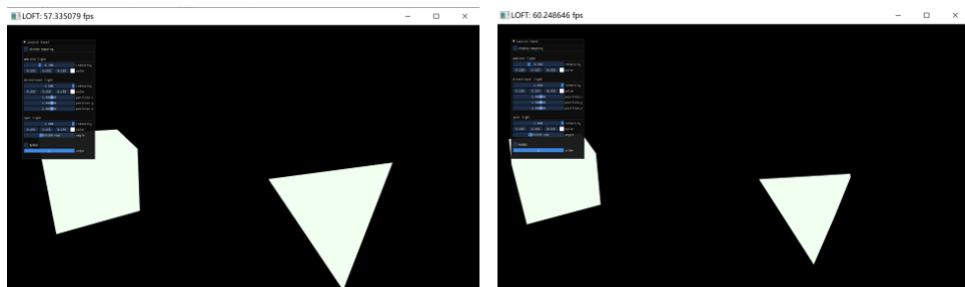


4. 几何变换

6个基本体素自转的同时，还绕相机公转：

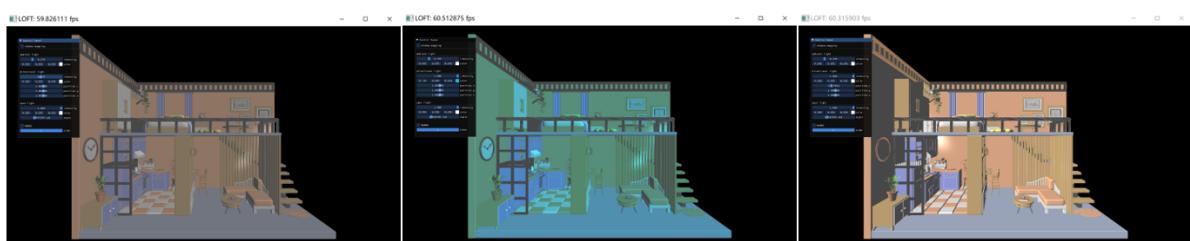


也可以放缩：



5. 光照模型

实验中实现了冯氏光照模型，可以改变光源的强度、颜色和位置：



6. 场景漫游

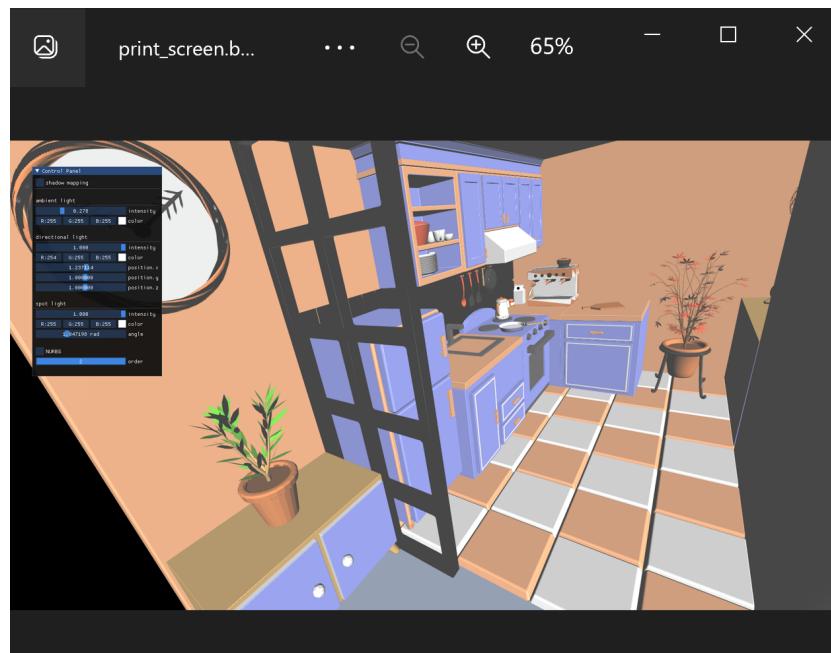
W / S / A / D 可以移动相机，按下右键并移动鼠标可以改变朝向，滚轮可以放大缩小：



7. 屏幕截取

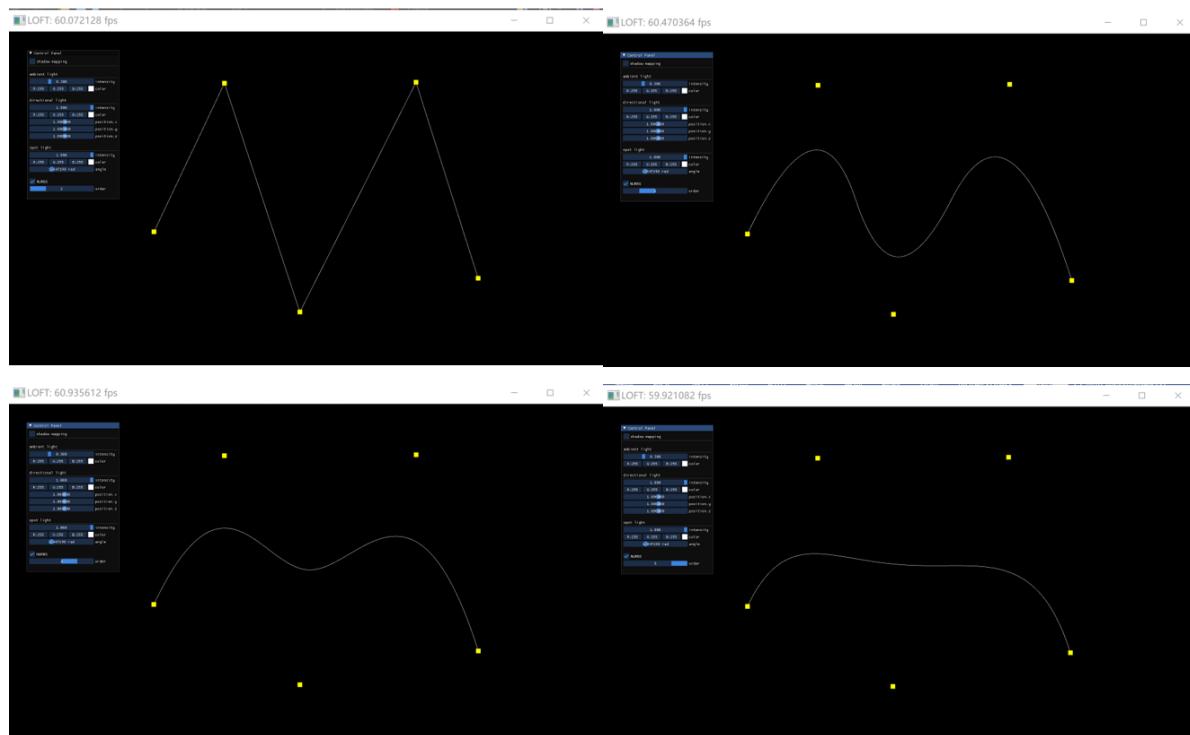
按下 **P** 键可以截屏：

screen shot has been saved to ../../media/print_screen.bmp

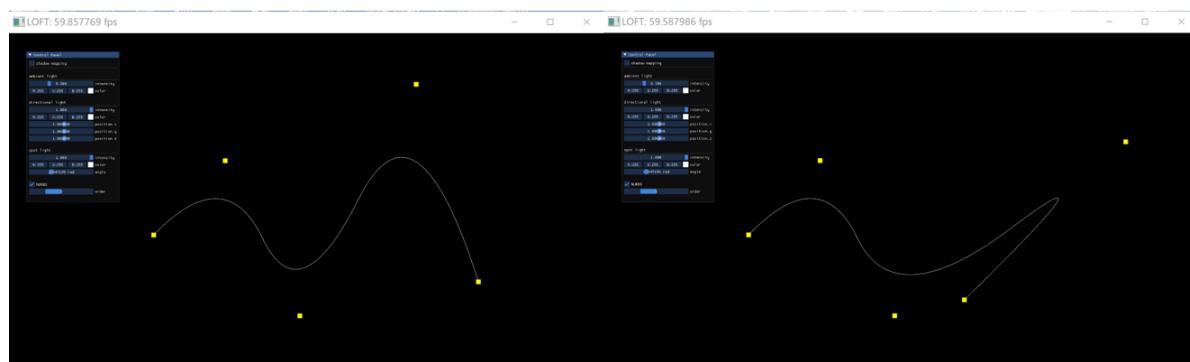


8. NURBS 曲线

可以鼠标指定控制点，并改变度数：



也可以改变控制点的位置：

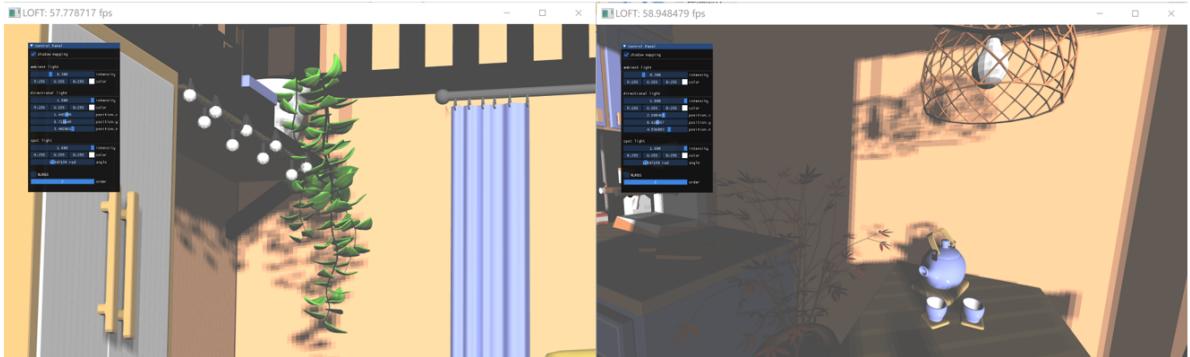


9. 实时阴影

利用阴影贴图可实现实时阴影的效果：



利用 `PCF` 柔化了阴影，有一定的抗锯齿效果：



四、讨论和心得

1. 从优秀的代码中学习跨平台编程

在导入 `.obj` 文件时，我的直接想法就是利用库函数 `getline` 逐行读取，但是阅读 `tiny obj loader` 的源代码，发现它另外实现了 `safeGetline` 函数。在 `Unix` 平台上换行符是 `'\n'`，`Windows` 上是 `'\r'\n'`，一些老式 `Mac` 平台则是 `'\r'`。尽管 `C++` 动态运行库可以处理不同平台的情况，但当人们跨平台移动数据文件时可能会出现问题。此外，实现细节中也有值得学习的地方，如该函数使用 `std::streambuf` 读取字符，比 `std::istream` 更快；且函数使用了 `sentry` 对象保证线程同步。

2. 对 `OpenGL` 的进一步学习

实验中出现了很多 `bug`，大部分是由对 `OpenGL` 的掌握不扎实导致的，在此记录备忘。

`GLSL` 中的 `flat`

在顶点着色器和片段着色器之前传递 `material_id` 时，出现了以下报错：

```
compile error:  
ERROR: 0:43: 'material_id' : int/uint varying in is not flat interpolated
```

原来，顶点和片段之间并不是一对一的映射。默认情况下，每个顶点的关联数据跨基元进行插值，以生成每个片段的相应关联数据，这就是平滑着色的作用。然而整型变量永远不会内插，必须被声明为 `flat`。`flat` 关键词指明该特定基元的光栅化过程中生成的每个片段都将获得相同的数据，也就是即便该图元由多个顶点定义，仅使用一个顶点的数据。这恰符合材质的要求，因为每个图元只能有一种材质。 ([opengl - "flat" qualifier in glsl? - Stack Overflow](#))

glVertexAttribPointer 与 int 类型

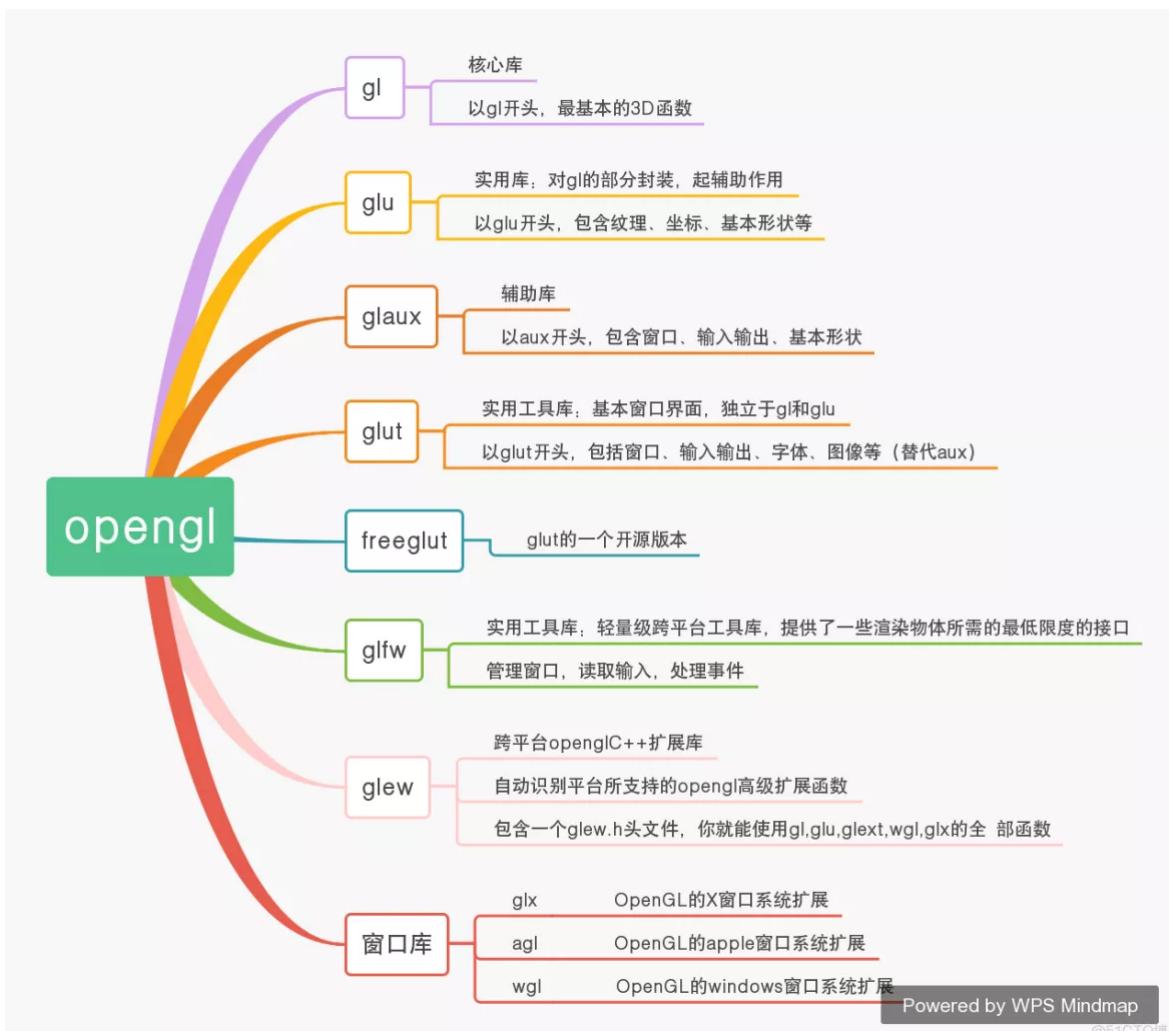
起初顶点的材质编号总是有误，费尽周章终于发现原来问题出在 `glVertexAttribPointer` 上。实验中利用该函数保存材质索引，索引是 `int` 类型，然而该函数似乎并不能接收 `GL_INT` 类型。尽管官方文档中表示 `glVertexAttribPointer` 和 `glVertexAttribIPointer` 都可以接收 `GL_INT` ([glVertexAttribPointer - OpenGL 4 - docs.gl](#)) :

```
type  
Specifies the data type of each component in the array. The symbolic constants  
GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, and  
GL_UNSIGNED_INT are accepted by glVertexAttribPointer and  
glVertexAttribIPointer. Additionally GL_HALF_FLOAT, GL_FLOAT,
```

但用 `glVertexAttribIPointer` 设置材质索引后就一切正常了。

glut、glfw、glew、glad

起初我本打算借助 `gluNurbssurface` 实现 NURBS 曲面，然而辛辛苦苦配置好 `freeglut` 之后，发现那些 `gluBeginSurface`、`glEvalCoord2f` 等等绘制的方式和熟悉的着色器流程大不相同。原来我们一直用的是 `glfw` 和 `glad` 配合，`glfw` 创建窗口界面，`glad` 实现对底层 `OpenGL` 接口封装，而另一种组合方式是 `freeglut` 和 `glew`：



3. 碎碎念

原先预留了近一个月的时间给另一个大程的选题。从零开始学习深度学习是一个极大的挑战，所以我的目标只是順利用 `jittor` 改写 [microsoft/CoCosNet-v2 \(github.com\)](https://github.com/microsoft/CoCosNet-v2)，能理解多少是多少。一路磕磕绊绊从卷积神经网络到生成对抗网络，再到循环神经网络，到最后终于能看懂大佬们的代码，我以为我快要成功了。然而不曾想，最大的困难竟然是 `jittor` 简短的说明文档。`jittor` 和 `torch` 还是有挺大差别的，甚至有些函数并没有对应的替代 ([jittor与torch之间的差异 CV小小白的博客-CSDN博客](#))。费尽心思调试了一周的代码，仍是问题不断。当初挑战深度学习需要勇气，现在在DDL之前变换选题更需要头铁。尽管还留下了很多遗憾，比如没能实现 `NURBS` 曲面的绘制，没能来得及学习 `BVH` 以实现碰撞检测和光线追踪等等，但这一周全身心的付出，一定会在图形学的学习之旅中画上浓墨重彩的一笔。