计算机图形学——期末大作业报告

姓名	学号	分工
曾毅	18340007	天空盒场景实现、PPT制作、课程展示演讲、期末报告撰写与整合
胡霆熙	18340057	天空盒加载优化、L-system文法与分形树部分实现、地面贴图的实现
李华靖	14348049	环境光照实现、交互式摄像机实现
侯少森	18340055	阴影的渲染与优化实现

1. L-System 文法

L-system是一种分形图形生成的方法，其主要原理是设定基本简单的绘图规则，然后让计算机根据这些规则进行反复迭代，就可以生成各种各样的图形来。在我们的L-system生成树中，通过设定初始字符串，迭代数次将字符串中的部分字符用一定的规则替换成特定字符串，最终就能得到一长串字符串。对字符串中的不同字符进行解释，就能用于分形树的生成。

在我们的代码中，L-system通过`Grammar`类实现。

1.1 文法规则的设定

在利用文法规则生成字符串前，我们要先设定好文法规则，即什么字符用什么字符串替代。文法规则被存放在数组`vector<pair<char, vector<std::string> > >generations;`中，该数组的每个元素是`pair`数对，数对元素为字符和字符串列表，包含被替换的字符和替换结果字符串。一个字符有可能有多个替换规则，因此需要将一个特定字符映射到一个字符串的列表中，每次随机选取一个字符串进行替换。实现向`generations`添加文法的成员函数如下：

```
// 输入：需要被替换的字符ch，替换的结果字符串ref
void Grammar::addGeneration(const char& ch, const string& ref){
    int id = find(ch);      // 查找替换文法中是否已经出现过该字符
    // 若没有出现过，则新增该字符的映射关系
    if (id == -1) {
        pair<char, vector<string> > temp;
        temp.first = ch;
        temp.second.push_back(ref);
        generations.push_back(temp);
        return;
    }
    // 否则直接在该字符的映射结果中加入该字符串
    generations[id].second.push_back(ref);
}
```

其中，`find`函数用于在`generations`中寻找是否出现过该需要被替换的字符。每次遍历`generations`的每个元素，如果找到了则返回对应下标，否则返回-1：

```
int Grammar::find(char ch){
    for (int i = 0; i < generations.size(); i++) {
        if (generations[i].first == ch)
            return i;
    }
    return -1;
}
```

1.2 L-system替换过程的实现

有了文法替换规则之后，就能进行字符串的迭代替换了。在具体代码中，迭代次数被保存在成员变量`lvl`中，字符串`start`和`result`分别表示初始字符串和迭代结果。

在每次迭代过程中，从头往后遍历整个字符串的每个字符，如果该字符存在替换规则，则在可替换的结果字符串中利用随机数随机选取一个进行替换。替换的结果使用`tmp`变量实时更新。具体实现如下：

```
// 迭代依据词法替换字符
void Grammar::iterateFor(int num){
    setLevel(num);      // 设置lvl
    srand(time(NULL));  // 设置随机数种子
    result = start;
    // 进行num次迭代替换
    for (int i = 0; i < num; i++) {
        string tmp = "";
        // 每次遍历整个字符串进行替换，将结果加入tmp
        for (int j = 0; j < result.size(); j++) {
            string t = search(result[j]);
            tmp += t;
        }
        result = tmp;
    }
}
```

其中，`search`函数能够使用给出的字符，利用之前设定好的文法替换规则，返回替换后的字符串：

```
string Grammar::search(char ch){
    int id = find(ch);
    // 如果替换规则中不存在该字符，则返回其本身（格式转换为字符串）
    if (id == -1) {
        string ret;
        ret.push_back(ch);
        return ret;
    }
    // 否则在替换结果中利用随机数随机选择一个返回
    int num = generations[id].second.size();
    int index = rand() % num;
    return generations[id].second[index];
}
```

由此就实现了L-system文法。通过设定文法替换规则、初始字符串、迭代次数，就能通过多次迭代与利用文法规则替换来得到一长串字符串，作为绘图的依据。

2. 分形树构建与纹理贴图

分形树由`FractalSystem`类实现，负责分形树的生成、建模、渲染等过程。具体的一棵树的生成通过调用`process`成员函数实现：

```
void FractalSystem::process(){
    initGrammar(level);      // 通过词法生成字符串
    generateFractal();       // 通过字符串生成树的建模参数
    DrawInit();              // 通过树的建模参数设置渲染参数
}
```

2.1 分形树的字符串生成与解释

具体的树的生成涉及到多个方面。为了使得树的结果更为真实，要考虑树的分支、枝条延伸长度与半径、树枝的具体方向等等。在分形树的构造函数中，对下面的变量进行初始化：

```
FractalSystem::FractalSystem(int lvl = 2){
    // 旋转角度
    dx = dz = 35.0f;
    dy = 30.0f;
    // 树干长度和半径
    length = 4.0;
    radius = 0.15f;
    // 长度半径衰减比例
    lengthFactor = 0.75;
    radiusFactor = 0.72;
    // 树干和树叶计数
    numTrunks = numLeafs = 0;
    // 树的深度
    level = lvl;
}
```

树枝不能只朝一个方向，因此需要进行旋转，要设置每次沿三个轴的旋转角度。树干的长度和半径为初始树干的值，随着迭代的加深，枝干的长度会按照衰减参数减小，从而使得更深层的树枝更细更短，从而使得结果更为真实。注意到分形树参数的深度参数`lvl`是可修改的（注意这里是树的深度而不是L-system字符串的迭代深度），这使得我们可以通过加深树的深度实现树的生长效果。

接下来，需要先通过L-system文法来生成表示整棵树的字符串。这已经通过上述的`Gramma`类实现。将其实例`gramma`作为分形树的成员变量，进行调用即可：

```
void FractalSystem::initGrammar(int levels){
    // 设置词法
    grammar.addGeneration('S', "F[^$X][*%X][&%X]");
    grammar.addGeneration('X', "F[^%D][&$D][/$D][*%D]");
    grammar.addGeneration('X', "F[&%D][*$D][/$D][^%D]");
    grammar.addGeneration('D', "F[^$X][*%FX][&%X]");
    // 设置起始字符串并迭代生成结果
    grammar.setStart("S");
    grammar.iterateFor(levels);
}
```

于是，`grammar`的`result`即为分形树的字符串。对于这整个字符串，需要对每个字符进行解释，用来描述出生成的整棵树。具体的解释规则如下：

字符	解释
F	当前树枝向前伸长一段距离
$	绕y轴正向旋转
%	绕y轴正向旋转
^	绕x轴正向旋转
&	绕x轴反向旋转
*	绕z轴正向旋转
/	绕z轴正向旋转
[	入栈，保存当前的树枝状态
]	出栈，读取历史树枝状态

其中旋转的“正向”和“反向”具体是顺时针还是逆时针不用考虑，只需要保证二者是相反的即可。而每次的变换都是对具体的树枝进行讨论的。为了实现树的分支，需要对树枝的状态进行入栈和出栈。每次入栈保存树枝状态之后进行变换，再对当前树枝进一步进行变换，从而能够生成分支。出栈时当前树枝不再变换，形成树梢和叶子，读取历史树枝状态后，再进行进一步变换。

2.2 通过字符串获取分形树参数

树枝和树叶的参数分别用如下的结构体进行保存：

```
// 树枝
struct Trunk{
    glm::vec3 start;    // 起点
    glm::vec3 end;      // 终点
    float radius;       // 半径
    float length;       // 长度
    int level;          // 层次（越大则树枝越细越短）
    Trunk() {
        start = end = glm::vec3(0.0f);
        level = 1;
    }
};
```

```
// 树叶
struct Leaf{
    glm::vec3 pos;      // 生成位置
    glm::vec3 dir;      // 延伸方向
};
```

之后，开始对生成的字符串进行解释，并在解释的过程中同步更新参数，得到每一个树枝和每一片树叶的参数。参数保存在数组`vector<Trunk> trunks`和`vector<Leaf> leafs`中，数量保存在成员变量`numTrunks`和`numLeafs`中。

具体的参数生成过程如下：

```
void FractalSystem::generateFractal()
{
```

```
    trunks.clear();
    leafs.clear();
    // 初始化第一根树枝
    curState.pos = glm::vec3(0, 0, 0);  // 初始位置
    curState.dir = glm::vec3(0, 1, 0);  // 初始方向
    curState.length = length;           // 初始长度
    curState.level = 1;                 // 初始层次
    curState.radius = radius;           // 初始半径
    std::stack<State>stacks;

    // 依据字符串生成树的参数
    for (int i = 0; i < grammar.getResult().size(); i++) {
        char ch = grammar.getResult()[i];
        Trunk tmp;
        switch (ch) {
        case 'F': {     // 向前生长
            tmp.start = curState.pos;
            curState.pos += curState.dir*(float)curState.length;
            tmp.end = curState.pos;         // 将终点向dir方向延伸
            tmp.radius = curState.radius;
            tmp.level = curState.level;
            trunks.push_back(tmp);
            break;
        }
        case '$': {     // 绕Y轴旋转
            curState.dir = Geometry::RotateY(curState.dir, dy);
            break;
        }
        case '%': {     // 绕Y轴反向旋转
            curState.dir = Geometry::RotateY(curState.dir, -dy);
            break;
        }
        case '^': {     // 绕X轴旋转
            curState.dir = Geometry::RotateX(curState.dir, dx);
            break;
        }
        case '&': {     // 绕X轴反向旋转
            curState.dir = Geometry::RotateX(curState.dir, -dx);
            break;
        }
        case '*': {     // 绕Z轴旋转
            curState.dir = Geometry::RotateZ(curState.dir, dz);
            break;
        }
        case '/': {     // 绕Z轴反向旋转
            curState.dir = Geometry::RotateZ(curState.dir, -dz);
            break;
        }
        case '[': {     // 入栈，生成分支
            stacks.push(curState);
            curState.length *= lengthFactor;    // 长度衰减
            curState.radius *= radiusFactor;    // 半径衰减
            curState.level += 1;                // 层次+1
            break;
        }
        case ']': {     // 出栈，生成树梢和叶子
            if (curState.level == grammar.getLevel()) {
                Trunk tm = trunks[trunks.size() - 1];
```

```
                Leaf rs;
                rs.dir = tm.end - tm.start;      // 树叶方向
                rs.pos = tm.end;                 // 树叶位置，即树枝末端
                leafs.push_back(rs);
            }
            curState = stacks.top();
            stacks.pop();
            break;
        }
        default:
            break;
        }
    }
}
```

2.3 构建树枝的渲染模型

得到树的所有树枝树叶的参数后，想要将整棵树显示出来，还需要对整棵树的渲染模型进行构建，包括树的顶点坐标、贴图坐标和各个顶点的法向量、读入贴图，并且构建OpenGL渲染需要的渲染管道、VAO、VBO等。这些内容在`DrawInit`成员函数中实现。首先对树的各个坐标进行构建：

```
    vector<glm::vec3>trunkVer;
    vector<glm::vec2>texcoord;
    vector<glm::vec3>normal;
    TrunkInit(trunks, trunkVer, texcoord, normal);
```

每一根树枝的上述三个参数使用圆柱结构体保存：

```
    struct cylinderNode {
        vector<glm::vec3>vertice;
        vector<glm::vec2>texcoord;
        vector<glm::vec3>normal;
    };
```

在`TrunkInit`函数中，实现了对树的顶点坐标、贴图坐标和各个顶点的法向量的生成。

在我们的实现中，树是分层的，每个树枝有自己的层次`level`。而每层的树枝只是位置和朝向不同，其长度和半径是完全相同的，因此考虑每一层只建模生成一根"标准"树枝的模型参数，再通过平移和旋转变换得到该层所有的树枝的模型参数。

首先对每一层的标准树枝进行建模，其起点为原点，向Z轴正方向延伸。每一层的长度和半径按照之前设置的衰减参数进行衰减。

```
    // 对每一层的树干进行初始建模（原点往z轴正方向延伸）
    for (int i = 0; i < level + 1; i++) {
        cylinderNode record;
        Geometry::CylinderMesh(curLength, curRadius, record.vertice,
record.texcoord, record.normal);
        cylinderTable[i + 1] = record;
        curLength *= lengthFactor;
        curRadius *= radiusFactor;
    }
```

其中，`CylinderMesh`函数生成具体的树枝坐标参数。具体坐标的生成方法如下：

考虑将每根树枝看做是一个没有顶面和底面的六棱柱，即用六个矩形来构建一根树枝。而每个矩形又可以看做是两个拼接的三角形，利用又因为树枝起点在原点，向Z轴正方向延伸，已知树枝半径和长度，利用简单的几何知识，很容易得到树枝的六个矩形的各个顶点的位置坐标与法向量，同时记录下对应的纹理坐标：

```
        // 每个圆柱用6个矩形表示
        unsigned int slice = 6;
        float delta = 360.0f / (float)(slice - 1.0);
        vector<glm::vec3> nrl;
        nrl.reserve(2 * (slice - 1));
        // 分别计算每个矩形的参数
        for (unsigned int x = 0; x < slice - 1; x++) {
            float angle = delta*x;
            float rc1 = radius*cos(glm::radians(angle));
            float rs1 = radius*sin(glm::radians(angle));
            float rc2 = radius*cos(glm::radians(angle + delta));
            float rs2 = radius*sin(glm::radians(angle + delta));
            //得到矩形的四个点的坐标
            glm::vec3 point1 = glm::vec3(rc1, rs1, len);
            glm::vec3 point2 = glm::vec3(rc1, rs1, 0.0);
            glm::vec3 point3 = glm::vec3(rc2, rs2, len);
            glm::vec3 point4 = glm::vec3(rc2, rs2, 0.0);
            // 记录坐标（矩形相当于2个三角形6个点）
            vertice.push_back(point1); vertice.push_back(point2);
vertice.push_back(point3);
            vertice.push_back(point3); vertice.push_back(point2);
vertice.push_back(point4);
            // 记录纹理坐标
            texcoord.push_back(glm::vec2(x, 5.0f));
texcoord.push_back(glm::vec2(x, -5.0f));
            texcoord.push_back(glm::vec2(x + 1, 5.0f));
texcoord.push_back(glm::vec2(x + 1, 5.0f));
            texcoord.push_back(glm::vec2(x, -5.0f));
texcoord.push_back(glm::vec2(x + 1, -5.0f));
            // 记录法向量
            nrl.push_back(CalcNormal(point1, point2, point3));
            nrl.push_back(CalcNormal(point3, point2, point4));
        }
```

为了实现树的光照效果，我们还需要各个顶点的法向量。每个顶点涉及三个三角形的构建，将这三个三角形面的法向量取均值作为顶点的法向量即可：

```
        // 计算每个点的法向量（邻接三个面的法向量的均值）
        glm::vec3 tmp[12];
        tmp[0] = tmp[10] = glm::normalize(nrl[0] + nrl[8] + nrl[9]);
        tmp[1] = tmp[11] = glm::normalize(nrl[0] + nrl[1] + nrl[9]);
        for (int x = 2, y = 0; x <= 9; x++, y++) {
            tmp[x] = glm::normalize(nrl[y] + nrl[y + 1] + nrl[y + 2]);
        }
        normal.push_back(tmp[0]), normal.push_back(tmp[1]),
normal.push_back(tmp[2]);
        normal.push_back(tmp[2]), normal.push_back(tmp[1]),
normal.push_back(tmp[3]);
        normal.push_back(tmp[2]), normal.push_back(tmp[3]),
normal.push_back(tmp[4]);
```

```
        normal.push_back(tmp[4]), normal.push_back(tmp[3]),
normal.push_back(tmp[5]);
        normal.push_back(tmp[4]), normal.push_back(tmp[5]),
normal.push_back(tmp[6]);
        normal.push_back(tmp[6]), normal.push_back(tmp[5]),
normal.push_back(tmp[7]);
        normal.push_back(tmp[6]), normal.push_back(tmp[7]),
normal.push_back(tmp[8]);
        normal.push_back(tmp[8]), normal.push_back(tmp[7]),
normal.push_back(tmp[9]);
        normal.push_back(tmp[8]), normal.push_back(tmp[9]),
normal.push_back(tmp[10]);
        normal.push_back(tmp[10]), normal.push_back(tmp[9]),
normal.push_back(tmp[11]);
```

对每一层的标准树枝进行建模后，通过平移和旋转变换得到该层其他树枝的模型参数。已知树枝的起点和延伸方向。首先将标准树枝平移到和所求树枝的起点重合，就将二者的基本位置对准了。将二者的方向向量进行点乘，可以得到二者相差角度的余弦值，从而得到二者的角度差；将二者的方向向量进行叉乘，可以得到垂直这两根树枝的方向，即旋转轴。已知旋转轴和旋转角度，对标准树枝进一步进行旋转即可。

```
    // 对每一个树枝进行移动和旋转，到达实际位置
    for (int x = 0; x < len; x++) {
        glm::mat4 nmats(1.0f);
        glm::mat4 trans = Geometry::GetTranMat(trunks[x].start, trunks[x].end,
nmats);
        int nums = cylinderTable[trunks[x].level].vertice.size();
        for (int y = 0; y < nums; y++) {
            glm::vec4 tmp =
trans*glm::vec4(cylinderTable[trunks[x].level].vertice[y], 1.0f);
            vertice.push_back(glm::vec3(tmp.x, tmp.y, tmp.z));
            tmp = nmats*glm::vec4(cylinderTable[trunks[x].level].normal[y],
1.0f);
            normal.push_back(glm::vec3(tmp.x, tmp.y, tmp.z));
        }
        texcoord.insert(texcoord.end(),
cylinderTable[trunks[x].level].texcoord.begin(),
            cylinderTable[trunks[x].level].texcoord.end());
    }
```

接下来就是构建OpenGL所需的渲染参数。绑定VAO、VBO，构建着色器类实例，读入贴图等。此处与实现整个分形树的技术手段关系不大，不在报告中进行赘述。

```
glGenBuffers(1, &trunkVBO);
glGenVertexArrays(1, &trunkVAO);
trunkShader = new Shader(std::string(ShaderPath + "Shaders/trunk.vs").c_str(),
                         std::string(ShaderPath + "Shaders/trunk.fs").c_str());
trunkTex.loadTexture(ShaderPath + "textures/bark.png", true);
... ...
```

2.4 构建树叶的渲染模型

为了简化模型，我们构建树叶不考虑树叶的形状，而是用纹理本身的形状来生成树叶。首先获取树叶的位置坐标和纹理坐标：

```
void FractalSystem::LeafInit(vector<Leaf>leafs, vector<glm::vec3>&vertice,
vector<glm::vec2>&texcoord){
    int len = leafs.size();
    vertice.reserve(len);
    for (int x = 0; x < len; x++) {
        leafs[x].dir = glm::normalize(leafs[x].dir);
        vertice.push_back(leafs[x].dir*LEAF_WIDTH + leafs[x].pos);
    }
}
```

树叶只生成在树梢末端，因此每个树叶只用在其树梢位置向延伸方向延伸一小段距离，用一个点进行建模表示。

接下来树叶的渲染参数的构建与树叶基本相同，只说明树叶特别的部分。

树叶会随着四季的变化而改变颜色，我们选择使用着色器来改变树叶颜色而不是纹理本身的颜色。为了更明显地表示四季下叶子颜色的变化，我们将片段着色器中的颜色设置为纯色。在我们的实现中能够手动切换四季，因此不同的季节对应着不同的着色器（以春季为例）：

```
leafsShader_spring = new Shader(std::string(ShaderPath +
"Shaders/leafs.vs").c_str(),
        std::string(ShaderPath + "Shaders/leafs_spring.fs").c_str());
```

所有的季节对应同一张纹理图片：

```
leafsTex.loadTexture(ShaderPath + "textures/leaves.png", false);
```

另外，上面已经提及了每片树叶不使用三角形建模而使用单个顶点建模，因此在渲染时要选择顶点模式：

```
glDrawArrays(GL_POINTS, 0, numLeafs);
```

如此一来，在对应顶点的位置就会显示出纹理图片的形状，而不是单个顶点或普通的三角形（纹理图片需要提前画出叶子的形状，将背景设置为透明色）。

3. 天空盒背景

首先，设计好天空盒的头文件，包括多个 `public` 方法和 `private` 变量。为了更加方便地在外部文件使用四季切换，这里设计了 `GetSeason` 、 `SetSeason` 和 `LoadSkyBox` 用来提供天空盒场景的季节切换。

```
class Skybox {
public:
    Skybox();
    ~Skybox();
    void Draw(glm::mat4 view, glm::mat4 projection, Season season);    // 绘制渲染
天空盒场景
    void SetSeason(Season new_season) { this->season = new_season; };  // 设置当前
天空盒的季节
```

```
    void LoadSkyBox();  // 加载四个季节的天空盒
    Season GetSeason() { return this->season; };    // 获取天空盒的当前季节

private:
    unsigned int skymap[4];          // 四季天空盒纹理编号
    unsigned int skyVAO, skyVBO;
    Season season = SPRING;          // 天空盒当前场景的季节，默认为春天
    Shader* shader;
    void loadCubeMap(vector<string> faces, int season);
};
```

3.1 天空盒加载与渲染

天空盒的立方体贴图加载由 `loadCubeMap` 函数实现，天空盒的场景渲染由 `Draw` 函数实现。

`loadCubeMap` 函数根据 `skymap` 对应的季节编码生成纹理，并将其绑定到`GL_TEXTURE_CUBE_MAP`纹理类型，然后从数组 `faces` 中逐个读取六个纹理的文件路径。根据路径读取图片的同时，得到图片文件的基本信息，从而使用 `glTexImage2D` 生成2D纹理。最后定义天空盒纹理的环绕方式和过滤方式。

```
void Skybox::loadCubeMap(vector<string> faces, int season) {
    glGenTextures(1, &(skymap[season]));
    glActiveTexture(GL_TEXTURE0);
    int width, height, nrComponents;
    unsigned char* image;
    glBindTexture(GL_TEXTURE_CUBE_MAP, skymap[season]);
    for (GLuint i = 0; i < faces.size(); i++) {
        image = stbi_load(faces[i].c_str(), &width, &height, &nrComponents, 0);
        GLenum format;
        if (nrComponents == 1)format = GL_RED;
        else if (nrComponents == 3)format = GL_RGB;
        else if (nrComponents == 4)format = GL_RGBA;
        if (image) {
            glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, format,
                width, height, 0, format, GL_UNSIGNED_BYTE, image);
        }
        else {
            std::cout << "Cube texture failed to load at path: " << faces[i] <<
std::endl;
        }
        stbi_image_free(image);
    }
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_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);
    glBindTexture(GL_TEXTURE_CUBE_MAP, 0);
}
```

`Draw` 函数则根据 `view matrix` 和 `projection matrix` 渲染天空盒场景，这里使用了前置深度测试来优化渲染过程。根据深度值比较进行选择性渲染，对于被场景物体遮盖住的天空盒背景将不渲染，从而提高渲染效率。因此，这里的 `glDepthFunc` 选择的是 `GL_LEQUAL` 参数，这样深度缓冲会为天空盒用最大深度值来填充深度缓冲。随后，根据当前季节绑定到其对应编码的纹理进行渲染。

```
void Skybox::Draw(glm::mat4 view, glm::mat4 projection, Season season) {
    this->season = season;
```

```
    glEnable(GL_CULL_FACE);
    glCullFace(GL_BACK);
    glDepthFunc(GL_LEQUAL);
    glDepthMask(GL_FALSE);
    shader->use();
    shader->setMat4("view", glm::mat4(glm::mat3(view)));
    shader->setMat4("projection", projection);
    glBindVertexArray(skyVAO);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_CUBE_MAP, skymap[season]);
    glDrawArrays(GL_TRIANGLES, 0, 36);
    glBindVertexArray(0);
    glDepthMask(GL_TRUE);
    glDisable(GL_CULL_FACE);
    glDepthFunc(GL_LESS);
}
```

3.2 四季变换

分别记录四个季节的图片文件路径，然后将记录路径的数组 `faces` 和相应编号传输给 `loadCubeMap` 函数进行纹理加载，四个季节对应四个不同的纹理。进行四季切换时，只需要修改当前天空盒绑定的纹理编号即可实现场景切换。

```
// 加载四个季节的天空盒贴图
void Skybox::LoadSkyBox() {
    vector<string> faces;
    faces.reserve(6);
    // 载入春天的天空盒图案
    faces.push_back(ShaderPath + "textures/skybox/spring/cq.jpg");
    faces.push_back(ShaderPath + "textures/skybox/spring/ch.jpg");
    faces.push_back(ShaderPath + "textures/skybox/spring/cs.jpg");
    faces.push_back(ShaderPath + "textures/skybox/spring/cd.jpg");
    faces.push_back(ShaderPath + "textures/skybox/spring/cz.jpg");
    faces.push_back(ShaderPath + "textures/skybox/spring/cy.jpg");
    loadCubeMap(faces, 0);
    faces.clear();
    // 夏
    faces.push_back(ShaderPath + "textures/skybox/summer/xq.jpg");
    faces.push_back(ShaderPath + "textures/skybox/summer/xh.jpg");
    faces.push_back(ShaderPath + "textures/skybox/summer/xs.jpg");
    faces.push_back(ShaderPath + "textures/skybox/summer/xd.jpg");
    faces.push_back(ShaderPath + "textures/skybox/summer/xz.jpg");
    faces.push_back(ShaderPath + "textures/skybox/summer/xy.jpg");
    loadCubeMap(faces, 1);
    faces.clear();
    // 秋
    faces.push_back(ShaderPath + "textures/skybox/autumn/qq.jpg");
    faces.push_back(ShaderPath + "textures/skybox/autumn/qh.jpg");
    faces.push_back(ShaderPath + "textures/skybox/autumn/qs.jpg");
    faces.push_back(ShaderPath + "textures/skybox/autumn/qd.jpg");
    faces.push_back(ShaderPath + "textures/skybox/autumn/qz.jpg");
    faces.push_back(ShaderPath + "textures/skybox/autumn/qy.jpg");
    loadCubeMap(faces, 2);
    faces.clear();
    // 冬季
    faces.push_back(ShaderPath + "textures/skybox/winter/dq.jpg");
    faces.push_back(ShaderPath + "textures/skybox/winter/dh.jpg");
```

```
    faces.push_back(ShaderPath + "textures/skybox/winter/ds.jpg");
    faces.push_back(ShaderPath + "textures/skybox/winter/dd.jpg");
    faces.push_back(ShaderPath + "textures/skybox/winter/dz.jpg");
    faces.push_back(ShaderPath + "textures/skybox/winter/dy.jpg");
    loadCubeMap(faces, 3);
}
```

4. 环境光照

传统光照：用传统的局部光照模型实现光照效果

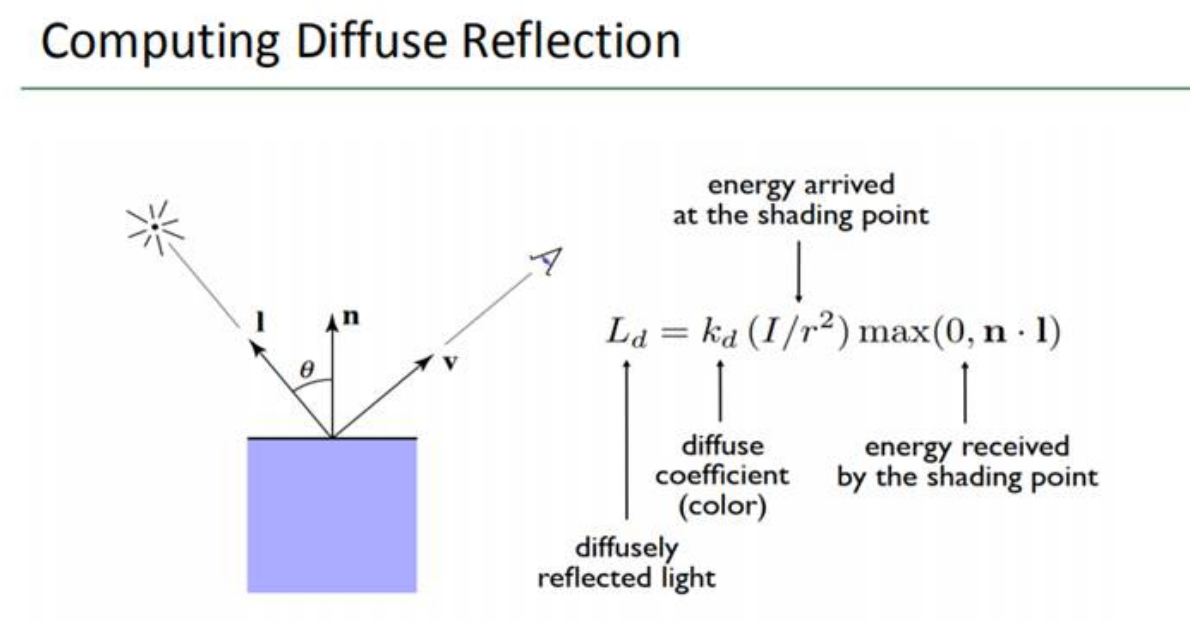
局部光照模型

最终颜色 = （环境光 + 漫反射光 + 镜面反射光）* 材质颜色；

（本项目场景的材质基本不进行镜面反射，故略去）

漫反射原理

已知：漫反射系数，平面法向量，入射光方向，光能量（太阳光能量，假设恒定），求漫反射光由下列公式给出。



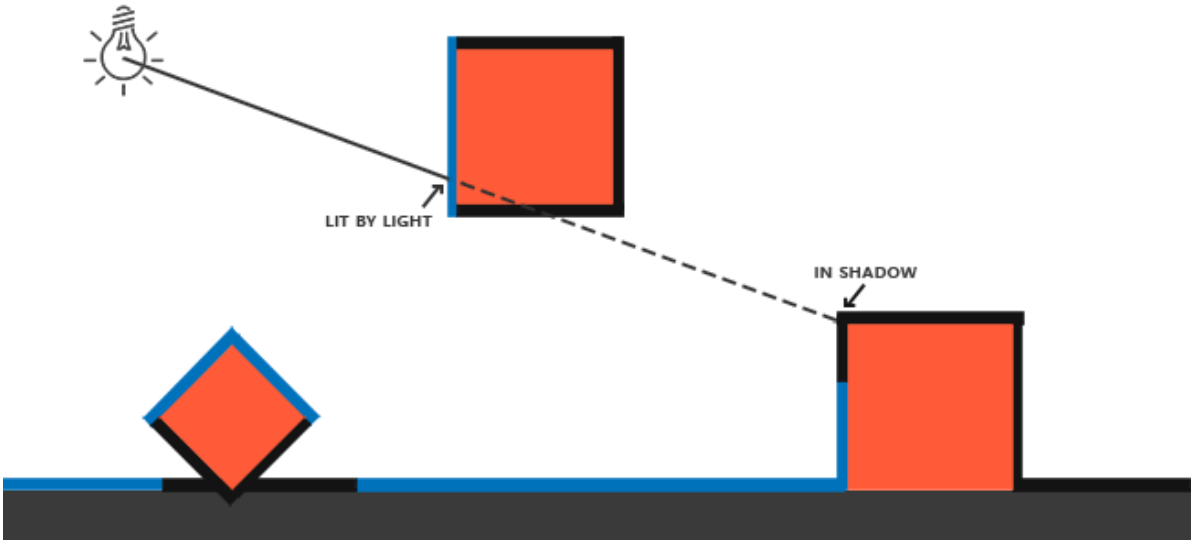
在opengl中实现光照，主要是对片段着色器的编程

片段着色器部分代码如下：

```
    vec3 lightDir = (light.direction);              //入射光方向的反方向
    float diff = max(dot(normal,lightDir),0.0);     //
    vec3 ambient = light.ambient*texColor;      //环境光系数*材质颜色
    vec3 diffuse = vec3(1.0f)*diff*texColor;      //计算漫反射
    return ambient+diffuse;                    //最终颜色
```

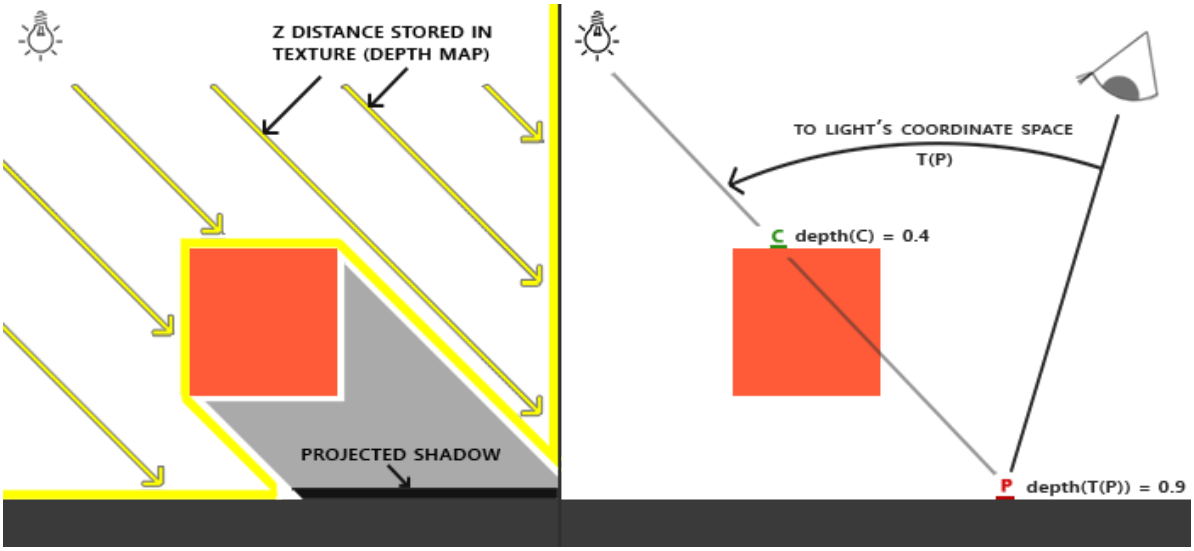
5. 阴影实现

阴影是光线被阻挡的结果；当一个光源的光线由于其他物体的阻挡不能够达到一个物体的表面的时候，那么这个物体就在阴影中了。阴影能够使场景看起来真实得多，并且可以让观察者获得物体之间的空间位置关系。场景和物体的深度感因此能够得到极大提升。



5.1 阴影映射

使用 `Shadow Mapping` (阴影贴图) 技术, 用来生成平行光的阴影, 其背后的思路十分简单: 以光的位置为视角进行渲染，能看到的东西都将被点亮，看不见的一定是在阴影之中了。如果绘制一条从光源出发的射线, 那么射线第一次击中的物体是可以被看到的, 然后看射线上的其他点是否比最近点更远, 如果是的话, 这个点就在阴影中。对从光源发出的射线上的成千上万个点进行遍历是个极端消耗性能的举措，实时渲染上基本不可取。可以采取相似举措，不用投射出光的射线, 而是使用深度缓冲。



5.2 深度贴图

首先创建帧缓冲对象以及深度纹理，提供给帧缓冲的深度缓冲使用。

```
...
    glGenFramebuffers(1, &depthMapFBO); // 创建帧缓冲对象
    GenDepthTex();
    glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO); //把生成的深度纹理作为帧缓冲的深度缓冲
```

```
glFramebufferTexture2D(GL_FRAMEBUFFER,GL_DEPTH_ATTACHMENT,GL_TEXTURE_2D,depthMap
,0);
    glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, depthMap, 0);
    glDrawBuffer(GL_NONE);
    glReadBuffer(GL_NONE);
    if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE) {
        cout << "Error to get frame buffer" << endl;
    }
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
...

void Scene::GenDepthTex() { // 创建深度纹理
    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_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,
GL_COMPARE_R_TO_TEXTURE);
    GLfloat borderColor[] = { 1.0,1.0,1.0,1.0 };
    glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
    glBindTexture(GL_TEXTURE_2D, 0);
}
```

在此之后还需要进行光源空间的变换，由于使用的是平行光，我们使用正交投影矩阵， 然后就要确定它的视锥体。显然我们需要将 `view` 视角下所有物体都包含到视锥体中，但是过大也没有意义，会引起不必要的渲染开销，另外也可以更充分的利用 `shadow depth map` 。

```
GLfloat near_plane = 1.0f, far_plane = 900.0f; //因为我们使用的是一个所有光线都平行的定向光。出于这个原因，我们将为光源使用正交投影矩阵，透视图将没有任何变形
GLfloat ws = 800.0f;
glm::mat4 lightProjection = glm::ortho(-ws, ws, -ws, ws, near_plane, far_plane);
glm::mat4 lightView = glm::lookAt(lightPos, glm::vec3(0.0f), glm::vec3(0.0, 1.0, 0.0)); //为了创建一个视图矩阵来变换每个物体，把它们变换到从光源视角可见的空间中，我们将使用glm::lookAt函数；这次从光源的位置看向场景中央。

lightSpaceMatrix1 = lightProjection * lightView; //二者相结合为我们提供了一个光空间的变换矩阵，它将每个世界空间坐标变换到光源处所见到的那个空间
```

然后开始生成深度贴图了， 主要分两个阶段，渲染深度贴图以及使用深度贴图渲染场景。

```
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
OutsideRenderShadow(model, camera, deltaTime, timeflow, season);
glBindFramebuffer(GL_FRAMEBUFFER, 0);

glViewport(0, 0, screenWidth, screenHeight);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
OutsideRender(model, view, projection, camera, deltaTime, timeflow, season);
```

最后用简单的顶点着色器以及片段着色器来渲染深度缓冲。

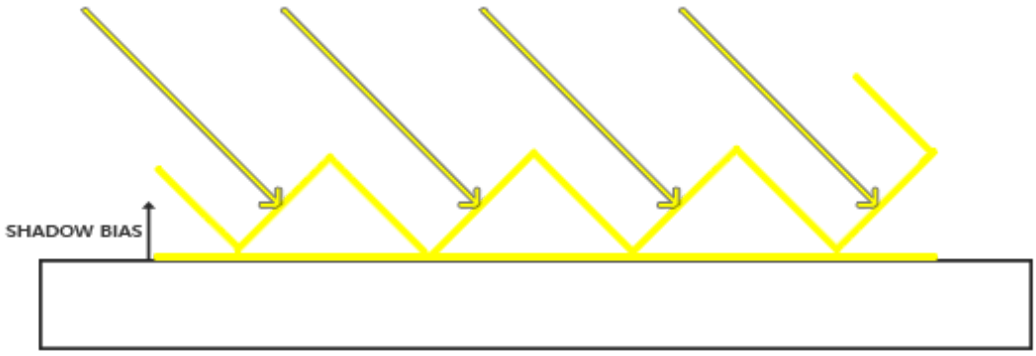
```
modelShader = new Shader(std::string(ShaderPath + "Shaders/model.vs").c_str(),
        std::string(ShaderPath + "Shaders/model.fs").c_str());// 阴影
shadowShader = new Shader(string(ShaderPath + "Shaders/shadow.vs").c_str(), //顶点着色器将一个单独模型的一个顶点，使用lightSpaceMatrix变换到光空间中
        string(ShaderPath + "Shaders/shadow.fs").c_str()); //由于没有颜色缓冲，最后的片段不需要任何处理，使用一个空片段着色器

shadowShader->use();
shadowShader->setMat4("lightSpaceMatrix", lightSpaceMatrix1);
modelShader->use();
modelShader->setInt("texture_diffuse1", 0);
modelShader->setVec3("dirLight.direction", glm::normalize(lightPos));
modelShader->setVec3("dirLight.ambient", ambientLight);
modelShader->setVec3("dirLight.diffuse", diffuseLight);
modelShader->setMat4("lightSpaceMatrix", lightSpaceMatrix1);
```

5.3 改进阴影贴图之阴影偏移

阴影失真 (Shadow Acne) 使得生成的阴影真实感不强。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。图片每个斜坡代表深度贴图一个单独的纹理像素。你可以看到，多个片段从同一个深度值进行采样。

虽然很多时候没问题，但是当光源以一个角度朝向表面的时候就会出问题，这种情况下深度贴图也是从一个角度下进行渲染的。多个片段就会从同一个斜坡的深度纹理像素中采样，有些在地板上面，有些在地板下面；这样我们所得到的阴影就有了差异。因为这个，有些片段被认为是在阴影之中，有些不在，由此会产生条纹样式。

可以采用阴影偏移 (Shadow Bias) 的技巧来解决这个问题，简单的对表面的深度应用一个偏移量，这样片段就不会被错误地认为在表面之下了。




```
glm::mat4 biasMatrix{
0.5,0.0,0.0,0.0,
0.0,0.5,0.0,0.0,
0.0,0.0,0.5,0.0,
0.5,0.5,0.5,1.0
};
lightSpaceMatrix2 = biasMatrix * lightSpaceMatrix1;
```

5.4 改进阴影贴图之PCF

`Shadow Map`的一个比较明显的缺点即是在生成的阴影边缘锯齿化很严重，而`PCF`则能有效地克服`Shadow Map`阴影边缘的锯齿。

对于`PCF`，其核心思路就是，如果是影子内部，则他周围的一圈点肯定也在阴影之中，如果是影子边缘，则他周围就会有些点不在阴影里，且越靠边，这些不在阴影中的邻居越多。所以检测一个点是否是在影子边缘，只要观察他的邻居就可以了。(`PCF`通过在绘制阴影时，除了绘制该点阴影信息之外还对该点周围阴影情况进行多次采样并混合来实现锯齿的柔化)

这里使用分层泊松采样, 为每个像素选择不同的样本来消除此条带。

```
vec2 poissonDisk[16] = vec2[](
   vec2( -0.94201624, -0.39906216 ),
   vec2( 0.94558609, -0.76890725 ),
   vec2( -0.094184101, -0.92938870 ),
   vec2( 0.34495938, 0.29387760 ),
   vec2( -0.91588581, 0.45771432 ),
   vec2( -0.81544232, -0.87912464 ),
   vec2( -0.38277543, 0.27676845 ),
   vec2( 0.97484398, 0.75648379 ),
   vec2( 0.44323325, -0.97511554 ),
   vec2( 0.53742981, -0.47373420 ),
   vec2( -0.26496911, -0.41893023 ),
   vec2( 0.79197514, 0.19090188 ),
   vec2( -0.24188840, 0.99706507 ),
   vec2( -0.81409955, 0.91437590 ),
   vec2( 0.19984126, 0.78641367 ),
   vec2( 0.14383161, -0.14100790 )
); //分层泊松采样

uniform sampler2DShadow shadowMap;

float ShadowCalculation(vec4 DepthCoord) //计算阴影
{
    float visibility = 1.0;
    float bias = 0.005;  //偏移
    visibility -= 0.2 * (1.0 - texture(shadowMap, vec3(DepthCoord.xy +
poissonDisk[0] / 6000.0,
                        (DepthCoord.z-bias) / DepthCoord.w) )); //执行透视除法
    visibility -= 0.2 * (1.0 - texture(shadowMap, vec3(DepthCoord.xy +
poissonDisk[3] / 6000.0,
                        (DepthCoord.z-bias) / DepthCoord.w) ));
    visibility -= 0.2 * (1.0 - texture(shadowMap, vec3(DepthCoord.xy +
poissonDisk[5] / 6000.0,
                        (DepthCoord.z-bias) / DepthCoord.w) ));
    visibility -= 0.2 * (1.0 - texture(shadowMap, vec3(DepthCoord.xy +
poissonDisk[7] / 6000.0,
                        (DepthCoord.z-bias) / DepthCoord.w) ));
    return visibility;
}
```

6. 交互式摄像机

实现相机类，已知相机位置，相机朝向，世界向上方向。通过叉乘求出相机正右方向与相机头顶方向。

```
glm::mat4 view;
view = glm::lookAt(glm::vec3(0.0f, 0.0f, 3.0f),
          glm::vec3(0.0f, 0.0f, 0.0f),
          glm::vec3(0.0f, 1.0f, 0.0f));
```

`glm::LookAt` 函数需要一个位置、目标和上向量。它可以创建一个和前面所说的同样的观察矩阵。

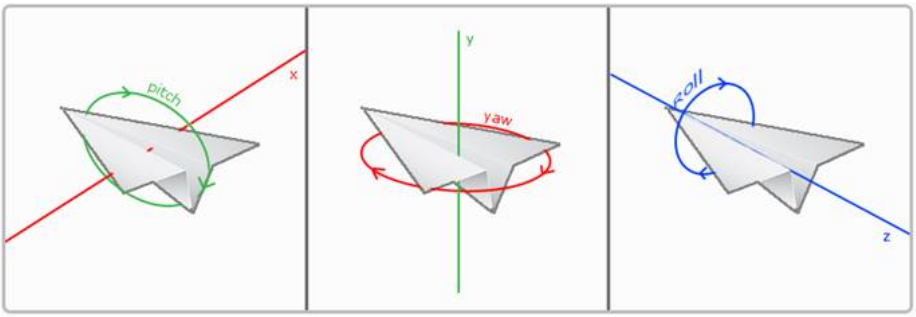
设置按键检测进行相机移动，部分代码：

```
if (direction == FORWARD)
    this->Position += this->Front * velocity;
if (direction == BACKWARD)
    this->Position -= this->Front * velocity;
if (direction == LEFTS)
    this->Position -= this->Right * velocity;
if (direction == RIGHTS)
    this->Position += this->Right * velocity;
```

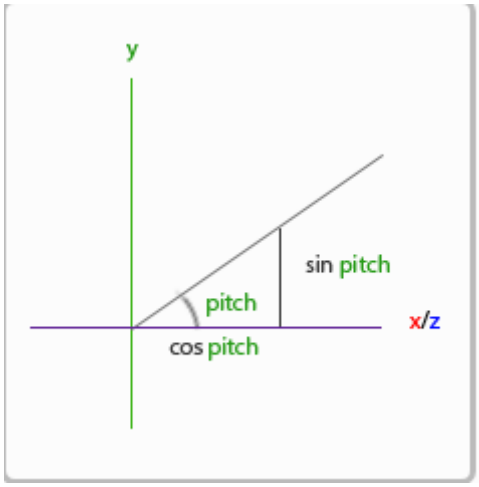
计算渲染时间差实现相机匀速移动：

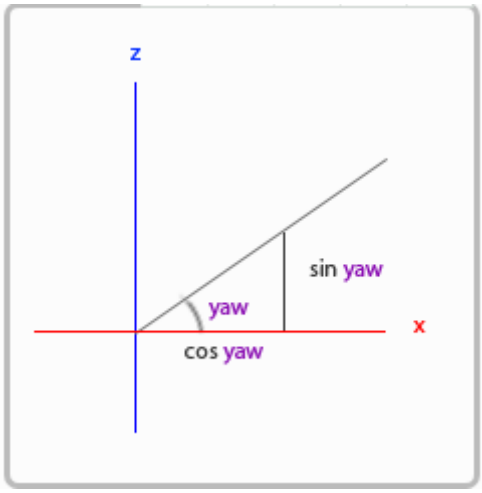
```
GLfloat velocity = this->MovementSpeed * deltaTime;
```

增加鼠标响应，计算鼠标坐标偏移量，求出俯仰角Pitch与偏航角Yaw。(滚转角Roll没必要，故不计算)



几何关系：





同样原理实现鼠标滚轮缩放镜头，并增加功能：

1.实现相机跳跃（模拟FPS游戏人物的跳跃，向上抛物线运动）

2.记录当前相机状态（位置，朝向，俯仰角，偏转角），实现不同相机状态的切换（目前能记录5个状态）。

3.相机照相，输出png文件至当前目录下。（glReadPixels函数读取当前颜色缓冲，stbi_write_png函数输出png图片）