2020 学年秋冬学期浙江大学

计算机图形学大作业

实验报告

指导老师：唐敏

助教：　　张智宇

小组成员：沈吕可晟

陆子仪

胡晶晶

徐闻语

目录

1 实验目的及项目概述

运用本学期计算机图形学课程所学知识，制作一个 ACT 类小游戏，具体类型为 FPS，第一人称射击游戏。

2 实验原理

2.1 基本要求

2.1.1 基本体素绘制（球，多面棱锥，多面棱柱的建模表达）

球：

使用 `gluSphere(uquadric, r, cradix, hradix)`函数绘制球体，四个参数分别代表绘制对象，球体半径，纵向细分度，横向细分度。

代码实现：

```
void GEO::drawBall(GLUquadricObj *uquadric,GLfloat r, GLfloat cradix,
GLfloat hradix) {
    uquadric = gluNewQuadric();
    glPushMatrix();
    gluSphere(uquadric, r, cradix, hradix);
    glPopMatrix();
}
```

多面棱锥：

代码实现：

```
void GEO::drawPyramid(GLfloat h, GLfloat r1, GLfloat r2, int n)
{
    int i;
    double Angle = 0;
    for (i = 0; i < n; i++)
    {
        glBegin(GL_TRIANGLES);

        glNormal3f(0.0, -1.0, 0.0);
        glVertex3f(0.0, 0.0, 0.0);
```

```
        glNormal3f(0.0, -1.0, 0.0);
        glVertex3f(r1 * sin(Angle * 3.1415926 / 180.0), 0.0,
            r1 * cos(Angle * 3.1415926 / 180.0));

        glNormal3f(0.0, -1.0, 0.0);
        glVertex3f(r1 * sin((Angle + 360.0 / n) * 3.1415926 / 180.0),
0.0,
            r1 * cos((Angle + 360.0 / n) * 3.1415926 / 180.0));

        glNormal3f(0.0, 1.0, 0.0);
        glVertex3f(0.0, h, 0.0);

        glNormal3f(0.0, 1.0, 0.0);
        glVertex3f(r2 * sin(Angle * 3.1415926 / 180.0), h,
            r2 * cos(Angle * 3.1415926 / 180.0));

        glNormal3f(0.0, 1.0, 0.0);
        glVertex3f(r2 * sin((Angle + 360.0 / n) * 3.1415926 / 180.0),
h,
            r2 * cos((Angle + 360.0 / n) * 3.1415926 / 180.0));

        glEnd();

        glBegin(GL_QUADS);

        glNormal3f(r1 * sin((Angle + 180.0 / n) * 3.1415926 / 180.0),
r1 * (r2 - r1) / h,
            r1 * cos((Angle + 180.0 / n) * 3.1415926 / 180.0));
        glVertex3f(r1 * sin(Angle * 3.1415926 / 180.0), 0.0,
            r1 * cos(Angle * 3.1415926 / 180.0));

        glNormal3f(r1 * sin((Angle + 180.0 / n) * 3.1415926 / 180.0),
r1 * (r2 - r1) / h,
            r1 * cos((Angle + 180.0 / n) * 3.1415926 / 180.0));
        glVertex3f(r1 * sin((Angle + 360.0 / n) * 3.1415926 / 180.0),
0.0,
            r1 * cos((Angle + 360.0 / n) * 3.1415926 / 180.0));

        glNormal3f(r1 * sin((Angle + 180.0 / n) * 3.1415926 / 180.0),
r1 * (r2 - r1) / h,
            r1 * cos((Angle + 180.0 / n) * 3.1415926 / 180.0));
        glVertex3f(r2 * sin((Angle + 360.0 / n) * 3.1415926 / 180.0),
h,
```

```
            r2 * cos((Angle + 360.0 / n) * 3.1415926 / 180.0));

        glNormal3f(r1 * sin((Angle + 180.0 / n) * 3.1415926 / 180.0),
r1 * (r2 - r1) / h,
            r1 * cos((Angle + 180.0 / n) * 3.1415926 / 180.0));
        glVertex3f(r2 * sin(Angle * 3.1415926 / 180.0), h,
            r2 * cos(Angle * 3.1415926 / 180.0));

        glEnd();

        Angle += 360.0 / n;
    }
}
```

多面棱台：

代码实现：

```
void GEO::drawPrism(GLfloat h, GLfloat r, int n)
{

    if (vArr)
        delete[] vArr;
    vArr = new GLfloat *[2 + 2 * n];
    fvArr = new int *[3 * n];
    vnArr = new GLfloat *[2 + n];
    fnArr = new int *[3 * n];
    v_num = 2 + 2 * n;
    vn_num = 2 + n;
    f_num = 3 * n;
    int i;
    double Angle = 0;
    for (i = 0; i < 2 + 2 * n; i++)
        vArr[i] = new GLfloat[3];
    for (i = 0; i < 3 * n; i++)
    {
        fvArr[i] = new int[5];
        fnArr[i] = new int[5];
    }
    for (i = 0; i < n + 2; i++)
        vnArr[i] = new GLfloat[3];

    vArr[0][0] = vArr[0][1] = vArr[0][2] = vArr[n + 1][0] = vArr[n +
1][2] = 0;
    vArr[n + 1][1] = h;
```

```
    for (i = 0; i < n; i++)
    {
        vArr[i + 1][0] = vArr[i + n + 2][0] = r * sin(Angle *
3.1415926 / 180.0);
        vArr[i + 1][1] = 0;
        vArr[i + n + 2][1] = h;
        vArr[i + 1][2] = vArr[i + n + 2][2] = r * cos(Angle *
3.1415926 / 180.0);
        Angle += 360.0 / n;

        fvArr[i][0] = fvArr[i + n][0] = 3;
        fvArr[i][1] = 1;
        fvArr[i + n][1] = n + 2;
        fvArr[i][2] = i + 2;
        fvArr[i][3] = (i + 1) % n + 2;
        fvArr[i + n][2] = i + n + 3;
        fvArr[i + n][3] = (i + 1) % n + n + 3;

        fvArr[i + 2 * n][0] = 4;
        fvArr[i + 2 * n][1] = i + 2;
        fvArr[i + 2 * n][2] = (i + 1) % n + 2;
        fvArr[i + 2 * n][3] = (i + 1) % n + n + 3;
        fvArr[i + 2 * n][4] = i + n + 3;

        fnArr[i][0] = fnArr[i + n][0] = 3;
        fnArr[i][1] = fnArr[i][2] = fnArr[i][3] = 1;
        fnArr[i + n][1] = fnArr[i + n][2] = fnArr[i + n][3] = 2;
        fnArr[i + 2 * n][1] = fnArr[i + 2 * n][2] = fnArr[i + 2 *
n][3] = fnArr[i + 2 * n][4] = i + 3;
    }
    vnArr[0][0] = vnArr[0][2] = vnArr[1][0] = vnArr[1][2] = 0;
    vnArr[0][1] = -1.0;
    vnArr[1][1] = 1.0;
    for (i = 2; i < n + 2; i++)
    {
        Angle = (i - 2.0) * 360.0 / n + 180.0 / n;
        vnArr[i][0] = sin(Angle * 3.1415926 / 180.0);
        vnArr[i][1] = 0.0;
        vnArr[i][2] = cos(Angle * 3.1415926 / 180.0);
    }
    for (int i = 0; i < 3 * n; i++)
    {
        if (fvArr[i][0] == 3)
        {
```

```
            glBegin(GL_TRIANGLES);

            glNormal3f(vnArr[fnArr[i][1] - 1][0], vnArr[fnArr[i][1] -
1][1],
                vnArr[fnArr[i][1] - 1][2]);
            glVertex3f(vArr[fvArr[i][1] - 1][0], vArr[fvArr[i][1] -
1][1],
                vArr[fvArr[i][1] - 1][2]);

            glNormal3f(vnArr[fnArr[i][2] - 1][0], vnArr[fnArr[i][2] -
1][1],
                vnArr[fnArr[i][2] - 1][2]);
            glVertex3f(vArr[fvArr[i][2] - 1][0], vArr[fvArr[i][2] -
1][1],
                vArr[fvArr[i][2] - 1][2]);

            glNormal3f(vnArr[fnArr[i][3] - 1][0], vnArr[fnArr[i][3] -
1][1],
                vnArr[fnArr[i][3] - 1][2]);
            glVertex3f(vArr[fvArr[i][3] - 1][0], vArr[fvArr[i][3] -
1][1],
                vArr[fvArr[i][3] - 1][2]);

            glEnd();
        }
        else
        {
            glBegin(GL_QUADS);

            glNormal3f(vnArr[fnArr[i][1] - 1][0], vnArr[fnArr[i][1] -
1][1],
                vnArr[fnArr[i][1] - 1][2]);
            glVertex3f(vArr[fvArr[i][1] - 1][0], vArr[fvArr[i][1] -
1][1],
                vArr[fvArr[i][1] - 1][2]);

            glNormal3f(vnArr[fnArr[i][2] - 1][0], vnArr[fnArr[i][2] -
1][1],
                vnArr[fnArr[i][2] - 1][2]);
            glVertex3f(vArr[fvArr[i][2] - 1][0], vArr[fvArr[i][2] -
1][1],
                vArr[fvArr[i][2] - 1][2]);
```

```
            glNormal3f(vnArr[fnArr[i][3] - 1][0], vnArr[fnArr[i][3] -
1][1],
                vnArr[fnArr[i][3] - 1][2]);
            glVertex3f(vArr[fvArr[i][3] - 1][0], vArr[fvArr[i][3] -
1][1],
                vArr[fvArr[i][3] - 1][2]);

            glNormal3f(vnArr[fnArr[i][4] - 1][0], vnArr[fnArr[i][4] -
1][1],
                vnArr[fnArr[i][4] - 1][2]);
            glVertex3f(vArr[fvArr[i][4] - 1][0], vArr[fvArr[i][4] -
1][1],
                vArr[fvArr[i][4] - 1][2]);
            glEnd();
        }
    }
}
```

2.1.2 基本三维网格导入导出

本项目建立 OBJ 类来存储导入导出工程的 OBJ 文件信息，并将模型与相应的贴图绑定。

OBJ 文件储存格式

OBJ 文件每一行的格式如下：
* 前缀 参数 1 参数 2 参数 3
*
* v（顶点）X Y Z（坐标值）
* vt（纹理坐标） U Y（值）*
* vn（法线向量） X Y Z（坐标值）
* f（表面）具体如下：
* f 1 2 3 表示以第 1、2、3 号顶点组成一个三角形面
* f 1/3 2/5 3/4 表示同上，同时其顶点的纹理坐标的索引值分别为 3、5、4
* f 1/3/4 2/5/6 3/4/2 表示同上，同时其顶点的法线索引值分别为 4、6、2
* f 1//4 2//6 3//2 则表示忽略纹理坐标
* 注意：纹理坐标从 1 开始记录，而不是从 0 开始

obj 中存放的是顶点坐标信息（v）,面的信息（f）,法线（vn），纹理坐标（vt）。其中坐标信息（v）、法线（vn）由三个数字表示，纹理坐标（vt）由两个数字表示，面的信息（f）由四个数字表示。

OBJ 模型导入

按照 obj 文件的数据格式读入，首先遍历一遍 obj 文件得到各个顶点数据的数量以及面的数量，然后为数据分配内存，逐个读入到数组中。

代码实现：

```
int OBJ::readFile() //将文件内容读到数组中去
{
    if (EmptyOBJ)
    {
        //cout << "Empty Class" << endl;
        return -1;
    }
    string s1, s2;
    GLfloat f1, f2, f3;
    int i;
    // vArr[i][0]储存该面的边数
    // 初始化储存点的数组为 v_num * 3 大小的二维数组
    vArr = new GLfloat *[v_num];
    for (i = 0; i < v_num; i++)
    {
        vArr[i] = new GLfloat[3];
        vArr[i][0] = vArr[i][1] = vArr[i][2] = 0;
    }
    // 初始化储存法线的数组为 vn_num * 3 大小的二维数组
    vnArr = new GLfloat *[vn_num];
    for (i = 0; i < vn_num; i++)
    {
        vnArr[i] = new GLfloat[3];
        vnArr[i][0] = vnArr[i][1] = vnArr[i][2] = 0;
    }
    // 初始化储存UV的数组为 vn_num * 2 大小的二维数组
    vtArr = new GLfloat *[vt_num];
    for (i = 0; i < vt_num; i++)
    {
        vtArr[i] = new GLfloat[2];
        vtArr[i][0] = vtArr[i][1] = 0;
    }
    // 初始化储存面顶点和法线和UV的数组为 f_num * 5 大小的二维数组
    fvArr = new int*[f_num];
    fnArr = new int*[f_num];
    ftArr = new int*[f_num];
    for (i = 0; i < f_num; i++)
```

```
    {
        fvArr[i] = new int[5];
        fnArr[i] = new int[5];
        ftArr[i] = new int[5];
        for (int j = 0; j < 5; j++)
        {
            fvArr[i][j] = fnArr[i][j] = ftArr[i][j] = 0;
        }
    }
    int N = v_num + vn_num + vt_num + f_num;
    int count = 0;
    // 打开文件
    ifstream infile(sFileName.c_str());
    string sline;//每一行
    // ii法线编号 jj顶点编号 vvUV编号 kk面编号
    int ii = 0, jj = 0, vv = 0, kk = 0;
    while (getline(infile, sline))
    {
        if (sline[0] == 'v')
        {
            if (sline[1] == 'n')
            { // 处理法向量的读入
                istringstream sin(sline);
                sin >> s1 >> f1 >> f2 >> f3;
                vnArr[ii][0] = f1;
                vnArr[ii][1] = f2;
                vnArr[ii][2] = f3;
                ii++;
            }
            else if (sline[1] == ' ')
            { // 处理顶点的读入
                istringstream sin(sline);
                sin >> s1 >> f1 >> f2 >> f3;
                vArr[jj][0] = f1;
                vArr[jj][1] = f2;
                vArr[jj][2] = f3;
                jj++;
            }
            else if (sline[1] == 't')
            { // 处理UV的读入
                istringstream sin(sline);
                sin >> s1 >> f1 >> f2;
                vtArr[vv][0] = f1;
                vtArr[vv][1] = f2;
```

```
            vv++;
        }
        count++;
        displayProgress(N, count);
    }
    if (sline[0] == 'f')
    { // 处理面的读入
        istringstream in(sline);
        GLfloat a;
        in >> s1; // 去掉前缀f
        int i, k;
        for (i = 1; i < 5; i++)
        {
            if (i == 4)
            {
                if (in.eof())
                {
                    fvArr[kk][0] = 3;
                    fnArr[kk][0] = 3;
                    ftArr[kk][0] = 3;
                    break;
                }
                else
                {
                    fvArr[kk][0] = 4;
                    fnArr[kk][0] = 4;
                    ftArr[kk][0] = 4;
                }
            }
            in >> s1;
            //cout << s1 << ' ' << endl;
            // 取得顶点索引和法线索引
            a = 0;
            // 获取顶点的坐标索引，注意索引从1开始编号
            for (k = 0; s1[k] != '/'; k++)
            {
                a = a * 10 + (s1[k] - 48);
            }
            fvArr[kk][i] = a;
            a = 0;
            // 跳过纹理UV部分
            for (k = k + 1; s1[k] != '/'; k++)
            {
                a = a * 10 + (s1[k] - 48);
```

```
                }
                ftArr[kk][i] = a;
                a = 0;
                // 获取顶点的法线索引，注意索引从1开始编号
                for (k = k + 1; s1[k]; k++)
                {
                    a = a * 10 + (s1[k] - 48);
                }
                fnArr[kk][i] = a;
            }
            kk++;
            count++;
            displayProgress(N, count);
        }
    }
    return 0;
}
```

为提高读取速度，将 obj 文件转成 data 格式文件进行数据读入，可大大提高读取效率。这里 data 文件的格式为:

[顶点坐标数量] [法线向量数量] [UV 坐标数量] [面数量]
[...具体数据...]

读取 data 格式文件的代码实现：

```
void OBJ::ReadCompact(const string& filename)
{
    int count = 0;
    sFileName = filename;
    ifstream file(filename, ios::in);
    if (!file.is_open())
    {
        cout << "**File Open Failed!**[" << filename << ']' << endl;
        return;
    }
    file >> v_num >> vn_num >> vt_num >> f_num;
    int N = v_num + vn_num + vt_num + f_num;
    int i;
    if (!EmptyOBJ)
    {
        for (i = 0; i < v_num; i++)
            delete[] vArr[i];
        delete[] vArr;
```

```
        for (i = 0; i < vn_num; i++)
            delete[] vnArr[i];
        delete[] vnArr;
        for (i = 0; i < f_num; i++)
        {
            delete[] fvArr[i];
            delete[] fnArr[i];
        }
        delete[] fvArr;
        delete[] fnArr;
    }
    // vArr[i][0]储存该面的边数
    // 初始化储存点的数组为 v_num * 3 大小的二维数组
    vArr = new GLfloat * [v_num];
    for (i = 0; i < v_num; i++)
    {
        vArr[i] = new GLfloat[3];
    }
    // 初始化储存法线的数组为 vn_num * 3 大小的二维数组
    vnArr = new GLfloat * [vn_num];
    for (i = 0; i < vn_num; i++)
    {
        vnArr[i] = new GLfloat[3];
    }
    // 初始化储存UV的数组为 vn_num * 2 大小的二维数组
    vtArr = new GLfloat * [vt_num];
    for (i = 0; i < vt_num; i++)
    {
        vtArr[i] = new GLfloat[2];
    }
    // 初始化储存面顶点和法线和UV的数组为 f_num * 5 大小的二维数组
    fvArr = new int* [f_num];
    fnArr = new int* [f_num];
    ftArr = new int* [f_num];
    for (i = 0; i < f_num; i++)
    {
        fvArr[i] = new int[5];
        fnArr[i] = new int[5];
        ftArr[i] = new int[5];
    }
    for (i = 0; i < v_num; i++)
    {
        file >> vArr[i][0] >> vArr[i][1] >> vArr[i][2];
        count++;
```

```
        displayProgress(N, count);
    }
    for (i = 0; i < vn_num; i++)
    {
        file >> vnArr[i][0] >> vnArr[i][1] >> vnArr[i][2];
        count++;
        displayProgress(N, count);
    }
    for (i = 0; i < vt_num; i++)
    {
        file >> vtArr[i][0] >> vtArr[i][1];
        count++;
        displayProgress(N, count);
    }
    for (i = 0; i < f_num; i++)
    {
        file >> fvArr[i][0] >> fvArr[i][1] >> fvArr[i][2] >>
fvArr[i][3] >> fvArr[i][4];
        file >> fnArr[i][0] >> fnArr[i][1] >> fnArr[i][2] >>
fnArr[i][3] >> fnArr[i][4];
        file >> ftArr[i][0] >> ftArr[i][1] >> ftArr[i][2] >>
ftArr[i][3] >> ftArr[i][4];
        count++;
        displayProgress(N, count);
    }
    EmptyOBJ = false;
    cout << "[" << Num << "][OBJ] - Read obj file: " << sFileName <<
" succeed!" << endl;
    cout << "[" << Num << "][OBJ] - v_num: " << v_num << ", vn_num: "
<< vn_num << ", vt_num: " << vt_num << ", f_num: " << f_num << endl;
}
```

OBJ 模型导出

将模型的顶点坐标信息、法线信息、面信息，纹理坐标信息等数据按照 obj 文件数据格式读入到新文件中。

代码实现：

```
void GEO::outputObj(const string& filename)
{
    ofstream file(filename, ios::out | ios::trunc);
    if (!file.is_open())
    {
```

```
        cout << "File Create Failed!" << endl;
        return;
    }
    file << "# This obj file is generated by geo.h" << endl;
    int i, j;
    for (i = 0; i < v_num; i++)
        file << "v " << vArr[i][0] << ' ' << vArr[i][1] << ' ' <<
vArr[i][2] << endl;
    for (i = 0; i < vn_num; i++)
        file << "vn " << vnArr[i][0] << ' ' << vnArr[i][1] << ' ' <<
vnArr[i][2] << endl;
    file << "g prism" << endl;
    for (i = 0; i < f_num; i++)
    {
        file << "f ";
        for (j = 0; j < fvArr[i][0]; j++)
        {
            file << fvArr[i][j + 1] << "//" << fnArr[i][j + 1] << ' ';
        }
        file << endl;
    }
    file.close();
}
```

2.1.3 基本材质、纹理的显示和编辑

贴图导入

以二进制格式打开 bmp 贴图文件，读入文件头，信息头等数据。此部分信息实现在 OBJ 类中，封装成为一个整体，无需单独设置。

由于不能在 openGL 初始化前就生成贴图（在实际操作中遇到了许多问题），因此贴图和 openGL 的绑定需要在 openGL 初始化之后进行，也就是不能在 init()函数以及 main()函数前进行贴图绑定。

并且，在实践中遇到了 32 位的 bmp 文件，以及 biSizeImage 有误的 bmp 文件，因此在该函数中一并处理了以上两种情况。

代码实现：

```
void OBJ::LoadBitmapFile()
{
    FILE* filePtr;                          // 文件指针
    BITMAPFILEHEADER bitmapFileHeader; // bitmap文件头
    int imageIdx = 0;                  // 图像位置索引
    unsigned char tempRGB;             // 交换变量
```

```
    //unsigned char* bitmapImageR;

    // 以"二进制+读"模式打开文件filename
    filePtr = fopen(sTextureName.c_str(), "rb");
    if (filePtr == NULL)
    {
        std::cout << "Such file does not exist!" << std::endl;
        EmptyTEXT = true;
        return;
    }

    // 读入bitmap文件图
    fread(&bitmapFileHeader, sizeof(BITMAPFILEHEADER), 1, filePtr);
    // 验证是否为bitmap文件
    if (bitmapFileHeader.bfType != BITMAP_ID) {
        std::cout << stderr << ", Error in LoadBitmapFile: the file is
not a bitmap file" << std::endl;
        EmptyTEXT = true;
        return;
    }

    // 读入bitmap信息头
    fread(&bitmapInfoHeader, sizeof(BITMAPINFOHEADER), 1, filePtr);
    // 将文件指针移至bitmap数据
    fseek(filePtr, bitmapFileHeader.bfOffBits, SEEK_SET);
    // 为装载图像数据创建足够的内存
    bitmapInfoHeader.biSizeImage = bitmapInfoHeader.biHeight *
bitmapInfoHeader.biWidth * bitmapInfoHeader.biBitCount / 8;
    bitmapImage = new unsigned char[bitmapInfoHeader.biSizeImage];
    // 验证内存是否创建成功
    if (!bitmapImage) {
        std::cout << stderr << ", Error in LoadBitmapFile: memory
error" << std::endl;
        EmptyTEXT = true;
        return;
    }

    // 读入bitmap图像数据
    fread(bitmapImage, 1, bitmapInfoHeader.biSizeImage, filePtr);
    // 确认读入成功
    if (bitmapImage == NULL) {
        std::cout << stderr << ", Error in LoadBitmapFile: memory
error" << std::endl;
        EmptyTEXT = true;
```

```
        return;
    }
    if (bitmapInfoHeader.biBitCount == 32)
    {
        //std::cout << "Reading 32 bitCount Image ..." << std::endl;
        unsigned char* bitmapImageReal;
        bitmapImageReal = new unsigned
char[(bitmapInfoHeader.biSizeImage * 3 / 4)];
        // 验证内存是否创建成功
        if (!bitmapImageReal) {
            std::cout << stderr << ", Error in LoadBitmapFile: memory
error" << std::endl;
            EmptyTEXT = true;
            return;
        }
        int imageIdxReal = 0;
        for (imageIdx = 0;
            imageIdx < bitmapInfoHeader.biSizeImage; imageIdx += 4) {
            bitmapImageReal[imageIdxReal] = bitmapImage[imageIdx + 2];
            bitmapImageReal[imageIdxReal + 1] = bitmapImage[imageIdx +
1];
            bitmapImageReal[imageIdxReal + 2] = bitmapImage[imageIdx];
            imageIdxReal += 3;
        }
        // 关闭bitmap图像文件
        fclose(filePtr);
        free(bitmapImage);
        bitmapImage = bitmapImageReal;
    }
    else if (bitmapInfoHeader.biBitCount == 24)
    {
        //std::cout << "Reading 24 bitCount Image ..." << std::endl;
        //由于bitmap中保存的格式是BGR，下面交换R和B的值，得到RGB格式
        for (imageIdx = 0;
            imageIdx < bitmapInfoHeader.biSizeImage; imageIdx += 3) {
            tempRGB = bitmapImage[imageIdx];
            bitmapImage[imageIdx] = bitmapImage[imageIdx + 2];
            bitmapImage[imageIdx + 2] = tempRGB;
        }
        // 关闭bitmap图像文件
        fclose(filePtr);
    }
    return;
}
```

贴图绑定

在 OpenGL 中创建纹理，设置当前纹理参数。绑定贴图由`glBindTexture(GL_TEXTURE_2D, tex_id)`函数来完成，同时通过`glNormal3f(GLfloatnx,GLfloatny,GLfloatnz)`函数来设置法线方向，`glVertex3f(GLfloatnx,GLfloatny,GLfloatnz)`函数绘制三角面。

代码实现：

```
void OBJ::texload()
{
    LoadBitmapFile();
    glGenTextures(1, &tex);
    glBindTexture(GL_TEXTURE_2D, tex);
    // 指定当前纹理的放大/缩小过滤方式
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D,
        0,     //mipmap层次(通常为，表示最上层)
        GL_RGB,    //我们希望该纹理有红、绿、蓝数据
        bitmapInfoHeader.biWidth, //纹理宽带，必须是n，若有边框+2
        bitmapInfoHeader.biHeight, //纹理高度，必须是n，若有边框+2
        0, //边框(0=无边框, 1=有边框)
        GL_RGB,    //bitmap数据的格式
        GL_UNSIGNED_BYTE, //每个颜色数据的类型
        bitmapImage); //bitmap数据指针
    EmptyTEXT = false;
    cout << "[" << Num << "][TEXT] - Reading Texture: " <<
sTextureName << "succeed! [" << bitmapInfoHeader.biWidth << " * " <<
bitmapInfoHeader.biHeight << "]" << endl;
}
```

贴图编辑

使用传入参数 alpha 控制两张贴图叠加的比例，遍历两张贴图数据，alpha 分配混合贴图中两张贴图的权重，总权重为 1。

代码实现：

```
void OBJ::texMix(float alpha)
{
    if (bitmapImageMix == NULL)
```

```
        bitmapImageMix = new unsigned char[bitmapInfoHeader2.biWidth *
bitmapInfoHeader2.biHeight * 3];
    int i;
    for (i = 0; i < bitmapInfoHeader2.biWidth *
bitmapInfoHeader2.biHeight * 3; i++)
    {
        bitmapImageMix[i] = bitmapImage[i] * alpha + bitmapImage2[i] *
(1.0 - alpha);
    }
    glBindTexture(GL_TEXTURE_2D, tex);
    // 指定当前纹理的放大/缩小过滤方式
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D,
        0,     //mipmap层次(通常为，表示最上层)
        GL_RGB,    //我们希望该纹理有红、绿、蓝数据
        bitmapInfoHeader2.biWidth, //纹理宽带，必须是n，若有边框+2
        bitmapInfoHeader2.biHeight, //纹理高度，必须是n，若有边框+2
        0, //边框(0=无边框, 1=有边框)
        GL_RGB,    //bitmap数据的格式
        GL_UNSIGNED_BYTE, //每个颜色数据的类型
        bitmapImageMix); //bitmap数据指针
    cout << "[" << Num << "][TEXTMIX]" << bitmapInfoHeader2.biWidth
<< " * " << bitmapInfoHeader2.biHeight << "]" << endl;
}
```

2.1.4 基本几何变换

平移，旋转，缩放等基本几何变换内嵌于程序的很多部分，没有单独实现，现以 NPC 的 chase 移动模式为例。

通过计算得到NPC的移动方向，头部的旋转角度，再结合obj文件本身的大小，缩放到适配当前场景的大小，通过 `glTranslatef(v_rand.x, v_rand.y, v_rand.z)`，`glScalef(v_Scale.x, v_Scale.y, v_Scale.z)`，`glRotatef(Angle, 0.0, 1.0, 0.0)` 函数实现几何变换。

代码实现：

```
void NPC::Chase(float x1, float x2, float z1, float z2, float x,
float y, float z, float speed)
{
    int Pace = 400;
    int R = rand();
```

```
    Para3 += 1;
    //Para4 += 1;
    if (Para3 > Pace) Para3 = 0;
    switch ((int)Para3 / (Pace / 4))
    {
    case 0:
        v_rand.y += 0.005;
        Para4 -= 0.4;
        break;
    case 1:
        Para4 += 0.4;
        break;
    case 2:
        v_rand.y -= 0.005;
        Para4 += 0.4;
        break;
    case 3:
        Para4 -= 0.4;
        break;
    default:
        break;
    }
    Para1 = (int)(x - v_rand.x);
    Para2 = (int)(z - v_rand.z);
    float Speed = getSpeed();
    float Norm = pow(Para1 * Para1 + Para2 * Para2, 0.5) / speed;
    if (v_rand.x + Para1 / Norm * Speed - v_position.x > x1 &&
v_rand.x + Para1 / Norm * Speed - v_position.x < x2 &&
        v_rand.z + Para2 / Norm * Speed - v_position.z > z1 &&
v_rand.z + Para2 / Norm * Speed - v_position.z < z2)
    {
        v_rand.x += Para1 / Norm * Speed;
        v_rand.z += Para2 / Norm * Speed;
    }
    float Angle = atan((float)Para1 / Para2) * 180 / 3.1415926;
    if (Para2 < 0) Angle += 180;
    Angle = (int)Angle / 10 * 10;
    glPushMatrix();
    glTranslatef(v_rand.x, v_rand.y, v_rand.z);
    glScalef(v_Scale.x, v_Scale.y, v_Scale.z);
    glRotatef(Angle, 0.0, 1.0, 0.0);
    int i;
    for (i = 0; i < 3; i++)
        obj[i].draw();
```

```
    glRotatef(Para4, 0.0, 1.0, 0.0);
    for (i = 3; i < 5; i++)
        obj[i].draw();
    glPopMatrix();
}
```

2.1.5 基本光照模型要求

全局环境光

在 main 函数中设置白色环境光全局光源，键盘 L 可控制光照的开关。

代码实现：

```
    GLfloat white[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_pos[] = { 0.0, 10.0, 0.0, 1.0 };
    glLightfv(GL_LIGHT0, GL_POSITION, light_pos);
    glLightfv(GL_LIGHT0, GL_AMBIENT, white);
    if (light_on)
        glEnable(GL_LIGHT0);
    else
        glDisable(GL_LIGHT0);
```

手电筒（聚光灯）

spot 嵌于 camera 类中，可以随摄像机漫游。spot 的位置坐标和摄像机保持一致，朝向与摄像机朝向有一个小角度，模拟右手拿手电筒的效果，键盘控制手电筒光源的开闭和颜色变化，光源移动则和摄像机保持一致

代码实现：

```
void Camera::SpotSet() {

    float x = m_Position.x;
    float y = m_Position.y;
    float z = m_Position.z;
    float viewx = m_View.x;
    float viewy = m_View.y;
    float viewz = m_View.z;
    float dd = 10;
    double Distance = pow(pow(viewx - x, 2) + pow(viewy - y, 2) +
pow(viewz - z, 2), 0.5);
    double ddx = dd * ((double)viewx - x) / Distance;
    double ddy = dd * ((double)viewy - y) / Distance;
```

```
    double ddz = dd * ((double)viewz - z) / Distance;

    GLfloat spot_pos[] = { 1.0,1.0,1.0,1.0 };
    spot_pos[0] = x;
    spot_pos[1] = y;
    spot_pos[2] = z;

    GLfloat spot_direction[3];
    spot_direction[0] = ddx;
    spot_direction[1] = ddy;
    spot_direction[2] = ddz;
    GLfloat light_color[] = { 1.0,1.0,1.0,1.0 };
    if (lightcolor == true) {
        light_color[0] = 1.0;
        light_color[1] = 0.0;
        light_color[2] = 0.0;
        light_color[3] = 1.0;
    }
    GLfloat white[] = { 1.0,1.0,1.0,1.0 };
    GLfloat Angle = 15.0f;

    glLightfv(GL_LIGHT1, GL_AMBIENT, light_color);
    glLightfv(GL_LIGHT1, GL_SPECULAR, white); //设置镜面光成分
    glLightfv(GL_LIGHT1, GL_DIFFUSE, white); //设置漫射光成分

    glLightfv(GL_LIGHT1, GL_SPOT_DIRECTION, spot_direction); //光源方向
    glLightfv(GL_LIGHT1, GL_POSITION, spot_pos);
    glLightf(GL_LIGHT1, GL_SPOT_CUTOFF, Angle); //裁减角度
    glLightf(GL_LIGHT1, GL_SPOT_EXPONENT, 2.0); //聚集度

    glEnable(GL_LIGHT1);
    glEnable(GL_LIGHTING);
}
```

2.1.6 能对建模后场景进行漫游

为以第一人称对场景进行漫游，建立 Camera 类，用 Camera 在场景中的位置的移动，朝向的变化模拟第一人称漫游。

Camera 基本参数

```
    Vector3       m_Position;     /* 位置 */
    Vector3       m_View;        /* 朝向 */
    Vector3       m_UpVector;     /* 向上向量 */
```

场景漫游移动

键盘控制摄像机前进后退和左右移动，改变摄像机位置向量，同时更新摄像机朝向向量

代码实现：

```
/** 左右移动摄像机 */
void Camera::yawCamera(float speed)
{
    Vector3 Cross = m_UpVector.crossProduct(m_View - m_Position);
    Cross = Cross.normalize();

    m_Position.x += Cross.x * speed * JumpSpeed * DashSpeed;
    m_Position.z += Cross.z * speed * JumpSpeed * DashSpeed;
    m_View.x += Cross.x * speed * JumpSpeed * DashSpeed;
    m_View.z += Cross.z * speed * JumpSpeed * DashSpeed;
    if (CollisionDetect())
    {
        /* 回滚 */
        m_Position.x -= Cross.x * speed * JumpSpeed * DashSpeed;
        m_Position.z -= Cross.z * speed * JumpSpeed * DashSpeed;
        m_View.x -= Cross.x * speed * JumpSpeed * DashSpeed;
        m_View.z -= Cross.z * speed * JumpSpeed * DashSpeed;
    }
}
/** 前后移动摄像机 */
void Camera::moveCamera(float speed)
{
    Vector3 vector = m_View - m_Position;
    vector = vector.normalize();
    m_Position.x += vector.x * speed * JumpSpeed * DashSpeed;
    m_Position.z += vector.z * speed * JumpSpeed * DashSpeed;
    m_View.x += vector.x * speed * JumpSpeed * DashSpeed;
    m_View.z += vector.z * speed * JumpSpeed * DashSpeed;
    if (CollisionDetect())
    {
        /* 回滚 */
        m_Position.x -= vector.x * speed * JumpSpeed * DashSpeed;
        m_Position.z -= vector.z * speed * JumpSpeed * DashSpeed;
        m_View.x -= vector.x * speed * JumpSpeed * DashSpeed;
        m_View.z -= vector.z * speed * JumpSpeed * DashSpeed;
    }
}
```

场景漫游旋转

鼠标位置变化控制摄像机朝向的变化，获得屏幕上鼠标的移动向量，分解到 Y 轴和 Z 轴上，按照角度进行左右或上下的旋转。

代码实现：

```
/**  旋转摄像机方向  */
void Camera::rotateView(float angle, float x, float y, float z)
{
    Vector3 newView;
    /** 计算方向向量 */
    Vector3 view = m_View - m_Position;

    /** 计算 sin 和cos值 */
    float cosTheta = (float)cos(angle);
    float sinTheta = (float)sin(angle);

    /** 计算旋转向量的x值 */
    newView.x = (cosTheta + (1 - cosTheta) * x * x) * view.x;
    newView.x += ((1 - cosTheta) * x * y - z * sinTheta) * view.y;
    newView.x += ((1 - cosTheta) * x * z + y * sinTheta) * view.z;

    /** 计算旋转向量的y值 */
    newView.y = ((1 - cosTheta) * x * y + z * sinTheta) * view.x;
    newView.y += (cosTheta + (1 - cosTheta) * y * y) * view.y;
    newView.y += ((1 - cosTheta) * y * z - x * sinTheta) * view.z;

    /** 计算旋转向量的z值 */
    newView.z = ((1 - cosTheta) * x * z - y * sinTheta) * view.x;
    newView.z += ((1 - cosTheta) * y * z + x * sinTheta) * view.y;
    newView.z += (cosTheta + (1 - cosTheta) * z * z) * view.z;

    /** 更新摄像机的方向 */
    m_View = m_Position + newView;
}
bool Camera::GetMouseMoveState() {
    return this->SeeMouseMove;
}

/** 鼠标控制旋转 */
void Camera::setViewByMouse()
{
    POINT mousePos;                                 /**< 保存当前鼠标位置 */
    int middleX = GetSystemMetrics(SM_CXSCREEN) >> 1;
```

```
    int middleY = GetSystemMetrics(SM_CYSCREEN) >> 1;
    float angleY = 0.0f;
    float angleZ = 0.0f;
    static float currentRotX = 0.0f;

    /** 得到当前鼠标位置 */
    GetCursorPos(&mousePos);
    ShowCursor(false);
    this->count--;
    /** 如果鼠标没有移动,则不用更新 */
    if ((mousePos.x == middleX) && (mousePos.y == middleY))
        return;
    /** 设置鼠标位置在屏幕中心 */
    SetCursorPos(middleX, middleY);
    /**< 得到鼠标移动方向 */
    angleY = (float)((middleX - mousePos.x)) / ANGLE_Y;
    angleZ = (float)((middleY - mousePos.y)) / ANGLE_Z;

    static float lastRotX = 0.0f;      /**< 用于保存旋转角度 */
    lastRotX = currentRotX;

    /** 跟踪摄像机上下旋转角度 */
    currentRotX += angleZ;

    /** 找到与旋转方向垂直向量 */
    Vector3 vAxis = m_View - m_Position;
    vAxis = vAxis.crossProduct(m_UpVector);
    vAxis = vAxis.normalize();

    ///旋转
    rotateView(angleZ, vAxis.x, vAxis.y, vAxis.z);

    /** 总是左右旋转摄像机 */
    rotateView(angleY, 0, 1, 0);
}
```

开镜

鼠标右键开镜，利用摄像机前移的原理实现模拟视野放大的效果

代码实现：

```
void Camera::setLook()
{
    if (zoom) {
```

```
        /** 计算方向向量 */
        Vector3 vector = m_View - m_Position;
        vector = vector.normalize();
        Vector3 pos, view;
        /** 更新摄像机 */
        pos.x = m_Position.x + vector.x * ZOOM_SIZE;
        pos.z = m_Position.z + vector.z * ZOOM_SIZE;
        pos.y = m_Position.y;
        view.x = m_View.x + vector.x * ZOOM_SIZE;
        view.z = m_View.z + vector.z * ZOOM_SIZE;
        view.y = m_View.y;
        gluLookAt(pos.x, pos.y, pos.z,
            view.x, view.y + shooting * BounceHeight, view.z,
            m_UpVector.x, m_UpVector.y, m_UpVector.z);
    }
    else {
        gluLookAt(m_Position.x, m_Position.y, m_Position.z,
            m_View.x, m_View.y + shooting * BounceHeight, m_View.z,
            m_UpVector.x, m_UpVector.y, m_UpVector.z);
    }
}
```

2.1.7 能够提供动画播放功能（多帧数据连续绘制），能够提供屏幕截取/保存功能

动画播放

实验中我们有两种动画：玻璃破碎动画、皮卡丘运动动画。两个动画播放的实现都依靠读取多帧 obj 连续绘制。

- 窗子破碎

读入 41 帧窗子 obj 文件，在 redraw 函数中用 num 计数器和 bool 型变量连续绘制

没有打碎窗子时，绘制第一帧窗子，窗子打碎时连续绘制，窗子打碎后绘制最后一帧窗子。

代码实现：

```
    if (broken && num < 41) {
        windows[num++].draw();
    }
    else
        if (num == 0)
```

```
            windows[0].draw();
        else if (num >= 41)
            windows[40].draw();
```

- 皮卡丘行走
  读入 4 帧皮卡丘 obj 文件，连续循环绘制，形成行走动画。

代码实现：

```
for (i = 0; i < 3; i++)
        obj[i].draw();
    switch (((int)Para3/4) % 4)
    {
    case 0:
        obj[3].draw();
        obj[4].draw();
        break;
    case 1:
        obj[5].draw();
        obj[6].draw();
        break;
    case 2:
        obj[7].draw();
        obj[8].draw();
        break;
    case 3:
        obj[5].draw();
        obj[6].draw();
        break;
    default:
        break;
    }
```

屏幕截取/保存

把当前屏幕的信息写入新文件，并指定路径保存，传入参数为所截取的长宽，按 m 键截取屏幕，截取后的图像会保存在根目录的 Screenshot 文件夹内。

代码实现：

```
bool SnapScreen(int width, int height)
{
    FILE *fp;           //文件指针
```

```
    unsigned char *imgdata;
    BITMAPINFOHEADER bi;
    BITMAPFILEHEADER bf;
    bi.biBitCount = 24;
    bi.biClrImportant = 0;
    bi.biClrUsed = 0;
    bi.biCompression = 0;
    bi.biHeight = height;
    bi.biPlanes = 1;
    bi.biSize = 40;
    bi.biSizeImage = 0;
    bi.biWidth = width;
    bi.biXPelsPerMeter = 10000;
    bi.biYPelsPerMeter = 10000;
    bf.bfOffBits = 54;
    bf.bfReserved1 = 0;
    bf.bfReserved2 = 0;
    bf.bfSize = width * height * 3 + 54;
    bf.bfType = 0x4d42;
    imgdata = (unsigned char *)malloc(sizeof(unsigned
char)*bf.bfSize);
    if (imgdata == NULL)
    {
        free(imgdata);
        printf("Exception: No enough space!\n");
        return false;
    }
    //像素格式设置4字节对齐
    glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
    //接收出像素的数据
    glReadPixels(0, 0, width, height, GL_BGR_EXT, GL_UNSIGNED_BYTE,
imgdata);
    time_t timep;
    struct tm *p;
    time(&timep);
    p = gmtime(&timep);
    int second = p->tm_sec;
    int year = p->tm_year +1900;
    int mon = p->tm_mon+1;
    int day = p->tm_mday;
    int hour = p->tm_hour + 8;
    int minute = p->tm_min;
    char cyear[30] = "";
    char cmonth[30] = "";
```

```
    char cday[30] = "";
    char chour[30] = "";
    char cminute[30] = "";
    char csecond[30] = "";
    char filename[50] = "Snapscreen/";
    itoa(year, cyear, 10);
    itoa(mon, cmonth, 10);
    itoa(day, cday, 10);
    itoa(hour, chour, 10);
    itoa(minute, cminute, 10);
    strcat(filename, cyear);
    strcat(filename, cmonth);
    strcat(filename, cday);
    strcat(filename, chour);
    strcat(filename, cminute);
    strcat(filename, "_SnapScreen.bmp");
    puts(filename);
    fp = fopen(filename, "wb+");
    if (fp == NULL)
    {
        printf("Exception: Fail to open file!\n");
        return false;
    }
    fwrite(&bf, sizeof(BITMAPFILEHEADER), 1, fp);
    fwrite(&bi, sizeof(BITMAPINFOHEADER), 1, fp);
    fwrite(imgdata, bf.bfSize, 1, fp);
    free(imgdata);
    fclose(fp);
    return true;
}
```

2.2 额外加分项要求

2.2.1 漫游时可实时碰撞检测

包围盒

碰撞检测中，包围盒已在程序外手动绘制完毕，程序运行时读取包围盒。包围盒为 AABB 三维包围盒。包围盒相关函数及方法已经在下述 Camera 类进行封装，详细代码示例见下。

```
bool Camera::CollisionDetect()
{
    int i;
```

```
    float x = m_Position.x, y = m_Position.y, z = m_Position.z;
    float x_max = x + m_Box.x, y_max = y + m_Box.y, z_max = z +
m_Box.z;
    float x_min = x - m_Box.x, y_min = y - m_Box.y, z_min = z -
m_Box.z;
    //printf("[%lf,%lf,%lf]", x, y, z);
    for (i = 0; i < Boxes_Count; i++)
    {
        if (!(x_min >= Boxes_Coords[i][0] || y_min >=
Boxes_Coords[i][1] || z_min >= Boxes_Coords[i][2] ||
            x_max <= Boxes_Coords[i][3] || y_max <= Boxes_Coords[i][4]
|| z_max <= Boxes_Coords[i][5]))
        {
            //printf("[(%lf, %lf, %lf), (%lf, %lf, %lf)] - ", x_max,
y_max, z_max, x_min, y_min, z_min);
            //printf("[(%lf, %lf, %lf), (%lf, %lf, %lf)]\n",
Boxes_Coords[i][0], Boxes_Coords[i][1],
            //    Boxes_Coords[i][2], Boxes_Coords[i][3],
Boxes_Coords[i][4], Boxes_Coords[i][5]);
            return true;
        }

    }
    return false;
}
```

游戏实例体现

游戏中人物并不能进行穿墙、穿过障碍物等行动。由于人物设定身高较高，在进入房间时，玩家开门后人物必须进行下蹲操作后方可进入房间，体现了包围盒的三维属性。

2.2.2 构建了基于此引擎的完整三维游戏，具有可玩性

游戏玩法

从前有一群快乐的皮卡丘生活在专教里，但是，来了一群外校的同学，他们要抢夺我们的皮卡丘去做实验。

作为正义的树莓学子，我们不能让他们得逞。以爱与正义的名义，拿起手中的小手枪击退敌人吧！

游戏中会不断自动生成会攻击玩家的 NPC。程序最初 NPC 在地图中随机走动，但当玩家进入 NPC 的视野中时，NPC 将主动追逐玩家。玩家被 NPC 攻击后血量会减少。当血槽清空后，玩家死亡，游戏结束。

玩家需要用手中的枪支攻击敌人，相应的，敌人被玩家攻击后血槽血量也会减少，当敌人血槽清空后死亡，但隔一段时间后即将在原地重生

游戏操作

- **键盘回调函数**

WSAD	前后左右	Shift	加速跑	Space	跳跃（支持二段跳）
C	下蹲/起立	ESC	显示鼠标	L	开关灯
O	开关手电筒	E	开关门	X	暂停游戏
Q	退出游戏				

为了使前后左右的移动流畅，不使用按下键盘才调用的键盘回调函数，而是使用 redraw()函数，也就是重新绘制的回调函数中调用，这样前后左右的移动操作会更加流畅。

代码实现：

```
void Camera::CameraKeyboard(int key, int x, int y){
    if (key == 27) { //ESC
        this->SetMouseMove(false);
        count--;
        while (count != 0) {
            ShowCursor(true);
            count++;
        }
        return;
    }

    switch (key){
        case 'q':{ exit(0); break; }
        case 'c':{
            if (upordown == false)
                downCamera();
            else
                upCamera();
            break;
        }
        case 'o':{ spot_on = !spot_on; break; }
        case ' ':
        {
            if (!Jump)
                Jump = JUMPDURATION * 2;
            else
                if (!SecondJump)
```

```
                SecondJump = JUMPDURATION * 2;
            break;
        }
        case '1':{ light_on = !light_on; break; }
        case '1':{ lightcolor = !lightcolor; break; }
        default:
            break;
    }
}

void Camera::MotionKeyboard(){
    //87: w  65:a  83:s  68:d
    if (KEY_DOWN(87))
        moveCamera(getSpeed()); //前进
    if (KEY_DOWN(83))
        moveCamera(-getSpeed()); //后退
    if (KEY_DOWN(65))
        yawCamera(getSpeed()); //左移
    if (KEY_DOWN(68))
        yawCamera(-getSpeed()); //左移
    if (KEY_DOWN(16))
        DashSpeed = 3.0;
    else
        DashSpeed = 1.0;
}
void KeyBoardCallBack(unsigned char k, int x, int y)
{
    if (k == 'm')
    {
        SnapScreen(swidth, sheight);
    }
    else if (k == 'b') {
        broken = true;
        PlayMusic((char*)"music/glass_broken.wav");
    }
    else if (k == 'x') {
        Pause = !Pause;
    }
    else if (k == 'f') {
        MyHp += 500;
    }
    else if (k == '2') {
        bs++;
    }
```

```
    else if (k == '3') {
        if (bs > 3){
            bs--;
        }
    }
    else if (k == '4') {
        geo.outputObj("Test.obj");
    }
    else if (k == '5') {
        lighty+=0.02;
        cout << "lighty"<<lighty << endl;
    }
    else if (k == '6') {
        lighty-=0.02;
    }
    m_Camera.CameraKeyboard(k, x, y);
    glutPostRedisplay();
}
void SpecialFunc(int k, int x, int y){
    m_Camera.CameraKeyboard(k, x, y);
    glutPostRedisplay();
}
```

- **鼠标回调函数**

右键点击开镜、左键点击射击、鼠标移动表示视角朝向变化，游戏界面隐藏光标。

左键射击使用 OpenGL 的拾取机制，首先对拾取对象命名，`glInitNames()`函数初始化姓名栈，在 `RandMoveDraw()` 函数中调用 `glPushName(PickNum)`函数向名字栈中压入名字，调用 `glPopName()`函数从栈顶弹出名字。

`glRenderMode(GL_SELECT)`函数进入选择模式，`gluPickMatrix((double)x, (double)viewport[3] - y, PICK_TOL, PICK_TOL, viewport)`函数设置拾取框，`glRenderMode(GL_RENDER)`函数返回拾取个数。

代码实现：

```
void mouseClick(int button,int state,int x, int y) {
    if (button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN) {
        m_Camera.SetZoom();
    }
    else if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN){
        m_Camera.Shot();
        int i;
        for (i = 0; i < 5; i++)
            npc[i].MousePick(button, state, x, y, swidth, sheight);
```

```
        ShowCursor(false);
        m_Camera.count--;
        m_Camera.SetMouseMove(true);
    }
}
void NPC::MousePick(int button, int state, int x, int y, GLfloat
width, GLfloat height) {
    if (HP && Pick)
    {
        glSelectBuffer(256, SelectBuffer);
        glGetIntegerv(GL_VIEWPORT, viewport);

        if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN) {
            printf("Left Button Click! %d %d\n", x, y);

            glRenderMode(GL_SELECT);
            glInitNames(); //初始化
            //glPushName(0);
            glMatrixMode(GL_PROJECTION);
            glPushMatrix();
            glLoadIdentity();
            gluPickMatrix((double)x, (double)viewport[3] - y,
PICK_TOL, PICK_TOL, viewport); //创建用于选择的投影矩阵栈
            GLfloat Ratio = (GLfloat)width / height;
            gluPerspective(45.0f, Ratio, 1.0f, 200.0f);  //设置投影方式

            glMatrixMode(GL_MODELVIEW);
            RenderMode = GL_SELECT;
            RandMoveDraw();
            glMatrixMode(GL_PROJECTION);
            glPopMatrix();

            glFlush();
            hits = glRenderMode(GL_RENDER);

            glViewport(0, 0, width, height);
            glMatrixMode(GL_PROJECTION);
            glLoadIdentity();
            gluPerspective(45.0f, Ratio, 1.0f, 200.0f);  //设置投影方式
            glMatrixMode(GL_MODELVIEW);
            printf("Hits: %d\n", hits);
            int modelselect = 0;
            if (hits > 0) {
                int n = 0;  double minz = SelectBuffer[1]; double maxz
```

```
= SelectBuffer[2];
                for (int i = 1; i < hits; i++) {
                    if (SelectBuffer[1 + i * 4] < minz) {
                        n = i; minz = SelectBuffer[1 + i * 4];
                    }
                }
                modelselect = SelectBuffer[3];
                printf("minz: %f maxz: %f Picking Object Name: %d\n",
minz, maxz, modelselect);
                if (modelselect == PickNum)
                {
                    //Hit = 200;
                    if (HP) {
                        PlayMusic((char*)"music/men_hurt.wav");
                        score++;
                        HP--;
                    }
                    if (!HP)
                        Respawn = RES_TIME;
                    obj[0].texMix(HP / 10.0);
                }
            }
            RenderMode = GL_RENDER;
            RandMoveDraw();
            glutPostRedisplay();
        }
    }
}
```

- **血量机制和 UI 提示**

玩家有一定初始血量，当被敌人靠近时会引发持续掉血，玩家自身在缓慢回血。敌人头顶有一个血条，击中敌人时，会引发敌人掉血，当敌人血条清空时敌人消失，一段时间后原地满血重生。

敌人血量通过实时绘制正方体并依据玩家与敌人角度实时调整绘制位置，使得玩家能够一直看到正面朝向自己的正方体血条，更好的进行判断。

由于代码比较琐碎，不在此展示。主要是在触发一些操作比如（射击击中敌人，敌人靠近玩家）时，表示血量的变量值作出变化，并实时显示在屏幕上。

屏幕右上角有玩家生命值的显示，以及分数提示，没击中一次敌人得一分。

屏幕左下角实时显示 FPS 值，当靠近门时提示可开门，屏幕中心显示绘制准星。

- **手持物体的稳定器算法**

在游戏中，玩家手持装备需要满足随视角的移动而左右前后移动，同时随视角的旋转（左右，俯仰）而同时旋转。同时该物品要出现在玩家视野的某个固定的方向，比如右前方等等。因此在这部分需要有一个较为复杂的变换。

```
void DrawStablizer() {
    ////////////////////////////////////////////////////////
    ///////////////摄像机稳定器V1.0.0/////////////////////
    ////////////////////////////////////////////////////////
    Vector3 temp = m_Camera.getPosition();
    float x = temp.x;
    float y = temp.y;
    float z = temp.z;
    Vector3 temp2 = m_Camera.getView();
    float viewx = temp2.x;
    float viewy = temp2.y;
    float viewz = temp2.z;

    float dx = 3;
    float dy = -3;
    float dd = 5;

    double Distance = pow(pow(viewx - x, 2) + pow(viewy - y, 2) +
pow(viewz - z, 2), 0.5);

    double ddx = dd * ((double)viewx - x) / Distance;
    double ddy = dd * ((double)viewy - y) / Distance;
    double ddz = dd * ((double)viewz - z) / Distance;

    double xzDistance = pow(pow(ddx, 2) + pow(ddz, 2), 0.5);

    double rx = -dx * ddz / xzDistance;
    double rz = dx * ddx / xzDistance;

    double PI = 3.1415926;

    //not rotate with view change
    glPushMatrix();
    glTranslatef(x + ddx + rx, y + ddy + dy, z + ddz + rz);
    coords.draw(); // 方向与坐标轴保持一致，随摄像机移动
    glTranslatef(0, 1, 0);
    int Flag1 = (ddz + rz) < 0 ? 0 : 1;
    int Flag2 = (ddz + rz) < 0 ? -1 : 1;
    glRotatef(atan((ddx + rx) / (ddz + rz)) * 180.0 / PI + Flag1 *
180.0, 0, 1, 0);
```

```
    glRotatef(atan(ddy / xzDistance) * 180.0 / PI, 1, 0, 0);
    coords.draw(); // 随摄像机移动和旋转
    glRotatef(-70, 0, 1, 0);
    handgun.draw();
    glPopMatrix();
}
```

2.2.3 具有一定的对象表达能力，能表达门、窗、墙等

- 门的开关

门的开关采用 Door 类，每一个对象代表一扇可开关的门，具体代码阐述见后对于类的描述。门的开关具体采用了碰撞检测与物体旋转的原理，利用一个较大范围的不可见碰撞盒判断人物已经进入了可开关该扇门的范围内，显示开关门提示，类似 PUBG，然后采用类中的两个状态判断变量，来对门完成旋转开门或关门。

```
void DOOR::FlashCallBack() {
    if (IsDoorOpen == false && IsDoorOpening == false &&
IsDoorClosing == false) {
        draw(0);
    }
    else if (IsDoorOpen == false && IsDoorOpening == true) {
        AddAngle();
        draw(angle);
    }
    else if (IsDoorOpen == true && IsDoorClosing == true) {
        SubAngle();
        draw(angle);
    }
    else if (IsDoorOpen == true && IsDoorOpening == false &&
IsDoorClosing == false) {
        draw(-90);
    }
}
void DOOR::PressECallBack() {
    if (IsDoorOpen == false) {
        ShowText((char*)"Press E to Open the Door", 160, 240);
        if (KEY_DOWN(69)) {
            PlayMusic((char*)"music/open_door.wav");
            IsDoorOpening = true;        //调整门开关状态
        }
    }
    else {
```

```
        ShowText((char*)"Press E to Close the Door", 160, 240);
        if (KEY_DOWN(69)) {
            PlayMusic((char*)"music/open_door.wav");
            IsDoorClosing = true;
        }
    }

}
void DOOR::AddAngle() {
    if (angle >= -90) {
        angle--;
    }
    else {
        IsDoorOpen = true;      //结束开门
        IsDoorOpening = false;
    }
}
void DOOR::SubAngle() {
    if (angle <= 0)
        angle++;
    else {
        IsDoorOpen = false;
        IsDoorClosing = false;    //结束关门
    }
}
void DOOR::SetDoorInfo(const string& RealOBJName, const string&
RealBMPName, const string& BoxesfileName, double x, double z) {
    RealDoor.setContent(RealOBJName, RealBMPName);
    Boxes.ReadBoxes(BoxesfileName);
    SetBoxesData(Boxes.getBoxesCount(), Boxes.getBoxesCoords());
    IsDoorOpen = false;
    IsDoorClosing = false;
    IsDoorOpening = false;
    posx = x;
    posz = z;
    angle = 0;

}
void DOOR::draw(int angle) {
    glPushMatrix();
    glTranslatef(posx, 0.0f, posz);
    glRotatef(angle, 0.0f, 1.0f, 0.0f);
    glTranslatef(-posx, 0.0f, -posz);
    RealDoor.draw();
```

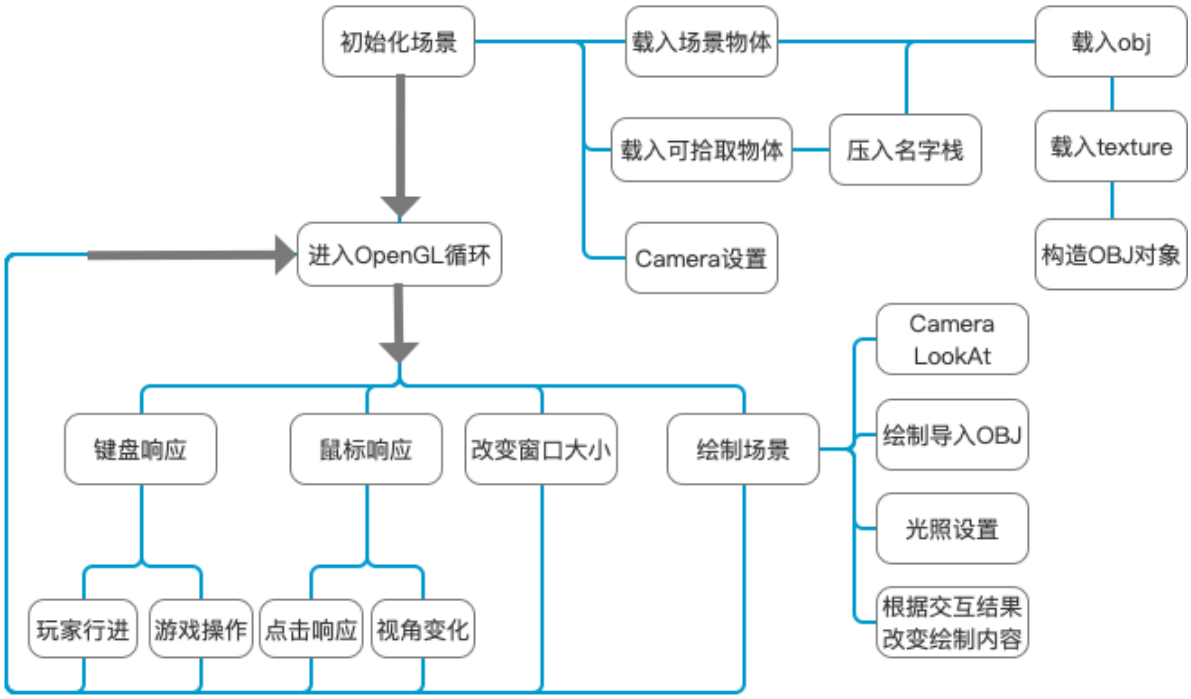
```
    glPopMatrix();
}

bool DOOR::CollisionDetect(Vector3 m_Position, Vector3 m_Box) {
    int i;
    float x = m_Position.x, y = m_Position.y, z = m_Position.z;
    float x_max = x + m_Box.x, y_max = y + m_Box.y, z_max = z +
m_Box.z;
    float x_min = x - m_Box.x, y_min = y - m_Box.y, z_min = z -
m_Box.z;
    //printf("[%lf,%lf,%lf]", x, y, z);
    for (i = 0; i < Boxes_Count; i++)
    {
        if (!(x_min >= Boxes_Coords[i][0] || y_min >=
Boxes_Coords[i][1] || z_min >= Boxes_Coords[i][2] ||
            x_max <= Boxes_Coords[i][3] || y_max <= Boxes_Coords[i][4]
|| z_max <= Boxes_Coords[i][5]))
        {
            return true;
        }

    }
    return false;
}
void DOOR::SetBoxesData(int count, GLfloat** data)
{
    Boxes_Count = count;
    Boxes_Coords = new GLfloat *[Boxes_Count];
    int i, j;
    for (i = 0; i < Boxes_Count; i++)
    {
        Boxes_Coords[i] = new GLfloat[6];
    }
    for (i = 0; i < Boxes_Count; i++)
    {
        for (j = 0; j < 6; j++)
        {
            Boxes_Coords[i][j] = data[i][j];
        }
    }
}
```

- 跃过桌子

我们为桌子也设置了碰撞检测盒，只有通过跳跃等操作才能翻越桌子，不然无法穿过。包括我们表达了墙等一系列对象，墙不可通行，桌子可以被翻越等等。

3 程序结构

3.1 程序运行流程



程序开始时，会有较长加载纹理及对象文件的时间，需要进行耐心等待。当程序自身完成初始化后，程序会不停的调用主函数绘制，其中包含键盘与鼠标的响应回调函数，来响应玩家的操作。

此处值得一提的是，由于采用普通键盘回调函数无法实现多按键同时响应的效果，所以我们采用了宏定义，判断某一按键状态的方法进行响应。

具体函数运行流程见上图。

3.2 程序类封装

3.2.1 Vector 类

向量类，便于管理各类向量的数据结构，重载运算符和单位化，叉乘，点积的运算实现。

提供接口：

```
    inline float length();                    /* 计算向量长度 */
    Vector3 normalize();                     /* 单位化向量 */
    float dotProduct(const Vector3& v);        /* 计算点积 */
    Vector3 crossProduct(const Vector3& v);     /* 计算叉积 */

    /* 重载操作符 */
    Vector3 operator + (const Vector3& v);
    Vector3 operator - (const Vector3& v);
    Vector3 operator * (float scale);
    Vector3 operator / (float scale);
    Vector3 operator - ();
```

3.2.2 Camera 类

Camera 类定义摄像机对象，用于在场景中进行漫游等操作，在摄像机类中实现 3 维的碰撞检测和聚光灯效果。
（1）负责修改摄像机的位置，观察点和速度等信息，实现漫游等操作。
（2）通过读入三维包围盒信息来进行碰撞检测。

外部接口：

```
/* 设置摄像机的位置, 观察点和向上向量 */
    void setCamera(float positionX, float positionY, float positionZ,
        float viewX, float viewY, float viewZ,
        float upVectorX, float upVectorY, float upVectorZ);
/* 鼠标旋转摄像机观察方向 */
    void setViewByMouse();
/* 跳跃实现 */
    void makeJump();
/* 放置摄像机 */
    void setLook();
/* 设置手电筒 */
    void SpotSet();
/* 回调函数 */
    void CameraKeyboard(int key, int x, int y);
```

```
    void MotionKeyboard();
```

内部接口（部分）：

```
/* 旋转摄像机方向 */
    void rotateView(float angle, float X, float Y, float Z);
/* 左右摄像机移动 */
    void yawCamera(float speed);
/* 前后移动摄像机 */
    void moveCamera(float speed);
/* 蹲下起立实现 */
    void downCamera();      // 蹲下
    void upCamera();        // 起立
/* 碰撞判断 */
    bool CollisionDetect();
```

3.2.3 OBJ 类

Object 类中存放法线、UV、面顶点等信息，负责实际场景里物体的描述和对该物体的几何变换、设置材质和设置纹理。
（1）负责 OBJ 物体的导入和导出。
（2）负责物体的材质设置，包括漫反射、镜面反射、和纹理的结合形式。
（3）负责物体的几何变换，同时读取导入的 OBJ 文件中的位置信息，对 OBJ 在场景中的位置进行定位。
（4）负责物体的纹理设置，贴图的绑定。

外部接口（部分）：

```
    /* 在opengl中绘制obj图形 */
    void draw();
    /* 导入 obj 模型和贴图 */
    void setContent(const string& obj, const string& tex, int data =
0);
    /* 导入包围盒*/
    void ReadBoxes(const string& filename);
    /* 贴图混合*/
    void texMix(float alpha);
```

内部接口（部分）：

```
    /* 统计.obj文件中的点、法线、面数量 */
    void getLineNum();
    /* 读取.obj文件中的信息 */
    int readFile();
```

```
    /* 读取BMP文件并且返回Bitmap，用于opengl绑定贴图 */
    void LoadBitmapFile();
    void LoadBitmapFile2();
    /* 绑定贴图 */
    void texload();
```

3.2.4 NPC 类

NPC 类负责场景中 NPC 的位置、大小、运动速度等基本属性进行设置，同时实现多帧连续绘制动画、人物运动和血量变化等功能，同时实现拾取模式的射击功能。

（1）负责 NPC 对象的基本属性，包括所处位置、大小、运动速度。

（2）结合 NPC 的位置地点坐标与玩家坐标计算，判断玩家是否被 NPC 发现，是否需要对玩家进行攻击。

（3）通过读入多个 OBJ 物体并进行实时播放，达到多帧连续绘制动画的效果。

外部接口（部分）：

```
    bool Found(float angle, float sight, float x, float y, float z);
    void Chase(float x1, float x2, float z1, float z2, float x, float
y, float z, float speed);
    void DisplayHP( float x, float y, float z);
    void resurrect();
    void DrawPika(float x1, float x2, float y, float z1, float z2,
float speed);
    void RandMove(float x1, float x2, float y, float z1, float z2,
float speed);
    void MousePick(int button, int state, int x, int y, GLfloat
width, GLfloat height);
```

内部接口（部分）：

```
    void RandMoveDraw();
    void CircularMove(float r, float speed);
    void setObjNum(int n);
    void setSpeed(float speed) { Speed = speed; }
    void setFPS(float fps) { FPS = fps; }
    Float getSpeed();
```

3.2.5 Door 类

Door 类作用于游戏中的门，并进行开门、关门等功能的实现。类内保存门旋转轴心、门的开闭状态等参数。
（1）负责实现门的开闭旋转动画。
（2）负责进行门的碰撞检测，从而判断当前状态点玩家靠近时是否需要提示“按 E 开门”。
外部接口（部分）：

```
    void SetDoorInfo(const string& RealOBJName, const string&
RealBMPName, const string& BoxesfileName, double x, double z);
//设置门的参数信息
    void SetBoxesData(int count, GLfloat** data);  //设置Box数据
    void draw(int angle);
    bool CollisionDetect(Vector3 m_Position, Vector3 m_Box);
//碰撞检测
    void PressECallBack();
//通过碰撞检测，来判断是否显示 E
```

内部接口（部分）：

```
    void FlashCallBack();    //通过获取当前门的状态
    void AddAngle();
    void SubAngle();
```

3.2.6 UI 设计

提供关于准星绘制、屏幕截取、音乐播放、文字显示、FPS 计算的函数

```
    float getFPS(int posx,int posy);
    void ShowText(char* text,int posx,int posy);
    bool SnapScreen(int width, int height);
    void DrawSightBead(GLuint swidth,GLuint sheight,int posx,int
posy,float r,int circlewidth,float srecwidth,float brecwidth,
    float ccircle,float scircle, float bcircle,float brecheight);
    void PlayMusic(char* filename);
```

3.2.7 GEO 类

基本几何体素（球，多面棱锥，多面棱柱）绘制及 OBJ 导出

```
    void drawPrism(GLfloat h, GLfloat r, int n);
    void drawBall(GLUquadricObj *uquadric, GLfloat r, GLfloat cradix,
GLfloat hradix);
    void drawPyramid(GLfloat h, GLfloat r1, GLfloat r2, int n);
    void outputObj(const string& filename);
```

4 小组分工

沈吕可晟(组长)：UI 类/Door 类/GEO 类部分 + 代码整合展示
陆子仪：OBJ 类/NPC 类/GEO 类部分/碰撞检测实现
胡晶晶：Camera 类/Vector 类/碰撞检测实现
徐闻语：主程序代码整合 + 贴图绘制 + OBJ 文件模型设计

5 附录及提交文件说明

1、Doc：

文件夹中包含此文档

2、Pro：

文件夹中为 CGProject 的工程文件，其中有 CGProject.sln 工程文件，以及 Readme.txt 说明文件，在 CGProject 文件夹中，存放有各个文件源代码，及贴图纹理、模型文件，具体说明见 Readme.txt

3、Exe：

文件夹中包含可执行文件及贴图纹理、模型文件，可以直接点开执行。

备注：贴图纹理、模型、音乐等素材文件在此提交整个文件中，为方便 exe 可以直接打开运行，存在两份。