# 计算机图形学大作业项目报告

> 小组成员与分工：
>
> 潘孟祺 3190104986 地图网格、选取，漫游与碰撞检测
>
> 刘逸飞 3190101847  流水线网格与物品网格，obj模型处理
>
> 孔泉 3190103368 obj文件处理模块，材质处理
>
> 王越嵩 3190105303 光照模型，阴影效果
>
> 帅子滔 3190105611 gui控制绘制，glfw使用
>
> 贡献比例均为1

## 1. 项目总体说明

### 1.1 项目概述

本项目实现了一个基于OpenGL的可编辑智能工业流水线3D场景，可通过键盘、鼠标进行控制。场景内部支持obj模型导入和材质渲染，实现了物体沿流水线的动态变化，实现了多种光照模型和阴影效果，支持自由漫游和碰撞检测。

目标为实现一个可编辑的仿真工厂流水线场景，可以在虚拟世界中仿真现实工厂运行状况，方便工厂建设时的设计，符合工业只能的目标。

### 1.2 文件结构

```
├─cpp
├─Debug
│  └─CG_final.tlog
├─img
├─include
│  ├─GL
│  ├─glad
│  ├─GLFW
│  ├─glm
│  │  ├─detail
│  │  ├─gtc
│  │  ├─gtx
│  │  └─simd
│  ├─GLU
│  ├─imgui
│  └─KHR
├─lib
└─x64
└─Debug
  └─CG_final.tlog
```

源代码保存在cpp目录中，截图结果保存再img目录中，include目录内存放包含的库内容

### 1.3 运行环境

开发和运行环境：Visual Studio 2019

使用的图形学库：glut, glew, nglm

### 1.4 创新点

流水线可以人工编辑，而非固定流水线，具有较高自由度。

在可编辑的流水线基础上能够进行漫游与碰撞检测，实时性较强。

通过obj模块导入复杂模型，实现更加精细的场景建模。

可以利用gui进行控制。

## 2. 模块介绍

### 2.1 选取与地图网格化

```
class Point
{
public:
    double x, y, z;
    Point();
    Point(double ix, double iy, double iz);
};
```

首先设计顶点类，简单地存储xyz坐标值。

```
class MapUnit
{
public:
    double GirdLen = MAPBOUND / MAPSIZE;
    Point leftbottom;
    Point rightup;
    int status;
    MapUnit();
    MapUnit(int s, Point lu, Point rb);
    void drawGrid();
    void drawModel();
};
```

之后设计MapUnit类，用于表示每一个地图块，存储地图网格内部的信息。地图为一个二维平面，我们将其置于xoy平面内部，用两个Point对象存储每个地图网格的左下和右上两个点的坐标。status变量存储地图网格所处的状态。两个draw函数负责绘制地图网格相关的模型。

```
class Map
{
public:
    double GirdLen = MAPBOUND / MAPSIZE;
    MapUnit map[MAPSIZE][MAPSIZE];
    Map();
    void drawMap();
    void Select(double x, double y, int status);
}
```

Map类表示对整个地图的网格化，GirdLen控制网格大小，MapUnit二维数组存储二维地图平面上的地图块信息，Select函数根据传入的x、y坐标值在地图网格上进行选取（地图网格位于xoy平面，只需xy坐标就可确定网格位置）。drawMap函数通过遍历二维数组map绘制整个地图的情况。

```
void SelectMap(int x, int y)//select a block from the map
{
    int viewport[4];
    double modelview[16];
    double projection[16];
    float winx, winy, winz;
    double objx, objy, objz;
    glGetDoublev(GL_MODELVIEW_MATRIX, modelview);
    glGetDoublev(GL_PROJECTION_MATRIX, projection);
    glGetIntegerv(GL_VIEWPORT, viewport);
    winx = (float)x;
    winy = (float)viewport[3] - (float)y - 1.0f;
    glReadBuffer(GL_BACK);
    glReadPixels(x, int(winy), 1, 1, GL_DEPTH_COMPONENT, GL_FLOAT, &winz);
    gluUnProject((GLdouble)winx, (GLdouble)winy, (GLdouble)winz, modelview,
projection, viewport, &objx, &objy, &objz);
    double ClickPosition[3];
    double ClickDirection[3];
    ClickPosition[0] = objx; ClickPosition[1] = objy; ClickPosition[2] = objz;

    ClickDirection[0] = ClickPosition[0] - CameraPosition[0];
    ClickDirection[1] = ClickPosition[1] - CameraPosition[1];
    ClickDirection[2] = ClickPosition[2] - CameraPosition[2];

    if (ClickDirection[2] == 0) return;
    normalize(ClickDirection);
    double len = (-1 * CameraPosition[2]) / ClickDirection[2];

    BackGroundScene.Select(CameraPosition[0] + len * ClickDirection[0],
CameraPosition[1] + len * ClickDirection[1], 1);
    PipeLine.Select(CameraPosition[0] + len * ClickDirection[0],
CameraPosition[1] + len * ClickDirection[1]);
}
```

地图选取的手段主要通过鼠标左键点击，需要根据鼠标点击在窗口中的位置和当前的观察方向确定鼠标点击处在三维CG世界中的位置并将其投影在地图网格上。

首先获得视窗矩阵、模型矩阵和透视矩阵的数据，通过矩阵运算将鼠标点击的屏幕坐标转换到透视观察的范围中，再根据透视矩阵的数据计算出在世界坐标中的位置。根据世界坐标中的位置和相机位置计算方向向量，最后沿该方向从相机到达地图平面，得到选取点在地图平面上的位置，根据其坐标做后续计算。

### 2.2 漫游与碰撞检测

漫游的重点在于视角控制，首先在鼠标点击函数MouseFunc中判断鼠标点击状态，当检测到右键按下后将视角旋转标志变量置1，在鼠标移动函数MotionFunc中根据鼠标在窗口中的移动改变旋转角度变量，根据旋转角度变量改变相机方向。

```
void MouseFunc(int button, int state, int x, int y)
{
    if (button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
    {
```

```
        CameraRotate = true;
    }
    if (button == GLUT_RIGHT_BUTTON && state == GLUT_UP)
    {
        CameraRotate = false;
    }
    if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN & !CameraSwitch)
    {
        SelectMap(x, y);
    }
}
void MotionFunc(int x, int y)
{
    y = WindowHeight - y;
    static int LastX = 0;
    static int LastY = 0;
    static float CameraRotateX = 0;
    static float CameraRotateY = 0;
    if (CameraRotate)
    {
        if (x > LastX)
        {
            CameraRotateX -= RotateChange;
            LastX = x;
        }
        else
        {
            CameraRotateX += RotateChange;
            LastX = x;
        }
        if (y > LastY)
        {
            CameraRotateY -= RotateChange;
            LastY = y;
        }
        else
        {
            CameraRotateY += RotateChange;
            LastY = y;
        }
    }
    CameraDirection[0] = sin(CameraRotateX);
    CameraDirection[1] = sin(CameraRotateY);
    CameraDirection[2] = -cos(CameraRotateX);
    normalize(CameraDirection);
}
```

移动时根据相机方向CameraDirection计算相机位置，实现漫游功能。

碰撞检测功能在漫游基础上实现，当计算下一个移动到的位置后将该坐标转化到流水线网格上，判断该网格内是否有物体。若有物体则阻塞移动，否则执行移动。如此便实现了碰撞检测。

```
static bool BlockCheck(double newx, double newy, double newz)
{
    int i = -1;
    int j = -1;
    double Len = PIPE_GRID_TIME * GRIDLEN;
```

```
    if (!CameraSwitch) return 0;
    if (newx > RoamLimit || -newz + InitalCameraPos > RoamLimit || newx < -
RoamLimit || -newz + InitalCameraPos < -RoamLimit || newy < BottomLimit || newy
> HeightLimit)
    {
        return 1;
    }
    else
    {
        i = (newx + 0.5 * MAPBOUND) / Len;
        j = (-newz + 0.5 * MAPBOUND) / Len;
        if (i >= 0 && j >= 0 && i < PIPESIZE && j < PIPESIZE && PipeLine.map[i]
[j].type != 0 && newy < PipelineComponetHeight) return 1;
        else return 0;
    }
}
```

### 2.3 流水线网格与物品网格

程序中设计了流水线网格类和物体类以及它们的map来让流水线和物体能够运动起来。

```
###### 单个流水线类
```

程序中设计了单个流水线类，可以通过它表示每个流水线的状态，方便进行绘制。

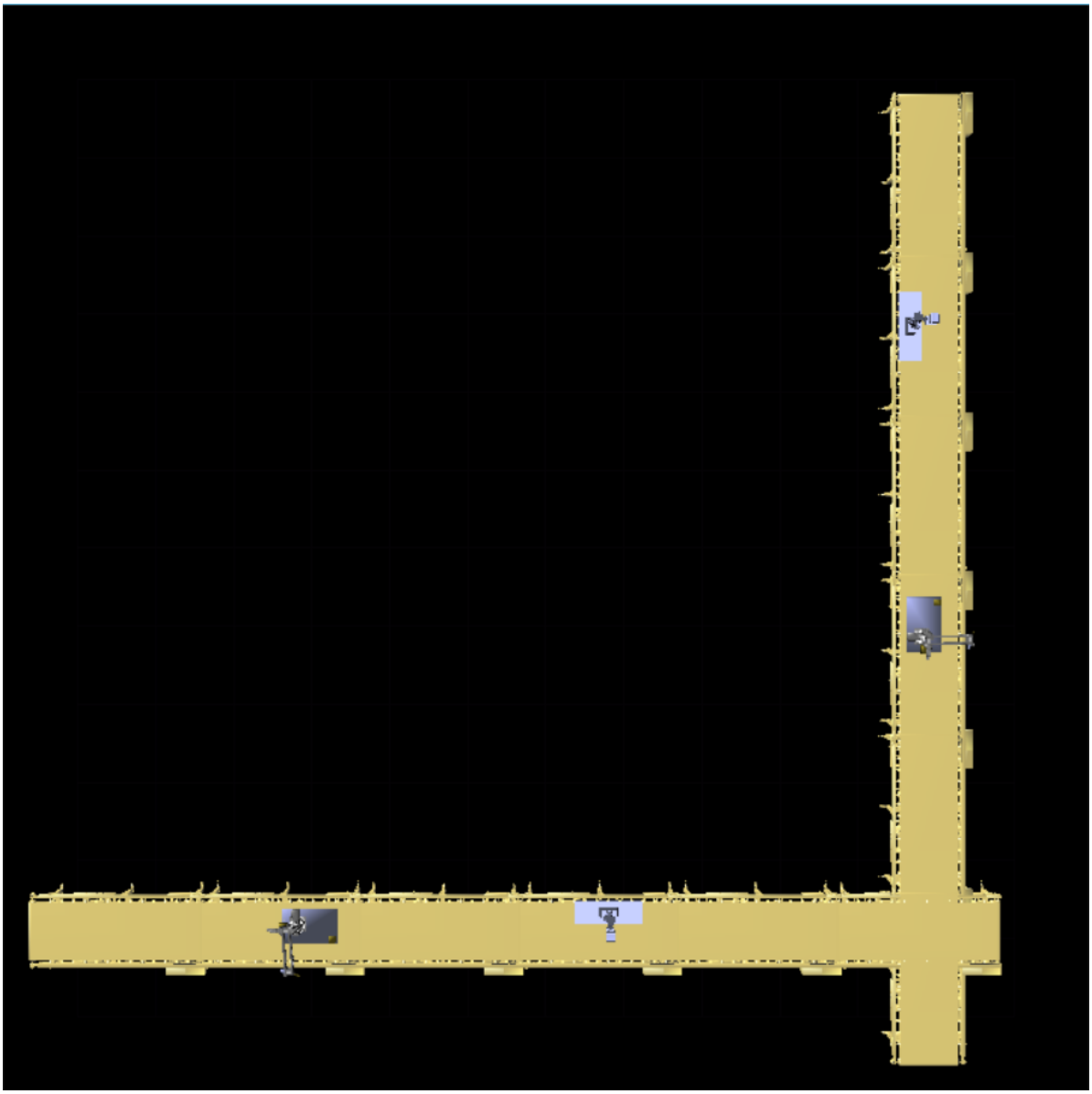
```
class PipelineUnit//流水线单元
{
public:
    Point leftbottom;
    Point rightup;
    int isMachine = 0;//是否是中间的处理机器
    double Len = PIPE_GRID_TIME * GRIDLEN;//每个流水线零件单元格的长度
    int type;
    //流水线的部件分类 在流水线数组中方便根据不同的数字绘制不同的流水线组件
    //0：没有流水线 1：流水线起点 朝向up 2：流水线起点 朝向left 3：流水线起点 朝向down 4：
流水线起点 朝向right
    //5：流水线终点 朝向up 6：流水线终点 朝向left 7：流水线终点 朝向down 8：流水线终点 朝向
right
    //9：普通流水线轨道up->down方向  10：普通流水线轨道left->right方向 11：普通流水线轨道
left->up方向
    //12：普通流水线轨道left->down方向 13：普通流水线轨道right->up方向 14：普通流水线
right->down方向
    //15：初始的起点
    //16 某些流水线上的处理装置。
    //17：普通流水线轨道down->up方向  18：普通流水线轨道right->left方向 19：普通流水线轨道
up->left方向
    //20：普通流水线轨道down->left方向 21：普通流水线轨道up->right方向 22：普通流水线
down->right方向
    //23及以上是其他流水线上的处理装置

    int modetype;//哪种流水线上的处理装制。
    PipelineUnit();
    PipelineUnit(int s, Point lu, Point rb);//用左上右下点初始化一个流水线单元
    void ChangeType(int s);
    void drawModel();//根据不同的type绘制不同的流水线组成部分
```

```
};
```

### 2.3.1 流水线网格类

```
class PipelineMap//流水线地图
{
public:
    double Len = PIPE_GRID_TIME * GRIDLEN;
    PipelineUnit map[PIPESIZE][PIPESIZE];//流水线单元的矩阵
    int haveBegin = 0;//是否有了起点
    std::vector<int> BeginPoint;//起点的坐标，方便之后物体向终点运动
    int isEnd = 0;//流水线是否画完了
    std::vector<int> nowEndPoint;
    //若未画完，则是画的流水线暂时末端的坐标；若已画完，则是流水线最终终点的坐标，方便之后物体向终点运动。
    int paintMode = 0;//绘制模式，若为0，则代表不再绘画，若为1，则代表可以从暂时的流水线末端向鼠标点击的点延申流水线
    //若为2，则可以将流水线上已经存在的普通流水线轨道改为某种加工器件。
    //若为3，则绘制结束，无法再改变流水线。
    int showmode = 0; //0则只绘制流水线，1会绘制流水线上运动的物体
    PipelineMap();
    void drawMap();//绘制所有的流水线。
    void Select(double x, double y);//根据输入的某个xy坐标改变流水线上的某个部分，如从无流水线到有流水线，从无机械臂到有机械臂
};
```

通过流水线网格类，可以实现的流水线自动绘制，即通过选取地图上的点，自动生成流水线。类似RTS游戏中建造围墙：只要确定好各个转向点，就能自动联通。还能够让流水线上的单元添加操作单元，例如给流水线添加机械臂（即修改流水线数组某单元的modetype值）等。



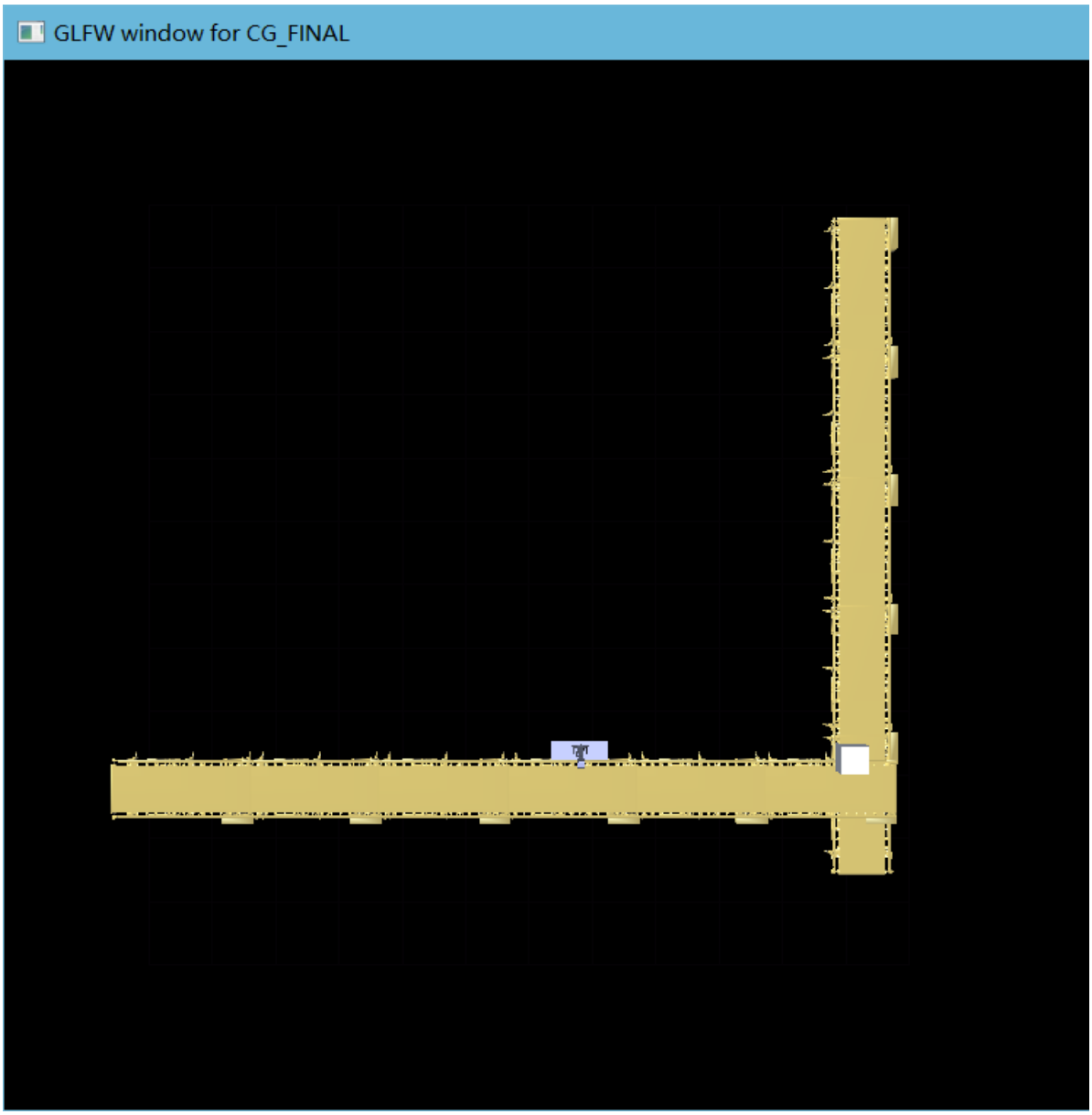
为了实现这样的绘制方法，我设计为当下一个转折点确定后，之前的流水线的重点才能决定好向哪个方向转弯（即流水线的type），直到最后按键停止绘制流水线为止。采用这种方式可以灵活的绘制流水线。

### 2.3.2 单个物品类

流水线上的单个物品，可组成物品地图后可以实现流水线上物品的运动。

```
class Product//被操作的工件
{
public:
    //int isLive = 0;//这个工件是否存在
    int isPipeLine = 0;//这个地方是否有流水线单元
    int isMachine = 0;//这个流水线单元是否是加工器
    double moveRatio = 0;//工件在某个流水线单元上运动完成的百分比，例如50%是走了一半
    double movespeed = 0.1;//每次移动移动的ratio量，相当于速度
    double pipelineZline = 0.3;//流水线的厚度，防止工件和流水线重叠。
    double large = 0.2;//画的货物的大小
    int hasPaint = 0;//是否被画过了
    int direction;//工件的下一个位置在当前位置的哪个方向？0：没有下一个位置
    //1：up 2：left 3：down 4：right
    Point moveBegin, moveEnd;//在这个流水线单元上的运动起始点和结束点
    Product();
    Product(PipelineUnit beginUnit);//用某个流水线节点初始化工件。
```

```
    //void MoveProduct();//工件不断改变位置
    void drawProduct();//画出工件
};
```




右下方的流水线上的工件的正方体就是一个物体。

### 2.3.3 物品网格类

之后我又写了productmap类，它通过储存所有product的信息来实现让物体在流水线上运动的效果。思路是让已经设计好（即以后固定不变了）的流水线网格类作为参数，让物品网格类进行初始化。流水线的上每个点的方向和和移动的起点和重点都进行了规定。这样物体在一个物品网格上由起点运动到重点，就转化为在单个物品类里面由起点运行到重点，再转移到下一个单个物品类里面。这样就能实现被操作的物体在流水线上运动的效果了。

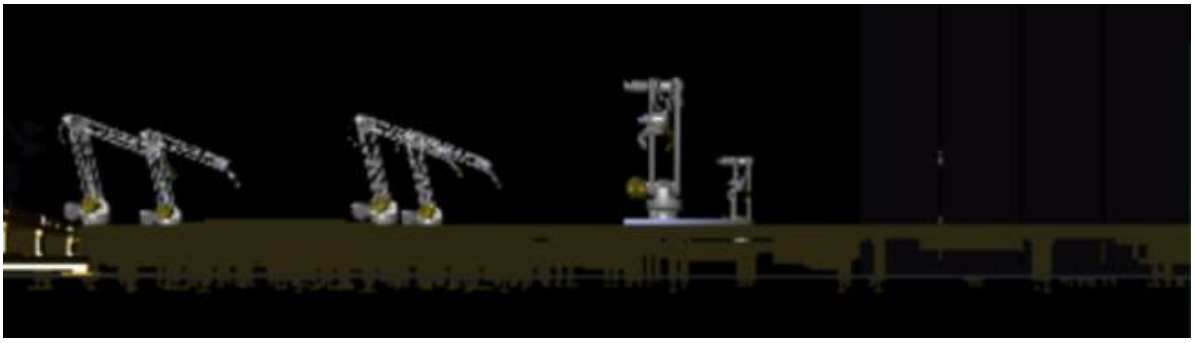
```
class ProductMap//物品地图
{
public:
    double Len = PIPE_GRID_TIME * GRIDLEN;
    Product map[PIPESIZE][PIPESIZE];

    std::vector<int> BeginPoint;//起点的坐标，方便之后物体向终点运动
```

```
    std::vector<int> nowEndPoint;//物品终点坐标
    std::vector<int> nowPaint;//正在绘制的物品的坐标
    int hasPaint = 0;
    int showmode = 0; //0则只绘制流水线，1会绘制流水线上运动的物体
    //PipeMap();
    ProductMap();
    ProductMap(PipelineMap pipemap);
    void drawMap();
};
```

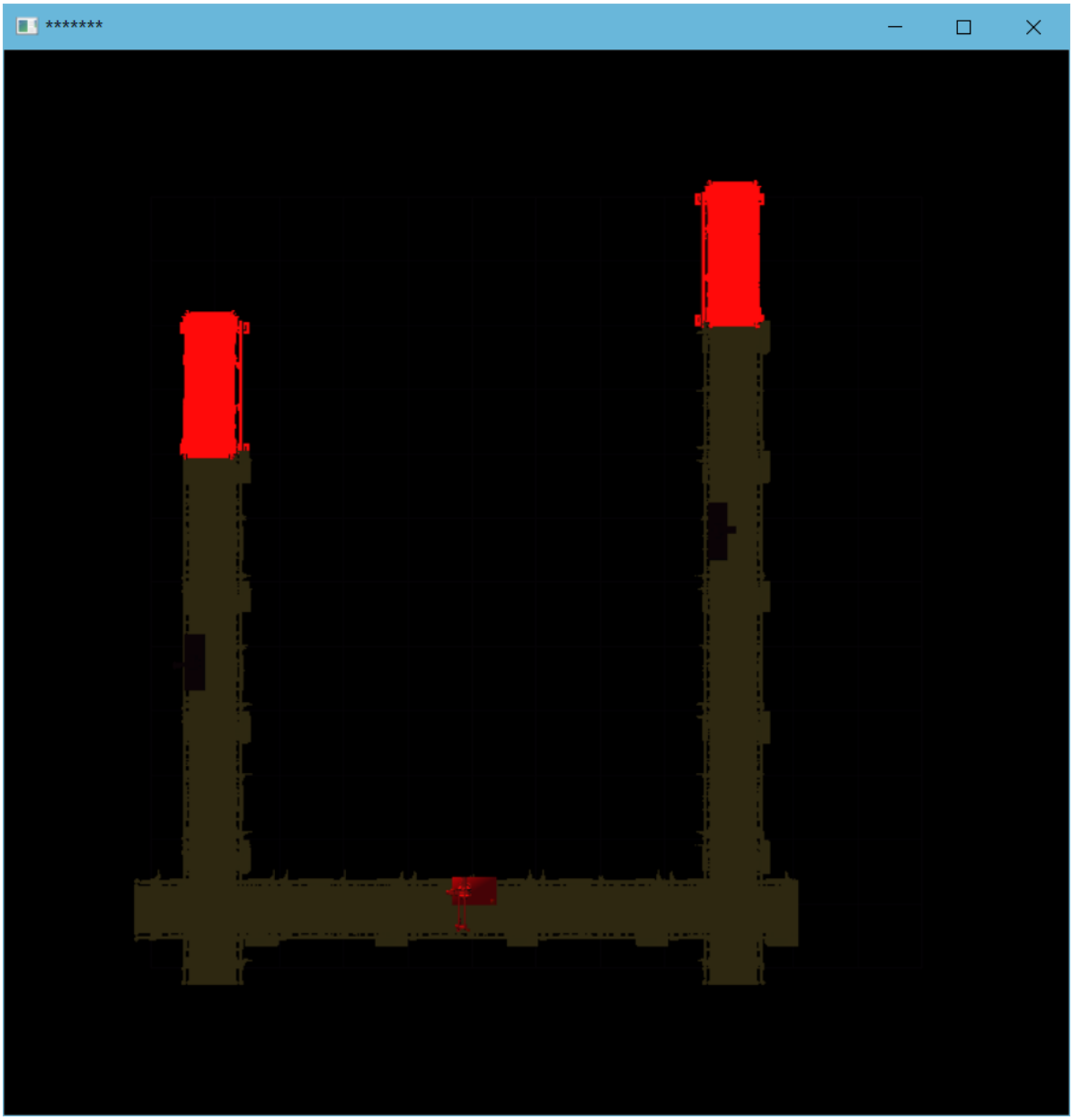
### 2.4 obj文件编辑

在obj文件导入的过程中，发现将导入后的obj文件在opengl中进行glScalef放大或缩小后，物体的材质等信息会发生改变，会导致物体变暗、反光度消失等一系列问题。

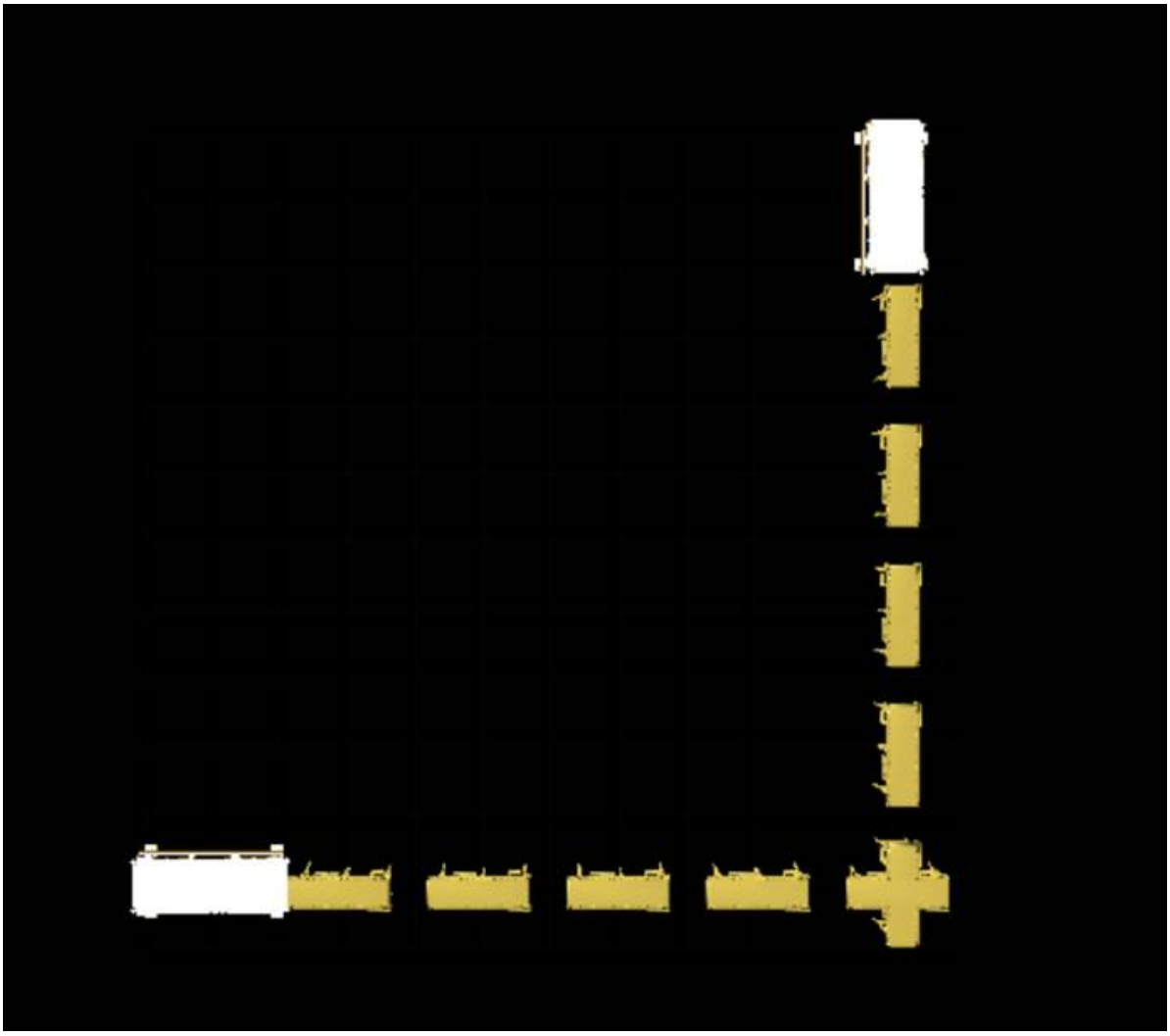
比如下图中的流水线颜色深到很难看清就是obj文件在opengl中进行glScalef放大导致的。



而且这样的obj文件对不同颜色的光没有明显的区别，例如当红光照射时，黄色的流水线和其他颜色的光照射时区别不大。



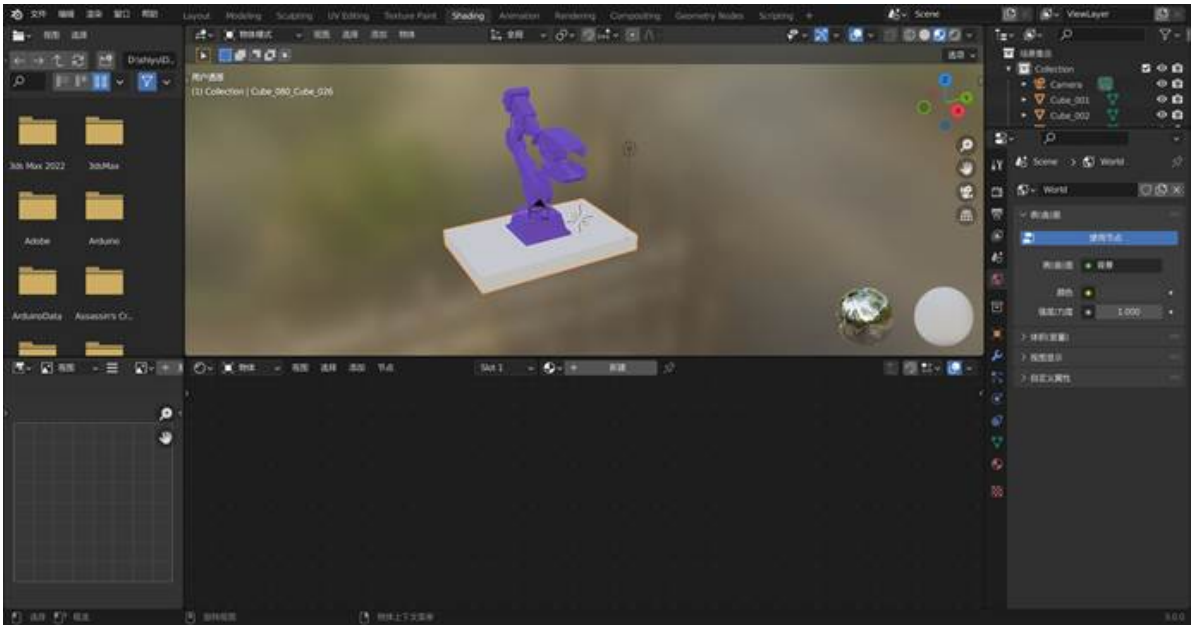

而它的正常效果是这样的，可以看到黄色的流水线非常有光泽，看起来颇有真实度。



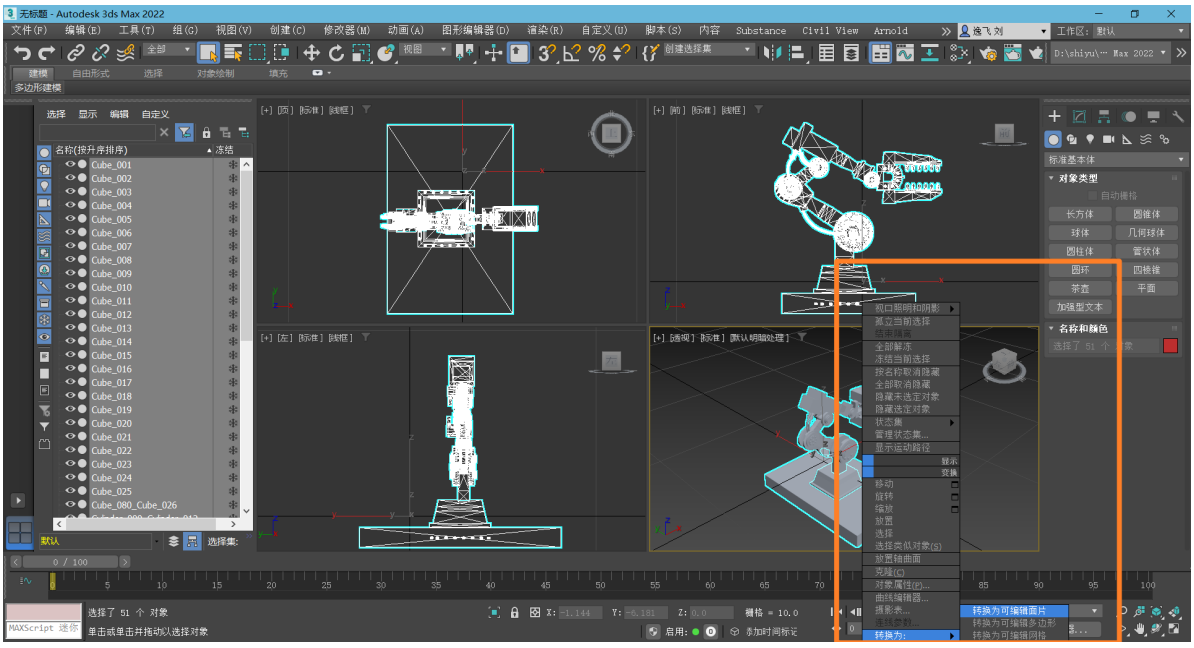
因此需要在程序中引入显示列表时，先用glScarlef测出各个方向上缩放的长度，再用blender软件对模型的长宽高进行调整，导出为obj文件。再将这个obj文件导入3dmax软件，再将模型用3dmax转化为三角形面片。

例如在下面的blender界面中，将机械臂的各个组件的长宽高修改为某一数字。

还能在操作相对简便的blender里面给物体添加材质。比起在mtl文件里直接编辑，和用看不懂界面的、较为复杂的3dmax添加贴图，简单很多。



之后需要用的3dmax的原因是负责导入obj的同学的软件只能识别3dmax导出的三角形面片，而不能识别其他的面片，如四边形面片。下图是转化为三角形面片的方法。



之后还需要用程序将每行行末的'\n'修改为'\r\n'，方便程序中obj文件读取。

### 2.5 读取与绘制obj文件（支持mtl）

Obj和mtl文件的出现是为了简化顶点绘制以及颜色纹理设置的复杂度。

#### 2.5.1 基本原理

obj文件保存了一系列顶点信息（位置坐标，法线向量，纹理坐标）以及由这些顶点构成的许多物体。不同的项目对应于以不同标头开头的行。

mtl文件中保存了一系列的材质，以newmtl分隔，以供obj文件中usemtl选用。本程序实现了Ka，Kd，Ks，Ns四个参数以及texture纹理贴图。一个典型的newmtl如图所示。

```
newmtl cornellbox_5_material
Ka 0.192250 0.192250 0.192250
Kd 0.507540 0.507540 0.507540
Ks 0.508273 0.508273 0.508273
Ns 51.200001
map_Kd textures/3368.bmp
```

我们要做的就是面向obj与mtl编程，读入文件内的顶点信息，然后调用顶点绘制的函数进行绘制，与此同时设置好加mtl给予的材质或者贴图即可。

### 2.5.2 数据结构

ObjFile类

```
class objFile
{
public:
    objFile(const char* objFileName);//构造objFile对象
    void objFileDraw();               //画出obj文件的所有物体
private:
    vector<Float3> mLocation;         //所有点的信息
    vector<Float3> mNormal;           //所有法线信息
    vector<Float2> mTexcoord;         //所有纹理坐标信息
    vector<objModel> objs;            //所有的物体
    map<string, Material> mtlMap;    //与.obj相关的.mtl文件的所有材质信息【后续可以添加纹理信息，只要拓展Material即可】
};
```

objModel类

```
class objModel
{
public:
    void push_face(Face* f)
    {
        mFace.push_back(*f);
    }
    void clear_faceVector() {
        int len = mFace.size();
        for (int i = 0; i < len; i++) {
            mFace.pop_back();
        }
    }
    void set_nameString(string* s)
    {
        Material_nameString = *s;
    }
    vector<Face>* get_mFace() {
        return &mFace;
    }
    string* get_nameString() {
        return &Material_nameString;
    }
private:
    vector<Face> mFace;               //三角面片的集合
    string Material_nameString;       //材质索引字符串
```

```
};
```

#### 2.5.3 具体实现

两个基本函数

```
objFile(const char* objFileName);
void objFileDraw();
```

·objFile作为构造函数，对obj和指定的mtl进行逐行解析并保存顶点、面片信息。

·objFileDraw作为公有方法，利用保存的信息进行了绘制。

设置材质

```
        /*打开纹理映射*/
        if (mtlMap[(*(objs[i].get_nameString()))].Kd_pic != "") {
            GLuint texture_enum = mtlMap[(*
(objs[i].get_nameString()))].Kd_pic_flag;
            glEnable(GL_TEXTURE_2D);
            glBindTexture(GL_TEXTURE_2D, texture[texture_enum]);
        }
        /*设置材质*/
        glMaterialfv(GL_FRONT, GL_AMBIENT, mtlMap[(*
(objs[i].get_nameString()))].Data[0]);
        glMaterialfv(GL_FRONT, GL_DIFFUSE, mtlMap[(*
(objs[i].get_nameString()))].Data[1]);
        glMaterialfv(GL_FRONT, GL_SPECULAR, mtlMap[(*
(objs[i].get_nameString()))].Data[2]);
        glMaterialfv(GL_FRONT, GL_SHININESS, mtlMap[(*
(objs[i].get_nameString()))].Data[3]);
```

·glBindtexture为物体绑定纹理。

·glMaterialfv为物体设置材质。

描点画图

```
        /*描点画图*/
        if (numOfFaces == 3) {
            glBegin(GL_TRIANGLES);
            for (auto faceIndex = objs[i].get_mFace()->begin(); faceIndex !=
objs[i].get_mFace()->end(); ++faceIndex)
            {
                //第一个点的法线，纹理，位置信息
                glNormal3fv(mNormal[faceIndex->vertex[0][2] - 1].Data);
                glTexCoord2fv(mTexcoord[faceIndex->vertex[0][1] - 1].Data);
                glVertex3fv(mLocation[faceIndex->vertex[0][0] - 1].Data);
                //第二个点的法线，纹理，位置信息
                glNormal3fv(mNormal[faceIndex->vertex[1][2] - 1].Data);
                glTexCoord2fv(mTexcoord[faceIndex->vertex[1][1] - 1].Data);
                glVertex3fv(mLocation[faceIndex->vertex[1][0] - 1].Data);
                //第三个点的法线，纹理，位置信息
                glNormal3fv(mNormal[faceIndex->vertex[2][2] - 1].Data);
                glTexCoord2fv(mTexcoord[faceIndex->vertex[2][1] - 1].Data);
                glVertex3fv(mLocation[faceIndex->vertex[2][0] - 1].Data);
            }
```

```
            glEnd();
        }
```

·glNormal3f指定顶点的法线信息。

·glTexCoord2fv指定顶点的纹理坐标。

·glVertex3f指定顶点的位置信息。

**2.6 光照模型**

首先在全局变量中加入有关光照的相关参数，如颜色、位置等。

```
GLfloat ambient[] = { 0.0, 0.0, 0.0, 1.0 };
GLfloat diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat shininess[] = { 0.0 };
GLfloat position[] = { -1.0f, 1.0, -1.0, 0.0 };
GLfloat position2[] = { 0.0, 0.0, 2.0, 0.0 };

int index = 0;
GLfloat color[][4] = {
    { 1.0f, 1.0f, 1.0f, 1.0f },
    { 1.0f, 0.0f, 0.0f, 1.0f },
    { 0.0f, 1.0f, 0.0f, 1.0f },
    { 0.0f, 0.0f, 1.0f, 1.0f },
};
```

在灯光函数中设置好灯光的属性。这里采用了`OpenGL`中的两个灯光`Light0`和`Light1`，分别设置环境光、散射光等灯光成分。

```
void lightinit()
{

    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);
    glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, color[index]);
    glLightfv(GL_LIGHT0, GL_SPECULAR, color[index]);
    glLightfv(GL_LIGHT0, GL_SHININESS, shininess);
    glLightfv(GL_LIGHT0, GL_POSITION, position);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    glLightfv(GL_LIGHT1, GL_AMBIENT, ambient);
    glLightfv(GL_LIGHT1, GL_DIFFUSE, color[index]);
    glLightfv(GL_LIGHT1, GL_SPECULAR, color[index]);
    glLightfv(GL_LIGHT1, GL_SHININESS, shininess);
    glLightfv(GL_LIGHT1, GL_POSITION, position2);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT1);
}
```

之后在绘制函数`display`中调用`lightinit`。一开始在`main`函数中调用，但是发现这样无法对灯光的位置、颜色等属性进行修改。必须在`display()`这样的回调函数中调用才能保证对灯光属性进行修改。

在`imgui`中，设置了可以更改灯光位置的窗口（`light_window`），可以通过拖动滑动条，对灯光的位置进行修改。同时，键盘输入`.`能够修改灯光的颜色。

在阴影方面，一开始的想法是使用阴影贴图算法绘制阴影，并且在测试工程里面实现了阴影功能（测试工程里绘制是用OpenGL自带的函数）。但是对`obj`文件使用这种方法会出现"海市蜃楼"的现象，效果不理想，因此未加入到正式工程中。正式工程中有光影的变换效果。

### 2.7 截屏保存

截屏主要通过读取当前画板上的像素数据并写入bmp文件中得到bmp格式的图片。主要通过函数`glReadPixel()`函数读取当前画板的像素信息，并通过 fwrite 函数写入bmp文件。需要注意bmp文件的文件信息头填写，函数编写时采用已有的bmp文件进行固定文件信息头内容的复制，并根据画板的具体信息填充如图片大小等其他内容。

具体代码及详细注释如下：

```
void grab(void)
{
    FILE* pDummyFile;  //指向另一bmp文件，用于复制它的文件头和信息头数据
    FILE* pWritingFile;  //指向要保存截图的bmp文件
    GLubyte* pPixelData;    //指向新的空的内存，用于保存截图bmp文件数据
    GLubyte  BMP_Header[BMP_Header_Length];
    GLint    i, j;
    GLint    PixelDataLength;   //BMP文件数据总长度

    // 计算像素数据的实际长度
    i = WindowWidth * 3;   // 得到每一行的像素数据长度
    while (i % 4 != 0)       // 补充数据，直到i是的倍数
        ++i;
    PixelDataLength = i * WindowHeight;  //补齐后的总位数

    // 分配内存和打开文件
    pPixelData = (GLubyte*)malloc(PixelDataLength);
    if (pPixelData == 0)
        exit(0);


    pDummyFile = fopen("img/bmp_example.bmp", "rb");//只读形式打开
    if (pDummyFile == 0)
        exit(0);

    string num = to_string(pic_num);
    string filename = "img/pic" + num + ".bmp";

    pWritingFile = fopen(filename.c_str(), "wb"); //只写形式打开
    if (pWritingFile == 0)
        exit(0);

    //把读入的bmp文件的文件头和信息头数据复制，并修改宽高数据
    fread(BMP_Header, sizeof(BMP_Header), 1, pDummyFile);  //读取文件头和信息头，占据
54字节
    fwrite(BMP_Header, sizeof(BMP_Header), 1, pWritingFile);
    fseek(pWritingFile, 0x0012, SEEK_SET); //移动到0x0012处，指向图像宽度所在内存
    i = WindowWidth;
    j = WindowHeight;
    fwrite(&i, sizeof(i), 1, pWritingFile);
    fwrite(&j, sizeof(j), 1, pWritingFile);
```

```
    // 读取当前画板上图像的像素数据
    glPixelStorei(GL_UNPACK_ALIGNMENT, 4);  //设置4位对齐方式
    glReadPixels(0, 0, WindowWidth, WindowHeight, GL_BGR, GL_UNSIGNED_BYTE,
pPixelData);

    // 写入像素数据
    fseek(pWritingFile, 0, SEEK_END);
    //把完整的BMP文件数据写入pWritingFile
    fwrite(pPixelData, PixelDataLength, 1, pWritingFile);

    // 释放内存和关闭文件
    fclose(pDummyFile);
    fclose(pWritingFile);
    free(pPixelData);
}
```

在`imgui`中，设计了一个截屏的按钮，按下按钮即可实现截屏。

需要注意的是，在项目中的`img`文件夹中，需要预先提供一个名为`bmp_example.bmp`的文件才能进行截图，否则会中途退出。截出的图像是`bmp`图像。

### 2.8 gui控制绘制

#### 2.8.1 ImGUI简介

ImGUI是与平台无关的C++轻量级跨平台图形界面库，适用于画面质量要求较高的场景，经常与OpenGL相结合，在directx和glfw的框架下使用。

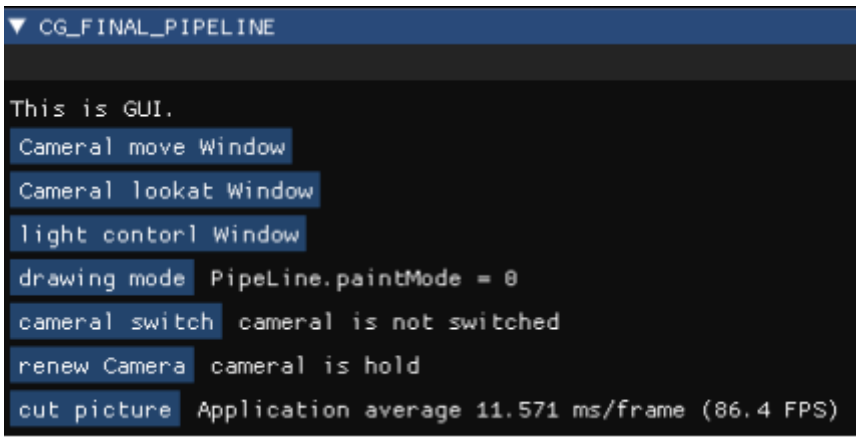
其中，在directx的框架下，ImGUI主要是通过创建win32的窗口，在main里面进行循环绘制，这个会造成阻塞，这也是我们最终选择更换glut为glfw的原因。而在glfw框架下，ImGUI利用glfw创建窗口时的句柄使得其可以在glfw的窗口内绘制GUI，ImGUI的实现是在main函数中glfw绘图的循环里面调用ImGUI的绘图函数来循环绘制GUI。

#### 2.8.2 本项目中ImGUI功能介绍

GUI中主要调节方式有：

- 通过拖拽slider对浮点型变量的值进行调整
- 通过button按钮激活一段程序的执行

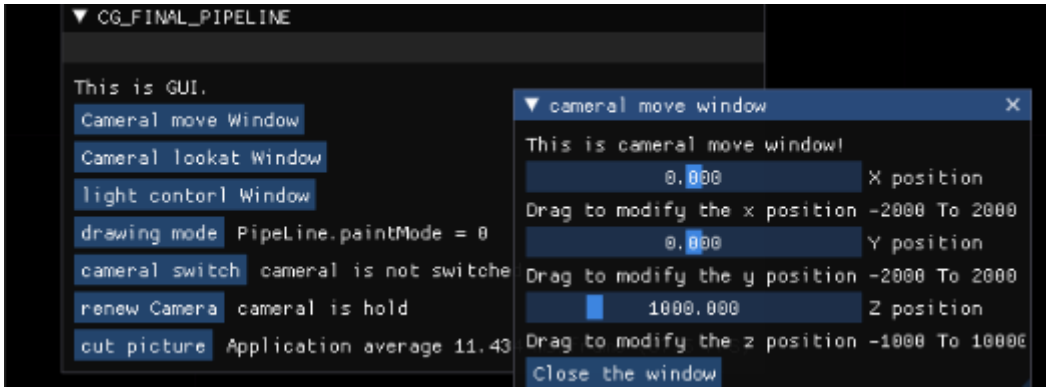
其功能区概览如下：



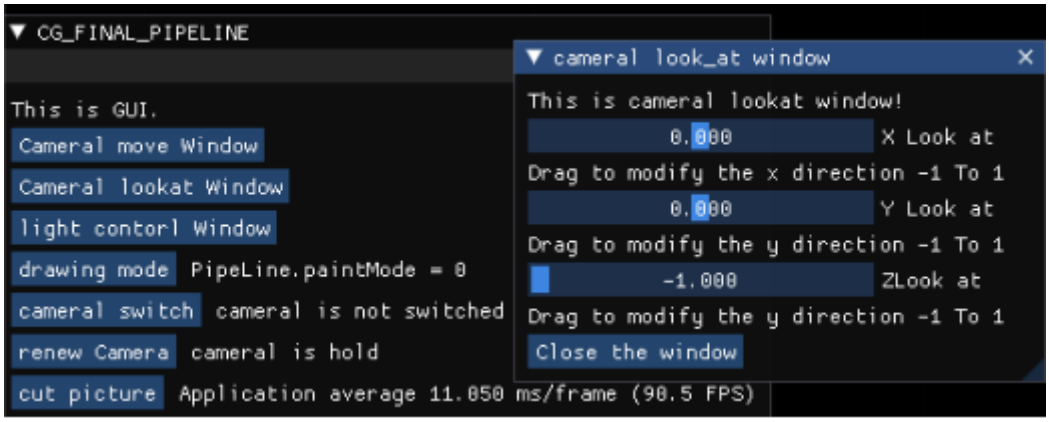

单击第一行的“cameral move window“会激活相机控制GUI窗口，点击后弹出一个新的GUI窗口，可以对相机的位置进行调整。第二行“cameral lookat window“、第三行“light control window“分对应控制相机面向和灯的位置的函数，原理和效果与第一行相似。

单击“drawing mode”可以对绘图模式进行调整，每次单击产生类似按下”p”键的效果，在PipeLine.paintMode小于3时，其值会加1，否则不变并使PipeLine.end为1，发出绘图完毕的信号；同时右边的文本中会显示出PipeLine.paintMode的值。单击“cameral_switch”会切换cameral模式，在不同相机模式下观察角度不同，并且在switched的情况下会触发边缘检测。单击“renew Cameral”会重置相机的位置等等属性；单击“cut picture”则可以进行截屏操作，图片会存放在相同目录的img文件夹下。

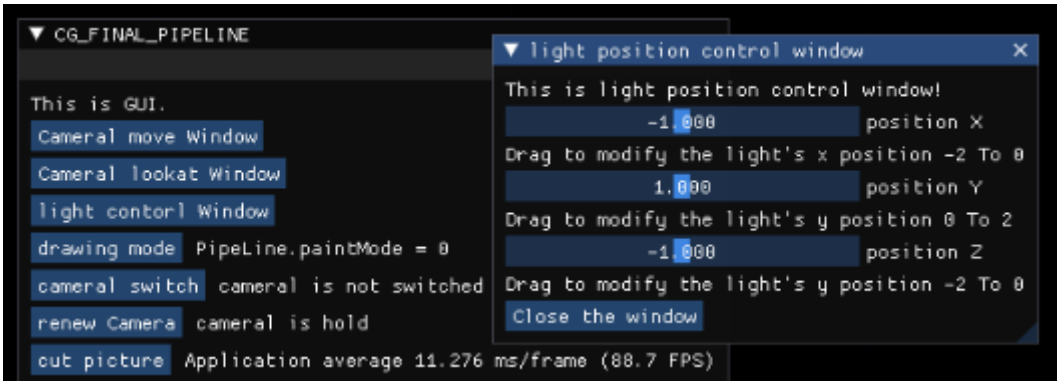
相机位置控制窗口的格局如图所示，我们通过拖动slider分别对X、Y、Z的值进行调整。我们将X、Y、Z的范围分别设置为[-2000,2000],[-2000,2000],[-1000,10000]。




相机面向的控制窗口的格局如图所示，我们通过拖动slider分别对X direction、Y direction、Z direction的值进行调整。我们将X direction、Y direction、Z direction的范围分别设置为[-1,1],[-1,1],[-1,1]。




灯光位置的控制窗口的格局如图所示，我们通过拖动slider分别对X、Y、Z的值进行调整。我们将X、Y、Z的范围分别设置为[-2,0],[0,2],[-2,0]。



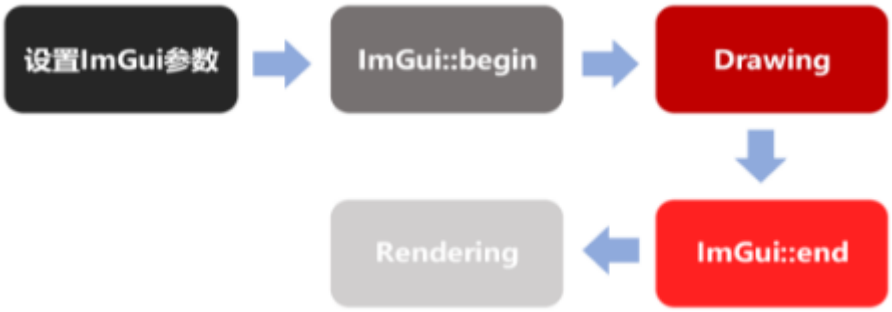

### 2.8.3 基本函数介绍

Imgui绘图常用函数

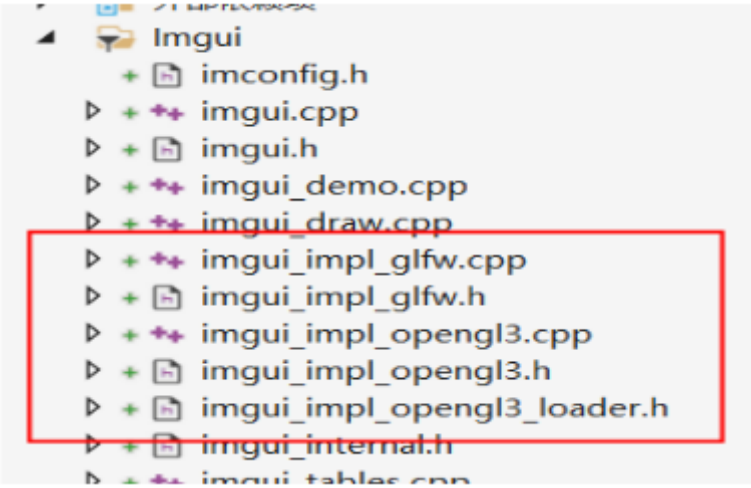
| 函数名 | 功能 | 参数意义 |
| --- | --- | --- |
| ImGui_ImplOpenGL3_NewFrame(); | 开启 Imgui的frame | |
| ImGui_ImplGlfw_NewFrame(); | 开启 Imgui的frame | |
| ImGui::Begin() | 开始ImGUI的绘图 | GUI名字，设置信息等 |
| ImGui::End() | 停止ImGUI的绘图 | |
| ImGui::NewLine() | 进入新的一行绘图 | |
| ImGui::SameLine(); | 提示下一行不用进行换行 | |
| ImGui::Text(); | 绘制文字信息 | 字符串内容 |
| ImGui::Render() | 对绘制内容进行渲染 | |
| ImGui::SliderFloat("float", &f, 0.0f, 1.0f); | 设置一个拖动调整的滑动框 可以对变量直接进行调整 | 第一个参数是字符串，第二个参数是变量的指针，最后是变量的取值范围 |
| ImGui::Button("Demo Window"); | 设置一个勾选框，可以将后面的变量设置为0、1的值 | 第一个为勾选框的名称 |

#### 2.8.4 Imgui绘制流程与代码实现

Imgui绘制流程图




我们引用的ImGUI的文件如图所示，其中ImplOpenGL3和 ImplGlfw是区别于其他绘图模式的重要文件：



对于一个完整的ImGUI绘图流程而言，我们首先需要设置ImGUI的绘制参数。例如，在我们的程序中我们进行了如下的设置。前三行的意义为创建新的frame供ImGUI使用；后面则是设置ImGUI的一系列的绘图参数。

```
ImGui_ImplOpenGL3_NewFrame();
ImGui_ImplGlfw_NewFrame();
ImGui::NewFrame();
const auto flags =
ImGuiWindowFlags_AlwaysAutoResize |
ImGuiWindowFlags_NoSavedSettings |
ImGuiWindowFlags_MenuBar;
```

上面设置的参数随后就在Begin中被用到，如下图所示；在Begin中我们传入窗口的名称、设置的flag值等等信息。随后，我们就进入绘制GUI的阶段，调用ImGUI的函数以及传入程序中的相关变量来进行绘制。在最后我们调用End函数结束并调用Render函数进行呈现。

```
//开始
if (!ImGui::Begin("CG_FINAL_PIPELINE", nullptr, flags)) {
ImGui::End();
}
……
//结束
ImGui::End();
```

### 2.9 GLFW部分介绍

#### 2.9.1 GLFW简介

在本次大作业中，我们最终选择使用更先进的glfw框架，并将GLUT更换为了GLFW，同时保留gl的传统绘图模式，并没有使用着色器的绘图方式。

（1）GLFW的特点

GLFW是一种更强大的OpenGL库，相比GLUT而言实用性更强，兼容更多的开源库，往往可以较为简便地实现glut难以完成的功能。GLFW不是采用回调机制的，它是由main函数中的循环来刷新和display的。正是这个特点使得GLFW的窗口管理更为容易，许多开源库的面向GLFW的程序在涉及到窗口有关的操作时，例如我们项目中使用的ImGUI库的针对GLFW编写的GUI绘制程序，大多不会造成阻塞，并且可以管理若干窗口的同时运行。

（2）GLFW的兼容性

GLFW不兼容GLUT库，但它是兼容GL和GLU库的，我们可以在vs的安装文件里找到。这意味着我们可以使用glxxxxx函数，但不能使用glutxxx函数。好在glut的大多数函数并不是其独有的，glut也是基于GL和GLU的。因此，我们在将GLUT更换为GLU库时，仅仅是删除了glut中快捷绘制立方体的函数和glutswapbuffer的函数。前者是快捷绘制solid的立方体，是不必要的；后者是交换双缓冲，以优化绘图闪烁的情况，但是GLFW本身也有用在每次主循环末尾的glfwswapbuffer函数可以很好地处理这种情况，因此我们去掉后也没有产生意外的结果。

#### 2.9.2 GLFW的配置以及本次项目中的配置

主要是需要配置ｖ ｓ项目属性中的C++引用目录或者包含目录、库目录、链接器中的附加依赖项。同时，需要使用x64的glfw32.lib，否则会报=出现目标计算机(x86)与库文件(x64)冲突的错误。另外，在附加依赖项中除了添加glfw3.lib外还要添加opengl32.lib。

除此之外，需要再手动添加GLU函数，可以在．\Windows Kits\10路径下的include和lib中找到对应文件，然后将他们复制到当前工程文件夹中，设置好引用目录和库目录即可使用。

#### 2.9.3 GLFW绘制流程

（1）创建窗口并初始化

使用glfwCreateWindow来创建window，传入的参数为窗口名称与窗口宽度与高度；同时我们在glfwWindowHint中设置glfw版本为 3 以及适用于opengl。随后我们通过glfwMakeContextCurrent激活上下文。

```
window = glfwCreateWindow(glfw_window_wide, glfw_window_height,
"GLFW window for CG_FINAL", NULL, NULL);
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
glfwMakeContextCurrent(window);
```

（2）设置监听以及绘图函数初始化

设置一系列的监听函数并对绘图、光照、GUI进行初始化

```
    glfwSetFramebufferSizeCallback(window, framebuffer);
    glfwSetCursorPosCallback(window, glfw_mouse);
    glfwSetMouseButtonCallback(window, glfw_mouse_button);
    init();
    glewInit();
    init_gui(window);
```

（3）主循环

循环中主要有监听函数、绘图函数、reshape函数、GUI绘制函数、交换缓冲函数、检查相关事件的函数。

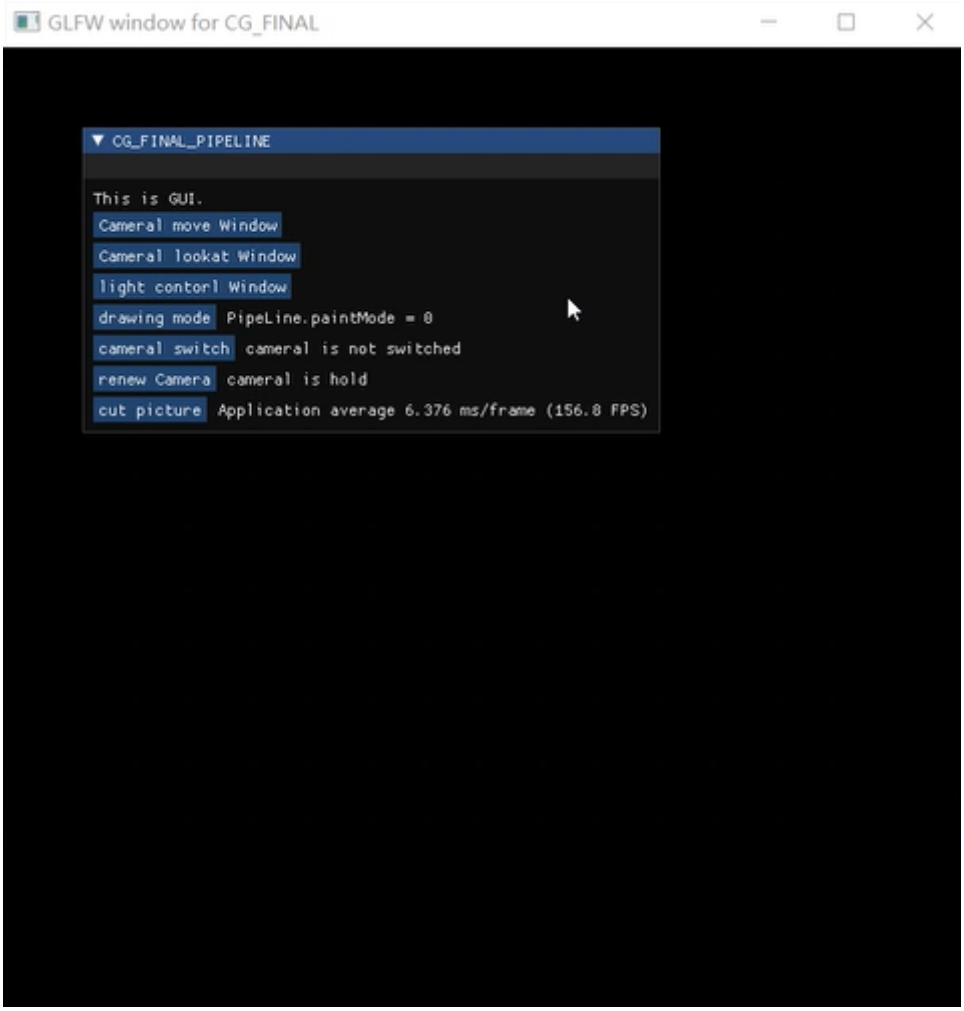
```
while (!glfwWindowShouldClose(window)){
       KEYInput(window);
       display_fw();
       reshape(glfw_window_wide, glfw_window_height);
       drawGUI();
       glfwSwapBuffers(window);
       glfwPollEvents();
    }
```

（4）结束绘制

使用glfwTerminate来结束glfw的绘制。

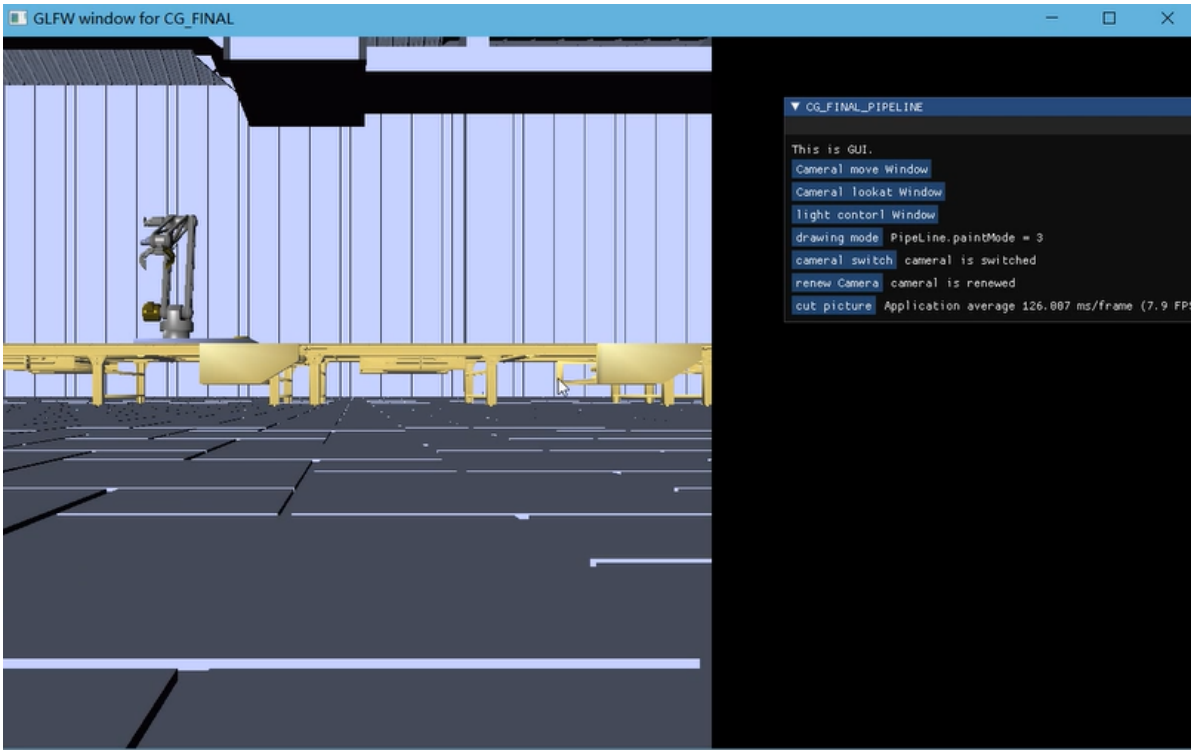
```
glfwTerminate();
```

# 3. 运行结果截图

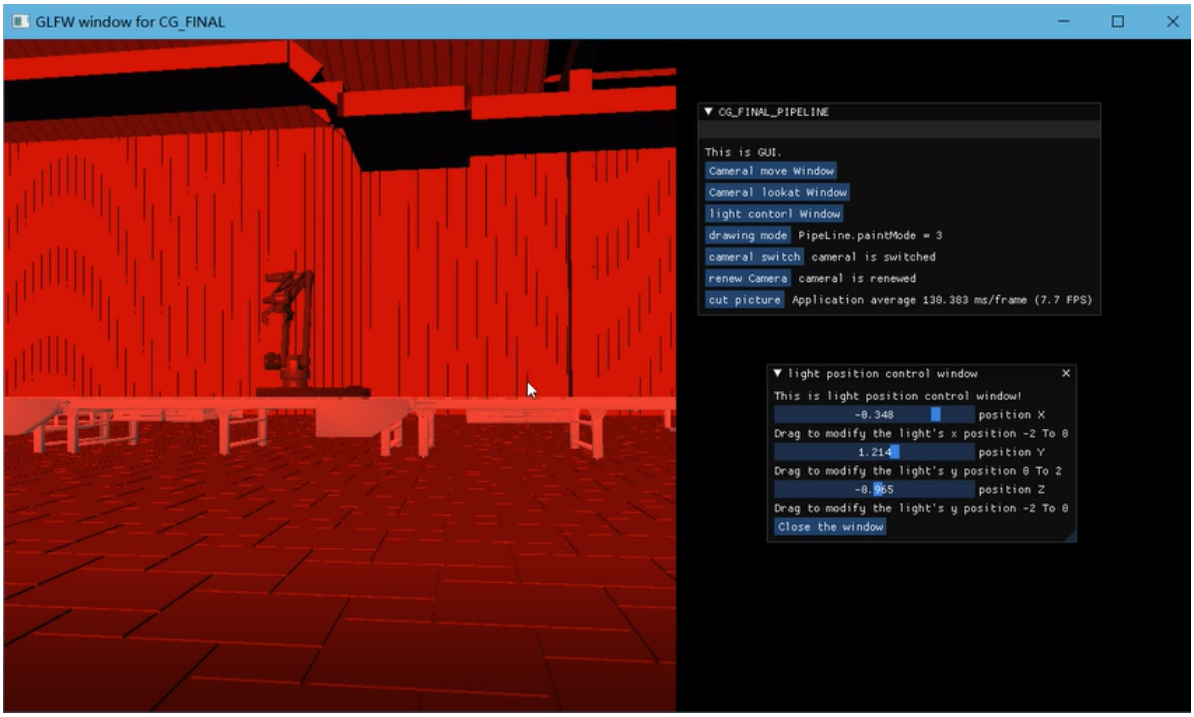



开始运行后的窗口。

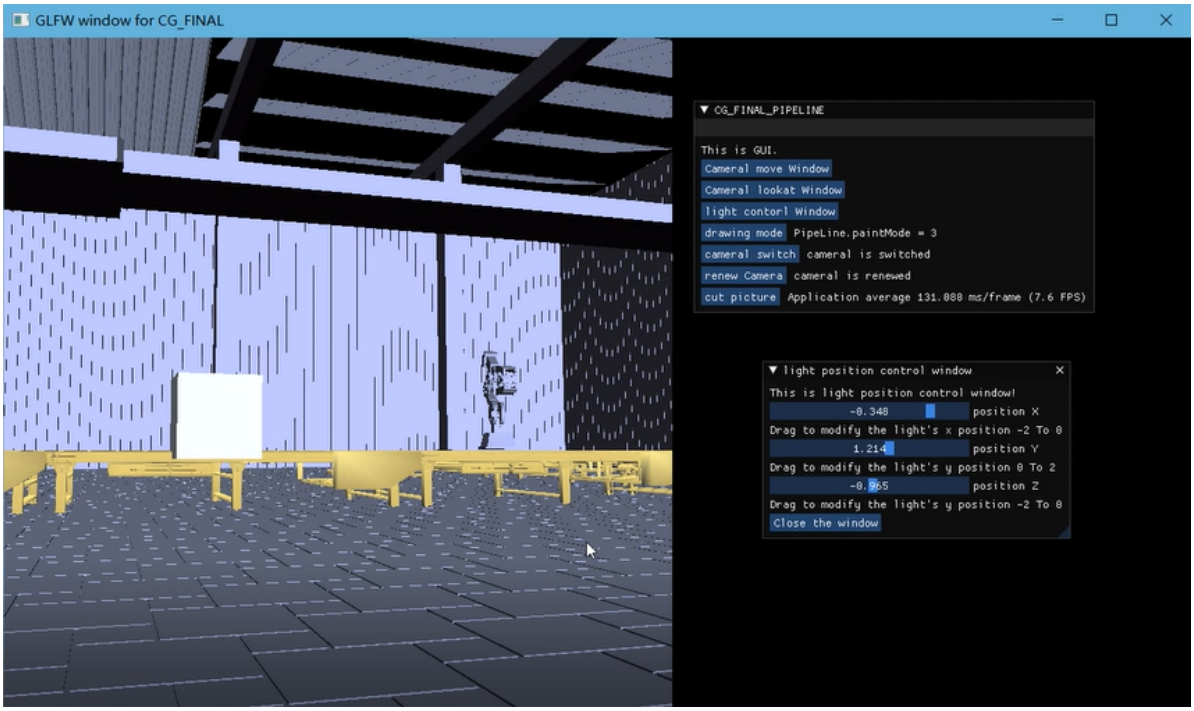

编辑流水线与流水线上的组件




漫游模式下观察编辑的流水线场景




改变光照颜色


GLFW window for CG_FINAL
OG_FINAL_PIPELINE
This is GUI.
Cameral move Window
Cameral lookat Window
light contorl Window
drawing mode PipeLine.paintMode = 3
cameral switch cameral is switched
renew Camera cameral is renewed
cut picture Application average 131.088 ms/frame (7.6 FPS)
light position control window
This is light position control window!
-0.348
position X
Drag to modify the light's x position -2 To 0
1.214
position Y
Drag to modify the light's y position 0 To 2
-0.765
position Z
Drag to modify the light's y position -2 To 0
Close the window

流水线上的物体移动