

# 虚拟环境交互绘制系统

3170104168 上官越

## 1. 任务：

- 熟悉图形绘制流水线，了解 OpenGL、DirectX、OpenGL ES 或 Vulkan 的基本原理，在笔记本或手机平台上自主建立交互三维图形环境。

## 2. 目标（实现 4 个功能）：

- 从文件中读取场景模型数据，应用所提供的函数，实现图形流水线的各个环节；
- 提供双目左右画面的同步绘制；
- 提供含多个物体的场景，用手指触控或鼠标来交互选择物体或光源，改变所选择物体、光源的方位和绘制参数；
- 能绘制出虚拟景物的纹理和所产生的阴影；
- 用手指触控或鼠标实现摄像机的自由飞行漫游功能。

## 3. 目标分析

### 3.1 图形流水线

#### 图形流水线



如图所示为图形流水线的基本流程，分为模型变换-》光照着色-》视点变换-》裁剪-》投影变换-》扫描转换-》显示几大步骤。

Opengl 中有许多存在的函数或框架可以帮助我们解决这个问题，要先看懂流水线每个环节的实际操作才可进行实现。

模型变换即表示物体从局部坐标系到世界坐标系的变换；光照着色应用 glsl 着色器，视点变换，裁剪和投影变换与 glulookat, gluperspective 等函数有关。通过一系列流程，最后绘制物体。

### 3.2 双目绘制

本质就是将屏幕分成两个部分，对应同一个空间坐标系。然而，两个部分的相机位置不同，所以导致出现所示物体出现偏差。二者所看的物体视点中心应当相同。

### 3.3 鼠标选择和移动

鼠标选择可以用多种方式实现，本质是如何将三维空间中的 3D 物体体现在特定视角下的二维平面上。可考虑的解决方法有：

- ① 采用 opengl 自带的解决方案，采用 glselectbuffer 指定存储点击记录的数组来判定选中，glpickmatrix 设置投影矩阵。如果选中模式开启，则用 glinitnames() 进行重绘，将待选中的物体通过 glpushname(id) 加入栈内。
- ② 将物体的中心点做一个三维到二维的变换，当鼠标点击距离与其不超过特定距离时判定选中
- ③ 将物体设定在一个立方体内，当鼠标点击的直线与该立方体相交时，判定选中。
- ④ 当鼠标点击的直线与物体的面相交时，判定选中。

移动可以选择不同的设定方式，如以相机所在位置为圆心，可限制一个自由度，然后鼠标 x, y 方向的移动表示球坐标系的两个偏移角，对物体进行等距移动；如 y 方向上的移动仅表示 y 坐标上的移动，x 方向的移动表示为相机视线在水平面上的法线方向上的移动。

### 3.4 绘制纹理

纹理的绘制本质是将 obj 上的点绑定到图片上的某个像素值，因此赋予了它一个颜色。再通过着色器对其进行渲染。

### 3.5 自由飞行漫游功能

自由飞行漫游即相机位置的自由移动和视角的随意转换。  
分别通过键盘（位置）和鼠标（视角）完成。

## 4. 数据结构

### 4.1 模型类：用来读取输入文件中的信息

在 loadModel 中，我们使用 Assimp 来加载模型至 Assimp 的一个叫做 scene 的数据结构中。经过 assimp 的一套处理将其数据转换为 mesh，并处理纹理和材质信息。（model 中有一部分没有写进来，因为没有使用）

```
class Model
{
public:
    vector<Texture> textures_loaded;

    // stores all the textures loaded so far, optimization to make sure
textures aren't loaded more than once.

    vector<Mesh> meshes; // render unit
    string directory;

    bool obj_chosen; // if the object has been selected
    vec3 obj_pos; // translate
    vec4 rotate; // rotate
    vec3 scale; // scale

    //get the parameters of how to render the model by matrix
    void getmatrix(vec3 obj_pos, vec4 rotate, vec3 scale) ;

    // constructor, filepath to a 3D model.
    Model(string const &path, bool gamma = false) ;

    // draws the model, and thus all its meshes
    void Draw(Shader shader);

private:
    // loads a model with supported ASSIMP extensions from file and
stores the resulting meshes in the meshes vector.

    void loadModel(string const &path);

    // processes a node in a recursive fashion.
```

```

void processNode(aiNode *node, const aiScene *scene);

Mesh processMesh(aiMesh *mesh, const aiScene *scene);

    // checks all material textures of a given type and loads the
textures if they're not loaded yet.

    // the required info is returned as a Texture struct.

    vector<Texture> loadMaterialTextures(aiMaterial *mat, aiTextureType
type, string typeName);
};

```

## 4.2 mesh 类

Mesh 代表的是单个的可绘制实体。

一个网格应该需要一系列的顶点，每个顶点包含一个位置向量、一个法向量和一个纹理坐标向量。一个网格还应该包含用于索引绘制的索引以及纹理形式的材质数据（漫反射/镜面光贴图）。

将所有需要的向量储存到一个叫做 Vertex 的结构体中，可以用它来索引每个顶点属性。除了 Vertex 结构体之外，还需要将纹理数据整理到一个 Texture 结构体中。

在初始化中，我们将所有必须的数据赋予了网格，我们在 setupMesh 函数中初始化缓冲，并最终使用接受了着色器的 Draw 函数来绘制网格。

```

struct Vertex {

    glm::vec3 Position;

    glm::vec3 Normal;

    glm::vec2 TexCoords;

    glm::vec3 Tangent;

    glm::vec3 Bitangent;

};

struct Texture {

    unsigned int id;

    string type;

    string path;

};

```

```

class Mesh {
public:
    /* Mesh Data */
    vector<Vertex> vertices;
    vector<unsigned int> indices;
    vector<Texture> textures;
    unsigned int VAO;

    /* Functions */
    // constructor
    Mesh(vector<Vertex> vertices, vector<unsigned int> indices,
vector<Texture> textures); nder the mesh
    void Draw(Shader shader);

private:
    /* Render data */
    unsigned int VBO, EBO;

    /* Functions */
    // initializes all the buffer objects/arrays
    void setupMesh();
};
#endif

```

#### 4.3 shader 类:

从硬盘读取着色器 (glsl), 然后编译并链接它们, 并对它们进行错误检测。Set.. 系列函数用于给着色器传入参数。

`glUseProgram(ID)`; 使用着色器。

构造函数读入, 编译着色器。

```

class Shader
{
public:

```

```

    unsigned int ID;

    Shader(const char* vertexPath, const char* fragmentPath, const
char* geometryPath = nullptr);

    // activate the shader

    // -----
-----

    void use();

    // utility uniform functions

    // -----
-----

    void setBool(const std::string &name, bool value) const;

    // -----
-----

    void setInt(const std::string &name, int value) const;

    // -----
-----

    void setFloat(const std::string &name, float value) const;

    // -----
-----

    void setVec2(const std::string &name, const glm::vec2 &value)
const;

    void setVec2(const std::string &name, float x, float y) const;

    // -----
-----

    void setVec3(const std::string &name, const glm::vec3 &value)
const;

    void setVec3(const std::string &name, float x, float y, float z)
const;

    // -----
-----

```

```

    void setVec4(const std::string &name, const glm::vec4 &value)
const;

    void setVec4(const std::string &name, float x, float y, float z,
float w);

    // -----
-----

    void setMat2(const std::string &name, const glm::mat2 &mat) const;

    // -----
-----

    void setMat3(const std::string &name, const glm::mat3 &mat) const;

    // -----
-----

    void setMat4(const std::string &name, const glm::mat4 &mat) const;

private:

    void checkCompileErrors(GLuint shader, std::string type);
};

```

#### \*4.4 glsl 着色器

把顶点和片段着色器储存为两个叫做.vs 和.fs 的文件

顶点着色器从顶点数据中直接接收输入。定义 location 这一元数据指定输入变量，协助顶点数据管理。于是可以在 CPU 上配置顶点属性。

片段着色器，它需要一个 vec4 颜色输出变量，因为片段着色器需要生成一个最终输出的颜色。

在顶点着色器中，输入为顶点的位置，法向量和纹理坐标，输出经过投影变换后的结果。在面着色器中，接受从顶点着色器传来的输入，并从 cpu 中获取光照、纹理参数。分别计算出环境光，材料作用得出的 ambient，diffuse 和 specular 参数，然后与纹理的 diffuse 进行相结合获得最终人眼所见的着色结果。

## 5. 主函数具体实现

### 5.1 图形流水线

流程:用 model 类读入 obj->设定相机位置/相机参数(gluperspective)

-》 矩阵变换 -》 加载着色器-》 着色器绘制  
设定为单一光照，背景不可移动，光源，导入的物体可移动。

## 5.2 双目绘制

设定便来给你布尔值 `flush`，在每次渲染时 `flush = !flush`，用于 `glfwSwapBuffers(window)` 以及双目绘制。`Flush = true` 时绘制左眼，否则绘制右眼。通过 `glScissor` 和 `glViewport` 对屏幕进行分区，交替绘制。两边唯一的差别是相机的位置不同，存在一个固定数值的偏差。

```
if (!flush) {
    eyemode = LEFT_CAMERA;

    glScissor(0, 0, SCR_WIDTH / 2, SCR_HEIGHT);
    glViewport(0, 0, SCR_WIDTH / 2, SCR_HEIGHT);
}
else {
    eyemode = RIGHT_CAMERA;

    glScissor(SCR_WIDTH / 2, 0, SCR_WIDTH / 2, SCR_HEIGHT);
    glViewport(SCR_WIDTH / 2, 0, SCR_WIDTH / 2, SCR_HEIGHT);
}
```

## 5.3 鼠标选择和移动

一开始尝试使用 3.3 ①中方法解决，但是一直有 bug 无法选中。后来改为了 3.3 中的②实现。

当中心点与点击点的距离差距小于一定特定距离且在各个物体中相对偏离最小时判定为选中。这种选中其实是不够精准的。

```
double minDis = 0.1;
static int minIndex = -1;
for (int i = 0; i < objs.size(); i++) {
    vec4 pos = projection * view * translate(mat4(1.0f),
objs[i].obj_pos) * vec4(0.0, 0.0, 0.0, 1.0);

    double dis = (pos.x / pos.w - x) * (pos.x / pos.w - x)
        + (pos.y / pos.w - y) * (pos.y / pos.w - y);

    dis = dis / objs[i].scale[0];
```



```

        if (abs(dis) < minDis) {
            minDis = abs(dis);
            minIndex = i;
        }
    }

    vec4 pos = projection * view * translate(mat4(1.0f), light_pos) *
vec4(0.0, 0.0, 0.0, 1.0);

    double dis = (pos.x / pos.w - x) * (pos.x / pos.w - x)
        + (pos.y / pos.w - y) * (pos.y / pos.w - y);
    if (abs(dis) < minDis) {
        minDis = abs(dis);
        movelight = true;
    }

    if (minIndex != -1 && !movelight) {
        objs[minIndex].obj_chosen = true;
    }

    if (action == GLFW_RELEASE && (minIndex != -1 || movelight ==
true)) {
        if (minIndex != -1) {
            objs[minIndex].obj_chosen = false;
            cout << objs[minIndex].obj_pos.x <<
objs[minIndex].obj_pos.y << objs[minIndex].obj_pos.z << endl;
        }

        movelight = false;
        minIndex = -1;
    }
}

```

松开鼠标后停止移动。

## 5.4 绘制纹理

在 model 中读取处理，在着色器中渲染。

```
//Model 的面着色器代码
#version 330 core
out vec4 FragColor;

struct Material {
    sampler2D diffuse;
    sampler2D specular;
    float shininess;
};

struct Light {
    vec3 position;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;

    float constant;
    float linear;
    float quadratic;
};

in vec3 FragPos;
in vec3 Normal;
in vec2 TexCoords;

uniform vec3 viewPos;
uniform Material material;
uniform Light light;
uniform sampler2D texture_diffuse1;

void main()
{
    // ambient
    vec3 ambient = light.ambient * texture(material.diffuse,
TexCoords).rgb;

    // diffuse
    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(light.position - FragPos);
    float diff = max(dot(norm, lightDir), 0.0);
```

```

    vec3 diffuse = light.diffuse * diff * texture(material.diffuse,
TexCoords).rgb;

    // specular
    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0),
material.shininess);
    vec3 specular = light.specular * spec * texture(material.specular,
TexCoords).rgb;

    // attenuation
    float distance = length(light.position - FragPos);
    float attenuation = 1.0 / (light.constant + light.linear * distance
+ light.quadratic * (distance * distance));

    ambient *= attenuation;
    diffuse *= attenuation;
    specular *= attenuation;

    vec3 result = ambient + diffuse + specular;

    FragColor = vec4(result, 1.0) + texture(texture_diffuse1, TexCoords);
}

```

## 5.5 自由飞行漫游功能

主程序中相机参数如下。

分别表示相机的位置，两眼视差，左右眼看的位置，视线角度等。

```

//cameras
vec3 lefteye (-0.1f, 0.5, 3.0f);
vec3 righteye(0.1f, 0.5, 3.0f);
vec3 delta (0.2, 0, 0);

vec3 left_viewat(-0.3f, 0.0f, 2);
vec3 right_viewat(-0.3f, 0.0f, 2);

int eyemode = -1;
GLfloat theta = 0.0f; //水平旋转的角度

```

```
GLfloat viewUp = 0.0f;      //上下旋转的角度
GLfloat speed = 0.05f;
vec3 headup(0, 1, 0);
```

自由漫游的实现:

计算视线向量, 求法向量, 由这两个向量决定水平面移动的方向。

```
else if (key == GLFW_KEY_W && action != GLFW_RELEASE) {
    lefteye.x += (left_viewat.x - lefteye.x)*speed;
    lefteye.z += (left_viewat.z - lefteye.z)*speed;
}

else if (key == GLFW_KEY_S && action != GLFW_RELEASE) {
    lefteye.x -= (left_viewat.x - lefteye.x)*speed;
    lefteye.z -= (left_viewat.z - lefteye.z)*speed;
}

else if (key == GLFW_KEY_A && action != GLFW_RELEASE) {
    lefteye.x += (left_viewat.z - lefteye.z)*speed;
    lefteye.z += -(left_viewat.x - lefteye.x)*speed;
}

else if (key == GLFW_KEY_D && action != GLFW_RELEASE) {
    lefteye.x -= (left_viewat.z - lefteye.z)*speed;
    lefteye.z -= -(left_viewat.x - lefteye.x)*speed;
}
```

根据相机面向的位置和相机位置决定水平面上的行进方向, 由 w, a, s, d 控制。竖直方向上的移动 i, d 则直接进行加减即可。

将鼠标的移动转化为水平、数值方向的相对偏移角, 进而根据相机所在位置求出新的视点。

```
float xoffset = xpos - lastX;
float yoffset = lastY - ypos;

lastX = xpos;
```

```

        lastY = ypos;

        theta += GLfloat(xoffset) / 1000;        //旋转改变量
        viewUp += GLfloat(yoffset) / 1000;        //上下改变量

        if (viewUp <= -80)viewUp = -80;
        if (viewUp >= 80)viewUp = 80;

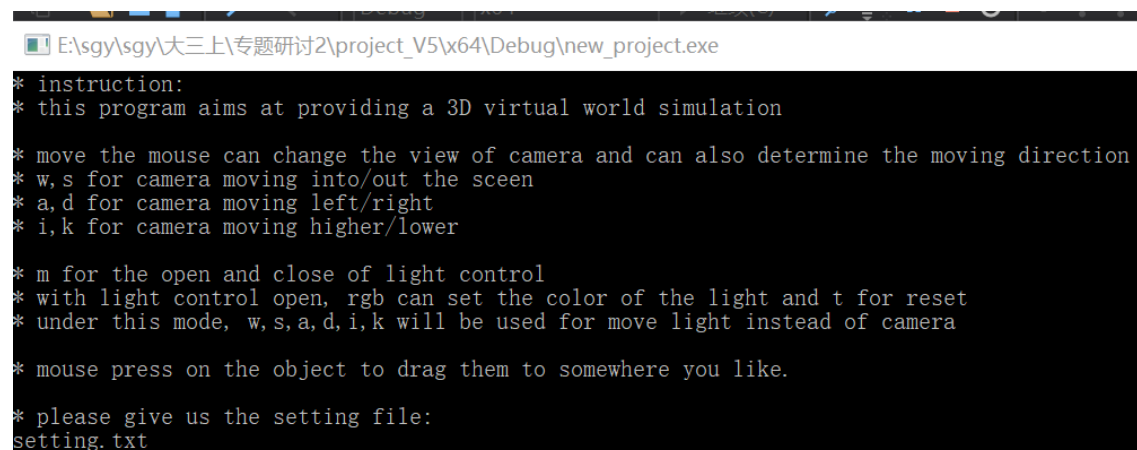
        left_viewat.x = float(lefteye.x +
cos(theta));                                // 新的参考点的位置
        left_viewat.z = float(lefteye.z + sin(theta));
        left_viewat.y = float(lefteye.y + viewUp);

        right_viewat = left_viewat;

```

## 6. 展示样例和操作指南

输入存放了基础设定的文件



```

E:\sgy\sgy\大三上\专题研讨2\project_V5\x64\Debug\new_project.exe
* instruction:
* this program aims at providing a 3D virtual world simulation

* move the mouse can change the view of camera and can also determine the moving direction
* w,s for camera moving into/out the sceen
* a,d for camera moving left/right
* i,k for camera moving higher/lower

* m for the open and close of light control
* with light control open, rgb can set the color of the light and t for reset
* under this mode, w,s,a,d,i,k will be used for move light instead of camera

* mouse press on the object to drag them to somewhere you like.

* please give us the setting file:
setting.txt

```

```

objs/cube.obj
4 0 4
0 0 1 0
1 1 1

objs/GrandBanks.obj
-5 -1 -5
0 0 1 0
1 1 1

objs/ONI.obj
-5 2.9 2.2
0 0 1 0
1 1 1

light_ambient
0.3 0.3 0.3

light_diffuse
0.3 0.3 0.3

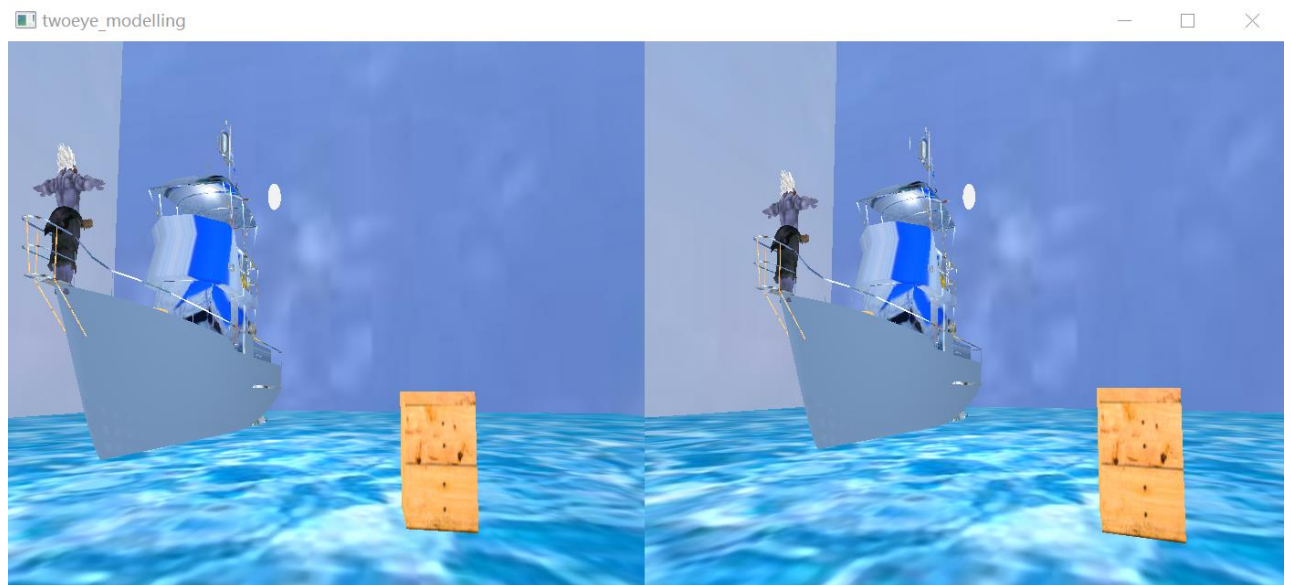
```

左图为一个输入样例

基础设定的格式/种类如下:

1. Obj 文件的目录和它的 translate, rotate, scale 矩阵
2. Light\_ambient, light\_diffuse 参数
3. Deltaeye: 双眼视差, 输入一个 vec3 向量
4. Camera: 设定左眼位置输入一个 vec3 向量

Glwf 窗口开始运行, 左边为左眼视图, 右边为右眼视图



W, a, s, d 可在水平面上四方移动相机

I, k 可在数值方向上移动相机

鼠标旋转可以改变视角

M: 开启光照设定模式, 开启后可通过光标改变光照的 ambient 和 diffuse 的数值 (增强/减弱)

N: 开启选中模式。点击物体即可进行拖动, 当点击物体时候摁 b, 即可使物体旋转。

## 相机漫游

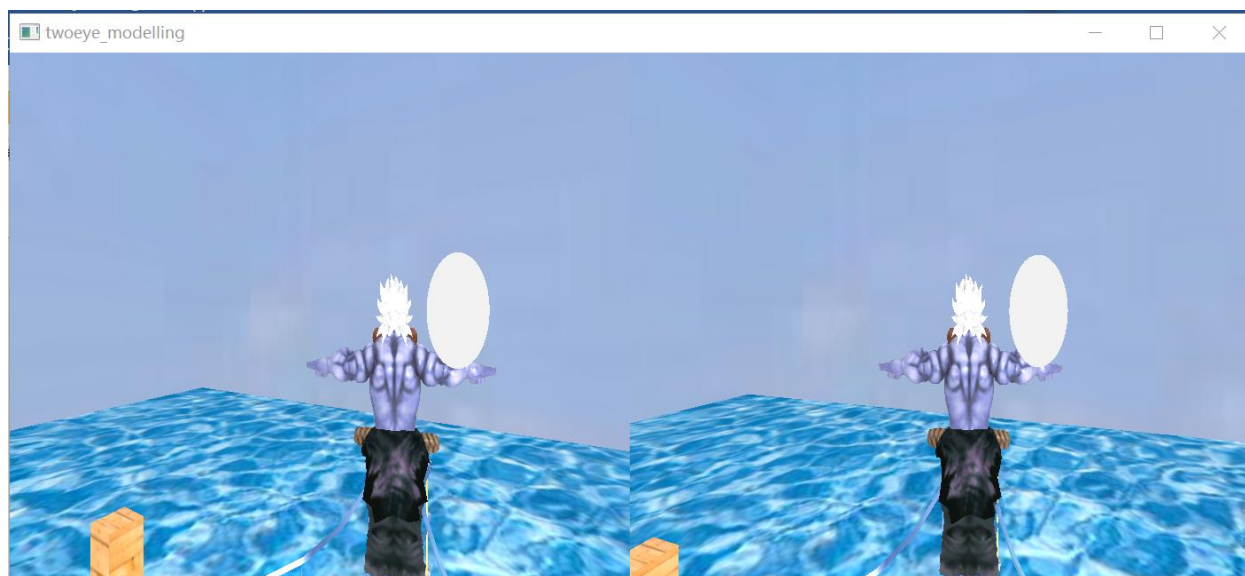


## 增强光照后

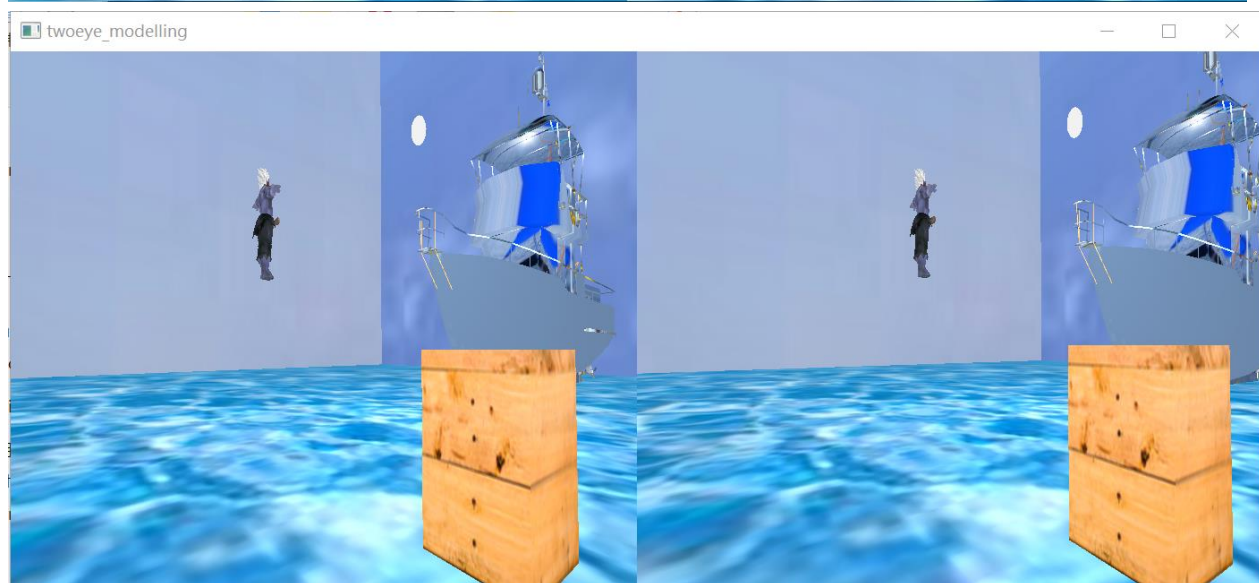
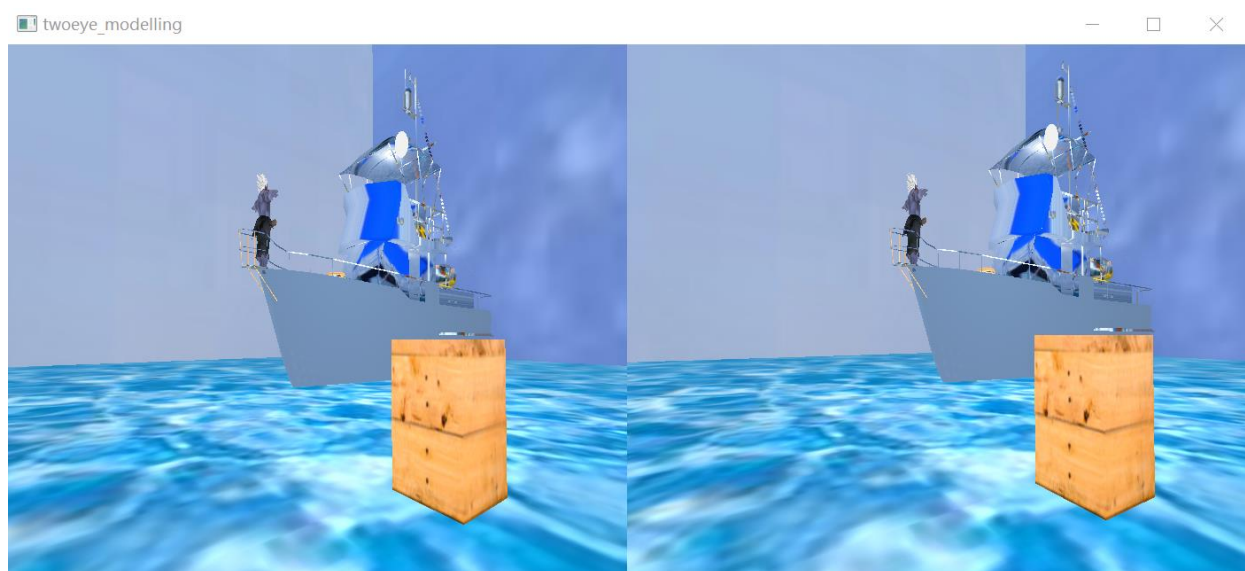


## 旋转物体



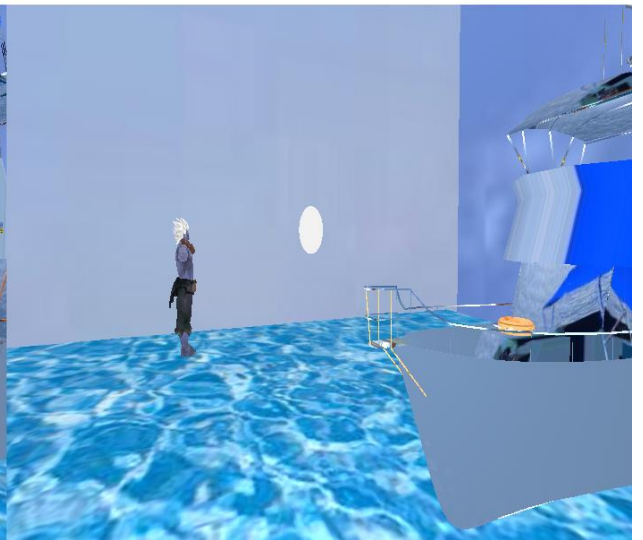
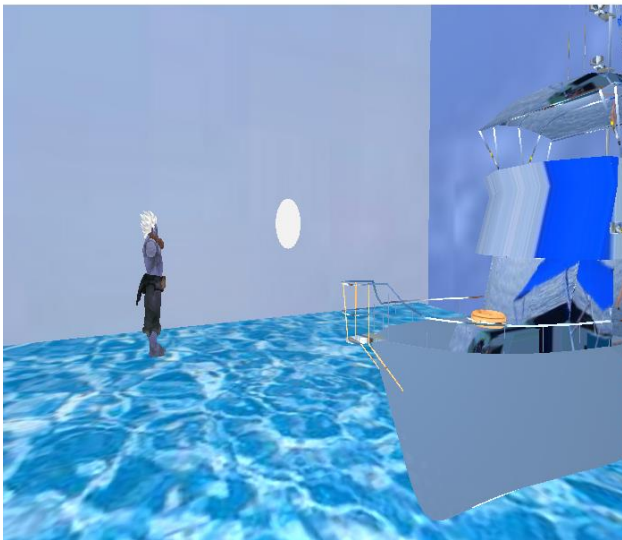


一些截图：

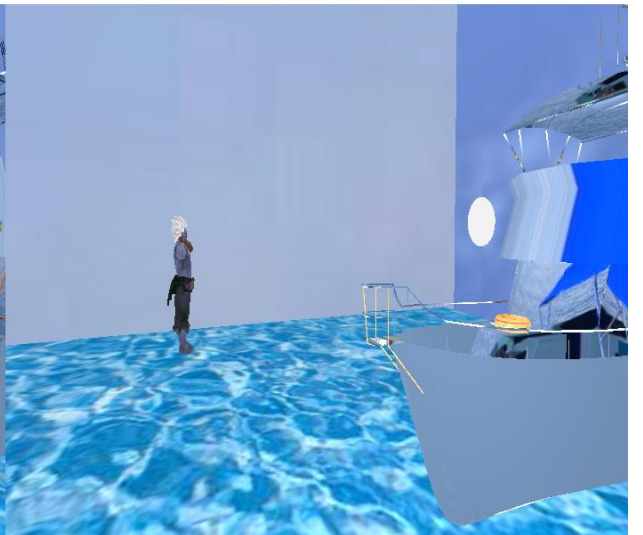
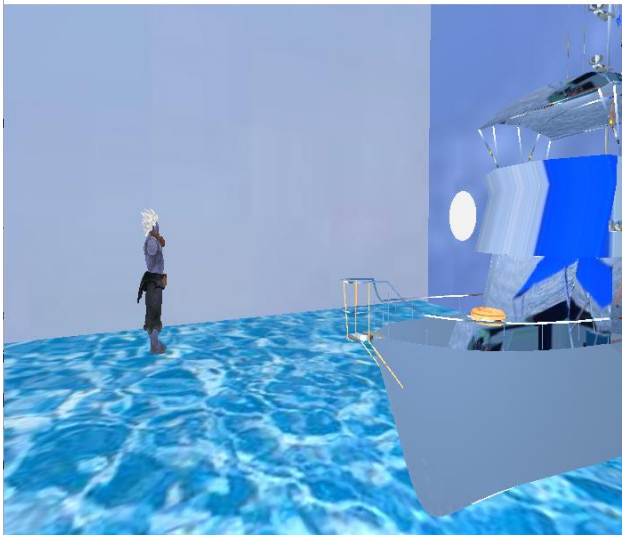




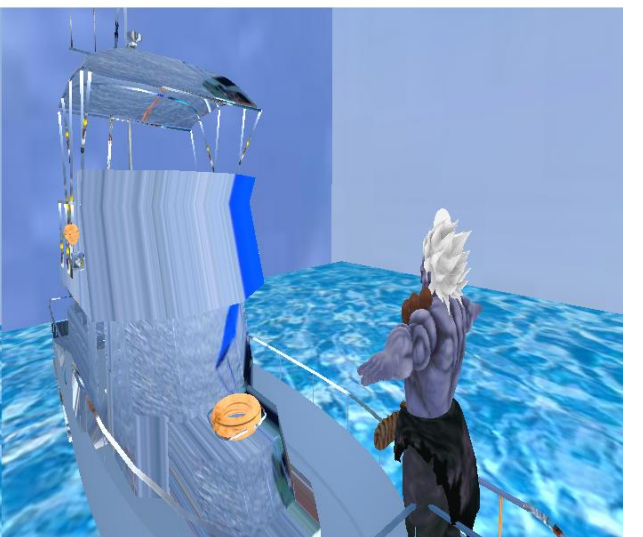
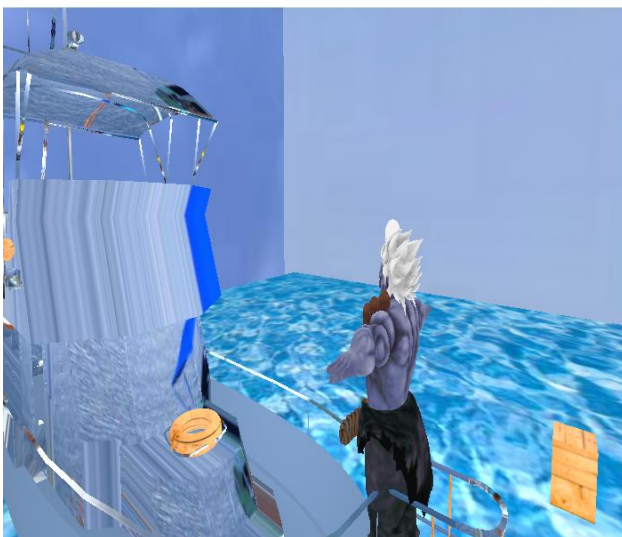
twoeye\_modelling



twoeye\_modelling



twoeye\_modelling



## 7. 改进方案

在物体的选中上还有提高的空间，视线与物体的面直接作交应该是最为精确的。

在参数的设置上，可以变得更为灵活。

## 8. 总结

通过本次实验，对 opengl 进行了从 0 到 1 的探索，熟悉了许多基本操作，对 gpu, cpu, 图形渲染流程等方面都有了更深刻的理解。

从配置上来说，这次大程的编写让我属于了 opengl 许多存在的内部以来，和不同库之间的依赖关系。如果调用顺序不对或者库的组合使用不对，在编译运行上会出现比较多的问题。

在具体实现上，我熟悉了 opengl 的渲染，较为熟练地掌握了 opengl 工局。

## 9. 参考教程

<https://learnopengl-cn.github.io/intro/>