



《计算机图形学实验》 实验报告

(期末 project)

学院名称：数据科学与计算机学院

专业（班级）：17 级计科

学生姓名：黄镇 江金昱 赖少玄
李静雯 李世然

学号：17341063 17341067 17341072
17341080 17341082

时间：2019 年 12 月 28 日

期末题：美丽的海底世界

一、实现目标及函数框架

（一）实现目标

1. 使用二次曲面贴图实现“海底天空”
2. 通过 obj 模型的载入和纹理贴图实现静态物体的渲染
3. 实现半透明物体的渲染
4. 使用雾模拟海底朦胧的视觉效果
5. 设置全局光照，为物体添加阴影，增加真实感
6. 模拟不同大小（几何模型的缩放）、不同种类鱼群向不同方向进行随机游走（以此来表示流体可视化）
7. 水草的飘动：几何模型的形变

（二）实验要求

1. 实现较复杂的算法（全局光照，网格细分与简化，几何模型的形变、压缩，顺序无关的半透明物体渲染等）；
2. 创建复杂场景、动画、交互；
3. 可视化内容（标量场、流场可视化，及其与几何模型的结合）

（三）实验环境

Qt Creator 4.9.1 (Enterprise)

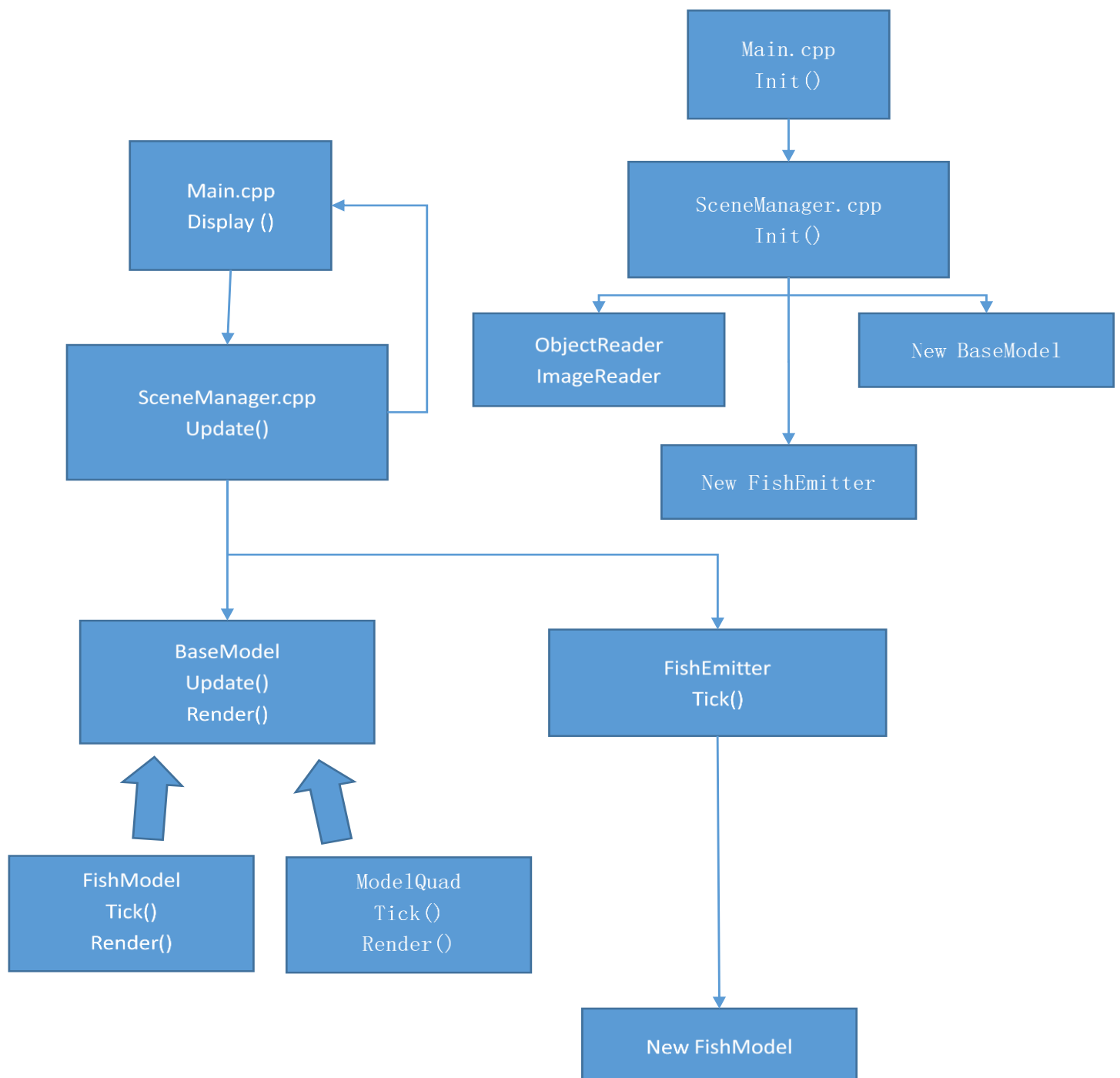
（四）实验分工

1. 总体实现目标思路（全员参与）
2. 读入 obj 和 image 文件和 basemodel 的实现（李静雯、李世然）
3. quadmodel 的实现（江金昱）
4. fishmodel 和 fishemitter 的实现（江金昱、赖少玄）

5. scenemanager 的实现 (黄镇)

6. waveplantsmodel (江金昱)

(五) 函数框架



二、实现方法

(一) ObjectReader

1. 功能：读取、解析模型文件（obj），并将信息保存下来。

用途：由 SceneManager 初始化加载所需要的 obj 文件，供 BaseModel 们共享访问。

2. 实现思路：

(1) 构造“三维数据结构体”、“一个面数据结构体”；

(2) 构造“模型读取类”，使用以上两个结构体，读取模型获得数据。

3. 实现代码：

(1) “三维数据结构体” `struct Float3`

①用 x\y\z 构成三维浮点数向量；

②可计算向量长度；

③实现三维向量的减法，重载减号。

```

/*#####
## 结构体: Float3
## 结构体描述: 三维数据, 可用来表示位置, 向量
#####*/

struct Float3
{
    float x, y, z;

    Float3()
        : x(0), y(0), z(0)
    { }

    Float3(float x0, float y0, float z0)
        : x(x0), y(y0), z(z0)
    { }

    /*得到向量长度*/
    float Length()
    {
        return sqrt(x*x + y * y + z * z);
    }

    /*重载 - 运算符*/
    Float3& operator-(const Float3& right)

```

```

    {
        x = this->x - right.x;
        y = this->y - right.y;
        z = this->z - right.z;

        return *this;
    }
};

```

(2) “一个面数据结构体” **struct Face**

①用齐次坐标四维向量表示位置；

②用齐次坐标四维向量表示法线及纹理坐标。

```

/*#####
## 结构体: Face
## 结构体描述: 存放一个面内的数据
#####*/
struct Face
{
    /*位置*/
    int position[3];
    /*法线*/
    int normal[3];
    /*纹理坐标*/
    int Texcoord[3];
};

```

(3) “模型读取类” **class ObjReader**

①构造函数；

②析构函数；

③判断是否加载成功的函数；

④实现模型加载函数：

(i) 将路径转化成绝对路径；

(ii) 打开文件，并检查错误，若打开失败则输出提示语并返回 false；

(iii) 若能成功打开，读取一行的数据，获取 v 开头的的数据，对这行数据根据第二个字符

进行分类，分为“纹理”数据、“法线”数据、“点的位置”数据；获取 f 开头的数据，读取面数据，包括每个点的“纹理”数据、“法线”数据、“点的位置”数据，然后分别将纹理、法线、顶点和面的数据和索引信息至对应变量中。

```

/*#####
## 类: ObjReader
## 类的描述: 模型读取类
#####*/
class ObjReader
{
public:
    /*filename:模型存放路径*/
    ObjReader(const char* filename);
    ~ObjReader();

    /*是否已经加载到内存*/
    bool IsLoad();

    /*顶点位置数据*/
    vector<Float3> vertexArray;
    /*纹理数据*/
    vector<Float3> texcoordArray;
    /*法线数据*/
    vector<Float3> normalArray;
    /*面的数据*/
    vector<Face> faceArray;

private:
    bool ObjLoadModel(const char* filename);
    bool isLoad;
};

```

//类的实现

```

ObjReader::ObjReader(const char* filename)
{
    isLoad = false;           // 初始化为未加载
    ObjLoadModel(filename); // 调用加载模型函数
}

ObjReader::~~ObjReader()
{ }

```

```
bool ObjReader::IsLoad()    // 判断模型是否加载成功
{
    return isLoad;
}

//加载模型文件
bool ObjReader::ObjLoadModel(const char* filename)
{
    // 转化成绝对路径
    string dir = QDir::currentPath().toStdString() + "/" + filename;
    const char* s = dir.c_str();

    // 打开文件, 并检查错误
    ifstream file;
    file.open(s, ios_base::in);

    if (!file.is_open())
    {
        cout << "open file failed." << endl; // 模型加载失败提示语
        return false;
    }

    string temp;          // 接受无关信息
    char szoneLine[256]; // 读取一行的数据

    // 循环读取
    while (file)
    {
        file.getline(szoneLine, 256);

        if (strlen(szoneLine) > 0)           // 该行不为空
        {
            if (szoneLine[0] == 'v')         // 获取 v 开头的数据
            {
                stringstream ssOneLine(szoneLine); // 数据存储到 stringstream 流
                if (szoneLine[1] == 't')         // 纹理信息
                {
                    ssOneLine >> temp;           //接收标识符 vt
                    Float3 tempTexcoord;         // 纹理坐标向量
                    // 存储纹理坐标
                    ssOneLine >> tempTexcoord.x >> tempTexcoord.y >> tempTexcoord.z;
                    tempTexcoord.y = 1 - tempTexcoord.y; // 解决纹理上下翻转问题
                }
            }
        }
    }
}
```

```

        texcoordArray.push_back(tempTexcoord);    // 将纹理坐标存入容器
    }
else if (szoneLine[1] == 'n')                    // 记录的法线信息
{
    ssOneLine >> temp;                            //接收标识符 vn

    Float3 tempNormal;                            //法线坐标向量
    // 存储法线坐标
    ssOneLine >> tempNormal.x >> tempNormal.y >> tempNormal.z;
    normalArray.push_back(tempNormal);            //将法线信息存入容器
}
else if (szoneLine[1] == ' ')                    // 点的位置信息
{
    ssOneLine >> temp;                            //接收标识符 v
    Float3 tempLocation;                          //位置坐标
    //存储位置坐标
    ssOneLine >> tempLocation.x >> tempLocation.y >> tempLocation.z;
    vertexArray.push_back(tempLocation);          // 将位置坐标存入容器
}
}

else if (szoneLine[0] == 'f')                    //记录面信息
{
    stringstream ssOneLine(szoneLine); //读取一行数据
    ssOneLine >> temp;                      //接收标识符 f

    // 一行的数据例子:f 1/1/1 2/2/2 3/3/3
    // 分别表示:位置索引/纹理索引/法线索引    三个点构成一个面
    // 接收类似于 1/1/1 的一组索引
    string vertexStr;
    Face tempFace;                            // 面矩阵
    for (int i = 0; i < 3; ++i) // 每个面 3 个点
    {
        ssOneLine >> vertexStr; // 从流中读取点的索引信息
        // 找到第一个'/'的位置,即找到点的位置信息
        size_t pos = vertexStr.find_first_of('/');
        string locIndexStr = vertexStr.substr(0, pos);    // 顶点位置信息
        // 找到第二个'/',即找到点的纹理坐标信息
        size_t pos2 = vertexStr.find_first_of('/', pos + 1);
        // 将索引信息从 string 转换为 int,顶点位置索引信息赋值
        tempFace.position[i] = atoi(locIndexStr.c_str());
        if (pos2 - pos == 1)                    // 如果是双斜线的情况即'//',例如 1//1
        {
            string norIndexStr = vertexStr.substr(pos2 + 1,
                vertexStr.length() - pos2 - 1); // 获取法线信息
        }
    }
}

```



```

        tempFace.normal[i] = atoi(norIndexStr.c_str());
        tempFace.Texcoord[i] = 0; // 标志 0: 在后续处理分配给 0 号 (0, 0, 0)
    }

    else // 单斜线的情况, 例如 1/1/1
    {
        string texIndexSrt = vertexStr.substr(pos + 1, pos2 - 1 - pos);
        // 点的纹理信息
        string norIndexSrt = vertexStr.substr(pos2 + 1,
            vertexStr.length() - 1 - pos2); // 点的法线信息

        tempFace.normal[i] = atoi(norIndexSrt.c_str()); // 存储纹理索引信息
        tempFace.Texcoord[i] = atoi(texIndexSrt.c_str()); // 存储法线索引信息
    }

    faceArray.push_back(tempFace);
}

//防止没有纹理坐标的情况发生, 给予默认值
if (texcoordArray.size() == 0)
    texcoordArray.push_back(Float3(0, 0, 0));

cout << "load model: " << filename << endl;

file.close();
isLoad = true; // 模型加载成功
return true;
}

```

（二）ImageReader

1. 功能：读取图片文件（PNG），并绑定到 Opengl。

用途：由 SceneManager 初始化加载所需要的 image 文件，供 BaseModel 们共享访问。

2. 实现思路：

——构造“图片读取类”，读取图片获得图片数据并绑定到 Opengl 的图片 ID。

3. 实现代码：

①使用 QImage 类型存放图片数据、使用 GLuint 存放绑定到 Opengl 的图片 ID；

②读取图片；

③将图片加载到内存。

——“图片读取类” `class ImageReader`

①构造函数；

②析构函数；

③判断是否加载成功的函数；

④实现图片加载函数：

（i）将路径转化成绝对路径；

（ii）打开文件，并检查错误，若打开失败则输出提示语并返回 false；

（iii）若能成功打开，设置每行图像数据的字节数，对每行进行渲染；先生成纹理，再绑定纹理，再将纹理像素映射为像素，最后制定纹理函数，并根据指定参数生成一个 2D 纹理，并获取图片 ID。

```
/*#####
## 类: ImageReader
## 类描述: 图片读取工具
#####*/
class ImageReader
{
public:
    ImageReader();
    ~ImageReader();
    /*filename:图片存放路径*/
    ImageReader(const char* filename);

    /*是否已经加载到内存*/
    bool IsLoad();
```

```
/*图片数据*/
QImage image;
/*绑定到 Opengl 的图片 ID*/
GLuint texID;

private:

/*filename:图片存放路径*/
bool LoadTexture(const char* filename); //加载图片到内存

/*是否已经加载到内存的标志*/
bool isLoad;
};
```

//类的实现

```
ImageReader::ImageReader(const char* filename)
{
    isLoad = false;

    LoadTexture(filename);
}

ImageReader::ImageReader()
{
    isLoad = false;
}

ImageReader::~ImageReader()
{ }

bool ImageReader::IsLoad()
{
    return isLoad;
}

bool ImageReader::LoadTexture(const char* filename)
{
    // 转化成绝对路径
    string dir = QDir::currentPath().toStdString() + "/" + filename;
    const char* s = dir.c_str();

    if (!image.load(s, "PNG"))
    {

```

```
        cout << "image : open file failed" << endl;
        return false;
    }

    // 加载图片数据和图片 ID
    // 即一行的图像数据字节数必须是 4 的整数倍，即读取数据时，读取 4 个字节用来渲染一行
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    // 生成纹理
    glGenTextures(1, &texID);
    // 绑定纹理
    glBindTexture(GL_TEXTURE_2D, texID);
    // 如何把纹理像素映射成像素
    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_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    // 指定纹理函数
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

    image = image.convertToFormat(QImage::Format_RGBA8888);
    // 根据指定的参数，生成一个 2D 纹理 (Texture)
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, image.width(), image.height(), 0, GL_RGBA,
GL_UNSIGNED_BYTE, image.bits());

    cout << "load image: " << filename << endl;

    isLoad = true;
    return true;
}
```

(三) ModelQuad(海底地面模型)

1. 用途：继承 BaseModel，重写了虚函数部分功能，能够被 SceneManager 管理。

2. 实现思路：

——继承 BaseModel 类，构造“海底地面模型”，画一个四方面片，用于模拟地面。

3. 实现代码：

①构造函数；

②析构函数；

③重写 BaseModel 基类的渲染函数 Render()，渲染面片：

(i) 加载模型 obj 和图片 image，若任意一个没有则停止渲染；

(ii) 构造位置矩阵和旋转、缩放矩阵；

(iii) 改变面片的位置；

(iv) 2. 海底的渲染：读入海底模型和图片后，初始化模型，然后使用旋转矩阵和缩放矩阵改变模型位置及大小，然后绑定图片，为每个点设置纹理、法线、位置等信息。

④为了统一，添加实现每帧更新的空函数。

```

/*#####
## 类: ModelQuad
## 父类: BaseModel
## 类描述: 继承模型类，画一个四方面片，用于模拟地面
#####*/
class ModelQuad : public BaseModel
{
public:
    /*
    position:模型初始化位置
    rotation:模型初始化旋转
    scale:模型缩放
    objdectId:模型 ID 号
    imgId:图片 ID 号
    enableBlend: 是否开启颜色混合（用于透明渲染
    enableTwoSide: 是否开启双面渲染（用于片状物体
    */
    ModelQuad(Float3 position, Float3 rotation, Float3 scale, int objectId, int imgId,
bool enableBlend = false, bool enableTwoSide = false);
    ~ModelQuad();

    /*重写基类渲染函数，用于渲染面片*/
    virtual void Render(ObjReader& obj, ImageReader& image);

```

```
protected:
    /*每帧更新函数*/
    virtual void Tick();
};
```

//类的实现

```
ModelQuad::ModelQuad(Float3 position, Float3 rotation, Float3 scale, int objectId, int
imgId, bool enableBlend, bool enableTwoSide)
    :BaseModel(position, rotation, scale, objectId, imgId, enableBlend, enableTwoSide)
{ }

ModelQuad::~ModelQuad()
{ }

void ModelQuad::Tick()
{ }

void ModelQuad::Render(ObjReader& obj, ImageReader& image)
{
    //如果没有指定模型或者图片即停止渲染
    if (!obj.IsLoad() || !image.IsLoad())
        return;

    //位置矩阵
    GLfloat T[][4] = { {1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {mPosition.x, mPosition.y,
mPosition.z, 1} };
    //绕 Z 轴旋转矩阵
    GLfloat RZ[][4] = { {cos(mRotation.z * rotationTrans), sin(mRotation.z * rotationTrans),
0, 0}, {-sin(mRotation.z * rotationTrans), cos(mRotation.z * rotationTrans), 0, 0}, {0, 0, 1,
0}, {0, 0, 0, 1} };
    //绕 X 轴旋转矩阵
    GLfloat RX[][4] = { {1, 0, 0, 0}, {0, cos(mRotation.x * rotationTrans), sin(mRotation.x *
rotationTrans), 0}, {0, -sin(mRotation.x * rotationTrans), cos(mRotation.x *
rotationTrans), 0}, {0, 0, 0, 1} };
    //绕 Y 轴旋转矩阵
    GLfloat RY[][4] = { {cos(mRotation.y * rotationTrans), 0, -sin(mRotation.y *
rotationTrans), 0}, {0, 1, 0, 0}, {sin(mRotation.y * rotationTrans), 0, cos(mRotation.y *
rotationTrans), 0}, {0, 0, 0, 1} };
    //缩放矩阵
    GLfloat S[][4] = { {mScale.x, 0, 0, 0}, {0, mScale.y, 0, 0}, {0, 0, mScale.z, 0}, {0, 0,
0, 1} };
```

```
// 改变位置
glPushMatrix();
glMultMatrixf(*T);
glMultMatrixf(*RZ);
glMultMatrixf(*RX);
glMultMatrixf(*RY);
glMultMatrixf(*S);

//绑定图片
glBindTexture(GL_TEXTURE_2D, image.texID);
for (auto face : obj.faceArray)
{
    //为每个点设置法线, 纹理, 位置信息
    glBegin(GL_TRIANGLES);

    for (int i = 0; i < 3; i++)
    {
        GLfloat normal[3] = { obj.normalArray[face.normal[i] - 1].x,
obj.normalArray[face.normal[i] - 1].y, obj.normalArray[face.normal[i] - 1].z };
        glNormal3fv(normal);

        //UV 扩展 10 倍
        GLfloat texCoord[3] = { obj.texcoordArray[face.Texcoord[i] - 1].x * 5,
obj.texcoordArray[face.Texcoord[i] - 1].y * 5, obj.texcoordArray[face.Texcoord[i] - 1].z *
5};

        glTexCoord3fv(texCoord);

        GLfloat position[3] = { obj.vertexArray[face.position[i] - 1].x,
obj.vertexArray[face.position[i] - 1].y, obj.vertexArray[face.position[i] - 1].z };
        glVertex3fv(position);
    }

    glEnd();
}

glPopMatrix();
}
```


（四）BaseModel

1. 模块功能:

BaseModel 类是通用模型的封装，具有逻辑更新接口（Tick()）和渲染绘制接口（Render(Param, Param)），并提取出位置、旋转、缩放、是否透明、是否双面渲染这些渲染状态信息，是能被 SenenManager 管理的基础单位。

2. 模块组成:

类中一共有 5 个成员函数，除去析构函数，其他成员函数各有自己的功能：构造函数初始化模型的信息，Update() 提供逻辑更新的接口，虚函数 tick() 提供模型更新的状态信息，Render() 提供渲染绘制接口和渲染状态信息。

3. 各函数的具体实现如下:

➤ 构造函数

```
BaseModel::BaseModel(Float3 position, Float3 rotation, Float3 scale, int
objectId, int imgId, bool enableBlend, bool enableTwoSide)
{
    //初始化位置，旋转，大小信息
    mPosition = position;
    mRotation = rotation;
    mScale = scale;

    //指定模型，图片 ID
    objId = objectId;
    imageId = imgId;

    //初始化是否混合
    isEnabledBlend = enableBlend;
    //初始化是否开启双面
    isEnabledTwoSide = enableTwoSide;
}
```

【说明】 Position 决定模型在场景中的位置，rotation 决定模型在场景中放置的角度，scale 决定模型的大小；根据 objID 和 imageID 加载模型和图片；isEnabledBlend 决定是否进行 Blending；isEnabledTwoSide 决定是否开启双面渲染。

➤ Update () 逻辑更新接口

```
void BaseModel::Update()
{
    //调用虚函数
    Tick();
}
```

【说明】Tick() 有各子类具体实现更新的信息；

➤ Render() 渲染绘制接口

```
void BaseModel::Render(ObjReader& obj, ImageReader& image)
{
    //如果没有指定模型或者图片即停止渲染
    if (!obj.IsLoad() || !image.IsLoad())
        return;

    //位置矩阵
    GLfloat T[][4] = { {1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1, 0},
    {mPosition.x, mPosition.y, mPosition.z, 1} };

    //绕 Z 轴旋转矩阵
    GLfloat RZ[][4] = { {cos(mRotation.z * rotationTrans),
sin(mRotation.z * rotationTrans), 0, 0}, {-sin(mRotation.z * rotationTrans),
cos(mRotation.z * rotationTrans), 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1} };

    //绕 X 轴旋转矩阵
    GLfloat RX[][4] = { {1, 0, 0, 0}, {0, cos(mRotation.x *
rotationTrans), sin(mRotation.x * rotationTrans), 0}, {0, -sin(mRotation.x *
rotationTrans), cos(mRotation.x * rotationTrans), 0}, {0, 0, 0, 1} };

    //绕 Y 轴旋转矩阵
    GLfloat RY[][4] = { {cos(mRotation.y * rotationTrans), 0, -
sin(mRotation.y * rotationTrans), 0}, {0, 1, 0, 0}, {sin(mRotation.y *
rotationTrans), 0, cos(mRotation.y * rotationTrans), 0}, {0, 0, 0, 1} };

    //缩放矩阵
    GLfloat S[][4] = { {mScale.x, 0, 0, 0}, {0, mScale.y, 0, 0}, {0, 0,
mScale.z, 0}, {0, 0, 0, 1} };

    // 改变位置
    glPushMatrix();
    glMultMatrixf(*T);
    glMultMatrixf(*RZ);
    glMultMatrixf(*RX);
    glMultMatrixf(*RY);
    glMultMatrixf(*S);
```

```
if (isEnabledBlend) //绘制透明物体时
{
    glEnable(GL_BLEND);
    glDepthMask(false); //设置深度缓冲区只读
}
else //绘制不透明物体
{
    glDepthMask(true); //设置深度缓冲区可写
    glDisable(GL_BLEND);
}

//是否开启双面渲染
if (isEnabledTwoSide)
    glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
else
    glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_FALSE);

//绑定图片
glBindTexture(GL_TEXTURE_2D, image.texID);
for (auto face : obj.faceArray)
{
    //为每个点设置法线, 纹理, 位置信息
    glBegin(GL_TRIANGLES);

    for (int i = 0; i < 3; i++)
    {
        GLfloat normal[3] =
        { obj.normalArray[face.normal[i] - 1].x, obj.normalArray[face.normal[i] - 1].y, obj.normalArray[face.normal[i] - 1].z };
        glNormal3fv(normal);

        GLfloat texCoord[3] =
        { obj.texcoordArray[face.Texcoord[i] - 1].x,
        obj.texcoordArray[face.Texcoord[i] - 1].y,
        obj.texcoordArray[face.Texcoord[i] - 1].z };
        glTexCoord3fv(texCoord);

        GLfloat position[3] =
        { obj.vertexArray[face.position[i] - 1].x, obj.vertexArray[face.position[i] - 1].y, obj.vertexArray[face.position[i] - 1].z };
        glVertex3fv(position);
    }
}
```

```
        glEnd();  
    }  
  
    glPopMatrix();  
}
```

【说明】

- ①首先判断模型和图片有无加载成功，如果没有则停止渲染。然后在渲染时，首先改变模型的位置，初始化旋转和缩放矩阵，通过矩阵乘法实现分别绕 XYZ 轴旋转调整角度并改变大小；
- ②然后在绘制透明物体时，开启颜色混合，开启双面渲染时，会使背对摄影机的那一面也得到渲染，会使透明效果更加逼真。
- ③接下来，绑定图片，为模型的每个点设置法线，纹理和位置信息。OpenGL 中是以面为单位进行渲染的，所以对模型中的每个面进行循环，以三角形作为图元，设置三角形每个顶点的属性。

（五）SceneManager（场景管理器）

1. 初始化时统一加载场景内模型，设置静态模型的空间位置等信息，设置全局光、雾效、天空球的信息
2. 每次渲染时先更新模型的逻辑接口、再进行渲染更新
3. 管理透明物体渲染排序：映射物体在相机 XZ 平面的位置，得到其与相机的距离，将距离作为 Key 存储到 Map，利用 Map 的自动排序功能按顺序渲染透明物体。
4. 代码实现：

【说明】SceneManager.h 和 SceneManager.cpp 里面主要定义了一个名为 SceneManager 的类，意为场景管理器，用来管理场景。这个类的作用主要是在初始化时分别使用 ObjReader 和 ImageReader 将模型和图片载入内存，初始化天空、光照、雾以及鱼群生成器等和场景有关的需要初始化的东西，并且定义了一个 Update 函数，用于渲染下一帧。

```
/*场景管理器：统管场景渲染*/
class SceneManager
{
public:
    SceneManager();
    ~SceneManager();

    /*初始化场景设置*/
    void Init();

    /*每帧调用，先更新场景物体的位置，再进行渲染*/
    void Update();

    /*存储透明渲染队列*/
    map<int, BaseModel*> translucentModel;

private:
    /*初始化天空*/
    void InitSky();
    /*初始化灯光*/
    void InitLight();
    /*初始化雾*/
    void InitFog();
    /*渲染天空*/
    void SetSky();
    /*初始化场景模型*/
    void InitModel();

    ImageReader sky;

    /*鱼群生成器*/
    FishEmitter* fishEmitter1;
    FishEmitter* fishEmitter2;
    FishEmitter* fishEmitter3;

    /*存储模型文件*/
    vector<ObjReader> objList;
    /*存储图片文件*/
    vector<ImageReader> imageList;

    /*存储实体模型*/
    vector<BaseModel*> modelList;
};
```

【说明】

①在 SceneManager 里面，我们首先定义了一个构造函数 SceneManager() 和一个析构函数 ~SceneManager()，其中构造函数里面啥也没有，因为初始化的工作都已经放在 Init() 里面了，析构函数主要用于释放内存，与本课程关系不大，故略过。

②SceneManager 里面最重要的函数就是 Init 和 Update。】Init 函数的主要作用，就是首先调用 InitModel 来初始化模型，然后调用 InitSky 来初始化天空，再调用 InitLight 来初始化光照，再调用 InitFog 来初始化雾，最后，初始化三个鱼群生成器 fishEmitter1、fishEmitter2 和 fishEmitter3。

```
void SceneManager::Init()
{
    InitModel(); //初始化模型

    InitSky(); //初始化天空
    InitLight(); //初始化光照
    InitFog(); //初始化雾

    //初始化鱼群生成器
    fishEmitter1 = new FishEmitter(10, 100, Float3(-10, 100, 50),
Float3(0, 25, 0), 12, 12, 40);
    fishEmitter2 = new FishEmitter(10, 100, Float3(-10, 100, -100),
Float3(0, 0, 0), 12, 13, -30);
    fishEmitter3 = new FishEmitter(10, 100, Float3(-10, 100, -100),
Float3(0, 0, 0), 12, 14, -60);
}
```

【说明】接下来将依次解释 InitModel、InitSky、InitLight 和 InitFog。

①首先是 InitModel——

(i) InitModel 函数的作用，就是首先使用前面写的 ObjReader 函数，从 objMesh 文件夹里面读取 13 个 obj 模型。然后使用前面写的 ImageReader 函数，从 Textures 文件夹里面读取 15 张图片。我们把这 13 个 obj 模型依次插入一个名为 objList 的 vector 里面，它们的下标从 0 到 12（分别注释在那 13 行的后面），同理，我们把这 15 张图片依次插入一个名为 imageList 的 vector 里面，它们的下标从 0 到 14（分别注释在那 15 行的后面）。

(ii) InitModel 在读完模型和图片之后, 就开始初始化实体物体和透明物体了。实体物体, 因为不需要排序, 所以使用 vector 即可。而透明物体需要排序, 所以为了方便, 我们用了 map 来存储透明物体。

```
void SceneManager::InitModel()
{
    //加载模型进内存
    objList.push_back(ObjectReader("objMesh/SM_URockB.obj")); //0
    objList.push_back(ObjectReader("objMesh/SM_URockC.obj")); //1
    此处省略 10 行
    objList.push_back(ObjectReader("objMesh/SM_land.obj")); //12

    //加载图片进内存
    imageList.push_back(ImageReader("Textures/T_URockB_BC.png")); //0
    imageList.push_back(ImageReader("Textures/T_URockC_BC.png")); //1
    此处省略 12 行
    imageList.push_back(ImageReader("Textures/T_FishC_BC.png"));
    //14

    //实体物体, 不需要排序
    modelList.push_back(new BaseModel(Float3(340, 0, -240), Float3(0, 0, 0), Float3(1.0, 1.0, 1.0), 0, 0));
    modelList.push_back(new BaseModel(Float3(420, 0, -110), Float3(0, 0, 0), Float3(1.0, 1.0, 1.0), 1, 1));
    此处省略 7 行
    modelList.push_back(new ModelQuad(Float3(0, 0, 0), Float3(0, 0, 0), Float3(1.0, 1.0, 1.0), 12, 9));

    //透明物体
    translucentModel.insert(pair<int, BaseModel*>((Float3(420, cameraPos.y, 380) - cameraPos).Length(), new BaseModel(Float3(420, -30, 380), Float3(0, 60, 0), Float3(1.0, 1.0, 1.0), 7, 7, true, true)));
    translucentModel.insert(pair<int, BaseModel*>((Float3(480, cameraPos.y, 330) - cameraPos).Length(), new BaseModel(Float3(480, -30, 330), Float3(0, 0, 0), Float3(1.0, 1.0, 1.0), 7, 7, true, true)));
    此处省略 6 行
    translucentModel.insert(pair<int, BaseModel*>((Float3(1100, cameraPos.y, 400) - cameraPos).Length(), new BaseModel(Float3(1100, -30, 350), Float3(0, 0, 0), Float3(1.0, 1.0, 1.0), 6, 7, true, true)));
}
```

【说明】实现物体透明的方法和原理（用的是需要排序的传统方法）

①我们的场景看似没有半透明物体，但其实是有的。我们的那些像纸片一样的鱼，其实是一张张矩形制片。如下图所示，这是一张矩形图片，其中鱼的部分是不透明的，而周围白色的部分是完全透明的。这样，我们渲染出来之后，看起来就不是画有鱼的图案的矩形白纸，而是形状像鱼一样的纸片了。



②除了鱼以外，还有水草等物体其实都是要作为半透明物体来渲染的。这里来说一下我们利用颜色混合的方法来实现半透明物体的方法和原理。

（i）首先，对于不透明的物体，我们是可以按照任意顺序渲染的，因为我们有深度缓冲区，存储每个像素距离人眼的距离。如果当前要渲染的物体的距离更近，则直接覆盖原来的值，表示现在的物体把原来的物体挡住了，否则，就不渲染当前的像素，表示现在的物体被原来的物体挡住了。这样，对于不透明物体，我们可以按照任意顺序渲染。

（ii）接下来就是透明物体了，对于透明物体，我们用颜色混合的方法进行渲染。我们先渲染远处的透明物体，再渲染近处的透明物体。我们在渲染近处的透明物体的时候，不是把颜色直接覆盖上去，而是根据之前的颜色、现在的颜色，以及透明度，来计算出实际看到的颜色，然后把这个像素的颜色改成相应的颜色。这样就实现了物体的透明效果。

（iii）那为什么我们要按照距离从远到近的顺序来渲染透明物体呢？原因很简单，因为我们在渲染的时候，对于深度缓冲区的处理，显卡并没有区分不透明物体和透明物体，如果我们先渲染了近的透明物体，那再渲染远的物体的时候，由于深度缓冲区存的距离比要渲染的物体的距离更近，所以远的物体就被直接忽略掉了，但实际上，由于前面的物体是透明的，后面的物体是不应该直接忽略掉的，而如果直接忽略掉了，就产生了错误的结果。

（iv）综上所述，按照我们用的传统方法，我们在渲染的时候，应该先渲染不透明物体，再渲染透明物体。其中，不透明物体可以按任意顺序渲染，透明物体则要按从远到近的顺序渲染，才能得到正确的结果。

【说明】最后再来说一下用于渲染下一帧的 Update 的代码，在 Update 的时候，我们先设置天空，然后，先更新 modellist 里面的不透明物体，再更新 BaseModel 里面的透明物体，然后更新一下鱼群生成器的位置。接着渲染的时候，我们先渲染不透明物体，再按照距离从远到近的顺序来渲染透明物体，这样我们就可以把我们的场景渲染出来了。

```
void SceneManager::Update()
{
    SetSky(); //设置天空

    for (auto cur : modellist)
    {
        cur->Update(); //更新不透明物体
    }

    map<int, BaseModel*> temp = translucentModel; //防止 update() 内对列表的更改
    for (auto cur : temp)
    {
        cur.second->Update(); //更新透明物体
    }

    //鱼群更新位置
    fishEmitter1->Tick();
    fishEmitter2->Tick();
    fishEmitter3->Tick();

    //先渲染实体
    for (auto cur : modellist)
    {
        cur->Render(objList[cur->objId], imageList[cur->imageId]);
    }

    //再渲染透明物体，利用 Map 的元素自动排序功能，按深度大小绘制透明物体
    for (map<int, BaseModel*>::reverse_iterator cur =
translucentModel.rbegin(); cur != translucentModel.rend(); cur++)
    {
        (*cur).second->Render(objList[(*cur).second->objId],
imageList[(*cur).second->imageId]);
    }
}
```

（六）FishModel（鱼模型）

• 关键部分说明：

管理鱼的运动，被 FishEmitter 创建，由 SceneManager 管理渲染以及逻辑更新

1. 初始化时在生成点进行位置随机偏移与速度随机，增加鱼群位置差异化

2. 鱼群流体数据：

①将时间 T 作为 $\sin(T)$ 的参数，得到的结果作为鱼群随时间变化的朝向因子 dir；

②dir*waveRange(表示最大旋转角度) 得到鱼群相对于出生时朝向的偏差 angle；

③将偏差转化成矢量方向，与速度相乘，得到当前帧的位移矢量 s；

④将 s 与上一次的位置相加，得到当前帧鱼的新位置；

⑤有了新的位置和方向就可以在场景内渲染出这条鱼。

• 代码说明：

BaseModel 类的子类，用一个四方形面片加纹理贴图来模拟一条鱼。

● 主要使用了继承自 BaseModel 的如下属性：

```
/*模型位置*/
Float3 mPosition;
/*模型旋转*/
Float3 mRotation;
/*模型缩放*/
Float3 mScale;
/*图片 Id 号*/
int imageId;

/*是否开启颜色混合*/
bool isEnableBlend;
/*是否开启双面渲染*/
bool isEnableTwoSide;
```

【说明】子类特化的属性如下，内含两个角度，一个是模型旋转（mRotation，三个方向的），这是鱼的初始角度，以下称作模型角度；另一个是 angle，代表鱼的角度偏移，以下称作偏移角度：

```
/*旋转角度，用于变换*/
float angle;
/*游动速度*/
float speed;
```

```

/*时间计时器*/
int time;
/*游动幅度范围*/
float waveRange;
/*在 XZ 平面上与相机的距离*/
int disFromCamera;

```

● 构造函数:

```

FishModel(Float3 position, Float3 rotation, Float3 scale, int objectId, int
imgId, bool enableBlend = false, bool enableTwoSide = false);

```

①先在初始化列表中调用基类的构造函数:

```

BaseModel(position, rotation, scale, objectId, imgId, enableBlend,
enableTwoSide)

```

②在函数体部分分别对位置，大小变量加上一个适当范围内的随机值，并随机初始化速度，以产生不同鱼之间的差异。同时分别给计时器，角度和幅度赋固定的初始值。

```

//随机化初始坐标
mPosition.y = mPosition.y + rand() % 300;
mPosition.z = mPosition.z + rand() % 300;

//随机化速度
speed = rand() % 5 / 10.0 + 2;

//初始化数据
time = 0;
angle = 0;
this->waveRange = 60;

//随机化大小
float s = (float)(rand() % 5) / 5 + 1.0;
mScale = Float3(s, s, s);

```

③计算出鱼与摄像机在 XZ 平面上的距离，并将这个距离到鱼的关系存储到场景的透明渲染队列（map）中：

```
//计算在 XZ 平面上与摄像机的距离
Float3 p1 = mPosition;
p1.y = cameraPos.y;
disFromCamera = (p1 - cameraPos).Length();

//根据与摄像机的距离加入到透明渲染队列，如果队列上已经有相同距离的物体存在，
//则将距离+1
while (scene.translucentModel.count(disFromCamera))
{
    disFromCamera++;
}
scene.translucentModel.insert(pair<int, BaseModel*>(disFromCamera, this));
```

● 析构函数：将自己从场景的透明渲染队列中移除

```
FishModel::~~FishModel()
{
    //释放场景内的自己
    scene.translucentModel.erase(disFromCamera);
}
```

● Init 函数：设定游动幅度的值

```
void FishModel::Init(float waveRange)
{
    //设置游动偏移
    this->waveRange = waveRange;
}
```

● render：渲染函数，根据当前的位置，角度，大小变量来渲染贴图面片（obj 是基类的 render 需要使用的参数，这里没有用到）

```
void FishModel::Render(ObjReader& obj, ImageReader& image)
{
    //渲染，如果图片没有加载就停止
    if (!image.IsLoad())
        return;

    // 改变位置
    glPushMatrix();
    glTranslatef(mPosition.x, mPosition.y, mPosition.z);
    glRotatef(mRotation.y + angle, 0.0, 1.0, 0.0);
```

```
glScalef(mScale.x, mScale.y, mScale.z);

//一个面的顶点
GLfloat vertices[][3] =
{
    {-10.0f, 10.0f, 0.0f}, {-10.0f, -10.0f, 0.0f},
    { 10.0f, 10.0f, 0.0f}, { 10.0f, -10.0f, 0.0f}
};

//开启混合，设定颜色混合模式
glEnable(GL_BLEND);
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
glDepthMask(false);          //设置深度缓冲区只读

//绑定图片
glBindTexture(GL_TEXTURE_2D, image.texID);

//绘制面
GLfloat n[] = { 0, 0, 1 };
glNormal3fv(n);

glBegin(GL_QUADS);
glTexCoord2f(0, 0);
glVertex3fv(vertices[0]);

glTexCoord2f(0, 1);
glVertex3fv(vertices[1]);

glTexCoord2f(1, 1);
glVertex3fv(vertices[3]);

glTexCoord2f(1, 0);
glVertex3fv(vertices[2]);
glEnd();

glPopMatrix();
}
```

- **Tick:** 主要用于每帧更新位置，角度等参数

更新计时器，根据计时器和得出当前偏移角度，根据模型角度和偏移角度得出游动方向，根据游动方向和速度更新鱼的位置：

```
//计时器+1
time += 1;

//计算当前鱼群角度转变
angle = sin(time/120.0f) * waveRange;
float angle2 = (angle + mRotation.y) * 3.14 / 180.0;

//游动方向
Float3 dir(cos(angle2), 0, -sin(angle2));

//更新位置
mPosition = Float3(mPosition.x + dir.x * speed, mPosition.y + dir.y * speed,
mPosition.z + dir.z * speed);
```

由于位置更新鱼与摄像机的距离也在改变，所以需要更新场景的透明渲染队列：

```
//刷新渲染列表
scene.translucentModel.erase(disFromCamera);           //移除之前的

Float3 p1 = mPosition;
p1.y = cameraPos.y;
disFromCamera = (p1 - cameraPos).Length(); //更新深度值
while (scene.translucentModel.count(disFromCamera)) //判断是否存在相同深度
{
    disFromCamera++;
}
scene.translucentModel.insert(pair<int, BaseModel*>(disFromCamera, this));
```

（七）FishEmitter（鱼群生成器）

鱼群生成器，用于在场景中管理一个鱼群，并可通过创建多个实例生成多个鱼群。

- 包含以下属性：所有这个鱼群生成器生成的鱼用相同的位置（mPosition），角度、（mRotation），模型 ID（实际没有用到）（objectId），图片 ID（imgID），偏角幅度（waveRange）作为构造函数的参数来初始化；计时器（time）与 freTimeCount 用来周期性地生成新的鱼，fishNumCount 为鱼群中鱼的最大数量；fishes 为存放每一条鱼指针的列表

```
/*生成器位置*/
Float3 mPosition;
/*生成器旋转角*/
Float3 mRotation;

/*时间计时器*/
int time;
/*鱼群中鱼的最大数量*/
int fishNumCount;
/*每隔这个时间生成一条鱼*/
int freTimeCount;
/*鱼的模型 ID 号*/
int objectId;
/*鱼模型的图片号*/
int imgId;
/*鱼游动的路线偏移角度*/
float waveRange;

/*存放鱼群列表*/
list<FishModel*> fishes;
```

- 构造函数：初始化所有成员变量（fishes 自动初始化为空列表）

```
FishEmitter::FishEmitter(int freTime, int fishNum, Float3 pos, Float3 rot,
int objectId, int imgId, float waveRange)
{
    //初始化信息
    freTimeCount = freTime;
    fishNumCount = fishNum;
    mPosition = pos;
    mRotation = rot;
    time = 0;

    this->objectId = objectId;
```

```

        this->imgId = imgId;
        this->waveRange = waveRange;
    }

```

- 析构函数：释放鱼群里的每一条鱼

```

FishEmitter::~FishEmitter()
{
    //释放鱼
    for (auto cur : fishes)
    {
        delete cur;
    }

    fishes.clear();
}

```

- Tick：每帧更新计数器，周期性地产生新的鱼加入鱼群，并在鱼数量超过最大后把最先加入的鱼剔除

```

void FishEmitter::Tick()
{
    //更新时间计数器
    time += 1;
    if (time > freTimeCount)
    {
        time = 0;

        //产生一条新鱼
        FishModel* cur = new FishModel(mPosition, mRotation,
Float3(1, 1, 1), objectId, imgId);
        cur->Init(waveRange);
        fishes.push_back(cur);
        //如果鱼的数量超出，释放队列最前方的鱼
        if (fishes.size() > fishNumCount)
        {
            delete fishes.front();
            fishes.pop_front();
        }
    }
}

```


（八）waveplantsmodel（实现水草的渲染以及飘动）

功能：载入并绘制水草，并且实现模拟水草在水中的飘动（模型的形变）。

水草的载入与渲染与 **obj** 的载入类似，这里重点描述如何实现水草的飘动

将水草每一个像素点的 z 轴坐标表示成关于 y 轴坐标以及时间的函数，这样就可以模拟水草在水中飘动的效果，公式如下：

$$floatZ = originalZ + \sin\left(\frac{time}{12} + y\right) * y$$

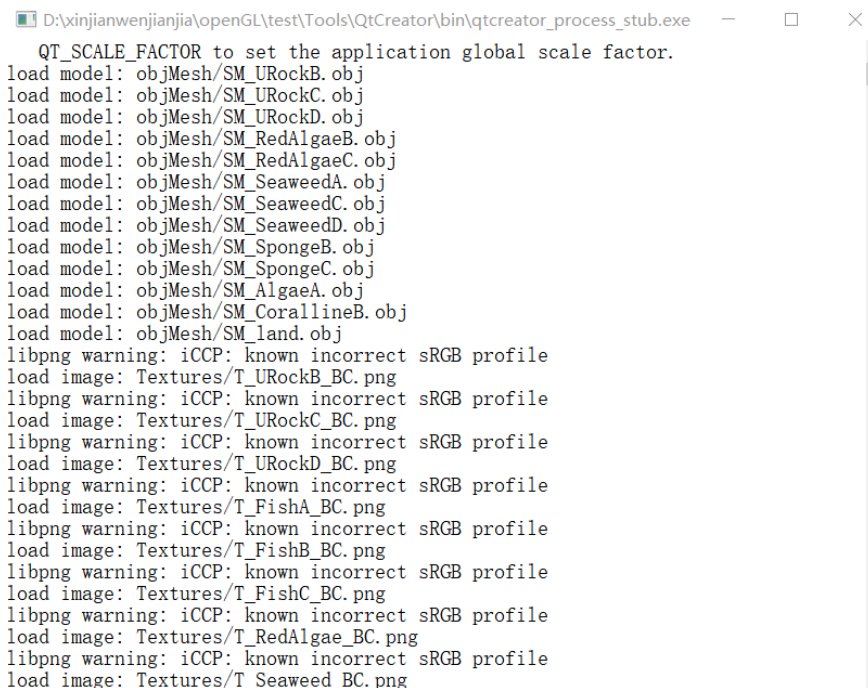
其中 *originalZ* 代表原来的 z 轴坐标，*floatZ* 表示模拟水草飘动的 Z 轴坐标。

代码如下：

```
float y = obj.vertexArray[face.position[i] - 1].y;
float z = sin(time/12.0f + y) * y / waveHeight
        +obj.vertexArray[face.position[i] - 1].z;
GLfloat position[3] = { obj.vertexArray[face.position[i] - 1].x, y, z};
glVertex3fv(position);
```

三、成品效果及分析

1. 加载模型及图片成功或失败的信息输出：

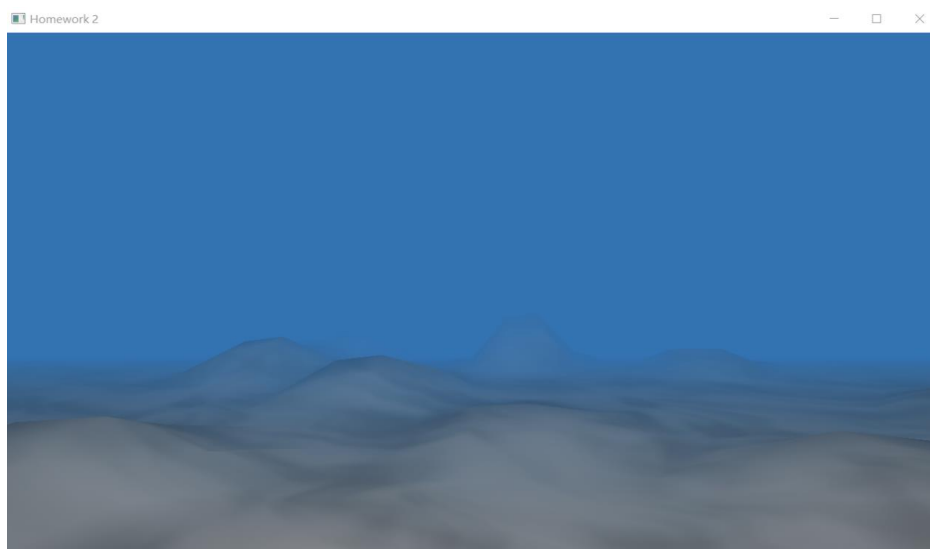


```
D:\xinjianwenjianjia\openGL\test\Tools\QtCreator\bin\qtcreator_process_stub.exe
QT_SCALE_FACTOR to set the application global scale factor.
load model: objMesh/SM_URockB.obj
load model: objMesh/SM_URockC.obj
load model: objMesh/SM_URockD.obj
load model: objMesh/SM_RedAlgaeB.obj
load model: objMesh/SM_RedAlgaeC.obj
load model: objMesh/SM_SeaweedA.obj
load model: objMesh/SM_SeaweedC.obj
load model: objMesh/SM_SeaweedD.obj
load model: objMesh/SM_SpongeB.obj
load model: objMesh/SM_SpongeC.obj
load model: objMesh/SM_AlgaeA.obj
load model: objMesh/SM_CorallineB.obj
load model: objMesh/SM_land.obj
libpng warning: iCCP: known incorrect sRGB profile
load image: Textures/T_URockB_BC.png
libpng warning: iCCP: known incorrect sRGB profile
load image: Textures/T_URockC_BC.png
libpng warning: iCCP: known incorrect sRGB profile
load image: Textures/T_URockD_BC.png
libpng warning: iCCP: known incorrect sRGB profile
load image: Textures/T_FishA_BC.png
libpng warning: iCCP: known incorrect sRGB profile
load image: Textures/T_FishB_BC.png
libpng warning: iCCP: known incorrect sRGB profile
load image: Textures/T_FishC_BC.png
libpng warning: iCCP: known incorrect sRGB profile
load image: Textures/T_RedAlgae_BC.png
libpng warning: iCCP: known incorrect sRGB profile
load image: Textures/T_Seaweed_BC.png
```

2. 海底渲染效果:



海底只加上天空的渲染效果:



3. 鱼群生成和鱼的运动以及静态物体的加载:

对比前两张图，可以看到鱼的数量增加，鱼的位置也发生了改变。两个红圈内为同一条鱼，可以观察到游动的角度发生偏转。



鱼群生成达到最大值:下图是程序运行十几秒之后的截图,可见鱼的数量不会一直增长下去,到达一个最大值后会保持稳定(总共三个鱼群)。



4. 水草的飘动



