浙江大学

计算机图形学 大程实验报告

1. 学生姓名 ： 刘佳润 学号： 3180105640

2. 学生姓名 ： 李沛瑶 学号： 3180101940

3. 学生姓名 ： 吴润朴 学号： 3180105182

4. 学生姓名 ： 代瑾宜 学号： 3180105181

2019~2020 春夏学期 2020 年 6 月 23 日

Part One：项目介绍

一、游戏背景介绍

Minecraft 是一款沙盒游戏，玩家可以在游戏中的三维空间里创造和破坏游戏里的方块，甚至在多人服务器与单人世界中体验不同的游戏模式，打造精妙绝伦的建筑物，创造物和艺术品。该游戏重点在于让玩家根据自己的想法去探索、交互，并且改变游戏开局生成的随机化地图。

本工程主要实现了一个小型的 Minecraft 游戏，游戏开局随机生成地图，玩家可以根据自己的需要建造或者修改自己的模型，构建属于自己的世界地图。本工程在实践过程中，根据《计算机图形学》课程所学，使用 obj 格式导入了特色化的树木、飞鸟和太阳等模型及其个性化贴图，使得游戏场景更加丰富，并实现了场景漫游、连续 obj 绘制实现动画效果、光源效果切换、导入 obj 模型、更换匹配不同贴图、游戏中截图、导出 obj 模型等等丰富的功能。本工程旨在实现游戏基本功能的同时，丰富场景变化，融入个人特色创新，实践和应用课程所学知识，锻炼对计算机图形学相关技术的熟练运用能力。

二、玩家操作指南

WASD：前后左右移动
Z：视角远近缩放（望远镜模式）
L：调节光照，切换白天/黑夜模式
Q：切换要放置的方块的类型
鼠标移动：转动视角
鼠标左键：选中并击碎已有的方块
鼠标右键：建立新的方块
空格：弹跳
P：截图（截图保存在 screenshot 文件夹中）
Q：切换基本方块的不同贴图
O：导出 OBJ 格式文件（保存在 obj 文件夹中）
F：切换飞行模式

Part Two：工程结构

一、文件结构及配置环境介绍（包括怎么配置工程，怎么运行 exe）

- 文件结构：

include 文件夹：存放工程中所用到的外置库，包括 glut、glew、glm 等。

lib 文件夹：存放所用到的 lib 文件（静态数据连接库），起到链接程序和函数（或子过程）的作用。

chunk 文件夹：引用区块生成技术，在工程中配合柏林噪声用于随即地形的生成。如图展示，游戏中的世界地形由随机产生的 chunk 组成。

screenshot 文件夹：存放游戏过程中生成的截图。

shaders 文件夹：存放 shaders 类中需要用到的 glsl 语言文件。

texture 文件夹：存放游戏中所用到的贴图。

obj 文件夹：存放游戏中导入的 obj 格式模型。

export 文件夹：存放游戏中导出的 obj 格式模型。

● 二、不同模块及模块间的交互与功能介绍

1. 场景渲染部分

CubeRender 模块：方块的渲染

Chunk 模块：区块与地形的生成，调用柏林噪声函数和方块渲染模块

Skybox 模块：天空盒的渲染

World 模块：世界的生成，调用 Skybox 模块 Chunk 模块与 Tree 模块（obj 读取）与 BallRender 模块

Camera 模块：主要负责实现游戏中的场景漫游，通过移动位置坐标、调整视角坐标来实现漫游。下文中有 camera 技术的详细实现方式介绍。

Light 模块：光照类

2. 物体渲染部分

ObjReader 模块：obj 格式文件的读入

Tree 模块：对 obj 格式文件的绘制与渲染

ObjWriter 模块：obj 格式文件的导出

BallRender 模块：球体的渲染

3. 资源管理部分

Texture 模块：纹理图片导入与绑定，其中包括二维纹理和三维纹理数组

Shader 模块：着色器编译

ResourceManagement 模块：资源管理，一个静态类。所有物体渲染的纹理和着色器调用通过该类的函数完成。

4. 交互与游戏功能部分

Screenshot 模块：截图功能

Game 模块：设定交互动作调用函数，导入资源，场景渲染、物体渲染、视角渲染

● 三、包含的外置库介绍

1. glew 库：OpenGL 自带的扩展库
2. glm 库：第三方数学库，用于对相关向量、矩阵进行操作

glm 常用的数据类型

- **vec2** 二维向量
- **vec3** 三维向量
- **vec4** 四维向量
- **mat2** 二阶矩阵
- **mat3** 三阶矩阵
- **mat4** 四阶矩阵

说明：对于 vec4 四维向量而言，第四个维度中 1 代表坐标，0 代表方向

常用的函数

- **glm::radians()** 角度制转弧度制
- **glm::translate()** 创建一个平移矩阵，第一个参数是目标矩阵，第二个参数是平移的方向向量
- **glm::rotate()** 创建一个将点绕某个轴旋转 x 弧度的旋转矩阵，第一个参数是弧度，第二个参数是旋转轴
- **glm::scale()** 创建一个缩放矩阵，第一个参数是目标矩阵，第二个参数是缩放系数

创建裁剪矩阵的函数,位于 glm/ext/matrix_clip_space.hpp，这个文件存放裁剪空间相关的 API

- **glm::ortho(float left, float right, float bottom, float top, float zNear, float zFar)**；前两个参数指定了平截头体的左右坐标，第三和第四参数指定了平截头体的底部和顶部。通过这四个参数我们定义了近平面和远平面的大小，然后第五和第六个参数则定义了近平面和远平面的距离。
- **glm::perspective(float fovy, float aspect, float zNear, float zFar)**；第一个参数为视锥上下面之间的夹角，第二个参数为宽高比，即视窗的宽/高，第三第四个参数分别为近截面和远界面的深度

除此之外，利用 glm 库构造某些特殊的矩阵也是非常常用的：

- **构造模型矩阵**

```
glm::mat4 mat = transmat3 * transmat2 * transmat1 * mat;
```

- **构造视图矩阵**

```
glm::mat4 mat = glm::LookAt(CameraPos, CameraTarget, upVector);
```

若相机正置，则 upVector = glm:vec3(0, 1, 0)

- **构造投影矩阵**

```
glm::mat4 mat = glm::perspective(FoV, AspectRatio, NearClipPlane, FarClipPlane);
```

3. SOIL 库：一个简单的适用于 OpenGL 的图形读写库，用于导入、解析 jpg、png 等格式的图片

Part Three：核心技术

一、图形资源管理和图形渲染管线技术

1. 纹理

本工程中所渲染的物体基本都是带有纹理的。有关纹理和贴图的附着不再赘述，特别说明的一点是在对游戏中的立方体（包括天空盒，用到环境纹理）进行纹理附着的时候用到了“立方体纹理（Cube Texture）”技术，由一个立方体（六个面/六张纹理）每个面上的二维图组成，是一个包含了上下左右前后六个面的纹理组，需要用到 uvw 三个维度的坐标来描述。具体在下文会介绍。

2. 着色器

首先说明一下 glsl——我们编写的着色器采用 OpenGL 的 glsl 语言，这是一种类 C 语言，为图形计算量身定制的，它包含一些针对向量和矩阵操作的有用特性。其基本格式如下：

```
#version version_number    //声明版本
in type in_variable_name;     //输入
out type out_variable_name;   //输出
uniform type uniform_name;    //uniform
int main()
{
  // 处理输入并进行一些图形操作
  ...
  // 输出处理过的结果到输出变量
  out_variable_name = weird_stuff_we_processed;
}
```

glsl 语言包括 C 的大多数数据类型，还支持向量和矩阵两种容器。

我们希望每个着色器都可以独立输入输出，进行高效的数据交流和传递。glsl 定义了 in 和 out 来实现输入输出。只要一个输出变量与下一个着色器阶段的输入匹配，就会传递下去。

顶点着色器：从顶点数据中直接接受输入。使用 location 这一元数据指定输入变量。一般我们用额外的 layout 标识来将其链接到顶点数据。

片元着色器需要一个 vec4 颜色输出变量，因为片元着色器需要生成一个最终输出的颜色。综上，我们需要在发送方着色器中声明一个输出，在接收方着色器中声明一个类似的输入。当二者类型、名字一致时，OpenGL 就会在链接程序对象时将其链接在一起。

Uniform 是一种从 CPU 应用向 GPU 着色器发送数据的方式。uniform 是全局的，全局意味着 uniform 变量必须在每个着色器程序对象中都是独一无二的，而且它可以被着色器程序的任意着色器在任意阶段访问，然后，无论你把 uniform 值设置成什么，uniform 会一直保存它们的数据，直到它们被重置或更新。

本工程中的 Shader 类是对着色器进行读取、编译的接口类。

3. 图形资源管理

我们把纹理和着色器的接口都交给资源管理器 ResourceManager 类。通过这个函数统一执行、渲染。这是一个静态的、实际不存在的类。我们只是想通过这个接口更加统一、完整地执行渲染。

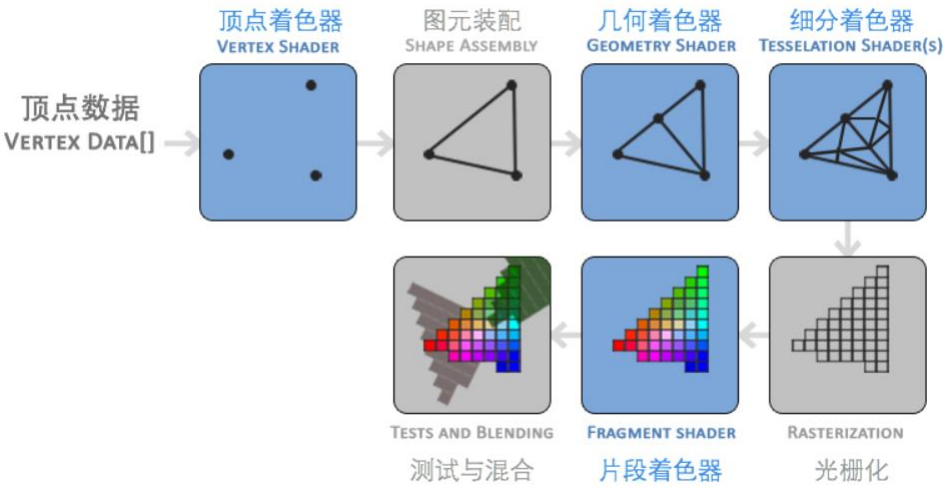
我们将该类的构造函数保护，保证了其他类不能创建类的实例，同时可以调用到静态域。

从文件读取的函数在类的私有域中，共有函数通过接口调用。

4. 图形渲染管线（Pipeline）技术

应用着色器的时候，采用了图形渲染管线（Pipeline）。图形渲染管线指的是对一些原始数据经过一系列的处理变换并最终把这些数据输出到屏幕上的整个过程。

图形渲染管线的整个处理流程可以被划分为几个阶段，上一个阶段的输出数据作为下一个阶段的输入数据，是一个串行的，面向过程的执行过程。每一个阶段分别在 GPU 上运行各自的数据处理程序，这个程序就是着色器。



流程说明：

顶点数据是一些顶点的集合，顶点一般是 3 维的点坐标组成。

基本图元（Primitives）包括点，线段，三角形等，是构成实体模型的基本单位，需要在传入顶点数据的同时通知 OpenGL 这些顶点数据要组成的基本图元类型。

顶点着色器（Vertex Shader）包含对一些顶点属性（数据）的基本处理。

基本图元装配（Primitive Assembly）把所有输入的顶点数据作为输入，输出制定的基本图元。

几何着色器（Geometry Shader）把基本图元形式的顶点的集合作为输入，可以通过产生新顶点构造出新的（或是其他的）基本图元来生成其他形状。

细分着色器（Tessellation Shaders）可以把基本图元细分为更多的基本图形，创建出更加平滑的视觉效果。

光栅化（Rasterization）即像素化，把细分着色器输出的基本图形映射为屏幕上网格的像素点，生成供片段着色器处理的片段（Fragment），光栅化包含一个剪裁操作，会舍弃超出定义的视窗之外的像素。

片段着色器（Fragment Shader）的主要作用是计算出每一个像素点最终的颜色，通常片段着色器会包含 3D 场景的一些额外的数据，如光线，阴影等。

测试与混合是对每个像素点进行深度测试，Alpha 测试等测试并进行颜色混合的操作，这些测试与混合操作决定了屏幕视窗上每个像素点最终的颜色以及透明度。

在渲染一个物体的时候，涉及到两个数据传输的流程，一个是将数据传输到 GPU 中的内存、另一个是将 GPU 中的数据传输给显卡。GPU 的内存通过**顶点缓冲对象（VBO，Vertex Buffer Object）**来管理内存，它会存在 GPU 内存中存储大量顶点。一般通过如下方式调用、生成、绑定 VBO，我们以游戏中的区块生成为例：

```
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
// 将准备好的顶点数据复制到缓冲的内存中，用static格式表示基本不会改变数据，缓冲区相对静态
glBufferData(GL_ARRAY_BUFFER, data.size() * sizeof(float), &data[0], GL_STATIC_DRAW);
// 指定顶点的解析方式
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(float) * 12, (void*)0);
```

glVertexAttribPointer 中的参数的意义分别是：

第一个参数为顶点着色器中 `layout (location=0) in vec3 position;` 中的 location 的值。

第二个参数为第二个参数指定顶点属性的维数，如果是 vec3，它由 3 个值组成，所以大小是 3。

第三个参数为数据的类型。

第四个参数为是否希望数据被标准化，如果我们设置为 GL_TRUE，所有数据都会被映射到 0（对于有符号型 signed 数据是-1）到 1 之间。显然我们不希望这样。

第五个参数叫做步长(stride)，它告诉我们在连续的顶点属性组之间的间隔。

第六个参数表示位置数据在缓冲中起始位置的偏移量(offset)。

为了简化绘制 VBO 时的流程，我们提出 **VAO（Vertex Array Object），顶点数组对象**。当一个 VAO 被创建绑定之后，任何随后的顶点属性调用都会储存在这个 VAO 中。 这样一来如果有多个 VBO 对象，在渲染绘制时，就不用执行很多次前面提到的渲染程序，只需要执行一次绑定的 VAO 的渲染 API 即可。

VAO 的生成与绑定与上文类似：

```
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
```

那么我们在渲染物体的时候，只需要调用 VAO 即可。

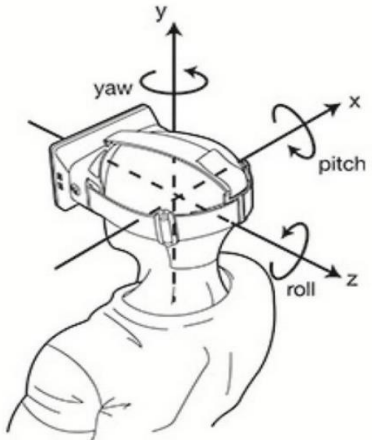
```
glBindVertexArray(vao);
glDrawArrays(GL_TRIANGLES, 0, vertsNum);
```

我们有时也会对顶点进行索引处理，采用一系列的整数数组来作为 indices，通过调用这个来获取相应位置的顶点坐标。此时用到的函数为 glDrawElements()。

● 二、camera 视角技术

在 Camera 类中，我们模拟了一个人的头部来作为视角。通过移动位置坐标、调整视角坐标来实现漫游。

在 3D 系统中的人头动作牵扯到三个角：roll/pitch/yaw。根据笛卡尔的三维空间右手系，如下图所示。pitch 是围绕 X 轴旋转，也叫做俯仰角。yaw 是围绕 Y 轴旋转，也叫偏航角。roll 是围绕 Z 轴旋转，也叫翻滚角。这三个角度合称为姿态角，或者欧拉角。在本游戏中，我们不牵扯翻滚角，即假设这个人不会歪头。



举例说明。我们在计算人的视线来确定“Front”即正前方这个方向的时候，用了如下

代码：

```
glm::vec3 front;
front.x = cos(glm::radians(this->Yaw)) * cos(glm::radians(this->Pitch));
front.y = sin(glm::radians(this->Pitch));
front.z = sin(glm::radians(this->Yaw)) * cos(glm::radians(this->Pitch));
this->Front = glm::normalize(front);
```

● 三、mousepicker 技术

简要来说，mousepicker（鼠标拾取）是指根据鼠标在屏幕上的二位位置位置计算出一条在三维世界中的射线，同时考虑三维物体的空间位置关系，计算出最近的相交物体位置，将其作为选中的物体。

在介绍 glm 库中提到了，我们建立了投影矩阵 projectionmatrix 和视角矩阵 viewmatrix，其中投影矩阵是由世界确定，视角矩阵调用了 camera 类中的相关成员变量。从二维到三维的计算过程涉及到一系列相关的矩阵变换。这一过程经历了从屏幕上的坐标到 OpenGL 坐标（Normalized），再到立体化，再转换到视角矩阵（Eye Space），再到世界坐标（World Space）。

在这一过程中，转换到视角坐标运用了投影矩阵的转置矩阵，转换到世界坐标运用了视角坐标的转置矩阵。通过 glm 库的函数 inverse()就可以实现

最后我们进行相交判断：通过前面计算出来的射线，让它与我们的实体相交。通过获取射线的方向、当前 camera 的位置以及目标单位立方体中心的坐标，我们可以得到它与哪些面相交，并得到交点。通过比较 distance（视距），可以判断最近的交点与所在的面。

● 四、地形渲染与生成技术

Minecraft 游戏中的基本地形单位叫做 chunk（区块）。工程虽然没有实现无限大地图，但是地形的渲染方式也是使用 chunk。在 chunk 的渲染中也是采用了基本的渲染管线，绑定了区块的 vao。但是区块中方块的位置（主要是海拔高度）的确定是采用了一种 3D 地形生成的技术，叫做 Perlin Noise（柏林噪声）。

Perlin noise 是一种非常有用强大的算法，常用于生成看似杂乱而又有序的内容。尤其特别适合用于游戏和其他视觉媒体。

我们以二维的柏林噪声为例，下图是该点在 2 维空间上的表示：

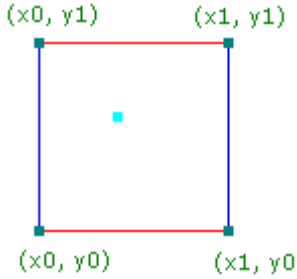


图 1：小蓝点代表输入值在单元正方形里的空间坐标，其他 4 个点则是单元正方形的各顶点

接着，我们给 4 个顶点（在 3 维空间则是 8 个顶点）各自生成一个伪随机的梯度向量。梯度向量代表该顶点相对单元正方形内某点的影响是正向还是反向的（向量指向方向为正向，相反方向为反向）。而伪随机是指，对于任意组相同的

输入，必定得到相同的输出。因此，虽然每个顶点生成的梯度向量看似随机，实际上并不是。这也保证了在梯度向量在生成函数不变的情况下，每个坐标的梯度向量都是确定不变的。

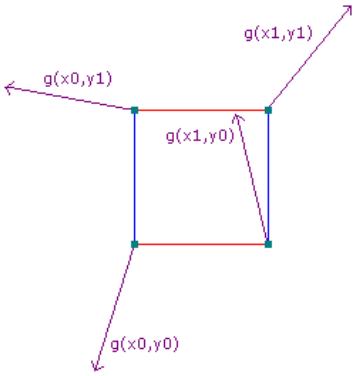


图 2：各顶点上的梯度向量随机选取结果

请注意，上图所示的梯度向量并不是完全准确的。在本文所介绍的改进版柏林噪声中，这些梯度向量并不是完全随机的，而是由 12 条单位正方体（3 维）的中心点到各条边中点的向量组成：

```
(1,1,0),(-1,1,0),(1,-1,0),(-1,-1,0),
```

```
(1,0,1),(-1,0,1),(1,0,-1),(-1,0,-1),
```

```
(0,1,1),(0,-1,1),(0,1,-1),(0,-1,-1)
```

采用这些特殊梯度向量的原因在 Ken Perlin's SIGGRAPH 2002 paper: Improving Noise 这篇文章里有具体讲解。

接着，我们需要求出另外 4 个向量（在 3 维空间则是 8 个），它们分别从各顶点指向输入点（蓝色点）。下面有个 2 维空间下的例子：

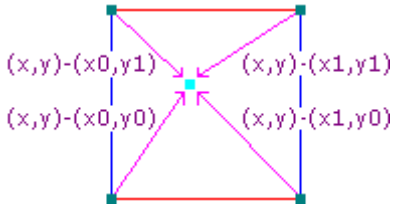


图 3:各个距离向量

接着，对每个顶点的梯度向量和距离向量做点积运算，我们就可以得出每个顶点的影响值：

```
grad.x * dist.x + grad.y * dist.y + grad.z * dist.z
```

这正是算法所需要的值，点积运算为两向量长度之积，再乘以两向量夹角余弦：

dot(vec1,vec2) = cos(angle(vec1,vec2)) * vec1.length * vec2.length

换句话说，如果两向量指向同一方向，点积结果为：

1 * vec1.length * vec2.length

如果两向量指向相反方向，则点积结果为：

-1 * vec1.length * vec2.length

如果两向量互相垂直，则点积结果为 0。

0 * vec1.length * vec2.length

不难看出，顶点的梯度向量直接决定了这一点。下面通过一副彩色图，直观地看下各顶点的影响值：

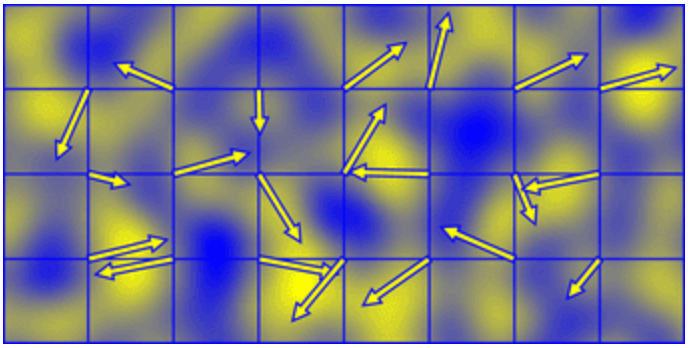


图 4：2D 柏林噪声的影响值

下一步，我们需要对 4 个顶点的影响值做插值，求得加权平均值（在 3 维空间则是 8 个）。算法非常简单（2 维空间下的解法）：

```
// Below are 4 influence values in the arrangement:
int g1, g2, g3, g4;
int u, v;
// These coordinates are the location of the input coordinate in its unit square.
// For example a value of (0.5,0.5) is in the exact center of its unit square.
int x1 = lerp(g1,g2,u);
int x2 = lerp(g3,h4,u);
int average = lerp(x1,x2,v);
```

至此，整个柏林噪声算法还剩下最后一块拼图了：如果直接使用上述代码，由于是采用 lerp 线性插值计算得出的值，虽然运行效率高，但噪声效果不好，看起来会不自然。我们需要采用一种更为平滑，非线性的插值函数：**fade 函数**，通常也被称为 **ease curve**(也作为缓动函数在游戏中广泛使用)：

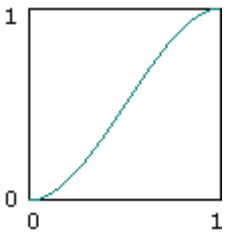


图 5：ease curve

ease curve 的值会用来计算前面代码里的 u 和 v，这样插值变化不再是单调的线性变化，而是这样一个过程：初始变化慢，中间变化快，结尾变化又慢下来（也就是在当数值趋近于整数时，变化变慢）。这个用于改善柏林噪声算法的 fade 函数可以表示为以下数学形式：

$$6t^5 - 15t^4 + 10t^3$$

为了使得生成的地形更加真实，我们在生成噪声的时候使用了**倍频（Octave）**技术。随着倍频增大，噪声对于最终叠加噪声的影响程度变小。做数据预处理时，就很适合使用多组倍频叠加来模拟更自然的噪声（比如用于提前生成游戏地形等）。

$$frequency = 2^i$$

$$amplitude = persistence^i$$

● 五、OBJ 文件读取

OBJ 文件是 Alias|Wavefront 公司为它的一套基于工作站的 3D 建模和动画软件"Advanced Visualizer"开发的一种标准 3D 模型文件格式.。其不包含动画，材质特性，贴图路径，动力学，粒子等信息，而主要支持多边形模型。Obj 文件不需要任何文件头（还是用几行简单的注释作为文件开头）。OBJ 文件由一行行文本组成，注释行以符号“#”为开头。有字的行都由一两个标记字母也就是关键字(Keyword)开头，关键字可以说明这一行是什么样的数据。在本次实验中主要用到的数据类型包括以下几种：

1.模型顶点（v）：在 obj 模型中模型的顶点是通过 v 进行标识记录的，格式为“v x y z”。其中 xyz 分别表示 xyz 坐标值。

```
v  19.3312 -0.3449 -0.2700
v  19.3934 -0.4898 -0.1651
```

2.顶点法线（vn）：法线决定着对应对象的可见性。法线朝内，对象不可见。法线朝外，对象可见。模型面的法线是可以根据模型顶点法线进行计算得到，所以 obj 中的法线用顶点法线表示就可以。

```
vn 0.7267 -0.4639 -0.5067
vn -0.9536 0.2941 0.0643
```

3.顶点贴图坐标（vt）：通过 uv 坐标就可以将模型与对应贴图

```
vt 0.6999 0.6571 0.0000
vt 0.7102 0.6651 0.0000
```

4.模型面（f）：模型面对象就像一个容器将上面的顶点坐标、顶点法线、顶点 uv 通过索引值进行组织在一起，封闭为模型体。其表达格式为:”f verticeIndex\vtIndex\vnIndex”其中 verticeIndex 表示顶点序列号，vtIndex 表示 uv 索引号，vnIndex 表示法线索引号，其中 vtIndex，vnIndex 可以缺失，但顶点索引必须有。

```
f 1/1/1 2/2/2 3/3/3
f 4/4/4 5/5/5 6/6/6
```

通过顺序读取 obj 文件中的信息，可以由模型面信息中的索引得到所有面所包含顶点的位置，法线，uv 等信息，将它们顺序储存（数量未知情况下使用 vector），将其复制到 VBO 中，在和 VAO 绑定之后进行绘制(此为优化方式)，即可将模型导入。导出 obj 文件则与之相反，即将此模型所有面的顶点信息按照上述格式写出即可。

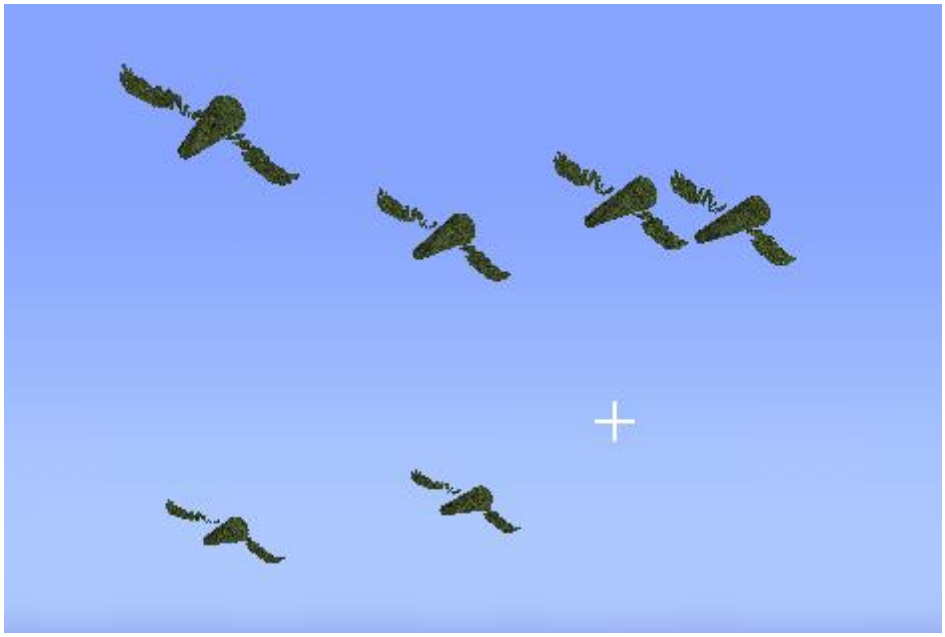
动画播放功能通过连续读取 obj 文件，对其网格进行多次的重绘来实现。在游戏中展现为天空中的飞鸟，不仅由简单的位置平移，还能体现翅膀震动摇摆的细节。具体实现方式：利用了 opengl 计时器，在自己定的回调函数 timer 里修改下一次应该读取的 obj 文件地址，将其作为参数传入即将调用的游戏更新函数，再通过 glutPostRedisplay()对屏幕进行重绘。由于 obj 文件只记录静态模型的点面信息，所以要实现动画播放需要事先得到连续动态模型所有关键帧的 obj 文件，再通过插值得到中间帧。动画的流畅程度可以通过改变计时器回调函数调用间隔时间来修改，也可以通过不同类型的插值来实现改变（一般使用线性插值，但是当流畅度要求较高时，需通过物理方法来实现，例如手臂的摆动，符合人体力学的插值方法所得到的动画效果更佳）

Part Four：基本功能与加分项的实现与说明

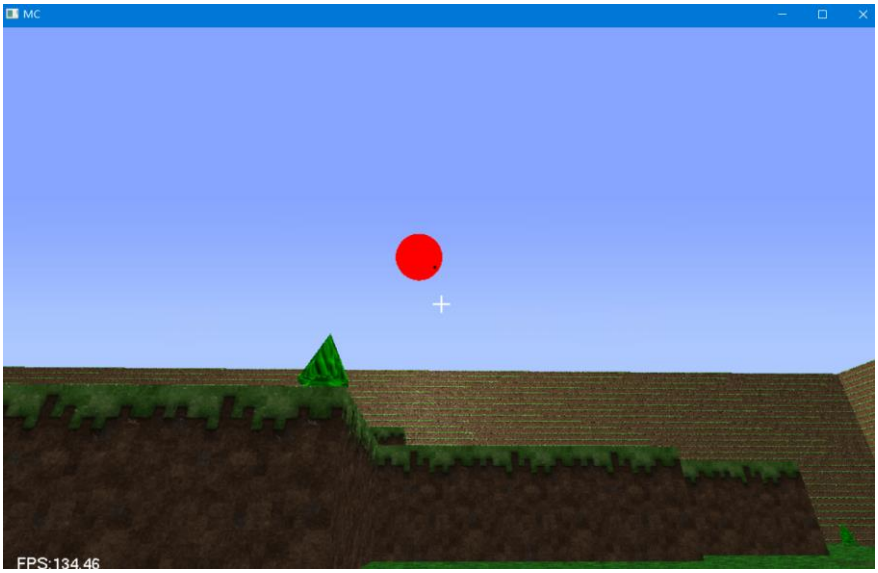
- 一、具有基本体素的建模表达能力

游戏中的基本立方体模型和树木模型展示：

特色模型飞鸟展示：

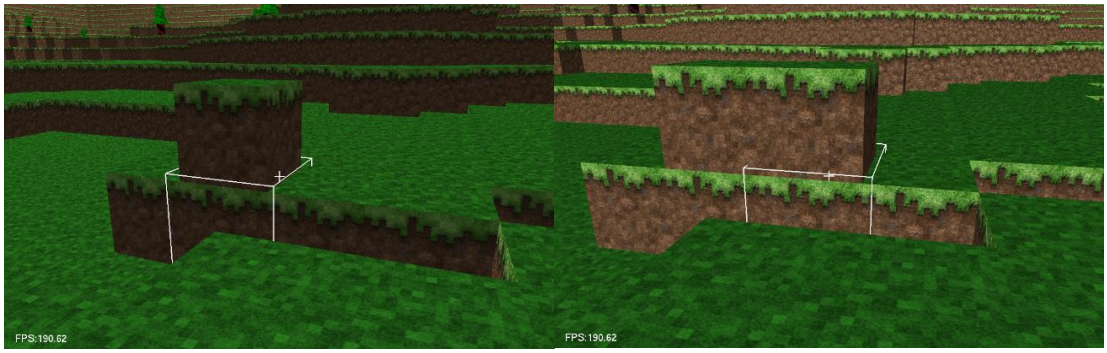


游戏中的“太阳”（球体）绘制展示：



在游戏中通过鼠标交互建立基本立方体的效果展示：

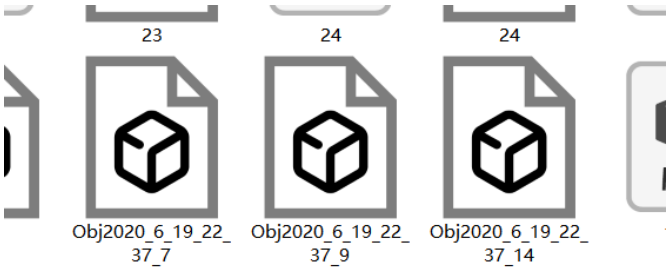
构建前：　　构建后：

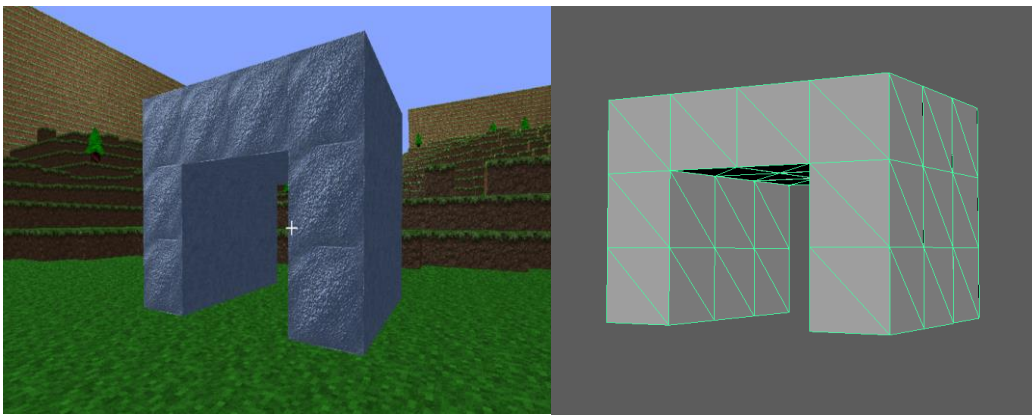


- **二、具有基本三维网格导入导出功能（OBJ 格式）**

游戏中实现的树木模型是根据 OBJ 格式的文件导入的模型：

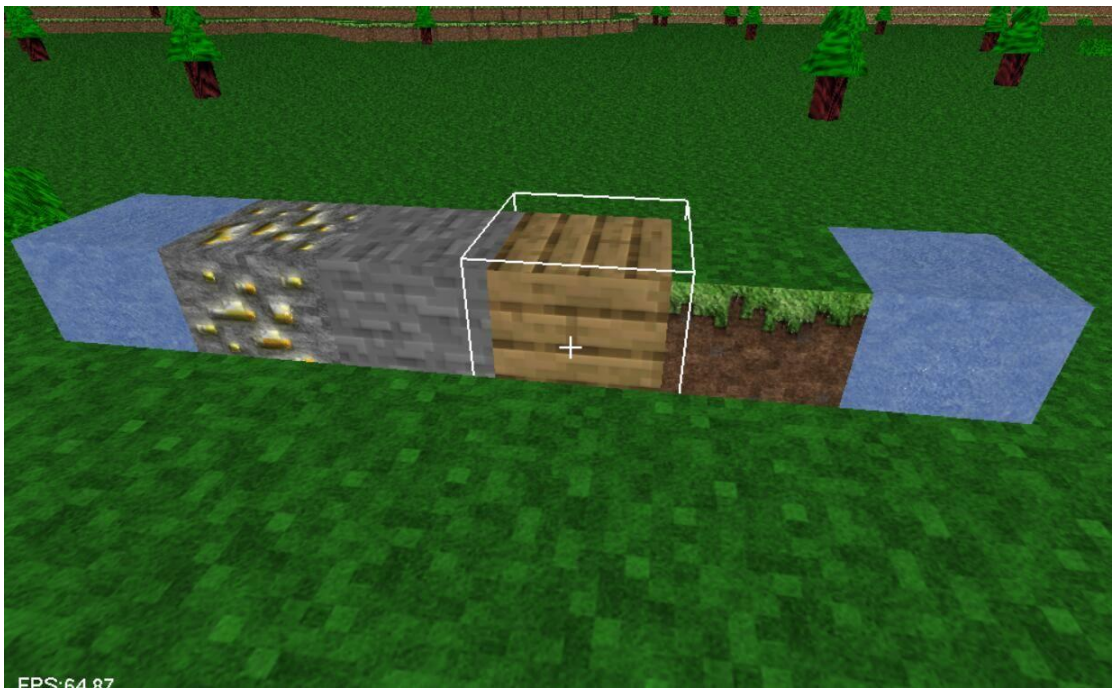
在游戏中放置的方块可以通过 0 键导出为 obj，并根据导出时间自动命名，保存在 obj 文件夹下。在 maya 等软件中导入可以显示基本形体：





- **三、具有基本材质、纹理的显示和编辑能力**

对于基本元素实现按 Q 键切换不同贴图，效果如下：

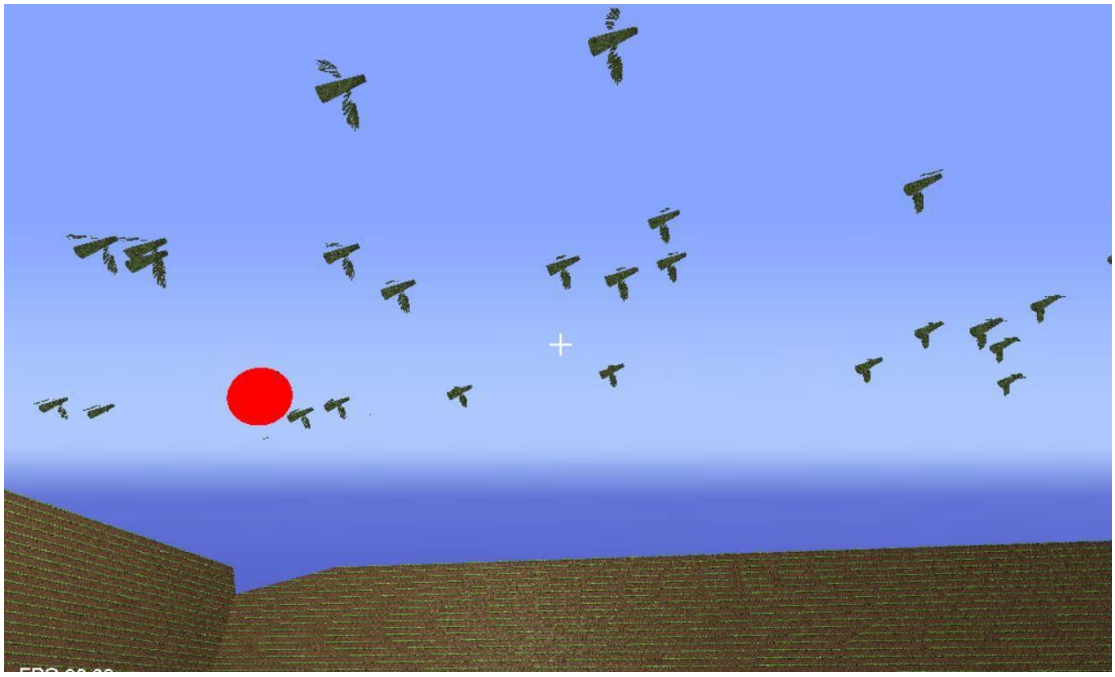


特别注意，我们的所有方块的纹理都是 3D 纹理：实现了三张贴图的混合，包括基本色彩，法线贴图和凹凸贴图。

另外，由于使用了着色器技术，纹理的效果可以和光照效果混合，产生反光的效果。

- **四、具有基本几何变换功能（旋转、平移、缩放等）**

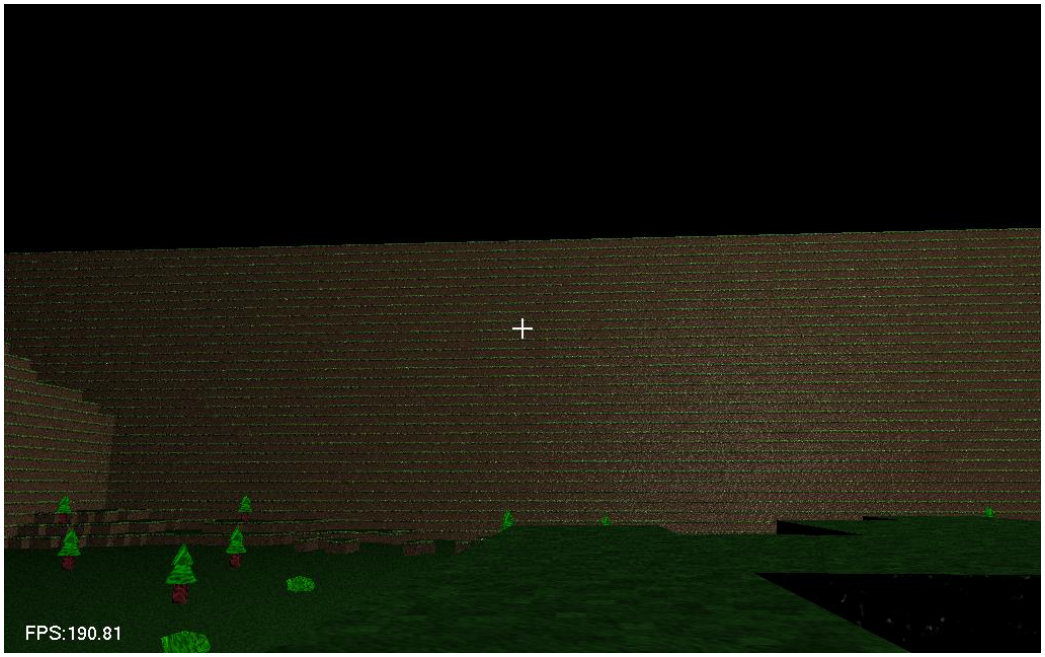
游戏中的“太阳”可以随光源进行移动，平移、旋转。

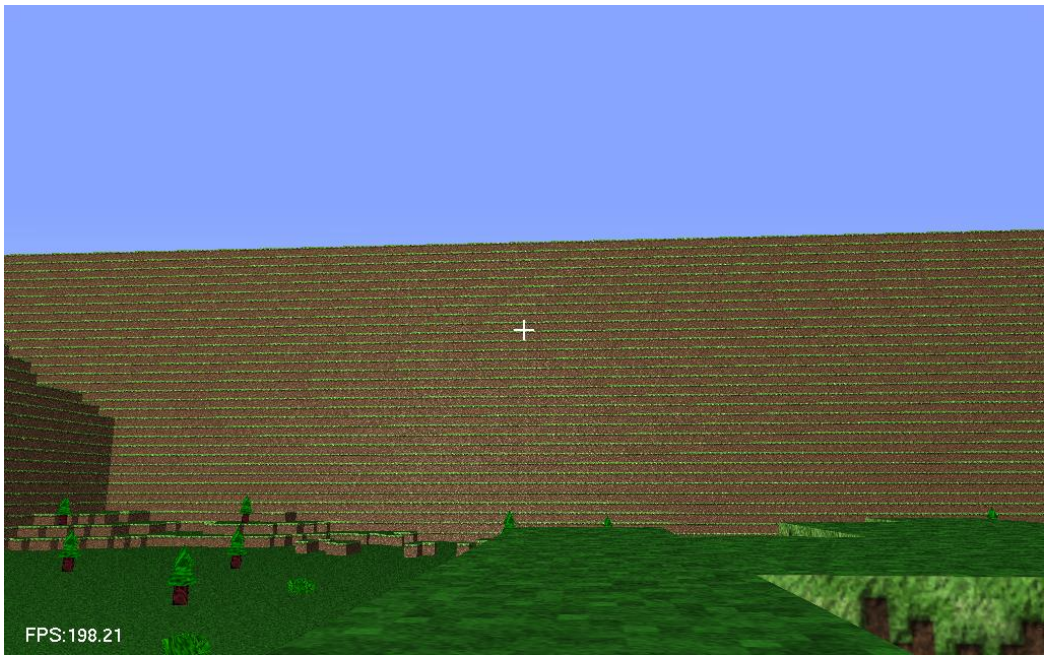


- **五、基本光照明模型要求，并实现基本的光源编辑（如调整光源**

的位置，光强等参数）

游戏中实现了“1”键对光线进行调节，在调节到最黑暗的程度下开启夜间模式，展示如下。





- ## 六、能对建模后场景进行漫游如 Zoom In/Out，Pan，Orbit，Zoom To Fit 等观察功能

  游戏中的 WASD 键可以操作人物的移动，空格键操作跳跃，进行场景中的漫游。
  Z 键可以对视角进行缩放，即“望远镜模式”。

- ## 七、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

  如图，在游戏中按 P 键可以实现对当前游戏场景的截图，并根据截图时间自动命名，保存在 screenshot 文件夹下：

如图，游戏中实现了空中飞鸟，通过多帧数据连续绘制实现动画播放的效果：

- 八、漫游时可实现碰撞检测

  实现的原理是简单的六面检测（包围盒）。

- 九、实现较为完整三维游戏

- 十、光照与纹理效果的叠加（shader）

Part Five：工程进度管理

- 一、组内成员分工

刘佳润：整体框架设计，基本要素建模表达，基本材质纹理表达，简单碰撞检测，绑定 VBO，VAO，优化 FPS，整合资源管理器

李佩瑶：obj 文件导入与导出，截图功能，利用柏林噪声生成初始化地形

吴润朴：连续 obj 读入实现动画，基本几何变换，基本光照与光照调整

代瑾宜：场景漫游的实现，着色器编写

- 二、大程完成进展表及重要时间节点

5.20 号：讨论主题，完成整体框架构成与基本要素的选择以及实现功能的种类

5.24 号：完成整体框架的搭建

5.28 号：完成基本几何体的建模，游戏场景整体框架搭建完成。

6.1 号：完成 obj 文件的导入与导出功能，实现了 obj 文件的连续读取以及动画的绘制。

6.5 号：完成场景整体光照的设置，为场景中的几何体增加纹理贴图。

6.10 号：完成场景漫游，视角变换功能以及截图功能

6.14 号：碰撞检测的完善

6.17 号：完成对程序的整体优化，包括引入 VBO，VAO 以及显示列表进行渲染

6.20 号：整体修改完成，撰写报告。

Part Six：总结与反思

1. 对于整体框架的搭建，我们参照了以往 opengl 图形学实验的代码，来设计了一套完成的游戏进程，包括了鼠标键盘的输入响应，更新，屏幕的重绘等等。在整体的搭建上，我们刚开始没能很好的考虑到各个模块之间的联系，耦合情况，所以初始时耗费时间较长。
2. 在完成过程中我们学习了许多图形学的新内容，自主去了解了包括 obj 文件格式，柏林噪声（产生随机地形），VAO，VBO 实现渲染的加速，以及怎样编写 shader，运用自己编写的着色器来控制渲染流水线上的渲染细节，在使得程序能够正常运行的基础上，节省了大量的时间，内存，提高了场景的绘制效率，从而提高了游戏的流畅程度，将整个程序变得更加高效。
3. 在整个项目的完成过程中，我们组员之间进行了大量的沟通与交流，因为此次的图形学大作业要求实现的是一个完整的游戏，游戏的各个功能模块之间存在着耦合，不能够单纯的分配任务后就自己写自己的，所以通过这次的大作业，也使得我们明白了合作完成工程的中有效的沟通与交流是多么重要。
4. 除开技术方面的实现，我们通过此次大作业也明白了设计一个游戏，需要考虑很多方面，

通过技术实现丰富的功能是其一，同等重要的是整个游戏给用户的体验，包括流畅程度以及场景绘制的美感与设计感，所以我们在保证整体功能不受影响的基础上，为游戏场景增添了很多有趣的要素，有趣的功能，使得用户的游戏体验更好。

5. 在完成代码时，我们也遇到了很多问题，包括内存溢出，读写异常等等问题，我们也通过一系列的努力将问题解决。
6. 我们对我们的游戏也提出了一定的改进意见，比如放置方块的种类是否可以多样，比如圆柱，球体，三角等等，还有纹理的叠加，我们此次大作业实现了三维纹理，但是未能实现双重纹理，这一点也可以在后续过程进行改进。再碰撞检测方面，我们实现的简单包围盒法碰撞检测，还未能检测出具体是那一个面产生的碰撞，在这一点上也有改进空间。由于我们游戏 mycraft 的整体风格是像素风，所以在讨论的时候我们舍弃了对于更加逼真的光照效果的实现（比如光线跟踪算法等），但是在后期我们还希望能够实现光照的优化，使得场景光照效果更加真实。最后，我们还提出可以对场景的丰富度进行提高，现在看来我们的场景还较为单调，缺少实际生活元素，后期还可以增添类似于水，火等元素进去。