

浙江大学



**Computer Graphics**

**Final Project**

Date: 2019-06-26

# The War of Particle

Ke Yu

Gao Xinyu

Wu Sihan

Zhong Zipeng

(Ordered by Initial)

## Index

Chapter 1 综述 .....	5
1.1 项目功能介绍 .....	5
1.2 OpenGL 版本及介绍 .....	5
1.3 游戏逻辑 .....	6
Chapter 2 简单几何体的建模表达 .....	6
2.1 简单几何体绘制 .....	6
Chapter 3 复杂模型的导入导出与材质纹理编辑 .....	8
3.1 Obj 格式的导入与导出 .....	8
3.1.1 Obj 格式分析 .....	8
3.1.2 Obj 导入导出 .....	9
3.2 模型的材质与纹理 .....	10
Chapter 4 基本几何变换与场景漫游 .....	11
4.1 基本几何变换 .....	11
4.1.1 摄像机类 .....	11
4.1.2 飞船 spaceship 类的几何属性 .....	12
4.2 场景漫游 .....	13
4.2.1 漫游的依赖逻辑 .....	13
4.2.2 视角的切换 .....	13
4.3 场景中所有物体的比例协调和位置关系 .....	13
Chapter 5 光源设置、编辑与实时阴影 .....	14
5.1 光源设置: .....	14
5.2 光照编辑: .....	14
5.3 实时阴影 .....	15
5.3.1 阴影映射 .....	15
5.3.2 阴影修正 .....	17
Chapter 6 天空盒 .....	19
6.1 天空盒简介 .....	19
6.2 天空盒的创建 .....	20
6.2.1 立方体贴图纹理 .....	20
6.2.2 加载天空盒 .....	20
6.2.3 显示天空盒 .....	20
Chapter 7 粒子系统 .....	21
7.1 两个粒子系统的介绍 .....	21
7.2 粒子光晕——泛光 .....	22
7.2.1 帧缓冲 .....	22

7.2.2 高斯模糊 .....	22
7.2.3 gamma 矫正和 HDR .....	22
Chapter 8 碰撞系统与爆炸特效 .....	23
8.1 碰撞检测 .....	23
8.2 爆炸特效 .....	25
Chapter 9 动画播放与屏幕截取 .....	25
9.1 动画播放 .....	25
9.2 屏幕截取 .....	26
References .....	29

# Chapter 1 综述

## 1.1 项目功能介绍

本次图形学大作业，我们按照题目要求，制作了一个类似于飞机太空大战的游戏。

**程序能够基本功能实现：**

- 1) 基本基本体素（立方体、球、圆柱、圆锥、多面棱柱、多面棱台）的建模表达能力；
- 2) 能够导入导出 obj 格式的三维网格信息；3) 能够对导入的 obj 文件以及基本体素进行材质、纹理的显示和编辑；4) 对于游戏场景中的立体元素，能够进行基本几何变换，包括对飞机的平移、旋转，陨石等物体的自由移动；5) 在光照方面，能够显示光照以及阴影显示；6) 游戏开始后，飞机能够通过鼠标以及键盘进行自由的漫游活动；7) 游戏结束后，会出现“Gameover”的动画特效，以及能够对当前场景进行截图，导出并保存图片；

**程序的高级功能实现：**

- 1) 漫游时能够进行实时碰撞检测；2) 光照明模型细化，全局光照明（光子跟踪）；3) 能够实现粒子特效，飞机尾部喷射出的飞行火焰，以及射出的子弹都属于粒子特效；4) 构建了完整的三维射击类游戏，有很好的可玩性；5) 极强的游戏性使得程序具有一定的对象表达能力，能够良好的构建出太空飞机大战的效果；

## 1.2 OpenGL 版本及介绍

我们没有使用早期的 OpenGL 使用立即渲染模式，而是使用了 OpenGL3.3 版本的核心模式(Core-profile)，同时使用了 GLFW，GLAD 这两个库。GLFW 是一个 C 语言库，以及，它提供了一些渲染物体所需的最低限度的接口。它允许用户创建 OpenGL 上下文，定

义窗口参数以及处理用户输入。GLAD 是一个开源的库，它能解决我们上面提到的那个繁琐的问题。GLAD 的配置与大多数的开源库有些许的不同，GLAD 使用了一个在线服务。在这里我们能够告诉 GLAD 需要定义的 OpenGL 版本，并且根据这个版本加载所有相关的 OpenGL 函数。

## 1.3 游戏逻辑

这个游戏是玩家控制飞机模型在太空环境中漫游，飞机能够发射子弹，击中散落在宇宙空间中的、随机漫游的陨石和其他物体，同时击中后这些物体会发生碰撞检测，产生一个破碎效果；游戏中又一个生命周期系统，即飞船有一定数目的生命值，在陨石等物体碰撞一定次数后，自身发生破碎，并出现 “GameOver” 的特效。

# Chapter 2 简单几何体的建模表达

## 2.1 简单几何体绘制

想要绘制带有立体感的几何体，就需要将 2D 的画面转换为 3D 的画面，因此我们需要利用坐标变换来实现 opengl 中。

在开始进行 3D 绘图时，我们首先创建一个模型矩阵，这个模型矩阵包含了位移、缩放与旋转操作，它们会被应用到所有物体的顶点上，以变换它们到全局的世界空间。接下来我们需要创建一个观察矩阵。我们想在场景里面稍微往后移动，以使得物体变成可见的。在一个顶点着色器运行的最后，OpenGL 期望所有的坐标都能落在一个特定的范围内，且任何在这个范围之外的点都应该被裁剪掉，剩下的坐标就将变为屏幕上可见的

片段。

本次项目中，一共绘制了 4 种几何体一共 20 个，分别是立方体、球体、圆柱体和圆锥体。20 个几何体在 z 方向移动（即从视野前方朝向飞船飞来）。

### 2.1.1 立方体

在绘制时，一共需要三种坐标，第一类坐标是 3D 位置坐标，用来确定物体在场景中的位置；第二类坐标为法线坐标，与光照联系密切，由于顶点本身并没有表面，所以我们需要利用法线坐标来计算出物体的表面；第三类坐标是纹理坐标，用来标明该从纹理图像的哪个部分采样，并之后在图形的其它片段上进行片段插值。有了三类坐标之后，绑定顶点数组对象和顶点缓冲对象，即可利用顶点着色器来处理顶点数据，并利用片段着色器进行渲染计算并作最后的输出。

正方体的面与顶点少并且相对规则，所以直接声明一个顶点数组并将全部信息包含进去即可。

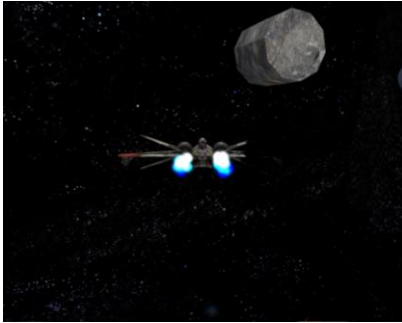
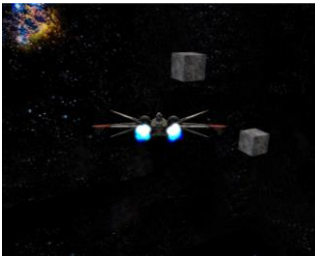


### 2.1.2 圆柱体+圆锥体+球体

由于圆柱体、圆锥体和球体 UV、法线、面以及纹理的信息相对比较复杂，我们可以从该类基本几何体的 obj 文件中获取相应的信息（利用文本文件打开 obj 文件，具体信息在下一章 obj 格式分析中会详细提到），并将数据部分复制到我们绘制几何体所要用的头文件中。

但是此时，几何体的法线、UV、面以及纹理的信息都是分开的，我们需要对不同几何体的这些信息整理到一个 vertices 数组中，并做成类似我们画正方体时所用的顶点数组的排列方式，再利用类似的方法绘制这些图形即可。

几何体绘制的相关顶点信息以及函数均在 GeometricSolid.h 头文件中。

**效果演示：**

圆柱体	立方体
	
圆锥体	球体
	

## Chapter 3 复杂模型的导入导出与材质纹理编辑

### 3.1 Obj 格式的导入与导出

#### 3.1.1 Obj 格式分析

OBJ 文件不需要任何种文件头(File Header), 尽管经常使用几行文件信息的注释作为文件的开头。OBJ 文件由一行行文本组成, 注释行以符号 “#” 为开头, 空格和空行可以随意加到文件中以增加文件的可读性。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。多行可以逻辑地连接在一起表



示一行，方法是在每一行最后添加一个连接符(/)。注意连接符(/)后面不能出现空格或 Tab 格，否则将导致 文件出错。

下列关键字可以在 OBJ 文件使用。在这个列表中，关键字根据数据类型排列，每个关键字有一段简短描述。在 opengl 绘制三角形中，我们需要的数据有以下几个：

顶点数据 v      v 后跟着的三个数据即对应的顶点坐标

几何体顶点 vn    vn 后跟着的两个数据即对应的纹理坐标

贴图坐标点 vt    vt 后跟着的三个数据即对应的法线坐标

面 f      f 后跟着的数据往往是 " \*/\* \*/\* \*/\* \*/\* ..." 每个空格之间代表每个顶点的信息， "\*" 中存储着对应的索引，顺序依次是顶点坐标、纹理坐标、法线坐标。用 "/" 分割该点的不同索引，这几个点按顺序首尾连接，注意在 OBJ 文件中索引是从 1 而不是从 0 开始。

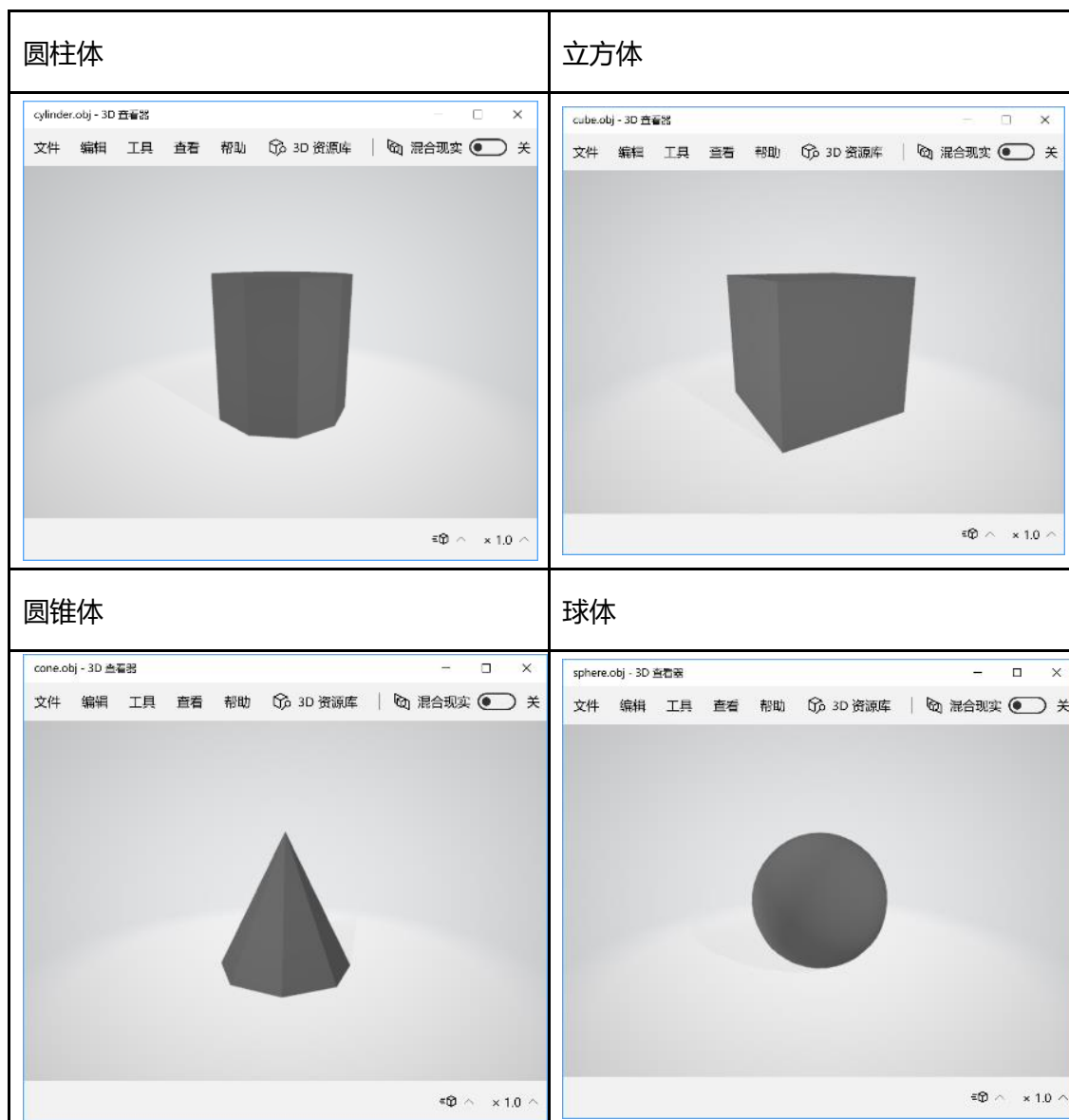
f 后可能跟着 3 个点以上的数据，代表这个面由多于 3 个点构成，此时我们需要将该面进行分割，分割成可绘制的三角面。

### 3.1.2 Obj 导入导出

知道了 obj 的文件格式后，我们就可以通过解析其文件内容来获得绘制所需要的基本信息，在这里我们选用 C 语言的 fscanf() 函数进行读取文件，其优点在于具有类似正则表达式的文件读取格式，可以按行便捷地读取数据，读取的时间也只与文件的行数成正比。借此我们就可以导入一般的 obj 格式的文件了，本次项目中导入的为飞船与陨石。

导出同理，整理获得要进行导出的 obj 基本信息后，利用 fprintf() 可以按行输出，就

可以得到导出的几何体。尝试用 win10 系统自带的 3D 查看器打开导出文件，效果如下：



## 3.2 模型的材质与纹理

物体的材质决定了物体对光照的反射与吸收情况，物体的纹理则决定了物体的外貌。材质文件 mtl 具有与 obj 文件很类似的组织形式。同样是关键字开头，后面跟着描述该关键字的对应数据，其中关于材质属性的以下几个：

环境光照系数  $K_a$   $K_a$  后跟着的三个数据即对应的 RGB 数值

漫反射系数  $K_d$   $K_d$  后跟着的三个数据即对应的 RGB 数值

镜面反射系数  $K_s$   $K_s$  后跟着的三个数据即对应的 RGB 数值

反光度  $N_s$   $N_s$  后跟着的单个数据即该材质的反光度

(在进行光照计算的时候, 我们只需要将他们与光照的 RGB 通道分别相乘即可)

在导入纹理的时候, 为了能够支持不同格式的纹理图片, 我们使用了 `stb_image` 库来进行读取基本图片, 读取图片后, 由于我们导入的 OBJ 物体上已经存在了纹理坐标, 所以不需要过多的额外操作, 只需要使用 `glGenTextures()` 等函数对材质进行基本设置, 在会之前绑定对应纹理即可。

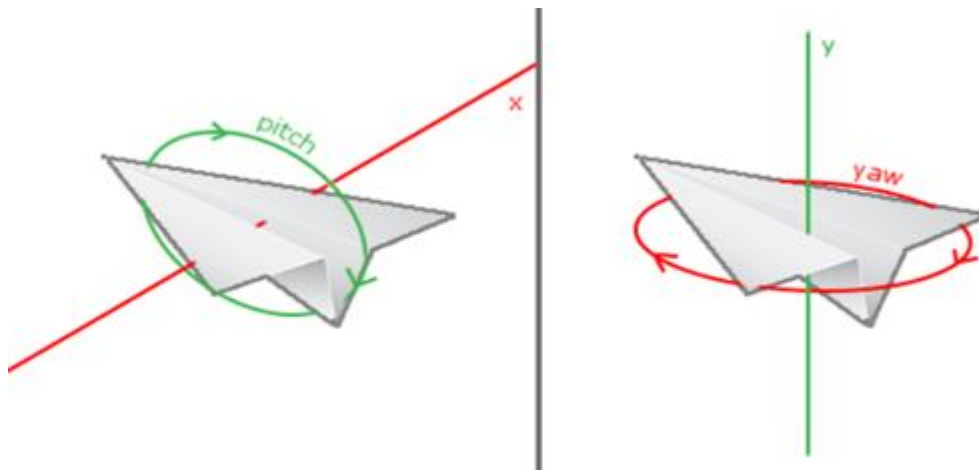
## Chapter 4 基本几何变换与场景漫游

### 4.1 基本几何变换

#### 4.1.1 摄像机类

场景的漫游基于摄像机类 (`Camera.h`), 在摄像机类中包含了摄像机类的基本属性和参数: 摄像机的位置 (`position`)、运动速度、俯仰角、航偏角、鼠标灵敏度、自身的三个矢量方向 (`front`、`up`、`right`)。Main 函数中所有的视口矩阵 (`view matrix`、`perspective matrix`) 全部从摄像机类中获得, 从而保证统一的场景透视逻辑。

· 俯仰角和航偏角:



通过计算鼠标的 X 偏移量和 Y 的偏移量，利用三角关系转化为俯仰角和航偏角的增量，再通过三角转换为 XYZ 轴方向矢量的分量。

- 鼠标灵敏度：

计算每一帧的时间差时，将时间差的数值叠加上鼠标灵敏度的系数。从而模拟控制鼠标的灵敏度。防止鼠标运动过快而造成的卡顿现象。

- 摄像机的漫游：

绘制新的每一帧时将摄像机的 position 进行增量，增量的数值为摄像机的 front 向量乘以摄像机的速度和时间差。由于回调函数会及时更新摄像机的 front 向量。于是我们可以得到正确的位置增量。

- 视口矩阵的获得

在每一帧内照相机获取飞船的位置，并相对于飞船设置自身合适的位置，然后将飞船的位置和摄像机自身的位置做差得到指向飞船的方向向量。以此为参数计算视口变换矩阵。并提供接口供外界获得。

## 4.1.2 飞船 spaceship 类的几何属性

飞船类的几何操作基于摄像机类，但不会自己生成视口矩阵。除此之外，飞船类可以进行

自身大小的缩放，方向的调整，以及控制自身的旋转等姿态。生成相对应的矩阵以供自身变换。

## **4.2 场景漫游**

### **4.2.1 漫游的依赖逻辑**

由于游戏逻辑的问题，场景漫游主要基于飞船进行。每时每刻不论飞船处于何种姿态，摄像机都只指向飞船。从另一点上讲，飞船的操作（及漫游操作）与视口的获得是分离进行的，这在一方面提高了操作的自由度，另一方面也使得代码逻辑变得清晰。我们甚至可以将视角绑定到任何一个物体上来获取虚拟世界的景象。

### **4.2.2 视角的切换**

通过设置照相机的位置和 front 向量朝向来获取对应的视角。游戏操作中支持五种视角变换：默认视角（中距离视角）、远距离视角（俯视视角）、近距离视角（飞船出仓视角）、驾驶员视角和旋转观察视角。不同的视角可以获得虚拟世界不同的观察效果，丰富了游戏体验。

## **4.3 场景中所有物体的比例协调和位置关系**

游戏场景中，存在导入的飞机模型、陨石模型以及各种基本几何体。飞机模型导入后缩小为原来的 0.001 倍，以协调和导入的陨石的大小比例。

同时，游戏中物体的初始位置，飞机位于坐标的原点位置，而其他几何体以及陨石，通过初始坐标赋值，在游戏进程中，根据写好的位移算法进行移动，以提供更好的游戏效果。

# Chapter 5 光源设置、编辑与实时阴影

## 5.1 光源设置：

OpenGL 的光照使用的是对现实世界进行简化的模型，这样处理起来会更容易一些。其中一个模型被称为冯氏光照模型(Phong Lighting Model)，其主要结构由 3 个分量组成：环境(Ambient)、漫反射(Diffuse)和镜面(Specular)光照。环境光照即为简化的全局照明；漫反射光照可以使物体产生显著的视觉，使物体上与光线方向越接近的片段能从光源处获得更多的亮度；镜面反射即添加高光效果；而在描述一个物体的材质时，可以利用这三个分量再加上反光度 (shininess)。

OpenGL 中有很多光照类型，本次实验中采用的是位置较远且无衰减的点光源 (初始位置为 glm::vec3(0.0f, 25.0f, 5.0f))，且可以对光的强度和光源位置进行编辑。

## 5.2 光照编辑：

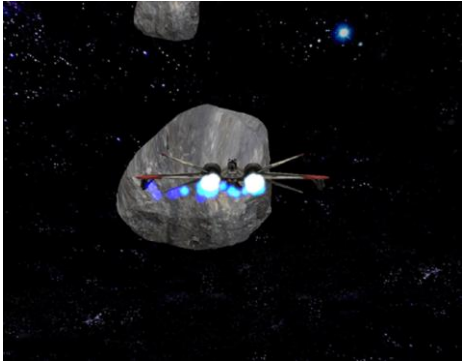
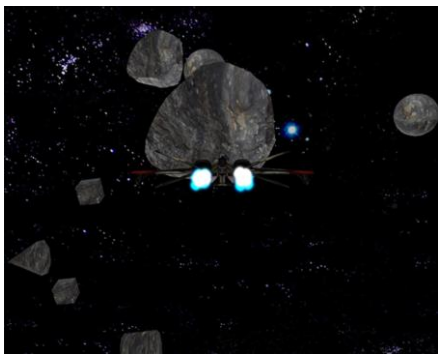

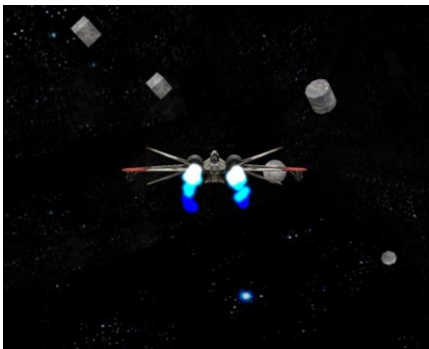
- 1、 位置编辑：利用 uniform 绑定着色器中的 lightPos 向量，并设置按键对 lightPos 的 x、y、z 分量的值进行修改。
- 2、 强度编辑：利用 uniform 绑定着色器中的 lighting 向量的强度系数 coef，并设置按键对该 coef 值进行修改 (1.0f <= coef <= 12.0f)。

注：coef 在片段着色器中的位置

```
vec3 lighting = coef*(ambient + (1.0 - shadow) * (diffuse + specular)) * color;
```

效果展示：

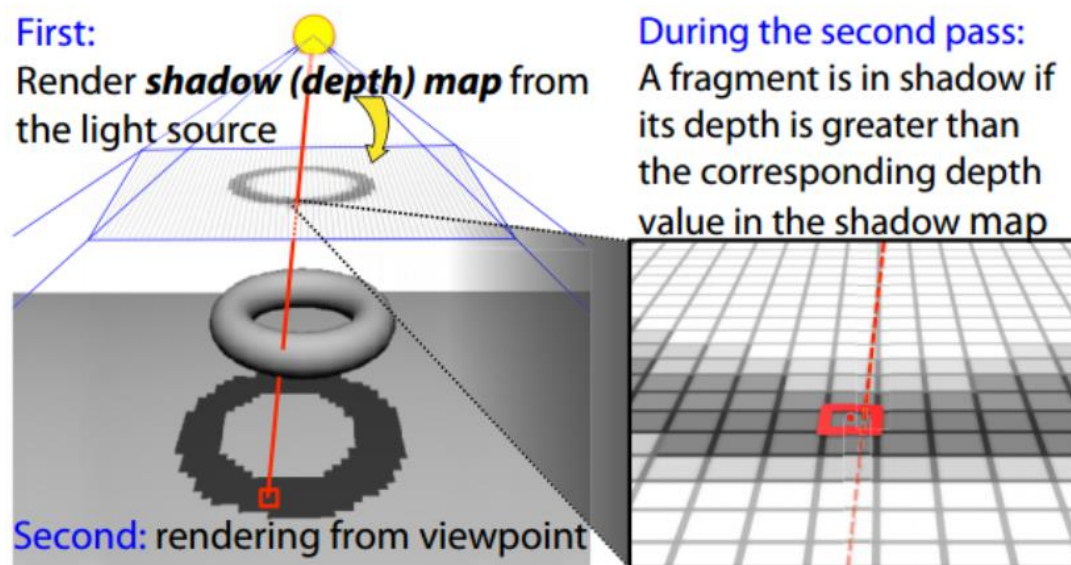
光源位置 1：	光源位置 2：
---------	---------

	
<p>光强 1:</p>	<p>光强 2:</p>
	

## 5.3 实时阴影

### 5.3.1 阴影映射

阴影映射的大致思路非常简单：先从光源位置进行渲染，对于光源发出的每条射线，找到最近距离，并存储为称为深度贴图；之后进行实际的渲染，对于每个表面点  $P$ ，只需找到和  $P$  在同一条光源发出射线上的深度贴图上的表项，比较  $P$  点到光源距离和表项值的大小，即可判断该点是否为阴影。



具体实现步骤：

### 第一步：生成深度贴图

生成一张深度贴图，即从光的透视图里渲染的深度纹理用于计算阴影。因为我们只关心深度值，所以要把纹理格式指定为 `GL_DEPTH_COMPONENT`，且渲染过程中不需要颜色缓冲。将深度值渲染到纹理的帧缓冲后，就可以从光的视角开始生成深度贴图了。

当以光的透视图进行场景渲染的时候，我们会用一个简单的着色器叫做 `shadowDepthShader`，其顶点着色器的作用是将一个单独模型的一个顶点使用 `lightSpaceMatrix` 变换到光空间中。由于没有颜色缓冲，最后的片元不需要任何处理，所以片元着色器使用一个空像素着色器即可。运行完该空像素着色器之后，深度缓冲会被更新。最后利用 `shadowDepthShader` 着色器调用所有相关的绘制函数来渲染场景，并在需要的地方设置相应的模型矩阵，即可渲染出场景的深度贴图。

### 第二步：生成阴影

正确地生成深度贴图以后我们就可以开始生成阴影了。首先需要在顶点着色器中需要先进行光空间的变换，即利用同一个 `lightSpaceMatrix`，把世界空间顶点位置转换为光空间。顶点着色器传递一个普通的经变换的世界空间顶点位置和光空间的顶点位置给像



素着色器。

像素着色器则加入了 shadowCalculation 函数来计算阴影。这个像素着色器还需要两个额外输入，一个是光空间的片元位置和第一个渲染阶段得到的深度贴图。

首先要检查一个片元是否在阴影中，把光空间片元位置转换为裁切空间的标准化设备坐标。当顶点着色器输出一个裁切空间顶点位置到 gl\_Position 时，OpenGL 自动进行一个透视除法，将裁切空间坐标的范围 -w 到 w 转为 -1 到 1，由于深度贴图深度值取值均为 [0, 1]，所以要进行进一步的变换将 projCoords 中的值变换到 [0, 1]。

有了这些投影坐标，我们就能从深度贴图中采样得到 0 到 1 的结果，从第一个渲染阶段的 projCoords 坐标直接对应于变换过的标准化坐标。最后，我们将光的位置视野下最近的深度 closestDepth 与片元当前深度（其等同于投影向量的 z 坐标）进行比较，如果 currentDepth 高于 closetDepth，那么片元就在阴影中。

## 5.3.2 阴影修正

### 修正 1：修正斑马纹

**问题：**由于阴影贴图受限于解析度，在距离光源比较远的情况下，多个片元会从同一个斜坡的深度纹理像素中采样，导致我们所得到的阴影就有了差异，出现严重的斑马纹效果。

尤其是当光源以一个角度照向多个表面的时候，这种问题会变得更加严重。

**方法：**我们可以用**阴影偏移** (shadow bias) 的技巧来解决这个问题，我们简单的对表面的深度（或深度贴图）应用一个偏移量，这样片元就不会被错误地认为在表面之下了。

### 修正 2：修正悬浮 (Peter panning) 阴影失真

**问题：**使用阴影偏移是对物体的实际深度应用了平移。偏移有可能足够大，以至于可以看出阴影相对实际物体位置的偏移，使得物体即使接触在另一个物体的表面上，依然看起来

像是悬浮的状态。


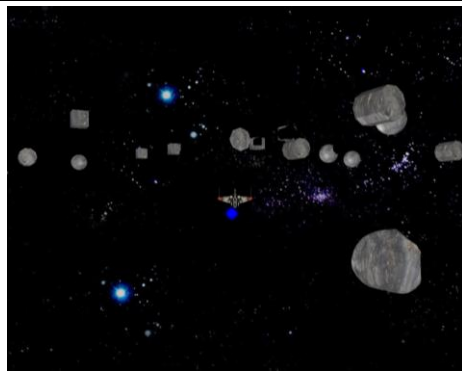
**方法：**OpenGL 默认是背面剔除，当渲染深度贴图时候使用正面剔除（front face culling）时即可解决大部分悬浮问题。

### 修正 3：修正阴影的锯齿边

**问题：**因为深度贴图有一个固定的解析度，多个片元对应于一个纹理像素。结果就是多个片元会从深度贴图的同一个深度值进行采样，这几个片元便得到的是同一个阴影，这就会产生锯齿边。

**方法：**PCF（percentage-closer filtering）是一种多个不同过滤方式的组合，它产生柔和阴影，使它们出现更少的锯齿块和硬边。核心思想是从深度贴图中多次采样，每一次采样的纹理坐标都稍有不同。每个独立的样本可能在也可能不再阴影中。所有的次生结果接着结合在一起，进行平均化，我们就得到了柔和阴影。

**阴影最终效果图：**

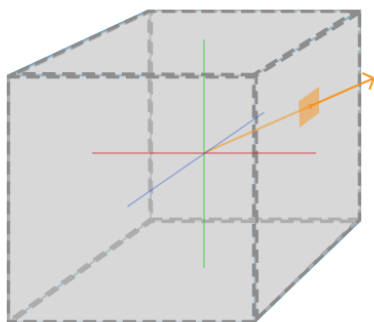
效果 1:	效果 2:
	

# Chapter 6 天空盒

## 6.1 天空盒简介

所谓“天空盒”，其本质上是一个“立方体贴图”。立方体贴图就是一个包含了 6 个 2D 纹理的纹理，每个 2D 纹理都组成了立方体的一个面：一个有纹理的立方体。你可能会奇怪，这样一个立方体有什么用途呢？为什么要把 6 张纹理合并到一张纹理中，而不是直接使用 6 个单独的纹理呢？立方体贴图有一个非常有用的特性，它可以通过一个方向向量来进行索引/采样。假设我们有一个  $1 \times 1 \times 1$  的单位立方体，方向向量的原点位于它的中心。

使用一个橘黄色的方向向量来从立方体贴图上采样一个纹理值会像是这样：



**方向向量的大小并不重要，只要提供了方向，OpenGL 就会获取方向向量（最终）所击中的纹素，并返回对应的采样纹理值。**

如果我们假设将这样的立方体贴图应用到一个立方体上，采样立方体贴图所使用的方向向量将和立方体（插值的）顶点位置非常相像。这样子，只要立方体的中心位于原点，我们就能使用立方体的实际位置向量来对立方体贴图进行采样了。接下来，我们可以将所有顶点的纹理坐标当做是立方体的顶点位置。最终得到的结果就是可以访问立方体贴图上正确面纹理的一个纹理坐标。

## 6.2 天空盒的创建

### 6.2.1 立方体贴图纹理

立方体贴图是和其它纹理一样的，所以如果想创建一个立方体贴图的话，我们需要生成一个纹理，并将其绑定到纹理目标上，之后再做其它的纹理操作。这次要绑定到

GL\_TEXTURE\_CUBE\_MAP:

因为立方体贴图包含有 6 个纹理，每个面一个，我们需要调用 `glTexImage2D` 函数 6 次，参数和之前教程中很类似。但这一次我们将纹理目标(target)参数设置为立方体贴图的一个特定的面，告诉 OpenGL 我们在对立方体贴图的哪一个面创建纹理。这就意味着我们需要对立方体贴图的每一个面都调用一次 `glTexImage2D`。

由于我们有 6 个面，OpenGL 给我们提供了 6 个特殊的纹理目标，专门对应立方体贴图的一个面，对立方体的左右上下前后。

### 6.2.2 加载天空盒

加载天空盒，我们实际上是加载 6 张图片，作为立方体贴图的纹理。

我们使用了 `#include <stb_image.h>` 这个图片加载的函数库，通过导入路径进行图片的加载。

```
1. vector<std::string> faces {"rt", "lf", "up", "dn", "ft", "bk"};
2. for (int i = 0; i != faces.size(); i++) { faces[i] = "skybox/purplenebula_"
    + faces[i] + ".jpg"; }
3. unsigned int cubemapTexture = loadCubemap(faces); //Load the skybox
```

### 6.2.3 显示天空盒

由于天空盒是绘制在一个立方体上的，和其它物体一样，我们需要另一个 VAO、VBO 以

及新的一组顶点。用于贴图 3D 立方体的立方体贴图可以使用立方体的位置作为纹理坐标来采样。当立方体处于原点(0, 0, 0)时，它的每一个位置向量都是从原点出发的方向向量。这个方向向量正是获取立方体上特定位置的纹理值所需要的。正是因为这个，我们只需要提供位置向量而不用纹理坐标了。

与此同时，如果你运行一下的话你就会发现出现了一些问题。我们希望天空盒是以玩家为中心的，这样不论玩家移动了多远，天空盒都不会变近，让玩家产生周围环境非常大的印象。所以我们添加如下语句：

```
glm::mat4 view = glm::mat4(glm::mat3(camera.GetViewMatrix()));
```

这将移除任何的位移，但保留旋转变换，让玩家仍然能够环顾场景

## Chapter 7 粒子系统

### 7.1 两个粒子系统的介绍

游戏中包含两个粒子系统：发射子弹的 Gun 粒子系统和飞机尾焰的 Fire 粒子系统。由于粒子系统涉及大量的粒子，会严重影响系统运行的性能。因此生命周期的管理和查找的优化变得十分重要。在控制生命周期时采用重置的方式。尾焰中的粒子消失后会重置在飞机发动机内继续向后喷射；子弹在飞行一段时间后从集合中 delete，直到左键触发才添加新的粒子进行绘制。

在查找方面，由于不可逆的时间上的先后逻辑关系，最新一个需要 delete 的粒子在数组中一定出现在刚刚消失粒子的下一个位置。因此每次只需要记录上一个粒子的下标并不断更新。这样可以几乎每次在  $O(1)$  的时间内找到对应的粒子。查询效率得到了飞速地提升。

## 7.2 粒子光晕——泛光

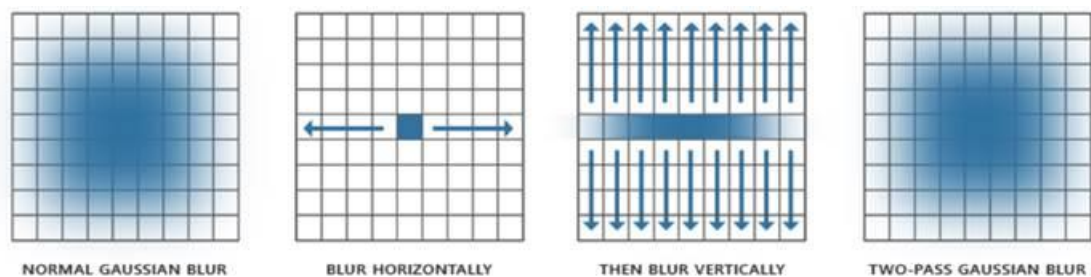
在粒子的绘制方面，采用小立方体模拟粒子。再配合帧缓冲的泛光技术完成光晕效果，从而使尾焰看起来更加的真实。

### 7.2.1 帧缓冲

帧缓冲是屏幕用于绘制的数据块，包含绘制每一帧所需的颜色等信息。默认的帧缓冲是设备屏幕，在这种情况下不容易对每一帧进行后期处理。我们的做法是使用自定义的帧缓冲，在每次绘制时将基本信息绘制到自定义的帧缓冲当中，然后对自定义的帧缓冲进行处理，最后在输出到屏幕上。

### 7.2.2 高斯模糊

对自定义的帧缓冲进行高斯模糊是反光中最重要的一步。对自定义帧缓冲拿到的数据的亮色部分进行高斯模糊，从而使图像中高亮的部分的轮廓看起来模糊



将模糊的帧处理与之前的基本帧相叠加，便可以看到发光物体周围的光晕，体现在粒子系统中就是闪闪发光的粒子及其光晕。

### 7.2.3 gamma 矫正和 HDR

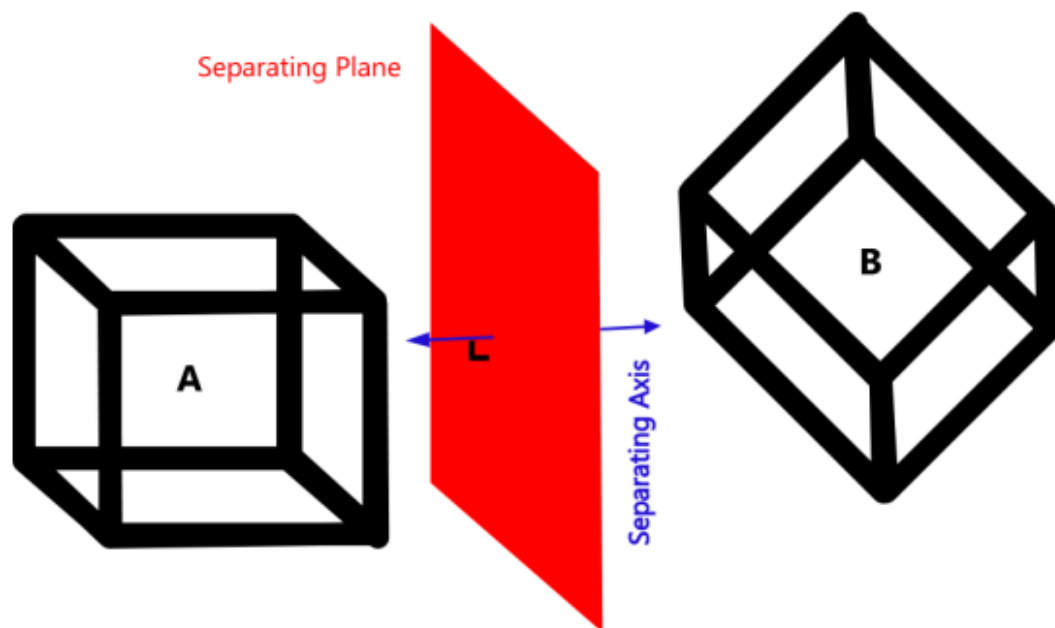
Gamma 矫正可以矫正因输出设备而造成的非线性的色彩偏差，使用 gamma 矫正则可以将其在线性空间进行编辑，这对实验中纹理和天空盒的加载以及火焰颜色的调整十分关键。

HDR 可以使得光照和色彩的强度不被限制在 0-1 之间，便于场景光线的协调。在片元着色其中对颜色信息进行色调映射后使得全部色彩信息被映射到 0-1 之间再进行输出。

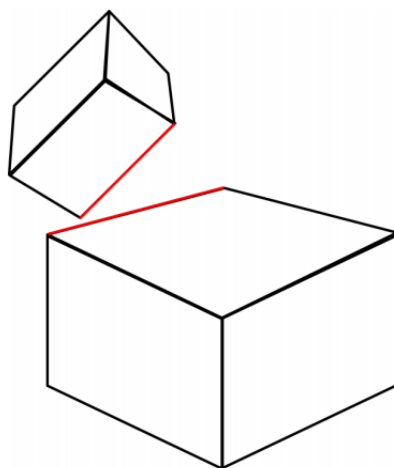
## Chapter 8 碰撞系统与爆炸特效

### 8.1 碰撞检测

由于我们的飞船的可以进行旋转的，因此传统的 AABB 包围盒不能够满足我们的需求，因此使用了基于 SAP 原理的 OBB（Oriented Bounding Box）。这种碰撞检测原理十分简单易懂，就是看是否能找到将两个包围盒分割开来的一个平面，如果能找到，说明这两个包围盒没有发生碰撞，否则发生了碰撞。



为了找到这样的平面，我们很容易想到从两个立方体的六对平行面入手，但是这并不适用于下图这种情况。



在这种情况下，分割面的法线刚好与来自两个立方体的各一个面的法线垂直，因此可能的分割面总共有 6（与立方体的面平行的面）+9（与两个立方体的面垂直的面）种可能。

体现到数学表达式上，就是

$$|T \cdot L| > |(W_A \cdot A_x) \cdot L| + |(H_A \cdot A_y) \cdot L| + |(D_A \cdot A_z) \cdot L| + |(W_B \cdot B_x) \cdot L| + |(H_B \cdot B_y) \cdot L| + |(D_B \cdot B_z) \cdot L|$$

T – 两个立方体中心坐标相减得到的法线

L – 分割面的法线

A/B – 进行碰撞检测的两个包围盒

W/H/D – 包围盒在这个方向上的半长

如果满足该条件，则说明在这个分割面 L 上物体没有发生碰撞。

但由于场景中的物体过多（光是物体的两两组合的时间复杂度就是  $O(n^2)$ ）每次检测又需要大量的点乘叉乘运算，因此我们在进行碰撞检测前，先计算了物体之间的距离，当物体之间的距离小于阈值之后，才进行基于分割面的碰撞检测。

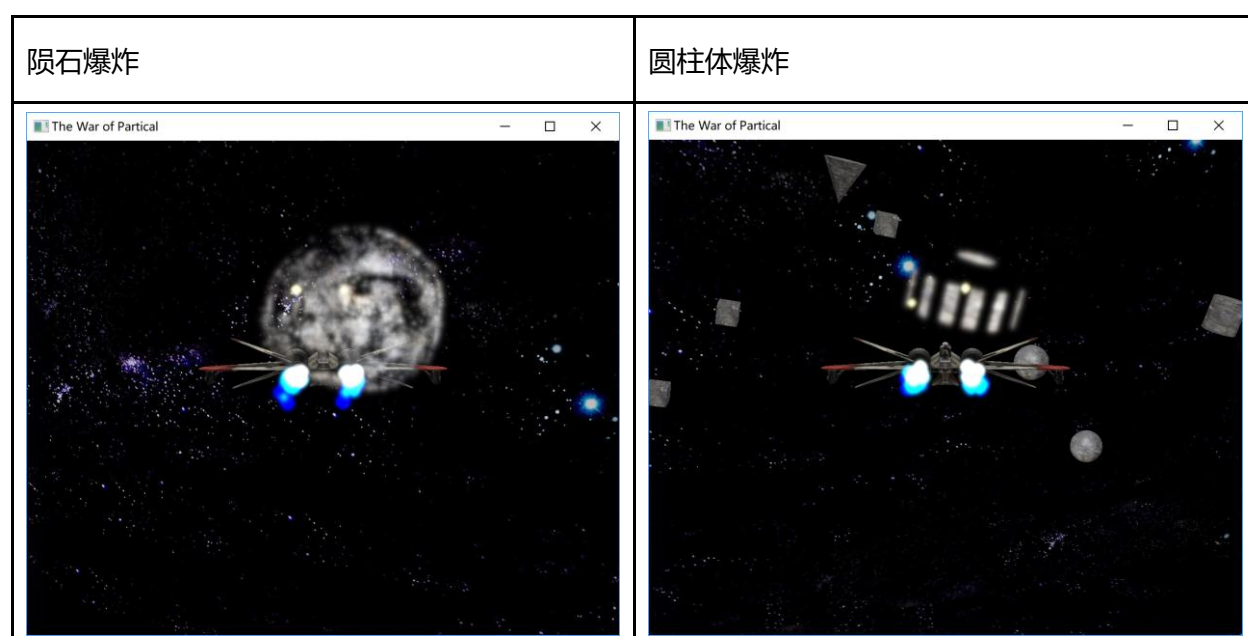
除此之外，对于飞船这种不是完全规则的物体，我们可以使用层次包围盒：即通过多个小的包围盒来提高包围精度，使得包围盒恰好包围住物体，但是这样导致的开销的也会相对增大，进行碰撞检测时也就增加了一定的计算量。



## 8.2 爆炸特效

为了让物体在消失时有一定的消失特效，我们加入了爆炸效果，其原理仍然是基于三角形的绘制。当物体要消失时，我们将物体按面各自沿着法线移动不一样的距离，这样就形成了一个物体远离中心散开的效果，同时我们也加入了之前粒子系统提到的光晕泛光效果，使得物体的破碎更加具有震撼力。

但由于物体的破碎是基于面的移动进行的，因此对于导入到 OBJ 能够效果较好，而完全规则的几何体则效果稍差一些。



# Chapter 9 动画播放与屏幕截取

## 9.1 动画播放

动画播放功能表现在飞船在撞击陨石时，在飞船爆炸后窗口会出现交替变换的 gameover 画面。具体的实现方法是利用 2D 纹理绘制的方法将纹理所需要的图片覆盖整

个屏幕。设置一个全局布尔变量，当飞船在场景中飞行时，该布尔量值为 false；当飞船撞击陨石以后，该布尔变量变为 true，关闭深度测试并显示 gameover 画面的纹理。

纹理切换的方法是是利用函数实时修改片段着色器中两个纹理的 MixValue 值，使得两张纹理图片的占比呈正弦函数变化，进而产生动画播放的效果：

```
float mixValue = 0;
mixValue= (sin(glFWGetTime()) / 2.0f) + 0.5f;
ourShader.setFloat("pro", mixValue);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture1);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, texture2);
ourShader.use();
glBindVertexArray(gameoverVAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

效果展示：



(窗口中会不断播放两种画面渐变转换的动画)

## 9.2 屏幕截取

屏幕截取功能是利用 glReadPixels()函数截取屏幕当前的实时像素，并将其存入数组当中：

```
unsigned char* data = new unsigned char[width*height*comp];

glPixelStorei(GL_PACK_ALIGNMENT, 1);
glReadPixels(0, 0, width, height, GL_RGB, GL_UNSIGNED_BYTE, &data[0]);
```

但此时如果直接将像素信息存入文件，得到的图片会是上下翻转的，因此需要将

该数组中的元素随着 y 方向上下翻转：

```
for (int i = 0; i < height; i++) {  
    for (int j = 0; j < width * 3; j++)  
        newdata[i*width*comp + j] = data[width * (height - i - 1) * comp + j];  
}
```

之后再利用 `std_image_write` 中写入不同图片格式的函数，保存为

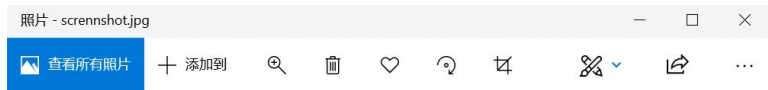
jpg/png/bmp 格式：

```
switch (type)  
{  
case PNG:  
    stbi_write_png(file.c_str(), width, height, comp, newdata, width * 4);  
    break;  
case JPG:  
    stbi_write_jpg(file.c_str(), width, height, comp, newdata, 100);  
    break;  
case BMP:  
    stbi_write_bmp(file.c_str(), width, height, comp, newdata);  
    break;  
default:  
    std::cout << "save texture failed" << std::endl;  
    break;  
}
```

这样，每次我们触发截屏按键，程序就会执行 `saveimage()` 函数进行截屏，并生

成对应图片格式的文件。

**截图效果展示：**



## Author List

几何体绘制、光源阴影编辑、动画播放与屏幕截取：吴思晗

天空盒、场景布置、游戏逻辑：柯宇

粒子系统、飞机编辑、视角编辑：高新宇

OBJ 文件的导入导出，MTL 文件的导入、碰撞检测、爆炸特效：钟子鹏

## References

[1] LearnOpenGL CN <https://learnopengl-cn.github.io/>

[2] 多边形碰撞 -- SAT 方法 <https://www.cnblogs.com/programmer-kaima/p/5195781.html>