# 计算机图形学大作业报告

冰川与火鸟

515030910309 张然

目录

# 一、项目简介

## 1. 功能与操作简介

本项目搭建了一个冰川场景。

场景中包括冰面、天空。拥有白天、夜晚、黄昏/黎明三个时间段，雨、雪、暴风、雾等天气。

场景中有一只火鸟，以及一些散落在地图上的火焰。火鸟接近蓝色的火焰，可将其引爆，之后火焰变为红色，然后在一段时间后变回蓝色。

使用者可以控制火鸟的移动、视角的转换、时间变化快慢、天气切换等。

使用鼠标控制视角，滚轮放大和缩小。

使用 WS 控制火鸟沿当前视角方向前进，AD 控制火鸟转向。

使用数字键切换天气：

1：晴天

2：雨天

3：无风雪天

4：暴风雪

使用+、-改变时间变化速度

（由于以上均为粘滞键，按键有事需要长按才能生效）

## 2. 代码简介

使用了 OpenGL 的核心模式，并利用了一些外部的库与代码。与头文件同名的 vs 文件为模型对应的顶点着色器，fs 为片段着色器。

CG.cpp：主函数，包括了创建窗口，创建、绘制场景中的所有物体，控制系统风力，接受用户输入，控制天气等功能。

Camera.h：响应用户鼠标的滚轮和移动事件，转换视角。同时根据绑定的物体的运动状态修改相机的位置与方向。

Character.h：响应用户的键盘 WASD 输入，改变角色的运动状态。同时控制角色（火鸟）的翅膀挥动、粒子效果等。

Fire.h：控制场景中的火苗的属性。检测与主角的距离，控制火苗爆炸。

Floor.h：创建冰川地形。

Fog.h：控制雾的范围与变化。

Light.h：控制太阳的属性、移动。

Loader.h：读入模型。

Mesh.h：将读入的模型与 VAO、VBO 绑定，加载纹理。

Particle.h：粒子生成器，根据输入的各种属性生成粒子。

Rain.h / Snow.h：控制雨、雪粒子系统。

Sky.h：控制天空的属性。

Shader_m.h：加载着色器。

Stb_image.h：读入图片作为纹理。

Glad.c：在运行时查找 opengl 函数的具体位置。

最后三个文件为直接从外部导入，未作修改。此外，还使用了 glfw、glm 等库（详见“外部辅助与库”）。

## 3. 与作业要求的对应

（1）场景建模及渲染，场景添加光照，光照可交互控制（20 分，按照场景复杂度、渲染效果等评分）

场景使用 blender 建模。尝试外部模型导入，但效果不好于是放弃。渲染有太阳与天空的渲染、雾效的渲染等。

场景中太阳、火焰、火鸟均有光照。

光照的交互控制：可控制时间变化快慢、与火焰交互可以改变火焰光照的强度、颜色。

（2）设计实现粒子系统特效，粒子运动有物理仿真，可实时切换粒子系统中的粒子三维模型（30 分，按照粒子系统的特效复杂度、计算模型、算法效率等评分）

粒子系统有雨、雪、火焰、火鸟翅膀。

雨、雪、火焰与场景中的风有物理关系，粒子的位置、速度、加速度均使用了物理学规律。

可在控制天气转换的时候，实时切换粒子系统中的粒子三维模型。

（3）粒子的三维模型的光照、纹理映射（20 分，按照粒子的视觉外观实现方法的复杂度、实现效果评分）

粒子使用三维模型，雪粒子与场景中光源有光照计算。火粒子考虑到计算效率，在生成器的位置设置了光源营造例子发光的效果。

火粒子、雪粒子均有纹理映射。

（4）设计实现粒子系统的交互控制（20 分，根据交互方式的新颖、自然和交互响应评分）

使用者可以控制火鸟的移动、视角的转换、时间变化快慢、天气切换等。

# 二、场景实现

## 1. 冰川

冰川使用了 Blender 手动绘制的模型，并添加纹理坐标。然后导入场景。

因为希望场景比较大，起初使用了非常大的模型进行建模与导入，但加载时间非常长。于是采用了小模型建好模，平滑之后放大的方法，效果比较好。

由于地面有起伏会出现穿模现象。起初的办法是利用类似 AABB 包围盒的方法。在生成地形图的时候，由于模型是放大之后的，200x200 大小的矩阵只有几千的点拥有高度信息。

于是对每个点进行广度优先，将高度值赋给周围坐标。但广度优先赋值高度的做法误差很大。也考虑了使用样条曲面进行恢复与计算，但是难度较高，没有实现。

当角色前进方向上的位置高度发生变化时，同时根据地形图改变角色的高度。但角色会发生“瞬移”的问题。于是采用了使角色每次 update 修改一点高度值的方法。但是角色在静止时可能发生“颤抖”。

最后，由于时间紧张，没有比较好的生成地形图的方法，只能采用将角色固定在比较高的位置的最简单办法。

## 2. 太阳与时间变化

场景光照会随着时间变化。一共有日出（日落）、白天、夜晚三种效果。在白天会有太阳，在晚上由于月亮的光晕效果的实现比较困难，因此没有制作月亮。

算法涉及 light、sky、floor 三个对象。Light 决定了光源（太阳）的位置、颜色，以及他们的运动方式。Floor 指雪地的模型。Sky 代表着天空。

（1）参数调整

在日出、白天、夜晚修改上述三个对象的参数，即可营造出不同的效果。

在 Light 对象中需要修改光源的位置（与 xoz 平面的夹角）和颜色。位置改变了光源的位置，颜色会影响天空与地面的颜色。光源还有衰减系数的属性，但并不需要修改。

在 Sky 对象中需要修改天空的颜色。之所以天空颜色不能保持一定，是因为在白天，太阳光为白色的，天空需要呈现蓝色，即自己本身的颜色。而在日出的时候，太阳光为橙色，如果天空依然是蓝色，在计算的时候颜色混合呈现的颜色会改变。

在 Sky 对象中还需要在光源的衰减系数的基础上进行补足，因为 Sky 与 floor 的衰减系数是不同的，因此需要分开修改。Sky 衰减系数较弱，为了尽量保证在白天的时候天空都是一致的颜色。但是在雪地上并不能用这样的算法。

此外，还需要修改镜面反射系数、高光区半径、按照不同的 RGB 比例修改光强，均是在调整过程中为了尽量接近真实。

在 Floor 对象中需要进行与 Sky 中类似的参数修改。此外，光源在 Sky 对象上使用的是点光源。在 floor 上为了接近真实，使用的是平行光。而在夜晚没有光源的时候，为了营造雪地发光的效果，将光源沿原点旋转 180 度，然后使用衰减系数较高的点光源，就可以实现了。

当然，三种状态不能瞬变。为了实现渐变效果，在日出、日落前后各 30 度的区间内进行了线性插值。即：

$$V = (1 - \alpha) * V1 + \alpha * V0$$

这个算法在渐变过程中效果比较差，但实现非常简单，由于同时进行插值的参数非常多，使用复杂的模型也很难会有更好的效果，因此就此为止。

（2）光照计算：

Sky 和 Floor 都使用了 Phong 光照模型，即：

$$I = I_{pa}k_a + \sum(I_{pd}k_d \cos i + I_{ps}k_s \cos^n \theta)$$

综合考虑环境光、漫反射、镜面反射、纹理或物体颜色得出结果。此外，为了营造雪地上较好的镜面反射效果，在一定条件下，使用了 Blinn-Phong 的光照模型。对比如下：

Blinn-Phong

Phong

此外，还进行了光照的衰减运算：

$$F_{att} = \frac{1.0}{K_c + K_l * d + K_q * d^2}$$

常数项始终为 1，一次项和二次项会根据 Sky 和 Floor 的不同进行调整。

## 3. 天空

天空使用了 Blender 建模的半球体。

为了实现白天耀眼的太阳的蓝色的天空的效果，天空使用了蓝色。而太阳的实现并不是一个具体的太阳物体，而是点光源在他之后的 Sky 幕布上进行了镜面反射，达到了相似的效果。这个算法需要大量的参数调整与试验，可能出现天空颜色不一致，地面太亮，太阳太大或大小等诸多问题。此算法的实现，一方面由于用镜面反射实现，耀眼的效果比较真实，且有光晕的效果。另一方面，先对于使用高斯模糊，纹理叠加的泛光算法，计算量小。

最初使用的是 Skybox。但是，固定的 Skybox 随着时间变化实现的日夜效果很差。由于纹理是一定的，并且都是平面，当光源在场景中移动体现的效果很假。如下是使用夜晚效果的 skybox 与在白天的雪地（由于距离较远没有出现镜面反射相对较暗）同时出现的场景：

Sky 起初用的球体模型有 64 条纬线和 32 条经线，在显示的时候会看到很明显的片面棱角，于是进一步提高精度实现效果。此外，由于建模导入的球体法向量向外，因此在导入的时候要将法向量反向，保证正确的光照计算。

## 4. 雾效

为了实现雨、雪时阴天、大雾的效果，实现了雾效。

实现方法是，设定好能见的最大范围、最小范围、雾的颜色。近处的景象完全呈现，超出最大范围的完全使用雾的颜色。然后再片断着色器中将计算的最终结果与雾的颜色按照距离比例进行混合。注意距离是相对于视角的，不然会发生雾固定在一定范围的问题。

## 5. 风

在系统中，使用了一个记录风产生的加速度的变量。在计算粒子运动状况的时候，加入计算。在角色运动的时候，每次运动都会向着风的加速度的方向移动一段距离，制造出了逆风减速，顺风加速的效果。

在角色静止的时候，不会受到风的作用。一方面静止状态效果比较好，另一方面在角色不受用户输入的情况下移动，相机的跟踪会出现奇怪的现象，为了节约时间只能放弃。

起初使用了储存 glm::vec3 的三维数组，储存了不同的风的信息。但是十分占用空间的同时，风的偏差较大效果会与完全随机运动类似，较小的时候因为粒子本身已经拥有不同的速度，与所有加速度都相同差别很小。此外，为了节省空间，风也只在有粒子的范围内产生，所有粒子要将浮点数坐标值经过计算，映射到风的数组的某一个点，计算量很大，所以放弃。

# 三、粒子系统

## 1. 粒子生成器

创建粒子生成器需要以下参数：

粒子的 shader 与 mesh。

粒子系统的初始粒子数目、最大粒子数目、每次 update 复活的粒子数目（grow）及其改变的速度。

粒子生成器的生成的位置及其范围、速度及其范围、角度及其范围、初始颜色及其范围、结束颜色、大小及其范围、旋转范围与旋转轴。粒子变化基本采用了随机分布，少数采用了高斯分布。

粒子采用了实例化。初始化时，将所有粒子都初始化，push 进 vector，将初始粒子数目的例子的寿命赋值。然后每次绘制时，将一定数目（grow）、寿命小于零的粒子复活，即重新赋寿命，然后计算其他需要更新的属性。然后计算并绘制所有寿命大于零的例子。他们的运动状态根据风力算出加速度，然后修改速度，最终计算出新的位置。最后修改寿命。

如果粒子 grow 改变的速度不为零，则在初始的时候 grow 设置为 0，然后逐步增加。

粒子有在与坐标轴平行的矩形平面（或立方体）生成和任意方向矩形生成两种模式。与坐标轴平行实现，只需要在粒子生成器的位置生成之后，增加 xyz 三个方向的随机量。任意方向的矩形，需要宽度、深度两个向量，然后在粒子生成器的位置生成之后，分别在两个方向用随机量偏移，且 xyz 的随机量需要相同。

## 2. 雪

雪粒子使用了球体模型，贴上了雪的纹理。根据暴风雪和无风时的雪设置不同的参数。

雪的生成器会随着角色移动，否则全图计算计算量比较大。

一开始的想法是，为了降低计算量，同时保证视野内有尽可能多的粒子，进行了优化

A.将相机位置始终放在粒子生成器的中心，但是实际视野只占了粒子系统的四分之一。并且，在沿坐标轴方向和角分线方向上看到的范围大小也是不同的。

B.根据相机的位置和方向，平移并旋转矩形，是相机视野占据粒子系统约四分之三，不能完全重合的原因是，在视野变换过程中可能出现明显的例子分布不均的现象。同时，这个方法也保证了可以缩小矩形面积，增大粒子密度，而粒子总数不变。

C.为了便于计算，使结果更加真实，将矩形替换为扇形。具体的计算方法是，在视角右侧边界上生成点，然后随机旋转（0，90）度。

但是实际在视角转换的过程中分布不均的现象依旧比较明显，于是放弃。将相机固定在了粒子生成器中心。

此外，局部下雪可能会“穿帮”。所以在扩大生成器的范围与密度的同时，需要增大雪的大小范围，创建出“近大远小”的视觉效果。

在制作暴风雪的时候，若像真实情况一样，从天空高出下雪，会因为“局部下雪”的原因，粒子被风吹走，角色周围没有雪。根据风的位置与风力动态改变生成器与角色的相对位置非常麻烦。于是将暴风雪的粒子生成器制作成了与 y 轴垂直的平面，然后拥有较大的 xoz 平面上的初速度，营造出暴雪的感觉。

# 3. 雨

在调节效果的时候，遇到了参数相互影响，制约的问题。计算间隔越短，或速度越小，寿命需要越长。在移动过程中要保证不出现粒子分布不均，需要降低生成器的高度，并减少寿命。

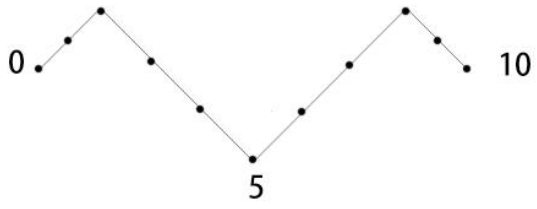
在粒子比较小的时候，即使设定很多粒子，场景中还是很少。于是放大了粒子模型的大小，发现现象有所缓解。但是在 fs 文件中修改和在 glm::scale 函数中修改，看到的粒子数目仍然不一样多，可能的原因是在 fs 中修改，计算较快，于是粒子显示较多。

粒子显示会有闪烁、虚线等问题。于是通过不断修改粒子模型，由球体，变为针状，最后修改为上窄下粗的形状，发现问题仍然不能很好解决。最终为雨的例子加入了 0.1 的透明度，比较接近真实。

透明度这里，Blend 使用了最常用的 GL_SRC_ALPHA 和 GL_ONE_MINUS_SRC_ALPHA。在之后会有更详细的说明。

# 4. 火鸟

火鸟由 11 个粒子系统组成，编号为 0~10。他们的坐标都是相对于角色的位置的，初始状态与 xoy 平面上。




（1）翅膀挥动

0~4 为左侧翅膀，6~10 为右侧翅膀，5 作为头部。

根据预先设定好的“翅根”、“翅尖”的角度范围，线性插值修改角度，然后计算这些点的位置。

之后，用相邻的点组成的向量作为生成范围的“宽度”更新粒子系统。用位置加上偏移量更新粒子系统的生成位置。如火鸟处在运动状态，则用沿着方向的反方向、长度为 0.3 的向量作为“深度”，否则用长度为 0.15 的向量。之所以需要深度，是因为如果永远都在一条线段上生成的话，在火鸟运动的时候会出现明显的断层现象。而区分运动状态和静止状态是因为，在静止状态深度太大的话，翅膀会变成矩形。

5 的寿命和随机量比较大，作为身体。2、8 粒子系统平均寿命较长，其余渐次缩短。为

了模糊阶梯状的翅膀，增大了 x、z 方向上速度的随机量。

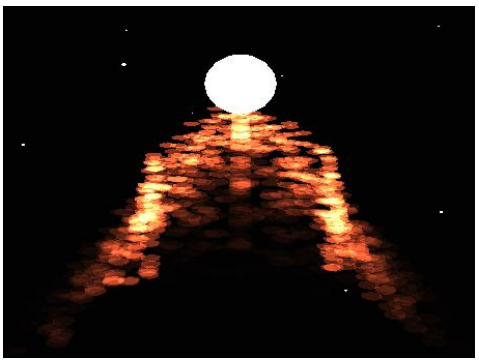
（2）火鸟运动

在火鸟运动的过程中，方向会发生变化。此时粒子系统的位置仍按照初始平面计算，之后根据火鸟方向与 z 轴的夹角旋转平面，使翅膀始终与前进方向垂直。

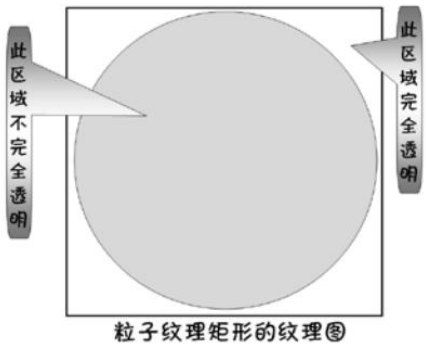
具体的计算方法为：计算 z 轴单位向量与前进方向（始终为单位向量）内积得到 cos θ，然后计算出 sin θ，为了防止翅膀绕中心旋转，需要根据具体的方向，判断三角函数的正负对应情况。

（3）火鸟粒子与绘制

翅膀的粒子模型也尝试了 2D 面片，在面对火鸟的时候效果与 3D 类似，但是从侧面看的时候，面片效果非常明显。如果将所有例子实时按照视角旋转计算量也很大，于是选择了 3D 模型。下图为 2D 面片效果：



使用的纹理由于中心是火焰，四周是黑的，所以在绘制的时候判断，如果纹理坐标在黑色区域就将透明度设为 0。此外，所有的粒子系统必须使用不同的 mesh，但是他们的纹理是相同的，重复载入会浪费空间、时间，因此将第一个载入后后面的直接将 texture 赋值即可。



粒子纹理矩形的纹理图

在混合（Blend）绘制的时候，源因子使用了 GL_SRC_ALPHA（因子等于源的 alpha 值），目标因子使用了 GL_ONE（目标因子为 1），由目标贡献全部的 alpha 比例的颜色值，再加上源按照 alpha 值贡献的颜色值，叠加下会使粒子发亮，将深红色的粒子绘制出了火焰的效果。但是，在白天的时候，由于场景较亮，粒子会变成白色，目前没有很好的解决办法。

$$\bar{C}_{result} = \bar{C}_{source} * F_{source} + \bar{C}_{destination} * F_{destination}$$

- $\bar{C}_{source}$：源颜色向量。这是源自纹理的颜色向量。
- $\bar{C}_{destination}$：目标颜色向量。这是当前储存在颜色缓冲中的颜色向量。
- $F_{source}$：源因子值。指定了alpha值对源颜色的影响。
- $F_{destination}$：目标因子值。指定了alpha值对目标颜色的影响。

（4）火鸟模型

火鸟初始的设计是，在鸟模型的基础上，加上上述的翅膀。

但是实际操作起来，找到合适的鸟模型很困难，在没有骨架的基础上实现运动的效果也很难，于是没有这样做。

## 5. 火

火焰粒子有红、蓝两种配色，具体的系统使用了与火鸟翅膀相似的属性，不做赘述。

（1）火焰的爆炸

在火鸟距离与火焰小于一定程度，且火焰为蓝色时，启动爆炸。爆炸会将粒子系统的速度变大，速度范围变为以位置为圆心的立方体，将随机模式变为高斯分布而不是随机分布（随机分布的爆炸为很明显的立方体而不是球体），粒子系统 grow 设为 0，所有寿命小于零的粒子更新寿命。

之后，每次 update 会检查粒子系统爆炸是否已经结束（因为 grow 为 0，不会有新粒子产生，主需要检查所有粒子的寿命都小于零即可）。若已结束，则复原粒子系统属性，仅将颜色修改为红色。

（2）火焰状态的改变

火焰每次绘制会将更新燃烧时间，若大于阈值，则修改颜色变为蓝色。

（3）多光源（与火鸟的翅膀相同）

在 floor、snow 两个需要进行多光源运算的物体的片断着色器中，加入点光源数组。点光源数组拥有位置、颜色、衰减系数、环境光（ambient）、漫反射（diffuse）、镜面反射（specular）向量等属性。然后按照与单个光源相同的计算方法即可得到多光源下的效果。

如果在白天，雪会使用 1.0 的环境光，如果是在晚上，会减弱环境光，加入光源计算，使光源附近的雪会被照亮。此外，光源在雪上的衰减会减弱，环境光和漫反射会加强，尽量避免雪粒子明显的阴影效果。

一个光源对它的 ambient、diffuse 和 specular 光照有着不同的强度。环境光照通常会设置为一个比较低的强度，因为我们不希望环境光颜色太过显眼。光源的漫反射分量通常设置为光所具有的颜色，通常是一个比较明亮的白色。镜面光分量通常会保持为 vec3(1.0)，以最大强度发光。

火的位置会在中心附近做高斯随机，从阴影看出火在抖动的效果。

# 四、交互控制

## 1. 相机

相机拥有如下功能：

响应鼠标滑动事件，修改视角；

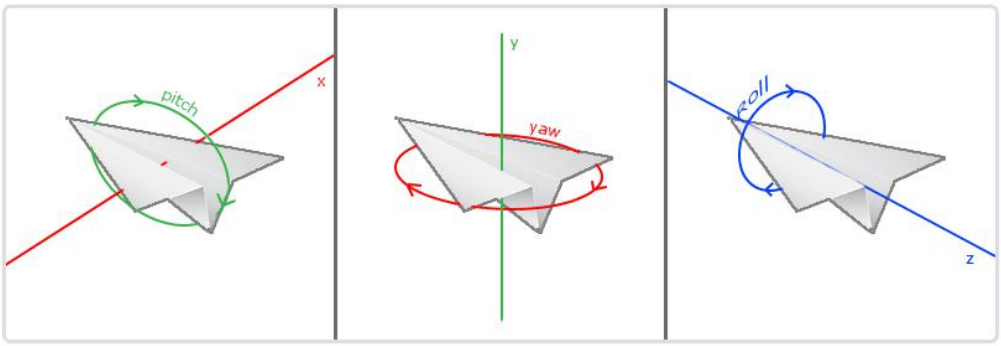
响应滚轮事件，修改范围；

根据绑定的角色的运动状态修改位置与角度（具体见下一节）。

初始化的时候放在与绑定的角色有一定距离的位置，并且面向角色。计算此时的角度。

在获得 viewMatrix 的时候，返回 vec3(position, position+front, up)。在获得 projectionMatrix 的时候，需要获得他的缩放比（Zoom）。更多相机的控制需要结合角色，参见下一节。

在算初始位置以及相机的旋转时，要理解三个角度：




在本项目中，只涉及到 pitch 和 yaw 两个角度，特别注意的是他们的初始值。其中 pitch 沿 xoz 平面为 0，yaw 沿 x 轴正方向值为 0。

# 2. 角色

角色控制方式有：

移动鼠标，镜头上下自由系统，左右绕角色旋转

键盘 W、S 按键使角色沿当前相机方向前进

键盘 A、D 按键使角色左右旋转，同时镜头绕角色旋转相同角度

（1）算法：

角色和相机都有位置、方向（指相机的 front 方向）两个属性。

移动鼠标时，记录鼠标 xoffset、yoffset，乘以灵敏度 MouseSensitivity，计算相机方向，将相机沿角色向相机反方向移动一段距离，并修改相机方向使他始终在 xoz 平面上指向物体。

使用键盘 W、S 按键，首先计算物体当前前进方向与相机方向的夹角，使物体旋转相应角度，沿相机方向前进后退。然后将新的位置传给相机，然后更新相机状态。

将直线旋转 θ 角度使用到的公式：

$$x = x0 * \cos(\theta) - y0 * \sin(\theta);$$

$$y = x0 * \sin(\theta) + y0 * \cos(\theta);$$

使用键盘 A、D 按键，首先使物体旋转相应角度，然后更新相机状态。

更新相机状态：每次物体运动后，将相机沿角色向相机反方向移动一段距离，并修改相机方向使他始终在 xoz 平面上指向物体。

（2）注意事项

由于相机是三维坐标(x0, y0, z0)，而物体前进方向是二维坐标(x1,0,z1)，并且都进行了归一化，所以在相互转换的时候需要进行转换。

使用的公式为：

二维转三维：

$$x0 = x1 * \sqrt{1 - y0^2};$$

```
z0 = z1 * sqrt(1 - y0^2);
```

三维转二维：

```
angle = arctan(z0 / x0);
x1 = cos(angle);
z1 = sin(angle);
```

在计算角度的时候，经常用到了 arctan，但是由于函数值域仅限于（-π/2，π/2），而实际上则需要 2π的范围，因此需要根据两直线所在的象限，在计算出 arctan 后进一步处理。

在修改 front 之后，需要修改 up 和 right，以防止错误。

```
if (x1 >= 0 && x0 <= 0)angle+= pi;
if (x1 < 0 && x0 > 0)angle+= pi;
```

一开始在 A、D 控制角色旋转带动相机的时候，运行函数使相机始终面对角色，发现计算量比较大，而且角度转换十分繁琐，并且相机较角色存在滞后现象。于是改为有一个共有的旋转角速度值，两个函数共同使用，减少的计算量与计算难度。

起初物体在相对光源运动的时候光照是固定的，可能的原因的光源与法线的关系发生了变化，但是法线没有改变。因此用法线矩阵

```
Normal = mat3(transpose(inverse(model))) * aNormal;
```

代替原来的发现 aNormal 发现即可以解决问题。

# 3. 其余交互的实现

（1）天气系统的切换

修改是否有雾需要改变环境雾的最大和最小范围，若范围大于天空，则没有雾。

系统中雨、雪、暴雪三个天气是同时存在的，修改他们的 grow 值可以修改开始和停止状态。同时，为了过度比较平滑，初始值都设置为了 0。

为了减少计算量，使用过只绘制指定天气的方法，但是切换十分生硬。

控制风只需要修改风的值就可以了。

（2）时间系统的加减速

时间系统是通过每次 update 将太阳旋转一定角度实现的，因此只需要修改每次改变的角度大小即可实现。但是，由于浮点数的不确定，发现角度不能减小到 0，因此在小于一定阈值的时候，会把角度强行设置为 0。

同时，由于获取的是粘滞键，所有此类操作都做成了幂等操作。设置过按键修改时间系统开始停止切换的功能，但是按一次按键，函数会被执行多次，所以放弃。

此外，glfw 的 release 是在不按下的时候就会触发，而不是松开按键的时候触发，也产生了一些不便利。

# 五、辅助功能

## 1. Loader.h

使用了第一次作业的模型加载，只是为了适应 VAO 和 VBO 的要求，进行了一些修改。

打开 obj 文件后调用 loader 解析文件生成物体。v 生成顶点，vn 生成顶点法线，vt 生成贴图坐标点。f 生成面和面的法向量，在 blender 生成模型的时候，选择了只有三角面片，并输出纹理坐标，因此只需要处理此类情况即可。解析成功后，将最终结果输入到 VBO 中等待绘制。使用了结构 vector<vertex(position, normal, texture)>，在 C++中，vector 以及结构内部都是连续的，因此可以直接将 vector 传到 VBO 中即可有正确的结果。

由于建模工具不太熟悉，物体的方向、法线方向与实际需求有些差距。为了简单，在读入的时候进行了一些处理。

## 2. Mesh.h

mesh 中储存了模型的顶点信息，及其 VAO、VBO、纹理的 index。定点信息按照位置、法向量、纹理坐标的顺序写入 VBO。

## 3. Shader_m.h

它可以从硬盘读取着色器，然后编译并链接它们，并对它们进行错误检测。

# 六、外部辅助与库

## 1. 核心模式，GLFW 与 GLAD

在本项目中，使用了核心模式。在写 fs、vs 的时候，的确感受到，虽然复杂度上升了，从输入，vs，fs，输出的流程中有很多需要新学习的东西，但是的确拥有了更高的自由度。例如在这个基础上，光照模型需要自己实现，但也可以比较自由的修改。

GLFW 是一个专门针对 OpenGL 的 C 语言库，它提供了一些渲染物体所需的最低限度的接口。

因为 OpenGL 只是一个标准/规范，具体的实现是由驱动开发商针对特定显卡实现的。由于 OpenGL 驱动版本众多，它大多数函数的位置都无法在编译时确定下来，需要在运行时查询。GLAD 是一个开源的库，它能解决上面提到的那个繁琐的问题。（引自 learnopengl）

## 2. GLM

本项目中使用了 glm 进行向量、矩阵的运算。在上一次作业中，使用的是自定义的结构。但是在本次项目中，因为有很多矩阵、向量运算，而 glm 已经提供了完善的接口，因此直接使用了。

Glfw、glm、glad 均在附带的库中。

## 3. Learn OpenGL

由于第一次使用核心模式制作 CG，很多地方难以入手，因此在原理、代码等方面参考

了 learnOpenGL，特别是着色器语言、窗口创建等部分。但是除了 shader 之外全部都进行了修改或重新撰写。

例如相机部分，原来的相机只能前进、后退、绕自己旋转。但是本项目中将角色与相机绑定，通过很多计算提高交互体验，相机行为发生了很大改变。（已在上文叙述）

## 4. Blender

上文提到，由于很难找到自己需要的模型，因此使用 blender 建了一些比较简单的模型。也趁此机会入门了建模软件。使用了纹理模式、雕刻模式等功能。

模型比较粗糙，也请多担待。

## 5. Stb_image

使用该库读入图片，用于之后的纹理使用。

## 6. Mmsystem.h

使用该库播放音效。但是因为不能同时播放多个文件，音效循环头尾链接很明显等诸多因素，最终项目并没有音效。

# 七、项目感想

本次项目收获非常多。除了了解了现代 opengl、着色器的写法，也将课上学习过的知识进行了运用，例如光照计算模型、视景体、粒子系统等等。

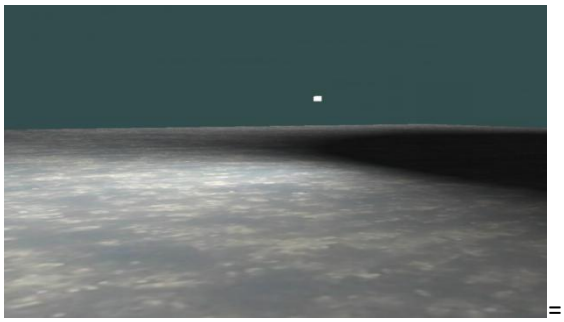
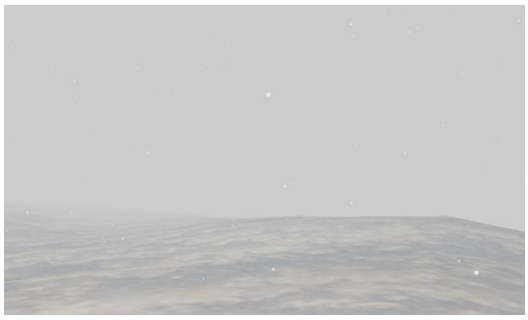
除了功能的实现，为了达到更好的效果，在调整参数上花费了非常多的时间。每一个模型的调整、粒子系统的属性、光照的属性、角色的运动等等都进行了大量的常识，虽然最终效果仍然不如人意，但是已经较功能刚实现的时候好了许多。

在本次项目中，也切身体会到了不能只着眼于代码，无论是建模、配色、视觉效果，还是最终没有实现的音效，都对结果有很大的影响，希望自己在未来的学习中，在学习技能的同时可以提高这些方面的技能，制作出更好的作品。

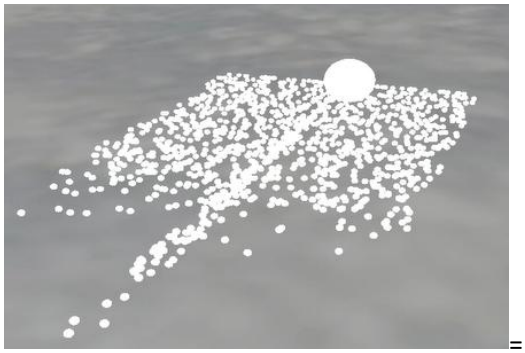
谢谢您的阅读。

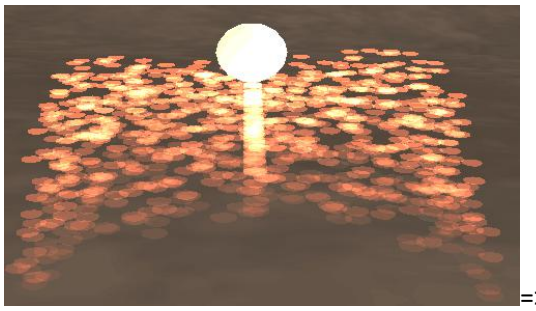
一些凑字数的对比：

场景

 =》 

角色

 =》  =》

 =>