

计算机图形学期末项目报告

第四组

组长	组员1	组员1	组员1	组员1
苏瀚添	宋志强	宋孝天	夏显卓	冼圣杰

项目介绍与效果

项目为赛车场景的建立和模拟，实现了基本场景的建立和车的运动。
开始时候的场景，在倒计时结束之前，车不能加速，前面存在路障。



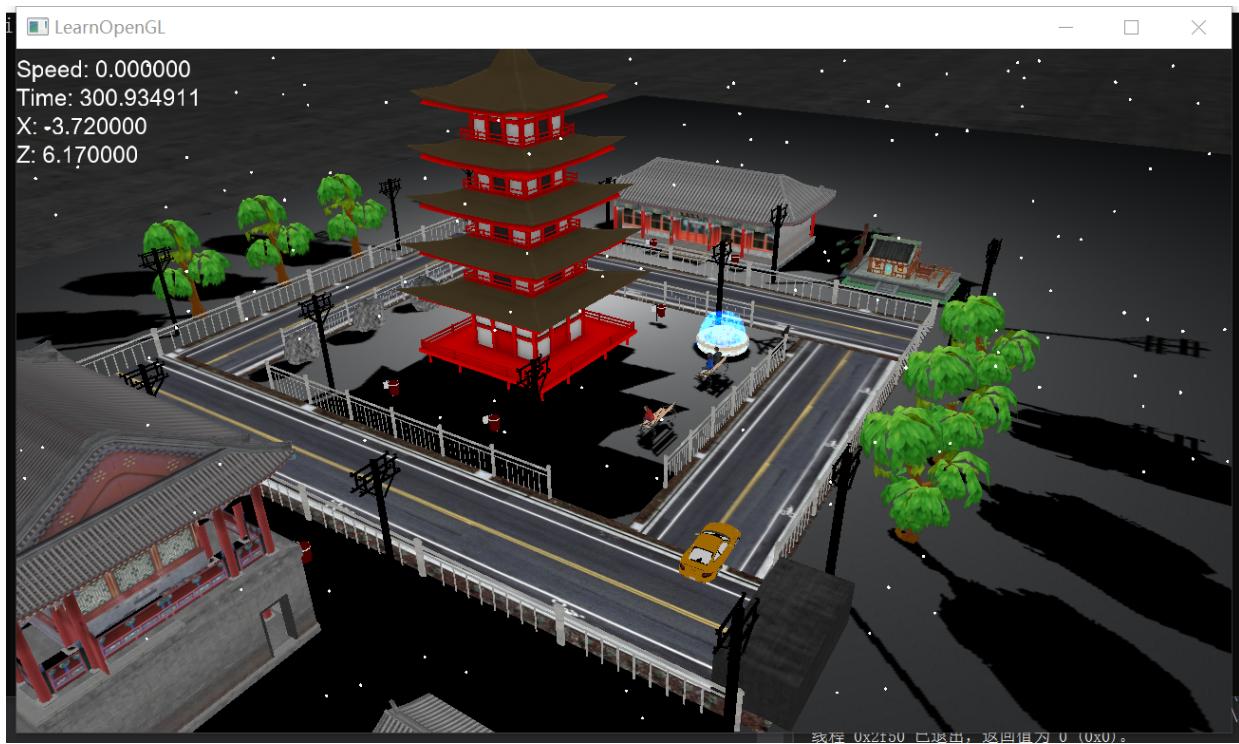
在最后两秒的时候，路障开始展现“爆破效果”，然后路障消失，汽车可以向前加速。另外，使用 wasd 可以控制汽车的运动。



可以看到，场景中存在粒子效果实现的动态喷泉和场景中正在飘雪，左上角可以查看到文本渲染的相关信息。



按space键，可以从对汽车的控制中解放，转换到对摄像头的控制，方便查看整个场景，可以清楚地看到利用blender构建的模型的静态阴影。



开发环境与第三方库

开发环境为window+vs2015+openGL3.3, 用到的第三方库有：glew, glfw3, glm, freetype, assimp。

实现功能列表

Basic	Bonus
Camera Roaming	Sky Box
Simple lighting and shading(phong)	Display Text
Texture mapping	Collision Detection
Shadow mapping	Particle System
Model import & Mesh viewing (Assimp)	Explosion Effect

另外，我们还实现了车的运动系统。

实现功能介绍

模型部分说明

场景的构建需要模型和场景处理，主要分为如下两个阶段。

一、场景建模

这次场景的构建所用的大多数复杂模型来自：<https://archive3d.net>，在这个免费模型网站找到合适的模型之后用blender导入模型，并控制模型的整体布局，对于一些简单模型（路，石头等）使用blender自己动手建模，以达到优化模型大小的目的。

二、静态阴影构建

模型阴影构建是在小组展示之后完成的，主要思路是通过blender获取模型的阴影贴图，将这个贴图和模型进行组合，实现模型静态投影的效果。

Sky Box

这部分主要用到立方体贴图的技术，创建一个包含了整个场景的（大）立方体，它包含周围环境的6个图像。主要包含下列步骤：

1. 加载6个天空盒的纹理贴图
2. 将纹理贴图映射到一个大立方体中
3. 将立方体渲染

具体代码实现在代码文件中体现

文字渲染

由于OpenGL本身并没有包含任何的文本处理能力，我们必须自己定义一套全新的系统让OpenGL绘制文本到屏幕上。

FreeType

FreeType是一个能够用于加载字体并将他们渲染到位图以及提供多种字体相关的操作的软件开发库。FreeType的真正吸引力在于它能够加载TrueType字体。

TrueType字体不是用像素或其他不可缩放的方式来定义的，它是通过数学公式（曲线的组合）来定义的。类似于矢量图像，这些光栅化后的字体图像可以根据需要的字体高度来生成。通过使用TrueType字体，你可以轻易渲染不同大小的字形而不造成任何质量损失。

渲染文本

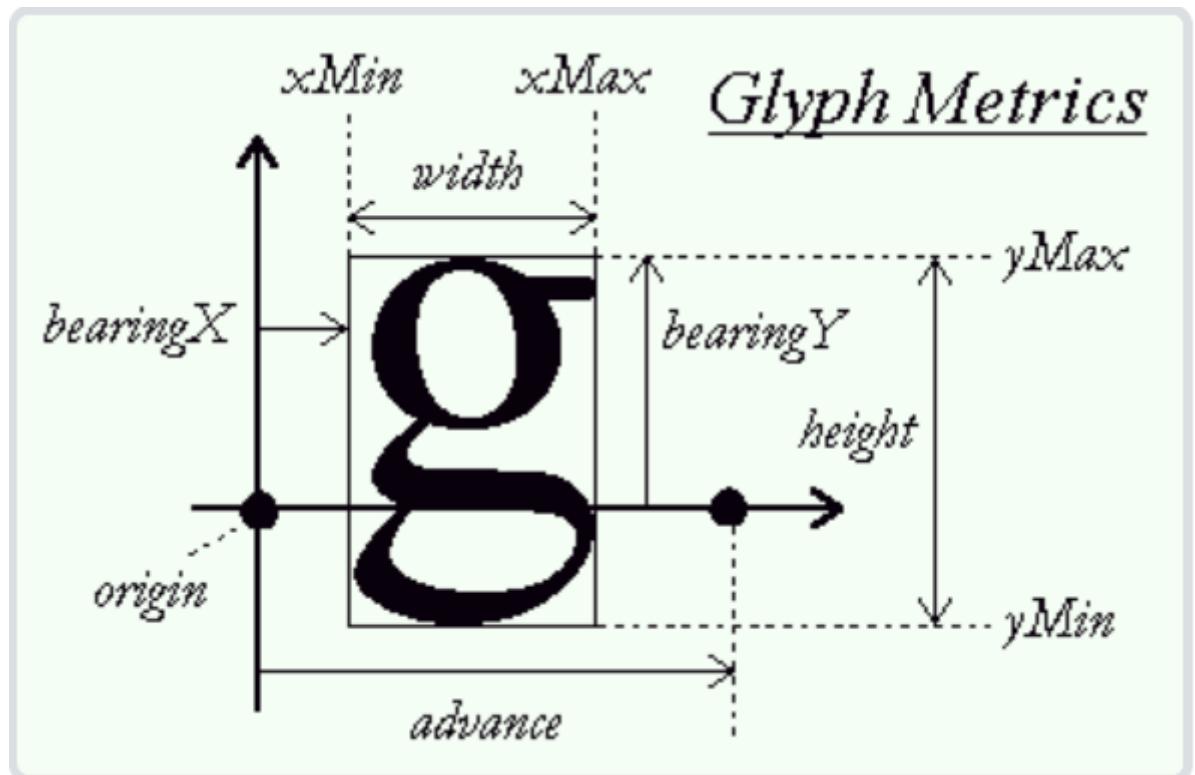
这里不介绍如何使用FreeType，简单说一下是如何显示文本的。

一个FreeType面中包含了一个字形的集合。我们告诉FreeType去创建一个8位的灰度位图，我们可以通过`face->glyph->bitmap`来访问这个位图。

使用FreeType加载的每个字形没有相同的大小。使用FreeType生成的位图的大小恰好能包含这个字符可见区域。例如生成用于表示'.'的位图的大小要比表示'X'的小得多。因此，FreeType同样也加

载了一些度量值来指定每个字符的大小和位置。

下面这张图展示了FreeType对每一个字符字形计算的所有度量值。



在需要渲染字符时，我们可以加载一个字符字形，获取它的度量值，并生成一个纹理，但每一帧都这样做会非常没有效率。我们应将这些生成的数据储存在程序的某一个地方，在需要渲染字符的时候再去调用。我们会定义一个非常方便的结构体，并将这些结构体存储在一个map中。

我们只生成ASCII字符集的前128个字符。对每一个字符，我们生成一个纹理并保存相关数据至Character结构体中，之后再添加至Characters这个映射表中。这样子，渲染一个字符所需的所有数据就都被储存下来备用用了。

我们这里将纹理的internalFormat和format设置为GL_RED。通过字形生成的位图是一个8位灰度图，它的每一个颜色都由一个字节来表示。因此我们需要将位图缓冲的每一字节都作为纹理的颜色值。这是通过创建一个特殊的纹理实现的，这个纹理的每一字节都对应着纹理颜色的红色分量（颜色向量的第一个字节）。

着色器：

我们将使用下面的顶点着色器来渲染字形：

```
#version 330 core
layout (location = 0) in vec4 vertex;
uniform mat4 projection;
void main()
{
    gl_Position = projection * vec4(vertex.xy, 0.0, 1.0);
    TexCoords = vertex.zw;
}
```

我们将位置和纹理纹理坐标的数据合起来存在一个vec4中。这个顶点着色器将位置坐标与一个投影矩阵相乘，并将纹理坐标传递给片段着色器：

```
#version 330 core
in vec2 TexCoords;
out vec4 color;
uniform sampler2D text;
uniform vec3 textColor;
void main()
{
    vec4 sampled = vec4(1.0, 1.0, 1.0, texture(text, TexCoords).r);
    color = vec4(textColor, 1.0) * sampled;
}
```

片段着色器有两个uniform变量：一个是单颜色通道的字形位图纹理，另一个是颜色uniform，它可以用来自调整文本的最终颜色。我们首先从位图纹理中采样颜色值，由于纹理数据中仅存储着红色分量，我们就采样纹理的r分量来作为取样的alpha值。通过变换颜色的alpha值，最终的颜色在字形背景颜色上会是透明的，而在真正的字符像素上是不透明的。我们也将RGB颜色与textColor这个uniform相乘，来变换文本颜色。

我们需要启用混合才能让这一切行之有效：

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

对于投影矩阵，我们将使用一个正射投影矩阵(Orthographic Projection Matrix)。对于文本渲染我们（通常）都不需要透视，使用正射投影同样允许我们在屏幕坐标系中设定所有的顶点坐标，如果我们使用如下方式配置：

```
glm::mat4 projection = glm::ortho(0.0f, 800.0f, 0.0f, 600.0f);
```

我们设置投影矩阵的底部参数为0.0f，并将顶部参数设置为窗口的高度。这样做的结果是我们指定了y坐标的范围为屏幕底部(0.0f)至屏幕顶部(600.0f)。这意味着现在点(0.0, 0.0)对应左下角（译注：而不是窗口正中间）。

最后要做的事是创建一个VBO和VAO用来渲染四边形。现在我们在初始化VBO时分配足够的内存，这样我们可以在渲染字符的时候再来更新VBO的内存。

```
GLuint VAO, VBO;
 glGenVertexArrays(1, &VAO);
 glGenBuffers(1, &VBO);
 glBindVertexArray(VAO);
 glBindBuffer(GL_ARRAY_BUFFER, VBO);
 glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) * 6 * 4, NULL, GL_DYNAMIC_DRAW);
 glEnableVertexAttribArray(0);
 glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 4 * sizeof(GLfloat), 0);
```

```
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

每个2D四边形需要6个顶点，每个顶点又是由一个4float向量组成，因此我们将VBO的内存分配为6 * 4个float的大小。

碰撞检测

我们把3D的碰撞检测用2D的来实现，将复杂的工作简化。使用原理是AABB - AABB 碰撞

AABB - AABB 碰撞

AABB代表的是轴对齐碰撞箱(Axis-aligned Bounding Box)，碰撞箱是指与场景基础坐标轴（2D中的是x和y轴）对齐的长方形的碰撞外形。与坐标轴对齐意味着这个长方形没有经过旋转并且它的边线和场景中基础坐标轴平行（例如，左右边线和y轴平行）。这些碰撞箱总是和场景的坐标轴平行，这使得所有的计算都变得更简单。

Breakout中几乎所有的物体都是基于长方形的物体，因此很理所应当地使用轴对齐碰撞箱来进行碰撞检测。这就是我们接下来要做的。

那么我们如何判断碰撞呢？当两个碰撞外形进入对方的区域时就会发生碰撞，例如定义了第一个物体的碰撞外形以某种形式进入了第二个物体的碰撞外形。对于AABB来说很容易判断，因为它们是与坐标轴对齐的：对于每个轴我们要检测两个物体的边界在此轴向是否有重叠。因此我们只是简单地检查两个物体的水平边界是否重合以及垂直边界是否重合。如果水平边界和垂直边界都有重叠那么我们就检测到一次碰撞。

将这一概念转化为代码也是很直白的。我们对两个轴都检测是否重叠，如果都重叠就返回碰撞：

```
GLboolean CheckCollision(GameObject &one, GameObject &two) // AABB - AABB
{
    // x轴方向碰撞？
    bool collisionX = one.Position.x + one.Size.x >= two.Position.x &&
                      two.Position.x + two.Size.x >= one.Position.x;
    // y轴方向碰撞？
    bool collisionY = one.Position.y + one.Size.y >= two.Position.y &&
                      two.Position.y + two.Size.y >= one.Position.y;
    // 只有两个轴向都有碰撞时才碰撞
    return collisionX && collisionY;
}
```

我们使用栅栏作为边界，下面是我们设置的位置：

```
...
if (car_position.x < -4 && car_position.x > -4.17) {
    car_position.x = -4.17;
}
```

```

...
if (car_position.z < -4.7 && car_position.z > -4.88) {
    car_position.z = -4.88;
}
...
cat_forward_acceleration = -0.2f;

```

如果检测到碰撞，就给车设置加速度为 `-0.2`，这样车的速度就会减下来。

小车的运动系统

为了比较好地模拟真实环境下，汽车的形式状态。我们进入加速度的机制，缓解由于汽车速度突变带来的不舒适感。以下为汽车状态更新函数：

```

void autoUpdate() {
    car_forward_speed += cat_forward_acceleration*deltaTime;
    car_forward_speed = car_forward_speed > 0.0f ? car_forward_speed : 0.
    0f;
    car_rotation_ratio += car_rotation_acceleration*deltaTime;
    if (car_rotation_ratio != 0.0f) {
        // std::cout << "Here rotate, car_rotation_ratio : " << car_rotation_ratio << " car_forward_speed : " << car_forward_speed << std::endl;
        /*std::cout << "Sin : " << sin(glm::radians(car_rotation_ratio))
        << " Cos : " << cos(glm::radians(car_rotation_ratio)) << std::endl;*/
        car_forward.z = (-1)*sin(glm::radians(car_rotation_ratio +
90.0f));
        car_forward.x = cos(glm::radians(car_rotation_ratio + 90.0f));
    }
    car_position.z += car_forward.z*car_forward_speed*deltaTime;
    car_position.x += car_forward.x*car_forward_speed*deltaTime;
    // 为了保持第一人称，同样实时更新摄像机的位置
    camera_position.x = car_position.x - car_forward.x;
    camera_position.z = car_position.z - car_forward.z;
    camera_position.y = car_position.y + 0.4f;
}

// other function
{
    // 摄像机的projection和view，这样就能实现摄像机一直注释汽车尾部的效果
    projection = glm::perspective(45.0f, (float)SCR_WIDTH / (float)SCR_HEIG
HT, 0.1f, 100.0f);
    view = glm::lookAt(camera_position, car_position, glm::vec3(0.0f, 1.0f,
0.0f));
}

```

可以看到，我们把汽车的前行和转弯都合并到了`car_forward`中，逐帧更新汽车的实时位置和速度。

当玩家按下减速键时，给小车一个较大的反向加速度，当玩家不前进也不刹车时，摩擦力给小车一个较小的反向加速度。以下是键盘监听函数：

```
void processInput(GLFWwindow *window, int key, int scancode, int action,
int mods)
{
    if (key == GLFW_KEY_ESCAPE)
        glfwSetWindowShouldClose(window, true);
    if (action == GLFW_PRESS && key == GLFW_KEY_SPACE) {
        if (status == STATUS::NONE) {
            status = STATUS::MOVE;
            std::cout << "Switch status from NONE to MOVE" << std::endl;
        }
        else {
            status = STATUS::NONE;
            std::cout << "Switch status from MOVE to NONE" << std::endl;
        }
    }
    if (status == STATUS::MOVE) {
        if (action == GLFW_PRESS) {
            if (key == GLFW_KEY_A)
                car_rotation_acceleration = 50.0f;
            if (key == GLFW_KEY_D)
                car_rotation_acceleration = -50.0f;
            if (key == GLFW_KEY_W)
                cat_forward_acceleration = 0.2f;
            if (key == GLFW_KEY_S)
                cat_forward_acceleration = -0.8f;
        }

        if (action == GLFW_RELEASE) {
            if (key == GLFW_KEY_A || key == GLFW_KEY_D)
                car_rotation_acceleration = 0.0f;
            if (key == GLFW_KEY_W || key == GLFW_KEY_S)
                cat_forward_acceleration = -0.1f;
        }
    }
}
```

Particle System

在项目中，用粒子系统实现了下雪的效果和喷泉的效果。

使用粒子系统主要用到了opengl的Transform Feedback的特性。粒子系统主要参照了：[opengl教程](#)

其实现思路是在 GS (如果没有使用 GS 则是 VS) 处理之后，我们可以将变换之后的图元存放到一个特殊的缓存——Transform Feedback Buffer 中。此外，我们可以决定图元是否将继续按照以前的流程进行光栅化。在下一帧渲染的时候，上一帧中输出的顶点信息可以

在这一帧中作为顶点缓存使用，在这样的一个循环过程中，我们可以不借助于应用程序来实现对粒子信息的更新。

在实现了粒子系统框架后，可以更改框架中粒子系统的运动参数从而更改粒子的行为，项目中正是利用这个来实现了下雪的效果和喷泉的效果。

对粒子运动的控制主要在粒子对应的update几何着色器中实现。

对于下雪的粒子处理，主要实现的是初始位置在天空，然后加速度朝下，给予一定水平方向的速度，使其像是飘雪。

对于喷泉的粒子处理，如下所示：

喷泉正在朝着y轴方向喷涌，从xoz平面看去，你会看到一个圆形。也就是说在xoz平面，它360度全方位喷洒，这里暂不考虑风力等因素。所以x和z方向上，它的速度方向的分量Vx和Vz是在360度之间随机选择，或者说在-180.0f到180.0f之间随机。那么得出x和y方向的速度初始分量如下：

```
Vx=sin(R1×Rand(a/2))×cos(R2×2×Pi)  
Vz=sin(R1×Rand(a/2))×sin(R2×2×Pi)
```

喷泉粒子的只受重力影响，所以设置其加速度为向下的重力加速度。

然后根据上面的速度，在update几何着色器中，修改相关速度即可得到相应的粒子系统。

Explosion Effect

在开始时，实现了路障的爆炸效果，使用的是opengl几何着色器的思想。步骤如下：

1. 使用减法获取了两个平行于三角形表面的向量a和b。
2. 使用叉乘来获取垂直于a和b的一个向量，即法向量。
3. 使用几何着色器，来实现模型的mesh朝法向量的方向移动，这样就能达到看起来像是爆炸的效果。

核心代码如下：

```
vec3 GetNormal()  
{  
    vec3 a = vec3(gl_in[0].gl_Position) - vec3(gl_in[1].gl_Position);  
    vec3 b = vec3(gl_in[2].gl_Position) - vec3(gl_in[1].gl_Position);  
    return normalize(cross(a, b));  
}  
  
vec4 explode(vec4 position, vec3 normal)  
{  
    float magnitude = 2.0;  
    vec3 direction = normal * ((sin(time) + 1.0) / 2.0) * magnitude;  
    return position + vec4(direction, 0.0);  
}
```

问题与解决方案

在完成项目的过程中主要遇到问题有如下几点：

1. 模型网站模型格式混乱：

这点在建模初期体现比较明显，开始时对于blender使用很陌生，导致加载出下载模型位置混乱，同时也没有贴图，经过对blender一段时间的使用，学会了如何对模型进行贴图，同时也了解了.mtl文件相对路径和绝对路径的修改方式，解决了贴图问题。

2. 模型过大，电脑难以运行：

在项目开始阶段构建了一个场景的demo，构建之后大约有100M，电脑加载时间很长。查找相关资料决定使用blender自带的优化工具，同时对模型的三部分进行了优化：首先是模型存储方式，将多个单模型整合为一个整体模型。其次是对重复顶点的删除，大部分免费模型都是很随意的，不记存储代价的，导致重复顶点过多。最后就是对于模型面数的减少，由于网站下载的都为单模型，模型比较精细，而我们这个项目重点不在于模型的精细程度，因此减少一定的面数来换取存储空间是十分值得的。最终整体模型从170M优化到了7.6M，运行的流畅性得到了很大提升。

3. glad不支持某些opengl3.3的函数：

在完成粒子系统的过程中，发现粒子系统需要用到的glTransformfeedback相关的函数在glad中不支持，解决办法是放弃了glad，选择了glew。

小组成员分工

成员	分工
苏瀚添	basic场景构建、天空盒、粒子系统、爆炸效果、项目整合、展示PPT与报告等。
宋志强	文本显示、碰撞处理、整理代码。
宋孝天	模型的建立整合、静态阴影处理。
夏显卓	运动系统构建、帮忙debug和解决问题
冼圣杰	运动系统构建、帮忙debug和解决问题

文件结构说明

各个文件都基本能够从其命名上查看其属于什么文件，需要注意的是在shaders文件里面放的都是着色器，在res文件里面放的都是模型文件和贴图、fonts文件里面放的是字体。

展示后的改进

因为我们小组在16周就展示，当时项目还没有完全完成，PPT制作得也不够完整。在展示完后，根据老师和TA的建议，我们对我们的项目和PPT文件进行了改进，改进包括下列方面：

1. 添加碰撞处理，使车的运动被限制在跑道内。

2. 添加了阴影的处理。
3. 添加了开始时的路障和爆破结果的实现。
4. 优化了代码和文件结构，使不同的文件分文件夹放好。
5. 优化了汽车的运动系统，使其运动起来更加自然。
6. PPT上面添加了项目效果图介绍、完成的最终技术说明和分工说明。
7. 更改了PPT的一些错误。