计算机图形学期末项目

模拟驾驶

小组成员

颜承橹 15322244

谭发豪 16340202

吴聪 16340237

谢浩峰 16340253

2019.07.18

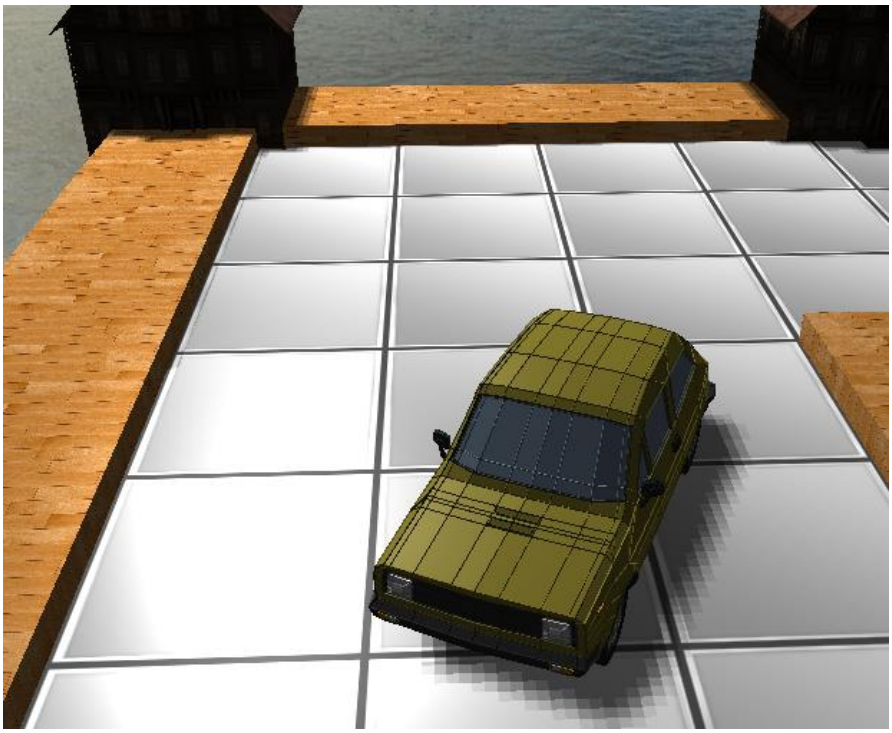
目录

一、 项目主题

我们组选择的项目主题是模拟驾驶。

在这个场景中，玩家可以操控汽车进行移动。汽车与场景之间存在交互，比如汽车无法驶出边界，又比如当汽车撞到其他物体时，会触发相应的事件，这里我们实现的是物体被撞后会发生爆炸的效果。此外，为了使模拟驾驶更加真实，我们还添加了汽车加速减速的物理效果，并搭载了一些辅助功能，例如将汽车的当前速度以文字的形式显示在窗口，又或是让玩家能自由切换视角。

最终实现的结果预览如下：



二、 开发环境与第三方库

项目的开发环境：Win10+Visual Studio+OpenGL

使用的第三方库：GLFW，GLAD，Assimp，FreeType，glm

三、 实现功能列表

Basic：

1. Camera Roaming
2. Simple lighting and shading
3. Texture mapping
4. Shadow mapping
5. Model import& Mesh viewing

Bonus：

1. Sky Box
2. Display Text
3. Collision Detection
4. Particle System
5. Explosion Effect
6. Physical effects

四、 功能介绍

Basic 部分

1. 相机漫游

在该项目中，我们设置了两个摄像头，一个与小车绑定，一个是自由镜头。通过“C”键可实现摄像头之间的切换。

自由镜头的移动受 WASD 键控制，对应前后左右四个方向。

```
void ProcessKeyboard(Camera_Movement direction, float deltaTime)
{
    float velocity = MovementSpeed * deltaTime;
    glm::vec3 front = glm::normalize(glm::vec3(Front.x, 0, Front.z));
    if (direction == FORWARD)
        Position += Front * velocity;
    if (direction == BACKWARD)
        Position -= Front * velocity;
    if (direction == LEFT)
        Position -= Right * velocity;
    if (direction == RIGHT)
        Position += Right * velocity;
```

与车绑定的镜头共有六个方向，分别是前、后、左前、左后、右前和右后。为了使镜头能在车转向时与小车的朝向保持一致，故在转向时还添加了类似鼠标左右移动的操作。

```
// Processes input received from any keyboard-like input system. Accepts input parameter
in the form of camera defined ENUM (to abstract it from windowing systems)
void ProcessKeyboard(Camera_Movement direction, glm::vec3 frontOfCar, glm::vec3 upOfCar,
float deltaTime)
{

    float velocity = MovementSpeed * deltaTime;
    glm::vec3 rightOfCar = glm::normalize(glm::cross(frontOfCar, upOfCar));
    //glm::vec3 front = glm::normalize(glm::vec3(Front.x, 0, Front.z));
    if (direction == FORWARD)
        Position += frontOfCar * velocity;
    if (direction == BACKWARD)
        Position -= frontOfCar * velocity;
    if (direction == LEFT_FORWARD)
        ProcessMouseMovement(-20, 0, TRUE);
    if (direction == RIGHT_FORWARD)
        ProcessMouseMovement(10, 0, TRUE);
    if (direction == LEFT_BACKWARD)
        ProcessMouseMovement(20, 0, TRUE);
    if (direction == RIGHT_BACKWARD)
        ProcessMouseMovement(-10, 0, TRUE);
```

2. 简单光照

在本项目中，我们使用了两种光照模型，Phong 光照模型和 Blinn 光照模型。通过“B”键可以实现摄像头之间的切换。

```
uniform bool blinn;

void main()
{
    vec3 color = texture(aTexture, fs_in.TexCoords).rgb;
    // ambient
    vec3 ambient = 0.05 * color;
    // diffuse
    vec3 lightDir = normalize(lightPos - fs_in.FragPos);
    vec3 normal = normalize(fs_in.Normal);
    float diff = max(dot(lightDir, normal), 0.0);
    vec3 diffuse = diff * color;
    // specular
    vec3 viewDir = normalize(viewPos - fs_in.FragPos);
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = 0.0;
    if(blinn)
    {
        vec3 halfwayDir = normalize(lightDir + viewDir);
        spec = pow(max(dot(normal, halfwayDir), 0.0), 32.0);
    }
    else
    {
        vec3 reflectDir = reflect(-lightDir, normal);
        spec = pow(max(dot(viewDir, reflectDir), 0.0), 8.0);
    }
    vec3 specular = vec3(0.3) * spec; // assuming bright white light color
    FragColor = vec4(ambient + diffuse + specular, 1.0);
}
```

使用 uniform 变量 blinn 来实现控制着色器的光照渲染模式。

Phong 光照模型很好，但 Phong 光照模型有一个问题，当观察向量和反射向量之间的夹角若超过 90°，则镜面高光分量会因为向量点积为负数而结果为 0。

Blinn 光照模型引入了半程向量来解决这个问题，半程向量即光线与视角的一半方向上的单位向量，当半程向量与法线向量越接近时，镜面光分量就越大。

3. 纹理映射

由于在这样相对较大的应用中，我们需要频繁地重用纹理，故我们为其定义了一个纹理类，该纹理类接受一个字节(Byte)数组以及宽度和高度，并（根据设定的属性）生成一个 2D 纹理图像（我们设计的纹理类是为 2D 纹理设计的）。

```
// Texture2D is able to store and configure a texture in OpenGL.
// It also hosts utility function for easy management.
class Texture2D
{
public:
    // Holds the ID of the texture object
    GLuint ID;
    // Texture image dimensions
    GLuint Width, Height;   // Width and height of loaded image in pixels
    // Texture Format
    GLuint Internal_Format; // Format of texture object
    GLuint Image_Format;    // Format of loaded image
    // Texture configuration
    GLuint Wrap_S;  // Wrapping mode on S axis
    GLuint Wrap_T;  // Wrapping mode on T axis
    GLuint Filter_Min;  // Filtering mode if texture pixels < screen pixels
    GLuint Filter_Max;  // Filtering mode if texture pixels > screen pixels
    // Constructor (sets default texture modes)
    Texture2D();
    // Generates texture from image data
    void Generate(GLuint width, GLuint height, unsigned char* data);
    // Binds the texture as the current active GL_TEXTURE_2D texture object
    void Bind() const;

};
```

我们没有将文件加载相关的代码嵌入到纹理类中，这是因为我们另外设计了一个所谓资源管理器的实体，专门加载本项目的所有资源，纹理的文件加载机制在其中实现。

```
Texture2D ResourceManager::loadTextureFromFile(const GLchar *file, GLboolean alpha)
{
    // Create Texture object
    Texture2D texture;
    if (alpha)
    {
        texture.Internal_Format = GL_RGBA;
        texture.Image_Format = GL_RGBA;
    }
    // Load image
    int width, height, nrComponents;
    unsigned char *image = stbi_load(file, &width, &height, &nrComponents, 0);
    if (image)
    {
        GLenum format;
        if (nrComponents == 1)
        {
            texture.Internal_Format = GL_RED;
            texture.Image_Format = GL_RED;
        }
        else if (nrComponents == 3)
        {
            texture.Internal_Format = GL_RGB;
            texture.Image_Format = GL_RGB;
        }
        else if (nrComponents == 4)
        {
            texture.Internal_Format = GL_RGBA;
            texture.Image_Format = GL_RGBA;
        }
    }
    // Now generate texture
    texture.Generate(width, height, image);
    // And finally free image data
    stbi_image_free(image);
    return texture;
}
```

4. 阴影渲染

阴影渲染主要使用 Shadow Mapping 的方式来实现。

初步阴影渲染完成以后还会存在各种问题，比如阴影失真，阴影悬浮，阴影贴图采样越界，以及阴影的锯齿问题。

我们主要使用了阴影偏移来解决阴影失真，压缩光源投影的近远平面来减缓阴影悬浮，重设置贴图环绕方式（采用 GL_CLAMP_TO_BORDER 且设置 Border 颜色）来解决阴影贴图采样越界，适当增加深度贴图解析度与采用 PCF 技术来抗锯齿。

5. 模型导入与网格

模型的导入使用了第三方库 assimp，在 LearnOpenGL 中已经给出了相应的模型导入示例（需要构建 Model 类和 Mesh 类）。

需要说明的是，LearnOpenGL 中提供的代码对待导入的模型是有要求的，它要求待导入的模型必须是 obj 格式的且必须带有相应的 mtl 文件，此外，obj 模型的所有模型部件（网格）都至少有一个漫反射贴图，也就是说，这份代码不支持没有贴图的模型（哪怕是只有一小部分没有贴图）。如果需要读取这样的模型，需要修改 Model 类中的 processMesh 函数部分如下：

```
for (unsigned int i = 0; i < mesh->mNumVertices; i++)
{
    Vertex vertex;
    glm::vec3 vector; // we declare a placeholder vector since assimp uses its
    // positions
    if (mesh->HasPositions())
    {
        vector.x = mesh->mVertices[i].x;
        vector.y = mesh->mVertices[i].y;
        vector.z = mesh->mVertices[i].z;
        vertex.Position = vector;
    }
    // normals
    if (mesh->HasNormals())
    {
        vector.x = mesh->mNormals[i].x;
        vector.y = mesh->mNormals[i].y;
        vector.z = mesh->mNormals[i].z;
        vertex.Normal = vector;
    }
    // texture coordinates
    if (mesh->mTextureCoords[0]) // does the mesh contain texture coordinates?
    {
        glm::vec2 vec;
        // a vertex can contain up to 8 different texture coordinates. We thus
        // use models where a vertex can have multiple texture coordinates so w
        vec.x = mesh->mTextureCoords[0][i].x;
        vec.y = mesh->mTextureCoords[0][i].y;
        vertex.TexCoords = vec;
    }
    else
        vertex.TexCoords = glm::vec2(0.0f, 0.0f);
    if (mesh->HasTangentsAndBitangents())
    {
        // tangent
        vector.x = mesh->mTangents[i].x;
        vector.y = mesh->mTangents[i].y;
        vector.z = mesh->mTangents[i].z;
        vertex.Tangent = vector;
        // bitangent
        vector.x = mesh->mBitangents[i].x;
        vector.y = mesh->mBitangents[i].y;
        vector.z = mesh->mBitangents[i].z;
        vertex.Bitangent = vector;
    }
    vertices.push_back(vertex);
```

Bonus 部分

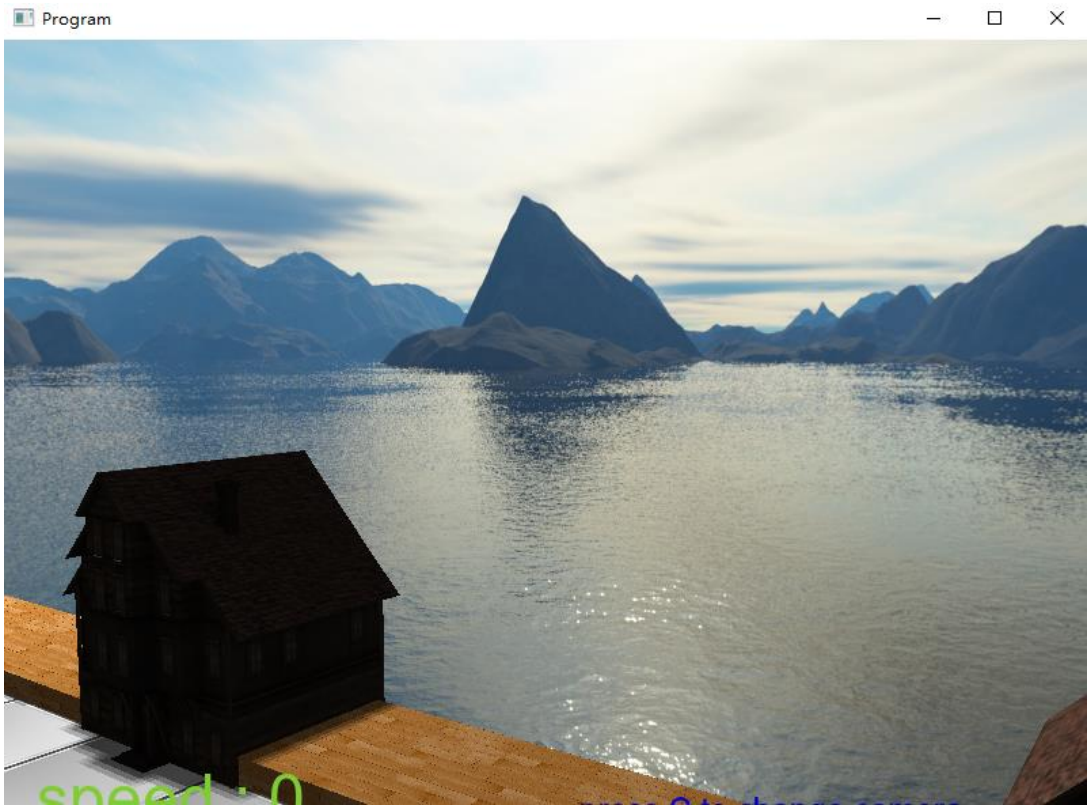
1. 天空盒

Skybox 就是实现一个贴了 6 个 2D 纹理的立方体，包含整个场景与环境。

首先按照常规的方法渲染一个贴了纹理的立方体。为了使玩家移动的时候不会觉得天空盒在变近，在传入 view 的时候，要将 4x4 矩阵变换矩阵的位移部分移除掉，只保留旋转变换：

```
// Skybox Shader
ResourceManager::GetShader("skyShader").Use().SetMatrix4("view",
    glm::mat4(glm::mat3(this->camera->GetViewMatrix())));
```

优化的时候，使用提前深度测试来节省资源。最终实现的结果如下：



2. 爆炸效果

爆炸效果主要由一个爆炸着色器来实现，通过在渲染管线中动态修改每一个片段的位置来达到爆炸效果。

爆炸着色器主要包括三个着色器：顶点着色器、几何着色器和片段着色器。

顶点着色器主要包括了坐标系统的转换，通过投影矩阵将模型的局部坐标转换为投影坐标。

几何着色器是区别与其他（阴影、光照等）着色器的主要点。在几何着色器中，着色器重新计算每一个片段的坐标，再传给片段着色器进行渲染。在几何着色器中，一个爆炸函数将每一个片段根据其位置和法向量确定要爆炸的方向，即每个片段向自己的法向量方向爆炸，然后再根据一个随时间单调递增的变量来计算爆炸出去的距离。

最后是片段着色器，片段着色器和其他的片段着色器一致，计算片段的实际颜色（加上光照影响）。

```
vec4 explode(vec4 position, vec3 normal)
{
    float magnitude = 10.0;
    vec3 direction = normal * ((sin(time) + 1.0) / 2.0) *
    magnitude;
    return position + vec4(direction, 0.0);
}

void main() {
    vec3 normal = GetNormal();

    gl_Position = explode(gl_in[0].gl_Position, normal);
    TexCoords = gs_in[0].texCoords;
    EmitVertex();
    gl_Position = explode(gl_in[1].gl_Position, normal);
    TexCoords = gs_in[1].texCoords;
    EmitVertex();
    gl_Position = explode(gl_in[2].gl_Position, normal);
    TexCoords = gs_in[2].texCoords;
    EmitVertex();
    EndPrimitive();
}
```

实际的渲染效果如下图：

3. 碰撞检测

碰撞检测的原理其实判断小车下一步的位置是否与其他物体重叠，若发生重叠则返回碰撞，将这一概念转化为代码是很直白的。在这个项目中需要检测的碰撞有两项，一是小车与道路边界的碰撞，而是小车与小屋模型的碰撞。

小车与道路边界的碰撞检测发生在汽车坐标更新的时候，当小车依据输入的指令发

生位移时，会计算新的坐标是否与边界重叠，从而来决定是否更新坐标。

```
void Game::Update(GLfloat dt)
{
    glm::vec3 position = carPos + carShift;
    float size = 39;
    float road = 6;
    float border = 1;
    if (position.x > border && position.z > border && position.x < (size - border) &&
    position.z < (size - border)) {
        if (position.x < road || position.z<road || position.x >(size - road) ||
        position.z >(size - road))
            carPos = position;
    }

    //particles->Update(dt, 2, carShift, carPos, glm::vec2(100.0f, 100.0f));
    carShift = glm::vec3(0.0f, 0.0f, 0.0f);

    particleSystem->Update(dt, camera->Position);
}
```

小车与小屋模型的碰撞则是在渲染场景的时候检测。通过将小屋的 x、y 坐标存储在数组中，然后计算车与小屋的欧式距离，来判断是否发生了碰撞。

```
void Game::IsConflict(int i) {
    bool collisionX = abs(carPos.x + carSizeX - 35.4) <= 1.5 ||
    abs(carPos.x - carSizeX - 35.4) <= 1.5;
    bool collisionY = abs(carPos.z + carSizeY - brokehousez[i])
    <= 1.5 || abs(carPos.z - carSizeY - brokehousez[i]) <= 1.5;
    if (collisionX && collisionY) {
        brokehouseflag[i] = true;
    }
}
```

4. 文字渲染

文字渲染使用到了 FreeType 库，其原理就是加载字体并为每一个字形生成位图以及计算几个度量值。在需要的时候将其取出并贴到指定的 2D 方块中,从而实现文字渲染。

文字渲染的流程如下。首先，加载 arial.ttf 字体文件，并设置每个字的大小

```
// FreeType
FT_Library ft;
// All functions return a value different than 0 whenever an error occurred
if (FT_Init_FreeType(&ft))
    std::cout << "ERROR::FREETYPE: Could not init FreeType Library" << std::endl;

// Load font as face
FT_Face face;
if (FT_New_Face(ft, "../Resources/fonts/arial.ttf", 0, &face))
    std::cout << "ERROR::FREETYPE: Failed to load font" << std::endl;

// Set size to load glyphs as
FT_Set_Pixel_Sizes(face, 0, 48);
```

然后对字符集中前 128 个字符生成对应的纹理并储存在 Characters 映射表中

```
// Now store character for later use
Character character = {
    texture,
    glm::ivec2(face->glyph->bitmap.width, face->glyph->bitmap.rows),
    glm::ivec2(face->glyph->bitmap_left, face->glyph->bitmap_top),
    face->glyph->advance.x
};
Characters.insert(std::pair<GLchar, Character>(c, character));
```

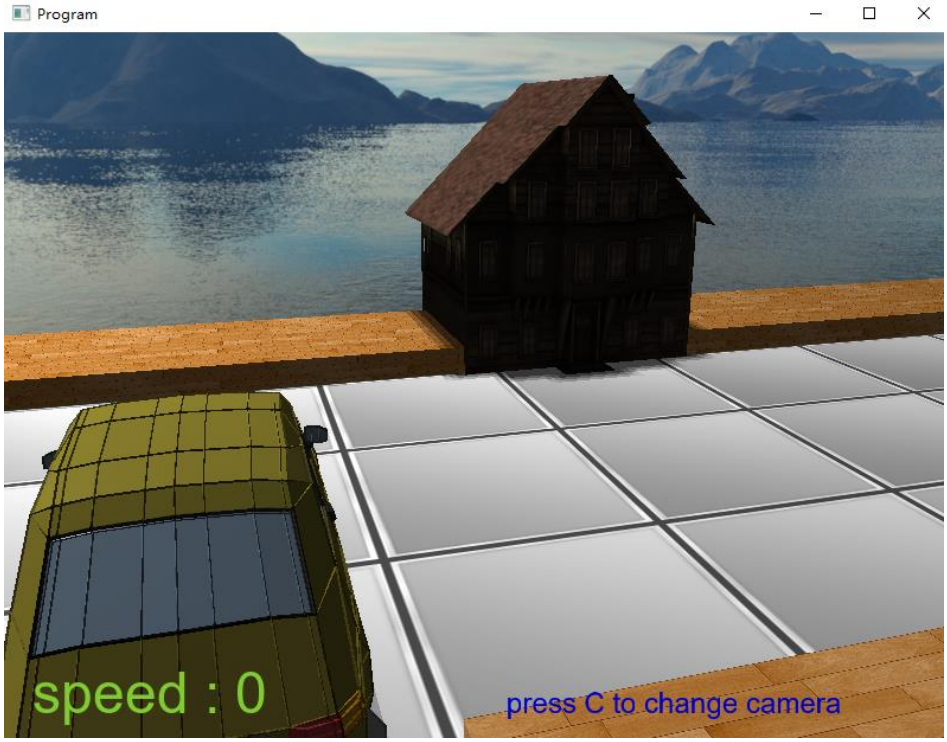
在渲染时，启动混合并让位图字体的纹理背景保持透明，

```
// Set OpenGL options

glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

渲染一个字符串时，会从之前创建的 Characters 映射表中取出对应的 Character 结构体，并根据字符的度量值来计算四边形的维度。根据四边形的维度，就能动态计算出六个描述四边形的顶点，并使用 glBufferSubData 函数更新 VBO 所管理内存的内容。

最终渲染出来的结果如下：



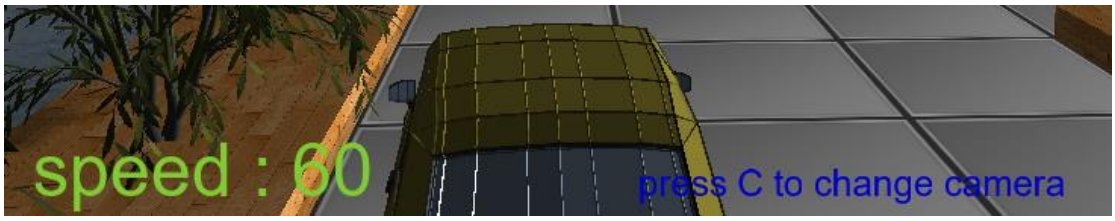
5. 物理效果

该项目实现了一个汽车加速减速的物理效果。当车由静止转换为运动时，车的速度会逐渐上升直至一个阈值。但松开 WASD 方向键后，车会保留运动的惯性，速度也会有一个降低到 0 的过程。其实现的原理就是对方向键的输入和当前的速度做判断。当速度未到达阈值时，若方向键与小车方向一致，则会加速，否则就减速。

为了能更明显地体现这一点，我们运用文字渲染加以辅助，使其更直观地体现出运动时速度的变化。

```
if (!camera.freecamera) {
    frontOfCar = SimpleScene.getFrontOfCar();
    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS) {
        if (SimpleScene.carSpeed < 20) {
            SimpleScene.carSpeed += 1;
        }
        //camera.ProcessKeyboard(FORWARD, frontOfCar, upOfCar, deltaTime);
        if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS) {
            camera.ProcessKeyboard(LEFT_FORWARD, frontOfCar, upOfCar, deltaTime);
            SimpleScene.ProcessInput(window, LEFT_FORWARD, frontOfCar, upOfCar, deltaTime);
            int s = 3 * SimpleScene.carSpeed;
            SimpleScene.str = "speed : " + to_string(s);
```

最终效果如下：车速逐渐上升并达到最大值 60.



6. 粒子系统

所谓粒子（Particle）其实就是一个一个小多边形的渲染，并且随机或按某种物理规律，分配其速度与位置，让其落下或喷发。

因为粒子是大量图形的渲染，如果不断渲染的话，内存会不断增加，因此要重复利用资源。具体实现时，每个粒子都要有一个存活时间，当时间到了就要将取消这个粒子的渲染，将其删掉：

```
struct Particle {
    glm::vec3 position;
    glm::vec4 color;
    glm::vec3 velocity;
    glm::vec3 acceleration;

    glm::vec2 rotateAngle;
    float rotateSpeed;
    float lifetime;
    float scale;
    float dec;

    Particle()
        : position(0.0f), velocity(0.0f), color(1.0f), lifetime
        (1.0f), scale(1.0f), acceleration(1.0f), dec(1.0f),
        rotateAngle(0.0f), rotateSpeed(0.0f) { }
};
```

粒子的渲染初始化与普通多边形渲染过程一致，这里不赘述。
需要一个函数，来随机分配一个粒子的各个参数（如位置，速度）：

```
void respawnParticle(Particle& particle) {
    //初始化颜色（白色）
    r = 1.0f;
    g = 1.0f;
    b = 1.0f;
    a = 1.0f;
    particle.color = glm::vec4(r, g, b, a);

    //初始化坐标
    x = range_x * (rand() / float(RAND_MAX)) - range_x / 2;
    z = range_z * (rand() / float(RAND_MAX)) - range_z / 2;
    y = skyHeight;
    particle.position = glm::vec3(x, y, z);

    //初始化速度
    vx = rand() / float(RAND_MAX) * VELOCITY_X_MAX;
    vy = rand() / float(RAND_MAX) * (VELOCITY_Y_MAX -
    VELOCITY_Y_MIN) + VELOCITY_Y_MIN;
    vz = 0;
    particle.velocity = glm::vec3(vx, vy, vz);

   //旋转角度
   rx = (rand() / float(RAND_MAX)) * ROTATE_ANGLE_MAX;
   ry = (rand() / float(RAND_MAX)) * ROTATE_ANGLE_MAX;
   //旋转速度
   rv = rand() / float(RAND_MAX) * ROTATE_VELOCITY;

   particle.rotateAngle = glm::vec2(rx, ry);
   particle.rotateSpeed = rv;
```

然后再实现一个 Update 函数，更新粒子状态，让其飘动：

```
void Update(float dt, glm::vec3 cameraPos)
```

最后实现 Draw 函数，设置其 shader 参数，进行粒子的渲染。

五、 遇到的问题和解决方案

1. 阴影渲染失败

解决方案：由于在设计项目主框架的时候没有仔细考虑到对场景中的对象应用阴影的情况，故在初期实现阴影渲染的时候出现了一些问题。我们的框架是创建一个场景中的对象的时候需要为其制定一个着色器，而要实现阴影映射，我们的每个对象应该需要先使

用深度着色器来先渲染一次以获得深度贴图，为此，我们将每个对象的 Shader 成员设置为 public 的，在渲染时，我们先将 Shader 成员设为深度着色器，然后得到深度贴图后再设置为本身合适的着色器。此外，我们也稍微修改了一下场景中对象的 Draw 成员函数，使其额外再接收一个 DepthMap 参数（也即深度贴图），当然对于导入模型的阴影实现也需要我们修改 LearnOpenGL 中提供的 Model 类以及 Mesh 类的 Draw 函数，方式同样是使其额外再接受一个深度贴图。

2. 场景渲染过程慢，且占用的内存随着程序的运行越来越大
解决方案：通过查看代码，我们发现场景的初始化被放到了循环渲染中，这导致与场景相关的数据被不断生成，最终程序占用的内存越来越大。 因此我们将场景的初始化和渲染分开，从而解决了这个物体。场景中的物体也都是按照先初始化再渲染的顺序，因此我们的程序加载完成后，能够十分流畅地运行。

3. 出现鱼眼镜头的问题
解决方案：通过检测代码，发现是 projection 的问题，误把 45.0 和 radius(45.0)写反了。

4. 小车模型的原点不是模型的中心
由于小车模型的原点不是模型中心，在进行车的转向时，会发现左转的半径大于右转半径，从而产生一种矛盾感。此外，在碰撞检测时，由于检测的是模型原点的坐标，因此可能会出现逆行时小车还是驶出边界的情况。
解决方案：记录小车的朝向，根据小车的 front 和 up 方向可以计算出 right 的对应向量。利用 center = original + x* font +z*right 就可以计算出模型的中点。以中点来进行操作就不会再出现上述的问题。

六、 小组成员分工

姓名	学号	分工	贡献度
颜承橹	15322244	文字渲染，场景布置 小车移动	25%
谭发豪	16340202	场景布置，粒子系统 天空盒	25%
吴聪	16340237	搭建框架，阴影渲染 模型导入	25%
谢浩峰	16340253	爆炸效果，碰撞检测 阴影渲染	25%

七、 个人报告汇总

15322244 颜承橹

我在期末项目中主要负责的工作是文字渲染、场景布置、相机设置和小车移动。

文字渲染这部分参考了教程，实现的方法是使用 FreeType 库将字符加载成位图并封装好，等到需要的时候再从映射表中取出，并将文字渲染在四边形上。

场景布置这块的麻烦之处在于模型不太好找，而且即便是找到了模型，还需要再修改一下它的配置文件才能正常导入。所以也要感谢一下帮忙寻找模型的其他组员。

小车移动这块要将碰撞检测也考虑进去，因此我还花了些功夫去测量场景的 size。而且之前由于模型原点与模型中心不重合的问题，导致碰撞检测的效果不太好，不过所幸问题后来还是解决了。

总的来说，这次的期末项目做的比较简单。不过麻雀虽小五脏俱全，该有的功能基本上的实现了，我们也尽己所能把从计算机图形学课上学到的知识都运用到了这个项目上，我个人感觉是收获良多的。

16340202 谭发豪

我主要负责场景建模，粒子系统，天空盒的工作，在实现的过程中得到了一点关于渲染时优化的经验。

场景建模是我负责的工作中比较重要的一部分，因为场景决定了这个项目的基础内容之一。我们的场景主要就是车子移动的地板和挡住车子的强。以地板 plane 为例，一开始我想实现一个动态的场景类，从而方便传入参数就能修改场景，我在每次 Render 的时候，都会根据决定好的地板形状，传入其长宽参数，然后从 VAO 开始渲染。但是这样做的话，每次 Render 的时候进行一次从 VAO 开始的数据初始化，使得程序运行时消耗的内存不断提升，越来越卡。

为了避免这个问题，我最终还是选择了静态的场景类。不过，“静态”指的是 VAO 数据的初始化不变（在类中只执行一次初始化），以 1.0f 为单位。而渲染的时候，依然可以传入长宽参数，然后对 model 进行变换，如 scale，translate，以此达到方便修改场景渲染的目的。

实现粒子的时候，其优化过程也是一个需要注意的地方。如果每个粒子都进行从初始化到渲染再到消除的过程，这个内存消耗可能会较大。因此要懂得重复利用已有资源，实现一个数组存放粒子，对其重复使用。

本次项目实现的内容不难，基本都是已学过的知识。只是实现加分点的时候要自学还没学过的内容。不过即使是做过的知识点，在实际大项目中将其融汇在一起也是一件比较难的事，主要难在将其彻底分离，不至于类与类之间耦合而牵一发动全身。这点要感谢我的组员实现了一个良好的框架。

16340237 吴聪

我主要负责项目整体框架搭建，模型导入，阴影渲染及其优化的工作。

项目整体框架的搭建部分主要参考了 LearnOpenGL 中最后的 breakout 游戏的框架。但 breakout 游戏是一个 2D 平面游戏，而我们的项目是 3D 的，为解决这个问题，对于原先 breakout 游戏的框架我做了一些取舍，主要保留了其中的资源管理器，纹理类，着色器类，游戏对象等。此外，原先的 breakout 游戏在设计时并没有考虑到模型导入的情

况，所以其中的资源管理器只是用于管理纹理和着色器，我做了扩展使其能管理导入模型。有了资源管理器，那么我们就能确保场景中的所有对象都只被创建一次，但被多次使用（Draw），这也是我们搭建项目框架的一个重要的原因。

模型导入部分所使用的代码主要来自 LearnOpenGL，需要使用第三方库 assimp。但 LearnOpenGL 给出的代码使用范围比较有限。后来比较认真地了解了 assimp 的一些接口的知识稍微做了一些改动，以允许导入模型能够不具备纹理贴图和法线贴图。

阴影渲染和优化就都是老内容了，唯一比较新的部分是需要处理导入模型的阴影，此外，为了得到比较好的阴影映射效果，需要较为精细地设置近远平面的平面大小以及近远平面的距离。

虽然项目做的比较简单，但通过做这次项目，我较为全面地复习了这一学期所学的计算机图形学的知识，也大概了解到更加复杂的项目应该怎么做，还是收货很多的，此外，也感谢我的组员们在本次项目中的积极参与与配合。

16340253 谢浩峰

我主要负责阴影渲染、爆炸效果和碰撞检测，在这次的课程项目中学习到了不少在平时作业中得不到的经验。

阴影渲染中我一开始按照作业的经验来做，但是并没有成功，后来这部分被队友吴聪实现了，从他的实现中我学习到了如何在一个较为复杂的框架中实现阴影渲染，这给了我很多的经验，一个作业中的功能可能在一个简单的框架中比较容易实现，但是在一个较为复杂的框架里面是完全不同的。然后我在这个基础上，实现了不同物体之间的阴影，即一个物体在另外一个物体上产生的阴影。

同时我还实现了基础的碰撞检测，即简单的 AABB 碰撞，基本的 AABB 碰撞只需要检测车的位置，同时检测地图边缘和房子的位置，通过位置的简单判断来触发爆炸效果和限制车辆的移动。

然后是爆炸效果，爆炸效果主要是通过一个爆炸着色器，这个实现起来并不算太难，主要是几何着色器之前使用的较少，通过这次实现爆炸效果，我对渲染管线有了更深刻的认识。

总体来说，这次课程项目我主要的收获是在一个较为复杂，较大的框架中实现功能可能和一个简单的作业有着较大的区别。同时对渲染管线等计算机图形学中的概念有了更深刻的认识。