计算机图形学大程报告

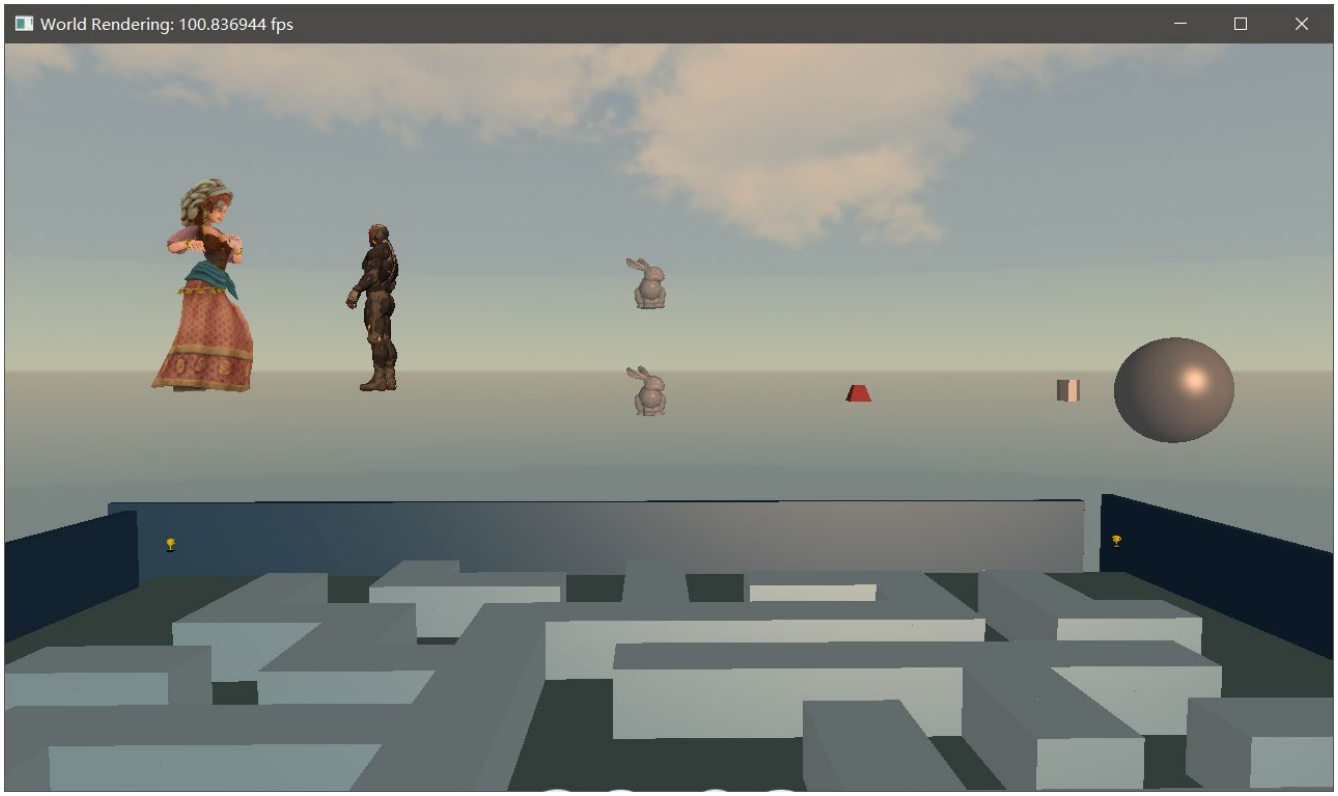
陈德瀚 3190102203
王子墨 3190100466
乔一帆 3190105134
范源颢 3180103574

1.总体说明

计算图形学期末大程，实现了一个第一人称走迷宫的游戏场景搭建，实现了所有的基础功能和额外功能中的 AABB 包围盒碰撞检验，在游戏性方面，实现了随机迷宫的生成。

基本操作说明:

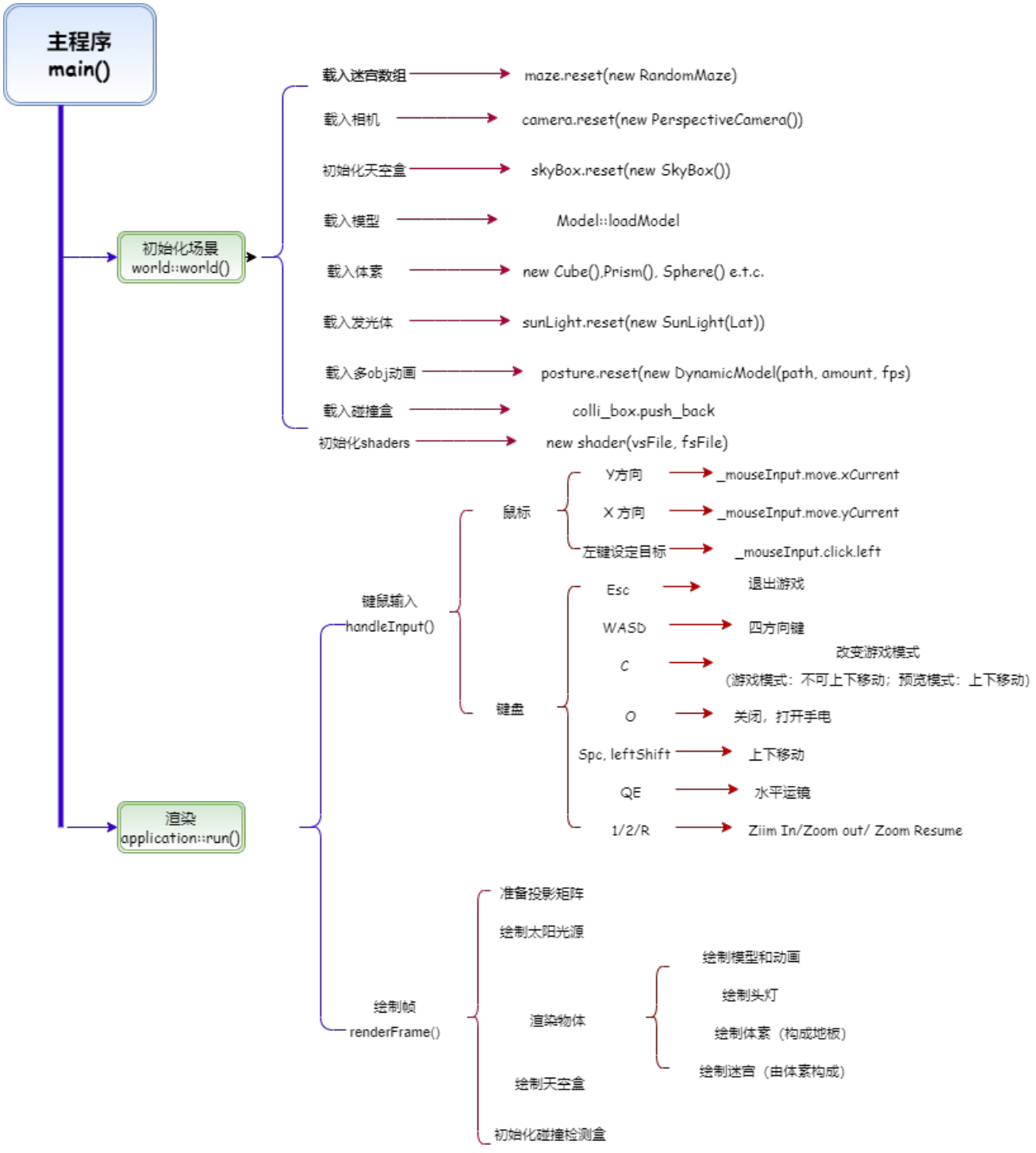
- 鼠标控制镜头朝向和移动方向;
- WASD 控制当前方向上的前进、后退、左右移动；空格键（Spc）竖直向上移动；左 shift 竖直向下移动; QE 表示摄像机向左/向右旋转;
- 1 拉近视角；2 拉远视角；R 恢复正常视角
- 将屏幕中心对准物体，按一下左键锁定（物体不能离得太远）；按住 4 围绕锁定点水平旋转；按 3 解除锁定
- 按 P 键截图，截图文件会保存在工程目录下的 snapshot.bmp
- 摄像机移动时会自动碰撞检测。注意按住 4 的旋转不会进行检测
- C 表示游戏状态切换：0 状态玩家不可进行上下移动，1 状态玩家方可进行上下移动，按 C 切换
- 实现双光源：运动光源太阳和玩家位置发出的手电光，手电光按 O 可关闭。



2.运行平台

Visual Studio 2019+, C++运行环境，请使用 x86 输出
需要使用 vcpkg 事先安装 glfw3, glad, glm 和 assimp

3.系统框架



4.分模块功能与实现

4.1 基本要求

由验收结果，全部完成

4.1.1 基本几何体体素建模

具有基本体素（立方体、球、圆柱、圆锥、多面棱柱、多面棱台）的建模表达能力。

在专门的文件夹：Basic_model 中，包含了两个子文件夹 include 和 source，分别包含了各种基本体素的头文件（.h）和源文件（.cpp），这些类都继承自 object3d

球体:

我们在 sphere.h 中制定了球的半径和经线、纬线的分辨率:

```
class Sphere : public Object3D
{

private:

    struct Material mat;

    /*OpenGL Data*/
    unsigned int VBO_Sphere = 0;
    unsigned int VAO_Sphere = 0;
    unsigned int EBO_Sphere = 0;

    /*Radius of Sphere*/
    float m_fRadius = 0.0f;
    /*Longitude Slices*/
    unsigned int m_unLongitudeSlices = 0;
    /*Latitude Slices*/
    unsigned int m_unLatitudeSlices = 0;

    std::vector<float> vecVertexPoints;
    std::vector<unsigned int> vecIndexPoints;
```

之后我们根据经纬网计算出球面的像素点:

```
void Sphere::CalculateSphere()
{
```

```
    //const unsigned int unSizeOfVertexArray = m_unLatitudeSlices *
m_unLongitudeSlices;
    //float* fVertexArrayPtr = new float[unSizeOfVertexArray];

    /*Define all the needed data to create a Sphere*/
    const float fPI = acosf(-1);
    /*Vertex Pos in 3D space*/
    float fX = 0.0f;
    float fY = 0.0f;
    float fZ = 0.0f;
    /*Data needed to calculate the needed indeces to draw 2 triangles in 1 sector of
the Sphere*/
    unsigned int unPoint1 = 0;
    unsigned int unPoint2 = 0;
    /*Angles THETA and PHI needed to calculate from degrees to 3D points*/
    float fLongitudeAngleTHETA = 0.0f;
    float fLatitudeAnglePHI = 0.0f;

    /*Calculating the steps needed to decide where and how many
      vertecies are needed to draw the Sphere*/
    float fLongitudeSteps = 2 * fPI / Sphere::m_unLongitudeSlices;
    float fLatitudeSteps = fPI / Sphere::m_unLatitudeSlices;

    /*Beginning with the Latitude*/
    for (unsigned int i = 0; i <= m_unLatitudeSlices; ++i)
    {
        /*Starting from 90 degrees and going to -90 degrees*/
        fLatitudeAnglePHI = fPI / 2 - i * fLatitudeSteps;
        /*Calculating Z*/
        /* (r * sin(PHI)) */
        fZ = m_fRadius * sinf(fLatitudeAnglePHI);

        /*Going to the Longitude*/
        for (unsigned int j = 0; j <= m_unLongitudeSlices; ++j)
        {
            /*Starting from 0 degrees and going to 360 degrees*/
            fLongitudeAngleTHETA = j * fLongitudeSteps;

            /*Calculating X*/
            /* (r * cos(PHI) * cos(THETA)) */
            fX = m_fRadius * cos(fLatitudeAnglePHI) * cos(fLongitudeAngleTHETA);
            /* (r * cos(PHI) * sin(THETA)) */
            fY = m_fRadius * cos(fLatitudeAnglePHI) * sin(fLongitudeAngleTHETA);
```

```
            vecVertexPoints.push_back(fX);
            vecVertexPoints.push_back(fY);
            vecVertexPoints.push_back(fZ);
        }
    }

    /*Calculating Indecies*/
    for (unsigned int i = 0; i < m_unLatitudeSlices; ++i)
    {
        unPoint1 = i * (m_unLongitudeSlices + 1);
        unPoint2 = unPoint1 + (m_unLongitudeSlices + 1);

        for (unsigned int j = 0; j < m_unLongitudeSlices; ++j, ++unPoint1, ++unPoint2)
        {
            if (i != 0)
            {
                vecIndexPoints.push_back(unPoint1);
                vecIndexPoints.push_back(unPoint2);
                vecIndexPoints.push_back(unPoint1 + 1);
            }

            if (i != (m_unLatitudeSlices - 1))
            {
                vecIndexPoints.push_back(unPoint1 + 1);
                vecIndexPoints.push_back(unPoint2);
                vecIndexPoints.push_back(unPoint2 + 1);
            }
        }
    }

}
```

圆柱体:

圆柱体的建模我们需要的是上方圆面、下方圆面的坐标，我们通过将圆柱体近似为多面柱体，分别绘制一个柱面的两个三角面片得到圆柱体的坐标信息。

```
    down_vertices[0] = glm::vec3(0.0f, -1.0f, 0.0f);
    up_vertices[0] = glm::vec3(0.0f, 1.0f, 0.0f);

    float pi = 3.1415;
    int num = 362;
```

```
    for (int i = 1;i <= num;i++) {
        float angle = i * 2 * pi / 360;
        down_vertices[i] = glm::vec3(cos(angle), -1.0f, sin(angle));
        up_vertices[i] = glm::vec3(cos(angle), 1.0f, sin(angle));
    }

    around_vertices[0] = down_vertices[1];
    int n = 726;
    for (int i = 1;i <= n;i+=2) {
        around_vertices[i] = down_vertices[(i + 1) / 2];
        around_vertices[i + 1] = up_vertices[(i + 1) / 2];
    }

    collision.cylinder_box(2.0f, 1.0f);
```

圆锥:

圆锥的建模主要依靠将锥底采样，和顶点组成三角面片，得到坐标

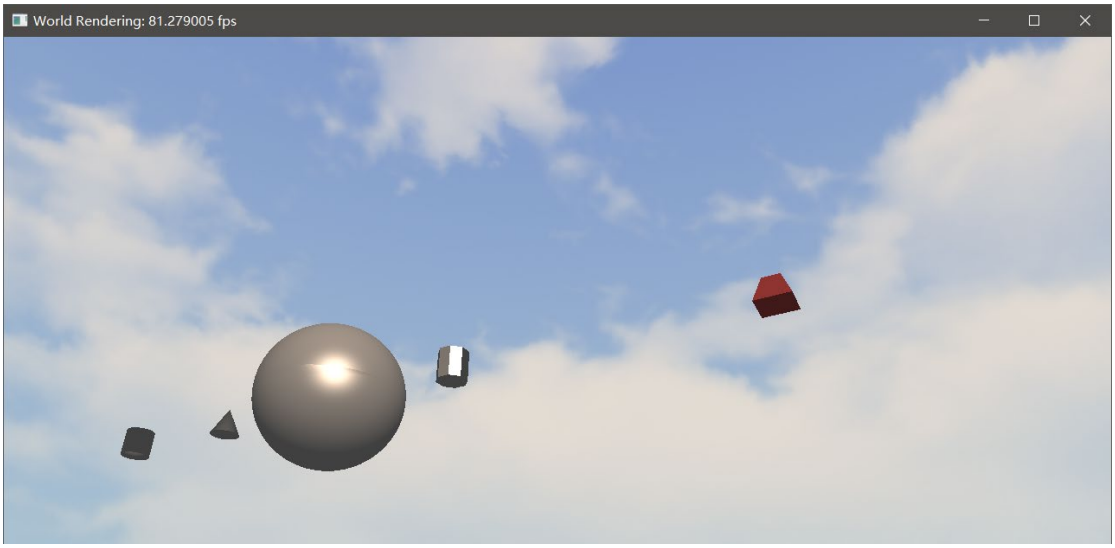
```
// set up OpenGL src
    vertices[0] = glm::vec3(0.0f, 1.0f, 0.0f);
    float pi = 3.1415;
    int num = 362;
    for (int i = 1;i <= num;i++) {
        float angle = i * 2 * pi / 360;
        vertices[i] = glm::vec3(cos(angle), -1.0f, sin(angle));
    }
```

多面体:

至于不包含球面的物体，我们直接计算他们的顶点位置即可，这里不再赘述。

效果:

下图是体素的合影。



4.1.2 基本三维网格导入

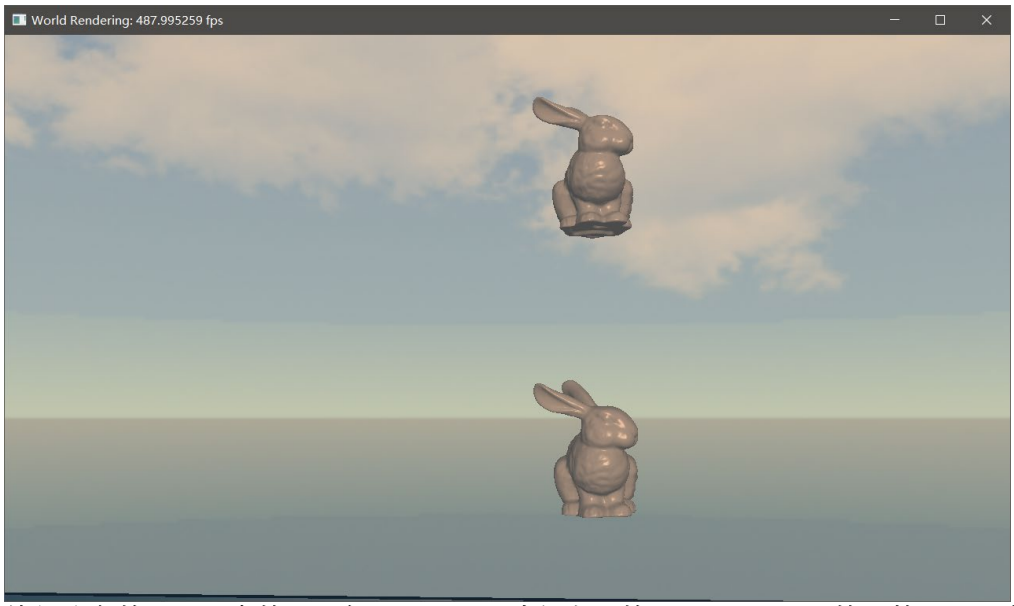
具有基本三位网络导入导出功能，不要求处理文件中的纹理和材质信息。

我们实现了将斯坦福小兔的三维模型通过网格化的方法导入的过程，相关的代码在 objmodel.cpp 中，我们首先通过解析文件名获得物体存储位置，之后我们通过 getline 直接读取字符 v、vn、f 读取相关信息：

```
ObjModel::ObjModel(string const& path) {
    ifstream file;
    file.open(path, ios::in);
    if (file.is_open() == false) {
        throw std::runtime_error("fail to open obj file\n");
    }

    char buffer[200];
    vector<glm::vec3> pos;
    vector<glm::vec3> nor;
    while (!file.eof()) {
        file.getline(buffer, 200);

        if (buffer[0] == 'v' && buffer[1] == ' ')
        {
            glm::vec3 t;
            sscanf_s(buffer, "v %f %f %f", &(t.x), &(t.y), &(t.z));
            pos.push_back(t);
        }
        else if (buffer[0] == 'v' && buffer[1] == 'n' && buffer[2] == ' ')
```

```
        {
            glm::vec3 n;
            sscanf_s(buffer, "vn %f %f %f", &(n.x), &(n.y), &(n.z));
            nor.push_back(n);
        }
        else if (buffer[0] == 'f' && buffer[1] == ' ')
        {
            unsigned int p1, n1, p2, n2, p3, n3;
            struct ver temp;
            sscanf_s(buffer, "f %u//%u %u//%u %u//%u", &p1, &n1, &p2, &n2, &p3, &n3);

            temp.pos = pos[p1-1];
            temp.normal = nor[n1-1];
            vertices.push_back(temp);
            temp.pos = pos[p2-1];
            temp.normal = nor[n2-1];
            vertices.push_back(temp);
            temp.pos = pos[p3-1];
            temp.normal = nor[n3-1];
            vertices.push_back(temp);
        }
    }
    initGL();
    return;
}
```

效果如下：



值得注意的是，图中的两只兔子，一只是我们自己的 objloader 导入的函数，另一个是借助第三方库 assimp 导入的结果，我们通过对比二者的效果判断自己的 obj 文件导入函数是否正确，如图，我们的效果和第三方库 assimp 相比已经可以以假乱真。

出了斯坦佛小兔之外，我们还实现了纳米套装和奖杯的模型导入，具体可参考本文首图

4.1.3 基本材质，纹理显示和编辑能力

纹理：以天空盒为例

我们在小实验中实现了天空盒，为此，我们将相关的技术迁移到了大实验中。在本次大程中，我同样适用天空盒作为一种立方体纹理的实现方式。

我们首先在 manualtexture.cpp 中实现 TextureCubemap 类，在其初始化时，我们根据初始化的文件路径参数构建出立方体纹理

```
TextureCubemap::TextureCubemap(const std::vector<std::string>& filenames)
    : _paths(filenames) {
    assert(filenames.size() == 6);

    // bind the textureID
    glBindTexture(GL_TEXTURE_CUBE_MAP, _handle);
    std::cout << "this texture is " << _handle << std::endl;
    // load data into faces
    int width, height, channels;
    unsigned char* data;
    for (unsigned int i = 0; i < filenames.size(); i++) {

        // bmp image doesn't need flip, other images should flip
        stbi_set_flip_vertically_on_load(true);
        if (_paths[i].find("bmp") <= _paths[i].length()) {

            stbi_set_flip_vertically_on_load(false);
        }

        data = stbi_load(_paths[i].c_str(), &width, &height, &channels, 0);
        if (data) {
            glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB, width, height,
0, GL_RGB, GL_UNSIGNED_BYTE, data);
            stbi_image_free(data);
            std::cout << "loading data from" << _paths[i].c_str() << std::endl;
        }
        else {
            std::stringstream ss;
            ss << "texture object operation failure at path" << _paths[i].c_str();
            cleanup();
            throw std::runtime_error(ss.str());
            stbi_image_free(data);
        }
```

```
    }
    // set other details
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
    glBindTexture(GL_TEXTURE_CUBE_MAP, 0);
    // --------------------------------------------------
}
```

之后，天空盒类（skybox）将根据已经实现好的 TextureCubemap 函数，绑定 vao,vbo, 并且设定相关的 shader.glsl 代码

```
    try {
        // init texture
        _texture.reset(new TextureCubemap(textureFilenames));

        const char* vertCode =
            "#version 330 core\n"
            "layout(location = 0) in vec3 aPosition;\n"
            "out vec3 texCoord;\n"
            "uniform mat4 projection;\n"
            "uniform mat4 view;\n"
            "void main() {\n"
            "   texCoord = aPosition;\n"
            "   gl_Position = (projection * view * vec4(aPosition, 1.0f)).xyww;\n"
            "}\n";

        const char* fragCode =
            "#version 330 core\n"
            "out vec4 color;\n"
            "in vec3 texCoord;\n"
            "uniform samplerCube cubemap;\n"
            "uniform vec4 duskColor;\n"
            "void main() {\n"
            "   color = texture(cubemap, texCoord) * duskColor;\n"
            "}\n";

        _shader.reset(new Shader(vertCode, fragCode));
    }
    catch (const std::exception&) {
        cleanup();
        throw;
        }
```

效果如之前截图中的背景所示，这里就不再赘述了。

材质

为了反映物体的发光情况，我们给每个物体都设置了材质属性，包括其反光度、反光颜色等等，主要设置在头文件 material.h 中：

```
#pragma once
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

struct Material {
    //材质颜色光照
    glm::vec4 Ka;
    //漫反射
    glm::vec4 Kd;
    //镜反射
    glm::vec4 Ks;
    float shininess;
};
```

之后，以正方体为例，我们通过接口将改变物体材质的函数 setKa, setKd 等开放出来，以便我们进行修改.

```
class Cube : public Object3D {
public:
    Cube();
    ~Cube();

    void Draw(Shader &shader);

    inline void SetKa(glm::vec4 ka) { mat.Ka = ka; }
    inline void SetKd(glm::vec4 kd) { mat.Kd = kd; }
    inline void SetKs(glm::vec4 ks) { mat.Ks = ks; }
    inline void SetShiniess(float shininess) { mat.shininess = shininess; }

    AABB collision;

private:
    // notice that the normal data hasn't been set

    struct Material mat;
……省略内容
}
```

这样我们就可以通过这一接口进行颜色的改变了，如下：

```
cube->SetKa(glm::vec4(25.0f / 255.0f, 51.0f / 255.0f, 76.0f / 255.0f, 1.0f));
cube->SetKd(glm::vec4(0x33 / 255.0f, 0x66 / 255.0f, 0x99 / 255.0f, 1.0f));
```

当然，相应的，shader 代码中也会需要进行带材质的光照运算：

```
vec3 diffuse = spotLight.color * max(dot(lightDir, normal), 0.0f) * vec3(material.kd);
vec3 specular = spotLight.color * pow(max(dot(viewDir, reflectDir), 0.0f),
material.shininess) * vec3(material.ks);
float distance = length(spotLight.position - FragPos);
float attenuation = 1.0f / (spotLight.kc + spotLight.kl * distance + spotLight.kq *
distance * distance);
return spotLight.intensity * attenuation * (diffuse + specular);
```

diffuse 和 specular 分别表示漫反射光照和镜面反射光照。

4.1.4 基本几何变换

旋转、平移、缩放等

我们在 world.cpp 中调用物体的进行绘制的时候,通常都会需要改变物体的位置、大小,为此，我们在所有三维物体的基类 object3d.h 中设定了平移、旋转和缩放的向量和四元数:

```
#pragma once
#define PI 3.1415926f

#include <glm/glm.hpp>
#include <glm/ext.hpp>

class Object3D {
public:
    glm::vec3 position = { 0.0f, 0.0f, 0.0f };
    // q = ( cos(theta/2), (x, y, z)sin(theta/2))
    // 绕轴（x，y，z）旋转theta角
    glm::quat rotation = { 1.0f, 0.0f, 0.0f, 0.0f };
    glm::vec3 scale = { 1.0f, 1.0f, 1.0f };

    glm::vec3 getFront() const;

    glm::vec3 getUp() const;

    glm::vec3 getRight() const;

    glm::mat4 getModelMatrix() const;
};
```

我们在 world.cpp 中通过修改相关向量进行平移和缩放

```
cube->scale = glm::vec3(100.0f, 1.0f, 100.0f);
cube->position = glm::vec3(0.0f, -1.0f, 0.0f);
cube->Draw(*basicShader);
```

在最后绘图的时候，Draw 函数会根据设定的旋转、平移和缩放向量得到变换矩阵，最后将此内容发射到 shader 中。

```
void Cube::Draw(Shader& shader) {
```

```
    glDepthFunc(GL_LESS);
    shader.use();
    shader.setVec4("material.ka", mat.Ka);
    shader.setVec4("material.kd", mat.Kd);
    shader.setVec4("material.ks", mat.Ks);
    shader.setFloat("material.shininess", mat.shininess);
    shader.setMat4("model", getModelMatrix());

    glBindVertexArray(vao);
    glDrawArrays( GL_TRIANGLES, 0, 36 );
    glBindVertexArray(0);
}
```

其中:

```
glm::mat4 Object3D::getModelMatrix() const {
    return glm::translate(glm::mat4(1.0f), position) *
        glm::mat4_cast(rotation) *
        glm::scale(glm::mat4(1.0f), scale);
}
```

4.1.5 基本光照模型和光源编辑

调整光源的位置、光强等参数

本实验中，我们分别实现了平行光源和聚光光源，二者的属性都可编辑。

平行光

实验中，我们通过地理信息知识实现了太阳光，太阳光类（SunLight）继承自平行光类（DirectionalLight），我们通过上文中提到的基本体素和几何变换的知识，可以比较轻易的绘制出球形的太阳，并模拟日出日落。

另一方面，我们需要根据太阳高度角的计算知道平行太阳光的方向，相关的代码如下:

```
    void SunLight::updateLight(const float& deltaTime)
{
    subsolarLongitude += deltaTime * angularVelocity;
    innerXYZ = { CELESTIAL_RADIUS * cos(this->subsolarLatitude) * (-1) *
sin(this->subsolarLongitude),
            CELESTIAL_RADIUS * sin(this->subsolarLatitude),
            CELESTIAL_RADIUS * cos(this->subsolarLatitude) *
cos(this->subsolarLongitude)
    };
    position =  innerXYZ * rotateMat;
    glm::vec3 top = { 0.0f, 1.0f, 0.0f };
```

```
    float theta = acos(glm::dot(top, position) / glm::length(position));
    elevationAngle = acos(0) - theta;



    float sinvalue = sin(elevationAngle + CIVILIAN_TWILIGHT_ANGLE);
    if (sinvalue > 0) {
        intensity = /*sunIntensityBase * */ pow(sinvalue, 0.4f);
        color = glm::vec3(pow(sinvalue, 0.005f),
        pow(sinvalue, 0.25f),
        pow(sinvalue, 0.5f));
    }
    else {
        intensity = 0;
        color = { 0.0, 0.0, 0.0 };
    }
}
```

之后，我们只需要在 world.cpp 中读取时间即可模拟日出日落，实现太阳光方向属性的修改，其中，相关宏的意义如下：

```
#define CELESTIAL_RADIUS 1000.0f // 太阳距离，
#define CIVILIAN_TWILIGHT_ANGLE 0.09f // 以弧度计民用晨昏蒙影
```

聚光

聚光我们的实现方式是从摄像头发出的电光，小实验中，我们已经实现过相关内容，我们在这里也和当时一样，在 shader 的 GLSL 代码中假如全局的聚光结构：

```
// spot light data structure declaration
struct SpotLight {
    vec3 position;
    vec3 direction;
    float intensity;
    vec3 color;
    float angle;
    float kc;
    float kl;
    float kq;
};
```

之后我们直接在 world.cpp 中实现从摄像头发出的手电光：

```
// 点灯位于眼睛上方，微微向下倾斜。如果恰在眼睛处，则视野中始终为正圆光斑，是不行的。
    if (switch_on) {
        basicShader->setVec3("spotLight.position", eyes + 0.5f *
glm::normalize(camera->getUp()));
        basicShader->setVec3("spotLight.direction", camera->getFront() - 0.1f *
glm::normalize(camera->getUp()));
```

```
        // 光照设定
        basicShader->setFloat("spotLight.intensity", 10.0f);
        basicShader->setVec3("spotLight.color", glm::vec3(1.0f, 1.0f, 1.0f));
        basicShader->setFloat("spotLight.angle", 0.2f);
        basicShader->setFloat("spotLight.kc", 1.0f);
        basicShader->setFloat("spotLight.kl", 0.0f);
        basicShader->setFloat("spotLight.kq", 0.2f);
    }
    else
    {
        basicShader->setVec3("spotLight.position", eyes + 0.5f *
glm::normalize(camera->getUp()));
        basicShader->setVec3("spotLight.direction", camera->getFront() - 0.1f *
glm::normalize(camera->getUp()));
        // 光照设定
        basicShader->setFloat("spotLight.intensity", 0.0f); //-1.0f + 0.0f
        basicShader->setVec3("spotLight.color", glm::vec3(1.0f, 1.0f, 1.0f));
        basicShader->setFloat("spotLight.angle", 0.2f);
        basicShader->setFloat("spotLight.kc", 1.0f);
        basicShader->setFloat("spotLight.kl", 0.0f);
        basicShader->setFloat("spotLight.kq", 0.2f);
}
```

这样一来，结合我们在 4.1.3 中设定的物体材料反光参数，shader 可以由此绘出光照效果。

4.1.6 场景漫游

我们设计一个摄像机类，其中派生出两种摄像机：正交摄像机和透视摄像机：

```
class Camera : public Object3D {
public:
    glm::mat4 getViewMatrix() const;
    virtual glm::mat4 getProjectionMatrix() const = 0;
};


class PerspectiveCamera : public Camera {
public:
    float fovy;
    float aspect;
    float znear;
    float zfar;
public:
    PerspectiveCamera(float fovy, float aspect, float znear, float zfar);
```

```
    ~PerspectiveCamera() = default;

    glm::mat4 getProjectionMatrix() const override;
};


class OrthographicCamera : public Camera {
public:
    float left;
    float right;
    float bottom;
    float top;
    float znear;
    float zfar;
public:
    OrthographicCamera(float left, float right, float bottom, float top, float znear,
float zfar);

    ~OrthographicCamera() = default;

    glm::mat4 getProjectionMatrix() const override;
};
```

实验中，我们通常使用透视摄像机，在设定照相机位置和角度之后，我们可以构建透视变换的矩阵，将相关的世界坐标转换为我们在屏幕上实现的坐标系。在进行摄像机移动时，我们这里只需要将摄像机当作一个普通的物体进行设置就可以了，以按 D 向右移动为例:

```
if (_keyboardInput.keyStates[GLFW_KEY_D] != GLFW_RELEASE) {
        //camera->position += cameraMoveSpeed * camera->getRight();
        if (state == 0) {
            glm::vec3 direction = camera->getRight();
            direction.y = 0.0f;
            direction = glm::normalize(direction);
            CameraCollisionCheck(camera->position, cameraMoveSpeed * direction);
        }
        else {
            CameraCollisionCheck(camera->position, cameraMoveSpeed *
camera->getRight());
        }
}
```

其中 state 是控制游戏状态的函数，0 状态下不可进行上下移动，因此我们去除摄像机移动的上下分量。

水平运镜的实现如下，我们需要修改照相机的角度:

```
//QE实现pan（水平运镜）
    if (_keyboardInput.keyStates[GLFW_KEY_Q] != GLFW_RELEASE) {
```

```
        glm::quat temp_rotation = { 1.0f * cos(deltaAngle), 0.0f, 1.0f *
sin(deltaAngle), 0.0f };
        camera->rotation = temp_rotation * camera->rotation;
    }

    if (_keyboardInput.keyStates[GLFW_KEY_E] != GLFW_RELEASE) {
        glm::quat temp_rotation = { 1.0f * cos(-deltaAngle), 0.0f, 1.0f * sin(-
deltaAngle), 0.0f };
        camera->rotation = temp_rotation * camera->rotation;
    }
```

视场缩放（Zoom In/Zoom out）,我们只需要调整 fovy 即可，注意我们给视场的调整设置了一定的限制。

```
// 按1键进行Zoom In
    if (_keyboardInput.keyStates[GLFW_KEY_1] != GLFW_RELEASE) {
        if (camera->fovy > 0.0174533) {
            camera->fovy -= deltaFovy;
        }
    }

    // 按2键进行Zoom Out
    if (_keyboardInput.keyStates[GLFW_KEY_2] != GLFW_RELEASE) {
        if (camera->fovy < 3.1241393) {
            camera->fovy += deltaFovy;
        }
    }
```

之后，我们调用 glm 自带的 perspective(fovy, aspect, znear, zfar)即构造出透视变换的矩阵，作为 shader 的接受变量即可。

最后，我们也实现了 Zoom to Fit 和 Orbit，相关代码如下：

```
// 按3键进行Zoom to Fit，使用一次后该目标失效
    if (_keyboardInput.keyStates[GLFW_KEY_3] != GLFW_RELEASE) {
        if (setTarget) {
            camera->position += 0.5f * viewDir;
            setTarget = false;
        }
    }

    // 按4键进行Orbit：根据目标点的z与x坐标发出竖直轴，相机自身位置绕该轴旋转
    if (_keyboardInput.keyStates[GLFW_KEY_4] != GLFW_RELEASE) {
        if (setTarget) {
            //glm::vec2 dxz = glm::mat2x2(cos(deltaAngle), sin(deltaAngle), -
sin(deltaAngle), cos(deltaAngle)) * glm::vec2(camera->position.x - target.x,
camera->position.z - target.z);
            camera->position.x = cos(deltaAngle) * (camera->position.x - target.x) +
sin(deltaAngle) * (camera->position.z - target.z) + target.x;
```

```
            camera->position.z = -sin(deltaAngle) * (camera->position.x - target.x) +
cos(deltaAngle) * (camera->position.z - target.z) + target.z;
            glm::quat temp_rotation = { 1.0f * cos(0.5f * deltaAngle), 0.0f, 1.0f *
sin(0.5f * deltaAngle), 0.0f };
            camera->rotation = temp_rotation * camera->rotation;
        }
    }
```

4.1.7动画播放：连续 obj 模型文件读取及截屏

动画播放

我们在读取连续的 obj 模型时，建立了独立的 DynamicModel 类进行处理，其基本思想是以此调用Model类的构造函数得出一帧帧的静态模型,储存在Model构成的向量容器中，然后在通过静态模型的的绘图函数，按照一定的时间频率绘制出来:

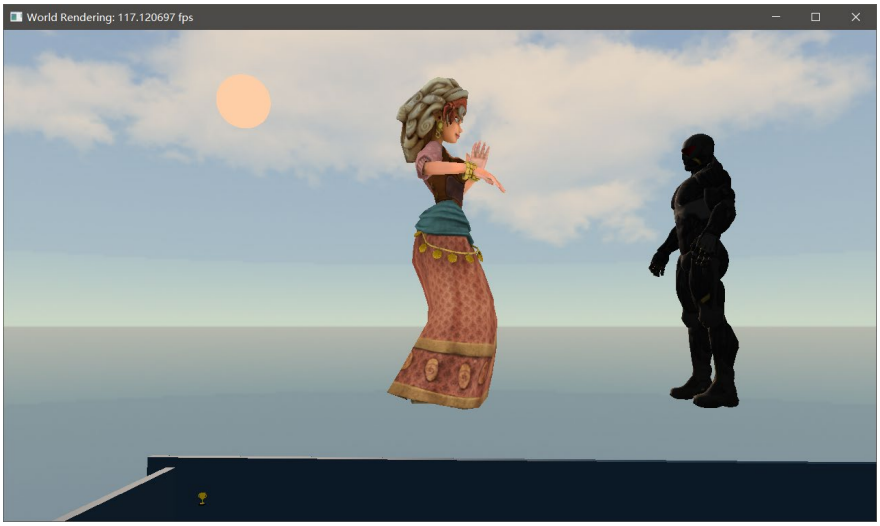
```
DynamicModel::DynamicModel(string const& path, int amount, float fps) :amount(amount),
dt(1.0f/fps)
{
    //string temp = path;
    int digitlength = (int)(log10(amount) + 1);
    int templength;
    for (int i = 0; i < amount; i++) {
        char* num = new char[digitlength];
        for (int j = 0; j < digitlength; j++) {
            num[j] = '0';
        }
        if (i == 0) {
            templength = 1;
        }
        else {
            templength = (int)(log10(i) + 1);
        }
        char* tmp = num + (digitlength - templength);
        _itoa_s(i, tmp, sizeof(tmp) , 10);
        string localtemp = path + num + ".obj";
        std::cout << localtemp << std::endl;
        m.push_back(Model(localtemp));
    }
}

void DynamicModel::Draw(Shader& shader, float time)
{
```

```
    m[(int)(time / dt) % amount].Draw(shader);
}
```

我们有:

```
class DynamicModel : public Object3D
{
public:
    DynamicModel(string const& fragpath, int amount, float fps);
    void Draw(Shader& shader, float time);
    void setPosition(glm::vec3 ps);
    void setRotation(glm::quat rt);
    void setScale(glm::vec3 sc);

private:
    glm::vec3 globalPosition = { 0.0f, 0.0f, 0.0f };
    glm::quat globalRotation = { 1.0f, 0.0f, 0.0f, 0.0f };
    glm::vec3 globalScale = { 1.0f, 1.0f, 1.0f };
    int amount;
    float dt;
    vector<Model> m;
};
```

效果如下:



截屏

截屏的实现如下所示，我们没有将其设定到独立的函数中，按住 P 键，我们会把截图储存在工程目录下的 snapshot.bmp

```
// 按P键截屏
    if (_keyboardInput.keyStates[GLFW_KEY_P] != GLFW_RELEASE) {
        FILE* pDummyFile;  //指向另一bmp文件，用于复制它的文件头和信息头数据
        FILE* pWritingFile;  //指向要保存截图的bmp文件
        GLubyte* pPixelData;    //指向新的空的内存，用于保存截图bmp文件数据
        GLubyte  BMP_Header[BMP_Header_Length];
        GLint    i, j;
        GLint    PixelDataLength;   //BMP文件数据总长度

        // 计算像素数据的实际长度
        i = _windowWidth * 3;   // 得到每一行的像素数据长度
        while (i % 4 != 0)      // 补充数据，直到i是的倍数
            ++i;
        PixelDataLength = i * _windowHeight;  //补齐后的总位数

        // 分配内存和打开文件
        pPixelData = (GLubyte*)malloc(PixelDataLength);
        if (pPixelData == 0)
            exit(0);

        fopen_s(&pDummyFile, "bitmapheader.bmp", "rb");

        fopen_s(&pWritingFile, "snapshot.bmp", "wb");
```

```
        //把读入的bmp文件的文件头和信息头数据复制，并修改宽高数据
        fread(BMP_Header, sizeof(BMP_Header), 1, pDummyFile);  //读取文件头和信息头，
占据54字节
        fwrite(BMP_Header, sizeof(BMP_Header), 1, pWritingFile);
        fseek(pWritingFile, 0x0012, SEEK_SET); //移动到0X0012处，指向图像宽度所在内存
        i = _windowWidth;
        j = _windowHeight;
        fwrite(&i, sizeof(i), 1, pWritingFile);
        fwrite(&j, sizeof(j), 1, pWritingFile);

        // 读取当前画板上图像的像素数据
        glPixelStorei(GL_UNPACK_ALIGNMENT, 4);  //设置4位对齐方式
        glReadPixels(0, 0, _windowWidth, _windowHeight, GL_BGR, GL_UNSIGNED_BYTE,
pPixelData);

        // 写入像素数据
        fseek(pWritingFile, 0, SEEK_END);
        //把完整的BMP文件数据写入pWritingFile
        fwrite(pPixelData, PixelDataLength, 1, pWritingFile);

        // 释放内存和关闭文件
        fclose(pDummyFile);
        fclose(pWritingFile);
        free(pPixelData);
    }
```

4.2 额外要求

我们实现了额外要求中的碰撞检测，其主要的实现思路是 AABB 碰撞盒，游戏性方面，我们则主要实现了随机生成的迷宫，利用的算法是 Prim 算法

4.2.1 碰撞检测

通过给物体设立包围她的碰撞盒，将所有物体的碰撞盒放置到一个全局的碰撞盒向量中，这样，在渲染时，我们只要判断这些向量中，碰撞盒是否重叠，就可以监测物体之间是否存在碰撞。

首先，当我们在绘图之前，我们可以根据物体的变换矩阵直接得出物体的碰撞盒位置，物体的变换矩阵可以直接给 shader 确定物体的位置，我们也可以用相关的函数得到包围她的碰撞盒的位置。

```
void AABB::update_box(glm::mat4 model_matrix) {
    std::vector<glm::vec4> point(8);
    point[0] = glm::vec4(x_min, y_min, z_min, 1.0f);
```

```
    point[1] = glm::vec4(x_min, y_min, z_max, 1.0f);
    point[2] = glm::vec4(x_min, y_max, z_min, 1.0f);
    point[3] = glm::vec4(x_min, y_max, z_max, 1.0f);
    point[4] = glm::vec4(x_max, y_min, z_min, 1.0f);
    point[5] = glm::vec4(x_max, y_min, z_max, 1.0f);
    point[6] = glm::vec4(x_max, y_max, z_min, 1.0f);
    point[7] = glm::vec4(x_max, y_max, z_max, 1.0f);

    // calculate range
    for (auto p = point.begin(); p != point.end(); p++) {
        *p = model_matrix * *p;
    }
    x_range[0] = x_range[1] = point[0].x;
    y_range[0] = y_range[1] = point[0].y;
    z_range[0] = z_range[1] = point[0].z;
    for (auto p = point.begin()+1; p != point.end(); p++) {
        x_range[0] = fmin(x_range[0], p->x);
        x_range[1] = fmax(x_range[1], p->x);
        y_range[0] = fmin(y_range[0], p->y);
        y_range[1] = fmax(y_range[1], p->y);
        z_range[0] = fmin(z_range[0], p->z);
        z_range[1] = fmax(z_range[1], p->z);
    }

    // calculate center
    center.x = (x_range[0] + x_range[1]) / 2.0f;
    center.y = (y_range[0] + y_range[1]) / 2.0f;
    center.z = (z_range[0] + z_range[1]) / 2.0f;

}
```

所需的参数 model_matrix 本质上就是物体的变换矩阵，我们在 world.cpp 中调用时,对于任何一个需要绘制的模型 model，有：

```
model->colli_box.update_box(nanosuit->getModelMatrix());
colli_box.push_back(nanosuit->colli_box);
```

其中 colli_box 是我们设定的碰撞盒 AABB 类的向量对象（vector<AABB>）.

因为在我们的游戏中，运动的对象只有摄像机，所以我们只需要检验照相机和周围的物体的 AABB 碰撞盒是否有重叠即可，相关的函数如下：

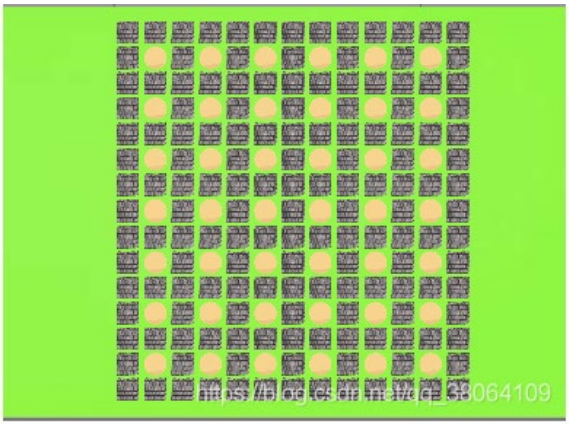
```
void world::CameraCollisionCheck(glm::vec3& camera_pos, glm::vec3 move)
{
    glm::vec3 dest = camera_pos + move;

    for (auto ibox = colli_box.begin(); ibox != colli_box.end(); ibox++) {
        bool is_collision =
            (dest.x > ibox->get_x_range().x - camera->znear) && (dest.x <
```

```
ibox->get_x_range().y + camera->znear) &&
            (dest.y > ibox->get_y_range().x - camera->znear) && (dest.y <
ibox->get_y_range().y + camera->znear) &&
            (dest.z > ibox->get_z_range().x - camera->znear) && (dest.z <
ibox->get_z_range().y + camera->znear);
        if (is_collision) {
            cout << "collision" << endl;
            return;
        }
    }
    camera_pos = camera_pos + move;
}
```

一旦发生碰撞，我们会有标准输出 collision 的警告，并且不再允许摄像机继续沿同一方向移动。

4.2.2 游戏性：随机迷宫生成

实验还实现了一个随机生成的迷宫，主要实现在 basic_mode 文件夹下的 random_maze.cpp 中，我们使用的是 prime 算法，首先将考虑一个二维数组绘制出如下所示的栅格



其中方形的是墙，圆形的是玩家可以存在的空间（在数组中分别用 1,0 表示），迷宫的目的是实现一个联通所有原型的树，并且把联通线上的墙打通（把数组中的 1 变成 0）。在玩家所在的位置出发，我们可以通过随机去往相邻的，未曾被访问过的圆形区域的方法，尝试构建树，如果发现四个相邻的圆形区域都已被访问，则随机从之前访问过的圆形区域中找到一个元素，作为新的出发点重复上述过程，直到所有的圆形区域都被穷尽。

相关代码实现如下：

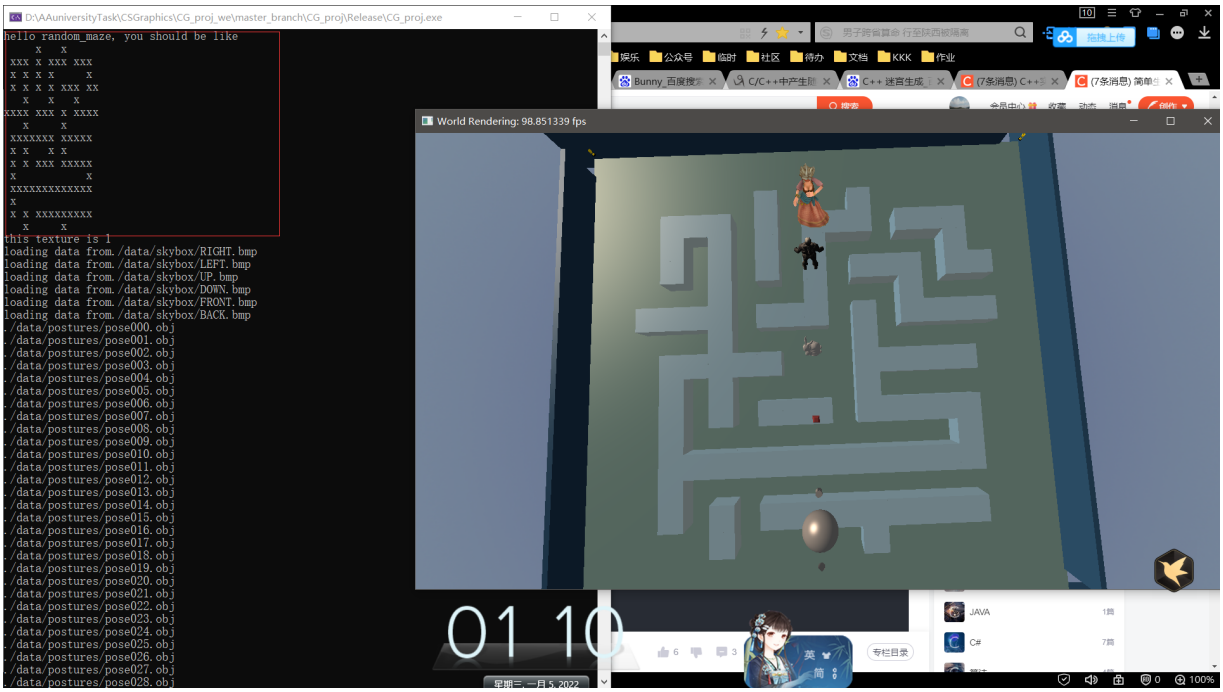
```
void RandomMaze::constructRandomMaze() {
    int count = row * column;
    int accsize = 0;
    int acc[64], noacc[64]; //acc for accessed, and noacc for non-accessed
    //64 = count = row * column = 8 * 8
```

```
    int offR[4] = { -1, 1, 0, 0 };//偏移量，4个量分别表示上下左右
    int offC[4] = { 0, 0, 1, -1 };//
    int offS[4] = { -1, 1, row, -row };

    for (int i = 0; i < count; i++)
    {
        acc[i] = 0;
        noacc[i] = 0;//开始所有点没有被访问
    }

    //起点（一直在中心）
    acc[0] = 36;
    int pos = acc[0];

    //第一个点存入
    noacc[pos] = 1;
    while (accsize < count)
    {
        //取出当下的点
        int x = pos % row;
        int y = pos / row;
        int offpos = -1;//用来记录偏移量
        int dir = 0; //标记便宜的方向

        while (++dir < 5)
        {
            //随机访问最近的点
            int point = (rand() % (4 - 0)) + 0;//[0,4)
            int repos;
            int move_x, move_y;
            //计算位移方向
            repos = pos + offS[point];
            move_x = x + offR[point];
            move_y = y + offC[point];

            //判断位移是否合法
            if (move_y > -1 && move_x > -1 && move_x < row && move_y < column &&
repos >= 0 && repos < count && noacc[repos] != 1)
            {
                noacc[repos] = 1;
                acc[++accsize] = repos;
                pos = repos;
                offpos = point;
```

```
                //相邻的格子中间打通
                MazeId[2 * x + 1 + offR[point]][2 * y + 1 + offC[point]] = 0;
                break;
            }
            else
            {
                if (accsize == count -1)
                {
                    return;
                }
                continue;
            }
        }

        if (offpos < 0)
        {//周边没有路了，从走过的路里重新找个起点
            int index = rand() % ((accsize + 1) - 0);
            pos = acc[index];
        }
    }
}
```

之后，我们在 world.cpp 中，根据 MazeId 拜访正方体即可构成墙壁。

效果如下：



我们会在标准输出中打出迷宫的形式和实际对比。我们发现，除了中心部分之外，整个迷宫是按照标准输出的提示构造的，至于中心部分的不同，是为了防止游戏开局卡死，在中心部分强制禁止墙壁的生成而设定的

5.其他

1. 这次实验我们指定在 Windows 平台下的 visual studio 中实现，尚未研究其在其他操作系统中的实现方式，在 Linux 和 MacOS 中的实现可能需要 cmake 进行跨平台支持，但是本次实验就不必再实现了
2. 在 visual studio 进行调试时，我们可以把 debug 模式调整为 release 模式，并且按下 Ctrl + F5 (开始执行，不调试)减少调试信息的输出，使得编译更加迅速。