个人报告

年级	**16级**	**专业（方向）**	**软件工程（电子政务）**
学号	**16340036**	**姓名**	**陈曦**

实现内容

天空盒以及文本渲染。

天空盒及文本渲染最终效果如下：

Sky Box（天空盒）：

天空盒是一个包含了整个场景的立方体，它包含周围环境的6个图像，让玩家以为他处在一个比实际大得多的环境当中，就像一片天空一样，但是玩家永远到不了天边，永远就在这片天空里面。所谓的天空盒其实就是将一个立方体展开，然后在六个面上贴上相应的贴图，在实际的渲染中，将这个立方体始终罩在摄像机的周围，让摄像机始终处于这个立方体的中心位置，然后根据视线与立方体的交点的坐标，来确定究竟要在哪一个面上进行纹理采样。

首先在网上找了几份六张天空盒素材照片，最后根据与篮球场适配度选择了下面的素材：

参考LearnOpenGL网站中有关天空盒部分，通过使用下面的函数加载天空盒：

```
unsigned int loadCubemap(vector<std::string> faces)
{
    unsigned int textureID;
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);

    int width, height, nrChannels;
    for (unsigned int i = 0; i < faces.size(); i++)
    {
        unsigned char *data = stbi_load(faces[i].c_str(), &width, &height, &nrChannels, 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);
        }
        else
        {
            std::cout << "Cubemap texture failed to load at path: " << faces[i] << std::endl;
            stbi_image_free(data);
        }
    }
    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);

    return textureID;

}
```

在调用这个函数之前，将合适的纹理路径按照立方体贴图枚举指定的顺序加载到一个vector中，如下：

```
vector<std::string> faces
{
    "../resource/resources/textures/skybox3/right.jpg",
    "../resource/resources/textures/skybox3/left.jpg",
    "../resource/resources/textures/skybox3/top.jpg",
    "../resource/resources/textures/skybox3/bottom.jpg",
    "../resource/resources/textures/skybox3/front.jpg",
    "../resource/resources/textures/skybox3/back.jpg"
};
cubemapTexture = loadCubemap(faces);
```

Display Text (显示文字，英文/平面):

参考LearnOpenGL网站中文本渲染部分，本次项目使用了FreeType库去渲染文本（“Bastekball”），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。

在项目里实现了加载一个从**Windows/Fonts**目录中拷贝来的TrueType字体文件**Inkfree.ttf**。首先初始化FreeType库，并将此字体加载为Face。

```
// FreeType
FT_Library ft = NULL;
// 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 = NULL;
if (FT_New_Face(ft, "C:/Windows/Fonts/Inkfree.ttf", 0, &face))
    std::cout << "ERROR::FREETYPE: Failed to load font" << std::endl;
```

定义结构体Character，并将其存储在一个map中，在需要渲染字符的时候调用。

```
struct Character {
    GLuint TextureID;
    glm::ivec2 Size;
    glm::ivec2 Bearing;
    GLuint Advance;
};

std::map<GLchar, Character> Characters;
```

对ASCII字符集的前128个字符中的每一个字符生成一个纹理并保存相关数据到Character结构体中，然后再添加至Characters这个映射表中，以存储渲染一个字符所需的所有数据备用。

```
// Disable byte-alignment restriction
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

// Load first 128 characters of ASCII set
for (GLubyte c = 0; c < 128; c++)
{
    // Load character glyph
    if (FT_Load_Char(face, c, FT_LOAD_RENDER))
    {
        std::cout << "ERROR::FREETYTPE: Failed to load Glyph" << std::endl;
        continue;
    }
    // Generate texture
```

```
            GLuint texture;
            glGenTextures(1, &texture);
            glBindTexture(GL_TEXTURE_2D, texture);
            glTexImage2D(
                GL_TEXTURE_2D,
                0,
                GL_RED,
                face->glyph->bitmap.width,
                face->glyph->bitmap.rows,
                0,
                GL_RED,
                GL_UNSIGNED_BYTE,
                face->glyph->bitmap.buffer
            );
            // Set texture options
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
            // 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));
        }
```

创建一个VBO和VAO用来渲染四边形

```
        // Configure VAO/VBO for texture quads
        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);
```

创建RenderText函数来渲染一个字符串，首先从之前创建的Characters映射表中取出对应的Character结构体，并根据字符的度量值（使用scale进行了缩放）来计算四边形的原点坐标（xpos和ypos）和它的大小（w和h），并生成6个顶点形成这个2D四边形，然后使用glBufferSubData函数更新VBO所管理内存的内容并渲染了这个四边形，代码如下：

```
    void RenderText(Shader shader, std::string text, GLfloat x, GLfloat y, GLfloat scale,
glm::vec3 color)
    {
        // Activate corresponding render state
        shader.use();
        shader.setVec3("color", color);
        glActiveTexture(GL_TEXTURE0);
        glBindVertexArray(VAO);

        // Iterate through all characters
        std::string::const_iterator c;
        for (c = text.begin(); c != text.end(); c++)
```

```
        {
            Character ch = Characters[*c];

            GLfloat xpos = x + ch.Bearing.x * scale;
            GLfloat ypos = y - (ch.Size.y - ch.Bearing.y) * scale;

            GLfloat w = ch.Size.x * scale;
            GLfloat h = ch.Size.y * scale;
            // Update VBO for each character
            GLfloat vertices[6][4] = {
                { xpos,     ypos + h,   0.0, 0.0 },
                { xpos,     ypos,       0.0, 1.0 },
                { xpos + w, ypos,       1.0, 1.0 },

                { xpos,     ypos + h,   0.0, 0.0 },
                { xpos + w, ypos,       1.0, 1.0 },
                { xpos + w, ypos + h,   1.0, 0.0 }
            };
            // Render glyph texture over quad
            glBindTexture(GL_TEXTURE_2D, ch.TextureID);
            // Update content of VBO memory
            glBindBuffer(GL_ARRAY_BUFFER, VBO);
            glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertices), vertices); // Be sure to
use glBufferSubData and not glBufferData

            glBindBuffer(GL_ARRAY_BUFFER, 0);
            // Render quad
            glDrawArrays(GL_TRIANGLES, 0, 6);
            // Now advance cursors for next glyph (note that advance is number of 1/64
pixels)
            x += (ch.Advance >> 6) * scale; // Bitshift by 6 to get value in pixels (2^6 = 64
(divide amount of 1/64th pixels by 64 to get amount of pixels))
        }
        glBindVertexArray(0);
        glBindTexture(GL_TEXTURE_2D, 0);
    }
```