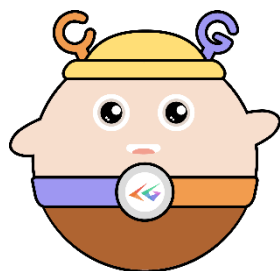


华中科技大学



计算机图形学课程

实验：纹理贴图 and 天空盒

目录

| | |
|----------------------|----|
| 1 实验概述 | 3 |
| 2 顶点数据 | 4 |
| 2.1 立方体顶点数据 | 4 |
| 2.2 天空盒顶点数组 | 4 |
| 3 纹理载入 | 5 |
| 3.1 创建纹理 | 5 |
| 3.2 纹理读取 | 6 |
| 3.3 纹理绑定 | 7 |
| 4 使用纹理 | 8 |
| 4.1 立方体着色器 | 8 |
| 4.1.1 顶点着色器 | 8 |
| 4.1.2 片元着色器 | 8 |
| 4.2 天空盒着色器 | 9 |
| 4.2.1 顶点着色器 | 9 |
| 4.2.2 片元着色器 | 9 |
| 5 立方体贴图和平面纹理对比 | 11 |
| 5.1 着色器中的区别 | 11 |
| 5.2 定义和设置上的区别 | 11 |
| 5.2.1 纹理定义 | 11 |
| 5.2.2 资源载入 | 12 |
| 5.3 天空盒位置的控制 | 12 |
| 6 总结 | 14 |

1 实验概述

这次实验我们主要学习如何绘制带有平面纹理的立方体，以及运用立方体贴图实现的天空盒。实验要求：

- (1) 平面纹理（之前实验的基础上，在立方体贴上纹理）
- (2) 立方体贴图（天空盒，背景的天空盒采用立方体贴图）

实验最终的实现效果：



图 1 实验最终实现效果

2 顶点数据

2.1 立方体顶点数据

这是立方体顶点数组，可以看到前三个 **float** 量是我们熟悉的顶点坐标位置，后面两个 **float** 量是顶点所对应的纹理 **UV** 坐标，通过这个 **UV** 坐标，我们可以控制将纹理上的哪一部分贴在三角形片元上。

```
const float vertices[] =
{
    -0.5f, -0.5f, -0.5f,  0.0f, 0.0f,
    0.5f, -0.5f, -0.5f,  1.0f, 0.0f,
    0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
    0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
    -0.5f,  0.5f, -0.5f,  0.0f, 1.0f,
    -0.5f, -0.5f, -0.5f,  0.0f, 0.0f,
    ...
};
```

2.2 天空盒顶点数组

紧接着我们看到天空盒的顶点数组。我们可以发现，天空盒的顶点数组是没有之前的纹理坐标的。

```
float skybox_vertices[] = {
    // positions
    -1.0f,  1.0f, -1.0f,
    -1.0f, -1.0f, -1.0f,
    1.0f, -1.0f, -1.0f,
    1.0f, -1.0f, -1.0f,
    1.0f,  1.0f, -1.0f,
    -1.0f,  1.0f, -1.0f,
    .....
}
```

因为我们使用的是立方体贴图，天空盒的顶点坐标即可对应它纹理坐标。我们可以直接将 顶点坐标作为立方体贴图的纹理坐标

3 纹理载入

我们用平面纹理包裹住之前实验实现的立方体，用立方体贴图包裹住天空盒，即可得到我们最后需要的结果。

为了实现纹理贴图，我们需要进行几步操作：首先进行纹理定义和设置，然后进行纹理资源载入，生成多级纹理，在使用前进行纹理绑定，以及最后在着色器中进行采样。

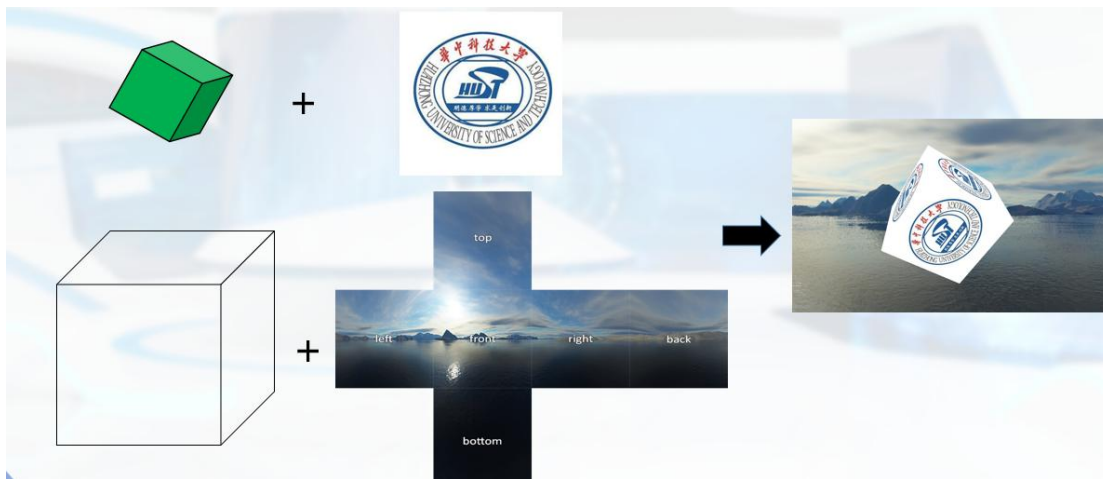


图 2 为立方体和天空盒加上纹理

3.1 创建纹理

首先我们定义一个纹理，绑定在 `GL_TEXTURE_2D` 上，然后就开始设置它的纹理属性，最后四行中，前两行是用来设置纹理的环绕方式。后两行则是设置纹理的过滤方式。设置这些纹理的属性有很多参数，下面我们就来介绍一下这里的参数有哪些，分别都是什么作用。

```
GLuint texture1;
glGenTextures(1, &texture1);
glBindTexture(GL_TEXTURE_2D, texture1);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

上面的代码中，`glTexParameteri` 函数是负责设置纹理的属性。

其中第四、第五行代码：

(1) `GL_TEXTURE_WRAP_S` 表示在 `s` 轴上纹理的环绕方式;

(2) `GL_TEXTURE_WRAP_T` 表示在 `T` 轴上纹理的环绕方式, 这里 `s` 和 `t` 等价于平面纹理图片的 `x` 轴和 `y` 轴;

(3) `GL_REPEAT` 是表示纹理重复出现, 它也是在不设置的情况下默认环绕方式。 `GL_MIRRORED_REPEAT` 也是重复图片, 但是他表示以镜像方式重复出现;

(4) `GL_CLAMP_TO_EDGE` 表示纹理坐标会被约束在 `0-1` 之间, 超出的部分会重复纹理坐标的边缘, 产生一种边缘被拉伸的效果。这种环绕方式通常会在我们设置纹理坐标超过 `0-1` 的范围时被使用到。

其中第六第七行代码:

(1) `GL_TEXTURE_MIN_FILTER` 是设置纹理在缩小时的过滤方式

(2) `GL_TEXTURE_MAG_FILTER` 是设置纹理在放大时的过滤方式。

过滤方式我们主要使用的有两种, 一种是 `GL_NEAREST` 即线性过滤, 这种过滤方法会产生颗粒状的图案, 但是也能更清晰的看到组成纹理的像素。 `GL_LINEAR` 即临近过滤, 它能够产生更平滑的图案, 但是也有更真实的输出。

3.2 纹理读取

接下来我们学习如何进行资源载入, 即把图片读入内存, 最终绑定到着色器。我们使用 `#include "stb_image.h"` 中的 `stbi_load`, 根据路径读取纹理图片, 并读取纹理的宽高和通道数, 将纹理数据存入 `data` 数组中, 最后判断是否读取成功, 若失败, 则报错, 若成功, 则进行下一步操作。部分代码如下:

```
//载入纹理资源
int width, height, nrchannels;//纹理长宽, 通道数
stbi_set_flip_vertically_on_load(true);
unsigned char *data = stbi_load("res/texture/contain.jpg", &width,
&height, &nrchannels, 0);
if (data)
{
    //生成纹理
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
GL_UNSIGNED_BYTE, data);
    glGenerateMipmap(GL_TEXTURE_2D);
}
else
{

```

```
std::cout << "Failed to load texture" << std::endl;
}
```

在读取纹理之后，我们还要将 **data** 中的纹理数据存入 **GL_TEXTURE_2D**，存入的时候我们需要同时输入纹理的宽高，接着我们使用 **glGenerateMipmap(GL_TEXTURE_2D)**生成多级渐远纹理。

3.3 纹理绑定

由于可能出现多个纹理，所以我们每次使用纹理绘制之前，最好进行一次绑定操作。**glActiveTexture(GL_TEXTURE0)**为选择 **GL_TEXTURE0** 纹理单元，将要使用的纹理绑定到 **GL_TEXTURE0** 纹理单元中，计算机有很多纹理单元 **GL_TEXTURE1 GL_TEXTURE0 GL_TEXTURE2 GL_TEXTURE3 GL_TEXTURE4**，使用时每个纹理最好对一个纹理单元。

接着我们使用 **glBindTexture(GL_TEXTURE_2D, texture1);**去绑定纹理，绑定后即可进行绘制。

```
glActiveTexture(GL_TEXTURE0); //绑定纹理
glBindTexture(GL_TEXTURE_2D, texture1);
.....
glDrawArrays(GL_TRIANGLES, 0, 36); //绘制
```

4 使用纹理

后面介绍如何在着色器中使用纹理，需要注意的是，立方体和天空盒使用纹理的方式是不一样的，我们先看看着色器中如何使用平面纹理。

4.1 立方体着色器

4.1.1 顶点着色器

立方体的顶点着色器负责的纹理方面的功能不多，只是将顶点对应的纹理坐标传入片元着色器。

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoord;
out vec2 TexCoord;
...
void main(){
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    TexCoord=vec2(aTexCoord.x,aTexCoord.y);
}
```

4.1.2 片元着色器

片元着色器则是根据传入的 **sampler2D** 类型的纹理和纹理对应坐标，使用 **texture** 进行采样，获取当前片元的颜色值。

```
#version 330 core
out vec4 FragColor;
in vec2 TexCoord;
uniform sampler2D texture1;
void main() {
    FragColor = texture(texture1,TexCoord);
}
```

4.2 天空盒着色器

4.2.1 顶点着色器

接着我们看到天空盒的着色器具体跟立方体的着色器到底有哪些不同。

天空盒的顶点着色器将顶点坐标直接作为纹理坐标传入片元着色器，可以发现平面纹理的纹理坐标是 **vec2** 类型，而立方体的纹理坐标则是 **vec3** 类型，并且 **pvm** 矩阵也发生了变化，这个我们后面再提，还有我们可以看到的是我们把 **z** 变量替换成了 **w**，这是为了使天空盒永远在所有物体之后，作为背景，使其深度值最大，不遮挡任何物体。

```
#version 330 core
layout (location = 0) in vec3 aPos;

out vec3 TexCoords;

uniform mat4 projection;
uniform mat4 view;

void main()
{
    TexCoords = aPos;
    vec4 pos = projection * view * vec4(aPos, 1.0);
    gl_Position = pos.xyww;
}
```

4.2.2 片元着色器

天空盒的片元着色器也与立方体的片元着色器有些不同，它传入的是 **samplerCube** 类型的立方体贴图，且纹理坐标也是 **vec3** 类型，但是最终还是通过 **texture** 进行采样。

```
#version 330 core
out vec4 FragColor;

in vec3 TexCoords;

uniform samplerCube skybox;

void main()
```

```
{  
    FragColor = texture(skybox, TexCoords);  
}
```

5 立方体贴图和平面纹理对比

5.1 着色器中的区别

我们刚才这是对立方体和天空盒着色器的区别进行了分析，发现天空盒着色器传入的纹理是 `samplerCube` 类型，那我们怎样向着色器传入 `samplerCube` 类型的纹理呢？它与立方体纹理的区别又在那里呢？

立方体贴图的六个面，每个面都对应一个纹理，且有其相应的顺序。有人可能会觉得为什么不能贴六个纹理而非要用立方体贴图呢？那是因为立方体贴图有其独特的属性，我们有时候可以直接使用方向向量对立方体贴图进行索引和采样。

我们可以设想一下，有一束光线从立方体的中心向任意一个方向射出，我们知道了射出光线的方向向量，即可在立方体贴图上找到对应的纹理坐标，获取到相应的颜色值，通过这种方法我们可以实现光的折射和反射，通过出射光和入射光的方向找到对应的颜色值。

5.2 定义和设置上的区别

我们之前讲了立方体贴图和平面纹理在着色器中的区别，那他们在别的方面还有什么区别吗？因为立方体贴图包括六个相关的平面纹理，所以他们在纹理的定义和设置和资源载入上还是有一些区别的

5.2.1 纹理定义

首先设置纹理属性值，首先是定义纹理 `id` 时，我们将 `GLuint` 换成了 `unsigned int`，这两种方式其实是完全等价的，所以使用的时候我们可以根据自己的需要来，接着就是绑定纹理，因为是立方体纹理，所以之前的绑定在 `GL_TEXTURE_2D` 变成了绑定在 `GL_TEXTURE_CUBE_MAP`，其他属性的设置基本相同，但是需要注意的是，由于以及是立体纹理而不仅仅是平面纹理了，所以他的环绕方式也多出了一个 `R` 轴的环绕方式，对应的是现实中的 `z` 轴。

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

```
        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_T
O_EDGE);
        glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_T
O_EDGE);
        glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_T
O_EDGE);
        return textureID;
    }
}
```

5.2.2 资源载入

资源载入时我们也需要一次性载入 6 张纹理，一般我们使用 **for** 循环载入，循环六次，同时每次载入纹理存储的地址也有所不同，不同的纹理存储在不同的地址，不过这些地址都是相邻的，所以每次只需要加 **i** 即可；

```
unsigned int load_cubemap(std::vector<std::string> faces)
{
    .....
    int width, height, nrchannels;
    for (unsigned int i = 0; i < 6; i++)
    {
        glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0,
GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
        .....
    }
    .....
    return textureID;
}
}
```

5.3 天空盒位置的控制

那么我们该如何摆放天空盒呢？首先天空盒是一个包裹着相机的大盒子，它不需要进行世界坐标系的模型变换，其次相机不管如何移动，应该始终在天空盒的中心点，那么我们如何取消天空盒着色器中 **view** 矩阵的位移而保存相机视角变换之类的操作呢

相机 **view** 矩阵，主要包含了位移和旋转操作，如图 4.3.1 所示，第一行公式为位移矩阵如何发挥其功能，而位移部分则是在 T_x , T_y , T_z 中体现， T_x , T_y , T_z 为位移量，我们只需要把 **view** 矩阵从 **4x4** 先转换为 **3x3** 矩阵，去掉位移部分，再转换回 **4x4** 矩阵，则可完全去掉位移量，并保持旋转效果。


$$\begin{bmatrix} a & 0 & 0 & T_x \\ 0 & b & 0 & T_y \\ 0 & 0 & c & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} ax + T_x \\ by + T_y \\ cz + T_z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \xrightarrow{\hspace{1cm}} \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} ax \\ by \\ cz \\ 1 \end{bmatrix}$$

图 3 去除 view 矩阵的位移部分

体现在程序中则是如下操作，先转换成 **mat3** 矩阵，再转换成 **mat4** 矩阵。

```
view = glm::mat4(glm::mat3(glm::lookAt(camera_position, camera_position  
+ camera_front, camera_up))); // 去除相机位移
```

6 总结

以上为本次实验的要点解析，下面是实验结果演示。运行之后我们移动视角可以发现，不管我们怎样移动相机的位置，天空盒相对于摄像机的位置始终保持不变。



图 4 实验最终实现效果