立方体纹理采样选择

首先确定选择面，然后另外两个坐标当做uv值（需要将-1 - 1范围变换到0 - 1）

选择反射向量中三个坐标的最大值，可以确定是哪个面（通过符号判断是正方向还是负方向），将另外两个坐标转换到uv值进行采样

下面我们从数学的角度来导出映射过程：

       第一步，给出一个3D向量[x,y,z]，首先找出值最大的那一维。以[-3.2, 5.1, -8.4]为例子， 这时最大维为Z，最大维用来把cube map从六张图定位到一张图上。-8.4为负数，对应了立方体的负Z面，因此接下来我们关注的焦点将是立方体负Z面对应的那张纹理；

       第二步，把向量中另外两维分别除以最大维，得到一个二维向量，(3.2/8.4, -5.1/8.4)。很容易知道，这个二维向量中的数值范围位于[-1, 1]之间。

       第三步，把上阶段中得到的二维向量转换到[0, 1]之间。很简单，把位于[-1, 1]之间的数转换到[0, 1]之间的方法为: (x + 1) / 2。对于上面的例子， 为3.2/8.4 \* 2 + 0.5 = 0.31, -5.1 / 8.4 \* 2 + 0.5 = 0.51。因此得到二维向量(0.31, 0.51)。这个二维向量就是我们用来在负Z轴上获得texel的纹理坐标。

void CubeMap::setRC()

{

cout << "cubeMap setup" << endl;

modelCube.initProgram("shaderCube.vert", "shaderCube.frag");

modelCube.bindAttrib(1, "a\_position");

modelCube.bindAttrib(2, "a\_color");

modelCube.bindAttrib(3, "a\_normal");

modelCube.bindAttrib(4, "a\_textureCoord");

string path[6];

path[0] = "..\\right.jpg";

path[1] = "..\\left.jpg";

path[2] = "..\\top.jpg";

path[3] = "..\\bottom.jpg";

path[4] = "..\\back.jpg";

path[5] = "..\\front.jpg";

modelCube.link();

modelCube.initCubeMapPath(path);

modelCube.initData(verticesCube11, indicesCube11, sizeof(verticesCube11) / sizeof(GL\_FLOAT), sizeof(indicesCube11) / sizeof(GL\_UNSIGNED\_INT));

modelCube.cubeBegin();

modelCube.showData();

modelCube2.initProgram("shaderCube.vert", "shaderCube.frag");

modelCube2.bindAttrib(1, "a\_position");

modelCube2.bindAttrib(2, "a\_color");

modelCube2.bindAttrib(3, "a\_normal");

modelCube2.bindAttrib(4, "a\_textureCoord");

cout << endl;

string path1[6];

path1[0] = "..\\test.jpg";

path1[1] = "..\\test.jpg";

path1[2] = "..\\test.jpg";

path1[3] = "..\\test.jpg";

path1[4] = "..\\test.jpg";

path1[5] = "..\\test.jpg";

modelCube2.link();

modelCube2.initCubeMapPath(path1);

modelCube2.setTranslate(0.0f, 0.0f, -140.0f);

modelCube2.setScale(0.5f, 0.5f, 0.5f);

modelCube2.setRotation(0.3f, 0.0f, -1.0f);

modelCube2.initData(verticesCube11, indicesCube11, sizeof(verticesCube11) / sizeof(GL\_FLOAT), sizeof(indicesCube11) / sizeof(GL\_UNSIGNED\_INT));

modelCube2.cubeBegin();

modelCube2.showData();

}

void CubeMap::render()

{

glDepthMask(GL\_FALSE);

modelCube.use();

modelCube.update();

modelCube.setAngle(0.0f);

modelCube.cubeDraw(projection, view);

glDepthMask(GL\_TRUE);

modelCube2.use();

modelCube2.update();

//modelCube2.setAngle(10.0f);

modelCube2.cubeDraw(projection, view);

}

void ModelTmp::cubeBegin()

{

glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_FILL);

glEnable(GL\_DEPTH\_TEST);

glGenBuffersARB(1, &vboId);

glBindBufferARB(GL\_ARRAY\_BUFFER, vboId);

glBufferDataARB(GL\_ARRAY\_BUFFER, sizeof(GL\_FLOAT)\* verticesData.size(), verticesData.data(), GL\_DYNAMIC\_DRAW);

glGenBuffersARB(1, &eboId);

glBindBufferARB(GL\_ELEMENT\_ARRAY\_BUFFER, eboId);

glBufferDataARB(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof(GL\_FLOAT)\* indicesData.size(), indicesData.data(), GL\_STATIC\_DRAW);

// 纹理

glGenTextures(1, &textureId);

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, textureId);

for (int i = 0; i < 6; i++)

{

GLbyte \*bytes = tgaManager.load(cubeTexturePath[i].c\_str());

if (bytes == NULL) cout << "load tga picture failed" << endl;

cout << "name = " << cubeTexturePath[i].c\_str() << endl;

cout << "component = " << tgaManager.getComponent() << endl;

cout << "width = " << tgaManager.getWidth() << endl;

cout << "height = " << tgaManager.getHeight() << endl;

glTexImage2D(GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X + i, 0, GL\_RGB, tgaManager.getWidth(), tgaManager.getHeight(),

0, GL\_RGB, GL\_UNSIGNED\_BYTE, bytes);

free(bytes);

// 设置纹理参数

glTexParameterf(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameterf(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

// 设置纹理环绕

glTexParameterf(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameterf(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexParameterf(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_R, GL\_CLAMP\_TO\_EDGE);

//// 设置纹理环境

//glTexEnvf(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_MODULATE);

//// 开启纹理对象

//glEnable(GL\_TEXTURE\_2D);

}

}

void ModelTmp::cubeDraw(glm::mat4 projection, glm::mat4 view)

{

// 顶点坐标与颜色

GLsizei stride = 13 \* sizeof(GL\_FLOAT);

GLuint \*vertexPointer = nullptr;

GLvoid \*colorPointer = ((GLubyte\*)NULL) + 3 \* sizeof(GL\_FLOAT);

GLvoid \*normalPointer = ((GLubyte\*)NULL) + 7 \* sizeof(GL\_FLOAT);

GLvoid \*texPointer = ((GLubyte\*)NULL) + 10 \* sizeof(GL\_FLOAT);

glBindBufferARB(GL\_ARRAY\_BUFFER, vboId);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, eboId);

glActiveTexture(GL\_TEXTURE0); // 激活0号纹理单元

glUniform1i(glProgram->getUniformLocation("our\_texture"), 0); // 告诉shader在0号纹理单元中的纹理进行采样

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, textureId); // 将textureId1绑定到0号纹理单元

glEnableVertexAttribArray(1);

glEnableVertexAttribArray(2);

glEnableVertexAttribArray(3);

glEnableVertexAttribArray(4);

glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, stride, NULL);

glVertexAttribPointer(2, 4, GL\_FLOAT, GL\_FALSE, stride, colorPointer);

glVertexAttribPointer(3, 3, GL\_FLOAT, GL\_TRUE, stride, normalPointer);

glVertexAttribPointer(4, 3, GL\_FLOAT, GL\_FALSE, stride, texPointer);

glPushMatrix();

glm::mat4 transform;

transform = glm::translate(transform, glm::vec3(translateX, translateY, translateZ));

transform = glm::rotate(transform, angle, glm::vec3(rotateX, rotateY, rotateZ));

transform = glm::scale(transform, glm::vec3(scaleX, scaleY, scaleZ));

setUniformMatrix4fv("projection", glm::value\_ptr(projection));

setUniformMatrix4fv("view", glm::value\_ptr(view));

setUniformMatrix4fv("transform", glm::value\_ptr(transform));

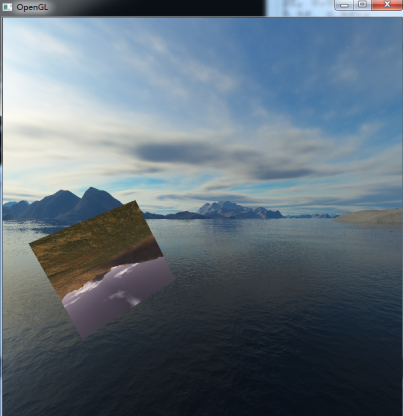
glDrawElements(GL\_TRIANGLES, sizeof(GL\_FLOAT)\* indicesData.size(), GL\_UNSIGNED\_INT, NULL);

glPopMatrix();

glBindBufferARB(GL\_ARRAY\_BUFFER, 0);

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, 0);

}



**优化**

现在我们在渲染场景中的其他物体之前渲染了天空盒。这么做没错，但是不怎么高效。如果我们先渲染了天空盒，那么我们就是在为了每个屏幕上的像素运行像素着色器，即使天空盒只有部分在显示着，fragment可以使用提前深度测试简单地被丢弃，这样就节省了我们宝贵的带宽。

所以最后渲染天空和就能够给我们带来轻微的性能提升。采用这种方式，深度缓冲被全部物体的深度值完全填充，所以我们只需要渲染通过前置深度测试的那部分天空的fragment就行了，而且能显著减少像素着色器的调用。问题是天空盒是一个1\*1\*1的立方体，极有可能会渲染失败，因为极有可能通不过深度测试，简单地不用深度测试渲染它不是解决方案，这是因为天空盒会在之后覆盖所有场景中其他物体，我们需要耍个花招让深度缓冲相信天空哥的深度缓冲有着最大深度值1.0，如此只要有个物体存在深度测试就会失败，看上去物体就在它的前面了。透视除法是在顶点着色器运行之后运行的，把gl\_Position的syz坐标除以w元素。我们从深度测试中了解到除法结果的z元素等于顶点的深度值，利用这个信息，我们可以把输出位置的z元素设置为它的w元素，这样就会导致z元素等于1.0了。因为，当透视除法应用后，它的z元素转换为w/w=1.0 （已经测试）

void CubeMap::render()

{

// 正常物体

**glDepthFunc(GL\_LESS);**

modelCube2.use();

modelCube2.update();

//modelCube2.setAngle(10.0f);

modelCube2.cubeDraw(projection, view);

// 立方体贴图

**glDepthFunc(GL\_LEQUAL);**

modelCube.use();

modelCube.update();

modelCube.setAngle(0.0f);

modelCube.cubeDraw(projection, view);

**glDepthFunc(GL\_LESS);**

}

顶点着色器：

in vec3 a\_position;

in vec4 a\_color;

in vec3 a\_normal;

in vec3 a\_textureCoord;

out vec3 out\_position;

out vec4 out\_color;

out vec3 out\_normal;

out vec3 out\_textureCoord;

uniform mat4 transform;

uniform mat4 view;

uniform mat4 projection;

void main(void)

{

vec4 pos = projection \* view \* transform \* vec4(a\_position, 1.0f);

**gl\_Position.xyw = pos.xyww;**

out\_position = view \* transform \* vec4(a\_position, 1.0f);

out\_color = a\_color;

out\_normal = mat3(transpose(inverse(view \* transform))) \* a\_normal;

out\_textureCoord = a\_textureCoord;

}

**环境贴图**

void CubeMap::render()

{

// 立方体贴图

glDepthMask(GL\_FALSE);

modelCube.use();

modelCube.update();

modelCube.setAngle(0.0f);

modelCube.cubeDraw(projection, view);

glDepthMask(GL\_TRUE);

// 正常物体

modelCube2.use();

modelCube2.setTextureId(modelCube.getTextureId());

modelCube2.cubeDraw(projection, view);

}

**顶点着色器**：

in vec3 a\_position;

in vec4 a\_color;

in vec3 a\_normal;

in vec3 a\_textureCoord;

out vec3 out\_position;

out vec4 out\_color;

out vec3 out\_normal;

out vec3 out\_textureCoord;

uniform mat4 transform;

uniform mat4 view;

uniform mat4 projection;

void main(void)

{

vec4 pos = projection \* view \* transform \* vec4(a\_position, 1.0f);

gl\_Position.xyw = pos.xyzw;

out\_position = view \* transform \* vec4(a\_position, 1.0f);

out\_color = a\_color;

out\_normal = mat3(transpose(inverse(view \* transform))) \* a\_normal;

out\_textureCoord = a\_textureCoord;

}

**片元着色器：**

in vec3 out\_position;

in vec4 out\_color;

in vec3 out\_normal;

in vec3 out\_textureCoord;

uniform samplerCube our\_texture;

void main(void)

{

//float depth = linearizeDepth(gl\_FragCoord.z);

vec3 l = normalize(out\_position);

vec3 r = reflect(l, normalize(out\_normal));

gl\_FragColor = textureCube(our\_texture, r) \* vec4(1.0f, 0.0f, 0.0f, 1.0f);

}

在片元着色器上的反射纹理上叠加了红色光：

