

2018-2019年度第二学期 00106501

计算机图形学



童伟华 管理科研楼1205室

E-mail: tongwh@ustc.edu.cn

中国科学技术大学 数学科学学院

<http://math.ustc.edu.cn/>





第四节 GLSL(III)

连接着色器与OpenGL



- 调用glCreateShader创建着色器对象
- 调用glShaderSource为着色器加载源代码
- 调用glCompileShader编译每个着色器
- 调用glCreateProgram创建程序对象
- 调用glAttachShader把着色器对象连接到程序对象
- 调用glLinkProgram 链接程序对象，生成可执行程序
- 调用glUseProgram安装可执行程序替换OpenGL固定功能流水线处理模块
- 用uniform和attribute (in, out) 变量 (或块) 在应用程序和着色器间通信

应用程序->着色器通信



■ OpenGL应用程序有两种方式向着色器发送值

- 使用内置和用户定义的全局变量
 - uniform变量或块
 - attribute (in, out) 变量或块
- 使用纹理
 - 纹理可解释为图像或数据数组

定义顶点属性



- 应用程序通过顶点属性的索引传送数据
- 通过下列方式之一指定自定义顶点属性索引：
 - 在着色器中通过layout描述，例如
`layout(location = 2) in vec4 a_vec;`
 - 在着色器程序链接之前，通过glBindAttribLocation描述，例如
`void glBindAttribLocation(GLuint program, GLuint index, const GLchar *name);`
 - 如果未采取上述两种方法，OpenGL会为着色器中出现的attribute自动分配顶点属性索引
- 当程序对象成功链接后，链接器生成一张活动属性变量名表
 - 用glGetAttribLocation查询活动属性变量的内存位置 或 着色器程序内省机制（新的方式，可以通过一致的方式查询所有信息）
`glGetProgramResourceIndex/glGetProgramResourceiv`等
 - 通过glVertexAttribPointer和 glEnableVertexAttribArray来使用顶点数组

活动属性变量

- **活动 (active) 属性变量：** 由编译器和链接器决定的顶点着色器执行时会访问的属性
 - 仅只声明从不使用的是非活动属性
 - 程序对象链接失败，如果活动的内置和自定义属性数目多于 `GL_MAX_VERTEX_ATTRIBS (16)`

layout修饰符



- 可以通过layout修饰符在着色器中指定属性变量的位置（索引， $\text{location} = \text{attribute index}$ ），例如

`layout(location = 2) in vec3 values[4];`

- 一般的语法：

`layout(qualifier1, qualifier2 = value, ...) variable definition`

- 在应用程序中，可直接索引设置属性变量的值

显式绑定索引



- `void glBindAttribLocation(GLuint program, GLuint index, const GLchar *name);`
 - 指定下一次链接时绑定到程序对象program中属性变量name的索引为index
 - 如果name是内置属性变量（以gl_开头），产生一个GL_INVALID_OPERATION错误
 - 如果name之前绑定过，则索引被index替换
 - 如果name是属性矩阵，index绑定到第一列
 - index取值[0, GL_MAX_VERTEX_ATTRIBS-1]
 - 程序成功链接后，绑定才生效

获取属性变量的索引



■ `GLint glGetAttribLocation(GLuint program,
 const GLchar *name);`

- 返回上一次链接时绑定到程序对象program中活动属性变量name的索引
- 如果name不是program的一个活动属性变量，或是内置属性变量（以gl_开头），返回-1
- 如果name是活动属性矩阵，返回矩阵第一列的索引；后一系列的索引为前一列的索引加1

设置顶点属性的值

■ 单精度浮点数据: void

```
glVertexAttribPointer(GLuint index, GLint  
size, GLenum type, GLboolean normalized,  
GLsizei stride, const GLvoid *pointer);
```

- 为索引值为index的属性变量的数组指定位置pointer和数据格式
- 启用/禁用属性index的顶点属性数组

```
void glEnableVertexAttribArray(GLuint index)
```

```
void glDisableVertexAttribArray(GLuint index)
```

- 启用后, 就可用glDrawArrays()等来为顶点属性变量加载值

■ 整数型数据: glVertexAttribIPointer类似

■ 双精度浮点数据: glVertexAttribLPointer类似

例子



■ 顶点着色器中:

```
layout(location = 0) in vec4 aPosition;
```

■ OpenGL 中:

```
int stride = sizeof(Vertex3D);  
GLint loc =  
glGetAttribLocation(shaderProgramObject,  
"aPosition");  
glVertexAttribPointer(loc, 3, GL_FLOAT,  
GL_FALSE, stride, BUFFER_OFFSET(0));  
glEnableVertexAttribArray(loc);
```

uniform 变量



- uniform 变量的值在着色器程序执行期内保持不变
 - 用户定义
 - 内置：变换矩阵、裁剪平面等
- 活动 (active) uniform 变量：由编译器和链接器决定的着色器执行时会访问的一致变量
 - 程序对象链接失败，如果活动的内置和自定义一致变量数目多于 `GL_MAX_VERTEX_UNIFORM_COMPONENTS` (512)

指定uniform变量



- 当程序对象成功链接后，链接器生成一张活动一致变量名表
 - 用glGetUniformLocation查询活动一致变量的内存位置
 - 调用glUniform*给一致变量加载值
- uniform变量的位置（索引）可以通过layout指定或系统自动分配，例如

```
layout(location = 2) uniform mat4
uModelViewMatrix;
```
- 注意： glUniform*设置uniform值的时候，都是针对当前的着色器程序

获取一致变量的索引



- `GLint glGetUniformLocation(GLuint program, const GLchar *name);`
 - 返回上一次链接时绑定到程序对象program中活动一致变量name的索引
 - 如果name不是program的一个活动一致变量，或是保留一致变量(以gl_开头)，返回-1
 - name不能是结构、结构数组、向量/矩阵的部分
 - 对结构和数组，在name里用"."和"[]"来指定结构字段和数组元素
 - 数组第一个元素的位置可在name中用数组名或数组名加"[0]"获取

设置一致变量的值



- `void glUniform{1234}{if}(GLint location, TYPE values);`
- `void glUniform{1234}{if}v(GLint location, GLsizei count, const TYPE *values);`
- `void glUniformMatrix{234}fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat *values);`
- `void glUniformMatrix{2x3,3x2,2x4,4x2,3x4,4x3}fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat *values);`
 - 设置当前使用程序对象在位置location的一致变量的值
 - ...

设置一致变量的值

- 向量形式把count组值加载到uniform数组的location位置开始的count个元素。这里，location不是数组元素下标
- 矩阵形式中的transpose为GL_TRUE表示values按行主序指定；否则，按列主序指定
- 上述函数可用于加载布尔型变量，自动转换类型
- 当函数名指定的大小和类型（除去布尔型）和着色器中声明的uniform变量不匹配时，报错
- 当count>1，而着色器中声明的uniform变量不是数组时，报错

例子



■ 着色器中:

```
uniform float specIntensity;  
uniform vec4 specColor;  
uniform float t[2];  
uniform vec4 colors[3];
```

例子 (续)



■ OpenGL 中:

```
GLint loc1,loc2,loc3,loc4;
float specIntensity = 0.98;
float sc[4] = {0.8,0.8,0.8,1.0};
float threshold[2] = {0.5,0.25};
float colors[12] = {0.4,0.4,0.8,1.0,
                   0.2,0.2,0.4,1.0,
                   0.1,0.1,0.1,1.0};
loc1 = glGetUniformLocation(p,"specIntensity");
glUniform1f(loc1, specIntensity);
loc2 = glGetUniformLocation(p,"specColor");
glUniform4fv(loc2, 1, sc);
loc3 = glGetUniformLocation(p,"t");
glUniform1fv(loc3, 2, threshold);
loc4 = glGetUniformLocation(p,"colors");
glUniform4fv(loc4, 3, colors);
```

uniform向量和数组

■ 着色器中:

```
uniform vec4 specColor;  
uniform float t[2];
```

■ OpenGL中:

```
GLint loc1,loc2,loc3;  
float sc[4] = {0.8,0.8,0.8,1.0};  
float threshold[2] = {0.5,0.25};  
loc2 = glGetUniformLocation(p,"specColor");  
glUniform4f(loc2, sc[0], sc[1], sc[2], sc[3]);  
// glUniform4fv(loc2, 1, sc);    /* 等价形式 */  
  
loct0 = glGetUniformLocation(p, "t[0]");  
glUniform1f(loct0, threshold[0]);  
loct1 = glGetUniformLocation(p, "t[1]");  
glUniform1f(loct1, threshold[1]);
```

着色器示例



■ 顶点着色器

- 波动
- 渐变
- 粒子系统
- 非真实感着色
- Phong光照

■ 片段着色器

- Phong光照
- 环境映射
- 凹凸映射

注意：下面的着色器示例程序是基于GLSL 1.2版，
仅适用于Compatibility profile

波动顶点着色器



```
uniform float time;  
uniform float xs, zs, // frequencies  
uniform float h; // height scale  
void main()  
{  
    vec4 t = gl_Vertex;  
    t.y = gl_Vertex.y + h*sin(time + xs*gl_Vertex.x)  
        + h*sin(time + zs*gl_Vertex.z);  
    gl_Position = gl_ModelViewProjectionMatrix*t;  
}
```

attribute变量和uniform变量在顶点着色器内只读

Idle回调函数

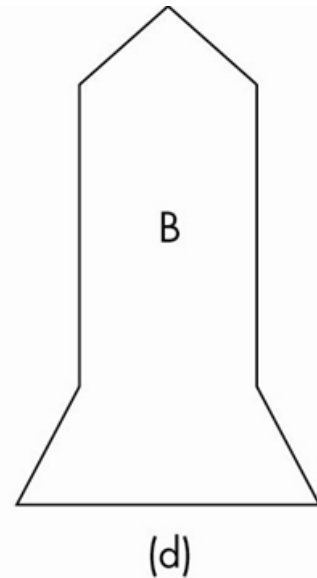
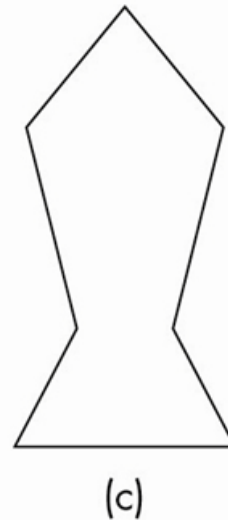
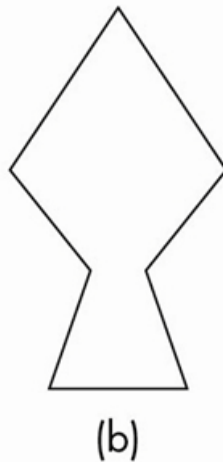
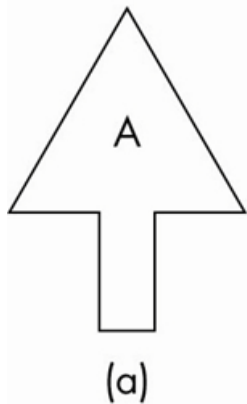


```
Glint timeParam;  
timeParam = glGetUniformLocation(program, "time");  
  
void idle()  
{  
    glUniform1f(timeParam,  
        (GLfloat) glutGet(GLUT_ELAPSED_TIME));  
    glutPostRedisplay();  
}
```

渐变 (morphing) 效果



- 渐变：一个物体平滑地变换到另一个物体
- 假设两个物体的顶点有一一对应关系
- 顶点着色器需要输出一个由对应顶点对插值得到的顶点



顶点着色程序



- 一个顶点通过gl_Vertex传入，对应顶点用顶点属性变量传入

```
attribute vec4 vertices2;  
uniform float time;
```

```
void main()  
{  
    float s = 0.5*(1.0+sin(0.001*time));  
    vec4 t = mix(gl_Vertex, vertices2, s);  
    gl_Position = gl_ModelViewProjectionMatrix*t;  
    gl_FrontColor = gl_Color;  
}
```

$\text{mix}(x, y, a)$ 返回 $x * (1.0-a) + y * a$

OpenGL程序



```
GLint vertices2Param;  
vertices2Param = glGetAttribLocation(program,  
    "vertices2");  
  
#define N 50  
GLfloat vertices_one[N][3], vertices_two[N][3];  
glBegin(GL_TRIANGLES);  
    for (int i = 0; i < N; i++)  
    {  
        glVertexAttrib3fv(vertices2Param,  
            vertices_two[i]);  
        glVertex3fv(vertices_one[i]);  
    }  
glEnd();
```

粒子 (particle) 系统



- 粒子系统的基本思想：用真实或自定义物理规律来控制粒子的运动
 - 每一时间步，要为每个粒子确定一个新位置
- 考虑符合牛顿定律的质点，质量为 m ，初始位置为 (x_0, y_0, z_0) ，初始速度为 (v_x, v_y, v_z) ，重力加速度为 g ，则在 t 时刻的位置为

$$x(t) = x_0 + v_x t,$$

$$y(t) = y_0 + v_y t + g t^2 / (2m),$$

$$z(t) = z_0 + v_z t.$$

顶点着色程序



```
attribute vec3 vel; //初始速度
uniform float g, m, t;
void main()
{
    vec3 object_pos;
    object_pos.x = gl_Vertex.x + vel.x * t;
    object_pos.y = gl_Vertex.y + vel.y * t + g/(2.0*m)*t*t;
    object_pos.z = gl_Vertex.z + vel.z * t;
    gl_Position = gl_ModelViewProjectionMatrix *
        vec4(object_pos,1);
}
```

非真实感着色



- 根据光线和法向的夹角给对象赋两种颜色
- 根据视线和方向的夹角把对象轮廓赋为黑色

```
const vec4 yellow = vec4(1.0, 1.0, 0.0, 1.0);
```

```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
```

```
const vec4 black = vec4(0.0, 0.0, 0.0, 1.0);
```

```
if(dot(L, N) > 0.5) gl_FrontColor = yellow;
```

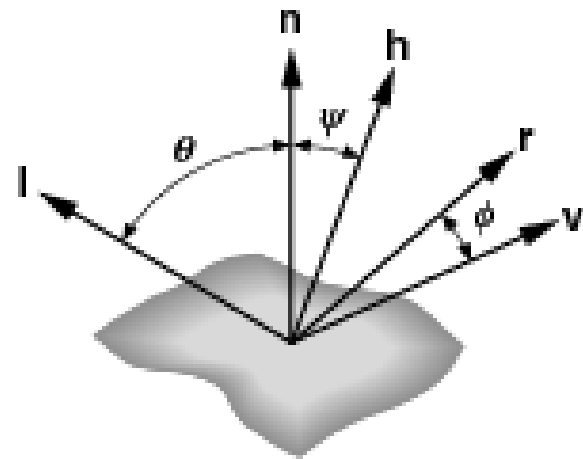
```
else gl_FrontColor = red;
```

```
if(abs(dot(E, N)) < 0.1) gl_FrontColor = black;
```

Phong光照模型



$$I = k_d I_d \mathbf{l} \cdot \mathbf{n} + k_s I_s (\mathbf{r} \cdot \mathbf{v})^\alpha + k_a I_a$$
$$I = k_d I_d \mathbf{l} \cdot \mathbf{n} + k_s I_s (\mathbf{n} \cdot \mathbf{h})^\beta + k_a I_a$$



内置uniform变量



```
struct gl_LightSourceParameters
{
    vec4 ambient; // Acli
    vec4 diffuse; // Dcli
    vec4 specular; // Scli
    vec4 position; // Ppli
    vec4 halfVector; // Derived: Hi
    vec3 spotDirection; // Sdli
    float spotExponent; // Srli
    float spotCutoff; // Crli // (range: [0.0,90.0], 180.0)
    float spotCosCutoff; // Derived: cos(Crli) // (range:
        [1.0,0.0],-1.0)
    float constantAttenuation; // K0
    float linearAttenuation; // K1
    float quadraticAttenuation; // K2
};

uniform gl_LightSourceParameters
    gl_LightSource[gl_MaxLights];
```

内置uniform变量



```
struct gl_LightModelParameters
{
    vec4 ambient; // Acs
};
uniform gl_LightModelParameters gl_LightModel;

struct gl_MaterialParameters
{
    vec4 emission; // Ecm
    vec4 ambient; // Acn
    vec4 diffuse; // Dcm
    vec4 specular; // Scm
    float shininess; // Srm
};
uniform gl_MaterialParameters gl_FrontMaterial;
uniform gl_MaterialParameters gl_BackMaterial;
```

Phong光照



- 光源位置 `gl_LightSource[i].position` 在视点坐标系中给出
- 视点在视点坐标系的原点
- N : 视点坐标系中的法向量
- L : 视点坐标系中的顶点到光源向量 (光线)
- E : 视点坐标系中的顶点到视点向量 (视线)
- R : 视点坐标系中的理想反射向量
- H : 视点坐标系中 L 和 E 的中值向量

改进的Phong顶点着色器



```
void main(void)
/*modified Phong vertex shader (without distance term)*/
{
    gl_Position = gl_ModelViewProjectionMatrix *
gl_Vertex;
    vec4 ambient, diffuse, specular;

    vec4 eyePosition = gl_ModelViewMatrix * gl_Vertex;
    vec4 eyeLightPos = gl_LightSource[0].position;
    vec3 N = normalize(gl_NormalMatrix * gl_Normal);
    vec3 L = normalize(eyeLightPos.xyz - eyePosition.xyz);
    vec3 E = -normalize(eyePosition.xyz);
    vec3 H = normalize(L + E);
```

改进的Phong顶点着色器



```
/*compute diffuse, ambient, and specular contributions*/
float f = 1.0;
float Kd = max(dot(L, N), 0.0);
float Ks = pow(max(dot(N, H), 0.0),
               gl_FrontMaterial.shininess);
if (dot(L,N) < 0.0) f = 0.0;

ambient = gl_FrontLightProduct[0].ambient;
diffuse = Kd*gl_FrontLightProduct[0].diffuse;
specular = f*Ks*gl_FrontLightProduct[0].specular;

gl_FrontColor = ambient + diffuse + specular;
}
```

基于片段的Phong光照



- 利用varying变量把属性从顶点着色器传递到片断着色器
 - 法向量 N
 - 光线向量 L
 - 视线向量 E

基于片段光照的顶点着色器



```
varying vec3 N, L, E;
```

```
void main()  
{  
    gl_Position = gl_ModelViewProjectionMatrix *  
                  gl_Vertex;  
  
    vec4 eyePosition = gl_ModelViewMatrix * gl_Vertex;  
    vec4 eyeLightPos = gl_LightSource[0].position;  
  
    N = normalize(gl_NormalMatrix * gl_Normal);  
    L = normalize(eyeLightPos.xyz - eyePosition.xyz);  
    E = -normalize(eyePosition.xyz);  
}
```

改进Phong光照片段着色器



```
varying vec3 N;  
varying vec3 L;  
varying vec3 E;  
  
void main()  
{  
    vec3 Normal = normalize(N);  
    vec3 Light   = normalize(L);  
    vec3 Eye     = normalize(E);  
    vec3 Half    = normalize(Eye + Light);
```

改进Phong光照片段着色器



```
float f = 1.0;
float Kd = max(dot(Normal, Light), 0.0);
float Ks = pow(max(dot(Half, Normal), 0.0),
               gl_FrontMaterial.shininess);

vec4 diffuse = Kd * gl_FrontLightProduct[0].diffuse;
if (dot(Normal, Light) < 0.0) f = 0.0;
vec4 specular = f * Ks *
gl_FrontLightProduct[0].specular;
vec4 ambient = gl_FrontLightProduct[0].ambient;

gl_FragColor = ambient + diffuse + specular;
}
```

效果对比



逐顶点光照



逐片段光照

采样器 (Samplers)



- 提供对纹理对象的访问
- 定义了1, 2, 和3维纹理以及立方体贴图的采样器
- 在着色器中:

```
uniform sampler2D myTexture;  
vec2 texcoord;  
vec4 texcolor = texture2D(mytexture, texcoord);
```

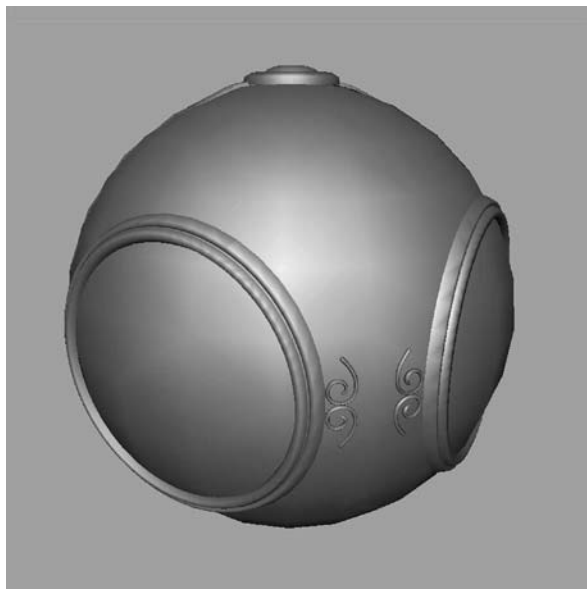
- 在应用程序中:

```
texMapLocation = glGetUniformLocation(myProg, "myTexture");  
glUniform1i(texMapLocation, 0);  
/* assigns to texture unit 0 */
```


片段着色器的应用



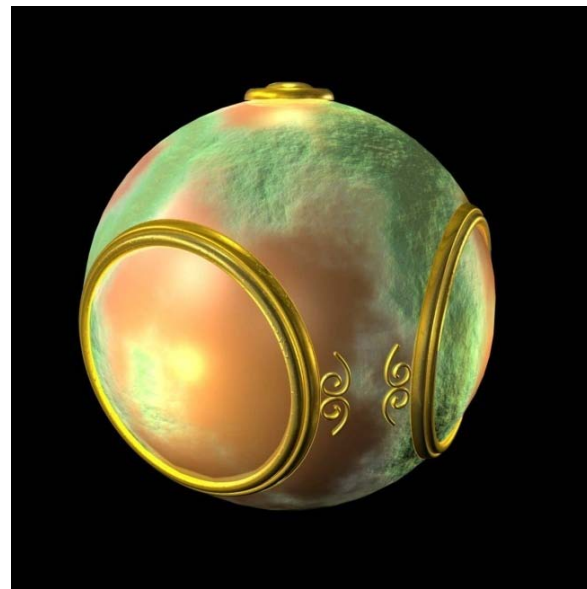
纹理映射



平滑明暗



环境映射



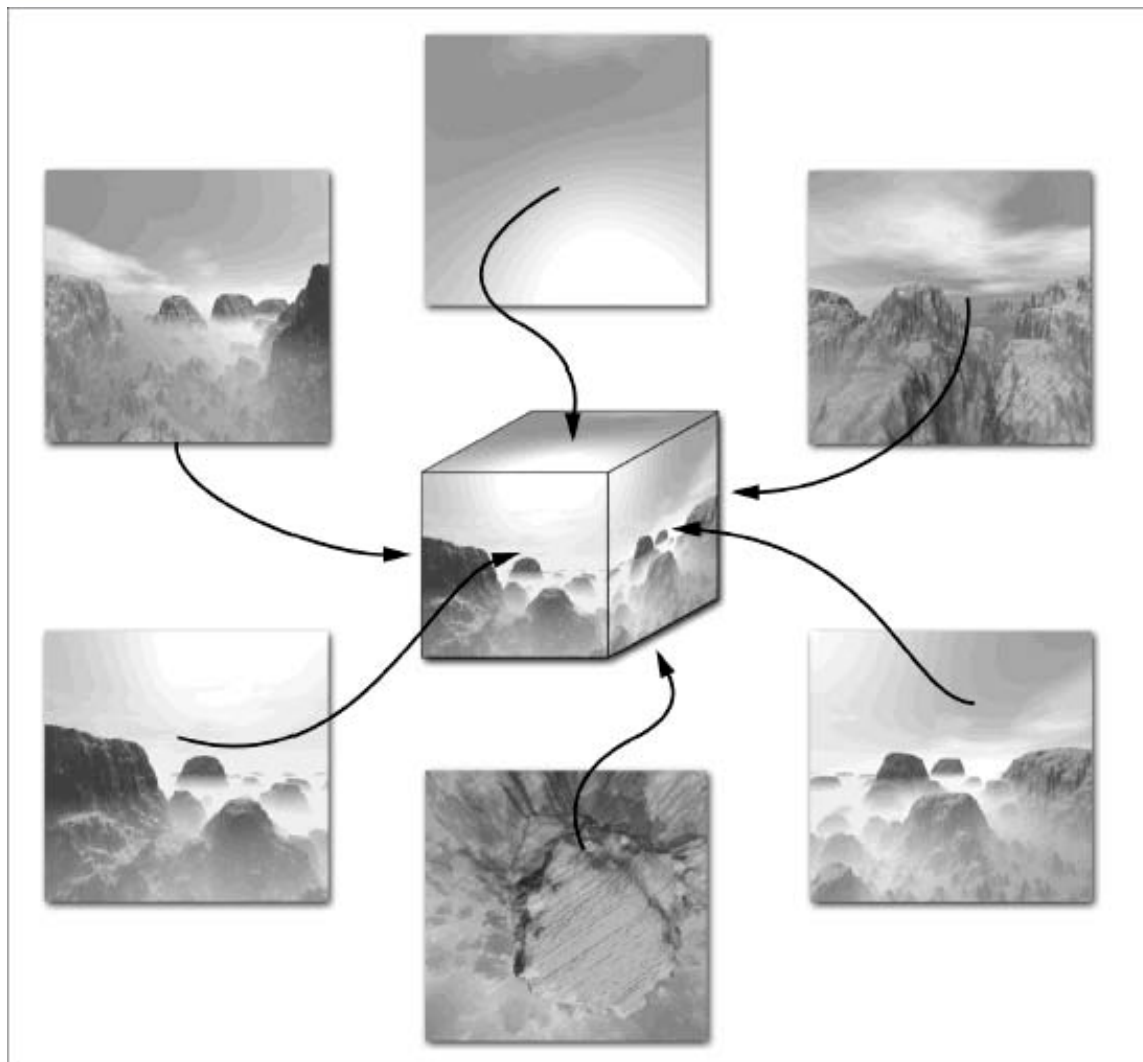
凹凸映射

立方体贴图

- 用六张2D纹理图组成立方图纹理
- OpenGL支持立方体贴图
- GLSL通过立方图采采样器来支持

```
vec4 texColor = textureCube(mycube, texcoord);
```
- 纹理坐标必须是3D的

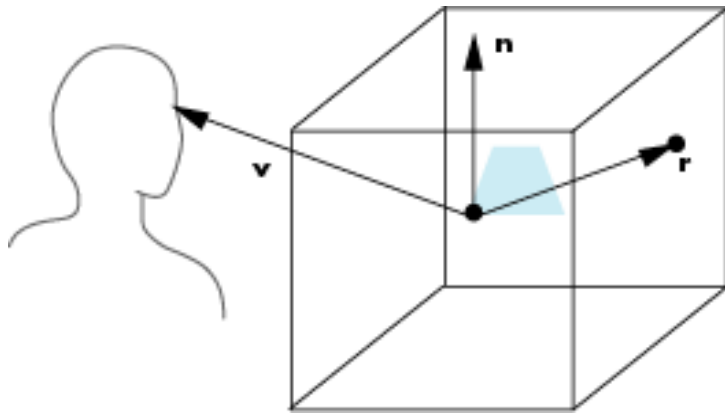
立方图纹理示例



环境映射



用反射向量在立方图中定位纹理



用着色器实现环境映射



- 通常在世界坐标系中计算环境映射，由于模型矩阵的作用，世界坐标系可能会不同于对象坐标系
 - 对象的位置和法向在对象坐标系中指定
 - 把模型矩阵作为uniform变量传递给着色器
- 也可用于反射贴图或折射贴图（例如模拟水）

反射贴图片段着色器

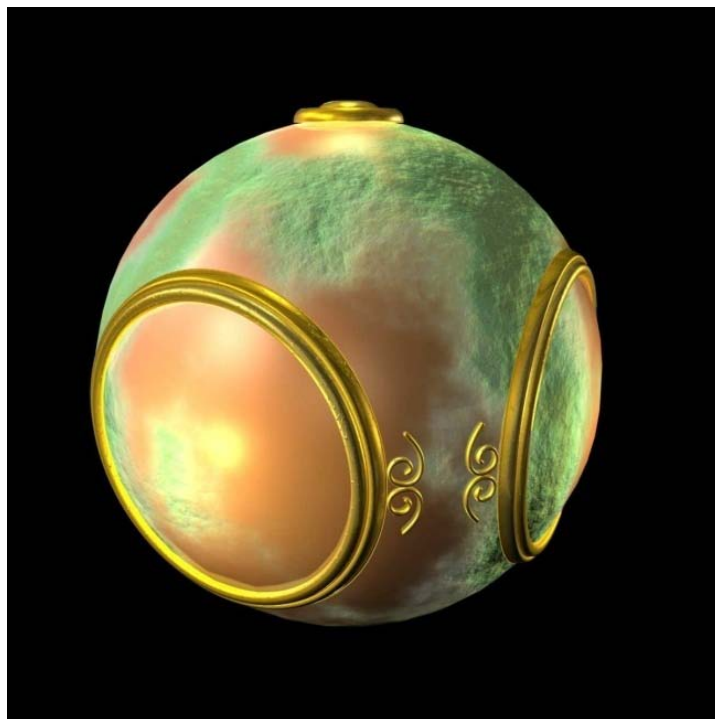


```
varying vec3 R;  
uniform samplerCube texMap;  
  
void main(void)  
{  
    gl_FragColor = textureCube(texMap, R);  
}
```

凹凸映射



- 对每个片段扰动法向
- 把扰动存储为纹理



Thanks for your attention!

