CS 4204 Computer Graphics

OpenGL Shading Language

Yong Cao

Virginia Tech

Reference: Ed Angle, Interactive Computer Graphics, University of New Mexico, class notes

Objectives

- ***Shader applications***
  - Vertex shaders
  - Fragment shaders
- ***Programming shaders***
  - Cg
  - GLSL

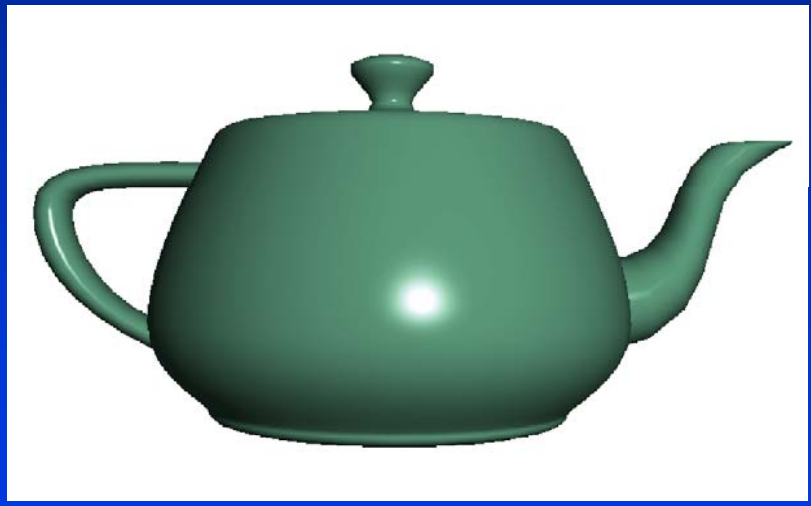
Vertex Shader Applications

- ***Moving vertices***
  - Morphing
  - Wave motion
  - Fractals
- ***Lighting***
  - More realistic models
  - Cartoon shaders

Fragment Shader Applications

Per fragment lighting calculations

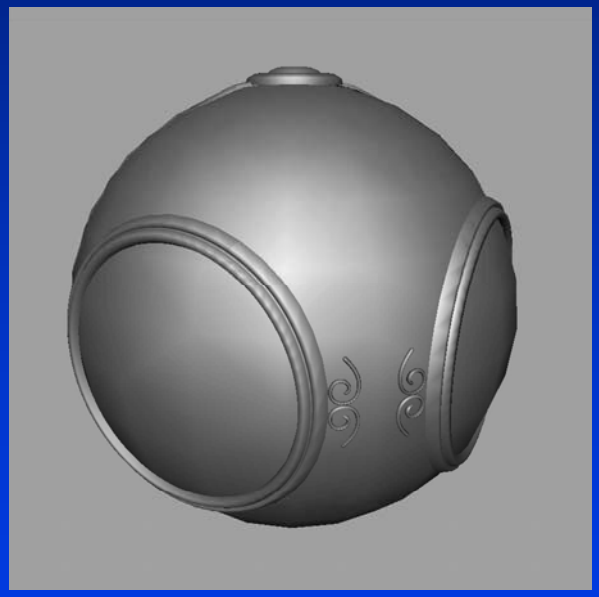
per vertex lighting



per fragment lighting

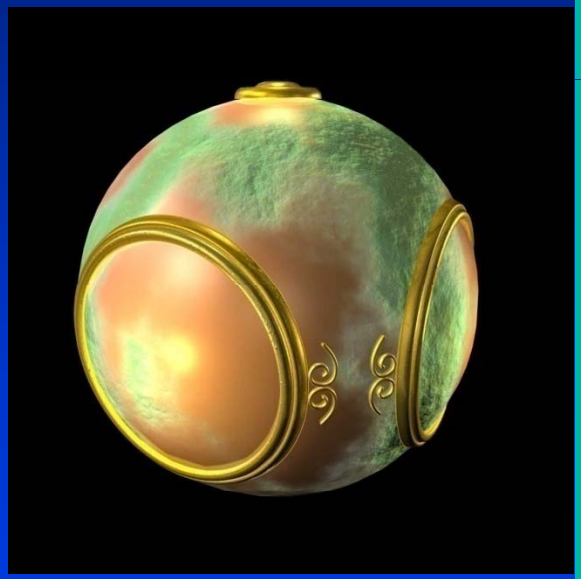
Fragment Shader Applications

Texture mapping



smooth shading

environment mapping



bump mapping

Writing Shaders

- ***First programmable shaders were programmed in an assembly-like manner***
- ***OpenGL extensions added for vertex and fragment shaders***
- ***Cg (C for graphics) C-like language for programming shaders***
  - Works with both OpenGL and DirectX
  - Interface to OpenGL complex
- ***OpenGL Shading Language (GLSL)***

GLSL

- ***OpenGL Shading Language***
- ***Part of OpenGL 2.0***
- ***High level C-like language***
- ***New data types***
  - Matrices
  - Vectors
  - Samplers
- ***OpenGL state available through built-in variables***

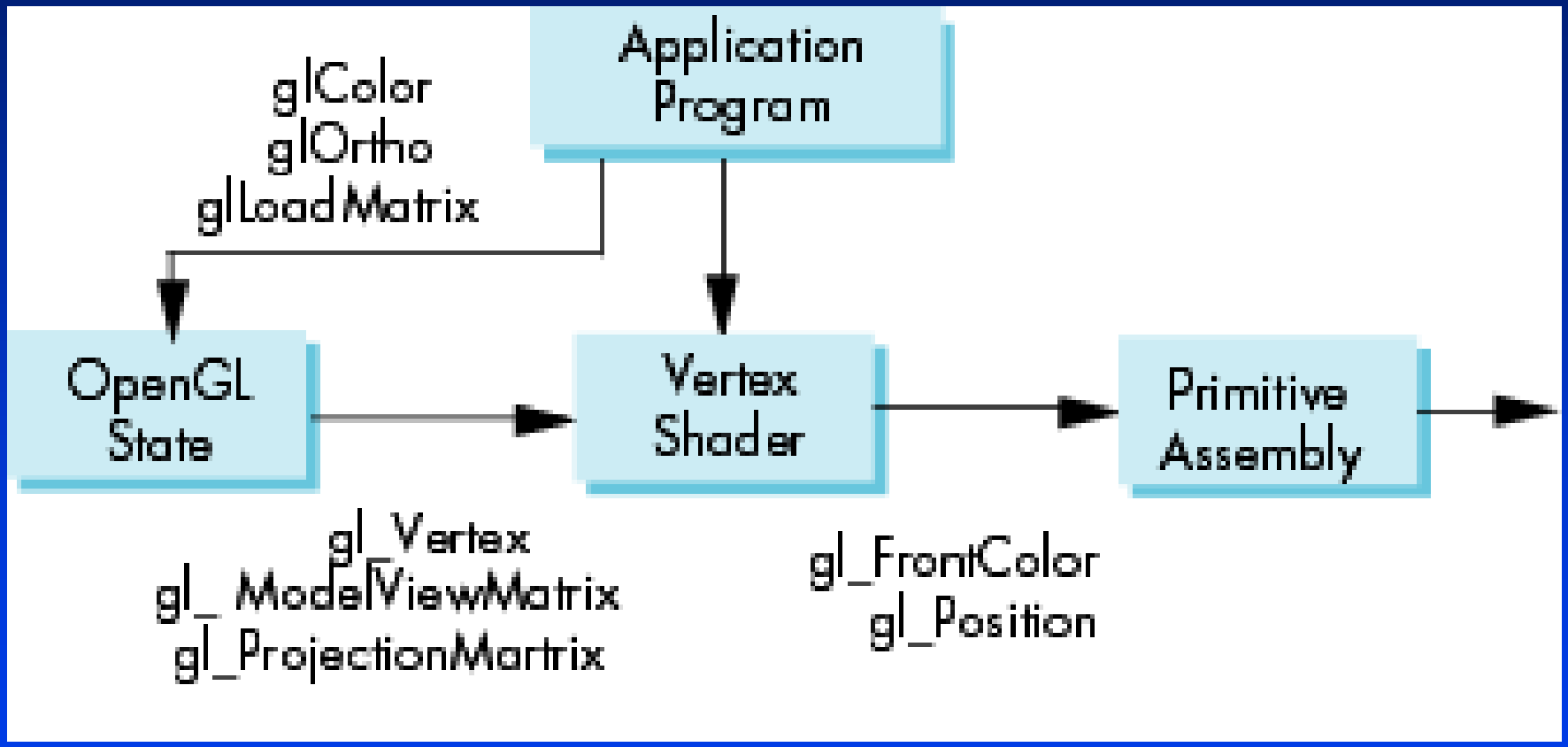
Simple Vertex Shader

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

void main(void)
{
    gl_Position = gl_ProjectionMatrix
        *gl_ModelViewMartrix*gl_Vertex;

    gl_FrontColor = red;
}
```

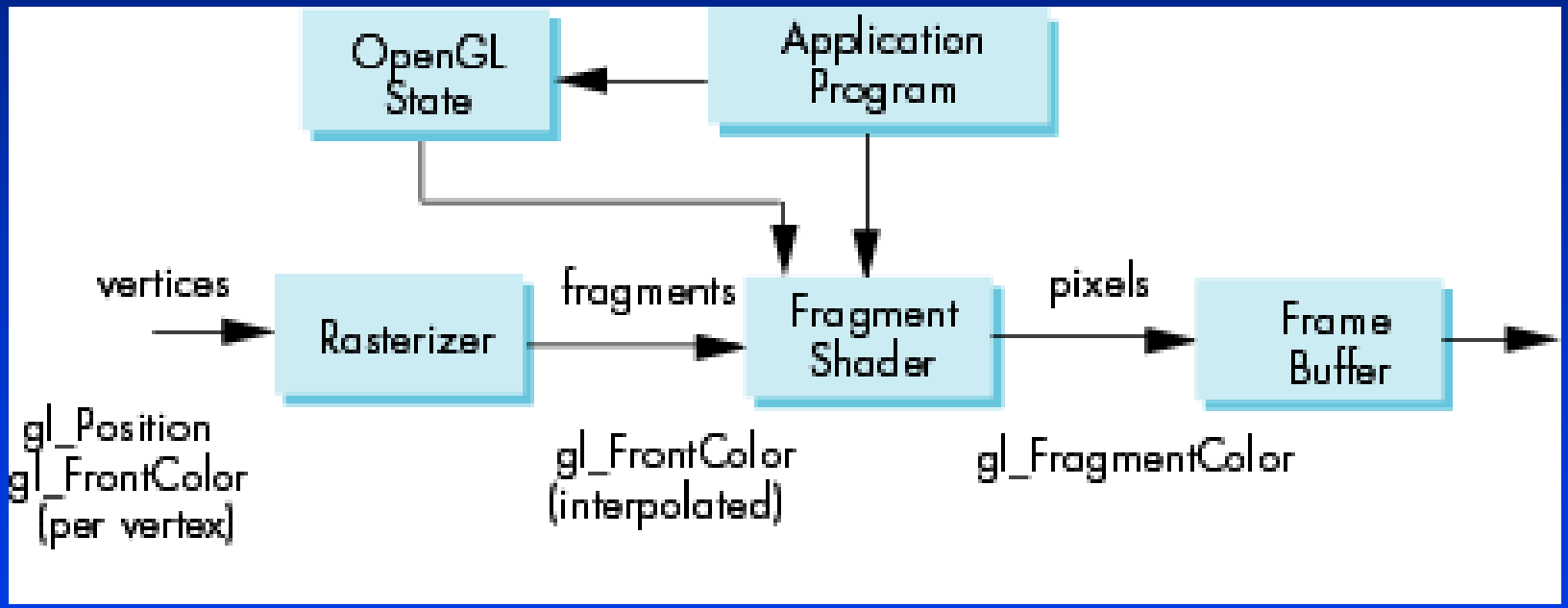
Execution Model



Simple Fragment Program

```
void main(void)
{
  gl_FragColor = gl_FrontColor;
}
```

Execution Model



Data Types

- ***C types: int, float, bool***
- ***Vectors:***
  - float vec2, vec 3, vec4
  - Also int (ivec) and boolean (bvec)
- ***Matrices: mat2, mat3, mat4***
  - Stored by columns
  - Standard referencing m[row][column]
- ***C++ style constructors***
  - vec3 a =vec3(1.0, 2.0, 3.0)
  - vec2 b = vec2(a)

Pointers

- *There are no pointers in GLSL*
- *We can use C structs which can be copied back from functions*
- *Because matrices and vectors are basic types they can be passed into and output from GLSL functions, e.g.*

  matrix3 func(matrix3 a)

Qualifiers

- ***GLSL has many of the same qualifiers such as `const` as C/C++***
- ***Need others due to the nature of the execution model***
- ***Variables can change***
  - Once per primitive
  - Once per vertex
  - Once per fragment
  - At any time in the application
- ***Vertex attributes are interpolated by the rasterizer into fragment attributes***

Attribute Qualifier

- ***Attribute-qualified variables can change at most once per vertex***
  - Cannot be used in fragment shaders
- ***Built in (OpenGL state variables)***
  - `gl_Color`
  - `gl_ModelViewMatrix`
- ***User defined (in application program)***
  - `attribute float temperature`
  - `attribute vec3 velocity`

Uniform Qualified

- *Variables that are constant for an entire primitive*
- *Can be changed in application outside scope of `glBegin` and `glEnd`*
- *Cannot be changed in shader*
- *Used to pass information to shader such as the bounding box of a primitive*

Varying Qualified

- ***Variables that are passed from vertex shader to fragment shader***
- ***Automatically interpolated by the rasterizer***
- ***Built in***
  - Vertex colors
  - Texture coordinates
- ***User defined***
  - Requires a user defined fragment shader

Example: Vertex Shader

```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
varying vec3 color_out;
void main(void)
{
  gl_Position =
   gl_ModelViewProjectionMatrix*gl_Vertex;
  color_out = red;
}
```

Required Fragment Shader

```
varying vec3 color_out;

void main(void)

{

  gl_FragColor = color_out;

}
```

Passing values

- *call by value-return*
- *Variables are copied in*
- *Returned values are copied back*
- *Three possibilities*
  - **in**
  - **out**
  - **inout**

Operators and Functions

- ***Standard C functions***
  - Trigonometric
  - Arithmetic
  - Normalize, reflect, length
- ***Overloading of vector and matrix types***

```
mat4 a;

vec4 b, c, d;

c = b*a; // a column vector stored as a 1d array

d = a*b; // a row vector stored as a 1d array
```

Swizzling and Selection

- ***Can refer to array elements by element using [] or selection (.) operator with***
  - x, y, z, w
  - r, g, b, a
  - s, t, p, q
  - **`a[2], a.b, a.z, a.p`** are the same
- ***Swizzling operator lets us manipulate components***

```
vec4 a;
a.yz = vec2(1.0, 2.0);
```

Objectives

- *Coupling GLSL to Applications*
- *Example applications*

Linking Shaders to OpenGL

- ***OpenGL Extensions (With GLEW library)***
  - ARB_shader_objects
  - ARB_vertex_shader
  - ARB_fragment_shader
- ***OpenGL 2.0***
  - Almost identical to using extensions
  - Avoids extension suffixes on function names

Program Object

- ***Container for shaders***
  - Can contain multiple shaders
  - Other GLSL functions

```
GLuint myProgObj;
myProgObj = glCreateProgram();
 /* define shader objects here */
glUseProgram(myProgObj);
glLinkProgram(myProgObj);
```

Reading a Shader

- *Shader are added to the program object and compiled*
- *Usual method of passing a shader is as a null-terminated string using the function* `glShaderSource`
- *If the shader is in a file, we can write a reader to convert the file to a string*

Shader Reader

```
char* readShaderSource(const char* shaderFile)
{
    struct stat statBuf;
    FILE* fp = fopen(shaderFile, "r");
    char* buf;

    stat(shaderFile, &statBuf);
    buf = (char*) malloc(statBuf.st_size + 1 * sizeof(char));
    fread(buf, 1, statBuf.st_size, fp);
    buf[statBuf.st_size] = '\0';
    fclose(fp);
    return buf;
}
```

Adding a Vertex Shader

```
GLint vShader;
GLunit myVertexObj;
GLchar vShaderfile[] = “my_vertex_shader”;
GLchar* vSource =
        readShaderSource(vShaderFile);
glShaderSource(myVertexObj,
          1, &vertexShaderFile, NULL);
myVertexObj =
          glCreateShader(GL_VERTEX_SHADER);
glCompileShader(myVertexObj);
glAttachObject(myProgObj, myVertexObj);
```

Vertex Attributes

- *Vertex attributes are named in the shaders*
- *Linker forms a table*
- *Application can get index from table and tie it to an application variable*
- *Similar process for uniform variables*

Vertex Attribute Example

```
GLint colorAttr;
colorAttr = glGetAttribLocation(myProgObj,
      "myColor");
/* myColor is name in shader */


GLfloat color[4];
glVertexAttrib4fv(colorAttrib, color);
/* color is variable in application */
```

Uniform Variable Example

```
GLint angleParam;
angleParam = glGetUniformLocation(myProgObj,
     "angle");
/* angle defined in shader */

/* my_angle set in application */
GLfloat my_angle;
my_angle = 5.0 /* or some other value */

glUniform1f(myProgObj, angleParam, my_angle);
```

Vertex Shader Applications

- ***Moving vertices***
  - Morphing
  - Wave motion
  - Fractals
- ***Lighting***
  - More realistic models
  - Cartoon shaders

Wave Motion Vertex Shader

```
uniform float time;
uniform float xs, zs;
void main()
{
float s;
s = 1.0 + 0.1*sin(xs*time)*sin(zs*time);
gl_Vertex.y = s*gl_Vertex.y;
gl_Position =
   gl_ModelViewProjectionMatrix*gl_Vertex;
}
```

Particle System

```
uniform vec3 init_vel;
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);
}
```

Modified Phong Vertex Shader I

```
void main(void)
/* modified Phong vertex shader (without distance term) */
{
    float f;
   /* compute normalized normal, light vector, view vector,
        half-angle vector in eye cordinates */
   vec3 norm = normalize(gl_NormalMatrix*gl_Normal);
   vec3 lightv = normalize(gl_LightSource[0].position
              -gl_ModelViewMatrix*gl_Vertex);
   vec3 viewv = -normalize(gl_ModelViewMatrix*gl_Vertex);
   vec3 halfv = normalize(lightv + norm);
   if(dot(lightv, norm) > 0.0) f = 1.0;
          else f = 0.0;
```

Modified Phong Vertex Shader II

```
    /* compute diffuse, ambient, and specular contributions */

    vec4 diffuse = max(0, dot(lightv, norm))*gl_FrontMaterial.diffuse
           *LightSource[0].diffuse;
    vec4 ambient = gl_FrontMaterial.ambient*LightSource[0].ambient;
    vec4 specular = f*gl_FrontMaterial.specular*
             gl_LightSource[0].specular)
            *pow(max(0, dot( norm, halfv)), gl_FrontMaterial.shininess);
     vec3 color = vec3(ambient + diffuse + specular)
     gl_FrontColor = vec4(color, 1);
     gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;
}
```

Pass Through Fragment Shader

```
/* pass-through fragment shader */
void main(void)
{
    gl_FragColor = gl_FrontColor;
}
```

Vertex Shader for per Fragment Lighting

```
/* vertex shader for per-fragment Phong shading */
varying vec3 normale;
varying vec4 positione;
void main()
{
   normale = gl_NormalMatrixMatrix*gl_Normal;
   positione = gl_ModelViewMatrix*gl_Vertex;
   gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;
}
```

Fragment Shader for Modified Phong Lighting I

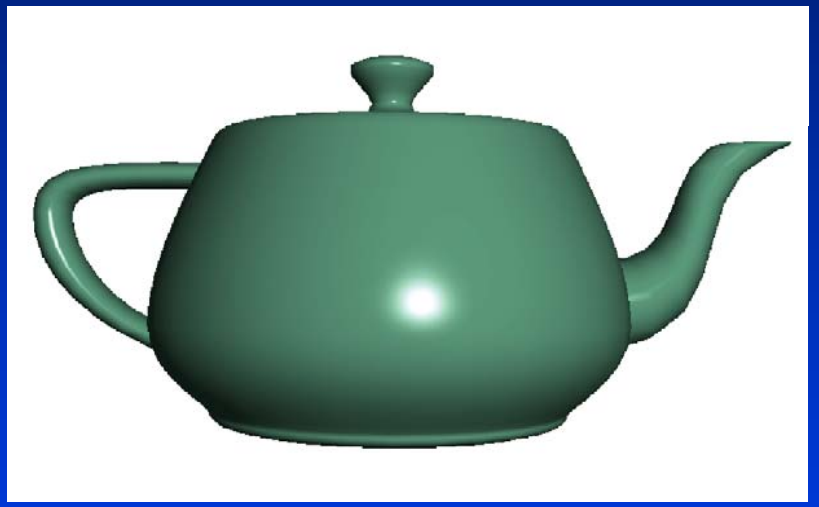
```
varying vec3 normale;
varying vec4 positione;
void main()
{
vec3 norm = normalize(normale);
vec3 lightv = normalize(gl_LightSource[0].position-positione.xyz);
vec3 viewv = normalize(positione);
vec3 halfv = normalize(lightv + viewv);
vec4 diffuse = max(0, dot(lightv, viewv))
   *gl_FrontMaterial.diffuse*gl_LightSource[0].diffuse;
vec4 ambient = gl_FrontMaterial.ambient*gl_LightSource[0].ambient;
```

Fragment Shader for Modified Phong Lighting II

```
int f;
if(dot(lightv, viewv)> 0.0) f =1.0);
else f = 0.0;
vec3 specular = f*pow(max(0, dot(norm, halfv),
        gl_FrontMaterial.shininess)
       *gl_FrontMaterial.specular*gl_LightSource[0].specular);
vec3 color = vec3(ambient + diffuse + specular);
gl_FragColor = vec4(color, 1.0);
}
```

Vertex vs Fragment Shader

per vertex lighting



per fragment lighting

Samplers

- *Provides access to a texture object*
- *Defined for 1, 2, and 3 dimensional textures and for cube maps*
- *In shader:*

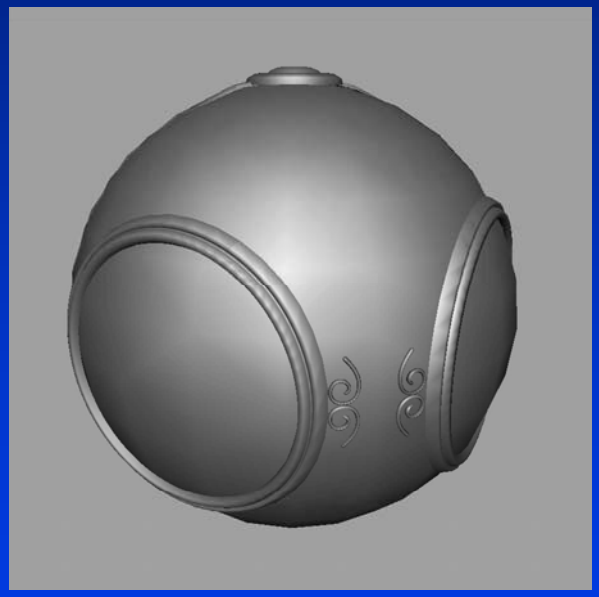
```
uniform sampler2D myTexture;
Vec2 texcoord;
Vec4 texcolor = texture2D(mytexture, texcoord);
```

- *In application:*

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

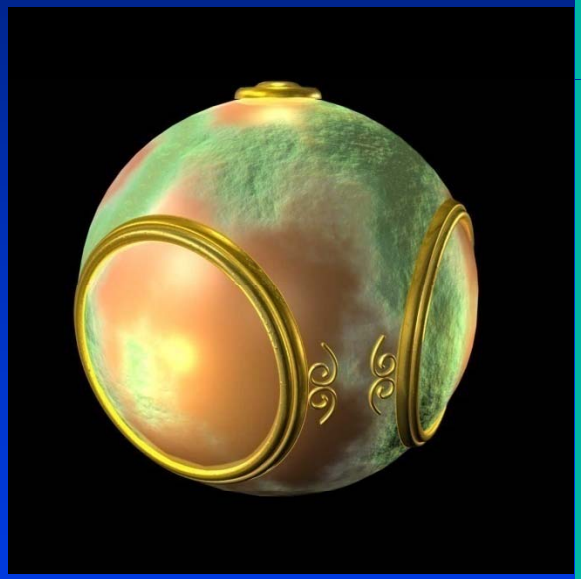
Fragment Shader Applications

Texture mapping



smooth shading

environment mapping



bump mapping

Cube Maps

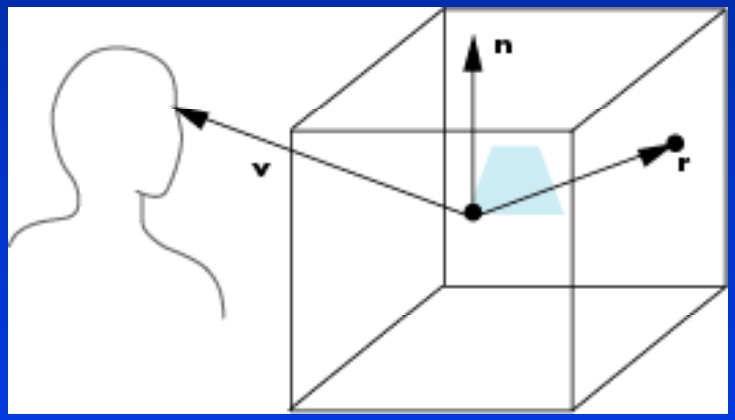
- ***We can form a cube map texture by defining six 2D texture maps that correspond to the sides of a box***
- ***Supported by OpenGL***
- ***Also supported in GLSL through cubemap sampler***

```
vec4 texColor = textureCube(mycube, texcoord);
```

- ***Texture coordinates must be 3D***

Environment Map

Use reflection vector to locate texture in cube map



Environment Maps with Shaders

- ***Environment map usually computed in world coordinates which can differ from object coordinates because of modeling matrix***
  - May have to keep track of modeling matrix and pass it shader as a uniform variable
- ***Can also use reflection map or refraction map (for example to simulate water)***

Environment Map Vertex Shader

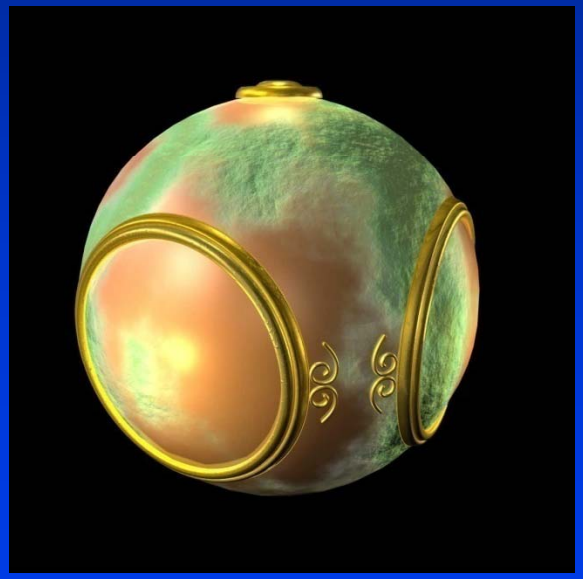
```
uniform mat4 modelMat;
uniform mat3 invModelMat;
uniform vec4 eyew;
void main(void)
{
   vec4 positionw = modelMat*gl_Vertex;
   vec3 normw = normalize(gl_Normal*invModelMat.xyz);
   vec3 lightw = normalize(eyew.xyz-positionw.xyz);
   eyew = reflect(normw, eyew);
   gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;
}
```

Environment Map Fragment Shader

```
/* fragment shader for reflection map */
varying vec3 reflectw;
uniform samplerCube MyMap;
void main(void)
{
    gl_FragColor = textureCube(myMap, reflectw);
}
```

Bump Mapping

- *Perturb normal for each fragment*
- *Store perturbation as textures*



Normalization Maps

- ***Cube maps can be viewed as lookup tables 1-4 dimensional variables***
- ***Vector from origin is pointer into table***
- ***Example: store normalized value of vector in the map***
  - Same for all points on that vector
  - Use “normalization map” instead of normalization function
  - Lookup replaces sqrt, mults and adds