Vertex- and Fragment Shaders

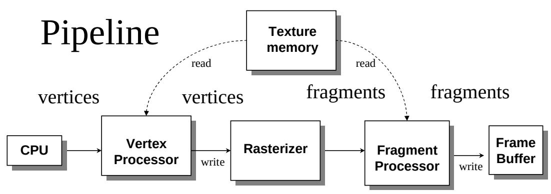
Slides modified by Ulf Assarsson.
Originals are mainly made by Edward Angel
but also by Magnus Bondesson.

Reading Material

MUST read
- These slides
- OH 231-256 by Magnus Bondesson

May also read:
- Angel, chapter 9, pages 449-493
- Excellent introduction to GLSL here:
  - http://www.lighthouse3d.com/opengl/glsl/index.php?intro
  - Or simply google on "GLSL Tutorial"

Pipeline

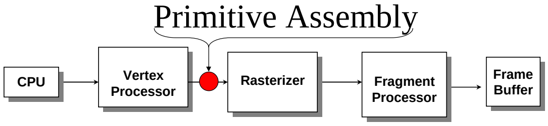


- Programmable pipeline
  - First introduced by NVIDIA GeForce 3
  - Supported by high-end commodity cards
    - NVIDIA, ATI, 3D Labs
  - Software Support
    - Direct X 8 , 9, 10 (HLSL)
    - OpenGL Extensions
    - OpenGL Shading Language (GLSL)
    - Cg

Vertex Shader

- Input data can be
  - (x,y,z,w) coordinates of a vertex (glVertex)
  - Normal vector
  - Texture Coordinates
  - RGBA color
  - OpenGL state
  - Additional user-defined data in GLSL
- Produces
  - Position in clip coordinates
  - Vertex color

Primitive Assembly



- Vertices are next assembled into objects
  - Polygons
  - Line Segements
  - Points
- Clipping
  - Against unit cube
  - Polygon clipping can create new vertices
- Viewport mapping

Fragment Shader

- Takes in output of rasterizer (fragments)
  - Vertex values have been interpolated over primitive by rasterizer
- Outputs a fragment
  - Color, e.g. from shading + textures
  - (Depth)
- Fragments still go through fragment tests
  - Hidden-surface removal
  - alpha

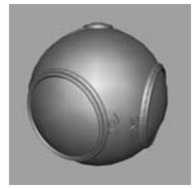
Fragment Operations

Done after fragment shading:

- Antialiasing
- Scissoring
- Alpha test
- Blending
- Dithering
- Logical Operation
- Masking
- (Fog)

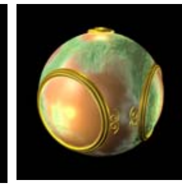
Fragment Shader Applications

Texture mapping



smooth shading

environment mapping



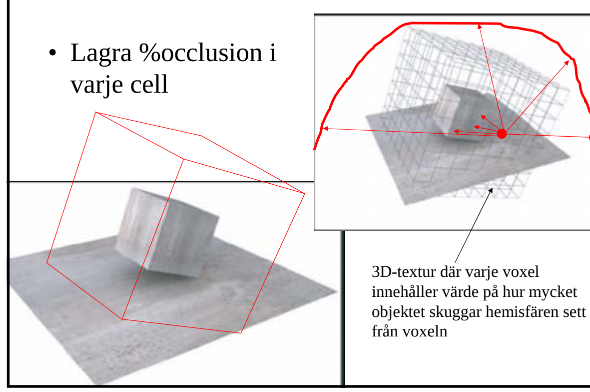
bump mapping

Writing Shaders

- If we use a programmable shader we must do *all* required functions of the fixed function processor
- 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)

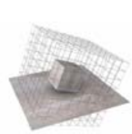
Contact Shadows

- Lagra %occlusion i varje cell



3D-textur där varje voxel innehåller värde på hur mycket objektet skuggar hemisfären sett från voxeln

```
!!ARBfp1.0
        # input
        PARAM World[4] = { program.local[0..3] };
        PARAM objID     = program.local[4];
        PARAM outval   = { 0.0, 0.0, 0.0, 1.0 };
        # Textures
        define PositionTexture          "texture[0]";
        define CubeTexture              "texture[1]";
ifdef nVidia
        # nVidia version
        ATTRIB FragmentPosition = fragment.position;
        # pixel position (x,y,z)
        define mTEX "TEX";
        define texture_target "RECT";
else
        # ATI version
        ATTRIB FragmentPosition = fragment.texcoord[0];
        # pixel position (x,y,z)
        define mTEX "TXP";
        define texture_target "2D";
endif
        TEMP Position;
        mTEX Position, FragmentPosition, PositionTexture, texture_target;     # get world position of fragment
        TEMP cubePos;
        # Transform position into cube coordinates
        DP4 cubePos.x, World[0], Position;
        DP4 cubePos.y, World[1], Position;
        DP4 cubePos.z, World[2], Position;
        TEX  result.color, cubePos, CubeTexture , 3D;
```



GLSL

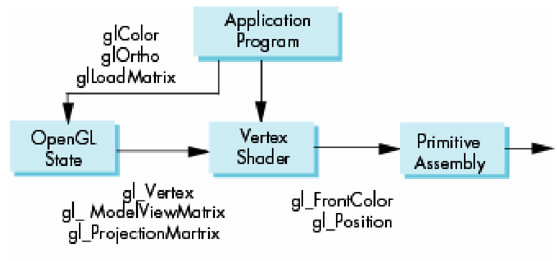
- OpenGL Shading Language
- Part of OpenGL 2.0
- High level C-like language
- New data types
  - Matrices
  - Vectors
  - Samplers
- OpenGL state available through built-in variables

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;
}
```

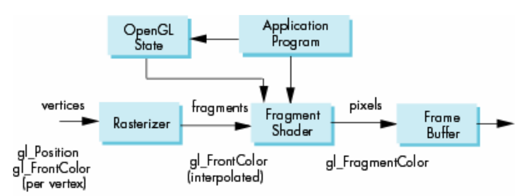
Execution Model



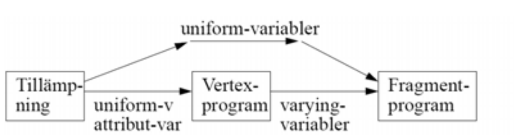
Simple Fragment Program

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

Execution Model



GLSL



Vertexprogrammet		Fragmentprogrammet	
In	**Ut**	**In**	**Ut**
gl_Vertex	gl_Position	gl_Color	gl_FragColor
gl_ModelViewMatrix	gl_TexCoord[i]	gl_SecondaryColor	
gl_ModelViewProjectionMatrix	gl_FrontColor	gl_TexCoord[i]	
gl_LightSource[i]	gl_BackColor	gl_FrontMaterial	
gl_MultiTexCoord0-7		gl_BackMaterial	
gl_Normal		gl_LightSource[i]	
gl_NormalMatrix		...	
gl_Color			

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

Qualifiers

Qualifiers give a special meaning to the variable. In GLSL the following qualifiers are available:

- **const** - the declaration is of a compile time constant
- **attribute** – (only used in vertex shaders, and read-only in shader) global variables that may change per vertex, that are passed from the OpenGL application to vertex shaders
- **uniform** – (used both in vertex/fragment shaders, read-only in both) global variables that may change per primitive (may not be set inside glBegin,/glEnd)
- **varying** - used for interpolated data between a vertex shader and a fragment shader. Available for writing in the vertex shader, and read-only in a fragment shader.

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_MultiTexCoord0`**
- 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; (e.g. instead of gl_FrontColor = red;)
}
```

Required Fragment Shader

```
varying vec3 color_out;
void main(void)
{
  gl_FragColor = color_out;1
}
```

[1]instead of gl_FragColor = gl_FrontColor;

Built-in Uniforms

```
uniform   mat4  gl_ModelViewMatrix;
uniform   mat4  gl_ProjectionMatrix;
uniform   mat4  gl_ModelViewProjectionMatrix;
uniform   mat3  gl_NormalMatrix;
uniform   mat4  gl_TextureMatrix[n];

struct gl_MaterialParameters {
  vec4  emission;
  vec4  ambient;
  vec4  diffuse;
  vec4  specular;
  float shininess;
};
uniform gl_MaterialParameters gl_FrontMaterial;
uniform gl_MaterialParameters gl_BackMaterial;
```

Built-in Uniforms

```
struct gl_LightSourceParameters {
  vec4  ambient;
  vec4  diffuse;
  vec4  specular;
  vec4  position;
  vec4  halfVector;
  vec3  spotDirection;
  float spotExponent;
  float spotCutoff;
  float spotCosCutoff;
  float constantAttenuation
  float linearAttenuation
  float quadraticAttenuation
};
Uniform gl_LightSourceParameters
   gl_LightSource[gl_MaxLights];
```

Uniform Variable Example

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

// set angle to 5.0
glUniform1f(myProgObj, angleParam, 5.0);
```

Vertex Attribute Example

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

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

Uniform Variables

Assume that a shader with the following variables is being used:

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

In the application, the code for setting the variables could be:

```
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};

Get → loc1 = glGetUniformLocationARB(p,"specIntensity");
Set → glUniform1fARB(loc1,specIntensity);
loc2 = glGetUniformLocationARB(p,"specColor");
glUniform4fvARB(loc2,1,sc);
loc3 = glGetUniformLocationARB(p,"t");
glUniform1fvARB(loc3,2,threshold);
loc4 = glGetUniformLocationARB(p,"colors");
glUniform4fvARB(loc4,3,colors);
```

Built-in Varyings

```
varying   vec4  gl_FrontColor      // vertex
varying   vec4  gl_BackColor;      // vertex
varying   vec4  gl_FrontSecColor;  // vertex
varying   vec4  gl_BackSecColor;   // vertex

varying   vec4  gl_Color;          // fragment
varying   vec4  gl_SecondaryColor; // fragment

varying   vec4  gl_TexCoord[];     // both
varying   float gl_FogFragCoord;   // both
```

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);
```

Operators

- grouping: ()
- array subscript: []
- function call and constructor: ()
- field selector and swizzle: .
- postfix: ++ --
- prefix: ++ -- + - !

Operators

- binary: * / + -
- relational: < <= > >=
- equality: == !=
- logical: && ^^ ||
- selection: ?:
- assignment: = *= /= += -=

Reserved Operators

- prefix: ~
- binary: %
- bitwise: << >> & ^ |
- assignment: %= <<= >>= &= ^= |=

Scalar/Vector Constructors

- **No casting**

```
float f; int i; bool b;
vec2 v2; vec3 v3; vec4 v4;

vec2(1.0 ,2.0)
vec3(0.0 ,0.0 ,1.0)
vec4(1.0 ,0.5 ,0.0 ,1.0)
vec4(1.0)                  // all 1.0
vec4(v2 ,v2)
vec4(v3 ,1.0)

float(i)
int(b)
```

Matrix Constructors

```
vec4 v4; mat4 m4;

mat4( 1.0, 2.0, 3.0, 4.0,
      5.0, 6.0, 7.0, 8.0,
      9.0, 10., 11., 12.,
      13., 14., 15., 16.)   // row major

mat4( v4, v4, v4, v4)
mat4( 1.0)                  // identity matrix
mat3( m4)                   // upper 3x3
vec4( m4)                   // 1st column
float( m4)                  // upper 1x1
```

Accessing components

- component accessor for vectors
  - xyzw rgba stpq [i]
- component accessor for matrices
  - [i] [i][j]

Swizzling & Smearing

- R-values

```
vec2 v2;
vec3 v3;
vec4 v4;

v4.wzyx // swizzles, is a vec4
v4.bgra // swizzles, is a vec4
v4.xxxx // smears x, is a vec4
v4.xxx  // smears x, is a vec3
v4.yyxx // duplicates x and y, is a vec4
v2.yyyy // wrong: too many components for type
```

Flow Control

- expression ? trueExpression : falseExpression
  a = (a>b) ? a: b;
- if, if-else
  ```
  if() {
    ...
  }
  ```
- for, while, do-while
  ```
  for() {        while() {        do {
    ...            ...              ...
  }              }                } while();
  ```
- return, break, continue
- discard (fragment only)

Built-in functions

- Angles & Trigonometry
  - **radians, degrees, sin, cos, tan, asin, acos, atan**
- Exponentials
  - **pow, exp2, log2, sqrt, inversesqrt**
- Common
  - **abs, sign, floor, ceil, fract, mod, min, max, clamp**

Built-in functions

- Interpolations
  - **mix**(x,y,a) **x*(1.0-a) + y*a**)
  - **step**(edge,x) **x <= edge ? 0.0 : 1.0**
  - **smoothstep**(edge0,edge1,x)

    t = (x-edge0)/(edge1-edge0);

    t = clamp(t, 0.0, 1.0);

    return t*t*(3.0-2.0*t);

Built-in functions

- Geometric
  - **length, distance, cross, dot, normalize, faceForward, reflect**
- Matrix
  - **matrixCompMult**
- Vector relational
  - **lessThan, lessThanEqual, greaterThan, greaterThanEqual, equal, notEqual, notEqual, any, all**

Built-in functions

- Texture
  - **texture1D, texture2D, texture3D, textureCube**
  - **texture1DProj, texture2DProj, texture3DProj, textureCubeProj**
  - **shadow1D, shadow2D, shadow1DProj, shadow2Dproj**
- Vertex
  - **ftransform, e.g. gl_Position = ftransform();**

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 */
```

Loading Textures

- Bind textures to different units as usual

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D,myFirstTexture);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D,mySecondTexture);
```

- Then load corresponding sampler with texture unit that texture is bound to

```
glUniform1iARB(glGetUniformLocationARB(
  programObject,”myFirstSampler”),0);
glUniform1iARB(glGetUniformLocationARB(
  programObject,”mySecondSampler”),1);
```

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;
}
```

Loading the shaders

```
void setShaders() {

     GLuint v = glCreateShader(GL_VERTEX_SHADER);
     GLuint f = glCreateShader(GL_FRAGMENT_SHADER);

     char * vs = textFileRead("toon.vert");
     char * fs = textFileRead("toon.frag");

     glShaderSource(v, 1, (const char **) &vs, NULL);
     glShaderSource(f, 1, (const char **)  &fs, NULL);
     free(vs);free(fs);

     glCompileShader(v);
     glCompileShader(f);

     GLuint p = glCreateProgram();
     glAttachShader(p,f);
     glAttachShader(p,v);

     glLinkProgram(p);
     glUseProgram(p);                    // 0 disables vertex/fragment shaders
}
```

Vertex Shader Applications

- Moving vertices
  - –Morphing
  - –Wave motion
  - –Fractals
- Lighting
  - –More realistic models
  - –Cartoon shader

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);
}
```

Vertex vs Fragment Shader

per vertex lighting

per fragment lighting

Lighting Calculations

- Done on a per-vertex basis Phong model
  $I = k_d I_d \mathbf{l} \cdot \mathbf{n} + k_s I_s (\mathbf{v} \cdot \mathbf{r})^{\alpha} + k_a I_a$

- Phong model requires computation of **r** and **v** at every vertex

Calculating the Reflection Term

angle of incidence = angle of reflection

$\cos \theta_i = \cos \theta_r$ or $\mathbf{r} \cdot \mathbf{n} = \mathbf{l} \cdot \mathbf{n}$

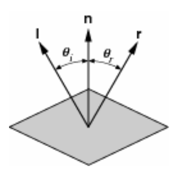
r, **n**, and **l** are coplanar

$\mathbf{r} = \alpha \mathbf{l} + \beta \mathbf{n}$

normalize

$1 = \mathbf{r} \cdot \mathbf{r} = \mathbf{n} \cdot \mathbf{n} = \mathbf{l} \cdot \mathbf{l}$

solving: $\mathbf{r} = 2(\mathbf{l} \cdot \mathbf{n})\mathbf{n} - \mathbf{l}$



Halfway Vector

Blinn proposed replacing $\mathbf{v} \cdot \mathbf{r}$ by $\mathbf{n} \cdot \mathbf{h}$ where

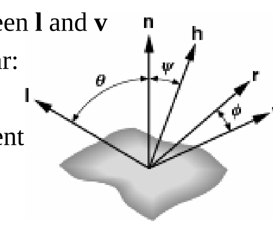
$\mathbf{h} = (\mathbf{l}+\mathbf{v})/|\mathbf{l} + \mathbf{v}|$

$(\mathbf{l}+\mathbf{v})/2$ is halfway between **l** and **v**

If **n**, **l**, and **v** are coplanar:

$\psi = \phi/2$

Must then adjust exponent so that $(\mathbf{n} \cdot \mathbf{h})^{e'} \approx (\mathbf{r} \cdot \mathbf{v})^{e}$



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);
}
```

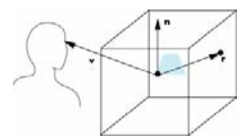
Cube Maps

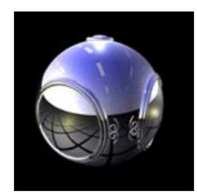
- We can form a cube map texture by defining six 2D texture maps that correspond to the sides of a box
- Supported by OpenGL
- Also supported in GLSL through cubemap sampler

  `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

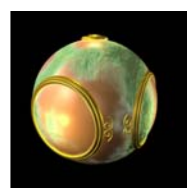
```
uniform mat4 modelMat;
uniform mat3 invModelMat;
varying vec4 reflectw;
void main(void)
{
  vec4 positionw = modelMat*gl_Vertex;
  vec3 normw = normalize(gl_Normal*invModelMat.xyz);
  vec3 lightw = normalize(eyew.xyz-positionw.xyz);
  reflectw = reflect(normw, eyew);
  gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;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

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

Per-Vertex Operations

- Vertex locations are transformed by the model-view matrix into eye coordinates
- Normals must be transformed with the inverse transpose of the model-view matrix so that v·n=v’ ·n’ in both spaces
  - Assumes there is no scaling
  - May have to use autonormalization
- Textures coordinates are generated if autotexture enabled and the texture matrix is applied