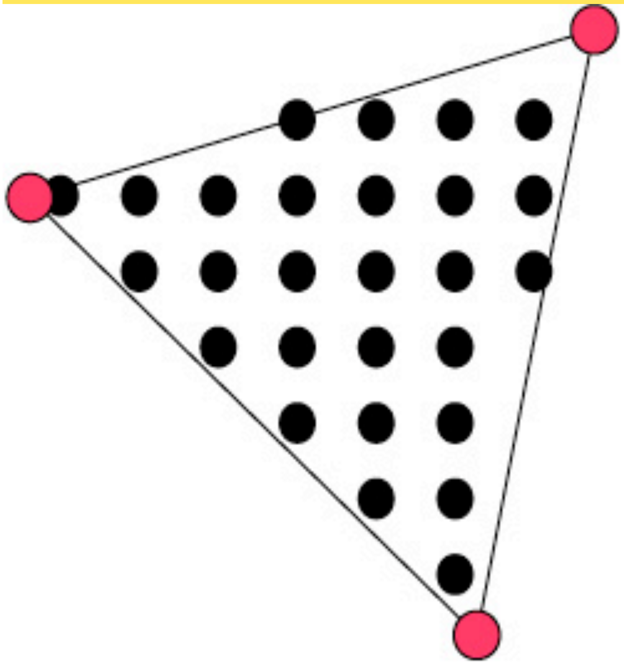
Vertex-, Geometry- and Fragment Shaders

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

Excellent introduction to GLSL here:

- http://www.lighthouse3d.com/opengl/glsl/index.php?intro
- Or simply google on ”GLSL Tutorial”

What is vertex and fragment (pixel) shaders?



- Memory: Texture memory (read + write) typically 256 Mb – 1GB
- Program size: unlimited instructions (but smaller is faster)
- Instructions: mul, rcp, mov,dp, rsq, exp, log, cmp, jnz…

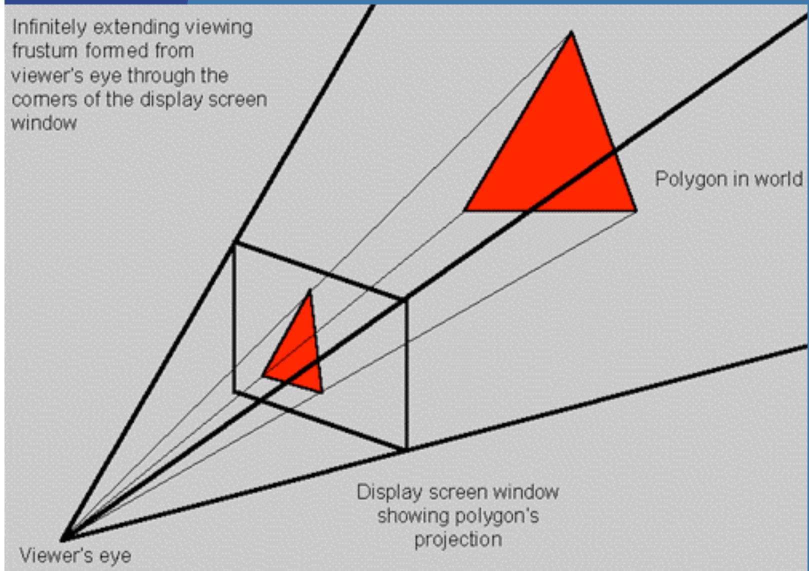
● (red) For each vertex, a vertex program (vertex shader) is executed

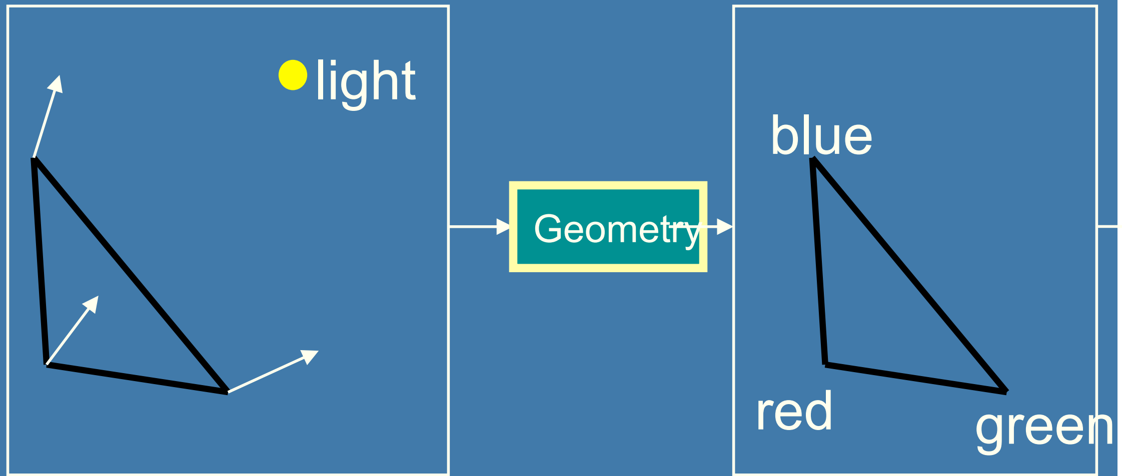
● (black) For each fragment (pixel) a fragment program (fragment shader) is executed

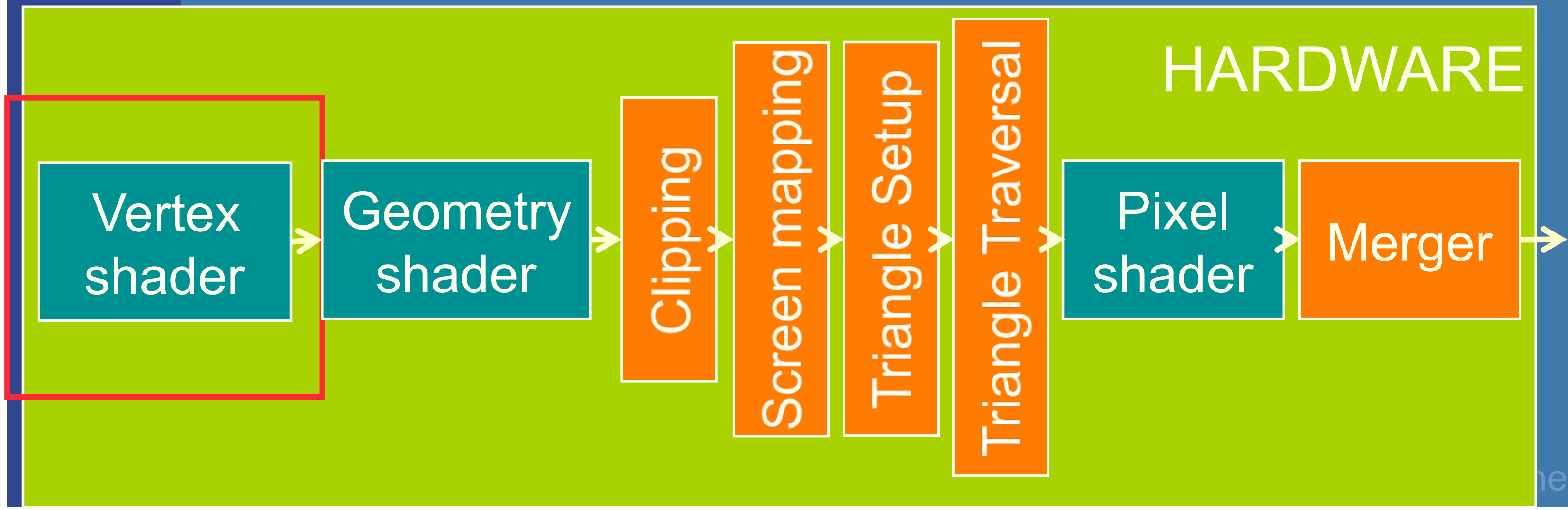
Hardware design

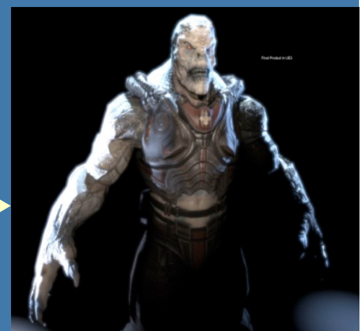
Vertex shader:

- Lighting (colors)
- Screen space positions









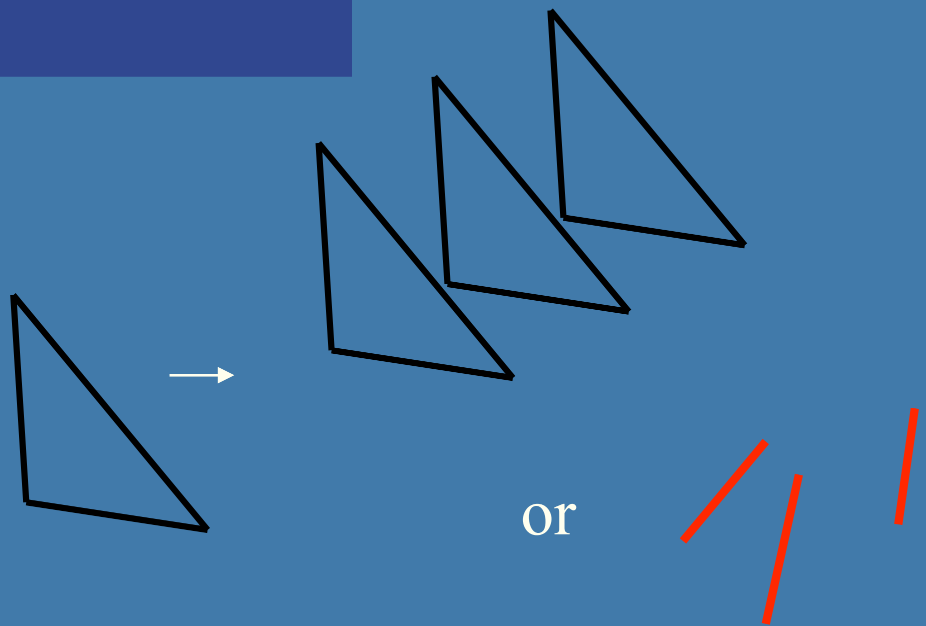
Display

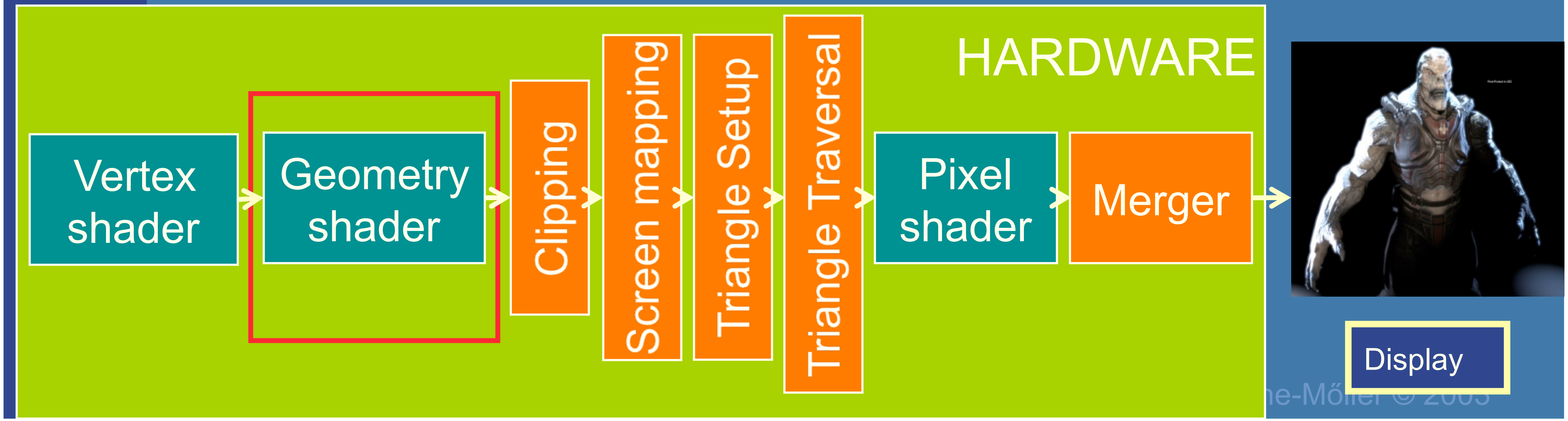
Hardware design

Geometry shader:

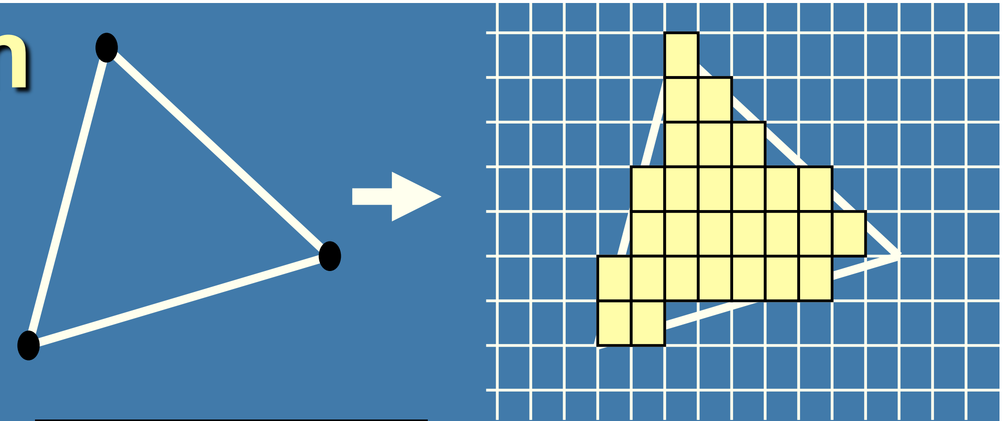
•One input primitive

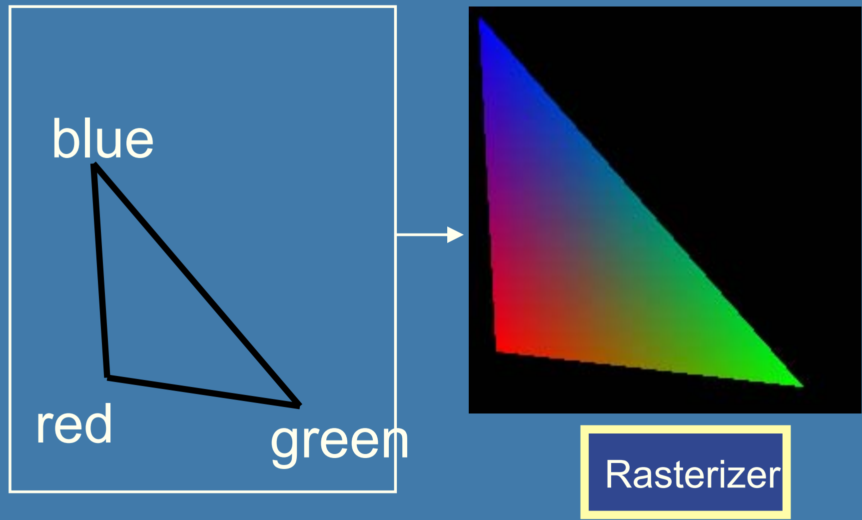
•Many output primitives



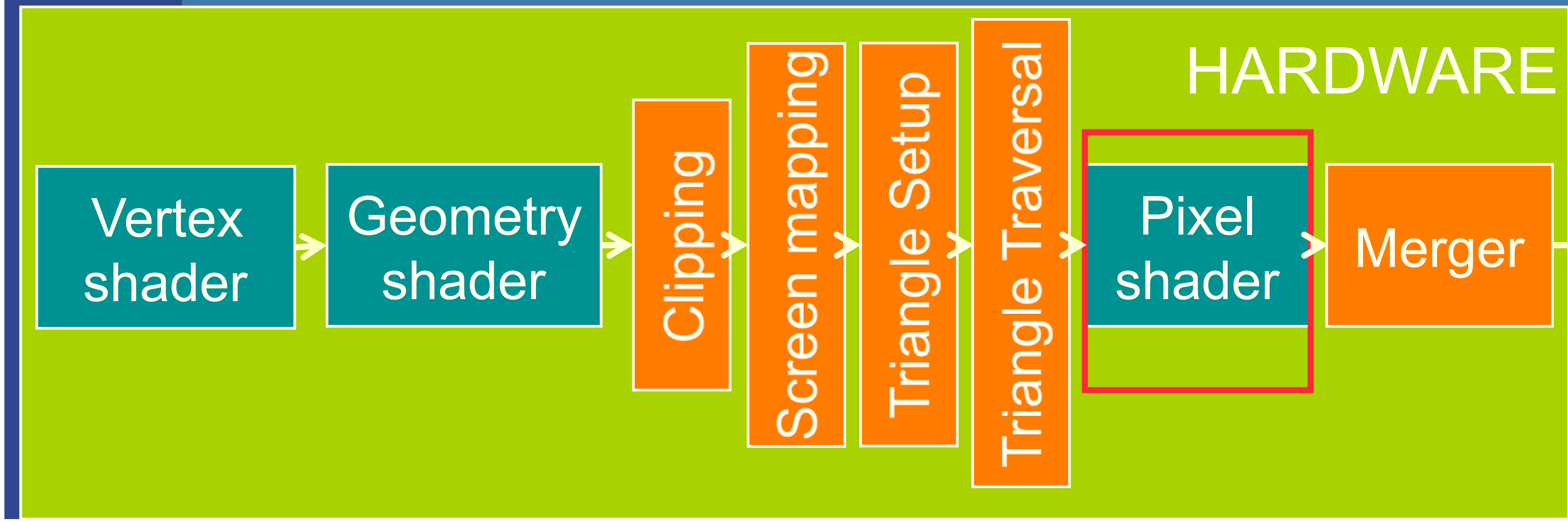


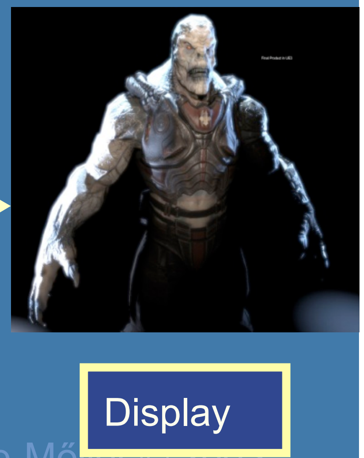
Hardware design





Pixel Shader:
Compute color using:
- Textures
- Interpolated data (e.g. Colors + normals)





Cg - ”C for Graphics” (NVIDIA)

```
if (slice >= 0.0h) {
    half gradedEta = BallData.ETA;
    gradedEta = 1.0h/gradedEta;  // test hack
    half3 faceColor = BgColor; // blown out - go to BG color
    half c1 = dot(-Vn,Nf);
    half cs2 = 1.0h-gradedEta*gradedEta*(1.0h-c1*c1);

    if (cs2 >= 0.0h) {
        half3 refVector = gradedEta*Vn+((gradedEta*c1-sqrt(cs2))*Nf);
        // now let's intersect with the iris plane
        half irisT = intersect_plane(IN.OPosition,refVector,planeEquation);
        half fadeT = irisT * BallData.LENS_DENSITY;
        fadeT = fadeT * fadeT;
        faceColor = DiffPupil.xxx;  // temporary (?)
        if (irisT > 0) {
            half3 irisPoint = IN.OPosition + irisT*refVector;
            half3 irisST = (irisScale*irisPoint) + half3(0.0h,0.5h,0.5h);
            faceColor = tex2D(ColorMap,irisST.yz).rgb;
        }
        faceColor = lerp(faceColor,LensColor,fadeT);
        hitColor = lerp(missColor,faceColor,smoothstep(0.0h,GRADE,slice));
    }
}
```

PixelShader 3.0

```
// if (-dir.z/|dir| > cos(PI/4)) t1 = zero
dp3 r6.w, r6, r6
rsq r6.w, r6.w
mad r0.w, -r6.z, r6.w, -CosPiOverFour
cmp r10.y, r0.w, Zero, r10.y

// set r10 to 0 if Disc <= 0
cmp r10.xy, -r7.w, Zero, r10

// compute r1 and r2 clipped
mad r1.xyz, r6, r10.x, r4                  // IP0
mad r2.xyz, r6, r10.y, r4                  // IP1

// project
rcp r11.w, r1.z
mad r1.xyz, r1, r11.w, NegZ                // P0
rcp r11.w, r2.z
mad r2.xyz, r2, r11.w, NegZ                // P1

// Compute area
texld r3, r1, ATan2Texture                 // theta0
texld r4, r2, ATan2Texture                 // theta1

crs r5.z,r1,r2                                  // z = 2
abs r5.z,r5.z

mov r3.y, r4.x
texld r4, r3, SphAreaTexture         // lookup theta/PI
```

← normalization (dp3 / rsq lines)

- Float, int
- Instructions operate on 1,2,3 or 4 components
  - x,y,z,w or
  - r,g,b,a
- Free Swizzling
- Only read from texture
- (Only write to pixel (8 output buffers))

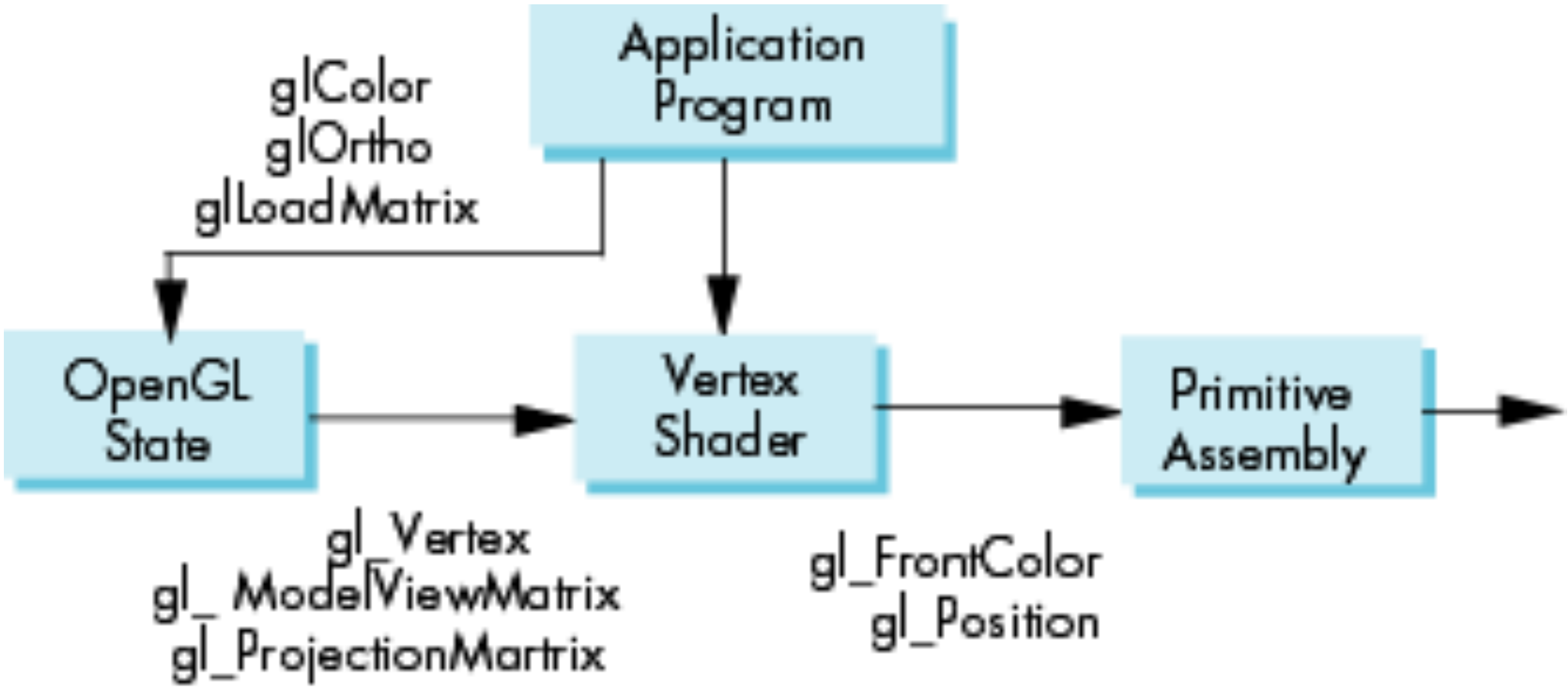
GLSL

- 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



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 (attributes + uniforms)
- Produces
    - Position in clip coordinates
    - Vertex color

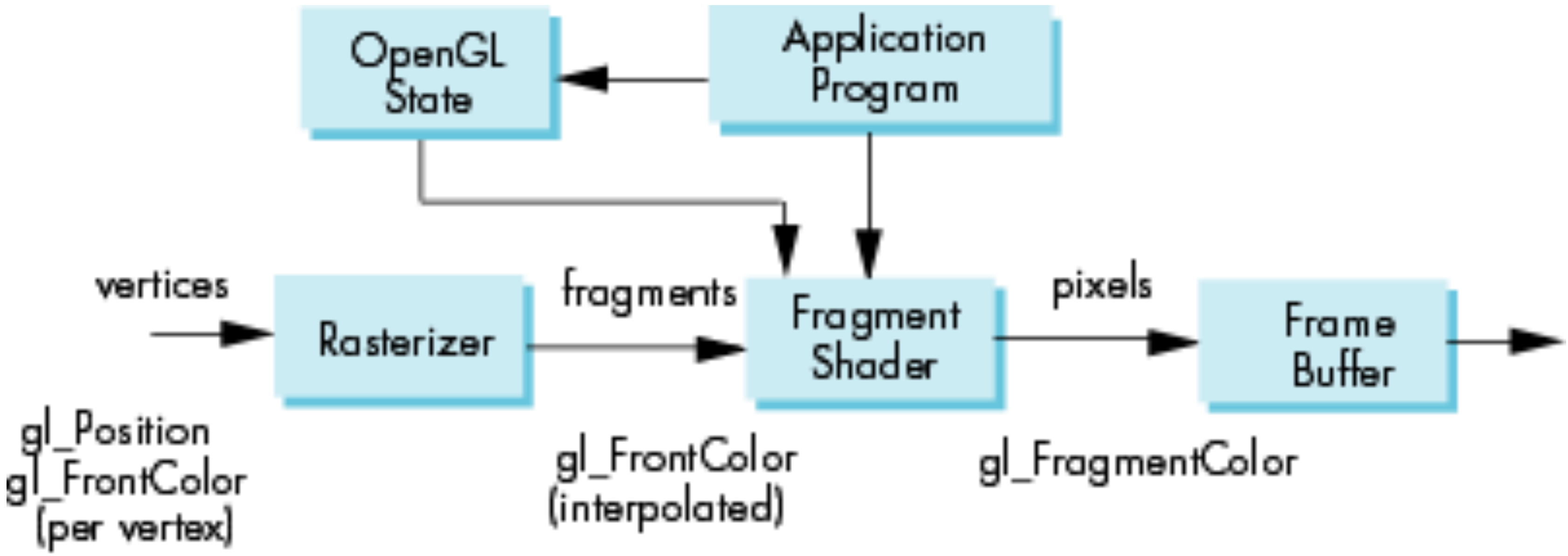
Fragment Shader

- 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

Simple Fragment Program

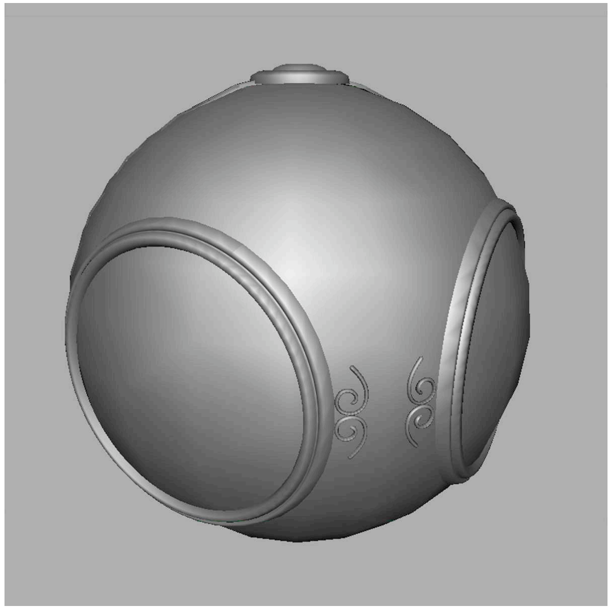
```
void main(void)
{
  gl_FragColor = gl_FrontColor;
}
```

Execution Model



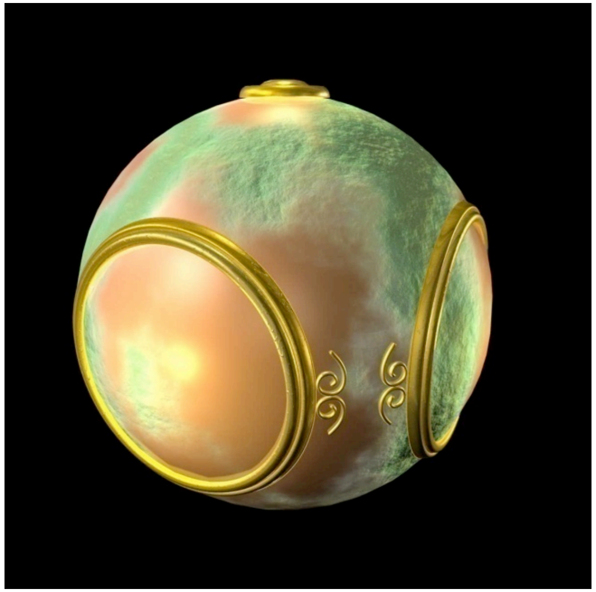
Fragment Shader Applications

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)

GLSL

Built in State Variables

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, vec3, 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)

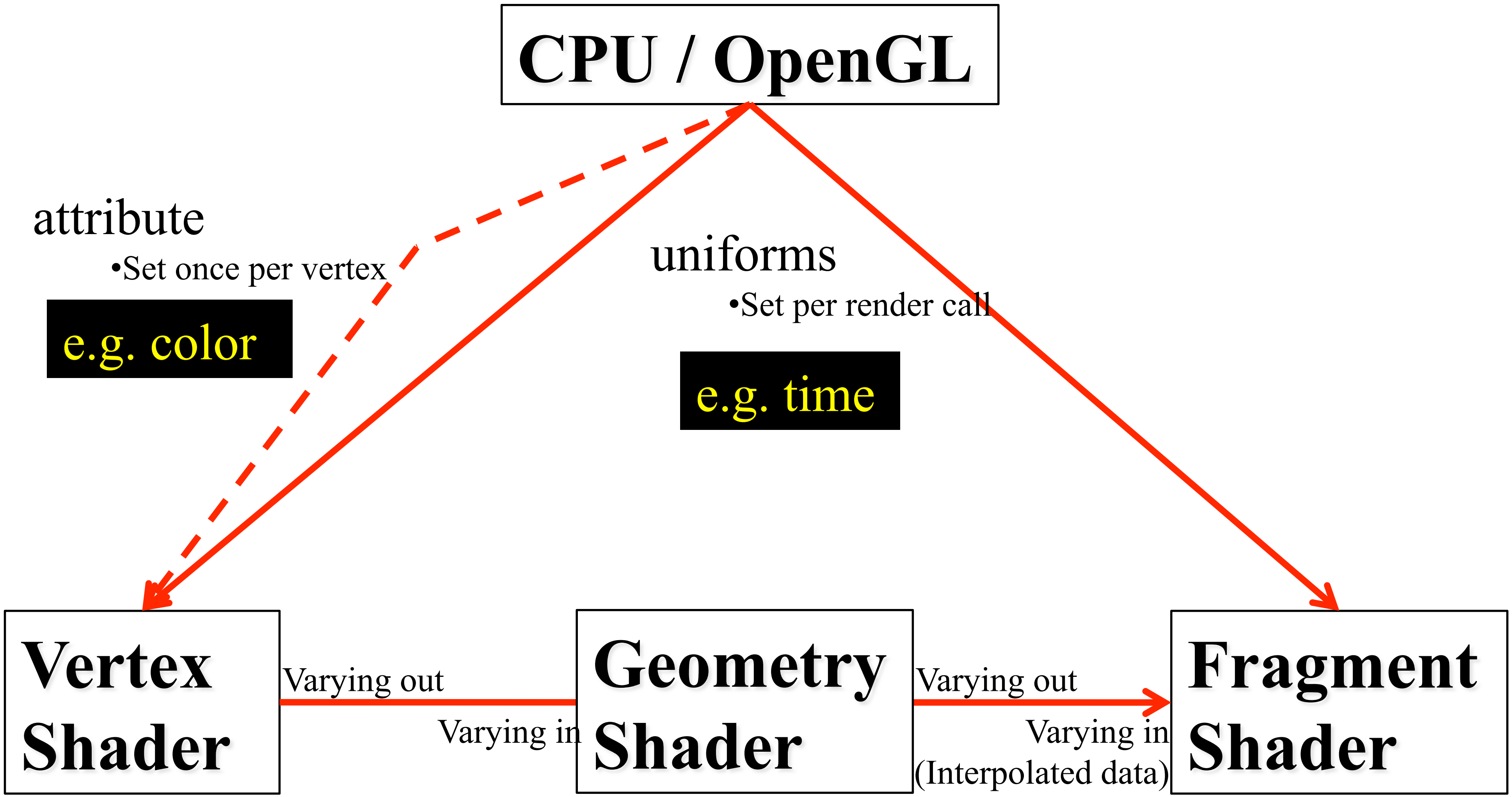
Pointers

- 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


CPU / OpenGL
attribute
•Set once per vertex
e.g. color
uniforms
•Set per render call
e.g. time
Vertex Shader
Varying out
Varying in
Geometry Shader
Varying out
Varying in
(Interpolated data)
Fragment Shader

Geometry shader built-in outputs:

- varying out vec4 gl_FrontColor;
- varying out vec4 gl_BackColor;
- varying out vec4 gl_FrontSecondaryColor;
- varying out vec4 gl_BackSecondaryColor;
- varying out vec4 gl_TexCoord[]; // at most gl_MaxTextureCoords
- varying out float gl_FogFragCoord;

Geometry shader inputs:

- varying in vec4 gl_FrontColorIn[gl_VerticesIn];
- varying in vec4 gl_BackColorIn[gl_VerticesIn];
- varying in vec4 gl_FrontSecondaryColorIn[gl_VerticesIn];
- varying in vec4 gl_BackSecondaryColorIn[gl_VerticesIn];
- varying in vec4 gl_TexCoordIn[gl_VerticesIn][]; // at most will be// gl_MaxTextureCoords
- varying in float gl_FogFragCoordIn[gl_VerticesIn];
- varying in vec4 gl_PositionIn[gl_VerticesIn];
- varying in float gl_PointSizeIn[gl_VerticesIn];
- varying in vec4 gl_ClipVertexIn[gl_VerticesIn];

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(colorAttr, color);
/* color is variable in application */
```

Used in glBegin() glEnd() like glNormal3f()

Or use glVertexAttribPointer(). This way you can store (besides position, normal, color and texture coord) additional values for every vertex.

Used with glDrawArrays()

Varying 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;)
}
```

Varying Example: Fragment Shader

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

[1]instead of gl_FragColor = gl_FrontColor;

Vertex Shader Applications

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

Toon Shader Example

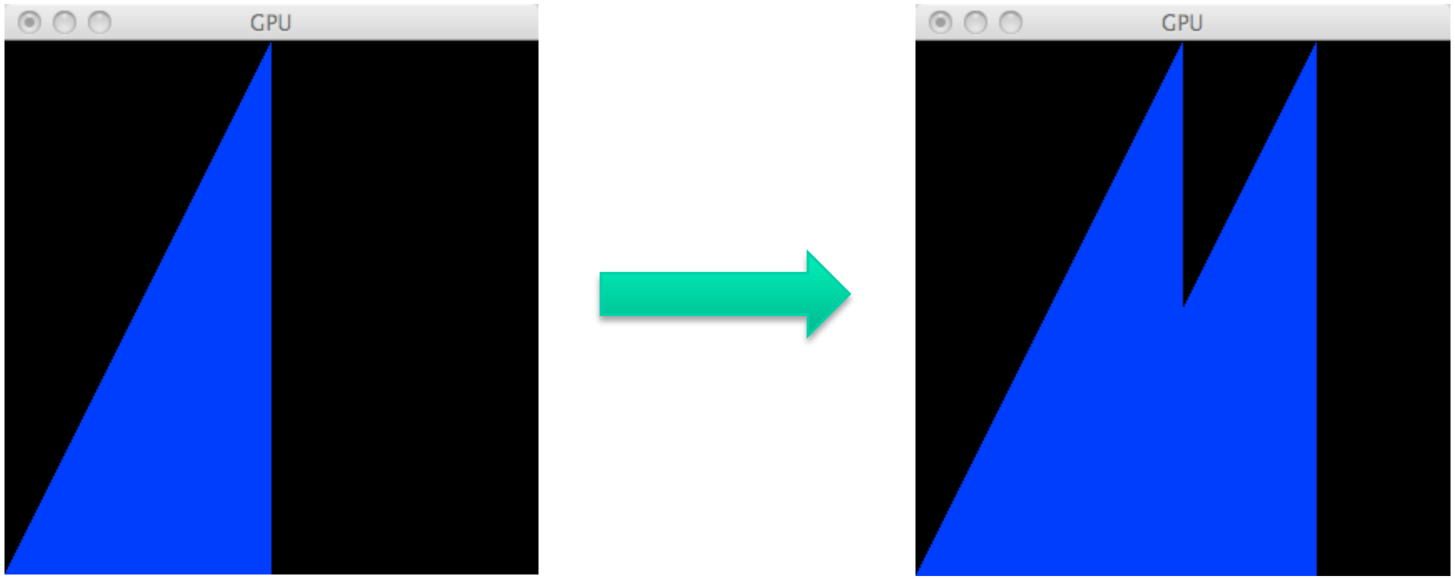
```
/* File main.cpp
  Simple Demo for GLSL
  www.lighthouse3d.com
*/

#include <GL/glew.h>
#include <GL/glut.h>
#include <stdio.h>
#include <stdlib.h>
#include "textfile.h"

GLhandleARB v,f,p;
float lpos[4] = {1,0.5,1,0};

void changeSize(int w, int h) {
	float ratio = 1.0* w / h;

	// Reset the coordinate system before modifying
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	// Set the correct perspective.
	gluPerspective(45,ratio,1,1000);

	// Set the viewport to be the entire window
	glViewport(0, 0, w, h);

	glMatrixMode(GL_MODELVIEW);
}

void renderScene(void) {

	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

	glLoadIdentity();
	gluLookAt(0.0,0.0,5.0,
		      0.0,0.0,-1.0,
			  0.0f,1.0f,0.0f);

	glLightfv(GL_LIGHT0, GL_POSITION, lpos);
	glutSolidTeapot(1);

	glutSwapBuffers();
}

void processNormalKeys(unsigned char key, int x, int y) {

	if (key == 27)
		exit(0);
}

void setShaders() {

	char *vs = NULL,*fs = NULL,*fs2 = NULL;

	v = glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
	f = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);

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

	const char * ff = fs;
	const char * vv = vs;

	glShaderSourceARB(v, 1, &vv,NULL);
	glShaderSourceARB(f, 1, &ff,NULL);

	free(vs);free(fs);

	glCompileShaderARB(v);
	glCompileShaderARB(f);

	p = glCreateProgramObjectARB();
	glAttachObjectARB(p,f);
	glAttachObjectARB(p,v);

	glLinkProgramARB(p);
	glUseProgramObjectARB(p);
}

int main(int argc, char **argv) {
	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
	glutInitWindowPosition(100,100);
	glutInitWindowSize(320,320);
	glutCreateWindow("MM 2004-05");

	glutDisplayFunc(renderScene);
	glutIdleFunc(renderScene);
	glutReshapeFunc(changeSize);
	glutKeyboardFunc(processNormalKeys);

	glEnable(GL_DEPTH_TEST);
	glClearColor(1.0,1.0,1.0,1.0);

	glewInit();
	if (GLEW_ARB_vertex_shader && GLEW_ARB_fragment_shader)
		printf("Ready for GLSL\n");
	else {
		printf("No GLSL support\n");
		exit(1);
	}

	setShaders();
	glutMainLoop();
	return 0;
}
```

```
FILE toon.vert
// simple toon vertex shader
// www.lighthouse3d.com


varying vec3 normal, lightDir;		// Interpolated variables to the fragment shader

void main()
{
	lightDir = normalize(vec3(gl_LightSource[0].position));
	normal = normalize(gl_NormalMatrix * gl_Normal);

	gl_Position = ftransform(); // will transform vertex exactly similar as the fixed pipeline
}
```

```
FILE toon.frag
// simple toon fragment shader
// www.lighthouse3d.com

varying vec3 normal, lightDir;		// Interpolated variables from the vertex shader

void main()
{
	float intensity;
	vec3 n;
	vec4 color;

	n = normalize(normal);
	intensity = max(dot(lightDir,n),0.0);

	if (intensity > 0.98)
		color = vec4(0.8,0.8,0.8,1.0);
	else if (intensity > 0.5)
		color = vec4(0.4,0.4,0.8,1.0);
	else if (intensity > 0.25)
		color = vec4(0.2,0.2,0.4,1.0);
	else
		color = vec4(0.1,0.1,0.1,1.0);

	gl_FragColor = color;
}
```

```
// textfile.cpp
//
// simple reading and writing for text files
//
// www.lighthouse3d.com
//
// You may use these functions freely.
// they are provided as is, and no warranties, either implicit,
// or explicit are given
//////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <string.h>


char *textFileRead(char *fn) {


	FILE *fp;
	char *content = NULL;

	int count=0;

	if (fn != NULL) {
		fp = fopen(fn,"rt");

		if (fp != NULL) {
		  fseek(fp, 0, SEEK_END);
		  count = ftell(fp);
		  rewind(fp);

		  if (count > 0) {
			content = (char *)malloc(sizeof(char) * (count+1));
			count = fread(content,sizeof(char),count,fp);
			content[count] = '\0';
		  }
		  fclose(fp);
		}
	}
	return content;
}

int textFileWrite(char *fn, char *s) {

	FILE *fp;
	int status = 0;

	if (fn != NULL) {
		fp = fopen(fn,"w");

		if (fp != NULL) {

			if (fwrite(s,sizeof(char),strlen(s),fp) == strlen(s))
				status = 1;
			fclose(fp);
		}
	}
	return(status);
}
```

Simple Geometry shader demo



Geometry shader

```
#version 120
#extension GL_EXT_geometry_shader4 : enable
void main(void){
    //Pass-thru vertices!
    for(i=0; i< gl_VerticesIn; i++){
        gl_Position = gl_PositionIn[i];
        EmitVertex();
    }
    EndPrimitive();

    //New piece of geometry!  Add translation
    for(i=0; i< gl_VerticesIn; i++){
        gl_Position = gl_PositionIn[i];
        gl_Position.xy += vec2(0.5,0);
        EmitVertex();
    }
    EndPrimitive();
}
```

Loading the shaders

```
void setShaders() {
    GLuint v = glCreateShader(GL_VERTEX_SHADER);
    GLuint f = glCreateShader(GL_FRAGMENT_SHADER);
    GLuint f = glCreateShader(GL_GEOMETRY_SHADER_EXT);

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

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

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

    glProgramParameteriEXT(p,GL_GEOMETRY_INPUT_TYPE_EXT,GL_TRIANGLES);
    glProgramParameteriEXT(p,GL_GEOMETRY_OUTPUT_TYPE_EXT,GL_TRIANGLES);
    GLint temp;
    glGetIntegerv(GL_MAX_GEOMETRY_OUTPUT_VERTICES_EXT,&temp);
    glProgramParameteriEXT(p,GL_GEOMETRY_VERTICES_OUT_EXT,temp);

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

VERY IMPORTANT

ALL THE FOLLOWING SLIDES ARE FOR YOUR CONVENIENCE ONLY

AND IS OPTIONAL

BONUS MATERIAL

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

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 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};

loc1 = glGetUniformLocationARB(p,"specIntensity");
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);
```

Get → loc1 = glGetUniformLocationARB(p,"specIntensity");

Set → glUniform1fARB(loc1,specIntensity);

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: = *= /= += -=

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( progra
  mObject,"myFirstSampler"),0);
glUniform1iARB(glGetUniformLocationARB( progra
  mObject,"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);
    GLuint f = glCreateShader(GL_GEOMETRY_SHADER_EXT);

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

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

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

    glProgramParameteriEXT(p,GL_GEOMETRY_INPUT_TYPE_EXT,GL_TRIANGLES);
    glProgramParameteriEXT(p,GL_GEOMETRY_OUTPUT_TYPE_EXT,GL_TRIANGLES);
    GLint temp;
    glGetIntegerv(GL_MAX_GEOMETRY_OUTPUT_VERTICES_EXT,&temp);
    glProgramParameteriEXT(p,GL_GEOMETRY_VERTICES_OUT_EXT,temp);

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

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

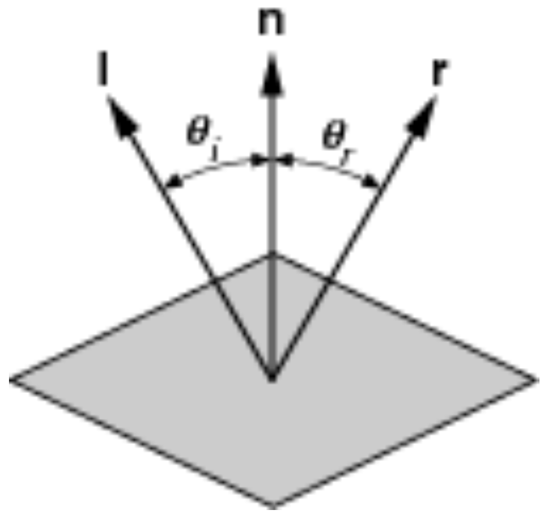
r, **n**, and **l** are coplanar

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

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

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

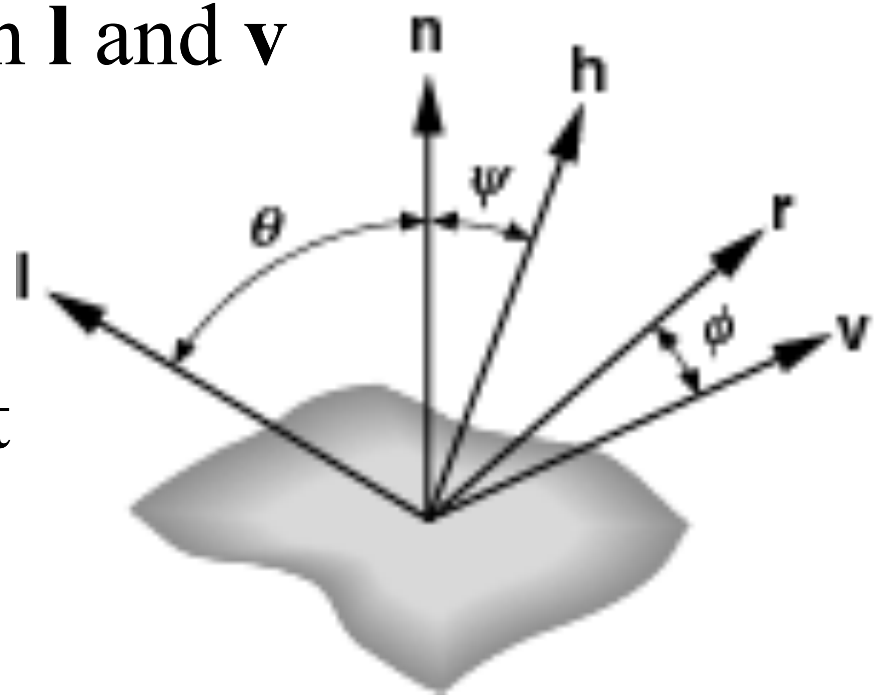
$(\mathbf{l}+\mathbf{v})/2$ is halfway between **l** and **v**

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

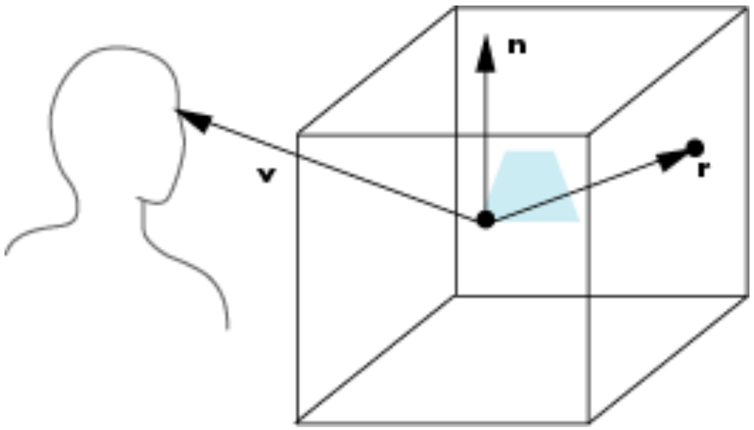
- We can form a cube map texture by defining six 2D texture maps that correspond to the sides of a box
- Supported by OpenGL
- Also supported in GLSL through cubemap sampler

```
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

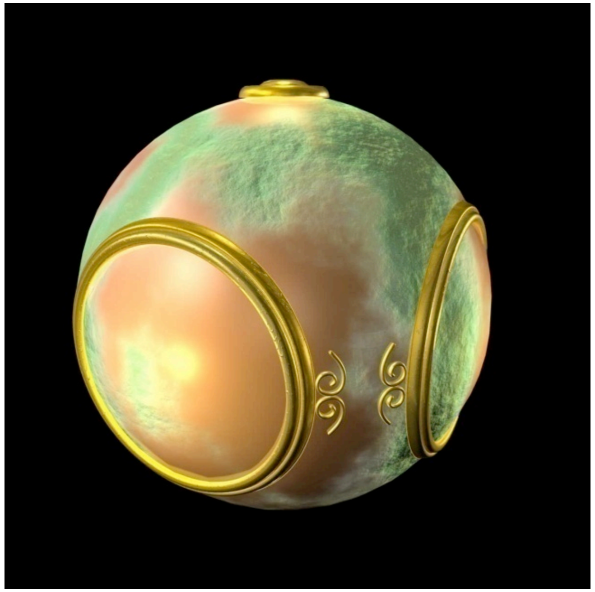
```
uniform mat4 modelMat;
uniform mat3 invModelMat;
varying vec4 reflectw;
void main(void)
{
  vec4 positionw = modelMat*gl_Vertex;
  vec3 normw = normalize(gl_Normal*invModelMat.xyz);
  vec3 lightw = normalize(eyew.xyz-positionw.xyz);
  reflectw = reflect(normw, eyew);
  gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;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

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