UNIVERSITY of HOUSTON

DEPARTMENT OF COMPUTER SCIENCE

COSC 4370 Fall 2023

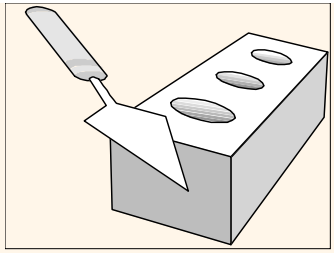
Interactive Computer Graphics

M & W 5:30 to 7:00 PM

Prof. Victoria Hilford

PLEASE TURN your webcam ON

NO CHATTING during LECTURE



COSC 4370

5:30 to 7

PLEASE

LOG IN

CANVAS

Please close all other windows.

NEXT.

10.23.2023 (M 5:30 to 7) (18)	Homework 7	**Lecture 9** **(Programmable Shaders)**
10.25.2023 (W 5:30 to 7) (19)		**Lecture 10** **(Modeling and Hierarchy)**
10.30.2023 (M 5:30 to 7) (20)		**PROJECT 3**
11.01.2023 (W 5:30 to 7) (21)		**EXAM 3 REVIEW**
11.06.2023 (M 5:30 to 7) (22)		**EXAM 3**

COSC 4370 – Computer Graphics

Lecture 9

Programmable Shaders

Chapter 9

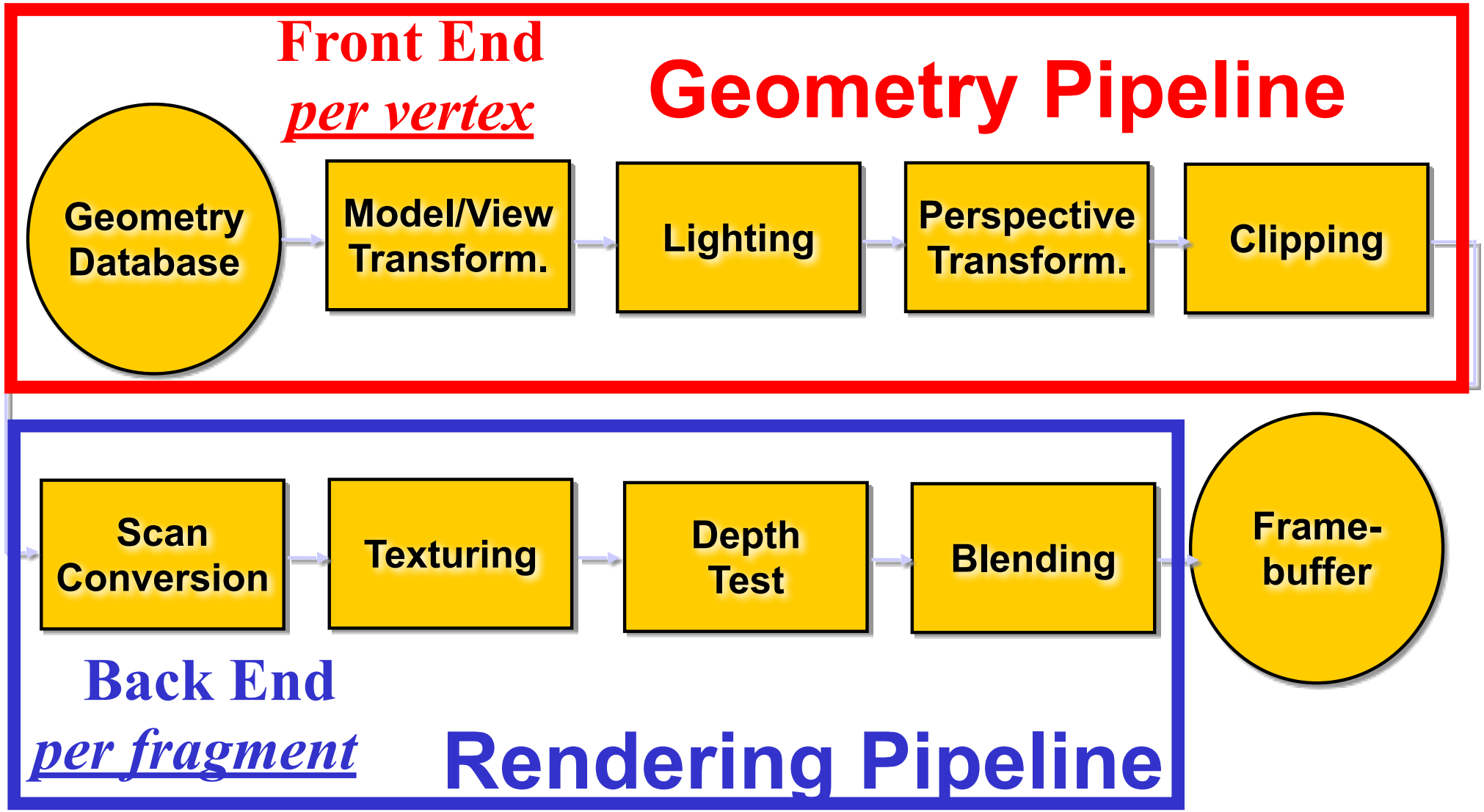
Programmable Pipelines

In developing the graphics pipeline, we assumed that each box in the pipeline has a fixed functionality. Consequently, when we wanted to use light material interactions to determine the colors of an object, we were limited to the modified Phong model because it was the only lighting model supported by the fixed-function pipeline in the OpenGL specification and, until recently, the only model supported by most hardware. In addition to having only one lighting model available, **lighting calculations were done only for each vertex** fixed-function vertex processor. The resulting vertex colors were then interpolated over the primitive by the fixed-function fragment processor. **If we wanted to use some other lighting or shading model, we had to resort to an off-line renderer.**

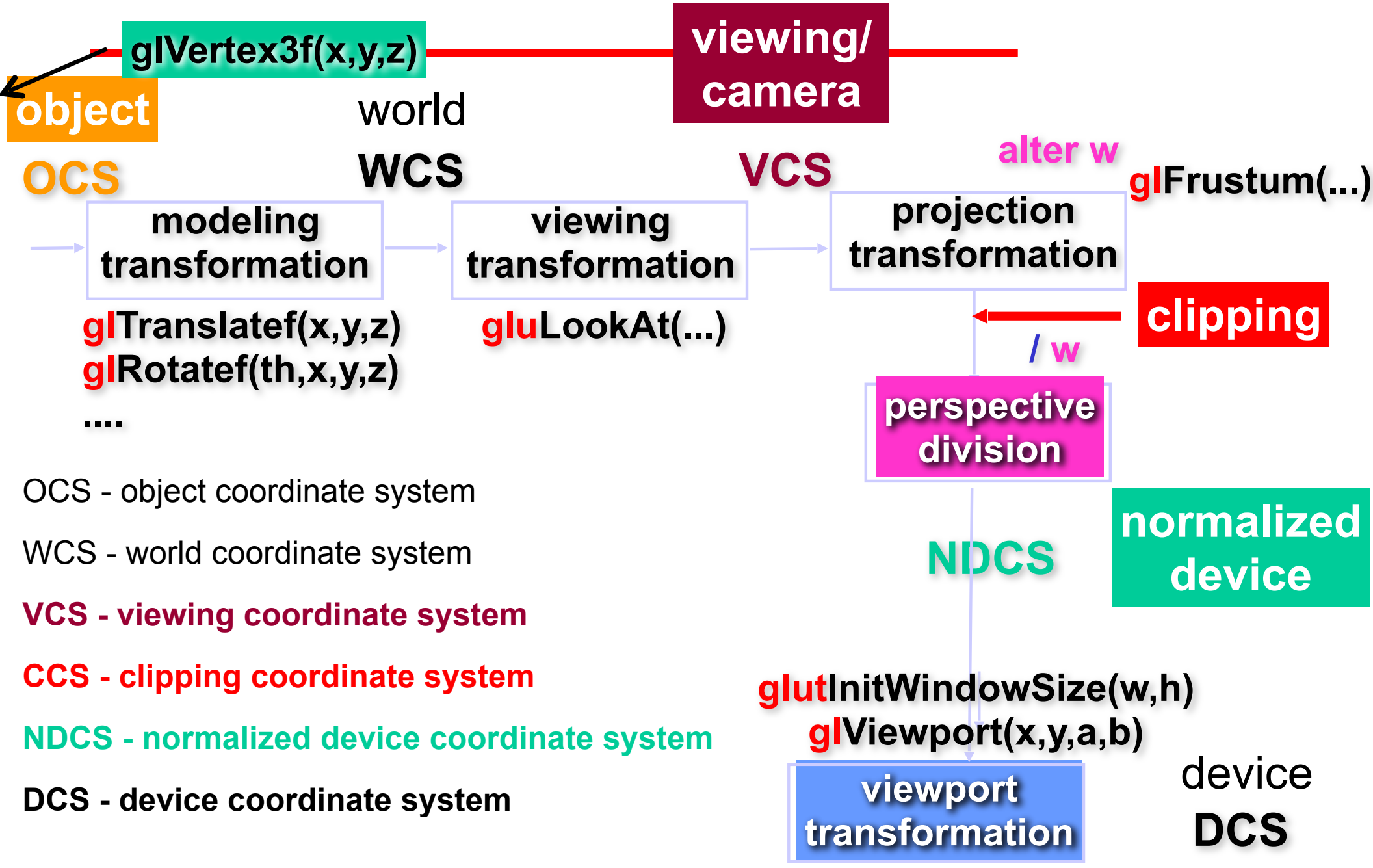
Over the past few years, graphics processors have changed dramatically. **Both the vertex processor** and **fragment processor are now user programmable.** We can write programs called **vertex shaders** and **fragment shaders** to achieve complex visual effects at the **same rate** as the **standard fixed-function pipeline**.

In this chapter, we introduce the concept of programmable shaders. First, we review some of the efforts to develop languages to describe shaders. These efforts culminated in the **OpenGL Shading Language (GLSL),** which is now a **standard part of OpenGL**. We then use GLSL to develop a variety of **vertex shaders** that compute vertex properties, including their positions and colors. Finally, we develop **fragment shaders** that let us program the calculations performed on each fragment and ultimately determine the color of each pixel. Our discussion of **fragment shaders** will also introduce many new ways of using **texture mapping**

The Graphics Pipeline



Projective Rendering Pipeline



OCS - object coordinate system

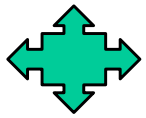
WCS - world coordinate system

VCS - viewing coordinate system

CCS - clipping coordinate system

NDCS - normalized device coordinate system

DCS - device coordinate system



Texture Coordinates

- Assumed rectangular polygons then we can generate texture coordinates

- Textures coordinates form a 4X4 texture matrix. Can be manipulated in the same manner as the model-view and projection matrices.

- We use it by setting the current matrix mode as follows:

```
glMatrixMode(GL_TEXTURE);
```

- We can use this matrix to scale and orient textures and to cre effects in which the texture moves with the object, the camera or the lights.

Geometric Calculations

- **Geometric data**: set of **vertices** + **type**

  Can come from program, evaluator, display list

  type: **point**, **line**, **polygon**

  Vertex data can be

  - **Position** (x,y,z,w) coordinates of a **vertex** (**gl**Vertex)
  - **Normal** vector (**gl**Normal)
  - **Texture Coordinates** (**gl**TexCoord)
  - **RGBA color** (**gl**Color)
  - **Other data: color indices, edge flags**
  - **Additional user-defined data in GLSL**

gl_

Model/View Transform.

Per-Vertex Operations

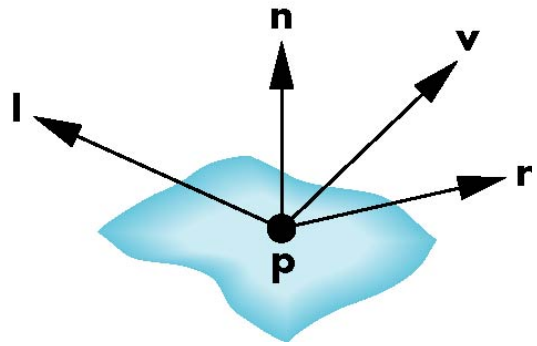
- **Vertex** locations are transformed by the **model-view matrix** into **eye coordinates**

- **Normals** must be transformed with the inverse transpose of the **model-view matrix** so that

  v ● **n** = **v'** ● **n'** in both spaces

- **Textures coordinates** are generated if autotexture enabled and the **texture matrix** is applied

Lighting Calculations



- Consider a **per-vertex** basis **Phong model**

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

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

Calculating the Reflection Term **r**

angle of incidence = angle of reflection

$\cos \theta_i = \cos \theta_r$ or **r** ● **n** = **l** ● **n**

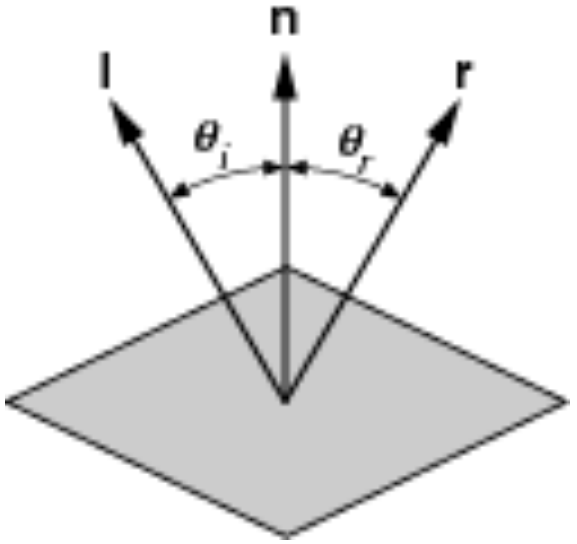
r, **n**, and **l** are coplanar

r = α**l** + β**n**

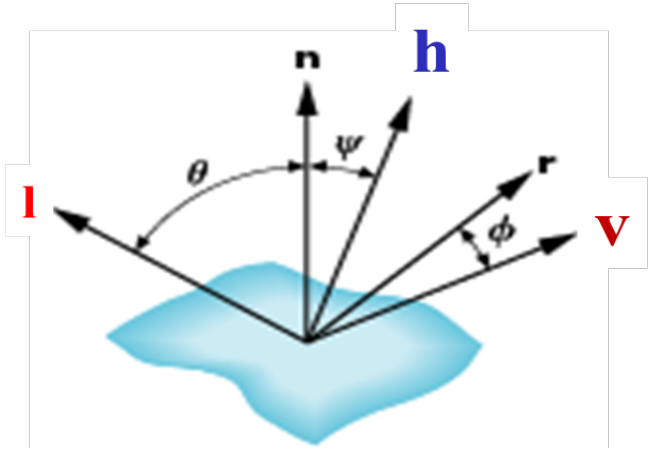
normalize

1 = **r** ● **r** = **n** ● **n** = **l** ● **l**

solving: **r** = 2(**l** ● **n**) **n** - **l**



OpenGL Lighting



- Modified Phong model **per-vertex**
  - Halfway vector **h**
  - Global ambient term
- Supported by hardware **per-vertex**

Halfway Vector h

Blinn proposed replacing **v** • **r** by **n** • **h** where

h = (**l** + **v**)/|**l** + **v**|

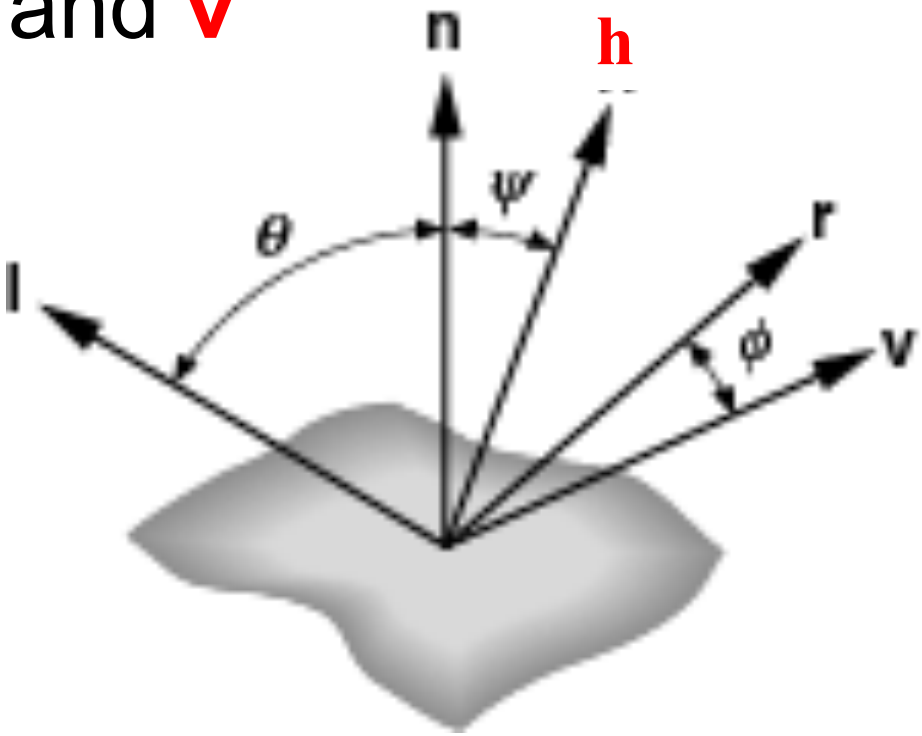
(**l**+**v**)/2 is halfway between **l** and **v**

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

$$\psi = \phi/2$$

Must then adjust exponent

so that (**n** • **h**)$^{e'}$ ≈ (**r** • **v**)^{e}



Primitive Assembly

- **Vertices** are next assembled into **objects**
  - **Polygons**
  - **Line Segments**
  - **Points**
- Transformation by projection matrix

Perspective Transform.

- Clipping
  - Against user defined planes
  - View volume, x=±w, y=±w, z=±w
  - Polygon clipping can create new **vertices**

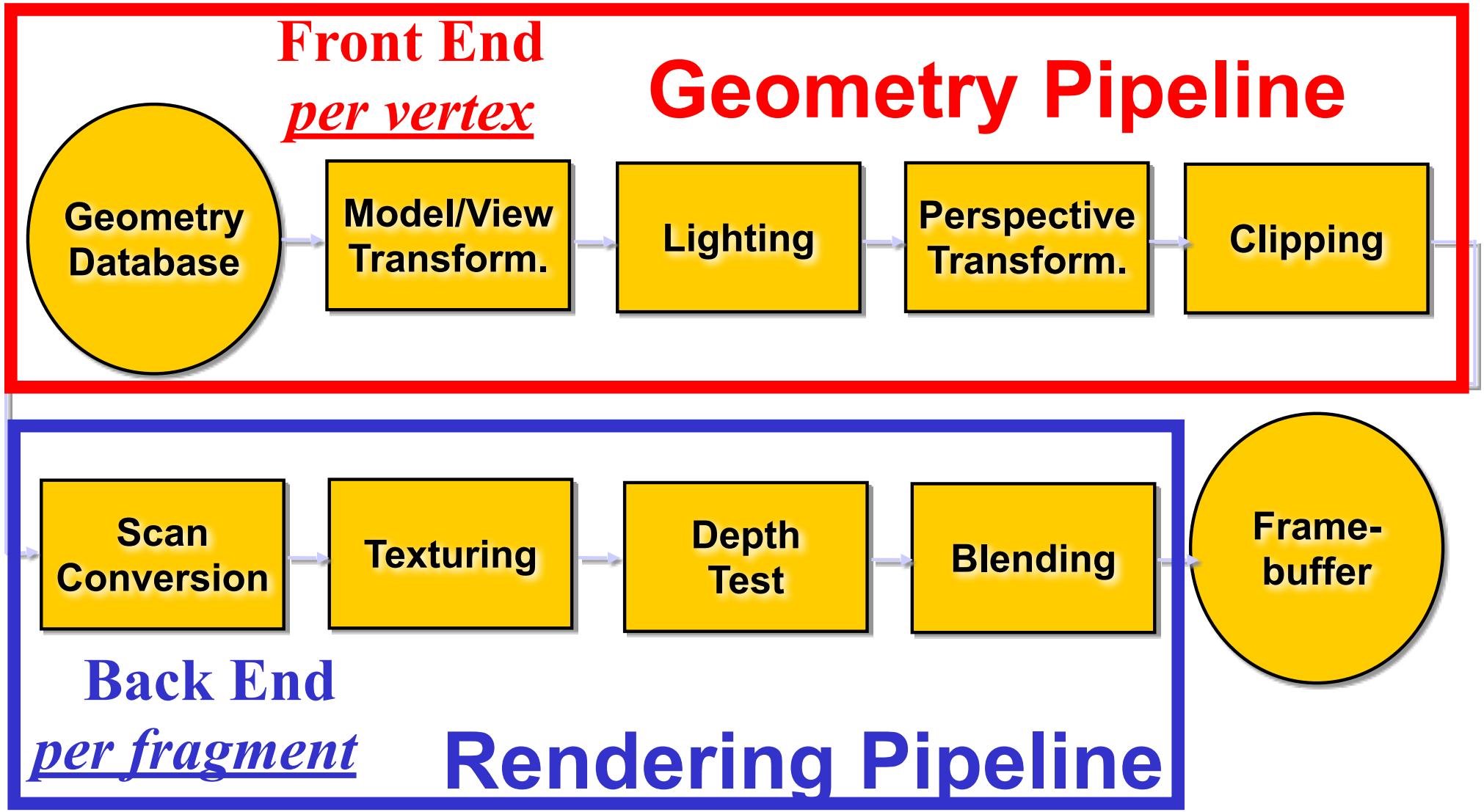
Clipping

- Perspective Division
- Viewport mapping

/ w

perspective division

The Graphics Pipeline



Scan Conversion

Rasterization

- **Geometric entities** are rasterized into **fragments**
- **Each fragment** corresponds to a point on an integer grid: a **displayed pixel**

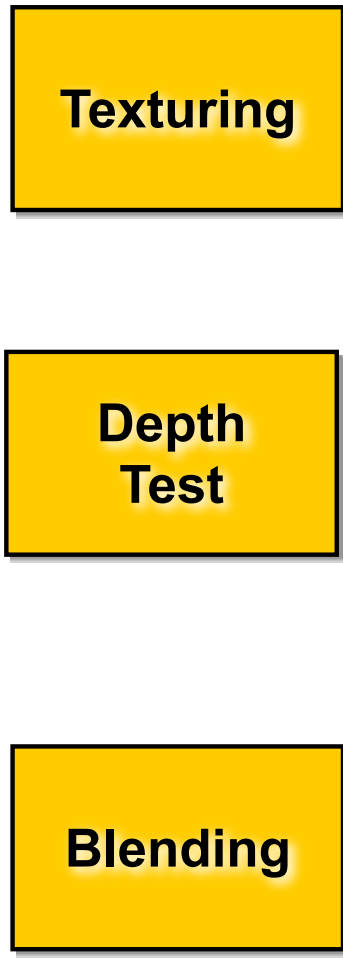
- Hence each **fragment** is a ***potential pixel***

- Each **fragment** has
  - A **color**
  - Possibly a **depth value**
  - **Texture coordinates**

Fragment Operations

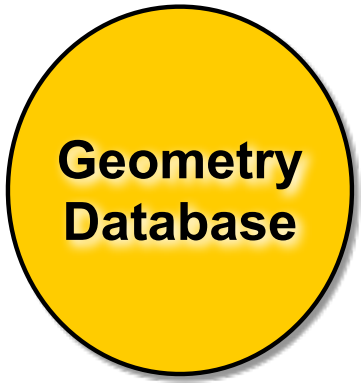
- Texture generation
- Fog
- Antialiasing
- Alpha test

- Blending
- Logical Operation
- Masking



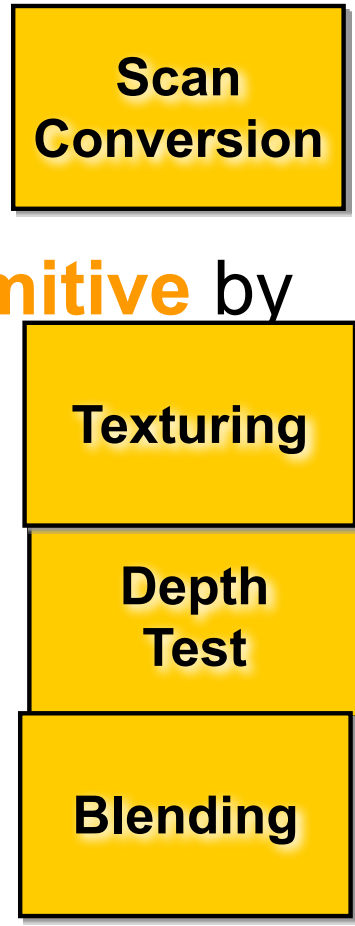
Vertex Processor
Vertex IN Vertex OUT

- Takes in vertices
  - Position attribute
  - Possibly color
  - ……
  - OpenGL state



- Produces
  - Vertex Position in clip coordinates
  - Vertex color
  - ……

Fragment Processor
Fragments IN Fragment OUT



- Takes in output of rasterizer (**fragments**)

  Vertex values have been interpolated **over primitive** by rasterizer

- Outputs a **fragment**

  Color

  Texture

- **Fragments** still go through **fragment tests**

  Hidden-surface removal

  Alpha

SOFTWARE
vs.
HARDWARE

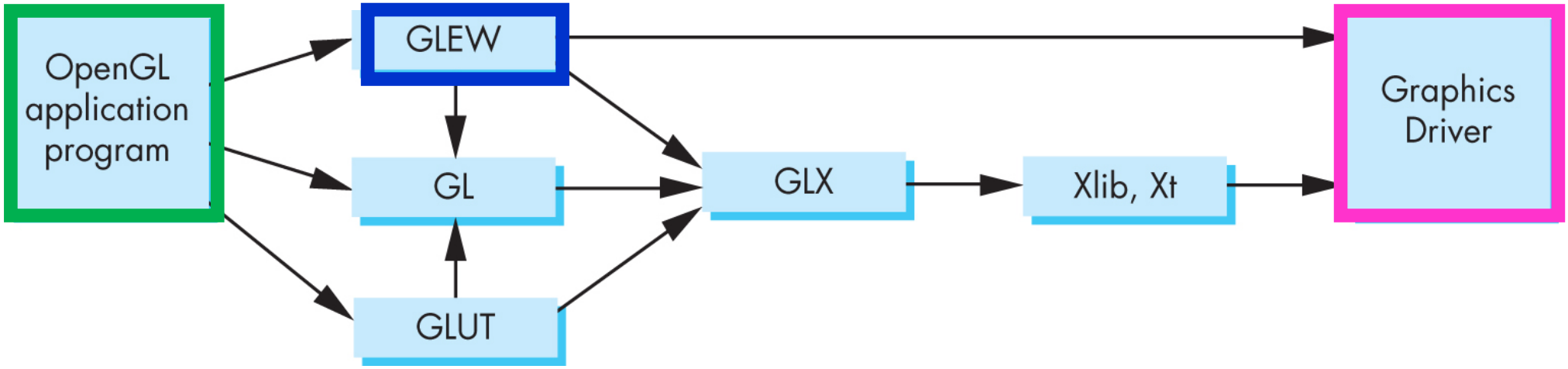
- What does it mean ???????

Instead of using the HARDWARE implementation I am going to use a SOFTWARE implementation (not in Assembly Language but in a High-Level Language – C “like”)

GLEW

- Open**GL E**xtension **W**rangler Library
- Makes it easy to access **OpenGL extensions** available on a particular system
- Avoids having to have specific entry points in Windows code
- Application needs only to include **glew.h** and run a **glewInit()**

Software Organization



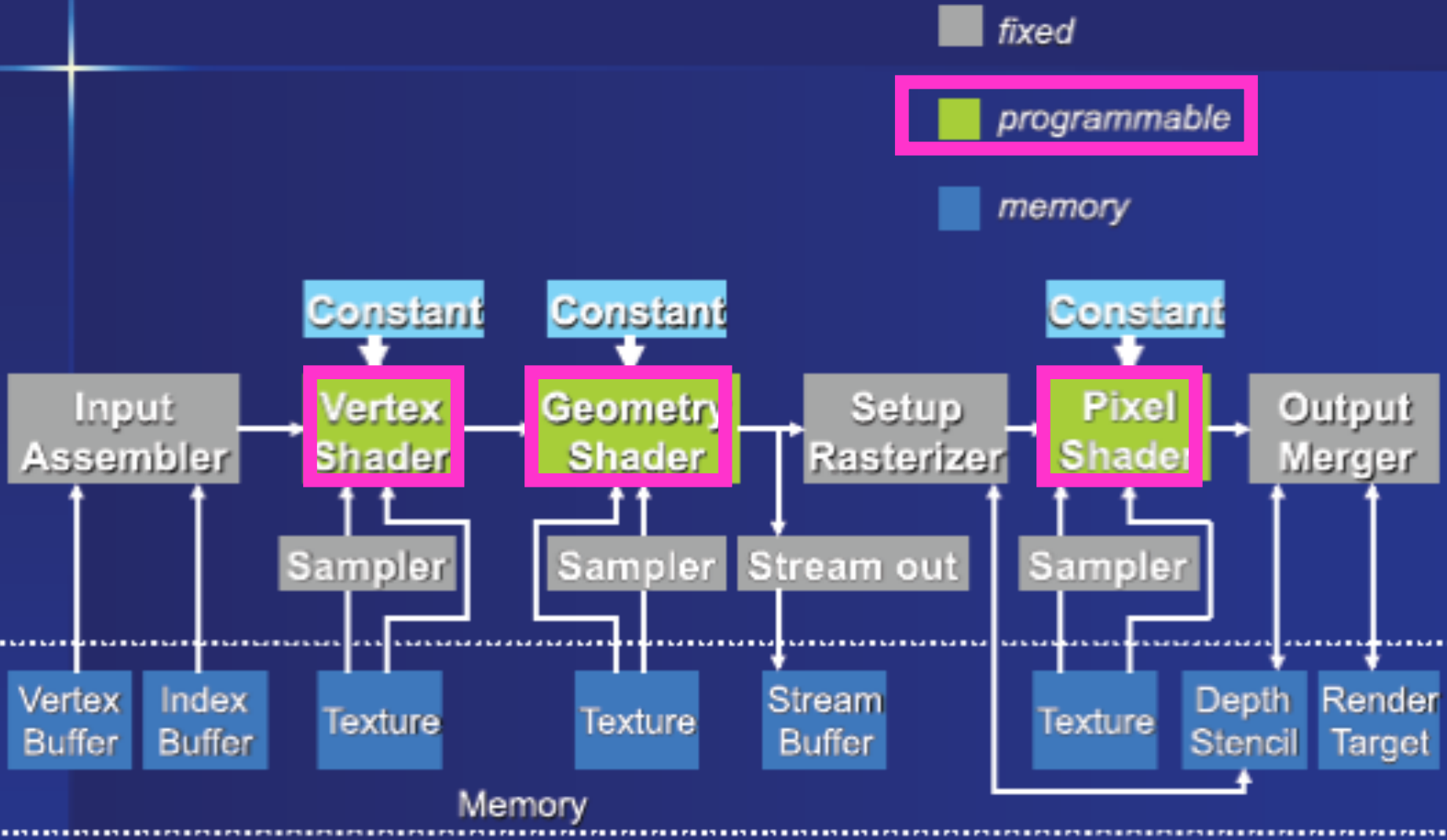
Programmable Shaders

- Replace **fixed-function vertex** and **fragment** processing by **programmable processors called shaders**

- Can replace **either** or **both**

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

The New Pipeline



Moving Vertices at the Application Level

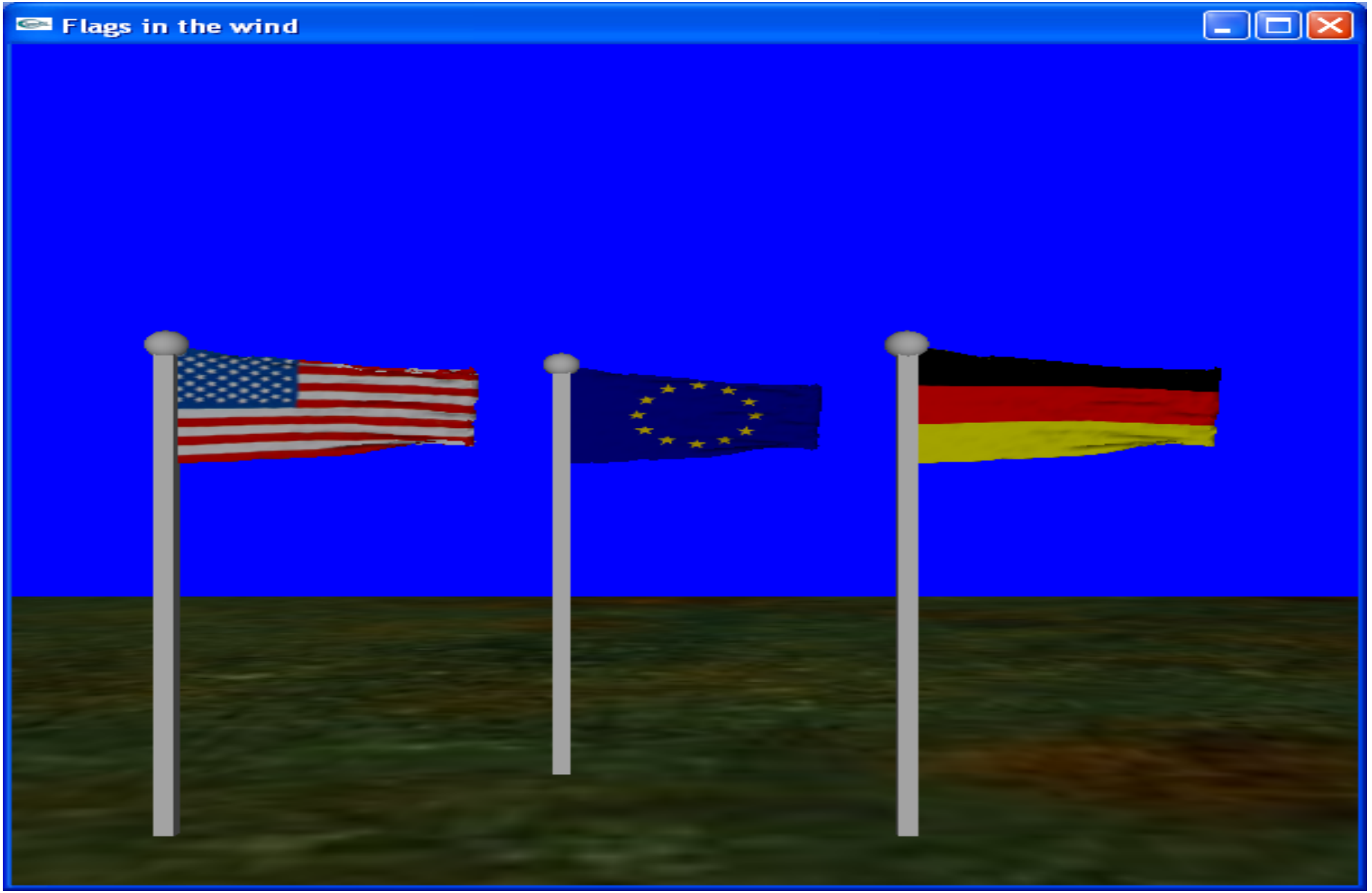
- Moving vertices

  Wave motion

Flags Simulation

```
140 void CFlag::UpdateVertices(float dtime, SF3dVector gravity, SF3dVector wind)
141 {
142     SF3dVector * a = new SF3dVector [m_DetailX * m_DetailY];  //acceleration array
143     for (int xc = 0; xc < m_DetailX; xc ++)
144         for (int yc = 0; yc < m_DetailY; yc++)
145            if (xc > 0)
146            {
147                 int ArrayPos = yc * m_DetailX + xc;
148                 SF3dVector VertexPos = m_Vertices[ArrayPos].pos;
149
150                 a[ArrayPos] = gravity;
151
152                 a[ArrayPos] =a[ArrayPos]+GetForce(xc-1,yc,   VertexPos,
    m_DirectDistance);
153                 a[ArrayPos] =a[ArrayPos]+GetForce(xc+1,yc,   VertexPos,
    m_DirectDistance);
154                 a[ArrayPos] =a[ArrayPos]+GetForce(xc,  yc-1, VertexPos,
    m_DirectDistance);
155                 a[ArrayPos] =a[ArrayPos]+GetForce(xc,  yc+1, VertexPos,
    m_DirectDistance);
156
157                 a[ArrayPos] =a[ArrayPos]+GetForce(xc-1,yc-1,VertexPos,
    m_AslantDistance);
158                 a[ArrayPos] =a[ArrayPos]+GetForce(xc-1,yc+1,VertexPos,
    m_AslantDistance);
159                 a[ArrayPos] =a[ArrayPos]+GetForce(xc+1,yc-1,VertexPos,
    m_AslantDistance);
160                 a[ArrayPos] =a[ArrayPos]+GetForce(xc+1,yc+1,VertexPos,
    m_AslantDistance);
161
162                 //wind:
163                 float DotProduct = Normalize3dVector(wind) * m_Vertices[ArrayPos].
    normal;
164                 if(DotProduct < 0.0f) DotProduct *= -1;
165                 a[ArrayPos]=a[ArrayPos] + wind * DotProduct;
166
167           } else a[yc * m_DetailX + xc] = NULL_VECTOR;
168
169     for (int xc = 0; xc < m_DetailX; xc ++)
170         for (int yc = 0; yc < m_DetailY; yc++)
171         {
172             int ArrayPos = yc * m_DetailX + xc;
173
174             m_Velocity[ArrayPos] = m_Velocity[ArrayPos]+a[ArrayPos]*dtime;
175
176             m_Velocity[ArrayPos] = m_Velocity[ArrayPos]*m_Damping;
177
178             m_Vertices[ArrayPos].pos = m_Vertices[ArrayPos].pos + m_Velocity
    [ArrayPos] * dtime;
179         }
180
181     delete[] a;
182
183     UpdateNormals();
184 }
```

OpenGL Flags Simulation



```
140 void CFlag::UpdateVertices(float dtime, SF3dVector gravity, SF3dVector wind)
141 {
142     SF3dVector * a = new SF3dVector [m_DetailX * m_DetailY];  //acceleration array
143     for (int xc = 0; xc < m_DetailX; xc ++)
144         for (int yc = 0; yc < m_DetailY; yc++)
145             if (xc > 0)
146             {
147                 int ArrayPos = yc * m_DetailX + xc;
148                 SF3dVector VertexPos = m_Vertices[ArrayPos].pos;
149
150                 a[ArrayPos] = gravity;
151
152                 a[ArrayPos] =a[ArrayPos]+GetForce(xc-1,yc,   VertexPos, m_DirectDistance);
153                 a[ArrayPos] =a[ArrayPos]+GetForce(xc+1,yc,   VertexPos, m_DirectDistance);
154                 a[ArrayPos] =a[ArrayPos]+GetForce(xc,  yc-1, VertexPos, m_DirectDistance);
155                 a[ArrayPos] =a[ArrayPos]+GetForce(xc,  yc+1, VertexPos, m_DirectDistance);
156
157                 a[ArrayPos] =a[ArrayPos]+GetForce(xc-1,yc-1,VertexPos, m_AslantDistance);
158                 a[ArrayPos] =a[ArrayPos]+GetForce(xc-1,yc+1,VertexPos, m_AslantDistance);
159                 a[ArrayPos] =a[ArrayPos]+GetForce(xc+1,yc-1,VertexPos, m_AslantDistance);
160                 a[ArrayPos] =a[ArrayPos]+GetForce(xc+1,yc+1,VertexPos, m_AslantDistance);
161
162                 //wind:
163                 float DotProduct = Normalize3dVector(wind) * m_Vertices[ArrayPos].normal;
164                 if(DotProduct < 0.0f) DotProduct *= -1;
165                 a[ArrayPos]=a[ArrayPos] + wind * DotProduct;
166
167             } else a[yc * m_DetailX + xc] = NULL_VECTOR;
168
```

Development

- **RenderMan Shading Language**
  - **Offline rendering**
- **Hardware Shading Languages**
  - UNC, Stanford
  - NVIDIA
  - **OpenGL Vertex Program Extension**
  - **OpenGL Shading Language GLSL**
  - Cg
    - OpenGL
    - Microsoft **HLSL**

RenderMan

Developed by Pixar

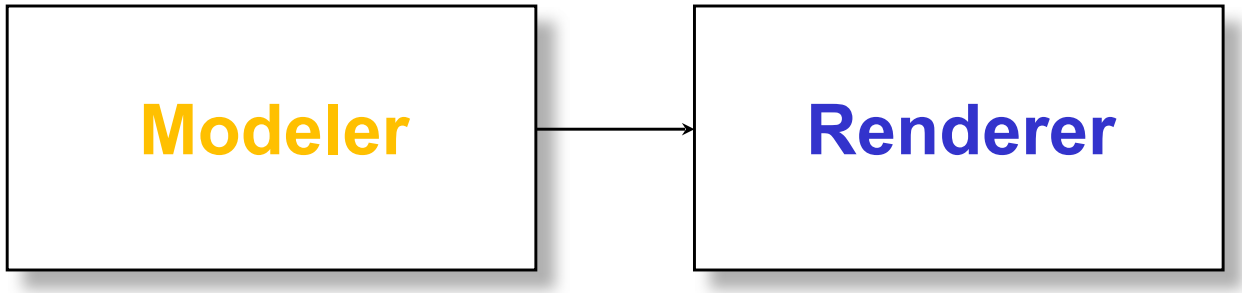
RenderMan—Hanrahan/Lawson

1. Abstract shading model based on optics for either global/local illumination
2. Define interface between rendering program and shading modules
3. High-level language

Three main kinds of shaders—light source, surface reflectance, volume

Modeling vs Rendering

interface file (RIB)



- Modeler outputs geometric model plus information for the renderer
  - Specifications of camera
  - Materials
  - Lights
- May have different kinds of renderers
  - Ray tracer
  - Radiosity
- How do we specify a shader?

Shading Trees

Shaders such as the **Phong model** can be written as algebraic expressions

$$I = k_d I_d \, \mathbf{l} \bullet \mathbf{n} + k_s I_s (\mathbf{v} \bullet \mathbf{r})^s + k_a I_a$$

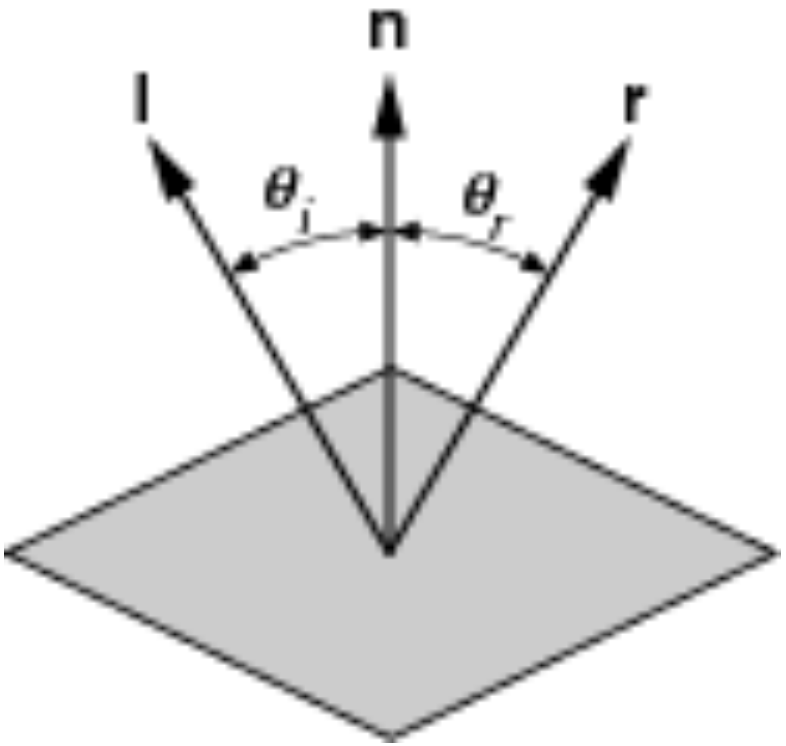
But expressions can be described by Trees

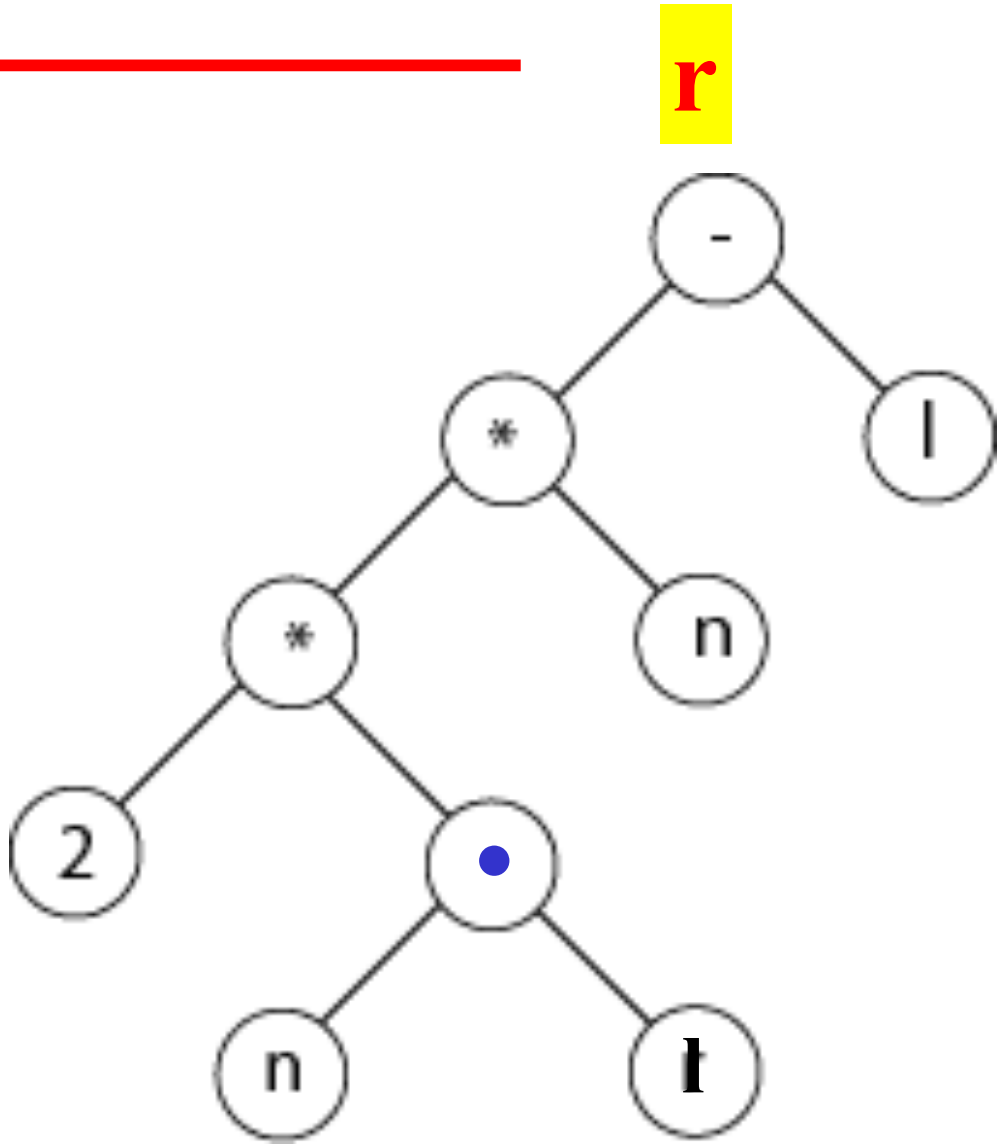
Need now operators such as **dot** and **cross products** and **new data types** such as **matrices** and **vectors**

Environmental variables are part of OpenGL State

Reflection Vector r

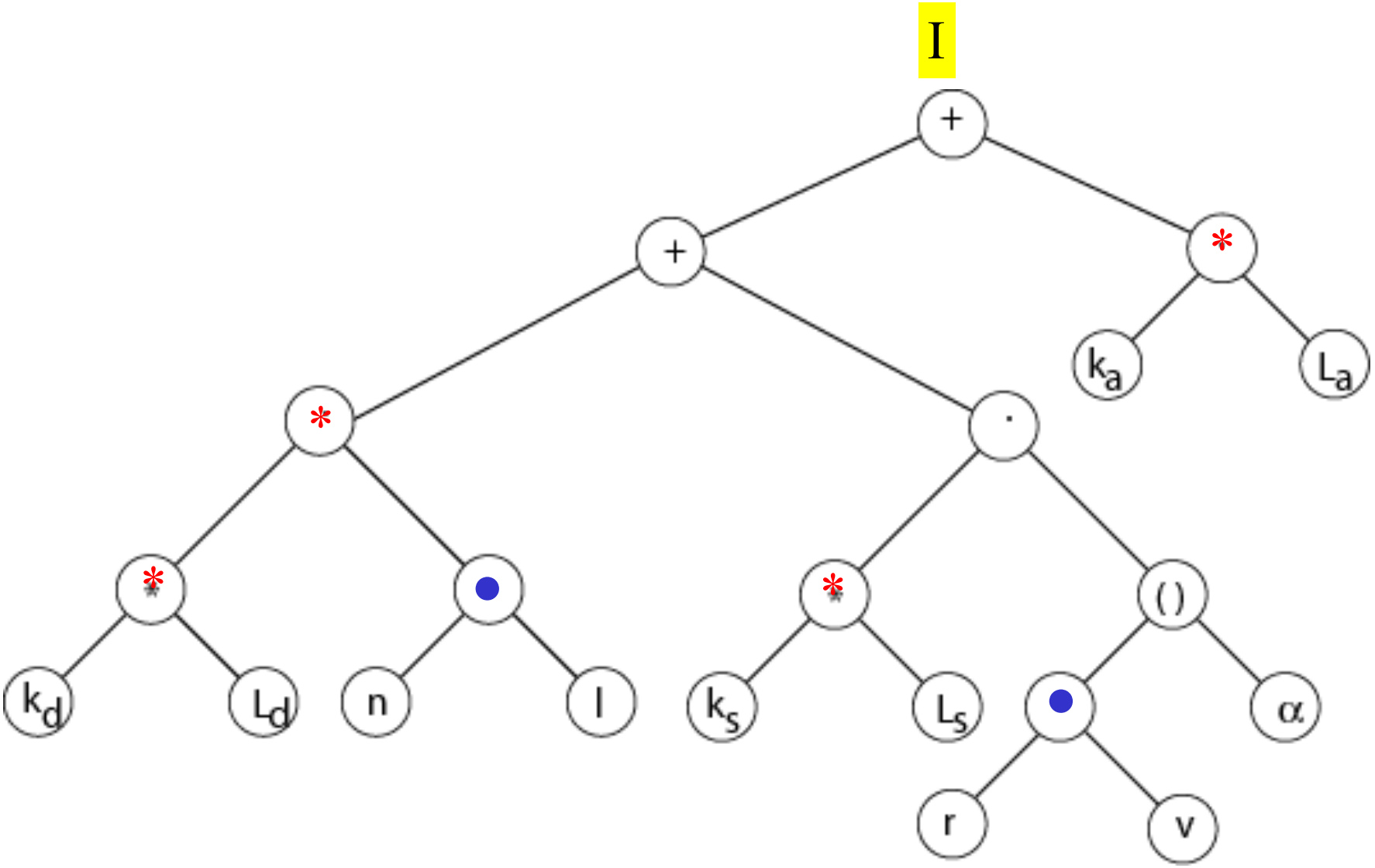
$\mathbf{r} = 2 * (\mathbf{l} \bullet \mathbf{n}) * \mathbf{n} - \mathbf{l}$





Phong Model

$$I = k_d * I_d * \mathbf{l} \bullet \mathbf{n} + k_s * I_s * (\mathbf{v} \bullet \mathbf{r})^s + k_a * I_a$$



GLSL I

- Open GL Shading Language GLSL

Chapter 9.3

Objectives
GLSL

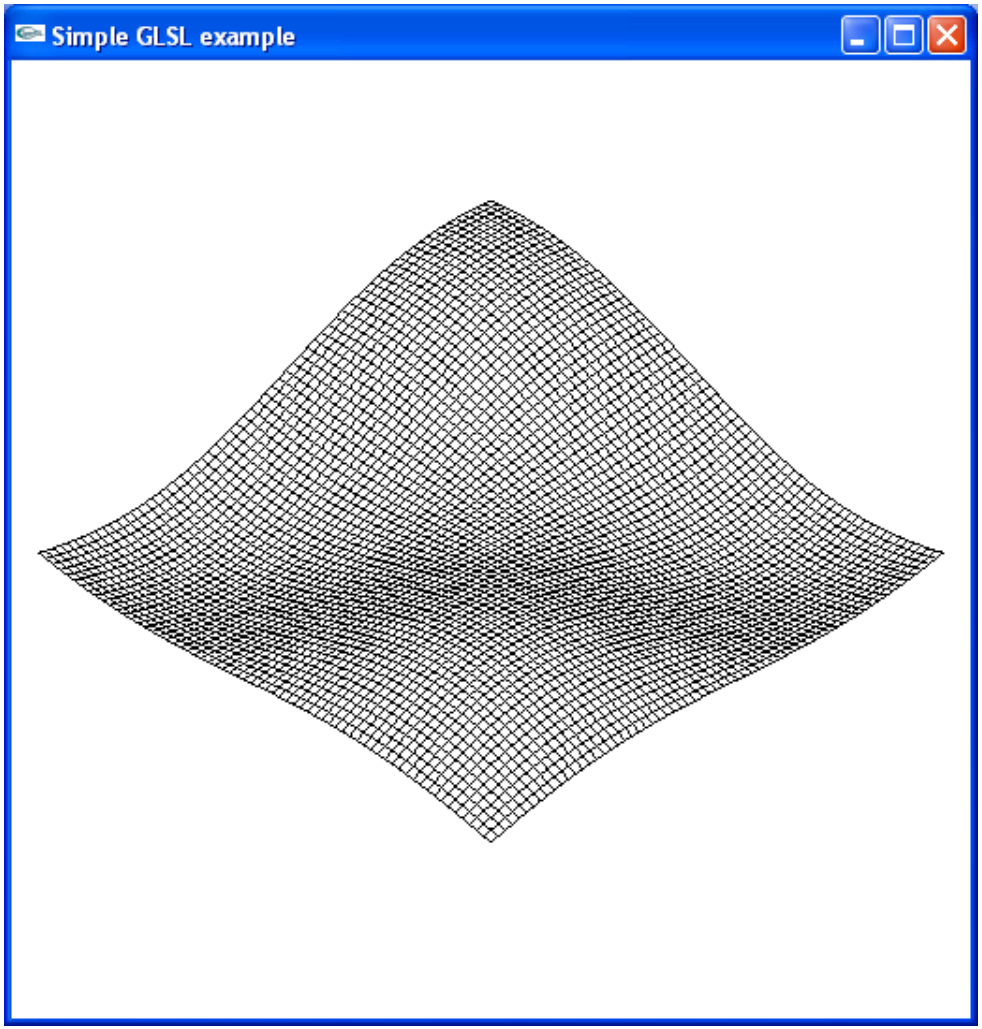
- Shader applications
  - **Vertex** shaders
  - **Fragment** shaders
- Programming shaders
  - Cg
  - **GLSL**

Shader Applications

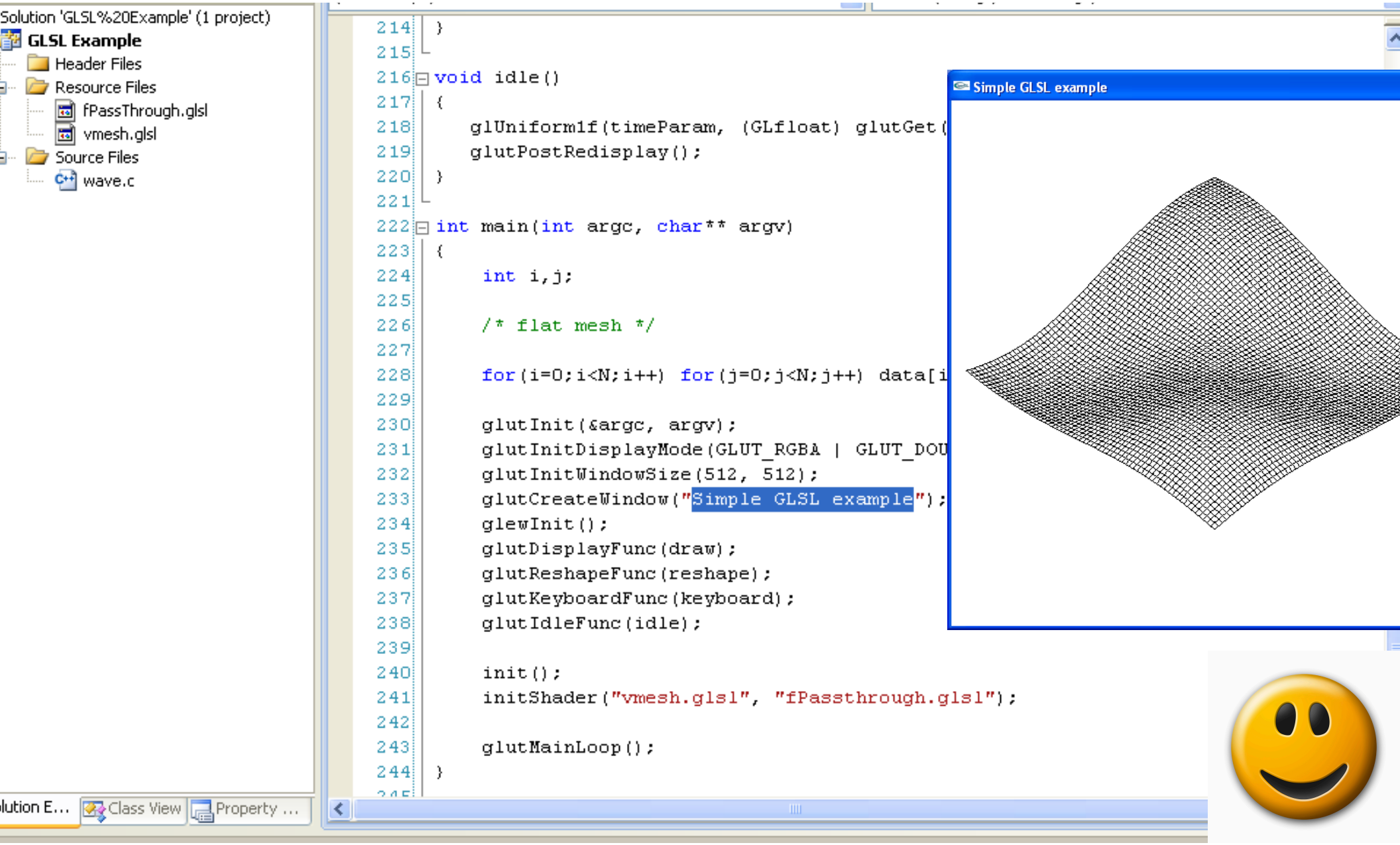
- Moving **vertices** (**Flag Sim** not a **shader**)
  - Morphing
  - **Wave motion**
  - Fractals
- **Lighting**
  - **More realistic models**
  - Cartoon shaders

Vertex Shader Applications

Wave motion



Wave Motion Vertex Shader



Wave Motion Vertex & Fragment Shaders

Visual Assert Test Window Help

Debug Win32 texenv

vmesh.glsl fPassThrough.glsl wave.c

```
1  uniform float time; /* in milliseconds */
2
3  void main()
4  {
       float s;
       vec4 t = gl_Vertex;
       t.y = 0.1*sin(0.001*time+5.0*gl_Vertex.x)*sin(0.001*time+5.0*gl_Vertex.z);
       gl_Position = gl_ModelViewProjectionMatrix * t;
       gl_FrontColor = gl_Color;
10 }
```

gl_

Server Explorer Toolbox

Tools Visual Assert Test Window Help

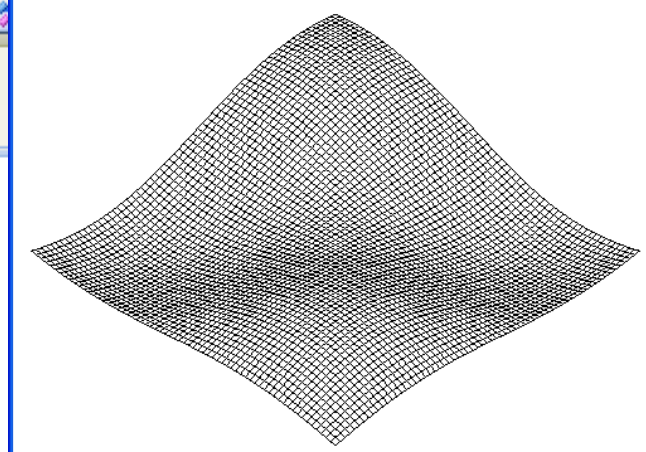
Debug Win32 texenv

vmesh.glsl fPassThrough.glsl wave.c

```
1  // fPassThrough.glsl
2  // Pass through fragment shader.
3
   void main()
   {
        gl_FragColor = gl_Color;
   }
9
10
11
```

gl_

Simple GLSL example



Fragment Shader Applications

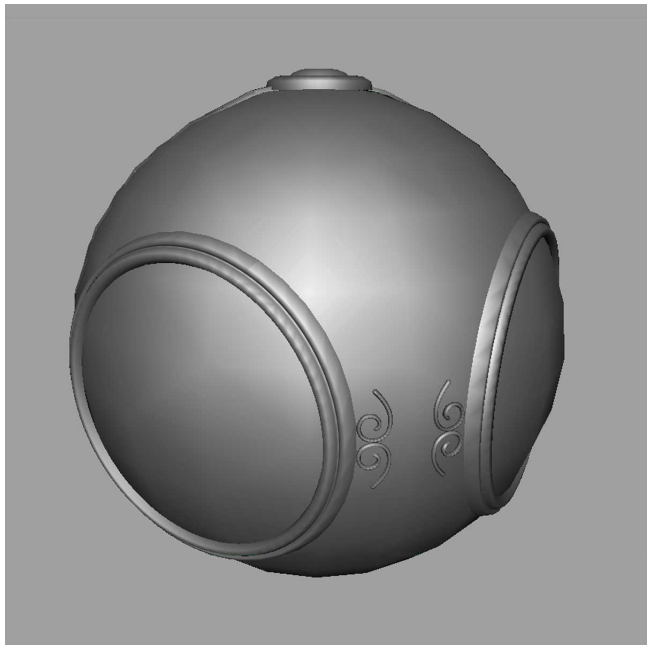
Per fragment lighting calculations

per vertex lighting

per fragment lighting

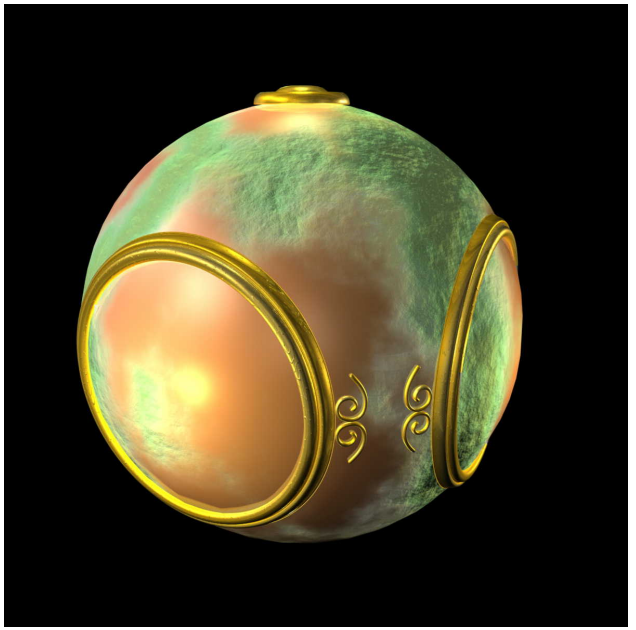
Fragment Shader Applications

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

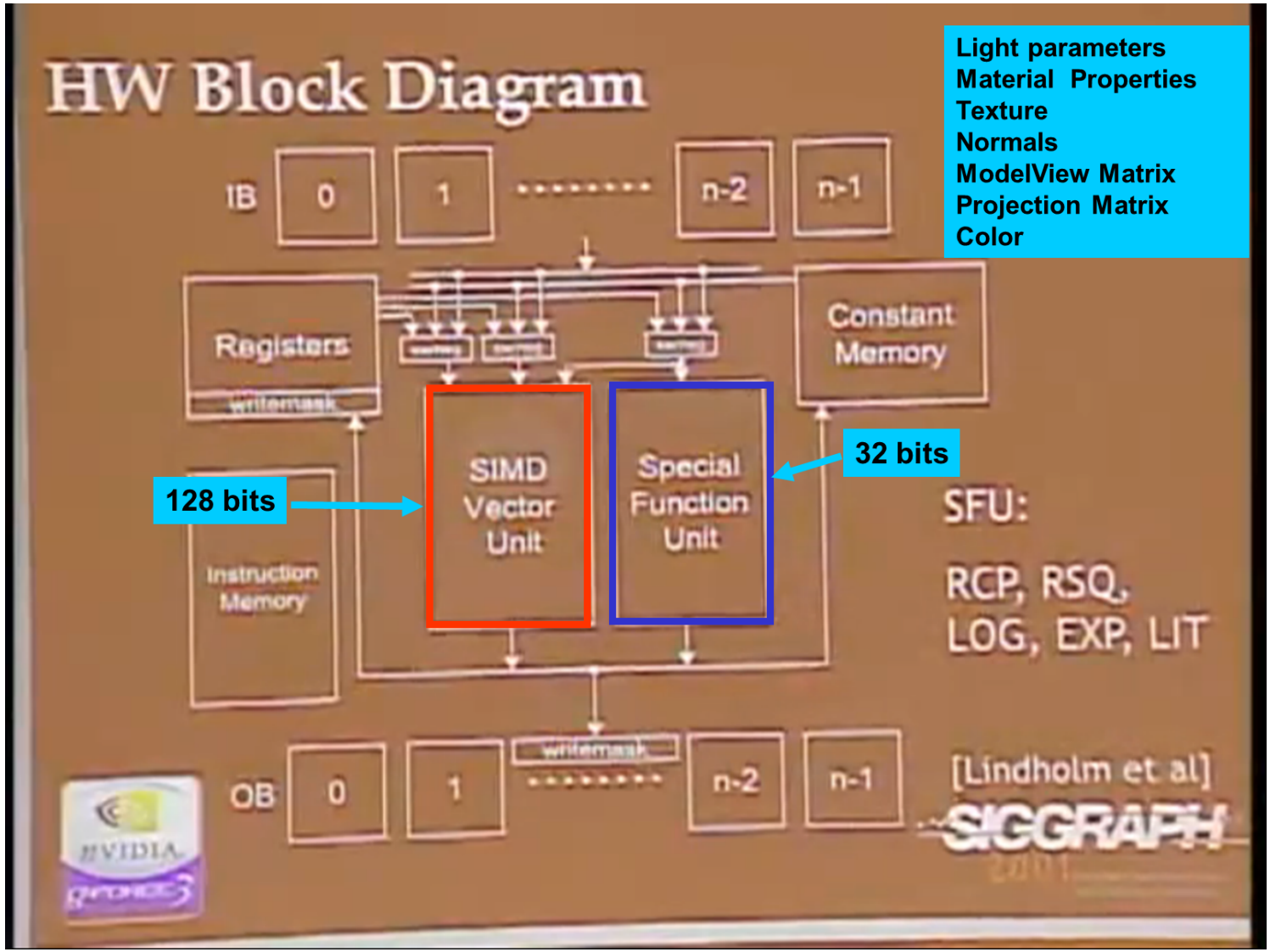
OpenGL Shading Language (**GLSL**)

GLSL I

Chapter 9.4

OpenGL Geometry Pipeline

- Most OpenGL functions **change the OpenGL State** but **do not cause anything** to flow down the pipeline that can change the display.

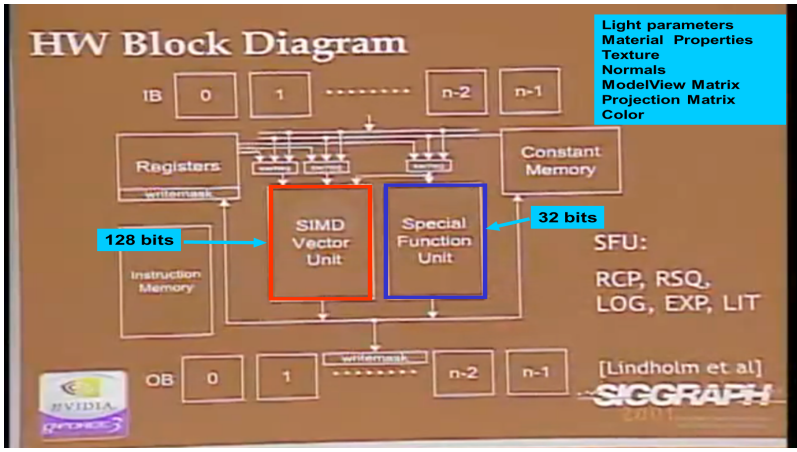


OpenGL Geometry Pipeline

However, once we execute any **glVertex function** or **any OpenGL function that generates vertices** the pipeline goes into action.

With the **fixed-function pipeline**, various calculations are made in the pipeline to determine if the primitive to which the **vertex** belongs is visible and, if it is, to **color the corresponding pixels** in the frame buffer.

These calculations generally **do not change the OpenGL State.** Subsequent executions of **glVertex** invoke the same calculations but use **updated values of state variables if any state changes have been made in between the calls** to **glVertex**.



OpenGL Geometry Pipeline

If we look at these calculations as the work of the software and hardware that implement the **vertex processor**, we see that there must be multiple types of **variables** involved in the execution of this code.

Some **variables** will change as the **output vertex attributes**, such as the **vertex** color, are computed.

Attribute-qualified variables

Other **variables**, whose values are **determined by the OpenGL State**, cannot be changed by the calculation.

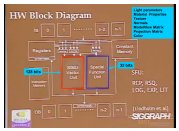
Built in Attribute-qualified variables

Most internal **variables** must be **initialized to their original values** after each **vertex** is processed so that the calculation for the **next vertex** will be identical.

Other **variables must be computed and passed onto the next stage of the pipeline**.

Varying-qualified variables

Consequently, when we **substitute a user-written vertex program for the fixed-function vertex processor**, we must be able to identify these different types of **variables**.



Variables

Attribute, Uniform, Varying, Const

Some **variables** will change as the **output vertex attributes**, such as the **vertex** color, are computed.

Other **variables**, whose values are **determine** **State**, cannot be changed by the calculation.

Most internal **variables** must be **initialized** after each **vertex** is processed so that the calculation for the **next vertex** will be identical.

Other **variables must be computed and passed onto the next stage of the pipeline**.

Application set **variables**

Attribute-qualified variables

Built in Attribute-qualified variables

Varying-qualified variables

Const-qualified **variables**

OpenGL Pixel Pipeline

A **fragment program** also has input **variables**, internal **variables**, and output **variables**. The major difference between a **fragment program** and a **vertex program** is that a **fragment program** will be executed for each **fragment**.

Some **values** will be **provided by the OpenGL State** and thus cannot be changed in the **shader**.

Others will be **changed only on a fragment-by-fragment basis**.

Most shader variables must be initialized for each fragment.

A typical application not only will change state **variables** and entities such as texture maps, but also may use **multiple vertex** and **fragment shaders**. Hence, we also must examine how to **load in shaders**, **how to link them into an GLSL program**, and how **to set this GLSL program to be the one that OpenGL uses**.

This process is not part of the **GLSL** language but is accomplished using a set of functions that are now part of the **OpenGL** core.

Because **each vertex triggers** an execution of the **current vertex shader** independent of any other vertex and, likewise, **each fragment** triggers the execution of the **current fragment shader**, there is the potential for extensive parallelism to speed the processing of both **vertices** and **fragments**.

GLSL

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

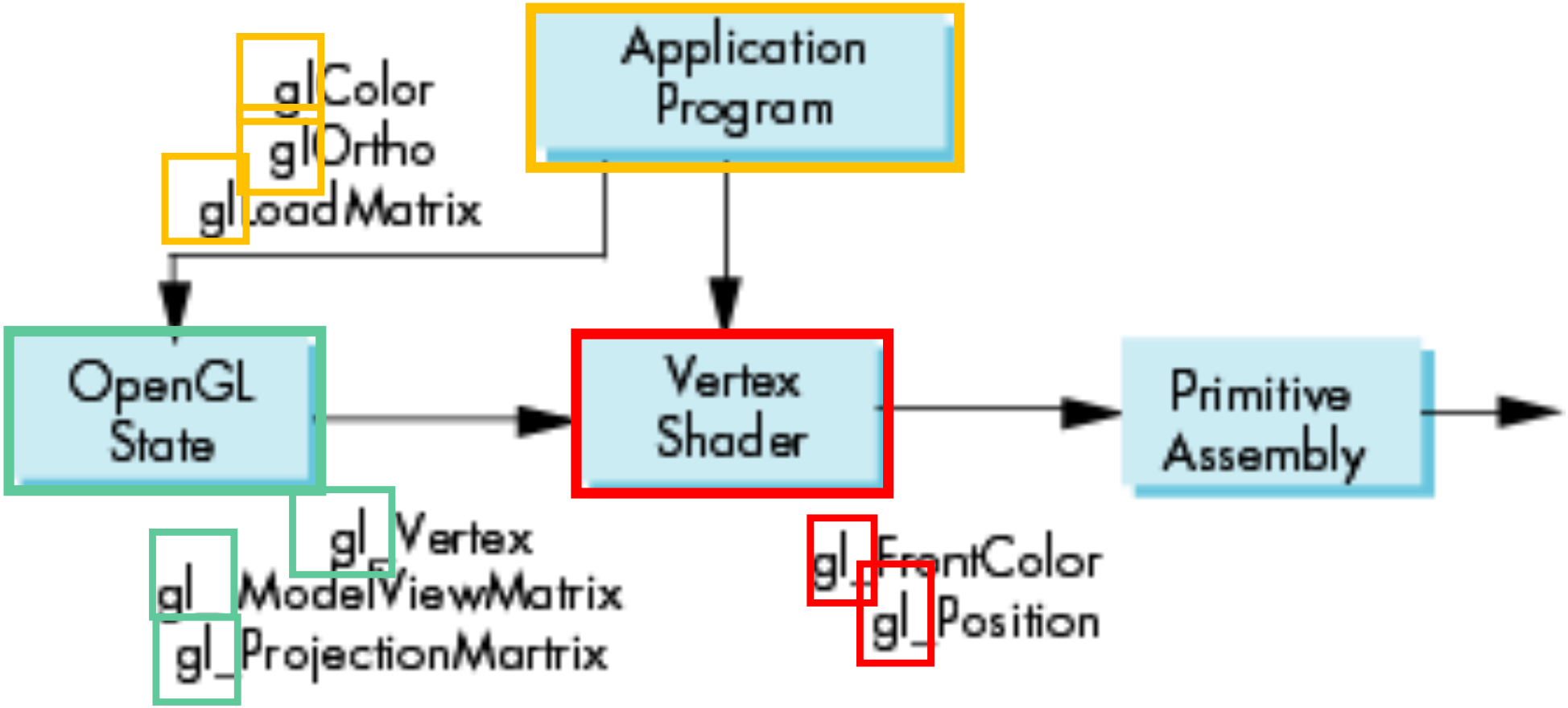
Simple Vertex Shader
it replaces the fixed-function vertex operation

```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
void main (void)
{
    gl_Position = gl_ProjectionMatrix*gl_ModelViewMatrix*gl_Vertex;
    gl_FrontColor = red;
}
```

gl_

For each vertex, the input to the vertex program can use the vertex position defined by the application program and most of the information that is in the OpenGL State, including the current color, texture coordinates, material properties, and transformation matrices. In addition, the application program can pass other application-specific information on a per-vertex basis to the vertex shader.

Execution Model



```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
void main (void)
{
    gl_Position = gl_ProjectionMatrix*gl_ModelViewMatrix*gl_Vertex;
    gl_FrontColor = red;
}
```

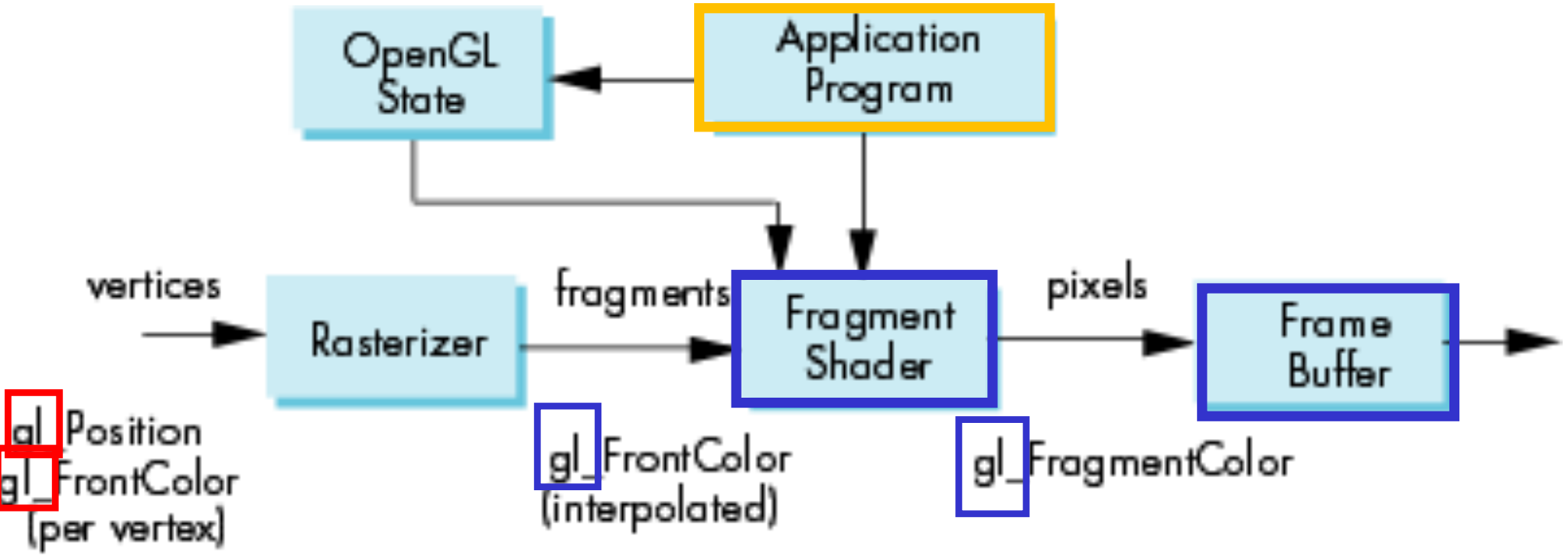
Simple Fragment Program

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

Scan Conversion

If the **vertex** is not eliminated by clipping, it goes on to primitive assembly and then to the **rasterizer**, which **generates fragments** that are then processed by the **fragment processor** using either **fixed-function fragment processing** or an application-defined **fragment program**, **or fragment shader.** **Vertex** attributes, such as **vertex colors** and **positions**, are **interpolated by the rasterizer** across a **primitive** to generate the corresponding **fragment** attributes.

Execution Model




```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
void main (void)
{
    gl_Position = gl_ProjectionMatrix*gl_ModelViewMatrix*gl_Vertex;
    gl_FrontColor = red;
}
```

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

GLSL 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-qualified **variables**

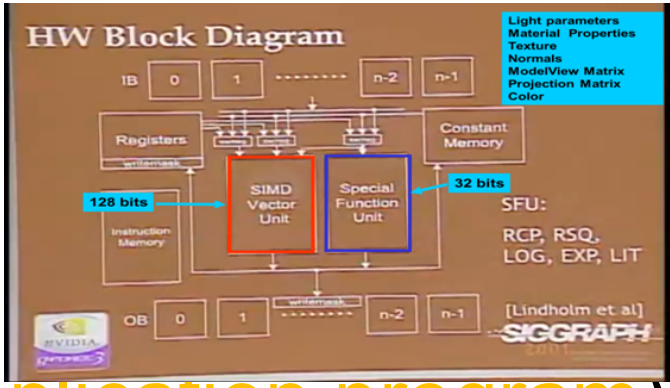
- **Attribute**-qualified **variables** **can change at most once** per **vertex**

  Cannot be used in **fragment** shaders

- **Built in Attribute**-qualified **variables** (**OpenGL State variables**)

  `gl_Color`

  `gl_ModelViewMatrix`



- **User defined** **variables** (defined in application program)

```
attribute float temperature
attribute vec3 velocity
```

Uniform-qualified variables

- **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

- **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;
}
```

Variables that are passed from vertex shader to fragment shader

User-defined varying variables that are set in the **vertex program** are **automatically interpolated by the rasterizer**. It does not make sense to define a **varying variable** in a **vertex shader** and not use it in a **fragment shader.**

Required Fragment Shader

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

- **Variables that are passed from vertex shader to fragment 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;
}
```

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);
```

GLSL II

Chapter 9.6

Objectives

- Coupling GLSL to Applications
- Example applications

Coupling GLSL to Applications

There are usually **eight steps in initializing one or more shaders in the application**:

1. Read the shader source.
2. Create a program **object**.
3. Create shader **objects**.
4. Attach the shader **objects** to the program **object**.
5. Compile the shaders.
6. Link everything together.
7. Select current program **object**.
8. Align **uniform** and **attribute** **variables** between the **Application** and the shaders.

(uniform-qualified **variables** `const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);`)

(attribute-qualified **variables** `varying vec3 color_out;` `uniform float time;`)

Coupling GLSL to Applications

There are usually **8 steps in initializing one or more shaders in the application**:

1. Read the shader source.

```
74      /* read shader files */
75      vSource = readShaderSource(vShaderFile);
76      if(vSource==NULL)
77      {
78          printf( "Failed to read vertex shader\n");
79          exit(EXIT_FAILURE);
80      }
81
82      fSource = readShaderSource(fShaderFile);
83      if(fSource==NULL)
84      {
85          printf("Failed to read fragment shader");
86         exit(EXIT_FAILURE);
87      }
```

Coupling GLSL to Applications

There are usually **8 steps in initializing one or more shaders in the application**:

2. Create a program **object**.
3. Create shader **objects**.

```
89      /* create program and shader objects */
90      vShader = glCreateShader(GL_VERTEX_SHADER);
91      fShader = glCreateShader(GL_FRAGMENT_SHADER);
92      program = glCreateProgram();
93
```

Coupling GLSL to Applications

There are usually **8 steps in initializing one or more shaders in the application**:

4. Attach the shader **objects** to the program **object**.

```
93
94      /* attach shaders to the program object */
95      glAttachShader(program, vShader);
96      glAttachShader(program, fShader);
97
```

Coupling GLSL to Applications

There are usually **8 steps in initializing one or more shaders in the application**:

5. Link everything together.

```
131
132     /* link  and error check */
133     glLinkProgram(program);
134     glGetProgramiv(program, GL_LINK_STATUS, &status);
135     if(status==GL_FALSE)
136     {
137         printf("Failed to link program object.\n");
138         glGetProgramiv(program, GL_INFO_LOG_LENGTH, &elength);
139         ebuffer = malloc(elength*sizeof(char));
140         glGetProgramInfoLog(program, elength, &elength, ebuffer);
141         printf("%s\n", ebuffer);
142         exit(EXIT_FAILURE);
143     }
144
```

Coupling GLSL to Applications

There are usually **8 steps in initializing one or more shaders in the application**:

7. Select current program **object**.

```
145     /* use program object */
146     glUseProgram(program);
```

Coupling GLSL to Applications

There are usually **8 steps in initializing one or more shaders in the application**:

8. Align **uniform** and **attribute** **variables** between the **application** and the shaders.

```
148         /* set up uniform parameter */
149         timeParam = glGetUniformLocation(program, "time");
150     }
```

Program Object

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

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

```
89      /* create program and shader objects */
90      vShader = glCreateShader(GL_VERTEX_SHADER);
91      fShader = glCreateShader(GL_FRAGMENT_SHADER);
92      program = glCreateProgram();
```

Reading a Shader

Shaders 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

fPassThrough.glsl | vmesh.glsl | wave.c*

```
uniform float time; /* in milliseconds */

void main()
{
    float s;
    vec4 t = gl_Vertex;
    t.y = 0.1*sin(0.001*time+5.0*gl_Vertex.x)*sin(0.001*time+5.0*gl_Vertex.z);
    gl_Position = gl_ModelViewProjectionMatrix * t;
    gl_FrontColor = gl_Color;
}
```

```
74
75  /* read shader files */
76  vSource = readShaderSource(vShaderFile);
77  if(vSource==NULL)
78  {
79      printf( "Failed to read vertex shader\n");
80      exit(EXIT_FAILURE);
    }
```

Shader Reader

```
#include <stdio.h>

char* readShaderSource(const char* shaderFile)
{
FILE* fp = fopen(shaderFile, "r");
char* buf;
long size;
if(fp == NULL) return(NULL);
fseek(fp, 0L, SEEK_END); /* end of file */
size = ftell(fp);
fseek(fp, )L, SEEK_SET); /* start of file */

buf = (char*) malloc(statBuf.st_size + 1 * sizeof(char));
fread(buf, 1, size, fp);
buf[size] = '\0';  /null termination */
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 attribute variables** are named in the shader
- Linker forms a **table**
- **Application** can **get index from table** and tie it to an **Application variable**
- Similar process for **uniform variables**

Vertex Attribute index Example

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

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

Uniform Variable index 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(angleParam, my_angle);
```

Objectives

- Coupling GLSL to Applications

- Example **Applications**

Chapter 9.7

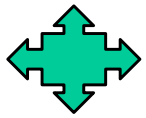
Although there is a **fair number of OpenGL functions** here, **most of them are used only during initialization and their usage does not change much from application to application**.

First, we set up various shaders and link them to program objects. This part can be put in an **initialization function in the OpenGL program**. **Second**, during the execution of the OpenGL program, we either **get values from shaders or send values to shaders**. Usually, we do these operations where we would normally **set values for colors, texture coordinates, normals, and other program variables with an application that uses the fixed-function pipeline**.

Vertex Shader Applications

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

Wave Motion Vertex Shader

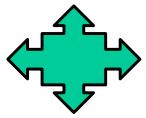


```
uniform float time;
uniform float xs, zs,          // frequencies
uniform float h;               // height scale
void main()
{
vec4 t =gl_Vertex;
t.y = gl_Vertex.y + h*sin(time + xs*gl_Vertex.x) + h*sin(time + zs*gl_Vertex.z)
gl_Position = gl_ModelViewProjectionMatrix*t;
}
```

In our first example, we scale each **vertex** so that the object appears to expand and contract. The **scale factor** varies sinusoidally, based on a **time parameter** that is passed in as a **uniform variable** from the **OpenGL program** as follows:

/* **vertex shader** that moves **vertex locations** sinusoidally */ uniform float time;

/* **value** provided by application program */

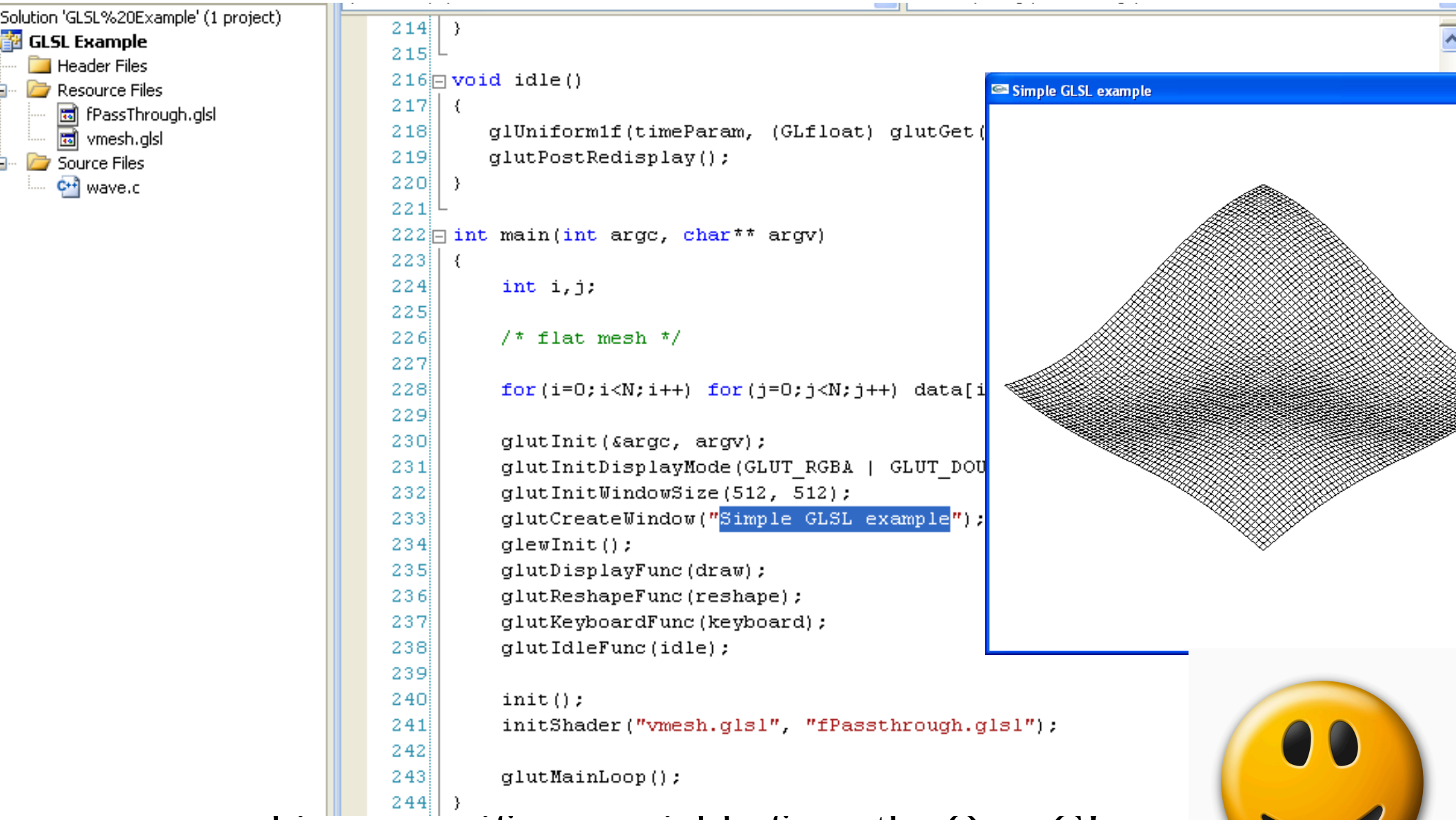


Wave Motion Vertex Shader

```
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;
}
```

Scale each **vertex** so that the **object appears to expand and contract**. The **scale factor varies sinusoidally**, based on a time parameter that is passed in as a **uniform variable** from the **OpenGL program**.

Wave Motion Vertex Shader



passed in as a uniform variable from the OpenGL pro

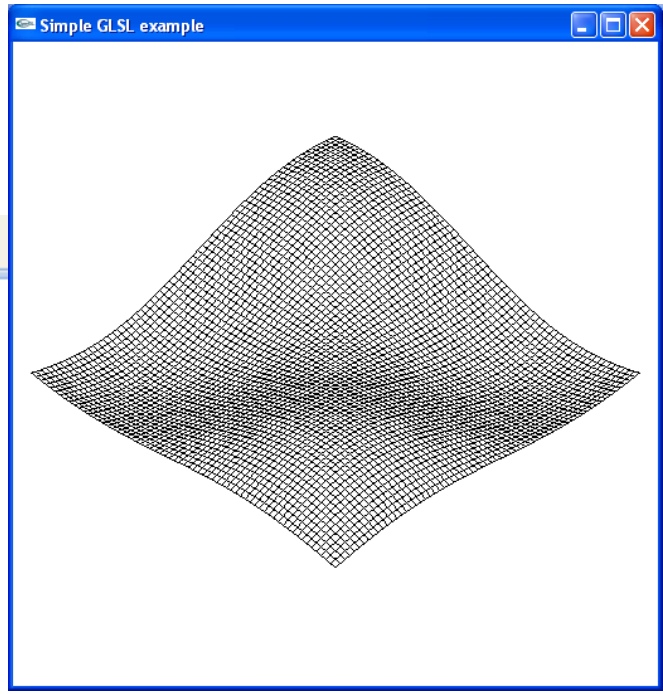
Wave Motion Vertex Shader

vmesh.glsl | fPassThrough.glsl | wave.c

```
1  uniform float time; /* in milliseconds */
2
3  void main()
4  {
5      float s;
6      vec4 t = gl_Vertex;
7      t.y = 0.1*sin(0.001*time+5.0*gl_Vertex.x)*sin(0.001*time+5.0*gl_Vertex.z);
8      gl_Position = gl_ModelViewProjectionMatrix * t;
9      gl_FrontColor = gl_Color;
10 }
11
```

vmesh.glsl | fPassThrough.glsl | wave.c

```
1  // fPassThrough.glsl
2  // Pass through fragment shader.
3
4  void main()
5  {
6      gl_FragColor = gl_Color;
7  }
8
9
10
11
```



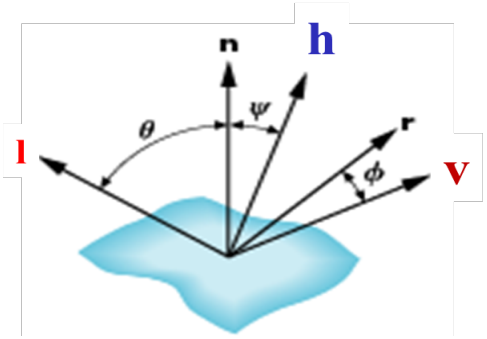
Particle System

Last Homework

```
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)
}
```

Particle systems provide one of the most important approaches to **simulating complex real-world objects, such as clouds, trees, water, and hair.**

Modified Phong Vertex Shader

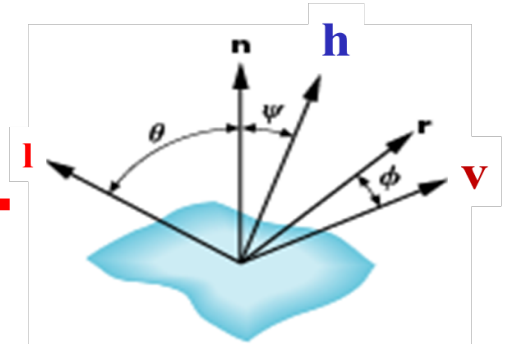


$$I = k_d I_d \, \mathbf{l} \bullet \mathbf{n} + k_s I_s (\mathbf{v} \bullet \mathbf{r})^{\alpha} + k_a I_a$$

```
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);
```

Modified Phong Vertex Shader



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

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

float Kd = max(dot(L, N), 0.0);
float Ks = pow(max(dot(N, H), 0.0), gl_FrontMaterial.shininess);
float Ka = 0.0;

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

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

Pass Through Fragment Shader

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

   float Kd = max(dot(L, N), 0.0);
   float Ks = pow(max(dot(N, H), 0.0), gl_FrontMaterial.shininess);
   float Ka = 0.0;

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

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

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

Vertex Shader for per Fragment Lighting

- Variables that are passed from vertex shader to fragment shader

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

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);
H = normalize(L + E);
}
```

Fragment Shader for Modified Phong Lighting

```
varying vec3 N;
varying vec3 L;
varying vec3 E;
varying vec3 H;

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

- **Variables that are passed from vertex shader to fragment shader**

Fragment Shader for Modified Phong Lighting

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

vec4 diffuse  = Kd * gl_FrontLightProduct[0].diffuse;
vec4 specular = Ks * gl_FrontLightProduct[0].specular;
vec4 ambient  = Ka * gl_FrontLightProduct[0].ambient;

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

Vertex vs Fragment Lighting

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);
```

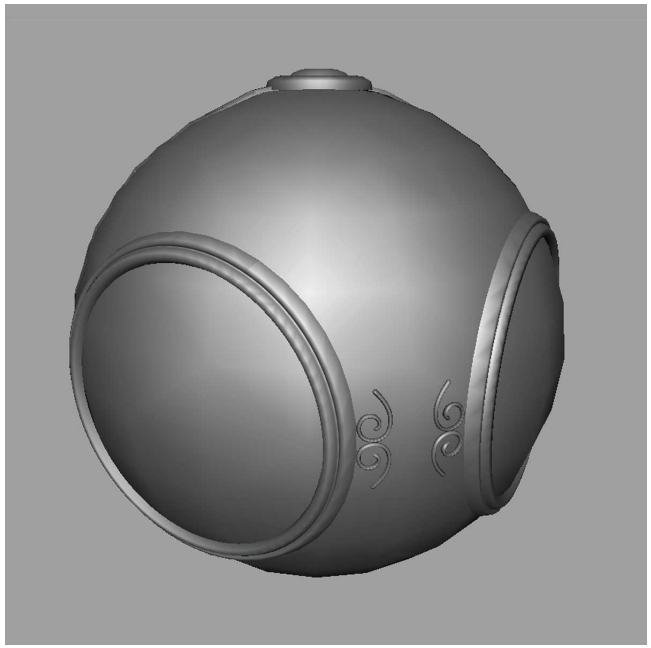
- **In application**:

```
texMapLocation = glGetUniformLocation(myProg,"myTexture");
glUniform1i(texMapLocation, 0);

/* assigns to texture unit 0 */
```

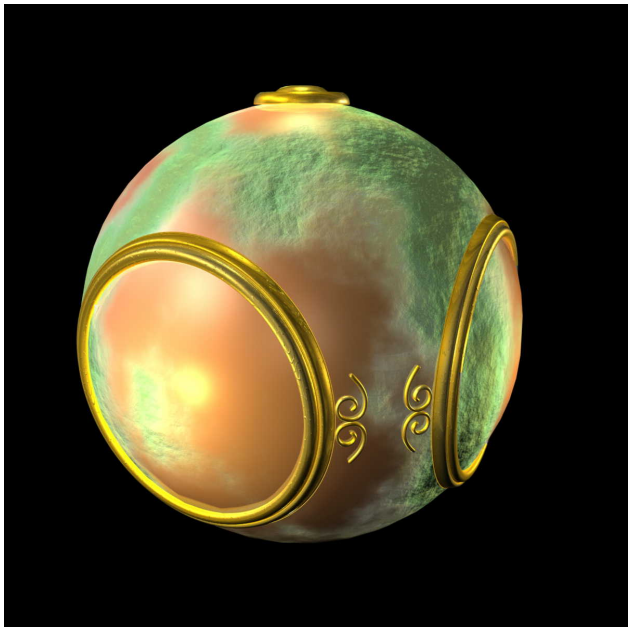
Fragment Shader Applications

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**

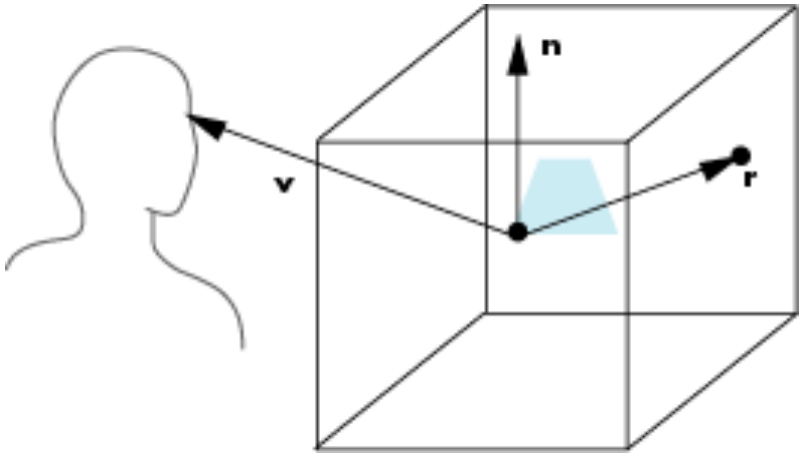
Also supported in **GLSL** through cubemap sampler

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

Texture coordinates must be 3D

Environment Map

Use reflection vector **r** 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 to the **shader** as a **uniform variable**

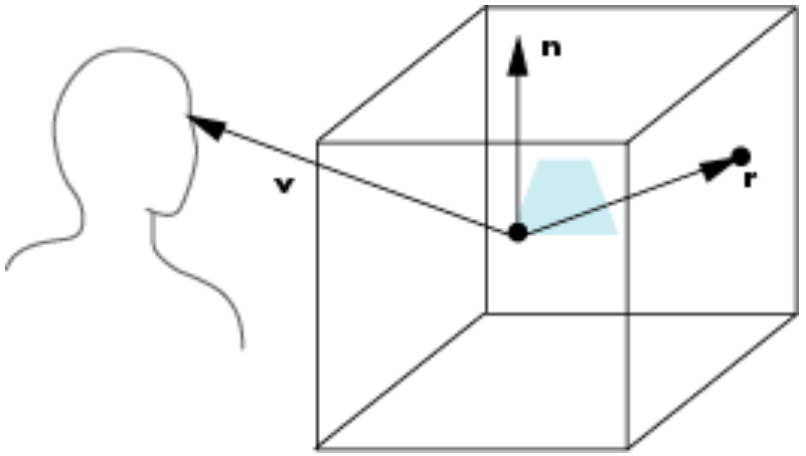
Can also use **reflection map** or **refraction map** (for example to simulate water)

Reflection Map Vertex Shader

varying vec3

- Variables that are passed from vertex shader to fragment shader

```
void main(void)
{
gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;
    vec3 N = normalize(gl_NormalMatrix*gl_Normal);
    vec4 eyePos = gl_ModelViewMatrix*gl_Vertex;
    R = reflect(-eyePos.xyz, N);
}
```



Reflection Map Fragment Shader

```
varying vec3 R;

void main(void)
{
  gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;
  vec3 N = normalize(gl_NormalMatrix*gl_Normal);
  vec4 eyePos = gl_ModelViewMatrix*gl_Vertex;
  R = reflect(-eyePos.xyz, N);
}
```

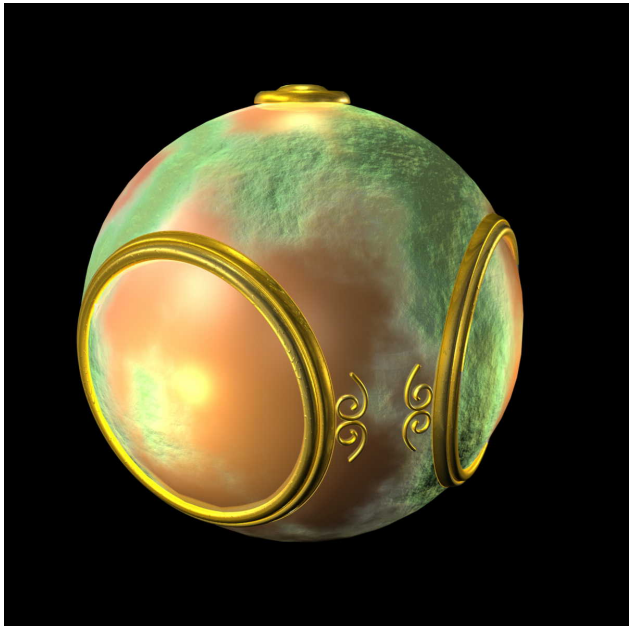
- **Variables that are passed from vertex shader to fragment shader**

```
varying vec3 R;
uniform samplerCube texMap;

void main(void)
{
gl_FragColor = textureCube(texMap, R);
}
```

Bump Mapping

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



NEXT.

10.25.2023 (W 5:30 to 7) (19)		**Lecture 10** **(Modeling and Hierarchy)**
10.30.2023 (M 5:30 to 7) (20)		**PROJECT 3**
11.01.2023 (W 5:30 to 7) (21)		**EXAM 3 REVIEW**
11.06.2023 (M 5:30 to 7) (22)		**EXAM 3**

At 6:45 PM.

End Class 18

VH, Download Attendance Report
Rename it:
10.23.2023 Attendance Report FINAL

VH, upload Lecture 9 to CANVAS.