Intro to GLSL (OpenGL Shading Language)

Cliff Lindsay

Talk Summary

Topic Coverage

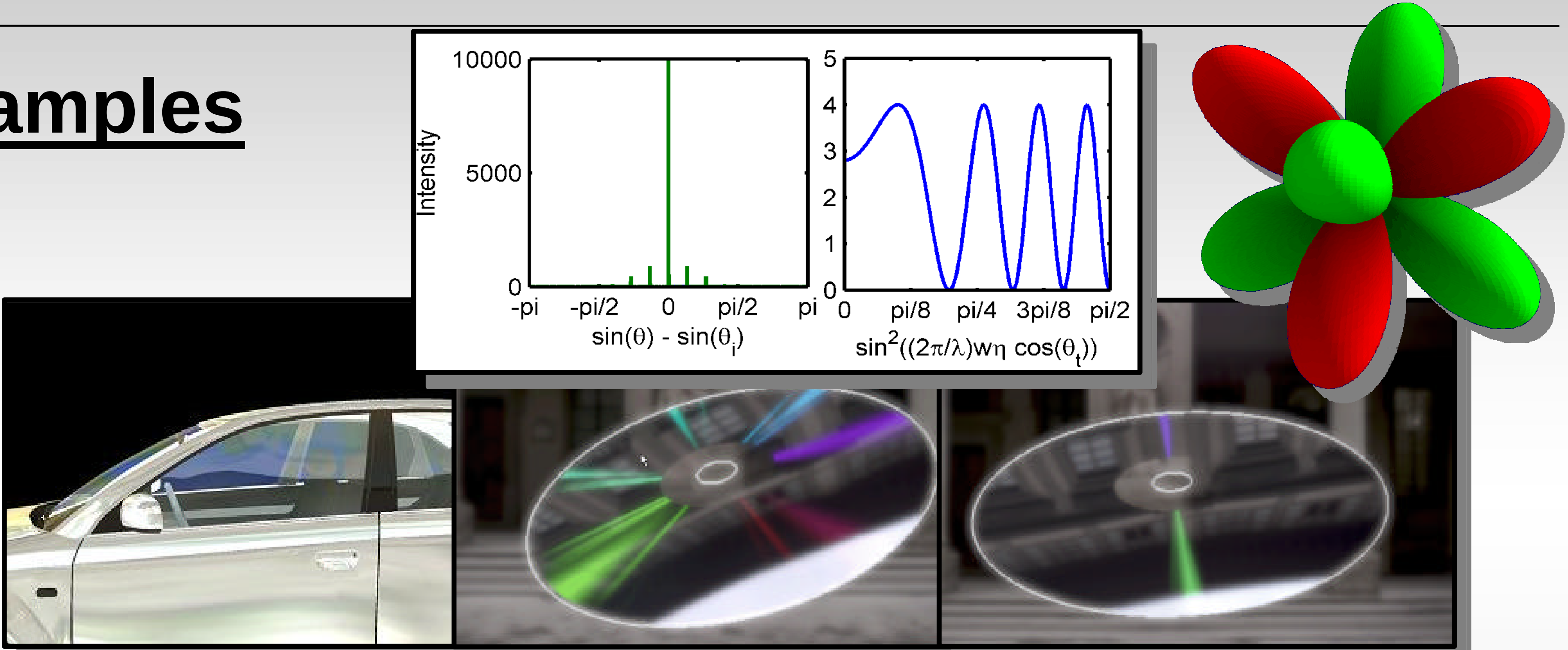
- Define Shading Languages (loosely)
- High Level View of GPU
- OpenGL Shading Language
- Example Shader

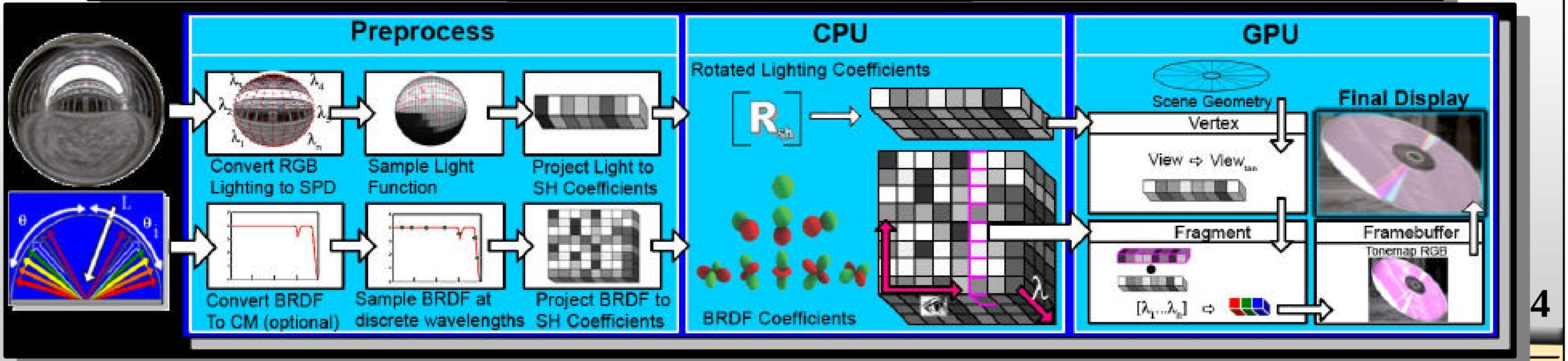
Who Am I?

- Ph.D. Student @ WPI
- Advisor = Emmanuel
- Interests:
  - *Real-time Rendering*
  - *Photorealistic Rendering*
  - *Image/Video Based Rendering*
  - *Computational Photography*
- Done: Published Papers, M.S. Thesis

Some Work We've Done

Samples





Preprocess

Convert RGB Lighting to SPD

Sample Light Function

Project Light to SH Coefficients

Convert BRDF To CM (optional)

Sample BRDF at discrete wavelengths

Project BRDF to SH Coefficients

CPU

Rotated Lighting Coefficients

BRDF Coefficients

GPU

Scene Geometry

Vertex

View ⇨ View_tan

Fragment

Final Display

Framebuffer

Tonemap RGB

Back To Lecture

Q: What is OpenGL Shading Language & Why do we need it?

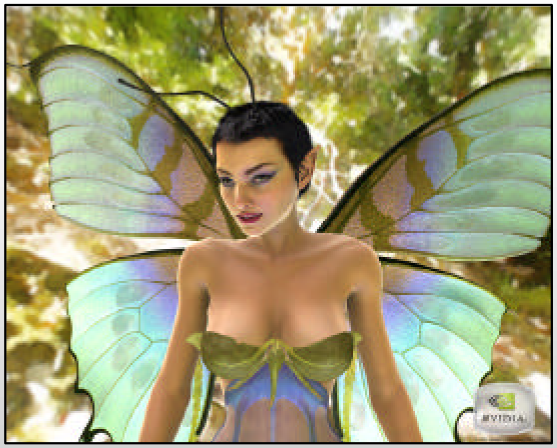
A:

- **OpenGL Fixed Function: Can only select from pre-defined effects (90's)**
  - **E.g. Only two shading models pre-defined**
- **Industry needs *flexibility* (new effects)**
- **GLSL = programmability + access to GPU internals**

Examples of New Effects

Complex Materials

Shadowing



Lighting Environments

Advanced Mapping

History of Shading Languages

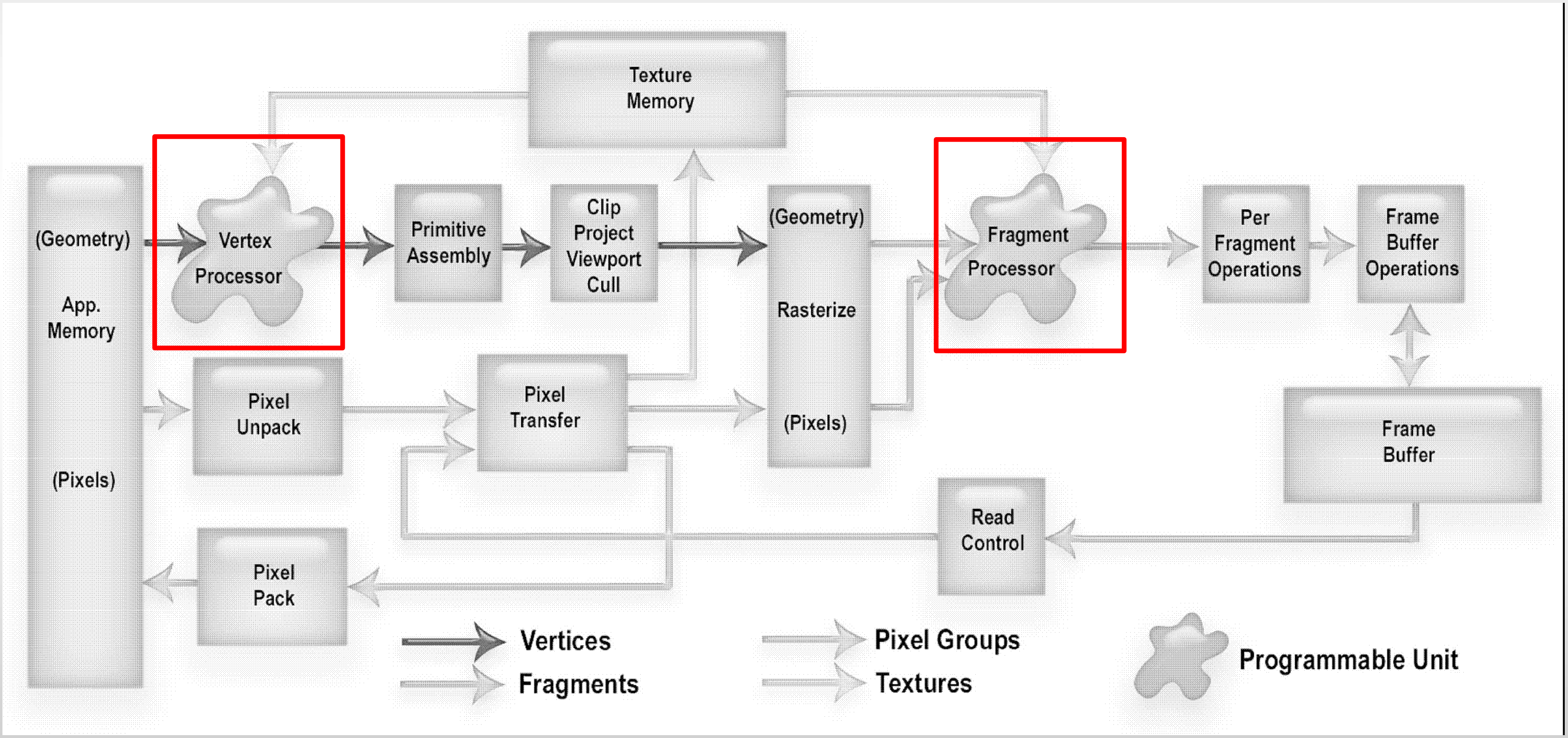
Big Players

- **RenderMan – Pixar, software based in toy story**
- **Cg – nVidia, 1st commercial SL**
- **HLSL – M$/nVidia, Cg & Xbox project**
- **GLSL – SGI, ARB/3DLabs**
- **Stanford RTSL - Academic SLs**

Several others more recently

Shader Pipeline

Programmable Graphics Pipeline



Programmable Pipeline

Programmable Functionality

- Exposed via small programs
- Language similar to c/c++
- Hardware support highly variable

Vertex Shaders

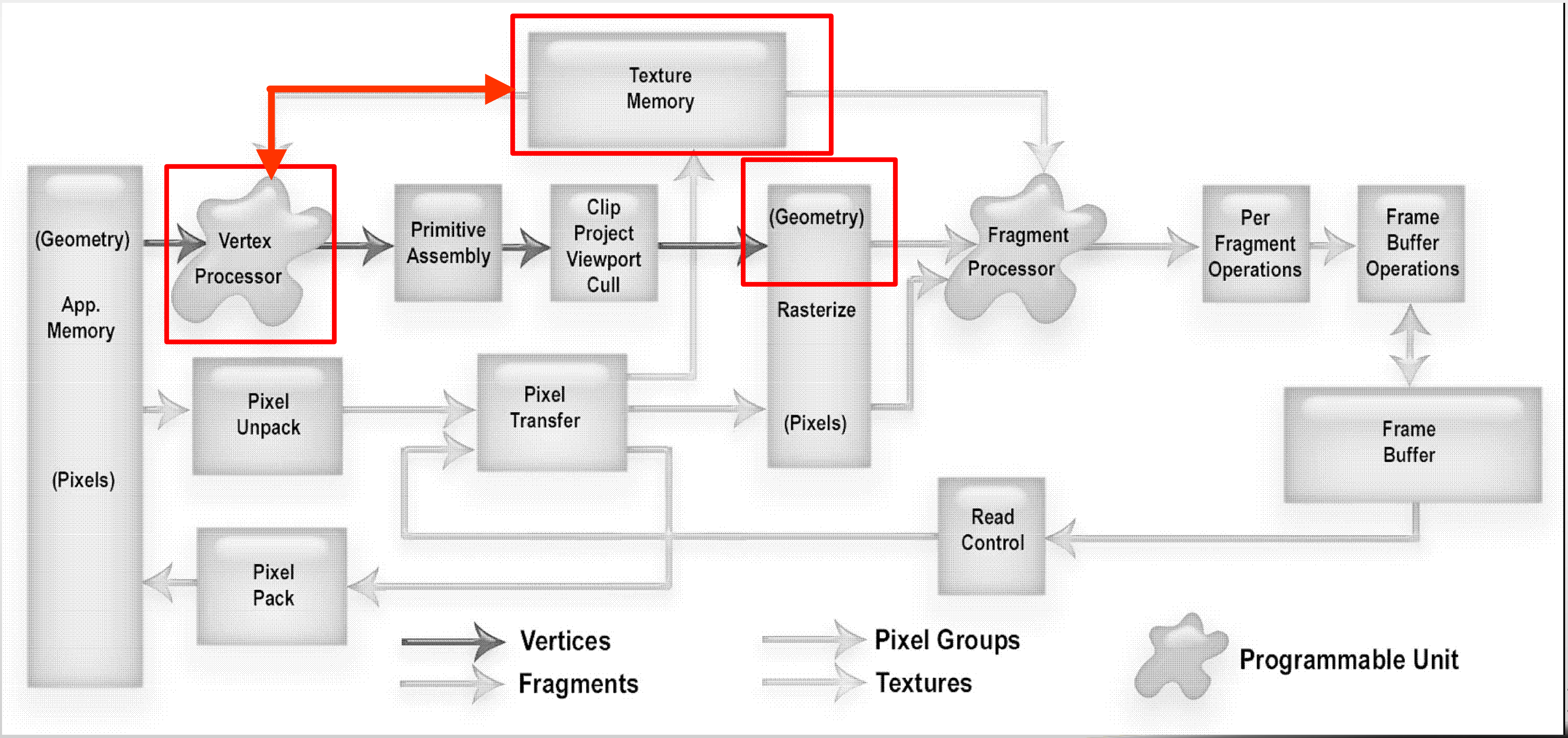
- Input: Application geometry & per vertex attributes
- Transform input in a meaningful way

Fragment Shaders

- Input: Perspective Correct Attributes (interpolated)
- Transform input into color or discard

Recent Advances

- Geometry Shaders
- Texture Fetching Within Vertex Shaders



In General

Some Fixed Functions Are Bypassed

Vertex Tasks

- Vertex Transformation
- Normal Transformation, Normalization
- Lighting
- Texture Coordinate Generation and Transformation

Fragment Tasks

- Texture accesses
- Fog
- Discard Fragment

Anatomy Of GLSL: Data Types

Scalar Types

- float - 32 bit, very nearly IEEE-754 compatible
- int - at least 16 bit, but not backed by a fixed-width register
- bool - like C++, but must be explicitly used for all flow control

Vector Types

- vec[2|3|4] - floating-point vector
- ivec[2|3|4] - integer vector
- bvec[2|3|4] - boolean vector

Matrix Types

- mat[2|3|4] - for 2x2, 3x3, and 4x4 floating-point matrices

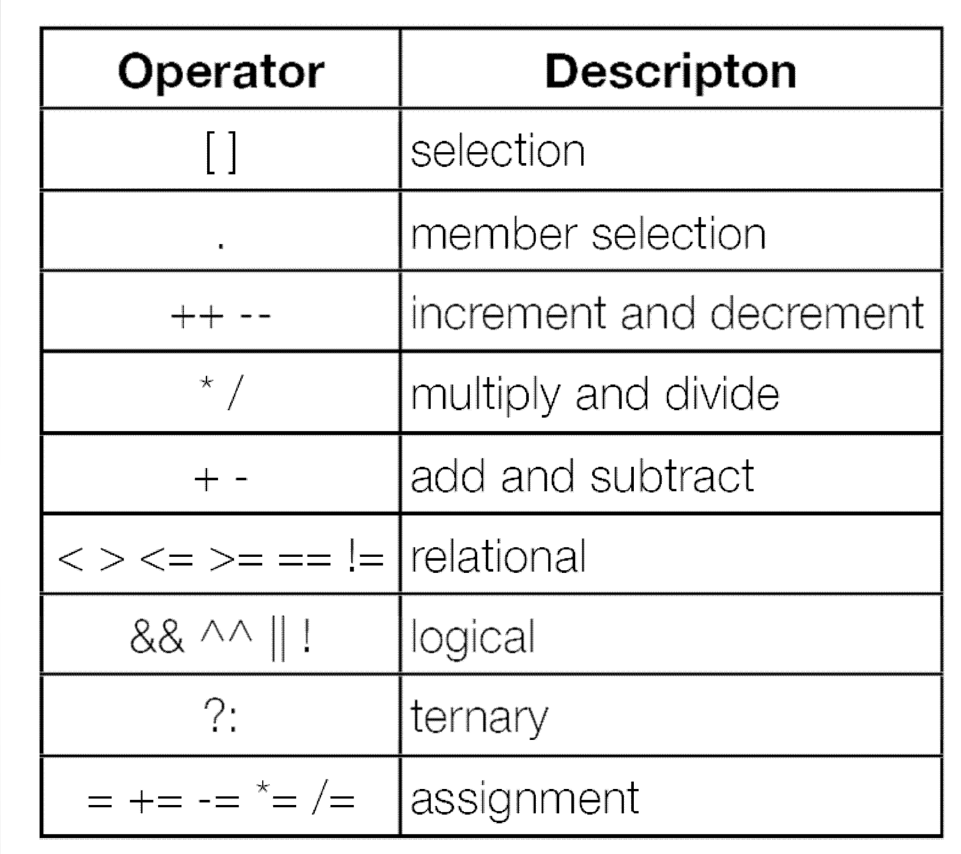
Sampler Types

- sampler[1|2|3]D - for texture data

Anatomy Of GLSL: Operations

Operators

- Behave like in C++
- Component-wise for vector & matrix
- Multiplication on vectors and matrices



Operator	Descripton
[ ]	selection
.	member selection
++ --	increment and decrement
* /	multiply and divide
+ -	add and subtract
< > <= >= == !=	relational
&& ^^ || !	logical
?:	ternary
= += -= *= /=	assignment

Examples:

```
Vec3 t = u * v

float f = v[2]

v.x = u.x + f
```

Anatomy Of GLSL: Structures

Arrays and Structs

- Can declare arrays as in C++ (i.e. vec3 foo[4];)
- Can also declare structs as in C++ (i.e struct foo{vec2 bar;};)

Swizzling

- Can use array-style access to get single vector values
- Swizzling operations via structure member selector (.) more powerful
- Can use only one set per access (.rgba .xyzw .stpq)

```
vec4 baz;
baz.rgba;  //is the same as baz
baz.xy;    //is a vec2
baz.b;     //is a float
baz[2];    //is the same as baz.b
baz.xb;    //illegal
baz.xxx;   //is a vec3
```

Anatomy Of GLSL: Global Qualifiers

Attribute (per vertex)

- Changing info passed app to vertex shader
- No integers, bools, structs, or arrays

Uniform (per primitive)

- Unchanging info passed app to vertex/fragment shader
- Cannot be written to in a shader

Varying (registers writing)

- Info passed from vertex shader to fragment shader
- Interpolated in a perspective-correct manner
- Write in vertex shader, but only read in fragment shader

Const

- To declare non-writable, constant variables

Examples:

Vertex Color

Light Position
Eye Position

Texture/Bump
Map Coords

i.e. p = **3.14**

Anatomy Of GLSL: Flow Control

Loops and Selection

- C++ style if-else
- C++ style for, while, and do

Functions

- Much like C++
- Entry point into a shader is void main()
- Overloading parameter (not return type)
- No support for recursion
- Call by value-return calling convention

Parameter Qualifiers

- in - copy in, but don’t copy out
- out - only copy out
- inout - copy in and copy out

Example Function

```
void ComputeTangent(
       in     vec3 N,
       out    vec3 T,
       inout  vec3 coord)
{

  if(dot(N, coord)>0)
       T = 1.0;
  else
       T = 0.0;
}
```

Anatomy Of GLSL: Built-in Funct

Wide Assortment

- Trigonometry (i.e. cos, sin, tan, etc.)
- Exponential (i.e. pow, log, sqrt, etc.)
- Common (i.e. abs, floor, min, clamp, mix, etc.)
- Geometry (i.e. length, dot, normalize, reflect, etc.)
- Vector relational (i.e. lessThan, equal, any, etc.)

Keep in Mind

- Need to watch out for common reserved keywords
- **Always use built-in functions, don’t implement your own**
- Some functions aren’t implemented on some cards

Anatomy Of GLSL: OpenGL State

Built-in Variables

- Always prefaced with gl_
- Accessible to both vertex and fragment shaders

Uniform Variables

- Matrices (i.e. ModelViewMatrix, ProjectionMatrix, inverses, transposes)
- Materials (in MaterialParameters struct, ambient, diffuse, etc.)
- Lights (in LightSourceParameters struct, specular, position, etc.)

Varying Variables

- FrontColor for colors
- TexCoord[] for texture coordinates

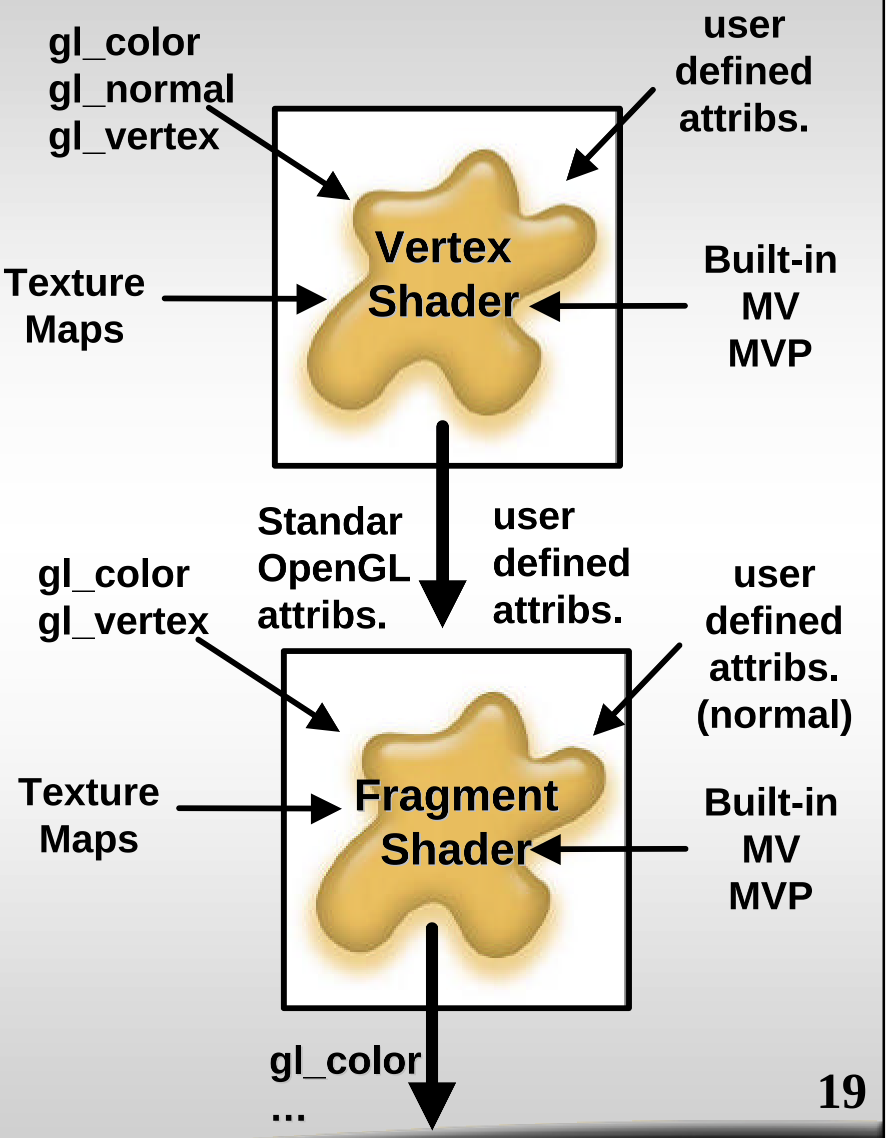
Anatomy Of GLSL: Special Vars

Vertex Shaders

- Have access to several vertex attributes:
  - gl_Color, gl_Normal, gl_Vertex, etc.
- Also write to special output variables:
  - gl_Position, gl_PointSize, etc.

Fragment Shaders

- Have access to special input variables:
  - gl_FragCoord, gl_FrontFacing, etc.
- Also write to special output variables:
  - gl_FragColor, gl_FragDepth, etc.



Example: Phong Shader

Questions?

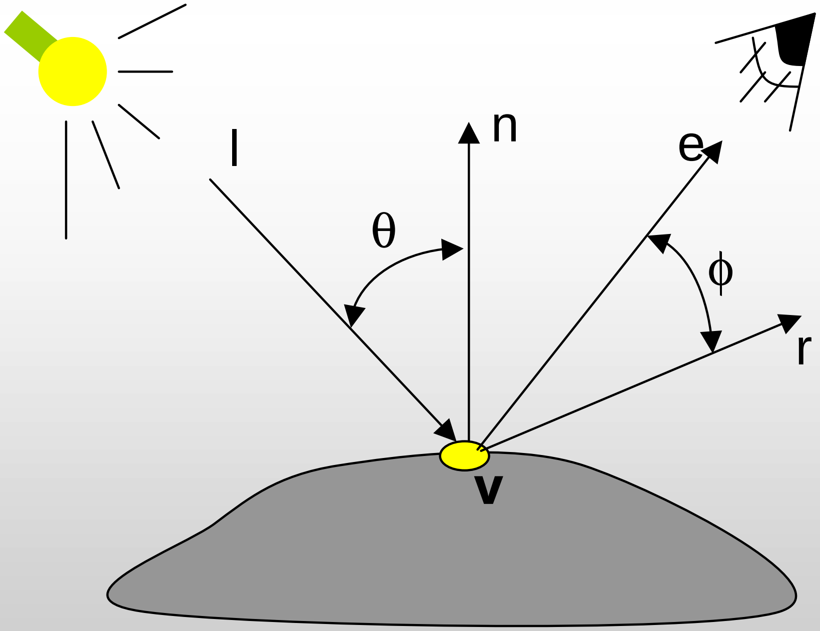
Goals

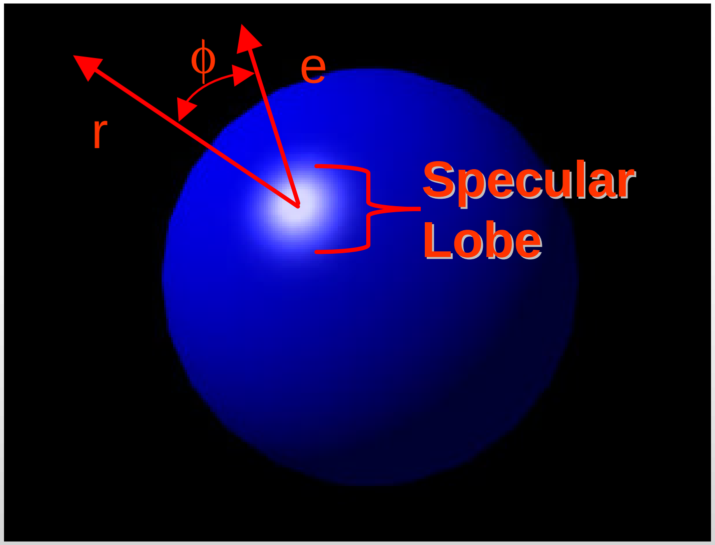
- Phong Illumination Review (1 slide)
- C/C++ Application Setup
- Vertex Shader
- Fragment Shader
- Debugging

Phong Shader Review

Illum = ambient + diffuse + specular

= Ka x I + Kd x I x (cos q) + Ks x I x cos^{n}(f)





[Diagram Courtesy of E. Agu]

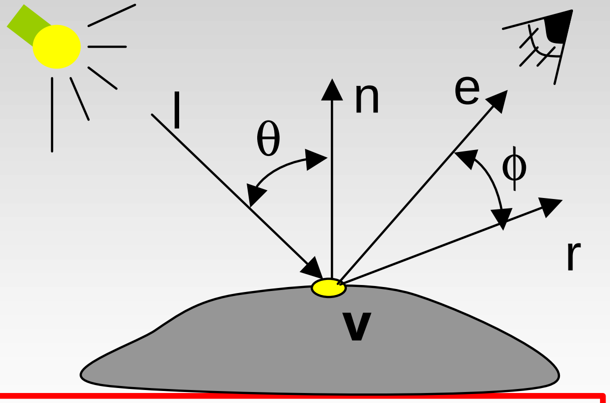
Phong Shader: Setup Steps

- **Step 1: Create Shaders**
  - Create handles to shaders
- **Step 2: Specify Shaders**
  - load strings that contain shader source
- **Step 3: Compiling Shaders**
  - Actually compile source (check for errors)
- **Step 4: Creating Program Objects**
  - Program object controls the shaders
- **Step 5: Attach Shaders to Programs**
  - Attach shaders to program obj via handle
- **Step 6: Link Shaders to Programs**
  - Another step similar to attach
- **Step 7: Enable Program**
  - Finally, let GPU know shaders are ready

Phong Shader: App Setup

```
GLhandleARB phongVS, phongkFS, phongProg; // handles to objects

// Step 1: Create a vertex & fragment shader object
phongVS = glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
phongFS = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);

// Step 2: Load source code strings into shaders
glShaderSourceARB(phongVS, 1, &phongVS_String, NULL);
glShaderSourceARB(phongFS, 1, &phongFS_String, NULL);

// Step 3: Compile the vertex, fragment shaders.
glCompileShaderARB(phongVS);
glCompileShaderARB(phongFS);

// Step 4: Create a program object
phongProg = glCreateProgramObjectARB();

// Step 5: Attach the two compiled shaders
glAttachObjectARB(phongProg, phongVS);
glAttachObjectARB(phongProg, phongFS);

// Step 6: Link the program object
glLinkProgramARB(phongProg);

// Step 7: Finally, install program object as part of current state
glUseProgramObjectARB(phongProg);
```

Phong Shader: Vertex



This Shader Does

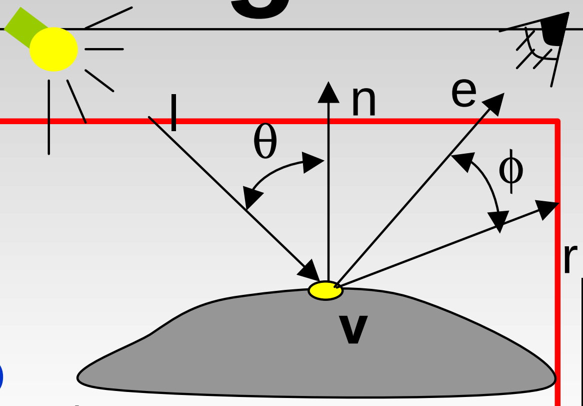
- Gives eye space location for v
- Transform Surface Normal
- Transform Vertex Location

```
varying vec3 N;
varying vec3 v;

void main(void)
{
   v = vec3(gl_ModelViewMatrix * gl_Vertex);
   N = normalize(gl_NormalMatrix * gl_Normal);

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Created For Use Within Frag Shader

(Update OpenGL Built-in Variable for Vertex Position)

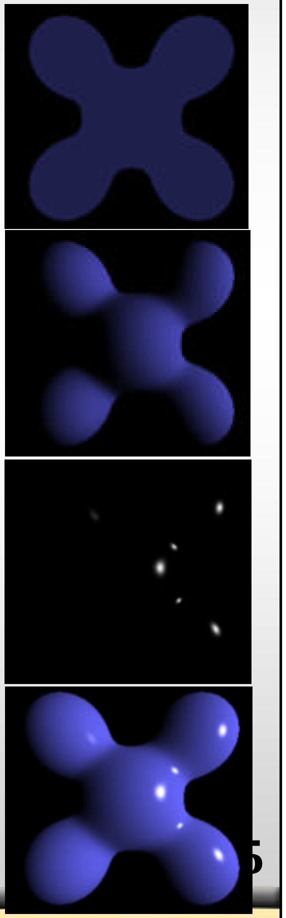
Phong Shader: Fragment



```
varying vec3 N;
varying vec3 v;
```

Passed in From VS

```
void main (void)
{
   // we are in Eye Coordinates, so EyePos is (0,0,0)
   vec3 L = normalize(gl_LightSource[0].position.xyz - v);
   vec3 E = normalize(-v);
   vec3 R = normalize(-reflect(L,N));

   //calculate Ambient Term:
   vec4 Iamb = gl_FrontLightProduct[0].ambient;

   //calculate Diffuse Term:
   vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(N,L), 0.0);

   // calculate Specular Term:
   vec4 Ispec = gl_FrontLightProduct[0].specular
                * pow(max(dot(R,E),0.0), gl_FrontMaterial.shininess);

   // write Total Color:
   gl_FragColor = gl_FrontLightModelProduct.sceneColor + Iamb + Idiff + Ispec;
}
```



Phong Shader: Debugging

****Many things will silently fail during setup****

- No good automatic debugging tools for GLSL yet exist
- Common show-stoppers:
  - Typos in shader source
  - Assuming implicit type conversion
  - Attempting to pass data to undeclared varying/uniform variables
- Extremely important to check error codes, use status functions like:
  - glGetObjectParameter{I|f}vARB (GLhandleARB shader, GLenum whatToCheck, GLfloat *statusVals)
- Subtle Problems
  - Type over flow
  - Shader too long
  - Use too many registers

Phong Shader: Demo

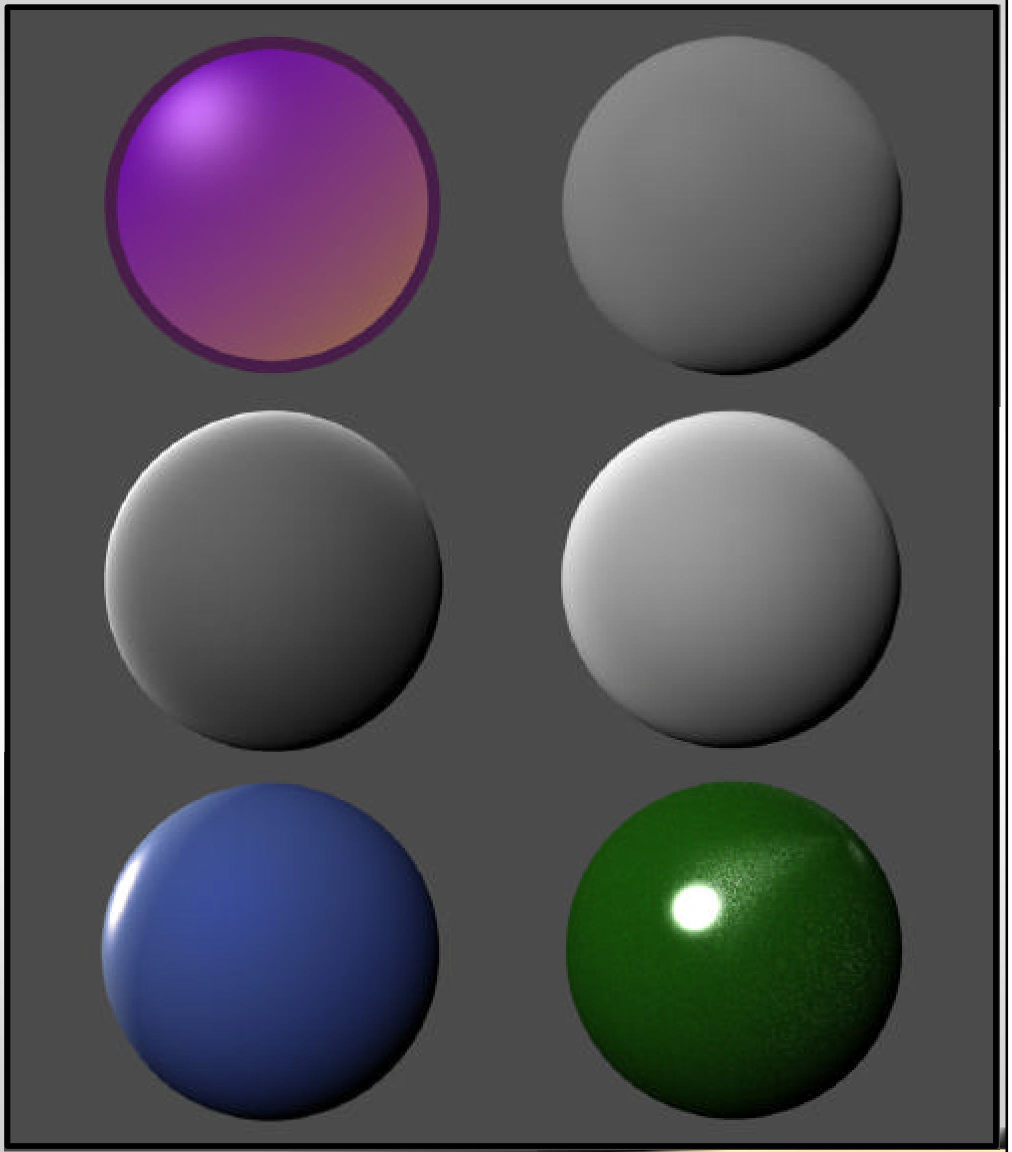
Click Me!

Assignment: Write A Shader

Reflection Models

- Ashikhmin-Shirley
- Fresnel
- Lafortune
- Ward
- Oren Nayer
- Velvet
- Car paint
- Gooch

We'll let you know which one(s) soon! (Next week)



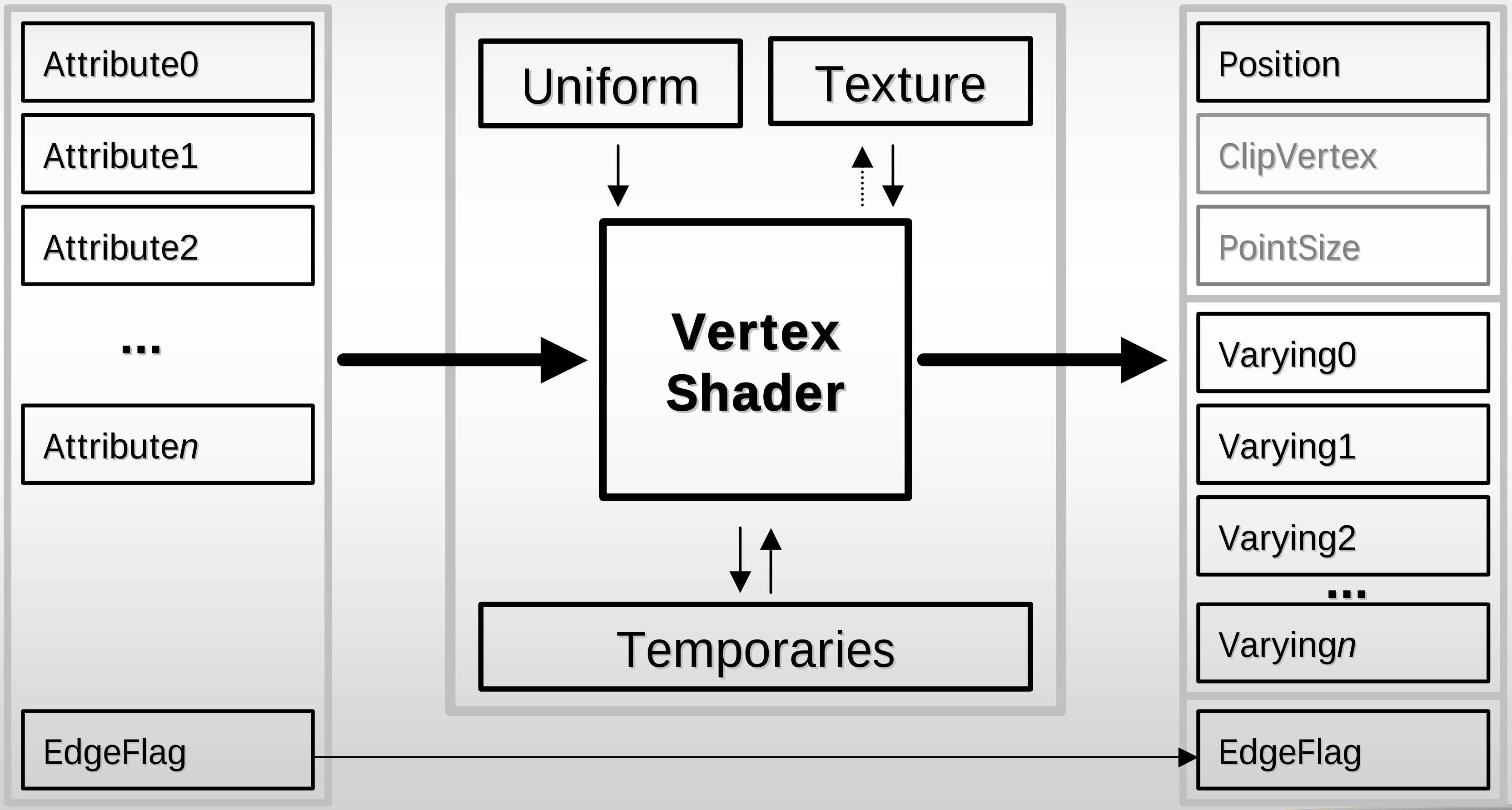
Questions?

References

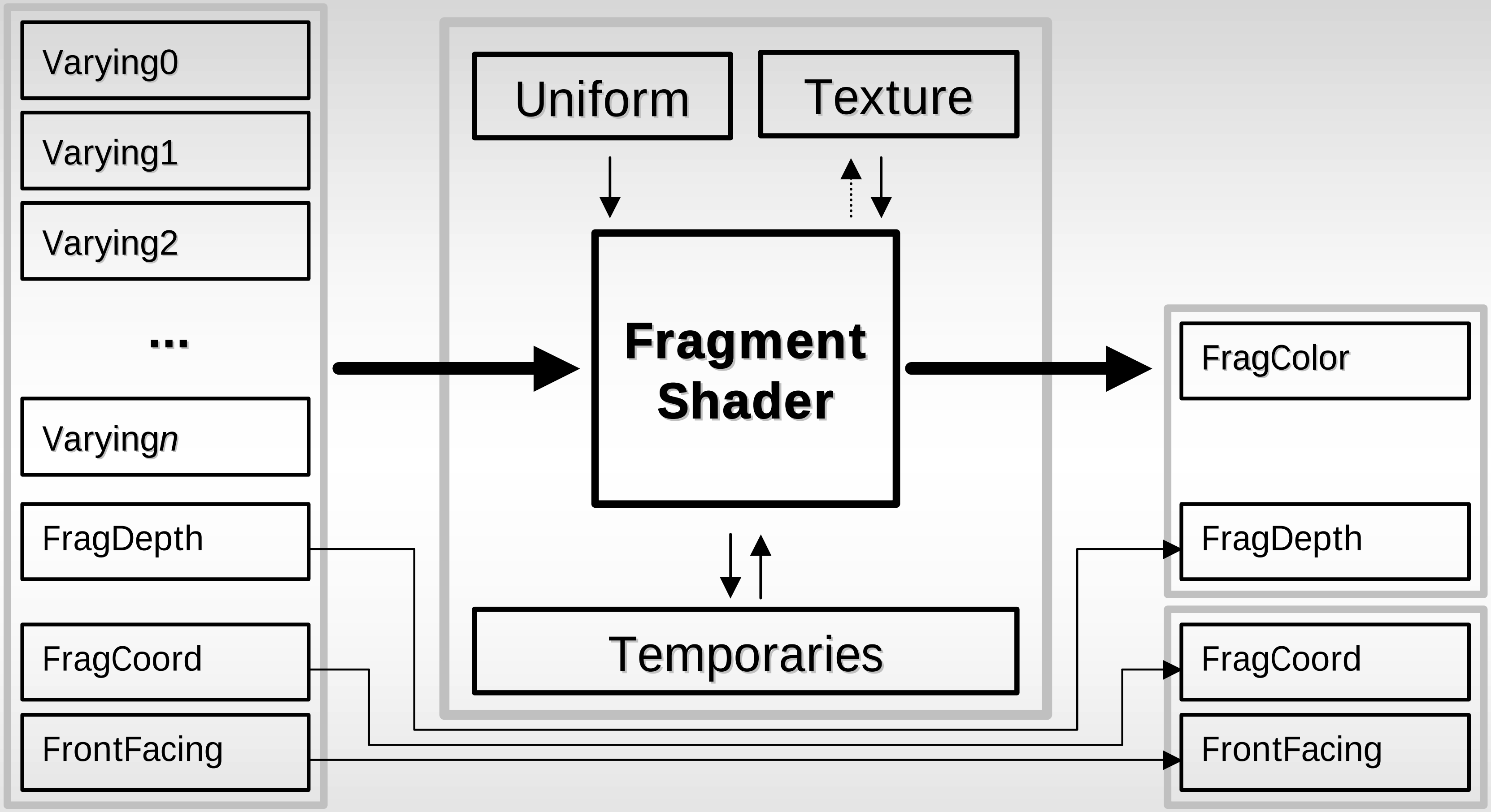
- OpenGL Shading Language (Orange Book), Randi Rost, 2004
- Intro GLSL, Talk Slides Randi Rost 3DLabs, 2005
- Intro GLSL, Teaching Slide, Mike Bailey (my ugrad graphics teacher) U of O, 2006
- Intro GLSL, Teaching Slides, Keith O'connor, GV2 (U of Dublin)
- OpenGL Shading Language, Teaching Slides, Jerry Talton, Stanford, 2006
- Real-time Shading, John Hart, 2002, AK Peters
- OpenGL 2.0 Specification, OpenGL ARB, 2004, OpenGL.org
- OpenGL Shading Language Specification, 2004, OpenGL.org
- The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics, Randima Fernando, Mark Kilgard, 2003

Shader Vertex Processing

All value are inputs to Shaders



Shader Fragment Processing



Same as vertex, all values are input into shader