EECS 487: Interactive Computer Graphics

Lecture 18:
- Programmable Shaders

Shader Programming: Basic Idea

Replace vertex and/or fragment computations with user program, or "shader"

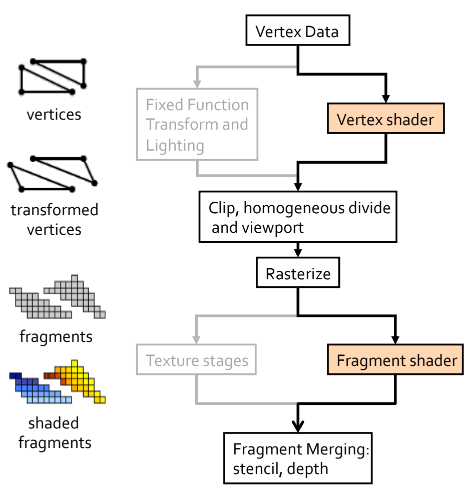
Shaders are small, stateless programs run on the GPU with a high degree of parallelism

Written in a high-level language that hides parallelism from the programmer: GLSL, HLSL, Cg

Graphics driver compiles shaders and links them into a program at application run-time, within an OpenGL or Direct3D program

Application activates the program to replace fixed-functionality of the graphics pipeline

Programmable GPU

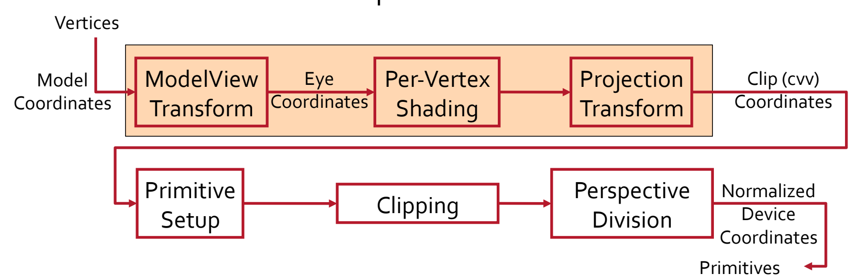


We're not covering
- geometry shader
- tessellation shader

The Vertex Shader

Vertex shader replaces
- vertex transformation
- normal transformation, normalization
- lighting
- texture coordinate generation and transformation

GLSL 1.2 has access to OpenGL states



Vertex Shader

Input: individual vertex in model coordinates

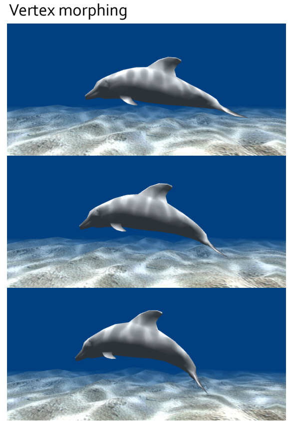
Output: individual vertex in clip (cvv) coordinates

Operate on individual vertex
- results generated from one vertex cannot be shared with other vertices

Cannot create or destroy vertex

Must do:
- transforms
- lighting
- (GLSL 1.2: matrices and lights are provided)

Vertex morphing



Why Use Vertex Shader?

Complete control of transform and lighting hardware

Custom vertex lighting

Custom vertex computations
- custom skinning and blending
- object/character deformation
- procedural deformation

Custom texture coordinate generation

Custom texture matrix operations

Complex vertex operations accelerated in hardware

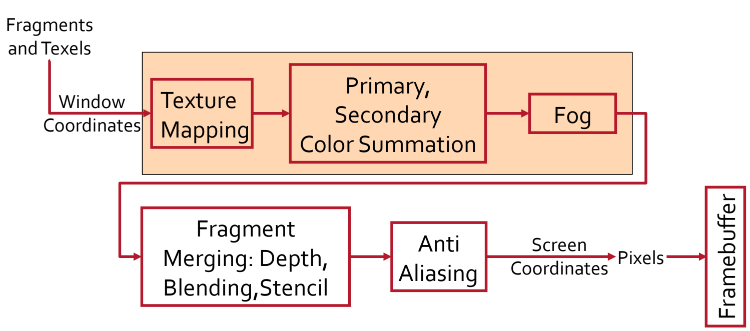
Offloading vertex computations frees up CPU
- more physics and simulation possible!
- particle systems

Zhu

The Fragment/Pixel Shader

Fragment shader replaces
- texture accesses & application
- fog and some fragment tests

GLSL 1.2 has access to OpenGL states



Fragment/Pixel Shader

Input: individual fragment in window coordinates

Output: individual fragment in window coordinates

Operate on individual fragment
- results generated from one fragment cannot be shared with other fragments
  - however, can use gradient computation (`dFdx()`, `dFdy()`) to derive how a value changes per pixel along the x and y screen coordinates

Texture coordinates
- store information for lookup in textures
  - more general than images-to-be-glued, e.g., bump mapping

What Can You Do with A Fragment Shader?

Per-pixel lighting

- looks much better than per-vertex lighting
  - true Phong shading
- per-pixel Fresnel term and Cook-Torrance lighting
- anisotropic lighting
- non-photorealistic rendering (NPR)
  - cartoon shading, hatching, Gooch lighting, image space techniques

Volumetric effects

Advanced bump mapping

Procedural textures and texture perturbation

And more …

Shader Programming Model

The GPU is a stream processor

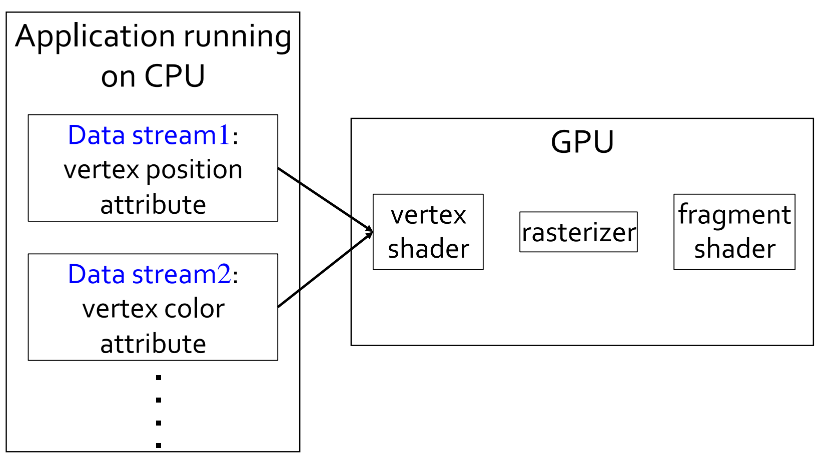
Each point in 3D space has one or more attributes defining it: position, surface normal, color, texture coordinates, etc.

Each attribute forms a stream of data that are fed to the shaders one after another

From the application to the vertex shader, each data stream contains the values of the same attribute (such as position) across vertices

The vertex shader is applied to each vertex of each graphics primitive in a stateless manner

Vertex Attribute Streams



Shader Programming Model

The rasterizer:

- does scan conversion: figures out which fragments are covered by the primitive, and
- interpolates an attribute across vertices to compute the corresponding per-fragment values

Data streams arriving at the fragment shader contain per-fragment attribute values (a much larger set than per-vertex attribute streams!)

The fragment shader is applied to each fragment covered by a graphics primitive in a stateless manner

Shader Programming: Outline

- Basic GLSL 1.2 syntax
- Vertex shader, with example
- Fragment shader, with vertex+fragment shaders example
- GLSL 1.2 syntax for vectors and matrices
- GLSL 1.2 built-in functions
- Data passing in GLSL 1.2
- Example 3: vertex to fragment shader data passing
- Built-in GLSL 1.2 global variables
- Example 4: built-in global variables usage
- Integrating GLSL with OpenGL

GLSL

Comes with OpenGL

Based on C, with some C++ features

Restricted programming model, to allow for transparent parallelization, threading, and load balancing

Graphics-friendly data types:

```
void, bool, int, float, // no double
vec2, vec3, vec4, // default to float
ivec2, bvec2, [ib]vec[34],
mat2, mat3, mat4, // square matrices only
structs, 1D arrays, functions
```

Control Flow

C-like expression for execution control:

- `if (bool) … else …`
- `for (i = 0; i< n; i++) loop`
- `do … while (bool)`

However, these conditional branching is much more expensive than in CPU → don't use too much of it, especially in fragment shader

THINK PARALLEL!

Same code applies to all vertices/fragments

Shader Code Snippet

```
void main() {
   const float f = 3.0;
   vec3 u(1.0), v(0.0, 1.0, 0.0);

   for (int i=0; i<10; i++)
      v = f * u + v;
   …
}
```

Seems like general purpose computing

- what's missing?

Missing Features

no pointers, no dynamically allocated memory

no recursion

no strings, no characters

no double (up to 4.0), byte, short, long, ...

no unsigned in 1.2, `uint` and `uvec[2-4]` in GLSL 1.3+

no file I/O

no `printf()`

focus is on (parallel) numerical computation

specify version by `#version n`
$n = \{110, 120, 130, 140, 150, 330, 400, 410, 420, 430, 440, 450\}$

Strong Typing

No automatic type conversion

```
float f = 1;    // WRONG
float f = 1.0; // much better
```

Instead of casting, use constructors:

```
vec3 rgb = {1.0, 1.0, 1.0 };
// C++ style constructor for type conversion:
vec4 rgba = vec4(rgb, 1.0);
vec2 rg = vec2(rgba);     // and masking
```

Example 1: Minimal Vertex Shader

What a vertex shader must minimally do
(what the rasterizer expects):

- transform vertex position from model to eye coordinates
- and then project to clip (cvv) coordinates
- finally, output the vertex position
- (GLSL 1.2: compute vertex lighting if `GL_LIGHTING` is on)

```
void
main(void)
{
  gl_Position = gl_ModelViewProjectionMatrix *
                gl_Vertex;
}
```

- all `gl_*` variables above are part of the OpenGL state, which GLSL 1.2 shader can access without declaring

Example 2: From Vertex Shader ...

OpenGL application code:

```
glColor3f(0.0,0.0,1.0);
glBegin(GL_TRIANGLES);
    glVertex3i(1, 0, 0);
    glVertex3i(0, 1, 0);
    glVertex3i(0, 0, 1);
glEnd();
```

with the following vertex shader:

```
void
main(void)
{
  gl_Position = gl_ModelViewProjectionMatrix *
                gl_Vertex;
  gl_FrontColor = gl_Color;
}
```

would color the vertices blue

Example 2: to the Rasterizer …

The rasterizer interpolates vertex attributes, such as positions and colors, across a primitive and generate the corresponding attributes for each fragment forming the primitive

Example 2: to the Fragment Shader

In the following fragment shader:

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

each pixel is colored blue

- the `gl_Color` here is not set by the OpenGL app, but is the per-fragment color interpolated by the rasterizer!
- (though in this example, they would have been the same)

Vector Components

Vector components can be accessed by:

- position (*xyzw*), color (*rgba*), texture-coordinates (*stpq*)
  - these are syntactic sugar only
  - they can't be mixed in a single selection
- or plain index: `a[i]`

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

v2.x    // returns a float
v2.z    // wrong: undefined for type
v4.rgba // returns a vec4
v4.stp  // returns a vec3
v4.b    // returns a float
v4.xy   // returns a vec2
v4.xgp  // wrong: mismatched component sets
```

Swizzling & Smearing

Swizzling operator lets you access any particular component(s) of a vector

- swizzle as R-values:

```
vec2 v2;
vec4 v4;

v4.wzyx // swizzles, returns a vec4
v4.bgra // swizzles, returns a vec4
v4.xxxx // smears x, returns a vec4
v4.xxx  // smears x, returns a vec3
v4.yyxx // swizzles and smears x and y,
        // returns a vec4
v2.yyyy // wrong: too many components for type
v2.xy = v2.yx // swaps components
```

Swizzling

- swizzle as L-values

```
vec4 v4 = vec4(1.0, 2.0, 3.0, 4.0);

v4.xw = vec2(5.0, 6.0); // (5.0, 2.0, 3.0, 6.0)
v4.wx = vec2(7.0, 8.0); // (8.0, 2.0, 3.0, 7.0)
v4.xx = vec2(9.0,10.0); // wrong: x used twice
v4.yz = 11.0;           // wrong: type mismatch
v4.yz = vec2(12.0);     // (8.0, 12.0, 12.0, 7.0)
```

Example: compute cross product $\mathbf{n} = \mathbf{u} \times \mathbf{v}$

$$\begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} \times \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{bmatrix}$$

Advantage: avoid intermediates and copies

```
n = u.yzx * v.zxy - u.zxy * v.yzx;
```

Matrix Components

Matrices are created, stored, and accessed in column major order

- `M[i]`: column `i` of `M`, as in C/C++, starts from 0
- `M[i][j]`: element in col `i`, row `j`

The `*` operator in GLSL is overloaded, hence:

```
mat4 M;
vec4 u,v,w;

u = M*v;
w = v*M;
```

work and compute the right, and different, results for `u` and `w`

Matrix Constructors

```
vec2 v1, v2;
mat2 m2;
mat4 m4;

mat4(1.0, 2.0, 3.0, 4.0,    // first column
     5.0, 6.0, 7.0, 8.0,    // second column
     9.0, 10., 11., 12.,    // third column
     13., 14., 15., 16.);   // fourth column

v1 = vec2(1.0,2.0);
v2 = vec2(3.0,4.0);
m2 = mat2(v1,v2);           // 1st col: 1.0, 2.0,
                            // 2nd col: 3.0, 4.0

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

Built-in Functions

Common

```
• abs, floor, ceil, min, max, mod(dividend, divisor),
  sign(x)  // 1.0 if x > 1, 0.0 if x = 0, else -1.0
  fract(x) // x - floor(x)
  clamp(x,low,high) // min(max(x, low), high)
```

Exponentials

```
• pow, exp2, log2, sqrt, inversesqrt (1/sqrt)
```

Angles & trigonometry

```
• radians, degrees, sin, cos, tan, asin, acos, atan
```

Interpolations

```
• mix(x,y, a)             // (1.0-a)x + ay, D3D: lerp(x,y,a)
• step(edge, x)           // x < edge ? 0.0 : 1.0
• smoothstep(edge0,edge1,x) // Hermite interpolation
                          // t = (x-edge0)/(edge1-edge0);
                          // t = clamp( t, 0.0, 1.0);
                          // return t*t*(3.0-2.0*t);
```

See APIs Table (http://web.eecs.umich.edu/~sugih/courses/eecs487/common/notes/APITables.xml)

Built-in Functions

Geometric
- `length (of vector)`, `distance (between 2 points)`, **`cross`**, **`dot`**, **`normalize`**, `faceForward`, `reflect (about normal)`

Matrix
- `matrixCompMult`

Vector relational
- `lessThan`, `lessThanEqual`, `greaterThan`, `greaterThanEqual`, `equal`, `notEqual`, `any`, `all`

Texture
- `texture1D`, `texture2D`, `texture3D`, `textureCube`
- `texture1DProj`, `texture2DProj`, `texture3DProj`, `textureCubeProj`
- `shadow1D`, `shadow2D`, `shadow1DProj`, `shadow2Dproj`

See APIs Table (http://web.eecs.umich.edu/~sugih/courses/eecs487/common/notes/APITables.xml)

Call-by Value-Return

Function parameters:
- `in`: copy in [default]
- `out`: copy out, undefined upon entrance
- `inout`: copy both

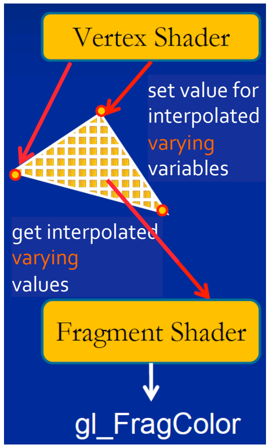
(no pointers or references)

Data Passing with Global Variables

OpenGL application passes data to the vertex shader using two types of GLSL 1.2 global variables: `uniform` and `attribute`

The vertex shader passes data to the fragment shader by having the rasterizer interpolate `attribute`s between vertices into per-fragment `varying` variables

The fragment shader can also access `uniform` global variables set by the OpenGL application



Fengo8

- both `attribute` and `varying` are deprecated in GLSL 1.3, use `in` and `out` instead
- these keywords have different meanings in Direct3D

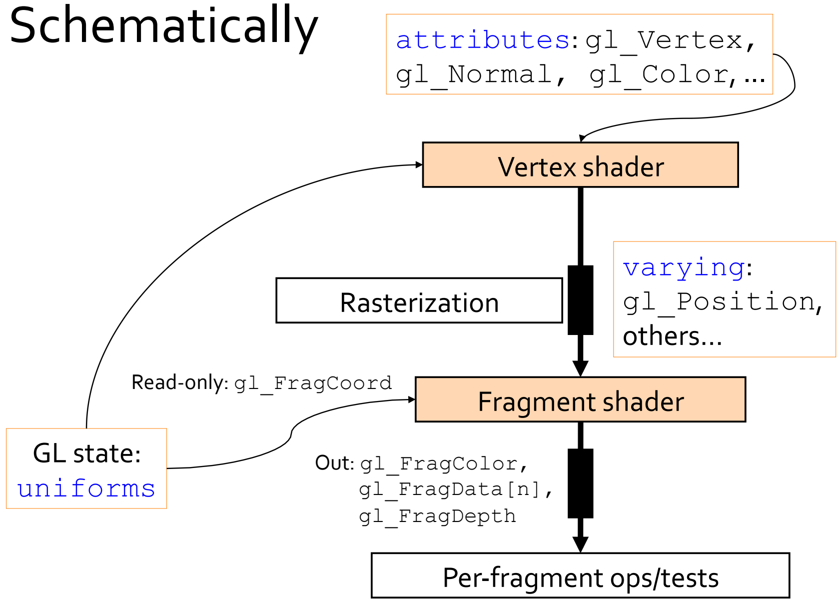
Global Variables: Type Qualifiers

`uniform`:
- set by app, can be changed at program run time, but constant across each execution of a shader and cannot be changed in the shader
- value constant across a set of primitives in a `glBegin`, `glEnd` block
- e.g., transformation and texture matrices, light position and direction

`attribute` (GLSL 1.3+: `in` in vs):
- per-vertex data, values vary per-vertex, e.g., normal, texture coordinates
- can be built-in, i.e., OpenGL state variables, which doesn't need to be declared (e.g., `gl_Vertex`), or
- can be application-specific (e.g., temperature per vertex, previous vertex value for morphing) , which must be `float` (GLSL 1.3+ allows `int`)
- cannot be changed inside shaders

`varying` (GLSL 1.3+: `out` in vs, `in` in fs):
- output from a vertex shader; input to a fragment shader
- declared and assigned per vertex, interpolated across a primitive, resulting in per-fragment value

Schematically



Example 3: Data Passing from the Vertex Shader …

Vertex shader part:

```
varying vec3 N;  // per-fragment normal

void main()
{
  // output vertex position in clip coordinates
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  // compute the vertex normal in eye coordinates:
  N = gl_NormalMatrix * gl_Normal;
}
```

Example 3: … to the Fragment Shader

Fragment shader part:

```
varying vec3 N;  // per-fragment normal

// Takes the normal (interpolated per pixel) and
// simply turns it into a color

void main()
{
   // N is interpolated across the triangle,
   // so normalize it:
   vec3 n = normalize(N);

   gl_FragColor = vec4(abs(n),1.0);
}
```

Built-in Global Variables

GLSL 1.2: OpenGL states are mapped to `uniform` and `attribute` globals for ease of programming

Some special variables must be updated, others are optional:

- vertex shader

```
vec4  gl_Position;      // must be written
vec4  gl_ClipPosition;  // may be written
float gl_PointSize;     // may be written
```

- fragment shader

```
vec4  gl_FragCoord;     // may be read
bool  gl_FrontFacing;   // may be read
float gl_FragDepth;     // may be read/written
float gl_FragColor;     // may be written
```

See APIs Table (http://web.eecs.umich.edu/~sugih/courses/eecs487/common/notes/APITables.xml)

Built-in attributes

```
attribute vec4  gl_Vertex;
attribute vec3  gl_Normal;
attribute vec4  gl_Color;
attribute vec4  gl_SecondaryColor;
attribute vec4  gl_MultiTexCoordn;
attribute float gl_FogCoord;
```


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;
```

See APIs Table (http://web.eecs.umich.edu/~sugih/courses/eecs487/common/notes/APITables.xml)

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 gl_FrontLightProduct[g].ambient;  // etc.
```

See APIs Table (http://web.eecs.umich.edu/~sugih/courses/eecs487/common/notes/APITables.xml)

Built-in varyings

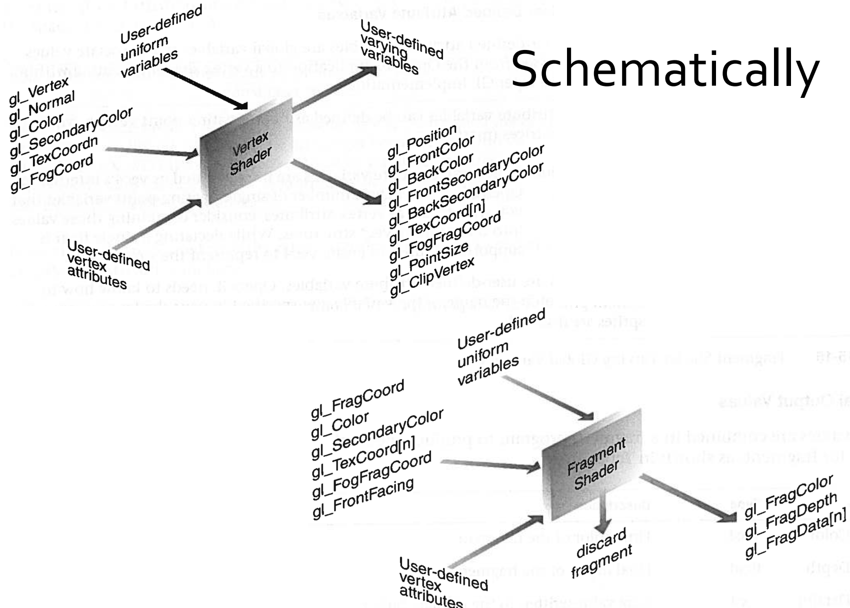
```
varying  vec4  gl_FrontColor      // vertex
varying  vec4  gl_BackColor;      // vertex
varying  vec4  gl_FrontSecColor;  // vertex
varying  vec4  gl_BackSecColor;   // vertex

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

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

See APIs Table (http://web.eecs.umich.edu/~sugih/courses/eecs487/common/notes/APITables.xml)

Schematically



See APIs Table (http://web.eecs.umich.edu/~sugih/courses/eecs487/common/notes/APITables.xml)

Example 4: Ivory – Vertex Shader

```
uniform vec4 lightPos; // not using built-in!

varying vec3 normal;
varying vec3 lightVec;
varying vec3 viewVec;

void main()
{
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  vec4 vert = gl_ModelViewMatrix * gl_Vertex; // eye coord

  normal   = gl_NormalMatrix * gl_Normal;
  lightVec = vec3(lightPos - vert);
  viewVec  = -vec3(vert);
}
```

Example 4: Ivory – Fragment Shader

```
varying vec3 normal;
varying vec3 lightVec;
varying vec3 viewVec;

void main()
{
  vec3 N = normalize(normal);
  vec3 L = normalize(lightVec);
  vec3 V = normalize(viewVec);
  vec3 H = normalize(L + V);

  float NdotL = dot(N, L);
  float NdotH = clamp(dot(N, H), 0.0, 1.0);

  // "Half-Lambert" technique for more pleasing diffuse term
  float diffuse  = 0.5 * NdotL + 0.5;  // what's this?
  float specular = pow(NdotH, 64.0);

  float result = diffuse + specular;

  gl_FragColor = vec4(result);
}
```