

# The Graphics Pipeline and OpenGL III: OpenGL Shading Language (GLSL 1.10)

Gordon Wetzstein

Stanford University

EE 267 Virtual Reality

Lecture 4

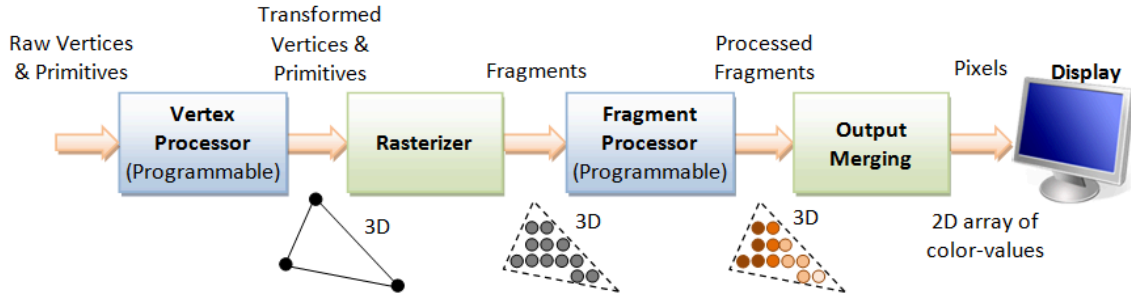
[stanford.edu/class/ee267/](http://stanford.edu/class/ee267/)



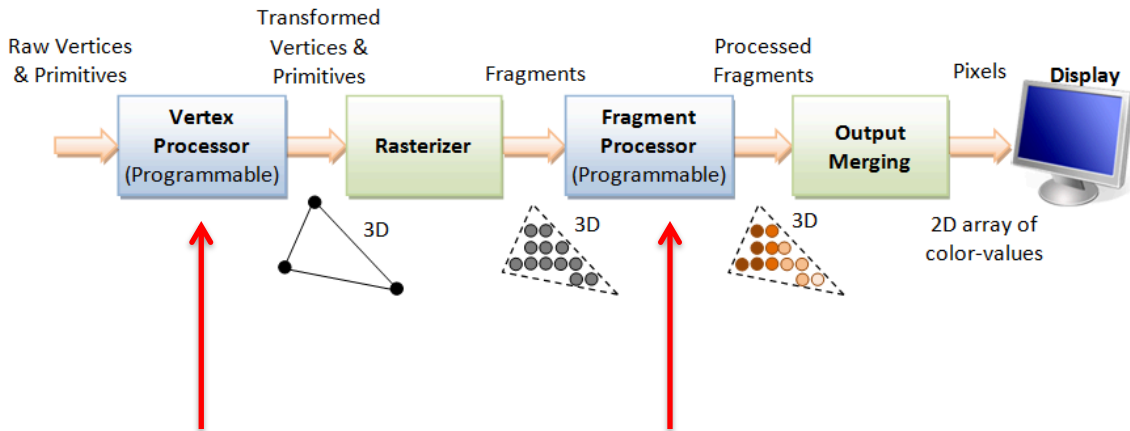
# Lecture Overview

- Review of graphics pipeline
- vertex and fragment shaders
- OpenGL Shading Language (GLSL 1.10)
- Implementing lighting & shading with GLSL vertex and fragment shaders

# Reminder: The Graphics Pipeline



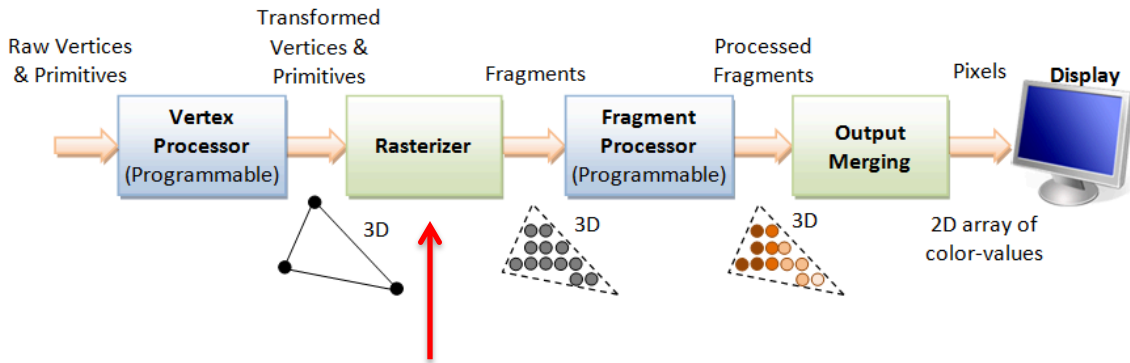
# Reminder: The Graphics Pipeline



- transforms
- (per-vertex) lighting
- ...

- texturing
- (per-fragment) lighting
- ...

# Reminder: The Graphics Pipeline

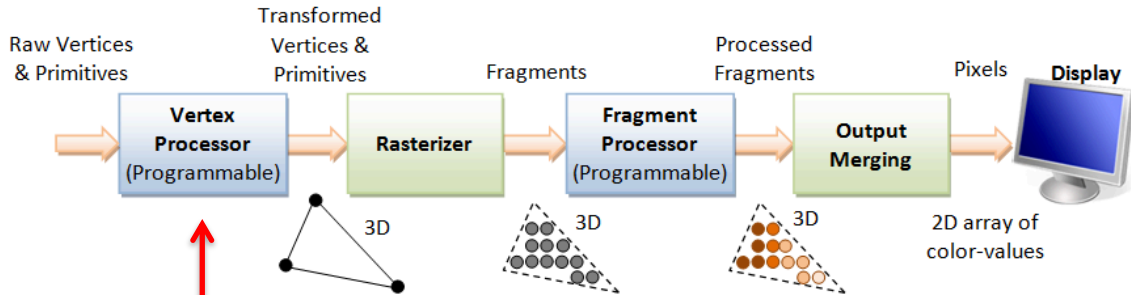


## The Rasterizer

Two goals:

1. determine which fragments are inside the primitives (triangles) and which ones aren't
2. interpolate per-vertex attributes (color, texture coordinates, normals, ...) to each fragment in the primitive

# Vertex Shaders



input

vertex shader (executed for each vertex in parallel)

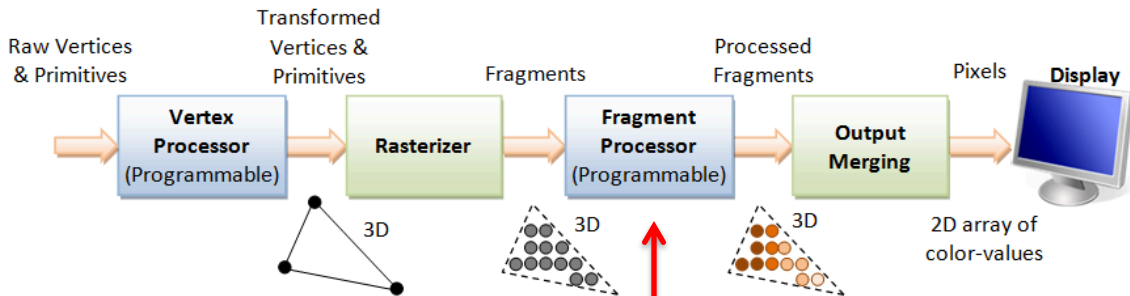
output

- vertex position, normal, color, material, texture coordinates
- modelview matrix, projection matrix, normal matrix
- ...

```
void main ()  
{  
    // do something here  
    ...  
}
```

- transformed vertex position (in clip coords), texture coordinates
- ...

# Fragment Shaders

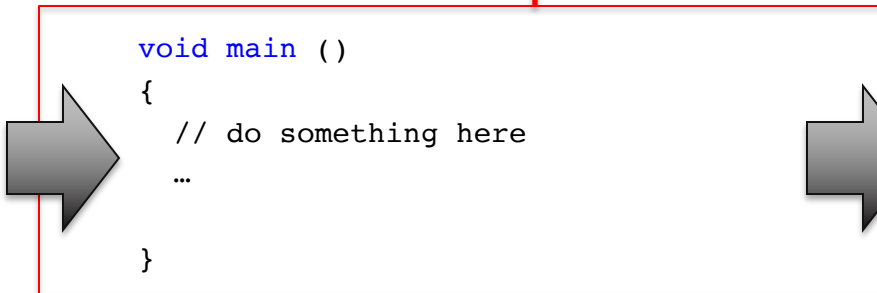


input

fragment shader (executed for each fragment in parallel)

output

- vertex position in window coords, texture coordinates
- ...



- fragment color
- fragment depth
- ...

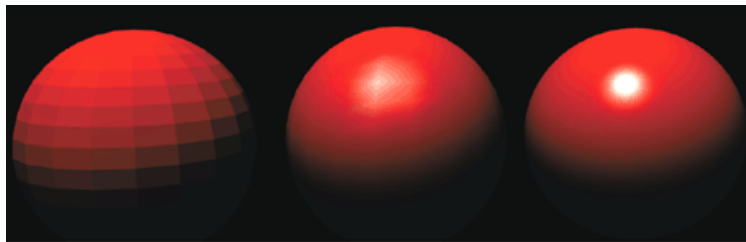
# Why Do We Need Shaders?

- massively parallel computing
- single instruction multiple data (SIMD) paradigm → GPUs are designed to be parallel processors
- vertex shaders are independently executed for each vertex on GPU (in parallel)
- fragment shaders are independently executed for each fragment on GPU (in parallel)



# Why Do We Need Shaders?

- most important: vertex transforms and lighting & shading calculations
- shading: how to compute color of each fragment (e.g. interpolate colors)
  1. Flat shading
  2. Gouraud shading (per-vertex lighting)
  3. Phong shading (per-fragment lighting)
- other: render motion blur, depth of field, physical simulation, ...



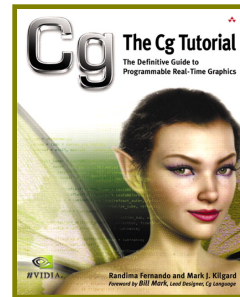
Flat

Gouraud

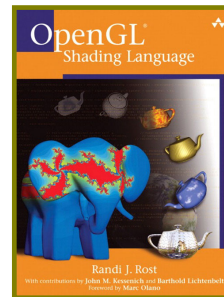
Phong

# Shading Languages

- Cg (C for Graphics – NVIDIA, deprecated)

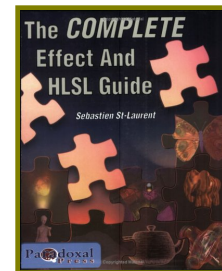


- GLSL (GL Shading Language – OpenGL)



EE 267

- HLSL (High Level Shading Language - MS Direct3D)



# Demo – Simple Vertex Shader



```
// variable passed in from JavaScript / three.js
uniform float deformation;

attribute vec3 position;
attribute vec3 normal;

uniform mat4 projectionMat;
uniform mat4 modelViewMat;

void main () // vertex shader
{
    // deform vertex position in object coordinates
    vec3 pos = position + deformation * normal;

    // convert to clip space
    gl_Position = projectionMat*modelViewMat*vec4(pos,1.0);

    // do lighting calculations here (in view space)
    ...
}
```

# Demo – Simple Fragment Shader



```
// variables passed in from JavaScript / three.js
varying vec2 textureCoords;

uniform sampler2D texture;
uniform float gamma;

void main () // fragment shader
{

    // texture lookup
    vec3 textureColor = texture2D(texture, textureCoords).rgb;

    // set output color by applying gamma
    gl_FragColor.rgb = pow(textureColor.rgb, gamma);

}
```

# Vertex+Fragment Shader – Gouraud Shading Template



vertex shader

```
// variable to be passed from vertex to fragment shader
varying vec4 myColor;

// variable passed in from JavaScript / three.js
uniform mat4 projectionMat;
uniform mat4 modelViewMat;
uniform mat3 normalMat;

attribute vec3 position;
attribute vec3 normal;

void main () // vertex shader – Gouraud shading
{
    // transform position to clip space
    gl_Position = projectionMat * modelViewMat * vec4(position,1.0);

    // transform position to view space
    vec4 positionView = modelViewMat * vec4(position,1.0);

    // transform normal into view space
    vec3 normalView = normalMat * normal;

    // do lighting calculations here (in view space)
    ...
    myColor = ...
}
```

fragment  
shader

```
// variable to be passed from vertex to fragment shader
varying vec4 myColor;
void main () // fragment shader – Gouraud shading
{ gl_FragColor = myColor; }
```

# Vertex+Fragment Shader – Phong Shading Template



vertex shader

```
// variable to be passed from vertex to fragment shader
varying vec4 myPos;
varying vec3 myNormal;

// variable passed in from JavaScript / three.js
uniform mat4 projectionMat;
uniform mat4 modelViewMat;
uniform mat3 normalMat;

attribute vec3 position;
attribute vec3 normal;

void main () // vertex shader – Phong shading
{
    // transform position to clip space
    gl_Position = projectionMat * modelViewMat * vec4(position,1.0);

    // transform position to view space
    myPos = modelViewMat * vec4(position,1.0);

    // transform normal into view space
    myNormal = normalMat * normal;
}
```

fragment  
shader

```
// variable to be passed from vertex to fragment shader
varying vec4 myPos;
varying vec3 myNormal;
void main () // fragment shader – Phong shading
{ // ... do lighting calculations here ...
    gl_FragColor = ...;
}
```

# Demo – General Purpose Computation Shader: Heat Equation



```
varying vec2 textureCoords;

// variables passed in from JavaScript / three.js
uniform sampler2D tex;
const float timestep = 1.0;

void main () // fragment shader
{

    // texture lookups
    float u = texture2D(tex, textureCoords).r;

    float u_xp1 = texture2D(tex, float2(textureCoords.x+1,textureCoords.y)).r;
    float u_xm1 = texture2D(tex, float2(textureCoords.x-1,textureCoords.y)).r;
    float u_yp1 = texture2D(tex, float2(textureCoords.x,textureCoords.y+1)).r;
    float u_ym1 = texture2D(tex, float2(textureCoords.x,textureCoords.y-1)).r;

    glFragColor.r = u + timestep*(u_xp1+u_xm1+u_yp1+u_ym1-4*u);

}
```

heat equation:  $\frac{\partial u}{\partial t} = \alpha \nabla^2 u \Rightarrow u^{(t+1)} = \Delta_t \alpha \nabla^2 u + u^{(t)}$

# OpenGL Shading Language (GLSL)

- high-level programming language for shaders
- syntax similar to C (i.e. has `main` function and many other similarities)
- usually very short programs that are executed in parallel on GPU
- good introduction / tutorial:

<https://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/>



# OpenGL Shading Language (GLSL)

- versions of OpenGL, WebGL, GLSL can get confusing
- here's what we use:
  - WebGL 1.0 - based on OpenGL ES 2.0; cheat sheet:  
[https://www.khronos.org/files/webgl/webgl-reference-card-1\\_0.pdf](https://www.khronos.org/files/webgl/webgl-reference-card-1_0.pdf)
  - GLSL 1.10 - shader preprocessor: `#version 110`
- reason: three.js doesn't support WebGL 2.0 yet

# GLSL – Vertex Shader Input/Output

input

input variables are either **uniform** (passed in from JavaScript, e.g. matrices) or **attribute** (values associated with each vertex, e.g. position, normal, uv, ...)

built-in  
output

```
vec4 gl_Position
```

vertex position in clip coordinates

# GLSL – Fragment Shader Input/Output

input

input variables are either **uniform** (passed in from JavaScript) or **varying** (passed in from vertex shader through rasterizer)

built-in  
output

**vec4 gl\_FragColor**

fragment color

# GLSL – Fragment Shader Input/Output

input

input variables are either **uniform** (passed in from JavaScript) or **varying** (passed in from vertex shader through rasterizer)

built-in  
output

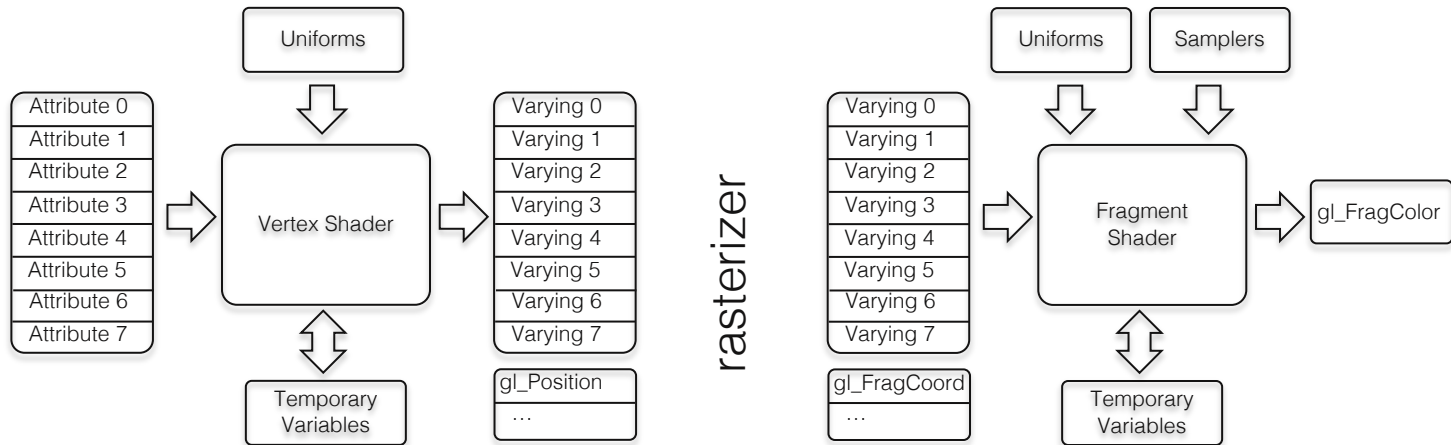
**vec4 gl\_FragColor**

fragment color

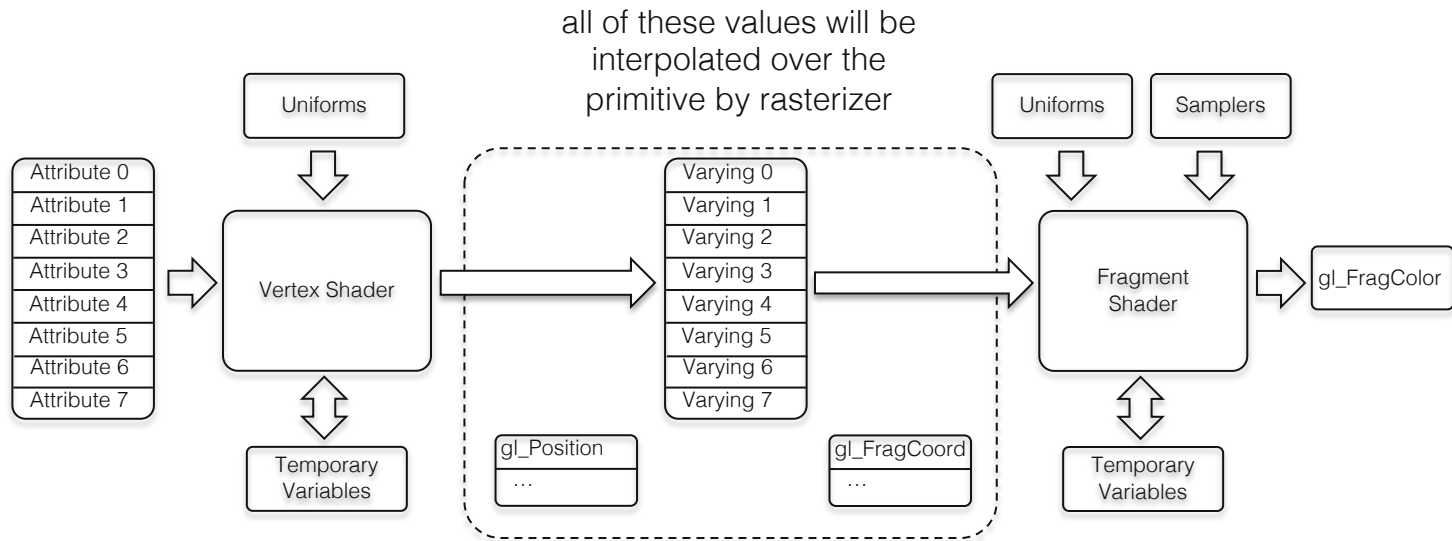
**float gl\_FragDepth**

value written to depth buffer, if not specified: gl\_FragCoord.z (only available in OpenGL but not WebGL)

# GLSL Shader



# GLSL Shader



# GLSL Data Types

<code>bool</code>	– boolean (true or false)
<code>int</code>	– signed integer
<code>float</code>	– 32 bit floating point
<code>ivec2, ivec3, ivec4</code>	– integer vector with 2, 3, or 4 elements
<code>vec2, vec3, vec4</code>	– floating point vector with 2, 3, or 4 elements
<code>mat2, mat3, mat4</code>	– floating point matrix with 2x2, 3x3, or 4x4 elements
<code>sampler2D</code>	– handle to a 2D texture
<code>attribute</code>	– per-vertex attribute, such as position, normal, uv, ...

# GLSL Data Types

## uniform type

- read-only values passed in from JavaScript,  
e.g. `uniform float` or `uniform sampler2D`

## vertex shader

```
uniform mat4 modelViewProjectionMatrix;  
  
varying vec2 textureCoords;  
  
attribute vec3 position;  
attribute vec2 uv;  
  
void main ()  
{  
    gl_Position = modelViewProjectionMatrix * vec4(position, 1.0);  
    textureCoords = uv;  
}
```

## fragment shader

```
uniform sampler2D texture;  
  
varying vec2 textureCoords;  
  
void main ()  
{  
    gl_FragColor = texture2D(texture, textureCoords);  
}
```



# GLSL Data Types

## varying type

- variables that are passed from vertex to fragment shader (i.e. write-only in vertex shader, read-only in fragment shader)
- rasterizer interpolates these values in between shaders!

### vertex shader

```
varying float myValue;
```

```
uniform mat4 modelViewProjectionMatrix;  
attribute vec3 position;
```

```
void main ()  
{  
  
    gl_Position = modelViewProjectionMatrix * vec4(position,1.0);  
  
    myValue = 3.14159 / 10.0;  
  
}
```

### fragment shader

```
varying float myValue;
```

```
void main ()  
{  
  
    gl_FragColor = vec4(myValue, myValue, myValue, 1.0);  
  
}
```

# GLSL – Simplest (pass-through) Vertex Shader

```
// variable passed in from JavaScript / three.js
uniform mat4 modelViewProjectionMatrix;

// vertex positions are parsed as attributes
attribute vec3 position;

void main () // vertex shader
{
    // transform position to clip space
    // this is similar to gl_Position = ftransform();
    gl_Position = modelViewProjectionMatrix * vec4(position,1.0);
}
```

# GLSL – Simplest Fragment Shader

```
void main () // fragment shader
{
    // set same color for each fragment
    gl_FragColor = vec4(1.0,0.0,0.0,1.0);
}
```

# GLSL – built-in functions

<code>dot</code>	dot product between two vectors
<code>cross</code>	cross product between two vectors
<code>texture2D</code>	texture lookup (get color value of texture at some tex coords)
<code>normalize</code>	normalize a vector
<code>clamp</code>	clamp a scalar to some range (e.g., 0 to 1)

`radians, degrees, sin, cos, tan, asin, acos, atan, pow, exp, log, exp2, log2, sqrt, abs, sign, floor, ceil, mod, min, max, length, ...`

good summary of OpenGL ES (WebGL) shader functions:

[https://www.khronos.org/files/webgl/webgl-reference-card-1\\_0.pdf](https://www.khronos.org/files/webgl/webgl-reference-card-1_0.pdf)

# Gouraud Shading with GLSL (only diffuse part) – Vertex Shader

```
uniform vec3 lightPositionView;  
uniform vec3 lightColor;  
uniform vec3 diffuseMaterial;
```



user-defined light & material properties

```
uniform mat4 projectionMat;  
uniform mat4 modelViewMat;  
uniform mat3 normalMat;
```



user-defined transformation matrices

```
attribute vec3 position;  
attribute vec3 normal;
```







per-vertex attributes

```
varying vec3 vColor;
```



color computed by Phong lighting model to  
be interpolated by rasterizer

# Gouraud Shading with GLSL (only diffuse part) – Vertex Shader

<pre>uniform vec3 lightPositionView; uniform vec3 lightColor; uniform vec3 diffuseMaterial;</pre>		user-defined light & material properties
<pre>uniform mat4 projectionMat; uniform mat4 modelViewMat; uniform mat3 normalMat;</pre>		user-defined transformation matrices
<pre>attribute vec3 position; attribute vec3 normal;</pre>		per-vertex attributes
<pre>varying vec3 vColor;</pre>		color computed by Phong lighting model to be interpolated by rasterizer

```
void main () // vertex shader  
{  
    // transform position to clip space  
    gl_Position = projectionMat * modelViewMat * vec4(position,1.0);  
  
    // transform vertex position, normal, and light position to view space  
    vec3 P = ...  
    vec3 L = ...  
    vec3 N = ...  
  
    // compute the diffuse term here  
    float diffuseFactor = ...  
  
    // set output color  
    vColor = diffuseFactor * diffuseMaterial * lightColor;  
}
```

# Gouraud Shading with GLSL (only diffuse part) – Fragment Shader

```
varying vec3 vColor;

void main () // fragment shader
{
    // set output color
    gl_FragColor = vec4(vColor,1.0);
}
```

# Phong Shading with GLSL (only diffuse part) – Vertex Shader

```
uniform mat4 modelViewMat;  
uniform mat4 projectionMat;  
uniform mat3 normalMat;
```



user-defined transformation matrices

```
attribute vec3 position;  
attribute vec3 normal;
```



per-vertex attributes

```
varying vec3 vPosition;  
varying vec3 vNormal;
```



vertex position & normal to be interpolated  
by rasterizer



# Phong Shading with GLSL (only diffuse part) – Vertex Shader

```
uniform mat4 modelViewMat;  
uniform mat4 projectionMat;  
uniform mat3 normalMat;
```



user-defined transformation matrices

```
attribute vec3 position;  
attribute vec3 normal;
```



per-vertex attributes

```
varying vec3 vPosition;  
varying vec3 vNormal;
```



vertex position & normal to be interpolated  
by rasterizer

```
void main () // vertex shader  
{  
    // transform position to clip space  
    gl_Position = projectionMat * modelViewMat * vec4(position,1.0);  
  
    // transform vertex position, normal, and light position to view space  
    vec3 P = ...  
    vec3 N = ...  
  
    // set output texture coordinate to vertex position in world coords  
    vPosition = P;  
  
    // set output color to vertex normal direction  
    vNormal = N;  
}
```

# Phong Shading with GLSL (only diffuse part) – Fragment Shader

```
uniform vec3 lightColor;  
uniform vec3 diffuseMaterial;  
uniform vec3 lightPositionWorld;
```



user-defined light & material properties

```
varying vec3 vPosition;  
varying vec3 vNormal;
```



vertex & normal positions interpolated to  
each fragment by rasterizer

# Phong Shading with GLSL (only diffuse part) – Fragment Shader

```
uniform vec3 lightColor;  
uniform vec3 diffuseMaterial;  
uniform vec3 lightPositionWorld;
```



user-defined light & material properties

```
varying vec3 vPosition;  
varying vec3 vNormal;
```



vertex & normal positions interpolated to each fragment by rasterizer

```
void main () // fragment shader  
{  
    // incoming color is interpolated by rasterizer over primitives!  
    vec3 N = vNormal;  
  
    // vector pointing to light source  
    vec3 L = ...  
  
    // compute the diffuse term  
    float diffuseFactor ...  
  
    // set output color  
    gl_FragColor.rgb = diffuseFactor * diffuseMaterial * lightColor;  
}
```

# GLSL - Misc

- swizzling: `vec4 myVector1;`  
`vec4 myVector2;`  
`vec3 myVector1.xxy + myVector2.zxy;`

- matrices are column-major ordering

- initialize vectors in any of the following ways:

```
vec4 myVector  = vec4(1.0, 2.0, 3.0, 4.0);  
vec4 myVector2 = vec4(vec2(1.0, 2.0), 3.0, 4.0);  
vec4 myVector3 = vec4(vec3(1.0, 2.0, 3.0), 4.0);
```

- these are equivalent: `myVector.xyzw = myVector.rgba`
- we omitted a lot of details...

# JavaScript & GLSL

goals:

- loading, compiling, and linking GLSL shaders (from a file) using JavaScript
- activating and deactivate GLSL shaders in JavaScript
- accessing uniforms from JavaScript

our approach (for labs and homeworks):

- use three.js to handle all of the above
- can do manually, but more work – we will shield this from you

# Summary

- GLSL is your language for writing vertex and fragment shaders
- each shader is independently executed for each vertex/fragment on the GPU
- usually require both vertex and fragment shader, but can “pass-through” data

# Further Reading

- GLSL tutorial: <https://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/>
- summary of built-in GLSL functions: <http://www.shaderific.com/glsl-functions/>
- GLSL and WebGL: <https://webglfundamentals.org/webgl/lessons/webgl-shaders-and-glsl.html>