An Introduction to the OpenGL Shading Language
Benj Lipchak
Rob Simpson
Bill Licea-Kane

Outline
- A little history (backstory)
- How the fixed function pipeline works
- How it’s replaced by GLSL
- Structure & syntax nitty-gritty
- How to integrate GLSL into OpenGL apps
- Some simple examples
- Resources

Traditional Graphics Pipeline

A simplified graphics pipeline
- Note that pipe widths vary
- Many caches, FIFOs, and so on not shown
## Fixed Functionality Pipeline

- **API**
- **Transform** and **Lighting**
- **Rasterizer**
- **Primitive Assembly**
- **Vertex Buffer**
- **Vertices**
- **Triangles/Lines/Points**
- **Texture Environment**
- **Depth**
- **Stencil**
- **Color**
- **Alpha Test**
- **Fog**
- **Dither**
- **Color Buffer**
- **Blend Buffer**
- **Objects**
- **Frame Buffer**

## Programmable Shader Pipeline

- **API**
- **Vertex Shader**
- **Rasterizer**
- **Primitive Assembly**
- **Vertex Buffer**
- **Vertices**
- **Triangles/Lines/Points**
- **Fragment Shader**
- **Per-Sample Operations**
- **Varyings**
- **Fragment Uniforms**

## Programmer’s Model

- **Vertex Uniforms**
- **Vertex Shader**
- **Attributes**
- **Fragment Uniforms**
- **Fragment Shader**

## Previous Programmability

- **Texture Shaders**
- **Register Combiners**
- **Assembly programs**
  - ARB_vertex_program
  - ARB_fragment_program
- **Messy!**

Needed general, readable & maintainable language

## Using Programmability

**1999-2002: ASM**

```
/* 0.0 */
 attrib vec4 IN;
 uniform float ModelViewProj;
 uniform float ModelViewIT;
 uniform float lightVec;
 uniform float halfVec;
 uniform float3 diffuseMaterial;
 uniform float3 ambientCol;
 uniform float3 specularMaterial;
 uniform float specexp;

void main() {
    vec4 Position = mul(ModelViewProj, IN.xyz);
    float3 normalVec = normalize(mul(ModelViewIT, IN.xyz));
    float diffuse = dot(normalVec, lightVec);
    float spec = dot(normalVec, halfVec);
    float4 lighting = lit(diffuse, spec, specexp);
    OUT.Color.rgb = lighting.y * diffuseMaterial + ambientCol + lighting.z * specularMaterial;
    OUT.Color.a = 1.0
    return OUT;
}
```

**Now: C-like**

```
# attrib vec4 IN;
# uniform float ModelViewProj;
# uniform float ModelViewIT;
# uniform float lightVec;
# uniform float halfVec;
# uniform float3 diffuseMaterial;
# uniform float3 ambientCol;
# uniform float3 specularMaterial;
# uniform float specexp;

void main() {
    vec4 Position = mul(ModelViewProj, IN.xyz);
    float3 normalVec = normalize(mul(ModelViewIT, IN.xyz));
    float diffuse = dot(normalVec, lightVec);
    float spec = dot(normalVec, halfVec);
    float4 lighting = lit(diffuse, spec, specexp);
    OUT.Color.rgb = lighting.y * diffuseMaterial + ambientCol + lighting.z * specularMaterial;
    OUT.Color.a = 1.0
    return OUT;
}
```

## Example Apps.

- **Custom transform, lighting, and skinning**

Example Apps.
Example Apps.

Custom cartoon-style lighting

Example Apps.

- Per-vertex set up for per-pixel bump mapping

Example Apps.

- Character morphing & shadow volume projection

Example Apps.

- Dynamic displacements of surfaces by objects

OpenGL 2.0 Logical Diagram

Vertex Shader Environment
Vertices: What You Don’t Get

Connectivity (neighbor face, edge, vtx)

Can’t Create/Destroy Vertices (coming soon)

Large Writable Memory

Vertices: Expensive (Slow!) Ops

Branches (if, for, while)

Large R/O Memory (textures)

Vertices: Workarounds

Connectivity (neighbor face, edge, vtx)

• Encode neighbor info as attributes

Can’t Create/Destroy Vertices

• Create: start w/ more than you need & specialize
• Destroy: move outside clip volume

Large Writable Memory

• But fragments do (frame buffer)

Vertices: Efficiency

Branches (if, for, while)

• (a<1)?b:c; unroll loops

Large R/O Memory (textures)

• Can put small tables in uniform arrays
Fragment Shader Environment

- Uniforms
- Textures
- Fragment Color(s)
- Varying 0
- Varying 1
- Varying 2
- Varying 3
- Varying 4
- Varying 5
- Window coord
- Front facing flag
- Point coord
- Fragment Depth

Fragment Processor

Flexibility for texturing and per-pixel pixel operations

Fragment processing replaces the following:

- Ops on interpolated values
- Texture access
- Texture application
- Fog
- Color sum
- Pixel Zoom
- Scale and bias
- Color table lookup
- Convolution
- Color matrix

FP does NOT replace

- Shading model
- Dithering
- Coverage
- Plane masking
- Pixel ownership test
- Histogram
- Scissor
- Minmax
- Stipple
- Alpha test
- Pixel packing and unpacking
- Depth test
- Stencil test
- Alpha blending
- Logical ops

Hello World!

```c
void main(void)
{
    // This is our Hello World vertex shader
    // Standard MVP transform
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

```c
void main(void)
{
    // This is our Hello World fragment shader
    // Set to a constant color (hint: look at it upside down)
    gl_FragColor = vec4(0.7734);
}
```

In General...

- Vertex processes bypassed
  - Vertex Transformation
  - Normal Transformation, Normalization
  - Lighting
  - Texture Coordinate Generation and Transformation
- Fragment processes bypassed
  - Texture accesses & application
  - Fog
Types

- void
- float vec2 vec3 vec4
- mat2 mat3 mat4
- int ivec2 ivec3 ivec4
- bool bvec2 bvec3 bvec4
- sampler2D, samplerCube, samplerShadow2D

Type qualifiers

- attribute
  - Changes per-vertex
  - eg. position, normal etc.
- uniform
  - Does not change between vertices of a batch
  - eg light position, texture unit, other constants
- varying
  - Passed from VS to FS, interpolated
  - eg texture coordinates, vertex color

Operators

- binary: * / + -
- relational: < <= > >=
- equality: == !=
- logical: & & ^ ^ | |
- selection: ?: assignment: *= /= += -=

Reserved Operators

- prefix: ~
- binary: %
- bitwise: << >> & ^ |
Scalar/Vector Constructors

No casting

float f; int i; bool b;
vec2 v2; vec3 v3; vec4 v4;
vec2(1.0, 2.0)
vec3(0.0, 0.0, 1.0)
vec4(1.0, 0.5, 0.0, 1.0)
vec4(1.0) // all 1.0
vec4(v2, v2)
vec4(v3, 1.0)
float(i)
int(b)

Matrix Constructors

vec4 v4; mat4 m4;
mat4(1.0, 2.0, 3.0, 4.0,
5.0, 6.0, 7.0, 8.0,
9.0, 10.0, 11.0, 12.0)
// row major
mat4(v4, v4, v4, v4)
mat4(1.0) // identity matrix
mat4(m4) // upper 3x3
vec4(m4) // 1st column
float(m4) // upper 1x1

Accessing components

component accessor for vectors
xyzw  rgba  stpq  [i]

component accessor for matrices
[i]  [i][j]

Vector components

vec2 v2;
vec3 v3;
vec4 v4;
v2.x // is a float
v2.z // wrong: undefined for type
v4.rgba // is a vec4
v4.stp // is a vec3
v4.b // is a float
v4.xy // wrong: mismatched component sets

Assembly Language

Source registers can be negated:

MOV R1, -R2;

before after
R1 x R2 x
R1 x R2 x
R1 x R2 x
R1 x R2 x
R1 x R2 x
R1 x R2 x
R1 x R2 x
R1 x R2 x

Source registers can be “swizzled”:

MOV R1, R2.yzwx;

before after
R1 x R2 x
R1 x R2 x
R1 x R2 x
R1 x R2 x
R1 x R2 x
R1 x R2 x
R1 x R2 x
R1 x R2 x
Assembly Language

Source registers can be negated and “swizzled”:

```
MOV R1, -R2.yzzx;
```

**before**

```
R1
x  y  z  w
a  b  c  d
```

**after**

```
R1
x  y  z  w
b  a  c  d
```

Source registers can be swizzled by “smearing”:

```
MOV R1, R2.w; # alternative to
# using R2.wwww
```

**before**

```
R2
x  y  z  w
7.0 3.0 6.0 2.0
```

**after**

```
R2
x  y  z  w
2.0 2.0 2.0 2.0
```

Swizzling & Smearing

R-values

```
vec2 v2;
vec3 v3;
vec4 v4;
v4.wzyx // swizzles, is a vec4
v4.bgra // swizzles, is a vec4
v4.xxxx // smears x, is a vec4
v4.xxx // smears x, is a vec3
v4.yyxx // duplicates x and y, is a vec4
v2.yyyy // wrong: too many components for type
```

Flow Control

```
expression ? trueExpression : falseExpression
if, if-else
for, while, do-while
return, break, continue
discard (fragment only)
```
Built-in variables

Attributes & uniforms

For ease of programming
OpenGL state mapped to variables

Some special variables are required to be written to, others are optional

Special built-ins

Vertex shader

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

Fragment shader

dint gl_FragColor;     // may be written
float gl_FragDepth;     // may be read/written

// If the shader does not statically write this value, then
// it will take the value of gl_FragCoord.z
vec4 gl_FragCoord;     // may be read
bool gl_FrontFacing;   // may be read
vec2 gl_PointCoord; // may be read

Attributes

Built-in (pre GLSL 3.0) ... now all user defined

attribute vec4 gl_Vertex;
attribute vec3 gl_Normal;
attribute vec4 gl_Color;
attribute vec4 gl_SecondaryColor;
attribute vec4 gl_MultiTexCoord[n];
attribute float gl_FogCoord;

User-defined (Use these for your attributes)

attribute vec3 myTangent;
attribute vec3 myBinormal;
Etc...

Built-in Uniforms (pre GLSL 3.0, now all user defined)

uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ProjectionMatrix;
uniform mat4 gl_ModelViewProjectionMatrix;
uniform mat3 gl_NormalMatrix;
uniform mat4 gl_TextureMatrix[n];
struct gl_MaterialParameters {
vec4 emission;
vec4 ambient;
vec4 diffuse;
vec4 specular;
float shininess;
};
uniform gl_MaterialParameters gl_FrontMaterial;
uniform gl_MaterialParameters gl_BackMaterial;

Built-in Uniforms (pre GLSL 3.0, now all user defined)

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];

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[n]; // both
varying float gl_FogFragCoord; // both
**Built-in functions**

**Angles & Trigonometry**
- radians, degrees, sin, cos, tan, asin, acos, atan

**Exponentials**
- pow, exp2, log2, sqrt, inversesqrt

**Common**
- abs, sign, floor, ceil, fract, mod, min, max, clamp

**Interpolations**
- mix(x,y,a) \( x*(1.0-a) + y*a \)
- step(edge,x) \( x <= edge \? 0.0 : 1.0 \)
- smoothstep(edge0,edge1,x)
  \[ t = \frac{x - edge0}{edge1 - edge0}; \]
  \[ t = \text{clamp}(t, 0.0, 1.0); \]
  \[ \text{return } t^3*(3.0-2.0*t); \]

**Geometric**
- length, distance, cross, dot, normalize, faceForward, reflect

**Matrix**
- matrixCompMult

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

**Texture**
- texture1D, texture2D, texture3D, textureCube
- texture1DProj, texture2DProj, texture3DProj, textureCubeProj
- shadow1D, shadow2D, shadow1DProj, shadow2DProj

**Vertex**
- ftransform

---

**Example: Vertex Shader**

```glsl
varying vec4 diffuseColor;
varying vec3 fragNormal;
varying vec3 lightVector;
uniform vec3 eyeSpaceLightVector;

void main(){
    vec3 eyeSpaceVertex = vec3(gl_ModelViewMatrix * gl_Vertex);
    lightVector = vec3(normalize(eyeSpaceLightVector - eyeSpaceVertex));
    fragNormal = normalize(gl_NormalMatrix * gl_Normal);
    diffuseColor = gl_Color;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

---

**Example: Fragment Shader**

```glsl
varying vec4 diffuseColor;
varying vec3 lightVector;
varying vec3 fragNormal;

void main(){
    float perFragmentLighting = max(dot(lightVector, fragNormal), 0.0);
    gl_FragColor = diffuseColor * perFragmentLighting;
}
```
Basic method

2 basic object types
- Shader object
- Program object

Create Vertex & Fragment Shader Objects
Compile both
Create program object & attach shaders
Link program
Use program

Compiling

void glShaderSource(GLuint shader, GLsizei nstrings, const GLchar **strings, const GLint *lengths);
//if lengths==NULL, assumed to be null-terminated

void glCompileShader(GLuint shader);

Attaching & Linking

void glAttachShader(GLuint program, GLuint shader);
//twice, once for vertex shader & once for fragment shader

void glLinkProgram(GLuint program);
//program now ready to use

void glUseProgram(GLuint program);
//switches on shader, bypasses FFP
//if program==0, shaders turned off, returns to FFP

In short...

GLuint programObject;
GLuint vertexShaderObject;
GLuint fragmentShaderObject;
unsigned char *vertexShaderSource = readShaderFile(vertexShaderFilename);
unsigned char *fragmentShaderSource = readShaderFile(fragmentShaderFilename);
programObject = glCreateProgram();
vertexShaderObject = glCreateShader(GL_VERTEX_SHADER);
fragmentShaderObject = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(vertexShaderObject, 1, (const char**)&vertexShaderSource, NULL);
glShaderSource(fragmentShaderObject, 1, (const char**)&fragmentShaderSource, NULL);
glCompileShader(vertexShaderObject);
glCompileShader(fragmentShaderObject);
glAttachObject(programObject, vertexShaderObject);
glAttachObject(programObject, fragmentShaderObject);
glLinkProgram(programObject);
glUseProgram(programObject);

Example

void setShaders() {
  char *vs,*fs;
  vs = glCreateShader(GL_VERTEX_SHADER);
  fs = glCreateShader(GL_FRAGMENT_SHADER);
  vs = textFileRead("toon.vert");
  fs = textFileRead("toon.frag");
  const char *vv = vs;
  const char *ff = fs;
  glShaderSource(v, 1, &vv, NULL);
  glShaderSource(f, 1, &ff, NULL);
  free(vs); free(fs);
  glCompileShader(v);
  glCompileShader(f);
  p = glCreateProgram();
  glAttachShader(p, v);
  glAttachShader(p, f);
  glLinkProgram(p);
  glUseProgram(p);
}
Other functions

Clean-up

void glDetachObject (GLuint container, GLuint attached);
void glDeleteObject (GLuint object);

Info Log

void glGetInfoLog (GLuint object, GLsizei maxLength, GLsizei *length, GLchar *infoLog);

See Getting Started Example

Loading Uniforms

void glUniform[1:2:3:4]{f|i} (GLint location,...);

Location obtained with

GLuint glGetUniformLocation (GLuint program, const GLuint *name);

Shader must be enabled with glUseProgramObject () before uniforms can be loaded

Loading Attributes

void glVertexAttrib{1234}{sfd} (GLint index,...);

Index obtained with

GLint glGetAttribLocation (GLuint program, const GLuint *name);

Alternate method

void glBindAttribLocation (GLuint program, GLuint index, const GLuint *name);

Program must be linked after binding attrib locations

Loading Textures

Bind textures to different units as usual

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D,myFirstTexture);

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D,mySecondTexture);

Then load corresponding sampler with texture unit that texture is bound to

glUniform1i (glGetUniformLocation ( programObject,"myFirstSampler"),0);

glUniform1i (glGetUniformLocation ( programObject,"mySecondSampler"),1);

Starter Shaders: color manipulation

// simple.fs

// copy primary color
void main(void)
{
  // Copy the primary color
  gl_FragColor = gl_Color;
}

// colorinvert.fs

// invert like a color negative
void main(void)
{
  // invert color components
  gl_FragColor.rgb = 1.0 - gl_Color.rgb;
  gl_FragColor.a = 1.0;
}

// grayscale.fs

// convert RGB to grayscale
void main(void)
{
  // Convert to grayscale using NTSC conversion weights
  float gray = dot(gl_Color.rgb, vec3(0.299, 0.587, 0.114));
  // replicate grayscale to RGB components
  gl_FragColor = vec3(gray, gray, gray);
  gl_FragColor.a = 1.0;
}

// sepia.fs

// convert RGB to sepia tone
void main(void)
{
  // Convert to grayscale using NTSC conversion weights
  float gray = dot(gl_Color.rgb, vec3(0.299, 0.587, 0.114));
  // convert grayscale to sepia
  gl_FragColor = vec3(gray * vec3(1.2, 1.0, 0.8), 1.0);
}
Starter Shaders: color manipulation

`heatSig.fs`

/// map grayscale to heat signature
uniform sampler1D sampler0;

void main(void)
{
  // Convert to grayscale using NTSC conversion weights
  float gray = dot(gl_Color.rgb, vec3(0.299, 0.587, 0.114));

  // look up heatSig value
  gl_FragColor = texture1D(sampler0, gray);
}

Starter Shaders: color manipulation

`fog.fs`

/// per-pixel fog
uniform float density;

void main(void)
{
  const vec4 fogColor = vec4(0.5, 0.8, 0.5, 1.0);
  // calculate 2nd order exponential fog factor
  // based on fragment's Z distance
  const float e = 2.71828;
  float fogFactor = (density * gl_FragCoord.z);
  fogFactor *= fogFactor;
  fogFactor = clamp(pow(e, -fogFactor), 0.0, 1.0);

  // Blend fog color with incoming color
  gl_FragColor = mix(fogColor, gl_Color, fogFactor);
}

Starter Shaders: convolution

`passthrough.fs`

/// pass through a single texel value
uniform sampler2D sampler0;

void main(void)
{
  gl_FragColor = texture2D(sampler0, gl_TexCoord[0].st);
}

Starter Shaders: convolution

`blur.fs`

/// blur (low-pass) 3x3 kernel
uniform sampler2D sampler0;
uniform vec2 tc_offset[9];

void main(void)
{
  vec4 sample[9];
  for (int i = 0; i < 9; i++)
  {
    sample[i] = texture2D(sampler0, gl_TexCoord[0].st + tc_offset[i]);
  }

  // 1 2 1
  // 2 1 2 / 13
  // 1 2 1
  gl_FragColor = (sample[0] + (2.0*sample[1]) + sample[2] +
                   (2.0*sample[3]) + sample[4] + (2.0*sample[5]) +
                   sample[6] + (2.0*sample[7]) + sample[8]) / 13.0;
}

Starter Shaders: convolution

`sharpen.vs`

1  2  1
Blur  2  1  2 / 13
1  2  1

-1 -1 -1
Sharpen -1  0 -1
-1 -1 -1
-1  0 -1
-1 -1 -1

LaPlace -1  8 -1
-1  0 -1
-1 -1 -1

Dilation max(kernel)
Erosion min(kernel)

Starter Shaders: vertex shaders

`simple.vs`

/// Generic vertex transformation,
// copy primary color
void main(void)
{
  // normal MVP transform
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

  // Copy the primary color
  gl_FrontColor = gl_Color;
}
Starter Shaders: vertex shaders

// diffuse.vs
// Generic vertex transformation,
// diffuse lighting based on one
// white light
uniform vec3 lightPos[1];
void main(void)
{
    // normal MVP transform
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    vec3 N = normalize(gl_NormalMatrix * gl_Normal);
    vec4 V = gl_ModelViewMatrix * gl_Vertex;
    vec3 L = normalize(lightPos[0] - V.xyz);
    // output the diffuse color
    float NdotL = dot(N, L);
    gl_FrontColor = gl_Color * vec4(max(0.0, NdotL));
}

Example: Fragment Shader

varying vec4 diffuseColor;
varying vec3 lightVector;
varying vec3 fragNormal;
void main()
{
    float perFragmentLighting = max(dot(lightVector, fragNormal), 0.0);
    gl_FragColor = diffuseColor * lightingFactor;
}

Starter Shaders: vertex shaders

// ptsize.vs
// Generic vertex transformation,
// attenuated point size
void main(void)
{
    // normal MVP transform
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    vec4 V = gl_ModelViewMatrix * gl_Vertex;
    normal = gl_NormalMatrix * gl_Normal;
    lightVec = vec3(lightPos - V.xyz);
    viewVec = -vec3(V.xyz);
}

Ivory – vertex shader

uniform vec4 lightPos;
varying vec3 normal;
varying vec3 lightVec;
varying vec3 viewVec;
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    vec4 vert = gl_ModelViewMatrix * gl_Vertex;
    normal = gl_NormalMatrix * gl_Normal;
    lightVec = vec3(lightPos - vert);
    viewVec = -vec3(vert);
}

Ivory – fragment shader

varying vec3 normal;
varying vec3 lightVec;
varying vec3 viewVec;
void main()
{
    vec3 norm = normalize(normal);
    vec3 L = normalize(lightVec);
    vec3 V = normalize(viewVec);
    vec3 halfAngle = normalize(L + V);
    float NdotL = dot(L, norm);
    float NdotH = clamp(dot(halfAngle, norm), 0.0, 1.0);
    // "Half-Lambert" technique for more pleasing diffuse term
    float diffuse = 0.5 * NdotL + 0.5;
    float specular = pow(NdotH, 64.0);
    float result = diffuse + specular;
    gl_FragColor = vec4(result);
}
Gooch – vertex shader

```glsl
uniform vec4 lightPos;

varying vec3 normal;

varying vec3 lightVec;

varying vec3 viewVec;

void main(){
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

  vec4 vert = gl_ModelViewMatrix * gl_Vertex;

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

Gooch – fragment shader

```glsl
uniform vec3 ambient;

varying vec3 normal;

varying vec3 lightVec;

varying vec3 viewVec;

void main(){
  const float b = 0.55;
  const float y = 0.3;
  const float Ka = 1.0;
  const float Kd = 0.8;
  const float Ks = 0.9;
  vec3 specularcolor = vec3(1.0, 1.0, 1.0);

  vec3 norm = normalize(normal);
  vec3 L = normalize(lightVec);
  vec3 V = normalize(viewVec);
  vec3 halfAngle = normalize(L + V);

  vec3 orange = vec3(.88, .81, .49);
  vec3 purple = vec3(.58, .10, .76);

  vec3 kCool = purple;
  vec3 kWarm = orange;

  float NdotL = dot(L, norm);
  float NdotH = clamp(dot(halfAngle, norm), 0.0, 1.0);
  float specular = pow(NdotH, 64.0);

  float blendval = 0.5 * NdotL + 0.5;
  vec3 Cgooch = mix(kWarm, kCool, blendval);

  vec3 result = Ka * ambient + Kd * Cgooch + specularcolor * Ks * specular;

  gl_FragColor = vec4(result, 1.0);
}
```

4.30 Layout

Name based matching

```glsl```
```
```
```glsl```
```
```
```glsl```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
4.30 Layout

Location based matching

```cpp
// vertex shader
layout (location = 0) out vec3 normalOut;
layout (location = 1) out vec4 colorOut;
```

```
// geometry shader
layout (location = 0) in vec3 normalIn[];
layout (location = 1) in vec4 colorIn[];
```

```
// fragment shader
layout (location = 0) in vec3 normalIn;
layout (location = 1) in vec4 colorIn;
```

An Introduction to the OpenGL Shading Language

4.30 Layout

Location based matching - Issues

Structs are tricky

```cpp
layout(location = 0) out struct S{
    vec3 normalOut;
    mat3 aMatrix;
    int a;
    float b;
} s;
```

The variable normalOut will get location 0

aMatrix will have locations 1,2 and 3, one for each line.

Location 4 will store a

Location 5 will store b

Next location is 6

What if we add to the struct? (what was defined as 6 is now wrong)

An Introduction to the OpenGL Shading Language

4.30 Layout

Interface blocks

```cpp
out Data {
    vec3 normal;
    vec3 eye;
    vec3 lightDir;
    vec2 texCoord;
} DataOut;
```

```
DataOut.normal = normalize(someVector);
```

An Introduction to the OpenGL Shading Language

4.30 Layout

Location based matching - Issues

Each location is a vector location, i.e. it can hold up to a vector of 4 elements, float or int.

```cpp
// vertex shader
layout (location = 0) out vec3 someAttribute[2];
layout (location = 1) out vec4 colorOut;
```

```cpp
main() {
    someAttribute[1] = …;
    colorOut = …;
    …
}
```

An Introduction to the OpenGL Shading Language

4.30 Layout

Why (without)?

- The locations assigned to parameters might not reflect their declared order.
- A parameter might not be assigned a location at all (if it is statically unreferenced in the shader code).
- Two different GL implementations might (indeed, will) assign locations differently.
- A single implementation might assign locations differently for two shaders that share parameters in common.

As it stands today, you can have clean shader code at the cost of messy application logic (and the loss of some useful mix-and-match functionality), or you can have clean application logic at the cost of uglier shader code.
4.30 Uniform Blocks

Uniform blocks are a very convenient feature for two reasons:
• Allow uniform sharing between programs – set once, use many times
• Allow setting multiple values at once

In application:
uniform ColorBlock {
  vec4 diffuse;
  vec4 ambient;
};

In shader:
...;
void main() {
  outputF = diffuse + ambient;
}

4.30 Uniform Blocks

The default storage for a block is implementation dependent. However, other options are available, and we can specify a storage mode for the block with a layout qualifier.

– std140: The packaging of the variables follows rules defined in the OpenGL specification. Blocks with this layout can be shared among shaders.
– shared: The storage is implementation dependent, but the compiler will ensure that the block is still shareable among different shaders.
– packed: The compiler will optimize the block’s storage, possibly removing any variables that are not used in the shader. These type of blocks should not be shared.

4.30 Uniform Blocks

Blocks are connected through binding points

Blocks can be connected through binding points. The binding point must be smaller than GL_MAX_UNIFORM_BUFFER_BINDINGS.

```c
GLuint bindingPoint = 1, buffer, blockIndex;
float myFloats[8] = {1.0, 0.0, 0.0, 1.0, 0.4, 0.0, 0.0, 1.0};
blockIndex = glGetUniformBlockIndex(p, "ColorBlock");
glUniformBlockBinding(p, blockIndex, bindingPoint);
generateBuffers(buffer);
generateBuffers(GL_UNIFORM_BUFFER, buffer);
generateBuffersType(GL_UNIFORM_BUFFER, GL_DYNAMIC_DRAW);
//To feed values to the shader's block, all that is required is to copy data to the buffer's data store.
```

Useful References

- http://www.3dshaders.com/
  • Home page for the "orange book" focused solely on GLSL
- http://www.opengl.org/sdk/
  • OpenGL SDK, including links to the below resources
- http://www.opengl.org/sdk/libs/OpenSceneGraph/glsl_quickref.pdf
  • one double-sided page cheat sheet to GLSL – indispensable!
  • This is the ultimate authority: the GLSL specification document
- http://www.opengl.org/sdk/docs/books/SuperBible/
  • Full reference and tutorial to OpenGL 2.1
  • All sample code downloadable for Windows, Mac OS X, and Linux