The OpenGL Shading Language

An Introduction
What Are Shaders, And Why Should I Care?

- A *programmable* replacement for parts of the fixed function pipeline

Shaders offer:

- Opportunity for Improved Visual Quality

- Algorithm Flexibility

- Performance Benefits

- The fixed function pipeline is emulated by shaders on current hardware

- Replaced portions of the fixed function pipeline *don’t exist* in new APIs
Types of Shaders

- *Vertex shaders* transform vertices, setup data for fragment shaders
- *Fragment shaders* operate on fragments generated by rasterization
- *Geometry shaders* create geometry on the GPU
- ... More
Programs

• A container for compiled shaders

• Provides the foundation to link shaders together
Where Do Shaders Fit In?

- Lets briefly examine the OpenGL Graphics Pipeline
  - Actually the OpenGL ES Graphics Pipeline
  - Simpler, Similar
Where Do Shaders Fit In?

http://www.khronos.org/opengles/2_X/img/opengles_1x_pipeline.gif
Where Do Shaders Fit In?

http://www.khronos.org/opengles/2_X/img/opengles_2U_pipeline.png
The Vertex Processor

- Common Uses
  - Vertex Transformation
  - Normal Transformation
  - Texture Coordinate Transformation and Generation
  - Animation
  - Setting Up Data For Fragment Shader
The Fragment Processor

- Common Uses
  - Operations on Interpolated Values
  - Texture Application
  - Lighting And Material Application
  - Ray tracing
  - Doing operations per fragment to make pretty pictures
GLSL Syntax

• Borrows syntax from C and C++

• Replaces I/O operations with qualified variables, special variables, and texture reads

• Provides booleans, integers and floating point scalar types

• Adds vector and matrix types and operations

• Provides structs and constant-sized arrays

• No pointers, casting, or implicit type promotion

• Shader entry point is called “main”
Branching and Looping

- if-else: Only on newer hardware
- for, while, do-while
- no switch
- no goto
- discard

- Available only in fragment shaders

- *Effectively* stops the computation and does not update the frame buffer
Vector Declaration

• vec2, vec3, vec4, bvec2, bvec3, bvec4, ivec2, ivec3, ivec4

• ivec2 A = ivec2( 1, 1 );

• vec2 B = vec2( A );

• vec3 C = vec3( 1.0, 1.0, 1.0 );

• vec4 D = vec4( C, 1.0 );
Swizzling

• We can access the components of the vector in one of four ways
  
  • [i], .xyzw, .rgba, .stpq (Equivalent, Use Defined by Semantics)

• We can “swizzle” vectors to access the components in arbitrary order
  vec4 A = vec4(1.0, 2.0, 3.0, 4.0).zywx; // This is legal
  float B = A.q; // This is legal
  vec4 C = A.rgba; // This is legal
  vec4 D = A.xgbq; // This is illegal

• Assigning to a swizzled vector
  vec3 E = vec3(0.0);
  E.x = A.w; // This is legal
  E.wyxz = A.xxxw; // This is legal
  E.xxxx = A.xyzw; // This is illegal
Matrix Declaration

- mat2, mat3, mat4
- Available in GLSL 1.20+
  - mat2x2, mat2x3, mat2x4
  - mat3x2, mat3x3, mat3x4
  - mat4x2, mat4x3, mat4x4
- mat4 = mat4(1.0)
- mat4 = mat4( vec4(1.0), vec4(2.0), vec4(3.0), vec4(4.0) );
Matrix “Swizzling”

- We can access the *columns* of a matrix as *vectors* with array syntax

```
mat4 A = mat4(1.0);
vec4 B = A[0];
vec3  C = A[1].xyz;
float D = A[0][0];
```
Creating And Installing Shaders And Programs

- Create shader object
- Supply source code for shader object
- Compile shader
- Create program object
- Attach shader to program
- Link program
- Use program
Creating And Installing Shaders And Programs

- Create shader object
- Supply source code for shader object
- Compile shader
- Create program object
- Attach shader to program
- Link program
- Use program
Creating And Installing Shaders And Programs

- Create shader object
- Supply source code for shader object
- Compile shader
- Create program object
- Attach shader to program
- Link program
- Use program
Creating And Installing Shaders And Programs

- Create shader object
- Supply source code for shader object
- **Compile shader**
- Create program object
- Attach shader to program
- Link program
- Use program
Creating And Installing Shaders And Programs

- Create shader object
- Supply source code for shader object
- Compile shader

**Create program object**
- Attach shader to program
- Link program
- Use program
Creating And Installing Shaders And Programs

- Create shader object
- Supply source code for shader object
- Compile shader
- Create program object
- Attach shader to program
- Link program
- Use program

glCreateShader
glShaderSource
glCompileShader
glCreateProgram
glAttachShader
Creating And Installing Shaders And Programs

- Create shader object: `glCreateShader`
- Supply source code for shader object: `glShaderSource`
- Compile shader: `glCompileShader`
- Create program object: `glCreateProgram`
- Attach shader to program: `glAttachShader`
- Link program: `glLinkProgram`
- Use program
Creating And Installing Shaders And Programs

- Create shader object: `glCreateShader`
- Supply source code for shader object: `glShaderSource`
- Compile shader: `glCompileShader`
- Create program object: `glCreateProgram`
- Attach shader to program: `glAttachShader`
- Link program: `glLinkProgram`
- Use program: `glUseProgram`
Creating And Installing Shaders And Programs

- `glCreateShader`
  - `glShaderSource`
  - `glCompileShader`

- `glCreateProgram`
  - `glAttachShader`
  - `glLinkProgram`
  - `glUseProgram`
  - `glAttachShader`
  - `glLinkProgram`
  - `glUseProgram`
An Example

GLuint program = 0;

const GLchar *vertexSource = "..........";
const GLchar *fragmentSource = ".........."

GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexSource, 0);
glCompileShader(vertexShader);

GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentSource, 0);
glCompileShader(fragmentShader);

program = glCreateProgram();
glAttachShader(program, vertexShader);
glAttachShader(program, fragmentShader);
glLinkProgram(program);

glUseProgram(program);

• Create shader object
• Supply source code for shader object
• Compile shader
• Create program object
• Attach shader to program
• Link program
• Use program
Reporting Compilation and Linking Errors

- glGetShaderInfoLog
  - Retrieve shader compilation errors

- glGetProgramInfoLog
  - Retrieve program linking errors
Getting Error Logs

GLsizei maxLength = 1024;
GLsizei length[3] = {0};
GLchar infoLog[3][1024] = {0};

glGetShaderInfoLog(vertexShader, maxLength, &length[0], infoLog[0]);
glGetShaderInfoLog(fragmentShader, maxLength, &length[1], infoLog[1]);
glGetProgramInfoLog(program, maxLength, &length[2], infoLog[2]);
Anatomy of A Shader

// Vertex Shader
void main() {
  gl_Position = gl_Vertex;
}

// Fragment Shader
void main() {
  gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}

• Shader entry point is main

  • returns void, takes no arguments

• How do we get data in and out of our shaders?

  • Qualified variables, special variables and texture reads replace I/O operations
Qualifiers

- Uniform
  - Input to vertex shader from application
  - Information that changes infrequently

- Attribute
  - Input to vertex shader from application
  - Information that changes frequently
Qualifiers

• Varying
  • Output of vertex shader
  • Input to fragment shader
  • Information interpolated between vertices
    • Color
    • Normals
    • Direction To Light Source
Qualifiers And Special Variables

// Vertex Shader
void main() {
    gl_Position = gl_Vertex;
}

// Fragment Shader
void main() {
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}

• What is the qualifier for gl_Vertex?

• What are the qualifiers for gl_Position and gl_FragColor?
Anatomy Of A Vertex Shader

// Vertex Shader
void main() {
    gl_Position = gl_Vertex;
}

• A vertex shader must write to gl_Position

• A vertex shader can write to gl_PointSize, gl_ClipVertex

• gl_Vertex is an attribute supplying the untransformed vertex coordinate

• gl_Position is an special output variable for the transformed vertex coordinate
Anatomy Of A Fragment Shader

// Fragment Shader
void main() {
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}

- A fragment shader can write to the following special output variables

  - gl_FragColor to set the color of the fragment

  - gl_FragData[n] to output to a specific render target

  - gl_FragDepth to set the fragment depth
Before We Get Any Further....

- Can I install a vertex shader with no fragment shader, or visa versa?
  - Yes! The fixed function pipeline will be used

- Can I attach multiple vertex or fragment shaders to a program?
  - Yes! But there should only be one main per attached shader type
An Example: Diffuse Shading

- OpenGL computes shading per vertex and interpolates across the surface

- We are going to compute ambient and diffuse shading per fragment

  - No attenuation

  - No specular term

- Easy to add (try it!)
An Example: Diffuse Shading

• Check List:

  • Transform Vertex and Normal
  
  • Compute The Vector From Vertex To Light Source
  
  • Pass Information From Vertex Shader To Fragment Shader
  
  • Compute Shading
Modifying the Vertex Shader

- Let’s modify the vertex program to transform vertices
- We need to multiply the vertex by the modelview and projection matrices

```c
// Vertex Shader
void main() {
    gl_Position = gl_Vertex;
}
```
Transforming Vertices

• Let’s modify the vertex program to transform vertices

• We need to multiply the vertex by the modelview and projection matrices

```c
// Vertex Shader
void main() {
    gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
}
```

Built In 4x4 Matrix Provided By OpenGL

• What are the qualifiers for gl_ModelViewMatrix, and gl_ProjectionMatrix?
Transforming Vertices

- Let's modify the vertex program to transform vertices

- We need to multiply the vertex by the modelview and projection matrices

```c
// Vertex Shader
void main() {
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Built In 4x4 Matrix Provided By OpenGL
Transforming Normals

- We need to multiply the normal by the normal matrix

```c
// Vertex Shader
vec3 frag_Normal;
void main() {
    frag_Normal = gl_NormalMatrix * gl_Normal;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```
Calculating The Light Vector

- Light Vector = Light Position - Vertex Position

```c
// Vertex Shader
vec3 frag_Light;
vec3 frag_Normal;
void main() {
  frag_Light = gl_LightSource[0].position.xyz - ........;
  frag_Normal = gl_NormalMatrix * gl_Normal;
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

- Light Position Is Stored In Eye Space

- Already multiplied by the modelview matrix!
Calculating The Light Vector

- Light Vector = Light Position - Vertex Position

```cpp
// Vertex Shader
vec3 frag_Light;
vec3 frag_Normal;
void main() {
    vec4 vertex = gl_ModelViewMatrix * gl_Vertex;
    frag_Light = gl_LightSource[0].position.xyz - vertex.xyz;
    frag_Normal = gl_NormalMatrix * gl_Normal;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

- Now the subtraction is occurring in the same space!
Using The Varying Qualifier

- If mark global variables as "varying" in the vertex shader...

```glsl
// Vertex Shader
varying vec3 frag_Light;
varying vec3 frag_Normal;
void main() {
    vec4 vertex = gl_ModelViewMatrix * gl_Vertex;
    frag_Light = gl_LightSource[0].position.xyz - vertex.xyz;
    frag_Normal = gl_NormalMatrix * gl_Normal;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```
Using The Varying Qualifier

- And in the fragment shader, we can pass *interpolated* information
  
- Data always moves from vertex shader to fragment shader

```cpp
// Fragment Shader
varying vec3 frag_Light;
varying vec3 frag_Normal;

void main() {
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```
Using The Varying Qualifier

• We need to calculate diffuse shading

• Formula is:

  \[ \text{Light}_{\text{Diffuse}} \times \text{Material}_{\text{Diffuse}} \times \max(0.0, \text{dot}(\text{Normal}, \text{Light})) \]

• Normal is the *normalized* surface normal

• Light is the *normalized* vector to the light source
// Fragment Shader
varying vec3 frag_Light;
varying vec3 frag_Normal;
void main() {
    vec3 N = normalize(frag_Light);
    vec3 L = normalize(frag_Normal);
    float nDotL = max(0.0, dot(N, L));
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
// Fragment Shader
varying vec3 frag_Light;
varying vec3 frag_Normal;
void main() {
    vec3 N = normalize(frag_Light);
    vec3 L = normalize(frag_Normal);
    float nDotL = max(0.0, dot(N, L));
    gl_FragColor = gl_FrontLightProduct[0].diffuse * nDotL;
}
Finish The Lighting Model

- Try it yourself!

- Global Ambient + attenuation * ( Ambient + Diffuse + Specular )
  
  - Global Ambient: gl_FrontLightModelProduct.sceneColor

  - Attenuation = 1.0 / A + B * Distance To Light + C * (Distance To Light)^2

    - A, B, C = Look at gl_LightSource[i].constantAttenuation ....

  - Specular = pow ( max(0.0, dot(Normal, HalfVector)), shininess )

    - gl_LightSource[i].halfVector, gl_FrontMaterial.shininess
Preparing for GLSL 1.30

• GLSL 1.30 will no longer provide built-in uniforms and attributes

  • We must pass in all values we wish to use

  • Requires less work per state change (Higher Performance)
Using The Uniform Qualifier

• Let’s replace `gl_ModelViewMatrix` and `gl_ProjectionMatrix` with our own uniform matrices

• We have to
  
  • Declare two uniform matrices
  
  • Pass matrix data to the shader
    
    • `glGetUniformLocation`
    
    • `glUniformMatrix4f`
    
    • `glUniform1f, glUniform2f, ... glUniformMatrix2f, glUniformMatrix3f, ...`
Using The Uniform Qualifier

// Vertex Shader
uniform mat4 ProjectionMatrix;
uniform mat4 ModelViewMatrix;
void main() {
    gl_Position = ProjectionMatrix * ModelViewMatrix * gl_Vertex;
}
Using The Uniform Qualifier

// Application Code
float projectionMatrix[16] = ......
float modelViewMatrix[16] = ......
GLuint projection = glGetUniformLocation(program, "ProjectionMatrix");
GLuint modelView = glGetUniformLocation(program, "ModelViewMatrix");
glUniformMatrix4fv(projection, 1, GL_FALSE, projectionMatrix);
glUniformMatrix4fv(modelView, 1, GL_FALSE, modelViewMatrix);
Using Attributes

- Available only in vertex shaders
- Declare global variable as attribute
  - attribute vec4 gl_Vertex;
- A large variety of glVertexAttrib calls
  - Prefer glVertexAttribPointer
glVertexAttribPointer

- Replaces all previous vertex array functionality

- Arguments: index, size, type, normalized, stride, pointer

  - size, type, stride and pointer similar to glVertexPointer

- normalized is a boolean

  - If GL_TRUE, values in pointer are mapped between 0 and 1
  - If GL_FALSE, values in pointer are directly converted to floats

- Must call glEnableVertexAttribArray(index) before using....

- What is index?
Attribute Index

- Every attribute has an index
  - A number (like a memory address) that identifies their location
  - We can define that number ourselves
    - `glBindAttribLocation(program, index, name)`
      - Must be called before calling `glLinkProgram`
  - We can let OpenGL define it for us
    - `glGetAttribLocation(program, name)`
Attribute Index

- gl_Vertex is an attribute

- Specified by calling glVertex

- Specified by calling glVertexAttrib* with index 0

- Specified by calling glVertexAttribPointer with index 0

  - Signals the end of data for a given vertex
Using The Attribute Qualifier

- Let’s use attributes to submit vertex data
  - We have to declare an attribute
  - We have to pass data to that attribute
    - We will use glVertexAttrib for the demonstration....
Using The Attribute Qualifier

// Vertex Shader
uniform mat4 ProjectionMatrix;
uniform mat4 ModelViewMatrix;
attribute vec4 Vertex;
void main() {
    gl_Position = ProjectionMatrix * ModelViewMatrix * Vertex;
}
Using The Attribute Qualifier

// Application Code
...
glBindAttribLocation(program, 0, "Vertex");
gLinkProgram(program);
...
glBegin(GL_QUADS);
glVertexAttrib4f(0, -10.0f, -10.0f, -25.0f, 1.0f);
glVertexAttrib4f(0, 10.0f, -10.0f, -25.0f, 1.0f);
glVertexAttrib4f(0, 10.0f, 10.0f, -25.0f, 1.0f);
glVertexAttrib4f(0, -10.0f, 10.0f, -25.0f, 1.0f);
glEnd();
An Example

• Lets try drawing with glutSolidTeapot
  
  • We won’t be calling glVertexAttrib
  
  • But glVertexAttrib with index 0 is the same as calling glVertex
Demo:
What Just Happened?

- We are setting the modelview matrix....
What Just Happened?

• We are setting the modelview matrix.....

  • before calling glutSolidTeapot

    • glutSolidTeapot calls glRotate

    • Slightly annoying when dealing with glut* objects

    • In general not an issue