CMPS160 – Shader-based OpenGL Programming

All slides originally from Prabath Gunawardane, et al. unless noted otherwise
Shader gallery I

Above: Demo of Microsoft’s XNA game platform
Right: Product demos by nvidia (top) and Radeon (bottom)
Shader gallery II


Above: Ben Cloward (“Car paint shader”)
What is… the shader?

The next generation:
Introduce *shaders*, programmable logical units on the GPU which can replace the “fixed” functionality of OpenGL with user-generated code.

By installing custom shaders, the user can now completely override the existing implementation of core per-vertex and per-pixel behavior.
What are we targeting?

- OpenGL shaders give the user control over each *vertex* and each *fragment* (each pixel or partial pixel) interpolated between vertices.

- After vertices are processed, polygons are *rasterized*. During rasterization, values like position, color, depth, and others are interpolated across the polygon. The interpolated values are passed to each pixel fragment.
A Vertex Program processes each vertex
Vertex Shader – inputs and outputs

- Per-vertex attributes
- Custom variables
- Vertex Shader
- Color, Position
- Custom variables
vertex shaders can be used to move/animate verts
Geometry Shader

Takes as input vertices and connectivity information and produces more primitives
Fragment Shader – inputs and outputs

- Primitives
- Vertex attributes (normals, texture coordinates, etc.)
- Geometry Shader

Outputs:
- Primitives
- Vertex color, position, etc.
A Fragment Program processes each fragment
Fragment Shader – inputs and outputs

- Color
- Texture coords
- Fragment coords
- Front facing
- Texture data
- Material
- Lighting
- etc…

Fragment Shader

- Fragment color
- Fragment depth

Custom variables
Each pixel is calculated individually
What can you override?

Per vertex:
- Vertex transformation
- Normal transformation and normalization
- Texture coordinate generation
- Texture coordinate transformation
- Lighting
- Color material application

Per fragment (pixel):
- Operations on interpolated values
- Texture access
- Texture application
- Fog
- Color summation

Optionally:
- Pixel zoom
- Scale and bias
- Color table lookup
- Convolution
Think parallel

- Shaders are compiled from within your code
  - They used to be written in assembler
  - Today they’re written in high-level languages (😊)
- They execute on the GPU
- GPUs typically have multiple processing units
- That means that multiple shaders execute in parallel!
What’re our options?

- There are several popular languages for describing shaders, such as:
  - **HLSL**, the *High Level Shading Language*
    - Author: Microsoft
    - DirectX 8+
  - **Cg**
    - Author: nvidia
  - **GLSL**, the *OpenGL Shading Language*
    - Author: the Khronos Group, a self-sponsored group of industry affiliates (ATI, 3DLabs, etc)
The language design in GLSL is strongly based on ANSI C, with some C++ added.

- There is a preprocessor--**#define**, etc!
- Basic types: int, float, bool
- Vectors and matrices are standard: `vec2`, `mat2 = 2x2`; `vec3`, `mat3 = 3x3`; `vec4`, `mat4 = 4x4`
- Texture samplers: `sampler1D`, `sampler2D`, etc are used to sample multidimensional textures
- Functions can be declared before they are defined, and operator overloading is supported.
Some differences from C/C++:
- No pointers, strings, chars; no unions, enums; no bytes, shorts, longs; no unsigned. No switch() statements.
- There is no implicit casting (type promotion):
  ```
  float foo = 1;
  ```
  fails because you can’t implicitly cast `int` to `float`.
- Explicit type casts are done by constructor:
  ```
  vec3 foo = vec3(1.0, 2.0, 3.0);
  vec2 bar = vec2(foo);  // Drops foo.z
  ```
- Function parameters are labeled as `in` (default), `out`, or `inout`.
- Functions are called by `value-return`, meaning that values are copied into and out of parameters at the start and end of calls.
A Quick Peek at a shader program usage

```cpp
QGLShader *vshader = new QGLShader(QGLShader::Vertex);
const char *vsrc =
"attribute highp vec4 vertex;\n"attributes highp vec4 texCoord;\n"attribute mediump vec3 normal;\n"uniform mediump mat4 matrix;\n"varying highp vec4 texc;\n"varying mediump float angle;\n"void main(void)\n"{\n" vec3 toLight = normalize(vec3(0.0, 0.3, 1.0));\n" angle = max(dot(normal, toLight), 0.0);\n" gl_Position = matrix * vertex;\n" texc = texCoord;\n"}\n";
vshader->compileSourceCode(vsrc);
```
How do the shaders communicate?

There are three types of shader parameter in GLSL:

- **Uniform parameters**
  - Set throughout execution
  - Ex: surface color

- **Attribute parameters**
  - Set per vertex
  - Ex: local tangent

- **Varying parameters**
  - Passed from vertex processor to fragment processor
  - Ex: transformed normal
What happens when you install a shader?

- All the fixed functionality is overridden.
- It’s up to you to replace it!
  - You’ll have to transform each vertex into viewing coordinates manually.
  - You’ll have to light each vertex manually.
  - You’ll have to apply the current interpolated color to each fragment manually.
- The installed shader replaces all OpenGL fixed functionality for all renders until you remove it.
Shader sample one – ambient lighting

// Vertex Shader
attribute highp vec4 vertex;
uniform mediump mat4 modelviewmatrix;
void main() {
    gl_Position = modelviewmatrix * vertex;
}

// Fragment Shader
void main() {
    gl_FragColor = vec4(0.2, 0.6, 0.8, 1);
}
Shader sample one – ambient lighting
Shader sample one – ambient lighting

- Notice the C-style syntax
  - `void main() { ... }`
- The vertex shader uses two inputs, `vertex` and the model-view-projection matrix; and one standard output, `gl_Position`.
  - The line
    ```gl_Position = modelviewmatrix * vertex;```
    applies the model-view-projection matrix to calculate the correct vertex position in perspective coordinates.
- The fragment shader applies basic ambient lighting, setting its one standard output, `gl_FragColor`, to a fixed value.
Shader sample two – diffuse lighting

// Vertex Shader

varying mediump vec3 Norm;
varying mediump vec3 ToLight;
attribute highp vec4 vertex;
uniform mediump mat4 modelviewmatrix;
uniform mediump mat4 normalmatrix;
uniform mediump vec3 lightposition;

void main()
{
    gl_Position = modelviewmatrix * vertex;
    Norm = normalmatrix * normal;
    ToLight = vec3(lightposition - vertex);
}

// Fragment Shader

varying mediump vec3 Norm;
varying mediump vec3 ToLight;

void main()
{
    const vec3 DiffuseColor = vec3(0.2, 0.6, 0.8);
    float diff = clamp(dot(normalize(Norm), normalize(ToLight)), 0.0, 1.0);
    gl_FragColor = vec4(DiffuseColor * diff, 1.0);
}
Shader sample two – diffuse lighting
Shader sample two – diffuse lighting

- This examples uses *varying parameters* to pass info from the vertex shader to the fragment shader.
  - The varying parameters *Norm* and *ToLight* are automatically linearly interpolated between vertices across every polygon.
  - This represents the normal at that exact point on the surface.
  - The exact diffuse illumination is calculated from the local normal.
- This is the Phong shading technique (usually seen for specular highlights) applied to diffuse lighting.
Shader sample two – diffuse lighting

Notice the different matrix transforms used in this example:

\[
gl\_Position = gl\_ModelViewProjectionMatrix * gl\_Vertex;
\]

\[
Norm = gl\_NormalMatrix * gl\_Normal;
\]

\[
ToLight = vec3(gl\_LightSource[0].position - (gl\_ModelViewMatrix * gl\_Vertex));
\]

The \texttt{gl\_ModelViewProjectionMatrix} transforms a vertex from local coordinates to perspective coordinates for display, whereas the \texttt{gl\_ModelViewMatrix} transforms a point from local coordinates to eye coordinates. We use eye coordinates because lights are (usually) defined in eye coordinates.

The \texttt{gl\_NormalMatrix} transforms a normal from local coordinates to eye coordinates; it holds the inverse of the transpose of the upper 3x3 submatrix of the model-view transform.
Shader sample three – Gooch shading

// From the Orange Book

varying float NdotL;
varying vec3 ReflectVec;
varying vec3 ViewVec;

void main () {
    vec3 ecPos = vec3(gl_ModelViewMatrix * gl_Vertex);
    vec3 tnorm = normalize(gl_NormalMatrix * gl_Normal);
    vec3 lightVec = normalize(gl_LightSource[0].position.xyz - ecPos);
    ReflectVec = normalize(reflect(-lightVec, tnorm));
    ViewVec = normalize(-ecPos);
    NdotL = (dot(lightVec, tnorm) + 1.0) * 0.5;
    gl_Position = ftransform();
    gl_FrontColor = vec4(vec3(0.75), 1.0);
    gl_BackColor = vec4(0.0);
}

gl_FrontColor = vec4(vec3(0.75), 1.0);
gl_BackColor = vec4(0.0);

vec3 SurfaceColor = vec3(0.75, 0.75, 0.75);
vec3 WarmColor = vec3(0.1, 0.4, 0.8);
vec3 CoolColor = vec3(0.6, 0.0, 0.0);
float DiffuseWarm = 0.45;
float DiffuseCool = 0.045;

void main() {
    vec3 kcool = min(CoolColor + DiffuseCool * vec3(gl_Color), 1.0);
    vec3 kwarm = min(WarmColor + DiffuseWarm * vec3(gl_Color), 1.0);
    vec3 kfinal = mix(kcool, kwarm, NdotL) * gl_Color.a;
    vec3 nreflect = normalize(ReflectVec);
    vec3 nview = normalize(ViewVec);
    float spec = max(dot(nreflect, nview), 0.0);
    spec = pow(spec, 32.0);
    gl_FragColor = vec4(min(kfinal + spec, 1.0), 1.0);
}
Shader sample three – Gooch shading
*Gooch shading* is not a shader technique per se.

It was designed by Amy and Bruce Gooch to replace photorealistic lighting with a lighting model that highlights structural and contextual data.

- They use the diffuse term of the conventional lighting equation to choose a map between ‘cool’ and ‘warm’ colors.
- This is in contrast to conventional illumination where diffuse lighting simply scales the underlying surface color.
- This, combined with edge-highlighting through a second renderer pass, creates models which look more like engineering schematic diagrams.
In the vertex shader source, notice the use of the built-in ability to distinguish front faces from back faces:

\[
\text{gl\_FrontColor} = \text{vec4}\left(\text{vec3}\left(0.75\right), 1.0\right); \\
\text{gl\_BackColor} = \text{vec4}\left(0.0\right);
\]

This supports distinguishing front faces (which should be shaded smoothly) from the edges of back faces (which will be drawn in heavy black.)

In the fragment shader source, this is used to choose the weighted diffuse color by clipping with the \( a \) component:

\[
\text{vec3 kfinal} = \text{mix}\left(\text{kcool}, \text{kwarm}, \text{NdotL}\right) \times \text{gl\_Color.}
\]

Here \( \text{mix()} \) is a GLSL method which returns the linear interpolation between \( \text{kcool} \) and \( \text{kwarm} \). The weighting factor (‘\( t \)’ in the interpolation) is \( \text{NdotL} \), the diffuse lighting value.
Using shaders in your OpenGL program

```
Program
  glVertexShader
    glVertexShaderSource
    glVertexCompileShader
  glVertexLinkProgram
  glVertexUseProgram

Fragment Shader
  gl ShaderSource
    gl CompileShader

Vertex Shader
  gl ShaderSource
    gl CompileShader
```


OpenGL Versions

- 1.0 – 1.5 Fixed function pipeline
- 2.0 - 2.1 Add support for programmable shaders, retain backward compatibility
- 3.0 adopts deprecation model but retains backward compatibility
- 3.1 fixed-function pipeline and associated functions removed
- ES 1.1 Stripped down fixed-function version
- ES 2.0 Shader-only version
## Differences from Fixed-function pipeline

<table>
<thead>
<tr>
<th>Instead of…</th>
<th>Use!</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>glTranslate()</code></td>
<td>Vertex/Attribute Arrays</td>
</tr>
<tr>
<td><code>glRotate()</code></td>
<td><code>glDrawArrays()</code></td>
</tr>
<tr>
<td><code>glVertex()</code></td>
<td><code>glDrawElements()</code></td>
</tr>
<tr>
<td><code>glNormal()</code></td>
<td>Utility methods for matrix manipulation</td>
</tr>
</tbody>
</table>
Simple Qt OpenGL Shader example

A striped down version of hellogl_es2 Qt example
Steps in your OpenGL program

1. Compile and Link the shaders
   - Passing variables to shader programs

2. Allocate object data
   - Vertices, Normals, Texture coordinates, etc.

3. Draw object
Compile and Link the shaders (1)

```cpp
QGLShader *vshader = new QGLShader(QGLShader::Vertex);
const char *vsrc =
   "attribute highp vec4 vertex;\n" 
   "attribute highp vec4 texCoord;\n" 
   "attribute mediump vec3 normal;\n" 
   "uniform mediump mat4 matrix;\n" 
   "varying highp vec4 texC;\n" 
   "varying mediump float angle;\n" 
   "void main(void)\n" 
   "{\n" 
   "   vec3 toLight = normalize(vec3(0.0, 0.3, 1.0));\n" 
   "   angle = max(dot(normal, toLight), 0.0);\n" 
   "   gl_Position = matrix * vertex;\n" 
   "   texC = texCoord;\n" 
   "}\n";

vshader->compileSourceCode(vsrc);

QGLShader *fshader = new QGLShader(QGLShader::Fragment);
const char *fsrc =
   "varying highp vec4 texC;\n" 
   "uniform sampler2D tex;\n" 
   "varying mediump float angle;\n" 
   "void main(void)\n" 
   "{\n" 
   "   highp vec3 color = texture2D(tex, texC.st).rgb;\n" 
   "   color = color * 0.2 + color * 0.8 * angle;\n" 
   "   gl_FragColor = vec4(clamp(color, 0.0, 1.0), 1.0);\n" 
   "}\n";
fshader->compileSourceCode(fsrc);
```

Compile and Link the shaders (2)

```cpp
QGLShaderProgram program;

fshader->compileSourceCode(fsrc);
program.addShader(vshader);
program.addShader(fshader);
program.link();

 vertexAttr = program.attributeLocation("vertex");
 normalAttr = program.attributeLocation("normal");
 texCoordAttr = program.attributeLocation("texCoord");
 matrixUniform = program.uniformLocation("matrix");
 textureUniform = program.uniformLocation("tex");
```
Passing in variables to shaders

```cpp
QGLShader *vshader = new QGLShader(QGLShader::Vertex);
const char *vsrc =
  "attribute highp vec4 vertex;\n  attribute highp vec4 texCoord;\n  attribute mediump vec3 normal;\n  uniform mediump mat4 matrix;\n  "
  "varying highp vec4 texc;\n  "varying mediump float angle;\n  "void main(void)\n  "{\n    vec3 toLight = normalize(vec3(0.0, 0.3, 1.0));\n    angle = max(dot(normal, toLight), 0.0);\n    gl_Position = matrix * vertex;\n  }\n";

fshader->compileSourceCode(vsrc);

program.addShader(vshader);
program.addShader(fshader);
program.link();

vertexAttr = program.attributeLocation("vertex");
normalAttr = program.attributeLocation("normal");
texCoordAttr = program.attributeLocation("texCoord");
matrixUniform = program.uniformLocation("matrix");
textureUniform = program.uniformLocation("tex");
```
Allocate object data

```cpp
{
    glBindTexture(GL_TEXTURE_2D, m_uiTexture);
    GLfloat afVertices[] = {
        -0.5, 0.5, 0.5, 0.5,-0.5,0.5,-0.5,-0.5,0.5,
        0.5, -0.5, 0.5, -0.5,0.5,0.5,0.5,0.5,0.5,
        -0.5, -0.5, -0.5, 0.5,-0.5,-0.5,-0.5,0.5,-0.5,
        0.5, 0.5, -0.5, -0.5,0.5,0.5,-0.5,0.5,-0.5,
        0.5, -0.5, -0.5, 0.5,-0.5,0.5,0.5,0.5,-0.5,
        0.5, 0.5, 0.5, 0.5,0.5,-0.5,0.5,-0.5,0.5,
        -0.5, 0.5, -0.5, -0.5,0.5,-0.5,-0.5,-0.5,-0.5,
        -0.5, -0.5, 0.5, -0.5,0.5,-0.5,0.5,-0.5,0.5,
        0.5, 0.5, -0.5, -0.5,0.5,0.5,-0.5,0.5,-0.5,
        -0.5, 0.5, 0.5, 0.5,0.5,-0.5,0.5,0.5,0.5,
        -0.5, -0.5, -0.5, 0.5,-0.5,0.5,0.5,-0.5,-0.5,
        0.5, -0.5, 0.5, 0.5,0.5,-0.5,0.5,-0.5,0.5,
    };
    program.setAttributeArray(vertexAttr, afVertices, 3);
}
```
// Manage our own ModelView matrix
QMatrix4x4 modelview;
modelview.rotate(m_fAngle, 0.0f, 1.0f, 0.0f);
modelview.rotate(m_fAngle, 1.0f, 0.0f, 0.0f);
modelview.rotate(m_fAngle, 0.0f, 0.0f, 1.0f);
modelview.scale(m_fScale);
modelview.translate(0.0f, -0.2f, 0.0f);

// Bind the vertex and fragment shader that we've created
program.bind();
// Pass the ModelView matrix to the vertex shader in as a uniform variable
program.setUniformValue(matrixUniform, modelview);
paintTexturedCube();
// Release the vertex and fragment shader. We could use several vertex / fragment shaders for
program.release();

glDisable(GL_DEPTH_TEST);
Draw Object (2)

```java
program.setUniformValue(textureUniform, 0);  // use texture unit 0

program.enableAttributeArray(vertexAttr);
program.enableAttributeArray(normalAttr);
program.enableAttributeArray(texCoordAttr);

glDrawArrays(GL_TRIANGLES, 0, 36);

program.disableAttributeArray(vertexAttr);
program.disableAttributeArray(normalAttr);
program.disableAttributeArray(texCoordAttr);
```
Recap

- Shaders give a powerful, extensible mechanism for programming the vertex and pixel processing stages of the GPU pipeline.
- GLSL is a portable, multiplatform C-like language which is compiled at run-time and linked into an executable shader program.
- Shaders can be used for a long list of effects, from procedural geometry and non-photorealistic lighting to advanced textures, fog, shadows, raycasting, and visual effects; in fact, many of the topics covered in this course!
Resources

http://www.lighthouse3d.com/opengl/gls/
The GLSL API

To install and use a shader in OpenGL:
1. Create one or more empty shader objects with `glCreateShader`.
2. Load source code, in text, into the shader with `glShaderSource`.
3. Compile the shader with `glCompileShader`.
4. Create an empty program object with `glCreateProgram`.
5. Bind your shaders to the program with `glAttachShader`.
6. Link the program (ahh, the ghost of C!) with `glLinkProgram`.
7. Register your program for use with `glUseProgram`.

1. The compiler cannot detect every program that would cause a crash. (And if you can prove otherwise, see me after class.)
What is... the shader?

Lecture one...

Local space
World space
Viewing space
3D screen space

Process vertices
Clipping, projection, backface culling
Process pixels
2D display space – plot pixels

Closer to the truth (but still a terrible oversimplification)
What is… the shader?

- World space
- Viewing space
- 3D screen space
- Local space

Lecture one...

Process vertices
- Clipping, projection, backface culling

Process pixels
- 2D display space – plot pixels

Ex: computing diffuse shading color per vertex; transforming vertex position; transforming texture co-ordinates

Ex: interpolating texture coordinates across the polygon; interpolating the normal for specular lighting; textured normal-mapping

Closer to the truth (but still a serious oversimplification)

“Wouldn’t it be great if the user could install their own code into the hardware to choose these effects?”
OpenGL programmable processors

GLSL – design goals (NEEDED?)

GLSL was designed with the following in mind:

- Work well with OpenGL
- Shaders should be optional extras, not required.
- Fit into the design model of “set the state first, then render the data in the context of the state”
  - Support upcoming flexibility
  - Be hardware-independent
- The GLSL folks, as a broad consortium, are far more invested in hardware-independence than, say, nvidia.
- That said, they’ve only kinda nailed it: I get different compiler behavior and different crash-handling between my high-end home nVidia chip and my laptop Intel x3100.
  - Support inherent parallelization
  - Keep it streamlined, small and simple
OpenGL’s Geometric Primitives

- GL_POINTS
- GL_LINES
- GL_LINE_STRIP
- GL_LINE_LOOP
- GL_TRIANGLE_STRIP
- GL_TRIANGLES
- GL_TRIANGLE_FAN
Particle systems on the GPU

- Shaders extend the use of texture memory dramatically. Shaders can write to texture memory, and textures are no longer limited to being a two-dimensional plane of RGB(A).
- A particle system can be represented by storing a position and velocity for every particle.
- A fragment shader can render a particle system entirely in hardware by using texture memory to store and evolve particle data.

Image by Michael Short
Particle systems with shaders

Slide 17 of Lutz Latta’s “Everything About Particle Effects”, delivered at the Game Developers Conference ’07 (San Francisco).