Programming with OpenGL
Part 3: Shaders

CS 432/537 Interactive Computer Graphics
Prof. David E. Breen
Department of Computer Science
Objectives

• Simple Shaders
  - Vertex shader
  - Fragment shaders
• Programming shaders with GLSL
• Finish first program
Vertex Shader Applications

• Moving vertices
  - Transformations
    • Modeling
    • Projection
  - Morphing
  - Wave motion
  - Fractals
  - Particle systems

• Lighting
  - More realistic shading models
  - Cartoon shaders
Fragment Shader Applications

Per fragment lighting calculations

per vertex lighting (Gouraud shading)  per fragment lighting (Phong shading)

Fragment Shader Applications

Texture mapping

Procedural textures  environment mapping  bump mapping

Writing Shaders

• First programmable shaders were programmed in an assembly-like manner
• OpenGL extensions added vertex and fragment shaders
• Cg (C for graphics) C-like language for programming shaders
  - Works with both OpenGL and DirectX
  - Interface to OpenGL complex
• OpenGL Shading Language (GLSL)
GLSL

- OpenGL Shading Language
- Part of OpenGL 2.0 and up
- High level C-like language
- New data types
  - Matrices
  - Vectors
  - Samplers
- As of OpenGL 3.1, application must provide shaders
Execution Model

Vertices → Vertex Processor → Clipper and Primitive Assembler → Rasterizer → Fragment Processor → Pixels

Vertex data
Shader Program

Application Program → Vertex Shader

glDrawArrays → Vertex Shader → Primitive Assembly

Simple Vertex Shader

input from application (GLSL 1.5)
in vec4 vPosition;
void main(void)
{
    gl_Position = vPosition; \text{ Simple pass-through}
}

built in variable

Use “attribute vec4 vPosition” for GLSL 1.4
Execution Model

Diagram:

- Vertices → Vertex Processor → Clipper and Primitive Assembler → Rasterizer → Fragment Processor → Pixels

- Application

- Shader Program

- Rasterizer → Fragment Shader

- Frame Buffer

- Fragment Color
Simple (Old) Fragment Program

void
main()
{
    gl_FragColor = vec4( 1.0, 0.0, 0.0, 1.0 );
}

Every fragment simply colored red
out vec4 fragcolor;
void main(void)
{
fragcolor = vec4(1.0, 0.0, 0.0, 1.0);
}

Every fragment simply colored red
Data Types

- C types: int, float, bool, uint, double
- Vectors:
  - float vec2, vec3, vec4
  - Also int (ivec), boolean (bvec), uvec, dvec
- Matrices: mat2, mat3, mat4
  - Stored by columns
  - Standard referencing m[row][column]
- C++ style constructors
  - vec3 a = vec3(1.0, 2.0, 3.0)
  - vec2 b = vec2(a)
Pointers

• There are no pointers in GLSL
• We can use C structs which can be copied back from functions
• Because matrices and vectors are basic types they can be passed into and out from GLSL functions, e.g.
  mat3 func(mat3 a)
Qualifiers

- GLSL has many of the same qualifiers such as `const` as C/C++
- Need others due to the nature of the execution model
- Variables can change
  - Once per vertex
  - Once per primitive
  - Once per fragment
  - At any time in the application
- Vertex attributes are interpolated by the rasterizer into fragment attributes
Attribute Qualified

• Attribute-qualified variables can change at most once per vertex
• There are a few built in variables such as gl_Position but most have been deprecated
• User defined (in application program)
  - Use ‘in’ qualifier to get to shader
    - in float temperature
    - in vec3 velocity
Uniform Qualifier

• Variables that are constant for an entire primitive
• Can be changed in application and sent to shaders
• Cannot be changed in shader
• Used to pass information to shader such as the bounding box of a primitive
Varying Qualifier

• Variables that are passed from vertex shader to fragment shader
• Automatically interpolated by the rasterizer
• Old style used the varying qualifier
  ```glsl
  varying vec4 color;
  ```
• Now use **out** in vertex shader and **in** in the fragment shader
  ```glsl
  out vec4 color;
  ```
Example: Vertex Shader

```cpp
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
in vec4 vPosition;
out vec4 color_out;
void main(void)
{
    gl_Position = vPosition;
    color_out = vPosition.x * red;
}
```
in vec4 color_out;
void main(void)
{
    // Now deprecated
    gl_FragColor = color_out;
}

// in latest version use
// out vec4 fragcolor;
// fragcolor = color_out;
User-defined functions

• Similar to C/C++ functions
• Except
  - Cannot be recursive
  - Specification of parameters

returnType MyFunction(in float inputValue, out int outputValue, inout float inAndOutValue);
Passing values

- call by **value-return**
- Variables are copied in
- Returned values are copied back
- Three possibilities
  - in
  - out
  - inout
Operators and Functions

- Standard C functions
  - Trigonometric
  - Arithmetic
  - Normalize, reflect, length

- Overloading of vector and matrix types

```csharp
mat4 a;
vec4 b, c, d;
c = b*a; // a column vector stored as a 1d array
d = a*b; // a row vector stored as a 1d array
```
Swizzling and Selection

- Can refer to array elements by element using [] or selection (.) operator with
  - x, y, z, w
  - r, g, b, a
  - s, t, p, q
  - a[2], a.b, a.z, a.p are the same

- **Swizzling** operator lets us manipulate components

```cpp
vec4 a, b;
a.yz = vec2(1.0, 2.0);
a.xw = b.yy;
```
Programming with OpenGL
Part 4: Color and Attributes
Objectives

• Expanding primitive set
• Adding color
• Vertex attributes
• Uniform variables
OpenGL Primitives

- **GL_POINTS**
- **GL_LINES**
- **GL_LINE_STRIP**
- **GL_LINE_LOOP**
- **GL_TRIANGLES**
- **GL_TRIANGLE_STRIP**
- **GL_TRIANGLE_FAN**
Polygon Issues

- OpenGL will only display triangles
  - Simple: edges cannot cross
  - Convex: All points on line segment between two points in a polygon are also in the polygon
  - Flat: all vertices are in the same plane
- Application program must tessellate a polygon into triangles (triangulation)
- OpenGL 4.1 contains a tessellator
Polygon Testing

- Conceptually simple to test for simplicity and convexity
- Time consuming
- Earlier versions assumed both and left testing to the application
- Present version only renders triangles
- Need algorithm to triangulate an arbitrary polygon
Good and Bad Triangles

• Long thin triangles render badly

• Equilateral triangles render well
• Maximize minimum angle
• Delaunay triangulation for unstructured points
Triangularization

- Convex polygon

- Start with abc, remove b, then acd, ....
Non-convex (concave)
Recursive Division

- There are a variety of recursive algorithms for subdividing concave polygons
Attributes determine the appearance of objects

- Color (points, lines, polygons)
- Size and width (points, lines)
- Stipple pattern (lines, polygons)
- Polygon mode
  - Display as filled: solid color or stipple pattern
  - Display edges
  - Display vertices

Only a few (glPointSize) are supported by OpenGL functions
RGB color

- Each color component is stored separately in the frame buffer
- Usually 8 bits per component in buffer
- Color values can range from 0.0 (none) to 1.0 (all) using floats or over the range from 0 to 255 using unsigned bytes
Smooth Color

• Default is *smooth* shading
  - OpenGL interpolates vertex colors across visible polygons

• Alternative is *flat shading*
  - Color of first vertex determines fill color
  - Handle in shader
Setting Colors

• Colors are ultimately set in the fragment shader but can be determined in either shader or in the application.

• Application color: pass to vertex shader as a uniform variable (next lecture) or as a vertex attribute.

• Vertex shader color: pass to fragment shader as varying variable (next lecture).

• Fragment color: can alter via shader code.