GLSL I

Ed Angel

Professor of Computer Science, Electrical and Computer Engineering, and Media Arts

Director, Arts Technology Center

University of New Mexico

Objectives

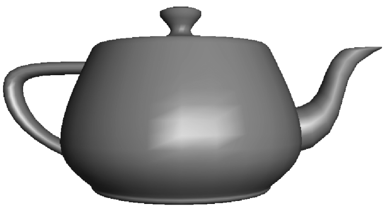
- Shader applications
  - Vertex shaders
  - Fragment shaders
- Programming shaders
  - Cg
  - GLSL

Vertex Shader Applications

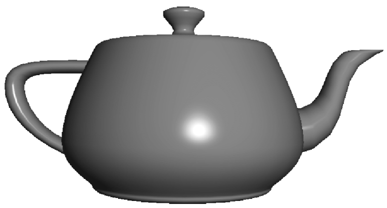
- Moving vertices
  - Morphing
  - Wave motion
  - Fractals
- Lighting
  - More realistic models
  - Cartoon shaders

Fragment Shader Applications

Per fragment lighting calculations



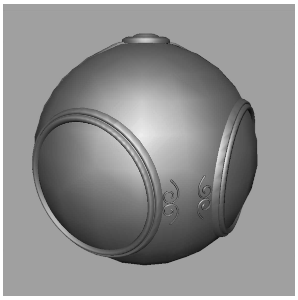
per vertex lighting

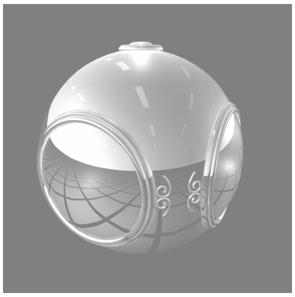


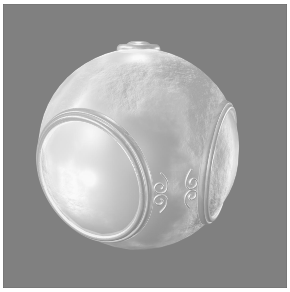
per fragment lighting

Fragment Shader Applications

Texture mapping


smooth shading


environment mapping


bump mapping

Writing Shaders

- First programmable shaders were programmed in an assembly-like manner
- OpenGL extensions added for 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

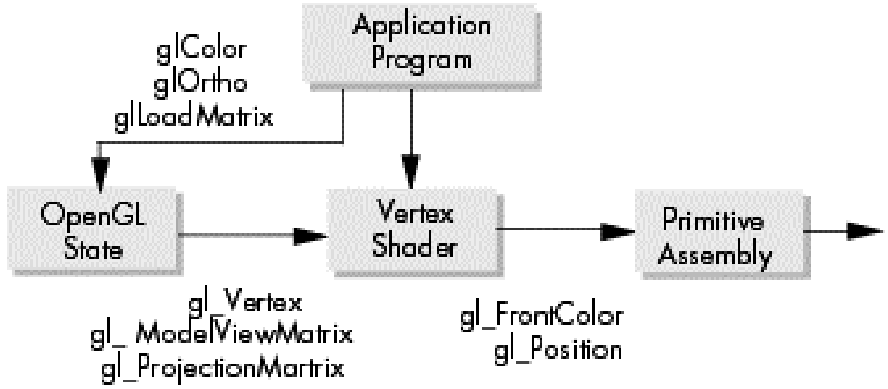
- OpenGL Shading Language
- Part of OpenGL 2.0
- High level C-like language
- New data types
  - Matrices
  - Vectors
  - Samplers
- OpenGL state available through built-in variables

Simple Vertex Shader

```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
void main(void)
{
   gl_Position = gl_ProjectionMatrix
        *gl_ModelViewMartrix*gl_Vertex;

   gl_FrontColor = red;
}
```

Execution Model



Application Program

glColor
glOrtho
glLoadMatrix

OpenGL State

Vertex Shader

Primitive Assembly

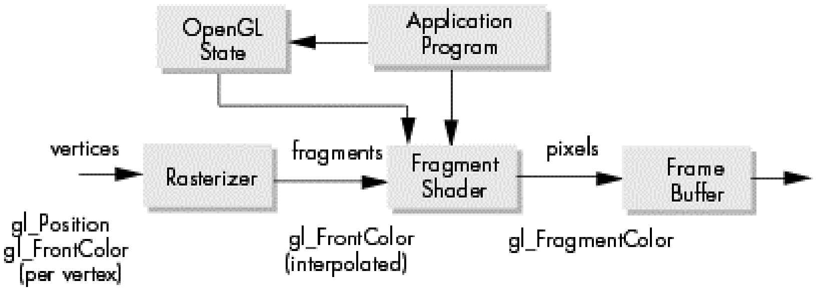
gl_Vertex
gl_ ModelViewMatrix
gl_ProjectionMartrix

gl_FrontColor
gl_Position

Simple Fragment Program

```
void main(void)
{
  gl_FragColor = gl_FrontColor;
}
```

Execution Model



Data Types

- C types: int, float, bool
- Vectors:
  - float vec2, vec 3, vec4
  - Also int (ivec) and boolean (bvec)
- 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 output from GLSL functions, e.g.

  matrix3 func(matrix3 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 primitive
  - Once per vertex
  - Once per fragment
  - At any time in the application
- Vertex attributes are interpolated by the rasterizer into fragment attributes

Attribute Qualifier

- Attribute-qualified variables can change at most once per vertex
  - Cannot be used in fragment shaders
- Built in (OpenGL state variables)
  - **`gl_Color`**
  - **`gl_ModelViewMatrix`**
- User defined (in application program)
  - **`attribute float temperature`**
  - **`attribute vec3 velocity`**

Uniform Qualified

- Variables that are constant for an entire primitive
- Can be changed in application outside scope of **`glBegin`** and **`glEnd`**
- Cannot be changed in shader
- Used to pass information to shader such as the bounding box of a primitive

Varying Qualified

- Variables that are passed from vertex shader to fragment shader
- Automatically interpolated by the rasterizer
- Built in
  - Vertex colors
  - Texture coordinates
- User defined
  - Requires a user defined fragment shader

Example: Vertex Shader

```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
varying vec3 color_out;
void main(void)
{
  gl_Position =
   gl_ModelViewProjectionMatrix*gl_Vertex;
  color_out = red;
}
```

Required Fragment Shader

```
varying vec3 color_out;
void main(void)
{
  gl_FragColor = color_out;
}
```

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

```
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

```
vec4 a;
a.yz = vec2(1.0, 2.0);
```