Lecture 09: Shaders (Part 1)

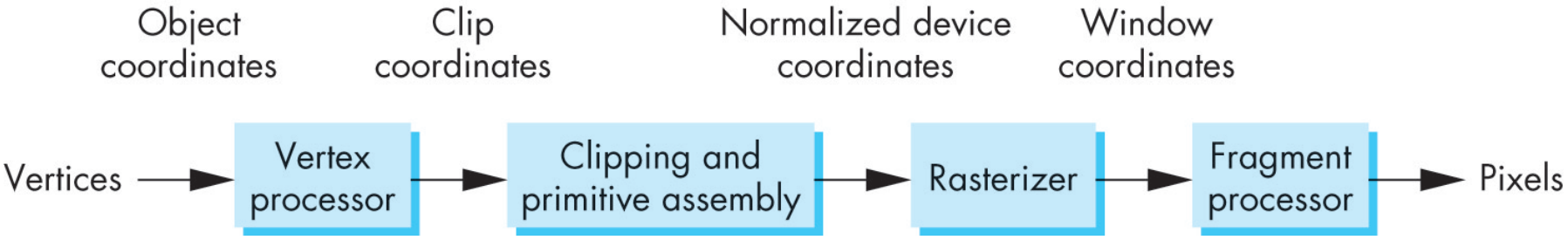
CSE 40166 Computer Graphics

Peter Bui

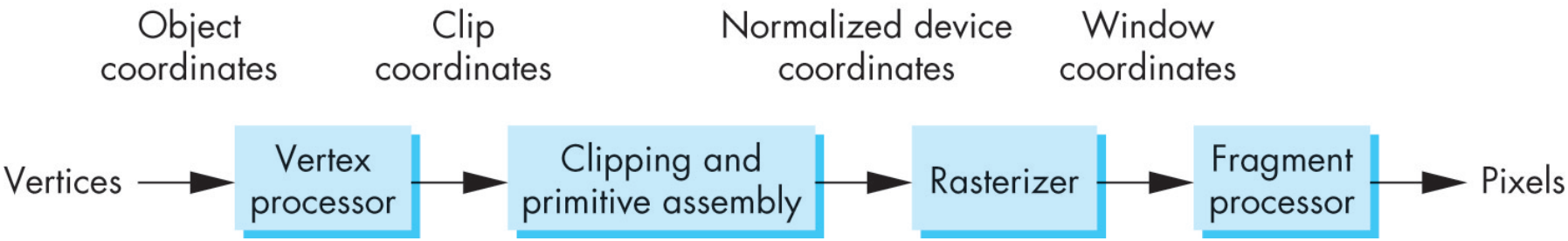
University of Notre Dame, IN, USA

November 9, 2010

OpenGL Rendering Pipeline



OpenGL Rendering Pipeline (Pseudo-Code)




```
1  for gl_Vertex in GL_VERTICES:
2      // Process vertices
3      gl_Position   = gl_ModelViewProjectionMatrix * gl_Vertex
4      gl_FrontColor = gl_Color;
5
6      // Clip and assemble
7      primitive = Assemble(gl_Position, gl_FrontColor)
8      if not Clipped(primitive):
9          PRIMITIVES.append(primitive)
10
11 // Rasterize primitives into fragments
12 FRAGMENTS = Rasterize(PRIMITIVES)
13
14 for fragment in FRAGMENTS:
15     // Process fragments
16     gl_FragColor = ApplyLighting(gl_Color)
```

OpenGL Rendering Pipeline (Pros and Cons)

Pros

Has the following features:

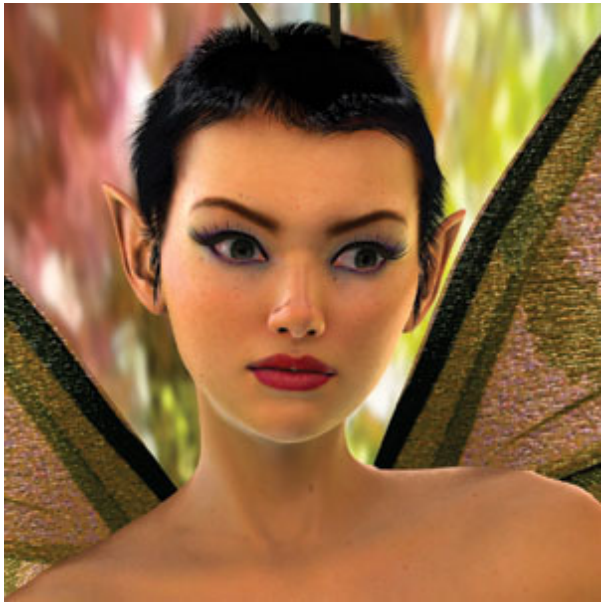
- **Concurrency**: Each vertex and fragment can be processed in parallel.
- **Fast**: Hardware implements of common functions.
- **Good Enough**: Modified Phong lighting model yields adequate images.

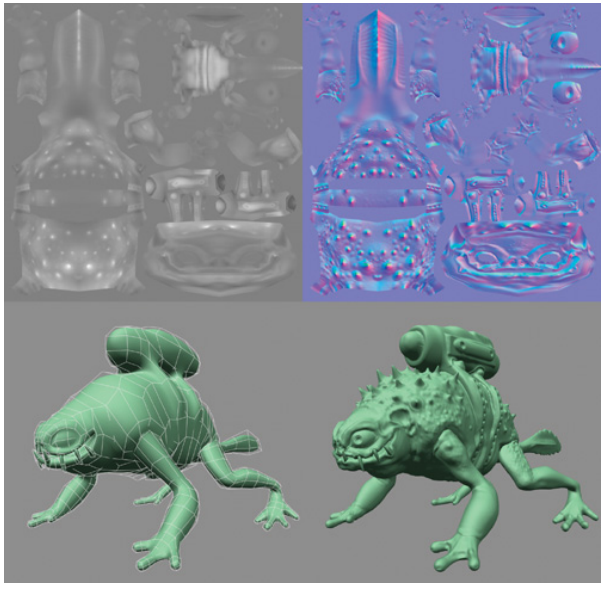
Cons

Hard to achieve the following:

- **Photorealism**: Skin, fabrics, fluids, translucent materials, refraction.
- **Non-photorealism**: Simulate brush strokes or cartoonlike shading effects.

Difficult to Render

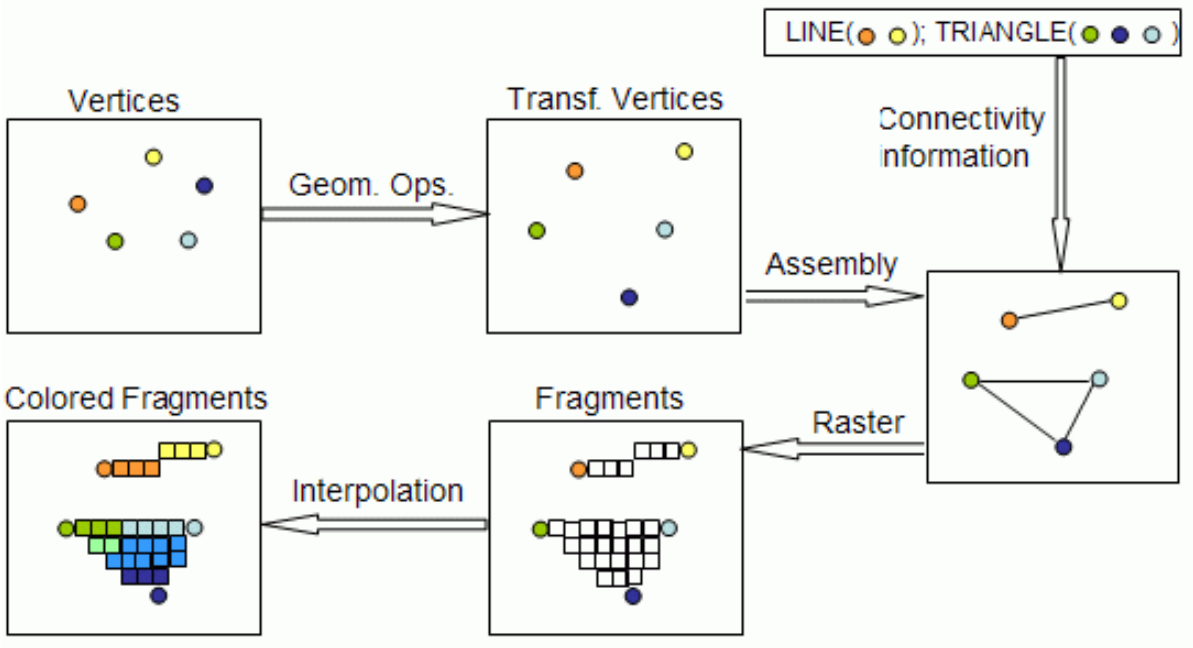




From Fixed to Programmable (Problem)

Problem

Although OpenGL pipeline is fast it is limited due to its **fixed** functionality.

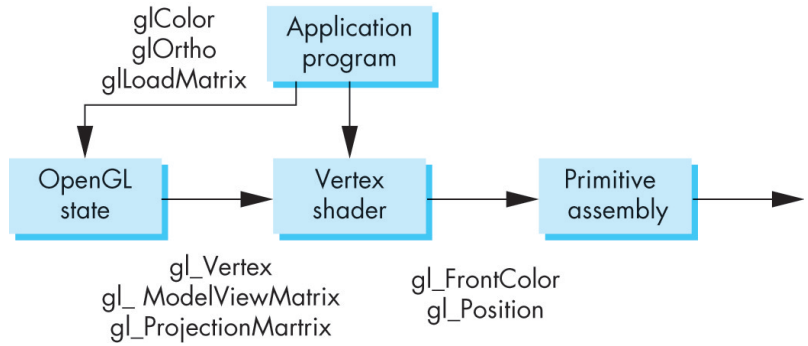


From Fixed to Programmable (Solution)

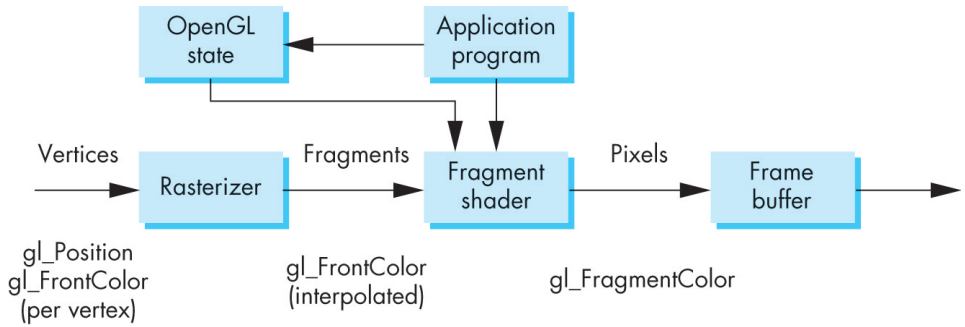
Solution

Allow developers to replace fixed functions with their own **programmable** *shaders*.

- **Vertex Shader**: Replace for vertex transformation stage.



- **Fragment Shader**: Replace fragment texturing and coloring stage.



From Fixed to Programmable (Responsibilities)

Vertex Shader

- Vertex position using ModelView and Projection matrices.
- Lighting per vertex or per pixel.
- Color and texture computation.
- *Must at least set* `gl_Position` *variable.*

Fragment Shader

- Computing colors, texture coordinates per pixel.
- Texture application.
- Fog computation.
- *Output result by setting* `gl_FragColor`*.*

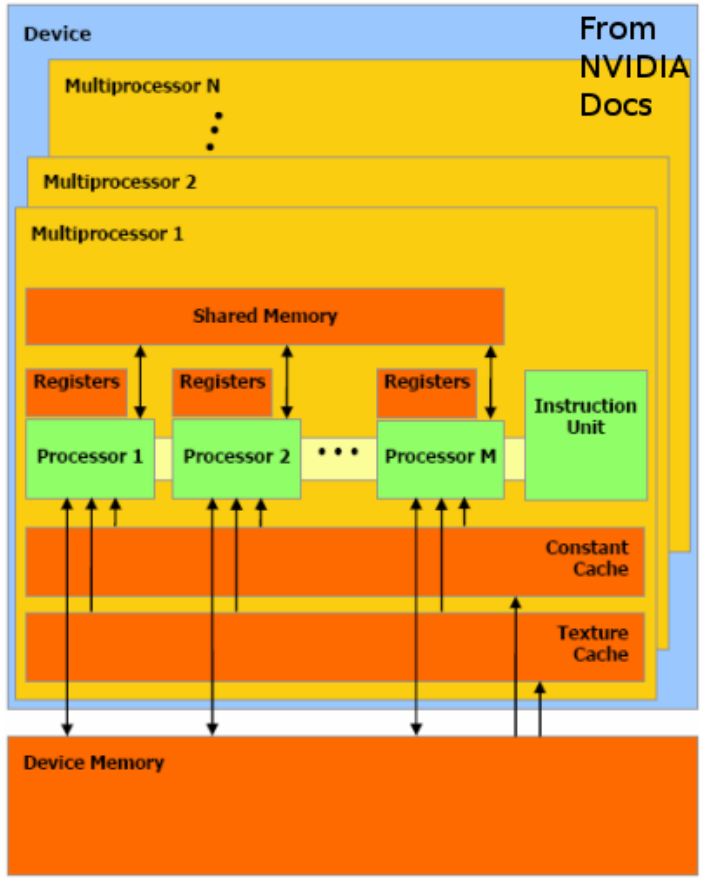
If we use a shader we must re-program ALL functionality.

From Fixed to Programmable (Architecture)

Observation

Vertex and Fragment processor is a parallel computing unit.

- **Data Parallelism**: Each data item can be processed independently.
- **SPMD**: Single Program Multiple Data.
- **Stream Processors**: Many simple processors attached to fixed computational units.



GLSL: OpenGL Shading Language

- Based on the C programming language (with some C++isms).
- Shader application is run per vertex or per fragment with results set in particular global variables.

Example: Pass through vertex shader

```
1  void main()
2  {
3      gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
4  }
```

Example: Pass through fragment shader

```
1  void main()
2  {
3      gl_FragColor = gl_Color;
4  }
```

GLSL: Data Types

Data Types

- **Scalar**: Floating point (`float`), integer (`int`), boolean `bool`).
- **Vectors**: One dimensional arrays with *swizzling operator*.
- **Matrices**: Square two-dimensional arrays with overloaded operators.
- **Arrays and Structs**: Same as in C.

Example: Data types

```
1  void main()
2  {
3      float f[16];
4      int i = 0;
5      vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
6      mat4 m = gl_ProjectionMatrix;
7      float r, g, b;
8
9      r = red.r;
10     g = red.g;
11     b = red.b;
12     red.rgb = vec3(0.5, 0.0, 0.0);
13 }
```

GLSL: Data Qualifiers

- **const**: Variable is unchangeable by the shader.
- **attribute**: Variable changes at most once per vertex in vertex shader.
- **uniform**: Variable set in application program for an entire batch of primitives.
- **varying**: Provide mechanism for conveying data from a vertex shader to fragment shader.

Example: scaling vertex shader

```
1  uniform float ElapsedTime;
2
3  void main()
4  {
5      float s;
6
7      s = 1.0 + 0.5*sin(0.005*ElapsedTime);
8
9      gl_Position   = gl_ModelViewProjectionMatrix*(vec4(s, s, s, 1.0)*gl_Vertex);
10     gl_FrontColor = gl_Color;
11 }
```

GLSL: Operators and Functions

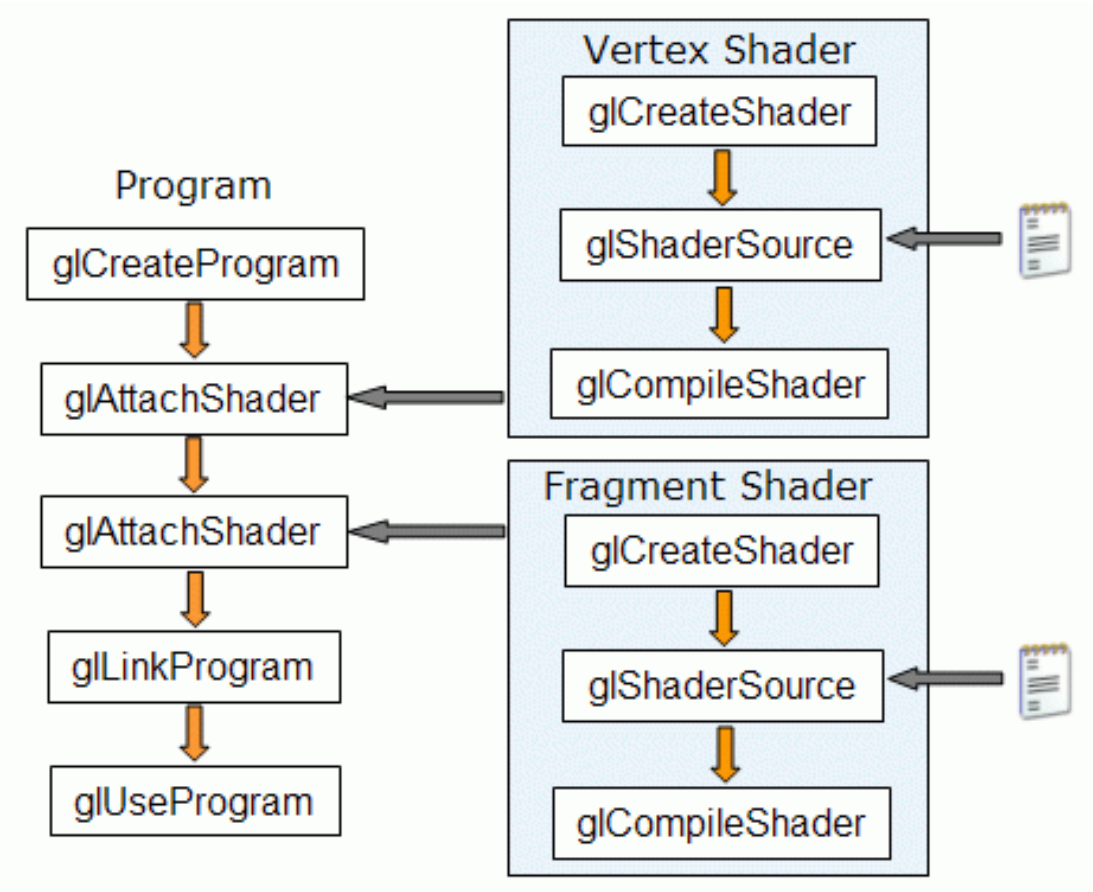
- Matrix-vector operations behave as expected.
- *Swizzling* operator allows for convenient access to elements of a vector: `x,y,z,w; r,g,b,a; s,t,p,q`.
- Can access built-in functions:
  - **Trigonometric**: `asin, acos, atan`.
  - **Mathematical**: `pow, log2, sqrt, abs, max, min`.
  - **Geometric**: `length, distance, dot, normalize, reflect`.
  - **Fixed**: `ftransform`.
- Can make your own functions, but must qualify variables as `in`, `out`, `inout`.

Example: Pass through vertex shader (ftransform)

```
1  void main()
2  {
3      gl_Position = ftransform();
4  }
```

GLSL: Attaching, Compiling, and Linking Shaders

To use shaders in an OpenGL program, we need to load them into memory, attach them, compile them and link the program.



GLSL: GLEW

In order to load, compile, and use shaders, we must use OpenGL extensions. To manage the use of these extensions, we use the `GLEW` library.

```
1  // Include GLEW before GLUT
2  #include <GL/glew.h>
3  #include <GL/glut.h>
4
5  initialize()
6  {
7      GLenum result;
8      // Note: Must have OpenGL context first!
9
10     // Initialize and check GLEW
11     result = glewInit();
12     if (result != GLEW_OK) {
13         fprintf(stderr, "unable to initialize glew: %s\n",
14         glewGetErrorString(result));
15         exit(EXIT_FAILURE);
16     }
17     // GLSL require at least OpenGL 2.0
18     if (!glewIsSupported("GL_VERSION_2_0")) {
19         fprintf(stderr, "OpenGL 2.0 is not supported\n");
20         exit(EXIT_FAILURE);
21     }
22 }
```

GLSL: Attaching, Compiling, and Linking Shaders (Code)

```
1  GLuint vert_id, frag_id, prog_id;
2  void load_shaders(const char *vert_path, const char *frag_path)
3  {
4      char *vert_src, *frag_src;
5
6      // Create program, vertex and fragment shaders IDs
7      prog_id = glCreateProgram();
8      vert_id = glCreateShader(GL_VERTEX_SHADER);
9      frag_id = glCreateShader(GL_FRAGMENT_SHADER);
10
11     // Read shader source into memory buffers
12     vert_src = read_shader(vert_path);
13     frag_src = read_shader(frag_path);
14
15     // Bind shader source
16     glShaderSource(vert_id, 1, (const char **)&vert_src, NULL);
17     glShaderSource(frag_id, 1, (const char **)&frag_src, NULL);
18
19     // Compile shaders
20     glCompileShader(vert_id);
21     glCompileShader(frag_id);
22
23     // Attach shaders to program
24     glAttachShader(prog_id, vert_id);
25     glAttachShader(prog_id, frag_id);
26
27     // Link program
28     glLinkProgram(prog_id);
29 }
```

GLSL: Using Shaders and Cleaning up

```
1  void display()
2  {
3      glUseProgram(prog_id);  // Use shader program
4      // Render code
5      glUseProgram(0);        // Disable shader program
6  }
7
8  void cleanup()
9  {
10     // Detach shaders from program
11     glDetachShader(prog_id, vert_id);
12     glDetachShader(prog_id, frag_id);
13
14     // Delete shader (must be detached)
15     glDeleteShader(vert_id);
16     glDeleteShader(frag_id);
17
18     // Delete program
19     glDeleteProgram(prog_id);
20 }
```

GLSL: Debugging

```
1  void print_shader_log(GLuint id) { // Print shader log
2      char *buffer;
3      GLint buffer_written;
4      GLint buffer_size;
5
6      glGetShaderiv(id, GL_INFO_LOG_LENGTH, &buffer_size);
7      if (buffer_size > 0) {
8          buffer = malloc(sizeof(char) * buffer_size);
9          glGetShaderInfoLog(id, buffer_size, &buffer_written, buffer);
10         fprintf(stderr, "Shader %u Log: %s\n", id, buffer);
11         free(buffer);
12     }
13 }
14 void print_program_log(GLuint id) { // Print program log
15     char *buffer;
16     GLint buffer_written;
17     GLint buffer_size;
18
19     glGetProgramiv(id, GL_INFO_LOG_LENGTH, &buffer_size);
20     if (buffer_size > 0) {
21         buffer = malloc(sizeof(char) * buffer_size);
22         glGetProgramInfoLog(id, buffer_size, &buffer_written, buffer);
23         fprintf(stderr, "Program %u Log: %s\n", id, buffer);
24         free(buffer);
25     }
26 }
27
28 print_shader_log(vert_id);
29 print_shader_log(frag_id);
30 print_program_log(prog_id);
```

Example: Red vertex shading

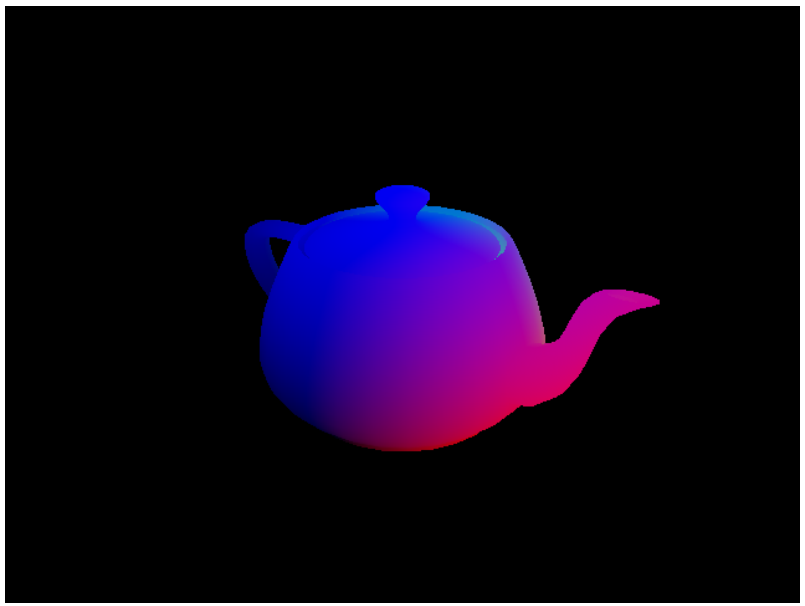
Vertex Shader

```
1  // Color all vertices red
2  const vec4 RedColor = vec4(1.0, 0.0, 0.0, 1.0);
3
4  void main()
5  {
6      gl_Position   = ftransform();
7      gl_FrontColor = RedColor;
8  }
```

Example: Color based on vertex

Vertex Shader

```
1  // Color all vertices based on normalized vertex coordinates
2  const vec4 RedColor = vec4(1.0, 0.0, 0.0, 1.0);
3
4  void main()
5  {
6      gl_Position    = ftransform();
7      gl_FrontColor = normalize(gl_Vertex);
8  }
```



Example: Scaling over time

Vertex Shader

```
1  // Scale vertices over time
2  uniform float ElapsedTime;
3
4  void main()
5  {
6      float s;
7
8      s = 1.0 + 0.5*sin(0.005*ElapsedTime);
9
10     gl_Position   = gl_ModelViewProjectionMatrix*(vec4(s, s, s, 1.0)*gl_Vertex);
11     gl_FrontColor = gl_Color;
12 }
```

OpenGL

```
1  void set_elapsed_time()
2  {
3      GLint etloc;
4
5      /* Set shader uniform variable */
6      etloc = glGetUniformLocation(prog_id, "ElapsedTime");
7      glUniform1f(etloc, glutGet(GLUT_ELAPSED_TIME));
8  }
```

Example: Flatten and Wave

Vertex Shader

```
1  uniform float ElapsedTime;
2
3  void main()
4  {
5      vec4 v = vec4(gl_Vertex);
6
7      v.y = sin(5.0 * v.x + ElapsedTime*0.01)*0.25;
8
9      gl_Position   = gl_ModelViewProjectionMatrix * v;
10     gl_FrontColor = gl_Color;
11 }
```

Example: Toon Shading

Vertex Shader

```
1  varying vec3 Normal;
2
3  void main()
4  {
5      Normal = gl_NormalMatrix * gl_Normal;
6      gl_Position = ftransform();
7  }
```

Fragment Shader

```
1  varying vec3 Normal;
2
3  void main()
4  {
5      float intensity;
6      vec4 color;
7      vec3 n = normalize(Normal);
8
9      intensity = dot(vec3(gl_LightSource[0].position), n);
10
11     if (intensity > 0.95)      color = vec4(1.0, 0.5, 0.5, 1.0);
12     else if (intensity > 0.50) color = vec4(0.6, 0.3, 0.3, 1.0);
13     else if (intensity > 0.25) color = vec4(0.4, 0.2, 0.2, 1.0);
14     else                       color = vec4(0.2, 0.1, 0.1, 1.0);
15
16     gl_FragColor = color;
17 }
```

Resources

- **OpenGL Shading Language Specs**: http://www.opengl.org/documentation/glsl/
- **Lighthouse 3D GLSL Tutorial**: http://www.lighthouse3d.com/opengl/glsl/index.php?intro
- **Neon Helium GLSL Tutorial**: http://nehe.gamedev.net/data/articles/article.asp?article=21
- **Clockwork Coders GLSL Tutorial**: http://www.clockworkcoders.com/oglsl/tutorials.html
- **Swiftless Tutorials**: http://www.swiftless.com/glsltuts.html