OpenGL Shaders

Brian Dicks

What are shaders?

- Shaders are programs that run on your graphics hardware
  - They are different from CPU programs, and are usually not general purpose
- There are five types of shaders
  - Vertex shaders
  - Fragment shaders (also known as pixel shaders)
  - Geometry shaders
  - Hull and domain shaders (for tessellation)
  - Compute shaders
- We will cover vertex and fragment shaders

Vertex Shaders

- Graphics hardware is very powerful
  - Can perform many vector operations in parallel
- Transferring data between the CPU and GPU can be slow
  - Can store vertexes in VRAM
- Can we get the GPU to special vertex transformations, such as skeletal animation, for us?

Vertex Shader

- Vertex shaders operate per-vertex
  - Applied before primitives are created
- Can't create new vertices
- One vertex at a time
- Some hardware supports texture access in VS
- Only required output of a VS is the vertex's final position
- Outputs are interpolated

Vertex Shader

- Some applications for VS
  - Skeletal animation
  - Wave effect
  - Passing additional information for fragment shaders

OpenGL Pipeline

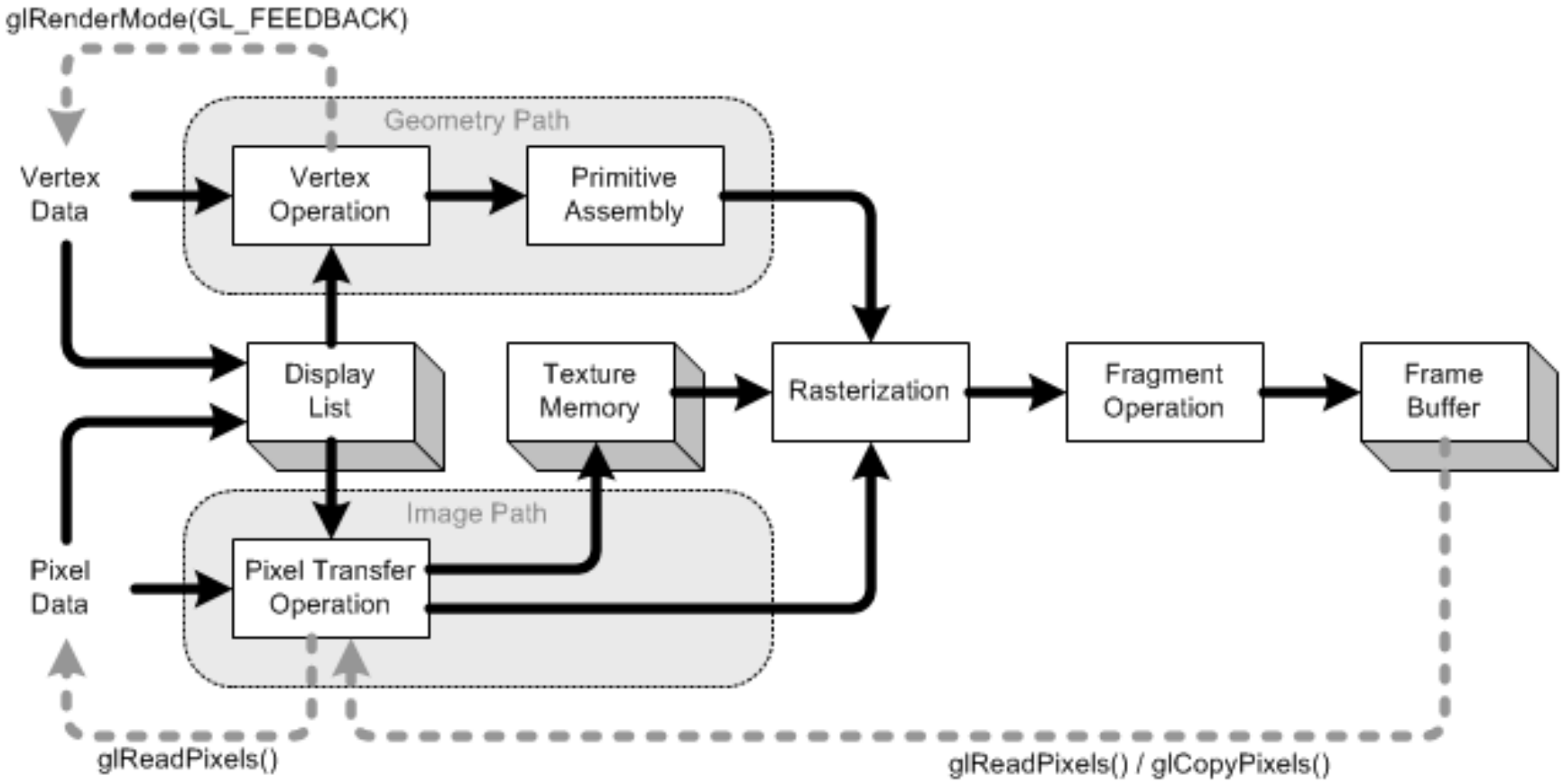

Image source: Songho.ca

Fragments

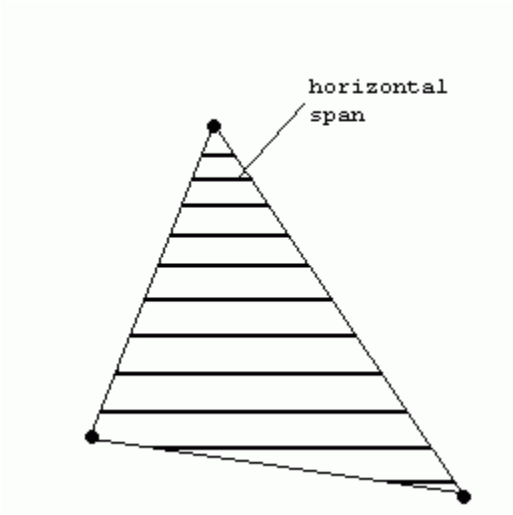

Image source: DevMaster.net

- What is a fragment?
  - Horizontal line after rasterization
  - Many OpenGL testing operations are done at the fragment level
    - Scissor test, alpha test, stencil test, depth test
  - The OpenGL pipeline has been modified to allow us to perform operations per fragment

Fragment Shaders

- Fragment shaders use information from the vertex shader
- VS and FS work together to achieve effects
- This is where the ‘drawing’ is performed, any per pixel effect is done here
  - Texturing, lighting, etc.
- Must output a color

Fragment Shaders

- Uses of FS
  - Phong shading
  - Normal mapping
  - X-ray effect
  - Refraction

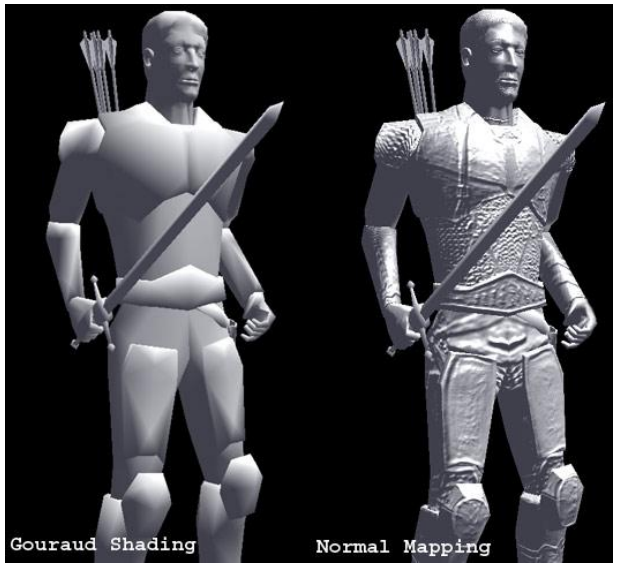

Image from 3dkingdoms.com

Image from ozone3d.net

Other Shaders

- Geometry shaders
  - These are executed after vertex shaders
  - Geometry shaders turn vertices into primitives
  - Can emit new vertices
  - Optional
- Tessellation related shaders
  - After vertex shaders, but before geometry shaders
  - Used to create geometry from patches
  - Optional
  - More efficient than using geometry shaders for the same purpose

Other Shaders

- Compute Shaders
  - Not based on the graphics pipeline
  - Used for general purpose programming
  - Programming model is somewhat restrictive
    - Some programs do not map well to this model
    - Highly parallel programs with no branching work well
  - CUDA and OpenCL have features that allow them to share resources with OpenGL
  - Direct3D has DirectCompute

Writing Shaders

- If you are on Windows, the methods to create shaders are missing from the implemention
- You need to either use OpenGL extension loading (the hard way), or use the GLee library (much easier)
  - http://elf-stone.com/glee.php
- You can add the .c and .h files for GLee to your project
  - #include “GLee.h”

Writing Shaders

- Data type qualifiers
  - attribute – per-vertex input
  - uniform – per object
    - Must be outside of glEnd/glBegin
  - varying – communication between shaders
  - const
- Common types
  - float, vec2, vec3, vec4
  - mat2, mat3, mat4
  - sampler1D, sampler2D, sampler3D
  - samplerCube
  - Others exist (such as for integers)
    - May be slower than using FP depending on hardware

Writing Shaders

- Data type qualifiers
  - attribute – per-vertex input
  - uniform – per object
    - Must be outside of glEnd/glBegin
  - varying – communication between shaders
  - const
- Common types
  - float, vec2, vec3, vec4
  - mat2, mat3, mat4
  - sampler1D, sampler2D, sampler3D
  - samplerCube
  - Others exist (such as for integers)
    - May be slower than using FP depending on hardware

Example Vertex Shader

```
// uniform float intensityScale;
// attribute vec4 vertexColor;
varying float intensity; // for writing

void main()
{
      vec3 lightDir =
            normalize(vec3(gl_LightSource[0].position));
      intensity = dot(lightDir,gl_Normal);

      gl_Position = ftransform();
}
```

Example Fragment Shader

```
varying float intensity; // read only
// uniform float intensityScale;
// can't use attributes here

void main()
{
        vec4 color;
        if (intensity > 0.95)

                color = vec4(1.0,0.5,0.5,1.0);
        else if (intensity > 0.5)
                color = vec4(0.6,0.3,0.3,1.0);
        else if (intensity > 0.25)
                color = vec4(0.4,0.2,0.2,1.0);
        else
                color = vec4(0.2,0.1,0.1,1.0);
        gl_FragColor = color;

}
```

So how do we get OpenGL to use our shaders?

- First, we need to generate an ID
  - Gluint glCreateShader(GL_VERTEX_SHADER);
  - Gluint glCreateShader(GL_FRAGMENT_SHADER);
- Now we tell OpenGL where to find the source
  - glShaderSource(shaderId, 1, &char_array, 0);
- Compile
  - glCompileShader(shaderId);
- Check for errors
- Vertex and fragment shaders are compiled separately

So how do we get OpenGL to use our shaders?

- Checking for errors
  - int status;
  - glGetShaderiv(shader_id, GL_COMPILE_STATUS, &status)
  - if status is GL_FALSE, compile failed, get the error message
    - Get size of error log
      - glGetShaderiv(shader_id,GL_INFO_LOG_LENGTH,&log_length);
    - Allocate GLchar array (errorLog) of log_length
    - Get the actual log
      - glGetShaderInfoLog(shader_id, log_length, &dummy_int, errorLog)
    - print
    - Free array

Now we must link our shaders into a program

- Generate program ID
  - GLuint glCreateProgram()
- Tell linker about the programs
  - glAttachShader(program_id, vertex_shader_id)
    glAttachShader(program_id, fragment_shader_id)
- Link
  - glLinkProgram(program_id)
- Check if there is an error
  - glGetProgramiv(gProgram,GL_LINK_STATUS,&result)
- Getting the error message is similar
  - glGetProgramiv instead of glGetShaderiv
  - glGetProgramInfoLog instead of glGetShaderInfoLog

Getting OpenGL to use our shader

- Activate a program with
  - glUseProgram(program_id)
    - Must be outside of glBegin()/glEnd()
- We can go back to the fixed-function pipeline
  - glUseProgram(0);
- Using a uniform variable
  - GLint glGetUniformLocation(GLuint program, const GLchar *name);
  - glUniform3f(uniform_id, float x, float y, float z)
- Using an attribute
  - GLint glGetAttribLocation(program_id, const GLchar* name)
  - glAttribute3f(attribute_id, float x, float y, float z);

Accessing textures

- Must use samplers
- Simply uniform integer variables that correspond to the texture unit
- Default to 0
- Function to access sampler
  - texture2D(sampler_var, texture coordinates)

Questions?