Shaders and GLSL

Prof. George Wolberg
Dept. of Computer Science
City College of New York
Objectives

• Introduce shaders
  • Vertex shaders
  • Fragment shaders
    - Introduce a standard program structure
• Initialization steps and program structure
• Review sample shaders
Graphics Pipeline

- Vertices stream into vertex processor and are transformed into new vertices
- These vertices are collected to form primitives
- Primitives are rasterized to form fragments
- Fragments are colored by fragment processor
Simplified Pipeline Model

Application → GPU Data Flow → Framebuffer

Vertices → Vertex Processing → Rasterizer → Fragment Processing → Pixels

- Vertex Shader
- Fragment Shader
Execution Model

Application Program (C++)

Shader Program

Vertex data

GPU

glDrawArrays

Vertex Shader (GLSL)

Primitive Assembly

to Rasterizer
Execution Model

- Application Program (C++)
- Shader Program
- Rasterizer
- Fragment Shader (GLSL)
- Frame Buffer
- Fragment
- Fragment Color

Angel/Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012
Writing Shaders

• As of OpenGL 3.1, application programs must provide shaders
  - Application programs reside on CPU
  - Shader programs reside on GPU
• OpenGL extensions added for vertex and fragment shaders
• Shaders are written with the OpenGL Shading Language (GLSL)
GLSL:
OpenGL Shading Language

• Part of OpenGL 2.0 and up
• High level C-like language
• New data types
  - Matrices (mat2, mat3, mat4)
  - Vectors (vec2, vec3, vec4, ...)
  - Samplers (sampler1D, sampler2D, ...)
• New qualifiers: in, out, uniform
• Similar to Nvidia’s Cg and Microsoft HLSL
• New OpenGL functions to compile, link, and get information to shaders
Differences between GLSL and C

• Matrix and vector types are built into GLSL
  - they can be passed into and output from GLSL functions, e.g. mat3 func(mat3 a)

• GLSL is designed to be run on massively parallel implementations
  - Recursion is not allowed in GLSL
  - No pointers in GLSL
  - Precision requirements for floats are not as strict as IEEE standards that govern C implementations
GLSL Data Types

- Scalar types: float, int, bool
- Vector types: vec2, vec3, vec4
  ivec2, ivec3, ivec4
  bvec2, bvec3, bvec4
- Matrix types: mat2, mat3, mat4
- Texture sampling: sampler1D, sampler2D, sampler3D, samplerCube
- C++ Style Constructors
  
  \[
  \text{vec3 } a = \text{vec3}(1.0, 2.0, 3.0);
  \]
Qualifiers (1)

• GLSL has many of the same qualifiers 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
Qualifiers (2)

• **in, out**
  - Copy vertex attributes and other variable into and out of shaders

```
in  vec2 texCoord;
out  vec4 color;
```

• **uniform**
  - shader-constant variable from application

```
uniform float time;
uniform  vec4 rotation;
```
Simple Vertex Shader

```
in vec4 vPosition;
void main(void)
{
    gl_Position = vPosition;
}
```

- Input from application; may use `attribute` instead of `in`
- Must link to variable in application
- Built-in variable
Simple Fragment Program

```c
void main(void)
{
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```
Attribute Qualifier

• 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` or `attribute` qualifier to get to shader
    - `in float temperature`
    - `attribute vec3 velocity`
Varying Qualified

- Variables that are passed from vertex shader to fragment shader
- Automatically interpolated by the rasterizer
- Old style used the varying qualifier
  ```
  varying vec4 color;
  ```
- Now use `out` in vertex shader and `in` in the fragment shader
  ```
  out vec4 color;
  ```
Attribute and Varying Qualifiers

- Starting with GLSL 1.5 attribute and varying qualifiers have been replaced by in and out qualifiers
- No changes needed in application
- Vertex shader example:

```glsl
#version 1.4
attribute vec3 vPosition;
varying vec3 color;

#version 1.5
in vec3 vPosition;
out vec3 color;
```
Uniform Qualified

- 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
Built-in Variables

• `gl_Position`
  - (required) output position of current vertex

• `gl_PointSize`
  - pixel width/height of the point being rasterized

• `gl_FragCoord`
  - input fragment position

• `gl_FragDepth`
  - input depth value in fragment shader
Simple Vertex Shader

```
#version 450

in  vec4 a_Position;
in  vec4 a_Color;
out vec4 color;

void main()
{
    color = a_Color;
    gl_Position = a_Position;
}
```
#version 450

in vec4 color;
out vec4 fColor; // fragment’s final color

void main()
{
    fColor = color;
    // OR: gl_FragColor = color_out;
}
Operators and Functions

• Standard C functions
  - Arithmetic: sqrt, power, abs
  - Trigonometric: sin, asin
  - Graphical: length, reflect

• Overloading of vector and matrix types

```c
mat4 a;
vec4 b, c, d;
c = b*a; // a row vector stored as a 1D array
d = a*b; // a column 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
cvec4 a;
a.yz = vec2(1.0, 2.0);
```
Programming with OpenGL: More GLSL

Prof. George Wolberg
Dept. of Computer Science
City College of New York
Objectives

- Coupling shaders to applications
  - Reading
  - Compiling
  - Linking
- Vertex Attributes
- Setting up uniform variables
- Example applications
Getting Your Shaders into OpenGL

- Shaders need to be compiled and linked to form an executable shader program
- OpenGL provides the compiler and linker
- A program must contain
  - vertex and fragment shaders
  - other shaders are optional

These steps need to be repeated for each type of shader in the shader program

- Create Program: `glCreateProgram()`
- Create Shader: `glCreateShader()`
- Load Shader Source: `glShaderSource()`
- Compile Shader: `glCompileShader()`
- Attach Shader to Program: `glAttachShader()`
- Link Program: `glLinkProgram()`
- Use Program: `glUseProgram()`
Linking Shaders with Application

- Read shaders
- Compile shaders
- Create a program object
- Link everything together
- Link variables in application with variables in shaders
  - Vertex attributes
  - Uniform variables
Program Object

• Container for shaders
  - Can contain multiple shaders
  - Other GLSL functions

Gluint program = glCreateProgram();

// define shader objects here
glUseProgram (program);
glLinkProgram (program);
Reading a Shader

• Shaders are added to the program object and compiled
• Usual method of passing a shader is as a null-terminated string using the function `glShaderSource`
• If the shader is in a file, we can write a reader to convert the file to a string
Adding a Vertex Shader (1)

```cpp
GLuint LoadShaders(const char * vertex_file_path, const char * fragment_file_path) {

    // Create the shaders
    GLuint VertexShaderID = glCreateShader(GL_VERTEX_SHADER);
    GLuint FragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER);

    // Read the Vertex Shader code from the file
    std::string VertexShaderCode;
    std::ifstream VertexShaderStream(vertex_file_path, std::ios::in);
    if (VertexShaderStream.is_open()) {
        std::string Line = "";
        while (getline(VertexShaderStream, Line))
            VertexShaderCode += "\n" + Line;
    VertexShaderStream.close();
    }

    // Read the Fragment Shader code from the file
    std::string FragmentShaderCode;
    std::ifstream FragmentShaderStream(fragment_file_path, std::ios::in);
    if (FragmentShaderStream.is_open()) {
        std::string Line = "";
        while (getline(FragmentShaderStream, Line))
            FragmentShaderCode += "\n" + Line;
    FragmentShaderStream.close();
    }
}
```
GLint Result = GL_FALSE;
int InfoLogLength;

// Compile Vertex Shader
printf("Compiling shader : %s\n", vertex_file_path);
char const * VertexSourcePointer = VertexShaderCode.c_str();
glShaderSource (VertexShaderID, 1, &VertexSourcePointer , NULL);
glCompileShader(VertexShaderID);

// Check Vertex Shader
glGetShaderiv(VertexShaderID, GL_COMPILE_STATUS, &Result);
glGetShaderiv(VertexShaderID, GL_INFO_LOG_LENGTH, &InfoLogLength);
std::vector<char> VertexShaderErrorMessage(InfoLogLength);
glGetShaderInfoLog(VertexShaderID, InfoLogLength, NULL, &VertexShaderErrorMessage[0]);
fprintf(stdout, "%s\n", &VertexShaderErrorMessage[0]);

// Compile Fragment Shader
printf("Compiling shader : %s\n", fragment_file_path);
char const * FragmentSourcePointer = FragmentShaderCode.c_str();
glShaderSource (FragmentShaderID, 1, &FragmentSourcePointer , NULL);
glCompileShader(FragmentShaderID);

// Check Fragment Shader
glGetShaderiv(FragmentShaderID, GL_COMPILE_STATUS, &Result);
glGetShaderiv(FragmentShaderID, GL_INFO_LOG_LENGTH, &InfoLogLength);
std::vector<char> FragmentShaderErrorMessage(InfoLogLength);
glGetShaderInfoLog(FragmentShaderID, InfoLogLength, NULL, &FragmentShaderErrorMessage[0]);
fprintf(stdout, "%s\n", &FragmentShaderErrorMessage[0]);
Adding a Vertex Shader (3)

```c

// Link the program
fprintf(stdout, "Linking program\n");
GLuint ProgramID = glCreateProgram();
glAttachShader(ProgramID, VertexShaderID);
glAttachShader(ProgramID, FragmentShaderID);
glLinkProgram(ProgramID);

// Check the program
glGetProgramiv(ProgramID, GL_LINK_STATUS, &Result);
glGetProgramiv(ProgramID, GL_INFO_LOG_LENGTH, &InfoLogLength);
std::vector<char> ProgramErrorMessage( max(InfoLogLength, int(1)) );
glGetProgramInfoLog(ProgramID, InfoLogLength, NULL, &ProgramErrorMessage[0]);
fprintf(stdout, "%s\n", &ProgramErrorMessage[0]);

glDeleteShader(VertexShaderID);
glDeleteShader(FragmentShaderID);

return ProgramID;
```

A Simpler Way

• Qt created a routine to make it easy to load shaders
  
  ```cpp
  #include <QGLShaderProgram>
  QGLShaderProgram program;
  program.addShaderFromSourceFile(QGLShader::Vertex, ":/vshader.glsl");
  program.addShaderFromSourceFile(QGLShader::Fragment, ":/fshader.glsl");
  ```

• Fails if shaders don’t compile, or program doesn’t link

• Add shader programs in qrc file:

  ```xml
  <RCC>
      <qresource prefix="/">
          <file>vshader.glsl</file>
          <file>fshader.glsl</file>
      </qresource>
  </RCC>
  ```
Associating Shader Variables and Data

- Vertex attributes are named in the shaders
- Linker forms a table
- Application can get index from table and tie it to an application variable
- Similar process for uniform variables
Vertex Attribute Example

GLuint positionID = glGetAttribLocation( program, "a_Position" );
glEnableVertexAttribArray( positionID );
glVertexAttribPointer(positionID, // attribute at location positionID
    2, // size
    GL_FLOAT, // type
    GL_FALSE, // normalized?
    0, // stride
    (void *) 0 // array buffer offset
);
Uniform Variable Example

GLint angleID; // location of angle defined in shader
angleID = glGetUniformLocation(program, "angle");

// my_angle set in application
GLfloat my_angle = 5.0 // or some other value

glUniform1f(angleID, my_angle);
Adding Color

• If we set a color in the application, we can send it to the shaders as a vertex attribute or as a uniform variable depending on how often it changes.

• Let’s associate a color with each vertex.

• Set up an array of same size as positions.

• Send to GPU as a vertex buffer object.
vec3  base_colors[4] = {vec3(1.0, 0.0, 0.0), ....
vec3  colors[NumVertices];
vec3  points[NumVertices];

// in loop setting positions

colors[i] = base_colors[color_index]
points[i] = ......
// this will identify our buffer
GLuint vertexbuffer

// generate 1 buffer, put the resulting identifier in vertexbuffer
glGenBuffers(1, &vertexbuffer);

// the following commands will talk about our “vertexbuffer” buffer
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);

// give our vertices to OpenGL; pass NULL to load data later
glBufferData(GL_ARRAY_BUFFER, sizeof(points) + sizeof(colors),
NULL, GL_STATIC_DRAW);

// load data separately
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(points), points);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(points), sizeof(colors),
    colors);
Second Vertex Array

// vPosition and vColor identifiers in vertex shader

loc1 = glGetAttribLocation(program, “vPosition”);
glEnableVertexAttribArray(loc1);
glVertexAttribPointer(loc1, 3, GL_FLOAT, GL_FALSE, 0,
                     (void *) 0);

loc2 = glGetAttribLocation(program, “vColor”);
glEnableVertexAttribArray(loc2);
glVertexAttribPointer(loc2, 3, GL_FLOAT, GL_FALSE, 0,
                     (void *) sizeof(points));

// draw the triangles
glDrawArrays(GL_TRIANGLES, 0, NumVertices);
Vertex Shader Examples

• A vertex shader is initiated by each vertex output by `glDrawArrays()`
• A vertex shader must output a position in clip coordinates to the rasterizer
• Basic uses of vertex shaders
  - Transformations
  - Lighting
  - Moving vertex positions
in vec4 vPosition;
uniform float xs, zs, // frequencies
uniform float h;       // height scale
void main()
{
    vec4 t = vPosition;
    t.y = vPosition.y
        + h*sin(time + xs*vPosition.x)
        + h*sin(time + zs*vPosition.z);
    gl_Position = t;
}
Particle System

```glsl
in vec3 vPosition;
uniform mat4 ModelViewProjectionMatrix;
uniform vec3 init_vel;
uniform float g, m, t;

void main()
{
    vec3 object_pos;
    object_pos.x = vPosition.x + vel.x*t;
    object_pos.y = vPosition.y + vel.y*t + g/(2.0*m)*t*t;
    object_pos.z = vPosition.z + vel.z*t;
    gl_Position = ModelViewProjectionMatrix*vec4(object_pos,1);
}
```
// pass-through fragment shader
in vec4 color;
void main(void)
{
    gl_FragColor = color;
}
Fragment Shader Applications (1)

Per fragment lighting calculations

per vertex lighting

per fragment lighting
Fragment Shader Applications (2)

Texture mapping