Interactive Graphics
CSCI B481 – Spring 2017

Lecture 03 part A – 2017-01-17

Instructor:
Mitja Hmeljak,
http://pages.iu.edu/~mitja
mitja@indiana.edu
OpenGL ES Shaders and GLSL programming

Simple Shaders
  Vertex shaders
  Fragment shaders

Programming shaders with GLSL
**Vertex Shader Applications**

Vertex shader programs can have different applications – some of these may be used separately, or together:

- **Moving** vertices
  can also be used for more unusual effects:
  Morphing, Wave motion, Fractals
- **Lighting**
- More realistic models
- Non-realistic shaders, such as "Cartoon shaders"
"per fragment" lighting calculations can be used to improve realism:

per vertex lighting  per fragment lighting
"per fragment" calculations can be used for texture mapping, environment mapping, bump mapping...:

- smooth shading
- environment mapping
- bump mapping
Writing Shaders, *then* and *now*

• First programmable shaders were programmed in an assembly-like manner
• the OpenGL API used "extensions" with added functions for vertex and fragment shaders
• followed by higher-level languages such as Cg (C for graphics), a C-like language for programming shaders
  • Cg works with both OpenGL and DirectX
  • the interface between Cg and OpenGL can be complex
• followed by the OpenGL Shading Language (GLSL)
GLSL is the 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 and OpenGL ES 2.0, an application must provide shaders
a simple Vertex Shader

"attribute" input from the application (CPU) program

attribute vec4 vPosition;
void main(void)
{
    gl_Position = vPosition;
}

the vPosition GLSL (GPU) variable must be linked to a variable in the application (CPU) program

"gl_Position" is a built-in variable, predefined for GLSL vertex shaders
Execution Model: Vertex Shader

- Application Program
- Vertex Shader
- Primitive Assembly
- GPU
- glDrawArrays

Vertex data and Shader Program
Simple Fragment Program

precision mediump float;

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

the "mediump" precision qualifier specifies the range:

<table>
<thead>
<tr>
<th></th>
<th>FP Range</th>
<th>FP Magnitude Range</th>
<th>FP Precision</th>
<th>Integer Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>mediump</td>
<td>(-2^{14}, 2^{14})</td>
<td>(2^{-14}, 2^{14})</td>
<td>Relative 2^{-10}</td>
<td>(-2^{10}, 2^{10})</td>
</tr>
</tbody>
</table>

"gl_FragColor" is a built-in variable, predefined for GLSL fragment shaders
what is Shadertoy's "Image Shader"?

Shadertoy provides **partial access** to the fragment shader, by allowing users to access one function:

```glsl
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
  vec2 uv = fragCoord.xy / iResolution.xy;
  fragColor = vec4(uv, 0.5+0.5*sin(iGlobalTime), 1.0);
}
```
Execution Model: Fragment Shader

- Application
- Shader Program
- Rasterizer
- Fragment Shader
- Fragment
- Frame Buffer
- Fragment Color
Data Types

C style types: int, float, bool

Vectors:
  float vec2, vec3, vec4
  Also int (and ivec) and boolean (and 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)
No Pointers

• There are no pointers in GLSL

• Because matrices and vectors are basic types they can be passed into (and output from) GLSL functions, e.g.

  \[ \text{mat3 func(mat3 a)} \]

• variables are passed by copying
Qualifiers

GLSL has many of the same qualifiers such as \texttt{const} as C/C++
Need other qualifiers, due to the nature of the execution model

variables passed into GLSL shaders can change:
- Once per primitive: vars defined using the \texttt{uniform} qualifier
- Once per vertex: variables defined using the \texttt{attribute} qualifier
- Once per fragment: variables defined using the \texttt{varying} qualifier
- At any time in the application

Vertex attributes are \textit{interpolated} by the rasterizer into fragment attributes
The **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)

- `attribute float temperature`
- `attribute vec3 velocity`
The **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 *time* or a *bounding box* of a primitive or *transformation matrices*
The **Varying** Qualified

Variables that are passed from vertex shader to fragment shader

Automatically **interpolated** by the rasterizer ← ←

With OpenGL ES, GLSL uses the varying qualifier in both shaders

```glsl
varying vec4 color;
```

(note: GLSL in OpenGL ES 3.x uses the **out** qualifier in vertex shaders and the **in** qualifier in the fragment shaders, e.g.:

```glsl
out vec4 myColor;  // in vertex shader
in vec4 myColor;   // in fragment shader
```

because of the intermediate shaders available in OpenGL ES 3.x)
References


• Angel/Shreiner "Interactive Computer Graphics" textbook, chapter 1

• Munshi/Ginsburg/Shreiner "OpenGL ES 2.0 Programming Guide" textbook, chapter 1