Today’s Agenda

Shaders fundamentals

Programming with shader-based OpenGL
Shaders

“Like a function call – data are passed in, processed, and passed back out”

-- Shreiner et al, OpenGL Programming Guide

GLSL is like a complete C program, but without

- recursion
- Pointer
- Dynamic allocation of memory
Vertex Shader

Basic task: Sending vertices positions to the rasterizer

Advanced tasks:
- Transformation
  - Projection
- Moving vertices
  - Morphing
  - Wave motion
  - Fractals
- Processing color
A Simple Vertex Shader: triangles.vert
(Shreiner et al)

#version 430 core

in vec4 vPosition;

void main()
{
    gl_Position = vPosition;
}

Global variable, copied from the application to the shader

A built-in variable, passing data to the rasterizer
A Simple Fragment Shader

```glsl
#version 330 core

out vec4 fColor;

void main()
{
    fColor = vec4( 1.0, 0.0, 0.0, 1.0 );
}
```
Example of Vertex Shader: Color Processing

// vertex shader
#version 150
in vec4 vPosition;
out vec4 color;
void main()
{
    color = vec4( 0.5 + vPosition.x, 0.5 + vPosition.y, 0.5 + vPosition.z, 1.0 );
    gl_Position = vPosition;
}

// fragment shader
#version 150
in  vec4  color;
out vec4  fColor;
void main()
{
    fColor = color;
}
Declaring Variables

Allowed: Letters, numbers, “_”

Not allowed:
• Digits and “_” cannot appear as the first character
• Do not allow consecutive “_”
Data Types

Basic types: int, float, double, uint, bool

Fewer implicit conversion

\texttt{int f = false}

### Table 2.2 Implicit Conversions in GLSL

<table>
<thead>
<tr>
<th>Type Needed</th>
<th>Can Be Implicitly Converted From</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint</td>
<td>int</td>
</tr>
<tr>
<td>float</td>
<td>int, uint</td>
</tr>
<tr>
<td>double</td>
<td>int, uint, float</td>
</tr>
</tbody>
</table>

Data Types

Vectors:
- float vec2, vec3, vec4
- Also int (ivec) and boolean (bvec)

Matrices: mat2, mat3, mat4, mat3x4
- Stored by columns

mat2 M = \begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{pmatrix}
vec2 col1 = \begin{pmatrix} 1.0 \\ 2.0 \end{pmatrix}
vec2 col2 = \begin{pmatrix} 3.0 \\ 4.0 \end{pmatrix}
mat2 M = \begin{pmatrix} \text{col1} \\ \text{col2} \end{pmatrix}

Structures: grouping different types
Data Types

C++ style constructors

• Truncate a vector:
  – vec4 color = vec4(1.0, 2.0, 3.0, 1.0); vec3 rgb = vec3(color)

• Lengthen a vector:
  – vec3 white = vec3(1.0) → white = (1.0, 1.0, 1.0)
  – vec4 translucent = vec4 (white, 0.5)

• Initialize a matrix

\[
M = \begin{bmatrix}
4.0 & 0.0 & 0.0 \\
0.0 & 4.0 & 0.0 \\
0.0 & 0.0 & 4.0 \\
\end{bmatrix}
\]

Samplers: used to represent texture
Data Referencing

Standard referencing:
- float green = color[1];
- float m12 = mat[row=1][column=2]

Component access
- float green = color.g;
- float velocity_x = velocity.x

<table>
<thead>
<tr>
<th>Component Accessors</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x, y, z, w)</td>
<td>components associated with positions</td>
</tr>
<tr>
<td>(r, g, b, a)</td>
<td>components associated with colors</td>
</tr>
<tr>
<td>(s, t, p, q)</td>
<td>components associated with texture coordinates</td>
</tr>
</tbody>
</table>

Swizzling

Can refer to array elements by element using [] or selection (.)
operator

a[2], a.b, a.z, a.p are the same

vec3 green=color.ggg

vec4 revcolor=color.abgr
vec3 a; a.yz=color.gr

vec3 vector1=color.rrg
vec3 vector2=color.zrg
# Operators

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operators</th>
<th>Accepted types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>()</td>
<td>—</td>
<td>Grouping of operations</td>
</tr>
<tr>
<td>2</td>
<td>[ ]</td>
<td>arrays, matrices, vectors</td>
<td>Array subscripting</td>
</tr>
<tr>
<td></td>
<td>f()</td>
<td>functions</td>
<td>Function calls and constructors</td>
</tr>
<tr>
<td></td>
<td>. (period)</td>
<td>structures</td>
<td>Structure field or method access</td>
</tr>
<tr>
<td>3</td>
<td>++ --</td>
<td>arithmetic</td>
<td>Pre-increment and -decrement</td>
</tr>
<tr>
<td></td>
<td>+ -</td>
<td>arithmetic</td>
<td>Unary explicit positive or negation</td>
</tr>
<tr>
<td></td>
<td>~</td>
<td>integer</td>
<td>Unary bit-wise not</td>
</tr>
<tr>
<td></td>
<td>!</td>
<td>bool</td>
<td>Unary logical not</td>
</tr>
<tr>
<td>4</td>
<td>* / %</td>
<td>arithmetic</td>
<td>Multiplicative operations</td>
</tr>
<tr>
<td>5</td>
<td>+ -</td>
<td>arithmetic</td>
<td>Additive operations</td>
</tr>
<tr>
<td>6</td>
<td>&lt;&lt; &gt;&gt;</td>
<td>integer</td>
<td>Bit-wise operations</td>
</tr>
<tr>
<td>7</td>
<td>&lt; &gt; &lt;= &gt;=</td>
<td>arithmetic</td>
<td>Relational operations</td>
</tr>
<tr>
<td>8</td>
<td>== !=</td>
<td>any</td>
<td>Equality operations</td>
</tr>
<tr>
<td>9</td>
<td>&amp;</td>
<td>integer</td>
<td>Bit-wise and</td>
</tr>
<tr>
<td>10</td>
<td>^</td>
<td>integer</td>
<td>Bit-wise exclusive or</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>12</td>
<td>&amp;&amp;</td>
<td>bool</td>
<td>Logical and operation</td>
</tr>
<tr>
<td>13</td>
<td>^^</td>
<td>bool</td>
<td>Logical exclusive-or operation</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>a ? b : c</td>
<td>bool ? any : any</td>
<td>Ternary selection operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(inline “if” operation; if (a) then (b) else (c))</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>=</td>
<td>any</td>
<td>Assignment</td>
</tr>
<tr>
<td></td>
<td>+= -=</td>
<td>arithmetic</td>
<td>Arithmetic assignment</td>
</tr>
<tr>
<td></td>
<td>*= /=</td>
<td>arithmetic</td>
<td></td>
</tr>
<tr>
<td></td>
<td>%= &lt;&lt;= &gt;&gt;=</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;= ^=</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>, (comma)</td>
<td>any</td>
<td>Sequence of operations</td>
</tr>
</tbody>
</table>
Operators and Built-in Functions

Built-in functions
- Arithmetic, e.g., pow(), exp(), sqrt(), etc.
- Trigonometric, e.g., sin(), cos(), etc.
- Matrix functions, e.g., transpose(), inverse(), etc.
- Many more: normalize(), reflect(), length(), distance(), etc.

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
Example of Vertex Shader: Geometric Transformation

```glsl
#version 330 core

in vec4 vPosition;

in vec4 vColor;

out vec4 color;

uniform mat4 ModelViewProjectionMatrix;

void main()
{
    color = vColor;

    gl_Position = ModelViewProjectionMatrix * vPosition;
}
```

Type Qualifier

Define and modify the behavior of variables

- **Storage qualifiers: where the data come from**
  - `const`: read-only, must be initialized when declared
  - `in`: vertex attributes or from the previous stage
  - `out`: output from the shader
  - `uniform`: a global variable shared between all the shader stages
  - `buffer`: share buffer with application (r/w)

- **Layout qualifiers: the storage location**

- **Invariant/precise qualifiers: enforcing the reproducibility**
Uniform Qualifiers

A global variable

- Used to pass information to shader such as the bounding box and the transformation matrix of a primitive
- shared between all the shader stages
- Can be changed in the application and sent to shaders
- Cannot be changed in shader
How to set the value for uniform qualifiers?

GLSL compiler creates a table for all uniform variable when linking the program.

**Step1:** You need to get the index of the variable by

\[
glGetUniformLocation()\]

The index is not changed unless relinking the program

**Step2:** set the value using

\[
glUniform*() \text{ or } glUniformMatrix*()\]
An Example of Uniform Qualifiers (Shreiner et al)

In the shader

uniform GLint time;

In the application

GLint timeLoc; /* Uniform index for variable "time" in shader */
GLfloat timeValue; /* Application time */

timeLoc = glGetUniformLocation(program, "time");
glUniform1f(timeLoc, timeValue);
Example of Vertex Shader: Wave Motion

Vertex Shader

```glsl
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;
}
```

Example of Vertex Shader: 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 + init_vel.x*t;
    object_pos.y = vPosition.y + init_vel.y*t
                   + g/(2.0*m)*t*t;
    object_pos.z = vPosition.z + init_vel.z*t;
    gl_Position =
        ModelViewProjectionMatrix*vec4(object_pos,1);
}
```
Double Buffering

Updating the value of a uniform variable opens the door to animating an application
  • Execute glUniform in display callback
  • Force a redraw through glutPostRedisplay()

Need to prevent a partially redrawn frame buffer from being displayed → Double buffering

Draw into back buffer

Display front buffer

Swap buffers after updating finished
Adding Double Buffering

Request a double buffer
  • glutInitDisplayMode(GLUT_DOUBLE)

Swap buffers

void mydisplay()
{
  glClear(......);
  glDrawArrays();
  glutSwapBuffers();
}
Compiling Shaders

For each shader object,

Step1: create a shader object

```c
GLuint glCreateShader(GLenum type);
```

Type: GL_VERTEX_SHADER and
GL_FRAGMENT_SHADER

Step2: read the shader source

Step3: associate the shader source

with the shader object

```c
void glShaderSource(GLuint shader, GLsizei count,
                   const GLchar **string, const GLint *length);
```
Compiling Shaders

Step 4: compile a shader object

```c
void glCompileShader(GLuint shader);
```

Step 5: verify the shader compiled successfully

```c
GLint compiled;
glGetShaderiv( shader, GL_COMPILE_STATUS, &compiled );
```
Linking Shaders

Step 1: create a shader program
• Can contain multiple shaders

```c
GLuint glCreateProgram(void);
```

Step 2: attach the shader objects to program

```c
void glAttachShader(GLuint program, GLuint shader);
```

Step 3: link the shader program

```c
void glLinkProgram(GLuint program);
```

Step 4: verify the link is successful

```c
GLint linked;

glGetProgramiv( program, GL_LINK_STATUS, &linked );
```

Step 5: use the shader program

init()

ShaderInfo shaders[] = {
    { GL_VERTEX_SHADER, " triangles. vert" },
    { GL_FRAGMENT_SHADER, "triangles.frag" },
    { GL_NONE, NULL }
};

GLuint program = LoadShaders(shaders); // Load, compile and link shaders
glUseProgram(program);

glVertexAttribPointer(vPosition, 2, GL_FLOAT, 
GL_FALSE, 0, BUFFER_OFFSET(0));
glEnableVertexAttribArray(vPosition);

---

Initialize the vertex and fragment shaders
Location of shader attributes
Connect shader to a vertex-attribute array
An Example of Adding a Vertex Shader

```cpp
GLuint myProgObj;
myProgObj = glCreateProgram();

GLuint vShader;
GLunit myVertexObj;
GLchar vShaderfile[] = "my_vertex_shader";
GLchar* vSource = readShaderSource(vShaderFile);
glShaderSource(myVertexObj, 1, &vertexShaderFile, NULL);
myVertexObj = glCreateShader(GL_VERTEX_SHADER);
glCompileShader(myVertexObj);
glAttachObject(myProgObj, myVertexObj);

glLinkProgram(myProgObj);
glUseProgram(myProgObj);
```