SHADER PROGRAMMING

Based on Jian Huang’s lecture on Shader Programming
What OpenGL 15 years ago could do…

http://www.neilturner.me.uk/shots/opengl-big.jpg
What OpenGL can do now…
What’s Changed?

• 15 years ago:
  • Transform vertices with modelview/projection matrices.
  • Shade fragments with Phong lighting model only.

• Now:
  • Custom vertex transformation.
  • Custom lighting model.
  • More complicated visual effects.
    • Shadows
    • Displaced and detailed surfaces.
    • Simple reflections and refractions,
    • Etc.
  • More on this in 5625 next semester!
What’s Changed?

• 15 years ago:
  • Vertex transformation/fragment shading hardcoded into GPUs.

• Now:
  • More parts of the GPU are programmable (but not all).
GPU Pipeline in 2000

3D API: OpenGL or Direct3D

3D Application Or Game

CPU-GPU Boundary (AGP/PCIe)

GPU Command & Data Stream

Vertex Index Stream

Assembled Primitives

Raster Location Stream

Pixel Updates

GPU Front End

Primitive Assembly

Rasterization and Interpolation

Raster Operations

Frame Buffer
GPU Pipeline in 2004

3D API: OpenGL or Direct3D

3D Application Or Game

CPU-GPU Boundary (AGP/PCIe)

GPU Front End

Primitive Assembly

Rasterization and Interpolation

Raster Operations

Frame Buffer

3D API Commands

Vertex Index Stream

Assembled Primitives

Pixel Location Stream

Pixel Updates

CPU Command & Data Stream

GPU Command & Data Stream

Pre-transformed Vertices

Pre-transformed Fragments

Transformed Vertices

Transformed Fragments

Programmable Vertex Processor

Programmable Fragment Processor

3D Application Or Game

Transformed Vertices
GPU Pipeline Nowadays

http://www.realtimerendering.com/blog/direct3d-11-details-part-ii-tessellation/
Shader

• A small program to control a part of the graphics pipeline
  • Vertex shader controls vertex transformation.
  • Fragment shader controls fragment shading.
**Vertex Shader**

- Transform vertices from object space to clip space.
  - Doesn’t have to be modelview followed by projection.
- Compute other data that are interpolated with vertices.
  - Color
  - Normals
  - Texture coordinates
  - Etc.
What can we do with a vertex shader?

- Displaced/distorted surfaces
Fragment Shader

• Compute the color of a fragment.
• Take interpolated data from vertex shaders.
• Can read more data from:
  • Textures
  • User specified values.
What can we do with a vertex shader?

- Skinning
What can we do with a fragment shader?

• More complicated materials:
  • Glossy
  • Reflective, refractive
  • Rough, bumpy, lots of nooks and crannies
  • Wooden
What can we do with a fragment shader?

• Shadow

[Image: GLSL Shadow mapping]
GLSL
GLSL

• Similar to C/C++

• Used to write shaders
  • Vertex, fragment, geometry
  • We only cover vertex and fragment here.

• Based on OpenGL
  • First available in OpenGL 2.0 (2004)

• Competitors:
  • Nvidia Cg
  • Microsoft HLSL
GLSL Program

- A collection of shaders that run together.
  - At least one vertex shader or one fragment shader.
  - Should have both so we know its behavior completely.

- At any time, the GPU runs only one program.
  - Initially, OpenGL uses its own default program.
  - We can tell OpenGL to use our program.
  - We can also tell it to go back and use its initial program.
Shader

- Shader source code resembles C/C++ source code.
  - Similar data types, expressions, and control statements.
  - Functions are written in the same way.

- Entry point = “void main( )”
  - Not “int main(int argc, char **argv)” as in normal C.

- A shader’s source code can reside in many files.
  - However, only one of them must have the main function.

- Two main functions when writing a vertex shader and a fragment shader together.
Shader Structure

- /*
  * Multiple-lined comment
  */

  // Single-lined comment

  //
  // Global variable definitions
  //

  void main()
  {
    //
    // Function body
    //
  }
GREEN SHADER
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
void main()
{
    gl_FragColor = vec4(0, 1, 0, 1);
}
Suppose we have already created the program
We tell OpenGL to use it.
We then instruct OpenGL to draw a triangle:

```c
glMatrixMode(GL_PROJECTION);
klLoadIdentity();
gluOrtho2D(-1, 1, -1, 1);

glMatrixMode(GL_MODELVIEW);
klLoadIdentity();

glBegin(GL_TRIANGLES);
gIVertex3f(-0.5, -0.5, 0);
gIVertex3f( 0.5, -0.5, 0);
gIVertex3f( 0, 0.5, 0);
gIEnd();
```
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(-1, 1, -1, 1);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

glBegin(GL_TRIANGLES);
    glVertex3f(-0.5, -0.5, 0);
    glVertex3f( 0.5, -0.5, 0);
    glVertex3f( 0, 0.5, 0);
glEnd();
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(-1, 1, -1, 1);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

glBegin(GL_TRIANGLES);
glVertex3f(-0.5, -0.5, 0);
glVertex3f( 0.5, -0.5, 0);
glVertex3f( 0, 0.5, 0);
glEnd();

GLSL

gl_ModelViewProjectionMatrix is a special variable that is equal to the product of projection matrix and modelview matrix.
OpenGL/GLSL Plumbling

```c
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

GLSL

A vertex shader is executed each time a vertex is specified.

So, since there are three calls to glVertex3f, the above vertex shader is executed 3 times.

1\textsuperscript{st} time: gl\_Vertex = (-0.5, -0.5, 0, 1)
2\textsuperscript{nd} time: gl\_Vertex = (0.5, -0.5, 0, 1)
3\textsuperscript{rd} time: gl\_Vertex = (0, 0.5, 0, 1)
OpenGL/GLSL Plumbling

```c
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

GLSL

```c
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(-1, 1, -1, 1);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

glBegin(GL_TRIANGLES);
    glVertex3f(-0.5, -0.5, 0);
    glVertex3f( 0.5, -0.5, 0);
    glVertex3f( 0, 0.5, 0);
glEnd();
```

OpenGL

`gl_Vertex` is another special variable that holds the value we specify with `glVertex`
**OpenGL/GLSL Plumbling**

```glsl
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

**GLSL**

`gl_Position` is a special variable that holds the position of the vertex in clip space.

Since a vertex shader’s main output is the position in clip space. It must always set `gl_Position`.

So, the vertex shader above just does what OpenGL normally does.
void main()
{
    gl_FragColor = vec4(0,1,0,1);
}
void main()
{
  gl_FragColor = vec4(0,1,0,1);
}

GLSL

`gl_FragColor` is a special variable that stores the color of the output fragment.

Since a fragment shader computes the color of a fragment. It must always set `gl_FragColor`. 
vec4 is a data type of 4D vector.

Can be used to store
• homogeneous coordinate
• RGBA

vec4(0,1,0,1) constructs an RGBA tuple with R=0, G=1, B=0, A=1, which is green.
Demo 17
USING GLSL IN JAVA
To use a GLSL program…

- Follow the following 7 steps.

1. Create shader objects.
2. Read source code from files and feed them to the shader objects just created.
3. Compile the shader.
4. Create a program object.
5. Attach the shaders to the program.
6. Link the program.
7. Tell OpenGL to use your shaders instead of the default ones.
To use a GLSL program…

- Follow the following 7 steps.

1. Create shader objects.
2. Read source code from files and feed them to the shader objects just created.
3. Compile the shaders.
4. Create a program object.
5. Attach the shaders to the program.
6. Link the program.
7. Tell OpenGL to use your shaders instead of the default ones.
CS 4620 Framework

- A collection of classes that take away some pain from OpenGL programming.

- Features:
  - GUI
  - Camera control
  - Picking
  - Creating and using textures
  - Creating and using GLSL programs
  - More to come…

- A version is distributed with this lecture’s sample code.
Now, to use a GLSL program...

- Create a Program object.

```java
private Program greenShaderProgram = null;

public void init(GLAutoDrawable drawable) {
    final GL2 gl = drawable.getGL().getGL2();

    try {
        // Load, compile and link the shaders
        this.greenShaderProgram = new Program(
            gl,
            "src/shaders/green_shader.vert",
            "src/shaders/green_shader.frag");
    }
    catch (GlslException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
}
```
Now, to use a GLSL program…

- Use it. Draw stuff. Unuse it (if need be).
- Always change GLSL program outside glBegin(...) glEnd() block.

```cpp
greenShaderProgram.use();

gl.glBegin(GL2.GL_TRIANGLES);
gl.glVertex3f(-0.5f, -0.5f, 0.0f);
gl.glVertex3f( 0.5f, -0.5f, 0.0f);
gl.glVertex3f( 0.0f,  0.5f, 0.0f);
gl.glEnd();

greenShaderProgram.unuse();
```
GLSL DATA TYPES
GLSL Data Types

- Both GLSL and Java
  - float, int

- GLSL has, but Java has not
  - vec3, vec4, vec4: vectors
  - mat2, mat3, mat4: matrices
  - sampler1D, sampler2D, sample3D, samplerCube, etc: textures

- Java has, but GLSL has not
  - Object
  - String
  - etc
vec2

- Represents a vector in 2D. Each component is a float.

```cpp
vec2 a;
a.x = 0.0;
a.y = 1.0; // a = (0,1)

vec2 b;
b.s = 10.0;
b.t = 12.5; // b = (10,12.5)

vec2 c;
c[0] = 9.0;
c[1] = 8.0; // c = (9,8)
```
vec2

float p = a.t; // p = 1
float q = b[1] + c.x // q = 21.5

vec2 d = vec2(3,c.y * 2); // d = (3,16)

vec3 d = a + b; // d = (10,13.5)

vec3 e = b - c; // e = (1,4.5)

vec3 f = b * c; // f = (90,100)

vec3 g = 3 * a; // g = (0,3)

float h = length(c); // h = 12.042
vec3

vec3 a;
a.x = 10.0; a.y = 20.0; a.z = 30.0; // a = (10, 20, 30)
a.r = 0.1; a.g = 0.2; a.b = 0.3; // a = (0.1, 0.2, 0.3)
a.s = 1.0, a.t = 2.0; a.p = 3.0; // a = (1, 2, 3)

vec3 b = vec3(4.0, 5.0, 6.0);

vec3 c = a + b; // c = (5, 7, 9)
vec3 d = a - b; // d = (-3, -3, -3)
vec3 e = a * b; // e = (4, 10, 18)
vec3 f = a * 3; // e = (3, 6, 9)
float g = dot(a,b); // g = 32
vec3 h = cross(a,b); // h = (-5,6,-3)
float i = length(a); // i = 3.742
vec4

vec4 a;
a.x = 10.0; a.y = 20.0; a.z = 30.0; a.w = 40.0; // a = (10, 20, 30, 40)
a.r = 0.1; a.g = 0.2; a.b = 0.3; a.a = 0.4; // a = (0.1, 0.2, 0.3, 0.4)
a.s = 1.0; a.t = 2.0; a.p = 3.0; a.q = 4.0; // a = (1, 2, 3, 4)

vec4 b = vec4(5, 6, 7, 8);

vec4 c = a + b; // c = (6, 8, 10, 12)
vec4 d = a - b; // d = (-4, -4, -4, -4)
vec4 e = a * b; // e = (5, 12, 21, 32)
vec4 f = a * 3; // f = (3, 6, 9, 12)
float g = length(a); // g = 5.477
• Represents a 2 by 2 matrix. Each component is a float.

```cpp
mat2 A = mat2(1.0, 2.0, 3.0, 4.0); // in column-major order

vec2 x = vec2(1.0, 0.0);
vec2 y = vec2(0.0, 1.0);

vec2 a = A * x; // a = (1,2)
vec2 b = A * y; // b = (3,4)
```
mat3

mat3 A = mat3(1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0);
    // in column-major order

vec3 x = vec3(1.0, 0.0, 0.0);
vec3 y = vec3(0.0, 1.0, 0.0);
vec3 z = vec3(0.0, 0.0, 1.0);

vec3 a = A * x; // a = (1,2,3)
vec3 b = A * y; // b = (4,5,6)
vec3 c = A * z; // c = (6,7,8)
mat4

- 4x4 matrices. Can store affine transformations.

```cpp
mat4 A = mat4(1.0, 2.0, 3.0, 4.0,
               5.0, 6.0, 7.0, 8.0,
               9.0, 10.0, 11.0, 12.0,
               13.0, 14.0, 15.0, 16.0); // in column-major order

vec4 x = vec4(1.0, 0.0, 0.0, 0.0);
vec4 y = vec4(0.0, 1.0, 0.0, 0.0);
vec4 z = vec4(0.0, 0.0, 1.0, 0.0);
vec4 w = vec4(0.0, 0.0, 0.0, 1.0);

vec4 a = A * x; // a = (1,2,3,4)
vec4 b = A * y; // b = (5,6,7,8)
vec4 c = A * z; // c = (9,10,11,12)
vec4 d = A * w; // d = (13,14,15,16)
```
Array

- We can declare fixed-size arrays.
- Use C syntax.

```c
float A[4];

vec4 B[10];
B[3] = vec4(1,2,3,4); B[8].y = 10.0;
```
Twizzling

- Used to construct a vector from another vector by referring to multiple components at one time.

```cpp
vec4 a = vec4(1, 2, 3, 4);
vec3 b = a.xyz; // b = (1,2,3)
vec2 c = a.qp; // c = (4,3)
vec4 d = a.xxyy; // d = (1,1,2,2)
```
Type Conversion

• Syntax: `<<variable>> = <<type>>( <<value>> );`

• Expression on RHS = “constructor expression.”

• Example:

```c
float a = 1.0;
int b = int(a);
```
Type Conversion

• We can create larger vectors from smaller ones.

    vec2 a = vec2(1,2);
    vec2 b = vec2(3,4);

    vec4 c = vec4(a,b);  // c = (1,2,3,4)

    vec3 d = vec3(0,0,1);

    vec4 e = vec4(d,0);  // d = (0,0,1,0)

    vec4 f = vec2(0,a,3); // f = (0,1,2,3)