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.
What’s Changed?

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

• Now:
  • More parts of the GPU are programmable (but not all).
Shader Program

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

- Transform vertices from object space to clip space.
  - Conventionally modelview followed by projection
  - Can define custom transformation to clip space
- Compute other data that are interpolated with vertices.
  - Color
  - Normals
  - Texture coordinates
  - Etc.
What can we do with a vertex shader?

- Displaced/distorted surfaces
What can we do with a vertex shader?

- Skinning

Dolphin #1  Morphed Dolphin  Dolphin #2

Images courtesy of Microsoft
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 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

http://www.fabiensanglard.net/shadowmappingPCF/8x8kernel_nVidia.jpg
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

• Specifies how OpenGL should draw geometry.

• 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.
  • Must specify program to use before drawing geometry.
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.

- 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
Vertex Shader

uniform mat4 un_Projection;
uniform mat4 un_ModelView;

attribute vec3 in_Vertex;

void main()
{
    gl_Position = un_Projection * un_ModelView * vec4(in_Vertex, 1);
}
Fragment Shader

uniform mat4 un_Projection;
uniform mat4 un_ModelView;

void main()
{
    gl_FragColor = vec4(0,1,0,1);
}
OpenGL/GLSL Plumbering

• Suppose we have already created the program
• We tell OpenGL to use it.
• We then instruct OpenGL to draw a triangle:

```
// The vertices in our vertex buffer,
// initialized earlier
float [] vertices = {-0.5, -0.5, 0,
                    -0.5, -0.5, 0,
                           0,  0.5, 0};

// In the draw method
Program.use(triangleProgram);
triangleArray.draw();
Program.unuse(gl);
```
OpenGL/GLSL Plumbing

```cpp
uniform mat4 un_Projection;
uniform mat4 un_ModelView;

attribute vec3 in_Vertex;

void main()
{
    gl_Position = un_Projection * un_ModelView * vec4(in_Vertex, 1);
}

// The vertices in our vertex buffer,
// initialized earlier
float [] vertices = {-0.5, -0.5, 0,
    -0.5, -0.5, 0,
    0, 0.5, 0};

// In the draw method
Program.use(triangleProgram);

triangleArray.draw();

Program.unuse(gl);
```
A vertex shader is executed once for each vertex drawn. So this shader will be executed 3 times.
OpenGL/GLSL Plumbing

```glsl
uniform mat4 un_Projection;
uniform mat4 un_ModelView;

attribute vec3 in_Vertex;

void main()
{
    gl_Position = un_Projection * un_ModelView * vec4(in_Vertex, 1);
}
```

GLSL Vertex Shader

```c
// The vertices in our vertex buffer, // initialized earlier
float [] vertices = {-0.5, -0.5, 0,
                     -0.5, -0.5, 0,
                     0,  0.5,  0};

// In the draw method
Program.use(triangleProgram);
triangleArray.draw();
Program.unuse(gl);

OpenGL Code

Uniforms are one type of input to the shader. They are the same for each vertex drawn during one draw function. We will see how to set them later. These ones are the conventional projection and modelView matrices.
uniform mat4 un_Projection;
uniform mat4 un_ModelView;

attribute vec3 in_Vertex;

void main()
{
    gl_Position = un_Projection * un_ModelView * vec4(in_Vertex, 1);
}

// The vertices in our vertex buffer,
// initialized earlier
float [] vertices = {-0.5, -0.5, 0,
                     -0.5, -0.5, 0,
                     0,  0.5,  0};

// In the draw method
Program.use(triangleProgram);
triangleArray.draw();
Program.unuse(gl);

attribute variables link to vertex attributes. This one is set to the vertex position buffer. Each time the shader is executed, in_Vertex is set to the vertex currently being processed.
OpenGL/GLSL Plumbing

```glsl
uniform mat4 un_Projection;
uniform mat4 un_ModelView;

attribute vec3 in_Vertex;

void main()
{
    vec4 vertex = in_Vertex;
    gl_Position = un_Projection * un_ModelView * vertex;
}
```

**GLSL Vertex Shader**

`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 transforms each vertex position by the projection, model, and view matrices.
OpenGL/GLSL Plumbling

GLSL Fragment Shader

```
uniform mat4 un_Projection;
uniform mat4 un_ModelView;

void main()
{
    gl_FragColor = vec4(0, 1, 0, 1);
}
```

Fragment shaders are executed once per fragment (pixel).
OpenGL/GLSL Plumbing

`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`.

```glsl
uniform mat4 un_Projection;
uniform mat4 un_ModelView;

void main()
{
    gl_FragColor = vec4(0, 1, 0, 1);
}
```
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.
GreenTriangleDemo
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 shader program.
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 shader program.
CS 4620 Framework

- Contains Program class to abstract OpenGL calls.
- Usually use a subclass of the Program class
  - Add convenience methods
  - Help keep conventions straight
  - Controls mapping between attribute variables and vertex buffers
Now, to create a GLSL program…

• Create a Program object.

```
private SceneProgram greenShaderProgram = null;

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

    try {
        // Load, compile and link the shaders
        this.greenShaderProgram = new SceneProgram(
            gl,
            "greenShader.vs",
            "greenShader.fs");
    }
    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).

```javascript
Program.use(gl, greenShaderProgram);

// Draw Stuff
triangleArray.draw(gl);

Program.unuse(gl);
```
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
• Represents a vector in 2D. Each component is a float.

```
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)

vec2 d = a + b; // d = (10,13.5)

vec2 e = b - c; // e = (1,4.5)

vec2 f = b * c; // f = (90,100)

vec2 g = 3 * a; // g = (0,3)

float h = length(c); // h = 12.042
vec2

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

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)
vec3

cvec3 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 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
mat2

• Represents a 2 by 2 matrix. Each component is a float.

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 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.

mat4 A = mat2(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.

```cpp
float A[4];

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

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

```c
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:

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

• We can create larger vectors from smaller ones.

```cpp
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)
```
Uniform variables
Uniform Variable

- A GLSL variable the user can specify value from the C/Java side.
- Its value is constant while drawing a vertex array.
- Suitable for specifying
  - Material properties
  - Transformation matrices
  - Light sources
  - Texture
Declaring a Uniform Variable in GLSL

- Declare as a global variable (outside functions).
- Prefix the variable type with keyword “uniform.”
- Examples:

```
uniform float shininess;
uniform vec3 color;
uniform mat4 model_transform;
```

```c
void main()
{
    // Code here...
}
```
Caveats

• Uniform variables are shared between vertex and fragment shaders.
  • Declare once in vertex shader and once more in fragment shader.

• As a result, types of uniform variables in vertex and fragment shaders must be consistent.
  • Cannot have uniform int x; in vertex shader, but uniform float x; in fragment shader.

• Uniforms that are declared but not used are “optimized” out
  • OpenGL throws an error if you try to set a nonexistent uniform
  • Using convenience methods in subclasses of Program fix this (they check if uniforms exist before setting them)
Using Uniform Variables in the CS4620 Framework

- Uniform variables are encapsulated by `Uniform` class.

- Use `Program.getUniform(<name>)` to get the instance corresponding to the name.

- Set values by `Uniform.set**(...)` methods.

```java
// In GLSL: uniform vec3 p;
program.use(gl);
Uniform p = program.getUniform("p");
if (p != null) {
    p.set3Float(gl, 1.0f, 2.0f, 3.0f);
}

// In GLSL: uniform int count;
program.use(gl);
Uniform countUniform = program.getUniform("count");
if (countUniform != null) {
    countUniform.setInt1(gl, 10);
}
```
Demo: Twisting
2D Twisting

- We transform the vertices according to the following equation:

\[
\begin{bmatrix}
  x' \\
y'
\end{bmatrix} =
\begin{bmatrix}
  \cos(t\sqrt{x^2 + y^2}) & -\sin(t\sqrt{x^2 + y^2}) \\
  \sin(t\sqrt{x^2 + y^2}) & \cos(t\sqrt{x^2 + y^2})
\end{bmatrix} \begin{bmatrix}
  x \\
y
\end{bmatrix}
\]

where
- \((x,y)\) is the vertex position in object space.
- \((x',y')\) is the vertex position in clip space.
- \(t\) is the twisting factor,
  which is stored in the uniform variable “un_Twist”
2D Twisting
#version 120

uniform mat4 un_Projection;
uniform mat4 un_ModelView;

uniform float un_Twist;

attribute vec3 in_Vertex;

void main()
{
    float angle = un_Twist * length(in_Vertex.xy);
    float s = sin(angle);
    float c = cos(angle);
    gl_Position.x = c * in_Vertex.x - s * in_Vertex.y;
    gl_Position.y = s * in_Vertex.x + c * in_Vertex.y;
    gl_Position.z = 0.0;
    gl_Position.w = 1.0;
}
#version 120

uniform mat4 un_Projection;
uniform mat4 un_ModelView;
uniform vec3 un_DiffuseColor;
uniform float un_Twist;

void main()
{
  gl_FragColor = vec4(un_DiffuseColor, 1);
}
GLSL Program Preparation in Java

```java
class GLSLProgramPreparation {

    public void init(GLAutoDrawable drawable) {
        if (!initialized)
            return;

        GL2 gl = drawable.getGL().getGL2();

        // set vertex positions and Initialize quadArray (a vertex array variable)

        // create shader program for drawing
        try {
            program = new TwistingProgram(gl, "twisting.vs", "twisting.fs");
        } catch (GlslException e) {
            System.err.println("FAIL: creating program");
            e.printStackTrace();
            System.exit(1);
        }

        // initialize transforms and uniforms
        program.setProjection(gl, Transforms.ortho3DH(-1f, 1f, -1f, 1f, .5f, -.5f));
        program.setModelView(gl, Transforms.identity3DH());
        program.setDiffuseColor(gl, new Vector3f(0.8f, 0.2f, 0.2f));
        program.setTwist(gl, 5.0f);
    }
}
```
Using the GLSL Program

```java
public void display(GLAutoDrawable drawable) {
    final GL2 gl = drawable.getGL().getGL2();

    gl.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    gl.glClear(GL2.GL_COLOR_BUFFER_BIT);

    Program.use(gl, program);

    quadArray.draw(gl);

    Program.unuse(gl);
}
```
Attribute variables
Attribute Variables

• A variable containing an attribute for a single vertex.
• Position, normal, texture coordinate, etc.
• Each time the shader is run, the attribute variables receive the values for the current vertex.
Attribute Mapping

- Attribute variables map to OpenGL buffers.
- OpenGL buffers have an index, GLSL attribute variables have a name.
- Must ensure the mapping from buffer indices to variable names is correct (the Program class and subclasses handle this).
Varying variables
Varying Variables

- Interface between vertex and fragment shaders.
- Vertex shader outputs a value at each vertex.
- Fragment shader reads a value interpolated to that fragment.
Declaring Varying Variables

• Declare as a global variable (outside functions).

• Syntax: varying <<type>> <<name>>;

• Example:

```
varying vec3 color;

void main()
{
    // Some code here...
}
```
Demo: Position as Color
Position as Color

• Compute the color of each fragment from its position in object space.

• \(\text{color} = \frac{(\text{position} + (1,1,1))}{2}\)
#version 120

uniform mat4 un_Projection;
uniform mat4 un_ModelView;

attribute vec3 in_Vertex;

varying vec3 ex_Color;

void main()
{
    gl_Position = un_Projection * un_ModelView * vec4(in_Vertex, 1);
    ex_Color = (in_Vertex.xyz + vec3(1,1,1)) * 0.5;
}
#version 120

uniform mat4 un_Projection;
uniform mat4 un_ModelView;

varying vec3 ex_Color;

void main()
{
    gl_FragColor = vec4(ex_Color, 1);
}