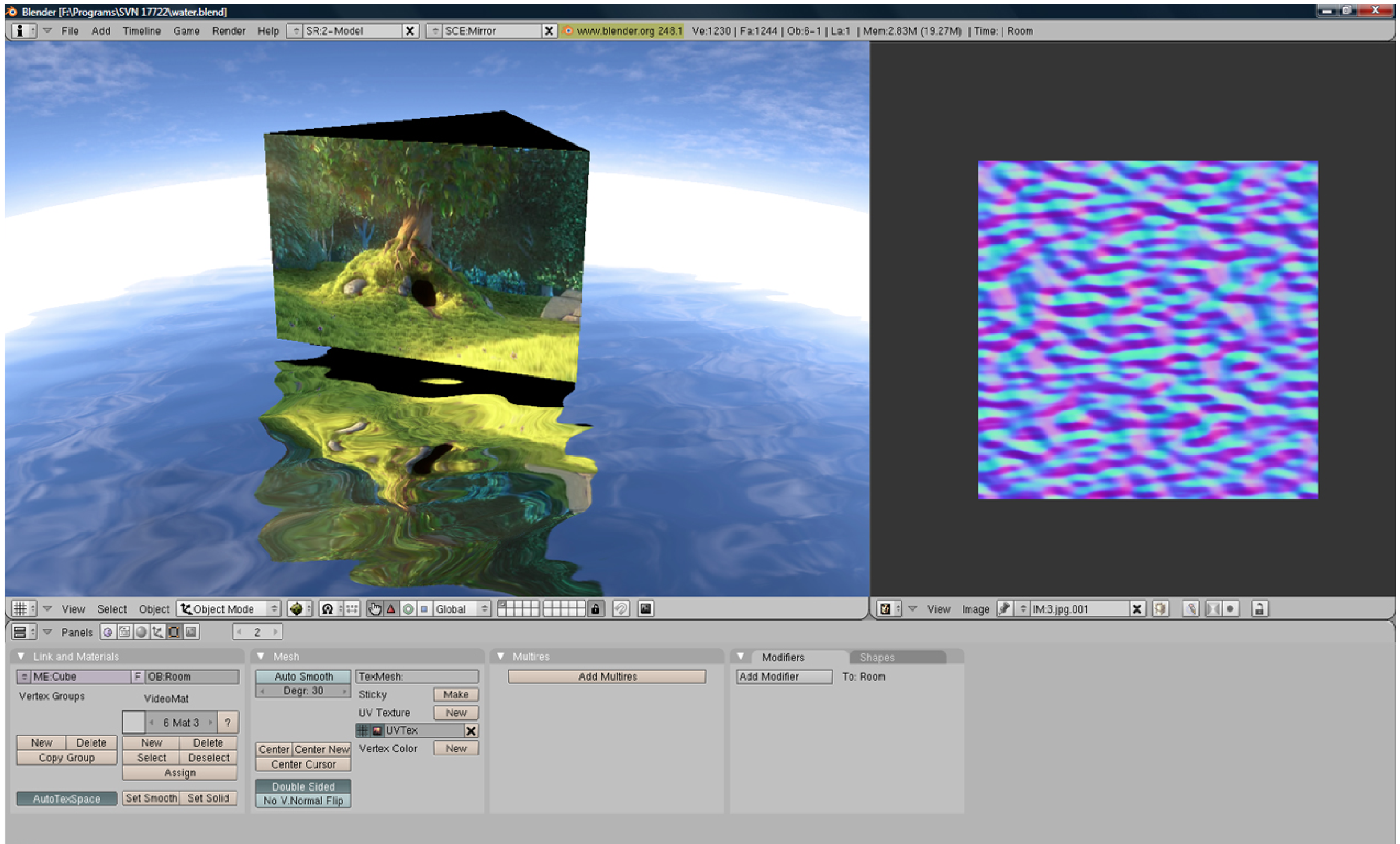
Shader programming

Based on Jian Huang's lecture on Shader Programming

What OpenGL could do 20 years ago

What OpenGL could do 5 years ago



What's Changed?

- 20 years ago:
  - Transform vertices with modelview/projection matrices.
  - Shade 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?

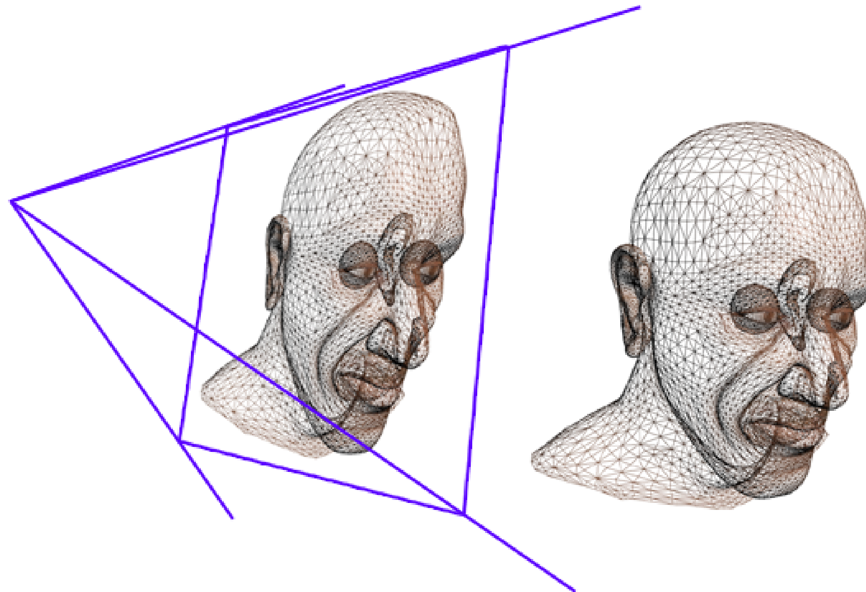
- 20 years ago:
  - Vertex transformation/fragment shading hardcoded into GPUs.
- Now:
  - More parts of the GPU are programmable (but not all).

Shader Program

- A small program to control parts of the graphics pipeline
- Consists of 2 (or more) separate parts:
  - Vertex shader controls vertex transformation.
  - Fragment shader controls fragment shading.

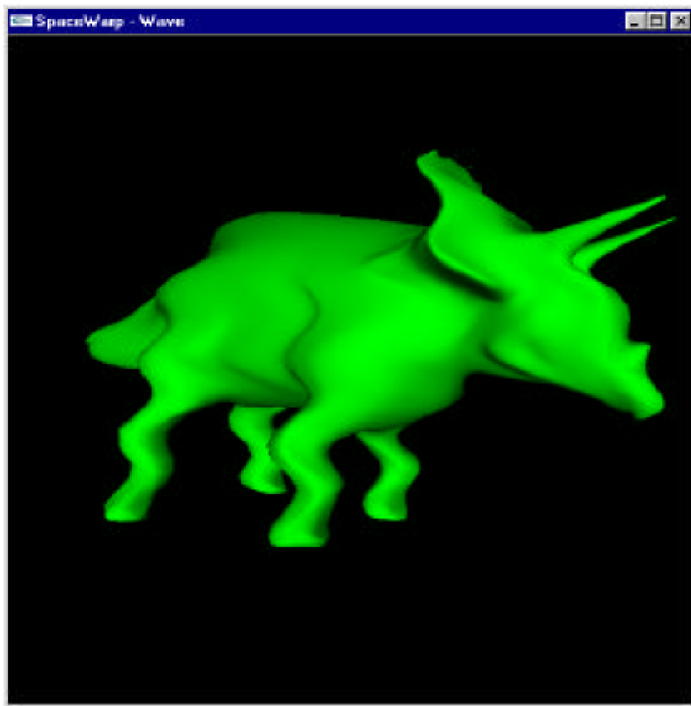
Vertex Shader

- 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.



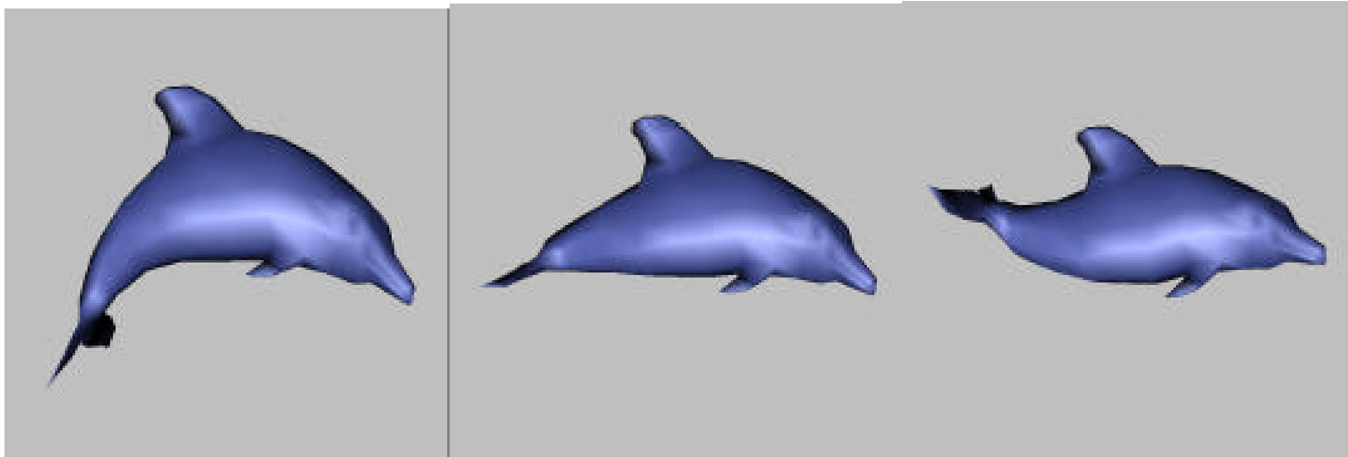
What Can a Vertex Shader Do?

Displaced/distorted surfaces



What Can a Vertex Shader Do?

Skinning/Animation

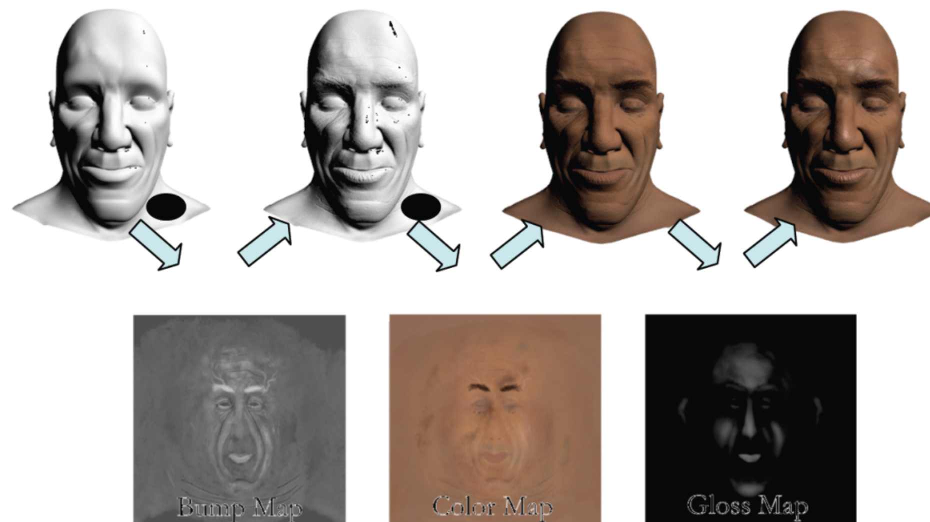


Dolphin #1 Morphed Dolphin Dolphin #2

Images courtesy of Microsoft

Fragment Shader

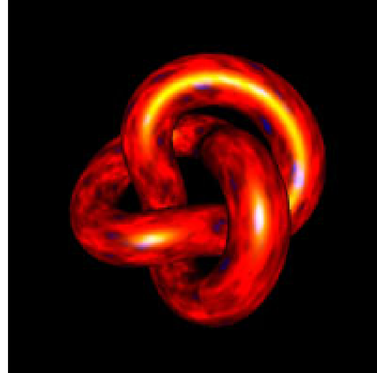
- Compute the color of a fragment (i.e. a pixel).
- Take interpolated data from vertex shaders.
- Can read more data from:
  - Textures
  - User specified values.



What Can a Fragment Shader Do?

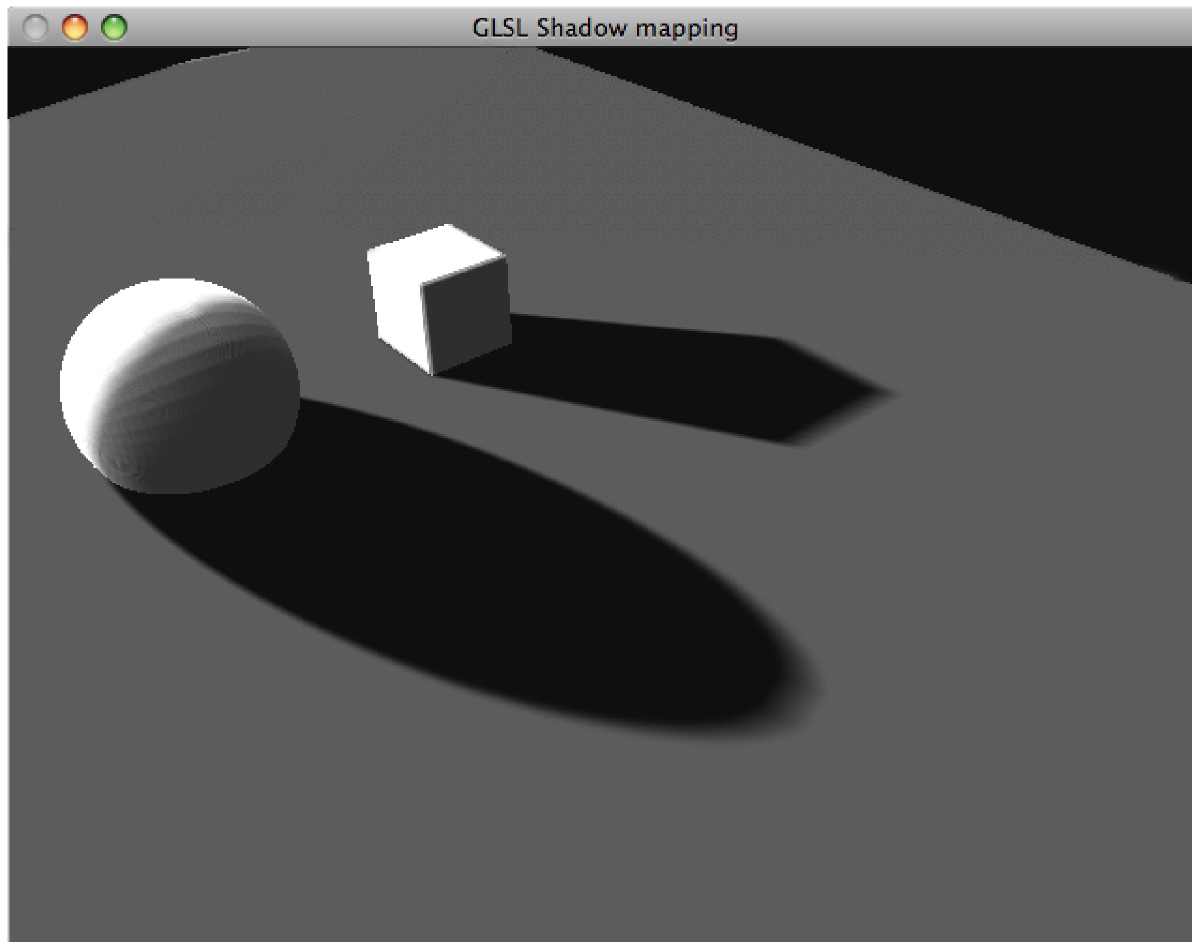
More Complicated, Detailed Materials

- Glossy
- Reflective, refractive
- Rough, bumpy, lots of nooks and crannies
- Wooden



What Can a Fragment Shader Do?

Shadows (including soft edges!)

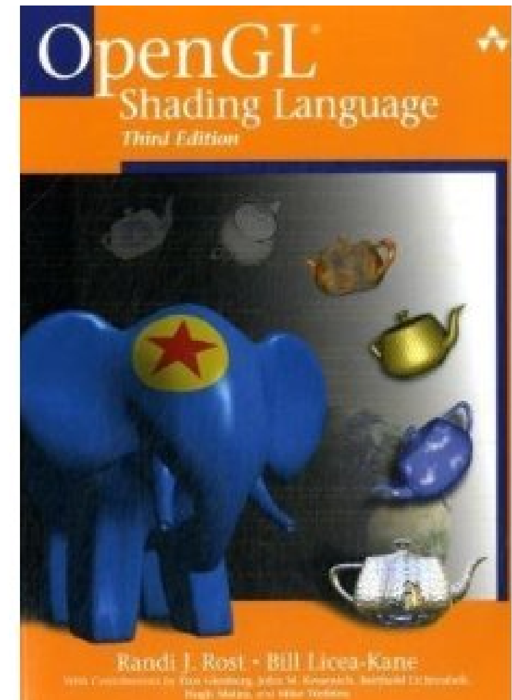


http://www.fabiensanglard.net/shadowmappingPCF/8x8kernel_nVidia.jpg

GLSL

GLSL

- Similar to C/C++
- Used to write shaders
  - Vertex, tessellation, geometry, fragment
  - 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

Green.vert:

```
#version 120

uniform mat4 un_Projection;
uniform mat4 un_ModelView;

attribute vec3 in_Vertex;

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

Vertex Shader

Green.vert:

```
#version 120

uniform mat4 un_Projection;
uniform mat4 un_ModelView;

attribute vec3 in_Vertex;

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

Each time the screen is drawn, this main() function is called once per vertex, as if it were in a for loop.

Vertex Shader

Green.vert:

```
#version 120

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 first thing to do is specify the GLSL version. We use version 1.20 in this class. (Note: other versions can be very different!)

Vertex Shader

Green.vert:

```
#version 120

uniform mat4 un_Projection;
uniform mat4 un_ModelView;

attribute vec3 in_Vertex;

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

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.

Vertex Shader

Green.vert:

```
#version 120

uniform mat4 un_Projection;
uniform mat4 un_ModelView;

attribute vec3 in_Vertex;

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

Attribute variables link to vertex attributes, or data associated with each vertex. This one is set to the vertex position buffer. Each time main() is executed, in_Vertex is set to the vertex currently being processed.

Vertex Shader

Green.vert:

```
#version 120

uniform mat4 un_Projection;
uniform mat4 un_ModelView;

attribute vec3 in_Vertex;

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

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.

This vertex shader just transforms each vertex position by the projection, model, and view matrices.

Fragment Shader

Green.frag:

```
#version 120

uniform mat4 un_Projection;
uniform mat4 un_ModelView;

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

Fragment Shader

Green.frag:

```
#version 120

uniform mat4 un_Projection;
uniform mat4 un_ModelView;

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

Each time the screen is drawn, this main() function is called once per pixel.

Fragment Shader

Green.frag:

```
#version 120

uniform mat4 un_Projection;
uniform mat4 un_ModelView;

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

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.

Fragment Shader

Green.frag:

```
#version 120

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 coordinates
- -RGBA color

vec4(0,1,0,1) constructs an RGBA tuple with R=0, G=1, B=0, A=1, which is green.

OpenGL/GLSL Plumbing

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

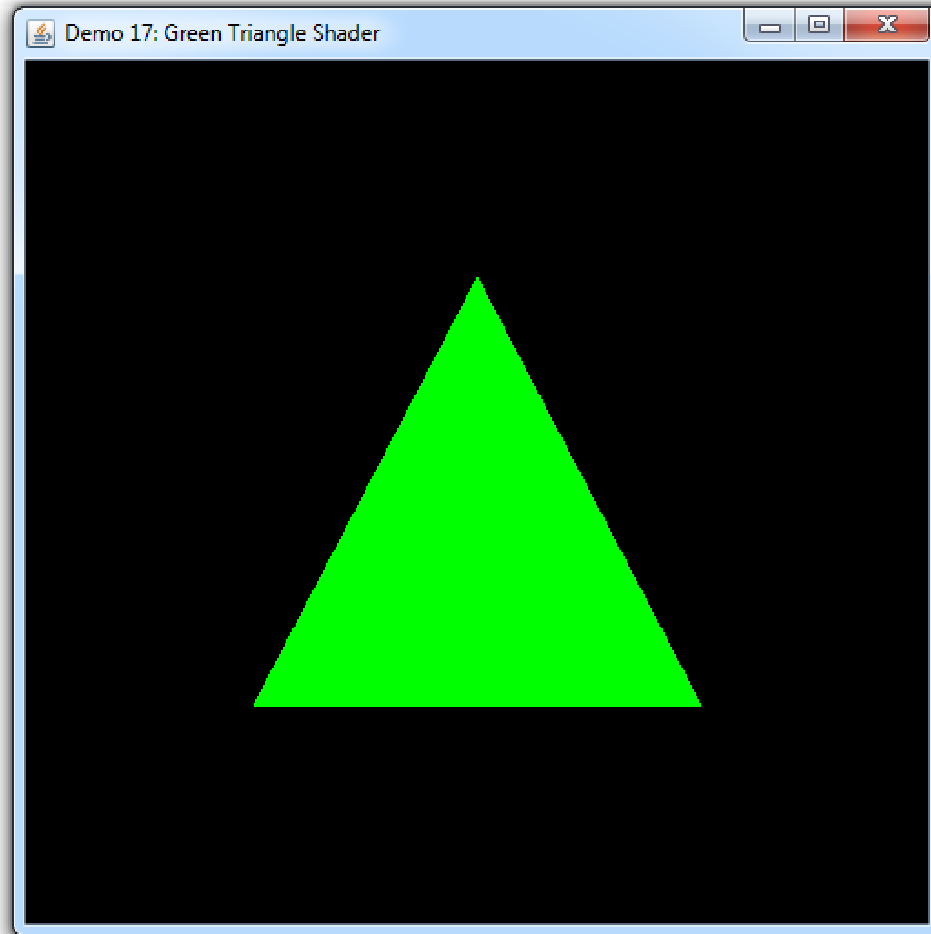
Green.java:

```
// 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();

glDrawElements(...);

GLProgram.unuse();
```

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.

WHAT A PAIN!

CS 4620 Framework

- Contains GLProgram class to abstract OpenGL calls!
  - Added convenience methods
  - Help keep conventions straight
  - Controls mapping between attribute variables and vertex buffers

Now, to create a GLSL program…

- Create a GLProgram object.

```
private GLProgram program;

public void onEntry(GameTime gameTime) {

  program = new GLProgram();
  program.quickCreateResource(
      "cs4620/gl/Grid.vert", // Path to vertex shader
      "cs4620/gl/Grid.frag", // Path to fragment shader
      null);                 // Optional attribute list
}
```

Now, to use a GLSL program…

- Use it. Draw stuff. Unuse it (if need be).

```
public void draw(GameTime gameTime) {
  // Use our GLSL program
  program.use();

  // Tell OpenGL to draw our mesh
  glDrawElements(GL_TRIANGLES, indexCount, GLType.UnsignedInt, 0);

  // Clean up
  GLProgram.unuse();
}
```

GLSL DATA TYPES

GLSL Data Types

- Both GLSL and Java
  - float, int
- GLSL has, but Java has not
  - vec2, vec3, 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.

```
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
```

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
```

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

```
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 = 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 (size known at compile time)
- Use C syntax.

```
float A[4];
A[0] = 5; A[3] = 10;

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

Swizzling

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

```
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.

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