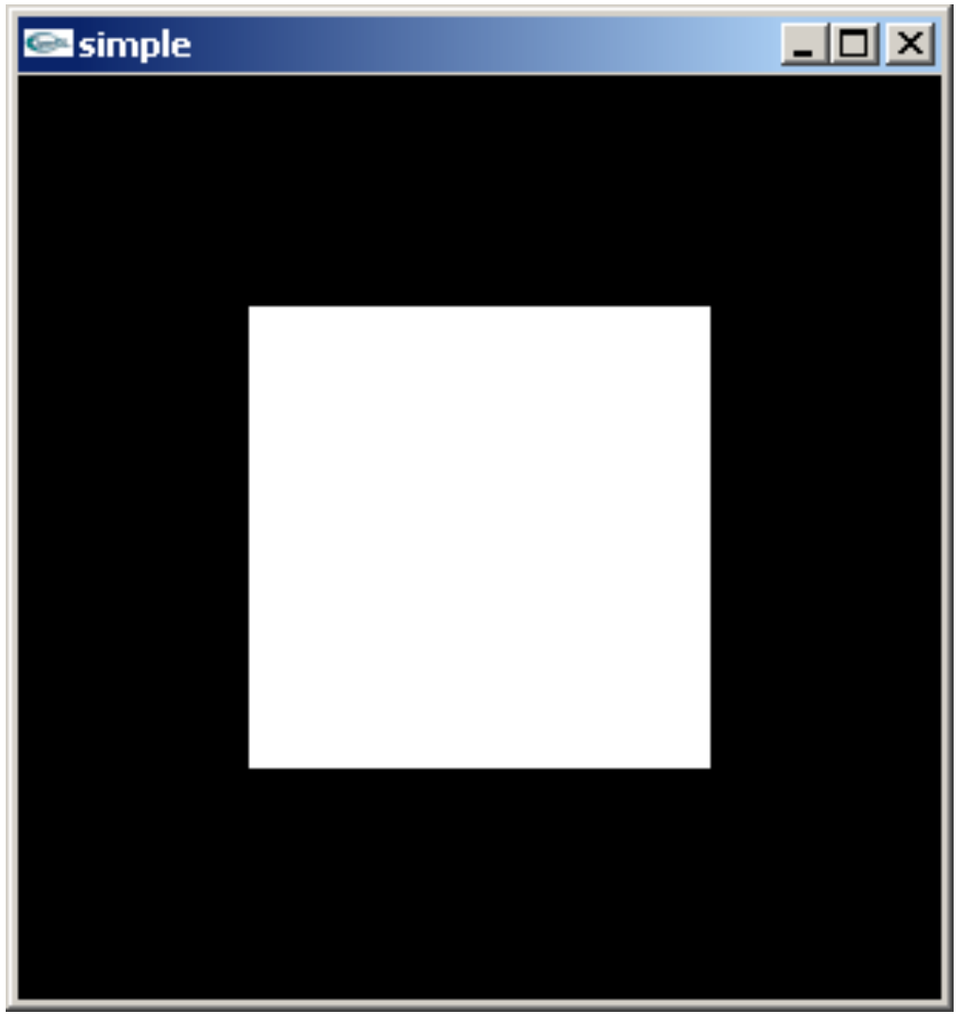
The University of New Mexico

A Simple Program

Generate a square on a solid background



It used to be easy

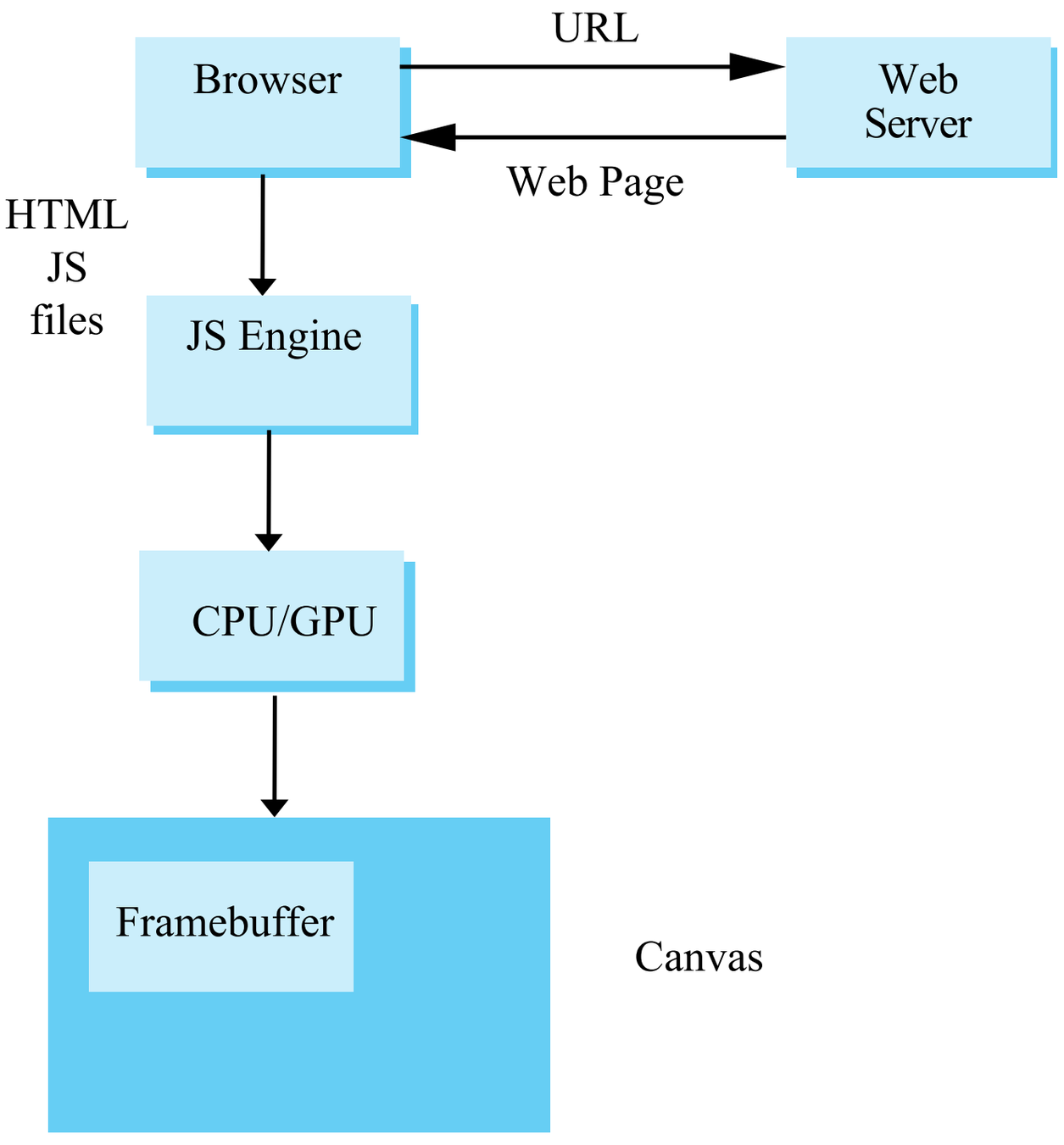
```
#include <GL/glut.h>
void mydisplay(){
      glClear(GL_COLOR_BUFFER_BIT);
      glBegin(GL_QUAD;
            glVertex2f(-0.5, -0.5);
            glVertex2f(-0,5, 0,5);
            glVertex2f(0.5, 0.5);
            glVertex2f(0.5, -0.5);
      glEnd()
}
int main(int argc, char** argv){
      glutCreateWindow("simple");
      glutDisplayFunc(mydisplay);
      glutMainLoop();
}
```

What happened?

- Most OpenGL functions deprecated
  - immediate vs retained mode
  - make use of GPU
- Makes heavy use of state variable default values that no longer exist
  - Viewing
  - Colors
  - Window parameters
- However, processing loop is the same

The University of New Mexico

Execution in Browser



Event Loop

- Remember that the sample program specifies a render function which is an *event listener* or *callback* function

  Every program should have a render callback

  For a static application we need only execute the render function once

  In a dynamic application, the render function can call itself recursively but each redrawing of the display must be triggered by an event

Lack of Object Orientation

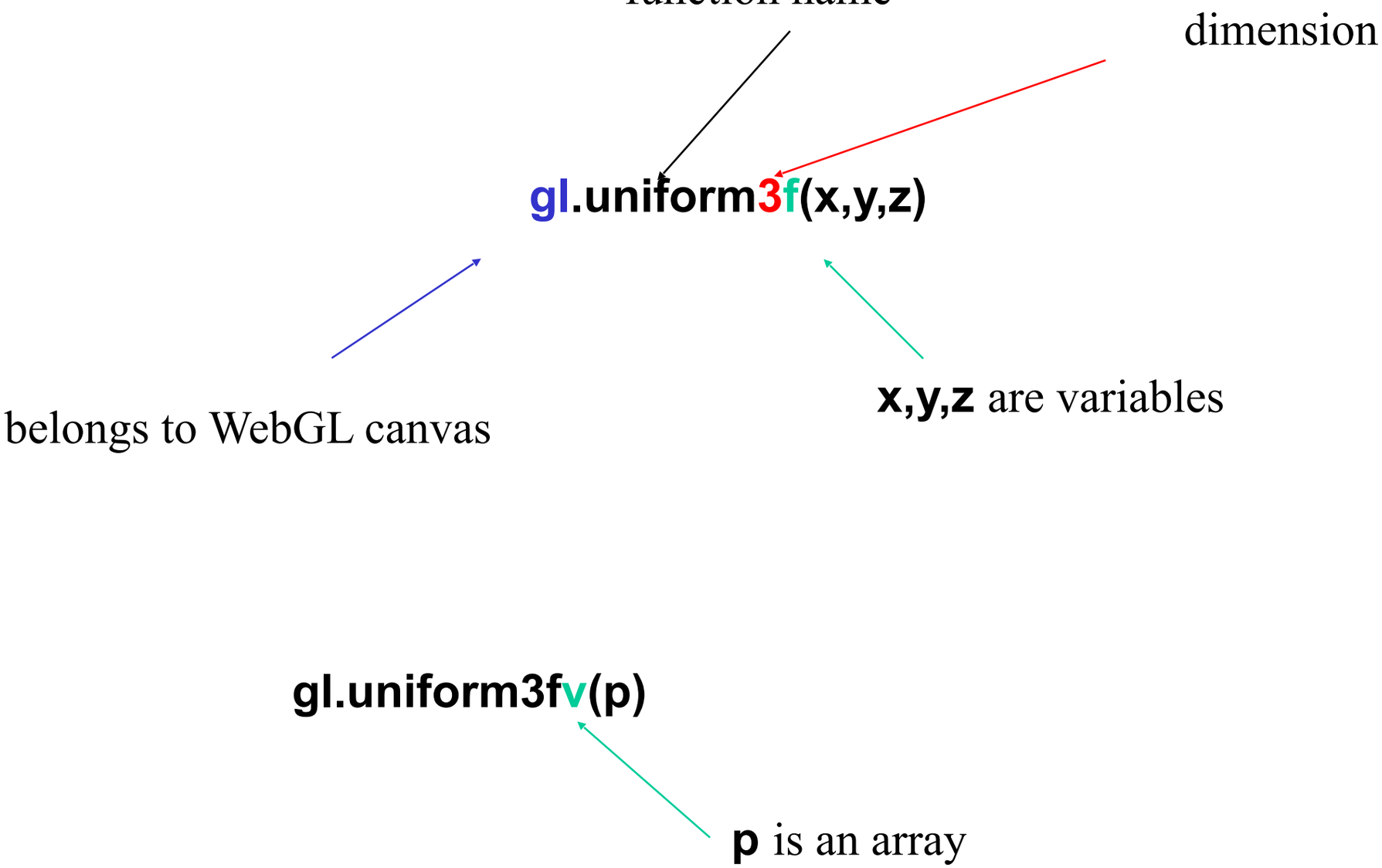
- All versions of OpenGL are not object oriented so that there are multiple functions for a given logical function
- Example: sending values to shaders

  `gl.uniform3f`

  `gl.uniform2i`

  `gl.uniform3dv`
- Underlying storage mode is the same

WebGL function format



WebGL constants

- Most constants are defined in the canvas object

  In desktop OpenGL, they were in #include files such as **`gl.h`**

- Examples

  desktop OpenGL

  - **`glEnable(GL_DEPTH_TEST);`**

  WebGL

  - **`gl.enable(gl.DEPTH_TEST)`**

  `gl.clear(gl.COLOR_BUFFER_BIT)`

WebGL and GLSL

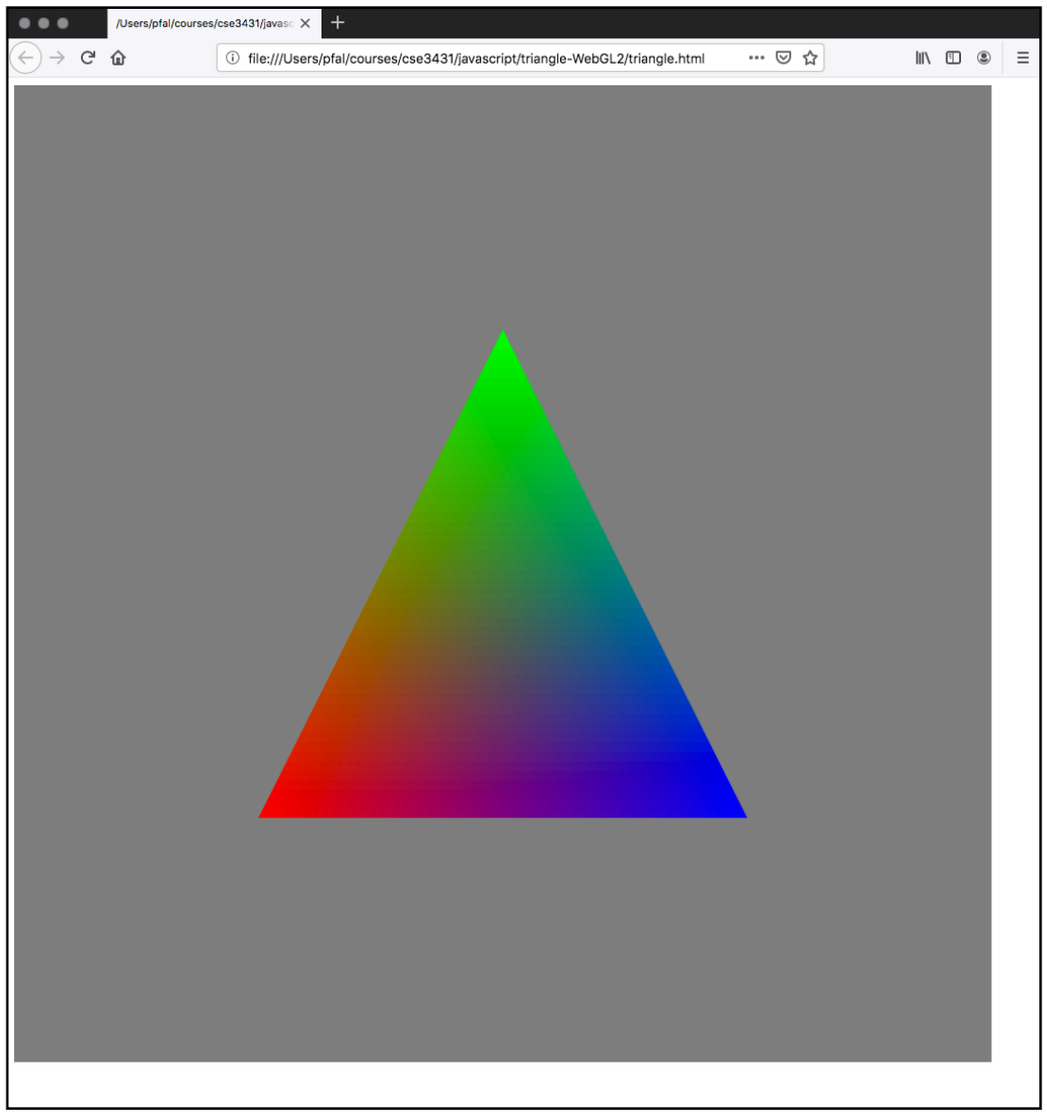
- WebGL requires shaders and is based less on a state machine model than a data flow model
- Most state variables, attributes and related pre 3.1 OpenGL functions have been deprecated
- Action happens in shaders
- Job of application is to get data to GPU

GLSL

- OpenGL Shading Language
- C-like with
  - Matrix and vector types (2, 3, 4 dimensional)
  - Overloaded operators
  - C++ like constructors
- Similar to Nvidia's Cg and Microsoft HLSL
- Code sent to shaders as source code
- WebGL functions compile, link and get information to shaders

The University of New Mexico

Triangle Program



WebGL

- Five steps

  Describe page (HTML file)
  - request WebGL Canvas
  - read in necessary files

  Define shaders (HTML file)
  - could be done with a separate file (browser dependent)

  Compute or specify data (JS file)

  Send data to GPU (JS file)

  Render data (JS file)

triangle.html

```
<!DOCTYPE html>
<html>
<head>
<script id="vertex-shader" type="x-shader/x-vertex">#version 300 es
in vec4 vPosition;
in vec4 vColor ;

out vec4 fColor;
void main()
{
    gl_Position = vPosition;
    fColor = vColor ;
}
</script>
<script id="fragment-shader" type="x-shader/x-fragment">#version 300 es

precision mediump float;
in vec4 fColor ;
layout (location=0) out vec4 fragColor ;
void main()
{
    fragColor = fColor;
}
</script>
```

Shaders

- We assign names to the shaders that we can use in the JS file
- These are trivial pass-through (do nothing) shaders that set the two variables

  gl_Position (built-in)

  fragColor (user defined)
- Note both shaders are full programs
- Note vector type vec4
- Must set precision in fragment shader

triangle.html (cont)

```
<script type="text/javascript" src="../Common/webgl-utils.js"></script>
<script type="text/javascript" src="../Common/initShaders.js"></script>
<script type="text/javascript" src="../Common/MV.js"></script>
<script type="text/javascript" src="triangle.js"></script>
</head>

<body>
<canvas id="gl-canvas" width="512" height="512">
Oops ... your browser doesn't support the HTML5 canvas element
</canvas>
</body>
</html>
```

Calls in webgl-utils.js. If successful opens a 512x512 canvas with WebGL2 support.

The University of New Mexico

Files

- **`../Common/webgl-utils.js`**: Standard utilities for setting up WebGL context in Common directory on website
- **`../Common/initShaders.js`**: contains JS and WebGL code for reading, compiling and linking the shaders
- **`../Common/MV.js`**: our matrix-vector package
- **`triangle.js`**: the application file

triangle.js

```
var gl;
var canvas ;

window.onload = function init(){
      canvas = document.getElementById( "gl-canvas" );

    gl = WebGLUtils.setupWebGL( canvas );
    if ( !gl ) { alert( "WebGL isn't available" );
}
   // Three Vertices

 var vertices = [
     vec4( -0.5, -0.5, 0.0, 1.0 ),
     vec4( 0.0,  0.5, 0.0 , 1.0 ),
     vec4( 0.5, -0.5, 0.0, 1.0 ),
     ];
var vertexColors = [
     vec4( 1.0, 0.0, 0.0, 1.0 ),  // red
     vec4( 0.0, 1.0, 0.0, 1.0 ),  // green
     vec4( 0.0, 0.0, 1.0, 1.0 ),  // blue
   ];
```

The University of New Mexico

Notes

- **`onload`**: determines where to start execution when all code is loaded
- canvas gets WebGL context from HTML file
- vertices use vec4 type in MV.js
- JS array is not the same as a C or Java array

  object with methods

  vertices.length // 4
- Values in clip coordinates

The University of New Mexico

triangle.js (cont)

```
//  Configure WebGL
gl.viewport( 0, 0, canvas.width, canvas.height );
gl.clearColor( 0.5, 0.5, 0.5, 1.0 ); // grey

 //  Load shaders and initialize attribute buffers

var program = initShaders( gl, "vertex-shader", "fragment-shader" );
gl.useProgram( program );

 // Load the data into the GPU
 var bufferId = gl.createBuffer();
 gl.bindBuffer( gl.ARRAY_BUFFER, bufferId );
 gl.bufferData( gl.ARRAY_BUFFER, flatten(vertices), gl.STATIC_DRAW );

 // Associate out shader variables with our data buffer

 var vPosition = gl.getAttribLocation( program, "vPosition" );
 gl.vertexAttribPointer( vPosition,4, gl.FLOAT, false, 0, 0 );
 gl.enableVertexAttribArray( vPosition );
… // similarly for colors
render() ;
};
```

triangle.js (cont)

```
 ...
render() ;
};

var render = function() {
    gl.clear( gl.COLOR_BUFFER_BIT);

    gl.drawArrays( gl.TRIANGLES, 0, 3 );
    requestAnimFrame(render);
}
```

Canvas and OpenGL

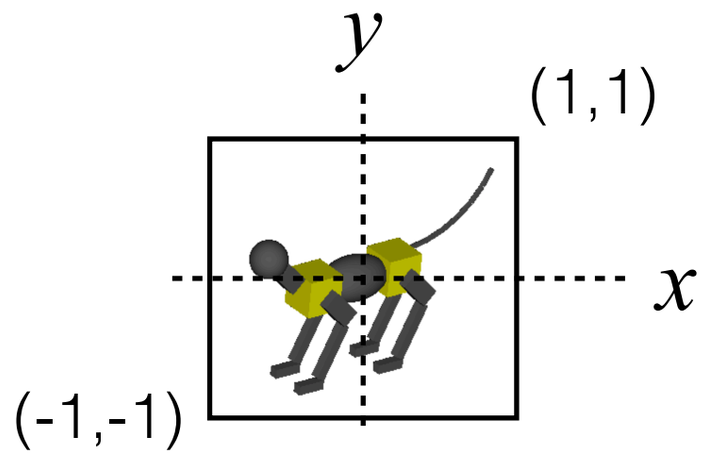
From triangle.html:

```
<canvas id="gl-canvas" width="512" height="512">        // These are pixels!
Oops ... your browser doesn't support the HTML5 canvas element
</canvas>
```

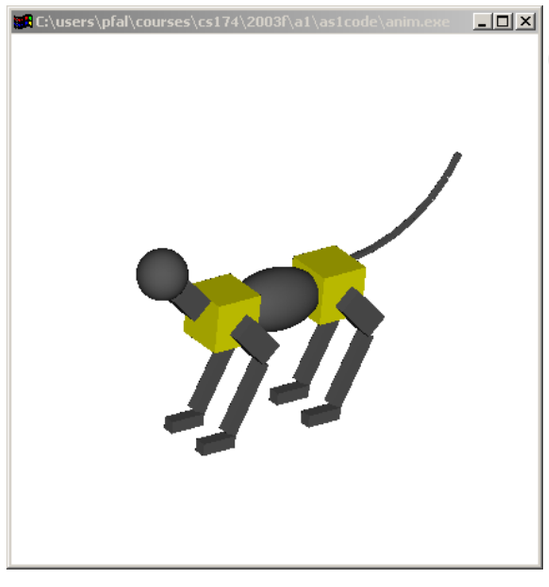
From triangle.js

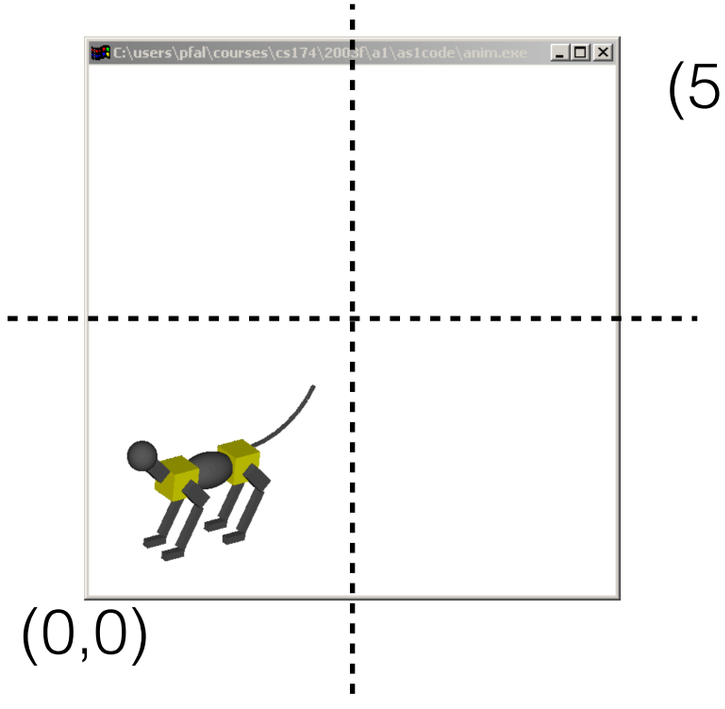
```
gl.viewport( 0, 0, canvas.width, canvas.height );
```

- In **clip coordinates** the viewport in WebGL is [-1,1]
- Normally we set it by the projection transformation



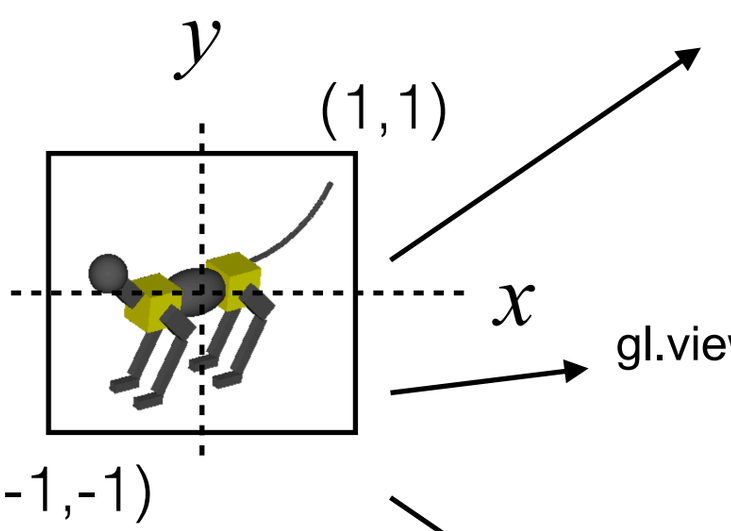
gl.viewport(0, 0, canvas.width, canvas.height); —- gl.viewport(0, 0, canvas.width/2.0, canvas.height/2.0);



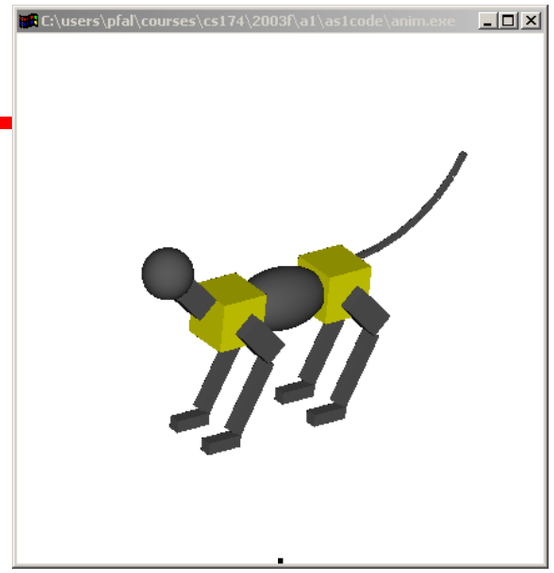


Canvas and OpenGL

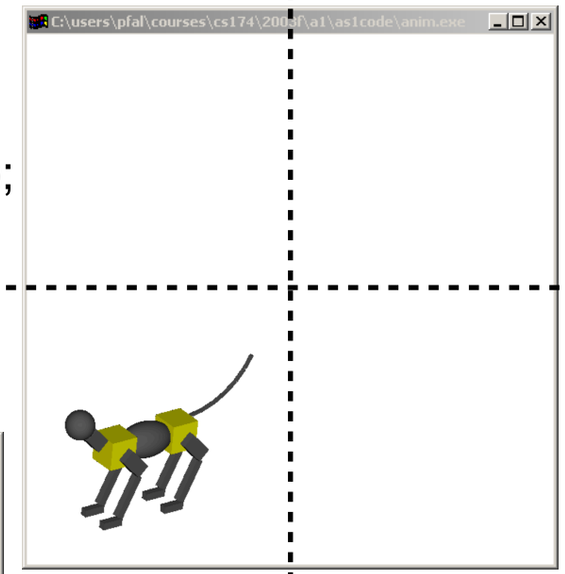
- So, the viewport transformation decides which part of the window the image will cover



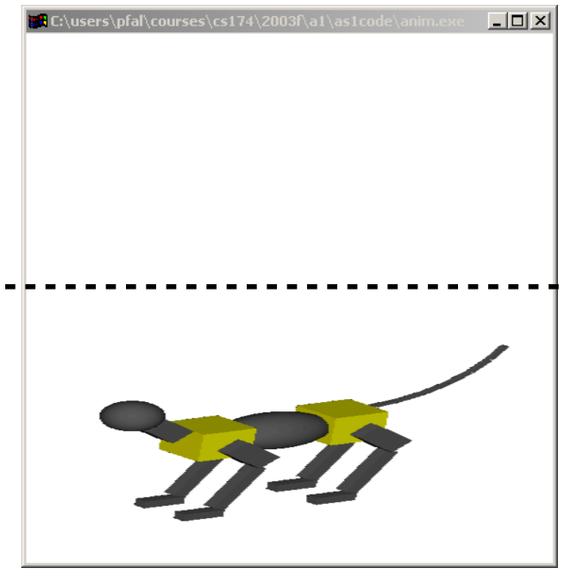
gl.viewport(0, 0, canvas.width, canvas.height);



gl.viewport(0, 0, canvas.width/2.0, canvas.height/2.0);



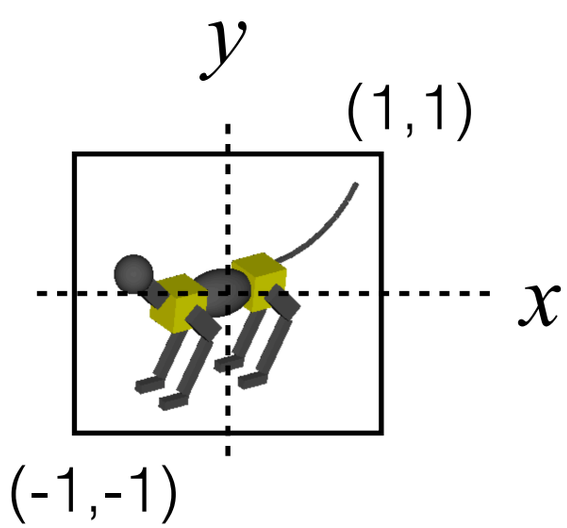
gl.viewport(0, 0, canvas.width, canvas.height/2.0);



The University of New Mexico

Canvas and OpenGL

- Where is z?
- By convention from the screen towards your eyes (right-handed system)



The University of New Mexico

Notes

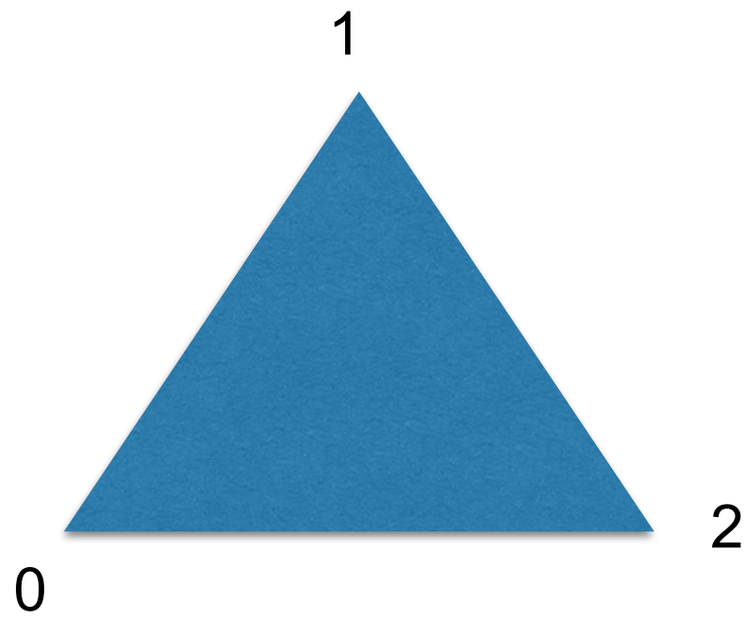
- **`initShaders`** used to load, compile and link shaders to form a program object
- Load data onto GPU by creating a **vertex buffer object** on the GPU

  Note use of flatten() to convert JS array to an array of float32's
- Finally we must connect variable in program with variable in shader

  need name, type, location in buffer

triangle.js (cont)

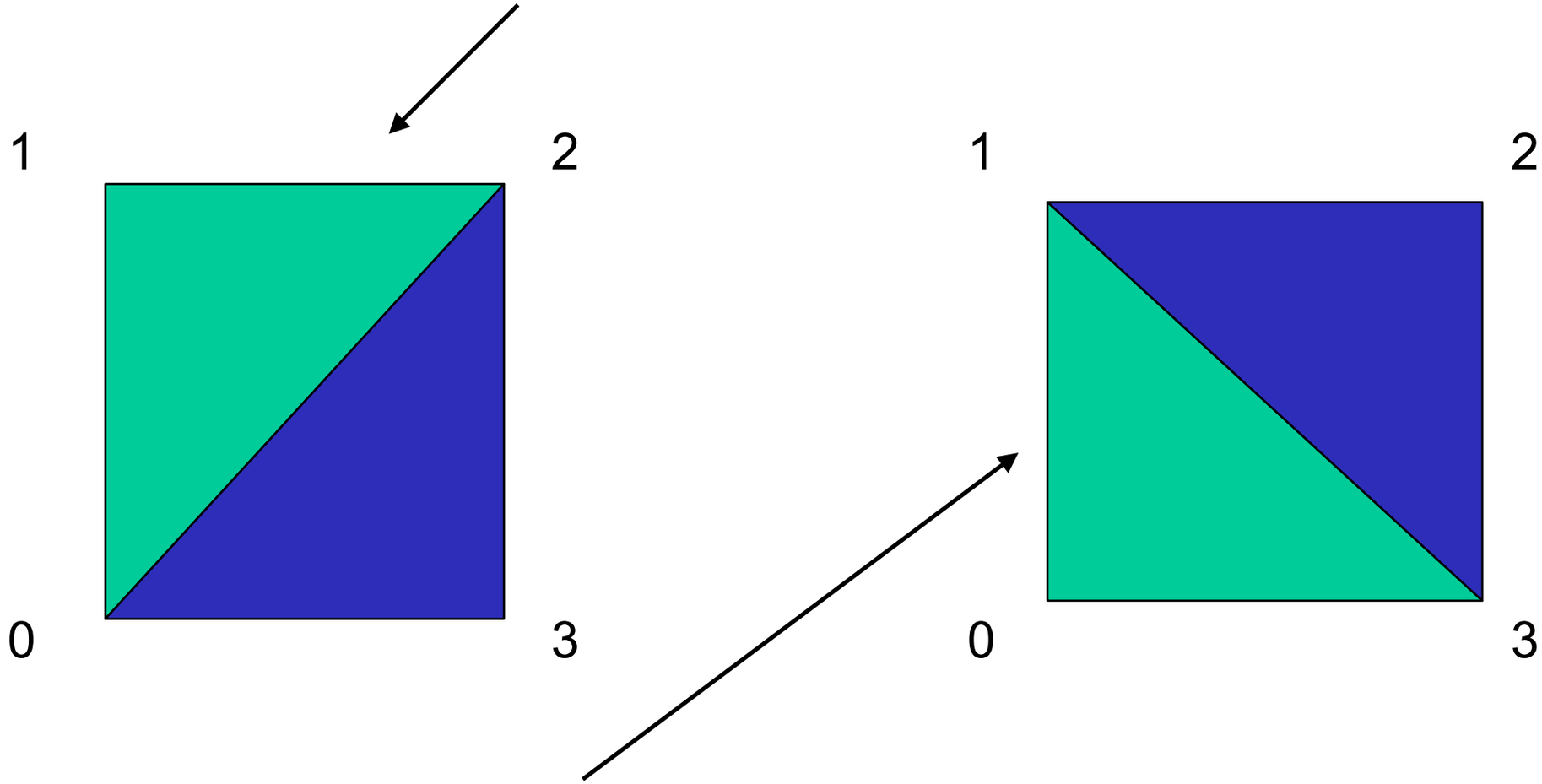
```
    render();
};  // end of onload()

function render() {
    gl.clear( gl.COLOR_BUFFER_BIT );
    gl.drawArrays( gl.TRIANGLES, 0, 3 );
}
```



Triangles, Fans or Strips

gl.drawArrays(gl.TRIANGLES, 0, 6); // 0, 1, 2, 0, 2, 3

gl.drawArrays(gl.TRIANGLE_FAN, 0, 4); // 0, 1 , 2, 3



gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4); // 0, 1, 3, 2

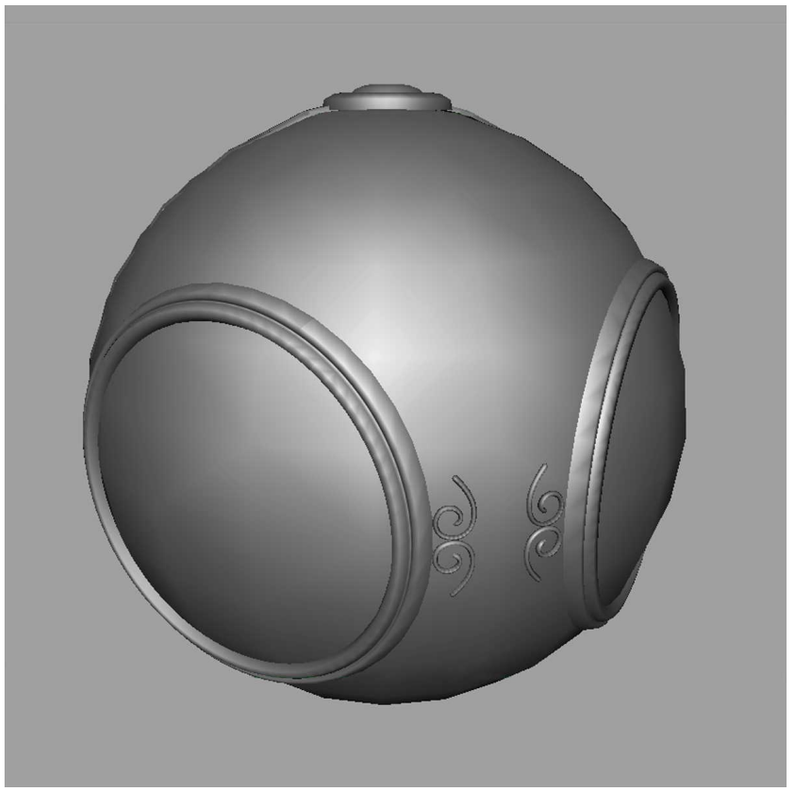
Vertex Shader Applications

- Moving vertices
  - Morphing
  - Wave motion
  - Fractals
- Lighting
  - More realistic models
  - Cartoon shaders

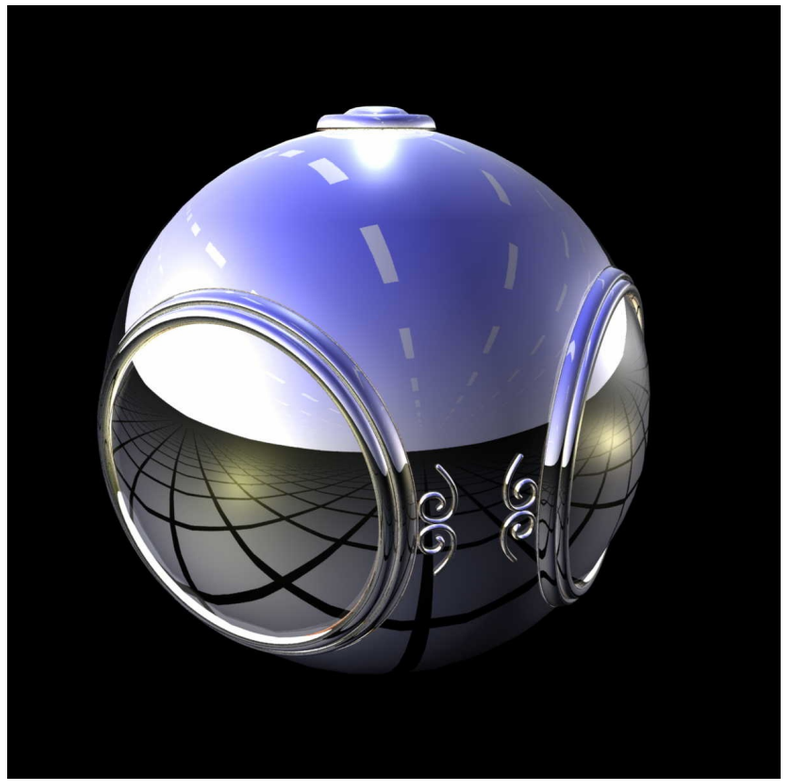
The University of New Mexico

Fragment Shader Applications

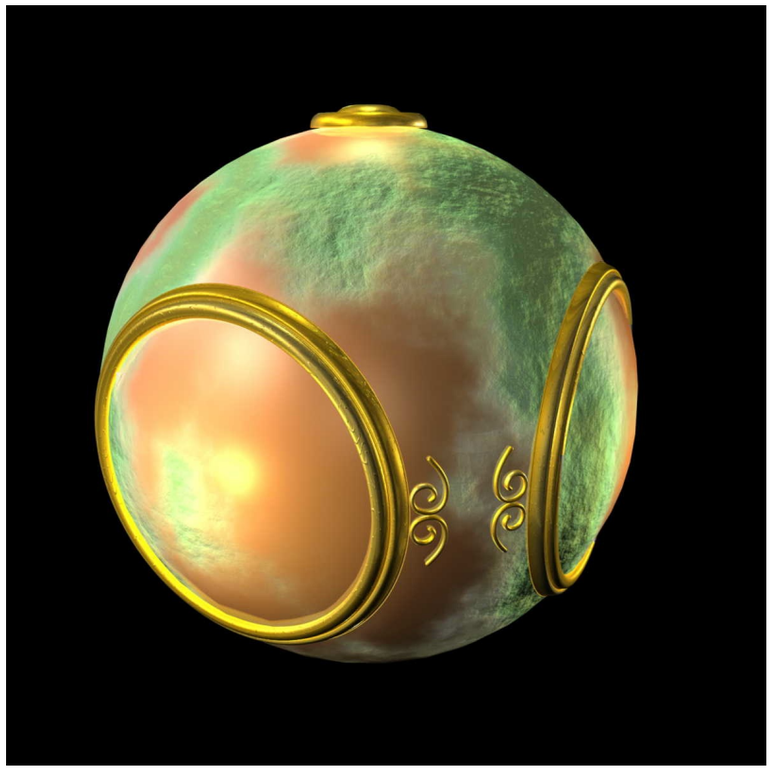
Texture mapping



smooth shading



environment mapping



bump mapping

Writing Shaders

- First programmable shaders were programmed in an assembly-like manner
- OpenGL extensions added functions for vertex and fragment shaders
- Cg (C for graphics) C-like language for programming shaders
  - Works with both OpenGL and DirectX
  - Interface to OpenGL complex
- OpenGL Shading Language (GLSL)

GLSL

- 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, application must provide shaders

Simple Vertex Shader

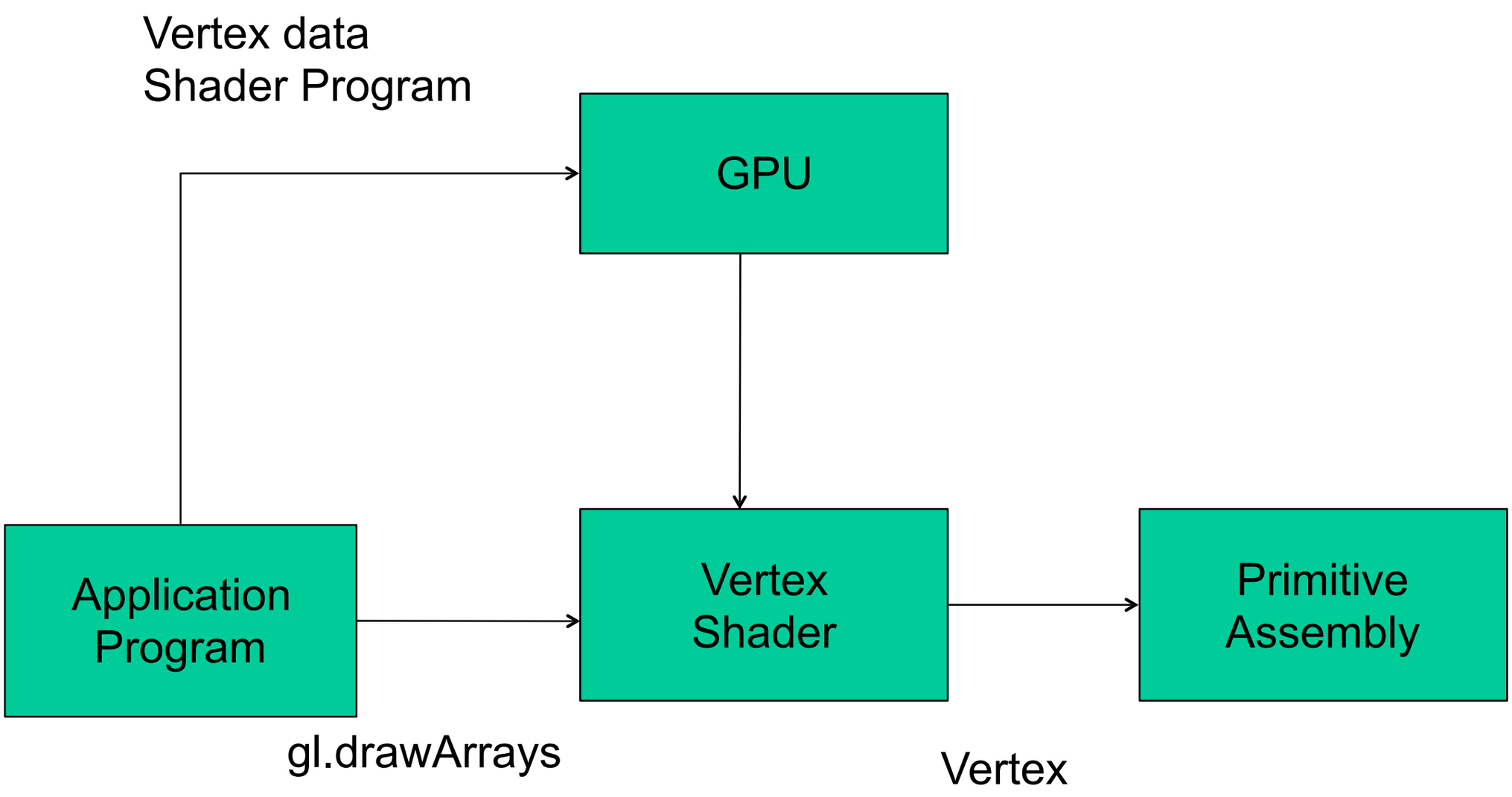
input from application

```
in vec4 vPosition;
void main(void)
{
    gl_Position = vPosition;
}
```

must link to variable in application

built in variable

Execution Model

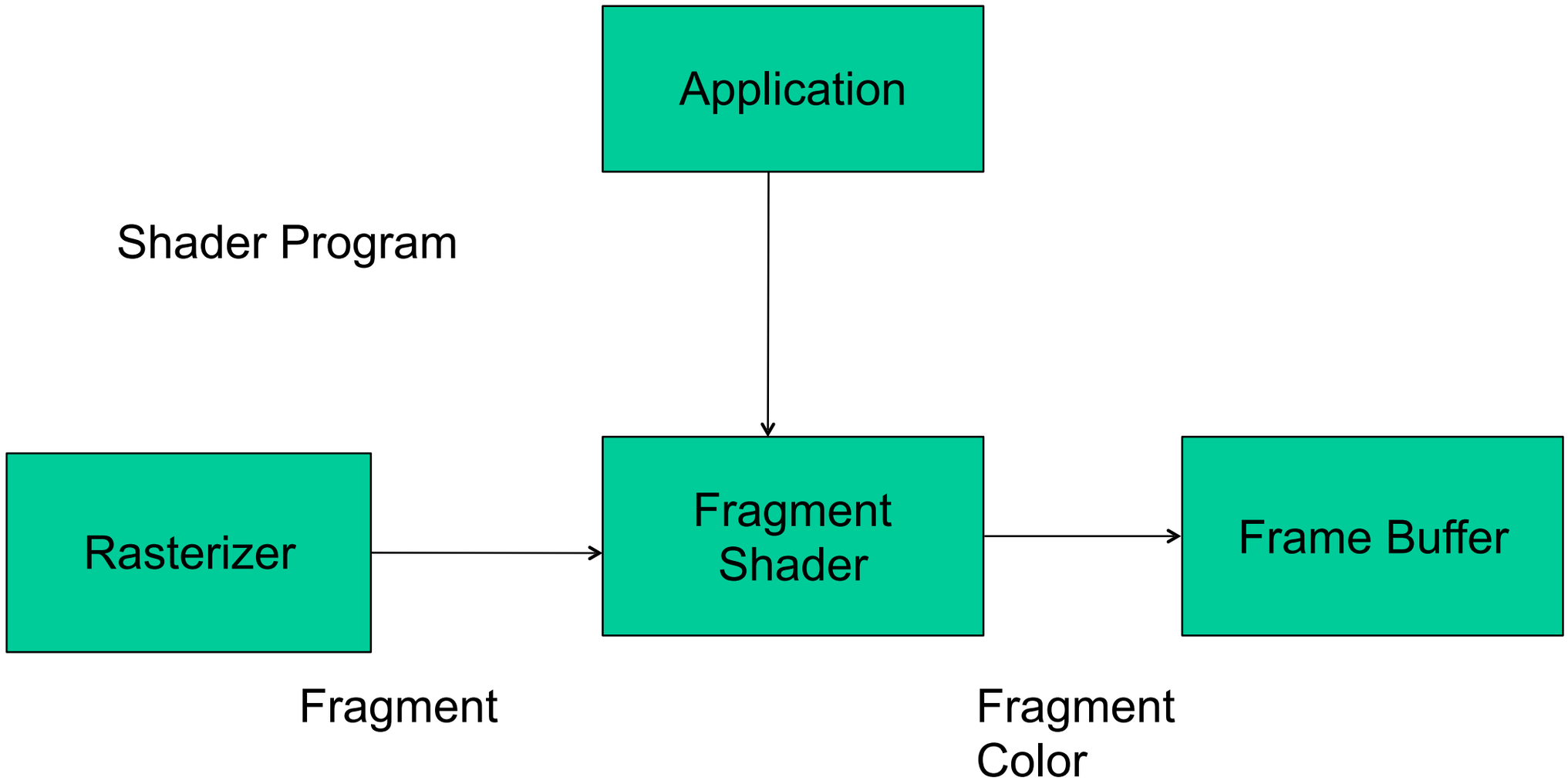


Simple Fragment Program

```
precision mediump float;
layout (location=0) out vec4 fragColor ;
void main(void)
{
   fragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

The University of New Mexico

Execution Model



The University of New Mexico

Data Types

- C types: int, float, bool
- Vectors:
  - float vec2, vec3, vec4
  - Also int (ivec) and boolean (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
- We can use C structs which can be copied back from functions
- Because matrices and vectors are basic types they can be passed into and output from GLSL functions, e.g.

  mat3 func(mat3 a)
- variables passed by copying

Qualifiers

- GLSL has many of the same qualifiers such as **`const`** as C/C++
- Need others due to the nature of the execution model
- Variables can change
  - Once per primitive
  - Once per vertex
  - Once per fragment
  - At any time in the application
- Vertex attributes are interpolated by the rasterizer into fragment attributes

Attributes

- Attribute 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)

  `in float temperature`

  `in vec3 velocity`

  old versions of GLSL used **`attribute`** qualifiers for attribute variables

Uniform Qualified

- 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

Varying Qualified

- Variables that are passed from vertex shader to fragment shader
- Automatically interpolated by the rasterizer
- Older WebGL, GLSL uses the varying qualifier in both shaders

```
varying vec4 color;
```

- Recent versions of WebGL use **out** in vertex shader and **in** in the fragment shader

```
out vec4 color; //vertex shader
in vec4 color;  // fragment shader
```

Our Naming Convention

- Input attributes (*in*) passed to vertex shader have names beginning with v (v Position, vColor) in both the application and the shader

  Note that these are different entities with the same name

- Output attributes (*out* in vshader, *in* in fshader) variables begin with f (fColor) in both shaders

  must have same name

- Uniform variables are unadorned and can have the same name in application and shaders

Example: Vertex Shader

```
in vec4 vPosition ;
in vec4 vColor;
out vec4 fColor;
void main()
{
  gl_Position = vPosition;
  fColor = vColor;
}
```

Corresponding Fragment Shader

```
precision mediump float;

in vec4 fColor;
layout (location=0) out vec4 fragColor;
void main()
{
  fragColor = fColor;
}
```

Sending Colors from Application

```
var cBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
gl.bufferData( gl.ARRAY_BUFFER, flatten(colors),
                                gl.STATIC_DRAW );

var vColor = gl.getAttribLocation( program, "vColor" );
gl.vertexAttribPointer( vColor, 4, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vColor );
```

*//***glVertexAttribPointer**(GLuint *index*, GLint *size*, GLenum *type*, GLboolean *normalized*, GLsizei *stride*, const GLvoid * *pointer*);

Sending a Uniform Variable

```
// in application

vec4 color = vec4(1.0, 0.0, 0.0, 1.0);
colorLoc = gl.getUniformLocation( program, ”color" );
gl.uniform4f( colorLoc, color);
```

```
// in fragment shader (similar in vertex shader)

uniform vec4 color;

void main()
{
    vec4 c = color;
…..
}
```

Operators and Functions

- Standard C functions
  - Trigonometric
  - Arithmetic
  - Normalize, reflect, length
- 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
```

Swizzling and Selection

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

  x, y, z, w

  r, g, b, a

  s, t, p, q

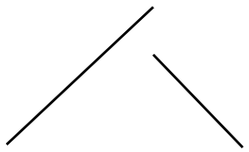
  `a[2], a.b, a.z, a.p` are the same

- **Swizzling** operator lets us manipulate components

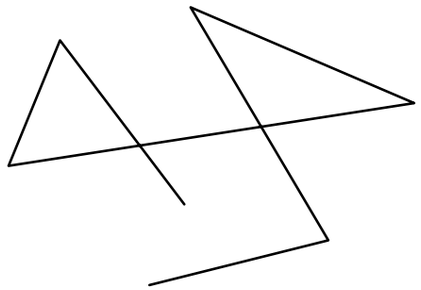
```
vec4 a, b;
a.yz = vec2(1.0, 2.0);
b = a.yxzw;
```

WebGLPrimitives

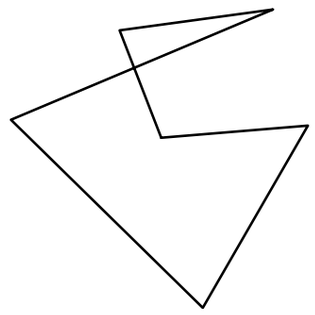
GL_POINTS



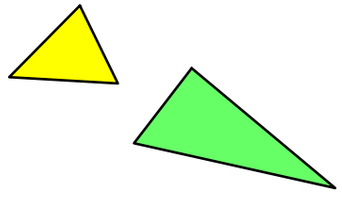
GL_LINES



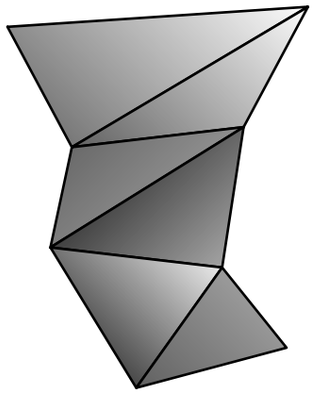
GL_LINE_STRIP



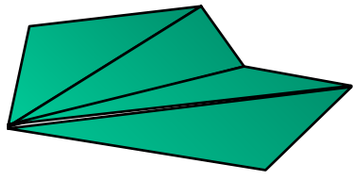
GL_LINE_LOOP



GL_TRIANGLES



GL_TRIANGLE_STRIP



GL_TRIANGLE_FAN

Polygon Issues

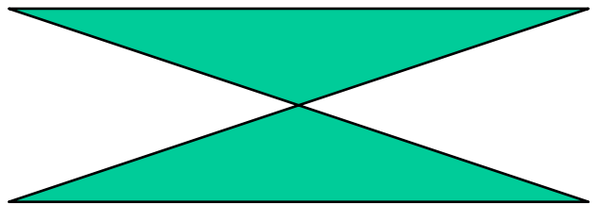
- WebGL will only display triangles

  Simple: edges cannot cross

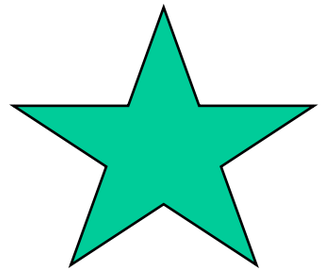
  Convex: All points on line segment between two points in a polygon are also in the polygon

  Flat: all vertices are in the same plane

- Application program must tessellate a polygon into triangles (triangulation)
- OpenGL 4.1 contains a tessellator but not WebGL



nonsimple polygon



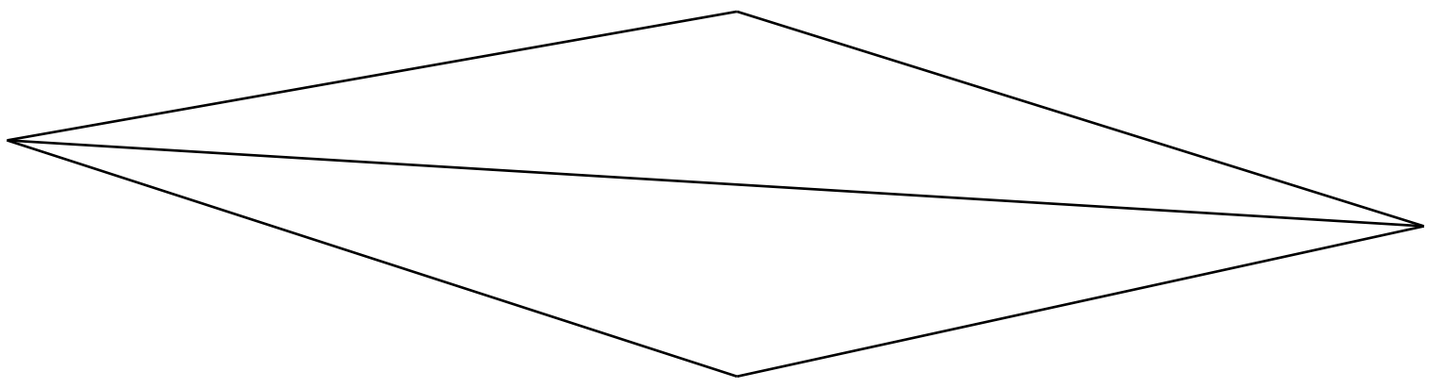
nonconvex polygon

Polygon Testing

- Conceptually simple to test for simplicity and convexity
- Time consuming
- Earlier versions assumed both and left testing to the application
- Present version only renders triangles
- Need algorithm to triangulate an arbitrary polygon

Good and Bad Triangles

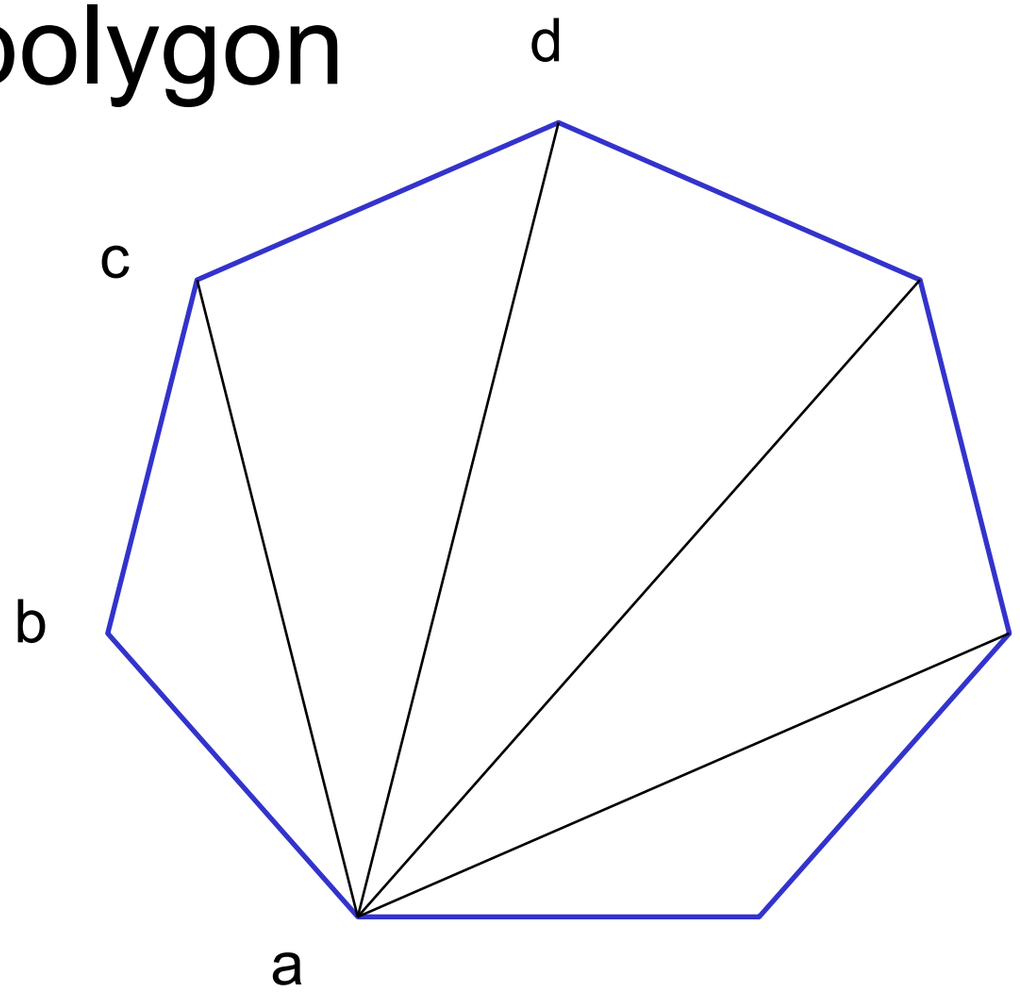
- Long thin triangles render badly



- Equilateral triangles render well
- Maximize minimum angle
- Delaunay triangulation for unstructured points

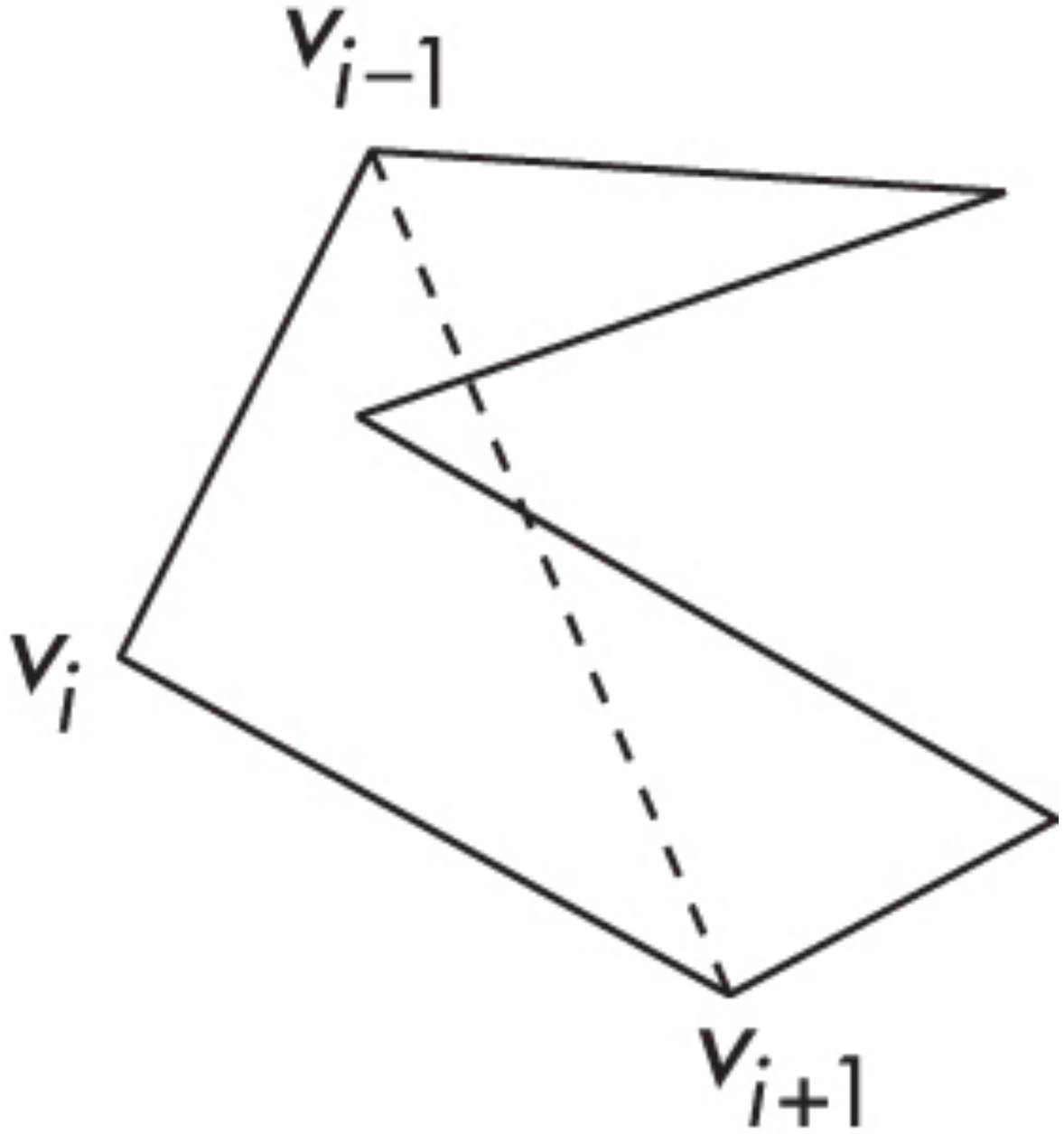
Triangularization

- Convex polygon



- Start with abc, remove b, then acd, ….

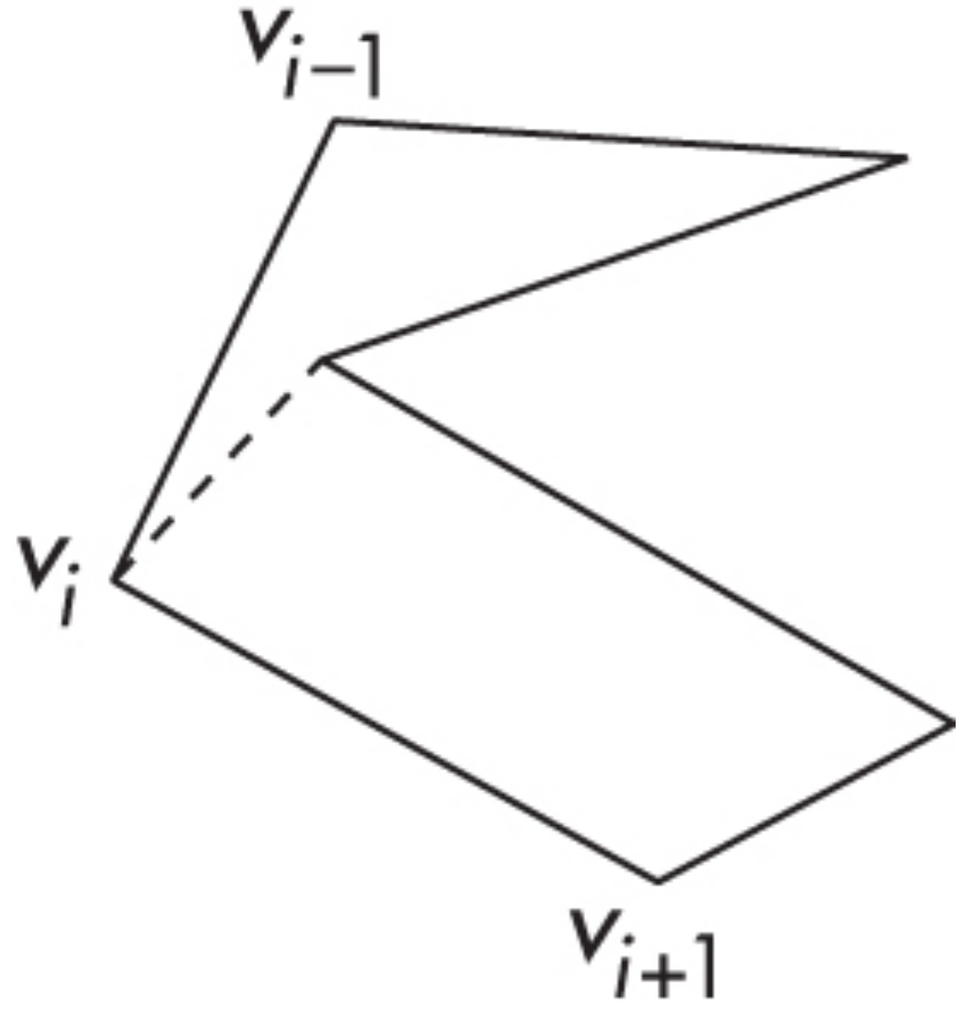
Non-convex (concave)



The University of New Mexico

Recursive Division

- Find leftmost vertex and split



Linking Shaders with Application

- Read shaders
- Compile shaders
- Create a program object
- Link everything together
- Link variables in application with variables in shaders
  - Vertex attributes
  - Uniform variables

Program Object

- Container for shaders
  - Can contain multiple shaders
  - Other GLSL functions

```
var program = gl.createProgram();

gl.attachShader( program, vertShdr );
gl.attachShader( program, fragShdr );
gl.linkProgram( program );
```

The University of New Mexico

Reading a Shader

- Shaders are added to the program object and compiled
- Usual method of passing a shader is as a null-terminated string using the function
- gl.shaderSource(fragShdr, fragElem.text);
- If shader is in HTML file, we can get it into application by getElementById method
- If the shader is in a file, we can write a reader to convert the file to a string

Adding a Vertex Shader

```
var vertShdr;
var vertElem =
    document.getElementById( vertexShaderId );

vertShdr = gl.createShader( gl.VERTEX_SHADER );

gl.shaderSource( vertShdr, vertElem.text );
gl.compileShader( vertShdr );

// after program object created
gl.attachShader( program, vertShdr );
```

Shader Reader

- Following code may be a security issue with some browsers if you try to run it locally

  Cross Origin Request

```
function getShader(gl, shaderName, type) {
        var shader = gl.createShader(type);
        shaderScript = loadFileAJAX(shaderName);
        if (!shaderScript) {
            alert("Could not find shader source:
                    "+shaderName);
        }
}
```

Precision Declaration

- In GLSL for WebGL we must specify desired precision in fragment shaders
  - artifact inherited from OpenGL ES
  - ES must run on very simple embedded devices that may not support 32-bit floating point
  - All implementations must support mediump
  - No default for float in fragment shader
- Can use preprocessor directives (#ifdef) to check if highp supported and, if not, default to mediump

Pass Through Fragment Shader

```
#ifdef GL_FRAGMENT_SHADER_PRECISION_HIGH
  precision highp float;
#else
  precision mediump float;
#endif

layout (location=0) out vec4 fragColor ;
in vec4 fcolor;
void main(void)
{
    fragColor = fcolor;
}
```

The University of New Mexico

Triangle Program