

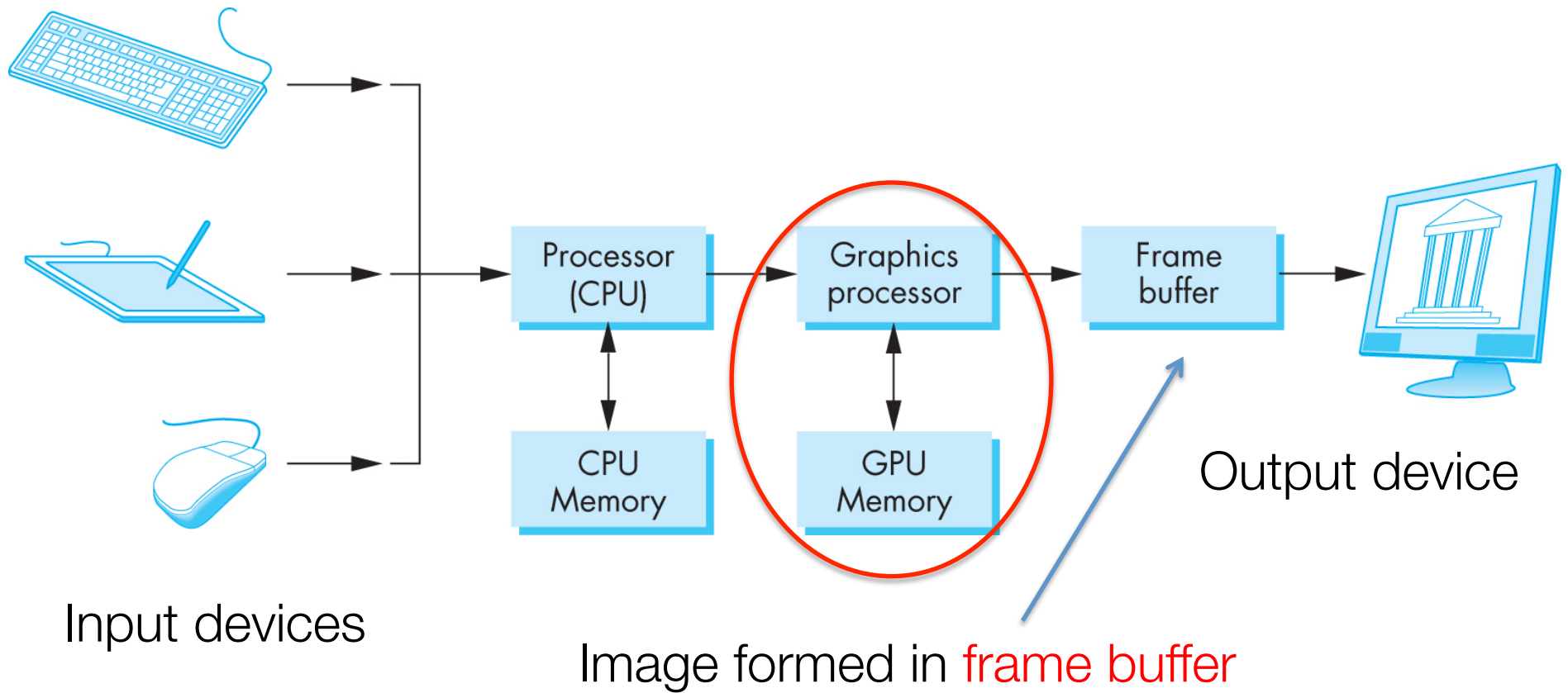


Learning Objectives

- Students completing this lecture will be able to
 - Explain these terms in your own words: vertex, fragment, VBO, qualifiers (attribute, uniform, varying), shader
 - Explain the relationships among HTML code, JS code, and shader code in a WebGL program
 - Describe the roles of vertex shader and fragment shader and the process of sending data to GPU
 - Write shader programs to arrange triangle primitives into meaningful figures

Computer Graphics using WebGL

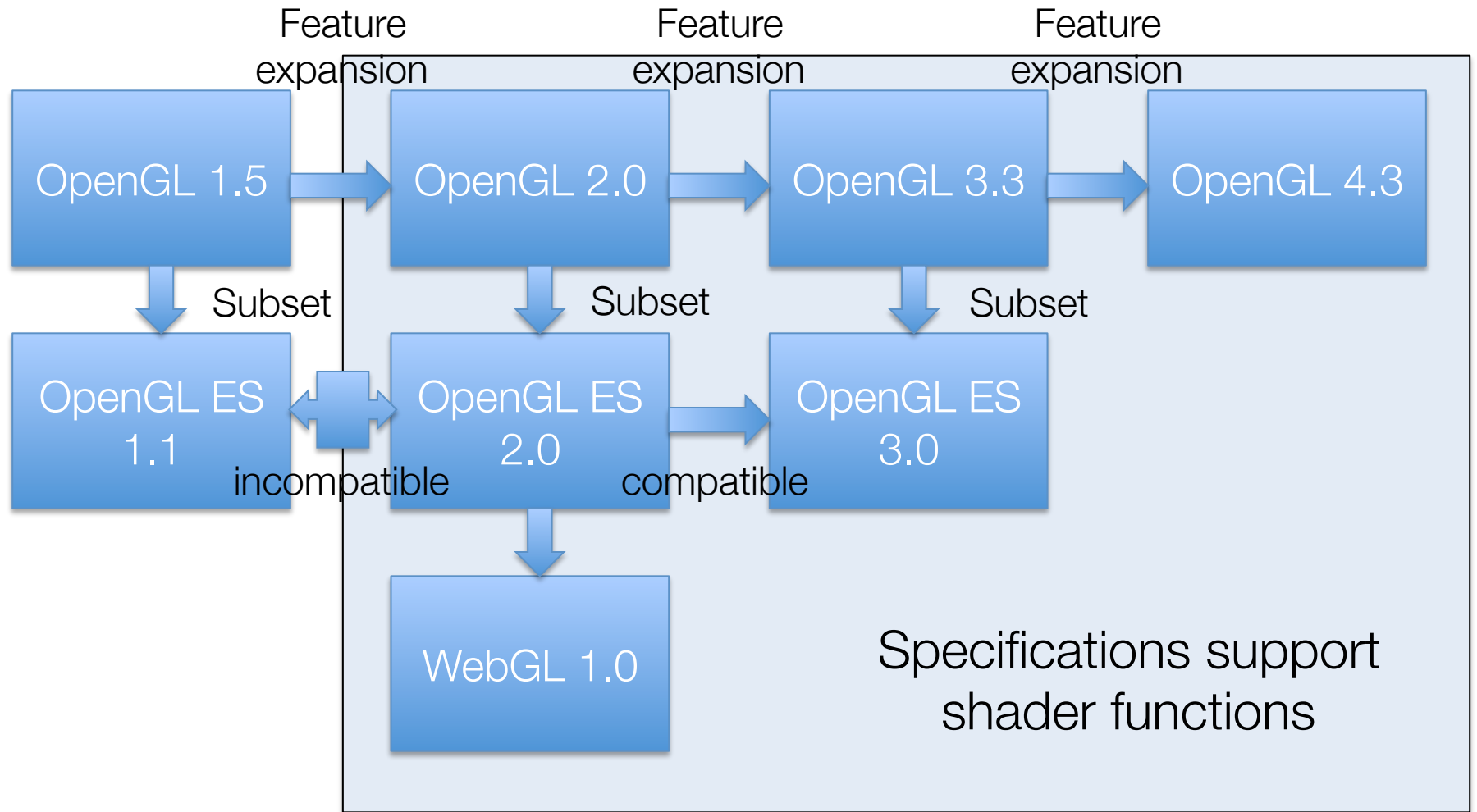
Basic Graphics System



WebGL

- Shader-based language
 - Integrate with HTML5
 - Code runs in latest browsers
 - Makes use of local hardware
 - No system dependencies
- **Top-down**, programming-oriented approach
 - Starting with a high-level description of what an application is supposed to do
 - Breaking the specification down into simpler and simpler pieces, until a level has been reached that corresponds to the primitives of the programming language to be used

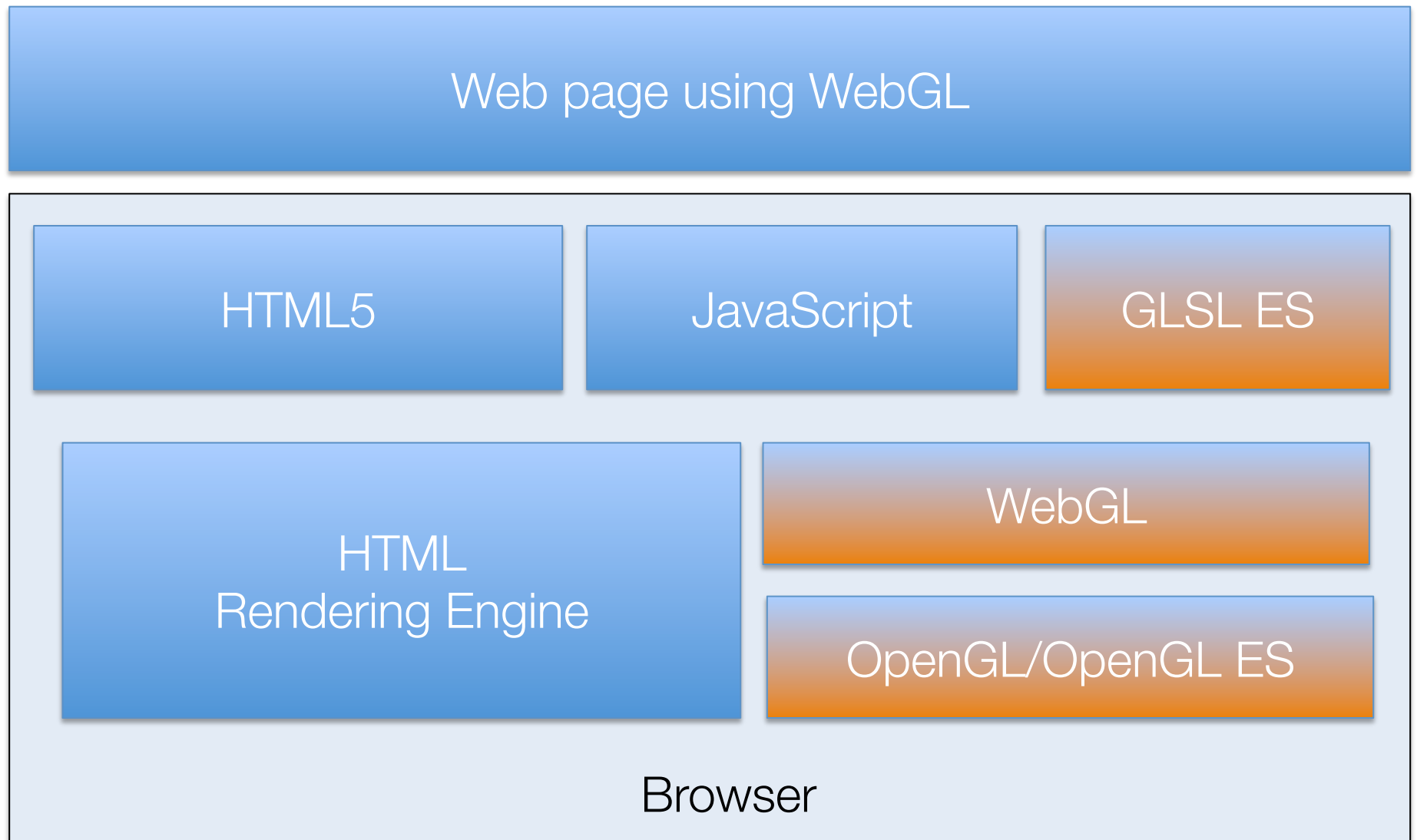
OpenGL, OpenGL ES and WebGL



WebGL

- WebGL is based on OpenGL **ES** (for **Embedded Systems**) 2.0 version
- OpenGL ES have been standardized as a subset of specific versions of OpenGL
- Programmable shader functions are highlight of OpenGL 2.0
- Shading language used in OpenGL ES 2.0 is based on **OpenGL Shading Language (GLSL)**

Software Architecture



Coding in WebGL



- Can run WebGL on any recent browser
 - Chrome
 - Firefox (+ Firebug), we use this in our class
 - Safari
 - IE
- Code written in JavaScript
- JS runs within browser
 - Use local resources

Advantages of WebGL

- Start developing 3D graphics applications using only a text editor and browser
- Do not need to deal with compilation issues with OpenGL under different operating systems
- Easily publish the 3D graphics applications using standard web technologies
- Leverage the full functionality of the browser
- A lot of material is available for learning and using WebGL

Disadvantages of WebGL

- Very limited debugging environment and capability available (you need to be very careful when writing WebGL programs!)
- Share your WebGL program = share your source code (what if I want to share the program but not share the code?)
- Anything else?

References

- *Interactive Computer Graphics (7th Edition)*, recommended textbook
- *The OpenGL Programmer's Guide (8th Edition)*
- *OpenGL ES 2.0 Programming Guide*
- *WebGL Programming Guide*
- *WebGL Beginner's Guide* (ebook available via ND library)
- *WebGL: Up and Running*
- *JavaScript: The Definitive Guide* (ebook available via ND library)

Web Resources

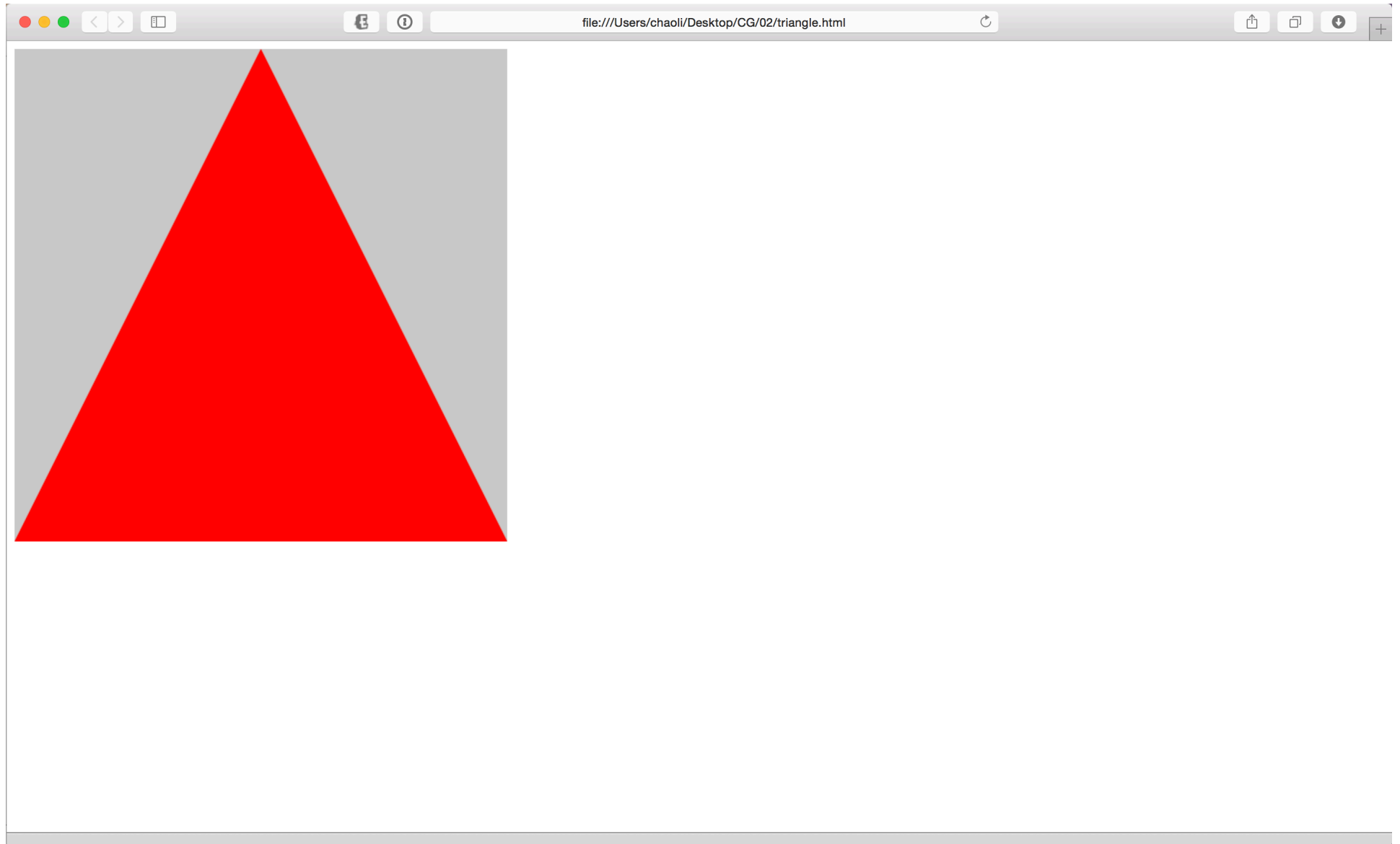
- www.cs.unm.edu/~angel/
- www.cs.unm.edu/~angel/WebGL/7E
- www.opengl.org
- <http://get.webgl.org> (check whether your browser supports WebGL or not)
- www.khronos.org/webgl
- www.chromeexperiments.com/webgl
- www.learningwebgl.com

Draw a Triangle

Example: Draw a Triangle

- Each application consists of (at least) two files: a HTML file and a JavaScript file
- HTML
 - Describe page
 - Include utilities
 - Include shaders
- JavaScript (JS)
 - Contain the graphics

Example: triangle.html



Example Code (HTML File)

```
<!DOCTYPE html>
<html>
<head>
// vertex shader
<script id="vertex-shader" type="x-shader/x-vertex">
attribute vec4 vPosition;
void main() {
    gl_Position = vPosition;
}
</script>
// fragment shader
<script id="fragment-shader" type="x-shader/x-fragment">
precision mediump float;
void main() {
    gl_FragColor = vec4( 1.0, 0.0, 0.0, 1.0 ); // red
}
</script>
```


Shaders

- Vertex shader
 - Program that describes the traits (position, colors, and so on) of a vertex
 - A **vertex** is a point in 2D/3D space, such as the corner or intersection of a 2D/3D shape
- Fragment shader
 - Program that deals with per-fragment processing such as lighting
 - A **fragment** can be considered as a kind of pixel (picture element)

Shaders

- We assign names to the shaders that we can use in the JS file
- These are trivial pass-through (do nothing) shaders which set the two required built-in variables
 - `gl_Position`
 - `gl_FragColor`
- Note both shaders are full programs
- Notice vector type `vec4` in the shader and vector type `vec2` in the application
- Must set precision in fragment shader

HTML File (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>
```

Files



- `../Common/webgl-utils.js`: standard utilities for setting up WebGL context (**provided already, always include**)
- `../Common/initShaders.js`: contains JS and WebGL code for reading, compiling and linking the shaders (**provided already, always include**)
- `../Common/MV.js`: our matrix-vector package (**provided already, always include**)
- `triangle.js`: the application file (**we write it**)

JS File

```
"use strict"; // we do this for all WebGL code we write
var gl;
var vertices;

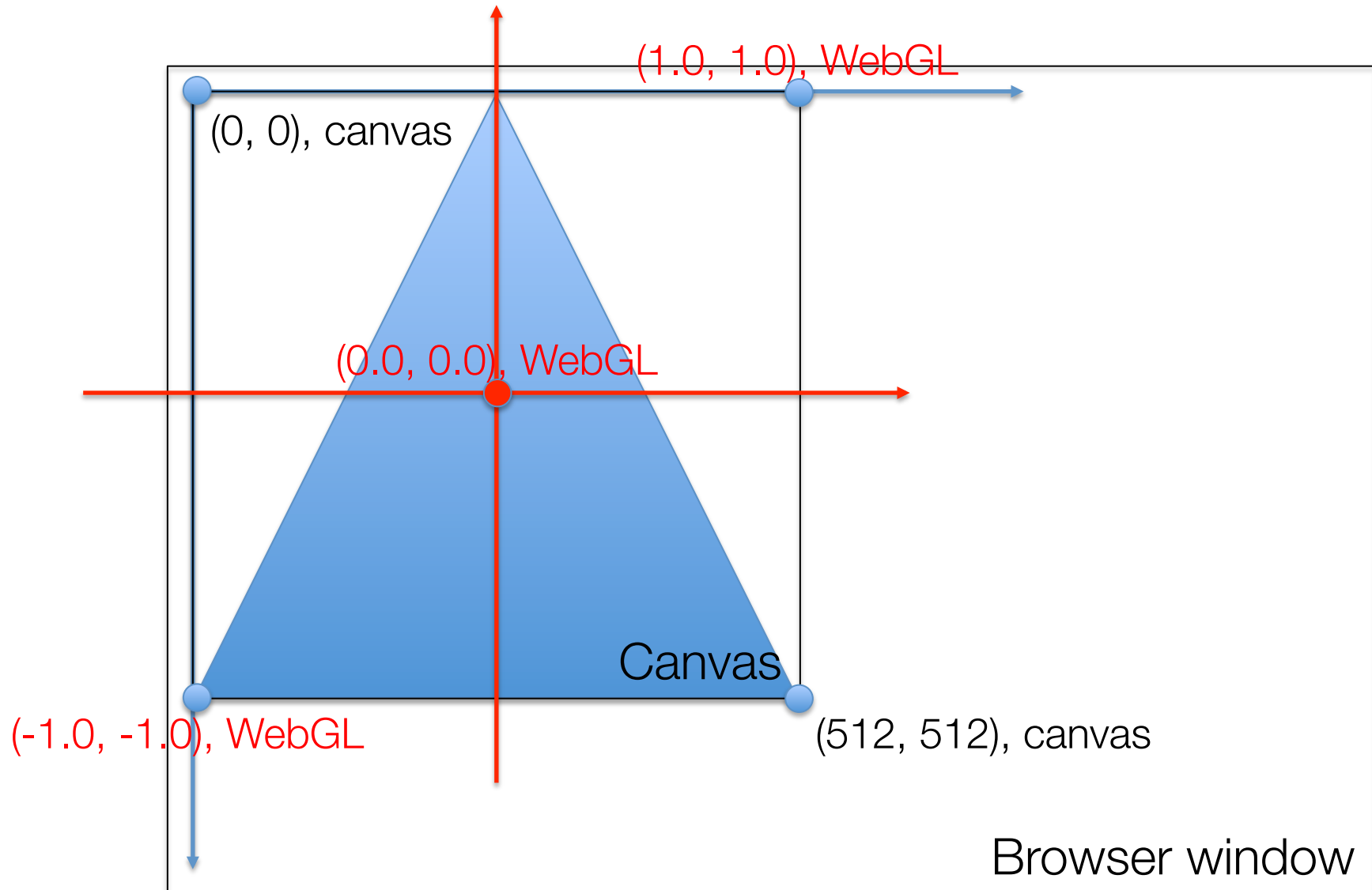
window.onload = function init() {
    var canvas = document.getElementById("gl-canvas");
    gl = WebGLUtils.setupWebGL( canvas );
    if ( !gl ) { alert("WebGL isn't available"); }

    // Three vertices
    var vertices = [
        vec2( -1, -1 ),
        vec2(  0,  1 ),
        vec2(  1, -1 )
    ];
};
```

Notes

- `onload`: determines where to start execution when all code is loaded
- Canvas gets WebGL context from HTML file
- Vertices use `vec2` type in `MV.js`
- JS array is not the same as a C or Java array
 - Object with methods
 - `vertices.length // 3`

WebGL Coordinate System



JS File (cont)

```
// Configure WebGL
gl.viewport( 0, 0, canvas.width, canvas.height );
gl.clearColor( 0.8, 0.8, 0.8, 1.0 ); // gray

// 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 );
```


Notes

- `initShaders` used to load, compile and link shaders to form a program object
- Load data onto GPU by creating a **vertex buffer object** (VBO) on the GPU
 - Note the use of `flatten()` to convert JS array to an array of `float32`'s

Typed Arrays

- JS has typed arrays that are like C arrays

```
var a = new Float32Array(3)
```

```
var b = new Uint8Array(3)
```

- Generally, we prefer to work with standard JS arrays and convert to typed arrays only when we need to send data to the GPU with the `flatten` function in `MV.js`

Notes

- Finally we must connect variable in program/application with variable in shader
 - Need name, type, location in buffer

JS File (cont)

application variable

```
// Associate our shader variables with our data  
buffer
```

```
var vPosition = gl.getAttributeLocation( program,  
    "vPosition" );
```

```
gl.vertexAttribPointer( vPosition, 2, gl.FLOAT,  
    false, 0, 0 );
```

```
gl.enableVertexAttribArray( vPosition );
```

```
render();
```

```
};
```

shader quantifier

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

You can certainly use
different names for
these two!



Five Steps to Use Buffer Objects

- Create a buffer object (`gl.createBuffer()`)
- Bind a buffer object to a target (`gl.bindBuffer()`)
- Write data into a buffer object (`gl.bufferData()`)
- Assign the buffer object to an attribute variable (`gl.vertexAttribPointer()`)
- Enable the assignment to an attribute variable (`gl.enableVertexAttribArray()`)



Five Steps for WebGL Setup

- 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)

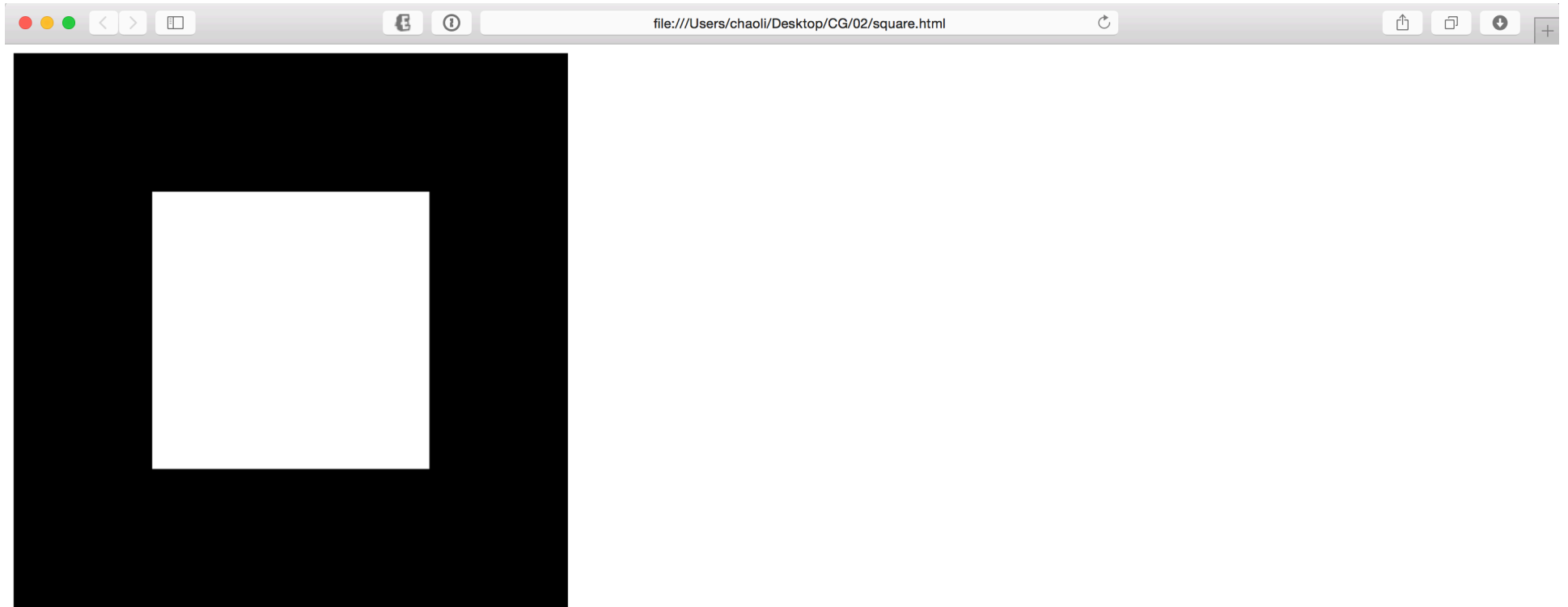


Exercise

- Download `triangle.html` and `triangle.js` to your computer and run them from there
- Edit the two files to change the color and display more than one triangle
- Bring up the Firebug debugging environment for examination

Draw a Square

Draw a Square



square.html

```
<!DOCTYPE html>
<html>
<head>

// vertex shader
<script id="vertex-shader" type="x-shader/x-vertex">
attribute vec4 vPosition;
void main()
{
    gl_Position = vPosition;
}
</script>

// fragment shader
<script id="fragment-shader" type="x-shader/x-fragment">
precision mediump float;
void main()
{
    gl_FragColor = vec4( 1.0, 1.0, 1.0, 1.0 );
}
</script>
```

square.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="square.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>
```

square.js

```
"use strict";
var gl;

window.onload = function init(){
    var canvas = document.getElementById("gl-canvas");
    gl = WebGLUtils.setupWebGL( canvas );
    if ( !gl ) { alert("WebGL isn't available"); }

    // Four vertices
    var vertices = [
        vec2( -0.5, -0.5 ),
        vec2( -0.5,  0.5 ),
        vec2(  0.5,  0.5 ),
        vec2(  0.5, -0.5 )
    ];
```

square.js (cont)

```
// Configure WebGL
gl.viewport( 0, 0, canvas.width, canvas.height );
gl.clearColor( 0.0, 0.0, 0.0, 1.0 );

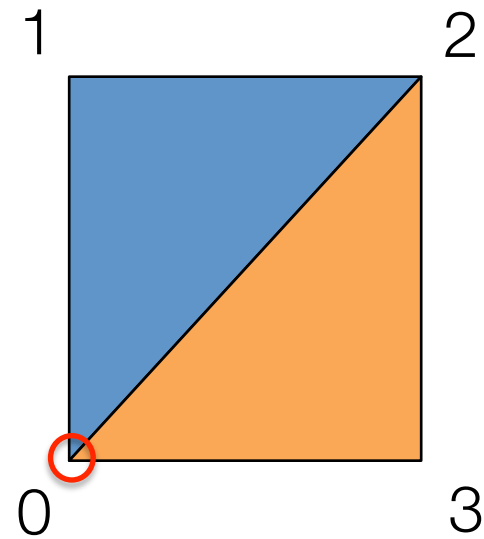
// Load shaders and initialize attribute buffers
var program = initShaders( gl, "vertex-shader", "fragment-shader" );
gl.useProgram( program );

// 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 );

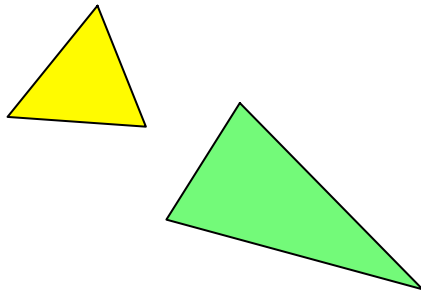
// Associate out shader variables with our data buffer
var vPosition = gl.getAttribLocation( program, "vPosition" );
gl.vertexAttribPointer( vPosition, 2, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vPosition );
```

square.js (cont)

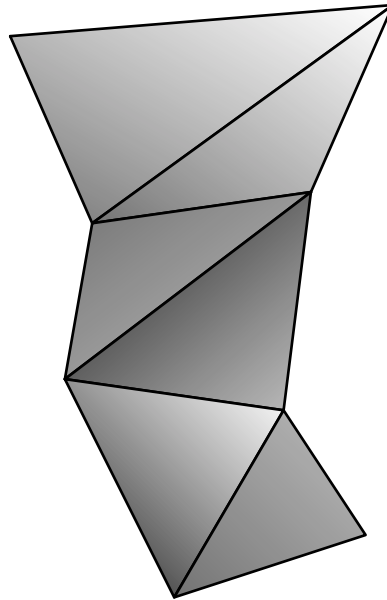
```
    render();  
};  
  
function render() {  
    gl.clear( gl.COLOR_BUFFER_BIT );  
    gl.drawArrays( gl.TRIANGLE_FAN, 0, 4 );  
}
```



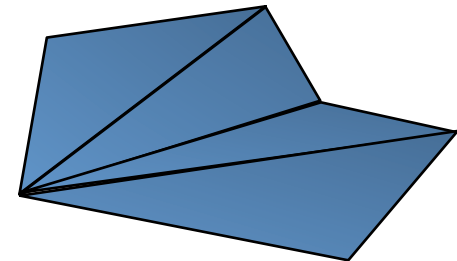
Triangles, Fans or Strips



GL_TRIANGLES



GL_TRIANGLE_STRIP

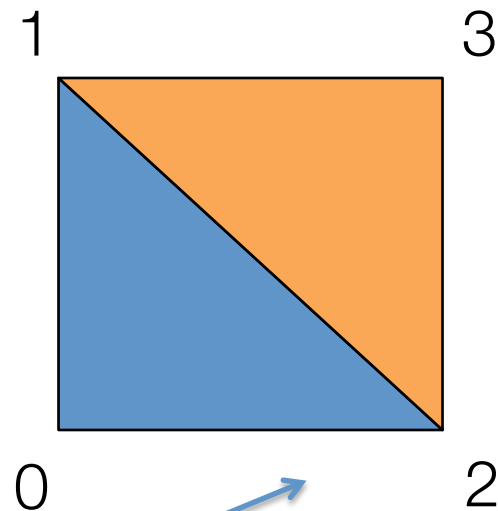
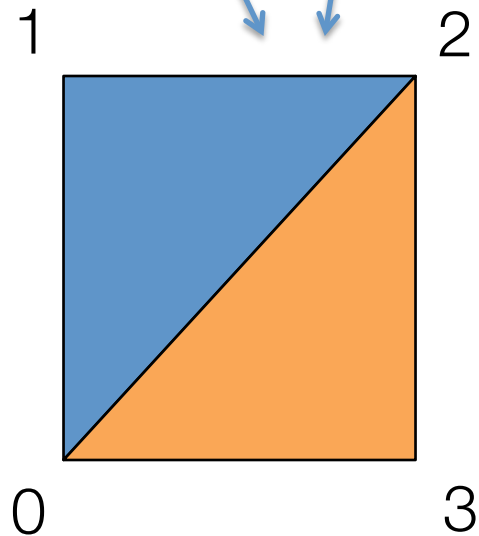


GL_TRIANGLE_FAN

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, 2, 3
```

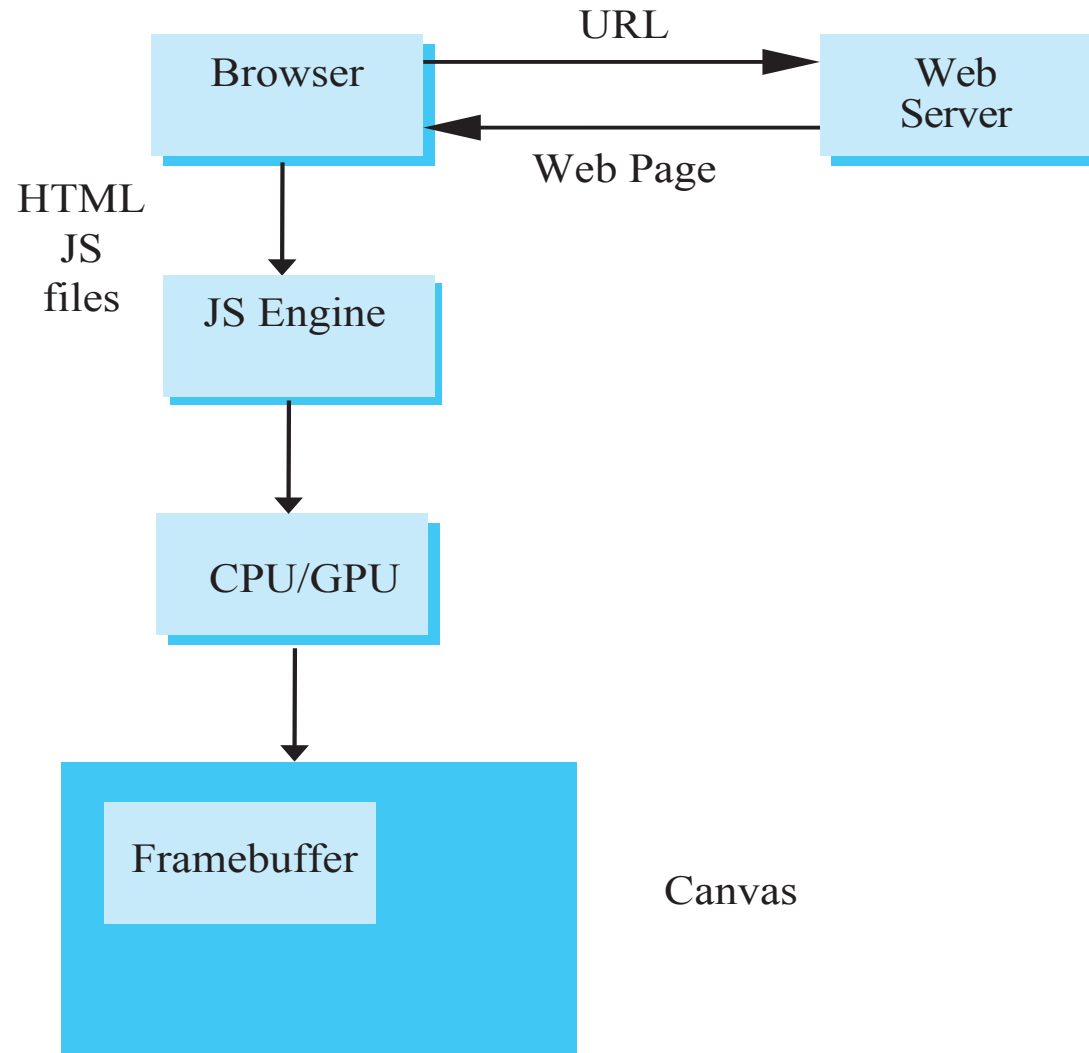



Exercise

- Download `square.html` and `square.js` to your computer and run them from there
- Edit the two files to draw triangles or triangle strip

WebGL and GLSL

Execution in Browser



Program Execution

- WebGL runs within the browser
 - Complex interaction among the operating system, the window system, the browser and your code (HTML and JS)
- Simple model
 - Start with HTML file
 - Files read in asynchronously
 - Start with onload function
 - Event driven input

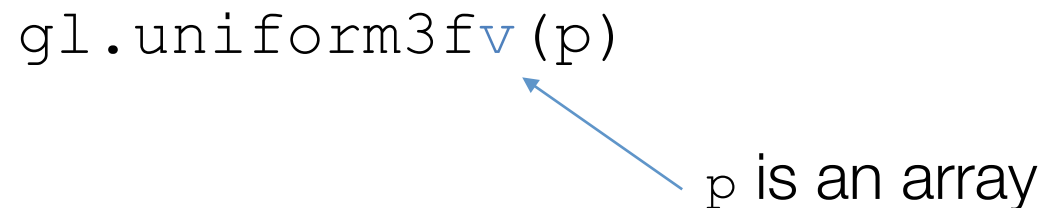
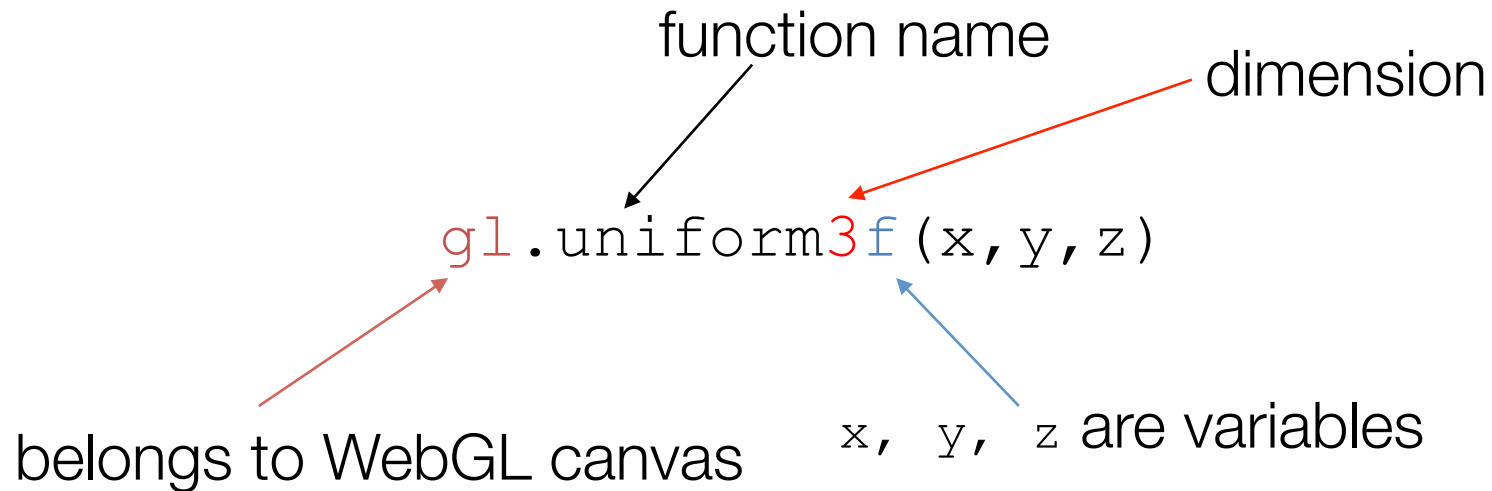
Event Loop

- 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` // single-precision float
 - `gl.uniform2i` // integer
 - `gl.uniform3dv` // double-precision float as vector or array
- 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

```
glClear( GL_COLOR_BUFFER_BIT );
```
 - WebGL

```
gl.clear( gl.COLOR_BUFFER_BIT );
```


GLSL

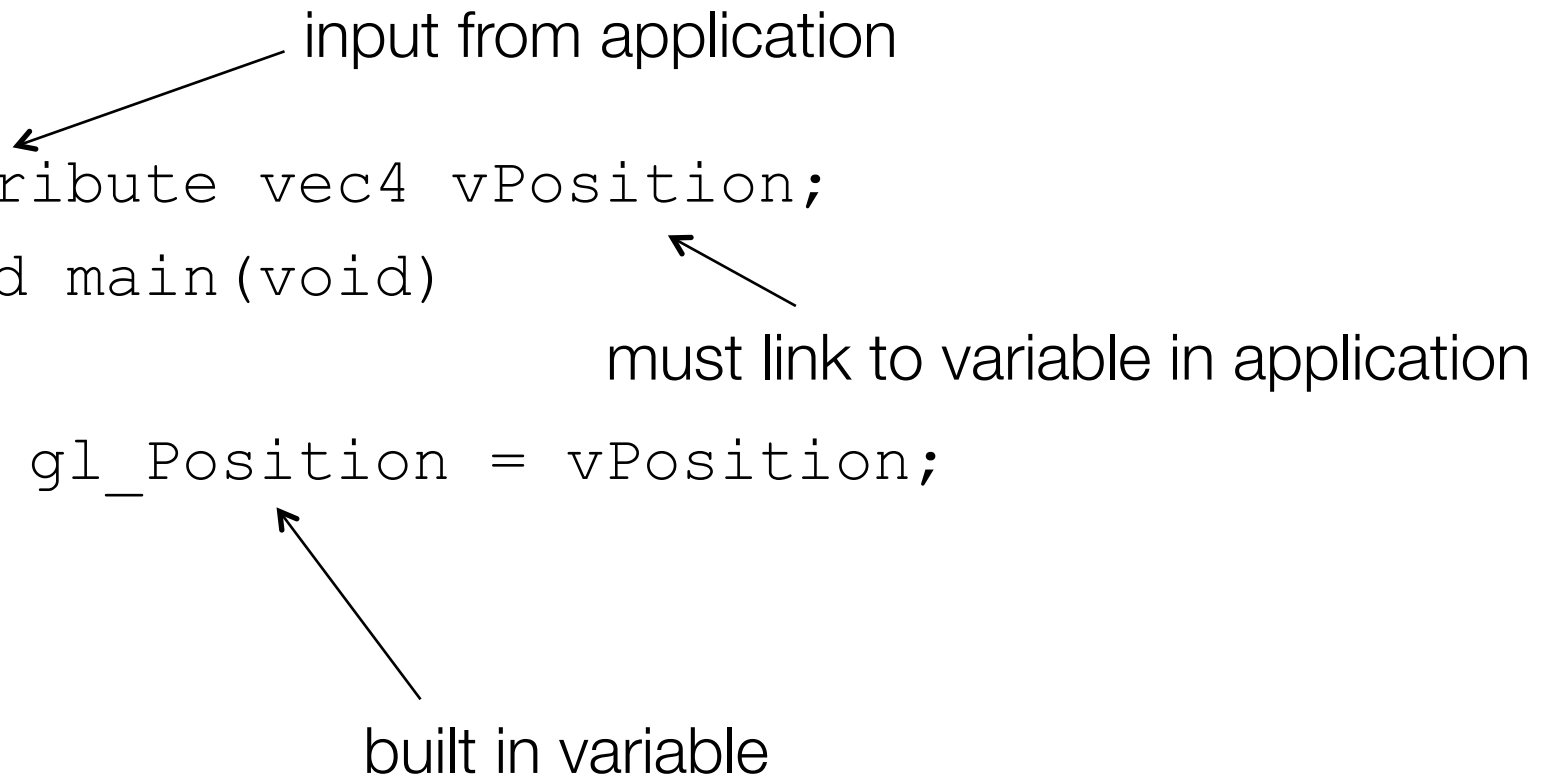
- OpenGL Shading Language
- C-like with
 - Matrix and vector types (2, 3, 4 dimensional)
 - Overloaded operators
 - C++ like constructors
- Similar to Nvidia's Cg and Microsoft HLSL
- Code sent to shaders as source code

WebGL 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
- WebGL functions compile, link, and get information to shaders

Shaders

Simple Vertex Shader



input from application

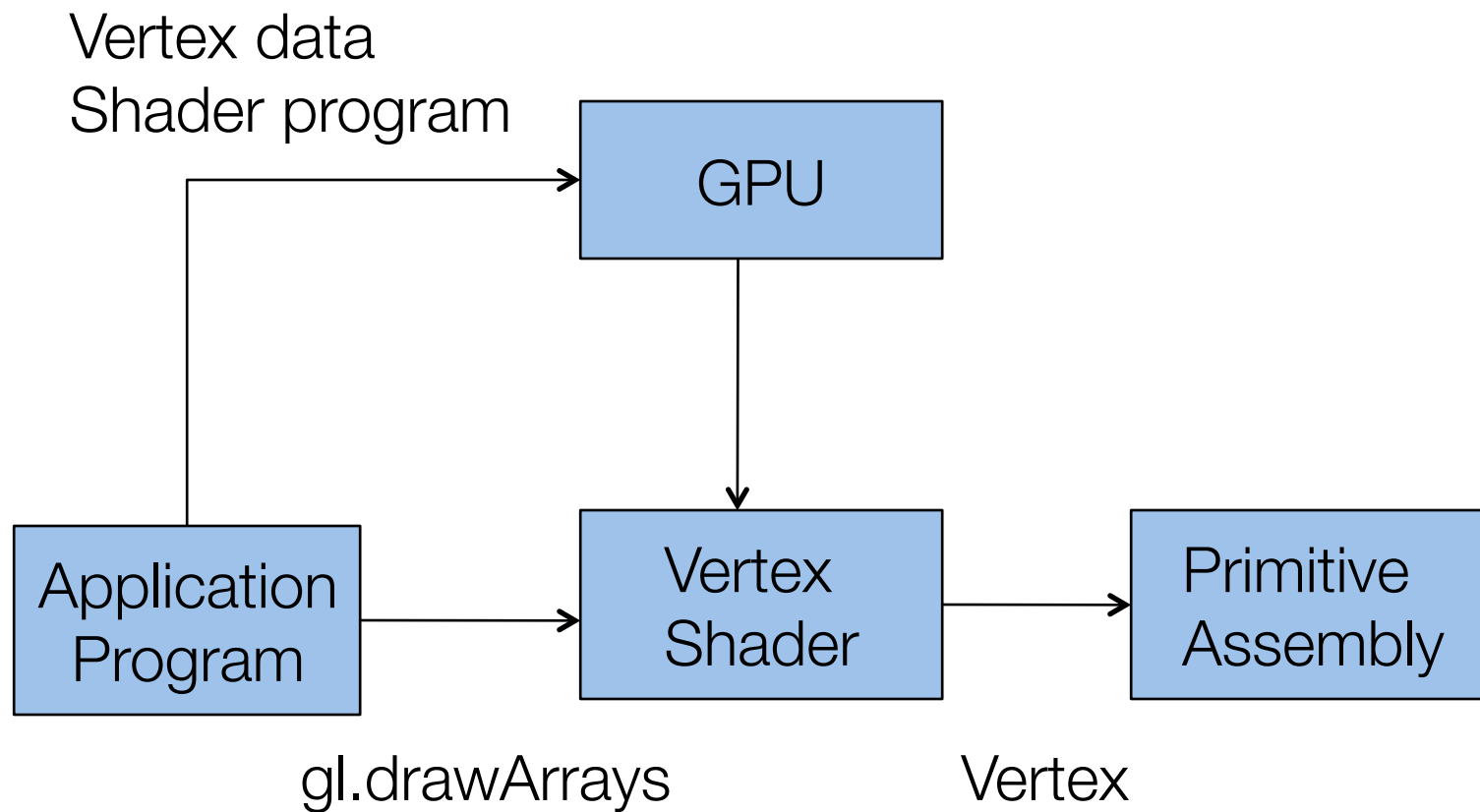
```
attribute vec4 vPosition;  
void main(void)  
{  
    gl_Position = vPosition;  
}
```

must link to variable in application

built in variable

The diagram illustrates the components of a simple vertex shader. It shows a code snippet with three annotations: an arrow pointing to 'vPosition' in the attribute declaration labeled 'input from application', an arrow pointing to 'vPosition' in the assignment statement labeled 'must link to variable in application', and an arrow pointing to 'gl_Position' in the assignment statement labeled 'built in variable'.

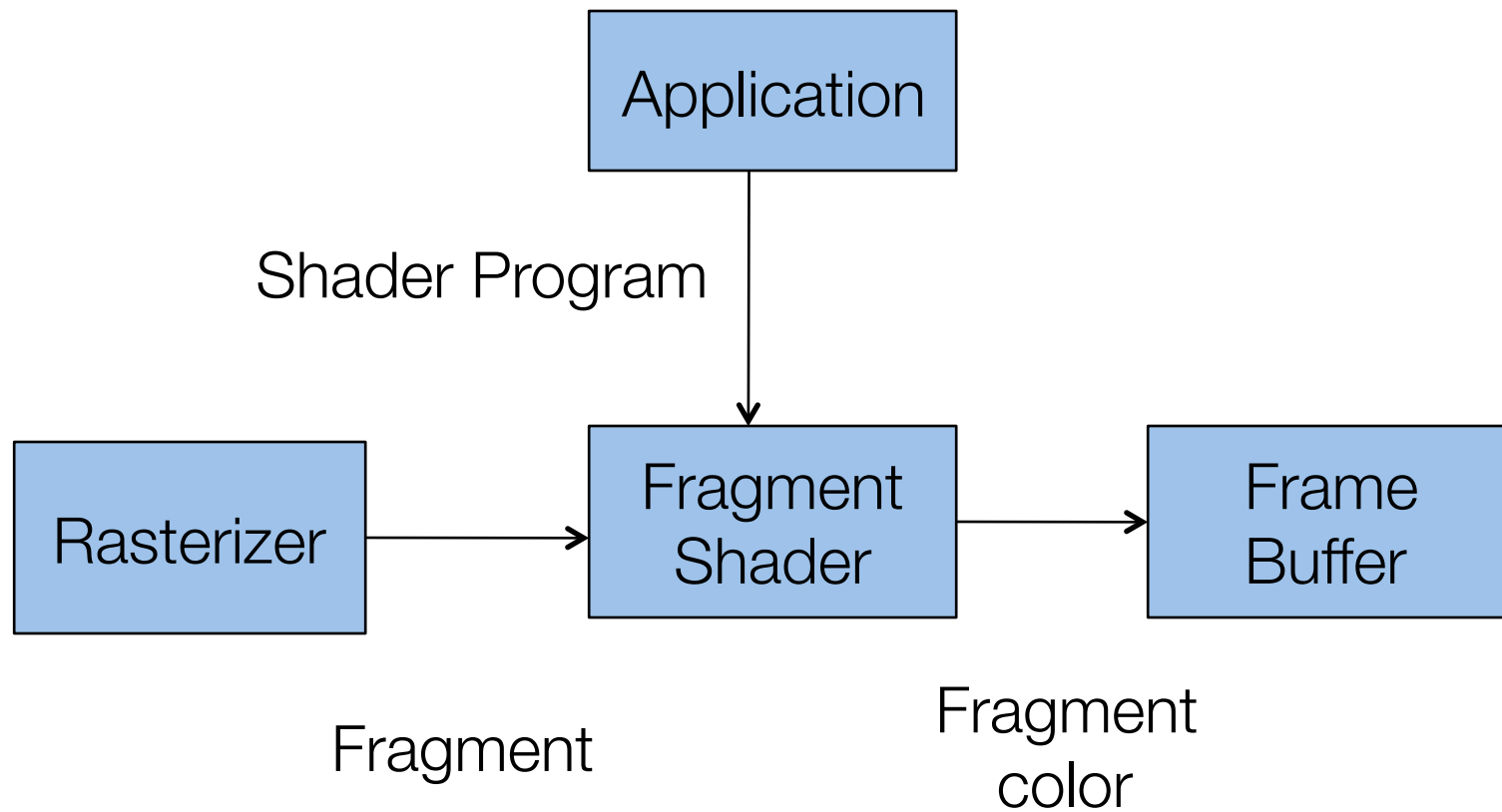
Execution Model



Simple Fragment Program

```
precision mediump float;  
void main(void)  
{  
    gl_FragColor = vec4( 1.0,0.0,0.0,1.0 );  
}
```

Execution Model



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 (e.g., a triangle)
 - Once per vertex
 - Once per fragment
 - At any time in the application
- Vertex attributes are interpolated by the rasterizer into fragment attributes

Attribute Qualifier

- Attribute-qualified 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)
 - `attribute float temperature`
 - `attribute vec3 velocity`

Uniform Qualifier

- 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
 - A bounding box of a primitive
 - Transformation matrices
 - Color of a triangle

Varying Qualifier

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

```
varying vec4 color;
```

Our Naming Convention

- Attributes passed to vertex shader have names beginning with `v` (`vPosition`, `vColor`) in both the application and the shader
 - Note that these are different entities with the same name
 - ```
var vColor = gl.getAttribLocation(program,
 "vColor");
```
- Varying 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

# 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 // typically used for positions
  - r, g, b, a // typically used for colors
  - s, t, p, q // typically used for textures
  - 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, 3.0, 4.0);
b = a.yxzw;
```





## Exercise

- 1. Modify `triangle.html` and `triangle.js` to passing in a uniform color to fragment shader
- 2. Modify `triangle.html` and `triangle.js` to passing three different colors (red, green, and blue) from application to vertex and fragment shaders (the colors within the triangle will be linearly interpolated by the graphics hardware)

# Sending a Uniform Variable (each **primitive** has a different color)

```
// in application
var colour = vec4(1.0, 0.0, 0.0, 1.0);
var colorLoc = glGetUniformLocation(program,
"color");
gl.uniform4fv(colorLoc, colour);

// in fragment shader (similar in vertex shader)
uniform vec4 color;
void main()
{
 gl_FragColor = color;
}
```

See example:  
[triangle-uni-fcolor.html](http://triangle-uni-fcolor.html)

For Exercise 1

# Useful Tips for Exercise 2

- We could store vertex positions and colors into separate arrays
  - Two separate VBOs
  - Easy to deal with and set parameters in `gl.vertexAttribPointer()`
  - However, for the position and color of the same vertex, they are separated in different VBOs

See example:  
[triangle-var-vcolor.html](http://triangle-var-vcolor.html)

For Exercise 2

# Sending Colors from Application (each **vertex** has a different color)

```
var colors = [vec3(1.0, 0.0, 0.0),
 vec3(0.0, 1.0, 0.0),
 vec3(0.0, 0.0, 1.0)
];
```

```
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, 3, gl.FLOAT, false, 0,
0); // RGB, three components
gl.enableVertexAttribArray(vColor);
```

See example:  
<triangle-var-vcolor.html>

For Exercise 2

# Example: Vertex Shader

```
attribute vec4 vColor;
varying vec4 fColor;
void main()
{
 gl_Position = vPosition;
 fColor = vColor;
}
```

See example:  
[triangle-var-vcolor.html](http://triangle-var-vcolor.html)

For Exercise 2

# Corresponding Fragment Shader

```
precision mediump float;
varying vec4 fColor;
void main()
{
 gl_FragColor = fColor;
}
```

See example:  
[triangle-var-vcolor.html](http://triangle-var-vcolor.html)

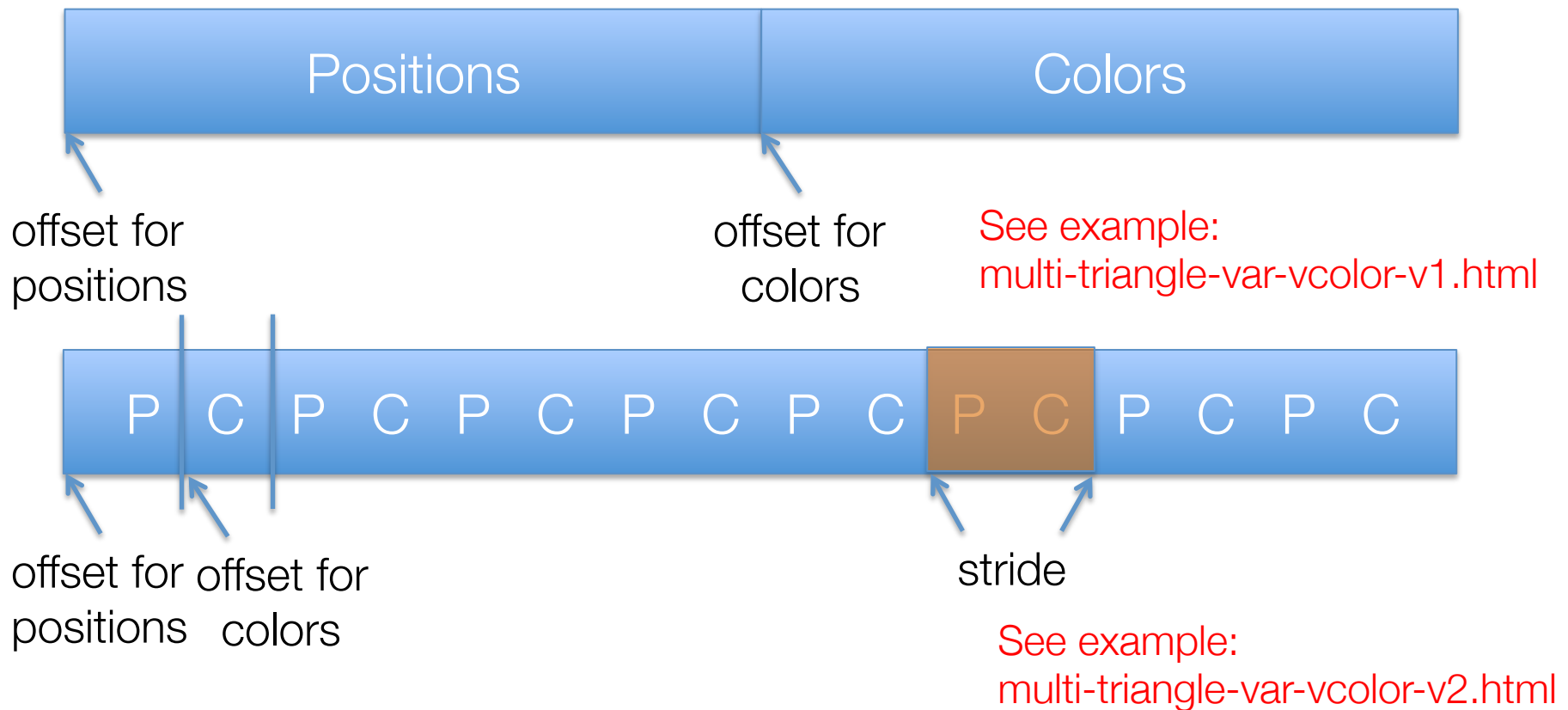
For Exercise 2

# Let Us Draw Two Triangles

- We could store vertex positions and colors into separate arrays
  - Two separate VBOs
  - Easy to deal with and set parameters in  
`gl.vertexAttribPointer()`
- Or, we could store vertex positions and colors into the same array
  - One single VBO
  - Pay special attention to parameter settings in  
`gl.vertexAttribPointer()`

# Useful Tips (draw two triangles)

- `gl.vertexAttribPointer()`
- 5<sup>th</sup> parameter: stride, if continuous, it is 0
- 6<sup>th</sup> parameter: offset, where the first element appears
- Both are specified in terms of **the number of bytes!**





Linking Shaders  
(handled by `initShaders.js`)

# 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);
```

# Reading a Shader

- Shaders are added to the program object and are 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 points
  - 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

varying vec4 fcolor;
void main(void)
{
 gl_FragColor = fcolor;
}
```