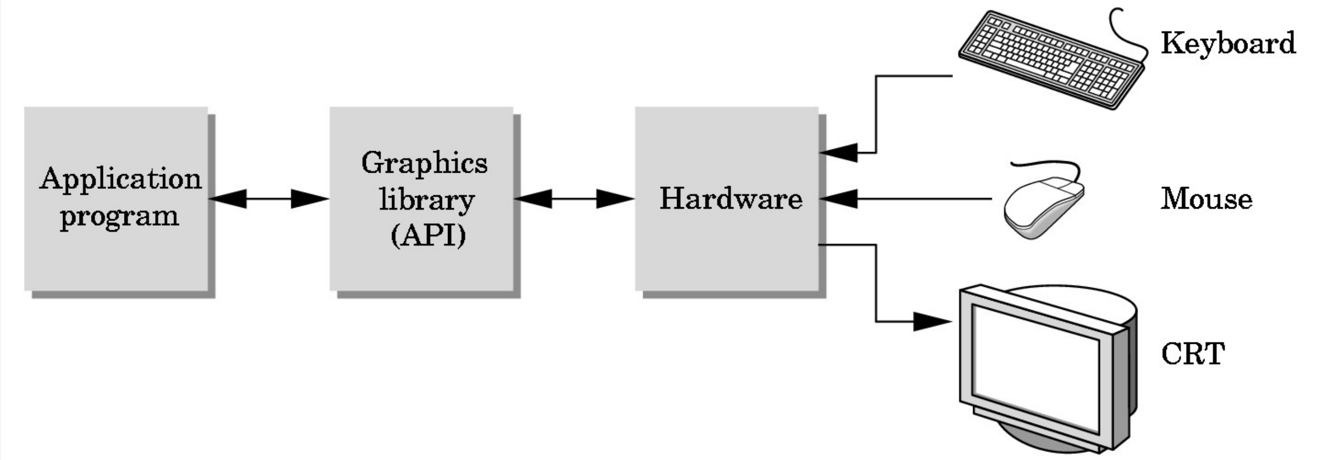
Graphics

Jungchan Cho
Dept. of Software, Gachon University

Review of Graphics Pipeline

The Programmer's Interface

- Programmer sees the graphics system through a software interface: the Application Programmer Interface (API)

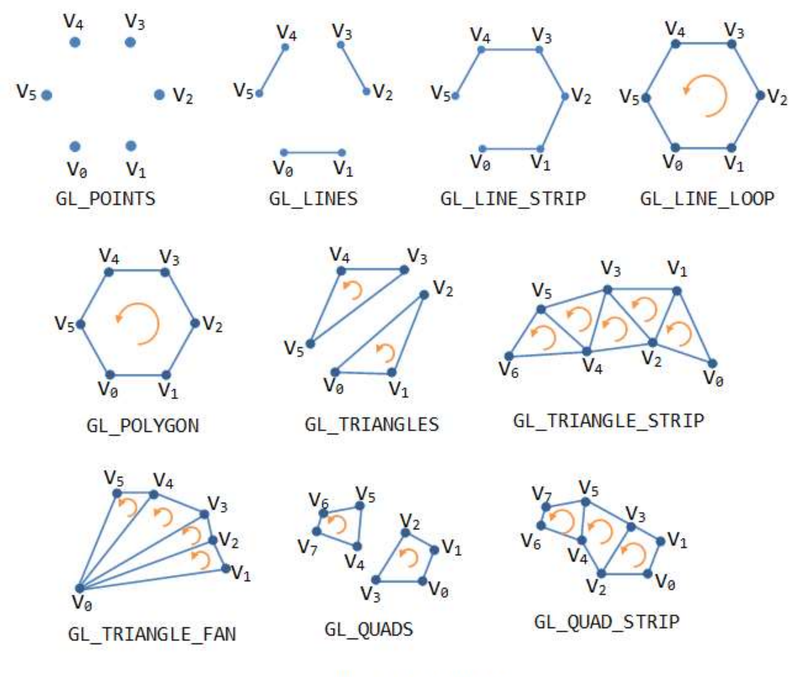


API Contents

- Functions that specify what we need to form an image
  - Objects
  - Viewer
  - Light Source(s)
  - Materials
- Other information
  - Input from devices such as mouse and keyboard
  - Capabilities of system

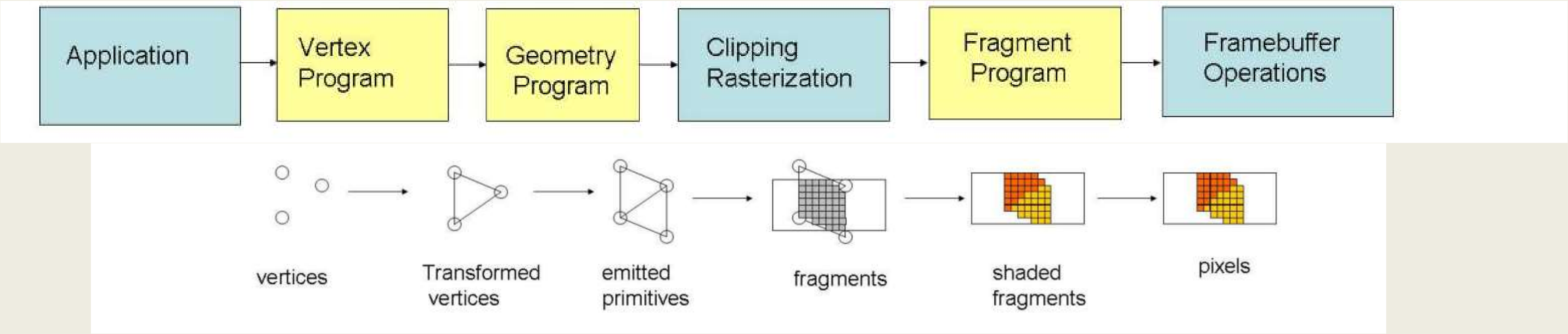
Object Specification

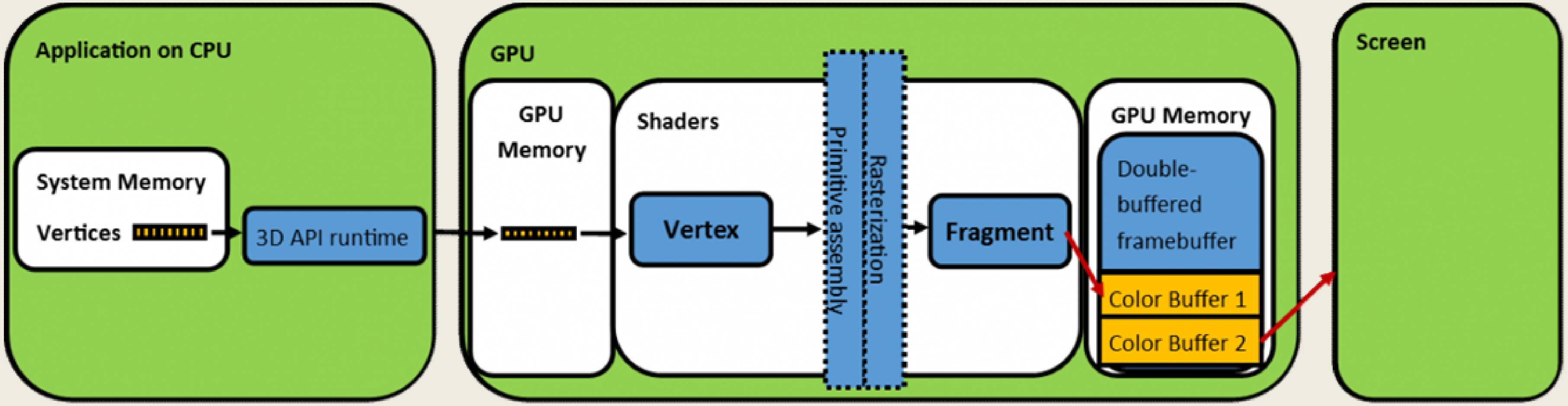
- Most APIs support a limited set of primitives including
  - Points (0D object)
  - Line segments (1D objects)
  - Polygons (2D objects)
  - Some curves and surfaces
    - Quadrics
    - Parametric polynomials



OpenGL Primitives

- All are defined through locations in space or *vertices*

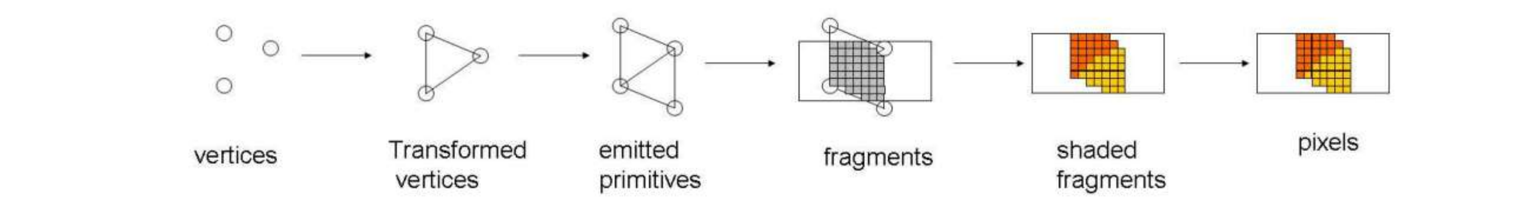




- Vetter et al.," Non-rigid multi-modal registration on the GPU. In Medical Imaging 2007: Image Processing," *International Society for Optics and Photonics*, vol. 6512, pp. 651228, Mar. 2007.
- https://www.gamedev.net/articles/programming/graphics/introduction-to-the-graphics-pipeline-r3344/

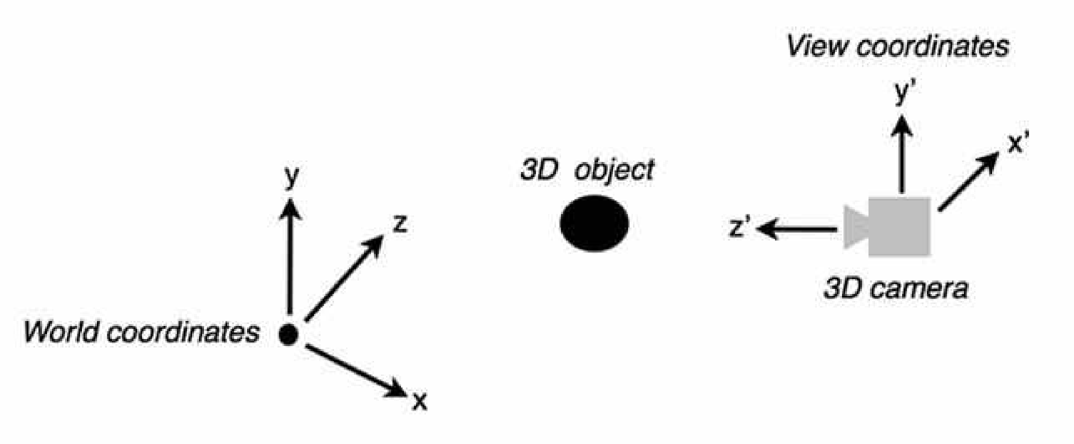
Graphics architectures

- Most graphics architecture uses *pipelining*
  - Similar to an assembly line in a car plant
    - Cons: significant delay between starts and ends
    - Pros: throughput (number of produced cars in a given time) is much higher than if single team builds each car
- Graphics pipeline
  - Vertex processing, clipping and primitive assembler, rasterizer and fragment processing
  - All steps can be implemented in hardware



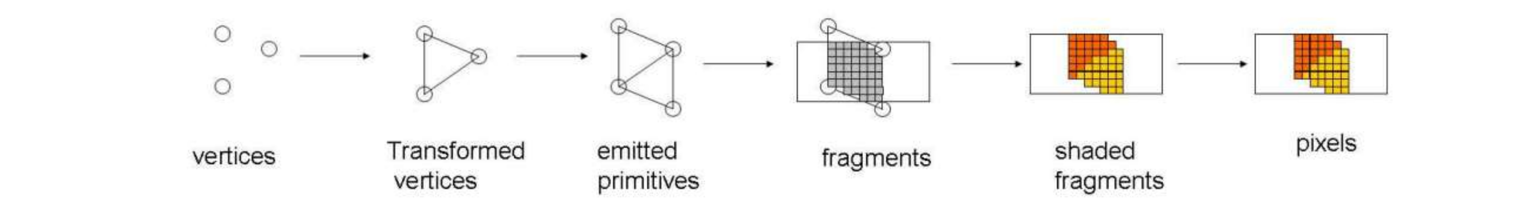
Graphics architectures

- Vertex processing
  - Each vertex is processed independently
  - Two major functions of vertex processing
    - Coordinate transformations
      - Converts *object coordinates* into *camera coordinates*
      - Converts *camera coordinates* into *screen coordinates*
      - Typically implemented by a series of matrix operations
    - Compute a color for each vertex



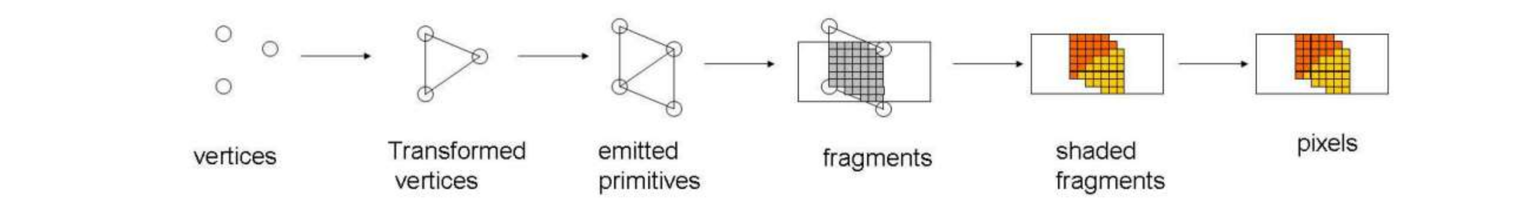
Graphics architectures

- Clipping and primitive assembly
  - Identify parts of objects which should be invisible due to clipping volume (3D area for clipping)
  - Must be done on a primitive by primitive basis
    - Mainly due to the computational efficiency
    - We must assemble sets of vertices into primitives such as line seg ments or polygon before clipping
- Rasterization
  - Generate fragments from clipped primitives
    - A fragment is a pixel carrying color, location, depth, etc.
    - E.g. generate pixels inside triangle for filled triangle
    - Similar to “vector” to “bitmap” conversion



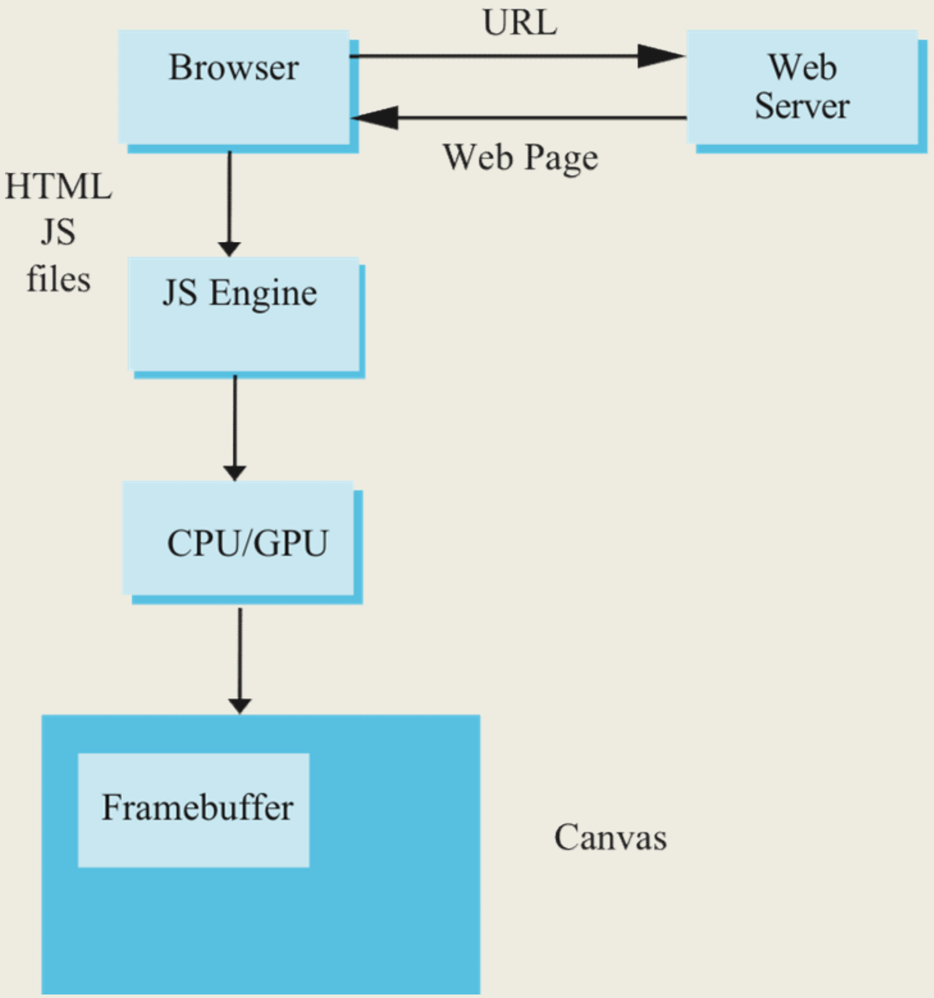
Graphics architectures

- Fragment processing
  - Update the pixels in the frame buffer from fragments
  - Several things should be considered in this stage
    - Visibility of fragment is checked from depth information
    - Color of a fragment may be altered by texture mapping or bump mapping
    - Effects (e.g. translucent, blending) are considered
- Programmable pipeline
  - Typically pipeline architectures had a fixed functionality
    - E.g. support of only one lighting model
  - Recent graphic architecture allow application to modify ve rtex processor and fragment processor



WebGL

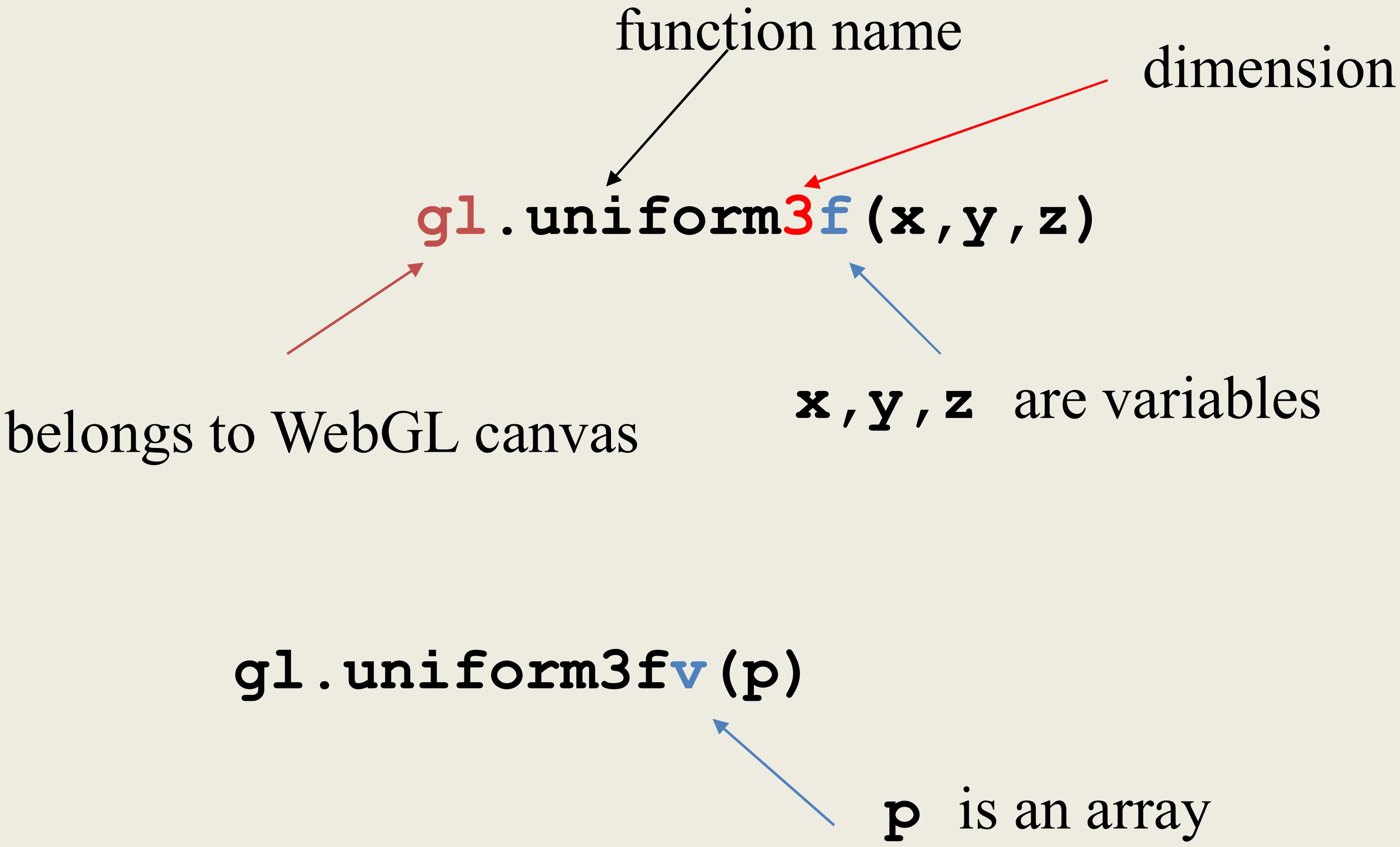
Execution in Browser



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



function name

dimension

```
gl.uniform3f(x,y,z)
```

belongs to WebGL canvas

`x,y,z` are variables

```
gl.uniform3fv(p)
```

`p` is an array

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**

Programming with WebGL: Complete Programs

Triangle

- Example: Draw a triangle
  - Each application consists of (at least) two files
  - HTML file and a JavaScript file
- HTML
  - describes page
  - includes utilities
  - includes shaders
- JavaScript
  - contains the graphics

Objectives

- Build a complete first program
  - Introduce shaders
  - Introduce a standard program structure
- Simple viewing
  - Two-dimensional viewing as a special case of three-dimensional viewing
- Initialization steps and program structure

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

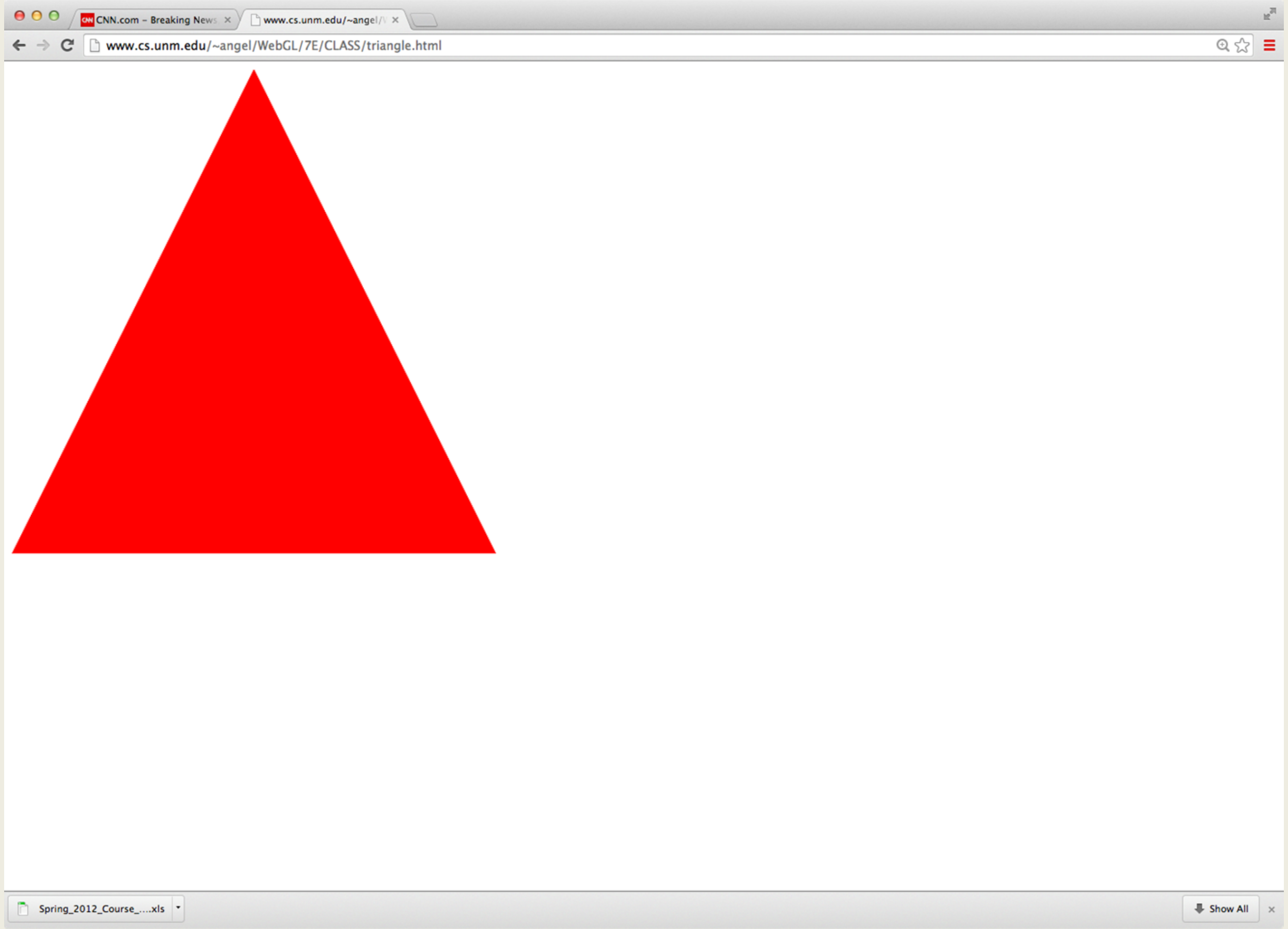
WebGL

- 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)

Coding in WebGL

- Can run WebGL on any recent browser
  - Chrome
  - Firefox
  - Safari
  - IE
- Code written in JavaScript
- JS runs within browser
  - Use local resources

Example: triangle.html


CNN.com – Breaking News
www.cs.unm.edu/~angel/
www.cs.unm.edu/~angel/WebGL/7E/CLASS/triangle.html
Spring_2012_Course_....xls
Show All

https://www.cs.unm.edu/~angel/WebGL/7E/CLASS/triangle.html

Example Code

A **shader** is a computer program that runs on the graphics processing unit (GPU) and is used to do shading – the production of appropriate levels of light and darkness within a image.

https://slideplayer.com/slide/9489254/

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

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

<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 );
}
</script>
```

Example Code

A **shader** is a computer program that runs on the graphics processing unit (GPU) and is used to do shading – the production of appropriate levels of light and darkness within a image.

https://slideplayer.com/slide/9489254/

- These are trivial pass-through (do nothing) shaders that which set the two required **built-in variables**
  - **gl_Position**
  - **gl_FragColor**
- Note both shaders are full programs
- Note vector type vec2
- Must set **precision** in **fragment** shader

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

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



<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 );
}
</script>
```

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>
```

Canvas defines a rendering space where JavaScript draws images using canvas APIs.

JS File

```
var gl;
var points;

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 )
];
```

The "load" event occurs when a document is fully loaded. Usually we need to wait for this event before we start running our JavaScript code.

JS File

```
var gl;
var points;

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 )
];
```

```
<canvas id="gl-canvas" width="512"
height="512">
```

It gives us a reference to the canvas.
* DOM: Document Object Model

JS File

```
var gl;
var points;

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 )
];
```

gl = canvas.get Context("webgl")

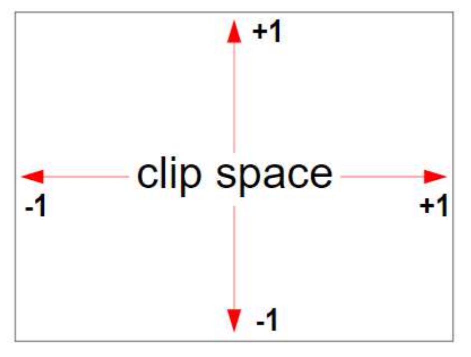
The WebGLRenderingContext interface provides **an interface to the OpenGL ES 2.0 graphics rendering context** for the drawing surface of an HTML <canvas> element.

https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext

JS File

```
var gl;
var points;

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 )
];
```

vertices use vec2 type in MV.js



JS File (cont)

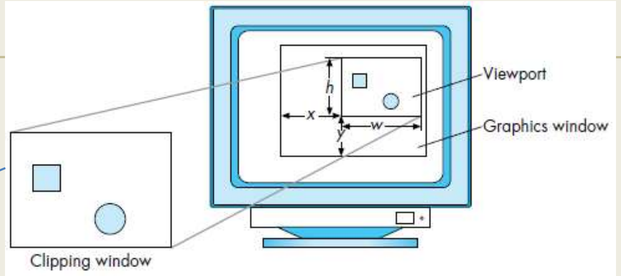


FIGURE 2.37 A mapping to the viewport.

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

JS File (cont)

The function call “gl.ClearColor(1.0, 1.0, 1.0, 1.0);” specifies an RGB-color clearing color that is white, i.e., RGBA.

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

JS File (cont)

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

initShaders used to load, compile and link shaders to form a program object

JS File (cont)

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

    gl.bindBuffer( gl.ARRAY_BUFFER, bufferId );
    gl.bufferData( gl.ARRAY_BUFFER, flatten(vertices), gl.STATIC_DRAW );
```

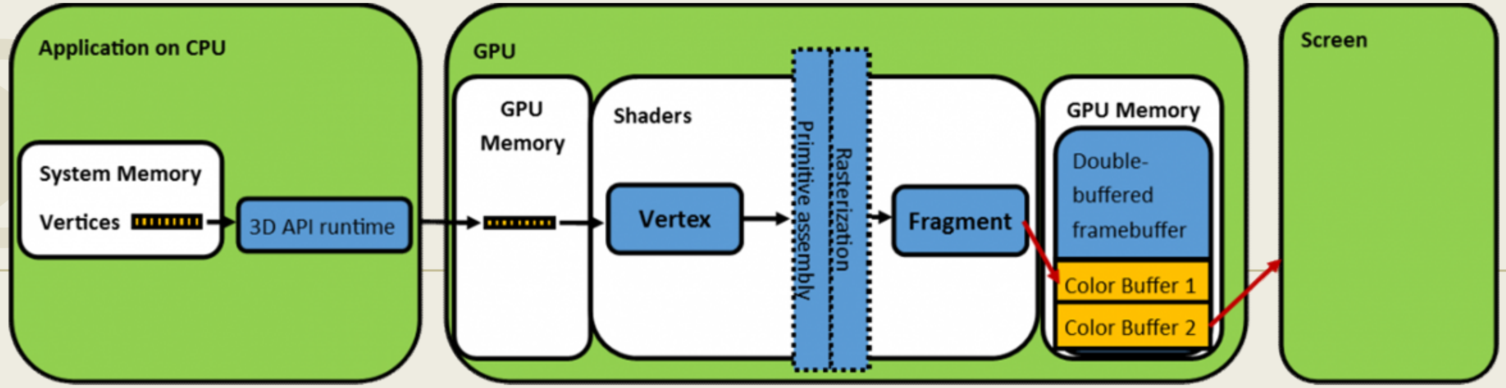
initShaders used to load, compile and link shaders to form a program object

Instruct WebGL to run the Shader Program.

JS File (cont)

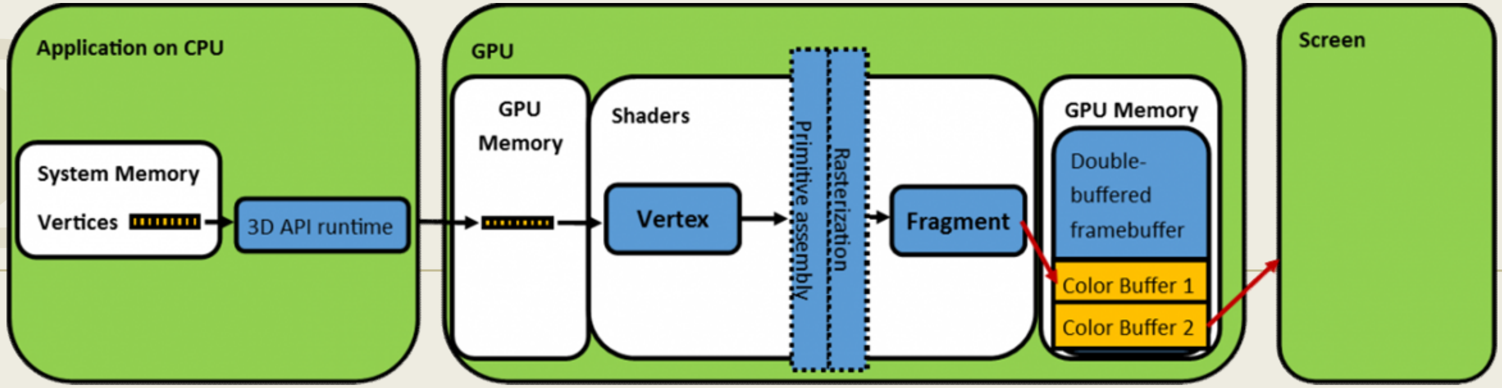
```
//  Configure WebGL
//
    gl.viewport( 0, 0, canvas.width, canvas.height );
    gl.clearColor( 1.0, 1.0, 1.0, 1.0 );

//  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 );
```

Load data onto GPU by creating a **vertex buffer object** on the GPU



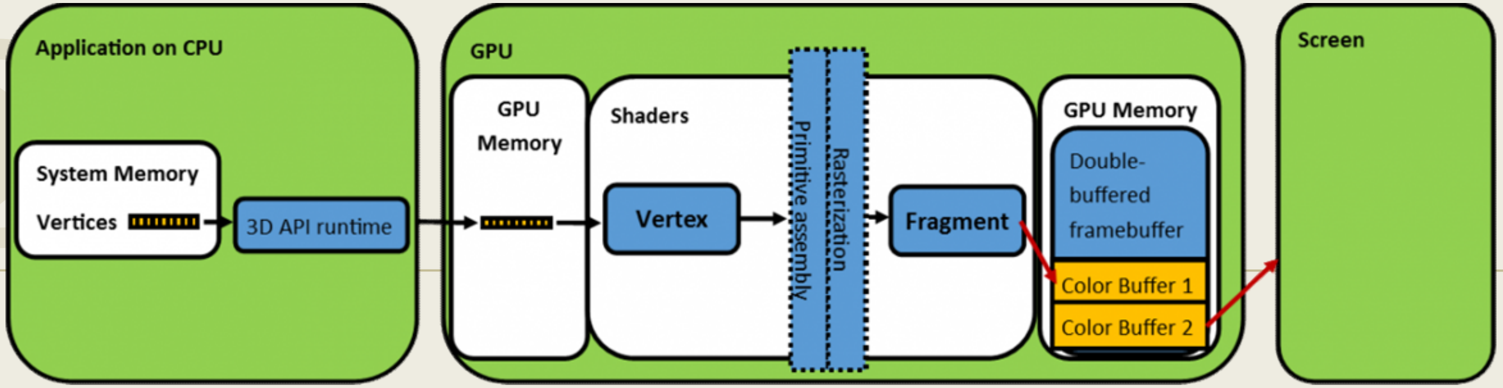
```
//  Configure WebGL
//
    gl.viewport( 0, 0, canvas.width, canvas.height );
    gl.clearColor( 1.0, 1.0, 1.0, 1.0 );

//  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 );
```

Load data onto GPU by creating a **vertex buffer object** on the GPU



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

gl.ARRAY_BUFFER is a bind point
You can manipulate many resources in WebGL with a global bind point.

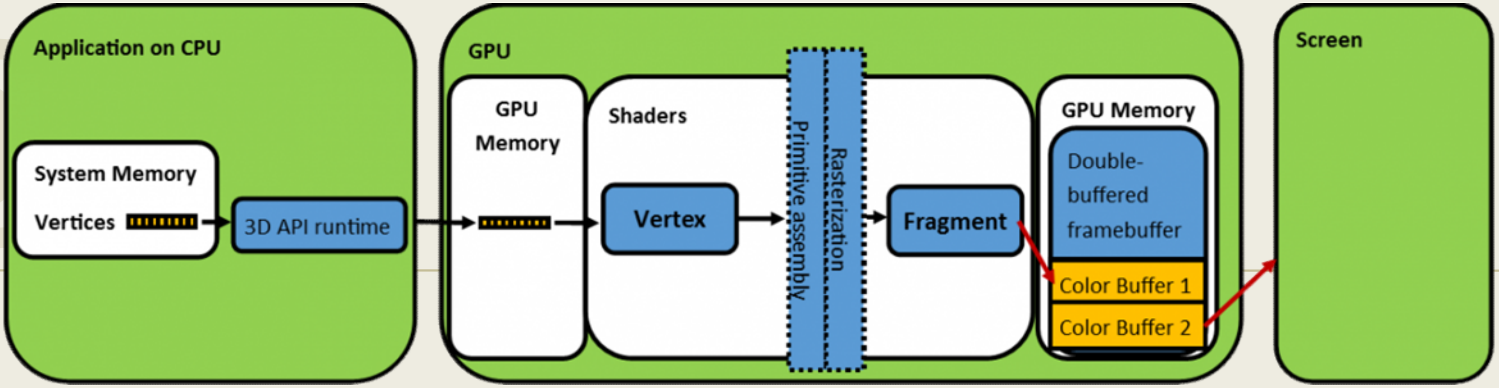


```
//  Configure WebGL
//
    gl.viewport( 0, 0, canvas.width, canvas.height );
    gl.clearColor( 1.0, 1.0, 1.0, 1.0 );

//  Load shaders and initialize attribute buffers

    var program = initShaders( gl, '
    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 );
```

gl.bufferData copies the data to the bufferId on the GPU.

Note use of flatten() to convert JS array to an array of float32's

```
// Three Vertices

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



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

gl.STATIC_DRAW tells WebGL how to use the data. ➔ gl.STATIC_DRAW notifies WebGL that data is unlikely to be changed.

JS File (cont)

- ✓ Finally we must connect variable in program with variable in shader ➔ name, type, location in buffer

```
// 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 );
      render();
};

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

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

JS File (cont)

✓ We need to tell WebGL how to get the data from the buffer we set before and give it to the attribute of the shader.

```
// 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 );
     render();
};

funct
   gl.
   gl.                                                 ;
}
```

```
gl.vertexAttribPointer(
  positionAttributeLocation,
  size, type, normalize,
  stride, offset
);
```

```
// Three Vertices

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

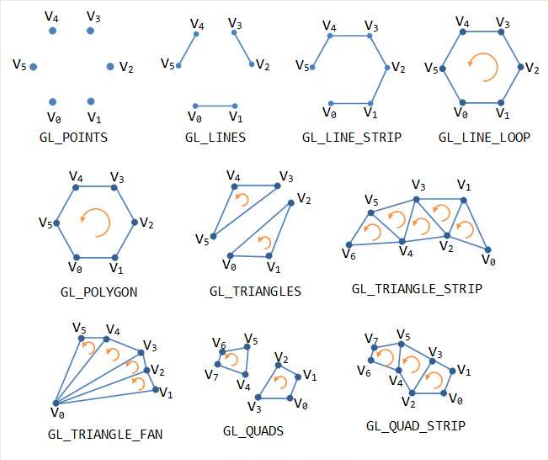
JS File (cont)

- ✓ Must be done on a primitive by primitive basis
  - ✓ Mainly due to the computational efficiency
  - ✓ We must assemble sets of vertices into primitives such as line segments or polygon before clipping

```
// 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 );
    render();
};

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

Number of vertices required to draw a triangle

gl.drawArrays(primitiveType, offset, count);



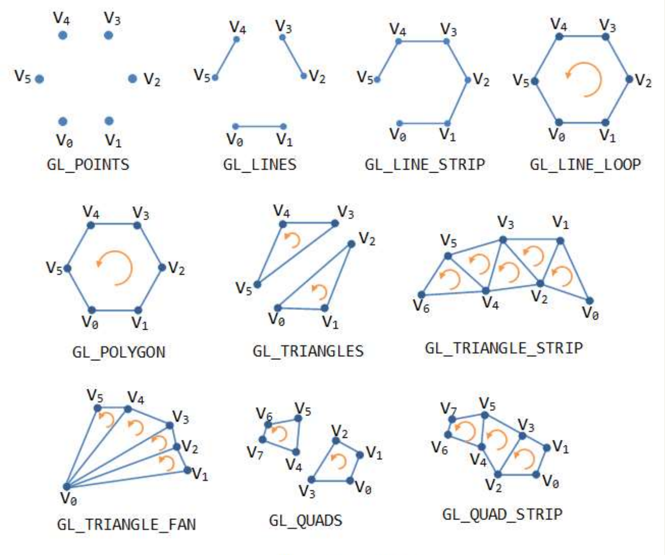
OpenGL Primitives

Coding in WebGL

1. Write an HTML code with a <canvas> tag.
   <canvas> tag provides a drawing space for WebGL.
   Then, write JavaScript a code that creates a canvas reference for GLRendingContext generation.
2. Write codes for the Vertex shader and Fragment shader.
3. Write a WebGL API code that creates shader objects that manage Vertex shaders and Fragment shaders.
   After loading the shader codes on shader objects, compile the objects.
4. Create a program object and attach compiled shaders to the object. You can then link the program object to specify that WebGL use them for rendering.
5. Create a WebGL buffer object and load the Vertex data containing geometric information (such as triangles) into the buffer.
6. Specifies which attributes in the shader should be connected to the buffer and draws geometric information (triangle) on the screen.

Exercise 1

- Run triangle.html
- Load the triangle.html and triangle.js to your computer and run them from there
- **Edit the two files to change the color and display two triangles**



OpenGL Primitives

JavaScript Notes

- JavaScript (JS) is the language of the Web
  - All browsers will execute JS code
  - JavaScript is an interpreted object-oriented language
- References
  - Flanagan, JavaScript: The Definitive Guide, O'Reilly
  - Crockford, JavaScript, The Good Parts, O'Reilly
  - Many Web tutorials

JS Notes

- Is JS slow?
  - JS engines in browsers are getting much faster
  - Not a key issues for graphics since once we get the data to the GPU it doesn’t matter how we got the data there
- JS is a (too) big language
  - We don’t need to use it all
  - Choose parts we want to use
  - Don’t try to make your code look like C or Java

JS Notes

- Very few native types:
  - numbers
  - strings
  - booleans
- Only one numerical type: 32 bit float
  - var x = 1;
  - var x = 1.0; // same
  - potential issue in loops
  - two operators for equality == and ===
- Dynamic typing

Scoping

- Different from other languages
- Function scope
- variables are *hoisted* within a function
  - can use a variable before it is declared
- Note functions are first class objects in JS

JS Arrays

- JS arrays are **objects**
  - inherit methods
  - **var a = [1, 2, 3];**

    is not the same as in C++ or Java
  - a.length // 3
  - a.push(4); // length now 4
  - a.pop(); // 4
  - avoids use of many loops and indexing
  - Problem for WebGL which expects C-style arrays

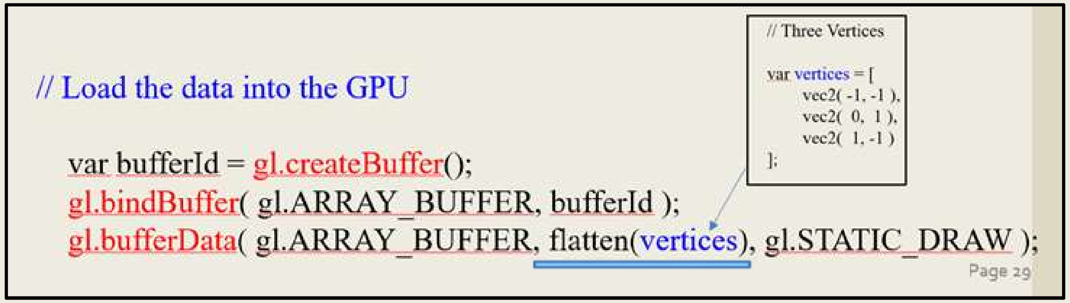
Typed Arrays

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

We already used this ➔



```
// Three Vertices

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

```
// 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 );
```

Page 29

A Minimalist Approach

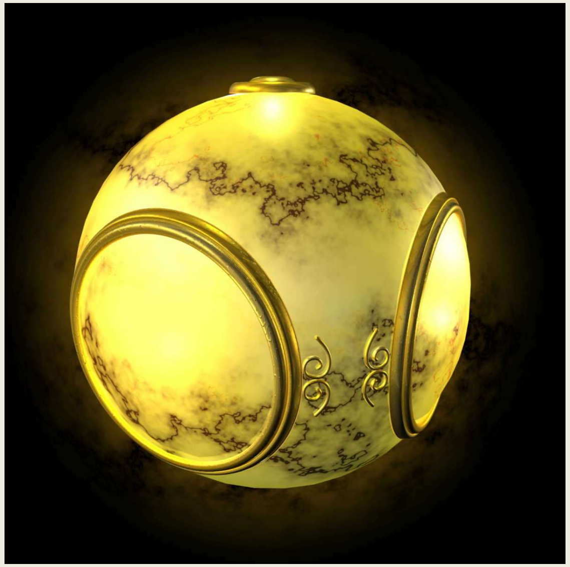
- We will use only core JS and HTML
  - no extras or variants
- No additional packages
  - CSS
  - JQuery
- Focus on graphics
  - examples may lack beauty

Computer Graphics

- *Computer graphics* deals with all aspects of creating images with a computer
  - Hardware
  - Software
  - Applications

Example

- Where did this image come from?



- What hardware/software did we use to produce it?

Preliminary Answer

- **Application**: The object is an artist’s rendition of the sun for an animation to be shown in a domed environment (planetarium)
- **Software**: Maya for modeling and rendering but Maya is built on top of OpenGL
- **Hardware**: PC with graphics card for modeling and rendering