ANATOMY OF YOUR FIRST WEBGL PROGRAM

By Raymond Pang

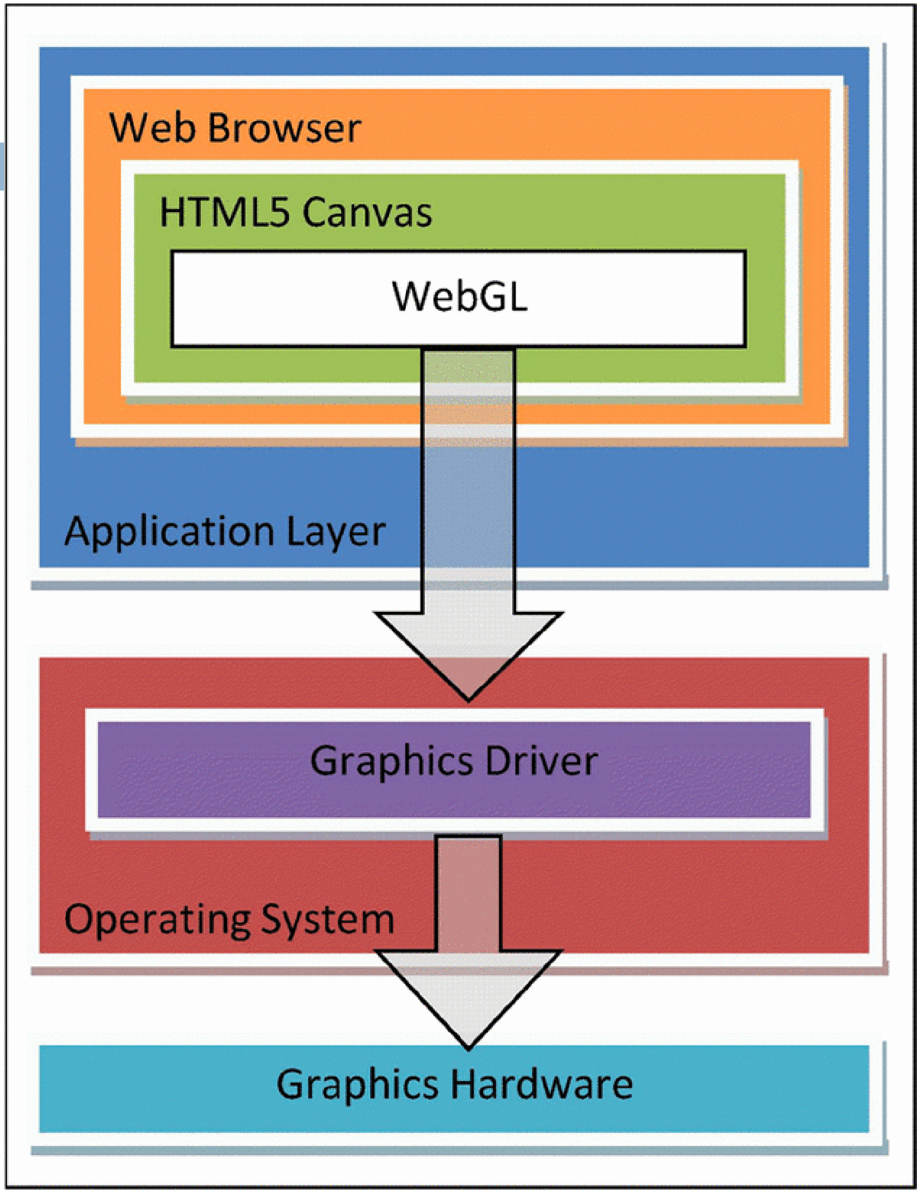
What is WebGL?

- The new standard for 3D Graphics on the Web
- Supported natively by latest browsers including Chrome, Firefox
  - No plugin required

- WebGL is based on the OpenGL ES standard
  - Now version 1.0
- Graphics API
  - Special for real-time rendering

Architecture

- Javascript and HTML Canvas
- Direct access to client's graphics hardware via
  - Browser
  - Graphics Driver



System Requirements

Supported by all major browsers

- Firefox 4.0 or above
- Google Chrome 11 or above
- Safari (OSX 10.6 or above)
  - WebGL is disabled by default but you can switch it on by enabling the **Developer menu and then checking the Enable WebGL option**
- Opera 12 or above

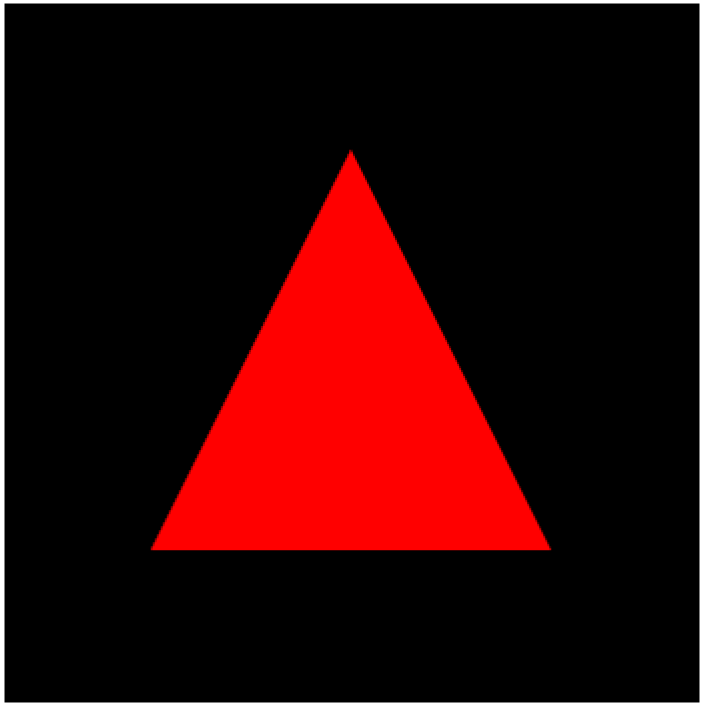
Part I

2D Rendering in WebGL

The Triangle Example

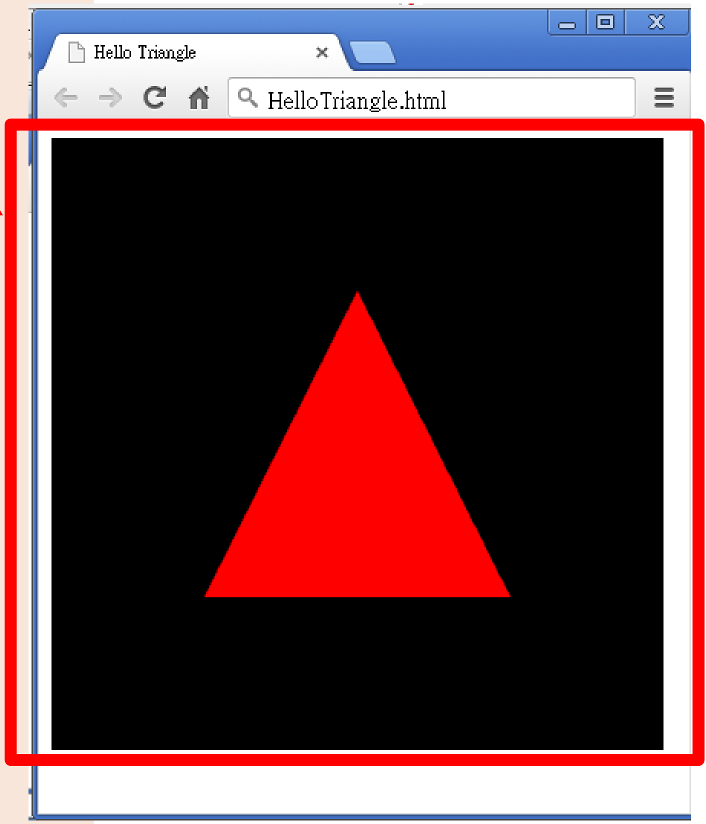
- We start with a 2D example:
  - HelloTriangle.html
  - HelloTriangle.js

  And other utility files from the book:
  - webgl-utils.js
  - webgl-debug.js
  - cuon-utils.js



HTML in Triangle Example

```
<body onload="main()">
    <canvas id="webgl" width="400"
height="400">
    Please use a browser that supports "canvas"
    </canvas>

    <script src="webgl-utils.js"></script>
    <script src="webgl-debug.js"></script>
    <script src="cuon-utils.js"></script>
    <script src="HelloTriangle.js"></script>
  </body>
```



Canvas

Initialize a WebGL Canvas

HelloTriangle.html

```
<body onload="main()">
    <canvas id="webgl" width="400"
height="400">
```

Inside the function main:

```
// Retrieve <canvas> element
  var canvas =
document.getElementById('webgl');

  // Get the rendering context for WebGL
  var gl = getWebGLContext(canvas);
  if (!gl) {
    console.log('Failed to get the rendering context for
WebGL');
    return;
  }
```

WebGL Context

- getWebGLContext
  - Return a WebGL's rendering context object from canvas
  - A handle to apply rendering in WebGL

```
var gl = getWebGLContext(canvas);
```

- call gl.XXXX(…) for render or change the states in the WebGL

Vertice Buffer Object

- Next, we define vertices to draw
- initVertexBuffers is a method defined ourselves

```
// Write the positions of vertices to a vertex shader
var n = initVertexBuffers(gl);
if (n < 0) {
  console.log('Failed to set the positions of the vertices');
  return;
}
```

initVertexBuffers

- Define 3 Vertices
- Store them in the Vertex Buffer Object

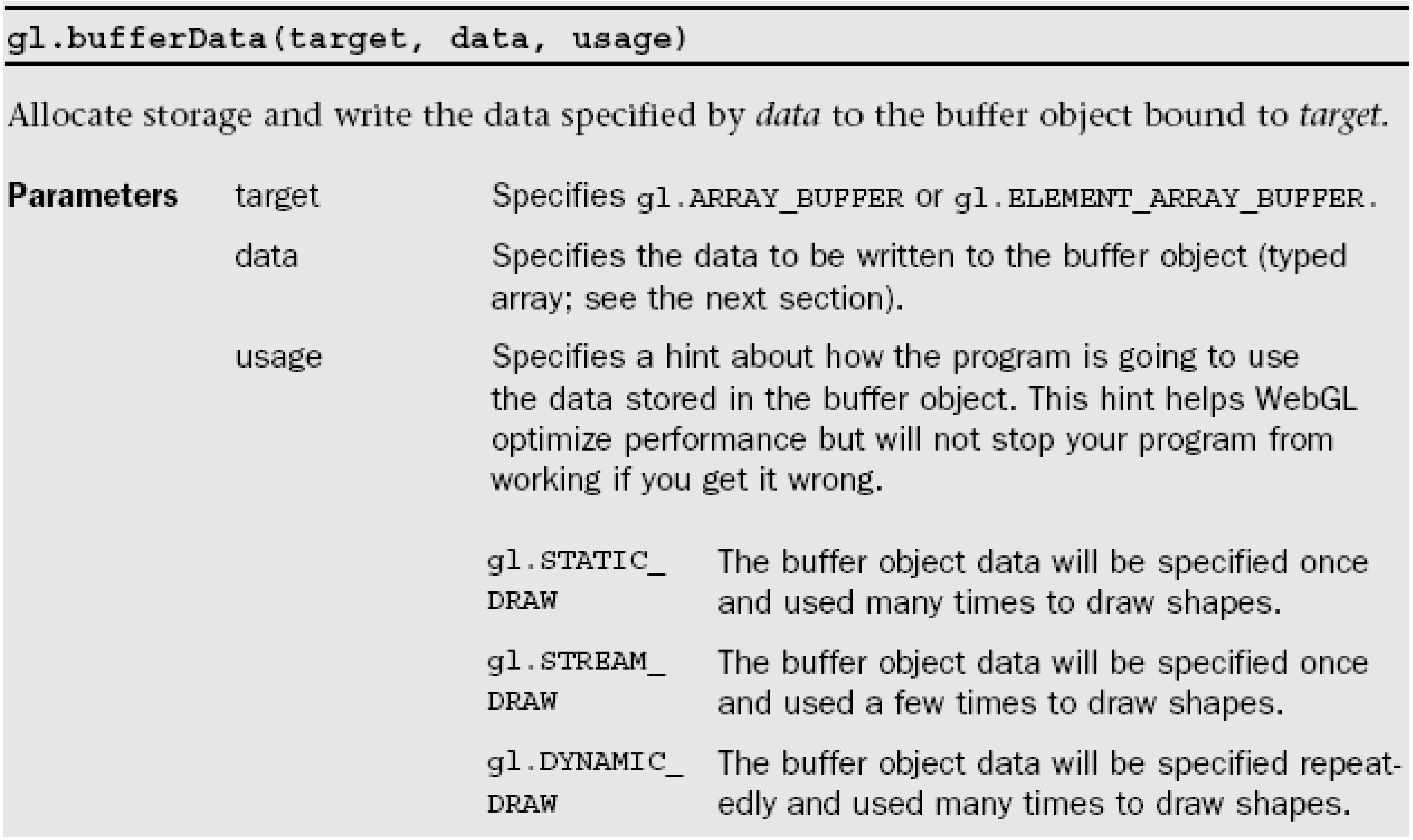
```
function initVertexBuffers(gl) {
  var vertices = new Float32Array([
    0, 0.5,   -0.5, -0.5,   0.5, -0.5
  ]);
  var n = 3; // The number of vertices

  // Create a buffer object
  var vertexBuffer = gl.createBuffer();
  if (!vertexBuffer) {
    console.log('Failed to create the buffer object');
    return -1;
  }
……
```

initVertexBuffers

- Bind our VBO with Array_buffer in the engine
- Write data into VBO
- We need binding before use buffer as OpenGL is a state machine

```
…….
  // Bind the buffer object to target
  gl.bindBuffer(gl.ARRAY_BUFFER,
vertexBuffer);

  // Write data into the buffer object
  gl.bufferData(gl.ARRAY_BUFFER,
vertices, gl.STATIC_DRAW);


  …..
```

gl.bufferData



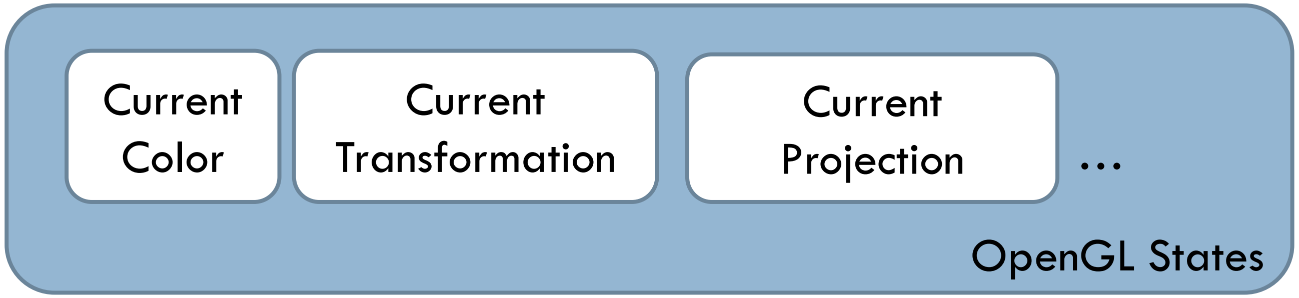
```
gl.bufferData(target, data, usage)
```

Allocate storage and write the data specified by *data* to the buffer object bound to *target*.

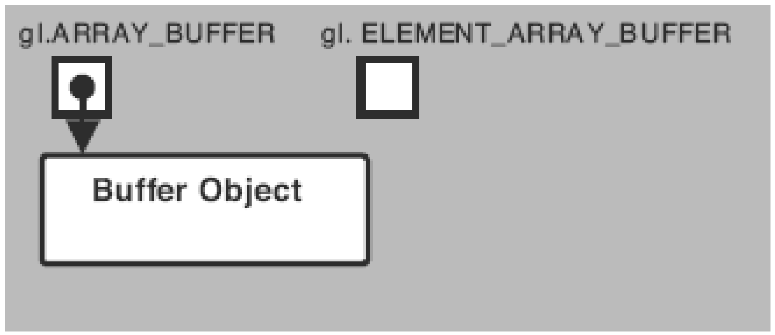
Parameters	target	Specifies `gl.ARRAY_BUFFER` or `gl.ELEMENT_ARRAY_BUFFER`.	
	data	Specifies the data to be written to the buffer object (typed array; see the next section).	
	usage	Specifies a hint about how the program is going to use the data stored in the buffer object. This hint helps WebGL optimize performance but will not stop your program from working if you get it wrong.	
		`gl.STATIC_DRAW`	The buffer object data will be specified once and used many times to draw shapes.
		`gl.STREAM_DRAW`	The buffer object data will be specified once and used a few times to draw shapes.
		`gl.DYNAMIC_DRAW`	The buffer object data will be specified repeatedly and used many times to draw shapes.

OpenGL as a State Machine

- So, before going on further, we should understand OpenGL is a state machine
  - It only keeps one single status for all states



- Every time if you want to apply something new to the current OGL element (objects), you must change the current states



Drawing the Vertices

- Back to the main function, we have

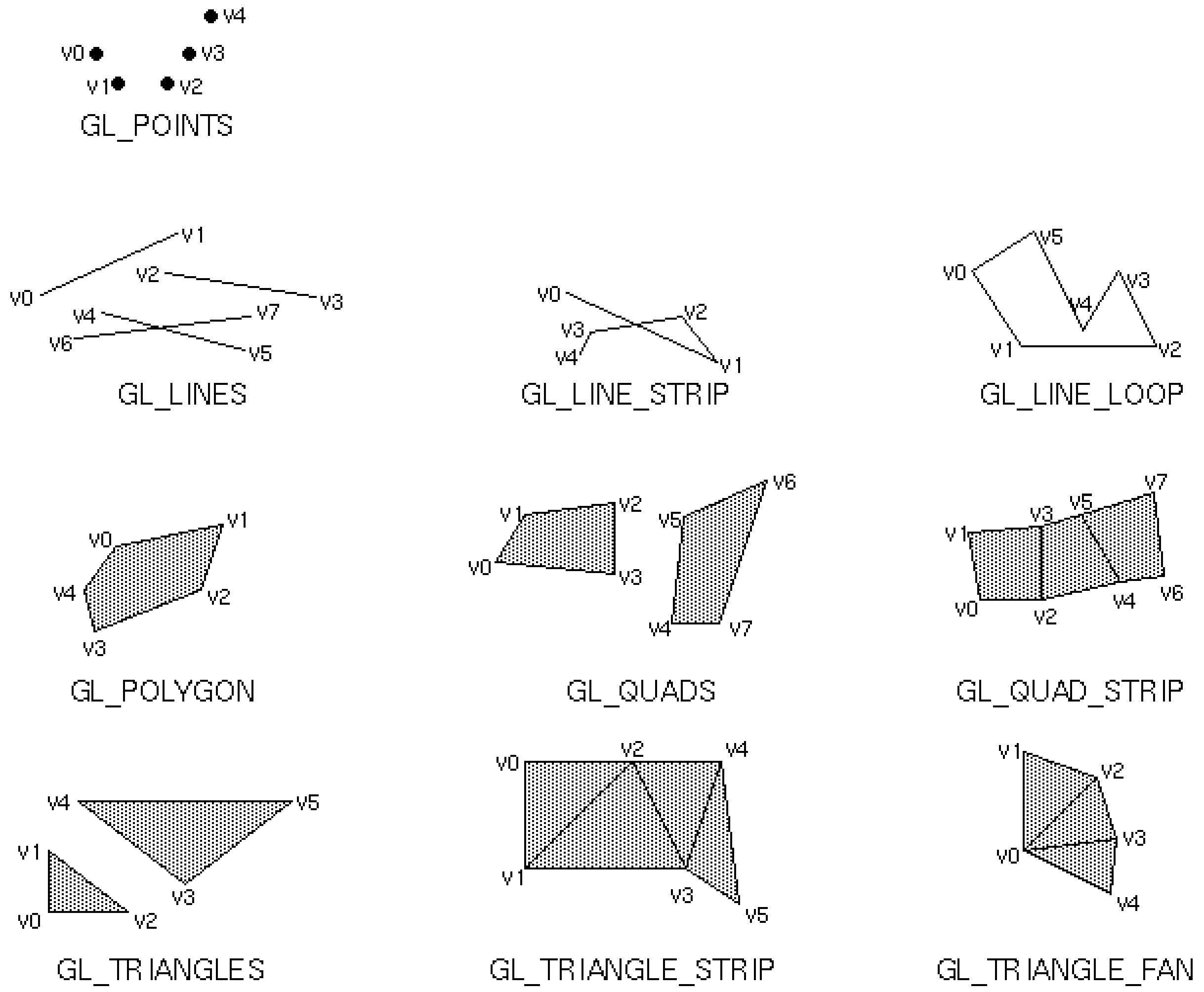
```
// Draw the triangle
gl.drawArrays(gl.TRIANGLES, 0, n);
```

Starting index

Ending index

- Vertices are just discrete points in space
- Need to tell also what to draw
  - gl.TRIANGLES
- 2nd and 3rd parameters tell the starting and ending indices of vertex to draw respectively

Other Methods to Draw



Vertex & Fragment Shaders

- You may find that some of the code are skipped and not discussed like:

main

```
if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
    console.log('Failed to intialize shaders.');
    return;
  }
```

```
// Assign the buffer object to a_Position variable
  gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);

  // Enable the assignment to a_Position variable
  gl.enableVertexAttribArray(a_Position);
....
```

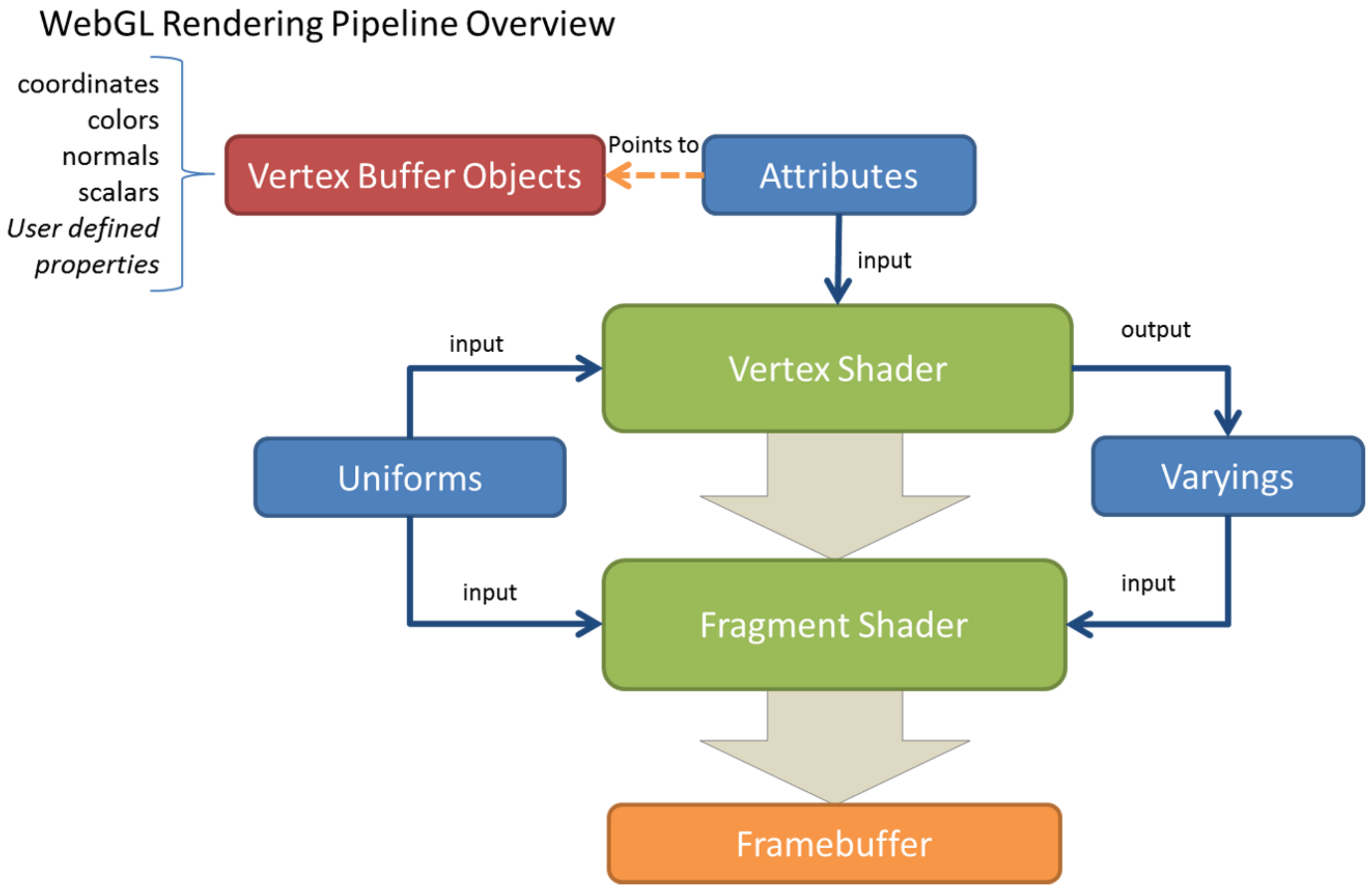
initVertexBuffer

Vertex & Fragment Shaders

These code are related to an advance component in the rendering pipeline, they are :

- Vertex Shader
- Fragment Shader
- Shaders are small programs that
  - Put into the graphics hardware
  - Runs in parallel for every element (e.g. vertex / pixel)
- More details in later lessons…

The Rendering Pipeline in WebGL



The Vertex Shader

- Define the vertex shader

main

```
if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
    ……
```

```
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
  'void main() {\n' +
  '  gl_Position = a_Position;\n' +
  '}\n';
```

Code of vertex shader

The Vertex Shader

- Actually, it is a rather simple shader
- Take a look at initVertexBuffer

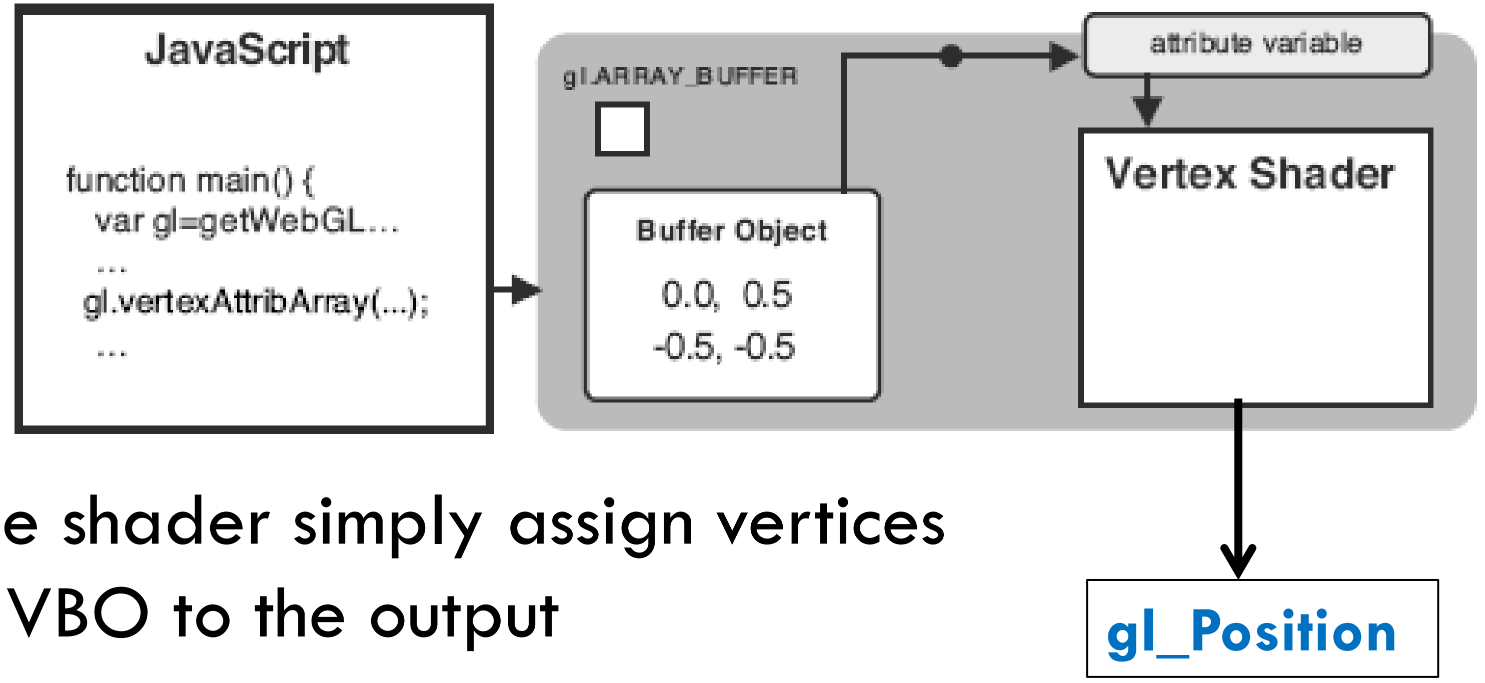
initVertexBuffer

```
// Assign the buffer object to a_Position variable
  gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);
….
```

```
'attribute vec4 a_Position;\n' +
  'void main() {\n' +
  '  gl_Position = a_Position;\n' +
  '}\n
```

The Vertex Shader

- gl_Position is one default output for vertex shader
  - Represent the final position in camera space



- The shader simply assign vertices in VBO to the output

The Fragment Shader

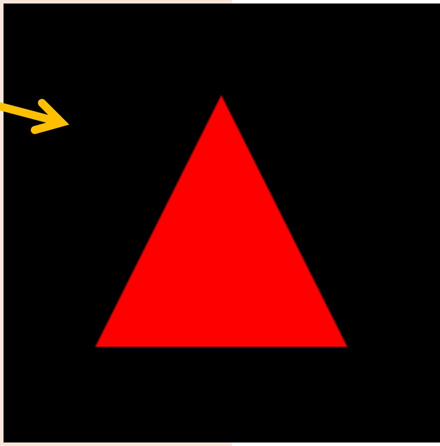
- What we do in fragment shader is
  - Set the output color of pixels in Red (1,0,0,1) in RGBA
- Again, gl_FragColor is one default output in fragment shader, corresponding to the pixel color

```
var FSHADER_SOURCE =
  'void main() {\n' +
  '  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);\n' +
  '}\n';
```

Clear Background

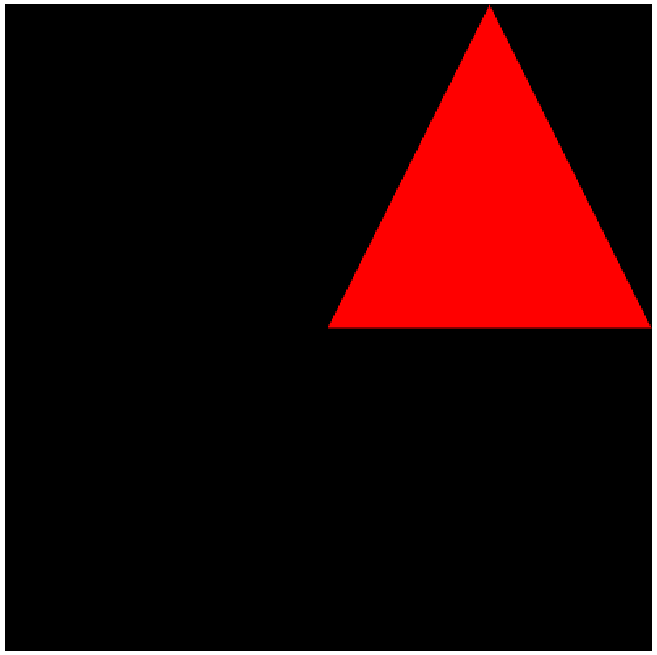
Codes to define background color and do the clear of background

```
// Specify the color for clearing <canvas>
  gl.clearColor(0, 0, 0, 1);

  // Clear <canvas>
  gl.clear(gl.COLOR_BUFFER_BIT);

  // Draw the rectangle
  gl.drawArrays(gl.TRIANGLES, 0, n);
```



Transformation in WebGL

- During lecture, we have taught about what is transformation and the 3 commonly used transformations
- In the **TranslatedTriangle** example
  - We translate the triangle to the top right corner
  - How we can apply the translation within the code?



Translation in Vertex Shader

- In standard OpenGL, we have function "glTranslatef" to easily apply translation (dX, dY, dZ) to a certain object, e.g.

```
glTranslatef(dX, dY, dZ);
```

- Unfortuneately, WebGL left the task to programmer and we must add it explicitly in the vertext shader

Translation in Vertex Shader

- In the vertex shader/program, we make following modifications (in red)

```
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
  'uniform vec4 u_Translation;\n' +
  'void main() {\n' +
  '  gl_Position = a_Position + u_Translation;\n' +
  '}\n';
```

- The vertex shader is performing on each vertex independently

Translation in Vertex Shader

- Notice the variables in shader is with type “vec4”
  - It is a vector with 4 dimensions (homogenous coordinate)

```
gl_Position = a_Position + u_Translation;
```

- Example:

a_Position $\boxed{0.0 , 0.5 , 0.0, 1.0}$

$+$

u_Translation; $\boxed{0.5 , 0.5 , 0.0, 0.0}$

$\downarrow$

gl_Position $\boxed{0.5 , 1.0 , 0.0, 1.0}$

Passing Parameters to Vertex Shader

- How and where **u_Translation** is set?
- First, obtain the handle or storage location of the variable ‘u_Translation’ inside vertex shader

```
var u_Translation = gl.getUniformLocation(gl.program,
'u_Translation');
```

The Handle

- Then, set the parameters with unifrom4f method

```
gl.uniform4f(u_Translation, Tx, Ty, Tz, Tw);
```

Passing Parameters to Vertex Shader

- The parameter passing as a whole:

```
// The translation distance for x, y, and z direction
var Tx = 0.5, Ty = 0.5, Tz = 0.0;
…..
// Pass the translation distance to the vertex shader
  var u_Translation = gl.getUniformLocation(gl.program,
'u_Translation');
  if (!u_Translation) {
    console.log('Failed to get the storage location of u_Translation');
    return;
  }
  gl.uniform4f(u_Translation, Tx, Ty, Tz, 0.0);
```

W component have tobe 0
in homogenous coordinate

Passing Parameters to Vertex Shader

- The two u_Translation can be different
- The green one must match the name in vertex program

```
 var u_Translation =
gl.getUniformLocation(gl.program,
'u_Translation');
```

```
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
  'uniform vec4 u_Translation;\n' +
…..
```

Passing Vertices to Vertex Shader

- The following code is to enable the buffer object assignment in the shader
  - i.e. To pass the vertex positions one by one into shader (var a_Position)

```
 // Assign the buffer object to the attribute variable
  var a_Position = gl.getAttribLocation(gl.program,
'a_Position');
…
  gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);

  // Enable the assignment to a_Position variable
  gl.enableVertexAttribArray(a_Position);
```

Summary of Steps in Translation

JS

- Get storage location of translation variable and vertex buffer in shader
- gl.**getUniformLocation**
- gl.**getAttribLocation**

JS

- Set the translation variable via the storage location
- gl.**uniform4f**

Shader

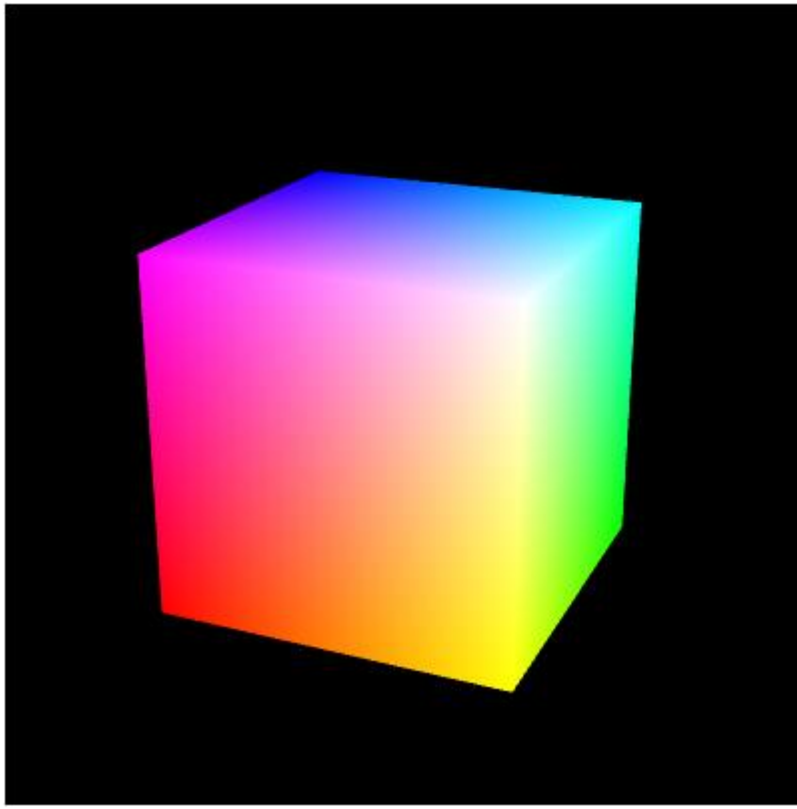
- Adding the vertex position to the translation variable
- gl_Position = **a_Position + u_Translation;**

Part II

3D Rendering in WebGL

Drawing 3D Objects

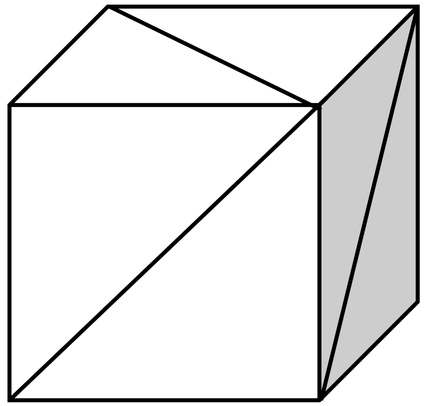
- HelloCube example
  - How a cube can be drawn
  - How we rotate the cube
  - How to apply color on vertices



Cube Drawing

- 6 squares are drawn separately
- Each of the square is composed of 2 triangles
- The 8 corner vertices are defined as

```
var verticesColors = new Float32Array([
   // Vertex coordinates and color
    1.0,  1.0,  1.0,    1.0,  1.0,  1.0,  // v0 White
   -1.0,  1.0,  1.0,    1.0,  0.0,  1.0,  // v1 Magenta
   -1.0, -1.0,  1.0,    1.0,  0.0,  0.0,  // v2 Red
    1.0, -1.0,  1.0,    1.0,  1.0,  0.0,  // v3 Yellow
    1.0, -1.0, -1.0,    0.0,  1.0,  0.0,  // v4 Green
    1.0,  1.0, -1.0,    0.0,  1.0,  1.0,  // v5 Cyan
   -1.0,  1.0, -1.0,    0.0,  0.0,  1.0,  // v6 Blue
   -1.0, -1.0, -1.0,    0.0,  0.0,  0.0   // v7 Black
 ]);
```



VertexBuffer

- Similar to our triangle example, we have to
  - Bind our VBO with Array_buffer in the engine
  - Write data into VBO

```
var vertexColorBuffer = gl.createBuffer();
...
  // Bind the buffer object to target
  gl.bindBuffer(gl.ARRAY_BUFFER,
vertexColorBuffer);

  // Write data into the buffer object
  gl.bufferData(gl.ARRAY_BUFFER,
verticesColors, gl.STATIC_DRAW);


  .....
```

Cube Drawing

- Define the triangles
  - Using indices of the vertices

```
var indices = new Uint8Array([
    0, 1, 2,   0, 2, 3,    // front
    0, 3, 4,   0, 4, 5,    // right
    0, 5, 6,   0, 6, 1,    // up
    1, 6, 7,   1, 7, 2,    // left
    7, 4, 3,   7, 3, 2,    // down
    4, 7, 6,   4, 6, 5     // back
 ]);
```

```
  1.0,  1.0,  1.0,  ...
 -1.0,  1.0,  1.0, ...
 -1.0, -1.0,  1.0, ...
  1.0, -1.0,  1.0, ...
  1.0, -1.0, -1.0, ...
  1.0,  1.0, -1.0, ...
 -1.0,  1.0, -1.0, ...
 -1.0, -1.0, -1.0, ...
```

IndexBuffer

- Similar to vertex buffer
- We create a Buffer Object for indices

```
var indexBuffer = gl.createBuffer();
```

- Then, bind and set the index buffer with ELEMENT_ARRAY_BUFFER parameter

```
// Bind the buffer object to target
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);

  // Write the indices to the buffer object
  gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, indices,
gl.STATIC_DRAW);
  …..
```

Drawing the IndexBuffer into triangles

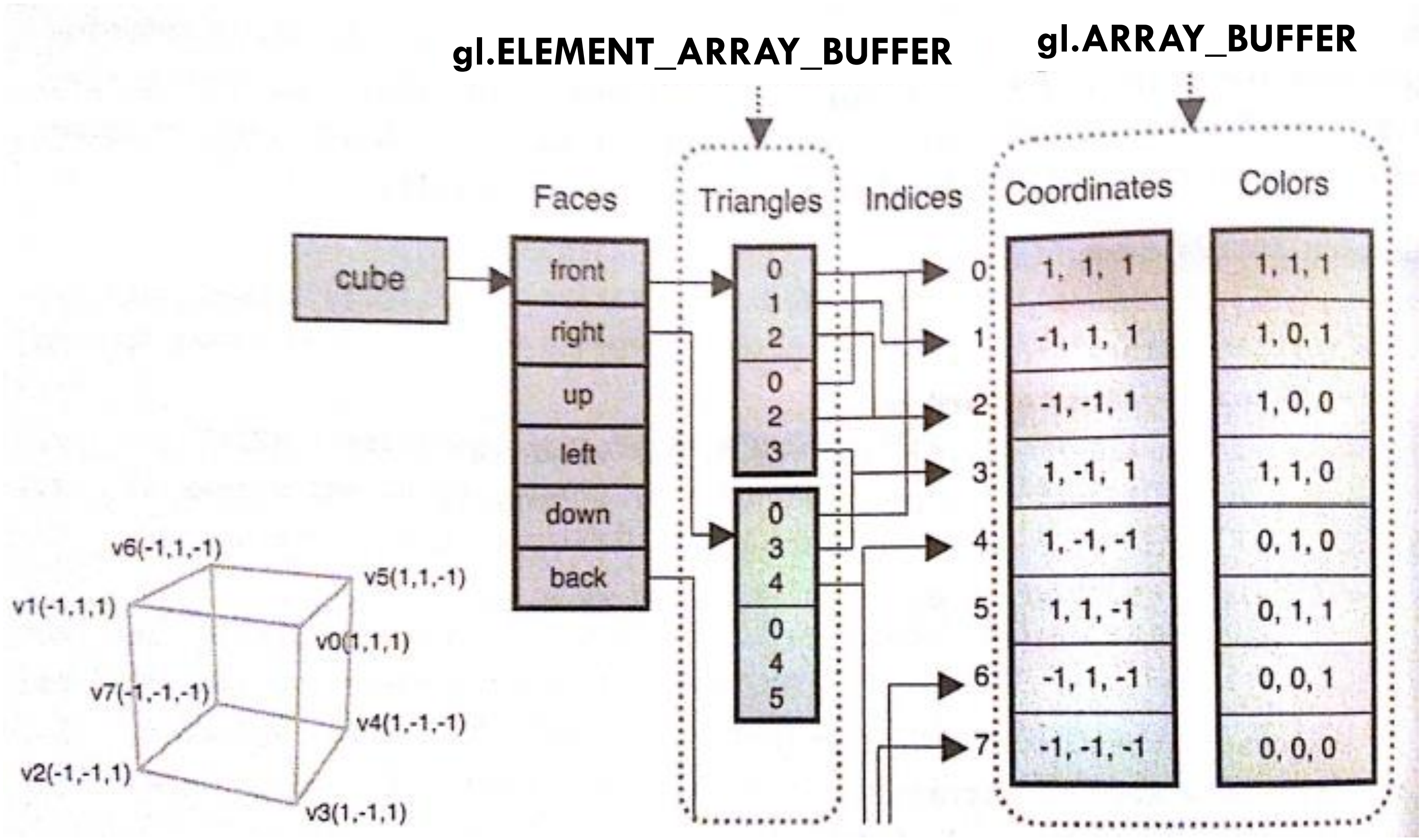
- Finally, drawing the index buffer with "drawElements" method

```
// Draw the cube
gl.drawElements(gl.TRIANGLES, n, gl.UNSIGNED_BYTE, 0);
```

- We specify we are drawing triangles, the number of indices (n) , and data type of indices used

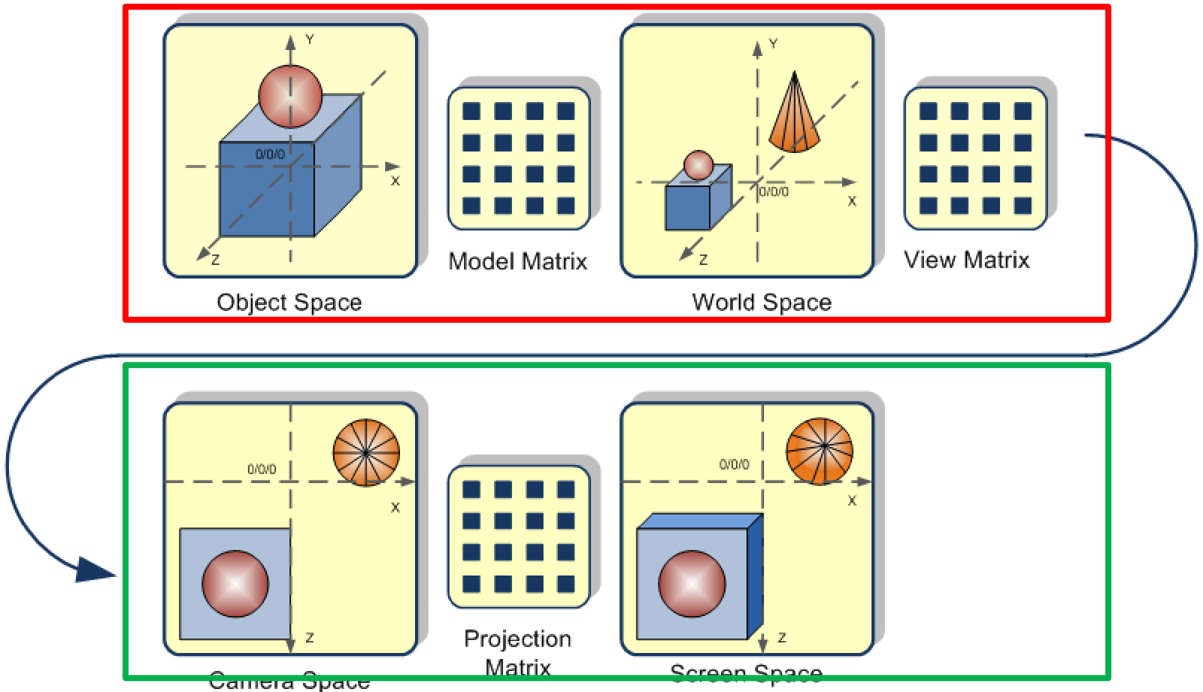
Cube Drawing

- Whole picture of cube drawing



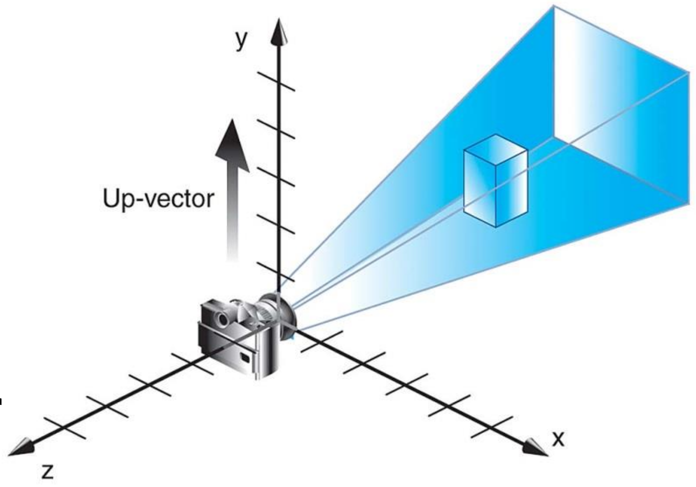
ModelView & Projection Matrix in WebGL

- As discussed in lecture 3, we have two matrices in a rendering engine
  - ModelView (from object space to camera space)
  - Projection (from camera space to screen space)



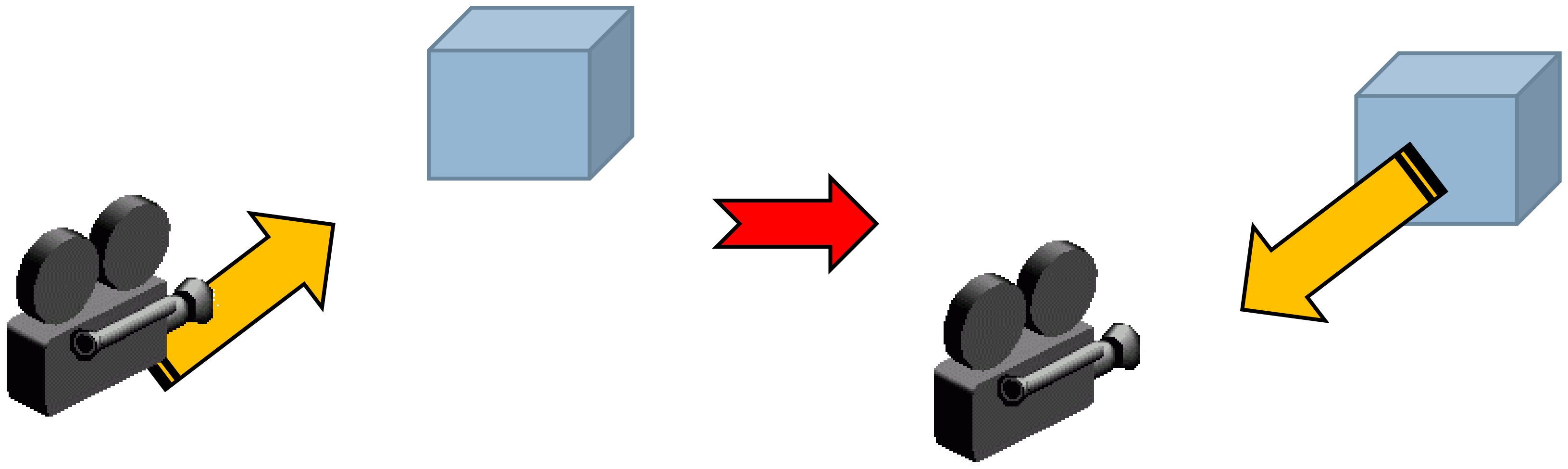
Transformation in OpenGL

- By default, your camera is positioned at the origin (i.e. (0,0,0)) and facing to the neaative Z
- So, when you are drawing something at Z=0, you can't see it !!
- Move a bit to –ve Z will let you see the shapes



Moving the Camera

- In standard OpenGL, there is no such a concept of "moving the Camera"
- To achieve this effect, we can move object instead
- E.g. to move the camera towards to object, we can move the object towards the camera instead



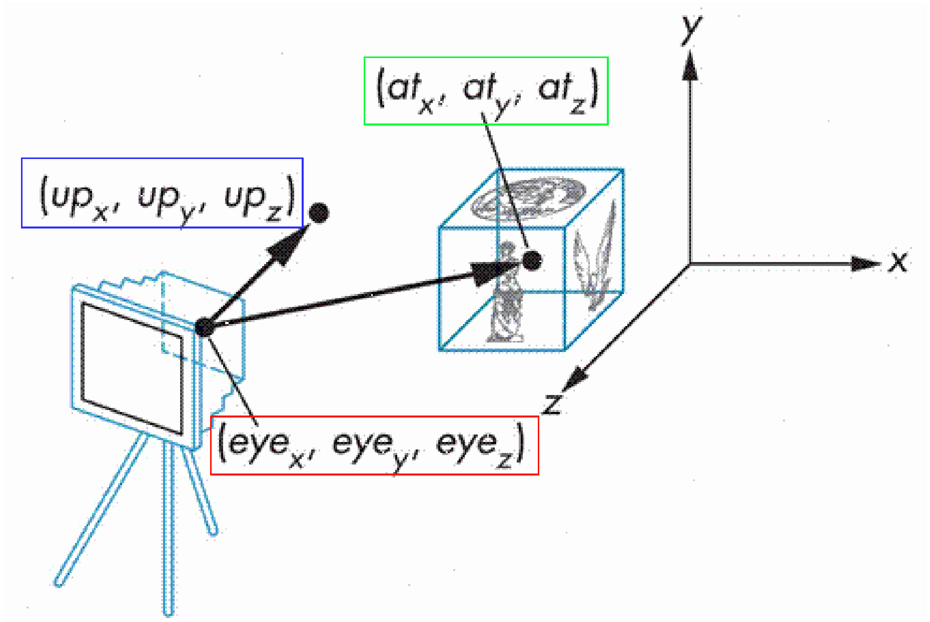
lookAt in WebGL

- A convenient function is available in WebGL
  - to help computing the matrix for the purpose form, i.e. an inverse of transformation matrix of the camera
- This method is called “LookAt” which is defined within the Matrix4 class:

```
void Matrix4::LookAt(
double eyex, double eyey, double eyez,
double atx, double aty, double atz,
double upx, double upy, double upz );
```

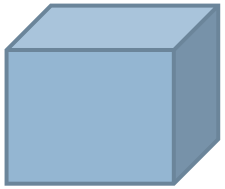
LookAt in WebGL

- We have to provide 3 vectors/vertices to gluLookAt
  - Eye position (camera position)
  - Look at position
  - Up vector (the upward direction of your camera)
- So totally we have 9 parameters



LookAt in WebGL

- In HelloCube sample, we have
  - Object : (0,0,0) world center
  - Camera : (3,3,7)
  - Camera up direction : (0,1,0) Y axis



```
var mvpMatrix = new Matrix4();
....
mvpMatrix.lookAt(3, 3, 7, 0, 0, 0, 0, 1, 0);
```

- i.e. our camera is at (3,3,7), looking at the origin (0,0,0), and our up direction is y-axis (0,1,0)

Projection Matrix in WebGL

- The projection matrix is defined for the conversion from camera space to screen space
  - Either perspective or orthographic projections can be used
- In the HelloCube sample, we use the perspective:

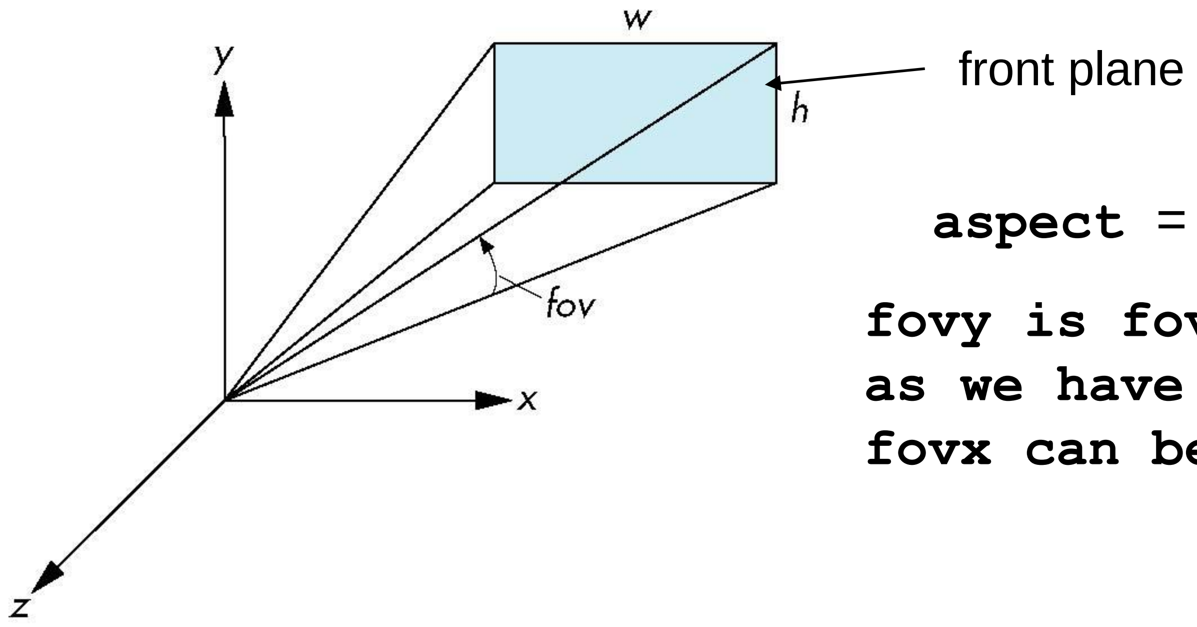
```
mvpMatrix.setPerspective(30, 1, 1, 100);
```

  - For orthographic, please refers to “setOrtho” method
- Here, we fov to 30 degree, with aspect ratio 1, near plane is 1 and far plane is 100

gluPerspective

- The definition of the method is like

```
void Matrix4::setPerspective(double fovy, double aspect, double zNear, double zFar);
```



`aspect = w/h`

`fovy is fov in y direction, as we have aspect ratio, fovx can be determined`

Passing the Matrices to Shader

- After we defined the Modelview and projection matrices, we pass it to the vertex shader
- Steps are rather similar to examples before

```
 var u_MvpMatrix = gl.getUniformLocation(gl.program,
'u_MvpMatrix');
……
// Pass the model view projection matrix to u_MvpMatrix
 gl.uniformMatrix4fv(u_MvpMatrix, false,
mvpMatrix.elements);
```

Transformation in Vertex Shader

- Parameters for the method "uniformMatrix4fv"

```
gl.uniformMatrix4fv(location,  istranspose,  values)
```

  - The second parameter tells if the matrix need to transpose
- The corresponding shader tries to multiply the vertex position to the matrix "u_MvpMatrix"

```
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' + …
   'uniform mat4 u_MvpMatrix;\n' + …
  'void main() {\n' +
  '  gl_Position = u_MvpMatrix * a_Position;\n' +  …
  '}\n';
```

Apply Colors on Vertices

- Finally, we look at how we apply colors to vertices
- We find the vertice array are defined together with the colors
  - Colors are in RGB values normalized to [0-1]

```
var verticesColors = new Float32Array([
   // Vertex coordinates and color
    1.0,  1.0,  1.0,    1.0,  1.0,  1.0,  // v0 White
   -1.0,  1.0,  1.0,    1.0,  0.0,  1.0,  // v1 Magenta
   -1.0, -1.0,  1.0,    1.0,  0.0,  0.0,  // v2 Red
    1.0, -1.0,  1.0,    1.0,  1.0,  0.0,  // v3 Yellow
    1.0, -1.0, -1.0,    0.0,  1.0,  0.0,  // v4 Green
    1.0,  1.0, -1.0,    0.0,  1.0,  1.0,  // v5 Cyan
   -1.0,  1.0, -1.0,    0.0,  0.0,  1.0,  // v6 Blue
   -1.0, -1.0, -1.0,    0.0,  0.0,  0.0   // v7 Black
 ]);
```

Apply Colors on Vertices

- In the javascript, we create buffer object together with the vertice positions
- Bind and assign values to the buffer object

```
var vertexColorBuffer = gl.createBuffer();
…
  // Bind the buffer object to target
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexColorBuffer);

  // Write data into the buffer object
  gl.bufferData(gl.ARRAY_BUFFER, verticesColors,
gl.STATIC_DRAW);


  …..
```

Apply Colors on Vertices

- In WebGL, we also need to tell how to handle the colors in both vertex and fragment shader
- Steps are similar to applying matrix transformation
- In the vertex shader:

```
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
  'attribute vec4 a_Color;\n' +
  'uniform mat4 u_MvpMatrix;\n' +
  'varying vec4 v_Color;\n' +
  'void main() {\n' +
  '  gl_Position = u_MvpMatrix * a_Position;\n' +
  '  v_Color = a_Color;\n' +
  '}\n';
```

Apply Colors on Vertices

- In Fragment Shader:

```
var FSHADER_SOURCE =
  '#ifdef GL_ES\n' +
  'precision mediump float;\n' +
  '#endif\n' +
  'varying vec4 v_Color;\n' +
  'void main() {\n' +
  '  gl_FragColor = v_Color;\n' +
  '}\n';
```

Obtain from Vertex Shader

- We can find that both shader just simply assign the color values (vec4 type) to its output

Apply Colors on Vertices

- Again, we are not going to discuss in detail the following code, which just to enable the buffer object assignment in the shader

```
// Assign the buffer object to a_Color and enable the assignment
  var a_Color = gl.getAttribLocation(gl.program, 'a_Color');
  if(a_Color < 0) {
    console.log('Failed to get the storage location of a_Color');
    return -1;
  }
  gl.vertexAttribPointer(a_Color, 3, gl.FLOAT, false, FSIZE * 6, FSIZE * 3);
  gl.enableVertexAttribArray(a_Color);
```

Depth Test

- Worth to notice that we have enabled depth testing in WebGL by invoking glEnable with parameter DEPTH_TEST
  - This will make the drawing of object from front to back in the scene, instead of purely based on the drawing order

```
……
gl.enable(gl.DEPTH_TEST);
// apply to faces/vertices afterwards
……
```

Summary

- We have explained how to setup a WebGL in browser, so that we can start use it to draw stuff
- We have learnt how to draw simple shapes in WebGL
- Also Transformation, Coloring in WebGL
- Remember WebGL is a state machine which only keeps current status of different attribute states
- **Reference : WebGL Programming Guide**