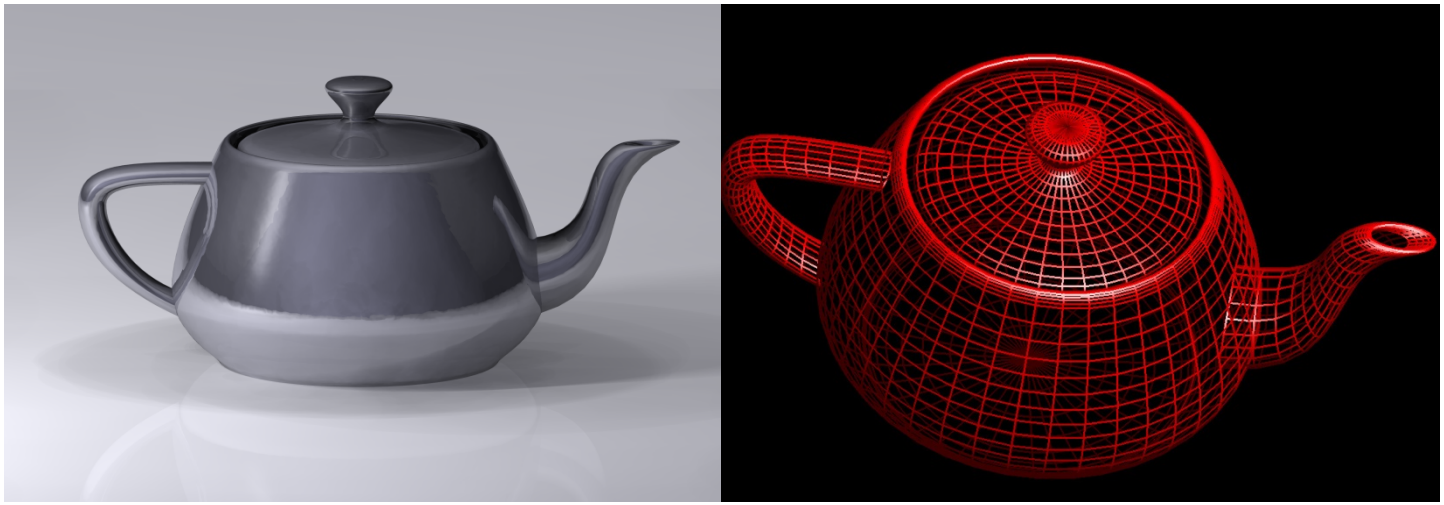


# *Foundations of Computer Graphics*

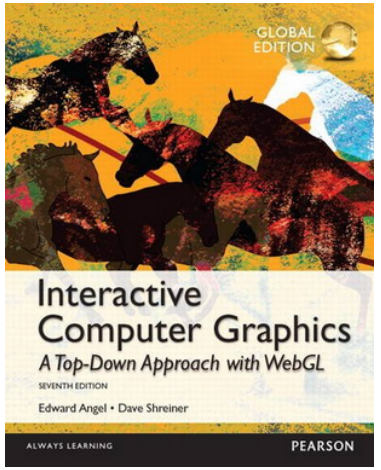
Saurabh Ray

# Reading

Reading for Lecture 2: Sections 2.1-2.4, 2.8.

Reading for Lecture 3: Sections 2.10, 3.1.

I suggest reading whole chapters even though not everything in the book was discussed in class.



# WebGL Example: Clear the clutter

A simple example that clears the canvas with a background color.

**File:** ClearCanvas.html

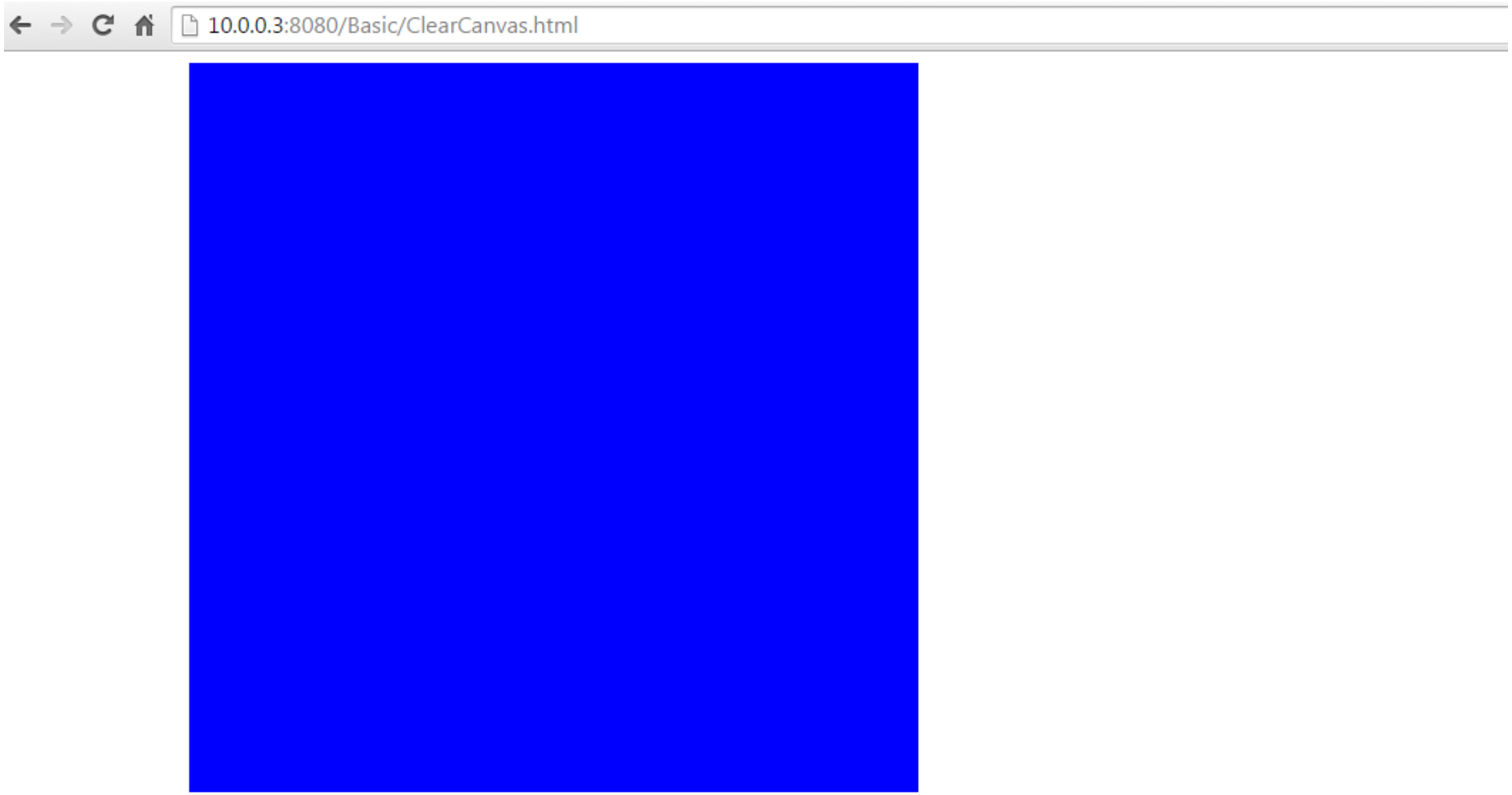
```
<html>
<head>
  <script type="text/javascript" src="../Common/webgl-utils.js"></script>
  <script type="text/javascript" src="ClearCanvas.js"></script>
</head>
<body>
   <canvas id="gl-canvas" width="512" height="512">
    HTML5 Canvas not supported!
   </canvas>
</body>
</html>
```

**File:** ClearCanvas.js

```
window.onload = function init() {
            var canvas = document.getElementById("gl-canvas");
            var gl = WebGLUtils.setupWebGL( canvas );
            if(!gl){alert("WebGL setup failed!");}

            gl.clearColor(0.0, 0.0, 1.0, 1);
            gl.clear(gl.COLOR_BUFFER_BIT);
}
```

# WebGL Example: Clear the clutter

Loading the html file `ClearCanvas.html` in a browser gives you:



# WebGL Example: Clear the clutter

**File:** `ClearCanvas.html`

```
<html>
<head>
  <script type="text/javascript" src="../Common/webgl-utils.js"></script>
  <script type="text/javascript" src="ClearCanvas.js"></script>
</head>
<body>
   <canvas id="gl-canvas" width="512" height="512">
    HTML5 Canvas not supported!
   </canvas>
</body>
</html>
```

*We are asking the browser to load these Javascript files.*

*Displayed if the* `canvas` *element is not supported.*

An HTML file consists of a bunch of **tags**.

Each tag describes some document content. Format: `<tag> content </tag>`

`<head>` ... `</head>` : information about the document

`<body>` ... `</body>` : visible content

`webgl-utils.js`: contains useful functions for setting up WebGL.

# WebGL Example: Clear the clutter

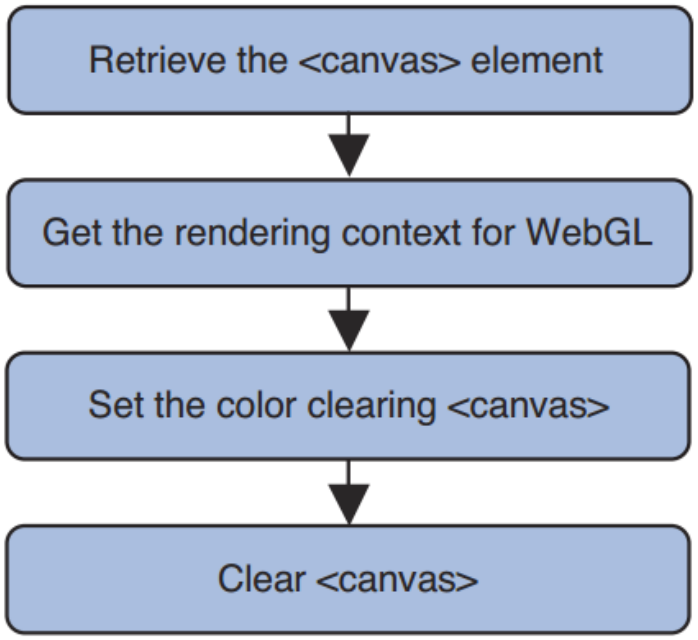
**File:** ClearCanvas.js

*This function is called once all scripts are loaded.*

```
window.onload = function init() {
    var canvas = document.getElementById("gl-canvas");
    var gl = WebGLUtils.setupWebGL( canvas );
    if(!gl){alert("WebGL setup failed!");}

    gl.clearColor(0.0, 0.0, 1.0, 1);
    gl.clear(gl.COLOR_BUFFER_BIT);
}
```

*retrieve canvas element*

`gl` *is now a javascript object that contains all webgl functions and parameters.*




*RGBA : (Red, Green, Blue, Alpha)*

*Alpha is for opaqueness.*

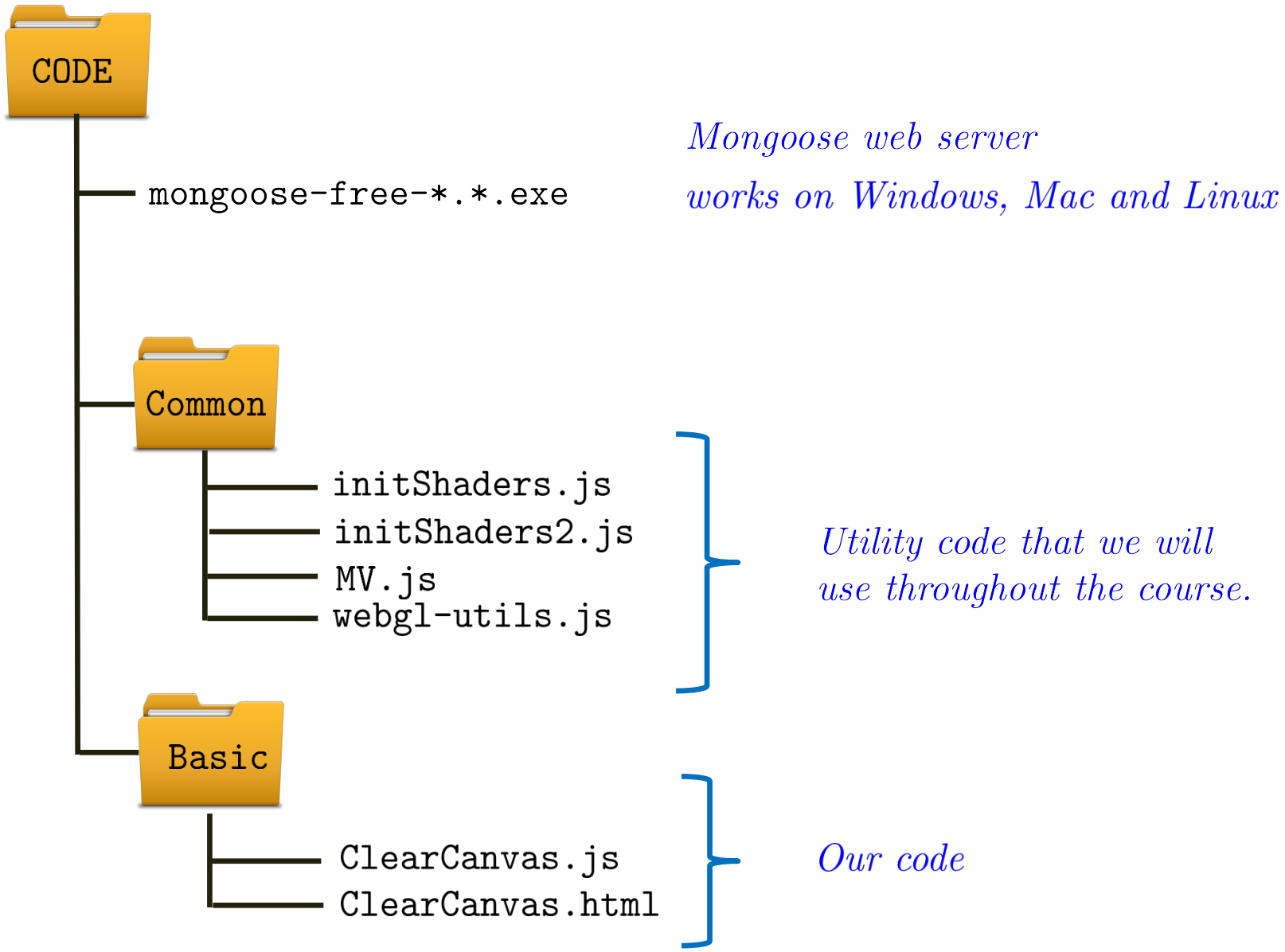
$\alpha = 0$: *fully transparent*

$\alpha = 1$: *fully opaque*

*We are asking WebGL to use the color buffer.*

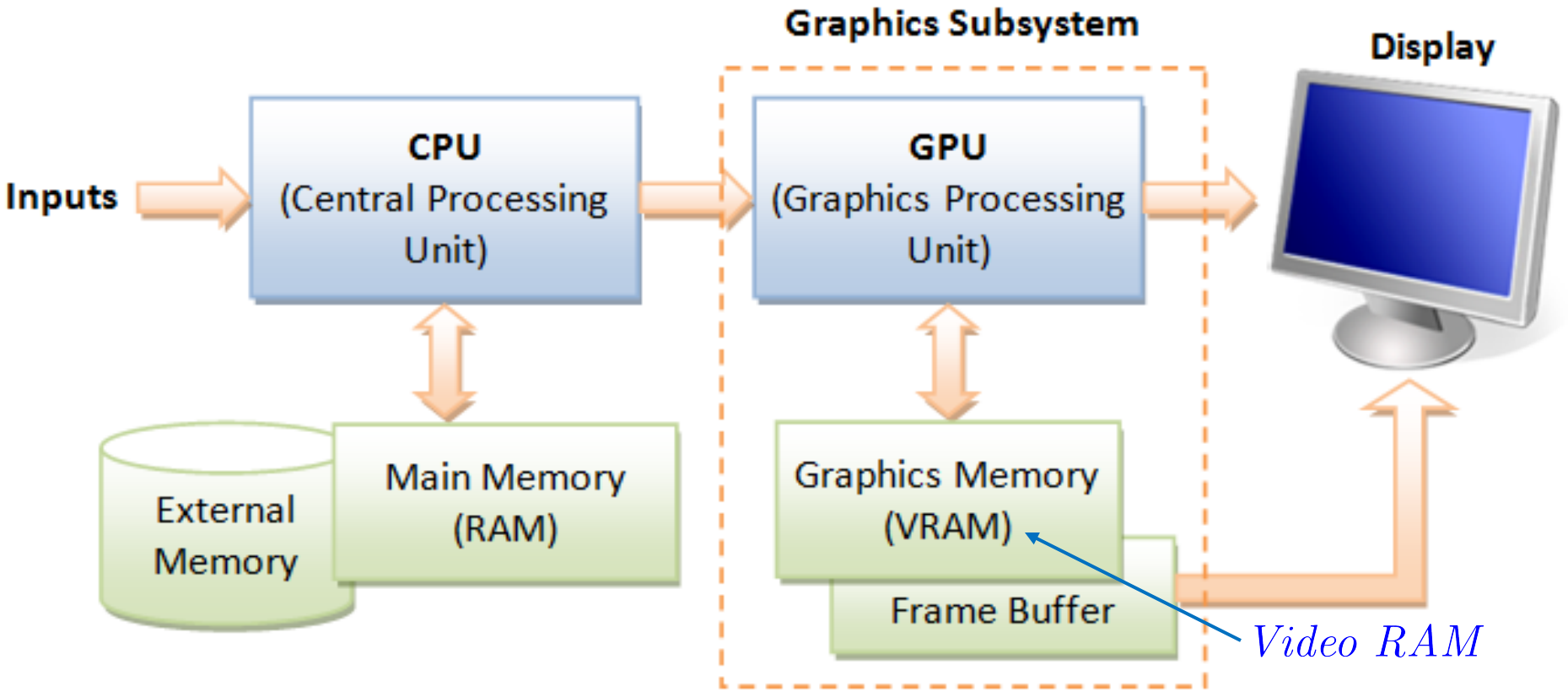
# WebGL Example: Clear the clutter

**Folder Structure:**



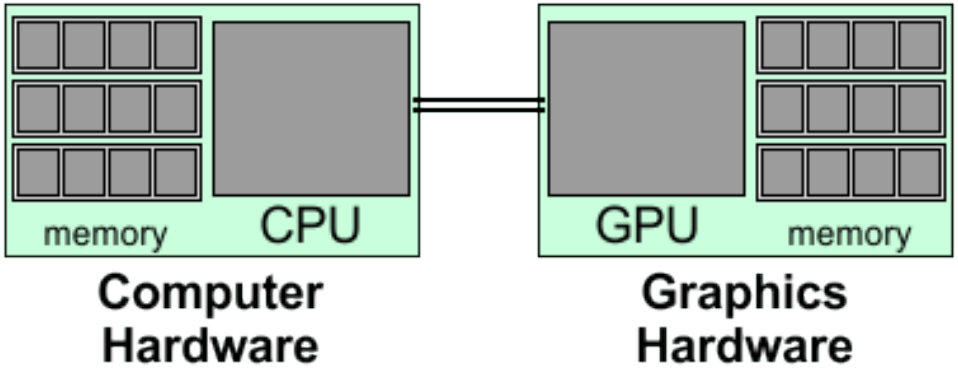

Can't go any further without learning about the graphics system.

# Graphics System



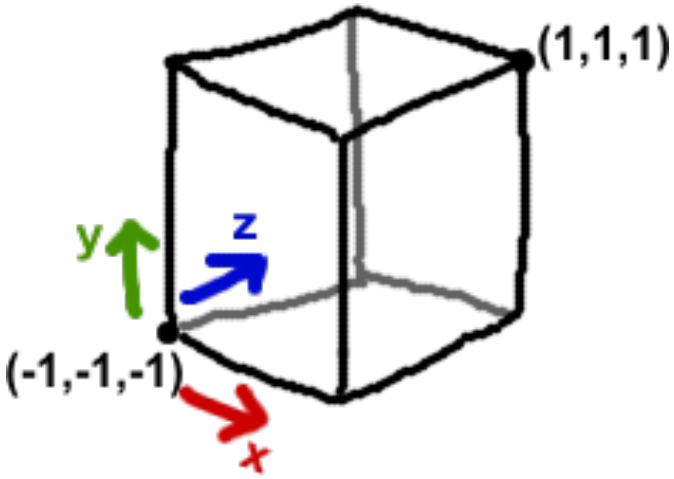

GPU: a specialized electronic circuit to speed up the computation required for graphics.

# Graphics System

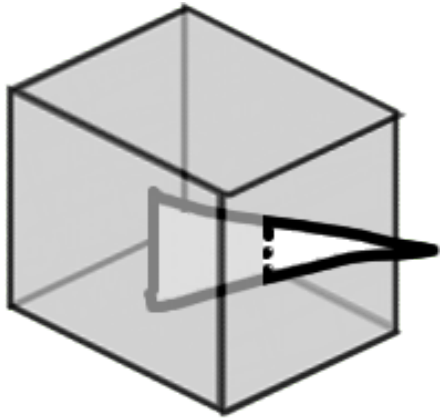



- CPU and GPU are separate entities and have separate memories
- GPU specializes in running multiples copies of the same program that operate on different data. The programs are usually small.
- Main bottleneck: communication between CPU and GPU. Should be minimized.

# WebGL Graphics Model

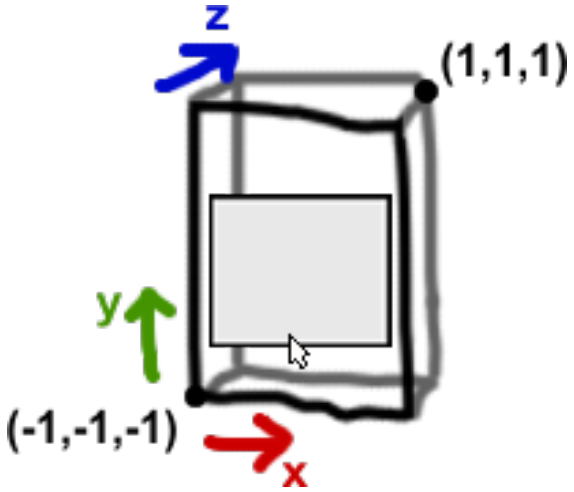



All 3d graphics must be done within this cube.



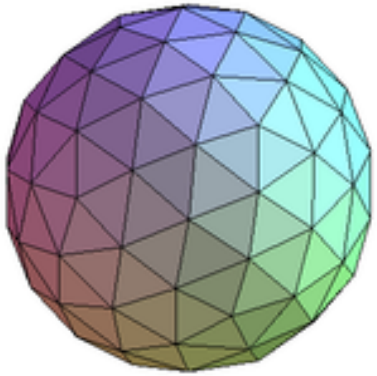
Anything going out is clipped.

Whatever is drawn into the view volume is projected to the 2D screen by dropping the $z$-coordinate.




**Note:** Left Handed Coordinate System!

# WebGL Graphics Model



Typically you model an object as a surface formed by a bunch of triangles.

Then you draw each triangle.

Thats a lot of data: coordinates of all vertices. Very slow but can't avoid..

Now suppose that the model is moved - say it is rotated and translated.

This changes the coordinates of all the vertices.

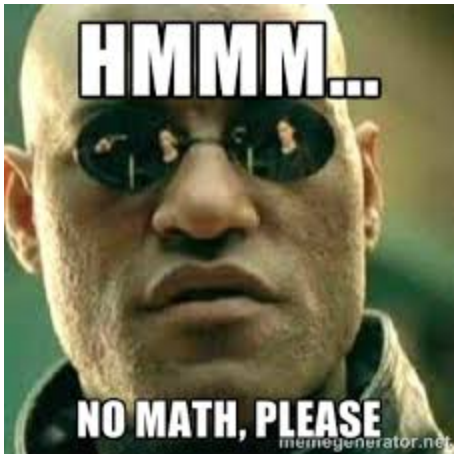
What do you do? Send all that data again?

# WebGL Graphics Model

No! The transformation can be encoded as small matrix. Just send the matrix.

Don't transform the data in the CPU.

Just do it in the GPU.




In reality, its more like this:

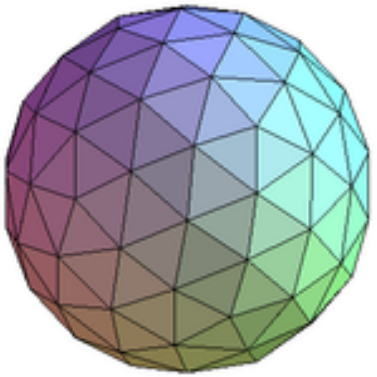
$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

## WebGL say

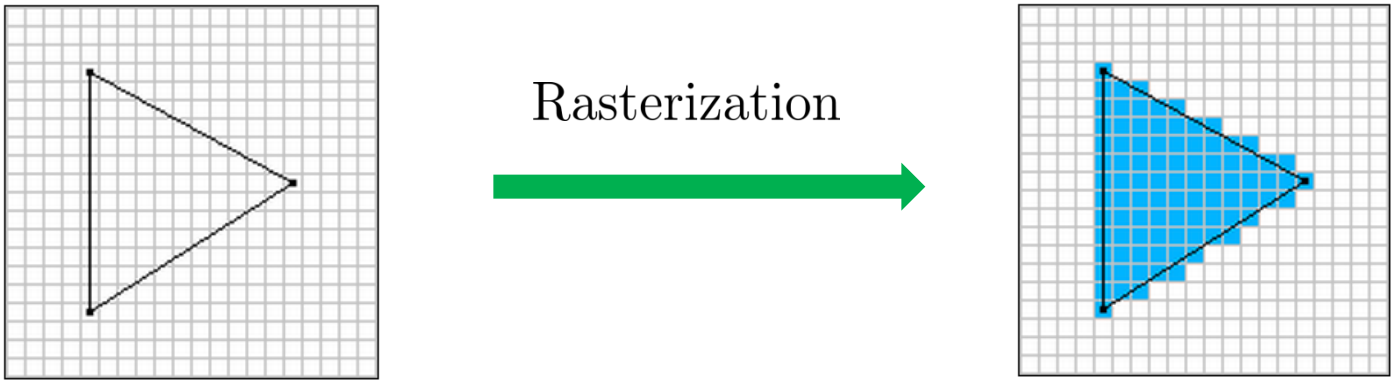
Do whatever you want in the **vertex shader**.
I will run it once for each vertex.

# WebGL Graphics Model

Ok, so we have transformed all the vertices in the GPU.



How do we draw the triangles? Our screens can only draw pixels!

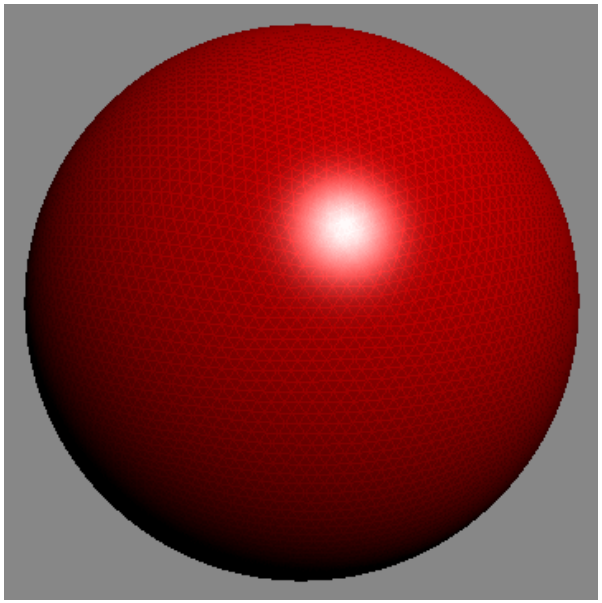



Each triangle is converted to **fragments** by the **rasterizer**.

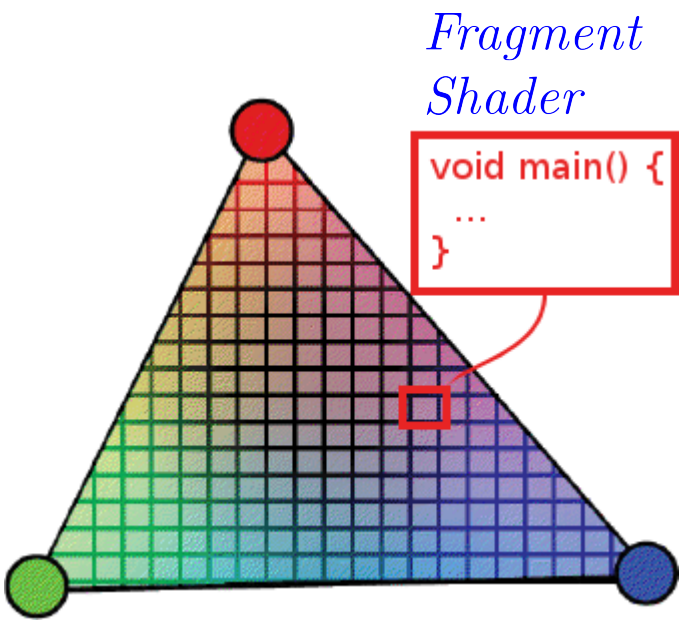
*We use the term 'fragment' instead of 'pixel'.*

# WebGL Graphics Model

In general, different pixels of the same triangle may have different colors.



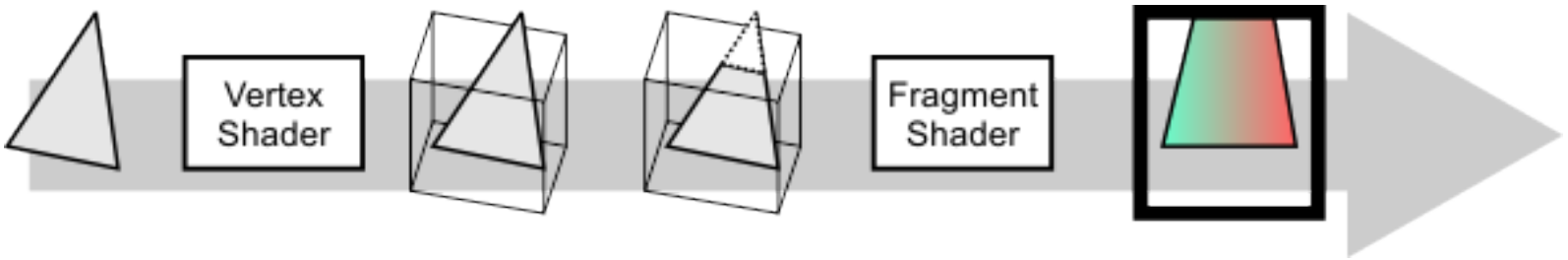
The GPU runs a **fragment shader** for each fragment to compute its final color.




Do whatever you want in the **fragment shader**. I will run it once for each fragment.

# WebGL Graphics Model

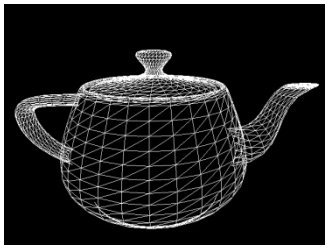
**Life in the graphics pipeline:**




The **vertex and fragment shaders are programmable**, i.e., we provide the the code for these.

We have no control over the rasterizer.

# Graphics Pipeline



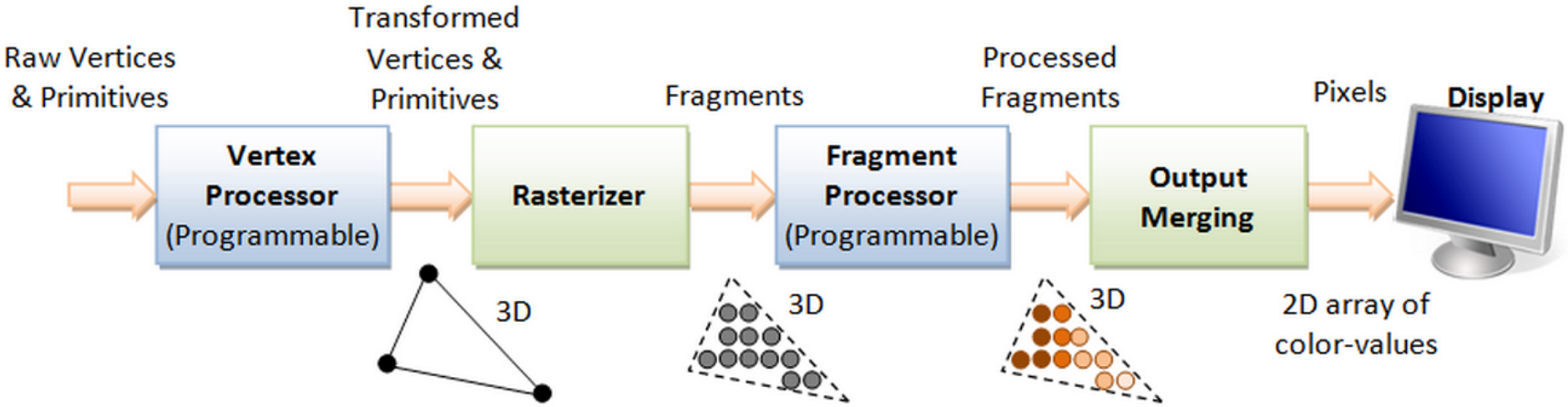
3D Model

Geometric Processing

Rasterization

Fragment Processing

Output



How do you transfer data and programs (shaders) to the GPU?



There are WebGL functions that do it for you!

# WebGL Example: Draw a point

**File:** DrawPoint.html

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;charset=utf-8" >

<title>Draw Point</title>
```

```
<script id="vertex-shader" type="x-shader/x-vertex">
void main(){
    gl_PointSize = 10.0;
    gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
}
</script>
```

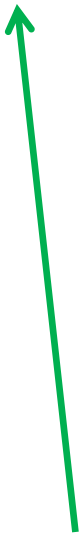
*written in GLSL ES*

```
<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>
```

*written in GLSL ES*

# WebGL Example: Draw a point

```
<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="DrawPoint.js"></script>
</head>

<body>
<canvas id="gl-canvas" width="512" height="512">
  HTML5 Canvas not supported!
</canvas>
</body>
</html>
```



*We are loading a few more JS files.*

# WebGL Example: Draw a point

**File:** DrawPoint.js

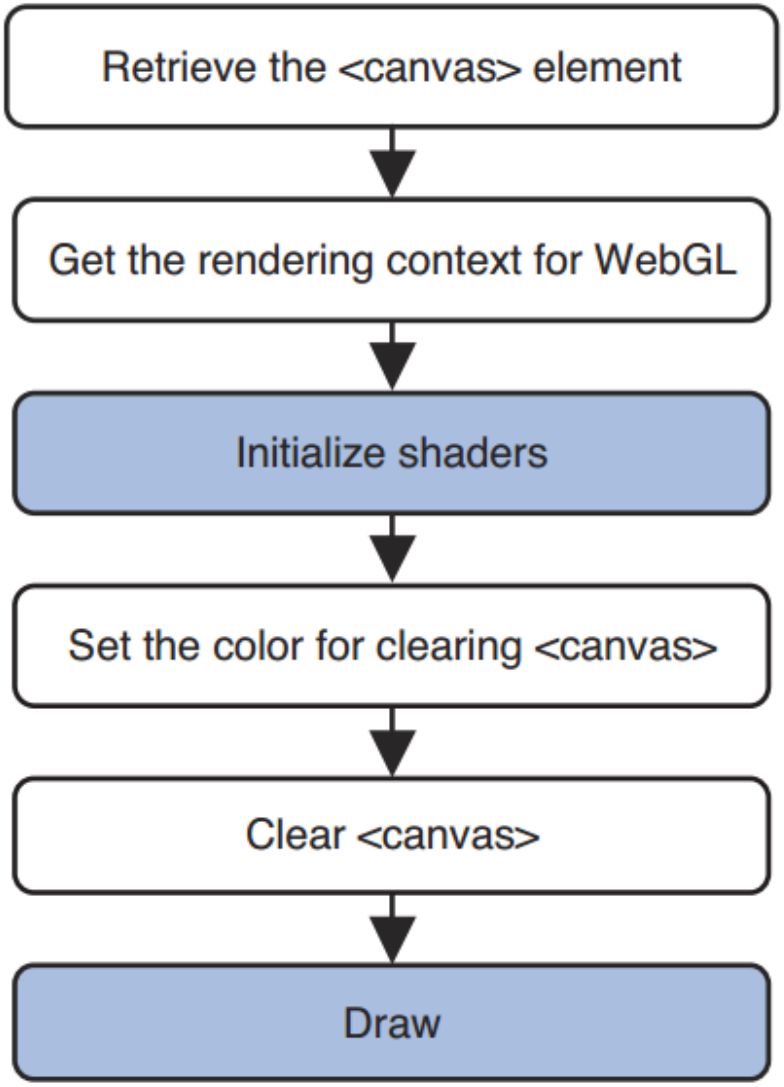
```
"use strict";
var gl; // global variable

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

    // 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 );

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

    //Draw a point
    gl.drawArrays(gl.POINTS,0,1);
};
```

*Fetch shader codes, compile them and link them to get a* `program`*. From now on use this* `program`*.*

*Draw 1 point starting at id 0.*

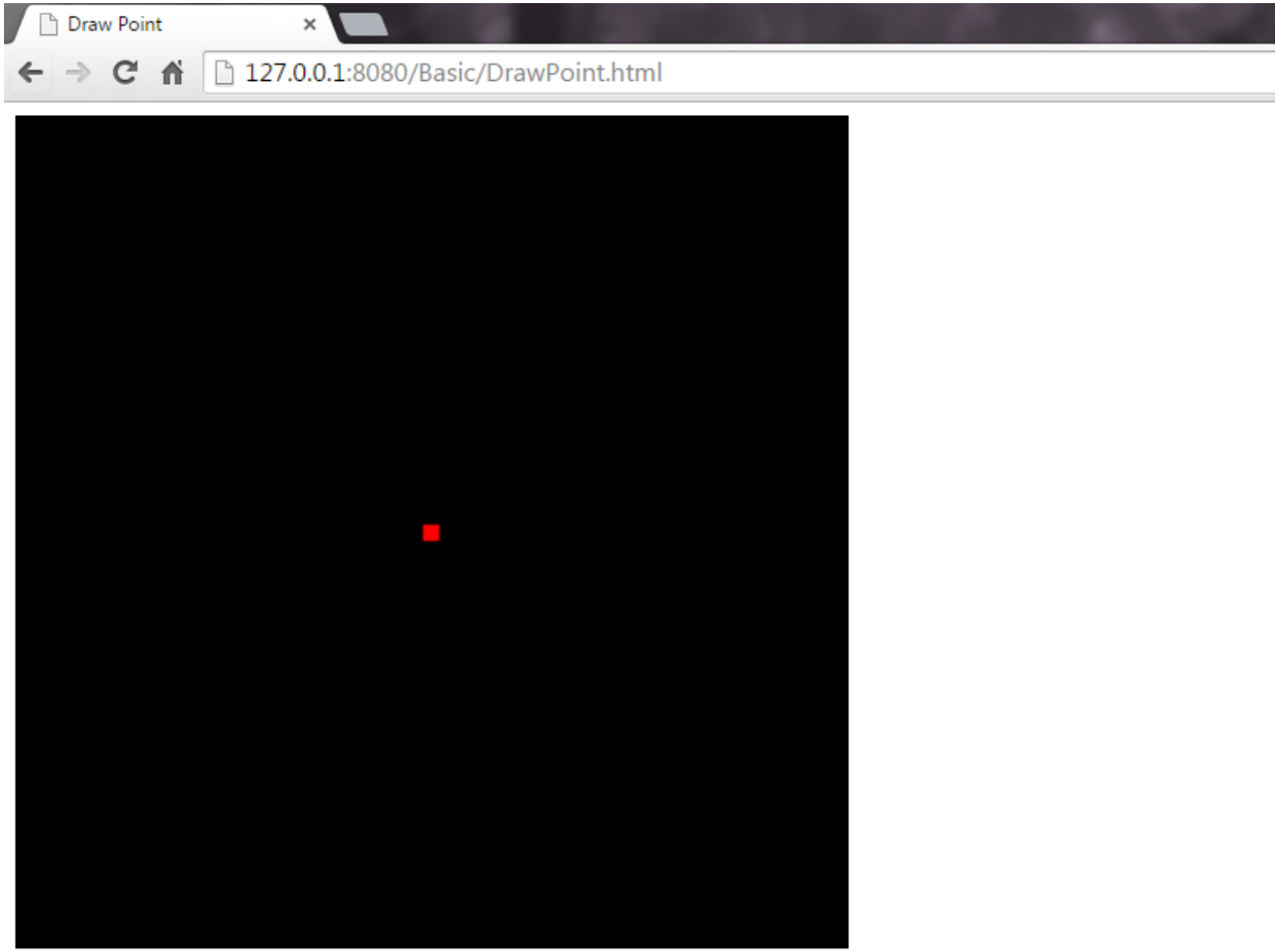
# WebGL Example: Draw a point

**Flow:**

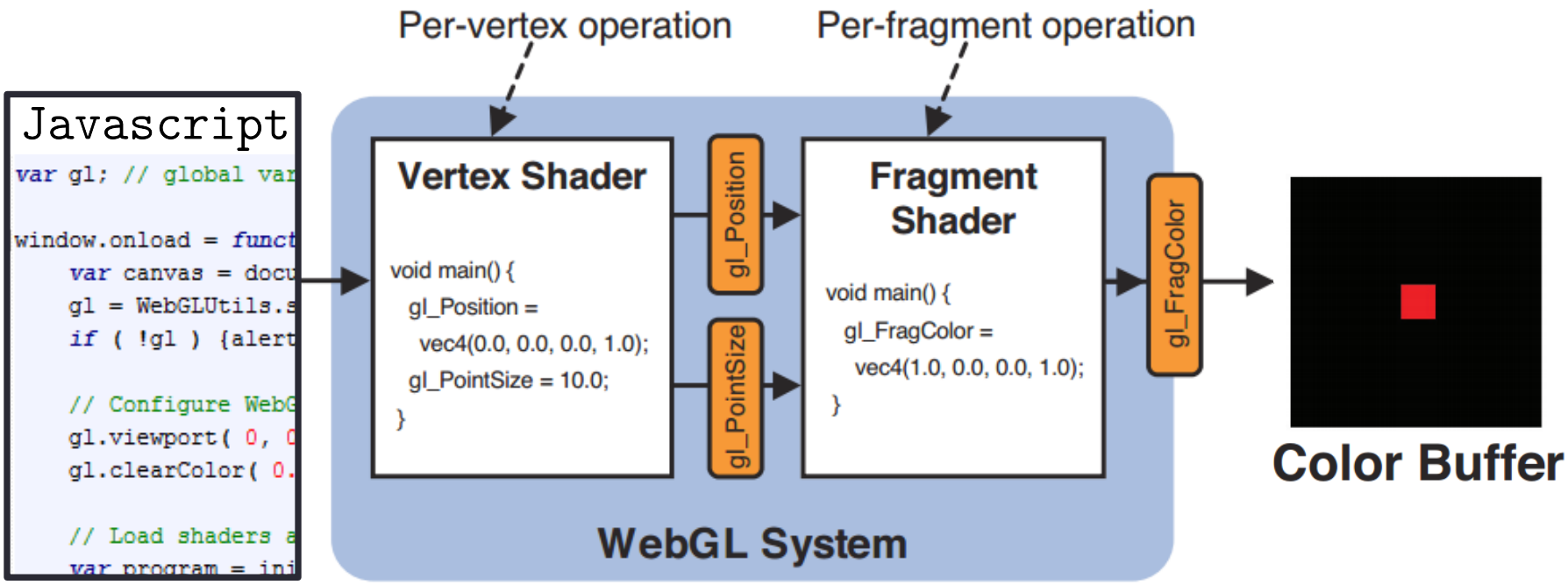


# WebGL Example: Draw a point

**Output:**



# WebGL Example: Draw a point




`gl.drawArrays(gl.POINTS, 0, 1);` specifies that we are drawing one vertex (point) starting from vertex id 0.

`gl_Position`, `gl_PointSize` and `gl_FragColor` are built-in variables.

# WebGL Example: Draw a point

**Vertex Shader:**

```
void main(){
    gl_PointSize = 10.0;
    gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
}
```

Note that `gl_Position` has four coordinates: $x, y, z$ and $w$.

We will see later why it is four dimensional.

For now, we will always set $w = 1$ and while doing 2D graphics we set $z = 0$.

# WebGL Example: Draw a triangle

In the last example, the coordinate of the point was hardcoded.

That worked because we just had to draw one point (i.e., one vertex).

We need to be able to set the coordinates from the Javascript code.

This is done by setting up a buffer in the GPU and moving data into that buffer.

# WebGL Example: Draw a triangle

**File:** DrawTriangle.html

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;charset=utf-8">

<title>Draw Triangle</title>

<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="DrawTriangle.js"></script>

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

# WebGL Example: Draw a triangle

```
<script id="fragment-shader" type="x-shader/x-fragment">
precision mediump float;

void main(){
    gl_FragColor = vec4( 1.0, 0.0, 1.0, 1.0 );
}
</script>
</head>
<body>
   <canvas id="gl-canvas" width="512" height="512">
    HTML5 Canvas not supported!
   </canvas>
</body>
</html>
```

# WebGL Example: Draw a triangle

**File:** DrawTriangle.js

```
"use strict";
var gl; // global variable

window.onload = function init() {
    // Set up WebGL
    var canvas = document.getElementById("gl-canvas");
    gl = WebGLUtils.setupWebGL( canvas );
    if(!gl){alert("WebGL setup failed!");}

    // Clear canvas
    gl.clearColor(0.0, 1.0, 0.0, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT);

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

# WebGL Example: Draw a triangle

*x and y coordinates of three vertices.*

```
// Load data into a buffer
var vertices = [ 0.0, 0.5, 0.4, -0.7, 0.6, 0.9];
var vBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);

// Do shader plumbing
var vPosition = gl.getAttribLocation(program, "vPosition");
gl.vertexAttribPointer(vPosition, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(vPosition);

//Draw
gl.drawArrays(gl.TRIANGLES,0,3);
};
```

*create a buffer and make it current*

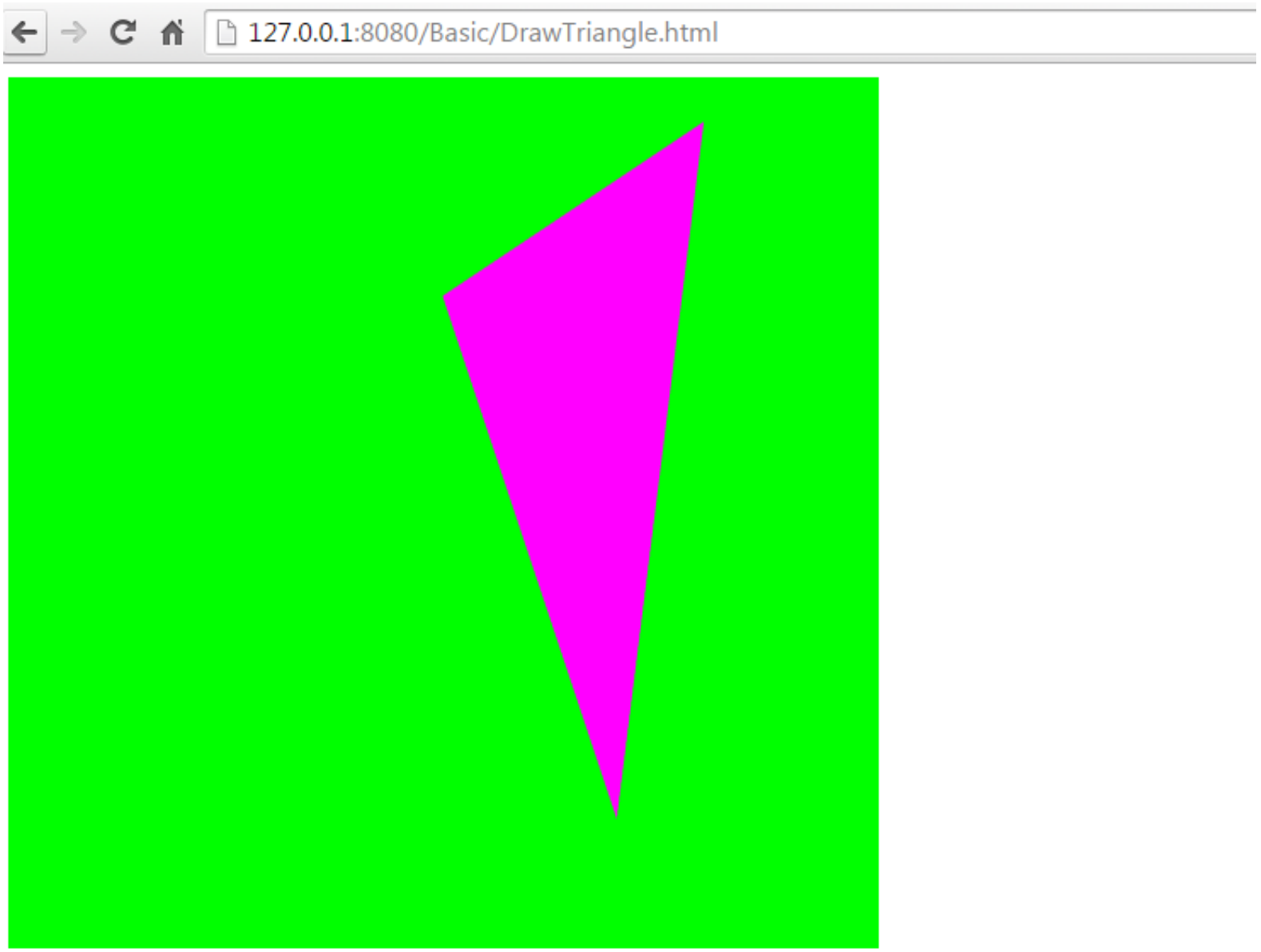
*Transfer data*

*telling the shader how to read the data*

WebGL requires C style arrays. Javascript arrays are objects.

# WebGL Example: Draw a triangle

**Output:**



# drawTriangle

Look at the code in `drawTriangle.html` and `drawTriangle.js`

**Questions:**

Which part of the code runs in the CPU? Which part runs in the GPU?

What is `gl`?

We have two coordinates for each vertex. How does the GPU know this?

Why do we use `new Float32Array(vertices)` in `gl.bufferData(...)`?

Where are we setting the color of the triangle?

What will happen if we change the 0.6 in the `vertices` array to 1.6?

`vPosition` is of type `vec4` in the vertex shader but we are passing only two coordinates. Where are the other two set?

Why is the last argument of `gl.drawArrays(...)` 3 and not 1?

# Coding Practice

**Tasks:**

Draw an equilateral triangle whose centroid is at $(0, 0)$.

Draw a square with side length 0.4 whose center is at $(0, 0)$.

Change the vertex shader so that the square moves to the right by 0.3.

Change the color of the square to blue.

Draw a triangle and a square but put their data in different buffers in the GPU.

Change the vertex shader so that the entire picture is 1.5 times the original size and is rotated by 30 degrees in the counter-clockwise direction around the origin.