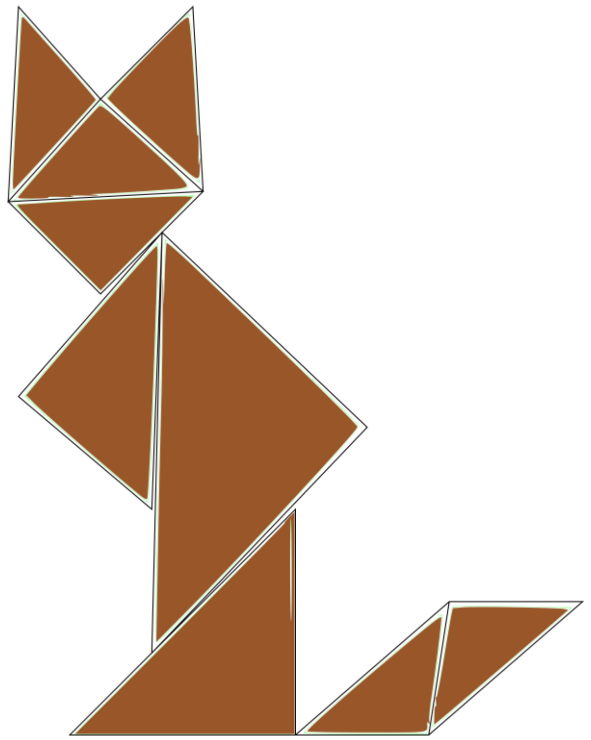


# GETTING STARTED WITH PROGRAMMING IN WEBGL

Dr. Bhupendra Singh

Assistant Professor

UPES Dehradun

## TOPICS COVERED

## INTRODUCTION

- WebGL derives its roots from the OpenGL graphics library which was designed to create cross platform desktop applications.
- First version of WebGL was released in the year 2011 and the second on 2017 by the Khronos group.
- With the help of WebGL a modern graphics application can be developed for the Web without the need of any other external libraries or installing any other software.

## INTRODUCTION(CONT.)

- Khronos group defines the specifications for WebGL. It’s a consortium of more than 170 companies to support, develop and standardize open, non-profit 3D graphics application development.
- OpenGL, OpenGL ES, Open CL, OpenXR, Vulkan, and many other specifications are some the offerings by the group.

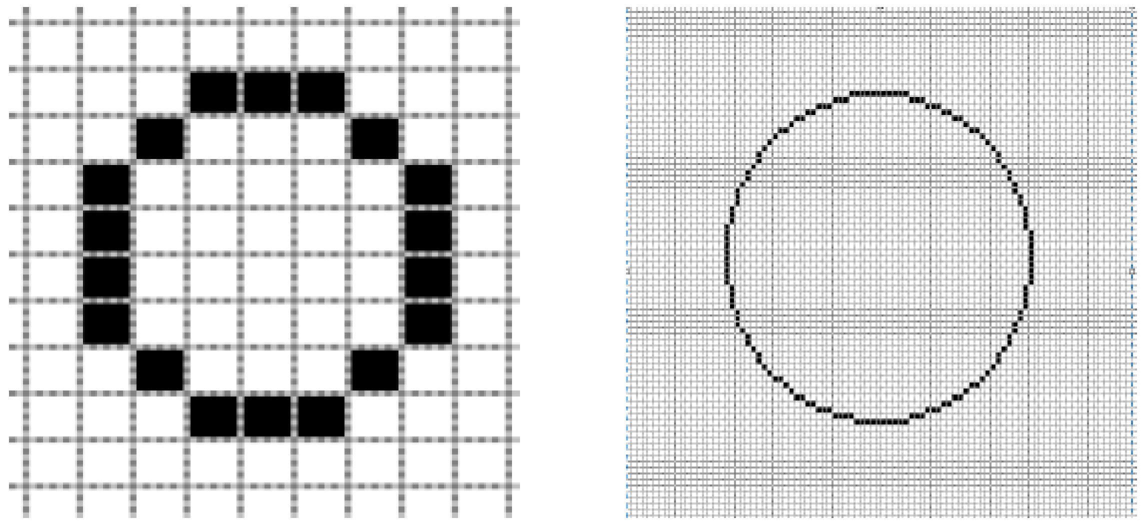
## MINIMAL HTML CODE TO START

```
<html >
    <head >
        <title >WebGL - Program Title </ title >
    </ head >
    <body onload =" WebGL ();">
        <canvas id=" canvas " width =" 1200 " height =" 600">
                 Your browser does not support HTML5
         </ canvas >
        <br/>
     <script src=" JS_filename .js"></ script >
   </ body >
</ html >
```

# MINIMAL JS CODE TO START

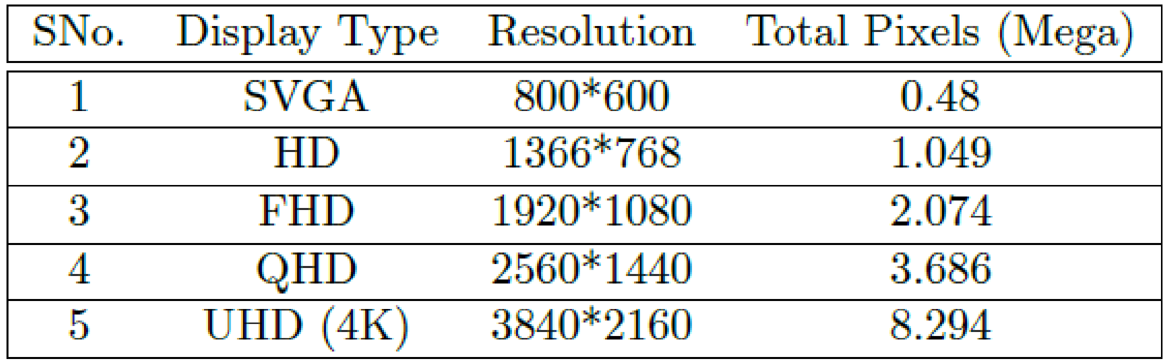
```
var WebGL = function () {
   var gl, shaderProgram;
   gl = initializeWebGL(gl);
}
function initializeWebGL(gl){
   var canvas = document.getElementById('canvas');
   canvas.width = window.innerWidth;;
   canvas.height = window.innerHeight;;
   gl = canvas.getContext('webgl2');
   if(!gl){
      alert('Your browser does not support WebGL');
      return;
   }
   return gl;
}
```

## MODERN COMPUTER GRAPHICS RESOLUTIONS



(a) Circle at low resolution (b) Circle at high resolution

Figure 1.1: Shape of the circle at various screen resolutions.



| SNo. | Display Type | Resolution | Total Pixels (Mega) |
|---|---|---|---|
| 1 | SVGA | 800*600 | 0.48 |
| 2 | HD | 1366*768 | 1.049 |
| 3 | FHD | 1920*1080 | 2.074 |
| 4 | QHD | 2560*1440 | 3.686 |
| 5 | UHD (4K) | 3840*2160 | 8.294 |

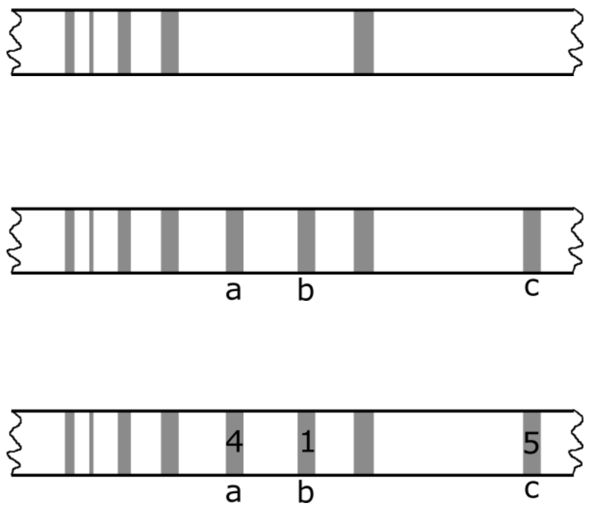
Table 1.1: Common Display Resolutions

The pixel resolution directly dictates the quality of the image or graphics rendered on the screen.

# MEMORY USAGE BY A PROGRAM

- Lets understand how data defined by the program is stored in memory.

```
Int main()
{
        int a, b, c;
        a= 4;
        b = 1;
        c= a + b;
}
```



Initial memory blocks

Memory blocks after variable declaration

Memory blocks after variable assignments

## PROGRAMMING WITHOUT UTILIZING GPU

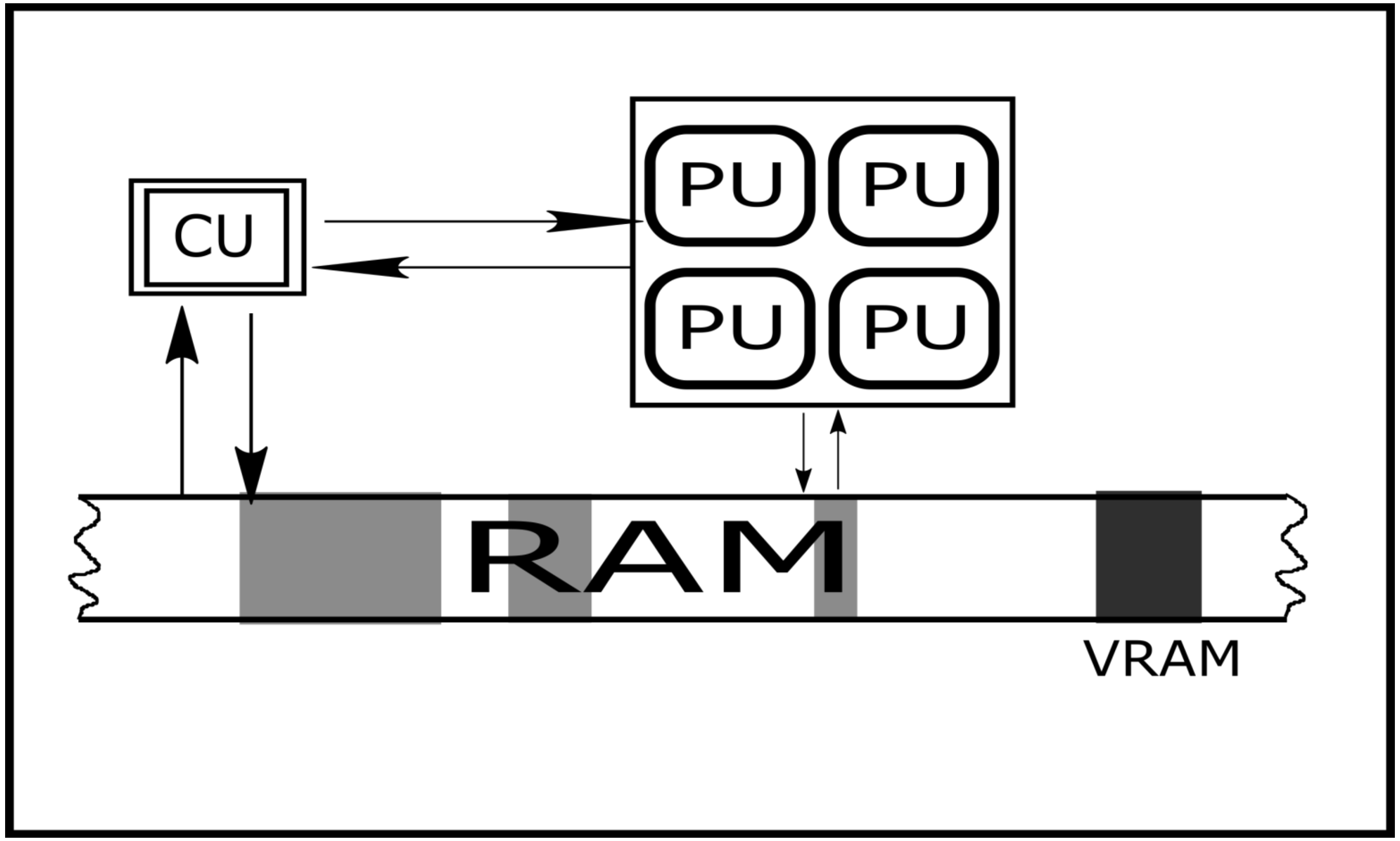
The concepts of memory allocation and data storage play their role when we talk about coding in WebGL.

CU is control unit.

PU is processing unit.

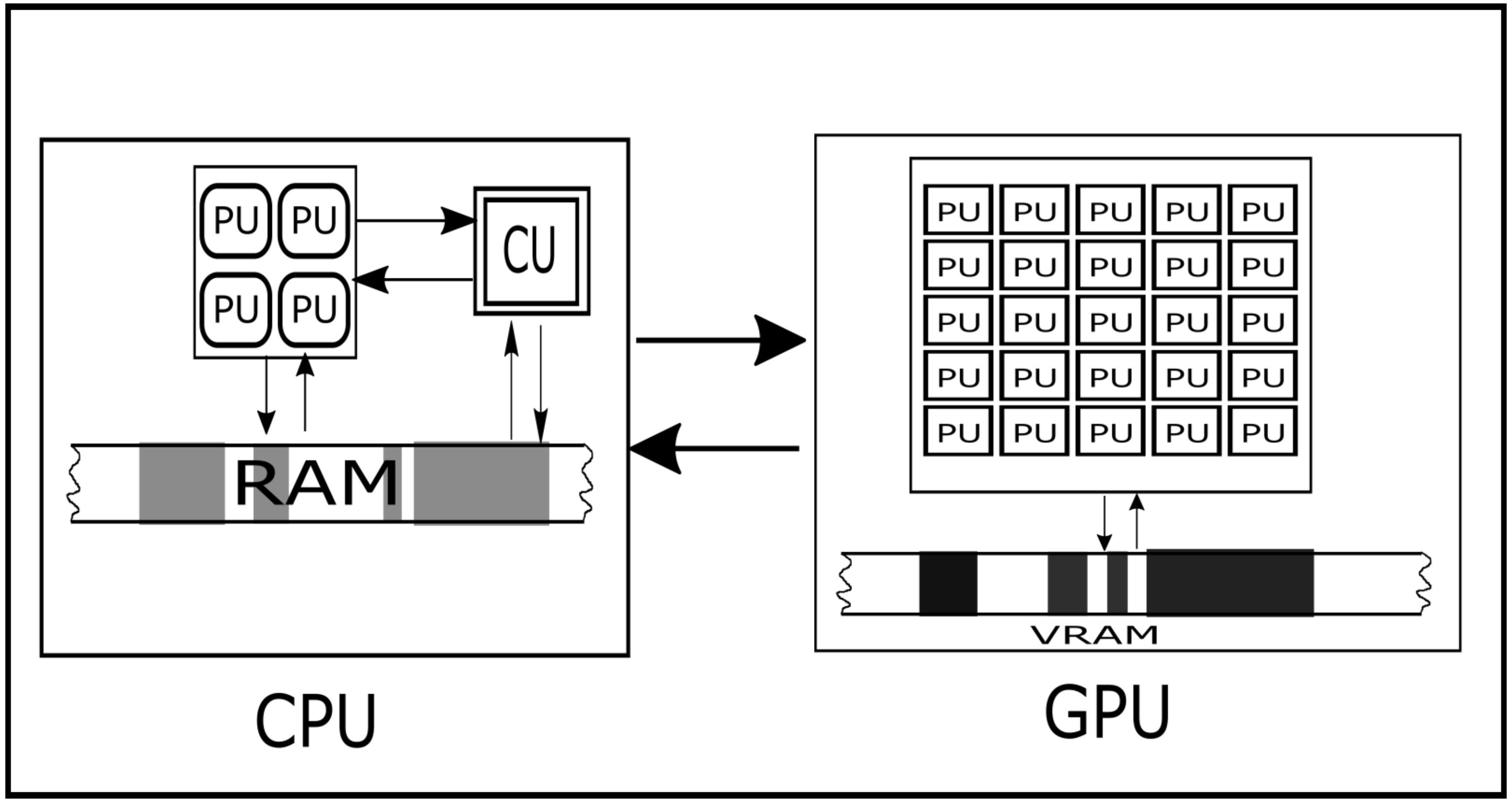
RAM is normal memory.

VRAM is for video memory used for graphics.



## GPU BASED PROGRAMMING

It is desired to not only execute the code on GPU but also to load the data there.



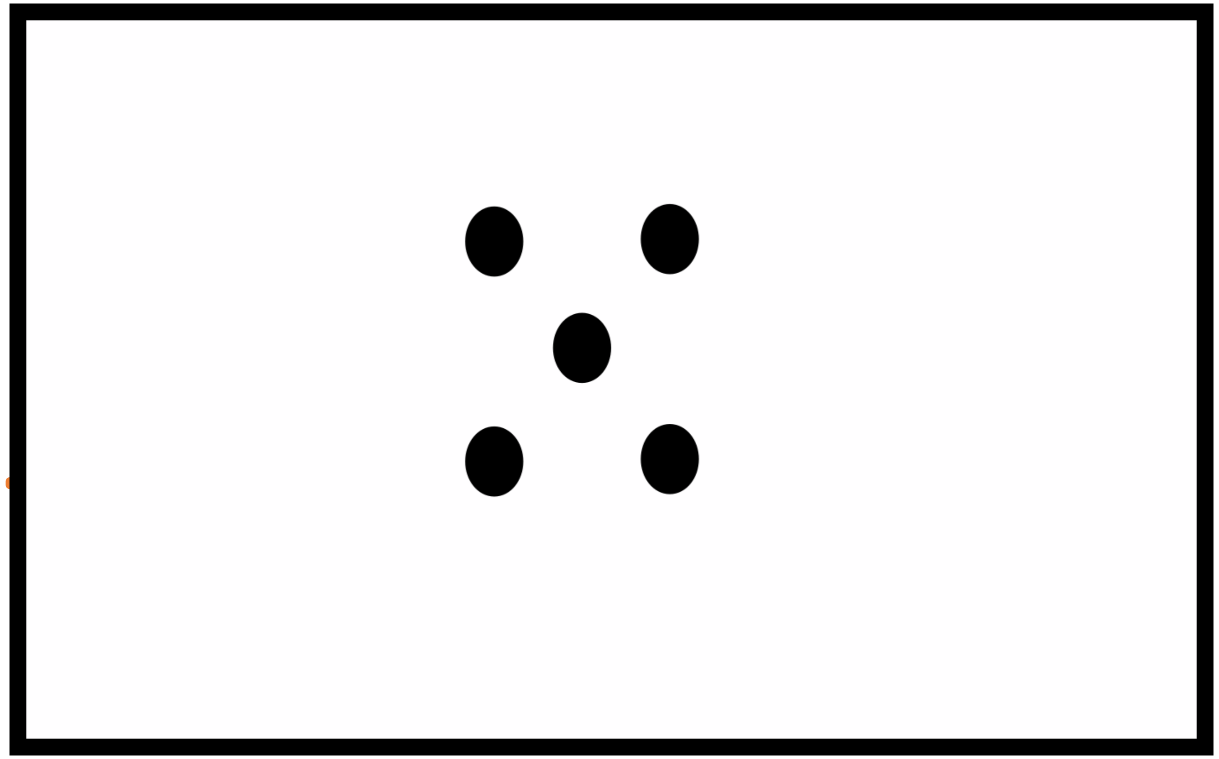
## THREE BLOCKS OF GRAPHICS PROCESSING

Vertex Shader

Rasterizer

Fragment Shader

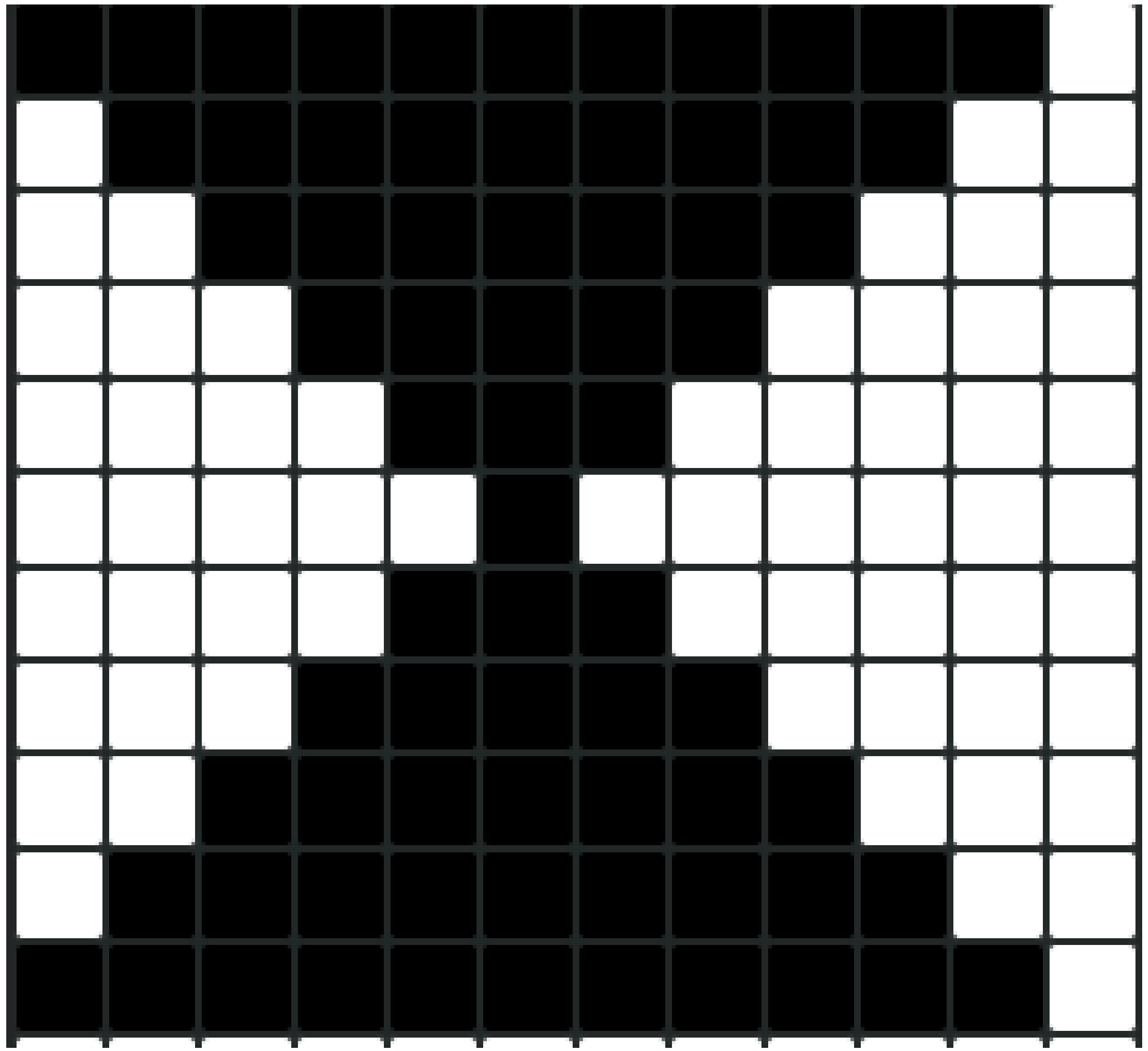
# VERTEX SHADER



- Maps the pixels defined by programmer on the actual computer screen.
- For each defined vertices, an equal number of vertex shaders are called which runs in parallel on the GPU's PU.
- For a two-triangle geometry with 5 vertices, we can expect 5 copies of vertex shader running in parallel to generate the mapping for 5 vertices on the computer screen.

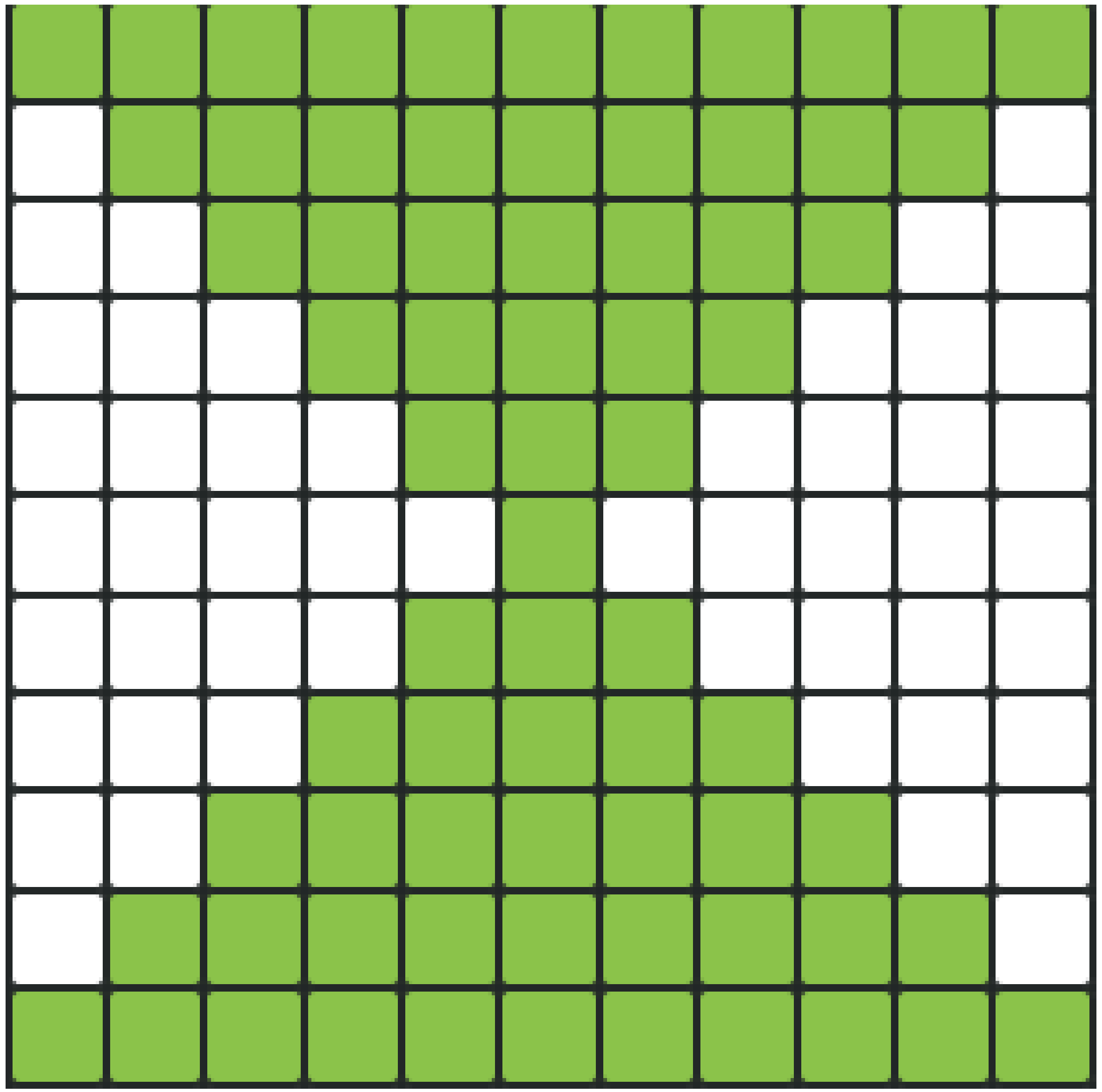
## RASTERIZER

Does the task of identifying all the pixels which are in between the vertices (which are generated by the VS).

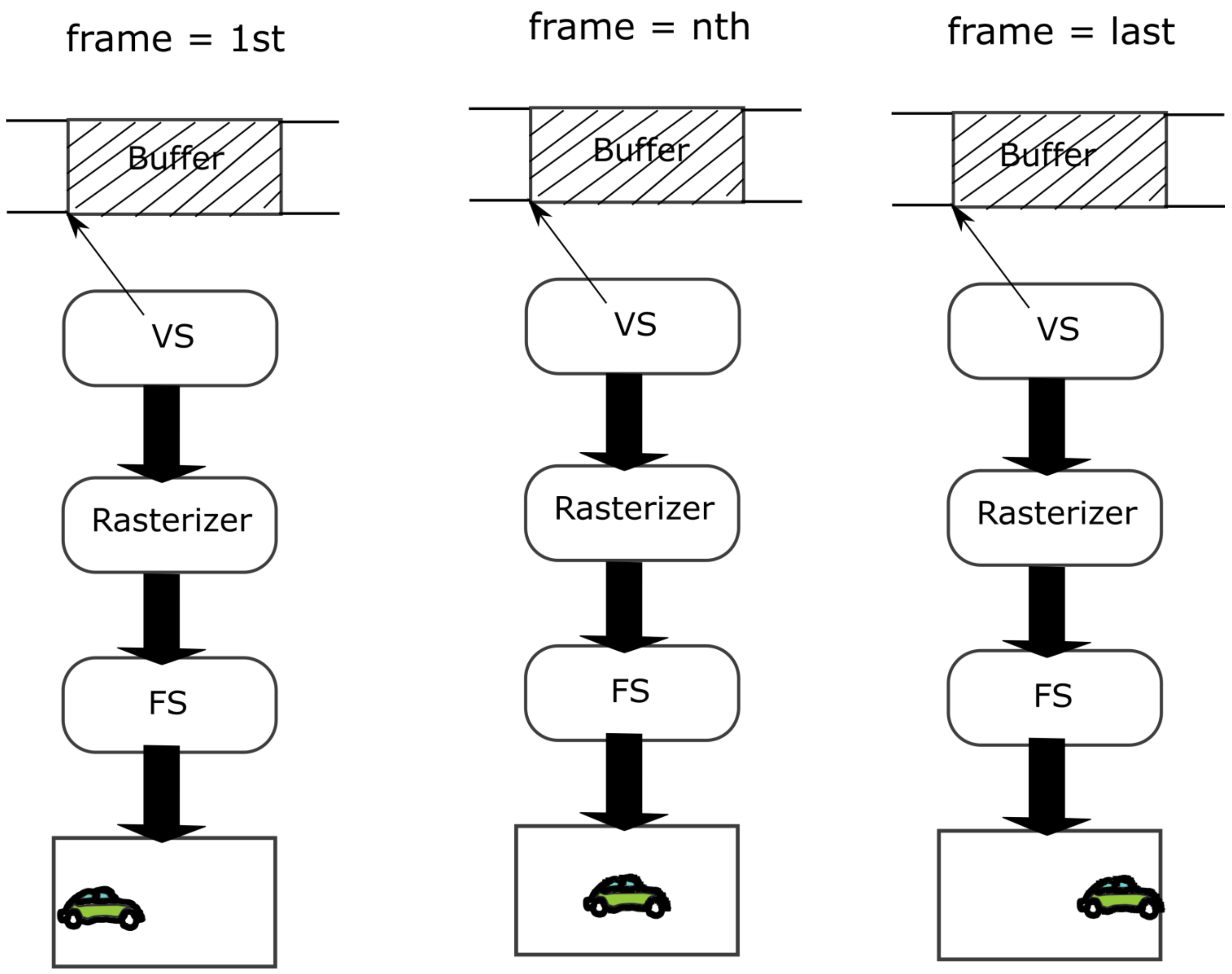


## FRAGMENT SHADER

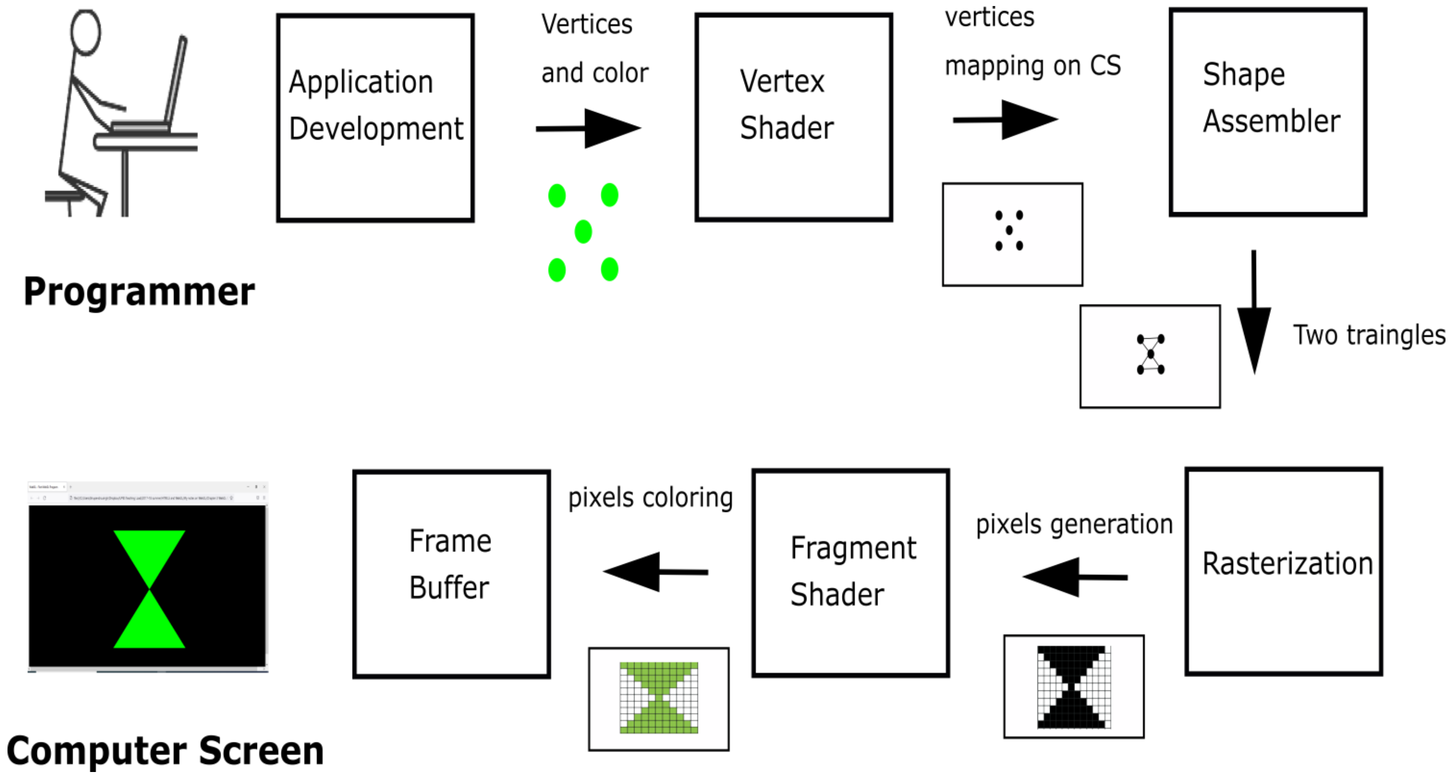
For each pixel identified by the rasterizer, it starts coloring them.



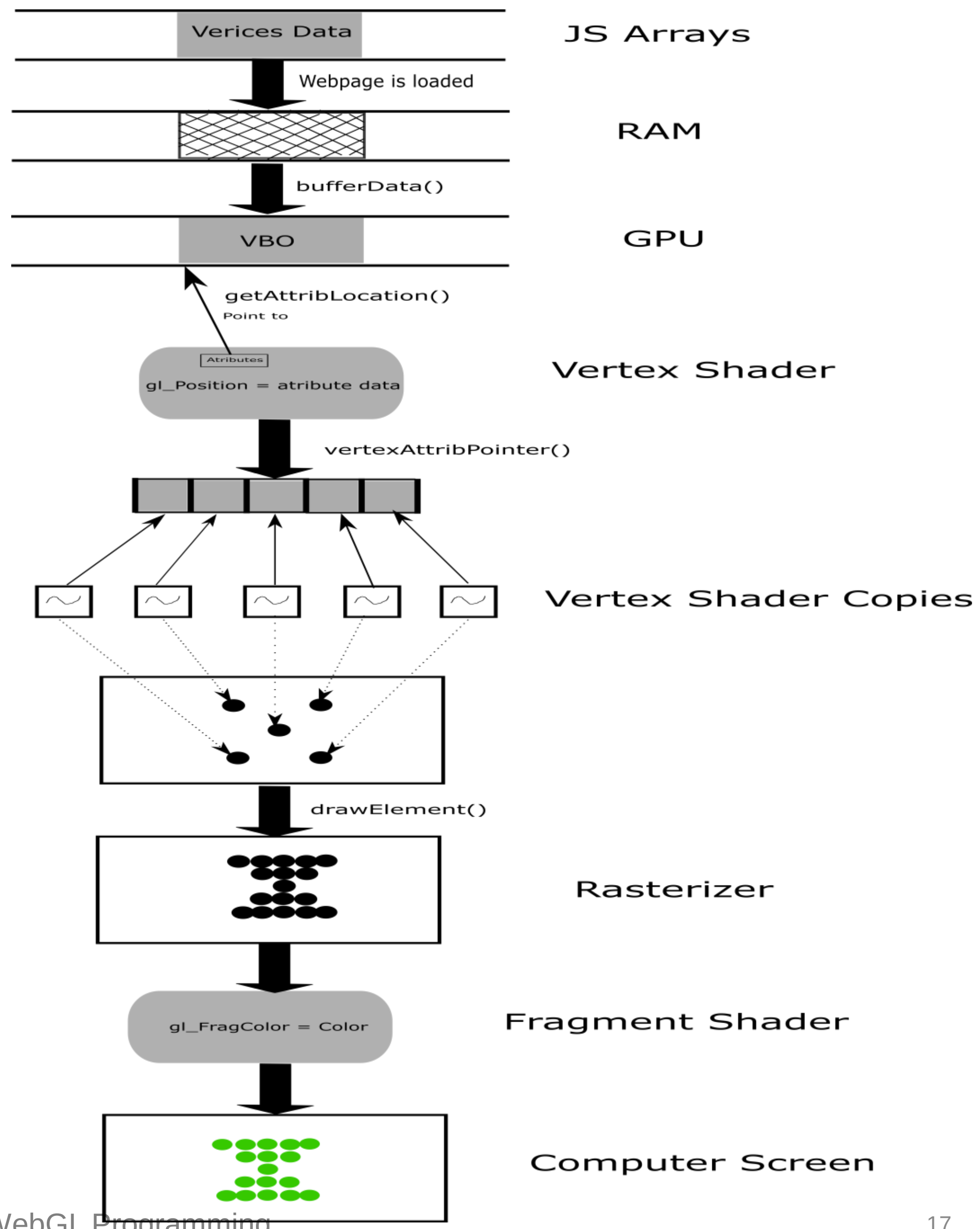
## WORKING OF THE TRIO.



## WEBGL PROGRAMMING ILLUSTRATION



# DATA FLOW IN WEBGL PROGRAM



## 18 STEPS OF WEBGL PROGRAMMING

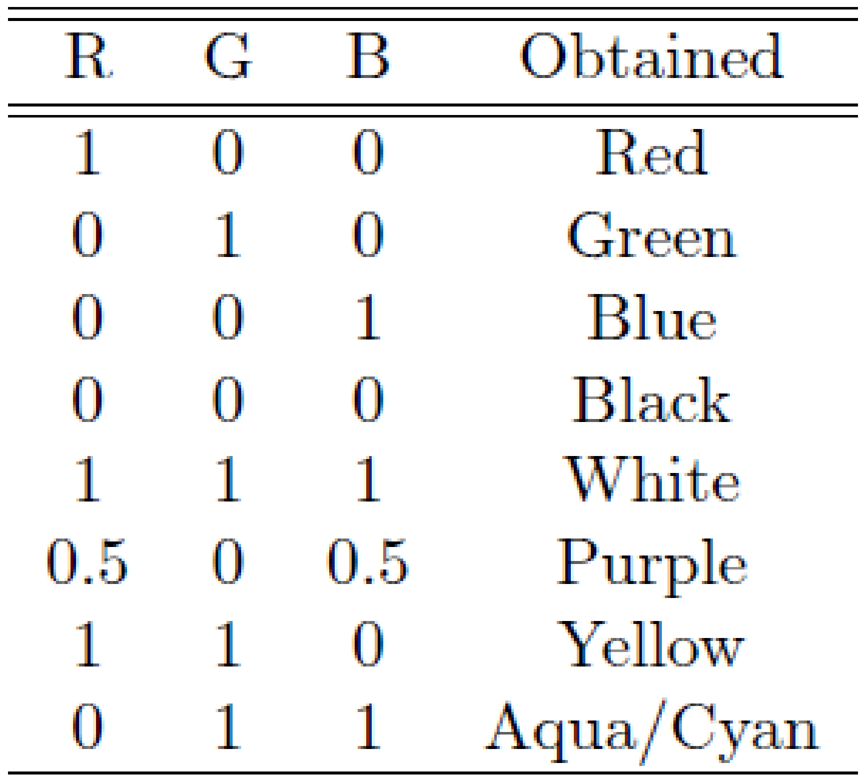
- Even a simplistic WebGL program requires around 100 lines of code.
- But creating complex geometry requires only few additional steps.
- We can group them in similar logical blocks.
- Once these 18 steps are mastered, programming in WebGL is fun.

## 18 STEPS OF WEBGL PROGRAMMING (STEP 1)

Step 1 (Set background color): First, specify the color with the help of Quadlet(R, G, B, Alpha) and then clear the buffer related to the background.

```
gl. clearColor (0, 0, 0, 1.0) ;
gl. clear (gl. COLOR_BUFFER_BIT );
// Note : The default background color in WebGL is
white .
```

## COLOR COMBINATIONS



| R | G | B | Obtained |
|---|---|---|---|
| 1 | 0 | 0 | Red |
| 0 | 1 | 0 | Green |
| 0 | 0 | 1 | Blue |
| 0 | 0 | 0 | Black |
| 1 | 1 | 1 | White |
| 0.5 | 0 | 0.5 | Purple |
| 1 | 1 | 0 | Yellow |
| 0 | 1 | 1 | Aqua/Cyan |

Table 1.2: RGB triplet for various colors

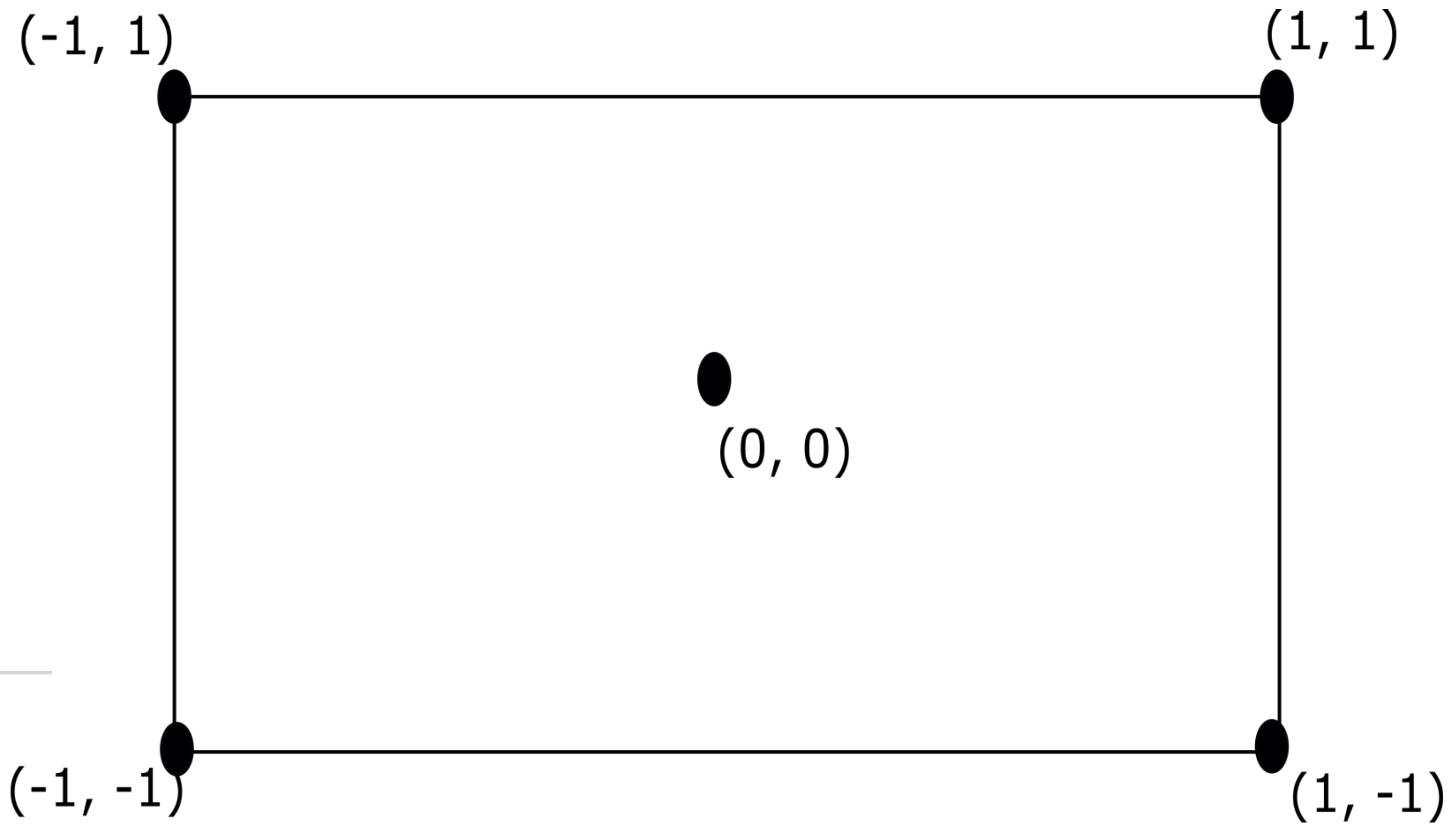
## 18 STEPS OF WEBGL PROGRAMMING (STEP 2)

Step 2 (Specify vertices data):

Specify the coordinates (X, Y, Z) of the geometry and other information if any, related to each vertex in the form of a JS array.

```
var verticesDataArrayJS =
[ // X,   Y,    Z
  -0.3,  0.7 , 0,      //A
  0.3 ,   0.7 , 0,      //B
  0,       0,    0,      //C
  0.3 ,  -0.7,  0,     //D
  -0.3, -0.7,  0      //E
]; .
```

## WEBGL DEFAULT COORDINATE SYSTEM



## 18 STEPS OF WEBGL PROGRAMMING (STEP 3)

Step 3 (Specify how to connect the points): Specify the order with which the coordinates defined in Step 2 will be connected.

```
var IndicesArrayJS =
[
    0, 1, 2,    // ABC
    2, 3, 4    // CDE
];
```

## 18 STEPS OF WEBGL PROGRAMMING (STEP 4)

Step 4 (Create GPU memory buffer): for holding vertices data.

```
var rectVBO = gl. createBuffer ();
gl. bindBuffer (gl. ARRAY_BUFFER , rectVBO );
```

## 18 STEPS OF WEBGL PROGRAMMING (STEP 5)

Step 5 (Pass the vertices data to the buffer created previously).

```
gl.bufferData (gl. ARRAY_BUFFER ,
                     new Float32Array ( verticesDataArrayJS ),
                     gl. STATIC_DRAW );
```

## 18 STEPS OF WEBGL PROGRAMMING (STEP 6)

Step 6 (Pass the indices data to GPU buffer): repeat steps 4 and 5 for the indices data.

```
var rectIBO = gl. createBuffer ();
gl. bindBuffer (gl. ELEMENT_ARRAY_BUFFER , rectIBO );
gl. bufferData (gl. ELEMENT_ARRAY_BUFFER ,
                    new Uint16Array ( IndicesArrayJS ),
gl. STATIC_DRAW );
```

## SHADER SIDE PROGRAMMING

- THE SEVEN STEPS OF SHADER SIDE CODING (VS AND FS) ARE SEPARATED RIGHT NOW AND DISCUSSED TOWARDS THE END.
- FOR THE CONVENIENCE OF THE CONNECTIVITY OF OUR DISCUSSION. SHADERPROGRAM = GETSHADERPROGRAM (GL);

## 18 STEPS OF WEBGL PROGRAMMING (STEP 14)

Step 14 (Use the shader program):

```
gl. useProgram ( shaderProgram );
```

## 18 STEPS OF WEBGL PROGRAMMING (STEP 15)

Step 15 (Get access to GPU for geometry coordinates):

Get the pointer to the geometry coordinates defined in vertex shader through the shader program.

```
var positionAttribLocation = gl.getAttribLocation (shaderProgram , ’
                                                  geometryCoordinatesGPU ’);
```

## 18 STEPS OF WEBGL PROGRAMMING (STEP 16)

Step 16 (Enable Vertex Attribute Array):

```
gl.enableVertexAttribArray( positionAttribLocation );
```

## 18 STEPS OF WEBGL PROGRAMMING (STEP 17)

Step 17 (Buffer data definition):

Define how the data on the GPU buffer (or VBO) is arranged. So that the pointer defined in Step 8 can access the data from it.

```
gl.vertexAttribPointer (
                    positionAttribLocation , // Attribute location
                    3, // Number of elements per attribute
                    gl.FLOAT , // Type of elements
                    gl.FALSE , // Normalization
                    3 * Float32Array . BYTES_PER_ELEMENT ,
                    // Size of an individual vertex
                    0 // Offset from the beginning of a single vertex to
                    this attribute
                    );
```

## 18 STEPS OF WEBGL PROGRAMMING (STEP 18)

Step 18 (Draw the geometry): Issue the draw command to generate the geometry as defined by the indices and the type of primitive to create.

```
gl.drawElements( gl.TRIANGLES ,
                 IndicesArrayJS . length , gl. UNSIGNED_SHORT,
                 0
                );
```

## 18 STEPS OF WEBGL PROGRAMMING (STEP 7) SHADER SIDE PROGRAMMING

Step 6 (Pass the indices data to GPU buffer): repeat steps 4 and 5 for the indices data.

```
var rectIBO = gl. createBuffer ();
gl. bindBuffer(gl. ELEMENT_ARRAY_BUFFER , rectIBO );
gl. bufferData(gl. ELEMENT_ARRAY_BUFFER ,
                        new Uint16Array ( IndicesArrayJS ),
                        gl. STATIC_DRAW
                       );
```

## SEVEN STEPS OF SHADER SIDE PROGRAMMING

```
shaderProgram = getShaderProgram (gl);
```

## 18 STEPS OF WEBGL PROGRAMMING (STEP 7) SHADER SIDE PROGRAMMING

Step 7 (Define vertex shader text): Define the code of the vertex shader in the form of JS text.

```
var vertexShaderText = ‘# version 300 es
                                # pragma vscode_glsllint_stage : vert
                                in vec3 geometryCoordinatesGPU ;
                                void main ()
                                {
                                     gl_Position = vec4 ( geometryCoordinatesGPU , 1.0)
;
                                }‘;
```

## 18 STEPS OF WEBGL PROGRAMMING (STEP 8) SHADER SIDE PROGRAMMING

Step 8 (Create actual vertex shader):

```
var vertexShader = gl. createShader (gl. VERTEX_SHADER );
gl. shaderSource ( vertexShader , vertexShaderText );
```

## 18 STEPS OF WEBGL PROGRAMMING (STEP 9) SHADER SIDE PROGRAMMING

Step 9 (Compile vertex shader):

```
gl. compileShader( vertexShader );
if (! gl. getShaderParameter ( vertexShader , gl. COMPILE_STATUS )) {
    console . error ('ERROR compiling vertex shader !');
    return ;
}
```

## 18 STEPS OF WEBGL PROGRAMMING (STEP 10) SHADER SIDE PROGRAMMING

Step 10: Repeat the above 3 steps for the fragment shader.

```
var fragmentShaderText = ‘# version 300 es
                                        # pragma vscode_glsllint_stage : frag
                                         precision mediump float ;
                                         out vec4 fragColor ;
                                         void main ()
                                         {
                                                 fragColor = vec4 (0.0 , 1.0 , 0.0 , 1.0)
;
                                         }‘;
```

## 18 STEPS OF WEBGL PROGRAMMING (STEP 10) SHADER SIDE PROGRAMMING

Step 10: Repeat the above 3 steps for the fragment shader.

```
var fragmentShader = gl. createShader (gl. FRAGMENT_SHADER );
gl. shaderSource ( fragmentShader , fragmentShaderText );
gl. compileShader ( fragmentShader );
if (! gl. getShaderParameter ( fragmentShader , gl. COMPILE_STATUS )) {
    console . error (’ERROR compiling fragment shader !’);
    return ;
}
```

## 18 STEPS OF WEBGL PROGRAMMING (STEP 11) SHADER SIDE PROGRAMMING

Step 11 (Shader program): With the compiled vertex and fragment shader, create the shader program.

```
var shaderProgram = gl.createProgram ();
gl.attachShader( shaderProgram , vertexShader );
gl.attachShader( shaderProgram , fragmentShader );
```

## 18 STEPS OF WEBGL PROGRAMMING (STEP 12) SHADER SIDE PROGRAMMING

Step 12 (Link shader program):

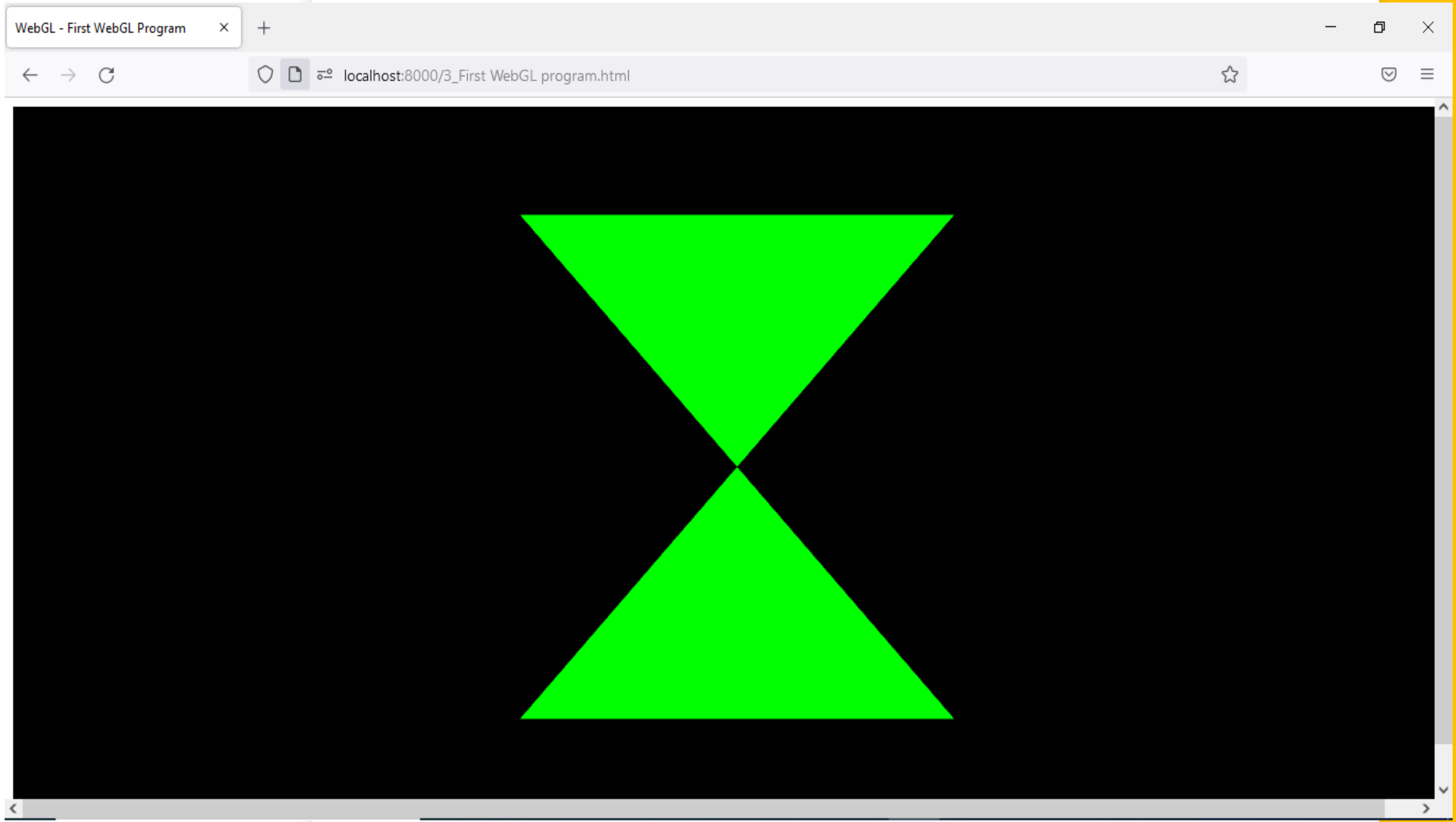
```
gl.linkProgram ( shaderProgram );
if (! gl.getProgramParameter ( shaderProgram , gl.LINK_STATUS )) {
    console.error (’ERROR linking program !’);
    return ;
}
```

## 18 STEPS OF WEBGL PROGRAMMING (STEP 13) SHADER SIDE PROGRAMMING

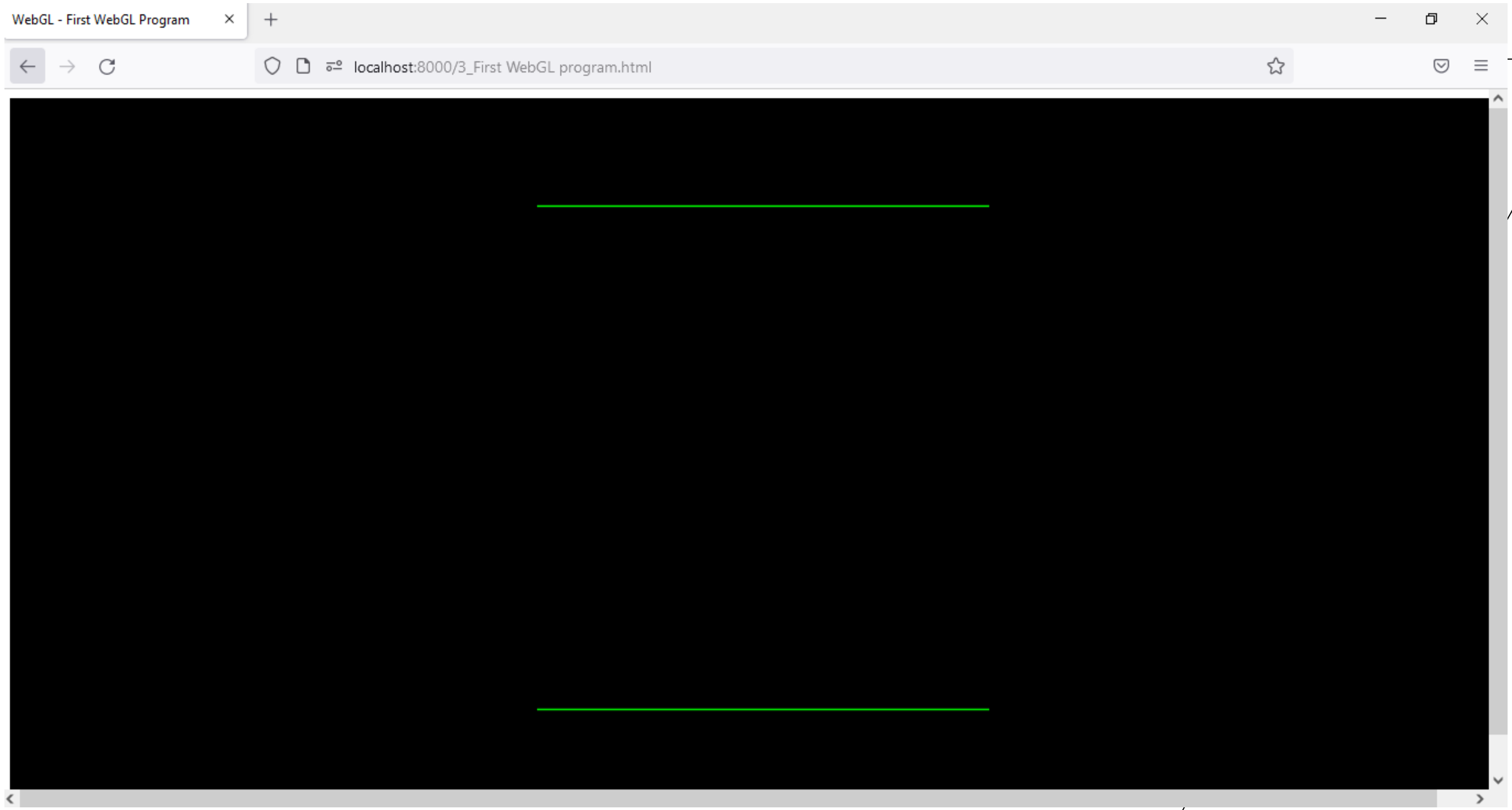
Step 13 (Validate Shader program):

```
gl.validateProgram( shaderProgram );
if( !gl.getProgramParameter( shaderProgram, gl.VALIDATE_STATUS )) {
    console.error (’ERROR validating program !’);
    return ;
}
```

# FINAL OUTPUT

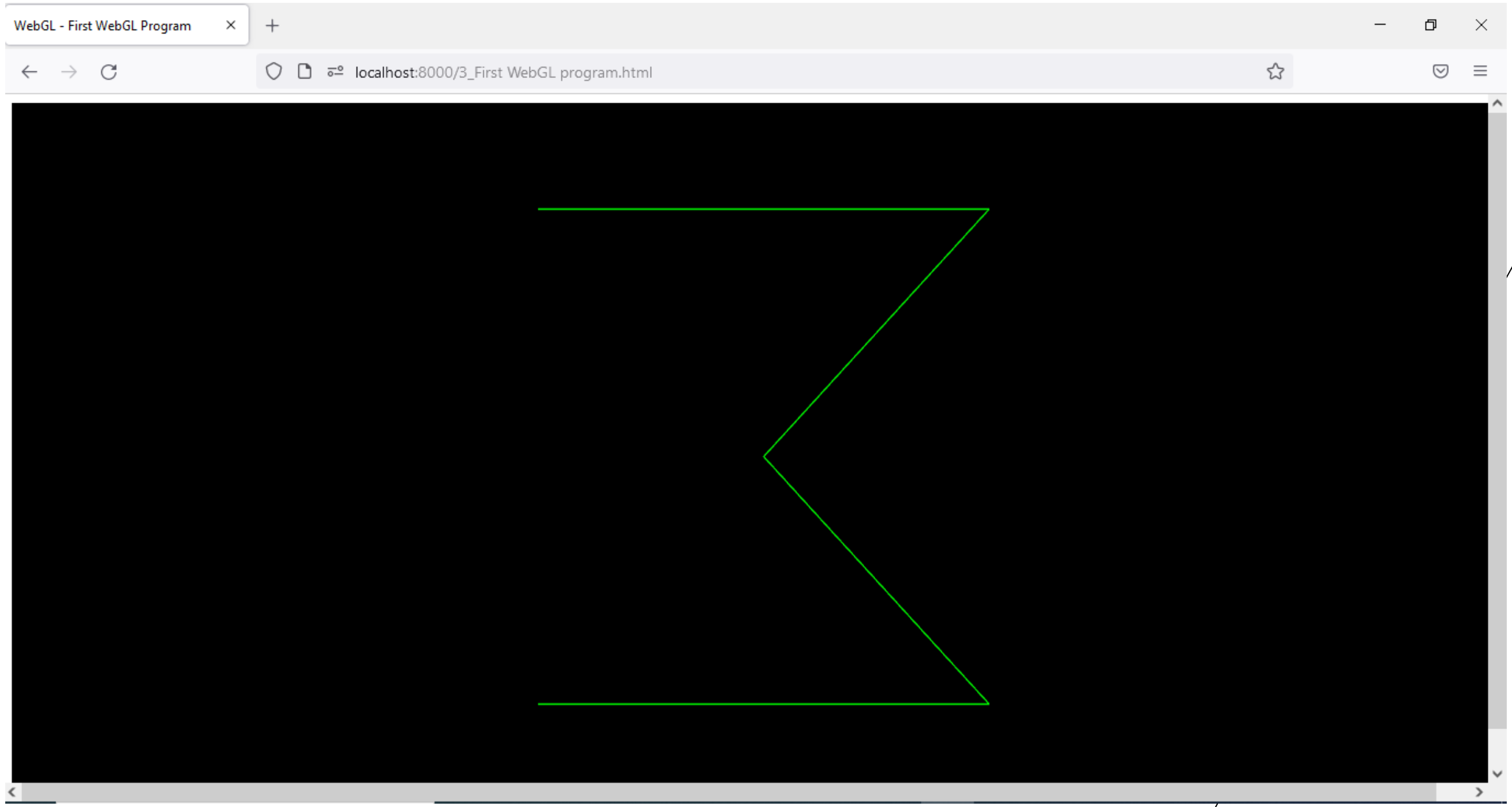

WebGL - First WebGL Program
localhost:8000/3_First WebGL program.html

## OUTPUT FOR DIFFERENT PRIMITIVE TYPE


WebGL - First WebGL Program
localhost:8000/3_First WebGL program.html

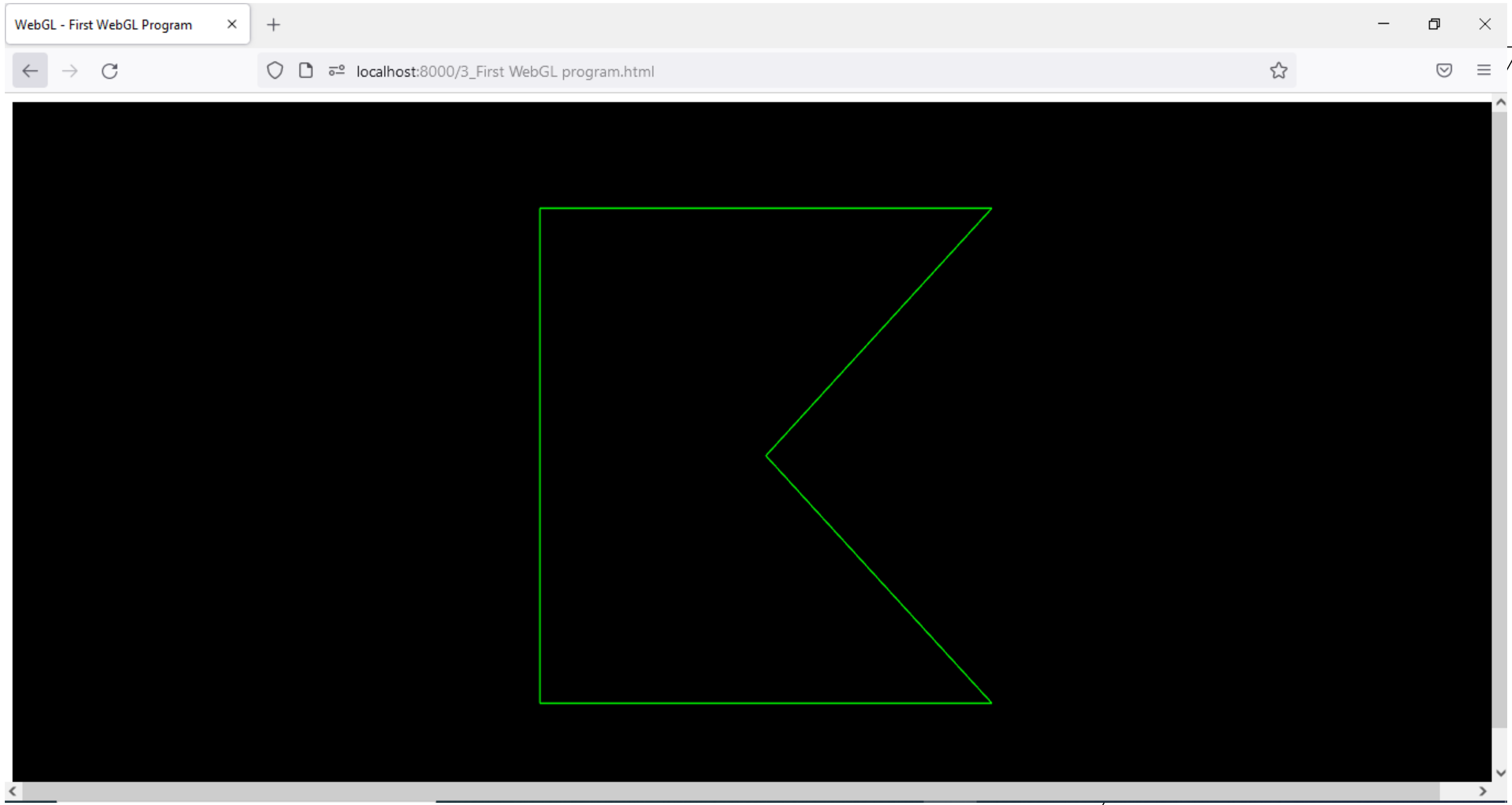
**GL.LINES**

## OUTPUT FOR DIFFERENT PRIMITIVE TYPE

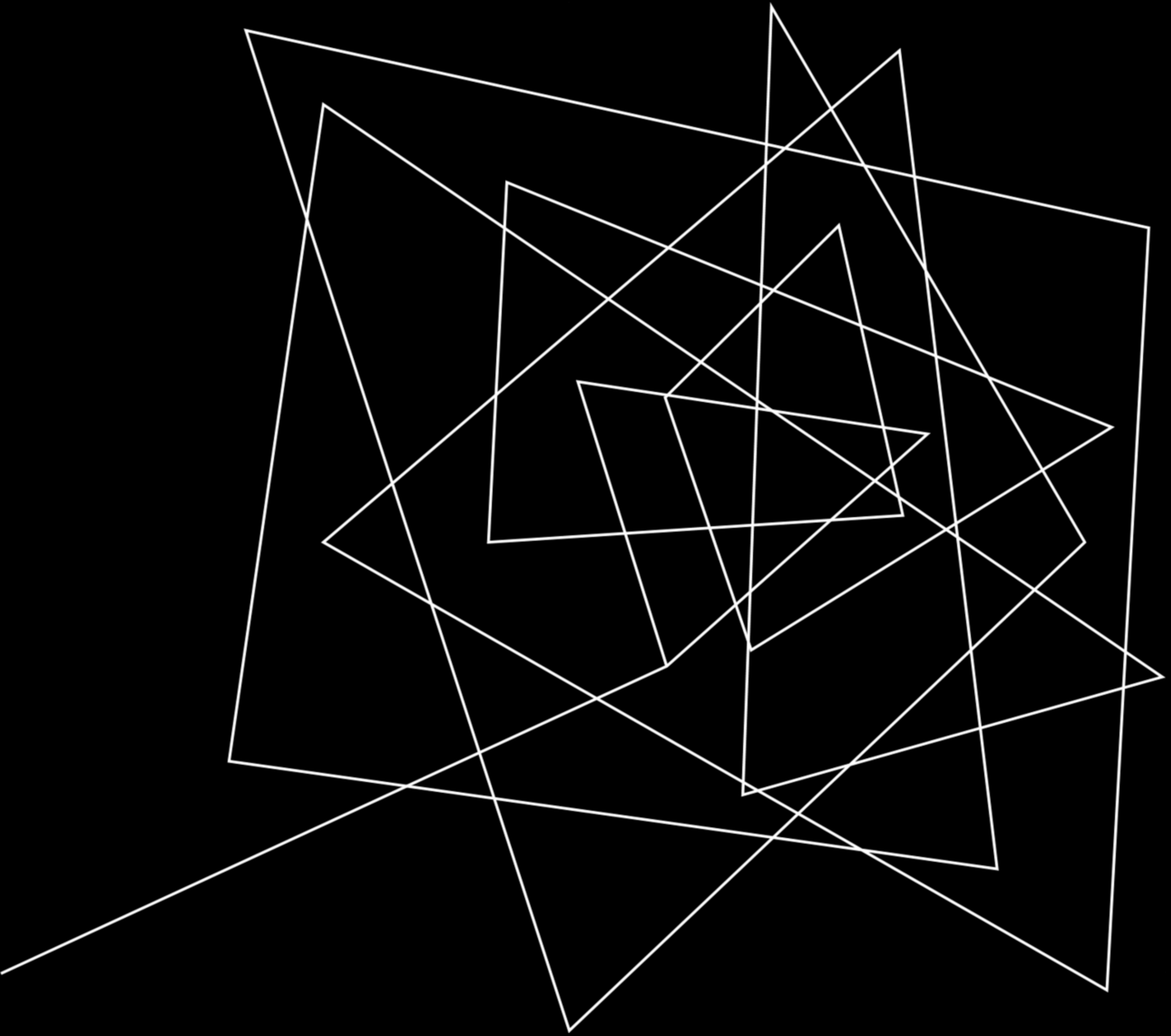

WebGL - First WebGL Program
localhost:8000/3_First WebGL program.html

**GL.LINES_STRIP**

## OUTPUT FOR DIFFERENT PRIMITIVE TYPE

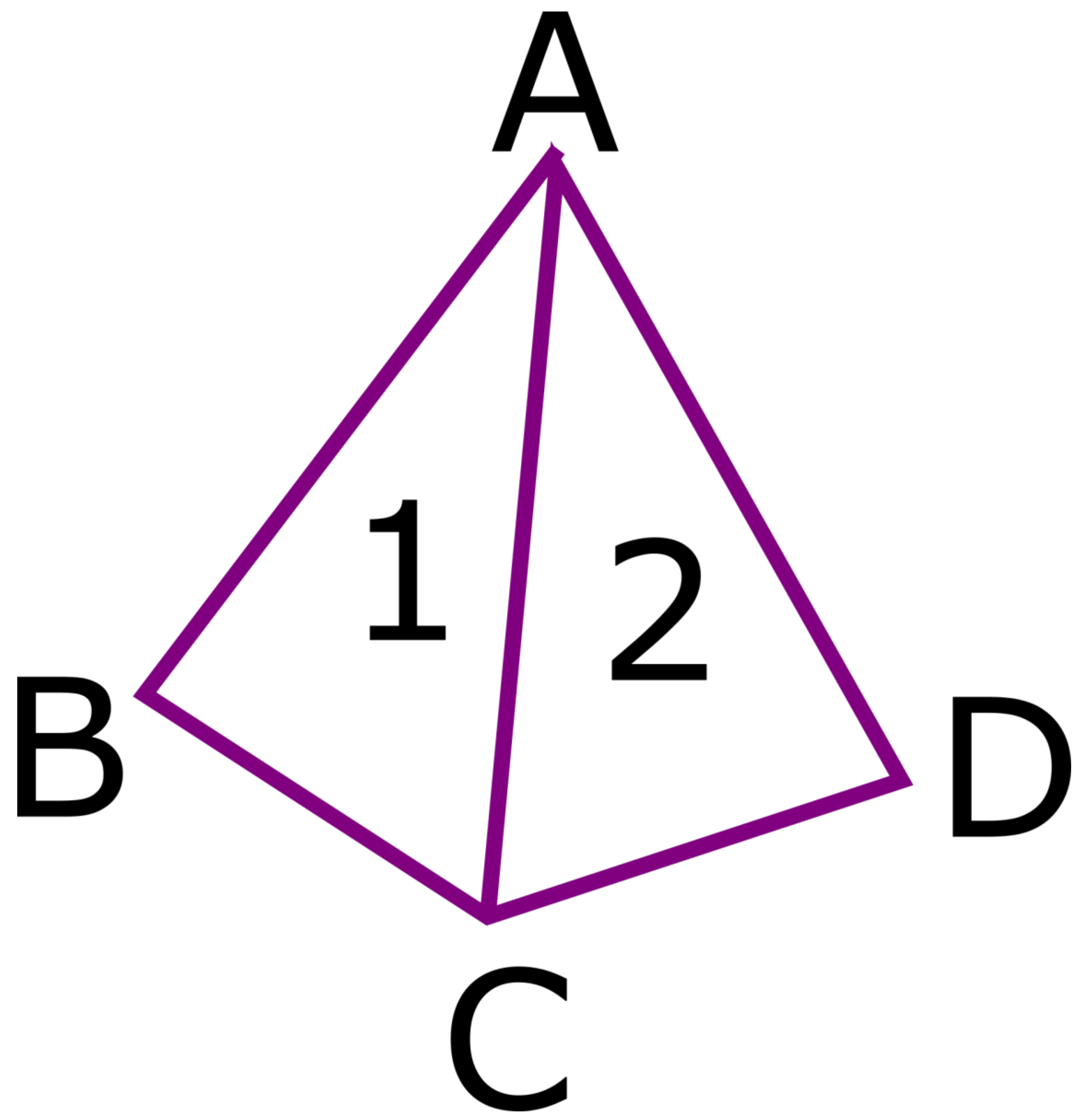

WebGL - First WebGL Program
localhost:8000/3_First WebGL program.html

GL.LINES_LOOP



## DRAWING MORE GEOMETRIES

## LETS DRAW PYRAMID



## DRAWING A PYRAMID

Lets first specify the purple color for the pyramid in Fragment Shader at Step 10.

```
var fragmentShaderText = ‘# version 300 es
                          # pragma vscode_glsllint_stage : frag
                          precision mediump float ;
                          out vec4 fragColor ;
                          void main ()
                          {
                             fragColor = vec4 (0.5 , 0.0 , 0.5 , 1.0);
                          }‘;
```

## DRAWING A PYRAMID

In Step 2, specify the coordinates of the required pyramid shape.

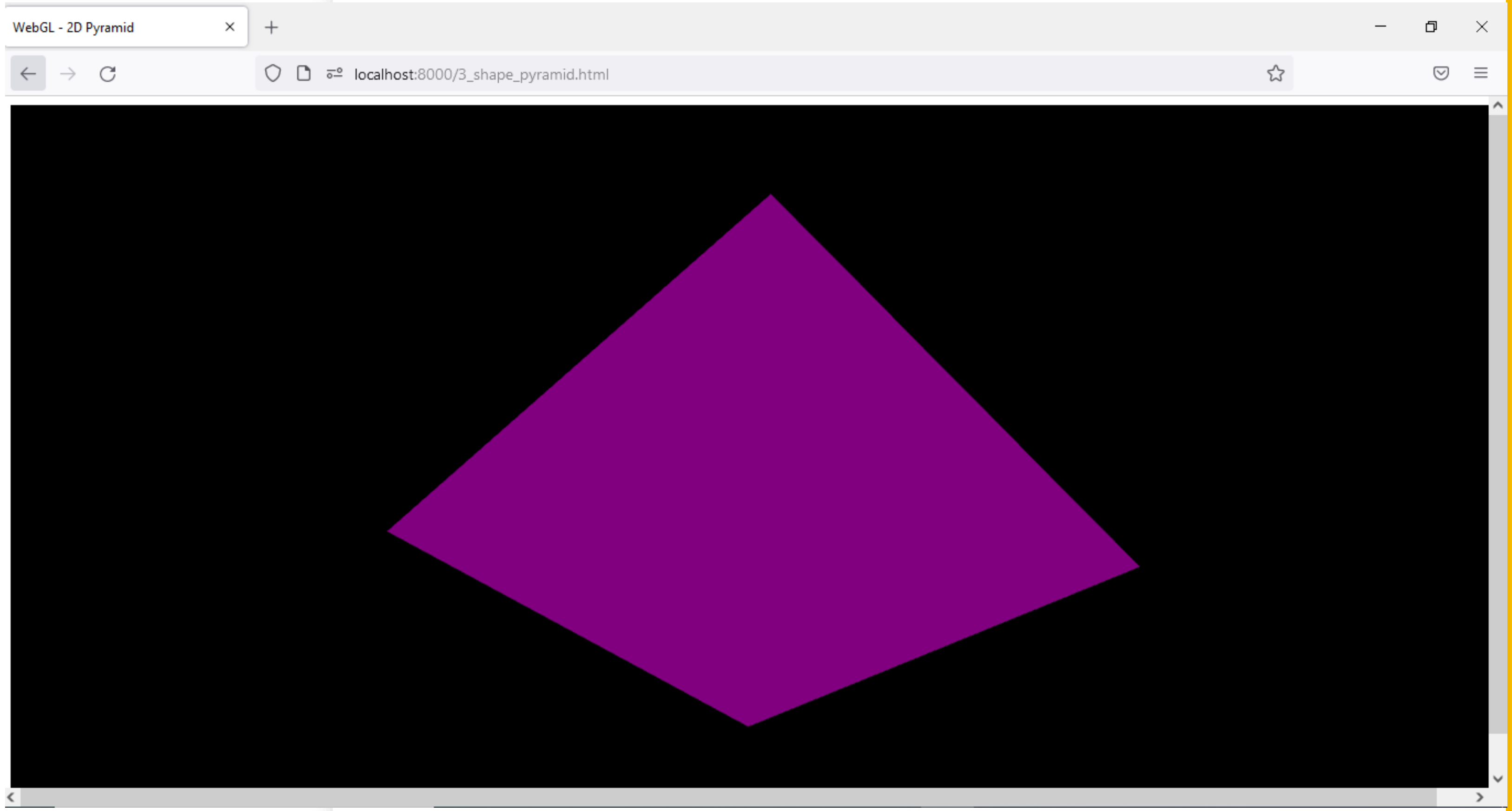
```
var verticesDataArrayJS =
                         [  // X,      Y,        Z
                              0.01 ,  0.75 ,   0,    //A
                             -0.5,  -0.2,      0,    //B
                             -0.02, -0.75 ,   0,    //C
                              0.5 ,   -0.3,     0     //D
                         ];
```

## DRAWING A PYRAMID

In Step 3: Define the indexes of the vertices which create the Triangle 1 and Triangle 2.

```
var verticesDataArrayJS =
                                          [
                                             //A -0, B -1, C -2, D -3
                                             0, 2, 1, // Triangle 1: ACB
                                             0, 3, 2 // Triangle 2: ADC
                                         ];
```

## FINAL OUTPUT



## DRAWING AN OUTLINED PYRAMID

- The output is indeed a 2D pyramid, but it is not the same as what we have sketched on paper.
- We expected the pyramid with the outline only and not the filled one.
- Thus, the required primitive here is gl.LINES and not gl.TRIANGLES.
- To draw such a geometry we need to do changes in our code in two steps.

## OUTLINED PYRAMID

First, in Step 18 specify the correct primitive.

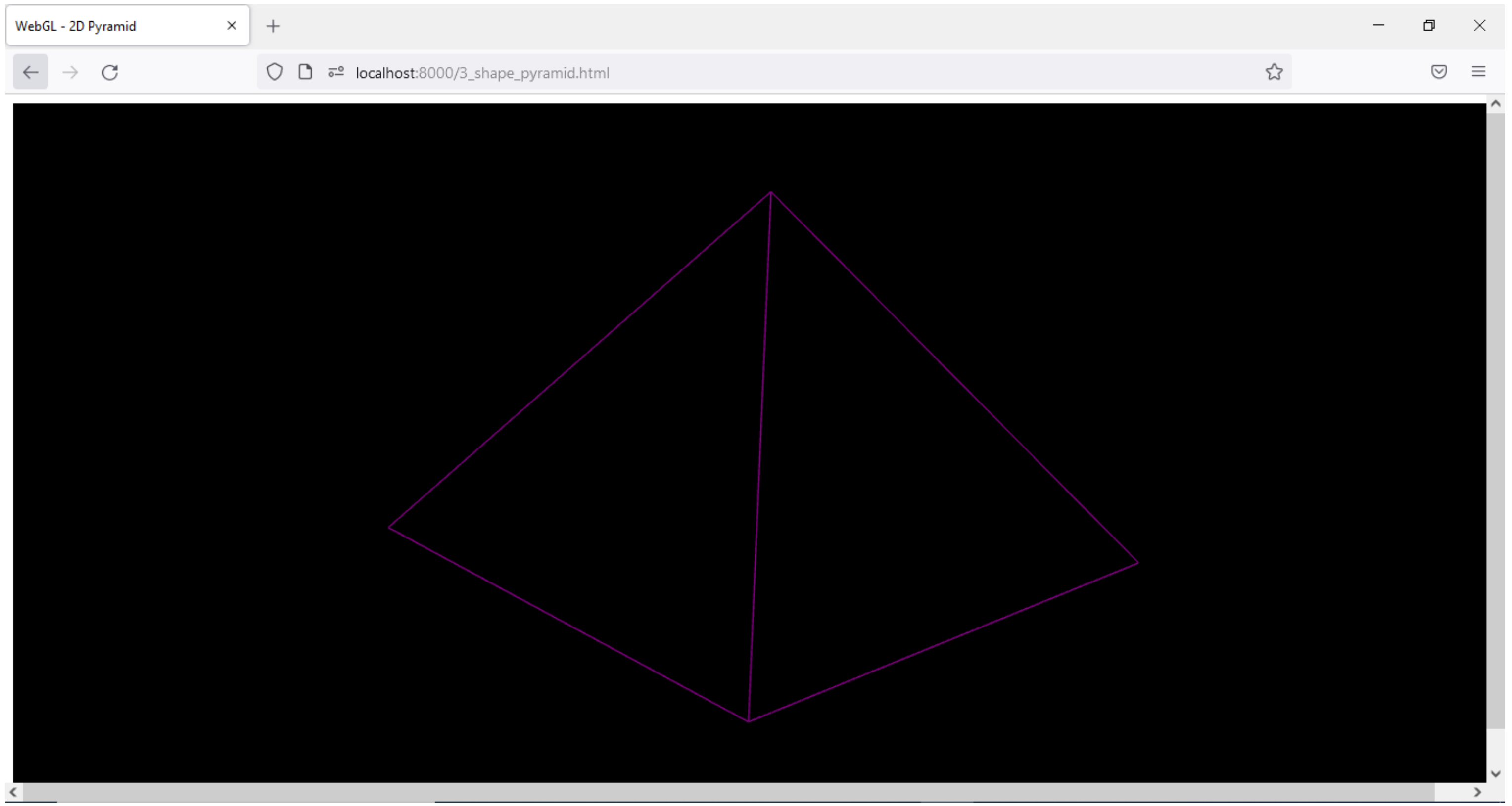
```
gl.drawElements(gl.LINES,
                IndicesArrayJS.length,
                gl.UNSIGNED_SHORT,
                0);
```

## OUTLINED PYRAMID

Second, specify the order of the indices of the vertices to draw the required lines.

```
var IndicesArrayJS =
                [
                    //A -0, B -1, C -2, D -3
                    0, 3, // Line 1: AD
                    3, 2, // Line 2: DC
                    2, 0, // Line 3: CA
                    2, 1, // Line 4: CB
                    1, 0 // Line 5: BA
                ];
```

# FINAL OUTPUT GEOMETRY



## SUMMARY

In order to understand programming concepts in WebGL, it is important to first know the underlying concepts for achieving parallel processing. In WebGL the data is separately provided from the actual computing units i.e., Vertex Shader, Fragment Shader.

## QUESTIONS, QUERIES?

You can check out my short book titled "Getting started with WebGL" available on kindle for detailed description of the topics covered here.

If you still have any queries or doubt regarding the content, do connect with me @ bhupendra.bisht59@gmail.com with subject line Questions in WebGL