M30242-Graphics and Computer Vision

Lecture 02: WebGL Programs

Overview

- Structure of WebGL programs
- Program components
  - Shaders
  - Buffers

HTML File

```
<!DOCTYPE HTML>
<html lang="en">
<head>
   <title>Some Title</title>
   <meta charset="utf-8">
   <script type="text/javascript”>
      //Your WebGL Code here.
   </script>
</head>

<!- the entry point of a WebGL program is in HTML body -->
<body onload="startup();">
<canvas id="myGLCanvas" width="500" height="500"></canvas>
</body>

</html>
```

Cont'd

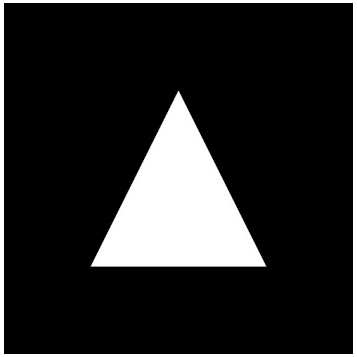
- The execution of the program is started by the `onload` event handler defined on the <body> tag with a JavaScript function `startup()` (the function itself is not shown here), which is the entry point into your WebGL application.
- The browser triggers the `onload` event when the user enters the web page, and the document and all external content is fully loaded.
- Within the `<body>` is an HTML5 `<canvas>` tag with an ID. Through the ID, you can access it from the JavaScript code.
- The `<canvas>` tag is the drawing surface that is used for WebGL, and this is where your WebGL graphics will end up within the web page.

Scripts

```
<html lang="en">

  <head>
    <script>
      <External Javescript libraries for WebGL debugging, matrix calculation, etc>
    </script>

    <script>
      <Vertex shader script>
    </script>

    <script>
      <Fragment shader script>
    </script>

    <script>
      <Here is main body of WebGL program and a Javascript function startup() is defined
       to set the entry point to WebGL program>
    </script>
  </head>

  <body onload="startup();">
    <canvas id="myGLCanvas"  width="500" height="500"> <canvas>
  </body>

<html>
```

WebGL Programs

- We show the structure of WebGL program via an example.
- The program only draws a 2D triangle on a web page.



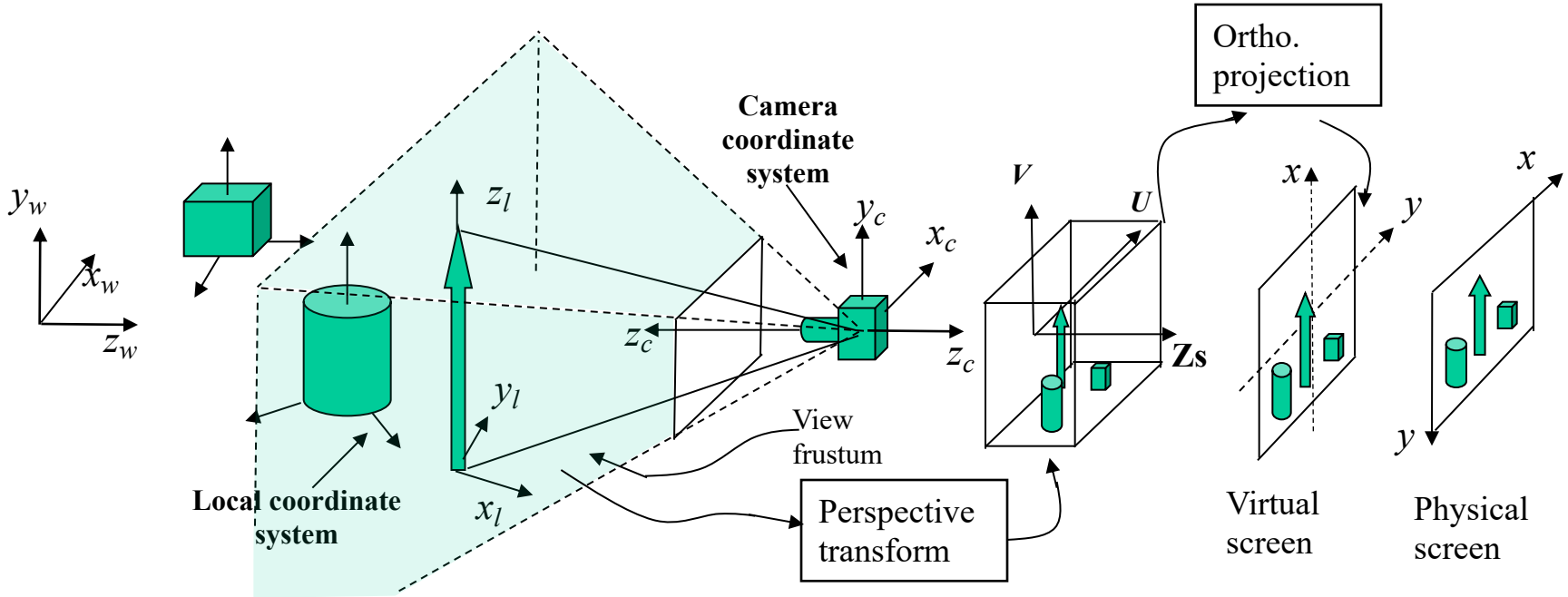
- Although the graphics is very simple, the program has all the necessary components of a WebGL program.

Shader Scripts

The shader scripts look like these:

```
<script id="shader-vs" type="x-shader/x-vertex">
attribute vec3 aVertexPosition;

void main() {
   gl_Position = vec4(aVertexPosition, 1.0);
}
</script>


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

void main() {
   gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
</script>
```

- Review of the pipeline



Script for WebGL Program

```
var gl;
var canvas;
var shaderProgram;
var vertexBuffer;

function startup() {
    //retrieve html canvas
    canvas = document.getElementById("myGLCanvas");
    //create wbegl context. Here, the debugging context is created
    //by calling a function in library "webgl-debug.js"
    gl = WebGLDebugUtils.makeDebugContext(createGLContext(canvas));
    setupShaders();
    setupBuffers();
    //Set the background - set the color to clear the canvas with
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    draw();
}
```

User defined functions are shown in **bold** case

Variables and Functions

- The example program contains some variables and functions:
  - Global variables: `gl, canvas, shaderProgram, vertexBuffer`.
  - A function for triggering WebGL program: `Startup()`
  - A function to create the context: `createGLContext()`
  - `setupShaders()` that sets up the shaders
  - `setupBuffers()` that creates buffers where geometric and other data for drawing will go
  - `loadShaderFromDOM()` that loads the shaders from shader scripts to the WebGL pipeline.
  - `draw()` that setup the viewport and do the actual drawing.

Start the Program

- When the `onload` event is triggered and `startup()` is called, the first thing this method does is to use the method `document.getElementById()` to get a reference (via variable `canvas`) to the `<canvas>` tag in the HTML page:

```
function startup() {
  canvas = document.getElementById("myGLCanvas");
  gl = createGLContext(canvas);
  …
}
```

- Method `document.getElementById()` is not part of WebGL. It is part of the Document Object Model (DOM) API, which defines how to access the objects within a web page.
- Then the function `createGLContext()` is called with the canvas as its argument. This creates a WebGLRenderingContext object and puts it in a global variable `gl` (or any name you like, but we need to access it through out the program, so it should be short and meaningful).

WebGLRenderingContext

```
//Create WebGL context. Creating context for drawing 2D or 3D graphics.
// E.g., for 2D drawing call getContext("2D")
function createGLContext(canvas) {
    var names = ["webgl", "experimental-webgl"];
    var context = null;
    for (var i=0; i < names.length; i++) {
        try {
            context = canvas.getContext(names[i]);
        } catch(e) {}
        if (context) {
                break;
        }
    }
    if (context) {
        context.viewportWidth = canvas.width;
        context.viewportHeight = canvas.height;
    } else {
        alert("Failed to create WebGL context!");
    }
    return context;
}
```

Cont'd

- The function `createGLContext()` creates a WebGLRenderingContext object by calling the standard HTML5 method `canvas.getContext()` with a standard context name, which could be:
  - `"experimental-webgl"` - supposed to be a temporary name for the WebGLRenderingContext for use during development of the WebGL specification, or
  - `"webgl"` - was supposed to be used once the specification was finalized.
- The WebGLRenderingContext is the interface that basically presents the complete WebGL API. Whenever we need to call a method in WebGL API, we call

  ```
  gl.someMethod();
  ```

  for example:

  ```
  gl.createShader();
  ```

  refers to the method `createShader()` that is part of the WebGL API.

Load Shader Scripts from DOM

- To create a WebGL shader that can be uploaded to the GPU and use for rendering, we first need to create a shader object, load the source code into the shader object, and then compile and link the shader. Function **`loadShaderFromDOM()`** does this.

```
var shaderScript = document.getElementById(id);
```

  retrieves the shader script from the html document. The parameter `id` could be "`shader-vs`"(vertex shader) or "`shader-fs`" (fragment shader).
- The script is then converted into a string stored in `shaderSource` as the source code in OpenGL ES shading language:

```
var shader;//declare a variable
shader = gl.createShader(gl.FRAGMENT_SHADER); or
shader = gl.createShader(gl.VERTEX_SHADER);
```

- Then load the shader source code `(shaderSource)` to the `shader` object and compile it:

```
gl.shaderSource(shader, shaderSource);//load source code to shader object
gl.compileShader(shader); //compile the shader
```

```
//Load shaders from DOM (document object model). This function will be
//called in setupShaders(). The parameters for argument id will be
//"shader-vs" and "shader-fs”
function loadShaderFromDOM(id) {
    var shaderScript = document.getElementById(id);
    // If there is no shader scripts, the function exist
    if (!shaderScript) {
         return null;
    }

    // Otherwise loop through the children for the found DOM element and
    // build up the shader source code as a string
    var shaderSource = "";
    var currentChild = shaderScript.firstChild;
    while (currentChild) {
         if (currentChild.nodeType == 3) { // 3 corresponds to TEXT_NODE
                  shaderSource += currentChild.textContent;
         }
         currentChild = currentChild.nextSibling;
    }
```

Cont’d on next slide

```
    //Create a WebGL shader object according to type of shader, i.e.,
    //vertex or fragment shader.
    var shader;
    if (shaderScript.type == "x-shader/x-fragment") {
          //call WebGL function createShader() to create fragment
          //shader object
          shader = gl.createShader(gl.FRAGMENT_SHADER);
    } else if (shaderScript.type == "x-shader/x-vertex") {
          //call WebGL function createShader() to create vertx shader obj.
          shader = gl.createShader(gl.VERTEX_SHADER);
    } else {
          return null;
    }
    //load the shader source code (shaderSource) to the shader object.
    gl.shaderSource(shader, shaderSource);
    gl.compileShader(shader); //compile the shader
    //check compiling status.
    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
          alert(gl.getShaderInfoLog(shader));
          return null;
    }
    return shader;
}
```

Create Shader Programs

```
function setupShaders() {
    //Create vertex and fragment shaders
    vertexShader = loadShaderFromDOM("shader-vs");
    fragmentShader = loadShaderFromDOM("shader-fs");
    //create a webgl program object
    shaderProgram = gl.createProgram();
    //load the compiled shaders(compilation has been done in loadShaderFromDOM()
    // function) to the program object
    gl.attachShader(shaderProgram, vertexShader);
    gl.attachShader(shaderProgram, fragmentShader);
    //link shaders and check linking status
    gl.linkProgram(shaderProgram);
    if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
          alert("Failed to setup shaders");
    }
    //activate the program
    gl.useProgram(shaderProgram);

    /*add a property (named as vertexPositionAttttribute)to the shader program
    object. The property is the attribute in the vertex shader, which has been
    loaded to the program object. Function getAttribLocation() finds the pointer to
    this attribute. We need to access this property in other part of the program*/
    shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
                                            "aVertexPosition");
}
```

Cont'd

- In function **`setupShaders`**`()`, vertex and fragment shaders are first created by calling **`loadShaderFromDOM()`**:

```
vertexShader = loadShaderFromDOM("shader-vs");
fragmentShader = loadShaderFromDOM("shader-fs");
```

- Then a WebGL program object is created

```
shaderProgram = gl.createProgram();
```

- The following statements attach the compiled vertex/fragment shader to the `shadeProgram` object, links everything together to the program and activate the program so that WebGL can use it.

```
gl.attachShader(shaderProgram, vertexShader);
gl.attachShader(shaderProgram, fragmentShader);
gl.linkProgram(shaderProgram);
gl.useProgram(shaderProgram);
```

Cont'd

- At the end of function **`setupShaders`**`()`, the method `gl.getAttribLocation()` is called to find the index/pointer within the program object for attribute of the vertex shader, `aVertexPosition:` and then activate it (it is disabled by default)

```
shaderProgram.vertexPositionAttribute = gl.getAttribLocation
                                    (shaderProgram, "aVertexPosition");
```

- The pointer/index is saved in the `shaderProgram` object as a new property that is named `vertexPositionAttribute`.
- Note: In JavaScript, a new property of an object can be created simply by assigning a value to it. The object `shaderProgram` does not have a predefined property called `vertexPositionAttribute`, but this property is created by assigning a value to it.
- Later in the `draw()` function, the pointer that is saved in property `vertexPositionAttribute` will be used to **connect** the buffer containing the vertex data to the attribute `aVertexPosition` in the vertex shader.

Data Buffers

- Buffers are places for data.
- All data, e.g., vertex coordinates, texture coordinates, indices, colours must be stored in their buffers.
- In this simple program, only the buffer for the vertex coordinates of a triangle is used.
- The buffers need to be created with the assigned type and bound to the data by calling functions in WebGL API

Set up Buffers

```
function setupBuffers() {
    //A buffer object is first created by calling gl.createBuffer()
    vertexBuffer = gl.createBuffer();
    //Then created buffer is initialised to be type of gl.ARRAY_BUFFER by
    // calling bindBuffer(). There are two type WebGL buffers. The other
    //is gl.ELEMENT_ARRAY_BUFFER.
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
    //Actual coordinates for the vertices
    var triangleVertices = [
                            0.0,  0.5, 0.0,
                           -0.5, -0.5, 0.0,
                            0.5, -0.5, 0.0
                           ];
    //Load the vertex data to the buffer
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices),
                                                gl.STATIC_DRAW);
    //Add properties to vertexBuffer object
    vertexBuffer.itemSize = 3;      //3 coordinates of each vertex
    vertexBuffer.numberOfItems = 3; //3 vertices in all in this buffer
}
```

Cont'd

- After having the shaders in place, we set up the buffers for the vertex data.
- In a typical WebGL program, there will be buffers for different types of data, e.g., buffer for normals, buffer for texture coordinates, etc. In this example, we only have a buffer for the vertex coordinates. This is done in function **`setupBuffers()`.**
- The function starts by calling `gl.createBuffer()` to create a WebGLBuffer object that is assigned to the **global** variable `vertexBuffer:`

```
vertexBuffer = gl.createBuffer();
```

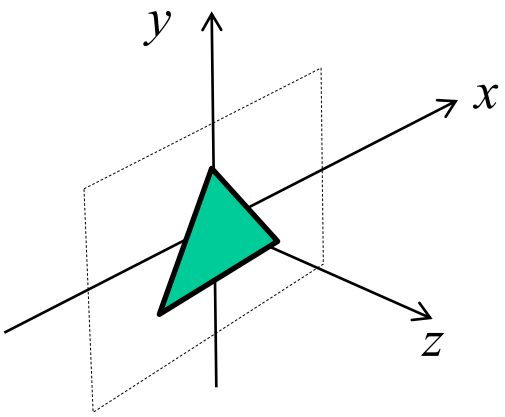
- Then the create buffer is bound to WebGL pipeline as `gl.ARRAY_BUFFER`.

```
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
```

Actual Data

- The actual vertex data (coordinates) for the triangle are specified in the JavaScript array `triangleVertices`.

```
var triangleVertices = [
        0.0,  0.5,  0.0,
       -0.5, -0.5,  0.0,
        0.5, -0.5,  0.0
];
```



Cont'd

- The default coordinate system has its origin with coordinates (0, 0, 0) in the centre of the viewport.
  - The x-axis is horizontal and pointing to the right,
  - The y-axis is pointing upwards, and
  - The z-axis is pointing out of the screen towards you.
- All three axes stretch from –1 to 1.
  - The lower-left corner of the viewport will then have the coordinates x = −1 and y = −1, and the upper-right corner of the viewport will have the coordinates x = 1 and y = 1.
  - Since this example draws a 2D triangle, all the z-values are set to zero so the triangle will be drawn in the xy-plane at z = 0.
- This is the 3D screen space in the graphics pipeline.

Cont'd

- Next, a `Float32Array` object is created from the JavaScript array that contains the vertices.

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

- The call to `gl.bufferData()` writes the vertices data to the currently bound WebGLBuffer object. This call tells WebGL which data it should place in the buffer object that was created with `gl.createBuffer()`.
- Then two new properties are added to the `vertexBuffer` object, which will be needed later:

```
vertexBuffer.itemSize = 3;
vertexBuffer.numberOfItems = 3;
```

  - `itemSize` specifies how many components exist for each attribute (vertex).
  - `numberOfItems` which specifies the number of items or vertices that exist in this buffer.

Drawing

```
function draw() {
    /*setup a viewport that is the same as the canvas using function
    viewport(int x, int y, sizei w, sizei h) where x and y give the x and
    y window coordinates of the viewport's lower left corner and w and h
    give the viewport's width and height.*/

    gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);

    /*fill the canvas with solid colour. Default is black
    If other colour is desirable using function gl.clearColor (r,g,b,a) */
    gl.clear(gl.COLOR_BUFFER_BIT);//COLOR_BUFFER_BIT flag that enables
                                  //the view buffer for colouring

    /*Link the pointer to "aVertexPosition" (Still remember it?) to the
    currently bound gl.ARRAY_BUFFER. See function setupBuffers() */
    gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                  vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

    //Draw the triangle
    gl.drawArrays(gl.TRIANGLES, 0, vertexBuffer.numberOfItems);
}
```

Cont'd

- The code for drawing the actual scene is in function `draw()`.
- First, the viewport is specified. When a WebGL context is created, the viewport is initialized to a rectangle with its origin at (0, 0) and a width and height equal to the canvas width and height. This means that the call to `gl.viewPort()` does not actually modify anything in this example.
- The method `gl.clear()` with the argument `gl.COLOR_BUFFER_BIT` tells WebGL to clear the colour buffer to the colour that was previously specified with `gl.clearColor()` (black, in function `startup()`).

Cont'd

- The next two statements link the actual data (they are bound to the `gl.ARRAY_BUFFER` target in function `setupBuffers`), to the attribute, `aVertexPosition`, and make it active:

```
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                       vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
```

- The last statement

```
gl.drawArrays(gl.TRIANGLES, 0, vertexBuffer.numberOfItems);
```

  tells WebGL draw the data as triangular arrays

- Other drawing options are available, but we will defer their discussion to later lectures

Cont'd

- The method `gl.vertexAttribPointer()`
  - The first argument assigns the WebGLBuffer object currently bound to the `gl.ARRAY_BUFFER` target to a vertex attribute passed in as index.
  - The second argument is the number of components per attribute, 3 in this example (x, y, and z coordinate for each vertex position) and this value is stored as a property named `itemSize` of the `vertexBuffer` object.
  - The third argument specifies that the values in the vertex buffer object should be interpreted as floats. If you send in data that does not consist of floats, the data needs to be converted to floats before it is used in the vertex shader.
  - The fourth argument is called the **normalized flag**, and it decides how non-floating point data should be converted to floats. In this example, the values in the buffer are floats, so the argument will not be used anyway.
  - The fifth argument is called **stride.** Value zero means that the data is stored sequentially in memory.
  - The sixth and last argument is the offset into the buffer, and since the data starts at the start of the buffer, this argument is set to zero as well.

Vertex & Fragment Shaders

- All WebGL programs must have both a vertex shader and a fragment shader.
- Shaders are written in OpenGL Shading Language. We write them as Scripts that will be converted into strings and loaded to the program object of the WebGL pipeline.
- We have the simplest shaders here:

```
<script id="shader-vs" type="x-shader/x-vertex">
attribute vec3 aVertexPosition;
void main() {
   gl_Position = vec4(aVertexPosition, 1.0);
}
</script>
```

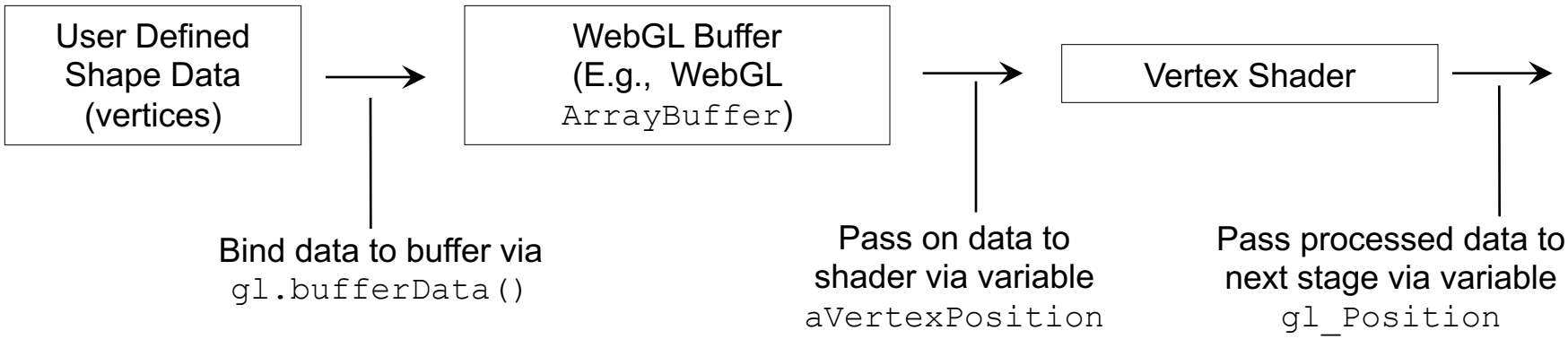
```
<script id="shader-fs" type="x-shader/x-fragment">
precision mediump float;
void main() {
   gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
</script>
```

Vertex Shader

- The first line defines a variable `aVertexPosition` of type `vec3`, which means it is a vector with three components.
- The variable is of type `attribute`. Attributes are special input variables that are used to pass per vertex data from the WebGL API (vertex buffer) to the vertex shader (Vertex data are defined in vertex buffer, not in shaders). Think `attributes` as the interface variables to pass data to the shaders.
- The next line of the vertex shader declares a `main()` function, which is the entry point for the execution of the vertex shader.
- The body of the `main()` function is very simple and just pass through the incoming vertex, `aVertexPosition`, to the built-in variable `gl_Position`. We call such shaders pass-through shaders, because nothing has been done by the programmer.

Vertex Shader

- **`gl_Position`** is a predefined or built-in variable of WebGL pipeline
- it is of the type vec4:
  - It uses homogeneous coordinates – a 4-tuple with the fourth component set 1.
  - It contains the position of the vertex when the vertex shader is finished with it, and it is passed on to the next stage in the WebGL pipeline.
  - All vertex shaders must assign a value to this predefined variable
- To make the vertex data go through the API and end up in the `aVertexPosition` attribute, we have to set up a buffer for the vertex data and connect the buffer to the `aVertexPosition` attribute.



Fragment Shader

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

- The fragment shader uses a `precision` qualifier to declare that the precision used for floats in the fragment shader should be of *medium* precision (32-bit).
- The same as for the vertex shader, the `main()` function defines the entry point of the fragment shader. In this example, It writes a `vec4` representing the colour white into the built-in variable **`gl_FragColor`**.
- **`gl_FragColor`** is defined as a four-component vector that contains the output colour in RGBA format that the fragment has when the fragment shader is finished with the fragment.

Summary

Typical HTML For WebGL

```
<html lang="en">

  <head>
    <script>
      <External Javescript libraries for WebGL debugging, matrix calculation, etc>
    </script>

    <script>
      <Vertex shader script>
    </script>

    <script>
      <Fragment shader script>
    </script>

    <script>
      <Here is main body of WebGL program and a Javascript function startup() is defined
       to set the entry point to WebGL program>
    </script>
  </head>

  <body onload="startup();">
    <canvas id="myGLCanvas"  width="500" height="500"> <canvas>
  </body>

<html>
```

WebGL Program Structure

startup()

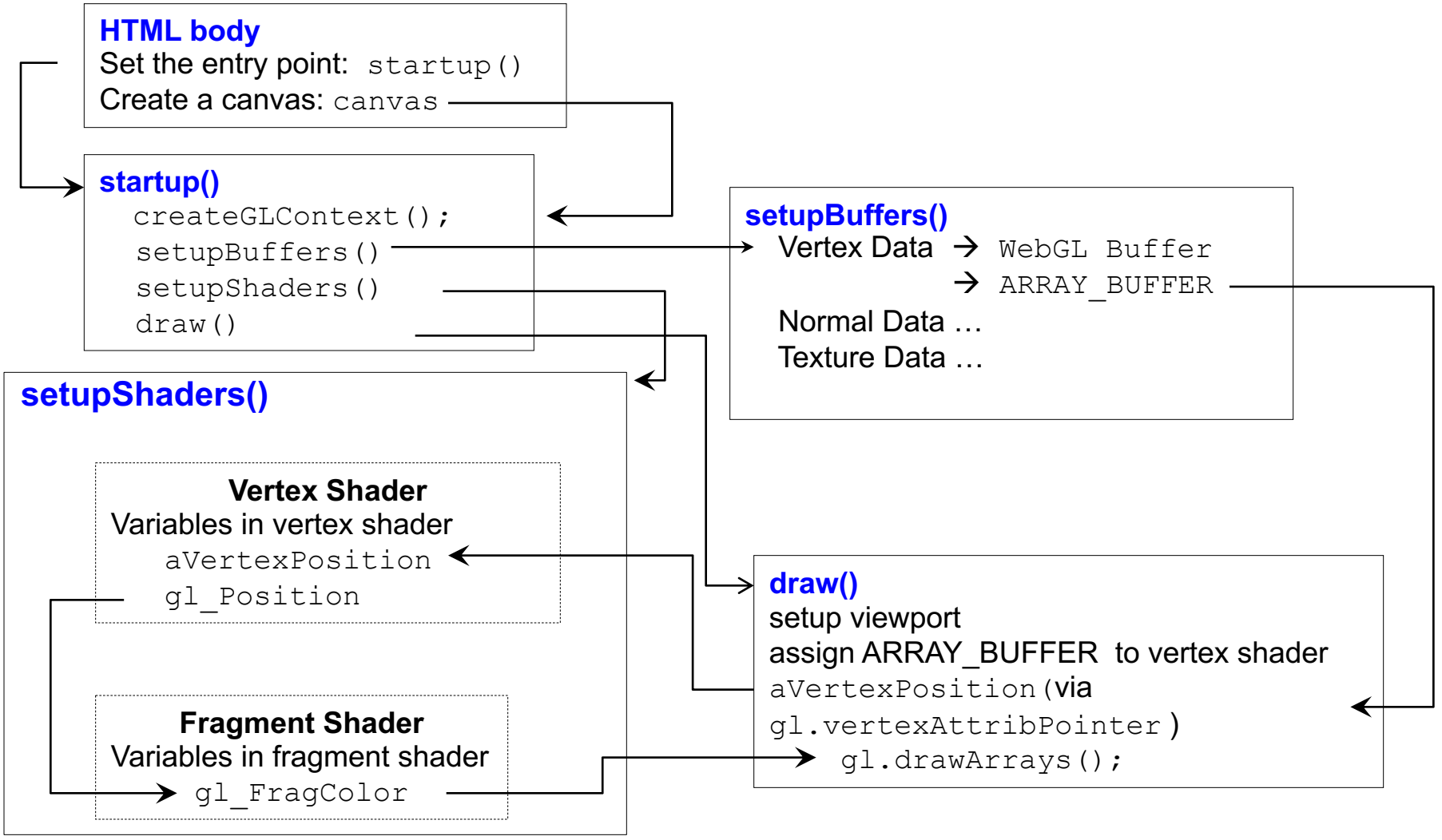
Create WebGL context: `createGLContext();`

Setup shaders: `setupShaders();`

Setup buffers: `setupBuffers();`

Draw graphics: `draw()`

WebGL Program Follow



Further Reading

- Anyuru, A., WebGL Programming – Develop 3D Graphics for the Web
  - Chapter 2