3D Graphics Programming Tools

WebGL

Dr. Xianhui Cherry Che
x.che@qmul.ac.uk

Queen Mary
University of London

Learning Objectives

- Understand the basic infrastructure of WebGL and how it differs from (or relates to) OpenGL.
- Get familiar with the graphics pipeline in WebGL.
- Practice the use of WebGL with a basic exercise.

- This lecture is **optional**, and it helps to gain further understanding of 3D graphic programming tools.

Topics

- WebGL Overview
- WebGL Graphics Pipeline
- Draw a Triangle

Queen Mary University of London

OpenGL-Related APIs

API	Technology Used
OpenGL ES	It is the library for 2D and 3D graphics on embedded systems - including consoles, phones, appliances, and vehicles. OpenGL ES 3.1 is its latest version. It is maintained by the Khronos Group www.khronos.org
JOGL	It is the Java binding for OpenGL. JOGL 4.5 is its latest version and it is maintained by jogamp.org.
WebGL	It is the JavaScript binding for OpenGL. WebGL 1.0 is its latest version and it is maintained by the khronos group.
OpenGLSL	OpenGL Shading Language. It is a programming language which is a companion to OpenGL 2.0 and higher. It is a part of the core OpenGL 4.4 specification. It is an API specifically tailored for embedded systems such as those present on mobile phones and tablets.

WebGL

- WebGL (Web Graphics Library) is the new standard for 3D graphics on the Web, designed for rendering 2D graphics and interactive 3D graphics.
- It is derived from OpenGL's ES 2.0 library which is a low-level 3D API for phones and other mobile devices.
- It is a JavaScript API that can be used with HTML5.
- HTML 5 has several features to support 3D graphics such as 2D Canvas, WebGL, SVG, 3D CSS transforms, and SMIL.
- WebGL code is written within the **<canvas>** tag of HTML5. It is a specification that allows Internet browsers access to Graphic Processing Units (GPUs) on those computers where they are used.
- Official reference document:
  - https://www.khronos.org/registry/webgl/specs/latest/2.0/
  - https://www.khronos.org/files/webgl/webgl-reference-card-1_0.pdf

WebGL vs. OpenGL

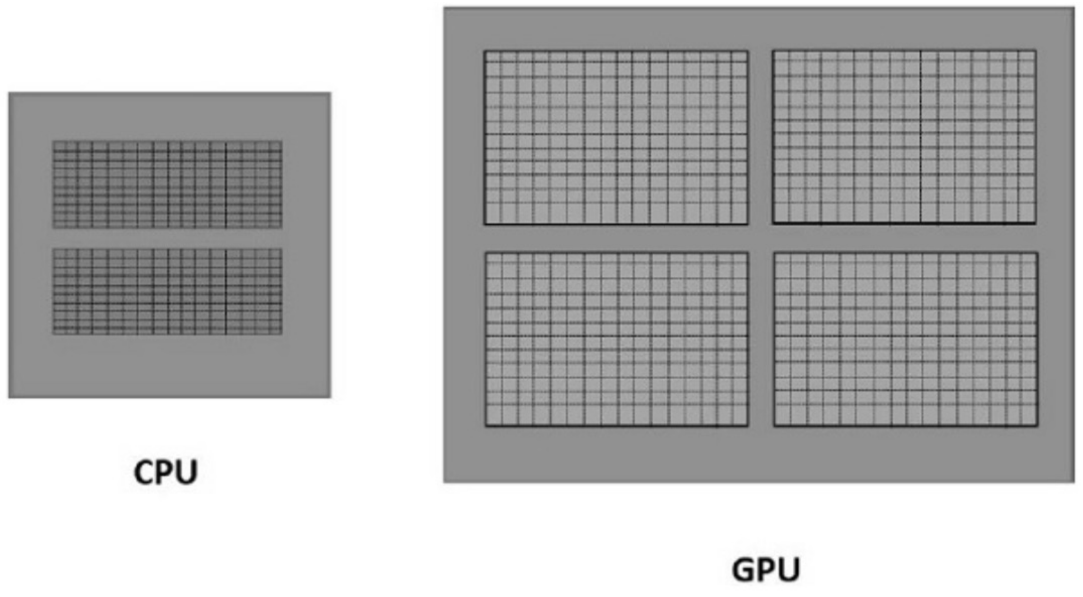
	WebGL	OpenGL
Definition	It is designed for rendering 2D and 3D graphics.	It is a cross-language and platform API to render 2D and 3D vector graphics.
Application	It is mainly used to run in the browser for web applications.	It is mainly used in desktop applications.
Programmed	It is programmed in JavaScript programming.	It is written in C++ language.
Features	It has fewer features comparatively.	It has many features to make the application or graphics more interactive.
Pipeline	In WebGL, there is no fixed-function pipeline.	In OpenGL, there is a fixed function pipeline.
Website	https://www.khronos.org/webgl/	Opengl.org

Rendering

- There are two types of rendering –
  - **Software Rendering** – All the rendering calculations are done with the help of CPU.
  - **Hardware Rendering** – All the graphics computations are done by the GPU.
- Rendering can be done locally or remotely.
  - **Server-Based Rendering** – If the image to be rendered is way too complex, then rendering is done remotely on a dedicated server having enough of hardware resources required to render complex scenes.
  - **Client-Based Rendering** – Rendering can also be done locally by the CPU..
- WebGL follows a client-based rendering approach to render 3D scenes. All the processing required to obtain an image is performed locally using the client's graphics hardware.

GPU

- GPU is a single chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines capable of processing a minimum of 10 million polygons per second.
- Unlike multi-core processors with a few cores optimized for sequential processing, a GPU consists of thousands of smaller cores that process parallel workloads efficiently. Therefore, the GPU accelerates the creation of images in a frame buffer intended for output to a display.



GPU-Accelerated Computing

- In GPU accelerated computing, the application is loaded into the CPU. Whenever it encounters a computing-intensive portion of the code, then that portion of code will be loaded and run on the GPU. It gives the system the ability to process graphics in an efficient way.
- GPU will have a separate memory and it runs multiple copies of a small portion of the code at a time. The GPU processes all the data in its local memory. Therefore, the data that is needed to be processed by the GPU should be loaded/copied to the GPU memory and then be processed.
- The communication overhead between the CPU and GPU should be reduced to achieve faster processing of 3D programs.

HTML5 Canvas

- The existing canvas element of HTML-5 to write WebGL applications.
- Canvas has three attributes: id/class, width, and height.

```
<html>
<head>
<style>
#mycanvas{border:1px solid red;}
</style>
</head>
<body>
<canvas id = "mycanvas" width = "100" height = "100">
</canvas>
</body>
</html>
```

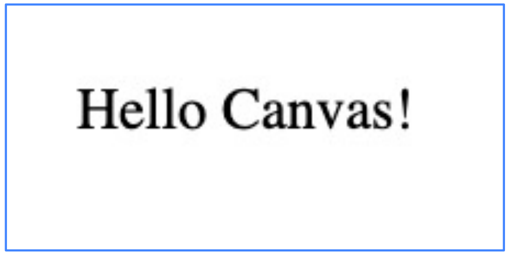
Output →

Rendering Context

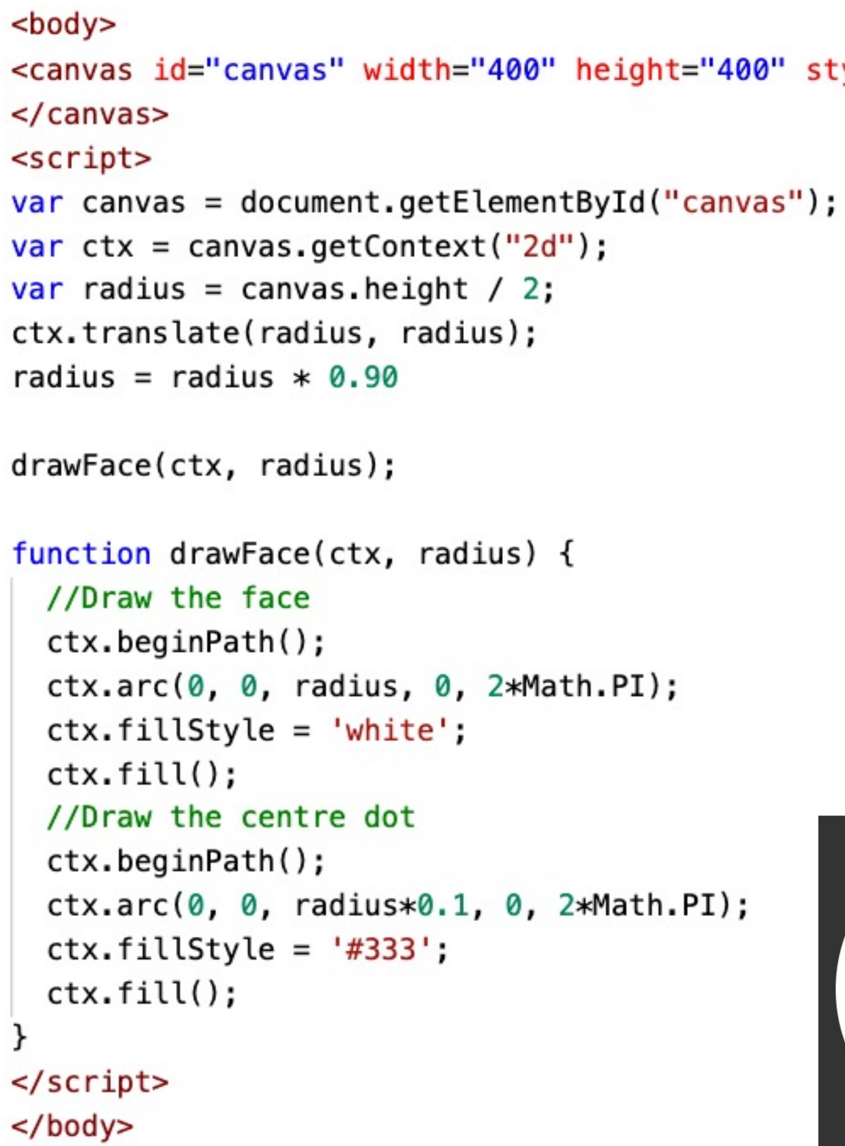
- This scripting language access the rendering context and draw on on the canvas.

```
<html>
<body>
<canvas id = "mycanvas" width = "600" height = "200"></canvas>
<script>
var canvas = document.getElementById('mycanvas');
var context = canvas.getContext('2d');
context.font = '20pt Calibri';
context.fillText('Hello Canvas!', 70, 70);
</script>
</body>
</html>
```

Output →


Hello Canvas!

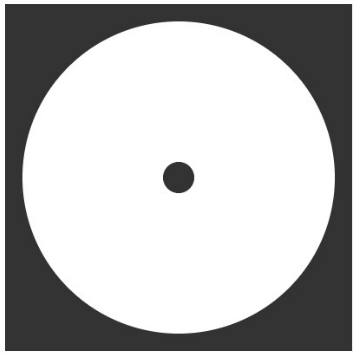
Recap: EBU6305

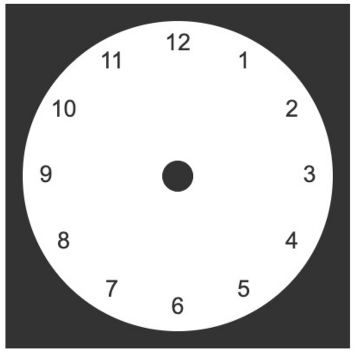


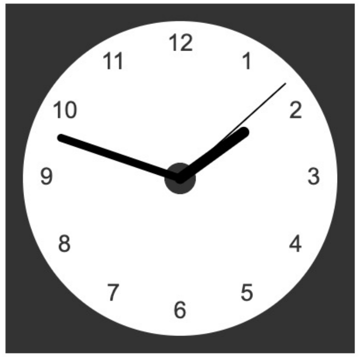
```
<body>
<canvas id="canvas" width="400" height="400" style="background-color:■#333">
</canvas>
<script>
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");
var radius = canvas.height / 2;
ctx.translate(radius, radius);
radius = radius * 0.90

drawFace(ctx, radius);

function drawFace(ctx, radius) {
  //Draw the face
  ctx.beginPath();
  ctx.arc(0, 0, radius, 0, 2*Math.PI);
  ctx.fillStyle = 'white';
  ctx.fill();
  //Draw the centre dot
  ctx.beginPath();
  ctx.arc(0, 0, radius*0.1, 0, 2*Math.PI);
  ctx.fillStyle = '#333';
  ctx.fill();
}
</script>
</body>
```







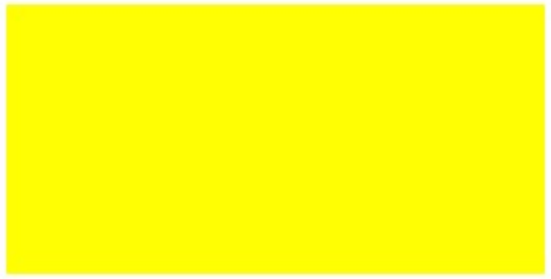
WebGL Context

- To create a WebGL rendering context on the canvas element, the string **experimental-webgl** should be passed instead. Some browsers support only '**webgl**'.

```
<html>
<canvas id = 'mycanvas'></canvas>
<script>
var canvas = document.getElementById('mycanvas');
var gl = canvas.getContext('experimental-webgl');
gl.clearColor(1.0,1.0,0.0,1);
gl.clear(gl.COLOR_BUFFER_BIT);
</script>
</html>
```

Look familiar? ☺

Output →



WebGL Context

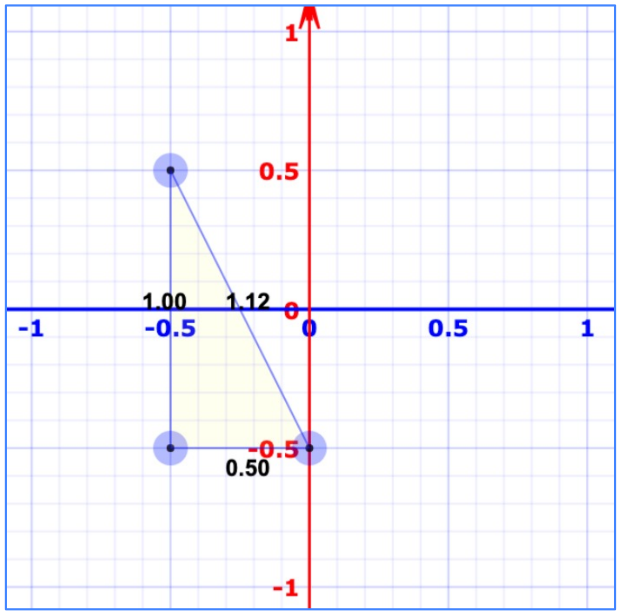
- To deal with exceptions:

```
<html>
<canvas id = 'mycanvas'></canvas>
<script>
var canvas = document.getElementById('mycanvas');
var gl = canvas.getContext('experimental-webgl’);
if (!gl){// Give users a message – Sorry no WebGL for you!}
…
…
</script>
</html>
```

Vertices in WebGL

- Unlike OpenGL, there are no predefined methods in WebGL to render the vertices directly. They have to be stored manually using JavaScript arrays.
- Example:

```
var vertices = [ -0.5, 0.5, -0.5,-0.5, 0.0, -0.5]
```



(-0.5, 0.5)
(-0.5, -0.5)
(0.0, -0.5)

Topics

- WebGL Overview
- WebGL Graphics Pipeline
- Draw a Triangle

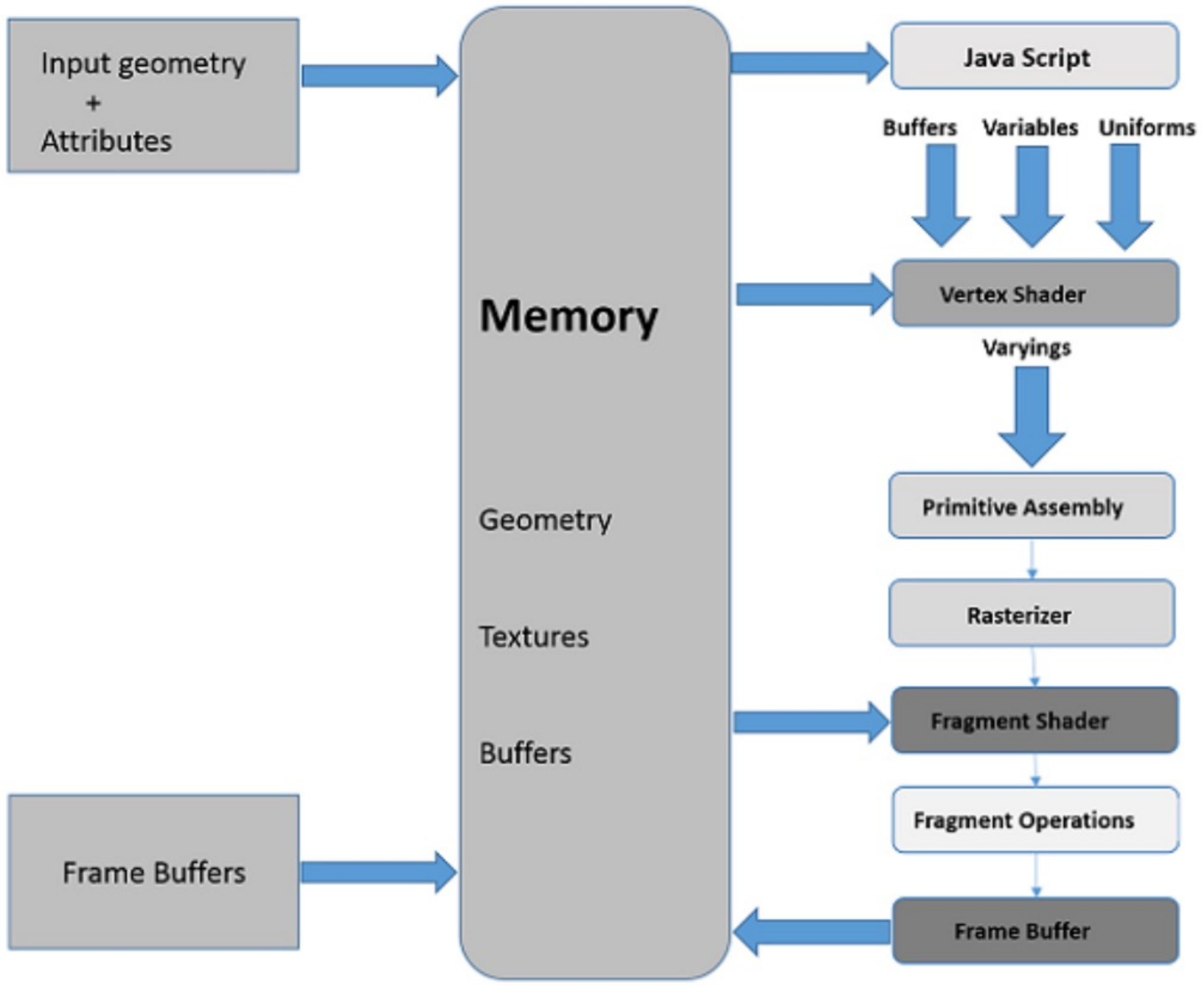
Buffers

- Buffers are the memory areas of WebGL that hold the data.
- **Frame buffer** is a portion of graphics memory that hold the scene data. This buffer contains details such as width and height of the surface, colour of each pixel, depth and stencil buffers.
- **Vertex buffer** stores data about the vertices. Vertex buffer object is also known as VBO.
- **Index buffer** stores data about the indices.

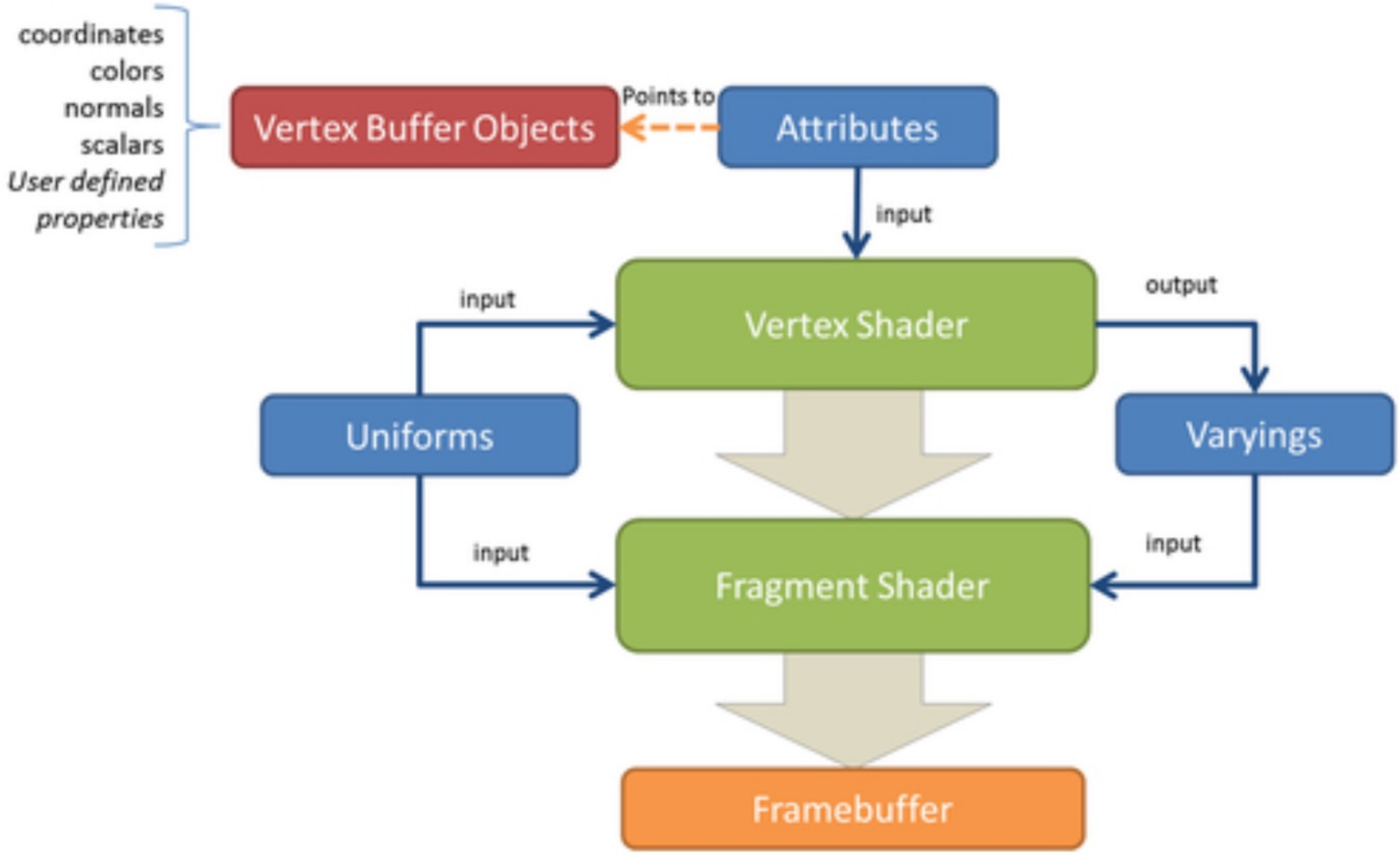
Shader Programs

Vertex Shader

WebGL Graphics Pipeline



WebGL Graphics Pipeline

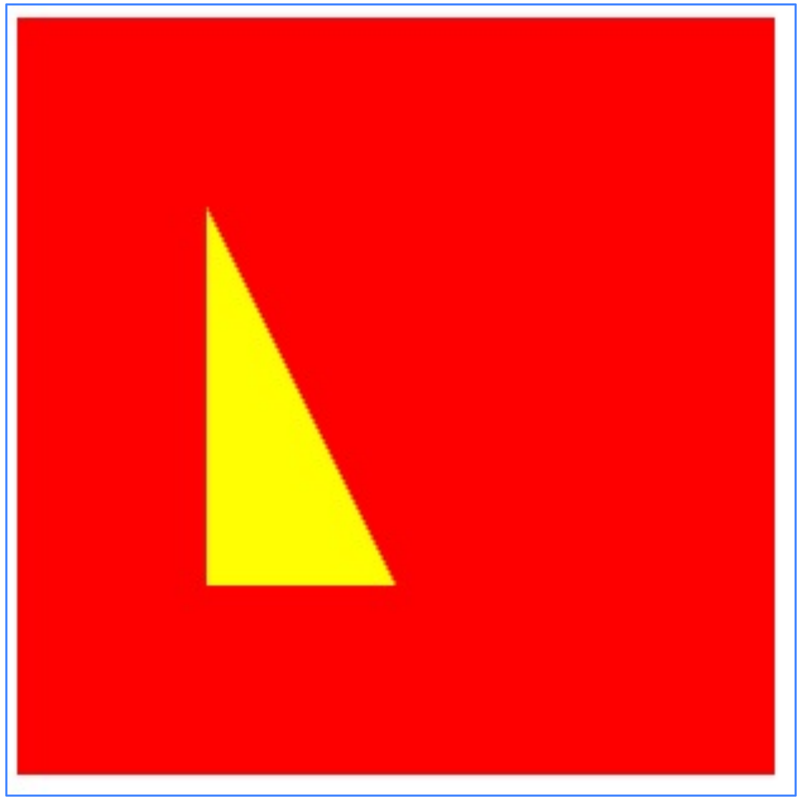


Topics

- WebGL Overview
- WebGL Graphics Pipeline
- Draw a Triangle

Draw a Triangle

- Use WebGL to draw a triangle:



The first WebGL program is not as easy as other Hello World programs. Nevertheless, let's begin!

Draw a Triangle – Part 1

- Prepare the canvas and get WebGL context

```
var canvas = document.getElementById('my_Canvas');
var gl = canvas.getContext('experimental-webgl');
```

Draw a Triangle – Part 2

- Define the geometry and store it in vertex buffer object

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

// Create a new vertex buffer object
var vertex_buffer = gl.createBuffer();

// Bind an empty array buffer to it
gl.bindBuffer(gl.ARRAY_BUFFER, vertex_buffer);

// Pass the vertices data to the buffer
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);

// Unbind the buffer
gl.bindBuffer(gl.ARRAY_BUFFER, null);
```

Draw a Triangle – Part 3

- Create and compile the vertex shader

```
// Vertex shader source code
var vertCode = 'attribute vec2 coordinates; void main(void) {gl_Position =
vec4(coordinates,0.0, 1.0);}';

//Create a vertex shader object
var vertShader = gl.createShader(gl.VERTEX_SHADER);

//Attach vertex shader source code
gl.shaderSource(vertShader, vertCode);

//Compile the vertex shader
gl.compileShader(vertShader);
```

Draw a Triangle – Part 4

- Create and compile the fragment shader

```
//Fragment shader source code
var fragCode = 'void main(void) {gl_FragColor = vec4(1.0, 1.0, 0.0, 1);}';

// Create fragment shader object
var fragShader = gl.createShader(gl.FRAGMENT_SHADER);

// Attach fragment shader source code
gl.shaderSource(fragShader, fragCode);

// Compile the fragment shader
gl.compileShader(fragShader);
```

Draw a Triangle – Part 5

- Create and compile the shader program

```
// Create a shader program object to store combined shader program
var shaderProgram = gl.createProgram();

// Attach a vertex shader
gl.attachShader(shaderProgram, vertShader);

// Attach a fragment shader
gl.attachShader(shaderProgram, fragShader);

// Link both programs
gl.linkProgram(shaderProgram);

// Use the combined shader program object
gl.useProgram(shaderProgram);
```

Draw a Triangle – Part 6

- Associate the shader programs to the vertex buffer objects

```
//Bind vertex buffer object
gl.bindBuffer(gl.ARRAY_BUFFER, vertex_buffer);

//Get the attribute location
var coord = gl.getAttribLocation(shaderProgram, "coordinates");

//point an attribute to the currently bound VBO
gl.vertexAttribPointer(coord, 2, gl.FLOAT, false, 0, 0);

//Enable the attribute
gl.enableVertexAttribArray(coord);
```

Draw a Triangle – Part 7

- Drawing the required object (triangle)

```
// Clear the canvas
gl.clearColor(1, 0, 0.0, 1.0);

// Enable the depth test
gl.enable(gl.DEPTH_TEST);

// Clear the colour buffer bit
gl.clear(gl.COLOR_BUFFER_BIT);

// Set the view port
gl.viewport(0,0,canvas.width,canvas.height);

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

Draw a Triangle – Full Program

```
<html> <body><canvas width = "300" height = "300" id = "my_Canvas"></canvas>
<script>
var canvas = document.getElementById('my_Canvas');
var gl = canvas.getContext('experimental-webgl');
var vertices = [-0.5, 0.5, -0.5, -0.5, 0.0, -0.5,];
var vertex_buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertex_buffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
gl.bindBuffer(gl.ARRAY_BUFFER, null);
var vertCode =
'attribute vec2 coordinates; void main(void) {gl_Position = vec4(coordinates,0.0, 1.0);}';
var vertShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertShader, vertCode);
gl.compileShader(vertShader);
var fragCode = 'void main(void) {gl_FragColor = vec4(1.0, 1.0, 0.0, 1);}';
var fragShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragShader, fragCode);
gl.compileShader(fragShader);
var shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, vertShader);
gl.attachShader(shaderProgram, fragShader);
gl.linkProgram(shaderProgram);
gl.useProgram(shaderProgram);
gl.bindBuffer(gl.ARRAY_BUFFER, vertex_buffer);
var coord = gl.getAttribLocation(shaderProgram, "coordinates");
gl.vertexAttribPointer(coord, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(coord);
gl.clearColor(1, 0, 0.0, 1.0);
gl.enable(gl.DEPTH_TEST);
gl.clear(gl.COLOR_BUFFER_BIT);
gl.viewport(0,0,canvas.width,canvas.height);
gl.drawArrays(gl.TRIANGLES, 0, 3);
</script></body></html>
```

Questions?

x.che@qmul.ac.uk

Acknowledgement: www.tutorialspoint.com