GPU Programming with WebGL

Frederick Li

Durham University

Recap …

- Major Tasks in CG:
  - 3D Modelling, 3D Environment (Scene) Construction, Rendering

- General Graphics System

  - Functionalities of Graphics Processor:
    3D and 2D drawing commands

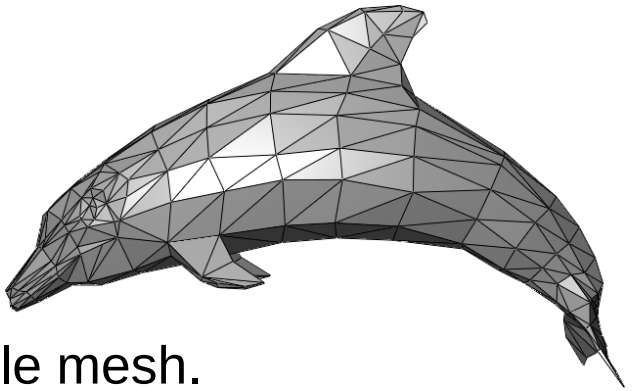
  - Frame Buffer:
    Memory size, Double buffering, adaptation to stereoscopic display (e.g. VR goggles)
  - Display controller:
    display frequency, interaction with frame buffer

This Lesson

- Concept of GPU programming
- Understand programmable rendering pipeline
- Start to learn WebGL, knowing how to render simple polygon models (or meshes)

- **Polygon model / mesh:** comprises a set of connected polygons to represent an object.

In CG, we usually assume triangle as the type of polygon to use because triangles are always flat.



So, a polygon mesh is also called a triangle mesh.

Preparation

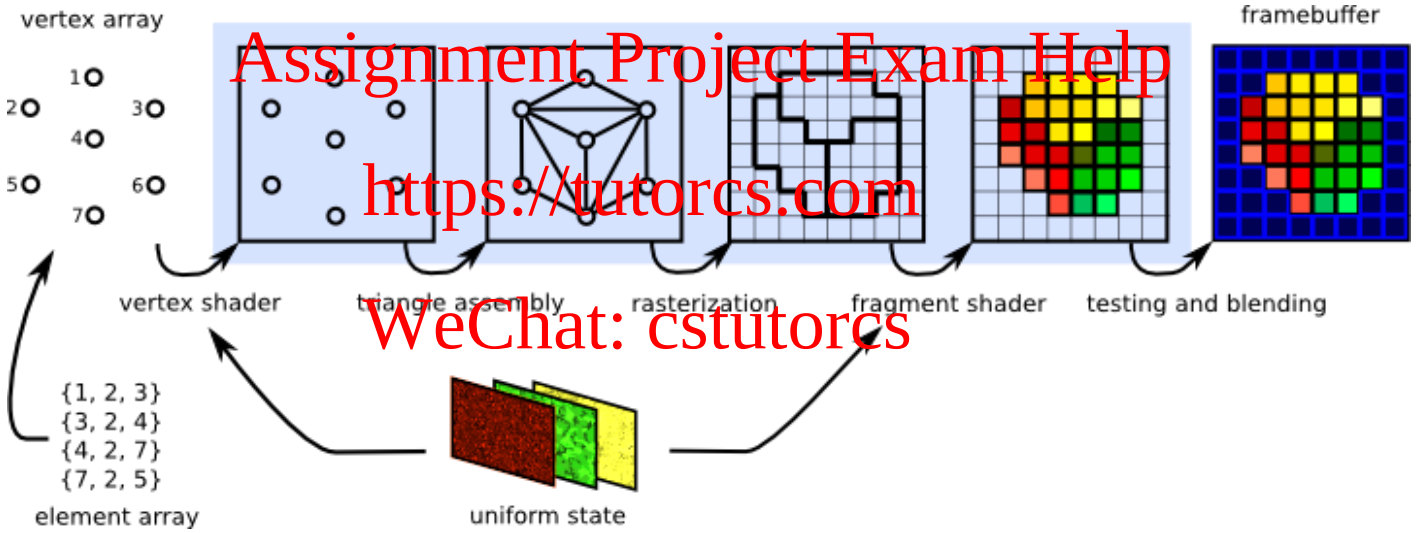
- Are you ready?
- Knowledge: HTML5, CSS3, Javascript
- Learning curve is a bit high: GPU programming follows a parallel programming paradigm.
- Software requirements:
  - WebGL-enabled browser (check with http://get.webgl.org/)
  - Text editor
  - No complicated IDE is required

Why WebGL?

- Cross-platform, browser-based
- Hardware-based rendering
- Support programmable rendering pipeline
  (an implementation of OpenGL ES 2.0 which is designed for embedded and hand-held devices)
- Zero setup effort before you can start programming

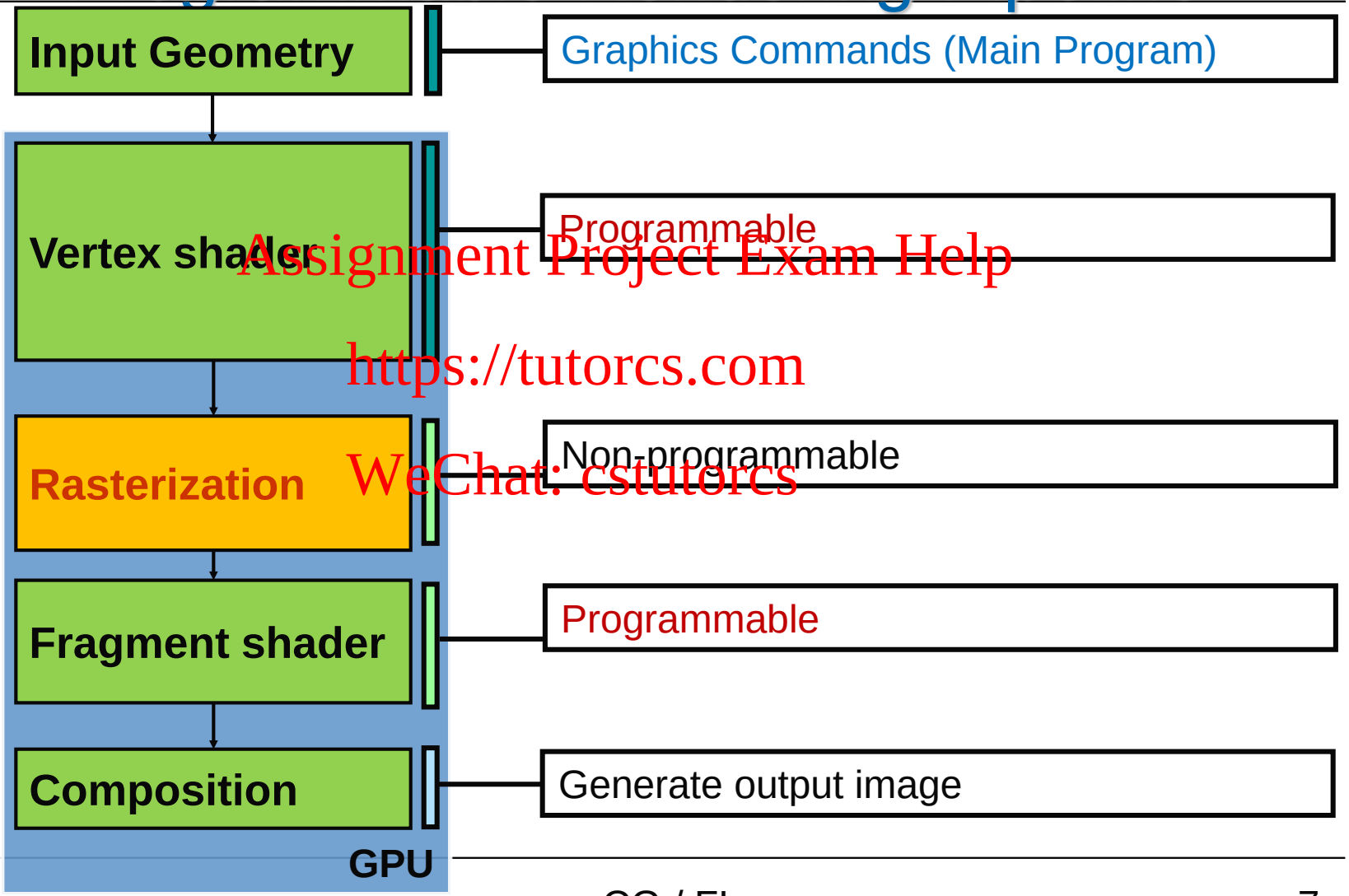
GPU Programming

- Graphics processing unit (GPU)
  - Typically comprises hundreds to thousands of processors
  - Process graphics primitives in parallel



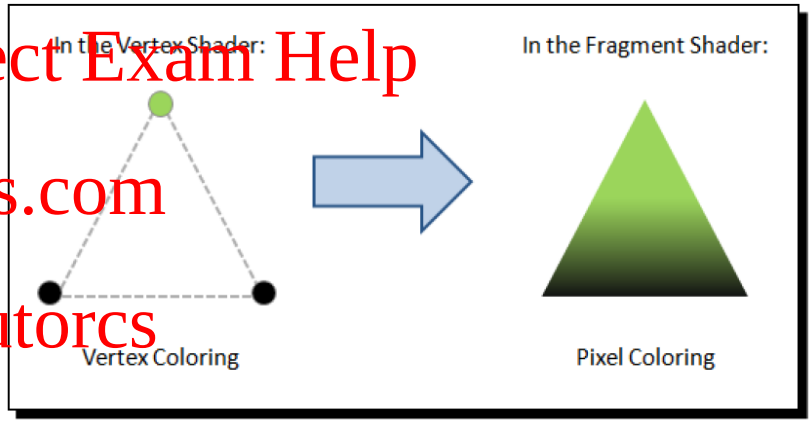
- Programmable rendering pipeline
  - Vertex shader and Fragment shader are programmable
  - GPU programming is also called shader programming

A Simplified View of Programmable Rendering Pipeline



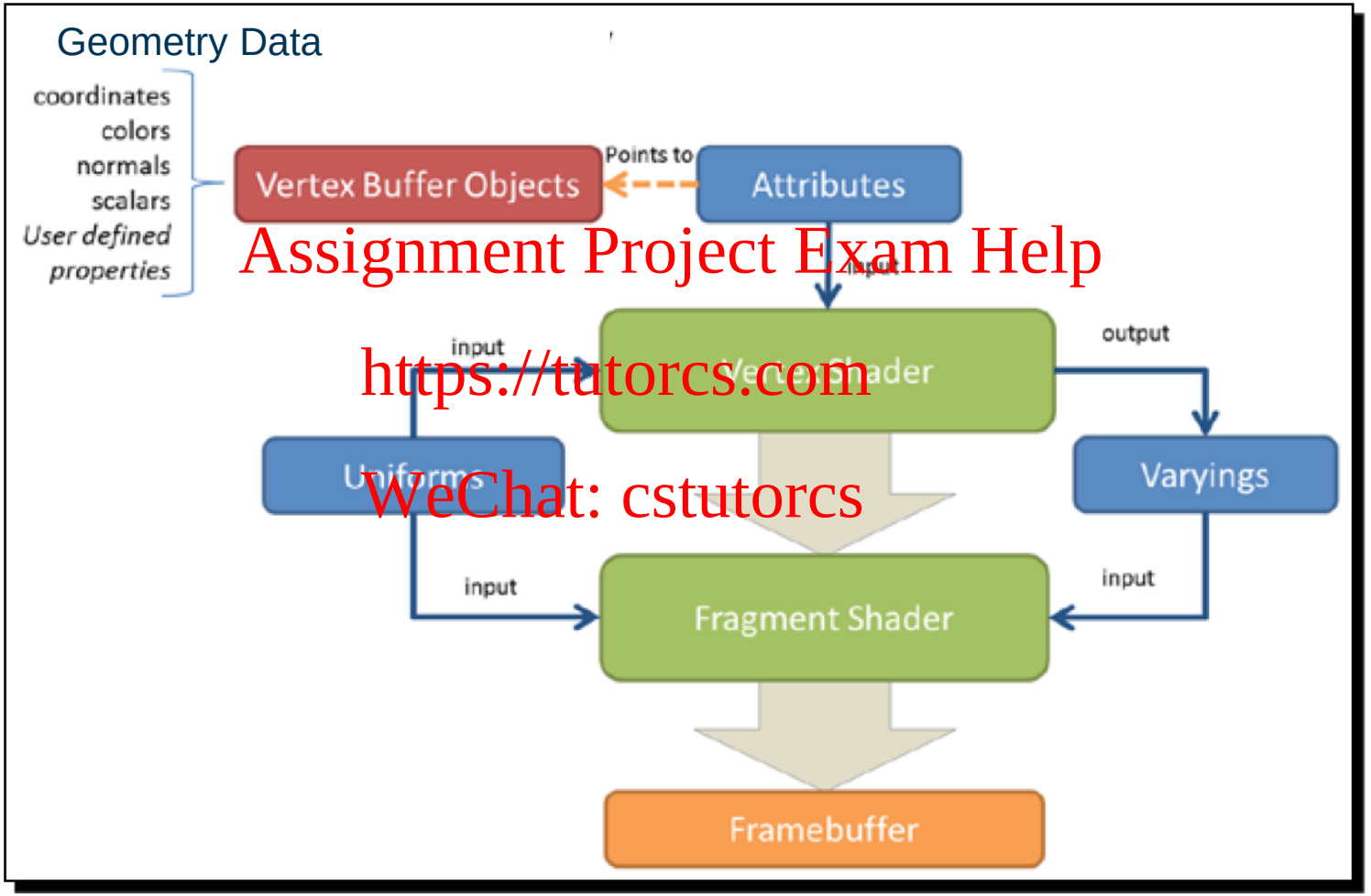
Shaders

- Vertex shader
  - Manipulates per-vertex data such as vertex coordinates, normals, colors, and texture coordinates.
- Fragment shader
  - Deals with surface points for processing
  - Main goal: calculate colour for each pixel that will display on the screen.



- Rasterization process: a black box (non-programmable), generates fragments from outputs of vertex shader

WebGL Rendering and Data Flow



Data Structures

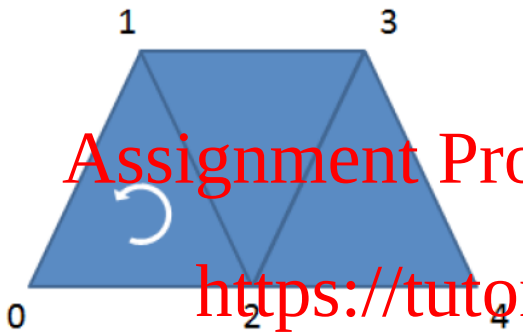
- **Vertex Buffer Objects (VBOs):** contain the data that WebGL requires to describe the geometry that is going to be rendered.
- **Index Buffer Objects (IBOs):** contain integers that use as references pointing to data in VBOs, in order to enable the reuse of the same vertex

- **Attributes**, **uniforms**, and **varyings** are the three different types of variables that you will find when programming with shaders.

  - **Attributes**: input variables used in the vertex shader, e.g. vertex coordinates, colour, normal. (dynamic)
  - **Uniforms:** input variables available for the vertex shader and the fragment shader, e.g. light position. (static)
  - **Varyings:** are used for passing data from the vertex shader to the fragment shader.

Define a Geometry

Vertex and Indices



coordinates

Vertex array = [0,0,0, 20,20,0, 0,10,20, 20,30,0, 0,30,20, 20,0,0, 30,20,40,0]

Triangle 1 Triangle 2 Triangle 3

Setup WebGL Buffer Objects

- Define geometry in the main program (Javascript):

```
var vertices = [-50.0, 50.0, 0.0, -50.0,-50.0, 0.0, 50.0,-
50.0, 0.0, 50.0, 50.0, 0.0];
```

- WebGL Buffer Object:


```
var myBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, myBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices),
gl.STATIC_DRAW); // Apply to myBuffer, since WebGL is a state
machine
gl.bindBuffer(gl.ARRAY_BUFFER, null); // Unbind a buffer
```

- Note:

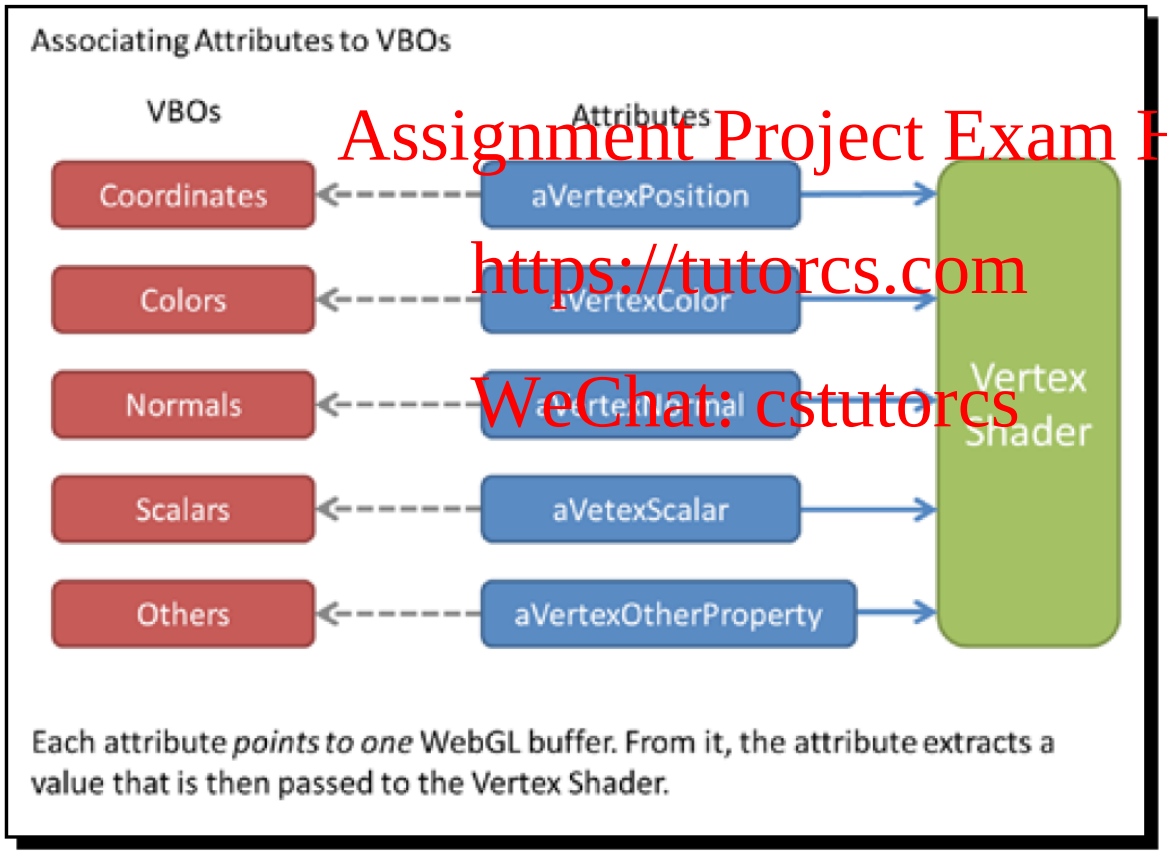
  Vertex data: `gl.ARRAY_BUFFER` **(for VBOs)**

  Index data: `gl.ELEMENT_ARRAY_BUFFER` **(for IBOs)**

  WebGL Typed Arrays: `Int8Array, Uint8Array, Int16Array, Uint16Array, Int32Array, UInt32Array, Float32Array, and Float64Array.`

Associate Attributes to VBOs

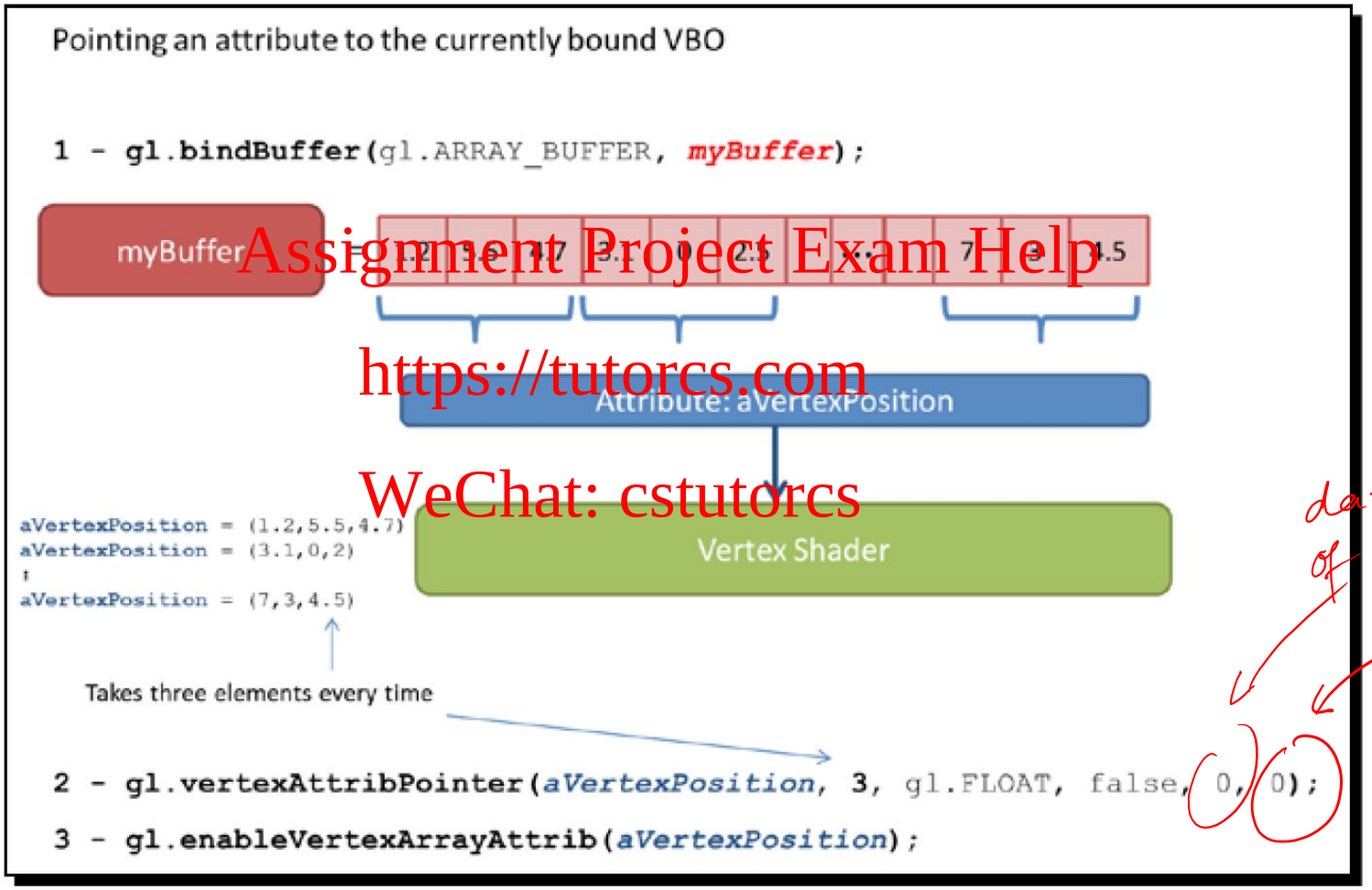
- Each vertex shader attribute will refer to one and only one buffer, which stores input modelling data



Steps:

1. Bind a VBO
2. Point an attribute to the currently bound VBO
3. Enable the attribute

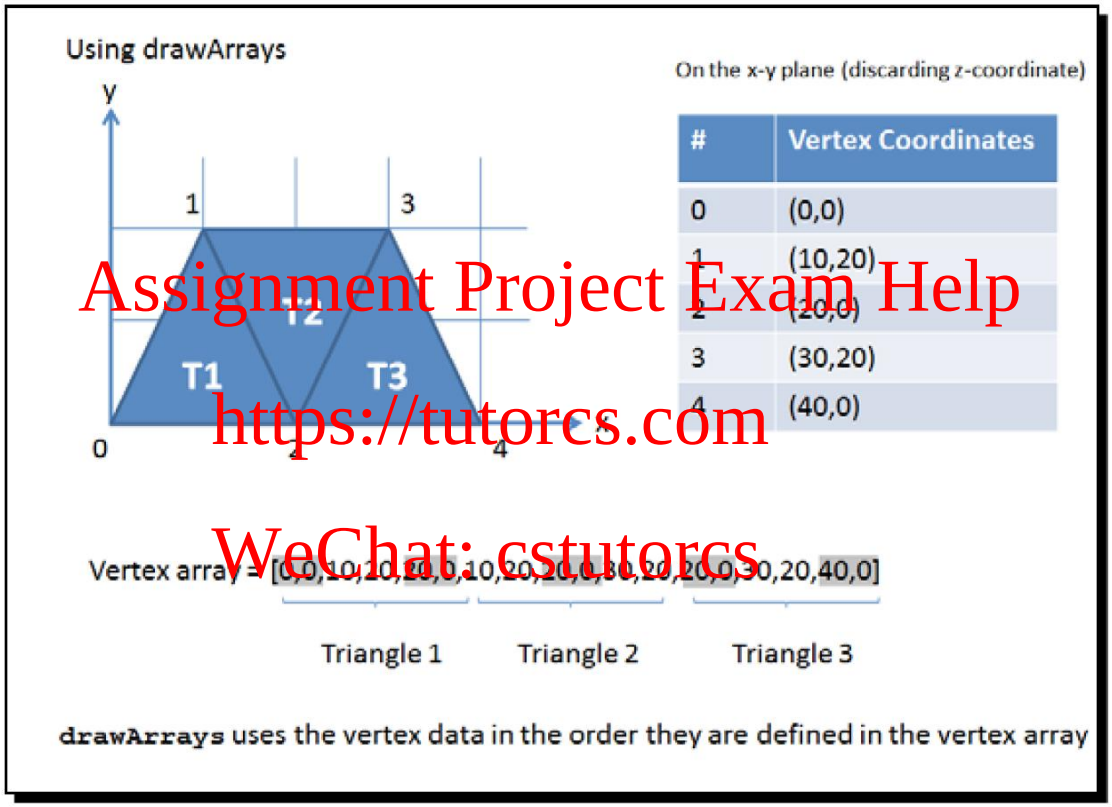
Mapping VBO data to Vertex Shader



Rendering – Ask the shaders to work

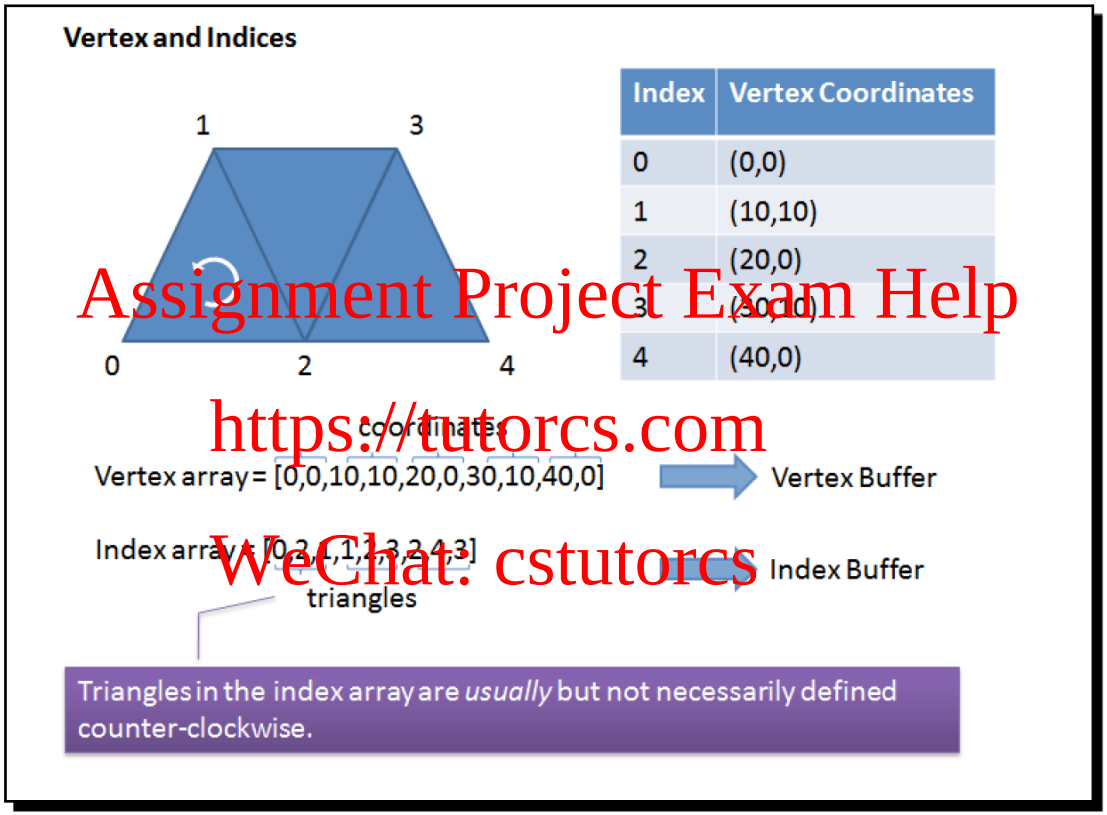
- Rendering can start, once we have defined VBOs and have mapped them to the corresponding vertex shader attributes.
- Functions **drawArrays** and **drawElements** are used for writing on the framebuffer:
  - **drawArrays** uses vertex data in the order in which it is defined in the buffer to create the geometry.
  - In contrast, **drawElements** uses indices to access the vertex data buffers and create the geometry.
  - **Mode:** Represents the type of primitive that we are going to render. Possible values for mode are:
    `gl.POINTS, gl.LINE_STRIP, gl.LINE_LOOP, gl.LINES, gl.TRIANGLE_STRIP, gl.TRIANGLE_FAN, and gl.TRIANGLES`
  - Note that only enabled arrays are used.

Using drawArrays



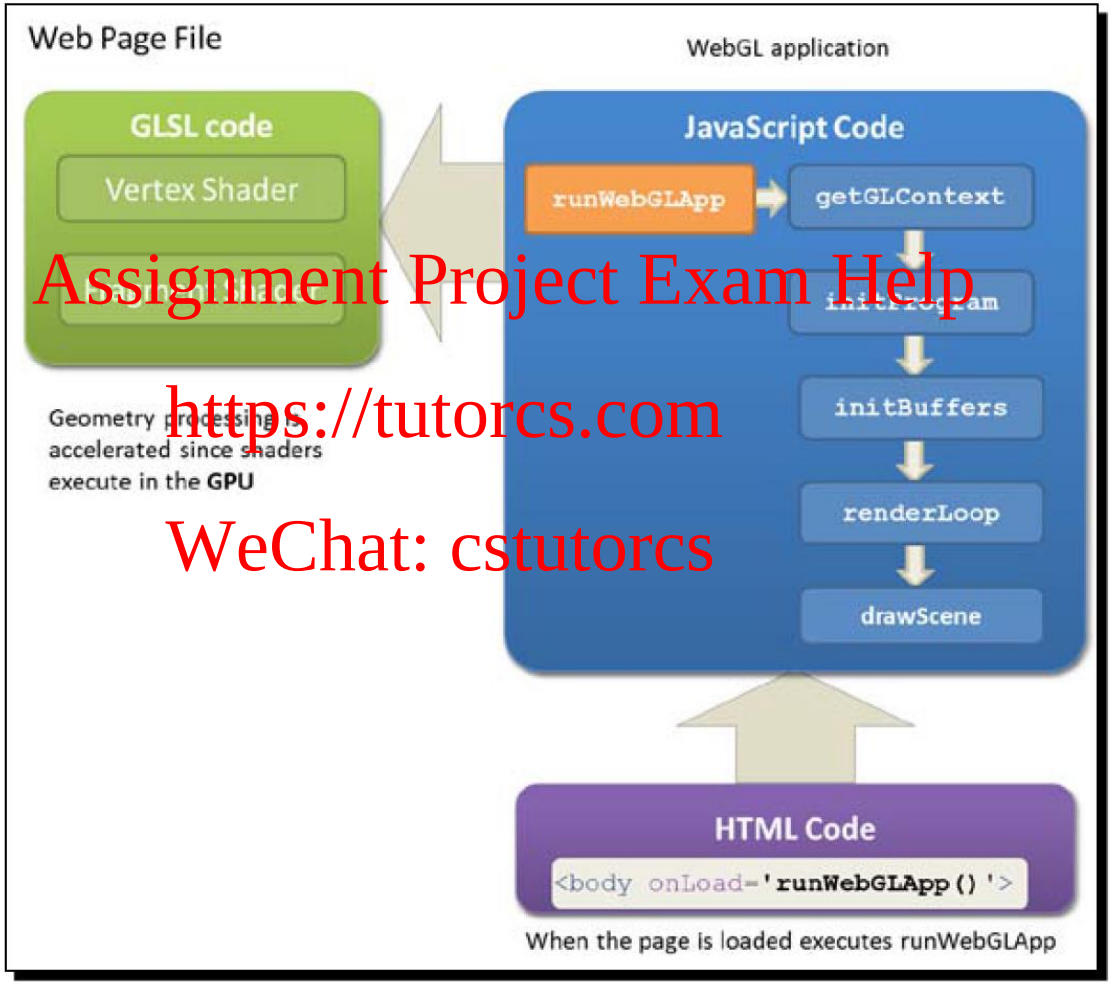
- gl.drawArrays(Mode, First, Count)
  First: starting element in the enabled arrays;
  Count: number of elements to be rendered.

Using drawElements



- gl.drawElements(Mode, Count, Type, Offset)
- Note: We need at least two buffers: a VBO and an IBO.
  Type: UNSIGNED_BYTE or UNSIGNED_SHORT.

Typical WebGL Program Structure



Example – VBO and Vertex Shader Attribute

```
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 vertex data

Associate vertex data with VBO

```
// Bind the buffer object to target
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
// Write data into the buffer object (modified once and used many times)
gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
```



```
var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
if (a_Position < 0) {
  console.log('Failed to get the storage location of a_Position');
  return -1;
}
// 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);
```

Define vertex shader attribute and Associate data from VBO

Shader – Draw from vertex data

Ask shaders to draw

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


Vertex & Fragment shaders

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

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


Each instruction applies in parallel to each element of the input data.

Involve Per-Vertex Attribute

```
function initVertexBuffers(gl) {
  var verticesColors = new Float32Array([
    // Vertex coordinates and color
     0.0,  0.5,  1.0,  0.0,  0.0,
    -0.5, -0.5,  0.0,  1.0,  0.0,
     0.5, -0.5,  0.0,  0.0,  1.0,
  ]);
  var n = 3;
```

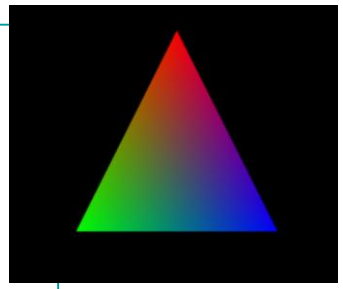
One buffer contains both vertices and colors

```
// Bind the buffer object to target
gl.bindBuffer(gl.ARRAY_BUFFER, vertexColorBuffer);
gl.bufferData(gl.ARRAY_BUFFER, verticesColors, gl.STATIC_DRAW);
```



```
var FSIZE = verticesColors.BYTES_PER_ELEMENT;
//Get the storage location of a_Position, assign and enable buffer
var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
if (a_Position < 0) {
  console.log('Failed to get the storage location of a_Position');
  return -1;
}
gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, FSIZE * 5, 0);
gl.enableVertexAttribArray(a_Position);  // Enable the assignment of the buffer object

// Get the storage location of a_Position, assign buffer and enable
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 * 5, FSIZE * 2);
gl.enableVertexAttribArray(a_Color);  // Enable the assignment of the buffer object
```



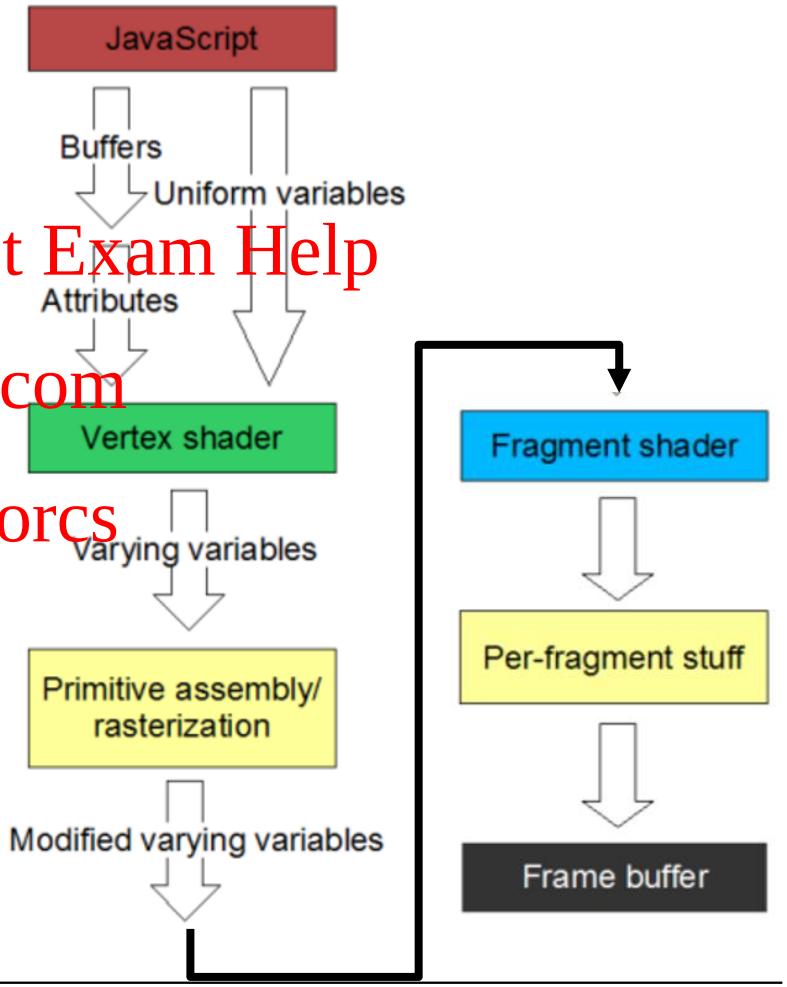
Render vertex data with attributes

Draw by shaders

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

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

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



Reference

Reference: WebGL Programming Guide [Ch. 3]