WebGL an intro

1) Creating a Canvas

```
<!doctype html>
<html>
          <head>
               <title>A blank canvas</title>
               <style>
                         body{ background-color: LightSkyBlue ; }
                         canvas{ background-color: white; }
                 </style>
         </head>
         <body>
                 <canvas id="my-canvas" width="400" height="300">
              Your browser does not support the HTML5 canvas element.
                 </canvas>
         </body>
</html>
```

2) Getting a context from the canvas

```
<!doctype html>
<html>
<head> <title>WebGL Context</title>
    <style> body { background-color: grey;}
        canvas { background-color: white;  }  </style>
    <script>
        window.onload = setupWebGL;
        var gl = null;
        function setupWebGL() {
            var canvas = document.getElementById("my-canvas");
            try {
                gl = canvas.getContext("experimental-webgl");
            } catch (e) {}
            if (gl) {
                //set the clear color to red
                gl.clearColor(1.0, 0.0, 0.0, 1.0);
                gl.clear(gl.COLOR_BUFFER_BIT);
            } else alert("Error: Your browser does not support WebGL.");
        }
    </script>
</head>
<body> <canvas id="my-canvas" width="400" height="300”></canvas> </body>
</html>
```

Context

- To draw inside of a canvas element you have to specify the api
- two canvas contexts: "2D" and "webgl”
- To obtain a context, call the canvas method getContext
- takes a context name as a first parameter and an optional second argument.
  - First argument context name "webgl” or "experimental-webgl".
  - The optional second argument can contain buffer settings and may vary by browser implementation.

3) Drawing something

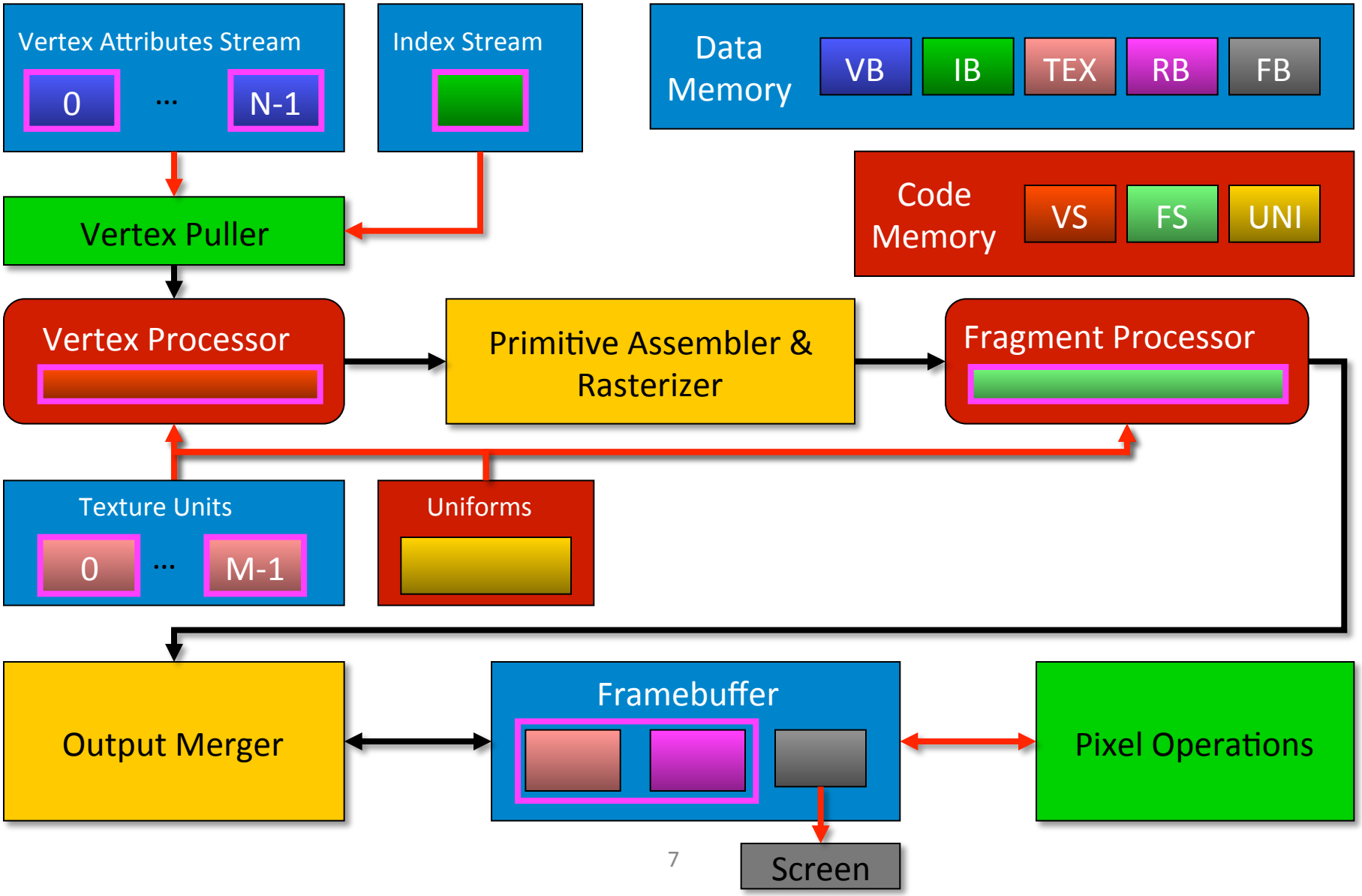
```
<!doctype html>
<html>
<head>
    <title>Two Triangles</title>
    <style> body { background-color: grey; }
        canvas {  background-color: white; } </style>
    <script id="shader-vs" type="x-shader/x-vertex”> . . . </script>
    <script id="shader-fs" type="x-shader/x-fragment"> . . . </script>
    <script>
        function initWebGL() { }
        function setupWebGL() {  }
        function initShaders() {   }
        function makeShader(src, type) {  }
        function setupBuffers() {  }
        function drawScene() { }
    </script>
</head>

<body onload="initWebGL()">
    <canvas id="my-canvas" width="400" height="300">
        Your browser does not support the HTML5 canvas element.
    </canvas>
</body>
</html>
```

Drawing

- Much more complicated
  - Need to know the pipeline
  - Need to setup buffers and program for the pipeline stages
  - Need to feed the buffer into the pipeline to actually draw something

OpenGL|ES 2.0: Architecture (simplified)



OpenGL|ES 2.0: API Structure

Pipeline Configuration & Data

- **Context**
  - Capabilities Query
  - Error Query
  - Creation / Activation / Destruction not part of specifications (EGL for this)
- **Objects**
  - Resources creation / edit / bind / destruction
  - Data Set & Get
- **Configurable Stages**
  - Enable / Disable
  - Constants / Parameters Set & Get
- **Programmable Stages**
  - Accept user-defined programs (shaders)

Execution

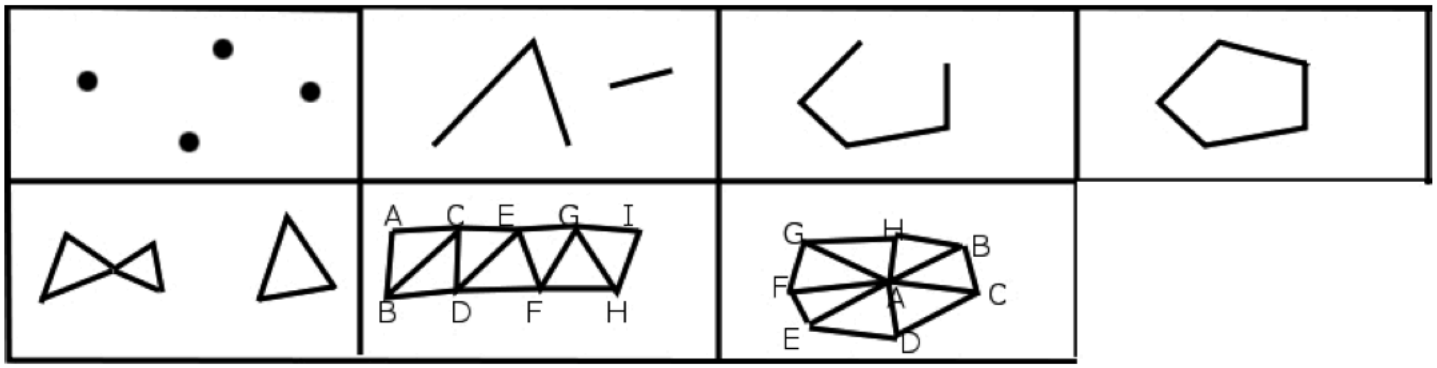
- **Vertex Pipeline**
  - Geometric primitives draw (activate Vertex Puller)
- **Pixel Pipeline**
  - Framebuffer clear & readback (activate Pixel Operations)
- **Synchronization**
  - Command Buffer flush & wait

The Drawing Buffers

- A *buffer* is a block of memory that can be written to and read from, and temporarily stores data.
- The color buffer holds color information—red, green, and blue
- The depth buffer stores information on a pixel's depth component (z-value)

Primitive Types

- *Primitives* are the graphical building blocks
- In WebGL, there are three primitive types: points, lines and triangles
- seven ways to render them: POINTS, LINES, LINE_STRIP, LINE_LOOP, TRIANGLES, TRIANGLE_STRIP, and TRIANGLE_FAN



Vertex Data

- All data associated with a vertex needs to be streamed (passed along) from the JavaScript API to the Graphics Processing Unit (GPU).
- create vertex buffer objects (VBOs) that will hold vertex attributes such as position, color, normal, and texture coordinates.
- These vertex buffers are then sent to a **shader** program that can use and manipulate the passed-in data.
- Using shaders instead of having fixed functionality is central to WebGL

Vertex Buffer Objects (VBOs)

- VBO stores data about a particular attribute of vertices.
- Can be position, color, a normal vector, texture coordinates, or something else.
- A buffer can also have multiple attributes interleaved

```
var myBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, myBuffer);
void bufferData(GLenum target, ArrayBuffer data, GLenum
usage)
```

3.1) initWebGL

```
var gl = null,
    canvas = null,
    glProgram = null,
    fragmentShader = null,
    vertexShader = null;
var vertexPositionAttribute = null,
    trianglesVerticeBuffer = null;

function initWebGL() {
    canvas = document.getElementById("my-canvas");
    try {
         gl = canvas.getContext("webgl") ||
            canvas.getContext("experimental-webgl");
    } catch (e) {}
    if (gl) {
        setupWebGL();
        initShaders();
        setupBuffers();
        drawScene();
    } else {
        alert("Error: Your browser does not support WebGL.");
    }
}
```

3.2) setupWebGL

```
function setupWebGL() {
            //set the clear color to a shade of green
            gl.clearColor(0.1, 0.5, 0.1, 1.0);
            gl.clear(gl.COLOR_BUFFER_BIT);
        }
```

3.3) initShaders

```
function initShaders() {
            //get shader source
            var fs_source = document.getElementById('shader-fs').innerHTML,
                vs_source = document.getElementById('shader-vs').innerHTML;
            //compile shaders
            vertexShader =   makeShader(vs_source, gl.VERTEX_SHADER);
            fragmentShader = makeShader(fs_source, gl.FRAGMENT_SHADER);
            //create program
            glProgram = gl.createProgram();
            //attach and link shaders to the program
            gl.attachShader(glProgram, vertexShader);
            gl.attachShader(glProgram, fragmentShader);
            gl.linkProgram(glProgram);
            if (!gl.getProgramParameter(glProgram, gl.LINK_STATUS)) {
                alert("Unable to initialize the shader program.");
            }
            //use program
            gl.useProgram(glProgram);
        }
```

3.3.1) The Shaders

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

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

3.4) makeShader

```
function makeShader(src, type) {
            //compile the vertex shader
            var shader = gl.createShader(type);
            gl.shaderSource(shader, src);
            gl.compileShader(shader);
            if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
                alert("Error compiling shader: " +
                    gl.getShaderInfoLog(shader));
            }
            return shader;
        }
```

3.4) setupBuffers

```
 function setupBuffers() {
            var triangleVertices = [
                //left triangle
                -0.5, 0.5, 0.0,
                 0.0, 0.0, 0.0,
                -0.5, -0.5, 0.0,
                //right triangle
                0.5, 0.5, 0.0,
                 0.0, 0.0, 0.0,
                0.5, -0.5, 0.0
];
            trianglesVerticeBuffer = gl.createBuffer();
            gl.bindBuffer(gl.ARRAY_BUFFER, trianglesVerticeBuffer);
            gl.bufferData(gl.ARRAY_BUFFER, new
Float32Array(triangleVertices), gl.STATIC_DRAW);
        }
```

Drawing

- In the drawScene,
- assign the vertex shader attribute aVertexPosition's location to a variable— vertexPositionAttribute.
- enable array data for the attribute and bind our array to the current buffer.
- assign trianglesVerticeBuffer data to the value stored in our vertexPositionAttribute variable.
- tell the vertexAttribPointer that our data has three components (x,y,z) per vertex.
- Finally, call drawArrays with a primitive type of gl.TRIANGLES, the starting vertex and the total number of vertices to render.

3.5) drawScene

```
 function drawScene() {
   vertexPositionAttribute = gl.getAttribLocation(glProgram,
                    "aVertexPosition");
   gl.enableVertexAttribArray(vertexPositionAttribute);
   gl.bindBuffer(gl.ARRAY_BUFFER, trianglesVerticeBuffer);
   gl.vertexAttribPointer(vertexPositionAttribute, 3,
                    gl.FLOAT, false, 0, 0);
   gl.drawArrays(gl.TRIANGLES, 0, 6);
}
```

Clip coords

- WebGL has a default clip volume centered at the origin (0,0,0) and +/- 1 along x,y,z axes.
- The clip volume defines the (x,y,z) points that will be rendered by the fragment shader.
- Any fragment (pixel) within the clipping volume is rendered, and points outside of it are discarded (clipped).
- The vertex shader transforms points to a final gl_Position.
- a clip test is done on each fragment, with those falling within the clip volume continuing on to the fragment shader.

Viewport

- The viewport defines where the origin (lower-left) point (x,y) to render on the canvas should be located, and
- what width and height of the canvas to render onto. We set the viewport with the API call: void viewport(GLint x, GLint y, GLsizei width, GLsizei height);
- Setting the origin to (0, 0) and the width and height equal to the canvas dimensions will fill the entire canvas. This is done with the following code:
- gl.viewport(0, 0, canvas.width, canvas.height);

Adding Color

- We need to
- Add the color data (a new buffer)
- Change the shader to use the data
- Change the drawing procedure

Color Data

```
 var triangleVerticeColors = [
                    //left triangle
                     1.0, 0.0, 0.0,
                     1.0, 1.0, 1.0,
                     1.0, 0.0, 0.0,

                    //right triangle
                    0.0, 0.0, 1.0,
                     1.0, 1.0, 1.0,
                    0.0, 0.0, 1.0,
                ];

trianglesColorBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, trianglesColorBuffer);
gl.bufferData(gl.ARRAY_BUFFER,
             new Float32Array(triangleVerticeColors), gl.STATIC_DRAW);
```

ColorShaders

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

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


<script id="shader-fs" type="x-shader/x-fragment">
            varying highp vec4 vColor;
            void main(void) {
               gl_FragColor = vColor;
            }
</script>
```

Color DrawScene

```
function drawScene()
{
 vertexPositionAttribute = gl.getAttribLocation(glProgram,"aVertexPosition");
 gl.enableVertexAttribArray(vertexPositionAttribute);
 gl.bindBuffer(gl.ARRAY_BUFFER, trianglesVerticeBuffer);
 gl.vertexAttribPointer(vertexPositionAttribute, 3, gl.FLOAT, false, 0, 0);

 vertexColorAttribute = gl.getAttribLocation(glProgram, "aVertexColor");
 gl.enableVertexAttribArray(vertexColorAttribute);
 gl.bindBuffer(gl.ARRAY_BUFFER, trianglesColorBuffer);
  gl.vertexAttribPointer(vertexColorAttribute, 3, gl.FLOAT, false, 0, 0);

 gl.drawArrays(gl.TRIANGLES, 0, 6);
}
```

Animating

- We need to:
  - Continuously redraw
  - Change the geometry buffer at each frame

Animating

- requestAnimationFrame

```
initShaders();
setupBuffers();
(function animLoop(){
  setupWebGL();
  setupDynamicBuffers();
  drawScene();
  requestAnimFrame(animLoop, canvas);
})();
```

Animating

```
function setupDynamicBuffers()
{
    //limit translation amount to -0.5 to 0.5
 var x_translation = Math.sin(angle)/2.0;
var triangleVertices = [ //left triangle
-0.5 + x_translation, 0.5,
 0.0, 0.0 + x_translation, 0.0, 0.0,
-0.5 + x_translation, -0.5, 0.0,
//right triangle
0.5 + x_translation, 0.5, 0.0,
0.0 + x_translation, 0.0, 0.0,
0.5 + x_translation, -0.5, 0.0
];

angle += 0.01;
trianglesVerticeBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, trianglesVerticeBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices),
gl.DYNAMIC_DRAW);
}
```

What is WebGL

- Actually what *will* be (standardization in progress)
- *Specification* owned by the Khronos Group
- Supported (and defined) by all major web browsers devs

- JavaScript bindings to OpenGL|ES 2.0
- Almost 1-to-1 mapping, some modifications to meet JS developers' habits and security issues

- Enables home computers and mobile devices to *natively* access 3D content directly from web pages (no external plug-ins)

OpenGL|ES 2.0

- Stripped version of OpenGL, focused on Embedded Systems

- Programmable : NO fixed function pipeline
  - Immediate mode (glBegin / End) → Use vertex / index buffers
  - Transform Stage (matrix stacks) → Explicit shaders uniforms
  - Lighting (lighting model, light sources, materials) → Lighting computation through shaders code
  - Named vertex attributes (glVertexPointer(…), …) → Generic attributes through glVertexAttribPointer(index, …)

- Data entirely resides on GL resources (buffers, textures)
  - *Buffer Centric* API
- Restrictions
  - Texture formats, data types, shading language limitations, ...