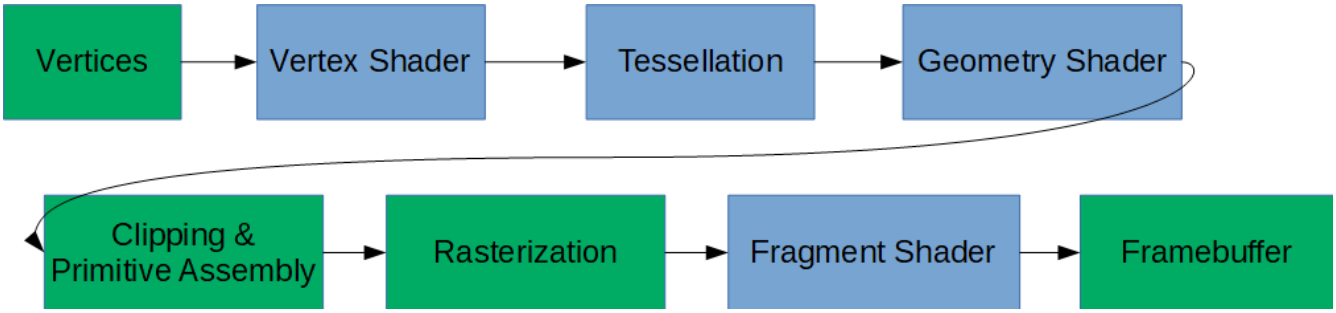
Introduction to WebGL

Notes for me:

- webglstats.com
- webglreport.com
-

The *Modern* Graphics Pipeline



The blocks in blue are *programmable* stages of the pipeline. The green blocks are provided by the rendering API; usually performed on the GPU directly.

A *shader* is a program that runs on the GPU and outputs data specific to the programmable stage in which it is executed. For example, a vertex shader takes as input a vertex and outputs a vertex (transformed by view and projection, for example).

Vertices

Vertices are prepared by the application and passed to the rendering pipeline in some kind of array or stream of data. These vertices can be sent one time to the GPU and reused over many different frames; they do not have to be sent once per frame...we'll learn more about this later.

The primitives (usually triangles) formed from the vertices are also provided by the application during this stage.

Vertex Shader (required)

Programmable stage during which each vertex is (usually) transformed from a local coordinate system to a projected coordinated system. Examples of operations that can take place (per-vertex) during vertex shading:

- Compute the color (lighting)
- Transform from local to world coordinates

- Transform from world to camera view
- Transform by projection (e.g., perspective)

There is a one-to-one mapping of an input vertex to an output vertex. This shader does not modify the source data, the output of the shader is a new vertex.

Tessellation (optional)

Programmable stage during which patches of vertices are transformed into a new patch. This shader may increase or decrease the number of primitives contained within the patch.

Geometry Shader (optional)

Similar to a tessellation shader, the geometry shader accepts a primitive (a triangle) and outputs zero or more primitives. Like the vertex shader, the geometry shader can transform the vertices in the primitive, in addition to outputting new primitives.

Clipping & Primitive Assembly

Vertices are assembled into a sequence of primitives. These primitives are then processed for clipping by removing complete or parts of primitives that are not visible after the viewport transformation are clipped. Additionally, primitives that are not facing the viewport space are removed from processing. The remaining primitives (or parts of primitives) are sent to the rasterization stage.

Rasterization

Primitives are processed into their fundamental fragments (usually pixels) in the viewport. A fragment contains a number of pieces of data that are passed into the next stage, including:

- Screen space position
- Interpolated color
- Interpolated normal
- interpolated u,v coordinates

Either during this stage or as pre-processing before the fragment shader, a *z-test* is performed to see if the fragment is potentially visible. If it fails the z-test, the fragment is not sent to the fragment shader for further processing.

Fragment Shader (optional, but really required)

Programmable stage used to determine the framebuffer properties for each fragment. The following are examples of outputs from this shader:

- Color for each channel (red, green, blue, alpha)

- Depth
- Normal

During this stage data from textures or other (previously computed) framebuffers can be combined to determine the output values for each fragment.

Framebuffer

This is the storage for each fragment (pixel) from the pipeline. Typically we think of the data in the framebuffer as the RGB color to be displayed, but it can be anything an application wants. We'll learn more about this as the semester moves along.

What is a Shader?

Again, a shader is a program that executes on the graphics hardware, the GPU, rather than on the CPU. The language used to write the shaders is based on the C language, but customized and extended for the purpose of graphics rendering.

GPUs are designed with lot's of vector processing units, hundreds and thousands of them; probably soon tens of thousands! Each of these vector processing units can execute a shader in parallel. In reality the vector processing units are grouped together, into say blocks of 32 or 64, and a single shader program is executed on all of these blocks at the same time, but of course, over different data. This means a high degree of parallelism is taking place on the GPU.

Here is most simple vertex shader possible:

```
void main(void)
{
      gl_Position = ftransform();
}
```

Here is the most simple fragment shader possible:

```
void main(void)
{
      gl_FragColor  = glColor;
}
```

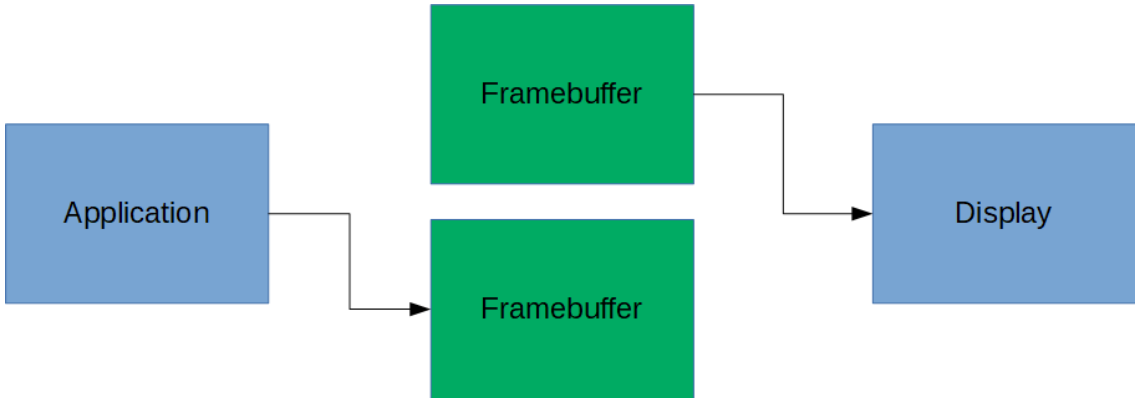
This is just a cursory look, we'll learn a lot more very soon.

GLSL Shader Reference: http://www.shaderific.com/glsl/

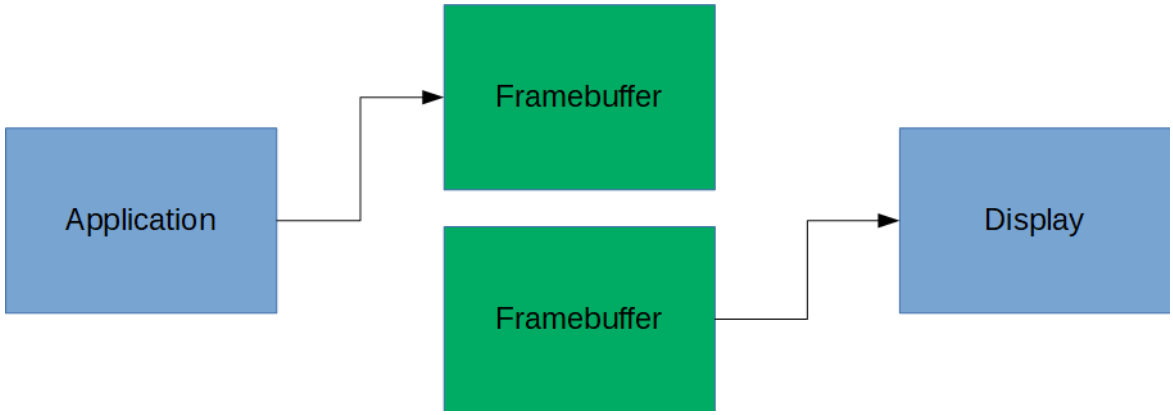
Double Buffering

Because it isn't (usually) possible for a program to generate a full displayable frame of data as it is being drawn to the screen, it is necessary to have two framebuffers. One framebuffer for display, while another framebuffer is being written. When the buffer being written to is complete, the write and display buffers are swapped (just a flag, data isn't copied). When the framebuffers are swapped, this is known as a *page flip*.

Frame #1



Frame #2



Double buffering means that all of the pixels in a frame are re-written each frame, rather than only modifying the pixels that have changed. For 2D applications this may not be optimal, but for 3D games, almost all pixels change every frame.

Triple Buffering: A program may finish writing to the backbuffer some number of milliseconds before the framebuffers can be swapped (because of the refresh rate of the display). The time from once it finishes writing until the ability to swap buffers is wasted time that could have been used to start working on the next frame. In triple buffering, there is a third buffer that the application can switch to and start writing results into while waiting for the just completed framebuffer is used for displayed. This allows a program to achieve a higher framerate than is possible with a double buffer scheme.

Introduction to OpenGL (and WebGL)

A low level rendering API intended for use in interactive applications. The following are aspects of OpenGL:

- OpenGL is an open standard, maintained by the Khronos Group (a non-profit industry consortium)
- OS independent; “easily” portable to various computing platforms
- Hardware platform independent. Can be implemented entirely in software, but intended for hardware.
- Language independent (in a sense). The actual API is defined in C, but bindings to many languages exist.
- History
  - OpenGL 1.0 (1992). Based on SGI “Iris gl”
  - OpenGL 2.0 (2004). First inclusion of GLSL.
    - OpenGL ES 2.0 (WebGL) 2007. Primarily based on OpenGL 2.0, but eliminates most of the fixed-function pipeline, having only a programmable pipeline (which is similar to OpenGL 3.0).
  - OpenGL 3.0 (2008). Deprecated fixed-function API.
  - OpenGL 4.0 (2010)
  - OpenGL 4.3 (2012). Compute Shaders, full compatibility with OpenGL ES 3.0.
    - OpenGL ES 3.0 (2012).
  - OpenGL 4.6 (2017) – Current specification.

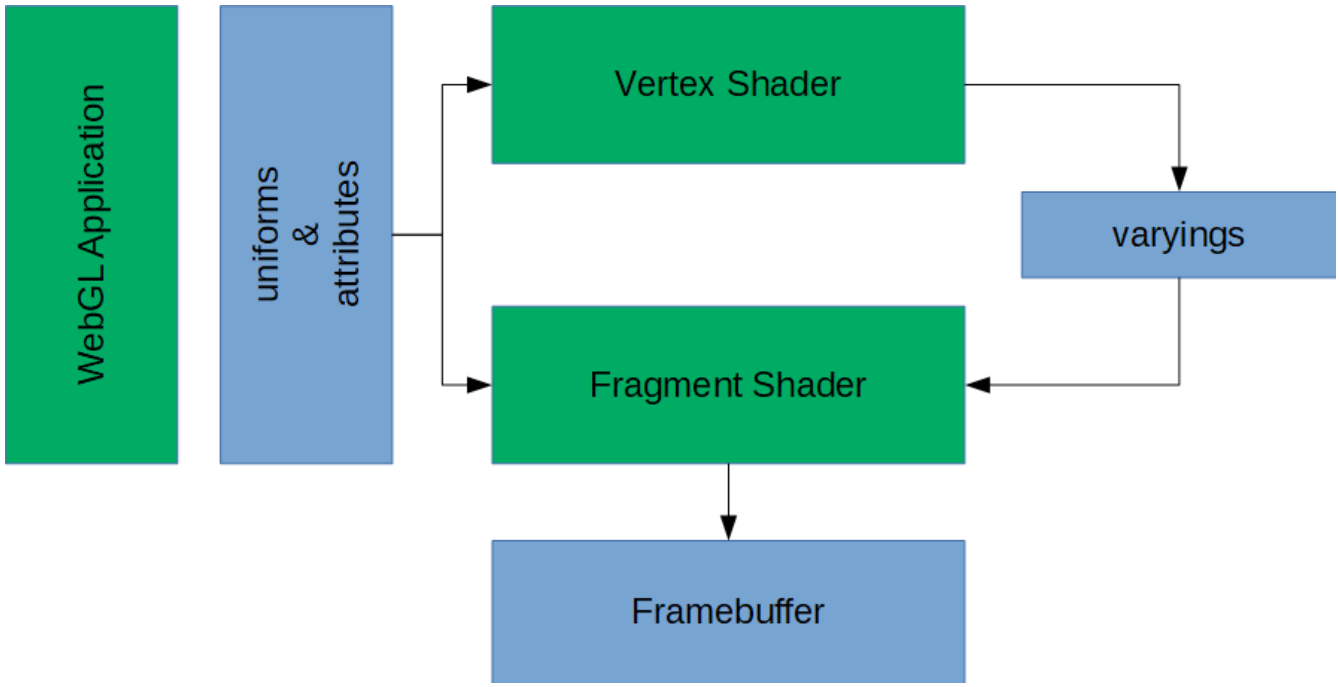
Key Features

- Programmable pipeline
- Double-buffering
- Z-Buffering
- Textures (multiple image formats)
  - Mip-mapping
- Buffer objects
- Off-screen rendering
- Extensible architecture
- (deprecated) lighting
- (deprecated) texture mapping
- (deprecated) alpha blending
- (deprecated) anti-aliasing

The WebGL Pipeline

http://www.webgltutorials.org/simplified-webgl-pipeline.html

WebGL follows the graphics pipeline as illustrated earlier. The diagram below provides additional detail specific to how WebGL specifically handles some aspects of the pipeline.



The Framebuffer is then rendered into an HTML `<canvas>` element.

- **uniform** (per primitive) : Constant for the entire draw call; think of it like a constant.
- **attribute** (per vertex) : Things like positions, colors, normals, uv coordinates; the kind and number of these may change on a per primitive call.
- **varying** (per fragment) : Things like position, color, normal, uv coordinates; vary on a per-fragment basis. Interpolated (by the GPU) from the per-vertex values.

WebGL automatically takes care of double buffering. The buffers are swapped during the called to `requestAnimationFrame()`. In OpenGL programs, the `SwapBuffers()` function is called to tell the GPU when to swap the buffers. This call is being performed inside of the `requestAnimationFrame()` instead.

A First WebGL Program

Steps in a basic WebGL program:

1. Define an HTML <canvas> element
2. Obtain the WebGL context
3. Prepare raw data (e.g., geometry)
4. Prepare shaders
5. Prepare buffer objects
6. Specify buffer object attributes
7. Request animation frame
    8. Reset framebuffer & depth buffer
    9. Draw primitives
    10. Return to Step 7

Step 1 : Define an HTML <canvas> element

```
<!DOCTYPE HTML>
<html lang = "en-US">
<head>
      <meta charset = "UTF-8">
      <title>WebGL - First</title>
</head>
<body>
      <canvas id = "canvas-main" width = "800" height = "600"></canvas>

      <script>
            var MySample = {};
      </script>
      <script src = "scripts/driver.js"></script>
</body>
</html>
```

Step 2 : Obtain the WebGL context

```
let canvas = document.getElementById('canvas-main');
let gl = canvas.getContext('webgl');
```

`getContext` returns a `WebGLRenderingContext`, which contains the WebGL API we use. Obtain the context one time and keep a reference to it for your whole program. We are only using WebGL 1.0, but a ‘webgl2’ context is available for WebGL 2.0.

Step 3 : Prepare raw data

The geometry and other data need to be stored in JavaScript arrays. Because the data needs to be in known layouts, the arrays need to be defined using specific data types, which is generally unusual for typical JavaScript programs.

Data are passed to WebGL in buffer objects. A primitive (such as a triangle) is a combination of two buffers. A vertex buffer (or VBO) and an index buffer. The vertex buffer defines all the unique vertices in the primitive, while the index buffer defines the indices of the vertices in the primitive.

The following defines the vertices and indices for a single triangle primitive:

```
let vertices = new Float32Array([
      0.0, 0.5, 0.0,
      0.5, 0.0, 0.0,
      -0.5, 0.0, 0.0
]);

let indices = new Uint16Array([ 0, 1, 2 ]);
```

The vertices are x, y, z triples that specify the position of the vertex. The indices are indexes into the vertex array that specify which vertex position to use for the primitive vertex. The index is per-triple (x,y,z). In other words, index 0 is for the first (x,y,z) position, then index 1 is for the second (x,y,z) position and so on.

Additionally, we are going to define unique colors at each of the vertices:

```
let vertexColors = new Float32Array([
      1.0, 0.0, 0.0,
      0.0, 1.0, 0.0,
      0.0, 0.0, 1.0
]);
```

The colors are RGB triples at each vertex; values range [0, 1].

Step 4 : Prepare buffer objects

Once the raw data is loaded into memory, the data needs to be transferred to WebGL by creating the various buffer objects. The memory for the buffer objects is held by the GPU memory, rather than in the main system RAM (unless have unified memory architecture). For our simple WebGL program we have three buffer objects: vertex, color, and index.

Vertex Buffer

```
let vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
gl.bindBuffer(gl.ARRAY_BUFFER, null);
```

There are two kinds of buffers that can be created. The first, `ARRAY_BUFFER`, is a buffer of actual things. The second, `ELEMENT_ARRAY_BUFFER`, is a buffer of indices into another buffer.

`STATIC_DRAW` is a hint to the API that the contents of the buffer are likely to be used often, but not change often. Other options include `STREAM_DRAW` (not used often) and `DYNAMIC_DRAW` (used and change often).

WebGL is a state machine, you set a state and that state is active until you change it. We use `bindBuffer` to change the state by specifying the buffer that subsequent commands are applied to. `bufferData` is called to create/initialize the WebGL buffer and transfer raw data.

Index Buffer

```
let indexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, indices, gl.STATIC_DRAW);
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, null);
```

Here we use `ELEMENT_ARRAY_BUFFER` because the index buffer indexes the vertex buffer created earlier.

Color Buffer

```
let vertexColorBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexColorBuffer);
gl.bufferData(gl.ARRAY_BUFFER, vertexColors, gl.STATIC_DRAW);
gl.bindBuffer(gl.ARRAY_BUFFER, null);
```

Step 5 : Prepare shaders

There are two shaders that must be prepared, a vertex and fragment shader. Similar to a C program, a shader has source, is compiled, and then linked into a larger context, or what is called a *program*.

Vertex Shader

```
let vertexShaderSource = `
      attribute vec4 aPosition;
      attribute vec4 aColor;
      varying vec4 vColor;
      void main()
      {
            gl_Position = aPosition;
            vColor = aColor;
      }`;

let vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, vertexShaderSource);
gl.compileShader(vertexShader);
console.log(gl.getShaderInfoLog(vertexShader));// for debugging
```

Fragment Shader

```
let fragmentShaderSource = `
      precision lowp float;
      varying vec4 vColor;
      void main()
      {
            gl_FragColor = vColor;
```

```
        }`;

    let fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
    gl.shaderSource(fragmentShader, fragmentShaderSource);
    gl.compileShader(fragmentShader);
```

Shader Program

Once the two vertex and fragment shaders are compiled, they need to be combined into an overall shader program that can be specified on a per draw call basis.

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

    gl.useProgram(shaderProgram);
```

The final step in the code above is to update the WebGL state to use the shader program for all subsequent draw calls. This state can be changed between every draw call if desired (but not recommended). You'll want to batch draw calls together to re-use shader programs as much as possible.

Precision Qualifiers

If you are going to use a float in a fragment shader, you have to specify the precision.

What in the world are the precision qualifiers?

- `highp` : the highest precision available
- `lowp` : the lowest precision available
- `mediump` : a precision between the lowest and highest available precision

For vertex shaders the following precisions are declared by default:

- `precision highp float;`
- `precision highp int;`
- `precision lowp sampler2D;`
- `precision lowp samplerCube;`

For fragment shaders the following precisions are declared by default:

- `precision mediump int;`
- `precision lowp sampler2D;`
- `precision lowp samplerCube;`

Step 6 : Specify buffer object attributes

In order for WebGL to know how to pass data from a buffer object into a shader, the attributes in the shader need to be associated with the data from the buffer. While doing this, the layout of the data in the buffer is specified. This is necessary because the data in the buffer is essentially completely user customizable, rather than having to conform to pre-defined data layouts.

```
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
let position = gl.getAttribLocation(shaderProgram, 'aPosition');
gl.vertexAttribPointer(position, 3, gl.FLOAT, false,
      vertices.BYTES_PER_ELEMENT * 3, 0);
gl.enableVertexAttribArray(position);

gl.bindBuffer(gl.ARRAY_BUFFER, vertexColorBuffer);
let color = gl.getAttribLocation(shaderProgram, 'aColor');
gl.vertexAttribPointer(color, 3, gl.FLOAT, false,
      vertexColors.BYTES_PER_ELEMENT * 3, 0);
gl.enableVertexAttribArray(color);

//
// This sets which buffer to use for the draw call in the render function.
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, buffers.indexBuffer);
```

Step 7 : Request animation frame

```
function animationLoop() {
      update();
      render();

      //
      // Only need to do if updating the view.  Otherwise, the
      // rendering will stay persistent.
      requestAnimationFrame(animationLoop);
}

requestAnimationFrame(animationLoop);
```

Step 8 : Reset framebuffer & depth buffer

```
gl.clearColor(
      0.392156862745098039215686274509803921568627450980392156862745098,
      0.584313725490196078431372549019607843137254901961,
      0.929411764705882352941176470588235294117647058824, 1.0);
gl.clearDepth(1.0);
gl.depthFunc(gl.LEQUAL);
gl.enable(gl.DEPTH_TEST);

gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

Step 9 : Draw Primitives

```
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, buffers.indexBuffer);
gl.drawElements(gl.TRIANGLES, data.indices.length, gl.UNSIGNED_SHORT, 0);
```

`bindBuffer` is needed to specify (change the WebGL state) which index buffer to use for the `drawElements` call.

`drawElements` is used to draw WebGL primitives. Right now we are drawing a single, simple triangle, but there are other primitives that can be drawn through this call.

Step 10 : Return to Step 7

This is already handled in the `animationLoop` function.

Additional WebGL

Specifying ‘uniform’ Parameters on Shaders

There is a whole family of gl.uniformXX API calls, one for each of the various data types you might want to send to a shader. In the example below, we are sending a matrix to the shader.

```
let uMatrix = […];
let location = gl.getUniformLocation(shader, ‘uThing’);
gl.uniformMatrix4fv(location, false, uMatrix);
```

The third parameter to the `gl.uniformMatrix4fv` is whether or not to transpose the matrix. Guess what, it must be `false` for WebGL; so deal with it!

A uniform can be set once per object, which is really once per drawElements call.

Link to the various types of uniforms calls…

https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/uniform

JavaScript to OpenGL Matrices

Due to the way JavaScript stores arrays, you need to perform a transpose on the JavaScript array before giving it to OpenGL for use. Consider the following translation matrix as defined using JavaScript code…

```
let matTranslate = [
      1,  0,  0, object.center.x,
      0,  1,  0, object.center.y,
      0,  0,  1, object.center.z,
      0,  0,  0, 1
];
```

We look at it and it all looks great. However, when OpenGL sees the matrix, it is going to interpret the first 4 values as the first column in the matrix. It will think of the matrix like this...

```
let matTranslate = [
      1,  0,  0, 0,
      0,  1,  0, 0,
      0,  0,  1, 0,
      object.center.x,  object.center.y,  object.center.x, 1
];
```

Because of this, before sending to OpenGL you need to transpose the matrix. Here you go…

```
//-----------------------------------------------------------------
//
// Transpose a matrix.
// Reference: https://jsperf.com/transpose-2d-array
//
//-----------------------------------------------------------------
function transposeMatrix4x4(m) {
```

```
    let t = [
        m[0], m[4], m[8], m[12],
        m[1], m[5], m[9], m[13],
        m[2], m[6], m[10], m[14],
        m[3], m[7], m[11], m[15]
    ];
    return t;
}
```