PIXELS GONE WILD: NOTES

ATTRIBUTES AND BUFFERS:

- In webGL data is to be defined in the Models.js file and each model(collection of data) is imported in the main.js file.
- All segments of data specified in a model are called attributes and are passed on to the vertex shader.
- We use buffers ie; a vertex buffer object(VBO) to store and pass data onto the vertex shader.
- Usually buffers contain things like positions, normals, texture coordinates, vertex colors, etc. although you're free to put anything you want in them.
- We use vertex array objects(VAO) to specify how these attributes are segmented throughout our buffer and lets the vertex shader interpret that data.

USAGE:

- VBO:
  - Create an array of floats to put data in it.
  - Create a VBO using `gl.createBuffer()`
  - Bind the VBO using `gl.bindBuffer()`
  - Pass data from the float array to the buffer using `gl.bufferData()`
- VAO:
  - Create a VAO using `gl.createVertexArray()`
  - Bind the VABO using `gl.bindVertexArray()`
  - For each data segment in the buffer:
    - Allocate a pointer to refer to each segment.
    - Enable the Pointer using `gl.enableVertexAttribArray(index)`
    - Set the Pointer attributes by using `gl.vertexAttribPointer(index, size, type, normalized, stride, offset)`
      - `index: index of the vertex attribute.`
      - `size: number of components per vertex attribute.`
      - `type: data type of each component.`
      - `normalized: boolean specifying whether values should be normalized.`
      - `stride: offset(in bytes) between beginning of consecutive vertex attributes.`
      - `offset: offset(in bytes) of the first component in the vertex array.`
  - While rendering we bind the VAO that is to be used using `gl.bindVertexArray()`
  - Lastly we specify the draw method `gl.drawArrays(mode, first, count)`
  - `mode: primitive to render.`
  - `first: starting index of array points.`
  - `count: number of indices to render.`
- EXAMPLE:

```
export class Triangle extends Model {
    setup() {
        const data = new Float32Array([
            -0.2, -0.2,  0.0, // vertex 1
             0.0,  0.0,  1.0, // color for v1
             0.2, -0.2,  0.0, // vertex 2
             0.0,  1.0,  0.0, // color for v2
             0.0,  0.2,  0.0, // vertex 3
             1.0,  0.0,  0.0, // color for v3
        ]);

        this.vbo = this.gl.createBuffer();
        this.gl.bindBuffer(this.gl.ARRAY_BUFFER, this.vbo);

        this.gl.bufferData(this.gl.ARRAY_BUFFER, data, this.gl.STATIC_DRAW);

        this.vao = this.gl.createVertexArray();
        this.gl.bindVertexArray(this.vao);

        this.gl.enableVertexAttribArray(0);
        this.gl.vertexAttribPointer(0, 3, this.gl.FLOAT, false, 24, 0);
        // stride is 24 since 3 float values(4 bytes each) for vertices and offset is
0 since it starts from the beginning.

        this.gl.enableVertexAttribArray(1);
        this.gl.vertexAttribPointer(1, 3, this.gl.FLOAT, false, 24, 12);
        // stride is same while offset is 12 since it has to skip 12 bytes to reach
the color attribute.
    }
    render() {
        this.gl.bindVertexArray(this.vao);
        this.gl.drawArrays(this.gl.TRIANGLES, 0, 3);
        // first is 0 since it has to start drawing from the starting index and count
is 3 for 3 vertices that make up a triangle .
    }
}
```

- NOTE:
  - you can use multiple VBO's with a single VAO but have to deal with binding the correct VBO while specifying VAO pointers.

SHADERS AND VARYING:

- WebGL runs on the GPU on your computer. The code that runs on the gpu are called shaders and we will deal with 2 types of shaders:
  - Vertex shader: runs for every vertex in your program.
  - Fragment shader: runs for every pixel on your screen.
- Shaders are each written in a very strictly typed C/C++ like language called GLSL. Paired together they form a shader program.
- Varying is a type of variable that can be passed on from the vertex shader to the fragment shader.
- Data passthrough from VBO's and VAO's are limited to the vertex shader. So we use varyings to pass data such as color data, texture coordinates etc. using varyings.

USAGE:

- SHADER PROGRAM:
  - The Shader.js file has utility functions to simplify the process of compiling and creating shader programs.
  - Import your vertex shader and fragment shader in your main.js.
  - Create a shader object using `new Shader(gl)` while passing the webGL context as `gl`.
  - Use the `createShaders(VERTEX_SHADER_SRC, FRAGMENT_SHADER_SRC)` method to compile and create your shader program.
  - Before any shader usage you have to specify which shader program is to be used. This is done by the `gl.useProgram(SHADER.program)` call where `SHADER` is your shader object.
- VERTEX SHADER:
  - The vertex shader is a code segment that runs for every vertex in your data buffer.
  - All coordinates ie; x, y, z coordinates specified lie in the clip space; ie an imaginary space lying between (-1, -1) to (1, 1).
  - All vertices are mapped on to the clip space and anything that lies outside it gets clipped.
  - Clip space coordinates always go from -1 to +1 no matter what size your canvas is.

```
#version 300 es

layout (location=0) in vec3 position; // incoming position attribute from the vao and
vbo, specifying x, y, z coordinates.
layout (location=1) in vec3 color;    // incoming color data specifying r, g, b
components of each vertex.

out vec3 vColor;

void main() {
        gl_Position = vec4(position, 1.0); // gl_Position is a global variable
```

```
specifying the final position of the vertex after calculation.
        vColor = color;
}
```

- FRAGMENT SHADER:
    - The fragment shader is a code segment that runs for every pixel/fragment available.

```
#version 300 es
precision highp float;

out vec4 fragColor;

in vec3 vColor; // color attribute passed on from the vertex shader.

void main() {
    fragColor = vec4(vColor,1.0); // the outgoing fragColor variable is the final
color of the fragment that is rendered on screen.
    // notice the output is a vec4 with attributes R, G, B and A
    // here webGL smoothly blends the colors between all 3 vertices.
}
```

- VARYINGS:
    - In both code segments you see vColor being specified.
    - vColor is a varying that is passed on from the Buffer to the vertex shader and in the vertex shader we update the value of the varying vColor to the incoming color attribute.
    - In the vertex shader vColor is specified using `out` as it goes through to the next stage ie; the fragment stage in the pipeline.
    - In the fragment shader vColor is specified using `in` as it comes in from the previous stage in the pipeline.

UNIFORMS:

- Uniforms are global variables you set before you execute your shader program.
- These are a type of variables that can change its data frequently.
- Time, Mouse Position, Resolution are a few common uniforms used in shader programming.
- It's not just limited to these you can pass any variable as a uniform.

USAGE:

- Specify a uniform in your shader ie; vertex shader or fragment shader(both even) using the `uniform` keyword.
- Find the location of the uniform using `gl.getUniformLocation(SHADER.program, "uniformName")` where `uniformName` should match with the variable name in your shader.
- In each draw call pass data to the uniform location using `gl.uniform1f(LOCATION, data)` where `LOCATION` is the uniform location.
- Instead of using `uniform1f` to pass just a single float as uniform you can use `uniform[1234]f|i|v` for passing higher order uniforms.
- EXAMPLE:

```
// main.js

        const uTimeLocation = gl.getUniformLocation(globalShader.program, "uTime");
// finding uTime location

        function renderLoop() {
        // updating time data
                currentTime = performance.now();
                elapsedTime = (currentTime - startTime) / 1000;


        // passing data to uniform location
                gl.uniform1f(uTimeLocation, elapsedTime);

                data.render();
                requestAnimationFrame(renderLoop);
        }
```

```
// vert.glsl

#version 300 es

layout (location=0) in vec3 position;
```

```
uniform float uTime; // specifying the uniform.

void main() {
        gl_Position = vec4(position*sin(uTime), 1.0); // using the uniform
}
```