OpenGL ES 2.0's view of the polygon projection pipeline

(from https://www.khronos.org/opengles/2_X/)
Execution Model: Vertex and Fragment Shaders

Vertex data, Uniform vars, and Shader Program

Application (CPU) Program

glDrawArrays()

Vertex Shader

Primitive Assembly

Vertex

Rasterizer

Fragment Shader

Frame Buffer

Shader Program and Uniform Vars

Application (CPU) Program

Frame Buffer

Fragment Color
a simple Vertex Shader

in this case we have an "attribute" input to the vertex shader from the application (CPU) program

the vPosition GLSL (GPU) variable must be linked to a variable in the application (CPU) program

```
attribute vec4 vPosition;
void main(void)
{
   gl_Position = vPosition;
}
```

"gl_Position" is a built-in GLSL variable, predefined for GLSL vertex shaders, it's the vertex shader's only mandatory output
Simple Fragment Program

```c
precision mediump float;

void main(void)
{
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

the "mediump" precision qualifier specifies the range:

<table>
<thead>
<tr>
<th>Precision</th>
<th>FP Range</th>
<th>FP Magnitude Range</th>
<th>FP Precision</th>
<th>Integer Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>mediump</td>
<td>$(-2^{14}, 2^{14})$</td>
<td>$(2^{-14}, 2^{14})$</td>
<td>Relative $2^{-10}$</td>
<td>$(-2^{10}, 2^{10})$</td>
</tr>
</tbody>
</table>

"gl_FragColor" is a built-in GLSL variable, predefined for GLSL fragment shaders, it's the vertex shader's only mandatory output
naming conventions for GLSL variables

in the "7th Ed." Angel – Shreiner textbook sample code

if you read example programs in the 6th and 7th edition of the Angel – Shreiner, these are the naming conventions used in that textbook sample code:

**attribute** variables passed to vertex shader have names beginning with v (vPosition, vColor) in both the application and the shader

Note that these are *different entities with the same name:*

*CPU based variables ≠ GPU-based variables*

**varying** variables begin with f (fColor) in both shaders, and *must* use the same name in both shaders

**uniform** variables are unadorned and *can* have the same name in application and shaders
variable naming example: Vertex Shader

```glsl
attribute vec4 vColor;
attribute vec4 vPosition;

varying vec4 fColor;

void main() {
    gl_Position = vPosition;
    fColor = vColor;
}
```
variable naming example: the corresponding Fragment Shader

precision mediump float;

varying vec3 fColor;

void main() {
    gl_FragColor = fColor;
}
Sending Color Data from Application to Shader

```javascript
var cBuffer = glCreateBuffer();
glBindBuffer( ARRAY_BUFFER, cBuffer );
glBufferData( ARRAY_BUFFER, colors,
STATIC_DRAW );

vColor = glGetAttribLocation( program, "vColor" );
glVertexAttribPointer( vColor, 3, FLOAT, false, 0, 0 );
glEnableVertexAttribArray( vColor );
```
Sending a Uniform Variable

// in main application code:
color = (1.0, 0.0, 0.0, 1.0);
colorLoc = glGetUniformLocation(program, "color");
glUniform4f(colorLoc, color);

// in fragment shader (and similarly in vertex shader)
uniform vec4 color;

void main()
{
    gl_FragColor = color;
}
Operators and Functions

Standard C functions

- Trigonometric
- Arithmetic
-Normalize, reflect, length

Overloading of vector and matrix types

```c
mat4 a;
vec4 b, c, d;
c = b*a; // a column vector stored as a 1d array
d = a*b; // a row vector stored as a 1d array
```
Swizzling and Selection

GLSL code can refer to array elements by element using [] or the selection (.) operator with:

- x, y, z, w
- r, g, b, a
- s, t, p, q

\[ a[2], \ a.b, \ a.z, \ a.p \] are the same

the **swizzling** operator can be used to manipulate components:

```cpp
vec4 a, b;
a.yz = vec2(1.0, 2.0, 3.0, 4.0);
b = a.yxzw;
```
OpenGL ES Graphics Primitives

- POINTS
- LINES
- LINE_STRIP
- TRIANGLES
- TRIANGLE_STRIP
- TRIANGLE_FAN
- LINE_LOOP
Using Buffer Objects

As indicated earlier, a buffer object is a mechanism provided by the WebGL system that provides a memory area allocated in the system (see Figure 3.4) that holds the vertices you want to draw. By creating a buffer object and then writing the vertices to the object, you can pass multiple vertices to a vertex shader through one of its attribute variables.

**Figure 3.4** Passing multiple vertices to a vertex shader by using a buffer object
Using Buffer Objects: five steps

1. Create a buffer object (gl.createBuffer()).
2. Bind the buffer object to a target (gl.bindBuffer()).
3. Write data into the buffer object (gl.bufferData()).
4. Assign the buffer object to an attribute variable (gl.vertexAttribPointer()).
5. Enable assignment (gl.enableVertexAttribArray()).
Using Buffer Objects: five steps

1. Create a buffer object (gl.createBuffer()).
2. Bind the buffer object to a target (gl.bindBuffer()).
3. Write data into the buffer object (gl.bufferData()).
4. Assign the buffer object to an attribute variable (gl.vertexAttribPointer()).
5. Enable assignment (gl.enableVertexAttribArray()).
Using Buffer Objects: five steps

// Create a buffer object
var vertexBuffer = gl.createBuffer();
if (!vertexBuffer) {
    console.log('Failed to create a buffer object');
    return -1;
}

// Bind the buffer object to a target
(gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);

// Write data into the buffer object
(gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);

var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
...

// 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);
Using Buffer Objects:

step 1 -- create a Buffer Object

The state before and after execution of step 1 function `glCreateBuffer()`: when the function is executed, it results in a single buffer object being created in the OpenGL ES system.
Using Buffer Objects:

step 2 – bind a Buffer Object to a target

After creating a buffer object, the second step is to **bind** it to a “**target**” using `glBindBuffer()`. The **target** tells OpenGL ES what **type of data** the buffer object contains, allowing it to deal with the contents correctly.
Using Buffer Objects:
step 3 – write data into a Buffer Object

After binding the buffer object to a target, step 3 **allocates** storage and **writes** data to the buffer. The `glBufferData()` function takes the vertex data defined in the CPU program and writes the data to the buffer object bound to `ARRAY_BUFFER`. 
Using Buffer Objects:

step 4 – assign the Buffer Object to an Attribute Variable in the shader

`glVertexAttribAttribPointer()` assigns an array of values (the vertices) to an attribute variable, to assign a reference (or handle) to the buffer object to an attribute variable:

(side note: `glVertexAttrib[1234]f()` can also be used to assign data to an attribute variable – but it assigns a single data value at a time to an attribute variable.)
Using Buffer Objects:

step 5 – **enable the Assignment to an Attribute Variable**

finally, to access the buffer object in the vertex shader, enable the assignment of the buffer object to an attribute variable by using `glEnableVertexAttribArray()`

after these 5 steps are completed, you can **finally** run the vertex shader by calling `glDrawArrays()`, which draws the primitive using the vertex coordinates you just stored in the buffer object.