WebGL Programming

Marco Wang <m.aesophor@gmail.com>

AGENDA

- Prerequisites
- What is WebGL?
- How is WebGL Different from OpenGL?
- Pipeline (OpenGL 1.1/2.0/3.0 vs WebGL)
- Moving from OpenGL to WebGL

- Example 1 - 2D RGB Triangle
- Example 2 - 3D Colorcube

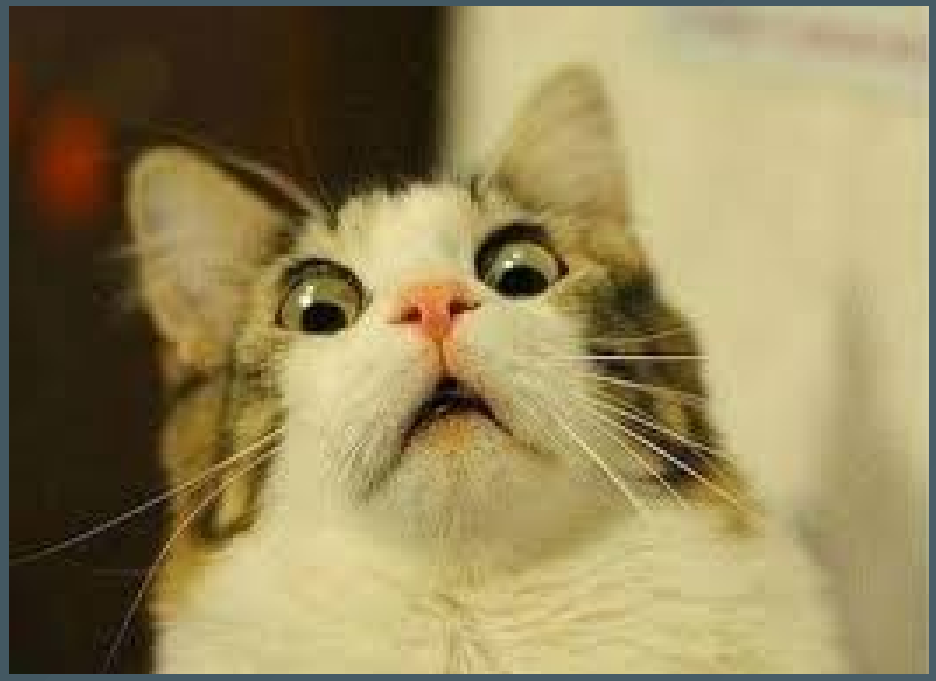
Prerequisites

- An HTTP Web Server
  - Windows: xampp 或 WampServer
  - Linux: Apache 或 nginx
- Any Text Editor
- Any Web Broswer that Supports HTML5

Today's slides: https://bit.ly/2RiNsGE
Today's source code: https://bit.ly/2CyQ9f7

What is WebGL?

“ WebGL is an API to access the GPU ”



How is WebGL Different from OpenGL?

Drawing a triangle in OpenGL is as simple as follows

```
void triangle(GLfloat* v1, GLfloat* v2, GLfloat* v3) {
    glBegin(GL_TRIANGLES);
        glVertex3fv(v1);
        glVertex3fv(v2);
        glVertex3fv(v3);
    glEnd();
}
```

How is WebGL Different from OpenGL?

To draw a triangle in WebGL, you'll need to ...

- Write your own vertex shader and a fragment shader
- Compile shaders using WebGL API
- Allocate a chunk of memory on GPU
- Upload your data (vertices, colors etc) to GPU memory
- Tell GPU how to deal with the data you've uploaded
- `gl.drawArrays(gl.TRIANGLES, 0, 3);`

There's no gl.begin(), gl.end() or such. They have been deprecated since OpenGL 3.0, and removed since 3.1

Pipeline

> OpenGL 1.1 used a fixed-function pipeline for graphics processing.

There is no way we could change what happens at each stage in the pipeline. The functionality is fixed.

> OpenGL 2.0 has optional programmable pipeline.

Programmers could optionally replace certain stages in the pipeline with their own programs.

Pipeline

> Since OpenGL 3.0, fixed-function pipeline has been deprecated and removed since 3.1

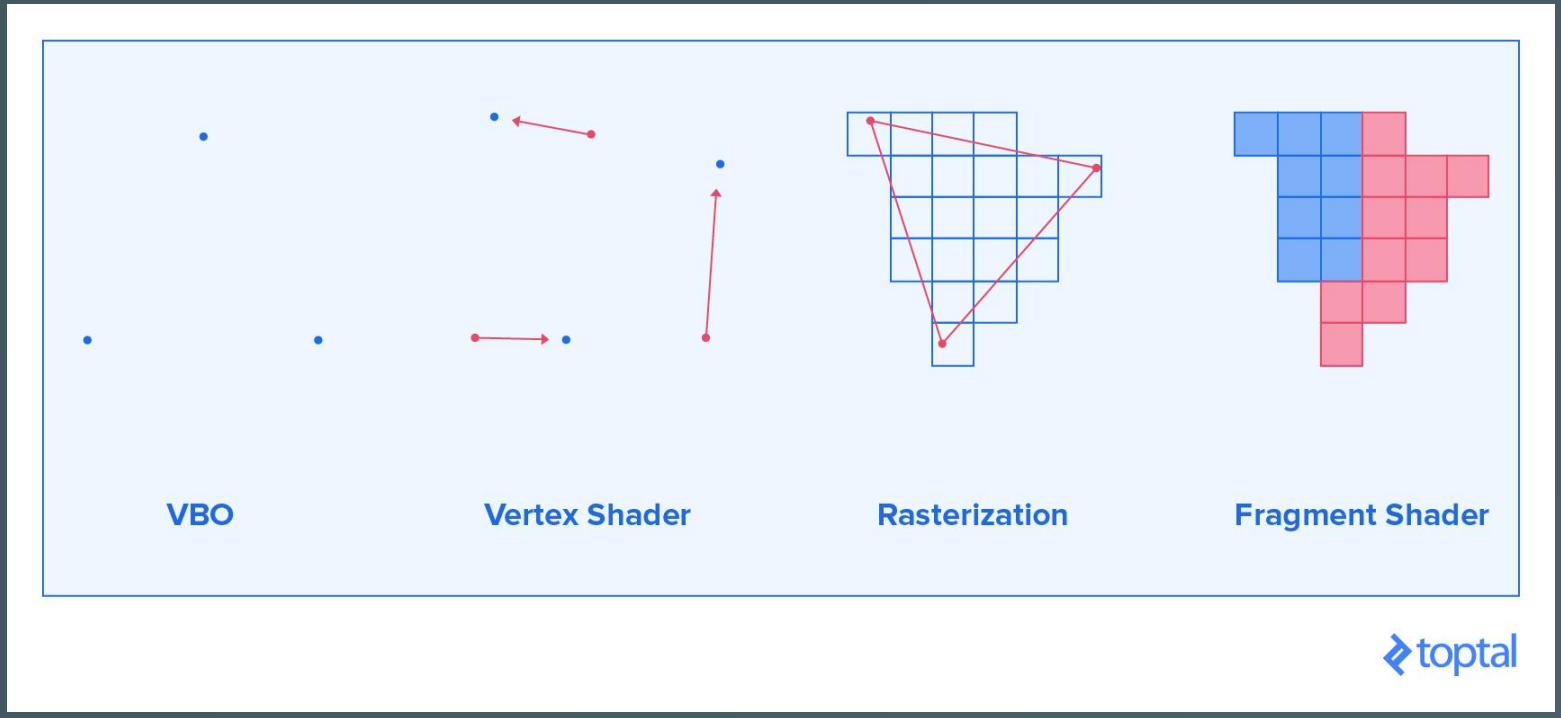
> WebGL 2.0 is based on OpenGL 3.0 ES, which uses a mandatory programmable pipeline.

In WebGL, we have to implement the following parts of pipeline ourselves!

- Vertex Shader
- Fragment Shader
- "Upload" data to GPU memory

The Two Mandatory Shaders

- Vertex Shader: handles vertex coordinate
- Fragment Shader: handles pixel color



“ Vertex Shader is a shader program that will be executed once for each vertex in a primitive.

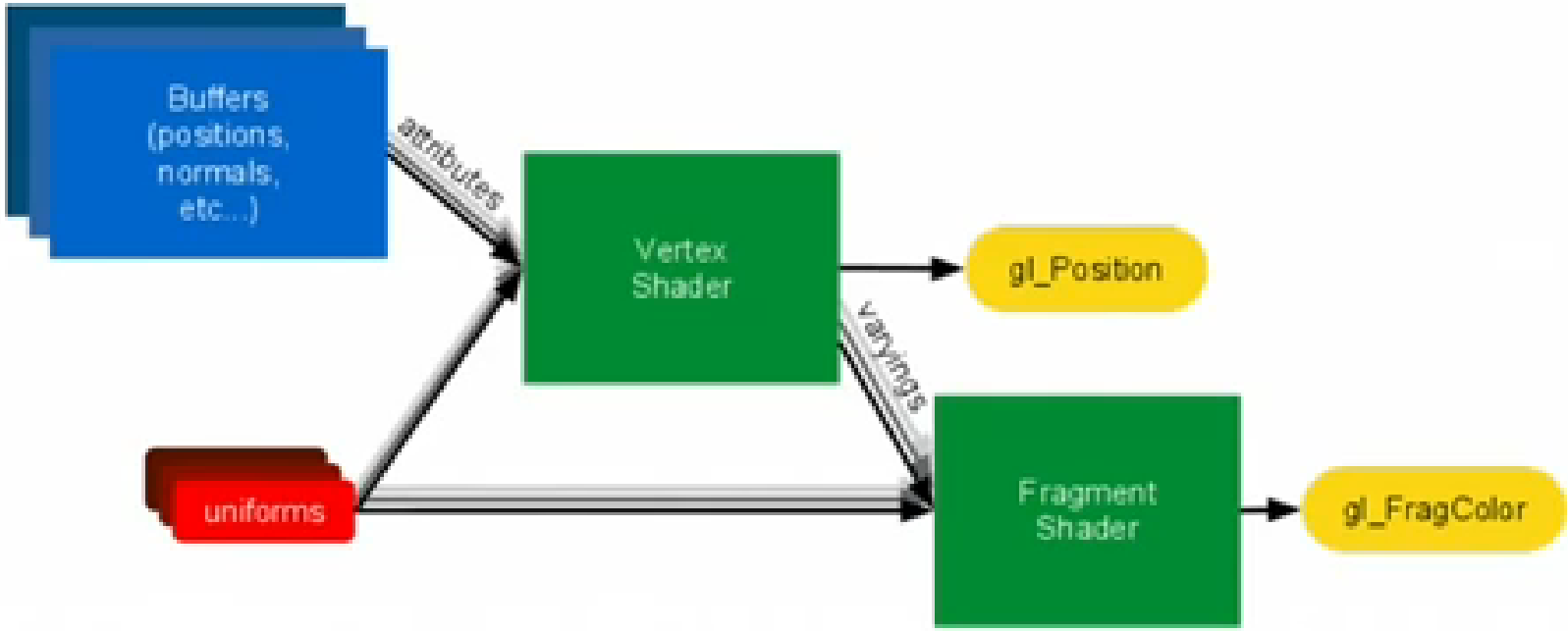
A vertex shader must compute the vertex coordinates in the clip coordinate system. ”

“ Fragment Shader / Pixel Shader is a shader program that will be executed once for each pixel in a primitive.

A fragment shader must compute a color for the pixel, or discard it. ”

WebGL is SIMPLE!

- A brief review



Example 1

2D RGB Triangle

Step 1

Preparations

Create `triangle.html` in your web server's root directory

```
$ cd /var/www/localhost/htdocs
$ vim triangle.html
```

onload="main();" means to call main() in `triangle.js` when the html is loaded.

```
<html>
    <head>
        <title>WebGL - RGB Triangle</title>
    </head>

    <body onload="main();">
        <canvas id="main_canvas" width="650" height="500"></canvas>
        <script src="triangle.js"></script>
    </body>
</html>
```

Create `triangle.js` (in the same directory)

```
function init() {
    canvas = document.getElementById('main_canvas');

    gl = canvas.getContext('webgl')
        || canvas.getContext('experimental-webgl');

    if (!gl) {
        alert("Your browser does not support WebGL");
        return;
    }
}

function main() {
    init();
}
```

Step 2

Writing and Compiling Custom Shaders

Vertex Shader & Fragment Shader

```
precision mediump float;

attribute vec2 vertPosition;
attribute vec3 vertColor;
varying vec3 fragColor;

void main() {
   gl_Position = vec4(vertPosition, 0.0, 1.0);
   fragColor = vertColor;
}
```

```
precision mediump float;

varying vec3 fragColor;

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

Writing and Compiling Our Shaders

We'll start with vertex shader first. Put the source code in an array, and join each word with '\n'.

```
var vertexShaderSrc = [
    'precision mediump float;',
    'attribute vec2 vertPosition;',
    'attribute vec3 vertColor;',
    'varying vec3 fragColor;',
    '',
    'void main() {',
    '   gl_Position = vec4(vertPosition, 0.0, 1.0);',
    '   fragColor = vertColor;',
    '}'
].join('\n');
```

Next, the fragment shader.

Like C/C++, each shader has a main() as entry point.

```
var fragmentShaderSrc = [
    'precision mediump float;',
    'varying vec3 fragColor;',
    '',
    'void main() {',
    '   gl_FragColor = vec4(fragColor, 1.0);',
    '}'
].join('\n');
```

Uniform? Attribute? Varying?

These are like `const`, `extern` or whatever modifiers in C/C++

- Uniform: 從外部透過glUniform*() 函數傳入。相當於C語言的const，shader只能讀不可改。
- Attribute: Vertex Shader的input。可存頂點、顏色
- Varying: 用在Shader之間傳遞變數，即Vertex Shader和Fragment Shader之間的橋樑。

```
modifier | type | var name
--------------------------
attribute vec2 vertPosition;
attribute vec3 vertColor;
varying vec3 fragColor;
```

Use the Following Code to Compile Vertex Shader

The parameter `vertexShaderSource` has to be a String.

```
// Compile vertex shader from source.
var vertexShader = gl.createShader(gl.VERTEX_SHADER);

gl.shaderSource(vertexShader, vertexShaderSource);
gl.compileShader(vertexShader);

if (!gl.getShaderParameter(vertexShader, gl.COMPILE_STATUS)) {
    throw "[ERROR] error compiling vertex shader: "
        + gl.getShaderInfoLog(vertexShader);
}
```

Use the Following Code to Compile Fragment Shader

The parameter `fragmentShaderSource` has to be a String.

```
// Compile fragment shader from source.
var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);

gl.shaderSource(fragmentShader, fragmentShaderSource);
gl.compileShader(fragmentShader);

if (!gl.getShaderParameter(fragmentShader, gl.COMPILE_STATUS)) {
    throw "[ERROR] error compiling fragment shader: "
        + gl.getShaderInfoLog(fragmentShader);
}
```

Create Program and Attach Two Shaders

A WebGLProgram object contains two shaders, we'll call gl.attachShader() to tell our program which shaders to use.

```
var program = gl.createProgram();

gl.attachShader(program, vertexShader);
gl.attachShader(program, fragmentShader);
gl.linkProgram(program);

if (!gl.getProgramParameter(program, gl.LINK_STATUS)) {
    throw "[ERROR] error linking program: "
        + gl.getProgramInfoLog(program);
}
```

```
gl.useProgram(program);
```

Step 3

Setting Up Data (Vertices & Colors)

Define the vertices in a javascript array

```
var triangleVertices = [
    0.0, 0.5,
    -0.5, -0.5,
    0.5, -0.5
];
```

First, we have to allocate some memory on GPU

```
var verticesBuffer = gl.createBuffer();
```

and save the pointer to the shader variable `vertPosition` in javascript.

```
var positionAttribLocation = gl.getAttribLocation(program, 'vertPosition');
```

Then bind our local `triangleVertices` array to the buffer we've allocated on GPU.

```
gl.bindBuffer(gl.ARRAY_BUFFER, verticesBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices), gl.STATIC_DRAW)
```

What's with that `new Float32Array()` ???

- In javascript, all numbers in an array is 64-bit.
- But WebGL requires 32-bit data!
- So we will wrap the array in a `Float32Array`.

Tell WebGL how to use the data we've uploaded

```
gl.vertexAttribPointer(
        positionAttribLocation,             // attribute location.
        2,                                  // number of elements per attribute
        gl.FLOAT,                           // type of elements
        gl.FALSE,                           //
        2 * Float32Array.BYTES_PER_ELEMENT, // size of an individual vertex, i.e.,
        0                                   // how many elements to skip
);
```

For performance sake, all attributes are disabled by default (even if you have called bindBuffer()).

We have to tell WebGL to use the attributes

```
gl.enableVertexAttribArray(positionAttribLocation);
```

Now Apply the Same Rule to Colors Definition.

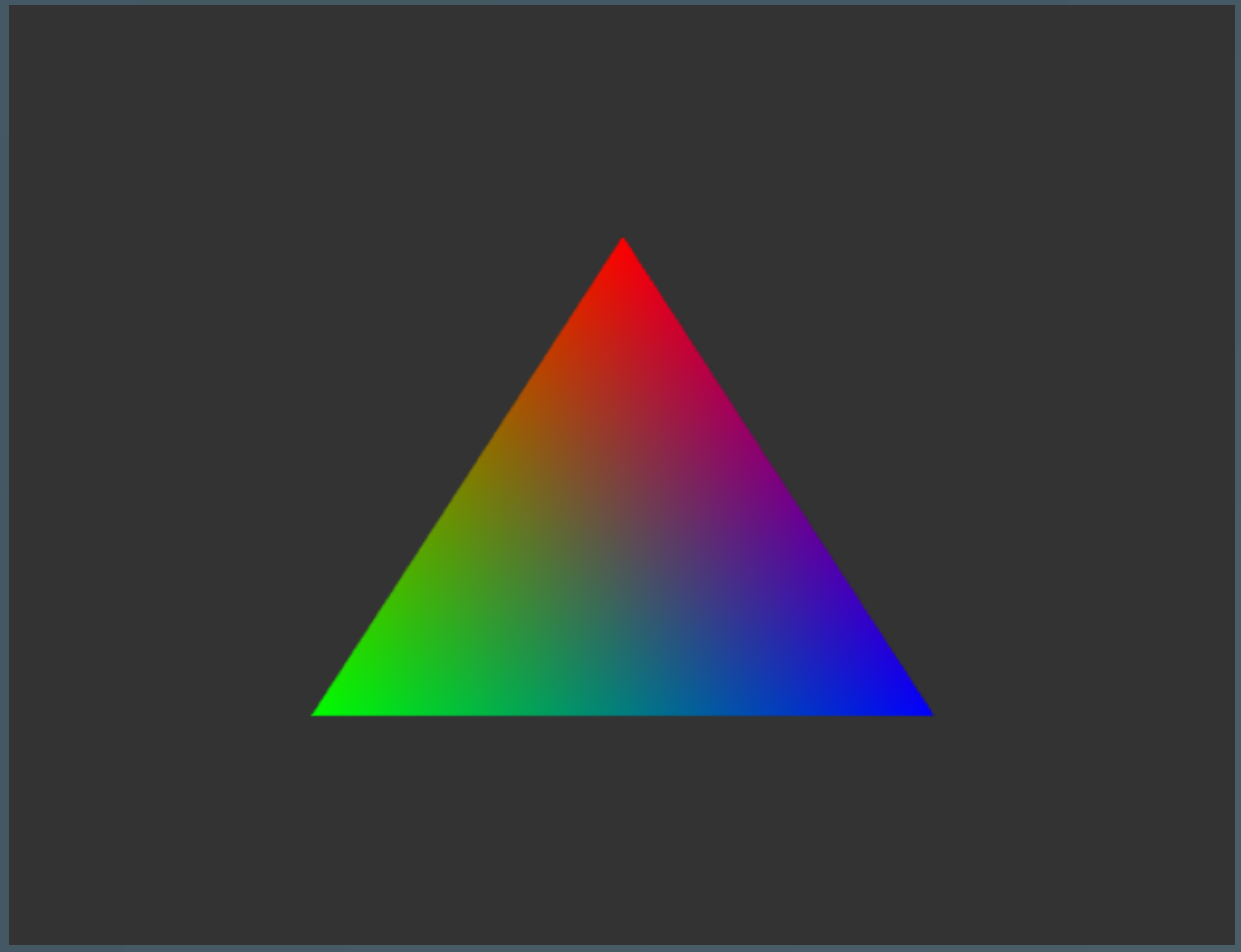
```
var triangleColors = [
    1.0, 0.0, 0.0,
    0.0, 1.0, 0.0,
    0.0, 0.0, 1.0
];
```

```
gl.bindBuffer(gl.ARRAY_BUFFER, colorsBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleColors), gl.STATIC_DRAW);

gl.vertexAttribPointer(colorAttribLocation, 3, gl.FLOAT, gl.FALSE,
        3 * Float32Array.BYTES_PER_ELEMENT, 0);

gl.enableVertexAttribArray(colorAttribLocation);
```

Finally we can draw it.

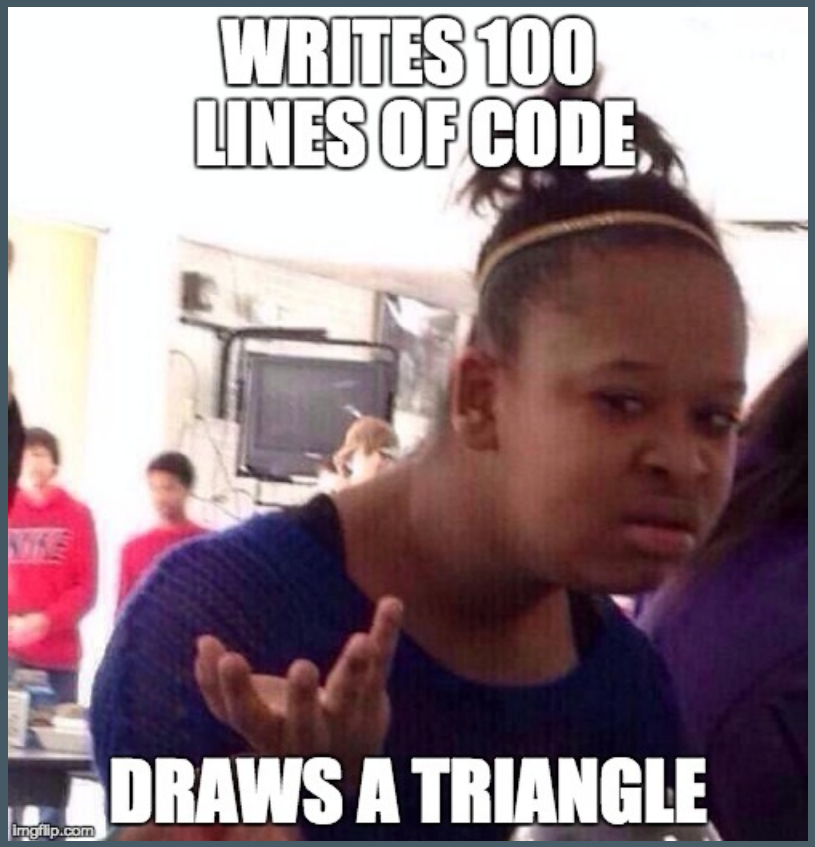
```
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

Result



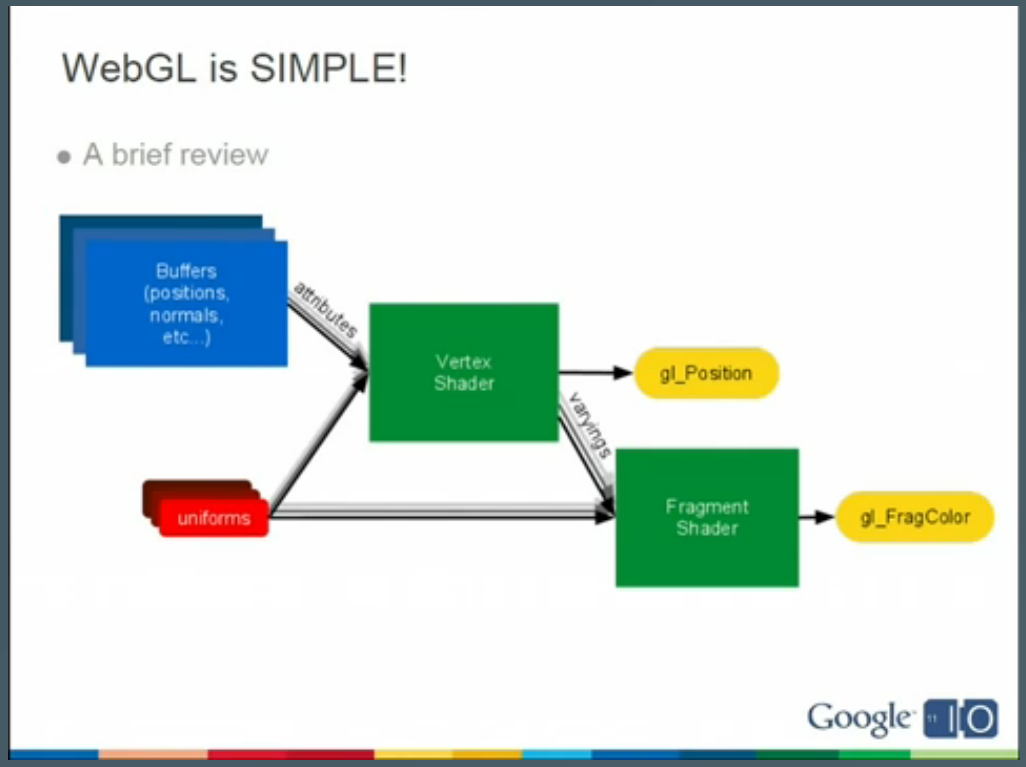
Wut?

In OpenGL, we could draw a colorcube within 72 lines of code



Revisiting Pipeline

The data (i.e., attributes) are passed into Vertex Shader first, and then the only way to pass data into Fragment Shader is through `varying variables`.



Revisiting Uniform, Attribute, Varying

- Uniform: 從外部透過glUniform*() 函數傳入。相當於C語言的const，shader只能讀不可改。
- Attribute: Vertex Shader的input。可存頂點、顏色
- Varying: 用在Shader之間傳遞變數，即Vertex Shade和Fragment Shader之間的橋樑。

Two Special Variables

- gl_Position: Coordinate of current vertex
- gl_FragColor: Color of current vertex

Passing Data from Vertex Shader to Fragment Shader via `varying` keyword

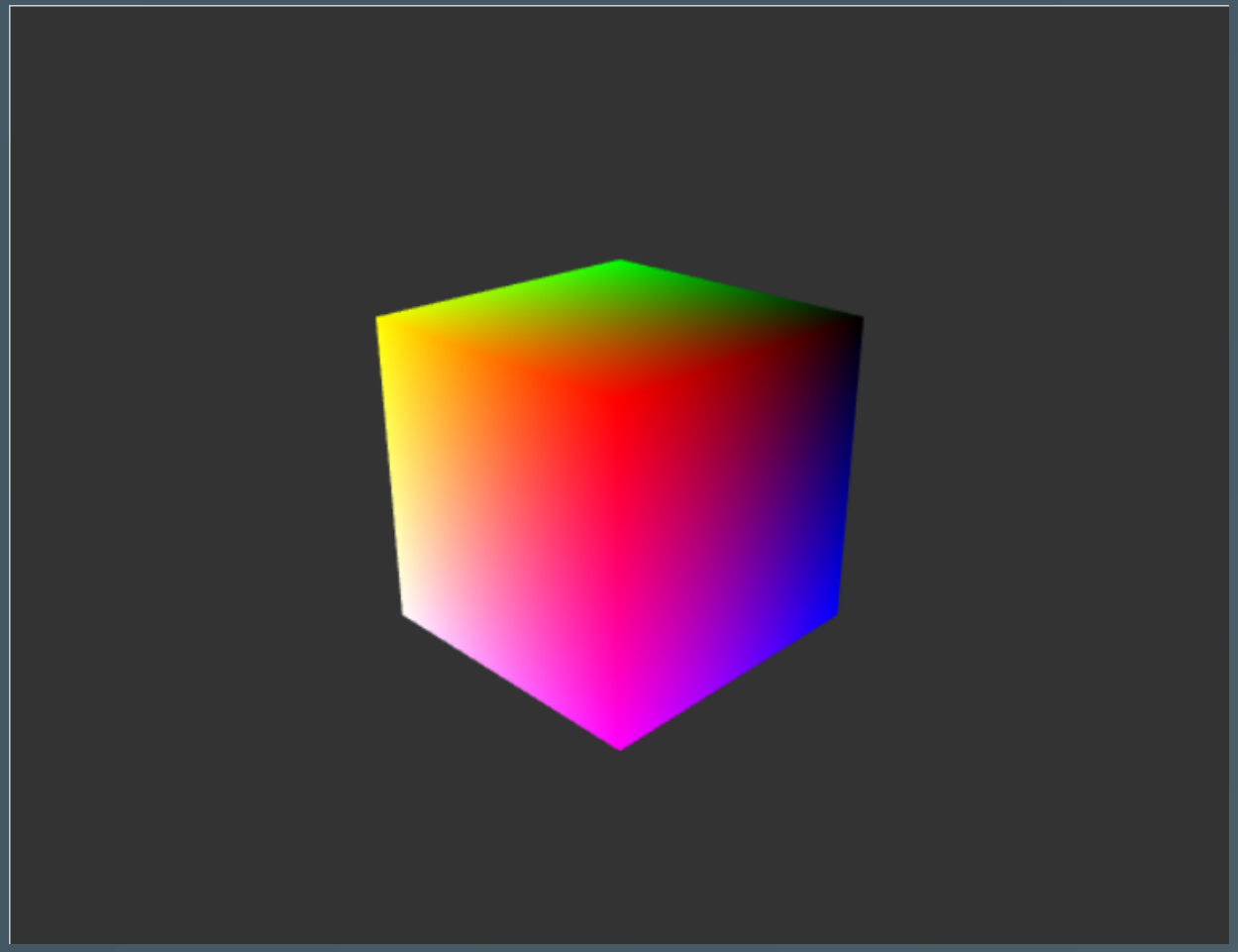
```
attribute vec2 vertPosition;
attribute vec3 vertColor;
varying vec3 fragColor;

void main() {
   gl_Position = vec4(vertPosition, 0.0, 1.0);
   fragColor = vertColor;
}
```

```
varying vec3 fragColor;

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

Example 2

3D Colorcube

Result



Result