WebGPU from WebGL

This article is meant for people who already know WebGL and want to start using WebGPU.

If you're coming from WebGL to WebGPU it's worth noting that many of the concepts are the same. Both WebGL and WebGPU let you run small functions on the GPU. WebGL has vertex shaders and fragment shaders. WebGPU has the same plus compute shaders. WebGL uses GLSL as its shading language. WebGPU uses WGSL. While they are different languages the concepts are mostly the same.

Both APIs have attributes, a way to specify data pulled from buffers and fed to each iteration of a vertex shader. Both APIs have uniforms, a way to specify values shared by all iterations of a shader function. Both APIs have varyings, a way to pass data from a vertex shader to a fragment shader and interpolate between values computed by the vertex shader when rasterizing via a fragment shader. Both APIs have textures and samplers, ways to provide 2D or 3D data and sample it (filter multiple pixels into a single value). Both APIs provide ways to render to textures. And, both have a bunch of settings for how pixels are blended, how the depth buffer and stencil buffers work, etc…

The biggest difference is WebGL is a stateful API and WebGPU is not. By that I mean in WebGL there is a bunch of global state. Which textures are currently bound, which buffers are currently bound, what the current program is, what the blending, depth, and stencil settings are. You set those states by calling various API functions like `gl.bindBuffer`, `gl.enable`, `gl.blendFunc`, etc…, and they stay what you set them *globally* until you change them to something else.

By contrast, In WebGPU there is almost no *global* state. Instead, there are the concepts of a *pipeline* or *render pipeline* and a *render pass* which together effectively contain most of the state that was global in WebGL. Which textures, which attributes, which buffers, and all the various other settings. Any settings you don't set have default values. You can't modify a pipeline. Instead, you create them and after that they are immutable. If you want different settings you need to create another pipeline. *render passes* do have some state, but that state is local to the render pass.

The second-biggest difference is that WebGPU **is lower level** than WebGL. In WebGL many things connect by names. For example, you declare a uniform in GLSL and you look up its location

```
loc = gl.getUniformLocation(program, 'nameOfUniform');
```

Another example is varyings, in a vertex shader you use `varying vec2 v_texcoord` or `out vec2 v_texcoord` and in the fragment shader you declare the corresponding varying naming it `v_texcoord`. The good part of this is if you mistype the name you'll get an error.

WebGPU, on the other hand, everything is entirely connected by index or byte offset. You don't create individual uniforms like WebGL, instead you declare uniform blocks (a structure that declares your uniforms). It's then up to you to make sure you manually organize the data you pass to the shader to match that structure. Note: WebGL2 has the same concept, known as Uniform Blocks, but WebGL2 also had the concept of uniforms by name. And, even though individual fields in a WebGL2 Uniform Block needed to be set via byte offsets, (a) you could query WebGL2 for those offsets and (b) you could still look up the block locations themselves by name.

In WebGPU on the other hand **EVERYTHING** is by byte offset or index (often called '*location*') and there is no API to query them. That means it's entirely up to you to keep those locations in sync and to manually compute byte offsets.

To give a JavaScript analogy:

```
function likeWebGL(inputs) {
  const {position, texcoords, normal, color} = inputs;
  ...
}

function likeWebGPU(inputs) {
  const [position, texcoords, normal, color] = inputs;
  ...
}
```

In the `likeWebGL` example above, things are connected by name. We can call `likeWebGL` like this

```
const inputs = {};
inputs.normal = normal;
inputs.color = color;
inputs.position = position;
likeWebGL(inputs);
```

or like this

```
likeWebGL({color, position, normal});
```

Notice because they are connected by names, the order of our parameters does not matter. Further, we can skip a parameter (`texcoords` in the example above) assuming the function can run without `texcoords`.

On the other hand with `likeWebGPU`

```
const inputs = [];
inputs[0] = position;
inputs[2] = normal;
inputs[3] = color;
likeWebGPU(inputs);
```

Here, we pass in our parameters in an array. Notice we have to know the locations (indices) for each input. We need to know that `position` is index 0, `normal` is at index 2 etc. Keeping the locations for the code inside (WGSL) and outside (JavaScript/WASM) in sync in WebGPU is entirely your responsibility.

Other notable differences

- The Canvas

  WebGL manages the canvas for you. You choose antialias, preserveDrawingBuffer, stencil, depth, and alpha when you create the WebGL context and after that WebGL manages the canvas itself. All you have to do is set `canvas.width` and `canvas.height`.

  WebGPU you have to do much of that yourself. If you want a depth buffer you create that yourself (with or without a stencil buffer). If you want anti-aliasing, you create your own multisample textures and resolve them into the canvas texture.

  But, because of that, unlike WebGL, you can use one WebGPU device to render to multiple canvases. 🎉🤩

- WebGPU does not generate mipmaps.

  In WebGL you could create a texture's level 0 mip and then call `gl.generateMipmap` and WebGL would generate all the other mip levels. WebGPU has no such function. If you want mips for your textures you have to generate them yourself.

  Note: this article has code to generate mips.

- WebGPU requires samplers

  In WebGL1, samplers did not exist or to put it another way, samplers were handled by WebGL internally. In WebGL2 using samplers was optional. In WebGPU samplers are required.

- Buffers and Textures can not be resized

  In WebGL you could create a buffer or texture and then at anytime, change its size. For example if you called `gl.bufferData` the buffer would be reallocated. If you called `gl.texImage2D` the texture would be reallocated. A common pattern with textures was to create a 1x1 pixel placeholder that lets you start rendering immediately and then load an image asynchronously. When the image was finished loading you'd update the texture in place.

  In WebGPU texture and buffer sizes, usage, formats are immutable. You can change their contents but you can not change anything else about them. This means, patterns in WebGL where you were changing them, like the example mentioned above, need to be refactored to create a new resource.

  In other words, instead of

  ```
  // pseudo code
  const tex = createTexture()
  fillTextureWith1x1PixelPlaceholder(tex)
  imageLoad(url).then(img => updateTextureWithImage(tex, image));
  ```

  You need to change your code to effectively something like

  ```
  // pseudo code
  let tex = createTexture(size: [1, 1]);
  fillTextureWith1x1PixelPlaceholder(tex)
  imageLoad(url).then(img => {
      tex.destroy();  // delete old texture
      tex = createTexture(size: [img.width, img.height]);
      copyImageToTexture(tex, image));
  });
  ```

Let's compare WebGL to WebGPU

Shaders

Here is a shader that draws textured, lit, triangles. One in GLSL and the other in WGSL.

GLSL

```
const vSrc = `
uniform mat4 u_worldViewProjection;
uniform mat4 u_worldInverseTranspose;

attribute vec4 a_position;
attribute vec3 a_normal;
attribute vec2 a_texcoord;

varying vec2 v_texCoord;
varying vec3 v_normal;

void main() {
  gl_Position = u_worldViewProjection * a_position;
  v_texCoord = a_texcoord;
  v_normal = (u_worldInverseTranspose * vec4(a_normal, 0)).xyz;
}
`;

const fSrc = `
precision highp float;

varying vec2 v_texCoord;
varying vec3 v_normal;

uniform sampler2D u_diffuse;
uniform vec3 u_lightDirection;

void main() {
  vec4 diffuseColor = texture2D(u_diffuse, v_texCoord);
  vec3 a_normal = normalize(v_normal);
  float l = dot(a_normal, u_lightDirection) * 0.5 + 0.5;
  gl_FragColor = vec4(diffuseColor.rgb * l, diffuseColor.a);
}
`;
```

WGSL

```
const shaderSrc = `
struct VSUniforms {
  worldViewProjection: mat4x4f,
  worldInverseTranspose: mat4x4f,
};
@group(0) binding(0) var<uniform> vsUniforms: VSUniforms;

struct MyVSInput {
    @location(0) position: vec4f,
    @location(1) normal: vec3f,
    @location(2) texcoord: vec2f,
};

struct MyVSOutput {
  @builtin(position) position: vec4f,
  @location(0) normal: vec3f,
  @location(1) texcoord: vec2f,
};

@vertex
fn myVSMain(v: MyVSInput) -> MyVSOutput {
  var vsOut: MyVSOutput;
  vsOut.position = vsUniforms.worldViewProjection * v.position;
  vsOut.normal = (vsUniforms.worldInverseTranspose * vec4f(v.normal, 0.0)).xyz;
  vsOut.texcoord = v.texcoord;
  return vsOut;
}

struct FSUniforms {
  lightDirection: vec3f,
};

```

```
@group(0) binding(1) var<uniform> fsUniforms: FSUniforms;
@group(0) binding(2) var diffuseSampler: sampler;
@group(0) binding(3) var diffuseTexture: texture_2d<f32>;

@fragment
fn myFSMain(v: MyVSOutput) -> @location(0) vec4f {
  var diffuseColor = textureSample(diffuseTexture, diffuseSampler, v.texcoord);
  var a_normal = normalize(v.normal);
  var l = dot(a_normal, fsUniforms.lightDirection) * 0.5 + 0.5;
  return vec4f(diffuseColor.rgb * l, diffuseColor.a);
}
`;
```

Notice in many ways they aren't all that different. The core parts of each function are very similar. `vec4` in GLSL becomes `vec4f` in WGSL, `mat4` becomes `mat4x4f`. Other examples include `int` -> `i32`, `uint` -> `u32`, `ivec2` to `vec2i`, `uvec3` to `vec3u`.

GLSL is C/C++ like. WGSL is Rust like. One difference is types go on the left in GLSL and on the right in WGSL.

GLSL

```
// declare a variable of type vec4
vec4 v;

// declare a function of type mat4 that takes a vec3 parameter
mat4 someFunction(vec3 p) { ... }

// declare a struct
struct Foo { vec4 field; };
```

WGSL

```
// declare a variable of type vec4f
var v: vec4f;

// declare a function of type mat4x4f that takes a vec3f parameter
fn someFunction(p: vec3f) -> mat4x4f { ... }

// declare a struct
struct Foo { field: vec4f, };
```

WGSL has the concept that if you do not specify the type of variable it will be deduced from the type of the expression on the right whereas GLSL required you to always specify the type. In other words in GLSL

```
vec4 color = texture(someTexture, someTextureCoord);
```

Above you needed to declare `color` as a `vec4` but in WGSL you can do either of these

```
var color: vec4f = textureSample(someTexture, someSampler, someTextureCoord);
```

or

```
var color = textureSample(someTexture, someSampler, someTextureCoord);
```

In both cases `color` is a `vec4f`.

On the other hand, the biggest difference is all the `@???` parts. Each one is declaring exactly where that particular piece of data is coming from. For example, notice that uniforms in the vertex shader and uniforms the fragment shader declare their `@group(?) binding(?)` and that it's up to you to make sure they don't clash. Above the vertex shader uses `binding(0)` and the fragment shader `binding(1)`, `binding(2)`, `binding(3)` In the example above there are 2 uniform blocks. We could have used 1. I chose to use 2 to more separate the vertex shader from the fragment shader.

Another difference between WebGL and WebGPU is that in WebGPU you can put multiple shaders in the same source. In WebGL a shader's entry point was always called `main` but in WebGPU when you use a shader you specify which function to call.

Notice in WebGPU the attributes are declared as parameters to the vertex shader function vs GLSL where they are declared as globals outside the function and unlike GLSL where if you don't choose a location the compiler will assign one, in WGSL we must supply the locations.

For varyings, in GLSL they are also declared as global variables whereas in WGSL you declare a structure with locations for each field, you declare your vertex shader as returning that structure, and you return an instance of that structure in the function itself. In the fragment shader you declare your function as taking these inputs.

The code above uses the same structure for both the vertex shader's output and the fragment shader's input, but there is no requirement to use the same structure. All that is required is the locations match. For example this would work:

```
struct MyFSInput {
  @location(0) the_normal: vec3f,
  @location(1) the_texcoord: vec2f,
};

@fragment
fn myFSMain(v: MyFSInput) -> @location(0) vec4f
{
  var diffuseColor = textureSample(diffuseTexture, diffuseSampler, v.the_texcoord);
  var a_normal = normalize(v.the_normal);
  var l = dot(a_normal, fsUniforms.lightDirection) * 0.5 + 0.5;
  return vec4f(diffuseColor.rgb * l, diffuseColor.a);
}
```

This would also work

```
@fragment
fn myFSMain(
  @location(1) uv: vec2f,
  @location(0) nrm: vec3f,
) -> @location(0) vec4f
{
  var diffuseColor = textureSample(diffuseTexture, diffuseSampler, uv);
  var a_normal = normalize(nrm);
  var l = dot(a_normal, fsUniforms.lightDirection) * 0.5 + 0.5;
  return vec4f(diffuseColor.rgb * l, diffuseColor.a);
}
```

Again, what matters is the that the locations match, not the names.

Another difference to notice is `gl_Position` in GLSL has just a special location `@builtin(position)` for a user declared structure field in WGSL. Similarly, the output of the fragment shader is given a location. In this case `@location(0)`. This is similar to using `gl_FragData[0]` in WebGL1's `WEBGL_draw_buffers` extension. Here again, if you wanted to output more than a single value, for example to multiple render targets, you'd declare a structure and assign locations just like we did for the output of the vertex shader.

Getting the API

WebGL

```
function main() {
  const gl = document.querySelector('canvas').getContext('webgl');
  if (!gl) {
    fail('need webgl');
    return;
  }
}

main();
```

WebGPU

```
async function main() {
  const adapter = await navigator.gpu?.requestAdapter();
  const device = await adapter?.requestDevice();
  if (!device) {
    fail('need a browser that supports WebGPU');
    return;
  }

  ...
}

main();
```

Here, `adapter` represents the GPU itself whereas `device` represents an instance of the API on that GPU.

Probably the biggest difference here is that getting the API in WebGPU is asynchronous.

Creating Buffers

WebGL

```
function createBuffer(gl, data, type = gl.ARRAY_BUFFER) {
  const buf = gl.createBuffer();
  gl.bindBuffer(type, buf);
  gl.bufferData(type, data, gl.STATIC_DRAW);
  return buf;
}

const positions = new Float32Array([1, 1, -1, 1, 1, 1, 1, -1, 1, 1, -1, -1, -1, 1, 1, -1, 1, -1, -1, -1, -1, -1, -1, 1, -1, 1, 1, 1, 1, 1, 1, 1, -1, -1, 1, -1, -1, -1, -1, 1, -1, -1, 1, -1, 1, -1, -1, 1, 1, 1, 1, -1, 1, 1, -1, -1, 1, 1, -1, 1, -1, 1, -1, 1, 1, -1, 1, -1, -1, -1, -1, -1]);
const normals   = new Float32Array([1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, -1, 0, 0, -1, 0, 0, -1, 0, 0, -1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, -1, 0, 0, -1, 0, 0, -1, 0, 0, -1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, -1, 0, 0, -1, 0, 0, -1, 0, 0, -1]);
const texcoords = new Float32Array([1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1]);
const indices   = new Uint16Array([0, 1, 2, 0, 2, 3, 4, 5, 6, 4, 6, 7, 8, 9, 10, 8, 10, 11, 12, 13, 14, 12, 14, 15, 16, 17, 18, 16, 18, 19, 20, 21, 22, 20, 22, 23]);

const positionBuffer = createBuffer(gl, positions);
const normalBuffer = createBuffer(gl, normals);
const texcoordBuffer = createBuffer(gl, texcoords);
const indicesBuffer = createBuffer(gl, indices, gl.ELEMENT_ARRAY_BUFFER);
```

WebGPU

```
function createBuffer(device, data, usage) {
  const buffer = device.createBuffer({
    size: data.byteLength,
    usage,
    mappedAtCreation: true,
  });
  const dst = new data.constructor(buffer.getMappedRange());
  dst.set(data);
  buffer.unmap();
  return buffer;
}

const positions = new Float32Array([1, 1, -1, 1, 1, 1, 1, -1, 1, 1, -1, -1, -1, 1, 1, -1, 1, -1, -1, -1, -1, -1, -1, 1, -1, 1, 1, 1, 1, 1, 1, 1, -1, -1, 1, -1, -1, -1, -1, 1, -1, -1, 1, -1, 1, -1, -1, 1, 1, 1, 1, -1, 1, 1, -1, -1, 1, 1, -1, 1, -1, 1, -1, 1, 1, -1, 1, -1, -1, -1, -1, -1]);
const normals   = new Float32Array([1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, -1, 0, 0, -1, 0, 0, -1, 0, 0, -1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, -1, 0, 0, -1, 0, 0, -1, 0, 0, -1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, -1, 0, 0, -1, 0, 0, -1, 0, 0, -1]);
const texcoords = new Float32Array([1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1]);
const indices   = new Uint16Array([0, 1, 2, 0, 2, 3, 4, 5, 6, 4, 6, 7, 8, 9, 10, 8, 10, 11, 12, 13, 14, 12, 14, 15, 16, 17, 18, 16, 18, 19, 20, 21, 22, 20, 22, 23]);

const positionBuffer = createBuffer(device, positions, GPUBufferUsage.VERTEX);
const normalBuffer = createBuffer(device, normals, GPUBufferUsage.VERTEX);
const texcoordBuffer = createBuffer(device, texcoords, GPUBufferUsage.VERTEX);
const indicesBuffer = createBuffer(device, indices, GPUBufferUsage.INDEX);
```

You can see, at a glance, these are not too different. You call different functions, but otherwise it's pretty similar.

Creating a Texture

WebGL

```
const tex = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, tex);
gl.texImage2D(
    gl.TEXTURE_2D,
    0,    // level
    gl.RGBA,
    2,    // width
    2,    // height
    0,
    gl.RGBA,
    gl.UNSIGNED_BYTE,
    new Uint8Array([
      255, 255, 128, 255,
      128, 255, 255, 255,
      255, 128, 255, 255,
      255, 128, 128, 255,
    ]));
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
```

WebGPU

```
const tex = device.createTexture({
  size: [2, 2],
  format: 'rgba8unorm',
  usage:
    GPUTextureUsage.TEXTURE_BINDING |
    GPUTextureUsage.COPY_DST,
});
device.queue.writeTexture(
    { texture: tex },
    new Uint8Array([
      255, 255, 128, 255,
      128, 255, 255, 255,
      255, 128, 255, 255,
      255, 128, 128, 255,
    ]),
    { bytesPerRow: 8, rowsPerImage: 2 },
    { width: 2, height: 2 },
);

const sampler = device.createSampler({
  magFilter: 'nearest',
  minFilter: 'nearest',
});
```

Again, not all that different. One difference is there are usage flags in WebGPU that you need to set depending on what you plan to do with the texture. Another is that in WebGPU we need to create a sampler which is optional in WebGL

Compiling shaders

WebGL

```
function createShader(gl, type, source) {
  const sh = gl.createShader(type);
  gl.shaderSource(sh, source);
  gl.compileShader(sh);
  if (!gl.getShaderParameter(sh, gl.COMPILE_STATUS)) {
    throw new Error(gl.getShaderInfoLog(sh));
  }
  return sh;
}

const vs = createShader(gl, gl.VERTEX_SHADER, vSrc);
const fs = createShader(gl, gl.FRAGMENT_SHADER, fSrc);
```

WebGPU

```
const shaderModule = device.createShaderModule({code: shaderSrc});
```

A minor difference, unlike WebGL, we can compile multiple shaders at once.

In WebGL, if your shader didn't compile it is up to you to check the `COMPILE_STATUS` with `gl.getShaderParameter` and then if it failed, pull out the error messages with a call to `gl.getShaderInfoLog`. If you didn't do this no errors are shown. You'd likely just get an error later when you tried to use the shader program.

In WebGPU, most implementations will print an error to the JavaScript console. Of course you can still check for errors yourself but it's really nice that if you do nothing you'll still get some useful info.

Linking a Program / Setting up a Pipeline

A pipeline, or more specifically a "render pipeline", represents a pair of shaders used in a particular way. Several things that happen in WebGL are combined into one thing in WebGPU when creating a pipeline. For example, linking the shaders, setting up attributes parameters, choosing the draw mode (points, line, triangles), setting up how the depth buffer is used.

Here's the code.

WebGL

```
function createProgram(gl, vs, fs) {
  const prg = gl.createProgram();
  gl.attachShader(prg, vs);
  gl.attachShader(prg, fs);
  gl.linkProgram(prg);
  if (!gl.getProgramParameter(prg, gl.LINK_STATUS)) {
    throw new Error(gl.getProgramInfoLog(prg));
  }
  return prg;
}

const program = createProgram(gl, vs, fs);

...

gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
gl.vertexAttribPointer(positionLoc, 3, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(positionLoc);

gl.bindBuffer(gl.ARRAY_BUFFER, normalBuffer);
gl.vertexAttribPointer(normalLoc, 3, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(normalLoc);

gl.bindBuffer(gl.ARRAY_BUFFER, texcoordBuffer);
gl.vertexAttribPointer(texcoordLoc, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(texcoordLoc);

....

gl.enable(gl.DEPTH_TEST);
gl.enable(gl.CULL_FACE);
```

WebGPU

```
const pipeline = device.createRenderPipeline({
  layout: 'auto',
  vertex: {
    module: shaderModule,
    buffers: [
      // position
      {
        arrayStride: 3 * 4, // 3 floats, 4 bytes each
        attributes: [
          {shaderLocation: 0, offset: 0, format: 'float32x3'},
        ],
      },
      // normals
      {
        arrayStride: 3 * 4, // 3 floats, 4 bytes each
        attributes: [
          {shaderLocation: 1, offset: 0, format: 'float32x3'},
        ],
      },
      // texcoords
      {
        arrayStride: 2 * 4, // 2 floats, 4 bytes each
        attributes: [
          {shaderLocation: 2, offset: 0, format: 'float32x2',},
        ],
      },
    ],
  },
  fragment: {
    module: shaderModule,
    targets: [
      {format: presentationFormat},
    ],
  },
  primitive: {
    topology: 'triangle-list',
    cullMode: 'back',
  },
  depthStencil: {
    depthWriteEnabled: true,
    depthCompare: 'less',
    format: 'depth24plus',
  },
  ...(canvasInfo.sampleCount > 1 && {
      multisample: {
        count: canvasInfo.sampleCount,
      },
  }),
});
```

Parts to note:

Shader linking happens when you call `createRenderPipeline` and in fact `createRenderPipeline` is a slow call as your shaders might be adjusted internally depending on the settings. You can see, for `vertex` and `fragment` we specify a shader `module` and specify which function to call via `entryPoint`. WebGPU then needs to make sure those 2 functions are compatible with each other in the same way that linking two shaders into a program in WebGL checks the shaders are compatible with each other.

In WebGL we call `gl.vertexAttribPointer` to attach the current `ARRAY_BUFFER` buffer to an attribute *and* to specify how to pull data out of that buffer. In WebGPU we only specify how to pull data out of buffers when creating the pipeline. We specify what buffers to use later.

In the example above you can see `buffers` is an array of objects. Those objects are called `GPUVertexBufferLayout`. Within each one is an array of attributes. Here we're setting up to get our data from 3 different buffers. If we interleaved the data into one buffer we'd only need one `GPUVertexBufferLayout` but its `attribute` array would have 3 entries.

Also note, here is a place where we have to match `shaderLocation` to what we used in the shader.

In WebGPU we set up the primitive type, cull mode, and depth settings here. That means if we want to draw something with any of those settings different, for example if we want to draw some geometry with triangles and later with lines, we have to create multiple pipelines. Similarly if the vertex layouts are different. For example if one model has positions and texture coordinates separated in separated buffers, another has them in the same buffer but offset, and yet another has them interleaved, all 3 would require their own pipeline.

The last part, `multisample`, we need if we're drawing to a multi-sampled destination texture. I put that in here because by default, WebGL will use a multi sampled texture for the canvas. To emulate that requires adding a `multisample` property. `presentationFormat` and `canvasInfo.sampleCount` are something we'll cover below.

Preparing for uniforms

WebGL

```
const u_lightDirectionLoc = gl.getUniformLocation(program, 'u_lightDirection');
const u_diffuseLoc = gl.getUniformLocation(program, 'u_diffuse');
const u_worldInverseTransposeLoc = gl.getUniformLocation(program, 'u_worldInverseTranspose');
const u_worldViewProjectionLoc = gl.getUniformLocation(program, 'u_worldViewProjection');
```

WebGPU

```
const vUniformBufferSize = 2 * 16 * 4; // 2 mat4s * 16 floats per mat * 4 bytes per float
const fUniformBufferSize = 3 * 4;      // 1 vec3 * 3 floats per vec3 * 4 bytes per float

const vsUniformBuffer = device.createBuffer({
  size: vUniformBufferSize,
  usage: GPUBufferUsage.UNIFORM | GPUBufferUsage.COPY_DST,
});
const fsUniformBuffer = device.createBuffer({
  size: fUniformBufferSize,
  usage: GPUBufferUsage.UNIFORM | GPUBufferUsage.COPY_DST,
});
const vsUniformValues = new Float32Array(2 * 16); // 2 mat4s
const worldViewProjection = vsUniformValues.subarray(0, 16);
const worldInverseTranspose = vsUniformValues.subarray(16, 32);
const fsUniformValues = new Float32Array(3);  // 1 vec3
const lightDirection = fsUniformValues.subarray(0, 3);
```

In WebGL we look up the locations of the uniforms. In WebGPU we create buffers to hold the values of the uniforms. The code above then creates TypedArray views into larger CPU side TypedArrays that hold the values for the uniforms. Notice `vUniformBufferSize` and `fUniformBufferSize` are hand computed. Similarly, when creating views into the typed arrays the offsets and sizes are hand computed. It's entirely up to you to do those calculations. Unlike WebGL, WebGPU provides no API to query these offsets and sizes.

Note, a similar process exists for WebGL2 using Uniform Blocks but if you've never used Uniform Blocks then this will be new.

Preparing to draw

In WebGL we'd get straight to drawing at this point but in WebGPU we have some work left.

We need to create a bind group. This lets us specify what resources our shaders will use

WebGL

```
// happens at render time
gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, tex);
```

WebGPU

```
// can happen at init time
const bindGroup = device.createBindGroup({
  layout: pipeline.getBindGroupLayout(0),
  entries: [
    { binding: 0, resource: { buffer: vsUniformBuffer } },
    { binding: 1, resource: { buffer: fsUniformBuffer } },
    { binding: 2, resource: sampler },
    { binding: 3, resource: tex.createView() },
  ],
});
```

Again, notice the `binding` and `group` must match what we specified in our shaders.

In WebGPU we also create a render pass descriptor vs WebGL where these settings are set via stateful API calls or handled automatically.

WebGL

```
gl.clearColor(0.5, 0.5, 0.5, 1.0);
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

WebGPU

```
const renderPassDescriptor = {
  colorAttachments: [
    {
      // view: undefined, // Assigned later
      // resolveTarget: undefined, // Assigned Later
      clearValue: [0.5, 0.5, 0.5, 1],
      loadOp: 'clear',
      storeOp: 'store',
    },
  ],
  depthStencilAttachment: {
    // view: undefined,  // Assigned later
    depthClearValue: 1,
    depthLoadOp: 'clear',
    depthStoreOp: 'store',
  },
};
```

Note that many of the settings in WebGPU are related to where we want to render. In WebGL, when rendering to the canvas, all of this was handled for us. When rendering to a framebuffer in WebGL these settings are the equivalent of calls to `gl.framebufferTexture2D` and/or `gl.framebufferRenderbuffer`.

Setting Uniforms

WebGL

```
gl.uniform3fv(u_lightDirectionLoc, v3.normalize([1, 8, -10]));
gl.uniform1i(u_diffuseLoc, 0);
gl.uniformMatrix4fv(u_worldInverseTransposeLoc, false, m4.transpose(m4.inverse(world)));
gl.uniformMatrix4fv(u_worldViewProjectionLoc, false, m4.multiply(viewProjection, world));
```

WebGPU

```
m4.transpose(m4.inverse(world), worldInverseTranspose);
m4.multiply(viewProjection, world, worldViewProjection);

v3.normalize([1, 8, -10], lightDirection);

device.queue.writeBuffer(vsUniformBuffer, 0, vsUniformValues);
device.queue.writeBuffer(fsUniformBuffer, 0, fsUniformValues);
```

In the WebGL case we compute a value and pass it to `gl.uniform???` with the appropriate location.

In the WebGPU case we write values into our typed arrays and then copy the contents of those typed arrays to the corresponding GPU buffers.

Note: In WebGL2, if we were using Uniform Blocks, this process is almost exactly the same except we'd call `gl.bufferSubData` to upload the typed array contents.

Resizing the drawing buffer

As mentioned near the start of the article, this is one place that WebGL just handled for us but in WebGPU we need to do ourselves.

WebGL

```
function resizeCanvasToDisplaySize(canvas) {
  const width = canvas.clientWidth;
  const height = canvas.clientHeight;
  const needResize = width !== canvas.width || height !== canvas.height;
  if (needResize) {
    canvas.width = width;
    canvas.height = height;
  }
  return needResize;
}
```

WebGPU

```
// At init time
const canvas = document.querySelector('canvas');
const context = canvas.getContext('webgpu');

const presentationFormat = navigator.gpu.getPreferredFormat(adapter);
context.configure({
  device,
  format: presentationFormat,
});

const canvasInfo = {
  canvas,
  presentationFormat,
  // these are filled out in resizeToDisplaySize
  renderTarget: undefined,
  renderTargetView: undefined,
  depthTexture: undefined,
  depthTextureView: undefined,
  sampleCount: 4,  // can be 1 or 4
};

// --- At render time ---

function resizeToDisplaySize(device, canvasInfo) {
  const {
    canvas,
    context,
    renderTarget,
    presentationFormat,
    depthTexture,
    sampleCount,
  } = canvasInfo;
  const width = Math.max(1, Math.min(device.limits.maxTextureDimension2D, canvas.clientWidth));
  const height = Math.max(1, Math.min(device.limits.maxTextureDimension2D, canvas.clientHeight));

  const needResize = !canvasInfo.renderTarget ||
                     width !== canvas.width ||
                     height !== canvas.height;
  if (needResize) {
    if (renderTarget) {
      renderTarget.destroy();
    }
    if (depthTexture) {
      depthTexture.destroy();
    }

    canvas.width = width;
    canvas.height = height;

    if (sampleCount > 1) {
      const newRenderTarget = device.createTexture({
        size: [canvas.width, canvas.height],
        format: presentationFormat,
        sampleCount,
        usage: GPUTextureUsage.RENDER_ATTACHMENT,
      });
      canvasInfo.renderTarget = newRenderTarget;
      canvasInfo.renderTargetView = newRenderTarget.createView();
    }

    const newDepthTexture = device.createTexture({
      size: [canvas.width, canvas.height,
      format: 'depth24plus',
      sampleCount,
      usage: GPUTextureUsage.RENDER_ATTACHMENT,
    });
    canvasInfo.depthTexture = newDepthTexture;
    canvasInfo.depthTextureView = newDepthTexture.createView();
  }
  return needResize;
}
```

You can see above there's a bunch of work to do. If we need to resize, we need to manually destroy the old textures (color and depth) and create new ones. We also need to check that we don't go over the limits, something WebGL handled for us, at least for the canvas.

Above, the property `sampleCount` is effectively the analog of `antialias` property of the WebGL context's creation attributes. `sampleCount: 4` would be the equivalent of WebGL's `antialias: true` (the default), whereas `sampleCount: 1` would be the equivalent of `antialias: false` when creating a WebGL context.

Another thing not shown above, WebGL would try not to run out of memory meaning if you asked for a 16000x16000 canvas, WebGL might give you back a 4096x4096 canvas. You could find out what you actually got back by looking at `gl.drawingBufferWidth` and `gl.drawingBufferHeight`.

The reasons WebGL did this are (1) stretching a canvas across multiple monitors might make the size larger than the GPU can handle (2) the system might be low on memory and instead of just crashing, WebGL would return a smaller drawingbuffer.

In WebGPU, checking those 2 situations is up to you. We're checking for situation (1) above. For situation (2) we'd have to check for out of memory ourselves and like everything else in WebGPU, doing so is asynchronous.

```
device.pushErrorScope('out-of-memory');
context.configure({...});
if (sampleCount > 1) {
  const newRenderTarget = device.createTexture({...});
  ...
}

const newDepthTexture = device.createTexture({...});
...
device.popErrorScope().then(error => {
  if (error) {
    // we're out of memory, try a smaller size?
  }
});
```

Drawing

WebGL

```
gl.viewport(0, 0, gl.canvas.width, gl.canvas.height);
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

...
gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, tex);

gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
gl.vertexAttribPointer(positionLoc, 3, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(positionLoc);

gl.bindBuffer(gl.ARRAY_BUFFER, normalBuffer);
gl.vertexAttribPointer(normalLoc, 3, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(normalLoc);

gl.bindBuffer(gl.ARRAY_BUFFER, texcoordBuffer);
gl.vertexAttribPointer(texcoordLoc, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(texcoordLoc);

...

gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indicesBuffer);

gl.drawElements(gl.TRIANGLES, 6 * 6, gl.UNSIGNED_SHORT, 0);
```

WebGPU

```
if (canvasInfo.sampleCount === 1) {
    const colorTexture = context.getCurrentTexture();
    renderPassDescriptor.colorAttachments[0].view = colorTexture.createView();
} else {
  renderPassDescriptor.colorAttachments[0].view = canvasInfo.renderTargetView;
  renderPassDescriptor.colorAttachments[0].resolveTarget = context.getCurrentTexture().createView();
}
renderPassDescriptor.depthStencilAttachment.view = canvasInfo.depthTextureView;

const commandEncoder = device.createCommandEncoder();
const passEncoder = commandEncoder.beginRenderPass(renderPassDescriptor);
passEncoder.setPipeline(pipeline);
passEncoder.setBindGroup(0, bindGroup);
passEncoder.setVertexBuffer(0, positionBuffer);
passEncoder.setVertexBuffer(1, normalBuffer);
passEncoder.setVertexBuffer(2, texcoordBuffer);
passEncoder.setIndexBuffer(indicesBuffer, 'uint16');
passEncoder.drawIndexed(indices.length);
passEncoder.end();
device.queue.submit([commandEncoder.finish()]);
```

Note that I repeated the WebGL attribute setup code here. In WebGL, this can happen at init time or at render time. In WebGPU we set up how to pull data out of the buffers at init time, but we set the actual buffers to use at render time.

In WebGPU, we need to update our render pass descriptor to use the textures we may have just updated in `resizeToDisplaySize`. Then we need to create a command encoder and begin a render pass.

Inside the render pass we set the pipeline, which is kind of like the equivalent of `gl.useProgram`. We then set our bind group which supplies our sampler, texture, and the 2 buffers for our uniforms. We set the vertex buffers to match what we declared earlier. Finally, we set an index buffer and call `drawIndexed`, which is the equivalent of calling `gl.drawElements`.

Back in WebGL we needed to call `gl.viewport`. In WebGPU, the pass encoder defaults to a viewport that matches the size of the attachments so unless we want a viewport that doesn't match we don't have to set a viewport separately.

In WebGL we called `gl.clear` to clear the canvas. Whereas in WebGPU we had previously set that up when creating our render pass descriptor.

Working Examples:

WebGL

WebGPU

Another important thing to notice, We're issuing instructions to something referred to as the `device.queue`. Notice that when we uploaded the values for the uniforms we called `device.queue.writeBuffer` and then when we created a command encoder and submitted it with `device.queue.submit`. That should make it pretty clear that we can't update the buffers between draw calls within the same command encoder. If we want to draw multiple things we'd need multiple buffers or multiple sets of values in a single buffer.

Drawing Multiple Things

Let's go over an example of drawing multiple things.

As mentioned above, to draw multiple things, at least in the most common way, we'd need a different uniform buffer for each thing so that we can provide a different set of matrices. Uniform buffers are passed in via bind groups so we also need a different bind group per object.

WebGL

```
  const numObjects = 100;
  const objectInfos = [];

  for (let i = 0; i < numObjects; ++i) {
    const across = Math.sqrt(numObjects) | 0;
    const x = (i % across - (across - 1) / 2) * 3;
    const y = ((i / across | 0) - (across - 1) / 2) * 3;

    objectInfos.push({
      translation: [x, y, 0],
    });
  }
```

WebGPU

```
  const vUniformBufferSize = 2 * 16 * 4; // 2 mat4s * 16 floats per mat * 4 bytes per float
  const fUniformBufferSize = 3 * 4;      // 1 vec3 * 3 floats per vec3 * 4 bytes per float

  const fsUniformBuffer = device.createBuffer({
    size: Math.max(16, fUniformBufferSize),
    usage: GPUBufferUsage.UNIFORM | GPUBufferUsage.COPY_DST,
  });
  const fsUniformValues = new Float32Array(3);  // 1 vec3
  const lightDirection = fsUniformValues.subarray(0, 3);

  const numObjects = 100;
  const objectInfos = [];

  for (let i = 0; i < numObjects; ++i) {
    const vsUniformBuffer = device.createBuffer({
      size: Math.max(16, vUniformBufferSize),
      usage: GPUBufferUsage.UNIFORM | GPUBufferUsage.COPY_DST,
    });

    const vsUniformValues = new Float32Array(2 * 16); // 2 mat4s
    const worldViewProjection = vsUniformValues.subarray(0, 16);
    const worldInverseTranspose = vsUniformValues.subarray(16, 32);

    const bindGroup = device.createBindGroup({
      layout: pipeline.getBindGroupLayout(0),
      entries: [
        { binding: 0, resource: { buffer: vsUniformBuffer } },
        { binding: 1, resource: { buffer: fsUniformBuffer } },
        { binding: 2, resource: sampler },
        { binding: 3, resource: tex.createView() },
      ],
    });

    const across = Math.sqrt(numObjects) | 0;
    const x = (i % across - (across - 1) / 2) * 3;
    const y = ((i / across | 0) - (across - 1) / 2) * 3;

    objectInfos.push({
      vsUniformBuffer,  // needed to update the buffer
      vsUniformValues,  // needed to update the buffer
      worldViewProjection,  // needed so we can update this object's worldViewProject
      worldInverseTranspose,  // needed so we can update this object's worldInverseTranspose
      bindGroup, // needed to render this object
      translation: [x, y, 0],
    });
  }
```

Note that in this example we're sharing the `fsUniforms`, its buffer and values which contains the lighting direction we included `fsUniformBuffer` in the bind group but it's defined outside of the loop as there is only 1.

For rendering we'll setup the shared parts, then for each object, update its uniform values, copy those to the corresponding uniform buffer, and encode the command to draw it.

WebGL

```
  function render(time) {
    time *= 0.001;
    resizeCanvasToDisplaySize(gl.canvas);
    gl.viewport(0, 0, gl.canvas.width, gl.canvas.height);

    gl.enable(gl.DEPTH_TEST);
    gl.enable(gl.CULL_FACE);
    gl.clearColor(0.5, 0.5, 0.5, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    gl.useProgram(program);

    const projection = mat4.perspective(30 * Math.PI / 180, gl.canvas.clientWidth / gl.canvas.clientHeight, 0.5, 100);
    const eye = [1, 4, -46];
    const target = [0, 0, 0];
    const up = [0, 1, 0];

    const view = mat4.lookAt(eye, target, up);
    const viewProjection = mat4.multiply(projection, view);

    gl.uniform3fv(u_lightDirectionLoc, vec3.normalize([1, 8, -10]));
    gl.uniform1i(u_diffuseLoc, 0);

    gl.activeTexture(gl.TEXTURE0);
    gl.bindTexture(gl.TEXTURE_2D, tex);

    gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
    gl.vertexAttribPointer(positionLoc, 3, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(positionLoc);

    gl.bindBuffer(gl.ARRAY_BUFFER, normalBuffer);
    gl.vertexAttribPointer(normalLoc, 3, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(normalLoc);

    gl.bindBuffer(gl.ARRAY_BUFFER, texcoordBuffer);
    gl.vertexAttribPointer(texcoordLoc, 2, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(texcoordLoc);

    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indicesBuffer);
    objectInfos.forEach(({translation}, ndx) => {
    const world = mat4.translation(translation);
    mat4.rotateX(world, time * 0.9 + ndx, world);
    mat4.rotateY(world, time + ndx, world);

    gl.uniformMatrix4fv(u_worldInverseTransposeLoc, false, mat4.transpose(mat4.inverse(world)));
    gl.uniformMatrix4fv(u_worldViewProjectionLoc, false, mat4.multiply(viewProjection, world));

    gl.drawElements(gl.TRIANGLES, 6 * 6, gl.UNSIGNED_SHORT, 0);
    });

    requestAnimationFrame(render);
  }
```

WebGPU

```
  function render(time) {
    time *= 0.001;
    resizeToDisplaySize(device, canvasInfo);

    if (canvasInfo.sampleCount === 1) {
        const colorTexture = context.getCurrentTexture();
        renderPassDescriptor.colorAttachments[0].view = colorTexture.createView();
    } else {
      renderPassDescriptor.colorAttachments[0].view = canvasInfo.renderTargetView;
      renderPassDescriptor.colorAttachments[0].resolveTarget = context.getCurrentTexture().createView();
    }
    renderPassDescriptor.depthStencilAttachment.view = canvasInfo.depthTextureView;

    const commandEncoder = device.createCommandEncoder();
    const passEncoder = commandEncoder.beginRenderPass(renderPassDescriptor);

    // Of course these could be per object but since we're drawing the same object
    // multiple times, just set them once.
    passEncoder.setPipeline(pipeline);
    passEncoder.setVertexBuffer(0, positionBuffer);
    passEncoder.setVertexBuffer(1, normalBuffer);
    passEncoder.setVertexBuffer(2, texcoordBuffer);
    passEncoder.setIndexBuffer(indicesBuffer, 'uint16');

    const projection = mat4.perspective(30 * Math.PI / 180, canvas.clientWidth / canvas.clientHeight, 0.5, 100);
    const eye = [1, 4, -46];
    const target = [0, 0, 0];
    const up = [0, 1, 0];

    const view = mat4.lookAt(eye, target, up);
    const viewProjection = mat4.multiply(projection, view);

    // the lighting info is shared so set these uniforms once
    vec3.normalize([1, 8, -10], lightDirection);
    device.queue.writeBuffer(fsUniformBuffer, 0, fsUniformValues);

    objectInfos.forEach(({
      vsUniformBuffer,
      vsUniformValues,
      worldViewProjection,
      worldInverseTranspose,
      bindGroup,
      translation,
    }, ndx) => {
      passEncoder.setBindGroup(0, bindGroup);

      const world = mat4.translation(translation);
      mat4.rotateX(world, time * 0.9 + ndx, world);
      mat4.rotateY(world, time + ndx, world);
      mat4.transpose(mat4.inverse(world), worldInverseTranspose);
      mat4.multiply(viewProjection, world, worldViewProjection);

      device.queue.writeBuffer(vsUniformBuffer, 0, vsUniformValues);
      passEncoder.drawIndexed(indices.length);
    });
    passEncoder.end();
    device.queue.submit([commandEncoder.finish()]);

    requestAnimationFrame(render);
  }
  requestAnimationFrame(render);
}
```

There isn't much difference from our single cube but the code has been slightly re-arranged to put shared things outside the object loop. In this particular case, as we're drawing the same cube 100 times we don't need to update vertex buffers or index buffers, but of course we could change those per object if we needed to.

WebGL

WebGPU

The important part to take away is that unlike WebGL, you'll need uniform buffers for any uniforms that are object specific (like a world matrix), and, because of that you also may need a unique bind group per object.

Other random differences

Z clip space is 0 to 1

In WebGL Z clip space was -1 to +1. In WebGPU it's 0 to 1 (which btw makes way more sense!)

Y axis is down in framebuffer, viewport coordinates

This is the opposite of WebGL though in clip space Y axis is up (same as WebGL)

In other words, returning (-1, -1) from a vertex shader will reference the lower left corner in both WebGL and WebGPU. On the other hand, setting the viewport or scissor to `0, 0, 1, 1` references the lower left corner in WebGL but the upper left corner in WebGPU.

WGSL uses `@builtin(???)` for GLSL's `gl_XXX` variables.

`gl_FragCoord` is `@builtin(position) myVarOrField: vec4f` and unlike WebGL, goes down the screen instead of up so 0,0 is the top left vs WebGL where 0,0 is the bottom left.

`gl_VertexID` is `@builtin(vertex_index) myVarOrField: u32`

`gl_InstanceID` is `@builtin(instance_index) myVarOrField: u32`

`gl_Position` is `@builtin(position) vec4f` which may be the return value of a vertex shader or a field in a structure returned by the vertex shader

There is no `gl_PointSize` and `gl_PointCoord` equivalent because points are only 1 pixel in WebGPU. Fortunately it's easy to draw points yourself.

You can see other builtin variables here.

WGSL only supports lines and points 1 pixel wide

According to the spec, WebGL2 could support lines larger than 1 pixel, but in actual practice no implementations did. WebGL2 did generally support points larger than 1 pixel but, (a) lots of GPUs only supported a max size of 64 pixels and (b) different GPU would clip or not clip based on the center of the point. So, it's arguably a good thing WebGPU doesn't support points of sizes other than 1. This forces you to implement a portable point solution.

WebGPU optimizations are different than WebGL

If you take a WebGL app and directly convert it to WebGPU you might find it runs slower. To get the benefits of WebGPU you'll need to change the way you organize data and optimize how you draw. See this article on WebGPU optimization for ideas.

Note: If you are comparing WebGL to WebGPU in the article on optimization here are 2 WebGL samples you can use to compare

- Drawing up to 30000 objects in WebGL using standard WebGL uniforms
- Drawing up to 30000 objects in WebGL using uniform blocks
- Drawing up to 30000 objects in WebGL using global/material/per object uniform blocks
- Drawing up to 30000 objects in WebGL using one large uniform buffer

Another article, if you're comparing performance of WebGL vs WebGPU see this article.

If you were already familiar with WebGL then I hope this article was useful.