WebGPU Fundamentals

This article will try to teach you the very fundamentals of WebGPU.

If you already know WebGL, read this.

WebGPU is an API that lets you do 2 basic things.

1. Draw triangles/points/lines to textures

2. Run computations on the GPU

That is it!

Everything about WebGPU after that is up to you. It's like learning a computer language like JavaScript, or Rust, or C++. First you learn the basics, then it's up to you to creatively use those basics to solve your problem.

WebGPU is an extremely low-level API. While you can make some small examples, for many apps it will likely require a large amount of code and some serious organization of data. As an example, three.js which supports WebGPU consists of ~550k bytes of minified JavaScript, and that's just its base library. That does not include loaders, controls, post-processing, and many other features. Similarly, there's TensorFlow, which core plus the WebGPU backend is ~600k bytes of minified JavaScript and also does not include support for all of the various tensorflow optional features.

The point being, if you just want to get something on the screen you're far better off choosing a library that provides the large amount of code you're going to have to write when doing it yourself.

On the other hand, maybe you have a custom use case or maybe you want to modify an existing library or maybe you're just curious how it all works. In those cases, read on!

Getting Started

It's hard to decide where to start. At a certain level, WebGPU is a very simple system. All it does is run 3 types of functions on the GPU: Vertex Shaders, Fragment Shaders, and Compute Shaders.

A Vertex Shader computes vertices. The shader returns vertex positions. For every group of 3 vertices the vertex shader function returns, a triangle is drawn between those 3 positions.[1]

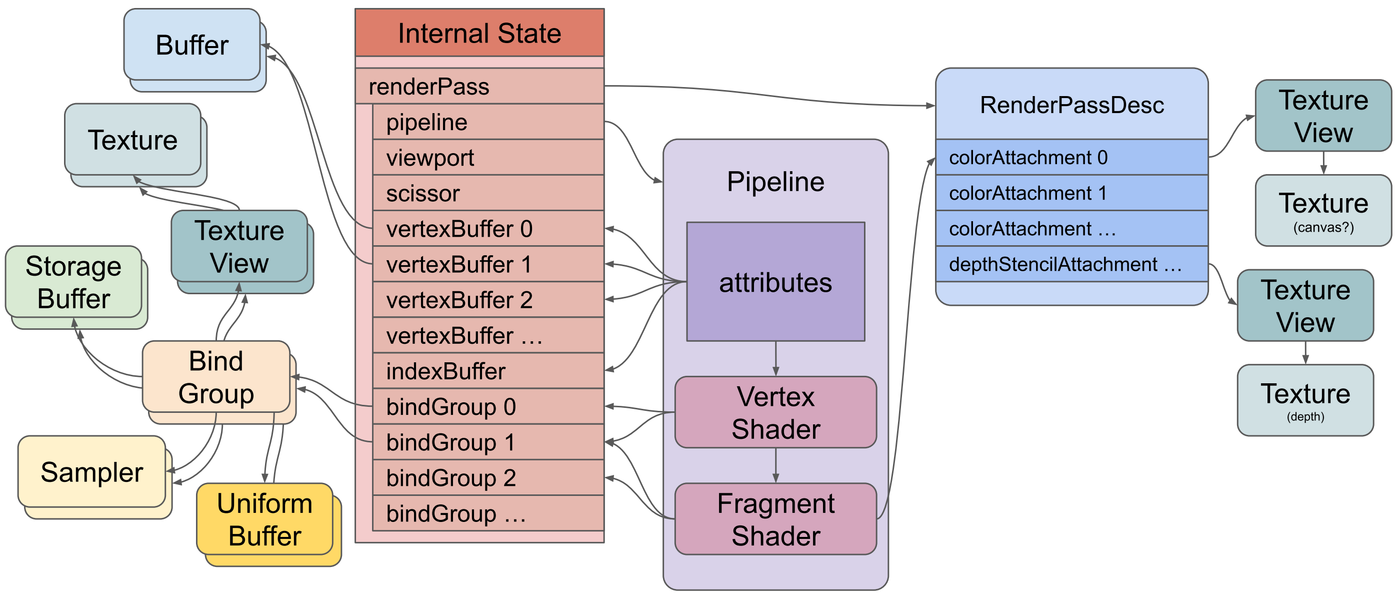
A Fragment Shader computes colors.[2] When a triangle is drawn, for each pixel to be drawn the GPU calls your fragment shader. The fragment shader then returns a color.

A Compute Shader is more generic. It's effectively just a function you call and say "execute this function N times". The GPU passes the iteration number each time it calls your function so you can use that number to do something unique on each iteration.

If you squint hard, you can think of these functions as similar to the functions to pass to `array.forEach` or `array.map`. The functions you run on the GPU are just functions, just like JavaScript functions. The part that differs is they run on the GPU, and so to run them you need to copy all the data you want them to access to the GPU in the form of buffers and textures and they only output to those buffers and textures. You need to specify in the functions which bindings or locations the function will look for the data. And, back in JavaScript, you need to bind the buffers and textures holding your data to the bindings or locations. Once you've done that you tell the GPU to execute the function.

#

Maybe a picture will help. Here is a *simplified* diagram of WebGPU setup to draw triangles by using a vertex shader and a fragment shader:



What to notice about this diagram:

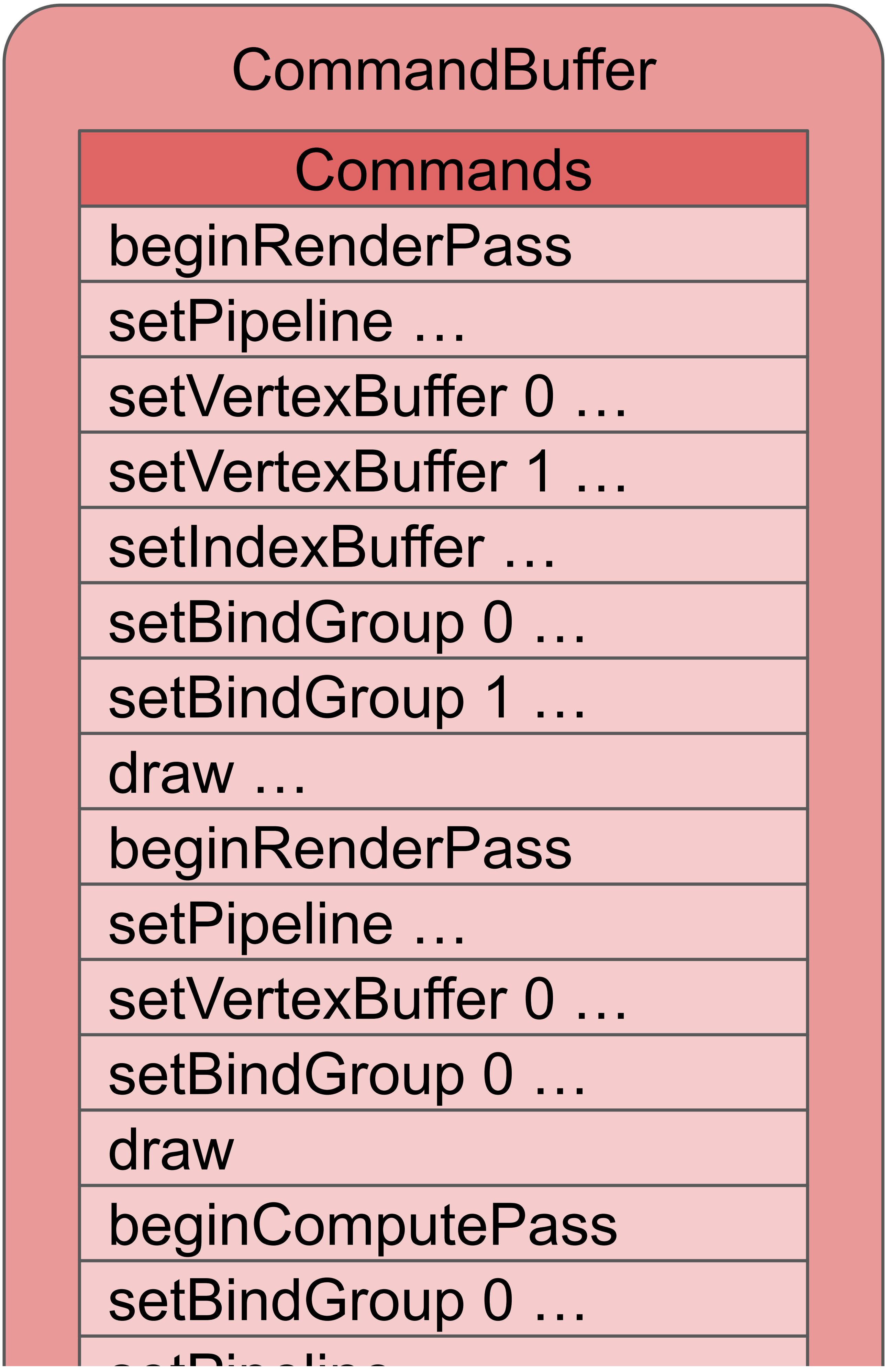
- There is a **Pipeline**. It contains the vertex shader and fragment shader the GPU will run. You could also have a pipeline with a compute shader.
- The shaders reference resources (buffers, textures, samplers) indirectly through **Bind Groups**
- The pipeline defines attributes that reference buffers indirectly through the internal state
- Attributes pull data out of buffers and feed the data into the vertex shader
- The vertex shader may feed data into the fragment shader
- The fragment shader writes to textures indirectly through the render pass description

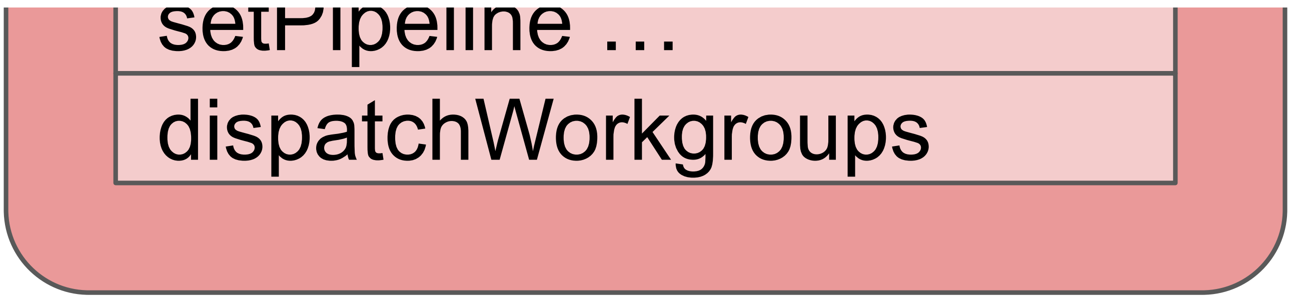
To execute shaders on the GPU, you need to create all of these resources and set up this state. Creation of resources is relatively straightforward. One interesting thing is that most WebGPU resources can not be changed after creation. You can change their contents but not their size, usage, format, etc… If you want to change any of that stuff you create a new resource and destroy the old one.

Some of the state is set up by creating and then executing command buffers. Command buffers are literally what their name suggests. They are a buffer of commands. You create command encoders. The encoders encode commands into the command buffer. You then *finish* the encoder and it gives you the command buffer it created. You can then *submit* that command buffer to have WebGPU execute the commands.

Here is some pseudo-code for encoding a command buffer followed by a representation of the command buffer that was created.

```
• encoder = device.createCommandEncoder()
• // draw something
• {
•   pass = encoder.beginRenderPass(...)
•   pass.setPipeline(...)
•   pass.setVertexBuffer(0, …)
•   pass.setVertexBuffer(1, …)
•   pass.setIndexBuffer(...)
•   pass.setBindGroup(0, …)
•   pass.setBindGroup(1, …)
•   pass.draw(...)
•   pass.end()
• }
• // draw something else
• {
•   pass = encoder.beginRenderPass(...)
•   pass.setPipeline(...)
•   pass.setVertexBuffer(0, …)
•   pass.setBindGroup(0, …)
•   pass.draw(...)
•   pass.end()
• }
• // compute something
• {
•   pass = encoder.beginComputePass(...)
•   pass.beginComputePass(...)
•   pass.setBindGroup(0, …)
•   pass.setPipeline(...)
•   pass.dispatchWorkgroups(...)
•   pass.end();
• }
• commandBuffer = encoder.finish();
```


CommandBuffer
Commands
beginRenderPass
setPipeline …
setVertexBuffer 0 …
setVertexBuffer 1 …
setIndexBuffer …
setBindGroup 0 …
setBindGroup 1 …
draw …
beginRenderPass
setPipeline …
setVertexBuffer 0 …
setBindGroup 0 …
draw
beginComputePass
setBindGroup 0 …



Once you create a command buffer, you can *submit* it to be executed:

- `device.queue.submit([commandBuffer]);`

The ‘simplified diagram of WebGPU setup’ shown previously represents the state at a *single* `draw` command in the command buffer. Executing the commands will set up the *internal state* and then the `draw` command will tell the GPU to execute a vertex shader (and indirectly a fragment shader). The `dispatchWorkgroup` command will tell the GPU to execute a compute shader.

I hope that gave you some mental image of the state you need to set up. Like mentioned above, WebGPU has 2 basic things it can do:

1. Draw triangles/points/lines to textures

2. Run computations on the GPU

We’ll go over a small example of doing each of those things. Other articles will show the various ways of providing data to these things. Note that this will be very basic. We need to build up a foundation of these basics. Later we’ll show how to use them to do things people typically do with GPUs like 2D graphics, 3D graphics, etc…

#

Drawing triangles to textures

WebGPU can draw triangles to textures. For the purpose of this article, a texture is a 2D rectangle of pixels.[3] The `<canvas>` element represents a texture on a webpage. In WebGPU we can ask the canvas for a texture and then render to that texture.

To draw triangles with WebGPU we have to supply 2 “shaders”. Again, Shaders are functions that run on the GPU. These 2 shaders are:

1. Vertex Shaders

   Vertex shaders are functions that compute vertex positions for drawing triangles/lines/points

2. Fragment Shaders

   Fragment shaders are functions that compute the color (or other data) for each pixel to be drawn/rasterized when drawing triangles/lines/points

Let’s start with a very small WebGPU program to draw a triangle.

We need a canvas to display our triangle:

```
<canvas></canvas>
```

then we need a `<script>` tag to hold our JavaScript:

```
<canvas></canvas>
<script type="module">

... javascript goes here ...

</script>
```

All of the JavaScript below will go inside this script tag.

WebGPU is an asynchronous API so it's easiest to use in an async function. We start off by requesting an adapter, and then requesting a device from the adapter.

```
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();
```

The code above is fairly self-explanatory. First, we request an adapter by using the `?.` optional chaining operator. so that if `navigator.gpu` does not exist then `adapter` will be undefined. If it does exist then we'll call `requestAdapter`. It returns its results asynchronously so we need `await`. The adapter represents a specific GPU. Some devices have multiple GPUs.

From the adapter, we request the device but again use `?.` so that if adapter happens to be undefined then device will also be undefined.

If the `device` is not set, it's likely the user has an old browser.

Next, we look up the canvas and create a `webgpu` context for it. This will let us get a texture to render to. That texture will be used to display the canvas in the webpage.

```
  // Get a WebGPU context from the canvas and configure it
  const canvas = document.querySelector('canvas');
  const context = canvas.getContext('webgpu');
  const presentationFormat = navigator.gpu.getPreferredCanvasFormat();
  context.configure({
    device,
    format: presentationFormat,
  });
```

Again, the code above is pretty self-explanatory. We get a `"webgpu"` context from the canvas. We ask the system what the preferred canvas format is. This will be either `"rgba8unorm"` or `"bgra8unorm"`. It's not really that important what it is but querying it will make things faster for the user's system.

We pass that as `format` into the webgpu canvas context by calling `configure`. We also pass in the `device` which associates this canvas with the device we just created.

Next, we create a shader module. A shader module contains one or more shader functions. In our case, we'll make 1 vertex shader function and 1 fragment shader function.

```
  const module = device.createShaderModule({
    label: 'our hardcoded red triangle shaders',
    code: `
      @vertex fn vs(
        @builtin(vertex_index) vertexIndex : u32
      ) -> @builtin(position) vec4f {
        let pos = array(
          vec2f( 0.0,  0.5),  // top center
          vec2f(-0.5, -0.5),  // bottom left
          vec2f( 0.5, -0.5)   // bottom right
        );

        return vec4f(pos[vertexIndex], 0.0, 1.0);
      }

      @fragment fn fs() -> @location(0) vec4f {
        return vec4f(1.0, 0.0, 0.0, 1.0);
      }
    `,
  });
```

Shaders are written in a language called WebGPU Shading Language (WGSL) which is often pronounced wig-sil. WGSL is a strongly typed language which we'll try to go over in more detail in another article. For now, I'm hoping with a little explanation you can infer some basics.

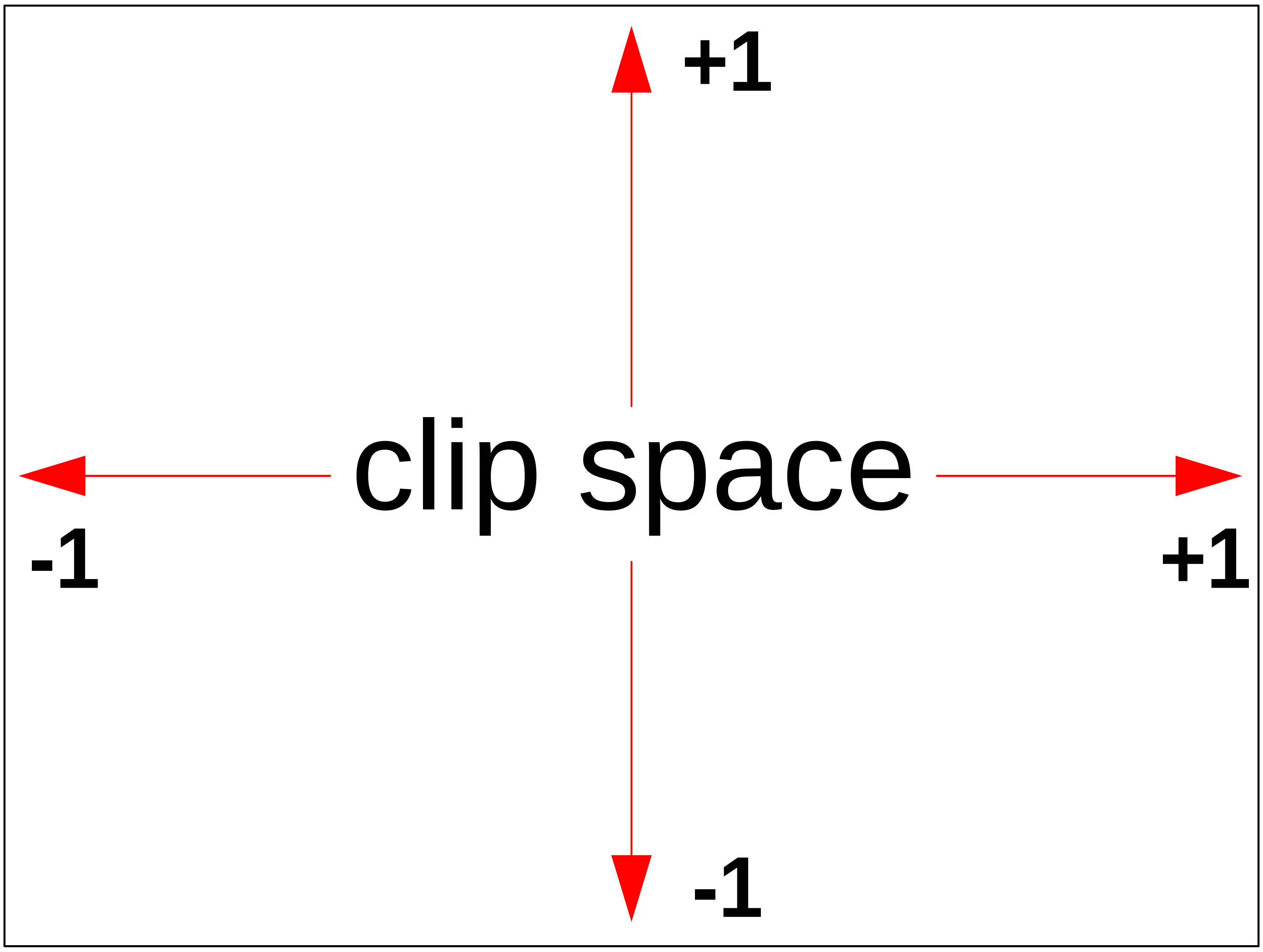
Above we see a function called `vs` is declared with the `@vertex` attribute. This designates it as a vertex shader function.

```
      @vertex fn vs(
        @builtin(vertex_index) vertexIndex : u32
      ) -> @builtin(position) vec4f {
         ...
```

It accepts one parameter we named `vertexIndex`. `vertexIndex` is a `u32` which means a *32-bit unsigned integer*. It gets its value from the builtin called `vertex_index`. `vertex_index` is like an iteration number, similar to `index` in JavaScript's `Array.map(function(value, index) { ... })`. If we tell the GPU to execute this function 10 times by calling `draw`, the first time `vertex_index` would be `0`, the 2nd time it would be `1`, the 3rd time it would be `2`, etc…[4]

Our `vs` function is declared as returning a `vec4f` which is a vector of four 32-bit floating point values. Think of it as an array of 4 values or an object with 4 properties like `{x: 0, y: 0, z: 0, w: 0}`. This returned value will be assigned to the `position` builtin. In "triangle-list" mode, every 3 times the vertex shader is executed a triangle will be drawn connecting the 3 `position` values we return.

Positions in WebGPU need to be returned in *clip space* where X goes from -1.0 on the left to +1.0 on the right, and Y goes from -1.0 at the bottom to +1.0 at the top. This is true regardless of the size of the texture we are drawing to.



The vs function declares an array of 3 `vec2f`s. Each `vec2f` consists of two 32-bit floating point values.

- `    let pos = array(`
- `      vec2f( 0.0,  0.5),  // top center`
- `      vec2f(-0.5, -0.5),  // bottom left`
- `      vec2f( 0.5, -0.5)   // bottom right`
- `    );`

Finally it uses `vertexIndex` to return one of the 3 values from the array. Since the function requires 4 floating point values for its return type, and since `pos` is an array of `vec2f`, the code supplies `0.0` and `1.0` for the remaining 2 values.

- `    return vec4f(pos[vertexIndex], 0.0, 1.0);`

Note that for drawing something in 2D we usually only need the x and y values for position. The z value is used for depth testing and will come up in [the article on orthographic projection](). The w value is used for perspective divide and will come up in [the article on perspective projection](). For now, setting z to 0.0 and w to 1.0 is what we need to draw the triangle.

The shader module also declares a function called `fs` that is declared with `@fragment` attribute making it a fragment shader function.

- `  @fragment fn fs() -> @location(0) vec4f {`

This function takes no parameters and returns a `vec4f` at `location(0)`. This means it will write to the first render target. We'll make the first render target our canvas texture later.

- `    return vec4f(1, 0, 0, 1);`

The code returns `1, 0, 0, 1` which is red. Colors in WebGPU are usually specified as floating point values from `0.0` to `1.0` where the 4 values above correspond to red, green, blue, and alpha respectively.

When the GPU rasterizes the triangle (draws it with pixels), it will call the fragment shader to find out what color to make each pixel. In our case, we're just returning red.

One more thing to note is the `label`. Nearly every object you can create with WebGPU can take a `label`. Labels are entirely optional but it's considered *best practice* to label everything you make. The reason is that when you get an error, most WebGPU implementations will print an error message that includes the labels of the things related to the error.

In a normal app, you'd have 100s or 1000s of buffers, textures, shader modules, pipelines, etc… If you get an error like `"WGSL syntax error in shaderModule at line 10"`, if you have 100 shader modules, which one got the error? If you label the module then you'll get an error more like `"WGSL syntax error in shaderModule('our hardcoded red triangle shaders') at line 10` which is a way more useful error message and will save you a ton of time tracking down the issue.

Now that we've created a shader module, we next need to make a render pipeline:

```
const pipeline = device.createRenderPipeline({
  label: 'our hardcoded red triangle pipeline',
  layout: 'auto',
  vertex: {
    entryPoint: 'vs',
    module,
  },
  fragment: {
    entryPoint: 'fs',
    module,
    targets: [{ format: presentationFormat }],
  },
});
```

In this case, there isn't much to see. We set `layout` to `'auto'` which means to ask WebGPU to derive the layout of data from the shaders. We're not using any data though.

We then tell the render pipeline to use the `vs` function from our shader module for a vertex shader and the `fs` function for our fragment shader. Otherwise, we tell it the format of the first render target. "render target" means the texture we will render to. When we create a pipeline we have to specify the format for the texture(s) we'll use this pipeline to eventually render to.

Element 0 for the `targets` array corresponds to location 0 as we specified for the fragment shader's return value. Later, we'll set that target to be a texture for the canvas.

One shortcut, for each shader stage, `vertex` and `fragment`, if there is only one function of the corresponding type then we don't need to specify the `entryPoint`. WebGPU will use the sole function that matches the shader stage. So we can shorten the code above to:

```
const pipeline = device.createRenderPipeline({
  label: 'our hardcoded red triangle pipeline',
  layout: 'auto',
  vertex: {
    entryPoint: 'vs',
    module,
  },
  fragment: {
    entryPoint: 'fs',
    module,
    targets: [{ format: presentationFormat }],
  },
});
```

Next up we prepare a GPURenderPassDescriptor which describes which textures we want to draw to and how to use them.

```
const renderPassDescriptor = {
  label: 'our basic canvas renderPass',
  colorAttachments: [
    {
      // view: <- to be filled out when we render
      clearValue: [0.3, 0.3, 0.3, 1],
      loadOp: 'clear',
      storeOp: 'store',
    },
  ],
};
```

A GPURenderPassDescriptor has an array for `colorAttachments` which lists the textures we will render to and how to treat them. We'll wait to fill in which texture we actually want to render to. For now, we set up a clear value of semi-dark gray, and a `loadOp` and `storeOp`. `loadOp: 'clear'` specifies to clear the texture to the clear value before drawing. The other option is `'load'` which means load the existing contents of the texture into the GPU so we can draw over what's already there. `storeOp: 'store'` means store the result of what we draw. We could also pass `'discard'` which would throw away what we draw. We'll cover why we might want to do that in another article.

Now it's time to render.

```
function render() {
  // Get the current texture from the canvas context and
  // set it as the texture to render to.
  renderPassDescriptor.colorAttachments[0].view =
      context.getCurrentTexture().createView();

  // make a command encoder to start encoding commands
  const encoder = device.createCommandEncoder({ label: 'our encoder' });

  // make a render pass encoder to encode render specific commands
  const pass = encoder.beginRenderPass(renderPassDescriptor);
  pass.setPipeline(pipeline);
  pass.draw(3);  // call our vertex shader 3 times
  pass.end();

  const commandBuffer = encoder.finish();
  device.queue.submit([commandBuffer]);
}

render();
```

First, we call `context.getCurrentTexture()` to get a texture that will appear in the canvas. Calling `createView` gets a view into a specific part of a texture but with no parameters, it will return the default part which is what we want in this case. For now, our only `colorAttachment` is a texture view from our canvas which we get via the context we created at the start. Again, element 0 of the `colorAttachments` array corresponds to `@location(0)` as we specified for the return value of the fragment shader.

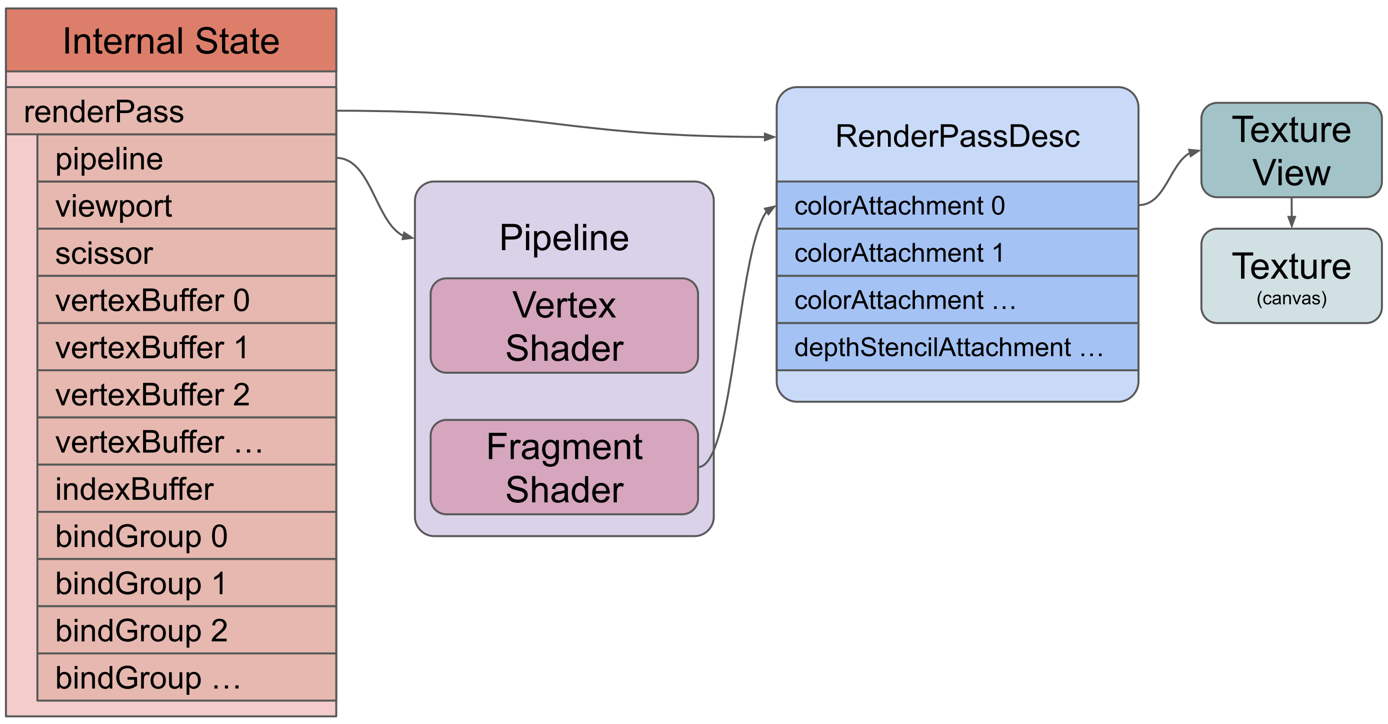
Next, we create a command encoder. A command encoder is used to create a command buffer. We use it to encode commands and then "submit" the command buffer it created to have the commands executed.

We then use the command encoder to create a render pass encoder by calling `beginRenderPass`. A render pass encoder is a specific encoder for creating commands related to rendering. We pass it our `renderPassDescriptor` to tell it which texture we want to render to.

We encode the command, `setPipeline`, to set our pipeline and then tell it to execute our vertex shader 3 times by calling `draw` with 3. By default, every 3 times our vertex shader is executed a triangle will be drawn by connecting the 3 values just returned from the vertex shader.

We end the render pass, and then finish the encoder. This gives us a command buffer that represents the steps we just specified. Finally, we submit the command buffer to be executed.

When the `draw` command is executed, this will be our state.



We've got no textures, no buffers, no bindGroups but we do have a pipeline, a vertex and fragment shader, and a render pass descriptor that tells our shader to render to the canvas texture.

The result.

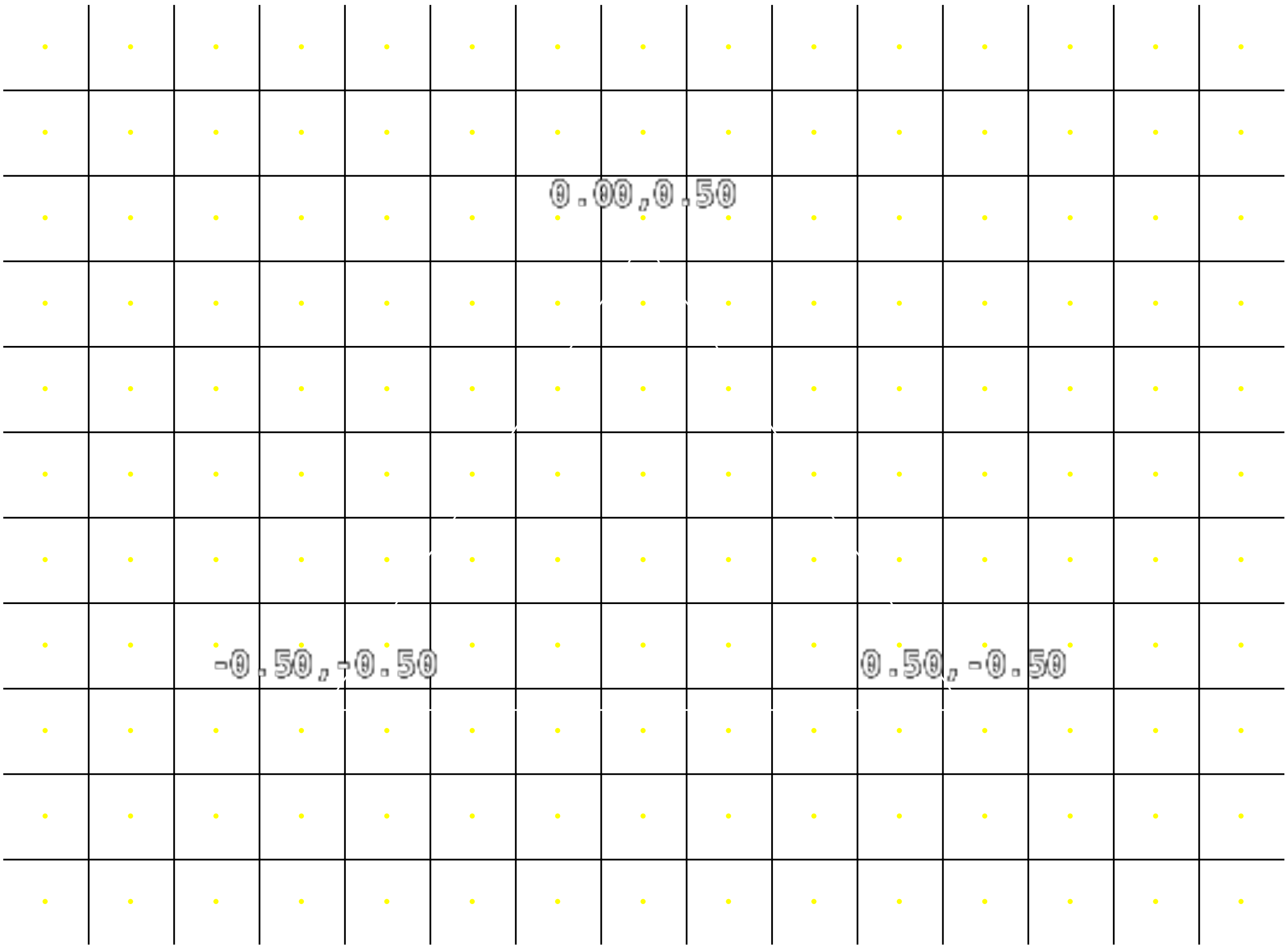
It's important to emphasize that all of these functions we called like `setPipeline`, and `draw` only add commands to a command buffer. They don't actually execute the commands. The commands are executed when we submit the command buffer to the device queue.

#

WebGPU takes every 3 vertices we return from our vertex shader and uses them to rasterize a triangle. It does this by determining which pixels' centers are inside the triangle. It then calls our fragment shader for each pixel to ask what color to make it.

Imagine the texture we are rendering to was 15x11 pixels. These are the pixels that would be drawn to

Need WebGPU



drag the vertices

So, now we’ve seen a very small working WebGPU example. It should be pretty obvious that hard coding a triangle inside a shader is not very flexible. We need ways to provide data and we’ll cover those in the following articles. The points to take away from the code above,

- WebGPU just runs shaders. It’s up to you to fill them with code to do useful things
- Shaders are specified in a shader module and then turned into a pipeline
- WebGPU can draw triangles
- WebGPU draws to textures (we happened to get a texture from the canvas)
- WebGPU works by encoding commands and then submitting them.

#

Run computations on the GPU

Let’s write a basic example for doing some computation on the GPU.

We start off with the same code to get a WebGPU device.

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

Then we create a shader module.

```
const module = device.createShaderModule({
  label: 'doubling compute module',
  code: `
    @group(0) @binding(0) var<storage, read_write> data: array<f32>;

    @compute @workgroup_size(1) fn computeSomething(
      @builtin(global_invocation_id) id: vec3u
    ) {
      let i = id.x;
      data[i] = data[i] * 2.0;
    }
  `,
});
```

First, we declare a variable called `data` of type `storage` that we want to be able to both read from and write to.

```
    @group(0) @binding(0) var<storage, read_write> data: array<f32>;
```

We declare its type as `array<f32>` which means an array of 32-bit floating point values. We tell it we're going to specify this array on binding location 0 (the `binding(0)`) in bindGroup 0 (the `@group(0)`).

Then we declare a function called `computeSomething` with the `@compute` attribute which makes it a compute shader.

```
    @compute @workgroup_size(1) fn computeSomething(
      @builtin(global_invocation_id) id: vec3u
    ) {
      ...
```

Compute shaders are required to declare a workgroup size which we will cover later. For now, we'll just set it to 1 with the attribute `@workgroup_size(1)`. We declare it to have one parameter `id` which uses a `vec3u`. A `vec3u` is three unsigned 32-bit integer values. Like our vertex shader above, this is the iteration number. It's different in that compute shader iteration numbers are 3 dimensional (have 3 values). We declare `id` to get its value from the built-in `global_invocation_id`.

You can *kind of* think of compute shaders as running like this. This is an over simplification but it will do for now.

```
// pseudo code
function dispatchWorkgroups(width, height, depth) {
  for (z = 0; z < depth; ++z) {
    for (y = 0; y < height; ++y) {
      for (x = 0; x < width; ++x) {
        const workgroup_id = {x, y, z};
        dispatchWorkgroup(workgroup_id)
      }
    }
  }
}

function dispatchWorkgroup(workgroup_id) {
  // from @workgroup_size in WGSL
  const workgroup_size = shaderCode.workgroup_size;
  const {x: width, y: height, z: depth} = workgroup_size;
  for (z = 0; z < depth; ++z) {
    for (y = 0; y < height; ++y) {
      for (x = 0; x < width; ++x) {
        const local_invocation_id = {x, y, z};
        const global_invocation_id =
            workgroup_id * workgroup_size + local_invocation_id;
        computeShader(global_invocation_id)
      }
    }
  }
}
```

Since we set `@workgroup_size(1)`, effectively the pseudo-code above becomes:

```
// pseudo code
function dispatchWorkgroups(width, height, depth) {
  for (z = 0; z < depth; ++z) {
    for (y = 0; y < height; ++y) {
      for (x = 0; x < width; ++x) {
        const workgroup_id = {x, y, z};
        dispatchWorkgroup(workgroup_id)
      }
    }
  }
}

function dispatchWorkgroup(workgroup_id) {
  const global_invocation_id = workgroup_id;
  computeShader(global_invocation_id)
}
```

Finally, we use the `x` property of `id` to index `data` and multiply each value by 2.

```
        let i = id.x;
        data[i] = data[i] * 2.0;
```

Above, `i` is just the first of the 3 iteration numbers.

Now that we've created the shader, we need to create a pipeline.

```
  const pipeline = device.createComputePipeline({
    label: 'doubling compute pipeline',
    layout: 'auto',
    compute: {
      module,
    },
  });
```

Here we just tell it we're using a `compute` stage from the shader `module` we created and since there is only one `@compute` entry point WebGPU knows we want to call it. `layout` is `'auto'` again, telling WebGPU to figure out the layout from the shaders. [5]

Next, we need some data.

```
  const input = new Float32Array([1, 3, 5]);
```

That data only exists in JavaScript. For WebGPU to use it, we need to make a buffer that exists on the GPU and copy the data to the buffer.

```
  // create a buffer on the GPU to hold our computation
  // input and output
  const workBuffer = device.createBuffer({
    label: 'work buffer',
    size: input.byteLength,
    usage: GPUBufferUsage.STORAGE | GPUBufferUsage.COPY_SRC | GPUBufferUsage.COPY_DST,
  });
  // Copy our input data to that buffer
  device.queue.writeBuffer(workBuffer, 0, input);
```

Above, we call `device.createBuffer` to create a buffer. `size` is the size in bytes. In this case, it will be 12 because the size in bytes of a `Float32Array` of 3 values is 12. If you're not familiar with `Float32Array` and typed arrays then see this article.

Every WebGPU buffer we create has to specify a `usage`. There are a bunch of flags we can pass for usage but not all of them can be used together. Here we say we want this buffer to be usable as `storage` by passing `GPUBufferUsage.STORAGE`. This makes it compatible with `var<storage,...>` from the shader. Further, we want to be able to copy data to this buffer so we include the `GPUBufferUsage.COPY_DST` flag. And finally, we want to be able to copy data from the buffer so we include `GPUBufferUsage.COPY_SRC`.

Note that you can not directly read the contents of a WebGPU buffer from JavaScript. Instead, you have to "map" it which is another way of requesting access to the buffer from WebGPU because the buffer might be in use and because it might only exist on the GPU.

WebGPU buffers that can be mapped in JavaScript can't be used for much else. In other words, we can not map the buffer we just created above and if we try to add the flag to make it mappable, we'll get an error that it is not compatible with usage `STORAGE`.

So, in order to see the result of our computation, we'll need another buffer. After running the computation, we'll copy the buffer above to this result buffer and set its flags so we can map it.

```
// create a buffer on the GPU to get a copy of the results
const resultBuffer = device.createBuffer({
  label: 'result buffer',
  size: input.byteLength,
  usage: GPUBufferUsage.MAP_READ | GPUBufferUsage.COPY_DST
});
```

`MAP_READ` means we want to be able to map this buffer for reading data.

In order to tell our shader about the buffer we want it to work on, we need to create a bindGroup.

```
// Setup a bindGroup to tell the shader which
// buffer to use for the computation
const bindGroup = device.createBindGroup({
  label: 'bindGroup for work buffer',
  layout: pipeline.getBindGroupLayout(0),
  entries: [
    { binding: 0, resource: { buffer: workBuffer } },
  ],
});
```

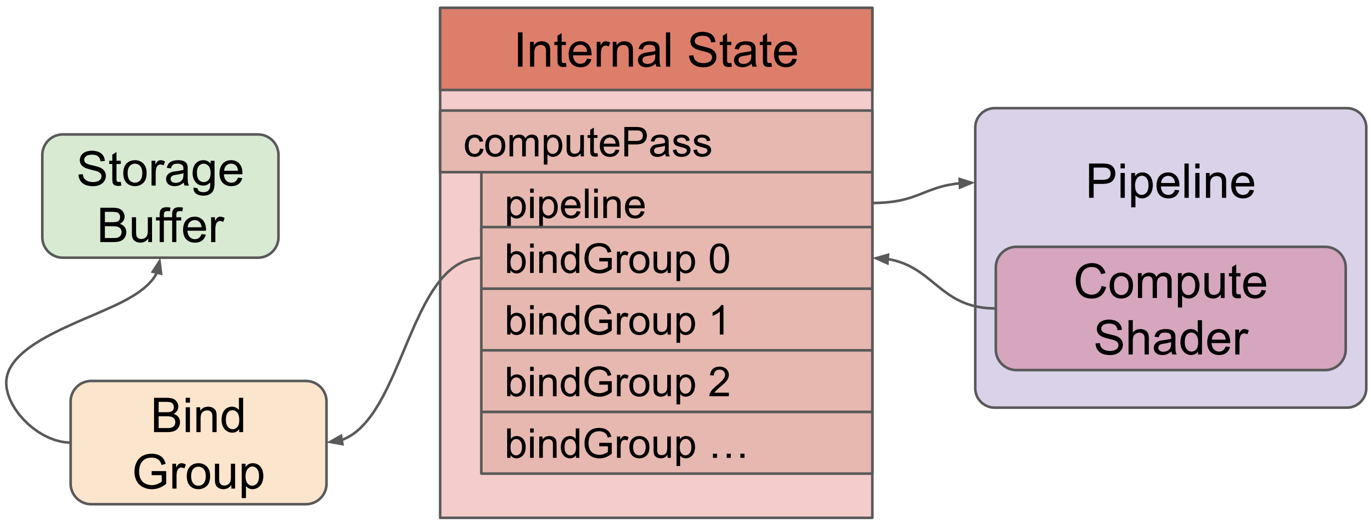
We get the layout for the bindGroup from the pipeline. Then we set up bindGroup entries. The 0 in `pipeline.getBindGroupLayout(0)` corresponds to the `@group(0)` in the shader. The `{binding: 0 ...` of the `entries` corresponds to the `@group(0) @binding(0)` in the shader.

Now we can start encoding commands.

```
// Encode commands to do the computation
const encoder = device.createCommandEncoder({
  label: 'doubling encoder',
});
const pass = encoder.beginComputePass({
  label: 'doubling compute pass',
});
pass.setPipeline(pipeline);
pass.setBindGroup(0, bindGroup);
pass.dispatchWorkgroups(input.length);
pass.end();
```

We create a command encoder. We start a compute pass. We set the pipeline, then we set the bindGroup. Here, the `0` in `pass.setBindGroup(0, bindGroup)` corresponds to `@group(0)` in the shader. We then call `dispatchWorkgroups` and in this case, we pass it `input.length` which is `3` telling WebGPU to run the compute shader 3 times. We then end the pass.

Here's what the situation will be when `dispatchWorkgroups` is executed.



After the computation is finished we ask WebGPU to copy from `workBuffer` to `resultBuffer`.

```
// Encode a command to copy the results to a mappable buffer.
encoder.copyBufferToBuffer(workBuffer, 0, resultBuffer, 0, resultBuffer.size);
```

Now we can `finish` the encoder to get a command buffer and then submit that command buffer.

```
// Finish encoding and submit the commands
const commandBuffer = encoder.finish();
device.queue.submit([commandBuffer]);
```

We then map the results buffer and get a copy of the data.

```
// Read the results
await resultBuffer.mapAsync(GPUMapMode.READ);
const result = new Float32Array(resultBuffer.getMappedRange());

console.log('input', input);
console.log('result', result);

resultBuffer.unmap();
```

To map the results buffer, we call `mapAsync` and have to `await` for it to finish. Once mapped, we can call `resultBuffer.getMappedRange()` which with no parameters will return an `ArrayBuffer` of the entire buffer. We put that in a `Float32Array` typed array view and then we can look at the values. One important detail, the `ArrayBuffer` returned by `getMappedRange` is only valid until we call `unmap`. After `unmap`, its length will be set to 0 and its data no longer accessible.

Running that we can see we got the result back, all the numbers have been doubled.

We’ll cover how to really use compute shaders in other articles. For now, you hopefully have gleaned some understanding of what WebGPU does. EVERYTHING ELSE IS UP TO YOU! Think of WebGPU as similar to other programming languages. It provides a few basic features and leaves the rest to your creativity.

What makes WebGPU programming special is these functions, vertex shaders, fragment shaders, and compute shaders, run on your GPU. A GPU could have over 10000 processors which means they can potentially do more than 10000 calculations in parallel which is likely 3 or more orders of magnitude than your CPU can do in parallel.

#

Simple Canvas Resizing

Before we move on, let’s go back to our triangle drawing example and add some basic support for resizing a canvas. Sizing a canvas is actually a topic that can have many subtleties so there is an entire article on it. For now though let’s just add some basic support.

First, we’ll add some CSS to make our canvas fill the page.

```
<style>
html, body {
  margin: 0;       /* remove the default margin          */
  height: 100%;    /* make the html,body fill the page   */
}
canvas {
  display: block;  /* make the canvas act like a block   */
  width: 100%;     /* make the canvas fill its container */
  height: 100%;
}
</style>
```

That CSS alone will make the canvas get displayed to cover the page but it won’t change the resolution of the canvas itself so you might notice, if you make the example below large, like if you click the full-screen button, you’ll see the edges of the triangle are blocky.

`<canvas>` tags, by default, have a resolution of 300x150 pixels. We’d like to adjust the resolution of the canvas to match the size it is displayed. One good way to do this is with a `ResizeObserver`. You create a `ResizeObserver` and give it a function to call whenever the elements you’ve asked it to observe change their size. You then tell it which elements to observe.

```
    ...
    render();

    const observer = new ResizeObserver(entries => {
      for (const entry of entries) {
        const canvas = entry.target;
        const width = entry.contentBoxSize[0].inlineSize;
        const height = entry.contentBoxSize[0].blockSize;
        canvas.width = Math.max(1, Math.min(width, device.limits.maxTextureDimension2D));
        canvas.height = Math.max(1, Math.min(height, device.limits.maxTextureDimension2D));
      }
      // re-render
      render();
    });
    observer.observe(canvas);
```

In the code above, we go over all the entries but there should only ever be one because we’re only observing our canvas. We need to limit the size of the canvas to the largest size our device supports otherwise WebGPU will start generating errors that we tried to make a texture that is too large. We also need to make sure it doesn’t go to zero or again we’ll get errors. See the longer article for details.

We call `render` to re-render the triangle at the new resolution. We removed the old call to `render` because it’s not needed. A `ResizeObserver` will always call its callback at least once to report the size of the elements when they started being observed.

The new size texture is created when we call `context.getCurrentTexture()` inside `render` so there’s nothing left to do.

> Note: The code above does not handle responding to zoom which may change the resolution of the canvas. It also doesn’t deal with higher resolutions for high-res displays. For those issues, see the article on resizing the canvas.

In the following articles, we'll cover various ways to pass data into shaders.

- inter-stage variables
- uniforms
- storage buffers
- vertex buffers
- textures
- constants

Then we'll cover the basics of WGSL.

This order is from the simplest to the most complex. Inter-stage variables require no external setup to explain. We can see how to use them using nothing but changes to the WGSL we used above. Uniforms are effectively global variables and as such are used in all 3 kinds of shaders (vertex, fragment, and compute). Going from uniform buffers to storage buffers is trivial as shown at the top of the article on storage buffers. Vertex buffers are only used in vertex shaders. They are more complex because they require describing the data layout to WebGPU. Textures are the most complex as they have tons of types and options.

I'm a little bit worried these articles will be boring at first. Feel free to jump around if you'd like. Just remember if you don't understand something you probably need to read or review these basics. Once we get the basics down, we'll start going over actual techniques.

One other thing. All of the example programs can be edited live in the webpage. Further, they can all easily be exported to jsfiddle and codepen and even stackoverflow. Just click "Export".

The code above gets a WebGPU device in a very terse way. A more verbose way would be something like

```
async function start() {
  if (!navigator.gpu) {
    fail('this browser does not support WebGPU');
    return;
  }

  const adapter = await navigator.gpu.requestAdapter();
  if (!adapter) {
    fail('this browser supports webgpu but it appears disabled');
    return;
  }

  const device = await adapter.requestDevice();
  device.lost.then((info) => {
    console.error(`WebGPU device was lost: ${info.message}`);

    // 'reason' will be 'destroyed' if we intentionally destroy the device.
    if (info.reason !== 'destroyed') {
      // try again
      start();
    }
  });

  main(device);
}
start();

function main(device) {
  ... do webgpu ...
}
```

`device.lost` is a promise that starts off unresolved. It will resolve if and when the device is lost. A device can be lost for many reasons. Maybe the user ran a really intensive app and it crashed their GPU. Maybe the user updated their drivers. Maybe the user has an external GPU and unplugged it. Maybe another page used a lot of GPU, your tab was in the background and the browser decided to free up some memory by losing the device for background tabs. The point to take away is that for any serious apps you probably want to handle losing the device.

Note that `requestDevice` always returns a device. It just might start lost. WebGPU is designed so that, for the most part, the device will appear to work, at least from an API level. Calls to create things and use them will appear to succeed but they won't actually function. It's up to you to take action when the `lost` promise resolves.

---

1. There are actually 5 modes.

   - `'point-list'`: for each position, draw a point
   - `'line-list'`: for each 2 positions, draw a line
   - `'line-strip'`: draw lines connecting the newest point to the previous point
   - `'triangle-list'`: for each 3 positions, draw a triangle (**default**)
   - `'triangle-strip'`: for each new position, draw a triangle from it and the last 2 positions

   ↩

2. Fragment shaders indirectly write data to textures. That data does not have to be colors. For example, it’s common to output the direction of the surface that pixel represents. ↩

3. Textures can also be 3D rectangles of pixels, cube maps (6 squares of pixels that form a cube), and a few other things but the most common textures are 2D rectangles of pixels. ↩

4. We can also use an index buffer to specify `vertex_index`. This is covered in the article on vertex-buffers. ↩

5. `layout: 'auto'` is convenient but it’s impossible to share bind groups across pipelines using `layout: 'auto'`. Most of the examples on this site never use a bind group with multiple pipelines. We’ll cover explicit layouts in another article. ↩