Introduction To WebGPU

Brandon Jones, Google

CIS 565, Fall 2022

These Slides - bit.ly/3D3eNCw

Who am I?

Brandon Jones

Google Chrome GPU team

Contributed to:

- WebGL
- WebXR
- WebGPU

What we’ll be covering

- Why WebGPU is being developed
- Compare WebGPU and WebGL
- Why WebGPU is more efficient
- Deeper dive into core WebGPU concepts
- Features to aid in debugging
- General tips

WebGL

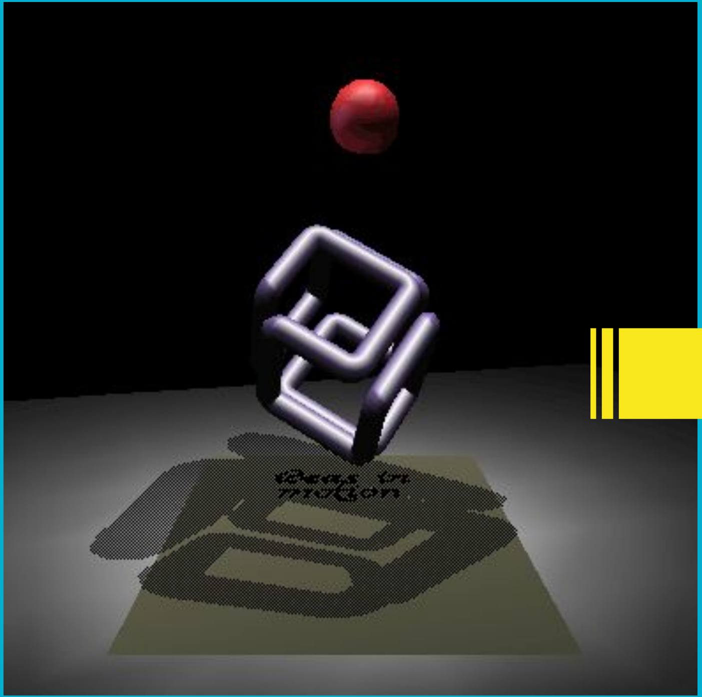
Ubiquitous realtime 3D graphics

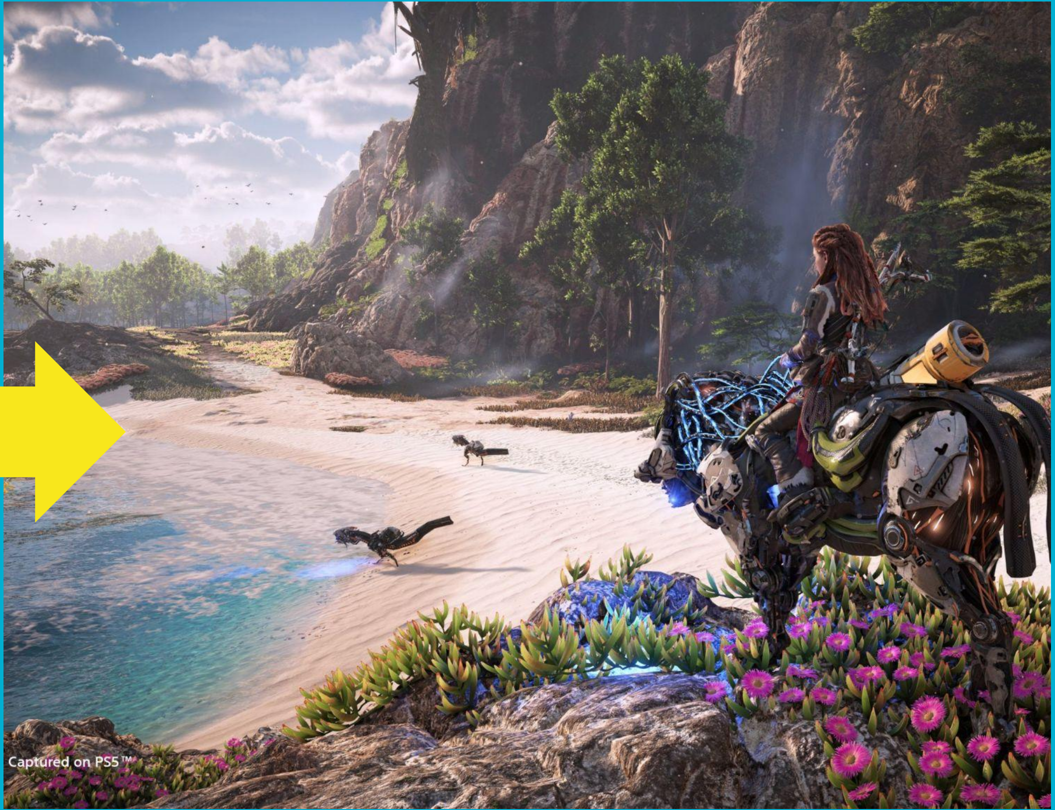
Supported by every major browser
on every major platform

WebGL has been the web’s GPU API for 11 years

- Managed by the Khronos group
- WebGL 2.0 (2017)
  - Based on OpenGL ES 3.0 (2012)
- WebGL 1.0 (2011)
  - Based on OpenGL ES 2.0 (2007)
- Mobile-oriented APIs
- Based on desktop OpenGL (1993!)

Realtime graphics have evolved since 1993...





Native graphics APIs have evolved as well

- Vulkan (2016)
  - Created by the Khronos group
  - Cross-platform
  - “Successor” to OpenGL
- D3D12 (2015)
  - Created by Microsoft
  - Only for Windows/XBox
  - Successor to D3D10/11
- Metal (2014)
  - Created by Apple
  - Only for Apple devices

WebGL can’t keep up

- OpenGL and OpenGL ES aren’t receiving major new features
- OpenGL was never designed with the web in mind
- Core API patterns are detrimental to performance and usability

WebGPU

A modern GPU API for the web

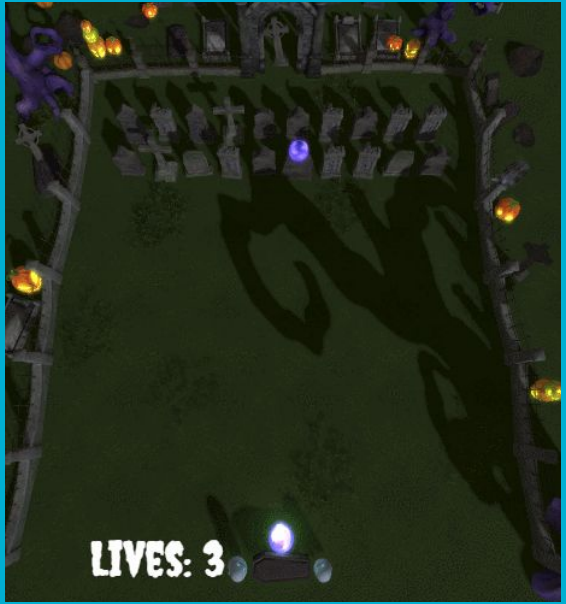
- Exposes capabilities of Metal, D3D12, and Vulkan
- Web-native API
- Modern Features
- Built with extensibility in mind

New WebGPU features

- Compute Shaders
- Indirect drawing
- Render Bundles
- External textures (for more efficient video)
- More flexible Canvas integration
- Improved debugging features
- No global state!

Demos!

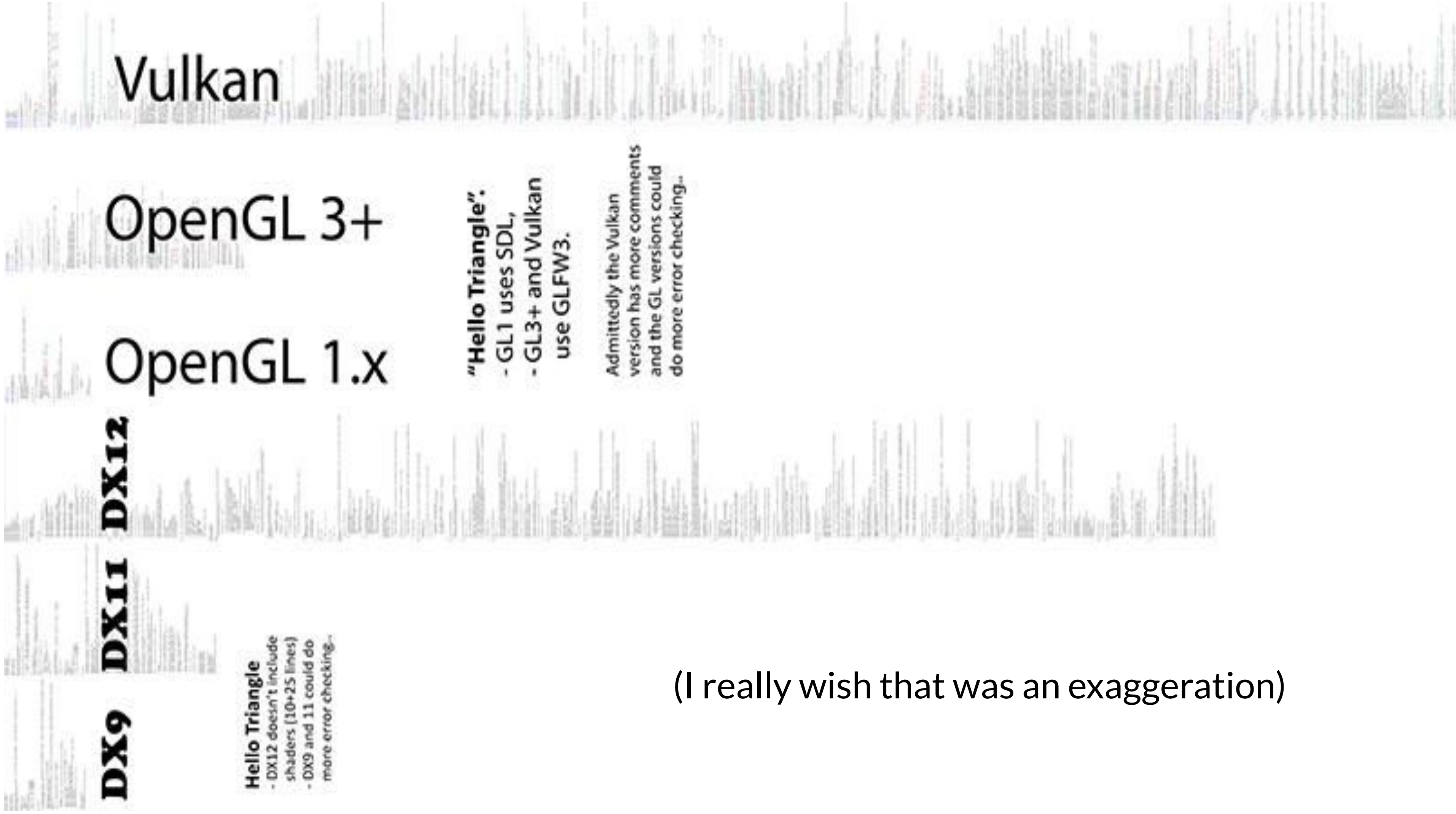
Compute Metaballs (Source)



Spookyball (Source)

Wait… why not WebVulkan?

Vulkan can be a bit intimidating.



(I really wish that was an exaggeration)

[illegible]

Web APIs

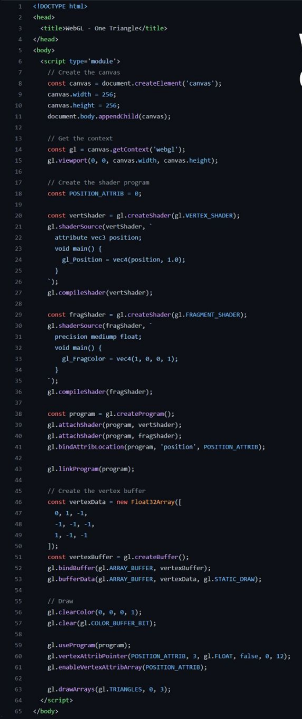
- Code you write on one device should have a reasonable expectation of running on every device. (Setting aside performance differences.)
- Incompatibility should always be opt-in!
- Web APIs are safe. Period.
    - No undefined behavior
    - No access to memory not owned by the page
    - No access to sensitive data (camera, microphone, etc) without user's consent.

Native APIs

- Different APIs for different devices
- Expose platform differences out of necessity to enable peak performance.
- Happily allow data races, undefined behavior, etc.
- Model requires more trust in the app developer/distributor.

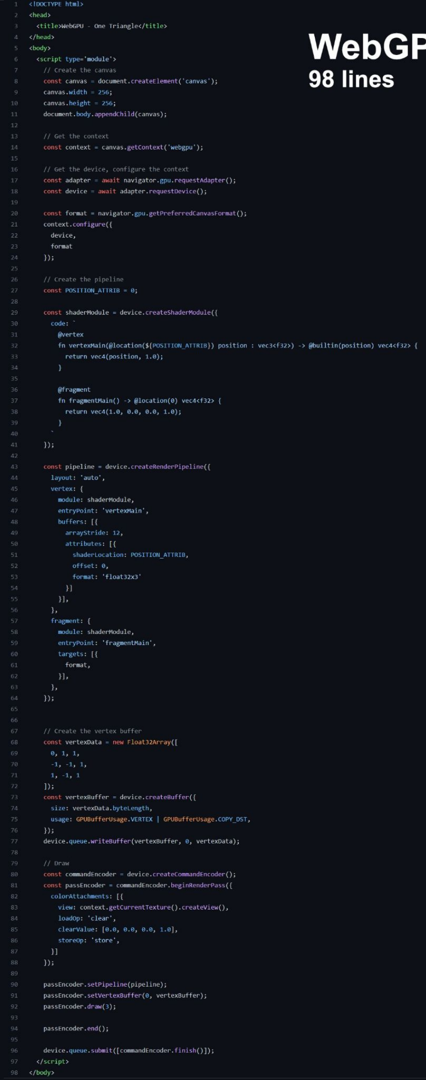
WebGPU is built for the web

- Expose only the common features
- Smooth over differences in limits/requirements
- Remove complexity where it doesn’t make sense for the web
- Use patterns that web developers are familiar with
- Leverages underlying modern native APIs *and* more web-friendly patterns to improve performance over WebGL.



WebGL
65 lines

```html
<!DOCTYPE html>
<head>
  <title>WebGL - One Triangle</title>
</head>
<body>
  <script type='module'>
    // Create the canvas
    const canvas = document.createElement('canvas');
    canvas.width = 256;
    canvas.height = 256;
    document.body.appendChild(canvas);

    // Get the context
    const gl = canvas.getContext('webgl');
    gl.viewport(0, 0, canvas.width, canvas.height);

    // Create the shader program
    const POSITION_ATTRIB = 0;

    const vertShader = gl.createShader(gl.VERTEX_SHADER);
    gl.shaderSource(vertShader, `
      attribute vec3 position;
      void main() {
        gl_Position = vec4(position, 1.0);
      }
    `);
    gl.compileShader(vertShader);

    const fragShader = gl.createShader(gl.FRAGMENT_SHADER);
    gl.shaderSource(fragShader, `
      precision mediump float;
      void main() {
        gl_FragColor = vec4(1, 0, 0, 1);
      }
    `);
    gl.compileShader(fragShader);

    const program = gl.createProgram();
    gl.attachShader(program, vertShader);
    gl.attachShader(program, fragShader);
    gl.bindAttribLocation(program, 'position', POSITION_ATTRIB);

    gl.linkProgram(program);

    // Create the vertex buffer
    const vertexData = new Float32Array([
      0, 1, -1,
      -1, -1, -1,
      1, -1, -1
    ]);
    const vertexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, vertexData, gl.STATIC_DRAW);

    // Draw
    gl.clearColor(0, 0, 0, 1);
    gl.clear(gl.COLOR_BUFFER_BIT);

    gl.useProgram(program);
    gl.vertexAttribPointer(POSITION_ATTRIB, 3, gl.FLOAT, false, 0, 12);
    gl.enableVertexAttribArray(POSITION_ATTRIB);

    gl.drawArrays(gl.TRIANGLES, 0, 3);
  </script>
</body>
```



WebGPU
98 lines

```html
<!DOCTYPE html>
<head>
  <title>WebGPU - One Triangle</title>
</head>
<body>
  <script type='module'>
    // Create the canvas
    const canvas = document.createElement('canvas');
    canvas.width = 256;
    canvas.height = 256;
    document.body.appendChild(canvas);

    // Get the context
    const context = canvas.getContext('webgpu');

    // Get the device, configure the context
    const adapter = await navigator.gpu.requestAdapter();
    const device = await adapter.requestDevice();

    const format = navigator.gpu.getPreferredCanvasFormat();
    context.configure({
      device,
      format
    });

    // Create the pipeline
    const POSITION_ATTRIB = 0;

    const shaderModule = device.createShaderModule({
      code: `
        @vertex
        fn vertexMain(@location(${POSITION_ATTRIB}) position : vec3<f32>) -> @builtin(position) vec4<f32> {
          return vec4(position, 1.0);
        }

        @fragment
        fn fragmentMain() -> @location(0) vec4<f32> {
          return vec4(1.0, 0.0, 0.0, 1.0);
        }
      `
    });

    const pipeline = device.createRenderPipeline({
      layout: 'auto',
      vertex: {
        module: shaderModule,
        entryPoint: 'vertexMain',
        buffers: [{
          arrayStride: 12,
          attributes: [{
            shaderLocation: POSITION_ATTRIB,
            offset: 0,
            format: 'float32x3'
          }]
        }],
      },
      fragment: {
        module: shaderModule,
        entryPoint: 'fragmentMain',
        targets: [{
          format,
        }],
      },
    });


    // Create the vertex buffer
    const vertexData = new Float32Array([
      0, 1, 1,
      -1, -1, 1,
      1, -1, 1
    ]);
    const vertexBuffer = device.createBuffer({
      size: vertexData.byteLength,
      usage: GPUBufferUsage.VERTEX | GPUBufferUsage.COPY_DST,
    });
    device.queue.writeBuffer(vertexBuffer, 0, vertexData);

    // Draw
    const commandEncoder = device.createCommandEncoder();
    const passEncoder = commandEncoder.beginRenderPass({
      colorAttachments: [{
        view: context.getCurrentTexture().createView(),
        loadOp: 'clear',
        clearValue: [0.0, 0.0, 0.0, 1.0],
        storeOp: 'store',
      }]
    });

    passEncoder.setPipeline(pipeline);
    passEncoder.setVertexBuffer(0, vertexBuffer);
    passEncoder.draw(3);

    passEncoder.end();

    device.queue.submit([commandEncoder.finish()]);
  </script>
</body>
```

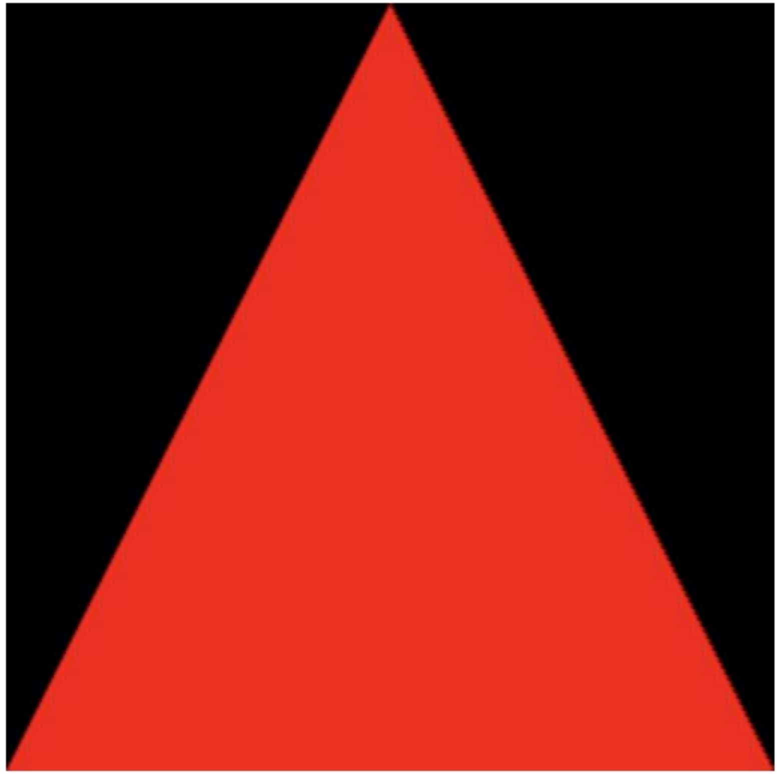
This allows us to keep the complexity of WebGPU to a minimum, so it's not too much more code than WebGL.

And it's *significantly* less complex than the native APIs!

Code to draw 1 triangle

WebGL vs WebGPU

“Hello Triangle” comparison



WebGL vs WebGPU - Initialization

```
// WebGL
const gl = canvas.getContext('webgl');
```

```
// WebGPU
const adapter = await navigator.gpu.requestAdapter();
const device = await adapter.requestDevice();

const context = canvas.getContext('webgpu');
context.configure({
   device,
   format: 'bgra8unorm',
});
```

WebGL vs WebGPU - Buffers

```
const vertexData = new Float32Array([
   0, 1, -1,
   -1, -1, -1,
   1, -1, -1
]);

const vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.bufferData(gl.ARRAY_BUFFER, vertexData,
    gl.STATIC_DRAW);
```

```
const vertexData = new Float32Array([
   0, 1, 1,
   -1, -1, 1,
   1, -1, 1
]);

const vertexBuffer = device.createBuffer({
   size: vertexData.byteLength,
   usage: GPUBufferUsage.VERTEX | GPUBufferUsage.COPY_DST,
});
device.queue.writeBuffer(vertexBuffer, 0, vertexData);
```

WebGL vs WebGPU - Shaders

```
const vertShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertShader, `
 attribute vec3 position;
 void main() {
   gl_Position = vec4(position, 1);
 }`);
gl.compileShader(vertShader);

const fragShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragShader, `
 precision mediump float;
 void main() {
   gl_FragColor = vec4(1, 0, 0, 1);
 }`);
gl.compileShader(fragShader);
```

```
const shaderModule = device.createShaderModule({
 code: `
   @vertex
   fn vertexMain(@location(0) pos : vec3<f32>) ->
       @builtin(position) vec4<f32> {
     return vec4(pos, 1.0);
   }
   @fragment
   fn fragmentMain() -> @location(0) vec4<f32> {
     return vec4(1.0, 0.0, 0.0, 1.0);
   }`
});
```

WebGL vs WebGPU - Programs/Pipelines

```
const program = gl.createProgram();
gl.attachShader(program, vertShader);
gl.attachShader(program, fragShader);
gl.bindAttribLocation(program, 'position', 0);

gl.linkProgram(program);

// Lots of error checking here normally…
```

```
const pipeline = device.createRenderPipeline({
 layout: 'auto',
 vertex: {
   module: shaderModule, entryPoint: 'vertexMain',
   buffers: [{
     arrayStride: 12,
     attributes: [{
       shaderLocation: 0, offset: 0, format: 'float32x3'
     }]
   }],
 },
 fragment: {
   module: shaderModule, entryPoint: 'fragmentMain',
   targets: [{ format, }],
 },
});
```

WebGL vs WebGPU - Drawing

```
gl.clearColor(0, 0, 0, 1);
gl.clear(gl.COLOR_BUFFER_BIT);

gl.useProgram(program);

gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.vertexAttribPointer(0, 3, gl.FLOAT, false, 12, 0);
gl.enableVertexAttribArray(0);

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

```
const commandEncoder = device.createCommandEncoder();
const passEncoder = commandEncoder.beginRenderPass({
 colorAttachments: [{
   view: context.getCurrentTexture().createView(),
   loadOp: 'clear',
   clearValue: [0.0, 0.0, 0.0, 1.0],
   storeOp: 'store',
 }]
});

passEncoder.setPipeline(pipeline);
passEncoder.setVertexBuffer(0, vertexBuffer);
passEncoder.draw(3);
passEncoder.end();

device.queue.submit([commandEncoder.finish()]);
```

But why is WebGPU more efficient?

Explained with sandwiches

Imagine WebGL is a sandwich shop.

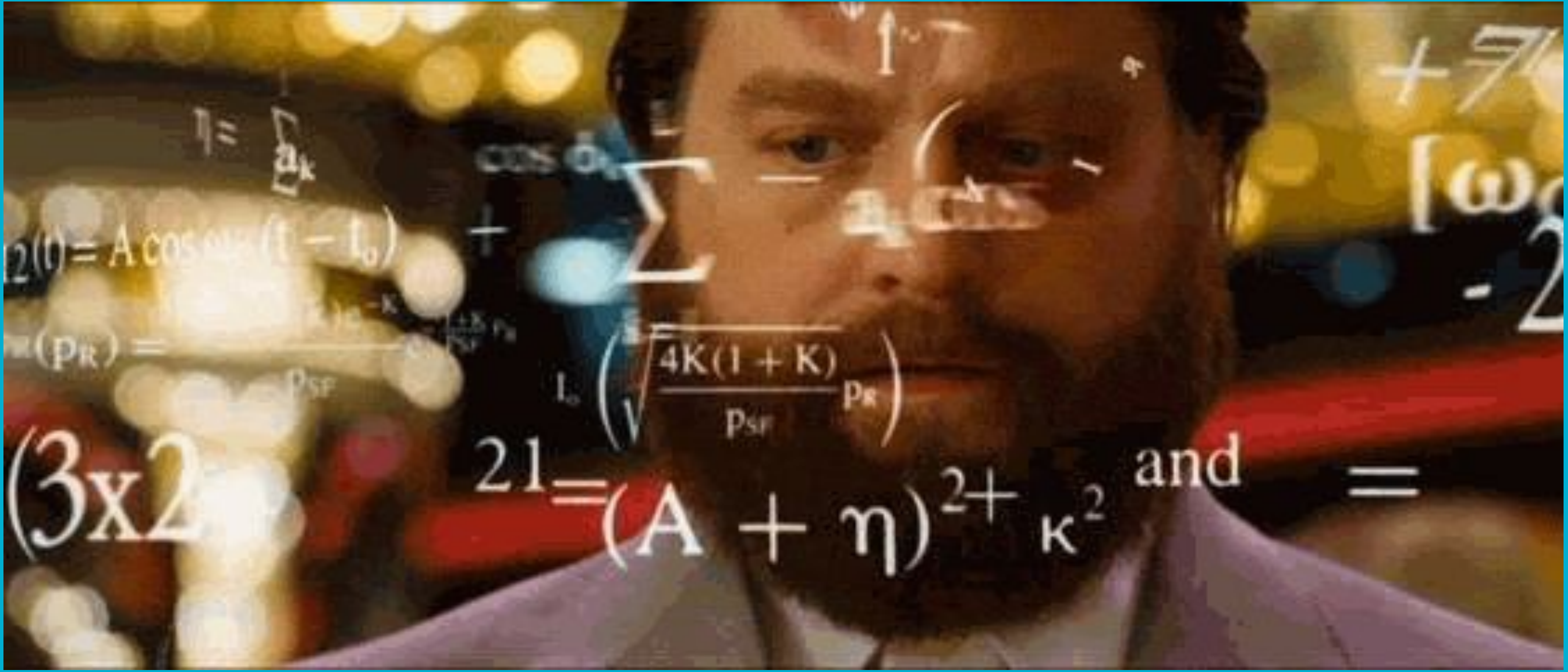
Every day you walk in and order the same sandwich from a waiter, who passes your order off to the kitchen.

But the situation in the kitchen is complicated:

The chef speaks a different language than you ordered in.

The ingredients are laid out in a confusing way.

Certain sandwiches have been deemed health code violations.



So to make things run smoothly the waiter translates your order to the chef’s native language, re-orders the ingredients, and checks for health code compliance before handing it off to the kitchen. Every single time.

Then one day the WebGPU deli opens up next door!

They have a weird rule: All orders must be submitted to the front desk in advance!

Once your order has been approved they inform you your sandwich will now be known as "The #53"

That feels like too much overhead for a single sandwich, right?

But the next time you come in you call out “53, please!” and your sandwich is done in record time!

But guess what? WebGPU deli and the WebGL sandwich shop share the **SAME** kitchen!

So why is the new one faster?

It’s because when you submitted your order to the front desk they took the time to translate it, re-organize the ingredients for optimal retrieval, and check for regulation compliance. *Once.*

It’s more work up-front for you, sure, and that can feel annoying or intimidating at first, but now you’re far less reliant on the ~~driver~~ waiter being clever in order to speed things up.

Draw Time vs. Setup Time

```
// WebGL, During Draw
gl.vertexAttribPointer(0, 3, gl.FLOAT, false, 12, 0);
gl.enableVertexAttribArray(0);

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

```
// WebGPU, During Setup
const pipeline = device.createRenderPipeline({
 vertex: {
   buffers: [{
     arrayStride: 12,
     attributes: [{
       shaderLocation: 0,
       offset: 0,
       format: 'float32x3'
     }]
   }],
 },
 primitive: { topology: 'triangle-list' },
 // … Some properties omitted for brevity
});
```

Draw Time vs. Setup Time

```
// WebGL, During Draw
gl.vertexAttribPointer(0, 3, gl.FLOAT, false, 12, 0);
gl.enableVertexAttribArray(0);

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

```
// WebGPU, During Setup
const pipeline = device.createRenderPipeline({
 vertex: {
   buffers: [{
     arrayStride: 12,
     attributes: [{
       shaderLocation: 0,
       offset: 0,
       format: 'float32x3'
     }]
   }],
 },
 primitive: { topology: 'triangle-list' },
 // … Some properties omitted for brevity
});
```

Draw Time vs. Setup Time

```
// WebGL, During Draw
gl.vertexAttribPointer(0, 3, gl.FLOAT, false, 12, 0);
gl.enableVertexAttribArray(0);

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

```
// WebGPU, During Setup
const pipeline = device.createRenderPipeline({
 vertex: {
   buffers: [{
     arrayStride: 12,
     attributes: [{
       shaderLocation: 0,
       offset: 0,
       format: 'float32x3'
     }]
   }],
 },
 primitive: { topology: 'triangle-list' },
 // … Some properties omitted for brevity
});
```

Draw Time vs. Setup Time

```
// WebGL, During Draw
gl.vertexAttribPointer(0, 3, gl.FLOAT, false, 12, 0);
gl.enableVertexAttribArray(0);

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

```
// WebGPU, During Setup
const pipeline = device.createRenderPipeline({
 vertex: {
   buffers: [{
     arrayStride: 12,
     attributes: [{
       shaderLocation: 0,
       offset: 0,
       format: 'float32x3'
     }]
   }],
 },
 primitive: { topology: 'triangle-list' },
 // … Some properties omitted for brevity
});
```

Draw Time vs. Setup Time

```
// WebGL, During Draw
gl.vertexAttribPointer(0, 3, gl.FLOAT, false, 12, 0);
gl.enableVertexAttribArray(0);

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

```
// WebGPU, During Setup
const pipeline = device.createRenderPipeline({
 vertex: {
   buffers: [{
     arrayStride: 12,
     attributes: [{
       shaderLocation: 0,
       offset: 0,
       format: 'float32x3'
     }]
   }],
 },
 primitive: { topology: 'triangle-list' },
 // … Some properties omitted for brevity
});
```

Draw Time vs. Setup Time

```
// WebGL, During Draw
gl.vertexAttribPointer(0, 3, gl.FLOAT, false, 12, 0);
gl.enableVertexAttribArray(0);

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

```
// WebGPU, During Setup
const pipeline = device.createRenderPipeline({
 vertex: {
   buffers: [{
     arrayStride: 12,
     attributes: [{
       shaderLocation: 0,
       offset: 0,
       format: 'float32x3'
     }]
   }],
 },
 primitive: { topology: 'triangle-list' },
 // … Some properties omitted for brevity
});
```

```
// RenderPipelineDescriptor properties

layout: pipelineLayout,

vertex: {
   module: gpuShaderModule,
   entryPoint: "vertexMain",
   buffers: [{
     arrayStride: 16,
     stepMode: "vertex",
     attributes: [{
       format: "float32x4",
       offset: 0,
       shaderLocation: 0,
     }],
   }],
   constants: {
     constantName: 1.0,
   },
 },
```

```
fragment: {
   module: gpuShaderModule,
   entryPoint: "fragmentMain",
   targets: [{
     format: "bgra8unorm",
     blend: {
       color: {
         operation: "add",
         srcFactor: "one",
         dstFactor: "zero",
       },
       alpha: {
         operation: "add",
         srcFactor: "one",
         dstFactor: "zero",
       },
     },
     writeMask: GPUColorWrite.ALL,
   }],
   constants: {
     constantName: 1.0,
   },
 }
```

```
depthStencil: {
  format: "depth24plus-stencil8",
  depthWriteEnabled: true,
  depthCompare: "less",
  stencilFront: {
    compare: "always",
    failOp: "keep",
    depthFailOp: "keep",
    passOp: "keep",
  },
  stencilBack:{
    compare: "always",
    failOp: "keep",
    depthFailOp: "keep",
    passOp: "keep",
  },
  stencilReadMask: 0xFFFFFFFF,
  stencilWriteMask: 0xFFFFFFFF,
  depthBias: 0,
  depthBiasSlopeScale: 0,
  depthBiasClamp: 0,
},
```

```
primitive: {
  topology: "triangle-strip",
  stripIndexFormat: "uint32",
  frontFace: "ccw",
  cullMode: "none",
},

multisample: {
  count: 4,
  mask: 0xFFFFFFFF,
  alphaToCoverageEnabled: true,
}
```

WebGPU core concepts

GPUAdapter

- `await navigator.gpu.requestAdapter(options);`
- Adapters are (more or less) the GPUs available to the device.
- Can be software! (SwiftShader)
- Only returns one at a time, but can request an adapter that meets certain criteria.
  - `{ powerPreference: 'low-power' }`
- Gives basic description of the GPU
  - `{vendor: “nvidia”, architecture: “turing” }`
- Lists the features and limits available to the GPUDevice.

GPUDevice

- `await adapter.requestDevice(options);`
- Primary interface for the API
- Creates resources like Textures, Buffers, Pipelines, etc.
- Has a GPUQueue for executing commands
- Roughly equivalent to a WebGLRenderingContext

Features

- Roughly equivalent to WebGL’s extensions
- Typically things not supported on all implementations/systems
    - `"texture-compression-bc"`
    - `"timestamp-query"`
- Adapter lists which ones are available.
- Must be specified when the requesting a Device or they won’t be active.

Limits

- Numeric limits of GPU capabilities
    - `maxTextureDimension2D`
    - `maxBindGroups`
    - `maxVertexBuffers`
- Each has a baseline that all WebGPU implementations must support.
- Adapter reports the actual system limits.
- Devices will only have access to the default limits unless otherwise specified when requesting the Device.

GPUCanvasContext

- Allows WebGPU to render to the canvas
- Must be configured after creation to associate it with a device.
- Multiple canvases can be configured to use the same device
- Can be reconfigured as needed
- Provides a texture to render into

```
// During initialization
const context = canvas.getContext('webgpu');
context.configure({
   device,
   format: 'bgra8unorm',
});

// During frame loop
const renderTarget = context.getCurrentTexture();
```

Resource types

- **GPUBuffer**
  - Specifies size and usage.
  - Uniforms, Vertices, Indices, General data
- **GPUTexture**
  - Specifies 1D/2D/3D, size, mips, samples, format, usage.
- **GPUTextureView**
  - Subset of a texture for sampling, render targets.
  - Specifies usage as cube map, array texture, etc.
- **GPUSampler**
  - Specifies Texture filtering/wrapping behavior.
- All have an immutable “shape” after creation, but the contents can be changed.

```
const buffer = device.createBuffer({
   size: 2048, // Bytes
   usage: GPUBufferUsage.VERTEX | GPUBufferUsage.COPY_DST,
});

const texture = device.createTexture({
  size: { width: 64, height: 64 },
  mipLevelCount: 4,
  format: 'rgba8unorm',
  usage: GPUTextureUsage.TEXTURE_BINDING,
});
const textureView = texture.createView({
  baseMipLevel: 1,
  mipLevelCount: 1,
});

const sampler = device.createSampler({
  magFilter: "nearest",
  minFilter: "linear",
  mipmapFilter: "linear",
  addressModeU: "repeat",
  addressModeV: "clamp-to-edge",
});
```

Shading language - WGSL

```
// Vertex Shader

struct Camera {
  projection : mat4x4<f32>,
  view : mat4x4<f32>,
};
@group(0) @binding(0) var<uniform> camera : Camera;

struct VertexInput {
  @location(0) position : vec3<f32>,
  @location(1) texCoord : vec2<f32>,
};

struct VertexOutput {
  @builtin(position) position : vec4<f32>,
  @location(0) texCoord : vec2<f32>,
};

@stage(vertex)
fn vertexMain(input : VertexInput) -> VertexOutput {
  var output : VertexOutput;
  output.texCoord = input.texCoord;
  output.position = camera.projection * camera.view * vec4(input.position, 1.0);
  return output;
}
```

```
// Compute Shader

struct GlobalState {
  deltaT : f32,
}
@group(0) @binding(0) var<uniform> globalState : GlobalState;

struct Particle {
  pos : vec2<f32>,
  vel : vec2<f32>,
}
@group(0) @binding(1) var<storage, read> particlesIn : array<Particle>;
@group(0) @binding(2) var<storage, read_write> particlesOut : array<Particle>;

@compute @workgroup_size(64)
fn main(@builtin(global_invocation_id) GlobalInvocationID : vec3<u32>) {
  let index : u32 = GlobalInvocationID.x;

  let vPos = particlesIn[index].pos;
  let vVel = particlesIn[index].vel;

  particlesOut[index].pos = vPos + (vVel * globalState.deltaT);
  particlesOut[index].vel = vVel + (vec3(0.0, 0.0, -9.8) * globalState.deltaT);
}
```

Pipelines

- Comes in **GPURenderPipeline** and **GPUComputePipeline** flavors
- Links WGSL shaders (via **GPUShaderModules**)
- **GPURenderPipeline** sets majority of relevant state for rendering
- Immutable after creation (Noticing a pattern?)

```
const pipeline = device.createComputePipeline({
  layout: pipelineLayout,
  compute: {
    module: shaderModule,
    entryPoint: 'computeMain',
  }
});
```

Queue

- Device has a default **GPUQueue**, which is the only one that you can use for now. (Might expand it in future API versions).
- Used to submit commands to the GPU.
- Also has handy helper functions for writing to buffers and textures.
- These are the easiest ways to set the contents of these resources, and you're encouraged to use them!

```
device.queue.writeBuffer(buffer, 0, typedArray);
device.queue.writeTexture({ texture: dstTexture },
                          typedArray,
                          { bytesPerRow: 256 },
                          { width: 64, height: 64 });
```

Recording GPU commands

- Create a **GPUCommandEncoder** from the device
- Perform copies between buffers/textures
- Begin render or compute passes
- Creates a **GPUCommandBuffer** when finished.
- Command buffers don't do anything until submitted to the queue.
- Cannot reuse a command buffer after it's been submitted.

```
const commandEncoder = device.createCommandEncoder();
commandEncoder.copyBufferToBuffer(bufferA, 0,
                                  bufferB, 0, 256);

const passEncoder = commandEncoder.beginComputePass();
passEncoder.setPipeline(pipeline);
passEncoder.setBindGroup(0, bindGroup);
passEncoder.dispatchWorkgroups(128);
passEncoder.end();

const commandBuffer = commandEncoder.finish();
device.queue.submit([commandBuffer]);
```

Passes

- A **render pass** can use GPURenderPipelines, bind vertex/index buffers, issue draw calls, and writes to a texture or series of textures.
- A **compute pass** can use GPUComputePipelines, and issues dispatch calls.
- Can’t record other types of commands while the a pass is active.
- Both pass types can set bind groups.

```
const renderPass = commandEncoder.beginRenderPass({
  colorAttachments: [{
    view: context.getCurrentTexture().createView(),
    loadOp: 'clear',
    clearValue: [0.0, 0.0, 0.0, 1.0],
    storeOp: 'store',
  }]
});
renderPass.setPipeline(renderPipeline);
renderPass.setBindGroup(0, bindGroup);
renderPass.setVertexBuffer(0, vertexBuffer);
renderPass.draw(3);
renderPass.end();

const computePass = commandEncoder.beginComputePass();
computePass.setPipeline(computePipeline);
computePass.setBindGroup(0, bindGroup);
computePass.dispatchWorkgroups(128);
computePass.end();
```

Exposing resources to shaders

BindGroups overview

Setting a uniform in WebGL

```
// JavaScript
const lightPosUni = gl.getUniformLocation(program,
  'lightPostion');

gl.uniform3f(lightPosUni, 0, 1, 0);
```

```
// GLSL Shader
uniform vec3 lightPosition;
```

Setting a uniform is WebGPU is… not quite so simple.

Exposing resources to shaders

- Values from your program get into shaders one of two ways: A binding, or a vertex attribute.
- The bindings need to be declared in two places. The first, is in the shader itself.
- Bindings are each associated with a group, and a binding number of your choice.

```
// WGSL code
struct GlobalState {
  deltaT : f32,
}
@group(0) @binding(0)
var<uniform> state : GlobalState;

struct Particle {
  pos : vec2<f32>,
  vel : vec2<f32>,
}
@group(1) @binding(1)
var<storage, read> particlesIn : array<Particle>;
@group(1) @binding(2)
var<storage, read_write> particlesOut : array<Particle>;

@compute @workgroup_size(64)
fn main(@builtin(global_invocation_id) GlobalInvocationID : vec3<u32>) {
  let index : u32 = GlobalInvocationID.x;

  let vPos = particlesIn[index].pos;
  let vVel = particlesIn[index].vel;

  particlesOut[index].pos = vPos + (vVel * state.deltaT);
  particlesOut[index].vel = vVel + (vec3(0.0, 0.0, -9.8) * state.deltaT);
}
```

Exposing resources to shaders

- Values from your program get into shaders one of two ways: A binding, or a vertex attribute.
- The bindings need to be declared in two places. The first, is in the shader itself.
- Bindings are each associated with a group, and a binding number of your choice.

```
// WGSL code
struct GlobalState {
  deltaT : f32,
}
@group(0) @binding(0)
var<uniform> state : GlobalState;

struct Particle {
  pos : vec2<f32>,
  vel : vec2<f32>,
}
@group(1) @binding(1)
var<storage, read> particlesIn : array<Particle>;
@group(1) @binding(2)
var<storage, read_write> particlesOut : array<Particle>;

@compute @workgroup_size(64)
fn main(@builtin(global_invocation_id) GlobalInvocationID : vec3<u32>) {
  let index : u32 = GlobalInvocationID.x;

  let vPos = particlesIn[index].pos;
  let vVel = particlesIn[index].vel;

  particlesOut[index].pos = vPos + (vVel * state.deltaT);
  particlesOut[index].vel = vVel + (vec3(0.0, 0.0, -9.8) * state.deltaT);
}
```

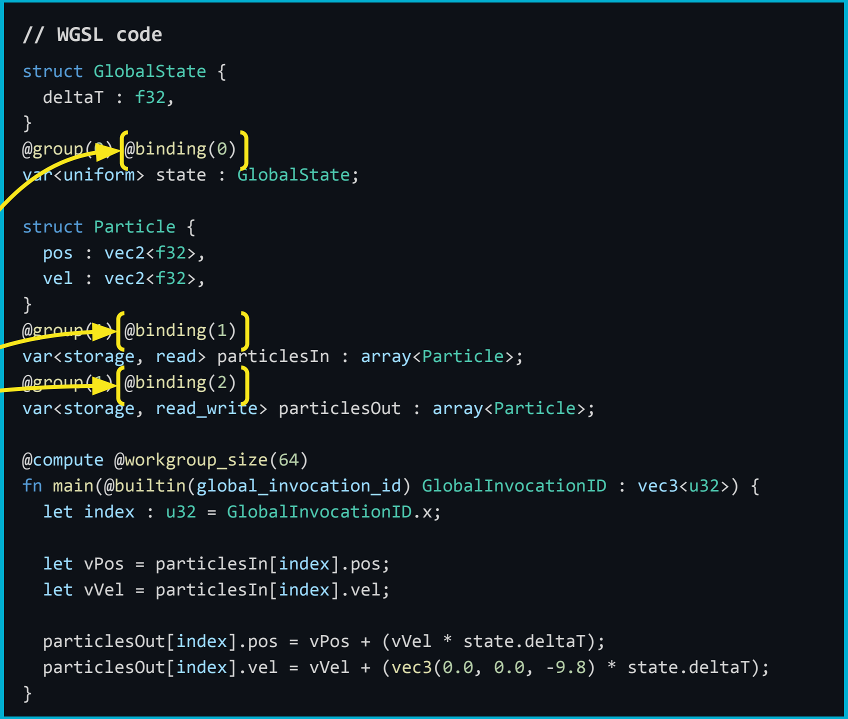
Exposing resources to shaders

- Values from your program get into shaders one of two ways: A binding, or a vertex attribute.
- The bindings need to be declared in two places. The first, is in the shader itself.
- Bindings are each associated with a group, and a binding number of your choice.

```
// WGSL code
struct GlobalState {
  deltaT : f32,
}
@group(0) @binding(0)
var<uniform> state : GlobalState;

struct Particle {
  pos : vec2<f32>,
  vel : vec2<f32>,
}
@group(1) @binding(1)
var<storage, read> particlesIn : array<Particle>;
@group(1) @binding(2)
var<storage, read_write> particlesOut : array<Particle>;

@compute @workgroup_size(64)
fn main(@builtin(global_invocation_id) GlobalInvocationID : vec3<u32>) {
  let index : u32 = GlobalInvocationID.x;

  let vPos = particlesIn[index].pos;
  let vVel = particlesIn[index].vel;

  particlesOut[index].pos = vPos + (vVel * state.deltaT);
  particlesOut[index].vel = vVel + (vec3(0.0, 0.0, -9.8) * state.deltaT);
}
```

Exposing resources to shaders

- Values from your program get into shaders one of two ways: A binding, or a vertex attribute.
- The bindings need to be declared in two places. The first, is in the shader itself.
- Bindings are each associated with a group, and a binding number of your choice.



```
// WGSL code
struct GlobalState {
  deltaT : f32,
}
@group(0) @binding(0)
var<uniform> state : GlobalState;

struct Particle {
  pos : vec2<f32>,
  vel : vec2<f32>,
}
@group(1) @binding(1)
var<storage, read> particlesIn : array<Particle>;
@group(1) @binding(2)
var<storage, read_write> particlesOut : array<Particle>;

@compute @workgroup_size(64)
fn main(@builtin(global_invocation_id) GlobalInvocationID : vec3<u32>) {
  let index : u32 = GlobalInvocationID.x;

  let vPos = particlesIn[index].pos;
  let vVel = particlesIn[index].vel;

  particlesOut[index].pos = vPos + (vVel * state.deltaT);
  particlesOut[index].vel = vVel + (vec3(0.0, 0.0, -9.8) * state.deltaT);
}
```

Exposing resources to shaders

- Next, in JavaScript, you'll create a corresponding **GPUBindGroupLayout** for each group.
- It needs to match the resource definitions in the shader.
- Also indicates what shader types the resource can be exposed to. (Vertex, Fragment, Compute)

```
const bindGroupLayout = device.createBindGroupLayout({
  entries: [{
    binding: 0,
    visibility: GPUShaderStage.COMPUTE,
    buffer: {
       type: 'read-only-storage'
    }
  }, {
    binding: 1,
    visibility: GPUShaderStage.COMPUTE,
    buffer: {
       type: 'storage'
    },
  }],
});
```

Exposing resources to shaders

```js
const bindGroupLayout = device.createBindGroupLayout({
  entries: [{
    binding: 0,
    visibility: GPUShaderStage.COMPUTE,
    buffer: {
       type: 'read-only-storage'
    }
  }, {
    binding: 1,
    visibility: GPUShaderStage.COMPUTE,
    buffer: {
       type: 'storage'
    },
  }],
});
```

```wgsl
// WGSL code
struct GlobalState {
  deltaT : f32,
}
@group(0) @binding(0)
var<uniform> state : GlobalState;

struct Particle {
  pos : vec2<f32>,
  vel : vec2<f32>,
}
@group(1) @binding(1)
var<storage, read> particlesIn : array<Particle>;
@group(1) @binding(2)
var<storage, read_write> particlesOut : array<Particle>;

@compute @workgroup_size(64)
fn main(@builtin(global_invocation_id) GlobalInvocationID : vec3<u32>) {
  let index : u32 = GlobalInvocationID.x;

  let vPos = particlesIn[index].pos;
  let vVel = particlesIn[index].vel;

  particlesOut[index].pos = vPos + (vVel * state.deltaT);
  particlesOut[index].vel = vVel + (vec3(0.0, 0.0, -9.8) * state.deltaT);
}
```

Exposing resources to shaders

```
const bindGroupLayout = device.createBindGroupLayout({
  entries: [{
    binding: 0,
    visibility: GPUShaderStage.COMPUTE,
    buffer: {
       type: 'read-only-storage'
    }
  }, {
    binding: 1,
    visibility: GPUShaderStage.COMPUTE,
    buffer: {
       type: 'storage'
    },
  }],
});
```

```
// WGSL code
struct GlobalState {
  deltaT : f32,
}
@group(0) @binding(0)
var<uniform> state : GlobalState;

struct Particle {
  pos : vec2<f32>,
  vel : vec2<f32>,
}
@group(1) @binding(1)
var<storage, read> particlesIn : array<Particle>;
@group(1) @binding(2)
var<storage, read_write> particlesOut : array<Particle>;

@compute @workgroup_size(64)
fn main(@builtin(global_invocation_id) GlobalInvocationID : vec3<u32>) {
  let index : u32 = GlobalInvocationID.x;

  let vPos = particlesIn[index].pos;
  let vVel = particlesIn[index].vel;

  particlesOut[index].pos = vPos + (vVel * state.deltaT);
  particlesOut[index].vel = vVel + (vec3(0.0, 0.0, -9.8) * state.deltaT);
}
```

Exposing resources to shaders

- The bind group layouts for all the groups are passed to the pipeline when you create it via a **GPUPipelineLayout**
- Each entry in the pipeline layout corresponds to one of the groups in the shaders
- You *can* have the pipeline generate the layouts for you, but that limits their usefulness

```
const pipelineLayout = device.createPipelineLayout({
  bindGroupLayouts: [
    bindGroupLayoutA, // @group(0)
    bindGroupLayoutB, // @group(1)
  ]
});

const pipeline = device.createComputePipeline({
  layout: pipelineLayout,
  compute: {
    module: shaderModule,
    entryPoint: 'computeMain',
  }
});
```

Exposing resources to shaders

- After that, you create a new **GPUBindGroup** with the bind group layout that actually points to the resources you want to use
- Resources can be part of multiple bind groups, and even in the same bind group multiple times (as long as the usage doesn't conflict)

```
// WGSL code
struct GlobalState {
  deltaT : f32,
}
@group(0) @binding(0)
var<uniform> state : GlobalState;

struct Particle {
  pos : vec2<f32>,
  vel : vec2<f32>,
}
@group(1) @binding(1)
var<storage, read> particlesIn : array<Particle>;
@group(1) @binding(2)
var<storage, read_write> particlesOut : array<Particle>;

@compute @workgroup_size(64)
fn main(@builtin(global_invocation_id) GlobalInvocationID : vec3<u32>) {
  let index : u32 = GlobalInvocationID.x;

  let vPos = particlesIn[index].pos;
  let vVel = particlesIn[index].vel;

  particlesOut[index].pos = vPos + (vVel * state.deltaT);
  particlesOut[index].vel = vVel + (vec3(0.0, 0.0, -9.8) * state.deltaT);
}
```

Exposing resources to shaders

```
const bindGroup = device.createBindGroup({
  layout: bindGroupLayout,
  entries: [{
    binding: 0,
    resource: { buffer: inputBuffer },
  }, {
    binding: 1,
    resource: {
      buffer: outputBuffer,
      offset: 256,
      size: 1024
    },
  }],
});
```

```
// WGSL code
struct GlobalState {
  deltaT : f32,
}
@group(0) @binding(0)
var<uniform> state : GlobalState;

struct Particle {
  pos : vec2<f32>,
  vel : vec2<f32>,
}
@group(1) @binding(1)
var<storage, read> particlesIn : array<Particle>;
@group(1) @binding(2)
var<storage, read_write> particlesOut : array<Particle>;

@compute @workgroup_size(64)
fn main(@builtin(global_invocation_id) GlobalInvocationID : vec3<u32>) {
  let index : u32 = GlobalInvocationID.x;

  let vPos = particlesIn[index].pos;
  let vVel = particlesIn[index].vel;

  particlesOut[index].pos = vPos + (vVel * state.deltaT);
  particlesOut[index].vel = vVel + (vec3(0.0, 0.0, -9.8) * state.deltaT);
}
```

Exposing resources to shaders

```
const bindGroup = device.createBindGroup({
  layout: bindGroupLayout,
  entries: [{
    binding: 0,
    resource: { buffer: inputBuffer },
  }, {
    binding: 1,
    resource: {
      buffer: outputBuffer,
      offset: 256,
      size: 1024
    },
  }],
});
```

```
// WGSL code
struct GlobalState {
  deltaT : f32,
}
@group(0) @binding(0)
var<uniform> state : GlobalState;

struct Particle {
  pos : vec2<f32>,
  vel : vec2<f32>,
}
@group(1) @binding(1)
var<storage, read> particlesIn : array<Particle>;
@group(1) @binding(2)
var<storage, read_write> particlesOut : array<Particle>;

@compute @workgroup_size(64)
fn main(@builtin(global_invocation_id) GlobalInvocationID : vec3<u32>) {
  let index : u32 = GlobalInvocationID.x;

  let vPos = particlesIn[index].pos;
  let vVel = particlesIn[index].vel;

  particlesOut[index].pos = vPos + (vVel * state.deltaT);
  particlesOut[index].vel = vVel + (vec3(0.0, 0.0, -9.8) * state.deltaT);
}
```

Putting it all together

- Once you've jumped through all those hoops, drawing or dispatching is easy
- setBindGroup needs to be called for every @group in the pipeline
- Bind groups don't snapshot the contents of their resources, so updates the underlying buffers/textures are visible to the shader.
- Updating uniforms in WebGPU typically means creating a bind group with the uniform buffer once, then using **writeBuffer()** or similar to update the buffer every frame.

```
const uniformArray = new Float32Array([performance.now()]);
device.queue.writeBuffer(uniformBuffer, 0, uniformArray);

const commandEncoder = device.createCommandEncoder();
const passEncoder = commandEncoder.beginComputePass();

passEncoder.setPipeline(pipeline);
passEncoder.setBindGroup(0, uniformBindGroup);
passEncoder.setBindGroup(1, particlesBindGroup);
passEncoder.dispatchWorkgroups(128);

passEncoder.end();
device.queue.submit([commandEncoder.finish()]);
```

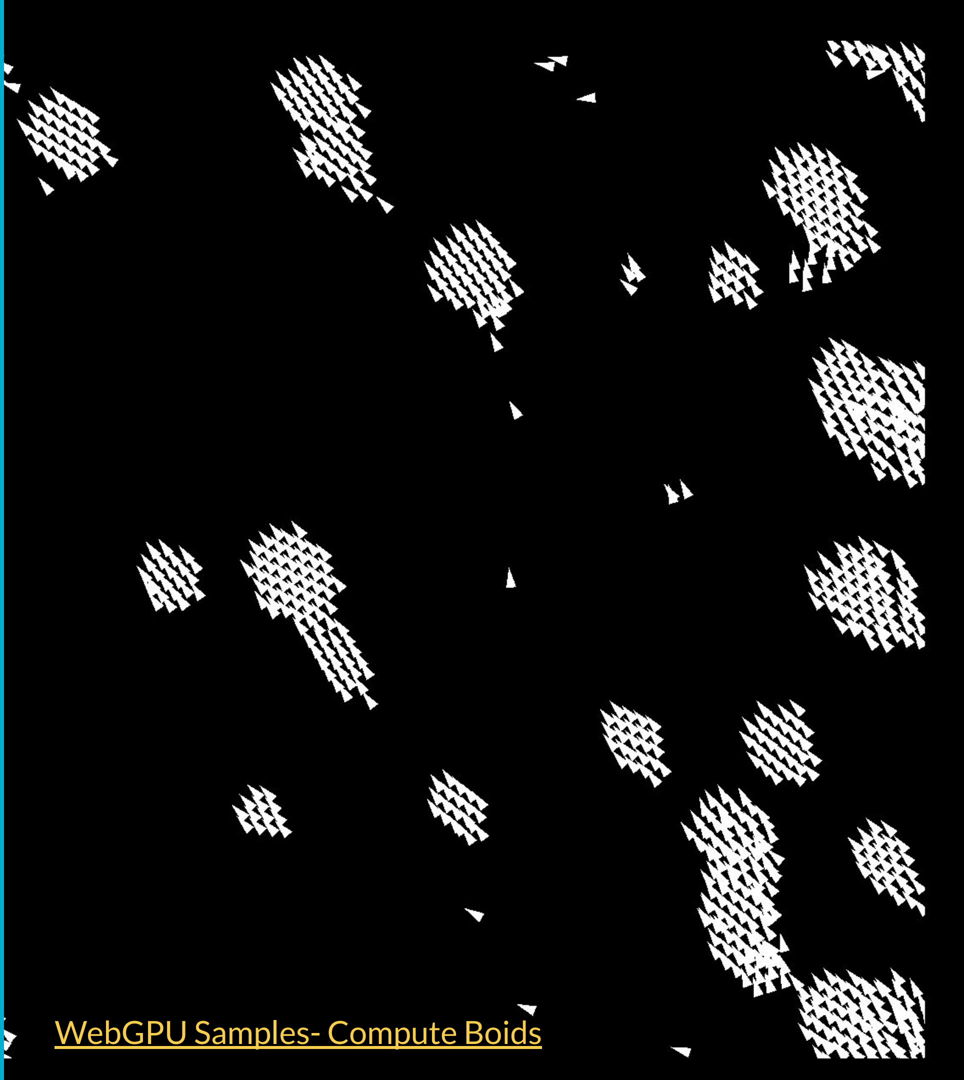
Important efficiency note!

- Bind groups can be shared between pipelines that use the same bind group layout
- Thus, a common and effective pattern is to place any values that will be common to your entire scene (like camera uniforms) in one bind group that you bind once at the top of the render pass and re-use for every pipeline
- Try to group bind group values based on how frequently you need to change them

```
renderPass.setBindGroup(0, frameBindGroup);

for (const material of scene.materials) {
  renderPass.setPipeline(material.pipeline);
  renderPass.setBindGroup(1, material.bindGroup);

  for (const mesh of meshesByMaterial(material)) {
    for (let i = 0; i < mesh.vertexBuffers; ++i) {
      renderPass.setVertexBuffer(i, mesh.vertexBuffers[i]);
    }

    renderPass.setBindGroup(2, mesh.bindGroup);
    renderPass.draw(mesh.drawCount);
  }
}
```

How WebGPU Helps developers

Debugging APIs



WebGPU Samples- Compute Boids

Find the bug, WebGL edition:

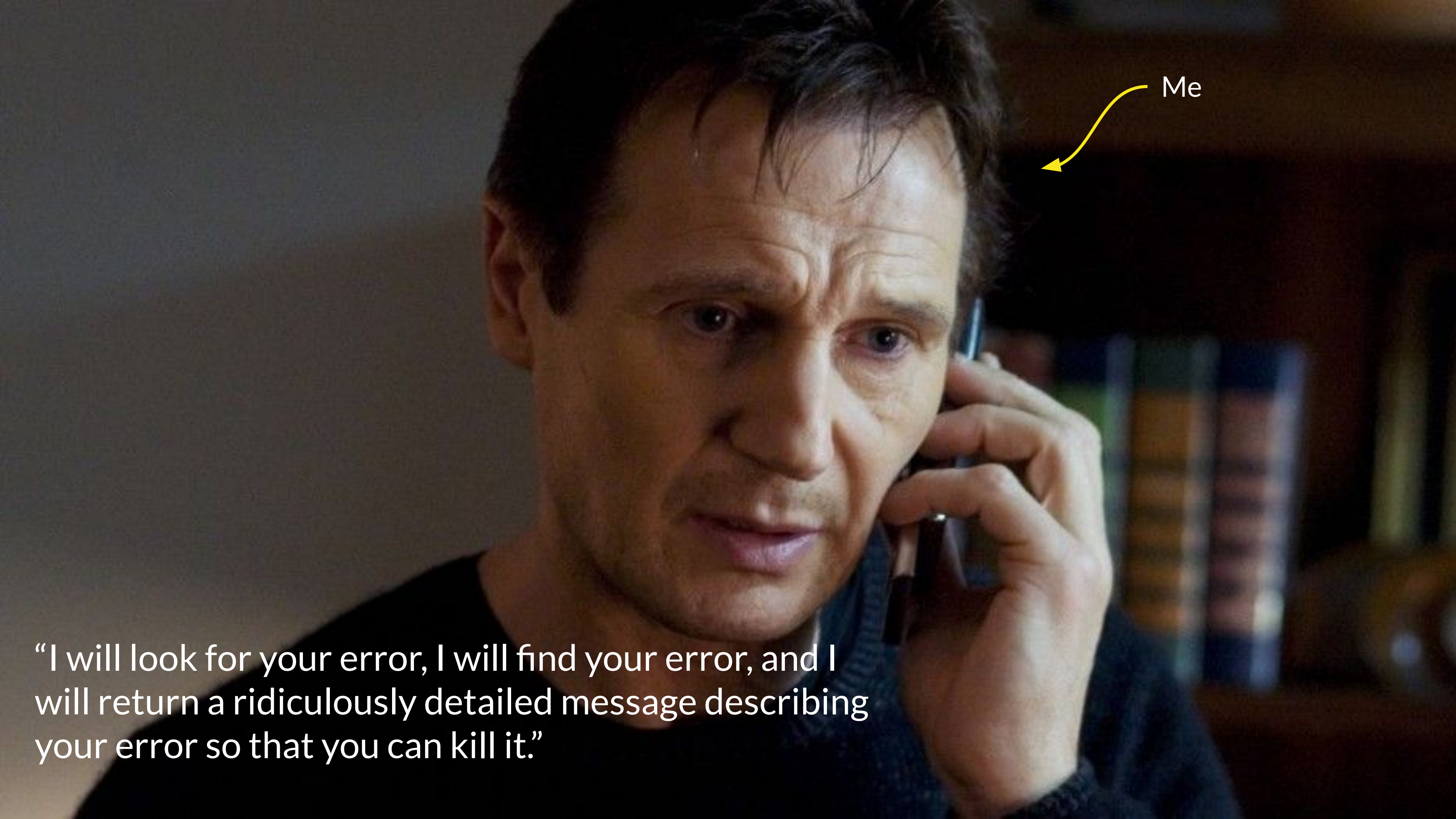
This variant of the previous “Hello Triangle” code doesn’t draw anything. Why?

```
// Create the vertex buffer
const vertexData = new Float32Array([
  0, 1, -1,
  -1, -1, -1,
  1, -1, -1
]);
const vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, vertexBuffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, vertexData, gl.STATIC_DRAW);

// Draw
gl.clearColor(0, 0, 0, 1);
gl.clear(gl.COLOR_BUFFER_BIT);

gl.useProgram(program);
gl.vertexAttribPointer(POSITION_ATTRIB, 3, gl.FLOAT, false, 12, 0);
gl.enableVertexAttribArray(POSITION_ATTRIB);

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

Hey, WebGL? What went wrong here?

WebGL
WebGL
GL_INVALID_OPERATION

....Thanks WebGL.



“I will look for your error, I will find your error, and I will return a ridiculously detailed message describing your error so that you can kill it.”

Find the bug, WebGPU edition:

This code has roughly the same bug as the WebGL version.

```
// Create the vertex buffer
const vertexData = new Float32Array([
  0, 1, 1,
  -1, -1, 1,
  1, -1,
]);
const vertexBuffer = device.createBuffer({
  size: vertexData.byteLength,
  usage: GPUBufferUsage.INDEX | GPUBufferUsage.COPY_DST,
});
device.queue.writeBuffer(vertexBuffer, 0, vertexData);

// Draw
const commandEncoder = device.createCommandEncoder();
const passEncoder = commandEncoder.beginRenderPass(/*...*/);
passEncoder.setPipeline(pipeline);
passEncoder.setVertexBuffer(0, vertexBuffer);
passEncoder.draw(3);
passEncoder.end();

device.queue.submit([commandEncoder.finish()]);
```

Chrome Console:

⚠️[Buffer] usage (BufferUsage::(CopyDst|Index)) doesn't include BufferUsage::Vertex.
 - While encoding [RenderPassEncoder].SetVertexBuffer(0, [Buffer], 0).

This code has roughly the same bug as the WebGL version.

```
]);
const vertexBuffer = device.createBuffer({
  size: vertexData.byteLength,
  usage: GPUBufferUsage.INDEX | GPUBufferUsage.COPY_DST,
});
device.queue.writeBuffer(vertexBuffer, 0, vertexData);

// Draw
const commandEncoder = device.createCommandEncoder();
const passEncoder = commandEncoder.beginRenderPass(/*...*/);
passEncoder.setPipeline(pipeline);
passEncoder.setVertexBuffer(0, vertexBuffer);
passEncoder.draw(3);
passEncoder.end();

device.queue.submit([commandEncoder.finish()]);
```

Chrome Console:

```
⚠️[Buffer] usage (BufferUsage::(CopyDst|Index)) doesn't include BufferUsage::Vertex.
    - While encoding [RenderPassEncoder].SetVertexBuffer(0, [Buffer], 0).
```

This code has roughly the same bug as the WebGL version.

```
]);
const vertexBuffer = device.createBuffer({
  size: vertexData.byteLength,
  usage: GPUBufferUsage.INDEX | GPUBufferUsage.COPY_DST,
});
device.queue.writeBuffer(vertexBuffer, 0, vertexData);

// Draw
const commandEncoder = device.createCommandEncoder();
const passEncoder = commandEncoder.beginRenderPass(/*...*/);
passEncoder.setPipeline(pipeline);
passEncoder.setVertexBuffer(0, vertexBuffer);
passEncoder.draw(3);
passEncoder.end();

device.queue.submit([commandEncoder.finish()]);
```

Chrome Console:

⚠️[Buffer] usage (BufferUsage::(CopyDst|Index)) doesn't include BufferUsage::Vertex.
    - While encoding [RenderPassEncoder].SetVertexBuffer(0, [Buffer], 0).

This code has roughly the same bug as the WebGL version.

```
]);
const vertexBuffer = device.createBuffer({
  size: vertexData.byteLength,
  usage: GPUBufferUsage.INDEX | GPUBufferUsage.COPY_DST,
});
device.queue.writeBuffer(vertexBuffer, 0, vertexData);

// Draw
const commandEncoder = device.createCommandEncoder();
const passEncoder = commandEncoder.beginRenderPass(/*...*/);
passEncoder.setPipeline(pipeline);
passEncoder.setVertexBuffer(0, vertexBuffer);
passEncoder.draw(3);
passEncoder.end();

device.queue.submit([commandEncoder.finish()]);
```

But what about in more complex scenarios?

Failures in larger programs are far harder to identify, even with a detailed message like that.

The more context that the error can give you, the better.

Introducing:

Labels

Your new favorite debugging feature!

Label usage:

```
const vertexBuffer = device.createBuffer({
  label: 'Player vertices',
  size: vertexData.byteLength,
  usage: GPUBufferUsage.INDEX | GPUBufferUsage.COPY_DST,
});

const passEncoder = commandEncoder.beginRenderPass({
  label: 'Primary render pass',
  colorAttachments: [{
    view: context.getCurrentTexture().createView(),
    loadOp: 'clear',
    clearValue: [0.0, 0.0, 0.0, 1.0],
    storeOp: 'store',
  }]
});
```

Chrome Console:

```
⚠️[Buffer "Player vertices"] usage (BufferUsage::(CopyDst|Index))
doesn't include BufferUsage::Vertex.
    - While encoding [RenderPassEncoder "Primary render pass"]
        .SetVertexBuffer(0, [Buffer "Player vertices"], 0).
```

Also introducing:

Debug Groups

Your *other* new favorite debugging feature!

Debug group usage:

```
const commandEncoder = device.createCommandEncoder();
commandEncoder.pushDebugGroup('Main Render Loop');

  commandEncoder.pushDebugGroup('Render Scene');
    renderGameScene(commandEncoder);
  commandEncoder.popDebugGroup();

  commandEncoder.pushDebugGroup('Render UI');
    renderGameUI(commandEncoder);
  commandEncoder.popDebugGroup();

commandEncoder.popDebugGroup();
device.queue.submit([commandEncoder.finish()]);
```

Chrome Console:

```
⚠️[Buffer "Player vertices"] usage (BufferUsage::(CopyDst|Index))
doesn't include BufferUsage::Vertex.
   - While encoding [RenderPassEncoder "Primary"]
       .SetVertexBuffer(0, [Buffer "Player vertices"], 0).

   Debug group stack:
   > "Render Player"
   > "Render Scene"
   > "Main Render Loop"
```

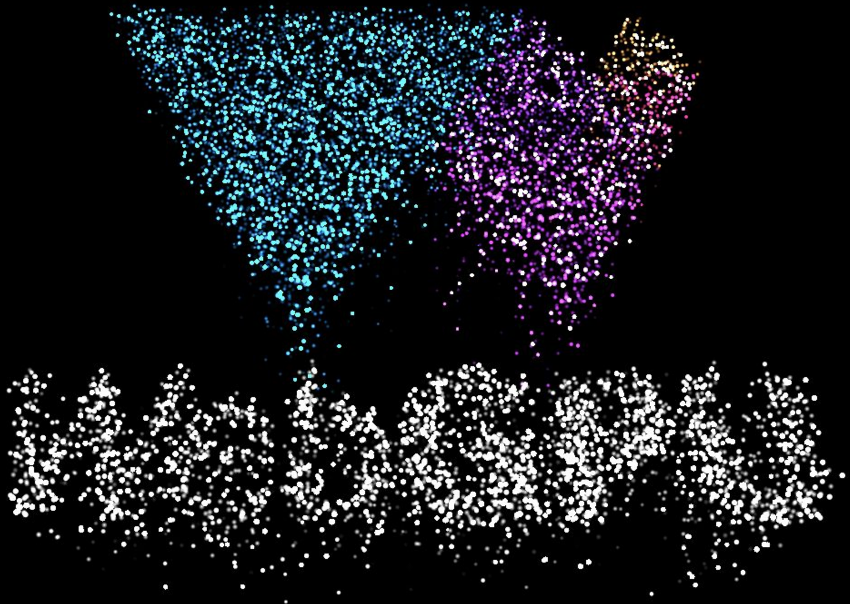
Chrome Console:

```
⚠️Binding offset (256) is larger than the size (144) of [Buffer "Projection Buffer"].
    - While validating entries[0] as a Buffer
    - While validating [BindGroupDescriptor "FrameBindGroup"] against [BindGroupLayout]
    - While calling CreateBindGroup([BindGroupDescriptor "Frame Bind Group"]).

  Debug group stack:
    > "Drawing Metaballs"
    > "Main Render Pass"
    > "Frame Loop"
```

A more complex example from one of my WebGPU apps

WebGPU tips

Best practices for performance



WebGPU Samples- Particles

Minimize the number of pipelines you use

- More pipelines == more state switching == less performance
- May not be trivial, depending on where your assets come from
- Wrote an entire article mostly focused on this: Efficiently rendering glTF models

Create pipelines in advance

- Creating a pipeline and then immediately using it works, but don’t do it!
- Create functions return immediately, start work on a different thread
- When you use it, the queue execution needs to wait for pending pipeline creations to finish
- This causes significant jank
- Make sure you leave some time between create and first use

```
const pipeline = device.createComputePipeline({
 compute: {
   module: shaderModule,
   entryPoint: 'computeMain'
 }
});
const commandEncoder = device.createCommandEncoder();
const passEncoder = commandEncoder.beginComputePass();
passEncoder.setPipeline(pipeline);
passEncoder.setBindGroup(0, bindGroup);
passEncoder.dispatchWorkgroups(128);
passEncoder.end();
device.queue.submit([commandEncoder.finish()]);
```

Create pipelines in advance

- Or, even better, use the **create*PipelineAsync** variants!
- The promise resolves when the pipeline is ready to use without any stalling

```
device.createComputePipelineAsync({
 compute: {
  module: shaderModule,
  entryPoint: 'computeMain'
 }
}).then((pipeline) => {
 const commandEncoder = device.createCommandEncoder();
 const passEncoder = commandEncoder.beginComputePass();
 passEncoder.setPipeline(pipeline);
 passEncoder.setBindGroup(0, bindGroup);
 passEncoder.dispatchWorkgroups(128);
 passEncoder.end();
 device.queue.submit([commandEncoder.finish()]);
});
```

Use RenderBundles!

- Render bundles are pre-recorded, partial, reusuable, render passes.
- They can contain most rendering commands (except for things like setting the scissor/viewport.)
- Can be “replayed” as part of an actual render pass later on.

```
const encoder = device.createRenderBundleEncoder({
 colorFormats: ['bgra8unorm'],
 depthStencilFormat: 'depth24plus',
});

encoder.setPipeline(pipeline);
encoder.setBindGroup(0, bindGroupA);
encoder.setVertexBuffer(0, vertexBufferA);
encoder.draw(1024);

encoder.setBindGroup(0, bindGroupB);
encoder.setVertexBuffer(0, vertexBufferB);
encoder.setIndexBuffer(indexBuffer);
encoder.drawIndexed(2048);

const renderBundle = encoder.finish();
```

Use RenderBundles!

- Render bundles can be executed alongside regular render pass commands
- The render pass state is reset to defaults before and after every bundle execution.
- Primarily for reducing JavaScript overhead of drawing. GPU performance is the same either way.

```
const renderPass = encoder.beginRenderPass(
 descriptor);

renderPass.setPipeline(renderPipeline);
renderPass.draw(3);

renderPass.executeBundles([renderBundle]);

renderPass.setPipeline(renderPipeline);
renderPass.draw(3);

renderPass.end();
```

Additional Resources

- WebGPU Samples - Written by members of the Chrome team
- Raw WebGPU - A great introductory tutorial
- All the cores, none of the canvas - Compute-focused introduction
- Efficiently rendering glTF models - Focuses on efficient WebGPU patterns
- WebGPU best practices - Some targeted best practice advice for specific APIs
- WebGPU Spec, WGSL Spec - For a bit of light reading
- Matrix Chat - Chat with other WebGPU devs and the browser implementers!
- Babylon.js, Three.js, - Libraries with WebGPU support

Career Advice

- When you make things, try to make them in publicly visible spaces
- Even if you’re looking for a career developing native apps, having online projects that can be viewed instantly is great for demonstrating your experience

Career Advice

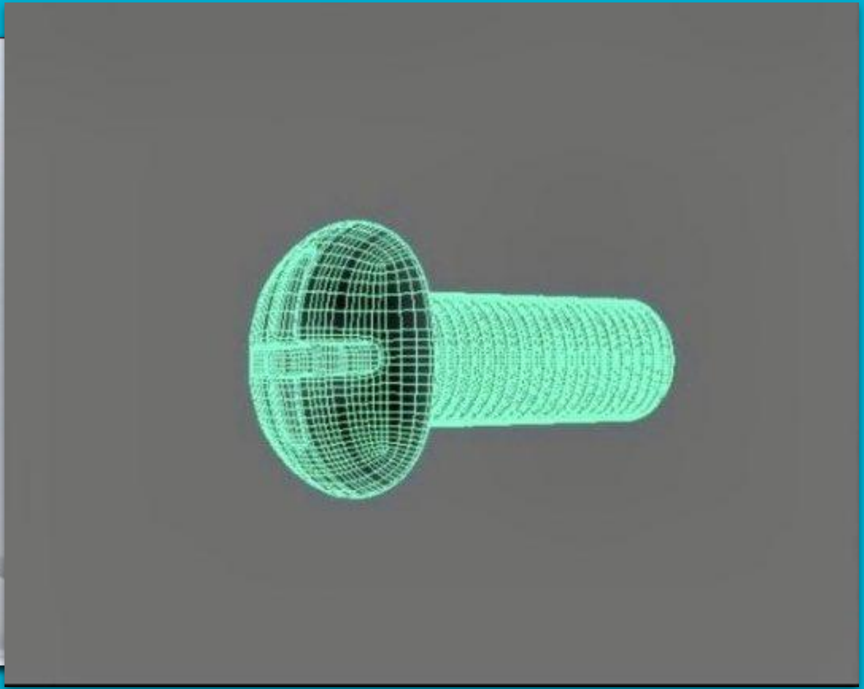
- Building your own engine from scratch is a fun, educational exercise, but it’s rarely the right choice for a project you need to ship
- Finding ways to bend existing frameworks (Three.js, Unity, etc) to your will can be hard, but it’s a valuable skillset
- Building focused libraries/tools that compliment or fill gaps in existing ecosystems is also extremely valuable

Career Advice

- Learn some of the art tools in your field!
- Teaching myself the basics of Blender has been a skillset I’ve fallen back on again and again
- You don’t even have to be a good artist, just figure out how to import, tweak, combine, and export models

Career Advice

- Learn how to diagnose/avoid common performance problems in code and art assets



Thank you!

Happy to take questions

Feel free to reach out any time!

Brandon Jones

Email - bajones@google.com

Twitter - @tojiro

These Slides - bit.ly/3D3eNCw