WebGPU Bind Group best practices

Introduction

In WebGPU resources are exposed to a shader through the `GPUBindGroup` structure, alongside the layout definitions that support it: `GPUBindGroupLayout` and `GPUPipelineLayout`, and the declarations of the bind groups in the shaders themselves. The shaders, the layouts, and the bind groups all need to be made compatible, often repeating information at each stage. As a result this aspect of the API can feel unnecessarily complicated at first glance, and frustrating to work with.

This doc is focused on explaining why the Bind Group interface is structured the way it is and how to best take advantage of it.

Bind Group usage overview

First, let's walk through a simple usage of bind groups and how each stage relates to one another. If you are familiar with WebGL, bind groups is the mechanism that takes the place of setting WebGL uniforms through the `gl.uniform*()` calls, though they're more closely equivalent to WebGL 2.0's Uniform Buffer Objects. In all cases, you are taking some data supplied by the application (values in a buffer, textures, or samplers) and making them accessible to the shader that you are about to run.

To explain how this works, let's walk through how some common piece of rendering data can be exposed to a WebGPU shader: camera values, a model transform, a base color texture, and a sampler.

Shaders and Bind Group Layouts

First we need to declare those values in the WGSL shader code, which would look something like this:

```
// vertexModuleA source:

// Declaration of bind groups used by the vertex shader
struct Camera {
  projection : matrix4x4f,
  view : matrix4x4f,
  position: vec3f,
};
@group(0) @binding(0) var<uniform> camera : Camera;

@group(0) @binding(1) var<uniform> model : matrix4x4f;

// The remainder of this shader doesn't affect the bind groups.
struct VertexOutput {
  @builtin(position) position : vec4f,
  @location(0) texcoord : vec2f,
};

@vertex fn vertexMain(
    @location(0) position : vec3f,
    @location(1) texcoord : vec2f) -> VertexOutput {
  var output : VertexOutput;
  output.position = camera.projection * camera.view * model * vec4f(position, 1);
  output.texcoord = texcoord;
  return output;
}

// fragmentModuleA source:
```

```
// Declaration of bind groups used by the fragment shader
@group(0) @binding(2) var baseColor : texture_2d<f32>;
@group(0) @binding(3) var baseColorSampler : sampler;

// The remainder of this shader doesn't affect the bind groups.
@fragment fn fragmentMain(
    @location(0) texcoord : vec2f) -> @location(0) vec4f {
  return textureSample(baseColor, baseColorSampler, texcoord);
}
```

A few things to note here: Each of the values being exposed by the application are given a `@group` and a `@binding` number. Multiple bindings can share the same `@group`, and must have unique (but not necessarily sequential) `@binding` values within a given `@group`. For the uniforms, which will come from buffers supplied by the app, the type can either be a struct (like the `camera` uniform) or a single value (like the `model` uniform).

In the associated application code, a `GPUBindGroupLayout` needs to be defined which has an entry for every `@binding` in a given `@group` from the shader. For `@group(0)` in the shader snippet above, such a layout would look like this:

```
const bindGroupLayout = gpuDevice.createBindGroupLayout({
  entries: [{
    binding: 0, // camera uniforms
    visibility: GPUShaderStage.VERTEX,
    buffer: {},
  }, {
    binding: 1, // model uniform
    visibility: GPUShaderStage.VERTEX,
    buffer: {},
  }, {
    binding: 2, // baseColor texture
    visibility: GPUShaderStage.FRAGMENT,
    texture: {},
  }, {
    binding: 3, // baseColor sampler
    visibility: GPUShaderStage.FRAGMENT,
    sampler: {},
  }]
});
```

Note that each entry has an explicit `binding` which matches up with the `@binding` values in the shader. They also state which shader stages they are visible to (which can be multiple stages). And finally it indicates the binding type, like `buffer`, `texture`, or `sampler`. Each of those types have options to further specify different types of binding, such as if the buffer type is `'storage'` rather than `'uniform'`. But if the defaults are appropriate at the very least an empty dictionary must be set to signal the type, as seen above.

Pipelines and Pipeline Layouts

When creating a render or compute pipeline, the bind groups layouts are passed in via a `GPUPipelineLayout`. The pipeline layout is a list of the `GPUBindGroupLayouts` used by the pipeline, ordered by `@group` index. Since these shaders only uses one group, we only need a single entry:

```
const pipelineLayout = gpuDevice.createPipelineLayout({
  bindGroupLayouts: [
    bindGroupLayout, // @group(0)
  ]
});

const pipelineA = gpuDevice.createRenderPipeline({
  layout: pipelineLayout,
  // Most render pipeline values omitted for simplicity.
  vertex: {
    module: vertexModuleA,
    entryPoint: 'vertexMain'
  },
```

```
  fragment: {
    module: fragmentModuleA,
    entryPoint: 'fragmentMain'
  }
});
```

If any `@group` in the shaders doesn't have a matching entry in the pipeline layout, or any `@binding` in the shader doesn't have a matching entry in the `@group`'s corresponding bind group layout, pipeline creation will fail with an error.

Resources and Bind Groups

Next, a `GPUBindGroup` can be created which points at the actual resources that will be exposed to the shader. First ensure that the resources that are going to be exposed to the shader have been created with the appropriate sizes and usages:

```
const cameraBuffer = gpuDevice.createBuffer({
  size: 144, // Room for two 4x4 matrices and a vec3
  usage: GPUBufferUsage.COPY_DST | GPUBufferUsage.UNIFORM
});

const modelBuffer = gpuDevice.createBuffer({
  size: 64, // Room for one 4x4 matrix
  usage: GPUBufferUsage.COPY_DST | GPUBufferUsage.UNIFORM
});

const baseColorTexture = gpuDevice.createTexture({
  size: { width: 256, height: 256 }
  usage: GPUTextureUsage.COPY_DST | GPUTextureUsage.TEXTURE_BINDING,
  format: 'rgba8unorm',
});

const baseColorSampler = gpuDevice.createSampler({
  magFilter: "linear",
  minFilter: "linear",
  mipmapFilter: "linear",
  addressModeU: "repeat",
  addressModeV: "repeat",
});
```

Then create a bind group that points to those resources, using the same `GPUBindGroupLayout` that was given to the pipeline layout during render pipeline creation:

```
const bindGroup = gpuDevice.createBindGroup({
  layout: bindGroupLayout,
  entries: [{
    binding: 0,
    resource: { buffer: cameraBuffer },
  }, {
    binding: 1,
    resource: { buffer: modelBuffer },
  }, {
    binding: 2,
    resource: baseColorTexture.createView(),
  }, {
    binding: 3,
    resource: baseColorSampler,
  }],
});
```

Note again that each entry has an explicit `binding` which matches up with the `@binding` values in the shader and the bind group layout. The `buffer` bindings can also be given explicit offsets and sizes at bind group creation time, if necessary. From this point on the resources that the bind group points at cannot be changed, but the resources themselves can still have their contents updated, and those changes will be reflected in the

shader. In other words: Bind groups do not create snapshots of the resources.

Setting Bind Groups and Pipelines

Finally, when recording a command buffer, the buffers should be updated with the latest uniform values and the bind group should be set prior to one of the `draw*()` or `dispatch*()` methods being called in a render or compute pass respectively:

```
// Place the most recent camera values in an array at the appropriate offsets.
const cameraArray = new Float32Array(36);
cameraArray.set(projectionMatrix, 0);
cameraArray.set(viewMatrix, 16);
cameraArray.set(cameraPosition, 32);

// Update the camera uniform buffer
device.queue.writeBuffer(cameraBuffer, 0, cameraArray);

// Update the model uniform buffer
device.queue.writeBuffer(modelBuffer, 0, modelMatrix);

// Record and submit the render commands for our scene.
const commandEncoder = device.createCommandEncoder();
const passEncoder = commandEncoder.beginRenderPass({ /* Omitted for simplicity */ });

passEncoder.setPipeline(pipelineA);
passEncoder.setBindGroup(0, bindGroup); // @group(0)
passEncoder.draw(128);

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

`setBindGroup()` must be called once for each `@group` present in the shader/pipeline layout. If no bind group was set or the bind group given doesn't use the corresponding bind group layout in the pipeline's pipeline layout, then the call to `draw()` or `dispatchWorkgroups()` will fail. Assuming everything is compatible, however, the shader will be executed with the resources from the currently set bind groups.

Auto layout generation

If you are familiar with the WebGL flow for uniforms, the above steps likely feel incredibly verbose. Additionally, you may notice that there's a fair amount of repetition involved, which increases the chance that you may update the code in one location and forget to update it in another, leading to errors.

There is a mechanism available to simplify at least part of this process, but it's one that should be used with caution: `layout: 'auto'`.

When creating a render or compute pipeline, you have the option of passing in the `'auto'` keyword rather than an explicit `GPUPipelineLayout`, at which point the pipeline will create it's own internal pipeline layout based on the bindings declared in the shader. Then when creating bind groups to use with that pipeline, the auto-generated layouts can be queried from the pipeline with `getBindGroupLayout(index)`.

```
const autoPipelineA = gpuDevice.createRenderPipeline({
  layout: 'auto',
  // Most render pipeline values omitted for simplicity.
  vertex: {
    module: vertexModuleA,
    entryPoint: 'vertexMain'
  },
  fragment: {
    module: fragmentModuleA,
    entryPoint: 'fragmentMain'
  }
});
```

```
const autoBindGroupA = gpuDevice.createBindGroup({
  layout: autoPipelineA.getBindGroupLayout(0), // @group(0)
  entries: [{
    binding: 0,
    resource: { buffer: cameraBuffer },
  }, {
    binding: 1,
    resource: { buffer: modelBuffer },
  }, {
    binding: 2,
    resource: baseColorTexture.createView(),
  }, {
    binding: 3,
    resource: baseColorSampler,
  }],
});
```

This eliminates the need to manually create bind group layouts or pipeline layouts, which can make it seems like an appealing option.

The downside of `layout: 'auto'`

Using auto-generated layouts comes with an important caveat, however. The bind group layouts it produces (and thus and bind groups created with them) can only be used with the pipeline that generated the layout.

To see why that's a problem, consider a scenario where we have a version of our above shader that swaps the red and blue channel of the texture:

```
// fragmentModuleB source
@group(0) @binding(2) var baseColor : texture_2d<f32>;
@group(0) @binding(3) var baseColorSampler : sampler;

@fragment fn fragmentMain(
    @location(0) texcoord : vec2f) -> @location(0) vec4f {
  return textureSample(baseColor, baseColorSampler, texcoord).bgra;
}
```

If we used `layout: 'auto'` when creating the pipeline which uses this shader, we would also have to create a new bind group using the auto-generated layout which would only be compatible with that specific pipeline:

```
const autoPipelineB = gpuDevice.createRenderPipeline({
  layout: 'auto',
  // Most render pipeline values omitted for simplicity.
  vertex: {
    module: vertexModuleA,
    entryPoint: 'vertexMain'
  },
  fragment: {
    // Note that we're using the second fragment shader with the first vertex shader
    module: fragmentModuleB,
    entryPoint: 'fragmentMain'
  }
});

const autoBindGroupB = gpuDevice.createBindGroup({
  layout: autoPipelineB.getBindGroupLayout(0), // @group(0)
  entries: [{
    binding: 0,
    resource: { buffer: cameraBuffer },
  }, {
    binding: 1,
    resource: { buffer: modelBuffer },
  }, {
    binding: 2,
    resource: baseColorTexture.createView(),
```

```
  }, {
    binding: 3,
    resource: baseColorSampler,
  }],
});
```

As you can see, this bind group is identical to the one we created for the first pipeline aside from the layout, so we're just duplicating work. That duplication of work is also visible when recording the render pass:

```
const commandEncoder = gpuDevice.createCommandEncoder();
const passEncoder = commandEncoder.beginRenderPass({ /* Omitted for simplicity */ });

passEncoder.setVertexBuffer(0, vertexBuffer);

passEncoder.setPipeline(autoPipelineA);
passEncoder.setBindGroup(0, autoBindGroupA); // @group(0)
passEncoder.draw(128);

passEncoder.setPipeline(autoPipelineB);
passEncoder.setBindGroup(0, autoBindGroupB); // @group(0)
passEncoder.draw(128);

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

We're calling `setBindGroup()` twice now with effectively the same data, which is not something that the underlying implementation is likely to optimize away. Changing any piece of state while in a render/compute pass has a performance cost, and paying that cost to end up with effectively the same state anyway is never desirable.

Hard to predict

Another consideration is that that when using `layout: 'auto'` the resulting bind group layout may not always be what you expected. Consider the following compute shader:

```
// computeModuleA source:
struct GlobalState {
  timeDelta : f32,
  gravity : vec3f
}
@group(0) @binding(0) var<uniform> globalState : GlobalState;

struct Particle {
  pos : vec2f,
  vel : vec2f,
}
@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;
  particlesOut[index].vel = vVel + vec3f(0, 0, -9.8);
}
```

By glancing at the shader code above, you may think that this would be the appropriate bind group for that pipeline:

```
const computePipelineA = device.createComputePipeline({
  layout: 'auto',
```

```
  compute: {
    module: computeModuleA,
    entryPoint: 'computeMain',
  }
});

const computeBindGroupA = gpuDevice.createBindGroup({
  layout: computePipelineA.getBindGroupLayout(0), // @group(0)
  entries: [{
    binding: 0,
    resource: { buffer: globalStateBuffer },
  }, {
    binding: 1,
    resource: { buffer: particleInputBuffer },
  }, {
    binding: 2,
    resource: { buffer: particleOutputBuffer },
  }],
});
```

But that will result in an error! Why? As it turns out the `globalState` uniform was never statically used in the shader body, so during pipeline creation it was omitted by the automatic layout creation.

In this case this would most likely represent a bug in the shader, which probably wants to make use of both the `timeDelta` and `gravity` variables when updating the particles, so it would be relatively easy to fix. But this scenario could just as easily have come about because the uniform use was commented out while debugging, in which case a bind group that worked perfectly well previously suddenly begins failing.

Using explicit bind group layouts sidesteps these issues by not forcing you to have a detailed understanding of how WebGPU's internal shader processing works.

Use sparingly

As a result of the above considerations, effective use of `layout: 'auto'` is likely to be limited to pipelines that have unique resource needs. Some compute shaders or post-processing render passes, for example, may use combinations of resources that no other pipeline in the application needs, and as such using letting it auto-generate the layout may be a reasonable choice.

However, any time you have multiple pipelines that need the same data, or at least data that fits the same structure, you should *always* prefer to use explicitly defined pipeline layouts.

Bind Group reuse

Explicit pipeline layouts allow for bind groups to be re-used between pipelines, which can be a big win for efficiency.

Looking back at the `shaderModuleB` source, we can see that it's using the same bindings in the same locations as in `shaderModuleA`, which means they can both use the same bind group layouts (and in this case pipeline layout as well):

```
const pipelineB = gpuDevice.createRenderPipeline({
  layout: pipelineLayout,
  // Most render pipeline values omitted for simplicity.
  vertex: {
    module: vertexModuleA,
    entryPoint: 'vertexMain'
  },
  fragment: {
    module: fragmentModuleB,
    entryPoint: 'fragmentMain'
  }
```

```
});
```

And now when recording our render commands we only have to set the bind group once, with both pipelines making use of it.

```
const commandEncoder = gpuDevice.createCommandEncoder();
const passEncoder = commandEncoder.beginRenderPass({ /* Omitted for simplicity */ });

passEncoder.setVertexBuffer(0, vertexBuffer);

passEncoder.setBindGroup(0, bindGroup); // @group(0)

passEncoder.setPipeline(pipelineA);
passEncoder.draw(128);

passEncoder.setPipeline(pipelineB);
passEncoder.draw(128);

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

Bind group subset reuse

This pattern applies in scenarios where the resources used by a pipeline is only a subset of the resource in the bind group as well! For example, what if we have yet another variant of our fragment shader that renders the mesh with a solid color, and thus has no need for the texture or sampler:

```
// fragmentModuleC source
@fragment fn fragmentMain(
    @location(0) texcoord : vec2f) -> @location(0) vec4f {
  return vec4<32f>(1.0, 0.0, 1.0, 1.0);
}
```

A pipeline which uses this shader can still use the same bind group layout as the previous ones, even though it didn’t make use of every binding in it:

```
const pipelineC = gpuDevice.createRenderPipeline({
  layout: pipelineLayout,
  // Most render pipeline values omitted for simplicity.
  vertex: {
    module: vertexModuleA,
    entryPoint: 'vertexMain'
  },
  fragment: {
    module: fragmentModuleC,
    entryPoint: 'fragmentMain'
  }
});

const commandEncoder = gpuDevice.createCommandEncoder();
const passEncoder = commandEncoder.beginRenderPass({ /* Omitted for simplicity */ });

passEncoder.setVertexBuffer(0, vertexBuffer);

passEncoder.setBindGroup(0, bindGroup); // @group(0)

passEncoder.setPipeline(pipelineA);
passEncoder.draw(128);

passEncoder.setPipeline(pipelineB);
passEncoder.draw(128);

passEncoder.setPipeline(pipelineC);
passEncoder.draw(128);

passEncoder.end();
```

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

In this scenario `pipelineC` simply ignores the texture and sampler that have been bound, while the other two pipelines make use of them. It's worth noting that even if a pipeline doesn't make use of a particular resource the driver will still do the work to make it accessible to the shaders, so it's a good idea to ensure the resources in your bind groups are necessary and used by at least some of the pipelines that share it's layout.

Grouping resources based on frequency of change

Thus far we've only been dealing with a single bind group, but it's fairly obvious from the above code that WebGPU is architected to make use of multiple bind groups. By using multiple `@group()` indexes in the shader, the resources you provide can be split across multiple bind groups, each requiring their own bind group layout. That's additional work to create the layouts and bind groups, as well as additional binding calls, though, so what's the benefit of breaking bind groups up like that?

First, let's consider a more realistic rendering pattern than the simplistic examples above, and how the bind group resources we're exposing relate to it. We can expect any scene to be made up of multiple meshes, all of which will have their own transform matrix. Additionally, each mesh will have a material (simplified here to just a texture) which may be shared among several other meshes. (ie: A brick or concrete material will probably be used in multiple places.) Finally there are some values in the bind group, like the camera uniforms, which will be the same for everything in the scene.

With the current structure that uses a single monolithic bind group, in order to represent a scene like that we'd need to duplicate a lot of the bind group information for each mesh, because they all require at least one unique piece of data. In psuedocode it would look something like this:

```
const renderableMeshes = [];

function createSceneBindGroups(meshes) {
  for (const mesh of meshes) {
    mesh.bindGroup = gpuDevice.createBindGroup({
      layout: bindGroupLayout,
      entries: [{
        binding: 0,
        resource: { buffer: cameraBuffer },
      }, {
        binding: 1,
        resource: { buffer: mesh.modelMatrixBuffer },
      }, {
        binding: 2,
        resource: mesh.material.baseColorTexture.createView(),
      }, {
        binding: 3,
        resource: mesh.material.baseColorSampler,
      }],
    });

    renderableMeshes.push(mesh);
  }
}

function renderScene(passEncoder) {
  // Assume all meshes can use the same pipeline, for simplicity
  passEncoder.setPipeline(pipelineA);

  for (mesh of renderableMeshes) {
    passEncoder.setBindGroup(0, mesh.bindGroup);
    passEncoder.setVertexBuffer(0, mesh.vertexBuffer);
    passEncoder.draw(mesh.drawCount);
  }
}
```

While this *will* draw the meshes correctly, it's not the most efficient way to do it because there's a lot of repeated setting of the same state. Again, even though a subset of the bind group resources are unchanged between calls to `setBindGroup()`, you shouldn't count on the implementation/driver to optimize that away for you. Even if it *was* taken care of by the driver on one platform it probably won't be on all of them.

So what's the solution? We can split the resources into groups based on the frequency of how often they need to change. The camera uniforms don't change for the entirety of the render pass, so they can be in their own bind group. Material properties change semi-frequently, but not for every mesh, so they can go into separate bind group. And finally, the model matrix is different for every draw call, so it belongs in yet another bind group.

This results in an updates shader that looks like this. Pay close attention to the changes in the `@group` and `@binding` indices:

```
// shaderModuleD source:

struct Camera {
  projection : matrix4x4f,
  view : matrix4x4f,
  position: vec3f,
};
@group(0) @binding(0) var<uniform> camera : Camera;

@group(1) @binding(0) var baseColor : texture_2d<f32>;
@group(1) @binding(1) var baseColorSampler : sampler;

@group(2) @binding(0) var<uniform> model : matrix4x4f;

// The remainder of this shader doesn't affect the bind groups.
struct VertexOutput {
  @builtin(position) position : vec4f,
  @location(0) texcoord : vec2f,
};

@vertex fn vertexMain(
    @location(0) position : vec3f,
    @location(1) texcoord : vec2f) -> VertexOutput {
  var output : VertexOutput;
  output.position = camera.projection * camera.view * model * vec4f(position, 1);
  output.texcoord = texcoord;
  return output;
}

// The remainder of this shader doesn't affect the bind groups.
@fragment fn fragmentMain(
    @location(0) texcoord : vec2f) -> @location(0) vec4f {
  return textureSample(baseColor, baseColorSampler, texcoord);
}
```

The corresponding bind group layouts and pipeline layout also need to be updated to account for the new arrangement of resources:

```
const cameraBindGroupLayout = gpuDevice.createBindGroupLayout({
  entries: [{
    binding: 0, // camera uniforms
    visibility: GPUShaderStage.VERTEX,
    buffer: {},
  }]
});
const materialBindGroupLayout = gpuDevice.createBindGroupLayout({
  entries: [{
    binding: 0, // baseColor texture
    visibility: GPUShaderStage.FRAGMENT,
    texture: {},
  }, {
    binding: 1, // baseColor sampler
```

```
    visibility: GPUShaderStage.FRAGMENT,
    sampler: {},
  }]
});
const meshBindGroupLayout = gpuDevice.createBindGroupLayout({
  entries: [{
    binding: 0, // model uniform
    visibility: GPUShaderStage.VERTEX,
    buffer: {},
  }]
});

const pipelineDLayout = gpuDevice.createPipelineLayout({
  bindGroupLayouts: [
    cameraBindGroupLayout, // @group(0)
    materialBindGroupLayout, // @group(1)
    meshBindGroupLayout, // @group(2)
  ]
});

const pipelineD = gpuDevice.createRenderPipeline({
  layout: pipelineDLayout,
  // Most render pipeline values omitted for simplicity.
  vertex: {
    module: shaderModuleD,
    entryPoint: 'vertexMain'
  },
  fragment: {
    module: shaderModuleD,
    entryPoint: 'fragmentMain'
  }
});
```

And finally, the bind groups themselves need to be created with the new groupings. This time we will also create them with an eye towards reducing duplication. Additionally, we'll take the opportunity to sort our meshes in a way that will reduce the amount of bind group setting that we need to do during the render loop:

```
let cameraBindGroup;
const renderableMaterials = new Map();

function createSceneBindGroups(meshes) {
  // Create a single bind group for the camera uniforms
  cameraBindGroup = gpuDevice.createBindGroup({
    layout: cameraBindGroupLayout,
    entries: [{
      binding: 0,
      resource: { buffer: cameraBuffer },
    }],
  });

  for (const mesh of meshes) {
    // Find or create a renderableMaterials entry for the mesh's material.
    // renderableMaterials will contain the bind group and associated meshes for
    // each material.
    let renderableMaterial = renderableMaterials.get(mesh.material);
    if (!renderableMaterial) {
      const materialBindGroup = gpuDevice.createBindGroup({
        layout: materialBindGroupLayout,
        entries: [{
          binding: 0,
          resource: mesh.material.baseColorTexture.createView(),
        }, {
          binding: 1,
          resource: mesh.material.baseColorSampler,
        }],
      });
      renderableMaterial = {
        meshes: [],
        bindGroup: materialBindGroup
```

```
      };
      renderableMaterials.set(mesh.material, renderableMaterial);
    }

    // Store meshes grouped by the material that they use.
    renderableMaterial.meshes.push(mesh);

    // Create a bind group for the mesh's transform
    mesh.bindGroup = gpuDevice.createBindGroup({
      layout: meshBindGroupLayout,
      entries: [{
        binding: 0,
        resource: { buffer: mesh.modelMatrixBuffer },
      }],
    });
  }
}

function renderScene(passEncoder) {
  // Assume all meshes can use the same pipeline, for simplicity
  passEncoder.setPipeline(pipelineA);

  // Set the camera bind group once, since it applies to all meshes
  passEncoder.setBindGroup(0, cameraBindGroup); // @group(0)

  // Loop through all the materials and set the material bind group once for each of them.
  for (const material of renderableMaterials.values()) {
    passEncoder.setBindGroup(1, material.bindGroup); // @group(1)

    // Loop through each mesh using the current material, bind it's information, and draw.
    for (const mesh of material.meshes) {
      passEncoder.setBindGroup(2, mesh.bindGroup); // @group(2)
      passEncoder.setVertexBuffer(0, mesh.vertexBuffer);
      passEncoder.draw(mesh.drawCount);
    }
  }
}
```

You can see that the above code establishes a kind of hierarchy of state changes, with the values that change least frequently being set at the very top of the function and then working into progressively tighter loops that each represent more frequently changing values. In broad terms, this is the type of pattern that you want to strive for when performing any sort of rendering or compute operations in WebGPU: Set state as infrequently as possible, and break state up based on how frequently it changes.

Note that many WebGPU implementations will be limited to 4 bind groups at a time (check `limits.maxBindGroups` on the `GPUAdapter` to determine the system's actual limit, and pass the desired count in when calling `requestDevice()` to raise the limit in your own application.) This should be enough for most cases, though. The separation of bind groups into "Per Frame/Per Material/Per Draw" state as seen above is quite common.

Group indices matter

The WebGPU API is technically agnostic to the order that bind groups are declared in and the order that `setBindGroup()` is called in. Placing the camera bind group at `@group(2)` and the model bind group at `@group(0)` works as expected. However, the underlying native APIs may have preferences about the order that the groups are declared and set in for performance purposes. Thus, to ensure the best performance across the board, you should prefer to have `@group(0)` contain the values that change least frequently between draw/dispatch calls, and each subsequent `@group` index contain data that changes at progressively higher frequencies.

Reusing pipeline layouts

In some cases you may find that you have a pipeline that doesn't make use of a bind group, but otherwise

could share bind group state with other pipelines in your application.

For example, let's take another look at the shader above that didn't make use of the texture and sampler now that we're splitting the bind groups up. You can see that if we remove the unused `@binding`s we're left with a gap in the group indices:

```
// shaderModuleE source:

struct Camera {
  projection : matrix4x4f,
  view : matrix4x4f,
  position: vec3f,
};
@group(0) @binding(0) var<uniform> camera : Camera;

@group(2) @binding(0) var<uniform> model : matrix4x4f;

struct VertexOutput {
  @builtin(position) position : vec4f,
  @location(0) texcoord : vec2f,
};

@vertex fn vertexMain(
    @location(0) position : vec3f,
    @location(1) texcoord : vec2f) -> VertexOutput {
  var output : VertexOutput;
  output.position = camera.projection * camera.view * model * vec4f(position, 1);
  output.texcoord = texcoord;
  return output;
}

@fragment fn fragmentMain(
    @location(0) texcoord : vec2f) -> @location(0) vec4f {
  return vec4f(1, 0, 1, 1);
}
```

You may be tempted to put unused bindings in there for the sake of keeping the pipeline layouts matching, but there's no need to do that. You can still use the same pipeline layout with this shader as you did with the others, even though it includes a bind group layout that's not referenced here.

```
const pipelineE = gpuDevice.createRenderPipeline({
  layout: pipelineDLayout, // Re-using the same pipeline from above
  // Most render pipeline values omitted for simplicity.
  vertex: {
    module: shaderModuleE,
    entryPoint: 'vertexMain'
  },
  fragment: {
    module: shaderModuleE,
    entryPoint: 'fragmentMain'
  }
});
```

Please note, however, that at draw time you will still need to set a bind group for every bind group layout in the pipeline layout, whether the shader used it or not.

```
const commandEncoder = gpuDevice.createCommandEncoder();
const passEncoder = commandEncoder.beginRenderPass({ /* Omitted for simplicity */ });

passEncoder.setVertexBuffer(0, vertexBuffer);

passEncoder.setPipeline(pipelineE);

passEncoder.setBindGroup(0, cameraBindGroup);
passEncoder.setBindGroup(1, materialBindGroup); // Required even though it's unused!
passEncoder.setBindGroup(2, meshBindGroup);
```

```
passEncoder.draw(128);

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

Have fun, and make cool stuff!

Bind group management can take some getting used to initially, especially if you are already familiar with the simpler (but less efficient) patterns used by WebGL 1.0 uniforms. Once you get the hang of it, however, you'll find that it gives you much more explicit control over how and when shader resources are updated, which in turn leads to less overhead and faster applications!

As a quick review: Despite it seeming like additional work, keep in mind that explicitly setting up your own bind group layouts and pipeline layouts is usually the right choice for anything other than the most trivial applications, and `layout: 'auto'` should be reserved for one-off pipelines that share little to no state with the rest of the app. Try to get as many pipelines as possible to re-use the same bind group layouts, and take care to break up bind group resources based on how frequently you update them.

Good luck on whatever projects are ahead of you, I can't wait to see what the spectacularly creative web community builds!