Codelabs

Your first WebGPU app

1. Introduction

Last Updated: 2023-04-13

What is WebGPU?

WebGPU is a new, modern API for accessing the capabilities of your GPU in web apps.

Modern API

Before WebGPU, there was WebGL, which offered a subset of the features of WebGPU. It enabled a new class of rich web content, and developers have built amazing things with it. However, it was based on the OpenGL ES 2.0 API, released in 2007, which was based on the even older OpenGL API. GPUs have evolved significantly in that time, and the native APIs that are used to interface with them have evolved as well with Direct3D 12, Metal, and Vulkan.

WebGPU brings the advancements of these modern APIs to the web platform. It focuses on enabling GPU features in a cross-platform way, while presenting an API that feels natural on the web and is less verbose than some of the native APIs it's built on top of.

Rendering

GPUs are often associated with rendering fast, detailed graphics, and WebGPU is no exception. It has the features required to support many of today's most popular rendering techniques across both desktop and mobile GPUs, and provides a path for new features to be added in the future as hardware capabilities continue to evolve.

Compute

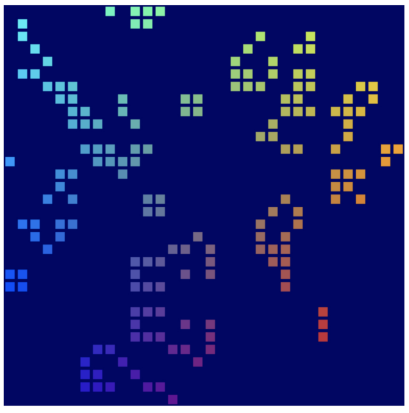
In addition to rendering, WebGPU unlocks the potential of your GPU for performing general purpose, highly parallel workloads. These *compute shaders* can be used standalone, without any rendering component, or as a tightly integrated part of your rendering pipeline.

In today's codelab you'll learn how to take advantage of both the rendering and compute capabilities of WebGPU in order to create a simple introductory project!

What you'll build

In this codelab, you build Conway's Game of Life using WebGPU. Your app will:

- Use WebGPU's rendering capabilities to draw simple 2D graphics.
- Use WebGPU's compute capabilities to perform the simulation.



Note: To simplify this codelab, all code snippets are vanilla JavaScript. WebGPU works perfectly well with TypeScript, though you'd need to include the @webgpu/types definitions. Once the API is more widely implemented, the types should become part of the standard TypeScript DOM and worker libraries.

The Game of Life is what's known as a cellular automaton, in which a grid of *cells* change state over time based on some set of rules. In the Game of Life cells become active or inactive depending on how many of their neighboring cells are active, which leads to interesting patterns that fluctuate as you watch.

What you'll learn

- How to set up WebGPU and configure a canvas.
- How to draw simple 2D geometry.
- How to use vertex and fragment shaders in order to modify what's being drawn.
- How to use compute shaders in order to perform a simple simulation.

This codelab focuses on introducing the fundamental concepts behind WebGPU. It's not intended to be a comprehensive review of the API, nor does it cover (or require) frequently related topics such as 3D matrix math.

What you'll need

- A recent version of Chrome (113 or later) on ChromeOS, macOS, or Windows. WebGPU is a cross-browser, cross-platform API but has not yet shipped everywhere.
- Knowledge of HTML, JavaScript, and Chrome DevTools.

Familiarity with other Graphics APIs, such as WebGL, Metal, Vulkan, or Direct3D, is **not required**, but if you have any experience with them you'll likely notice lots of similarities with WebGPU that may help jump-start your learning!

2. Get set up

Get the code

This codelab doesn't have any dependencies, and it walks you through every step needed to create the WebGPU app, so you don't need any code to get started. However, some working examples that can serve as checkpoints are available at https://glitch.com/edit/#!/your-first-webgpu-app. You can check them out and reference them as you go if you get stuck.

Use the developer console!

WebGPU is a fairly complex API with a lot of rules that enforce proper usage. Worse, because of how the API works, it can't raise typical JavaScript exceptions for many errors, making it harder to pinpoint exactly where the problem is coming from.

You *will* encounter problems when developing with WebGPU, especially as a beginner, and that's OK! The developers behind the API are aware of the challenges of working with GPU development, and have worked hard to ensure that any time your WebGPU code causes an error you will get back very detailed and helpful messages in the developer console that help you identify and fix the issue.

Keeping the console open while working on *any* web application is always useful, but it especially applies here!

3. Initialize WebGPU

Start with a <canvas>

WebGPU can be used without showing anything on the screen if all you want is to use it to do computations. But if you want to render anything, like we're going to be doing in the codelab, you need a canvas. So that's a good spot to start!

Create a new HTML document with a single `<canvas>` element in it, as well as a `<script>` tag where we query the canvas element. (Or use 00-starter-page.html from glitch.)

- Create an `index.html` file with the following code:

index.html

```
<!doctype html>

<html>
  <head>
    <meta charset="utf-8">
    <title>WebGPU Life</title>
  </head>
  <body>
    <canvas width="512" height="512"></canvas>
    <script type="module">
      const canvas = document.querySelector("canvas");

      // Your WebGPU code will begin here!
    </script>
  </body>
</html>
```

Note: Canvas elements have a default size of 300x150 pixels, which is small for this app's needs. Although it's not required to use WebGPU, you probably want to give the canvas a larger width and height. The example code above makes the canvas 512x512 CSS pixels, but you're free to choose whatever size you want.

Note: Giving the script tag a type of "`module`" allows you to use top-level awaits, which is useful for WebGPU initialization!

Request an adapter and device

Now you can get into the WebGPU bits! First, you should consider that APIs like WebGPU can take a while to propagate throughout the entire web ecosystem. As a result, a good first precautionary step is to check if the user's browser can use WebGPU.

1. To check if the `navigator.gpu` object, which serves as the entry point for WebGPU, exists, add the following code:

index.html

```
if (!navigator.gpu) {
  throw new Error("WebGPU not supported on this browser.");
}
```

Ideally, you want to inform the user if WebGPU is unavailable by having the page fall back to a mode that doesn't use WebGPU. (Maybe it could use WebGL instead?) For the purposes of this codelab, though, you just throw an error to stop the code from executing further.

Once you know that WebGPU is supported by the browser, the first step in initializing WebGPU for your app is to request a `GPUAdapter`. You can think of an adapter as WebGPU's representation of a specific piece of GPU hardware in your device.

2. To get an adapter, use the `navigator.gpu.requestAdapter()` method. It returns a promise, so it's most convenient to call it with `await`.

index.html

```
const adapter = await navigator.gpu.requestAdapter();
if (!adapter) {
  throw new Error("No appropriate GPUAdapter found.");
}
```

If no appropriate adapters can be found, the returned `adapter` value might be `null`, so you want to handle that possibility. It might happen if the user's browser supports WebGPU but their GPU hardware doesn't have all the features necessary to use WebGPU.

Most of the time it's OK to simply let the browser pick a default adapter, as you do here, but for more advanced needs there are arguments that can be passed to `requestAdapter()` that specify whether you want to use low-power or high-performance hardware on devices with multiple GPUs (like some laptops).

Once you have an adapter, the last step before you can start working with the GPU is to request a GPUDevice. The device is the main interface through which most interaction with the GPU happens.

3. Get the device by calling `adapter.requestDevice()`, which also returns a promise.

index.html

```
const device = await adapter.requestDevice();
```

As with `requestAdapter()`, there are options that can be passed here for more advanced uses like enabling specific hardware features or requesting higher limits, but for your purposes the defaults work just fine.

Configure the Canvas

Now that you have a device, there's one more thing to do if you want to use it to show anything on the page: configure the canvas to be used with the device you just created.

- To do this, first request a `GPUCanvasContext` from the canvas by calling `canvas.getContext("webgpu")`. (This is the same call that you'd use to initialize Canvas 2D or WebGL contexts, using the `2d` and `webgl` context types, respectively.) The `context` that it returns must then be associated with the device using the `configure()` method, like so:

index.html

```
const context = canvas.getContext("webgpu");
const canvasFormat = navigator.gpu.getPreferredCanvasFormat();
context.configure({
  device: device,
  format: canvasFormat,
});
```

There are a few options that can be passed here, but the most important ones are the *`device`* that you are going to use the context with and the *`format`*, which is the *texture format* that the context should use.

Textures are the objects that WebGPU uses to store image data, and each texture has a format that lets the GPU know how that data is laid out in memory. The details of how texture memory works are beyond the scope of this codelab. The important thing to know is that the canvas context provides textures for your code to draw into, and the format that you use can have an impact on how efficiently the canvas shows those images. Different types of devices perform best when using different texture formats, and if you don't use the device's preferred format it may cause extra memory copies to happen behind the scenes before the image can be displayed as part of the page.

Fortunately, you don't have to worry much about any of that because WebGPU tells you which format to use for your canvas! In almost all cases, you want to pass the value returned by calling `navigator.gpu.getPreferredCanvasFormat()`, as shown above.

> ★ **Note:** One big difference in how WebGPU works compared to WebGL is that because canvas configuration is separate from device creation you can have any number of canvases that are all being rendered by a single device! This will make certain use cases, like multi-pane 3D editors, much easier to develop.

Clear the Canvas

Now that you have a device and the canvas has been configured with it, you can start using the device to change the content of the canvas. To start, clear it with a solid color.

In order to do that—or pretty much anything else in WebGPU—you need to provide some commands to the GPU instructing it what to do.

1. To do this, have the device create a `GPUCommandEncoder`, which provides an interface for recording GPU commands.

index.html

```
const encoder = device.createCommandEncoder();
```

The commands you want to send to the GPU are related to rendering (in this case, clearing the canvas), so the next step is to use the `encoder` to begin a Render Pass.

Render passes are when all drawing operations in WebGPU happen. Each one starts off with a `beginRenderPass()` call, which defines the textures that receive the output of any drawing commands performed. More advanced uses can provide several textures, called *attachments*, with various purposes such as storing the depth of rendered geometry or providing antialiasing. For this app, however, you only need one.

2. Get the texture from the canvas context you created earlier by calling `context.getCurrentTexture()`, which returns a texture with a pixel width and height matching the canvas's `width` and `height` attributes and the `format` specified when you called `context.configure()`.

index.html

```
const pass = encoder.beginRenderPass({
  colorAttachments: [{
```

```
      view: context.getCurrentTexture().createView(),
      loadOp: "clear",
      storeOp: "store",
    }]
  });
```

The texture is given as the `view` property of a `colorAttachment`. Render passes require that you provide a `GPUTextureView` instead of a `GPUTexture`, which tells it which parts of the texture to render to. This only really matters for more advanced use cases, so here you call `createView()` with no arguments on the texture, indicating that you want the render pass to use the entire texture.

You also have to specify what you want the render pass to do with the texture when it starts and when it ends:

- A `loadOp` value of `"clear"` indicates that you want the texture to be cleared when the render pass starts.
- A `storeOp` value of `"store"` indicates that once the render pass is finished you want the results of any drawing done during the render pass saved into the texture.

Once the render pass has begun you do... nothing! At least for now. The act of starting the render pass with `loadOp: "clear"` is enough to clear the texture view and the canvas.

3. End the render pass by adding the following call immediately after `beginRenderPass()`:

index.html

```
pass.end();
```

It's important to know that simply making these calls does not cause the GPU to actually do anything. They're just recording commands for the GPU to do later.

4. In order to create a `GPUCommandBuffer`, call `finish()` on the command encoder. The command buffer is an opaque handle to the recorded commands.

index.html

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

5. Submit the command buffer to the GPU using the `queue` of the `GPUDevice`. The queue performs all GPU commands, ensuring that their execution is well ordered and properly synchronized. The queue's `submit()` method takes in an array of command buffers, though in this case you only have one.

index.html

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

Once you submit a command buffer, it cannot be used again, so there's no need to hold on to it. If you want to submit more commands, you need to build another command buffer. That's why it's fairly common to see those two steps collapsed into one, as is done in the sample pages for this codelab:

index.html

```
// Finish the command buffer and immediately submit it.
device.queue.submit([encoder.finish()]);
```

After you submit the commands to the GPU, let JavaScript return control to the browser. At that point, the browser sees that you've changed the current texture of the context and updates the canvas to display that texture as an image. If you want to update the canvas contents again after that, you need to record and submit a new command buffer, calling `context.getCurrentTexture()` again to get a new texture for a render pass.

6. Reload the page. Notice that the canvas is filled with black. Congratulations! That means that you've successfully created your first WebGPU app.

Pick a color!

To be honest, though, black squares are pretty boring. So take a moment before moving on to the next section in order to personalize it just a bit.

1. In the `device.beginRenderPass()` call, add a new line with a `clearValue` to the `colorAttachment`, like this:

index.html

```
const pass = encoder.beginRenderPass({
  colorAttachments: [{
    view: context.getCurrentTexture().createView(),
    loadOp: "clear",
    clearValue: { r: 0, g: 0, b: 0.4, a: 1 }, // New line
    storeOp: "store",
  }],
});
```

The `clearValue` instructs the render pass which color it should use when performing the `clear` operation at the beginning of the pass. The dictionary passed into it contains four values: `r` for *red*, `g` for *green*, `b` for *blue*, and `a` for *alpha* (transparency). Each value can range from `0` to `1`, and together they describe the value of that color channel. For example:

- `{ r: 1, g: 0, b: 0, a: 1 }` is bright red.

- `{ r: 1, g: 0, b: 1, a: 1 }` is bright purple.
- `{ r: 0, g: 0.3, b: 0, a: 1 }` is dark green.
- `{ r: 0.5, g: 0.5, b: 0.5, a: 1 }` is medium gray.
- `{ r: 0, g: 0, b: 0, a: 0 }` is the default, transparent black.

Note: Defining colors in WebGPU is very similar to setting CSS colors with the rgb() functional notation.

Pro Tip: For a little bit less typing, you can also use an array shorthand, giving values in RGBA order. Passing `[0, 0.5, 0.7, 1]` is the same as passing `{ r: 0, g: 0.5, b: 0.7, a: 1 }`.

The example code and screenshots in this codelab use a dark blue, but feel free to pick whatever color you want!

2. Once you've picked your color, reload the page. You should see your chosen color in the canvas.

4. Draw geometry

By the end of this section, your app will draw some simple geometry to the canvas: a colored square. Be warned now that it'll seem like a lot of work for such simple output, but that's because WebGPU is designed to render *lots* of geometry very efficiently. A side effect of this efficiency is that doing relatively simple things might feel unusually difficult, but that's the expectation if you're turning to an API like WebGPU—you want to do something a little more complex.

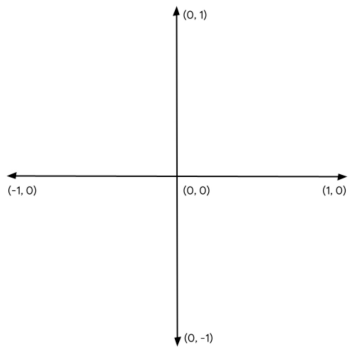
Understand how GPUs draw

Before any more code changes, it's worth doing a very quick, simplified, high-level overview of how GPUs create the shapes you see on screen. (Feel free to skip to the Defining Vertices section if you're already familiar with the basics of how GPU rendering works.)

Unlike an API like Canvas 2D that has lots of shapes and options ready for you to use, your GPU really only deals with a few different types of shapes (or *primitives* as they're referred to by WebGPU): points, lines, and triangles. For the purposes of this codelab you'll only use triangles.

GPUs work almost exclusively with triangles because triangles have a lot of nice mathematical properties that make them easy to process in a predictable and efficient way. Almost everything you draw with the GPU needs to be split up into triangles before the GPU can draw it, and those triangles must be defined by their corner points.

These points, or *vertices*, are given in terms of X, Y, and (for 3D content) Z values that define a point on a cartesian coordinate system defined by WebGPU or similar APIs. The structure of the coordinate system is easiest to think about in terms of how it relates to the canvas on your page. No matter how wide or tall your canvas is, the left edge is always at -1 on the X axis, and the right edge is always at +1 on the X axis. Similarly, the bottom edge is always -1 on the Y axis, and the top edge is +1 on the Y axis. That means that (0, 0) is always the center of the canvas, (-1, -1) is always the bottom-left corner, and (1, 1) is always the top-right corner. This is known as *Clip Space*.



The vertices are rarely defined in this coordinate system initially, so GPUs rely on small programs called *vertex shaders* to perform whatever math is necessary to transform the vertices into clip space, as well as any other calculations needed to draw the vertices. For example, the shader may apply some animation or calculate the direction from the vertex to a light source. These shaders are written by you, the WebGPU developer, and they provide an amazing amount of control over how the GPU works.

From there, the GPU takes all the triangles made up by these transformed vertices and determines which pixels on the screen are needed to draw them. Then it runs another small program you write called a *fragment shader* that calculates what color each pixel should be. That calculation can be as simple as *return green* or as complex as calculating the angle of the surface relative to sunlight bouncing off of other nearby surfaces, filtered through fog, and modified by how metallic the surface is. It's entirely under your control, which can be both empowering and overwhelming.

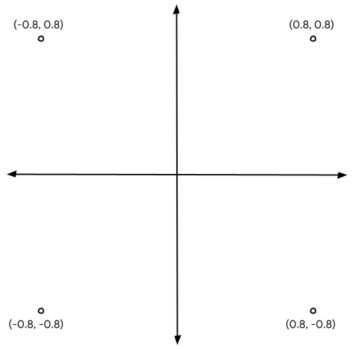
The results of those pixel colors are then accumulated into a texture, which is then able to be shown on screen.

Note: This codelab is only going to be dealing with 2D shapes, but this process still applies to 3D content, too! What's the difference between 2D and 3D according to the GPU? Just a bit of extra math! 3D content typically uses a series of matrices to transform the positions in a vertex shader (which is covered soon) prior to drawing the triangles in order to give the perception of depth and volume—but in the end almost everything the GPU draws is just triangles in clip space.

Define vertices

As mentioned earlier, The Game of Life simulation is shown as a grid of *cells*. Your app needs a way to visualize the grid, distinguishing active cells from inactive cells. The approach used by this codelab will be to draw colored squares in the active cells and leave inactive cells empty.

This means that you'll need to provide the GPU with four different points, one for each of the four corners of the square. For example, a square drawn in the center of the canvas, pulled in from the edges a ways, has corner coordinates like this:



In order to feed those coordinates to the GPU, you need to place the values in a TypedArray. If you're not familiar with it already, TypedArrays are a group of JavaScript objects that allows you to allocate contiguous blocks of memory and interpret each element in the series as a specific data type. For example, in a `Uint8Array`, each element in the array is a single, unsigned byte. TypedArrays are great for sending data back and forth with APIs that are sensitive to memory layout, like WebAssembly, WebAudio, and (of course) WebGPU.

For the square example, because the values are fractional, a `Float32Array` is appropriate.

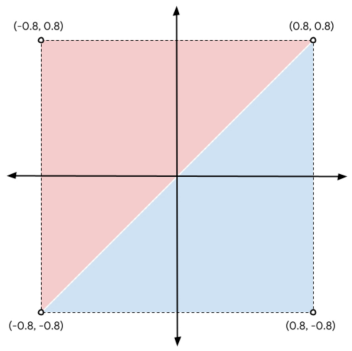
1. Create an array that holds all of the vertex positions in the diagram by placing the following array declaration in your code. A good place to put it is near the top, just under the `context.configure()` call.

index.html

```
const vertices = new Float32Array([
//   X,    Y,
  -0.8, -0.8,
   0.8, -0.8,
   0.8,  0.8,
  -0.8,  0.8,
]);
```

Note that the spacing and comment has no effect on the values; it's just for your convenience and to make it more readable. It helps you see that every pair of values makes up the X and Y coordinates for one vertex.

But there's a problem! GPUs work in terms of triangles, remember? So that means that you have to provide the vertices in groups of three. You have one group of four. The solution is to repeat two of the vertices to create two triangles sharing an edge through the middle of the square.



To form the square from the diagram, you have to list the (-0.8, -0.8) and (0.8, 0.8) vertices twice, once for the blue triangle and once for the red one. (You could also choose to split the square with the other two corners instead; it makes no difference.)

2. Update your previous `vertices` array to look something like this:

index.html

```
const vertices = new Float32Array([
//   X,    Y,
```

```
  -0.8, -0.8, // Triangle 1 (Blue)
   0.8, -0.8,
   0.8,  0.8,

  -0.8, -0.8, // Triangle 2 (Red)
   0.8,  0.8,
  -0.8,  0.8,
]);
```

Although the diagram shows a separation between the two triangles for clarity, the vertex positions are exactly the same, and the GPU renders them without gaps. It will render as a single, solid square.

> **Note:** You don't *have* to repeat the vertex data in order to make triangles. Using something called Index Buffers, you can feed a separate list of values to the GPU that tells it what vertices to connect together into triangles so that they don't need to be duplicated. It's like connect-the-dots! Because your vertex data is so simple, using Index Buffers is out of scope for this Codelab. But they're definitely something that you might want to make use of for more complex geometry.

Create a vertex buffer

The GPU cannot draw vertices with data from a JavaScript array. GPUs frequently have their own memory that is highly optimized for rendering, and so any data you want the GPU to use while it draws needs to be placed in that memory.

For a lot of values, including vertex data, the GPU-side memory is managed through `GPUBuffer` objects. A buffer is a block of memory that's easily accessible to the GPU and flagged for certain purposes. You can think of it a little bit like a GPU-visible TypedArray.

1. To create a buffer to hold your vertices, add the following call to `device.createBuffer()` after the definition of your `vertices` array.

index.html

```
const vertexBuffer = device.createBuffer({
  label: "Cell vertices",
  size: vertices.byteLength,
  usage: GPUBufferUsage.VERTEX | GPUBufferUsage.COPY_DST,
});
```

The first thing to notice is that you give the buffer a *label*. Every single WebGPU object you create can be given an optional label, and you definitely want to do so! The label is any string you want, as long as it helps you identify what the object is. If you run into any problems, those labels are used in the error messages WebGPU produces to help you understand what went wrong.

Next, give a *size* for the buffer in bytes. You need a buffer with 48 bytes, which you determine by multiplying the size of a 32-bit float (4 bytes) by the number of floats in your `vertices` array (12). Happily, TypedArrays already calculate their byteLength for you, and so you can use that when creating the buffer.

> **Note:** Get comfortable with this kind of byte-size math. Working with a GPU requires a fair amount of it!

Finally, you need to specify the *usage* of the buffer. This is one or more of the `GPUBufferUsage` flags, with multiple flags being combined with the `|` (bitwise OR) operator. In this case, you specify that you want the buffer to be used for vertex data (`GPUBufferUsage.VERTEX`) and that you also want to be able to copy data into it (`GPUBufferUsage.COPY_DST`).

The buffer object that gets returned to you is opaque—you can't (easily) inspect the data it holds. Additionally, most of its attributes are immutable—you can't resize a `GPUBuffer` after it's been created, nor can you change the usage flags. What you can change are the contents of its memory.

When the buffer is initially created, the memory it contains will be initialized to zero. There are several ways to change its contents, but the easiest is to call `device.queue.writeBuffer()` with a TypedArray that you want to copy in.

2. To copy the vertex data into the buffer's memory, add the following code:

index.html

```
device.queue.writeBuffer(vertexBuffer, /*bufferOffset=*/0, vertices);
```

Define the vertex layout

Now you have a buffer with vertex data in it, but as far as the GPU is concerned it's just a blob of bytes. You need to supply a little bit more information if you're going to draw anything with it. You need to be able to tell WebGPU more about the structure of the vertex data.

- Define the vertex data structure with a `GPUVertexBufferLayout` dictionary:

index.html

```
const vertexBufferLayout = {
  arrayStride: 8,
  attributes: [{
    format: "float32x2",
    offset: 0,
    shaderLocation: 0, // Position, see vertex shader
  }],
};
```

This can be a bit confusing at first glance, but it's relatively easy to break down.

The first thing you give is the *`arrayStride`*. This is the number of bytes the GPU needs to skip forward in the buffer when it's looking for the next vertex. Each vertex of your square is made up of two 32-bit floating point numbers. As mentioned earlier, a 32-bit float is 4 bytes, so two floats is 8 bytes.

Next is the *`attributes`* property, which is an array. Attributes are the individual pieces of information encoded into each vertex. Your vertices only contain one attribute (the vertex position), but more advanced use cases frequently have vertices with multiple attributes in them like the color of a vertex or the direction the geometry surface is pointing. That's out of scope for this codelab, though.

In your single attribute, you first define the *`format`* of the data. This comes from a list of `GPUVertexFormat` types that describe each type of vertex data that the GPU can understand. Your vertices have two 32-bit floats each, so you use the format `float32x2`. If your vertex data is instead made up of four 16-bit unsigned integers each, for example, you'd use `uint16x4` instead. See the pattern?

Next, the *`offset`* describes how many bytes into the vertex this particular attribute starts. You really only have to worry about this if your buffer has more than one attribute in it, which won't come up during this codelab.

Finally, you have the `shaderLocation`. This is an arbitrary number between 0 and 15 and must be unique for every attribute that you define. It links this attribute to a particular input in the vertex shader, which you will learn about in the next section.

Notice that though you define these values now, you're not actually passing them into the WebGPU API anywhere just yet. That's coming up, but it's easiest to think about these values at the point that you define your vertices, so you're setting them up now for use later.

Start with shaders

Now you have the data you want to render, but you still need to tell the GPU exactly how to process it. A large part of that happens with shaders.

Shaders are small programs that you write and that execute on your GPU. Each shader operates on a different *stage* of the data: *Vertex* processing, *Fragment* processing, or general *Compute*. Because they're on the GPU, they are structured more rigidly than your average JavaScript. But that structure allows them to execute very fast and, crucially, in parallel!

Shaders in WebGPU are written in a shading language called WGSL (WebGPU Shading Language). WGSL is, syntactically, a bit like Rust, with features aimed at making common types of GPU work (like vector and matrix math) easier and faster. Teaching the entirety of the shading language is well beyond the scope of this codelab, but hopefully you'll pick up some of the basics as you walk through some simple examples.

The shaders themselves get passed into WebGPU as strings.

- Create a place to enter your shader code by copying the following into your code below the `vertexBufferLayout`:

index.html

```
const cellShaderModule = device.createShaderModule({
  label: "Cell shader",
  code: `
    // Your shader code will go here
  `
});
```

To create the shaders you call `device.createShaderModule()`, to which you provide an optional `label` and WGSL `code` as a string. (Note that you use backticks here to allow multi-line strings!) Once you add some valid WGSL code, the function returns a `GPUShaderModule` object with the compiled results.

> **Note:** WGSL shaders have a reasonably strict compiler in order to enforce things like strongly typed values and eliminate undefined behavior, which is important for cross-platform compatibility. As a result, you'll definitely run into shader compiler errors as you build your own WebGPU apps! Don't stress about it too much, though. Like the rest of WebGPU, the WGSL compiler prints out very detailed messages to the developer console when things go wrong.

Define the vertex shader

Start with the vertex shader because that's where the GPU starts, too!

A vertex shader is defined as a function, and the GPU calls that function once for every vertex in your `vertexBuffer`. Since your `vertexBuffer` has six positions (vertices) in it, the function you define gets called six times. Each time it is called, a different position from the `vertexBuffer` is passed to the function as an argument, and it's the job of the vertex shader function to return a corresponding position in clip space.

It's important to understand that they won't necessarily get called in sequential order, either. Instead, GPUs excel at running shaders like these in parallel, potentially processing hundreds (or even thousands!) of vertices at the same time! This is a huge part of what's responsible for GPUs incredible speed, but it comes with limitations. In order to ensure extreme parallelization, vertex shaders cannot communicate with each other. Each shader invocation can only *see* data for a single vertex at a time, and is only able to output values for a single vertex.

In WGSL, a vertex shader function can be named whatever you want, but it must have the `@vertex` *attribute* in front of it in order to indicate which shader stage it represents. WGSL denotes functions with the `fn` keyword, uses parentheses to declare any arguments, and uses curly braces to define the scope.

1. Create an empty `@vertex` function, like this:

index.html (createShaderModule code)

```
@vertex
fn vertexMain() {

}
```

That's not valid, though, as a vertex shader must return *at least* the final position of the vertex being processed in clip space. This is always given as a 4-dimensional vector. Vectors are such a common thing to use in shaders that they're treated as first-class primitives in the language, with their own types like `vec4f` for a 4-dimensional vector. There are similar types for 2D vectors (`vec2f`) and 3D vectors (`vec3f`), as well!

2. To indicate that the value being returned is the required position, mark it with the `@builtin(position)` attribute. A `->` symbol is used to indicate that this is what the function returns.

index.html (createShaderModule code)

```
@vertex
fn vertexMain() -> @builtin(position) vec4f {

}
```

Of course, if the function has a return type, you need to actually return a value in the function body. You can construct a new `vec4f` to return, using the syntax `vec4f(x, y, z, w)`. The `x`, `y`, and `z` values are all floating point numbers that, in the return value, indicate where the vertex lies in clip space.

> **Note:** So what's `w`? The `w` value is the fourth component of a three-dimensional homogeneous vertex. Confused? That's OK! You don't have to worry about it much.
>
> In practical terms, having the `w` value there makes math with 4x4 matrices work, which is something that you do a lot of when rendering 3D graphics, but it's rare that you need to manipulate it directly. You want to just leave it as 1 for this codelab.

3. Return a static value of `(0, 0, 0, 1)`, and you technically have a valid vertex shader, although one that never displays anything since the GPU recognizes that the triangles it produces are just a single point and then discards it.

index.html (createShaderModule code)

```
@vertex
fn vertexMain() -> @builtin(position) vec4f {
  return vec4f(0, 0, 0, 1); // (X, Y, Z, W)
}
```

What you want instead is to make use of the data from the buffer that you created, and you do that by declaring an argument for your function with a `@location()` attribute and type that match what you described in the `vertexBufferLayout`. You specified a `shaderLocation` of `0`, so in your WGSL code, mark the argument with `@location(0)`. You also defined the format as a `float32x2`, which is a 2D vector, so in WGSL your argument is a `vec2f`. You can name it whatever you like, but since these represent your vertex positions, a name like *pos* seems natural.

4. Change your shader function to the following code:

index.html (createShaderModule code)

```
@vertex
fn vertexMain(@location(0) pos: vec2f) ->
  @builtin(position) vec4f {
  return vec4f(0, 0, 0, 1);
}
```

And now you need to return that position. Since the position is a 2D vector and the return type is a 4D vector, you have to alter it a bit. What you want to do is take the two components from the position argument and place them in the first two components of the return vector, leaving the last two components as `0` and `1`, respectively.

5. Return the correct position by explicitly stating which position components to use:

index.html (createShaderModule code)

```
@vertex
fn vertexMain(@location(0) pos: vec2f) ->
  @builtin(position) vec4f {
  return vec4f(pos.x, pos.y, 0, 1);
}
```

However, because these kinds of mappings are so common in shaders, you can also pass the position vector in as the first argument in a convenient shorthand and it means the same thing.

6. Rewrite the `return` statement with the following code:

index.html (createShaderModule code)

```
@vertex
fn vertexMain(@location(0) pos: vec2f) ->
  @builtin(position) vec4f {
  return vec4f(pos, 0, 1);
}
```

And that's your initial vertex shader! It's very simple, just passing out the position effectively unchanged, but it's good enough to get started.

Define the fragment shader

Next up is the fragment shader. Fragment shaders operate in a very similar way to vertex shaders, but rather than being invoked for every vertex, they're invoked for every pixel being drawn.

Fragment shaders are always called after vertex shaders. The GPU takes the output of the vertex shaders and *triangulates* it, creating triangles out of sets of three points. It then *rasterizes* each of those triangles by figuring out which pixels of the output color attachments are included in that triangle, and then calls the fragment shader once for each of those pixels. The fragment shader returns a color, typically calculated from values sent to it from the vertex shader and assets like textures, which the GPU writes to the color attachment.

Just like vertex shaders, fragment shaders are executed in a massively parallel fashion. They're a little more flexible than vertex shaders in terms of their inputs and outputs, but you can consider them to simply return one color for each pixel of each triangle.

A WGSL fragment shader function is denoted with the `@fragment` attribute and it also returns a `vec4f`. In this case, though, the vector represents a color, not a position. The return value needs to be given a `@location` attribute in order to indicate which `colorAttachment` from the `beginRenderPass` call the returned color is written to. Since you only had one attachment, the location is 0.

1. Create an empty `@fragment` function, like this:

index.html (createShaderModule code)

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

}
```

The four components of the returned vector are the red, green, blue, and alpha color values, which are interpreted in exactly the same way as the `clearValue` you set in `beginRenderPass` earlier. So `vec4f(1, 0, 0, 1)` is bright red, which seems like a decent color for your square. You're free to set it to whatever color you want, though!

2. Set the returned color vector, like this:

index.html (createShaderModule code)

```
@fragment
fn fragmentMain() -> @location(0) vec4f {
  return vec4f(1, 0, 0, 1); // (Red, Green, Blue, Alpha)
}
```

And that's a complete fragment shader! It's not a terribly interesting one; it just sets every pixel of every triangle to red, but that's sufficient for now.

Just to recap, after adding the shader code detailed above, your `createShaderModule` call now looks like this:

index.html

```
const cellShaderModule = device.createShaderModule({
  label: 'Cell shader',
  code: `
    @vertex
    fn vertexMain(@location(0) pos: vec2f) ->
      @builtin(position) vec4f {
      return vec4f(pos, 0, 1);
    }

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

Note: You can also create a separate shader module for your vertex and fragment shaders, if you want. That can be beneficial if, for example, you want to use several different fragment shaders with the same vertex shader.

Create a render pipeline

A shader module can't be used for rendering on its own. Instead, you have to use it as part of a `GPURenderPipeline`, created by calling device.createRenderPipeline(). The render pipeline controls *how* geometry is drawn, including things like which shaders are used, how to interpret data in vertex buffers, which kind of geometry should be rendered (lines, points, triangles...), and more!

The render pipeline is the most complex object in the entire API, but don't worry! Most of the values you can pass to it are optional, and you only need to provide a few to start.

- Create a render pipeline, like this:

index.html

```
const cellPipeline = device.createRenderPipeline({
  label: "Cell pipeline",
  layout: "auto",
  vertex: {
    module: cellShaderModule,
    entryPoint: "vertexMain",
    buffers: [vertexBufferLayout]
  },
  fragment: {
    module: cellShaderModule,
    entryPoint: "fragmentMain",
    targets: [{
      format: canvasFormat
    }]
  }
});
```

Every pipeline needs a `layout` that describes what types of inputs (other than vertex buffers) the pipeline needs, but you don't really have any. Fortunately, you can pass `"auto"` for now, and the pipeline builds its own layout from the shaders.

Next, you have to provide details about the `vertex` stage. The `module` is the GPUShaderModule that contains your vertex shader, and the `entryPoint` gives the name of the function in the shader code that is called for every vertex invocation. (You can have multiple `@vertex` and `@fragment` functions in a single shader module!) The *buffers* is an array of `GPUVertexBufferLayout` objects that describe how your data is packed in the vertex buffers that you use this pipeline with. Luckily, you already defined this earlier in your `vertexBufferLayout`! Here's where you pass it in.

Lastly, you have details about the `fragment` stage. This also includes a shader *module* and *entryPoint*, like the vertex stage. The last bit is to define the `targets` that this pipeline is used with. This is an array of dictionaries giving details—such as the texture `format`—of the color attachments that the pipeline outputs to. These details need to match the textures given in the `colorAttachments` of any render passes that this pipeline is used with. Your render pass uses textures from the canvas context, and uses the value you saved in `canvasFormat` for its format, so you pass the same format here.

That's not even close to all of the options that you can specify when creating a render pipeline, but it's enough for the needs of this codelab!

Note: Why is so much *stuff* packed into this one object? There are a couple of reasons for it. First is that not every combination of options is valid, and so by providing all the options in one place you can easily say at creation time whether or not the pipeline is usable. That makes drawing with the pipeline later on faster because you don't have to check as many things. (This is a big departure from WebGL, which has to validate a large number of settings with every draw call.)

The second reason is that it lets you tell the render pass a huge amount of information with a single JavaScript call when it comes time to draw. That cuts down on the amount of calls you need to make overall, which also makes your rendering more efficient. (Again, this is an area in which WebGL struggled.)

Draw the square

And with that, you now have everything that you need in order to draw your square!

1. To draw the square, jump back down to the `encoder.beginRenderPass()` and `pass.end()` pair of calls, and then add these new commands between them:

index.html

```
// After encoder.beginRenderPass()

pass.setPipeline(cellPipeline);
pass.setVertexBuffer(0, vertexBuffer);
pass.draw(vertices.length / 2); // 6 vertices

// before pass.end()
```

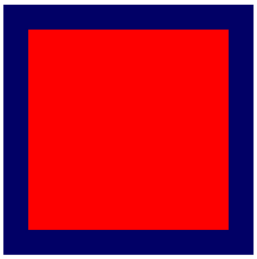
This supplies WebGPU with all the information necessary to draw your square. First, you use `setPipeline()` to indicate which pipeline should be used to draw with. This includes the shaders that are used, the layout of the vertex data, and other relevant state data.

Next, you call `setVertexBuffer()` with the buffer containing the vertices for your square. You call it with `0` because this buffer corresponds to the 0th element in the current pipeline's `vertex.buffers` definition.

this buffer corresponds to the 0th element in the current pipeline's `vertex.buffers` definition.

And last, you make the `draw()` call, which seems strangely simple after all the setup that's come before. The only thing you need to pass in is the number of vertices that it should render, which it pulls from the currently set vertex buffers and interprets with the currently set pipeline. You could just hard-code it to `6`, but calculating it from the vertices array (12 floats / 2 coordinates per vertex == 6 vertices) means that if you ever decided to replace the square with, for example, a circle, there's less to update by hand.

2. Refresh your screen and (finally) see the results of all your hard work: one big colored square.



5. Draw a grid

First, take a moment to congratulate yourself! Getting the first bits of geometry on screen is often one of the hardest steps with most GPU APIs. Everything you do from here can be done in smaller steps, making it easier to verify your progress as you go.

In this section, you learn:

- How to pass variables (called uniforms) to the shader from JavaScript.
- How to use uniforms to change the rendering behavior.
- How to use instancing to draw many different variants of the same geometry.

Define the grid

In order to render a grid, you need to know a very fundamental piece of information about it. How many cells does it contain, both in width and height? This is up to you as the developer, but to keep things a bit easier, treat the grid as a square (same width and height) and use a size that's a power of two. (That makes some of the math easier later.) You want to make it bigger eventually, but for the rest of this section, set your grid size to 4x4 because it makes it easier to demonstrate some of the math used in this section. Scale it up after!

- Define the grid size by adding a constant to the top of your JavaScript code.

index.html

```
const GRID_SIZE = 4;
```

Next, you need to update how you render your square so that you can fit `GRID_SIZE` times `GRID_SIZE` of them on the canvas. That means the square needs to be a lot smaller, and there needs to be a lot of them.

Now, one way you *could* approach this is by making your vertex buffer significantly bigger and defining `GRID_SIZE` times `GRID_SIZE` worth of squares inside it at the right size and position. The code for that wouldn't be too bad, in fact! Just a couple of for loops and a bit of math. But that's also not making the best use of the GPU and using more memory than necessary to achieve the effect. This section looks at a more GPU-friendly approach.

Create a uniform buffer

First, you need to communicate the grid size you've chosen to the shader, since it uses that to change how things display. You could just hard-code the size into the shader, but then that means that any time you want to change the grid size you have to re-create the shader and render pipeline, which is expensive. A better way is to provide the grid size to the shader as *uniforms*.

You learned earlier that a different value from the vertex buffer is passed to every invocation of a vertex shader. A uniform is a value from a buffer that is the same for every invocation. They're useful for communicating values that are common for a piece of geometry (like its position), a full frame of animation (like the current time), or even the entire lifespan of the app (like a user preference).

- Create a uniform buffer by adding the following code:

index.html

```
// Create a uniform buffer that describes the grid.
const uniformArray = new Float32Array([GRID_SIZE, GRID_SIZE]);
const uniformBuffer = device.createBuffer({
  label: "Grid Uniforms",
  size: uniformArray.byteLength,
  usage: GPUBufferUsage.UNIFORM | GPUBufferUsage.COPY_DST,
});
device.queue.writeBuffer(uniformBuffer, 0, uniformArray);
```

This should look very familiar, because it's almost exactly the same code that you used to create the vertex buffer earlier! That's because uniforms are communicated to the WebGPU API through the same GPUBuffer objects that vertices are, with the main difference being that the *`usage`* this time includes `GPUBufferUsage.UNIFORM` instead of `GPUBufferUsage.VERTEX`.

> ★ **Note:** You could use a `Uint32Array` here instead, as the grid size values aren't fractional, but in this specific case it would make the upcoming shader code changes a bit messier because you'd end up casting the values to floats in several places anyway. Just keep in mind that a float isn't the only type of data you can place in a `GPUBuffer`!

Access uniforms in a shader

- Define a uniform by adding the following code:

index.html (createShaderModule call)

```
// At the top of the `code` string in the createShaderModule() call
@group(0) @binding(0) var<uniform> grid: vec2f;

@vertex
fn vertexMain(@location(0) pos: vec2f) ->
  @builtin(position) vec4f {
  return vec4f(pos / grid, 0, 1);
}

// ...fragmentMain is unchanged
```

This defines a uniform in your shader called `grid`, which is a 2D float vector that matches the array that you just copied into the uniform buffer. It also specifies that the uniform is bound at `@group(0)` and `@binding(0)`. You'll learn what those values mean in a moment.

Then, elsewhere in the shader code, you can use the grid vector however you need. In this code you divide the vertex position by the grid vector. Since `pos` is a 2D vector and `grid` is a 2D vector, WGSL performs a component-wise division. In other words, the result is the same as saying `vec2f(pos.x / grid.x, pos.y / grid.y)`.

These types of vector operations are very common in GPU shaders since many rendering and compute techniques rely on them!

What this means in your case is that (if you used a grid size of 4) the square that you render would be one-fourth of its original size. That's perfect if you want to fit four of them to a row or column!

Create a Bind Group

Declaring the uniform in the shader doesn't connect it with the buffer that you created, though. In order to do that, you need to create and set a *bind group*.

A bind group is a collection of resources that you want to make accessible to your shader at the same time. It can include several types of buffers, like your uniform buffer, and other resources like textures and samplers that are not covered here but are common parts of WebGPU rendering techniques.

- Create a bind group with your uniform buffer by adding the following code after the creation of the uniform buffer and render pipeline:

index.html

```
const bindGroup = device.createBindGroup({
  label: "Cell renderer bind group",
  layout: cellPipeline.getBindGroupLayout(0),
  entries: [{
    binding: 0,
    resource: { buffer: uniformBuffer }
  }],
});
```

In addition to your now-standard `label`, you also need a `layout` that describes which types of resources this bind group contains. This is something that you dig into further in a future step, but for the moment you can happily ask your pipeline for the bind group layout because you created the pipeline with `layout: "auto"`. That causes the pipeline to create bind group layouts automatically from the bindings that you declared in the shader code itself. In this case, you ask it to `getBindGroupLayout(0)`, where the `0` corresponds to the `@group(0)` that you typed in the shader.

After specifying the layout, you provide an array of `entries`. Each entry is a dictionary with at least the following values:

- `binding`, which corresponds with the `@binding()` value you entered in the shader. In this case, `0`.
- `resource`, which is the actual resource that you want to expose to the variable at the specified binding index. In this case, your uniform buffer.

The function returns a `GPUBindGroup`, which is an opaque, immutable handle. You can't change the resources that a bind group points to after it's been created, though you *can* change the contents of those resources. For example, if you change the uniform buffer to contain a new grid size, that is reflected by future draw calls using this bind group.

Bind the bind group

Now that the bind group is created, you still need to tell WebGPU to use it when drawing. Fortunately this is pretty simple.

1. Hop back down to the render pass and add this new line before the `draw()` method:

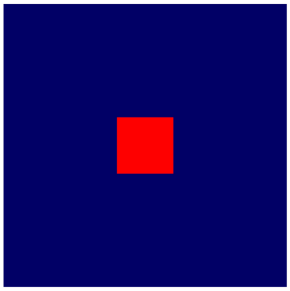
index.html

```
pass.setPipeline(cellPipeline);
pass.setVertexBuffer(0, vertexBuffer);

pass.setBindGroup(0, bindGroup); // New line!

pass.draw(vertices.length / 2);
```

The `0` passed as the first argument corresponds to the `@group(0)` in the shader code. You're saying that each `@binding` that's part of `@group(0)` uses the resources in this bind group.

And now the uniform buffer is exposed to your shader!

2. Refresh your page, and then you should see something like this:

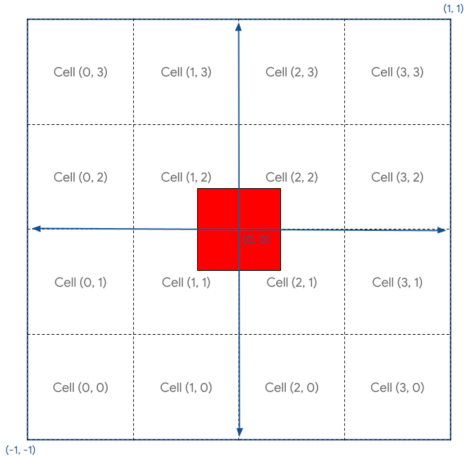


Hooray! Your square is now one-fourth the size it was before! That's not much, but it shows that your uniform is actually applied and that the shader can now access the size of your grid.

Manipulate geometry in the shader

So now that you can reference the grid size in the shader, you can start doing some work to manipulate the geometry you're rendering to fit your desired grid pattern. To do that, consider exactly what you want to achieve.

You need to conceptually divide up your canvas into individual cells. In order to keep the convention that the X axis increases as you move right and the Y axis increases as you move up, say that the first cell is in the bottom left corner of the canvas. That gives you a layout that looks like this, with your current square geometry in the middle:



Your challenge is to find a method in the shader that lets you position the square geometry in any of those cells given the cell coordinates.

First, you can see that your square isn't nicely aligned with any of the cells because it was defined to surround the center of the canvas. You'd want to have the square shifted by half a cell so that it would line up nicely inside them.

One way you could fix this is to update the square's vertex buffer. By shifting the vertices so that the bottom-right corner is at, for example, (0.1, 0.1) instead of (-0.8, -0.8), you'd move this square to line up with the cell boundaries more nicely. But, since you have full control over how the vertices are processed in your shader, it's just as easy to simply nudge them into place using the shader code!

1. Alter the vertex shader module with the following code:

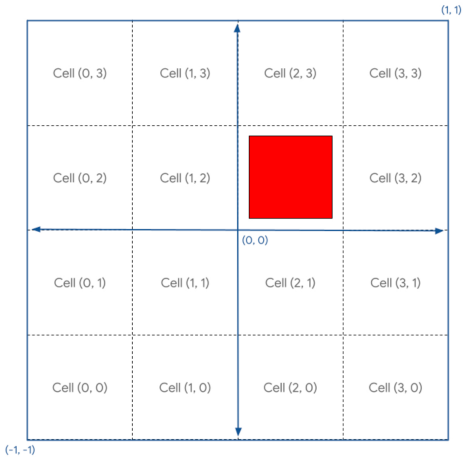
index.html (createShaderModule call)

```
@group(0) @binding(0) var<uniform> grid: vec2f;

@vertex
fn vertexMain(@location(0) pos: vec2f) ->
  @builtin(position) vec4f {

  // Add 1 to the position before dividing by the grid size.
  let gridPos = (pos + 1) / grid;

  return vec4f(gridPos, 0, 1);
}
```

Caution: For the sake of making it easier to read, the position calculation is broken out into its own variable. Notice that it's declared using `let`, but be aware that `let` in WGSL has a different meaning than `let` in JavaScript! In WGSL, `let` behaves more like JavaScript's `const` in that it indicates that the value of the variable won't change after assignment. If you do need to change a variable after declaring it in WGSL, use `var` instead.

Note: Since `pos` is a `vec2f` here, adding `1` does a component-wise addition. It's the same as saying `pos + vec2f(1, 1)`. The same works for subtraction, multiplication, and division!

This moves every vertex up and to the left by one (which, remember, is half of the clip space) *before* dividing it by the grid size. The result is a nicely grid-aligned square just off of the origin.



Next, because your canvas's coordinate system places (0, 0) in the center and (-1, -1) in the lower left, and you want (0, 0) to be in the lower left, you need to translate your geometry's position by (-1, -1) *after* dividing by the grid size in order to

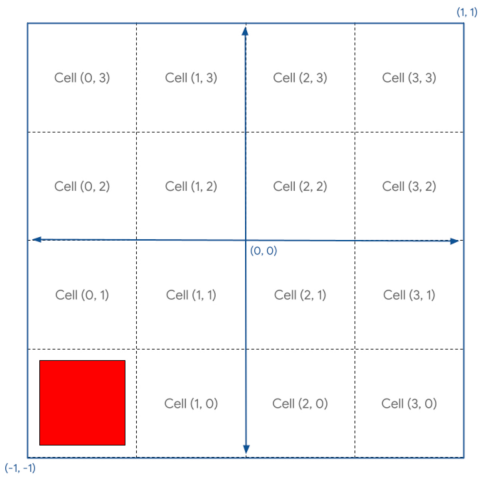
move it into that corner.

2. Translate your geometry's position, like this:

index.html (createShaderModule call)

```
@group(0) @binding(0) var<uniform> grid: vec2f;

@vertex
fn vertexMain(@location(0) pos: vec2f) ->
  @builtin(position) vec4f {

  // Subtract 1 after dividing by the grid size.
  let gridPos = (pos + 1) / grid - 1;

  return vec4f(gridPos, 0, 1);
}
```

And now your square is nicely positioned in cell (0, 0)!



What if you want to place it in a different cell? Figure that out by declaring a `cell` vector in your shader and populating it with a static value like `let cell = vec2f(1, 1)`.

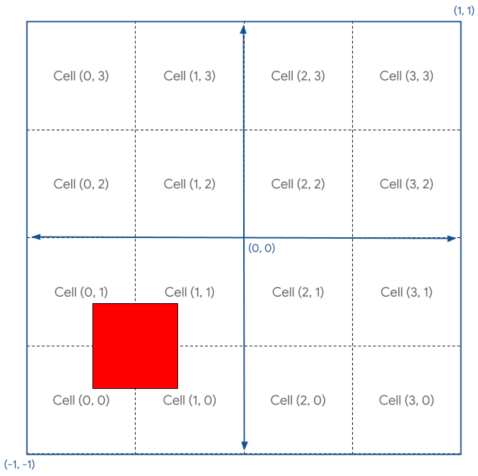
If you add that to the `gridPos`, it undoes the `- 1` in the algorithm, so that's not what you want. Instead, you want to move the square only by one grid unit (one-fourth of the canvas) for each cell. Sounds like you need to do another divide by `grid`!

3. Change your grid positioning, like this:

index.html (createShaderModule call)

```
@group(0) @binding(0) var<uniform> grid: vec2f;

@vertex
fn vertexMain(@location(0) pos: vec2f) ->
  @builtin(position) vec4f {

  let cell = vec2f(1, 1); // Cell(1,1) in the image above
  let cellOffset = cell / grid; // Compute the offset to cell
  let gridPos = (pos + 1) / grid - 1 + cellOffset; // Add it here!

  return vec4f(gridPos, 0, 1);
}
```

If you refresh now, you see the following:



Hm. Not quite what you wanted.

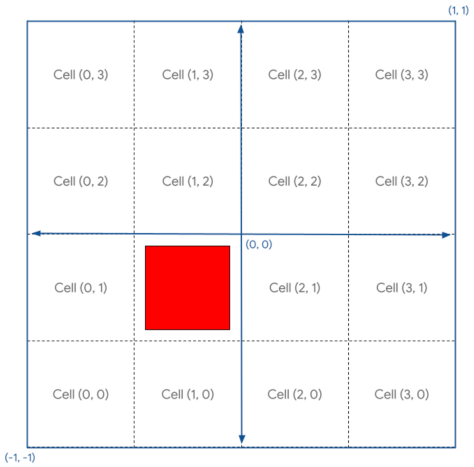
The reason for this is that since the canvas coordinates go from -1 to +1, it's *actually 2 units across*. That means if you want to move a vertex one-fourth of the canvas over, you have to move it 0.5 units. This is an easy mistake to make when reasoning with GPU coordinates! Fortunately, the fix is just as easy.

4. Multiply your offset by 2, like this:

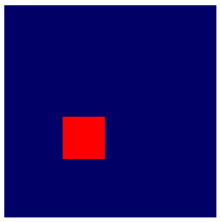
index.html (createShaderModule call)

```
@group(0) @binding(0) var<uniform> grid: vec2f;

@vertex
fn vertexMain(@location(0) pos: vec2f) ->
  @builtin(position) vec4f {

  let cell = vec2f(1, 1);
  let cellOffset = cell / grid * 2; // Updated
  let gridPos = (pos + 1) / grid - 1 + cellOffset;

  return vec4f(gridPos, 0, 1);
}
```

And this gives you exactly what you want.



The screenshot looks like this:



Furthermore, you can now set `cell` to any value within the grid bounds, and then refresh to see the square render in the desired location.

Draw instances

Now that you can place the square where you want it with a bit of math, the next step is to render one square in each cell of the grid.

One way you could approach it is to write cell coordinates to a uniform buffer, then call *draw* once for each square in the grid, updating the uniform every time. That would be very slow, however, since the GPU has to wait for the new coordinate to be written by JavaScript every time. One of the keys to getting good performance out of the GPU is to minimize the time it spends waiting on other parts of the system!

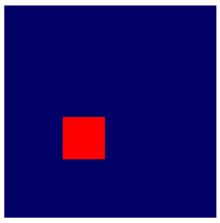
Instead, you can use a technique called instancing. Instancing is a way to tell the GPU to draw multiple copies of the same geometry with a single call to `draw`, which is much faster than calling `draw` once for every copy. Each copy of the geometry is referred to as an *instance*.

1. To tell the GPU that you want enough instances of your square to fill the grid, add one argument to your existing draw call:

index.html

```
pass.draw(vertices.length / 2, GRID_SIZE * GRID_SIZE);
```

This tells the system that you want it to draw the six (`vertices.length / 2`) vertices of your square 16 (`GRID_SIZE * GRID_SIZE`) times. But if you refresh the page, you still see the following:



Why? Well, it's because you draw all 16 of those squares in the same spot. Your need to have some additional logic in the shader that repositions the geometry on a per-instance basis.

In the shader, in addition to the vertex attributes like `pos` that come from your vertex buffer, you can also access what are known as WGSL's *built-in values*. These are values that are calculated by WebGPU, and one such value is the `instance_index`. The `instance_index` is an unsigned 32-bit number from `0` to `number of instances - 1` that you can use as part of your shader logic. Its value is the same for every vertex processed that's part of the same instance. That means your vertex shader gets called six times with an `instance_index` of `0`, once for each position in your vertex buffer. Then six more times with an `instance_index` of `1`, then six more with `instance_index` of `2`, and so on.

To see this in action, you have to add the `instance_index` built-in to your shader inputs. Do this in the same way as the position, but instead of tagging it with a `@location` attribute, use `@builtin(instance_index)`, and then name the argument whatever you want. (You can call it `instance` to match the example code.) Then use it as part of the shader logic!

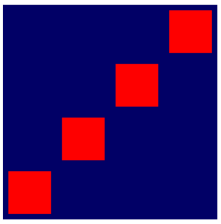
2. Use `instance` in place of the cell coordinates:

index.html

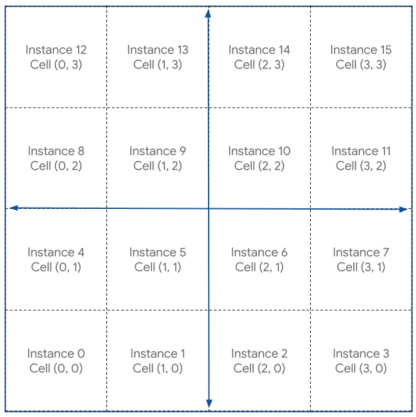
```
@group(0) @binding(0) var<uniform> grid: vec2f;

@vertex
fn vertexMain(@location(0) pos: vec2f,
              @builtin(instance_index) instance: u32) ->
  @builtin(position) vec4f {

  let i = f32(instance); // Save the instance_index as a float
  let cell = vec2f(i, i);
  let cellOffset = cell / grid * 2; // Updated
  let gridPos = (pos + 1) / grid - 1 + cellOffset;

  return vec4f(gridPos, 0, 1);
}
```

Caution: This is an area where WGSL's strong typing may trip up developers who are more familiar with JavaScript than other strongly typed languages. The **`instance_index`** is a **`u32`**, or unsigned 32-bit integer. But you want **`cell`** to be a vector of floating point values because that's how the position math is done. As a result, you have to explicitly tell the shader to treat **`instance`** as a float when creating the **`cell`** vector by wrapping it in a **`f32()`** function. This is known as *casting* the type.

If you refresh now you see that you do indeed have more than one square! But you can't see all 16 of them.



That's because the cell coordinates you generate are (0, 0), (1, 1), (2, 2)... all the way to (15, 15), but only the first four of those fit on the canvas. To make the grid that you want, you need to transform the `instance_index` such that each index maps to a unique cell within your grid, like this:



The math for that is reasonably straightforward. For each cell's X value, you want the modulo of the `instance_index` and the grid width, which you can perform in WGSL with the `%` operator. And for each cell's Y value you want the `instance_index` divided by the grid width, discarding any fractional remainder. You can do that with WGSL's `floor()` function.

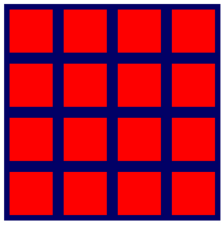
3. Change the calculations, like this:

index.html

```
@group(0) @binding(0) var<uniform> grid: vec2f;

@vertex
fn vertexMain(@location(0) pos: vec2f,
              @builtin(instance_index) instance: u32) ->
  @builtin(position) vec4f {

  let i = f32(instance);
  // Compute the cell coordinate from the instance_index
  let cell = vec2f(i % grid.x, floor(i / grid.x));

  let cellOffset = cell / grid * 2;
  let gridPos = (pos + 1) / grid - 1 + cellOffset;

  return vec4f(gridPos, 0, 1);
}
```

After making that update to the code you have the long-awaited grid of squares at last!

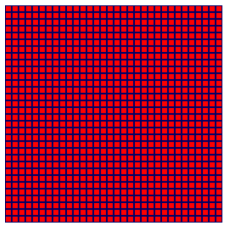
After making that update to the code you have the long awaited grid of squares at last!



4. And now that it's working, go back and crank up the grid size!

index.html

```
const GRID_SIZE = 32;
```



Tada! You can actually make this grid really, *really* big now and your average GPU handles it just fine. You'll stop seeing the individual squares long before you run into any GPU performance bottlenecks.

> **Note:** You might be thinking, "That could have all been done with one line of Canvas 2D code in a nested loop!" And, well... yeah! It could have! And if that level of rendering is all you're interested in then you're probably better off sticking with a simpler API like Canvas 2D. This codelab is focused on introducing core WebGPU concepts, and pushing the limits of the API's rendering capabilities is way, *way* out of scope.
>
> It's worth noting that if you have a use case where you want to use Canvas 2D for more simplistic rendering needs like this but still want to take advantage of WebGPU's compute capabilities, you can use the APIs cooperatively! It requires more data copies than if everything is done in WebGPU. See WebGPU — All of the cores, none of the canvas as an example of using WebGPU and Canvas 2D together.

6. Extra credit: make it more colorful!

At this point, you can easily skip to the next section since you've laid the groundwork for the rest of the codelab. But while the grid of squares all sharing the same color is serviceable, it's not exactly exciting, is it? Fortunately you can make things a bit brighter with a little more math and shader code!

Use structs in shaders

Until now, you've passed one piece of data out of the vertex shader: the transformed position. But you can actually return a lot more data from the vertex shader and then use it in the fragment shader!

The only way to pass data out of the vertex shader is by returning it. A vertex shader is always required to return a position, so if you want to return any other data along with it, you need to place it in a struct. Structs in WGSL are named object types that contain one or more named properties. The properties can be marked up with attributes like `@builtin` and `@location` too. You declare them outside of any functions, and then you can pass instances of them in and out of functions, as needed. For example, consider your current vertex shader:

index.html (createShaderModule call)

```
@group(0) @binding(0) var<uniform> grid: vec2f;

@vertex
fn vertexMain(@location(0) pos: vec2f,
              @builtin(instance_index) instance: u32) ->
  @builtin(position) vec4f {

  let i = f32(instance);
  let cell = vec2f(i % grid.x, floor(i / grid.x));
  let cellOffset = cell / grid * 2;
  let gridPos = (pos + 1) / grid - 1 + cellOffset;

  return  vec4f(gridPos, 0, 1);
}
```

- Express the same thing using structs for the function input and output:

index.html (createShaderModule call)

```
struct VertexInput {
  @location(0) pos: vec2f,
  @builtin(instance_index) instance: u32,
};

struct VertexOutput {
  @builtin(position) pos: vec4f,
};

@group(0) @binding(0) var<uniform> grid: vec2f;

@vertex
fn vertexMain(input: VertexInput) -> VertexOutput  {
  let i = f32(input.instance);
  let cell = vec2f(i % grid.x, floor(i / grid.x));
  let cellOffset = cell / grid * 2;
  let gridPos = (input.pos + 1) / grid - 1 + cellOffset;

  var output: VertexOutput;
  output.pos = vec4f(gridPos, 0, 1);
```

```
  return output;
}
```

Notice that this requires you to refer to the input position and instance index with `input`, and the struct that you return first needs to be declared as a variable and have its individual properties set. In this case, it doesn't make too much difference, and in fact makes the shader function a bit longer, but as your shaders grow more complex, using structs can be a great way to help organize your data.

Pass data between the vertex and fragment functions

As a reminder, your `@fragment` function is as simple as possible:

index.html (createShaderModule call)

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

You are not taking any inputs, and you are passing out a solid color (red) as your output. If the shader knew more about the geometry that it's coloring, though, you could use that extra data to make things a bit more interesting. For instance, what if you want to change the color of each square based on its cell coordinate? The `@vertex` stage knows which cell is being rendered; you just need to pass it along to the `@fragment` stage.

To pass any data between the vertex and fragment stages, you need to include it in an output struct with a `@location` of our choice. Since you want to pass the cell coordinate, add it to the `VertexOutput` struct from earlier, and then set it in the `@vertex` function before you return.

1. Change the return value of your vertex shader, like this:

index.html (createShaderModule call)

```
struct VertexInput {
  @location(0) pos: vec2f,
  @builtin(instance_index) instance: u32,
};

struct VertexOutput {
  @builtin(position) pos: vec4f,
  @location(0) cell: vec2f, // New line!
};

@group(0) @binding(0) var<uniform> grid: vec2f;

@vertex
fn vertexMain(input: VertexInput) -> VertexOutput  {
  let i = f32(input.instance);
  let cell = vec2f(i % grid.x, floor(i / grid.x));
  let cellOffset = cell / grid * 2;
  let gridPos = (input.pos + 1) / grid - 1 + cellOffset;

  var output: VertexOutput;
  output.pos = vec4f(gridPos, 0, 1);
  output.cell = cell; // New line!
  return output;
}
```

2. In the `@fragment` function, receive the value by adding an argument with the same `@location`. (The names don't have to match, but it's easier to keep track of things if they do!)

index.html (createShaderModule call)

```
@fragment
fn fragmentMain(@location(0) cell: vec2f) -> @location(0) vec4f {
  // Remember, fragment return values are (Red, Green, Blue, Alpha)
  // and since cell is a 2D vector, this is equivalent to:
  // (Red = cell.x, Green = cell.y, Blue = 0, Alpha = 1)
  return vec4f(cell, 0, 1);
}
```

3. Alternatively, you could use a struct instead:

index.html (createShaderModule call)

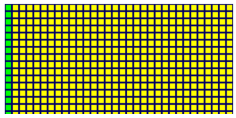
```
struct FragInput {
  @location(0) cell: vec2f,
};

@fragment
fn fragmentMain(input: FragInput) -> @location(0) vec4f {
  return vec4f(input.cell, 0, 1);
}
```

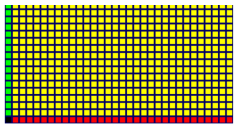
4. Another alternative**,** since in your code both of these functions are defined in the same shader module, is to reuse the `@vertex` stage's output struct! This makes passing values easy because the names and locations are naturally consistent.

index.html (createShaderModule call)

```
@fragment
fn fragmentMain(input: VertexOutput) -> @location(0) vec4f {
  return vec4f(input.cell, 0, 1);
}
```

No matter which pattern you chose, the result is that you have access to the cell number in the `@fragment` function and are able to use it in order to influence the color. With any of the above code, the output looks like this:





There are definitely more colors now, but it's not exactly nice looking. You might wonder why only the left and bottom rows are different. That's because the color values that you return from the `@fragment` function expect each channel to be in the range of 0 to 1, and any values outside of that range are clamped to it. Your cell values, on the other hand, range from 0 to 32 along each axis. So what you see here is that the first row and column immediately hit that full 1 value on either the red or green color channel, and every cell after that clamps to the same value.

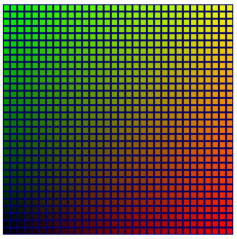
If you want a smoother transition between colors, you need to return a fractional value for each color channel, ideally starting at zero and ending at one along each axis, which means yet another divide by `grid`!

5. Change the fragment shader, like this:

index.html (createShaderModule call)

```
@fragment
fn fragmentMain(input: VertexOutput) -> @location(0) vec4f {
  return vec4f(input.cell/grid, 0, 1);
}
```

Refresh the page, and you can see that the new code *does* give you a much nicer gradient of colors across the entire grid.



While that's certainly an improvement, there's now an unfortunate dark corner in the lower left, where the grid becomes black. When you start doing the Game of Life simulation, a hard-to-see section of the grid will obscure what's going on. It would be nice to brighten that up.

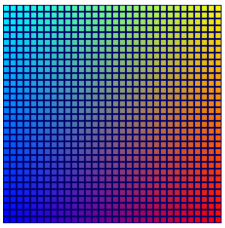
Fortunately, you have a whole unused color channel—blue—that you can use. The effect that you ideally want is to have the blue be brightest where the other colors are darkest, and then fade out as the other colors grow in intensity. The easiest way to do that is to have the channel *start* at 1 and subtract one of the cell values. It can be either `c.x` or `c.y`. Try both, and then pick the one you prefer!

6. Add brighter colors to the fragment shader, like this:

createShaderModule call

```
@fragment
fn fragmentMain(input: VertexOutput) -> @location(0) vec4f {
  let c = input.cell / grid;
  return vec4f(c, 1-c.x, 1);
}
```

The result looks quite nice!



This isn't a critical step! But because it looks better, it's included it in the corresponding checkpoint source file, and the rest of the screenshots in this codelab reflect this more colorful grid.

7. Manage cell state

Next, you need to control which cells on the grid render, based on some state that's stored on the GPU. This is important for the final simulation!

All you need is an on-off signal for each cell, so any options that allow you to store a large array of nearly any value type works. You might think that this is another use case for uniform buffers! While you *could* make that work, it's more difficult because uniform buffers are limited in size, can't support dynamically sized arrays (you have to specify the array size in the shader), and can't be written to by compute shaders. That last item is the most problematic, since you want to do the Game of Life simulation on the GPU in a compute shader.

Fortunately, there's another buffer option that avoids all of those limitations.

Create a storage buffer

Storage buffers are general-use buffers that can be read and written to in compute shaders, and read in vertex shaders. They can be *very* large, and they don't need a specific declared size in a shader, which makes them much more like general memory. That's what you use to store the cell state.

> ★ **Note:** If storage buffers are so much more flexible, why bother with uniform buffers at all? It actually depends on your GPU hardware! There's a good chance that uniform buffers are given special treatment by your GPU in order to allow them to update and be read faster than a storage buffer, so for smaller amounts of data that have the potential to update frequently (like model, view, and projection matrices in 3D applications), uniforms are typically the safer choice for better performance.

1. To create a storage buffer for your cell state, use what—by now—is probably starting to be a familiar-looking

snippet of buffer creation code:

index.html

```
// Create an array representing the active state of each cell.
const cellStateArray = new Uint32Array(GRID_SIZE * GRID_SIZE);

// Create a storage buffer to hold the cell state.
const cellStateStorage = device.createBuffer({
  label: "Cell State",
  size: cellStateArray.byteLength,
  usage: GPUBufferUsage.STORAGE | GPUBufferUsage.COPY_DST,
});
```

Just like with your vertex and uniform buffers, call `device.createBuffer()` with the appropriate size, and then make sure to specify a usage of `GPUBufferUsage.STORAGE` this time.

You can populate the buffer the same way as before by filling the TypedArray of the same size with values and then calling `device.queue.writeBuffer()`. Because you want to see the effect of your buffer on the grid, start by filling it with something predictable.

2. Activate every third cell with the following code:

index.html

```
// Mark every third cell of the grid as active.
for (let i = 0; i < cellStateArray.length; i += 3) {
  cellStateArray[i] = 1;
}
device.queue.writeBuffer(cellStateStorage, 0, cellStateArray);
```

Read the storage buffer in the shader

Next, update your shader to look at the contents of the storage buffer before you render the grid. This looks very similar to how uniforms were added previously.

1. Update your shader with the following code:

index.html

```
@group(0) @binding(0) var<uniform> grid: vec2f;
@group(0) @binding(1) var<storage> cellState: array<u32>; // New!
```

First, you add the binding point, which tucks right underneath the grid uniform. You want to keep the same `@group` as the `grid` uniform, but the `@binding` number needs to be different. The `var` type is `storage`, in order to reflect the different type of buffer, and rather than a single vector, the type that you give for the `cellState` is an array of `u32` values, in order to match the `Uint32Array` in JavaScript.

> **Note:** Why 32 bit ints? Doesn't that seem wasteful if all you need are booleans? Well, yeah! It is! But also, the GPU only cleanly exposes data of a few types because it can work with them fast. You can't specify that you're exposing an array of bytes, for example, because it wouldn't work well with certain assumptions GPUs make about data alignment.
>
> Now you *could* save space and use bitmasking tricks such as storing the active state for 32 different cells in each value of that array. There's actually a long history of developers finding clever ways to pack the data they need into GPU-approved types like that! But that would make the example code far more complex. And the truth is that for a use case as relatively simple as this one, you don't need to worry about the memory impact too much.

Next, in the body of your `@vertex` function, query the cell's state. Because the state is stored in a flat array in the storage buffer, you can use the `instance_index` in order to look up the value for the current cell!

How do you turn off a cell if the state says that it's inactive? Well, since the active and inactive states that you get from the array are 1 or 0, you can scale the geometry by the active state! Scaling it by 1 leaves the geometry alone, and scaling it by 0 makes the geometry collapse into a single point, which the GPU then discards.

2. Update your shader code to scale the position by the cell's active state. The state value must be cast the state to a `f32` in order to satisfy WGSL's type safety requirements:

index.html

```
@vertex
fn vertexMain(@location(0) pos: vec2f,
              @builtin(instance_index) instance: u32) -> VertexOutput {
  let i = f32(instance);
  let cell = vec2f(i % grid.x, floor(i / grid.x));
  let state = f32(cellState[instance]); // New line!

  let cellOffset = cell / grid * 2;
  // New: Scale the position by the cell's active state.
  let gridPos = (pos*state+1) / grid - 1 + cellOffset;

  var output: VertexOutput;
  output.pos = vec4f(gridPos, 0, 1);
  output.cell = cell;
  return output;
}
```

Add the storage buffer to the bind group

Before you can see the cell state take effect, add the storage buffer to a bind group. Because it's part of the same `@group` as the uniform buffer, add it to the same bind group in the JavaScript code, as well.

- Add the storage buffer, like this:

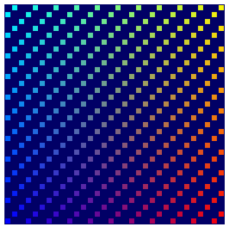
index.html

```
const bindGroup = device.createBindGroup({
  label: "Cell renderer bind group",
  layout: cellPipeline.getBindGroupLayout(0),
  entries: [{
    binding: 0,
    resource: { buffer: uniformBuffer }
  },
  // New entry!
  {
    binding: 1,
    resource: { buffer: cellStateStorage }
```

```
    }],
  });
```

Make sure that the `binding` of the new entry matches the `@binding()` of the corresponding value in the shader!

With that in place, you should be able to refresh and see the pattern appear in the grid.



Use the ping-pong buffer pattern

Most simulations like the one you're building typically use at least *two* copies of their state. On each step of the simulation, they read from one copy of the state and write to the other. Then, on the next step, flip it and read from the state they wrote to previously. This is commonly referred to as a ping pong pattern because the most up-to-date version of the state bounces back and forth between state copies each step.

Why is that necessary? Look at a simplified example: imagine that you're writing a very simple simulation in which you move any active blocks right by one cell each step. To keep things easy to understand, you define your data and simulation in JavaScript:

```
// Example simulation. Don't copy into the project!
const state = [1, 0, 0, 0, 0, 0, 0, 0];

function simulate() {
  for (let i = 0; i < state.length-1; ++i) {
    if (state[i] == 1) {
      state[i] = 0;
      state[i+1] = 1;
    }
  }
}

simulate(); // Run the simulation for one step.
```

But if you run that code, the active cell moves all the way to the end of the array in one step! Why? Because you keep updating the state in-place, so you move the active cell right, and then you look at the next cell and... hey! It's active! Better move it to the right again. The fact that you change the data at the same time that you observe it corrupts the results.

By using the ping pong pattern, you ensure that you always perform the next step of the simulation using *only* the results of the last step.

```
// Example simulation. Don't copy into the project!
const stateA = [1, 0, 0, 0, 0, 0, 0, 0];
const stateB = [0, 0, 0, 0, 0, 0, 0, 0];

function simulate(in, out) {
  out[0] = 0;
  for (let i = 1; i < in.length; ++i) {
    out[i] = in[i-1];
  }
}

// Run the simulation for two step.
simulate(stateA, stateB);
simulate(stateB, stateA);
```

1. Use this pattern in your own code by updating your storage buffer allocation in order to create two identical buffers:

index.html

```
// Create two storage buffers to hold the cell state.
const cellStateStorage = [
  device.createBuffer({
    label: "Cell State A",
    size: cellStateArray.byteLength,
    usage: GPUBufferUsage.STORAGE | GPUBufferUsage.COPY_DST,
  }),
  device.createBuffer({
    label: "Cell State B",
     size: cellStateArray.byteLength,
     usage: GPUBufferUsage.STORAGE | GPUBufferUsage.COPY_DST,
  })
];
```

2. To help visualize the difference between the two buffers, fill them with different data:

index.html

```
// Mark every third cell of the first grid as active.
for (let i = 0; i < cellStateArray.length; i+=3) {
  cellStateArray[i] = 1;
}
device.queue.writeBuffer(cellStateStorage[0], 0, cellStateArray);

// Mark every other cell of the second grid as active.
for (let i = 0; i < cellStateArray.length; i++) {
  cellStateArray[i] = i % 2;
}
device.queue.writeBuffer(cellStateStorage[1], 0, cellStateArray);
```

> ★ **Note:** As the above code snippet shows, once you call **`writeBuffer()`**, you don't have to preserve the contents of the source TypedArray any more. At that point, the contents have been copied and the GPU buffer is guaranteed to receive the data as it was at the time the call is made. This allows you to reuse the JavaScript object for the next upload, which saves on memory!

3. To show the different storage buffers in your rendering, update your bind groups to have two different variants, as well:

well.

index.html

```
const bindGroups = [
  device.createBindGroup({
    label: "Cell renderer bind group A",
    layout: cellPipeline.getBindGroupLayout(0),
    entries: [{
      binding: 0,
      resource: { buffer: uniformBuffer }
    }, {
      binding: 1,
      resource: { buffer: cellStateStorage[0] }
    }],
  }),
   device.createBindGroup({
    label: "Cell renderer bind group B",
    layout: cellPipeline.getBindGroupLayout(0),
    entries: [{
      binding: 0,
      resource: { buffer: uniformBuffer }
    }, {
      binding: 1,
      resource: { buffer: cellStateStorage[1] }
    }],
  })
];
```

Set up a render loop

So far, you've only done one draw per page refresh, but now you want to show data updating over time. To do that you need a simple render loop.

A render loop is an endlessly repeating loop that draws your content to the canvas at a certain interval. Many games and other content that want to animate smoothly use the `requestAnimationFrame()` function to schedule callbacks at the same rate that the screen refreshes (60 times every second).

This app can use that, as well, but in this case, you probably want updates to happen in longer steps so that you can more easily follow what the simulation is doing. Manage the loop yourself instead so that you can control the rate at which your simulation updates.

1. First, pick a rate for our simulation to update at (200ms is good, but you can go slower or faster if you like), and then keep track of how many steps of simulation have been completed.

index.html

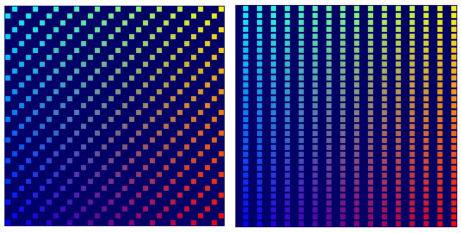
```
const UPDATE_INTERVAL = 200; // Update every 200ms (5 times/sec)
let step = 0; // Track how many simulation steps have been run
```

2. Then move all of the code you currently use for rendering into a new function. Schedule that function to repeat at your desired interval with `setInterval()`. Make sure that the function also updates the step count, and use that to pick which of the two bind groups to bind.

index.html

```
// Move all of our rendering code into a function
function updateGrid() {
  step++; // Increment the step count

  // Start a render pass
  const encoder = device.createCommandEncoder();
  const pass = encoder.beginRenderPass({
    colorAttachments: [{
      view: context.getCurrentTexture().createView(),
      loadOp: "clear",
      clearValue: { r: 0, g: 0, b: 0.4, a: 1.0 },
      storeOp: "store",
    }]
  });

  // Draw the grid.
  pass.setPipeline(cellPipeline);
  pass.setBindGroup(0, bindGroups[step % 2]); // Updated!
  pass.setVertexBuffer(0, vertexBuffer);
  pass.draw(vertices.length / 2, GRID_SIZE * GRID_SIZE);

  // End the render pass and submit the command buffer
  pass.end();
  device.queue.submit([encoder.finish()]);
}

// Schedule updateGrid() to run repeatedly
setInterval(updateGrid, UPDATE_INTERVAL);
```

And now when you run the app you see that the canvas flips back and forth between showing the two state buffers you created.



With that, you're pretty much done with the rendering side of things! You're all set to display the output of the Game of Life simulation you build in the next step, where you finally start using compute shaders.

Obviously there is so much more to WebGPU's rendering capabilities than the tiny slice that you explored here, but the rest is beyond the scope of this codelab. Hopefully, it gives you enough of a taste of how WebGPU's rendering works, though, that it helps make exploring more advanced techniques like 3D rendering easier to grasp.

8. Run the simulation

Now, for the last major piece of the puzzle, performing the Game of Life simulation in a compute shader!

Now, for the last major piece of the puzzle: performing the Game of Life simulation in a compute shader!

Use compute shaders, at last!

You've learned abstractly about compute shaders throughout this codelab, but what exactly are they?

A compute shader is similar to vertex and fragment shaders in that they are designed to run with extreme parallelism on the GPU, but unlike the other two shader stages, they don't have a specific set of inputs and outputs. You are reading and writing data exclusively from sources you choose, like storage buffers. This means that instead of executing once for each vertex, instance, or pixel, you have to tell it how many invocations of the shader function you want. Then, when you run the shader, you are told which invocation is being processed, and you can decide what data you are going to access and which operations you are going to perform from there.

Compute shaders must be created in a shader module, just like vertex and fragment shaders, so add that to your code to get started. As you might guess, given the structure of the other shaders that you've implemented, the main function for your compute shader needs to be marked with the `@compute` attribute.

1. Create a compute shader with the following code:

index.html

```
// Create the compute shader that will process the simulation.
const simulationShaderModule = device.createShaderModule({
  label: "Game of Life simulation shader",
  code: `
    @compute
    fn computeMain() {

    }`
});
```

Because GPUs are used frequently for 3D graphics, compute shaders are structured such that you can request that the shader be invoked a specific number of times along an X, Y, and Z axis. This lets you very easily dispatch work that conforms to a 2D or 3D grid, which is great for your use case! You want to call this shader `GRID_SIZE` times `GRID_SIZE` times, once for each cell of your simulation.

Due to the nature of GPU hardware architecture, this grid is divided into *workgroups*. A workgroup has an X, Y, and Z size, and although the sizes can be 1 each, there are often performance benefits to making your workgroups a bit bigger. For your shader, choose a somewhat arbitrary workgroup size of 8 times 8. This is useful to keep track of in your JavaScript code.

2. Define a constant for your workgroup size, like this:

index.html

```
const WORKGROUP_SIZE = 8;
```

You also need to add the workgroup size to the shader function itself, which you do using JavaScript's template literals so that you can easily use the constant you just defined.

3. Add the workgroup size to the shader function, like this:

index.html (Compute createShaderModule call)

```
@compute
@workgroup_size(${WORKGROUP_SIZE}, ${WORKGROUP_SIZE}) // New line
fn computeMain() {

}
```

This tells the shader that work done with this function is done in (8 x 8 x 1) groups. (Any axis you leave off defaults to 1, although you have to at least specify the X axis.)

> **Note:** For more advanced uses of compute shaders, the workgroup size becomes more important. Shader invocations within a single workgroup are allowed to share faster memory and use certain types of synchronization primitives. You don't need any of that, though, since your shader executions are fully independent.
>
> You could make the workgroup size (1 x 1 x 1), and it would still work correctly, but that also restricts how well the GPU can run the shader in parallel. Picking something bigger helps the GPU divide the work better.
>
> There is a theoretical ideal workgroup size for every GPU, but it's dependent on architectural details that WebGPU doesn't expose, so usually you want to pick a number driven by the requirements of the shader. Lacking that, given the wide range of hardware that WebGPU content may run on, 64 is a good number that's unlikely to exceed any hardware limits but still handles large enough batches to be reasonably efficient. (8 x 8 == 64, so your workgroup size follows this advice.)

As with the other shader stages, there's a variety of `@builtin` values that you can accept as input into your compute shader function in order to tell you which invocation you're on and decide what work you need to do.

4. Add a `@builtin` value, like this:

index.html (Compute createShaderModule call)

```
@compute @workgroup_size(${WORKGROUP_SIZE}, ${WORKGROUP_SIZE})
fn computeMain(@builtin(global_invocation_id) cell: vec3u) {

}
```

You pass in the `global_invocation_id` builtin, which is a three-dimensional vector of unsigned integers that tells you where in the grid of shader invocations you are. You run this shader once for each cell in your grid. You get numbers like `(0, 0, 0)`, `(1, 0, 0)`, `(1, 1, 0)` ... all the way to `(31, 31, 0)`, which means that you can treat it as the cell index you're going to operate on!

Compute shaders can also use uniforms, which you use just like in the vertex and fragment shaders.

5. Use a uniform with your compute shader to tell you the grid size, like this:

index.html (Compute createShaderModule call)

```
@group(0) @binding(0) var<uniform> grid: vec2f; // New line

@compute @workgroup_size(${WORKGROUP_SIZE}, ${WORKGROUP_SIZE})
fn computeMain(@builtin(global_invocation_id) cell: vec3u) {

}
```

Just like in the vertex shader, you also expose the cell state as a storage buffer. But in this case, you need two of them! Because compute shaders don't have a required output, like a vertex position or fragment color, writing values to a

Because compute shaders don't have a required output, like a vertex position or fragment color, writing values to a storage buffer or texture is the only way to get results out of a compute shader. Use the ping-pong method that you learned earlier; you have one storage buffer that feeds in the current state of the grid and one that you write out the new state of the grid to.

6. Expose the cell input and output state as storage buffers, like this:

index.html (Compute createShaderModule call)

```
@group(0) @binding(0) var<uniform> grid: vec2f;

// New lines
@group(0) @binding(1) var<storage> cellStateIn: array<u32>;
@group(0) @binding(2) var<storage, read_write> cellStateOut: array<u32>;

@compute @workgroup_size(${WORKGROUP_SIZE}, ${WORKGROUP_SIZE})
fn computeMain(@builtin(global_invocation_id) cell: vec3u) {

}
```

Note that the first storage buffer is declared with `var<storage>`, which makes it read-only, but the second storage buffer is declared with `var<storage, read_write>`. This allows you to both read and write to the buffer, using that buffer as the output for your compute shader. (There is no write-only storage mode in WebGPU).

Next, you need to have a way to map your cell index into the linear storage array. This is basically the opposite of what you did in the vertex shader, where you took the linear `instance_index` and mapped it to a 2D grid cell. (As a reminder, your algorithm for that was `vec2f(i % grid.x, floor(i / grid.x))`.)

7. Write a function to go in the other direction. It takes the cell's Y value, multiplies it by the grid width, and then adds the cell's X value.

index.html (Compute createShaderModule call)

```
@group(0) @binding(0) var<uniform> grid: vec2f;

@group(0) @binding(1) var<storage> cellStateIn: array<u32>;
@group(0) @binding(2) var<storage, read_write> cellStateOut: array<u32>;

// New function
fn cellIndex(cell: vec2u) -> u32 {
  return cell.y * u32(grid.x) + cell.x;
}

@compute @workgroup_size(${WORKGROUP_SIZE}, ${WORKGROUP_SIZE})
fn computeMain(@builtin(global_invocation_id) cell: vec3u) {

}
```

And, finally, to see that it's working, implement a really simple algorithm: if a cell is currently on, it turns off, and vice versa. It's not the Game of Life yet, but it's enough to show that the compute shader is working.

8. Add the simple algorithm, like this:

index.html (Compute createShaderModule call)

```
@group(0) @binding(0) var<uniform> grid: vec2f;

@group(0) @binding(1) var<storage> cellStateIn: array<u32>;
@group(0) @binding(2) var<storage, read_write> cellStateOut: array<u32>;

fn cellIndex(cell: vec2u) -> u32 {
  return cell.y * u32(grid.x) + cell.x;
}

@compute @workgroup_size(${WORKGROUP_SIZE}, ${WORKGROUP_SIZE})
fn computeMain(@builtin(global_invocation_id) cell: vec3u) {
  // New lines. Flip the cell state every step.
  if (cellStateIn[cellIndex(cell.xy)] == 1) {
    cellStateOut[cellIndex(cell.xy)] = 0;
  } else {
    cellStateOut[cellIndex(cell.xy)] = 1;
  }
}
```

> **Note:** The `cell.xy` syntax here is a shorthand known as *swizzling*. It's equivalent to saying `vec2(cell.x, cell.y)` and is an easy way to get the components you need out of a vector in a different configuration. Swizzling is very flexible! You can put the components of the vector (up to four of them) in any order and repeat them as needed to do things like `cell.zyx` or `cell.yyyy`.

And that's it for your compute shader—for now! But before you can see the results, there are a few more changes that you need to make.

Use Bind Group and Pipeline Layouts

One thing that you might notice from the above shader is that it largely uses the same inputs (uniforms and storage buffers) as your render pipeline. So you might think that you can simply use the same bind groups and be done with it, right? The good news is that you can! It just takes a bit more manual setup to be able to do that.

Any time that you create a bind group, you need to provide a `GPUBindGroupLayout`. Previously, you got that layout by calling `getBindGroupLayout()` on the render pipeline, which in turn created it automatically because you supplied `layout: "auto"` when you created it. That approach works well when you only use a single pipeline, but if you have multiple pipelines that want to share resources, you need to create the layout explicitly, and then provide it to both the bind group and pipelines.

To help understand why, consider this: in your render pipelines you use a single uniform buffer and a single storage buffer, but in the compute shader you just wrote, you need a second storage buffer. Because the two shaders use the same `@binding` values for the uniform and first storage buffer, you can share those between pipelines, and the render pipeline ignores the second storage buffer, which it doesn't use. You want to create a layout that describes *all* of the resources that are present in the bind group, not just the ones used by a specific pipeline.

1. To create that layout, call `device.createBindGroupLayout()`:

index.html

```
// Create the bind group layout and pipeline layout.
const bindGroupLayout = device.createBindGroupLayout({
  label: "Cell Bind Group Layout",
  entries: [{
    binding: 0,
    visibility: GPUShaderStage.VERTEX | GPUShaderStage.COMPUTE,
```

```
    buffer: {} // Grid uniform buffer
  }, {
    binding: 1,
    visibility: GPUShaderStage.VERTEX | GPUShaderStage.COMPUTE,
    buffer: { type: "read-only-storage"} // Cell state input buffer
  }, {
    binding: 2,
    visibility: GPUShaderStage.COMPUTE,
    buffer: { type: "storage"} // Cell state output buffer
  }]
});
```

This is similar in structure to creating the bind group itself, in that you describe a list of `entries`. The difference is that you describe what type of resource the entry must be and how it's used rather than providing the resource itself.

In each entry, you give the `binding` number for the resource, which (as you learned when you created the bind group) matches the `@binding` value in the shaders. You also provide the `visibility`, which are `GPUShaderStage` flags that indicate which shader stages can use the resource. You want both the uniform and first storage buffer to be accessible in the vertex and compute shaders, but the second storage buffer only needs to be accessible in compute shaders. You can also make resources accessible to fragment shaders with these flags, but you have no need to do so here.

Finally, you indicate what type of resource is being used. This is a different dictionary key, depending on what you need to expose. Here, all three resources are buffers, so you use the `buffer` key to define the options for each. Other options include things like `texture` or `sampler`, but you don't need those here.

In the buffer dictionary, you set options like what `type` of buffer is used. The default is `"uniform"`, so you can leave the dictionary empty for binding 0. (You do have to at least set `buffer: {}`, though, so that the entry is identified as a buffer.) Binding 1 is given a type of `"read-only-storage"` because you don't use it with `read_write` access in the shader, and binding 2 has a type of `"storage"` because you *do* use it with `read_write` access!

Once the `bindGroupLayout` is created, you can pass it in when creating your bind groups rather than querying the bind group from the pipeline. Doing so means that you need to add a new storage buffer entry to each bind group in order to match the layout you just defined.

2. Update the bind group creation, like this:

index.html

```
// Create a bind group to pass the grid uniforms into the pipeline
const bindGroups = [
  device.createBindGroup({
    label: "Cell renderer bind group A",
    layout: bindGroupLayout, // Updated Line
    entries: [{
      binding: 0,
      resource: { buffer: uniformBuffer }
    }, {
      binding: 1,
      resource: { buffer: cellStateStorage[0] }
    }, {
      binding: 2, // New Entry
      resource: { buffer: cellStateStorage[1] }
    }],
  }),
  device.createBindGroup({
    label: "Cell renderer bind group B",
    layout: bindGroupLayout, // Updated Line

    entries: [{
      binding: 0,
      resource: { buffer: uniformBuffer }
    }, {
      binding: 1,
      resource: { buffer: cellStateStorage[1] }
    }, {
      binding: 2, // New Entry
      resource: { buffer: cellStateStorage[0] }
    }],
  }),
];
```

And now that the bind group has been updated to use this explicit bind group layout, you need to update the render pipeline to use the same thing.

3. Create a `GPUPipelineLayout`.

index.html

```
const pipelineLayout = device.createPipelineLayout({
  label: "Cell Pipeline Layout",
  bindGroupLayouts: [ bindGroupLayout ],
});
```

A pipeline layout is a list of bind group layouts (in this case, you have one) that one or more pipelines use. The order of the bind group layouts in the array needs to correspond with the `@group` attributes in the shaders. (This means that `bindGroupLayout` is associated with `@group(0)`.)

4. Once you have the pipeline layout, update the render pipeline to use it instead of `"auto"`.

index.html

```
const cellPipeline = device.createRenderPipeline({
  label: "Cell pipeline",
  layout: pipelineLayout, // Updated!
  vertex: {
    module: cellShaderModule,
    entryPoint: "vertexMain",
    buffers: [vertexBufferLayout]
  },
  fragment: {
    module: cellShaderModule,
    entryPoint: "fragmentMain",
    targets: [{
      format: canvasFormat
    }]
  }
});
```

Create the compute pipeline

Just like you need a render pipeline to use your vertex and fragment shaders, you need a compute pipeline to use your

compute shader. Fortunately, compute pipelines are *far* less complicated than render pipelines, as they don't have any state to set, only the shader and layout.

- Create a compute pipeline with the following code:

index.html

```
// Create a compute pipeline that updates the game state.
const simulationPipeline = device.createComputePipeline({
  label: "Simulation pipeline",
  layout: pipelineLayout,
  compute: {
    module: simulationShaderModule,
    entryPoint: "computeMain",
  }
});
```

Notice that you pass in the new `pipelineLayout` instead of `"auto"`, just like in the updated render pipeline, which ensures that both your render pipeline and your compute pipeline can use the same bind groups.

Compute passes

This brings you to the point of actually making use of the compute pipeline! Given that you do your rendering in a render pass, you can probably guess that you need to do compute work in a compute pass. Compute and render work can both happen in the same command encoder, so you want to shuffle your `updateGrid` function a bit.

1. Move the encoder creation to the top of the function, and then begin a compute pass with it (before the `step++`).

index.html

```
// In updateGrid()
// Move the encoder creation to the top of the function.
const encoder = device.createCommandEncoder();

const computePass = computeEncoder.beginComputePass();

// Compute work will go here...

computePass.end();

// Existing lines
step++; // Increment the step count

// Start a render pass...
```

Just like compute pipelines, compute passes are much simpler to kick off than their rendering counterparts because you don't need to worry about any attachments.

You want to do the compute pass before the render pass because it allows the render pass to immediately use the latest results from the compute pass. That's also the reason that you increment the `step` count between the passes, so that the output buffer of the compute pipeline becomes the input buffer for the render pipeline.

2. Next, set the pipeline and bind group inside the compute pass, using the same pattern for switching between bind groups as you do for the rendering pass.

index.html

```
const computePass = computeEncoder.beginComputePass();

// New lines
computePass.setPipeline(simulationPipeline),
computePass.setBindGroup(0, bindGroups[step % 2]);

computePass.end();
```

3. Finally, instead of drawing like in a render pass, you dispatch the work to the compute shader, telling it how many workgroups you want to execute on each axis.

index.html

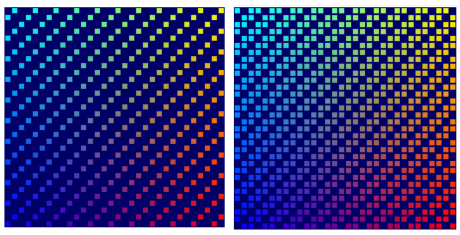
```
const computePass = computeEncoder.beginComputePass();

computePass.setPipeline(simulationPipeline),
computePass.setBindGroup(0, bindGroups[step % 2]);

// New lines
const workgroupCount = Math.ceil(GRID_SIZE / WORKGROUP_SIZE);
computePass.dispatchWorkgroups(workgroupCount, workgroupCount);

computePass.end();
```

Something *very important* to note here is that the number you pass into `dispatchWorkgroups()` is **not** the number of invocations! Instead, it's the number of workgroups to execute, as defined by the `@workgroup_size` in your shader.

If you want the shader to execute 32x32 times in order to cover your entire grid, and your workgroup size is 8x8, you need to dispatch 4x4 workgroups (4 * 8 = 32). That's why you divide the grid size by the workgroup size and pass that value into `dispatchWorkgroups()`.

> **Note:** Compute work can only be dispatched by workgroup, so if you have a workload that's not an even divisor of the workgroup size, you have two options. You can either change the workgroup size (which has to be done at shader module creation time, and has relatively high overhead), or you can round up the number of workgroups you dispatch, and then, in the shader, check to see if you're over the desired `global_invocation_id` and need to return early.

Now you can refresh the page again, and you should see that the grid inverts itself with each update.



Implement the algorithm for the Game of Life

Before you update the compute shader to implement the final algorithm, you want to go back to the code that's initializing the storage buffer content and update it to produce a random buffer on each page load. (Regular patterns don't make for very interesting Game of Life starting points.) You can randomize the values however you want, but there's an easy way to start that gives reasonable results.

1. To start each cell in a random state, update the `cellStateArray` initialization to the following code:

index.html

```
// Set each cell to a random state, then copy the JavaScript array
// into the storage buffer.
for (let i = 0; i < cellStateArray.length; ++i) {
  cellStateArray[i] = Math.random() > 0.6 ? 1 : 0;
}
device.queue.writeBuffer(cellStateStorage[0], 0, cellStateArray);
```

Now you can finally implement the logic for the Game of Life simulation. After everything it took to get here, the shader code may be disappointingly simple!

First, you need to know for any given cell how many of its neighbors are active. You don't care about which ones are active, only the count.

2. To make getting neighboring cell data easier, add a `cellActive` function that returns the `cellStateIn` value of the given coordinate.

index.html (Compute createShaderModule call)

```
fn cellActive(x: u32, y: u32) -> u32 {
  return cellStateIn[cellIndex(vec2(x, y))];
}
```

The `cellActive` function returns one if the cell is active, so adding the return value of calling `cellActive` for all eight surrounding cells gives you how many neighboring cells are active.

3. Find the number of active neighbors, like this:

index.html (Compute createShaderModule call)

```
fn computeMain(@builtin(global_invocation_id) cell: vec3u) {
  // New lines:
  // Determine how many active neighbors this cell has.
  let activeNeighbors = cellActive(cell.x+1, cell.y+1) +
                        cellActive(cell.x+1, cell.y) +
                        cellActive(cell.x+1, cell.y-1) +
                        cellActive(cell.x, cell.y-1) +
                        cellActive(cell.x-1, cell.y-1) +
                        cellActive(cell.x-1, cell.y) +
                        cellActive(cell.x-1, cell.y+1) +
                        cellActive(cell.x, cell.y+1);
```

This leads to a minor problem, though: what happens when the cell you're checking is off the edge of the board? According to your `cellIndex()` logic right now, it either overflows to the next or previous row, or runs off the edge of the buffer!

> **Caution:** Unlike some languages, indexing outside the bounds of a buffer in WGSL isn't *unsafe*, but it is unpredictable. The language makes sure you still get *some* value from the buffer, so you can't accidentally end up reading data from another process or anything like that, but it's almost certain to not give you the results you want. And the behavior may be different, depending on the platform and browser you use, so you should never depend on out-of-bounds accesses.

For the Game of Life, a common and easy way to resolve this is to have cells on the edge of the grid treat cells on the opposite edge of the grid as their neighbors, creating a kind of wrap-around effect.

4. Support grid wrap-around with a minor change to the `cellIndex()` function.

index.html (Compute createShaderModule call)

```
fn cellIndex(cell: vec2u) -> u32 {
  return (cell.y % u32(grid.y)) * u32(grid.x) +
         (cell.x % u32(grid.x));
}
```

By using the `%` operator to wrap the cell X and Y when it extends past the grid size, you ensure that you never access outside the storage buffer bounds. With that, you can rest assured that the `activeNeighbors` count is predictable.

Then you apply one of four rules:

- Any cell with fewer than two neighbors becomes inactive.
- Any active cell with two or three neighbors stays active.
- Any inactive cell with exactly three neighbors becomes active.
- Any cell with more than three neighbors becomes inactive.

You can do this with a series of if statements, but WGSL also supports switch statements, which are a good fit for this logic.

5. Implement the Game of Life logic, like this:

index.html (Compute createShaderModule call)

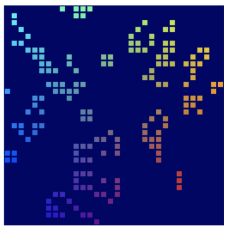
```
let i = cellIndex(cell.xy);

// Conway's game of life rules:
switch activeNeighbors {
  case 2: { // Active cells with 2 neighbors stay active.
    cellStateOut[i] = cellStateIn[i];
  }
  case 3: { // Cells with 3 neighbors become or stay active.
    cellStateOut[i] = 1;
  }
  default: { // Cells with < 2 or > 3 neighbors become inactive.
    cellStateOut[i] = 0;
  }
}
```

For reference, the final compute shader module call now looks like this:

index.html

```
const simulationShaderModule = device.createShaderModule({
  label: "Life simulation shader",
  code: `
    @group(0) @binding(0) var<uniform> grid: vec2f;

    @group(0) @binding(1) var<storage> cellStateIn: array<u32>;
    @group(0) @binding(2) var<storage, read_write> cellStateOut: array<u32>;

    fn cellIndex(cell: vec2u) -> u32 {
      return (cell.y % u32(grid.y)) * u32(grid.x) +
             (cell.x % u32(grid.x));
    }

    fn cellActive(x: u32, y: u32) -> u32 {
      return cellStateIn[cellIndex(vec2(x, y))];
    }

    @compute @workgroup_size(${WORKGROUP_SIZE}, ${WORKGROUP_SIZE})
    fn computeMain(@builtin(global_invocation_id) cell: vec3u) {
      // Determine how many active neighbors this cell has.
      let activeNeighbors = cellActive(cell.x+1, cell.y+1) +
                            cellActive(cell.x+1, cell.y) +
                            cellActive(cell.x+1, cell.y-1) +
                            cellActive(cell.x, cell.y-1) +
                            cellActive(cell.x-1, cell.y-1) +
                            cellActive(cell.x-1, cell.y) +
                            cellActive(cell.x-1, cell.y+1) +
                            cellActive(cell.x, cell.y+1);

      let i = cellIndex(cell.xy);

      // Conway's game of life rules:
      switch activeNeighbors {
        case 2: {
          cellStateOut[i] = cellStateIn[i];
        }
        case 3: {
          cellStateOut[i] = 1;
        }
        default: {
          cellStateOut[i] = 0;
        }
      }
    }
  `
});
```

And... that's it! You're done! Refresh your page and watch your newly built cellular automaton grow!



9. Congratulations!

You created a version of the classic Conway's Game of Life simulation that runs entirely on your GPU using the WebGPU API!

What's next?

- Review the WebGPU Samples

Further reading

- WebGPU — All of the cores, none of the canvas
- Raw WebGPU
- WebGPU Fundamentals
- WebGPU Best Practices

Reference docs

- WebGPU Specification
- WGSL Specification