WebGPU Buffer Uploads

Introduction

In WebGPU, the `GPUBuffer` is one of the primary objects you'll be working with. It, along side `GPUTextures` represent the majority of the data that your application pushes to the GPU for rendering. In WebGPU buffers are used for Vertex and Index data, uniforms, generic storage for compute and fragment shaders, and as a staging area for texture data.

This doc is focused on the best ways to get data into these buffers, regardless of their ultimate use.

Buffer data flow

Before we get into the mechanics of setting buffer data, though, let's talk about what it looks like under the hood.

In general, you can consider WebGPU to be working with two types of memory: Memory that is GPU accessible and memory that is CPU accessible and able to efficiently copy to the GPU accessible memory. Any time you want to access data from a shader (vertex, fragment, or compute) it *must* be in GPU accessible memory, and any time you want to access data from JavaScript it *must* be in CPU accessible memory. Buffers can be either GPU or CPU accessible, but not both, and Textures are always only GPU accessible.

On some devices, like a phone, these may in fact be the same memory pool. On others, like a PC with a discrete graphics card, they may be on different physical boards and only able to communicate across a PCIe bus or similar. Because we're developing for the web, we want to be able to write a single code path that works on the widest array of devices. As a result WebGPU does not differentiate between those memory configurations in the same way that, say, Vulkan does. Everything is treated as if it has distinct CPU and GPU memory pools, and it's up to the WebGPU implementation to optimize for specific architectures where it can.

This means that all data going into GPU accessible memory will take approximately the same path:

- A "staging" buffer is created with CPU accessible memory that can be written to and copied from. (usage: GPUBufferUsage.MAP_WRITE | GPUBufferUsage.COPY_SRC)
- The staging buffer is mapped for writing (via `mapAsync()`), which makes it's memory writable as a JavaScript `ArrayBuffer`.
- Data is placed in the array buffer.
- The staging buffer is unmapped.
- A copy command (ie: `copyBufferToBuffer()` or `copyBufferToTexture()`) is used to copy the data out of the staging buffer and into a GPU accessible destination.

And a similar path is used to read data back from GPU accessible memory:

- A "staging" buffer is created with CPU accessible memory that can be copied to and read from. (usage: GPUBufferUsage.MAP_READ | GPUBufferUsage.COPY_DST)
- A copy command (ie: `copyBufferToBuffer()` or `copyTextureToBuffer()`) is used to copy the data out of a GPU accessible destination and into the staging buffer.
- The staging buffer is mapped for reading (via `mapAsync()`), which makes it's memory readable as a JavaScript `ArrayBuffer`.
- The data is read out of the array buffer.
- The staging buffer is unmapped.

As you'll see, some of the methods below hide some of these steps by making them implicit, but you can generally assume that's what's happening behind the scenes in most cases.

So, with that established, let's look at some more concrete methods of getting data into GPU accessible buffers in different scenarios.

When in doubt, `writeBuffer()`!

The very first thing to establish is that if you ever have a question about what the best way is to get data into a particular buffer, the `writeBuffer()` method is always a safe fallback that doesn't have many downsides.

`writeBuffer()` is a convenience method on `GPUQueue` that copies values from an `ArrayBuffer` into a `GPUBuffer` in whatever way the user agent deems best. Generally this will be a fairly efficient path, and in some cases it can be the *most* efficient path! (In most cases the user agent will manage an implicit staging buffer for you when you call `writeBuffer()`, but on some architectures it's feasible that it could skip that step.)

Specifically, if you are using WebGPU from WASM code, `writeBuffer()` is the preferred path. This is because WASM apps would need to perform an additional copy from the WASM heap when you use a mapped buffer.

In summary, the advantages of using `writeBuffer()` are:

- Preferred route for WASM apps.
- Lowest overall code complexity.
- Sets the buffer data immediately.
- If your data is already in an `ArrayBuffer`, avoids allocation/copy of a mapped `ArrayBuffer`.
- Avoids the need to set the contents of a mapped buffer's array to zero before returning it.
- Allows the user agent to pick an (presumably optimal) pattern for uploading the data to the GPU.

And really, there's not any explicit downsides to using this path. Depending on the exact usage pattern you may be able to write a more custom buffer management system that gets better performance in one situation or another, but `writeBuffer()` is an extremely solid catch-all solution for setting buffer data.

Here's an example of using `writeBuffer()`. You can see the code is very brief:

```
// At some point during the app startup...
const projectionMatrixBuffer = gpuDevice.createBuffer({
  size: 16 * Float32Array.BYTES_PER_ELEMENT, // Large enough for a 4x4 matrix
  usage: GPUBufferUsage.VERTEX | GPUBufferUsage.COPY_DST, // COPY_DST is required
});

// Whenever the projection matrix changes (ie: window is resized)...
function updateProjectionMatrixBuffer(projectionMatrix) {
  const projectionMatrixArray = projectionMatrix.getAsFloat32Array();
  gpuDevice.queue.writeBuffer(projectionMatrixBuffer, 0, projectionMatrixArray, 0, 16);
}
```

Buffers that are written once and never change

There's many cases where you will create a buffer whose contents need to be set once on creation and then never changed again. A simple example would be the vertex and index buffers for a static mesh: The buffer itself needs to be filled with the mesh data immediately after the buffer is created, after which any changes to the mesh during the render loop will be done with a transform matrix or maybe mesh skinning the the vertex shader. The only time the buffer contents will change after it's initially set is when it's eventually destroyed.

In this case, one of the best ways to set the buffer's data is using the `mappedAtCreation` flag when calling `createBuffer()`. This creates the buffer in a mapped state, so that `getMappedRange()` can be called immediately after creation. This provides an `ArrayBuffer` to be filled, after which `unmap()` is called and the buffer data is set! In practice the browser will almost certainly need to do a copy of the array buffer contents to the GPU in the background after `unmap()` is called, but you can generally be assured that it's done in an efficient manner. (Just like in the `writeBuffer()` case, most of the time this will mean the user agent is managing an implicit staging buffer for you.)

The primary advantage of this approach is that if your buffer data is being created dynamically, you can save on at least one CPU-side copy by generating the data directly into the mapped buffer.

Advantages to this approach:

- Sets the buffer data immediately.
- No specific usage flags are required.
- Data can be written directly into the mapped buffer to avoid a CPU-side copy.

Disadvantages:

- Only works for newly created buffers.
- User agent must zero out the buffer before it's mapped.
- If data is already in an `ArrayBuffer`, requires another CPU-side copy.

Here's an example of using `mappedAtCreation` to set some static vertex data:

```
// Creates a grid of vertices on the X, Y plane
function createXYPlaneVertexBuffer(width, height) {
  const vertexSize = 3 * Float32Array.BYTES_PER_ELEMENT; // Each vertex is 3 floats (X,Y,Z position)

  const vertexBuffer = gpuDevice.createBuffer({
    size: width * height * vertexSize, // Allocate enough space for all the vertices
    usage: GPUBufferUsage.VERTEX, // COPY_DST is not required!
    mappedAtCreation: true,
  });

  const vertexPositions = new Float32Array(vertexBuffer.getMappedRange()),

  // Build the vertex grid
  for (let y = 0; y < height; ++y) {
    for (let x = 0; x < width; ++x) {
      const vertexIndex = y * width + x;
      const offset = vertexIndex * 3;

      vertexPositions[offset + 0] = x;
      vertexPositions[offset + 1] = y;
      vertexPositions[offset + 2] = 0;
    }
  }

  // Commit the buffer contents to the GPU
  vertexBuffer.unmap();

  return vertexBuffer;
}
```

Buffers that are written to frequently

If you have buffers that change frequently (such as once per frame) then updating them efficiently is slightly more complicated. Though before we go any further it should be noted that in many cases using `writeBuffer()` will be a perfectly acceptable path to take from a performance perspective!

Applications that want to have more explicit control over their memory usage, though, can use what's known as a staging buffer ring. This technique uses a rotating set of staging buffers and to continuously "feed" a GPU accessible buffer with new data. Each time the data is updated it first checks to see if a previously used staging buffer is already mapped and ready to use, and if so writes the data into that. If not, a new staging buffer is created with mappedAtCreation set to true so that it can immediately be populated. After the data is copied

GPU-side the staging buffer is immediately mapped again, and once the mapping is complete it's placed in the queue of buffers which are ready for use. If the buffer data is updated frequently this typically results in a list of 2-3 staging buffers that are cycled through.

This approach is the most complicated in terms of buffer management, and uses more staging memory on an ongoing basis than the others. It can be good for the GPU's work pipelining, though, and gives you lots of control and the ability to tweak for specific situations.

Advantages:

- Limits buffer creation.
- Doesn't wait on previously used buffers to be mapped.
- Staging buffer re-use means initialization costs are only paid once per set.
- Data can be written directly into the mapped buffer to avoid a CPU-side copy.

Disadvantages:

- Higher complexity than other methods.
- Higher ongoing memory usage.
- User agent must zero out the staging buffers the first time they are mapped.
- If data is already in an `ArrayBuffer`, requires another CPU-side copy.

Here's an example of how a staging buffer ring could work setting vertex data:

```
const waveGridSize = 1024;
const waveGridBufferSize = waveGridSize * waveGridSize * 3 * Float32Array.BYTES_PER_ELEMENT;
const waveGridVertexBuffer = gpuDevice.createBuffer({
  size: waveGridBufferSize,
  usage: GPUBufferUsage.VERTEX | GPUBufferUsage.COPY_DST,
});
const waveGridStagingBuffers = [];

// Updates a grid of vertices on the X, Y plane with wave-like motion
function updateWaveGrid(time) {
  // Get a new or re-used staging buffer that's already mapped.
  let stagingBuffer;
  if (waveGridStagingBuffers.length) {
    stagingBuffer = waveGridStagingBuffers.pop();
  } else {
    stagingBuffer = gpuDevice.createBuffer({
      size: waveGridBufferSize,
      usage: GPUBufferUsage.MAP_WRITE | GPUBufferUsage.COPY_SRC,
      mappedAtCreation: true,
    });
  }

  // Fill in the vertex grid values.
  const vertexPositions = new Float32Array(stagingBuffer.getMappedRange()),
  for (let y = 0; y < height; ++y) {
    for (let x = 0; x < width; ++x) {
      const vertexIndex = y * width + x;
      const offset = vertexIndex * 3;

      vertexPositions[offset + 0] = x;
      vertexPositions[offset + 1] = y;
      vertexPositions[offset + 2] = Math.sin(time + (x + y) * 0.1);
    }
  }
  stagingBuffer.unmap();

  // Copy the staging buffer contents to the vertex buffer.
  const commandEncoder = gpuDevice.createCommandEncoder({});
  commandEncoder.copyBufferToBuffer(stagingBuffer, 0, waveGridVertexBuffer, 0, waveGridBufferSize);
  gpuDevice.queue.submit([commandEncoder.finish()]);
  // Immediately after copying, re-map the buffer. Push onto the list of staging buffers when the
  // mapping completes.
  stagingBuffer.mapAsync(GPUMapMode.WRITE).then(() => {
    waveGridStagingBuffers.push(stagingBuffer);
  });
}
```

When it’s math all the way down, generate your data on the GPU!

While it’s beyond the scope of this document, I’d be remiss if I didn’t mention the ultimate technique for getting data into a buffer fast: Generating it on the GPU! Specifically, WebGPU’s compute shaders are an *excellent* tool for populating buffers efficiently. Doing so has the massive upside of not requiring any staging buffers and thus avoiding the need for copies. But, of course, GPU-side buffer generation only really works if your data can be completely algorithmically computed and doesn’t apply to things like models loaded from a file.

Real world example

If you want to see these techniques (and a few others) at work in the real world, you should check out my WebGPU Metaballs demo. Use the “metaballMethod” drop-down to select the buffer population method to use, though don’t expect to see much performance difference between them (with the exception of the compute shader method.) You can also see the code for each of the techniques, with comments explaining each. It also details a couple more patterns not covered here, primarily because the circumstances in which they’d be the most efficient path are pretty rare.

Further reading

If you want to learn more about the mechanics of buffer use, I’d recommend looking through the WebGPU Explainer and relevant bits of the WebGPU spec. The spec, in particular, isn’t exactly what I’d call “light reading” but it describes the expected behavior of WebGPU buffers in great detail.

Have fun, and make cool stuff!

The variety of patterns that can be used to get data onto the GPU can make this area of WebGPU feel confusing and possibly a bit intimidating, but it doesn’t have to be! The number one thing to keep in mind is that this flexibility exists to offer high end, professional apps a way to tightly control their performance. For the average WebGPU developer you can and should start off by using the easiest approaches: calling `writeBuffer()` to update buffers and maybe using `mappedAtCreation` for buffers that only need to be set once. **These aren’t “dumbed down” helper functions! They’re the recommended, high performance route that just so happens to also be the simplest to write.** Only try to get fancier if you see that writing to buffers is a bottleneck for your application and you can identify an alternative technique that works well for your use case.

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