WebGPU – Best Practices

WebGL/WebGPU meetup, Feb 2022
https://bit.ly/3KFiLo8

Who am I?

https://bit.ly/3KFiLo8

- Chrome team @ Google for 9 years
- WebGPU
  - Spec editor
  - Implementation
- WebXR
  - Spec editor
  - Implementation
- WebGL implementation

I also like to build demos with the tech I help implement!

[WebGPU Clustered Shading]

[WebGPU Metaballs]

https://spookyball.com

https://bit.ly/3KFiLo8

A whirlwind tour of some Best Practices

https://bit.ly/3KFiLo8

Use labels *everywhere*!

Every object in WebGPU can be given a label, either at creation time or later with the `label` attribute. Use them obsessively! It has practically zero runtime cost and will make debugging much, MUCH easier.

```
const projectionMatrixBuffer = gpuDevice.createBuffer({
  label: 'Projection Matrix Buffer',
  size: 12 * Float32Array.BYTES_PER_ELEMENT, // Oops! Should have been 16!
  usage: GPUBufferUsage.VERTEX | GPUBufferUsage.COPY_DST,
});
const projectionMatrixArray = new Float32Array(16);

gpuDevice.queue.writeBuffer(projectionMatrixBuffer, 0, projectionMatrixArray);
```

Validation errors will use labels when reporting the error messages, making the cause easier to identify:

```
// Console output
Write range (bufferOffset: 0, size: 64) does not fit in [Buffer "Projection Matrix Buffer"] size (48).
```

https://bit.ly/3KFiLo8

Use debug groups

Command buffers allow you to push and pop debug groups, which are simply strings indicating what section of code is executing. Validation errors will show the stack in the error message.

```
commandEncoder.pushDebugGroup('Frame ${frameIndex}');
  commandEncoder.pushDebugGroup('Clustered Light Compute Pass');
    updateClusteredLights(commandEncoder);
  commandEncoder.popDebugGroup();
  commandEncoder.pushDebugGroup('Main Render Pass');
    renderScene(commandEncoder);
  commandEncoder.popDebugGroup();
commandEncoder.popDebugGroup();
```

```
// Console output
Binding sizes are too small for bind group [BindGroup] at index 0

Debug group stack:
> “Main Render Pass”
> “Frame 234”
```

https://bit.ly/3KFiLo8

Load Image Textures from Blobs

Use `ImageBitmaps` created from `Blobs` for best decode performance of JPG/PNG textures.

```
async function webGPUTextureFromImageUrl(gpuDevice, url) {
  const response = await fetch(url);
  const blob = await response.blob();
  const source = await createImageBitmap(blob);

  const textureDescriptor = {
    label: `Image Texture ${url}`,
    size: { width: source.width, height: source.height },
    format: 'rgba8unorm',
    usage: GPUTextureUsage.TEXTURE_BINDING | GPUTextureUsage.COPY_DST
  };
  const texture = gpuDevice.createTexture(textureDescriptor);

  gpuDevice.queue.copyExternalImageToTexture({ source }, { texture }, textureDescriptor.size);

  return texture;
}
```

https://bit.ly/3KFiLo8

...but prefer Compressed Textures!

You'll get the best performance by using compressed textures whenever possible.

WebGPU will ship with at least three different texture compression types:

- texture-compression-bc
- texture-compression-etc2
- texture-compression-astc

Support will depend on hardware capabilities, but all platforms will be required to support either BC (aka: DXT, S3TC) or **both** ETC2 and ASTC compression, so you are guaranteed access to texture compression.

Highly recommended to use a supercompressed texture format (such as Basis Universal) that can be easily transcoded to whichever format the device supports rather than shipping two different formats.

I've built a library for efficiently loading compressed textures in both WebGL and WebGPU that you are free to use or reference: https://github.com/toji/web-texture-tool

https://bit.ly/3KFiLo8

Case study: Meet “The Paddle”

Relatively simple asset.
5 PNG textures, 2048x2048 each.
2,580 Tris.
Animations for the Ghosts.
~12 Mb when exported from Blender.

https://bit.ly/3KFiLo8

Say hello to gltf-transform

gltf-transform has a handy CLI that you can run a lot of operations from, such as compressing textures with the KHR_texture_basisu extension:

```
> gltf-transform etc1s paddle.glb paddle2.glb
paddle.glb (11.92 MB) → paddle2.glb (1.73 MB)
```

https://bit.ly/3KFiLo8

More gltf-transform goodness

In addition to compressing textures, the gltf-transform library can also resize them, resample animations, apply Draco mesh compression, and more! The asset I ship in Spookyball is **< 5%** of the original size.

```
> gltf-transform resize paddle.glb paddle2.glb --
width 1024 --height 1024
> gltf-transform etc1s paddle2.glb paddle2.glb
> gltf-transform resample paddle2.glb paddle2.glb
> gltf-transform dedup paddle2.glb paddle2.glb
> gltf-transform draco paddle2.glb paddle2.glb
  paddle.glb (11.92 MB) → paddle2.glb (596.46 KB)
```

https://bit.ly/3KFiLo8

Buffer uploads

There's a lot of ways to get data into a buffer. When in doubt, `writeBuffer()` is never the wrong answer! When using WebGPU from WASM `writeBuffer()` should *usually* be preferred because it avoids an additional data copy.

```
const projectionMatrixBuffer = gpuDevice.createBuffer({
  label: 'Projection Matrix Buffer',
  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);
}
```

https://bit.ly/3KFiLo8

Prefer asynchronous pipeline creation

If you don't need a pipeline for rendering or compute immediately, use `createRenderPipelineAsync` and `createComputePipelineAsync` instead of their synchronous counterparts.

The "synchronous" version returns a handle to a pipeline which may still be compiling behind the scenes. If you use it right away, the GPU will stall until the pipeline is ready.

The async version doesn't resolve until the pipeline is ready to use without causing stalls.

```
const computePipeline = gpuDevice.createComputePipeline({/* ... */});

computePass.setPipeline(computePipeline);
computePass.dispatch(32, 32); // Will probably jank while shaders compile!

const asyncComputePipeline = await gpuDevice.createComputePipelineAsync({/* ... */});

computePass.setPipeline(asyncComputePipeline);
computePass.dispatch(32, 32); // Shaders already done compiling, No jank. :D
```

https://bit.ly/3KFiLo8

Use implicit Pipeline Layouts sparingly

Implicit pipeline layouts are a convenience for simple pipelines (especially isolated compute pipelines) but they prevent you from sharing bind groups and can change unexpectedly when you update the shader. Prefer explicit layouts for all but the simplest situations.

```
const computePipeline = await gpuDevice.createComputePipelineAsync({
  // No layout
  compute: {
    module: computeModule,
    entryPoint: 'computeMain'
  }
});

const computeBindGroup = gpuDevice.createBindGroup({
  layout: computePipeline.getBindGroupLayout(0);
  entries: [{
    binding: 0,
    resource: { buffer: storageBuffer },
  }]
});
```

https://bit.ly/3KFiLo8

Share BindGroups/BindGroupLayouts

For example: you will frequently have values that are constant for the entire frame. In that case, you want to create a single bind group layout that is used by every pipeline in the same bind group index.

```
const cameraBindGroupLayout = device.createBindGroupLayout({
  label: `Camera uniforms BindGroupLayout`,
  entries: [{
    binding: 0,
    visibility: GPUShaderStage.VERTEX | GPUShaderStage.FRAGMENT,
    buffer: {},
  }]
});

const cameraBindGroup = gpu.device.createBindGroup({
  label: `Camera uniforms BindGroup`,
  layout: cameraBindGroupLayout,
  entries: [{
    binding: 0,
    resource: { buffer: cameraUniformsBuffer, },
  }],
});
```

https://bit.ly/3KFiLo8

Share BindGroups/BindGroupLayouts

For example: you will frequently have values that are constant for the entire frame. In that case, you want to create a single bind group layout that is used by every pipeline in the same bind group index.

```
const renderPipelineA = gpuDevice.createRenderPipeline({
  label: `Render Pipeline A`,
  layout: gpuDevice.createPipelineLayout([cameraBindGroupLayout, materialBindGroupLayoutA]),
  /* Etc... */
});

const renderPipelineB = gpuDevice.createRenderPipeline({
  label: `Render Pipeline B`,
  layout: gpuDevice.createPipelineLayout([cameraBindGroupLayout, materialBindGroupLayoutB]),
  /* Etc... */
});
```

https://bit.ly/3KFiLo8

Share BindGroups/BindGroupLayouts

Then you can ideally set up a render loop that looks something like the following, where the common bind group is only set once:

```
const renderPass = commandEncoder.beginRenderPass({/* ... */});

renderPass.setBindGroup(0, cameraBindGroup);

for (const pipeline of activePipelines) {
  renderPass.setPipeline(pipeline.gpuRenderPipeline);
  for (const material of pipeline.materials) {
    renderPass.setBindGroup(1, material.gpuBindGroup);
    for (const mesh of material.meshes) {
      renderPass.setVertexBuffer(0, mesh.gpuVertexBuffer);
      renderPass.draw(mesh.drawCount);
    }
  }
}

renderPass.endPass();
```

Thank you!

Brandon Jones

Twitter: @Tojiro

Email: bajones@google.com

These slides: https://bit.ly/3KFiLo8

More best practices