Vulkanised 2024

The 6th Vulkan Developer Conference
Sunnyvale, California | February 5-7, 2024

Preparing WebGPU's Vulkan Backend for Android

Brandon Jones, Google

This presentation: bit.ly/42sfEcr

Intro to WebGPU

WebGPU Overview

Modern Graphics/Compute API for the web

WebGL Successor

Abstraction over Vulkan, D3D12, Metal

Adheres to patterns of modern APIs

Avoids memory management/synchronization minutia

Focus on common feature set with extension mechanism

WebGPU Overview

Designed for the web, but with native headers as well

Implementations in C++ (Dawn) and Rust (wgpu)

Bindings for other languages

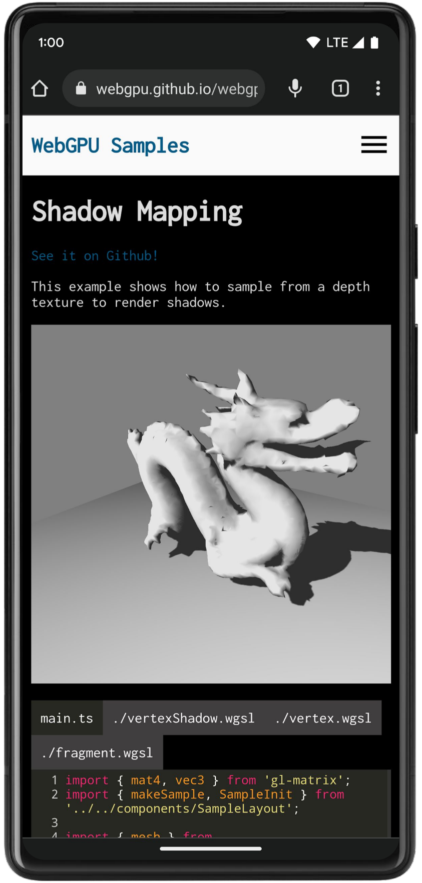
Shipping in Chrome

Windows/Mac/ChromeOS in Chrome 113 (April 2023)

Android in Chrome 121 (January 2024!)

- Initially on Android 12+
- ARM and Qualcomm GPUs
- More OS versions, GPUs planned.

In progress in Firefox, Safari (behind a flag on both)

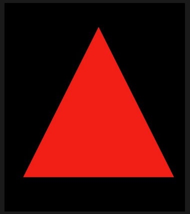


Demo!

https://playcanvas.com/demos/arealights/

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

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

const shaderModule = device.createShaderModule({ code: `
  @vertex fn vertMain(@location(0) pos : vec3f) ->
      @builtin(position) vec4f {
    return vec4f(pos, 1);
  }

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

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

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

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

device.queue.writeBuffer(vertexBuffer, 0, vertexData);

const commandEncoder = device.createCommandEncoder();

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

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

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



Heavily benefited from hindsight

WebGPU drew on the lessons learned from the modern APIs shipping
AND years of experience shipping WebGL.

Example: Implicit vs. explicit state transitions

- Feedback from a Vulkan implementor that “almost everyone misused barriers”
- Influenced our decision to be implicit

Example: Text-based shader language (WGSL)

- Beneficial to beginners, good fit for web-based content
- Useful for applying workarounds during cross-compilation

Dawn's Vulkan Implementation

Where does Chrome use Vulkan?

Vulkan is used to power WebGPU on ChromeOS and Android

Eventually will ship Linux with a Vulkan backend too.

Baseline requirements for WebGPU support:

- Vulkan 1.1
  - *Or* Vulkan 1.0 w/ VK_KHR_maintenance1 & VK_KHR_maintenance2
- fragmentStoresAndAtomics & fullDrawIndexUint32 features
- maxImageArrayLayers limit ≥ 256
- BC *or* (ETC2 && ASTC) texture compression

Why not Vulkan on Windows?

Dawn has a Vulkan backend for Windows, but we currently only ship the D3D12 backend in Chrome.

For some vendors D3D driver quality is better

Better interop with other parts of the platform we need to interface with

- DirectComposition
- Media Foundation
- Etc.

Browser looks more like an OS than a game, which influences a lot of decisions

Highlights!

From Corentin Wallez (On WebGPU API implementation):

Vulkan spec is VERY detailed

Validation layers work

Vulkan group @ Khronos is very responsive to questions and bugs

- Fixes have been very fast

More Highlights!

From David Neto (on WGSL shading language implementation):

Vulkan's memory consistency model is well developed and covers everything from the API to the shading language.

Saved a ton of work by having WGSL specify mapping to Vulkan Memory Model

SPIR-V inspired WGSL spec to be very precise about which bytes are accessed when, floating point behavior, etc.

Challenges

WebGPU is an API, not a game, app, or engine

Need to ensure that every possible code path is reliable

- Don't have the luxury of skipping an effect on a buggy piece of hardware
- We see ALL the driver bugs

Vulkan is a Bring-Your-Own-Utils API, so Dawn got to build all the same helper classes as everyone else

Vulkan is our biggest backend

Metal: 6k

D3D12: 9k

Vulkan: 13k LOC

+Lots of common code in our platform-agnostic frontend.

Shipping on Android

Highlight: It mostly worked first try!

Majority of Dawn's Vulkan backend simply worked when we got it running on Android

Testament to the portability of Vulkan and the work of our team

Majority of issues were around resource sharing with Chrome/between processes

Plenty of new and exciting driver bugs too

Robust WebGPU CTS was invaluable!

112k+ conformance test cases

Helps identify both Dawn implementation issues and driver issues

Lesson learned from WebGL, which has its own extremely valuable CTS test suite.

Canvas format issues

WebGPU spec guarantees canvases can allocate rgba8-unorm and bgra8-unorm surfaces

Due to Chrome's multiprocess architecture any canvas rendering must go through an AHardwareBuffer

AHardwareBuffer has an RGBA8 format, but no BGRA8 format

Solution was to do a just-in-time copy to the AHardwareBuffer-backed textures if needed

Devs can avoid overhead by checking the navigator.gpu.preferredCanvasFormat()

Splitting command buffers for fun and profit

Frequent crash on one vendor when a texture modified in a compute shader was sampled or written to in a subsequent render pass.

But ONLY if they were in the same VkCommandBuffer.

Fortunately, Dawn doesn't build VkCommandBuffers directly from user calls.

- Calls are recorded into intermediate format and replayed into VkCommandBuffers at submit time.
- Allows us to detect the issue when building command buffers and silently split them.

SPIR-V fixes when cross compiling from WGSL

Several issues were able to be resolved completely within our WGSL cross-compiler, Tint

Passing Matrices as function args was causing a crash on one vendor’s drivers

Re-written by Tint to be a pointer

Would have been much more difficult to work around if we consumed SPIR-V directly

Sometimes workarounds are impractical

On some devices passing an index of 0xFFFFFFFF (primitive restart value) with a “-list” topology would trigger a device loss.

No reasonable way to detect it. Definitely not going to scan every index buffer before every draw, and the device loss triggers before the associated vertex shader is called.

This is clearly bad app behavior that’s trivial developers to fix.

Device loss is annoying, but it’s not dangerous.

So… let it be! Better than blocklisting.

Enumerating the devices should be safe… right?

We know of at least one (decade old) GPU where calling vkEnumeratePhysicalDevices causes a crash.

We use information gathered from the enumerated devices to inform our blocklist.

So we need a blocklist to block some devices from checking our blocklist…

We still have work to do!

Extending Android OS and GPU support

Investigate mobile-specific optimizations

- Have exposed mobile-friendly features, like float16 shader operations.

Ongoing maintenance

Thank you!

Brandon Jones

Email: bajones@google.com

Mastodon: https://mastodon.social/@tojiro