moz://a

Introduction to WebGPU

the future of graphics and compute on the Web

Mozilla All Hands, 31th Jan 2020

Dzmitry Malyshau @kvark
Graphics Engineer

Agenda

WebGPU: why and what?

Architecture: how?

Wrap-up

Act I

protocol


moz://a
Impossible today
But may be in the future?
Lots of small objects
Large worlds
Rich materials
Screenshot from
RDR2 trailer, PS4

Situation

- Developers want to have rich content running portably on the Web
- For a game written against Dx1y/Vulkan/Metal, adding a WebGL port is a major undertaking
- Applications are CPU-limited by the 100s of draw calls
- Yet no multi-threading is possible
- No portable compute shader support

OpenGL

Render like it’s 1992

Really, what's wrong with WebGL/OpenGL?

- OpenGL is slowly becoming a thing of the past
  - Apple deprecates OpenGL in 2018, there is no WebGL 2.0 support yet
  - Microsoft not supporting OpenGL in UWP
  - IHVs focus on Vulkan and DX12 drivers
- WebGL ends up translating to Dx11 (via Angle) on Windows by major browsers
  - Could translate to Vulkan, but source API becomes the bottleneck

OpenGL: technical issues

- Changing a state can cause the driver to recompile the shader, internally
  - Causes 100ms freezes during the experience…
  - Missing concept of pipelines
- Challenging to optimize for mobile
  - Rendering tile management is critical for power-efficiency but handled implicitly
  - Missing concept of render passes
- Challenging to take advantage of more threads
  - Purely single-threaded, becomes a CPU bottleneck
  - Missing concept of command buffers
- Tricky data transfers
  - Dx11 doesn’t have buffer to texture copies
- Given that WebGL2 is not universally supported, even basic things like sampler objects are not fully available to developers

OpenGL: Evolution

GPU all the things!

Quiz:

Who started WebGPU?

hint: not Apple

History

of WebGPU

***2016 H2*:**
experiments
by browser
vendors

1

… to date

WebGL-Next

Khronos Vancouver F2F

History

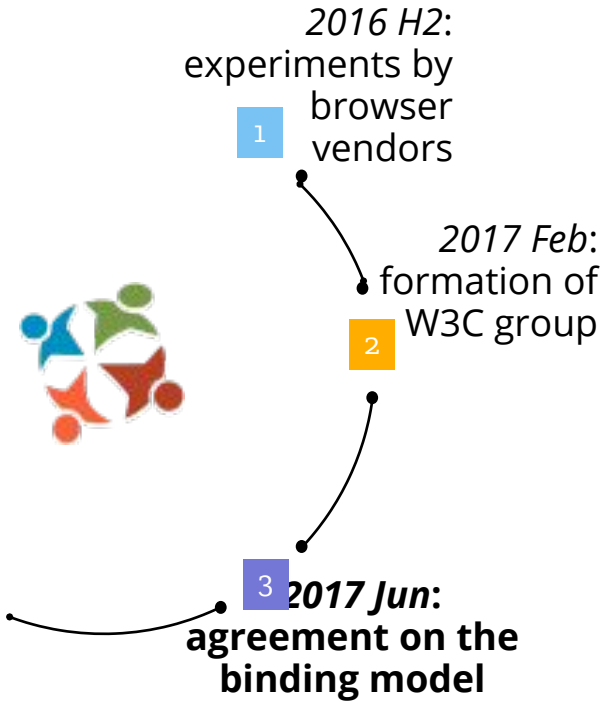
of WebGPU

2016 H2:
experiments by
1 browser
vendors

***2017 Feb*:**
formation of
2 **W3C group**

History

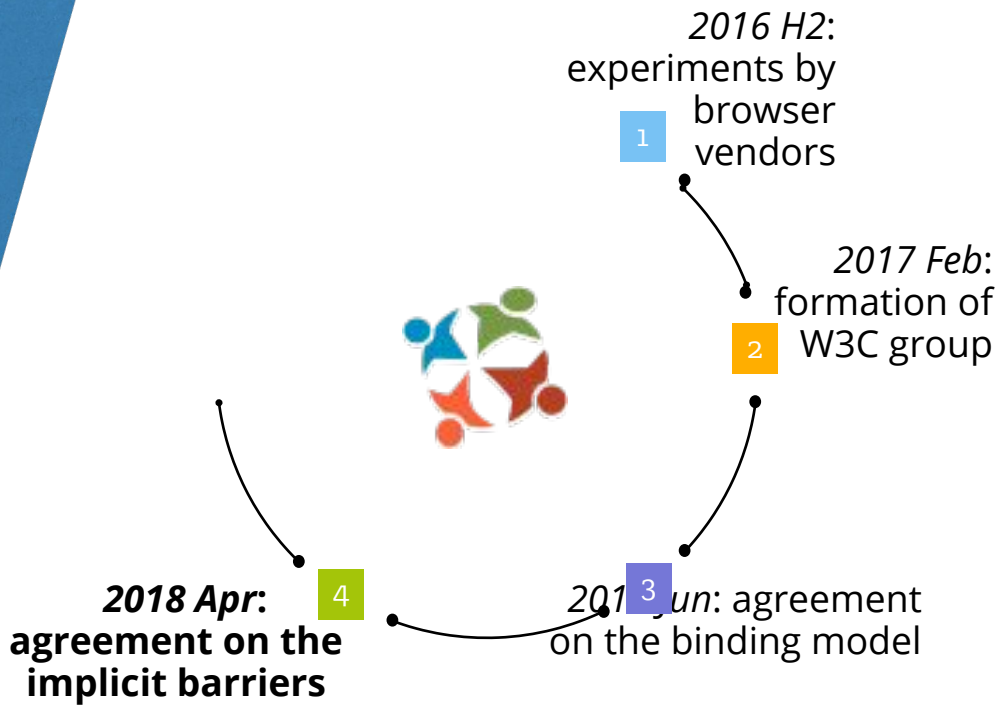
of WebGPU



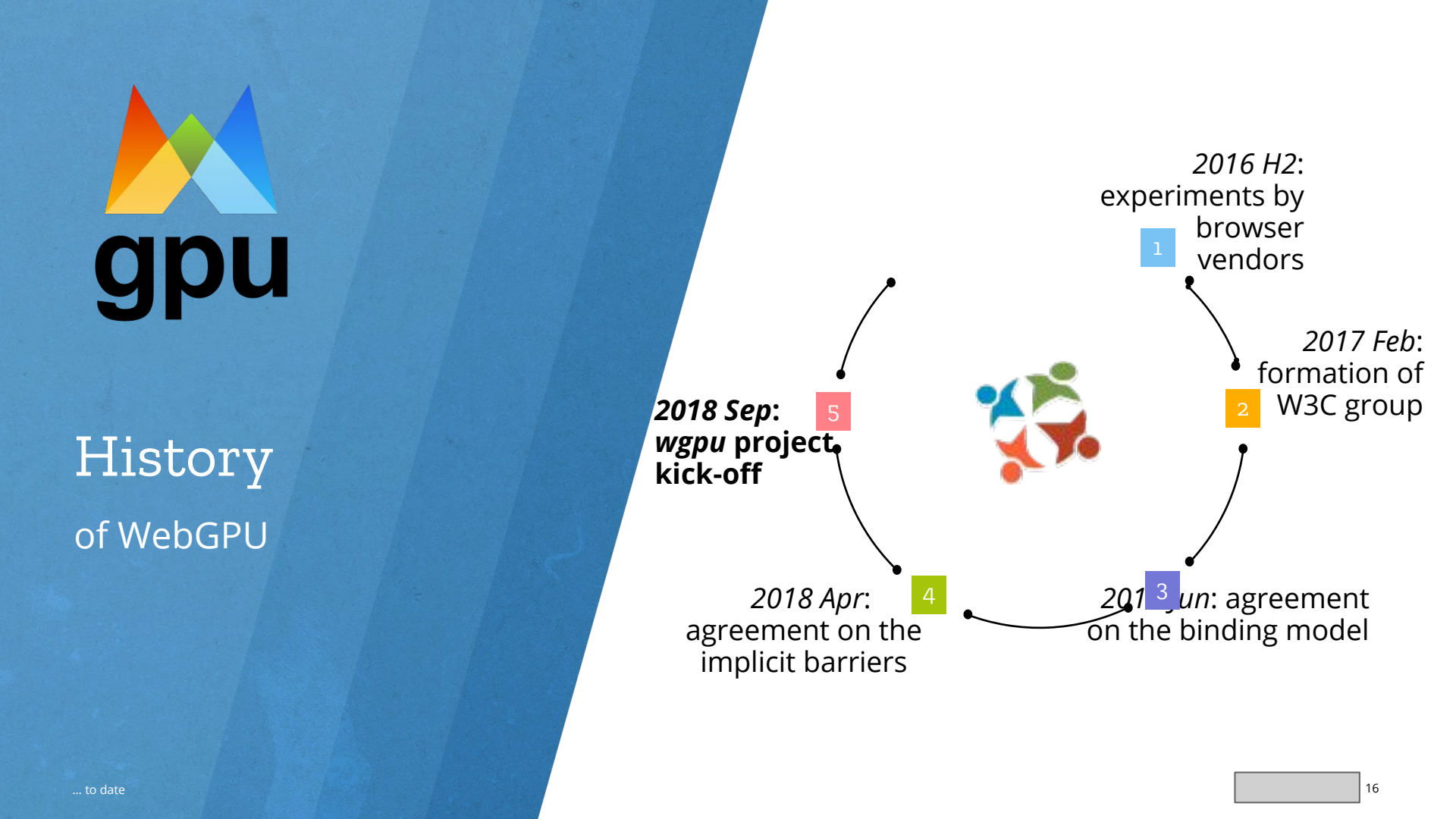
... to date

History

of WebGPU

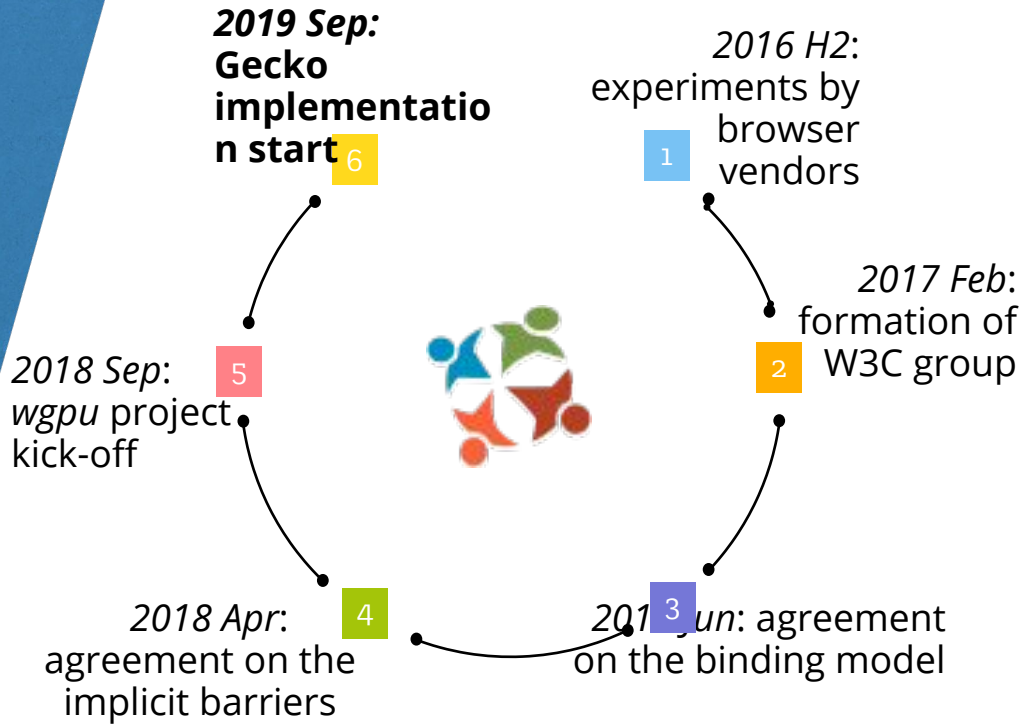


... to date


gpu
History
of WebGPU
2016 H2: experiments by browser vendors
1
2017 Feb: formation of W3C group
2
2017 Jun: agreement on the binding model
3
2018 Apr: agreement on the implicit barriers
4
2018 Sep: wgpu project kick-off
5
... to date

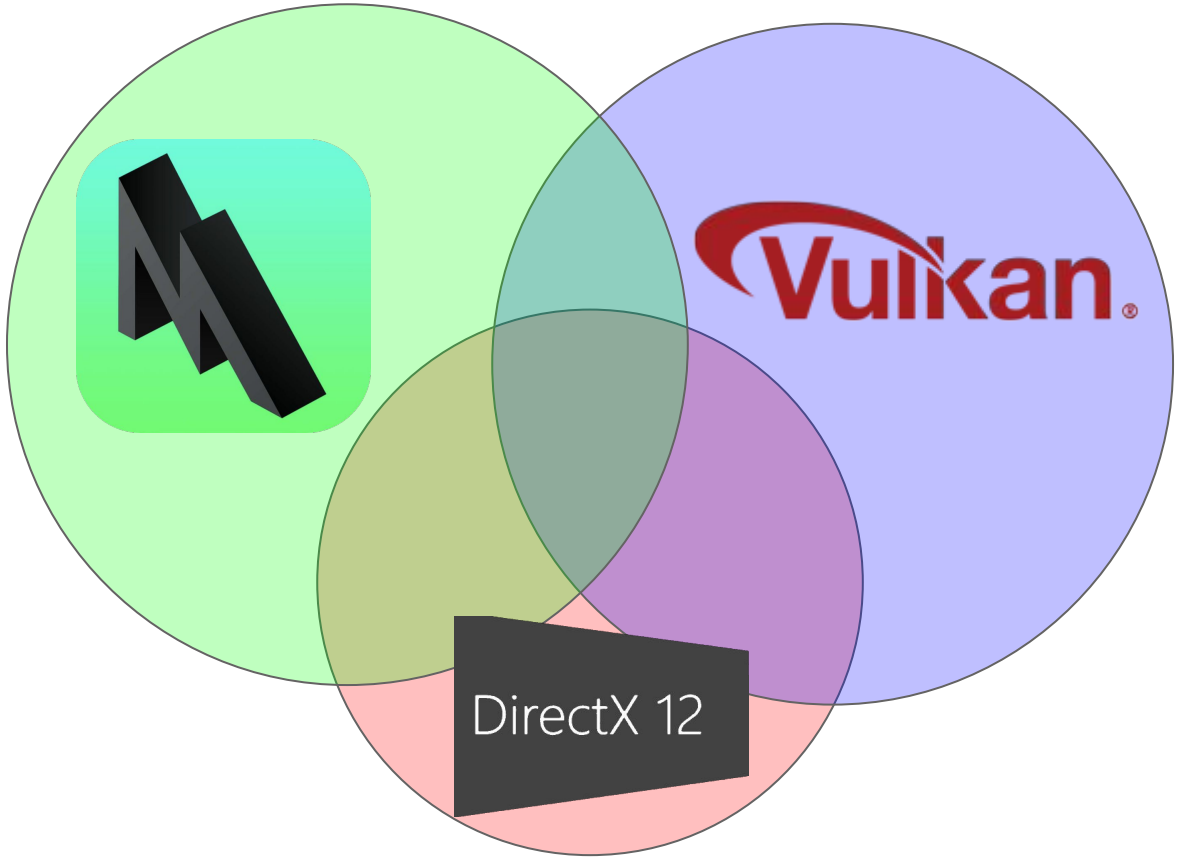
History

of WebGPU



... to date

What is WebGPU?



How standards proliferate

(insert XKCD #927 here)

Design Constraints

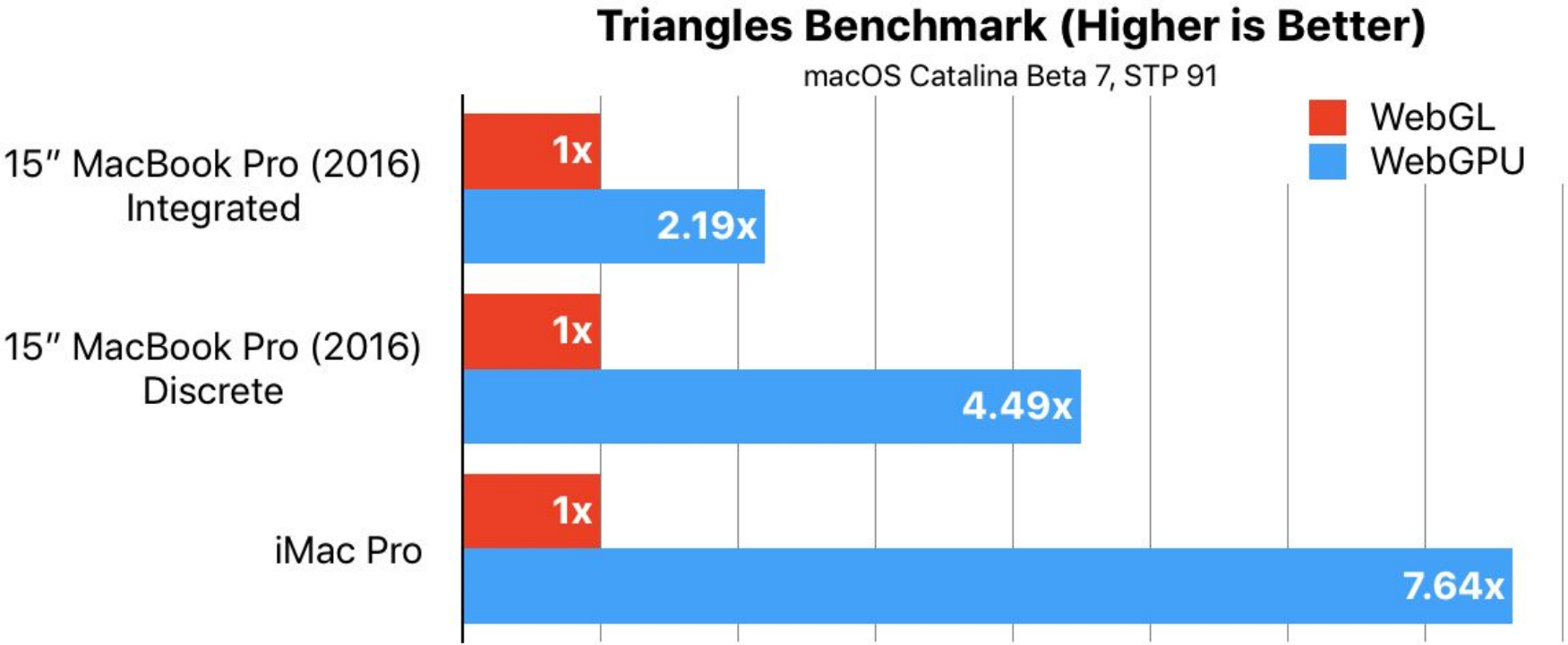
security

portability

performance

usability

Early benchmarks by Safari team



Example: initialization - device

```
const adapter = await navigator.gpu.requestAdapter();
const device = await adapter.requestDevice();
const context = canvas.getContext('gpupresent');
const swapChainDescriptor = {
    device,
    format: "bgra8unorm",
};
swapChain = context.configureSwapChain(swapChainDescriptor);
```

Example: initialization - swap chain

```
const adapter = await navigator.gpu.requestAdapter();
const device = await adapter.requestDevice();
const context = canvas.getContext('gpupresent');
const swapChainDescriptor = {
    device,
    format: "bgra8unorm",
};
swapChain = context.configureSwapChain(swapChainDescriptor);
```

Example: uploading vertex data

```
const verticesBufferDescriptor = { size: verticesArray.byteLength, usage: GPUBufferUsage.VERTEX };
const [verticesBuffer, verticesArrayBuffer] = device.createBufferMapped(verticesBufferDescriptor);
new Float32Array(verticesArrayBuffer).set(...); // vertex data
verticesBuffer.unmap();
const positionAttributeDescriptor = {
    shaderLocation: positionAttributeNum,  // [[attribute(0)]]
    offset: 0,
    format: "float4",
};
const vertexBufferDescriptor = {
    attributes: [positionAttributeDescriptor],
    arrayStride: vertexSize,
    stepMode: "vertex",
};
```

Example: declaring vertex formats

```
const verticesBufferDescriptor = { size: verticesArray.byteLength, usage: GPUBufferUsage.VERTEX };
const [verticesBuffer, verticesArrayBuffer] = device.createBufferMapped(verticesBufferDescriptor);
new Float32Array(verticesArrayBuffer).set(...); // vertex data
verticesBuffer.unmap();
const positionAttributeDescriptor = {
    shaderLocation: positionAttributeNum,  // [[attribute(0)]]
    offset: 0,
    format: "float4",
};
const vertexBufferDescriptor = {
    attributes: [positionAttributeDescriptor],
    arrayStride: vertexSize,
    stepMode: "vertex",
};
```

Quiz:

Is WebGPU an explicit API?

hint: what is explicit?

Feat: implicit memory

WebGPU:

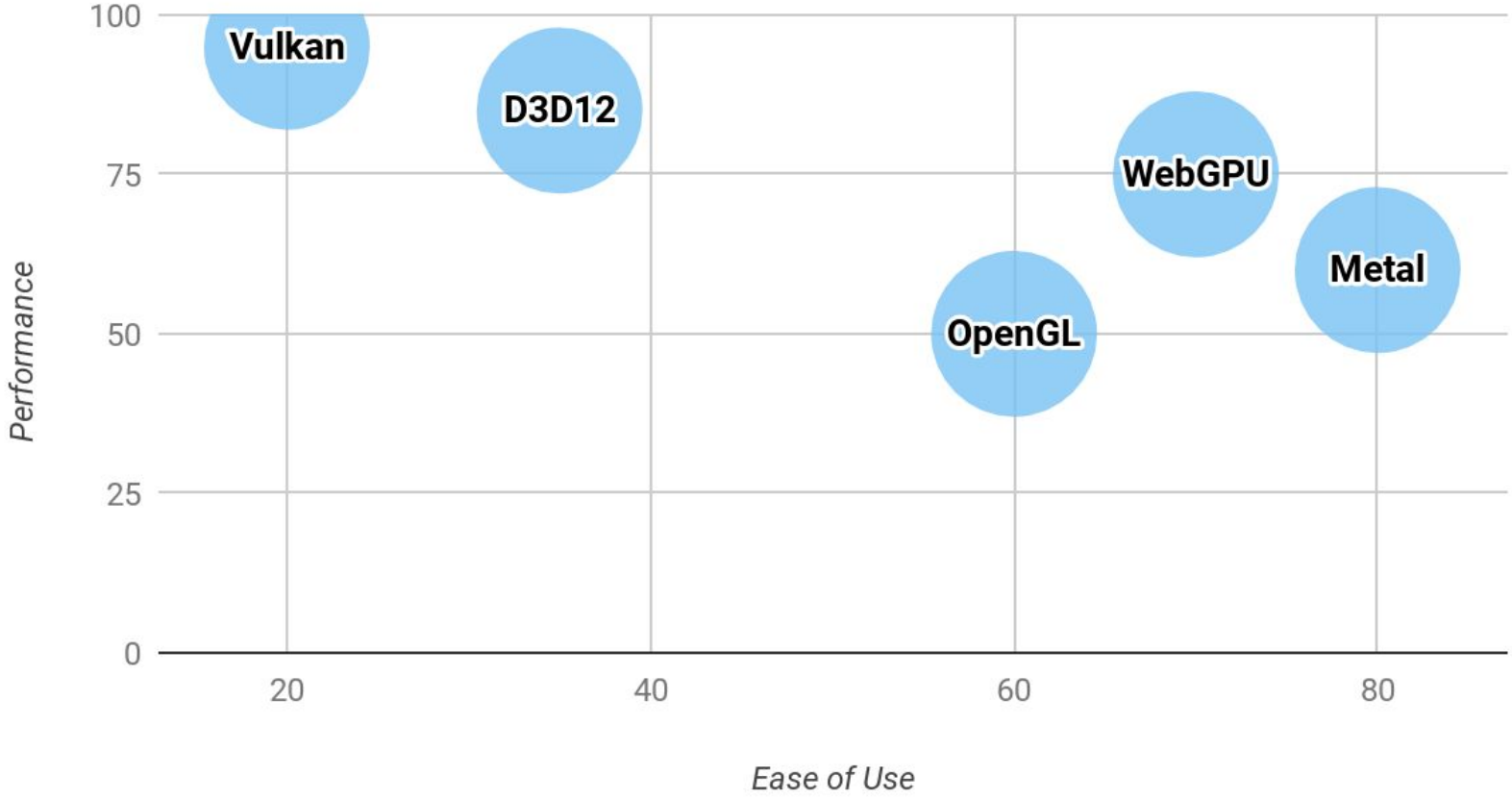
```
texture = device.createTexture({..});
```

Vulkan:

```
image = vkCreateImage();
reqs = vkGetImageMemoryRequirements();
memType = findMemoryType();
memory = vkAllocateMemory(memType);
vkBindImageMemory(image, memory);
```

Metal could be close to either

API trade-offs



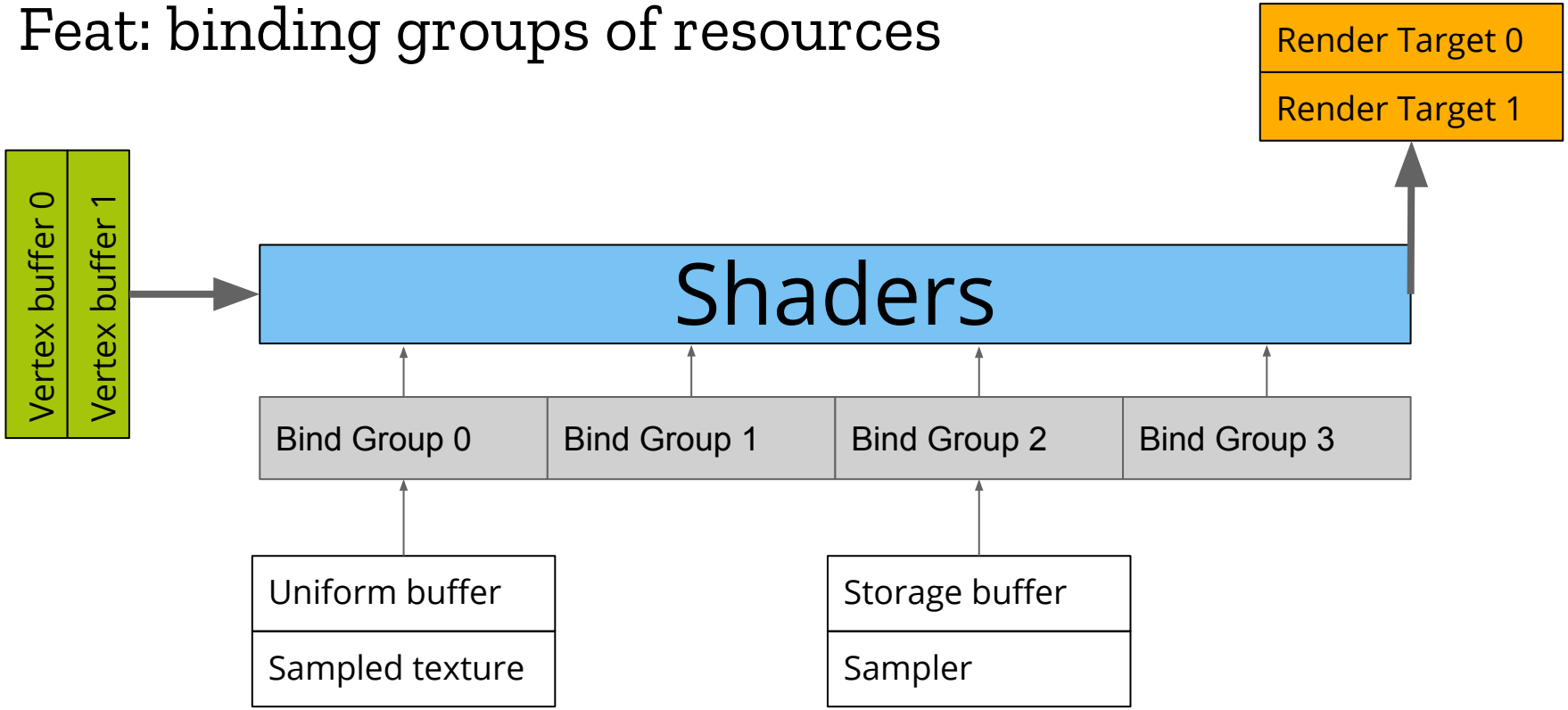
Example: declaring shader data

```
const transformBufferBindGroupLayoutBinding = {
    binding: transformBindingNum, // id[[(0)]]
    visibility: GPUShaderStage.VERTEX,
    type: "uniform-buffer",
};
const bindGroupLayout = device.createBindGroupLayout({
    bindings: [transformBufferBindGroupLayoutBinding],
});
const pipelineLayout = device.createPipelineLayout({
    bindGroupLayouts: [bindGroupLayout],
});
```

Example: instantiating shader data

```
const bindGroup = device.createBindGroup({
    layout: bindGroupLayout,
    bindings: [{
        binding: transformBindingNum,
        resource: {
            buffer: transformBuffer,
            offset: 0,
            size: transformSize,
        },
    }],
});
```

Feat: binding groups of resources



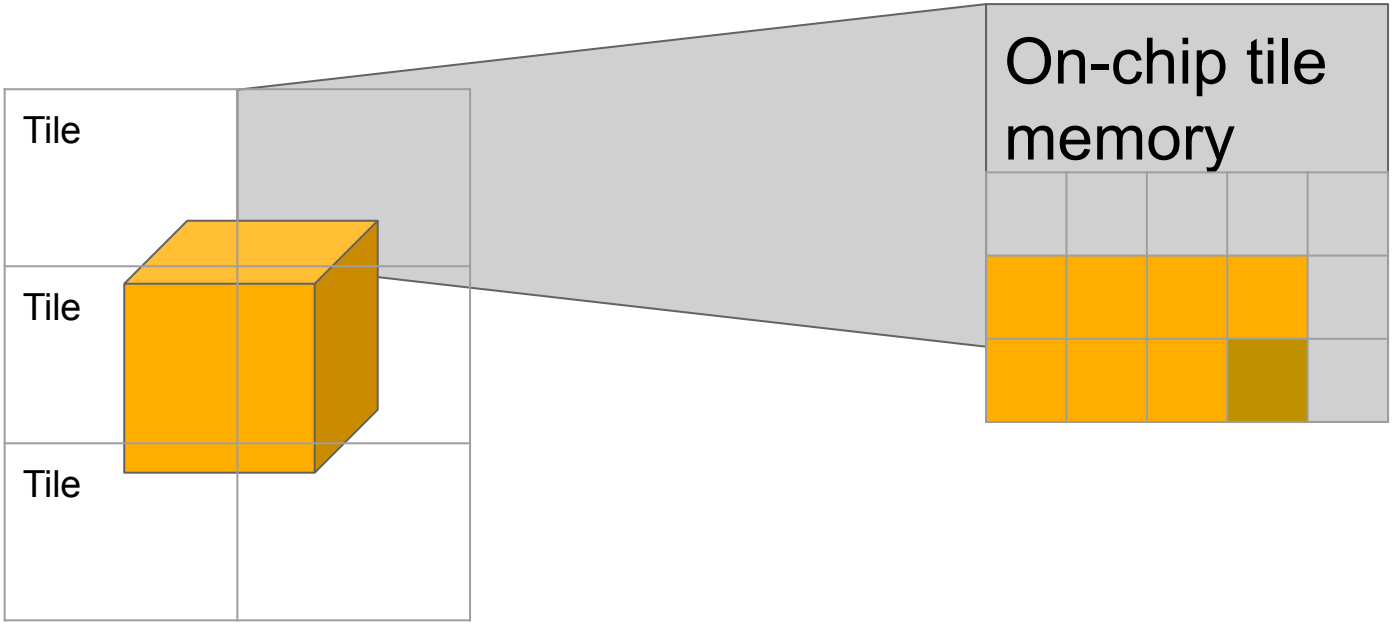
Example: creating the pipeline

```
const colorState = {
    format: "bgra8unorm",
    alphaBlend: { srcFactor: "one", dstFactor: "one", operation: "add" },
    colorBlend: { srcFactor: "src-alpha", dstFactor: "one-minus-src-alpha", operation: "add" },
    writeMask: GPUColorWrite.ALL,
};
const pipeline = device.createRenderPipeline({
    layout: pipelineLayout,
    vertexStage: { module: shaderModule, entryPoint: "vertex_main" },
    fragmentStage: { module: shaderModule, entryPoint: "fragment_main" },
    primitiveTopology: "triangle-list",
    colorStates: [colorState],
    depthStencilState: { depthWriteEnabled: true, depthCompare: "less", format: "depth24plus-stencil8" },
    vertexState: { vertexBuffers: [vertexBufferDescriptor] },
});
```

Example: render pass

```
  const passEncoder = commandEncoder.beginRenderPass({
    colorAttachments: [{
      attachment: swapChain.getCurrentTexture().createDefaultView()
      loadOp: “clear”,
      storeOp: “store”,
    }],
    depthStencilAttachment: { attachment: depthTexture.createDefaultView() }
  });
  passEncoder.setPipeline(pipeline);
  passEncoder.setVertexBuffers(0, [verticesBuffer], [0]);
  passEncoder.setBindGroup(0, bindGroup);
  passEncoder.draw(36, 1, 0, 0);
  passEncoder.endPass();
```

Feat: render passes



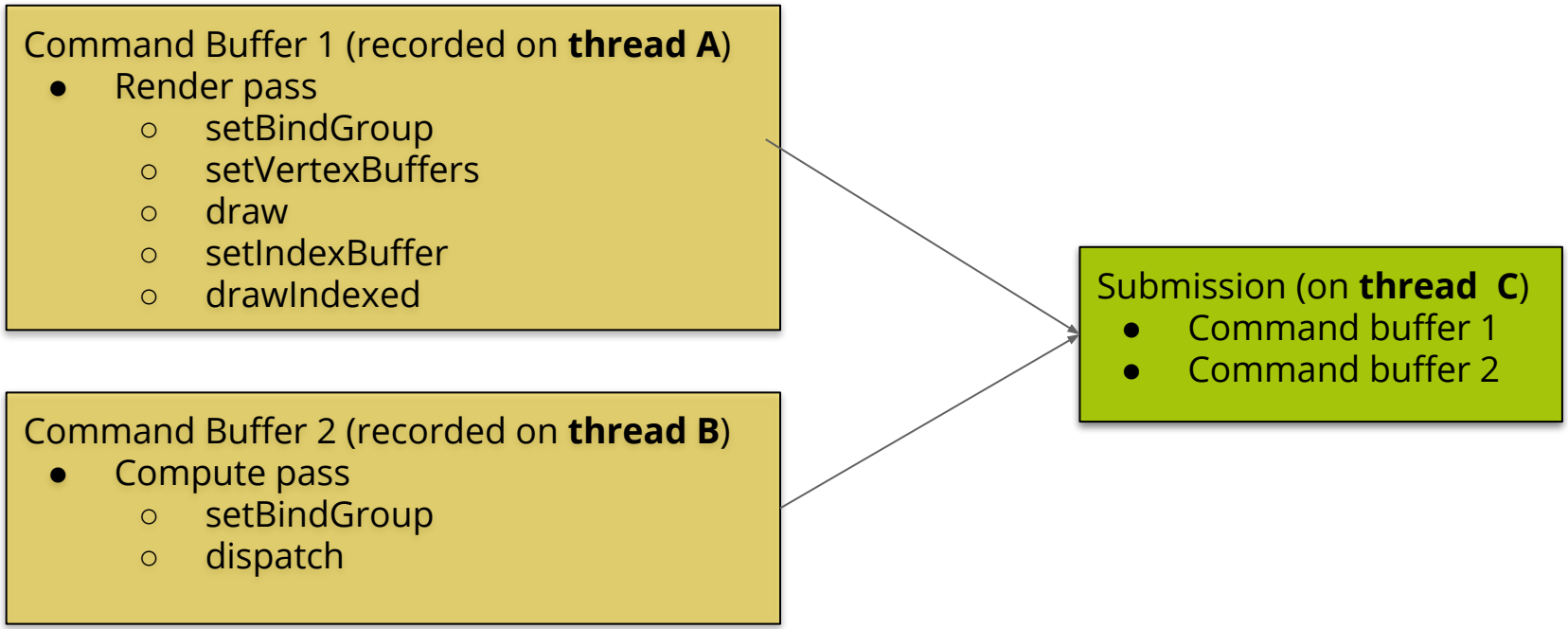
Example: state setting

```
const passEncoder = commandEncoder.beginRenderPass({
    colorAttachments: [{
        attachment: swapChain.getCurrentTexture().createDefaultView()
        loadOp: “clear”,
        storeOp: “store”,
    }],
    depthStencilAttachment: { attachment: depthTexture.createDefaultView() }
});
passEncoder.setPipeline(pipeline);
passEncoder.setVertexBuffers(0, [verticesBuffer], [0]);
passEncoder.setBindGroup(0, bindGroup);
passEncoder.draw(36, 1, 0, 0);
passEncoder.endPass();
```

Example: state setting

```
const passEncoder = commandEncoder.beginRenderPass({
    colorAttachments: [{
        attachment: swapChain.getCurrentTexture().createDefaultView()
        loadOp: “clear”,
        storeOp: “store”,
    }],
    depthStencilAttachment: { attachment: depthTexture.createDefaultView() }
});
passEncoder.setPipeline(pipeline);
passEncoder.setVertexBuffers(0, [verticesBuffer], [0]);
passEncoder.setBindGroup(0, bindGroup);
passEncoder.draw(36, 1, 0, 0);
passEncoder.endPass();
```

Feat: multi-threading



Example: work submission

```
const commandEncoder = device.createCommandEncoder();
commandEncoder.copyBufferToBuffer(stagingBuffer, 0, transformBuffer, 0, transformSize);
// record some passes here...
const commandBuffer = commandEncoder.finish();
device.defaultQueue.submit([commandBuffer]);
```

Feat: implicit barriers

Tracking resource usage

Command stream:

RenderPass-A {..}

Copy()

RenderPass-B {..}

ComputePass-C {..}

Texture usage

OUTPUT_ATTACHMENT

↓

COPY_SRC

↓

SAMPLED

↓

STORAGE

Buffer usage

STORAGE_READ

↓

COPY_DST

↓

VERTEX + UNIFORM

↓

STORAGE

Space for optimization

Quiz:

Is WSL the chosen shading language?

hint: what is WSL?

API: missing pieces

- Multi-queue
- Shading language

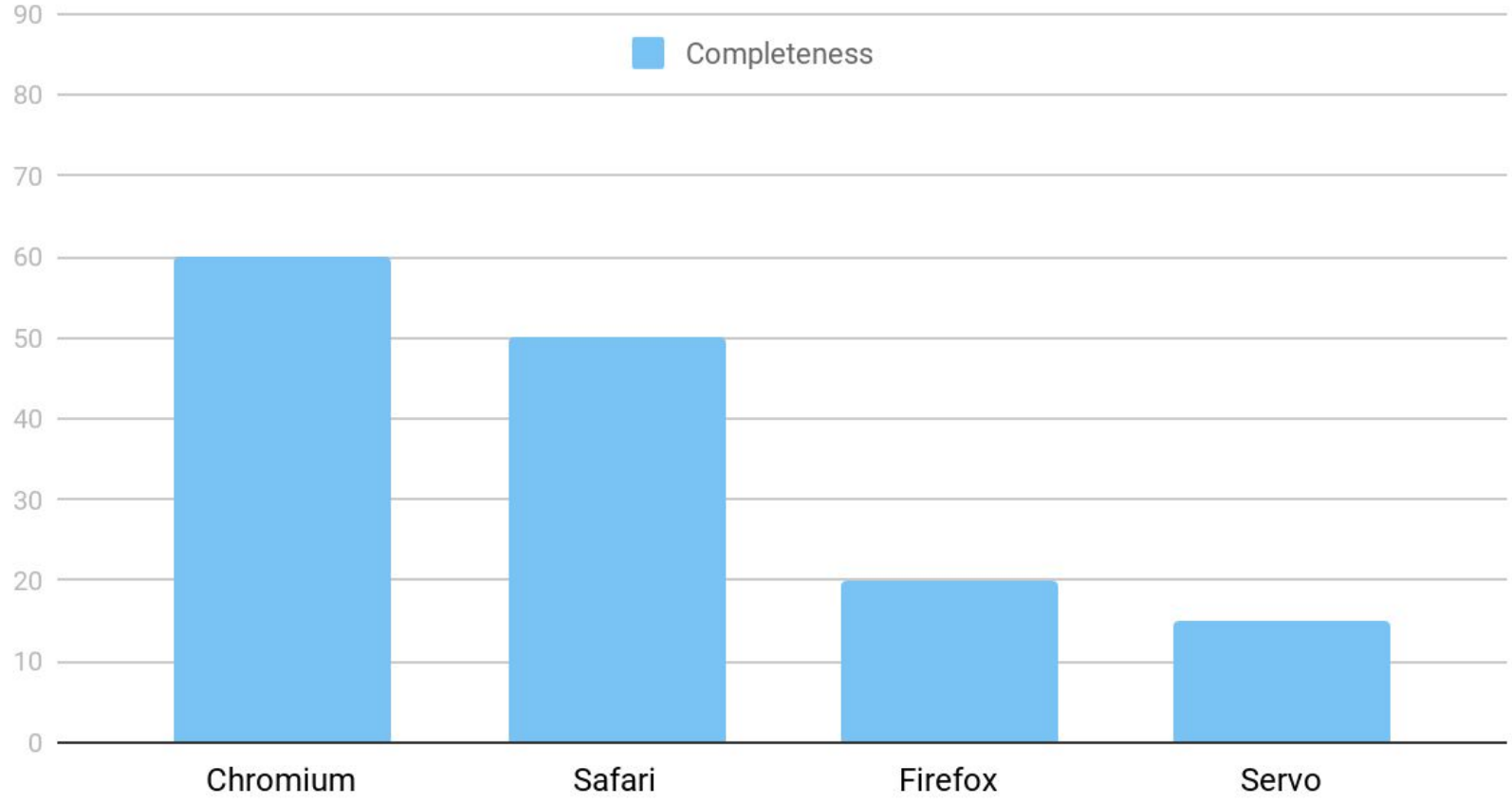
Incomplete:

- CPU transfers

Demo time!

Computing all the things, with Firefox

Are we WebGPU yet?



Quiz:

Is WebGPU only for the Web?

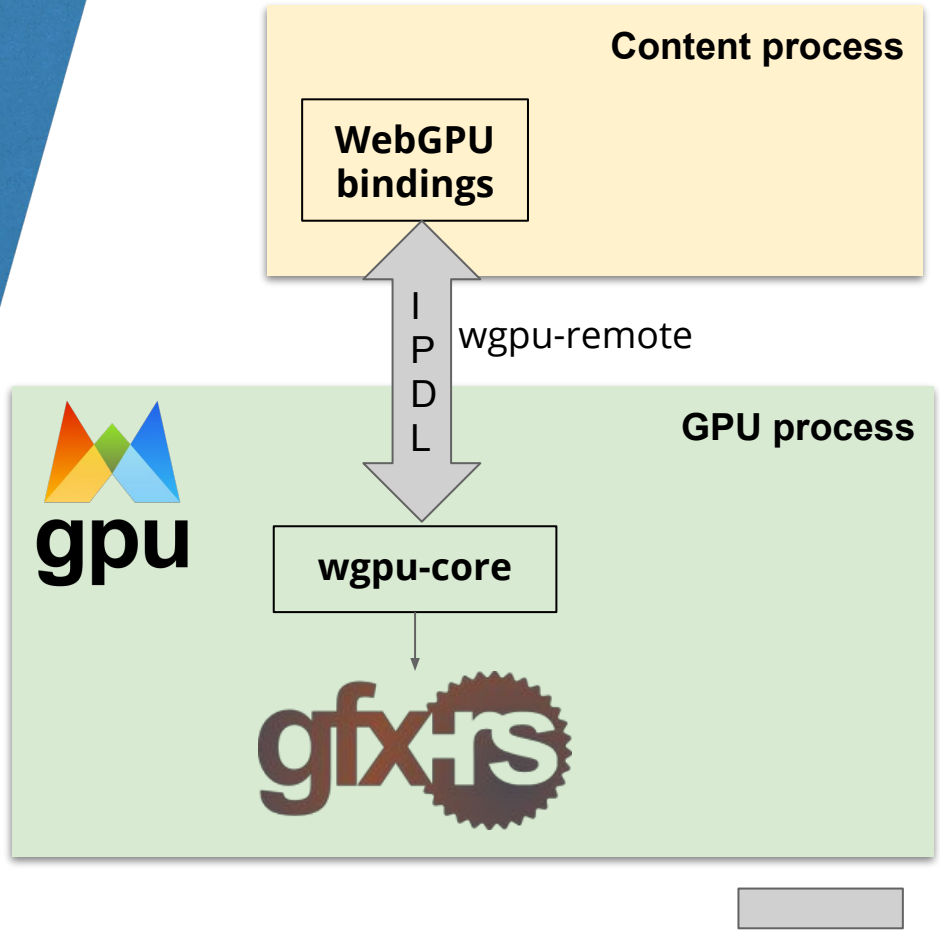
hint: what is explicit?

Act II

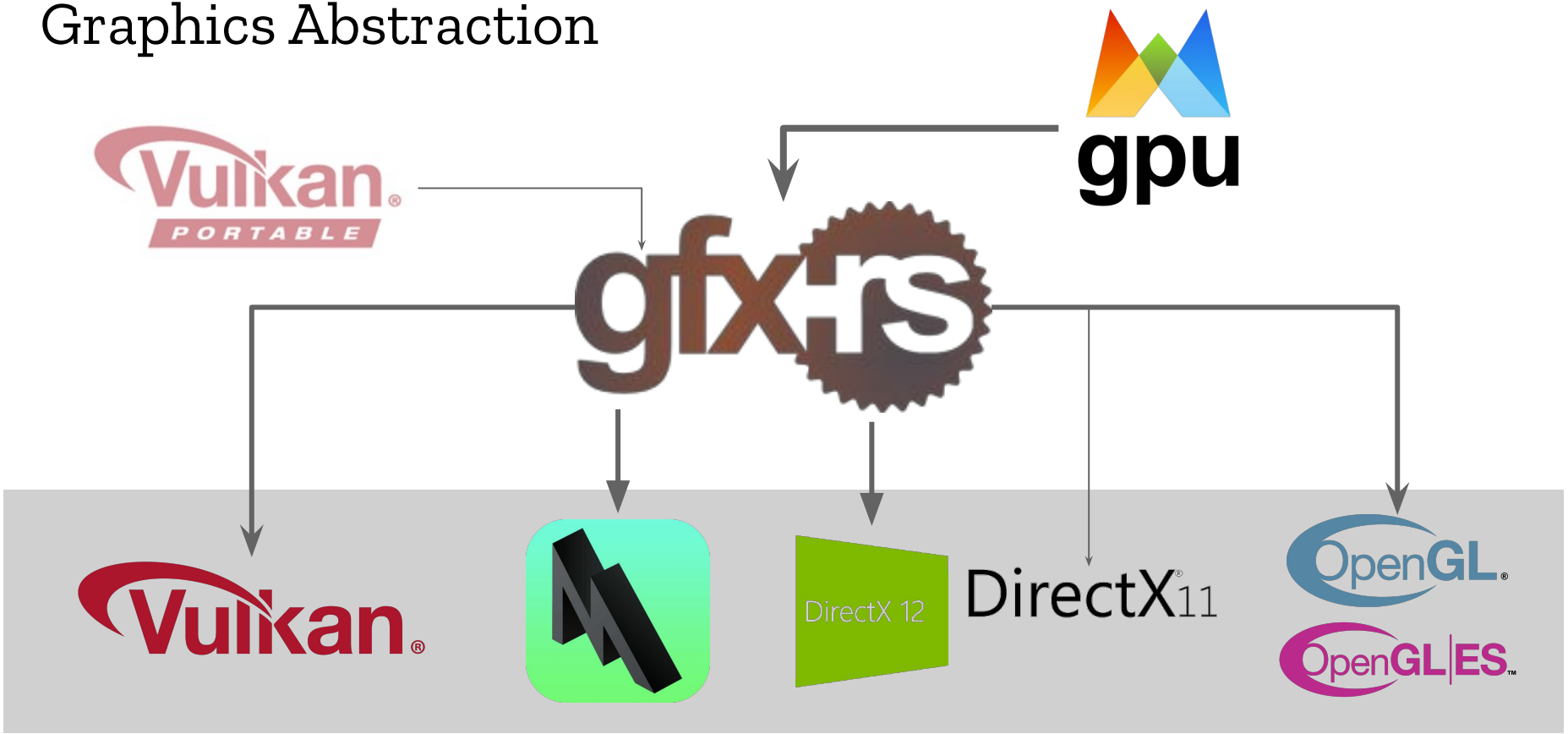
implementation

Architecture

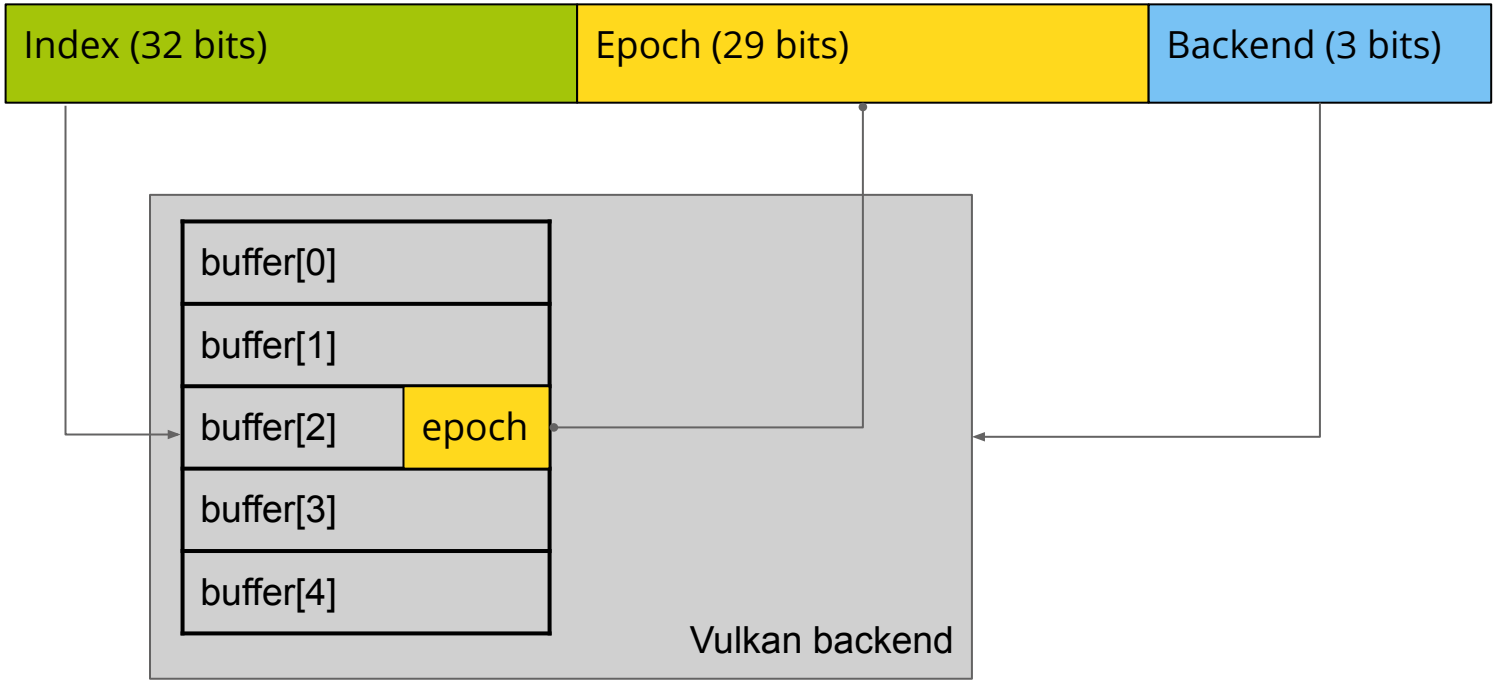
high-level



Graphics Abstraction



Identifiers and object storage



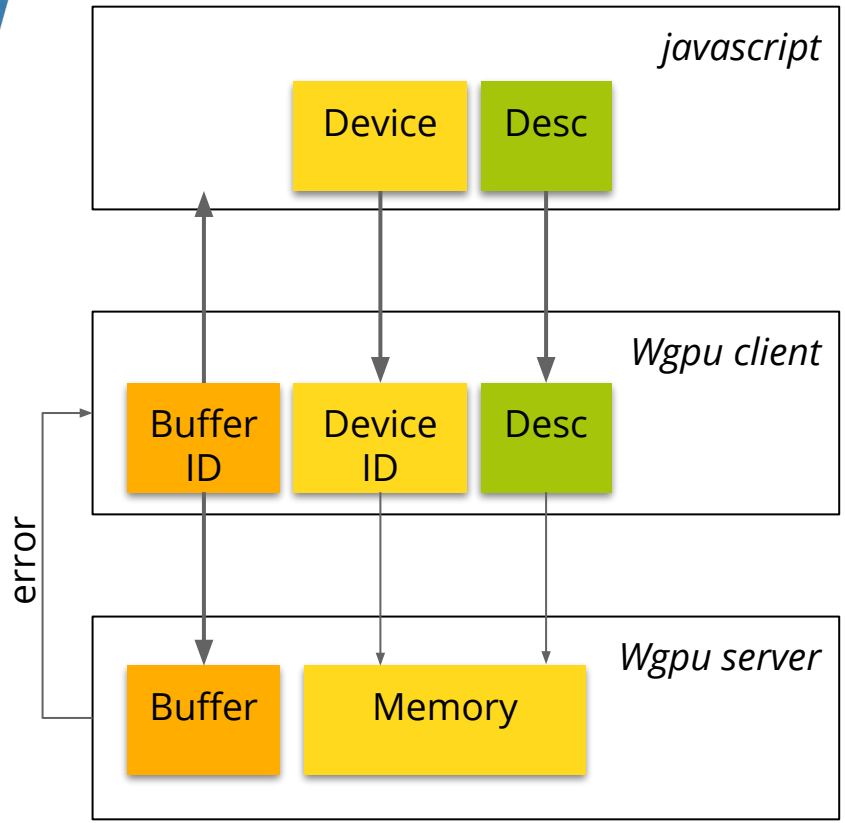
Backend polymorphism

```
Impl Context {
    pub fn device_create_buffer<B: GfxBackend>(&self, ...) { … }
}

#[no_mangle]
pub extern "C" fn wgpu_server_device_create_buffer(
        global: &Global, self_id: id::DeviceId, desc: &core::resource::BufferDescriptor, new_id: id::BufferId
) {
    gfx_select!(self_id => global.device_create_buffer(self_id, desc, new_id));
}
```

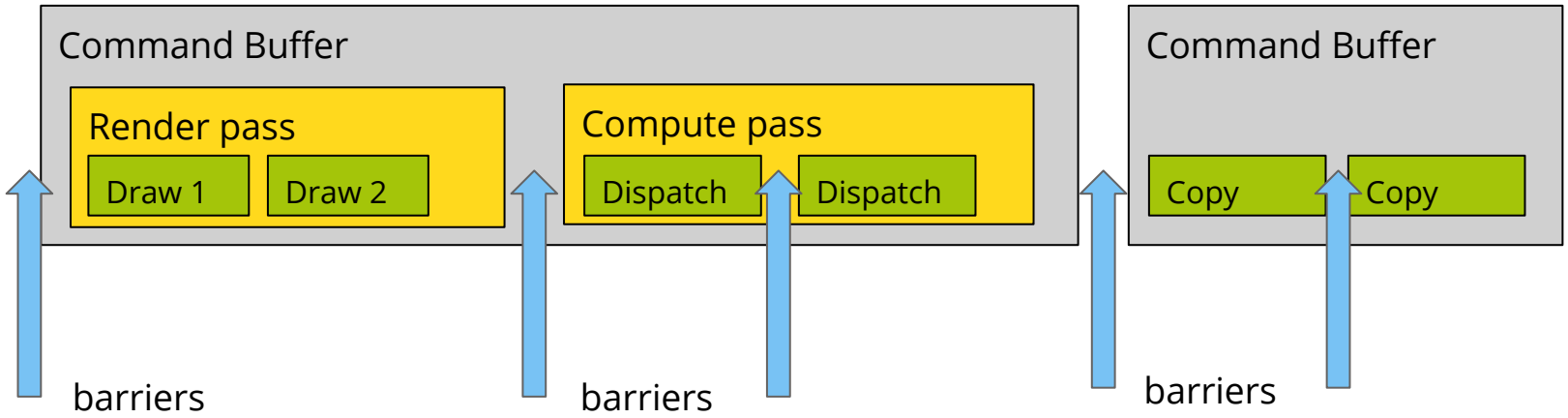
Buffer creation

and Id management



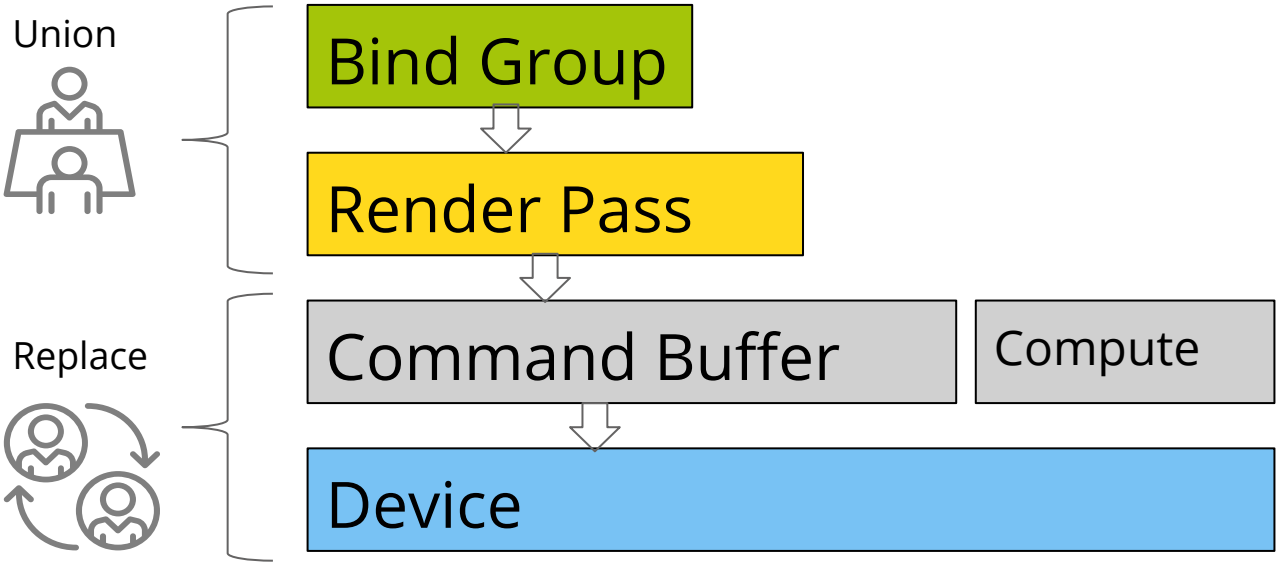
As-if synchronous

Usage tracking: sync scopes

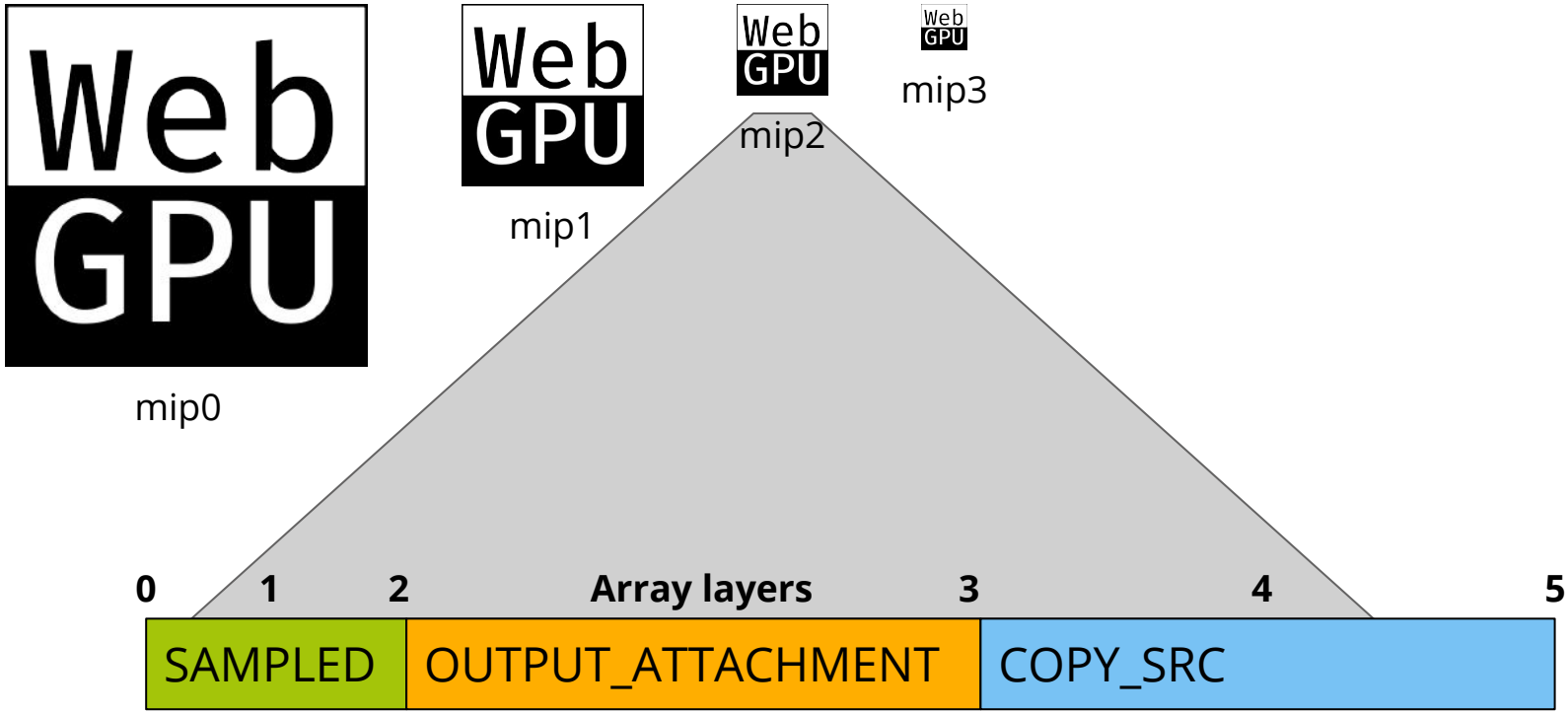


Old -> Expected -> New

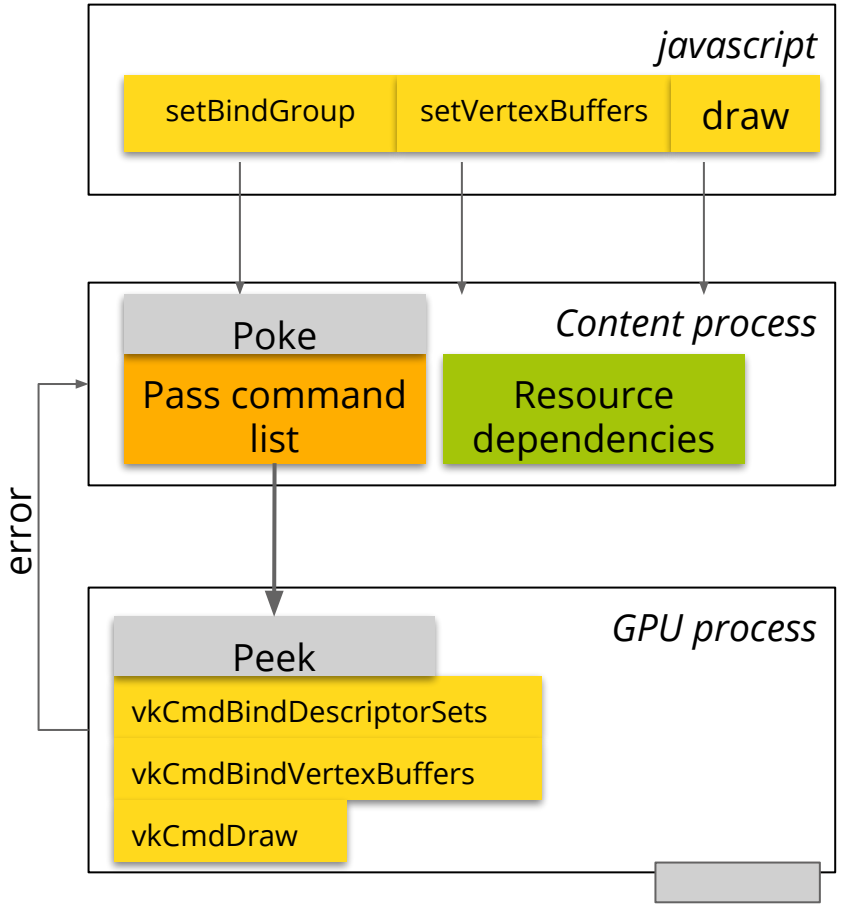
Usage tracking: merging



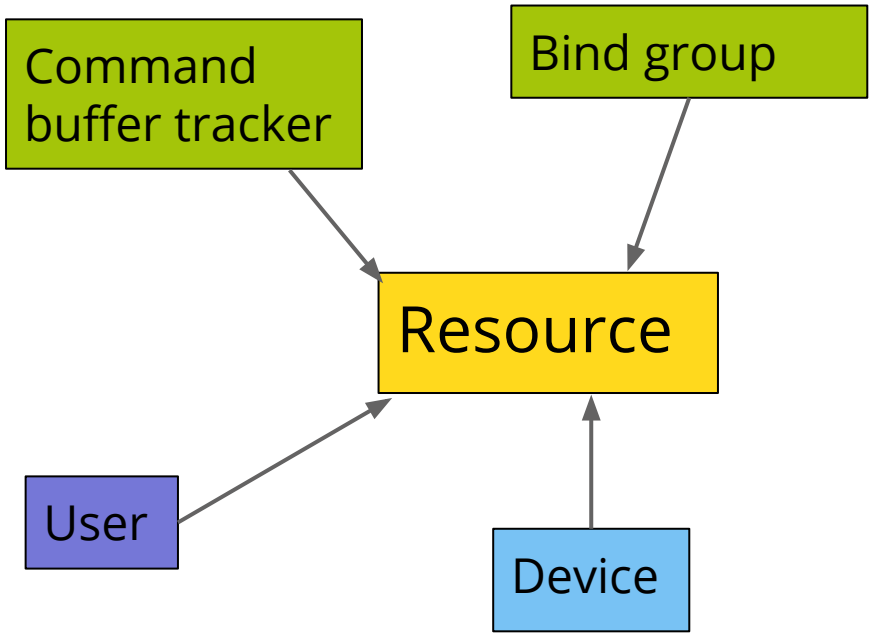
Usage tracking: sub-resources



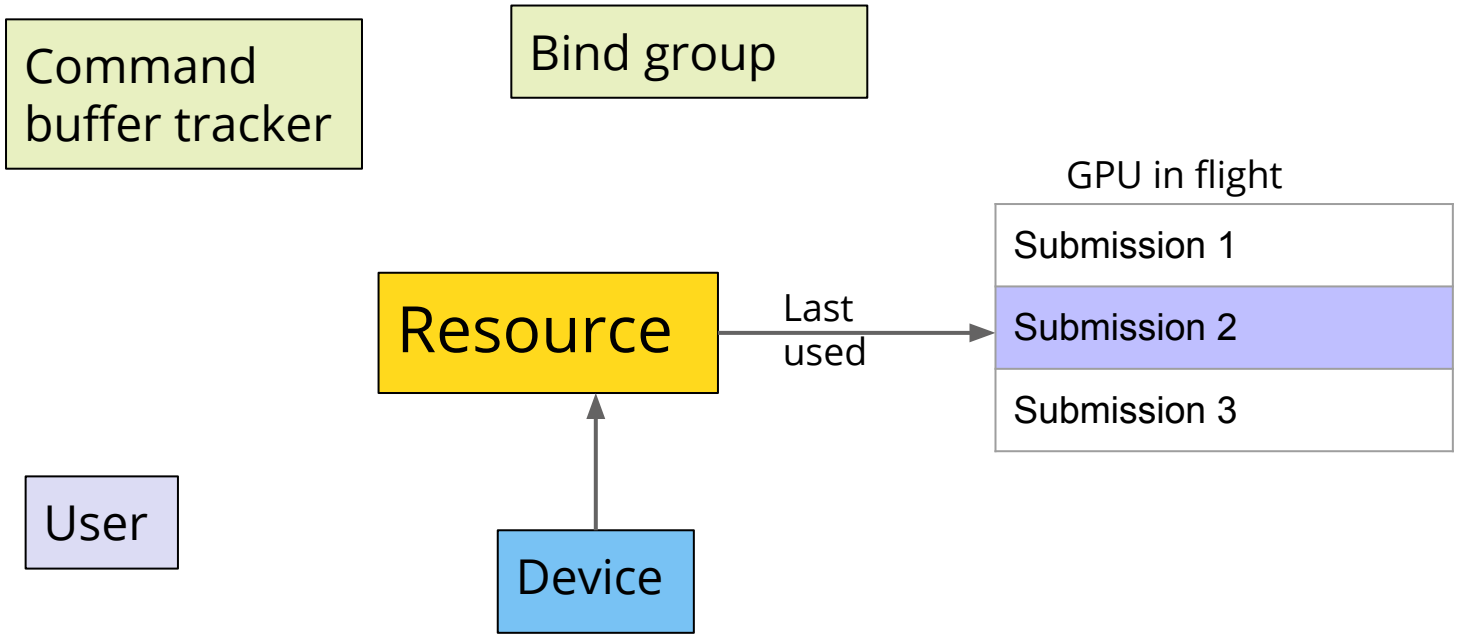
Pass recording



Lifetime tracking: ref counts



Lifetime tracking: freeing



Implementation: future work

- Presentation
- Error handling
- API coverage (including rendering)
- Platform coverage
- Optimization
- Sharing with WebRender

Act III

roundup

Conclusions

Risks and Challenges

Links

https://github.com/gpuweb/gpuweb - upstream spec

https://github.com/gfx-rs/wgpu - our implementation in Rust

https://dawn.googlesource.com/dawn - Google's implementation

http://kvark.github.io/web/gpu/gecko/2019/12/10/gecko-webgpu.html - details

https://webkit.org/blog/9528/webgpu-and-wsl-in-safari/ - Safari's features

"WebGPU - An Explicit Graphics API for the Web" - presentation by Austin Eng (Google)

https://archive.fosdem.org/2018/schedule/event/rust_vulkan_gfx_rs/ - Fosdem talk (2018)

moz://a

Contact us!

#gfx:mozilla.org - Mozilla graphics team

#wgpu:matrix.org - Rust WebGPU impl

#WebGPU:matrix.org - upstream API

moz://a

Thank You

gfx-rs/wgpu **community** (contributions)

Joshua Groves (wgpu reviews)

Boris Zbarsky (DOM reviews)

Jeff Muizelaar (wizdom)

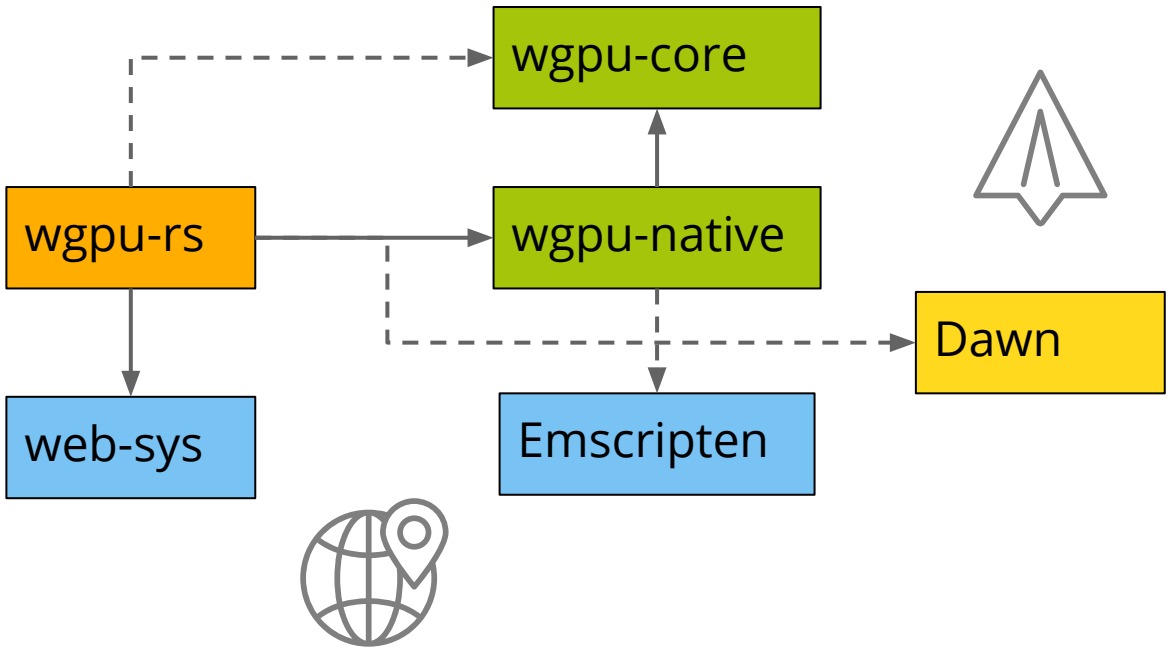
Jeffrey Gilbert (a bit of everything)

Corentin Wallez (feedback)

Bonus: wgpu-rs

the Rusty bindings

Wgpu-rs: project structure



Wgpu-rs: Rust features

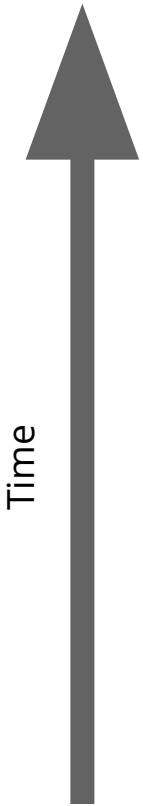
- Swapchain frame: &mut Swapchain
- CommandEncoder: !Sync, !Send
- Render/Compute pass: &mut CommandEncoder
- Render/Compute pass: &[Resource]

Bonus: Dawn Wire

the Google way

Deprecated slides

History



2020 ??? - agreement on the shading language?

2019 Sep - Gecko implementation start

2018 Sep - *wgpu* project kick-off

2018 Apr - agreement on the implicit barriers

2017 Jun - agreement on the binding model

2017 Feb - formation of W3C community group

2016 H2 - experiments by browser vendors

Architecture