WebGPU
An Explicit Graphics API for the Web

Austin Eng, Google*
enga@google.com

Many thanks to my teammates
Corentin Wallez, Kai Ninomiya, and many others at Google

*I do not officially represent Google

Review: Why use explicit APIs like Vulkan?

Many slides taken from Corentin's 2016 CIS 565 guest lecture
and Kai's 2017 CIS 565 guest lecture

Review: Why use explicit APIs like Vulkan?

- Explicit memory management
- Multithreading
- Async compute
- ...and more!

Texture resizing in OpenGL

User resizing texture:

- Resize the texture
- Use it
- :D

Driver resizing texture:

- Allocate new memory
- Use new memory
- :D

Texture resizing in OpenGL

User resizing texture:

- Resize the texture
- Use it
- :D

Driver resizing texture:

- Allocate new memory
  - Insert fence
  - Check the fence every frame?
  - Garbage collect memory
- Use new memory
- :/

Texture resizing in OpenGL

User resizing texture:

- Resize the texture
- Use it
- :D

Driver resizing texture:

- Allocate new memory
    - Insert fence
    - Check the fence every frame?
    - Garbage collect memory
        - Dirty uniforms passed to shaders
        - Dirty framebuffers
        - Dirty texture buffers
- Use new memory
- :(

Why: Predictable behavior and performance

Applications can:

- Control when expensive operations happen
- Have low variance frame timing (VR)
- Be smarter than the OpenGL driver

Why: Consoles

Graphics development on console:

- Direct access to the hardware
- Manual memory management
- Getting to that last 1% of performance
- Multithreading

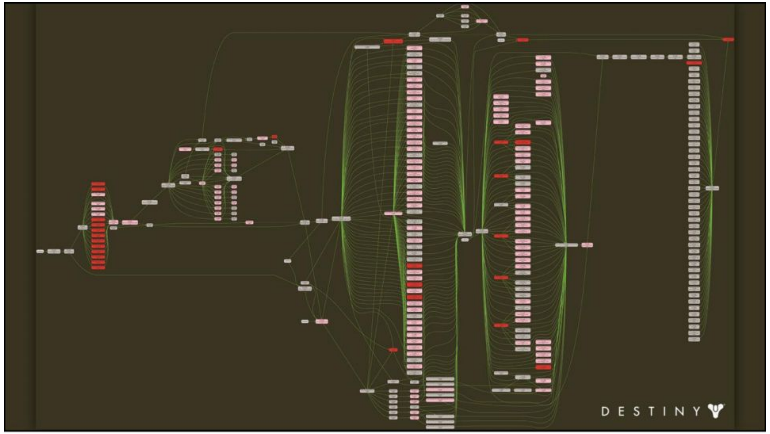
Developers want that on PC too.

Why: Multithreading

[Destiny's Multi-threaded Renderer Architecture by Natalya Tatarchuk]

- Simulation
- Determine views
  (for rendering, shadow-mapping, etc.)
- Compute visibility
- Extract data for rendering
- Generate draw calls

(decouple)



Command buffers enable multithreading

Thread 1

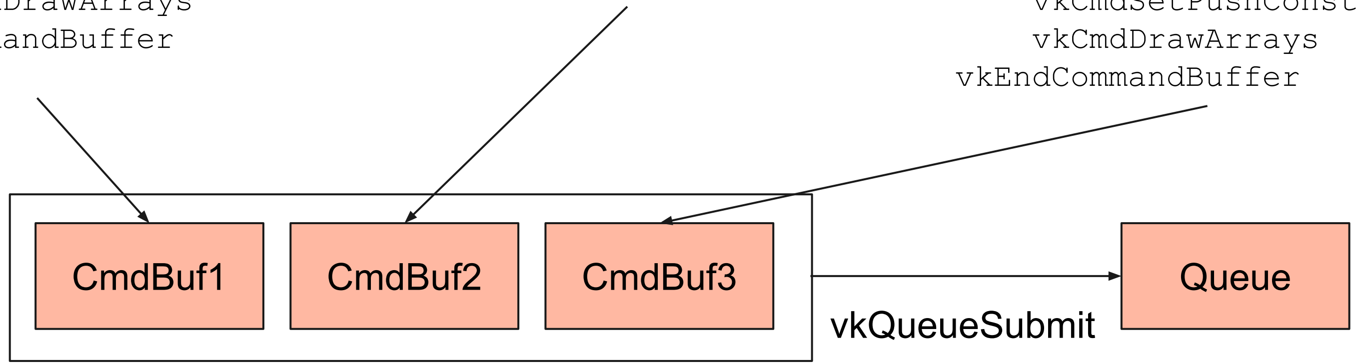
```
vkBeginCommandBuffer
    vkCmdSetPipeline
    vkCmdDrawArrays
    vkCmdSetScissor
    vkCmdDrawArrays
vkEndCommandBuffer
```

Thread 2

```
vkBeginCommandBuffer
    vkCmdSetPipeline
    vkCmdDrawArrays
vkEndCommandBuffer
```

Thread 3

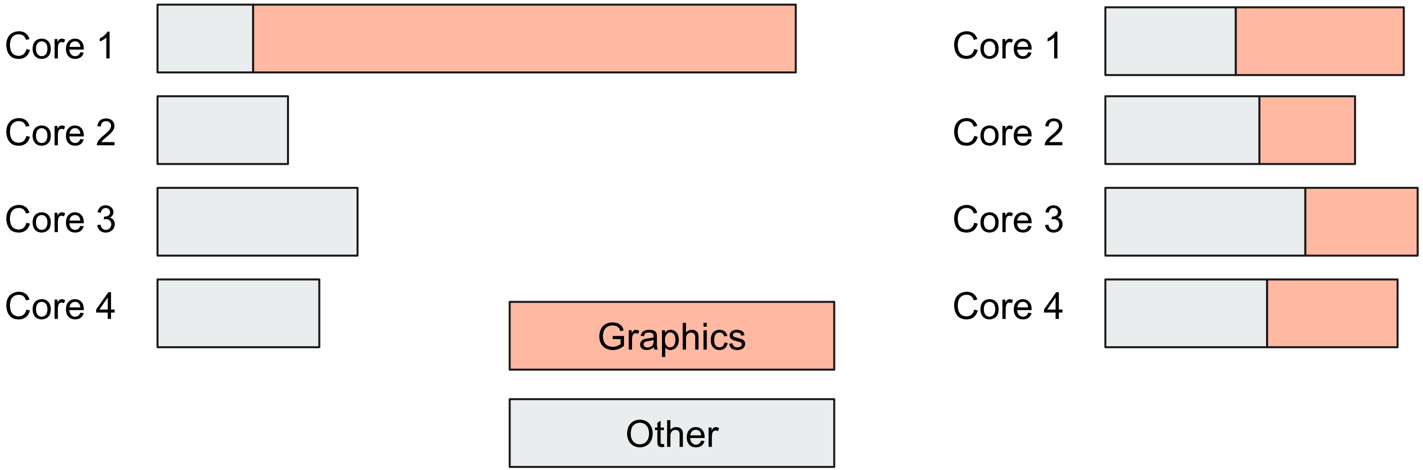
```
vkBeginCommandBuffer
    vkCmdSetPipeline
    vkCmdDrawArrays
    vkCmdSetPipeline
    vkCmdSetPushConstants
    vkCmdDrawArrays
vkEndCommandBuffer
```



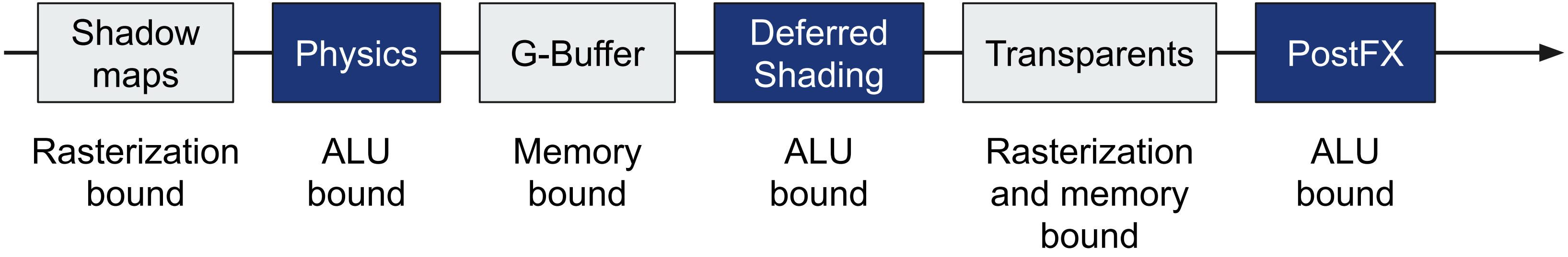
Why: Multithreading

Single-threaded APIs

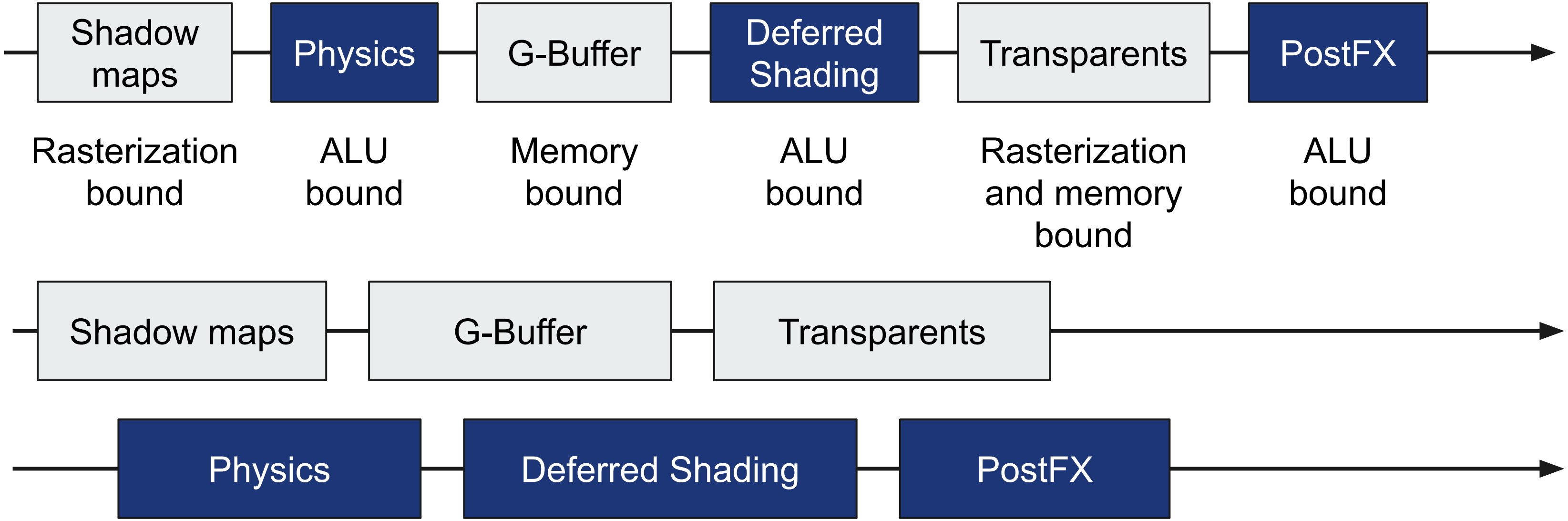
Multi-threaded APIs



Why: Async Compute



Why: Async Compute



Case Study: Vulkan Grass Rendering (project 6)

We almost have async compute! How can we do better?

- Compute:
  - Apply forces
  - Update `Blade` buffer
  - Cull blades
- Memory barrier (compute->graphics)
  Waits for compute pipeline to finish.
- Graphics: Rasterize + Tessellate

Case Study: Vulkan Grass Rendering (project 6)

- Decouple physics and culling
  - Compute expensive physics for several frames in the future simultaneously
  - This step is camera-independent

- Compute culled blades for the next frame
- Memory barrier (compute->graphics)

  Does not wait. Blades were culled while rendering the previous frame.
- Graphics: Rasterize + Tessellate

Explicit Graphics APIs on the Web

https://github.com/gpuweb/gpuweb

A Few Goals:

- Security & Stability
  - A website can’t be allowed to read your data
  - Native APIs allow unsafe operations and undefined behavior
- Portability
  - Create an API to map onto D3D12, Metal, and Vulkan
  - The Web should work the same everywhere, no matter what platform
- Fast
  - Multithreading
  - WebAssembly
  - Web Workers

It’s happening, but it’s hard...

- See Kai’s presentation to learn about the process of designing this API
- Reaching agreement with the other browser vendors takes a lot of time and discussion

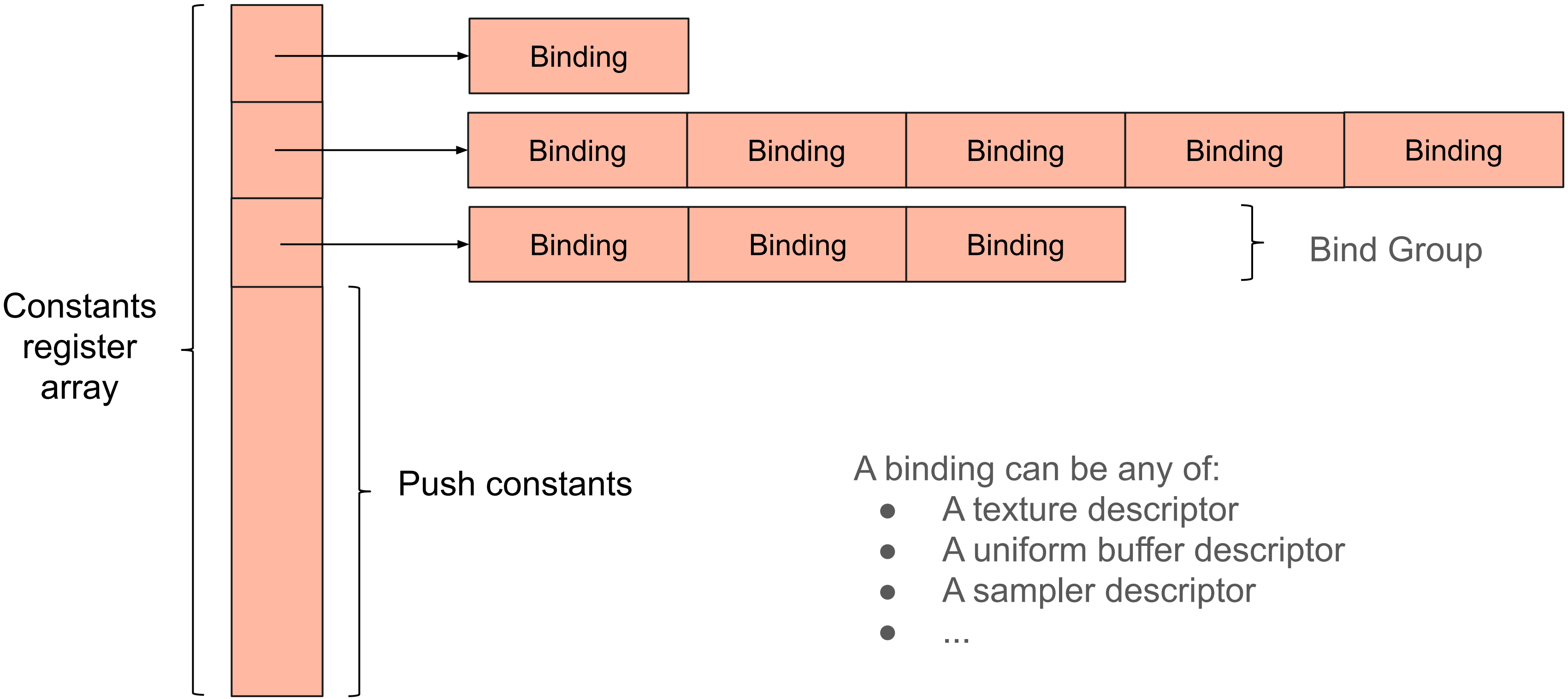
Dawn, a WebGPU implementation*

API overview, examples, assorted details, and cool things

https://dawn.googlesource.com/dawn

*API subject to change

API Overview: Resource Binding


Binding
Binding
Binding
Binding
Binding
Binding
Binding
Binding
Binding
Bind Group
Constants
register
array
Push constants
A binding can be any of:
A texture descriptor
A uniform buffer descriptor
A sampler descriptor
...

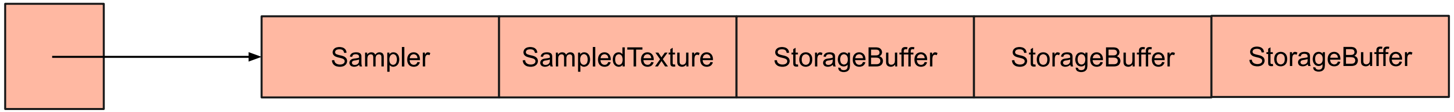
Resource Binding

Very similar to Vulkan:

- Pipeline layouts, composed of bind group layouts, define the structure of resource bindings for a pipeline
- Bind groups are created from bind group layouts and contain references to resources (buffer views, texture views, etc.)
- Bind groups are set on a pipeline when recording a command buffer

Resource Binding in Dawn

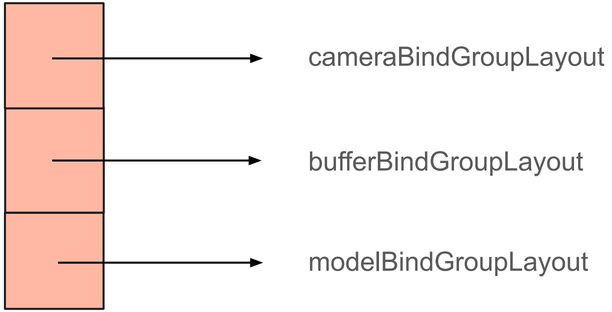
```
// Create bind group layouts
dawn::BindGroupBinding bufferBindings[] = {
        { 0, dawn::ShaderStageBit::Compute, dawn::BindingType::Sampler },        // (binding = 0) G-buffer sampler
        { 1, dawn::ShaderStageBit::Compute, dawn::BindingType::SampledTexture }, // (binding = 1) G-buffer
        { 2, dawn::ShaderStageBit::Compute, dawn::BindingType::StorageBuffer },  // (binding = 2) index buffer
        { 3, dawn::ShaderStageBit::Compute, dawn::BindingType::StorageBuffer },  // (binding = 3) vertex buffer
        { 4, dawn::ShaderStageBit::Compute, dawn::BindingType::StorageBuffer },  // (binding = 4) output color buffer
};
dawn::BindGroupLayoutDescriptor bufferBindGroupLayoutDesc { nullptr, 5, bufferBindings };
dawn::BindGroupLayout bufferBindGroupLayout = device.CreateBindGroupLayout(&bufferBindGroupLayoutDesc);
```




```
// Create other bind group layouts...
```

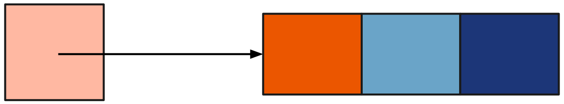
Resource Binding in Dawn

```
// Create pipeline

dawn::BindGroupLayout bindGroupLayouts[] = {
      cameraBindGroupLayout,     // (set = 0)
      bufferBindGroupLayout,     // (set = 1)
      modelBindGroupLayout,      // (set = 2)
};
dawn::PipelineLayoutDescriptor pipelineLayoutDesc { nullptr, 3, bindGroupLayouts };
dawn::PipelineLayout pipelineLayout = device.CreatePipelineLayout(&pipelineLayoutDesc);

dawn::ShaderModule csModule = utils::CreateShaderModule(device, dawn::ShaderStage::Compute,
kComputeShaderString);
dawn::ComputePipelineDescriptor computePipelineDesc{nullptr, pipelineLayout, csModule, "main"};
dawn::ComputePipeline computePipeline = device.CreateComputePipeline(&computePipelineDesc);
```



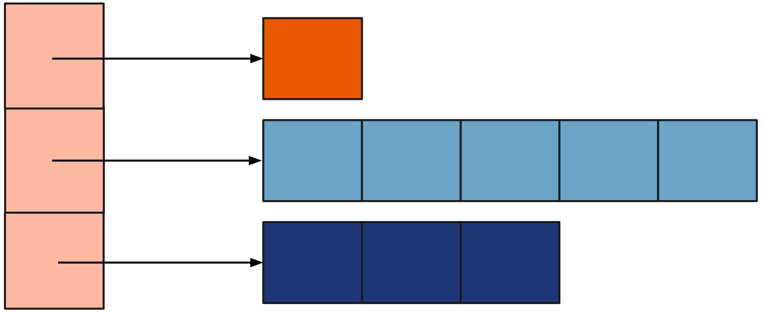
Resource Binding in Dawn

```
// Create camera bind group
dawn::BindGroupBinding bindings[] = {
      { 0, dawn::BindingType::BufferView, cameraBufferView },
};
dawn::BindGroupDescriptor bindGroupDesc { cameraBindGroupLayout, 1, bindings }
dawn::BindGroup cameraBindGroup = device.CreateBindGroup(&bindGroupDesc);

// Create bind groups for all models
for (Model* model : models) {
      dawn::BindGroupBinding bindings[] = {
            { 0, dawn::BindingType::BufferView, model->bufferView },
            { 1, dawn::BindingType::TextureView, model->textureView },
            { 2, dawn::BindingType::Sampler, model->sampler },
      };
      dawn::BindGroupDescriptor bindGroupDesc { modelBindGroupLayout, 3, bindings }
      model->modelBindGroup = device.CreateBindGroup(&bindGroupDesc);
}
```



Resource Binding in Dawn

```
// Set bind groups
dawn::ComputePassEncoder pass = builder.BeginComputePass();
pass.SetComputePipeline(computePipeline);
pass.SetBindGroup(0, cameraBindGroup);
for (ModelGroup* modelGroup : modelGroups) {
       pass.SetBindGroup(1, modelGroup->bufferBindGroup);
       for (Model* model : modelGroup->GetModels()) {
              pass.SetBindGroup(2, model->modelBindGroup);
              pass.Dispatch(1280, 960, 1);
       }
}
pass.EndPass();
```



API Overview: Pipelines

Render / Compute Pipelines

A big object that defines fixed-function state and format of the inputs and outputs:

- Pipeline layout (set of bind group layouts)
- Compiled shaders

Render pipelines only:

- Various state
    - Blending, depth, stencil, input format, etc.
- Framebuffer attachment formats

Creating a Render Pipeline

```
// Create depth stencil state
dawn::DepthStencilStateDescriptor depthStencilStateDesc;
depthStencilStateDesc.depthWriteEnabled = true;
depthStencilStateDesc.depthCompare = dawn::CompareFunction::Less;
dawn::DepthStencilState depthStencilState =
        device.CreateDepthStencilState(&depthStencilStateDesc);

// Create vertex input and attribute state
dawn::VertexAttributeDescriptor vertexAttribs[] = {
        {0, 0, 0, dawn::VertexFormat::FloatR32G32B32A32},
        {1, 1, 0, dawn::VertexFormat::FloatR32}};
dawn::VertexInputDescriptor vertexInputs[] = {
        {0, 0, dawn::InputStepMode::Vertex},
        {1, 0, dawn::InputStepMode::Instance}};
dawn::InputStateDescriptor inputStateDesc;
inputStateDesc.indexFormat = dawn::IndexFormat::UInt32;
inputStateDesc.attributes = vertexAttribs;
inputStateDesc.numAttributes = 2;
inputStateDesc.inputs = vertexInputs;
inputStateDesc.numInputs = 2;

// Create attachment states
dawn::Attachment colorAttachments[] = {{ dawn::TextureFormat::R8G8B8A8Uint }};
dawn::Attachment depthStencilAttachment { dawn::TextureFormat::D32FloatS8Uint };
dawn::AttachmentsState attachmentsState { colorAttachments, 1, depthStencilAttachment };
```

```
// Create pipeline layout
dawn::PipelineLayoutDescriptor pipelineLayoutDesc;
pipelineLayoutDesc.numBindGroupLayouts  =  4;
pipelineLayoutDesc.bindGroupLayouts = bindGroupLayouts;
dawn::PipelineLayout pipelineLayout =
        device.CreatePipelineLayout(&pipelineLayoutDesc);

// Create render pipeline
dawn::RenderPipelineDescriptor renderPipelineDesc;
renderPipelineDesc.vertexStage =
        dawn::PipelineStageDescriptor { vsModule, "main" };
renderPipelineDesc.fragmentStage =
        dawn::PipelineStageDescriptor { fsModule, "main" };
renderPipelineDesc.primitiveTopology = dawn::PrimitiveTopology::TriangleList;
renderPipelineDesc.depthStencilState = depthStencilState;
renderPipelineDesc.inputState = inputState;
renderPipelineDesc.attachmentsState = attachmentsState;

dawn::RenderPipeline pipeline =
        device.CreateRenderPipeline(&renderPipelineDesc);
```

API Overview: Command Submission

Render/Compute Passes

- Encode a group of commands into the command buffer
  Render passes: setVertexBuffers(...), draw(...), etc.
  Compute passes: dispatch(...)

  Render passes:
- Contain attachment descriptions
  - g-buffers, color buffers, etc.

Implicit Resource Transitions

- Resources must not change usage within a pass
  ex.) Transition from vertex to uniform buffer
- Resources are synchronized:
  - At pass boundaries, to transition usage
  - For UAVs between dispatch() calls
- Implicit resource transitions make application development significantly easier
- Explicit transitions are faster, but forgetting them leads to undefined behavior

Example Render / Compute Passes

```
// Example command buffer for a particle simulation
dawn::CommandBuffer createCommandBuffer(
                const dawn::RenderPassDescriptor& renderPass,
                uint32_t i) {
        static const uint32_t zero = 0u;
        auto& bufferDst = particleBuffers[(i + 1) % 2]; // ping pong between these
        dawn::CommandBufferBuilder builder = device.CreateCommandBufferBuilder();
        {
                dawn::ComputePassEncoder pass = builder.BeginComputePass();
                pass.SetComputePipeline(computePipeline);
                pass.SetBindGroup(0, bindGroups[i]); // This where bufferDst is bound for writing the particle attributes
                pass.Dispatch(kNumParticles, 1, 1);
                pass.EndPass();
        }
        {
                dawn::RenderPassEncoder pass = builder.BeginRenderPass(renderPass);
                pass.SetRenderPipeline(renderPipeline);
                pass.SetVertexBuffers(0, 1, &bufferDst, &zero); // Bind bufferDst as a vertex buffer for particles
                pass.SetVertexBuffers(1, 1, &modelBuffer, &zero);
                pass.DrawArrays(3, kNumParticles, 0, 0);
                pass.EndPass();
        }
        return builder.GetResult();
}
```

```
static uint32_t pingpong = 0;
void frame() {
        dawn::CommandBuffer commandBuffer =
                createCommandBuffer(renderPass, pingpong);
        queue.Submit(1, &commandBuffer);
        pingpong = (pingpong + 1) % 2;
}
```

Implementing Timeline Fences

(simplified)

And cool things I’ve learned in my first few months about interprocess communication and GPU servicification.

What is a Fence?

- A synchronization primitive used to wait for execution on the GPU to complete
- For WebGPU, we've settled on "numerical fences"
  - Monotonically increasing values indicate a timestamp in GPU execution history.
    Hence, the name "timeline fences"

What is a Fence?

```
queue.Submit(1, &commands1); // submit commands1
queue.Signal(fence, 1u);
queue.Submit(1, &commands2); // submit commands2
queue.Signal(fence, 2u);
queue.Submit(1, &commands3); // submit commands3
queue.Signal(fence, 3u);

// Some time later...
uint64_t completedValue = fence.GetCompletedValue();

// Suppose completedValue == 2.
// That means that commands1 and commands2 have finished executing.
// commands3 may not have finished executing.
```

Implementing Timeline Fences in Dawn

```
struct Fence {
	uint64_t signalValue = 0;
	uint64_t completedValue = 0;
};

struct Queue {
	struct SignaledFence {
		Fence fence;
		VkFence nativeFence;
		uint64_t signalValue;
	};

	std::vector<SignaledFence> signaledFences;
};
```

```
void Queue::Signal(Fence fence, uint64_t signalValue)
{
	if (signalValue <= fence.signalValue) {
		// Validation error: Fence values must
		// increase monotonically
		return;
	}
	fence.signalValue = signalValue;
	VkFence nativeFence;
	vkCreateFence(device, createInfo, nullptr,
	              &nativeFence);
	vkQueueSubmit(queue, 0, nullptr, nativeFence);
	signaledFences.push_back(
	  SignaledFence{
	    fence, nativeFence, signalValue});
}
```

Implementing Timeline Fences in Dawn

```
struct Fence {
	uint64_t signalValue = 0;
	uint64_t completedValue = 0;
};

struct Queue {
	struct SignaledFence {
		Fence fence;
		VkFence nativeFence;
		uint64_t signalValue;
	};

	std::vector<SignaledFence> signaledFences;
};
```

A Fence stores the last signaled value and the value that has completed execution on the GPU

```
ence fence, uint64_t signalValue)

ue <= fence.signalValue) {
dation error: Fence values must
ease monotonically
		return;
	}
	fence.signalValue = signalValue;
	VkFence nativeFence;
	vkCreateFence(device, createInfo, nullptr,
	              &nativeFence);
	vkQueueSubmit(queue, 0, nullptr, nativeFence);
	signaledFences.push_back(
	  SignaledFence{
	    fence, nativeFence, signalValue});
}
```

Implementing Timeline Fences in Dawn

When we signal a Fence, create a native vkFence and signal it on a queue.

Add the fence to a list of signaled fences we will check later

```
struct Queue {
      struct SignaledFence {
            Fence fence;
            VkFence nativeFence;
            uint64_t signalValue;
      };

      std::vector<SignaledFence> signaledFences;
};
```

```
void Queue::Signal(Fence fence, uint64_t signalValue)
{
      if (signalValue <= fence.signalValue) {
            // Validation error: Fence values must
            // increase monotonically
            return;
      }
      fence.signalValue = signalValue;
      VkFence nativeFence;
      vkCreateFence(device, createInfo, nullptr,
                    &nativeFence);
      vkQueueSubmit(queue, 0, nullptr, nativeFence);
      signaledFences.push_back(
        SignaledFence{
          fence, nativeFence, signalValue});
}
```

Implementing Timeline Fences in Dawn

```
void Queue::DoThisOccasionally() {
      for (auto it = signaledFences.begin(); it != signaledFences.end();) {
            if (vkGetFenceStatus(device, it.nativeFence) == VK_SUCCESS) {
                  // The native fence is complete. Update the completedValue
                  it.fence.completedValue = it.signalValue;
                  it = signaledFences.erase(it);
            } else {
                  it++;
            }
      }
}

uint64_t Fence::GetCompletedValue() {
      return completedValue;
}
```

Every once in a while, go through the list of all fences and update the fences that have completed.

Returns a Fence's completedValue

Implementing Timeline Fences in Dawn

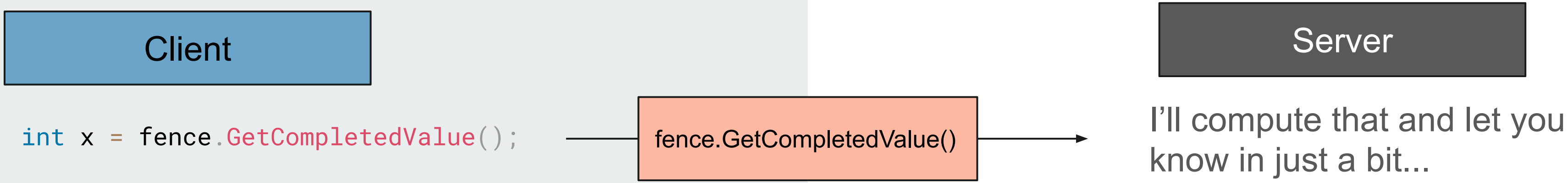
```
void Queue::DoThisOccasionally() {
      for (auto it = signaledFences.begin(); it != signaledFences.end();) {
            if (vkGetFenceStatus(device, it.nativeFence) == VK_SUCCESS) {
                  // The native fence is complete. Update the completedValue
                  it.fence.completedValue = it.signalValue;
                  it = signaledFences.erase(it);
            } else {
                  it++;
            }
      }
}
```

```
uint64_t Fence::GetCompletedValue() {
      return completedValue;
}
```

This doesn't "just work" on the Web :(

The *client* browser talks to our *server* Dawn implementation via interprocess communication using a *command buffer.*

The client does not run Dawn, it asks a service to execute commands.



This doesn't "just work" on the Web :(

The *client* browser talks to our *server* Dawn implementation via interprocess communication using a *command buffer.*

The client does not run Dawn, it asks a service to execute commands.

Client

```
int x = fence.GetCompletedValue();
```

fence.GetCompletedValue()

Server

I'll compute that and let you know in just a bit...

?!? This is supposed to be synchronous. What do I assign to x!?

Timeline Fences: Client-Side State Tracking

Client

	signaledValue	completedValue
fence	0	0

```
queue.Signal(fence, 2u);
```

Server

Timeline Fences: Client-Side State Tracking

Client

	signaledValue	completedValue
fence	2	0

```
queue.Signal(fence, 2u);
            clientQueueSignalStub(...);
```

Server

```
serverQueueSignalStub(...);
   queue.Signal(fence, 2u);
   fence.onCompletion(2u, ForwardFenceValue);
```

Timeline Fences: Client-Side State Tracking

Client

	signaledValue	completedValue
fence	2	0

```
queue.Signal(fence, 2u);
          clientQueueSignalStub(...);
```

```
int x = fence.GetCompletedValue();  // x <-- 0
```

Server

```
serverQueueSignalStub(...);
   queue.Signal(fence, 2u);
   fence.onCompletion(2u, ForwardFenceValue);
```

Timeline Fences: Client-Side State Tracking

Client

	signaledValue	completedValue
fence	2	2

```
queue.Signal(fence, 2u);
         clientQueueSignalStub(...);
```

```
int x = fence.GetCompletedValue();  // x <-- 0
```

```
        handleFenceValueUpdate(...);
```

Server

```
serverQueueSignalStub(...);
  queue.Signal(fence, 2u);
  fence.onCompletion(2u, ForwardFenceValue);
```

```
// Some time later...
ForwardFenceValue(fence, 2u);
```

Timeline Fences: Client-Side State Tracking

Client

	signaledValue	completedValue
fence	2	2

```
queue.Signal(fence, 2u);
          clientQueueSignalStub(...);
```

```
int x = fence.GetCompletedValue();  // x <-- 0
```

```
handleFenceValueUpdate(...);
```

```
// Some time later...
int y = fence.GetCompletedValue();  // y <-- 2
```

Server

```
serverQueueSignalStub(...);
   queue.Signal(fence, 2u);
   fence.onCompletion(2u, ForwardFenceValue);
```

```
// Some time later...
ForwardFenceValue(fence, 2u);
```

This Client / Server separation exists for *every* object in Dawn.

It's actually pretty simple, but this concept was foreign to me when I was first introduced

What is actually happening here?

```
dawn::Buffer buffer =
    device.CreateBuffer(&descriptor);

buffer.SetSubData(0, 10, data);
```

- The Client doesn't have any real buffers
- The Client asks the Server to execute commands
- How does this code actually call `buffer.SetSubData(0, 10, data);`?

Objects in Dawn (simplified)

```
dawn::Buffer buffer =
    device.CreateBuffer(&descriptor);
```

- Get a free `ObjectID`* for the bind group
- Allocate a “Buffer” Object
    - This is pretty much just
      ```
      struct ClientBuffer {
          uint32_t id;
      };
      ```
- Tell the server to create a real bind group and map it to `ObjectID`
- Return the ClientBuffer

*This is actually two ids for reasons I won’t explain

Objects in Dawn (simplified)

```
dawn::Buffer buffer =
    device.CreateBuffer(&descriptor);
```

- Get a free `ObjectID`* for the bind group
- Allocate a "Buffer" Object
  - This is pretty much just
    ```
    struct ClientBuffer {
        uint32_t id;
    };
    ```
- Tell the server to create a real bind group and map it to `ObjectID`
- Return the `ClientBuffer`

→

- Actually create a real Buffer
- Map the `ObjectID` to the created buffer

*This is actually two ids for reasons I won't explain

Objects in Dawn (simplified)

```
buffer.SetSubData(0, 10, data);
```

- ```
  BufferSetSubDataCmd cmd {
          buffer.id,
          0, 10, data
  };
  ```
  ```

Objects in Dawn (simplified)

```
buffer.SetSubData(0, 10, data);
```

- ```
  BufferSetSubDataCmd cmd {
        buffer.id,
        0, 10, data
  };
  ```

- Lookup the `ObjectID` and get a pointer to a Buffer
- Execute `buffer.SetSubData(0, 10, data);`
  ```

Summary

Communicating between the Client and Server can be slow

- Transfer as little information as possible
  - Don’t send large objects between the Client and Server
  - Use ObjectIds which give the Client a “handle” to Server objects
- Reduce Client-Server dependencies so the Client is not blocked
  - Objects can be created and their ObjectIds used in other commands without needing to wait for the server

Demo :)

Career Advice?

To prepare for the future,
Don’t optimize for the future.

Tomorrow is inherently uncertain.

Don’t pour too much energy into perfecting a future that may never occur.

More specifically:

- Don’t make decisions out of fear of future regret.
- Appreciate and enjoy the opportunities before you now.