

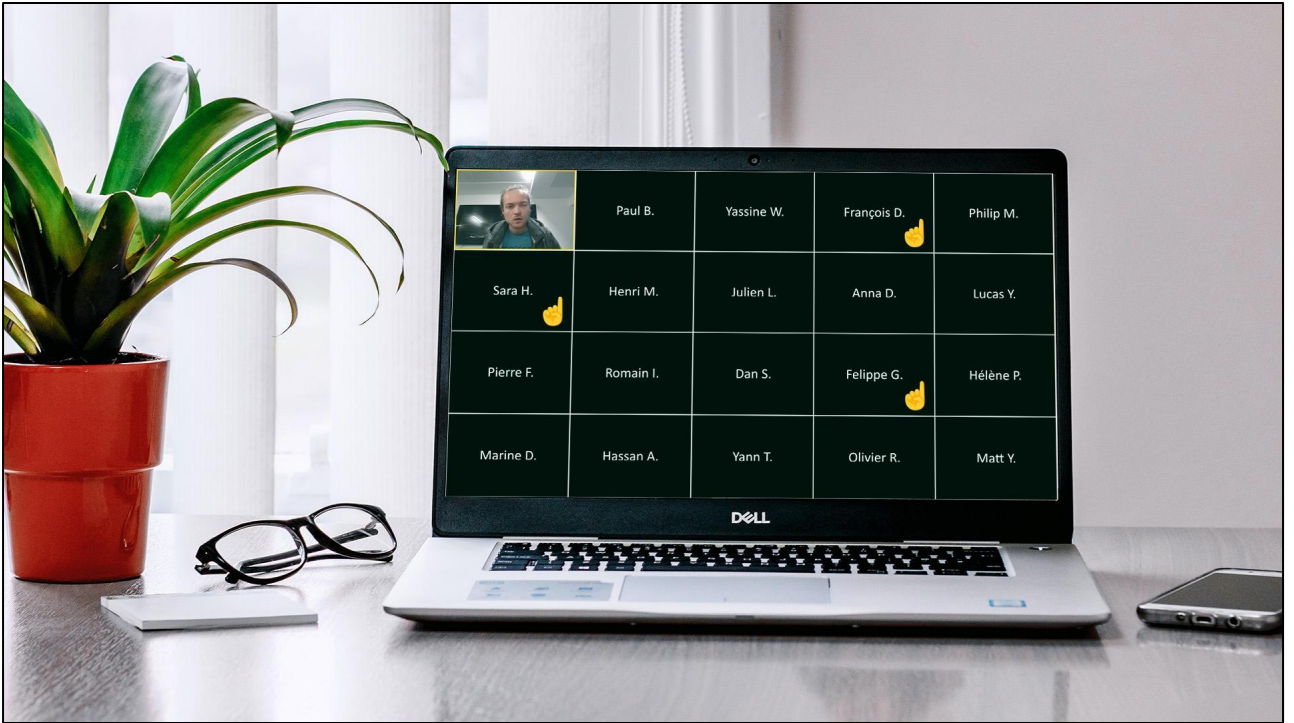
QCon London, March 2023



WebGPU is Not Just About the Web

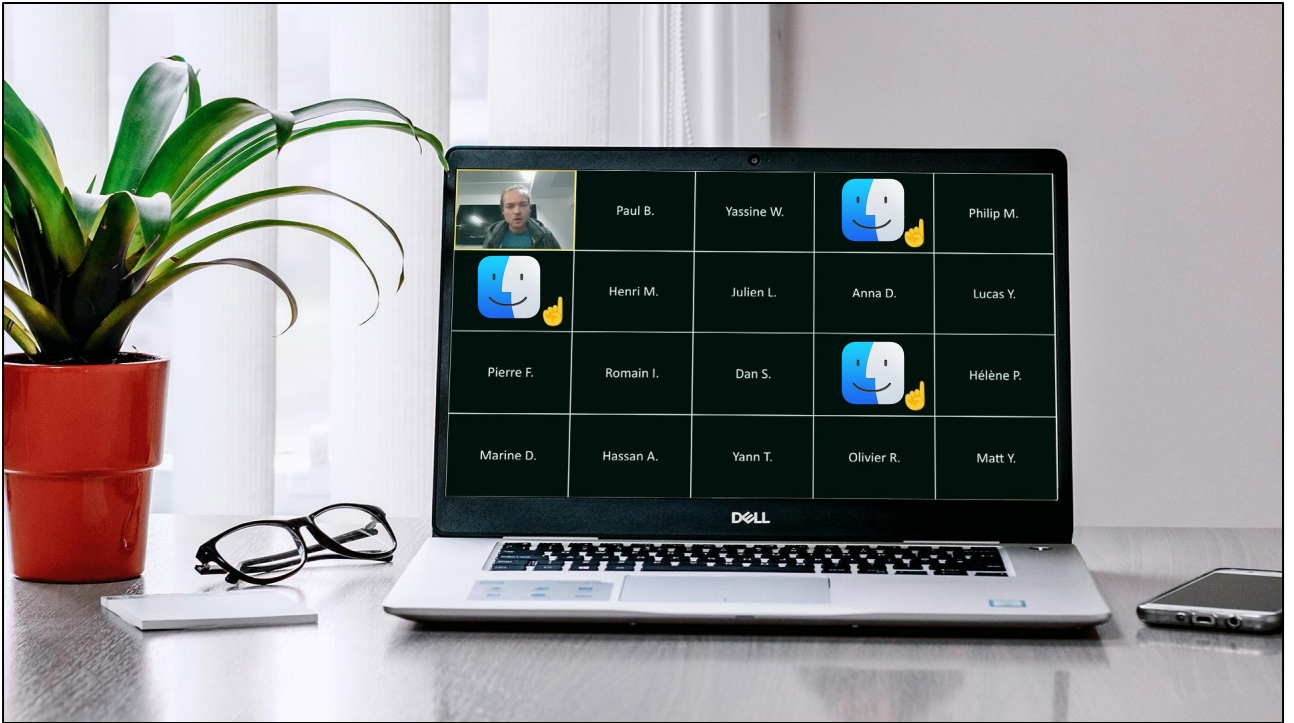
Élie Michel
Adobe Research

emichel@adobe.com



This was a classroom in 2020: a bunch of white names on a black background, and a teacher, you, speaking in the void. Your students are doing a (not so) practical session for a Computer Graphics lecture.

You see these raised hands? These are **macOS** users.



All macOS users have the very same problem: the initial code base does not build... This is when you realize that having a **portable code base** for your lecture is important, and that it is tricky when the lecture is about **real-time graphics!**



And this is you **live-backporting** your codebase to an older version of OpenGL in front of your (remote) students because Apple stopped at 4.1 and is now even **deprecating OpenGL** altogether. Have fun!

Key Points

- **State of native GPU programming**
It's complicated to find a good portable graphics API, WebGPU is a good candidate.
- **WebGPU for native development**
How to get started nowadays.
- **Current state of WebGPU**
Limitations and remaining design decisions.

This brings us to today's topic: we'll first see why it is so **uneasy to have a portable code base** for real-time graphics, and more generally GPU programming, and how **WebGPU can help** here.

Then gets our hands **more concretely** on WebGPU in the context of native desktop applications (as opposed to web), as it's also been designed for this use case!

And finally a note about whether it is **the right time** to switch or not, since WebGPU is still a Work in Progress.

About myself



What is the most relevant to this talk:

- I write a lot of **research prototypes** with real-time graphics.
- I **teach** Computer Graphics.

<https://github.com/eliemichel>

<http://eliemichel.fr>



<https://eliemichel.github.io/LearnWebGPU>



I am researcher at **Adobe Research**, and this lead me to do a lot of prototypes where I need both to **iterate quickly** and to access low-level-enough things. Each time I restart I can wonder whether I'm using the right tools so every once in a while I update my toolbox.

Teaching CG makes me wonder even more what should one learn to get started in graphics now or in a few years from now. This is how I started writing my **Learn WebGPU guide** for native C++, thinking about students in ca. 2 years from now.

I also do a lot of other things, check out <http://eliemichel.fr> and my GitHub profile!

Key Points

- State of native GPU programming
It is considered to be a good portable graphics API. WebGL2 is a good candidate.
- WebGL2 for native development
How to get started? How to use it?
- Current state of WebGL2
Limitations and remaining design decisions.

Graphics APIs

Graphics APIs

- Why do we need one?
- What are the options?
- What is the problem?

Graphics API

Assumptions

- We need to use the GPU (for **massively parallel computing**, SIMD).
- We want our code to be **portable**.

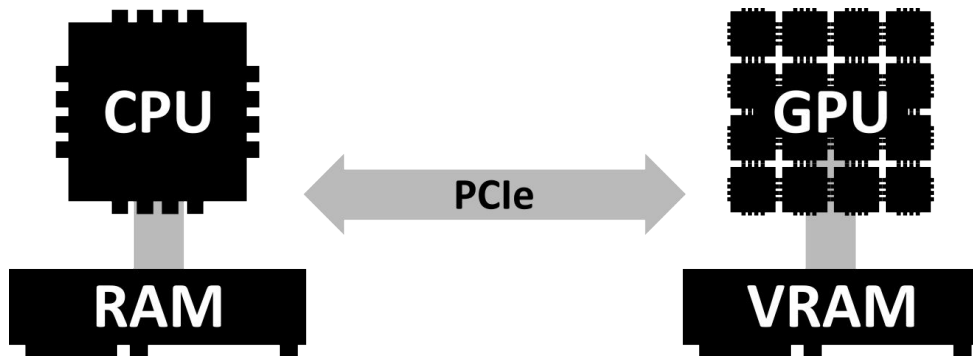
Before looking into graphics API, let's assume that we want to do GPU programming, otherwise there's no point obviously. But you are likely to need GPU computing, because it's about **much more than 3D graphics**, it's about massively parallel (Single Instruction Multiple Data-parallel) computation, which covers a lot of topics like various simulation (mechanics, chemistry, etc.) or neural net evaluation, more generally large matrix products, etc. You can get amazing perf improvement with GPU programming done right if your problem is indeed parallelizable. Plus using the GPU frees the CPU so that it can focus on other tasks.

Second assumption: you want a minimum of **portability**. If you'd only target a single OS, I'd recommend to look into this OS' idiomatic API for graphics.

Graphics API

Why do we need one?

The GPU is a **peripheral**, i.e. it is a **remote processor**.



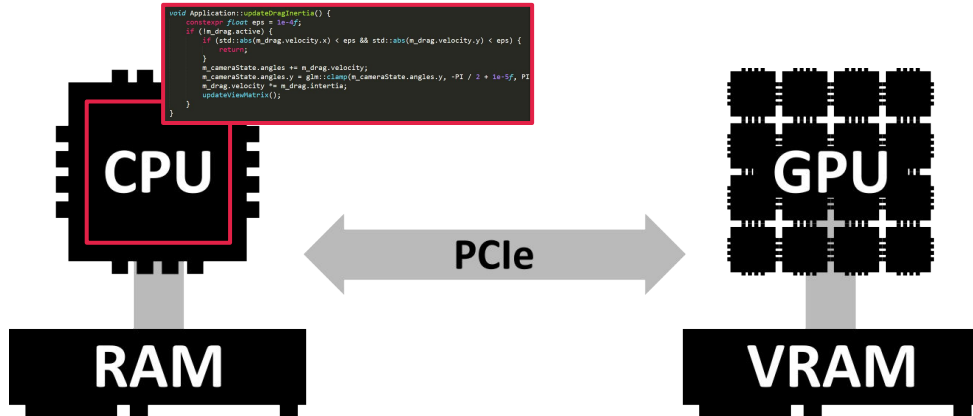
First important thing: the GPU is **far away** from the main processor. In the case of high-performance computing, this matters a lot. If you're used to web front dev, think of the GPU as a server, and the CPU would be the client.

Each processor has its own memory; RAM for the CPU, VRAM for the GPU, and everything communicates through a PCIe/wire connection.

Graphics API

Why do we need one?

The GPU is a **peripheral**, i.e. it is a **remote processor**.

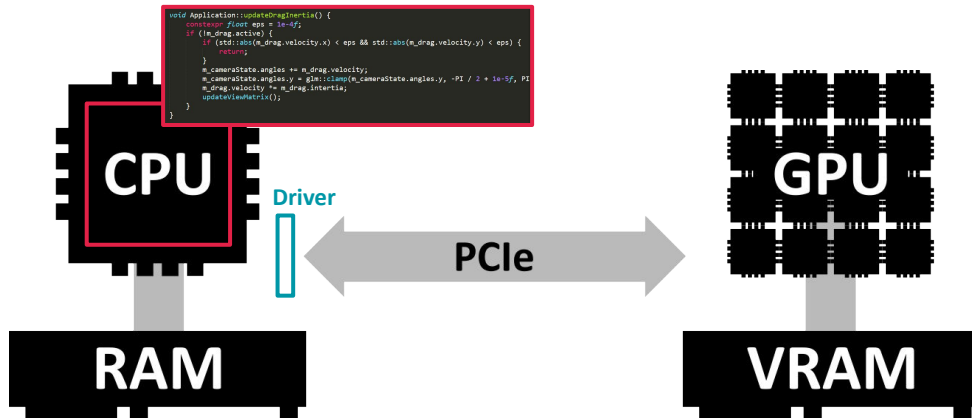


As programmers, we write code for the CPU and only **indirectly** interact with the GPU.

Graphics API

Why do we need one?

The GPU is a **peripheral**, i.e. it is a **remote processor**.



More exactly, we talk only with the **driver** of the GPU, and this is where the Graphics API is used: it's the language that the driver, or any layer on top of it, speaks.

NB: Sometimes it's a bit more than an API, it can also come with an SDK of utilities provided around the core API.

Graphics API

What are the options?

Vendor specific APIs.

OS specific APIs.

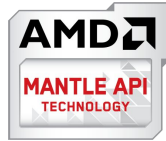
Portable APIs.

There are many different drivers, hardware vendors, operating systems, etc. So we end up with a lot of different APIs.

Graphics API

What are the options?

Vendor specific APIs.



OS specific APIs.

Portable APIs.

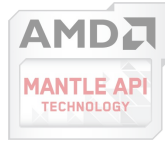
Hardware manufacturers have their own API, even their own undocumented low level API I assume. And sometimes they provide public APIs specific to their devices, the most common one being **CUDA**, used for General Purpose GPU programming (GPGPU).

I mention Mantle here to show another example, although it's no longer a thing, it was successfully proposed as a base for Vulkan so now AMD moved to Vulkan.

Graphics API

What are the options?

Vendor specific APIs.



OS specific APIs.



Portable APIs.

Then comes OS-specific APIs. DirectX/Direct3D is a whole family of APIs actually, each version being quite different from the previous ones (we're at D3D12 now), it has been used a lot for video games in the past.

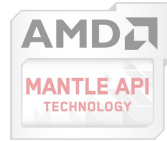
Metal is much more recent but has really become the main API for all Apple devices.

And I am citing the PlayStation API just as an example of other devices that have their own ecosystem, but we are focusing on desktop here.

Graphics API

What are the options?

Vendor specific APIs.



OS specific APIs.



Portable APIs.



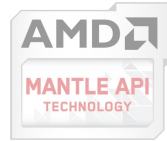
Finally the portable APIs, developed by consortiums that usually take longer to make decisions but unify the use of graphics APIs.

The main one has been OpenGL for a long time, and it comes with a flavor for embedded and mobile systems (OpenGL ES), which is ported to the Web (WenGL).

Graphics API

What are the options?

Vendor specific APIs.



OS specific APIs.



Portable APIs.



OpenGL was focusing on 3D graphics, and its design started when it was the only thing GPUs were used for. When non-graphics use started (GPGPU), a second API, OpenCL, was designed. It is a portable response to CUDA, even though CUDA might have a larger user base actually.

Graphics API

What are the options?



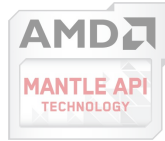
OpenGL being backward compatible, it eventually became complicated to keep on supporting it all. In 2018, Apple started deprecating it, maybe because it is making graphics drivers too heavy, maybe because they want to force their Metal API.

Anyways, the need for a next gen OpenGL had been considered already, and its name is Vulkan. Vulkan is much more low level than OpenGL, but it is also closer to the design of nowadays' GPUs, supports both desktop and embedded/mobile devices, etc... But sadly it is **not supported by Apple**. (There's MoltenVK to address this, but it's not official).

Graphics API

What are the options?

Vendor specific APIs.



OS specific APIs.



Portable APIs.



And now there is WebGPU, that we can see as a graphics API (even though it is not implemented at driver level but rather in the application's software), and WebGPU is meant to be very portable because it is a strict requirement for the Web!

Graphics API

What are the options?

A truly **portable** API?

Tough choice...

	Windows 7	Windows 10	Windows 11	macOS	iOS	Android	Linux	Raspberry PI
OpenGL 4.5+								
OpenGL ES 3.2+								
Direct3D 11								
Direct3D 12								
Direct3D 12 RT								
Metal								
Vulkan								
OpenCL								

Source: <https://www.ravbug.com/graphics/>

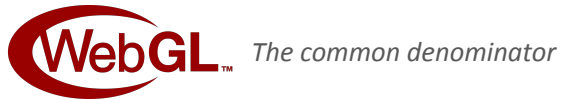
	A	B	C	D	E	F	G
1		OpenGL	OpenGL ES + ANGLE	Direct3D 11/12	Metal	Vulkan	GNM
2	Windows	Yes	Yes (GL/D3D9/D3D11/VK)	Yes	No	Yes	No
3	macOS	-Yes [1]	-Yes (GL/MTL) [1]	No	Yes	Yes (MoltenVK)	No
4	iOS	No	-Yes (ES/MTL) [1]	No	Yes	Yes (MoltenVK)	No
5	tvOS	No	-Yes (ES/MTL) [1]	No	Yes	Yes (MoltenVK)	No
6	Linux	Yes	Yes (GL/VK)	Yes (DXVK)	No	Yes	No
7	Android	No	Yes (ES/VK)	No	No	Yes	No
8	Switch	Yes	Yes	No	No	Yes	No
9	PS4	No	No	No	No	No	Yes
10	Xbox One	Unknown [2]	Yes (D3D11)	Yes	No	No	No
11	PS5	No?	No?	No	No	No?	Unknown
12	Xbox SX	Unknown [2]	Yes? (D3D11)	Yes	No	No	No
13	Web	No	Yes (WebGL)	No	No	No	No
14	Stadia	No	Yes (VK)	No	No	Yes	No
15	% Coverage	31	85	31	23	62	8

Source: <https://twitter.com/13xforever/status/1364893881368793089>

Graphics API

How does the Web do?

Portability is a **strict** requirement.



	A	B	C	D	E	F	G
1		OpenGL	OpenGL ES + ANGLE	Direct3D 11/12	Metal	Vulkan	GNM
2	Windows	Yes	Yes (GL/D3D9/D3D11/VK)	Yes	No	Yes	No
3	macOS	-Yes [1]	-Yes (GL/MTL) [1]	No	Yes	Yes (MoltenVK)	No
4	iOS	No	-Yes (ES/MTL) [1]	No	Yes	Yes (MoltenVK)	No
5	tvOS	No	-Yes (ES/MTL) [1]	No	Yes	Yes (MoltenVK)	No
6	Linux	Yes	Yes (GL/VK)	Yes (DXVK)	No	Yes	No
7	Android	No	Yes (ES/VK)	No	No	Yes	No
8	Switch	Yes	Yes	No	No	Yes	No
9	PS4	No	No	No	No	No	Yes
10	Xbox One	Unknown [2]	Yes (D3D11)	Yes	No	No	No
11	PS5	No?	No?	No	No	No?	Unknown
12	Xbox SX	Unknown [2]	Yes? (D3D11)	Yes	No	No	No
13	Web	No	Yes (WebGL)	No	No	No	No
14	Stadia	No	Yes (VK)	No	No	Yes	No
15	% Coverage	3	66	31	23	62	8

The web cannot negotiate with portability. The solution adopted with WebGL was to support the common denominator of all devices (OpenGL ES), but this is thus **limited by the lowest end device.**

Graphics API

How does the Web do?



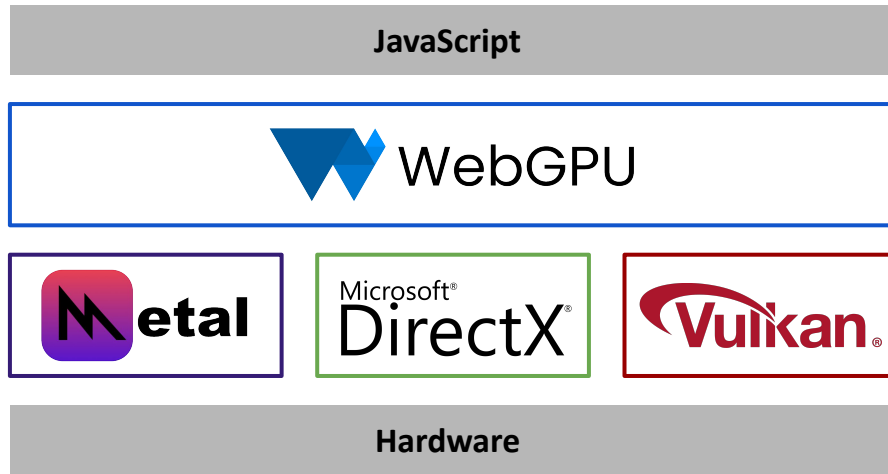
	A	B	C	D	E	F	G
	OpenGL	OpenGL ES + ANGLE	Direct3D 11/12	Metal	Vulkan	GNM	
1							
2	Windows	Yes	Yes (GL/D3D9/D3D11/VK)	Yes	No	Yes	No
3	macOS	-Yes [1]	-Yes (GL/MTL) [1]	No	Yes	Yes (MoltenVK)	No
4	iOS	No	-Yes (ES/MTL) [1]	No	Yes	Yes (MoltenVK)	No
5	tvOS	No	-Yes (ES/MTL) [1]	No	Yes	Yes (MoltenVK)	No
6	Linux	Yes	Yes (GL/VK)	Yes (DXVK)	No	Yes	No
7	Android	No	Yes (ES/VK)	No	No	Yes	No
8	Switch	Yes	Yes	No	No	Yes	No
9	PS4	No	No	No	No	No	Yes
10	Xbox One	Unknown [2]	Yes (D3D11)	Yes	No	No	No
11	PS5	No?	No?	No	No	No?	Unknown
12	Xbox SX	Unknown [2]	Yes? (D3D11)	Yes	No	No	No
13	Web	No	Yes (WebGL)	No	No	No	No
14	Stadia	No	Yes (VK)	No	No	Yes	No
15	% Coverage	3	6	31	23	62	8



How can WebGPU do differently? The designers of WebGPU were facing the very same problem as us trying to build a portable native application (the web browser, in their case).

Graphics API

How does the Web do?

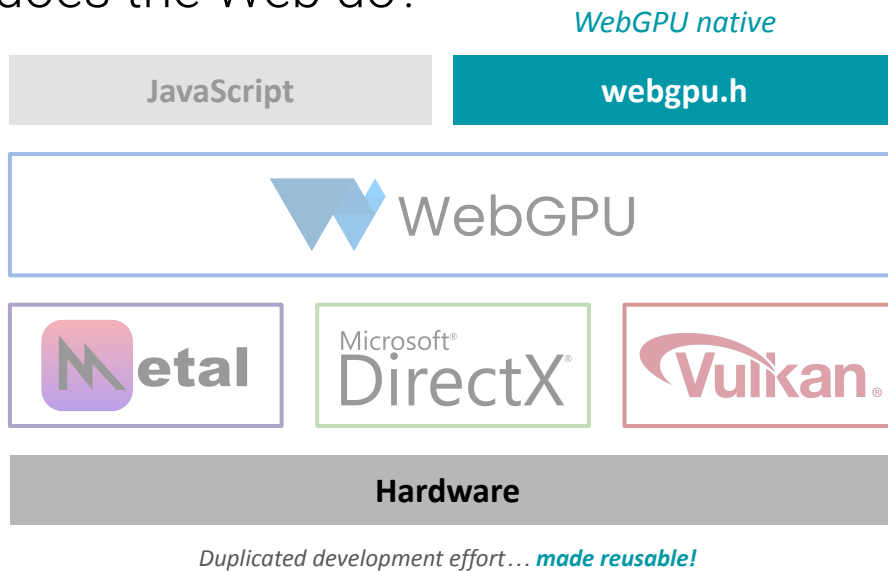


Duplicated development effort...

They could not use out-of-the-box portability, they had no choice but to **implement WebGPU multiple times**, on top of multiple graphics APIs.

Graphics API

How does the Web do?

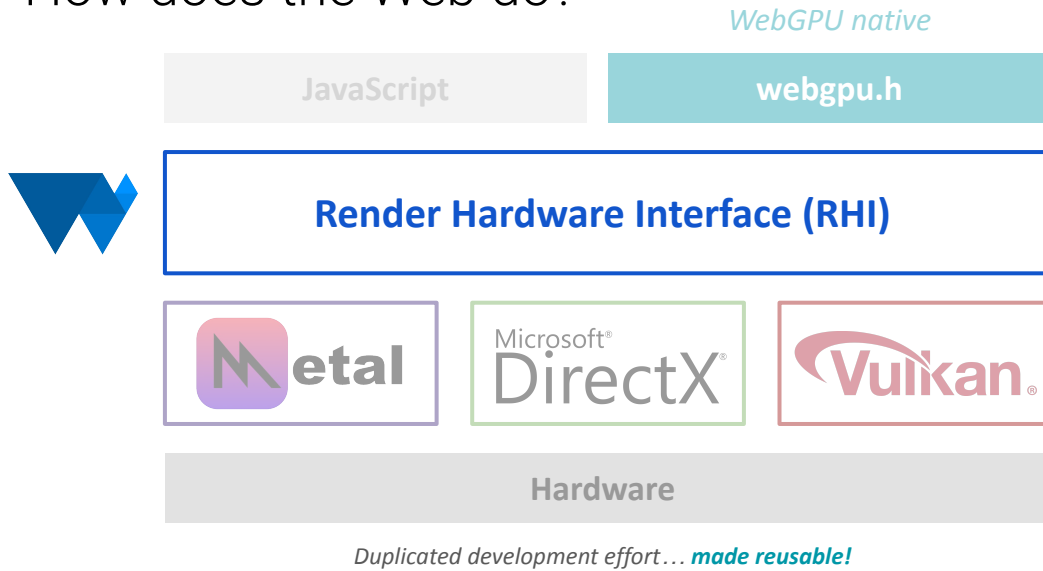


The good news is: they **shared** this annoying work! Besides exposing WebGPU to in-browser JavaScript, the browser developers also made this abstraction layer **available for native development**.

Official webgpu.h header: <https://github.com/webgpu-native/webgpu-headers>

Graphics API

How does the Web do?



Note that given the overall state of graphics APIs, this is actually a somewhat **common pattern** in projects that intend to be largely portable. It is often called a **Render Hardware Interface (RHI)**.

Graphics API

Render Hardware Interface (RHI)



Render Hardware Interface (RHI)

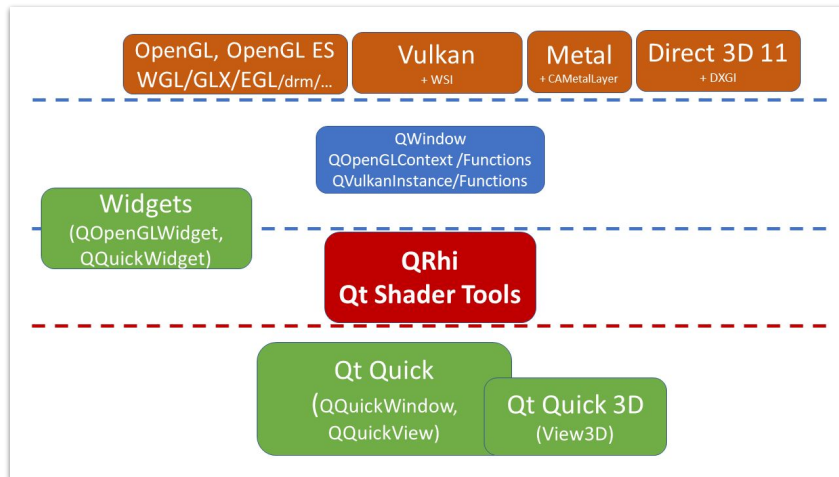
Unreal Engine RHI: <https://docs.unrealengine.com/5.0/en-US/API/Runtime/RHI>

Unreal Engine has its game-specific RHI.



Graphics API

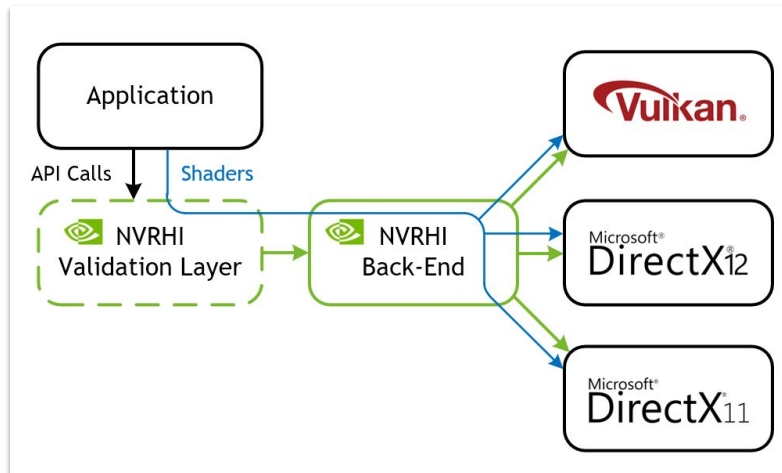
Render Hardware Interface (RHI)



Qt RHI: <https://doc.qt.io/qt-6/topics-graphics.html>

Qt contains a QRhi abstraction layer that does a similar job.

Render Hardware Interface (RHI)

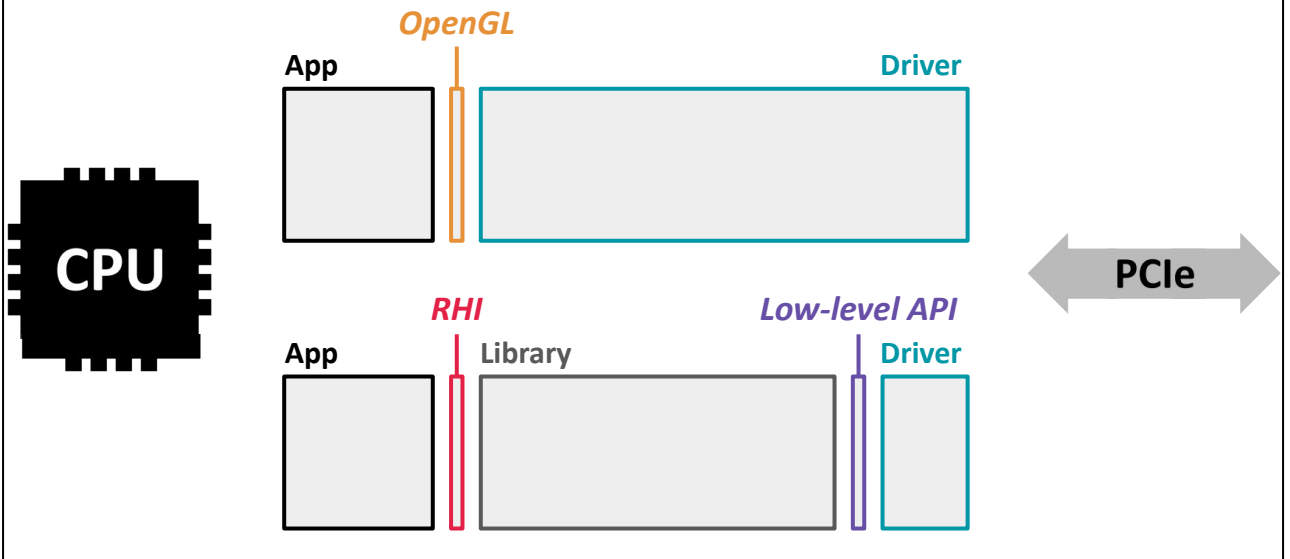


NVRHI: <https://github.com/NVIDIAGameWorks/nvrhi>

This is another example of domain-agnostic RHI proposed by NVidia (although I'm not sure it is intensively used)

Graphics API

Render Hardware Interface (RHI)



There is a shift from having the user-facing graphics API maintained by driver developers to having an extra layer enabling **third party intermediate libraries**. This makes the life of hardware manufacturers and OSes easier and more focused, and enables RHIs that are potentially domain specific, or that are challenging each others to provide nice and efficient features.

Graphics API Recap

- **Portability + Performance**
We need a Render Hardware Interface (RHI).
- **Standard and domain agnostic**
These are requirements of WebGPU.
- **Future-proof**
WebGPU will be maintained for a long time.



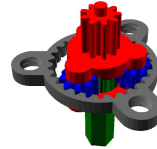
To recap:

1. In order to be **both portable and targeting high performance applications**, WebGPU needs to be a Render Hardware Interface (**RHI**) implemented on multiple low level APIs.
2. Contrary to other RHI, it is domain-agnostic, because the Web is used for potentially anything!
3. I believe that WebGPU is **future-proof**, because it will become *the* graphics API for the web, so it will **need to be maintained** and evolve, and there will be an **important user base** that can develop techniques and documentation that are also useful to native applications.

Graphics API

Bonus

- Reasonable level of abstraction
More modern than OpenGL, easier to use than Vulkan.
- Concurrent implementations
Chrome and Firefox are challenging for the best performances.



As a bonus, I find it a **nice trade-off** between OpenGL and Vulkan, in my experience. And having **multiple concurrent implementation** will likely drive performances up! (Think how we ended up doing something efficient with JavaScript although it started as a hacky language.)

Key Points

- State of native GPU programming
- It's complicated to get a good portable graphics API, WebGL2 is a good candidate
- WebGL2 has native development
- How to get started knowledge
- Current state of WebGL2
- Limitations and remaining design decisions

WebGPU native

How to get started

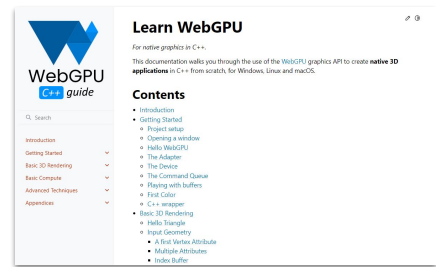
Let us now dive a little bit in the code!

Getting started with WebGPU native

Outline

- How to build a hello world?
- What is a typical application skeleton?
- How to debug?

To go further:



<https://eliemichel.github.io/LearnWebGPU>

Of course I don't have the time here to go through a full intro, so I'll first show the **very first steps**, namely building a project using WebGPU, then show a basic **application skeleton** and finish with an important note about how to **debug** GPU programming, because it requires different tools than CPU programming.

For more info I invite you all to follow my detailed programming guide!

<https://eliemichel.github.io/LearnWebGPU>

Getting started with WebGPU native

Hello World

Program Skeleton

Debugging

Getting started with WebGPU native

First lines

```
#include <webgpu/webgpu.h>
```

```
int main (int, char**) {
```

```
}
```

Of course we start by **including the header**. It is commonly located in a `webgpu/` directory in the projects I've consulted (and in particular it is mandatory in projects using emscripten)

Getting started with WebGPU native

First lines

```
#include <webgpu/webgpu.h>

int main (int, char**) {
    // 1. Create a descriptor
    WGPUInstanceDescriptor desc = {};
    desc.nextInChain = nullptr;

    // 2. Create the instance using this descriptor
    WGPUInstance instance = wgpuCreateInstance(&desc);

}
```

Then we create our first WebGPU object. Objects are always created using a `wgpuCreateSomething` function, which always takes a pointer to a `SomethingDescriptor` as argument. This is a way to pass a lot of argument without having complicated function signatures, and to easily have auxiliary function build the descriptor.

The function returns a **handle** of type `WGPUSomething`, which is **simply a pointer** so it can be costlessly copied around.

Descriptors always have a `nextInChain` field, that must be set to `nullptr` (or `NULL` in C) unless we are using the **extension mechanism** that this pointer is for.

Getting started with WebGPU native

First lines

```
#include <webgpu/webgpu.h>

int main (int, char**) {
    // 1. Create a descriptor
    WGPUInstanceDescriptor desc = {};
    desc.nextInChain = nullptr;

    // 2. Create the instance using this descriptor
    WGPUInstance instance = wgpuCreateInstance(&desc);

    // 3. Check for errors
    if (!instance) {
        std::cerr << "Could not initialize WebGPU!" << std::endl;
        return 1;
    }

    // 4. Display the object (WGPUInstance is a simple pointer).
    std::cout << "WGPU instance: " << instance << std::endl;
}
```

We can detect **errors** by checking that the returned handle is non null.

Getting started with WebGPU native

First lines

Descriptor

Handle

```
#include <webgpu/webgpu.h>

int main (int, char**) {
    // 1. Create a descriptor
    WGPUInstanceDescriptor desc = {};
    desc.nextInChain = nullptr;

    // 2. Create the instance using this descriptor
    WGPUInstance instance = wgpuCreateInstance(&desc);

    // 3. Check for errors
    if (!instance) {
        std::cerr << "Could not initialize WebGPU!" << std::endl;
        return 1;
    }

    // 4. Display the object (WGPUInstance is a simple pointer).
    std::cout << "WGPU instance: " << instance << std::endl;
}
```

This Descriptor+Handle pattern is a recurring idiom of the API.

Getting started with WebGPU native

Simple API Types

Descriptor Encapsulates **arguments** passed to object *Create* procedures.
Always contain a nextInChain pointer for extension mechanisms.

Handle **Opaque** representation of driver-side objects.
Can be passed by value, it's just a pointer.

Simple API Types

- Descriptor** Encapsulates **arguments** passed to object *Create* procedures.
Always contain a nextInChain pointer for extension mechanisms.
- Handle** **Opaque** representation of driver-side objects.
Can be passed by value, it's just a pointer.
- Enum** Named int32.
Values formed as WGPUEnumType_EnumValue.
- Struct** Mainly used to hierarchise info in descriptors.

There are also 2 families of types that appear in the API, enum and struct, which are both used for organizing the construction of descriptor. And this is it, **no convoluted types, and no hidden state!**

Getting started with WebGPU native Building

- Where is **webgpu.h**?
- Where are **symbols** defined?

(Not in the driver)

```
#include <webgpu/webgpu.h>

int main (int, char**) {
    // 1. Create a descriptor
    WGPUInstanceDescriptor desc = {};
    desc.nextInChain = nullptr;

    // 2. Create the instance using this descriptor
    WGPUInstance instance = wgpuCreateInstance(&desc);

    // 3. Check for errors
    if (!instance) {
        std::cerr << "Could not initialize WebGPU!" << std::endl;
        return 1;
    }

    // 4. Display the object (WGPUInstance is a simple pointer).
    std::cout << "WGPU instance: " << instance << std::endl;
}
```

Okey, this code sample is simple enough, but how do we build it exactly? Where do I get this **webgpu.h** file, and where are they effectively implemented? (i.e. how do I **link** my program to WebGPU?)

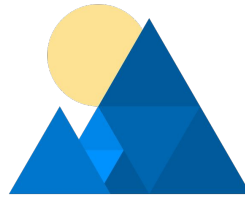
Building with WebGPU

Multiple distributions



wgpu-native (Firefox)

<https://github.com/gfx-rs/wgpu-native>



Dawn (Chrome)

<https://dawn.googlesource.com/dawn>



emscripten

emscripten

<https://emscripten.org/>

For OpenGL, we had Glad <https://glad.dav1d.de>

The answer is: there are **multiple answers**. Both Firefox and Chrome had their WebGPU backend be a standalone project that we can link against. And when cross-compiling to the web using emscripten, the compiler converts calls to `webgpu.h` symbols into calls to the JavaScript WebGPU API.

When first looking at WebGPU, I was looking for a dev experience as simple as using glad for OpenGL: just get a few header and C files and copy them into your project, or automatically generate them by including a `CMakeLists.txt`.

Building with WebGPU

Which distribution?



Rust-based *Cannot easily build from scratch in a C++ project*

Dynamic library *Auto-generated builds available*



Non-standard build system *(needs `depot_tools` and Python)*

Stripped-down pre-generated version?

Not straightforward, could get at least a CMake/Python only version

Zig build? (<https://github.com/hexops/mach-gpu-dawn>)

No MSVC version

When first looking at WebGPU, I was looking for a **dev experience as simple as using glad** for OpenGL: just get a few header and C files and copy them into your project, or automatically generate them by including a CMakeLists.txt. I want WebGPU to remain a **simple dependency**, that does not impose anything to the parent project.

But it was **not that easy** out-of-the-box. Because an OpenGL wrangler like glad is a thin layer that just queries procedures that are actually implemented by the driver, but a WebGPU backend is actually a lot more code.

First, wgpu-native is a **rust** library, so no easy way to build from scratch while doing a C++ project. Hence I turned to Dawn, but it uses a **non-standard build system**, which meant disrupting my build system although I wanted to **keep it simple**. I built a small version of Dawn, but it is still taking some time to build.

I then turned to **precompiled binaries**, first considering Zig for a dependency-less cross-platform build of Dawn, but no support for MSVC, so then turning back to wgpu-native: since we're using precompiled binaries, it is not a problem that it is written in rust, and they are regularly auto-generating builds for all platforms.

Building with WebGPU

Which distribution?

Repackaged WebGPU distributions:

- **CMake integration** (emcmake-ready)
- **Minimum dependencies**
- **Interchangeable distributions**



#define WEBGPU_BACKEND_WGPU
Curated precompiled binaries



#define WEBGPU_BACKEND_DAWN
Get source using FetchContent
No depot_tool, only Python



<https://github.com/eliemichel/WebGPU-distribution>

On the long run, this should be handled by wgpu-native and Dawn themselves imho

I ended up with **two solutions** that are (almost) transparently interchangeable, cross platform, and minimize dependencies.

The fastest one is wgpu-native prebuilds (with some fixes because at the time some builds were broken or misnamed). And for from-source build there is the Dawn option, which still requires Python to autogenerate some parts of the code but for which I could get rid of the depot_tools dependency.

Even though I would have preferred not to, I ended up repackaging what I called **distributions** of WebGPU. I believe this will be eventually provided by the upstream libraries, I'm going to open PRs (might take some time on Dawn's side though, not easy to disrupt an existing build system cause nobody wants to spend time on it usually).

Getting started with WebGPU native Building

webgpu ————— **Any distribution**
 CMakeLists.txt
 main.cpp

```
add_subdirectory(webgpu)

add_executable(App main.cpp)

target_link_libraries(App PRIVATE webgpu)

target_copy_webgpu_binaries(App)
```

```
> cmake -B build && cmake --build build
> build/App
WGPU instance: 0000025FAF74C390
```

```
#include <webgpu/webgpu.h>

int main (int, char**) {
  // 1. Create a descriptor
  WGPUInstanceDescriptor desc = {};
  desc.nextInChain = nullptr;

  // 2. Create the instance using this descriptor
  WGPUInstance instance = wgpuCreateInstance(&desc);

  // 3. Check for errors
  if (!instance) {
    std::cerr << "Could not initialize WebGPU!" << std::endl;
    return 1;
  }

  // 4. Display the object (WGPUInstance is a simple pointer).
  std::cout << "WGPU instance: " << instance << std::endl;
}
```

So, let's get back to building our hello world! It's now as simple as getting one of the distributions (the webgpu directory next to the main.cpp), adding a very simple CMakeLists.txt and building **like any other CMake project!**

Nothing crazy happens here but if you see a non null pointer after "WGPU instance:" you're good to go.

Hello World

Getting started with WebGPU native

Program Skeleton

Debugging

Getting started with WebGPU native

Program skeleton

```
void Application::run() {
```

```
}
```

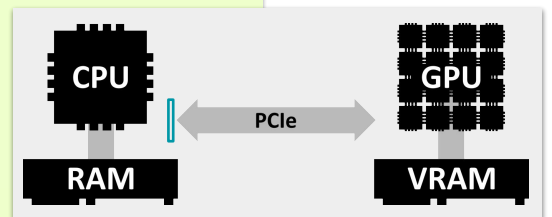
(Note that I am wrapping this skeleton into an `Application` class, to avoid using globals, and because that's usually what one does for many other reasons.)

Program skeleton

Device creation

```
void Application::run() {  
    initDevice();    // Different for native or web target  
  
}
```

Device:
CPU-side object



The first step is to initialize the **Device**, which is a logical object representing the underlying graphics device. This is an important step when it comes to figuring out a good performance/portability trade-off as this is where we consider **device capabilities** (more on this later on).

Also, this step is **slightly different for native and web dev**, this is one of the very few places where there will be a `#ifdef __EMSCRIPTEN__` in your code base.

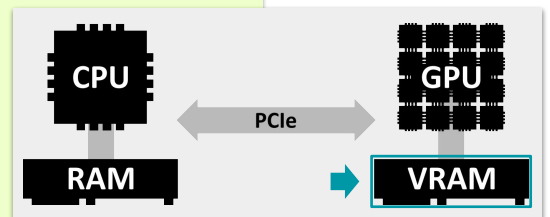
Program skeleton

Resource loading

```
void Application::run() {
    initDevice();    // Different for native or web target

    loadResources(); // Textures, Buffers, Samplers, Shaders
}
}
```

GPU-side memory
allocation
+
Data upload



Then we setup resources, namely we **allocate** memory on the VRAM, and **upload** data to this memory. Memory is laid out in a special way on the GPU, **allocators are very specific to the usage** that is made with the data. First there is the distinction between buffers and textures, because their access is wired up differently, and for each of these there are numerous usage flags that must be set up carefully to enable optimizations (like memory mapping, streaming etc.)

Textures are accessed in such a specific way that there is an entire object dedicated to setting up the way data is queried: the **samplers**.

Among resources, there are also **shaders**, which are programs meant to run on the GPU itself (contrary to our C++ code that only runs on the CPU, remember).

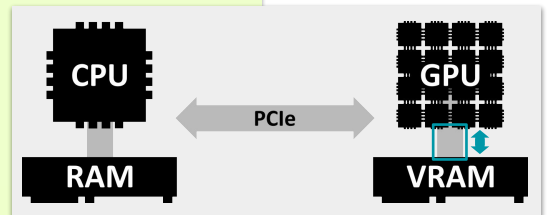
(Of course, resources can be updated, rebuilt or added dynamically during the life of the application, this is just a basic skeleton for illustrative purpose.)

Program skeleton

Resource binding

```
void Application::run() {  
    initDevice();    // Different for native or web target  
  
    loadResources(); // Textures, Buffers, Samplers, Shaders  
  
    initBindings(); // Expose resources to shaders (Texture view)  
  
}
```

Memory access
from shaders



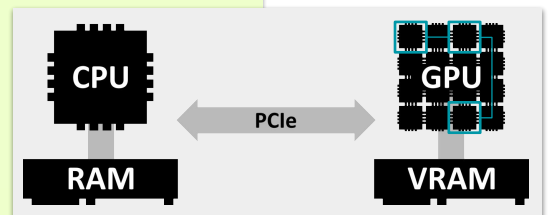
Once memory is set up, we can define what a particular shader execution will be able to see and/or edit from this memory. Again, this is used by the backend/driver to more efficiently lay out resources and organize computations at low level.

Program skeleton

Pipeline creation

```
void Application::run() {  
    initDevice();    // Different for native or web target  
  
    loadResources(); // Textures, Buffers, Samplers, Shaders  
  
    initBindings(); // Expose resources to shaders (Texture view)  
  
    initPipelines(); // Prepare configurations of the GPU's pipeline  
  
}
```

Fixed stages
+
Programmable stages



Lastly, the GPU chip has an important specificity that the CPU does not have: its execution process is a **mix of fixed function and programmable stages**. The fixed part of this pipeline is only tuned by setting up some pipeline parameters, and we must prepare one or multiple pipeline setups ahead of time, so that in the application's main loop we just quickly switch and invoke them.

Program skeleton

Pipeline creation

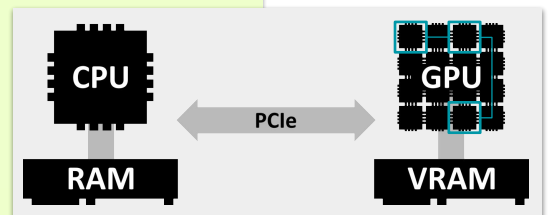
```
void Application::run() {
    initDevice();    // Different for native or web target

    loadResources(); // Textures, Buffers, Samplers, Shaders

    initBindings(); // Expose resources to shaders (Texture view)

    initPipelines(); // Prepare configurations of the GPU's pipeline
}
}
```

Fixed stages
+
Programmable stages
|
WGSL shaders



The **programmable stages** are tuned by specifying an entire program, which is what a **shader** is for. In WebGPU there are 3 types of shaders: *vertex*, *fragment* and *compute* shaders.

WebGPU comes with its own **dedicated shader language called WGSL**, which has a slightly different syntax from the usual GLSL/HLSL but accesses the same underlying primitives. On desktop, it is possible to use other shader languages, in particular SPIR-V, but this will **not** be available for the web on the long run.

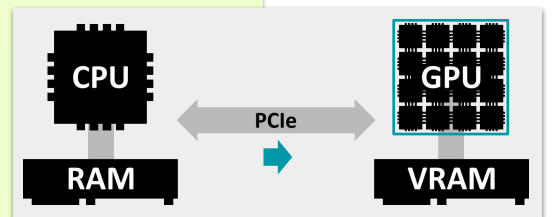
Both Firefox and Chrome provide a **shader cross-compiler** (resp. *Naga* and *Tint*) that can be used to convert code bases from GLSL/HLSL (and even SPIR-V) to WGSL.

Program skeleton

Commands

```
void Application::run() {  
    initDevice();    // Different for native or web target  
  
    loadResources(); // Textures, Buffers, Samplers, Shaders  
  
    initBindings(); // Expose resources to shaders (Texture view)  
  
    initPipelines(); // Prepare configurations of the GPU's pipeline  
  
    submitCommands(); // Main core  
  
}
```

Copy
Draw
Dispatch



We're now all set, we can use the command queue from the CPU to invoke 3D rendering and compute pipelines. From the CPU's point of view, this is a **“fire and forget” operation**: the command is emitted and the CPU program continues without waiting for the GPU to respond or even to have started considering the command. What is ensured is that commands will be processed in the way they were sent, which is enough for a lot of cases!

Program skeleton

Read back

```
void Application::run() {
    initDevice();    // Different for native or web target

    loadResources(); // Textures, Buffers, Samplers, Shaders

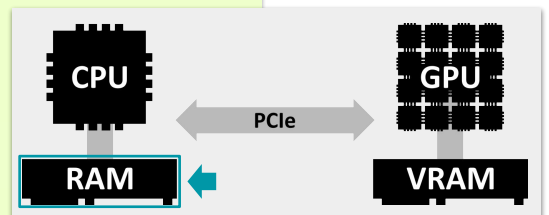
    initBindings(); // Expose resources to shaders (Texture view)

    initPipelines(); // Prepare configurations of the GPU's pipeline

    submitCommands(); // Main core

    fetchResult();  // Get data back from the GPU
}
}
```

Asynchronously get
data back from the
GPU



There are however some cases where it is needed to **get some data back** from the GPU. Always keep in mind that this takes time, wrt. the typical notion of time of this kind of application, so do it **only if really needed**. If you can process things on the GPU in a compute shader instead, it is usually better.

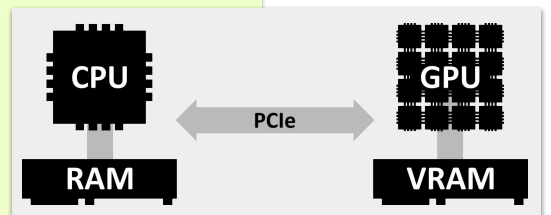
You will eventually need it though, either to get back some result that must be stored on disk, or because you need information from the GPU evaluation to allocate new buffers (the GPU cannot allocate itself). In this case, the read operation is **asynchronous**. You specify a callback, that will be called once the read operation is done. This way, your program can keep on running other things, it is not blocked by the lengthy read operation. This pattern can make coding a bit tricky sometime, or lead to simply writing a polling loop (see example later on) when you really need to block until you got the result.

Program skeleton

Termination

```
void Application::run() {  
    initDevice();    // Different for native or web target  
  
    loadResources(); // Textures, Buffers, Samplers, Shaders  
  
    initBindings(); // Expose resources to shaders (Texture view)  
  
    initPipelines(); // Prepare configurations of the GPU's pipeline  
  
    submitCommands(); // Main core  
  
    fetchResult(); // Get data back from the GPU  
  
    cleanup(); // Different for wgpu and Dawn!  
}
```

Destroy resources
+
Drop/Release objects



And of course, at the end of the program comes the clean up, freeing resources. This is where **wgpu-native** and **Dawn** still disagree on how the API should look like (more on this later on), so you will have to use the backend-specific macros until they finally settle!

Program skeleton

Swap Chain *(for graphics use only)*

```
void Application::run() {
    initDevice();    // Different for native or web target
    initSwapChain(); // For graphics
    loadResources(); // Textures, Buffers, Samplers, Shaders

    initBindings(); // Expose resources to shaders (Texture view)

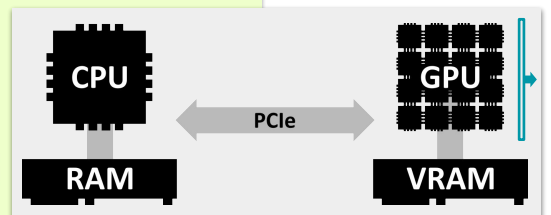
    initPipelines(); // Prepare configurations of the GPU's pipeline

    submitCommands(); // Main core

    fetchResult(); // Get data back from the GPU

    cleanup(); // Different for wgpu and Dawn!
}
```

Configure
framebuffer
presentation



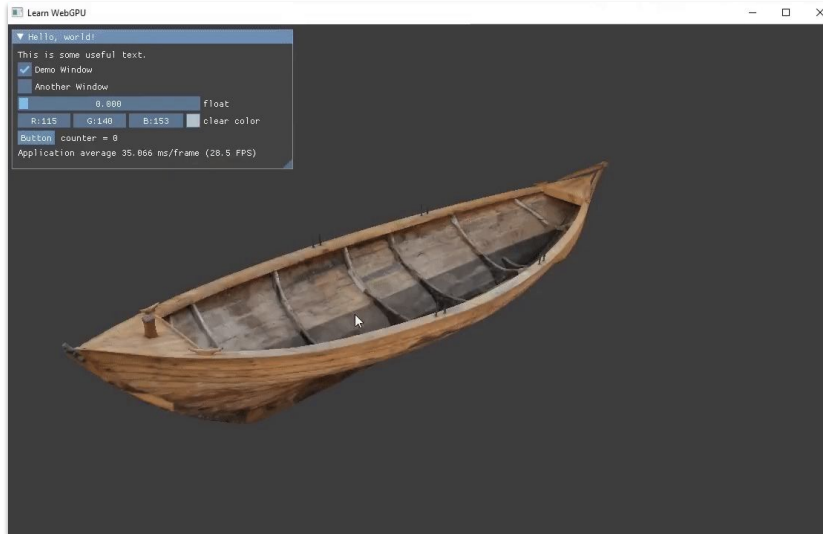
One extra step is needed when you want to display things on screen (usually the case when doing 3D graphics, but is not mandatory if you are doing GPGPU): the **swap chain** handles the *presentation* of rendered images onto the screen, via the *framebuffer* provided by the OS. This OS-part is typically handled by a cross-platform windowing library like GLFW.

Program skeleton

Demo

Extra utilities:

- `glfw3webgpu`
<https://github.com/eliemichel/glfw3webgpu>
- `imgui_impl_wgpu`
https://github.com/eliemichel/imgui/tree/eliemichel/portable_wgpu_backend



Here is an example of 3D renderer wrote using WebGPU, running on desktop. It works with either wgpu-native or Dawn.

Small **extras** I had to develop for this: A procedure to **get a WebGPU surface out of a GLFW window** (I mostly got it from wgpu-native's demo). This belongs imho in GLFW because it is highly OS specific, and GLFW is here to hide the OS-specificities away. Secondly, I use **Dear ImGui** for the UI, like a lot of people for this kind of small demos, but its wgpu **backend** was not entirely ready. It was working only with Dawn and emscripten, so you can get it from my branch (until the PR is merged).

Program skeleton


Focus on device creation

```
void Application::run() {  
    initDevice(); // Different for native or web target  
    initSwapChain(); // For graphics  
    loadResources(); // Textures, Buffers, Samplers, Shaders  
  
    initBindings(); // Expose resources to shaders (Texture view)  
  
    initPipelines(); // Prepare configurations of the GPU's pipeline  
  
    submitCommands(); // Main core  
  
    fetchResult(); // Get data back from the GPU  
  
    cleanup(); // Different for wgpu and Dawn!  
}
```

I am not going to detail all these steps, but let's focus on the device initialization, which can easily be underlooked or quickly copy-pasted once for all because it feels uninteresting.

Getting started

Device creation



```
void Application::init() {
    initDevice(); // Different for native or web target
}

void Application::initDevice() {
    // Textures, buffers, shaders, shaders
    initResources(); // Prepare resources to shaders (Texture view)
    initPipeline(); // Prepare configurations of the GPU's pipeline
    submitCommand(); // into core
    fetchResult(); // Get data back from the GPU
    cleanup(); // Different for native and web!
}
```

Getting started

Device creation *(for native targets)*

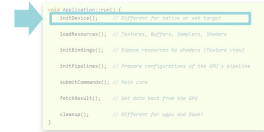
```
void Application::initDevice() {
    WGPUInstanceDescriptor desc;
    desc.nextInChain = nullptr;
    WGPUInstance instance = wgpuCreateInstance(&desc);

    WGPURequestAdapterOptions adapterOpts;
    adapterOpts.nextInChain = nullptr;
    // [...]
    WGPUAdapter adapter = wgpuInstanceRequestAdapterSync(instance, &adapterOpts);

    RequiredLimits requiredLimits = /* important! */;

    WGPUDeviceDescriptor deviceDesc;
    deviceDesc.nextInChain = nullptr;
    deviceDesc.requiredLimits = &requiredLimits;
    // [...]
    this->device = wgpuAdapterRequestDeviceSync(adapter, &deviceDesc);
}
```

It's verbose...



First thing: it's a bit verbose for C++. That's a side problem, but let's address it.

(NB: this part about the C++ style wrapper was not presented live, but I include it in the notes since the slides were ready anyways.)

Getting started

Device creation (for native targets)

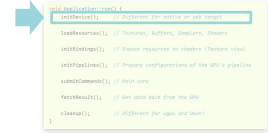
```
void Application::initDevice() {
    WGPUInstanceDescriptor desc;
    desc.nextInChain = nullptr;
    WGPUInstance instance = wgpuCreateInstance(&desc);

    WGPURequestAdapterOptions adapterOpts;
    adapterOpts.nextInChain = nullptr;
    // [...]
    WGPUAdapter adapter = wgpuInstanceRequestAdapterSync(instance, &adapterOpts);

    RequiredLimits requiredLimits = /* important! */;

    WGPUDeviceDescriptor deviceDesc;
    deviceDesc.nextInChain = nullptr;
    deviceDesc.requiredLimits = &requiredLimits;
    // [...]
    this->device = wgpuAdapterRequestDeviceSync(adapter, &deviceDesc);
}
```

It's verbose...



Almost always
nullptr

Among things cluttering the code, there is this `nextInChain` pointer for extensions that is almost always set to null.

Getting started

Device creation (for native targets)

```
void Application::initDevice() {
  WGPUInstanceDescriptor desc;
  desc.nextInChain = nullptr;
  WGPUInstance instance = wgpuCreateInstance(&desc);

  WGPURequestAdapterOptions adapterOpts;
  adapterOpts.nextInChain = nullptr;
  // [...]
  WGPUAdapter adapter = wgpuInstanceRequestAdapterSync(instance, &adapterOpts);

  RequiredLimits requiredLimits = /* important! */;

  WGPUDeviceDescriptor deviceDesc;
  deviceDesc.nextInChain = nullptr;
  deviceDesc.requiredLimits = &requiredLimits;
  // [...]
  this->device = wgpuAdapterRequestDeviceSync(adapter, &deviceDesc);
}
```

It's verbose...

```
#if APPLICATION_VERSION < 2
std::shared_ptr<WGPUInstance> Application::initDevice() {
  self.assert(); // Throws: assert()
  self.init(); // These resources to create (device desc)
  self.initAdapter(); // These configurations of the adapter
  self.initDevice(); // This code
  return nullptr; // this does back from the dev
  cleanup(); // Different for open and closed
}
```

C-style OOP

There is also the C-style naming that is needed to make method membership clear but don't benefit from C++ usual OOP notation.

Getting started

Device creation *(for native targets)*

```

void Application::initDevice() {
    WGPUInstanceDescriptor desc;
    desc.nextInChain = nullptr;
    WGPUInstance instance = wgpuCreateInstance(&desc);

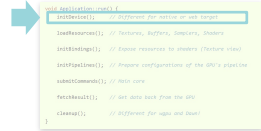
    WGPURequestAdapterOptions adapterOpts;
    adapterOpts.nextInChain = nullptr;
    // [...]
    WGPUAdapter adapter = wgpuInstanceRequestAdapterSync(instance, &adapterOpts);

    RequiredLimits requiredLimits = /* important! */;

    WGPUDeviceDescriptor deviceDesc;
    deviceDesc.nextInChain = nullptr;
    deviceDesc.requiredLimits = &requiredLimits;
    // [...]
    this->device = wgpuAdapterRequestDeviceSync(adapter, &deviceDesc);
}

```

It's verbose...



Not in webgpu.h

*It's even more verbose
in practice*

It's actually even more verbose in reality because these synchronous variants are my own construction around WebGPU's async functions.

Getting started

Device creation *(for native targets)*

```

void Application::initDevice() {
    WGPUInstanceDescriptor desc;
    desc.nextInChain = nullptr;
    WGPUInstance instance = wgpuCreateInstance(&desc);

    WGPURequestAdapterOptions adapterOpts;
    adapterOpts.nextInChain = nullptr;
    // [...]
    WGPUAdapter adapter = wgpuInstanceRequestAdapterSync(instance, &adapterOpts);

    RequiredLimits requiredLimits = /* important! */;

    WGPUDeviceDescriptor deviceDesc;
    deviceDesc.nextInChain = nullptr;
    deviceDesc.requiredLimits = &requiredLimits;
    // [...]
    this->device = wgpuAdapterRequestDeviceSync(adapter, &deviceDesc);
}

```

It's verbose...

```

// Application.cpp
Application::Application() {
    // [...]
}
// Application.h
class Application {
public:
    Application();
    ~Application();
};

```

C-style namespace

Also the C-style poor man's namespacing is quite redundant.

Getting started

Device creation (for native targets)

```
#include <webgpu/webgpu.hpp>
using namespace wgpu;
```

```
void Application::initDevice() {
    Instance instance = createInstance(InstanceDescriptor{});

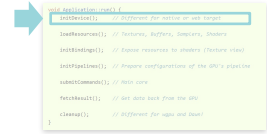
    RequestAdapterOptions adapterOpts;
    // [...]
    Adapter adapter = instance.requestAdapterSync(adapterOpts);

    RequiredLimits requiredLimits = /* important! */;

    DeviceDescriptor deviceDesc;
    deviceDesc.requiredLimits = &requiredLimits;
    // [...]
    this->device = adapter.requestDeviceSync(deviceDesc);
}
```

Using `webgpu.hpp` for lighter code
<https://github.com/eliemichel/WebGPU-Cpp>

- Compatible with **any distribution**
- **Included** in WebGPU-distribution
- No overhead



For all these reasons, and even a few more, I've developed a thin **C++ wrapper** around WebGPU. It is only cosmetic, it hides nothing and introduces **no overhead**. Converting from and to this wrapper can be done transparently. I'm not a big fan of wrappers in general but I'm happy with this solution!

The wrapper is **auto-generated** from the source, so that it can easily handle differences among backends, and tightly follow their evolution.

Note that a C++ wrapper is provided by Dawn, but afai it is not portable to other backends because it directly calls Dawn-specific things instead of only relying on the official `webgpu.h` header.

Getting started

Device creation *(for native targets)*

```
void Application::initDevice() {
    Instance instance = createInstance(InstanceDescriptor{});

    RequestAdapterOptions adapterOpts;
    // [...]
    Adapter adapter = instance.requestAdapterSync(adapterOpts);

    RequiredLimits requiredLimits = /* important! */;

    DeviceDescriptor deviceDesc;
    deviceDesc.requiredLimits = &requiredLimits;
    // [...]
    this->device = adapter.requestDeviceSync(deviceDesc);
}
```

Two-stage creation

1. Adapter

↓ Limits

2. Device

Limit = Max texture dimension, Max buffer size, Max vertex attributes, ...

Let's get back to device initialization now. It is a **2 stage process**: we first **get** the adapter, for which we can query physical limits and capabilities, and we then **create** a device that compiles with the adapter. **The adapter is given, the device is chosen** (though its creation may fail).

Device creation

Adapter and device limits

Option A: *Use default device limits*

- Potential failure anywhere in the code (*limits exceeded*)
- Different behavior on different runtime contexts

Option B: *Specify strict limits*

- Only fails at device creation
- Same limits for everybody (*once the device is created*)

➔ **Better portability**

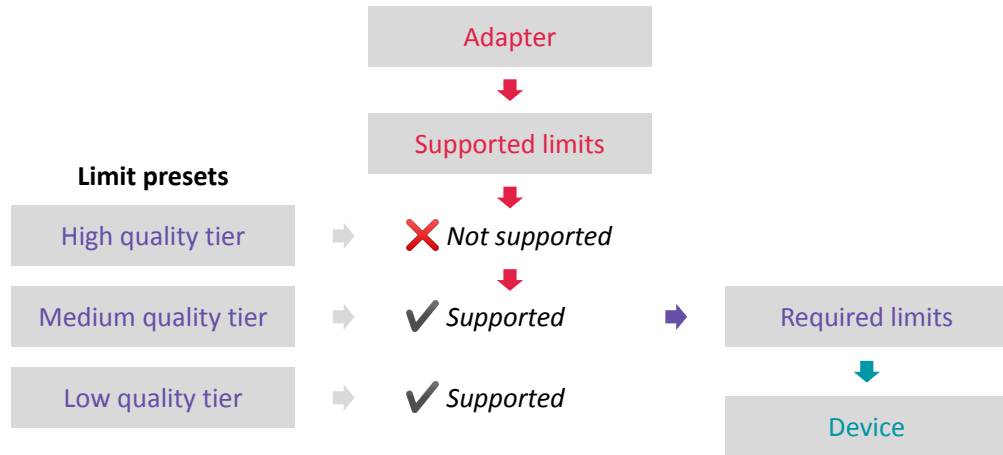
There are two way of managing device creation: the **lazy** one and the **good** one.

If you just pass through and automatically create a device that has the same limits than the adapter, you will **not statically know** the device capabilities, which means they could be infringed at any time by your program. No way to now in advance, or you need a way to carry the capabilities around everywhere you may exceed them. A lot of room for **annoying hard-to-reprduce bugs**.

The good solution is to explicitly specify upon device creation all the requierements of your program. **Start with the strictest**, and everytime you exceed the limits on your device, increase the requirements. I'll then know that **if the device creation step passes, everything else down the road should go all right!** This is what this device creation mechanism is for actually.

Device creation

Adapter and device limits



Of course if you want your application to be portable while benefitting from the capabilities of high quality devices, you will need to consider **multiple paths of execution**. But again with this mechanism everything can be **settled at initialization**. Then you just need to remember which quality tier your application is running on to decide on high level strategies, or to conditionally disable some of your features.

Device creation

For web target

```
#include <emscripten/html5_webgpu.h>

void Application::initDevice() {
    // Device setup goes in the JavaScript shell
    // that populates `Module.preinitializedWebGPUDevice`
    this->device = emscripten_webgpu_get_device();
}
```

In emscripten's shell JavaScript:

```
const adapter = await navigator.gpu.requestAdapter({ /* ... */ });
const device = await adapter.requestDevice({ requiredLimits: /* important! */ });
Module.preinitializedWebGPUDevice = device;
```

JavaScript

One last note: as I was saying things are different when **cross-compiling** for the Web. The device creation in this case is handled **on the JavaScript side**, in a few lines of code wrapping the WebAssembly module invocation. Emscripten then provides a way to get this device on the C++ side.

Hello World

Program Skeleton

Getting started with WebGPU native

Debugging

An important part of coding is debugging, and debugging GPU code has some specificities.

Debugging

Error callbacks

Right after device creation:

```
wgpuDeviceSetUncapturedErrorCallback(device, /* ... */);
```

```
wgpuDeviceSetDeviceLostCallback(device, /* ... */);
```

With `wgpu.hpp`:

```
auto handle = device.setUncapturedErrorCallback([](ErrorType type, char const* message) {  
    std::cout << "Device error: (type " << type << " )";  
    if (message) std::cout << "\n" << message;  
    std::cout << std::endl; // <-- Put a breakpoint here!  
});
```


First thing to do, **without an ounce of hesitation**, is to set up device error callback. Without this you're walking in the dark and likely getting nowhere. Whenever an error occurs in a call to a webgpu procedure, the "uncaptured error callback" is called with details coming from the backend to help one debug. It is also good to **put a breakpoint** in there so that you can inspect the stack where it failed (like for any debugging).

Debugging

Error callbacks

Dawn gives better error messages:

 **wgpu-native**
MissingTextureUsage(MissingTextureUsageError { actual: STORAGE_BINDING, expected: TEXTURE_BINDING })

 **Dawn**
[Texture "DualContouring Position"] usage (TextureUsage::StorageBinding) doesn't include TextureUsage::TextureBinding.
- **While validating** entries[2] as a Texture.
Expected entry layout: { binding: 6, visibility: ShaderStage::Compute, texture: { sampleType: TextureSampleType::Float, viewDimension: TextureViewDimension::e3D, multisampled: 0 } }
- **While validating** [BindGroupDescriptor "DualContouring Bake Fill"] against [BindGroupLayout "DualContouring Bake Fill"]
- **While calling** [Device "My Device"].CreateBindGroup([BindGroupDescriptor "DualContouring Bake Fill"]).

Note that wgpu-native and Dawn return quite different types of messages. **Dawn is more human readable**, and uses object labels to give hints, which is nice. Messages from wgpu-native are automatically formatted by rust, they usually contain the information needed but can be a bit more cryptic and laconic.

Debugging

Error callbacks *(shader compiler)*



Naga (wgpu-native)

```
Validation(ShaderError { source: "\n\t@location(0) positi[...]", label: None, inner:
WithSpan { inner: EntryPoint { stage: Fragment, name: "fs_main", source: Function(Expression { handle: [17],
source: Compose(ComponentCount { given: 3, expected: 2 }) }) }, spans: [(Span { start: 1050, end: 1074 },
"naga::Expression [17]") ] } })
```



Tint (Dawn)

```
Tint WGLSL reader failure: :39:20 error: no matching initializer for vec2<f32>(abstract-float,
abstract-float, abstract-float)
```

4 candidate initializers:

```
vec2(x: T, y: T) -> vec2<T> where: T is abstract-int, abstract-float, f32, f16, i32, u32 or bool
vec2(T) -> vec2<T> where: T is abstract-int, abstract-float, f32, f16, i32, u32 or bool
[...]
```

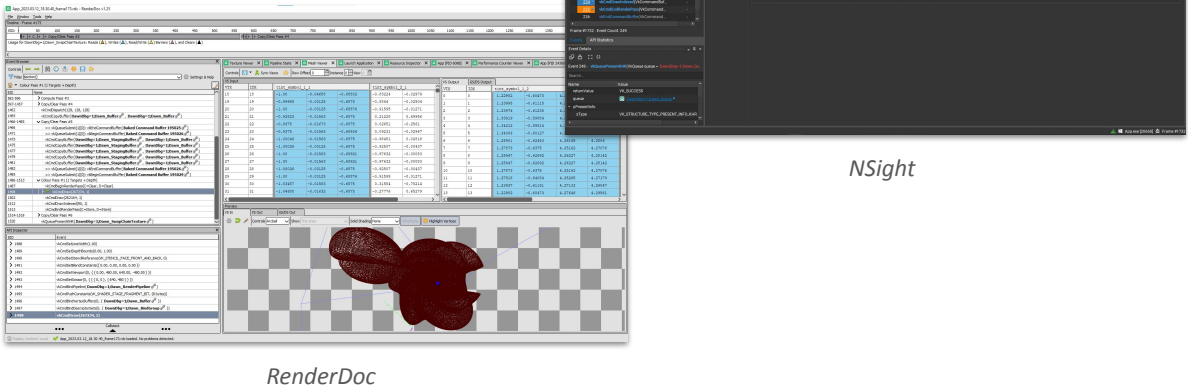
```
let lightColor1 = vec2<f32>(1.0, 0.9, 0.6);
                ^^^^
```

- While validating [ShaderModuleDescriptor]
- While calling [Device "My Device"].CreateShaderModule([ShaderModuleDescriptor]).

Here is another example, coming this time from the shader cross-compiler. Again, Dawn is clearer.

Debugging Dev Tools

Graphics debuggers



RenderDoc

NSight

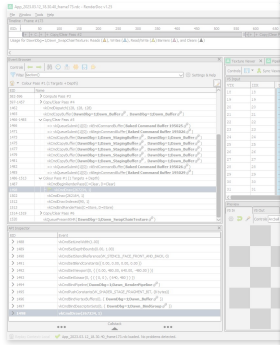
Now callback and breakpoints are nice to debug things that happen on the CPU, but when issues are on the GPU side, we're still in the dark. Especially because communication between CPU and GPU take time, it is not possible to have such a thing as a breakpoint there, neither can we introspect a stack. Instead, we **record** everything that the CPU sends to the GPU so that we can **replay** it step by step.

Graphics debugger are dedicated to this task, the most common ones being:

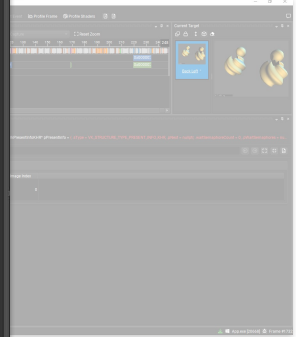
- RenderDoc: <https://renderdoc.org>
- NVidia NSight Graphics: <https://developer.nvidia.com/nsight-graphics>

Debugging Dev Tools

Graphics debuggers



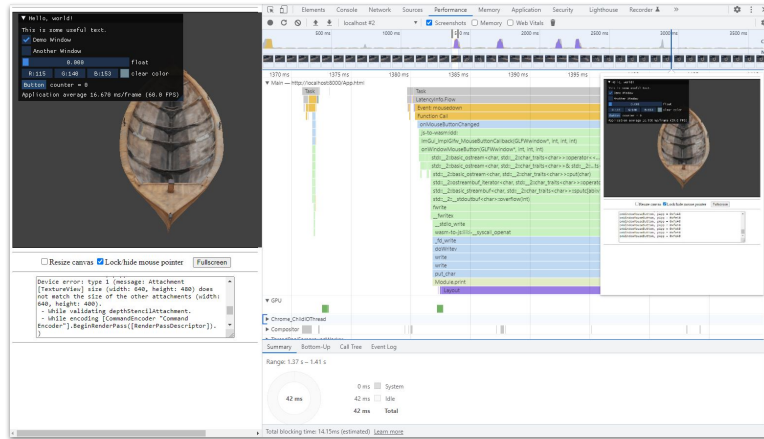
Event	Description	CPU ms	GPU ms
26	vkResetFences(VkDevice device = DawnDbg=1,Dawn_Dev...	<0.01	-
27	// Coherent mapped memory update	-	-
28	vkQueueSubmit(VkQueue queue = DawnDbg=1,Dawn_Qu...	0.11	-
29	// Primary command buffer 12 commands	-	-
30	vkBeginCommandBuffer(VkCommandBuffer commandBuffe...	-	-
31	vkCmdPipelineBarrier(VkCommandBuffer commandBuffe...	-	-
32	vkCmdCopyBuffer(VkCommandBuffer commandBuffer = ...	-	-
33	vkCmdBindPipeline(VkCommandBuffer commandBuffer ...	-	-
34	vkCmdPipelineBarrier(VkCommandBuffer commandBuffe...	-	-
35	vkCmdBindDescriptorSets(VkCommandBuffer command_...	-	-
36	vkCmdDispatch(VkCommandBuffer commandBuffer = 0x...	-	1.63
37	vkCmdBindPipeline(VkCommandBuffer commandBuffer ...	-	-
38	vkCmdPipelineBarrier(VkCommandBuffer commandBuffe...	-	-
39	vkCmdBindDescriptorSets(VkCommandBuffer command_...	-	-
40	vkCmdDispatch(VkCommandBuffer commandBuffer = 0x...	-	<0.01
41	vkCmdBindPipeline(VkCommandBuffer commandBuffer ...	-	-
42	vkCmdPipelineBarrier(VkCommandBuffer commandBuffe...	-	-
43	vkCmdBindDescriptorSets(VkCommandBuffer command_...	-	-
44	vkCmdDispatch(VkCommandBuffer commandBuffer = 0x...	-	2.31
45	vkCmdPipelineBarrier(VkCommandBuffer commandBuffe...	-	-
46	vkCmdPipelineBarrier(VkCommandBuffer commandBuffe...	-	-
47	vkCmdCopyBuffer(VkCommandBuffer commandBuffer = ...	-	-



Vulkan commands

A small problem when doing using WebGPU (or any RHI) is that the debugger only captures calls to the **low-level API**, so we need to manually figure out the mapping between these low-level API calls and our initial WebGPU commands. Maybe when WebGPU becomes common enough debuggers will start providing this but likely not in the near future.

Debugging Dev Tools



Chrome Dev Tools

As an alternative, when cross-compiling to the Web we can benefit from **the web browser's debug tools**, which **know WebGPU natively**. They are not as mature as desktop graphics debuggers that have been around for a while but this can also be helpful.

Debugging Documentation

The screenshot shows the table of contents for the WebGPU JavaScript specification. It includes sections for Introduction, Motivation, Security Considerations, and various technical details like the Shader Module and Compute Shaders.

WebGPU JavaScript spec
<https://www.w3.org/TR/webgpu>

The screenshot shows the table of contents for the WebGPU Shading Language (WGSL) specification. It includes sections for Introduction, Grammar, and various types of operations like Scalars, Vectors, and Matrices.

WGSL spec
<https://gpuweb.github.io/gpuweb/wgsl/>

```

30 #ifndef WEBGPU_H_
31 #define WEBGPU_H_
32
33 #if defined(WGPU_SHARED_LIBRARY)
34 #   if defined(_WIN32)
35 #       if defined(WGPU_IMPLEMENTATION)

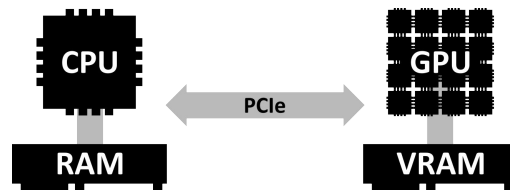
```

And read the **webgpu.h** header...

<https://github.com/webgpu-native/webgpu-headers/blob/main/webgpu.h>

Of course chasing bugs also comes with inspecting the documentation, so here are some links to the WebGPU spec. **The spec is for JavaScript**, but in a lot of cases concepts maps naturally to the native API. I also often end up reading `webgpu.h` directly to get the fine details.

Debugging Profiling



Warning: **different timelines!**

Together with debugging usually comes profiling/benchmarking. Again things are a bit unusual compared to CPU programming, because of the asynchronicity: **the CPU and GPU live on different timelines**, and the CPU never knows when the GPU starts and ends processing commands.

Debugging Profiling

In theory: **Timestamp queries**

```
QuerySetDescriptor querySetDesc;  
querySetDesc.count = 1;  
querySetDesc.type = QueryType::Timestamp;  
QuerySet querySet = device.createQuerySet(querySetDesc);  
  
ComputePassTimestampWrite timestampWrites;  
timestampWrites.location = ComputePassTimestampLocation::Beginning;  
timestampWrites.queryIndex = 0;  
timestampWrites.querySet = querySet;  
  
// [...]
```

In theory, a mechanism is provided for this: **timestamp queries**. These are timers injected in the command queue, thus running on the GPU and measuring true GPU-side timings.

Debugging Profiling

In theory: **Timestamp queries**

```
QuerySetDescriptor querySetDesc;
querySetDesc.count = 1;
querySetDesc.type = QueryType::Timestamp;
QuerySet querySet = device.createQuerySet(querySetDesc);
```

In practice:

```
ComputePassTimestamp
timestampWriter.local
timestampWriter.query
timestampWriter.query
// [...]
```

Device error: (type 1)

Timestamp queries are disallowed because they may expose precise timing information.

- While validating [QuerySetDescriptor]
- While calling [Device "My Device"].CreateQuerySet([QuerySetDescriptor]).



Device error: (type 4)

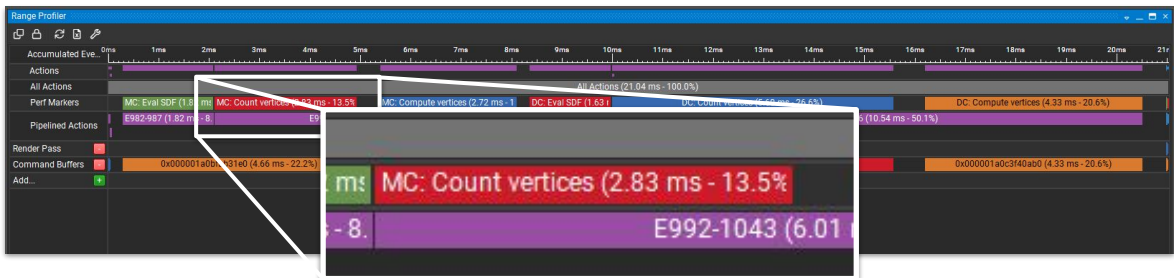
MissingFeatures(MissingFeatures(TIMESTAMP_QUERY))

But for **security** reasons, they are not available (or just not implemented in the case of wgpu?). Since this is a web only constraint, it may be possible to get them to work when targeting only desktop applications, at least at dev time, but this means the application's behavior cannot rely on these to make choices about execution strategy.

Debugging Profiling

Using Debug Markers

```
computePass.pushDebugGroup("MC: Count vertices"); // <-- Start
computePass.setPipeline(pipeline);
computePass.setBindGroup(0, bindGroup, 0, nullptr);
computePass.dispatchWorkgroups(resolution - 1, resolution - 1, resolution - 1);
computePass.popDebugGroup(); // <-- End
```



Another nice profiling utility that does work (Dawn only at the moment) are **debug groups**. This can be used to visualize custom profiling sequences directly in Nsight!

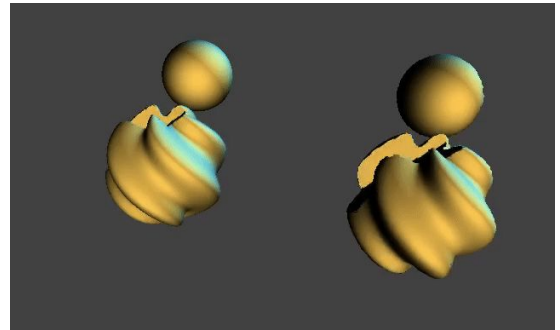
Debugging

Example: Distance Field Contouring

Setup:

- 2M voxels
- 1 signed distance and normal per voxel
- Dynamic scene

Goal: Generate a **contour mesh on the fly**



Marching Cubes

Dual Contouring

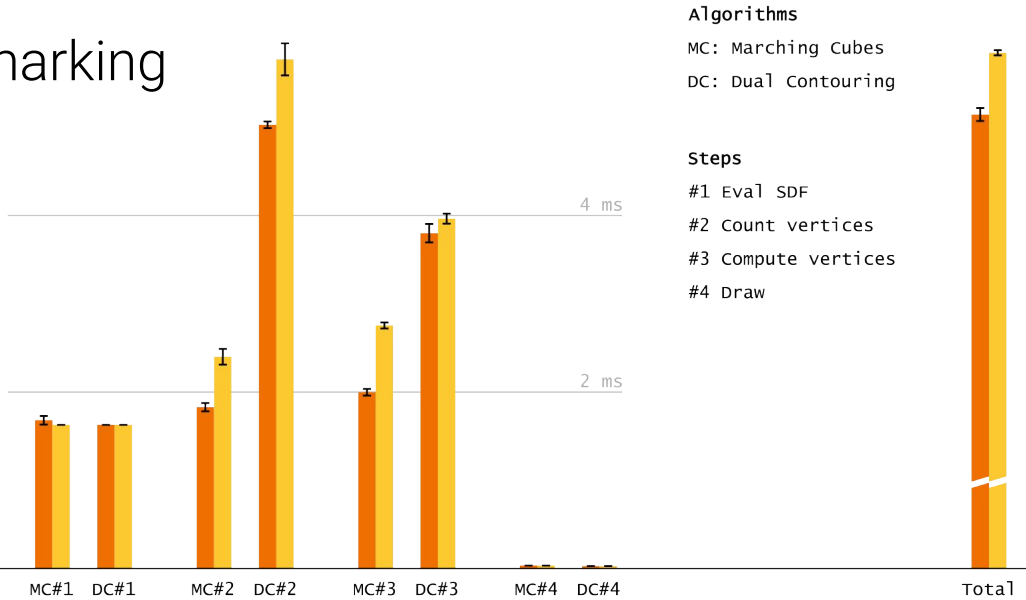
In order to illustrate this profiling part, I've done a little experiment in a scenario that mixes both compute and 3D rendering pipelines. I compare two different SDF contouring algorithms on two different backends.

Debugging

Benchmarking

(gpu time)

■ wgpu-native
■ Dawn



Vulkan backend, Windows 10, Nvidia Titan RTX

Be careful with these figures. I did not go through a detailed benchmarking experiment. Is the difference due to Tint injecting bound clipping when indexing arrays?

Is there an equivalent of chrome-canary's

--enable-dawn-features=disable_robustness that I should have enabled somewhere in the cmake config to make Dawn competitive? Was the test scenario challenging enough to stress test tricky parts of the API? Also this was tested on the Vulkan backend only, how does it go for others? How does this compete with a manual Vulkan implementation?

I think an interesting message here is actually that wgpu-native and Dawn have similar enough performances, showing how WebGPU enables the backend to be quite close to the metal.

Key Points

- State of native GPU programming
 - It's complicated to get a good portable graphics API, WebGL2 is a good candidate.
- WebGL2U for native development
 - How to get initial knowledge.
- Current state of WebGL2U
 - Limitations and remaining design decisions.

Is WebGPU ready enough?

(for native applications)

WebGPU is going to be a thing, that's almost certain, but **is it too early** to make the move?

Is WebGPU ready enough?

Backend support

	wgpu-native	Dawn
Parent project	Firefox	Chrome
Language	Rust	C++
Vulkan	✓	✓
Metal	✓	✓
D3D11	✗	⚠ (via GLES)
D3D12	✓ (W10+ only)	✓
OpenGL ES	ok (no iOS)	✗
OpenGL	✗	✓

Good support for modern desktop

Detailed support matrices:

- wgpu-native: <https://github.com/gfx-rs/wgpu#supported-platforms>
- Dawn: <https://dawn.googlesource.com/dawn/+//HEAD/docs/support.md>

When it comes to desktop application, backend support has become good enough for both wgpu-native and Dawn.

Is WebGPU ready enough?

Some pain points

Overall I like using WebGPU, but there are still a few pain points.

Is WebGPU ready enough?

Some pain points

- Async operations

```
bool done = false;
buffer.mapAsync(MapMode::Read, 0, sizeof(Counts), [&](BufferMapAsyncStatus status) {
    if (status == BufferMapAsyncStatus::Success) {
        const Data* countData = (const Data*)buffer.getConstMappedRange(0, sizeof(Data));
        vertexCount = countData->pointCount;
        mapBuffer.unmap();
    }
    done = true;
});

while (!done) {
    // Do nothing, this checks for ongoing asynchronous operations
    // and call their callbacks if needed
    queue.submit(0, nullptr);
}
```

Async operations sometimes require to write this weird polling loop where I **submit an empty queue** just for the backend to check whether one of my callbacks should be called. There are backend specific alternatives but nothing official.

Is WebGPU ready enough?

Some pain points

- Async operations
- Shader module **introspection**

```
@group(0) @binding(0)
var<uniform> uniforms: Uniforms;

@group(0) @binding(1)
var distance_grid_write: texture_storage_3d<rgba16float,write>;

// [...]
```

WGSL

Mapping binding id to variable name?

When working with shaders, I miss a lot the possibility of **querying binding locations by variable name**, like `glGetUniformLocation` does in OpenGL. In theory one can call Tint or Naga for this but it's highly backend-dependent and not portable to the web.

Is WebGPU ready enough?

Some pain points

- Async operations
- Shader module **introspection**

```
const pipeline = device.createRenderPipeline({
  layout: "auto",
  // [...]
});
```

Implicit pipeline creation

Raised issues: <https://github.com/gpuweb/gpuweb/issues/2470>

```
@group(0) @binding(0)
var<uniform> uniforms: Uniforms;

@group(0) @binding(1)
var distance_grid_write: texture_storage_3d<rgba16float,write>;

// [...]
```

WGSL

Mapping binding id to variable name?

There is an **“auto” layout** that can be used on the web to make binding setup easier, but it has a lot of **caveats** and was **removed** from the native API (likely for the good, it is too “high-level magic” imho).

Is WebGPU ready enough?

Some pain points

- Async operations
- Shader module introspection
- Differences between implementations
 - ♦ Dawn gives better error messages
 - ♦ Dawn can only handle BGRA32Unorm color target (on native targets)
 - ♦ **Disagree on Drop/Release**
 - ♦ Dawn uses FilterMode instead of MipmapFilterMode
 - ♦ wgpu-native has no support for multiview rendering
 - ♦ Dawn has no support for custom render target
 - ♦ W/o stencil, stencilLoadOp must be Clear (wgpu-native) or Undefined (Dawn)
 - ♦ Naga (wgpu-native) is much more strict than Tint (Dawn)
 - ♦ wgpu-native does not support type aliases in WGSL
 - ♦ etc.

And of course there are still many little differences between backends, since the WebGPU API has not reached version 1.0 yet. They are going to converge eventually, but I'd like to focus on one of them: the drop vs release semantics for freeing WebGPU objects. This is the one provoking most #ifdefs in practice.

Differences between implementations

Lifetime management



Drop (*wgpu-native*)

```
wgpuDeviceDrop(device);
```

= "This object will never be used ever again"



Release (*Dawn*)

```
wgpuDeviceRelease(device);
```

= "Personally, I'm done with this object"

```
wgpuDeviceReference(device);
```

= "I'm using this, don't free it until I release!"

*User-exposed
reference counting*

More info: <https://github.com/wgpu-native/webgpu-headers/issues/9>

wgpu-native proposes a Drop procedure, which prevents anybody else from reusing the object. Dawn exposes a kind of **reference counting** mechanism where one part of the code can "reference" the object to increment the counter, then "release" to decrement and the object is effectively destroyed only if its reference counter falls to zero.

Reference counting is more high level, and one can argue that it should be the responsibility of the user of the API to **manage lifetime**. On the other hand, Dawn pragmatically notes that all implementations so far already have a reference counting under the hood, so it does not cost much to expose it.

Is WebGPU ready enough?

Limitations

Timestamp queries

Shader compilation time

What control on caching?

Tiled rendering [\[link\]](#)

Can we detect it?

Other debates: <https://kvark.github.io/webgpu-debate>

Pain points may eventually be addressed. Here are some limitations for which it is not clear whether they will: we have mentioned the security issue there is with timestamp queries.

Another question is **shader compilation**: it is done at runtime, when uploading the shader onto the GPU (because the compilation highly depends on the device), and in applications that use a lot of different shaders this can be a performance bottleneck. In the portable and secure environment that the web is, how can such a hardware-dependent problem be addressed? The application cannot provide its own cache, but could it have control over the browser's or driver's shader cache?

Another example (that I'm not so familiar with myself but I know exists and is a key portability point for mobile devices): some low-energy portable devices (e.g., smartphones) use **tiled rendering** to limit the need for memory bandwidth, how can this specificity be taken care of from WebGPU? Can we get to know about the presence of such constraint? Or is it too much information for the web to leak? There is always a **tension between privacy and performance**.

For more cases of core question points, kvark's "webgpu-debate" pages are a very interesting source of information: <https://kvark.github.io/webgpu-debate>

Is WebGPU ready enough?

Extension mechanism

“I want feature X but it is not in WebGPU!”

➔ Forget about the Web target then.

Generic extension mechanism: **nextInChain**

```
ShaderModuleGLSLDescriptor glslDesc; // in wgpu.h
glslDesc.chain.next = nullptr;
glslDesc.chain.sType = SType::ShaderModuleWGSLDescriptor;
glslDesc.code = glslSource.c_str();

ShaderModuleDescriptor shaderDesc;
shaderDesc.nextInChain = &glslDesc.chain;
// [...]
```

Extension for GLSL shaders

You need something but it is not in WebGPU? If your feature is not in WebGPU, it may be available in one browser or another as an extension, or not available at all on the web but available on desktop (e.g., GLSL shaders, timestamp queries).

The mechanism is always the same: a WGPU “**SType**” is defined as well as an enum key for it. The `nextInChain` pointer points to such a struct and the `sType` field tells how to interpret the pointer.

Is WebGPU ready enough?

Extension mechanism

"I want feature X but it is not in WebGPU!"

➡ Forget about the Web target then.

Generic extension mechanism: **nextInChain**

Custom extensions?

Must support all low-level APIs

Backend-specific

(I did not try it)

I haven't tried myself yet, but one could use this mechanism to add custom extensions by themselves in theory (needs to implement it for all backends, or to advertise correctly in adapter's capabilities whether the extension is available or not). Could be used for vendor specific extensions (e.g., RTX RayTracing kernels, DLSS, etc.) although idk how handy it is currently to integrate this into either wgpu or Dawn backend.

Conclusion

Conclusion

Should I use WebGPU

Yes.

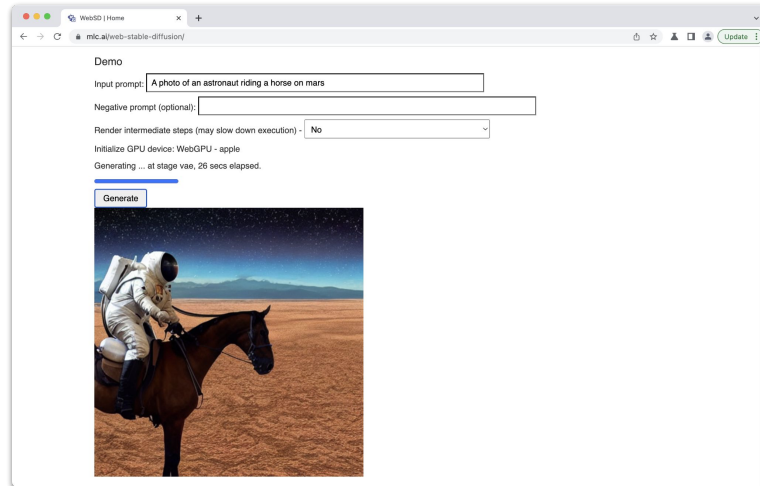
- **Domain agnostic modern API**
(Not too low level, but enable efficient code paths)
- **Future oriented**
(Likely to become the most used graphics API eventually)
- **Unfinished, but very active development**
(Two concurrent implementations)
- You get a **web-ready** code base for free

In conclusion, here are all the reasons why I myself decided to start switching to WebGPU for my prototypes:

1. It is a **nice API**, so I did not have to force myself into switching
2. I am now convinced it **will become the standard API** for graphics programming even for desktop applications, so I don't feel I am wasting the investment
3. The developers are very **responsive**, it's exciting to follow and you get quick answer when you stumble upon issues, there are **active communities**.
4. You get to **run your application on the web even if it was not your goal!** For me it is a real difference, because the Web is a "nice to have" but not a strict requirement, so WebGPU will naturally bring more applications to the web, application that would have not invested in it before. (Of course as of now it still requires special dev flags to run WebGPU in the browser though.)

Conclusion

Not just for 3D stuff



WebGPU for AI: <https://mlc.ai/web-stable-diffusion>

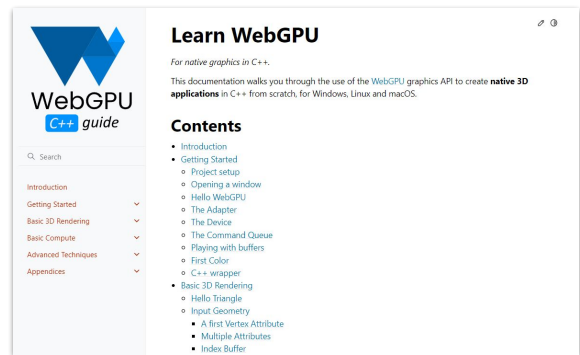
Just to cite one concrete example of a trendy **non-3D related use of GPU programming: Artificial Intelligence** (as we call it)! (Continuous) machine learning is all about big matrix/vector products under the hood, which GPUs excel at doing.

Conclusion

Going further



<https://eliemichel.github.io/LearnWebGPU>



Help around the LearnWebGPU C++ Guide:

 <https://discord.gg/2Tar4Kt564>

Other communities:

 <https://matrix.to/#/#Wgpu:matrix.org>

 <https://matrix.to/#/#WebGPU:matrix.org>

I hope you enjoyed this presentation, again for more operational details I invite you to read my **WebGPU C++ programming guide**. It is still a **work in progress**, but can lead you already to having a simple 3D rendering pipeline up and working, and the very base of compute shaders. More to come there, I am excited about continuing this project and providing a clear documentation for WebGPU! Feel free to also **give feedback** through the guide's feedback bake.

You can join the support Discord server I've open to go with the guide, and of course join wgpu and Dawn online chat on Matrix!

Contact me either there or at emichel@adobe.com

End of slideshow