

GPU Web 2020-10-21 VF2F Day 3

Chair: Coentn / Dean

Scribe: Ken / Austin

Location: Google Meet

[Doc for Day 1](#)

[Doc for Day 2](#)

Tentative agenda

- Method of ensuring GPUShaderModules can contain MTLLibraries [#1064](#) (Myles)
- Capability-querying APIs for GPUAdapter
 - Make GPUAdapter.extensions a setlike interface [#1098](#) (Kai)
 - Add a limit-querying API for GPUAdapter [#1100](#) (Kai)
- D3D12 does not support SRC_COLOR used in SrcBlendAlpha slot [#65](#) (Dzmitry?)
- Reconsider the name `OUTPUT_ATTACHMENT` [#1153](#) (Coentn)
- Multiqueue (Dzmitry, Coentn)
 - Multi-Queue Investigation [#1065](#)
 - Strawman Multi-Queue Proposal [#1066](#)
 - Multi-queue proposal with explicit transfers [#1073](#)
- Proposal for importing Web platform images in WebGPU [#1154](#)
- PR burndown
 - Add aspect back to GPUTextureCopyView [#873](#) (Austin)
 - createBindGroup: Require a superset of the layout's bindings [#1061](#) (Coentn)
 - Add filtered texture and sampler binding types [#1076](#) (Dzmitry)
 - GPUColor: remove sequence overload from the union [#1079](#) (Kai)
- WGSL
 - Project: <https://github.com/orgs/gpuweb/projects/2>
 - Define the interface of an entry point in a WGSL program ([#774](#))
 - Should WGSL support unicode identifiers? ([#1160](#))
 - Reorder expression sections ([#1136](#))
 - Introduce operator precedence table ([#1111](#))
 - Placement of read_only attribute ([#1159](#))
 - what is the initial value of a workgroup variable? ([#1137](#))
 - Entry point taking in/out as parameters ([#1155](#))
 - Buffer indices should be unsigned ([#1135](#))
 - Remove return requirement ([#1156](#))
- Agenda for next meeting

Attendance

- Apple
 - Dean Jackson
 - Myles C. Maxfield
 - Robin Morisset
- Google
 - Austin Eng
 - Corentin Wallez
 - Dan Sinclair
 - David Neto
 - Kai Ninomiya
 - Ryan Harrison
 - Sarah Mashayekhi
- Kings Distributed Systems
 - Dominic Cerisano
- Microsoft
 - Damyan Pepper
 - Greg Roth
 - Rafael Cintron
- Mozilla
 - Dzmitry Malyshau
 - Jeff Gilbert
- Matijs Toonen
- Mehmet Oguz Derin
- Michael Shannon
- Timo de Kort

Define the interface of an entry point in a WGSL program ([#774](#))



- DN: Stack of PRs: 1093 (approved 2 weeks ago modulo “buffer” -> “storage”); 1142 (overview of shader interface; pipeline inputs and outputs; builtin variables) which is receiving review attention from Dan and Dzmitry; and yet-to-be written to add buffer and texture resources, and location counting.
- DN: 1093 has been updated for that rename.
- DS: I’ll take one last look before merging.

Should WGSL support unicode identifiers? ([#1160](#))

- MD: Many languages support this; very beneficial to support in identifiers and source code as well.

- DS: Previously defined source code is utf8, so in theory you could do comments with unicode. With Unicode identifiers, we'd have to normalize them the same for function names that are entrypoints.
- KN: Should be able to just use the plain unicode strings
- MM: Four levels of options:
 - 1. Don't do it for anything
 - 2. Unicode just for var names but not anything else
 - 3. Unicode for everything for everything except entrypoint names
 - 4. Unicode for all items, and everything has to normalize the same way.
- DS: (4) but after MVP
- MM: Think I agree. Not super important now.
- DM: entrypoint names as well?
- DS: Yea, weird inconsistencies if mixed.
- ...
- KN: Still want to hash in the backend output
- DS: Right but that's up to the backend
- MM: Counterargument - homographs - characters that look the same but are not. Could be annoying. Maybe a 5th level is allow unicode but only some of it.
- KN: Someone pointed out that unicode has some rules about identifiers so maybe we can use that. But I don't think we should do anything special about this.
- JG: I expect to be annoyed by this, but given JS supports unicode we should support it too.
- MM: Also if you'll need font fallback, which would break your monospace text editor
- DM: Implementation side, presumably we wouldn't use unicode entrypoints and won't pass directly
- JG/KN: Don't pass that directly in WebGL anyway.
- MM: Source string can't be in dest because of cross-site considerations.
<https://xkcd.com/327/>
- KN: Yea should just hash.
- MD: Wasm defines unicode separately. Maybe someone could look into why.
- JG: Sounds like WASM has a good reason. Limitations of binary format.
- DS: ok - agreed yes post MVP. We can figure out specifics later.

Operator Precedence (Umbrella issue [#1146](#))

- Reorder expression sections ([#1136](#))
 - **Quick Links**
 - **Precedence before:**
 - [gpuweb/pull/1111#issue comment-707941325](#) 
 - **Used to match:** HLSL, GLSL, C++
 - **Precedence after:**
 - [gpuweb/pull/1111#issuecomment-705039742](#) 
 - **Matches:** Python, Swift, Rust

- **What's different:** Bitwise ops get higher precedence
 - **Discussion**
 - KN: C has strange operator precedence, rooted in the fact that C is old and also didn't have booleans. So there's weird precedence in other languages like bitwise and comparisons.
 - KN: I've proposed a partially-ordered operator precedence. (**v2b** in [slides](#)) Tried to get rid of every ambiguous case and leaves us with what I think is a very consistent precedence. Consistent with modern languages and with C. Should eliminate confusion between WGSL and GLSL or WGSL and Swift/Rust
 - DN: I like it.
 - RM: It's also very .. that if we find out that we actually want more ordering we can always add it later. Removing it is less possible.
 - MM: If you wanted to turn this flow chart into a grammar - is it representable, or would it need host parsing checks.
 - KN: It's definitely possible in the general sense of a parser. I don't know if it's possible in the type of specific grammar we're trying to build.
 - RM: I expect it should work. I will try to verify this. I anticipate any issues will be fine.
 - DM: Any reason the bitshifts aren't separated?
 - KN: They could be, though I think it's fairly clear how they behave.
 - DM: I would be worried about it.
 - MD: How would we display it as a table.
 - Various: Just put the picture in the spec.
 - DC: Looking at the bitshifts - are they lower priority than x+y?
 - KN: Not valid to do x+y<<z. That would be a compiler error.
 - DC: Shouldn't they be at the same level of * / %?
 - KN: They technically are.
 - DS: Next step: Robin will try to make proposed v2b as a grammar.
 - DS: Split the bitshifts for now.
 - KN: Can't do x << y << z
 - DS: Can't do x & y | z either?
 - KN: no you can't, but you can x & y & z
 - DN: I think we need a table.
 - JG: And examples might be the best form of clarity.
- Introduce operator precedence table ([#1111](#))
- Partially-ordered precedence ([comment on #1146](#))
 - KN: [proposed partial ordering](#)

Placement of read_only attribute ([#1159](#))

- DS: trying to figure out read_only for buffers and then for textures. This issue is all of the proposed ways to do it. I believe I prefer option 7.

- DS: Comes down to:
 - remove read_only write_only from textures? makes it an attribute
 - then do we put the attribute on the variable? left or right? (I prefer left where it's part of attribute instead of type)
- DN: So if you want to pass a texture into a function, how do you determine the set of overloads? if I have a read only storage texture, and read_only is on the var, and I pass that into a function, does read_only have to be annotated on the parameter separately from the type? And if the callee needs to do something different in codegen?
- MM: I'd like overload resolution to be a function of types, and not anything else.
- DN: I agree
- DS: That works if we put it on the other side of the colon - the right side in the function
- MM: Yes that's what I prefer.
- DM: Does that match array strides when we pass arrays around?
- DS: Yes.
- DN: So basically read_only becomes a type attrib and gets carried through type aliases. So you can do:


```
type RTArr = [[stride 16]] array<vec4<f32>>;
type ro_buf = [[read_only]] Buf;
```
- MM: What's the reason for Possible 2 over putting _ro_ inside the type name.
- DS: How do you do that with structured buffers? The type is "buffer" so where do you put the read_only?
- DM: Could do struct_ro / struct_wo.
- MM: Oh so new keywords.
- DS: Yes. You can either put RO on the struct def, or put it on the type when you assign to var, or change the name to struct_ro. Can't think of another way.
- JG: block_ro ?
- DM: I like the ability to specify access qualifiers near the type but not a suffix. It is consistent and extendable. Possible 2 doesn't mention anything about textures
- DS: Textures would be the same as Possible 1.
- DS: We could even make [[sampled]] and attrib too..
- DM: I think that should be part of the type. No reason to hide it from the type. They already differ in requiring texel format for storage image, vs. sampled component type for sampled image..
- JG: If this were an attribute on a type.. do we have type aliases?
- DS: Yes.
- JG: Is this something you can typedef
- DS: Yes.
- DM: If you have a storage thing, we have to require the access to be specified. Vulkan default is read/write. We probably want the users to always tell us the access. Writing in the spec that you have to put one of the attribs is a little inelegant.
- DS: I was thinking read/write is default and one of ro wo limits it.
- DM: So you want users to get read write textures by accident? That's an extension.
- DC: Default readonly?

- JG: How do you describe writeonly in that case? it would really be `[[write_instead]]`
- DM: I was thinking like what metal has. You have the access attribute and you specify the..
- DS: But then you have to always provide it? what happens if you don't.
- DM: Error on creation
- DS: So we can do `[[readable, writable]]` buffer, and if you don't provide either it's an error?
- DN: When I think of sampled images, you can only do it in a certain way - readable. So i do think we should treat sampled different from storage. Now because we already have two different ways of handling sampled and storage textures, I think it's ok to have a diff way to describe how buffers are accessed. I think storage textures can have explicit read, explicit write,m and one day read write.
For buffers I'm okay with default RW as that's how people usually think.
- MM: Sounds compelling to me.
- DM: For WebGPU it makes a big different if something is readonly or R/W. We make different tradeoffs. I like the idea of readonly by default. If you want more, you say what you want to do. Aside from that, I was thinking `[[access(read_write)]]` buf which matches metal. You have to provide some access qualifier.
- .. does it affect multi Q - no.
- DS: `access(read_write)` just seems like more typing.
- DM: But it would be specified differently. You must specify "access" vs you must say A or A/B.
- MS: What about writable and writeonly? That would give you the readonly default.
- MM: Writeable is both? for textures?
- MS: For buffers for now.
- JG: Weird when the unstated default is a limited one, and then trying... or well those look like modifiers for the default. If the default is readonly, it's weird to modify that with writeonly. You're actually replacing the modifier, not adding it.
- DS: I think the answer is you must provide one. And later we can relax that. Can't go the other way.
- DS: Could leave textures as they are now and just change buffers.
- MM: Intuitively, I'd expected sampled-textures to be common and makes sense for them to be the default. For buffers I could see arguments on both sides for the default. So idk what do do about there, but I do think textures can be sampled by default.
- DN: To understand. For 1D texture, we're have the same ..
- DS: Don't have modifiers on sampled textures, they're just `sampled_texture`.
- DN: Was going to argue that for storage textures, readonly and writeonly are like template params. If it's essential to the behavior to the type then it's a type template parameter.

For storage textures:

- `texture_2d<read,rgba32float>`
- `texture_2d<write,rgba32float>`

- MS: If you're going to require something for buffers, it makes sense to just have `[[read]]` and `[[write]]` and you need at least one.
- DS: So `[[access(read, write)]]` or `[[access(read)]]`
- MM: Can we just accept this and see how it goes? Sampled textures by default and buffers requiring write or read.
- DM: Against the sampled texture by default. They're just different and specifying it is asking for trouble for no reason.
"`[[read_only]] texture_1d<rgba32float>`" should be a different type from "`texture_1d<f32>`", I think
- DS:
 - `var ro_tex : texture_1d<read, rgba32float>;`
 - `var tex : texture_sampled_1d<f32>`
 -
 - `[[block]] struct Buffer {`
 - `[[offset(0)]] a : i32;`
 - `};`
 - `[[binding(0), set(0)]]`
 - `var <storage> buf : [[access(read)]] Buffer;`
- DS: Or swap this so `texture_` is sampled and `storage_texture` is the longer one
- Code snippet in chat:
 - Dan Sinclair4:10 PM
 - `var ro_tex : texture_storage_1d<read, rgba32float>;`
 - `var tex : texture_1d<f32>`
 -
 - `[[block]] struct Buffer {`
 - `[[offset(0)]] a : i32;`
 - `};`
 - `[[binding(0), set(0)]]`
 - `var <storage> buf : [[access(read)]] Buffer;`

what is the initial value of a workgroup variable? ([#1137](#))

- DN: I messed up in the original. in Vulkan workgroup variables are NOT allowed to have an initializer. So if we agree on 0 then that's all good.
- DN: As it stands today, an impl on this on Vk would have to inject extra code into the shader to do the zero init. Painful b.c. you need to write code that works no matter how big the dispatch size is, and then a barrier.. and then the user part of the shader. The impl has to do this.
- CW: Experience from ANGLE ES3.1 that doing this efficiently was not trivial to impl, but we don't really have a choice.
- JG: Any pattern we can use to detect when we don't need to do the extra work? If the user wants to initialize to 1's, are we going to init to 0 and then init to 1.
- MM: Won't the driver get rid of the clear to 0?

- KN: Not trivial because it could be a cooperative thing between multiple shader invocations.
- JG: Then, if we can detect it, can we force people to do it? Spec-wise it's not so different because specs are as-if.
- KN: If they want all 1, maybe we can let them pass that.
- KN: In general I don't think we can reliably detect when a shader initializes everything.
- MM: In metal there's local thread group variables and there are pointers that point to shared memory that's arbitrarily sized.
- DN: This is about the first kind.
- JG: I like the idea of asking the user what they want it initialized to - but couldn't a form of that be to require workgroup vars to be initialized?
- DN: Right, what we have in other things is that if it doesn't have an initializer, we will do it for you to 0.
- MM: One of the goals here is portability. Simply saying that it needs an initializer is problematic because shaders could write different things.
- JG: constexpr.
- MM: So all threadgroup variables are globals?
- Yes.
- JG: I like the idea of an optional initializer and if you don't provide we zero it like everything else.
- MM: Metal has a cool feature where previous dispatches can share the storage. So you can kick off a dispatch beforehand. Not sure if other APIs have that.
- Nope.
- JG: sounds like we need some portable value for initialization. If we can init to 0's then it would be cool if we could initialize to something else as well. Would be bad if we got an extension in the future that made it easy to 0. If we claim this, we can't change the behavior in the future.
- MM: IIRC, this group was discussing clear colors, and if the type of the texture is an integer or floating point texture, you have to use different types. If we allow a sentinel other than 0, we're going to have the same types of problems where the type of the field has to match the type in the shader which is a little yucky. Feel like we should just do 0. If there's an initializer we can have an optimization and behave as-if.
- DN: I prefer 0. Haven't seen an initializer as a use case. Could consider post MVP. To be clear: I would like no initializer, and always make it 0.
- MM: Non threadgroup vars can have initializers, right?
- DN: In private and functions yes.
- MM: So what's the argument to not allow?
- DN: Because it's shared scratch space, it's no so clear how it would be done.
- MM: If it's constexpr, presumably it's the same.
- DN: 0 may be easier for an impl to do because of the type problem you were talking about. memset vs actual values.
- DN: impl could chop up the space for initialization, and zero it cooperatively. Could be an array. You don't know how big it'll be necessarily.

- KN: Yes this is the most common use case.
- MM: Do we support initializers in the module scope for arrays?
- DN: Yes, we do.
- MM: Right so one thread would fill in one segment of the array.
- DM: This is just something we have to do. It's not a feature. I would like the mechanics to be an impl detail. Maybe there will be a Vulkan extension to clear to 0. If everyone starts to init to non zero we wouldn't be able to go the fast path.
- DM: API doesn't tell the choice of fast/slow.
- DM: Can also be a feature for after MVP.
- MM: If we have all this code, we might as well just let people pass a value.
- CW: I made a proposal for such an extension. Because zeros are the same representation for every type there's a clear fast path. The rest is more complicated.
- JG: True. no one is arguing that most people want 0 and will be fastest. But it would suck if you wanted the flexibility of not zero and the impl already has the ability to do it. If they provide an initializer, we can do the slow thing to the custom value. But it would be faster than us doing fast 0 and the user doing slow custom value.
- MM: I'd be uncomfortable accepting an argument that says there might be a Vk extension sometime in the future. We can move one. Let's just accept 0 for now.

Entry point taking in/out as parameters ([#1155](#))

- DS: Came out of me writing WGSL and getting annoyed that in/out were the same between vertex and fragment. Would be nice to not have two declarations. DM proposed entrypoint params instead.
- MM: Did we discuss this a few days ago? I thought we agreed you can't have buffers as arguments to helper functions.
- DS: Right so it wouldn't work for buffers.
- DS:


```
[location(0)] var<out(vertex), in(fragment)> frag_color : vec4<f32>
```

 OR


```
[[location(0, fragment), location(1, vertex)]] var<in(vertex), out(fragment)> colour : vec4<f32>
```
- MM: Can I try to describe another way of phrasing your proposal:
If I wrote a naive program, I'd have one var that's an output from vertex and input to fragment. Your proposal says to deduplicate them.
- DS: Yes, but in the naive version you also need to have different names. This lets the names match up.
- DN: So if I'm in the middle of a helper function that accesses the variable, I need to know if it's been called from the vertex or fragment function. That's what the storage class tells me.
- DS: True.
- DN: Right now it's a unique name and look up the usage right away.
- DS: In SPIR-V we need to list the ids on the entrypoint so we need to know if the helper is in vertex or fragment.

- DN: ...
- MM: When you do variable name lookups, don't you start at an entrypoint and already know what entrypoint you're in?
- DS: In Tint that depends on if you're outputting the entire module or a single entrypoint. We don't necessarily track if it's the entire module. We only deduplicate the function if it uses global vars that are different.
- ... and would need to generate two copies of the function if used from two different stages.
- MM: Which you would have had to done anyway. So seems like there's no downside and only upside?
- DN: I think it looks way more complicated.
- DS: Maybe the answer is we leave it for post MVP. Mostly thought of it because I was writing the same thing a lot. In other APIs you usually don't write everything in one file.
- DM: I think it's a problem and we should solve it, but what Dan suggests is indeed complicated. I think we should have inputs and outputs as args. and bindings are global.
- MM: Does that solve this? Doesn't let you use the same thing for in and out. because output is a struct and input is multiple args.
- DS: The entrypoint way would work. I just don't want to type multiple different names for the same thing.
- MM: The idea Kai was referring to - putting the struct as a param as both the input and as an output. That's what I did for WSL, but I regretted it. It's not a real struct it just holds params you didn't want to type. The compiler had to pack/unpack from this struct for not much benefit.
- KN: Right you need location bindings on the struct members so it's special.
- DM: That's how it looks in Metal, SPIR-V and HLSL - or at least it's an option.
- MM: Also, I regretted this because in the WSL compiler there's this fake struct, but I defined it so it could be nested. If we limit it to not nested, it might solve the concern.
- DM: I think both SPIR-V and MSL limit to depth 1.
- DN: For I/O? I don't think SPIR-V limits it.
- MM: Code to fill in members of the struct was complicated enough that it didn't seem worth it.
- MM: Q about Dan's proposal: what happens when WebGPU adds tessellation shaders or mesh shaders and stuff like that.
- KN: I think Dan's proposal makes it quite simple because it's just one variable that represents two different variables depending on which stage you're in, so it's straightforward to understand what happens. You can reuse as appropriate between different stages if you had a geo shader in the middle.
- DS: And the proposal doesn't make it required. It's only if you want multiple do you need the stages.
- DN: Tessellation has a weird thing where per stage is different from per patch. Quite subtle. Not 1-1. Maybe in that case you don't share and that's ok. Patch variables would be 1-1, but non-patch wouldn't be.
- DS: People like structs better than this for inputs/outputs?

- MM: Required? Can I have some in a struct and some globals.
- DS: I'd say it would have to be a struct.
- DM: Some of the outputs could be pipeline-specific like builtins so you want them outside the struct.
- DS: So you have a fragment struct, and vertex struct.
- KN: Or no struct on the input side.
- MM: Seems like the simple thing would be inputs as args, and they can have one layer of struct
- DS: And there wouldn't be globals. It's either in the struct or in the param list.
- MM: Yea, we probably shouldn't have both params and globals, but if we pick params, it's probably okay to have one layer deep of a struct.
If we want it to be not globals, the vertex shader returns three varyings, and those need to be inputs to the fragment shader.
- DN: Okay, so we're using the return of a struct value return multiple things.
 - Example:


```

          ■ [[stage(vertex)]] fn myvertexShader(
              [[location(0)]] a: i32,
              [[location(1)]] b:f32)
              -> ( [[location(0)]] outa: i32, [[location(1)]] outb: f32) {
                  outa = 2*a;
                  outb = 2*b;
              }
              // because this is an entry point, outa and outb are as if declared as
              var<out>. So the are default-zero-initialized and we can read and write to
              them.
              // If we think about extending this to helper functions, then after-arrow
              declarations are ... an open question because now they might be the
              RHS of an assignment in the caller.
          
```
- DN: We could have an arrow to a list of named tuple parameters..
- MM: Helper functions?
- DS: Okay I'll write up the struct proposal and we can discuss at the next meeting.

Buffer indices should be unsigned ([#1135](#))

Remove return requirement ([#1156](#))

Agenda for next meeting

- Skip next week. Meet on 11/2.
- Cancel 11/3? - keep it for now
- Multiqueue proposal
- GPUShaderModule