GPU Web 2018-07-18

Chair: Corentin / Dean
Scribe: Ken
Location: Google Hangout

Minutes from last meeting

TL;DR

- PR 62 “Remove WebGPUBindGroupBinding.count”: approved
- Apple contributed their WHLSL work in the https://github.com/gpuweb/whlsl repo
- Pass types:
    - Metal team says there is no reason why copy commands are in a pass
    - Metal team says compute passes are to scope resource used.
    - Compute passes are a hint to help implementations
    - AI: Make a list of things they might help for
- Render target aliasing
    - Concern that the render graph might not appeal to all applications, will be rebuilt every frame and there are constraints on D3D / VK for resource placement.
    - Purge / unpurge is nice but the same thing as allocation / deallocation.
    - Discussion around residency in D3D12

Tentative agenda

- PR 62
- Pass types
- Render target aliasing: https://github.com/gpuweb/gpuweb/issues/63
- Agenda for next meeting

Attendance

- Apple
    - Dean Jackson
    - Myles C. Maxfield
    - Thomas Denney
- Google
    - Corentin Wallez
    - Dan Sinclair
    - Kai Ninomiya

    - Ken Russell
- Microsoft
    - Chas Boyd
    - Rafael Cintron
- Mozilla
    - Dzmitry Malyshau
    - Jeff Gilbert
- Joshua Groves

PR 62

- Non-contentious; unanimous approval
- JG: would just like to see an answer from Apple and Microsoft about whether we need a "count" on the bind group's resources
- MM: Apple has no opinion
- RC: no concern; was this originally done to reduce complexity?

New WHLSL workspace

- https://github.com/gpuweb/whlsl
- New intern Thomas on Dean and Myles' team, working on Metal backend for WHLSL.

Pass types

- CW: last meeting agreed we need to have render passes, but don't necessarily agree if we need compute or blit passes.
- MM: talked with Metal team. Got more understanding about why they designed it.
    - 1. For blit commands, every input and output, all the arguments are explicitly inside the particular command itself. The team was saying there's no reason blit commands need to be inside a pass. As long as they weren't inside any other pass.
    - CW: position was that we slightly preferred to not have blit passes. Would make sense for it to match Metal's if we had it. Metal's is more constraining. Vulkan/D3D12, can just clear a texture; can't in Metal.
    - DM: makes sense. My position is that they don't give us anything. If we go the same route on compute passes we wouldn't need them either.
    - 2. Primary reason Metal team said compute passes are important: they scope the resources you attach to commands. StartComputePass, attach resources, do a blit. When ending compute passes, none of those resources are attached any more.
    - DM: good point. Lack of resource inheritance gets in the way of compute passes. Makes sense to me.

- CW: were slightly in favor of compute passes for scoping. In our case it's not necessarily for packaging resources though it would help with D3D12. Would help ensure no image layout transitions in compute passes. Decompose memory barrier placement problem a bit nicer.
- DJ: so we all agree, no explicit blit pass, and yes for compute passes?
- CW: yes. No blit pass, have compute and render passes on encoder.
- RC: one reason we had render passes: better on tiling hardware. We want compute passes so it's easier for the API to validate?
- DM: validation isn't the biggest point - it's the resource binding model, which are scoped to a pass.
- RC: what does begin/end give you? Attach/Attach, Dispatch/Dispatch, then attach another thing?
- CW: if you do Attach, Dispatch, copy in/out of a thing you had attached, attach again, re-dispatch – knowing what resources were touched by what is complicated. If you force to only have attach/dispatch in compute passes, it's more clear what's happening for impl.
- JG: is it? We know what a compute shader is allowed to touch. What its current state is when you call Dispatch. We don't have a bindless model.
- CW: for example: in D3D12, you'll want to fill a DescriptorHeap with all of the data you need for command submission. If you have passes you know exactly the state of the resources you need. Instead of putting some of them in the heap, you issue one more command, you have to split the heap as a consequence.
- RC: so once you start a pass you have to do all your attachments at the beginning? none in the middle etc?
- CW: no, you can do that, just no attachments from previous passes.
- RC: saying it would help on D3D12 because you would attach everything at the beginning.
- JG: sounds like what you want is that this is a hint that you can coalesce stuff.
- CW: yes. Barriers / resources / descriptors / uploads / etc.
- RC: so "stuff" can only happen at the beginning?
- CW: stuff is not attaching resources – it's copying and things like that. You have an online problem with a command buffer – see things one by one, want to do things at the time you see the commands. Doing a pass: doing a bunch of stuff where everything's always the same.
- KR: doesn't this Attach concept restrict the resources you can touch?
- CW: yes, for example can't sample and write at the same time.
- KR: so it's restricting what you can touch from the entire heap to things you explicitly said.
- KN: yes. gives programmer guarantees about what impl will do automatically for you at what time.
- RC: what will restrictions be?
- CW: in compute pass, only commands you can use: Dispatch, DispatchIndirect, SetBindGroup, SetPipeline, (and one more). Restrictions on resource usage.

Can't write and read from the same resource in the same pass unless it's a storage resource and then we insert memory barriers automatically. Constrains what can happen, helps us do a better job.
    - JG: can you write up what this will help with?
    - CW: yes, can write up about this.
    - JG: right now compute passes sound like a hint. Not against those, but would like to know better why it's useful.
    - KN: my intuition is that it's a guarantee that expensive operations won't be done at inopportune times.
    - CW: we'll write about it.

Render Target Aliasing

- MM: got feedback from Metal team that aliasing is important. Screens are big and high density, and don't have a lot of memory on mobile devices. Wasting screen space size render buffers is bad.
- MM: found 3 mechanisms for doing this.
    - 1. At beginning of frame, app tells runtime a plan of: for this draw call, here are inputs and outputs. Before any actual draw call occurs. Then runtime can figure out which resources can be aliased, and move them around.
        - Reason we can't do what native APIs do: they don't provide the safety guarantees the web does. If you use 2 resources on top of each other you get undefined results.
    - 2. Make some objects called Heap. Heap can produce Textures. Any Texture created by a Heap can be combined with any other Texture. Know that this set of resources can all be on top of each other, but can't mix with these others. Graph coloring.
        - DM: one color per heap?
        - MM: yes. Programmer promises that these resources won't be used at the same time.
    - 3. Comes from idea that impl can move resources around. For unrelated reasons, this is a requirement. Browser must be in control. App can provide hints. So, browser is free to move resources wherever. Browser needs to know when putting resources on top of each other whether contents of that resource can be clobbered. If app says it's done with a resource, browser can put any other resource on top of it.
- DM: small feedback
    - 1. Vulkan sub-passes: application has to declare what it's going to do at init time; then it can do multiple graphs per frame. Doesn't make things better; just explaining why it doesn't work for us.
- CW: Apps don't use Vulkan multi-sub-pass feature. Unless they're on a tiler they don't need to explicitly fit things in tiles. Frostbite's presentations do indicate that some apps do create graphs.

    - KN: vulkan supports aliasing, too.
- CW: on heaps: on vulkan and d3d12 where you explicitly allocate GPU memory it's hard to know in advance how big the heap should be. Slight different way: you have a heap, pass it a bunch of texture descriptors. Create this one, or clobber this and create this one. You provide more info up front about what kinds of textures you want to be able to fit into the heap.
- CB: not just about what to create but feedback in case of failure. Would get failure at init time that you're not getting aliasing when you wanted it.
- CW: good point.
- MM: ownership of where resources live needs to be owned by the browser. Browser needs to be able to alias/not alias things based on app feedback.
- DM: don't understand the init error. Talking about using an entirely different format from scratch. Don't see a failure path; not reinterpreting existing data.
- CB: swizzle patterns / page sizes / cacheability are not just pixel format issues. Not sure if hardware needing this still exists though.
- CW: sort of like Vulkan's memory requirements.
- DM: concern about heaps: works if 1:1 replacement between resources. What if allocating 3 smaller textures instead of one big texture?
- CB: this was an explicit goal of D3D.
- CW: Place allocate resource.
- DM: proposed interface does not support this.
- CW: if you say "want this thing here, and that there", it's just one step away from the graph idea.
- DM: problem with graphs is that we don't always know ahead of time what we need. For resources, though, we do know which ones can alias with each other.
- JG: only if you mark the ones you're done with. For example, OpenGL's InvalidateFramebuffer. Otherwise, might need to preserve outputs from render pass. You're saying outputs from render pass are fixed?
- DM: if we have a graph then we can build this model. But providing a graph ahead of time is not feasible. App will want to change it frame to frame.
- CW: Frostbite rebuilds the graph frame to frame.
- DM: explaining the aliasing in terms of which resources can alias would be best. Expand solution (2). Trying to point out deficiency of WebGPU texture heap API. Don't have a feedback on which is best.
- DM: think purgeability API is the best. Think people would enjoy using it. One note: if you start using the resource then the implementation has to ensure it's no longer aliased. Might have to copy things out.
- MM: would you prefer "Unpurge" too?
- DM: not sure. Issue performance warning? Unpurge would help, I think.
- CW: we'll have to check that every resource is alive, eventually. Will want apps to be able to say, now, destroy this resource, even if it's not GC'd. You'll still have a dangling texture object; have to ensure no use after free.
- JG: validation would have to do this anyway.

- CW: purge/unpurge is same validation.
- DM: not concerned about validation as much as hidden cost of binding resoure.
- CW: this is why purge/unpurge is nice.
- RC: Myles, in your code for purge – in the purging world, make any textures at all?
- MM: no, that's not right. I'll update it.
- RC: in impl of CreateTexture, browser shouldn't go find space for the texture? Browser will create it for you, but upon unpurge() would actually allocate?
- MM: such a web standard would be defined by semantics. It's one impl strategy. Later, see what you can coalesce. Both impls would satisfy the constraints.
- JG: how do you handle overcommit? Can conceive of, these two things aliasa because they never purge at the same time. But what if in one frame everything becomes resident, and you don't have enough VRAM?
- MM: naive answer: that should work the same way as without heaps. WebGPU without this: if you make too many resources something bad happens. At which point does the bad thing happen? Resource creation or (with this proposal) silently? Maybe we do need an unpurge function, that's where memory is committed to resource.
- JG: wonder if this is really just the same as allocating a texture and making it more easy to revive it. Questions of whether a texture is alive or dead are similar to unpurged/purged.
- CW: to back up Jeff's point: if you purge the texture and unpurge it, it might have moved. Then have to invalidate a bunch of things that might refer to it, recreate command buffers, etc. If it's explicit then the app knows about this and recreates its command buffers explicitly. Maybe impl has a fast path for reusing recently freed vram.
- MM: that argues for no API changes at all if impl is smart enough to make resource allocation fast.
- JG: maybe we want to preserve TextureViews and reattach them to Textures later.
- DM: re-attaching?
- JG: TextureView represents structure/range of the resource. Purging might be detaching actual Texture resource. It's a view but doesn't own the storage.
- CW: let's avoid the OpenGL model where everything's mutable.
- JG: if that's all mutable state then so is purging.
- CW: agree. Leaves a bad taste in the mouth to reintroduce mutable state that trickles down to other state as in OpenGL.
- JG: already have mutable state in the form of usage.
- CW: it's not mutable state any more. We were arguing for it, but now we have implicit usage.
- JG: it's hidden mutable state.
- KN: it's not trickle-down.
- JG: this sounds like more of a philosophical objection. Want to think about it more.
- CB: on the heaps: allows impl to know outside the main loop how much video memory you have. Worried about cases where you overflow, spill, etc. and won't know until runtime.

- CW: problem is: when you have a very flexible heap e.g. D3D12, your QA says “never go above that limit”. Can’t do this with WebGPU. App would have at loading time to produce all the different graphs that could happen at any time in the came.
- MM: question about mutable state. The way D3D works is if the system gets a low memory warning, thinks it has to move resources from GPU -> CPU, it just notifies the app, right?
- CW: actually that’s what it does. It moves things out from under you. But each time you specify resource you give it a residency list. OS knows to leave those alone.
- MM: is that how bindless works?
- CW: yes.
- RC: in D3D you have to manage residency yourself. If the system gets overcommitted it will put things in system memory for you. In D3D12 it doesn’t have any heuristics, so it could put things in system memory that you use all the time. You’re responsible for your own residency.
- MM: that’s where I was going – the OS gives you a notification that you need to move stuff around.
- RC: you’re responsible for this.
- CW: you say “make this resident now”.
- RC: one suggestion: could we have a variation on heaps, and instead of saying which textures you want memory associated with – say instead “i want all of these textures overlapping with each other”? can’t sample from those simultaneously? Then we don’t have to figure out allocations as we process and scan commands. Know more things ahead of time.
- MM: yes, you’re saying the texture / heap creation function needs to know about which textures you’ll produce?
- RC: yes, e.g. all of these textures *must* overlap with each other.
- CW: think that will work. We’ve been debating whether for example you want 1 vs. 3 textures overlapping in the same region.

Agenda for next meeting

- Google: Figure out how to use the Hangouts Meet version
- Have a F2F! September 27, hosted at Apple Campus (thank you Myles!)
- OK by Dean to move the meeting an hour earlier.
- CW: do a writeup of compute passes for the next meeting
- Research more on aliasing, continue discussion.