GPU Web 2020-03-16

Chair: Corentin
Scribe: Austin
Location: Google Meet

TL;DR

- Next F2F location still TBD, waiting to see if we can colocate with the next WebGL F2F
- Managing on-chip memory. #435
    - Discussion on how this could be extended to full multipass (and not just 2).
    - Concern about how it interacts with Vulkan render pass compatibility.
    - CW and JG to study the implementability of the proposal.
- Data upload (#605 is new)
    - Q&A on the proposal.
    - Concern that it would be good to have a size and offset for mapping for reading.
- PR Burndown
    - #613 Merge arrayLayerCount into size.depth
        - Confusion on what arrayLayerCount is for cubemap views.

Tentative agenda

- Managing on-chip memory.
- Data upload (#605 is new)
- PR burndown
- Agenda for next meeting

Attendance

- Apple
    - Dean Jackson
    - Justin Fan
    - Myles C. Maxfield
- Google
    - Austin Eng
    - Corentin Wallez
    - Kai Ninomiya
    - Shrek Shao
    - Ryan Harrisson
- Intel
    - Yunchao He

- Microsoft
    - Chas Boyd
    - Damyan Pepper
    - Rafael Cintron
- Mozilla
    - Dzmitry Malyshau
    - Jeff Gilbert
- Mehmet Oguz Derin
- Pelle Johnsen
- Timo de Kort

Administrivia

- CW: Next F2F is cancelled. The one in May. Briefly said the next one would be in Toronto, but it seems we might be able to do it in Phoenix at the next Khronos F2F. DO we want to co-host there? or would we rather do Toronto?
- DJ: We should wait to see if Khronos confirms, and if so we should do that.
- KN: +1. Khronos' plan was to do the one after Phoenix (edit: which is Osaka) for next time (edit: in January). We could do Toronto then. Might be the winter.
- JG: Let's wait, but generally we should co-meet whenever WebGL WG meets in North America.

Managing on-chip memory. #435

- JG: Sounds like there are only two subpasses. That's a limitation.
- MM: Yes. One of the goals was to try to make it as small and simple and non-scary as possible so people use it. The reason was that feedback that VkSubpasses were too complicated. Wanted to make this as simple as possible. Also, the only use case was deferred rendering which only needs two.
- JG: The default for deferred rendering is two, but I don't think we should have that restriction. My understanding of the feedback is not that subpasses are hard, but that subpass dependencies are hard while getting performance right. So I don't think from a user-facing perspective that if the user doesn't have to dictate dependencies, it's much less scary than Vulkan.
- MM: Thanks. that's pretty good feedback.
- CW: How would you remove the subpass dependencies while still able to translate to Vulkan?
- JG: The way I read the proposal, you say what the intermediate formats are, but you don't say what barriers you have. Assuming that it's possible for us to implement this well, it should be possible to implement N subpasses well.
- KN: The way VkSubpasses work is you describe a DAG and I think that if we had users describe the resource dependencies in a relatively simple way, we could do it. But you have to know ahead of time before recording the subpasses what the dependencies are.

We could do that in the implementation, but that requires deferred recording of the pass into the command buffer,

- MM: Yea I originally thought it was a DAG, but I don't think that's true. Each subpasses is a part of the graph. If you took an acyclic graph and did a coloring, and put all the matching colors in the same pass.. that's what you supply. It's not an arbitrary graph. I agree that in Vk the deps are very explicit, and in this proposal they're intentionally not explicit. We're trying to find a medium between powerful and very useful. If we wanted multiple subpasses rather than just two, I think the logical way to do that is there could be a two dimensional array of intermediate formats.
- JG: And this is only for pixel local storage, not image block storage, right?
- MM: Yes in Metal you can only access the one texel that's your frag coord. The only way you get the whole thing in Metal is a Tile shader. We're not proposing that.
- CW: One worry of more subpasses is that the subpass dependencies between parts of the subpasses are part of render pass compatibility. It can influence the way the driver does allocation of stuff in the tile.
- JG: Wasn't that removed from forward versions of Vulkan? I remember an issue like this.
- CW: No, that was multisampleResolve for a single subpass renderpass.
- CW: Maybe we could say WebGPU runs in a restricted mode and extend usage validation to go across subpasses. This way we know what pipeline barriers to make.
- DM: Are you trying to eliminate the problem of external dependencies?
- CW: Eliminating the problem of Vulkan subpass dependencies being part of render pass compatibility.
- DM: Right, so if we deduce the deps based on what the user provides in commands, then this won't work. We may need to make a new RenderPass instance, etc.
- CW: Sorry wasn't clear. We could say that … so you don't have a thing where subpass A writes to a storage buffer and subpass B reads from it. That's not allowed. The idea is to keep one WebGPU pipeline one VkPipeline by forcing all subpass dependencies to be the same.
- JG: So in the case of no subpass dependencies.. do we know when we create the pipeline or is it always this two pass mode? Might be misunderstanding. Need to look into this more anyway.
- CW: Like MM said, subpass deps are like the coloring of the DAG, and we need the shape of the DAG at pipeline creation time.
- JG: Wonder if part of the problem with subpass deps is a tooling problem. If you're generating stuff on the fly, it'll be hard to build good subpass dependencies out of it. One way we could potentially improve this is by making it easier for real world content to use it.
- CW: We worry that two-subpass model is not enough for real world content..?
- JG: No I think there will be content that uses it. I would say it feels artificially restrictive and not a huge simplicity over having N subpasses.
- MM: One of the things people may have missed that I'd like to call out is that in this proposal, the new objects that describe the intermediates are not dictionaries, they're actual objects. This is so the browser can detect what type is passed and we can extend

it later. Today there's an IntermediatesDescriptor, but in the future we could have a MultiIntermediatesDescriptor. It's extensible. Maybe it makes sense to start with something simple like this.

- CW: Internally I was asking if we have data on how multi subpass renderpasses are used on shipping titles. Do you have data on this from the Metal side with PixelLocalStorage? Is two subpasses enough for most titles?
- MM: We've found that because ImageBlocks are significantly different from any other API, developers generally don't use them and do deferred rendering using the slow mode.
- JG: Makes sense. Useful to reframe that if we can get two subpasses to be acceptable, we should look at N. But 2 would be better than where we are today. So on our side, that would be more work on investigating how the implementation would work.
- JG / CW: Yea can look into it. Can't promise an actual implementation.

Data upload (#605 is new)

- CW: tries to address the concern about MapAsync over-copying things. There's still MapAsync. Once the Promise resolves, the buffer is in the mapped state and you can ask for specific ranges. Also removes CreateBufferMapped because now the buffer itself hands out mappings. It's now a creation argument. Personally, I think it's an improvement over MapAsync in almost every way.
- MM: Is this going to be used for uploads and downloads or just downloads?
- CW: Idea is it's both. Download case: Create buffer with MAP_READ | COPY_DST. Copy into the buffer, then MapAsync(). GPU process will send data back to the Renderer. Indeed there is overcopying for the Read usecase. The whole buffer needs to be copied over every time. I think it's fine to assume that people will create buffers correctly sized for reads. Alternatively, we can do dirty range tracking specifically for MAP_READ buffers, and copy over only what's been touched.
- MM: What was the reason for not giving offset/size to MapAsync?
- CW: Need to know at frame N-2 which ranges you will need.
- MM: If a developer didn't know, couldn't they just pass args for the whole buffer, and get the same as what this proposal is today.
- MM: And you say this is in addition to the writeBuffer API?
- CW: Yes.
- MM: I think if both are true, then this satisfies me.
- DP: In the upload case, I call MapAsync, and at some point the promise completes, then I call GetMappedRange() then Unmap() then Submit(). Then I immediately called MapAsync again. What triggers MapAsync completing?
- CW: Completes not before the buffer is no longer in use. Not after all previous commands have been finished.
- DP: If I wanted to have a single large buffer that one half the CPU writes into and sort of double-buffer within one buffer, that's not supported?

- CW: No it is not. The idea is these buffers are in the Upload heap. so you always copy from these buffers. You don't need to do a real ring buffer. You can still have a large uniform buffer which is copied into from this staging.
- DP: Think I confused myself.
- CW: MapWrite is a D3D12 Upload Heap. I think we should not allow people to shoot themselves in the foot and use that as vertex data. So you don't really need to use double buffering because there's always staging.
- JG: What happens if I do MapAsync() and then enqueue writes into the buffer?
- CW: You can enqueue those commands, but you can't submit them. Queue submit validation will check that all buffers used are in the Unmap state.
- DM: Looks like GetMappedRange is not useful for readbacks. Since you have to read the whole thing from the GPU.
- MM: That's what I was saying. Think it would be better if MapAsync had args.
- CW: Could split back into MapReadAsync and MapWriteAsync. MapReadAsync would return a Promise<ArrayBuffer>
- JG: Still think it would be cool to map subranges of the buffer. But if we had that then you would want to map more than one subrange at a time.
- CW: What I'm hearing is wanting to map subranges of data so we only need to send that data back. Overall seems fine but we need to refine the readback use case.
- DM: Also, I think the proposal works well when the implementation is required to keep the staging buffer around for the lifetime of the buffer.
- CW: DM is right. The assumption is that when you create a mappable buffer, there's a persistent shared memory region backing it.
- RC: I have a question about multiple frames. On the first frame I ever render, I make a fresh new buffer, write to it and submit it. Later on, if I call map async, how do I know when it will come back?
- CW: In a way it's similar to the current MapWriteAsync. Developers should re-request as soon as they have finished using the buffer.
- JG: If using a promise like this makes people happier than using fences, then I'm fine with it. I think I do want to see subrange support. I don't think there's anything that blocks this though.
- CW: (...) I gave up on subrange proposal.
- JG: This proposal is sort of the same as migrating a buffer to state mappable, waiting on a fence, and then calling map after that. If you do it wrong, then we'll generate an error.
- CW: While I agree it's very similar to fallible mapping, I think that this proposal helps simplify a lot of the cross process shenanigans.
- JG: I like it, I'd like to expand on it.
- MM: If you do expand on it, please describe how it works with multi queue.
- JG: Think it would work the same. MapAsync with a promise is just a simplification of migrating to another queue or mappable state, then waiting on a fence, then mapping it.
- DM: My thoughts: It doesn't work well for reading. It doesn't make much sense for small writes either. Think that if we have writeToBuffer, we could just have the MapReadAsync() and MapWriteAsync() that just work with a single range. The PR

brings more complication to the API in order to get close to the ability to upload small ranges. Think the direction is slightly misleading because writeToBuffer is already there.
- JG: I'm not assuming that. It doesn't color my feedback whether or not WriteToBuffer is there. I like this anyway.
- MM: I actually agree with everything DM said, but in the interest of progress, I'd like to see this move forward.

PR burndown

- #613 Merge arrayLayerCount into size.depth
    - Changes GPUTextureDescriptor to get rid of arrayLayerCount and uses size.depth instead. In the past, one of them always had to be 1.
    - JG: Makes this match what GL does basically? Where 2D array "depth" is the number of slices and not actually the depth of the slices.
    - DM: Raised a concern on the PR.
    - KN: Do you think that is a spec level thing or against the concept?
    - DM: Cube Arrays…
    - CW: That's a property of the view, not the texture.
    - JG: Was ambiguous before and the change makes it maybe more ambiguous. Number of cube maps or the number of cube map images?
    - KN: Was an issue before this change as well. It's number of cube map images.

Agenda for next meeting

- KN: CTS problem if not figured out on email list. Would like more feedback there if people have time.
- JG: Buffer mapping
- JG: On-chip memory
- YH: Query API