So as to not be anonymous animals, the doc is shared for writing with the google accounts that are invited to the meeting. If you can't edit let cwallez@google.com know. Also be sure to be in "Edit" mode and not "Suggest" mode.

GPU Web 2020-07-13

Chair: Dean
Scribe: Austin / Ken
Location: Google Meet

Tentative agenda

- (Re: Re: Fwd:) Depth-stencil copies, again. #652 (comment)
- robust access of vertex and index data (again)
- Index format being part of the pipeline #767 (alternative proposals)
- PR burndown
- Agenda for next meeting

Attendance

- Apple
  - Dean Jackson
  - Justin Fan
  - Myles C. Maxfield
- Google
  - Austin Eng
  - Brandon Jones
  - Idan Raiter
  - James Darpinian
  - Kai Ninomiya
  - Ken Russell
- Microsoft
  - Rafael Cintron
- Mozilla
  - Dzmitry Malyshau
  - Jeff Gilbert
- Kings Distributed Systems (KDS)
  - Daniel Desjardins
  - Wes Garland
  - Dominic Cerisano
- Matijs Toonen

- Mehmet Oguz Derin
- Timo de Kort

(Re: Re: Fwd:) Depth-stencil copies, again. #652 (comment)

- Updates on behavior in D3D12 (by spec) and Metal (mostly anecdotal)
- KN: IIRC behavior was previously undocumented; now it's documented
- KN: read docs Rafael sent. For copying stencil aspect of combined D/S, they do get tightly packed, like in Vk. More strict: in D3D12 for depth copies they zero out the unused 8 bits. Stencil copies work exactly the same way.
- KN: Metal documentation is buggy IMO. Says something's true for both depth/stencil, but then later talks about how it works for depth, ignoring stencil. Austin tried it, finds it does pack stencil bits tightly. Seems the behavior's the same everywhere and we can include copies into/out of stencil aspect of D/S formats in the API. Just have to verify Metal's behavior works the same everywhere and isn't hardware dependent.
- DJ: even if behavior's that way, still want to double-check with the Metal team to make sure it's not an accident or documentation issue. Let's assume the results are correct, and we'll take an AI to confirm with the Metal team.
- MM: I think I already have an AI for that.
- MM: what decision does this evidence affect? What will we do with this?
- KN: helps us decide whether we can allow copies to/from the stencil format. E.g., said we wouldn't do copies to/from D24+ aspect. But for stencil aspect, want to make sure we can do it without repacking overhead. All APIs, when you extract stencil bits, give you 8888, so no emulation / repacking needed. Good candidate for inclusion in the API.
- MM: so strategy is that copies to/from depth/stencil formats should never turn into compute shader invocations?
- KN: the ones we're settling on now, yes. More complex impls, we could work on later.
- MM: so if authors do want to copy depth aspect they'd have to write the compute shader?
- KN: yes.
- MM: and if copy becomes faster on future versions of Vk, etc., they won't get the perf benefit?
- KN: correct. I don't want to say this is excluded for 1.0; just tabled for now. Converting copies from ~2 weeks ago aren't off the table for 1.0. Should revisit for exactly the reason you raise.
- MM: OK.
- MM: if all impls pack stencil the same way, we can say it's enabled on all platforms?
- KN: correct - don't need any polyfilling compute shader.
- MM: I'll take the AI to investigate how the stencil aspect of copy commands is handled
- DJ: Kai, you'll update the spec to describe this?
- KN: I'll take an AI to figure out spec changes.

robust access of vertex and index data (again)

- Outline of things to discuss:
    - 1) An out-of-bounds vertex read in Vulkan can be several things (zero, any value from within buffer, (0,0,0,x) vectors). What is allowed in the spec?
        - DJ: do we want it to be interoperable? Strange that it allows you to do so many different things.
        - JG: the vector rules are that's so if you have an RGB type it can return (0, 0, 0, 1).
        - DJ: what about (0, 0, 0, MAX_INT)? (wonders mostly to himself)
        - MM: rules should be same as shading language - any of these values are OK - spec should document what's allowed. We already discussed this in the past.
        - DM: worried that we can have an app that has just a few indices out-of-range, will fail on just a few platforms. To me, using robust buffer access means that our API is much less portable.
        - JG: agree it's less portable, but don't think much less so. We have this in WebGL, it's not that bad, haven't heard complaints, and we're shipping robust buffer access in a lot of places. We have a pref in Firefox to force CPU-side validation for this, can help for testing. If you flip the pref you should get a warning in the console.
        - DJ: you can't tell for everything, right?
        - KR: WebGL doesn't have vertex pulling.
        - DJ: if testing impl on something that returns something from inside the vertex buffer, everything looks fine, but on impl that returns 0, it looks badly broken
        - MM: Clarification: of course interoperable behaviors would be preferable. The question is what the perf cost is. Maybe we need an investigation.
        - KR: It's significant. It was such a big cost in ANGLE that Intel made a patch to use RBA. I think we should break down the problem piece by piece. WE should have consistent rules about what the hardware does and expose that. If we can have validation layers, that would be good.
        - JG: Would mention it changes backwards-compatibly if we change this in the future. Could actually defer this for now as long as normal RBA rules are okay.
        - DM: would like to know more about previous benchmarks done in WebGL group. If you have to validate all access to uniform buffers we wouldn't have that here - we have minBufferSize. We have indirect draws too, validating requires CPU side validation.
        - JG: Don't know if we still have the benchmarks, but this is a pretty important that that multiple vendors implement already. I think we should do this (perf investigation) outside of this discussion.

- DM: Our decision does depend on perf though.
- JG: I know what you mean, and it would be nicer to be more portable, but if we can hit the minimum viable product that WebGL has w.r.t to this, then we should do that, and then see if we can do it better and risk backtracking in a non-backwards compat way.
- MM: I think the MVP argument would be the other way. If we should start with an MVP, we should be most restrictive in the beginning.
- JG: You want to be more restrictive in a way that is backwards compatible. In an MVP case we can go from loose RBA to tight RBA, but we can't go the other way. If we have tight RBA and decided we can't ship and loosen it, that's backwards incompat. If we want backwards compat we need to start loose and then become strict.
- DM: Important thing here: when we're talking about using RBA, we only mean Vulkan.
- KR: No it is used in D3D12.
- JG: D3D12 loosened that up
- KN: Relative to D3D11.
- RC: it depends. For things like index buffers, indexing out of bounds protection is on by default. If you yourself update some descriptors to root descriptors, then those APIs just take a GPUBufferAddress, and you don't get RBA. Newer APIs like RT and mesh shaders do that as well. In general I agree with Ken and Jeff that we should use these RBA facilities and not do our own. I think WebGL made a good choice here.
- MM: Maybe we should proceed for now, and someone, perhaps DM, can make an investigation and if necessary reopen the issue. Is that fair?
- KR: I think that's fair. Pointing out that Idan is actively working on this now, and should get some agreement.
- DM: IIUC, D3D has more strict semantics than RBA? I'm arguing to use precisely what D3D says (sampling outside gives 0s)
- JG: Not necessarily what D3D provides. RBA stuff in vulkan is a child of the RBA that MSFT asked IHVs to put in hardware in D3D11. That's why it's there in OpenGL and in Vulkan. D3D has loosened it in some cases, but it's the same sort of idea which is why it functions the same in newer APIs.
- DM: Do we have a place where that is properly investigated (d3d12)?
- RC: I have not done a perf investigation, but if you want more details on when it does/doesn't happen I can give that to you.
- DM: Yes, that's what I mean: what does D3D12 guarantee?
- RC: vertex / index OOB is always on by default, and any descriptor on the heap that has a size, that one has RBA on by default. Descriptors as gpu virtual addresses in root descriptors (no size), there is no RBA. Newer APIs take virtual address and there is no RBA.
- MM: What other way of checking would work?

        - RC: Have the impl put the size in a constant buffer and check it in the shader.
        - MM: Means you have to use vertex pulling?
        - RC: Yea, or on the CPU you have a shadow copy and check them all.
        - MM: That’s what I was getting at. Looking at all the indices is an O(n) operation which is no good, especially when the indices are populated by the GPU. In WebGL we store the maximum index of an index buffer and keep it updated throughout time. Can we use the same sort of trick?
        - RC: The impl is up to us of course, but we have to give the shader more information so we can check to see if the operation is valid.
        - JG: In WebGL it’s not sufficient to store the max index because you can use subranges. You need to do full subrange tracking
        - KR: What we’re talking about is effectively bindless, not about index buffers specifically.
        - KN: To clarify, RC said when you access a vertex buffer in fixed function (not vertex pulling) you always get RBA.
        - RC: Yes, but in new APIs like mesh shaders, you give buffers, called vertex, but you get no RBA.
        - MM: Getting back to just talking about indexed draw calls, is one of the acceptable behaviors to mark the draw an an error and not execute at all?
        - IR: We do have an issue on this: #320. It seem back then we wanted to allow discarding.
        - JG: FWIW that's the WebGL behavior - either the draw call's invalid and produces and error, or uses RBAB.
        - MM: so if we have multiple behaviors, then one acceptable behaviors should be dropping the entire draw call
    - 2) For out-of-bounds, do we care about the size of the buffer, or the size of the binding? (Vulkan RBA will allow any value from within buffer, even outside binding).
        - IR: if you only bind a small chunk of the buffer as the binding, we can still give stuff outside the binding, as long as it's inside the buffer itself.
        - DM: apps that use large buffers and sub-allocate out of them will give random values.
        - MM: Goes back to the perf question. We could enforce stronger guarantees, but the question is how much it costs.
        - KN: for vertex buffers think we don't have a choice but to follow what we do for #1, above.
        - IR: if we decide to use Vulkan RBA, we just wanted to say that this is something that'll happen, and should be noted. See if it breaks anything for users.
    - 3) Can an implementation no-op out-of-bounds calls (not planning to, but we should discuss if valid)
        - (based on discussion above, yes discarding is allowed)

- 4) When buffer is too small (index / vertex). We have a few options:
    - Raise validation error
    - Clamp the call (reduce indexCount / vertexCount)
    - Rely on RBA to fetch the invalid vertices
    - (Note validation error can't happen for indirect calls, it'll have to be one of the last two options with clamping possible using a compute shader before the call)
    - The choice here could affect pipeline statistics queries for how many vertices were drawn. Might have to choose more specific rules.
    - JG: is it just pipeline statistics?
    - IR: not sure
    - KN: I'm not too worried about pipeline stats - those numbers can't be spec'd anyway. More imp't question is whether you are guaranteed to get a validation error or not.
    - MM: why is this different from question (1)?
    - IR: there are easy cases where we have to rely on RBA to check them - if make indexed draw call and the index count is way too big, outside the index buffer. Do we want to add validation errors to the spec?
    - JG: probably more hesitant to do that because of GPU-sourced commands. Had a lot of issues in WebGL where different browsers would no-op things in different places. Draw call, draw 0 points - would you no-op it or run all of the pre-draw validation incl. framebuffer checks?
    - IR: does that mean you're leaning toward still doing the draw call?
    - JG: I prefer to have it do as much validation as the general-case command would do. Do all the validation, decide you draw 0 points, and then bail. No early-outs. Keeps the spec simpler.
    - IR: makes sense.
    - AE: above, we decided we'd allow discarding draw calls if they're out-of-bounds. This is in that category.
    - JG: think we have leeway where we can provide message through web console, as well as through ErrorScope. Apps whether in development or live on the Internet have these messages in the error console. Tells us what to look for.
    - KN: open question whether we want to expose an error when a draw call gets no-op'd, since that's only one impl possibility. Exposing error at API level should be considered harder before we do it. In general, I'm in favor of using mechanisms we already have in place, like robust buffer access, to do things we'd otherwise validate. Would be really nice to say there were a way to tell the implementation to check all possible error cases and tell me if I've gone out of bounds. Might be slow, but would eliminate undefined cases.
    - MM: I agree with that. Could be a validation or something in the Developer Tools that could be enabled.

    - 5) Indirect calls
        - AE: If buffer containing indirect arguments is too small, think based on this discussion we'd do the same thing - clamp, zero, etc.
        - MM: so if buffer contains just a single float or int? Do something like replicating the value multiple times?
        - AE: could also do validation of DrawArrays that your vertex buffer contains at least vertexCount values.
        - KN: not sure whether we should do this CPU-side or not. Pretty cheap.
        - MM: all of these - DrawIndexed with index buffer too small, DrawArrays with vertex buffer too small, DrawIndirect with indirect buffer too small. should all behave the same.
        - KN: I agree.
- DM: I'll try to get some numbers. Concerned about perf:
    - RBAB in Vk still has a cost. If we do it ourselves we can do better in some cases. With minBufferSize we can do better with structured fields. With vertex pulling, this is what AMD does internally. Maybe the overhead isn't that high. Hope we can get this more portable than returning any random numbers.
    - DJ: also interesting to know, of the impls, how accurate are they.
    - MM: you mean which devices have which behavior?
    - DJ: which devices support RBA behavior but don't implement the correct semantics. Maybe none - not sure you have a conformance suite for this.
    - MM: DM, I'd like to see your numbers.
    - RC: for D3D, the robust buffer access behaviors have tests. They're very stringent. For index buffers, vertex buffers, etc. (for instance), RBAB defines to return a 0 index. And 1.0 can sometimes be in the alpha channel. Similar for compute shaders.

Index format being part of the pipeline #767 (alt proposals)

- API surface tweaks for index formats / primitive restart values
- KN: Made the following proposal in the issue:
    - A) `{ primitiveTopology: "line-strip", vertexState: { indexFormat: "uint16" } }` and in drawIndexed it's validated that the current setIndexBuffer's indexFormat matches the pipeline's indexFormat
    - B) `{ primitiveTopology: "line-strip-uint16" }` and ^
    - C) `{ primitiveTopology: "line-strip", vertexState: { primitiveRestartValue: 0xFFFF } }` and this pipeline can only be used with setIndexBuffer("uint16")
- Same validation, but maybe we should lower this to be closer to underlying APIs. And make shape of the API more flexible. If we can support custom primitive restart values we could do so in the future.
- MM:
    - Option (b) sounds terrible
    - Option (c) should be an enum rather than an integer

- RC: for (c) could someone turn off primitive restart?
- KN: that's something that seems more possible to add as extension later on. Works with any of these possibilities. Index format undefined == use it without (?) primitive restart later on. Unclear way to spec that primitive restart's disabled though. Better to say, primitive restart on...
- MM: Metal doesnt' let you disable primitive restart.
- KN: right. If we can't do this now then it could be an extension. Talking about making the API more flexible for future extensions.
- MM: I don't think this is particularly important. The extension can solve this.
- RC: if you use (a) and have undefined vertex format, how do you know if the indices are 32- or 16-bit?
- KN: we don't allow undefined with the line/tri strip primitives.
- RC: if you put undefined there, and line strip, that's how we'd know there's no primitive restart value?
- KN: if we had an extension that let you disable primitive restart, yes, with the current API that's how you'd specify it.
- RC: so I'd say line_strip, uint16, and something else to say no restart?
- KN: if we had index format in the PipelineDescriptor, like now, yes that's how you'd specify it. But line strips (?) without index format don't need to specify ahead of time. (?)
- RC: how do you know 16- vs. 32-bit indices?
- KN: SetIndexBuffer in the Command Buffer.
- RC: and what about mismatches?
- KN: mismatch on draw call during command encoding. Current index buffer != current pipeline.
- MM: of options (a) and (c) if you set the index format to uint16 and only use uint16 when encoding command buffers, ..., can only use uint16? Same ways of describing the same concept?
- KN: yes, all supposed to be same ways to describe same concept.
- RC: if I put index format "undefined" but then put uint16 and call SetIndexBuffer, that's how you know primitive restart is disabled. What about putting index format uint16 and later SetIndexBuffer undefined?
- KN: if using strips then index format is required in pipeline desc. draw time, if index format was spec'd in pipeline, that index format == the one you specified with SetIndexBuffer.
- DM: I like (a) because it doesn't need the spec to define a mapping between something else and the index format. Just say it's the same value here and there. I also added (d) to the issue reducing redundancy a little bit.
- DJ: does anyone like (b) or (c)?
- KN: I proposed them - think small value in (c)
- JG: I see a little value in (c). Users might get a weird error like their primitive restart value isn't the magic value we expect. Could open it up later. But motion in the APIs has been to allow it to be settable, but in most APIs it's mostly supposed to be all 1's depending on the bit width.
- MM: I read (d) on the issue and like it the most. Splitting concept of strips vs. not-strips seems like a better way to solve the problem.
- JG: I'll have to think about (d) more.

- KN: I made a comment about taking index format out of vertex state. Not opposed to the idea.
- DJ: we'll come back to this next week then. Come with your favorite alternatives.

PR burndown

- Let's' do this next week.
- MM: can we do maxBufferBindingSize? #874
- MM: right now if you pass 0 and nothing, they mean the same thing. If we have two different behaviors: do validation at draw time, and ahead of time - passing 0 rather than nothing should be distinct. 0 means ahead of time validation, passing nothing means do draw-call-based validation.
- DM: ahead of time validation is not possible with the value of 0. We agree to include at least 1 element of an unsized buffer.
- MM: I see. Argument is weaker now, but still existent - using out-of-band messaging is preferable to in-band messaging.
- DM: I buy the argument, but want to note that on the C bindings side it will still be 0.
- MM: native bindings can do whatever they want.
- KN: I'm split on this. Is this field an indication to do validation ahead of time, or indication of what min buffer binding size is? In the latter case, 0 *is* literally the min buffer binding size. In the former case, then makes sense to use `undefined`.
- DJ: let's keep discussion going in the issue.

Agenda for next meeting

- To be typed in here offline