Why is a single depth buffer sufficient for this vulkan swapchain render loop?

Asked 4 years, 6 months ago Modified 7 months ago Viewed 4k times

13

I was following the vulkan tutorial at https://vulkan-tutorial.com/ and at the depth buffering chapter, the author Alexander Overvoorde mentions that "We only need a single depth image, because only one draw operation is running at once." This is where my issue comes in.

I've read many SO questions and articles/blog posts on Vulkan synchronization in the past days, but I can't seem to reach a conclusion. The information that I've gathered so far is the following:

Draw calls in the same subpass execute on the gpu *as if* they were in order, but only if they draw to the framebuffer (I can't recall exectly where I read this, it might have been a tech talk on youtube, so I am not 100% sure about this). As far as I understood, this is more GPU hardware behavior than it is Vulkan behaviour, so this would essentially mean that the above is true in general (including across subpasses and even render passes) - which would answer my question, but I can't find any clear information on this.

The closest I've gotten to getting my question answered is this reddit comment that the OP seemed to accept, but the justification is based on 2 things:

- "there is a queue flush at the high level that ensures previously submitted render passes are finished"
- "the render passes themselves describe what attachments they read from and write to as external dependencies"

I see neither any high level queue flush (unless there is some sort of explicit one that I cannot find for the life of me in the specification), nor where the render pass describes dependencies on its attachments - it describes the attachments, but not the dependencies (at least not explicitly). I have read the relevant chapters of the specification multiple times, but I feel like the language is not clear enough for a beginner to fully grasp.

I would also really appreciate Vulkan specification quotes where possible.

Edit: to clarify, the final question is: What synchronization mechanism guarantees that the draw call in the next command buffer is not submitted until the current draw call is finished?

graphics 3d vulkan

Share

edited Jun 15, 2020 at 14:10

Improve this question

Follow

asked Jun 14, 2020 at 10:29

cluntraru

133 ● 1 ● 8

1 This is not part of the question, but I'd also appreciate pointers on articles or preferably books on this sort of GPU behavior that's relevant to programming. I've ordered both Learning Vulkan by Parminder Singh and Vulkan Programming Guide by Graham Sellers, but they haven't arrived yet and don't seem to include too much on GPU hardware anyway (I might be wrong though). I unfortunately don't like the format of the Vulkan Cookbook by Pawel Lapinski that I read is one of the better options - I much prefer properly grasping the theory and doing things myself than following a "recipe" – cluntraru Jun 14, 2020 at 10:30

Does this answer your question? Synchronization between drawcalls in Vulkan – krOoze Jun 14, 2020 at 11:13

1 Not exactly. It explains that draws in the same subpass are executed as if in order, but what about different subpasses (or almost identical command buffers submitted twice in a row in this case)? It mentions that those are governed by external dependencies, but the only external subpass dependency here would be the one used to synchronize with the imageAcquired semaphore that has no srcAccessMask and, as a result, should not actually wait on any color attachment stages since there is no resource it needs to access (correct me if I'm wrong). – cluntraru Jun 14, 2020 at 11:27

1 Ran out of characters... in this case, wouldn't the following drawFrame() call run with no synchronization between the 2 and cause something like: render 1 starts, then render 2 starts, then render 1 finishes, then present 1 starts, then

render 2 finishes etc.? In this case, the depth buffer would be reused and presumably broken. – cluntraru Jun 14, 2020 at 11:30

1 @cluntraru: To answer your question would require analyzing the dependency graph of the tutorial code in question, which is a pretty big thing to ask. For all I know, there are explicit events or semaphore waits that prevent overlap between frames. I don't know if that's true, but I also don't know it isn't true. And the only way to find out would be to read the entire code of the tutorial. – Nicol Bolas Jun 14, 2020 at 14:00

4 Answers

Sorted by: Highest score (default)

16

I'm afraid, I have to say that the Vulkan Tutorial is wrong. In its current state, it can not be guaranteed that there are no memory hazards when using only one single depth buffer. However, it would require only a very small change so that only one depth buffer would be sufficient.

Let's analyze the relevant steps of the code that are performed within `drawFrame`.

We have two different queues: `presentQueue` and `graphicsQueue`, and `MAX_FRAMES_IN_FLIGHT` concurrent frames. I refer to the "in flight index" with `cf` (which stands for `currentFrame = (currentFrame + 1) % MAX_FRAMES_IN_FLIGHT`). I am using `sem1` and `sem2` to represent the different arrays of semaphores and `fence` for the array of fences.

The relevant steps in pseudocode are the following:

```
vkWaitForFences(..., fence[cf], ...);
vkAcquireNextImageKHR(..., /* signal when done: */
sem1[cf], ...);
vkResetFences(..., fence[cf]);
vkQueueSubmit(graphicsQueue, ...
    /* wait for: */ sem1[cf], /* wait stage: *,
COLOR_ATTACHMENT_OUTPUT ...
    vkCmdBeginRenderPass(cb[cf], ...);
      Subpass Dependency between EXTERNAL -> 0:
          srcStages = COLOR_ATTACHMENT_OUTPUT,
          srcAccess = 0,
          dstStages = COLOR_ATTACHMENT_OUTPUT,
          dstAccess = COLOR_ATTACHMENT_WRITE
      ...
      vkCmdDrawIndexed(cb[cf], ...);
      (Implicit!) Subpass Dependency between 0 ->
EXTERNAL:
          srcStages = ALL_COMMANDS,
          srcAccess =
COLOR_ATTACHMENT_WRITE|DEPTH_STENCIL_WRITE,
          dstStages = BOTTOM_OF_PIPE,
          dstAccess = 0
    vkCmdEndRenderPass(cb[cf]);
    /* signal when done: */ sem2[cf], ...
    /* signal when done: */ fence[cf]
);
vkQueuePresent(presentQueue, ... /* wait for: */
sem2[cf], ...);
```

The draw calls are performed on one single queue: the `graphicsQueue`. We must check if commands on that `graphicsQueue` could theoretically overlap.

Let us consider the events that are happening on the `graphicsQueue` in chronological order for the first two frames:

```
img[0] -> sem1[0] signal -> t|...|ef|fs|lf|co|b ->
sem2[0] signal, fence[0] signal
```

```
img[1] -> sem1[1] signal -> t|...|ef|fs|lf|co|b ->
sem2[1] signal, fence[1] signal
```

where `t|...|ef|fs|lf|co|b` stands for the different pipeline stages, a draw call passes through:

- `t` ... `TOP_OF_PIPE`
- `ef` ... `EARLY_FRAGMENT_TESTS`
- `fs` ... `FRAGMENT_SHADER`
- `lf` ... `LATE_FRAGMENT_TESTS`
- `co` ... `COLOR_ATTACHMENT_OUTPUT`
- `b` ... `BOTTOM_OF_PIPE`

While there **might** be an implicit dependency between `sem2[i] signal -> present` and `sem1[i+1]`, this only applies when the swap chain provides only one image (or if it would always provide the same image). In the general case, this can not be assumed. That means, there is nothing which would delay the **immediate** progression of the subsequent frame after the first frame is handed over to `present`. The fences also do not help because after `fence[i] signal`, the code waits on `fence[i+1]`, i.e. that also does not prevent progression of subsequent frames in the general case.

What I mean by all of that: The second frame starts rendering **concurrently** to the first frame and there is nothing that prevents it from accessing the depth buffer concurrently as far as I can tell.

The Fix:

If we wanted to use only a single depth buffer, though, we can fix the tutorial's code: What we want to achieve is that the `ef` and `lf` stages wait for the previous draw call to complete before resuming. I.e. we want to create the following scenario:

```
img[0] -> sem1[0] signal -> t|...|ef|fs|lf|co|b ->
sem2[0] signal, fence[0] signal
img[1] -> sem1[1] signal ->
t|...|________|ef|fs|lf|co|b -> sem2[1] signal,
fence[1] signal
```

where `_` indicates a wait operation.

In order to achieve this, we would have to add a barrier that prevents subsequent frames performing the `EARLY_FRAGMENT_TEST` and `LATE_FRAGMENT_TEST` stages at the same time. There is only one queue where the draw calls are performed, so only the commands in the `graphicsQueue` require a barrier. The "barrier" can be established by using the subpass dependencies:

```
vkWaitForFences(..., fence[cf], ...);
vkAcquireNextImageKHR(..., /* signal when done: */
sem1[cf], ...);
vkResetFences(..., fence[cf]);
vkQueueSubmit(graphicsQueue, ...
    /* wait for: */ sem1[cf], /* wait stage: *,
EARLY_FRAGMENT_TEST...
    vkCmdBeginRenderPass(cb[cf], ...);
      Subpass Dependency between EXTERNAL -> 0:
          srcStages =
EARLY_FRAGMENT_TEST|LATE_FRAGMENT_TEST,
```

```
          srcAccess =
DEPTH_STENCIL_ATTACHMENT_WRITE,
          dstStages =
EARLY_FRAGMENT_TEST|LATE_FRAGMENT_TEST,
          dstAccess =
DEPTH_STENCIL_ATTACHMENT_WRITE|DEPTH_STENCIL_ATTACHM
      ...
      vkCmdDrawIndexed(cb[cf], ...);
      (Implicit!) Subpass Dependency between 0 ->
EXTERNAL:
          srcStages = ALL_COMMANDS,
          srcAccess =
COLOR_ATTACHMENT_WRITE|DEPTH_STENCIL_WRITE,
          dstStages = BOTTOM_OF_PIPE,
          dstAccess = 0
    vkCmdEndRenderPass(cb[cf]);
    /* signal when done: */ sem2[cf], ...
    /* signal when done: */ fence[cf]
);
vkQueuePresent(presentQueue, ... /* wait for: */
sem2[cf], ...);
```

This should establish a proper barrier on the `graphicsQueue` between the draw calls of the different frames. Because it is an `EXTERNAL -> 0`-type subpass dependency, we can be sure that renderpass-external commands are synchronized (i.e. sync with the previous frame).

Update: Also the wait stage for `sem1[cf]` has to be changed from `COLOR_ATTACHMENT_OUTPUT` to `EARLY_FRAGMENT_TEST`. This is because layout transitions happen at `vkCmdBeginRenderPass` time: after the first synchronization scope (`srcStages` and `srcAccess`) and before the second synchronization scope (`dstStages` and `dstAccess`). Therefore, the swapchain image must be

available there already so that the layout transition happens at the right point in time.

Share Improve this answer Follow

edited Jun 16, 2020 at 9:58

answered Jun 15, 2020 at 22:55

j00hi

5,910 ● 3 ● 52 ● 86

Thank you for the complete answer! I think it might be worth clarifying that the old subpass dependency is not deleted (still needed for semaphore sync), and that there are now 2 from EXTERNAL to 0, just to avoid potential confusion for anyone else reading this. Also, shouldn't dstAccess in the fix also include DEPTH_STENCIL_ATTACHMENT_READ, not just WRITE? – cluntraru Jun 16, 2020 at 6:35

Regarding the "old subpass dependency": Do you mean that `COLOR_ATTACHMENT_OUTPUT -> COLOR_ATTACHMENT_OUTPUT` would still be necessary? The batch's `COLOR_ATTACHMENT_OUTPUT` stages must wait for `sem1[cf]` to signal anyways. I do not think that such a dependency must be included in the subpass dependencies, unless I am overlooking something. – j00hi Jun 16, 2020 at 7:55

Regarding the two subpass dependencies: Are you referring to the second "`(Implicit!)`" subpass dependency? – j00hi Jun 16, 2020 at 7:55

Regarding the `dstAccess`: Yes, you are right. A depth test always performs read and write access. The read access must see the cleared depth buffer values already. Synchronizing with `DEPTH_STENCIL_ATTACHMENT_WRITE` only is not sufficient. I'll update it to

`DEPTH_STENCIL_ATTACHMENT_WRITE|DEPTH_STENCIL_ATTACHMENT_READ`. Thanks for catching that! – j00hi Jun 16, 2020 at 7:57

1 @DanielMarques of course that would be an option---maybe even a good one since it theoretically allows to parallelize more. It's just that the question was about using a single depth buffer, which could be advantageous in extremely memory-limited settings, for example. – j00hi Sep 28, 2021 at 7:16

4

No, rasterization order does not (per specification) extend outside a single subpass. If multiple subpasses write to the same depth buffer, then there should be a `VkSubpassDependency` between them. If something outside a render pass writes to the depth buffer, then there should also be explicit synchronization (via barriers, semaphores, or fences).

FWIW I think the vulkan-tutorial sample is non-conformant. At least I do not see anything that would prevent a memory hazard on the depth buffer. It seems that the depth buffer should be duplicated to `MAX_FRAMES_IN_FLIGHT`, or explicitly synchronized.

The sneaky part about undefined behavior is that wrong code often works correctly. Unfortunately making sync proofs in the validation layers is little bit tricky, so for now only thing that remains is to simply be careful.

Futureproofing the answer:
What I do see is conventional WSI semaphore chain

(used with `vkAnquireNextImageKHR` and `vkQueuePresentKHR`) with `imageAvailable` and `renderFinished` semaphores. There is only one subpass dependency with `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT`, that is chained to the `imageAvailable` semaphore. Then there are fences with `MAX_FRAMES_IN_FLIGHT == 2`, and fences guarding the individual swapchain images. Meaning two subsequent frames should run unimpeded wrt each other (except in the rare case they acquire the same swapchain image). So, the depth buffer seems to be unprotected between two frames.

Share Improve this answer Follow

edited Jun 15, 2020 at 22:45

answered Jun 15, 2020 at 17:15

krOoze

13.2k ● 1 ● 22 ● 35

This what I suspected as well, leading to my posting the question. Thank you for putting in the time to look through the code, I know it was a lot to ask. – cluntraru Jun 15, 2020 at 19:55

If Vulkan is so complex even experts disagree on simple things (like writing to a depth buffer correctly) then that's an indication Vulkan just needs a do over. The fact incorrect code often still runs (at least on most platforms) makes the situation even worse. – JustinBlaber Jun 28 at 13:40

2

Yes, I also spent some time trying to figure out what was meant by the statement "We only need a single depth image, because only one draw operation is running at once."

That didn't make sense to me for a triple buffered rendering setup where work is submitted to the queues until MAX_FRAMES_IN_FLIGHT is reached - there's no guarantee that all three aren't running at once!

Whilst the single depth image worked OK, triplicating everything so each frame uses a fully independent set of resources (blocks and all) would seem to be the safest design and yielded identical performance under test.

Share Improve this answer Follow

answered May 6, 2022 at 23:32

MarvinWS
41 • 5

How the "triplicating everything ... " design fares in terms of memory, in a more complex rendering scenario: multipass rendering for cascade shadow for example where you have 4 layered depth textures ... is that feasable? – Avi Nov 25, 2023 at 10:22

Excellent question!! Since posting, my render engine has evolved a bit - I have linked list based OIT transparency up and running now which needs around 256MB to handle three to four layers of full-screen transparency. I'm certainly not triplicating that... At the moment with full 3D PBR and OIT transparency workflows as well as 2D ortho, I'm running around 665 MB of image buffers. What I'm thinking though is to pull it back to double buffering in terms of frame generation, but maintaining a triple buffered presentation.

Haven't done cascaded shadow mapping yet, they're next. :)
– MarvinWS Nov 26, 2023 at 11:29

0

I believe the tutorial's code is correct and needs only a single depth buffer to function, for the following reasons:

- Command buffer submissions to a single queue respect submission order. While drivers are free to reorder commands in the absence of synchronization mechanisms such as semaphores and fences, submission order becomes significant once such mechanisms are adopted:

  > "Command buffer submissions to a single queue respect submission order and other implicit ordering guarantees, but otherwise may overlap or execute out of order. Other types of batches and queue submissions against a single queue (e.g. sparse memory binding) have no implicit ordering constraints with any other queue submission or batch. Additional explicit ordering constraints between queue submissions and individual batches can be expressed with semaphores and fences."

- Commands split across different command buffers and submitted to the queue in the order they were recorded will execute as if those commands were recorded to a single command buffer:

> "Command buffer boundaries, both between primary command buffers of the same or different batches or submissions as well as between primary and secondary command buffers, do not introduce any additional ordering constraints. In other words, submitting the set of command buffers (which can include executing secondary command buffers) between any semaphore or fence operations execute the recorded commands as if they had all been recorded into a single primary command buffer, except that the current state is reset on each boundary."

- A wait operation defined by a semaphore passed to `vkQueueSubmit()` will block execution of all commands appearing later in submission order, including commands recorded to other command buffers:

  > "The second synchronization scope includes every command submitted in the same batch. In the case of vkQueueSubmit, the second synchronization scope is limited to operations on the pipeline stages determined by the destination stage mask specified by the corresponding element of pWaitDstStageMask. Also, in the case of vkQueueSubmit, the second synchronization

> scope additionally includes all commands that occur later in submission order."

- Execution dependencies for a render pass help ensure that commands which certain other commands depend on will finish before the dependent commands begin execution. The render pass in the Vulkan Tutorial establishes such a happens-before constraint on commands submitted prior to the `vkCmdBeginRenderPass()` which the program calls every frame:

  > "If srcSubpass is equal to VK_SUBPASS_EXTERNAL, the first synchronization scope includes commands that occur earlier in submission order than the vkCmdBeginRenderPass used to begin the render pass instance. Otherwise, the first set of commands includes all commands submitted as part of the subpass instance identified by srcSubpass and any load, store, or multisample resolve operations on attachments used in srcSubpass. In either case, the first synchronization scope is limited to operations on the pipeline stages determined by the source stage mask specified by srcStageMask."

Imagine the following sequence of commands has been submitted to the queue:

```
// frame 1
vkAcquireNextImageKHR(); // signals
imageAvailableSemaphores[0].
RecordRenderPass();
vkQueueSubmit(); // waits on
imageAvailableSemaphores[0] at
COLOR_ATTACHMENT_OUTPUT stage.
vkQueuePresentKHR();

// frame 2
vkAcquireNextImageKHR(); // signals
imageAvailableSemaphores[1].
RecordRenderPass();
vkQueueSubmit(); // waits on
imageAvailableSemaphores[1] at
COLOR_ATTACHMENT_OUTPUT stage.
vkQueuePresentKHR();

// frame 3
vkAcquireNextImageKHR(); // signals
imageAvailableSemaphores[0].
RecordRenderPass();
vkQueueSubmit(); // waits on
imageAvailableSemaphores[0] at
COLOR_ATTACHMENT_OUTPUT stage.
vkQueuePresentKHR();
```

In addition to the semaphores referenced above there is also a fence between each frame to ensure that command buffers aren't modified while in their *pending* state, but these fences are not responsible for ensuring that the depth buffer isn't accessed simultaneously by two different frames so I'm ignoring them for the purposes of this analysis.

The first `vkQueueSubmit()` command will cause the driver to wait for `imageAvailableSemaphores[0]` to be signaled before moving on to the `COLOR_ATTACHMENT_OUTPUT` stage.

Because the wait operation blocks execution according to submission order, it will also prevent execution of the drawing and presentation commands in frames 2 and 3 until `imageAvailableSemaphores[0]` has been signaled in the first frame. The execution dependency on the render pass further prevents draw commands for subsequent passes from executing before the first pass has reached the `COLOR_ATTACHMENT_OUTPUT` stage.

Frames are therefore guaranteed to be executed in order (frame 2 won't be rendered before frame 1), and each frame will have finished writing to the depth buffer (in the `LATE_FRAGMENT_TESTS` stage preceding the `COLOR_ATTACHMENT_OUTPUT` stage) before subsequent frames can write to the depth buffer.

edited Apr 26 at 17:51

answered Dec 22, 2023 at 21:58

Adrian Lopez
1,753 1 17 23