20180203 - Optimized Swapchain in Vulkan II

Continued from 20180202 - Optimized Swapchain in Vulkan

Prior post covered best known practices, this looks ahead to pushing the limit of what is possible.

Summary on Theory of Latency Optimization
Very rough latency estimations for FIFO_RELAXED, CPU input to first line scanout.

> 2.0-3.0 frames of latency - non-late-latch 2-deep swapchain
> 1.0-2.0 frames of latency - beginning frame late-latch, almost 100% rendering pre-acquire (implement late-latch)
> 1.0-1.5 frames of latency - beginning frame late-latch, 50% rendering pre-acquire (increase stability in cost/frame)
> 0.5-1.0 frames of latency - middle of frame late-latch, 50% rendering pre-acquire (increase view-independent work)
> 0.0-0.5 frames of latency - middle of frame late-latch, 0% rendering pre-acquire (frame adapts on GPU to hit v-sync)
> 0.0-0.1 frames of latency - ending of frame late-latch, 0% rendering pre-acquire (max view-independent work)

Typically these steps require designing the engine around the concept from the beginning, including the first step of late-latch (implies GPU-side culling without CPU-readback). Also non-CRT display latency quickly places small improvements into the realm of diminishing returns. For instance if LCD has effectively 16 ms of latency, going above and beyond to reduce app-side latency by 2 ms might not even be perceptable.

In the context of these changes, the possible improvements for front-buffer rendering over FIFO_RELAXED are marginal. At the point where the majority of frame is view-independent work, there is very little to overlap with scanout.

Jumping to the End Game

If following the recommendations in the prior posting from 20180302, getting to 0% rendering pre-acquire implies waiting on the acquire semaphore before starting GPU work for a frame. This is not going to be good for GPU utilization, so another method will be required (getting rid of that semaphore). Likewise moving to frame-adaptive rendering strategy requires estimating where the next frame scanout starts. Then, on the GPU, using indirect dispatches to adjust the amount of view-independent work. This is ultimately to push the view-dependent work as close as possible to the start of scanout. More on all this later ...

Moving to almost no view-dependent work is a mighty challenge. The intermediate ground that I like is to do non-triangle rendering into a octahedron, then do a late-frame fisheye projection into the swapchain only applying the camera rotation transformation at the end of the frame. End-of-frame late-latch used for low latency camera direction. Mid-frame late-latch used for camera position and "button" related actions. Clearly not practical for normal games.

Value or Lack of Value from MAILBOX?
MAILBOX can never provide good animation/motion quality. There is always going to be judder as frames are rendered for a different moment of time then they are presented for, and this time difference is highly variable. In terms of visual quality, rendering faster than v-sync and using MAILBOX is always worse quality than rendering slower and steadly hitting v-sync.

MAILBOX mostly serves as a hack to enable the bad practice of latching gamepad/mouse input on the CPU before rendering the frame, to have better input latency than would be possible if the title limited to display refresh rendering interval. At the point where an engine is "doing it right" and late-latching input, latency ends up even lower, and there is no value in MAILBOX.

One might also attempt to justify MAILBOX under the claims that the differences in input frequency and display frequency cause jitter, and thus motion quality will always be poor (since camera motion is a function of user input). This claim is false, in that non-player-input-controlled motion is still important, and that it is possible to resample (interpolate) input to the display frequency.

As for cases where the engine renders slower than v-sync, it is much better IMO to automatically adapt frame cost such that the title can maintain v-sync. There are a lot of possibilities here, too many for this post.

One of the fails of modern computer engineering is spending a lot of time making a poor solution run a little bit better, instead of just adopting a good solution. MAILBOX is a poor solution, so I don't bother with it.

This logic extends to anything "windowed". While in the prior post I recommended a 3-deep swapchain and MAILBOX/IMMEDIATE for "windowed" cases because no one ever late-latches, I don't personally ever use this. Instead I drop back to 2-deep swapchain which acts like FIFO regardless of selected display mode.

Killing off the Acquire Semaphore?
Warning, this section is just theory, I'm still working out the design before I try and build it. The quest is to find an even better method for dynamically adaptive workloads while minimizing input latency. With a side goal to enable a replay of a

single command buffer per frame to saturate the GPU. Think of this as a minimal overhead "shadertoy" which can adapt to hit frame rate, with aim to do something well beyond a toy.

A possible maybe-illegal workaround to re-enable GPU saturation with a single command buffer per frame might be to skip connecting the acquire semaphore. This removes all the back-pressure which limits GPU work to display refresh rate. Meaning new frame always starts when last frame ends. So this would depend on the GPU work adjusting itself to fill out the time until scanout, and possibly a CPU-side protection to avoid submiting frames at an average which is faster than v-sync. The challenge is in estimating scanout start. The semaphore connecting the command buffer to present still exists, which means it is possible to miss a refresh period. That might provide all that is required.

Acquire supports a 'fence', which is ironically the worst design ever under common usage, but might provide a way to detect missed frames without disrupting GPU work. The key is to make waiting on the 'fence' never block unless a prior frame misses. Miss detection can be provided by checking wall-clock time frame difference after the wait on fence returns. If the time diff roughly doubles, then there was a miss. The end-of-command-buffer bottom-of-pipe time-stamp associated with this miss, provides a time which is known to pass into scanout. So then backoff the estimated start-of-scanout based on that data.

Starting with a simple design below. If GPU work is too small for frame, this guarantees overwrite of active scanout (which is bad). But at least this setup establishes something which limits peak submit rate to refresh rate.

```
overwriting active scanout region -->|    |<-          actual scanout for labeled frame
                                     |    |               |
                [__scanoutA__][__scanoutB__][__scanoutA__][__scanoutB__]
       [gpuA] [gpuB] [gpuA]          [gpuB]        [gpuA]
 [cpuA]a[cpuB]b[cpuA]         a[cpuB]       b[cpuA]
                               |            |
  send GPU B work before acquire fence B    |
                                            |
                           fence on acquire B, then present B
```

As GPU work gets larger, problem goes away if write to swapchain is only at end of GPU work (which it would be).

```
write to swapchain here is not a problem -->| |<-       actual scanout for labeled frame
                                            | |           |
                [__scanoutA__][__scanoutB__][__scanoutA__][__scanoutB__]
       [__gpuA__][__gpuB__][__gpuA__][__gpuB__]    [__gpuA__]
 [cpuA]a[cpuB]b[cpuA]         a[cpuB]       b[cpuA]
                               |            |
  send GPU B work before acquire fence B    |
                                            |
                           fence on acquire B, then present B
```

If CPU/GPU load expands to entire frame, everything works out fine.

```
                                          labeled scanout
                                            |
  [__scanoutA__][__scanoutB__][__scanoutA__][__scanoutB__]
                              [____gpuB____]
                a[____cpuB___]           | |
                                        >| |<-- no end of frame write conflict
```

So this establishes a way to start with a small workload and adaptively increase to fill the frame. Now it just needs a mechanism to adapt on a miss.

```
                                            |  rescan A  |
  [__scanoutA__][__scanoutB__][__scanoutA__][____miss____][__scanoutB__][__scanoutA__][__scanoutB__]
                              [_____gpuB____][_____gpuA____]            [____gpuB____]
                a[____cpuB___]b[____cpuA___]              a[____cpuB___]
                                                          |
           miss detected a frame late, start to adjust -->|
```

In the cases I'm interested in, there is really no CPU load (represented in the next drawing). Next drawing displays the theory for adaptively finding scanout start and backing off.

```
                                            |  rescan A  |
  [__scanoutA__][__scanoutB__][__scanoutA__][____miss____][__scanoutB__][__scanoutA__][__scanoutB__]
               [_____gpuA____][_____gpuB____][_____gpuA____]  [_________gpuB________][____gpuA____]
                a[B]          b[A]          |             a[B]          b[A]          |
                                            |             |                           |
 pushed out estimated scanout start causing miss          |                           |
                                                          |                           |
                                                                      miss is noticed          bring back start estimation to safe place
                                                                                               note frame after miss is free to run long
```

Estimated command buffer work-flow.

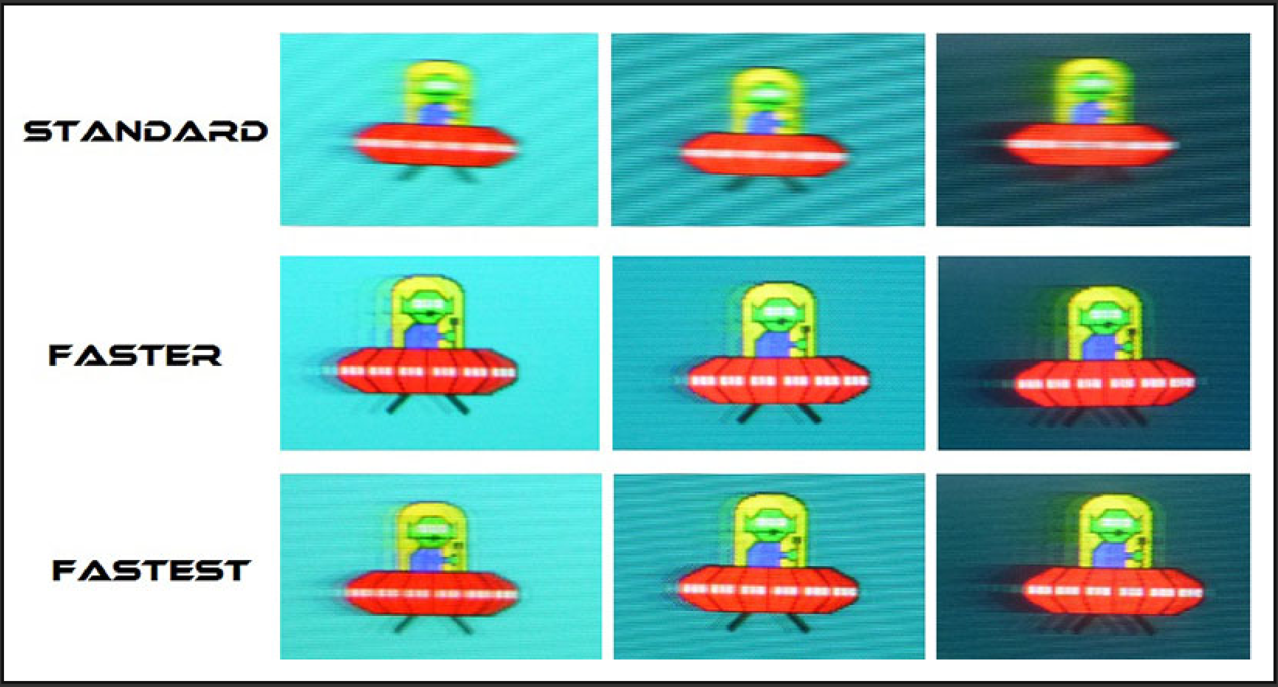
```
Bottom-of-pipe timestamp at end of prior command buffer
Start of next command buffer (next frame)
Copy timestamp to gpu buffer (API fail, the timestamp is already physicaly in a buffer, just give me buffer access)
Launch indirect generation kernel
 - Spin waiting for the timestamp write (don't want to block later kernel dispatch)
 - Read HOST data on if a miss happened
 - Compute new estimated start-of-scanout position (using timestamp datas from prior frame)
 - Compute amount of work to get close to that position (writes "volatile" to indirect arguments, avoids barrier later)
Set event Z
Launch independent work wrapped in bottom-of-pipe timestamps (so cost is known)
Wait event Z (won't actually wait due to independent work)
Launch indirect work wrapped in bottom-of-pipe timestamps (so next frame gets new time/workitem)
Etc
```

Enough of a plan to start building ...

Obsession With Frame-Rate
Thoughts on Blur Busters 240/480/1000 Hz And Beyond.

There are two aspects to the importance of frame rate: input latency and motion blur reduction. Using massive frame rates to attempt to solve "scan-and-hold" induced motion blur is just silly. CRT at 80 Hz already solved this. Jacking up frame rate to 1000 Hz in an attempt make an LCD as good as a CRT is fail. That is a 12x cost in rendering. Strobe backlight or black frame insertion is a better solution to induced motion blur than ultra-high framerate.

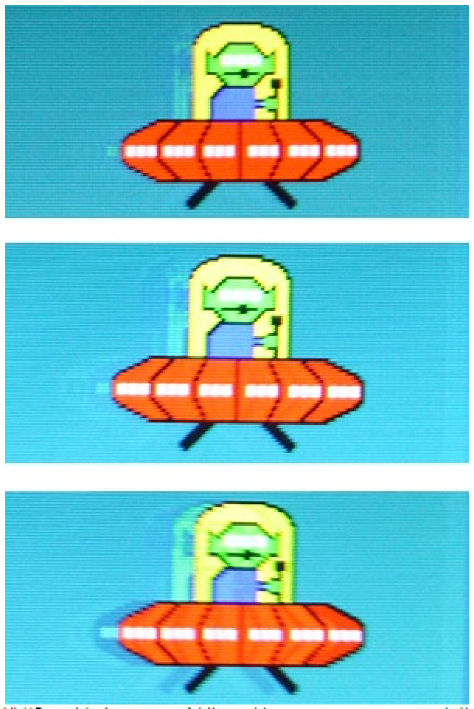
For example, below are some pursuit camera results on this Samsung 144 Hz display review on TFT Central. The "standard" is no strobed backlight, "fastest" is the strobe time fully minimized, all at 144 Hz. Notice even at 144 Hz this display ghosts as it cannot physical switch pixels fast enough.



As for input latency, scanout time sets limits on input latency, assuming the display can change physical pixel value faster than scanout supports. Given the option of switching to optimal swap and late-latch, vs doing it the classic non-late-latch method combined with rendering twice as fast, it is clearly better to just late-latch, no contest really.

As for pixel switching time: CRT, Plasma, and LED (as in indoor/outdoor signs with giant pixels), have fast switching time. LCDs don't switch very fast, and I can only suspect OLED does not either, because both OLED and Plasma and LED all are physically 1-bit/channel pixels modulated PWM to my knowledge, and OLED for the most part has no perceptual dithering (unlike Plasma for instance).

For LCD, results from this Asus 240 Hz display on TFTCentral show diminishing returns. See strobed backlight shot below. The display does 240 Hz with scan-and-hold, but peaks at 144 Hz for strobed backlight. And with the strobed backlight, one can see variable amount of the prior frame. Suggests that the display can really only switch pixels at around 72 Hz (~13 ms) but given that would be before flicker fusion (given the high display brightness), higher frame rates are necessary. *I'd likely try to run a frame rate lower than 144 Hz to limit the ghosting, perhaps 90 Hz. And at this stage, for non-touch-the-screen-cases, I'm not sure if there is much value (after late-latch is implemented) to increase frame rate just to reduce input latency beyond the physical limits of pixel switching time.*


ULMB enabled, upper, middle and lower screen area cross-talk