Vulkanised 2026
The 8th Vulkan Developer Conference
San Diego, USA | February 9-11, 2026

Smooth, Stutter-Free Animation with Vulkan

Tom Murphy, Google
Ian Elliott, Google
Ady Abraham, Google



Goals of this Talk

The Primary Goal of this presentation is to:

Help you create Vulkan animations that appear smooth and stutter-free to everybody, even the most discerning viewers

Make your Animations look smooth and stutter-free to even the most discerning users

Secondary Goals:

- Discuss ways to minimize latency
- Get your feedback on some potential future techniques

Note: This is a short introduction to a large topic

Terminology

Visual Anomaly
An irregularity in an animation that deviates significantly from the expected or normal appearance, including the following:

- **Stutter, or “jank”**
  When motion appears jerky, inconsistent, or freezes momentarily; often caused by dropped frames, mismatched frame rates, inefficient rendering, or poor animation timing

- **Tearing**
  When using IMMEDIATE mode, the display shows parts of 2 frames

- **Latency** (most frequently “Input Latency”)
  The time delay between user input and the corresponding visual change appearing on the display

An example of tearing

Both animations use the exact same 11 frames of Eadweard Muybridge's "The Horse in Motion" (1878).

Consistent Pacing

100ms per frame (Fixed)

Bad Pacing

158ms delay

Some business impact of stuttering games

Ruins Game Experience

“Stuttery gameplay breaks the illusion and makes the game less enjoyable”

-From Mark Dochtermann, former CTO of First Person Shooters EA and Android Games Eng Lead

Reputational Damage and Poor Reviews

Player frustration leads to negative reviews and reduced ratings

Reduced Sales and Revenue

Negative reviews lead to uninstalls and fewer installs/purchases

- If a game stutters during initial gameplay, *day-1 retention can drop by 30-40%!*

Principles for Smooth, Stutter-Free Animation

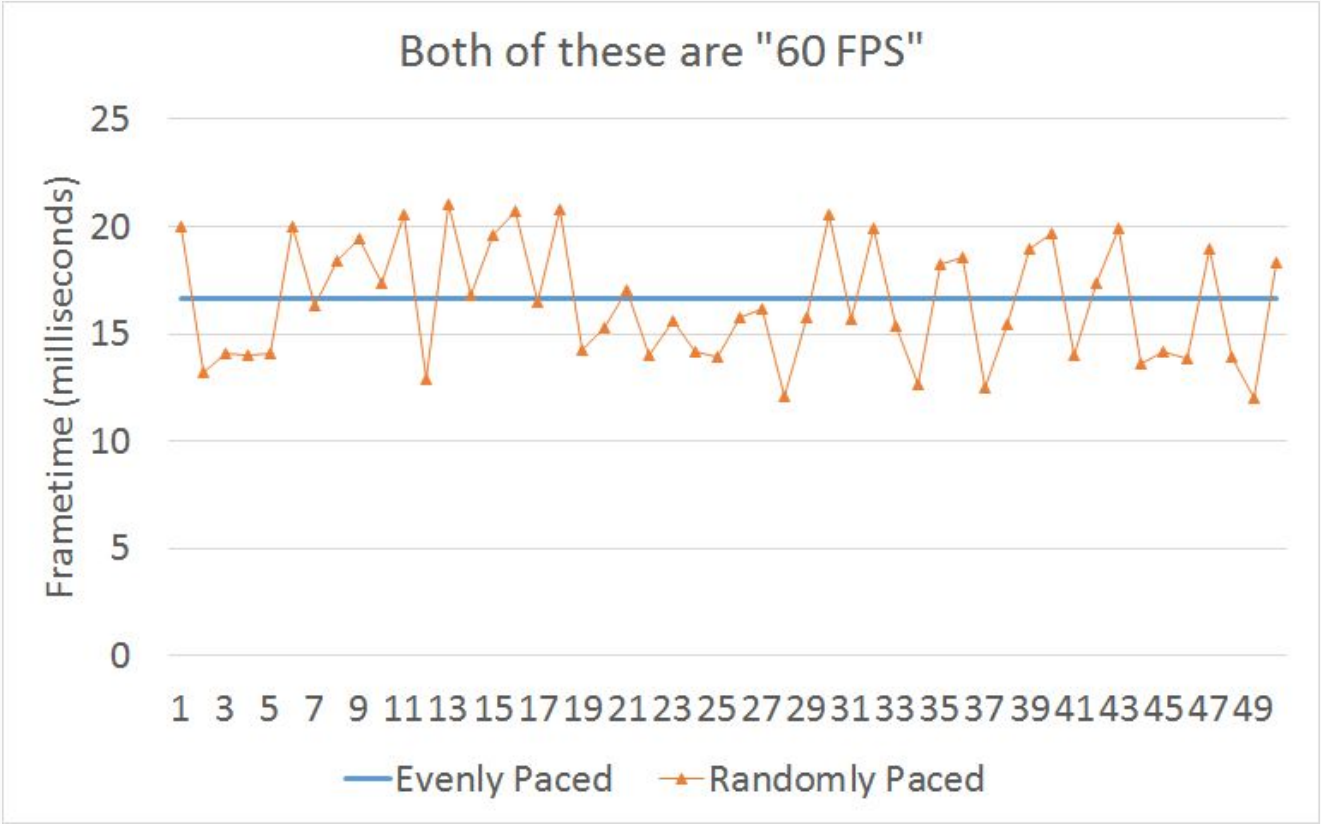
1. Successive frames must be displayed to the user at a consistent rate

2. Synchronize content time with presentation time

tl;dr:
you can't just "render and present, render and present, ..."

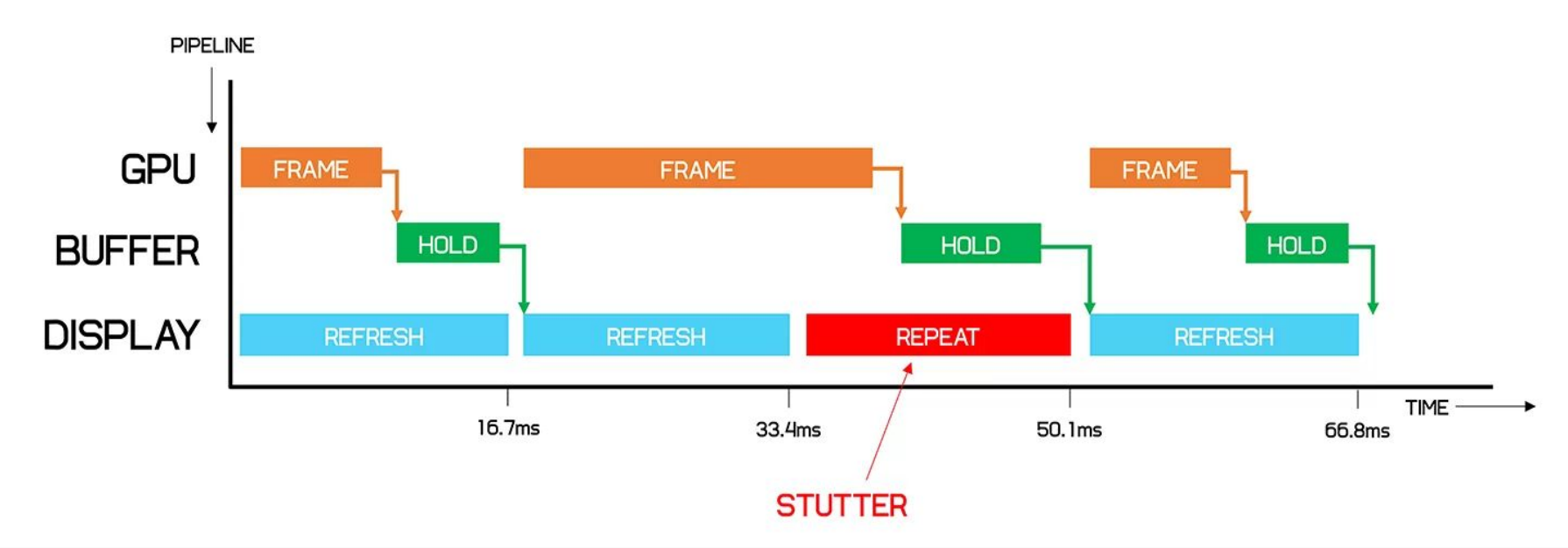
If you do, your frames will not render at a consistent rate, and discerning users will experience stutter

Stutter: FPS vs Frame Time



From https://www.reddit.com/r/pcmasterrace/comments/8mabxh/frame_pacing_why_one_games_60fps_looks_great_and/

Inconsistent Frame Times



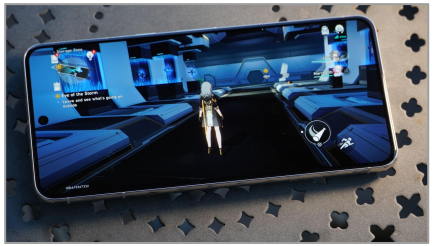
From: https://www.techspot.com/article/1668-how-many-fps-do-you-need/

(A quick side note)

Displays come in many flavors:

- Fixed Refresh Rate (FRR)
  - Traditional displays with one refresh rate
- Variable Refresh Rate (VRR)
  - Content can trigger a vsync within a long period of time
    - Typically limited to full-screen (can switch to FRR otherwise)
- Adaptive Refresh Rate (ARR)
  - LTPO panels in newer mobile devices
  - Can switch between many refresh rates without a mode switch

We'll primarily address FRR displays in this talk, but we'll also discuss VRR



Hands on: Improving stutter with VK_GOOGLE_display_timing

- A brief walkthrough of how to use it
- Source code is available!

Step-by-Step: Achieving smooth, stutter-free animation with an FRR display

Here's the Vulkan process steps (details to follow):

1. Query refreshDuration of the display
2. Pick a multiple of refreshDuration ⇒ desired FPS
3. Pick a starting desiredPresentTime and start rendering and presenting frames
4. Get feedback for past frames
5. Adjust desiredPresentTime based on feedback
6. Step and repeat (i.e. continue to monitor and adjust)

1) Query refreshDuration of the display

- Query the time between vsyncs (in nanoseconds)
- **`vkGetRefreshCycleDurationGOOGLE`** returns:
  - **`VkRefreshCycleDurationGOOGLE::refreshDuration`**

- Note: the refresh rate is 1/refreshDuration
  - For example: a 60 Hz display has a refreshDuration of 16,666,666,666 nsec
- See the table on the next slide for more examples

Relationship between refresh rate and refreshDuration

Refresh Rate	refreshDuration (msec)	refreshDuration (nsec)
144 FPS	6.64	6,644,444,444ns
120 FPS	8.33	8,333,333,333ns
90 FPS	11.11	11,111,111,111ns
60 FPS	16.67	16,666,666,666ns

Note: Android's refreshDuration value isn't exact

- Have observed 1+ msec deviations (e.g. 15.6 - 17.8 msec for 60 Hz)
- We'll show how to deal with this

2) Pick your target frame duration

- *Frame duration* determines *frame rate* (FPS)
- **Very important:**
  <u>frame duration should always be a multiple of refreshDuration!</u>
  - Not doing so will cause stutter
- Goal: a multiple that allows sufficient rendering time so that frames will NEVER be late for vsync
- This may require some guesswork at start-up time
  - Different frame rates may be needed for different HW
  - For example: 2X faster HW may yield smooth 2X FPS
- After a few frames, you can query to find out whether your guess was correct
- See examples for a 120 Hz display on the next slide
  - For example: a 1X multiplier yields 8.33 msec / frame)

Frame rates and times with different multipliers on a 120 Hz display

Multiplier	Frame Rate	Frame Time (msec)	Frame Time (nsec)
1X	120 FPS	8.33	8,333,333,333ns
2X	60 FPS	16.67	16,666,666,666ns
3X	40 FPS	25.00	25,000,000,000ns
4X	30 FPS	33.33	33,333,333,333ns
5X	24 FPS	41.67	41,666,666,666ns
6X	20 FPS	50.00	50,000,000,000ns

3) Pick a starting desiredPresentTime and start rendering and presenting frames

- The desiredPresentTime is given to vkQueuePresentKHR
  - It tells the presentation engine (PE) to not display the frame until that time
- This may require some guesswork, but here's a good heuristic:

```
frameDuration = refreshDuration * multiple;
desiredPresentTime = currentTime + frameDuration;
```

- For successive frames (i.e. frame n+1):

$$desiredPresentTime^{n+1} = desiredPresentTime^{n} + frameDuration;$$

- Note: Stuttering may happen for the first few frames
  - Step 5 eliminates any stuttering caused by a poor guess
  - If this early stuttering is a concern, you can do the following until step 5:
    - After rendering your frame content, render a large black rectangle on top of it
    - The feedback you get will take into account the render time for your content

4) Query for feedback about the past frames

- Query for statistics:
  - Call **vkGetPastPresentationTimingGOOGLE** to get an array of **VkPastPresentationTimingGOOGLE**, which contains:

```
typedef struct VkPastPresentationTimingGOOGLE {
    uint32_t    presentID;          // Application-provided ID from vkQueuePresentKHR
    uint64_t    desiredPresentTime; // Application-provided present-no-sooner time
    uint64_t    actualPresentTime;  // When the frame was actually presented
    uint64_t    earliestPresentTime;// Either actualPresentTime or an earlier vsync
    uint64_t    presentMargin;      // Time from rendering completion until PE starts
                                    // processing the frame
} VkPastPresentationTimingGOOGLE;
```

- Note: Android has a multi-frame latency (e.g. results for a 60 Hz display for frame n come at frame n+5)
  - The reason why there can be stuttering for the first few frames

5) Adjust desiredPresentTime–part 1

The feedback lets you zero in on smooth, stutter-free animation:

1. Do you need to adjust your frame rate?
    - If your frames are taking too long, increase the multiplier
    - If your frames are getting done way early, decrease the multiplier
    - Balance between competing goals:
        - Desire: Highest possible frame rate
        - Must: Never miss vsync
        - Guidance: Not too high of a frame rate

5) Adjust desiredPresentTime–part 2

2. Do you need to adjust `desiredPresentTime`?
    - Assuming a sustainable frame frame rate (part 1)
    - Adjust `desiredPresentTime` for that frame rate
    - Balance between competing goals:
        - Desire: Minimize latency by picking latest possible desiredPresentTime
        - Must: Never miss vsync & deal with refreshDuration inaccuracy
        - Guidance: Leave margin to deal with things like render time variability, clock drift, system background tasks, etc.

Heuristic: set `desiredPresentTime` in the middle between vsyncs:

$$\texttt{desiredPresentTime}^{n+1} = \texttt{actualPresentTime}^{n} + \texttt{frameDuration} + (\texttt{refreshDuration} / 2);$$

6) Continue to monitor and adjust

- Deal with any clock drift and inaccuracies in `refreshDuration`
  - Keep `desiredPresentTime` far enough away from vsync to avoid early/late frames
- New scenes may have different performance characteristics
  - Do you need to adjust frame rate higher or lower?
  - Do you need to adjust `desiredPresentTime` earlier or later?
  - Do you need to compile shaders or create pipelines?
    - Rather than stutter, you might hide the work:
      - Offline shader compilation
      - Shader/pipeline caching
      - Do the work in another thread
      - Do the work when displaying transitional content (e.g. a “cut scene”)

Dealing with stutter caused / revealed by…

Frames displayed at an inconsistent rate

Solved by using the steps in this presentation, with either:

- VK_GOOGLE_display_timing
- VK_EXT_present_timing

Shader compilation, pipeline creation, etc.

Try to hide this via things like:

- Offline shader compilation
- Shader/pipeline caching
- Do compilation/create in a separate thread

The hardware has changed!

Write your code to handle things like:

- Faster or slower CPU, GPU clock speeds
- Memory bandwidth
- Thermal throttling

Unrelated background processes

Can't completely defend against this, but you can leave margin in your `desiredPresentTime`

Variable Refresh Rate Displays (VRR)

What about variable refresh rate (VRR) displays?

- *VK_EXT_present_timing* adds API surface to support VRR

- VRR is more forgiving than FRR, but stutter is still possible
  - It's "forgiving" in the same way that stutter on a 144 Hz display is not as noticeable as on a 60 Hz display
  - The simplistic way VRR is used is to
    - Render and present, render and present, ...
    - No pacing!
  - Discerning users will notice variability in frame duration from things different render times, clock changes, system hiccups, etc.

- Plus: VRR is typically for full-screen use
  - When 2+ windows are visible, some presentation engine (PE) switch to FRR
    - Your software may need to switch between VRR and FRR on those platforms/devices

VRR displays (cont’d)

- First principles still apply!
  - Successive frames must be displayed to the user at a consistent rate
    - Not when rendering is complete
  - Frame movement must match frame generation intervals
    - Not earlier or later

- Your code should always:
  - Work in terms of `desiredPresentTime`
  - Leave some margin to deal with the unexpected
  - Monitor and adjust based on feedback

Minimizing Latency

Minimizing latency for interactivity

- **Latency** is the time delay between user input processing and the corresponding visual change appearing on the display
- You need to adjust when your input-render loop starts
  - Start too early ⇒ more latency
  - Start too late ⇒ miss vsync and get stutter
  - **Rule of Thumb:** Acquire and apply input as close to rendering as possible.
    - XR/VR - Exploits this via Asynchronous Reprojection to prevent motion sickness.
  - Balance between competing goals:
    - Desire: Minimize latency by picking latest possible desiredPresentTime
    - Must: Never miss vsync & deal with refreshDuration inaccuracy
    - Guidance: Leave margin to deal with things like render time variability, clock drift, system background tasks, etc.

Heuristic: set `desiredPresentTime` in the middle between vsyncs:

$$\texttt{desiredPresentTime}^{n+1} = \texttt{actualPresentTime}^{n} + \texttt{frameDuration} + (\texttt{refreshDuration} / 2);$$

Minimizing latency for interactivity

There are no Vulkan extensions that fully address latency

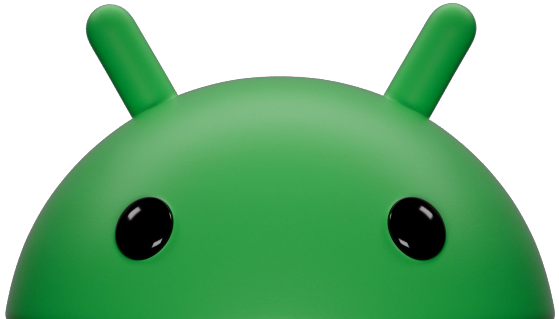
- The following can help you see your margin, but can't adjust when your input-render loop starts:
  - `VkPastPresentationTimingGOOGLE::actualPresentTime`
  - `VkPastPresentationTimingGOOGLE::earliestPresentTime`
  - `VkPastPresentationTimingGOOGLE::presentMargin`
- See the guidance in the Vulkan specification about VK_GOOGLE_display_timing

Must use platform APIs to adjust when your input-render loop starts

- For example: Android has Choreographer ...
  - A callback wakes up a thread when it's time to start rendering
  - You can look at how Android's Swappy library uses Choreographer

Thank you

Any Questions?



Vulkan Extensions

VK_GOOGLE_display_timing

The original extension to help achieve Smooth Animation

- Designed for fixed refresh rate (FRR) displays
- Widely available on Android (since 8.0 or Oreo)
- Started in 2016 and released in 2017
- Valve ported to some AMD Linux drivers
    - More?

VK_EXT_present_timing

Newer and more general (and complex) extension

- Expect it to start appearing in 2026
- Started in 2018 and released in late 2024
- Adds support for:
    - Variable refresh rate (VRR) displays
    - Additional OSs, clock domains, and window systems

Both are introduced in the “Present Timing Queries” spec section

Demos That Show Stutter

Two Balls with One Stuttering and One Smooth

https://pippy360.github.io/framepacing/index2.html

Digital Foundry's discussion of From Software's 30FPS stutter woes

https://www.youtube.com/watch?v=M7bWbWUKHmM

Eadweard Muybridge's "The Horse in Motion"

https://pippy360.github.io/framepacing/horse2.html

Links to Demo Source Code

- Unfortunately, at publication time we don't have:
  - A complete, up-to-date demo of VK_GOOGLE_display_timing
  - Any demo that uses VK_EXT_present_timing

- The following are still useful links:
  - The LunarG cube demo
    - Contains early use of VK_GOOGLE_display_timing
  - Swappy (Android frame pacing library)
    - Shows the use of Choreographer to address latency, but is not properly using VK_GOOGLE_display_timing

Some argue that stuttering is okay …

You'd never move like this …

So, don't let your animations move like this either!

Friends don't let friends have stuttering animations

(source)