Vulkanised 2026

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

Frames in Flight Demystified

Charles Giessen, LunarG

Who am I?

- Been working at LunarG since 2019
- Maintain several bits of the Ecosystem
  - loader, vulkaninfo, api dump, utility-libraries
- Wrote & maintain vk-bootstrap
- Vulkan Discord server active member & moderator
- Contributed to vulkan-tutorial.com's Frames In Flight chapter

Outline

- Pipelining overview
- What Frames in Flight means
- Implementation of Frames in Flight
- Memory usage patterns
- Practical considerations
- Swapchain

Target Audience

- Beginner & Intermediate developers
- Assumes completion of “Hello Triangle”
  - Can record and submit command buffers
  - Can upload data from host to device
- Anyone who hears frames in flight and thinks this:

LUNARG

Pipelining

LUNARG

What is Pipelining?

What is Pipelining?

- Series of data processing stages

LUNARG

What is Pipelining?

- Series of data processing stages

Data

LUNARG

What is Pipelining?

- Series of data processing stages

Data

LUNARG

What is Pipelining?

- Series of data processing stages

LUNARG

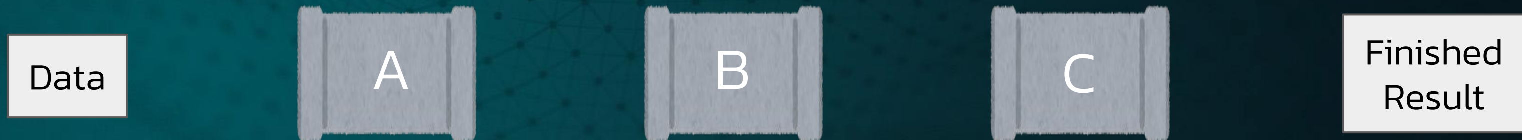
What is Pipelining?

- Series of data processing stages

LUNARG

What is Pipelining?

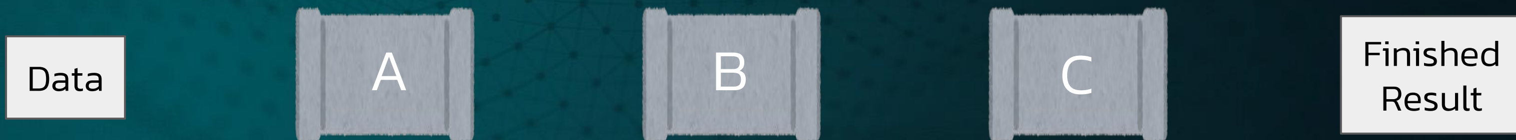
- Series of data processing stages



LUNARG

What is Pipelining?

- Series of data processing stages
- Output of one stage is the input of the next stage



LUNARG

What is Pipelining?

- Series of data processing stages
- Output of one stage is the input of the next stage

LUNARG

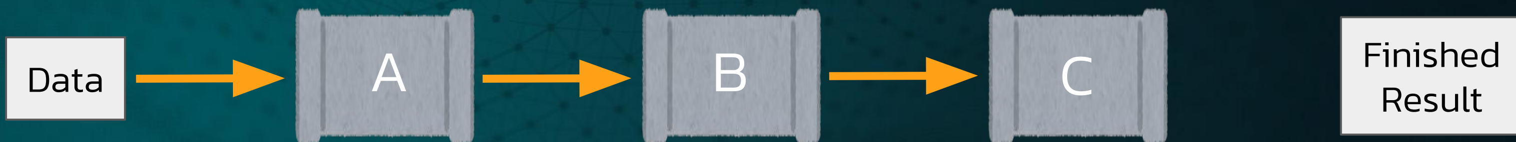
What is Pipelining?

- Series of data processing stages
- Output of one stage is the input of the next stage

LUNARG

What is Pipelining?

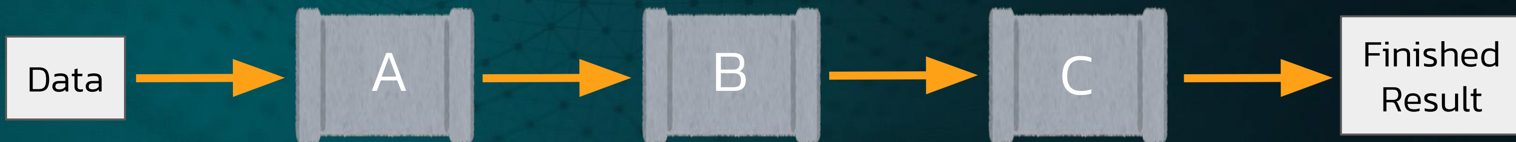
- Series of data processing stages
- Output of one stage is the input of the next stage



LUNARG

What is Pipelining?

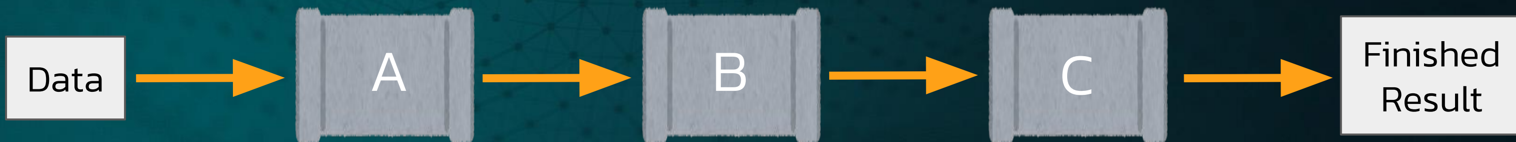
- Series of data processing stages
- Output of one stage is the input of the next stage



LUNARG

What is Pipelining?

- Series of data processing stages
- Output of one stage is the input of the next stage
- Each stage can be performed simultaneously



LUNARG

Video games are an example

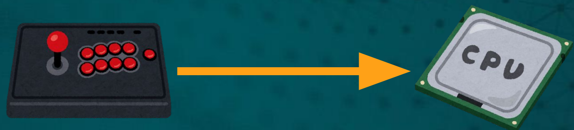
Video games are an example

- Gather input from mouse/keyboard/controller

LUNARG

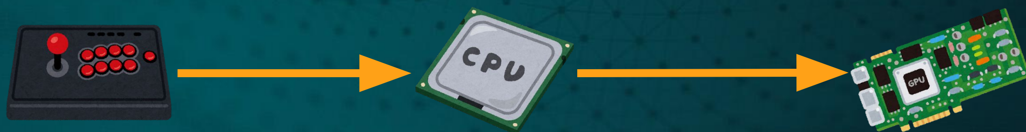
Video games are an example

- Gather input from mouse/keyboard/controller
- Gameplay/Physics/Simulation calculated on CPU



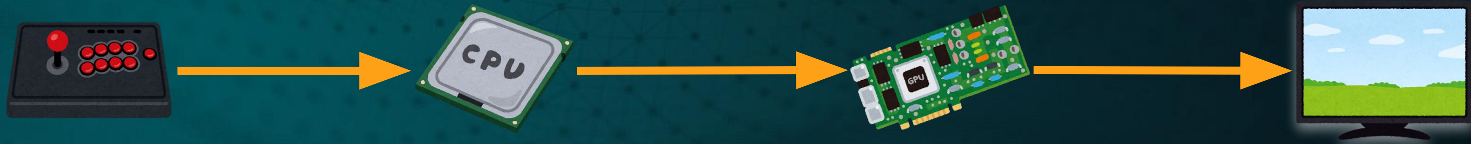
Video games are an example

- Gather input from mouse/keyboard/controller
- Gameplay/Physics/Simulation calculated on CPU
- Rendering on GPU

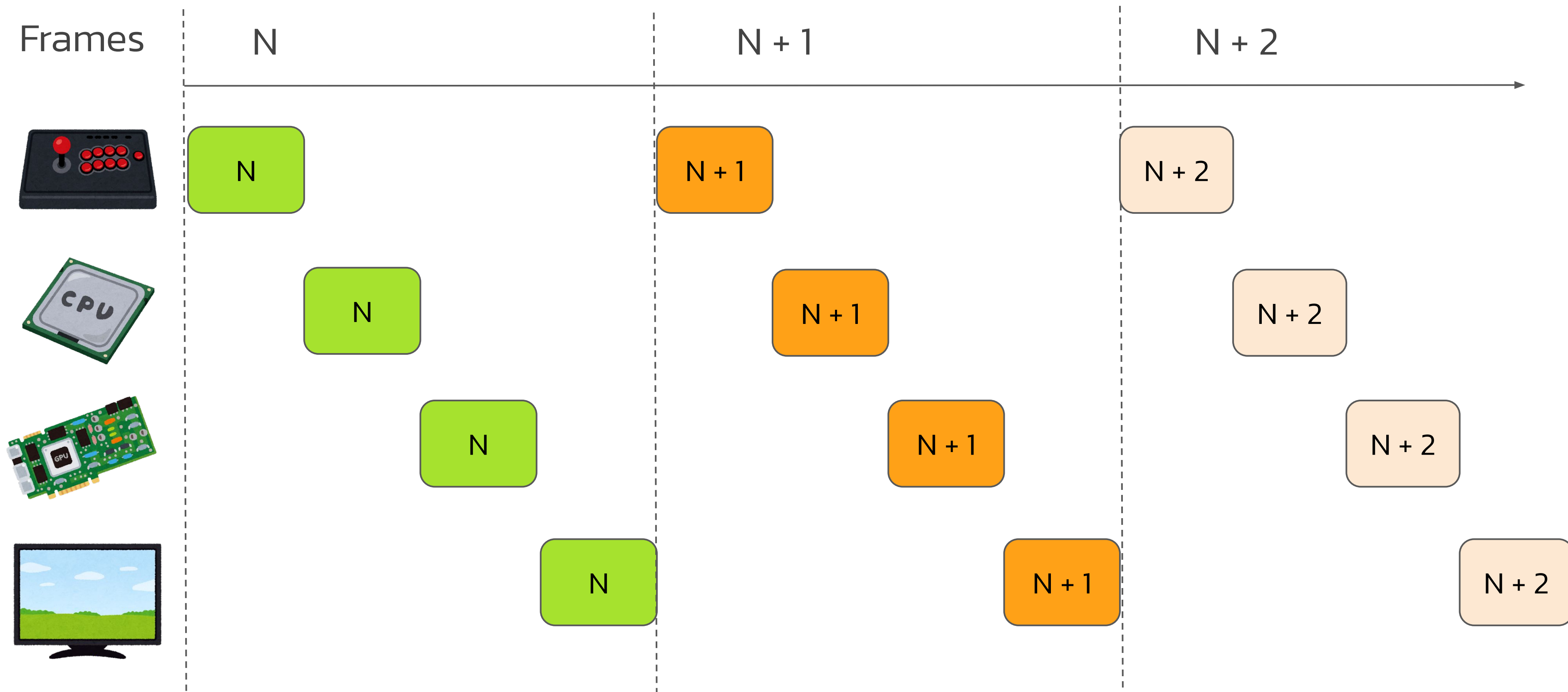


Video games are an example

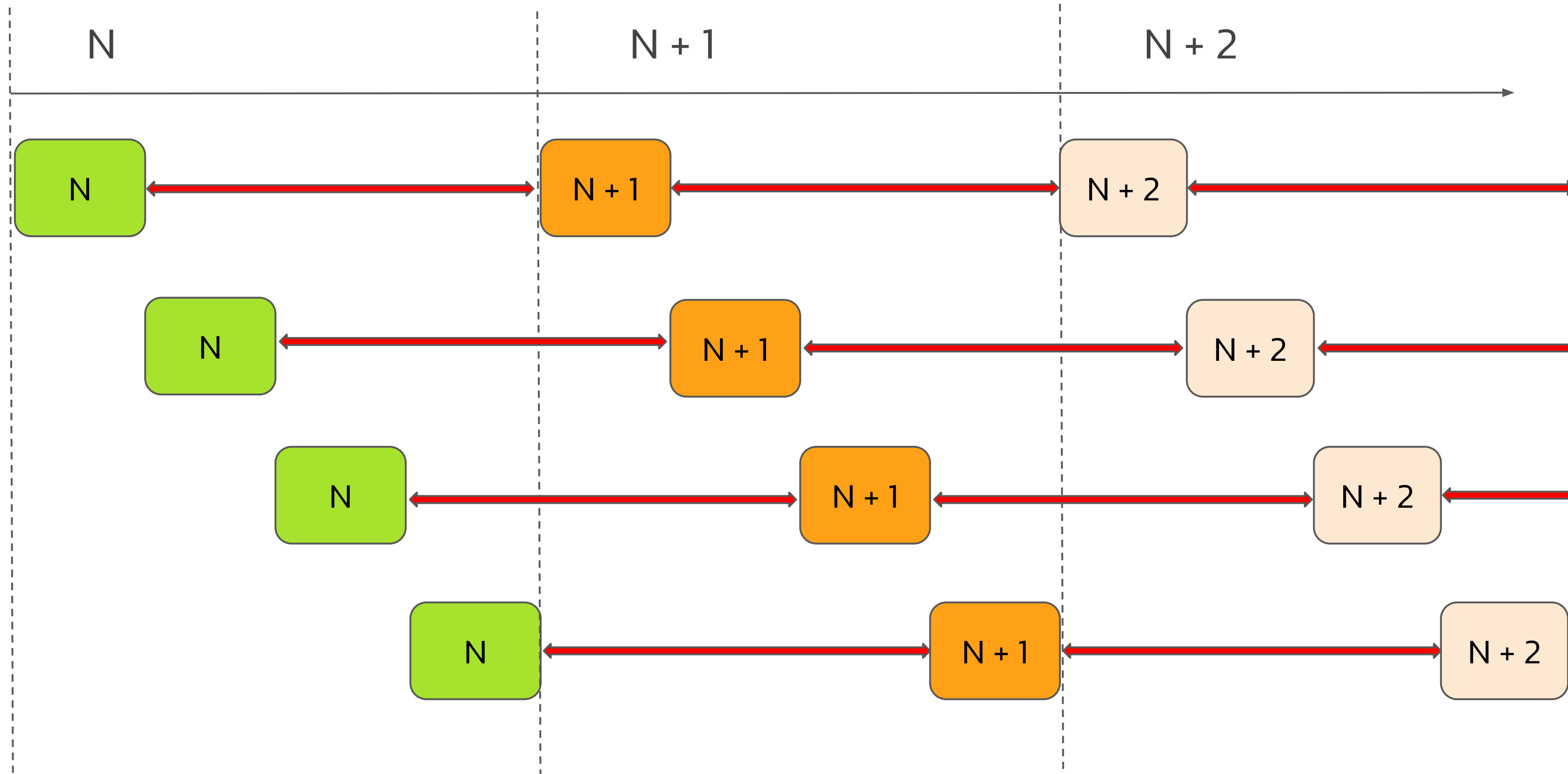
- Gather input from mouse/keyboard/controller
- Gameplay/Physics/Simulation calculated on CPU
- Rendering on GPU
- Present image to monitor



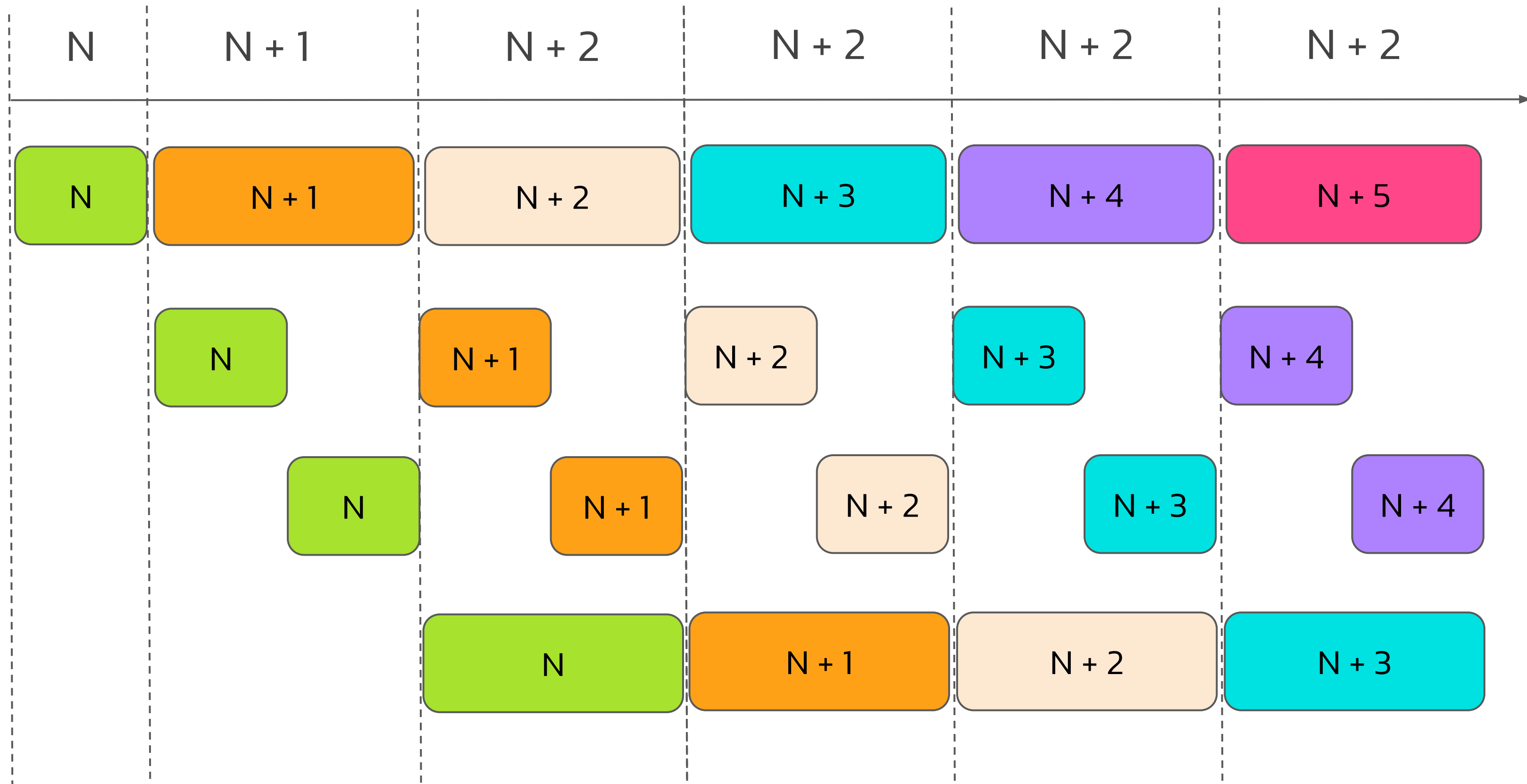
Without pipelining



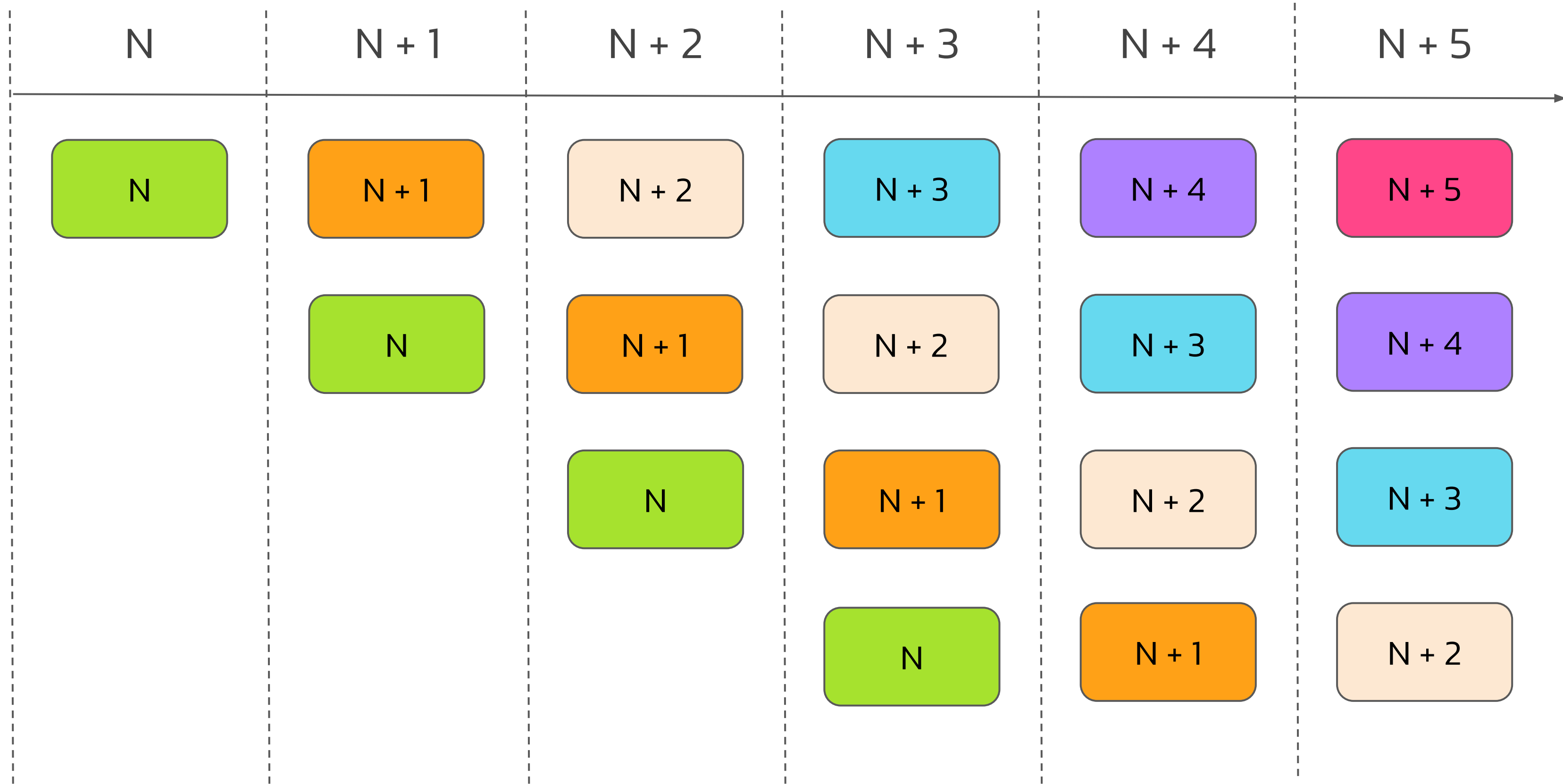
Without pipelining – idle time highlighted



With basic pipelining



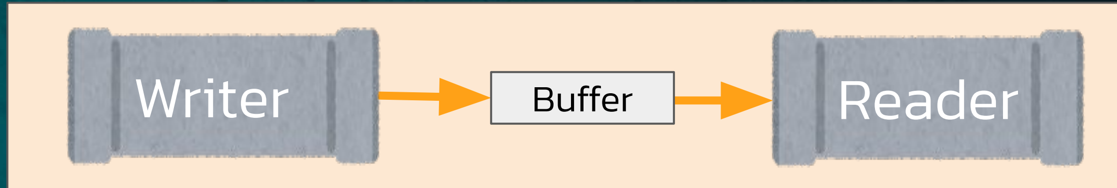
With deep pipelining – (our goal!)



Multiple Buffering

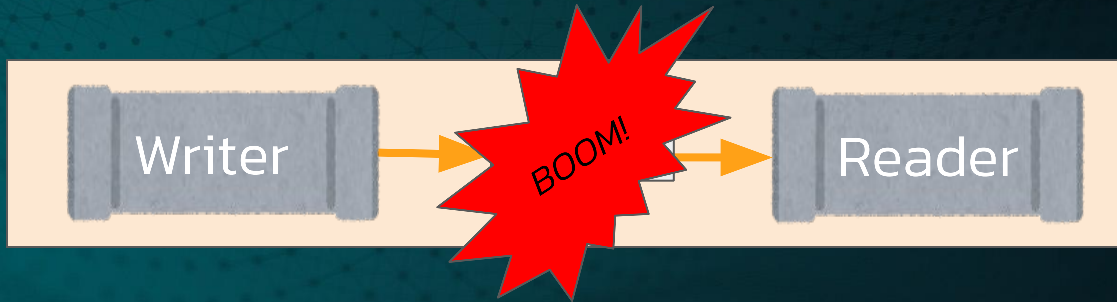
Multiple Buffering

- Stages cannot read from and write to the same data buffer



Multiple Buffering

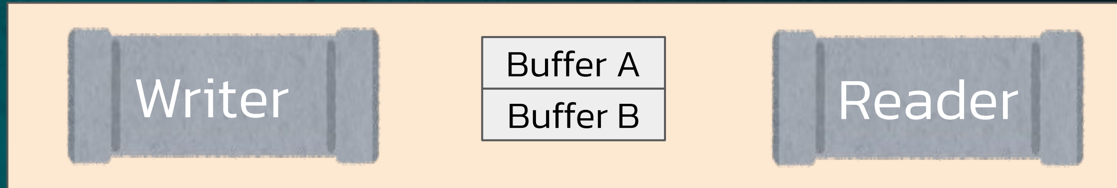
- Stages cannot read from and write to the same data buffer



LUNARG

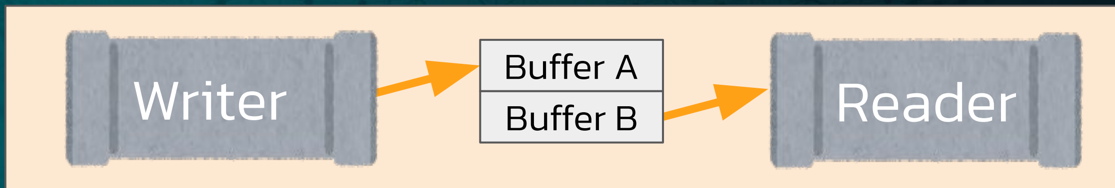
Multiple Buffering

- Stages cannot read from and write to the same data buffer
- Solution: Use multiple input/output buffers!



Multiple Buffering

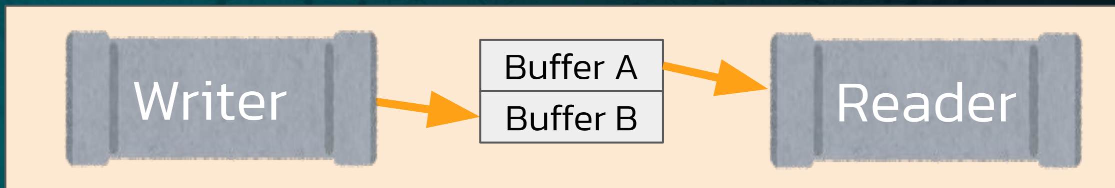
- Stages cannot read from and write to the same data buffer
- Solution: Use multiple input/output buffers!
- Write to one buffer, read from a different buffer



LUNARG

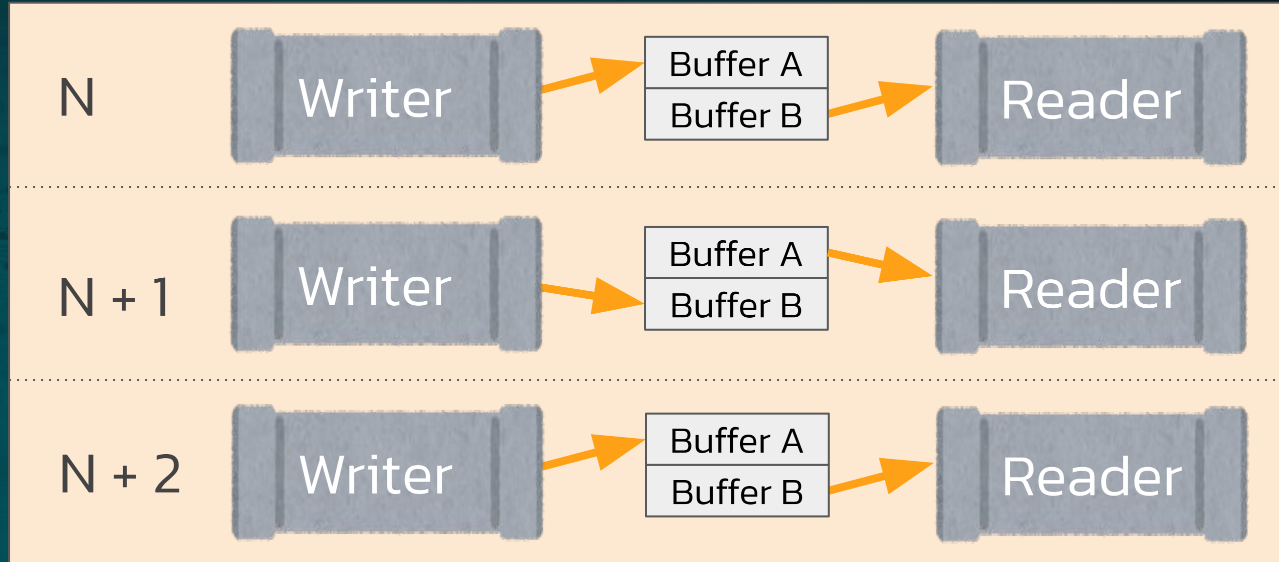
Multiple Buffering

- Stages cannot read from and write to the same data buffer
- Solution: Use multiple input/output buffers!
- Write to one buffer, read from a different buffer
- Then swap the buffers used when both stages complete



LUNARG

Multiple Buffering



LUNARG

So what *is* Frames in Flight?

LUNARG

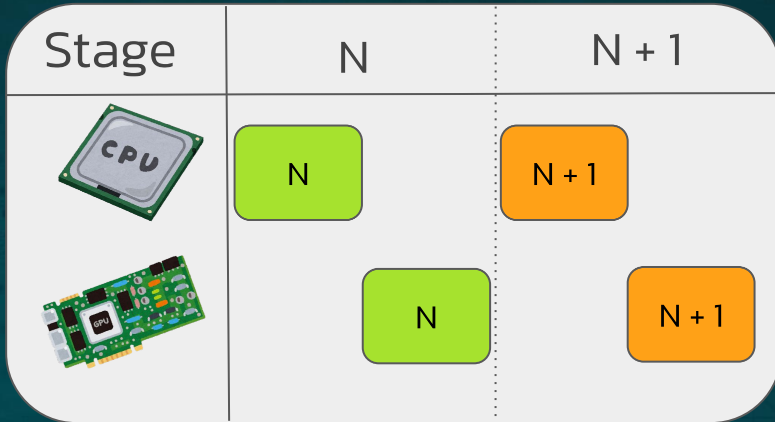
What Frames in Flight means

- The number of buffers used when pipelining Vulkan
- “Frames” A.K.A buffers of data
- “In Flight” A.K.A at the same time, in progress, concurrently
- The number of concurrent submissions in a Vulkan renderer
- A submission is all of the inputs and commands needed to render an output image
  - VkBuffer, VkCommandBuffer, VkDescriptorSet, VkImage, etc.

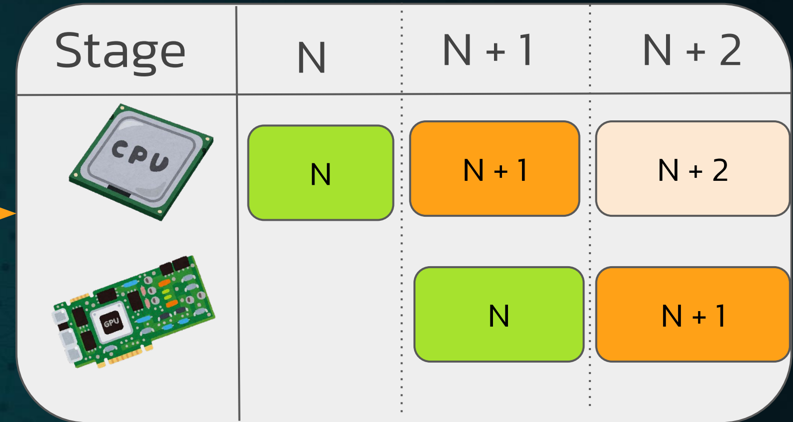
LUNARG

What Frames in Flight lets us do

Without



With



Clarifications

- Not multiple frames on the GPU concurrently
- Different from the swapchain images count

Purpose of Frames in Flight

- Increased Performance by reducing waiting
    - Lets the CPU and GPU work at the same time
    - “I bought the hardware, I’m going to use all of the hardware”
- Manage complex render state
    - Prior API’s may have implemented this in the driver

LUNARG

Implementation

Implementation

Implementation

- Create N submission groups, one for each frame in flight

Submission Group 0

Submission Group 1

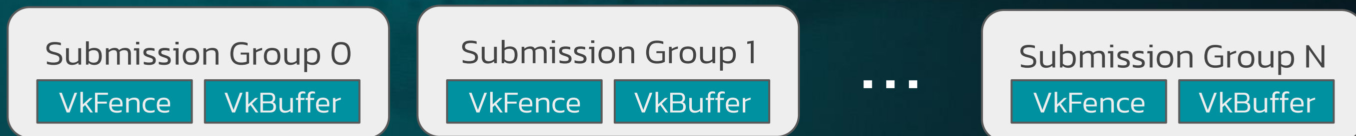
...

Submission Group N

LUNARG

Implementation

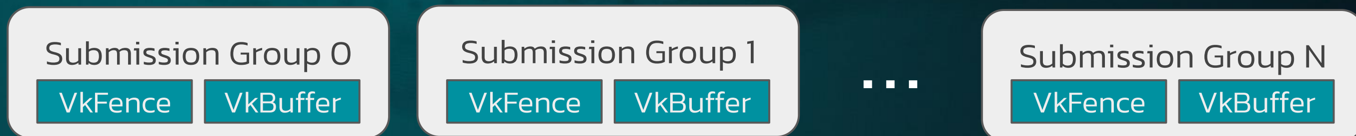
- Create N submission groups, one for each frame in flight
- Duplicate “live” resources, one into each submission group
  - Command buffers, memory allocations, sync objects, descriptor sets, etc



LUNARG

Implementation

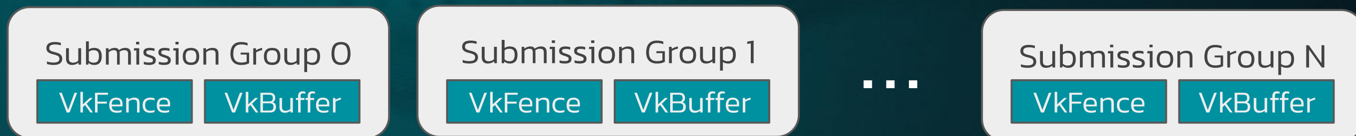
- Create N submission groups, one for each frame in flight
- Duplicate “live” resources, one into each submission group
  - Command buffers, memory allocations, sync objects, descriptor sets, etc
  - Ignore “layout” objects – pipeline, pipeline layouts, renderpass, & similar



LUNARG

Implementation

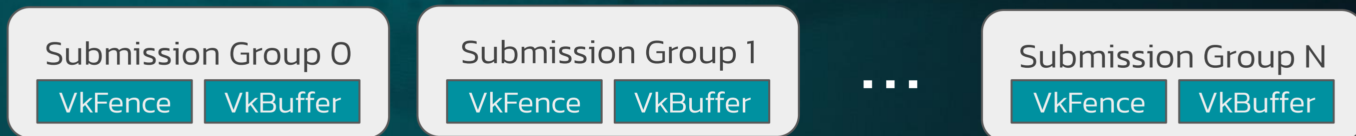
- Create N submission groups, one for each frame in flight
- Duplicate “live” resources, one into each submission group
  - Command buffers, memory allocations, sync objects, descriptor sets, etc
  - Ignore “layout” objects – pipeline, pipeline layouts, renderpass, & similar
  - Also ignore device & queue, and everything before creating a device



LUNARG

Implementation

- Create N submission groups, one for each frame in flight
- Duplicate “live” resources, one into each submission group
    - Command buffers, memory allocations, sync objects, descriptor sets, etc
    - Ignore “layout” objects – pipeline, pipeline layouts, renderpass, & similar
    - Also ignore device & queue, and everything before creating a device
- For every frame, alternate which submission group is used
    - Use an integer index to keep track of the currently active group



LUNARG

When rendering a frame

- Given a group, make sure the prior submission has finished
  - Requires waiting on a sync primitive like VkFence
- Perform all per-frame logic and updates
  - Modifying descriptors, uploading data, recording command buffers
- Submit command buffers
  - Use current submission group's sync primitive, E.G VkFence
- Increment the current submission group index variable

Basic Example Pseudo-Code

```
int FRAMES_IN_FLIGHT_COUNT = 2;
int cur_index = 0;
struct PerFrameData {
    VkFence fence;
    VkCommandBuffer cmd_buf;
    VkBuffer uniform_data;
};
PerFrameData data [ FRAMES_IN_FLIGHT_COUNT ];

void Render() {
    vkWaitForFences( data[cur_index].fence ); // Fence starts life in signaled state
    vkResetFences( data[cur_index].fence );
    upload_data( data[cur_index].uniforms );
    record_command_buffer( data[cur_index].cmd_buf, data[cur_index].uniform_data );
    vkQueueSubmit( data[cur_index].cmd_buf, data[cur_index].fence );
    cur_index = (cur_index + 1 ) % FRAMES_IN_FLIGHT_COUNT;
}
```


LUNARG

Implementation details

- Straightforward
- Up to 2x performance improvement!
- Must be careful with indexing, reads, and writes
  - Must maintain correct synchronization
- Increases memory footprint drastically!
  - Memory allocations, buffers, images, descriptor sets, etc

LUNARG

Memory Usage Patterns

LUNARG

Static memory

- Many resources are not modified after creation
- Don’t need to duplicate things that don’t change
- Examples:
  - Textures
  - Meshes
  - Shader constants
  - Descriptors/Descriptor Sets

LUNARG

Stage-Local memory

- Memory that isn't an input/output to another stage
- Created and consumed by a single stage
- Example: Depth buffer
  - Cleared at the start of a frame
  - Read & written by draw calls
  - Discarded at the end of a frame
- Also includes memory used between frames
  - Ex: Color history buffer for TAA

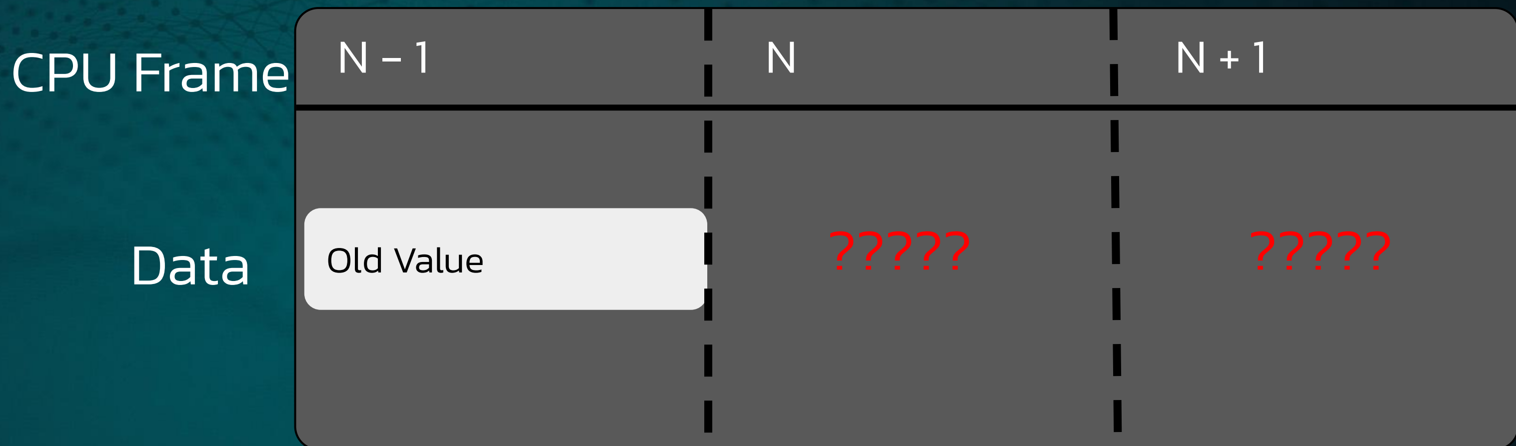
Dynamic memory

- Data that is changing every frame
- Best to duplicate
- Examples:
  - Positions/Rotations/Scale
  - Camera details
  - Animation values

Infrequently Dynamic Memory

Infrequently Dynamic Memory

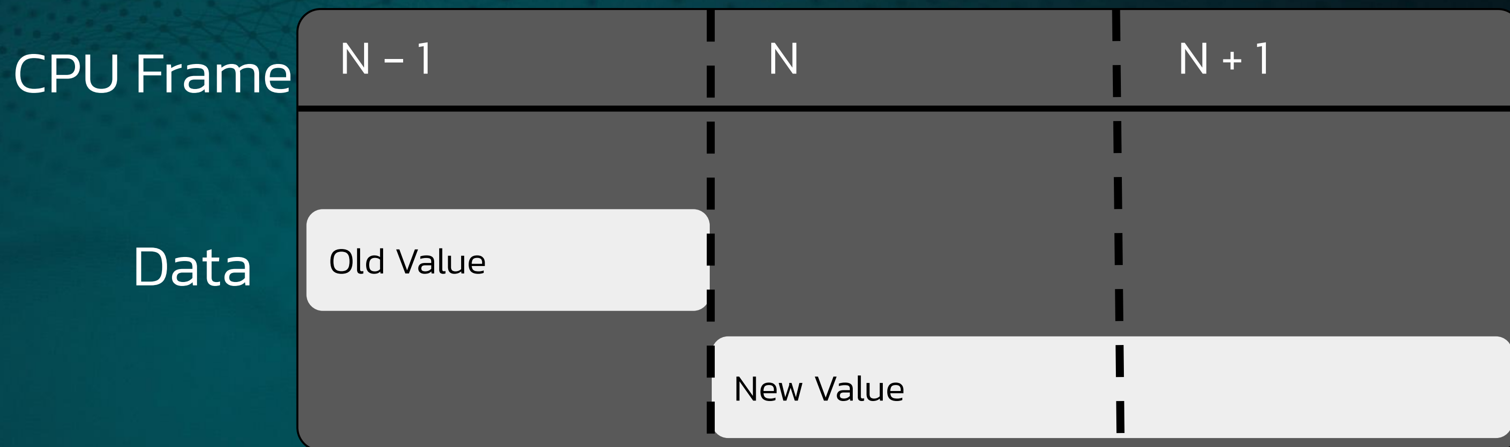
- Can change value, but not often enough to duplicate all the time



LUNARG

Infrequently Dynamic Memory

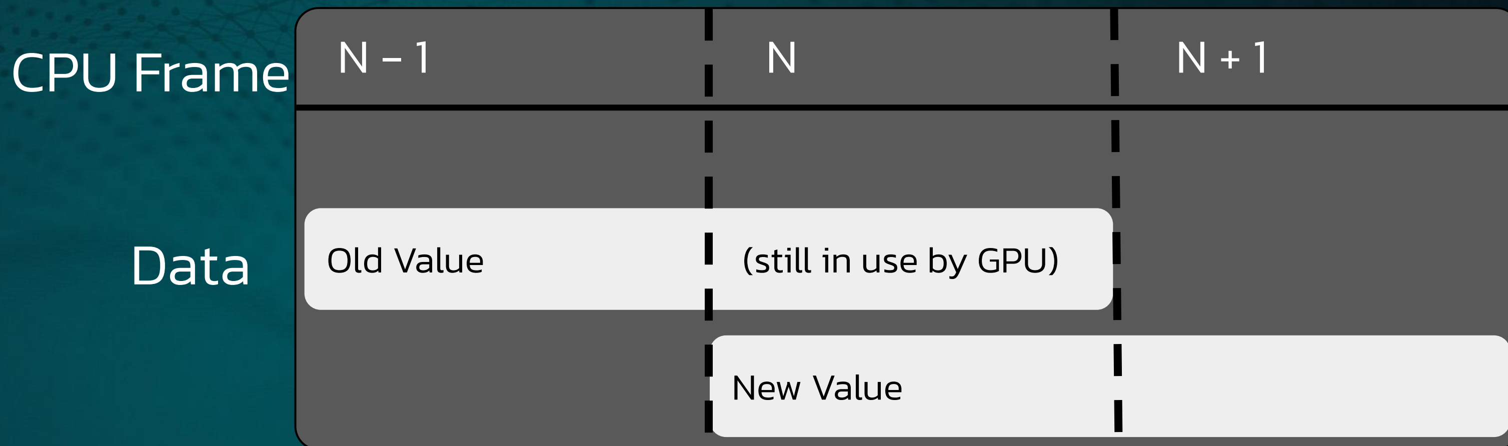
- Can change value, but not often enough to duplicate all the time
- Solution: Duplicate on demand



LUNARG

Infrequently Dynamic Memory

- Can change value, but not often enough to duplicate all the time
- Solution: Duplicate on demand
- Keep the duplicate alive only as long as needed



LUNARG

On-Demand Duplication

- Create a new resource with the updated values and use it
- Put the old resource in a ‘Deletion Queue’ for tracking
    - Prior submissions are still using it!
- When submissions are no longer using the old one, destroy it
    - Use submission group sync primitives to guarantee it
- Don’t need to fully destroy – recycling is an option
    - Consider using memory pools for buffer data

LUNARG

Practical Considerations

How many Frames in Flight should we have?

- Latency And Memory usage go up with more Frames in Flight
- Throughput does not
- 2 Frames in Flight is generally optimal
  - Good balance of throughput, latency, and memory usage
- 1 Frame in Flight is viable in the right circumstances
  - Not bottlenecked on target hardware

Pro tips

- Read and write to the correct memory!
  - Do not use augmented assignment (+=, -=) between inputs and outputs!
  - Example: `output += input * other_data;`
- Label objects with Frames in Flight index for debugging
- When debugging, reduce Frames in Flight to 1 to isolate bugs

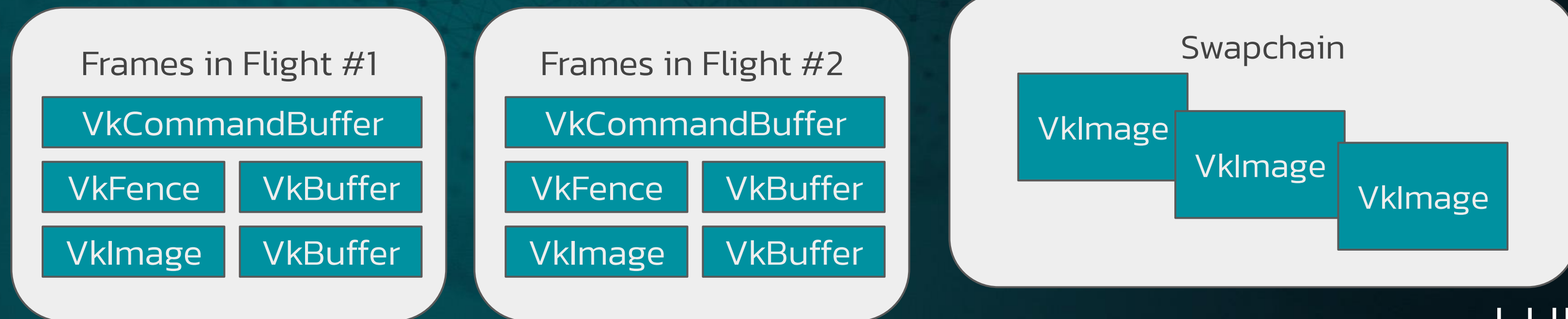
LUNARG

Swapchain

LUNARG

Swapchain and Frames in Flight

- Swapchain is the final destination for frames
  - Contains multiple images for the same reasons we duplicate resources
- Number of swapchain images decided by the driver
- Index used each frame also decided by the driver
  - Hence why we have to call vkAcquireNextImageKHR



LUNARG

Swapchain Synchronization

- Sync points:
  - vkAcquireNextImageKHR -> vkQueueSubmit
  - vkQueueSubmit -> vkQueuePresent
- Acquire to Submit: Can use VkSemaphore (binary) or VkFence
  - Use Frames in Flight count sync objects
- Submit to Present: Must use VkSemaphore (binary)
  - Use Swapchain Image count sync objects

Swapchain Integration Pseudo-Code

```
int FRAMES_IN_FLIGHT_COUNT = 2;
int cur_index = 0;
struct PerFrameData {
    VkFence fence;
    VkSemaphore image_acquired_sem;
    VkCommandBuffer cmd_buf;
};
struct PerImageData {
    VkImage image;
    VkSemaphore render_done_sem;
};
PerFrameData data [ FRAMES_IN_FLIGHT_COUNT ];
PerImageData image_data [ SWAPCHAIN_IMAGE_COUNT ];
void Render() {
    vkWaitForFences( data[cur_index].fence ); // Fence starts life in signaled state
    vkResetFences( data[cur_index].fence );
    uint32_t image_index;
    vkAcquireNextImageKHR(&image_index, /* signal */ data[cur_index].image_acquired_sem)
    record_command_buffer( data[cur_index].cmd_buf, image_data[image_index].image);
    vkQueueSubmit( data[cur_index].cmd_buf, data[cur_index].fence,
        /* wait */ data[cur_index].image_acquired_sem
        /* signal */ image_data[image_index].render_done_sem );
    vkQueuePresent( image_data[image_index].image, /* wait */ image_data[image_index].render_done_sem);

    cur_index = (cur_index + 1 ) % FRAMES_IN_FLIGHT_COUNT;
}
```


LUNARG

Summary

LUNARG

Summary

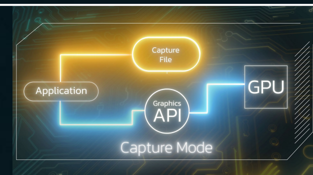
- Frames in Flight is just pipelining applied to Vulkan
- It allows CPU and GPU to run concurrently
- 2 Frames in Flight is fine

LUNARG
POWER YOUR SUCCESS

Thank you – KHRONOS GROUP

Come to the LunarG Table!

See KosmicKrisp & GFXReconstruct



Take the 2026 Vulkan Ecosystem Survey!

LunarG Presentations

Vulkanised 2026

LunarG Presentations

Shading Languages Symposium 2026

Thank you!

Questions?

LUNARG

POWER YOUR SUCCESS

LUNARG®
POWER YOUR SUCCESS

Bonus slides:

What about these Frames in Flight?

LUNARG

Frame in flight 0

vkAcquireNextImage
image_acquired_semaphore
render_done_fence
vkQueueSubmit
render_done_semaphore
vkQueuePresent

VkSwapchainKHR

VkImage 0	VkImage 1	VkImage 2

Frame in flight 1

... render_done_fence vkQueueSubmit ...

Frame resources in use

CPU

vkWaitForFences unblocks

Frame in flight 0

vkAcquireNextImage
CPU work for frame in flight 0
vkQueueSubmit
vkQueuePresent
NextFrame()

Frame in flight 1

vkWaitForFences
vkAcquireNextImage
CPU work for frame in flight 1
vkQueueSubmit
vkQueuePresent
NextFrame()
vkWaitForFences

CPU idle / GPU bound

Frame in flight 0

vkAcquireNextImage
CPU work for frame in flight 0
vkQueueSubmit
vkQueuePresent
NextFrame()

Frame in flight 1

⋮

(rendering done)
(image available)
(rendering done)
(image available)

GPU idle / CPU bound

busy
unsignaled / idle
signaled
waiting / waited on

written to
read from
unused

------➤ unblocks

Made by David DiGioia
davidofjoy@gmail.com

https://github.com/David-DiGioia/vulkan-diagrams?tab=readme-ov-file#render-loop