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

Ergonomic Vulkan with Rust & Bevy

An ECS Look at Synchronization

Hodaka Morishima, Independent

BEVY

A refreshingly simple data-driven game engine

Free and Open Source Forever!

```rust
#[derive(Component)]
struct Position {
    x: f32,
    y: f32,
}
```

```rust
#[derive(Component)]
struct Enemy {
    blood: u32,
}
```

Entity(0)	Entity(1)	Entity(2)	Entity(3)
Position{x:0,y:1}	Position{x:2,y:3}	Position{x:4,y:5}	Position{x:6,y:7}

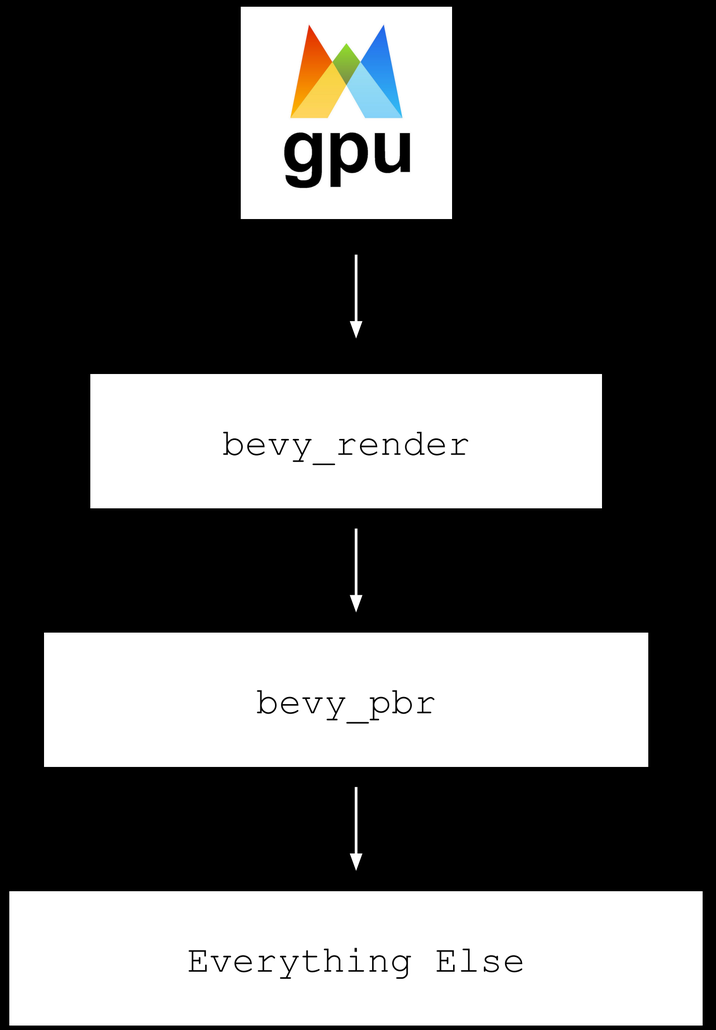
```rust
#[derive(Resource)]
struct GameState {
    stopped: bool,
}
```

Entity(0)	Entity(2)
Position{x:0,y:1}	Position{x:4,y:5}
Enemy{blood:0}	Enemy{blood:0}

```rust
struct Entity(u64);
```

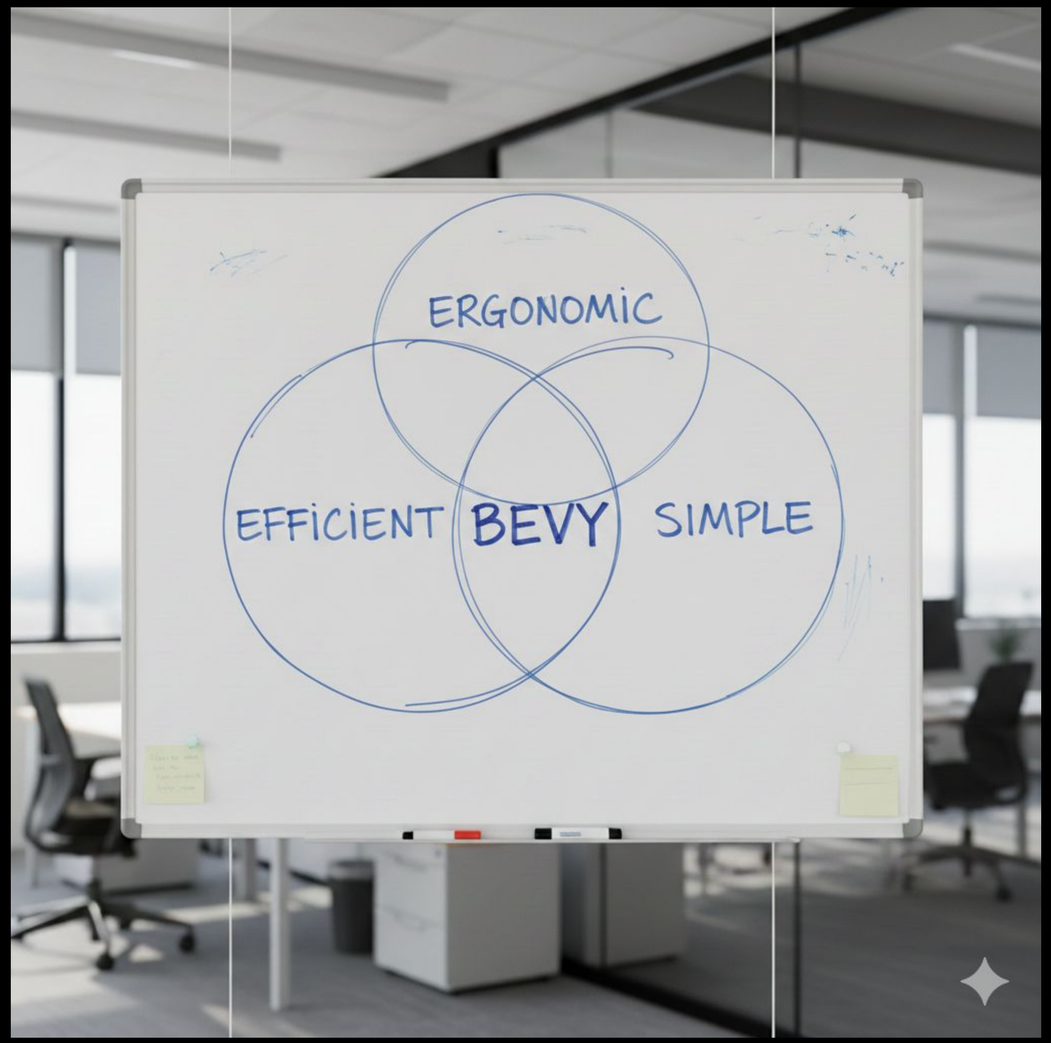
```rust
fn print_position_system(query: Query<&Position>, resource: Res<GameState>) {
    if resource.stopped {
        return;
    }
    for position in &query {
        println!("position: {} {}", position.x, position.y);
    }
}
```

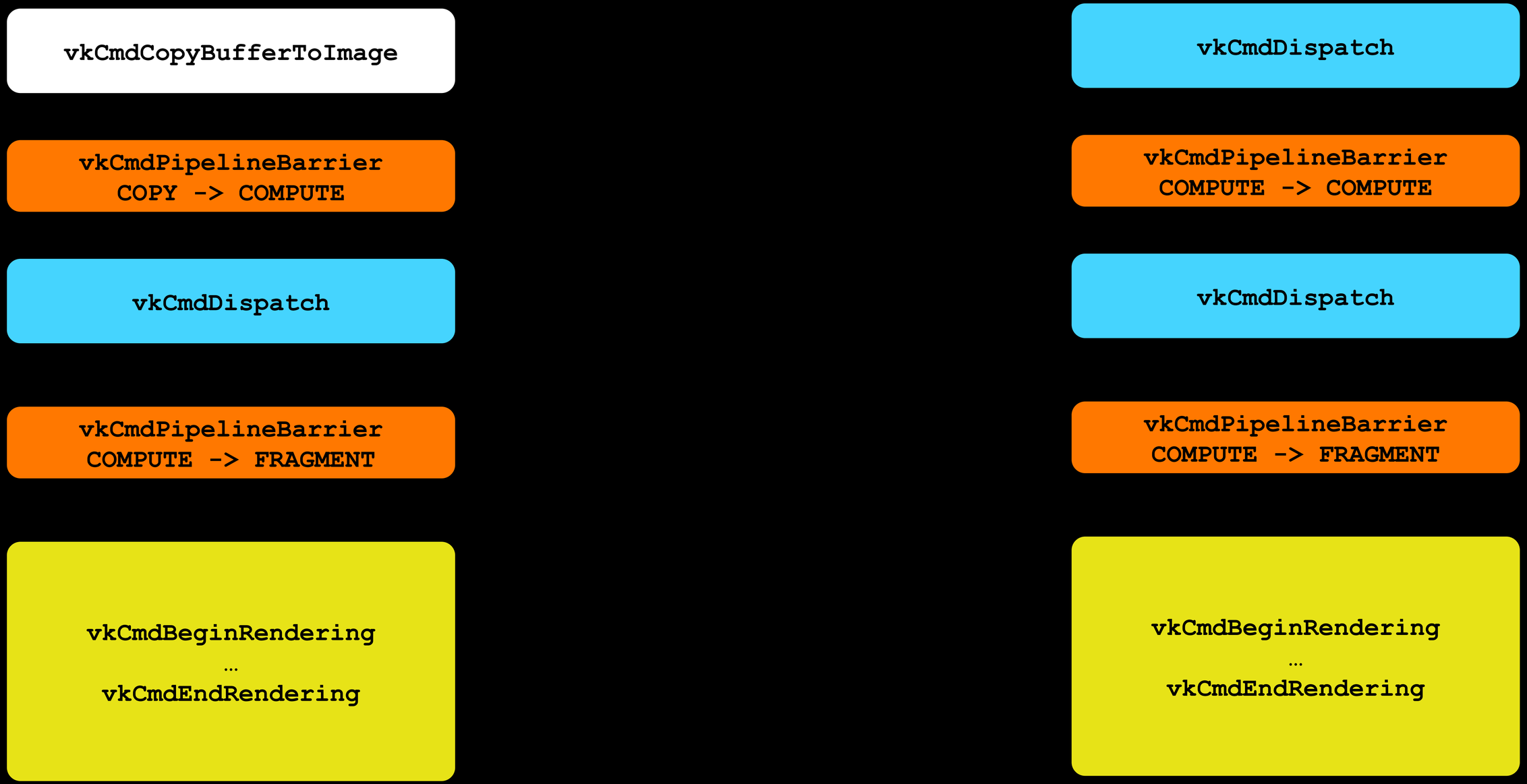
```rust
fn print_position_system(query: Query<&Position, &Enemy>) {
    …
}

fn modify_position_system(query: Query<&Position, &mut Enemy>) {
    …
}


fn main() {
    App::new()
        .add_plugins(DefaultPlugins)
        .add_systems(Update, print_position_system)
        .add_systems(Update, modify_position_system).before(print_position_system))


        // launch the app!
        .run();
}
```

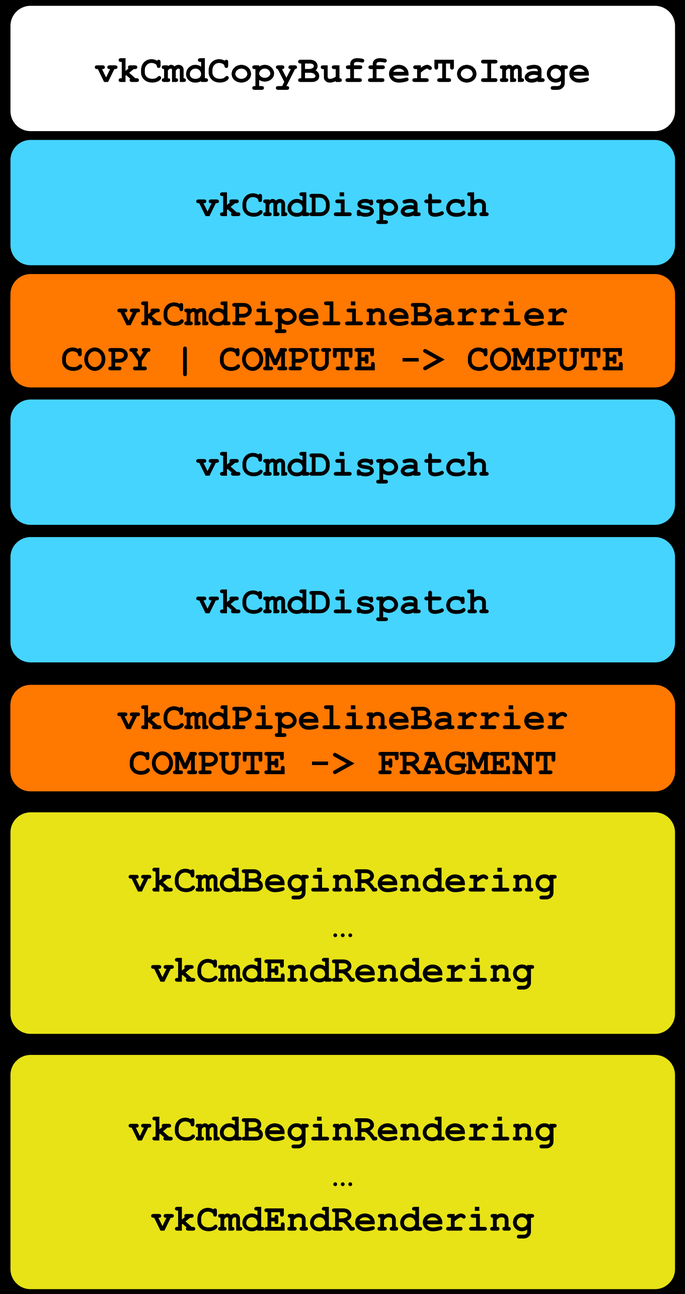

gpu
bevy_render
bevy_pbr
Everything Else

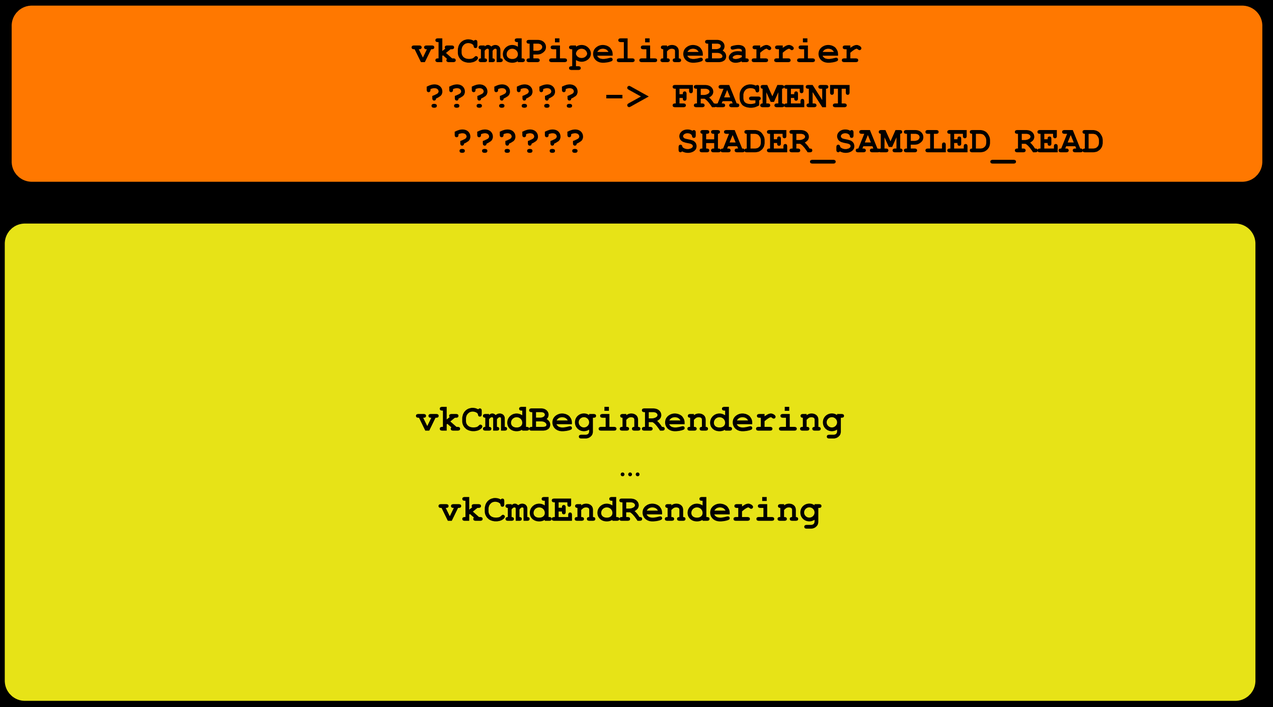
+

Ergonomic Vulkan with Rust & Bevy: An ECS Look at Synchronization


ERGONOMIC
EFFICIENT
BEVY
SIMPLE


vkCmdCopyBufferToImage
vkCmdPipelineBarrier
COPY -> COMPUTE
vkCmdDispatch
vkCmdPipelineBarrier
COMPUTE -> FRAGMENT
vkCmdBeginRendering
...
vkCmdEndRendering
vkCmdDispatch
vkCmdPipelineBarrier
COMPUTE -> COMPUTE
vkCmdDispatch
vkCmdPipelineBarrier
COMPUTE -> FRAGMENT
vkCmdBeginRendering
...
vkCmdEndRendering


vkCmdCopyBufferToImage
vkCmdDispatch
vkCmdPipelineBarrier
COPY | COMPUTE -> COMPUTE
vkCmdDispatch
vkCmdDispatch
vkCmdPipelineBarrier
COMPUTE -> FRAGMENT
vkCmdBeginRendering
...
vkCmdEndRendering
vkCmdBeginRendering
...
vkCmdEndRendering


vkCmdPipelineBarrier
??????? -> FRAGMENT
?????? SHADER_SAMPLED_READ
vkCmdBeginRendering
…
vkCmdEndRendering

Queue family 0

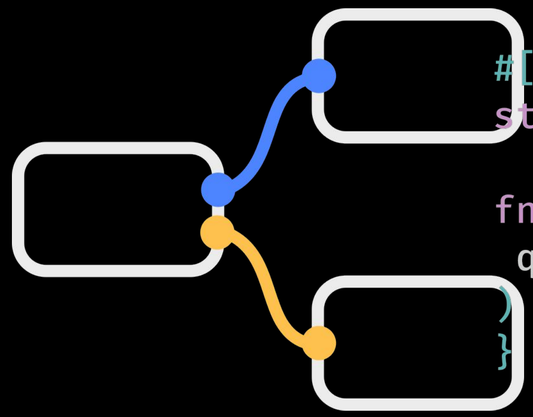
Queue family 1

vkCmdBeginRendering
…
vkCmdEndRendering

vkCmdPipelineBarrier
queueFamily 0 ->

vkCmdPipelineBarrier
-> queueFamily 1

vkCmdDispatch



```rust
#[derive(Component)]
struct Player;

fn system(
 q: Query<(Entity, &Player)>
) {
}
```

Conversation 107 Commits 78 Checks 42 Files changed 93 +6,537 −10,630

tychedelia commented on Dec 15, 2025 · edited Member

render-graph-as-systems

> **Note**
>
> Remember to check hide whitespace in diff view options when reviewing this PR

This PR removes the `RenderGraph` in favor of using systems.

Motivation

The `RenderGraph` API was originally created when the ECS was significantly more immature. It was also created with the intention of supporting an input/output based slot system for managing resources that has never been used. While resource management is an important potential use of a render graph, current rendering code doesn't make use of any patterns relating to it.

Since the ECS has improved, the functionality of `Schedule` has basically become co-extensive with what the `RenderGraph` API is doing, i.e. ordering bits of system-like logic relative to one another and executing them in a big chunk. Additionally, while there's still desire for more advanced techniques like resource management in the graph, it's desirable to implement those in ECS terms rather than creating more `RenderGraph` specific abstraction.

In short, this sets us up to iterate on a more ECS based approach, while deleting ~3k lines of mostly unused code.

Reviewers
- alice-i-cecile
- cart ✓
- IceSentry
- ecoskey
- +5 more reviewers
  - dcvz
  - atlv24 ✓
  - ickshonpe
  - JMS55
  - andriyDev ✓

Assignees
No one assigned

Labels
A-Rendering C-Feature D-Domain-Expert M-Migration-Guide M-Release-Note S-Needs-SME X-Blessed

Merged **Replace `RenderGraph` with systems** #22144
cart merged 78 commits into `bevyengine:main` from `tychedelia:render-graph-as-…` 5 days ago

At a high level: We use `Schedule` as our "sub-graph." Rather than running the graph, we run a schedule. Systems can be ordered relative to one another.

The render system uses a `RenderGraph` schedule to define the "root" of the graph. `core_pipeline` adds a `camera_driver` system that runs the per-camera schedules. This top level schedule provides an extension point for apps that may want to do custom rendering, or non-camera rendering.

`CurrentView` / `ViewQuery`

Rendering
Status: Done +1 more

Milestone
0.19

A typical Vulkan Render Graph implementation

…and other things the “render graph at home” doesn’t do

GPU Synchronization

Analyzes dependencies between passes to automatically insert the correct pipeline barriers and semaphores

Image Layout Transitions

Tracks the state of every image resource and automatically transition them to the optimal layout

Memory Aliasing

Identify the timeline of resource usages and alias logical resources to use the same underlying physical memory

Render Pass Merging

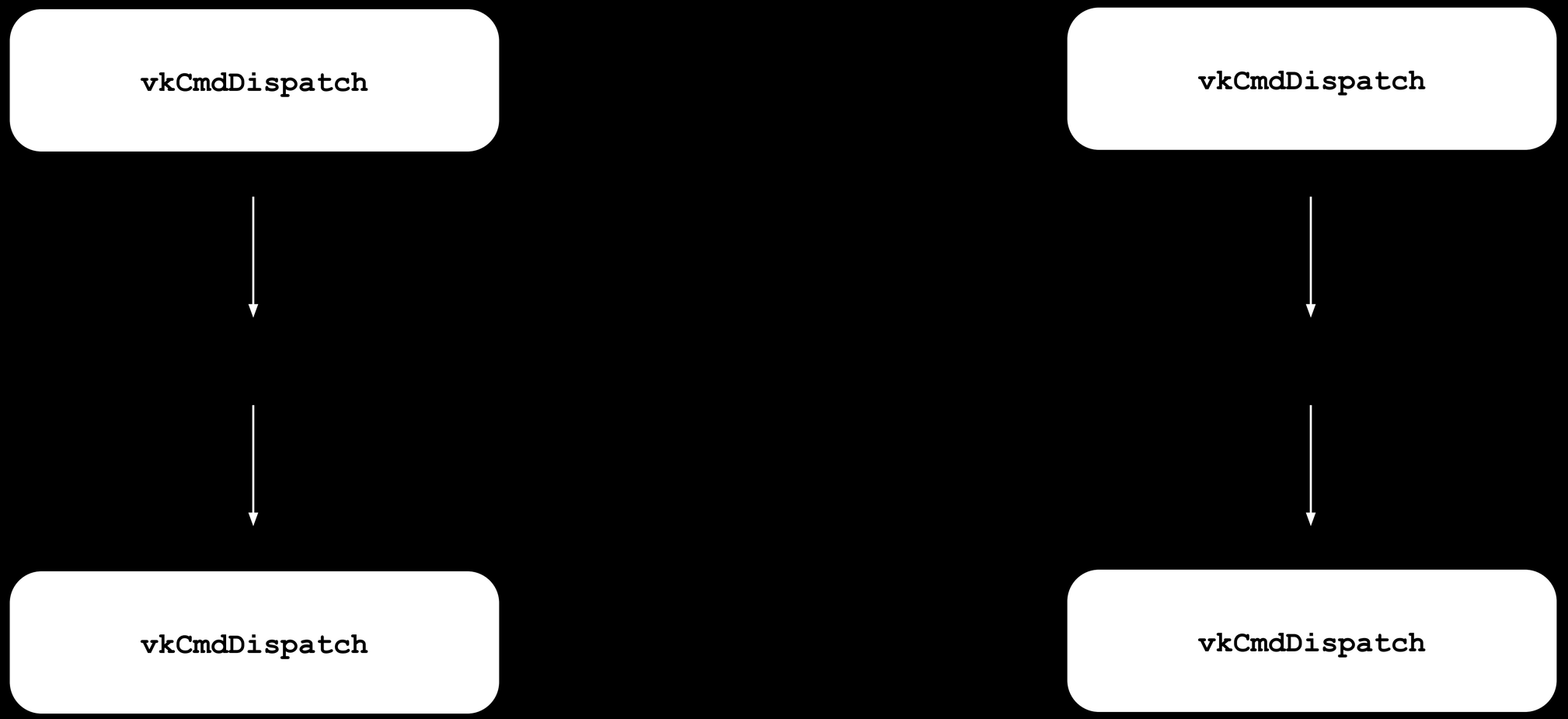
Identify passes that can be combined into a single render pass - crucial for tile-based GPU

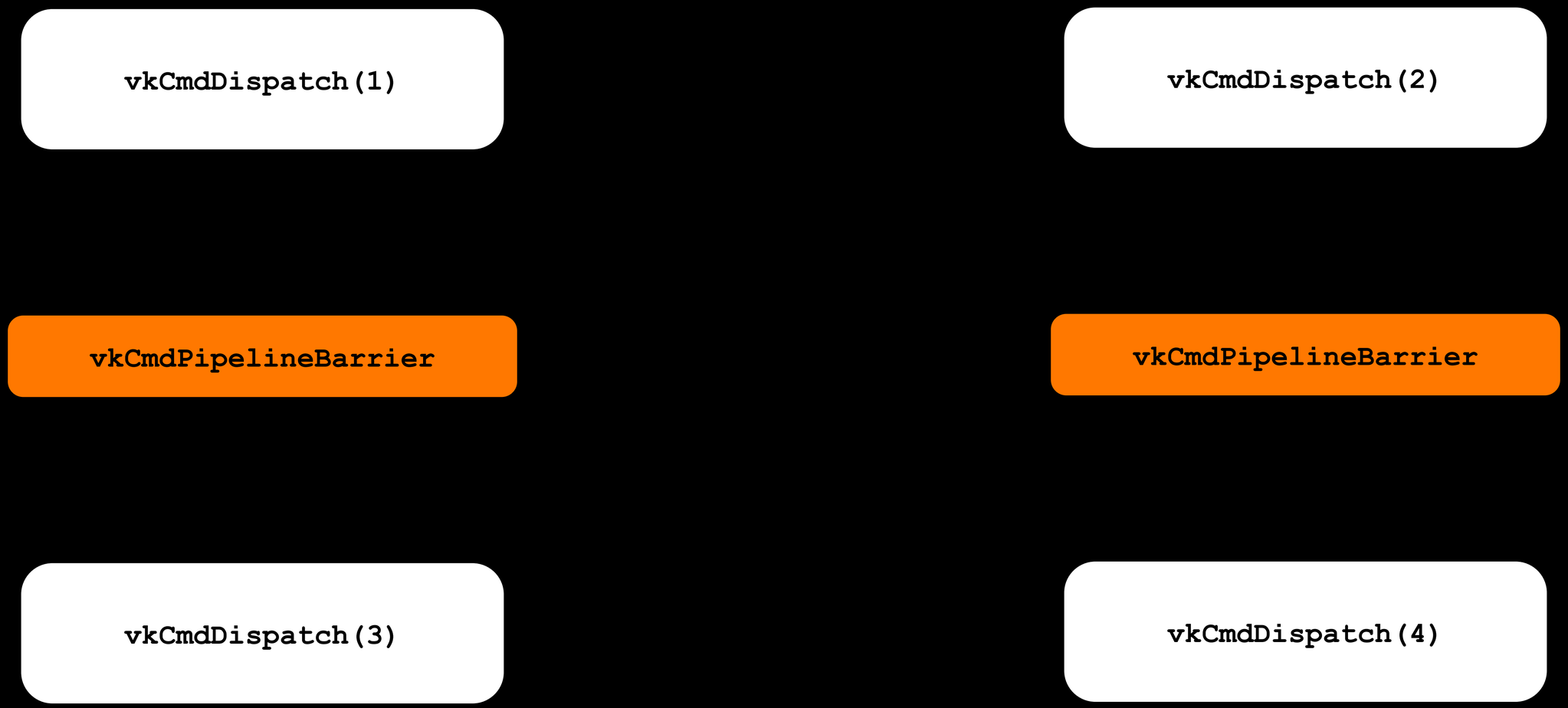
Resource Lifetime Management

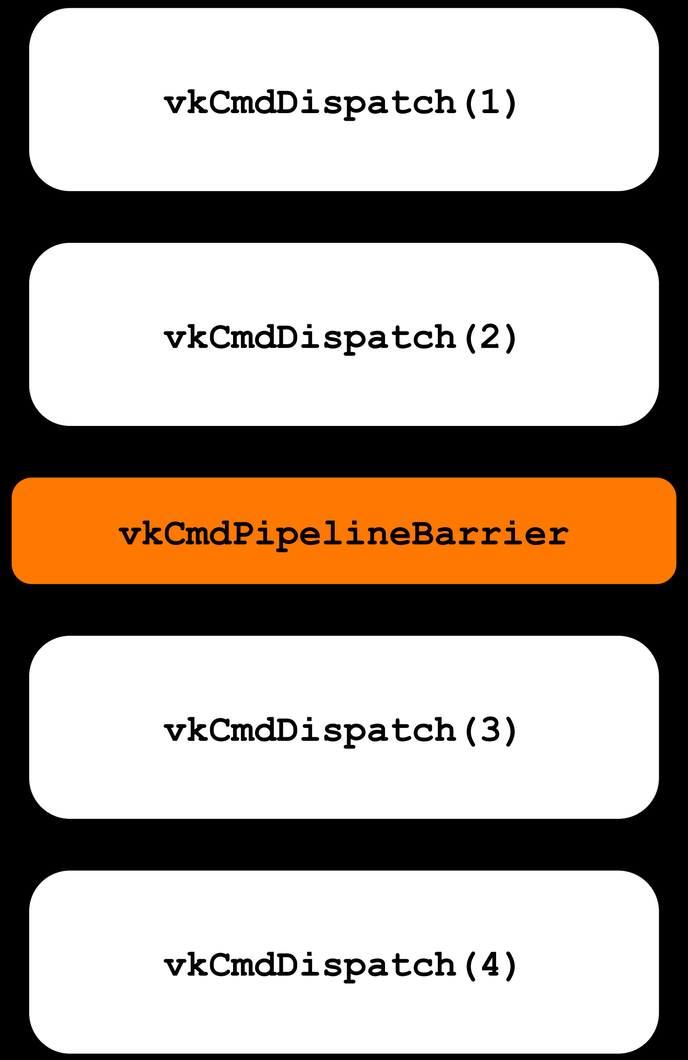
Track resource lifetimes and ensure that resources stay alive while the command buffer stays pending

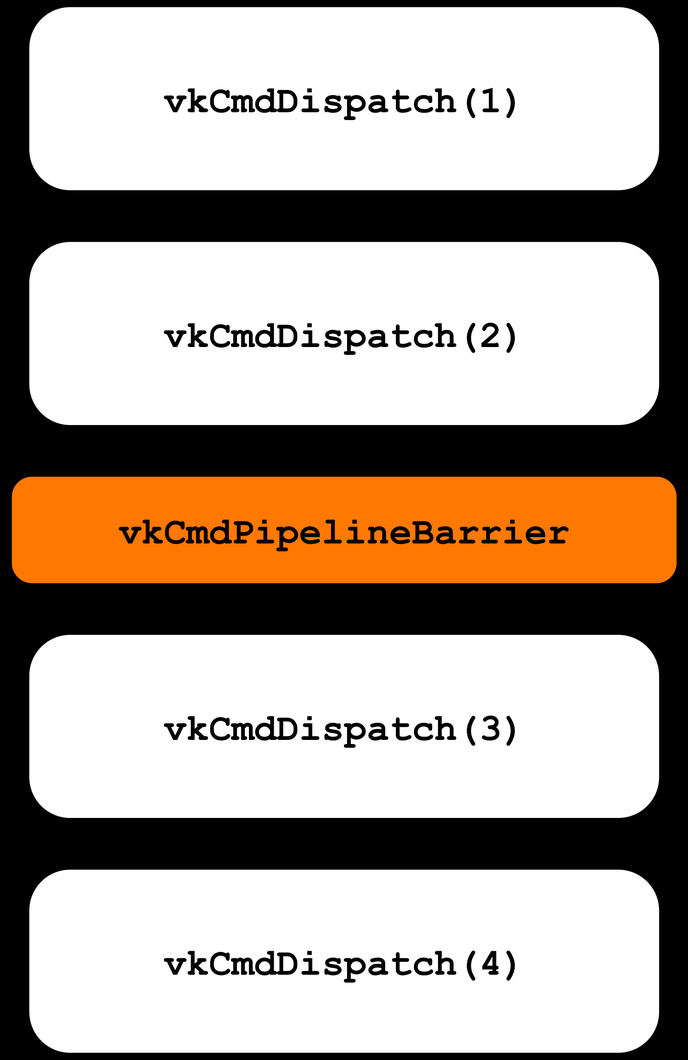
Parallel Encoding

Allows independent render graph nodes to encode command buffers in parallel

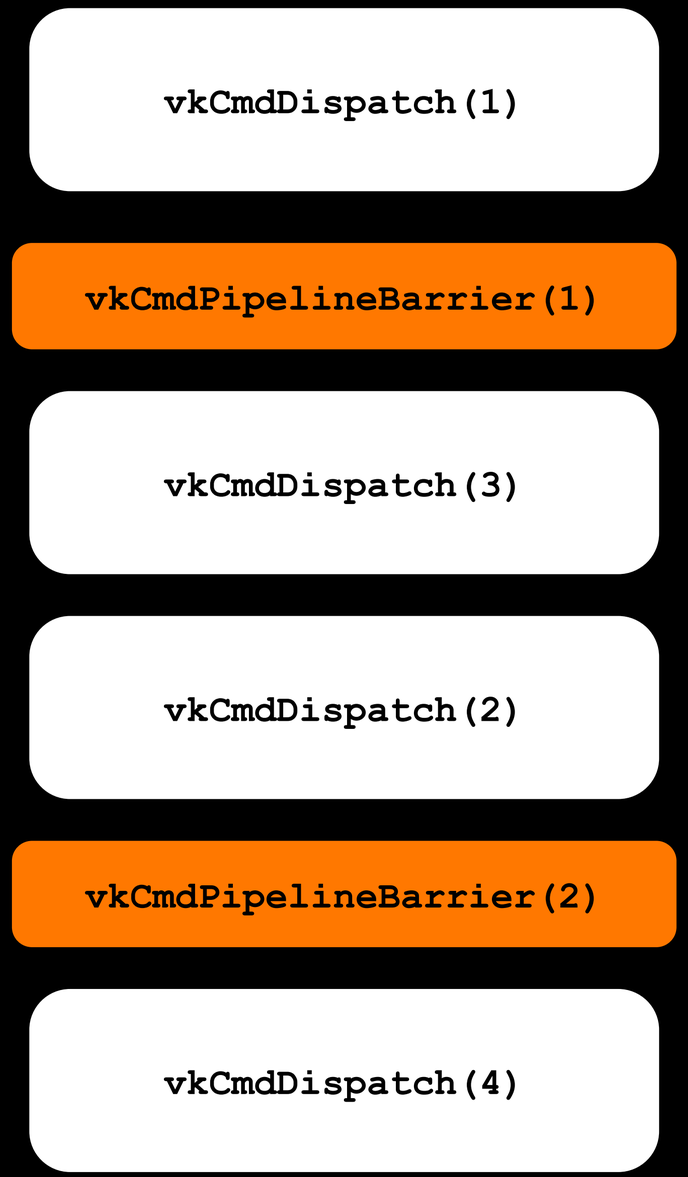

vkCmdDispatch
vkCmdDispatch
vkCmdDispatch
vkCmdDispatch


vkCmdDispatch(1)
vkCmdDispatch(2)
vkCmdPipelineBarrier
vkCmdPipelineBarrier
vkCmdDispatch(3)
vkCmdDispatch(4)

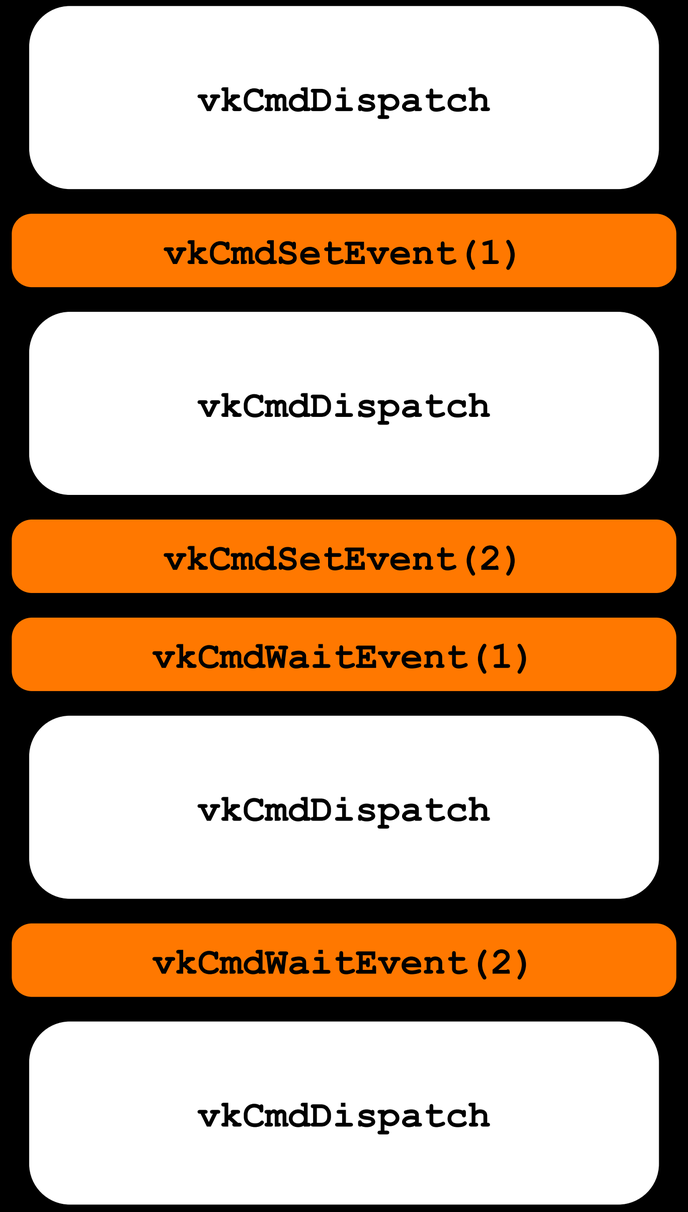

vkCmdDispatch(1)
vkCmdDispatch(2)
vkCmdPipelineBarrier
vkCmdDispatch(3)
vkCmdDispatch(4)




```rust
impl Plugin for MyPlugin {
    fn build(&self, app: &mut App) {
        app.add_systems(PostUpdate, (
            dispatch1,
            dispatch2,
        ).before(pipeline_barrier));

        app.add_systems(PostUpdate, (
            dispatch3,
            dispatch4,
        ).after(pipeline_barrier));
    }
}
```




```rust
impl Plugin for MyPlugin1 {
    fn build(&self, app: &mut App) {
        app.add_systems(PostUpdate, (
            dispatch2,
            pipeline_barrier2,
            dispatch4,
        ).chain());
    }
}
impl Plugin for MyPlugin2 {
    fn build(&self, app: &mut App) {
        app.add_systems(PostUpdate, (
            dispatch1,
            pipeline_barrier1,
            dispatch3,
        ).chain());

    }
}
```


vkCmdDispatch
vkCmdSetEvent(1)
vkCmdDispatch
vkCmdSetEvent(2)
vkCmdWaitEvent(1)
vkCmdDispatch
vkCmdWaitEvent(2)
vkCmdDispatch

```
fn computepass() {

    dispatch1();

    // Hey Claude, put some magic barrier here.
    // Dispatch3 will use a texture produced by Dispatch1.

    dispatch3();

}
```

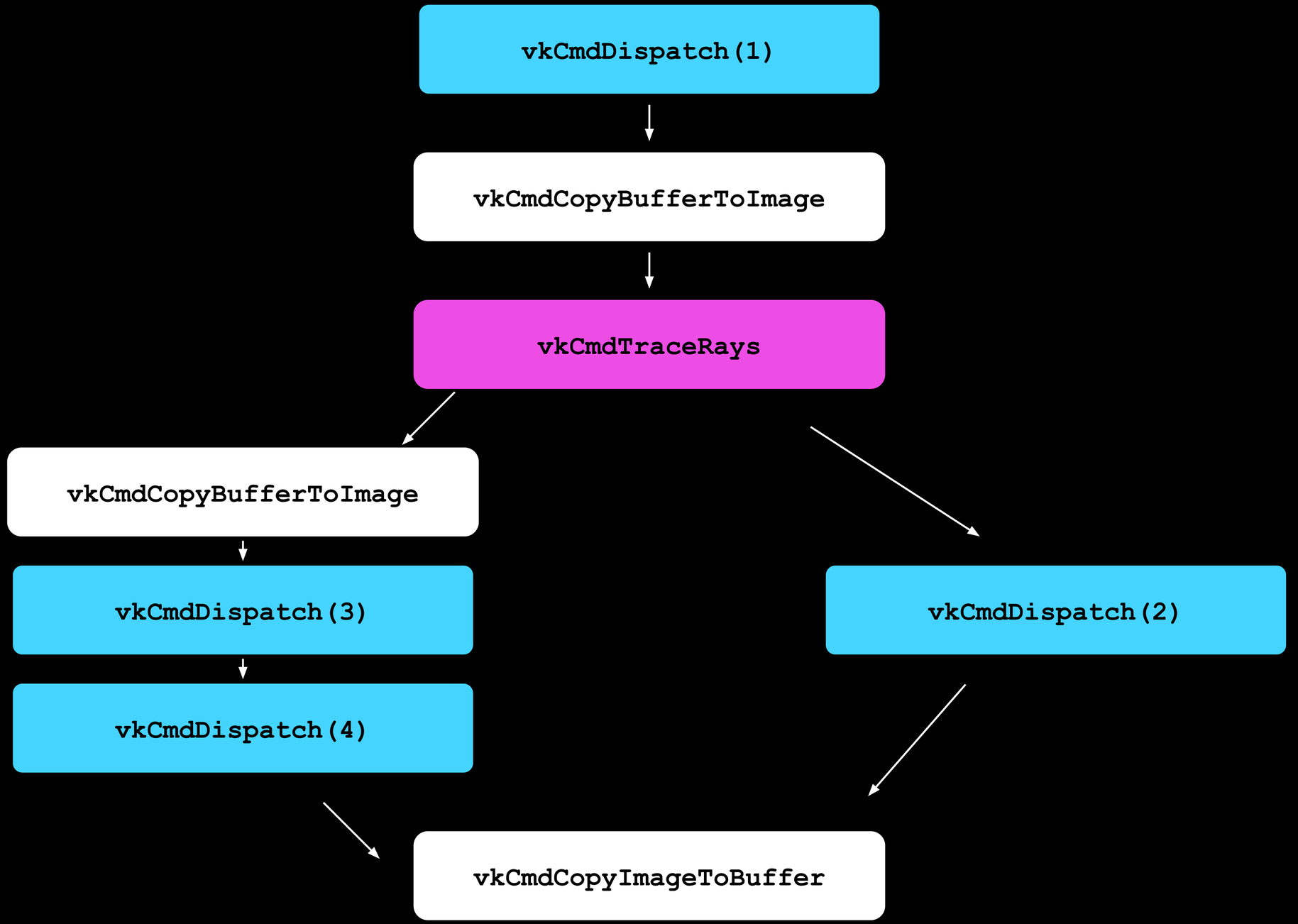
```rust
fn computepass1() {

    dispatch1();

    yield;

    dispatch3();

}
```

```rust
fn computepass2() {

    dispatch2();

    yield;

    dispatch4();

}
```

```rust
impl Plugin for MyPlugin {
    fn build(&self, app: &mut App) {
        app.add_systems(PostUpdate, computepass1);

        app.add_systems(PostUpdate, computepass2);
    }
}
```

```
fn computepass() {
    dispatch1();
    yield;
    computepass2();
    yield;
    zip(computepass3(), dispatch2());
    yield;
    copy_image_to_buffer();
}
```

```
fn computepass2() {
    copy_buffer_to_image();
    yield;
    trace_rays();
fn computepass3() {
}
    copy_buffer_to_image();
    yield;
    dispatch3();
    yield;
    dispatch4();
}
```


vkCmdDispatch(1)
vkCmdCopyBufferToImage
vkCmdTraceRays
vkCmdCopyBufferToImage
vkCmdDispatch(3)
vkCmdDispatch(4)
vkCmdDispatch(2)
vkCmdCopyImageToBuffer

```rust
fn pass1() {
    draws1();
    yield;
    dispatch1();
}
```

```rust
fn pass2() {
    dispatch2();
    yield;
    draws2();
}
```

```rust
impl Plugin for MyPlugin {
    fn build(&self, app: &mut App) {
        app.add_systems(PostUpdate, pass1);

        app.add_systems(PostUpdate, pass2);
    }
}
```

```
fn pass2() {
fn pass1() {
   dispatch2();
   yield;
   draws1();
   draws2();
   yield;
   dispatch1();
 }
 }
```

A typical Vulkan Render Graph implementation

…and other things the “render graph at home” doesn’t do

GPU Synchronization

Analyzes dependencies between passes to automatically insert the correct pipeline barriers and semaphores

Image Layout Transitions

Tracks the state of every image resource and automatically transition them to the optimal layout

Memory Aliasing

Identify the timeline of resource usages and alias logical resources to use the same underlying physical memory

Render Pass Merging

Identify passes that can be combined into a single render pass - crucial for tile-based GPU

Resource Lifetime Management

Track resource lifetimes and ensure that resources stay alive while the command buffer stays pending

Parallel Encoding

Allows independent render graph nodes to encode command buffers in parallel

```rust
fn pass1<'a>(
    encoder: &mut CommandEncoder<'a>, // 'a indicates the command buffer lifetime
    pipeline: GraphicsPipeline) {

    let pipeline: &'a GraphicsPipeline = encoder.retain(pipeline);
    // pipeline will live until the command buffer was dropped


    // bind_pipeline enforces that the reference must outlive 'a
    encoder.bind_pipeline(pipeline);
}
```

Trying to extend the Rust safety model to the GPU is a **great** idea 

```rust
fn pass1<'a>(
    encoder: &mut CommandEncoder<'a>, // 'a indicates the command buffer lifetime
    pipeline: GraphicsPipeline) {

    let pipeline: &'a GraphicsPipeline = encoder.retain(pipeline);

    // Bind once, use forever!
    encoder.bind_pipeline(vk::PipelineBindPoint::GRAPHICS, pipeline);
    encoder.bind_pipeline(vk::PipelineBindPoint::COMPUTE, pipeline);
    encoder.bind_pipeline(vk::PipelineBindPoint::RAY_TRACING_KHR, pipeline);
}
```

```rust
struct GPUMutex<T> {
    semaphore: vk::Semaphore,
    value: u64,
    item: Box<T>
}



fn pass1<'a>(
    encoder: &mut CommandEncoder<'a>,
    image: GPUMutex<Image>) {

    // 1. Have the COMPUTE_SHADER stage of this command buffer wait on
    //    GPUMutex::semaphore until it reaches GPUMutex::value
    // 2. Swap GPUMutex::semaphore to be the timeline semaphore that this
    //    command encoder is going to signal upon completion
    let image: &'a Image = encoder.lock(&image, COMPUTE_SHADER);

}
```

```rust
async fn run(image: &GPUMutex<Image>) {
    // Wait on the GPUMutex, blocking the current thread
    image.unwrap_block();


    // Wait on the GPUMutex asynchronously
    image.unwrap_block_async().await;
}
```

```rust
fn run<'a>(
    encoder: &mut CommandEncoder<'a>,
    image: &GPUMutex<Image>, // What state is this in?
) {
    let image = encoder.lock(image, vk::PipelineStageFlags2::COPY);



    encoder.image_barrier(
        image,
        /* before_access */ // what should I put here??,
        /* after_access */ Access::COPY_WRITE,
        /* old_layout */ // what should I put here???,
        /* new_layout */ vk::ImageLayout::TRANSFER_DST_OPTIMAL,
        …
    );


    encoder.copy_buffer_to_texture(
      some_buffer,
      image,
      &[vk::BufferImageCopy { .. }],
      vk::ImageLayout::TRANSFER_DST_OPTIMAL
    );
}
```

Tracking Resource States

- Track Globally: A centralized HashMap<Resource, State>
- Interim Barrier: Global States + Per-Command-Buffer State

Vulkanised 2024: Vulkan Synchronization Made Easy - Grigory Dzhavadyan

- Render Graph
- Trust but Verify

```rust
fn run<'a>(
    encoder: &mut CommandEncoder<'a>,
    image: &GPUMutex<Image>, // What state is this in?
) {
    let image = encoder.lock(image, vk::PipelineStageFlags2::COPY);



    encoder.image_barrier(
        image,
        /* before_access */ // what should I put here??,
        /* after_access */ Access::COPY_WRITE,
        /* old_layout */ // what should I put here???,
        /* new_layout */ vk::ImageLayout::TRANSFER_DST_OPTIMAL,
        …
    );


    encoder.copy_buffer_to_texture(
      some_buffer,
      image,
      &[vk::BufferImageCopy { .. }],
      vk::ImageLayout::TRANSFER_DST_OPTIMAL
    );
}
```

```rust
fn run<'a>(
    encoder: &mut CommandEncoder<'a>,
    image: &GPUMutex<Image>,
    state: &mut ResourceState,
) {
    let image = encoder.lock(image, vk::PipelineStageFlags2::COPY);


    encoder.image_barrier(
        image,
        /* before_access */ state.access,
        /* after_access */ Access::COPY_WRITE,
        /* old_layout */ state.layout,
        /* new_layout */ vk::ImageLayout::TRANSFER_DST_OPTIMAL,
        …
    );


    encoder.copy_buffer_to_texture(
      some_buffer,
      image,
      &[vk::BufferImageCopy { .. }],
      vk::ImageLayout::TRANSFER_DST_OPTIMAL
    );
}
```

```rust
#[derive(Resource)]
struct MyTexture {
    image: GPUMutex<Image>,
    state: ResourceState
}
```

```rust
#[derive(Component)]
struct BLASBackBuffer {
    buffer: GPUMutex<Buffer>,
    state: ResourceState
}
```

```rust
fn system(
    render_state: RenderState,
    texture: ResMut<MyTexture>
) {
    render_state.record(|encoder| {
        let image = encoder.lock(&texture.image, vk::PipelineStageFlags2::COPY);
        encoder.use_image_resource(
            image,
            &mut texture.state,
            Access::COPY_WRITE,
            vk::ImageLayout::TRANSFER_DST_OPTIMAL,
            ... );

        encoder.copy_buffer_to_texture(
            some_buffer,
            image,
            &[vk::BufferImageCopy { .. }],
            vk::ImageLayout::TRANSFER_DST_OPTIMAL
        );
    });
}
```

```rust
fn system(
    render_state: RenderState,
    texture: ResMut<MyTexture>
) {
    let texture_state: ResourceStateGuard = texture.state.eventually(
        Access::COPY_WRITE,
    );
    render_state.record(|encoder| {
        let image = encoder.lock(&texture.image, vk::PipelineStageFlags2::COPY);
        encoder.use_image_resource(
            image,
            &mut texture_state,
            Access::COPY_WRITE,
            vk::ImageLayout::TRANSFER_DST_OPTIMAL,
             ... );

        encoder.copy_buffer_to_texture(
            some_buffer,
            image,
            &[vk::BufferImageCopy { .. }],
            vk::ImageLayout::TRANSFER_DST_OPTIMAL
        );
    });
}
```

```rust
#[derive(Component)]
struct MyTextureArray {
    texture_array: GPUMutex<Image>,
    state: Vec<ResourceState>  // One per array layer
}
```

```rust
#[derive(Component)]
struct MyTextureCollection {
    texture_array: GPUMutex<Vec<Image>>,
    state: ResourceState  // Many images sharing the same state
}
```

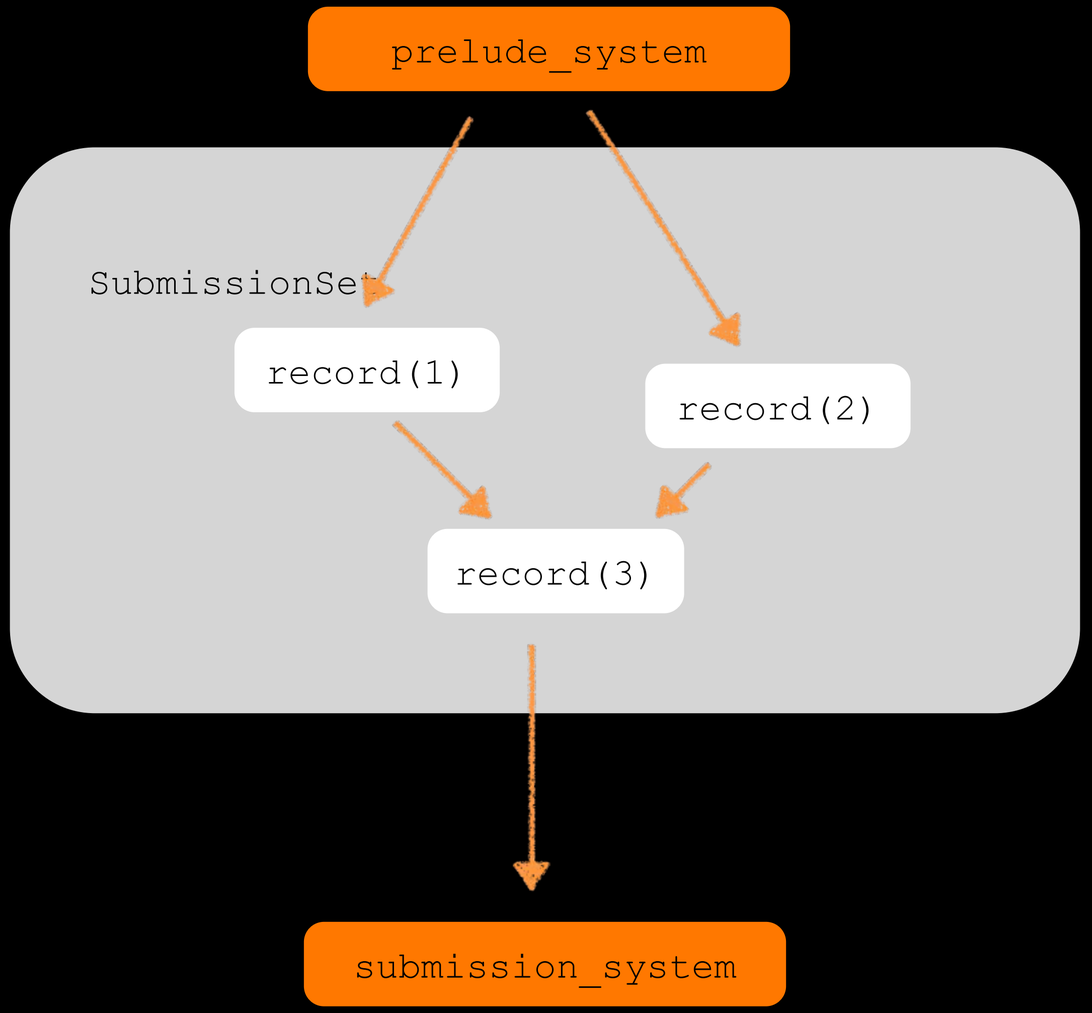
SubmissionSet and RenderSet

…and other free stuff we get from bevy

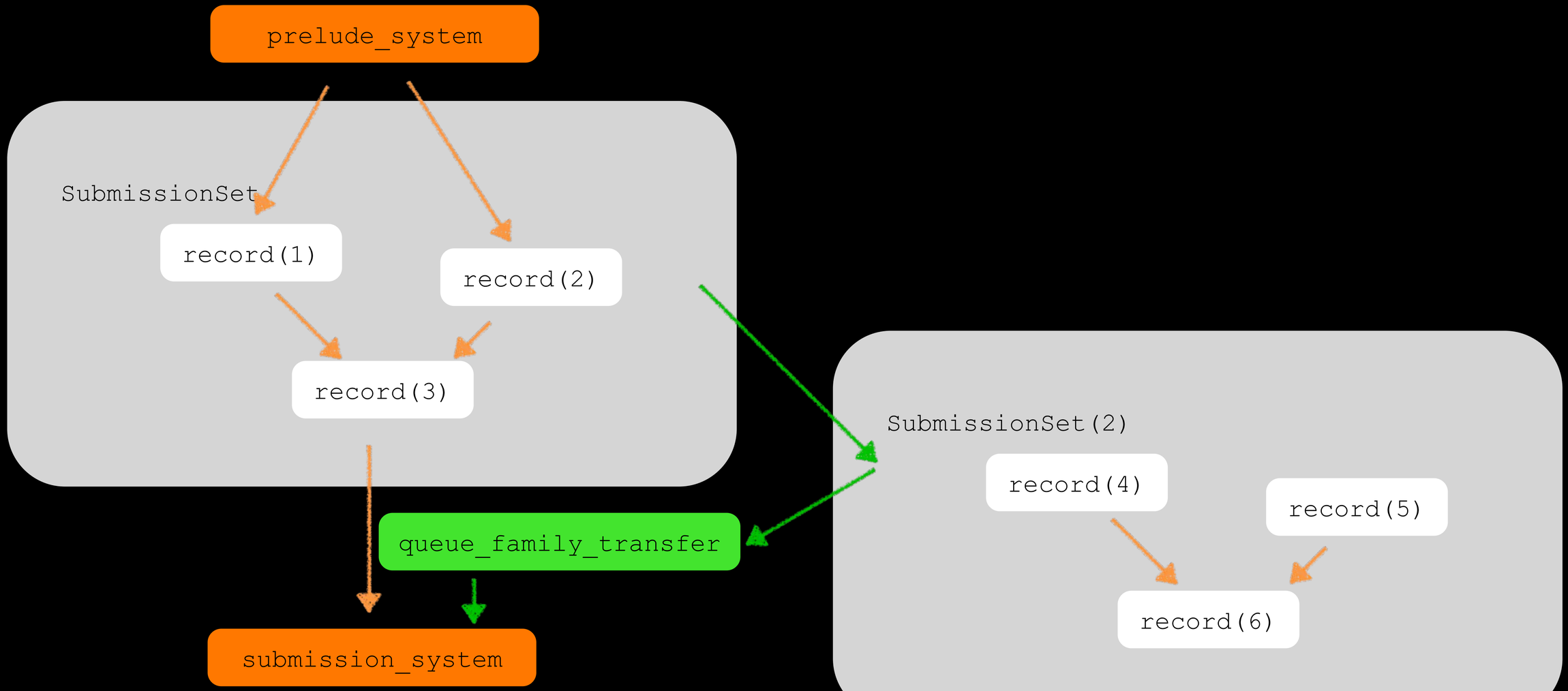
- SubmissionSet -> vkQueueSubmit
- RenderSet -> vkCmdBeginRendering
- Systems serialized in a set, share command pools, record to the same command buffer
- Parallel command buffer recording between SubmissionSets
- Parallel submission between queues
- Queues are Resources; may alias

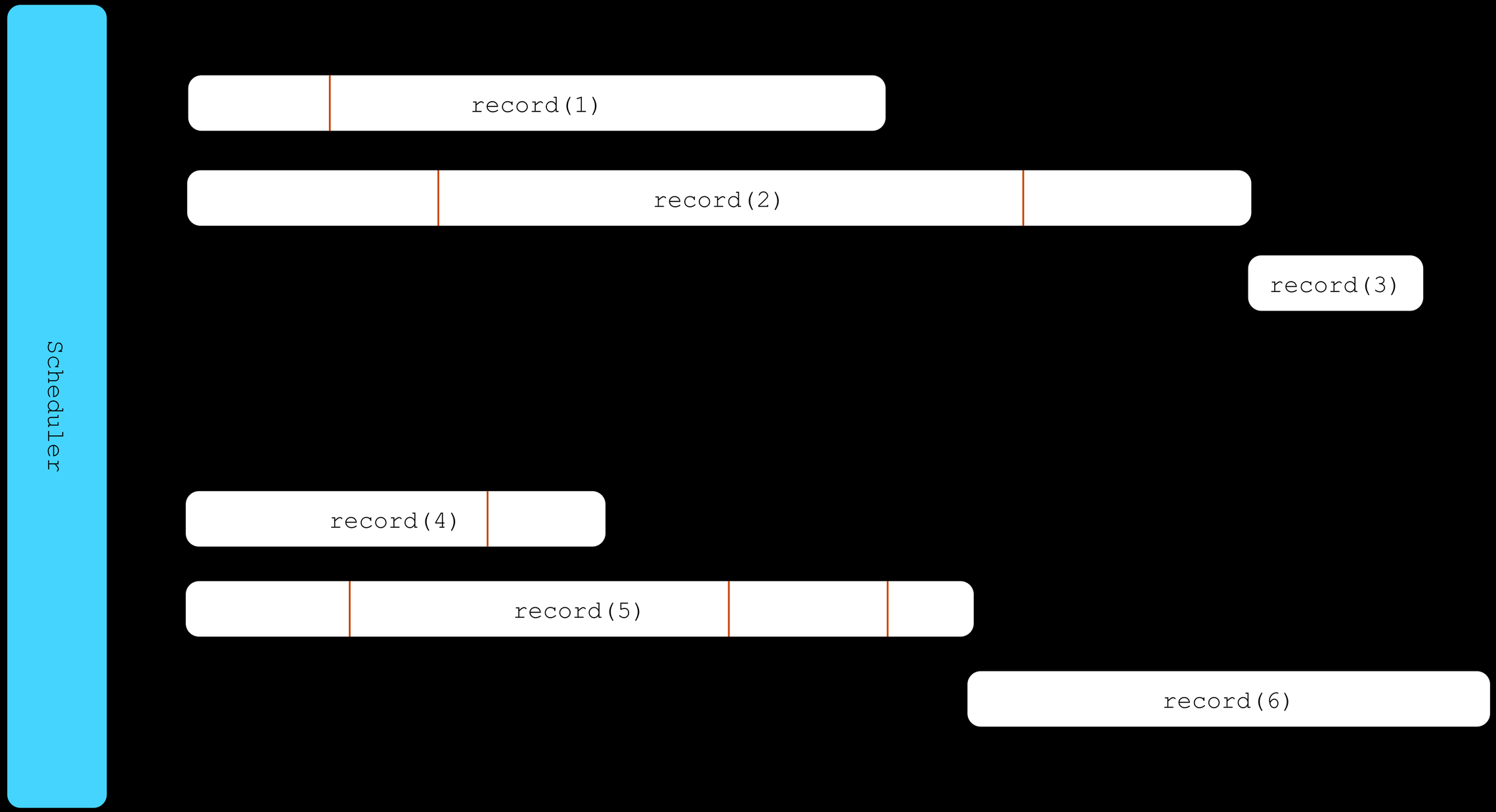
```rust
#[derive(SystemSet)]
pub struct PostProcessingSystemSet;
fn queue_system(queue: ResMut<RenderQueue>) {
    queue.bind_sparse( ... );
}struct MyPlugin;
impl Plugin for MyPlugin {
    fn build(&self, app: &mut App) {
        app.add_submission_set::<ComputeQueue>(
            PostProcessingSystemSet
          );


        app.add_systems(PostUpdate, (
            bloom,
            ambient_occlusion,
            lens_flare,
            auto_exposure,
            depth_of_field,
        ).in_set(PostProcessingSystemSet));
    }
}
```

ScheduleBuildPass



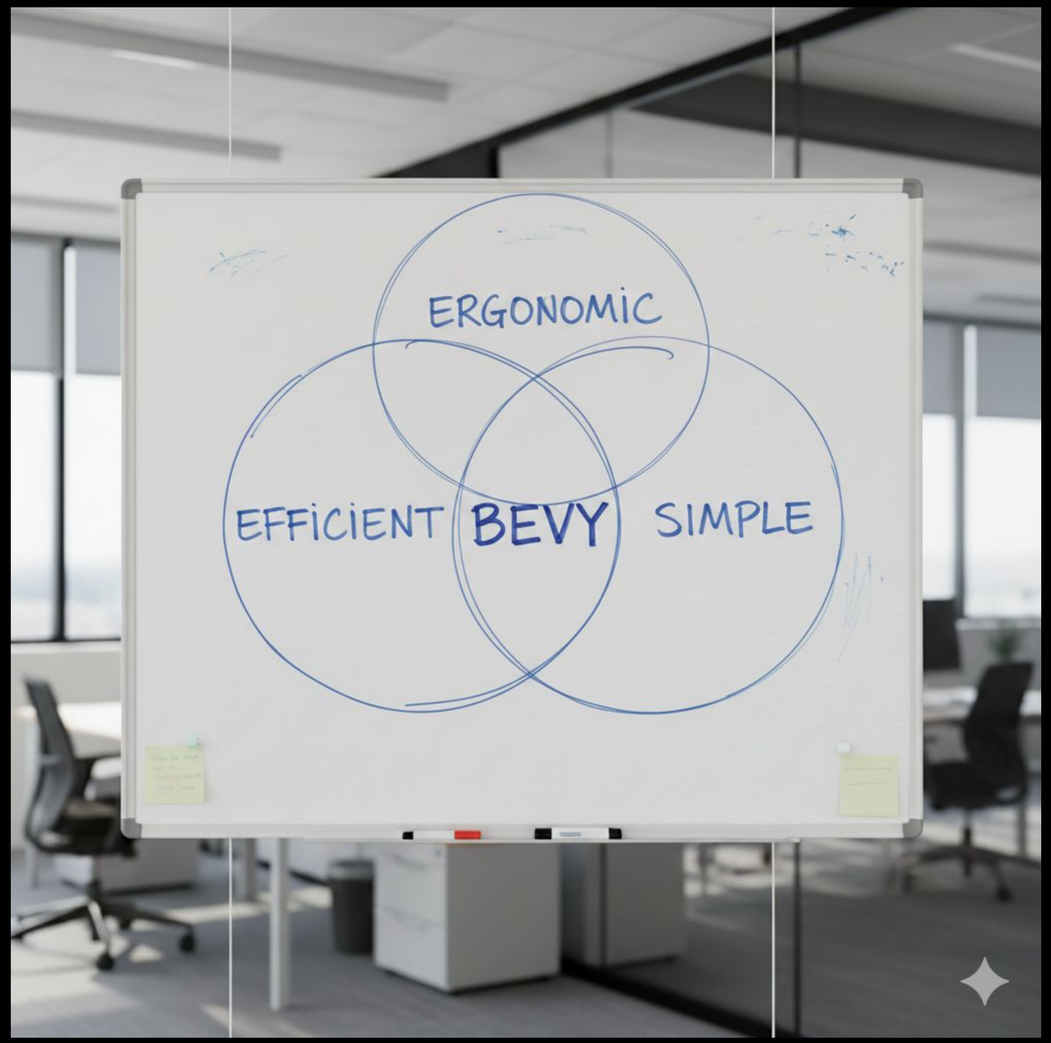
ScheduleBuildPass




Scheduler
record(1)
record(2)
record(3)
record(4)
record(5)
record(6)

Limitations (for now)

- Rust coroutine not stable yet; usability issue w.r.t lifetimes (#68923)
- Need to emulate coroutine with async await
- barrier merging within a single system for now


ERGONOMIC
EFFICIENT
BEVY
SIMPLE

Pumicite

https://github.com/dust-engine/pumicite

Thank you!