

Better render passes in PanVK

Faith Ekstrand
XDC 2026



COLLABORA

Open First

The story of PanVK

- Alyssa wrote an OpenGL driver
- Boris turned it into a Vulkan driver
- A bunch of code got extracted from GL and re-used
- The team worked on it for a few years
- We have Vulkan 1.4!



The problem:

**A lot of the shared components
were designed for OpenGL**



This includes `pan_fb_info`

The pan_fb_info design

- It's designed for OpenGL and is very immediate-mode
 - Describes the state as it is now, without looking to the future
 - This is fine for OpenGL because we don't know the future
 - We know if there **was** a clear but not if there **will be** a resolve
 - Vulkan is more declarative

The pan_fb_info design

- It's designed for OpenGL and is very immediate-mode
- Framebuffer attachments include everything:
 - The image to load from / store to
 - The load op and clear color, if any
 - A `.discard` flag on the off chance we don't store
 - This is very rare in OpenGL

The pan_fb_info design

- It's designed for OpenGL and is very immediate-mode
- Framebuffer attachments include everything
- Splitting the render pass is done by smashing the load op
 - First, force a spill
 - Smash the .preLoad bit to force a load on the next draw

The pan_fb_info design

- It's designed for OpenGL and is very immediate-mode
- Framebuffer attachments include everything
- Splitting the render pass is done by smashing the load op
- Multisampled is multisampled
 - There's no option for doing MSAA rendering and resolving as part of the store
 - OpenGL always needs the MSAA data to exist
 - There are tricks you can pull but they get hairy...

The pan_fb_info design

- It's designed for OpenGL and is very immediate-mode
- Framebuffer attachments include everything
- Splitting the render pass is done by smashing the load op
- Multisampled is multisampled
- Tile layout is directly tied to store images

The pan_fb_info design

- It's designed for OpenGL and is very immediate-mode
- Framebuffer attachments include everything
- Splitting the render pass is done by smashing the load op
- Multisampled is multisampled
- Tile layout is directly tied to store images
- No concept of render area



Case study:

Vulkan MSAA resolves

Case study: Vulkan MSAA resolves

- PanVK followed the OpenGL flow
 - MSAA data is written out for MSAA attachments
 - If requested, we do a Vulkan meta resolve
 - Samples from the MSAA attachment and does the average in a shader
 - For 16x MSAA, this is 16 load/store ops per pixel



Case study: Vulkan MSAA resolves

- PanVK followed the OpenGL flow
- Multi-sampled Vulkan apps can do better
 - Use a render pass resolve
 - Vulkan is declarative so the app can tell us up-front
 - Use `LOAD_OP_CLEAR`
 - This should let us avoid the tile load
 - Use `STORE_OP_DONT_CARE` for the MSAA attachment



Case study: Vulkan MSAA resolves

- PanVK followed the OpenGL flow
- Multi-sampled Vulkan apps can do better
- Mali is designed for this!
 - We can tell the store hardware to write out the single-sampled average
 - But only for blendable formats
 - We also have post-frame shaders
 - If we use the hardware properly, we do a single store per pixel

Case study: Vulkan MSAA resolves

- PanVK followed the OpenGL flow
- Multi-sampled Vulkan apps can do better
- Mali is designed for this!
- None of this is possible with `pan_fb_info`

Case study: Vulkan MSAA resolves

- PanVK followed the OpenGL flow
- Multi-sampled Vulkan apps can do better
- Mali is designed for this!
- None of this is possible with `pan_fb_info`
- But we left a **TODO** comment!



COLLABORA

Case study:

Render area clears

Case study: Render are clears

- pan_fb_info doesn't have a render area
- When the render area is the whole FB, we use tile clears
 - LOAD_OP_CLEAR turns into .clear
- When we have a partial render area:
 - We set .preload (our LOAD_OP_LOAD equivalent)
 - We immediately do a draw which clears just the render area
- Most of the loaded data is thrown away



Case study:

Multisample render to single sample

Case study: MSRTSS

- This extension is literally built for this hardware
 - Single-sample data is loaded into the MSAA temporary attachment
 - Drawing happens with MSAA
 - MSAA data is automatically resolved at the end of the render pass
 - MSAA data is discarded, only the single-sampled version persists
- But we were literally doing that. Every step.
 - 1x load + 16x store + 16x load + 1x store
 - Should be 1x load + 1x store



How do we do better?



A new framebuffer abstraction: `pan_fb_layout` and friends

A new framebuffer abstraction

Split into three pieces:

- `pan_fb_layout` describes the tile buffer
 - This only contains the formats of attachments
 - It knows **nothing** about the load/store images
 - Has a single sample count for the framebuffer
 - Has a render area and a tiling area
 - The tiling area is just render area aligned to tiles



A new framebuffer abstraction

Split into three pieces:

- `pan_fb_layout` describes the tile buffer
- `pan_fb_load` describes the load/clear at the start
 - Specifies a load op: image, clear or none (don't care)
 - Load ops are specified separately for in-bounds and border pixels
 - Specifies a load image (if either load op is image)
 - Specifies the clear value, if any



A new framebuffer abstraction

Split into three pieces:

- `pan_fb_layout` describes the tile buffer
- `pan_fb_load` describes the load/clear at the start
- `pan_fb_store` describes the store at the end
 - Specifies a store op: store or none
 - Specifies a store image (if the store op is store)
 - Specifies an optional MSAA resolve op



A new framebuffer abstraction

Split into three pieces:

- `pan_fb_layout` describes the tile buffer
- `pan_fb_load` describes the load/clear at the start
- `pan_fb_store` describes the store at the end

The three are combined to generate a HW framebuffer descriptor.



A new framebuffer abstraction

- The big change is that you can mix and match!
 - We only require that the load and store both be compatible with the layout
 - There's no requirement that they use the same images

A new framebuffer abstraction

- The big change is that you can mix and match!
- Render pass splitting is handled by changing load/store:
 - Unsplit render passes use load/store from the API
 - Split render passes use a spill sequence
 - First pass: API load, scratch store
 - Middle passes: scratch load, scratch store
 - Last pass: scratch load, API store
 - API load/store can be whatever, as long as we have scratch memory somewhere

A new framebuffer abstraction

- The big change is that you can mix and match!
- Render pass splitting is handled by changing load/store:
- Load now handles render areas
 - The hardware always clears before running the load shader
 - The render area is passed into the load shader as a push constant
 - For any part that isn't an image load, we discard
 - This lets us load border pixels but clear the render area with minimal bandwidth

A new framebuffer abstraction

- The big change is that you can mix and match!
- Render pass splitting is handled by changing load/store:
- Load now handles render areas
- We can now do an MSAA resolve as part of the store
 - We couldn't before because load/store/scratch images were tied
 - Now load, store, and scratch are all separate
 - Scratch can be MSAA while store is single-sampled



It gets better!

Enter `pan_fb_resolve...`

Framebuffer resolve shaders

- MSAA resolves as part of the store are limited
 - For color, only blendable formats are allowed
 - Depth and stencil only support sample-zero mode

Framebuffer resolve shaders

- MSAA resolves as part of the store are limited
- `pan_fb_resolve` lets us do better:
 - Specifies a full parallel copy of render targets
 - Can also specify image loads
 - Everything is specified separately for in-bounds and border pixels
 - Supports every MSAA resolve op Vulkan knows
 - sample-zero, average, min, max
 - It's a shader so we can do anything

Framebuffer resolve shaders

- MSAA resolves as part of the store are limited
- `pan_fb_resolve` lets us do better
- Lets us do render pass resolves with render area
 - The hardware writes whole tiles, which may go outside the resolve area
 - In the resolve, border pixels are set to load from the MSAA resolve target
 - In-bounds pixels copy from the render target with a resolve
 - When stored back to the resolve target, border pixels are unchanged

Framebuffer resolve shaders

- MSAA resolves as part of the store are limited
- `pan_fb_resolve` lets us do better
- Lets us do render pass resolves with render area
- The `pan_fb_resolve` API is fixed-function
 - Drivers don't have to think about generating shaders
 - They just call the shader builder, same as for load shaders

Framebuffer resolve shaders

- MSAA resolves as part of the store are limited
- `pan_fb_resolve` lets us do better
- Lets us do render pass resolves with render area
- The `pan_fb_resolve` API is fixed-function
- We can sometimes optimize it away
 - `pan_fb_fold_resolve_into_store()` tries to fold it all away



Results

Results

With the Sascha Willems multisampling example:

- With 2x MSAA: 590 -> 2605 (4.4x speedup)
- With 4x MSAA: 347 -> 2570 (7.4x speedup)
- With 8x MSAA: 188 FPS -> 2494 FPS (13.2x speedup)
- With 16x MSAA: 96.7 FPS -> 2483 FPS (25.7x speedup)

MSAA is now “free” (10% difference between 2x and 16x)



Thank you!



COLLABORA

Open First



We are hiring
col.la/careers



COLLABORA

Open First