arm

Understanding Render Passes

Version 1.0

Non-Confidential

Issue 02
102479_0100_02_en

Understanding Render Passes

Release information

Document history

Issue	Date	Confidentiality	Change
0100-02	22 March 2021	Non-Confidential	Initial release

Proprietary Notice

Confidentiality Status

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview

This guide discusses how render passes apply to Mali's tile-based GPU architecture via different Application Programming Interfaces (APIs), such as Vulkan and OpenGL ES.

The guide is useful to application developers new to this topic, or to those who want to build on a basic understanding of how render passes apply to Mali GPUs.

By the end of this guide, you will have a good understanding of how a Mali GPU interacts with the DRAM, and how it works between different APIs. You will also have learned about some of the key issues to consider when improving your application's rendering efficiency.

Finally, the guide will show you how to enable 4x Multi-Sample Anti-Aliasing (MSAA) with minimal impact to your application's performance.

2. How Render Passes Work

In simple terms, a render pass is a single execution of the rendering pipeline. A render pass renders an output image into a set of framebuffer attachments in memory. Each attachment requires initialization in tile memory at the start of the render pass and may then need to be written back out to memory at the end.

Some attachments are likely to be transient. For example, an application that uses both color and depth attachments during the render, but the application only needs to keep the color attachment for use in later rendering operations. In this scenario depth is transient and can simply be discarded at the end of the render pass.

Because of the need to minimize both the start and end of pass overheads, render passes are essential for tile-based renders. However, not all APIs natively support render passes due to how these APIs have evolved. Because of this, it is important to understand what we mean by a render pass, and how to use the APIs to construct them.

Render Passes in Vulkan

Vulkan has added explicit support for render passes in the API with the `VkRenderPass` structure, and for the individual framebuffer attachments the render pass contains via the `VkAttachmentDescription` structure.

Each attachment description must specify explicit `loadOp` operations to perform at the start of the render pass, and `storeOp` operations to perform at the end of the render pass. Therefore, the API requires a clear statement of intent to be added by the application developer.

Render Passes in OpenGL

Unlike Vulkan, the older OpenGL ES API has no explicit render passes in the API, so the driver must infer which rendering operations form a single render pass.

For Mali GPUs, a render pass is submitted for processing when an API call changes the framebuffer or forces a flush of the queued work. The most common causes for ending a render pass are:

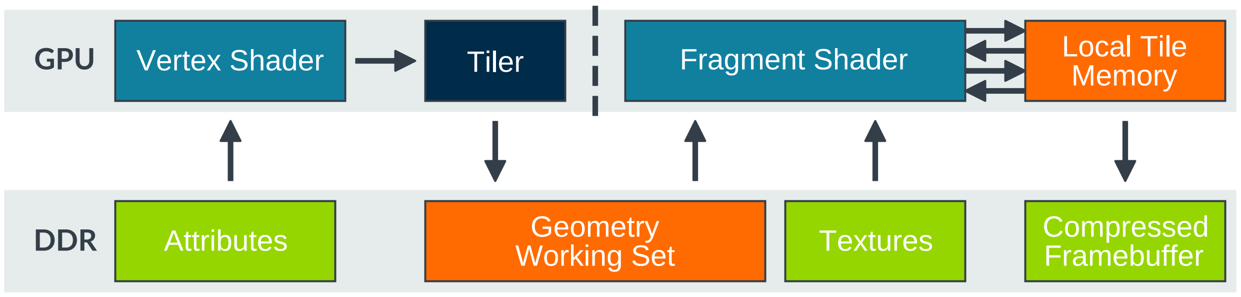
- The application calls `glBindFramebuffer()` to change the `GL_FRAMEBUFFER` or `GL_DRAW_FRAMEBUFFER` target.
- The application calls `glFramebufferTexture*()` or `glFramebufferRenderbuffer()` to change the attachments of the currently bound draw framebuffer object when the drawing is queued.
- The application calls `eglSwapBuffers()` to signal the end of a frame.
- The application calls `glFlush()` or `glFinish()` to explicitly flush any queued rendering.
- The application creates a `glFenceSync()` for some rendering in the current render pass and then calls `glClientWaitSync()` to wait on the completion of that work, or an equivalent behavior with a query object.

3. Mali GPUs and Tile Rendering

Mali GPUs process framebuffers as a series of small sub-regions called tiles. During fragment shading, the framebuffer working set for each tile is kept inside the GPU in a memory which is tightly coupled to each shader core. Keeping the memory inside the GPU minimizes the number of external DRAM accesses the GPU needs.

This image shows the data flow for a tile-based renderer:

Figure 3-1: Hardware flow



To get the most benefit from tile-based renderer, you must minimize the amount of memory traffic in and out of the tile memory. Too many reads or writes that access external memory can reduce the benefits of this approach. Specifically, this means avoiding:

- Reading in older framebuffer values at the start of a render pass if they are going to be overdrawn.
- Writing out values at the end of each render pass which are transient and are only needed for the duration of that render pass.

4. Efficient Render Passes

To get the best performance for each render pass, it is important to follow these basic steps to remove any redundant memory accesses:

First, make sure that each logical render pass in the application only turns into a single physical render pass when submitted to the hardware. Therefore, you should bind each framebuffer object only once and then make all required draw calls before switching to the next framebuffer. This step is important for OpenGL ES, where render passes are inferred.

Secondly, minimize the number of render passes and then merge adjacent render passes where possible. For example, one common inefficiency is a render pass for a 3D render, followed immediately by a render pass that applies a 2D UI overlay over the top of it.

In most cases, the UI drawing can be applied directly on to the 3D render, merging two passes into a single render pass. This trick avoids one round-trip via memory.

Minimizing Start of Tile Loads

Mali GPUs can initialize the tile memory to a clear color value at the start of a render pass without having to read back the old framebuffer content from memory. Before making any draw calls, ensure that you clear or invalidate all attachments at the start of each render pass. Unless you are deliberately drawing on top of what was rendered in a previous frame.

For OpenGL ES, you can use any of these calls to prevent a start of tile read from memory:

- `glClear()`
- `glClearBuffer*()`
- `glInvalidateFramebuffer()`

These must clear the entire framebuffer, not just a sub-region of it.

Caution

Only the start of tile clear is free. Calling `glClear()` or `glClearBuffer*()` after the first draw call in a render pass is not free, and this results in a per-fragment clear shader.

For Vulkan, set the `loadOp` for each attachment to either of:

- `VK_ATTACHMENT_LOAD_OP_CLEAR`
- `VK_ATTACHMENT_LOAD_OP_DONT_CARE`

Caution

If you call `VkCmdClear*()` commands to clear an attachment, or manually use a shader to write a constant color, it results in a per-fragment clear shader. To benefit from the fast fixed-function tile initialization, it is much more efficient to use the render pass `loadOp` operations.

On a Mali GPU, there is no performance difference between a start-of-pass operation and a start-of-pass invalidate. Operations can have hardware performance costs with GPUs from other vendors.

If you plan to completely cover the screen in opaque primitives, and have no dependency on the starting value, we recommend that you use an invalidate operation instead of a clear operation.

Minimizing End of Tile Stores

After a tile has been completed it is written back to main memory. For many applications, some of the framebuffer attachments may be transient. Because they are transient, they do not need to be kept beyond the duration of the render pass. It is important that the driver is notified of what attachments can be safely discarded.

For OpenGL ES, you can notify the driver that an attachment is transient by marking the content as invalid using a call to `glInvalidateFramebuffer()` as the last draw call in the render pass.

Note

If you write applications using OpenGL ES 2.0, you must use `glDiscardFramebufferExt()` from the `[EXT_discard_framebuffer]` `[EXT_dfb]` extension.

For Vulkan, set the `storeOp` for each transient attachment to `VK_ATTACHMENT_STORE_OP_DONT_CARE`. For more efficiency, the application can even avoid allocating physical backing memory for transient attachments by allocating the backing memory using `VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT` and constructing the `VkImage` with `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT`.

Handling Packed Depth-Stencil

GPUs commonly allocate depth and stencil attachments together in memory using a packed pixel format, such as `D24S8`. Due to the packed nature of this format, you must read neither attachment during load, and write neither attachment during store to get bandwidth savings.

To reliably get the best performance, we recommend:

- If you only need a depth buffer, allocate a depth-only format such as `D24` or `D24X8`, and never attach a stencil attachment.
- If you only need a stencil buffer, allocate a stencil-only format such as `S8`, and never attach a depth attachment.
- If you use a packed depth-stencil attachment, always attach both attachments, clear both attachments on load, and invalidate both attachments on store.
- If you use a packed depth-stencil attachment and need to continue using one of the attachments in a later render pass, invalidate the other at the end of the render pass. This may allow some bandwidth savings when using framebuffer compression.

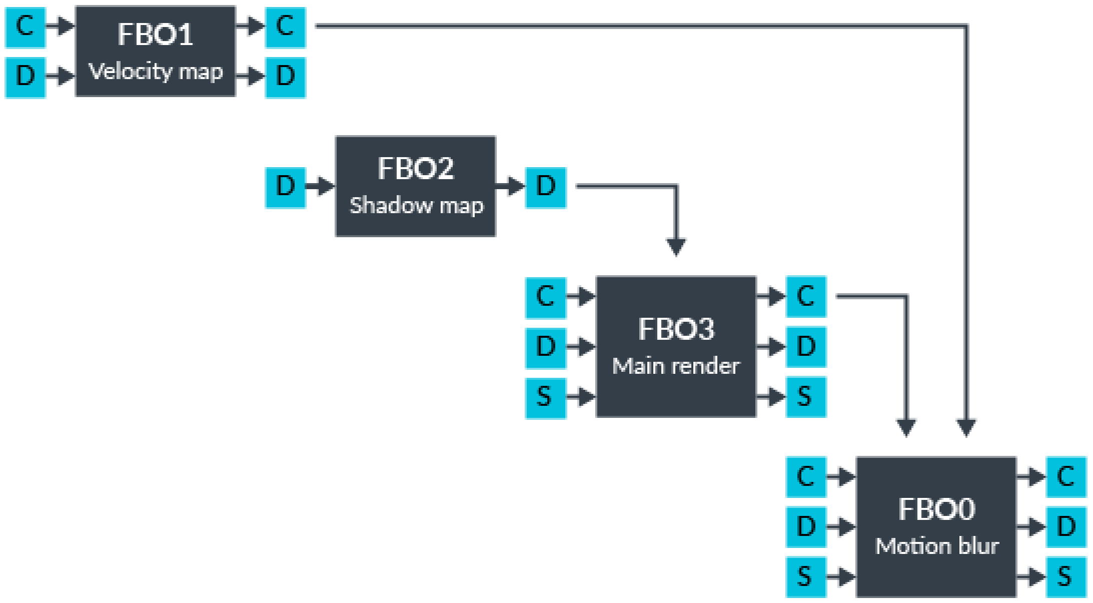
5. Worked Example

This example describes a 3D game rendering pipeline that builds a frame from four component render passes:

1. FBO1 is an off-screen pass that renders a velocity texture for motion blur. It uses color and depth attachments.
2. FBO2 is an off-screen pass that renders a depth shadow map for a dynamic light source. It uses only a depth attachment.
3. FBO3 is an off-screen pass that implements the main 3D render. It uses color, depth, and stencil attachments, and reads the output from FBO2 as an input texture.
4. FBO0 is the final window render which performs some 2D post-processing, combining the outputs from FBO1 and FBO3 as input textures to implement a motion blur effect.

The following render pass data flow diagram illustrates this rendering pipeline:

Figure 5-1: Basic pipeline



Each large box represents a single render pass. The smaller boxes to the left of each render pass represent input attachments in memory, and the smaller boxes to the right of each render pass represent the output attachments in memory.

Arrows that go in to the top of each render pass represent attachments rendered earlier that are then being used as textures in later passes.

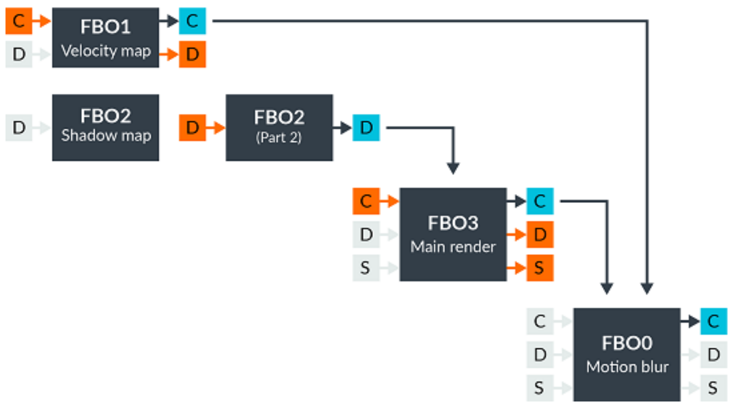
Inefficient Implementation

In practice, this abbreviated call sequence works, but it also breaks every efficiency rule at least once:

```
c
// Start rendering the off-screen shadow map pass
glBindFramebuffer(2);
glClear(GL_DEPTH_BUFFER_BIT);
glDrawElements(...);                 // Make some draws to FBO2
...

// Complete rendering the off-screen velocity pass
glBindFramebuffer(1);
glClear(GL_DEPTH_BUFFER_BIT);
glDrawElements(...);                 // Make all draws to FBO1
...

// Complete rendering the off-screen shadow map pass
glBindFramebuffer(2);
glDrawElements(...);                 // Make remaining draws to FBO2
...

// Complete rendering the off-screen main 3D pass
glBindFramebuffer(3);
glClear(GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
glDrawElements(...);                 // Make all draws to FBO3
...

// Complete rendering the window surface with motion blur
glBindFramebuffer(0);
glDrawElements(...);                 // Make all draws to FBO0
eglSwapBuffers();                    // Finish the frame
```

This is a render pass data flow diagram of an inefficient rendering pipeline:

Figure 5-2: Inefficient Timeline



Looking at this diagram, we can identify some issues. Each light grey arrow represents some bandwidth saved by a clear or an invalidate. However, each orange arrow represents some wasted bandwidth where a render pass reads, or writes, an attachment to main memory unnecessarily.

This happens because:

- The call sequence creates FBO2 as two hardware render passes because two distinct sets of API calls, separated by the rendering of FBO1, render it.

- FBO1 has redundant color read-backs because the call sequence does not the clear color attachment.
- FBO1 has redundant depth write-backs because the call sequence does not invalidate the depth attachment.
- FBO3 has redundant color read-backs because the call sequence does not clear the color attachment.
- FBO3 has redundant depth and stencil write-backs because the call sequence does not invalidate the attachments.

Note

Even though FBO0 does not have clears and invalidates, explicit calls are not necessary as the default behavior of EGL window surfaces is to implicitly clear all window surface attachments at the start of the render pass and then invalidate the depth and stencil at the end.

If we assume the application is rendering all passes at 1080p, with 32bpp for color (RGBA8), depth (D24X8), and depth-stencil (D24S8), the total wasted bandwidth is:

```
frameCost = x * y * bytes_per_pixel * (wasted_reads + wasted_writes)
          = 1920 * 1080 * (32 / 8) * (3 + 3)
          = 1920 * 1080 * 4 * 6
          = 49.7 MB
```

This is 2.99 GB/s at 60 FPS which is a significant amount of the system memory bandwidth and energy budget that is wasted on unnecessary memory traffic. Framebuffer compression and other GPU optimizations, such as hidden surface removal, can help reduce this.

However, it is more efficient if the application removes this redundancy to guarantee that it has no overhead.

Efficient Implementation

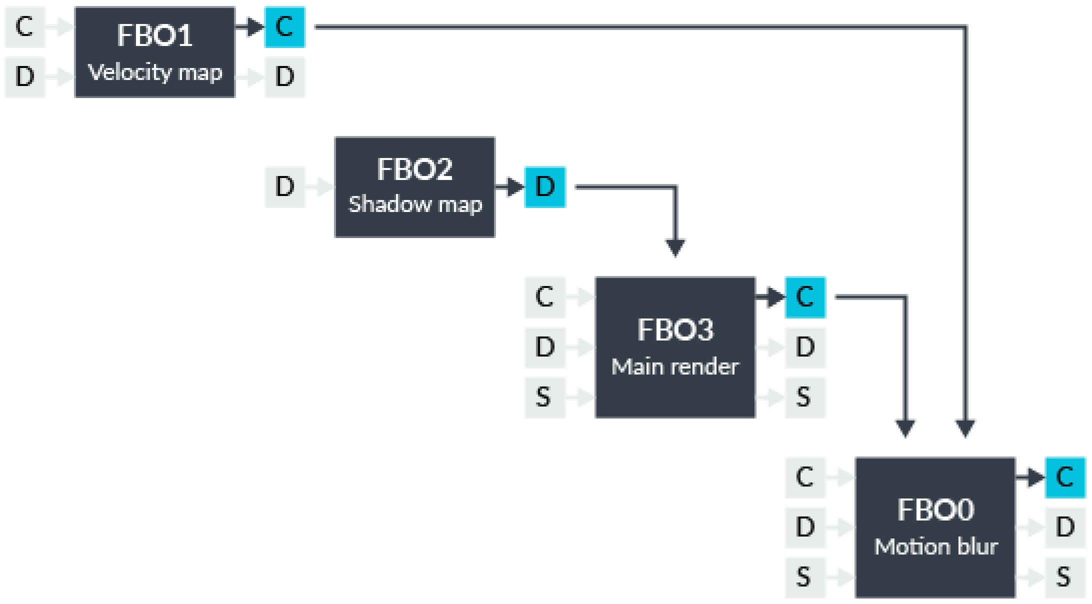
The following abbreviated call sequence implements the same rendering pipeline as the inefficient implementation. However, this time it follows our efficiency rules:

```
c
#define CLEAR_CDS (GL_COLOR_BUFFER_BIT | \
                   GL_DEPTH_BUFFER_BIT | \
                   GL_STENCIL_BUFFER_BIT)

static const GLenum INVALID_DS[2] = {
    GL_DEPTH_ATTACHMENT,
    GL_STENCIL_ATTACHMENT
};

// Complete rendering the off-screen shadow map pass
glBindFramebuffer(2);
glClear(CLEAR_CDS);                   // Clear all attachments
glDrawElements(...);                  // Make all draws to FBO2
...

// Complete rendering the off-screen velocity pass
glBindFramebuffer(1);
glClear(CLEAR_CDS);                   // Clear all attachments
```

```
glDrawElements(...);                 // Make all draws to FBO1
...
glInvalidateFramebuffer(GL_FRAMEBUFFER, 2, INVALID_DS)

// Complete rendering the off-screen 3D pass
glBindFramebuffer(3);
glClear(CLEAR_CDS);                  // Clear all attachments
glDrawElements(...);                 // Make all draws to FBO3
...
glInvalidateFramebuffer(GL_FRAMEBUFFER, 2, INVALID_DS);

// Complete rendering the window surface with motion blur
glBindFramebuffer(0);
glDrawElements(...);                 // Make all draws to FBO0
eglSwapBuffers();                    // Finish the frame
```

Looking at the render pass graph that this call sequence creates, the efficiency savings are clear to see when compared to the inefficient example:

Figure 5-3: Efficient Pipeline



Each light grey arrow represents some bandwidth saved by a clear or an invalidate, and compared to the inefficient example, there is no wasted memory bandwidth due to unnecessary tile loads or stores.

6. Multi-Sample Anti-Aliasing

Multi-Sample Anti-Aliasing (MSAA) is a low-cost approach for improving the quality of rendering by reducing the impact of jaggies along the edges of primitives. Jaggies are the result of aliasing due to pixel sampling of geometry.

MSAA is gaining importance on mobile devices due to the proliferation of augmented reality and virtual reality headsets. These headsets effectively make pixels larger because of the lenses scaling the screen to fill the field of vision.

MSAA works by using multiple samples-per-pixel for color and depth during the main rendering process, including storing these in the framebuffer. MSAA then reduces these samples to a single value per pixel once the render is completed. Mali GPUs are optimized for storing 4 samples per pixel, but higher MSAA levels are possible in some of the newer products for some additional performance cost.

A naive implementation of MSAA requires a sequence call that writes all the additional samples back to DRAM and then reads them back into the GPU where the GPU resolves them to a single value per pixel. This is a very expensive use of limited bandwidth and should therefore be avoided.

If we assume a 1440p panel rendering RGBA8 pixels at 60 FPS, which is a common VR configuration, then the bandwidth cost of these additional samples for 4xMSAA is:

```
python
bytesPerFrame4x = 2560 * 1440 * 4 * 4
bytesPerFrame1x = 2560 * 1440 * 4 * 1

# Additional 4x bandwidth is doubled because the additional samples
# are written by one pass and then re-read to resolve the final color
bytesPerFrame = ((bytesPerFrame4x * 2) + bytesPerFrame1x)
bytesPerSecond = bytesPerFrame * 60
               = 7.9 GB/s
```

In general, external DDR bandwidth costs 100mW per GB/s. Therefore, in our example, this overhead uses 790mW of our total 2.5 watt device power budget just to write the framebuffer.

One of the biggest benefits of a tile-based renderer is that the local memory can store all of the additional samples needed for MSAA during the render and resolve those back to a single color before the tile is written, so the bandwidth cost should be the same a single sampled framebuffer:

```
python
bytesPerFrame1x = 2560 * 1440 * 4 * 1

# All additional 4x bandwidth is kept entirely inside the tile memory
bytesPerSecond = bytesPerFrame1x * 60
               = 884 MB/s
```

This uses just 88mW of memory power. To benefit from this low-bandwidth inline resolve, the application has to opt-in and take explicit steps to use the technique.

For OpenGL ES, the application should use the Khronos EXT_msaa extension. This enables an implicit resolve at the end of a render pass rather than requiring a separate `glBlitFramebuffer()` resolve render pass.

For Vulkan, the render pass should provide single sampled `pResolveAttachments` to store the resolved data and set the `storeOp` for the multi-sampled attachments to `VK_ATTACHMENT_STORE_OP_DONT_CARE`. For additional efficiency the application can even avoid allocating physical backing memory for transient attachments by allocating the backing memory using `VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT` and constructing the `VkImage` with `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT`.

When you correctly keep all of the additional MSAA bandwidth inside the GPU, you can dramatically reduce the system bandwidth impact to just a ninth of that used by the naive implementation. This greatly improves both performance and energy.

7. Next Steps

In this article we have explored how you can efficiently construct render passes in your applications, regardless of whether you chose to use the Vulkan or OpenGL ES API, as well as how to implement a more efficient rendering pipeline.

Try enabling 4xMSAA on your application on a Mali GPU and test to see both the difference in visual quality, and the associated performance cost for enabling it.