The low-level mysteries of pipeline barriers

Frederic Garnier, Andrew Garrard

f.garnier@samsung.com  a.garrard@samsung.com
Overview

- Introduction
- What are barriers?
- How are barriers exposed in Vulkan?
- Case study and examples
Our partners

- Google Play
- EPIC Games
- FIRST TOUCH GAMES
- COCOS2D
- SQUARE ENIX
- SummerTime Studio
- netmarble Games
- NEXON
- DEEP SILVER
- FISH LABS
- Tencent Games
- 4:33 Creative Lab
- ACTION SQUARE
- NAT GAMES
- SUPER EVIL MEGACORP
- GAMEVIL
- XL GAMES
- HOUND 13
- COATSINK
- DIGITAL LEGENDS
- Vulkan
- Samsung
Who are we?

- Promote the use of Vulkan on Android
- Support game studios with issues on our devices, at a global scale
- Help game studios port their games to Vulkan
  - Performance, content tuning, DDK & platform support
Why do GPUs need barriers?
GPUs are highly parallel

- “Computer graphics is embarrassingly parallel”
- “In parallel computing, an **embarrassingly parallel** workload or problem [...] is one where *little or no effort* is needed to separate the problem into a number of parallel tasks.” - Wikipedia
  - Heh, heh, heh
- “All problems become scalar once you’ve thrown enough silicon at them”
  - - A. Garrard, 2018
GPUs are highly parallel

- Rasterisation is quite parallel
- Shading is parallel
- Memory access is parallel
  - Multiple usage-dependent caches and buffers
- All GPUs are parallel
  - But some are more than others - George Orwell
GPUs are hyperthreaded

- Graphics is dominated by memory access
  - Textures, frame buffers, vertex buffers
- Many threads let GPUs hide latency
  - ALUs are often quite deeply pipelined, but memory latency can be enormous
- Even more is in flight than the parallelism would suggest!
GPUs are heavily pipelined

- The “graphics pipeline” is actually a series of (mostly) parallel stages
- The pipeline can get backed up by slow components, so buffering is important to keep things flowing
- The GPU can be working on multiple elements at once within a stage
- Triangles covering a fragment may not finish shading in-order
Tiled GPUs are very heavily pipelined

- Tiled GPUs do all the rasterising for one tile independently of other tiles
- Fragments in one tile may get shaded before “earlier” primitives in another
- Rasterising may not be in the same frame as vertex shading
- Vertex shaders may be run repeatedly
- “How long did my shader take?” is a very complicated question

**Diagram:**

- Input assem.
- Vertex Shader
- TCS
- TES
- Geom.
- Raster
- Fragment
- Blend

- VB/IB
- Tile buffer(s)
- Texture
- Framebuffer
Even the memory is parallel

- GPUs depend heavily on caches
  - Random access to “fast” DRAM is much slower than you’d think
  - 1995: 100MHz SDRAM random access latency: ~20ns
  - 2018: DDR4-4800 random access latency: ~8ns (2.5x in 23 years)
- Textures might be in a special cache
- The frame buffer on a tiler may be a “special” cache
- The caches don’t necessarily snoop each other
Memory doesn’t look like reality

- Linear memory is really inefficient for texturing
  - Exact layout depends on cache size, ease of integration, and patents…
- “Layout” enables memory compression to save bandwidth
  - Lossless compression for the frame buffer, MS, depth buffer
  - Layout is usage-specific
  - The representation may not be consistent (e.g. NaNs)
  - Don’t confuse this with lossy texture compression (DXT, ETC, ASTC, PVRTC)
  - Barriers specify layout transitions
- The CPU’s view has to be non-proprietary
  - GPU vendors can’t expose details, because then they wouldn’t be able to change them
  - Still need a simple view for communication with the CPU
Magic just happens

- If *every* parallel operation needed manual sequencing, game developers would go mad*
  - (* more mad)
- The silicon designers get to go mad* instead
  - (* more mad)
- GPUs maintain the illusion primitives are rasterised in order
  - Typically blend units can sort out the mess
- ...so long as you’re only writing to the frame buffer
- Computer graphics is cheating and hoping no-one notices
  - Corollary: do what you like, but don’t get caught
  - - A. Garrard, 2018
There’s no such thing as magic

- Sometimes you do things the GPU can’t magic away
  - Abstracting away parallelism is easier in special cases than in general
  - Older APIs try to apply workarounds heuristically, which can cause unnecessary overhead

- Reading the frame buffer during rendering is hard
  - Requires pixel ordering guarantees, has representation issues

- Writing outside the framebuffer in any kind of shader is not strictly ordered automagically
  - E.g. intermediate outputs from vertex shaders

- Accessing the framebuffer other than the current pixel complicates tiling
GPGPU/Compute - more than pretty pictures

- Programmable GPUs used for more than just graphics since about 2001
- Custom compute shaders and APIs for many years
- More user control over read and write
- Much more requirement on the user to synchronise everything
Local dependencies

- Reads after framebuffer writes can have a framebuffer local dependency.
- In a tiler, this dependency can stay within the processing of the tile:
  - If you want to read anywhere in the framebuffer, you need the whole image to be rendered.
  - If you just want to read your current pixel, you can work within the current tile.
- Framebuffer local dependencies let you synchronise within the current tile processing rather than across the entire frame.
- Local processing can avoid a lot of unnecessary memory traffic.

![Diagram showing global and local access]

- Frame buffer memory (slow)
- On-chip tile memory (fast)

Global access

Local access
Subpass synchronisation - mostly magic

- Typical graphics usage pattern: writing one pixel to the frame buffer, then reading it back in a second subpass
  - Most common for deferred shading and programmable multisample resolves
  - Tilers can stay in tile memory for this
- Local dependencies get handled automatically
- Low synchronisation overhead
- Much less costly than a full framebuffer write to memory
  - FB writes appear much less costly on a desktop GPU
- This is why the subpass concept exists in Vulkan
- Only works for local access
  - Bear this in mind if you’re post-processing
Pipeline barriers

- Block operations after this point until operations before this point complete
- Dependencies apply to graphics pipeline stages
  - Block only the pipeline stages you need (especially on a tiler!)
- Dependencies can be framebuffer local, (but use subpasses!)
- Use between render passes (e.g. shadow map to main frame)
- Use to order compute operations
Events: sync while keeping things busy

- A simple barrier divides time into before and after
  - Work may have to stop and wait, stalling the GPU
  - But we wanted parallelism - how do we keep things going while we synchronise?
  - Doing a single barrier for multiple dependencies helps - but still stalls

- Events let you have multiple dependencies active
  - Wait only for the work you cared about, independent work can continue

- Events have a user-visible representation
  - The host CPU can access them too
  - Be careful not to make the GPU time out when doing this
Summary

- GPUs do lots of things at once
  - They get a lot slower if they can’t do this!
- Only the most basic ordering happens automatically
- Anything more complicated, you need to provide explicit synchronisation
- Use:
  - Subpass dependencies for local pixel framebuffer dependencies
  - Pipeline barriers to synchronise everything else
  - Events to keep the GPU busy while synchronising
- What’s actually going on may well be more complicated than this
- That’s the theory - how do you program it?
Frederic Garnier, Samsung Electronics

How are barriers exposed in Vulkan?
Disclaimer

- Results and our experience are based on Galaxy S7 to S9 devices using Arm Mali and Qualcomm GPUs
How are barriers exposed in Vulkan?

● Need to track resources in Vulkan and synchronise accordingly
  ○ Is the resource in the right state?
  ○ Are we writing to / reading to the resource in the correct order?
  ○ Have we taken care of execution and memory dependencies?

● Even lower-level…
  ○ Have we ensured that data is visible and available to the relevant stages?
void vkCmdPipelineBarrier(
    VkCommandBuffer          commandBuffer,
    VkPipelineStageFlags     srcStageMask,
    VkPipelineStageFlags     dstStageMask,
    VkDependencyFlags        dependencyFlags,
    uint32_t                 memoryBarrierCount,
    const VkMemoryBarrier*   pMemoryBarriers,
    uint32_t                 bufferMemoryBarrierCount,
    const VkBufferMemoryBarrier*  pBufferMemoryBarriers,
    uint32_t                     imageMemoryBarrierCount,
    const VkImageMemoryBarrier*   pImageMemoryBarriers);

typedef enum VkPipelineStageFlagBits {
    VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT = 0x00000001,
    VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT = 0x00000002,
    VK_PIPELINE_STAGE_VERTEX_INPUT_BIT = 0x00000004,
    VK_PIPELINE_STAGE_VERTEX_SHADER_BIT = 0x00000008,
    VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT = 0x00000010,
    VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT = 0x00000020,
    VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT = 0x00000040,
    VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT = 0x00000080,
    VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT = 0x00000100,
    VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT = 0x00000200,
    VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT = 0x00000400,
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT = 0x00000800,
    VK_PIPELINE_STAGE_TRANSFER_BIT = 0x00001000,
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT = 0x00002000,
    VK_PIPELINE_STAGE_HOST_BIT = 0x00004000,
    VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT = 0x00008000,
    VK_PIPELINE_STAGE_ALL_COMMANDS_BIT = 0x00010000,
} VkPipelineStageFlagBits;
How are barriers exposed in Vulkan?

```c
void vkCmdPipelineBarrier(
    VkCommandBuffer commandBuffer,
    VkPipelineStageFlags srcStageMask,
    VkPipelineStageFlags dstStageMask,
    VkDependencyFlags dependencyFlags,
    uint32_t memoryBarrierCount,
    const VkMemoryBarrier* pMemoryBarriers,
    uint32_t bufferMemoryBarrierCount,
    const VkBufferMemoryBarrier* pBufferMemoryBarriers,
    uint32_t imageMemoryBarrierCount,
    const VkImageMemoryBarrier* pImageMemoryBarriers);
```

typedef enum VkDependencyFlagBits {
    VK_DEPENDENCY_BY_REGION_BIT = 0x00000001;
} VkDependencyFlagBits;
How are barriers exposed in Vulkan?

```c
void vkCmdPipelineBarrier(
    VkCommandBuffer commandBuffer,
    VkPipelineStageFlags srcStageMask,
    VkPipelineStageFlags dstStageMask,
    VkDependencyFlags dependencyFlags,
    uint32_t memoryBarrierCount,
    const VkMemoryBarrier* pMemoryBarriers,
    uint32_t bufferMemoryBarrierCount,
    const VkBufferMemoryBarrier* pBufferMemoryBarriers,
    uint32_t imageMemoryBarrierCount,
    const VkImageMemoryBarrier* pImageMemoryBarriers);
```

typedef struct VkImageMemoryBarrier {
    VkStructureType sType;
    const void* pNext;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
    VkImageLayout oldLayout;
    VkImageLayout newLayout;
    uint32_t srcQueueFamilyIndex;
    uint32_t dstQueueFamilyIndex;
    VkImage image;
    VkImageSubresourceRange subresourceRange;
} VkImageMemoryBarrier;

How are barriers exposed in Vulkan?

```c
typedef struct VkImageMemoryBarrier {
    VkStructureType sType;
    const void* pNext;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
    VkImageLayout oldLayout;
    VkImageLayout newLayout;
    uint32_t srcQueueFamilyIndex;
    uint32_t dstQueueFamilyIndex;
    VkImage image;
    VkImageSubresourceRange subresourceRange;
} VkImageMemoryBarrier;
```

```c
typedef enum VkAccessFlagBits {
    VK_ACCESS_INDIRECT_COMMAND_READ_BIT = 0x00000001,
    VK_ACCESS_INDEX_READ_BIT = 0x00000002,
    VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT = 0x00000004,
    VK_ACCESS_UNIFORM_READ_BIT = 0x00000008,
    VK_ACCESS_INPUT_ATTACHMENT_READ_BIT = 0x00000010,
    VK_ACCESS_SHADER_READ_BIT = 0x00000020,
    VK_ACCESS_SHADER_WRITE_BIT = 0x00000040,
    VK_ACCESS_COLOR_ATTACHMENT_READ_BIT = 0x00000080,
    VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT = 0x00000100,
    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT = 0x00000200,
    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT = 0x00000400,
    VK_ACCESS_TRANSFER_READ_BIT = 0x00000800,
    VK_ACCESS_TRANSFER_WRITE_BIT = 0x00001000,
    VK_ACCESS_HOST_READ_BIT = 0x00002000,
    VK_ACCESS_HOST_WRITE_BIT = 0x00004000,
    VK_ACCESS_MEMORY_READ_BIT = 0x00008000,
    VK_ACCESS_MEMORY_WRITE_BIT = 0x00010000,
} VkAccessFlagBits;
```
How are barriers exposed in Vulkan?

typedef struct VkImageMemoryBarrier {
    VkStructureType sType;
    const void* pNext;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
    VkImageLayout oldLayout;
    VkImageLayout newLayout;
    uint32_t srcQueueFamilyIndex;
    uint32_t dstQueueFamilyIndex;
    VkImage image;
    VkImageSubresourceRange subresourceRange;
} VkImageMemoryBarrier;

typedef enum VkImageLayout {
    VK_IMAGE_LAYOUT_UNDEFINED = 0,
    VK_IMAGE_LAYOUT_GENERAL = 1,
    VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL = 2,
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL = 3,
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL = 4,
    VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL = 5,
    VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL = 6,
    VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL = 7,
    VK_IMAGE_LAYOUT_PREINITIALIZED = 8,
    VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL = 1000117000,
    VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL = 1000117001,
    VK_IMAGE_LAYOUT_PRESENT_SRC_KHR = 1000001002,
} VkImageLayout;
How are barriers exposed in Vulkan?

- Images in Vulkan are created with a tiling arrangement
  - Linear tiling
  - Optimal aka swizzled tiling
- Images that are created with optimal tiling require an explicit copy op
  - Possible to avoid this copy if using linear tiling mode
  - Useful if the texture is streamed in every frame...
- But images with linear tiling have a lot of limitations
  - No support for mipmaps
  - Only a few formats may be supported...
How are barriers exposed in Vulkan?

- Will go through pipeline barriers using the following example
  - Copy data to an optimal image from a buffer or a linear image that contains data...
  - Synchronize correctly to prepare the implementation for the copy operation...
  - Synchronize correctly to prepare the implementation for sampling the copied image...
void vkCmdPipelineBarrier(
    commandBuffer,
    VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,
    VK_PIPELINE_STAGE_TRANSFER_BIT,
    0,
    0
);

imageBarrier1 = {
    VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER,
    VK_NULL_HANDLE,
    0,
    VK_ACCESS_TRANSFER_WRITE_BIT,
    VK_IMAGE_LAYOUT_UNDEFINED,
    VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
    VK_QUEUE_FAMILY_IGNORED,
    VK_QUEUE_FAMILY_IGNORED,
    imageHandle,
    subResourcesRange;
};
Sampling data from a copied-to image

```c
void vkCmdPipelineBarrier(
    commandBuffer,
    VK_PIPELINE_STAGE_TRANSFER_BIT,
    VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,
    0,
    0,
    VK_NULL_HANDLE,
    0,
    VK_NULL_HANDLE,
    1,
    &imageBarrier2;
)
```

```c
imageBarrier2 = {
    VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER,
    VK_NULL_HANDLE,
    VK_ACCESS_TRANSFER_WRITE_BIT,
    VK_ACCESS_SHADER_READ_BIT,
    VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
    VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL,
    VK_QUEUE_FAMILY_IGNORED,
    VK_QUEUE_FAMILY_IGNORED,
    imageHandle,
    subResourcesRange;
};
```
Batch your pipeline barriers!

- Inserting a pipeline barrier (sync-point) within the command buffer has a CPU cost associated to it
  - Need to batch barriers as much as possible and flush at the right time!

```c
vkCmdPipelineBarrier(...,1,&barrier)
vkCmdPipelineBarrier(...,1,&barrier)
vkCmdPipelineBarrier(...,1,&barrier)
vkCmdPipelineBarrier(...,1,&barrier)
vkCmdPipelineBarrier(...,1,&barrier)
vkCmdPipelineBarrier(...,1,&barrier)
vkCmdPipelineBarrier(...,1,&barrier)
vkCmdPipelineBarrier(...,9, barriers.getData())
```
Subpass dependencies

- Images also need to be transitioned to the correct layout before presentation
- Preferably to transition as part of the render pass if possible
  - Can specify an image layout to use per-subpass and a final layout
  - Final layout is what the image transitions to at the end of the render pass
- Why not use a subpass dependency for the previous case?
  - Due to render pass scope .. copy command can only be called outside of a render pass instance
typedef struct VkAttachmentReference {
    uint32_t attachment;
    VkImageLayout layout;
} VkAttachmentReference;

typedef struct VkAttachmentDescription {
    VkAttachmentDescriptionFlags flags;
    VkFormat format;
    VkSampleCountFlagBits samples;
    VkAttachmentLoadOp loadOp;
    VkAttachmentStoreOp storeOp;
    VkAttachmentLoadOp stencilLoadOp;
    VkAttachmentStoreOp stencilStoreOp;
    VkImageLayout initialLayout;
    VkImageLayout finalLayout;
} VkAttachmentDescription;

typedef struct VkSubpassDependency {
    uint32_t srcSubpass;
    uint32_t dstSubpass;
    VkPipelineStageFlags srcStageMask;
    VkPipelineStageFlags dstStageMask;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
    VkDependencyFlags dependencyFlags;
} VkSubpassDependency;
Subpass dependencies

- Very simple example based on rendering a triangle or quad and presenting it
- Transition at the beginning of a render pass instance may happen out of order
  - Need to make sure presentation engine is done reading from the image
  - Subpass dependencies allow us to express execution and memory dependencies we need
  - Implicit subpass dependencies exist but not suitable for this use case
Subpass dependencies

- Transition the image when it can be rendered to..
  - i.e. when made available by semaphore & based on pWaitDstStageMask
- Not just limited to synchronising with presentation engine
  - Render passes can be used for off-screen rendering
  - Next one depends on previous one
  - Stage masks and access flags need to be set accordingly
Subpass dependencies - Implicit pre-dependency

colorAttachmentReference = {
    0,
    VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
};

attachmentDescription = {
    0,
    VK_FORMAT_R8G8B8_UNORM,
    VK_SAMPLE_COUNT_1_BIT,
    VK_ATTACHMENT_LOAD_OP_DONT_CARE,
    VK_ATTACHMENT_STORE_OP_STORE,
    VK_ATTACHMENT_LOAD_OP_DONT_CARE,
    VK_ATTACHMENT_STORE_OP_DONT_CARE,
    VK_IMAGE_LAYOUT_UNDEFINED,
    VK_IMAGE_LAYOUT_PRESENT_SRC_KHR
};

subpassDependency = {
    VK_SUBPASS_EXTERNAL,
    firstSubpass,
    VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,
    VK_PIPELINE_STAGE_ALL_COMMANDS_BIT,
    0,
    VK_ACCESS_INPUT_ATTACHMENT_READ_BIT |
    VK_ACCESS_COLOR_ATTACHMENT_READ_BIT |
    VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT |
    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT |
    VK_ACCESS_INPUT_ATTACHMENT_READ_BIT |
    VK_ACCESS_COLOR_ATTACHMENT_READ_BIT |
    VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT |
    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT,
    0
};
Subpass dependencies - Explicit dependency

```cpp
    colorAttachmentReference = {
        0,
        VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
    };

    attachmentDescription = {
        0,
        VK_FORMAT_R8G8B8_UNORM,
        VK_SAMPLE_COUNT_1_BIT,
        VK_ATTACHMENT_LOAD_OP_DONT_CARE,
        VK_ATTACHMENT_STORE_OP_STORE,
        VK_ATTACHMENT_LOAD_OP_DONT_CARE,
        VK_ATTACHMENT_STORE_OP_DONT_CARE,
        VK_IMAGE_LAYOUT_UNDEFINED,
        VK_IMAGE_LAYOUT_PRESENT_SRC_KHR
    };

    subpassDependency = {
        VK_SUBPASS_EXTERNAL,
        0,
        VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT,
        VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT,
        0,
        VK_ACCESS_COLOR_ATTACHMENT_READ_BIT |
        VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
        0
    };
```
Subpass dependencies

- Need a final layout transition to prepare for presentation...
  - Images need to be in `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR` layout
- An implicit post-render pass dependency also exists
  - We don’t need to explicitly define this
  - Defines that the transition happens after all work is done aka bottom of pipe
  - Semaphore guarantees execution dependency for us...
colorAttachmentReference = {
    0,
    VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
};

attachmentDescription = {
    0,
    VK_FORMAT_R8G8B8_UNORM,
    VK_SAMPLE_COUNT_1_BIT,
    VK_ATTACHMENT_LOAD_OP_DONT_CARE,
    VK_ATTACHMENT_STORE_OP_STORE,
    VK_ATTACHMENT_LOAD_OP_DONT_CARE,
    VK_ATTACHMENT_STORE_OP_DONT_CARE,
    VK_IMAGE_LAYOUT_UNDEFINED,
    VK_IMAGE_LAYOUT_PRESENT_SRC_KHR
};

subpassDependency = {
    lastSubpass,
    VK_SUBPASS_EXTERNAL,
    VK_PIPELINE_STAGE_ALL_COMMANDS_BIT,
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,
    VK_ACCESS_INPUT_ATTACHMENT_READ_BIT |
    VK_ACCESS_COLOR_ATTACHMENT_READ_BIT |
    VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT |
    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT |
    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT |
    0,
    0
};
Subpass dependencies - Explicit dependency

colorAttachmentReference = {
    0,
    VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
};

attachmentDescription = {
    0,
    VK_FORMAT_R8G8B8_UNORM,
    VK_SAMPLE_COUNT_1_BIT,
    VK_ATTACHMENT_LOAD_OP_DONT_CARE,
    VK_ATTACHMENT_STORE_OP_STORE,
    VK_ATTACHMENT_LOAD_OP_DONT_CARE,
    VK_ATTACHMENT_STORE_OP_DONT_CARE,
    VK_IMAGE_LAYOUT_UNDEFINED,
    VK_IMAGE_LAYOUT_PRESENT_SRC_KHR
};

subpassDependency = {
    VK_SUBPASS_EXTERNAL,
    0,
    VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT,
    VK_PIPELINE_BOTTOM_OF_PIPE_BIT,
    VK_ACCESS_COLOR_ATTACHMENT_READ_BIT |
    VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
    0,
    0
};
Case Study / Examples

- Transitioning image to a readable state… but using wrong stages
  - $srcStageMask = \text{FRAGMENT\_SHADER\_BIT}$
  - $dstStageMask = \text{VERTEX\_SHADER\_BIT} | \text{VERTEX\_INPUT\_BIT}$
Case Study / Examples

- Transitioning image to a readable state… but using wrong stages
  - $\text{srcStageMask} = \text{COLOR\_ATTACHMENT\_OUTPUT\_BIT}$
  - $\text{dstStageMask} = \text{FRAGMENT\_SHADER\_BIT}$
Case Study / Examples

L2R GLES build - 20.1ms frame time

L2R Vulkan (incorrect barrier stages) - 24.5ms (+3.4)

L2R Vulkan (incorrect barrier stages) - 12.4ms (-8.3)
Case Study / Examples

Unity Original GLES - 14.9ms

Unity VK - Using ALL_GRAPHICS_BIT - 13.7ms (-1.2)

Unity VK - Using optimized barriers + batching (-2.9)
Case Study / Examples

Original build - 36ms frame time

Optimized build - 25ms (-11) frame time
Frederic Garnier, Andrew Garrard
Galaxy GameDev, Samsung Electronics

http://developer.samsung.com/game

Thank you!

f.garnier@samsung.com
a.garrard@samsung.com
gamedev@samsung.com