Explicit graphics APIs

Corentin Wallez, Google
cwallez@google.com

What will presented

- Rationale for explicit APIs
- New concepts to expose old functionality
- New concepts to expose new functionality

Mostly about Vulkan, bridges to D3D12

Gross simplifications of how the hardware works

What will skipped

- Non-essential concepts
    - Only useful for X% performance gains
    - Trivial to understand
- The detail of the explicit API footguns

Why explicit APIs?

Texture resizing in OpenGL

User resizing texture:

- Resize the texture
- Use it
- :D

Driver resizing texture:

- Allocate new memory
- Use new memory
- :D

Texture resizing in OpenGL

User resizing texture:

- Resize the texture
- Use it
- :D

Driver resizing texture:

- Allocate new memory
  - Insert fence
  - Check the fence every frame?
  - Garbage collect memory
- Use new memory
- :/

Texture resizing in OpenGL

User resizing texture:

- Resize the texture
- Use it
- :D

Driver resizing texture:

- Allocate new memory
- Insert fence
- Check the fence every frame?
- Garbage collect memory
- Dirty uniforms passed to shaders
- Dirty framebuffers
- Dirty texture buffers
- Use new memory
- :(

Why: Predictable behavior and performance

Applications can:

- Control when expensive operations happen
- Have low variance frame timing (VR)
- Be smarter than the OpenGL driver

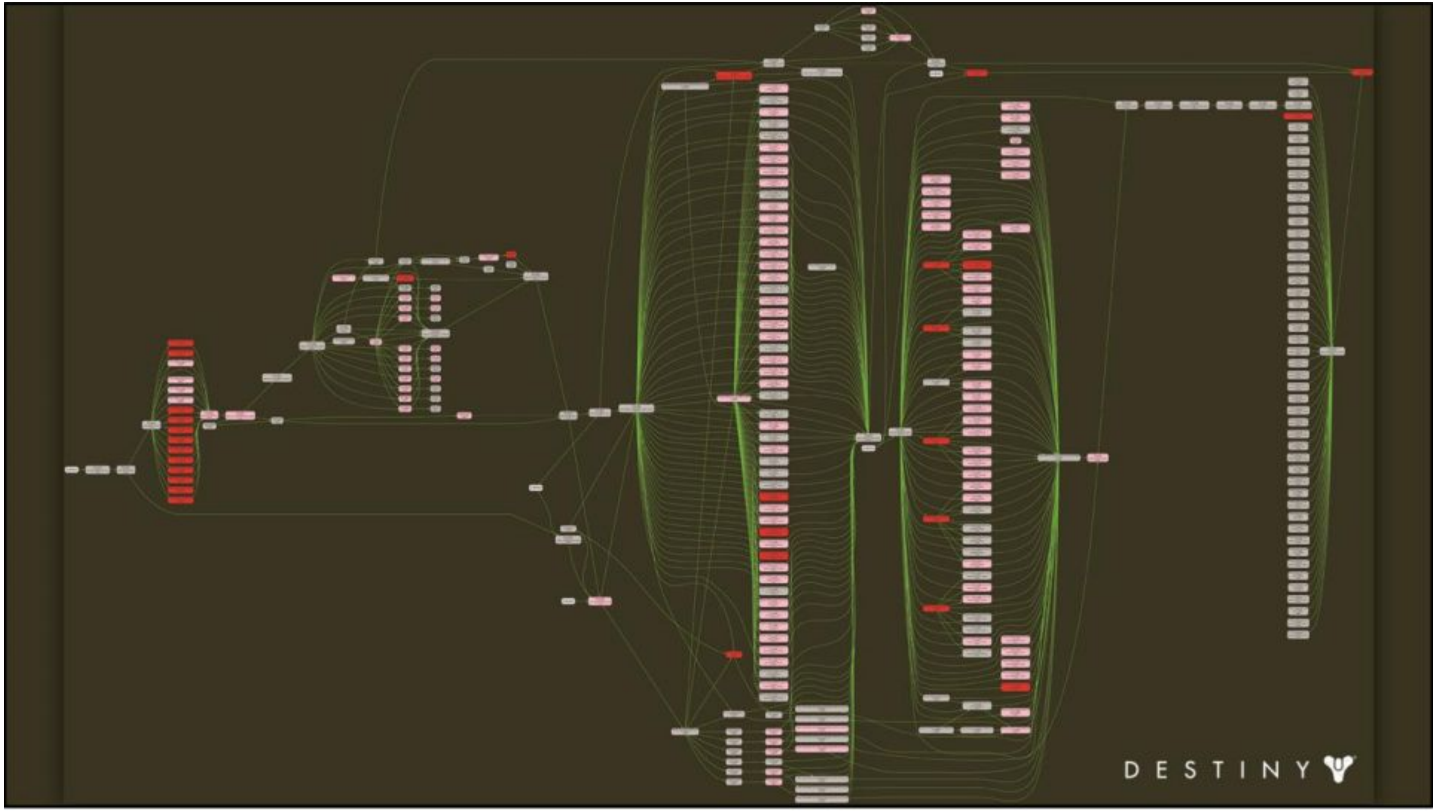
Why: Consoles

Graphics development on console:

- Direct access to the hardware
- Manual memory management
- Getting to that last 1% of performance
- Multithreading

Developers want that on PC too.

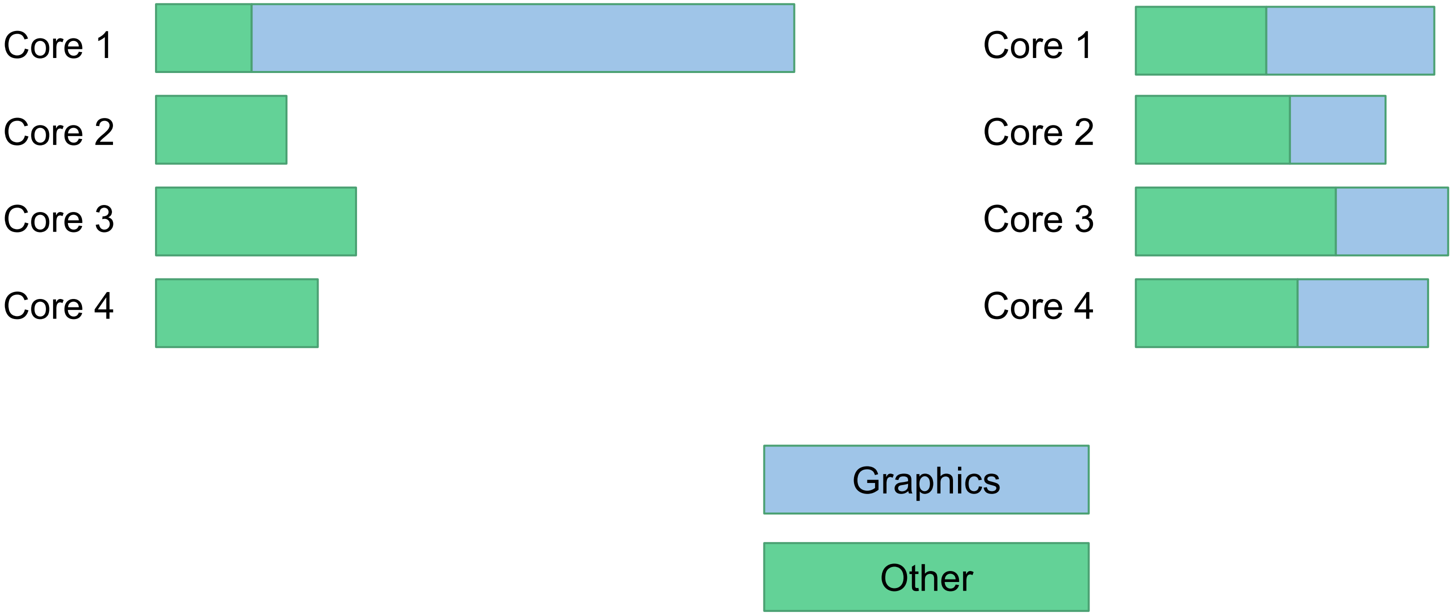
Why: Multithreading



Why: Multithreading

Single-threaded APIs

Multi-threaded APIs



Core 1

Core 2

Core 3

Core 4

Core 1

Core 2

Core 3

Core 4

Graphics

Other

Disadvantages of explicit APIs

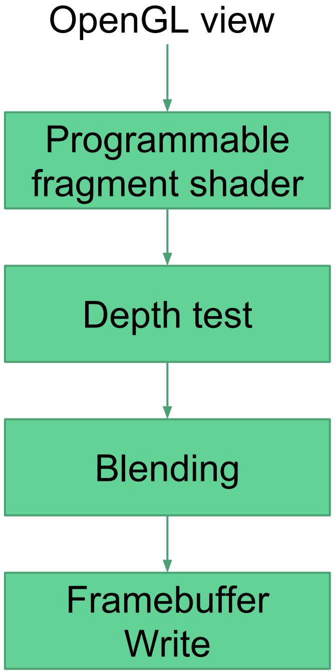
- Hard to understand: require deep knowledge of GPUs
- Hard to use: application is a driver
- Hard to use portably: application is a portable driver
- Hard to not explode: incorrect usage is a UB

Keep sanity by using validation layers on multiple hardware.

New concepts for old features

Pipelines

Pipelines



OpenGL view

Programmable fragment shader

glUseProgram

Depth test

glDepthFunc

Blending

glBlendFunc

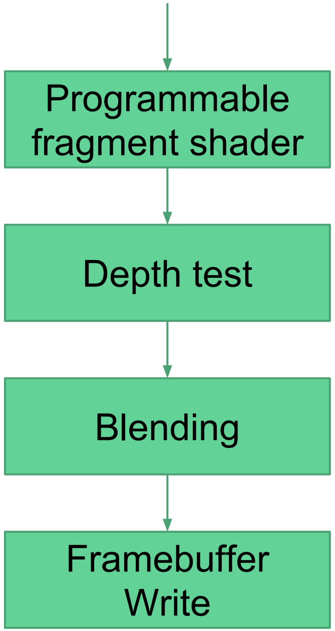
Framebuffer Write

glDrawBuffers

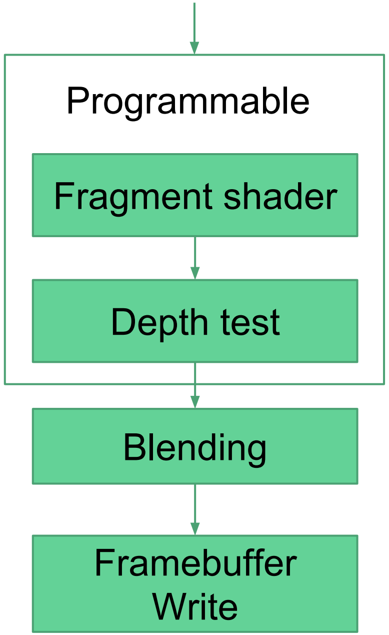
```
void main() {
    // stuff
}
```

Pipelines

OpenGL view



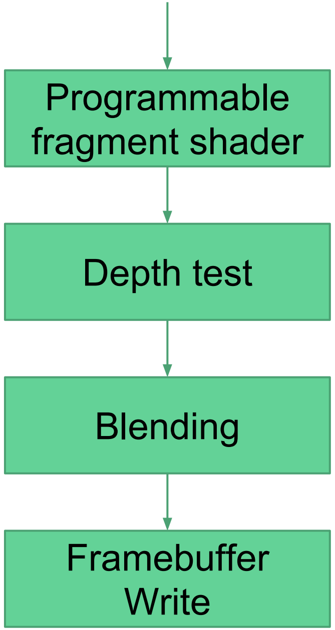
Actual GPU



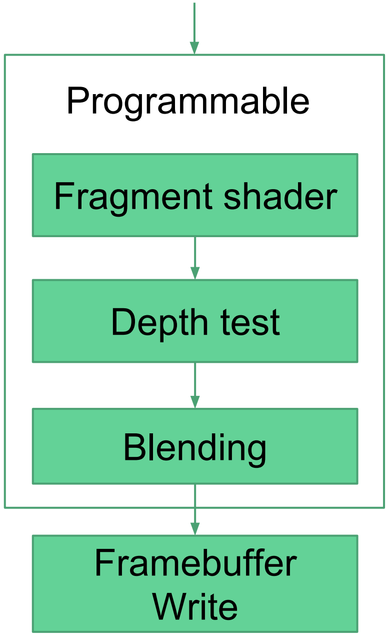
```
void main() {
    // stuff
}

void _start() {
    main();
    if (gl_FragDepth < _lastDepth) {
        discard;
    }
}
```

Pipelines

OpenGL view

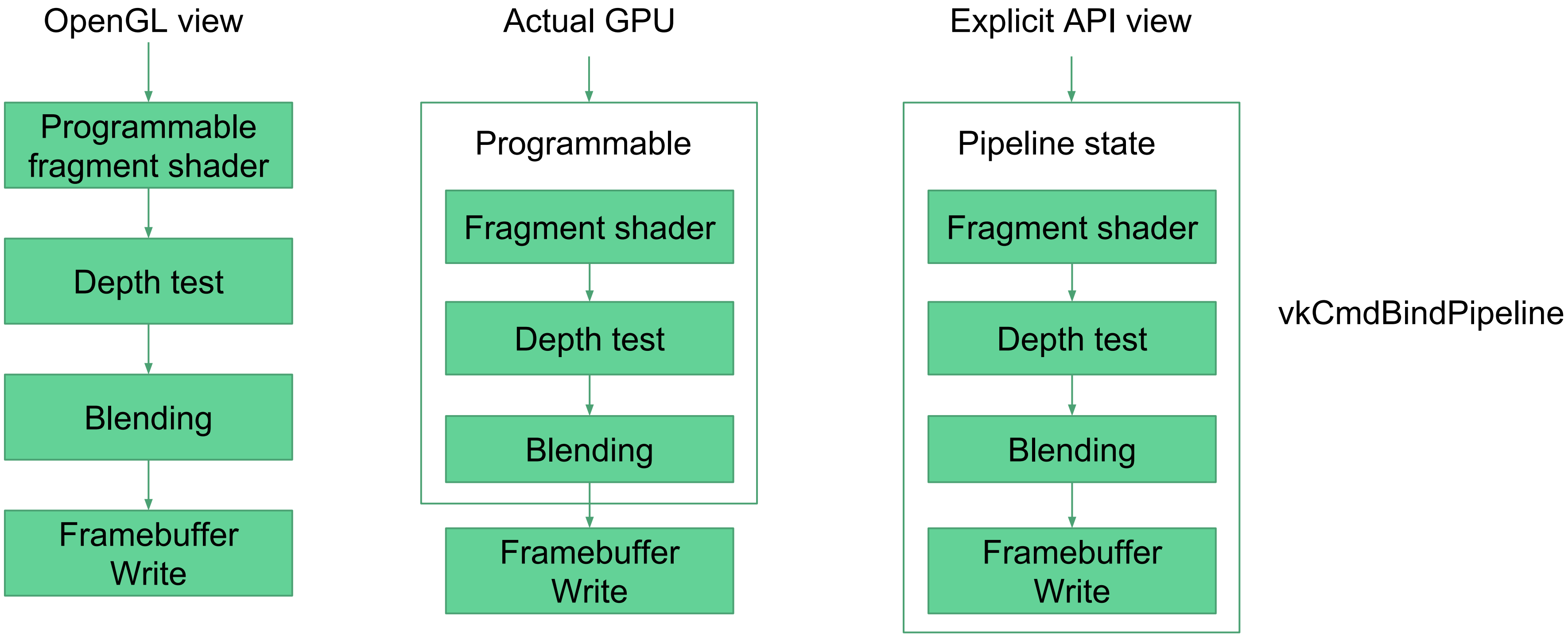


Actual GPU



```
void main() {
    // stuff
}

void _start() {
    main();
    if (gl_FragDepth < _lastDepth) {
        discard;
    }
    gl_FragColor =
        blend(gl_FragColor, _lastColor);
}
```

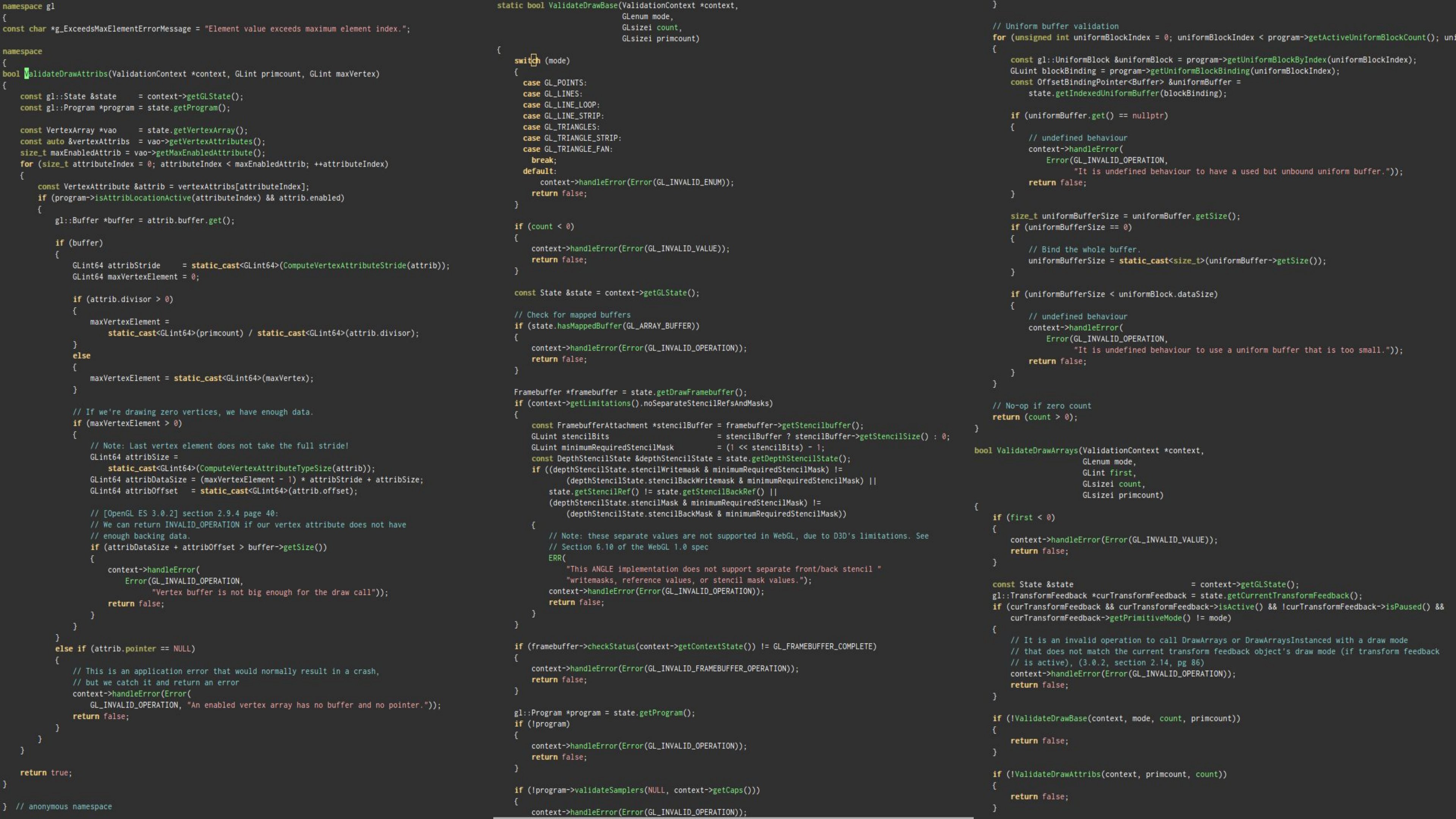
Pipelines


OpenGL view
Programmable fragment shader
Depth test
Blending
Framebuffer Write
Actual GPU
Programmable
Fragment shader
Depth test
Blending
Framebuffer Write
Explicit API view
Pipeline state
Fragment shader
Depth test
Blending
Framebuffer Write
vkCmdBindPipeline

Pipelines: What doesn’t go in

- Texture bindings, buffer bindings and friends
- Vertex and index buffers
- Some “continuous” fixed function state
  - Viewport, scissor
  - Depth clamp bounds, stencil masks and reference
  - ...

Command buffers



```cpp
namespace gl
{
const char *g_ExceedsMaxElementErrorMessage = "Element value exceeds maximum element index.";

namespace
{
bool ValidateDrawAttribs(ValidationContext *context, GLint primcount, GLint maxVertex)
{
    const gl::State &state     = context->getGLState();
    const gl::Program *program = state.getProgram();

    const VertexArray *vao     = state.getVertexArray();
    const auto &vertexAttribs  = vao->getVertexAttributes();
    size_t maxEnabledAttrib = vao->getMaxEnabledAttribute();
    for (size_t attributeIndex = 0; attributeIndex < maxEnabledAttrib; ++attributeIndex)
    {
        const VertexAttribute &attrib = vertexAttribs[attributeIndex];
        if (program->isAttribLocationActive(attributeIndex) && attrib.enabled)
        {
            gl::Buffer *buffer = attrib.buffer.get();

            if (buffer)
            {
                GLint64 attribStride     = static_cast<GLint64>(ComputeVertexAttributeStride(attrib));
                GLint64 maxVertexElement = 0;

                if (attrib.divisor > 0)
                {
                    maxVertexElement =
                        static_cast<GLint64>(primcount) / static_cast<GLint64>(attrib.divisor);
                }
                else
                {
                    maxVertexElement = static_cast<GLint64>(maxVertex);
                }

                // If we're drawing zero vertices, we have enough data.
                if (maxVertexElement > 0)
                {
                    // Note: Last vertex element does not take the full stride!
                    GLint64 attribSize =
                        static_cast<GLint64>(ComputeVertexAttributeTypeSize(attrib));
                    GLint64 attribDataSize = (maxVertexElement - 1) * attribStride + attribSize;
                    GLint64 attribOffset   = static_cast<GLint64>(attrib.offset);

                    // [OpenGL ES 3.0.2] section 2.9.4 page 40:
                    // We can return INVALID_OPERATION if our vertex attribute does not have
                    // enough backing data.
                    if (attribDataSize + attribOffset > buffer->getSize())
                    {
                        context->handleError(
                            Error(GL_INVALID_OPERATION,
                                  "Vertex buffer is not big enough for the draw call"));
                        return false;
                    }
                }
            }
            else if (attrib.pointer == NULL)
            {
                // This is an application error that would normally result in a crash,
                // but we catch it and return an error
                context->handleError(Error(
                    GL_INVALID_OPERATION, "An enabled vertex array has no buffer and no pointer."));
                return false;
            }
        }
    }

    return true;
}

}  // anonymous namespace
```

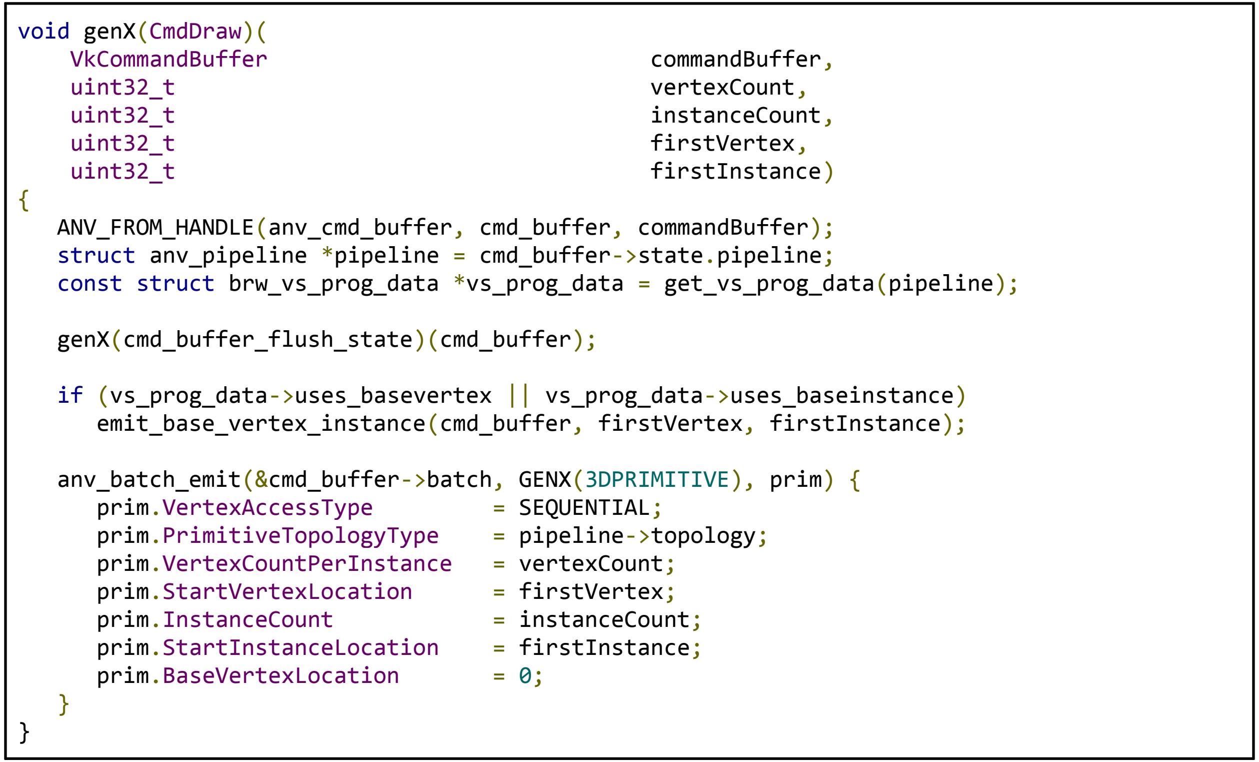
```cpp
static bool ValidateDrawBase(ValidationContext *context,
                             GLenum mode,
                             GLsizei count,
                             GLsizei primcount)
{
    switch (mode)
    {
      case GL_POINTS:
      case GL_LINES:
      case GL_LINE_LOOP:
      case GL_LINE_STRIP:
      case GL_TRIANGLES:
      case GL_TRIANGLE_STRIP:
      case GL_TRIANGLE_FAN:
        break;
      default:
          context->handleError(Error(GL_INVALID_ENUM));
        return false;
    }

    if (count < 0)
    {
        context->handleError(Error(GL_INVALID_VALUE));
        return false;
    }

    const State &state = context->getGLState();

    // Check for mapped buffers
    if (state.hasMappedBuffer(GL_ARRAY_BUFFER))
    {
        context->handleError(Error(GL_INVALID_OPERATION));
        return false;
    }

    Framebuffer *framebuffer = state.getDrawFramebuffer();
    if (context->getLimitations().noSeparateStencilRefsAndMasks)
    {
        const FramebufferAttachment *stencilBuffer = framebuffer->getStencilbuffer();
        GLuint stencilBits                         = stencilBuffer ? stencilBuffer->getStencilSize() : 0;
        GLuint minimumRequiredStencilMask          = (1 << stencilBits) - 1;
        const DepthStencilState &depthStencilState = state.getDepthStencilState();
        if ((depthStencilState.stencilWritemask & minimumRequiredStencilMask) !=
                (depthStencilState.stencilBackWritemask & minimumRequiredStencilMask) ||
            state.getStencilRef() != state.getStencilBackRef() ||
            (depthStencilState.stencilMask & minimumRequiredStencilMask) !=
                (depthStencilState.stencilBackMask & minimumRequiredStencilMask))
        {
            // Note: these separate values are not supported in WebGL, due to D3D's limitations. See
            // Section 6.10 of the WebGL 1.0 spec
            ERR(
                "This ANGLE implementation does not support separate front/back stencil "
                "writemasks, reference values, or stencil mask values.");
            context->handleError(Error(GL_INVALID_OPERATION));
            return false;
        }
    }

    if (framebuffer->checkStatus(context->getContextState()) != GL_FRAMEBUFFER_COMPLETE)
    {
        context->handleError(Error(GL_INVALID_FRAMEBUFFER_OPERATION));
        return false;
    }

    gl::Program *program = state.getProgram();
    if (!program)
    {
        context->handleError(Error(GL_INVALID_OPERATION));
        return false;
    }

    if (!program->validateSamplers(NULL, context->getCaps()))
    {
        context->handleError(Error(GL_INVALID_OPERATION));
```

```cpp
    }

    // Uniform buffer validation
    for (unsigned int uniformBlockIndex = 0; uniformBlockIndex < program->getActiveUniformBlockCount(); uni
    {
        const gl::UniformBlock &uniformBlock = program->getUniformBlockByIndex(uniformBlockIndex);
        GLuint blockBinding = program->getUniformBlockBinding(uniformBlockIndex);
        const OffsetBindingPointer<Buffer> &uniformBuffer =
            state.getIndexedUniformBuffer(blockBinding);

        if (uniformBuffer.get() == nullptr)
        {
            // undefined behaviour
            context->handleError(
                Error(GL_INVALID_OPERATION,
                      "It is undefined behaviour to have a used but unbound uniform buffer."));
            return false;
        }

        size_t uniformBufferSize = uniformBuffer.getSize();
        if (uniformBufferSize == 0)
        {
            // Bind the whole buffer.
            uniformBufferSize = static_cast<size_t>(uniformBuffer->getSize());
        }

        if (uniformBufferSize < uniformBlock.dataSize)
        {
            // undefined behaviour
            context->handleError(
                Error(GL_INVALID_OPERATION,
                      "It is undefined behaviour to use a uniform buffer that is too small."));
            return false;
        }
    }

    // No-op if zero count
    return (count > 0);
}

bool ValidateDrawArrays(ValidationContext *context,
                        GLenum mode,
                        GLint first,
                        GLsizei count,
                        GLsizei primcount)
{
    if (first < 0)
    {
        context->handleError(Error(GL_INVALID_VALUE));
        return false;
    }

    const State &state                          = context->getGLState();
    gl::TransformFeedback *curTransformFeedback = state.getCurrentTransformFeedback();
    if (curTransformFeedback && curTransformFeedback->isActive() && !curTransformFeedback->isPaused() &&
        curTransformFeedback->getPrimitiveMode() != mode)
    {
        // It is an invalid operation to call DrawArrays or DrawArraysInstanced with a draw mode
        // that does not match the current transform feedback object's draw mode (if transform feedback
        // is active), (3.0.2, section 2.14, pg 86)
        context->handleError(Error(GL_INVALID_OPERATION));
        return false;
    }

    if (!ValidateDrawBase(context, mode, count, primcount))
    {
        return false;
    }

    if (!ValidateDrawAttribs(context, primcount, count))
    {
        return false;
    }
```

Command buffers

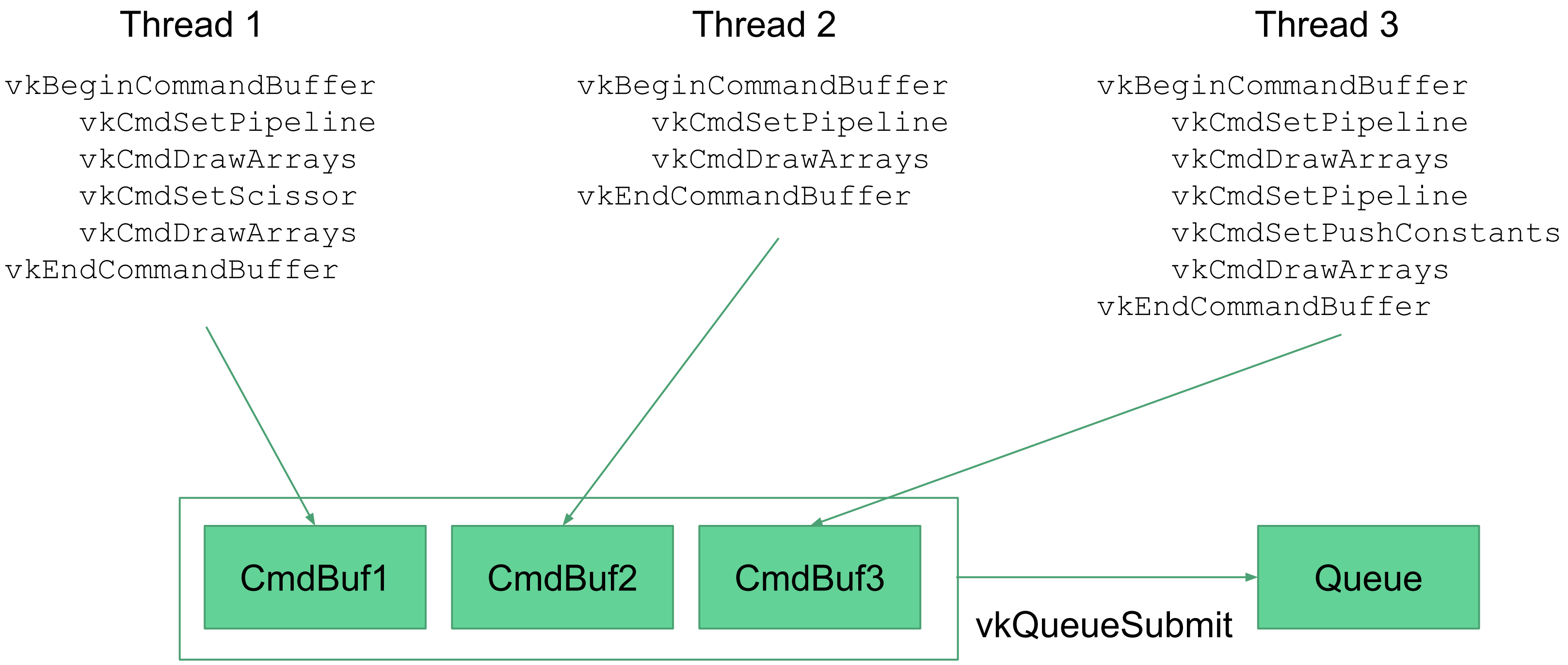
- Send commands as batch instead of iteratively
  - No need to do state tracking, everything is there
  - Multithreaded creation
- All GPUs consume commands from memory
  - Commands can be serialized directly
  - Can be reused (subject to conditions)

Command buffers



```
void genX(CmdDraw)(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    vertexCount,
    uint32_t                                    instanceCount,
    uint32_t                                    firstVertex,
    uint32_t                                    firstInstance)
{
   ANV_FROM_HANDLE(anv_cmd_buffer, cmd_buffer, commandBuffer);
   struct anv_pipeline *pipeline = cmd_buffer->state.pipeline;
   const struct brw_vs_prog_data *vs_prog_data = get_vs_prog_data(pipeline);

   genX(cmd_buffer_flush_state)(cmd_buffer);

   if (vs_prog_data->uses_basevertex || vs_prog_data->uses_baseinstance)
      emit_base_vertex_instance(cmd_buffer, firstVertex, firstInstance);

   anv_batch_emit(&cmd_buffer->batch, GENX(3DPRIMITIVE), prim) {
      prim.VertexAccessType          = SEQUENTIAL;
      prim.PrimitiveTopologyType     = pipeline->topology;
      prim.VertexCountPerInstance    = vertexCount;
      prim.StartVertexLocation       = firstVertex;
      prim.InstanceCount             = instanceCount;
      prim.StartInstanceLocation     = firstInstance;
      prim.BaseVertexLocation        = 0;
   }
}
```

Command buffers enable multithreading



Thread 1

```
vkBeginCommandBuffer
    vkCmdSetPipeline
    vkCmdDrawArrays
    vkCmdSetScissor
    vkCmdDrawArrays
vkEndCommandBuffer
```

Thread 2

```
vkBeginCommandBuffer
    vkCmdSetPipeline
    vkCmdDrawArrays
vkEndCommandBuffer
```

Thread 3

```
vkBeginCommandBuffer
    vkCmdSetPipeline
    vkCmdDrawArrays
    vkCmdSetPipeline
    vkCmdSetPushConstants
    vkCmdDrawArrays
vkEndCommandBuffer
```

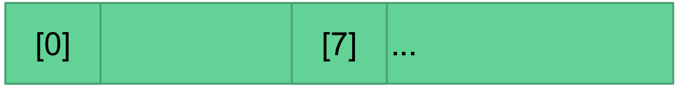
Binding model

Binding model

- How you pass stuff to shaders
  - Texture
  - Uniforms
  - …
- Simplified views of architectures:
  - Texture units (fixed function)
  - In-memory descriptors (bindless)

Fixed function uniforms

Constant register array



Updating one uniform

```
glUniform1f(program,foo);
```

Could become the following in the driver:

```
internalCmdBuf->SetCRegister(7, foo);
```

```
layout(location=7) float foo;
//Use foo
```

Gets compiled to

```
float foo = CRegister[7];
//Use foo
```

Fixed function textures

Texture units	OldTex A	Old Albedo	OldTex C	OldTex D

After updating one texture

```
glActiveTexture(GL_TEXTURE0 + 1);
glBindTexture(GL_TEXTURE_2D, newAlbedo);
```

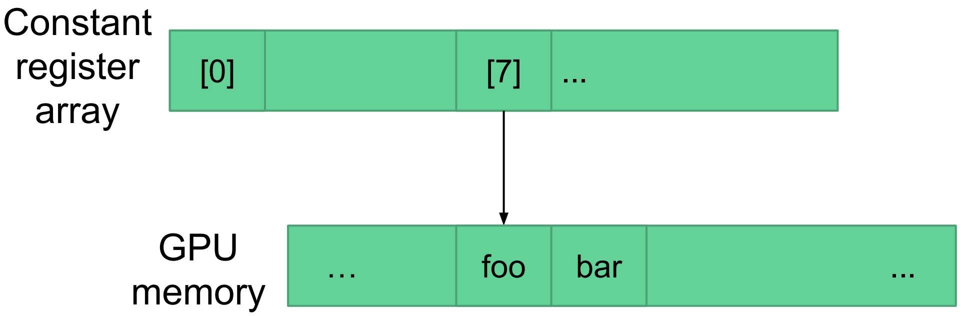
Texture units	OldTex A	New Albedo	OldTex C	OldTex D

```
layout(location=1) sampler2D albedo;
vec4 color = texture(albedo, texcoord);
```

Gets compiled to

```
vec4 color = TextureUnit1.Sample(texcoord);
```

Bindless uniforms AKA uniform buffers



Constant register array

[0]

[7]

...

GPU memory

…

foo

bar

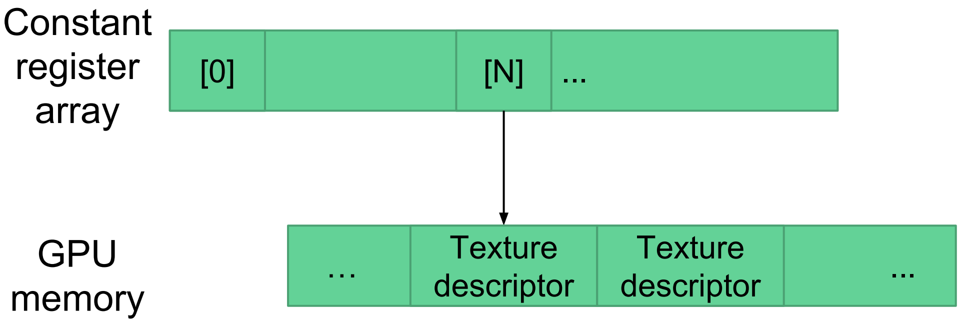
...

```
uniform Block {
    float foo;
    float bar;
} myBlock;

// Use myBlock.foo
```

Gets compiled to

```
Block* _myBlock = CRegister[7];
float _myBlock_foo = _myBlock->foo;

// Use _myBlock_foo
```

Bindless textures



Constant register array

[0]

[N]

...

GPU memory

…

Texture descriptor

Texture descriptor

...

```
layout(location=1) sampler2D albedo;
vec4 color = texture(albedo, texcoord);
```

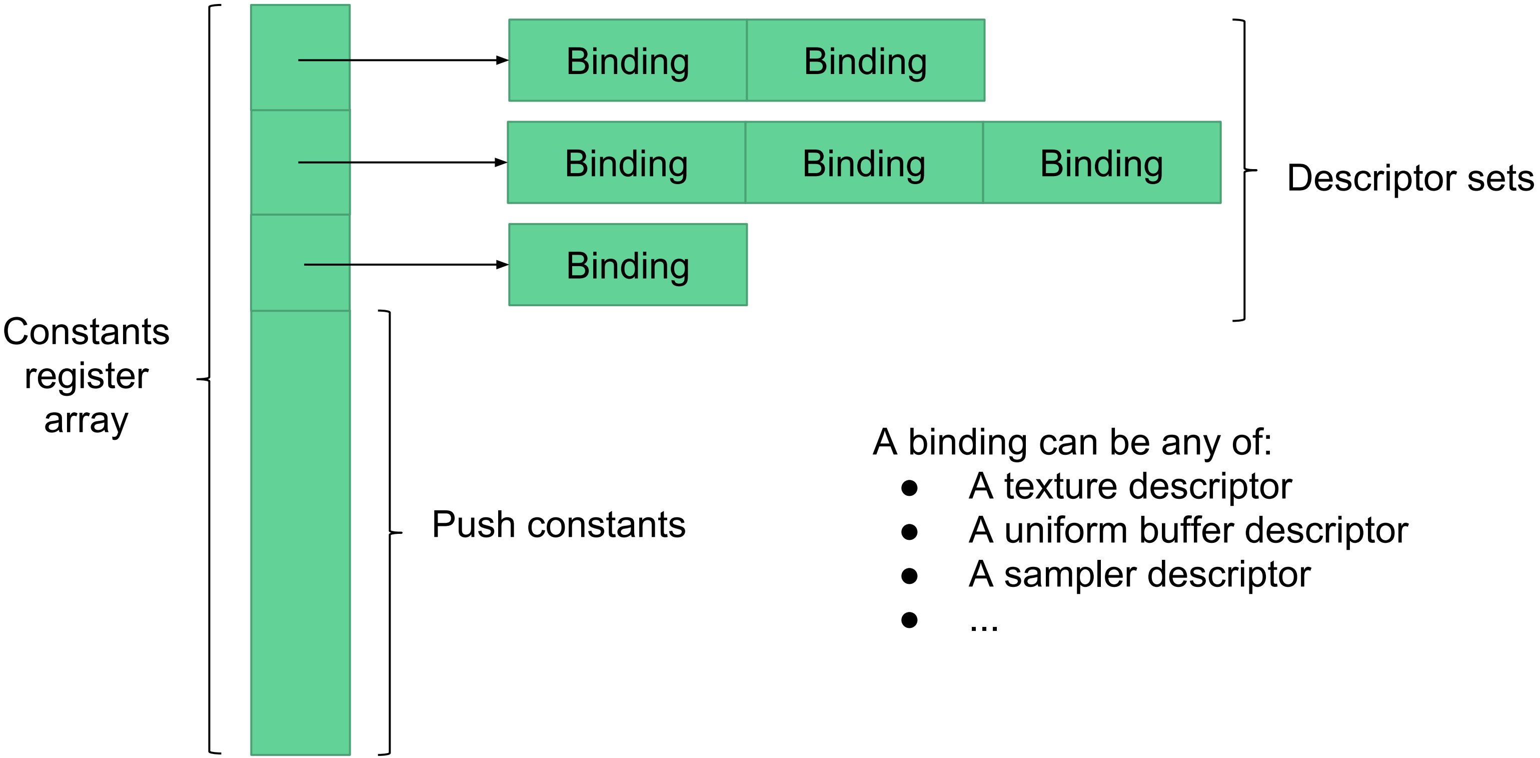
Gets compiled to

```
struct TextureDescriptor {
    ivec2 size;
    int format;
    void *data;
};

TextureDescriptor* textures = CRegister[N];
TextureDescriptor* albedo = &textures[1];
vec4 color = Sample2D(albedo, texcoord);
```

Problems with non-explicit binding models

- Binding model of non-explicit APIs.
  - OpenGL: bind one by one
  - D3D11 and Metal: change ranges of a binding table
- Problem for bindless hardware:
  - Need to keep copies of the binding tables while shader is in flight
  - Changing one binding requires a complete table copy

The Vulkan binding model in one slide



A binding can be any of:

- A texture descriptor
- A uniform buffer descriptor
- A sampler descriptor
- ...

The Vulkan binding model in GLSL

```
layout(set = 1, binding = 0) uniform texture2D albedo;

layout(push_constant) uniform Block {
    int member1;
    float member2;
    ...
} pushConstants;

float foo = texture(t, texcoord).r + pushConstants.member1;
```

Gets compiled to:

```
Descriptor* set1 = CRegister[1];
TextureDescriptor* albedo = set1[0];

pushConstants_member1 = CRegister[PUSH_CONSTANT_START + 0];
float foo = Sample2D(albedo, texcoord).r + pushConstants_member1;
```

The Vulkan binding model on the API side

- Create a vkDescriptorPool
  - Wrapper around a chunk of GPU memory
- Ask the driver to vkAllocateDescriptorSets in the pool
- Write the descriptor set
- Use it at draw-time with vkCmdBindDescriptorSet

Advantages of the Vulkan binding model

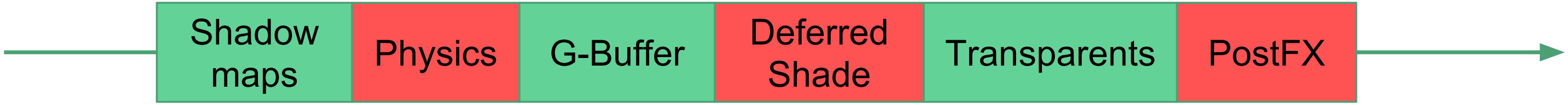
- Bindings can be grouped by usage frequency
  - No needless rewriting of descriptor data
  - Per-frame, per-material, per-object
- Maps well to bindless hardware with few constant registers
- GPU allocation of descriptor sets done by the application
- Scales down nicely to non-bindless hardware

New concepts for new features

Queues

Queues

- Represent a GPU “thread”
  - 1 hardware graphics queue usually
  - Use context switching to handle multiple logical queues
- Command buffers are submitted to a queue



Shadow maps

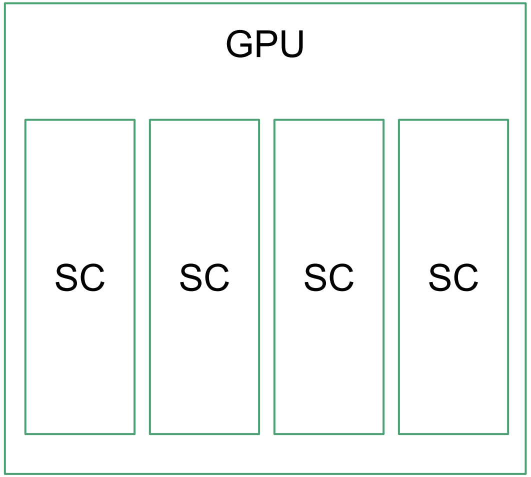
Physics

G-Buffer

Deferred Shade

Transparents

PostFX



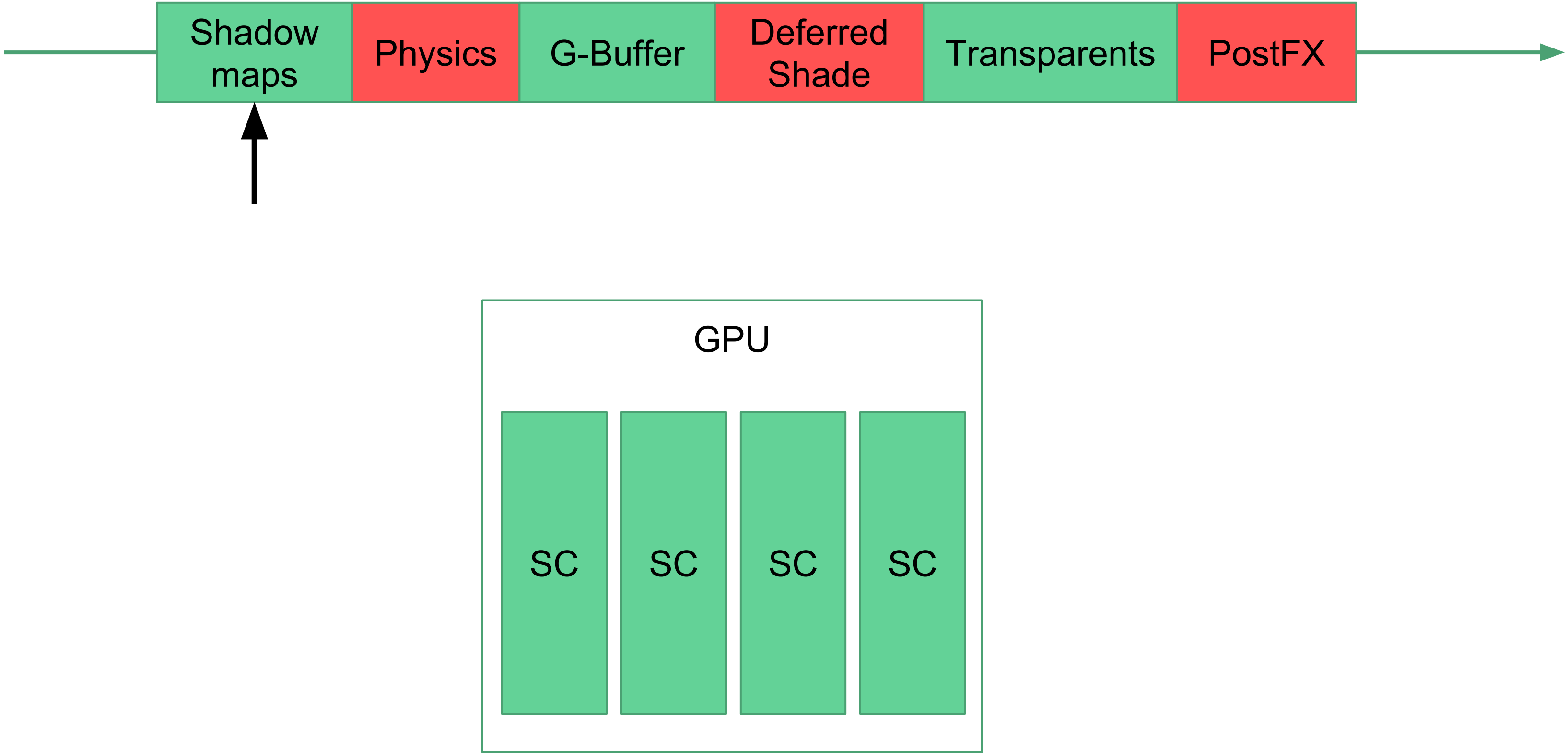
GPU

SC

SC

SC

SC



Shadow maps

Physics

G-Buffer

Deferred Shade

Transparents

PostFX

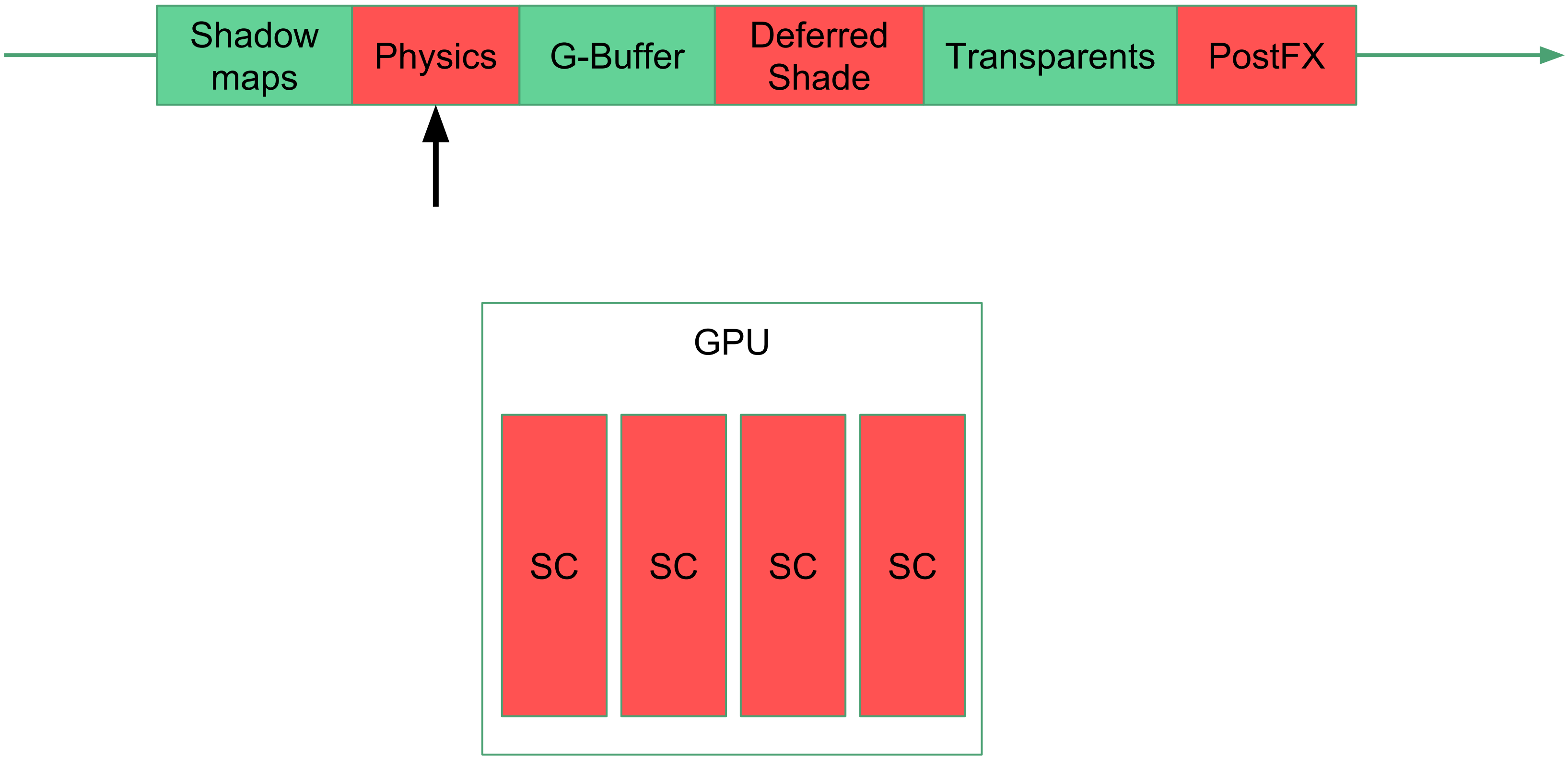
GPU

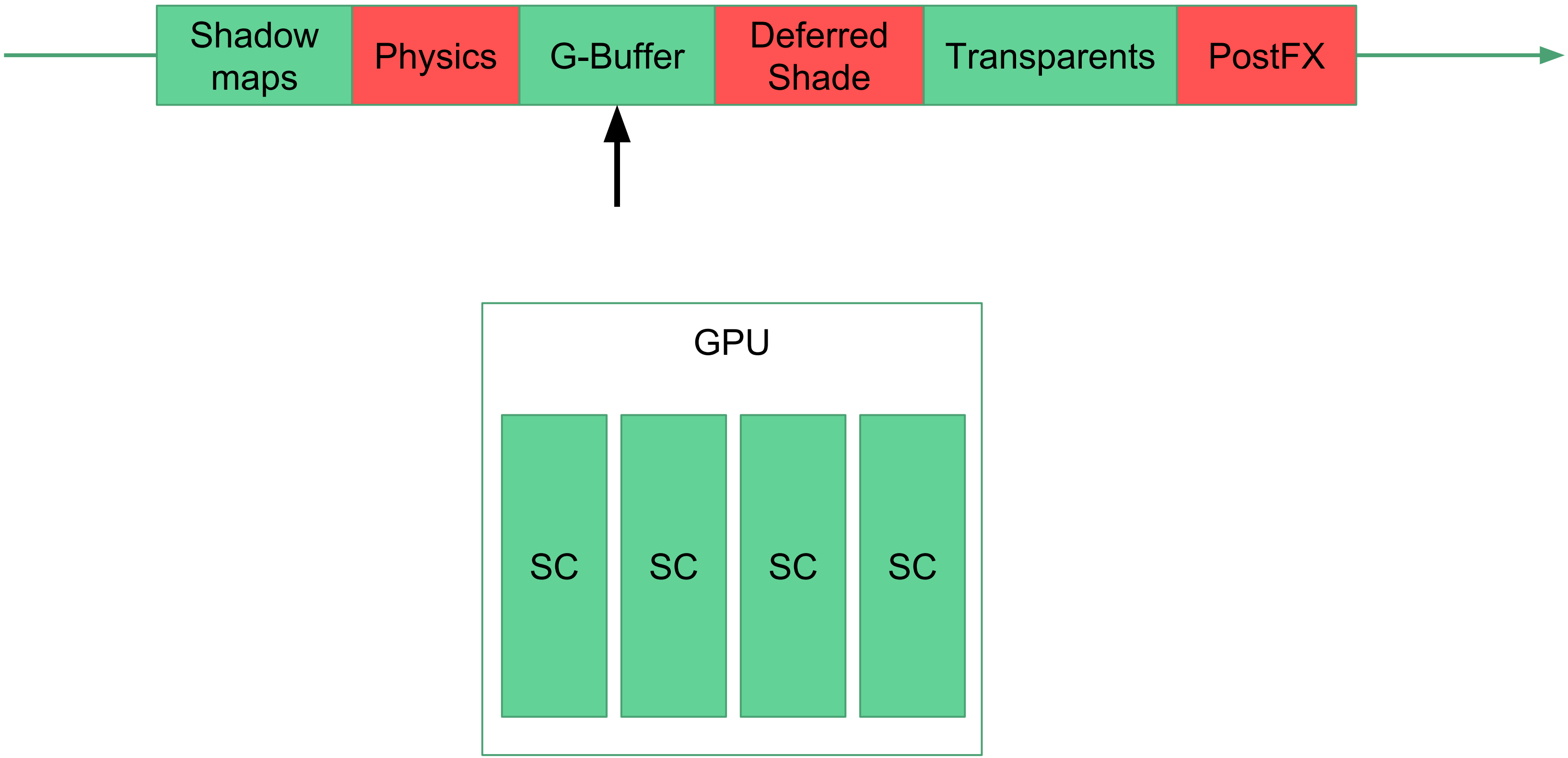
SC

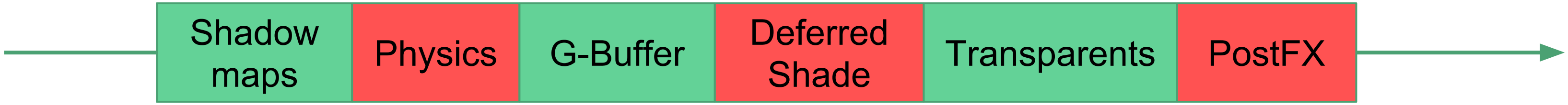
SC

SC

SC


Shadow maps
Physics
G-Buffer
Deferred Shade
Transparents
PostFX
GPU
SC
SC
SC
SC


Shadow maps
Physics
G-Buffer
Deferred Shade
Transparents
PostFX
GPU
SC
SC
SC
SC



Shadow maps

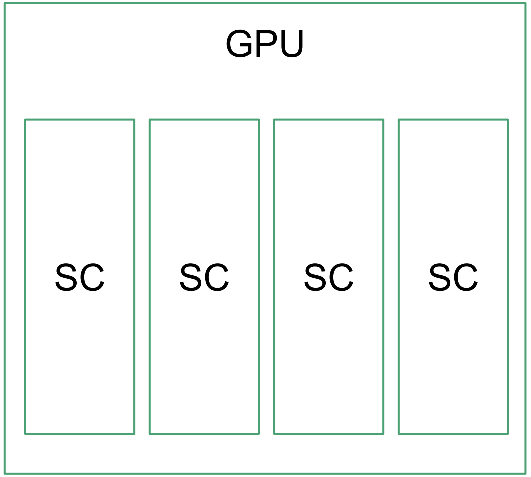
Physics

G-Buffer

Deferred Shade

Transparents

PostFX



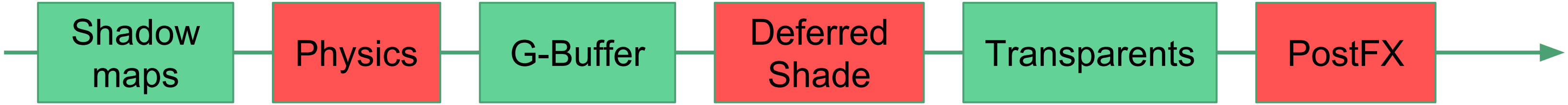
GPU

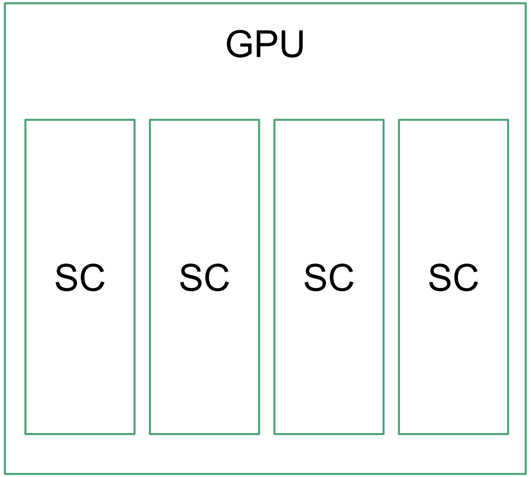
SC

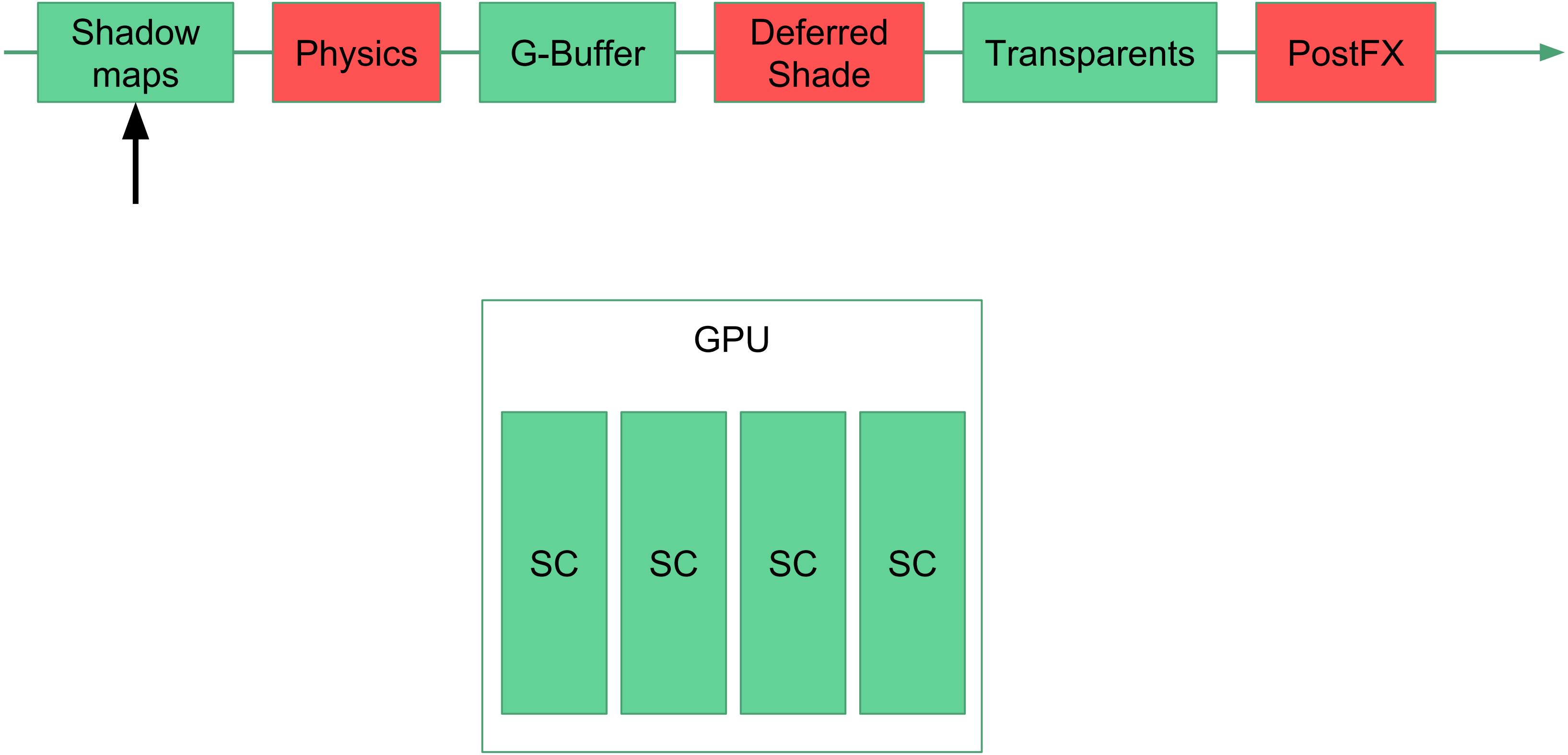
SC

SC

SC


Shadow maps
Physics
G-Buffer
Deferred Shade
Transparents
PostFX


GPU
SC
SC
SC
SC



Shadow maps

Physics

G-Buffer

Deferred Shade

Transparents

PostFX

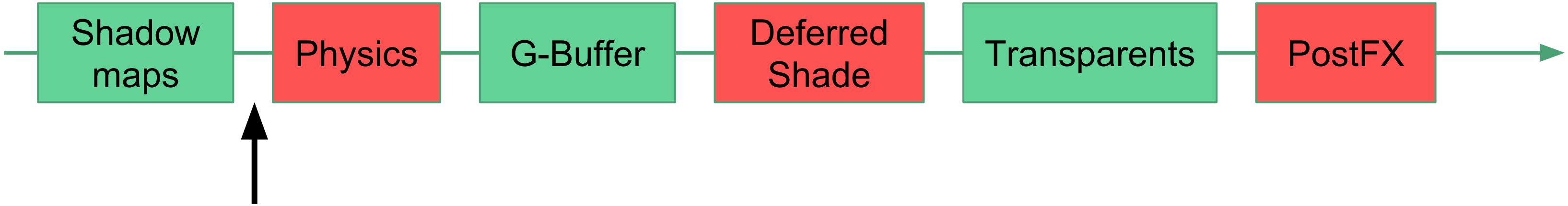
GPU

SC

SC

SC

SC



Shadow maps

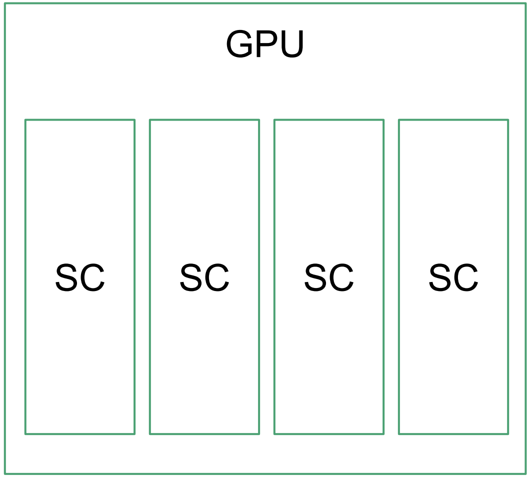
Physics

G-Buffer

Deferred Shade

Transparents

PostFX



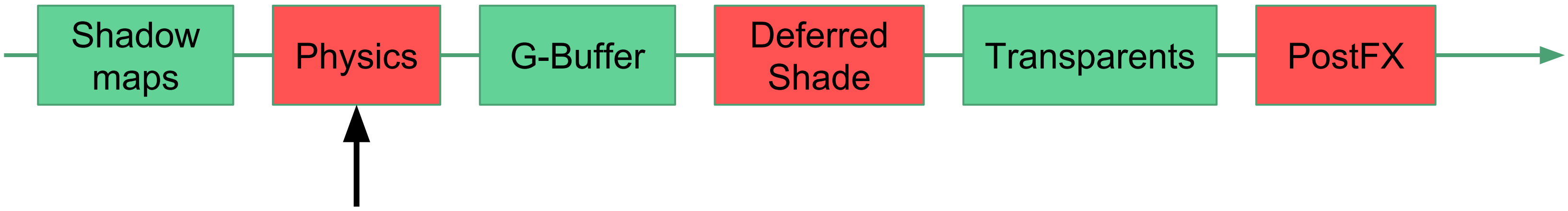
GPU

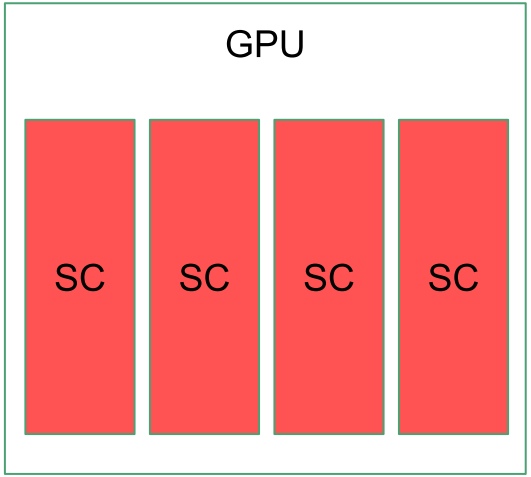
SC

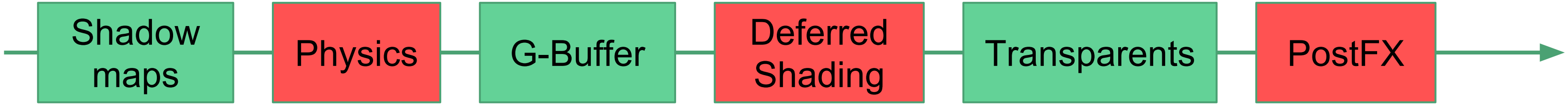
SC

SC

SC


Shadow maps
Physics
G-Buffer
Deferred Shade
Transparents
PostFX


GPU
SC
SC
SC
SC



Shadow maps

Physics

G-Buffer

Deferred Shading

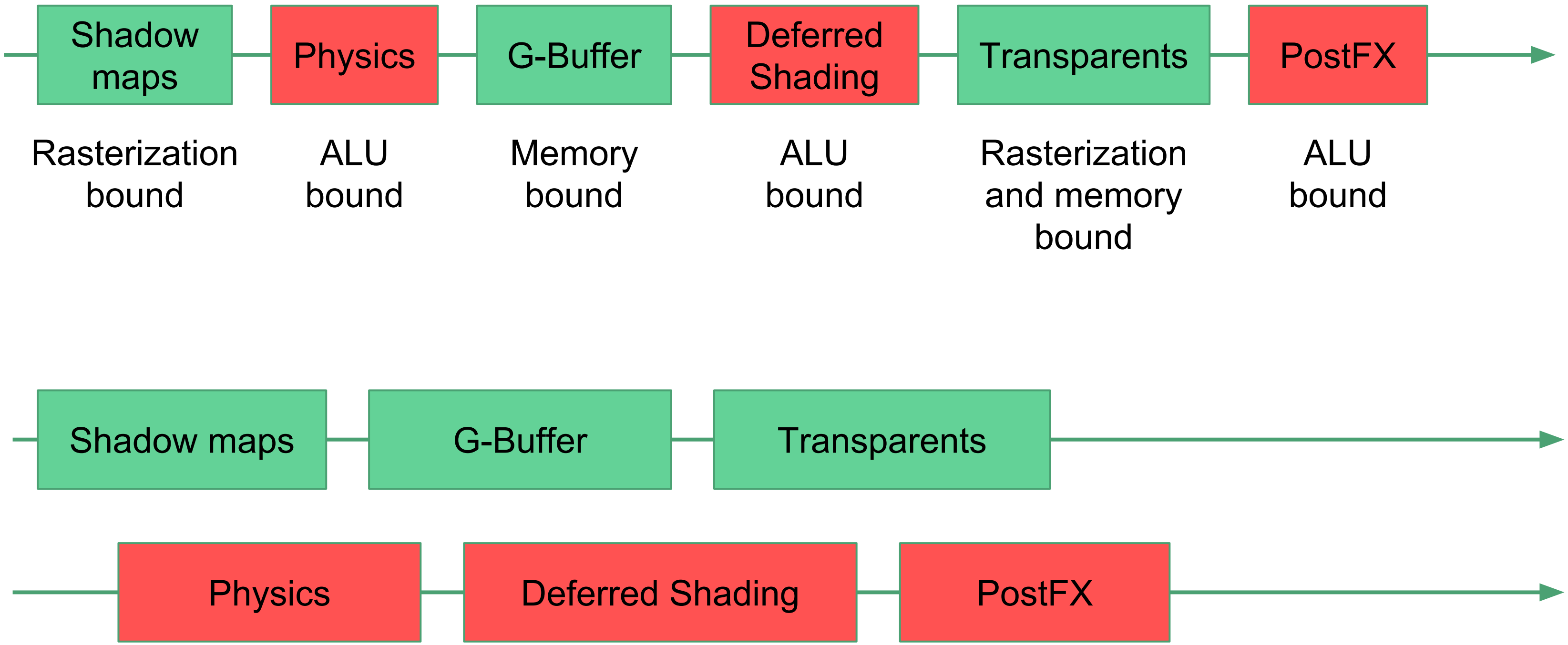
Transparents

PostFX

Rasterization bound	ALU bound	Memory bound	ALU bound	Rasterization and memory bound	ALU bound

Queues: Async compute

- Modern desktop GPUs can use simultaneously
  - 1 graphics queue
  - 8 compute queues
- Running compute and graphics in parallel gives large benefits
  - +~10% on NVIDIA
  - +~30% on AMD



Shadow maps

Physics

G-Buffer

Deferred Shading

Transparents

PostFX

Rasterization bound

ALU bound

Memory bound

ALU bound

Rasterization and memory bound

ALU bound

Shadow maps

G-Buffer

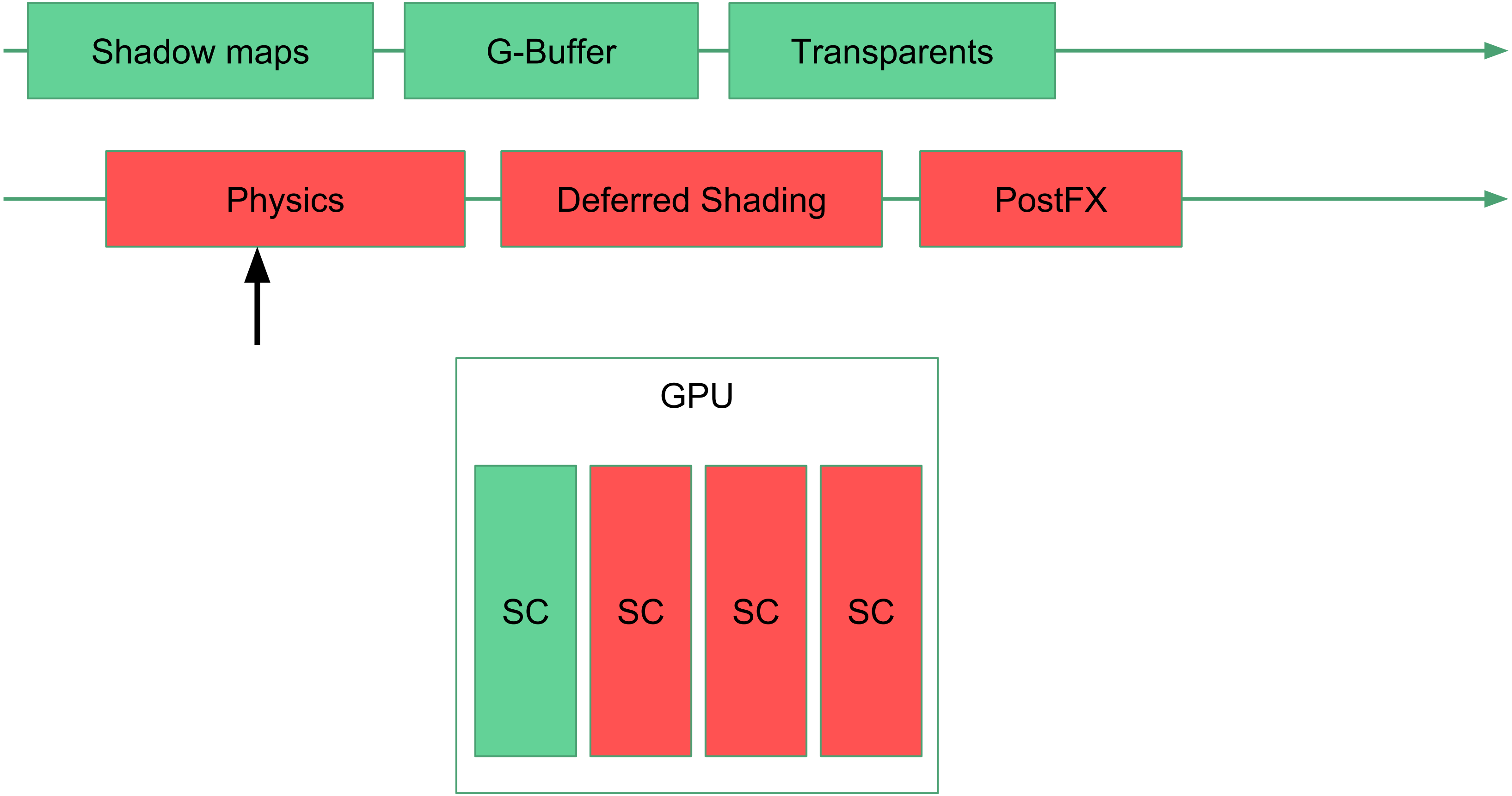
Transparents

Physics

Deferred Shading

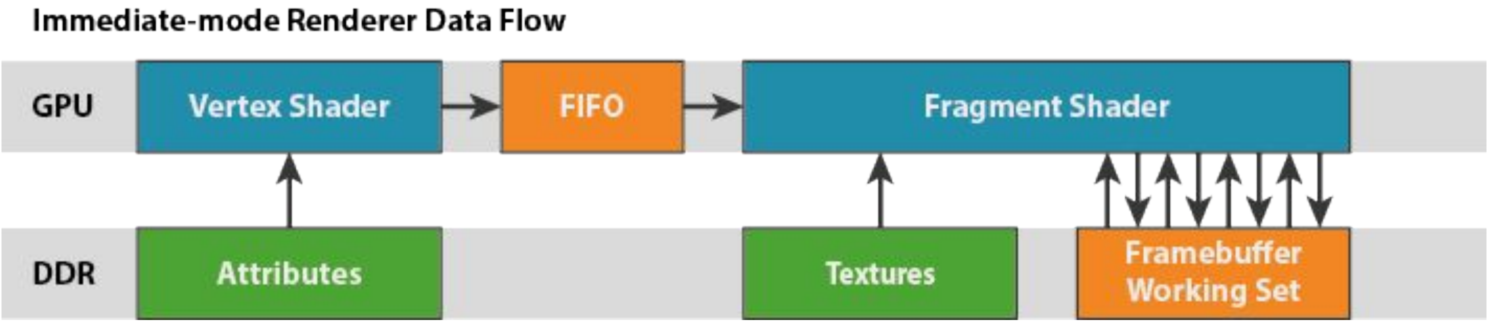
PostFX

Everything bound (in theory)


Shadow maps
G-Buffer
Transparents
Physics
Deferred Shading
PostFX
GPU
SC
SC
SC
SC

Render passes - Part one

Immediate mode rendering



```
foreach(triangle)
    foreach(fragment)
        load FBO data (color, depth, ...)
        call fragment shader
        store new FBO data
```

Problems of immediate mode rendering

```
foreach(triangle)
    foreach(fragment in triangle)
        load FBO data (color, depth, ...)
        call fragment shader
        store new FBO data
```

- Random accesses thrash the caches
- Loading and storing the same fragment multiple times costs power, especially on mobile

Tile based rendering

```
foreach(fragment)
    load FBO data (color, depth, ...)
    foreach(triangle)
        call fragment shader
    store new FBO data
```

- Idea: switch the loops
- Problem: storing triangles per pixel is too expensive

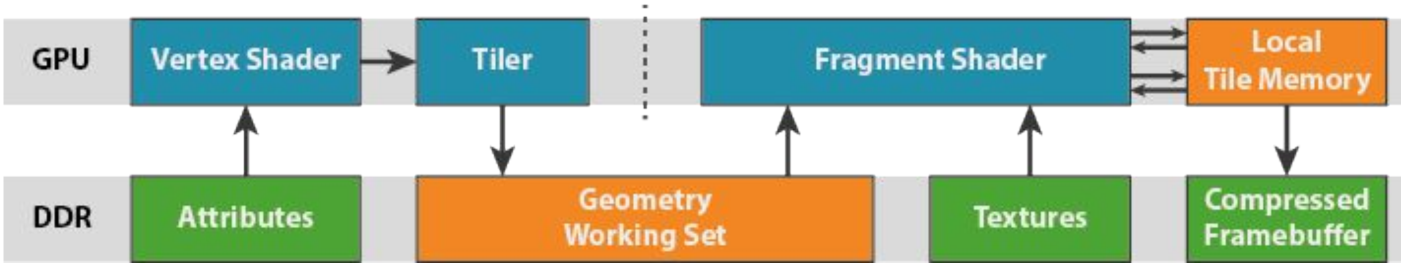
Tile based rendering

```
foreach(tile)
    load tile FBO data (color, depth, ...)
    foreach(triangle in tile)
        foreach(fragment in triangle in tile)
            call fragment shader
    store new tile FBO data
```

- Idea: split FBO in tiles, store triangles per tile
    - A tile can for example be a 16x16 square
- Helps with cache coherency, FBO stored as array of tiles
- One load and one store per pixel

Tile-based rendering

Tile-based Renderer Data Flow



Per-tile triangle lists

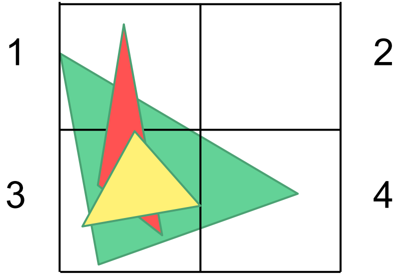
1

2

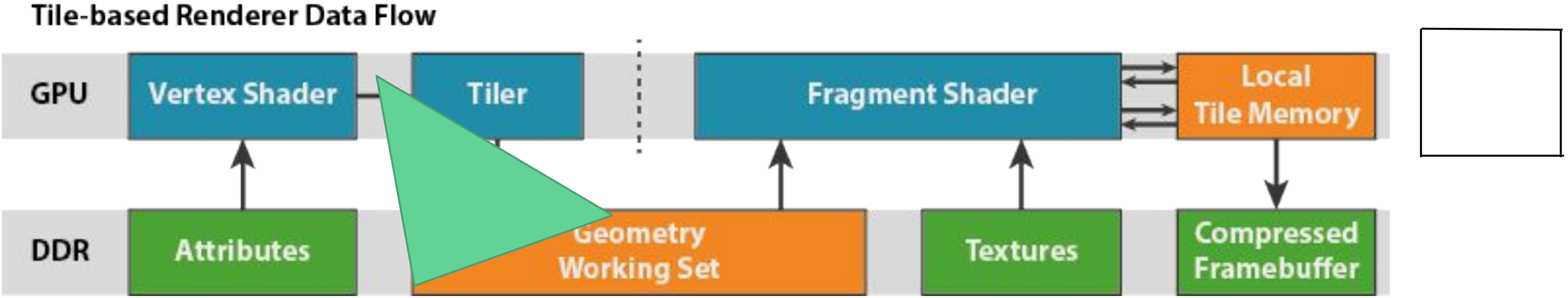
3

4

Final state:



Tile-based rendering



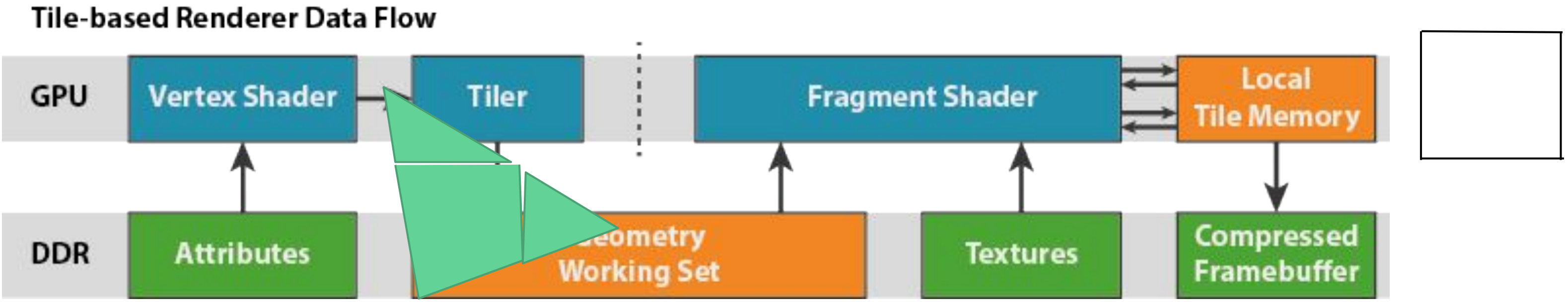
1

Per-tile triangle lists 2

3

4

Tile-based rendering



1

Per-tile triangle lists

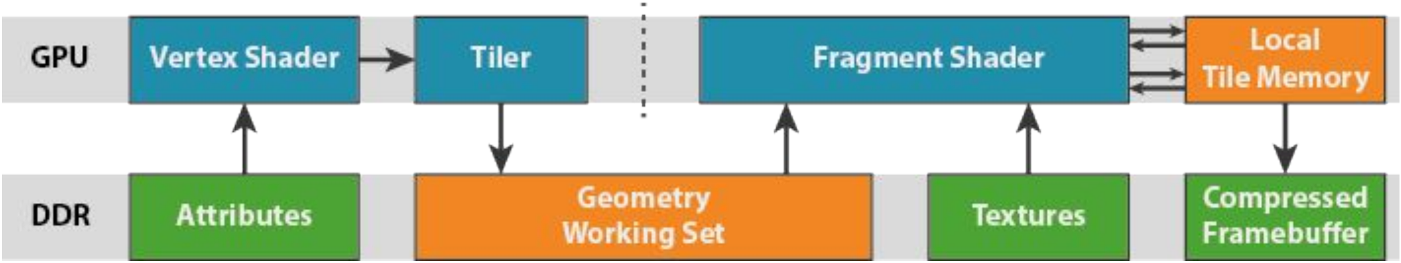
2

3

4

Tile-based rendering

Tile-based Renderer Data Flow



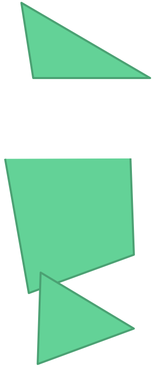
Per-tile triangle lists

1

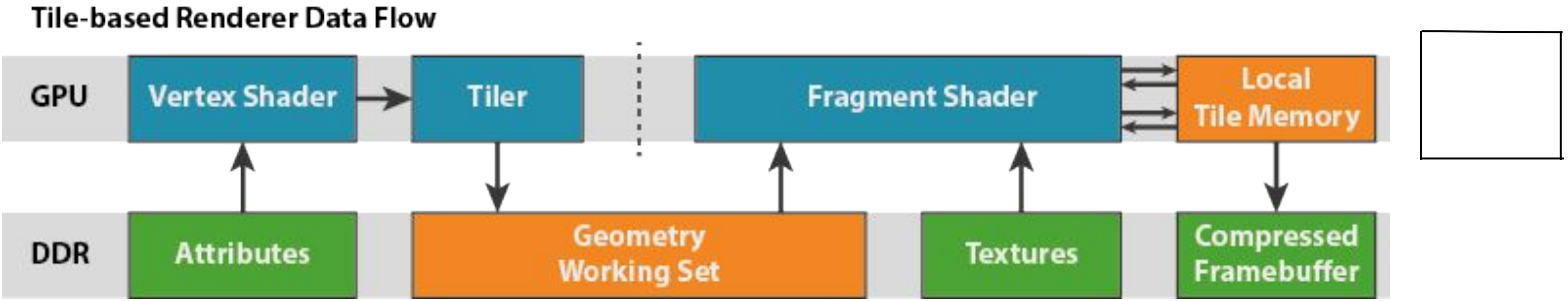
2

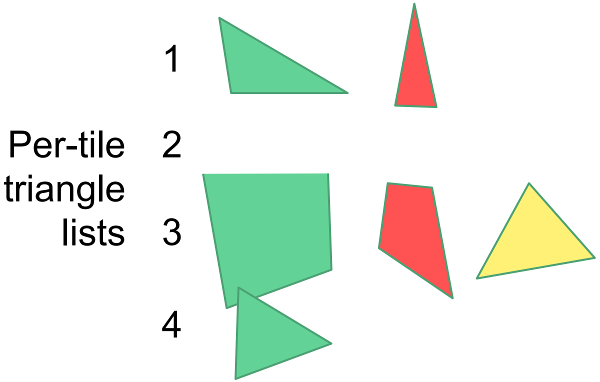
3

4

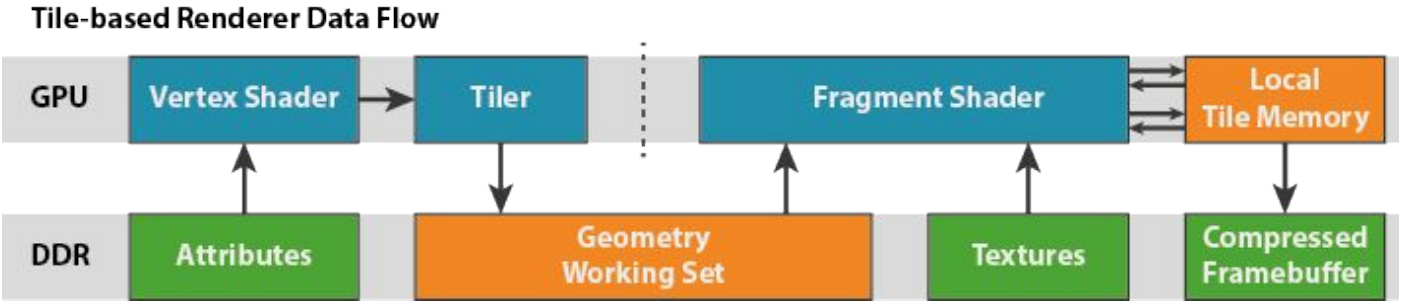


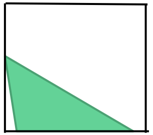
Tile-based rendering

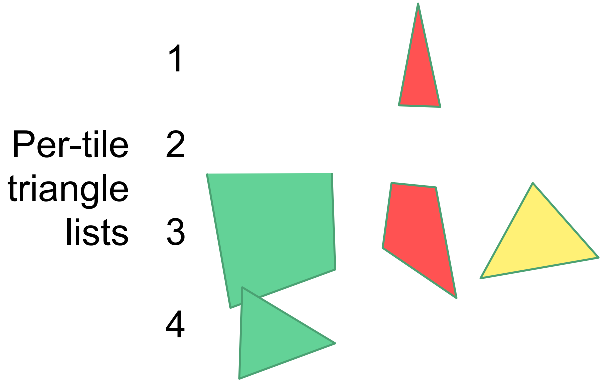




Tile-based rendering







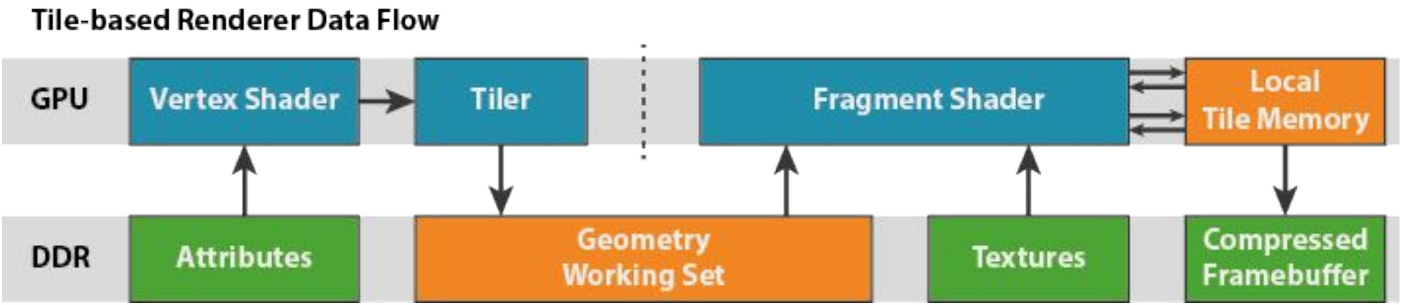
1

Per-tile triangle lists 2

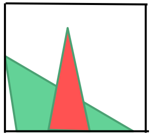
3

4

Tile-based rendering



Tile-based Renderer Data Flow



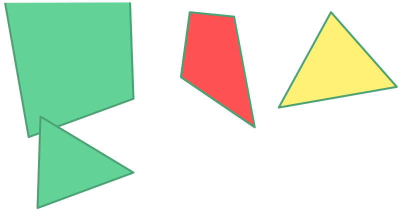
1

Per-tile triangle lists

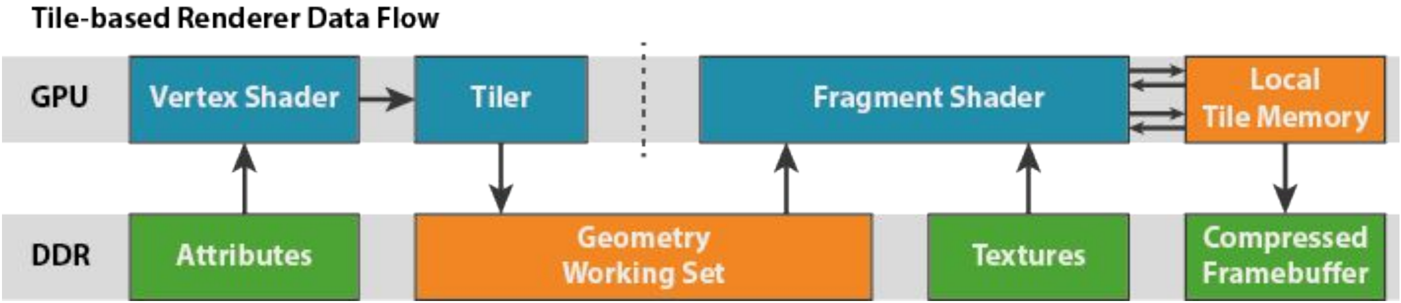
2

3

4



Tile-based rendering



Tile-based Renderer Data Flow

GPU

Vertex Shader

Tiler

Fragment Shader

Local Tile Memory

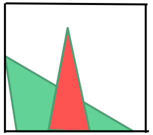
DDR

Attributes

Geometry Working Set

Textures

Compressed Framebuffer



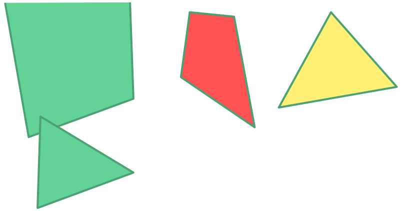
Per-tile triangle lists

1

2

3

4



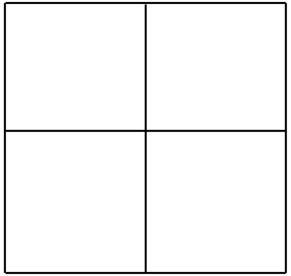
Rendertarget:

1

2

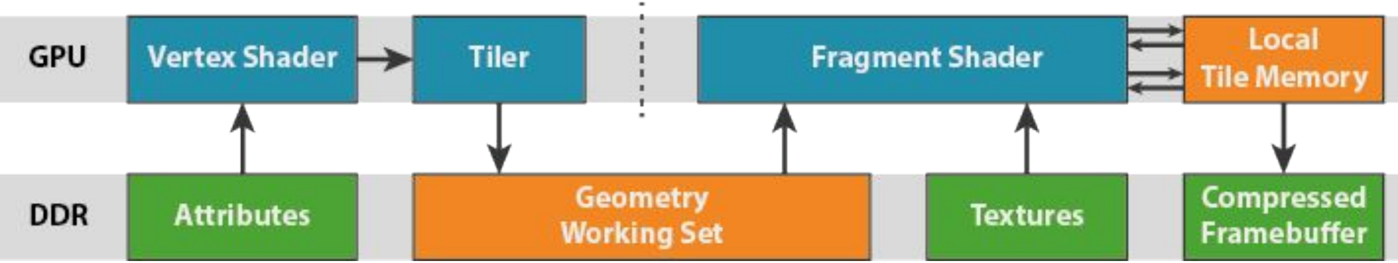
3

4

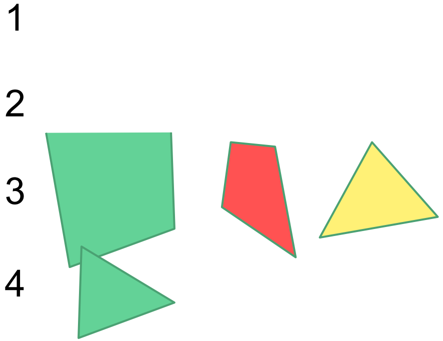


Tile-based rendering

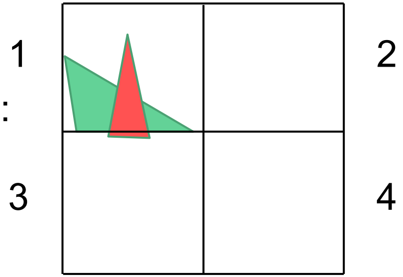
Tile-based Renderer Data Flow



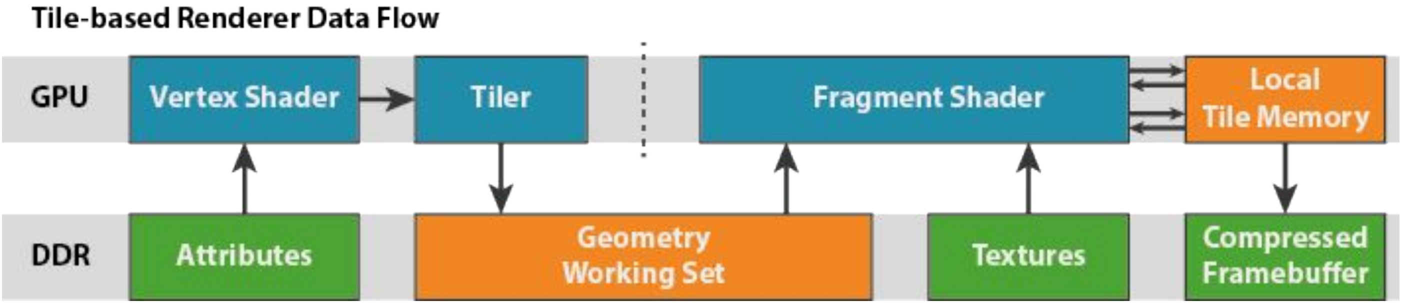
Per-tile triangle lists

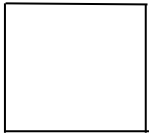


Rendertarget:



Tile-based rendering





Per-tile triangle lists

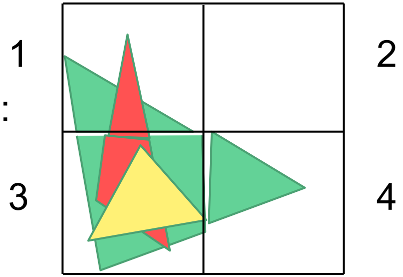
1

2

3

4

Rendertarget:



More optimizations

- What if
  - We know the initial depth is constant?
  - We only use the depth for testing and don’t care about the final value?
  - We don’t care about the initial color?

Vulkan Subpass

- Defines
  - the shape of the rendertarget (e.g. RGBA8, D24S8)
  - the tile initialization (LOAD, CLEAR, DONT_CARE)
  - the tile finalization (STORE, RESOLVE, DONT_CARE)
- A different subpass for each “framebuffer change”
- In the command buffers draws are done in subpasses

Render passes - Part two

Even more optimizations?

- Most tile memory has space for more than color and depth
    - Or HW can reduce tile size to make it happen
- Allow application to store custom data in tile cache
- Example optimization for deferred rendering

```
foreach(tile)
    create GBuffer
    store tile

foreach(tile)
    load GBuffer data
    do lighting
    store tile
```

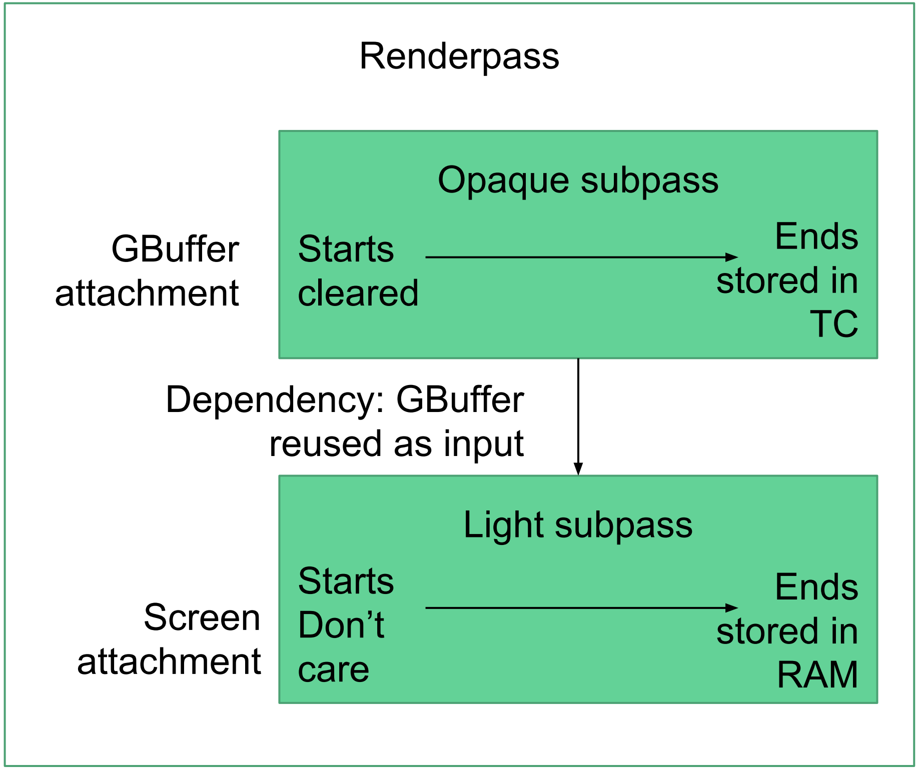
becomes

```
foreach(tile)
    create GBuffer
    do lighting
    store tile
```

Vulkan Renderpasses

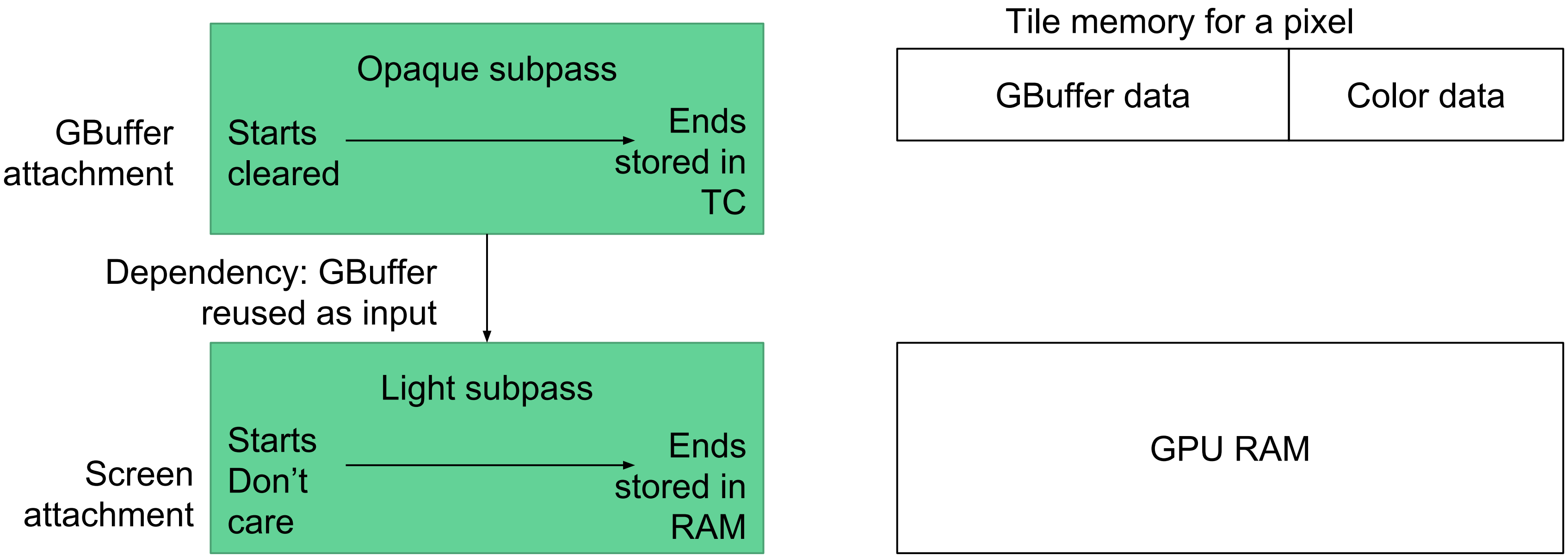
- Object describing the rendering algorithm:
  - All attachments used
  - Subpass with their attachments and load / store actions
  - Dependencies between subpasses
- Adds a new type descriptor type “input”, tells the driver it can keep data in tile memory.

Annotated GBuffer example in Vulkan



```
renderpass = vkCreateRenderPass(
    {GBuffer, Screen},
    {Opaque subpass, Light subpass},
    {dependency}
);

// Compile pipeline against renderpass

// Use renderpass in command buffer
cmdBuf->BeginRenderPass(renderpass)
    // Do Opaque pass commands
cmdBuf->NextSubpass()
    // Do Light pass commands
cmdBuf->EndRenderPass()
```

Annotated GBuffer example in Vulkan



GBuffer attachment

Opaque subpass

Starts cleared

Ends stored in TC

Dependency: GBuffer reused as input

Screen attachment

Light subpass

Starts Don’t care

Ends stored in RAM

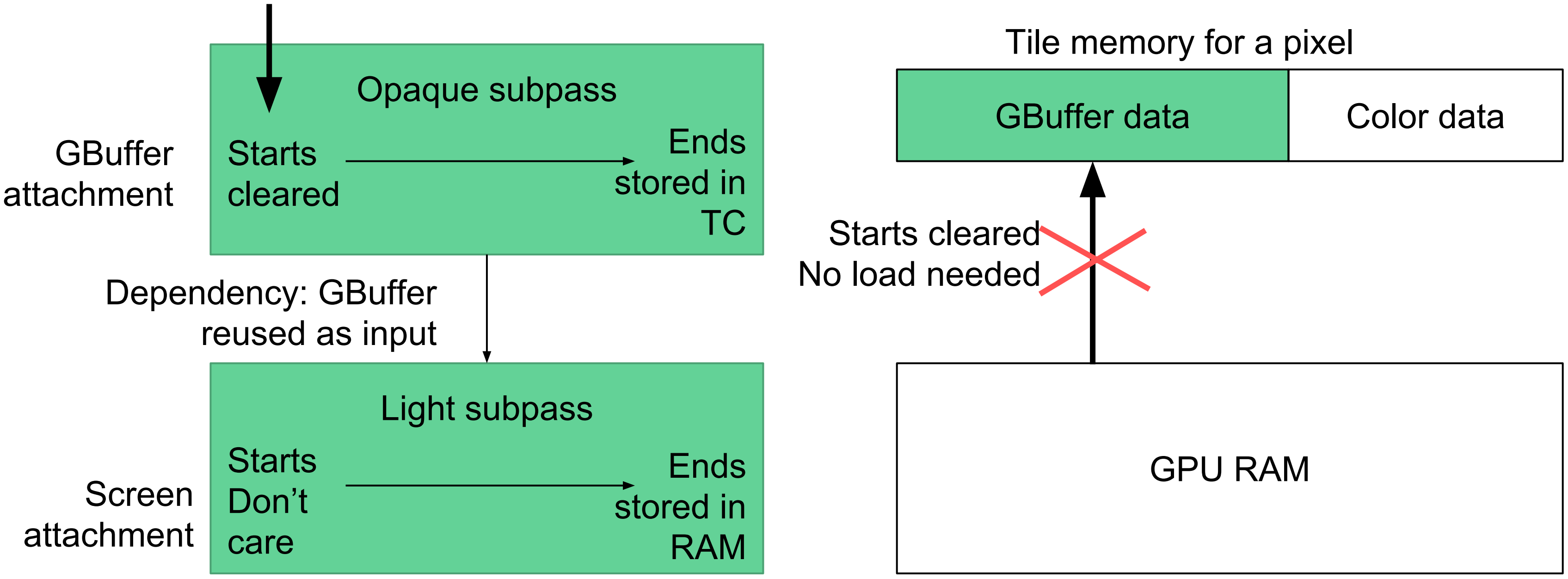
Tile memory for a pixel

GBuffer data

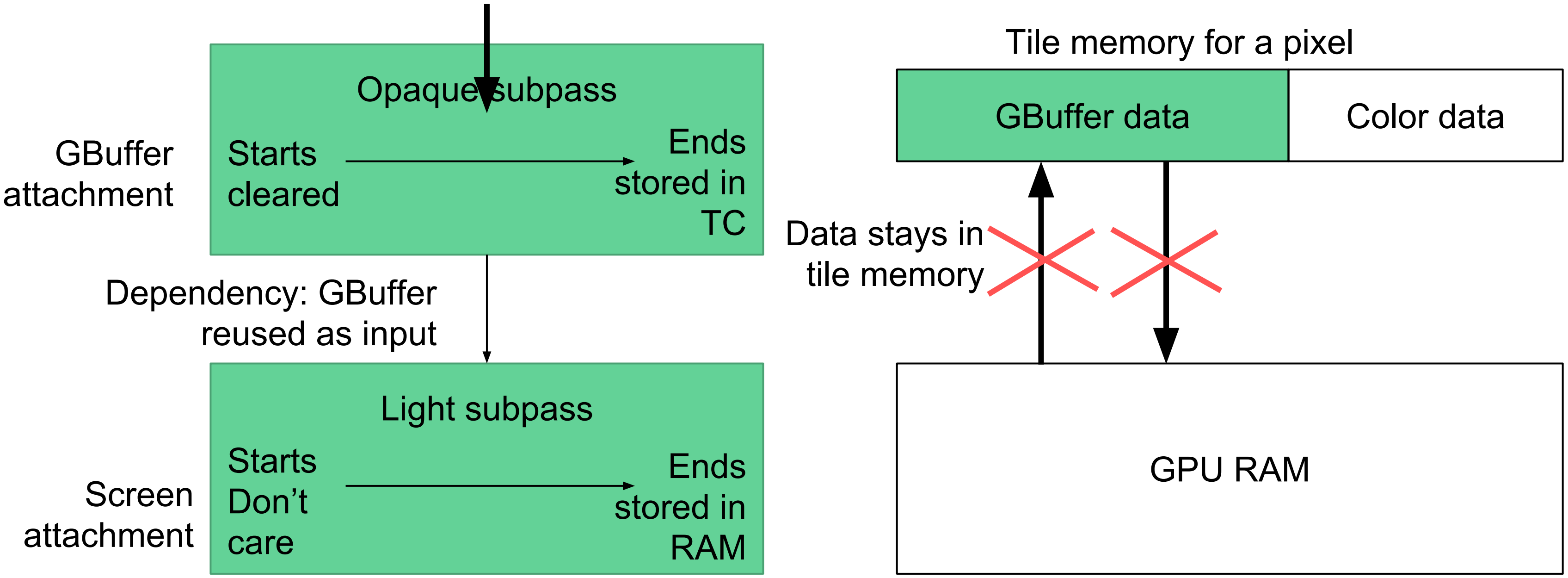
Color data

GPU RAM

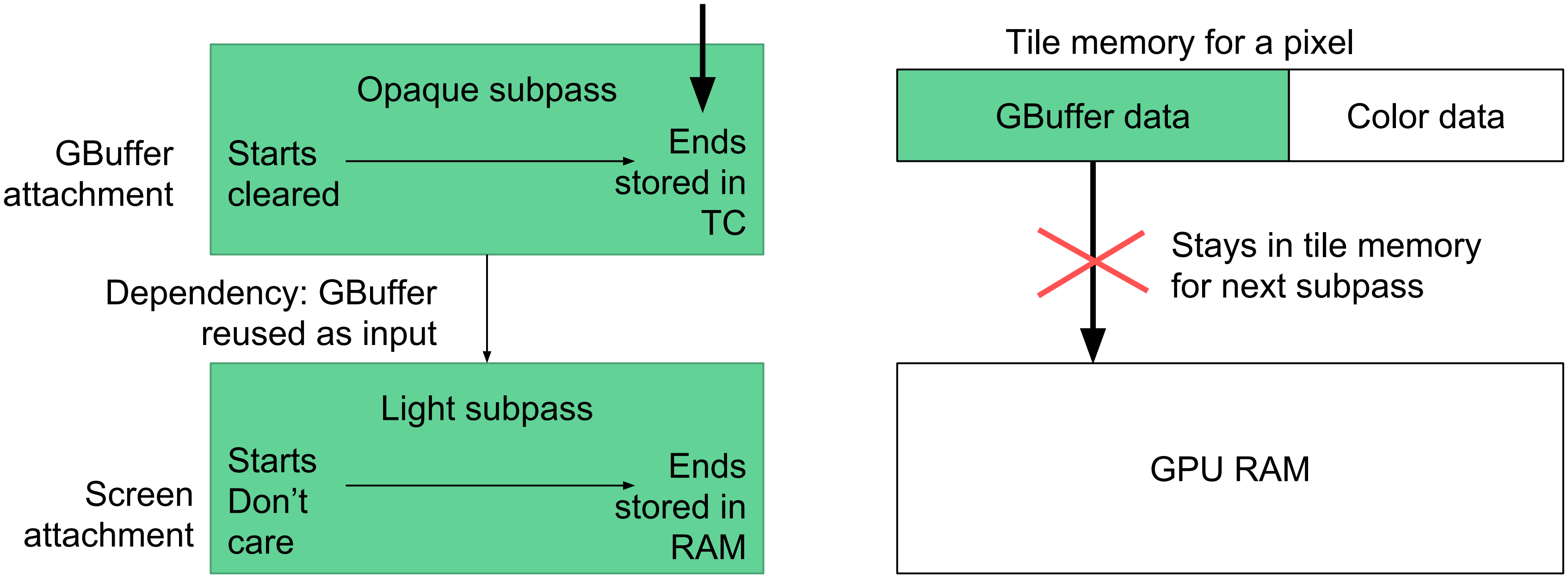
Annotated GBuffer example in Vulkan



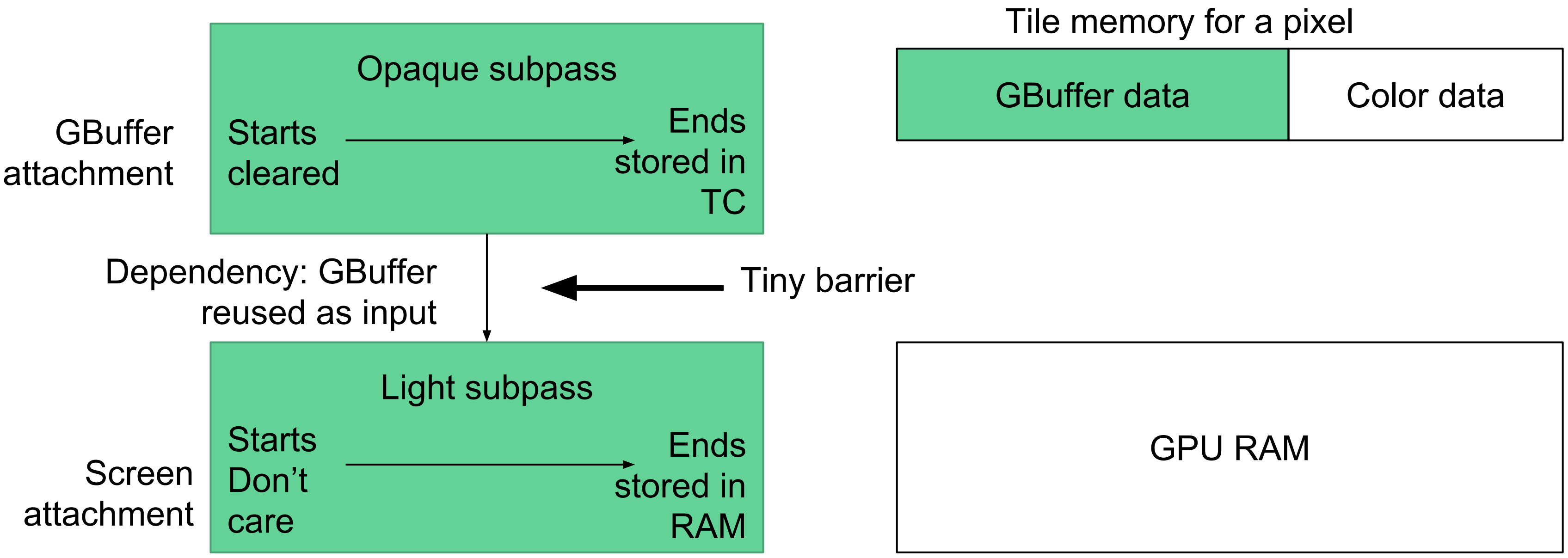
Annotated GBuffer example in Vulkan



Annotated GBuffer example in Vulkan



Annotated GBuffer example in Vulkan



GBuffer attachment

Opaque subpass

Starts cleared → Ends stored in TC

Dependency: GBuffer reused as input

Tiny barrier

Screen attachment

Light subpass

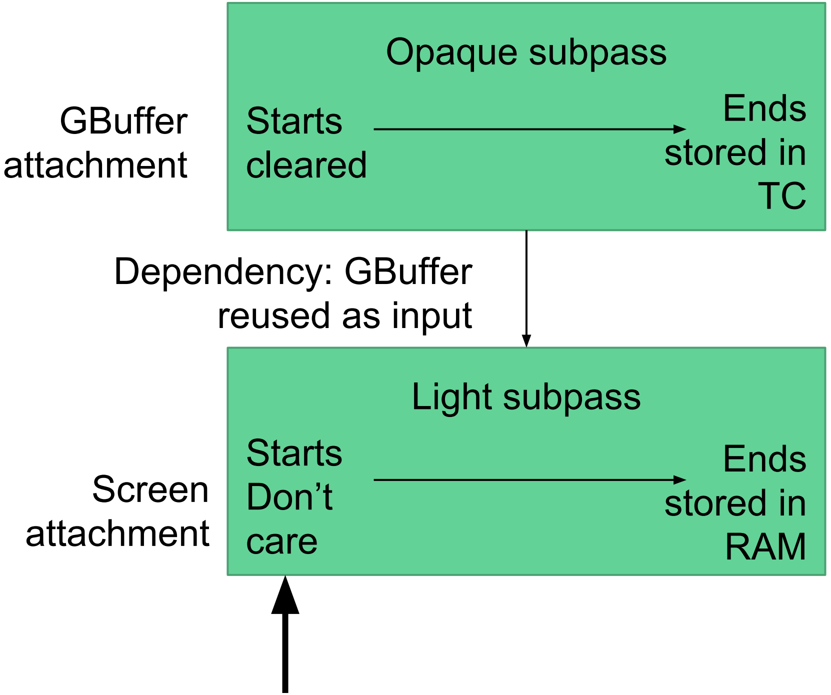
Starts Don’t care → Ends stored in RAM

Tile memory for a pixel

GBuffer data	Color data

GPU RAM

Annotated GBuffer example in Vulkan



Opaque subpass

GBuffer attachment

Starts cleared

Ends stored in TC

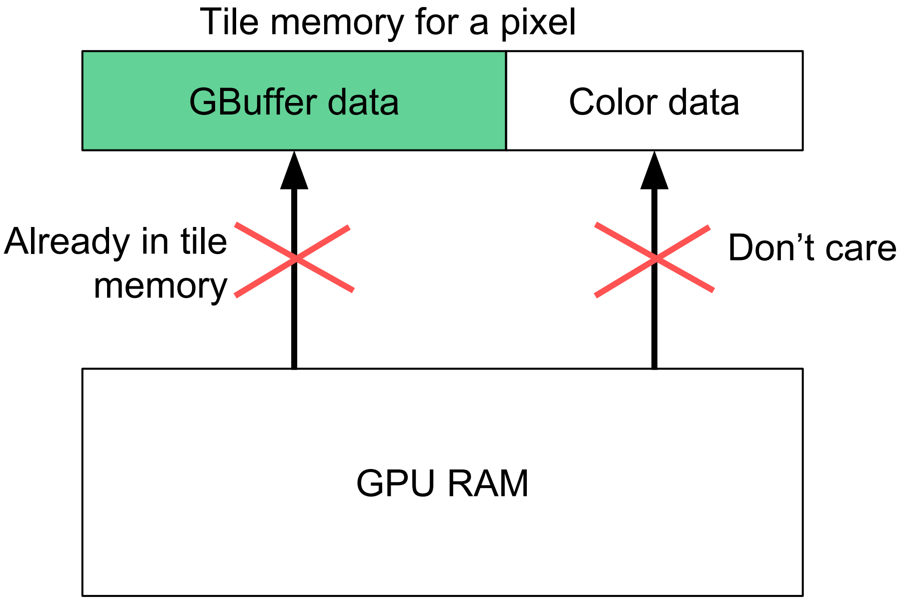
Dependency: GBuffer reused as input

Light subpass

Screen attachment

Starts Don’t care

Ends stored in RAM



Tile memory for a pixel

GBuffer data

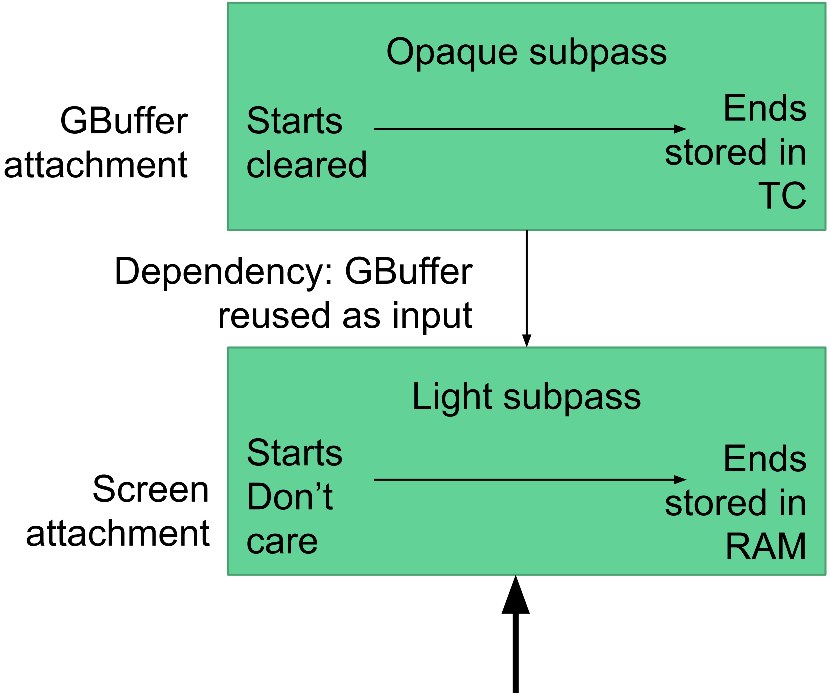
Color data

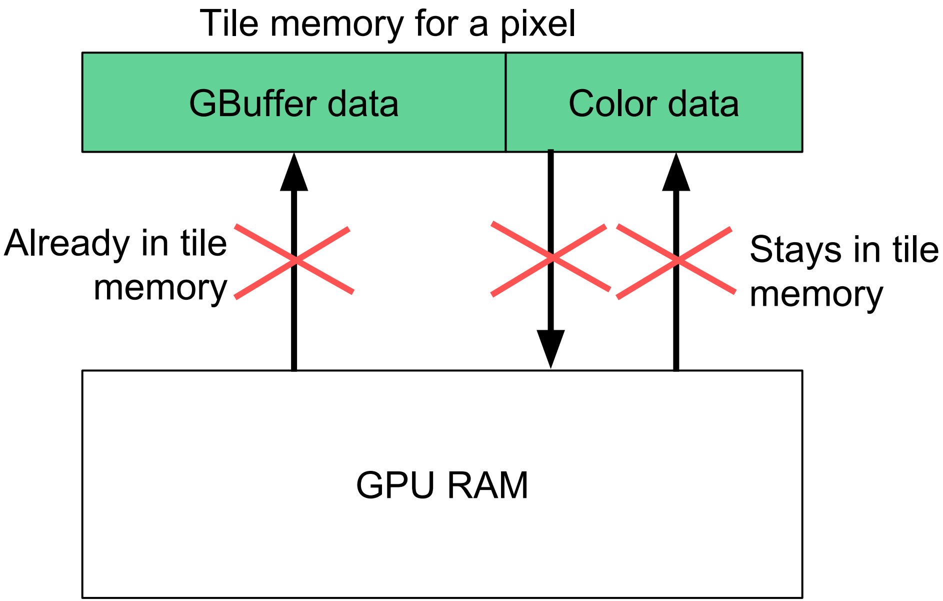
Already in tile memory

Don’t care

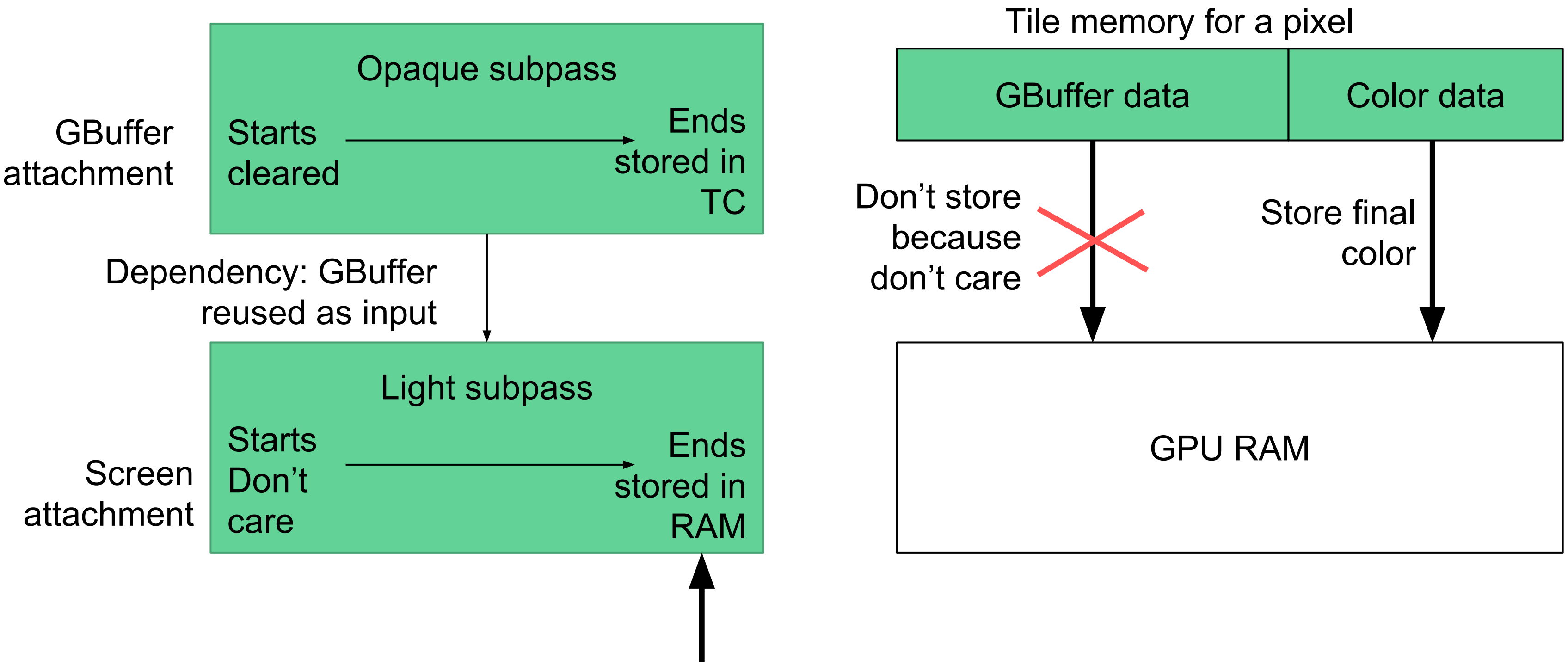
GPU RAM

Annotated GBuffer example in Vulkan





Annotated GBuffer example in Vulkan



GBuffer attachment

Opaque subpass

Starts cleared → Ends stored in TC

Dependency: GBuffer reused as input

Light subpass

Screen attachment

Starts Don't care → Ends stored in RAM

Tile memory for a pixel

GBuffer data

Color data

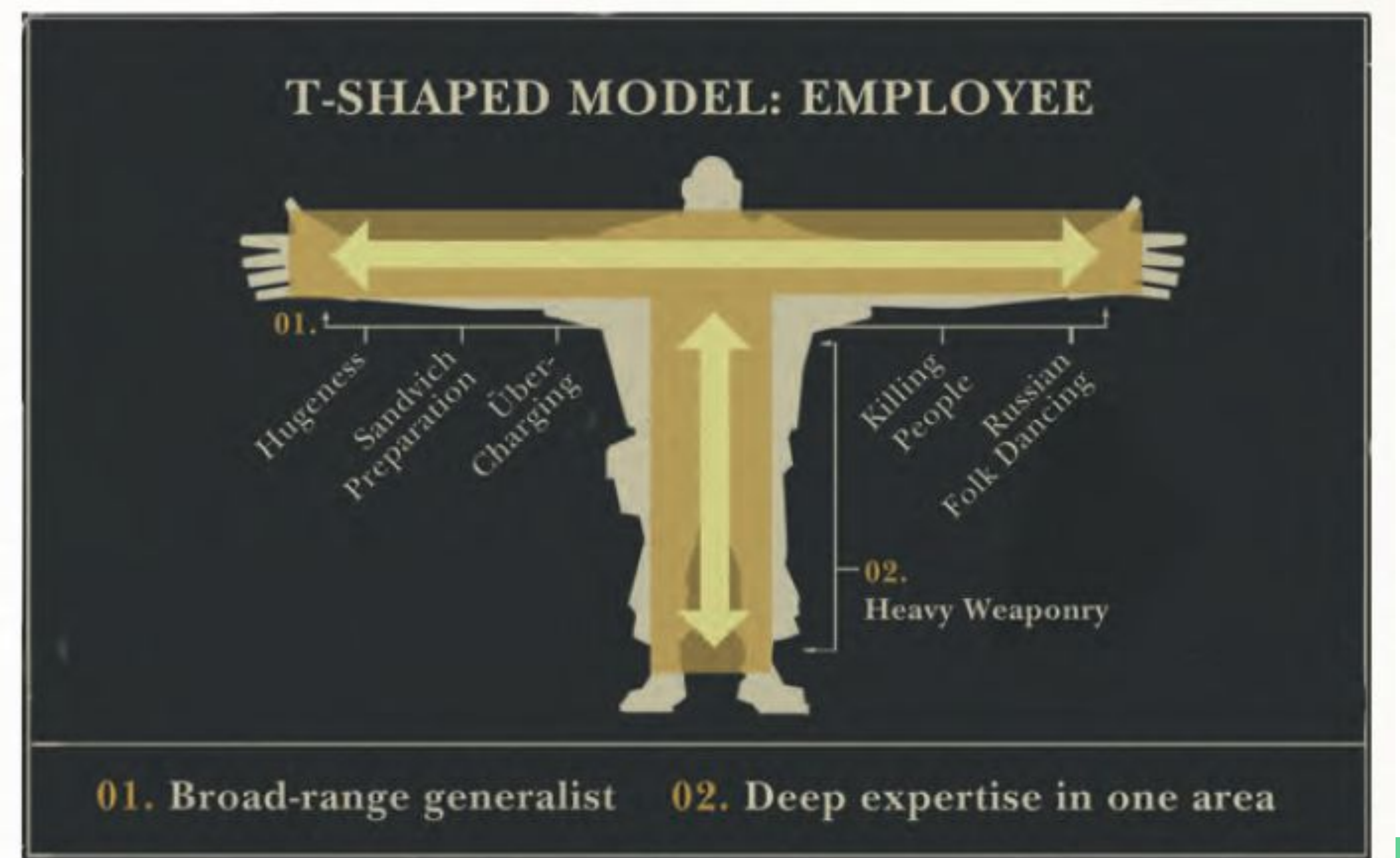
Don't store because don't care

Store final color

GPU RAM

Advice

Companies love T-Shaped people



Advice

- Ask lots of questions!
- In particular question existing APIs / concepts / practices
  - Often gives great insight
  - Grows the vertical bar of the T
- Communicate what you want and take the initiative

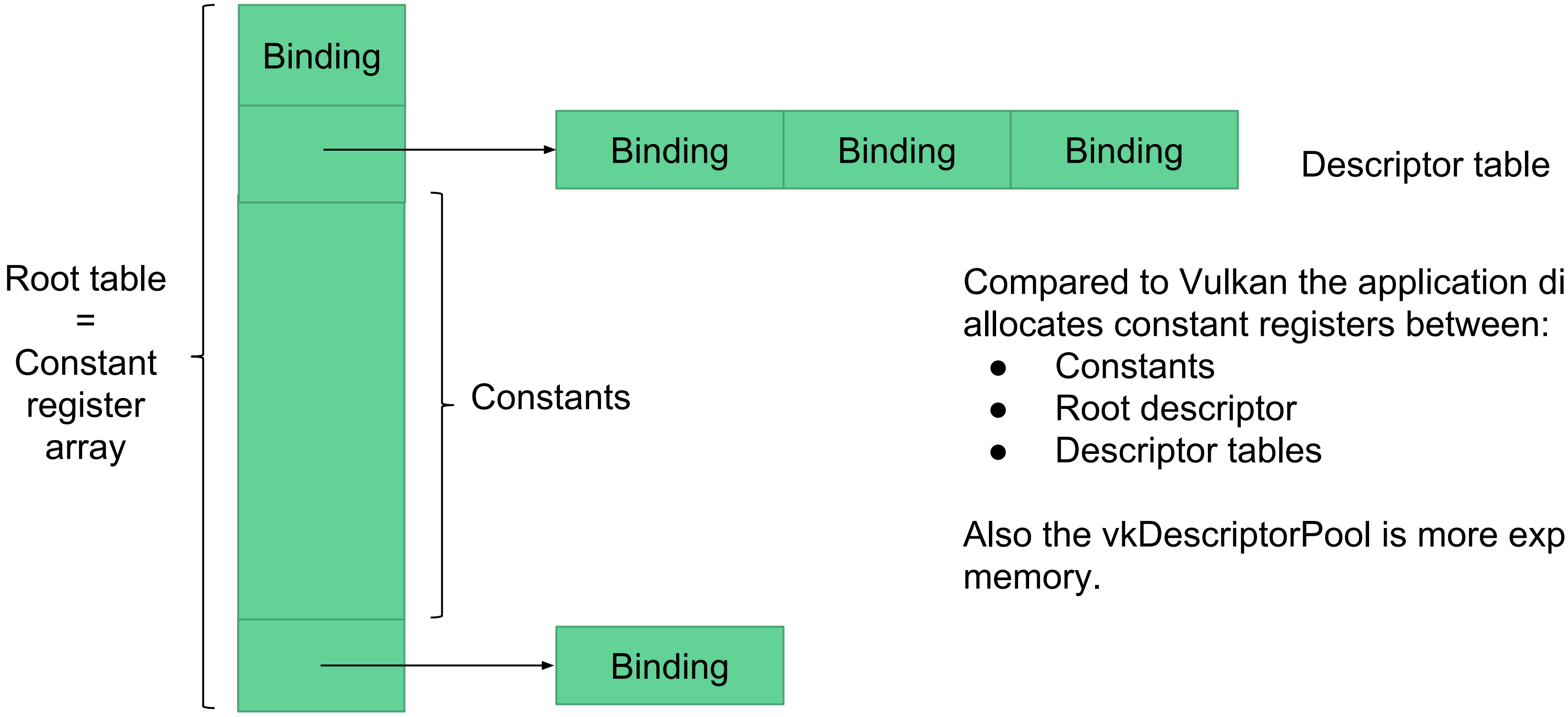
Any more questions?

Extra slides that didn’t make it in

The Vulkan binding model - Pipeline Layout

- vkPipelineLayout describes:
  - The descriptor sets and what they contain
  - The number of push constants used
- It caches “register allocation”
- Used to make things compatible between:
  - Shaders compiled in pipelines
  - Descriptor sets allocated in GPU memory

The D3D12 binding model in one slide



Compared to Vulkan the application directly allocates constant registers between:

- Constants
- Root descriptor
- Descriptor tables

Also the vkDescriptorPool is more explicitly GPU memory.