SIGGRAPH2015

Xroads of Discovery

SIGGRAPH2015
Xroads of Discovery
The 42nd International Conference and Exhibition
on Computer Graphics and Interactive Techniques

Using Next-Generation APIs on Mobile GPUs

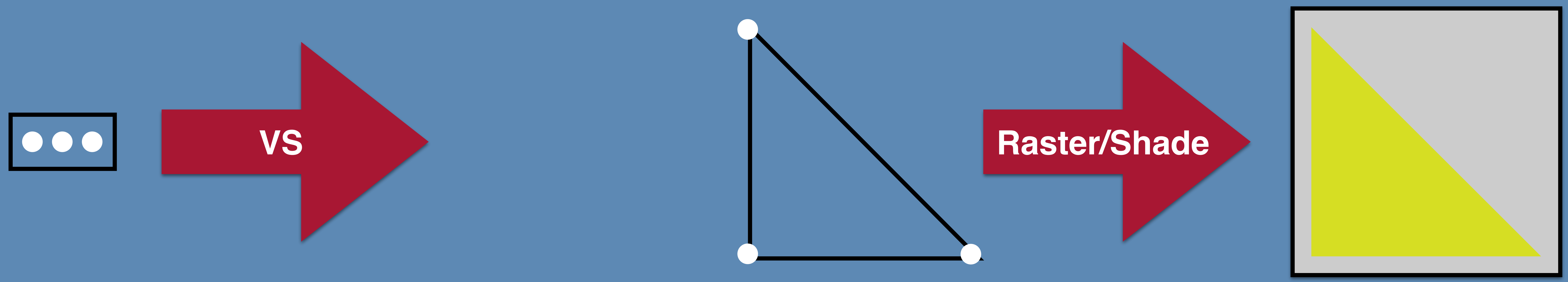
Jesse Hall
Google

~~Mobile~~Tile-Based Architectures

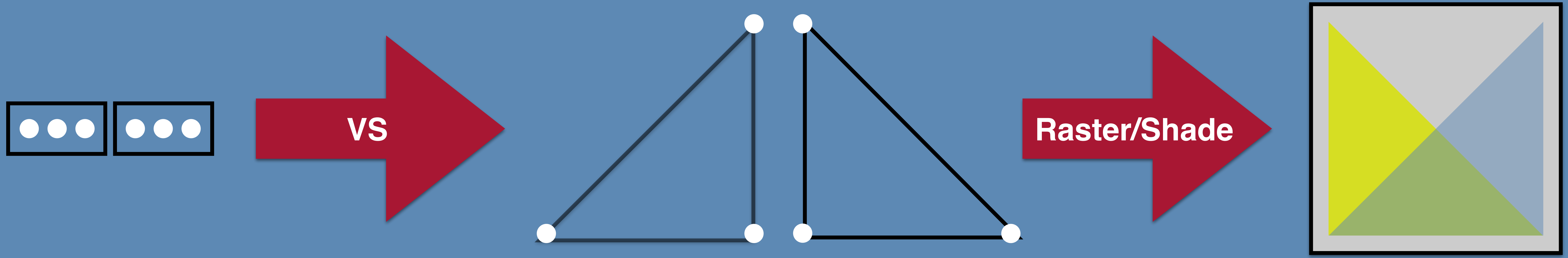
- “Mobile GPU” usually means “Tile-Based GPU”
  - Most Android and all iOS devices use tile-based rendering
  - Tiling reduces use of expensive off-chip memory bandwidth
- Vulkan and Metal make apps aware of tiling
  - Explicit render passes with load/store operations
  - Transient framebuffer attachments (Vulkan)
  - Two levels of command recording parallelism

Immediate Rendering

Immediate Rendering



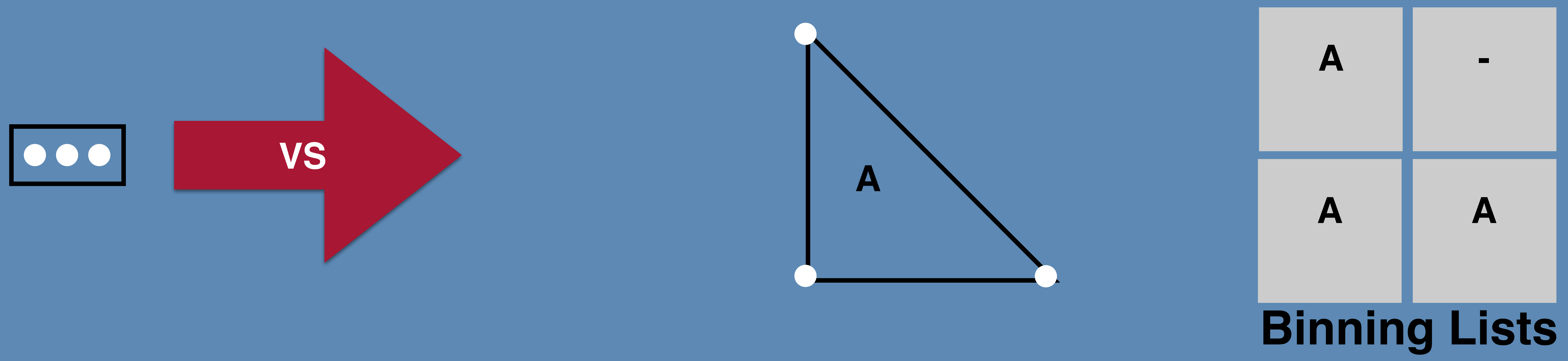
Immediate Rendering



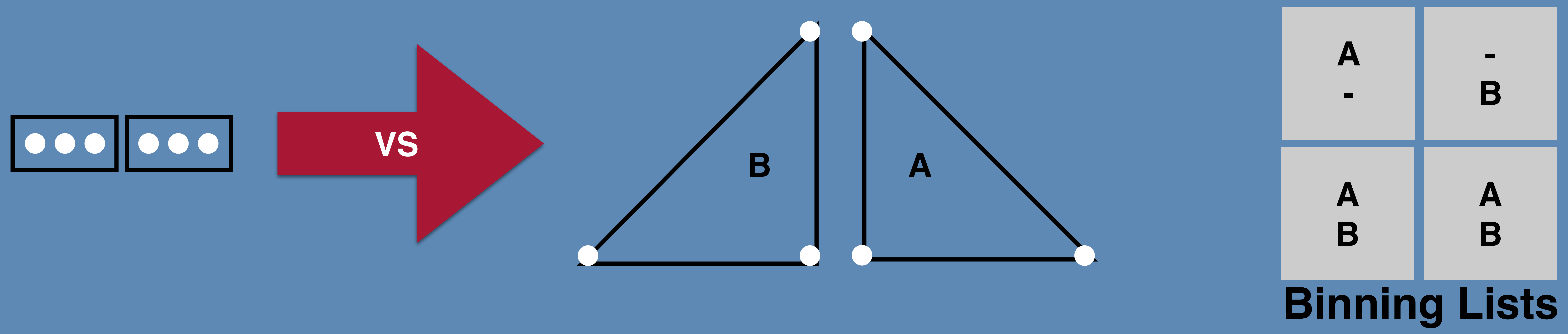
Tile-Based Rendering

Tile-Based Rendering

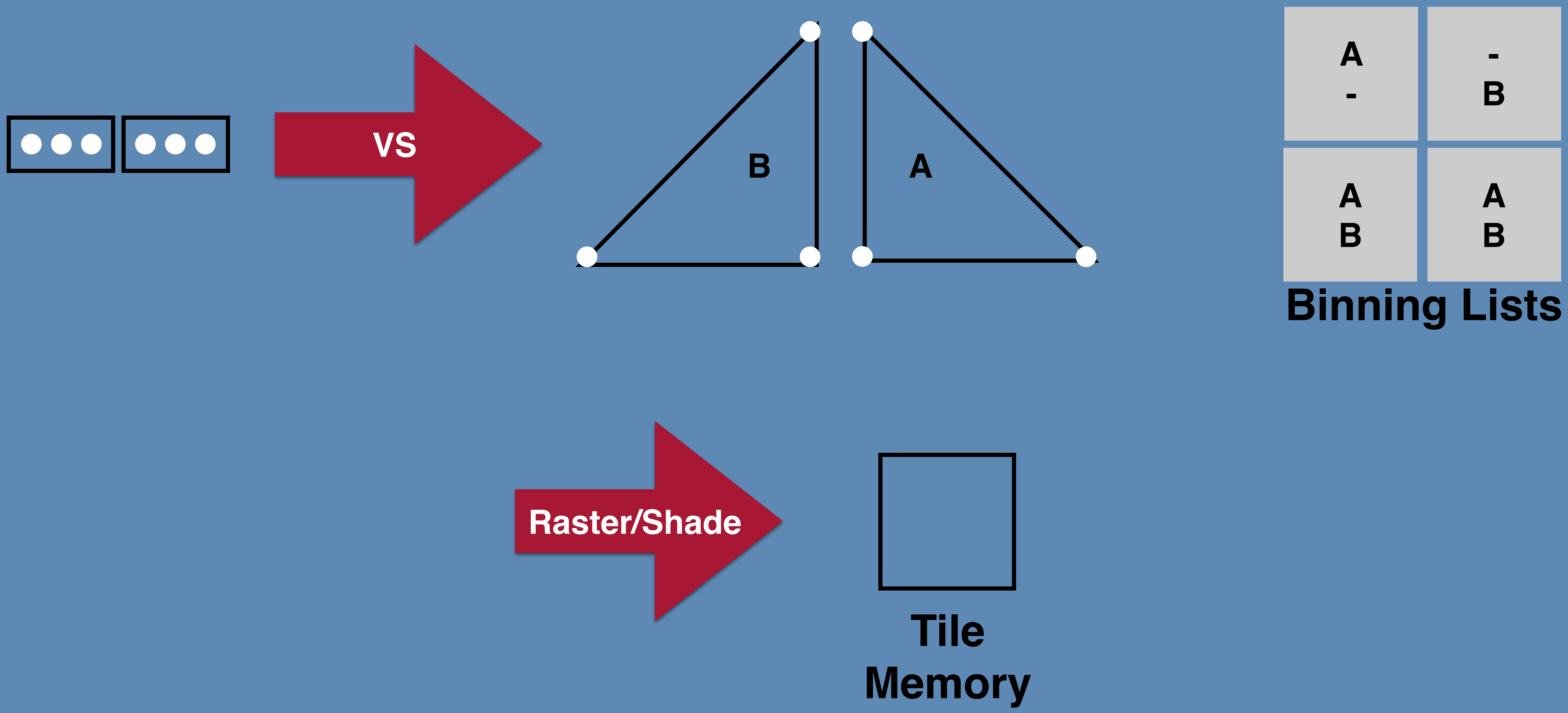
Tile-Based Rendering



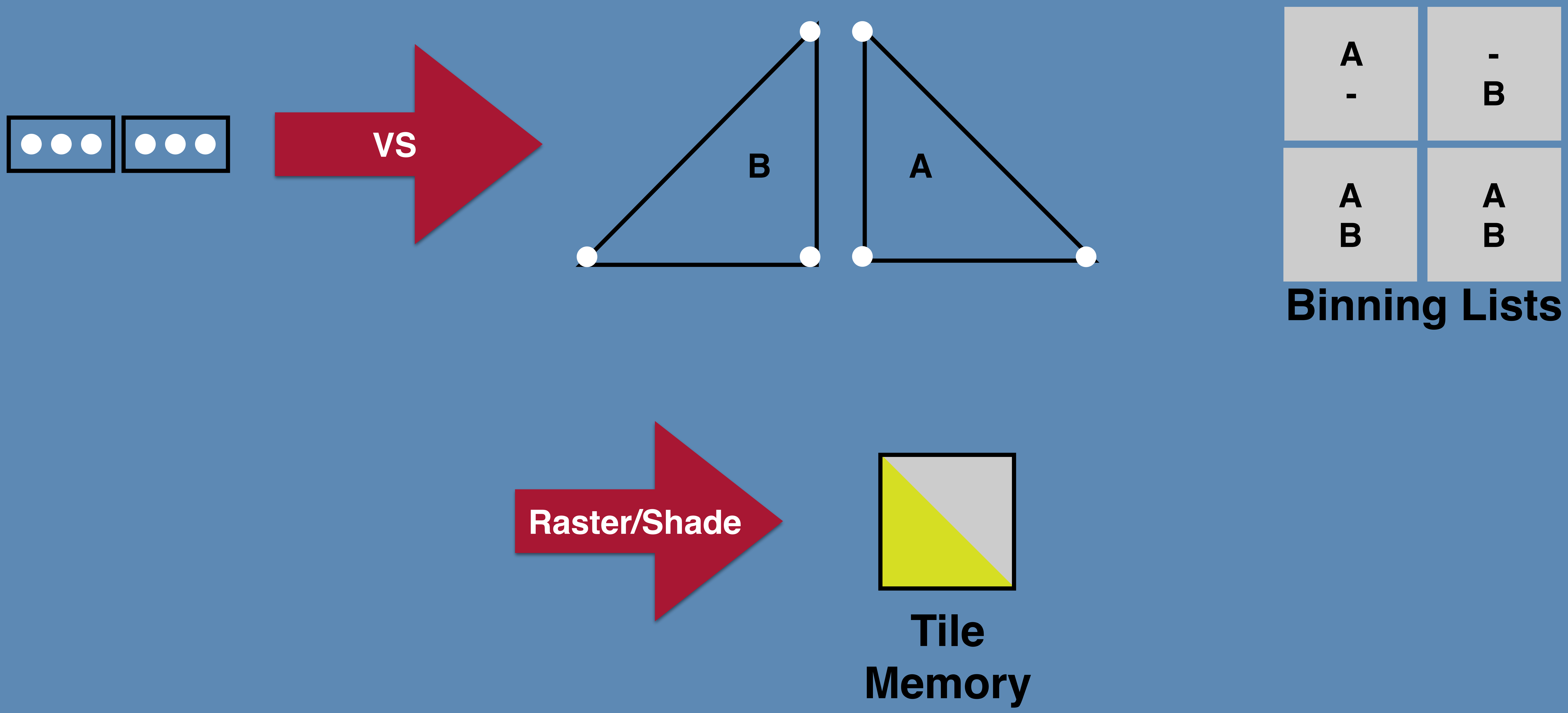
Tile-Based Rendering



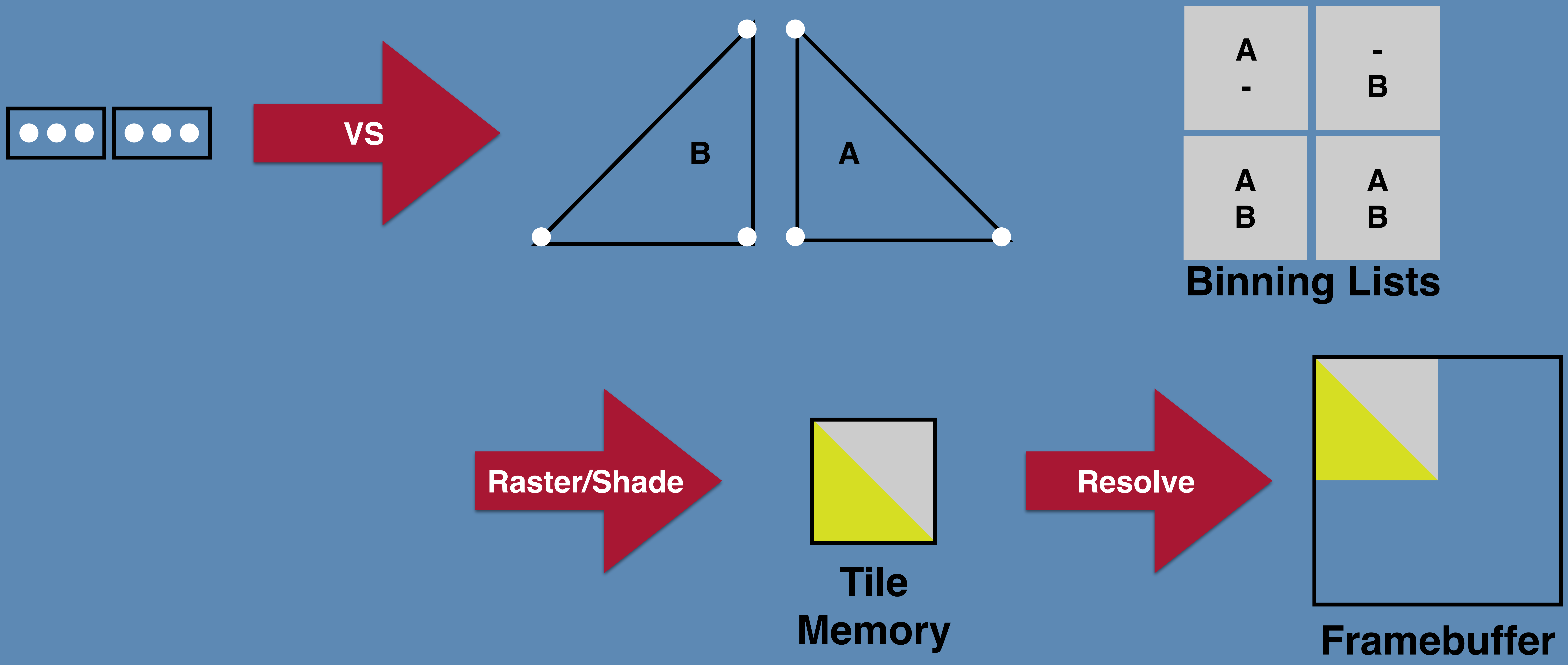
Tile-Based Rendering



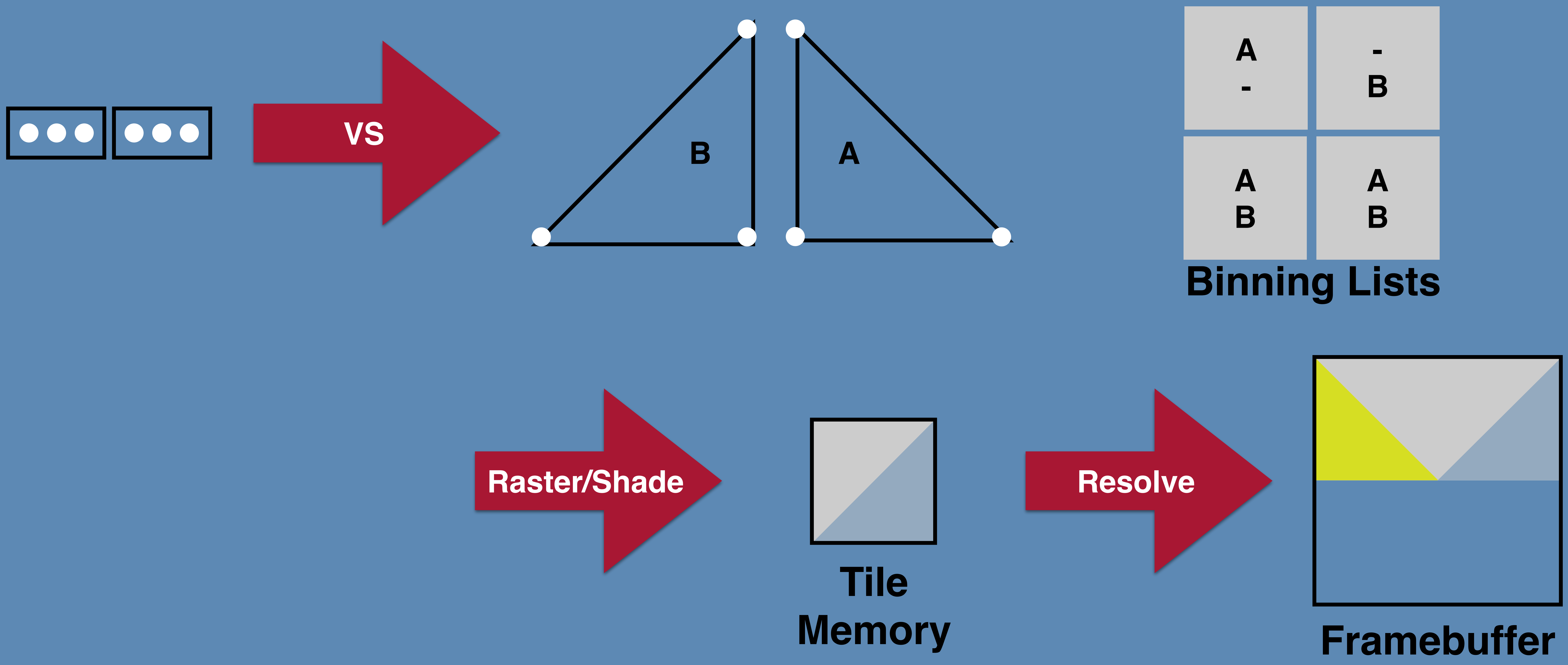
Tile-Based Rendering



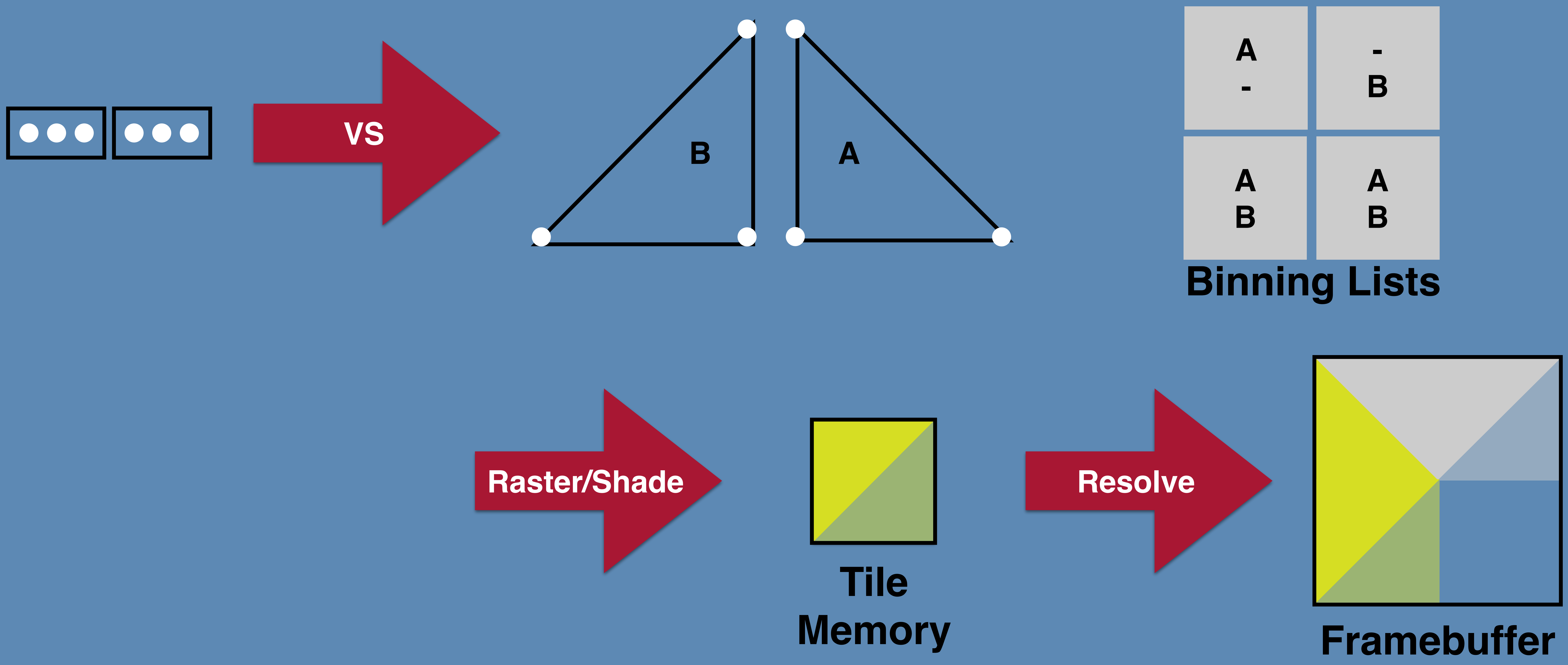
Tile-Based Rendering



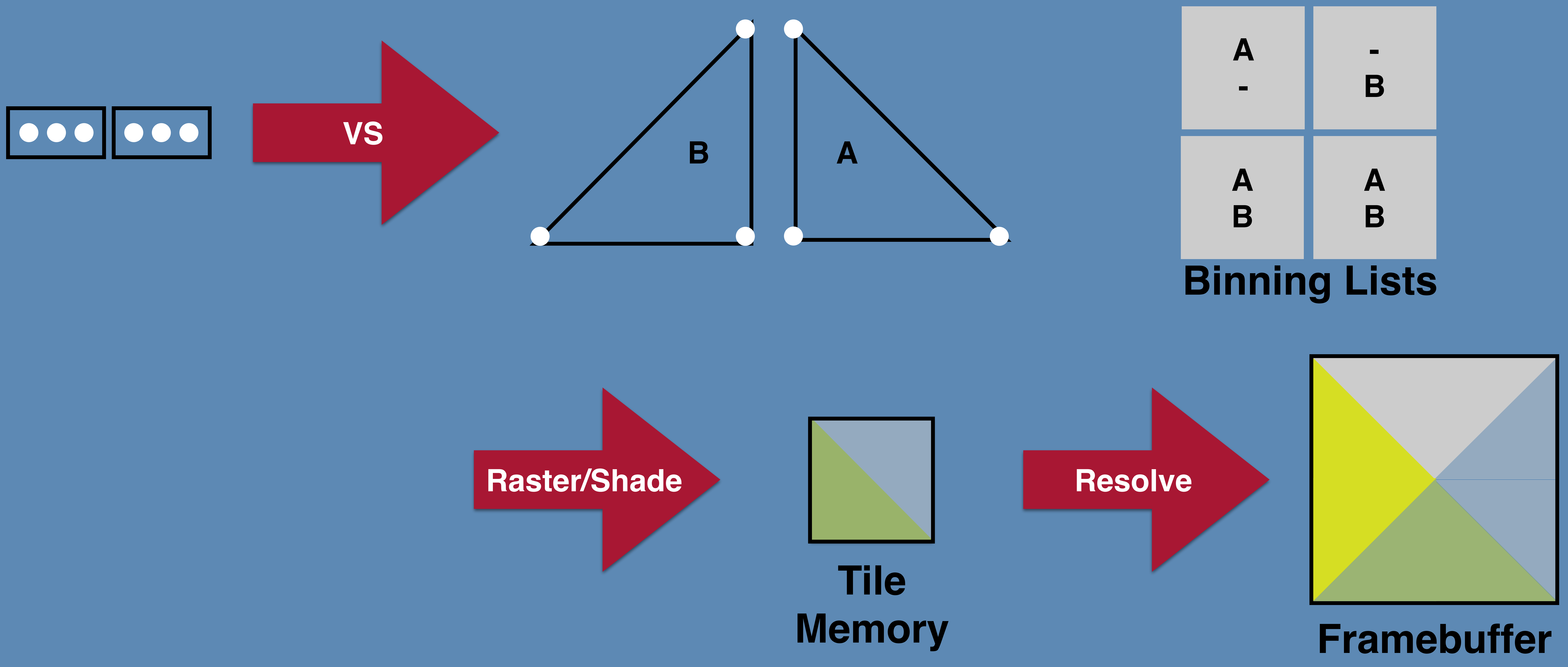
Tile-Based Rendering



Tile-Based Rendering



Tile-Based Rendering



Tile Memory Initialization

- **Load**: Load attachment contents into tile memory
  `VK_ATTACHMENT_LOAD_OP_LOAD | MTLLoadActionLoad`
- **Clear**: Initialize tile memory with solid color
  `VK_ATTACHMENT_LOAD_OP_CLEAR | MTLLoadActionClear`
- **Don't Care**: Tile memory is uninitialized, use when opaque color will be written to every sample
  `VK_ATTACHMENT_LOAD_OP_DONT_CARE | MTLLoadActionDontCare`

Tile Memory Disposition

- **Store**: Commit tile memory to memory
  `VK_ATTACHMENT_STORE_OP_STORE | MTLStoreActionStore`
- **Don't Care:** Attachment contents are undefined
  `VK_ATTACHMENT_STORE_OP_DONT_CARE | MTLStoreActionDontCare`

Tile Memory Multisample Resolve

- Multisample resolve occurs at end of render pass
  - Color attachments have a multisample `VkImage | MTLTexture`
  - And an optional single-sample resolve `VkImage | MTLTexture`
  - Metal: Use `MTLStoreActionMultisampleResolve` for attachment

Transient Attachments

- Depth images are typically cleared & discarded
  - Tilers never need to read or write off-chip memory for these!
- In Vulkan, can use a lazily-committed VkImage
  - Must always attach a VkImage that has VkDeviceMemory
  - Create VkImage with `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT`
  - But VkDeviceMemory may not have committed physical memory
  - Tilers will often be able to avoid committing memory

Multi-threading a Single Pass

- Passes are the unit of scheduling on some tilers
  - Must begin and end in the same command buffer
  - But we need to be able to parallelize recording of draws in a pass…
- Vulkan and Metal offer second-level parallelism
  - Vulkan: `VK_CMD_BUFFER_LEVEL_SECONDARY`
  - Metal: `MTLParallelRenderCommandEncoder`
  - Use when you need to issue draws in parallel within a single pass

Vulkan: Serial Pass

Vulkan: Serial Pass

Thread 1:

Vulkan: Serial Pass

Thread 1:

```
vkBeginCommandBuffer(PRIMARY)
```

Vulkan: Serial Pass

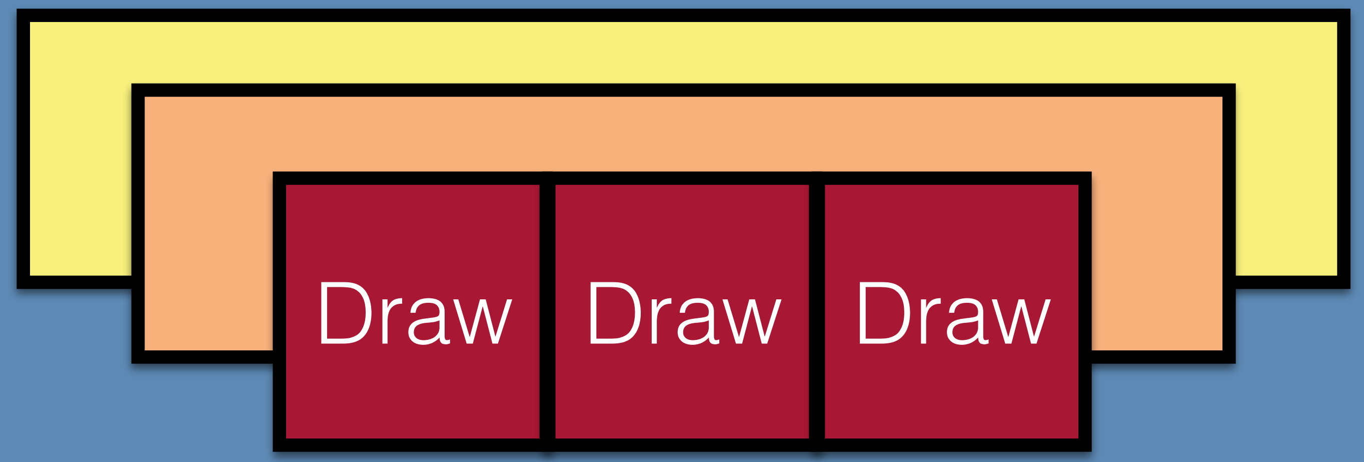
Thread 1:

```
vkBeginCommandBuffer(PRIMARY)
vkCmdBeginRenderPass
```

Vulkan: Serial Pass

Thread 1:

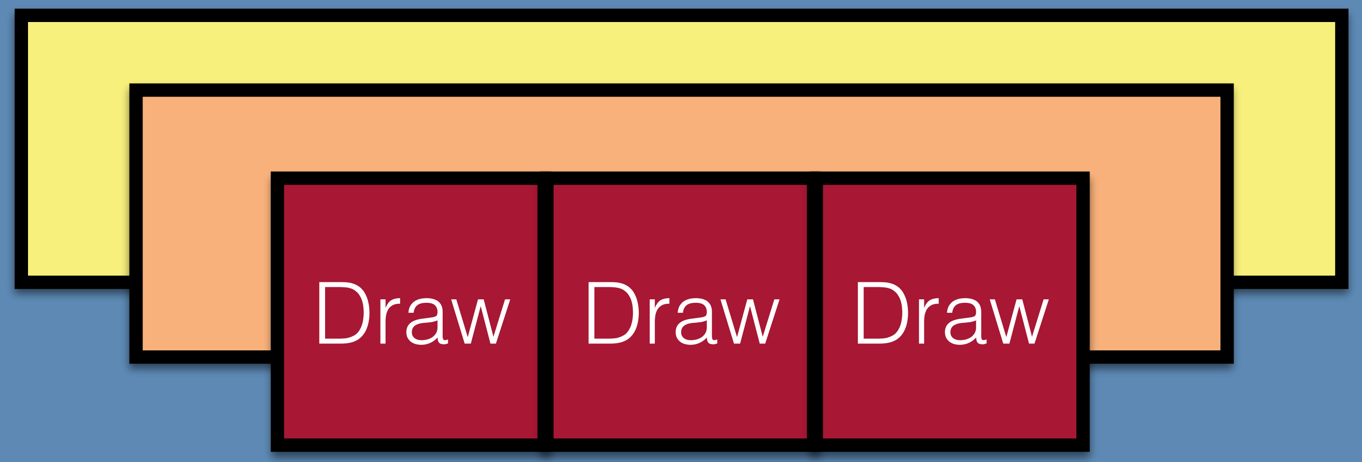
```
vkBeginCommandBuffer(PRIMARY)
vkCmdBeginRenderPass
vkCmdDraw [x3]
```



Vulkan: Serial Pass

Thread 1:

```
vkBeginCommandBuffer(PRIMARY)
vkCmdBeginRenderPass
vkCmdDraw [x3]
vkCmdEndRenderPass
vkEndCommandBuffer
```



Vulkan: Parallel Pass

Vulkan: Parallel Pass

Thread 1:

Threads 2..4:

Vulkan: Parallel Pass

Thread 1:

`vkBeginCommandBuffer(PRIMARY)`

Threads 2..4:

`vkBeginCommandBuffer(SECONDARY)`

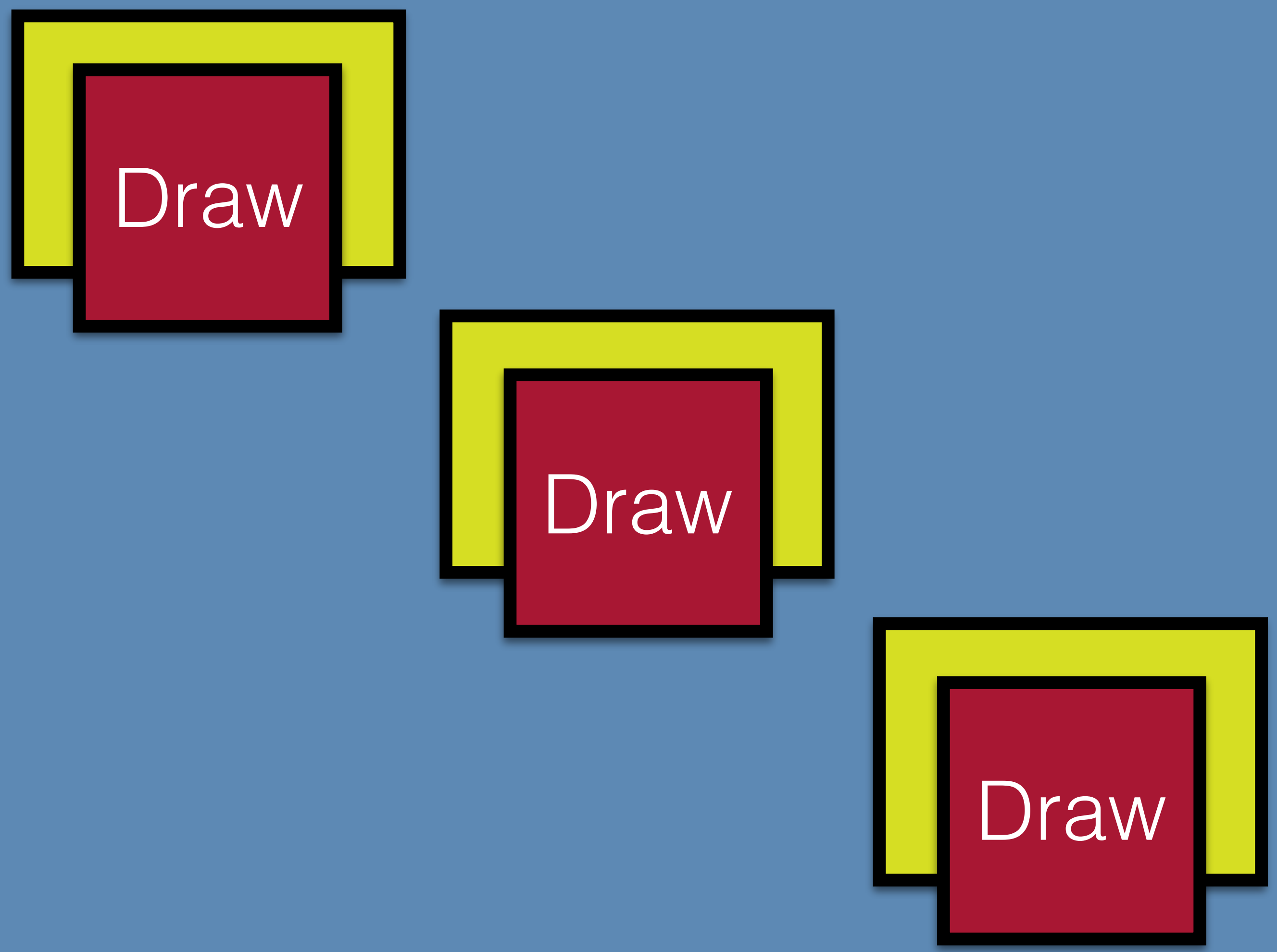
Vulkan: Parallel Pass

Thread 1:

```
vkBeginCommandBuffer(PRIMARY)
vkCmdBeginRenderPass
```

Threads 2..4:

```
vkBeginCommandBuffer(SECONDARY)
vkCmdDraw
```



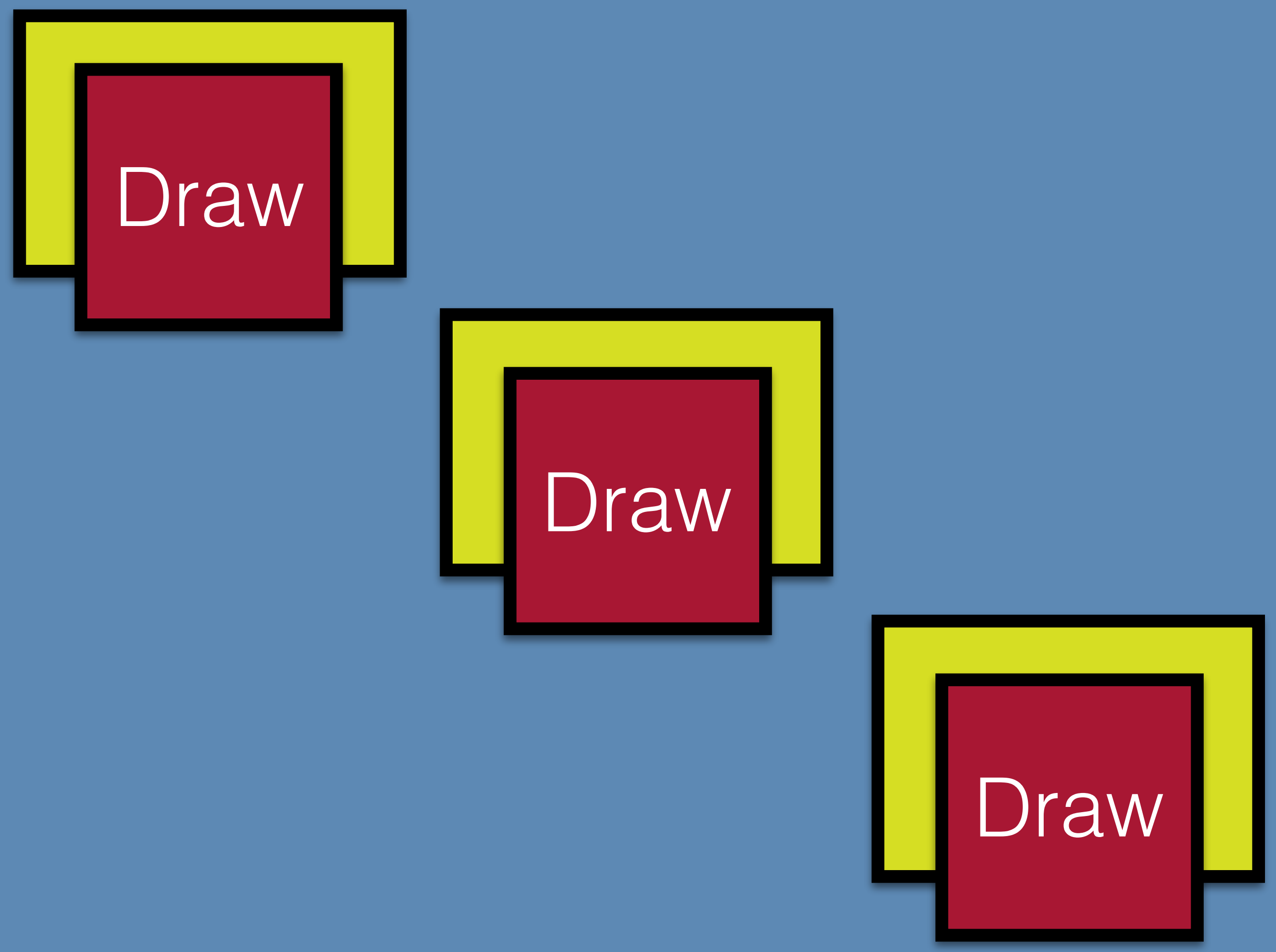
Vulkan: Parallel Pass

Thread 1:

```
vkBeginCommandBuffer(PRIMARY)
vkCmdBeginRenderPass
```

Threads 2..4:

```
vkBeginCommandBuffer(SECONDARY)
vkCmdDraw
vkEndCommandBuffer
```



Vulkan: Parallel Pass

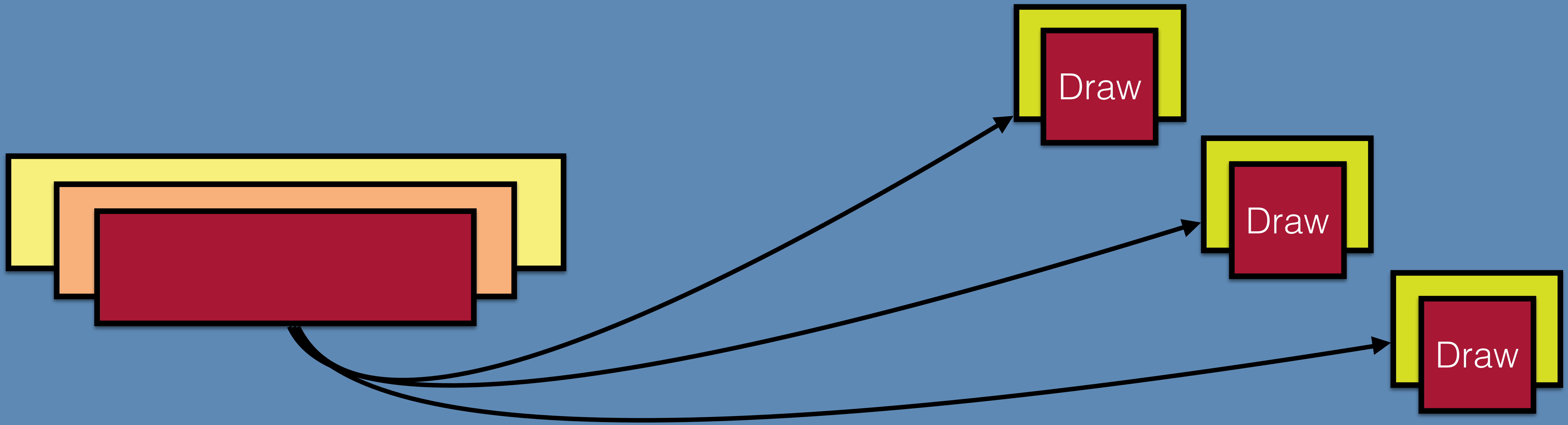
Thread 1:

```
vkBeginCommandBuffer(PRIMARY)
vkCmdBeginRenderPass

vkCmdExecuteSecondaryCommands
```

Threads 2..4:

```
vkBeginCommandBuffer(SECONDARY)
vkCmdDraw
vkEndCommandBuffer
```



Vulkan: Parallel Pass

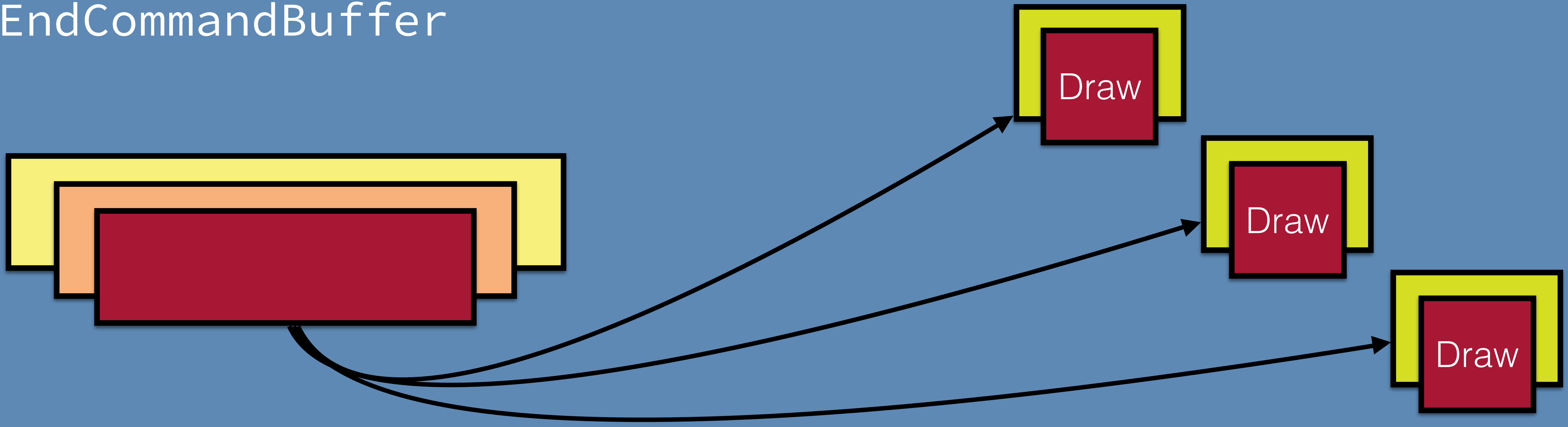
Thread 1:

```
vkBeginCommandBuffer(PRIMARY)
vkCmdBeginRenderPass

vkCmdExecuteSecondaryCommands
vkCmdEndRenderPass
vkEndCommandBuffer
```

Threads 2..4:

```
vkBeginCommandBuffer(SECONDARY)
vkCmdDraw
vkEndCommandBuffer
```



Metal: Serial Pass

Metal: Serial Pass

Thread 1:

Metal: Serial Pass

Thread 1:

```
cb = queue.commandBuffer()
```

Metal: Serial Pass

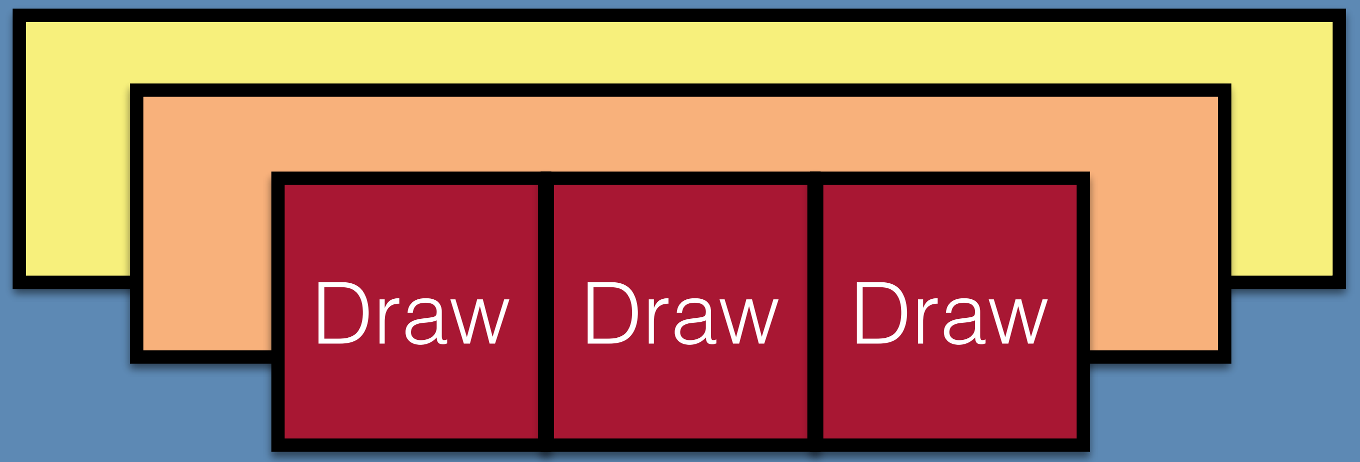
Thread 1:

```
cb = queue.commandBuffer()
rce = cb.renderCommandEncoder()
```

Metal: Serial Pass

Thread 1:

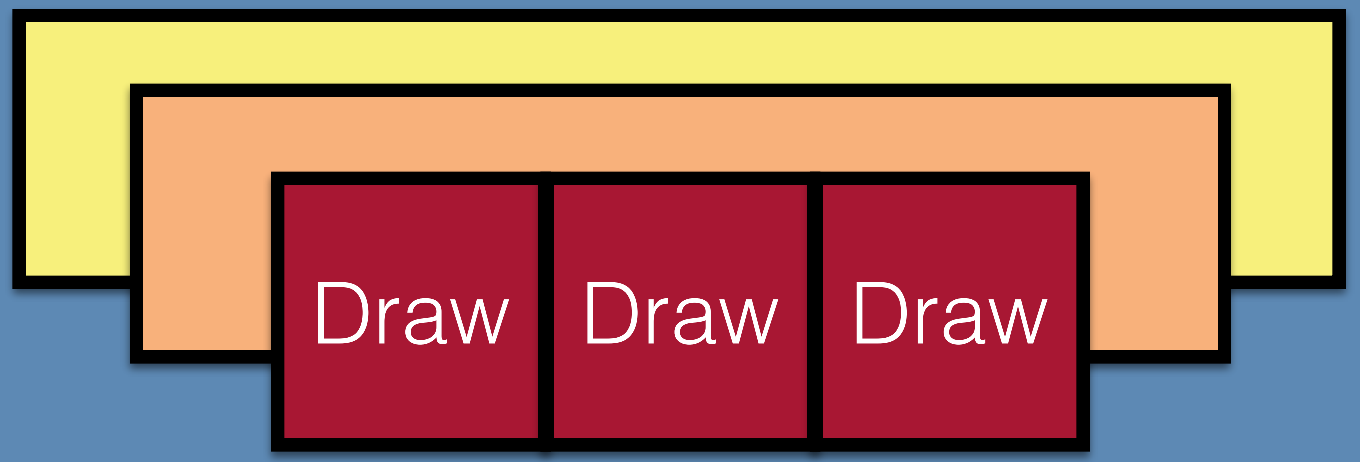
```
cb = queue.commandBuffer()
rce = cb.renderCommandEncoder()
rce.drawPrimitives() [x3]
```



Metal: Serial Pass

Thread 1:

```
cb = queue.commandBuffer()
rce = cb.renderCommandEncoder()
rce.drawPrimitives() [x3]
rce.endEncoding()
cb.commit()
```



Metal: Parallel Pass

Metal: Parallel Pass

Thread 1:

Threads 2..4:

Metal: Parallel Pass

Thread 1:

```
cb = queue.commandBuffer()
```

Threads 2..4:

Metal: Parallel Pass

Thread 1:

```
cb = queue.commandBuffer()
prce = cb.parallelRenderCommandEncoder()
```

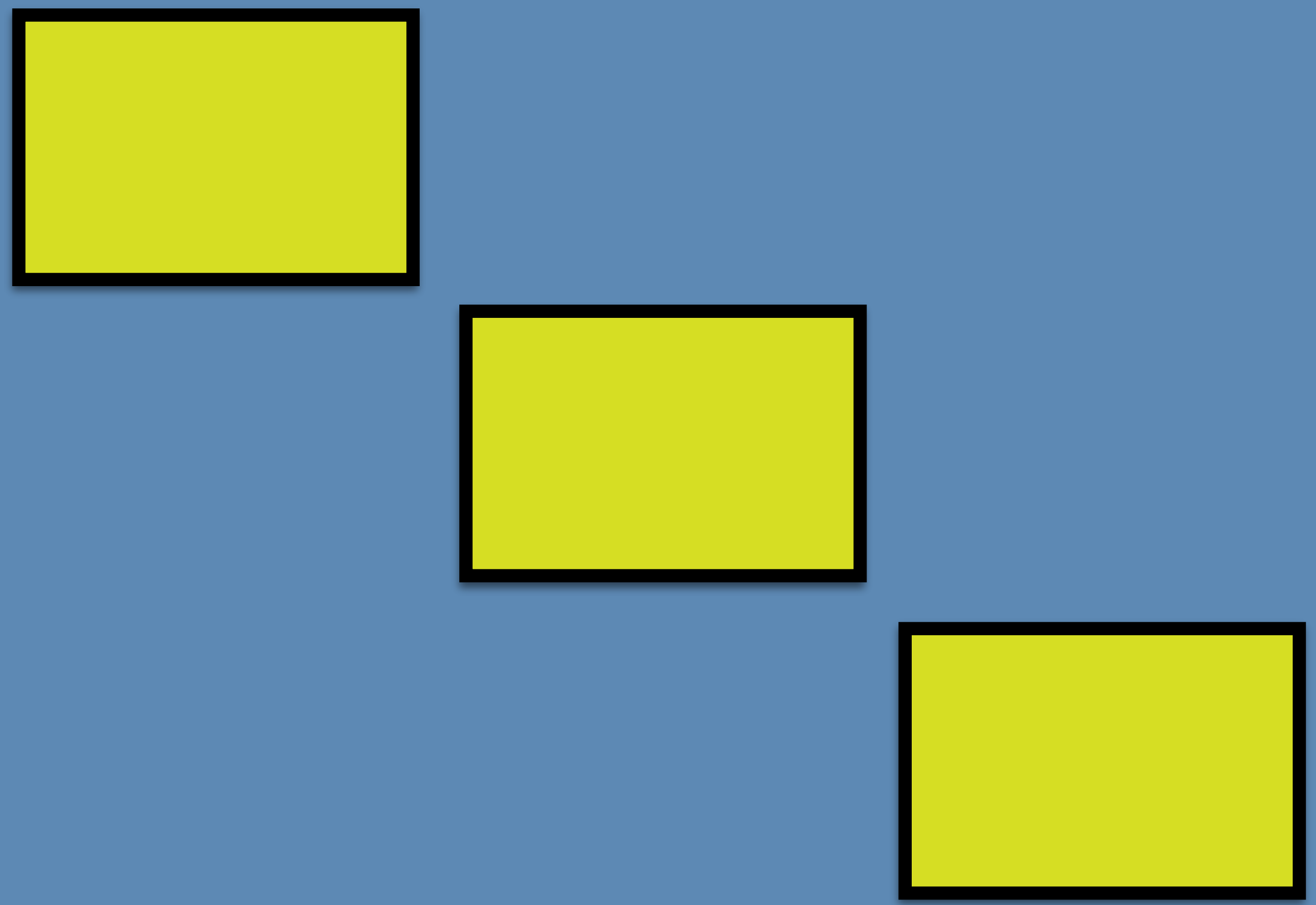
Threads 2..4:

Metal: Parallel Pass

Thread 1:

Threads 2..4:

```
cb = queue.commandBuffer()
prce = cb.parallelRenderCommandEncoder()
                                          rce = prce.renderCommandEncoder()
```



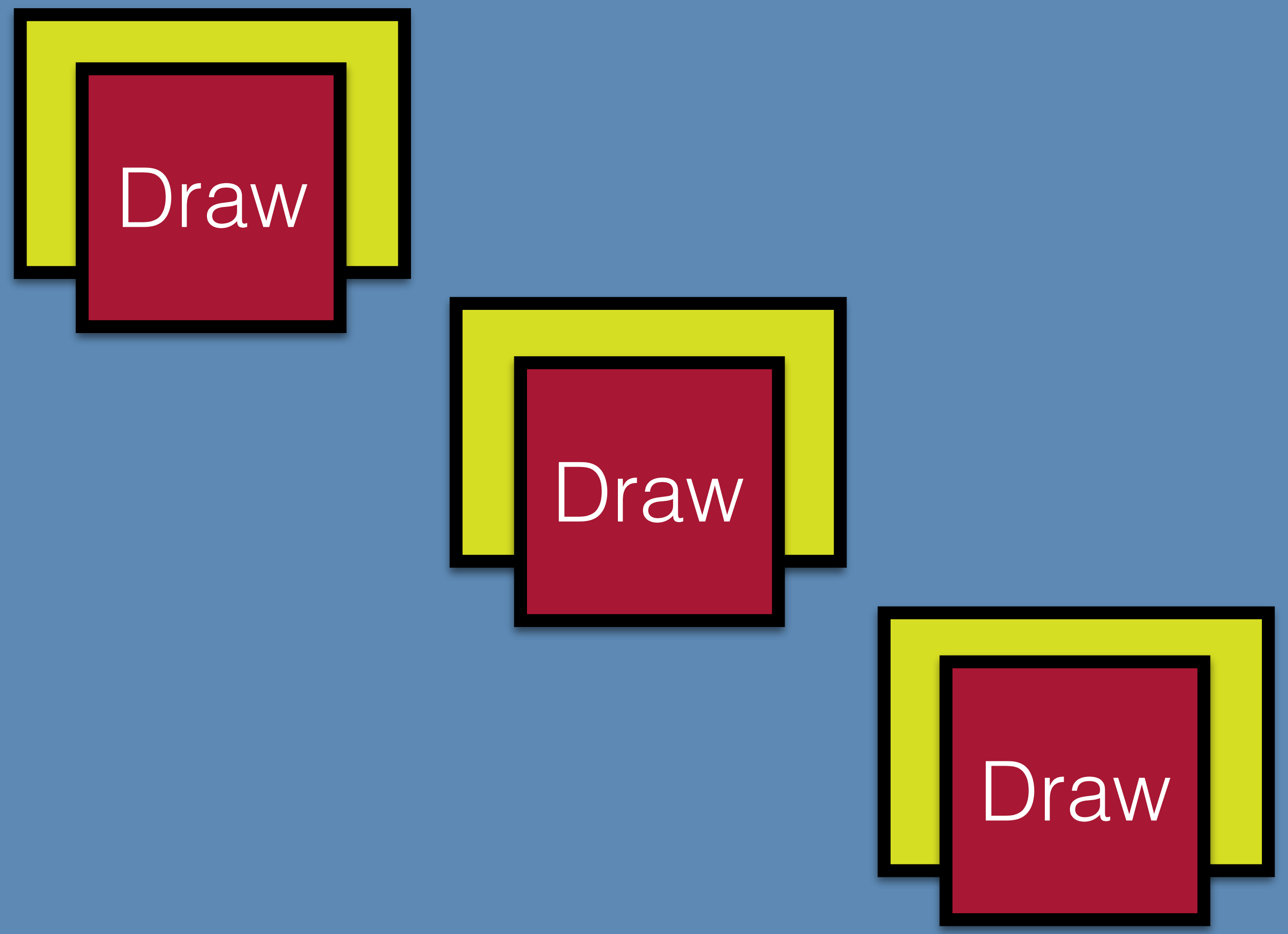
Metal: Parallel Pass

Thread 1:

```
cb = queue.commandBuffer()
prce = cb.parallelRenderCommandEncoder()
```

Threads 2..4:

```
rce = prce.renderCommandEncoder()
rce.drawPrimitives()
rce.endEncoding()
```



Metal: Parallel Pass

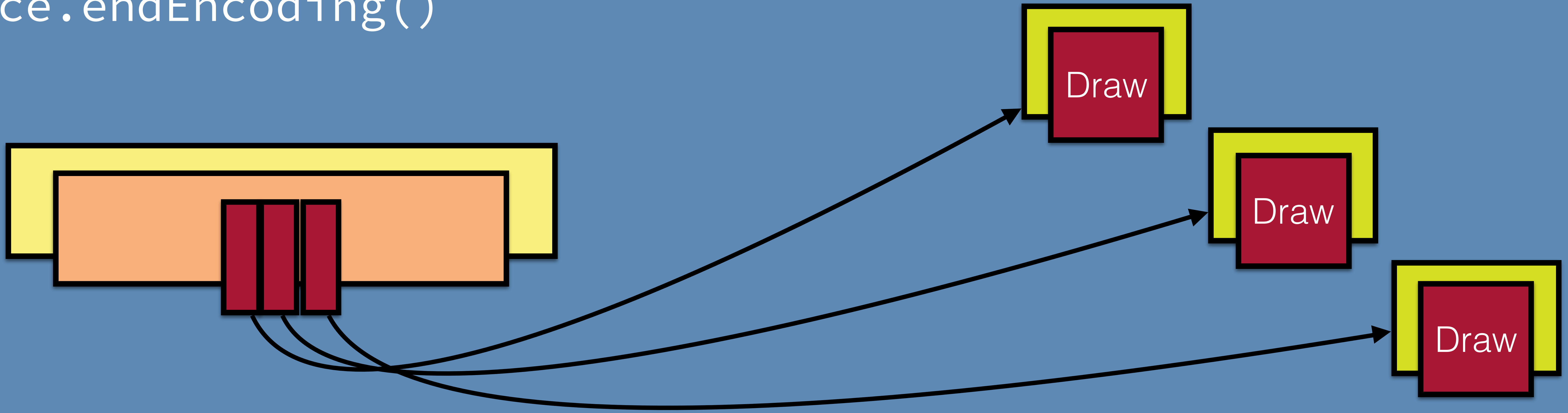
Thread 1:

```
cb = queue.commandBuffer()
prce = cb.parallelRenderCommandEncoder()
```

Threads 2..4:

```
rce = prce.renderCommandEncoder()
rce.drawPrimitives()
rce.endEncoding()
```

```
prce.endEncoding()
```



Metal: Parallel Pass

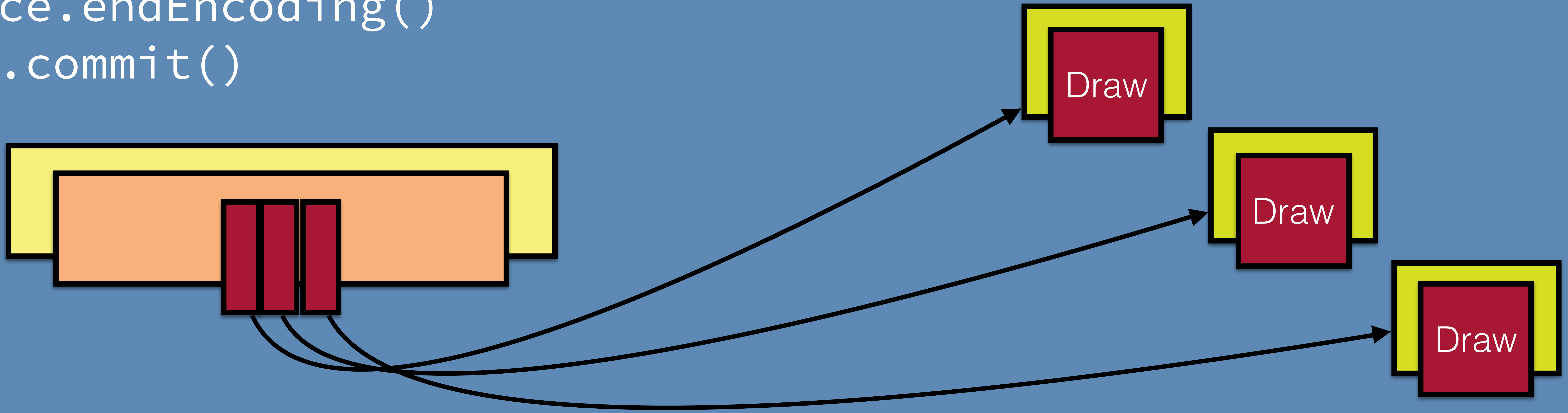
Thread 1:

```
cb = queue.commandBuffer()
prce = cb.parallelRenderCommandEncoder()
```

Threads 2..4:

```
rce = prce.renderCommandEncoder()
rce.drawPrimitives()
rce.endEncoding()
```

```
prce.endEncoding()
cb.commit()
```



Metal: Parallel Pass

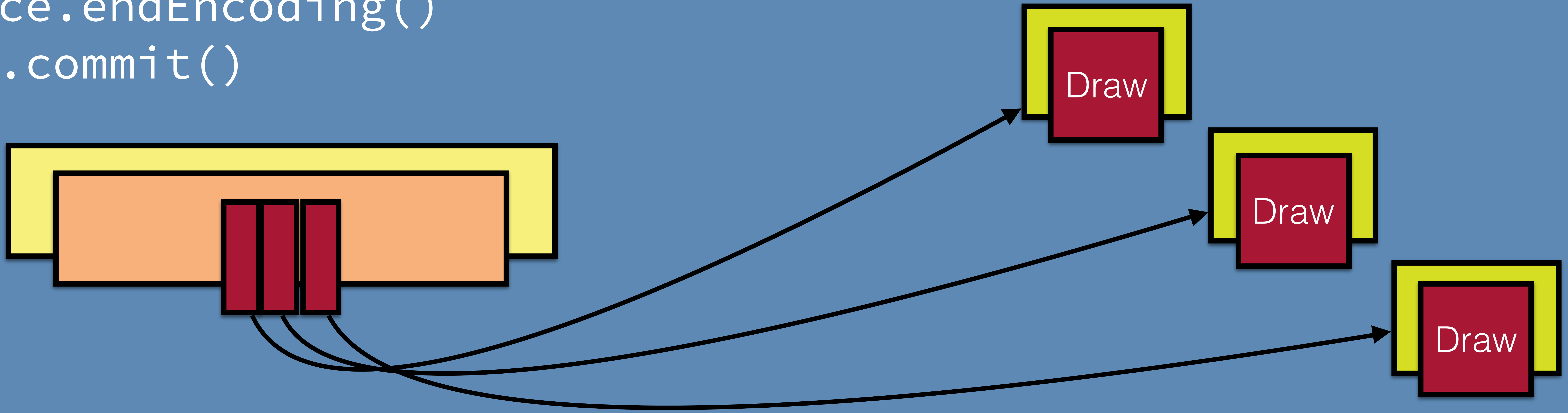
Thread 1:

```
cb = queue.commandBuffer()
prce = cb.parallelRenderCommandEncoder()
```

Threads 2..4:

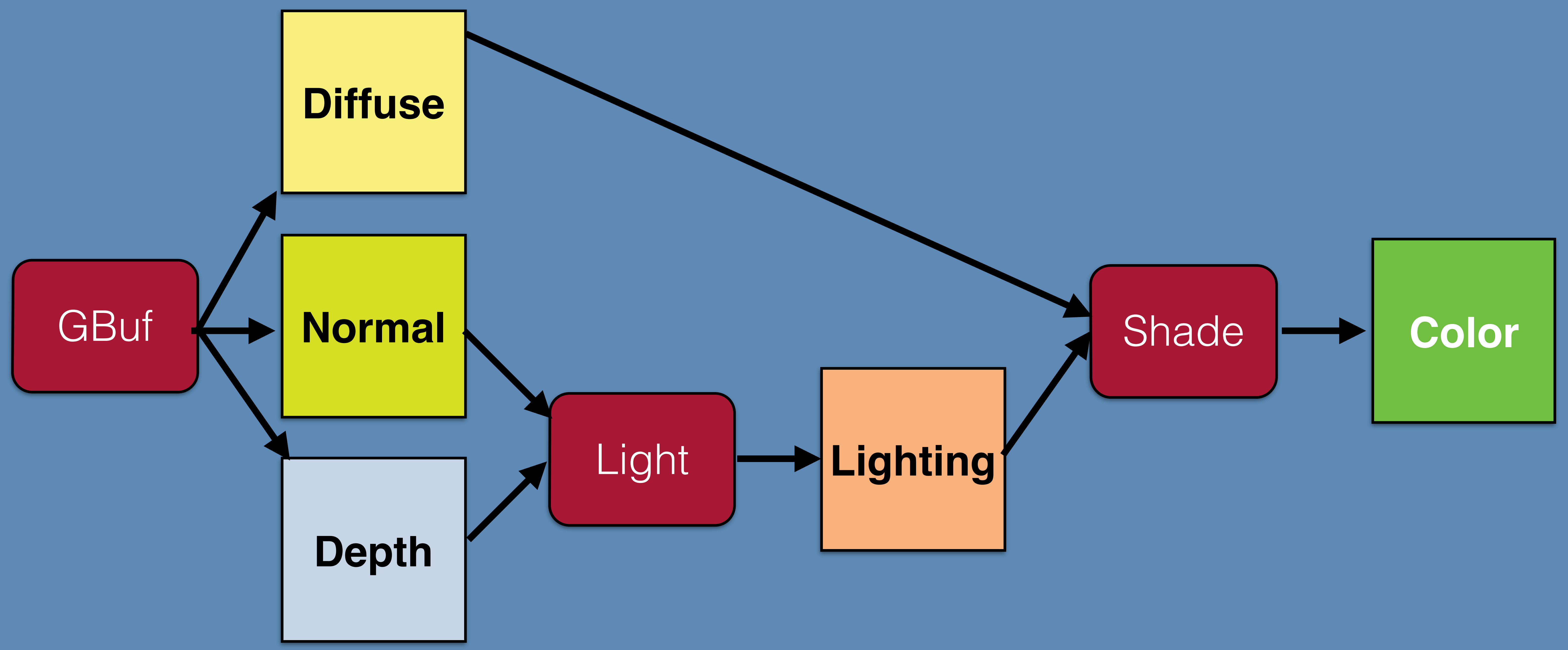
```
rce = prce.renderCommandEncoder()
rce.drawPrimitives()
rce.endEncoding()
```

```
prce.endEncoding()
cb.commit()
```



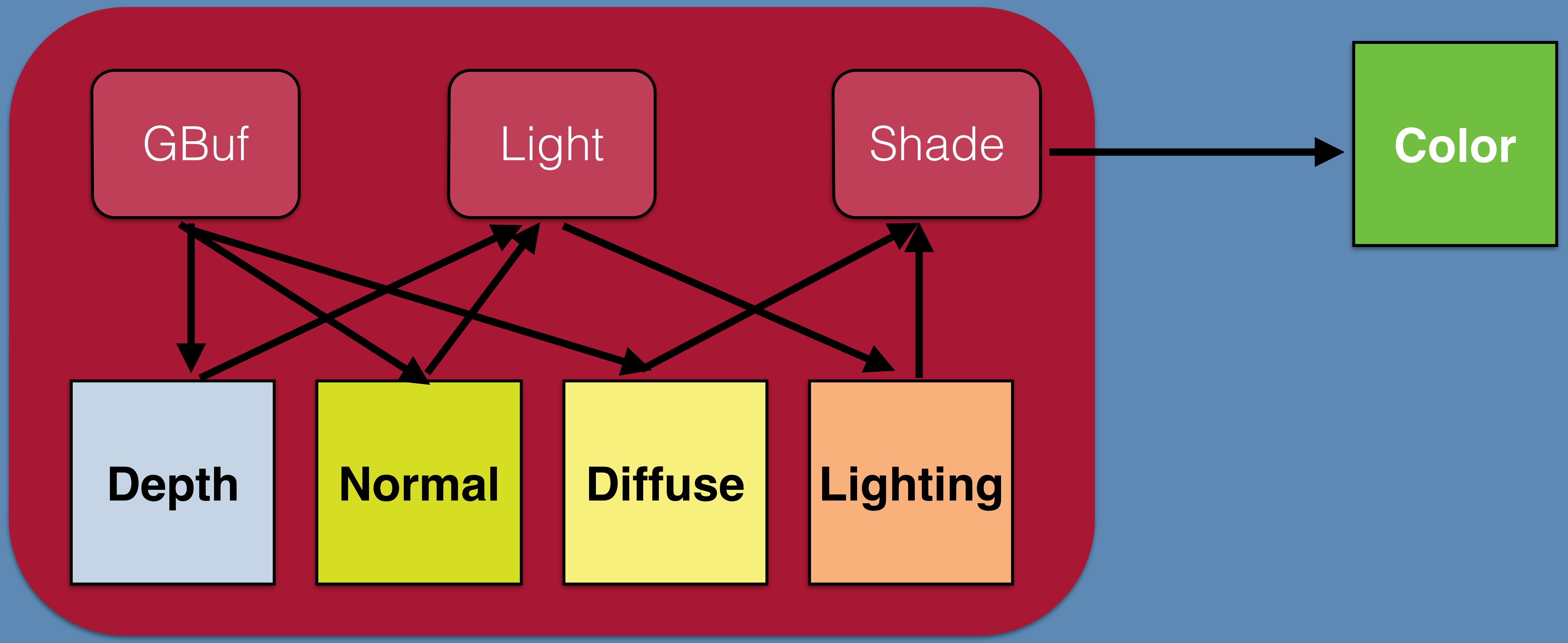
Render Pass Merging

- Vulkan allows merging some render passes
  - Render-to-texture-then-use, 1:1 pixel correspondence



Render Pass Merging

- Original passes become "subpasses"
- Intermediate data stays in on chip tile memory



Merging Render Passes

- Describe subpass dependencies to the driver
  - Driver maps to 1+ tiling passes and assigns per-pixel storage
  - Natural fallback to render-to-texture on immediate renderers
- Use description when creating Pipelines
- May be implemented as Pixel Local Storage
  - Merged passes don't guarantee fragment ordering in a pass
  - Drivers can expose a PLS extension for more explicit control

Summary

- Tile-based GPUs are first-class citizens now
- Choose load/store operations carefully
- Think about whether you need coarse (inter-pass) or fine (intra-pass) parallelism
- Look for opportunities to merge render passes