GDC
March 20-24, 2023
San Francisco, CA

# Cross-Platform Mobile and PC Rendering in ‘Earth Revival’

Chao Wang

Lead Engine Programmer / Nuverse Games


#GDC23

# Agenda

## Approaches to High-Quality Cross-Platform Rendering

1. Hybrid Occlusion Culling Method
2. Fully Parallel Scriptable Renderer
3. Scalable Cross-Platform Render Pipelines

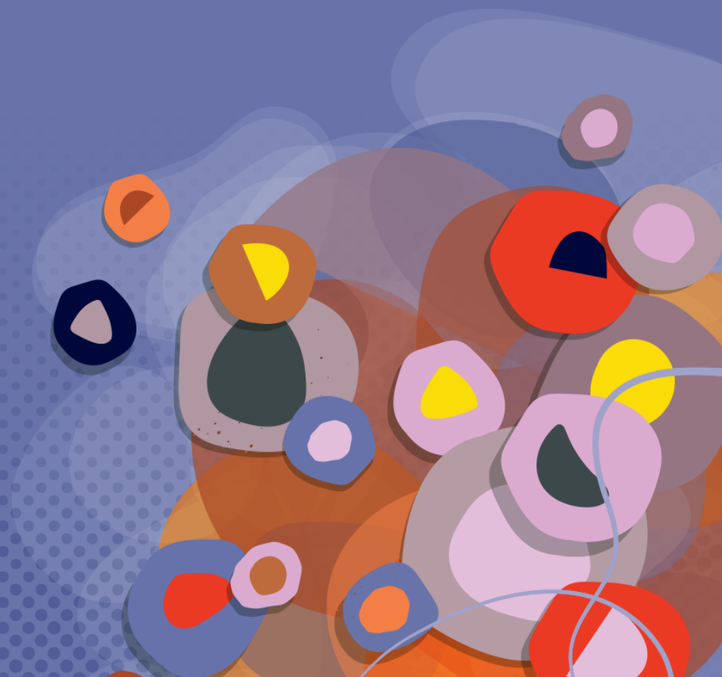
## A note about Unity-specific stuff

GDC March 20-24, 2023 San Francisco, CA

# Chapter 1. Hybrid Occlusion Culling Method

Umbra
Potential Visibility Sets
Hierarchical Z-Buffer
Software Occlusion Culling
…

How to Choose, or Beyond Choice?



# Occlusion Culling

Minimize objects and lights to render

- Require accurate culling.
- Low overhead for mobile.
- Effectiveness varies by scenario.
- Extensive experiments are necessary.

# Occlusion Culling: Extensive Testing

# Occlusion Culling: Extensive Testing








Comparative Testing on Umbra/PVS/Software/Query/Hi-Z

# Occlusion Culling: Comparative Testing

| Method | Tris Culled |
|---|---|
| Umbra | 70.8% |
| PVS | 70.1% |
| Software | 76.2% |
| Query | 82.7% |
| Hi-Z | 76.2% |

Pre-computed
(effective cell size: 5m)

Real-time

# Occlusion Culling: Art Workload

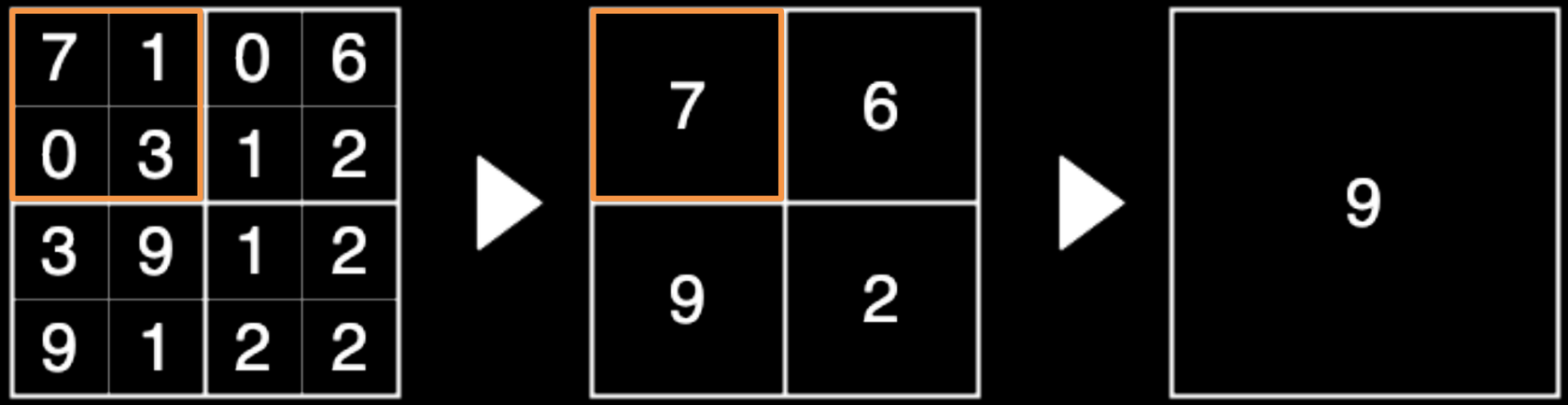
| Method | Art Workload |
| --- | --- |
| **Umbra** | 20 minutes of baking |
| **PVS** | 4 hours of baking |
| **Query** | Zero |
| **Hi-Z** | Zero |
| **Software** | Occluder Proxies |

# Occlusion Culling: Final Selection

| Method | Cross-platform Feasibility |
| --- | --- |
| **Query** | NO |
| **Hi-Z** | YES |

# Occlusion Culling: Our Hi-Z Method

Generate Hi-Z by down-sampling the depth buffer.



# Occlusion Culling: Our Hi-Z Method

Generate Hi-Z

- Compute shader vs. pixel shader
- PC&iOS: Compute Shader
- Android: Pixel Shader for GLES & Compute shader for Vulkan

| Method | GPU Time |
|---|---|
| Compute | 0.37 ms |
| Pixel Shader | 0.66 ms |

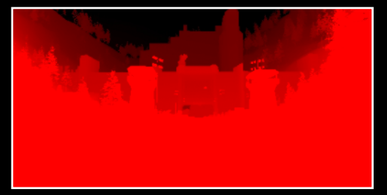
Results on iPhone X

Such a setup gives the lowest measured cost.

# Occlusion Culling: Our Hi-Z Method

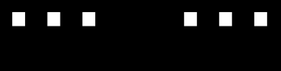
1559*720

1024*512



512*256

256*128



4*2

2*1

# Occlusion Culling: Our Hi-Z Method

## Visibility test with AABBs of objects

- Compute shader & indirect draw calls.
- CPU culling for Android with async readback
  - Multi-threading and SIMD.
  - 1~2ms of total job time; <1ms of time span.
  - Hide culling time with the execution of SRP.

# Occlusion Culling: Hi-Z's Drawbacks

- Hi-Z buffer is used in the next frame.
- Objects pop in if the camera is moving fast.

# Occlusion Culling: Intel’s Masked SOC

Masked Software Occlusion Culling (SOC)

- Proposed by Intel’s paper
- Optimized with SIMD and multi-threading
- Source code available

# Occlusion Culling: Intel's Masked SOC

## To minimize cost, filtering is applied to occluders:

- Only rasterize those with enough screen coverage.
- Sort by distance and rasterize up to a certain number.
- Configure a loading distance beyond which not to rasterize.

## Results compared with Hi-Z:

- No latency, but costs 0.9ms of CPU time on iPhone X.
- Similar culling rate, better when camera moves fast.
- Power consumption decreases by 5% without Hi-Z generation.

# Software Occlusion Culling: Drawbacks

- Depend on the quality of occluder proxies.
  - ✓Low polygon count
  - ✓Accurate occlusion
  - ✓Strict conservativeness
- A large world to populate.

# Occlusion Culling: A Hybrid Method

## Hi-Z and SOC to complement each other?

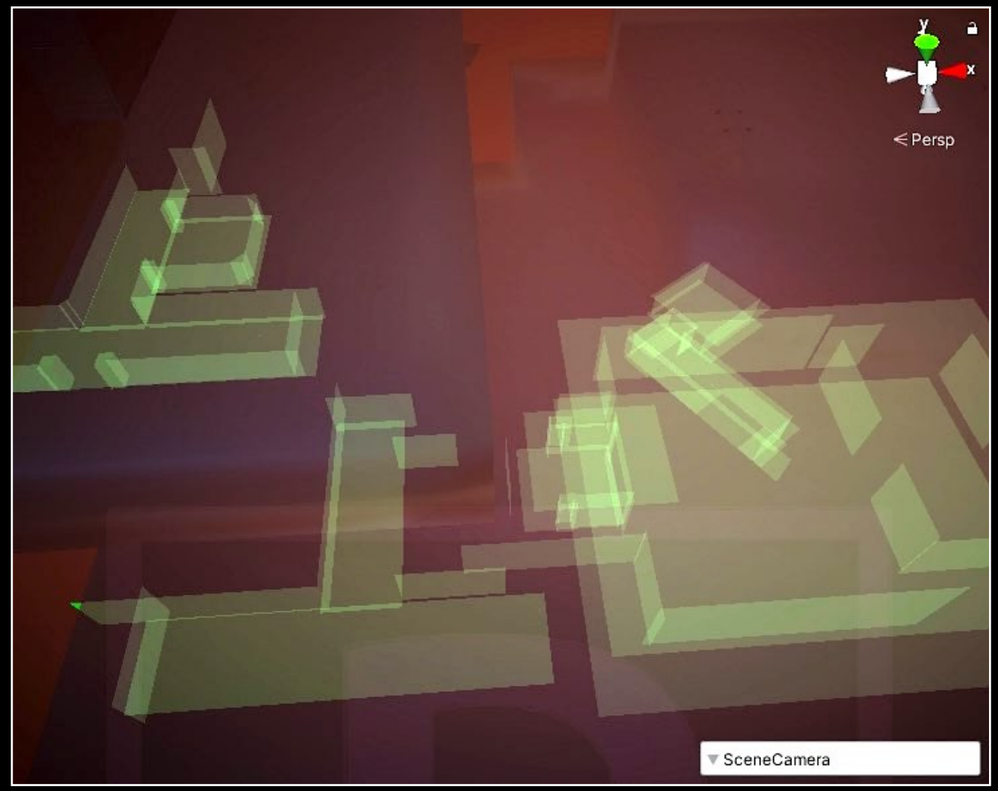
- Hi-Z is suitable apart from the latency artifacts.
- Artifacts are mostly observed in indoor scenes with:
  - ✓Camera at the edge of a wall,
  - ✓Large object occluded by this wall,
  - ✓Contrast between the background is high.

## Define Critical Walls

# Occlusion Culling: A Hybrid Method

## Critical walls

- Place proxies at critical walls.
- Get visible proxies screen coverage.
- Adaptively switch between methods
  - SOC if coverage exceeds threshold.
  - Hi-Z in most of the large world.



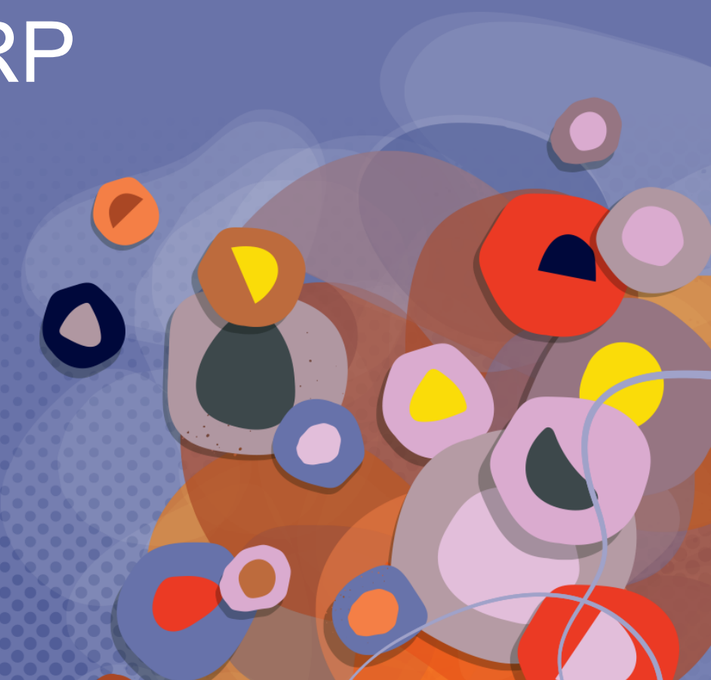
GDC March 20-24, 2023 San Francisco, CA

# Chapter 2. Fully Parallel Scriptable Renderer

Unity’s Philosophy: Control Everything with C# Scripts.

New multi-threaded rendering designed for SRP

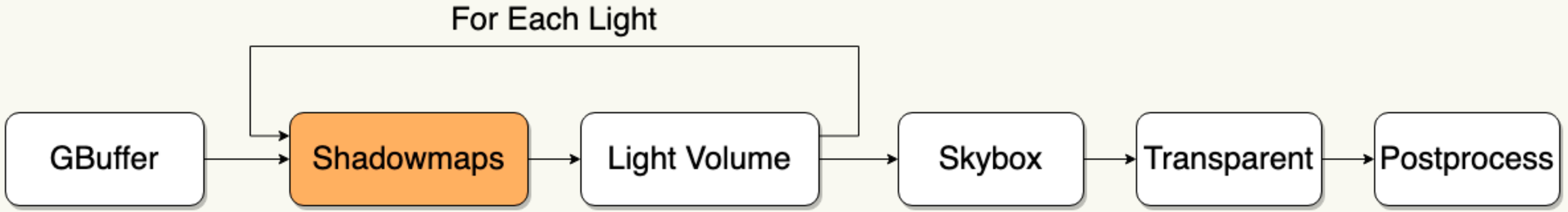
- Quick overview of SRP
- New Graphics Jobs explained
- Minimize the per-draw call overhead



# Built-in Render Pipeline

Let's start with Unity's built-in render pipeline.

A simplified version of the Built-in deferred render pipeline




Q: How to move shadowmap drawings to the front of GBuffer?

A: Modify the engine’s C++ source code.

# Scriptable Render Pipeline: A Glance

```
// Pseudo code of a deferred renderer.
// APIs simplified. SubPasses omitted.
void DeferredRender(ScriptableRenderContext context)
{
    foreach (var light in m_Lights) {
        // Begin a render pass and bind shadowmap RT.
        context.BeginRenderPass(light.shadowmapRT));
        // Render shadow-casting objects.
        context.DrawShadows(light);
        context.EndRenderPass();
    }

    // Begin a render pass and bind GBuffer MRTs.
    context.BeginRenderPass(m_GBufferMRTs));
    // Render opaque objects with the specified pass.
    context.DrawRenderers(shaderPass="GBuffer",
        renderQueues=Opaque+AlphaTest);
    foreach (var light in m_Lights) {
        // Setup any shader constants.
        context.SetConstantBuffer(light.CB);
        // Draw the light's volume geometry.
        context.DrawMesh(light.Geometry,
            light.WorldMatrix, m_LightingShader);
    }

    // Render Skybox.
    context.DrawSkybox();
    // Render forward transparent objects.
    context.DrawRenderers(shaderPass="Forward",
        renderQueues=Transparent);

    context.EndRenderPass();

    // Do post-processing.
    context.SetRenderTarget(m_FinalRT);
    context.DrawMesh(FullScreenQuad, m_PPShader);

    // Execute all render commands.
    context.Submit();
}
```

# Scriptable Render Pipeline: A Glance

```
// Pseudo code of a deferred renderer.
// APIs simplified. SubPasses omitted.
void DeferredRender(ScriptableRenderContext context)
{
    foreach (var light in m_Lights) {
        // Begin a render pass and bind shadowmap RT.
        context.BeginRenderPass(light.shadowmapRT));
        // Render shadow-casting objects.
        context.DrawShadows(light);
        context.EndRenderPass();
    }

    // Begin a render pass and bind GBuffer MRTs.
    context.BeginRenderPass(m_GBufferMRTs));
    // Render opaque objects with the specified pass.
    context.DrawRenderers(shaderPass="GBuffer",
        renderQueues=Opaque+AlphaTest);

    foreach (var light in m_Lights) {
        // Setup any shader constants.
        context.SetConstantBuffer(light.CB);
        // Draw the light's volume geometry.
        context.DrawMesh(light.Geometry,
            light.WorldMatrix, m_LightingShader);
    }

    // Render Skybox.
    context.DrawSkybox();
    // Render forward transparent objects.
    context.DrawRenderers(shaderPass="Forward",
        renderQueues=Transparent);

    context.EndRenderPass();

    // Do post-processing.
    context.SetRenderTarget(m_FinalRT);
    context.DrawMesh(FullScreenQuad, m_PPShader);

    // Execute all render commands.
    context.Submit();
}
```

# Scriptable Render Pipeline: A Glance

```
// Pseudo code of a deferred renderer.
// APIs simplified. SubPasses omitted.
void DeferredRender(ScriptableRenderContext context)
{
    foreach (var light in m_Lights) {
        // Begin a render pass and bind shadowmap RT.
        context.BeginRenderPass(light.shadowmapRT));
        // Render shadow-casting objects.
        context.DrawShadows(light);
        context.EndRenderPass();
    }

    // Begin a render pass and bind GBuffer MRTs.
    context.BeginRenderPass(m_GBufferMRTs));
    // Render opaque objects with the specified pass.
    context.DrawRenderers(shaderPass="GBuffer",
        renderQueues=Opaque+AlphaTest);
    foreach (var light in m_Lights) {
        // Setup any shader constants.
        context.SetConstantBuffer(light.CB);
        // Draw the light's volume geometry.
        context.DrawMesh(light.Geometry,
            light.WorldMatrix, m_LightingShader);
    }

    // Render Skybox.
    context.DrawSkybox();
    // Render forward transparent objects.
    context.DrawRenderers(shaderPass="Forward",
        renderQueues=Transparent);

    context.EndRenderPass();

    // Do post-processing.
    context.SetRenderTarget(m_FinalRT);
    context.DrawMesh(FullScreenQuad, m_PPShader);

    // Execute all render commands.
    context.Submit();
}
```

# Scriptable Render Pipeline: A Glance

```
// Pseudo code of a deferred renderer.
// APIs simplified. SubPasses omitted.
void DeferredRender(ScriptableRenderContext context)
{
    foreach (var light in m_Lights) {
        // Begin a render pass and bind shadowmap RT.
        context.BeginRenderPass(light.shadowmapRT));
        // Render shadow-casting objects.
        context.DrawShadows(light);
        context.EndRenderPass();
    }

    // Begin a render pass and bind GBuffer MRTs.
    context.BeginRenderPass(m_GBufferMRTs));
    // Render opaque objects with the specified pass.
    context.DrawRenderers(shaderPass="GBuffer",
        renderQueues=Opaque+AlphaTest);
    foreach (var light in m_Lights) {
        // Setup any shader constants.
        context.SetConstantBuffer(light.CB);
        // Draw the light's volume geometry.
        context.DrawMesh(light.Geometry,
            light.WorldMatrix, m_LightingShader);
    }

    // Render Skybox.
    context.DrawSkybox();
    // Render forward transparent objects.
    context.DrawRenderers(shaderPass="Forward",
        renderQueues=Transparent);

    context.EndRenderPass();

    // Do post-processing.
    context.SetRenderTarget(m_FinalRT);
    context.DrawMesh(FullScreenQuad, m_PPShader);

    // Execute all render commands.
    context.Submit();
}
```

# Scriptable Render Pipeline: A Glance

```
// Pseudo code of a deferred renderer.
// APIs simplified. SubPasses omitted.
void DeferredRender(ScriptableRenderContext context)
{
    foreach (var light in m_Lights) {
        // Begin a render pass and bind shadowmap RT.
        context.BeginRenderPass(light.shadowmapRT));
        // Render shadow-casting objects.
        context.DrawShadows(light);
        context.EndRenderPass();
    }

    // Begin a render pass and bind GBuffer MRTs.
    context.BeginRenderPass(m_GBufferMRTs));
    // Render opaque objects with the specified pass.
    context.DrawRenderers(shaderPass="GBuffer",
        renderQueues=Opaque+AlphaTest);
    foreach (var light in m_Lights) {
        // Setup any shader constants.
        context.SetConstantBuffer(light.CB);
        // Draw the light's volume geometry.
        context.DrawMesh(light.Geometry,
            light.WorldMatrix, m_LightingShader);
    }

    // Render Skybox.
    context.DrawSkybox();
    // Render forward transparent objects.
    context.DrawRenderers(shaderPass="Forward",
        renderQueues=Transparent);

    context.EndRenderPass();

    // Do post-processing.
    context.SetRenderTarget(m_FinalRT);
    context.DrawMesh(FullScreenQuad, m_PPShader);

    // Execute all render commands.
    context.Submit();
}
```

# Scriptable Render Pipeline: A Glance

```
// Pseudo code of a deferred renderer.
// APIs simplified. SubPasses omitted.
void DeferredRender(ScriptableRenderContext context)
{
    foreach (var light in m_Lights) {
        // Begin a render pass and bind shadowmap RT.
        context.BeginRenderPass(light.shadowmapRT));
        // Render shadow-casting objects.
        context.DrawShadows(light);
        context.EndRenderPass();
    }

    // Begin a render pass and bind GBuffer MRTs.
    context.BeginRenderPass(m_GBufferMRTs));
    // Render opaque objects with the specified pass.
    context.DrawRenderers(shaderPass="GBuffer",
        renderQueues=Opaque+AlphaTest);
    foreach (var light in m_Lights) {
        // Setup any shader constants.
        context.SetConstantBuffer(light.CB);
        // Draw the light's volume geometry.
        context.DrawMesh(light.Geometry,
            light.WorldMatrix, m_LightingShader);
    }

    // Render Skybox.
    context.DrawSkybox();
    // Render forward transparent objects.
    context.DrawRenderers(shaderPass="Forward",
        renderQueues=Transparent);

    context.EndRenderPass();

    // Do post-processing.
    context.SetRenderTarget(m_FinalRT);
    context.DrawMesh(FullScreenQuad, m_PPShader);

    // Execute all render commands.
    context.Submit();
}
```

# Scriptable Render Pipeline: Context Data

```
context.DrawRenderers(shaderPass=“GBuffer”, renderQueues=Opaque+AlphaTest);
```

Add a pre-depth pass for all the grasses

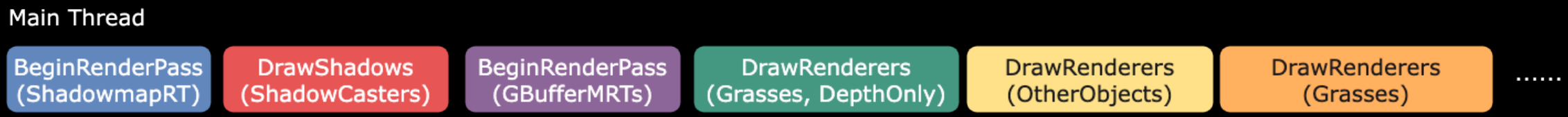
```
context.DrawRenderers(shaderPass="DepthOnly", filterMask=GrassLayerMask);
context.DrawRenderers(shaderPass="GBuffer", filterMask=~GrassLayerMask,
    renderQueue=…);
context.DisableKeyword(grassMaterial, ”_ALPHA_TEST_ON”);
context.DrawRenderers(shaderPass="GBuffer", filterMask=GrassLayerMask);
context.EnableKeyword(grassMaterial, ”_ALPHA_TEST_ON");
```

In summary, SRP provides a wide range of high-level render commands that allow us to customize the rendering pipeline with great flexibility.

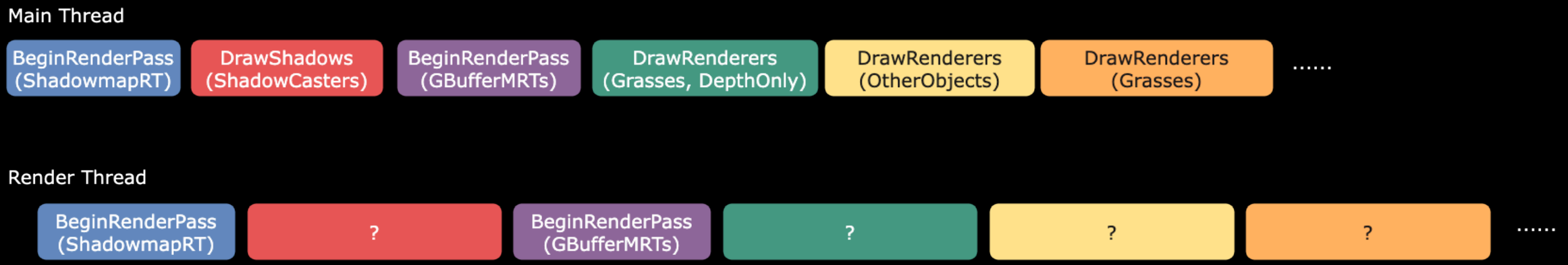
# Unity’s Graphics Jobs: A Glance

Starts from ScriptableRenderContext.Submit().

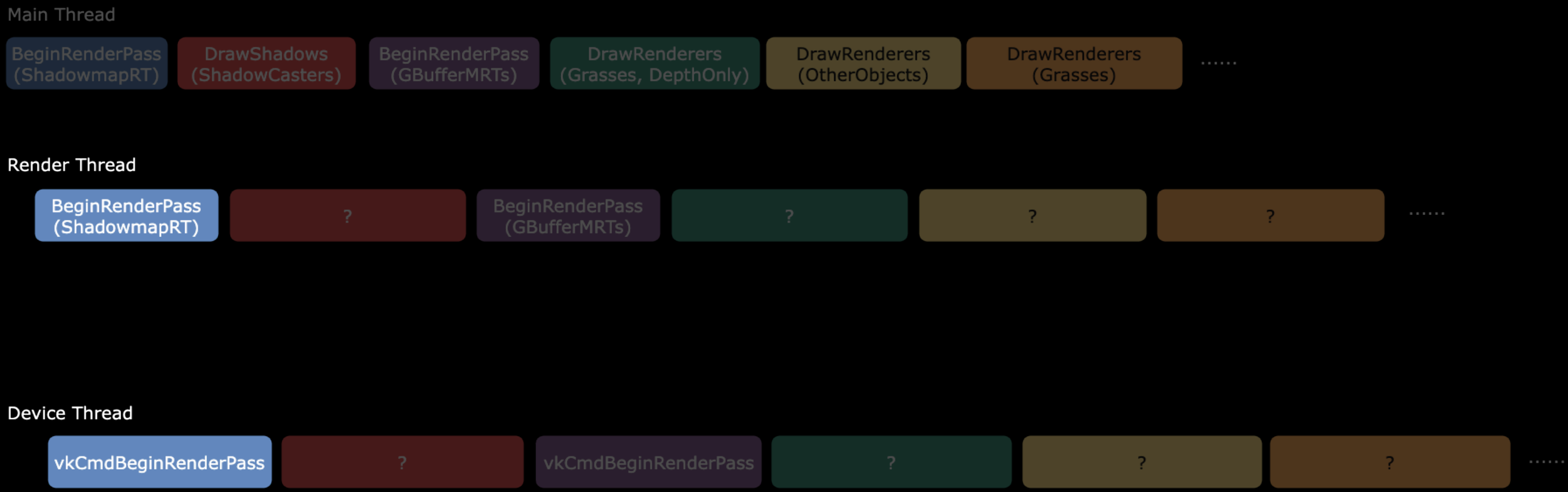
Subpass/EndRenderPass omitted for simplicity.



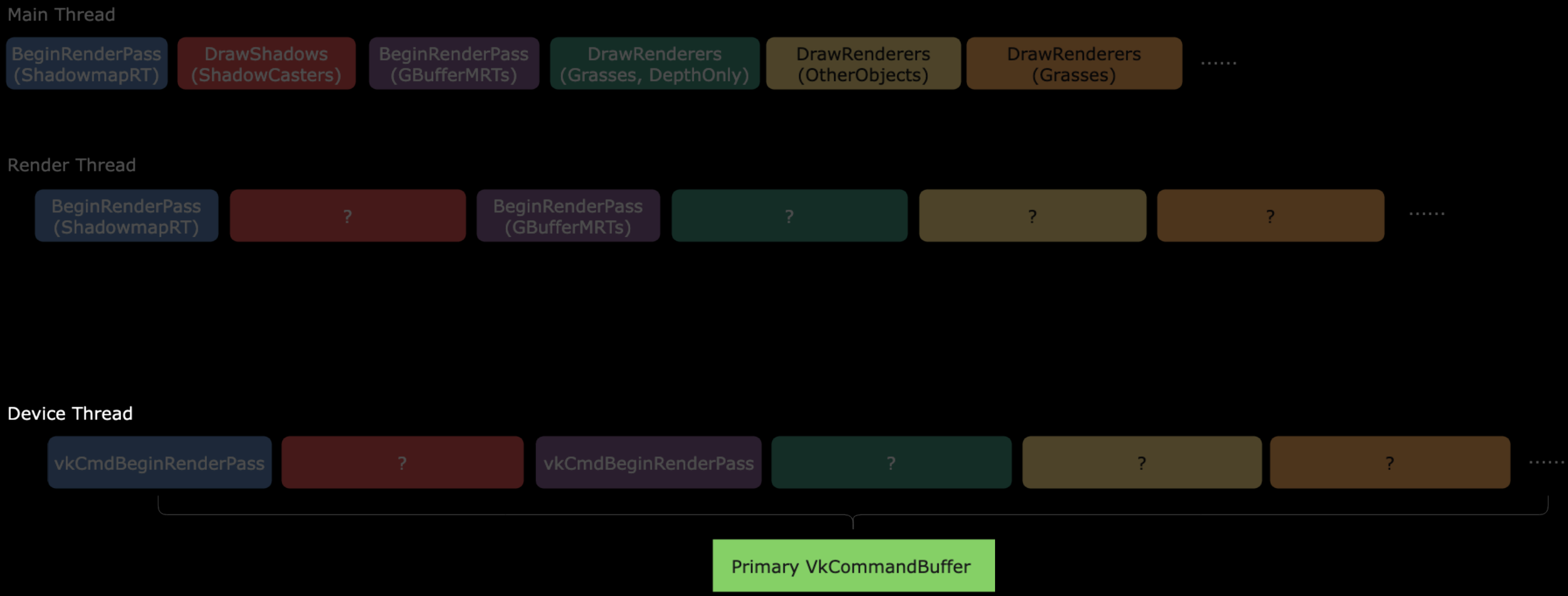
# Unity’s Graphics Jobs: A Glance



# Unity’s Graphics Jobs: A Glance



# Unity’s Graphics Jobs: A Glance

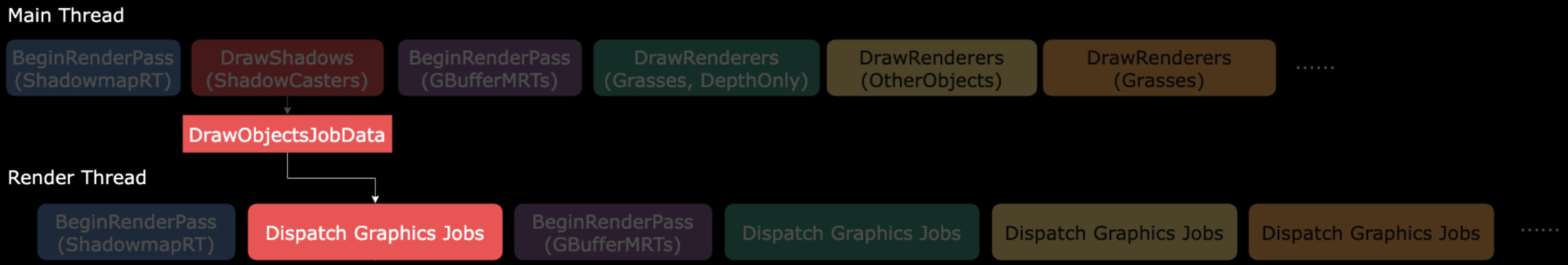


# Unity’s Graphics Jobs: A Glance

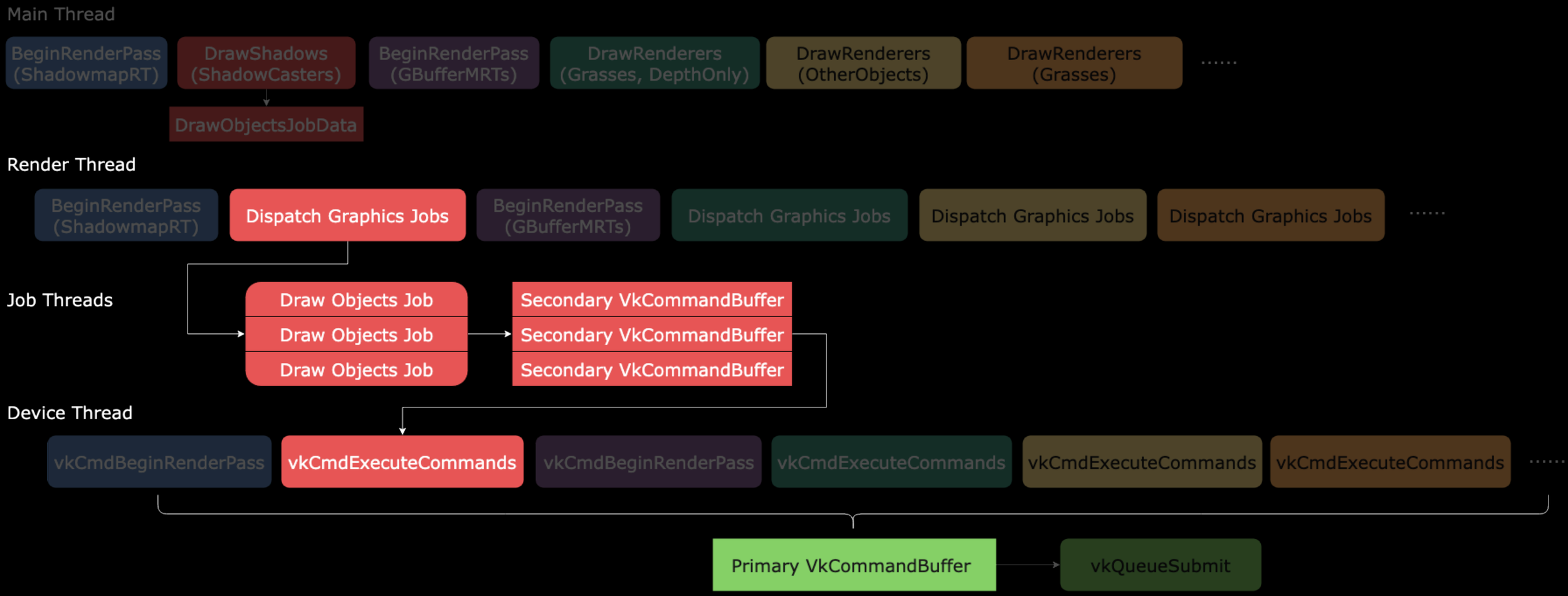

Split into <?> jobs:

- Number of objects
- Number of job threads
- Min objects per job

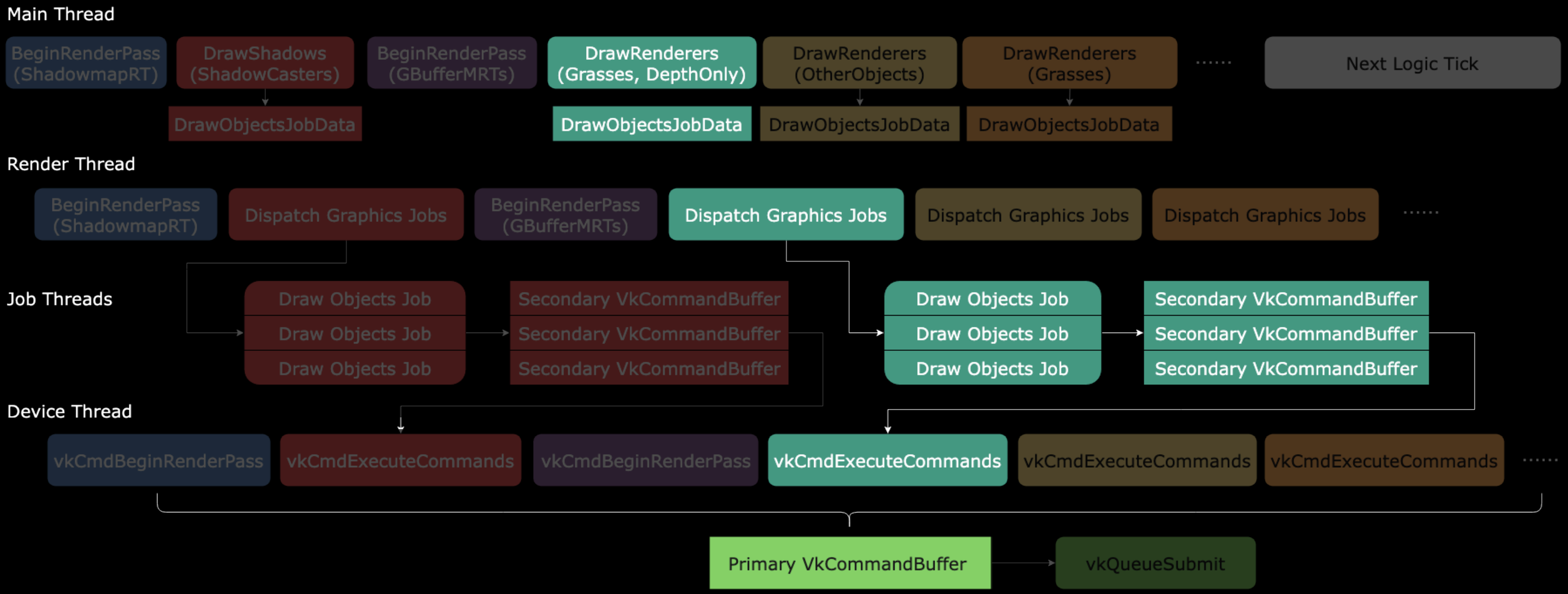
# Unity’s Graphics Jobs: A Glance



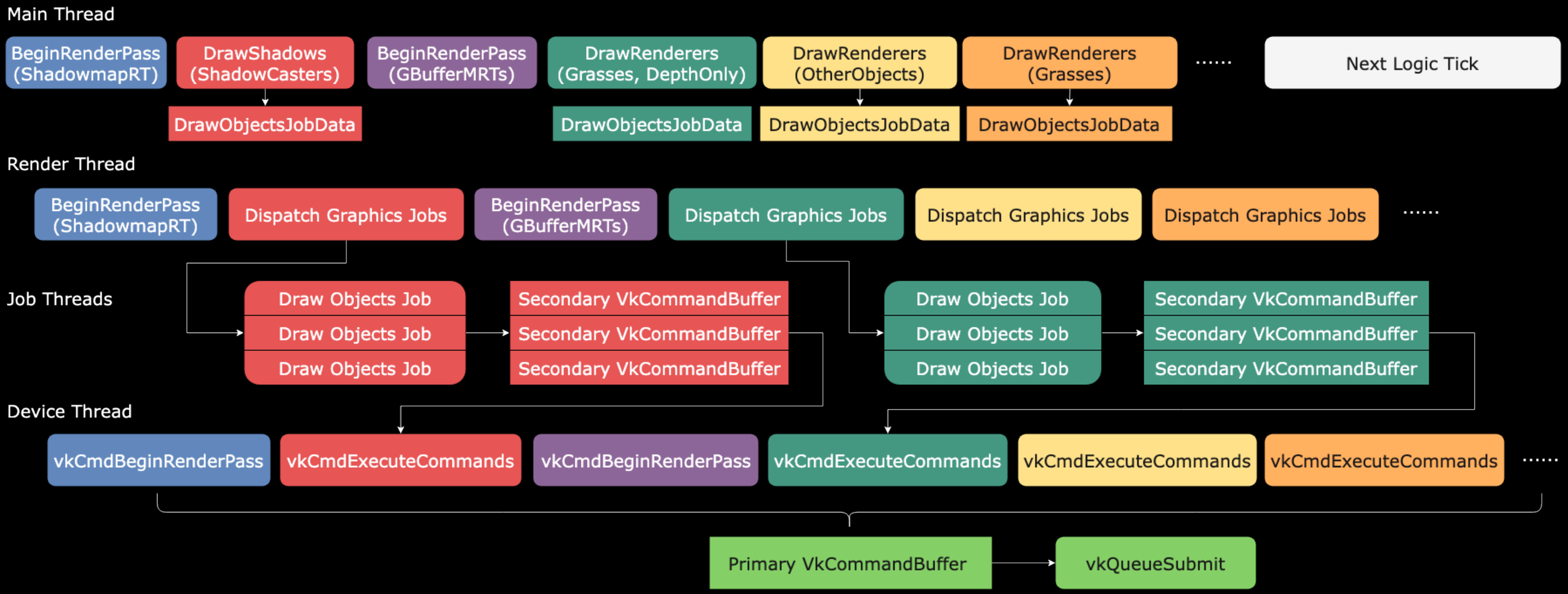
# Unity's Graphics Jobs: A Glance



# Unity's Graphics Jobs: A Glance



# Unity's Graphics Jobs: A Glance



# Unity’s Graphics Jobs: Analysis

Observations:

- Too many SCBs: Each DrawRenderers command produces multiple SCBs. SRP commands between adjacent DrawRenderers also produce an SCB.
- DrawRenderers commands with small numbers of draw calls: Processed sequentially, can’t always keep all the CPU cores busy.
- Both in a serialized manner: The main thread forwards commands to the render thread; the render thread receives and processes them.
- Parallelism depends on SCBs: Some Android devices have compatibility issues with SCBs, making it impossible to enable Graphics Jobs.

To address these issues, we designed a new Graphics Jobs architecture.

# Our New Graphics Jobs

✓SRP commands are executed by job threads.

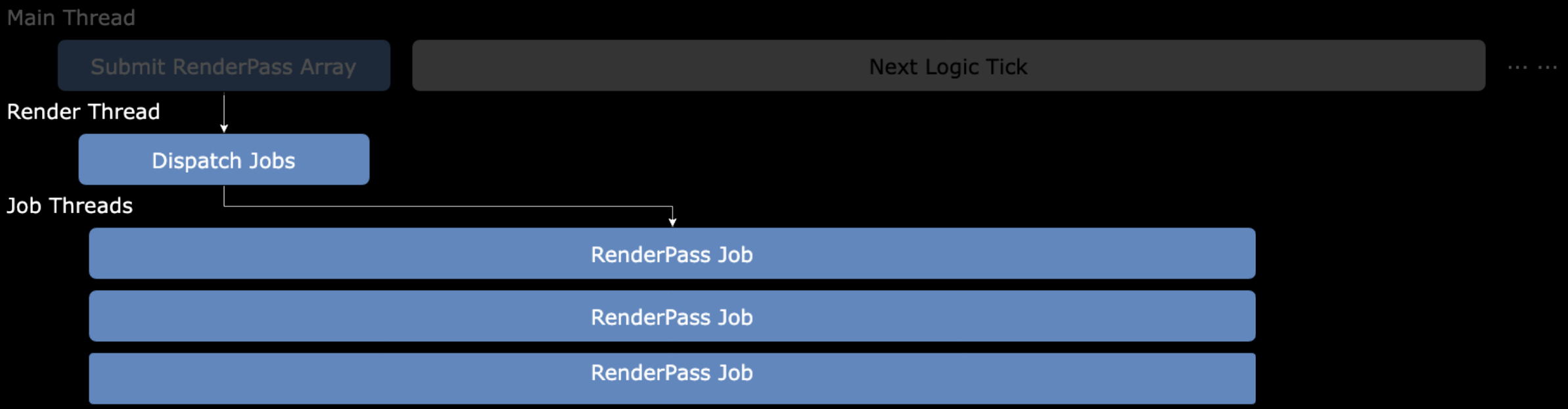
✓Render passes are made parallel to each other.

✓Merge all DrawRenderers and divide them into jobs.

# Our New Graphics Jobs

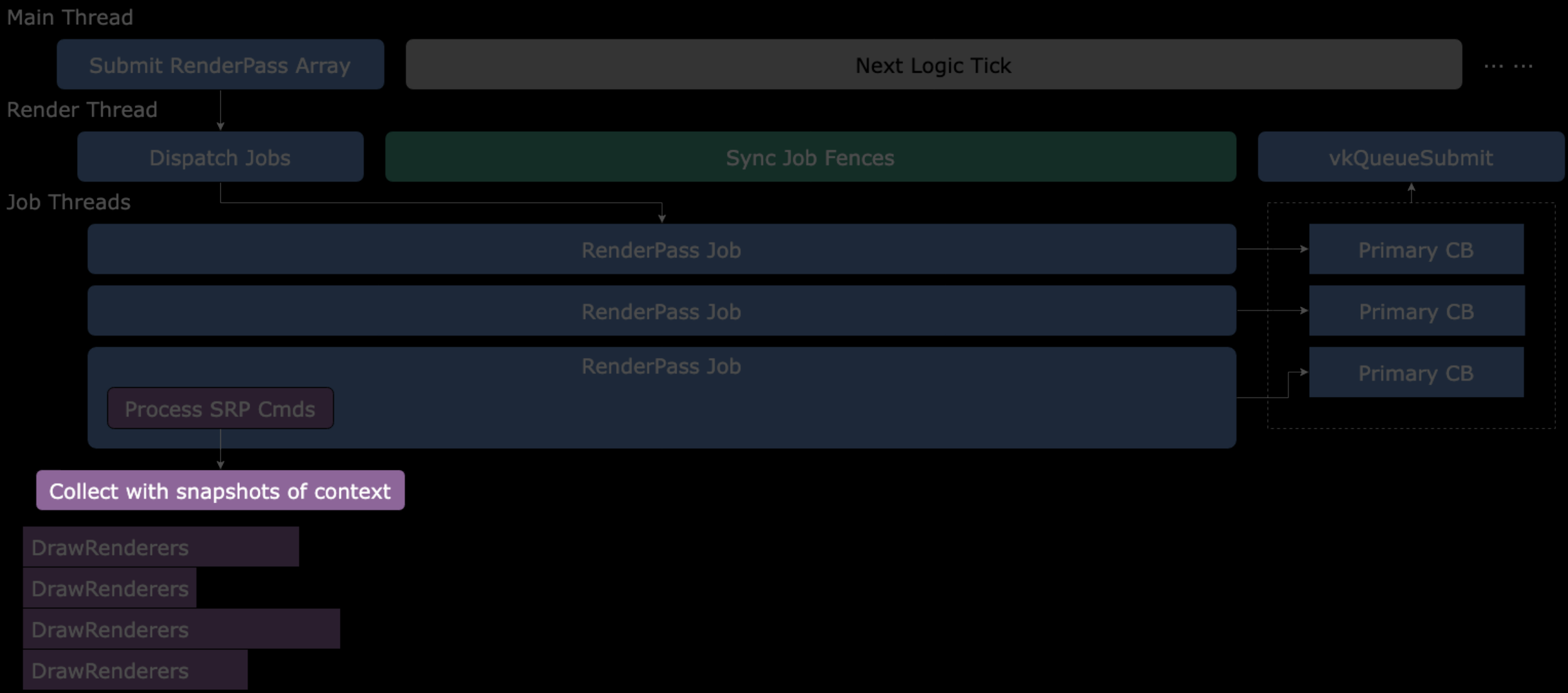

- Explicit definition of RenderPass objects
- Populate SRP commands into RenderPasses
- Submit an array of RenderPasses

# Our New Graphics Jobs

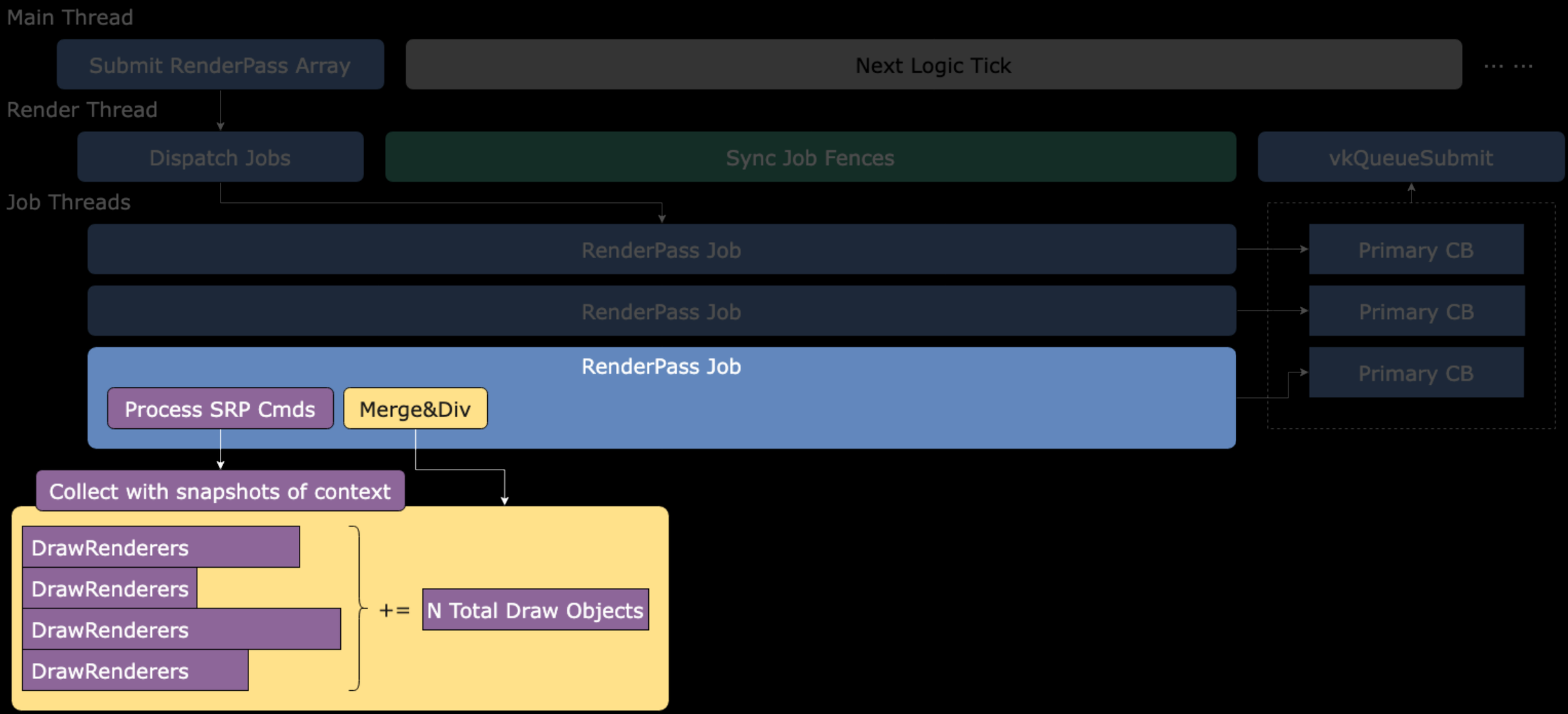


# Our New Graphics Jobs

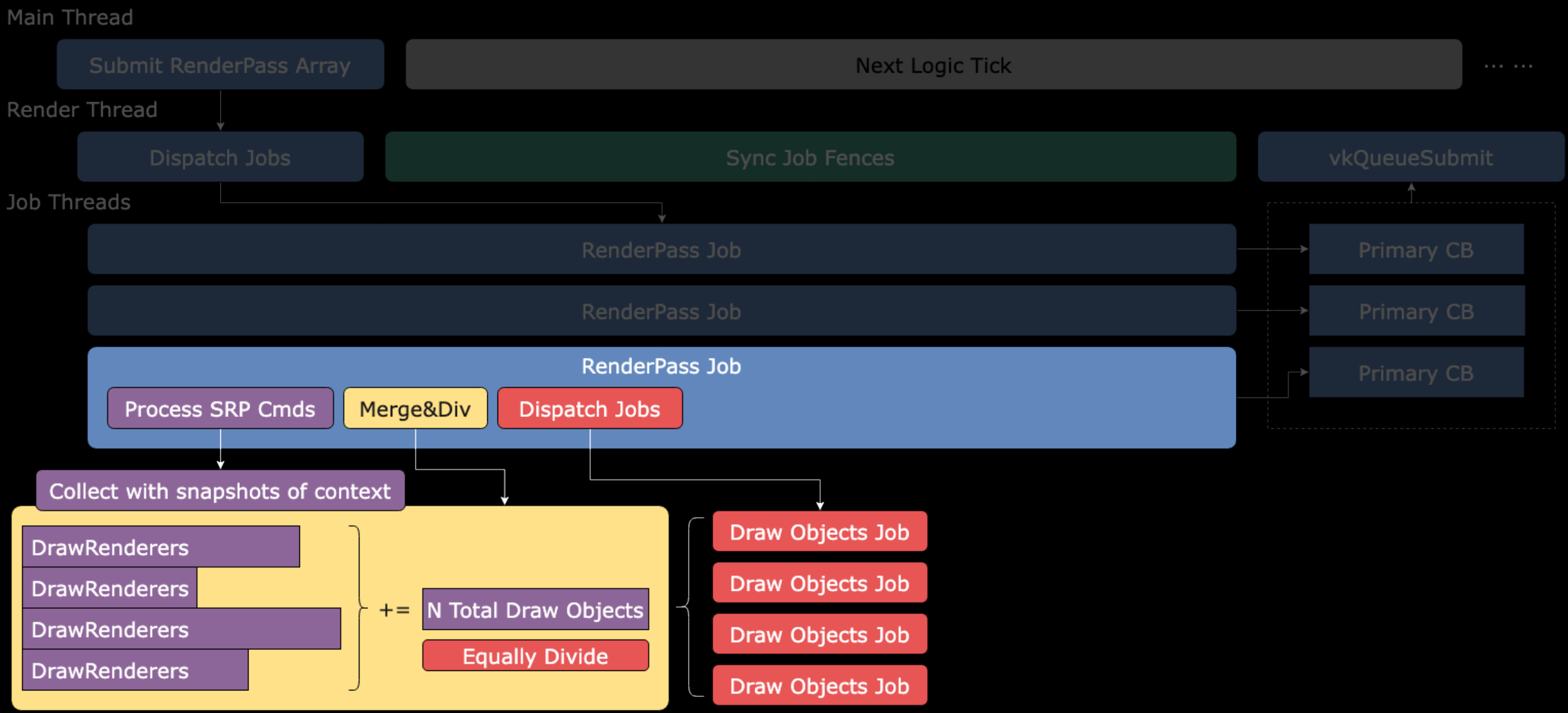
# Our New Graphics Jobs



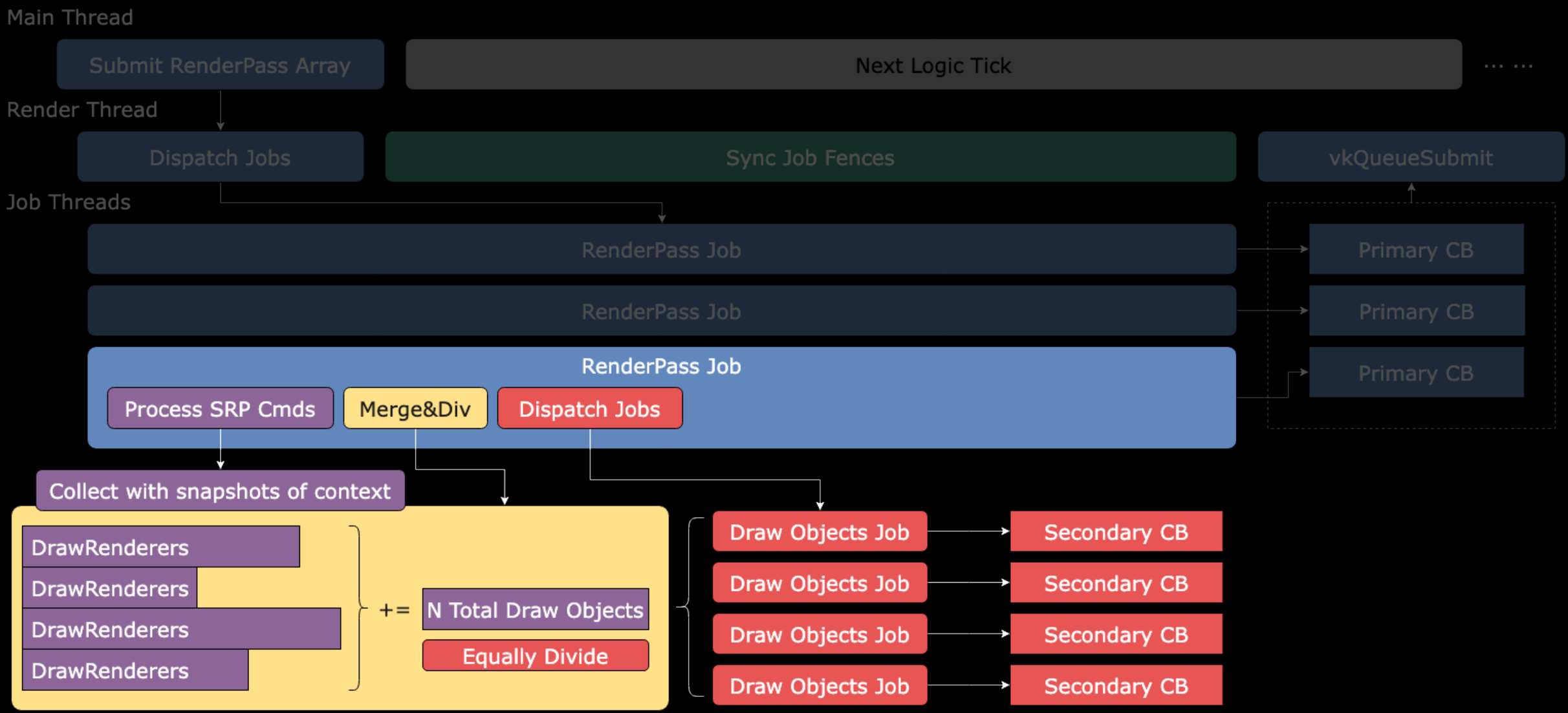
# Our New Graphics Jobs



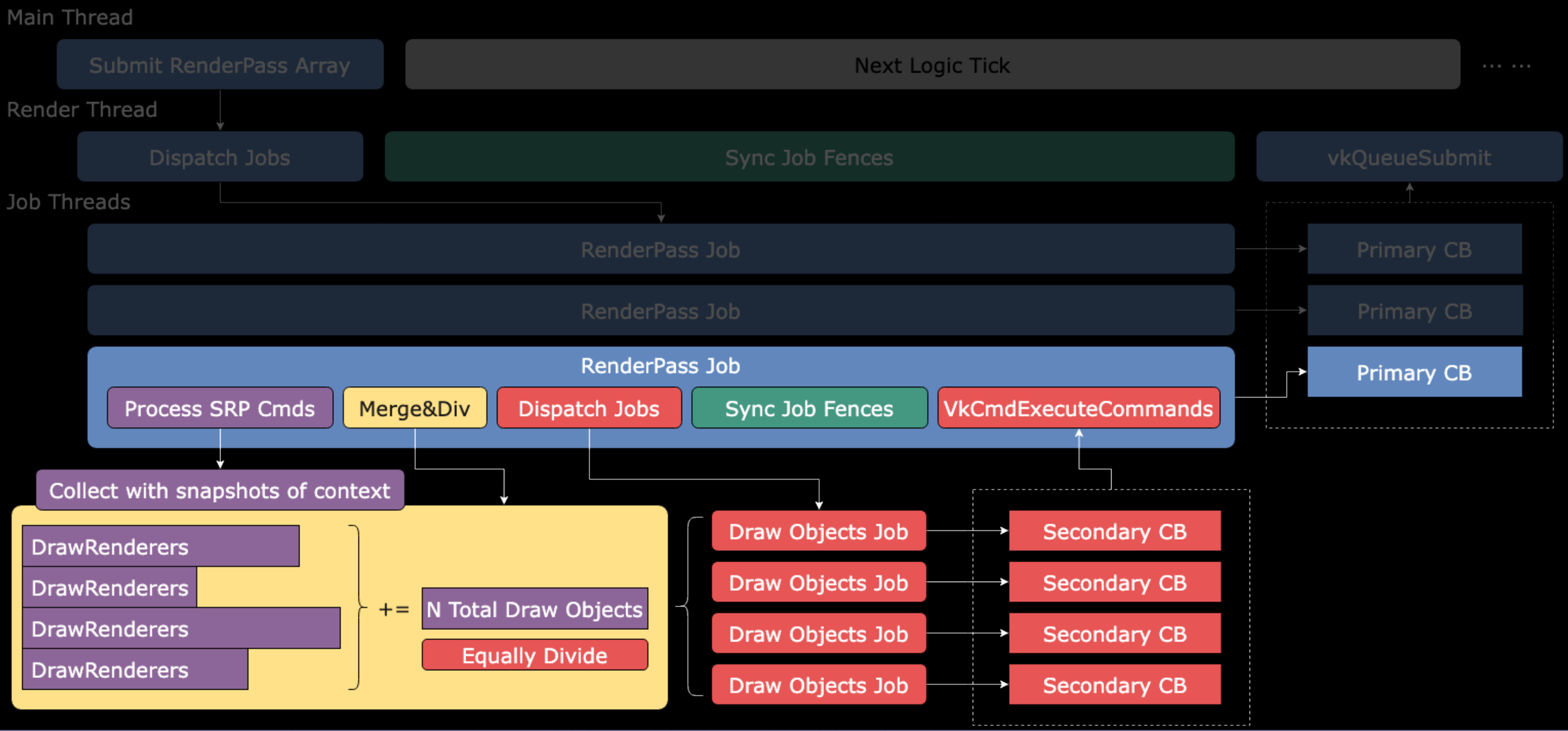
# Our New Graphics Jobs



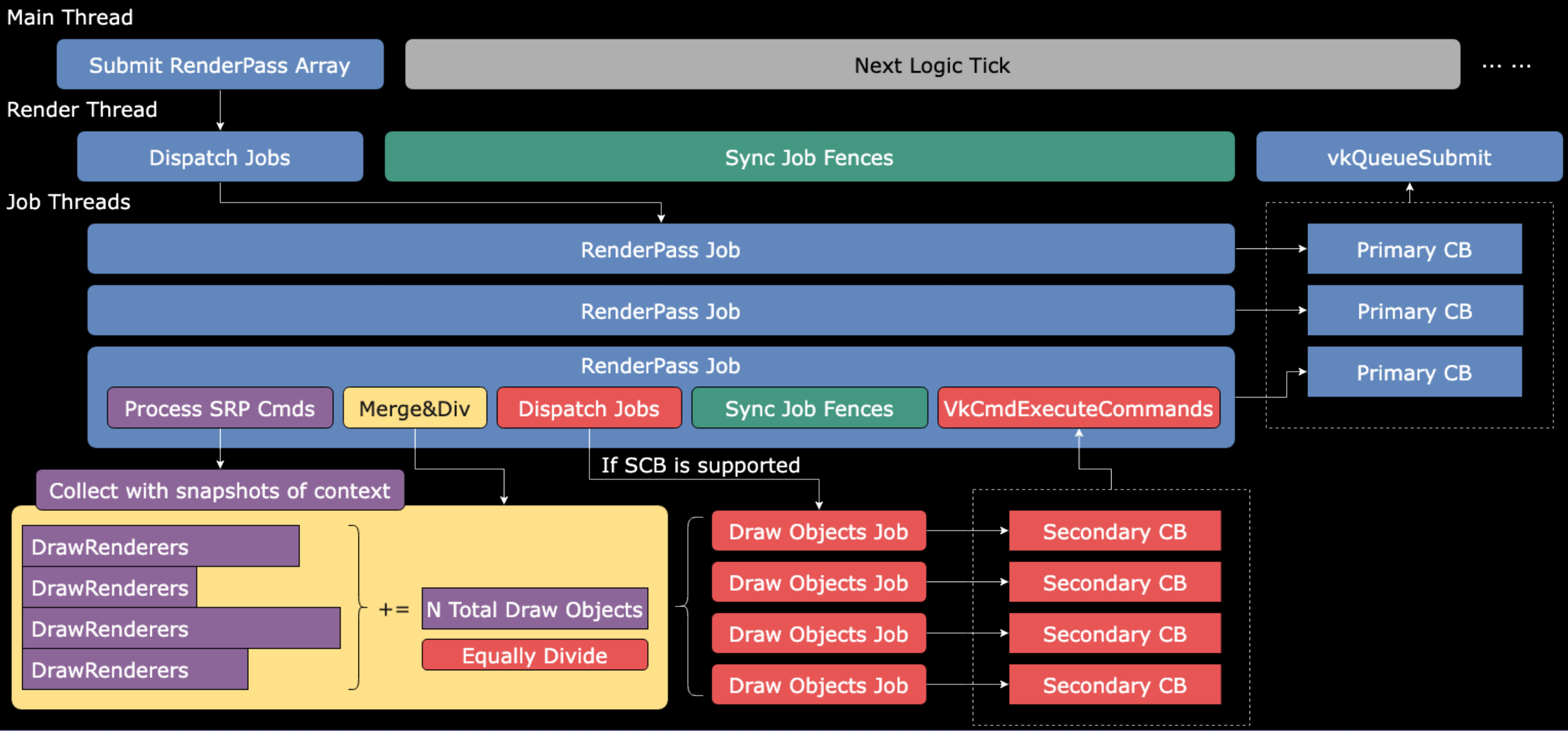
# Our New Graphics Jobs



# Our New Graphics Jobs



# Our New Graphics Jobs



# New Scriptable Render Pipeline API

```
// A ThreadedRenderPass will produce one primary CB.
var renderpass = context.CreateThreadedRenderPass(1920, 1080, samples, attachments, …);
// GBuffer subpass, will produce several secondary CBs.
var subpass0 = renderpass.CreateSubPass(colors=gbuffers, ...);
subpass0.DrawRenderers(shaderPass="DepthOnly", filterMask=GrassLayerMask);
subpass0.DrawRenderers(shaderPass="GBuffer", filterMask=~GrassLayerMask, renderQueues…);
...
// ThreadedRenderContext can encode multiple render passes into one primary CB.
// Use it to avoid large amounts of small jobs.
var postprocessContext = context.CreateThreadedRenderContext();
postprocessContext.SetRenderTarget(...);
postprocessContext.DrawMesh(m_FullScreenQuad, m_PPShader0);
postprocessContext.SetRenderTarget(...);
postprocessContext.DrawMesh(m_FullScreenQuad, m_PPShader1);
...
context.SubmitThreadedRenderPasses();  // Start graphics jobs.
```

# New Scriptable Render Pipeline API

```
// A ThreadedRenderPass will produce one primary CB.
var renderpass = context.CreateThreadedRenderPass(1920, 1080, samples, attachments, …);
// GBuffer subpass, will produce several secondary CBs.
var subpass0 = renderpass.CreateSubPass(colors=gbuffers, ...);
subpass0.DrawRenderers(shaderPass="DepthOnly", filterMask=GrassLayerMask);
subpass0.DrawRenderers(shaderPass="GBuffer", filterMask=~GrassLayerMask, renderQueues…);
...
// ThreadedRenderContext can encode multiple render passes into one primary CB.
// Use it to avoid large amounts of small jobs.
var postprocessContext = context.CreateThreadedRenderContext();
postprocessContext.SetRenderTarget(...);
postprocessContext.DrawMesh(m_FullScreenQuad, m_PPShader0);
postprocessContext.SetRenderTarget(...);
postprocessContext.DrawMesh(m_FullScreenQuad, m_PPShader1);
...
context.SubmitThreadedRenderPasses();  // Start graphics jobs.
```

# New Scriptable Render Pipeline API

```
// A ThreadedRenderPass will produce one primary CB.
var renderpass = context.CreateThreadedRenderPass(1920, 1080, samples, attachments, …);
// GBuffer subpass, will produce several secondary CBs.
var subpass0 = renderpass.CreateSubPass(colors=gbuffers, ...);
subpass0.DrawRenderers(shaderPass="DepthOnly", filterMask=GrassLayerMask);
subpass0.DrawRenderers(shaderPass="GBuffer", filterMask=~GrassLayerMask, renderQueues…);
...
// ThreadedRenderContext can encode multiple render passes into one primary CB.
// Use it to avoid large amounts of small jobs.
var postprocessContext = context.CreateThreadedRenderContext();
postprocessContext.SetRenderTarget(...);
postprocessContext.DrawMesh(m_FullScreenQuad, m_PPShader0);
postprocessContext.SetRenderTarget(...);
postprocessContext.DrawMesh(m_FullScreenQuad, m_PPShader1);
...
context.SubmitThreadedRenderPasses();  // Start graphics jobs.
```

# New Scriptable Render Pipeline API

```
// A ThreadedRenderPass will produce one primary CB.
var renderpass = context.CreateThreadedRenderPass(1920, 1080, samples, attachments, …);
// GBuffer subpass, will produce several secondary CBs.
var subpass0 = renderpass.CreateSubPass(colors=gbuffers, ...);
subpass0.DrawRenderers(shaderPass="DepthOnly", filterMask=GrassLayerMask);
subpass0.DrawRenderers(shaderPass="GBuffer", filterMask=~GrassLayerMask, renderQueues…);
...
// ThreadedRenderContext can encode multiple render passes into one primary CB.
// Use it to avoid large amounts of small jobs.
var postprocessContext = context.CreateThreadedRenderContext();
postprocessContext.SetRenderTarget(...);
postprocessContext.DrawMesh(m_FullScreenQuad, m_PPShader0);
postprocessContext.SetRenderTarget(...);
postprocessContext.DrawMesh(m_FullScreenQuad, m_PPShader1);
...
context.SubmitThreadedRenderPasses();  // Start graphics jobs.
```

# New Scriptable Render Pipeline API

```
// A ThreadedRenderPass will produce one primary CB.
var renderpass = context.CreateThreadedRenderPass(1920, 1080, samples, attachments, …);
// GBuffer subpass, will produce several secondary CBs.
var subpass0 = renderpass.CreateSubPass(colors=gbuffers, ...);
subpass0.DrawRenderers(shaderPass="DepthOnly", filterMask=GrassLayerMask);
subpass0.DrawRenderers(shaderPass="GBuffer", filterMask=~GrassLayerMask, renderQueues…);
...
// ThreadedRenderContext can encode multiple render passes into one primary CB.
// Use it to avoid large amounts of small jobs.
var postprocessContext = context.CreateThreadedRenderContext();
postprocessContext.SetRenderTarget(...);
postprocessContext.DrawMesh(m_FullScreenQuad, m_PPShader0);
postprocessContext.SetRenderTarget(...);
postprocessContext.DrawMesh(m_FullScreenQuad, m_PPShader1);
...
context.SubmitThreadedRenderPasses();  // Start graphics jobs.
```

# New Scriptable Render Pipeline: Future Works

- Issuing SRP commands is still done sequentially on Main Thread.

```
// Main thread creates all threaded contexts and dispatches commands-issuing jobs.
var renderpass = context.CreateThreadedRenderPass(1920, 1080, samples, attachments, ...);
var subpass0 = renderpass.CreateSubPass(colors=gbuffers, ...);
var subpass1 = renderpass.CreateSubPass(colors=gbuffers[0:1], inputs=gbuffers[1:], ...);
var deferredFence = DispatchDeferredRenderingJobs(subpass0, subpass1);

var postprocessContext = context.CreateThreadedRenderContext();
var postprocessFence = DispatchPostprocessingJob(postprocessContext);

// Wait for SRP commands ready and start new graphics jobs.
SyncJobFences([deferredFence, postprocessFence]);
context.SubmitThreadedContexts();
```

# Delving into Draw Objects Job: A Step Further

A draw objects job executes a large number of draw calls

- Bind shaders & set device states (once per batch)
- Fill and bind constant buffers & bind textures
- Bind VB&IB and call DrawIndexed(Instanced)

Unity's Philosophy: Control Everything with C# Scripts.

Optimization potential in filling constant buffers

- SRP allows scripting the rendering process, including shader constant values.
- This comes with a cost, as the constant buffers need to be filled for each draw.

# Constant Buffers: the Heavy Work

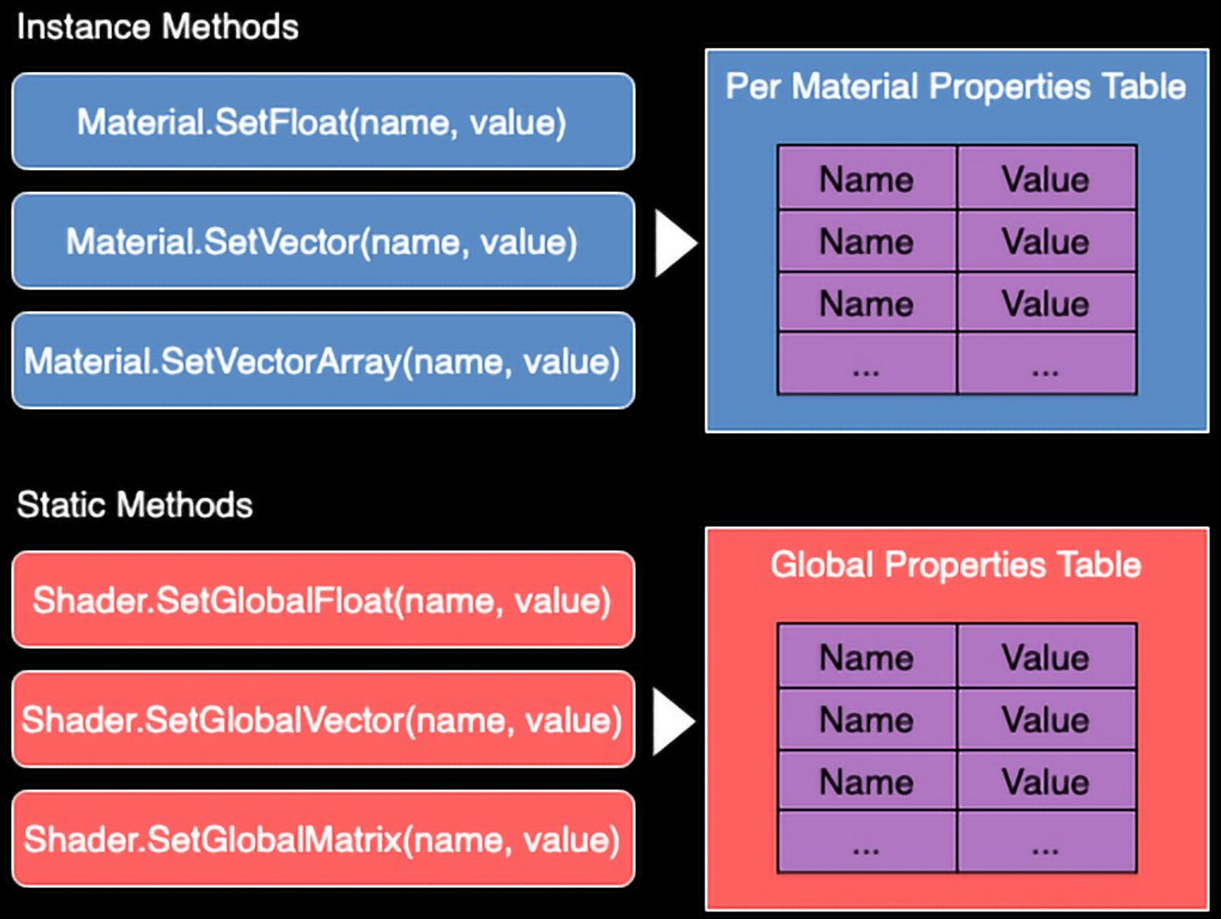
Instance Methods

Material.SetFloat(name, value)

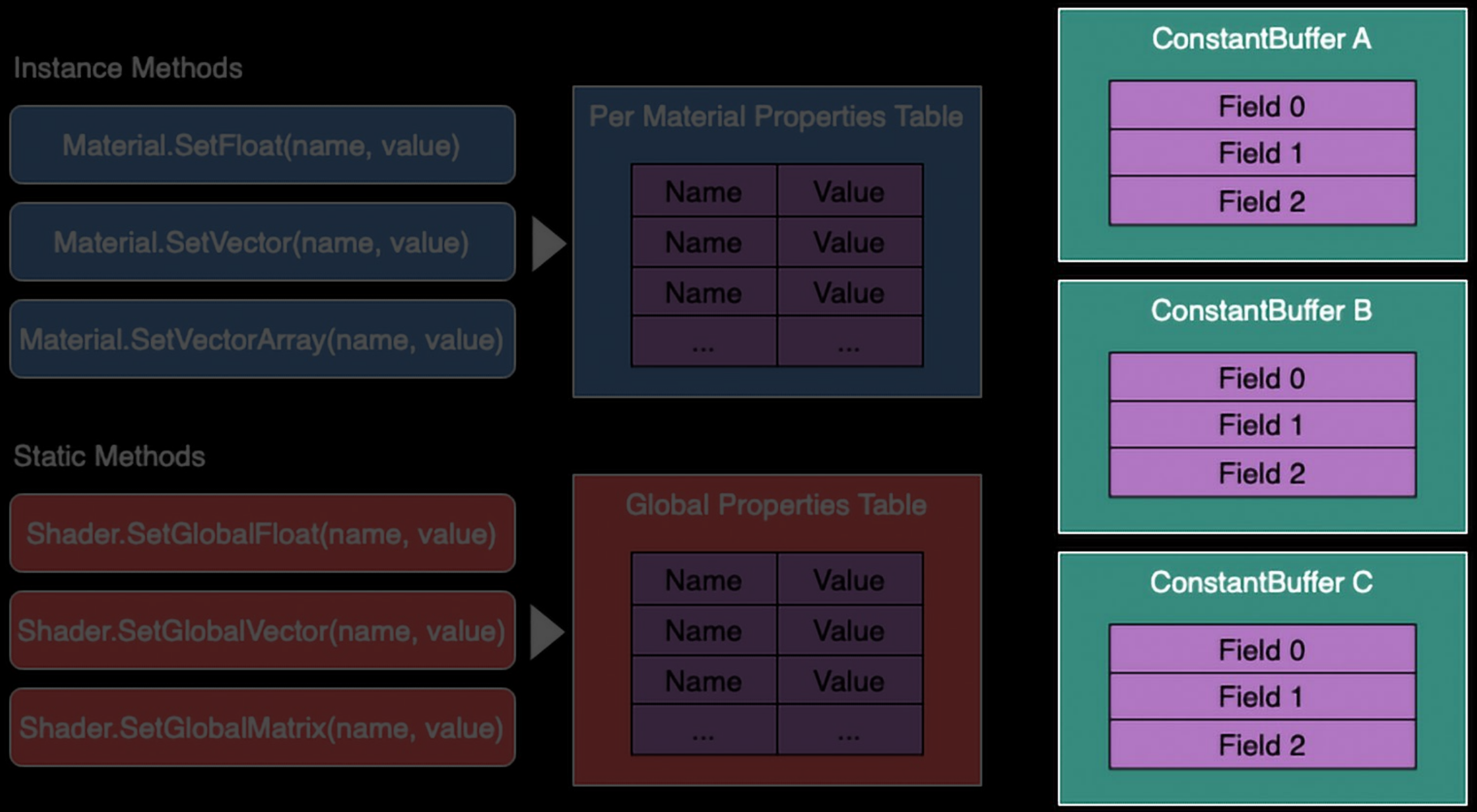
Material.SetVector(name, value)

Material.SetVectorArray(name, value)

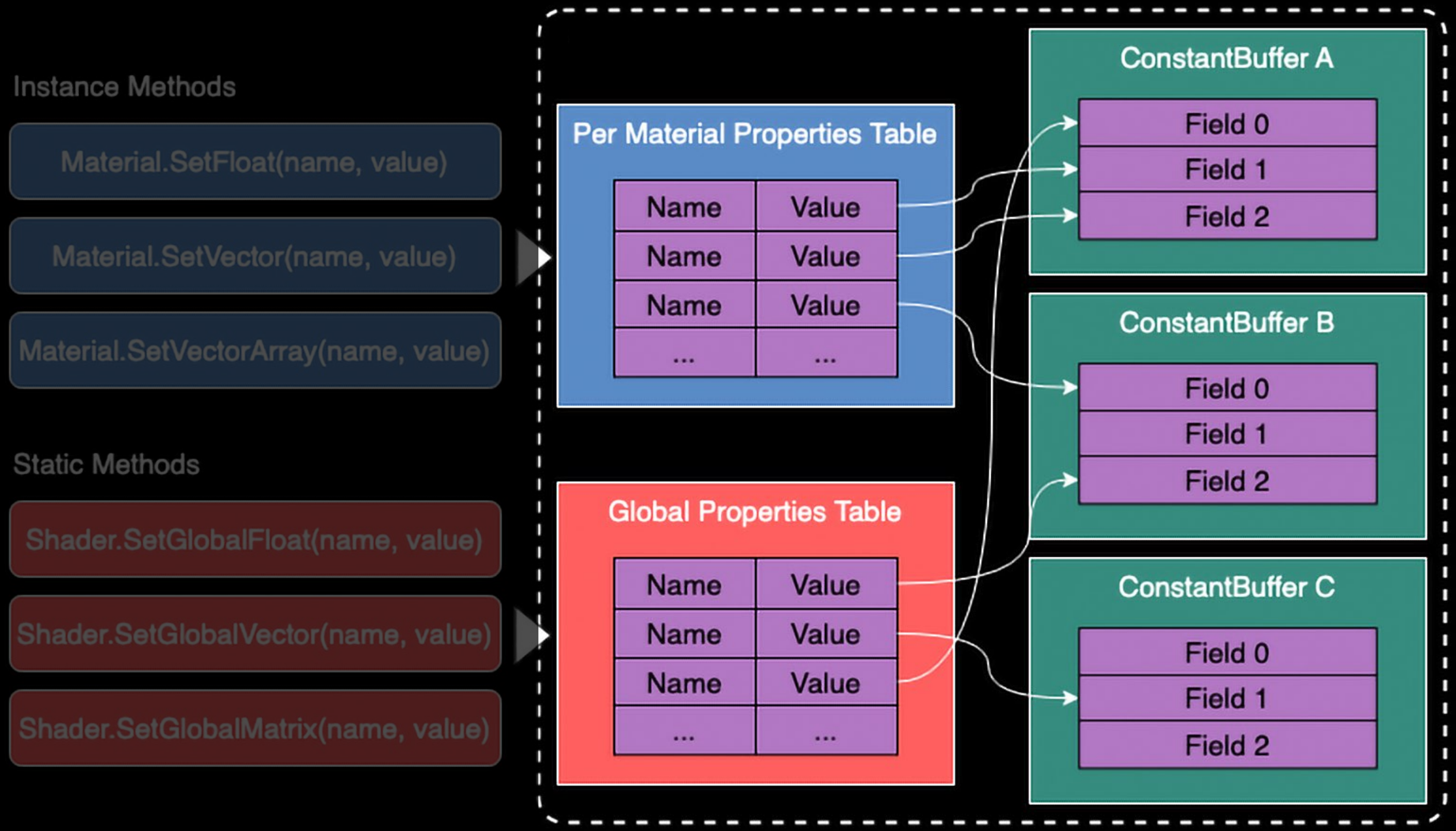
# Constant Buffers: the Heavy Work



# Constant Buffers: the Heavy Work



# Constant Buffers: the Heavy Work

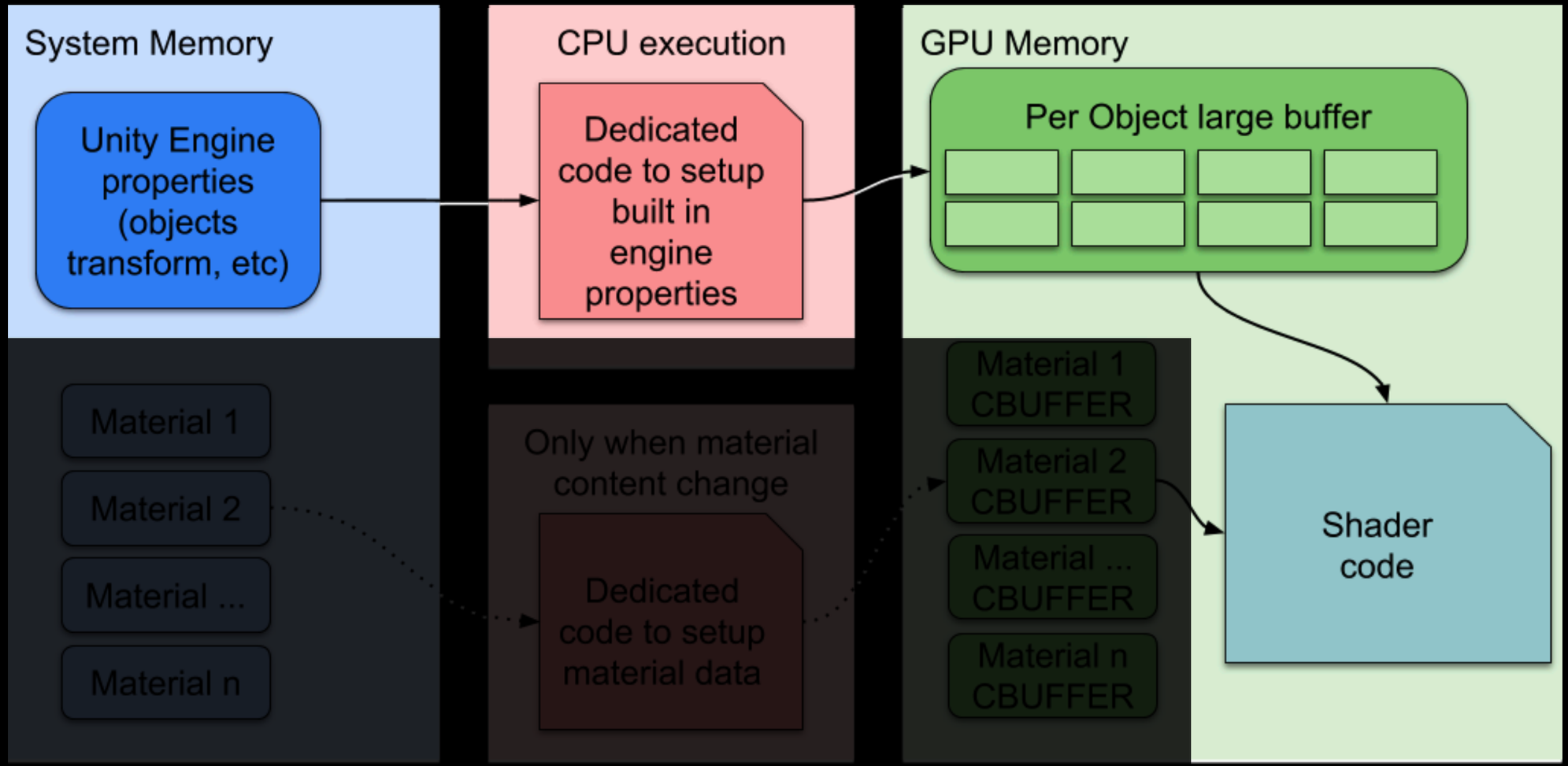


# Constant Buffers: SRP Batcher to the Rescue

Pre-defined constant buffers:

- UnityPerDraw: Built-in per-object data
- UnityPerMaterial: All material parameters

# Constant Buffers: SRP Batcher to the Rescue



# Constant Buffers: A Step Further

Other constant buffers still require Search&Fill when setting up a batch, unless explicitly created and bound:

```
Shader.SetGlobalConstantBuffer("UnityPerCamera", cameraCB, offset: 0, size: cameraCB.size);
```

Explicit lifetime definition: per frame, per camera, per render pass…

Constant buffer layout differs between platforms:

- Always use float4 and float4x4 instead of float3 and float3x3.
- Declare variables in decreasing size order, float4, then float2, then float.

Not friendly for code writing. Unnecessarily increases the size of constant buffers.

Furthermore, no support for half types, which are essential for performance.

# Constant Buffers: Shader Reflection

Shader

```
struct JNUnityPerCamera_Type
{
    float4 _Time;
    float4 _SinTime;
    float4 _CosTime;
    float4 unity_DeltaTime;
    float4 _TimeParameters;
    float3 _CameraPosWS;
    float _UnormCompressValue;
    ...
    float4 _SamplerMipControl;
    half4 _MainShadowData;
    float4 _RelativeInvViewProjection[4];
    ...
};
```

Auto-Gen C# Struct

```
[StructLayout(LayoutKind.Sequential, Pack = 1)]
public struct UnityPerCamera
{
#if UNITY_EDITOR_WIN || UNITY_STANDALONE_WIN
    ...
#elif UNITY_IOS
    ...
    private float3 data_CameraPosWS;
    private ushort _padding3;
    private ushort _padding4;
    ...
    private half4 data_MainShadowData;
    private ushort _padding21;
    private ushort _padding22;
    private ushort _padding23;
    private ushort _padding24;
    ...
```

# Constant Buffers: Shader Reflection

Auto-Gen C# Struct

```
[StructLayout(LayoutKind.Sequential, Pack = 1)]
public struct UnityPerCamera
{
#if UNITY_EDITOR_WIN || UNITY_STANDALONE_WIN
    ...
#elif UNITY_IOS
    ...
    public float4 MainShadowData { set { data_MainShadowData = new half4(value); } }
    ...
    private half4 data_MainShadowData;
    private ushort _padding21;
    private ushort _padding22;
    private ushort _padding23;
    private ushort _padding24;
    ...
```

GDC March 20-24, 2023 San Francisco, CA

# Chapter 3. Scalable Cross-Platform Render Pipeline

- Deferred Rendering: Why & How
- Render Graph: Essentials

# Deferred Rendering: Design Decisions

## Obvious benefits:

- Numerous dynamic lights, deferred decals, and many other rendering features.
- Overdraw from complex geometries will not affect the lighting stage.
- Align rendering results between PC & mobile.

## Important but not often mentioned:

- Splits shaders into G-buffer and lighting stages, resulting in smaller shader sizes.

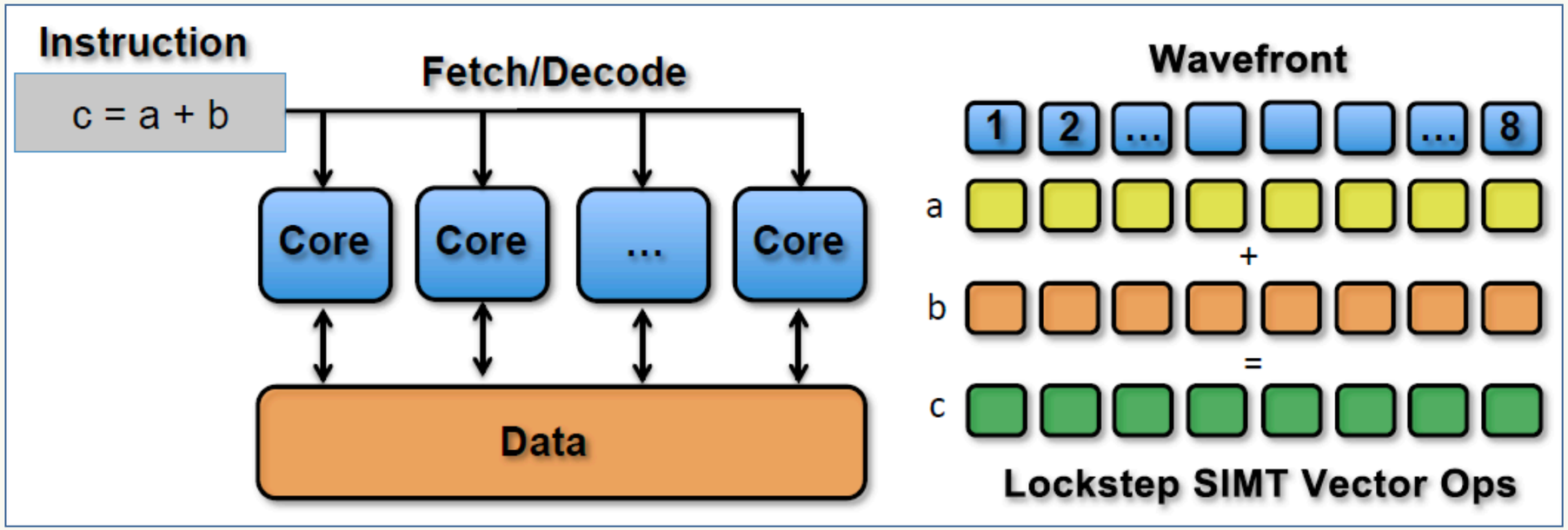
## Further "smaller shaders" decisions:

- Separate shadow mask stage from lighting.
- Use a separate shader to draw each shading model with stencil masks.

Avoid hitting the limit of instruction count and texture sampling. And more >>

# Small Shaders' Favor: A Modern GPU Architecture View

- A GPU consists of multiple Compute Units (CUs).
  - A CU executes multiple wavefronts concurrently.
    - A wavefront includes multiple threads.



# Small Shaders' Favor: A Modern GPU Architecture View

Latency Hiding

- Schedules another wavefront on long latency op.
- No context switching overhead.

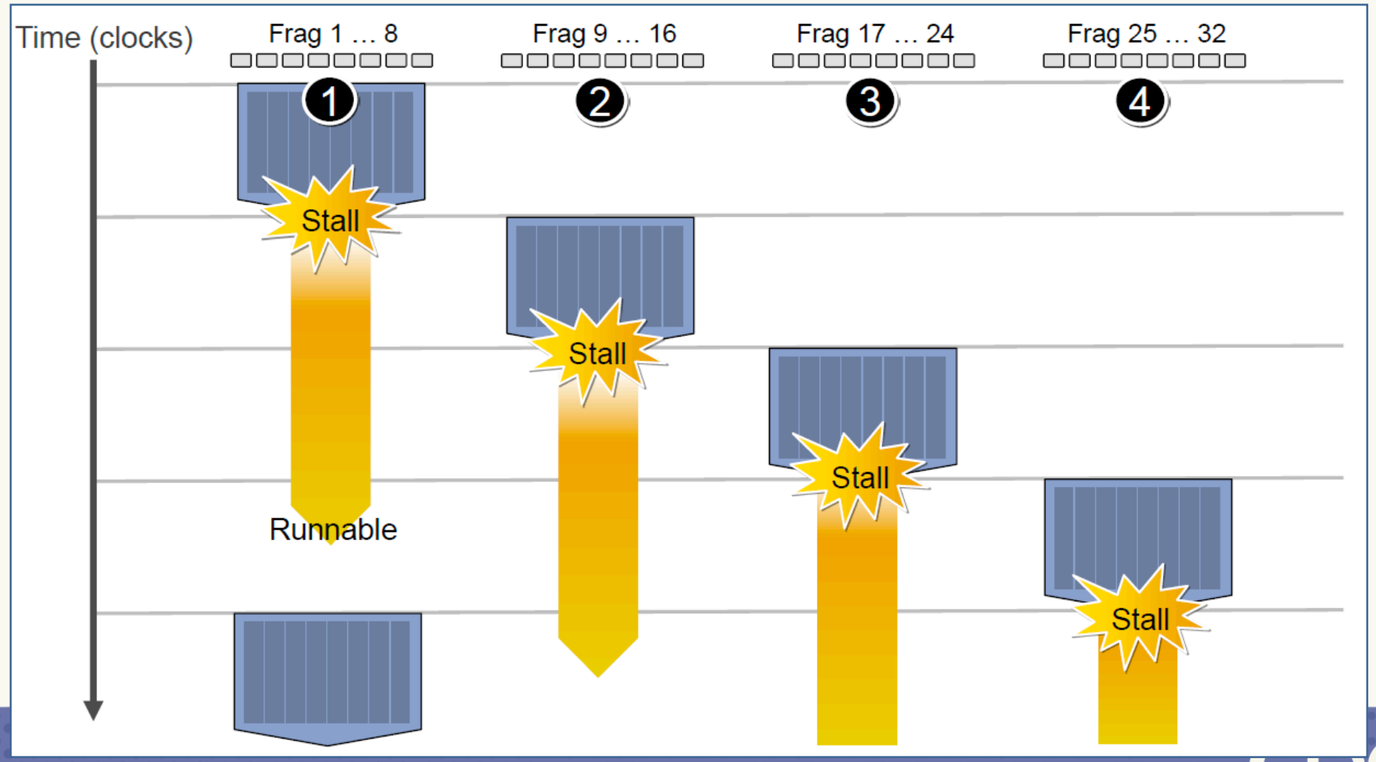
Larger shaders

=> Require more registers

=> Less active wavefronts

=> Lower occupancy

=> Difficult to hide latency

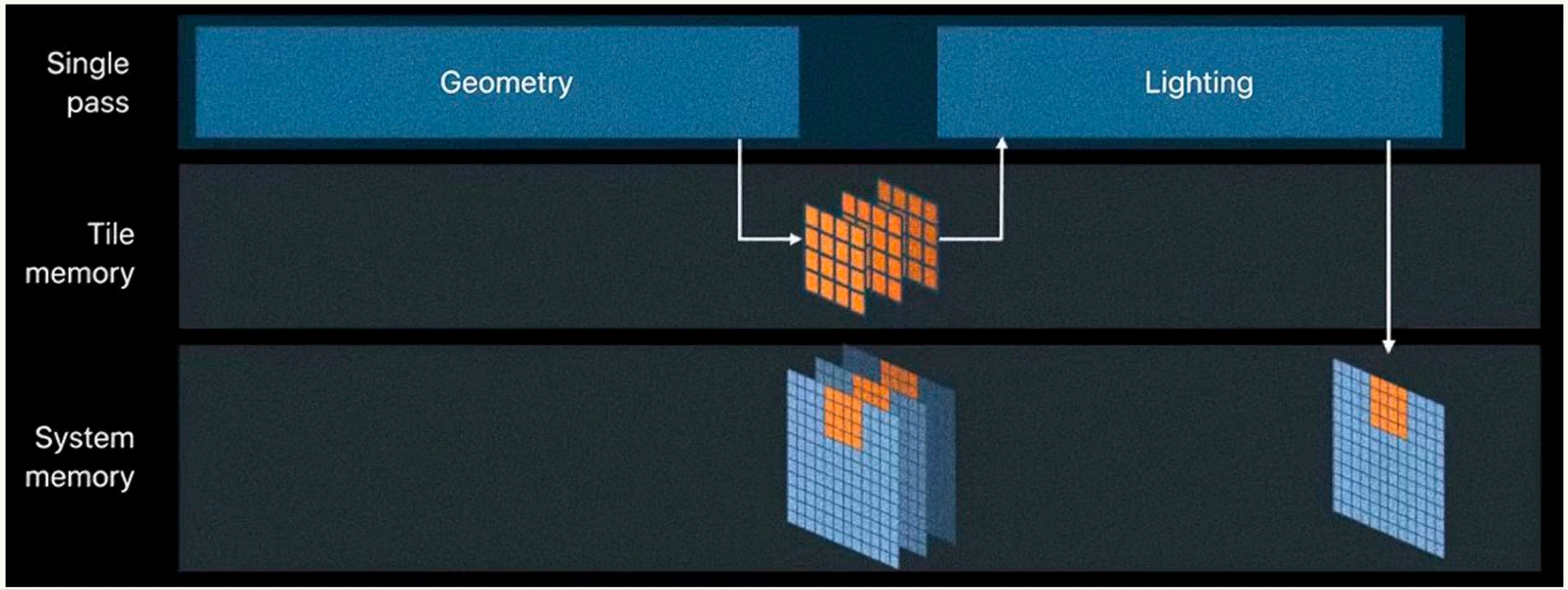


# Mobile GBuffer Storage: Tile-Based Approaches

DirectX/PC: Use the conventional MRT method.

Mobile platforms:

- Single render pass with no RT switches.
- GBuffers only exist on tile memory.



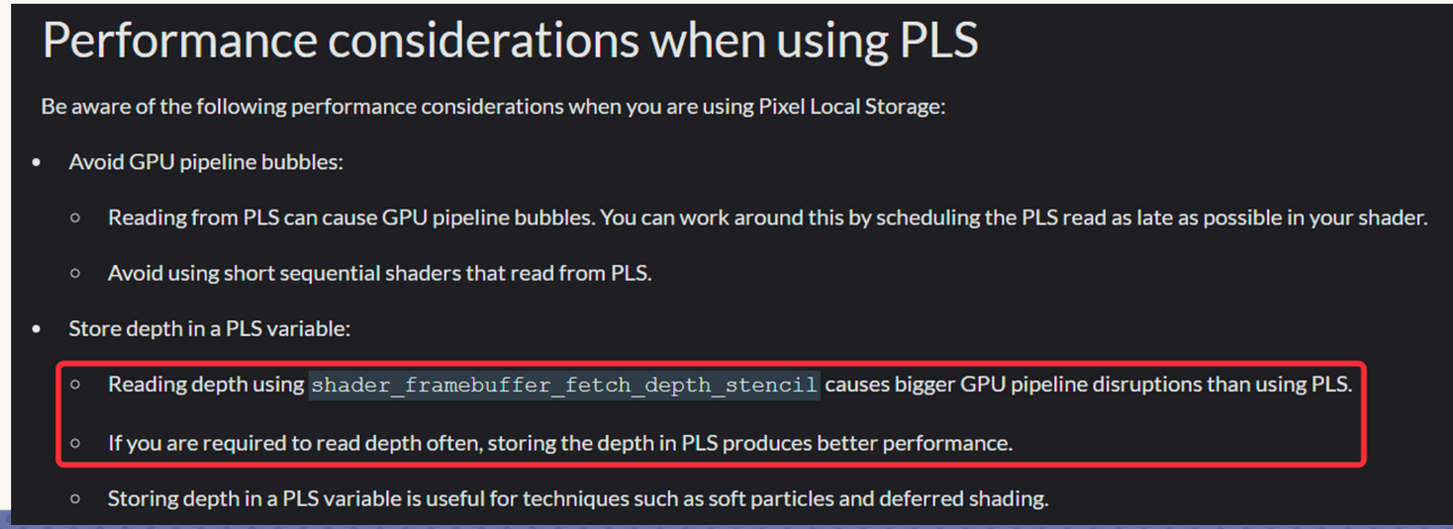
# Mobile GBuffer Storage: Platform Implementations

- Metal: RTs with dontCare store action and memoryless storage mode. Read with programmable blending.
- Vulkan: RTs with STORE_OP_DONT_CARE. Read with subpass input attachments.
- GLES/Adreno: Fetch with QCOM_shader_framebuffer_fetch_noncoherent. Prevent storing with glInvalidateFramebuffer.
- GLES/Mali: Use EXT_shader_pixel_local_storage to read/write per-pixel data in tile memory.

# Depth Storage: Platform Implementations

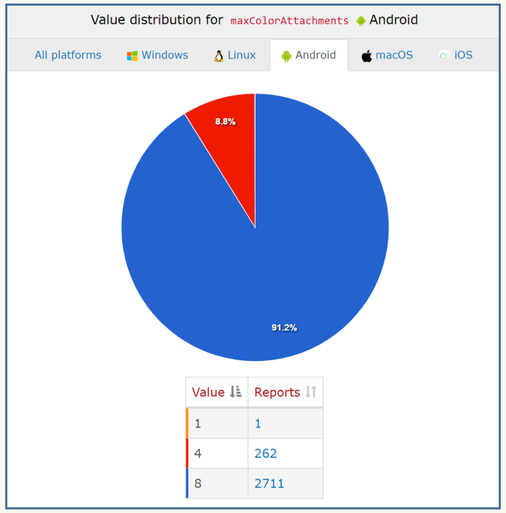
- Metal: Unable to read depth attachment. Use another color attachment as additional depth RT.
- Vulkan: Depth can be used as input attachment, but with bad performance. Use additional depth RT.
- GLES/Adreno: Fetch depth with ARM_shader_framebuffer_fetch_depth_stencil.
- GLES/Mali: ARM_shader_framebuffer_fetch_depth_stencil is supported but with a significant performance penalty. Use PLS instead.

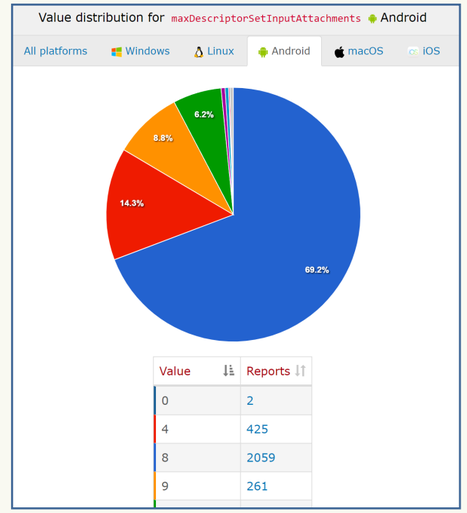
Conclusion: all but GLES/Adreno require additional depth storage.



# GBuffer Layout: Size Limitations

GPUInfo/Vulkan




Ideal GBuffer

- 4+ 32-bit colors values
- A 32-bit depth value

Platform Details

- DirectX & Metal: Support 8 render targets.
- Vulkan: Only 9% support <5 colors. All support ≥4 inputs.
- GLES/Adreno: Both GL_MAX_COLOR_ATTACHMENTS & GL_MAX_DRAW_BUFFERS must be ≥4, according to ES3 specs.
- GLES/Mali: PLS size is limited to 128 bits or four 32-bit color values.

# GBuffer Layout: Size Limitations

Adaptive Graphics API for Adreno:

- Vulkan is used by default if supported.
- For that 9% of devices, GLES is used for a larger size limit.

Conclusion: Except for GLES/Mali, GBuffer can contain at least 4 color values and one depth value.

Source of depth will be discussed equally in the following text.

# Universal GBuffer Layout for all platforms except GLES/Mali

| | R | G | B | A |
|---|---|---|---|---|
| ColorRT (R11G11B10) | Lerp(FogScatteringColor, Emission, FogTransmittance) | | | N/A |
| RT1 (R8G8B8A8) | Albedo | | | Shadow |
| RT2 (R8G8B8A8) | Normal.xyz | | | CubemapIndex |
| RT3 (R8G8B8A8) | Metallic | Roughness | Occlusion | FogTransmittance |
| DepthRT (R32f) | Depth | N/A | N/A | N/A |

Shadow: Written by the shadow mask stage with sampled shadow value.

# Universal GBuffer Layout for all platforms except GLES/Mali

| | R | G | B | A |
|---|---|---|---|---|
| ColorRT (R11G11B10) | Lerp(FogScatteringColor, Emission, FogTransmittance) | | | N/A |
| RT1 (R8G8B8A8) | Albedo | | | Shadow |
| RT2 (R8G8B8A8) | Normal.xyz | | | CubemapIndex |
| RT3 (R8G8B8A8) | Metallic | Roughness | Occlusion | FogTransmittance |
| DepthRT (R32f) | Depth | N/A | N/A | N/A |

CubemapIndex: Index into an array of environment reflection cubes for per-object/per-pixel cube without breaking batching.

Cubemaps are encoded as 2D octahedral maps.

# Universal GBuffer Layout for all platforms except GLES/Mali

| | R | G | B | A |
|---|---|---|---|---|
| ColorRT (R11G11B10) | Lerp(FogScatteringColor, Emission, FogTransmittance) | | | N/A |
| RT1 (R8G8B8A8) | Albedo | | | Shadow |
| RT2 (R8G8B8A8) | Normal.xyz | | | CubemapIndex |
| RT3 (R8G8B8A8) | Metallic | Roughness | Occlusion | FogTransmittance |
| DepthRT (R32f) | Depth | N/A | N/A | N/A |

Rayleigh + Mie + Analytical Height Fog => FogScatteringColor(FSC) & FogTransmittance(T)

FinalColor = (Emission + Lighting) * T + FSC * (1 - T)

= Emission * T + FSC * (1 - T) + Lighting * T

= Lerp(FSC, Emission, T) + Lighting * T

= ColorRT.rgb + Lighting * RT3.a

# Micro GBuffer Layout for GLES/Mali

| | R | G | B | A |
|---|---|---|---|---|
| ColorRT (R11G11B10) | Lerp(FogScatteringColor, Emission, FogTransmittance) | | | N/A |
| RT1 (R8G8B8A8) | Albedo | | | Occlusion |
| RT2 (R8G8B8A8) | Normal.xy (Octahedron Encoding) | | Metallic | Roughness |
| DepthRT (R32f) | Depth + CubemapIndex | N/A | N/A | N/A |

- No shadow: Shadowmaps are sampled in the lighting stage.
- No fog transmittance: Estimated using (1 - linear depth).

# Micro GBuffer Layout for GLES/Mali

| | R | G | B | A |
|---|---|---|---|---|
| ColorRT (R11G11B10) | Lerp(FogScatteringColor, Emission, FogTransmittance) | | | N/A |
| RT1 (R8G8B8A8) | Albedo | | | Occlusion |
| RT2 (R8G8B8A8) | Normal.xy (Octahedron Encoding) | | Metallic | Roughness |
| DepthRT (R32f) | Depth + CubemapIndex | N/A | N/A | N/A |

Two-channel normal value: Encoded using octahedron normal vector encoding.

# Micro GBuffer Layout for GLES/Mali

| | R | G | B | A |
|---|---|---|---|---|
| ColorRT (R11G11B10) | Lerp(FogScatteringColor, Emission, FogTransmittance) | | | N/A |
| RT1 (R8G8B8A8) | Albedo | | | Occlusion |
| RT2 (R8G8B8A8) | Normal.xy (Octahedron Encoding) | | Metallic | Roughness |
| DepthRT (R32f) | Depth + CubemapIndex | N/A | N/A | N/A |

Cubemap index: Packed with the depth value.

# GBuffer Layout: Shading Model Custom Data

| | R | G | B | A |
|---|---|---|---|---|
| ColorRT (R11G11B10) | Lerp(FogScatteringColor, Emission, FogTransmittance) | | | N/A |
| RT1 (R8G8B8A8) | Albedo | | | Shadow |
| RT2 (R8G8B8A8) | Normal.xy (Encoded) | | SecondNormal or Tangent (Encoded) | |
| | Normal.xyz | | | ClearCoatParam |
| RT3 (R8G8B8A8) | Subsurface Scattering | Roughness | TreeImposterParam | FogTransmittance |
| DepthRT (R32f) | Depth | N/A | N/A | N/A |

By replacing metallic, occlusion, and cubemapIndex, or using octahedron-encoded normal vector, we can obtain up to four 8-bit custom data using the universal GBuffer layout.

# Scalability Essentials: Render Graph

Utilize render graph for the max scalability

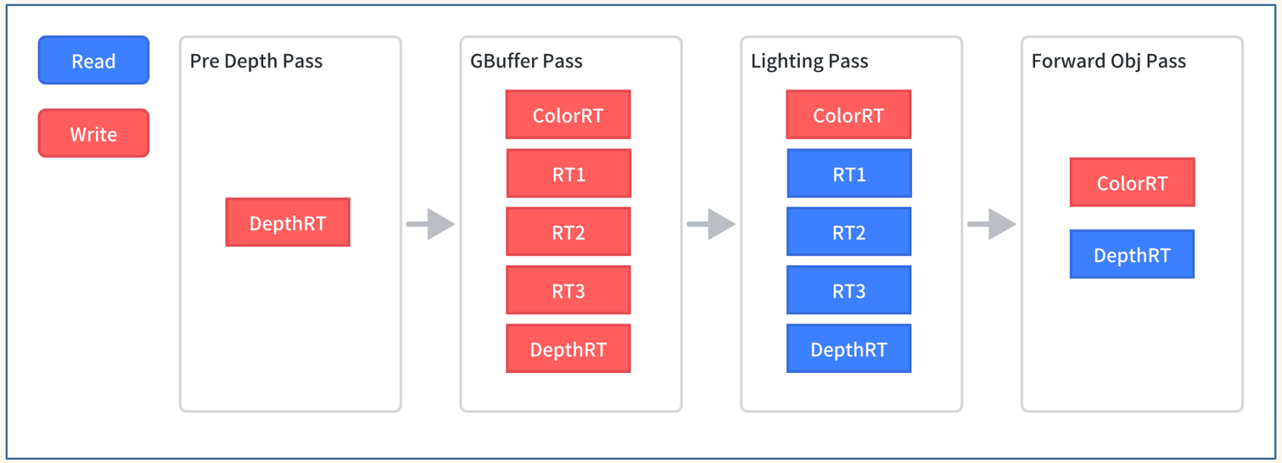
- Platforms x QualitySettings combinations of rendering features require auto dependency resolution.
- Older mobile devices without tile-based GPU architectures require fallback to forward rendering.
- The engine is shared across multiple games, requiring customization of various rendering pipelines.

Optimization techniques for cross-platform scalability >>

# Render Graph: Pass Combiner

Decoupled passes:

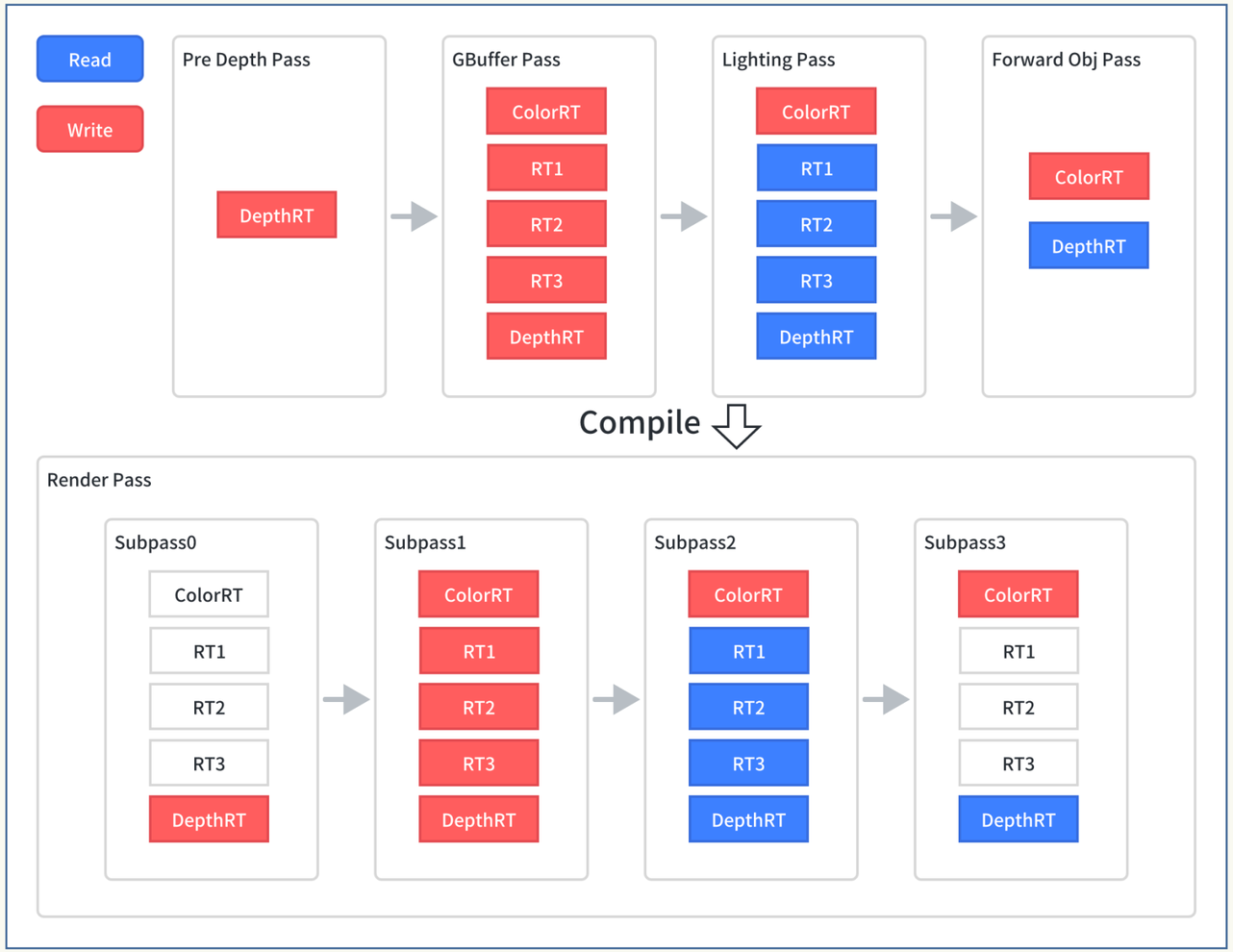
- No care about render passes or subpasses.
- Just declare render targets to read/write.



# Render Graph: Pass Combiner

Render graph compiler, whenever possible:

- Unions render targets of adjacent passes.
- Combines passes into a single render pass.
- GLES: Combines without subpasses, avoiding switching RTs in the middle.



# Render Graph: Auto Store Action

Automatically set store action and storage mode.

After pass combining:

- Set store action to `dontCare` if not consumed by subsequent render passes. Set to `store` otherwise.
- Set storage mode to `memoryless` if not used by previous or subsequent render passes.

Almost impossible to maintain without render graph.

# Render Graph: Pass Library

General-purpose pass library

- Build unique render pipelines by selecting passes.
- Most are common for both forward and deferred rendering.
- Has dedicated passes for deferred rendering.
- Common post-process passes.

```
// Forward renderer example
CreateRenderPass<PerObjectShadowPass>();
CreateRenderPass<AdditionalLightsShadowsPass>();
CreateRenderPass<MainLightStaticShadowCachePass>();
CreateRenderPass<MainLightCascadedShadowPass>();

CreateRenderPass<LightsCullingPass>();
CreateRenderPass<CreateForwardRTs>();

CreateRenderPass<PreDepthPass>();
CreateRenderPass<ForwardObjectsPass>()
    .Init(opaque: true, alphaTest: false);
CreateRenderPass<ForwardObjectsPass>()
    .Init(opaque: true, alphaTest: true);

CreateRenderPass<SkyBoxPass>();
CreateRenderPass<ForwardObjectsPass>()
    .Init(opaque: false, alphaTest: false);

// Common post-process passes
...
```

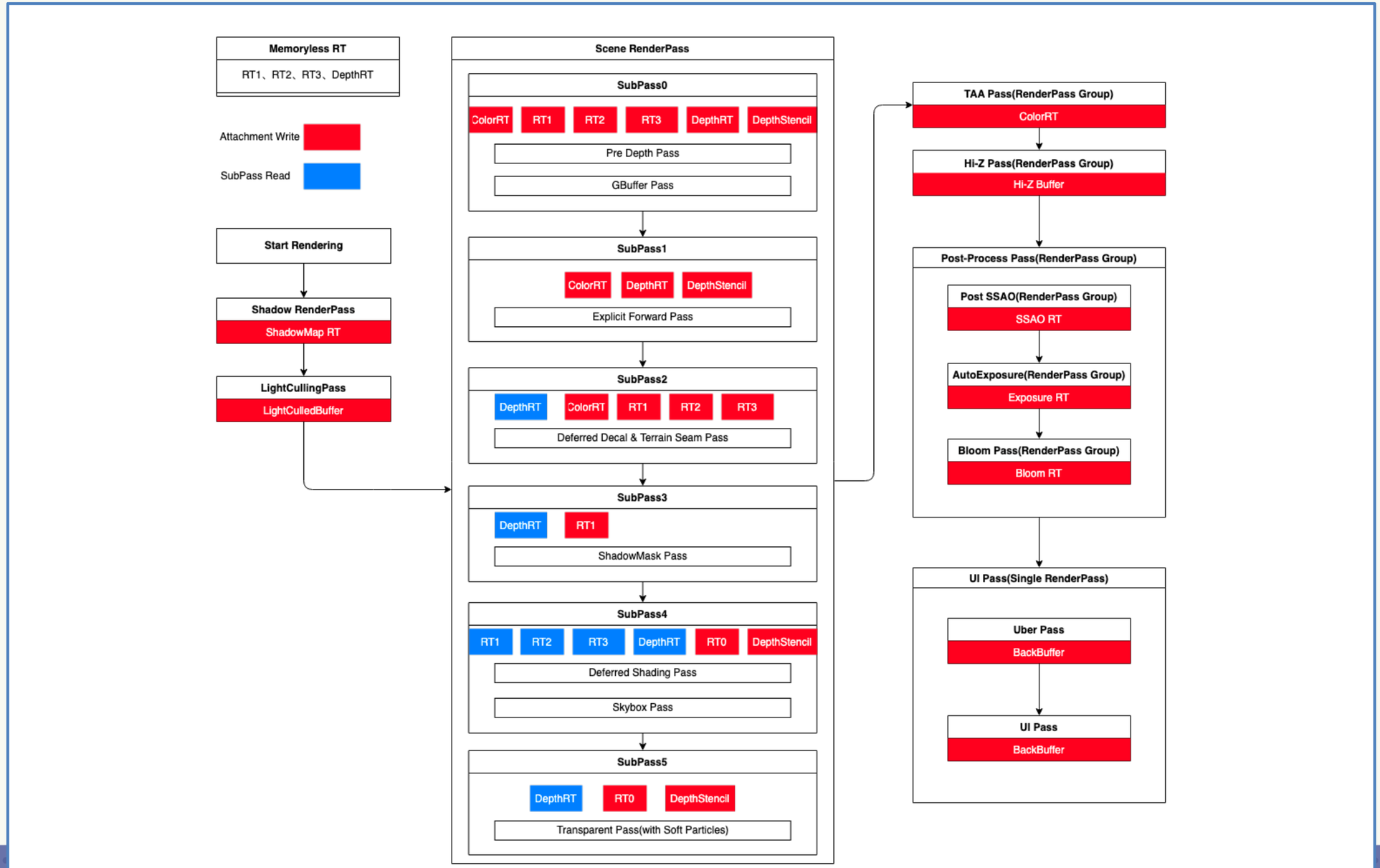
# Render Graph: Pass Library

General-purpose pass library

- Build unique render pipelines by selecting passes.
- Most are common for both forward and deferred rendering.
- Has dedicated passes for deferred rendering.
- Common post-process passes.

```
// Deferred renderer example
CreateRenderPass<PerObjectShadowPass>();
CreateRenderPass<AdditionalLightsShadowsPass>();
CreateRenderPass<MainLightStaticShadowCachePass>();
CreateRenderPass<MainLightCascadedShadowPass>();

CreateRenderPass<LightsCullingPass>();
CreateRenderPass<CreateDeferredRTs>();

CreateRenderPass<PreDepthPass>();
CreateRenderPass<GBufferPass>()
    .Init(opaque: true, alphaTest: false);
CreateRenderPass<GBufferPass>()
    .Init(opaque: true, alphaTest: true);
CreateRenderPass<DeferredDecalPass>();
CreateRenderPass<GTAOPass>();

CreateRenderPass<DeferredShadowMaskPass>();
CreateRenderPass<DeferredLightingPass>();

CreateRenderPass<SkyBoxPass>();
CreateRenderPass<ForwardObjectsPass>()
    .Init(opaque: false, alphaTest: false);

// Common post-process passes
...
```

# Render Graph: Frame



# Summary

- Hybrid OC: Render the minimum number of objects.
- Multi-threaded SRP: CPU cost as low as possible.
- Deferred & RenderGraph: GPU cost as low as possible.

High-quality real-time rendering for mobiles and PCs.

GDC
March 20-24, 2023
San Francisco, CA

# Bonus Chapter: Rendering Features

#GDC23

# Rendering Feature: Deferred Decals

Both universal and micro GBuffer support deferred decals.

Universal GBuffer:

- All the data that needs blending are placed in the RGB channels.
- The data that does not need blending are placed in the A channels.
- The shader outputs the blend factors to the A channels.
- Writing to the A channels is masked.

Micro GBuffer:

- PLS is a form of programmable blending.
- All the data that need blending are read from PLS.
- Normal vector decoding and decal blending are done in the shader.
- The blended data is then written back into PLS.

# Rendering Feature: Soft Terrain Seams

Soften seams between terrain and rocks/buildings/trees.

- Alpha-blending?
- Sampling terrain virtual textures?
- Jittering pixel depth offset + TAA?

High costs/unstable results

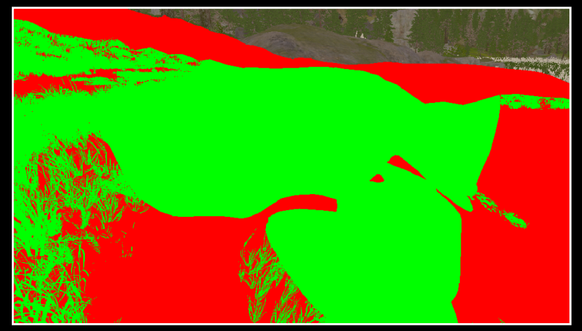
# Soft Terrain Seams: The Method

Render the terrain as usual.

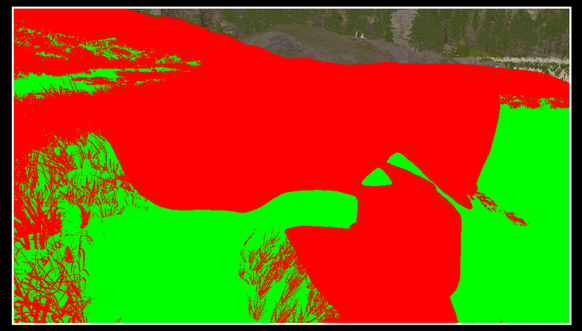
- Mark covered pixels with stencil.

Render the terrain again after GBuffer pass.

- Vertices slightly shifted toward the camera, increasing terrain coverage.
- Stencil test skips pixels covered by the first terrain pass.
- Only render terrain patches within 40m.



Stencil Test



Depth Test

Both Passed

The Seam

# Soft Terrain Seams: Equations

How much should each vertex be shifted forward?

```
float bias = _BiasParam * min(1.0f / abs(viewDirWS.y), 5.0f);
float4 positionCS = WorldToHClip(positionWS + viewDirWS * bias);
```

- Determined by artist set parameter and view direction.
- If camera facing forward, shift more for terrain coverage.
- If looking downward, a slight shift is enough.

# Soft Terrain Seams: Equations

How to determine the blend factor for the seam?

```
float terrainDepth = LinearEyeDepth(fragInput.positionCS.z);
float sceneDepth = LinearEyeDepth(depthRT);
float value = abs(sceneDepth - terrainDepth);
value *= smoothstep(2, 1, value);
fragOutput.a = saturate(value * _ScaleParam);
```

- Determined by depth difference.
- Zero diff means to show the rock surface.
- Larger diffs mean to gradually blend with the first pass terrain.
- Too large diffs mean to show the backgrounds.

# Soft Terrain Seams: More Applications

Can be applied to opaque decals to remove hard edges.

Alpha-blending off and minimal overdraw.


# Thanks

Chao Wang
Email: baixiaolou@bytedance.com
Discord: https://discord.gg/ZydzrEFShh