Vulkanised 2024

The 6th Vulkan Developer Conference
Sunnyvale, California | February 5-7, 2024

Mastering Chaos: Navigating Vulkan synchronization on O3DE engine.

Akio Gaule, O3DE

About this talk

- A practical view of Vulkan synchronization in a 3D application
- A “synchronized” journey through the rendering pipeline
- Who is presenting?
  - Akio Gaule, Graphics Engineer
  - Involved in O3DE graphic engine since early designs

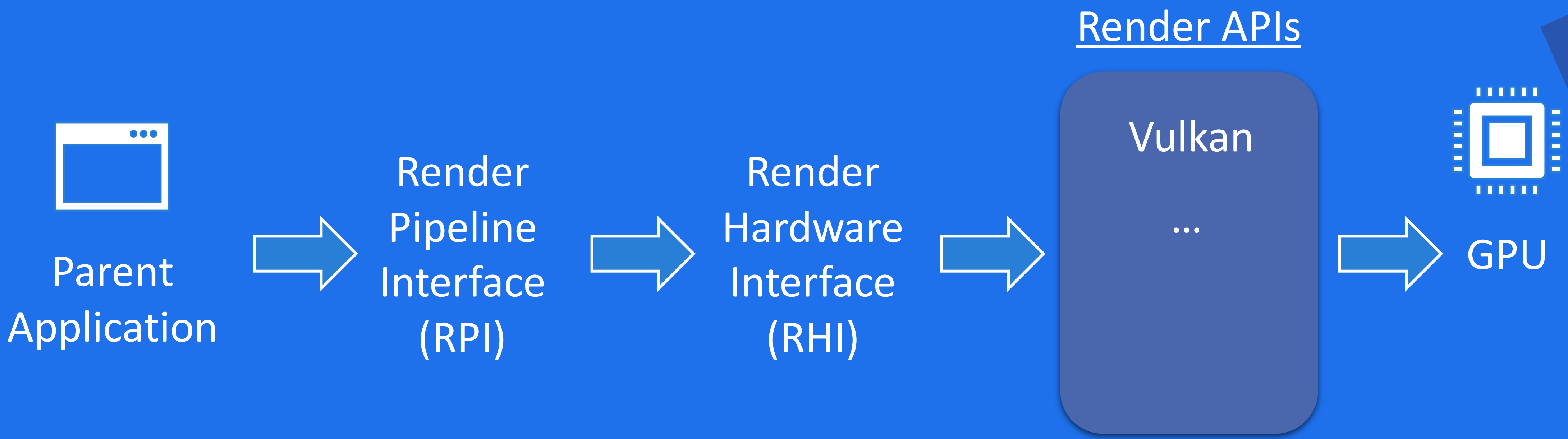
Agenda

- O3DE Overview
- Prepare: Gathering synchronization data
- Compile: Processing the data
- Execute: Submit GPU work

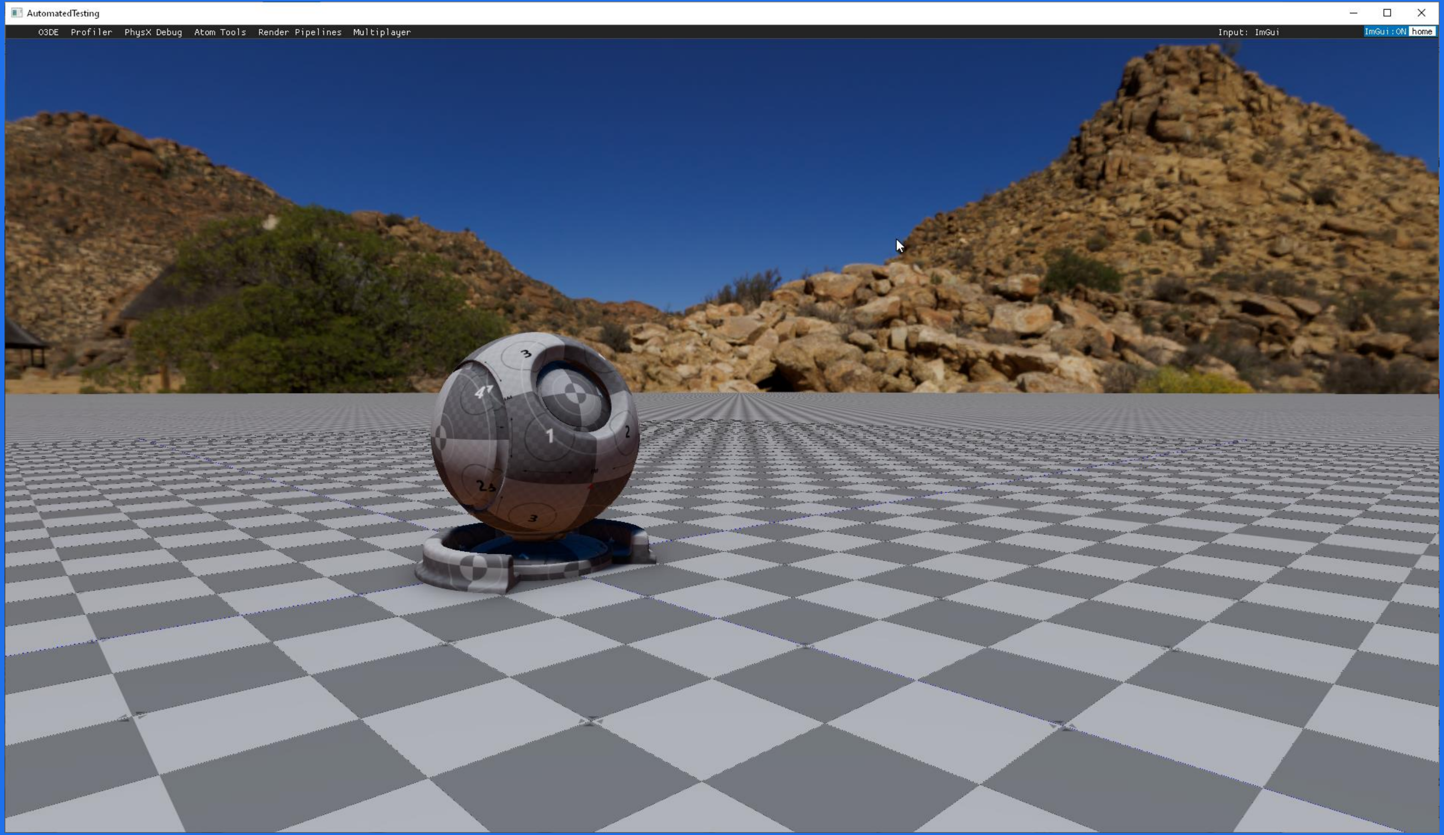
What is O3DE?

- O3DE is an open-source, real-time, multi-platform 3D engine.
- Supports Vulkan, DX12 and Metal rendering APIs.
- Runs on Windows, Linux, Android, iOS and MacOS.

Atom Overview

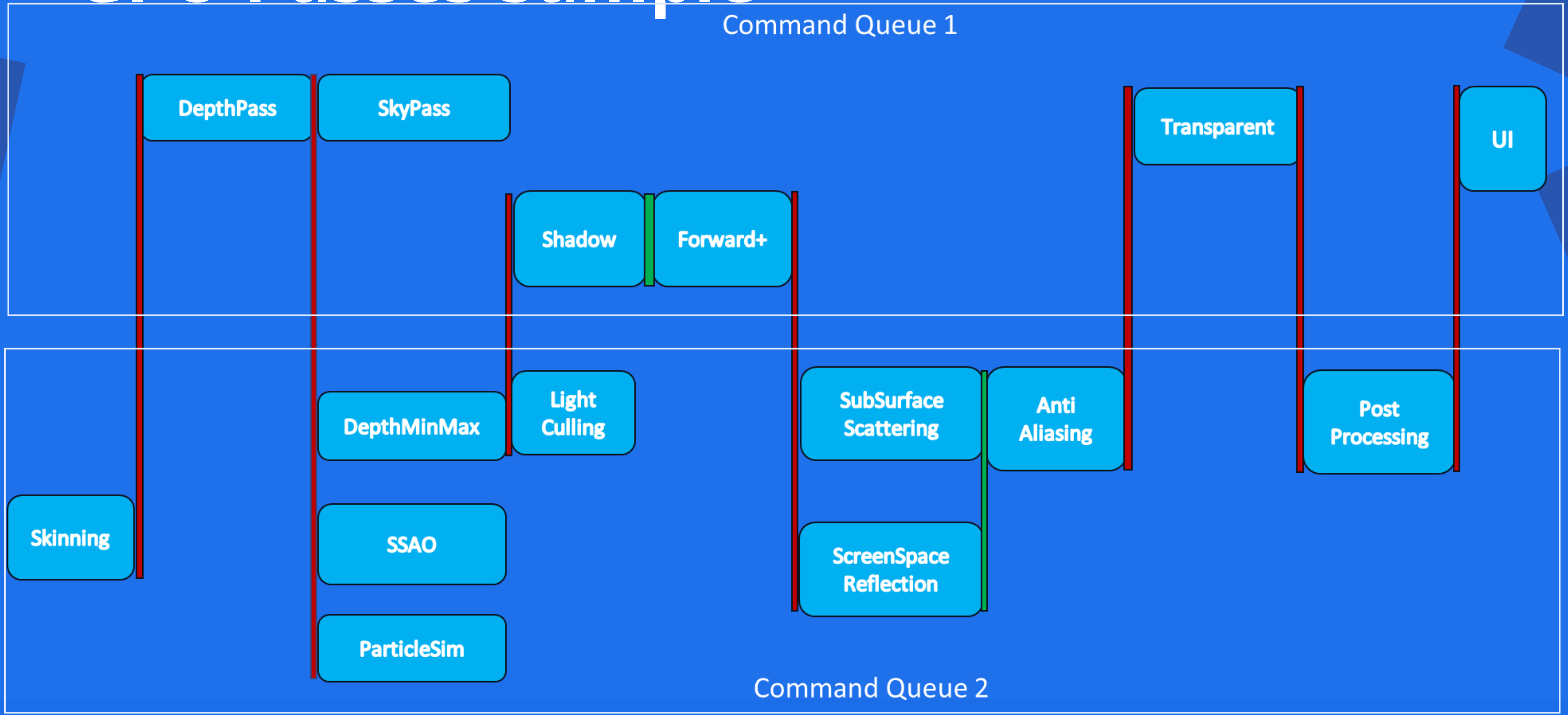


Synchronized "Chaos"



Action #	Name
	Frame #2703
0	Capture Start
1-5	Root
	PostProcessPass.LightAdaptation.LookModificationTransformPass.BlendColorGradingLuts
	OpaquePass.DiffuseGlobalIlluminationPass.Clear.IrradianceOutput
	MorphTargetPass
	SkinningPass
5-7	DepthPrePass.DepthPass
7-8	DepthPrePass.MSAAResolveDepthPass
8-9	DepthPrePass.DepthToLinearDepthPass
9-10	MotionVectorPass.CameraMotionVectorPass
	MotionVectorPass.MeshMotionVectorPass
	LightCullingPass.DepthTransparentMinPass
	LightCullingPass.DepthTransparentMaxPass
10-12	Shadows.Cascades.0
12-14	Shadows.Cascades.1
14-15	Shadows.Cascades.2
15-16	Shadows.Cascades.3
	LightCullingPass.LightCullingTilePreparePass
17-18	Shadows.FullscreenShadowPass
18-19	Shadows.FullscreenShadowBlur.VerticalBlur
20-21	Shadows.FullscreenShadowBlur.HorizontalBlur
	LightCullingPass.LightCullingPass
	LightCullingPass.LightCullingRemapPass
24-26	OpaquePass.Forward
	OpaquePass.ForwardSubsurface
	OpaquePass.MSAAResolveScatterDistancePass
26-27	OpaquePass.DiffuseGlobalIlluminationPass.DiffuseCompositePass
	OpaquePass.MSAAResolveDiffusePass
27-28	OpaquePass.SubsurfaceScatteringPass
29-30	OpaquePass.Ssao.DepthDownsample
31-32	OpaquePass.Ssao.SsaoCompute
33-34	OpaquePass.Ssao.SsaoBlur.VerticalBlur
35-36	OpaquePass.Ssao.SsaoBlur.HorizontalBlur
37-38	OpaquePass.Ssao.Upsample
	OpaquePass.Ssao.ModulateWithSsao
	OpaquePass.ReflectionsPass.ReflectionProbeStencilPass
	OpaquePass.ReflectionsPass.ReflectionProbeBlendWeightPass
40-41	OpaquePass.ReflectionsPass.ReflectionGlobalFullscreenPass
	OpaquePass.ReflectionsPass.ReflectionProbeRenderOuterPass
	OpaquePass.ReflectionsPass.ReflectionProbeRenderInnerPass
41-42	OpaquePass.SkyBoxPass
42-43	OpaquePass.ReflectionCompositePass
43-44	OpaquePass.MSAAResolveSpecularPass
44-45	OpaquePass.DiffuseSpecularMergePass
	TransparentPass.TransparentPass
45-46	PostProcessPass.SMAA1xApplyLinearHDRColorPass.SMAANeighborhoodBlending
	PostProcessPass.LightAdaptation.DownsampleSinglePassLuminance
47-48	PostProcessPass.LightAdaptation.LookModificationTransformPass.LookModificationComposite
48-49	PostProcessPass.LightAdaptation.DisplayMapperPass.AcesOutputTransform
49-50	AuxGeomPass
	LyShinePass.LyShineChildPass
50-58	UIPass.2DPass
58-61	UIPass.ImGuiPass
61-62	CopyToSwapChain
62	vkQueuePresentKHR(**SwapChainImage_0**)

GPU Passes Sample

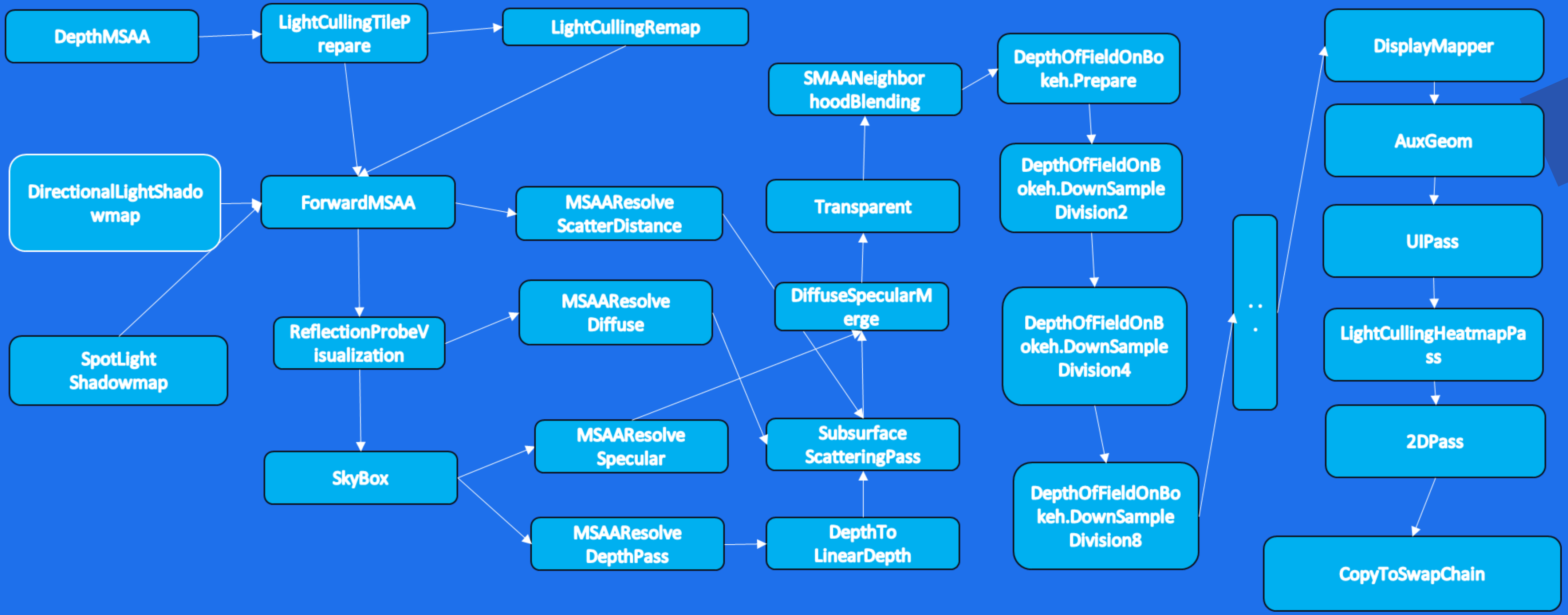


Atom Design Principles

- Frame structure must be declared beforehand.
- Passes encapsulate all graphics work.
- C++ or Data Driven (JSON).
- All resource synchronization is done under the hood by the engine. No API for direct synchronization.

Frame Graph

A directed acyclic graph of all the passes and the resources in a frame.



Frame Graph Overview

Phase 1	Phase 2	Phase 3
Frame graph creation (Serial)	**Frame graph compilation** (Partly parallelized)	**Frame graph execution** (Parallel)

Resource usage declaration

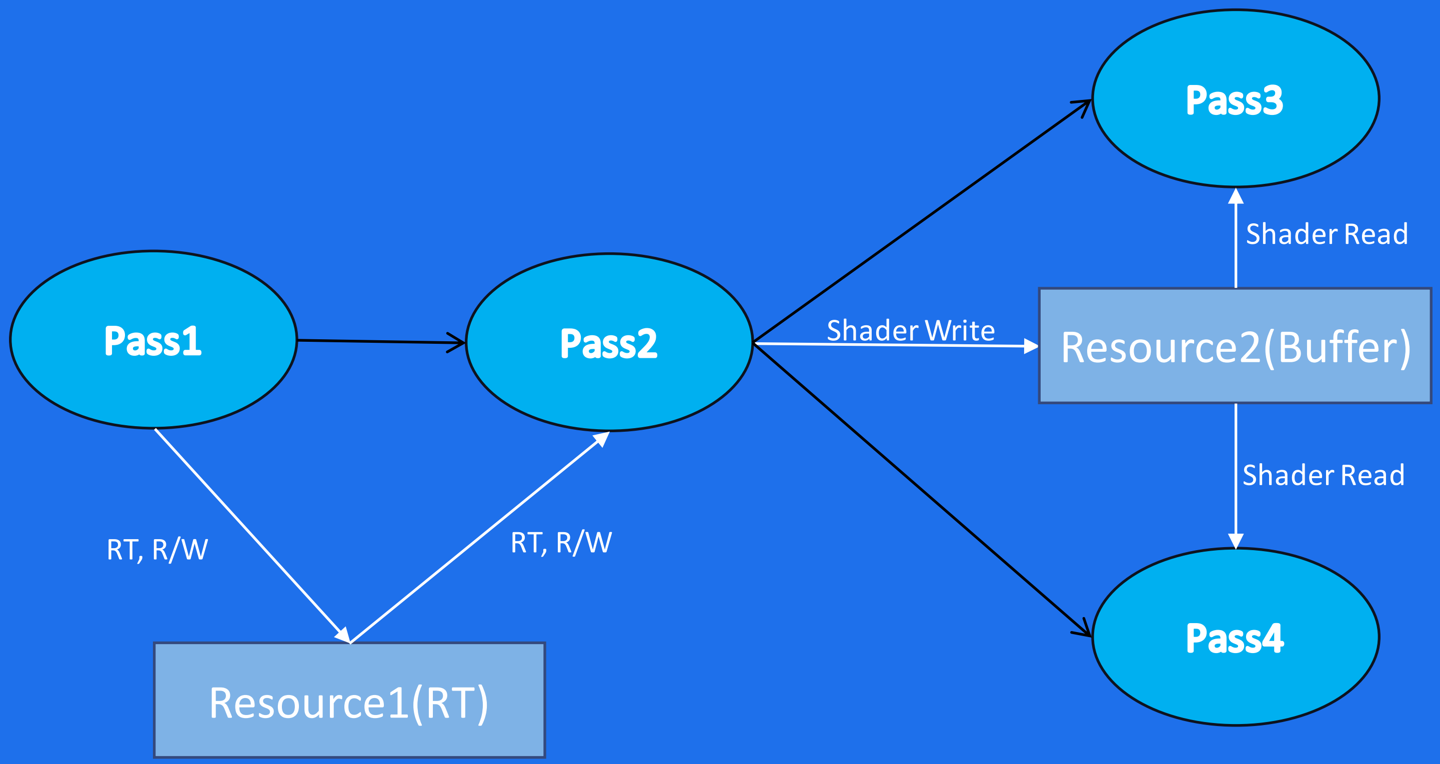
```
// Inputs...
{
    "Name": "BRDFTextureInput",
    "ShaderInputName": "m_brdfMap",
    "SlotType": "Input",
    "ScopeAttachmentUsage": "Shader"
},
{
    "Name": "DirectionalLightShadowmap",
    "ShaderInputName": "m_directionalLightShadowmap",
    "SlotType": "Input",
    "ScopeAttachmentUsage": "Shader",
    "ImageViewDesc": {
        "IsArray": 1
    }
},
{
    "Name": "ProjectedShadowmap",
    "ShaderInputName": "m_projectedShadowmaps",
    "SlotType": "Input",
    "ScopeAttachmentUsage": "Shader",
    "ImageViewDesc": {
        "IsArray": 1
    }
},
```

```
// Outputs...
{
    "Name": "DepthStencilOutput",
    "SlotType": "Output",
    "ScopeAttachmentUsage": "DepthStencil",
    "LoadStoreAction": {
        "ClearValue": {
            "Type": "DepthStencil"
        },
        "LoadAction": "Clear",
        "LoadActionStencil": "Clear"
    }
},
{
    "Name": "LightingOutput",
    "SlotType": "Output",
    "ScopeAttachmentUsage": "RenderTarget",
    "LoadStoreAction": {
        "LoadAction": "Clear"
    }
}
```

Resource usage types

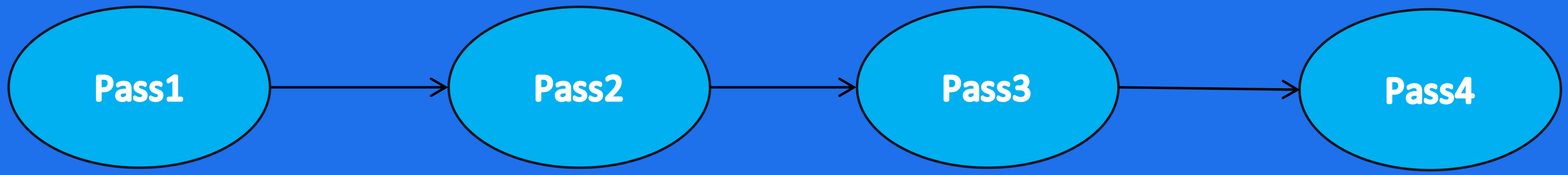
```cpp
enum class ScopeAttachmentUsage : uint32_t
{
    //! Error value to catch uninitialized usage of this enum
    Uninitialized = 0,

    //! Render targets use the fixed-function output merger stage on the graphics queue.
    RenderTarget,

    //! A depth stencil attachment uses the fixed-function depth-stencil output merger stage on the graphics queue.
    DepthStencil,

    //! A shader attachment is exposed directly to the shader with either read or read-write access.
    Shader,

    //! A copy attachment is available for copy access via CopyItem.
    Copy,

    //! A resolve attachment target
    Resolve,

    //! An attachment used for predication
    Predication,

    //! An attachment used for indirect draw/dispatch.
    Indirect,

    //! An attachment that allows reading the output of a previous subpass.
    SubpassInput,

    //! An attachment used as Input Assembly in the scope. Only needed for buffers that are modified by the GPU (e.g Skinned Meshes), not
    //! for static data.
    InputAssembly,

    //! An attachment used for specifying the framebuffer shading rates.
    ShadingRate,

    Count,
};
```

Track explicit resource dependency



Frame Graph Creation

- Remove orphan nodes
- Detect cycles
- Topological sort of nodes and build a flat list



Frame Graph Compiling

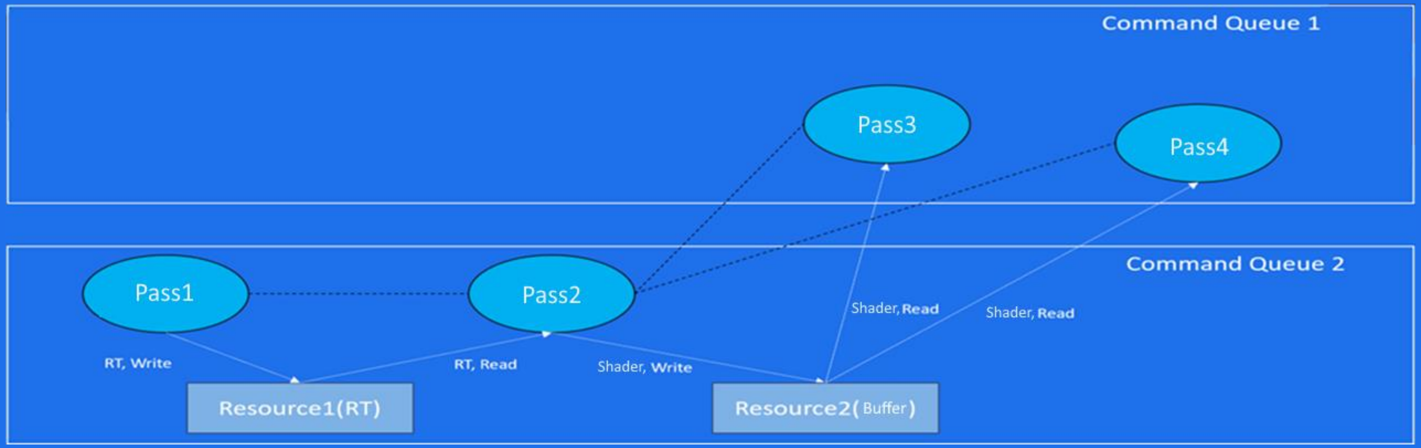
- Create synchronization primitives for different resource utilization
- Generated barriers and semaphores are stored within the pass
- They will be emitted during the Frame Graph execution phase

Resource Usage Barriers

- VkPipelineStageFlags are deduced from the resource usage
- VkAccessFlags are deduced from the access (read/write) and the usage
- VkImageLayout is deduced from resource usage
- Keep track of images sub resources layouts (using an interval map structure)

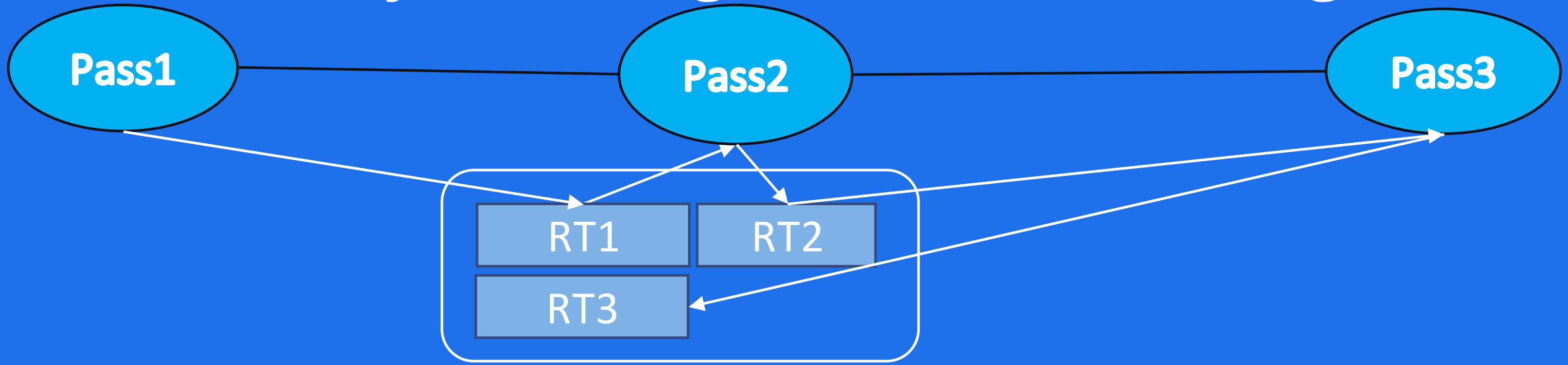
Cross queue synchronization

- Generate semaphores to synchronize passes along queues
- Generate release and acquire barriers to transfer ownership of resources
- Keep track of sub resources queue ownership



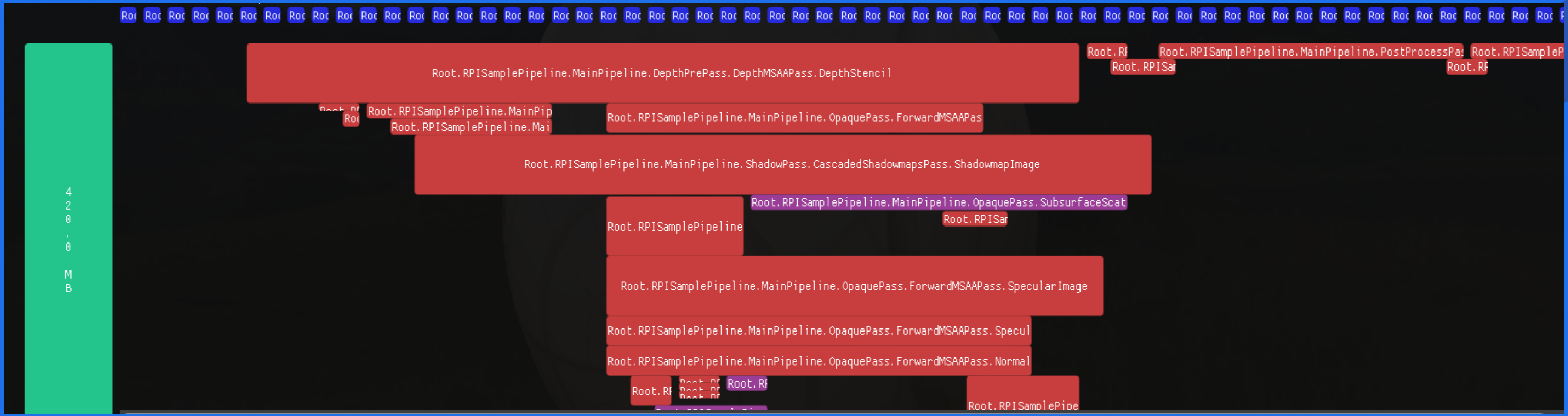
Aliasing barriers

- We know exactly how long a resource is being used



- Transient resources: resources valid only during the current frame
- No need to preserve content -> We can share memory
- Generate barriers on first usage due to memory overlap

Aliasing barriers



Clear and Resolve barriers

- Some passes use clear operations before beginning (we may not be able to use a renderpass clear value)
- Some passes have a resolve operation at the end
- We need to transition the resource to the proper state before the clear or resolve operation

Frame Graph Execution

- Passes record their GPU work into one or multiple command buffers
- Optimize generated barriers
- Pass barriers and semaphores are recorded into the command buffer

Optimize barriers

- Memory barriers are group into one
- Multiple barriers over the same sub resource in the same stage are merge into one
- Transform barriers into subpass dependencies if possible
- Use initial and final layout of VkAttachmentDescription for layout transitions in a renderpass

Submit barriers and semaphores

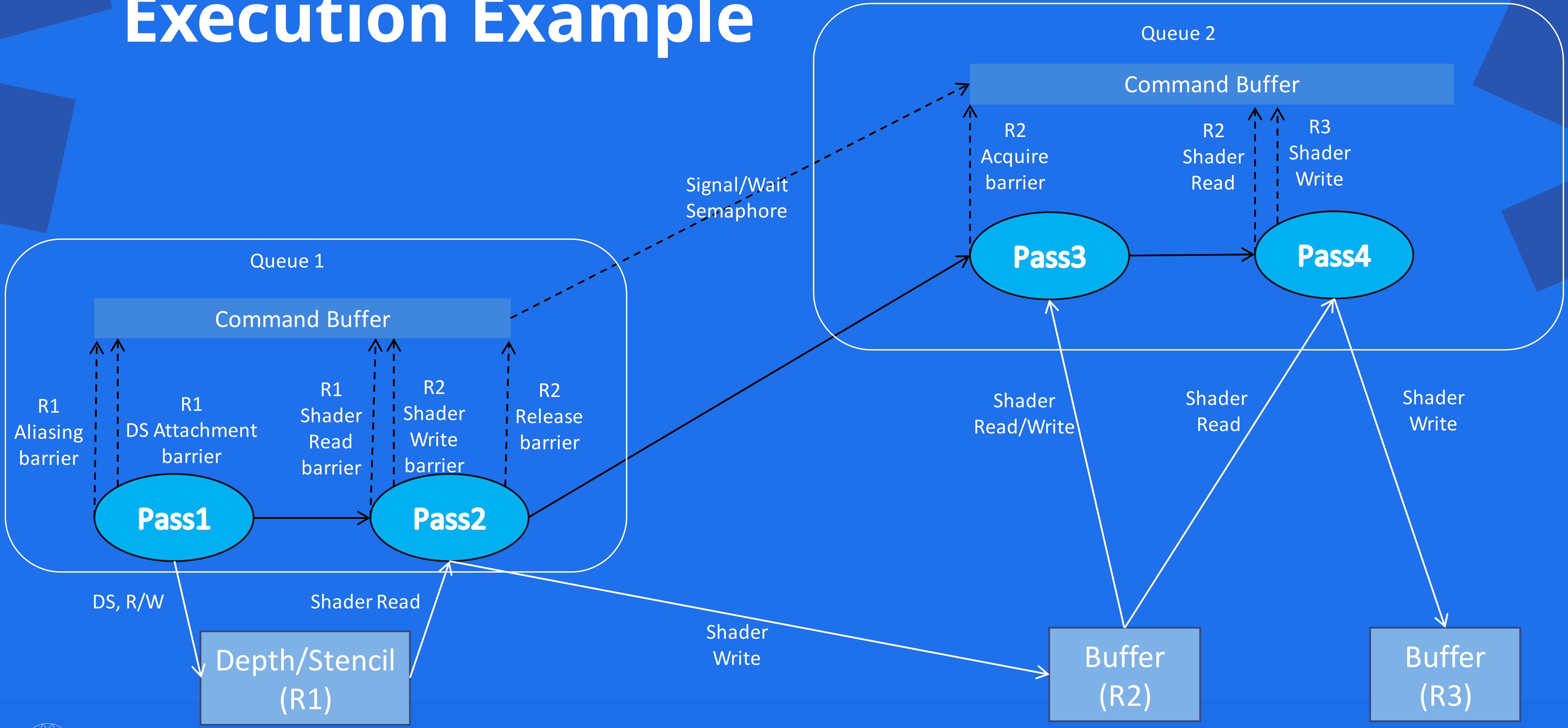
1. Aliasing
2. Clear
3. Prologue

   Pass Work

4. Epilogue
5. Resolve

- Wait and signal semaphores are added when submitting to the queue

Execution Example



Questions?

- https://o3de.org
- https://github.com/o3de
- Discord channel #sig-graphics-audio

Future Improvements

- Combine more barriers between stages (e.g. aliasing barriers with usage barriers)
- Collect more information about when a resource will be used (compute, vertex or fragment stage)
- Visual Debugging tools