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”
GPU Passes Sample

Command Queue 1

- DepthPass
- SkyPass
- Shadow
- Forward+
- Transparent
- UI

Command Queue 2

- Skinning
- DepthMinMax
- Light Culling
- SubSurface Scattering
- Anti Aliasing
- Post Processing
- SSAO
- ParticleSim
- ScreenSpace Reflection
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
- Frame graph creation
  (Serial)

Phase 2
- Frame graph compilation
  (Partly parallelized)

Phase 3
- Frame graph execution
  (Parallel)
Resource usage declaration

```json
// 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

```c
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

Pass 1 → Pass 2
- Shader Write
- RT, R/W

Pass 2 → Resource 1 (RT)
- RT, R/W

Resource 2 (Buffer)
- Shader Read
- Shader Read

Pass 3
- Shader Read

Pass 4
- Shader Read
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

![Cross queue synchronization diagram]
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 grouped into one
• Multiple barriers over the same sub resource in the same stage are merged 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