Godot: Using Vulkan in an Open Source Game Engine

Clay John, W4 Games / Godot Engine
Godot

- 2D and 3D game engine
- Open source (MIT licence)
- Godot 4 released March 2023
- Cross platform
  - Windows, macOS, Linux, Android, iOS, Web
  - And soon Switch, Xbox series X/S, and Playstation 5 through W4 Games
Godot

- Scene + Node system for building games
  - Data oriented design “server” backend
- Scripting
  - GDScript, C#, Anything you can bind to C
- Rendering backends
  - Vulkan, OpenGL/WebGL, D3D12, (soon) Metal
Planet Crashers: Demo by W4 Games
Godot does a lot of stuff!

- The renderer must be flexible
- Users love to explore and learn by trial and error
  - Can’t assume they will do things the “proper” way
- Need to have good performance in most cases
  - Avoid performance cliffs
1. Writing an approachable rendering backend
2. Vulkan-first approach
Writing an approachable rendering backend
Rendering backend

- Performance vs usability
- Exposes higher level concepts
  - I.e. RenderingDevice (our RHI)

```rust
func _create_uniform_set(texture_rd : RID, shader : RID) -> RID:
    var uniform := RDUniform.new()
    uniform.uniform_type = RenderingDevice.UNIFORM_TYPE_IMAGE
    uniform.binding = 0
    uniform.add_id(texture_rd)
    return rd.uniform_set_create([uniform], shader, 0)
```
Rendering backend

● Do work for the user
● Previously:
  ○ Hand-placed barriers
  ○ Hand-ordered renderpass order
● Worked ok, but not optimal
  ○ Tons of tricky synchronization errors
● Want to expose more ways to customize rendering
  ○ Direct access to RenderingDevice
Directed Acyclic Render Graph

- Inspired by article from Pavlo Muratov
- Our approach makes it as “automagic” as possible
- Record “commands”
  - Update/copy/clear a texture/buffer/etc.
  - Submit a “render list” or “compute list”
- Commands always start and end from valid state
- Commands inferred from rendering code
  - I.e. no user facing graph building is required
Directed Acyclic Render Graph

- Commands track consumed resources and usage
  - Can automatically minimize barriers and resource transitions
- Use “immutable” resources to improve state tracking
  - Textures/Buffers that are never updated after being created
  - Reduces cost of tracking significantly
  - Only track <10% of resources
Directed Acyclic Render Graph

- Frame time decrease of about 10% across devices
- Huge decrease in barriers
  - 60% - 80% reduction
- Massive improvement in synchronization issues
- Users no longer have to worry about barriers or resource transitions
- Adds some CPU overhead
  - But savings from efficient API usage offset the cost
<table>
<thead>
<tr>
<th>EID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Capture Start</td>
</tr>
<tr>
<td>5</td>
<td>=&gt; vkQueueSubmit(2)[0]: vkBeginCommandBuffer</td>
</tr>
<tr>
<td>6</td>
<td>=&gt; vkQueueSubmit(2)[0]: vkEndCommandBuffer</td>
</tr>
<tr>
<td>7</td>
<td>=&gt; vkQueueSubmit(2)[1]: vkBeginCommandBuffer</td>
</tr>
<tr>
<td>8-9</td>
<td>Command graph (L-1)</td>
</tr>
<tr>
<td>11-14</td>
<td>Render Setup (L0) (Copy)</td>
</tr>
<tr>
<td>16-17</td>
<td>Setup Sky (L0) (Copy)</td>
</tr>
<tr>
<td>19-20</td>
<td>Bake Light Cluster (L0) (Copy)</td>
</tr>
<tr>
<td>22-24</td>
<td>Command graph (L0) (Copy)</td>
</tr>
<tr>
<td>26-39</td>
<td>Render Depth Pre-Pass (L1) (Draw)</td>
</tr>
<tr>
<td>41-42</td>
<td>Command graph (L2) (Copy)</td>
</tr>
<tr>
<td>44-47</td>
<td>Render Opaque Pass (L2) (Copy)</td>
</tr>
<tr>
<td>49-63</td>
<td>Render Opaque Pass (L3) (Draw)</td>
</tr>
<tr>
<td>65-76</td>
<td>Draw Sky (L4) (Draw)</td>
</tr>
<tr>
<td>78-81</td>
<td>Render 3D Transparent Pass (L4) (Copy)</td>
</tr>
<tr>
<td>83-88</td>
<td>Render 3D Transparent Pass (L5) (Draw)</td>
</tr>
<tr>
<td>90-102</td>
<td>Tonemap (L6) (Draw)</td>
</tr>
<tr>
<td>104-116</td>
<td>Command graph (L7) (Draw)</td>
</tr>
<tr>
<td>105</td>
<td>vkCmdBeginRenderPass(Clear)</td>
</tr>
<tr>
<td>112</td>
<td>vkCmdDrawIndexed(6, 1)</td>
</tr>
<tr>
<td>113</td>
<td>vkCmdEndRenderPass(Store)</td>
</tr>
<tr>
<td>114-116</td>
<td>(L-1)</td>
</tr>
<tr>
<td>115</td>
<td>=&gt; vkQueueSubmit(2)[1]: vkEndCommandBuffer</td>
</tr>
<tr>
<td>116</td>
<td>vkQueuePresentKHR (Swapchain Image 28)</td>
</tr>
</tbody>
</table>
Vulkan-first approach
Vulkan First Approach

• Big believers in platform independent APIs
• Would like a “write once run everywhere” solution
• In 2024, that’s still not possible
• Designed our abstraction around Vulkan
  ○ That’s a bit strong, but Vulkan was our first supported API and informed a lot of our decisions
  ○ Vulkan style barriers, pipelines, specialization constants, render passes
We love specialization constants
Use them to manage shader permutations
Especially for things like performance settings which change once in awhile but impact many shaders
/* Specialization Constants (Toggles) */

layout(constant_id = 0) const bool sc_use_forward_gi = false;
layout(constant_id = 1) const bool sc_use_light_projector = false;
layout(constant_id = 2) const bool sc_use_light_soft_shadows = false;
layout(constant_id = 3) const bool sc_use_directional_soft_shadows = false;

/* Specialization Constants (Values) */

layout(constant_id = 6) const uint sc_soft_shadow_samples = 4;
layout(constant_id = 7) const uint sc_penumbra_shadow_samples = 4;
layout(constant_id = 8) const uint sc_directional_soft_shadow_samples = 4;
layout(constant_id = 9) const uint sc_directional_penumbra_shadow_samples = 4;
layout(constant_id = 10) const bool sc_decal_use_mipmaps = true;
layout(constant_id = 11) const bool sc_projector_use_mipmaps = true;

// not used in clustered renderer but we share some code with the mobile renderer that requires this.
const float sc_luminance_multiplier = 1.0;
Specialization constants

- Poor support on some devices
  - Mostly low end mobile
  - Dead branch elimination appears to work, but loops based on SCs don’t flatten
- Not supported in D3D12
- What to do?
  - Emulate
Emulating specialization constants

- Patch SPIR-V directly during pipeline creation
  - Did I mention we love SPIRV?
- Very simple SPIR-V parser
- Small cost (~100 μs)
- Only enable on problematic drivers
Emulating specialization constants: D3D12

- D3D12 has no equivalent concept to specialization constants
  - Have to use constants or preprocessor definitions
  - Recompile the entire shader with each change
- SPIRV-Cross helps, but you still have to recompile the shader with each change
Emulating specialization constants: D3D12

- First attempt, patch DXIL directly
  - SPIRV-Cross to convert to HLSL
  - Patch HLSL by replacing uses of spec constants with loads of volatile int
  - Compile to DXIL (trusting volatile loads are not optimizable)
  - Parse LLVM bitcode from DXIL and write down offsets of constants
  - Per specialization:
    - Replace volatile loads with literal patch values from pipeline
    - Trust the driver to optimize the rest of the way
    - Re-sign the DXIL container
Emulating specialization constants: D3D12

- We ended up switching to Mesa’s NIR instead of SPIRV-Cross for unrelated reasons
  - NIR, out-of-the-box, creates multiple DXIL versions of the shader
    - too many permutations
  - We want to have a patchable DXIL blob
Emulating specialization constants: D3D12

- Second attempt (modified SPIR-V-to-NIR and NIR-to-DXIL code from Mesa):
  - SPIR-V-to-NIR:
    - Transform SC ops to regular ops dealing with literals (using placeholder value)
    - Use a custom `nir_intrinsic_load_constant_non_opt`
  - NIR-to-DXIL:
    - Let NIR optimize as usual
    - When emitting ones of those non-optimizable loads, use a standard DXIL load operation, but write down the pair [constant_id, value] for later patching
  - Per pipeline: patch in proper spec constant values and re-sign de DXIL container
Emulating specialization constants: D3D12

- Slightly faster than first attempt (especially initial transpilation)
- Works with a wider range of values
- Not as efficient as Vulkan, but good enough
Thank you!
Further Reading

- Organizing GPU Work with Directed Acyclic Graphs
  - https://levelup.gitconnected.com/organizing-gpu-work-with-directed-acyclic-graphs-f3fd5f2c2af3

- Render graph design doc:
  - https://gist.github.com/reduz/980b9b2547d57e6a915b2bb7e1e76e08#file-rendering_device_v2-h-L92

- Render Graph Implementation:
  - https://github.com/godotengine/godot/pull/84976
Further Reading

- DXIL patching:
  - https://twitter.com/RandomPedroJ/status/1532725156623286272

- Specialization constants via NIR: