Vulkanised 2024

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

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

iOS panko

Kivano Games

Bytten Studio

Chasing Carrots

2Dynamic Games

iOS Poke the Ant

Save Sloth Studios

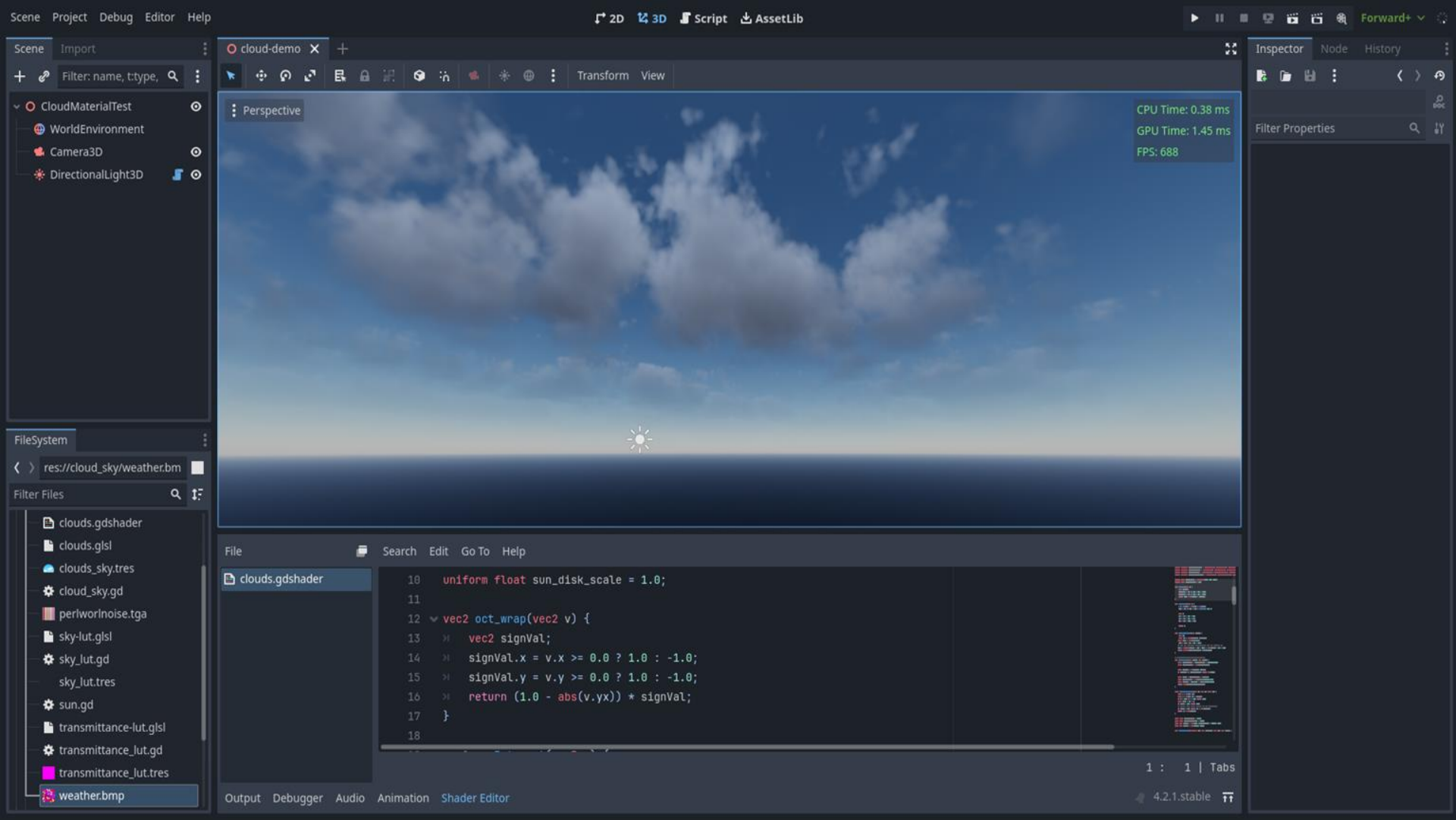
Mizizizizi

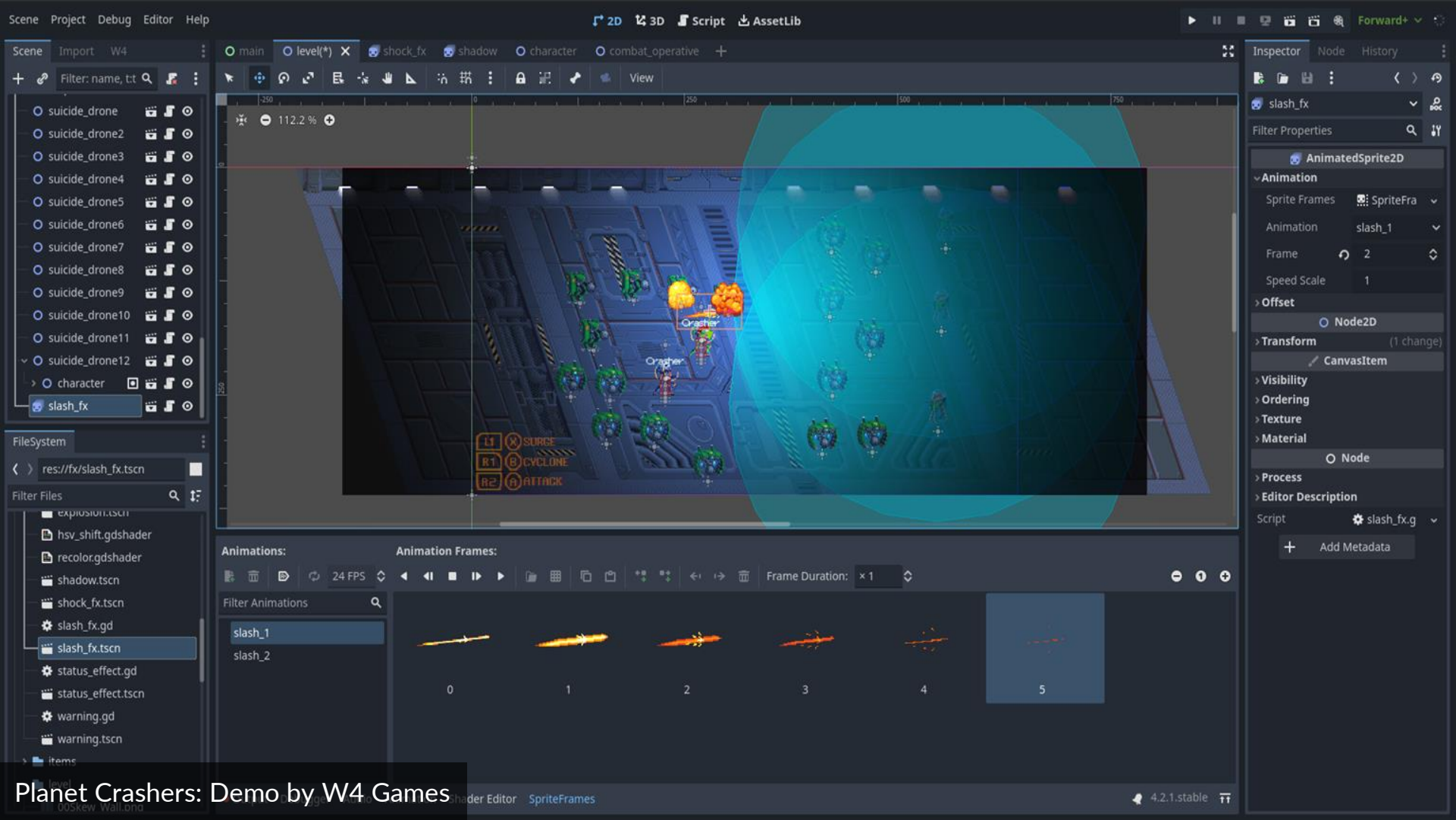
Bippinbits

Blobfish

Perfoon

Raffaele Picca


Scene Project Debug Editor Help
2D 3D Script AssetLib
Forward+
Scene Import
Filter: name, t:type,
CloudMaterialTest
WorldEnvironment
Camera3D
DirectionalLight3D
cloud-demo
Transform View
Perspective
CPU Time: 0.38 ms
GPU Time: 1.45 ms
FPS: 688
Inspector Node History
Filter Properties
FileSystem
res://cloud_sky/weather.bm
Filter Files
clouds.gdshader
clouds.glsl
clouds_sky.tres
cloud_sky.gd
perlworlnoise.tga
sky-lut.glsl
sky_lut.gd
sky_lut.tres
sun.gd
transmittance-lut.glsl
transmittance_lut.gd
transmittance_lut.tres
weather.bmp
File Search Edit Go To Help
clouds.gdshader
10 uniform float sun_disk_scale = 1.0;
11
12 vec2 oct_wrap(vec2 v) {
13 vec2 signVal;
14 signVal.x = v.x >= 0.0 ? 1.0 : -1.0;
15 signVal.y = v.y >= 0.0 ? 1.0 : -1.0;
16 return (1.0 - abs(v.yx)) * signVal;
17 }
18
1 : 1 | Tabs
Output Debugger Audio Animation Shader Editor
4.2.1.stable


Scene Project Debug Editor Help
2D 3D Script AssetLib
Forward+
Scene Import W4
suicide_drone
suicide_drone2
suicide_drone3
suicide_drone4
suicide_drone5
suicide_drone6
suicide_drone7
suicide_drone8
suicide_drone9
suicide_drone10
suicide_drone11
suicide_drone12
character
slash_fx
main level(*) shock_fx shadow character combat_operative
View
112.2 %
Crasher
Crasher
L1 X SURGE
R1 B CYCLONE
R2 A ATTACK
Inspector Node History
slash_fx
Filter Properties
AnimatedSprite2D
Animation
Sprite Frames SpriteFra
Animation slash_1
Frame 2
Speed Scale 1
Offset
Node2D
Transform (1 change)
CanvasItem
Visibility
Ordering
Texture
Material
Node
Process
Editor Description
Script slash_fx.g
Add Metadata
FileSystem
res://fx/slash_fx.tscn
Filter Files
hsv_shift.gdshader
recolor.gdshader
shadow.tscn
shock_fx.tscn
slash_fx.gd
slash_fx.tscn
status_effect.gd
status_effect.tscn
warning.gd
warning.tscn
items
Animations:
Animation Frames:
24 FPS
Frame Duration: × 1
Filter Animations
slash_1
slash_2
0
1
2
3
4
5
SpriteFrames
4.2.1.stable
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)

```
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

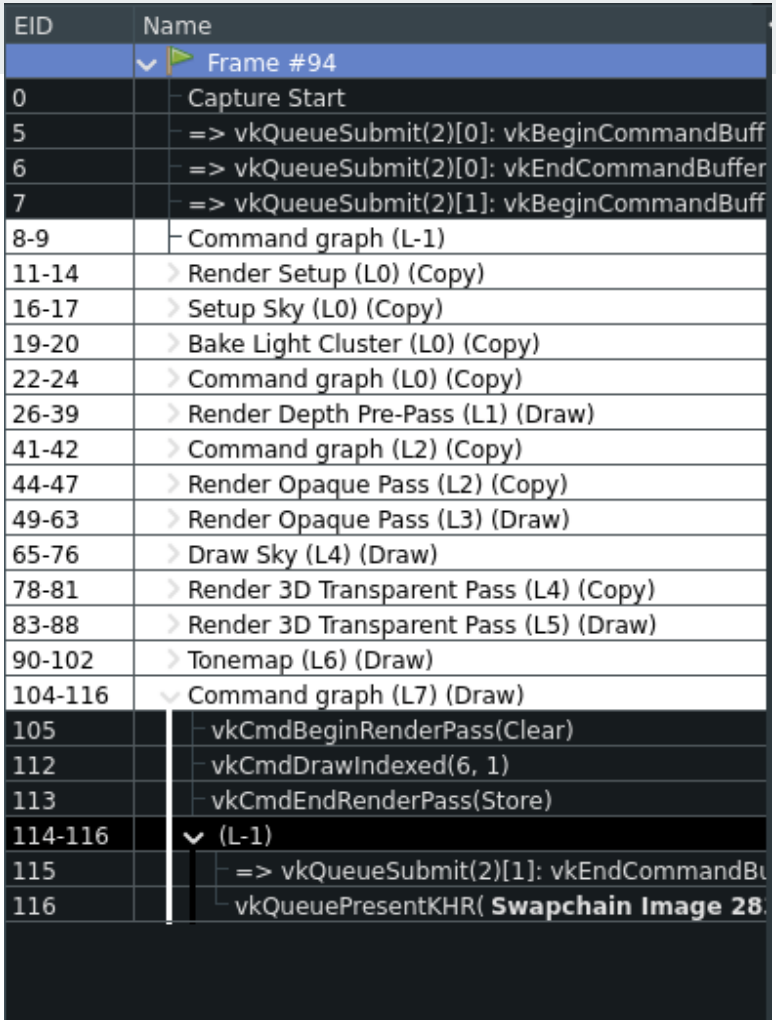
- Inspired by article from Pavlo Muratov
- Our approach makes it as “automagic” as possible
- Record “commands”
  - Update/copy/clear a texture/buffer/etc.
  - Submit a “render list” or “compute list”
- Commands always start and end from valid state
- Commands inferred from rendering code
  - I.e. no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


EID
Name
Frame #94
0
Capture Start
5
=> vkQueueSubmit(2)[0]: vkBeginCommandBuff
6
=> vkQueueSubmit(2)[0]: vkEndCommandBuffer
7
=> vkQueueSubmit(2)[1]: vkBeginCommandBuff
8-9
Command graph (L-1)
11-14
Render Setup (L0) (Copy)
16-17
Setup Sky (L0) (Copy)
19-20
Bake Light Cluster (L0) (Copy)
22-24
Command graph (L0) (Copy)
26-39
Render Depth Pre-Pass (L1) (Draw)
41-42
Command graph (L2) (Copy)
44-47
Render Opaque Pass (L2) (Copy)
49-63
Render Opaque Pass (L3) (Draw)
65-76
Draw Sky (L4) (Draw)
78-81
Render 3D Transparent Pass (L4) (Copy)
83-88
Render 3D Transparent Pass (L5) (Draw)
90-102
Tonemap (L6) (Draw)
104-116
Command graph (L7) (Draw)
105
vkCmdBeginRenderPass(Clear)
112
vkCmdDrawIndexed(6, 1)
113
vkCmdEndRenderPass(Store)
114-116
(L-1)
115
=> vkQueueSubmit(2)[1]: vkEndCommandBu
116
vkQueuePresentKHR(Swapchain Image 28

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

Specialization constants

- 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

Legend of the Nuku Warriors: Demo by W4 Games

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:
  - https://godotengine.org/article/d3d12-adventures-in-shaderland/