Understanding D3D12_ROOT_SIGNATURE_DESC: A Vulkan Developer's Perspective

Version 1.0

Phani Srikar

29-12-2024

Hieee all, Phani here. I'm in Himalayas for the NYE'25, a fun adventure for this "techbro" to **push myself out of the comfort zone** and start off the next journey. Ik nothing will change in a day but NYE gives you a reason to start something, take a break and reflect and pursue further, at the end of the day it's you who gotta make the choice. Recover from the burnout and get back at it soon! But I'm loving it here, the cold and the serene life in the hills is making me realize the capitalistic gains we chase over all the time instead of knowledge. I also went to this amazing and soothing cafe "Aaarcana" in Tapovan, Rishkesh. Will write more about this trip once I'm back in my next post. A lot of "We're cooked!" moments have been happening lol. Let's get into it for now shall we?

Introduction

As a long standing Vulkan developer myself who's exploring DirectX12 to port my Razix Engine, understanding `D3D12_ROOT_SIGNATURE_DESC` can feel analogous to Vulkan's `VkPipelineLayout` and `VkDescriptorSetLayout`. Both define how resources such as constant buffers, textures, and samplers are bound to shaders and how to manage descriptor Heaps via Tables/Sets. In this article I will try explain and understand the usage of `D3D12_ROOT_SIGNATURE_DESC` from a Vulkan developer's perspective, focusing on key similarities and differences, while walking through questions I ask myself while transitioning between the APIs.

`D3D12_ROOT_SIGNATURE_DESC` - The Basics

The `D3D12_ROOT_SIGNATURE_DESC` structure describes the root signature layout in DirectX12, it contains all the range of resources that will be bound per shader such as descriptor table (contains descriptors info per register space), static samplers and root constants, inline descriptors (any of the CBV, SRV or UAVs). This is the structure that defines it:

```
struct D3D12_ROOT_SIGNATURE_DESC {
    UINT NumParameters;
    const D3D12_ROOT_PARAMETER* pParameters;
    UINT NumStaticSamplers;
    const D3D12_STATIC_SAMPLER_DESC* pStaticSamplers;
    D3D12_ROOT_SIGNATURE_FLAGS Flags;
};
```

- `NumParameters` **and** `pParameters`**:** Define the number and details of root parameters, which include descriptor tables, 32-bit root constants, and inline descriptors
- `NumStaticSamplers` **and** `pStaticSamplers`**:** Specify static samplers that are immutable during the lifetime of the root signature
- `Flags`**:** Provide additional configuration options, such as enabling or disabling specific shader stages for visibility of root signature and resources

D3D12 vs Vulkan

Is `D3D12_ROOT_SIGNATURE_DESC` a Vulkan `VkPipelineLayout` in disguise?

Yes, `D3D12_ROOT_SIGNATURE_DESC` is similar to Vulkan's `VkPipelineLayout`. Both describe how resources are bound to shaders and how the GPU accesses them via tables. The key difference lies in how resources are setup in the heaps and accessed, with D3D12 offering root parameters and heap types like `CBV/SRV/UAV` and more memory control focued where as Vulkan relying heavily on descriptor sets and pipeline layouts with set types allowing mix and match in a granular layout focused setup.

What About Descriptor Heaps and Tables?

Descriptor tables in DirectX 12 are conceptually similar to Vulkan's descriptor sets. They group resources like **constant buffers (CBVs), shader resource views (SRVs), and unordered access views (UAVs are basically read/write enabled resources)**. You use
`D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE` to define them. Descriptor Heaps in DirectX 12 provide a contiguous memory region where descriptors are stored and can be accessed directly by shaders.

DirectX 12 defines these types of Descriptor Heaps, each suited for specific resource types:

- **CBV/SRV/UAV Heap**: Used for Constant Buffer Views (CBV), Shader Resource Views (SRV), and Unordered Access Views (UAV).
- **Sampler Heap**: Dedicated to sampler descriptors.
- **RTV/DSV Heap**: Used for Render Target Views (RTV) and Depth Stencil Views (DSV). These heaps are not shader-visible and are primarily managed on the CPU for creating depth and swapchain image targets.

Vulkan's `VkDescriptorPool`, on the other hand, acts as a more granular allocation system for `VkDescriptorSet` objects, which group descriptors and are bound to shaders at the set level. Vulkan takes a layout-focused approach, organizing descriptors into `VkDescriptorSet` objects with clearly defined layouts tied to the pipeline. DirectX 12, on the other hand, is more memory-focused, using Descriptor Heaps as a contiguous storage structures for shaders to access

Are Root Constants Equivalent to Vulkan Push Constants?

Yes in a way, you can define root constants in D3D12 using the param type (`D3D12_ROOT_PARAMETER_TYPE_32BIT_CONSTANTS`) while passing to `pParameters` and they are equivalent to Vulkan's push constants. Both allow small amounts of frequently updated data (like transformation matrices or bindless resource indices etc.) to be passed directly to shaders without using a buffer. For example, in Vulkan, you might define push constants for per-draw updates, and in D3D12, you'd use root constants for the same purpose. They are 32-bit `DWORDS` and are usually maxed out at 128/256-bytes or so, check with your GPU vendor for the device limits.

Note:- while using shader reflecting unlike Vulkan, DirectX12 can't know if they are root constants, it's defined as a `cbuffer` in HLSL. So use a constant name like `struct PushConstant` while reflecting to check if the shader is using a root constant. if so, pass that data via the root signature.

What Are Inline Descriptors, and are they like Vulkan Push Descriptors?

Inline descriptors (`D3D12_ROOT_PARAMETER_TYPE_CBV/SRV/UAV`) cabe be directly bound into the root signature without using descriptor tables. These are similar to Vulkan's `VK_KHR_push_descriptor` extension features, which allows you to bind resources dynamically and directly without pre-allocated descriptor sets. They are sometimes useful than push constants are they don't have restrictions on data size but have much higher overhead if bound as frequently as a root constant/push constant so thread with caution when using.

Typical Usage: Push Constants and Descriptor Tables

This what you'll be doing in most scenarios:

- **Descriptor Tables:** Use `D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE` for resource management. This is analogous to Vulkan's descriptor sets, select a heap type `(CBV_UAV_SRV/Sampler/DTV/RTV)` and define the resource structure for all heaps using descriptors.
- **Root/Push Constants:** Use `D3D12_ROOT_PARAMETER_TYPE_32BIT_CONSTANTS` for lightweight and frequent updates, just like Vulkan push constants. These are ideal for small data that changes per draw or dispatch. They are defined as cbuffer (I suggest don't specify and register binding info and leave it alone or not) in HLSL but you can pass/update it's data when binding the root signature.

```
// Define a descriptor table with SRVs and CBVs
D3D12_ROOT_DESCRIPTOR_TABLE descriptorTable = {};
descriptorTable.NumDescriptorRanges = 1;
descriptorTable.pDescriptorRanges = &descriptorRange;

// Define root constants
D3D12_ROOT_CONSTANTS rootConstants = {};
rootConstants.ShaderRegister = 0;
rootConstants.RegisterSpace = 0;
rootConstants.Num32BitValues = 4;

// Define root parameters
D3D12_ROOT_PARAMETER rootParameters[2] = {};
rootParameters[0].ParameterType = D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE;
rootParameters[0].DescriptorTable = descriptorTable;
rootParameters[0].ShaderVisibility = D3D12_SHADER_VISIBILITY_ALL;

rootParameters[1].ParameterType = D3D12_ROOT_PARAMETER_TYPE_32BIT_CONSTANTS;
rootParameters[1].Constants = rootConstants;
rootParameters[1].ShaderVisibility = D3D12_SHADER_VISIBILITY_VERTEX;

// Create the root signature
D3D12_ROOT_SIGNATURE_DESC rootSignatureDesc = {};
rootSignatureDesc.NumParameters = _countof(rootParameters);
rootSignatureDesc.pParameters = rootParameters;
rootSignatureDesc.Flags = D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT;
```

example of how a root signature is filled with a descriptor set and root constants

Misc stuff I ask myself

Why Not Use Inline Descriptors All the Time?

Inline descriptors are great for quick and direct resource binding when your data exceeds the size of a root constant, but they lack flexibility and can quickly cause binding overhead if bound and updated frequently.

How Are Static Samplers Used?

Static samplers are immutable samplers defined in the root signature. These are analogous to Vulkan's immutable samplers in descriptor set layouts. For example, if you have a fixed sampler configuration for certain textures, you can define them here.

How Do Flags Work?

The `Flags` field in `D3D12_ROOT_SIGNATURE_DESC` allows you to optimize or restrict certain stages. For example, you can specify `D3D12_ROOT_SIGNATURE_FLAG_DENY_PIXEL_SHADER_ROOT_ACCESS` to block resource access from the pixel shader stage, improving performance by helping driver optimize early.

How it's done in Razix Engine

When using HLSL with D3D12, I typically rely on root constants (`D3D12_ROOT_PARAMETER_TYPE_32BIT_CONSTANTS`) to emulate Vulkan's push constants. This way we can have lightweight and direct access to frequently updated data. And I use descriptor tables to handle resource binding in the usual way. Almost no inline descriptors at all and almost never update descriptor sets.

Here is an example:

```
#ifdef __GLSL__    // GLSL - shading language for Vulkan (SPIRV) & OpenGL.

    #define PUSH_CONSTANT(T)       [[vk::push_constant]] T pcData
    #define GET_PUSH_CONSTANT(mem) pcData.mem

#elif defined __HLSL__    // HLSL - DirectX backend shading language.

    #define PUSH_CONSTANT(T) \
        cbuffer T##Buffer    \
        {                    \
            T pcData;        \
        };
    #define GET_PUSH_CONSTANT(mem) pcData.mem

#endif

\begin{verbatim}
// Push constant structure
struct PushConstantData {
    float4 color;
    float4x4 transform;
};

PUSH_CONSTANT(PushConstantData);

// Descriptor tables for resource binding
Texture2D myTexture : register(t0, space0);
SamplerState mySampler : register(s0, space0);

// Shader code
float4 main(float4 position : POSITION) : SV_POSITION {
    float4 worldPosition = mul(GET_PUSH_CONSTANT(transform), position);
    return worldPosition * GET_PUSH_CONSTANT(color);
}
```

So while reflection I create descriptor bindings for all resources and when I get the type PushConstant, (all shaders use the same named struct) and skip it and let the users bind the resources based on the resource view hints given and assign the corresponding heap `(CBV_SRV_UAV/Sampler/DTV/RTV)`.

Closing Notes

By thinking of descriptor tables as descriptor sets for resource management and root constants (or `cbuffers` for emulation) as push constants for small, frequent updates, Vulkan devs can adapt to D3D12 intuitively. Inline descriptors can be thought of as a more direct but less flexible alternative using the vulkan push descriptor extension. Also, **Descriptor Heap is the memory backing structure for Descriptor Table resources**. They are in a sense analogoues to vulkan's VkDescriptorPool to allocate sets from but with more memory level control than vulkan.

That's it for today, until next time. Byee!