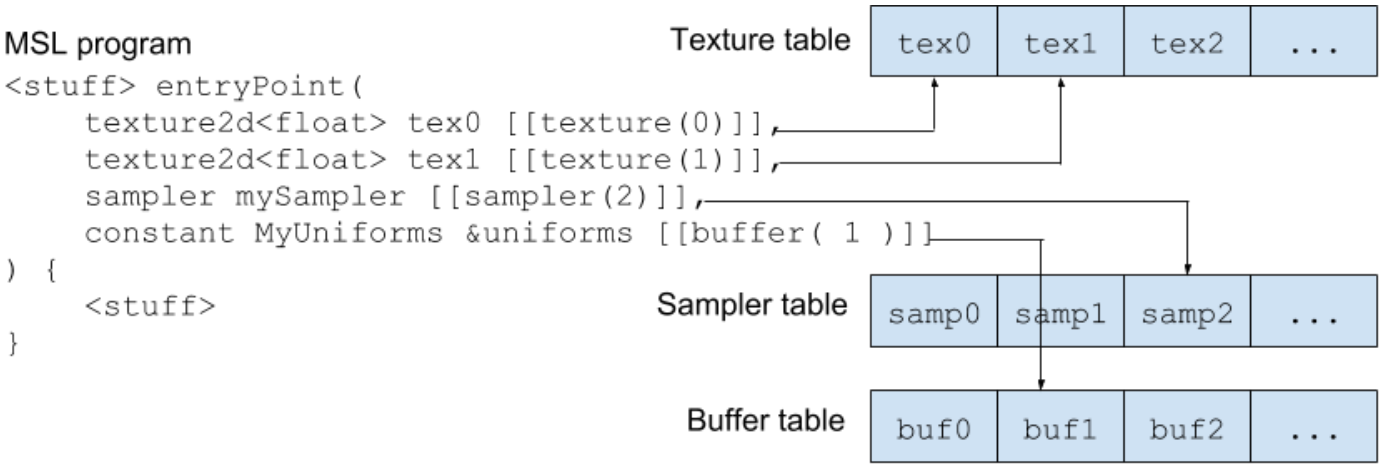
This is the detailed investigation I promised to put on Github last meeting. It explains at a high-level the binding models of all 3 explicit APIs, and exposes why we believe Vulkan's binding model should be used as a starting point.

See also these slides which contain more info about bindless vs. fixed-function and the Vulkan binding model, and our initial investigation for NXT (with some outdated code samples).

The binding models of explicit APIs

Metal's binding model

Metal's binding model is basically the same as OpenGL's and D3D11: there are per-stage global arrays of textures, buffers and samplers. Binding of shader variables to resources is done by tagging arguments to the main function with an index in the tables.



There are functions on the encoders (aka command buffers) to update ranges of the tables, with one update function for each stage, for each table type (and sometimes more than one, mostly for convenience). The update to the tables is immediate and the driver handles any synchronization that might be needed transparently. For example:

- MTLRenderCommandEncoder::setVertexBuffers(_:offsets:with:)
- MTLRenderCommandEncoder::setFragmentSamplerStates(_:with:)
- MTLComputeCommandEncoder::setTextures(_:with:)

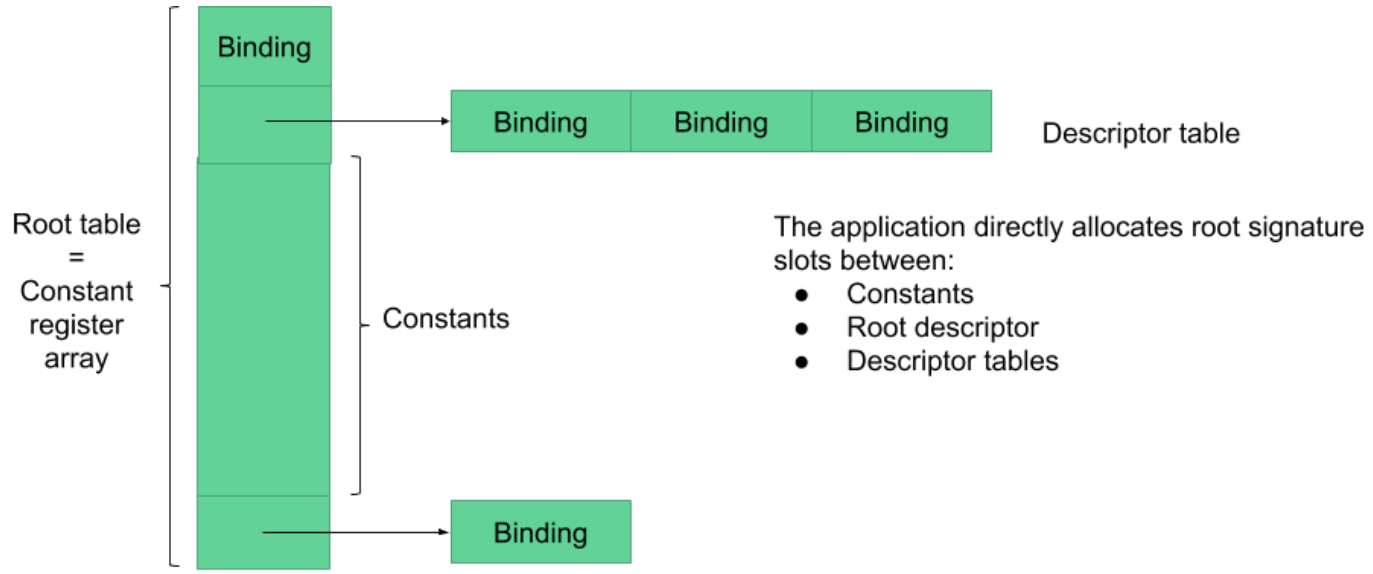
D3D12's binding model

D3D12's binding model is more geared towards "bindless" GPUs in which resources aren't set in registers in fixed-function hardware and referenced by their index, but instead described by a descriptor living in GPU memory and referenced by their descriptor's GPU virtual address.

Things look as if the shaders had only access to one global UBO by default, called "root signature" that is an array of elements that can be one of three things:

- A small constant
- An inlined descriptor for a resource
- A pointer to a range of a "descriptor heap"

A shamelessly re-used slide.



Individual elements root signature can be updated directly in command lists with updates appearing immediately to subsequent draws / dispatches. For example:

- ID3D12GraphicsCommandList::SetGraphicsRoot32BitConstant changes a constant at a specific index in the root signature
- ID3D12GraphicsCommandList::SetGraphicsRootShaderResourceView puts a texture / buffer descriptor at a specific index in the root signature
- ID3D12GraphicsCommandList::SetComputeRootDescriptorTable changes the pointer to a descriptor heap at a specific index in the root signature

Using a root signature in shaders is done like below. The root signature layout is described and gives each “binding” a register index, then these register indices are bound to variable names and finally, the “main” function is tagged with the root signature layout. The extra level of indirection using register indices seems to be to help with porting from D3D11 shaders.

```
// A example from the D3D12 docs, trimmed down, that defines a root signature with in order
//  - A constant buffer binding “b0”
//  - A pointer to a descriptor heap range containing…
//    - A constant buffer “b1”
//    - An array of 8 textures/buffers starting at “t1”
// - A constant named “b3”
#define MyRS1 \
              "CBV(b0), " \
              "DescriptorTable( CBV(b1), " \
                              "SRV(t1, numDescriptors = 8)), " \
              "RootConstants(num32BitConstants=3, b10)"


// Binding of part of the signature to variables names
cbuffer cbCS : register(b0) {
    uint4 g_param;
    float4 g_paramf;
};

// Using the root signature for this entry point (MyRS1 would be actually defined in a header).
[RootSignature(MyRS1)]
float4 main(float4 coord : COORD) : SV_Target
{
…
}
```

On the API side, a corresponding ID3D12RootSignature object must be created that represents the layout of the root signature. This is needed because the actual layout on GPU might not match what was declared, to allow the driver to optimize things or do some emulation on hardware that isn't bindless enough.

When compiling an ID3D12PipelineState, the root signature must be provided so that the compiled shader code can be specialized for the actual layout of the root signature on the GPU. Then in ID3D12GraphicsCommandList::SetGraphicsRootSignature must(?) be called before any update to the root table or call to ID3D12GraphicsCommandList::SetPipelineState, so that the command list too knows what the actual layout on the GPU is.

Having the same root signature used to compile multiple pipeline makes sure the driver doesn't have to shuffle data around when pipelines are changed, since they are guaranteed to access descriptors by following the same layout.

More info on root signatures can be found in the MSDN docs.

One thing we didn't mention yet is how to populate a descriptor heap with data. MSDN docs aren't super-clear but my understanding is that there are two types of descriptor heaps:

- Non-shader visible descriptor heaps that can be used for staging before copying to shader-visible heaps, or for descriptors used to set render-targets, since it is still not bindless on most HW.
- Shader-visible heaps, which can be written but not read by the CPU(?) and are essentially opaque heaps that can be on the GPU (for bindless use) and/or on CPU (for emulation on fixed-function HW).

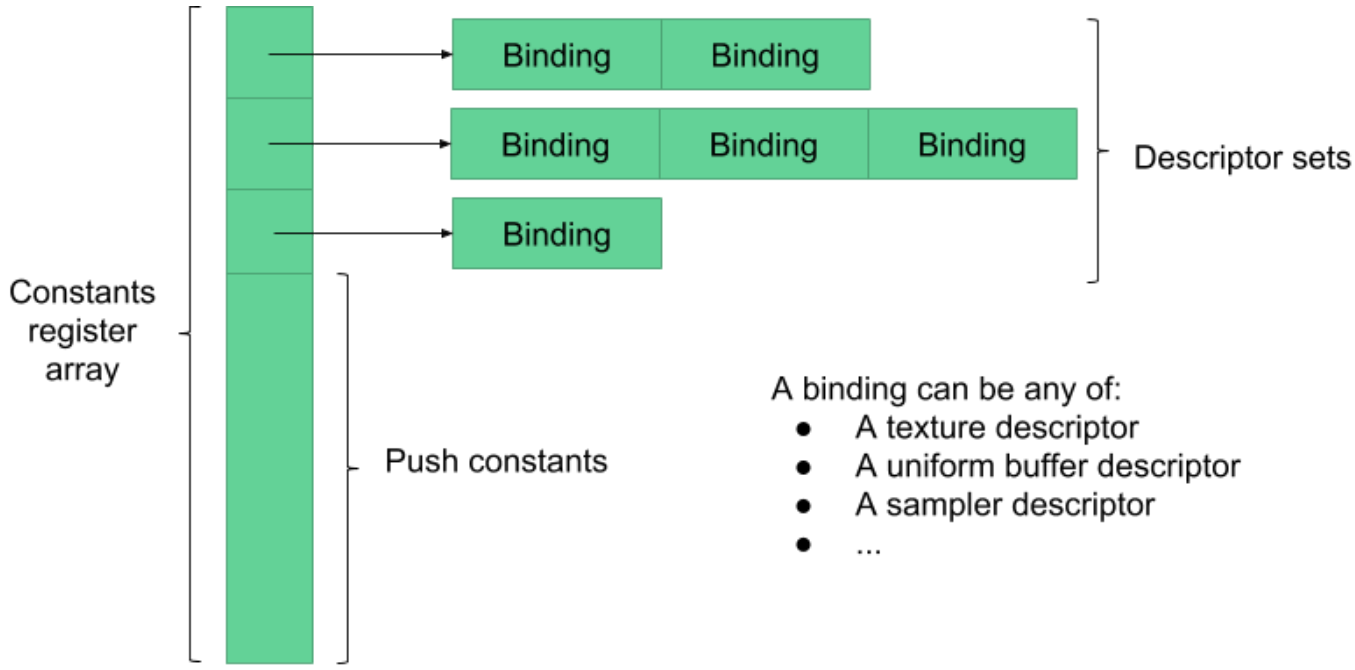
ID3D12Device::CopyDescriptors can be used or descriptors created directly in the heaps. Then to be able to use a heap with, for example SetGraphicsRootDescriptorTable, the heap must be bound to the current command list with ID3D12GraphicsCommandList::SetDescriptorHeaps. Limitations are that only one sampler heap and one SRV/UAV/CBV descriptor heap can be bound at a time. Also heaps cannot be arbitrary large on some hardware, with samplers being at max 2048 descriptors and SRV/UAV/CBV heaps a million descriptors. Switching heaps can cause a wait-for-idle on some hardware.

One restriction worth pointing out is that descriptor heaps cannot be created to contain both samplers and other descriptors at the same time.

Vulkan's binding model

Vulkan's binding model is essentially a subset of D3D12's and more opaque to the application. This is because Vulkan needs to run on mobile GPU that are more fixed-function than what D3D12 targets.

Another shameless slide reuse:



This is similar to D3D12's root signature except that:

- There are no descriptors directly in a root table.
- There is a fixed number (at least 4) of root descriptor tables.
- There is a fixed number of root constants.
- Also in Vulkan these two concepts are not presented together, instead each descriptor table is called a descriptor set and root constants are called push constants.

Binding shader variables to specific locations in a descriptor set is done like shown below (also shows push constants) (from this slide):

```
// Example in GLSL because SPIRV would be much more verbose
layout(set = 1, binding = 0) uniform texture2D albedo;

layout(push_constant) uniform Block {
    int member1;
    float member2;
    ...
} pushConstants;

float foo = texture(t, texcoord).r + pushConstants.member1;

// Could get compiled to:

Descriptor* set1 = RootConstantRegister[1];
TextureDescriptor* albedo = set1[0];

Int pushConstants_member1 =
    RootConstantRegister[PUSH_CONSTANT_START + 0];
float foo = Sample2D(albedo, texcoord).r
    + pushConstants_member1;
```

On the API side, like in D3D12, a layout object needs to be created and used to during pipeline creation (vkCreateGraphicsPipelines). This object is VkPipelineLayout (vkCreatePipelineLayout) and is made of multiple VkDescriptorSetLayouts (vkCreateDescriptorSetLayout). Then descriptor sets are used in a command buffer (without need for synchronization) via vkCmdBindDescriptorSets and push constants are set with vkCmdPushConstants.

Descriptors are created from a VkDescriptorPool that is similar to D3D12 descriptor heaps, except that the VkDescriptorSets returned are opaque and can only be written to or copied via specialized API functions (not general GPU copy like D3D12(?)).

More info about this in the specification's section on descriptor sets.

Metal 2

Metal 2 adds the Indirect Argument Buffer concept, which are opaque-layout buffers containing constants and descriptors that can be bound all at once. It is essentially a descriptor set.

In addition to allocating, populating and using IABs, applications must specify on the encoders which resources or resource heaps need to be resident for the draw. Also it looks like Metal allocates all descriptor in a single descriptor heap, and has the same type of limitation as D3D12, but for the whole app. (500.000 descriptors max, 2048 samplers max).

Why use Vulkan's binding model

During our NXT investigation we found that the Vulkan binding model would be the best one to use, as converting from Vulkan to D3D12 or Metal doesn't add much overhead while converting to D3D12 or Metal to Vulkan would be expensive. (and we are not smart enough to design a whole new binding model) Also for reasons, NXT uses the term "bind groups" instead of "descriptor sets".

Vulkan to Metal

When the pipeline layout is compiled we can do “register allocation” of the descriptors in their respective tables. Then the shaders can be changed to refer to the position in the table instead of the (set, location) as in Vulkan. Finally when a descriptor set is bound in the command buffers, we just need to call the necessary MTLGraphicsCommandEncoder::set<Textures/Buffers/Samplers> at most once each. The arguments can have been precomputed at descriptor set creation).

Vulkan to D3D12

Essentially we would only use special case root signatures that look like the Vulkan binding model presented above. Then all operations on descriptors sets and push constants decay naturally to their D3D12 counterparts.

One thing is that it looks like D3D12 can only use a limited number of descriptor heaps in a command list while (AFAIK) Vulkan doesn’t have this limitation. In NXT we plan to solve this mismatch by not exposing descriptor pools and doing our own descriptor set bookkeeping (even in the Vulkan backend): with some sort of compacting GC for descriptor sets we should be able to keep the number of heaps small.

Another issue is that D3D12 has separate samplers heap, which can solved by having potentially two root descriptor tables per descriptor set that has samplers. The hope is that in most applications samplers aren’t present in many places in a pipeline layout.

Metal to Vulkan

In Metal, each a pipeline essentially has its own pipeline layout / root signature layout, so in Vulkan new descriptor sets would have to be created at each pipeline changes. Things could work a bit better on D3D12, as the D3D11 on D3D12 layer shows (Metal and D3D11’s binding models are the same). However this would prevent a large class of optimizations applications can do by reusing parts of the root table or descriptor sets.

D3D12 to Vulkan

There are many small mismatches because D3D12 is more flexible and explicit than Vulkan:

- Translating from D3D12 to Vulkan would require runtime creation of descriptor sets when a root descriptor is set.
- Vulkan requires the number of each type of descriptor for descriptor pool creation, whereas D3D12 requires the total number of descriptor for descriptor heap creation.
- Descriptors aren’t placed explicitly in Vulkan, which would make supporting copy operations a challenge.
- Root descriptor tables can alias in D3D12, supporting this in Vulkan would require a ton of state tracking.
- ...

Inheritance

Another advantage of using Vulkan is that it has very good inheritance semantics for descriptor sets: if you set a pipeline with a different pipeline layout, but with descriptor set layouts matching up to position N, then you don’t need to bind these descriptor sets again. This could translate to not having to reset the whole tables in Metal, and only changing a couple root table descriptors in D3D12.