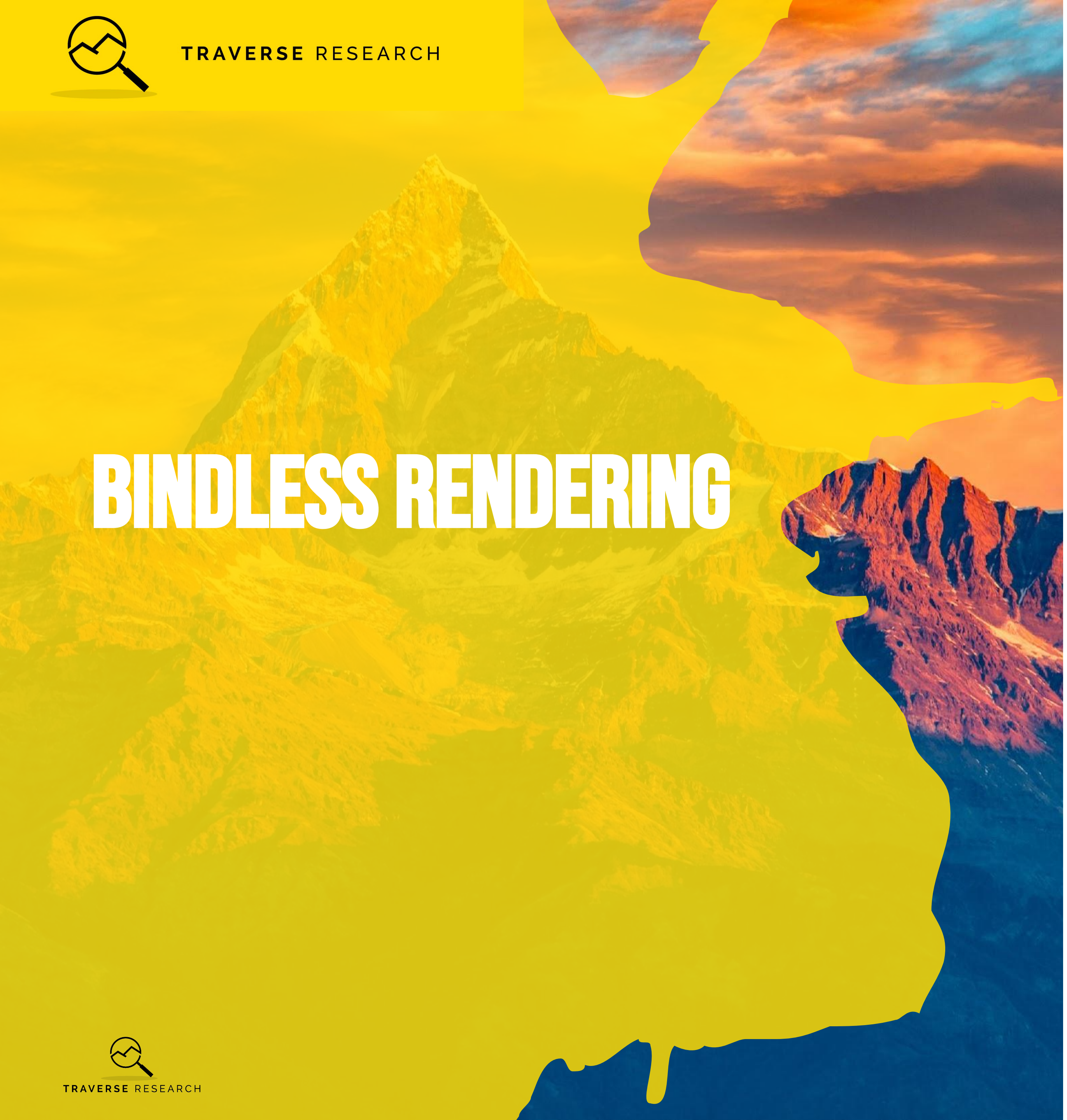
TRAVERSE RESEARCH

# Bindless Rendering in D3D12

NEXT SLIDE

Darius Bouma - Rendering Engineer at Traverse Research

TRAVERSE RESEARCH



# BINDLESS RENDERING

## OVERVIEW

01 Shader model pre-6.6 & 6.6 bindless.

02 Rendering resource handles

03 Descriptor management.

04 Bindless shader code.

05 Bonus

TRAVERSE RESEARCH

Bindless Rendering

# MOTIVATION

01 Significantly reduce API overhead.

02 Reduce rendering pipeline complexity.

03 Descriptor staging is a thing of the past.

04 Makes DXR 1.0 / 1.1 a lot easier to work with.

05 More lightweight rendering backend.

# TRADITIONAL "BINDFULL" PROBLEMS

## ROOT SIGNATURES

Each pipeline has a unique root signature.

Sometimes uses shader reflection.

A lot of management code and/or factories to make this all work.

## DESCRIPTOR STAGING

Staging descriptors from a big heap on the cpu to smaller gpu heaps.

A lot of management code.

Difficult to track.

## EXCESSIVE API CALLS

`CopyDescriptors` call per unique draw/dispatch.

`SetCompute/ SetGraphicsRootDescriptor` per unique draw/dispatch

The point of d3d12 is to have lower CPU overhead, let's make it count!

TRAVERSE RESEARCH

# BINDLESS RENDERING

## Setting up bindless heaps

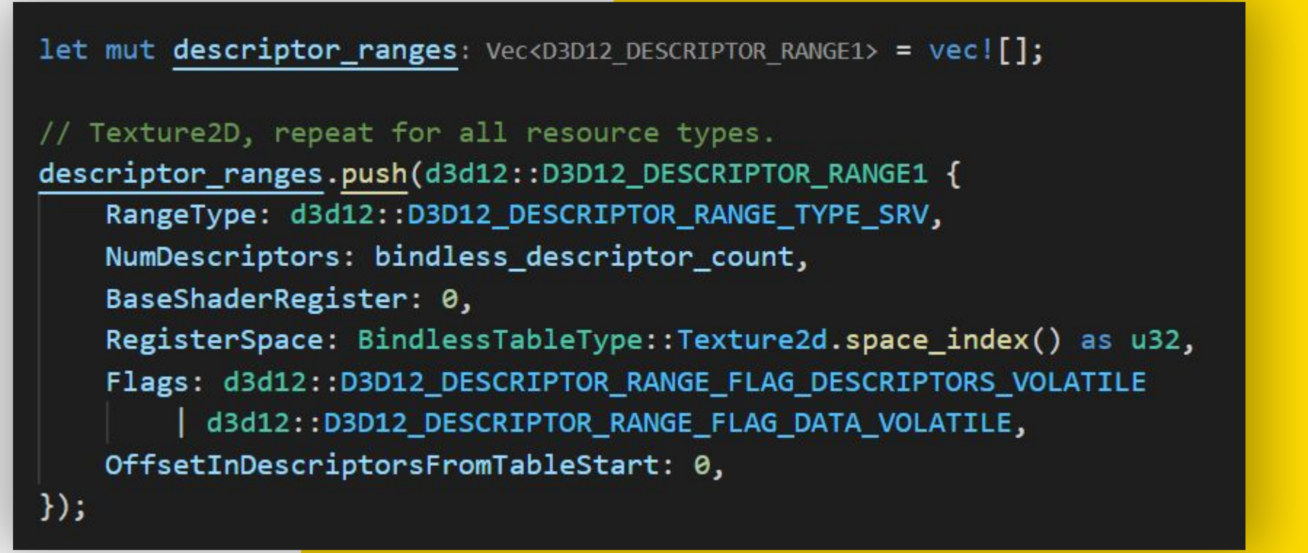
- A single shader visible descriptor heap for all CBV_SRV_UAV descriptors and optionally SAMPLER.
- Non-shader visible heaps still required for RTV and DSV.
- CBV_SRV_UAV descriptor heap can go up to 1,000,000+ descriptors
- SAMPLER heap is limited (2048) on most hardware.
- Make sure to check out hardware tiers. We assume hardware tier 3 for our framework “Breda”.

# BINDLESS RENDERING

## A “global” root signature.

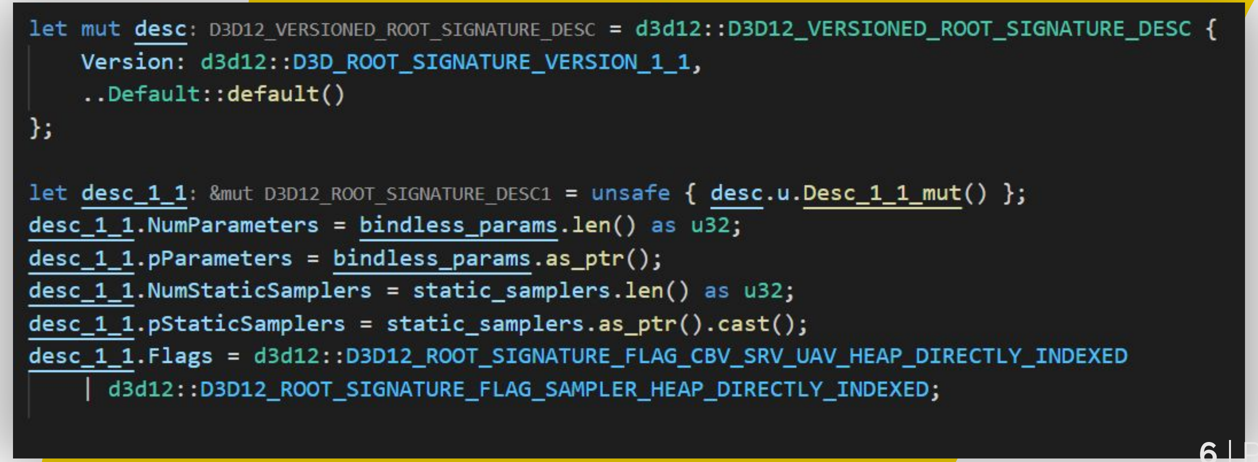
- A single root signature for all shaders.
- Assign register space per resource type.
- Set up root constants.
- Set up static samplers.

## SHADER MODEL 6.5 OR LOWER



```
let mut descriptor_ranges: Vec<D3D12_DESCRIPTOR_RANGE1> = vec![];

// Texture2D, repeat for all resource types.
descriptor_ranges.push(d3d12::D3D12_DESCRIPTOR_RANGE1 {
    RangeType: d3d12::D3D12_DESCRIPTOR_RANGE_TYPE_SRV,
    NumDescriptors: bindless_descriptor_count,
    BaseShaderRegister: 0,
    RegisterSpace: BindlessTableType::Texture2d.space_index() as u32,
    Flags: d3d12::D3D12_DESCRIPTOR_RANGE_FLAG_DESCRIPTORS_VOLATILE
        | d3d12::D3D12_DESCRIPTOR_RANGE_FLAG_DATA_VOLATILE,
    OffsetInDescriptorsFromTableStart: 0,
});
```

## SHADER MODEL 6.6+



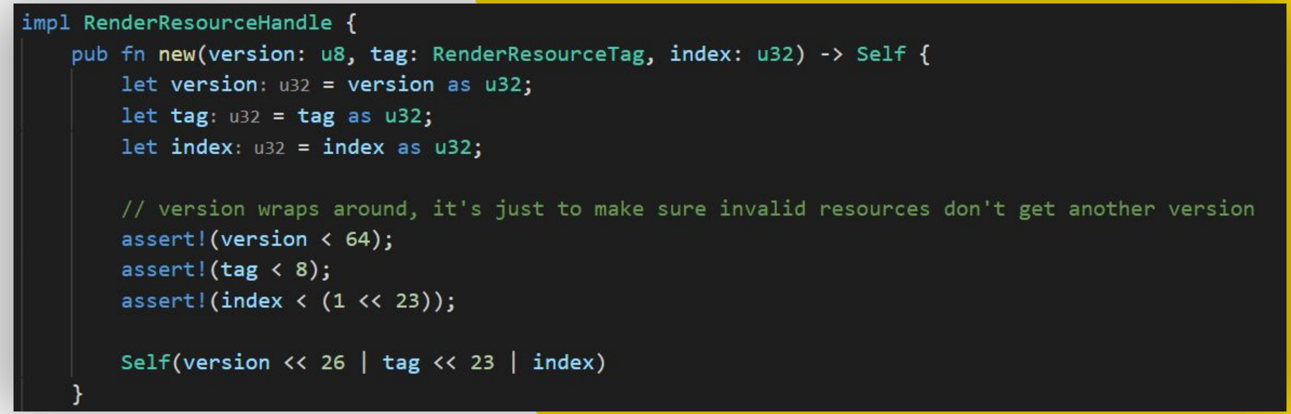
```
let mut desc: D3D12_VERSIONED_ROOT_SIGNATURE_DESC = d3d12::D3D12_VERSIONED_ROOT_SIGNATURE_DESC {
    Version: d3d12::D3D_ROOT_SIGNATURE_VERSION_1_1,
    ..Default::default()
};

let desc_1_1: &mut D3D12_ROOT_SIGNATURE_DESC1 = unsafe { desc.u.Desc_1_1_mut() };
desc_1_1.NumParameters = bindless_params.len() as u32;
desc_1_1.pParameters = bindless_params.as_ptr();
desc_1_1.NumStaticSamplers = static_samplers.len() as u32;
desc_1_1.pStaticSamplers = static_samplers.as_ptr().cast();
desc_1_1.Flags = d3d12::D3D12_ROOT_SIGNATURE_FLAG_CBV_SRV_UAV_HEAP_DIRECTLY_INDEXED
    | d3d12::D3D12_ROOT_SIGNATURE_FLAG_SAMPLER_HEAP_DIRECTLY_INDEXED;
```

# BINDLESS RENDERING

## Rendering resource handles

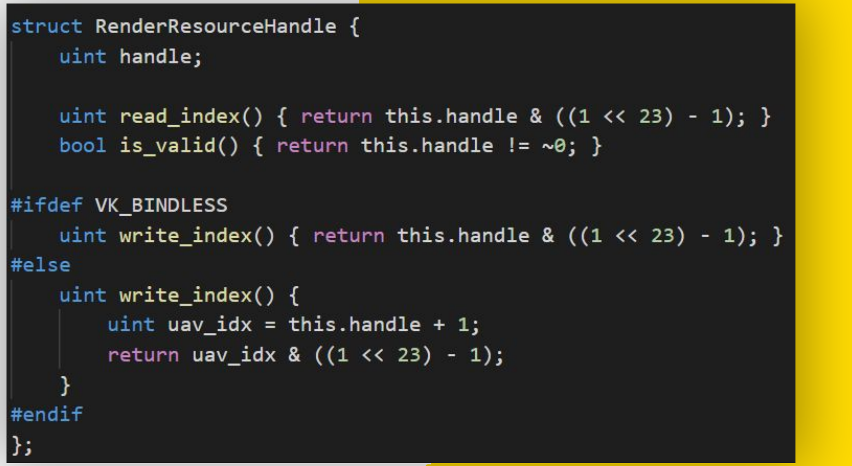
- A *RenderResourceHandle* maps 1:1 to an index in our bindless descriptor heap.
- *RenderResourceHandle* is simply a uint32 but we are not using all 32 bits, unused bits can be used for validation purposes.
- *RenderResourceHandles* are exclusively created during resource creation.
- What about SRV & UAV as they are separate descriptors?

### CPU IMPLEMENTATION



```rust
impl RenderResourceHandle {
    pub fn new(version: u8, tag: RenderResourceTag, index: u32) -> Self {
        let version: u32 = version as u32;
        let tag: u32 = tag as u32;
        let index: u32 = index as u32;

        // version wraps around, it's just to make sure invalid resources don't get another version
        assert!(version < 64);
        assert!(tag < 8);
        assert!(index < (1 << 23));

        Self(version << 26 | tag << 23 | index)
    }
```

### HLSL



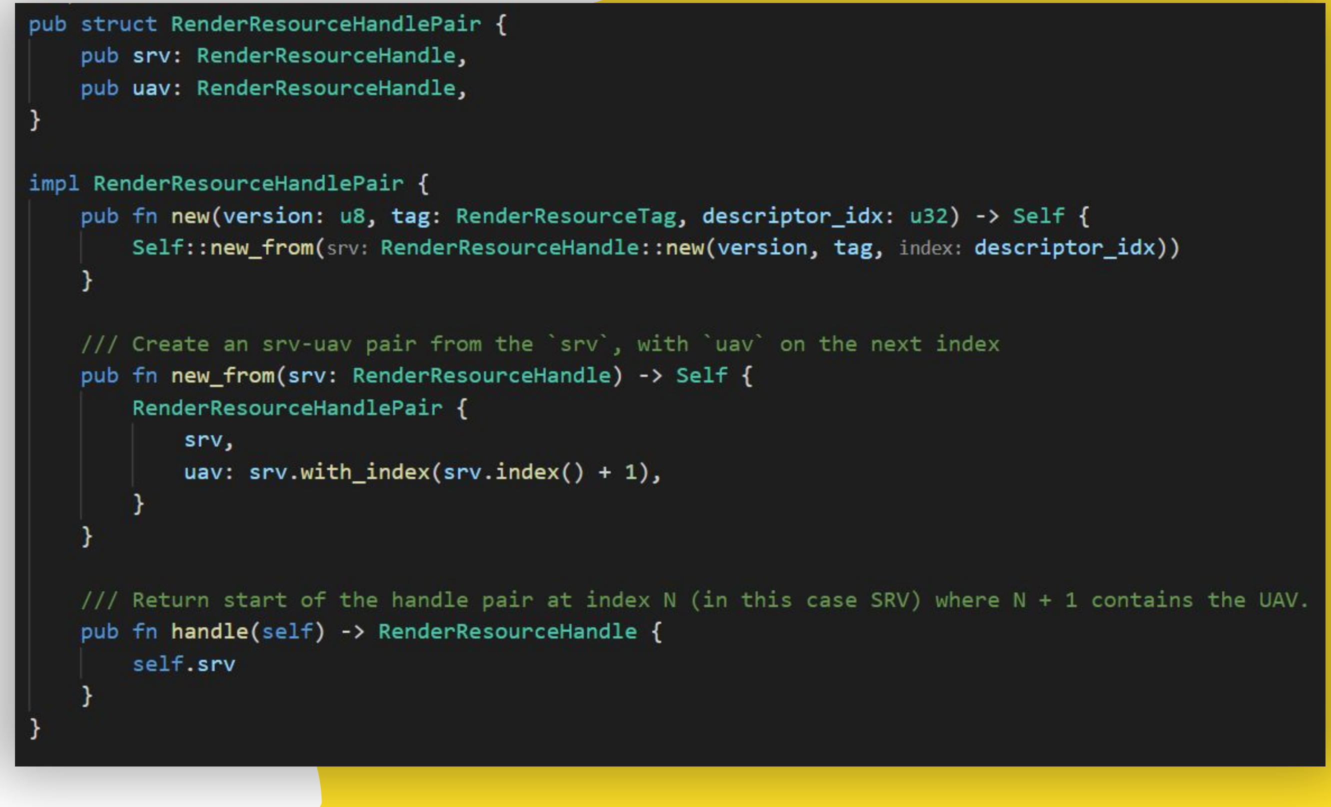
```hlsl
struct RenderResourceHandle {
    uint handle;

    uint read_index() { return this.handle & ((1 << 23) - 1); }
    bool is_valid() { return this.handle != ~0; }

#ifdef VK_BINDLESS
    uint write_index() { return this.handle & ((1 << 23) - 1); }
#else
    uint write_index() {
        uint uav_idx = this.handle + 1;
        return uav_idx & ((1 << 23) - 1);
    }
#endif
};
```

# BINDLESS RENDERING

## Rendering resource handles

- Introduction of *RenderResourceHandlePair*
- SRV & UAV descriptors are assumed at index { [n], [n+1] }
- Each (sub)resource has a *RenderResourceHandlePair*

## IMPLEMENTATION



```rust
pub struct RenderResourceHandlePair {
    pub srv: RenderResourceHandle,
    pub uav: RenderResourceHandle,
}

impl RenderResourceHandlePair {
    pub fn new(version: u8, tag: RenderResourceTag, descriptor_idx: u32) -> Self {
        Self::new_from(srv: RenderResourceHandle::new(version, tag, index: descriptor_idx))
    }

    /// Create an srv-uav pair from the `srv`, with `uav` on the next index
    pub fn new_from(srv: RenderResourceHandle) -> Self {
        RenderResourceHandlePair {
            srv,
            uav: srv.with_index(srv.index() + 1),
        }
    }

    /// Return start of the handle pair at index N (in this case SRV) where N + 1 contains the UAV.
    pub fn handle(self) -> RenderResourceHandle {
        self.srv
    }
}
```

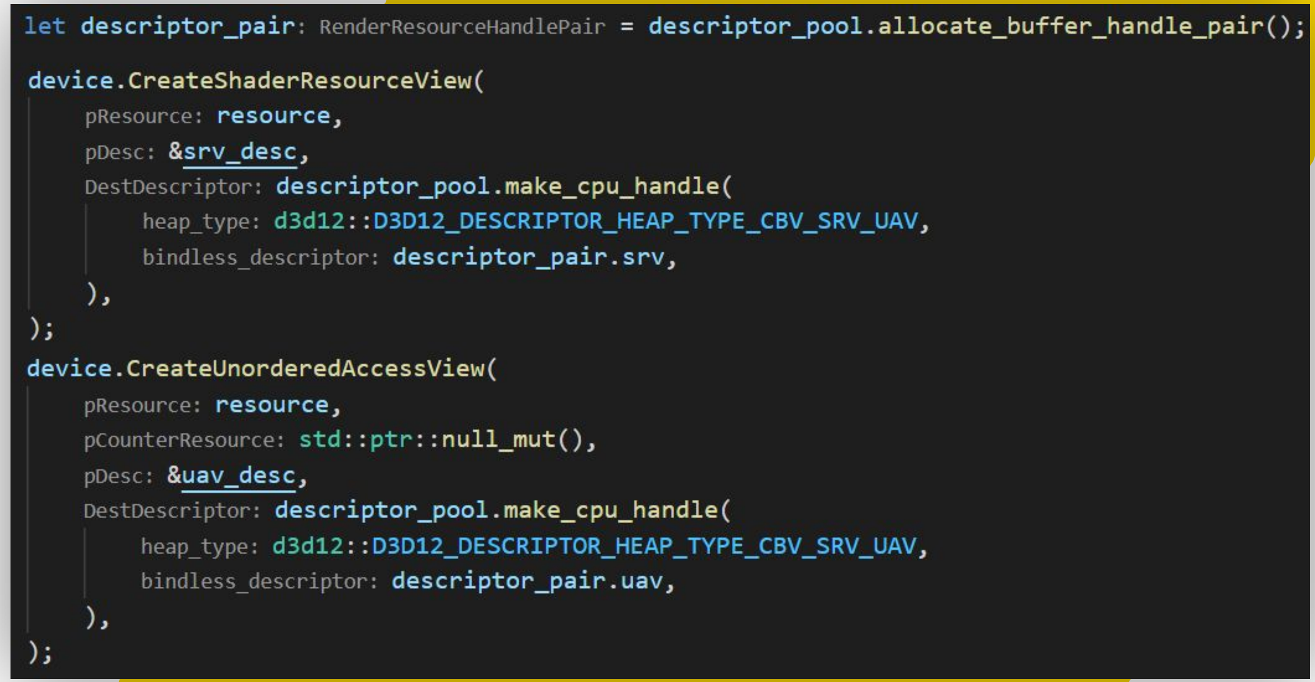
# BINDLESS RENDERING

## Rendering resource handles

- Each resource has a *RenderResourceHandlePair*, additionally Render targets/Depth stencils may have additional RTV / DSV handles in their according heap types.

- Create the views once during resource creation.

## VIEWS CREATED DURING RESOURCE INIT

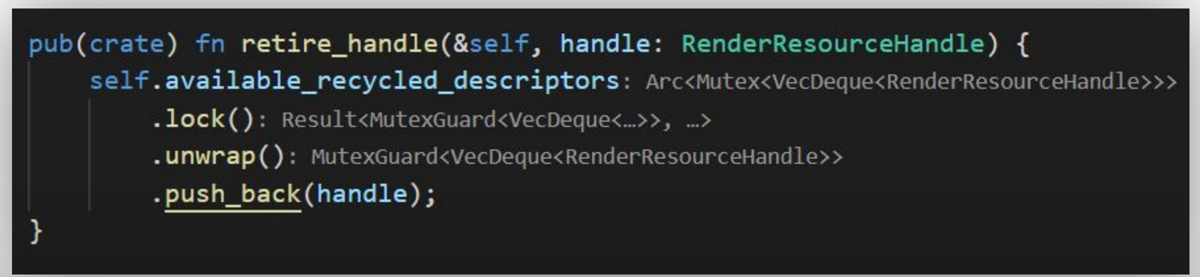


```
let descriptor_pair: RenderResourceHandlePair = descriptor_pool.allocate_buffer_handle_pair();

device.CreateShaderResourceView(
    pResource: resource,
    pDesc: &srv_desc,
    DestDescriptor: descriptor_pool.make_cpu_handle(
        heap_type: d3d12::D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV,
        bindless_descriptor: descriptor_pair.srv,
    ),
);
device.CreateUnorderedAccessView(
    pResource: resource,
    pCounterResource: std::ptr::null_mut(),
    pDesc: &uav_desc,
    DestDescriptor: descriptor_pool.make_cpu_handle(
        heap_type: d3d12::D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV,
        bindless_descriptor: descriptor_pair.uav,
    ),
);
```

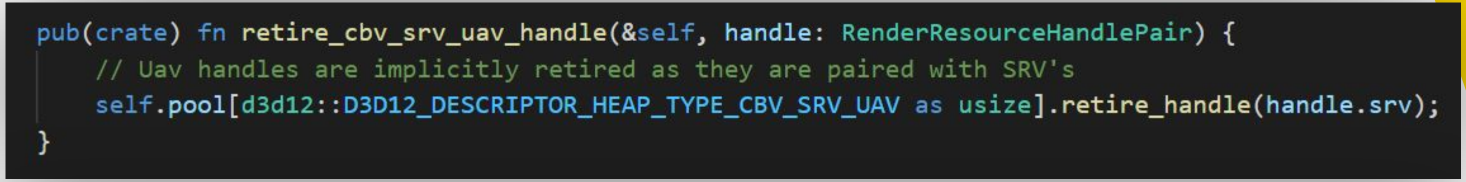
# BINDLESS RENDERING

## Resource handle management

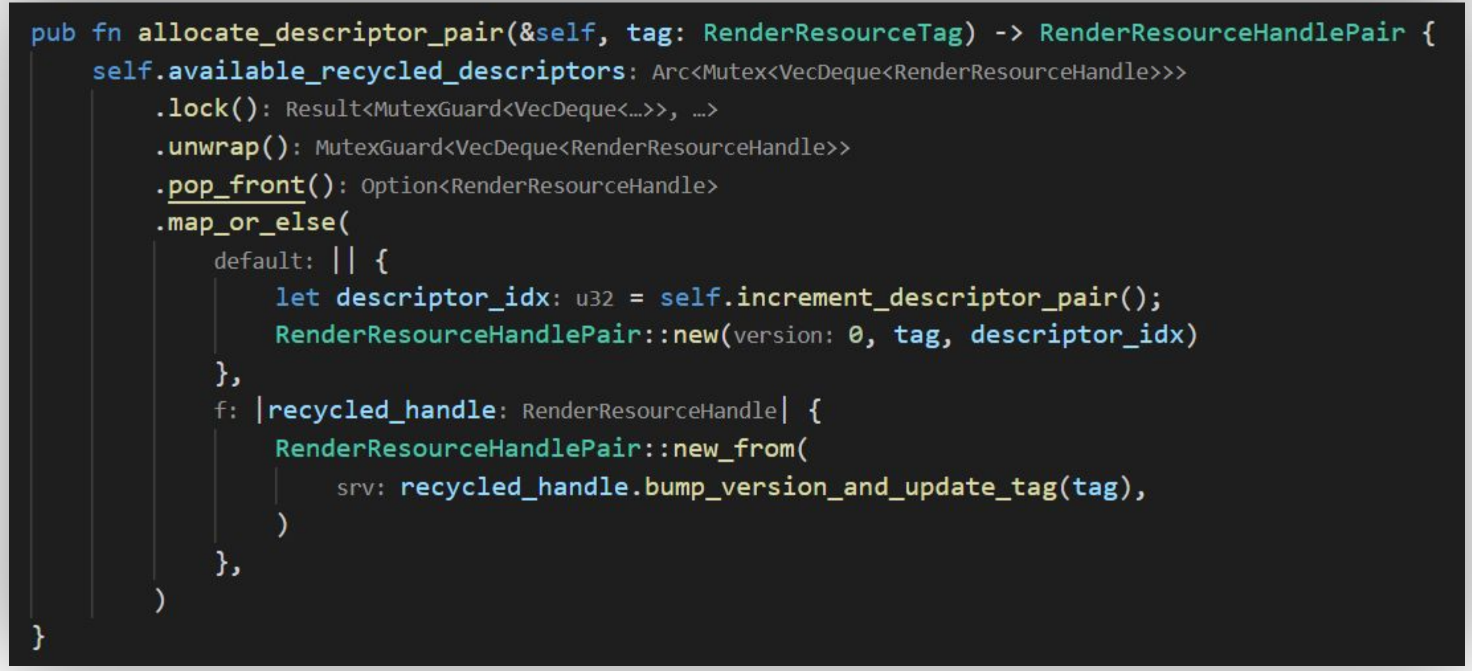
- Upon dropping resources, recycle the descriptor.
- Upon allocating RenderResourceHandlePairs, check for recycled handles.
- RenderResourceHandlePairs are recycled in FIFO order.
- Permanent fragmentation may occur if FIFO order is not respected.
- Only recycle descriptors when they are no longer in use!



```
pub(crate) fn retire_handle(&self, handle: RenderResourceHandle) {
    self.available_recycled_descriptors: Arc<Mutex<VecDeque<RenderResourceHandle>>>
        .lock(): Result<MutexGuard<VecDeque<…>>, …>
        .unwrap(): MutexGuard<VecDeque<RenderResourceHandle>>
        .push_back(handle);
}
```



```
pub(crate) fn retire_cbv_srv_uav_handle(&self, handle: RenderResourceHandlePair) {
    // Uav handles are implicitly retired as they are paired with SRV's
    self.pool[d3d12::D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV as usize].retire_handle(handle.srv);
}
```



```
pub fn allocate_descriptor_pair(&self, tag: RenderResourceTag) -> RenderResourceHandlePair {
    self.available_recycled_descriptors: Arc<Mutex<VecDeque<RenderResourceHandle>>>
        .lock(): Result<MutexGuard<VecDeque<…>>, …>
        .unwrap(): MutexGuard<VecDeque<RenderResourceHandle>>
        .pop_front(): Option<RenderResourceHandle>
        .map_or_else(
            default: || {
                let descriptor_idx: u32 = self.increment_descriptor_pair();
                RenderResourceHandlePair::new(version: 0, tag, descriptor_idx)
            },
            f: |recycled_handle: RenderResourceHandle| {
                RenderResourceHandlePair::new_from(
                    srv: recycled_handle.bump_version_and_update_tag(tag),
                )
            },
        )
}
```

TRAVERSE RESEARCH

# BINDLESS RENDERING

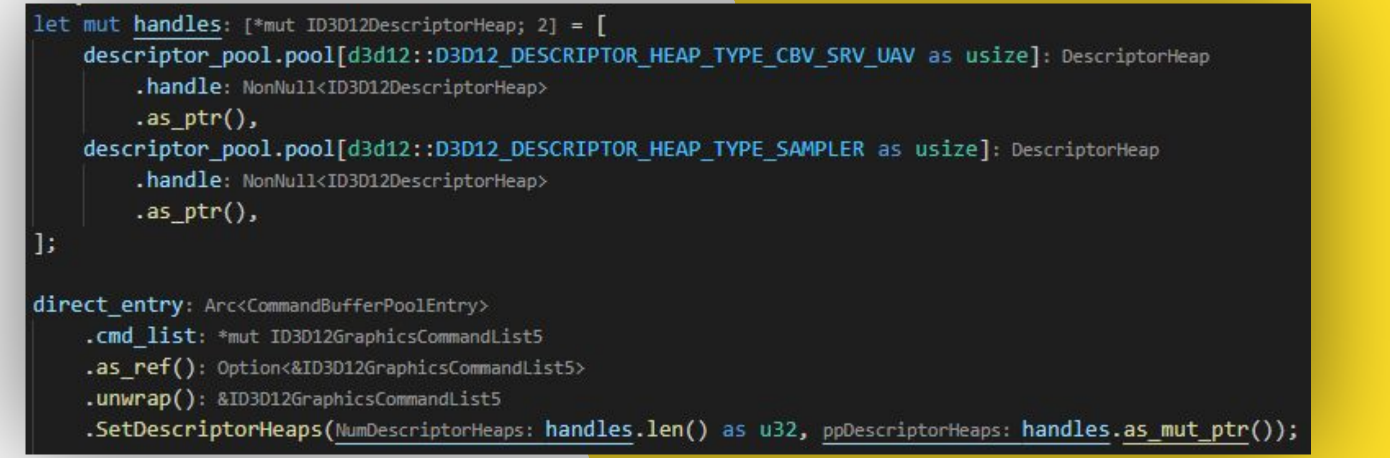
## Preparing for bindless rendering

01 Set descriptor heaps.

02 Set graphics/compute root signatures.

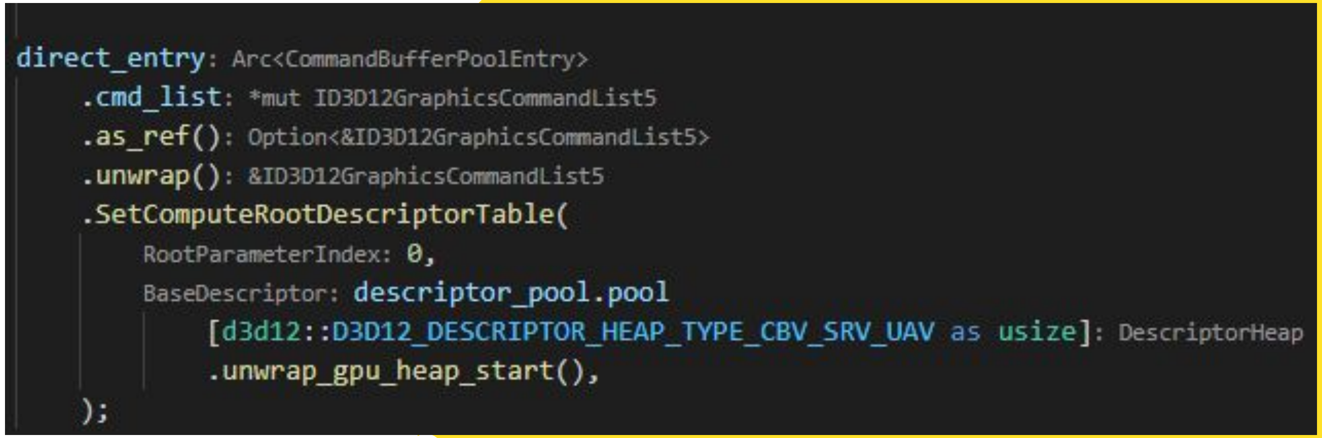
03 (sm6.5) bind descriptor tables.

## BIND HEAPS



```
let mut handles: [*mut ID3D12DescriptorHeap; 2] = [
    descriptor_pool.pool[d3d12::D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV as usize]: DescriptorHeap
        .handle: NonNull<ID3D12DescriptorHeap>
        .as_ptr(),
    descriptor_pool.pool[d3d12::D3D12_DESCRIPTOR_HEAP_TYPE_SAMPLER as usize]: DescriptorHeap
        .handle: NonNull<ID3D12DescriptorHeap>
        .as_ptr(),
];

direct_entry: Arc<CommandBufferPoolEntry>
    .cmd_list: *mut ID3D12GraphicsCommandList5
    .as_ref(): Option<&ID3D12GraphicsCommandList5>
    .unwrap(): &ID3D12GraphicsCommandList5
    .SetDescriptorHeaps(NumDescriptorHeaps: handles.len() as u32, ppDescriptorHeaps: handles.as_mut_ptr());
```

## SET DESCRIPTOR TABLES (SM6.5)



```
direct_entry: Arc<CommandBufferPoolEntry>
    .cmd_list: *mut ID3D12GraphicsCommandList5
    .as_ref(): Option<&ID3D12GraphicsCommandList5>
    .unwrap(): &ID3D12GraphicsCommandList5
    .SetComputeRootDescriptorTable(
        RootParameterIndex: 0,
        BaseDescriptor: descriptor_pool.pool
            [d3d12::D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV as usize]: DescriptorHeap
            .unwrap_gpu_heap_start(),
    );
```

Bindless Rendering

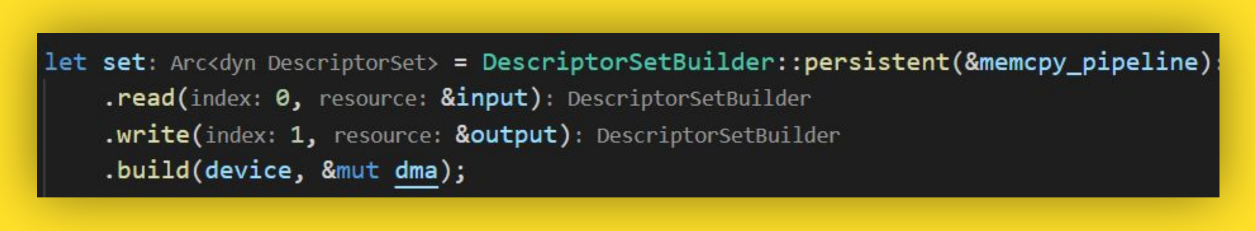
# DESCRIPTOR SETS

01 Create a buffer containing RenderResourceHandles.

02 The created buffer has a RenderResourceHandle.

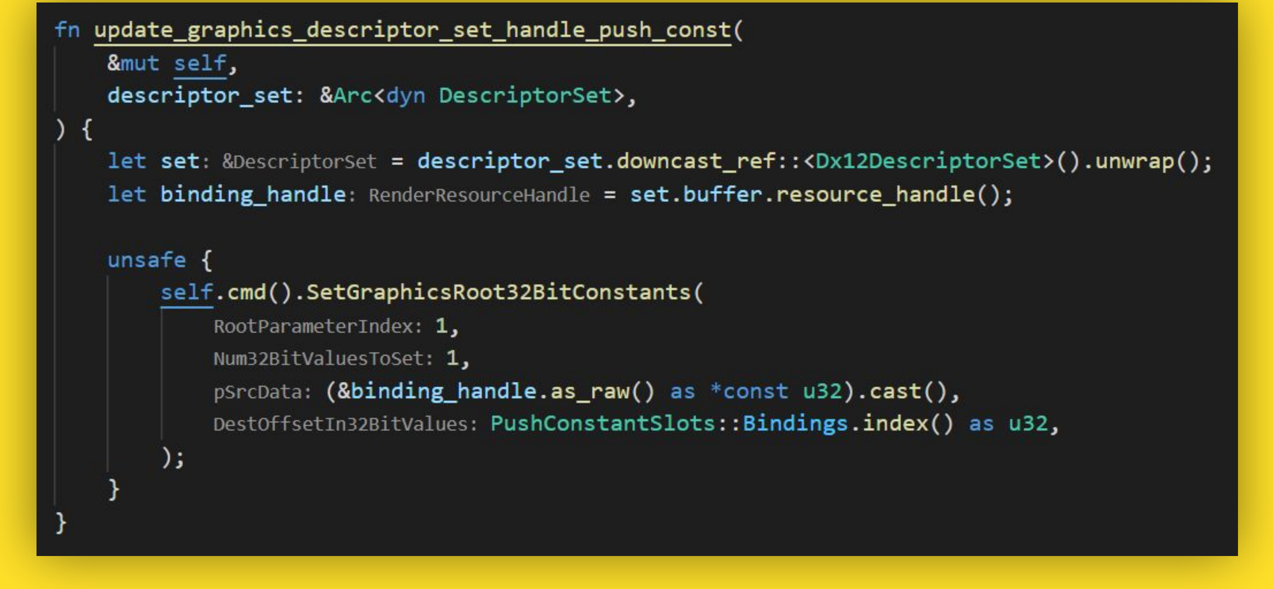
03 Set root constant with the handle of the created buffer.

## BUILD BUFFER WITH RENDER HANDLES



```
let set: Arc<dyn DescriptorSet> = DescriptorSetBuilder::persistent(&memcpy_pipeline)
    .read(index: 0, resource: &input): DescriptorSetBuilder
    .write(index: 1, resource: &output): DescriptorSetBuilder
    .build(device, &mut dma);
```

## SET HANDLE WITH PUSH CONSTANT



```
fn update_graphics_descriptor_set_handle_push_const(
    &mut self,
    descriptor_set: &Arc<dyn DescriptorSet>,
) {
    let set: &DescriptorSet = descriptor_set.downcast_ref::<Dx12DescriptorSet>().unwrap();
    let binding_handle: RenderResourceHandle = set.buffer.resource_handle();

    unsafe {
        self.cmd().SetGraphicsRoot32BitConstants(
            RootParameterIndex: 1,
            Num32BitValuesToSet: 1,
            pSrcData: (&binding_handle.as_raw() as *const u32).cast(),
            DestOffsetIn32BitValues: PushConstantSlots::Bindings.index() as u32,
        );
    }
}
```

TRAVERSE RESEARCH

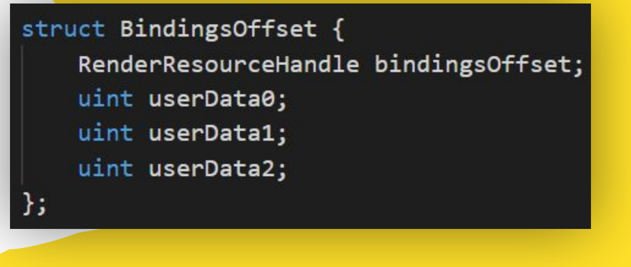
# BINDLESS RENDERING

## Shader model 6.5 bindless shaders

04 Load bindings in shader.

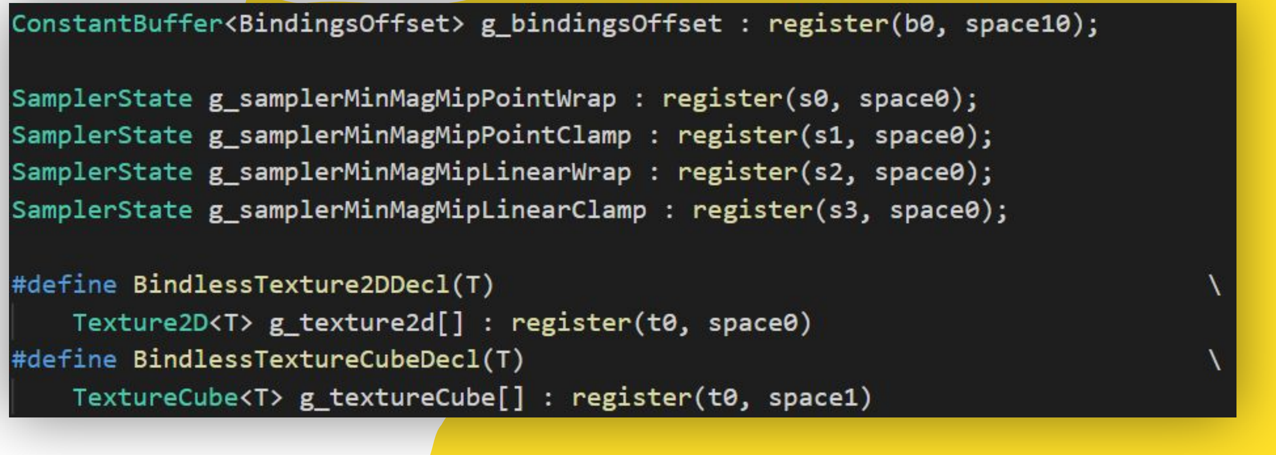
05 Dynamically index in descriptor heap.

## ROOT CONSTANTS



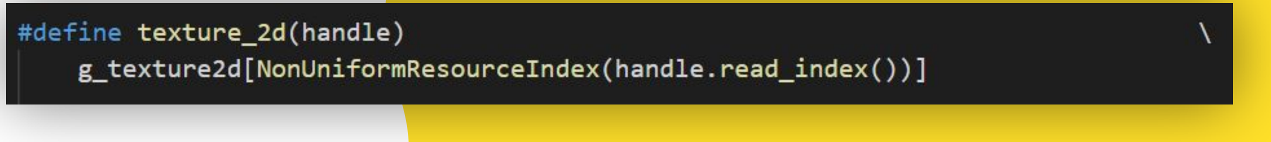
```
struct BindingsOffset {
    RenderResourceHandle bindingsOffset;
    uint userData0;
    uint userData1;
    uint userData2;
};
```

## DECLARATION OF ALL RESOURCES



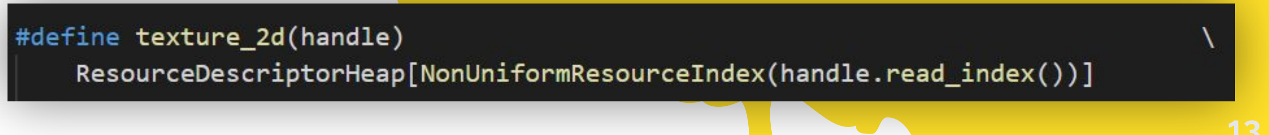
```
ConstantBuffer<BindingsOffset> g_bindingsOffset : register(b0, space10);

SamplerState g_samplerMinMagMipPointWrap : register(s0, space0);
SamplerState g_samplerMinMagMipPointClamp : register(s1, space0);
SamplerState g_samplerMinMagMipLinearWrap : register(s2, space0);
SamplerState g_samplerMinMagMipLinearClamp : register(s3, space0);

#define BindlessTexture2DDecl(T)                                          \
    Texture2D<T> g_texture2d[] : register(t0, space0)
#define BindlessTextureCubeDecl(T)                                        \
    TextureCube<T> g_textureCube[] : register(t0, space1)
```

## DYNAMIC INDEXING 6.5



```
#define texture_2d(handle)                                                \
    g_texture2d[NonUniformResourceIndex(handle.read_index())]
```

## DYNAMIC INDEXING 6.6

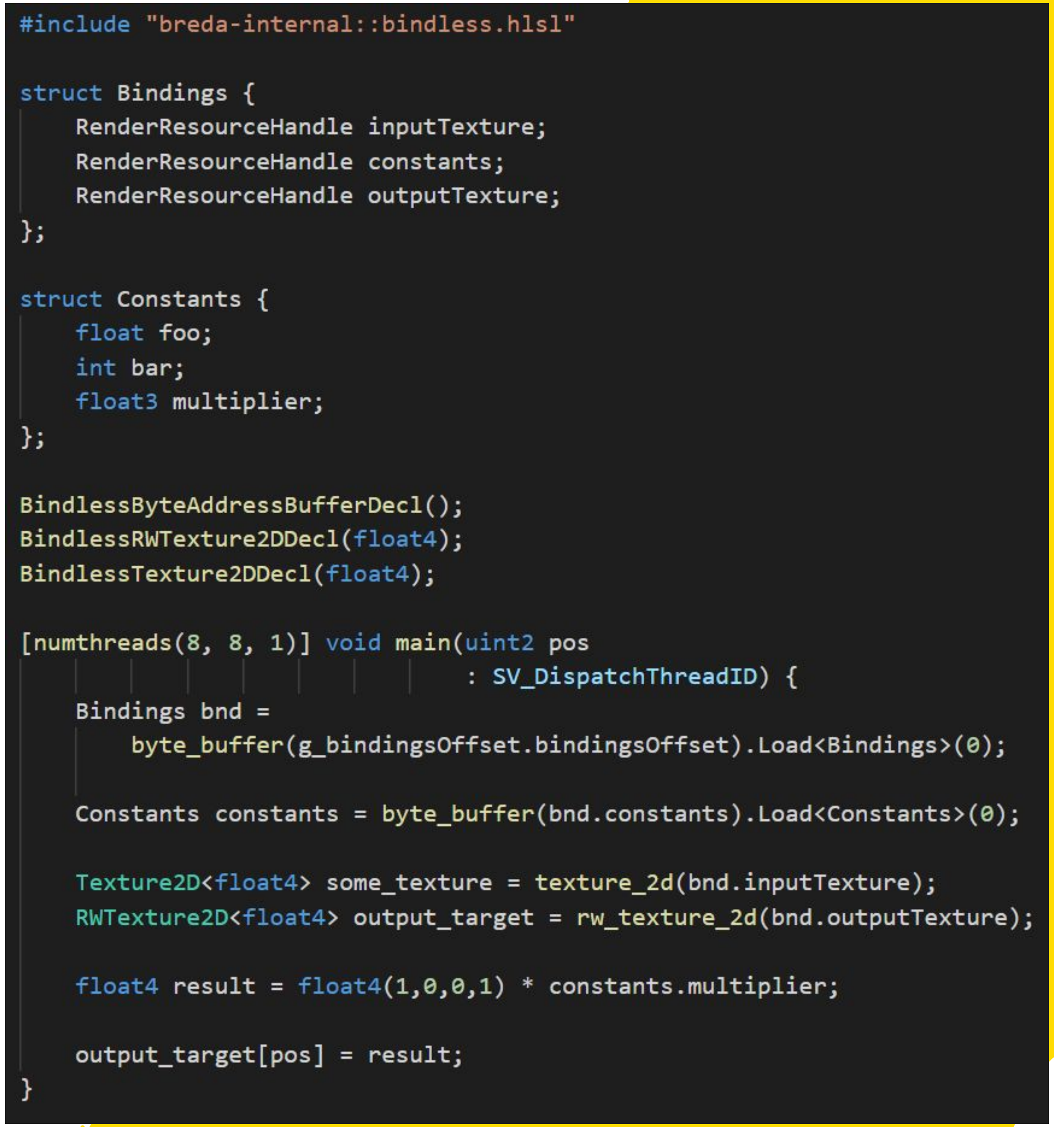


```
#define texture_2d(handle)                                                \
    ResourceDescriptorHeap[NonUniformResourceIndex(handle.read_index())]
```

# BINDLESS RENDERING

## Bindless shader example

- pre-sm6.6 approach is forced to use declarations and functions for dynamic indexing.
- We only use (RW)ByteAddressBuffers for buffer types, templated load and store helps a lot.
- Still not ideal, sm6.6 template approach is ideal.

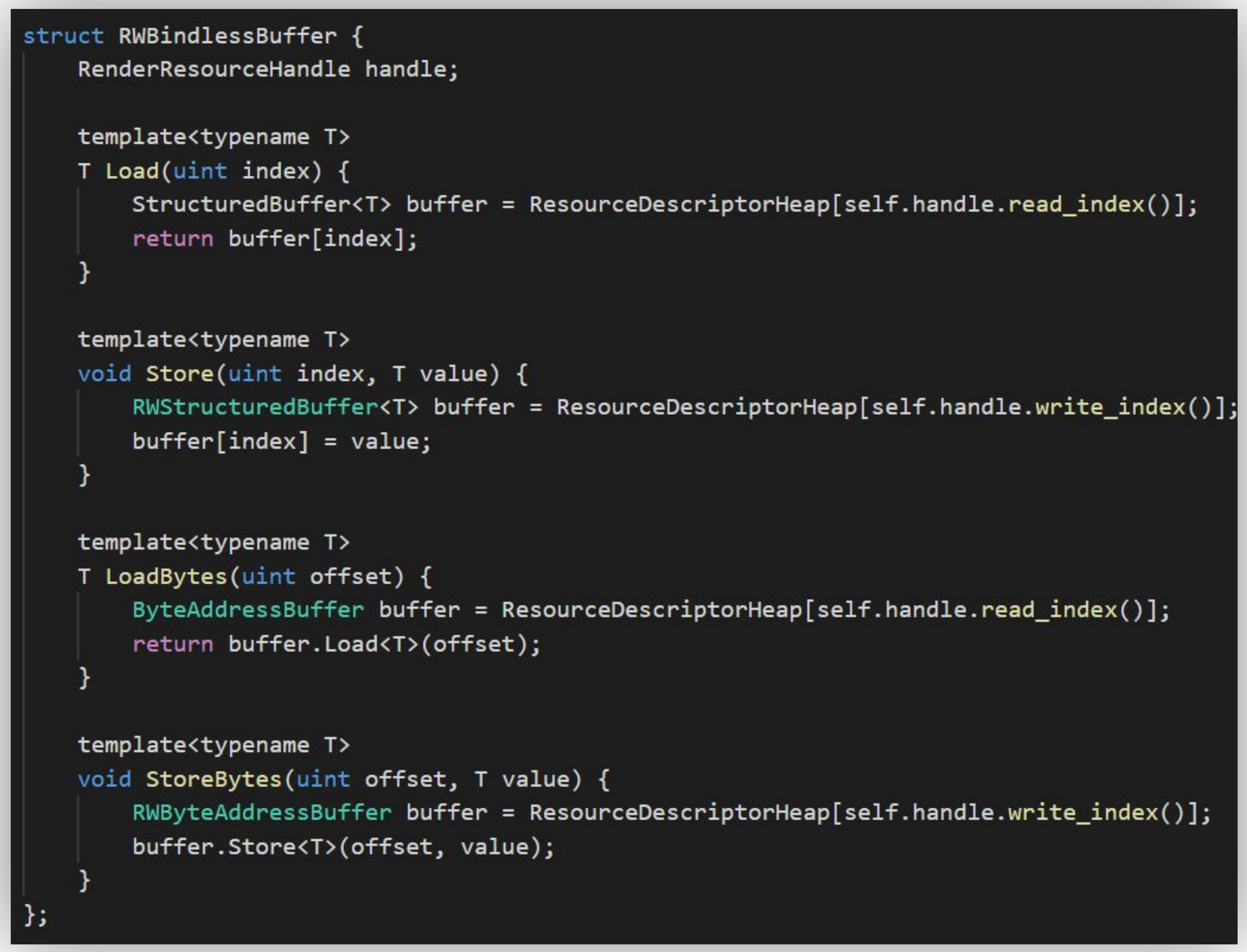


```
#include "breda-internal::bindless.hlsl"

struct Bindings {
    RenderResourceHandle inputTexture;
    RenderResourceHandle constants;
    RenderResourceHandle outputTexture;
};

struct Constants {
    float foo;
    int bar;
    float3 multiplier;
};

BindlessByteAddressBufferDecl();
BindlessRWTexture2DDecl(float4);
BindlessTexture2DDecl(float4);

[numthreads(8, 8, 1)] void main(uint2 pos
                                : SV_DispatchThreadID) {
    Bindings bnd =
        byte_buffer(g_bindingsOffset.bindingsOffset).Load<Bindings>(0);

    Constants constants = byte_buffer(bnd.constants).Load<Constants>(0);

    Texture2D<float4> some_texture = texture_2d(bnd.inputTexture);
    RWTexture2D<float4> output_target = rw_texture_2d(bnd.outputTexture);

    float4 result = float4(1,0,0,1) * constants.multiplier;

    output_target[pos] = result;
}
```

# BONUS SLIDE

## Future directions with templating

- More specialized RenderHandles with the use of templates.



```
struct RWBindlessBuffer {
    RenderResourceHandle handle;

    template<typename T>
    T Load(uint index) {
        StructuredBuffer<T> buffer = ResourceDescriptorHeap[self.handle.read_index()];
        return buffer[index];
    }

    template<typename T>
    void Store(uint index, T value) {
        RWStructuredBuffer<T> buffer = ResourceDescriptorHeap[self.handle.write_index()];
        buffer[index] = value;
    }

    template<typename T>
    T LoadBytes(uint offset) {
        ByteAddressBuffer buffer = ResourceDescriptorHeap[self.handle.read_index()];
        return buffer.Load<T>(offset);
    }

    template<typename T>
    void StoreBytes(uint offset, T value) {
        RWByteAddressBuffer buffer = ResourceDescriptorHeap[self.handle.write_index()];
        buffer.Store<T>(offset, value);
    }
};
```

# THANK YOU