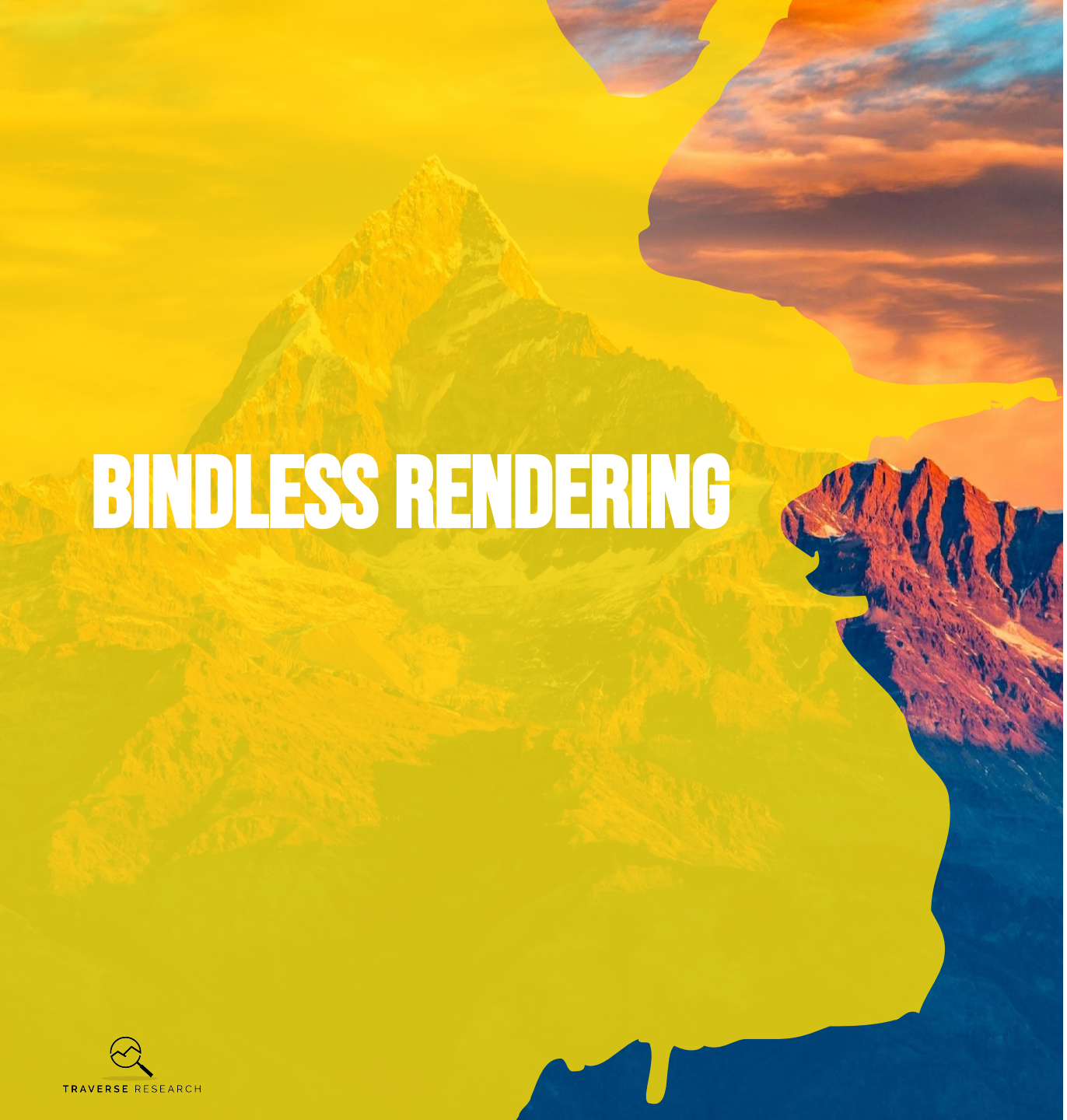
TRAVERSE RESEARCH

Setting up a bindless rendering pipeline

Darius Bouma
Rendering Engineer at Traverse Research
twitter.com/dbalthazr
mastodon.gamedev.place/@DBouma

Presented at the Khronos Vulkanised 2023 Conference



AGENDA

01 Setting a goal

02 Setup

03 Resource handles & management

04 VK Bindless indexing in HLSL

05 Questions

TRAVERSE RESEARCH

SETTING A GOAL

01 Create a bindless system that can be used between graphics APIs

02 Write API-agnostic shaders that support bindless resources in HLSL

03 Load any resource type from any location

04 Overcome the existing limitations with HLSL

IN A PERFECT WORLD

```
struct Material {
    Texture2D<float4> albedo;
    // etc etc...
};

struct Resources {
    StructuredBuffer<Material> materials;
    RWTexture2D<float4> output;
};

[numthreads(8, 8, 1)] void main(int2 threadId
                                : SV_DispatchThreadID) {
    Resources resources = loadResources<Resources>();

    // For illustration purproses
    uint someMaterialIndex = threadId.x;

    // Load the material
    Material material = resources.materials[someMaterialIndex];

    float4 someSample = material.ableDo.Load(threadId);

    // Do something with sample

    resources.output[threadId] = someSample;
}
```

SETUP

Categorizing descriptors

01 Resources can be categorized as the following types:

- Buffers
- Sampled images
- Storage images
- Acceleration structures
- Samplers

02 Vulkan 1.2 spec guarantees a minimum of 4 sets.

- We made the decision to use immutable samplers for the time being to meet minimum spec for certain devices
- VK_EXT_mutable_descriptor_type
- VK_EXT_descriptor_buffer

More implementation details can be found in our blogpost

CATEGORIZED DESCRIPTOR TYPES

```
pub enum BindlessTableType {
    Buffer,
    Texture,
    RwTexture,
    Tlas,
}
```

```
pub fn to_vk(self) -> vk::DescriptorType {
    match self {
        Self::Buffer => vk::DescriptorType::STORAGE_BUFFER,
        Self::Texture => vk::DescriptorType::SAMPLED_IMAGE,
        Self::RwTexture => vk::DescriptorType::STORAGE_IMAGE,
        Self::Tlas => vk::DescriptorType::ACCELERATION_STRUCTURE_KHR,
    }
}
```

SETUP

Descriptor pool

01 The vulkan *DescriptorPool* requires a predetermined amount of descriptors per resource type.

02 *descriptor_count* is set to an upper bound
- We simply use 100,000 here

03 Immutable samplers are treated separately, and are set to the user defined maximum amount of samplers

RETRIEVING DESCRIPTORPOOL SIZES

```
pub fn descriptor_pool_sizes(immutable_sampler_count: u32) -> Vec<vk::DescriptorPoolSize> {
    Self::ALL_TABLES &[BindlessTableType]
        .iter() Iter<BindlessTableType>
        .map(|table: &BindlessTableType| vk::DescriptorPoolSize {
            ty: table.to_vk(),
            descriptor_count: table.table_size(),
        }) impl Iterator<Item = DescriptorPoolSize>
        .chain(std::iter::once(vk::DescriptorPoolSize {
            ty: vk::DescriptorType::SAMPLER,
            descriptor_count: immutable_sampler_count,
        })) impl Iterator<Item = DescriptorPoolSize>
        .collect::<Vec<_>>()
}
```

CREATING DESCRIPTOR POOL

```
let descriptor_sizes: Vec<DescriptorPoolSize> =
    BindlessTableType::descriptor_pool_sizes(immutable_sampler_count: immutable_samplers.len() as u32);

let descriptor_pool_info: DescriptorPoolCreateInfoBuilder = vk::DescriptorPoolCreateInfo::builder() Des
    .pool_sizes(&descriptor_sizes) DescriptorPoolCreateInfoBuilder
    .flags(vk::DescriptorPoolCreateFlags::UPDATE_AFTER_BIND) DescriptorPoolCreateInfoBuilder
    .max_sets(4);

let pool_handle: DescriptorPool = unsafe {
    device &Arc<Device>
        .create_descriptor_pool(create_info: &descriptor_pool_info, allocation_callbacks: None) Result<D
        .unwrap()
};
```

SETUP

DescriptorSetLayout

01 A single *DescriptorSetLayout* and *PipelineLayout* is used for all pipelines

02 Each descriptor set contains a predetermined maximum amount of that specific resource type

03 If Immutable samplers are used, they need to be included here unlike Dx12

LAYOUT FLAGS & LAYOUT BINDINGS

```
let mut descriptor_binding_flags: Vec<DescriptorBindingFlags> = vec![
    vk::DescriptorBindingFlags::PARTIALLY_BOUND
        | vk::DescriptorBindingFlags::VARIABLE_DESCRIPTOR_COUNT
        | vk::DescriptorBindingFlags::UPDATE_AFTER_BIND,
];

let mut set: Vec<DescriptorSetLayoutBinding> = vec![vk::DescriptorSetLayoutBinding {
    binding: 0,
    descriptor_type: table.to_vk(),
    descriptor_count: table.table_size(),
    stage_flags: vk::ShaderStageFlags::ALL,
    p_immutable_samplers: std::ptr::null(),
}];
```

SETUP

PipelineLayout

01 Finally the pipeline layout only needs to know about the push constant slots.

02 Push constants are used to communicate our resource indices to the GPU:

- A handle to the buffer containing resources
- User data
- Debug handles for writing shader logging/asserts
- Version heap for bindless validation

PUSH CONSTANTS

```
// There are more push constant slots with debug functionality enabled
let num_push_constants: u32 = PushConstantSlots::num_push_constant_slots(debug) as u32;
let num_push_constants_sized: u32 = std::mem::size_of::<u32>() as u32 * num_push_constants;

let push_constant_range: PushConstantRange = ash::vk::PushConstantRange {
    stage_flags: vk::ShaderStageFlags::ALL,
    offset: 0,
    size: num_push_constants_sized,
};
```

Bindless Rendering

SETUP

Setting up command buffers

01 Once per command buffer, we bind our bindless descriptor sets (Graphics, Compute, RT)

02 Every pass we update the push constants to contain the associated “descriptorset buffer”

- This is just a buffer with indices to our resource array

BINDLESS SETS

```
unsafe {
    if self.queue_flags.contains(vk::QueueFlags::GRAPHICS) {
        self.device.cmd_bind_descriptor_sets(
            command_buffer: self.cmd,
            pipeline_bind_point: vk::PipelineBindPoint::GRAPHICS,
            self.allocation_handles VulkanDeviceAllocationHandles
                .descriptor_pool Arc<DescriptorPool>
                .bindless_pipeline_layout,
            first_set: 0,
            self.allocation_handles VulkanDeviceAllocationHandles
                .descriptor_pool Arc<DescriptorPool>
                .bindless_descriptor_sets(),
            dynamic_offsets: &[],
        );
    }
}
```

UPDATING PUSH CONSTANTS

```
let descriptor_set: &DescriptorSet = descriptor_set.downcast_ref::<VkDescriptorSet>().unwrap();
let handle: RenderResourceHandle = unsafe { descriptor_set.buffer.resource_handle() };

unsafe {
    self.device.cmd_push_constants(
        command_buffer: self.cmd,
        self.allocation_handles VulkanDeviceAllocationHandles
            .descriptor_pool Arc<DescriptorPool>
            .bindless_pipeline_layout,
        stage_flags: vk::ShaderStageFlags::ALL,
        offset: 0,
        constants: safe_transmute::transmute_one_to_bytes(from: &handle.internal_as_raw()),
    );
}
```

RESOURCE HANDLE

A *RenderResourceHandle* is essentially just a u32 with packed information.

- 23 bits for the index
- 2 bits to identify the resource type
- 1 bit for resource writability
- 6 bits to track the handle version

```rust
#[derive(Copy, Clone, Eq, PartialEq, Hash)]
#[repr(transparent)]
pub struct RenderResourceHandle(u32);
impl RenderResourceHandle {
    pub fn new(version: u8, tag: RenderResourceTag, index: u32, access_type: AccessType) -> Self {
        let version: u32 = version as u32;
        let tag: u32 = tag as u32;
        let index: u32 = index;
        let access_type: u32 = access_type.is_read_write() as u32;
        Self(version << 26 | access_type << 25 | tag << 23 | index)
    }
```

RESOURCE HANDLE

01 *RenderResourceHandles* are created exclusively during resource allocation

02 Handles are only recycled if both of the following conditions are met:
- The resource does not have ref counts anymore
- The GPU has finished executing work that references these resources

03 Small buffers containing resource handles are created to communicate resources to GPU using push constants

04 Handles contain validation bits to validate correct resource access on the GPU

05 Handles track their version, if a version mismatches on the GPU, validation fails

06 Upon validation failure, the resource is not read or written to, to avoid inevitable page fault

RESOURCE HANDLE

Allocating handles & writing descriptors

Descriptor allocation and recycling should be as simple as possible, we ended up with a FIFO queue.

- Descriptors are then updated in the actual vulkan *DescriptorPool* at the corresponding descriptor set, at array index *N*
- A *RenderResourceHandle* is constructed or recycled with index *N*

ALLOCATING A BUFFER HANDLE

```
pub fn allocate_buffer_handle(&self, buffer: vk::Buffer) -> RenderResourceHandle {
    const TYPE: BindlessTableType = BindlessTableType::Buffer;
    let handle: RenderResourceHandle =
        self.fetch_available_descriptor(tag: RenderResourceTag::Buffer, AccessType::ReadWrite);

    let buffer_info: DescriptorBufferInfo = vk::DescriptorBufferInfo {
        buffer,
        offset: 0,
        range: vk::WHOLE_SIZE,
    };

    let write: WriteDescriptorSetBuilder = vk::WriteDescriptorSet::builder() WriteDescriptorSetBuilder
        .dst_set(self.sets[TYPE.set_index()]) WriteDescriptorSetBuilder
        .dst_binding(0) WriteDescriptorSetBuilder
        .dst_array_element(handle.index()) WriteDescriptorSetBuilder
        .descriptor_type(TYPE.to_vk()) WriteDescriptorSetBuilder
        .buffer_info(std::slice::from_ref(&buffer_info));

    unsafe {
        self.device Arc<Device>
            .update_descriptor_sets(descriptor_writes: std::slice::from_ref(&write), descriptor_copies: &[])
    };

    handle
} fn allocate_buffer_handle
```

Bindless Rendering

RESOURCE HANDLE

Recycling ResourceHandles & validation

01 Once a handle is reused, the version, tag and writable bits are updated.

02 Additionally a mirror heap is tracked for validation purposes

03 Upon validation failure a small packet is written back to the CPU to indicate a validation error.

BUMPING VERSION AND TAG

```
pub fn bump_version_and_update_tag(
    self,
    tag: RenderResourceTag,
    access_type: AccessType,
) -> Self {
    let next_version: u32 = (self.version() + 1) % 64;
    Self::new(next_version as u8, tag, self.index(), access_type)
}
```

Bindless Rendering

BINDLESS HLSL

Setup

01 The previously created buffer containing *RenderResourceHandles* is a resource by itself

02 The *RenderResourceHandle* of this buffer is simply a packed u32, which we communicate to the GPU using push constants.

03 The shader then loads this buffer from the *ResourceDescriptorHeap* or in the case of Vulkan, loads it from the emulation layer.

HLSL DECLARATIONS

```
struct RenderResourceHandle {
    // TODO(Darius) Switch to bitfields once it's supported by SPIR-V
    // https://github.com/microsoft/DirectXShaderCompiler/issues/4295
    uint handle;

    bool isValid() { return this.handle != ~0; }
    uint resourceTag() { return (this.handle >> 23) & ((1 << 2) - 1); }
    bool isWritable() { return (this.handle >> 25) & 1; }
    uint version() { return (this.handle >> 26) & ((1 << 6) - 1); }

    uint readIndex() { return this.handle & ((1 << 23) - 1); }
    // Indices for both read and write are the same in our VK bindless impl.
#ifdef VK_BINDLESS
    uint writeIndex() { return this.readIndex(); }
#else
    // Uav index in dx12 is located at read index + 1.
    uint writeIndex() { return this.readIndex() + 1; }
#endif
};
```

```
struct BindingsData {
    RenderResourceHandle bindingsOffset;
    uint userData0;
    uint userData1;
    uint userData2;

#if defined(SHADER_LOGGING)
    RenderResourceHandle infoBufferHandle;
    uint commandIndex;
    RenderResourceHandle versionHeap;
#endif // SHADER_LOGGING
};
```

BINDLESS HLSL

Emulation layer

01 Vk does not have something like sm6.6 *ResourceDescriptorHeap* yet

02 We need to declare all possible resource types in advance with register overlapping to achieve roughly the same as the Dx12 counterpart

03 The usage of *StructuredBuffer<T>* is impossible, use *ByteAddressBuffer* instead.

HLSL PREDECLARING RESOURCES

```
#define ITERATE_TEXTURE_TYPES(ITERATOR, ...)                                    \
    ITERATOR(int, ##__VA_ARGS__)                                                \
    ITERATOR(uint, ##__VA_ARGS__)                                               \
    ITERATOR(float, ##__VA_ARGS__)                                              \
    ITERATOR(int2, ##__VA_ARGS__)                                               \
    ITERATOR(uint2, ##__VA_ARGS__)                                              \
    ITERATOR(float2, ##__VA_ARGS__)                                             \
    ITERATOR(int3, ##__VA_ARGS__)                                               \
    ITERATOR(uint3, ##__VA_ARGS__)                                              \
    ITERATOR(float3, ##__VA_ARGS__)                                             \
    ITERATOR(int4, ##__VA_ARGS__)                                               \
    ITERATOR(uint4, ##__VA_ARGS__)                                              \
    ITERATOR(float4, ##__VA_ARGS__)                                             \
    /*ITERATOR(int64_t, ##__VA_ARGS__)                                          \
    ITERATOR(uint64_t, ##__VA_ARGS__)*/

#define _GENERATE_TEXTURE_TYPE_SLOT(nativeType, textureType, bindingA, bindingB)\
    [[vk::binding(bindingA, bindingB)]] textureType<nativeType>                 \
        g_##textureType##nativeType[BINDLESS_DESCRIPTOR_HEAP_SIZE];

#define DEFINE_TEXTURE_TYPES_AND_FORMATS_SLOTS(textureType, bindingA, bindingB) \
    ITERATE_TEXTURE_TYPES(_GENERATE_TEXTURE_TYPE_SLOT, textureType, bindingA, bindingB)

DEFINE_TEXTURE_TYPES_AND_FORMATS_SLOTS(Texture1D, NUM_STATIC_SAMPLERS, 1)
DEFINE_TEXTURE_TYPES_AND_FORMATS_SLOTS(Texture2D, NUM_STATIC_SAMPLERS, 1)
DEFINE_TEXTURE_TYPES_AND_FORMATS_SLOTS(Texture3D, NUM_STATIC_SAMPLERS, 1)
```

Bindless Rendering

BINDLESS HLSL

Emulation layer

01 The emulation struct implements all possible resource types as operator overloads.

02 Each backend implements *DESCRIPTOR_HEAP* and *DESCRIPTOR_HEAP_UNIFORM*.

More detailed info available at our blogpost.

PREDECLARED RESOURCE IDENTIFIERS

```
template <typename T> struct Texture1DHandle { uint internalIndex; };
template <typename T> struct Texture2DHandle { uint internalIndex; };
template <typename T> struct Texture3DHandle { uint internalIndex; };
template <typename T> struct RWTexture1DHandle { uint internalIndex; };
template <typename T> struct RWTexture2DHandle { uint internalIndex; };
template <typename T> struct RWTexture3DHandle { uint internalIndex; };
```

EMULATION STRUCT

```
struct VulkanResourceDescriptorHeapInternal {
    ByteAddressBuffer operator[](ByteBufferHandle identifier) {
        return g_ByteAddressBuffer[NonUniformResourceIndex(identifier.internalIndex)];
    }
```

INTERFACE DECLARATIONS

```
static VulkanResourceDescriptorHeapInternal VkResourceDescriptorHeap;

#define DESCRIPTOR_HEAP(handleType, handle) \
    VkResourceDescriptorHeap[(handleType)handle]
#define DESCRIPTOR_HEAP_UNIFORM(handleType, handle) \
    VkResourceDescriptorHeap[(handleType)handle]
```

Bindless Rendering

BINDLESS HLSL

Emulation layer

01 Resource validation

02 Access emulation heap

03 Return requested value

04 Optionally discard read/write upon validation failure

TEMPLATED BINDLESS USING DESCRIPTOR_HEAP

```
struct Texture {
    RenderResourceHandle handle;

    template < typename TextureValue > TextureValue load1D(uint pos) {
        VALIDATE_RESOURCE_WITH_RETURN_VALUE(kNonWritable, kTextureResourceTag, this.handle, TextureValue);
        Texture1D<TextureValue> texture = DESCRIPTOR_HEAP(Texture1DHandle<TextureValue>, this.handle.readIndex());
        return texture.Load(uint2(pos,0));
    }

    template < typename TextureValue > TextureValue load2D(uint2 pos) {
        VALIDATE_RESOURCE_WITH_RETURN_VALUE(kNonWritable, kTextureResourceTag, this.handle, TextureValue);
        Texture2D<TextureValue> texture = DESCRIPTOR_HEAP(Texture2DHandle<TextureValue>, this.handle.readIndex());
        return texture.Load(uint3(pos,0));
    }
```

RESOURCE VALIDATION

VERSION MISMATCH FAILURE

```
[breda_render_backend_api::shader_logging][ERROR]
Compute Shader GPU resource validation failed:
Resource version mismatch in `gpu_validation` RenderResourceHandle of type `Buffer` has version: `0` Expected version: `1`.
Possible causes:
- A `RenderResourceHandle` was unsafely extracted by the user, where the handle outlived the resource.
- User copied raw `RenderResourceHandle` within a shader to a buffer for later reuse, this is not allowed!
```

RESOURCE TYPE MISMATCH

```
[breda_render_backend_api::shader_logging][ERROR]
Compute Shader GPU resource validation failed:
Resource access mismatch in `gpu_validation` handle is of type: `Texture`, Expected handle of type: `Buffer`.
```

WRITABILITY FAILURE

```
[breda_render_backend_api::shader_logging][ERROR]
Compute Shader GPU resource validation failed:
Tried writing to resource that is read-only in `gpu_validation` RenderResourceHandle has AccessType of: `ReadOnly`.
```

Bindless Rendering

BINDLESS HLSL

Resulting shader code

- Resulting shader code is similar to what Argument buffers achieve, but with more flexibility
- Load any resource from anywhere
- Optional validation that can be enabled per-shader to verify resource usage
- Waiting for *ResourceDescriptorHeap* to remove most of the macro magic

EXAMPLE SHADER CODE

```
#include "breda-render-backend-api::bindless.hlsl"

struct Bindings {
    ArrayBuffer foo;
    RwTexture output;
};

[numthreads(8, 8, 1)] void main(uint2 threadId
                                : SV_DispatchThreadID) {
    Bindings bnd = loadBindings<Bindings>();

    // We can load resources recursively regardless of their type,
    // as long as underlying value is represented in 32 bits.
    ArrayBuffer bar = bnd.foo.load<ArrayBuffer>(0);
    Texture baz = bar.load<Texture>(0);

    float4 sampledValue = baz.load<float4>(threadId);

    // We can also directly store values now.
    bnd.output.store2DUniform<float4>(threadId, sampledValue);
}
```


THANK YOU