vuk

Marcell Kiss

Apr 15, 2023

TOPICS:

CHAPTER

ONE

QUICKSTART

1. Grab the vuk repository
2. Compile the examples
3. Run the example browser and get a feel for the library:

```
git clone http://github.com/martty/vuk
cd vuk
git submodule init
git submodule update --recursive
mkdir build
cd build
mkdir debug
cd debug
cmake ../.. -G Ninja
cmake --build .
./vuk_all_examples
```

(if building with a multi-config generator, do not make the *debug* folder)

1.1 Context

The Context represents the base object of the runtime, encapsulating the knowledge about the GPU (similar to a VkDevice). Use this class to manage pipelines and other cached objects, add/remove swapchains, manage persistent descriptor sets, submit work to device and retrieve query results.

struct **ContextCreateParameters**

Parameters used for creating a *Context*.

Public Members

VkInstance **instance**

Vulkan instance.

VkDevice **device**

Vulkan device.

VkPhysicalDevice **physical_device**

Vulkan physical device.

VkQueue **graphics_queue** = VK_NULL_HANDLE

Optional graphics queue.

uint32_t **graphics_queue_family_index** = VK_QUEUE_FAMILY_IGNORED

Optional graphics queue family index.

VkQueue **compute_queue** = VK_NULL_HANDLE

Optional compute queue.

uint32_t **compute_queue_family_index** = VK_QUEUE_FAMILY_IGNORED

Optional compute queue family index.

VkQueue **transfer_queue** = VK_NULL_HANDLE

Optional transfer queue.

uint32_t **transfer_queue_family_index** = VK_QUEUE_FAMILY_IGNORED

Optional transfer queue family index.

bool **allow_dynamic_loading_of_vk_function_pointers** = true

Allow vuk to load missing required and optional function pointers dynamically If this is false, then you must fill in all required function pointers.

struct **FunctionPointers**

User provided function pointers. If you want dynamic loading, you must set vkGetInstanceProcAddr & vkGetDeviceProcAddr.

Subclassed by *vuk::Context*

class **Context** : public *vuk*::*ContextCreateParameters*::*FunctionPointers*

Public Functions

Context(*ContextCreateParameters* params)

Create a new *Context*.

Parameters
params – Vulkan parameters initialized beforehand

DeviceVkResource &**get_vk_resource**()

Return an allocator over the direct resource - resources will be allocated from the Vulkan runtime.

Returns
The resource

SwapchainRef **add_swapchain**(Swapchain)

Add a swapchain to be managed by the *Context*.

Returns
Reference to the new swapchain that can be used during presentation

void **remove_swapchain**(SwapchainRef)

Remove a swapchain that is managed by the *Context* the swapchain is not destroyed.

void **next_frame**()

Advance internal counter used for caching and garbage collect caches.

void **wait_idle**()

Wait for the device to become idle. Useful for only a few synchronisation events, like resizing or shutting down.

template<class **T**>
Handle<*T*> **wrap**(*T* payload)

Create a wrapped handle type (eg. a ImageView) from an externally sourced Vulkan handle.

Template Parameters
T – Vulkan handle type to wrap

Parameters
payload – Vulkan handle to wrap

Returns
The wrapped handle.

bool **is_timestamp_available**(*Query* q)

Checks if a timestamp query is available.

Parameters
q – the *Query* to check

Returns
true if the timestamp is available

std::optional<uint64_t> **retrieve_timestamp**(*Query* q)

Retrieve a timestamp if available.

Parameters
q – the *Query* to check

Returns
the timestamp value if it was available, null optional otherwise

std::optional<double> **retrieve_duration**(*Query* q1, *Query* q2)

Retrive a duration if available.

Parameters

- **q1** – the start timestamp *Query*
- **q2** – the end timestamp *Query*

Returns
the duration in seconds if both timestamps were available, null optional otherwise

Result<void> **make_timestamp_results_available**(std::span<const TimestampQueryPool> pools)

Retrieve results from `TimestampQueryPools` and make them available to retrieve_timestamp and retrieve_duration.

RGImage **acquire_rendertarget**(const struct RGCI &ci, uint64_t absolute_frame)

Acquire a cached rendertarget.

Sampler **acquire_sampler**(const SamplerCreateInfo &cu, uint64_t absolute_frame)

Acquire a cached sampler.

VkRenderPass **acquire_renderpass**(const struct RenderPassCreateInfo &ci, uint64_t absolute_frame)

Acquire a cached VkRenderPass.

struct PipelineInfo **acquire_pipeline**(const struct PipelineInstanceCreateInfo &ci, uint64_t absolute_frame)

Acquire a cached pipeline.

struct ComputePipelineInfo **acquire_pipeline**(const struct ComputePipelineInstanceCreateInfo &ci, uint64_t absolute_frame)

Acquire a cached compute pipeline.

struct RayTracingPipelineInfo **acquire_pipeline**(const struct RayTracingPipelineInstanceCreateInfo &ci, uint64_t absolute_frame)

Acquire a cached ray tracing pipeline.

struct DescriptorPool &**acquire_descriptor_pool**(const struct DescriptorSetLayoutAllocInfo &dslai, uint64_t absolute_frame)

Acquire a cached descriptor pool.

struct **Query**

Handle to a query result.

1.2 Submitting work

While submitting work to the device can be performed by the user, it is usually sufficient to use a utility function that takes care of translating a RenderGraph into device execution. Note that these functions are used internally when using :cpp:class:`vuk::Future`s, and as such Futures can be used to manage submission in a more high-level fashion.

Result<VkResult> *vuk*::**execute_submit_and_present_to_one**(*Allocator* &allocator, *ExecutableRenderGraph* &&executable_rendergraph, SwapchainRef swapchain)

Execute given *ExecutableRenderGraph* into API VkCommandBuffers, then submit them to queues, presenting to a single swapchain.

Parameters

- **allocator** – Allocator to use for submission resources
- **executable_rendergraph** – *ExecutableRenderGraph*s for execution
- **swapchain** – Swapchain referenced by the rendergraph

Result<void> *vuk*::**execute_submit_and_wait**(*Allocator* &allocator, *ExecutableRenderGraph* &&executable_rendergraph)

Execute given *ExecutableRenderGraph* into API VkCommandBuffers, then submit them to queues, then blocking-wait for the submission to complete.

Parameters

- **allocator** – Allocator to use for submission resources

- **executable_rendergraph** – *ExecutableRenderGraph*s for execution

Result<void> *vuk*::**link_execute_submit**(*Allocator* &allocator, Compiler &compiler, std::span<std::shared_ptr<struct *RenderGraph*>> rendergraphs)

Compile & link given *RenderGraph*s, then execute them into API VkCommandBuffers, then submit them to queues.

Parameters

- **allocator** – Allocator to use for submission resources
- **rendergraphs** – *RenderGraph*s for compilation

1.3 Allocators

Management of GPU resources is an important part of any renderer. vuk provides an API that lets you plug in your allocation schemes, complementing built-in general purpose schemes that get you started and give good performance out of the box.

1.3.1 Overview

class **Allocator**

Interface for allocating device resources.

The Allocator is a concrete value type wrapping over a polymorphic *DeviceResource*, forwarding allocations and deallocations to it. The allocation functions take spans of creation parameters and output values, reporting error through the return value of Result<void, AllocateException>. The deallocation functions can't fail.

struct **DeviceResource**

DeviceResource is a polymorphic interface over allocation of GPU resources. A *DeviceResource* must prevent reuse of cross-device resources after deallocation until CPU-GPU timelines are synchronized. GPU-only resources may be reused immediately.

Subclassed by *vuk::DeviceNestedResource*, *vuk::DeviceVkResource*

To facilitate ownership, a RAII wrapper type is provided, that wraps an Allocator and a payload:

template<typename **Type**>

class **Unique**

1.3.2 Built-in resources

struct **DeviceNestedResource** : public *vuk*::*DeviceResource*

Helper base class for DeviceResources. Forwards all allocations and deallocations to the upstream *DeviceResource*.

Subclassed by *vuk::DeviceFrameResource*, *vuk::DeviceSuperFrameResource*

struct **DeviceVkResource** : public *vuk*::*DeviceResource*

Device resource that performs direct allocation from the resources from the Vulkan runtime.

struct **DeviceFrameResource** : public *vuk*::*DeviceNestedResource*

Represents "per-frame" resources - temporary allocations that persist through a frame. Handed out by *DeviceSuperFrameResource*, cannot be constructed directly.

Allocations from this resource are tied to the "frame" - all allocations recycled when a *DeviceFrameResource* is recycled. Furthermore all resources allocated are also deallocated at recycle time - it is not necessary (but not an error) to deallocate them.

Subclassed by vuk::DeviceMultiFrameResource

struct **DeviceSuperFrameResource** : public *vuk*::*DeviceNestedResource*

DeviceSuperFrameResource is an allocator that gives out *DeviceFrameResource* allocators, and manages their resources.

DeviceSuperFrameResource models resource lifetimes that span multiple frames - these can be allocated directly from this resource Allocation of these resources are persistent, and they can be deallocated at any time - they will be recycled when the current frame is recycled This resource also hands out DeviceFrameResources in a round-robin fashion. The lifetime of resources allocated from those allocators is frames_in_flight number of frames (until the *DeviceFrameResource* is recycled).

1.3.3 Helpers

Allocator provides functions that can perform bulk allocation (to reduce overhead for repeated calls) and return resources directly. However, usually it is more convenient to allocate a single resource and immediately put it into a RAII wrapper to prevent forgetting to deallocate it.

namespace **vuk**

Functions

inline Result<*Unique*<VkSemaphore>, AllocateException> **allocate_semaphore**(*Allocator* &allocator, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate a single semaphore from an Allocator.

Parameters

- **allocator** – Allocator to use
- **loc** – Source location information

Returns
Semaphore in a RAII wrapper (Unique<T>) or AllocateException on error

inline Result<*Unique*<TimelineSemaphore>, AllocateException> **allocate_timeline_semaphore**(*Allocator* &allocator, SourceLocationAtFrame loc = VUK_HERE_AND_NOW

Allocate a single timeline semaphore from an Allocator.

Parameters

- **allocator** – Allocator to use
- **loc** – Source location information

Returns
Timeline semaphore in a RAII wrapper (Unique<T>) or AllocateException on error

inline Result<*Unique*<CommandPool>, AllocateException> **allocate_command_pool**(*Allocator* &allocator, const VkCommandPoolCreateInfo &cpci, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate a single command pool from an Allocator.

Parameters

- **allocator** – Allocator to use
- **cpci** – Command pool creation parameters
- **loc** – Source location information

Returns
Command pool in a RAII wrapper (Unique<T>) or AllocateException on error

inline Result<*Unique*<CommandBufferAllocation>, AllocateException> **allocate_command_buffer**(*Allocator* &allocator, const CommandBufferAllocationCreateInfo &cbci, SourceLocationAtFrame loc = VUK_HERE_AND_NO

Allocate a single command buffer from an Allocator.

Parameters

- **allocator** – Allocator to use
- **cbci** – Command buffer creation parameters

- **loc** – Source location information

Returns
Command buffer in a RAII wrapper (Unique<T>) or AllocateException on error

inline Result<*Unique*<VkFence>, AllocateException> **allocate_fence**(*Allocator* &allocator, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate a single fence from an Allocator.

Parameters

- **allocator** – Allocator to use
- **loc** – Source location information

Returns
Fence in a RAII wrapper (Unique<T>) or AllocateException on error

inline Result<*Unique*<Buffer>, AllocateException> **allocate_buffer**(*Allocator* &allocator, const BufferCreateInfo &bci, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate a single GPU-only buffer from an Allocator.

Parameters

- **allocator** – Allocator to use
- **bci** – Buffer creation parameters
- **loc** – Source location information

Returns
GPU-only buffer in a RAII wrapper (Unique<T>) or AllocateException on error

inline Result<*Unique*<Image>, AllocateException> **allocate_image**(*Allocator* &allocator, const ImageCreateInfo &ici, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate a single image from an Allocator.

Parameters

- **allocator** – Allocator to use
- **ici** – Image creation parameters
- **loc** – Source location information

Returns
Image in a RAII wrapper (Unique<T>) or AllocateException on error

inline Result<*Unique*<Image>, AllocateException> **allocate_image**(*Allocator* &allocator, const ImageAttachment &attachment, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate a single image from an Allocator.

Parameters

- **allocator** – Allocator to use
- **attachment** – ImageAttachment to make the Image from

- **loc** – Source location information

Returns
Image in a RAII wrapper (Unique<T>) or AllocateException on error

inline Result<*Unique*<ImageView>, AllocateException> **allocate_image_view**(*Allocator* &allocator, const ImageViewCreateInfo &ivci, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate a single image view from an Allocator.

Parameters

- **allocator** – Allocator to use
- **ivci** – Image view creation parameters
- **loc** – Source location information

Returns
ImageView in a RAII wrapper (Unique<T>) or AllocateException on error

inline Result<*Unique*<ImageView>, AllocateException> **allocate_image_view**(*Allocator* &allocator, const ImageAttachment &attachment, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate a single image view from an Allocator.

Parameters

- **allocator** – Allocator to use
- **attachment** – ImageAttachment to make the ImageView from
- **loc** – Source location information

Returns
ImageView in a RAII wrapper (Unique<T>) or AllocateException on error

1.3.4 Reference

class **Allocator**

Interface for allocating device resources.

The Allocator is a concrete value type wrapping over a polymorphic *DeviceResource*, forwarding allocations and deallocations to it. The allocation functions take spans of creation parameters and output values, reporting error through the return value of Result<void, AllocateException>. The deallocation functions can't fail.

Public Functions

inline explicit **Allocator**(*DeviceResource* &device_resource)

Create new Allocator that wraps a *DeviceResource*.

Parameters
device_resource – The *DeviceResource* to allocate from

Result<void, AllocateException> **allocate**(std::span<VkSemaphore> dst, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate semaphores from this Allocator.

Parameters

- **dst** – Destination span to place allocated semaphores into
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

Result<void, AllocateException> **allocate_semaphores**(std::span<VkSemaphore> dst, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate semaphores from this Allocator.

Parameters

- **dst** – Destination span to place allocated semaphores into
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

void **deallocate**(std::span<const VkSemaphore> src)

Deallocate semaphores previously allocated from this Allocator.

Parameters
src – Span of semaphores to be deallocated

Result<void, AllocateException> **allocate**(std::span<VkFence> dst, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate fences from this Allocator.

Parameters

- **dst** – Destination span to place allocated fences into
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

Result<void, AllocateException> **allocate_fences**(std::span<VkFence> dst, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate fences from this Allocator.

Parameters

- **dst** – Destination span to place allocated fences into
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

void **deallocate**(std::span<const VkFence> src)

Deallocate fences previously allocated from this Allocator.

Parameters
src – Span of fences to be deallocated

Result<void, AllocateException> **allocate**(std::span<CommandPool> dst, std::span<const VkCommandPoolCreateInfo> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate command pools from this Allocator.

Parameters

- **dst** – Destination span to place allocated command pools into
- **cis** – Per-element construction info
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

Result<void, AllocateException> **allocate_command_pools**(std::span<CommandPool> dst, std::span<const VkCommandPoolCreateInfo> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate command pools from this Allocator.

Parameters

- **dst** – Destination span to place allocated command pools into
- **cis** – Per-element construction info
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

void **deallocate**(std::span<const CommandPool> src)

Deallocate command pools previously allocated from this Allocator.

Parameters
src – Span of command pools to be deallocated

Result<void, AllocateException> **allocate**(std::span<CommandBufferAllocation> dst, std::span<const CommandBufferAllocationCreateInfo> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate command buffers from this Allocator.

Parameters

- **dst** – Destination span to place allocated command buffers into

- **cis** – Per-element construction info
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

Result<void, AllocateException> **allocate_command_buffers**(std::span<CommandBufferAllocation> dst, std::span<const CommandBufferAllocationCreateInfo> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate command buffers from this Allocator.

Parameters

- **dst** – Destination span to place allocated command buffers into
- **cis** – Per-element construction info
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

void **deallocate**(std::span<const CommandBufferAllocation> src)

Deallocate command buffers previously allocated from this Allocator.

Parameters
src – Span of command buffers to be deallocated

Result<void, AllocateException> **allocate**(std::span<Buffer> dst, std::span<const BufferCreateInfo> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate buffers from this Allocator.

Parameters

- **dst** – Destination span to place allocated buffers into
- **cis** – Per-element construction info
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

Result<void, AllocateException> **allocate_buffers**(std::span<Buffer> dst, std::span<const BufferCreateInfo> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate buffers from this Allocator.

Parameters

- **dst** – Destination span to place allocated buffers into
- **cis** – Per-element construction info
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

void **deallocate**(std::span<const Buffer> src)

Deallocate buffers previously allocated from this Allocator.

Parameters
src – Span of buffers to be deallocated

Result<void, AllocateException> **allocate**(std::span<VkFramebuffer> dst, std::span<const FramebufferCreateInfo> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate framebuffers from this Allocator.

Parameters

- **dst** – Destination span to place allocated framebuffers into
- **cis** – Per-element construction info
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

Result<void, AllocateException> **allocate_framebuffers**(std::span<VkFramebuffer> dst, std::span<const FramebufferCreateInfo> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate framebuffers from this Allocator.

Parameters

- **dst** – Destination span to place allocated framebuffers into
- **cis** – Per-element construction info
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

void **deallocate**(std::span<const VkFramebuffer> src)

Deallocate framebuffers previously allocated from this Allocator.

Parameters
src – Span of framebuffers to be deallocated

Result<void, AllocateException> **allocate**(std::span<Image> dst, std::span<const ImageCreateInfo> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate images from this Allocator.

Parameters

- **dst** – Destination span to place allocated images into
- **cis** – Per-element construction info
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

Result<void, AllocateException> **allocate_images**(std::span<Image> dst, std::span<const ImageCreateInfo> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate images from this Allocator.

Parameters

- **dst** – Destination span to place allocated images into
- **cis** – Per-element construction info
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

void **deallocate**(std::span<const Image> src)

Deallocate images previously allocated from this Allocator.

Parameters
src – Span of images to be deallocated

Result<void, AllocateException> **allocate**(std::span<ImageView> dst, std::span<const ImageViewCreateInfo> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate image views from this Allocator.

Parameters

- **dst** – Destination span to place allocated image views into
- **cis** – Per-element construction info
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

Result<void, AllocateException> **allocate_image_views**(std::span<ImageView> dst, std::span<const ImageViewCreateInfo> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate image views from this Allocator.

Parameters

- **dst** – Destination span to place allocated image views into
- **cis** – Per-element construction info
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

void **deallocate**(std::span<const ImageView> src)

Deallocate image views previously allocated from this Allocator.

Parameters
src – Span of image views to be deallocated

Result<void, AllocateException> **allocate**(std::span<PersistentDescriptorSet> dst, std::span<const PersistentDescriptorSetCreateInfo> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate persistent descriptor sets from this Allocator.

Parameters

- **dst** – Destination span to place allocated persistent descriptor sets into
- **cis** – Per-element construction info
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

Result<void, AllocateException> **allocate_persistent_descriptor_sets**(std::span<PersistentDescriptorSet> dst, std::span<const PersistentDescriptorSetCreateInfo> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate persistent descriptor sets from this Allocator.

Parameters

- **dst** – Destination span to place allocated persistent descriptor sets into
- **cis** – Per-element construction info
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

void **deallocate**(std::span<const PersistentDescriptorSet> src)

Deallocate persistent descriptor sets previously allocated from this Allocator.

Parameters
src – Span of persistent descriptor sets to be deallocated

Result<void, AllocateException> **allocate**(std::span<DescriptorSet> dst, std::span<const SetBinding> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate descriptor sets from this Allocator.

Parameters

- **dst** – Destination span to place allocated descriptor sets into
- **cis** – Per-element construction info
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

Result<void, AllocateException> **allocate_descriptor_sets_with_value**(std::span<DescriptorSet> dst, std::span<const SetBinding> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate descriptor sets from this Allocator.

Parameters

- **dst** – Destination span to place allocated descriptor sets into
- **cis** – Per-element construction info
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

Result<void, AllocateException> **allocate**(std::span<DescriptorSet> dst, std::span<const DescriptorSetLayoutAllocInfo> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate descriptor sets from this Allocator.

Parameters

- **dst** – Destination span to place allocated descriptor sets into
- **cis** – Per-element construction info
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

Result<void, AllocateException> **allocate_descriptor_sets**(std::span<DescriptorSet> dst, std::span<const DescriptorSetLayoutAllocInfo> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate descriptor sets from this Allocator.

Parameters

- **dst** – Destination span to place allocated descriptor sets into
- **cis** – Per-element construction info
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

void **deallocate**(std::span<const DescriptorSet> src)

Deallocate descriptor sets previously allocated from this Allocator.

Parameters
src – Span of descriptor sets to be deallocated

Result<void, AllocateException> **allocate**(std::span<TimestampQueryPool> dst, std::span<const VkQueryPoolCreateInfo> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate timestamp query pools from this Allocator.

Parameters

- **dst** – Destination span to place allocated timestamp query pools into
- **cis** – Per-element construction info
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

Result<void, AllocateException> **allocate_timestamp_query_pools**(std::span<TimestampQueryPool> dst, std::span<const VkQueryPoolCreateInfo> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate timestamp query pools from this Allocator.

Parameters

- **dst** – Destination span to place allocated timestamp query pools into
- **cis** – Per-element construction info
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

void **deallocate**(std::span<const TimestampQueryPool> src)

Deallocate timestamp query pools previously allocated from this Allocator.

Parameters
src – Span of timestamp query pools to be deallocated

Result<void, AllocateException> **allocate**(std::span<TimestampQuery> dst, std::span<const TimestampQueryCreateInfo> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate timestamp queries from this Allocator.

Parameters

- **dst** – Destination span to place allocated timestamp queries into
- **cis** – Per-element construction info
- **loc** – Source location information

Returns
Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

Result<void, AllocateException> **allocate_timestamp_queries**(std::span<TimestampQuery> dst, std::span<const TimestampQueryCreateInfo> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate timestamp queries from this Allocator.

Parameters

- **dst** – Destination span to place allocated timestamp queries into
- **cis** – Per-element construction info
- **loc** – Source location information

Returns

Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

void **deallocate**(std::span<const TimestampQuery> src)

Deallocate timestamp queries previously allocated from this Allocator.

Parameters

src – Span of timestamp queries to be deallocated

Result<void, AllocateException> **allocate**(std::span<TimelineSemaphore> dst, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate timeline semaphores from this Allocator.

Parameters

- **dst** – Destination span to place allocated timeline semaphores into
- **loc** – Source location information

Returns

Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

Result<void, AllocateException> **allocate_timeline_semaphores**(std::span<TimelineSemaphore> dst, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate timeline semaphores from this Allocator.

Parameters

- **dst** – Destination span to place allocated timeline semaphores into
- **loc** – Source location information

Returns

Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

void **deallocate**(std::span<const TimelineSemaphore> src)

Deallocate timeline semaphores previously allocated from this Allocator.

Parameters

src – Span of timeline semaphores to be deallocated

Result<void, AllocateException> **allocate**(std::span<VkAccelerationStructureKHR> dst, std::span<const VkAccelerationStructureCreateInfoKHR> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate acceleration structures from this Allocator.

Parameters

- **dst** – Destination span to place allocated acceleration structures into
- **loc** – Source location information

Returns

Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

Result<void, AllocateException> **allocate_acceleration_structures**(std::span<VkAccelerationStructureKHR> dst, std::span<const VkAccelerationStructureCreateInfoKHR> cis, SourceLocationAtFrame loc = VUK_HERE_AND_NOW())

Allocate acceleration structures from this Allocator.

Parameters

- **dst** – Destination span to place allocated acceleration structures into
- **loc** – Source location information

Returns

Result<void, AllocateException> : void or AllocateException if the allocation could not be performed.

void **deallocate**(std::span<const VkAccelerationStructureKHR> src)

Deallocate acceleration structures previously allocated from this Allocator.

Parameters

src – Span of acceleration structures to be deallocated

void **deallocate**(std::span<const VkSwapchainKHR> src)

Deallocate swapchains previously allocated from this Allocator.

Parameters

src – Span of swapchains to be deallocated

inline *DeviceResource* &**get_device_resource**()

Get the underlying *DeviceResource*.

Returns

the underlying *DeviceResource*

inline *Context* &**get_context**()

Get the parent *Context*.

Returns

the parent *Context*

1.4 Rendergraph

struct **Resource**

struct **RenderGraph** : public std::enable_shared_from_this<*RenderGraph*>

Public Functions

void **add_pass**(Pass pass, source_location location = source_location::current())

Add a pass to the rendergraph.

Parameters

pass – the Pass to add to the *RenderGraph*

void **add_alias**(Name new_name, Name old_name)

Add an alias for a resource.

Parameters

- **new_name** – Additional name to refer to the resource
- **old_name** – Old name used to refere to the resource

void **diverge_image**(Name whole_name, Subrange::Image subrange, Name subrange_name)

Diverge image. subrange is available as subrange_name afterwards.

void **converge_image_explicit**(std::span<Name> pre_diverge, Name post_diverge)

Reconverge image from named parts. Prevents diverged use moving before pre_diverge or after post_diverge.

void **resolve_resource_into**(Name resolved_name_src, Name resolved_name_dst, Name ms_name)

Add a resolve operation from the image resource `ms_name` that consumes `resolved_name_src` and produces `resolved_name_dst` This is only supported for color images.

Parameters

- **resolved_name_src** – Image resource name consumed (single-sampled)
- **resolved_name_dst** – Image resource name created (single-sampled)
- **ms_name** – Image resource to resolve from (multisampled)

void **clear_image**(Name image_name_in, Name image_name_out, Clear clear_value)

Clear image attachment.

Parameters

- **image_name_in** – Name of the image resource to clear
- **image_name_out** – Name of the cleared image resource
- **clear_value** – Value used for the clear
- **subrange** – Range of image cleared

void **attach_swapchain**(Name name, SwapchainRef swp)

Attach a swapchain to the given name.

Parameters

name – Name of the resource to attach to

void **attach_buffer**(Name name, Buffer buffer, Access initial = eNone)

Attach a buffer to the given name.

Parameters

- **name** – Name of the resource to attach to
- **buffer** – Buffer to attach
- **initial** – Access to the resource prior to this rendergraph

void **attach_buffer_from_allocator**(Name name, Buffer buffer, *Allocator* allocator, Access initial = eNone)

Attach a buffer to be allocated from the specified allocator.

Parameters

- **name** – Name of the resource to attach to
- **buffer** – Buffer to attach
- **allocator** – Allocator the Buffer will be allocated from
- **initial** – Access to the resource prior to this rendergraph

void **attach_image**(Name name, ImageAttachment image_attachment, Access initial = eNone)

Attach an image to the given name.

Parameters

- **name** – Name of the resource to attach to
- **image_attachment** – ImageAttachment to attach
- **initial** – Access to the resource prior to this rendergraph

void **attach_image_from_allocator**(Name name, ImageAttachment image_attachment, *Allocator* allocator, Access initial = eNone)

Attach an image to be allocated from the specified allocator.

Parameters

- **name** – Name of the resource to attach to
- **image_attachment** – ImageAttachment to attach
- **buffer** – Buffer to attach
- **initial** – Access to the resource prior to this rendergraph

void **attach_and_clear_image**(Name name, ImageAttachment image_attachment, Clear clear_value, Access initial = eNone)

Attach an image to the given name.

Parameters

- **name** – Name of the resource to attach to
- **image_attachment** – ImageAttachment to attach
- **clear_value** – Value used for the clear
- **initial** – Access to the resource prior to this rendergraph

void **attach_in**(Name name, *Future* future)

Attach a future to the given name.

Parameters

- **name** – Name of the resource to attach to
- **future** – *Future* to be attached into this rendergraph

void **attach_in**(std::span<*Future*> futures)

Attach multiple futures - the names are matched to future bound names.

Parameters

futures – Futures to be attached into this rendergraph

std::vector<*Future*> **split**()

Compute all the unconsumed resource names and return them as Futures.

void **release**(Name name, Access final)

Mark resources to be released from the rendergraph with future access.

Parameters

- **name** – Name of the resource to be released
- **final** – Access after the rendergraph

void **release_for_present**(Name name)

Mark resource to be released from the rendergraph for presentation.

Parameters

name – Name of the resource to be released

Public Members

Name **name**

Name of the rendergraph.

struct **ExecutableRenderGraph**

1.5 Futures

vuk Futures allow you to reason about computation of resources that happened in the past, or will happen in the future. In general the limitation of RenderGraphs are that they don't know the state of the resources produces by previous computation, or the state the resources should be left in for future computation, so these states must be provided manually (this is error-prone). Instead you can encapsulate the computation and its result into a Future, which can then serve as an input to other RenderGraphs.

Futures can be constructed from a RenderGraph and a named Resource that is considered to be the output. A Future can optionally own the RenderGraph - but in all cases a Future must outlive the RenderGraph it references.

You can submit Futures manually, which will compile, execute and submit the RenderGraph it references. In this case when you use this Future as input to another RenderGraph it will wait for the result on the device. If a Future has not yet been submitted, the contained RenderGraph is simply appended as a subgraph (i.e. inlined).

It is also possible to wait for the result to be produced to be available on the host - but this forces a CPU-GPU sync and should be used sparingly.

class **Future**

Public Functions

Future(std::shared_ptr<*RenderGraph*> rg, Name output_binding, DomainFlags dst_domain = DomainFlagBits::eDevice)

Create a *Future* with ownership of a *RenderGraph* and bind to an output.

Parameters

- **rg** –
- **output_binding** –

Future(std::shared_ptr<*RenderGraph*> rg, QualifiedName output_binding, DomainFlags dst_domain = DomainFlagBits::eDevice)

Create a *Future* with ownership of a *RenderGraph* and bind to an output.

Parameters

- **rg** –
- **output_binding** –

inline **Future**(ImageAttachment value)

Create a *Future* from a value, automatically making the result available on the host.

Parameters
value –

inline **Future**(Buffer value)

Create a *Future* from a value, automatically making the result available on the host.

Parameters
value –

inline FutureBase::Status &**get_status**()

Get status of the *Future*.

inline std::shared_ptr<*RenderGraph*> **get_render_graph**()

Get the referenced *RenderGraph*.

Result<void> **submit**(*Allocator* &allocator, Compiler &compiler)

Submit *Future* for execution.

Result<void> **wait**(*Allocator* &allocator, Compiler &compiler)

Wait for *Future* to complete execution on host.

template<class **T**>
Result<*T*> **get**(*Allocator* &allocator, Compiler &compiler)

Wait and retrieve the result of the *Future* on the host.

inline FutureBase *__get_control__()

Get control block for *Future*.

1.6 Composing render graphs

Futures make easy to compose complex operations and effects out of RenderGraph building blocks, linked by Futures. These building blocks are termed partials, and vuk provides some built-in. Such partials are functions that take a number of Futures as input, and produce a Future as output.

The built-in partials can be found below. Built on these, there are some convenience functions that couple resource allocation with initial data (*create_XXX()*).

namespace **vuk**

Functions

inline *Future* **host_data_to_buffer**(*Allocator* &allocator, DomainFlagBits copy_domain, Buffer dst, const void *src_data, size_t size)

Fill a buffer with host data.

Parameters

- **allocator** – Allocator to use for temporary allocations
- **copy_domain** – The domain where the copy should happen (when dst is mapped, the copy happens on host)
- **buffer** – Buffer to fill
- **src_data** – pointer to source data
- **size** – size of source data

template<class **T**>
Future **host_data_to_buffer**(*Allocator* &allocator, DomainFlagBits copy_domain, Buffer dst, std::span<*T*> data)

Fill a buffer with host data.

Parameters

- **allocator** – Allocator to use for temporary allocations
- **copy_domain** – The domain where the copy should happen (when dst is mapped, the copy happens on host)
- **dst** – Buffer to fill
- **data** – source data

inline *Future* **download_buffer**(*Future* buffer_src)

Download a buffer to GPUtoCPU memory.

Parameters
buffer_src – Buffer to download

inline *Future* **host_data_to_image**(*Allocator* &allocator, DomainFlagBits copy_domain, ImageAttachment image, const void *src_data)

Fill an image with host data.

Parameters

- **allocator** – Allocator to use for temporary allocations

- **copy_domain** – The domain where the copy should happen (when dst is mapped, the copy happens on host)
- **image** – ImageAttachment to fill
- **src_data** – pointer to source data

inline *Future* **transition**(*Future* image, Access dst_access)

Transition image for given access - useful to force certain access across different RenderGraphs linked by Futures.

Parameters

- **image** – input *Future* of ImageAttachment
- **dst_access** – Access to have in the future

inline *Future* **generate_mips**(*Future* image, uint32_t base_mip, uint32_t num_mips)

Generate mips for given ImageAttachment.

Parameters

- **image** – input *Future* of ImageAttachment
- **base_mip** – source mip level
- **num_mips** – number of mip levels to generate

template<class **T**>
std::pair<*Unique*<Buffer>, *Future*> **create_buffer**(*Allocator* &allocator, *vuk*::MemoryUsage memory_usage, DomainFlagBits domain, std::span<*T*> data)

Allocates & fills a buffer with explicitly managed lifetime.

Parameters

- **allocator** – Allocator to allocate this Buffer from
- **mem_usage** – Where to allocate the buffer (host visible buffers will be automatically mapped)

inline std::pair<Texture, *Future*> **create_texture**(*Allocator* &allocator, Format format, Extent3D extent, void *data, bool should_generate_mips)

Allocates & fills an image, creates default ImageView for it (legacy)

Parameters

- **allocator** – Allocator to allocate this Texture from
- **format** – Format of the image
- **extent** – Extent3D of the image
- **data** – pointer to data to fill the image with
- **should_generate_mips** – if true, all mip levels are generated from the 0th level

1.7 CommandBuffer

The CommandBuffer class offers a convenient abstraction over command recording, pipeline state and descriptor sets of Vulkan.

1.7.1 Setting pipeline state

The CommandBuffer encapsulates the current pipeline and descriptor state. When calling state-setting commands, the current state of the CommandBuffer is updated. The state of the CommandBuffer persists for the duration of the execution callback, and there is no state leakage between callbacks of different passes.

The various states of the pipeline can be reconfigured by calling the appropriate function, such as *vuk::CommandBuffer::set_rasterization()*.

There is no default state - you must explicitly bind all state used for the commands recorded.

1.7.2 Static and dynamic state

Vulkan allows some pipeline state to be dynamic. In vuk this is exposed as an optimisation - you may let the CommandBuffer know that certain pipeline state is dynamic by calling *vuk::CommandBuffer::set_dynamic_state()*. This call changes which states are considered dynamic. Dynamic state is usually cheaper to change than entire pipelines and leads to fewer pipeline compilations, but has more overhead compared to static state - use it when a state changes often. Some state can be set dynamic on some platforms without cost. As with other pipeline state, setting states to be dynamic or static persist only during the callback.

1.7.3 Binding pipelines & specialization constants

The CommandBuffer maintains separate bind points for compute and graphics pipelines. The CommandBuffer also maintains an internal buffer of specialization constants that are applied to the pipeline bound. Changing specialization constants will trigger a pipeline compilation when using the pipeline for the first time.

1.7.4 Binding descriptors & push constants

vuk allows two types of descriptors to be bound: ephemeral and persistent.

Ephemeral descriptors are bound individually to the CommandBuffer via *bind_XXX()* calls where *XXX* denotes the type of the descriptor (eg. uniform buffer). These descriptors are internally managed by the CommandBuffer and the Allocator it references. Ephemeral descriptors are very convenient to use, but they are limited in the number of bindable descriptors (*VUK_MAX_BINDINGS*) and they incur a small overhead on bind.

Persistent descriptors are managed by the user via allocation of a PersistentDescriptorSet from Allocator and manually updating the contents. There is no limit on the number of descriptors and binding such descriptor sets do not have an overhead over the direct Vulkan call. Large descriptor arrays (such as the ones used in “bindless” techniques) are only possible via persistent descriptor sets.

The number of bindable sets is limited by *VUK_MAX_SETS*. Both ephemeral descriptors and persistent descriptor sets retain their bindings until overwritten, disturbed or the the callback ends.

Push constants can be changed by calling *vuk::CommandBuffer::push_constants()*.

1.7.5 Vertex buffers and attributes

While vertex buffers are waning in popularity, vuk still offers a convenient API for most attribute arrangements. If advanced addressing schemes are not required, they can be a convenient alternative to vertex pulling.

The shader declares attributes, which require a *location*. When binding vertex buffers, you are telling vuk where each attribute, corresponding to a *location* can be found. Each *vuk::CommandBuffer::bind_vertex_buffer()* binds a single vuk::Buffer, which can contain multiple attributes

The first two arguments to *vuk::CommandBuffer::bind_vertex_buffer()* specify the index of the vertex buffer binding and buffer to binding to that binding. (so if you have 1 vertex buffers, you pass 0, if you have 2 vertex buffers, you have 2 calls where you pass 0 and 1 as *binding* - these don't need to start at 0 or be contiguous but they might as well be)

In the second part of the arguments you specify which attributes can be found the vertex buffer that is being bound, what is their format, and what is their offset. For convenience vuk offers a utility called *vuk::Packed* to describe common vertex buffers that contain interleaved attribute data.

The simplest case is a single attribute per vertex buffer, this is described by calling *bind_vertex_buffer(binding, buffer, location, vuk::Packed{ vuk::Format::eR32G32B32Sfloat })* - with the actual format of the attribute. Here *vuk::Packed* means that the formats are packed in the buffer, i.e. you have a R32G32B32, then immediately after a R32G32B32, and so on.

If there are multiple interleaved attributes in a buffer, for example it is [position, normal, position, normal], then you can describe this in a very compact way in vuk if the position attribute location and normal attribute location is consecutive: *bind_vertex_buffer(binding, buffer, first_location, vuk::Packed{ vuk::Format::eR32G32B32Sfloat, vuk::Format::eR32G32B32Sfloat })*. Finally, you can describe holes in your interleaving by using *vuk::Ignore(byte_size)* in the format list for *vuk::Packed*.

If your attribute scheme cannot be described like this, you can also use *vuk::CommandBuffer::bind_vertex_buffer()* with a manually built *span<VertexInputAttributeDescription>* and computed stride.

1.7.6 Command recording

Draws and dispatches can be recorded by calling the appropriate function. Any state changes made will be recorded into the underlying Vulkan command buffer, along with the draw or dispatch.

1.7.7 Error handling

The CommandBuffer implements "monadic" error handling, because operations that allocate resources might fail. In this case the CommandBuffer is moved into the error state and subsequent calls do not modify the underlying state.

class **CommandBuffer**

Public Functions

inline *Context* &**get_context**()

Retrieve parent context.

const RenderPassInfo &**get_ongoing_renderpass**() const

Retrieve information about the current renderpass.

Result<Buffer> **get_resource_buffer**(Name resource_name) const

Retrieve Buffer attached to given name.

Returns
the attached Buffer or RenderGraphException

Result<Image> **get_resource_image**(Name resource_name) const

Retrieve Image attached to given name.

Returns
the attached Image or RenderGraphException

Result<ImageView> **get_resource_image_view**(Name resource_name) const

Retrieve ImageView attached to given name.

Returns
the attached ImageView or RenderGraphException

Result<ImageAttachment> **get_resource_image_attachment**(Name resource_name) const

Retrieve ImageAttachment attached to given name.

Returns
the attached ImageAttachment or RenderGraphException

CommandBuffer &**set_descriptor_set_strategy**(DescriptorSetStrategyFlags ds_strategy_flags)

Set the strategy for allocating and updating ephemeral descriptor sets.

The default strategy is taken from the context when entering a new Pass

Parameters
ds_strategy_flags – Mask of strategy options

CommandBuffer &**set_dynamic_state**(DynamicStateFlags dynamic_state_flags)

Set mask of dynamic state in *CommandBuffer*.

Parameters
dynamic_state_flags – Mask of states (flag set = dynamic, flag clear = static)

CommandBuffer &**set_viewport**(unsigned index, Viewport vp)

Set the viewport transformation for the specified viewport index.

Parameters

- **index** – viewport index to modify
- **vp** – Viewport to be set

CommandBuffer &**set_viewport**(unsigned index, Rect2D area, float min_depth = 0.f, float max_depth = 1.f)

Set the viewport transformation for the specified viewport index from a rect.

Parameters

- **index** – viewport index to modify

- **area** – Rect2D extents of the Viewport
- **min_depth** – Minimum depth of Viewport
- **max_depth** – Maximum depth of Viewport

CommandBuffer &**set_scissor**(unsigned index, Rect2D area)

Set the scissor for the specified scissor index from a rect.

Parameters

- **index** – scissor index to modify
- **area** – Rect2D extents of the scissor

CommandBuffer &**set_rasterization**(PipelineRasterizationStateCreateInfo rasterization_state)

Set the rasterization state.

CommandBuffer &**set_depth_stencil**(PipelineDepthStencilStateCreateInfo depth_stencil_state)

Set the depth/stencil state.

CommandBuffer &**set_conservative**(PipelineRasterizationConservativeStateCreateInfo conservative_state)

Set the conservative rasterization state.

CommandBuffer &**broadcast_color_blend**(PipelineColorBlendAttachmentState color_blend_state)

Set one color blend state to use for all color attachments.

CommandBuffer &**broadcast_color_blend**(BlendPreset blend_preset)

Set one color blend preset to use for all color attachments.

CommandBuffer &**set_color_blend**(Name color_attachment, PipelineColorBlendAttachmentState color_blend_state)

Set color blend state for a specific color attachment.

Parameters

- **color_attachment** – the Name of the color_attachment to set the blend state for
- **color_blend_state** – PipelineColorBlendAttachmentState to use

CommandBuffer &**set_color_blend**(Name color_attachment, BlendPreset blend_preset)

Set color blend preset for a specific color attachment.

Parameters

- **color_attachment** – the Name of the color_attachment to set the blend preset for
- **blend_preset** – BlendPreset to use

CommandBuffer &**set_blend_constants**(std::array<float, 4> blend_constants)

Set blend constants.

CommandBuffer &**bind_graphics_pipeline**(PipelineBaseInfo *pipeline_base)

Bind a graphics pipeline for subsequent draws.

Parameters
pipeline_base – pointer to a pipeline base to bind

CommandBuffer &**bind_graphics_pipeline**(Name named_pipeline)

Bind a named graphics pipeline for subsequent draws.

Parameters
named_pipeline – graphics pipeline name

CommandBuffer &**bind_compute_pipeline**(PipelineBaseInfo *pipeline_base)

Bind a compute pipeline for subsequent dispatches.

Parameters
pipeline_base – pointer to a pipeline base to bind

CommandBuffer &**bind_compute_pipeline**(Name named_pipeline)

Bind a named graphics pipeline for subsequent dispatches.

Parameters
named_pipeline – compute pipeline name

CommandBuffer &**bind_ray_tracing_pipeline**(PipelineBaseInfo *pipeline_base)

Bind a ray tracing pipeline for subsequent draws.

Parameters
pipeline_base – pointer to a pipeline base to bind

CommandBuffer &**bind_ray_tracing_pipeline**(Name named_pipeline)

Bind a named ray tracing pipeline for subsequent draws.

Parameters
named_pipeline – graphics pipeline name

inline *CommandBuffer* &**specialize_constants**(uint32_t constant_id, bool value)

Set specialization constants for the command buffer.

Parameters

- **constant_id** – ID of the constant. All stages form a single namespace for IDs.
- **value** – Value of the specialization constant

inline *CommandBuffer* &**specialize_constants**(uint32_t constant_id, uint32_t value)

Set specialization constants for the command buffer.

Parameters

- **constant_id** – ID of the constant. All stages form a single namespace for IDs.
- **value** – Value of the specialization constant

inline *CommandBuffer* &**specialize_constants**(uint32_t constant_id, int32_t value)

Set specialization constants for the command buffer.

Parameters

- **constant_id** – ID of the constant. All stages form a single namespace for IDs.
- **value** – Value of the specialization constant

inline *CommandBuffer* &**specialize_constants**(uint32_t constant_id, float value)

Set specialization constants for the command buffer.

Parameters

- **constant_id** – ID of the constant. All stages form a single namespace for IDs.
- **value** – Value of the specialization constant

inline *CommandBuffer* &**specialize_constants**(uint32_t constant_id, double value)

Set specialization constants for the command buffer.

Parameters

- **constant_id** – ID of the constant. All stages form a single namespace for IDs.
- **value** – Value of the specialization constant

CommandBuffer &**set_primitive_topology**(PrimitiveTopology primitive_topology)

Set primitive topology.

CommandBuffer &**bind_index_buffer**(const Buffer &buffer, IndexType type)

Binds an index buffer with the given type.

Parameters

- **buffer** – The buffer to be bound
- **type** – The index type in the buffer

CommandBuffer &**bind_index_buffer**(Name resource_name, IndexType type)

Binds an index buffer from a *Resource* with the given type.

Parameters

- **resource_name** – The Name of the *Resource* to be bound
- **type** – The index type in the buffer

CommandBuffer &**bind_vertex_buffer**(unsigned binding, const Buffer &buffer, unsigned first_location, Packed format_list)

Binds a vertex buffer to the given binding point and configures attributes sourced from this buffer based on a packed format list, the attribute locations are offset with first_location.

Parameters

- **binding** – The binding point of the buffer
- **buffer** – The buffer to be bound
- **first_location** – First location assigned to the attributes
- **format_list** – List of formats packed in buffer to generate attributes from

CommandBuffer &**bind_vertex_buffer**(unsigned binding, Name resource_name, unsigned first_location, Packed format_list)

Binds a vertex buffer from a *Resource* to the given binding point and configures attributes sourced from this buffer based on a packed format list, the attribute locations are offset with first_location.

Parameters

- **binding** – The binding point of the buffer
- **resource_name** – The Name of the *Resource* to be bound
- **first_location** – First location assigned to the attributes
- **format_list** – List of formats packed in buffer to generate attributes from

CommandBuffer &**bind_vertex_buffer**(unsigned binding, const Buffer &buffer, std::span<VertexInputAttributeDescription> attribute_descriptions, uint32_t stride)

Binds a vertex buffer to the given binding point and configures attributes sourced from this buffer based on a span of attribute descriptions and stride.

Parameters

- **binding** – The binding point of the buffer
- **buffer** – The buffer to be bound
- **attribute_descriptions** – Attributes that are sourced from this buffer
- **stride** – Stride of a vertex sourced from this buffer

CommandBuffer &**bind_vertex_buffer**(unsigned binding, Name resource_name, std::span<VertexInputAttributeDescription> attribute_descriptions, uint32_t stride)

Binds a vertex buffer from a *Resource* to the given binding point and configures attributes sourced from this buffer based on a span of attribute descriptions and stride.

Parameters

- **binding** – The binding point of the buffer
- **resource_name** – The Name of the *Resource* to be bound
- **attribute_descriptions** – Attributes that are sourced from this buffer
- **stride** – Stride of a vertex sourced from this buffer

CommandBuffer &**push_constants**(ShaderStageFlags stages, size_t offset, void *data, size_t size)

Update push constants for the specified stages with bytes.

Parameters

- **stages** – Pipeline stages that can see the updated bytes
- **offset** – Offset into the push constant buffer
- **data** – Pointer to data to be copied into push constants
- **size** – Size of data

template<class **T**>
inline *CommandBuffer* &**push_constants**(ShaderStageFlags stages, size_t offset, std::span<*T*> span)

Update push constants for the specified stages with a span of values.

Template Parameters
T – type of values

Parameters

- **stages** – Pipeline stages that can see the updated bytes
- **offset** – Offset into the push constant buffer
- **span** – Values to write

template<class **T**>
inline *CommandBuffer* &**push_constants**(ShaderStageFlags stages, size_t offset, *T* value)

Update push constants for the specified stages with a single value.

Template Parameters
T – type of value

Parameters

- **stages** – Pipeline stages that can see the updated bytes
- **offset** – Offset into the push constant buffer

- **value** – Value to write

CommandBuffer &**bind_persistent**(unsigned set, PersistentDescriptorSet &desc_set)

Bind a persistent descriptor set to the command buffer.

Parameters

- **set** – The set bind index to be used
- **desc_set** – The persistent descriptor set to be bound

CommandBuffer &**bind_buffer**(unsigned set, unsigned binding, const Buffer &buffer)

Bind a buffer to the command buffer.

Parameters

- **set** – The set bind index to be used
- **binding** – The descriptor binding to bind the buffer to
- **buffer** – The buffer to be bound

CommandBuffer &**bind_buffer**(unsigned set, unsigned binding, Name resource_name)

Bind a buffer to the command buffer from a *Resource*.

Parameters

- **set** – The set bind index to be used
- **binding** – The descriptor binding to bind the buffer to
- **resource_name** – The Name of the *Resource* to be bound

CommandBuffer &**bind_image**(unsigned set, unsigned binding, ImageView image_view, ImageLayout layout = ImageLayout::eReadOnlyOptimalKHR)

Bind an image to the command buffer.

Parameters

- **set** – The set bind index to be used
- **binding** – The descriptor binding to bind the image to
- **image_view** – The ImageView to bind
- **layout** – layout of the image when the affected draws execute

CommandBuffer &**bind_image**(unsigned set, unsigned binding, const ImageAttachment &image, ImageLayout layout = ImageLayout::eReadOnlyOptimalKHR)

Bind an image to the command buffer.

Parameters

- **set** – The set bind index to be used
- **binding** – The descriptor binding to bind the image to
- **image_view** – The ImageAttachment to bind
- **layout** – layout of the image when the affected draws execute

CommandBuffer &**bind_image**(unsigned set, unsigned binding, Name resource_name)

Bind an image to the command buffer from a *Resource*.

Parameters

- **set** – The set bind index to be used

- **binding** – The descriptor binding to bind the image to
- **resource_name** – The Name of the *Resource* to be bound

CommandBuffer &**bind_sampler**(unsigned set, unsigned binding, SamplerCreateInfo sampler_create_info)

Bind a sampler to the command buffer from a *Resource*.

Parameters

- **set** – The set bind index to be used
- **binding** – The descriptor binding to bind the sampler to
- **sampler_create_info** – Parameters of the sampler

void ***_map_scratch_buffer**(unsigned set, unsigned binding, size_t size)

Allocate some CPUtoGPU memory and bind it as a buffer. Return a pointer to the mapped memory.

Parameters

- **set** – The set bind index to be used
- **binding** – The descriptor binding to bind the buffer to
- **size** – Amount of memory to allocate

Returns

pointer to the mapped host-visible memory. Null pointer if the command buffer has errored out previously or the allocation failed

template<class **T**>
inline *T* ***map_scratch_buffer**(unsigned set, unsigned binding)

Allocate some typed CPUtoGPU memory and bind it as a buffer. Return a pointer to the mapped memory.

Template Parameters

T – Type of the uniform to write

Parameters

- **set** – The set bind index to be used
- **binding** – The descriptor binding to bind the buffer to

Returns

pointer to the mapped host-visible memory. Null pointer if the command buffer has errored out previously or the allocation failed

CommandBuffer &**bind_acceleration_structure**(unsigned set, unsigned binding, VkAccelerationStructureKHR tlas)

Bind a sampler to the command buffer from a *Resource*.

Parameters

- **set** – The set bind index to be used
- **binding** – The descriptor binding to bind the sampler to
- **sampler_create_info** – Parameters of the sampler

CommandBuffer &**draw**(size_t vertex_count, size_t instance_count, size_t first_vertex, size_t first_instance)

Issue a non-indexed draw.

Parameters

- **vertex_count** – Number of vertices to draw

- **instance_count** – Number of instances to draw
- **first_vertex** – Index of the first vertex to draw
- **first_instance** – Index of the first instance to draw

CommandBuffer &**draw_indexed**(size_t index_count, size_t instance_count, size_t first_index, int32_t vertex_offset, size_t first_instance)

Isuse an indexed draw.

Parameters

- **index_count** – Number of vertices to draw
- **instance_count** – Number of instances to draw
- **first_index** – Index of the first index in the index buffer
- **vertex_offset** – value added to the vertex index before indexing into the vertex buffer(s)
- **first_instance** – Index of the first instance to draw

CommandBuffer &**draw_indexed_indirect**(size_t command_count, const Buffer &indirect_buffer)

Issue an indirect indexed draw.

Parameters

- **command_count** – Number of indirect commands to be used
- **indirect_buffer** – Buffer of indirect commands

CommandBuffer &**draw_indexed_indirect**(size_t command_count, Name indirect_resource_name)

Issue an indirect indexed draw.

Parameters

- **command_count** – Number of indirect commands to be used
- **indirect_resource_name** – The Name of the *Resource* to use as indirect buffer

CommandBuffer &**draw_indexed_indirect**(std::span<DrawIndexedIndirectCommand> commands)

Issue an indirect indexed draw.

Parameters
commands – Indirect commands to be uploaded and used for this draw

CommandBuffer &**draw_indexed_indirect_count**(size_t max_command_count, const Buffer &indirect_buffer, const Buffer &count_buffer)

Issue an indirect indexed draw with count.

Parameters

- **max_command_count** – Upper limit of commands that can be drawn
- **indirect_buffer** – Buffer of indirect commands
- **count_buffer** – Buffer of command count

CommandBuffer &**draw_indexed_indirect_count**(size_t max_command_count, Name indirect_resource_name, Name count_resource_name)

Issue an indirect indexed draw with count.

Parameters

- **max_command_count** – Upper limit of commands that can be drawn

- **indirect_resource_name** – The Name of the *Resource* to use as indirect buffer
- **count_resource_name** – The Name of the *Resource* to use as count buffer

CommandBuffer &**dispatch**(size_t group_count_x, size_t group_count_y = 1, size_t group_count_z = 1)

Issue a compute dispatch.

Parameters

- **group_count_x** – Number of groups on the x-axis
- **group_count_y** – Number of groups on the y-axis
- **group_count_z** – Number of groups on the z-axis

CommandBuffer &**dispatch_invocations**(size_t invocation_count_x, size_t invocation_count_y = 1, size_t invocation_count_z = 1)

Perform a dispatch while specifying the minimum invocation count Actual invocation count will be rounded up to be a multiple of local_size_{x,y,z}.

Parameters

- **invocation_count_x** – Number of invocations on the x-axis
- **invocation_count_y** – Number of invocations on the y-axis
- **invocation_count_z** – Number of invocations on the z-axis

CommandBuffer &**dispatch_indirect**(const Buffer &indirect_buffer)

Issue an indirect compute dispatch.

Parameters
indirect_buffer – Buffer of workgroup counts

CommandBuffer &**dispatch_indirect**(Name indirect_resource_name)

Issue an indirect compute dispatch.

Parameters
indirect_resource_name – The Name of the *Resource* to use as indirect buffer

CommandBuffer &**trace_rays**(size_t width, size_t height, size_t depth)

Perform ray trace query with a raytracing pipeline.

Parameters

- **width** – width of the ray trace query dimensions
- **height** – height of the ray trace query dimensions
- **depth** – depth of the ray trace query dimensions

CommandBuffer &**build_acceleration_structures**(uint32_t info_count, const VkAccelerationStructureBuildGeometryInfoKHR *pInfos, const VkAccelerationStructureBuildRangeInfoKHR *const *ppBuildRangeInfos)

Build acceleration structures.

CommandBuffer &**clear_image**(Name src, Clear clear_value)

Clear an image.

Parameters

- **src** – the Name of the *Resource* to be cleared

- **clear_value** – value to clear with

CommandBuffer &**resolve_image**(Name src, Name dst)

Resolve an image.

Parameters

- **src** – the Name of the multisampled *Resource*
- **dst** – the Name of the singlesampled *Resource*

CommandBuffer &**blit_image**(Name src, Name dst, ImageBlit region, Filter filter)

Perform an image blit.

Parameters

- **src** – the Name of the source *Resource*
- **dst** – the Name of the destination *Resource*
- **region** – parameters of the blit
- **filter** – Filter to use if the src and dst extents differ

CommandBuffer &**copy_buffer_to_image**(Name src, Name dst, BufferImageCopy copy_params)

Copy a buffer resource into an image resource.

Parameters

- **src** – the Name of the source *Resource*
- **dst** – the Name of the destination *Resource*
- **copy_params** – parameters of the copy

CommandBuffer &**copy_image_to_buffer**(Name src, Name dst, BufferImageCopy copy_params)

Copy an image resource into a buffer resource.

Parameters

- **src** – the Name of the source *Resource*
- **dst** – the Name of the destination *Resource*
- **copy_params** – parameters of the copy

CommandBuffer &**copy_buffer**(Name src, Name dst, size_t size)

Copy between two buffer resource.

Parameters

- **src** – the Name of the source *Resource*
- **dst** – the Name of the destination *Resource*
- **size** – number of bytes to copy (VK_WHOLE_SIZE to copy the entire "src" buffer)

CommandBuffer &**copy_buffer**(const Buffer &src, const Buffer &dst, size_t size)

Copy between two Buffers.

Parameters

- **src** – the source Buffer
- **dst** – the destination Buffer
- **size** – number of bytes to copy (VK_WHOLE_SIZE to copy the entire "src" buffer)

CommandBuffer &**fill_buffer**(Name dst, size_t size, uint32_t data)

Fill a buffer with a fixed value.

Parameters

- **dst** – the Name of the destination *Resource*
- **size** – number of bytes to fill
- **data** – the 4 byte value to fill with

CommandBuffer &**fill_buffer**(const Buffer &dst, size_t size, uint32_t data)

Fill a buffer with a fixed value.

Parameters

- **dst** – the destination Buffer
- **size** – number of bytes to fill
- **data** – the 4 byte value to fill with

CommandBuffer &**update_buffer**(Name dst, size_t size, void *data)

Fill a buffer with a host values.

Parameters

- **dst** – the Name of the destination *Resource*
- **size** – number of bytes to fill
- **data** – pointer to host values

CommandBuffer &**update_buffer**(const Buffer &dst, size_t size, void *data)

Fill a buffer with a host values.

Parameters

- **dst** – the destination Buffer
- **size** – number of bytes to fill
- **data** – pointer to host values

CommandBuffer &**memory_barrier**(Access src_access, Access dst_access)

Issue a memory barrier.

Parameters

- **src_access** – previous Access
- **dst_access** – subsequent Access

CommandBuffer &**image_barrier**(Name resource_name, Access src_access, Access dst_access, uint32_t base_level = 0, uint32_t level_count = VK_REMAINING_MIP_LEVELS)

Issue an image barrier for an image resource.

Parameters

- **resource_name** – the Name of the image *Resource*
- **src_access** – previous Access
- **dst_access** – subsequent Access
- **base_level** – base mip level affected by the barrier

- **level_count** – number of mip levels affected by the barrier

CommandBuffer &**write_timestamp**(*Query* query, PipelineStageFlagBits stage = PipelineStageFlagBits::eBottomOfPipe)

Write a timestamp to given *Query*.

Parameters

- **query** – the *Query* to hold the result
- **stage** – the pipeline stage where the timestamp should latch the earliest

CHAPTER

TWO

BACKGROUND

vuk was initially conceived based on the rendergraph articles of themaister (https://themaister.net/blog/2017/08/15/render-graphs-and-vulkan-a-deep-dive/). In essence the idea is to describe work undertaken during a frame in advance in a high level manner, then the library takes care of low-level details, such as insertion of synchronization (barriers) and managing resource states (image layouts). This over time evolved to a somewhat complete Vulkan runtime - you can use the facilities afforded by vuk's runtime without even using the rendergraph part. The runtime presents a more easily approachable interface to Vulkan, abstracting over common pain points of pipeline management, state setting and descriptors. The rendergraph part has grown to become more powerful than simple 'autosync' abstraction - it allows expressing complex dependencies via *vuk::Future* and allows powerful optimisation opportunities for the backend (even if those are to be implemented).

Alltogether vuk presents a vision of GPU development that embraces compilation - the idea that knowledge about optimisation of programs can be encoded into to tools (compilers) and this way can be insitutionalised, which allows a broader range of programs and programmers to take advantage of these. The future developments will focus on this backend(Vulkan, DX12, etc.)-agnostic form of representing graphics programs and their optimisation.

As such vuk is in active development, and will change in API and behaviour as we better understand the shape of the problem. With that being said, vuk is already usable to base projects off of - with the occasional refactoring. For support or feedback, please join the Discord server or use Github issues - we would be very happy to hear your thoughts!

CHAPTER

THREE

INDICES AND TABLES

- genindex

INDEX

V