# Vulkan Memory-Management and Defragmentation for Open-World Games

Masterarbeit im Fach Informatik

vorgelegt von

**Peter Eichinger**

geb. am 20.07.1987 in Nürnberg

angefertigt am

**Department Informatik**
**Lehrstuhl Graphische Datenverarbeitung**
**Friedrich-Alexander-Universität Erlangen-Nürnberg**

Betreuer: M. Sc. Darius Rückert

Betreuender Hochschullehrer: Prof. Dr. Marc Stamminger

Beginn der Arbeit: 06.11.2018

Abgabe der Arbeit: 15.05.2019

## Abstract

The size of virtual game worlds that can be explored by players is ever increasing. Many games allow exploration in an open-world, i.e. non-linear, fashion. Memory requirements become so high that it is not possible to keep the whole game world in memory simultaneously. This requires from the underlying graphics engine that resources are loaded and unloaded during gameplay. Older APIs, like OpenGL, required to simply create and destroy resources. In contrast, Vulkan requires explicit memory management. Programmers have to take care of the arrangement of memory in order to enable highly performant applications.

This thesis shows a system for efficient memory allocation and deallocation. Depending on the type and size of data, different approaches have to be taken. The memory management may generate distributed free ranges that make future allocations difficult. Therefore, a system to compact allocations and coalesce free space is presented.

# Contents

# Chapter 1

# Introduction

Vulkan is an Application Programming Interface (API) for performing rendering and general purpose compute tasks on modern Graphics Processing Units (GPUs). Because of the explicit nature of Vulkan, tasks that were performed by drivers in older APIs must now be implemented in applications or frameworks. By leaving the implementation up to the programmer, optimizations and specializations can be added that are specific to the type of program. One major difference to other graphics APIs, OpenGL and DirectX prior to version 11 is, that Vulkan requires users to take care of memory management.

Open-world games allow players to explore at least parts of the game-world in a non-linear fashion. This means that the player may choose to perform tasks in an order they choose or even forgo some challenges or locations.

The categorization is not strict, because some games have both linear and open-world sections. Besides the narrative structure, most open-world games feature one or multiple extensive maps for players to explore. Maps may be of a single city, a whole continent, or even an entire galaxy. One of the earliest games in this category, *Elite*[1], features a vast galaxy with hundreds of star systems for players to explore in any order they choose.

Linear games can load and unload assets during loading screens. However, with sizes exceeding 50 GiB for modern open-world games, like *Grand Theft Auto 5*[2], it is impossible to store the whole game-world in graphic memory at the same time. Therefore, in these cases resources have to be managed dynamically during gameplay. This should not negatively impact the gameplay experience, either by interrupting the gameplay altogether or introducing stuttering.

## 1.1 Related Work

In the following, works related to the topic memory management are considered.

Programs generally split the memory into two sections, the stack and the heap. The stack contains all variables that are local to functions. If a program enters a function adds all local variables to the stack. After the function ends the space is removed. In contrast,

[1]Elite (1984), developed by David Braben and Ian Bell

[2]Grand Theft Auto 5 (2013), published by Rockstar Games, developed by Rockstar North

the memory in the heap has no fixed life-time. Programmers have to allocate the memory. Depending on the features of the used programming language, freeing is either done manually or automatically. When allocating memory on the heap, the size does not have to be fixed but instead be calculated during runtime [12]. The term dynamic memory management means the management of heap memory.

Knuth describes dynamic memory management using linked lists. The heap consists of a linked list that describes the free ranges. Each entry contains the size of the free range and a pointer to the next entry. When memory is requested by the program, an entry in the list must be found and reduced in size. Different strategies for finding suitable free ranges are shown. The presented memory management system uses this approach, and therefore a more detailed description can be found in chapter 3 [9].

Knowlton describes buddy systems to dynamically manage memory. Requests for memory are fulfilled by recursively splitting the memory into pairs of equal sizes, called buddies, until the required size is reached. Adjacent buddies that become free are coalesced [8].

Randell explains the two types of fragmentation, internal and external. Internal fragmentation happens when the allocated size of memory is larger than the actual memory. External memory appears when empty but usable space appears in-between allocations. The effects fragmentation has on the effectiveness of dynamic memory management systems is analyzed as well. He discovered that by increasing the internal fragmentation, external fragmentation can be reduced. [11]

Wilson et al. created a survey paper related to dynamic memory management. It shows an overview about different approaches to memory management including free list management and buddy systems [16].

Johnstone and Wilson compare different memory allocation strategies using real world programs. They compare free list algorithms with the buddy system and come to the conclusion that free list algorithms in general provide better results, with respect to memory usage, than buddy systems. [6].

Other approaches to memory management, often used within the kernel of Operating Systems (OSs), are slab and slub allocation. Here preinitialized instances of frequently used data types are stored. Requests for these types can be fulfilled very quickly. Slab and slub allocations are used in the Linux kernel [2, 1]. This approach is not suitable for managing GPU-RAM, because most allocations have different types and sizes.

The works introduced above deal with the management of Central Processing Unit (CPU)-RAM exclusively. Hebert and Kubisch show different strategies for using Vulkan memory. A more detailed description is shown in chapter 3.

## 1.2 Structure & Overview

The following chapter gives an overview about the Vulkan API in general and topics related to this thesis. Furthermore, a comparison between Vulkan and OpenGL is made. Following the introduction to Vulkan, the memory management system for Vulkan is shown. Because of limitations, three different allocators are needed. The process for selecting the correct allocator is shown. The memory management can result in external fragmentation. Therefore, a process to compact the memory allocations and consequently free memory is shown in chapter 4. Following that, an evaluation of the systems is presented. Performance and fragmentation aspects of the memory management and defragmentation systems are analyzed. The thesis ends with the conclusions, containing possible future work and a summary.

# Chapter 2

# Vulkan API

In this chapter, an introduction to Vulkan is presented. For further study, the reader may wish to consult the Vulkan specification [14].

Due to a rapid development process and the young age of Vulkan, the specification is updated almost every month. In this thesis version *1.1.106* is used which can be found at [1]. The most current version can be found at [2].

Section 2.1 contains a brief comparison between OpenGL and Vulkan. The following sections give an overview of Vulkan concepts related to this thesis.

## 2.1 Differences to OpenGL

Vulkan and OpenGL are both specifications for graphics and compute APIs released by the *Khronos Group*. OpenGL provides a higher level of abstraction than Vulkan. Furthermore, OpenGL only allows calls from a single thread, while Vulkan allows a multi-threaded approach. There are two versions of the OpenGL specification, one for desktop and one for mobile devices. The Vulkan API works on both desktop and mobile devices [4, 3, 13]. The following paragraphs show further differences between the two APIs.

OpenGL stores the configuration of the rendering and compute pipeline using a single state machine. The state includes blend mode and the shaders to use. Functions to modify the state machine must be called from a single thread. The user has to take care that the rendering and compute pipelines are in the correct state before performing any work. Before rendering transparent geometry the correct blend state has to be set. If the user then tries to draw opaque geometry the blending state has to be reset. Vulkan encapsulates the whole state of the rendering or compute pipeline in pipeline objects. Users can specify which pipeline object to use for which task.

OpenGL users call functions to perform rendering or compute work directly. The functions return immediately and potential work for the GPU is performed asynchronously. In Vulkan users record all commands related to a rendering or compute task before execution in

[1] `https://github.com/KhronosGroup/Vulkan-Docs/releases/tag/v1.1.106`
[2] `https://www.khronos.org/registry/vulkan/specs/1.1/html/vkspec.html`

command buffers. After recording a command buffer, it is submitted to and subsequently executed by the GPU. The execution of the command buffer happens asynchronously. Command buffers may be reused. For example, the command buffer for rendering static level geometry can be recorded when the level is loading and subsequently be reused during each frame. This reduces the number of API-calls needed every frame.

There are major differences between memory management in Vulkan and OpenGL. In OpenGL, buffers and textures are directly created by the user. The OpenGL driver takes care of appropriate memory types as well as the allocations and deallocations. Furthermore, OpenGL takes care of the correct timing of deallocation of resources. Memory management in Vulkan is explicit, meaning users have to take care of memory allocations and deallocations as well as the selection of the optimal memory type. When using resources Vulkan users have to specify their usages. This is not required in OpenGL.

OpenGL provides a default framebuffer for displaying rendering results to the display; Vulkan uses swapchains to provide this functionality.

The following chapters explain Vulkan topics relevant to this thesis in more detail.

## 2.2 Command Buffer

To perform any tasks, like rendering and executing compute kernels, with Vulkan they must first be recorded in a command buffer. Multiple command buffers may be recorded in multiple threads simultaneously. After recording has finished, the sequence of commands can be submitted for execution on the GPU. Command buffers may be reused multiple times. GPU queues execute command buffers.

Section 5 of the Vulkan specification contains a detailed description about the creation, lifecycle and usage of command buffers [14].

## 2.3 Queues

In order to execute command buffers, they have to be submitted to a queue. Every Vulkan-compatible GPU must provide at least one queue that allows execution of graphics and compute work. Depending on the driver, API implementation, and device, multiple queues that can process command buffers simultaneously are available. Queues are grouped into families and each family has a set of capabilities. These include *Graphics* and *Compute* that allow rendering and compute workloads respectively, as well as *Transfer* which allows memory copy operations.

For example: An *nVidia RTX 2070* has 16 general purpose queues capable of compute, graphics and transfer operations, two dedicated for transfer operations, and 8 dedicated for compute workload. Performing work in a queue dedicated to a single task may be faster than using the general purpose queues.

When command buffers are submitted to a queue, the application continues immediately and they are executed asynchronously. To synchronize the execution of command buffers between each other or with the CPU, semaphores and fences are used respectively.

For further information about queues see Vulkan specification 4.3 [14].

## 2.4 Synchronization

Vulkan supports multiple types of synchronization that either synchronize on the GPU or between the GPU and CPU.

When submitting command buffers to a queue the user can specify a fence. The fence is signaled when all submitted command buffers have finished. Users can interrupt the current thread until the fence was signaled. This allows synchronization between the GPU and CPU.

Semaphores are used to order the execution of multiple command buffers on the GPU. As with fences, semaphores have to be specified when submitting command buffers to the GPU. Users may specify semaphores that have to be signaled before execution of the command buffer may start and semaphores that will be signaled after the command buffer has finished. This allows the user to specify dependencies between command buffers.

Synchronization is treated in chapter 6 of the Vulkan specification [14].

## 2.5 Shaders & Pipelines

The configuration required to perform rendering or compute tasks are stored in pipeline objects. The state includes, blending state, the shaders to use in each pipeline stage, and the type and number of uniforms.

Resources, i.e. buffers and textures, are bound to a pipeline with descriptor.

Chapter 9 of the Vulkan specification describes compute and graphics pipelines in more detail [14].

## 2.6 Descriptor Sets

A descriptor set assigns textures and buffers to binding locations in shaders. After creation, the assignment can be updated to link to different resources. For example, this allows for multiple different textures to be rendered. Descriptor sets are created from descriptor set layouts. The layout, defined by the pipeline, describes the type of resource that is bound to each binding location of a shader. Descriptor sets are managed in object pools [7]. Each pool is limited to a maximum number of descriptor sets instantiated simultaneously. The limits depend on the implementation and users may specify a lower limit when initializing Vulkan.

Vulkan specification Chapter 13 contains details about descriptor set layouts and descriptor sets [14].

## 2.7 Memory

Vulkan differentiates between host memory and device memory. Host memory is used by the implementation of the Vulkan API on the host. Device memory is used for storing resources

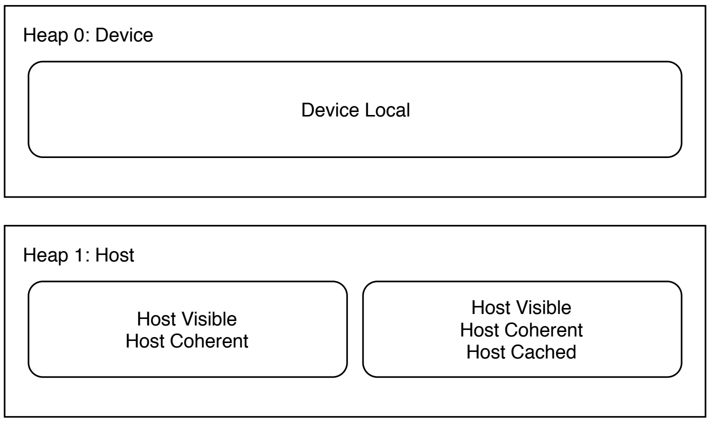


**Figure 2.1:** Memory heaps and types of an nVidia RTX 2070 GPU

but it does not have to be necessarily located on the device. Regardless of location, device memory is always accessible by the GPU.

Device memory must be managed by the application explicitly. Memory is organized into multiple heaps that may be located either on the host or device. For each heap there are potentially multiple types of memory, each with differing properties. Figure 2.1 shows the heaps and memory types for a nVidia RTX 2070 GPU. There are two heaps, one for host and one for device RAM. The device heap has a single memory type. The property *device local* specifies that the memory is not accessible by the CPU. The host heap is split into two memory types. Both types have the properties *host visible* and *host coherent*. The host visible property means that the memory may be mapped to a virtual memory address on the host. Host coherent denotes that host write operations do not need to be flushed to be visible by the device and vice versa. The second host memory type additionally has the property *host cached*. This property means that access to this memory is cached on the host. For example: Textures that are primarily read in shaders should be placed in memory on the device for maximum performance. However, users may want to place uniform buffers in host coherent memory. This has the advantage that access by the application is possible without calling Vulkan functions.

Vulkan does not provide methods to copy data between non-Vulkan CPU-RAM and Vulkan memory that is not host visible. Copy operations may copy between non-Vulkan RAM and host visible memory on the CPU or between host visible memory on the host and device local memory. Consequently, memory that is not visible by the CPU has to be initialized in two steps. Firstly, memory is copied from non-Vulkan RAM to host visible device memory and afterwards from host visible device memory to the device local, non-host visible memory.

In order to use memory, the user has to request it from the API. Every request for memory needs to specify the size and type. There is a restriction on the number of allocations, independent of the size of each allocation, that may exist simultaneously. The exact upper limit of allowed allocations depends on the implementation and device. This limit is one of the two major motivations, the other being improved performance, for the memory management system described in chapter 3. According to the Vulkan Hardware Database ca. 50% of all devices have a limit of 4096 simultaneous allocations [15].

Further description of memory in Vulkan can be found in section 10.2 of the Vulkan specification [14].

## 2.8 Resources

There are two types of resources defined in the Vulkan specification: buffers and images. Buffers are described in section 2.8.1 and images in section 2.8.2.

Before a resource can be used it has to be bound to previously allocated memory. After the initial binding of a resource to a memory range they cannot be re-bound to a different memory range. Resources do not need to fill an entire allocation and can be bound to an offset from the beginning of the memory allocation.

Due to memory alignment restrictions, memory that is allocated for resources may need to be larger than the actual data. Users have to query the required size and a memory allocation must be at least of this size in order to be usable for the resource.

The Vulkan API provides functions to copy data from one resource to another. This includes functions to copy data from images to buffers and vice versa. The source and target memory range used in any copy operation may not overlap. When using a resource for copy operations, either as source or target, special usage flags have to be provided when creating the resource.

Chapter 11 of the Vulkan specification [14] contains information about images and buffers. In particular section 11.6 explains the methods to allocate memory for resources and how to bind images and buffers to memory.

### 2.8.1 Buffers

Vulkan buffers are one dimensional arrays of data with additional properties attached to them. The properties include the usage of the buffer. It describes the possible operations that can be performed with the buffer. Examples for usages are: vertex buffer, index buffer or uniform buffer. It is possible to specify multiple usages for a single buffer. This allows for example to place vertex data and index data within one buffer.

More information on the usage and creation of buffers can be found in sections 11.1 and 11.2 of the Vulkan specification [14].

### 2.8.2 Images

Vulkan images represent multi-dimensional textures. Similarly to buffers they also have properties attached to them. The properties include image usage and layout. Image usages include: sampled (so they can be used by samplers in shaders) or as an attachment to a framebuffer.

In addition to usages images also have a layout. The layout restricts the operations that are possible on an image. For example: *ShaderReadOnlyOptimal* is a layout that is optimal for being read by shaders but may not be used in image copy operations. However, a texture with the layout *General* is usable by all operations in Vulkan, but may be slower than the layout ShaderReadOnlyOptimal when sampled in a shader.

Vulkan commands and functions generally do not use an image directly. The image view provides an abstraction level on top of images and allow the user to address the whole image or a sub-range of it. Image views can be created after the image was bound to memory.

Sections 11.3-11.5 of the Vulkan specification explain images, their usages and layouts, as well as image views in more detail [14].

## 2.9 Swapchain

The swapchain is part of the integration of Vulkan with the windowing system of the OS. Windowing systems provide drawable surfaces to applications. Swapchains are part of an extension, because Vulkan can be used without displaying any images. The swapchain allows to specify if double or triple buffering should be used and whether presenting rendered images should wait for vertical synchronization.

Swapchains contain the images the allocation can use to render to. After a rendering process for a frame has finished the user calls a function to present the current image using the window system. Potentially all images of the swapchain may be in use simultaneously. For example: While one image is in the process of being presented, another might be currently being rendered to and another one has just finished rendering and is waiting to be presented. The minimum and maximum number of images in a swapchain, depends on the windowing system surface. When creating the swapchain, users specify the exact number, $k$.

The fact that all images may be in use at the same time requires users of the Vulkan API to take care when deallocating resources and destroying or modifying descriptor sets. Deletions or modifications can be performed must happen at least $k$ frames after the last usage in a command buffer was added.

Chapter 32.9 of the Vulkan specification explains swapchains in more detail [14].

# Chapter 3

# Memory Management

Because Vulkan device memory must be managed explicitly, as seen in section 2.7, there is a need for a system for managing Vulkan device memory. In this chapter, the processes, algorithms, and classes that comprise the memory management system are described.

The presented system is needed if the limit on the maximum allocations may be reached by the application. Furthermore, using it should increase performance (see section 5.1 for a performance comparison).

A requirement of the memory management is that allocations and deallocations are possible from all threads. For example, in an open-world game new contents, like meshes or textures, may need to be loaded during gameplay. By moving memory allocations and deallocations to another thread the main thread is not interrupted. To ensure a correct behavior in a multi-threaded scenario all operations must be guarded by a mutex to ensure changes to the data structure are executed in sequence.

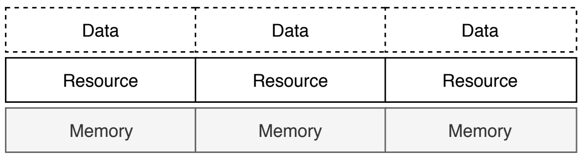
In the following, if not specified otherwise, the term memory is a shorthand for Vulkan device memory. Furthermore, the terms CPU memory and GPU memory will denote device memory located on the CPU and device memory located on the GPU respectively. To describe a strip of memory with a defined beginning and end, the term memory range will be used.

First the three techniques for memory allocations that are possible using Vulkan are shown. Afterwards, the representation of memory locations and free ranges is shown. Then the allocators which are used for the memory management processes are explained. Finally, the processes used for allocation and deallocation of memory is presented.

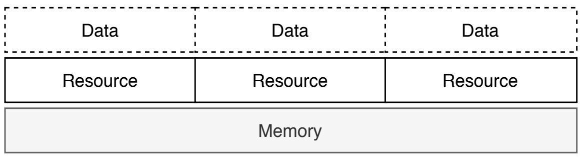
## 3.1 Allocation Techniques

Before the actual process of memory allocation and deallocation may be described, an overview over the possible techniques must be given.

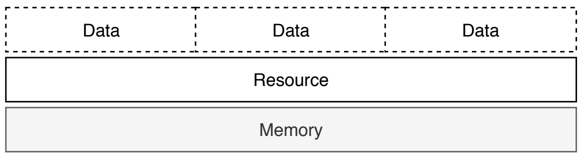
*Vulkan Memory Management* shows different techniques to place resources in memory [5]. Figure 3.1 shows a visual representation of these approaches. Terms for each approach and a short description follow.



**(a)** Unique memory allocation per resource



**(b)** Shared memory for multiple resources



**(c)** One resource contains multiple sub-resources

**Figure 3.1:** Approaches to memory allocation and resource creation in Vulkan

**Unique Memory** The data is placed in a unique resource which in turn gets memory exclusively allocated for it. Resources fill the whole memory allocation. Figure 3.1a shows this approach.

**Shared Memory** Multiple resources are placed within one memory allocation. When binding the resource an offset has to be specified. The Vulkan specification restricts the possible locations for each resource. Offsets of locations must be aligned to an implementation dependent value. See figure 3.1b.

**Sub-Resources** One main resource is used to store data for multiple sub-resources. The main resource fills the entire memory allocation. The sub-resources are defined by the user and are not represented by handles or objects in the Vulkan API. When using the sub-resources the user has to specify an offset and size relative to the beginning of the main resource. The alignment rules, as described in the shared memory approach, apply here as well. This approach is displayed in figure 3.1c.

There are advantages and drawbacks to every approach. When using $n$ resources the unique memory strategy requires $n$ allocated pieces of memory. Additionally, allocating memory, creating resources, and binding them costs runtime on the CPU and GPU. Both the shared memory and sub-resources approach potentially require fewer memory allocations. Fewer memory allocations allow more resources to be used at the same time because the number of memory allocations is limited (see section 2.7). For the sub-resources approach only one resource per memory allocation has to be created. The sub-resource approach is possible for buffers and images. When managing memory for textures it is possible to place multiple textures in one image by using image views. Each view is used for storing a texture. However, this approach is not preferred. Although it was not tested, the computational overhead for tightly packing the textures into a single image during runtime is higher than using the shared memory approach. The sub-resource approach is used solely for buffers.

However, the unique memory approach has a lower management overhead. Users do not have to take care of where to place resources and sub-resources. If the unique memory allo-

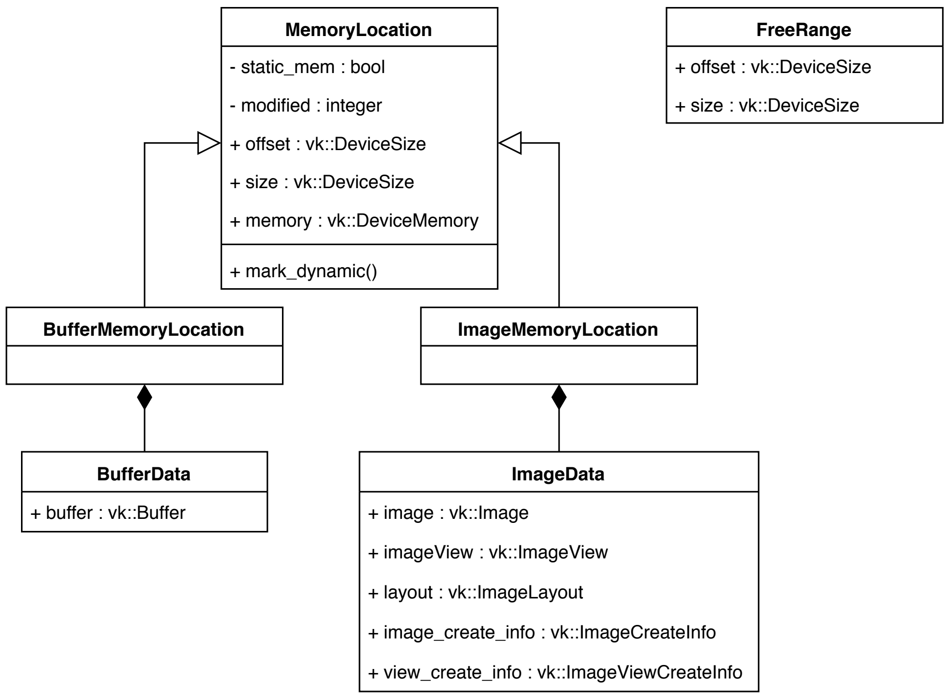


**Figure 3.2:** UML diagram of `MemoryLocation`, its sub-classes, and `FreeRange`

cation strategy is used exclusively no memory management has to be performed at all.

The memory allocation system, described below, implements all three approaches as every one is required. Both resource types are allocatable with the unique memory approach. Shared memory is used for image resources, while sub-resources are used for buffers. When using the sub-resource approach the term super-buffer is used for the Vulkan buffer that fills the entire memory allocation, while sub-buffers refer to the units of data that are placed within.

## 3.2 Memory Location & Free Range

The class `MemoryLocation` represents a contiguous piece of Vulkan memory. Figure 3.2 shows the Unified Modeling Language (UML)-class diagram of `MemoryLocation` and related sub-classes. Memory locations represent resources in memory. They contain a handle to the Vulkan memory allocation as well as the sub-range in which the resource is placed. When using the unique memory approach the offset is always 0 and the size of the allocation is equal to the size of the resource. For the shared memory and sub-resource approaches these values define the sub-range in which the resource is placed. The handle is used when copying data from and to the resource. Resources may be declared static or dynamic. Static and dynamic mean that a resource is writable or in a read-only mode respectively. The terminology comes from the usage in the defragmentation process (see chapter 4). Initially, all resources are static and may be marked dynamic if further write access is not necessary. Depending on the type of resource different data is associated with the memory location. For identifying buffers a handle to the Vulkan Buffer is needed. Images require handles to the associated

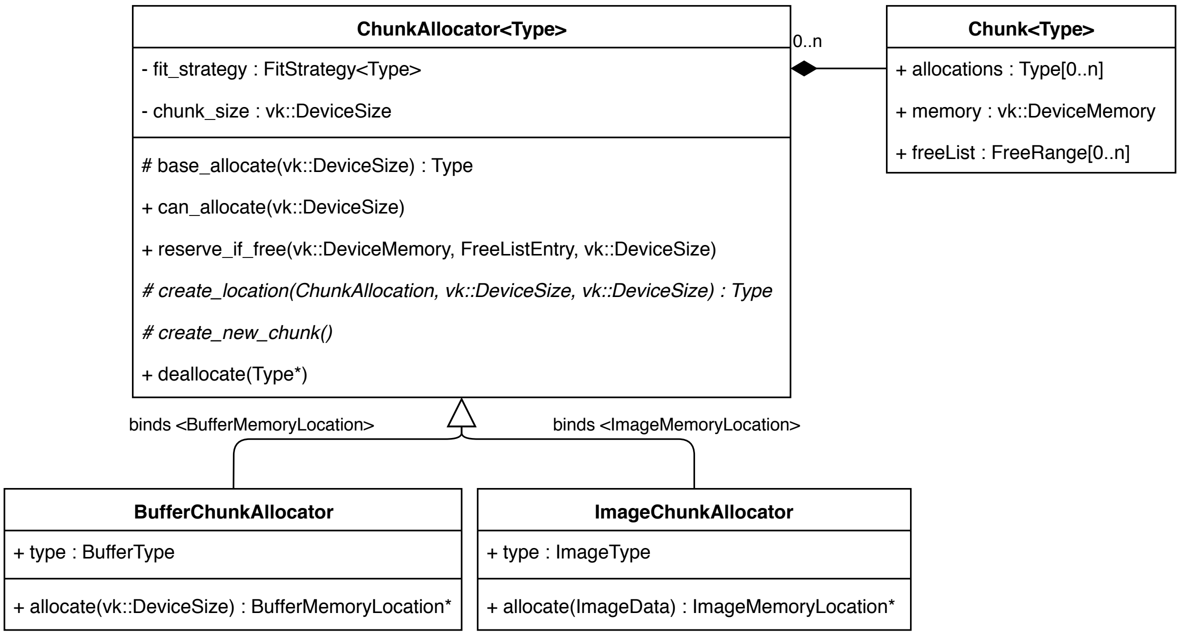

ChunkAllocator<Type>
- fit_strategy : FitStrategy<Type>
- chunk_size : vk::DeviceSize
# base_allocate(vk::DeviceSize) : Type
+ can_allocate(vk::DeviceSize)
+ reserve_if_free(vk::DeviceMemory, FreeListEntry, vk::DeviceSize)
# create_location(ChunkAllocation, vk::DeviceSize, vk::DeviceSize) : Type
# create_new_chunk()
+ deallocate(Type*)
0..n
Chunk<Type>
+ allocations : Type[0..n]
+ memory : vk::DeviceMemory
+ freeList : FreeRange[0..n]
binds <BufferMemoryLocation>
binds <ImageMemoryLocation>
BufferChunkAllocator
+ type : BufferType
+ allocate(vk::DeviceSize) : BufferMemoryLocation*
ImageChunkAllocator
+ type : ImageType
+ allocate(ImageData) : ImageMemoryLocation*

**Figure 3.3:** UML diagram of `ChunkAllocator`

image and image view. Furthermore, the current layout of the image is stored. Finally, the memory location for images, need to store the data that is needed to recreate the image and image view.

`FreeRange` represents a free memory range. It consists of the fields size and offset. The handle to the associated Vulkan memory is not stored because no Vulkan API operations are performed on free ranges.

In the following sections and chapters the size and offset of memory locations and free memory ranges is oftentimes required. Let the functions size$(m)$ and offset$(m)$ denote the size and offset of a memory location or free range $m$.

## 3.3 Chunk Allocator

The chunk allocator realizes the approaches shared memory and sub-resources described in section 3.1 for images and buffers respectively. Figure 3.3 shows an excerpt of the UML diagram for this allocator.

In "The Art of Computer Programming" a technique to manage dynamic storage allocations for heaps is shown [9]. Heaps consist of allocated regions and free ranges between them. Each free range contains its size and a pointer to the next free range, essentially forming a linked list of free memory spaces. When deallocating a piece of memory the newly created free range may be placed in the list using the following different approaches: FIFO, LIFO, ordered by size or ordered by address. When using the ordered approaches, the free range entries are sorted by address or size. The FIFO or LIFO approaches insert the free range entry either at the beginning or the end of the list respectively.

The chunk allocator orders the free ranges by address. This simplifies the coalescing of adjacent free ranges. The dynamic memory allocation technique shown in [9] store the free list within the heap that is managed. This approach is not viable for Vulkan device memory. Instead, the free list is stored separately in CPU memory. Storing it in device memory would have the potential drawback of needing to access memory on the GPU, which possibly has higher latency than CPU memory. There may be some host-device combinations where no dedicated memory for the GPU exists (for example integrated GPUs) but the allocator should provide optimal performance for the most systems.

When the heap described by Knuth runs out of free space it is extended by requesting more memory from the OS. Virtual memory addresses and the translation lookaside buffer make it possible to extend the heap [12, 9]. In Vulkan this extension is not possible, because memory allocations are fixed in size. If no free memory to fulfill a request is available, a new allocation has to be made. In the following these allocations are called chunks. This subdivision of the memory has a further consequence: Resources need to be placed within a single chunk.

Chunk sizes may vary between different allocators and depend on the type of resource and usage. For example: Uniform buffer sizes are often between 10 B and 1 KiB. A chunk size of 1 MiB was chosen to allow many allocations within a single chunk but not wasting memory by allocating too much memory. For all other types of buffers a size of 64 MiB was chosen. Vertex and index buffers are typically larger than uniform buffers. Chunks of image allocators have a size of 256 MiB. Modern games often have textures with a resolution of up to $4096 * 4096$ pixels and a size of up to 64 MiB. This assumes modern GPUs with multiple GiBs of RAM.

Each chunk maintains a list of the allocated resources and the available free ranges. Furthermore, each chunk stores a handle to the associated Vulkan memory. Memory locations and free ranges are always ordered by chunk and then by offset within a chunk. Chunks are ordered by their allocation time.

### 3.3.1 Allocation

An overview about the allocation process is given in figure 3.4. The following sections explain the steps involved.

The process of allocating memory for a resource begins with finding a contiguous strip of memory that is large enough to store the whole resource.

Wilson et al. describe the strategies *first-fit*, *best-fit*, and *worst-fit*. They furthermore show that no strategy outperforms, with respect to fragmentation, any other in every scenario. On average best-fit and worst-fit are slower than *first-fit* and *next-fit*[9, 16]. All of these strategies were implemented and the user may choose between them.

In section 5.2 a comparison of the quality with respect to fragmentation and performance is shown.

Strategies may fail to find a valid location for the resource. This is the case if no free list entry is large enough to hold the requested resource. Consequently, a new chunk must be allocated. Otherwise, the process continues with the resource creation.

Figure 3.5 shows an overview about a sample allocator. White squares represent free space and shaded squares represent allocated pieces of memory. This allocator uses a chunk

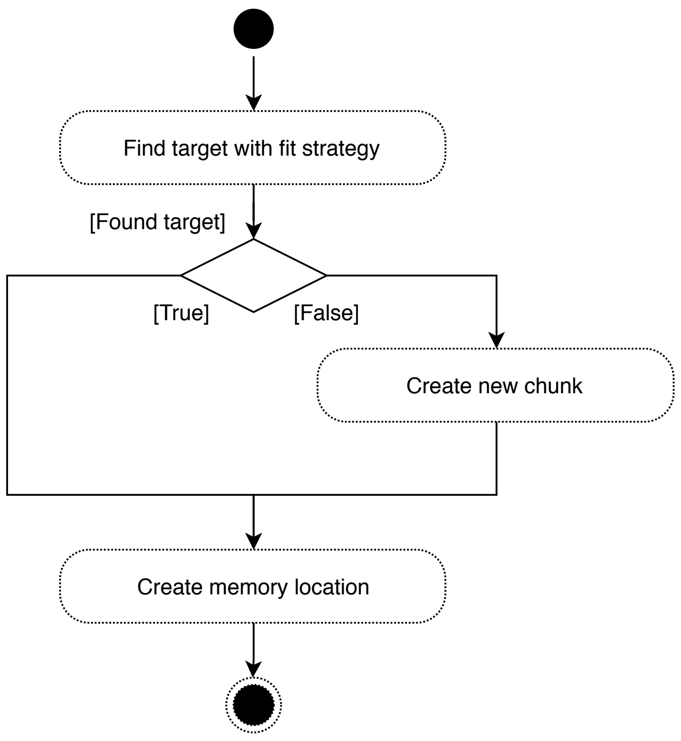


**Figure 3.4:** Activity diagram of the chunk allocator process

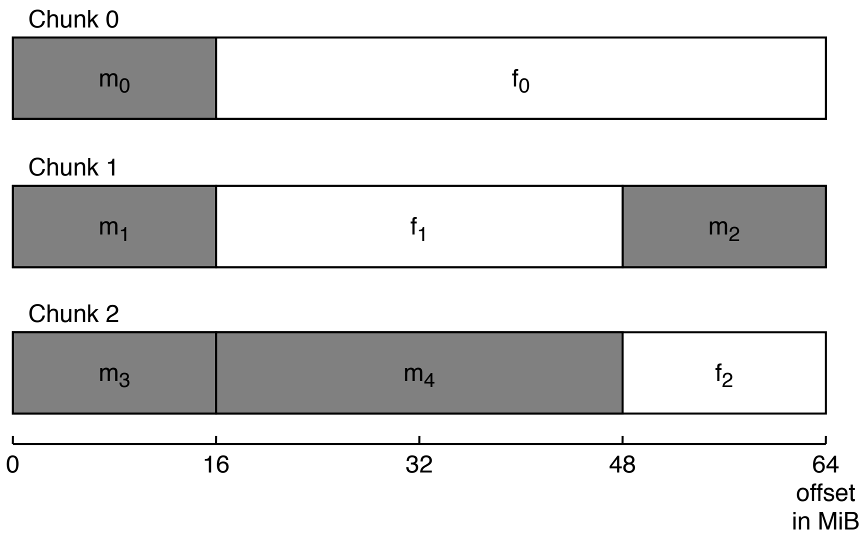


**Figure 3.5:** Example chunk allocator with multiple allocated regions

size of 64 MiB. In this sample memory for five resources $m_0, \ldots, m_4$ is allocated. Allocations $m_0, \ldots, m_3$ are 16 MiB in size. $m_4$ has a size of 32 MiB. The free spaces between these allocations is identified with $f_0, f_1$, and $f_2$. There are three chunks allocated in total.

The following paragraphs describe the functionality of the strategies. The allocator shown in figure 3.5 will be used to show which free list entry would be chosen by each strategy when allocating 16 MiB. The examples below are alternatives and not done in sequence. If not specified otherwise, searches are performed starting with the first free list entry in an ascending order as described in the previous section.

**First-Fit** If this strategy is used the allocation process will select the first free list entry that is large enough to store the resource. The free list entries are traversed by address order. Wilson et al. showed that this approach tends to move small allocations to lower addresses [16]. This strategy would use $f_0$ for the example resource.

**Best-Fit** With best-fit the smallest location, which is large enough to store the resource, is chosen. In case of multiple such locations, the first one will be used. The disadvantage, when compared to first-fit, is that all free ranges must be visited to find the target location. However, if there is a free list entry with the exact same size as the requested strip of memory, the search stops early. Wilson et al. determined another disadvantage of best-fit: The resulting free ranges after allocation tend to be too small to be usable. A lot of free space may be wasted. Best-fit will select $f_2$ for the sample allocation.

**Worst-Fit** This strategy selects the largest free list entry. Should there be multiple largest entries, the first one, as per the ordering described above, is selected. As with best-fit, the search stops when a free list of the same size as the resource is found. If not free range of the same size as the allocation request can be found all free range entries have to be traversed. According to Wilson et al., this strategy leads to too few free range entries that are large enough and therefore new chunks may have to be created earlier than with other strategies [16]. Worst-fit will select $f_0$ for the sample.

The result of this step is either a free range entry that is large enough to fulfill the memory request or a special value indicating that no range is large enough.

If no free range was found a new chunk has to be created. The creation process of the chunk differs for images and buffers. For images only the memory has to be allocated. The images and imageviews are created for each resource separately. For buffers a buffer has to be created. The new resource will be placed in the new chunk.

After either finding a location with the fit strategy or creating a new chunk the actual memory location can be created. The resulting location will be left aligned with the previously chosen free range. Remaining free space, if any, is placed after the resource. This step also differs slightly between images and buffers. Sub-buffers are not explicitly created with the Vulkan API and have no corresponding handle. Instead the process uses sub-ranges within the previously created super-buffer. It is sufficient to set the appropriate fields,`offset` `size`, and `memory`. The Vulkan image object must be created before the allocation process. This is

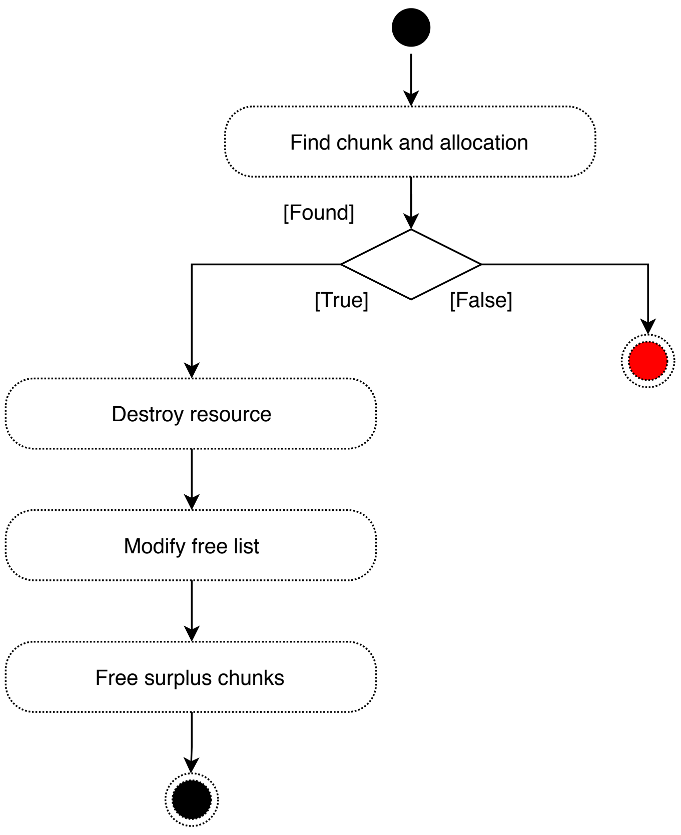


**Figure 3.6:** Activity diagram of the chunk deallocation process

necessary because the exact size of an image is only calculated after the creation of the image has taken place. Afterwards the image view, using the creation data stored in `ImageData`, will be created

### 3.3.2 Deallocation

The deallocation of memory in the chunk allocator involves multiple steps as well. Deallocation begins with finding the resource in the chunks. The allocation has to be performed by the same allocator. Otherwise, the program is in a possibly undefined state.

Afterwards the process continues with destroying the resource. During this step of the deallocation process of the chunk allocator the Vulkan handles associated with the resource are destroyed. For buffers nothing has to be done. The handle of the super-buffer may still be in use by other sub-buffers in the same chunk and may only be destroyed when the program ends or a chunk is freed. For textures, Vulkan image and image view instances are destroyed.

After the resource has been destroyed a new free list entry is created. The new entry is placed into the free list in an by address order . Adjacent free list entries are coalesced.

In the example shown in figure 3.5 should the user deallocate the resources $m_3$ and then $m_4$, three consecutive free range entries will be present. The first one from 0 B to 1 MiB (in

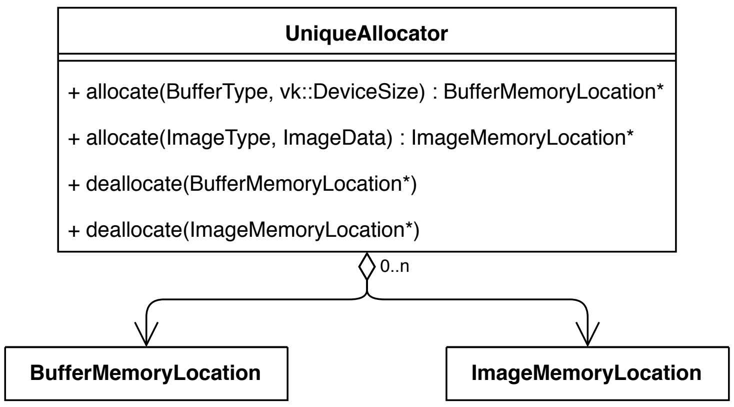

UniqueAllocator
+ allocate(BufferType, vk::DeviceSize) : BufferMemoryLocation*
+ allocate(ImageType, ImageData) : ImageMemoryLocation*
+ deallocate(BufferMemoryLocation*)
+ deallocate(ImageMemoryLocation*)
0..n
BufferMemoryLocation
ImageMemoryLocation

**Figure 3.7:** UML diagram of `UniqueAllocator`

place of $m_3$) the second from 1 MiB to 3 MiB (in place of $m_4$) and the third ($f_2$) from 3 MiB to 4 MiB. The three entries will be replaced by a single one from 0 B to 4 MiB.

Chunks at the end without any resources are freed. One free chunk is kept in order to speed up subsequent allocations.

## 3.4 Unique Allocator

The unique allocator is needed because the allocators for the shared memory or sub-resources approaches have a limit on the maximum size of a single allocation: The chunk size.

The unique allocator allows for allocations of any size for both types of resources. Furthermore, in contrast to the chunk allocator, the unique allocator allows allocations for all combinations of usage and memory type.

Figure 3.7 shows an excerpt of the UML class diagram of `UniqueAllocator` and related classes. This allocator implements the unique memory allocation approach as described in section 3.1 and figure 3.1a. This allocator is only used if the chunk allocator is not usable, due to the size constraint. For each resource a separate memory range is allocated.

## 3.5 Allocation Process

Figure 3.8 shows an activity diagram for the allocation process. Squares with rounded edges represent activities, where either the user or the process have to perform some work. Diamonds denote decisions where either one path is taken. The decisions depend on the last action or decision. To make identification in the text easier, the names of activities and decisions start with a numerical and alphabetical identifier respectively. Only minor parts of the process differ depending on the type of resource for which memory is requested. These differences will be explained in the respective parts below.

The process begins with a user asking for memory to store a resource (filled circle at the top) and ends with the user receiving a memory location (black circle with ring at the bottom;

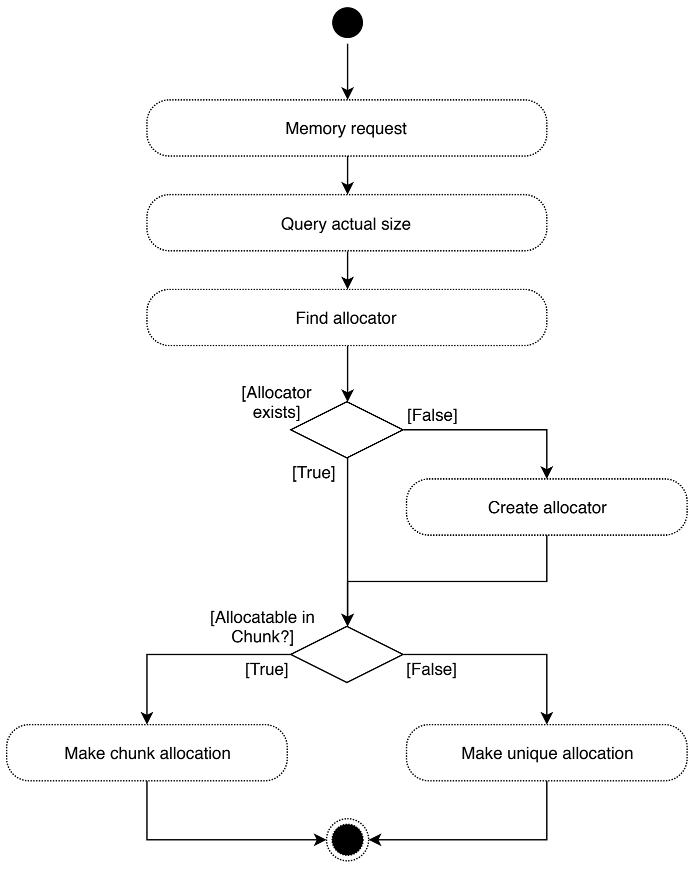


**Figure 3.8:** UML activity diagram of the allocation process

see section 3.2).

## Memory Request

The process begins with receiving a memory request by the user, containing all necessary information for the allocation. This process differs in parts between buffer- and image-allocations. Both resource types require a specification of the required memory type (see section 2.7) and intended usage (see section 2.8.1 and 2.8.2).

**Buffers** In addition to the information above, the request has to contain the size of the buffer.

**Images** The request for an image is more complex than for buffers. `ImageMemoryLocation` require an image and an image view. As described in section 2.8.2 the view may only be created after the image was created and bound to a piece of memory. The request has to contain the necessary information for the creation of the image and the view. The implementation requires an instance of the class `ImageData` (see figure 3.2) where only the fields with the creation information are filled.

#### Query Actual Size

This step is only needed for image allocations. As described in section 2.8 the actual size needed for an allocation of a resource may differ from the memory that the user actually uses. Before allocation may proceed the actual size of an image in memory has to be queried. To achieve this the Vulkan image instance has to be created beforehand. This process is done in this step.

#### Find Allocator

Each chunk allocator only manages allocation for a single combination of memory type and resource usage. In this action the allocator that corresponds to the requested type and usage is searched. This leads to decision ,a. Allocator exists'.

#### Allocator Exists

If an existing allocator for the requested usage and type combination exists the process continues with the decision. Otherwise a new chunk allocator for this combination has to be created.

#### Allocatable in Chunk

Whether or not the allocation can be done in the chunk allocator depends on the size of the requested piece of memory. This decision differs slightly for buffers and images.

The request for a buffer resource already contains the needed size.

The size of image resources cannot be calculated in advance, since the size depends on the implementation and used hardware (section 3.1). Therefore the image structure has to be created and the actual size of the image has to be requested from the Vulkan API.

If the size exceeds chunk size of the chunk allocator, the unique allocator has to be used and the process continues with '6. Make unique allocation'. The unique allocator is explained in section 3.4. Otherwise a chunk allocation happens ('5. Make chunk allocation'). The chunk allocation process is described in section 3.3.

## 3.6 Deallocation Process

The process of memory deallocation is shown in figure 3.9. The structure is similar to the allocation process. There are only differences on the implementation level that will not be shown here.

Deallocation begins with a request by the user. They have to specify an instance of one of the two `MemoryLocation` sub-classes, `ImageMemoryLocation` or `BufferMemoryLocation` that should be deallocated.

The next step *Find allocator* is identical to the action with the same name in the allocation process 3.5. If the allocation cannot be found the program is possibly in an undefined state.

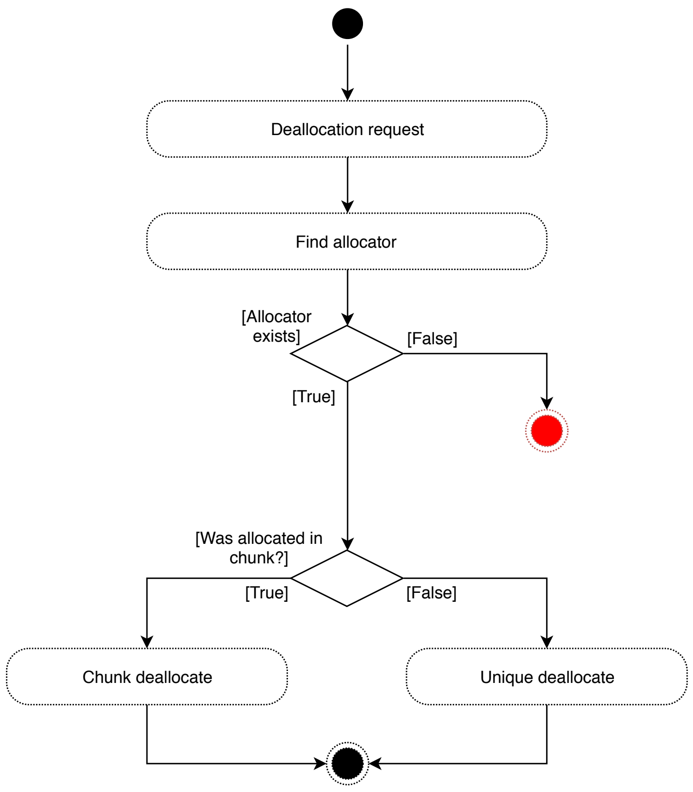


**Figure 3.9:** UML activity diagram of the deallocation process

It is not possible for an allocation to exist that does not have a corresponding allocator, since every allocation process leads to the construction of an allocator for that memory type.

Afterwards a check is performed whether the allocation was done with the chunk allocator or with the unique allocator. As in the allocation process the check depends on the size of the allocation. This time however, the size is known for both buffer and image resources, they were both created beforehand. Depending on the result either the deallocation process of the chunk allocator or the unique allocator is started (see section 3.3 and 3.4 respectively).

Chapter 4

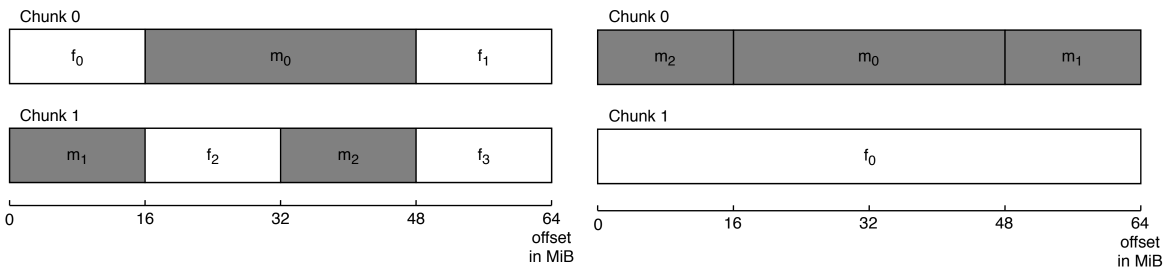
# Defragmentation

When deallocating resources using the memory manager free memory ranges may appear between other resources. This happens when resources were allocated using the sub-resource or shared memory approaches. The total free space available in the allocator is split into multiple free ranges. This is called external fragmentation. Additionally internal fragmentation, which occurs when the allocation process has to allocate more memory than required, exists. Internal fragmentation happens because of the alignment requirement of memory locations. This requirement is put in place by the Vulkan API and depends on the implementation [11].

In the following chapter a process to reduce the external fragmentation of free memory is presented. Here defragmenting means moving resources to a lower memory location. The defragmentation process is only applicable to the shared memory or sub-resource approaches to memory management. Using the unique memory approach, each memory allocation only contains a single resource and the lifetime of the memory and resource is linked. When the resource is destroyed the memory gets deallocated. Therefore, only resources that were allocated with the Chunk Allocator (section 3.3) need to be defragmented. Furthermore, allocation is only required when resources are created and destroyed during the runtime of the application. Open-world games are, for example, applications in which resources are loaded and unloaded during gameplay. Unloading resources leads to free memory ranges. The aim of defragmentation is to increase the size of the free memory ranges and allow allocations of greater sizes.

The figures 4.1a and 4.1b show an abstract graphical representation of two allocators $A$ and $B$. The type of resource and memory are not relevant to convey the motivation for defragmentation. Both have the same number and size of chunks as well as three allocations of the same size $m_0, m_1$, and $m_2$. Furthermore, both have the same amount of free space in total. The memory positions of the allocations, and consequently the free memory ranges, differs between both allocators. Allocator $A$ has four free memory ranges, $f_0, \ldots, f_3$ of 16 MiB each. However, allocator $B$ has one free range, $f_1$, of 64 MiB. When using allocator $A$, a new chunk has to be created when the user creates a resource with a size bigger than 1 MiB. Allocator $B$ may allocate one or multiple resources that are in total up to 4 MiB in size without needing to create a new chunk. The aim of the defragmentation process is to compact the allocations. This allows for a more efficient usage of the allocated chunks.

In the following sections the term *operation* means the action of moving a resource from one



**(a)** Allocator $A$; allocations spread out with small free spaces in each chunk

**(b)** Allocator $B$; allocations placed in a compact manner.

**Figure 4.1:** Chunk allocators with differing resource placement

memory location to another. *Source* and *target* denote the current and new memory locations respectively. Only operations where the target is ordered below the source are considered here. The tuple $(m, f)$ represents an operation to move the resource $m$ to the free space at $f$. Furthermore, the tuple $(m, f, \Delta w)$ expands upon the tuple $(m, f)$ with a weight that gets assigned during the prioritization step (see section 4.2).

As described in section 2.8, resources may not be re-bound to a different memory location after the initial binding. Vulkan only supplies mechanisms to copy data from one resource to another. Therefore, for the duration of the process multiple copies of the same resource may be present.

The defragmentation process should not disrupt the main thread of the program. Otherwise, stuttering or a reduction of the framerate may occur. Therefore, each defragmentation process, there is one for each allocator as described above, will run in a separate thread. Allocations and deallocations of resources interrupt the defragmentation process. Special care has to be taken when to perform the the copy of the data. Copying data uses memory bandwidth and may stall other operations. In order to prevent stuttering defragmentation should ideally be done while the GPU is not performing any work. Section 4.4, describing the step *Copy*, contains a description of a mechanism to reduce stuttering.

Placing memory allocations in the chunks is a kind of the bin-packing problem. The chunks of the allocator can be interpreted as bins and the memory allocations are the objects that should be distributed in them [10]. However, using bin-packing heuristics for defragmentation is not viable. Firstly, all objects that must be packed are already placed within the bins. Secondly, performing all necessary operations would require an enormous computational overhead. Furthermore, resources are potentially deallocated during the defragmentation process. This would require the bin-packing algorithm to start over. Lastly, the search for an optimal solution is a combinatorial NP-hard problem. The process for defragmentation therefore has to use heuristics specifically designed for this problem.

Defragmentation is an optional process. Some kinds of applications do not require it. For example: Programs that load all resources during startup and do not deallocate or allocate new resources during runtime. Free ranges only appear at the end of chunks and will remain for the duration of the program. No resource is small enough to fit these free ranges, otherwise they would have been filled during load-time of the application.

The defragmentation process begins with the selection of the operations. These three steps are described in sections 4.1 to 4.3 and results in the operations that will be performed. Search and prioritization is performed at the same time; possible operations get a priority assigned immediately after they are found. To complete an operation it has to go through three sequential stages (see sections 4.4 and 4.5).

The defragmentation process ends when all operations are finished and no new ones can be found. It will restart after a memory deallocation has occurred.

## 4.1 Search

At any point there may be multiple operations that can be performed. The possible operations for the allocator $A$ in figure 4.1a include $(m_1, f_0)$ and $(m_2, f_0)$. Moving $m_1$ either to $f_2$ or $f_3$ and $m_3$ to $f_3$ do not constitute operations as defined above. The target free space must not be located after the source memory location. On the other hand, there are no possible operations for allocator $B$ in figure 4.1b.

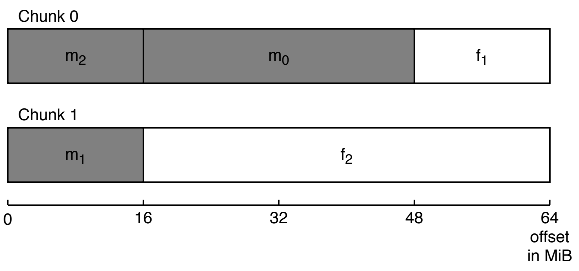


**Figure 4.2:** Allocator $A$ after an operation was performed

Some operations may be mutually exclusive, meaning the execution of one may make others impossible or invalid. Figure 4.2 shows the allocator $A$ after $(m_2, f_0)$ was executed. Operations involving $m_2$ and $(m_1, f_0)$ become invalid. The new position of $m_2$ is lower than the free range entries of the operations $(m_2, f_1), (m_2, f_2)$. $(m_1, f_0)$ is impossible, because $f_0$ does not exist anymore, since the free space is now occupied by $m_2$.

The algorithm shown in 4.1 shows the search for possible operations. It gets the allocations and free ranges of a chunk allocator as input and returns a set of all possible operations.

Only read-only resources are considered for defragmentation. Initially all resources are writable. After copying data to a resource, the user can mark a resource as read-only. During the execution of an operation there are two copies of it. Therefore, a need to synchronize the write accesses would be required. The synchronization could delay the write in such a way that it would be noticeable during the runtime of the application and should therefore be avoided.

```
input  : $M$ ordered set of all allocations, $F$ ordered set of all free ranges
output: $O$, set of all possible defragmentation operation tuples $(m, f)$
$O \leftarrow \emptyset$;
for $m \in M$ do
    for $f \in F,$ with $f < m$ do
        if $size(f) \geq size(m)$ then
            $O \leftarrow O \cup (m, f)$
        end
    end
end
```

**Algorithm 4.1:** Search algorithm for possible operations.

## 4.2 Prioritization

Operations get a weight assigned to them during this step.

The weight $\Delta w$ determines the importance for copying the resource $m$ to the free range $f$. The prioritization results in the tuple $(m, f, \Delta w)$ A lower weight for $\Delta w$ signifies an operation that is preferred. If there are multiple operations for a single resource, only the one with the lowest weight is considered for execution in the next step.

The weight of an operation $\Delta w$ is defined as follows:

$$\Delta w = w_{\text{after}} - w_{\text{before}} \tag{4.1}$$

The value $w_{\text{before}}$ is the weight of the current memory location of $m$. $w_{\text{after}}$ is the hypothetical weight of the memory location of the resource after it was moved to $f$.

The weight $w$ of a memory location is the sum of two criteria for the position and surrounding free space of a memory location.

$$w = w_{\text{loc}} + w_{\text{free}} \tag{4.2}$$

The process of determining the terms $w_{\text{loc}}$ and $w_{\text{free}}$ for a given memory location is described in sections 4.2.1 and 4.2.2 respectively.

### 4.2.1 Location Weight

The $w_{\text{loc}}$ term used in equation (4.2) represents the memory location ordering. Let $\text{loc}_1$ and $\text{loc}_2$ be two memory locations then

$$w_{\text{loc}_1} < w_{\text{loc}_2} \Leftrightarrow \text{loc}_1 < \text{loc}_2 \tag{4.3}$$

The concrete values for $w_{\text{loc}}$ are calculated as follows

$$w_{\text{loc}} = \alpha_{\text{loc}} \left( w_i + \frac{o}{n} \right) \tag{4.4}$$

with $w_i$ being the index of the chunk that contains the memory location, $o$ being the offset of the memory location within the chunk and $n$ being the total size of the chunk. The value $\alpha_{\text{loc}} \in [0; \infty)$ is a scaling factor to modify the prioritization with respect to the other term $w_{free}$.

This criterion is similar to a the first-fit strategy shown in the previous chapter. When using other strategies the defragmentation process might move a resource immediately after it was allocated. The defragmentation process should therefore only be used if the first-fit strategy is used.

Using this criterion, memory locations are preferred to be placed at the beginning of the allocator. Consequently, the free space moves to the end and multiple free ranges tend to coalesce. Furthermore, chunks at the end that become free during the defragmentation may be deallocated. This allows the reuse of memory for other types of resources. The typical value for $\alpha_{\text{loc}}$ is 1, it is only relevant as a scaling factor with the other criterion.

## 4.2.2 Free Space Weight

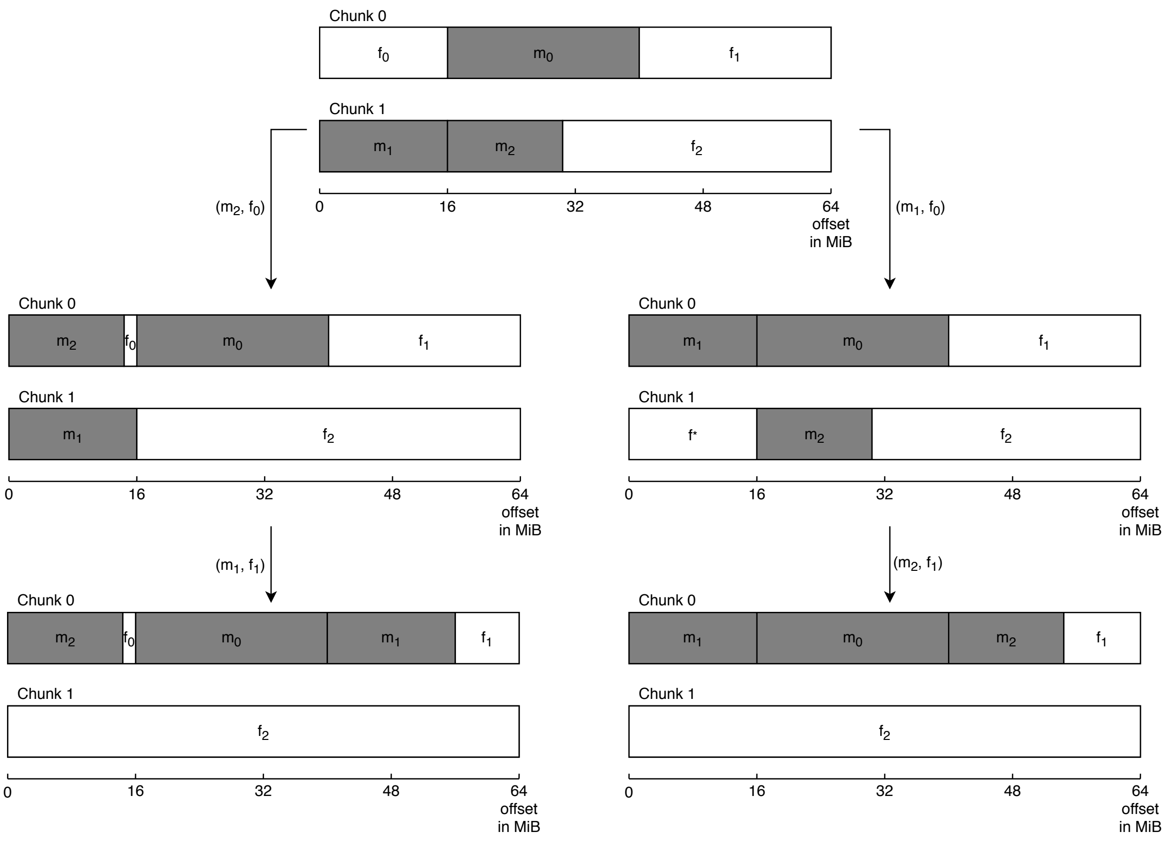


**Figure 4.3:** Two alternative operation sequences for an allocator

The free weight value $w_{\text{free}}$ assigns a numerical value to free ranges surrounding resources. An allocation may have up to two free ranges – one before and one after it. The value should

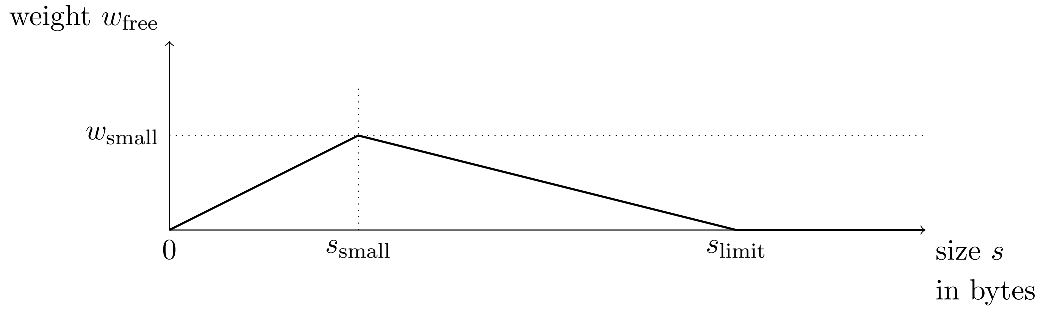


**Figure 4.4:** Plot of the function $h_{\text{free}}(s)$

give a greater value for free ranges that are possibly not usable for further resources.

Figure 4.3 shows the motivation for $w_{\text{free}}$. It shows a tree of possible operations that can be performed on an allocator. The current state of the allocator is the root of the tree. The allocator has three resources $m_0, m_1, m_2$ with the sizes 24 MiB, 1 MiB, and 14 MiB. $(m_2, f_0)$ and $(m_1, f_0)$ result in three free ranges. The location criterion shown above would select $(m_2, f_0)$. Performing this operation, the free range $f_0$ becomes too small to be used by future resources. However, it leads to a larger free space $f_2$. The second operation results in free ranges that are similarly sized to the already allocated resources. When only using the location criteria, described in section 4.2.1, operation $(m_2, f_0)$ would be preferred. To reduce the number of small free ranges created by defragmentation an additional criterion that assesses the free space around allocations is introduced. The benefit of the free range criterion can only be seen when further operations are performed. The leaves of the tree in figure 4.3 shows the results after two operations were performed. If $(m_2, f_0)$ and $(m_1, f_1)$ is performed there are three free memory ranges one of which is probably not usable. Should $(m_2, f_1)$ be executed after $(m_1, f_0)$ only two free ranges remain, all of them can be used for future resources. Therefore, a criterion that prefers to create small free ranges can help to reduce free memory fragmentation.

Figure 4.4 shows a plot of the function $h_{\text{free}}(s)$, which assigns a weight to a free range of the size $s$ in bytes. The values $s_{\text{small}}$, $s_{\text{limit}}$, and $w_{\text{small}}$ are configurable by the user. The function assigns free pieces of memory in the range of $(0, s_{\text{limit}})$ a weight greater than 0. Larger free entries get the weight 0 assigned. If two successive allocations have no free range between them the weight 0 is assigned as well. No free space surrounding allocations is the optimum. For values in the range $(0, s_{\text{small}}$ the weight increases linearly. Very small free ranges do not matter because the wasted space is small enough to be insignificant when compared with the size of the allocations. Sizes around $s_{\text{small}}$ are problematic because they have a significant size but are not usable for storing resources. Therefore, the value should depend on the typical size of the resources stored with the allocator. If the free memory range gets large enough $(> s_{\text{small}})$ the free range is more likely usable for future resources.

For example: The values for $s_{\text{small}}$ and $s_{\text{limit}}$ are 0.1% and 1% The weight $w_{\text{small}}$ has a value of 0.5. This ensures that the free weight criterion does not influence the heuristic as much as the location. Moving greater locations to lower free ranges, should be preferred

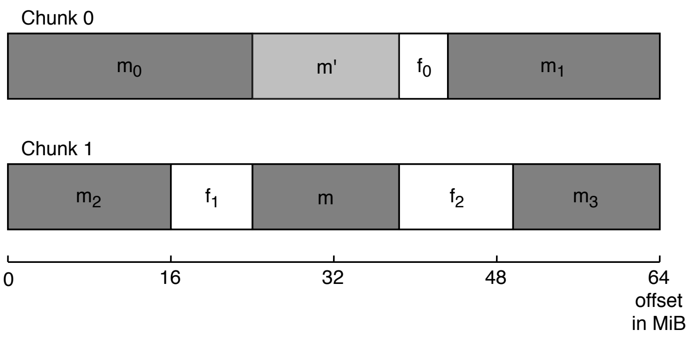


**Figure 4.5:** Effect on free ranges when removing or adding an allocation

When calculating the free range weight not only the free range surrounding the source or target memory location has to be analyzed. Figure 4.5 shows as sample operation where the memory $m$ is moved to $m'$. The new position affects the free memory surrounding the allocations $m_0$ and $m_1$. After the memory was moved, a new free range between $m_2$ and $m_3$ is created. Therefore, the free ranges of the surrounding allocations of source and the hypothetical target have an influence. Let $f_p(m)$ and $f_n(m)$ be the size of free space before and after an allocation $m$. Furthermore let $m_p$ and $m_n$ be the previous and next allocations of an allocation $m$ The value for $w_{\text{free}}$ is calculated as follows:

$$w_{\text{free}} = \sum_{a \in \{m, m_p, m_n\}} f_p(a) + f_n(a)$$

## 4.3 Selection

During this step the execution order of operations is determined.

```
input : $S_{\text{op}}$, set of prioritized defragmentation operation tuples $(m, f, \Delta w)$
output: $R$, list of recorded operations
$R \leftarrow \emptyset$;
for tuple $(m, f, \Delta w) \in S_{op}$ do
    $t \leftarrow$ ReserveIfFree($f$, $size(m)$);
    if $t \neq$ null then
        $R \leftarrow R \cup$ Record($m$,$t$)
    end
    $S_{\text{op}} \leftarrow S_{\text{op}} \setminus (m, f, \Delta w)$;
end
```

**Algorithm 4.2:** Select operations with the lowest priority.

The algorithm shown in 4.2 shows the selection step. `ReserveIfFree` represents a call to the function `reserve_if_free()` of the chunk allocator (see section 3.3). The function returns a `MemoryLocation` instance if the free range $f$ is still available. Otherwise it returns *null*. The free range may not be available anymore when an operation with a higher priority has already been selected or a new resource has been placed here in the meantime.

`Record` represents the second part of this step. The command buffer for the copy process is recorded here.

**Buffer** A function call to copy data from $m$ to $f$ is recorded into the command buffer.

**Image** Images must be in a specific layout so that they may be used in copy operations (see section 2.8.2). The images are generally in the layout *ShaderReadOnlyOptimal* to allow for maximum performance when using them in rendering tasks. This layout does not allow copy operations to take place. Therefore a compute shader that samples the source and stores the data in the target has to be used. Additionally, a new image and image view for the target memory location has to be created using the information stored in the memory location of the source (see section 3.2). Existing ones may not be modified after they have been bound to a memory range (see section 2.8.2).

## 4.4 Submit & Copy

The first step in the execution of an operation is the submit process. Here the command buffers, responsible for copying the data, are submitted to a queue on the GPU. After submission the copy process is performed on the GPU

Copying resources may delay the execution of other work on the GPU. Therefore, the copy process should be either controlled by the user or performed at a time such as to not interrupt other tasks.

The time it takes for copy processes to complete is measured. Let $d$ be a defragmentation operation, with $s_d$ being the size in KiB and $t_d$ the execution time in ns. The speed of a defragmentation operation $d$ is

$$v_d = \frac{s_d}{t_d} \tag{4.5}$$

The defragmentation process for a given memory type stores its speed $v_{\text{defrag}}$ using a moving average.

$$v_{n+1} = \frac{v_d + n \cdot v_n}{n+1} \tag{4.6}$$

There are two possible alternatives that can be considered for when to perform defragmentation. The figures 4.6 and 4.7 show an overview of a generalized update and render loop of an application. The image shows two Vulkan queues, one for general purpose tasks (rendering and compute) and another for transfer operations. Below that two threads of the application are shown. One main thread and one for a defragmentation process. For simplicity only one chunk allocator, and therefore defragmentation process, is used in this example. The main thread is responsible for updating the state of the application, recording a command buffer, and submitting all work related to the current frame to the GPU. The second thread is running the defragmentation for the chunk allocator.

### 4.4.1 User Controlled Defragmentation

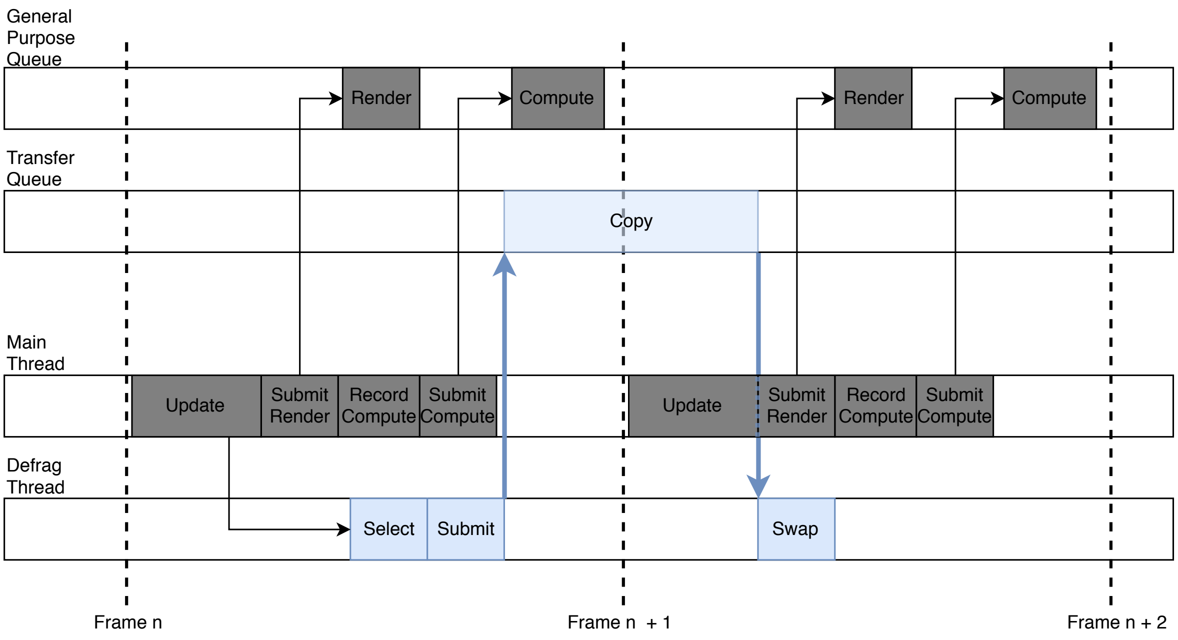


**Figure 4.6:** Performing user controlled defragmentation

Users may start and stop a full defragmentation process. When started, the defragmentation process is run until either the user cancels the process or all current operations have finished and no further operations are possible. After an operation has finished, new operations may be possible Users may want to perform the defragmentation during loading screens.

Figure 4.6 shows a potential problem with this approach. As part of the update in frame $n$ the defragmentation is started. After selection and submission the copy process starts immediately. During the same time the compute task is executed. Copying and running a compute command buffer simultaneously might degrade the performance of both operations.

### 4.4.2 Defragmentation During GPU Pauses

During gameplay the aforementioned method may not be viable. A full defragmentation process may delay the rendering of future frames and consequently cause stuttering. Therefore, depending on the game or application, a mechanism to time the execution of defragmentation operations is needed that does not interrupt the gameplay or negatively impact the user experience.

Figure 4.7 shows the overview of the CPU and GPU when using this approach. Shown in red, there are pauses during which the GPU is not performing any work. Executing operations during these pauses does not interrupt regular, frame-related command buffer execution. However, it is possible that sporadic or single-use command buffers may be performed at any time during the application lifetime. These are not considered here. The defragmentation thread performs the selection and submission of an operation during frame $n$. Using semaphores the

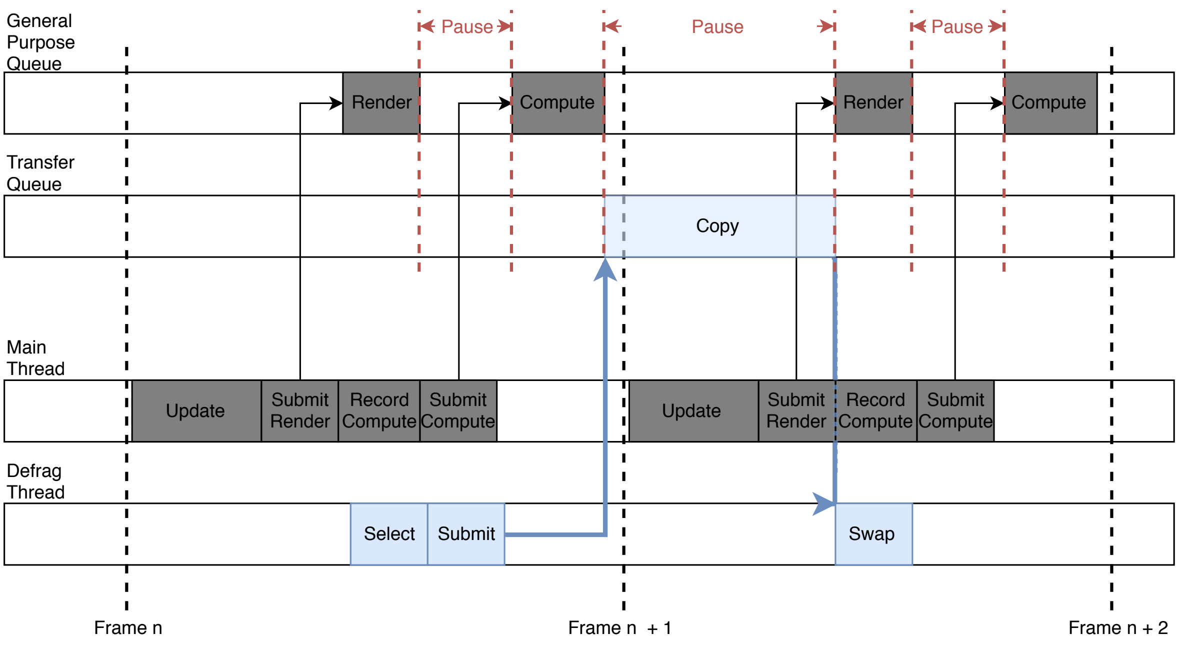


**Figure 4.7:** Typical tasks of a CPU and GPU during gameplay

actual copy is performed after the compute task has finished.

The start and end time of each command buffer is measured. Using these times, it is possible to determine all pauses for a frame $f$ $p(f) = p_0, \ldots, p_m$. The pause between frames is always associated with the previous frame.

In certain applications, like open-world games, the frame-timings can change drastically over the course of a few frames. This happens for instance when the player moves to an area that contains more geometry. Therefore, the last frames should be taken into account when determining the amount of time available for defragmentation. The process analyzes the last $k$ frames. This value is configurable by the user. With $n$ being the current frame, the time budget for defragmentation is calculated as follows:

$$t_{\text{defrag}} = \alpha \min_{i \in \{n-k, \ldots, k\}} p(i)$$

The factor $\alpha \in (0; 1]$ is a scaling factor to reduce the available time for defragmentation proportionally. This is useful, because it might be possible that the next frame has increased rendering times. In this process a conservative value of 0.8 was used. This might delay the defragmentation but allows for more variation of the render times.

The speed of a defragmentation process $v$ was recorded during the Copy stage. With this it is possible to fill the time budget $t_{\text{defrag}}$ with operations.

In the previous example only one defragmentation process was shown. When multiple processes are active the time $t_{\text{defrag}}$ has to be split between all processes. For this, time is alloted to the processes in a strict order. The first process may submit operations to the GPU until either no time or operations are left. After this the next process continues. If no time is left to distribute, no further defragmentation is performed. Any remaining operations must be submitted during future frames.

After the copying of the resource data of an operation has finished it is assigned to the *Swap* state. The operation now consists of a tuple $(s, t)$, with $s$ and $t$ being the source and target memory location respectively.

## 4.5 Swap & Free

As seen in figure 4.7, the swap step takes place after the copy operation has finished. The defragmentation process waits for the completion of the copy operations. After this two copies of the same resource exist, source $s$ and target $t$. From this point on, the target memory location will be used in command buffers. However, because of the swapchain the deallocation of the source memory location must occur at least $k$ frames later. The integer $k$ is the number of frames-in-flight of the swapchain (see section 2.9). During these frames both resources are potentially in use. The source memory location can be destroyed after $k$ frames have passed.

Swapping buffers only involves using the memory handle and offset of the target instead of the source. Image resources are bound to pipelines using descriptor sets(see sections 2.6 and 2.8.2). Although descriptor sets can be modified to use different resources, this is only possible while no command buffer uses them. For a few frames after the swap both resources are in use. Therefore, two descriptor sets must exist, one for the source location and one for the target location. As shown in section 2.6, only a limited number of descriptor sets may exist at any given time. Therefore, $k$ frames after the swap has occurred the descriptor set for the source will be destroyed.

The newly available free range, created by destroying the source, may lead to further defragmentation operations.

# Chapter 5

# Evaluation

In this chapter the setup, results, and interpretation of various tests of the memory management and defragmentation systems are shown. The hard- and software configuration of the computer that was used to perform the following evaluations is listed in Table 5.1.

| | |
|---|---|
| **CPU** | Intel Core i7-8700K @ 3.7GHz |
| **CPU-RAM** | 64 GiB |
| **GPU** | nVidia GeForce RTX 2070 (8 GiB RAM; driver 418.43) |
| **OS** | Ubuntu 18.04 (Kernel 4.15.0-47) |
| **Compiler** | GCC 7.3.0 |
| **Vulkan** | Version 1.1.106 |

**Table 5.1:** Hard- and software configuration of the test computer.

If not specified otherwise, the measurements are using a trimmed mean. The lowest and highest 1% of measurements are ignored for calculating the mean and standard deviation. The trimming was needed because most measurements are on small timescales, µs or ns and even minor interruptions caused by the GPU or CPU, like scheduling, can cause huge errors in the measured time.

One important motivation for the memory management system was improved performance. Section 5.1 contains a performance comparison of the chunk allocator and the unique allocator. The quality and speed of the fit-strategies of the chunk allocator (see section 5.2) is compared in section 5.2. Section 5.3 shows the improvements a defragmentation process has on the memory usage when using the chunk allocator.

## 5.1 Allocation and Deallocation Speed

When using the unique allocator a new piece of memory is allocated and deallocated for each resource. The following comparisons show the performance impact of memory allocations and

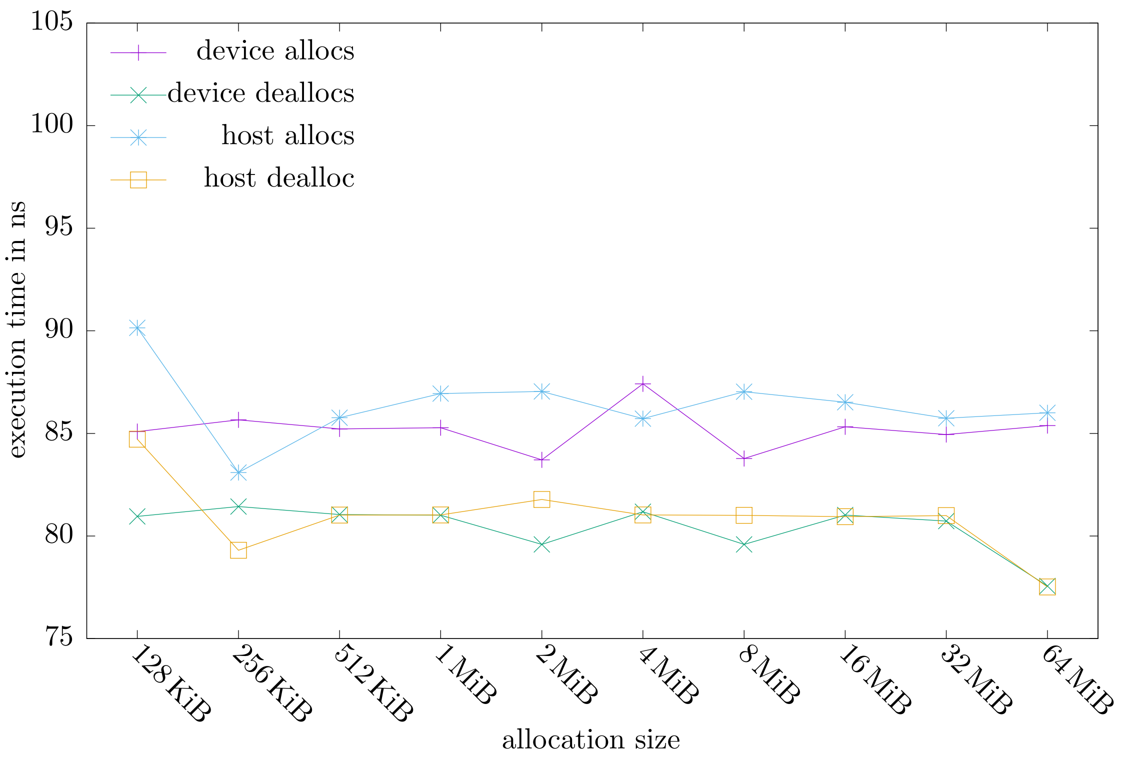


**Figure 5.1:** Plot of allocation and deallocation times using the chunk allocator

deallocations. Only the creation of resources is measured, not the time it takes to copy data to the resource.

### 5.1.1 Single Allocation

The first comparison compares the performance of the chunk and unique allocators (see sections 3.3 and 3.4) In this test a chunk allocator for vertex buffers with a chunk size of 64 MiB was used. The unique allocator is used to create vertex buffers as well. The test of the chunk allocator was performed with different sizes, ranging from 128 KiB to 64 MiB and each test was repeated 10,000 times. 64 MiB was the maximum because of the chunk size and larger allocations are not possible with this allocator. Unique allocator sizes range from 128 KiB to 512 MiB. Tests were performed using GPU memory and CPU memory. Figures 5.1 and 5.2 shows the results for the chunk and unique allocator respectively. The allocation of a chunk for the chunk allocator was not part of this test. Creating and destroying a single resource using the chunk allocator is much faster than with the unique allocator. Furthermore, the time it takes for allocations and deallocations with the chunk allocator is independent of the size.

For the unique allocator, a difference in allocation and deallocation times when using GPU memory or CPU memory is observed. The time for allocation and deallocation increases with the size when using GPU memory. For CPU memory the time stays roughly constant. When allocating memory of size $\leq$ 1 MiB the time for allocation and deallocation remains almost the

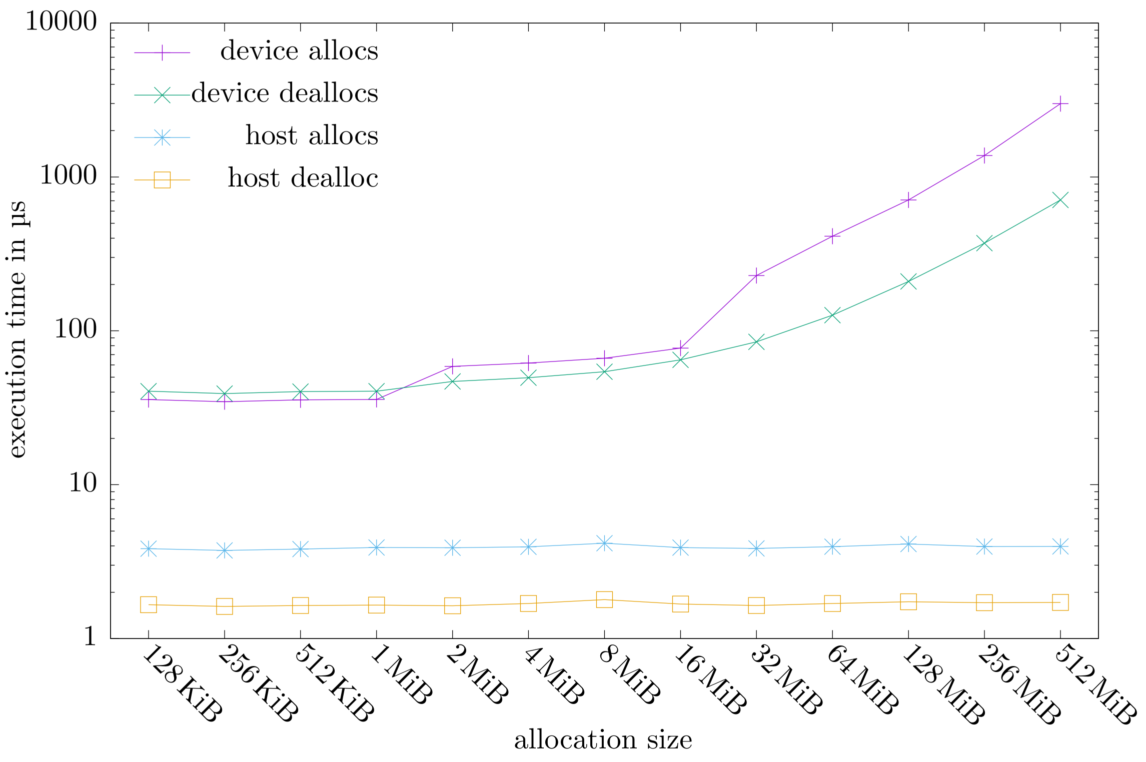


**Figure 5.2:** Plot of allocation and deallocation times using the unique allocator

same. Allocations of sizes between 2 MiB and 16 MiB also take almost the same time. When allocating memory of sizes >= 16 MiB the time for allocation and deallocation increases exponentially. No reason in the implementation of the unique allocator was found and it results from the equipment and configuration of the test system.

This test shows that allocating and deallocating memory using Vulkan takes most of the time when creating resources. Performance can be increased by using the sub-resource or shared memory approaches instead of the naive unique memory approach.

### 5.1.2 Multiple Allocations

The previous test only analyzes the speed of a single allocation and deallocation. However, users may wish to allocate multiple resources in quick succession before deallocating them. In the following tests 64 resources of the same size (ranging between tests from 128 KiB to 64 MiB) are allocated. After all resources were created they are all deallocated again. As with the test in section 5.1.1 the test is performed using the chunk and unique allocators as well as GPU memory and CPU memory. Each test was repeated 10,000 times and chunk allocators begin with no chunk allocated beforehand. In the figures 5.3 and 5.4 the value of the $x$-axis shows the size of a single allocation.

Results of the test for the chunk allocator are shown in figure 5.3. The time for allocations and deallocations of sizes up to 1 MiB is almost identical. Tests using these sizes fit inside one chunk. The deallocation when using sizes up to 1 MiB is noticeably faster than bigger

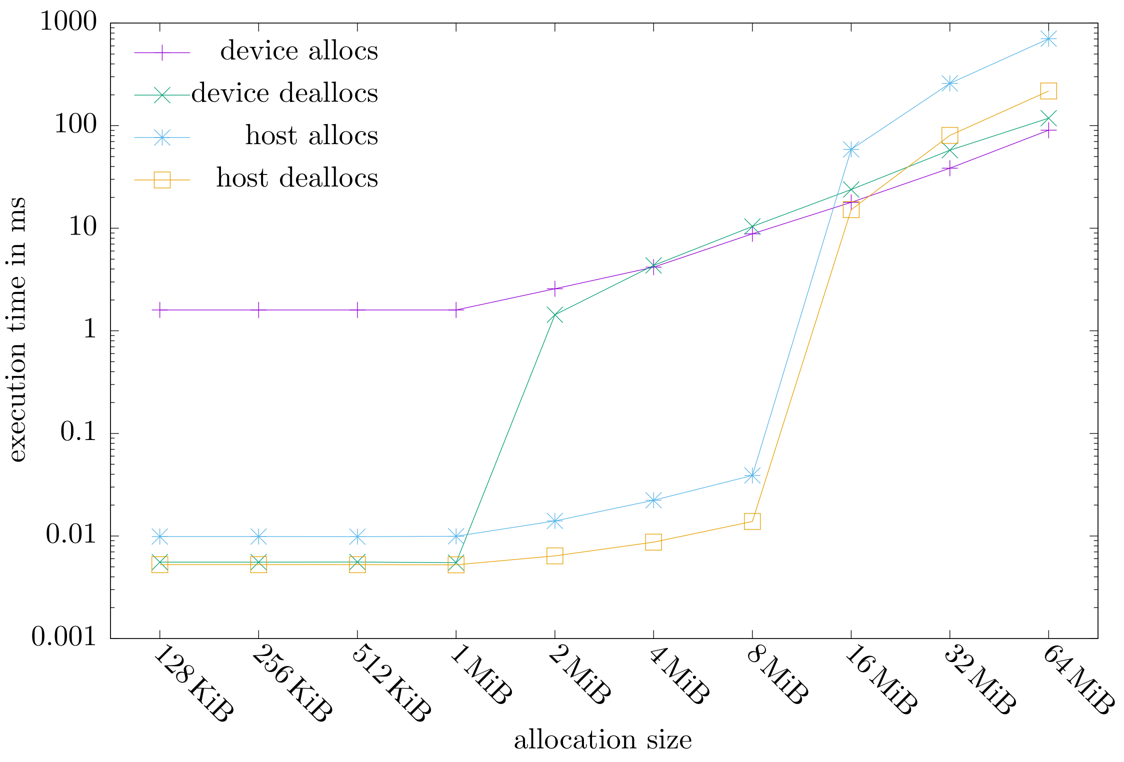


**Figure 5.3:** Plot for the chunk allocator of the execution time for test described in section 5.1.2.

sizes. After all resources were destroyed, chunks are deleted until only one free chunk remains. Because the tests with these sizes fit into a single chunk, no deallocation has to take place. Bigger sizes require multiple chunks and therefore chunks need to be destroyed. When using CPU memory a steep increase in execution time is noticeable when allocating 64 buffers with a size of 64 MiB each. This increase is probably caused by the OS.

Figure 5.4 shows the results for the unique allocator. The increase in allocation time for allocations larger or equal to 16 MiB using GPU memory is visible here as well. The time it takes to allocate memory of this size is of the same order of magnitude for the unique and chunk allocators. This happens because a single allocation fills a whole chunk and, therefore the number of chunks needed is the same as the number of memory allocations for the unique allocator.

## 5.2 Chunk Allocator Strategies

In this section the three strategies for finding free space that were implemented for the chunk allocator (see section 5.2) are compared. The three strategies are first-fit, best-fit, and worst-fit. Speed and efficiency is analyzed here. Because first-fit only has to search until the first free range is found it should perform faster in average than the other two strategies. Best-fit and worst-fit have to search through all free ranges if no free range of exact size can be found. The speed analysis compares the average allocation time using each strategy. For efficiency

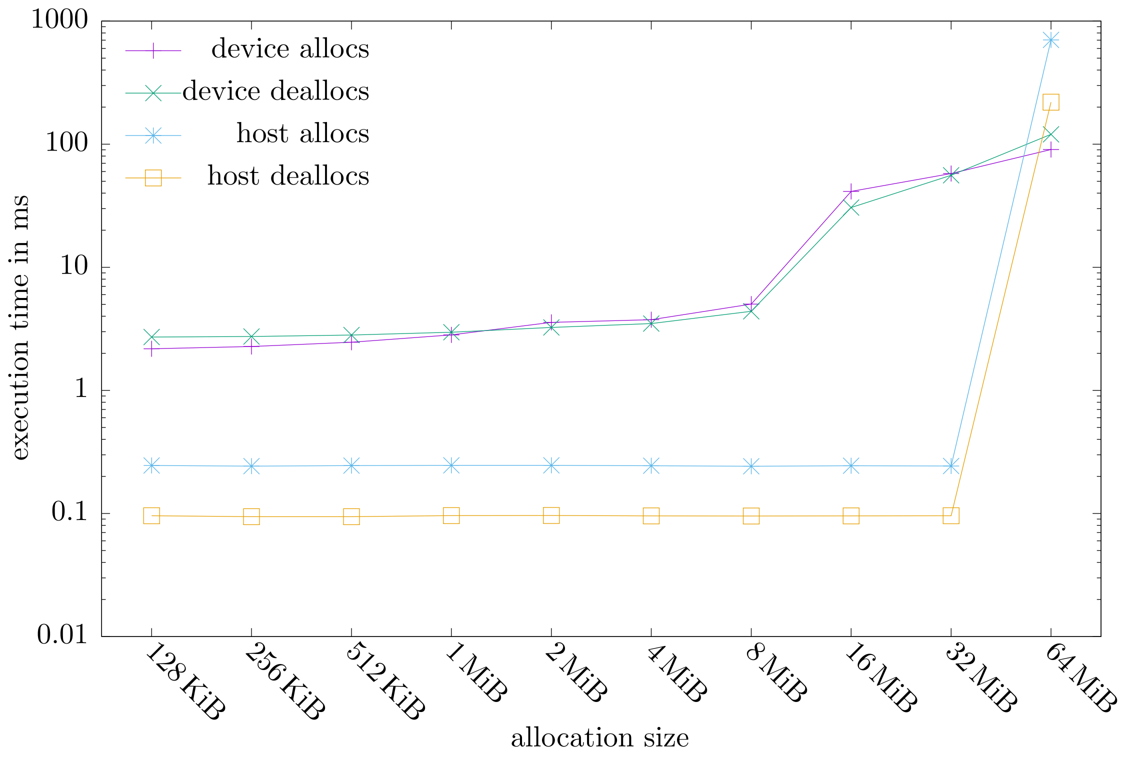


**Figure 5.4:** Plot for the unique allocator of the execution time for test described in section 5.1.2.

two criteria are analyzed. The first one is the fragmentation ratio $f$ (see equation (5.1)). The ratio gives a numerical value to the amount of fragmentation in a set of free ranges $F$. The value for $f$ is 0 if no fragmentation is present, i.e. there is only one free range. Fragmentation $f$ approaches 1 if there are many free ranges of similar sizes.

$$f = 1 - \frac{\max\limits_{f_i \in F} \text{size}(f_i)}{\sum\limits_{f_i \in F} \text{size}(f_i)} \tag{5.1}$$

The second benchmark is the amount of RAM needed to fit all allocations. Two strategies that have similar values for $f$ might require a different number of chunks (and therefore RAM). In this case, the strategy that uses fewer chunks is preferable.

| | $\bar{t}_{\text{search}}$ (%) | $\bar{t}_{\text{alloc}}$ (%) | $f$ | $\overline{n_c}$ (%) |
|---|---|---|---|---|
| first-fit | 0.075 µs (100%) | 4.374 µs (100%) | 0.284 | 26.69 (100%) |
| best-fit | 0.198 µs (205%) | 4.443 µs (102%) | 0.173 | 26.71 (100%) |
| worst-fit | 0.216 µs (245%) | 4.727 µs (108%) | 0.181 | 29.64 (111%) |

**Table 5.2:** Fragmentation and performance comparison of three fit-strategies.

Each test consists of a sequence of allocation rounds. In these rounds a certain number of resources are allocated. The resource-size and -number is identical for each strategy. Between each round a few deallocations occur. Ten-thousand different randomized sequences were tested with each strategy. The results are shown in table 5.2. The first column shows the name of the used fit-strategy. The following two columns show the average times of the strategy and allocation times. In parenthesis behind each value the value is compared to the first-fit strategy. $f$ is the average fragmentation ratio of all chunks. The value $\overline{n_c}$ is the average number of chunks still allocated after each test has ended. When only comparing the search times, best-fit and worst-fit take on average more than twice the time compared to first-fit. However, when the whole allocation process is taken into account the difference is less than 10%. Therefore, performance concerns with using different fit strategies are of a lower concern. But the results concerning the fragmentation and memory usage show that first-fit is superior in this test case.

## 5.3 Defragmentation

The defragmentation process aims to reduce fragmentation and memory usage by compacting memory allocations towards the beginning of the allocator. Tests to verify this follow.

Like the previous tests for the fit-strategy, multiple tests each consisting of several rounds of buffer-allocations are performed. Between each round a random number of buffers is deallocated. Each round consisted of 1 to 5 allocations and deallocations. The size of each allocation was between 256 KiB and 16 MiB. The chunk size was chosen to be 64 MiB. A small pause of 10 frames between each round was added. This allows the defragmentation process to run. After each round the current average fragmentation $f$ (see equation (5.1)) is measured. Additionally the number of currently allocated chunks is recorded.

In total 25 tests were performed with 50 allocation rounds each. There are four values that determine the behaviour and interaction of the location and free range heuristics. The values are $\alpha_{\text{loc}}$, a scaling factor for the location heuristic described in section 4.2.1, and the three values $s_{\text{small}}, s_{\text{limit}} and w_{\text{free}}$ for the free range heuristic (see section 4.2.2). Each set of tests was performed for varying combinations for these four values. Additonally the test set were performed with defragmentation but without heuristics, and without defragmentation.

| $\alpha_{\text{loc}}$ \ $w_{\text{free}}$ | 0 | 0.25 | 0.5 | 0.75 | 1.0 |
|---|---|---|---|---|---|
| 0 | 0.267 | 0.257 | 0.253 | 0.254 | 0.259 |
| 0.25 | 0.246 | 0.254 | 0.252 | 0.256 | 0.253 |
| 0.5 | 0.246 | 0.255 | 0.256 | 0.254 | 0.252 |
| 0.75 | 0.246 | 0.257 | 0.256 | 0.255 | 0.258 |
| 1 | 0.246 | 0.252 | 0.253 | 0.256 | 0.258 |

**(a)** Average fragmentation $\overline{f}$

| $\alpha_{\text{loc}}$ \ $w_{\text{free}}$ | 0 | 0.25 | 0.5 | 0.75 | 1.0 |
|---|---|---|---|---|---|
| 0 | 2.33 | 2.31 | 2.31 | 2.31 | 2.31 |
| 0.25 | 2.28 | 2.28 | 2.3 | 2.3 | 2.3 |
| 0.5 | 2.28 | 2.29 | 2.29 | 2.3 | 2.29 |
| 0.75 | 2.28 | 2.29 | 2.28 | 2.29 | 2.29 |
| 1 | 2.28 | 2.3 | 2.29 | 2.28 | 2.29 |

**(b)** Average chunk count $\overline{n_c}$

**Table 5.3:** Defragmentation test results for $s_{\text{small}} = 0.01\%$ and $s_{\text{limit}} = 0.1\%$

Results for the tests with enabled heuristics are found in tables 5.3 and 5.4. The first two tables show the results when using $s_{\text{small}} = 0.01\%$ and $s_{\text{limit}} = 0.1\%$. The second tables show the result when using $s_{\text{small}} = 0.1\%$ and $s_{\text{limit}} = 1\%$. Independent of these values, the best

| $\alpha_{\text{loc}}$ \ $w_{\text{free}}$ | 0 | 0.25 | 0.5 | 0.75 | 1.0 |
|---|---|---|---|---|---|
| 0 | 0.267 | 0.258 | 0.253 | 0.253 | 0.254 |
| 0.25 | 0.246 | 0.25 | 0.255 | 0.253 | 0.254 |
| 0.5 | 0.246 | 0.258 | 0.254 | 0.255 | 0.252 |
| 0.75 | 0.246 | 0.257 | 0.256 | 0.251 | 0.254 |
| 1 | 0.246 | 0.255 | 0.253 | 0.251 | 0.259 |

**(a)** Average fragmenation $\overline{f}$

| $\alpha_{\text{loc}}$ \ $w_{\text{free}}$ | 0 | 0.25 | 0.5 | 0.75 | 1.0 |
|---|---|---|---|---|---|
| 0 | 2.33 | 2.31 | 2.32 | 2.3 | 2.31 |
| 0.25 | 2.28 | 2.29 | 2.3 | 2.29 | 2.29 |
| 0.5 | 2.28 | 2.29 | 2.29 | 2.29 | 2.28 |
| 0.75 | 2.28 | 2.29 | 2.29 | 2.29 | 2.3 |
| 1 | 2.28 | 2.29 | 2.3 | 2.3 | 2.29 |

**(b)** Average chunk count $\overline{n_c}$

**Table 5.4:** Defragmentation test results for $s_{\text{small}} = 0.1\%$ and $s_{\text{limit}} = 1\%$

results are achieved when using the location heuristic exclusively. However, using only the free range heuristic is still better than using no heuristic at all.

In table 5.5 a comparison of the tests when defragmentation is disabled, enabled without heuristics, and when the optimal values (according to the previous results) are chosen. If the heuristics are disabled, all operations have a weight $\Delta w$ of 0. Operations are executed in the exact order they were found. The fragmentation ratio decreases when using defragmentation. The average number of chunks $n_c$ stays almost the same. This might happen because the tests use a small number of chunks. When using more chunks the difference might be more visible. The difference between defragmenatation enabled and disabled heuristics is smaller.

These tests show, that the defragmenation process improves the fragmenation produced by the chunk allocator. The results show that using both heuristics is not favorable. Using the location heuristic alone shows the best results.

| Defragmentation | $\alpha_{\text{loc}}$ | $w_{\text{free}}$ | $\overline{f}$ | $\overline{n_c}$ |
|---|---|---|---|---|
| disabled | — | — | 0.318 | 2.31 |
| enabled | 0 | 0 | 0.267 | 2.3 |
| enabled | 1 | 0 | 0.246 | 2.28 |

**Table 5.5:** Comparison between disabled and enabled defragmentation (with and without heuristics)

# Chapter 6

# Conclusions

## 6.1 Summary

Vulkan is a very verbose and explicit API. Many tasks that were performed by the driver in other, older APIs must now be done by programmers for each application.

The memory allocator shown in chapter 3 provides a simple interface to allocate and deallocate memory for Vulkan. The tests in chapter 5 show that using the more complex approaches, i.e. shared memory and sub-resources, provide enormous benefits in terms of speed when compared to the unique allocation approach. Even applications that do not perform dynamic memory allocations and deallocations during runtime benefit from the increased speed. Although the defragmentation process incurs runtime costs, it can lead to better memory utilization for the sub-resource and shared memory approaches.

## 6.2 Future work

The systems developed as part of this thesis have so far only been tested and used in small scale programs and test cases. Tests using larger applications, including games, should be performed in the future.

Using the defragmentation process reduces the fragmentation of the chunk memory allocator. However, when both heuristics are used, the result is worse than using only the address-based heuristic. Therefore, alternative heuristics should be developed that potentially improve the defragmentation process further.

Vulkan allows sparse bindings of memory. This means that resources do not need to be stored contiguously. Instead they can be split into multiple parts of equal size. Furthermore, this feature allows that parts are moved to different memory locations. A future version of the memory allocator should support this kind of memory allocation. Currently roughly 66% of all Vulkan-enabled devices support this feature. [15].

The functionality to delay defragmentation operations until the GPU is idling could be generalized. This would allow to generally delay command buffers until a GPU pause is detected. Tasks, like loading new parts of a level, could potentially be delayed in order to reduce or eliminate interruptions of other work on the GPU.

Other approaches, like buddy systems [8] or slab allocation, should be implemented and compared to the allocations system shown in chapter 3. The comparison should include the amount of fragmentation as well as the speed of allocation and deallocation.

Currently the unique allocator is used when allocations cannot be fulfilled by the chunk allocator, due to their size. If the size of an allocation approaches the chunk size, it might be useful to still allocate it with the unique allocator. Large allocations sometimes exclusively occupy a chunk and consequently increase the time it takes for free ranges to be found. Furthermore, they are seldom possible candidates for defragmentation. Therefore, it might be useful to introduce a limit to the chunk allocator above which allocations are performed with the unique allocator instead.

The memory process shown in chapter 3 assumes that enough RAM is available at all times. In real world scenarios this assumption cannot be made. If a type of memory cannot fulfill a memory request, another, potentially slower, memory type should be used instead.

The Vulkan Memory Allocator[1] implements memory management for Vulkan. A comparison of features and performance should be made.

[1]`https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator`

# List of Figures

# List of Tables

# Bibliography

[1] Jeff Bonwick et al. “The slab allocator: An object-caching kernel memory allocator.” In: *USENIX summer*. Vol. 16. Boston, MA, USA. 1994.

[2] Jonathan Corbet. *The SLUB allocator*. Apr. 11, 2007. URL: `https://lwn.net/Articles/229984/` (visited on 05/13/2019).

[3] The Khronos® Group. *OpenGL Overview*. 2019. URL: `https://www.khronos.org/opengl/` (visited on 04/13/2019).

[4] The Khronos® Group. *Vulkan Overview*. 2019. URL: `https://www.khronos.org/vulkan/` (visited on 04/13/2019).

[5] Chris Hebert and Christoph Kubisch. *Vulkan Memory Management*. 2016. URL: `https://developer.nvidia.com/vulkan-memory-management` (visited on 04/09/2019).

[6] Mark S. Johnstone and Paul R. Wilson. “The memory fragmentation problem: Solved?” In: *Acm Sigplan Notices*. Vol. 34. 3. ACM. 1998, pp. 26–36.

[7] Michael Kircher and Prashant Jain. “Pooling pattern”. In: *Proceedings of EuroPlop 2002* (2002).

[8] Kenneth C Knowlton. “A fast storage allocator”. In: *Communications of the ACM* 8.10 (1965), pp. 623–624.

[9] Donald Ervin Knuth. “The Art of Computer Programming”. In: 3rd ed. Vol. 1. Addison-Wesley, 1977. Chap. 2.5 Dynamic Storage Allocation, pp. 435–456. ISBN: 0-201-89683-4.

[10] Bernhard Korte et al. *Combinatorial optimization*. Vol. 2. Springer, 2012.

[11] Brian Randell. “A note on storage fragmentation and program segmentation”. In: *Communications of the ACM* 12 (July 1969), 365–ff. DOI: `10.1145/363156.363158`.

[12] Andrew S. Tanenbaum and Herbert Bos. *Modern operating systems*. 4th ed. Pearson, 2015. ISBN: 978-1292061429.

[13] The Khronos® Group. *OpenGL ES Overview*. 2019. URL: `https://www.khronos.org/opengles/` (visited on 04/13/2019).

[14] The Khronos® Vulkan Working Group. *Vulkan® 1.1.106 - A Specification*. 2019. URL: `https://www.khronos.org/registry/vulkan/specs/1.1/html/vkspec.html` (visited on 03/31/2019).

[15] Sascha Willems. *Vulkan Hardware Database - Value distribution for maxMemoryAllocationCount*. 2019. URL: `https://vulkan.gpuinfo.org/displaydevicelimit.php?name=maxMemoryAllocationCount` (visited on 05/07/2019).

[16] Paul R. Wilson et al. “Dynamic storage allocation: A survey and critical review”. In: *Memory Management*. Springer, 1995, pp. 1–116.

# Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Ich bin damit einverstanden, dass die Arbeit veröffentlicht wird und dass in wissenschaftlichen Veröffentlichungen auf sie Bezug genommen wird.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Graphische Datenverarbeitung, wird ein (nicht ausschließliches) Nutzungsrecht an dieser Arbeit sowie an den im Zusammenhang mit ihr erstellten Programmen eingeräumt.

Erlangen, den 15.05.2019

(Peter Eichinger)