Digital Dragons

KRAKOWSKI PARK TECHNOLOGICZNY

Memory Management in the APEX Engine

Lou Kramer
AMD

Daniel Isheden
Avalanche Studios Group

Digital Dragons

KRAKOWSKI PARK TECHNOLOGICZNY

Lou Kramer

AMD

Daniel Isheden

Avalanche Studios Group


AMD

AGENDA

Basics about memory management and tooling

- Memory types
- Committed / Placed Resources
- Over-Commitment
- Application, Driver, Operating System
- Tools

Memory Management in the APEX Engine

- The APEX Engine
- Resource types
- Problems & Solutions

This talk will focus only on PC with a dedicated GPU

API specific terminology is coloured in

Direct3D® 12

Vulkan®

MEMORY TYPES – RECAP

Physically:

VB, IB, CBV, SRV, RT, DS, UAV, ...

Video RAM (VRAM)

Memory Pool L1
Device Local

Copy... - PCIe®

System RAM (SysRAM)

Memory Pool L0

void* mappedPtr;

MEMORY TYPES – RECAP

Logically:

VB, IB, CBV, SRV, RT, DS, UAV, ...

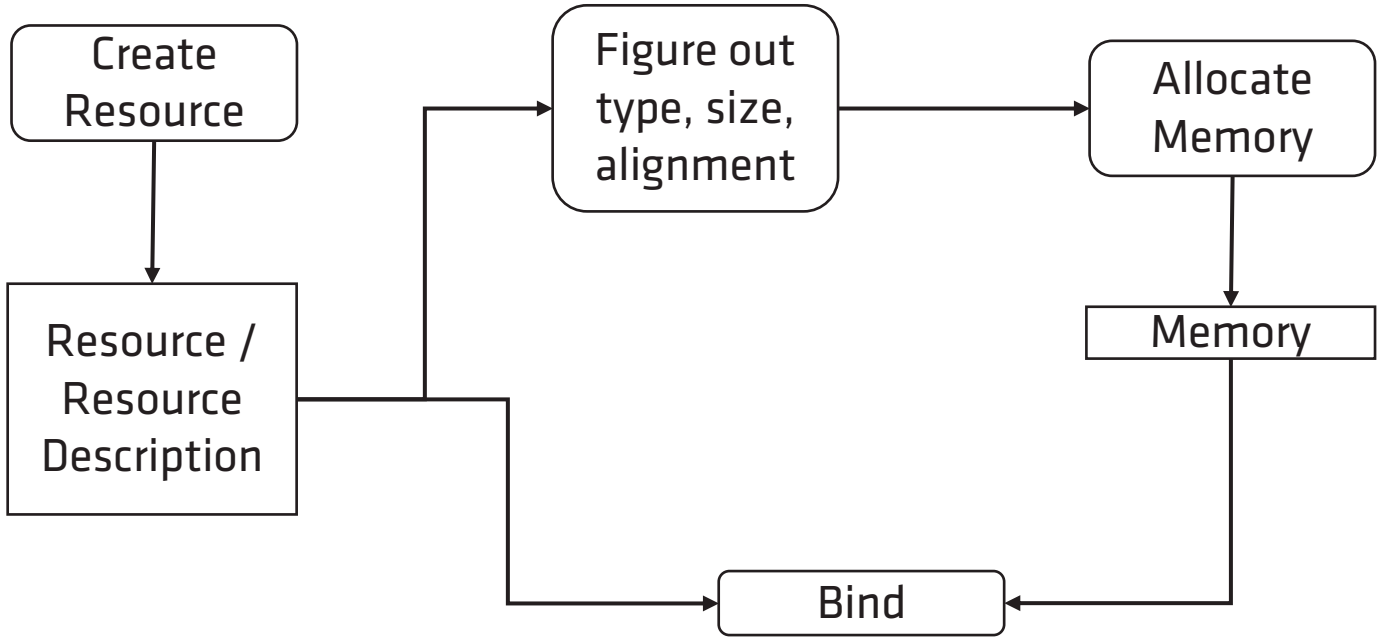
Default Heap
Device Local

void* mappedPtr;

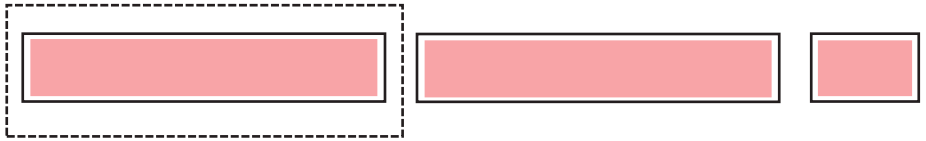
Upload Heap
Host Visible |
Host Coherent |

Readback Heap
Host Visible |
Host Coherent |
Host Cached

CREATE A RESOURCE



Method 1: Allocate just what's needed for the resource



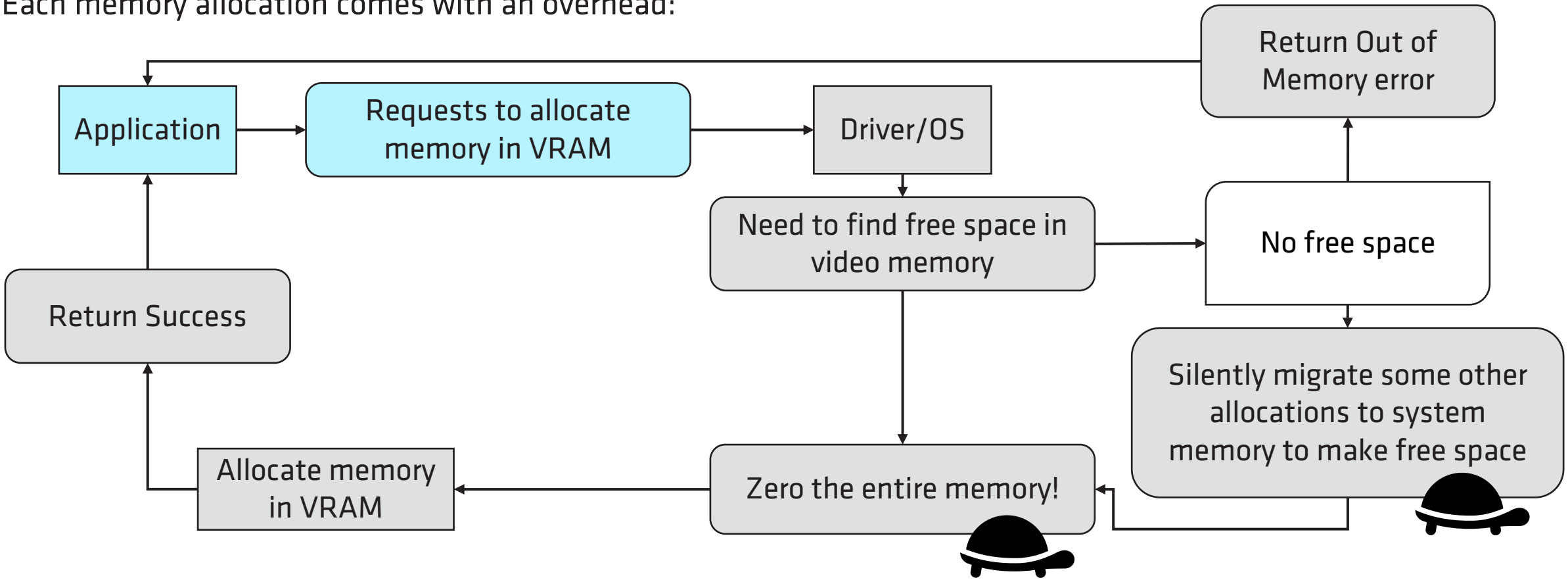
→ Committed Resource
→ Dedicated Allocation

Method 2: Allocate large blocks (e.g. 256 MiB) and sub-allocate

→ Placed Resource

COMMITTED RESOURCE

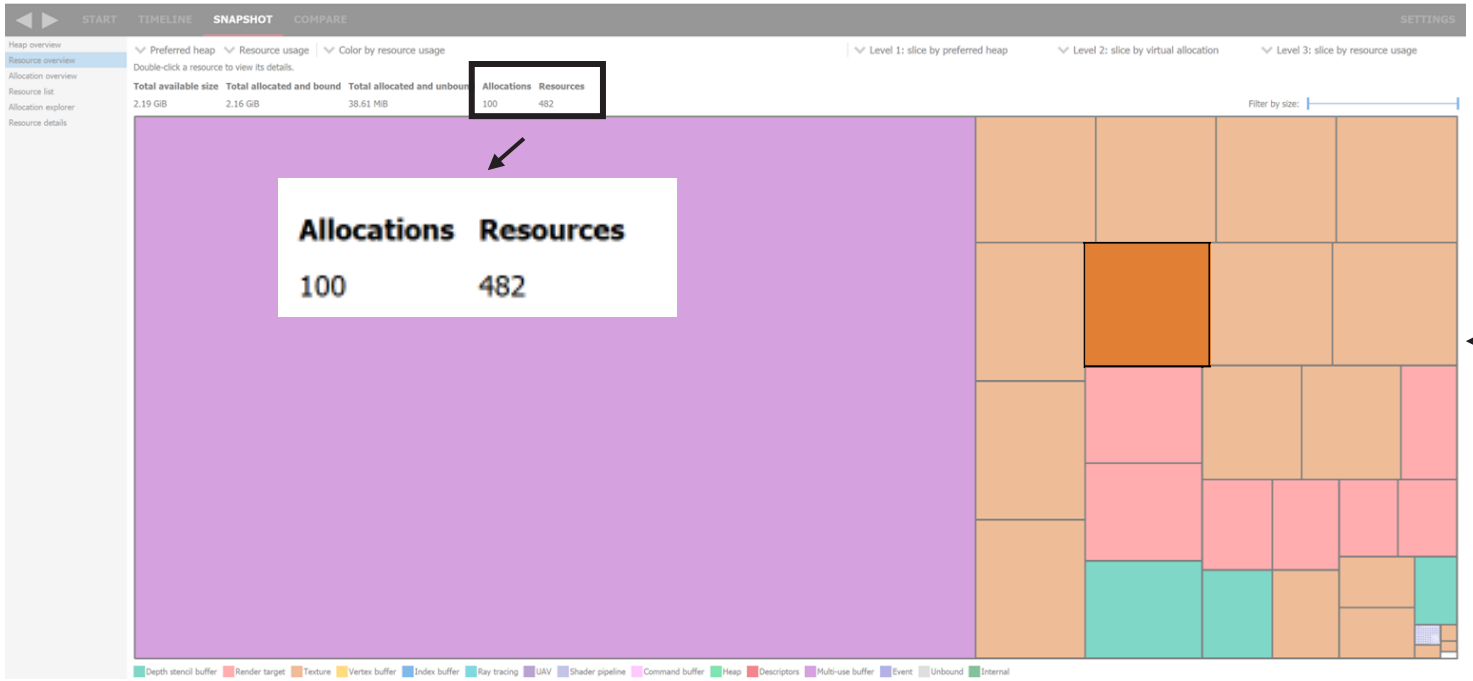
- A separate memory allocation per committed resource
- Each memory allocation comes with an overhead:



- This can take up to several milliseconds (or even seconds)

COMMITTED RESOURCE

- Once the memory is allocated and the resource is created,
- they don’t work any slower or different than a placed resource

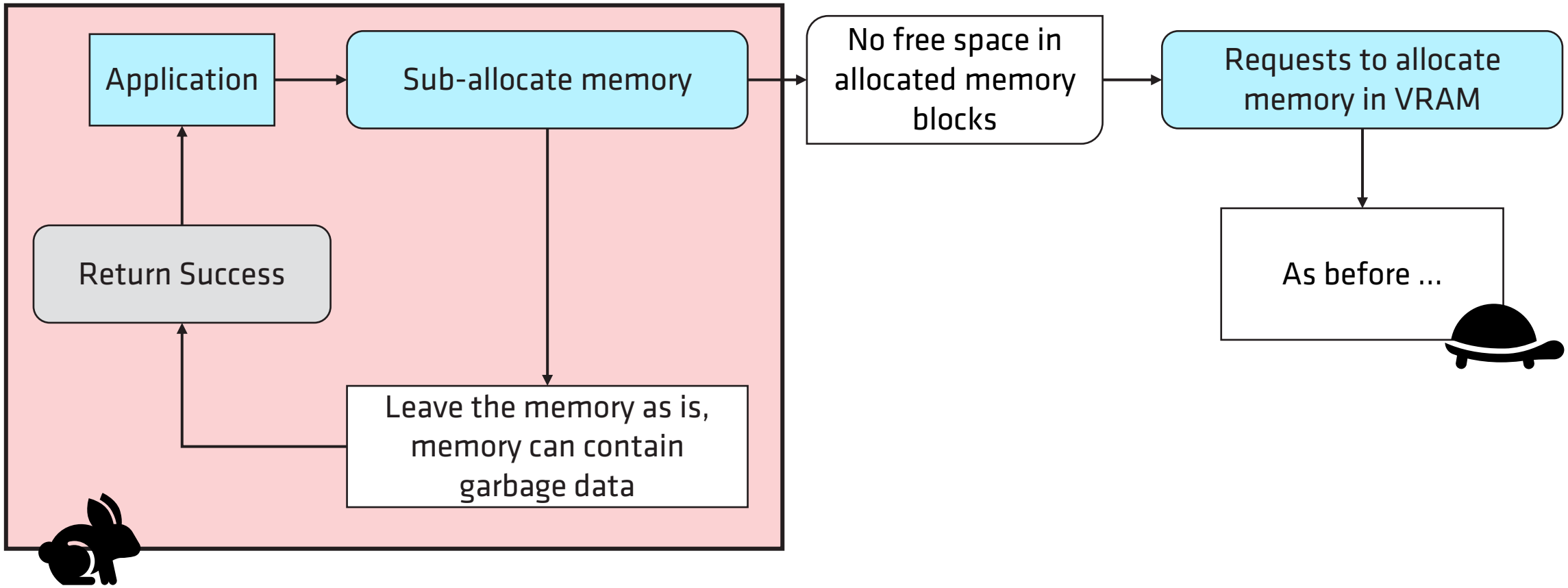


100 Allocations total

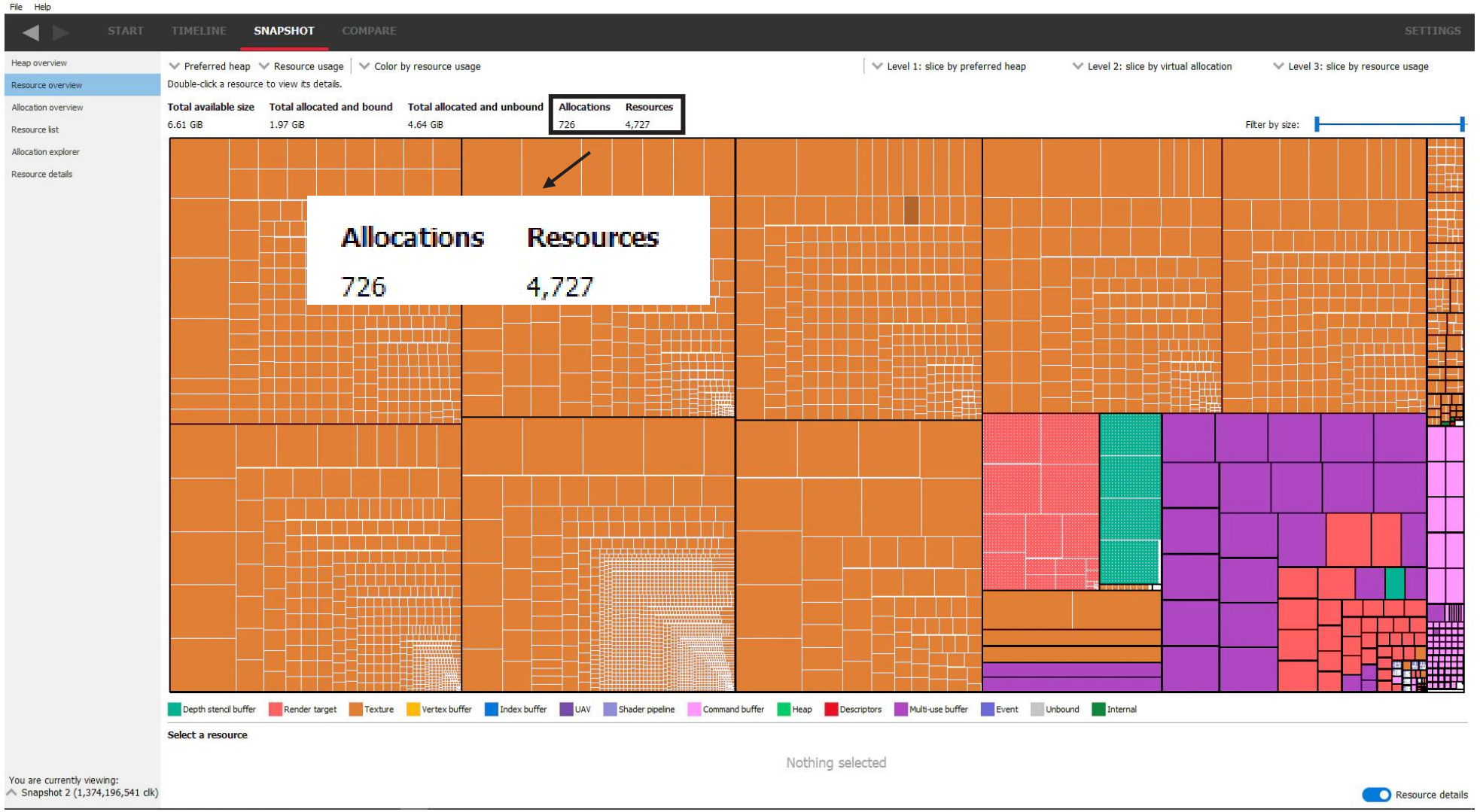
- On Vulkan®, there is also a limited maximum number of allocations (e.g., 4096)

PLACED RESOURCE

- A large memory block (e.g., 256 MiB) is allocated when needed
- Sub-allocate parts of them for the placed resource



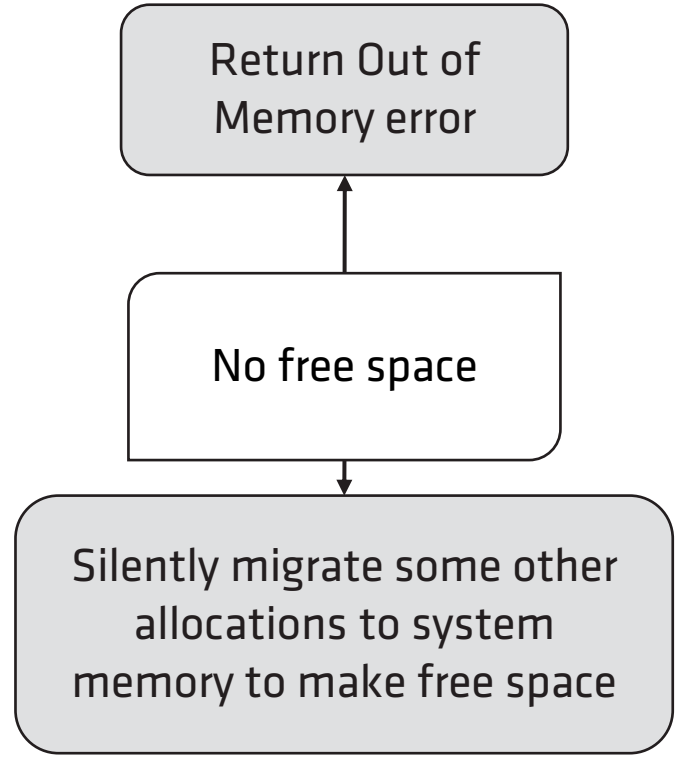
PLACED RESOURCE



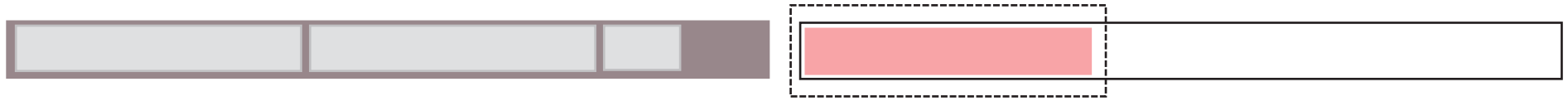
OVER-COMMITMENT

- Over commitment – when your video memory is full
  - New allocations may fail
  - Existing allocations can be migrated to system memory
  - → performance degradation

- Some resources tend to have a high impact on performance
- You really don't want them to migrate to SysRAM, e.g., render targets



- When silently migrated, the whole allocation is affected and all resources associated with it
  - Not just a single resource. Not just a single memory page

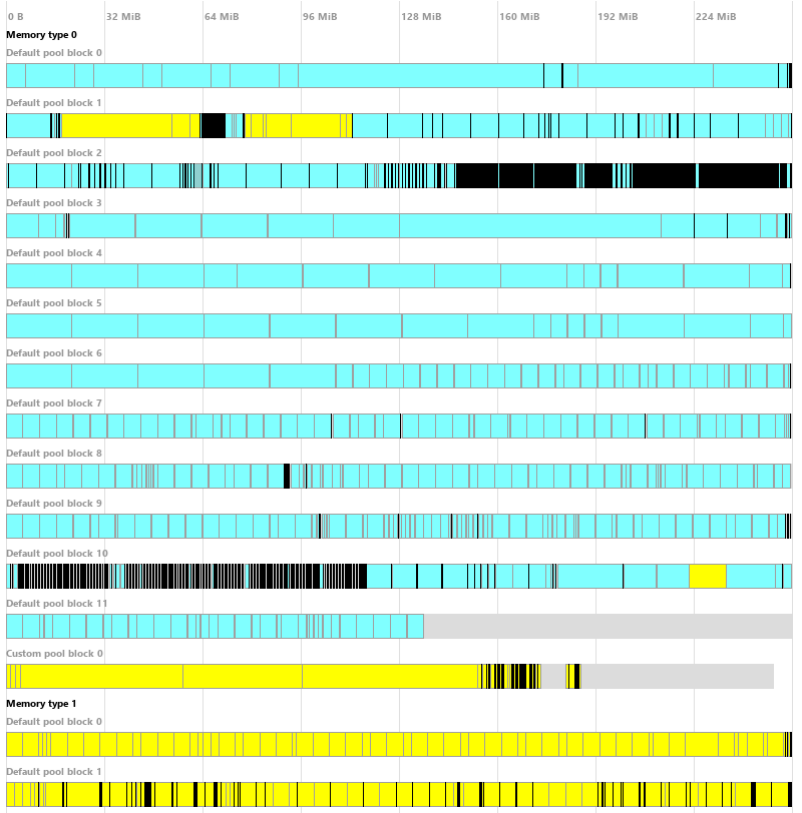


OVER-COMMITMENT

- How we can prevent a performance critical resource to get migrated to system memory?
- There is no explicit control
- No way to query for when and what

Application

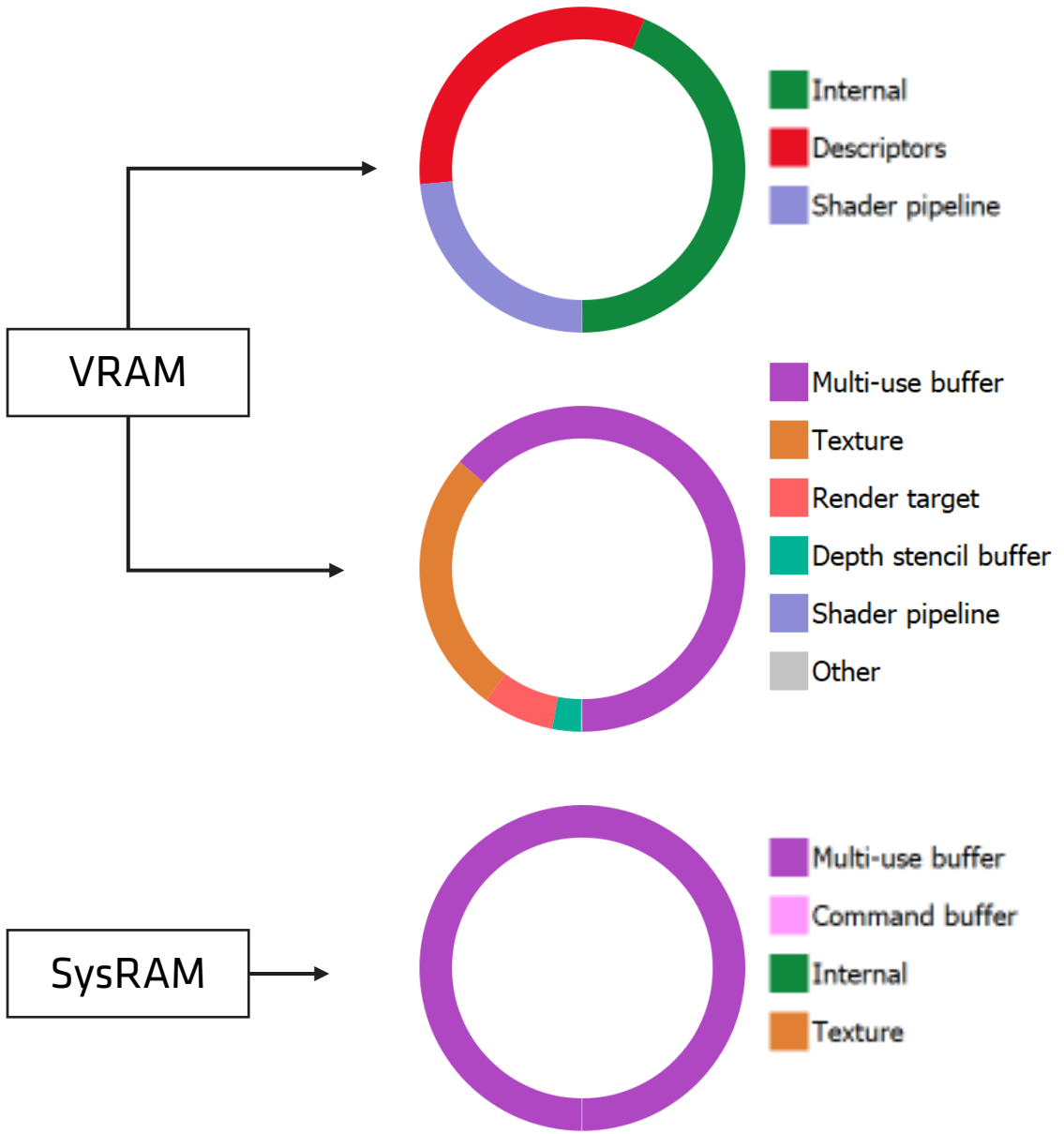
- Creates resources
- Destroys resources
- Sets a preferred heap
- Sets residency priority
- Knows about all resources and how they are used
- Can use Evict/Make Resident on Direct3D®12
  - Moving things is a fairly expensive operation and can cause stuttering



OVER-COMMITMENT

Driver

- Sets also residency priorities
- The driver allocates memory for implicit resources
  - Command buffers
  - Descriptors
  - Shader pipelines
  - Internal resources

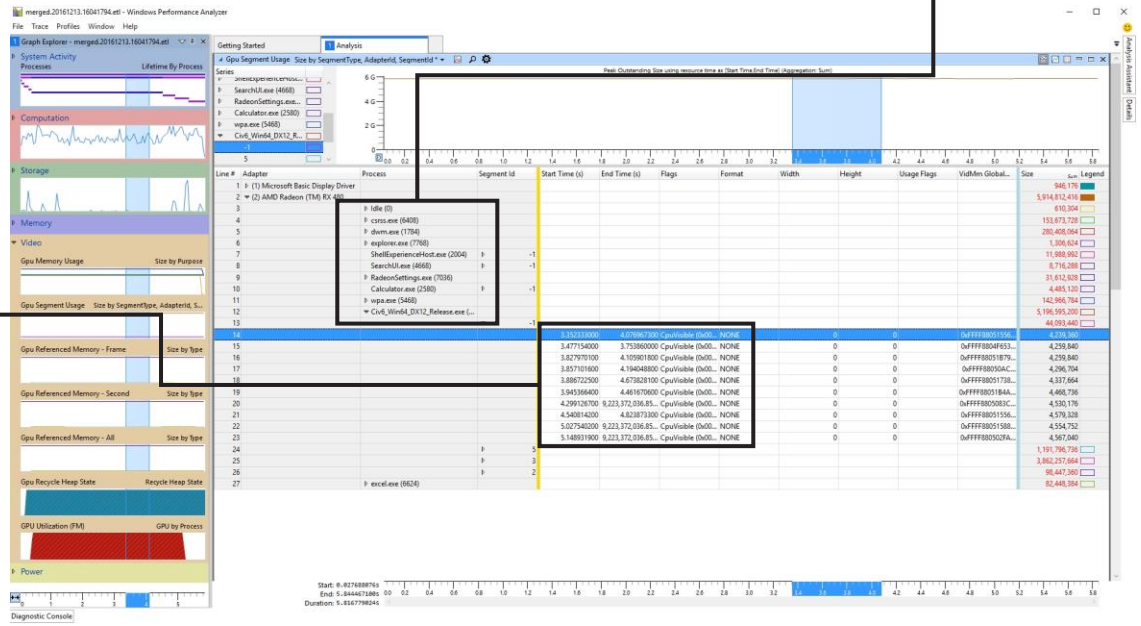


OVER-COMMITMENT

Operating System (Microsoft®'s Video Memory Manager)

- Knows about other applications running in parallel
  - Ensures that each process receives a fair share[1]
- Can migrate memory blocks to system memory
  - Ensures that the transition of video to system memory is invisible to the application[2]

-1				
	3.352333000	4.076967300	CpuVisible (0x00...	NONE
	3.477154000	3.753860000	CpuVisible (0x00...	NONE
	3.827970100	4.105901800	CpuVisible (0x00...	NONE
	3.857101600	4.194048800	CpuVisible (0x00...	NONE
	3.886722500	4.673828100	CpuVisible (0x00...	NONE
	3.945366400	4.461670600	CpuVisible (0x00...	NONE
	4.299126700	9,223,372,036.85...	CpuVisible (0x00...	NONE
	4.540814200	4.823873300	CpuVisible (0x00...	NONE
	5.027540200	9,223,372,036.85...	CpuVisible (0x00...	NONE
	5.148931900	9,223,372,036.85...	CpuVisible (0x00...	NONE



[1] https://docs.microsoft.com/en-us/windows-hardware/drivers/display/using-memory-segments-to-describe-the-gpu-address-space
[2] https://docs.microsoft.com/en-us/windows-hardware/drivers/display/mapping-virtual-addresses-to-a-memory-segment

OVER-COMMITMENT

We can try to increase the likelihood for a performance critical resource to stay in VRAM by:

Having **enough free space** on the VRAM

- Query for budget and stick to it
- DXGI_QUERY_VIDEO_MEMORY_INFO
- VK_EXT_memory_budget → VkPhysicalDeviceMemoryBudgetPropertiesEXT

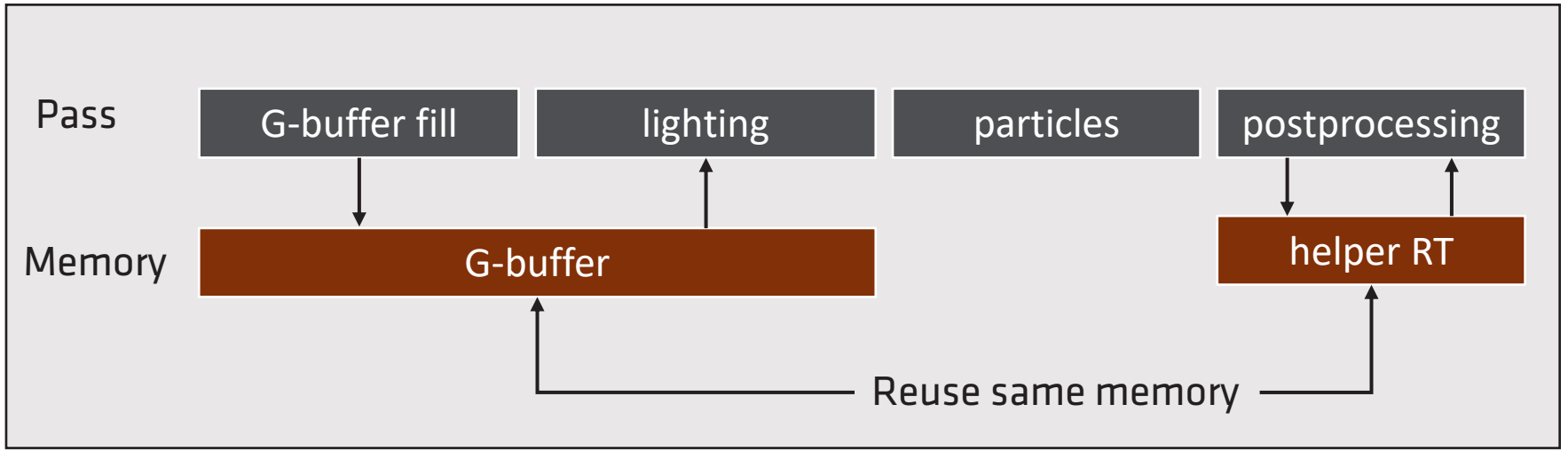
- Query regularly for usage and budget
- Try to stay at usage < budget

OVER-COMMITMENT

We can try to increase the likelihood for a performance critical resource to stay in VRAM by:

Having **enough free space** on the VRAM

- Free or evict memory blocks when possible before creating new resources
- Alias Memory



OVER-COMMITMENT

We can try to increase the likelihood for a performance critical resource to stay in VRAM by:

Having **enough free space** on the VRAM

- Place VB, IB, CBV that are read only once by the GPU to the upload heap (system memory)
  - Save memory for another copy of the resource
  - Can even save time that's needed for the transfer
  - Reading will be slower though
  - Good for buffers

VB, IB, CBV ...

Upload Heap
Host Visible |
Host Coherent |

void* mappedPtr;

OVER-COMMITMENT

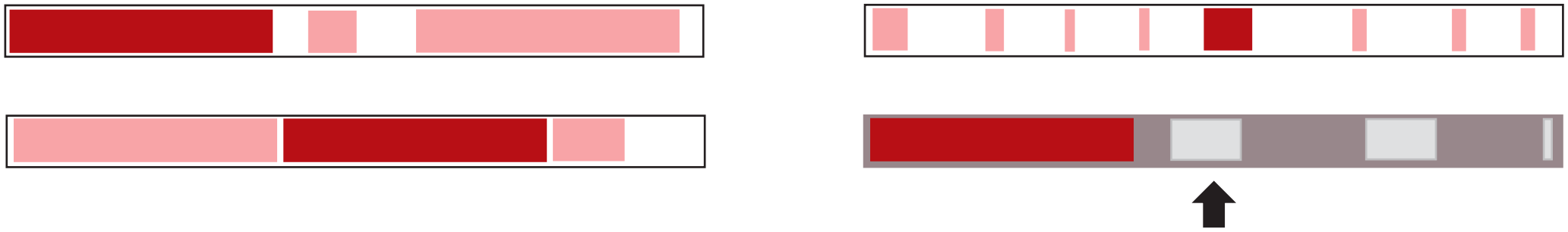
: performance critical resource

: not performance critical resource

We can try to increase the likelihood for a performance critical resource to stay in VRAM by:

Create performance critical resources as committed resources

- After creating critical resources as committed, set them high residency priority
- No need to allocate a new big chunk of memory – just the amount that is actually required gets allocated
  - increases chance there is still enough free space
- Critical resources are not scattered in different large allocated memory blocks
  - Whole memory block gets evicted and thus, everything that's in it
  - If every memory block contains a critical resource, you will always loose



OVER-COMMITMENT

We can try to increase the likelihood for a performance critical resource to stay in VRAM by:

Try out different memory block sizes for your placed resources

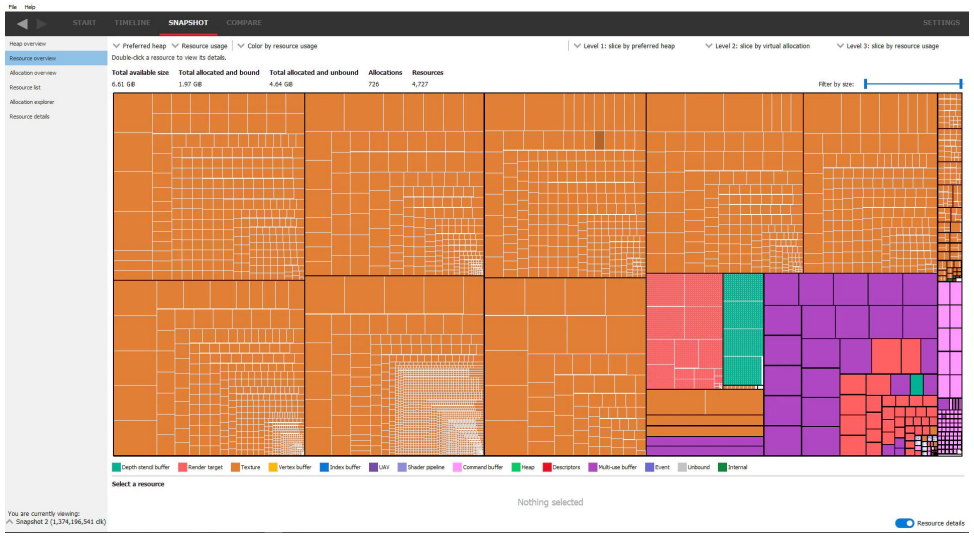
- The optimal size can vary depending on your specific case (e.g., 256 MiB, 64 MiB, ...)

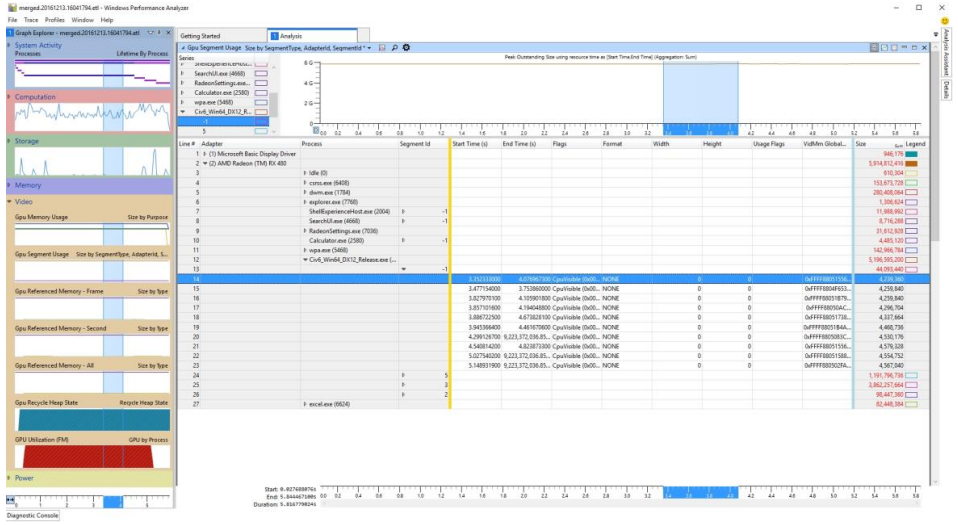
If your memory is too fragmented:

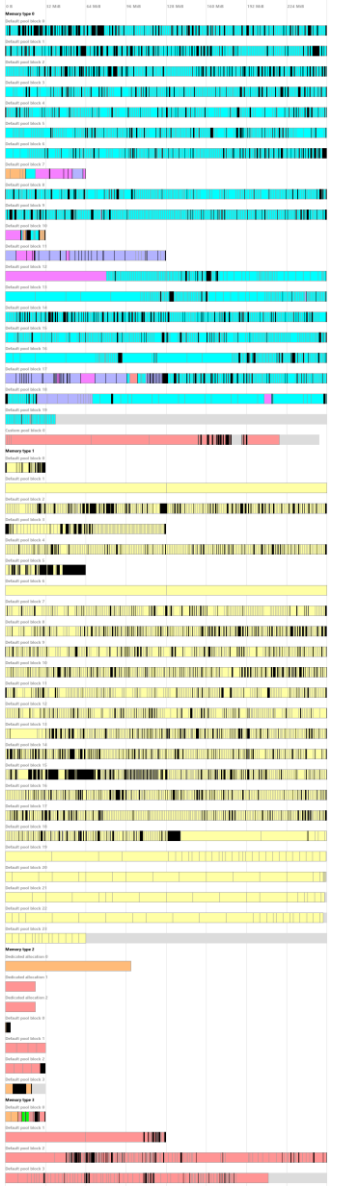
- Create custom pools for certain resources
  - e.g., resources that should certainly be freed when unloading a level
- When streaming resources in and out, fragmentation is expected
  - try to defragment

TOOLS

- VMA/D3D12MA JSON Dumps
- Radeon™ Memory Visualizer
- Windows® Performance Analyzer







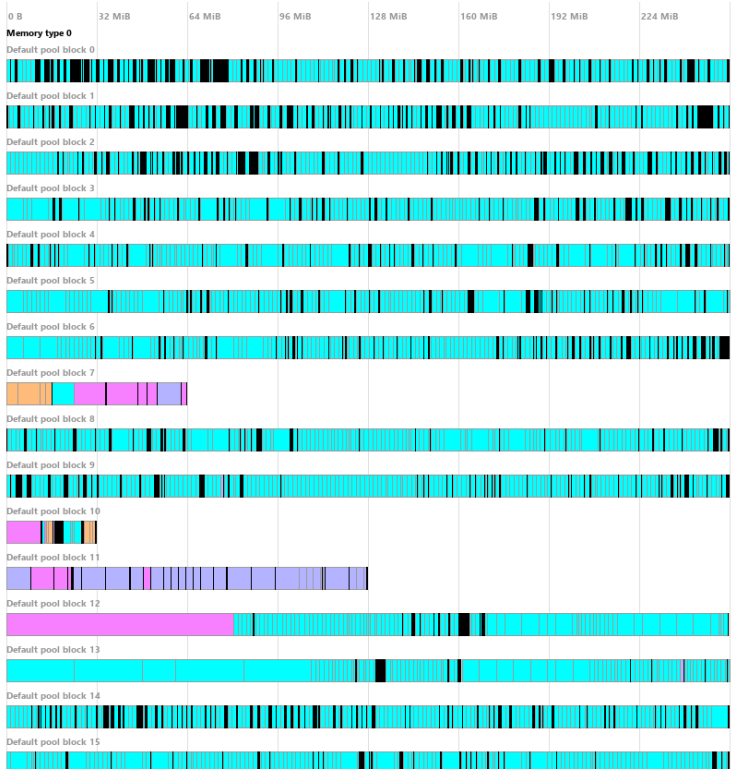
VMA/D3D12MA JSON DUMPS

- VMA and D3D12MA are open source memory management libraries for Vulkan ® and Direct3D® 12
- https://gpuopen.com/d3d12-memory-allocator/ and https://gpuopen.com/vulkan-memory-allocator/
- Both come with an auxiliary tool to visualize the internal state of the allocator

- Lists all the memory blocks for each heap and their size
- Shows the resources in each memory block
  - Resource size and type
- Shows free memory in each memory block

→ useful to analyse fragmentation

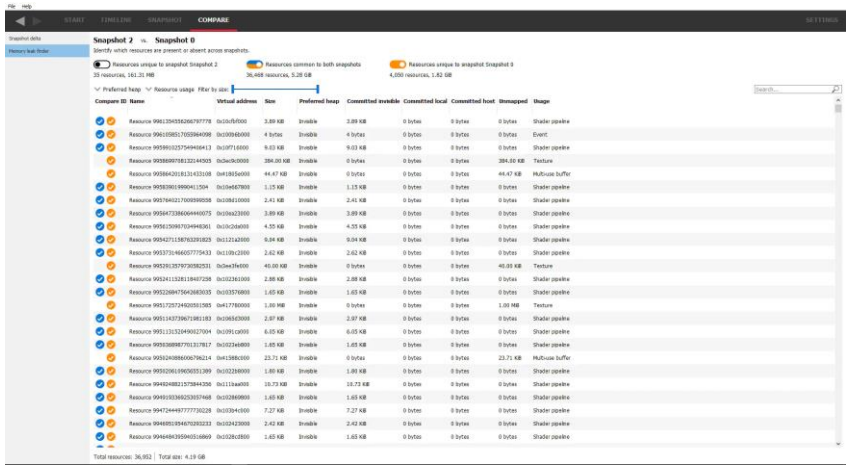
→ useful to determine if memory block size is a good fit for the application's resources

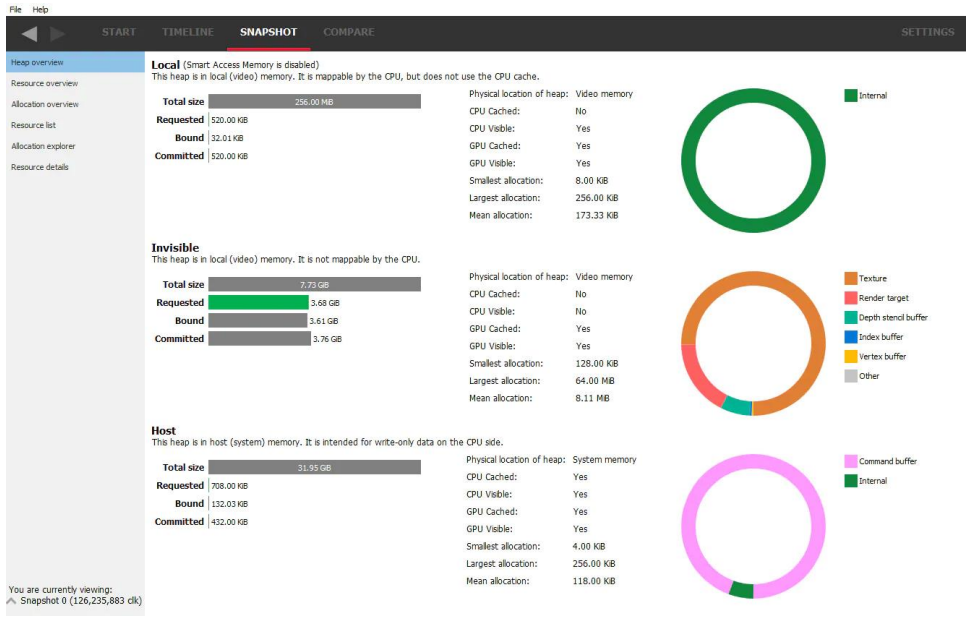


RADEON™ MEMORY VISUALIZER

- AMD tool to get an insight in how applications use memory for graphics resources from a driver perspective
- https://gpuopen.com/rmv/

- Lists the available heaps and the resources placed in them including driver internal resources
- You can compare two snapshots to find memory leaks





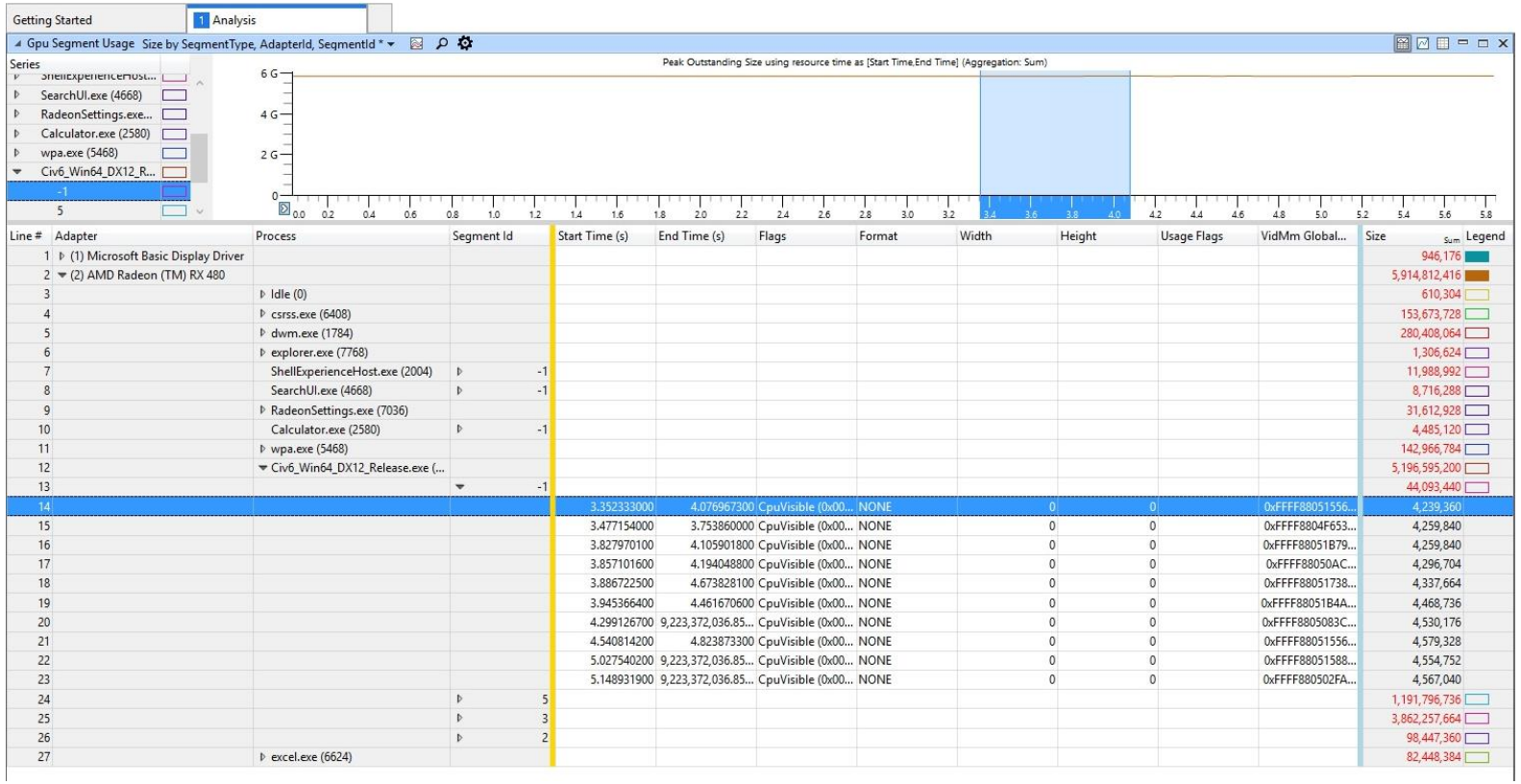
WINDOWS® PERFORMANCE ANALYZER

- Windows® Performance Analyzer is part of the Windows® Performance Toolkit, which is part of the Windows® 10 SDK

- Shows all current processes

- Shows how much memory each process allocated

- Lists all evicted memory blocks from VRAM to SysRAM under GPU Segment -1.



- The other GPU Segments map to the heaps you see in RMV

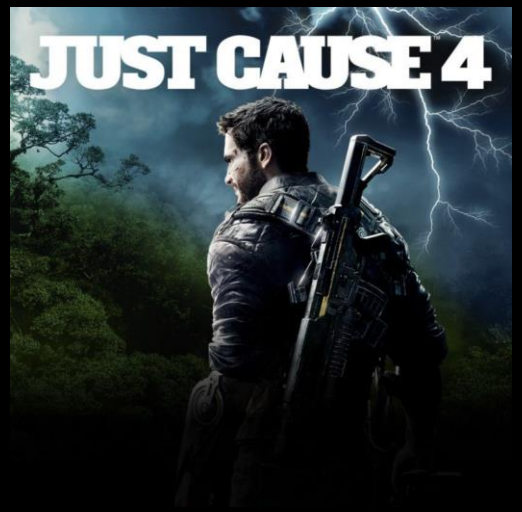
DISCLAIMER & ATTRIBUTIONS

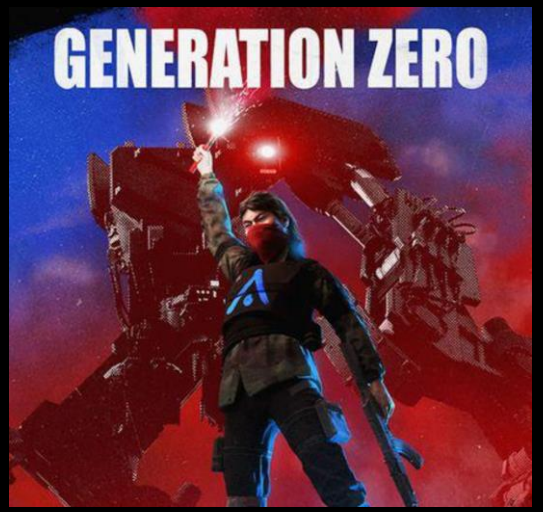
DISCLAIMER

The information contained herein is for informational purposes only and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale. GD-18


AMD

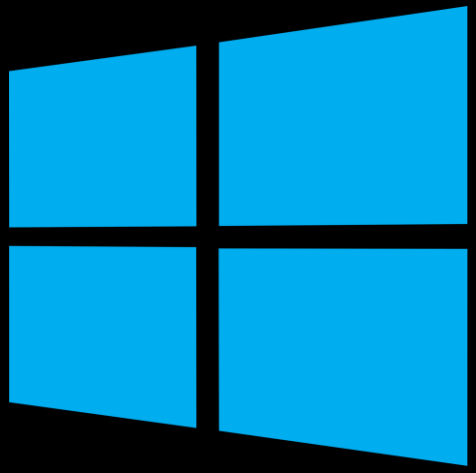
Avalanche
Studios Group





APEX

AVALANCHE OPEN WORLD ENGINE



APEX

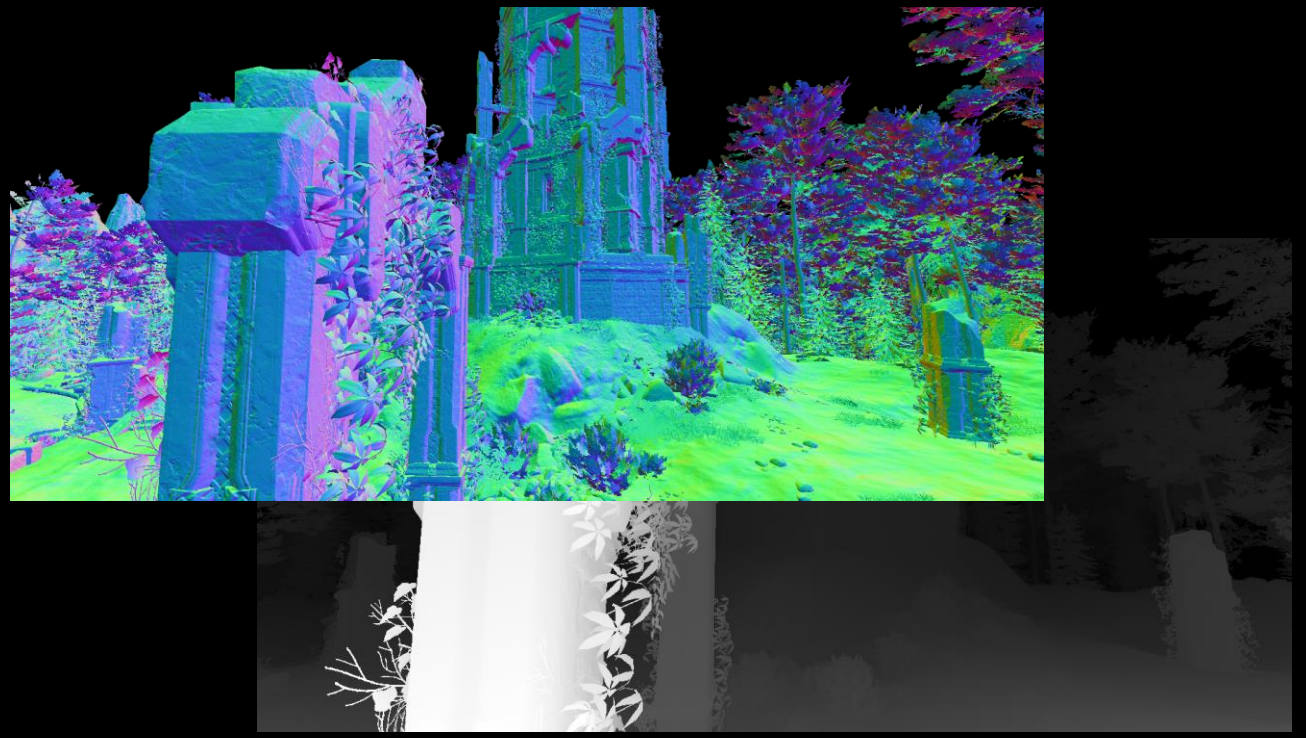
AVALANCHE OPEN WORLD ENGINE

Resources

- Many categories of resources
- Varying properties
  - Size
  - Number of resources
  - Lifetime
  - CPU/GPU usage
  - Performance importance
- D3D12MA and VMA

Render targets

- GPU read/write
  - Performance critical
- Resolution-dependent
  - 500-2000 MB
- Created at startup
  - Occasionally resized



GPU storage buffers

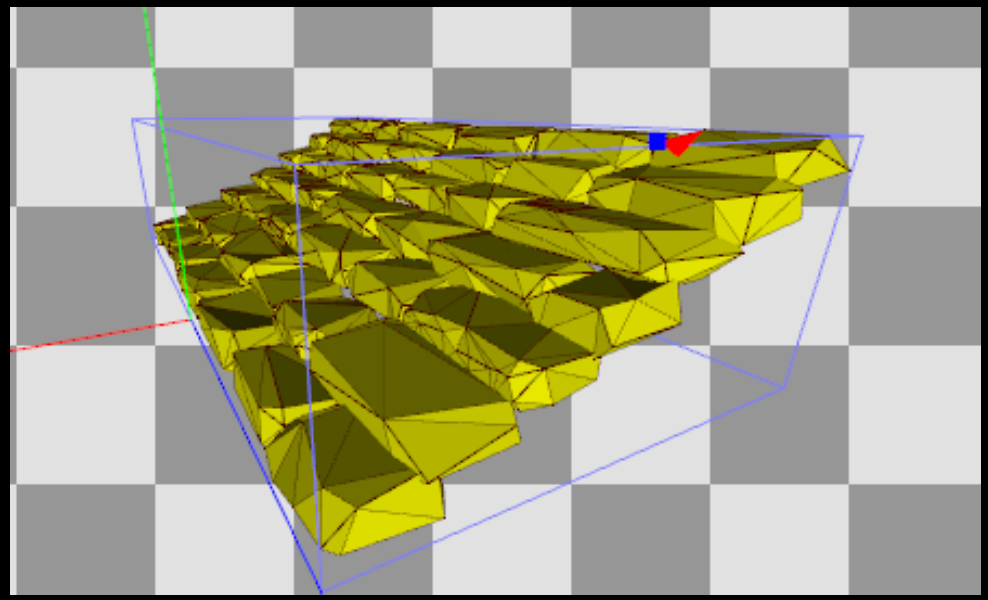
- Temporary buffers
  - Pre-skinned vertices
  - Compute shader generated terrain mesh
  - GPU-generated vegetation instance data
- GPU read/write
  - Sometimes performance critical
- ~ 100 MB
- Some allocated on startup, some on demand

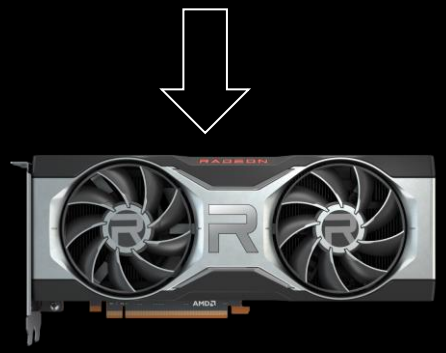
Buffer Contents

Element	m_ModelIndex	m_LodIndex	m_VisibilityI	m_NumTypes	m_TypeId
0	28	1	13	257	3
1	216	1	13	257	56
2	28	1	13	257	15
3	18	1	13	257	94
4	93	0	13	257	2
5	28	1	13	257	249
6	201	0	13	257	0
7	18	2	13	257	170
8	18	2	13	257	130
9	216	2	13	257	64
10	18	2	13	257	73
11	18	3	13	257	8
12	18	2	13	257	28
13	18	2	13	257	91
14	18	2	13	257	178

Streamed vertex/index/material buffers

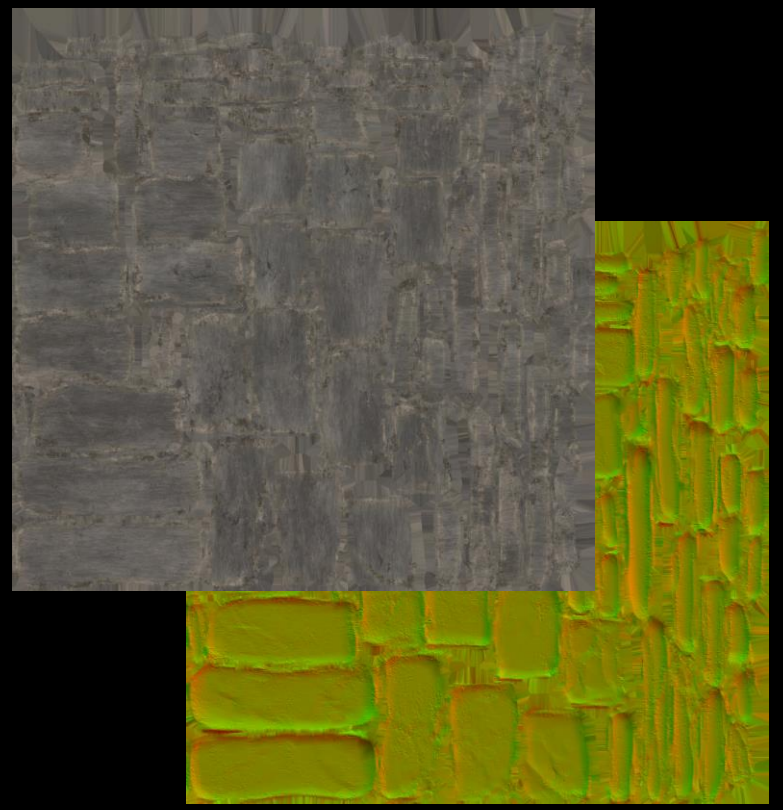
- GPU read-only
  - Copied from staging buffer
- Almost negligible size
- Streamed in and out





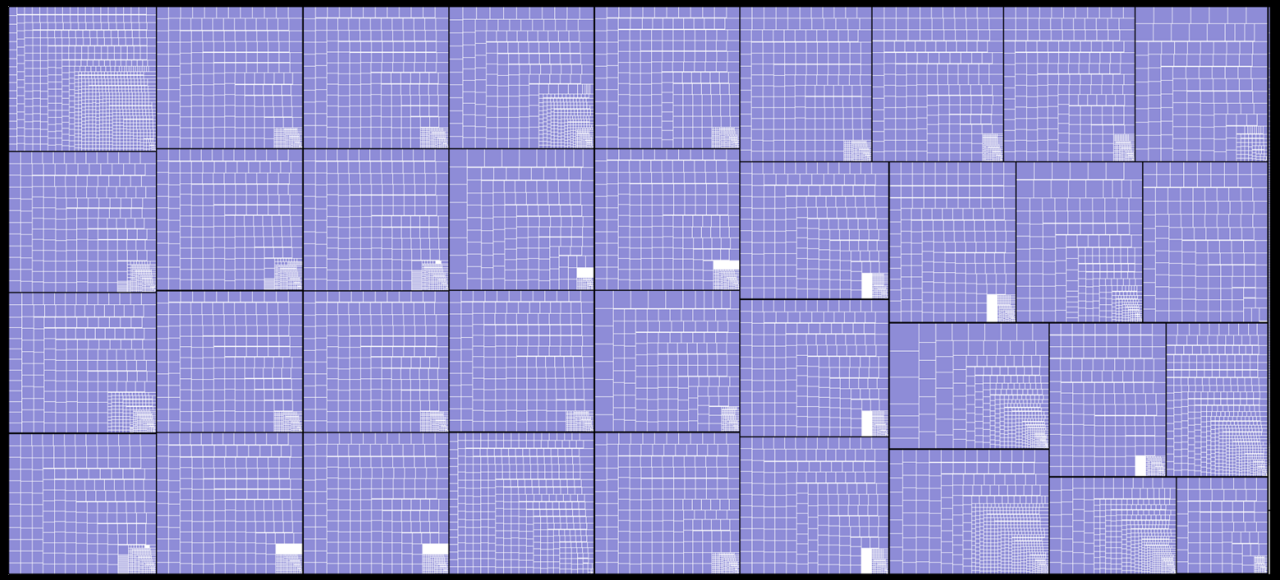
Streamed textures

- GPU read-only
  - Copied from staging buffer
- Loose memory budget
  - 500 - 2000+ MB
  - *Not* manual resource placement
  - Simply tracks total memory usage
  - Reference counted
- Large range of sizes
- Streamed in and out



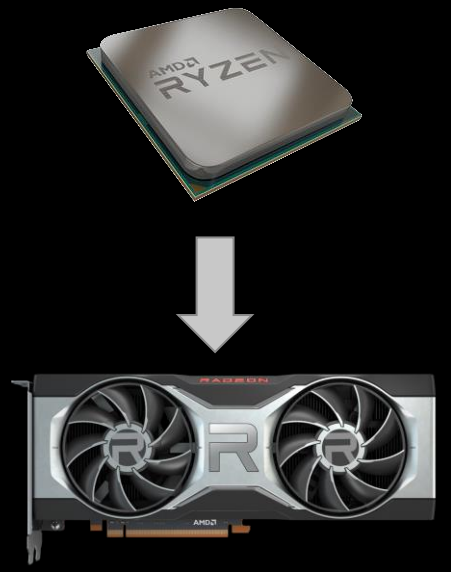
Shader pipelines

- They do take up VRAM!
  - Driver-managed
  - Visible in Radeon Memory Visualizer!
- Many, many shader permutations
- Almost 100 MB
  - ~256 B - 20 KB each



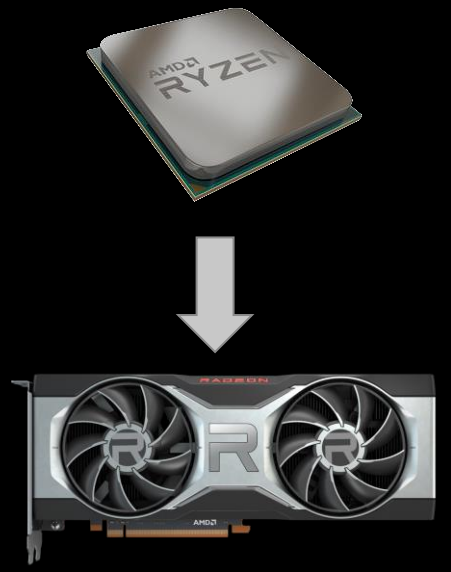
Constant buffers

- Camera matrices, constants, etc
- CPU generated each frame
- CPU RAM mapped memory
  - Caching on GPU hides cost
- Rotating buffers
  - Linear suballocator



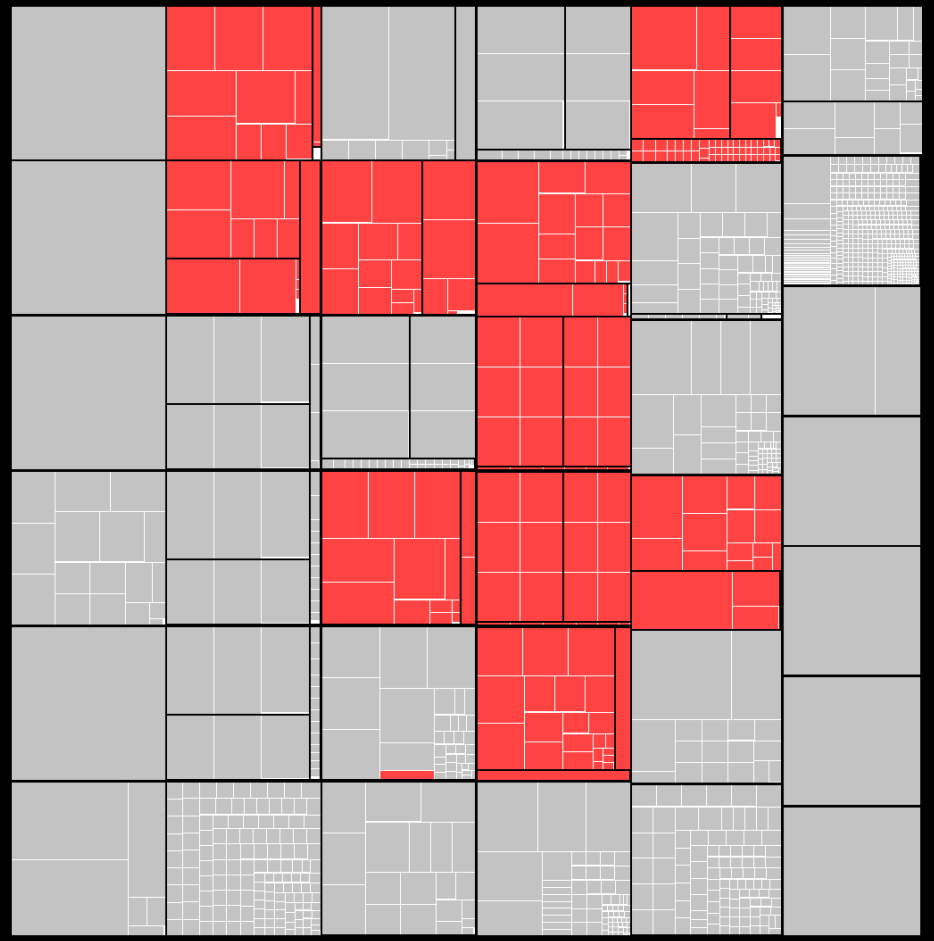
Staging buffers

- Temporary buffers in RAM
- Used to initialize buffers/textures
- Similar system to constant buffers



Performance issues

- Sudden drops to <10 FPS
  - Often after window resize
- GPU-limited
  - Abnormally high PCI-E bus load?
- Performance critical resource spilled to RAM!
  - Confirmed with Radeon Memory Visualizer

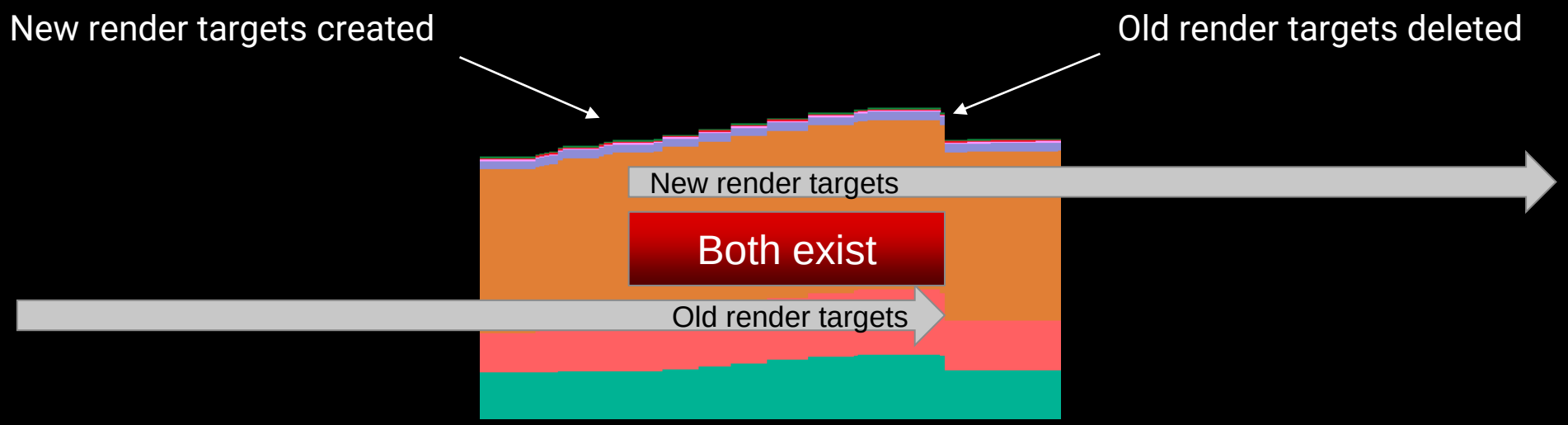


Problems

- Memory usage spikes
- Fragmentation
- Simply running out of VRAM

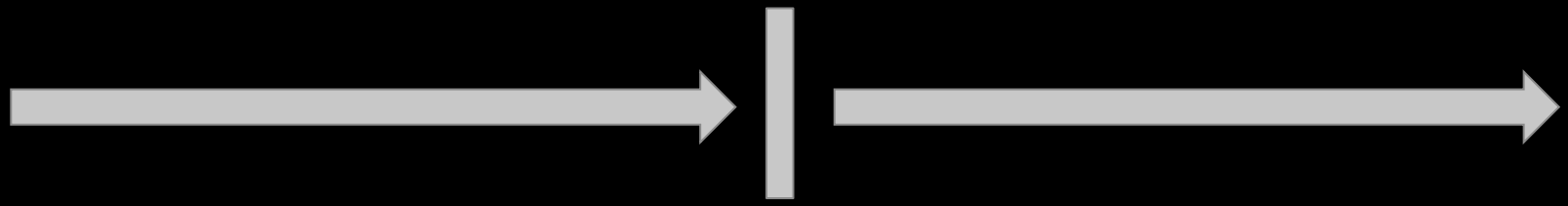
Problem - Memory spikes

- Cannot destroy resource in use by GPU
  - Engine defers deletion for 1-2 frames
- Resized render targets?
  - Massive VRAM usage spike
- Radeon Memory Visualizer useful



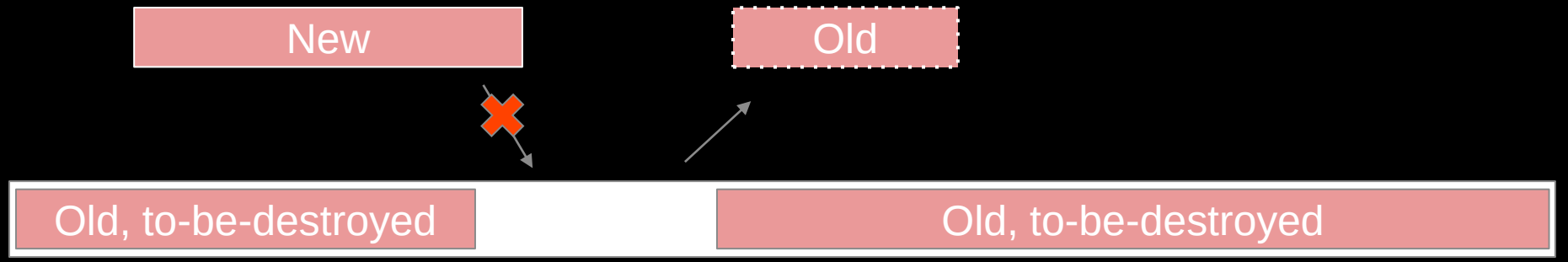
Solution - Immediate resource destruction

- Wait for GPU to finish all pending work
- Delete resources immediately
- Then create new resources



Solution - Resource recreation

- Avoid delete, create, delete, create, ...!
- First delete all resources
- THEN recreate all resources
  - Avoids potential fragmentation



Problem - Fragmentation

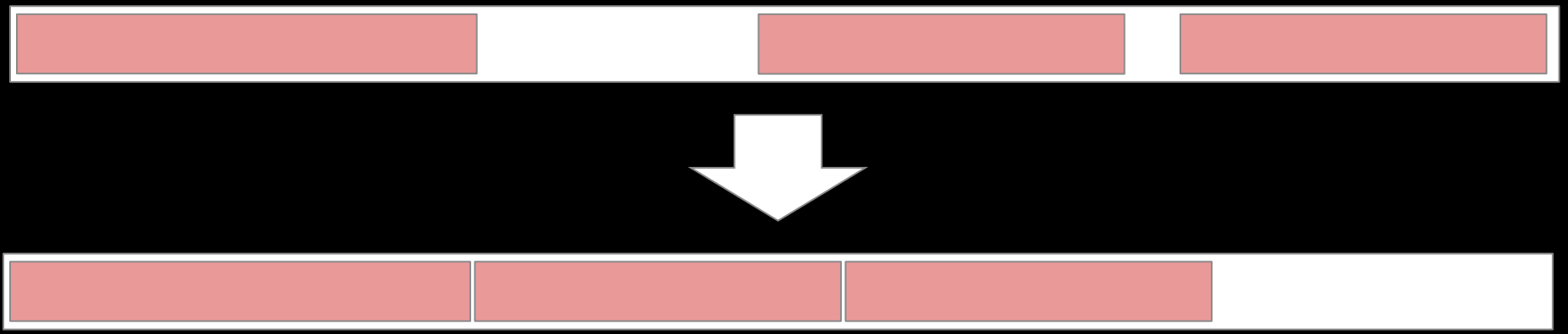
- Resources streamed in and out
- Fragmentation
- Inflates memory usage
  - Higher risk of paging
- Critical resources mixed in
  - Risk being paged out with the entire block
- Radeon Memory Visualizer useful

Solution - Committed resources

- Separate out performance critical resources
  - Create as committed resources
- Individual residency
  - No longer causes or suffers from fragmentation
- Which resources?
  - Render targets
  - Unordered access textures/buffers
  - “Large” textures
- Reminder: Max allocation limit!

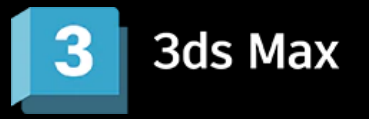
Solution - Defragmentation

- Background defragmentation
  - New D3D12MA feature
- No/few level switches
- Must update resource references
  - Resource tables/descriptor sets
  - Bindless resources



Problem - Simply running out of VRAM

- VRAM is scarce and highly contested
  - Multiple high resolution monitors
  - Heavy 3D artist software
  - Web browsers
  - Video recording software
  - etc…
- Spilling is still possible
  - Especially for our artists
- Can happen at any time
  - Want to minimize performance impact



Optimal resource priorities

1. Depth buffers
2. Render targets/unordered access textures
3. Compute shader intermediate buffers
4. Read-only textures
5. Read-only buffers

“Solution” - Assigning priorities

- Simple heuristic
  - High and normal priority resources
- Critical resources
  - Depth/color render targets
  - Unordered access textures/buffers
- Can only assign priorities to entire blocks
  - Critical resources are committed resources
- We don’t use Evict()/MakeResident()
  - Resource unusable after eviction (*not* the same as paging)
  - Difficult to identify eviction candidates

Avalanche
Studios Group

Digital Dragons

KRAKOWSKI PARK TECHNOLOGICZNY

Thank you

Lou Kramer

AMD

Daniel Isheden

Avalanche Studios Group