Candera Memory Pools Analysis

- Memory Pools Configuration
- Memory Pool Configuration
- “Unexpected” Allocations

Memory Pools Configuration

Introduction

If free memory in backing heap is exhausted, "Out of Memory" occurs. The possible reasons are as the below.

- Fragmentation of allocated blocks, not possible to fit in a new block.
- Initial analysis was wrong, configuration does not fit.
- “Unexpected” allocations.
- Executed use case cannot fit into supplied memory.

Please see the following pages:

- Memory Pool Configuration
- “Unexpected” Allocations

Memory Pool Configuration

Fragmentation

- A heap will fragment, when buffers with different lifetimes get allocated
- Fragmentation endangers system stability over run time



Heap Areas

- Permanent Allocation
  - will never return the allocated block to the backing heap
  - will be managed by the associated bin (bin free list)
- Transient Allocation
  - will immediately return the allocated block to the backing heap
- Two distinct memory areas
- The permanent area will never fragment
- The transient area can fragment



Default Memory Pool Example

Let’s have a look on an example:

```
static const MemoryPoolConfig config = FEATSTD_MEMORYPOOL_CONFIG_INITIALIZER(
    config,                          // name of the variable receiving the configuration
    "FeatStdDefaultMemoryPool",      // the name of the MemoryPool
    MemoryPoolFlags::None,           // Flags that control the memory pool behavior
    128,                             // all bins up to 128 bytes are small bins
    cPoolBinSpecs,                   // the bin configurations
    0,                               // preset allocations
    DefaultBackingHeap,              // type name of the backing heap
    &OnOutOfMemoryDef,               // optional the out of memory function
    0                                // optional OnDestroy function
```

```
static const BinSpecification cPoolBinSpecs[] = {
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None,  8), // 732
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 12), // 594
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 16), // 340
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 20), // 143
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 24), // 270
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 28), // 18
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 32), // 156
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 36), // 93
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 40), // 7
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 44), // 128
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 48), // 107
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 52), // 114
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 64), // 30
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 72), // 114
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 76), // 8
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 80), // 9
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 88), // 5
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 92), // 6
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 96), // 8
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 100), // 4
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 108), // 14
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 112), // 5
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 116), // 16
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 120), // 34
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 124), // 54
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 128), // 81
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 136), // 153
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 144), // 145
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 168), // 15
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 180), // 4
...
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 16384), // 1
    // mandatory final entry
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 0)
};
```

Memory Pool Example Outcome

- This configuration leads to fragmentation
- All flags (bin & pool) are set to *None* -> Allocations are always transient
  - Transient allocations can fragment
  - When all allocations are transient, fragmentation is certain
- Small Bin Limit is set to 128
  - All allocations up to 128 take the best fit bin size + 4 Byte overhead
  - All allocations > 128 take exactly the requested size + 8 Byte overhead
  - The configured bins > 128 are not considered at all, they just consume their header memory

Interpret MemoryPoolAnalysis.xlsx

When a permanent block with size 12800 Byte is used for 160 Byte allocation...

12800
C:\CGI-Studio_3.10\cgi_studio_candera\src\Candera\Engine2D\Core\RenderNode.cpp(66)

- The default behavior of Memory Pool for allocations is:

    1. Determine the bin according the requested buffer size
    2. Check the free list of the identified bin
    3. If no block is found in the free list, try to allocate a buffer from the backing heap.
    4. If the allocation from the backing heap fails, try to allocate from the next larger bins (overflow).

- In the example above, step 4 occurred -> Allocation of a new bin failed!

Proper Memory Pool Analysis

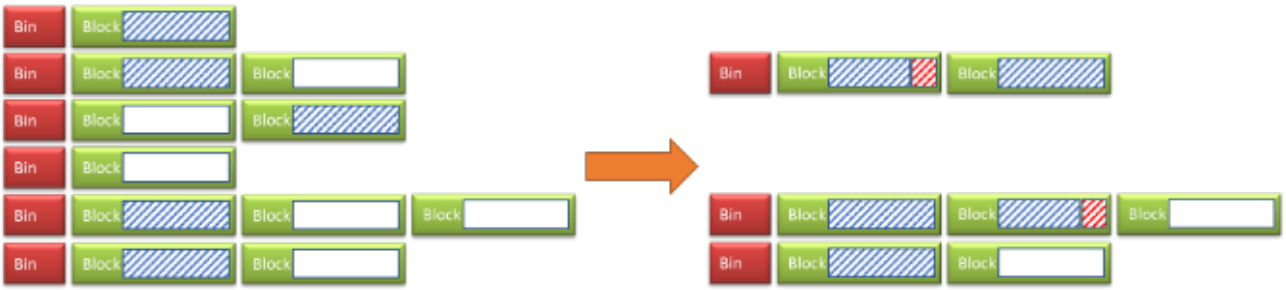
- A proper configuration for initial analysis…
- has a “small bin limit” of 4 Byte
    - All allocations shall take the requested size
- defines *PermanentOnly* flag on the pool
    - Bins shall be reused (freed blocks are returned to bin free list)
- defines additionally *NoOverflow* flag on the pool
    - inhibit overflow allocations
    - If specified, allocation requests from a bin will never overflow to a larger bin
    - Note: a test run with this flag might fail (Out of memory…) due to many unused blocks -> in this case remove the flag

Proper Memory Pool Configuration

- A proper final Memory Pool configuration…
- has a “small bin limit” in the upper area (where allocations are rarely)
    - Keep number of transient allocations to a minimum
- uses Flags on bins instead of the pool
    - defines *PermanentOnly* flag on small bins
    - Defines one bin with None flag, uses transient allocations for large blocks, which are rarely used (no reuse intended)
- uses few bins
    - balance wasted space vs. unused (rarely used) blocks

Wasted Space vs. Unused blocks

- Many permanent bins leads to many unused blocks
    - free blocks in bin free list
- Few permanent bins leads to memory “waste” in blocks
    - Allocation request smaller than block size
- The memory “waste” of few bins is smaller than free blocks from too many bins.
    - Free bin blocks are reused more often



Process of Configuration

- Take dumps with proper configuration for analysis
  - Play more than one use case to get better data
- If it is not possible to run with flag *NoOverflow* …
  - Remove the flag
  - Analysis is still valid, most overflows come from low size bins which are anyway “dense”
- Based on the results
  - Reduce the number of bins drastically
  - Choose a “small bin limit” where allocations are temporary requests Release of temporary memory vs. fragmentation

Reduce Number of Bins

Reduce the number of bins generously
Too many bins -> many free blocks not used
“Wasted Space” < Not used blocks!

```
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 8), // 732
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 12), // 594
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 16), // 340
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 20), // 143
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 24), // 270
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 28), // 18
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 32), // 156
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 36), // 93
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 40), // 7
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 44), // 128
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 48), // 107
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 52), // 114
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 64), // 30
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 68), // 1
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 72), // 114
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 76), // 8
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 80), // 9
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 84), // 1
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 88), // 5
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 92), // 6
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 96), // 8
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 100), // 4
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 108), // 14
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 112), // 5
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 116), // 16
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 120), // 34
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 124), // 54
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 128), // 81
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 132), // 1
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 136), // 153
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 140), // 2
```



```
..._SPECIFICATION(MemoryPoolBinFlags::<...>,  8),
..._SPECIFICATION(MemoryPoolBinFlags::<...>, 12),
..._SPECIFICATION(MemoryPoolBinFlags::<...>, 16),
..._SPECIFICATION(MemoryPoolBinFlags::<...>, 20),
..._SPECIFICATION(MemoryPoolBinFlags::<...>, 24),
..._SPECIFICATION(MemoryPoolBinFlags::<...>, 32),
..._SPECIFICATION(MemoryPoolBinFlags::<...>, 36),
..._SPECIFICATION(MemoryPoolBinFlags::<...>, 64),
..._SPECIFICATION(MemoryPoolBinFlags::<...>, 80),
..._SPECIFICATION(MemoryPoolBinFlags::<...>, 96),
..._SPECIFICATION(MemoryPoolBinFlags::<...>, 128),
..._SPECIFICATION(MemoryPoolBinFlags::<...>, 136),
```

Choose “Small Bin Limit”

- Remember: Avoid non used blocks!
- Summarize bins to small bins as far as possible
- “Small Bin Limit” shall start where allocations are temporary requests and fragmentation is more unlikely

```
...
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 224), // 4
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 256), // 6
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 272), // 9
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 320), // 1
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 328), // 1
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 336), // 1
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 344), // 1
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 352), // 9
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 360), // 1
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 368), // 1
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 380), // 1
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 416), // 1
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 440), // 3
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 444), // 1
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 460), // 4
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 476), // 2
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 504), // 2
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 512), // 7
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 520), // 2
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 532), // 1
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 572), // 1
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 600), // 1
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 616), // 1
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 704), // 6
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 800), // 1
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 1024), // 6
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 1248), // 1
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 1600), // 1
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 2048), // 1
```



```
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 272),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 352),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 476),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 532),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 704),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 1024),
```

```
..._BIN_SPECIFICATION(MemoryPoolBinFlags::<...>, 1600),
```

Result

```
static const BinSpecification cPoolBinSpecs[] = {
..._BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly,  8),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 12),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 16),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 20),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 24),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 32),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 36),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 64),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 80),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 96),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 128),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 136),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 144),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 220),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 272),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 352),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 476),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 532),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 704),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 1024),
..._BIN_SPECIFICATION(MemoryPoolBinFlags::None, 1600),


// mandatory final entry
..._BIN_SPECIFICATION(MemoryPoolBinFlags::None, 0)
};
```



```
// define the configuration for DefaultMemoryPool.
static const MemoryPoolConfig config =
FEATSTD_MEMORYPOOL_CONFIG_INITIALIZER(
    config,                         // name of the variable receiving the configuration
    "FeatStdDefaultMemoryPool",     // the name of the MemoryPool
    MemoryPoolFlags::None,          // Flags that control the memory pool behavior
    1024,                           // all bins up to 1024 bytes are small bins
    cPoolBinSpecs,                  // the bin configurations
    0,                              // preset allocations
    DefaultBackingHeap,             // type name of the backing heap
    &OnOutOfMemoryDef,              // optional the out of memory function
    0                               // optional OnDestroy function
```

Advanced Topics

- Pre-Allocation
  - Define how many permanent blocks shall be pre-allocated in each bin

- Combine with OverflowPreferred flag on bins
  - first try to overflow and then allocate from the backing heap
  - Limit overflows by adding flag OverflowBarrier to bin configuration
  - Such a configuration might improve the memory layout
- A sophisticated out of memory function could trigger the release of unneeded memory
  - Return value controls if the allocation request will be repeated or abandoned

“Unexpected” Allocations

VRAM Memory Pool

- [Applicable for platforms Traveo, Traveo II, RH850]
- Why in the VRAM 4 buffers are allocated for render buffers of a single Render Target?
- What happens:
  - During a transition first scene is unloaded, next scene is loaded
  - In case of startup, the first scene is the only one existing
  - Default Behavior of Courier:
    - when the last View on a Render Target is unloaded, the Render Target is also unloaded
  - The VramAllocator does never immediately free the memory, because the GPU might still write to it (Note: GPU runs
  - asynchronously to the CPU)
  - -> New 2 buffers are allocated when the second scene is loaded
- How to fix it?
- Send on startup a ViewPlaceholderReqMsg
  - increases the reference counter on Render target.
  - The message has two parameters
  - ViewId: this is a "reference" view which is used to determine the render target object
  - Load: true means to increase the reference count, false would decrease it (removes this placeholder from the internal list)
- Alternatively, you can also use a dummy scene with just a camera
  - This view would be always active and never unloaded.

Asset Cache

- From the Memory Pool Analysis Sheet:

16384
C:\CGI-Studio_3.10\cgi_studio_candera\src\CanderaAssetLoader\AssetLoaderBase\AssetCache.cpp(38)

- -> This is the cache used by the Asset Loader when copying images from flash to RAM
- Most or all images are directly rendered from flash
- The cache size is best practice for many platforms, but not when flash is directly accessible!
- -> Proposal: If flash is directly accessible, reduce the cache
  - E.g., to 1 kB (you win 15 kB!)
- CMake flag CANDERA_ASSET_CACHE_SIZE