4. Game Engine Support System (part. 1)

Game Player Experience Design

Prof. H. Kang

Keywords of prior knowledge

Hope these keywords help your study:

- Static class & variables
- Singleton
- Heap & Priority queue data structure
- Stack allocator & Heap allocator
- Context-switching

Game Engine Support System

Intro

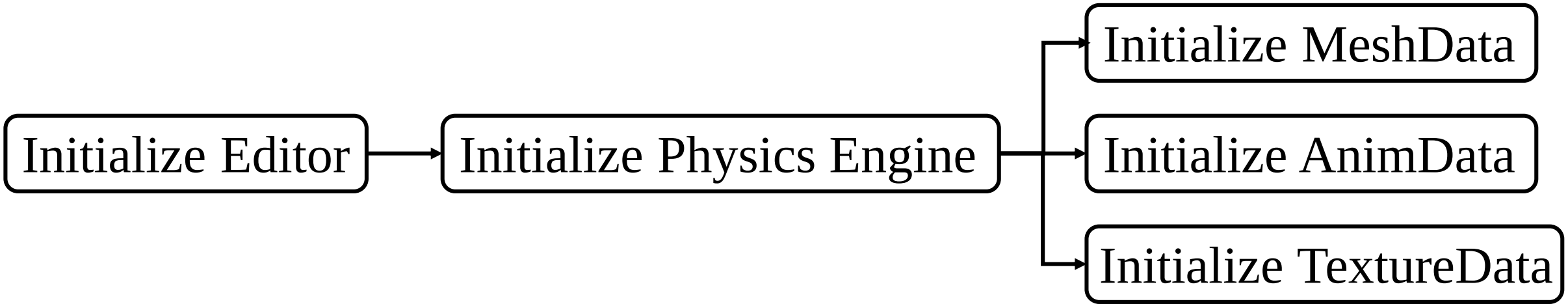
- Every game engine requires some low-level support systems that manage crucial tasks:
  - Starting up and shutting down the engine
  - Configuring engine and game features
  - Managing the engine's memory usage
  - Handling access to file system
  - Providing access to the wide range of heterogeneous asset types used by the game (meshes, textures, animations, audio, etc.)
  - Providing debugging tools for use by the game development team

- This lecture will focus on the lowest-level support systems found in most game engines.

Subsystems and Their Initialization

Subsystem (start-up and shut-down)

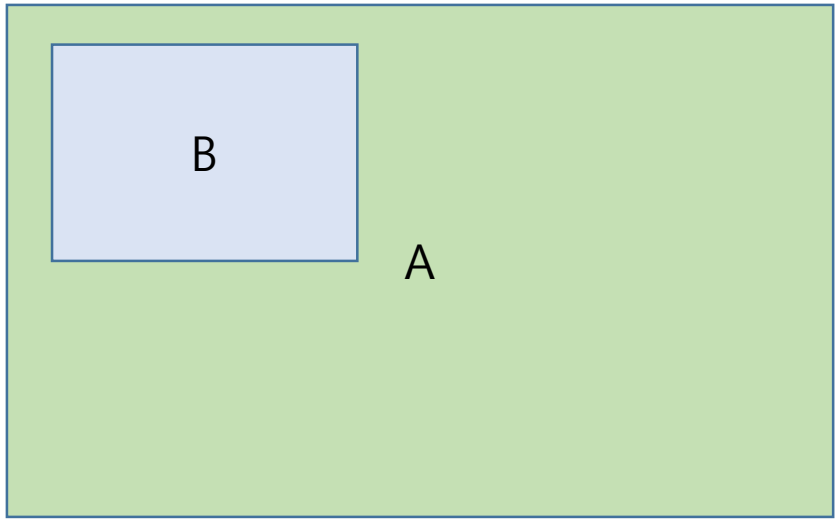
- A game engine is a complex piece of software consisting of many interacting subsystems.
- When the engine first starts up, each subsystem must be configured and initialized in a specific order.



Subsystems and Their Initialization

Subsystem (start-up and shut-down)

- Interdependencies between subsystems implicitly define the order in which they must be started
  - If subsystem B depends on subsystem A, then A will need to be started up before B can be initialized.
  - Shut-down typically occurs in the reverse order, so B would shut down first, followed by A.

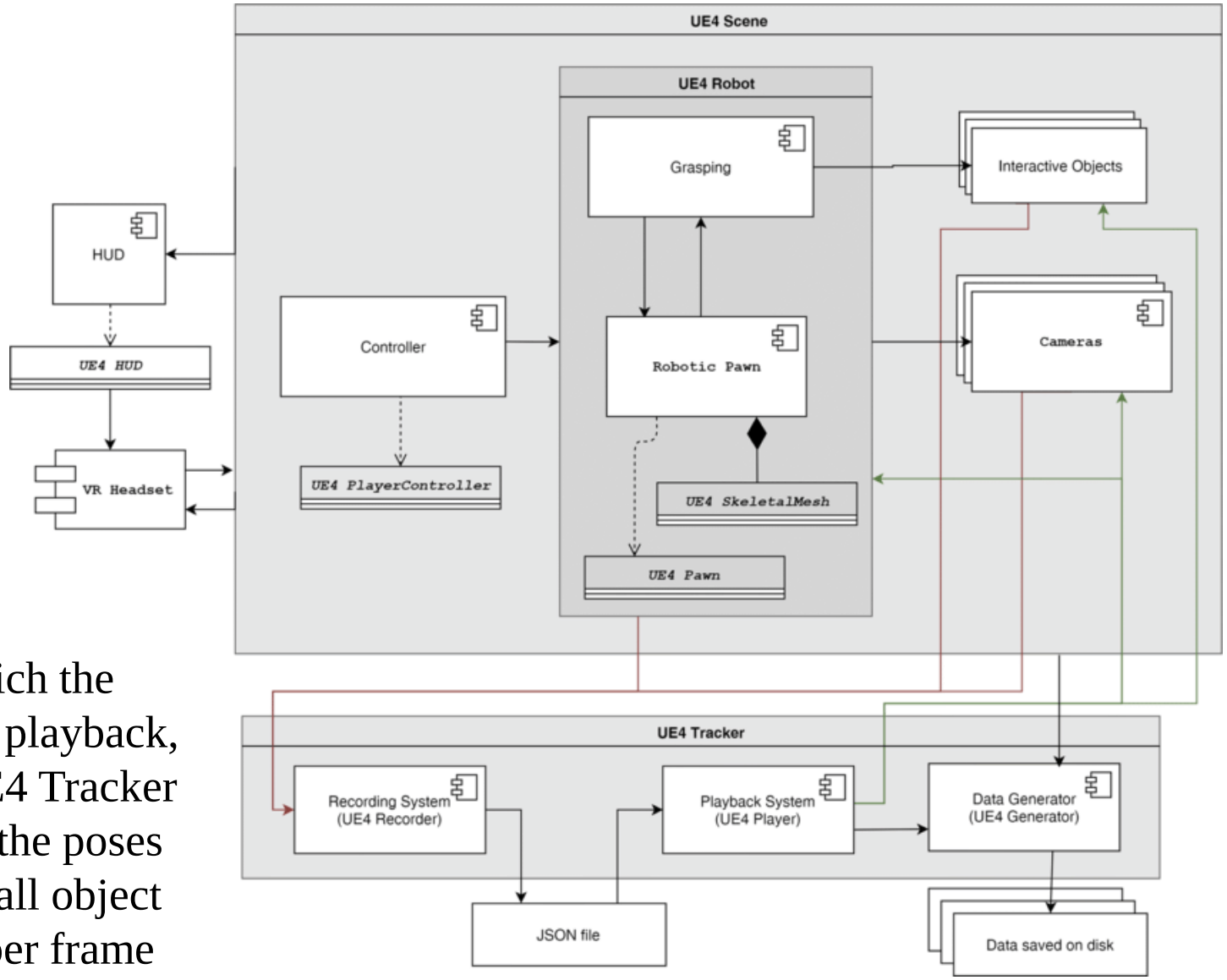


B depends on subsystem A

Subsystems and Their Initialization

Unreal subsystem diagram (example)

- Robotic Pawn, Controller, HUD, Multi-camera, Grasping, Recording, and Playback subsystems.



Note. Gray containers represent the scope in which the various systems act. For instance, the recording, playback, and data generation subsystems compose the UE4 Tracker component in which the recording system takes the poses of the joints of the pawn, the camera poses, and all object poses and then dumps all that information on a per frame basis.

C++ static initialization order

C++ static initialization order

- Since the programming language used in most modern game engines is C++, we should briefly consider whether C++'s native start-up and shut-down semantics can be leveraged in order to start up and shut down our engine's subsystems.
- In C++, global and static objects are constructed before the program's entry point (*main*(), or *WinMain*() under windows) is called.
- The destructors of global and static class instances are called after main() returns.
- These behaviors are not desirable for initializing and shutting down the subsystems of a game engine.

C++ static initialization order

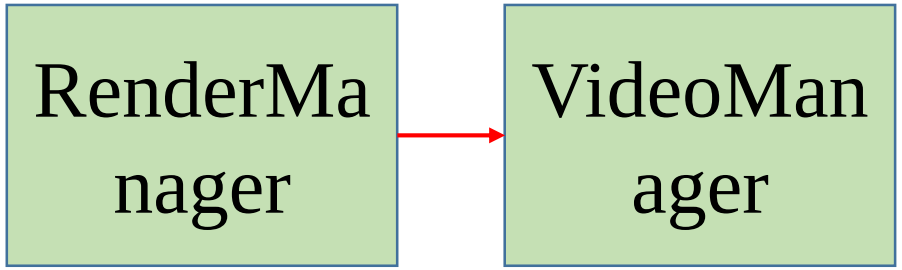
C++ static initialization order

- A common design pattern for implementing major subsystems is to define a singleton class (often called a manager) for each subsystem.
- Since we have no way to directly control construction and destruction order, we could not define our singleton instances as globals without dynamic memory allocation.
- Instead, we often use a trick: a static variable that is declared within a function will not be constructed before main() is called, but rather on the first invocation of that function.
- If our global singleton is function-static, we can control the order of construction for our global singletons.

C++ static initialization order

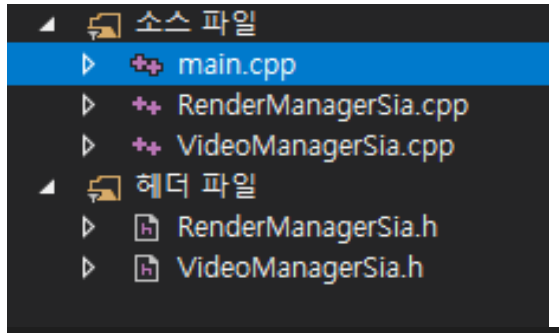
C++ static initialization order

- Assume that we want to initialize the manager in the order of RenderManager, and VideoManager.



C++ static initialization order

Example 1




```cpp
int main()
{
    printf("I'm main() \n");
    auto a = RenderManagerSia::get();
    auto b = VideoManagerSia::get();

    SIAMIZ sia_1(10, 1);
    SIAMIZ sia_2(20, 20);

    auto [data1, data2] = sia_1.getData();
    printf("%d, %d", data1, data2);

    if (auto val = sia_1.getValue(); val != NULL)
    {
        //do something
    }
    return 0;
}
```

Singletons

```cpp
class RenderManagerSia
{
private:
    static RenderManagerSia instance;
public:
    // Get the one and only instance.
    static RenderManagerSia& get()
    {
        return instance;
    }

    RenderManagerSia()
    {
        printf("I'm render() \n");
    }
    ~RenderManagerSia()
    {
        // Shut down the manager.
        // ...
    }
};
```

```cpp
class VideoManagerSia
{
private:
    static  VideoManagerSia instance;
public:
    // Get the one and only instance.
    static VideoManagerSia& get()
    {
        return instance;
    }

    VideoManagerSia()
    {
        printf("I'm video() \n");
    }
    ~VideoManagerSia()
    {
        // Shut down the manager.
        // ...
    }
};
```

C++ static initialization order

Example 1

- The intended order is main() - render() - video().
- From the novice coder's view, everything looks find!

```cpp
int main()
{
    printf("I'm main() \n");
    auto a = RenderManagerSia::get();
    auto b = VideoManagerSia::get();

    SIAMIZ sia_1(10, 1);
    SIAMIZ sia_2(20, 20);

    auto [data1, data2] = sia_1.getData();
    printf("%d, %d", data1, data2);

    if (auto val = sia_1.getValue(); val != NULL)
    {
        //do something
    }
    return 0;
}
```

code order

Result 1

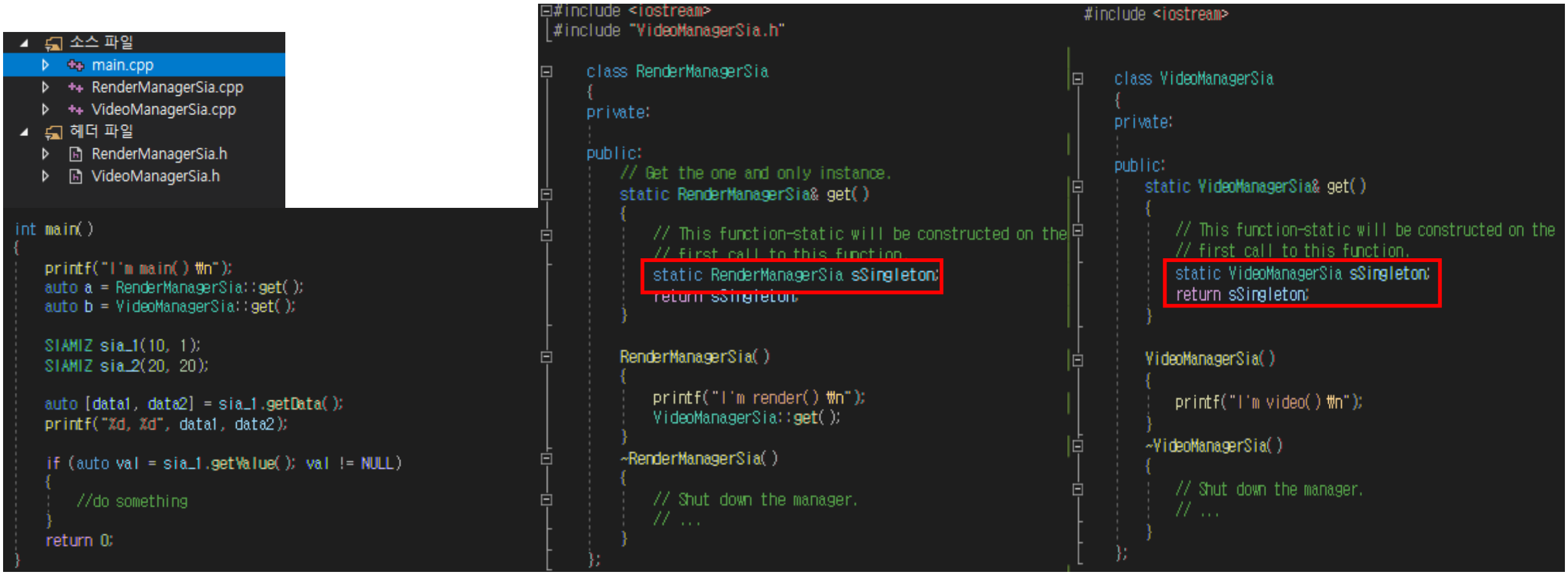
- However, initialization order is incorrect. Why?

```
I'm video()
I'm render()
I'm main()
```

result

C++ static initialization order

Example 2



C++ static initialization order

Result 2

- Now, initialization order is incorrect. Why?

```
I'm main()
I'm render()
I'm video()
```

C++ static initialization order

C++ static initialization order

- If the dynamic allocation of the singleton is included, the code will be shown as below:

```
class RenderManagerSia
{
private:

public:
    // Get the one and only instance.
    static RenderManagerSia& get()
    {
        static RenderManagerSia* gpSingleton = NULL;
        if (gpSingleton == NULL)
        {
            gpSingleton = new RenderManagerSia;
        }
        assert(gpSingleton);
        return *gpSingleton;
    }

    RenderManagerSia()
    {
        printf("I'm render() \n");
        VideoManagerSia::get();
    }
    ~RenderManagerSia()
    {
        // Shut down the manager.
        // ...
    }
};
```

```
class VideoManagerSia
{
private:

public:
    static VideoManagerSia& get()
    {
        static VideoManagerSia* gpSingleton = NULL;
        if (gpSingleton == NULL)
        {
            gpSingleton = new VideoManagerSia;
        }
        assert(gpSingleton);
        return *gpSingleton;
    }

    VideoManagerSia()
    {
        printf("I'm video() \n");
    }
    ~VideoManagerSia()
    {
        // Shut down the manager.
        // ...
    }
};
```

C++ static initialization order

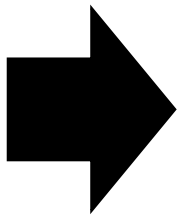
C++ static destruction order

- We have not discussed how to control the destruction order.
- We want to stick with the idea of singleton managers for our subsystems.
- Then, the simplest approach is to define explicit start-up and shut-down functions for each singleton manager class.

C++ static initialization order

C++ static destruction order

```
class RenderManager
{
public:
    RenderManager()
    {
        // do nothing
    }
    ~RenderManager()
    {
        // do nothing
    }
    void startUp()
    {
        // start up the manager...
    }
    void shutDown()
    {
        // shut down the manager...
    }
};
```



```
// ...

RenderManager           gRenderManager;
PhysicsManager          gPhysicsManager;
AnimationManager        gAnimationManager;
TextureManager          gTextureManager;
VideoManager            gVideoManager;
MemoryManager           gMemoryManager;
FileSystemManager       gFileSystemManager;
// ...

int main(int argc, const char* argv)
{
    // Start up engine systems in the correct order.
    gMemoryManager.startUp();
    gFileSystemManager.startUp();
    gVideoManager.startUp();
    gTextureManager.startUp();
    gRenderManager.startUp();
    gAnimationManager.startUp();
    gPhysicsManager.startUp();
    // ...

    // Run the game.
    gSimulationManager.run();

    // Shut everything down, in reverse order.
    // ...
    gPhysicsManager.shutDown();
    gAnimationManager.shutDown();
    gRenderManager.shutDown();
    gFileSystemManager.shutDown();
    gMemoryManager.shutDown();

    return 0;
}
```

C++ static initialization order

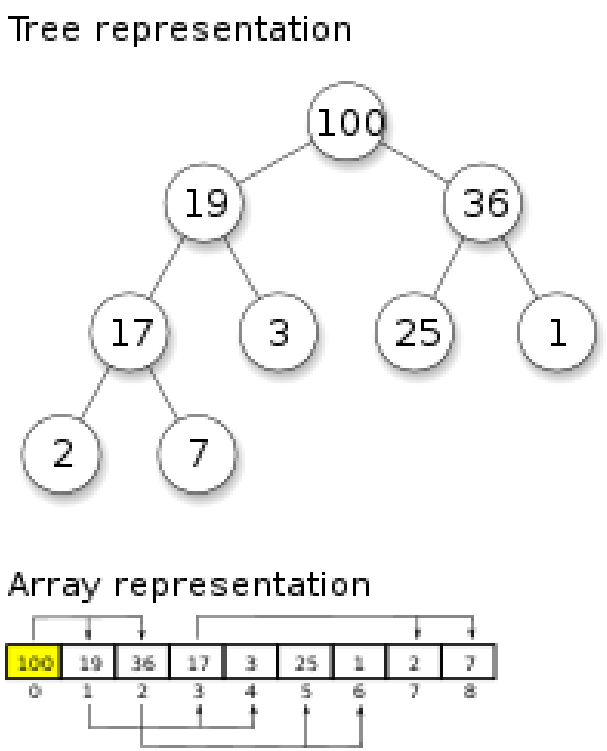
C++ static destruction order

- There are more elegant ways to accomplish this.
  - You could have each manager register itself into a global priority queue and then walk this queue to start up all managers in the proper order.
    - ✓ In a priority queue, an element with high priority is served before an element with low priority.
    - ✓ Priority queues are often implemented with heaps, but conceptually they differ from heaps: A priority queue is a concept like ‘a list’ or ‘a map’.

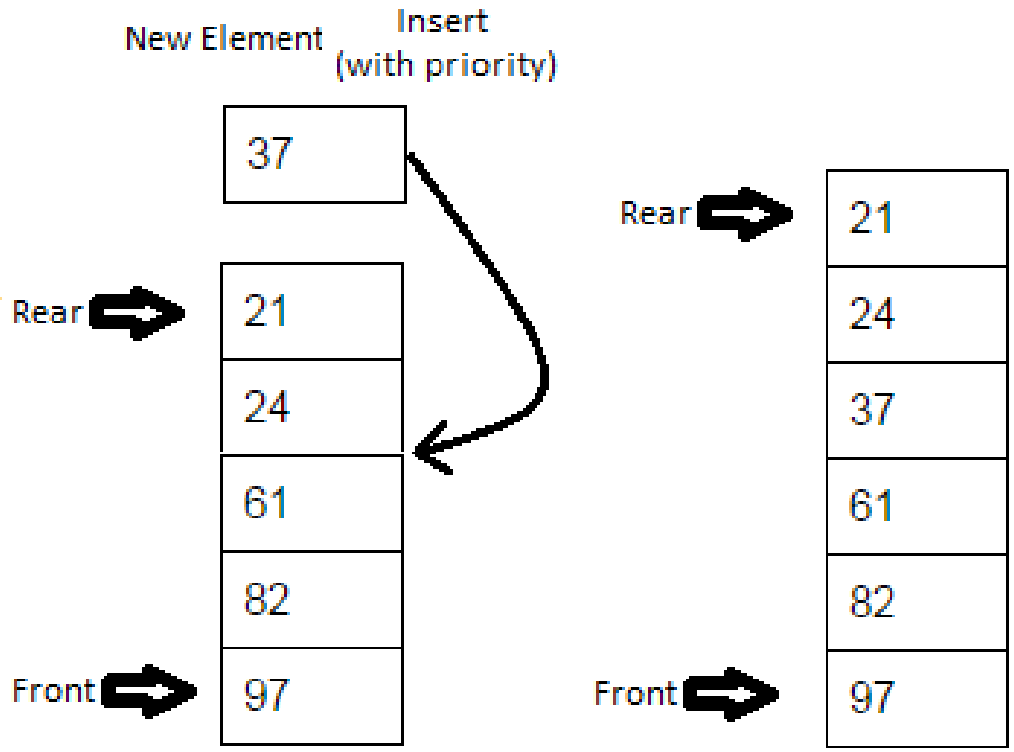
C++ static initialization order

C++ static destruction order

- You could define the manager-to-manager dependency graph by having each manager explicitly list the other managers upon which it depends and then write some code to calculate the optimal start-up order given their interdependencies.



Max-heap example[heap]



Priority queue example[pqueue]

UnrealEngine initialization

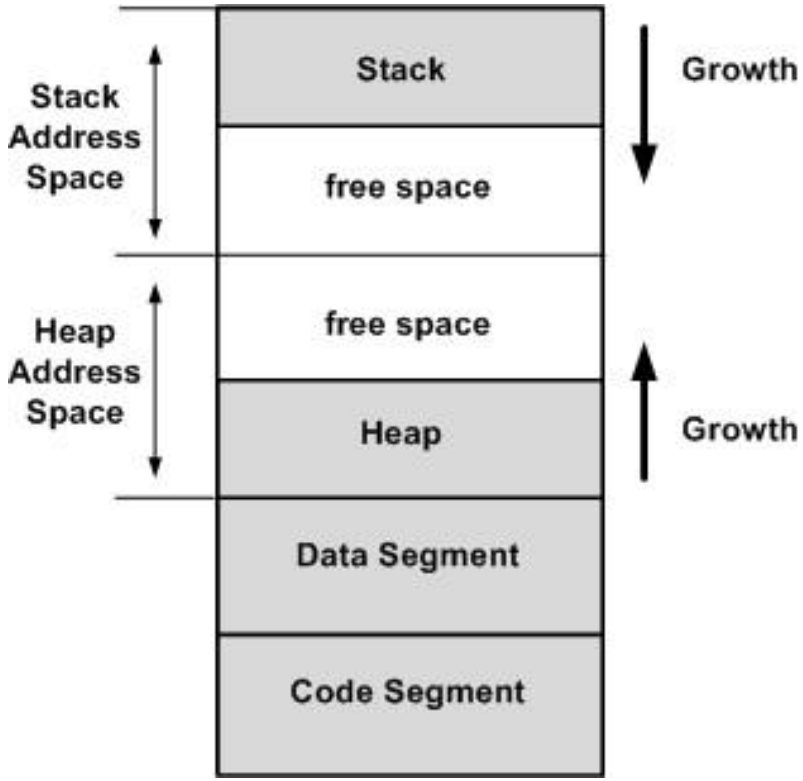
Unreal (LaunchEngineLoop.cpp)

```
        bDisableDisregardForGC |= FPlatformProperties::RequiresCookedData() && (GUseDisregardForGCOnDedicatedServers == 0);
    }

    // If std out device hasn't been initialized yet (there was no -stdout param in the command line) and
    // we meet all the criteria, initialize it now.
    if (!GScopedStdOut && !bHasEditorToken && !bIsRegularClient && !IsRunningDedicatedServer())
    {
        SCOPED_BOOT_TIMING("InitializeStdOutDevice");

        InitializeStdOutDevice();
    }

    bool bIsCook = bHasCommandletToken && (Token == TEXT("cookcommandlet"));
```

Memory management

Stack allocation vs. Heap allocation

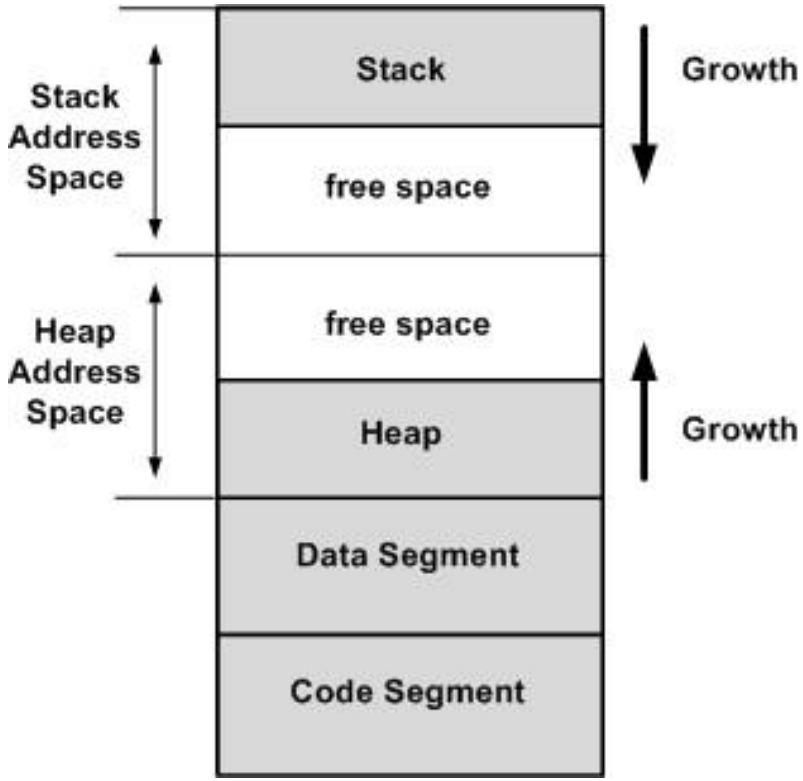
- Stack allocation happens on contiguous blocks of memory with the function call stack.
  - The size of memory to be allocated is known to the compiler and whenever a function is called.
  - Whenever the function call is over, the memory for the variables is de-allocated.
    - ✓ Therefore, programmers do not have to worry about memory allocation and de-allocation of stack variables.
  - This kind of memory allocation also known as Temporary memory allocation because as soon as the method finishes its execution all the data belongs to that method flushes out from the stack automatically.



Memory management

Stack allocation vs. Heap allocation

- Heap allocation happens during the execution of instruction written by programmers.
  - Every time when we made an object it always creates in Heap-space and the referencing information to these objects are always stored in Stack-memory.
  - Heap memory allocation isn't as safe as Stack memory allocation was because the data stored in this space is accessible or visible to all threads.
  - If a programmer does not handle this memory well, a memory leak can happen in the program.



Memory management

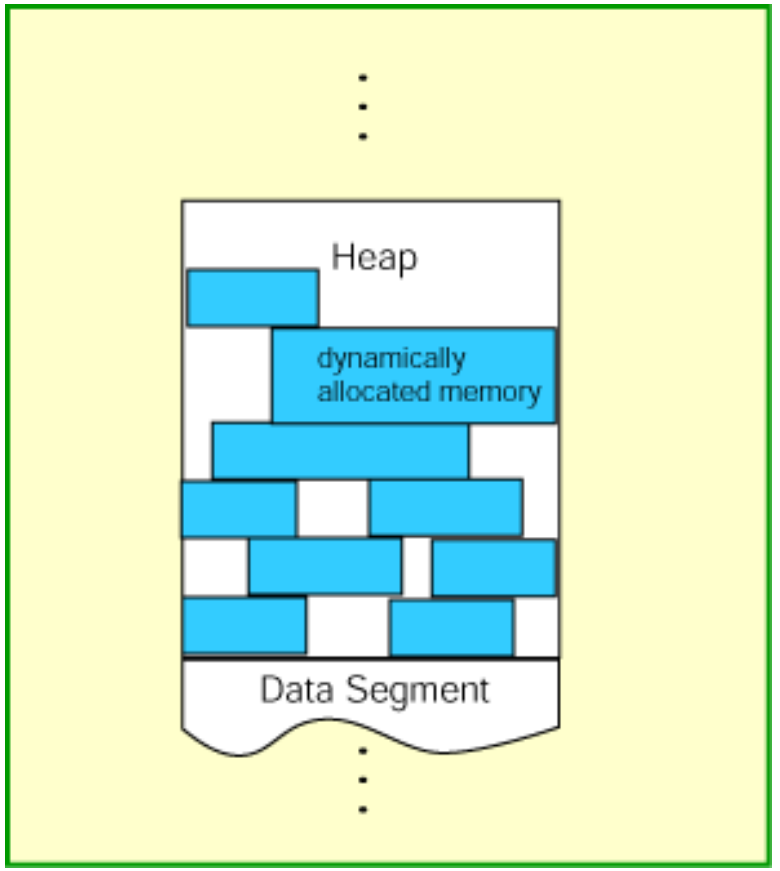
Memory management and game performance

- As game developers, we are always trying to make our code run more quickly.
- The performance of any piece of software is dictated not only by the algorithms it employs, or the efficiency with which those algorithms are coded, but also by how the program utilizes memory.

Memory management

Memory management and game performance

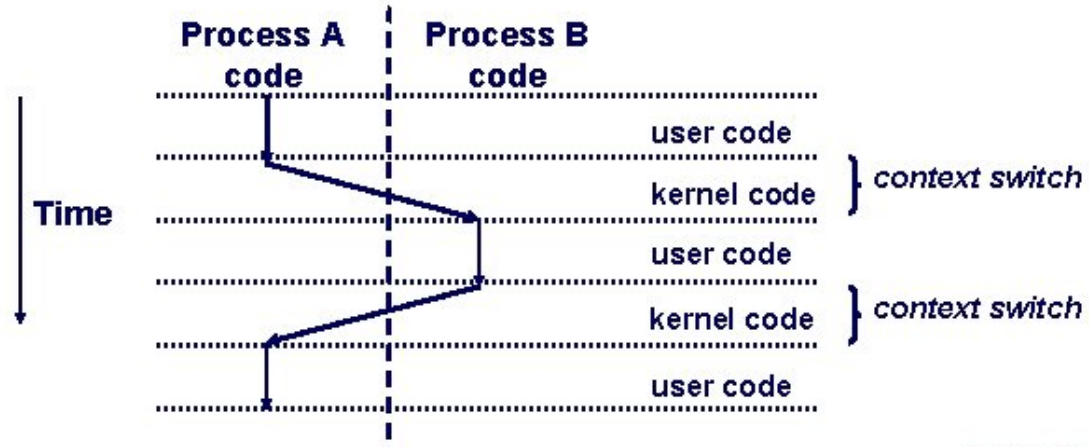
- Memory affects performance in two ways:
  - *Dynamic memory allocation* via malloc() or new() is a very slow operation. This can be improved by either avoiding dynamic allocation altogether or by making use of custom memory allocators that greatly reduce allocation costs.
  - On modern CPUs, the performance of a piece of software is often dominated by its memory access patterns. For example, contiguous blocks of memory can be operated on much more efficiently by the CPU than if that same data were to be spread out across a wide range of memory addresses.



Memory management

Dynamic memory allocation

- Dynamic memory allocation via *malloc()-free()* or *new()-delete()* operators is typically very slow.
- The high cost can be attributed to two main factors:
  - A heap allocator is a general-purpose facility, so it must be written to handle any allocation size, from one byte to one gigabyte. This requires a lot of management overhead.
  - On most operating systems a call to *malloc()* or *free()* must first context-switch from user mode into kernel mode, process the request and then context-switch back to the program. These context switches can be extraordinarily expensive.



Memory management

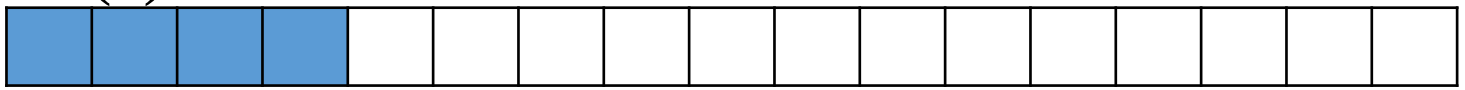
Dynamic memory allocation

- Let's look at the *malloc()-free()* package:
  - void **malloc*(size_t *size*)
    - if successful:
      - Returns a pointer to a memory block of at least *size* bytes, (typically) aligned to 8-byte boundary.
      - If *size* == 0, returns NULL
    - if unsuccessful: returns NULL (0) and sets errno.
  - void free(void **p*)
    - Returns the block pointed at by *p* to pool of available memory.
    - *p* must come from a previous call to *malloc()* or *realloc()*.
  - void *realloc(void **p*, size_t *size*)
    - Changes size of block *p* and returns pointer to new block.
    - Contents of new block unchanged up to min of old and new size.

Memory management

Dynamic memory allocation

- Allocation examples:
  - p1 = malloc(4)



  - p2 = malloc(5)

  - p3 = malloc(7)

  - free(p2)

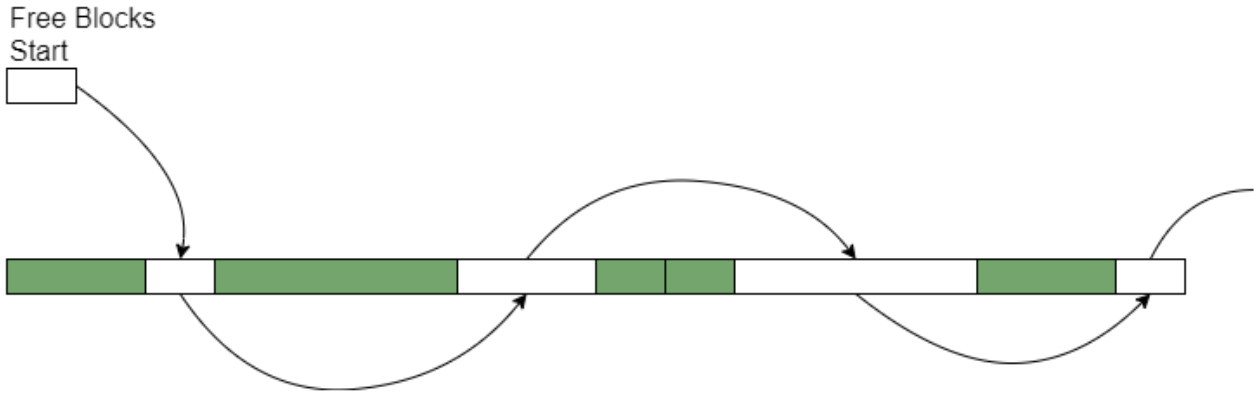
  - p4 = malloc(3)

Memory management

Dynamic memory allocation

- External Fragmentation example:
  - p5 = malloc(3)

  - External fragmentation arises when free memory is separated into small blocks and is interspersed by allocated memory.
  - There are many algorithms on how to mange smaller chunks out of a large block and they differ in speed and memory consumption.
  - However, such algorithms inevitably consume computational power.



an example of fragmentation handling[frag]

Memory management

Dynamic memory allocation

- A custom allocator can have better performance characteristics than the operating system's heap allocator for two reasons:
  - A custom allocator can satisfy requests from a pre-allocated memory block. This allows it to run in user mode and entirely avoid the cost of context-switching into the operating system.
  - By making various assumptions about its usage patterns, a custom allocator can be much more efficient than a general-purpose heap allocator.

Memory management

Stack-based allocators

- Many game allocate memory in a stack-like fashion.
- Whenever a new game level is loaded, memory is allocated for it. Once the level has been loaded, little or no dynamic memory allocation takes place.
- At the conclusion of the level (level cleared or failed), its data is unloaded and all of its memory can be freed.
- It makes a lot of sense to use a stack-like data structure for these kinds of memory allocations.

Memory management

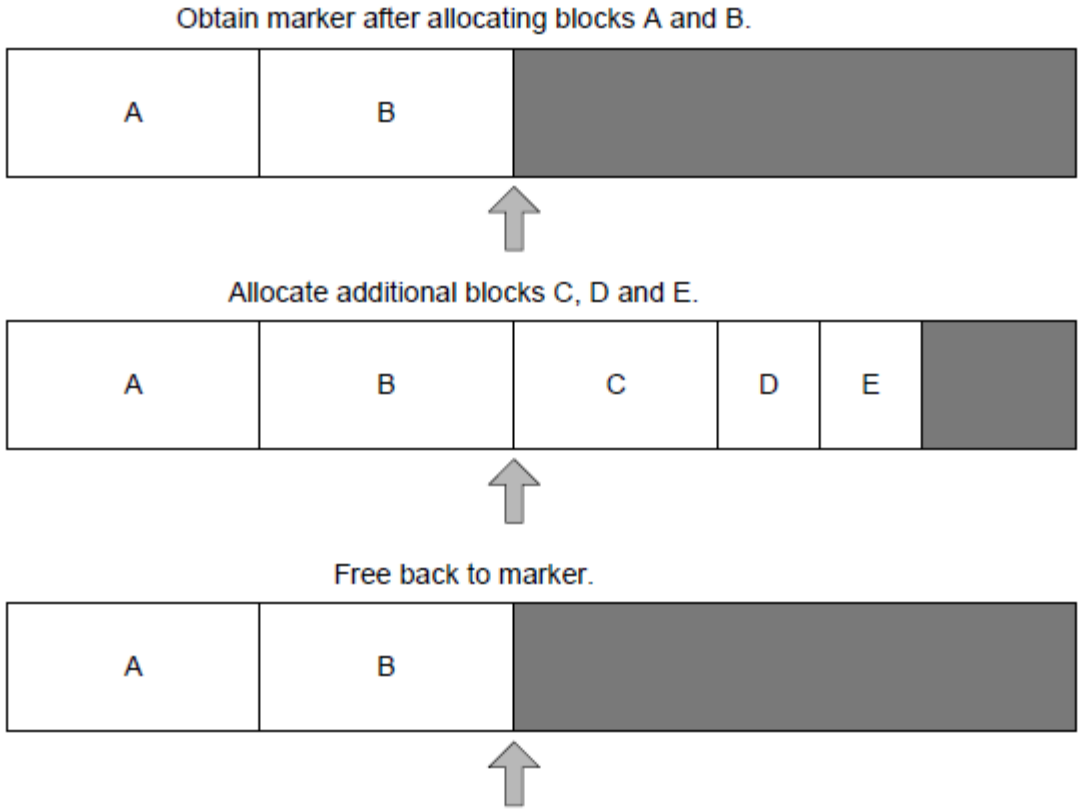
A stack allocator

- A stack allocator is very easy to implement.
- We simply allocate a large contiguous block of memory using malloc() or new(), or by declaring a global array of bytes.
- A pointer to the top of the stack is set and maintained.
- All memory addresses below this pointer are considered to be in use, and all addresses above it are considered to be free.
- The top pointer is initialized to the lowest memory address in the stack.
- Each allocation request simply moves the pointer up by the requested number of bytes.
- The most recently allocated block can be freed by simply moving the top pointer back down by the size of the block.

Memory management

A stack allocator

- Memory cannot be freed in an arbitrary order.
- All frees must be performed in an order opposite to that in which they were allocated.
- To enforce this, it provides a function that rolls the stack top back to a previously marked location, thereby freeing all blocks between the current top and the roll-back point.



Memory management

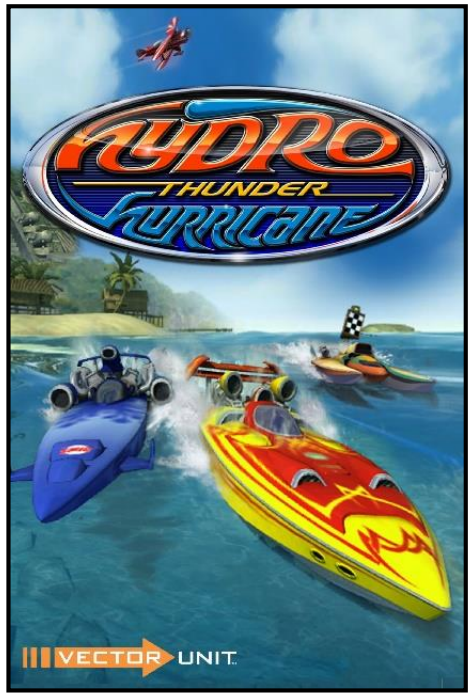
Double-ended stack allocators

- Double-ended stack allocator uses two stack allocator - one that allocates up from the bottom of the block and one that allocates down from the top of the block.
- A double-ended stack allocator is useful because it uses memory more efficiently by allowing a trade-off to occur between the memory usage of the bottom stack and the memory usage of the top stack.
  - In some situations, both stacks may use roughly the same amount of memory and meet in the middle of the block.
  - In other situations, one of the two stacks may eat up a lot more memory than the other stack, but all allocation requests can still be satisfied as long as the total amount of memory requested is not larger than the block shared by the two stacks.

Memory management

Double-ended stack allocators

- In Midway's *Hydro Thunder* arcade game, all memory allocations are made from a single large block of memory managed by a double-ended stack allocator.
  - The bottom stack is used for loading and unloading levels (race tracks).
  - The top stack is used for temporary memory blocks that are allocated and freed every frame.



Memory management

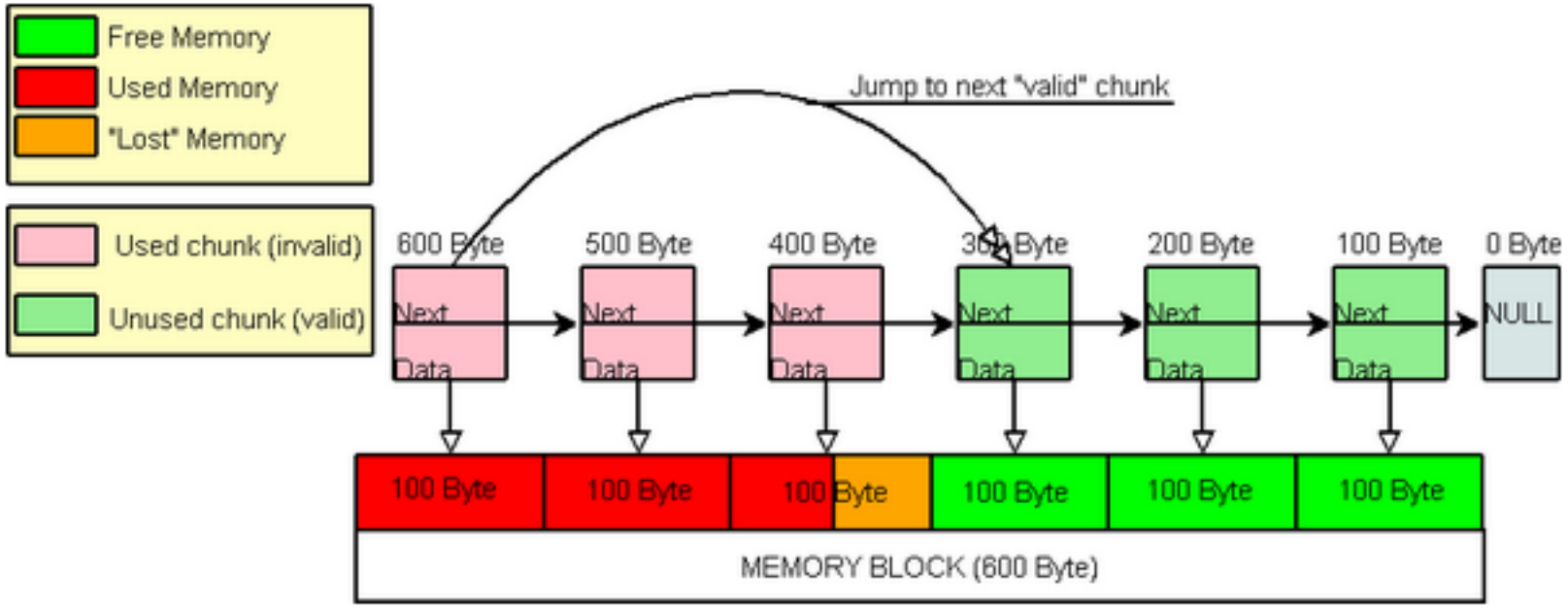
Pool allocators

- It’s quite common in game engine programming to allocate lots of small blocks of memory, each of which are the same size.
  - For example, we might want to allocate and free matrices, or iterators, or links in a linked list, or renderable mesh instances.
- For this type of memory allocation pattern, a pool allocator is often the perfect choice.
- A pool allocator works by pre-allocating a large block of memory whose size is an exact multiple of the size of the elements that will be allocated.
  - For example, a pool of 4×4 matrices would be an exact multiple of 64 bytes - that’s 16 elements per matrix times four bytes (32-bit floats) per element.

Memory management

Pool allocators

- Each element within the pool is added to a linked list of free elements; when the pool is first initialized, the free list contains all of the elements.
- When an allocation request is made, we simply grab the next free element off the free list and return it.
- When an element is freed, we simply track it back onto the free list.



memory pool example[pool]

Memory management

Single-frame and double-buffered memory allocators

- Virtually all game engines allocate at least some temporary data during the game loop. This data is either discarded at the end of each iteration of the loop or used on the next frame and then discarded.
- This allocation pattern is so common that many engines support *single-frame* and *double-buffered* allocators.

Memory management

Single-frame allocators

- A single-frame allocator is implemented by reserving a block of memory and managing it with a simple stack allocator as described above.
- At the beginning of each frame, the stack's top pointer is *cleared* to the bottom of the memory block.
- Allocations made during the frame grow toward the top of the block.
- Benefits:
  - Allocated memory needn't ever be freed - we can rely on the fact that the allocator will be cleared at the start of every frame.
  - Blindingly fast.
- Disadvantages:
  - Using a single-frame allocator requires a reasonable level of discipline on the part of the programmer.
  - Programmers must never cache a pointer to a single-frame memory block across the frame boundary.

Memory management

Double-buffered allocators

- A double-buffered allocator allows a block of memory allocated on frame *i* to be used on frame (*i*+1).
- To accomplish this, we create two single-frame stack allocators of equal size and then ping-pong between them every frame.
- This kind of allocator is extremely useful for caching the results of asynchronous processing on a multicore game console like Xbox360, PlayStation 3 or PlayStation 4.
- On frame *i*, the results can be stored into the buffer we provided.
- On frame (*i*+1), the buffers are swapped. Then the results of the previous frame are now in the inactive buffer, so they will not be cleared in this frame.

```
// Main Game Loop
while (true)
{
    // Clear the single-frame allocator every frame as
    // before.
    g_singleFrameAllocator.clear( );
    // Swap the active and inactive buffers of the double-
    // buffered allocator.
    g_doubleBufAllocator.swapBuffers( );
    // Now clear the newly active buffer, leaving last
    // frame's buffer intact.
    g_doubleBufAllocator.clearCurrentBuffer( );
    // ...
    // Allocate out of the current buffer, without

    // disturbing last frame's data. Only use this data
    // this frame or next frame. Again, this memory never
    // needs to be freed.
    void* p = g_doubleBufAllocator.alloc(nBytes);
    // ...
}
```

Reference

- [heap] https://en.wikipedia.org/wiki/Heap_(data_structure)
- [pqueue] https://www.javamadesoeasy.com/2015/01/priority-queues.html
- [frag] https://johnysswlab.com/the-price-of-dynamic-memory-allocation/
- [pool] https://www.codeproject.com/Articles/15527/C-Memory-Pool