Low Level Game Programming Report

Contents

[Overview / Instructions 2](#_Toc91707450)

[Memory Management 3](#_Toc91707451)

[New and Delete 3](#_Toc91707452)

[Header / Footer 3](#_Toc91707453)

[Heaps 3](#_Toc91707454)

[Memory Pools 4](#_Toc91707455)

[Framework Expansions 5](#_Toc91707456)

[XML File Loading 5](#_Toc91707457)

[Animation 5](#_Toc91707458)

[XML Structure 5](#_Toc91707459)

[Framework Optimizations 6](#_Toc91707460)

[File Writing Improvement 6](#_Toc91707461)

[Threading 6](#_Toc91707462)

[Octree 6](#_Toc91707463)

[Porting to Linux 7](#_Toc91707464)

[Basic Supporting Changes 7](#_Toc91707465)

[Appendix 8](#_Toc91707466)

[1. XML Structure 8](#_Toc91707467)

# Overview / Instructions

Add overview of application + description of the various test methods

My application consists of an integration of my custom memory management and the provided ray tracer framework with my own adjustments.

The main method executes a series of methods. Each method demonstrates a specific part of the assignment:

* MemoryManagerTest
  + This demos the overridden new and delete with various classes (custom and core types) and shows the heap system by creating and walking several heaps.
* MemoryPoolTest
  + This demos the memory pool functionality. It creates a series of objects then deletes one, then creates a new one. It walks the heap between each step and shows that the new object fills the same memory address as the deleted one.
* ThreadingTest
  + This demonstrates basic threading by creating two threads that output a number every second.
* RayTracerTest
  + This is the main ray tracer part of the application. It is setup to render several animation files.

# Memory Management

For the memory management section of the assignment, I implemented a series of techniques that enable memory tracking and error checking, as well as improve performance.

## New and Delete

I implemented both global and class specific overrides for the new and delete operators (as well as their array counterparts). The global overrides are implemented as C style functions. I implemented the class specific overrides in all of the custom classes, and to reduce code duplication, all of the custom classes inherit from a *ManagedObject* class. This class simply provides a wrapper for the global overrides that allows for the specification of the Heap to use (see below).

This also prevents an issue with ambiguity. Due to the nature of C++ if two functions have the same input types and name, they are ambiguous. Visual Studio / C++ includes a core *dll* which provides a new override with arguments of type *size\_t* and *void\**, whereas the custom override I implemented has arguments of type *size\_t* and Heap\*. This isn’t an issue for most class overrides because passing in a *Heap\** allows the compiler to distinguish automatically. However, for the default heap (see below), the *Heap\** is *nullptr*, which cannot be automatically used to decide between the two functions. To avoid the problem entirely and allow for future expansion, I made the decision to make all custom classes which are memory managed inherit from a single class. Since this inherited operator is scoped, there is no ambiguity.

The Global and Class Specific overrides perform the exact same function just that the global override always uses the Default Heap.

## Header / Footer

The override method performs the malloc but adds a custom header and footer to the memory allocation. The header and footer both contain a check value which is a known hex value that can be used to check for corruption. If the check value is no longer equal to the original value, then the memory was corrupted, indicating an issue with the memory management.

The Header also contains information about the size of the allocated memory which is required to accurately delete the memory (preventing memory leaks). It also contains a pointer to a Heap indicating where its memory should be tracked and pointers to the next and previous headers (within the heap)

## Heaps

I have implemented a heap system as well as a manager to control the heaps. The *HeapManager* is implemented as a static singleton allowing the same heaps to be accessed throughout the program. It provides methods to get a heap with a specific key, as well as the default heap which is a static heap separate to the other heaps which are stored in a map.

The *Heap* class is essentially a linked list of *Header\** that also tracks the total memory for the contained headers. They provide methods to add and subtract from the memory total as well as add new headers to the list and remove existing headers.

There is a separate heap for the *DefaultHeap.* This is a child class of *Heap* that specifically provides no *Heap\** when it is created (new override). The default heap is the only dynamically created objects that has no *Heap\** since every subsequent call will use either a specific heap or the default one.

Another feature of the heap manager is owning the current heap scope. The heap scope specifies what heap should be used as the default. It consists of a stack of heap pointers that can be pushed to and popped from. If the stack is empty, then the default heap is used. The primary purpose of the scope is to provide a way to specify which heap base type pointers such as *int\** should use. This is useful when debugging memory issues as it is clear which part of the code a specific type was created in.

## Memory Pools

I have implemented a memory pool system. The memory pools allocated batches of memory and allocate chunks to new objects. When an object is deleted, the chunk is freed up again and a future object to use. This provides more control over the allocation and memory addresses used. In theory it helps reduce the number of allocations performed because they are performed in bulk which is more efficient than many individual allocations.

My system includes a pool manager which stores the various pools. The pool manager creates a new pool for every new size of object. For example, if an object (including header/footer) is 16 bytes it is added to the 16-byte pool. If no such pool exists, then one is created. This works fine in small simple programs where there are only a few sizes of memory required, but in more complex situations it would probably be more performant to change over to a bucket system where the memory is placed in the largest bucket that fits them. Both systems have potential memory waste. The ‘perfect-fit’ system has a very small amount of wasted space in each pool but has a lot of additional overhead especially when it creates a lot of pools with only 1 allocation. The ‘bucket’ system has a smaller amount of pools so less overhead but is vulnerable to wasted space within the chunks as to avoid corruption any difference between the size of the bucket and the size of the allocation cannot be used.

The memory pools are not thread safe. This is because the allocated memory needs to be accessible on all threads and thus needs to be locked. However, the lock must be static so that it is shared between all threads and making a static lock is flawed due to the Static Initialization Order Fiasco. To alleviate the problem the memory pools are disabled when threading is in use.

# Framework Expansions

I implemented several changes and expansions to the provided framework. Notably I added XML file loading and animation. I also integrated my custom memory management and split the code into separate files.

## XML File Loading

I implemented XML file loading to load in the scenes for the ray tracer. To do this I made use of the TinyXML2 library. This provides all of the necessary functionality for loading, saving and parsing XML files.

The loaded scene is stored in an *Animation\** (see below)

## Animation

I implemented a custom animation system for the ray tracer. This system is incorporated into the XML structure I defined.

### XML Structure

See Appendix 1 for image of XML file.

The file consists of a list of spheres. Each sphere is a separate object in the scene. Within each sphere is 1 or more keyframes. Each keyframe defines the state of the sphere at a given frame.

The animation system uses the current frame to lerp between two keyframes. Any of the sphere variables; color, emission, radius, centre (position), reflection or transparency can be lerped between. Since the ray tracer produces a frame at a time it simply queries the animation at a given frame and renders the returned sphere objects.

To further develop the animation system, it would be good to make the keyframes specific to each parameter. This way the parameters could be keyframed at different points allowing for more complex transitions. Though this is technically possible with the current system, it would require manual calculation of the state at each keyframe which could lead to awkward numbers being stored. Additionally providing more complex methods of interpolations such as slerping would improve the complexity.

Another more general expansion would be the addition of other object shapes. Adding cubes, prisms and even complex models would greatly improve the complexity of the generated scenes.

The ability to animate the camera would also improve the dynamism of the system allowing for more cinematic animations than the current static ones.

# Framework Optimizations

I implemented several optimizations into the system.

## File Writing Improvement

The original system used a stream-based method of writing the image data to file. Whilst this is neat on the code side its performance is poor. Instead, I switched it over to use a C style file write method. This is much more performant.

Show Comparison of Original vs C Style (Performance Profiler / Timings)

## Threading

I implemented threading into the system. Threading means the system is able to process multiple frames simultaneously, providing better completion times. To do this I implemented a *ThreadManager*. The thread manager maintains a map of the active threads and handles there availability. The maximum number of threads is a multiple of the hardware capacity that was decided based on performing tests with various numbers of threads over 100 frames of animation.

Show Test

One major benefit of the thread manager vs just launching batches of threads and waiting for completion is that the thread manager has a separate thread that continuously checks to see if a thread is complete. This is achieved in several steps. First when a new thread is spun up it uses a lambda to execute its method. As well as executing the method the thread also updates a list of finished threads as its last action. The check thread, which is continuously looping, checks for when this list isn’t empty. When it contains something, it means a thread has finished. The check thread then joins the specified threads and marks them as available again. This means that threads can be used as soon as they are finished instead of having to wait for the whole batch to be joined. All of the thread manager variables are controlled via mutexes to prevent deadlocks and overwriting.

## Octree

I have implemented an octree system into the ray tracer. The Octree splits the scene into smaller and smaller cubes arranged in a 2x2x2 format. The objects (spheres) within the scene are stored in the octree based on whether they intersect the cubes. The smaller cubes are children of the larger cube that contains them.

The ray for a given pixel is first tested against the octree. It performs AABB-Ray intersections at descending levels of the tree until it reaches a leaf node. The leaf nodes are defined by a child node that has 2 or less objects contained within it, any less and you are doing unnecessary intersections and anymore and there might be too many objects to check.

The octree also stores any spheres that are outside of the largest level, and these are always checked.

The main optimization of an octree like this comes when the scene contains hundreds if not thousands of objects as the resulting sum of AABB intersections is less than checking every object in the scene. However, for the small simple scenes this ray tracer is using, the octree almost always adds additional overhead. This is obvious in the fact that at 1 layer of depth it already has 8 children, up to 4 of which can be intersected with a single ray. This is often already larger than the number of objects in the scene and to make the octree more useful multiple layers of depth would be required so its efficiency would drop even further.

Show Octree vs No Octree Data

## Optimization Evaluation

Once tables are created, evaluate the results here

# Porting to Linux

I chose to port my application to Linux. To do this I used Oracle VM VirtualBox to create a virtual machine, upon which I installed the latest version of Ubuntu (64bit). I then installed VS Code which is Linux compatible and synced my project using github. I then setup the compiler settings for g++ so I could compile and debug on Linux.

## Basic Supporting Changes

Several changes where necessary to make the application Linux compatible in its existing state.

First, I had to wrap any windows specific includes in a *#ifndef \_\_unix* preprocessor operation. I then went through the code and fixed any specific errors. One error that arose was my use of *fopen\_s* which is windows specific. I instead used *#ifdef \_\_unix* and used the standard *fopen*, keeping *fopen\_s* for windows only. I also disabled the automatic *ffmpeg* video generation when on Linux as the implementation required windows specific code. I also had to explicitly define a *ROOT\_DIRECTORY* for both platforms. This is because on windows I was using the ‘.’ in my file paths to indicate the current directory, i.e., where the .exe was running from. On Linux this syntax does not work so I explicitly defined the path to the application folder. In the code where the directory is required, I use the macro ‘*ROOT\_DIRECTORY’* which changes definition based on the platform. This way I don’t have to perform a platform check every time I want to use the directory.

TinyXML2 already supports the Linux platform.

# Appendix

## Text Description automatically generatedXML Structure