DUNN Isaac

[Company name]  [Company address]

Low level programming

Contents

[Content Review 2](#_Toc92412478)

[Memory Management 2](#_Toc92412479)

[Class based memory allocation 2](#_Toc92412480)

[Walking the heap 2](#_Toc92412481)

[Error checking 2](#_Toc92412482)

[Memory pooling 3](#_Toc92412483)

[Expansion of the framework 3](#_Toc92412484)

[Porting 3](#_Toc92412485)

[Optimisation of the framework 5](#_Toc92412486)

[Bibliography 11](#_Toc92412487)

# Content Review

## Memory Management

The memory management portion of the project has several features and as the name suggests manages the memory of the project.

### Class based memory allocation

First to be implemented is a class-based memory allocator via malloc and free methods which in c++ allow the program to dynamically allocate memory giving more control as to when memory can be allocated and freed whenever it is needed whereas previously it would be allocated on start-up and freed on exit which leads to unnecessary data being allocated and data that otherwise could be recycled so a memory allocation system has been set up to allow the program to organise the data throughout runtime.

The memory manager has also been set-up within its own memory management class. This is to keep track of how much memory has been used during runtime without having to use IDE debuggers which not only can be useful to developer but can also give information to the user about what the program is doing. It also separates all allocated data into separate heaps to help with organising the data.

The memory manager also gives new allocations a header and footer to store information about the payload i.e. its size and what heap it belongs to which can be useful for walking the heap and memory tracking. It can also be useful for checking memory integrity by giving it a dead code which can also be checked while walking the heap for error checking.

### Walking the heap

To keep track of how memory is being used a feature has been added to allow the setup of new heaps and for those heaps to be walked through to showcase for info of where and how much data is being put on the heap. This can help check for memory leaks and give debug information on overall data usage.

### Error checking

The use of the dead code method where the header in each block of data assigned to a heap has a small amount of empty data assigned to it this is used to check if data in the heap is being assigned properly allowing the program to check the integrity of the data in its heaps and provide that information to the user.

### Memory pooling

Memory pools have also been implemented into the project to organise memory into blocks of data which can allow for data stored in the blocks to be allocated and deallocated all at once reducing the need to have memory allocated whenever it is needed for example this is used to allocated data for rendering images in groups for each frame rendered with threads. As each block of data is create when needed this reduces the number of calls to malloc and free potentially speeding the program up.

## Expansion of the framework

While Json file extension has been added it has not yet been implemented to allow for animations or other features while this is not currently in the project it would be useful as it could reduce the amount of data to initially process about each of the spheres as currently all the data for the spheres is hard coded into the project and will need to process it when compiling.

## Porting

Some implementations have been done to port the project over to Linux systems as there are many differences between operating systems like Windows and Linux which can make the project fail for example the Windows.h library only works on windows this is used for a number of features like changing console colour so this will need to be changed.

To allow the project to both work in Windows and Linux the use of ifdef directives have been used with a case for Windows and for Linux these are like used similar to if else statements and are used to add conditions for segments of code depending on the platform used so errors can be avoided.

Text

Description automatically generated

Figure : image showing if def directives used in windows with linux and windows code

As seen in fig 1 the \_WIN32 ifdef while on windows shows code normally whereas in the \_\_linux\_\_ shows the code blanked out and vice versa shown in fig 2 this allows code to be effectively skipped while compiling allowing for cross platform compiling without any platform specific issues.

Text

Description automatically generated

Figure : image of code with if def directives in Linux

Changes made to port between Windows and Linux have been via threading as the thread and mutex libraries are not compatible with Linux as Linux uses its own threads and mutexes called pthreads and forking respectively so the changes made are to simply putting Windows and Linux libraries into their respective ifdef directives and as for the thread code see fig 1 on how they differ. Another difference is how mutexes work between platforms as in Linux mutexes automatically lock/ unlock where as Windows requires there to be a manual unlock and lock function before and after the code segment. Overall this shows the differences between platforms in how they handle techniques like threading and race condition issues. All code has been successfully ported over to Linux with the exception of the heap code which current throws a bad alloc error and allocates all memory onto the default heap however it still renders all images.

## Optimisation of the framework

The framework overall has been optimised in a number of ways and has been altered from the original program but to benchmark the program below is some data showing the performance of the unedited program showing CPU and memory usage as well as total memory allocations which is rather low compared to the altered version seen later as the unaltered version uses far less memory and CPU power however it is much slower than the altered version.

A screenshot of a computer

Description automatically generated with medium confidence

Figure : CPU and memory usage in unedited program

A screenshot of a computer

Description automatically generated with medium confidence

Figure : image showing memory allocations in unedited program

Text

Description automatically generated

Figure 5:image of unedited program without threading

One of the changes made being threading which has greatly increased the speed of the project by allowing many instances of the render function to be called in sync with the use of 4 threads making 4 frames in the same time it would take to render 1. As can be seen below threading has drastically decreased render time however an issue occurs where the threads output data at the same time causing it to be merged this can also cause race condition issues where data that both threads have access to is being edited at the same time leading to unwanted outcomes so mutexes have to added to solve this at a cost to performance however this is minimal compared to no threading at all. Another factor to consider while threading is the extra memory and processing cost to the program as while the program can use more cpu threads to complete tasks in sync and therefore faster it however will use more memory as multiple threads are allocating and deallocating memory at once and while this has only increased from around 7MB to 22MB with 4 threads in larger programs it may be important to limit thread count as the memory may exceed capacity.

Graphical user interface, application

Description automatically generated

Figure : CPU and memory usage without threading

Graphical user interface, application

Description automatically generated

Figure : with threading

Text

Description automatically generated

Figure :when threading is added

Text

Description automatically generated

Figure : threading without mutexes



Figure :time taken with threading and mutexes

The addition of the heap manager should also be mentioned as it speeds up the allocation and deallocation of memory by pre-allocating data ready for use. However for best performance removing the debugging screen will further decrease rendering time for the project as walking the heap and outputting data will take extra time.

A screenshot of a computer

Description automatically generated with medium confidence

Figure : image showing memory usage without the heap manager

A screenshot of a computer

Description automatically generated with medium confidence

Figure : image showing memory usage with heap manager



Figure :heap manager without debugging



Figure : heap manager with debugging

Memory pooling allow for large portions of data to be pre-allocated ready for use and deallocated on mass making it an effective way to allocate data for large groups of similar data such as the allocation of vec3f used in the creation of each rendered frame with vec3f used for the image and pixel vectors so allocating 8 at a time can reduce individual calls to malloc. While these speeds up performance issues with memory occur due to the fact that the ability to deallocate blocks is not working relying instead on the memory managers deallocate which while has better performance than simply using delete it still is not operating as expected and likely working less efficiently.



Figure : program with memory pools

Graphical user interface, text, application

Description automatically generated

Figure 14: memory usage with memory pool

Graphical user interface, text, application

Description automatically generated

Figure 15: memory usage without memory pool

A screenshot of a computer

Description automatically generated with medium confidence

Figure 13: image showing memory usage with heap manager but without memory pool

# Bibliography

**There are no sources in the current document.**