Creative Technologies Project: Creating a system based on a Data-Oriented paradigm that compliments an Object-Oriented workflow.

Alexander Allman
Alexander2.Allman@live.uwe.ac.uk
Supervisor: James Huxtable

Department of Computer Science and Creative Technology
University of the West of England
Coldharbour Lane
Bristol BS16 1QY

Research Report

Abstract

To create a system in which an Object-Oriented programmer can gain the performance benefits of the Data-Oriented paradigm. To aid with this an abstraction will be provided for the programmer to interface with. The core systems of this abstraction will be a memory manager and a job system. The memory manager will handle allocations, fetching stored data and the job system will process changes on the data stored by the memory manager in a threaded manner. The report investigates the most appropriate allocators and job system structures as well as data visualisation methods to present comparison data.

Keywords: Data-Oriented; Object-Oriented; Memory Allocators; Data Visualisation; Job System; Threading;

1. Introduction

The aim of this project is to create an interface in which a developer can leverage some of the performance benefits of a Data-Oriented system whilst still utilizing an Object-Oriented codebase. This will be done through a custom-made allocator and a job system for the developer to interface with. A piece of software that is created in a Data-Oriented manner is often more performant than the same piece of software created in an Object-Oriented manner; mainly due to better cache utilization and sequential locality (Meijer, 2019). The large downside to Data-Oriented programming is the removal of the problem domain from the codebase (Fabien, 2018); making it much more difficult for an outside programmer to understand without external documentation. Whilst this project will not be as performant as a fully Data-Oriented system. It adopts parts of the Data-Oriented paradigm to gain some performance whilst not suffering the downsides it comes with.

1.1 Report Aims

Through research this report will come to conclusions on these main items:

1. Which type of memory allocator would be approriate to use in conjunction to a large scale system with frequent allocations and deallocations?
2. What are the best practices for structure and user interactions in Job systems?
3. What are the best practices for gathering profiling data that independent to platform or IDE?
4. Which data visualisation system would be suitable for displaying comparison data in various formats?

2. Research methods

Memory allocators and Job systems are patterns that are commonly used in industry for the performance gains they can provide. Due to the popularity of these systems; methods for developing them and evaluating their pros and cons are well documented. Therefore, this report is based upon Secondary Qualitative research. By being able to examine existing implementations, an insight has been achieved in order to make more educated decisions. These decisions focus on which allocator is most flexible to allow for it to be used in any project, the best structure for a Job system with a semantic based thread-pool and finally how to gather and visualise the data to benchmark and compare the effectiveness of these systems.

3. Research findings

The role of a memory allocator is to take a request for an allocation where a specified amount of data is required and to return a pointer to where to store it. In this projects case, C++'s malloc function will be leveraged (cplusplus, 2019). Although since malloc must work in all use cases it is generic by design. This results in lower efficiency and potential slowdowns, due to sometimes needing privilege escalation (Trebino, 2019; Chen et al, 2016). The way custom allocators overcome this is by using malloc to get a large pool of memory once and manging it internally instead of calling malloc frequently for small allocations (Trebino, 2019). There are then various options on managing the allocated memory.

Linear Allocator

A linear allocator is the most basic form of allocator in which every time the developer wants to do a new allocation; the allocator return its pointer to its latest position then scrubs it along the size of the allocation ready for the next allocation request (Fig 1). Whilst this allocator is the fastest, it is also the least flexible as the user cannot release specific allocations and instead must release the whole block at once (Trebino, 2019).

Fig 1: Linear allocator visualisation (Trebino, 2019)

Pool Allocator

In pool allocators the block of memory is split into many chunks, generally defined by the size of the object you want to store (Aue, 2005). These chunks are then connected via a linked list (Fig 2). This allows for chunks to be deallocated during runtime and reallocated by traversing the list and searching for free unallocated blocks (Trebino, 2019; Ronell, 2004). When traversing the list, if no free or suitable size blocks can be found; a new block is created and linked to the last created blocks header (Trebino, 2019; Ronell, 2004). This solves the primary problem of allocations and deallocations during runtime, however, potentially traversing the whole list each time a new allocation is requested could become very slow. Especially if the system were to used Best fit search to be more memory efficient with allocations.

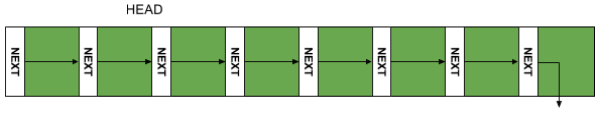

Fig 2: Pool allocator visualisation (Trebino, 2019)

Linked List Free List Allocator

The free list allocator is an extension of the pool allocator and whilst it is slightly slower than the Pool allocator as allocations are not always contiguous due to fragmentation (Hasan and Chang, 2006); it more than makes up for it in flexibility. This is done by storing references to unallocated blocks in a "Free List" (Trebino, 2019; Hasan et al, 2006). Traversing the free list for allocations is much faster as it doesn't have to search through chunks that are already in use. In addition, allocations are much more flexible as the allocator can split blocks into a perfect sized chunk for an allocation (Fig 3) and coalesce adjacent blocks on deallocations (Fig 4). Hasan and Chang (2006) describe further ways of optimising by having multiple segregated free lists. Each of these free lists would be responsible for storing allocations made on a specific size, generally in multiples of 8 as address alignments in most modern computers rest on 8-byte boundaries (Hasan and Chang, 2006). This means on new allocations the allocator can leave out the free lists that contain free chunks that would not contain enough space for the allocation.

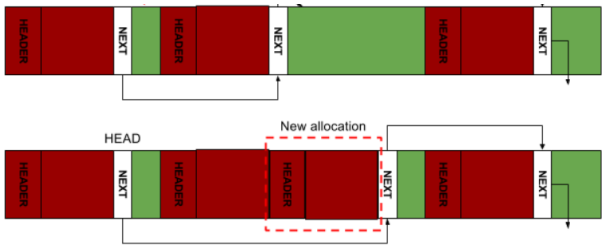

Fig 3: Splitting block for allocation (Trebino, 2019)

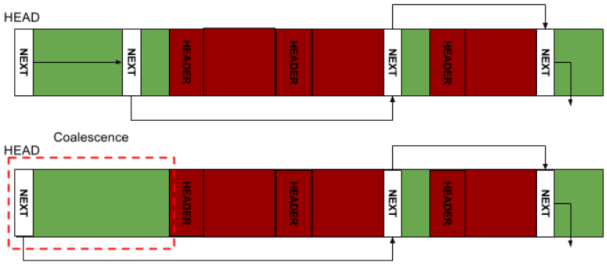

Fig 4: Coalescing blocks on deallocation (Trebino, 2019)

At its core a Job system involves having a queue of tasks that need to be completed. The Job system then takes these tasks and allots them to threads that the Job system creates or has initialised in a pool (János, 2018). The system would then wait for these tasks to be completed and once they are continuing execution of the program. This is what this projects Job system will be based upon but with some extra considerations.

Thread Pool Size

Pooling threads is one of the main optimisations for this system. Instead of creating a thread whenever a job is processed and incurring the overhead that comes with it; the system instead will initialise the threads on start-up at a one-time cost to save on processing time when its more vital. However, the size of the pool is something that needs consideration as having too many threads running at the same time results in thrashing virtual memory from constant context switching (Robison, 2007). The main way of solving this would be to have a test to run at start up to see at which amount of threads thrashing of the virtual memory appears then to set the max amount of threads a ratio of that. This ratio could then be customisable depending on how resource intensive the developer wants their software to be.

Job Batches

At times whilst developing some tasks will be dependant on being done in sequence. Similar to how in mathematics multiplication is done before addition; if they aren't calculated in this order then the result will be completely different. Through using Semaphores, threads can be stalled until others are finished (Walters, 2014). By using this concept, the developer can specify their Jobs into batches and then the Job system will complete all the jobs in batches in sequential order. This will be an optional feature however; allowing data that doesn't have dependencies to be processed in any order to reduce stalling of threads.

Data Gathering

Before data visualisation can be performed a way of gathering the data needs to be created. Since the project isn't interested in performance of the graphical aspects and is purely focused on the speed that certain code snippets are executed. Luckily C++ has a library for taking high resolution time stamps independent of platform (cplusplus, 2019). This will become the basis of a timer class implementation which will return the duration of a specified code snippet. These results can then be stored on a deque of a fixed length and be used by the data visualisation library to generate graphs from. The resolution of the timestamps will be decided once some preliminary data is collected.

Data Visualisation

For data visualisation Dear Imgui will be leveraged (Cornut, 2019). This library allows for widget windows to be rendered in DirectX with the functionality of plotting graphs with existing data (Fig 5). This allows for live data visualisation from the data being gathered mentioned previously. The alternative to this would have been saving and exporting the data out. Then using something akin to Microsoft Excel to build graphs from it. This is much more unfavourable though as it is much more work and does not provide live visualisations of data like the Dear Imgui solution.

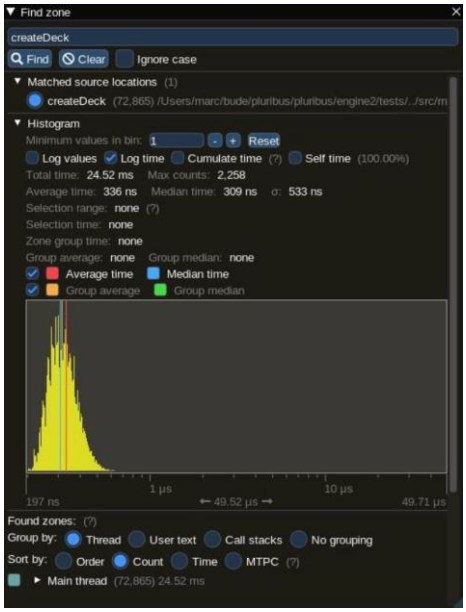

Fig 5: Custom window built using Imgui (Cornut, 2019)

4. Conclusion and recommendations

From here onwards some clear goals are set. Firstly, the current memory allocator implementation will be expanded further with the addition of block splitting and coalescence. Once the base project is complete, segregated free lists will also be added for improved performance. Both combined allow for quicker allocations and more efficient memory usage. Secondly, the job system will be expanded to include the use of semaphores for sequential processing and a benchmarking test to determine the size of the thread pool. Finally, methods for data gathering and visualisation have been decided upon and will be implemented in the coming weeks alongside a test scenario to start gathering data on the project's performance.

5. References

Aue, A. (2005) Improving Performance with Custom Pool Allocators for STL. *Dr.Dobb's The World of Software Development*[blog]. 01 September. Available from: https://www.drdobbs.com/cpp/improving-performance-with-custom-pool-a/184406243# [Accessed 21 November 2019].

Chen, X., Slowinska, A. and Bos, H. (2016) On the Detection of Custom Memory Allocators in C Binaries. *Empirical Software Engineering* [online]. 21 (3), pp. 753-777. [Accessed 21 November 2019].

Cornut, O. (2019) *DearImgui*. Available from: https://github.com/ocornut/imgui [Accessed 21 November 2019].

cplusplus (2019) *malloc*. Available from: http://www.cplusplus.com/reference/cstdlib/malloc/ [Accessed 21 November 2019].

cplusplus (2019) *high_resolution_clock*. Available from: http://www.cplusplus.com/reference/chrono/high_resolution_clock/ [Accessed 21 November 2019].

Fabian, R. (2018) *Data-oriented Design: Software Engineering For Limited Resources and Short Schedules*. United Kingdom: Richard Fabian.

Hasan, Y and Chang, J.M. (2006) A tunable hybrid memory allocator. *Journal of Systems and Software* [online]. 79 (8), pp. 1051-1063. [Accessed 24 November 2019].

János, T. (2018) Simple job system using standard C++. *Wicked Engine Net*[blog]. 24 November. Available from: https://wickedengine.net/2018/11/24/simple-job-system-using-standard-c/ [Accessed 21 November 2019].

Meijer, L. (2019) On DOTS: Entity Component System. *Unity Blog*[blog]. 8 March. Available from: https://blogs.unity3d.com/2019/03/08/on-dots-entity-component-system/ [Accessed 21 November 2019].

Ronell, M. (2004) A C++ Pooled, Shared Memory Allocator For Simulator Development. *Annual Symposium on Simulation* [online]. 1 (37), p. 187. [Accessed 24 November 2019].

Robison, A. (2007) Why Too Many Threads Hurts Performance, and What to do About It. *Codeguru Visual C++ / C++*[blog]. 6 April. Available from: https://www.codeguru.com/cpp/sample_chapter/article.php/c13533/Why-Too-Many-Threads-Hurts-Performance-and-What-to-do-About-It.htm [Accessed 21 November 2019].

Trebino, M. (2019) *memory-allocators*. Available from: https://github.com/mtrebi/memory-allocators [Accessed 21 November 2019].

Walters, A.G. (2014) Semaphores C++11 when Multithreading. *Austin G. Walters*[blog]. 4 May. Available from: https://austingwalters.com/multithreading-semaphores/ [Accessed 21 November 2019].

6. Bibliography

Desrochers, C. (2014) A Fast General Purpose Lock-Free Queue for C++. *Moody Camel*[blog]. Available from: http://moodycamel.com/blog/2014/a-fast-general-purpose-lock-free-queue-for-c++ [Accessed 21 November 2019].

Preshing, J. (2012) An Introduction to Lock-Free Programming. *Preshing on Programming*[blog]. 12 June. Available from: https://preshing.com/20120612/an-introduction-to-lock-free-programming/ [Accessed 21 November 2019].

Sundell, H., Tsigas, P. (2008) Lock-free Deques and Doubly Linked Lists. *Journal of Parallel and Distributed Computing* [online]. 68 (7), pp. 1008-1020. [Accessed 21 November 2019].

Vyukov, D. (2011) *Relacy Race Detector*. Available from: http://www.1024cores.net/home/relacy-race-detector [Accessed 21 November 2019].

Appendix A: Project Log:

Alexander Stephen Allman 16015338	Creating a live custom memory allocator and a system based on a Data oriented paradigm to allow for an Object oriented workflow to benefit from Data oriented performance.		
Date of session / log update	In progress tasks up to this date	Summary of achieved outcomes, be it research or practical	Questions or tasks to be taken forwards
24/09/2019	Refine proposal for a live memory management system.	Do more research into low level memory management. Use DoD but have a wrapper for it so that people using an OOP approach can use it.	N/A
01/10/2019	Ammendments to certain sections of the proposal with less focus on first person.	Final idea has been solidified, Guidance on parts such as the time managment has been adviced on.	Finish the proposal for the 10th. Begin creating a basic memory allocator in C++.
08/10/2019	Finalise proposal for hand-in on the 10th.	Proof Read of the current proposal draft. Given a list of ways to improve it .	Finish the proposal for the 10th. Begin creating a basic memory allocator in C++.
25/10/2019	Begin creating a Memory Allocator in C++.	Created a basic link list allocator using next fit search. Set up a small test case to see that the allocator works.	Extend the allocator to use best fit to get the best sized block for the piece of data being allocated.
01/11/2019	Implement Best fit search into the memory allocator for memory efficient useage of free allocations.	Implemented a basic linked list memory allocator using best fit search.	Start to implement Job system with a threadpooler that uses a metric to measure how many threads to pool.
07/11/2019	Implement the Job system with a thread pool in order the thread tasks for better performance.	A wall was hit today realising that my memory manager isn't set up correctly to be able to handle setting up a job system in its current state. Need to discuss with supervisor.	Implemented Job system with thread pooling. Work out solution to Memory Manager issue. Extend memory allocator to include memory block splitting and coalessing.
08/11/2019	Rework Memory Manager to suit the new proposed NPC layout. Implement Job system utilising the new Memory Manager. Extend memory allocator to include block splitting and coalessing.	Extended the Memory allocator to use a more efficient linked list. Started boilerplate for Job system but behind schedule on that due to issues with the Memory Manager.	Rework Memory Manager. Create NPC and NPC manager. After testing new Memory Manager with NPC test senario, begin fleshing out Job System. Splitting and coalessing of memory blocks.
15/11/2019	Rework Memory Manager. Create NPC and NPC manager.	Fixed issue with pointer arithmatic in memory allocator. Reworked memory allocator for use of generic types. Created NPC and NPC manager and used Memory manager to store their data in a small testcase.	Splitting and coalessing of memory blocks now that memory allocator is fixed. Create Job System. Write the first draft of the research report.
22/11/2019	Research and write the first draft of the research report.	Reserched and written the first draft; spoken to supervisor and obtained feedback to further the report.	Write final draft of research report using the feedback from supervisor. Splitting and coalessing of memory blocks. Create Job System.
29/11/2019	Write final draft of research report.	Final draft is completed. After speaking to supervisor no more ammendments need to be made. Updated electronic log from my paper notes of supervisor meetings.	Splitting and coalessing of memory blocks. Create Job system. Create timer class for getting profiling data.
06/12/2019	Splitting and coalessing of memory blocks Create Job System Create timer class for getting profiling data	Splitting and coalessing of memory blocks is now complete. Researched creating a timer class using STD::Chronos to get high precision timestamps between two predefined points in code.	Create Job system. Create timer class based of research.