DRAFT
An Exploration of Optimisation Techniques for Vulkan-based Particle Systems

Robin Wragg

October 15, 2019

1 Introduction

The aim of this project is to research and experiment with techniques for optimising the rendering speed of particle systems and Vulkan-based rendering pipelines. Techniques are collected from related literature and are applied to our own program, a particle system built in C++ and Vulkan. From this experience, and independent experimentation, we will collect and document the most appropriate techniques for easy consumption by those who may be considering implementing or improving their own particle system and/or Vulkan pipeline.

Multiprocessors will briefly be explored in the literature review, but we will be focussing on consumer-oriented systems with a single, multi-core CPU and a single GPU throughout this paper.

1.1 Particle Systems

Various ephemeral phenomena such as fire, rain, explosions and smoke can be challenging to render convincingly and efficiently using the mesh-of-triangles approach that is used for solid objects.

To create an accurate simulation of these phenomena is impossible to do in real-time, because it would require representing many trillions of individual molecules; completely prohibitive from a performance perspective. Particle systems are a way of approximating the behaviour of these systems. The technique involves rendering many emphparticles per frame; the number of particles is dependant on the intended realism/quality and the required rendering speed. Particle systems are more versatile than just a way to simulate realistic visual effects; they often used to render unrealistic phenomena, such as magical effects.

A particle can be any simple, renderable object; emphbillboards (flat, textured polygons that always face the camera) are perhaps the most common kind of particle, and can represent an individual droplet of rain or a section of a cloud. But a particle can be anything the framerate allows; full-blown meshes can be used, if the amount and complexity are low enough for real-time rendering.

Particle systems are a useful environment for experimenting with performance because the faster the particles render, the more particles are able to rendered without a noticeable drop in framerate, allowing improved visual results. This quantitative nature makes them appropriate for quantifying the performance impact of any changes made to the program.

1.2 Real-time Graphics Pipelines

(todo)

1.3 The Vulkan API

Vulkan is a cross-platform Application Programming Interface for rendering 3D graphics in real-time. It is developed by the non-profit consortium, the Khronos Group. Along with its contemporaries, Microsoft DirectX 12 and Apple Metal, it exists to provide graphics programmers with more control over the entire graphics pipeline, enabling a better balance in workload between the CPU and GPU. This was in response to its precursors, namely DirectX 11 and OpenGL, which were often bottlenecked by the single-threaded performance of the CPU.

1.4 Multi-threading

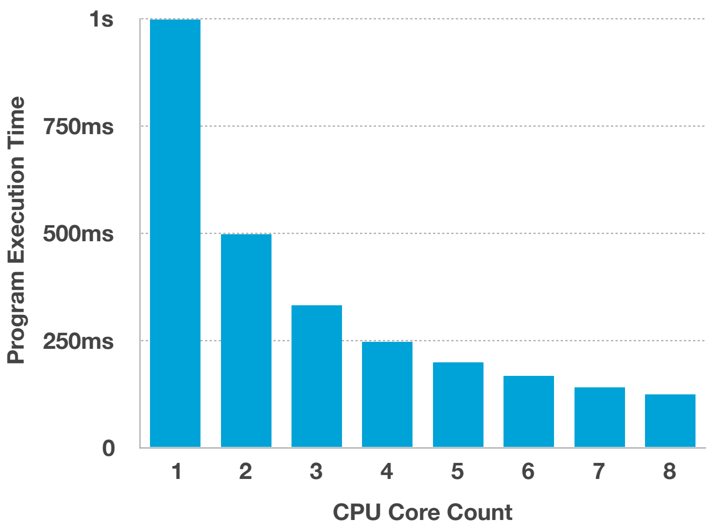

Figure 1: *Theoretical best-case execution time of a program that requires one second of CPU time, by CPU core count.*

Concurrent execution is the very common technique in modern computing of running multiple threads in a single program; the effect is equivalent to two or more processes with access to the same memory. This allows a CPU with multiple cores to run more than one of these threads at once.

All modern consumer CPUs have at least two physical cores, so the performance of a program can be half of its potential if it doesn't utilise multiple threads effectively in its performance-critical sections. Or inversely, a single-threaded program that requires one second of CPU time can reach a fraction of that time if rewritten as a multi-threaded program, but every additional core yields diminishing returns (see Figure 1).

2 Literature Review

2.1 Particle System Design

Tatarchuk (2006) has shown many impressive techniques for rendering various effects for a realistic rainy night in a city. Of particular note was skillful use particles based on pre-blurred normalmaps to create transparent water droplets that have realistic specular highlights from the surrounding street-lights. This is used convincingly both as rain directly from the sky and also as more of a waterfall-like effect to show water gushing out of drainpipes. The water would then splash into puddles using a pre-rendered splash sequence. The otherwise easy-to-spot repetitiveness of this splash was hidden by randomising the size and transparency of those particles, as well as flipping them horizontally. This could perhaps be improved even further by making larger splashes animate slightly slower, so that the actual simulated falling speed of the droplets within the splash would be convincingly similar regardless of particle size. An additional visual improvement could involve spawning new, tiny droplets in a generally upward velocity at the point of the biggest splashes; this could emphasise the impact of the splash.

Boulianne et al. (2007) implemented a biological system simulator using a 3D grid in which each element can hold one or zero particles. Their simulator needed to take into account the spacial locality of particles; a grid facilitates this by removing the need for distance calculation and particle search. Although their implementation did not have real-time graphics in mind, this grid-based approach could be applied to particle-based rendering for situations where the intended effect requires particles to react to each other based on their proximity. Additionally, "this system is expected to be suitable for acceleration with parallel customizable hardware," (Boulianne et al. 2007) meaning this technique would likely be appropriate for rendering real-time particle systems and the parallel nature of GPUs.

2.2 Techniques for Efficient Real-time Graphics

Crawford & O'Boyle (2018) noted that shader compiler optimisation can make a modest improvement to graphics performance, but it is highly dependent on shader code itself. With their test suite of the LunarGlass/LLVM optimisation framework and the GLSL shaders from GFXBench 4.0, they found that shaders can be sped up by as much as 25% by finding the best combination of compiler flags, but 1-4% should be expected in general. Common optimisations that shader compilers can perform are dead code elimination, factoring out conditionals, unrolling loops, coalescing multiple vector element assignments into a single swizzled vector assignment, global value numbering causing variable elimination, and simplifying arithmetic by reordering the statements. The study reviewed only source-to-source optimisations; source-to-machine-code optimisations weren't explored. Further speed-ups could be found in that area.

Referring to Vulkan and DirectX 12, Joseph (2016) states "the central focus of this new generation of APIs is to increase the amount of draw calls possible while decreasing the amount of overhead for the CPU." As graphics programmers, we can reinterpret this to indicate that it is critical to reduce the amount of time that the CPU and GPU are required to block each other to communicate, in order to get the most out of the hardware.

2.3 Multi-threaded Software Design

Coordination between threads is a necessary characteristic of reliable multi-threaded systems (Powell et al. 1991). Without coordination, race conditions and simultaneous unsafe memory accesses can occur, leading to unintended, unpredictable behaviour, often resulting in crashes due to memory access violations. Some ways to avoid these issues are presented below.

Mutexes are perhaps the most common mechanism to ensure thread coordination. A thread can attempt to *lock* a mutex; if the mutex was previously in an unlocked state, the attempt to lock is successful and that thread can continue as normal. The thread now "owns" the mutex. If the mutex is already locked, in most cases the thread will be blocked, and will wait until another thread *unlocks* the mutex (only the owning thread can unlock a mutex). The exception to this is when a `try_lock()` function or equivalent is called on the mutex instead of `lock()`; this will not block the thread, and will instead give the opportunity for the thread to do other work while it waits for Not all mutex implementation have `try_lock()`, but this functionality is available for this project as part of C++'s `std::mutex` (cppreference.com 2019).

Semaphores come in various kinds, but in general they are thread-safe counters. Specifics as to how a semaphore is used is up to its API and the programmer's needs, but a common convention is for a thread to perform a `wait()` operation on a semaphore, which will cause the thread to block until the semaphore's counter is greater than zero. At that point, or if it was already greater than zero, the semaphore will decrement by one and the thread will continue executing. A `post()` can be performed by any thread, which will increment the semaphore. If the counter is above zero and there is at least one thread waiting on the semaphore, one thread can now continue as stated above (Boost 2007).

A semaphore that is always either one or zero is equivalent to a mutex, except for one difference: A `post()` operation can be performed by any thread that has access to the semaphore, allowing any thread to unblock execution, instead of just the thread which called `wait()`.

Semaphores are being introduced into the C++ standard library as part of C++20 (open std.org 2018), so they aren't available to our C++17 environment, but we could use a separate library for this functionality if necessary, such as the Boost library collection's `interprocess_semaphore`, `named_semaphore`, and `anonymous_semaphore`.

Blocked threads are a significant cause of the reduction of maximum theoretical performance on multi-core systems (Alemany & Felten 1992). In the worst case, a *deadlock* can occur when all threads are blocked, waiting on each other indefinitely. In the case of mutexes, a good rule of thumb is to keep the areas of the program that are mutex-guarded as small, simple and fast as possible, to reduce the amount of time that mutexes are locked, thereby reducing the chance of blocked threads.

Although the effects of thread-blocking can be a significant challenge to remove entirely, there exist programming techniques that allow concurrent sections to communicate without blocking each other, such as *exponential backoff*, an algorithm commonly used in network coordination, that slows a process or thread in order to reduce congestion on a shared resource such as an Ethernet node (Goodman et al. 2019), or more applicably for us, a mutex or critical memory. This of course has the downside of one or more threads not operating at their maximum speed but it can have an overall benefit, depending on the bottlenecks of the program.

Optimistic concurrency control is another class of algorithms for non-blocking concurrency that involves validating a transaction performed on shared data before committing it (Herlihy 1993) in an attempt to detect whether data corruption had occurred due to simultaneous writes or reads. Again, this has a downside of requiring extra work per transaction.

3 Methodology

3.1 Equipment

We will be designing, profiling and optimising the program for desktop computer with an Intel Core i5 6400 at 2.7-3.3 GHz with four cores, 8GB of DDR4 RAM at (what frequency?) and an Nvidia Geforce GTX 1060 3GB. This graphics card is the most common among Steam users with a 14.5% share at the time of writing (Corporation 2019). Valve/Steam's published data on their users' CPUs suggests that at least 25% of users own a CPU with the same core count and a similar frequency as our i5 6400. 8GB is also the most popular memory amount at 37.1% of users, so this setup in all should give us a good example of how this program would perform on an average user's system.

3.2 Performance Profiling

(Discuss how I'll be using Vulkan's validation layers, Visual Studio's profiling tools, and concurrency visualisation tools such as RAD Game Tools' Telemetry. Talk about what I'll be looking for when using each tool, and their strengths and weaknesses.)

4 The Initial Implementation

(Describe the design decisions of the program before any optimisation is performed, and any Vulkan-specific implementation details relavant to the project.)

5 Optimisation & Analysis

(Describe in detail all the optimisation techniques I implemented and the changes in the performance characteristics of the program as I make changes to the code. Describe the critical thought processes along the way. Describe the complexity and other difficulties of implementing each optimisation. Towards the end of this section, focus on analysis including comparisons of related optimisations performed. Include charts showing how performance scales with the amount of particles. Include diagrams showing data pipelines.)

6 Conclusion

(Continue to analyse as in the previous section but take a bigger-picture approach and summarise the overall findings. Write in a specific style knowing that some readers will have read the abstract and then jumped to this section.)

References

Alemany, J. & Felten, E. W. (1992), 'Performance Issues in Non-blocking Synchronization on Shared-memory Multiprocessors', *University of Washington* pp. 125–134.

Boost (2007), 'Synchronization mechanisms'. Available at: https://www.boost.org/doc/libs/1_37_0/doc/html/interprocess/synchronization_mechanisms.html#interprocess.synchronization_mechanisms.semaphores (Accessed: 15 October 2019).

Boulianne, L., Dumontier, M. & Gross, W. J. (2007), 'A stochastic particle-based biological system simulator', *Summer Computer Simulation Conference 2007, SCSC'07, Part of the 2007 Summer Simulation Multiconference, SummerSim'07* **2**, 794–801.

Corporation, V. (2019), 'Steam hardware & software survey: September 2019'. Available at: https://store.

steampowered.com/hwsurvey (Accessed: 15 October 2019).

cppreference.com (2019), 'std::mutex'. Available at: https://en.cppreference.com/w/cpp/thread/mutex (Accessed: 11 October 2019).

Crawford, L. & O'Boyle, M. (2018), 'A Cross-platform Evaluation of Graphics Shader Compiler Optimization', *Proceedings - 2018 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2018* pp. 219–228.

Goodman, J., Greenberg, A. G., Madras, N. & March, P. (2019), 'Stability of Binary Exponential Backoff', *Journal of Chemical Information and Modeling* **53**(9), 1689–1699.

Herlihy, M. (1993), 'A Methodology for Implementing Highly Concurrent Data Objects', *ACM Transactions on Programming Languages and Systems (TOPLAS)* **15**(5), 745–770.

Joseph, S. (2016), An Exploratory Study of High Performance Graphics Application Programming Interfaces, PhD thesis, University of Tennessee at Chattanooga.

open std.org (2018), 'P1135r1: The c++20 synchronization library'. Available at: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1135r1.html (Accessed: 15 October 2019).

Powell, M. L., Kleiman, S. R., Barton, S., Shah, D., Stein, D. & Weeks, M. (1991), 'SunOS Multi-thread Architecture', pp. 1–14.

Tatarchuk, N. (2006), 'Artist-directable real-time rain rendering in city environments', *SIGGRAPH 2006 - ACM SIGGRAPH 2006 Courses* pp. 23–64.