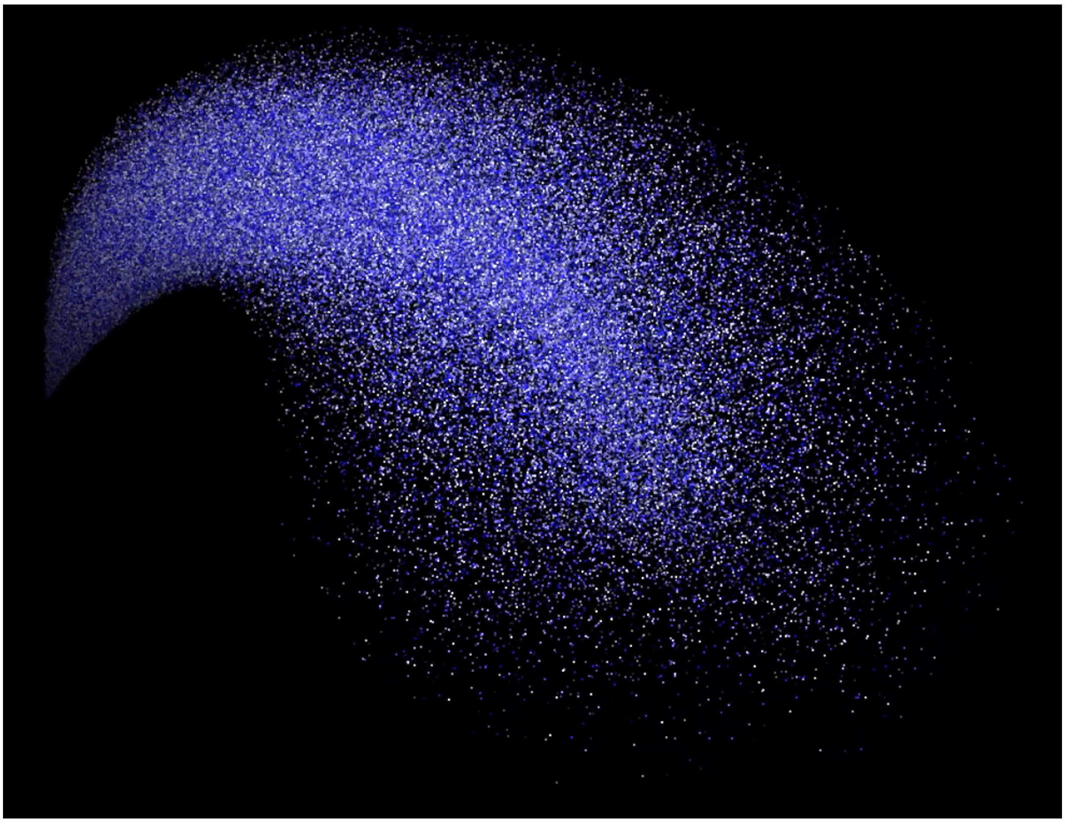
An Exploration of Optimisation Techniques for Vulkan-based Particle Systems



A Technical Report by Robin Wragg

2019

5,380 words

Abstract

A modern, consumer-level computer provides the ability for many tasks to be performed simultaneously and extremely rapidly. However, this power can only be taken advantage of if the software running on it is built and refined with the computer's capabilities in mind. Vulkan is a high-performance, real-time graphics API. It provides the ability for graphics programmers to control graphics processing hardware more explicitly and reliably than previous solutions. Particle systems are a method of efficiently rendering many moving components to a display simultaneously, in order to produce various ephemeral effects including smoke, fire and water.

In this paper, a particle system is built using C++17 and the Vulkan API to serve as a framework for testing and experimentation with graphics performance issues and optimisations. Areas of research include multithreading and Single-Intruction, Multiple-Data (SIMD). The particle system is continually tuned to run as fast as possible on a computer with a Intel quad-core CPU and an Nvidia GTX 1060. Rendering speed is recorded and analysed for a spread of different particle counts, and conclusions are drawn on the efficacy of all optimisation techniques that are attempted throughout the course of the project.

Contents

List of Figures

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++17 and Vulkan. From this research 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 and offline rendering will be briefly explored, but this paper is focussed on real-time applications running on consumer-oriented systems with a single, multi-core CPU and a single GPU.

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 standard for rendering solid objects. This is because creating an accurate simulation of ephemeral phenomena is impossible to do in real-time as 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 *particles* (discrete, simple objects) per frame; the number of particles is dependent on the intended realism or quality, and the required rendering speed. Particle systems are more versatile than just a way to simulate realistic visual effects; they are often used to render unrealistic phenomena, such as magical effects.

A particle can be any simple, renderable object; *billboards* (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; even traditional 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 and other objects in the scene are able to be rendered without a noticeable drop in framerate. This primarily gives more creative freedom to the artists on the project, but a noteworthy effect of simply increasing the particle count is often a *de facto* increase in visual fidelity. This is because an increase in the number of particles will bring the total count slightly closer to the aforementioned trillions of molecules that most particle systems aim to simulate.

Particle systems were chosen for this optimisation-based project because their numbers-based nature makes them appropriate for quantifying the performance impact of any changes made to the program, and any successful optimisations will permit an improved framerate and particle count for this particle system demonstration.

1.2 Offline Rendering

A particle system containing trillions of particles is generally not viable unless rendering is done offline, *id est* before the finished animation is due to play instead of rendering each frame as it is needed. This is done when interactivity is not required, such as the mode of operation for animated film production companies. In offline particle systems, the number of particles is still limited by how fast the frame can be rendered, but a film company can often afford to take hours to render each frame (Cook et al. 1987).

1.3 Real-time Graphics Pipelines

The many GPU-bound data conversions that take place every frame to produce a rendered image from meshes, vectors and matrices is known as a *graphics pipeline*.

In early versions of graphics APIs and hardware, this pipeline could not be modified by the programmer, and was called a *fixed-function* pipeline. Increasingly over the years, flexible technologies such as *programmable shaders* (customisable programs that run on the GPU) were introduced.

In modern graphics software, pipelines are highly customisable; several stages in the pipeline can be added and removed even during the execution of the program, and more kinds of shaders were made available to graphics programmers, such as *tessellation* and *geometry* shaders. Despite this flexibility, many stages and shaders are common across different graphics pipelines as they all have a similar job to do. The most common stages will be discussed below in their typical order of execution.

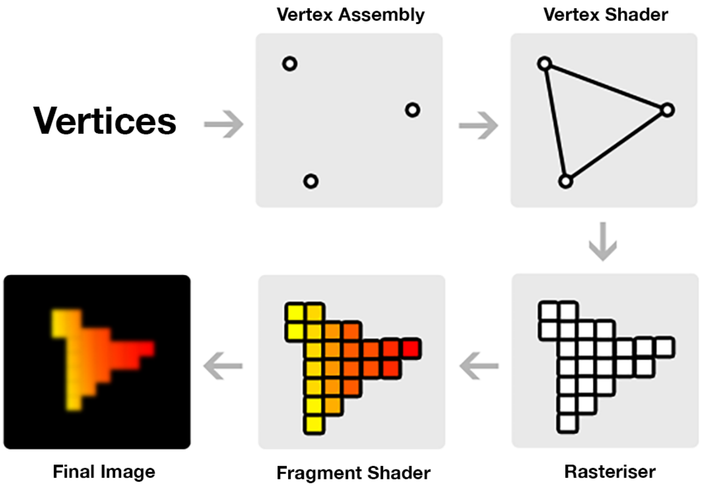


Figure 1: *A diagram of a basic graphics pipeline. Source: Open.gl (2019)*

Vertex Input Assembly is the initial stage, in which data is transferred to the GPU. The majority of this data contains 3D vectors in world-space. These are usually in sets of three which represent the corners of a triangle, the simplest possible shape that has area; this is the minimum requirement for rendering surfaces.

In addition to world-space vectors, each vertex may have a variety of other data associated with it. This is called *per-vertex data* or *vertex attributes*. Some examples are vertex colours, texture coordinates and pre-computed surface normals. Since this stage represents the boundary between the CPU and GPU, data is transferred in large, contiguous blocks, rather than one vertex or triangle at a time (Nvidia 2012), in order to mitigate the relatively slow communication due to the speed limit of electrons between the two components.

A block of per-vertex data may have the corners of triangles laid out in physical memory as *xyzxyzxyzxyzxyzxyzxyz*, or *xxxxyyyyzzzzxxxyyyyzzzz*. These details need to be communicated to the GPU at this stage. A complete explanation and demonstration of why these different memory layouts can be necessary is given later in the paper, when SIMD is implemented in Section 5.5.

Along with the per-vertex data, *modelview* and *projection* matrices for transforming the vertices into *normalised-device-space* are transferred to the GPU, which aid in translating, scaling and rotating vertices and rendering them to a *perspective* or *orthographic* view. However, these transformations are not used in this project; all our coordinates on the CPU are in normalised-device-space as producing a simple orthographic view does not require matrices.

The **Vertex Shader** is the next stage. It is a customisable program that operates on a single vertex at a time. Like all shaders, many instances of the program are running simultaneously to enable fast throughput. The vertex shader involves the actual transformation from world-space to normalised-device-space; the latter is a three-dimensional coordinate system in which each axis extends from -1 to +1. This space makes the pipeline more efficient as anything outside of this range can be discarded before rasterisation.

Rasterisation follows the vertex shader. This is where anything outside of normalised-device-space is discarded. Triangles are converted into *fragments*, which are discrete data in pixel-space. This is done by flattening device-space triangles to 2D, creating fragments based on whether the locations of pixels fall within those triangles.

The fragments of triangles that are hidden behind others are discarded here in a process known as *depth-testing*, which involves the use of a depth *buffer* to record and compare each fragment's depth in the scene, in order to allow triangles to occlude each other correctly.

The **Fragment Shader** is the final stage in a simple pipeline. Its main responsibility is to produce the final colour of each pixel. Any per-vertex data that has been passed on by the vertex shader gets transformed into per-fragment data; a vertex attribute that differs for each of the three corners of a triangle will be interpolated based on the position of the fragment in relation to the vertices (Lighthouse3D 2015). This is useful for lots of tasks including generating per-fragment surface normals efficiently, since there is dedicated hardware on the GPU that performs this task before the fragment shader executes. The interpolated per-fragment data can then be computed by the fragment shader to produce the final colour of the pixels.

There are many more generated fragments than there are vertices. This causes per-fragment operations to be more time-consuming than per-vertex operations, *ergo* a beneficial optimisation to perform in the case of a slow fragment shader is to move work to the vertex shader where possible.

1.4 The Vulkan API

Vulkan is a cross-platform (see Appendix A) Application Programming Interface for rendering 3D graphics in real-time (The Khronos Group 2019*d*). It is developed by the non-profit consortium, the Khronos Group (2019*c*). Along with its contemporaries, Direct3D 12 (Microsoft 2018*c*) and Metal (Apple 2014), 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 Direct3D 11 (Microsoft 2018*b*) and OpenGL (The Khronos Group 2019*b*), which were often bottlenecked by the single-threaded performance of the CPU (Joseph 2016).

1.5 Multi-threading

Concurrent execution is the 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. Amdahl's Law (Rodgers 1985) shown below is a formula describing the theoretical speedup of a task when the system's resources are improved, such as when more CPU cores are added (Hill & Marty 2017):

$$speedup(s) = \frac{1}{(1-p)+\frac{p}{s}}$$

Where *speedup* is the overall speed increase of the entire task, s is the speed increase of the part of the task that is improved by better system resources, and p is the original proportion of execution time that the improved part of the task occupied.

Amdahl's Law in its most optimistic case is when $p = 1$, which greatly simplifies to $speedup(s) = s$. If we improve the system by adding more processing cores, we introduce c, the number of processing cores in the system: $speedup(s) = cs$. Since c is a coefficient of s, every increment to c will result in a continually diminishing increase in $speedup(s)$.

This means that a single-threaded task that requires a constant amount CPU time can reach a fraction of that time if rewritten as multi-threaded, but every additional core yields diminishing returns (see Figure 2).

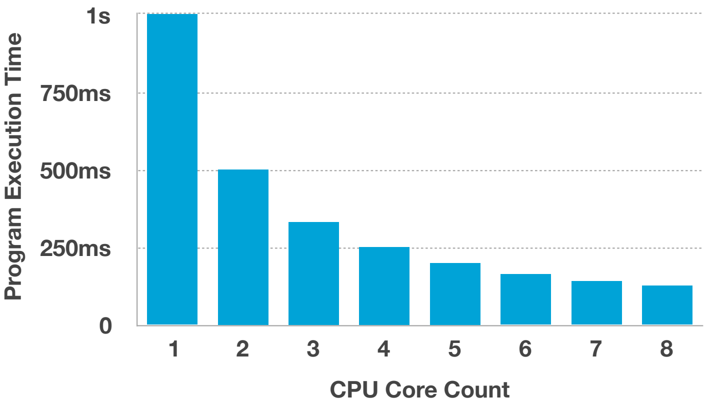


Figure 2: *An example of Amdahl's Law (Rodgers 1985): The best-case execution time of a task that requires one second of CPU time, by processor core count.*

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 of 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 (see Figure 3). The water would then splash into puddles using a pre-rendered splash sequence. This could perhaps be improved even further by making larger splashes animate slightly slower, so that the simulated falling speed of the droplets within the splash would be convincingly similar regardless of their 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.

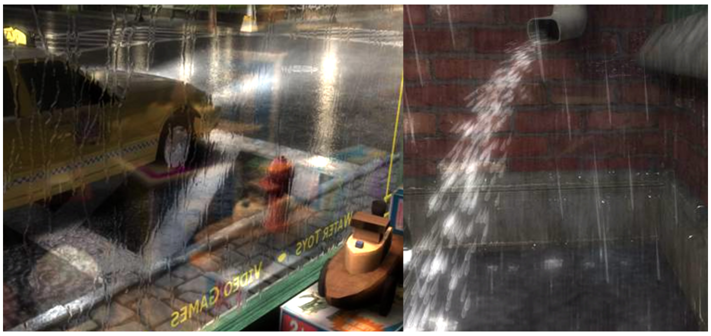


Figure 3: *Work by Tatarchuk (2006) on various rain effects.*

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, a technique known as *spacial hashing* (Lefebvre & Hoppe 2006). 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, Boulianne et al. (2007) states "this system is expected to be suitable for acceleration with parallel customizable hardware," *ergo* 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 (see Appendix B), global value numbering causing variable elimination, and simplifying arithmetic by re-ordering the statements. The study reviewed only source-to-source optimisations; source-to-machine-code optimisations were not explored. Further speed-ups could be found in that area.

Referring to Vulkan and Direct3D 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*c*).

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) *ergo* they are not available in the C++17 environment. Microsoft's `CreateSemaphore()` et al. (Microsoft 2018*a*) was used in this project.

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, programming techniques exist 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

A particle system will be built in C++17 and Vulkan 1.1 to resemble spraying water, similar to the drainpipe work done by Tatarchuk (2006). *Vertical synchronisation*, the option to synchronise the framerate with the vertical refresh rate of the monitor, will be disabled and the program will calculate the duration of the slowest frame out of every one hundred frames (the 99th percentile) and print it to the console.

The 99th percentile of durations is a superior measurement compared to the average duration of a frame for our purpose, as a video game must maintain a framerate at or lower than the refresh rate of a display, particularly when vertical synchronisation is enabled; if a game rendering to a 60Hz monitor (with a $16.\overline{6}$ms frame period) renders most frames every 10ms but one out of every hundred frames takes 17ms, some frames will not reach the display in time and in-game motion will appear to stutter. If the game maintains frame durations of 15ms with no spikes, all frames will reach the display, despite the average frame duration being slower than that of the former case (Intel 2015*a*). 99th-percentile measurements can indicate when the former case exhibits itself.

The amount of particles rendered per frame will be set to 100,000, 200,000, 300,000, 400,000, and 500,000. These numbers will be plotted alongside the frame durations. This will give indications as to whether any optimisation has its greatest effect on high or low particle counts. All measurements and profiling will be done on programs built with the "release" configuration to enable full compile-time optimisation.

The individual functions of the program will be profiled using tools such as Visual Studio's performance analyser (Microsoft 2017*a*) in order to gain an understanding of which parts of the code could be improved. The program will be optimised in stages. At every stage, the performance will be re-examined, new plots will be produced, and the next optimisation steps will be researched and evaluated.

3.1 Equipment

We will be designing, profiling and optimising the program for a desktop computer with an Intel Core i5 6400 at 2.7-3.3 GHz with four cores, 8GB of DDR4 RAM at 2133MHz and an Nvidia Geforce GTX 1060 3GB (see Appendix C). This graphics card is the most common among Steam users with a 14.5% share at the time of writing (Valve 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.

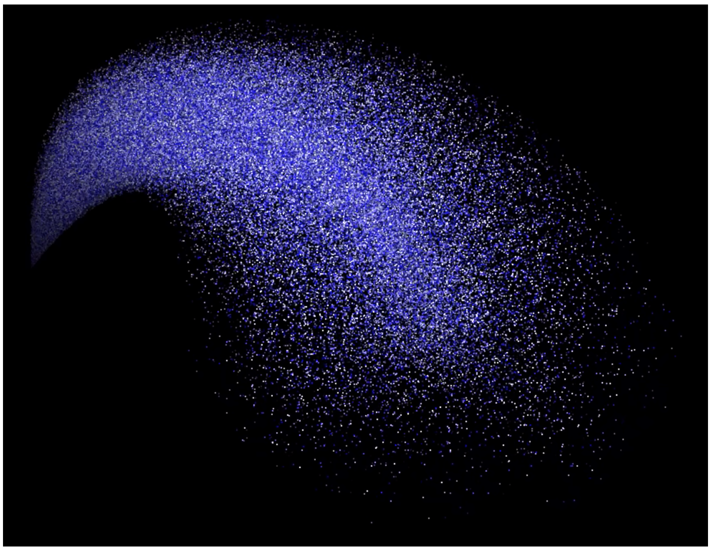


Figure 4: *The particle system, updating and rendering 500,000 particles.*

4 Initial Implementation

A Windows application was built in C++17, using version 2.0.10 of the SDL library (2019) for window creation and frame duration calculation. Version 0.9.9.6 of the GLM library (2019) was used to provide a 3D vector class along with functionality for vector mathematics.

A particles.cpp file was created that updates an array of `Particle` structures every frame, containing 3D positions and scalar brightness values:

```
struct Particle {
  vec3 position;
  float brightness;
};
```

Particle velocities were stored in a separate array since they did not need to be sent to the GPU. Updating the particles functioned by slightly reducing their velocities to simulate air resistance, increasing the downwards velocity to simulate gravity, and applying the velocities to the particles' positions:

```
void updateOneParticle(int32_t i, float stepSize) {
  velocities[i] *= 1 - stepSize * airResistance;
  velocities[i].y += gravity * stepSize;
  particles[i].position += velocities[i] * stepSize;

  if (particles[i].position.y > groundLevel) {
    respawn(&particles[i], &velocities[i]);
  }
}

void update(int particleCount, float deltaTime) {
  float stepSize = deltaTime * 0.5f;

  for (int i = 0; i < particles.size(); i++) {
    updateOneParticle(i, stepSize);
  }
}
```

Notice the call to `respawn()`. When a particle reaches a ground level `1.0f`, this function is called to reset the particle's position, randomise its brightness, and set its new velocity with a degree of randomness.

This algorithm has the problem of not intelligently spreading out the particles evenly across the scene, but this is not an issue in practice as the particles are positioned at a different height at the beginning of the program, such that they each take a different amount of time to hit ground level. This spreads out the respawn events across time. This is not a production-ready respawning system, but is robust enough for the purpose of this project.

A Vulkan 1.1 graphics system was created in graphics.cpp that consumes the array of `Particle` structures and sends them to the GPU. The Vulkan setup code was designed to find a graphics device whose driver supports a single `VkQueue` (a queue for executing GPU-bound commands) that can process both graphical and surface presentation commands. This is done by checking for the `VK_QUEUE_GRAPHICS_BIT` flag and ensuring that `vkGetPhysicalDeviceSurfaceSupportKHR()` returns `true`.

A vertex shader was written to render one point per particle and adjust the brightnesses based on how close the particle is to the viewer:

```
layout(location = 0) in vec3 position;
layout(location = 1) in float brightness;
layout(location = 0) out vec3 outColor;

void main() {
  gl_PointSize = 2;
  gl_Position = vec4(position, 1.0);

  // Set the colour based on brightness and dim the
  // fragment based on the particle's depth (position.z).
  outColor = vec3(brightness, brightness, 1.0);
  outColor *= (0.5 + (position.z-0.5) * 2);
}
```

The fragment shader simply colours the pixels using the value that was passed from the vertex shader:

```
layout(location = 0) in vec3 inColor;
layout(location = 0) out vec4 outColor;

void main() {
    outColor = vec4(inColor, 1.0);
}
```

5 Optimisation & Analysis

Initial measurements were taken of the program's frame durations. Figure 5 shows that these times scale mostly linearly by particle count apart from the duration for 500k particles, which showed slightly better proportional performance. However, it was discovered that the 99th percentile of any sample set of frame durations can vary by up to 10%; the non-linearity of these measurements falls within that margin of error. From this point onwards, efforts were taken to gather typical 99th-percentile measurements that did not appear to be anomalous.

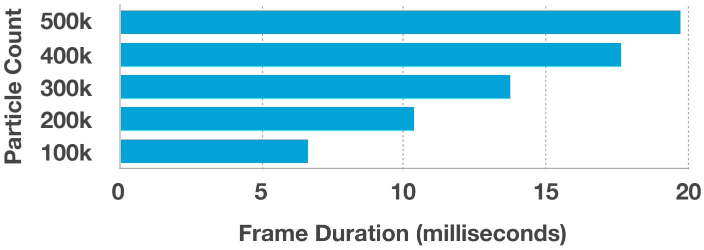


Figure 5: *99th-percentile frame durations for different particle counts, before any optimisations were performed.*

5.1 Multithreading & Random Number Generation

Profiling the program showed that the function that was taking the most time was `updateOneParticle()` in particles.cpp, but its contents was quite minimal:

```
void updateOneParticle(int32_t i, float stepSize) {
  velocities[i] *= 1 - airResistance * stepSize;
  velocities[i].y += gravity * stepSize;
  particles[i].position += velocities[i] * stepSize;

  if (particles[i].position.y > groundLevel) {
    respawn(&particles[i], &velocities[i]);
  }
}
```

Micro-optimisations could be performed here, but parallelising `updateOneParticle()` across all four CPU cores would provide a more significant performance increase. The first attempt at this involved creating four threads at the start of `update()`. Each thread would repeatedly request new particles to update:

```
int32_t requestParticleIndex() {
  int32_t index;
  particleMutex.lock();

  if (nextParticleIndexToUpdate < particles.size()) {
  index = nextParticleIndexToUpdate++;
  } else index = -1;

  particleMutex.unlock();
  return index;
}

void updaterThread(float stepSize) {
  while (true) {
    int32_t i = requestParticleIndex();
    if (i < 0) break;
    updateOneParticle(i, stepSize);
  }
}
```

The function that created the threads would wait for them to finish by calling their `std::thread::join()` method before returning. The program was confirmed to be utilising multiple CPU cores using Visual Studio's diagnostics (Figure 6), where total CPU usage was significantly above $\frac{1}{corecount} = 25\%$.

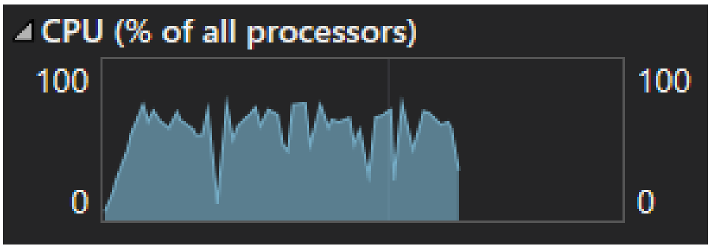


Figure 6: *Diagnostics showing utilisation of multiple CPU cores.*

After these changes a bug was discovered that caused particles to respawn with poor pseudo-random number generation (RNG) for velocities and brightnesses (Figure 7).

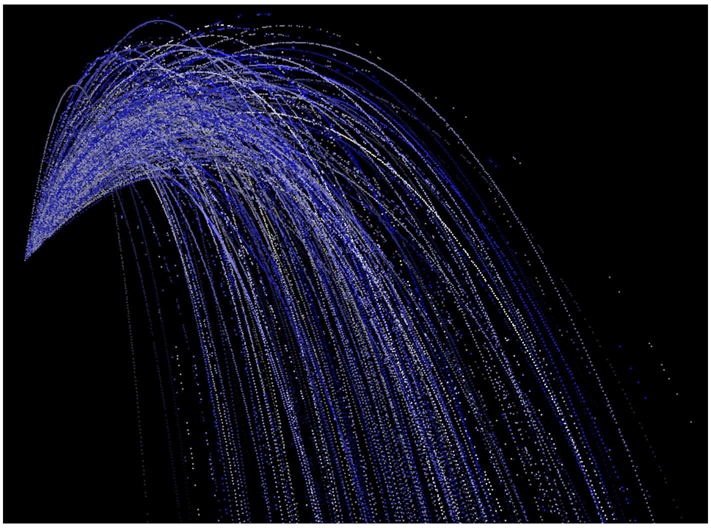


Figure 7: *Unnatural distributions of particles were exhibited after the first multithreading attempt.*

Debugging the program showed that the new multithreading code was performing correctly, so attention was given to the random number generation itself, implemented with the standard C library functions `rand()` and `srand()` (2019*b*). To debug this, calls to these functions were wrapped in mutex locks to ensure no race conditions or data corruption was occurring; this did not change the generated random numbers. To deduce whether the seed generation by `srand()` was thread-local and therefore generating bad values when a seed was not given for each thread, experimental code was written that would re-call `srand()` at the start of each thread instead of just once at the start of the program. The generated numbers still had a poor distribution, and further debugging confirmed that Microsoft's implementation of these functions has some kind of thread-based limitation.

Their documentation on `rand()` does not mention thread-safety (Microsoft 2017*b*), but Linux's documentation states its implementation "is not reentrant or thread-safe" (linux.die.net 2019) due to its internal state, but this still does not explain why Microsoft's `rand()` might not perform correctly across multiple threads, even when guarded by a mutex.

To move on with development, the multithreading code was moved to a separate git branch, and `rand()` and `srand()` were replaced with `mt19937` (cppreference.com 2019*a*), an C++ standard library class that implements the Mersenne Twister RNG algorithm (Matsumoto & Nishimura 1998). This generator requires more CPU time than `rand()`, but it was seen as necessary due to the visual quality of particle systems relying heavily on well-distributed RNG values. The resultant C++ is shown below.

```
// Returns a unit-clamped float using Mersenne Twister.
float randf() {
  static mt19937 randomGen(SDL_GetPerformanceCounter());

  // Generate a number and remove the lower offset.
  float randomNumber = randomGen() - randomGen.min();

  // Return a random number in the range 0.0f-1.0f.
  float range = randomGen.max() - randomGen.min();
  return randomNumber / range;
}
```

Frame durations were recorded for the single-threaded program with the fixed RNG. The multithreading branch was then merged back into master, and frame durations were recorded again.

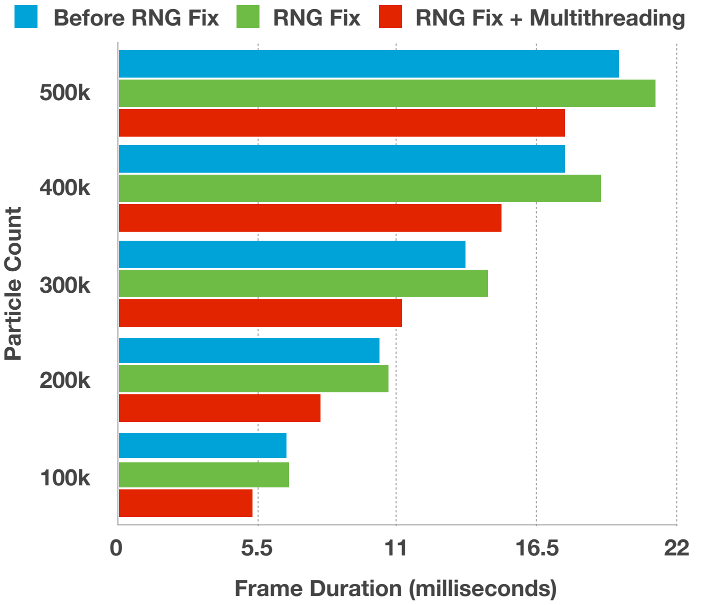
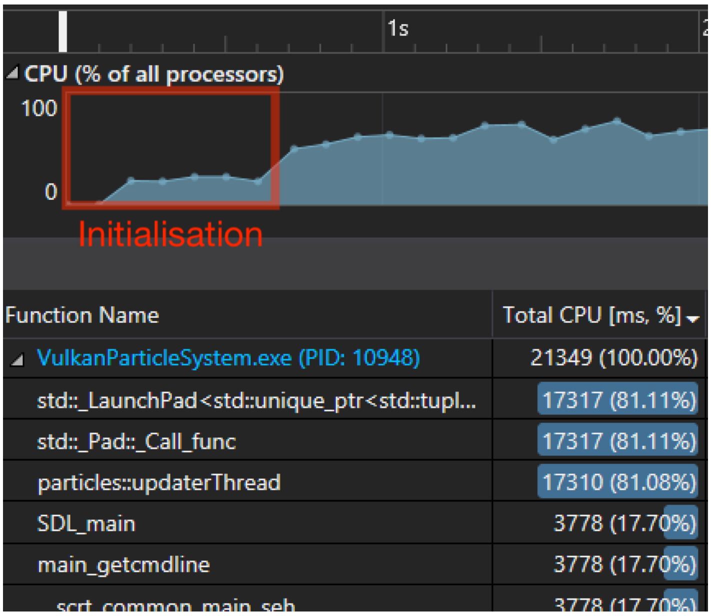


Figure 8: *Changes in performance caused by fixing a bug in the pseudo-random number generation and sharing the particle updates across multiple threads.*

We can see in Figure 8 that the Mersenne Twister RNG causes significantly longer frame durations at high particle counts, but the multithreaded program more than makes up for this. However, Better results than this were expected for multithreading, so investigation began into how to improve the the multithreading model.

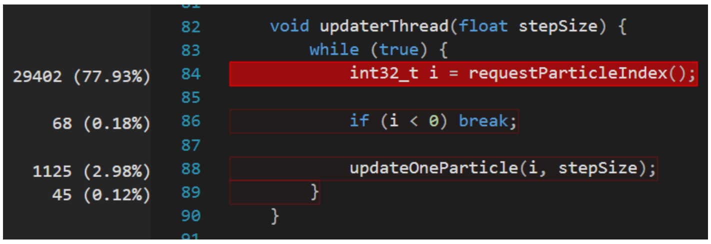
5.2 Mutexless Multithreading

CPU utilisation from 6 was re-evaluated and function execution times were profiled. Figure 9 shows that after initialisation, the program was taking advantage of only 60-80% of the CPU and the vast majority of time was spent executing `particles::updaterThread()`. This is the same for all figures of diagnostics in this paper.). Upon closer inspection of what was taking time inside this function it can be seen that `requestParticleIndex()` was very costly (Figure 10); the operations in this function were guarded by a mutex as described in Section 5.1, so the updater threads were spending a lot of their time waiting on each other there.



Function Name	Total CPU [ms, %]
VulkanParticleSystem.exe (PID: 10948)	21349 (100.00%)
std::_LaunchPad<std::unique_ptr<std::tupl...	17317 (81.11%)
std::_Pad::_Call_func	17317 (81.11%)
particles::updaterThread	17310 (81.08%)
SDL_main	3778 (17.70%)
main_getcmdline	3778 (17.70%)
scrt_common_main_seh	3778 (17.70%)

Figure 9: *The program made less-than-maximum use of the CPU during and after initialisation, spending most of its time in particles::updaterThread().*



```
                    void updaterThread(float stepSize) {
                        while (true) {
29402 (77.93%)              int32_t i = requestParticleIndex();

   68 (0.18%)               if (i < 0) break;

 1125 (2.98%)               updateOneParticle(i, stepSize);
   45 (0.12%)           }
                    }
```

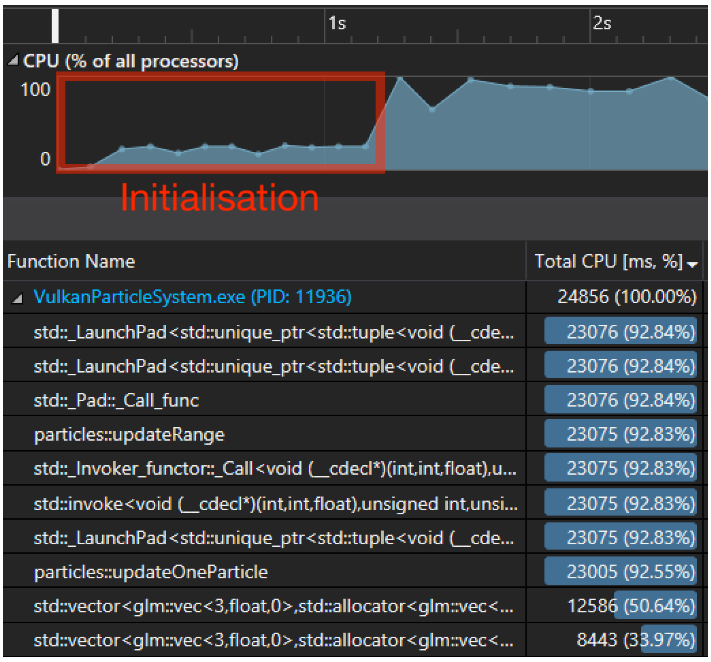
Figure 10: *Profiling showed execution time was dominated by requestParticleIndex(), in which multiple threads were waiting for each other to unlock a mutex.*

A new multithreading model was formulated that did not involve any thread-blocking, except when the threads had finished their work after each frame's updates. Instead of each thread requesting a particle one at a time (which was probably still a significant cost regardless of the mutex issue), threads would be designated a range of particles at the start of an update, and the threads would run until all particles were updated; a thread would only update its own particles, so no resources had to be shared between threads. Below is the replacement for the `updaterThread()` function (`updateOneParticle()` was not changed):

```
void updateRange(int startIndex,
                 int endIndex,
                 float stepSize) {
```

```
  for (int i = startIndex; i < endIndex; i++) {
    updateOneParticle(i, stepSize);
  }
}
```

The new CPU utilisation tended to be much closer to 100% (Figure 11), and most of the CPU time was now consumed by `updateOneParticle()` inside the new `updateRange()` function, which was to become a site of optimisation in Section 5.5.



Function Name	Total CPU [ms, %]
VulkanParticleSystem.exe (PID: 11936)	24856 (100.00%)
std::_LaunchPad<std::unique_ptr<std::tuple<void (__cde...	23076 (92.84%)
std::_LaunchPad<std::unique_ptr<std::tuple<void (__cde...	23076 (92.84%)
std::_Pad::_Call_func	23076 (92.84%)
particles::updateRange	23075 (92.83%)
std::_Invoker_functor::_Call<void (__cdecl*)(int,int,float),u...	23075 (92.83%)
std::invoke<void (__cdecl*)(int,int,float),unsigned int,unsi...	23075 (92.83%)
std::_LaunchPad<std::unique_ptr<std::tuple<void (__cde...	23075 (92.83%)
particles::updateOneParticle	23005 (92.55%)
std::vector<glm::vec<3,float,0>,std::allocator<glm::vec<...	12586 (50.64%)
std::vector<glm::vec<3,float,0>,std::allocator<glm::vec<...	8443 (33.97%)

Figure 11: *Aside from the initialisation stage, all CPU cores were better utilised after a change to mutexless, ranged threads.*

Newly-recorded frame durations (Figure 12) showed a modest performance improvement which appeared to scale well at higher particle counts, although that may have been due to the margin of error in recordings.

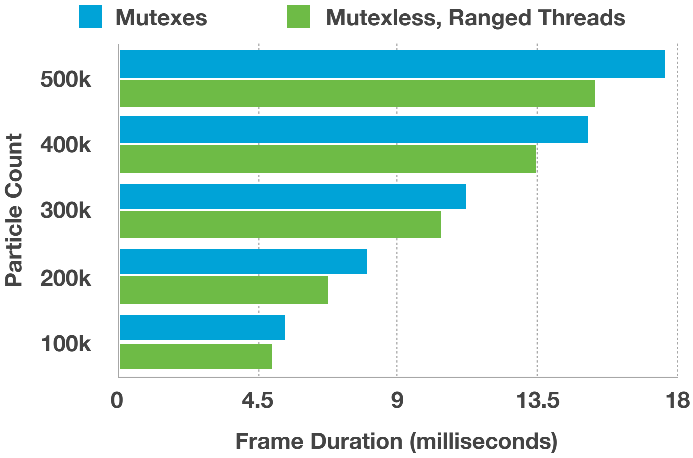


Figure 12: *Overall speed improvements due removing the primary cause of thread-blocking.*

5.3 Passing Particles by Reference

A problematic entry was spotted in the diagnostics at the bottom of Figure 11: "`std::vector<glm::vec<3,float,0>, std::allocator...`" The cause of this was difficult to diagnose as the release build had made much of the surrounding code unreachable from the debugger. Analysis a debug build revealed that large allocations were being performed by the `std::vector<Particle>` parameter to the call to `graphics::render()`.

The compiler, which has the ability to run code analysis and perform optimisations during the build, did not identify that this parameter was not being mutated during the call to `graphics::render()`, in which case it would have been automatically accessed by reference. The parameter's data was being passed by value, allocating and freeing over 1.5 MBs every time the function was called. The parameter was changed to a const reference:

```
void render(const vector<Particle> &particles) {
  ...
}
```

This prevented the parameter from being copied and allowed the program to run up to twice as fast, as shown in Figure 13. The change gave a smaller speedup at lower particle counts due to the smaller size of the data that was being copied in the pass-by-value version of the program.

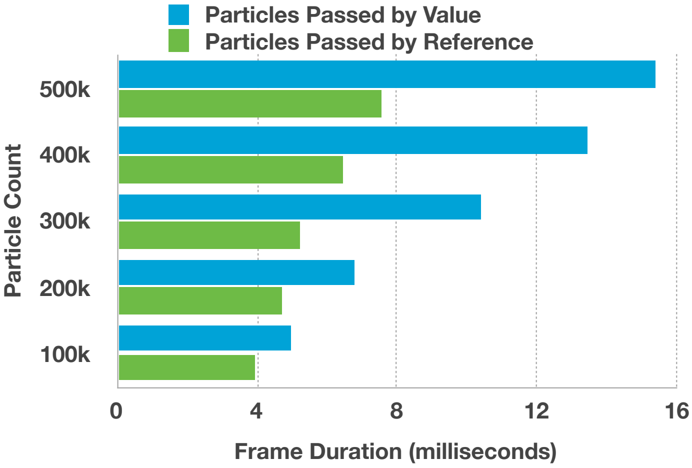
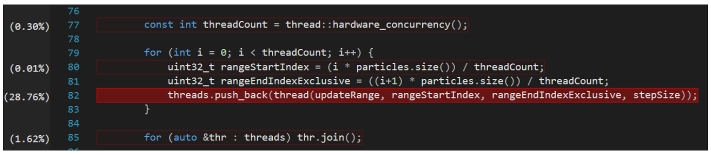


Figure 13: *Dramatic performance gains due to fixing a simple pass-by-value issue that caused large reallocations.*

5.4 Re-Using Threads



```
(0.30%)        const int threadCount = thread::hardware_concurrency();

               for (int i = 0; i < threadCount; i++) {
(0.01%)            uint32_t rangeStartIndex = (i * particles.size()) / threadCount;
                   uint32_t rangeEndIndexExclusive = ((i+1) * particles.size()) / threadCount;
(28.76%)           threads.push_back(thread(updateRange, rangeStartIndex, rangeEndIndexExclusive, stepSize));
               }

(1.62%)        for (auto &thr : threads) thr.join();
```

Figure 14: *Profiling showed that creating threads every frame had a significant cost and was an obvious target for improvement.*

During the search for the cause of the previous problem, thread creation was found to be very costly (Figure 14), causing a taking 1-2 milliseconds before work within the created threads could start. The multithreading model was improved again, using a dual-semaphore system that creates updater threads at initialisation and re-uses them in the update loop, illustrated with the following pseudo-code:

```
Semaphore onUpdateStart;
Semaphore onUpdateComplete;

function initialise() {
  for i in updaterThreadCount {
    createThread(updaterThread);
  }
}

function updaterThread() {
  while true {
    onUpdateStart.wait(1);
    updateParticleRange();
    onUpdateComplete.post(1);
  }
}

function updateAllParticles() {
  onUpdateStart.post(updaterThreadCount);
  onUpdateComplete.wait(updaterThreadCount);
}
```

When `updateAllParticles()` is called on the main thread, it allows the updater threads to continue while it waits. When all updater threads have updated their own particle ranges, they will notify `updateAllParticles()`, which will return and the main thread will move on to the rendering phase. The updater threads will wait for the next update at the top of their `while` loops.

Modest improvements were found in the results at high particle counts, with more significant improvements toward the lower counts (Figure 15). The cause of this needs further investigation, but we can speculate that repeated thread creation and destruction in the older version of the program might have had a throughput bottleneck; perhaps there is a FIFO queue-like structure for thread creation and/or destruction at the operating system level that has a maximum size, and processes must wait if the queue is full. This would be a more significant cost for low particle counts because the program would update more frequently, therefore keeping the queue full by attempting to create and destroy more threads per second.

5.5 Single Instruction, Multiple Data

The next task was to rewrite the update code in SIMD instructions, informed by the relative slowness of `updateOneParticle()`. CPU instructions generally operate on a single data point per instruction, but a SIMD instruction can operate on multiple data points, usually without a significant increase in execution time. In this project, we perform these instructions through the use of intrinsics; functions that we can write in C++ that the compiler turns into specially optimised machine code, including SIMD instructions. In the majority of cases, an instrinsic maps to a specific SIMD instruction.

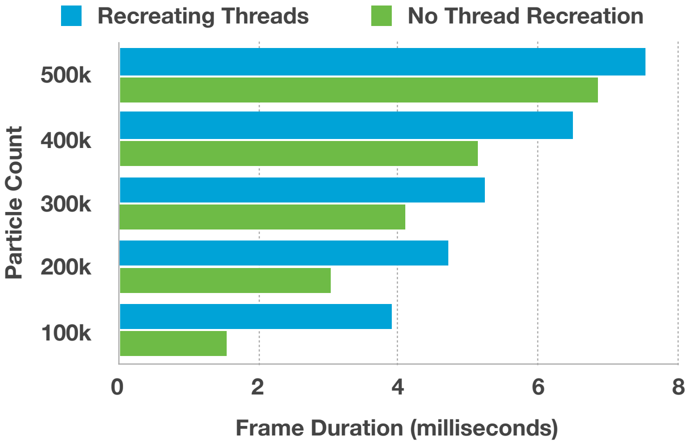


Figure 15: *The removal of thread creation from the frame loop generated frame duration reductions of up to 2.35ms, reducing in significance as the particle count grew.*

SIMD and intrinsics are not available on all x86-64 processors. The Intel Core i5 6400 processor supports multiple extensions to the standard x86-64 (Intel 2015*b*); this project's update phase is based on 32-bit floats, so the most appropriate extensions are the Advanced Vector Extensions (AVX) version 1. These extensions include instructions that operate on 32-bit floats with a *width* of eight, meaning each instruction can operate on eight floats at once.

An example of one of the intrinsics that are used in this project is `_mm256_add_ps(__m256 a, __m256 b)` (Intel 2019), which returns another `__m256`. This is a datatype which contains eight contiguous 32-bit floats ($32 \cdot 8 = 256$), known as a *vector*; literally an eight-dimensional vector of floats, hence why SIMD is sometimes referred to as *vectorisation*. `_mm256_add_ps(__m256 a, __m256 b)` takes a pair of these vectors, adds them together, and returns the result as another vector, enabling us to add eight pairs of floats together using only one instruction.

5.5.1 Memory Layouts

Consider the creation of an array in C++:

```
vec3 array[8];
```

This will lay out the components of `vec3` in physical memory in this order: *xyzxyzxyzxyzxyzxyzxyzxyz*. This is of course the most common way to store many instances of a `struct` in C or C++, and is known as *Array-of-Structures* or AoS (Blow 2019). However, correct usage of eight-wide SIMD requires that the components should be packed contiguously in memory, which means eight `vec3`s must store all their x components together, and the same for y and z. One memory layout that satisfies this is *xxxxxxxxyyyyyyyyzzzzzzzz*, known as *Structure-of-Arrays* or SoA (Blow 2019). This can be written in C++ as follows, hence its name:

```
struct {
  float xComponents[8];
  float yComponents[8];
  float zComponents[8];
};
```

5.5.2 Implementation

To make best use of SIMD in this project, all frequently-accessed particle data needed to be rearranged so that the components were adjacent. Here are the resultant `std::vectors` of `__m256`:

```
std::vector<__m256> positionsX;
std::vector<__m256> positionsY;
std::vector<__m256> positionsZ;

std::vector<__m256> brightnesses;

std::vector<__m256> velocitiesX;
std::vector<__m256> velocitiesY;
std::vector<__m256> velocitiesZ;
```

Each element of a `std::vector` now contains data of eight particles, so each `std::vector` was resized to $\frac{particlecount}{8}$ and filled with particle data at initialisation, using `_mm256_set*` instructions in loops. Each call to this class of instructions takes floats as parameters and efficiently arranges them into a `__m256`, which can then be placed into the `std::vectors`. Below is an excerpt from the new, SIMD-enabled `updateRange()` function, calculating the *x* components of particle positions:

```
for (uint32 i = rangeStart; i < rangeEnd; i++) {
  ...
  // SIMD equivalent of pos.x += vel.x * stepSize;
  positionsX[i] = _mm256_add_ps(positionsX[i],
    _mm256_mul_ps(velocitiesX[i], stepSizeVector));
  ...
}
```

This technique was repeated for the entire body of code responsible for spawning and updating particles. As shown in Figure 16, this gave a speedup of approximately 25%.

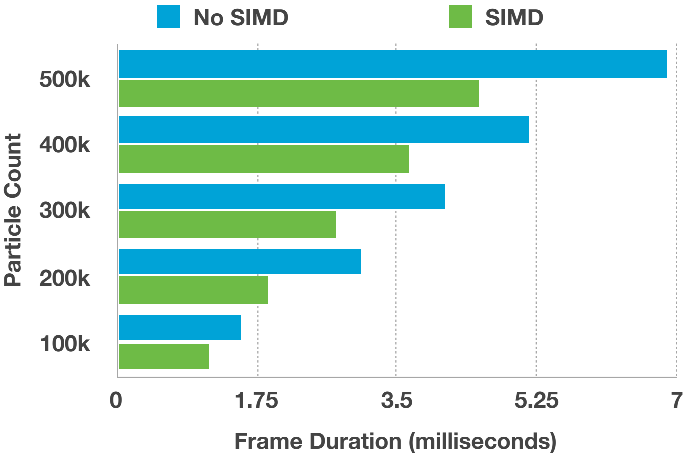


Figure 16: *Updating particles using eight-wide AVX intrinsics resulted in a significant and linear speed increase with respect to the particle count.*

6 Final Analysis

In the final build, the frame duration was found to be inversely and linearly proportional to the particle count. The exception to this was with particle counts below approximately 80,000, where the total cost of updating them became outweighed by the combined cost of all other per-frame operations. Figure 17 shows frame durations approached a minimum of 450µs.

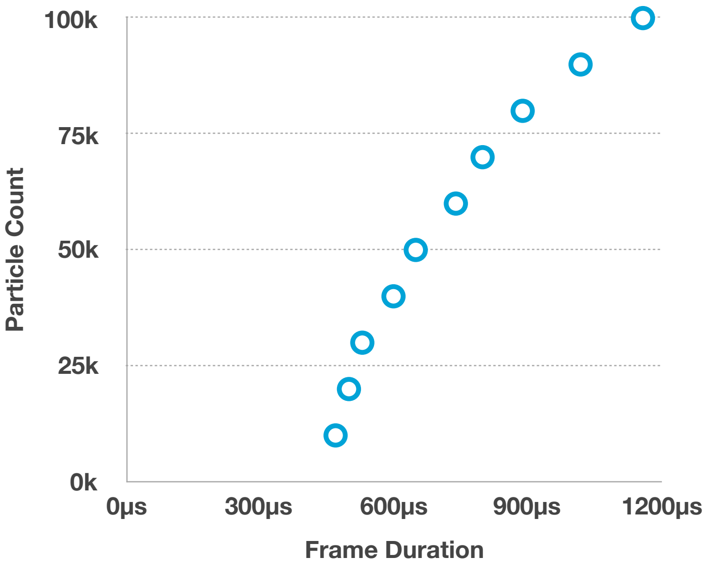


Figure 17: *Frame durations in the final program do not scale linearly with particle counts below 80,000.*

Relative speed increases after each stage of optimisation were compared in Figure 18. The first implementation of multithreading gave the most gains overall but at 500,000 particles, the primary contributor to faster performance was the work from Section 5.3 to pass particles to `graphics::render()` by reference, bolstering the *a priori* knowledge that copying large amounts of data has a high cost.

At 100,000 particles, the re-use of threads from Section 5.4 gave the largest speed increase, showing that frequent thread-recreation was significant performance cost in the previous multithreading implementations that went unnoticed due to most profiling being performed at higher particle counts. This demonstrates the importance of profiling the performance of a program in multiple scenarios and configurations.

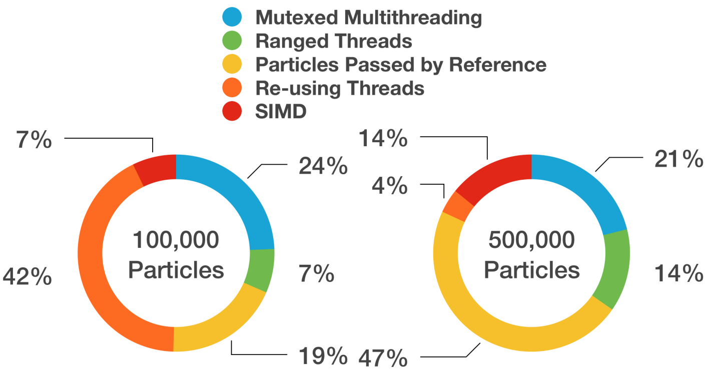


Figure 18: *Individual speedups caused by each optimisation phase, shown as percentages of total speedup over the initial version of the program.*

7 Conclusion

A large breadth of research topics across particle systems, multithreading, general graphics techniques and biological simulation proved beneficial for providing mental models for time-limited applications. The research in multithreading in particular contributed to an understanding the various potential solutions of the problem domain.

The particle system's performance was demonstrated to be entirely CPU-bound throughout all phases of development. Relevant optimisations centred around moving data between tasks efficiently and producing an appropriate multithreading model. This was done by mitigating the cost of creating threads and reducing the necessity for inter-thread communication. Additional productive work on parallelisation was implemented via the use of SIMD intrinsics.

A promising area of further research and development is to offload parallelisable operations to the GPU via the Vulkan API's support for compute shaders (Thomas 2014).

References

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

Apple (2014), 'Metal - Apple Developer'. Available at: https://developer.apple.com/metal (Accessed: 5 November 2019).

Blow, J. (2019), 'Discussion of SIMD, SOA, AOSOA. Followed by Questions & Answers'. Available at: https://www.youtube.com/watch?v=YGTZr6bmNmk (Accessed: 12 November 2019).

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.

Cook, R. L., Carpenter, L. & Catmull, E. (1987), 'The Reyes Image Rendering Architecture', *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1987* **21**(4), 95–102.

cppreference.com (2019*a*), 'Mersenne Twister Engine'. Available at: https://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine (Accessed: 10 November 2019).

cppreference.com (2019*b*), 'rand - cppreference.com'. Available at: https://en.cppreference.com/w/c/numeric/random/rand (Accessed: 10 November 2019).

cppreference.com (2019*c*), '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.

GLM (2019), 'OpenGL Mathematics'. Available at: https://glm.g-truc.net/0.9.9/index.html (Accessed: 6 November 2019).

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.

Hill, M. D. & Marty, M. R. (2017), 'Amdahl's Law in the Multicore Era', *Computer* **50**(6), 12–14.

Intel (2015*a*), 'An Often Overlooked Game Performance Metric - Frame Time'. Available at: https://software.intel.com/en-us/articles/an-often-overlooked-game-performance-metric-frame-time (Accessed: 12 November 2019).

Intel (2015*b*), 'Core i5 6400 Processor'. Available at: https://ark.intel.com/content/www/us/en/ark/products/88185/intel-core-i5-6400-processor-6m-cache-up-to-3-30-ghz.html (Accessed: 11 November 2019).

Intel (2019), 'Intel Intrinsics Guide'. Available at: https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=AVX (Accessed: 11 November 2019).

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

Lefebvre, S. & Hoppe, H. (2006), 'Perfect Spatial Hashing', *ACM SIGGRAPH 2006 Papers, SIGGRAPH '06* pp. 579–588.

Lighthouse3D (2015), 'GLSL Tutorial – Rasterization and Interpolation'. Available at: https://www.lighthouse3d.com/tutorials/glsl-tutorial/rasterization-and-interpolation/ (Accessed: 13 November 2019).

linux.die.net (2019), ‘srand(3) - Linux man page’. Available at: https://linux.die.net/man/3/srand (Accessed: 10 November 2019).

Matsumoto, M. & Nishimura, T. (1998), ‘Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator’, *Discrete Mathematics* .

Microsoft (2017*a*), ‘Measure app performance in Visual Studio’. Available at: https://docs.microsoft.com/en-us/visualstudio/profiling/?view=vs-2017 (Accessed: 6 November 2019).

Microsoft (2017*b*), ‘rand | Microsoft Docs’. Available at: https://docs.microsoft.com/en-us/previous-versions/398ax69y(v%3Dvs.140) (Accessed: 10 November 2019).

Microsoft (2018*a*), ‘CreateSemaphoreA function’. Available at: https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-createsemaphorea (Accessed: 13 November 2019).

Microsoft (2018*b*), ‘Direct3D 11 graphics’. Available at: https://docs.microsoft.com/en-us/windows/win32/direct3d11/atoc-dx-graphics-direct3d-11 (Accessed: 5 November 2019).

Microsoft (2018*c*), ‘Direct3D 12 graphics’. Available at: https://docs.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-graphics (Accessed: 5 November 2019).

Nvidia (2012), ‘Appendix A: Optimizing for the CPU and GPU’. Available at: http://developer.download.nvidia.com/NsightVisualStudio/2.2/Documentation/UserGuide/HTML/Content/Appendix_A_Optimizing.htm (Accessed: 13 November 2019).

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).

Open.gl (2019), ‘The Graphics Pipeline’. Available at: https://open.gl/drawing (Accessed: 6 November 2019).

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

Rodgers, D. P. (1985), ‘Improvements in Multiprocessor System Design’, *Conference Proceedings - Annual Symposium on Computer Architecture* pp. 225–231.

SDL (2019), ‘Simple DirectMedia Layer - Homepage’. Available at: https://www.libsdl.org (Accessed: 6 November 2019).

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

The Khronos Group (2019*a*), ‘MoltenVK Runtime User Guide’. Available at: https://github.com/KhronosGroup/MoltenVK/blob/master/Docs/MoltenVK_Runtime_UserGuide.md#limitations (Accessed: 12 November 2019).

The Khronos Group (2019*b*), ‘OpenGL Documentation Overview’. Available at: https://www.opengl.org/documentation (Accessed: 5 November 2019).

The Khronos Group (2019*c*), ‘The Khronos Group Inc’. Available at: https://www.khronos.org (Accessed: 5 November 2019).

The Khronos Group (2019*d*), ‘Vulkan Overview’. Available at: https://www.khronos.org/vulkan (Accessed: 5 November 2019).

Thomas, G. (2014), ‘Compute-Based GPU Particle Systems’, *Game Developers Conference GDC 2014* .

Valve Corporation (2019), ‘Steam Hardware & Software Survey: September 2019’. Available at: https://store.steampowered.com/hwsurvey (Accessed: 15 October 2019).

Appendix

A MoltenVK

Apple does not support Vulkan in any capacity, but MoltenVK and a Vulkan macOS/iOS framework exist as part of the Vulkan Portability Initiative (The Khronos Group 2019*a*). MoltenVK is a translation layer that sits between Apple’s Metal SDK and a Vulkan application. It can be thought of as a simulation of a Vulkan-capable GPU, enabling graphics developers to run their programs without having to write a bespoke Metal graphics backend. Tests on the cross-platform capabilities of Vulkan could feature the upcoming project on light and shadow rendering; it would be interesting to investigate to what extent the MoltenVK layer degrades performance compared to writing a program in Metal from scratch.

B Swizzling

Swizzling is syntactic sugar for assigning disparate sets of vector components; if we want to assign the x and y components of a `vec3` to the z and y components respectively of another `vec3`, we can do so in some languages and vector implementations simply as follows, in this GLSL as an example:

```
// Assignment without swizzling
vecB.z = vecA.x;
vecB.y = vecA.y;
```

```
// The same assignment with swizzling
vecB.zy = vecA.xy;
```

Graphics processors are often designed with the knowledge that this is a common operation, which allows swizzling to be more efficient than the non-swizzled version of the operation. Due to their efficiency, many optimising compilers will attempt to identify when non-swizzled operations can be swizzled, and will make that change before or during compilation.

C Full Computer Specification

- Windows 10.0.17763 with Windows SDK version 10.0.16299
- Asrock H110M-ITX/AC DDR4 Motherboard
- SanDisk SSD PLUS 240 GB 2.5 inch drive, up to 530 MB/s, Sata III
- Gigabyte Nvidia GTX 1060 3GB Windforce 2 with DDR5 VRAM
- Crucial 8 GB DDR4 2133 MT/s (PC4-17000) DIMM 288-Pin Memory
- Intel i5 6400 Skylake 2.7GHz Quad Core 1151 Socket Processor