CHALMERS

Particle System Simulation and Rendering on the Xbox 360 GPU

SEBASTIAN SYLVAN

Master's Thesis
Computer Science and Engineering Programme
CHALMERS UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Division of Computer Engineering
Göteborg, Sweden 2007

Abstract

Particle systems are an important technique for rendering a certain class of “fuzzy” object in 3D graphics in general, and games in particular.

This thesis presents a particle system simulated and rendered entirely on the GPU found in Microsoft’s Xbox 360® console, including sorting on the GPU to support correct rendering of non-commutative rendering modes, and geometry amplification for rendering non-point primitives.

Acknowledgements

I would like to thank my supervisor Ulf Assarsson at Chalmers University of Technology.

I would also like to thank Rare Ltd. and Microsoft Game Studios for allowing me to work on my Master's Thesis in their care. In particular, I owe a debt of gratitude to Tom Grove and Cliff Ramshaw at Rare for their mentorship and invaluable technical expertise.

TABLE OF CONTENTS

1 Introduction

Particle systems have long been an important part of 3D graphics to visualize various types of effects, such as smoke, fire or dust. Games in particular have used particles to simulate explosions, blood spatter and other effects(1).

In this thesis we explore the design space for particle systems on a modern gaming console. Though some of what we learn may be applicable in other areas, such as off line rendering for movie computer graphics, we will focus on achieving result which are practical for real-time usage in games.

1.1 Background

1.1.1 What is a particle system

A particle system is a simulation of small objects, particles, over time. These particles typically have a position and velocity, and are affected by forces such as gravity. Some implementations may include inter-particle effects and other more sophisticated features, but we shall focus on the more common case where particles are independent of each other, and simply simulate a small mass moving under the influence of the Newtonian laws of physics. These particles are emitted with some initial values, typically in great numbers, from an emitter, live for a period of time, and then die when certain conditions are met(1).

Traditionally particle systems have been simulated on the CPU each frame, and then uploaded to the GPU for rendering. In later years the graphics hardware has become more flexible, and it is now feasible to do certain types of general purpose computation on the GPU itself, without invoking the CPU at all (thereby saving CPU cycles and also avoiding issues with CPU/GPU synchronization). In this thesis we will explore using the GPU on the Xbox 360 gaming console to do our particle simulation and rendering.

1.1.1 A Brief Overview of the Xbox 360 Hardware

The Xbox 360 is powered by three-core PowerPC processor at 3.2 GHz with 512 MB of unified system/graphics memory and a custom ATI

graphics processor running at 500 MHz(2). The Xbox 360 graphics interface is a superset of Microsoft DirectX 9.

For the purposes of this thesis we will not use many of the features of the CPU, and will thus focus on the GPU.

The Xbox 360 GPU is unified architecture. It's capable of running 32 vertex and 64 pixel shader "threads" at the same time, each of which operating on a "vector" containing 64 elements (vertices, or pixels) that get processed at once. Each of these threads can make use of 48 ALU units, 16 vertex fetch units, and 16 texture fetch units. These resources are dynamically allocated to threads based on load(3).

An interesting consequence of this multithreaded shader architecture is that fetch latency can be "hidden" if there are enough ALU instructions in the shader. If a vector is waiting on a texture fetch, then other vectors can keep using the ALU in the meantime, avoiding a stall where the GPU is just sitting idle, typically resulting in effectively zero latency texture and vertex fetches.

Another unique feature of the Xbox 360 GPU is that it has 10 MB of very fast (256 GB/s) embedded DRAM where render targets are stored. This memory can not be used as the source of any fetch instructions. There are many reasons for why this is a good feature of the Xbox 360 GPU (e.g. very cheap anti aliasing) but since it is not possible to use the EDRAM as the source for any fetches, a copy-back to main memory is needed (called a "resolve", which occurs at a rate of 8 pixels per clock cycle) before it can be used in a shader. This extra cost is one of the reasons for why we do not use render-to-texture for our particle simulation (though the benefits of using another method, memory export, are the primary reason).

The Xbox 360 can fetch 32 bytes of data per fetch instruction, and uses an 8KB cache for vertex data, and 32KB of (16-way set associative) cache for texture data(3).

Perhaps the most revolutionary feature of the Xbox 360 GPU is the shader memory export. Using this it is possible to export data to system

memory from within any shader. This can be done in an arbitrary fashion (i.e. it's not restricted to outputting data in a stream, like with DirectX 10[1] (4), but supports full scattered memory writes) and can even be the sole purpose of a shader (a vertex shader does not need to have output passed on to the rasterizer – it could do its work entirely through memory export). We shall make heavy use of memory export, both for simulation (exploiting primarily the fact that we can use it in arbitrary shaders, and thus can run our simulation at the same time as the rendering) and for sorting (exploiting the scattering support to do proper compare-and-swap operations, and avoid the redundant copying inherent in the "ping-ponging" of render-to-texture based approaches).

[1] It should be noted, however, that both Nvidia and Ati have their own API for accessing their DirectX 10 GPU hardware in ways which do allow scattering, in CUDA(14) and CTM(15) respectively.

2 Problem Specification

The basic idea of implementing a particle system on a GPU is not new(5)(6), but the existing techniques have mostly been focused at consumer graphics hardware on the PC platform (some, like (5), even target vendor-specific extensions).

While there are many similarities, the differences between the PC platform and Xbox 360 are significant enough that any direct comparison between existing PC-based techniques and our technique for the Xbox 360, would be a bit like comparing the proverbial apples and oranges. So while the existing approaches *can* be implemented on the Xbox 360 GPU, we deliberately avoid setting up such straw man comparisons, and instead treat our technique as a completely separate approach.

In this thesis an attempt to find the "best practices" for implementing a GPU based particle system on the Xbox 360 is made. In doing so, existing techniques will be extended and adapted to make use of the many new hardware features available to maximize performance and flexibility. Existing approaches will be compared and contrasted qualitatively where appropriate, but there will not be any direct performance comparisons.

This thesis is an attempt to solve the following problems:

- ***Simulation*** The particles will be simulated over time wholly on the GPU, without any CPU intervention. The particles' positions will be updated based on velocity and acceleration. We will also incorporate effects such as basic collision detection, turbulence and drag.
- ***Sorting*** In order to render particle effects with non-commutative blending we will also require sorting, this will also be performed entirely on the GPU.
- ***Rendering*** We will explore various methods of rendering the particles on the Xbox 360 GPU.

3 Analysis and Method

This chapter summarises the implementation details and design choices in implementing the particle system described in this thesis on the Xbox 360 GPU. For all the shader code controlling the simulation of particles Microsoft's HLSL, High Level Shading Language (with snippets of inline microcode for the parts which aren't yet supported in HLSL) is used.

The particle system simulation is written as a vertex shader (with no output, a so called "multipass shader") using memory export for writing out the updated particle data in the same location. There is no need to use double buffering for this; writes can be made to the same buffer the simulation shader is reading from. Sorting is also done in an HLSL multipass vertex shader using memory export. Rendering uses both vertex and pixel shaders, both written in HLSL.

The following sections describe in detail the various things performed in these shaders.

3.1 Simulation

There are a few possible options for simulating a particle system. For example, some particle systems may be entirely stateless, i.e. they depend only on their initial values and the elapsed time, but most require more sophisticated integration of their properties. We will focus on *stateful* particle systems, since they are required for more advanced effects (such as turbulence, collision detection etc.). A stateful particle simulation works by numerically integrating the properties of each particle in small time steps (typically once per frame).

Since a particle represents an actual physical object, there are in principle only a finite set of properties that could be useful for simulation of Newtonian physics. For example:

- Position
- Velocity
- Mass
- Volume (or density)
- Inertia tensor

- Orientation
- Temperature

In practice however, many of these attributes are usually omitted because their impact is not significantly observable, or because they can be simulated with another attribute, *time*. So, for example, while a real “fire particle” may cool down below a certain temperature and "die" (i.e. become invisible) based on things such as velocity, air temperature etc., this can be simulated by just varying its intensity over time, and removing it after a certain amount of time has passed.

The most common attributes for particles in a particle system are(1)(5)(6):

- Position
- Velocity
- Time

There are some options on how to represent the time, however. One could store the time elapsed since the particle was spawned, or the time left until it is scheduled to "die". Both of these approaches have their merits. Using the time parameter to represent the time left until death means life times can be specified to vary randomly, rather than having a fixed life time for all particles in the system. However, this also means that certain time varying effects (e.g. having the colour depend on the time since birth, as in the fire particle example above) are more difficult to get right since one doesn’t know how long a given particle has been alive. A third option is to store two values for time, one storing how long the particle has lived, and the other storing its time of death. As we shall see later in section 3.1.3, the specific hardware constraints leave us with a “spare field”, which we can use to store a second time value.

Note that we have not yet specified *how* these values are stored, just what kind of properties we would like to be able to find about each of our particles. In practice we must find a specific way of storing this in a particle buffer. We explore the options for storing this in section 3.1.3.

3.1.1 The Simulation Step

The basic simulation step performed in the shader simply updates the physical properties of each particle based on the elapsed time, and takes the appropriate action for "dead" particles. Here follows a high level overview of the algorithm performed for each particle:

- Read a particle description (see 3.1.3)
- ***If*** the particle is dead:
    - Generate a new particle in its place (see 3.1.1.4)
- ***Else***
    - Integrate the particle position, taking gravity, drag, turbulence, and other physical effects into account (see 3.1.1.1, 3.1.1.3 and 3.1.1.3)
- Write the updated particle data to the output buffer

3.1.1.1 Integrating the Particles

At each simulation step we must perform an integration of various time varying properties for a particle. Our particle system is subject only to simple Newtonian physics, which can be approximated quite well with Euler integration.

We thus add up all the forces, convert to an acceleration based on the mass for a particle (which may either be constant, or varying per particle), and then integrate this acceleration for the time step in question.

3.1.1.2 Collision detection

It is also possible to perform some basic collision detection while simulating the particles. The implementation in this thesis used rudimentary collision detection against a plane as a proof of concept, but there's no reason why one couldn't perform collision detection and response against other types of colliders, as long as they can be conveniently stored as shader constants.

3.1.1.3 Turbulence Fields

While some particle systems look perfectly plausible even when simulated as if they exist in a vacuum, many do not. For more realistic animation of these systems turbulence fields can be introduced. A turbulence field can be defined as simply a function on particle position and

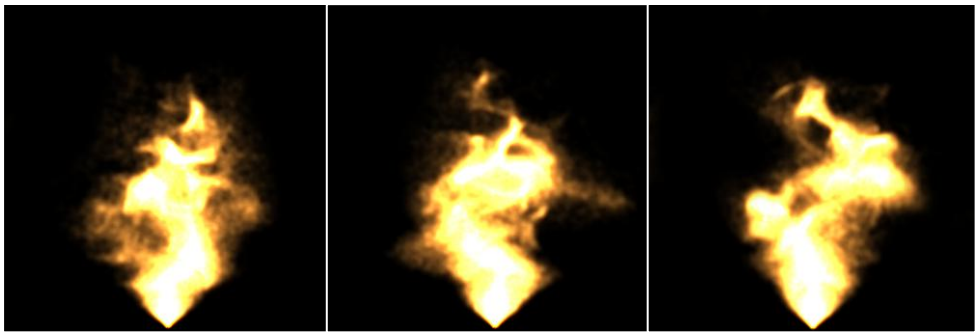

FIGURE 1 These screenshots show off a fire effect. It's achieved by spawning 128K particles in an roughly upwards direction, and subjecting the simulation to random turbulence, where the y-component of the turbulence vectors have been forced to always be positive, and the turbulence texture coordinates are animated with a sum of sine and cosine waves moving mostly up but with considerable horizontal movement, which produces the "swirly" effect of flames licking upwards. The turbulence texture used was a 16^3 32-bit volume texture. The frame rate is 334Hz (with 1.006 ms for simulation).

a field, yielding a force at each point. This can be used to simulate wind or the hot gasses swirling in a fire or explosion, for example. It should be noted that this is not the only possible way to describe turbulence; the turbulence field could contain velocities rather than forces, where particles passing through the field would simply adjust themselves to the surrounding turbulence velocity based on some drag coefficient. We opt to use a force field rather than a velocity field because this can be used in conjunction with "drag field"[2] (which simply introduces a force in a direction opposite, and a magnitude proportional, to the velocity of the particle in question) to produce the same effect, while still leaving the option of tweaking the two field parameters independently.

For some turbulence fields, it might be possible to compute the force at each point procedurally, but in general this is not feasible. We thus store the turbulence field as vectors in a 3D volume texture, and sample it for each particle using the position as the texture coordinate (scaled and offset to find the correct location within the field). Because 3D volume textures take up much space, and are essentially sampled at random with little spatial coherency it is important that the format we choose for this data is as small as possible so as to minimize cache misses. We found that a 32 bit value for

[2] A field of which simply "contains" air resistance, such that if the particles ventured outside of this field they would not be subject to drag. One could also view drag as a global property tied to the particle simulation itself, and not a field at all.

each vector gave sufficient resolution, using the DHenN3 format (3) which stores three components with 11 bits for two of them, and 10 bits for the third. However, even with this compact format large volume textures leads to very poor performance due to the highly pathological cache behaviour of this use case. Some of this can be alleviated by sorting the particle system based on spatial location as described in section 3.2.2, but in general we are limited to using fairly small turbulence textures. This is not necessarily a major problem, since turbulence is often just high frequency noise and can be tiled aggressively without major visible artefacts. The low resolution of our turbulence fields is further hidden a great deal by the approximate and ill-conditioned nature of our numerical integration – very small differences in initial direction and velocities give vastly different trajectories through the field. Furthermore, the texture coordinates of the turbulence field can be animated, either using a per-frame random offset, or using some smooth time-varying offset to achieve the appearance of motion (e.g. rising hot gasses in a fire animation), which also helps hide any remaining periodic artefacts.

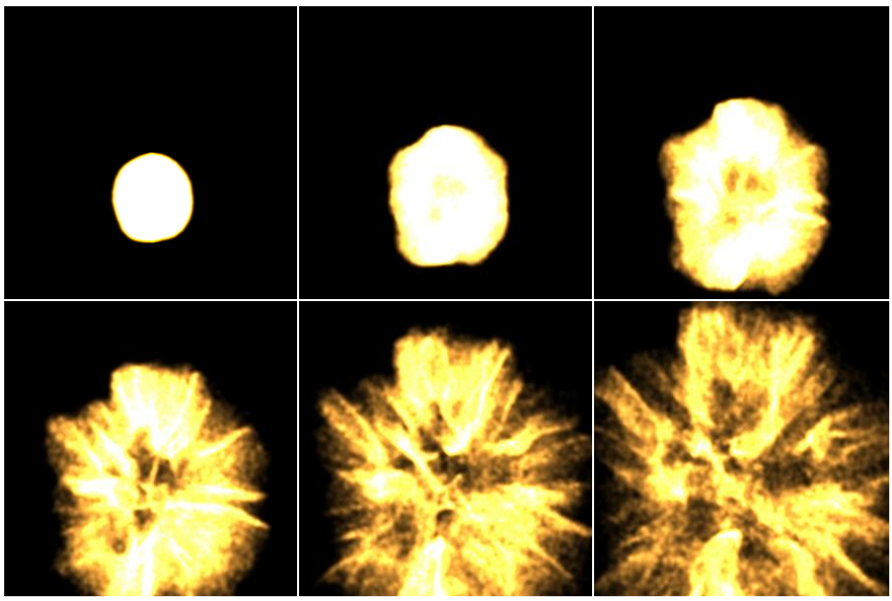

FIGURE 2 This series of screenshots (left to right, top to bottom) shows an explosion effect achieved by spawning 128K particles in random directions, with drag and turbulence. The turbulence texture used was a 16^3 32-bit velocity field with random vectors. The rendering time was 54 Hz (with 1.006ms for simulation). Note that while the screenshots are small, they covered the whole screen when rendering, which is the reason for the high render times.

Since the Xbox 360 has 32KB of texture cache, we need to keep the turbulence texture size at around 20^3 texels or less with a 32 bit texture format, to avoid poor cache performance.

Turbulence allows us to simulate a wide range of particle effects with a more rich behaviour than simply following a simple parabolic arc of motion. Just using a turbulence texture with random vectors can provide very good looking results for things such as fire and smoke, but of course the turbulence texture could be dynamically updated based on in-game conditions (e.g. dust swirling around beneath a helicopter), or based on more sophisticated simulations. Figure 1 and Figure 2 display a fire and an explosion effect respectively, achieved through using turbulence textures.

3.1.1.4 Emission

Many particle system effects need the appearance of a constant flow of particle (e.g. rain, fire, and steam). This requires that dead particles get reinitialised and emitted anew, somehow.

One way of solving the emission problem is to do it on the CPU(6). Using this method on the Xbox 360 is not an unreasonable proposition – due

to the unified memory architecture of the Xbox 360 there would be no need to copy a bunch of data across a slow bus to do this. However, touching the particle data with the CPU would likely require synchronisation with the GPU, or double buffering. Furthermore, we process each particle with the GPU anyway so it would be a shame to have to loop through the particles on the CPU as well to reinitialise dead particles, duplicating much of the work. We will therefore use the GPU to emit particles.

Alas, there is no access to a random generator on the GPU, nor is there any global state that could be used to implement one. One approach to get around this problem of emitting particles on the GPU is to use a lookup texture with CPU-generated random values. This texture would only need to be filled once or perhaps at a set time interval. The shader can then sample the random texture to determine things such as the new position, velocity and other random properties. In order for this to be useful, however, this texture needs to be sampled in different location for each particle. Some other GPU based particle systems do this by generating a texture coordinate for the random texture using the position and other properties of the dying particle(7). This has the notable drawback that if your particle system for some reason tends to cause particles to die with similar properties (e.g. if the system supports collision detection and the particles all end up in a concavity right on top of each other with zero velocity), this texture coordinate may only vary within a very small part of the random texture. This could produce "clumps" of particles due to the newly emitted particles sharing the same attributes.

On the Xbox 360 hardware there are no vertex input registers(3), instead each shader has full control and responsibility for fetching vertex input using an input index. This index can be used together with a random texture to find random values for each particle. To improve the randomness even more, a CPU-generated random integer offset can be supplied each frame, which is then added on to the index (this ensures that each particle will most likely sample in different locations on each emission). Care must be taken that the size of the random texture is large enough so that there are always more random values than the average number of particles emitted each frame (to avoid duplicate emissions).

The random texture itself can use any number of formats. We found that since for most systems only a comparatively small number of particles get emitted each frame the texture format has very little performance impact. Our implementation uses a 64 bit texture format, with four 16 bit fixed point values in the range [0,1) for each element. Of course, for systems where particles have very short life spans the emission rate may be comparatively higher and in that case it might be worthwhile to use e.g. a 32 bit format instead.

Because all dead particles a given frame will be emitted at the same time, a subtle banding artefact can sometimes be visible in particle systems with continuous flow (this may be alleviated after a few frames if the speed of the particles is randomized). One simple way to avoid this to some extent is to simply simulate the time "lost" due to the discrete nature of the time step simulation. So if a particle has, say, 10ms of life left, and the simulation is integrating a time step of 16ms, the particle will have actually been dead for 6ms by the time it gets killed and reemitted. It's important to simulate this time rather than just resetting the particle position (which will cause the particle to move away a slight distance from its initial position along its initial velocity direction, reducing the banding artefacts since this distance varies from particle to particle).

3.1.2 Simulating Particles Using Memory Export

To store the updated particles we use the memory export feature of the Xbox 360 GPU. This allows us to write to arbitrary locations within a buffer, with a few restrictions:

- The data buffer that we write to must be a homogenous stream of simple elements; each element can be almost any GPU format at least 32 bits in size (1, 2 or 4 floats, 2 or 4 halfs, 32 bit fixed point values, 32 bit RGBA values, 10:10:11 formats etc.). Unlike vertex streams the memory export stream cannot contain any structured elements.
- The memory export writes do not happen right away, so there is no guarantee that a subsequent read won't access stale data. It's thus important that we don't write to elements which we intend to later

read from in the same pass (the writes are flushed at the end of the pass, so reading from them in a subsequent pass is okay).

Despite these two restrictions, using memory export for doing general purpose computations on the GPU is fairly straightforward. We read a vertex element from a vertex stream, and then write it out to the exact same memory location when we've updated it, when the pass is finished the writes get flushed and we immediately have the output ready for consumption by the next pass. Note that since we export to the input stream the format restrictions for memory export also applies to the vertex stream (since they are one and the same).

Compare this with previous approaches where one would use a texture fetch in the vertex shader to dynamically override the position of a vertex in order to render it (6). Memory export allows us to achieve the same result as (5) (which uses ATI-specific OpenGL extensions called "Super Buffers" to reinterpret the data from a render target as a vertex buffer), in that we can use the results of the simulation directly as geometry data. As we shall see in section 3.2, memory export is more flexible since it allows us to easily perform *scattering* operations, unlike approaches based on rendering to a texture (or stream, as in DirectX 10(4)), and then reinterpreting the results[3].

Another option available to us due to memory export (but not when using previous methods) is to perform the particle simulation *in the same pass* as the rendering. Memory export can be used in arbitrary vertex shaders, so there is in principle nothing stopping us from doing our particle simulation in the same pass as the rendering, thereby eliminating an extra pass over the data. As we shall see in section 4.2 this is indeed substantially faster than doing simulation in a separate pass. In real-world scenarios one must be careful in utilizing this approach, however, since we often need to render particles several times (e.g. to support multiple viewports of the same scene

[3] Scattering could, in theory, be simulated by simply rendering a bunch of 1x1 texel quads to a texture, and using the vertex shader to offset the output position. This has at least one notable drawback: The Xbox 360 rasterizer works on 2x2 pixel quads(2), so rendering a single pixel at a time leaves ¾ of the rasterizer unused.

or when using a technique known as tiling[4]), but we do not want to simulate the particles more than once per frame. Furthermore some particle systems perform geometry amplification (see section 3.3.2) in the render pass to produce quads or other shapes, which is a scenario that does not lend itself well to simultaneous simulation since each particle will be processed multiple times[5] (once for each vertex in the shape, i.e. four for a quad).

3.1.3 Particle Representation

We already listed desirable properties for our particle representation in section 3.1 but how do we actually store this in memory? Clearly we would wish to keep the data format as small as possible to improve cache performance for our simulation, and we would definitely like to keep the size fewer than 32 bytes since that is the maximum size of a single fetch instruction (see 1.1.1). Let's review the components of our particle representation again, and reason about possible data formats.

- ***Position*** This needs to contain three values describing the particle's location in 3D space. We would almost certainly require these components to be of a floating point nature, since we cannot know beforehand the maximum extents of the particle simulation (which would be required in order to store the positions using fixed point data formats).
- ***Velocity*** This needs three components to describe the direction and magnitude of the velocity for each particle. In principle this could often be a fixed point format, since factors such as drag impose a "terminal velocity" in the medium which can be used to derive a fixed point representation, though we would prefer to keep this in a floating point format as well, for simplicity.
- ***Time*** This needs to contain either the time elapsed since birth or the time remaining until death, or both. This could be a fixed point

[4] Tiling is a clever technique to efficiently render high resolutions images despite only having 10 MB of EDRAM, whereby the “command buffer” is stored and “replayed” multiple times for different “tiles” of the screen (each occupying less than 10MB of memory), then once all the tiles have been rendered and resolved, stitched back together in the frame buffer.

[5] The memory export registers count as output registers, and cannot be skipped using predication(3). So there would be a cost for each vertex in the amplified geometry and not just the last one where the actual simulation happens.

format set at a given resolution (e.g. one tenth of a second -- remember that this only governs time-varying effects and the time of death, and not the simulation itself, so resolution need not be high).

The data format for the position, in particular, is a tough choice. A float3 (clocking in at 12 bytes) would be the initial choice since it gives high enough precision to store our particles in essentially any space we choose. The other option available if we limit ourselves to floating point formats is a half3 (three 16-bit floating point formats – 6 bytes). This gives significantly reduced range and precision, but only costs us half the amount of space. We tried both solutions and found that if we stored all positions in relation to a reference point (supplied as a per-frame float3 shader constant) the range and precision of the 16 bit floating point format was quite adequate for most cases (the average performance penalty of using 32 bit floats instead was roughly 35%). For moving particle systems, two reference points can be used; the reference point for the previous frame which is used to "decompress" the position data in the particle stream from half3 to float3, and the new reference point for the current frame which is used for "compressing" the position data into our particle representation again.

There is a limitation on the memory export support which affects our options on how to lay out our data; each memory export operation can only work on a homogenous stream of data. This need not be a major problem since we also have the ability to perform up to four memory export operations simultaneously – we could thus store for example the position data in one stream, the velocity data in another, and the time data in a third, each having their own optimal data format. Unfortunately there is also a substantial overhead in using multiple streams of data with the Xbox 360 GPU, since we need to perform a separate fetch for each.

Several different data configurations were tried, including ones with fixed point formats for the velocity, but in no case was the gain from reduced memory footprint (and thus improved cache performance) able to counteract the added cost of using multiple streams in the simulation step. We decided on using two half4s for the particle representation, giving us a total size of 16 bytes per particle. That gives us six 16-bit floating point values for the position and velocity, and two additional values for the time (time elapsed, and time remaining). The sign bits for the time values can be

used as "flags" to toggle certain behaviour. We will use one of these to indicate that a particle is to be "skipped", as detailed in section 3.2.5.

If the particle format exceeds 32 bytes (and thus cannot be fetched with a single fetch instruction), for example if very complex per-particle behaviour is needed, then it might be worthwhile to split the data up into two separate streams to reduce the particle size (and thus cache performance). In practice, however, there is probably no reason to use more than 32 bytes per particle for games, and the cost of the extra fetches may be far more significant than the improved cache coherency attained by using smaller particle formats.

3.2 Sorting

While many particle effects can be rendered in any order (e.g. systems using additive and multiplicative blending), there are some situations where an ordering needs to be imposed on the system. For these particle systems sorting is required.

There are several reasons for sorting particle data, we cover two of them in this thesis, discussed next.

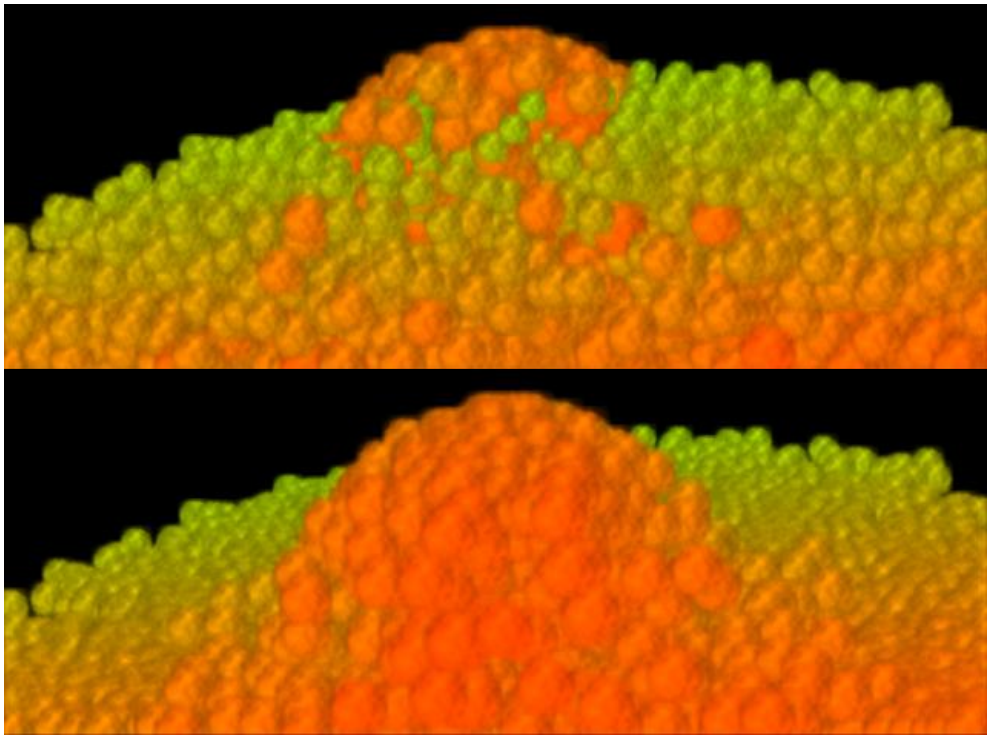

FIGURE 3 These two screenshots demonstrate the impact of sorting particles for a system with a fountain emitter with 8192 particles and a floor plane. The particles are tinted based on distance to better show off the artefacts due to incorrect rendering order (green far away, red nearer to the camera). The top screenshots shows a rendering with non-sorted particles, and the bottom screenshot shows them sorted by depth. Pay particular attention to the particles on the floor plane, and notice how they look like a flat surface at the bottom (as they should) but that there is no clue as to the shape of the particles in the top screenshot. The frame was 121Hz and 119Hz respectively (0.016ms for simulation and 0.11 ms for 5 sorting passes per frame).

3.2.1 Sorting for Visual Correctness

The most common reason for wanting to sort particles is for visual correctness. When using a non-commutative blending mode (such as alpha blending(8)) sorting is needed to ensure the correct order of operations. For example, if using a particle system to render dust it wouldn't make sense to use an additive or multiplicative blending mode since the particles essentially simulate occlusion. For such a system alpha blending can be used instead. Since this blend operation is not commutative care must be taken that the blending happens in the correct back to front order.

In practice, however, for large number of particles the visual impact of a small number of the particles being blended in the wrong order is neglible.

Thus for visual correctness we require only that the particles are processed in *roughly* back to front order[6].

The ideal sorting algorithm for visual correctness is thus one which we can spread out over multiple frames, improving the "sortedness" of the particles incrementally each frame, without having to pay the cost of a full sort. Using such an algorithm the amount of sorting that gets performed each frame could be tweaked on a case-by-case basis to achieve good quality while minimising the time spent sorting.

Figure 3 demonstrates the difference between rendering alpha blended particles in a non-sorted and sorted order. It is very difficult to demonstrate the effect of rendering alpha blended particles in the wrong order in a static screenshot, but it is extremely noticeable in motion as the particle system looses all sense of shape.

3.2.2 Sorting for Cache Coherency

When sampling a texture in the simulation of a particle it is sometimes useful to sort the particles such that the spatial coherency of these samples gets improved. One example is turbulence discussed in section 3.1.1.3. Because simulation with turbulence samples a 3D texture based on position, over time the randomness of the particle system will lead to more or less random sampling of the turbulence texture. This type of sampling behaviour is disastrous for cache coherency, and can cause crippling overall performance (see section 4.4.1)

By partitioning the particles into a 3D grid, and then sorting them by the index of the grid cell in which they lie, we can ensure that particles in the same grid cell will be processed roughly at the same time. Since the positions of the particles with a grid cell are close together, their sample locations in the 3D volume texture will also be close together, and so cache coherency is improved.

There are various aspects that need to be considered when choosing the dimensions of this 3D grid structure, as well as the order in which the grid

[6] This is of course highly subjective. The “sortedness” required for “good enough” results varies between different particle systems, and should be adjustable by an artist on a per system basis.

cells are indexed. Ideally we would want to "traverse" the grid cells in a pattern which gives good spatial locality (such as a 3D Hilbert curve (9)). This requires us to find an "index" for a grid cell given its spatial location, which requires finding the inverse of a space filling curve. Thus we need to find a space filling curve with a cheap inverse. We choose to simply traverse the grid cells along the x, y, and z axes, in that order. The inverse of this space filling curve is then simply:

$$index = x + y \times width + z \times width \times height$$

This means that we process the grid structure in "depth slice" order, and each such slice is processed in row-column order.

The Xbox 360 GPU is optimized for reading 2D textures[7], which means it's preferable to ensure that the samples in a particular cell read from the same 2D slices of the volume texture (and the number of slices touched by the particles in a cell, are as few as possible). So the grid should be arranged so that each slice of it covers as few slices of the volume texture as possible.

For this reason we should choose the “depth” dimension of the grid cells to be one texel deep for point sampled volume textures. This ensures that each particle within a single grid cell samples from the same slice in the volume texture. This keeps the number of slices "active" in the cache at a minimum since the grid cells are processed slice order (and the grids cells themselves are only one texel “deep” and thus the particles in them all sample from the same slice).

For trilinear sampled volume textures the optimal depth is one half texel for the same reason, since all particles within a grid slice of this depth will sample the same *two* volume texture slices. If a grid depth dimension of one texel were used, samples in a cell could potentially touch *three* slices (the slice corresponding directly to the grid slice, and the two neighbouring slices) due to the trilinear filtering.

It may seem that the optimal grid cell size would thus be one or one half texel cubed (which would ensure that the particles within a cell sample the

[7] At least that’s the conclusion that can be drawn from experiments; there doesn’t seem to be any good documentation on how the Xbox 360 cache works exactly.

exact same texels, and no more), but that's not the case. Consider the traversal of all the grid cells within a single grid slice. The very first grid cell to be processed will be the one located at (0, 0), then the rest of the row will be traversed, and only then will (0, 1) be processed. So while the grid cells (0, 0) and (0, 1) are *neighbours* and likely to produce good cache behaviour if processed in direct sequence (since particles which sample along the edge of the two cells will need largely the same texels for filtering), they are processed a number of cells apart. For large enough volume textures this may mean that the texels sampled in grid cell (0, 0) have long since vacated the cache by the time its neighbour in the following row is processed.

Traversing the slices along grid cells thus introduce a loss of locality because the traversal is not optimal with regards to locality. In other words, if we choose too small grid cells, locality will suffer[8] but if we choose to large grid cells the number of texels within each may be so high that the samples within it do not fit in the cache. So the dimensions of the grid cells in each slice should be chosen to be to be as large as possible while still fitting comfortably in the cache, while the depth should be one or one half texel depending on filtering mode as discussed above.

So for example, if the texture format is 4 bytes per texels (which is the format used for turbulence in section 3.1.1.3), that means 8192 texels can fit in the cache (since the Xbox 360 texture cache holds 32KB). If trilinear filtering is used, texels from two depth slices will need to be kept in the cache, meaning that the grid cells can only contain 4096 texels per slice each, or 64x64. So in other words, the texel dimensions for the grid cells should be (64, 64, 0.5), in this example. Note that this reasoning doesn't take into account that we also need to sample a texture for the random values used in particle emission (section 3.1.1.4), so it would be a good idea to view these dimensions as an upper bound. Also, note that this reasoning only takes the cache *size* into account and assumes a fully associative cache, which is not the case for the Xbox 360 (which has a 16-way set associative cache), but provides a good starting point.

[8] Not to mention the fact that we only have a limited number of bits (23 mantissa bits for IEEE 754 floating points numbers) for the grid cell indices, so an excessive number of grid cells is simply not expressible

3.2.3 Sorting Networks

A GPU can essentially be viewed as a very wide SIMD processor; as such any algorithm we wish to implement on it will need to be stated in a highly parallel form. Most common sorting algorithms, such as QuickSort or MergeSort, are very efficient when executed sequentially but map poorly to parallel formulations. Since the GPU is highly parallel we can't use algorithms in which "early" decisions influence "later" decisions (e.g. QuickSort where we need to process the entire sequence in each step to determine which elements are to be processed in the next). Thus we are looking for an algorithm which is data independent. Sorting networks fit the bill perfectly(10).

Sorting networks are networks of compare and swap (CAS) operations that, when traversed, results in a sorted sequence. The CAS operation implements the following procedure:

```
CAS( a, b )
{
   if ( a > b )
      return ( b, a )
   else
      return ( a, b )
}
```

In other words, CAS compares two elements and swaps their locations to ensure a certain ordering. Sorting networks were originally used for hardware circuits. We can think of them as a sequence of “levels”, where each level, triggered by a clock cycle, performs a number of CAS operations on the input from the previous level, and which eventually leads to a sorted sequence as the final result.

The key feature of sorting networks is that their topologies are independent of the actual data. The same number of CAS operations will be performed on elements in the same locations, regardless of the contents of these elements. This means that each level in such a network executes a

number of entirely independent CAS operations. This behaviour maps nicely to a GPU, we simply treat each level as a separate pass over the stream data, where the indices passed to the GPU each represents single CAS operations.

Sorting on the GPU using sorting networks has been suggested before (11)(6)(5), but due to a lack of scattered writes most of these algorithms have needed to resort to unintuitive (and expensive) workarounds. For example, to perform a CAS operation using previous techniques, the scattering can be converted to gathering by essentially performing the CAS operation twice, once for each output value. In other words, to compare elements *a* and *b*, one would write out the first element of *CAS(a,b)* as the output for *a*, and then for *b* perform the CAS operation again, only this time writing out the second element of the result. The redundancy evident in this strategy can be avoided by using memory export on the Xbox 360 GPU to perform a proper CAS. This leads to less memory traffic since each element is only fetched once rather than twice, and elements which aren't relevant for a given pass will simply not be processed at all (previous methods required that unused elements were copied through untouched, using predication to detect such cases). With memory export the algorithms can be written so that they only need to perform the exact number of operations required for each pass, each element in those operations need only be fetched once, and all other elements can be left untouched.

There are primarily two sorting networks in common use today, Odd-Even Merge sort, and the Bitonic Merge Sort(10). It has been found that the Bitonic Merge Sort can be implemented in a more efficient way than the Odd-Even Merge Sort on GPUs (for example in(11)), but many of these speedup techniques simply worked around the limitations of earlier hardware, rather than being an inherent property of the algorithm itself (11). Disregarding these platform specific implementation tricks, the odd-even merge sort actually uses fewer comparisons than the bitonic sort(12). The perhaps largest remaining benefit of the bitonic merge sort is that it’s possible to merge the two final passes of each merge operation into a single pass. This leads to *log n* fewer passes for a complete sort compared to the odd-even merge sort.

The main benefit of the odd-even merge sort is that it, unlike the bitonic merge sort, has the attractive property that it always leaves the stream more sorted than it was after each pass(6)(10)(11). This is a key property since we wish to sort our particles incrementally, using just a few passes per frame. Thus, the bitonic merge sort is necessarily ruled out, and the odd-even merge sort was chosen for the implementation in this thesis.

3.2.3.1 Odd-Even Merge Sort

The odd even merge sort, like any merge sort, is a divide-and-conquer algorithm which works by recursively splitting a sequence up in two equal-sized parts, sorting each independently and then merging the results back together. The key operation that separates various merge sorts is the merging step. While the original merge sort simply merges two streams together in a highly sequential and data-dependent fashion after each sub stream had been sorted, the odd even merge sort uses another merge algorithm.

The odd-even merge is *itself* a divide and conquer algorithm. It takes a stream, splits it up into two halves and merges them recursively. It can be a little mind bending to visualize how this “nested recursion” traverses the data stream, but it is key for understanding the algorithm.

```
merge (s)
{
      n = size(s)
      if ( n == 2 )
      {
         return CAS(s(0),s(1))
      }
      else
      {
         odds = s(1,3,..,n-1)
         evens = s(0,2,..,n-2)
         result = interleave(merge(evens),merge(odds))
         for ( i = (1,3,..,n-2) )
         {
            (x,y) = CAS(result(i),result(i+1))
            result(i)= x
            result(i+1)= y
         }
      }
}
```

In other words, for a two element stream *merge* simply compares them and return them in order. For any other length stream *merge* separates out the odd elements and the even elements into two streams (hence the name of the algorithm), and merges them independently in a recursive step, then it interleaves the results, and performs the CAS operation for every consecutive pair of elements starting with the second and third.

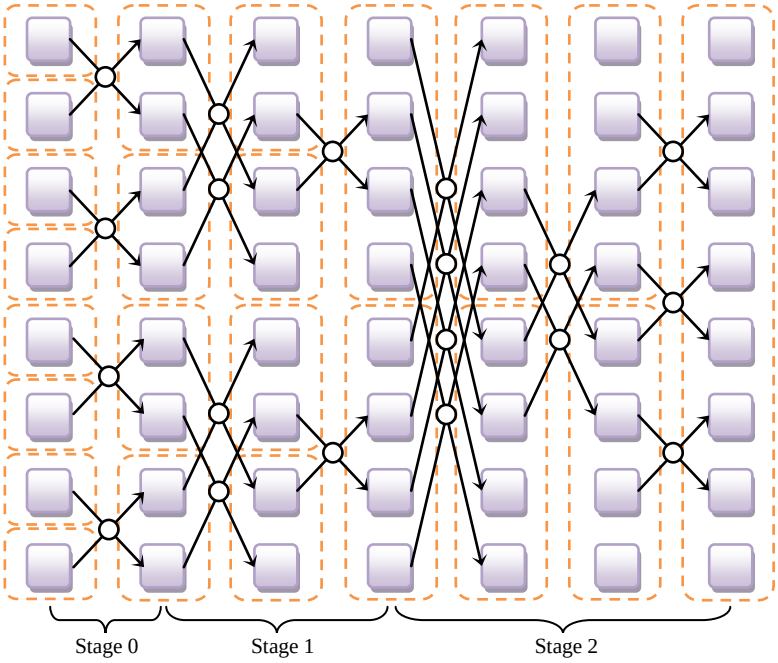

FIGURE 4 This diagram displays the sorting network for odd-even merge sort. Each circle represents a CAS operation. Each column of CAS operations represents a single stream operation (or “pass”). The “stages”, corresponding to “merge” operations, are indicated at the bottom. The dashed outlines indicate the subdivision of the stream caused by the recursive nature of the merge sort algorithm.

It may not be immediately clear that this algorithm is parallel in nature, but when drawing this up as a sorting network marking all the CAS operations, it can be easily seen that each is completely independent of the others within each "level" of the recursion “tree”. Figure 4 shows this sorting network, with each merge operation and each level within these marked. Again, this can be confusing, but it illustrates the workings of the odd-even merge sort graphically, so it’s important to understand it. Note the orange dashed outlines which correspond to the recursive subdivision of the list in the merge sort algorithm (the “divide step”). Also note that the various stages (each corresponding to a single *merge* operation, the “conquer” step) take an increasing number of passes to complete (unlike the standard merge sort algorithm where the merge step is equally expensive each time) because they are themselves recursive divide-and-conquer processes.

Since the odd-even merge sort is a divide and conquer algorithm, and the "conquer" part of the algorithm is itself a divide and conquer algorithm, we end up with a time complexity of $O(n \log^2 n)$. For a more formal time complexity analysis, consult (10), (11) or (12). This is worse than the theoretically optimal $O(n \log n)$, but in return the sorting can be performed in a highly parallel way entirely on the GPU.

While a recursive definition is elegant and lends itself nicely for proving that the definition leads to a correctly sorted stream (12), it is not terribly useful as is for the purposes of incrementally sorting a particle stream. The algorithm must be restated in a parallel form, such that the fundamental unit of computation is a "stream operation". This is fairly straight forward, since the stream operation in question would simply be the loop inside the merge function performing the CAS operations. The code must also be transformed into an imperative "loop" form which allows us to execute one, or a variable number of stream operations at a time. This is a bit tricky to get right, so the full C++ code demonstrating how to invoke the two stream operations *SortMerge* and *SortBaseCase* (which will be discussed shortly) one pass at a time are presented next.

```
bool SortStep()
{
   static int loopLen = 0, loopStep = 0;
   static bool loopRestart = true, loopIsInInner = false;

   if ( loopRestart )
   {
      loopLen = 2;
      loopIsInInner = false;
      loopRestart = false;
   }

   if ( !loopIsInInner )
   {
      SortBaseCase( loopLen );
      loopIsInInner = true;
      loopStep = loopLen / 4;
   }
   else
   {
      if ( loopLen > 2 )
      {
         SortMerge( loopLen, loopStep );
      }

      loopStep /= 2;

      if ( loopStep == 0 )
      {
         loopIsInInner = false;
         loopLen *= 2;
         if ( loopLen > NUM_PARTICLES)
         {
            loopRestart = true;
         }
      }
   }

   return loopRestart;
}
```

This function can be called over and over and will result in a single stream operation being performed each time. When the final stream operation in a sort has been performed, *SortStep* will return true, if there are more passes required to sort the sequence, it returns false.

Implementing the stream operation itself in HLSL is fairly straightforward. Of course the main operation is the CAS, which simply reads in two elements, compares them, and writes them out again in possibly swapped locations.

There is one subtle caveat. Since the Xbox 360 does not support integer instructions directly, one must be careful when doing any computations on

indices, since they will be executed using floating point arithmetic on the hardware. This leads to an interesting problem, namely that:

$$\frac{3}{3} \neq 1$$

We can see why when we consider that an operation like $^3/_3$ gets computed as $3 * {}^1/_3$, which, since ${}^1/_3 = 0.33333\ldots.$, will result in a floating point number close to, but less than, 1.0 (which, when working with integers, gets truncated to 0, rather than rounded to 1). To avoid this problem we need to implement a custom function for integer division and remainder operations, where we take some extra care to ensure the correctness of the result. Here's the HLSL code demonstrating this:

```
int2 DivRem( int x, int y )
{
   int d = x / y;          // division
   int r = x - d * y;      // remainder

   // handle off-by-one errors
   if ( r < 0 )
   {
      d -= 1;
      r += y;
   }
   else if ( r >= y )
   {
      d += 1;
      r -= y;
   }
   return int2(d,r);
}
```

This function first attempts to perform integer division, this may or may not produce an exact result depending on the values in *x* and *y*. Next the remainder is computed "manually" by simply multiplying the result of the division (which may be off by at most one due to the floating point precision issues) by the denominator and subtracting this number from the

numerator. If this remainder is negative then the result of the division is too large and needs to be corrected. If the remainder is larger or equal to the denominator then the result of the division is too small and the results need to be corrected in the other direction. If the remainder is positive, but less than the denominator, the results of the division were correct and no action is required.

Now we can take a look at the two stream operations, *SortMerge* and *SortBaseCase*, whose C++ implementations simply invoke their HLSL counterparts on the particle stream, transferring the *loopStep* and *loopLen* variables to shader constants, as well as another variable called *numIxPerList* which simply contains the total number of CAS operations required for each "chunk" (marked by orange dashed outlines in Figure 4) and can be used to figure out which chunk each CAS operation belongs to (remember each index passed to the shader corresponds to a *CAS operation*):

```
void SortMerge( int index : INDEX )
{
   int chunk = DivRem( index, numIxPerList ).x;
   int2 swapChunkRem = DivRem( chunk, loopStep );
   int currentIx = chunk*loopLen + (swapChunkRem.x*2 + 1)*loopStep
                 + swapChunkRem.y;
   int otherIx = currentIx + sort_step;
   CompareAndSwap( currentIx, otherIx );
}

void SortBaseCase( int index : INDEX )
{
   static const int listStep = loopLen / 2;
   int2 chunkRem = DivRem( index, listStep );
   int currentIx = chunkRem.x *2*listStep + chunkRem.y;
   int otherIx = currentIx + listStep;
   CompareAndSwap( currentIx, otherIx );
}
```

As we can see both these functions simply uses the index to identify which CAS operation it corresponds to (se Figure 4) in the relevant pass, then the two indices for this CAS are computed and passed to the CAS function which simply compares the sorting key and writes out the elements at swapped locations if (and only if) they are in the wrong order[9].

Looking at Figure 4, we see that the final pass of each "stage" consists of simply performing a CAS operation among adjacent elements, the second to last pass performs CAS between elements two indices apart and so on. Depending on the size of the elements it may thus be possible to fetch all of the data involved in such a CAS operation with a single fetch. For example, if the element data is just a single DWORD (and remember from section 1.1.1 we can fetch eight DWORDs at a time), the data for each CAS can be fetched in a single fetch operation for the three last passes in each merge stage (since they are 1, 2, and 4 elements apart, meaning that we can read the data needed if we fetch 2, 3, and 5 elements at a time respectively).

3.2.4 Sorting in Particle Systems

In order to make use of these sorting algorithms we must find some way of applying them to the particle data. There are essentially two alternatives, which we discuss next.

3.2.4.1 Sorting Using a Key-Index Stream

One alternative for sorting the particle stream is to simply produce a key-index pair for each particle(5)(6). The key contains the value on which the sorting acts (e.g. the distance to the viewer, or the index in a spatial grid structure), and the index simply points out the position of the particle in the particle stream. The upside to this approach is that we don't need to fetch quite as much data during the sort, which should lead to less bandwidth usage and better cache coherency.

The downside is that we must generate this Key-Index pair in each simulation pass, and we must also map the results of the sort back to the

[9] In practice, only the writes to the eM# registers get predicated, since the memory export address register (eA) counts as an output register and therefore cannot be conditionally written to using predication, but must always be written even if the elements do not get swapped(3).

particle stream somehow so that the particles can be processed in the sorted order.

The Xbox 360 GPU does not have true support for 32 bit integers, which means that the indices must either be limited to 65536 unique values (16 bits) or be represented using a 32 bit float. Since we aim to support far more particles than can be represented with 16 bits we opt for the 32 bit float approach. As mentioned in section 3.1.2, memory export works on homogenous data streams only, so the data type for our key-index streams must be float2, i.e. two 32 bit floats (one for the sorting key, and one for the index). This is half the size of the particle representation we chose in section 3.1.3.

To make use of the final sorted key-index stream one must somehow map this information back to the particle stream. Luckily, since we have full access to the vertex fetch functionality on the Xbox 360 (see 1.1.1) we can simply fetch a key-index pair, and then fetch a particle based on the index part of this pair. This can be done in the simulation pass, where we need to fetch the particles anyway.

To improve cache-coherency we choose to write out the result of the updated particle at the *sorted* index location, rather than the unsorted index location. This requires us to double-buffer the particle stream (since the current particle at the sorted index location may not have been processed yet and thus should not be overwritten), but leads to essentially unchanged cache coherency characteristics compared to just reading the particles in the order in which they appear in the particle stream. It should be stated explicitly that this last step would not be possible without the memory export functionality of the Xbox 360 GPU, and consequently you would get more or less random access patterns (and thus poor cache coherency) for the particle stream in previous implementations which use the key-index stream approach to sorting, such as(6).

Using a key-index stream to sort the particles introduces overhead in several places, because it requires:

- One additional fetch per particle in order to find out the location in the particle stream of the next particle.

- One additional write per particle to store the updated key-index pair for use in sorting in the next frame.
- Additional memory to store the key-index stream
- Additional memory to double-buffer the particle stream if we want to reorder it on the fly for improved cache coherency (and we do!).

Experimentation shows that the approximate performance overhead imposed on the simulation due to fetching via a key-index stream as opposed to just fetching the particle directly can be as high as a quite hefty 73%. The space overhead is 1.5 times the size of the particle stream, since an additional particle stream, and a key-index stream (half the size of the particle stream) are needed.

3.2.4.2 Sorting the Particle Stream Directly

Another option is to simply sort the particle stream directly, without going via a key-index stream. This method is not as feasible for particle system which use multiple textures for their particle representation (such as (5) or(6)), since it would greatly increase the number of fetches required in the sorting step (each component needs to be fetched and "moved"), while using our approach with memory export the number of fetches remains unchanged (we can retrieve an entire particle in with a single fetch instruction).

Using this approach has the benefit that all the overhead on the simulation pass can be avoided entirely, but the downside is that the sorting itself must read and write twice the amount of data, which leads to worse cache performance, and that we need to compute the sorting metric (distance, or grid cell index) for every sorting pass rather than just doing it once per frame. Another downside is that the optimisations mentioned in section 3.1.2 where we fetch multiple elements at a time to retrieve the input data for a compare and swap with a single fetch, wouldn't yield as much benefit since larger element sizes translates to fewer elements per fetch. In our example we can only do this optimization for the very last pass in each merging stage (since we can read exactly two particle elements per vertex fetch). In practice, the cost of sorting the particles directly were minimal (less than 7%) compared to the performance hit incurred on the simulation by using a separate key/index stream, especially since the sorting is spread out over multiple frames (see section 4.4).

3.2.5 Improving Temporal Coherency

As we shall see in section 4.4, sorting is an expensive operation, and despite our best efforts to speed it up, the cost of doing a full sort each frame is much too expensive to be feasible. The only reason sorting is still a reasonable proposition in practice is that our data exhibits *temporal coherency*. This means that we can perform only a few passes of our sorting algorithm each frame and still end up with acceptable results. It follows that a good way of improving the overall performance of our sorting algorithm is to take steps to improve temporal coherency in the hopes that this allows us to do fewer passes per frame.

Of course, one way of improving the temporal coherency is to just slow down the particles in the system. And indeed if this is possible to do while still achieving the effect you want (for example by animating the texture coordinates to simulate speedy motion while allowing the particles to move slower), it is a good idea to do so, but in general this isn't always possible.

The worst-behaved particles in most particle systems are the ones which have just died and been respawned. Their previous location can be very far away from their new position, and thus their position within the sorted stream may as a result be very far off indeed. A simple way of improving cache coherency is to avoid these worst offenders. We can do this by introducing an artificial delay (perhaps half a second or so) right after a particle is reborn where it will have all of its new properties, but it won't be simulated nor rendered for a few frames. Doing this gives the sorting algorithm time to move the newly created particle around to a better location in the particle stream before the effects of its poor location make themselves known (either through poor cache coherency, or through jarring visual artefacts, or both).

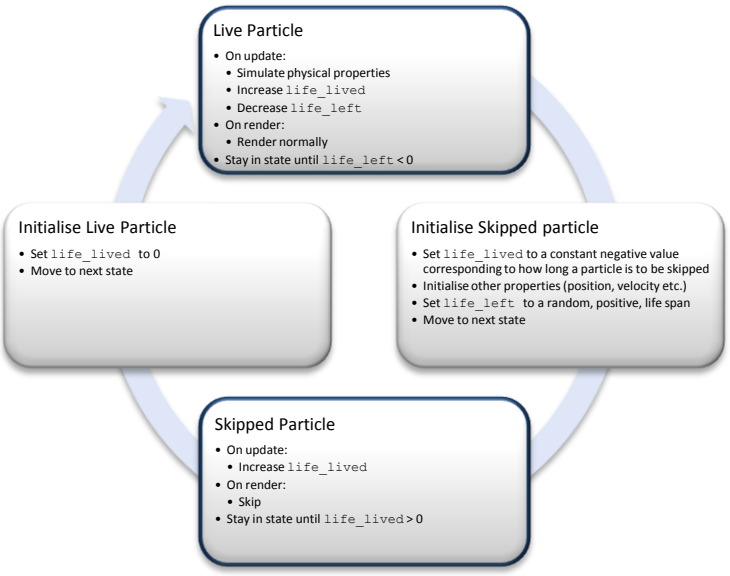

FIGURE 5 This state machine shows off the various states a particle can be in when we're using a "skipped" flag to improve temporal coherency. The two states without a border are instantaneous; they do some work and then immediately move to the next state. The two states with borders are the stable states for a particle.

We can implement such a delay by using the sign bit of one of the time values in our particle representation. We follow the convention that a negative time value represents a "skipped" particle. This particle won't be simulated, nor rendered. When a particle "dies" there are two cases to consider:

- If the particle is a skipped particle, then we generate a new particle which is not skipped. This can be done by simply resetting the time value and ensuring that the "skip" flag is not set.
- If the particle is not a skipped particle, we respawn the particle, reinitialising its parameters using random values, and set its "skip" flag to true. Figure 4 shows a state machine describing how particles are emitted, simulated, and skipped.

3.3 Rendering

Once we've performed simulation we must also render particles to the screen. Since we've performed all of our simulation on the GPU we need not do any expensive translations or copies here, but can draw our particle buffer directly. Either as point sprites (a native primitive on the Xbox 360 GPU), or using it as a basis for geometry amplification in the vertex shader.

3.3.1 Point Sprites

The Xbox 360 supports rendering of point sprites, so an obvious alternative is to simply render our particles as point sprites.

This is straightforward. We just draw the particle stream as "point" primitives, and in the vertex shader simply output the size and position for the particles in the vertex shader, and then sprite is automatically generated for us. As discussed in section 3.1.2, we have the option of doing the particle simulation at the same time as rendering; we can do this by just updating the particles in the vertex shader, writing it out using memory export, and then outputting the updated data to the pixel shader. Doing this, however, we must be careful not to simulate the particles multiple times if we're using tiling (see section 1.1.1).

In order to avoid "popping" when a particle centre goes outside of the viewing frustrum (but its resulting particle sprite would still be inside), care must be taken to adjust the guard band to force drawing of sprites even when their position is slightly outside of the screen extents.

In order to support "skipping" of particles (see section 3.2.5) the point primitive can be conditionally "killed" in the vertex shader. At the time of writing this operation is not supported in HLSL (not even as inline microcode), so the vertex shader needs to be implemented in microcode[10].

3.3.2 Geometry Amplification

While point sprites are very simple and efficient to use, sometimes they they aren't flexible enough for a certain effect. For example, one may want to generate a rectangle for each particle stretched out in the direction of motion, to give an added sense of motion. This isn't really feasible with point sprites since there is no control over the shape of the resulting sprite.

[10] I have, however, been told that this feature will be supported in HLSL in the June XDK and onward.

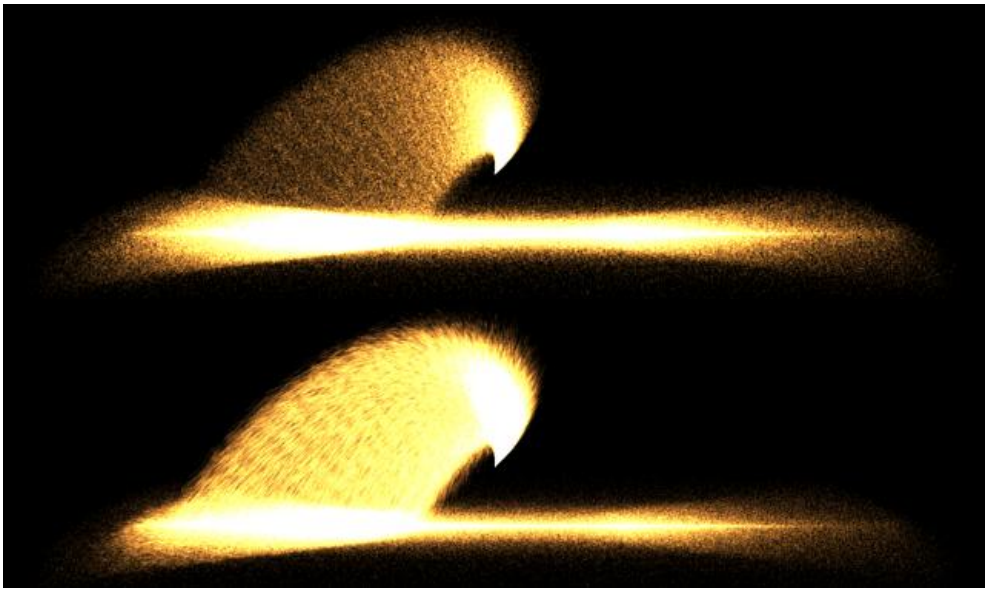

FIGURE 6 These two screenshots show the same particle system using point sprite rendering (top) and quad rendering with simulated motion blur by stretching the quads in the direction of motion (bottom). The data is the same for both systems and was captured from a simulation of a rotating cone emitter with two million particles and collision against a floor plane. Note the stretched and blurred "streaks" in the lower screenshot, compared to the simple points in the top screenshot. The frame rates for these two screenshots were 60.7 and 37.4 frames per second respectively, including simulation (taking 7.8 ms).

For these cases performing *geometry amplification* is required in order to go from a particle stream, to a stream of geometric shapes (such as quads). Luckily, the Xbox 360 GPU is extremely flexible when it comes to fetching vertex shader inputs, and gives us all the tools we need to perform arbitrary fetches, based on the index. So to generate, for example, a quad for each particle, you could simply draw a quad list, with an index buffer simply containing the indices 0,1,2…4*n*. In the vertex shader dividing the index by 4 yields the position in the particle stream for the particle corresponding to the vertex. The remainder of this division is the primitive-relative vertex index. Then it is simply a matter of fetching the correct particle from the particle stream, extracting the position, and then applying the appropriate offset to this position (based on primitive-relative vertex index, viewer direction, particle velocity etc.). Figure 6 shows an example of rendering quads with a motion blur effect achieved by stretching the particles in the direction of velocity.

Quad sprites can also be killed, to skip rendering of a particle, by just killing all of its vertices.

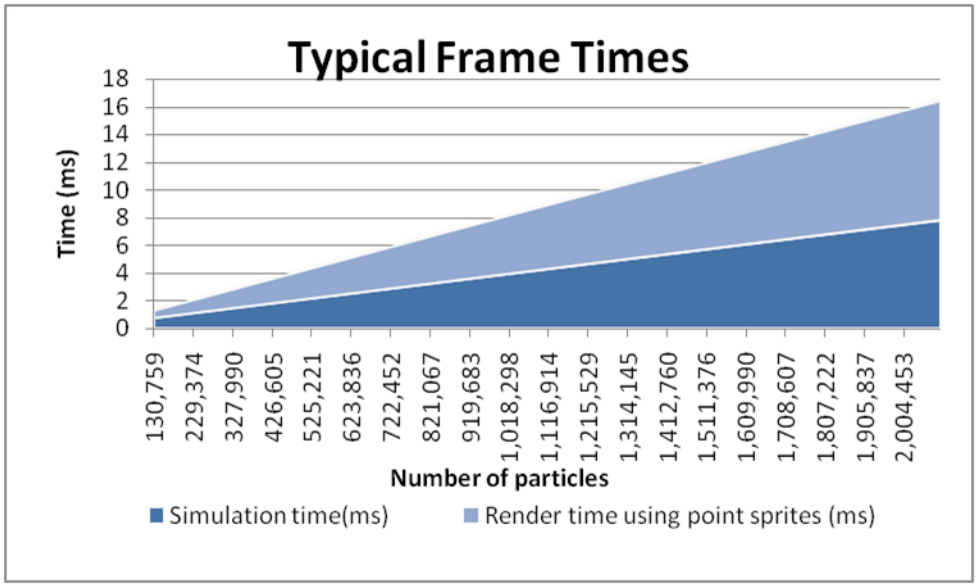

CHART 1 This chart demonstrates the overall frame time for a typical particle system with varying numbers of particles. The system demonstrated in this chart is a typical cone emitter, with a randomly varying life span between 0 and 3 seconds, 45° spread, and an exit speed of 2.5 meters per second, with gravity effects (9.83 m/s^2). The particles were simulated and rendered in separate passes. The rendering pass used point sprite rendering with additive blending.

4 Results

There are already several screenshots of the techniques described in this thesis distributed throughout the text, demonstrating the visual results. We now take a closer look at the performance.

4.1 Overall Performance

In Chart 1 the frame times for a typical particle system are presented. These results were captured by simulating and rendering the particle system separately, with a varying number of particles.

While the performance of the particle simulation varies in a fairly straightforward and unsurprising way when changing the number of particles, and other settings, it is not possible to present exact figures for all of these possibilities. Therefore, to give a well-rounded idea of the performance characteristics of the particle simulation and rendering under various circumstances, all screenshots of particle systems in this thesis also have frame timings presented in the caption (including simulation time). All screenshots were taken at the resolution of 1280x720, and where they have

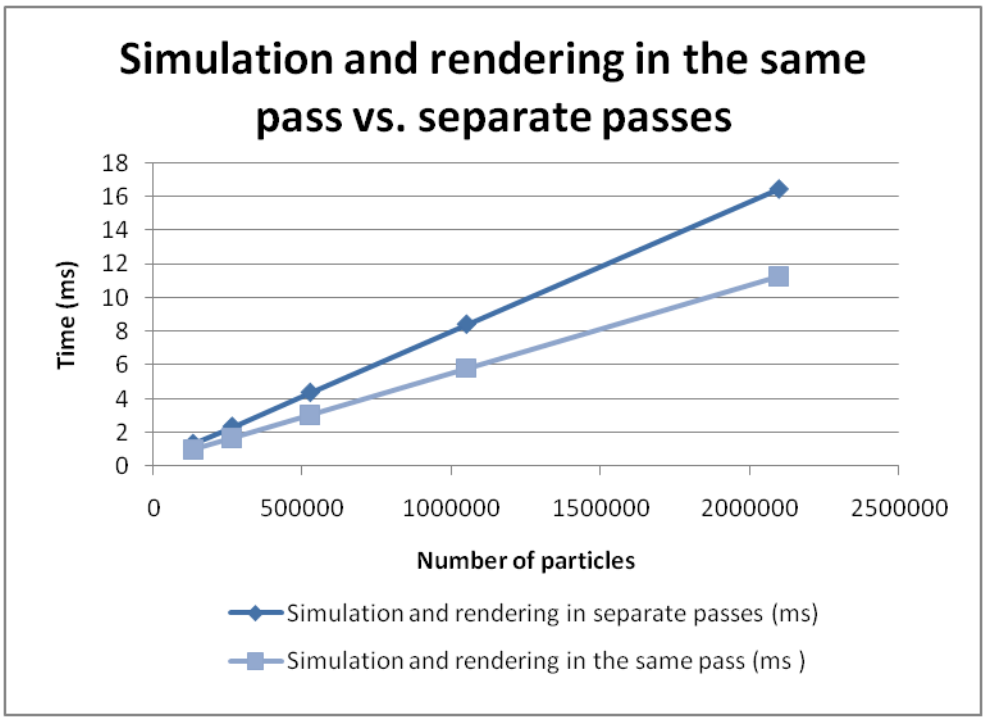

CHART 2 This chart demonstrates the performance difference between doing simulation and rendering in separate passes, versus doing them in the same pass. The particle system is the same as that in **CHART 1**.

been cropped they have been so only in one direction. It should be noted that rendering performance varies greatly with the size of the particles in screen space and their number, and in many of the screenshots the rendering takes far longer than the simulation for that particular viewpoint. One should thus be careful to deduce too much about the rendering times from these screenshots, and perhaps concentrate primarily on the simulation times (which do not vary with the viewpoint).

Chart 2 presents the performance characteristics of doing the simulation and rendering in separate passes, versus doing them both in the same pass. As we can see, there are substantial gains to be had by simulating in the

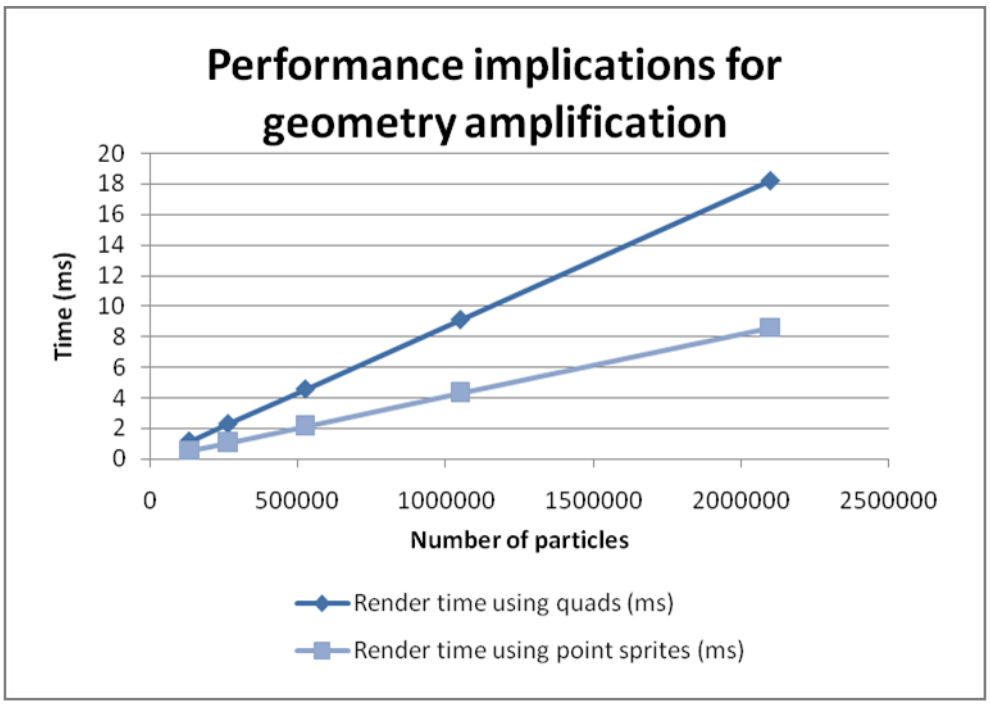

CHART 3 This chart demonstrates the performance implications of geometry amplifications. The settings for this benchmark are the same as those in **CHART 1**, and the quads produced by the geometry amplification are constructed to match the quads automatically generated by using point sprite rendering to eliminate other sources for performance differences.

same pass as rendering (a 31% speedup in the case of 2M particles). This mode of operation is made possible only though the use of memory export.[11]

4.2 Rendering

While the performance of rendering varies greatly depending on things like viewpoint, particle size, etc., making a complete presentation of rendering performance difficult, we can compare the different forms of rendering with each other. The particle system implementation in this thesis uses two rendering modes; point sprites rendering and quads through geometry amplification. Chart 3 presents the difference in performance for these two modes. The numbers include only rendering time (including the geometry amplification for the quads), the quads were matched to the size of

[11] Unfortunately at the time of writing there was an apparent bug in the driver software which caused intermittent crashes when using memory export in this way. The problem has been reported, and will hopefully be resolved in a future release. Simulating and rendering separately had no such problems, though.

the point sprites to produce a fair comparison. The performance difference is thus solely due to processing more geometry, and the logic needed to compute the quad corner positions in the vertex shader.

The absolute difference between these two rendering modes is constant for a given number of particles; the relative difference will vary depending on how expensive the rasterization of the particles is (which depends on pixel shader cost, the size of particles etc.). And indeed in practice, stretched quads tend to cover more pixels than point sprites, and will therefore take a larger performance hit than merely the cost of amplification. Figure 6 shows an example of this (the difference in rendering times between the two modes is 10.3 ms, while the cost of geometry amplification for 2 million particles is only 9.6 ms according to Chart 3).

4.3 Particle Simulation

The performance for simulation is fairly robust performance-wise, as it does not depend on the view point or any of the rendering properties for the particles. As such the measurements presented in Chart 1 tell pretty much the whole story.

Analyzing the simulation in PIX[12] reveals that the simulation is fetch bound. This means that most of the time is spent waiting on another particle to be fetched from the particle stream. This implies that there is room for extra ALU operations (such as more complicated collision detection) without changing the overall performance at all. The PIX analysis also shows that the vertex cache has a 50% hit ratio, which is entirely expected since the vertex format is half of the maximum fetch size (see section 3.1.3).

[12] Performance Investigator for Xbox. An indispensible tool for debugging and optimizing shaders on the Xbox 360.

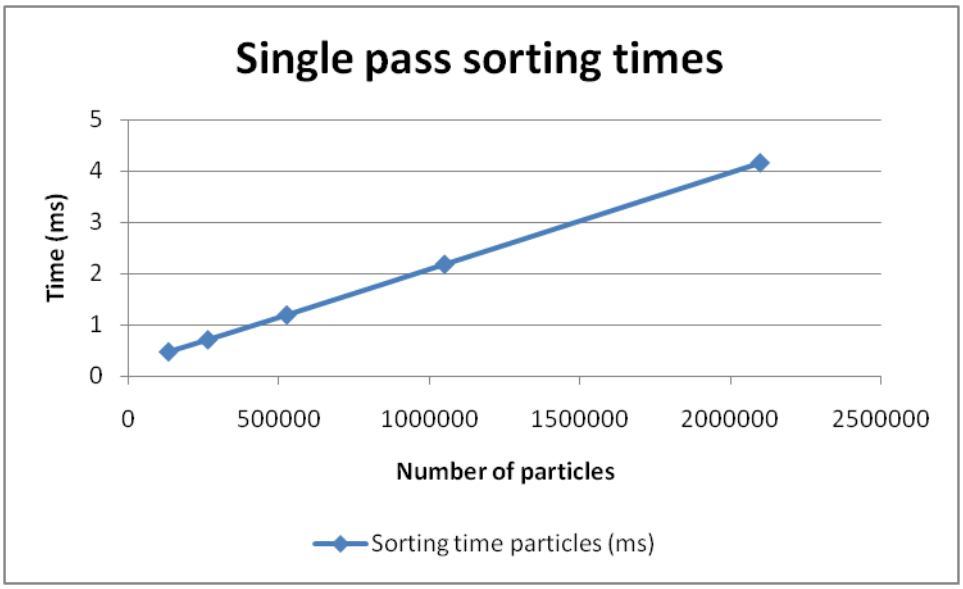

CHART 4 This chart shows off the average cost of a single sorting pass.

4.4 Sorting

Chart 4 presents the average cost of performing a single sorting pass for various particle counts. In practice, a small number of these passes per frame would be sufficient to get a "good enough" ordering for visual purposes (though case-by-case experimentation is probably warranted).

The cost for a sorting pass is linear with the number of particles, and the number of passes required is proportional to $log^2 n$. Even if with only a few sorting passes per frame, more passes will be required to achieve the same level of "sortedness" for larger number of particles (since a single pass represents a smaller fraction of the total number of passes required for a full sort).

Chart 5 displays the cost of sorting a sequence entirely. Note that this is not something that can typically be done in practice, as it is far too expensive (and wasteful to boot), but it gives a good idea of the performance characteristics.

4.4.1 Sorting for Cache Coherency

As mentioned in section 3.2.2, sorting can be used to improve locality for particles, reducing the pathological cache behaviour when, for example, sampling from a turbulence texture. We shall not linger too much on this subject, as it is not a major focus of this thesis.

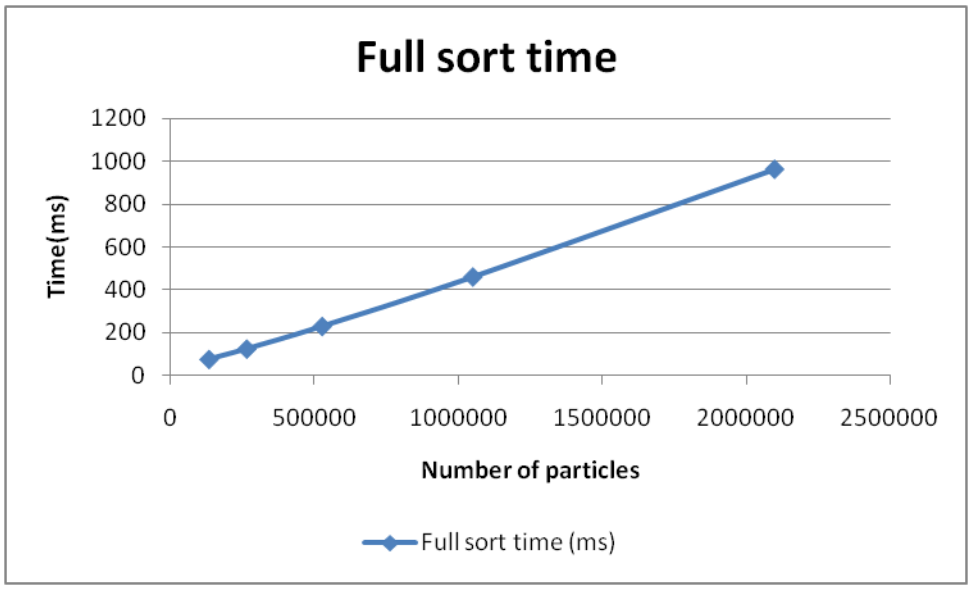

CHART 5 This chart shows off the time it takes to do a full sort for a varying number of particles. It may look linear at first glance, but in fact it follows the expected shape of $O(\log^2 n)$.

Chart 6 shows the performance characteristics for a cache bound particle simulation with a turbulence texture. While the exact performance figures varies from system to system depending on things such as temporal coherency etc., the *shape* of these curves are characteristic.

The benefit of any additional sorting pass decreases as the number of sorting passes increases. This is due to the fact that a particle system is not completely random; there is *some* frame to frame coherency. For example, consider a system which does a full sort each frame, after the first frame the system will already be "almost sorted" so it's clearly wasted effort to perform a full sort again in the second frame. This implies that the most "bang for the buck" when it comes to sorting is in the first couple of passes per frame, after that the improved cache behaviour will come at too great a relative cost. As Chart 6 shows, there is a cut-off point where the cost of additional sorting passes is higher than the benefit gained by improved cache behaviour. This point depends on a variety of factors such as the number of particles (a single sorting pass improves the "sortedness" for N particles more than it does for 2N particles), the speed of motion of particles (slow moving particles exhibit greater temporal coherency and therefore do not get unsorted as quickly), and the randomness of the particle's movements (particles that move through a strong turbulence field

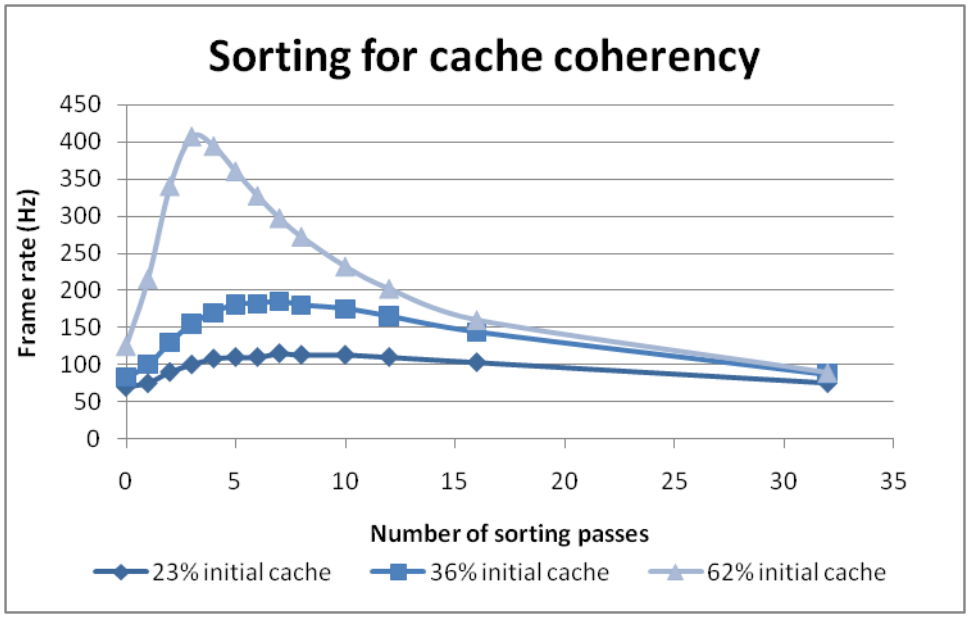

CHART 6 This chart demonstrates the performance of a cache bound particle system with 128K particles. The size of the volume texture sampled using the position was tweaked to produce three different initial cache hit rates, and then a varying number of sorting passes was added to see how performance improved in each

will tend to get disorganized quicker than particles which move through a vacuum).

This result implies that the best possible speedup will be obtained if the first couple of sorting passes is enough to get the cache hit ratio up to an acceptable level. A poster child example for when sorting helps performance is when 3-4 sorting passes can be used to bring cache coherency up from around 60% to 99-100% (and indeed we see this in Chart 6).

Sorting is an expensive operation. In fact, a complete sort of the particles in all but the most trivial systems is so costly that it completely dwarfs the cost of even highly pathological cache behaviours. However, if the system has some temporal coherency, it is possible to achieve better cache coherency through sorting, while only paying a fraction of the cost for a full sort. This temporal coherency is the *key* to using sorting for performance benefit. The more random the particle system, the less likely the prospect of speeding it up using sorting is.

It should be stated clearly that the “sweet spot” for when sorting actually *helps* performance is quite narrow. The system must exhibit very specific characteristics for sorting to be a net win (high cost and rate of cache misses, relatively few particles, high temporal coherency etc.).

5 Discussion

5.1 Summary

This thesis presents a system for simulating and rendering particle system entirely on the Xbox 360 GPU. It makes particular good use of the memory export facility to improve simulation times by performing it at the same time as the rendering, and also to perform incremental sorting of the particles for non-commutative blend modes and to improve cache coherency in certain circumstances. The performance achieved for the simulation is high – for most particle systems the limiting factor will be the cost of rendering the particles, rather than the cost of simulating them; even for systems with very simple render modes.

5.2 Future Work

One interesting property of our sorting algorithm is that it uses only two[13] of the four Memory Export elements(3). This is a wasted opportunity. One way to make use of the other two elements is to sort two equal length sequences at the same time, by simply storing them interleaved in one vertex buffer, and then having the CAS operations act on both lists at the simultaneously in lock-step (CAS(a,b) becomes CAS(2a,2b) and CAS(2a+1,2b+1)). If strict correctness of the sorting algorithm isn't important (which can be the case when sorting for cache coherency), then one could simply treat a single sequence this way and end up with two independently sorted sequences interleaved, which might be good enough for certain scenarios. If correctness is required, additional work is required to copy the two interleaved list into a buffer where they are stored separately, and then perform a merge operation (e.g. the odd-even merge as described in 3.2.3.1). This is a promising optimization since it appears that sorting is limited by memory export bandwidth, so increasing the efficiency of those writes can potentially lead to big wins.

In this thesis we only briefly discussed collision detection. It's conceivable to perform much more advanced collision detection than this.

[13] When sorting the particle elements directly, since the element type is half4 and we use two elements per particle. When sorting a key index stream we only use one of four, since the element type in that case is a single float2 per particle.

For example in (7) collision detection against a height field is performed by sampling a texture of plane equations approximating the surface. This can possibly be extended to work on arbitrary geometry, so long as the direction of particles colliding with this geometry is roughly constant (which is true for a variety of effects, such as rain), by simply rendering a view of the scene from the particle emitters viewpoint in the direction of the particles and storing the depth at each pixel. Particles could then be tested for collision against this depth map. Access to some form of normal for reflection against the collision geometry will likely be necessary, and can be gathered when rendering the depth map[14], or as a separate pass, or indeed when the collision occurs, by numerically computing the gradient of the depths. Other possible approaches would be to approximate geometry around the particle system by a series of simple primitives, such as spheres. The most important of these collision primitives could then be uploaded to the GPU in shader constants.

Another area that could do with some more attention is the rendering itself. We discussed two different ways of rendering sprites in this thesis, but there certainly are other options available. For example one could render so called “soft particles” which simply modify the alpha of a particle based on the distance to the scene geometry which gets rid of hard edges caused by intersections(13). Another possible strategy is called Soft Volume Particles where each particle is viewed as a 3D volume where the density is stored in a volume texture. The colour for each pixel in the particle is then computed by simply ray tracing through this volume, adding up occlusion and lighting along the ray(13). A potential combination of these two approaches would be to pre-integrate the occlusion and lighting normal at each point in the 3D density texture, assuming that the eye rays always pass through the same (x, y)-coordinate or each depth slice. Then these particles could be rendered as normal camera-facing sprites, but the depth of the geometry behind the sprite would be used to fetch the occlusion from the volume texture containing the pre-integrated densities. That way one would achieve “noisy” looking occlusion of a particle volume while only taking a single sample per particle pixel. This amounts to storing an arbitrary

[14] Though this would disable the “double speed” rendering available for depth-only passes on the Xbox 360(3).

monotonically increasing “occlusion function” for each pixel in a standard camera-facing 2D sprite, thereby achieving a bit more natural looking “soft” intersections against the background geometry. The downside being that the particles turn to face the camera like normal sprites – unlike fully volumetric “soft particles”.

Furthermore, it would be interesting to compare this particle system against a highly optimized CPU based simulation using the Xbox 360 VMX instruction set(2). One could speculate that it should be possible to identify specific (and very likely highly unrealistic) particle system characteristics where a CPU-based version outperforms a GPU based one (e.g. due to poor cache coherency for the GPU version). This would probably be a quite substantial undertaking, with little practical use, however.

6 Bibliography

1. **Reeves, William T.** *Particle systems - a technique for modelling a class of fuzzy objects.* s.l. : ACM Computer Graphics, 1983.

2. **Isensee, Pete.** *Xbox 360 Hardware Overview*. Xbox 360 Central. [Online] 2004. Registered developers only. https://xds.xbox.com/xbox360.

3. **Dougherty, Michael.** *Xbox 360 GPU Overview.* Xbox 360 Central. [Online] 2004. Registered developers only. https://xds.xbox.com/xbox360.

4. **Microsoft.** *Programming Guide (Direct3D 10) 2007.*

5. **Kipfer, Peter, Segal, Mark and Westermann, Rüdiger.** *UberFlow: A GPU-Based Particle Engine.* Grenoble : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. pp. 115-122. 2004.

6. **Latta, Lutz.** *Building a Million Particle System.* Game Developers Conference 2004.

7. **Microsoft.** GPUParticle Sample, Xbox 360 XDK. 2006. [Online] Registered developers only. https://xds.xbox.com/xbox360.

8. **Porter, Thomas and Duff, Tom.** *Compositing Digital Images.* : ACM Press, 1984.

9. **M, Trott.** *The Mathematica GuideBook for Programming.*: Springer-Verlag, pp. 93-97. 2004.

10. **Batcher, K E.** *Sorting networks and their applications.* Proceedings of the AFIPS Spring Joint Computer Conference 32. pp. 307-314. 1968.

11. **Westermann, Rüdiger and Kipfer, Peter.** Improved GPU Sorting. [book auth.] Matt Pharr and Fernando Randima. *GPU Gems 2.* 2002.

12. **Lang, H W.** Odd-even mergesort. 2007. [Online] http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/networks/oemen.htm.

13. **Microsoft.** *SoftParticle Sample, DirectX SDK.* 2006. DirectX Developer Center. [Online] https://xds.xbox.com/xbox360/xdk.aspx.

14. **NVidia.** *CUDA Programming Guide Version 0.8.2.* 2007.

15. **ATI.** *ATI CTM Guide, Technical Reference manual.* 2007.