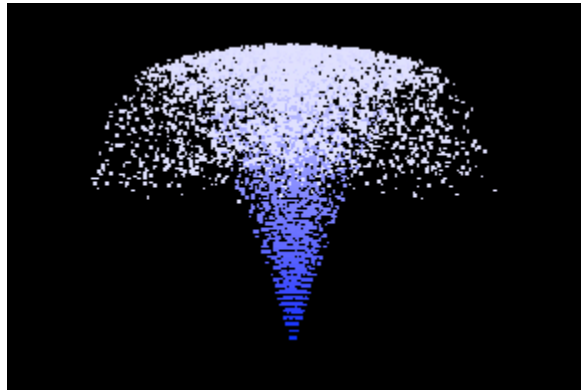


Workload Distribution and Parallelized Visibility Sorting Using the Synergistic Processor Elements on PlayStation 3



A Project Presented to the Faculty of The Guildhall
at Southern Methodist University

By Dave Loyd

(BS Computer Science, SMU, 2008)

In Partial Fulfillment of the Requirements for a Masters of Interactive Technology in Digital
Game Development with a Specialization in Software Development

07-08-2008

To the Graduate Faculty:

I am submitting herewith a project written by Dave Loyd entitled “Large-Scale Particle Systems on PlayStation 3”. I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Interactive Technology in Digital Game Development, with Specialization in Software Development.

Anton Ephanov, Supervisor

I have read this Project
and recommend its acceptance:

Wouter van Oortmerssen, Advisor

Accepted for the Faculty:

Dr. Peter Raad, Executive Director
The Guildhall at SMU

ACKNOWLEDGEMENTS

I would like to thank my supervisor Anton Ephanov for his help and support.

I would also like to thank my advisor, Wouter van Oortmerssen, for his time and assistance during the project.

Dave Loyd

M.I.T., The Guildhall at SMU, 2008

Large-Scale Particle Systems on PlayStation 3

Supervisor: Anton Ephanov

Master of Interactive Technology degree conferred July 12, 2008

Thesis / Project completed July 11, 2008

This thesis investigates the use of the Cell processor in the PlayStation 3 (PS3) to properly depth-sort particles in large-scale particle systems. Using PS3 Linux for the development platform, multiple sorting methods are explored to test efficiency for up to 500,000 particles in existence simultaneously.

Visibility sorting is broken up into two separate phases, using both the Power Processor Element (PPE) and the Synergistic Processor Elements (SPEs) in the Cell processor. On the PPE, a bucket sort algorithm is used to loosely sort the particles into a single array of data. This workload is then split into chunks and dispatched in parallel to the SPEs, where the second sorting phase is completed. In this second phase, both quick sort and Odd-Even Merge sort are tested for performance sorting the particles. Odd-Even Merge sort is then tested using only two or four passes to compare against a full sort.

The results demonstrate that sorting large numbers of particles is possible, though bottlenecks do exist. In particular, using more efficient algorithms on the SPEs can result in large speed gains, but ultimately the bucket sort performed on the PPE becomes the slowest component.

Table of Contents

List of Figures	6
List of Tables	7
Nomenclature	8
Chapter 1: Introduction	1
1.1 Project Overview	1
1.2 Objectives	1
Chapter 2: Field Review	3
2.1 Particle Systems	3
2.2 Past Research	4
2.3 PlayStation 3 Architecture	7
2.4 Configuring the Development Environment	8
Chapter 3: Methodology	10
3.1 Phases of Cell Processor Development	10
3.2 Inter-processor Communication	11
3.3 Rendering in PS3 Linux	13
3.4 Producing Particles	14
3.5 Visibility Sorting	16
Chapter 4: Results and Analysis	20
4.1 Particle Systems and Rendering	20
4.2 Data Collection	20
4.3 Analysis	25
Chapter 5: Conclusion	31
5.1 Summary	31
5.2 Future Research	31
Reference	32
Books	32
Articles	32

List of Figures

Figure 1: Cell Processor (Blachford 2005)	8
Figure 2: Software Rendering Pipeline.....	14
Figure 3: Particle Fountain Over Time	17
Figure 4: Unsorted Particles.....	19
Figure 5: Sorted Particles.....	19
Figure 6: Sorting Distribution.....	22
Figure 7: Quick sort	26
Figure 8: Odd-Even Merge sort	27
Figure 9: Odd-Even Merge sort (4 passes)	28
Figure 10: Odd-Even Merge sort (2 passes)	29

List of Tables

Table 1: 10,000 Particles (Quick sort)	23
Table 2: 100,000 Particles (Quick sort)	23
Table 3: 500,000 Particles (Quick sort)	23
Table 4: 8192 Particles (Odd-Even Merge sort)	23
Table 5: 131,072 Particles (Odd-Even Merge sort)	24
Table 6: 524,288 Particles (Odd-Even Merge sort)	24
Table 7: 131,072 (Odd-Even Merge sort 2 passes).....	24
Table 8: 131,072 (Odd-Even Merge sort 4 passes).....	24
Table 9: SPE Workload (2048 Particles)	25

Nomenclature

PS3	PlayStation 3
PPE	Power Processor Element
SPE	Synergistic Processor Element
DMA	Dynamic Memory Access

Chapter 1: Introduction

Particle systems are an important component to any modern videogame. Well-designed particle effects can enhance the visual appeal of a title and provide richer spaces with more immersive game play. With ever-increasing power, modern game consoles like PlayStation 3 (PS3) can perform more work and support more advanced features than previous systems. One of these features is the ability to properly depth-sort large numbers of particles to support graphics techniques such as non-commutative alpha blending. The keys to this goal are the PS3's Synergistic Processing Elements (SPEs) and the algorithms and methodologies employed to sort the particles.

The Cell Broadband Engine processor powering the PS3 is a revolution in microprocessor architecture (Blachford 2005). As a result, new design and programming standards are still being developed to produce efficient software systems that harness the potential of the Cell processor. Finding ways to efficiently distribute workloads and keep the SPEs busy are chief concerns in this arena. Developing ways to parallelize visibility sorting can help PS3 projects by encouraging the use of more visually pleasing effects for large-scale particle systems.

1.1 Project Overview

This project implements a two-phase sorting process to properly depth-sort large numbers of particles using the PS3 SPEs. The primary goals of this project are to efficiently distribute workloads across one or more SPEs and to parallelize the sorting for increased efficiency.

1.2 Objectives

The work of this project accomplishes two primary objectives and one secondary objective. One primary objective is to implement a two-phase sorting method which produces properly depth-sorted particles for large-scale particle systems. Second, the method should efficiently distribute work across the SPEs to make use of all available resources in the Cell processor. The secondary objective is to produce a basic particle rendering system on PS3 which demonstrates the results produced by sorting the particles.

Chapter 2: Field Review

The important first step toward a successful project such as this is to explore past research to see what others have done in the field. In addition, it is necessary to more clearly define the problem space and how past efforts map to it. This chapter will provide that definition, including the core goals and chief concerns related to the efforts of past work. Next, this chapter will examine three previous implementations of large particle systems, discussing the choices and challenges faced by each. Finally, this chapter will provide a brief overview of the PS3 and the process of setting up a development environment.

2.1 Particle Systems

Two terms need to be defined to keep the discussion unambiguous across all the research examples and this project. We define a particle system as a collection of numerous graphical objects which simulate some physical behavior in real-time. The graphical objects may be as simple as point-sprites or more complicated, like quads. The simulation is for many types of effects, such as fire, smoke, water, etc... We restrict the discussion to real-time simulations, since the focus of this project is video games in general and PS3 games in particular. Indeed, the papers discussed here were also focused on real-time goals.

The use of the term “numerous” in the definition is vague intentionally. This is related to the term “large”, as in “large particle systems”. Large, or numerous, refers to the number of particles simultaneously in use. The term is necessarily context-sensitive for a few reasons. First, it is a reflection of the hardware capabilities available for the discussion. Large for PS3 is not the same as large for PS2 or Xbox 360. Second, how many is “large” depends on factors involved in the particle system itself. A system with a large number of stateless point-sprite particles with no collision detection is not the same as a system with a large number of quads that include alpha-blending and visibility sorting. Thus it is important throughout this discussion to bear in mind that “large” is a relative term and the numbers cannot be directly compared in many cases.

2.2 Past Research

The overall goal in the three past efforts described in this chapter was to implement large-scale particle systems exclusively on the GPU. This goal encompasses a drive for reduced load on the CPU and bandwidth used in transferring data from CPU to GPU. It is also a drive toward increased parallelism using the power of the modern GPU architecture. While the fundamental goals are the same for a PS3 implementation, the underlying hardware architecture is not, so the discussion needs to be framed more correctly. The goal for the PS3 project is to implement visibility sorting and efficiently distribute the work among the SPEs in the Cell processor to sort large numbers of particles. Both reduced load on the PPE and increased parallelism are goals. As PS3 Linux has no access to the GPU, the focus of this project is on the use of out-of-core sorting algorithms for parallelized visibility sorting. In a real PS3 development environment, the data produced from this project would likely be fed into another component of a larger framework or sent directly to the GPU for rendering.

While working to achieve the aforementioned goals, certain choices faced each of the past researchers. Each one provided explanations for their choices, but did not universally agree on any one solution path.

The first choice is between stateless versus state-preserving particles. Stateless particles are described entirely by a set of initial conditions and a closed form function (Latta 2004). State-preserving particles retain attributes about each particle, such as position and velocity, and use some numerical integration scheme to update particles over time (Sylvan 2007). All of the examples in this paper use state-preserving particles, though this choice is not universal (Knott 2003).

A second fundamental decision is how to store information. In these three cases, that decision was made based on the available features of the target hardware and software. In one case, floating point textures were used both as input and output in a double-buffer setup (Kolb 2004, 124-125). In another case, the use of OpenGL SuperBuffers provided the means to allocate and use GPU memory for storing particle system information (Kipfer, 2004, 116). In the case of the Xbox 360 project, a feature called memory export was exploited to store

information (Sylvan, 2007, 9). Memory export allows for direct memory access from any shader and includes full scattered memory writes (Sylvan, 2007, 9).

Sorting is another major consideration, since this is a key component for increased parallel processing and it is an expensive thing to do, both computationally and in terms of memory use. Each of the examples discusses this decision in-depth, but the choice boils down to two options in all three cases: Bitonic Merge Sort vs. Odd-Even Merge Sort.

Odd-Even Merge Sort is a variation of merge sort. The array of data is split into two sub-arrays, one consisting of the even elements, one of the odd elements. This is applied recursively until the sub-array consists of only two elements. The sub-arrays are sorted and then merged back together (Lang 2007). This method makes it highly parallelizable, but requires that the number of elements to be sorted always be a power of two. Another advantage with Odd-Even Merge sort is that the sortedness of the data never decreases with any pass. Therefore, it is possible to spread the sorting over a few frames to achieve a fully sorted set without having to do everything in a single frame.

Where Odd-Even Merge sort works with a monotonically increasing array, Bitonic Merge Sort "...builds both an ascending and descending subsequence of keys (that is, a bitonic sequence) and then merges the two subsequences" (Kipfer 2005, 739). The main drawback to this method is that, unlike Odd-Even Merge sort, the sorting cannot be spread out over time using multiple passes.

Both sorting methods are held to be valid based on one's particular circumstances (Kipfer 2005, 733-746). In two cases (Kolb, 2004, 126-127) (Sylvan, 2007, 24-38) Odd-Even Merge Sort is used. Both papers cite the advantages that the complexity of the algorithm is the same regardless of the "sortedness" of the data and that the sortedness is always improved with each iteration. In the third case, however, Bitonic Merge Sort was used in a row-wise fashion to sort particles via key-index pairs in a 2D texture (Kipfer 2004, 118).

The choice of collision detection method is taken up by each example as well, though to varying degrees. Whether to support collisions with the world, or objects in the world, or even full inter-particle collisions is a decision guided by the goals of the project and constraints induced by previous choices, among other factors. Collision detection in general

is an interesting problem and taking it on in the context of a large-scale particle system is a significant challenge.

In the “UberFlow” case, one of the project goals was to develop “...the first particle engine that entirely runs on the programmable graphics hardware *and* includes effects such as inter-particle collision...” (Kipfer 2004, 121). Their implementation, which uses 2D sorted position textures, detects “many” of the collisions, but is not without some significant drawbacks (Kipfer, 2004, 119-120). For one thing, after each update of particle motion, the engine will handle collisions *or* sort them for view-depth, but not both (Kipfer 2004, 117). Another problem is that the collision detection requires particles to be sorted twice (Kipfer, 2004, 119). These, coupled with other problems or weaknesses with the implementation, make this a somewhat unattractive method to emulate. That said, if inter-particle collisions were a significant goal of a project, this would certainly be valuable information and might serve as a basis for further research.

In the second case, depth maps are used as implicit representations of objects for use in collision detection (Kolb, 2004, 127-128). This method tests each particle for distance against each of a series of depth maps to determine if it has penetrated a collider object. Collision response uses normals, which must be stored either with the depth value in the same texture or separately (Kolb, 2004, 128-129). This method seems at first to be an attractive route for collision detection with some rigid body objects. However, considering the memory limitations of the PS3 SPEs, it may not be possible to maintain all the necessary information at sufficiently high enough resolution to provide adequate results. Furthermore, this component makes up a significant portion of the research and so should be a major goal of any project that seeks to extend this kind of work.

One other case, which is not one of the three core examples in this paper, offers a similar approach. This method also employs distance/normal grids for collisions and particle flows along a surface. “This method is not physically accurate, but it provides visually plausible results.” (Ilmonen, 2006, 1). Depending on the project goals, this may also be an interesting approach to investigate further.

In the Xbox 360 case, collision detection played a very minor role, whereas the majority of research went toward sorting. In this case, only collisions against a plane were implemented as a “proof of concept” (Sylvan, 2007, 13).

In all three cases examined here, researchers faced some similar and some distinct challenges. However, each one made choices toward rendering large-scale particle systems on their target platforms. While some of the same challenges face this project, the primary motivation is different. The goals here are to effectively use the SPEs in the Cell processor in the PS3 to perform parallelized visibility sorting on large numbers of particles. As mentioned previously, the GPU is not accessible to PS3 Linux, so the rendering component discussed in previous research is not a focus for this project; rather the focus is on producing sorted result data which would be used in later stages of a pipeline or transferred to the GPU to perform the rendering.

2.3 PlayStation 3 Architecture

The core of the PS3 is the Cell Broadband Engine (CBE), depicted below.

Cell Processor Architecture

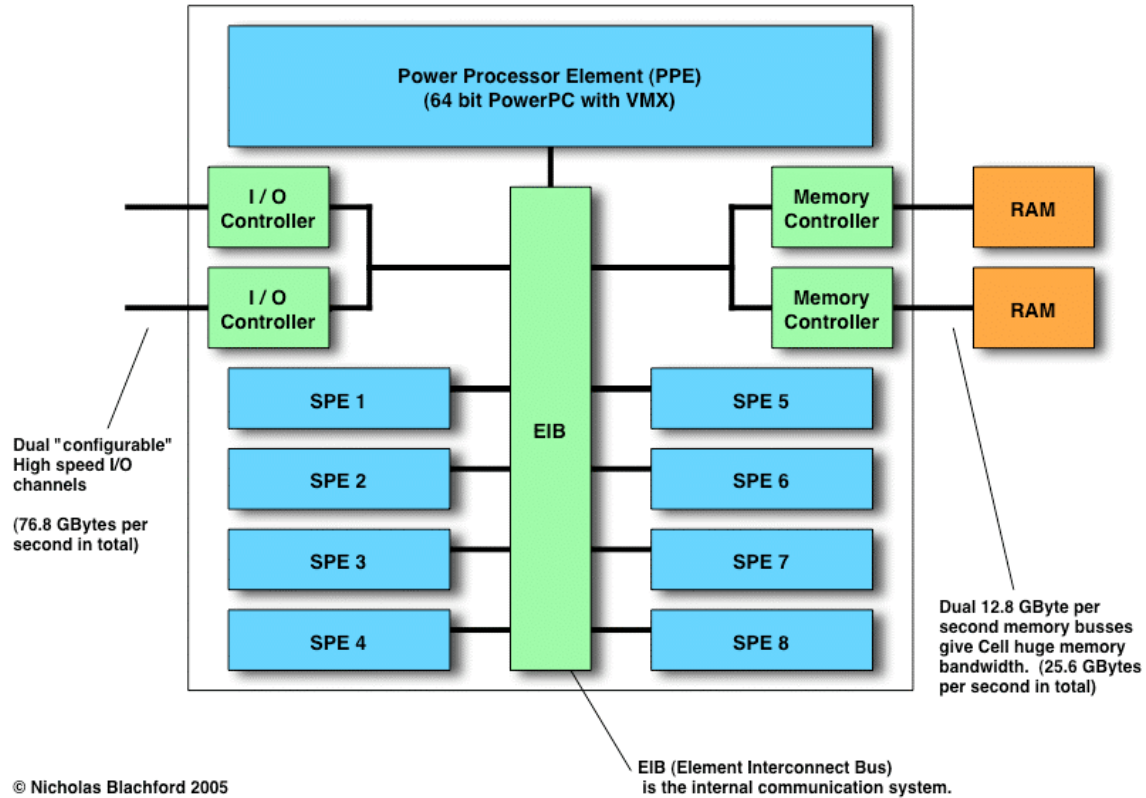


Figure 1: Cell Processor (Blachford 2005)

The Power Processor Element (PPE) is capable of two simultaneous hardware threads and is typically used for control over the Synergistic Processor Elements (SPEs). Each of the SPEs is a vector processor, containing a Synergistic Processor Unit (SPU), 256KB Local Store (LS), and Memory Flow Control (MFC) which handles DMA communication with the central Element Interconnect Bus (EIB) (Day 2006, 3).

The graphics processing unit (GPU) in the PlayStation 3 is an NVIDIA "Reality Synthesizer" (RSX). As mentioned previously, PS3 Linux is not allowed access to the GPU. Therefore, this project will focus only on the Cell processor.

2.4 Configuring the Development Environment

There is more than one way to configure an environment to develop software for the Cell processor in general and for PS3 Linux in particular. In the previous generation (PS2 Linux), a common setup included using Visual Studio on a Windows PC for developing code and either compiling directly on the PS2 or cross-compiling and moving the compiled executables to the PS2 over the network. This approach can still be employed for PS3 Linux, but additional tools and resources are available, providing a wider range of choices for development environments.

One important tool available from IBM is the IBM Full System Simulator, part of the IBM SDK for Multicore Acceleration Version 3.0. This tool provides a Cell simulator that can be run on x86 processors under Linux. Also included with the SDK is a suite of tools for the Eclipse IDE for developing Cell programs. While the SDK is platform independent (i.e. it is not specifically designed for developing PS3 applications), it does provide comprehensive information about every aspect of the utilization of the Cell processor through the simulator.

When it comes to PS3 Linux, access is easier and resources are more widely available than in the previous generation. In the past, you needed a specific PS2 disc to boot into Linux. With the PS3, booting into Linux is basically as easy as selecting “Other OS” from the system menu. In addition, there are multiple distributions of Linux available, one of the more popular ones being Yellow Dog Linux. This more accessible system, coupled with the vastly more powerful hardware, means that another development environment choice is to do everything in Linux, either with some development on a Linux PC over the network or even entirely on the PS3 itself. The latter of these, though, seems less attractive, due in part to some pitfalls in getting the display to work (Bartlett 2007).

The IBM SDK, which runs exclusively on Linux, coupled with the availability of the SDK in Yellow Dog Linux, makes a full Linux environment an attractive configuration for PS3 Linux development. The tools available in Eclipse provide a full IDE and the simulator provides the ability to debug and tune Cell applications with a full range of debugging options, including performance metrics, breakpoints, etc... In addition, IBM provides a full range of documentation and resources for Cell development through its developerWorks website.

Chapter 3: Methodology

The primary objectives for this project are to implement parallelized visibility sorting and to keep the SPEs busy with well-distributed workloads. These objectives serve as a basis for the research goal of making efficient parallel use of the SPEs and offloading work from the PPE.

The core components of this project are the particle rendering system, the algorithms used for visibility sorting, and the profiler, which will collect the data used for analysis. The implementation of the particle rendering system was completed in phases roughly corresponding to the “typical Cell software development flow” (Day 2006, 29). The profiler was developed simultaneously, allowing for the collection of data early and often throughout the project. The algorithm used for visibility sorting was critical to efficiency and developed in the later phases of the project. This chapter will discuss each of these aspects of the project, as well as the final product produced as a result of this endeavor.

3.1 Phases of Cell Processor Development

As the Cell processor provides a new architecture (see Chapter 2), so the developer needs to provide new development processes to produce the best results in the time available. This project followed one prescribed method (Day 2006, 29). The main phases in this method are first to get everything running strictly on the PPE, then to move some of the heavy lifting to a single SPE, and finally to spread the work among multiple SPEs.

In the first phase of this project, the entire rendering pipeline was developed, input systems were setup, particle systems were built, and data was collected all on a single thread on the PPE. While this was slow, it did provide a basis for ensuring that things were working correctly while avoiding synchronization and communication problems by trying to use the SPEs too early.

In the second phase, small numbers of particles were sorted on a single SPE. During this phase, the PPE still provided the majority of the work, including physics and rendering

updates, but sorting was handled exclusively on the SPE. Communication was established with the SPE and synchronization issues were resolved as a primary goal in this phase.

In the third phase, workloads are setup and distributed to multiple SPEs simultaneously to attain maximum performance. It is crucial that the previous phases be completed before attempting this phase, as troubleshooting problems becomes much more complex with two to six SPEs involved instead of one.

3.2 Inter-processor Communication

Communication and synchronization between the PPE and the SPEs can be done in multiple ways. Each method provides trade-offs between features and levels of complexity. This section briefly describes the four methods and discusses the one used for this project.

The first method, and the one used for this project, is the use of built-in mailboxes in each SPE. In a nutshell, mailboxes function as a First-In, First-Out (FIFO) queues specific to each SPE. This mechanism provides an immediate means of communication and message passing. Each SPE has a single inbound mailbox, which is four messages deep. Messages can only be in the form of 32-bit values.

The mailbox communication mechanism proved most useful, since the FIFO functionality is built-in and libraries already provide methods to access and use them. The messages passed were simple commands, such as the PPE instructing an SPE to fetch the next block of data for sorting. Also, when a message is read from the mailbox, it is consumed (i.e. removed from the FIFO). Each SPE also has two outbound mailboxes, each only a single message deep. The outbound mailboxes behave just like the inbound mailbox, except they are for the local SPE to write to and the PPE to read from. After completing a block of work, SPEs use this mailbox to notify the PPE that they are finished.

Since the SPE has only two outbound mailboxes, this presents one potential issue when using this mechanism. That is, an SPE will block if it tries to write a message to a full outbound mailbox. This can be a problem if the PPE does not poll the SPE mailboxes often enough, since it will not consume those messages and the SPE could stall waiting for the

communication to occur. On the other hand, polling by the PPE generates traffic on the EIB, which could slow down DMA transfers which are performing useful work. Therefore, the PPE cannot simply poll all the time as this would cause unnecessary traffic.

The communication method in this project uses mailboxes in a fork-and-join fashion to dispatch sorting work among available SPEs. During the dispatch phase, if an SPE inbound mailbox is not empty, the PPE will poll it for work completion and move on to attempt to dispatch the work to another SPE. During the join phase, the PPE polls each mailbox, waiting for all work to complete.

As mentioned previously, mailboxes can only contain 32-bit values. This presents another challenge, since in the case of sorting the SPE needs to know the address of the unsorted data, the destination address for sorted data, and the number of things to sort. The solution used in this project is to provide to each SPE a struct that contains this information at initialization time. The PPE updates this struct with new data before dispatching the “fetch new work” command. The downside of this solution is that the SPE has to do two DMA get operations for every job. The first is used to update new addresses and the count of items to sort. The second is used to actually begin fetching the data to sort. This also increases the overhead of keeping track of this data for each SPE and maintaining it outside of the FIFO system.

Another method of communication is signaling. Each SPE contains “...two identical signal notification registers...” (Arevalo, 2008, 187). This allows the PPE to send 32-bit messages to an SPE via these registers. This method, while not difficult to implement, does not provide a robust system like mailboxes that can be used as a FIFO. One advantage of signals over mailboxes is a very low latency for non-blocking signal operations, but the lack of other capabilities outweighs this advantage for the needs of this project.

A third method of communication is the use of SPE Events. “Events is an SPE mechanism that enables a code that runs on the SPU to trace events which are external to the program execution” (Arevalo, 2008, 199). Both the PPE and the SPEs can monitor events involving DMA, mailboxes, signals, etc... This can be done either synchronously, where the program explicitly checks the event, or asynchronously, where the program provides an event handler which catches event interrupts. While this mechanism is more powerful than

mailboxes and more efficient in terms of traffic on the EIB, it is also more time-consuming to implement. Event handling activities, which are largely automatic in mailboxes, must be manually implemented. Due to the highly compressed development time for this project, the built-in functionality of mailboxes proved more attractive than the lower latency and broader range of communication options available in this mechanism.

The fourth and final method of communication discussed here is the use of the Atomic Cache hardware. This mechanism is best used by extending "...the principles behind the handling of a mutex lock or an atomic addition" (Arevalo, 2008, 207). This is also valuable for accessing and updating shared data structures between the PPE and SPEs. However, since the data for sorting is divided into separate chunks for each SPE, no data is shared. Despite the high speed and low latency of this method, mailboxes offer a mechanism that suited the project needs more closely.

3.3 Rendering in PS3 Linux

In order to produce graphical output to the screen under PS3 Linux, it is necessary to develop an entire software rendering pipeline. This is due to the fact that PS3 Linux does not have access to the RSX GPU in the PS3. A software rendering solution was developed for this project to fulfill this requirement.

Drawing pixels on screen in PS3 Linux is handled via a frame buffer. For this project, a double-buffered solution was developed using this frame buffer. PS3 Linux will give a program access to the frame buffer and the ability to render a (front) buffer to screen. The renderer will then push pixel data into the back buffer for rendering the next frame.

Beyond the frame buffer, the rest of the software rendering pipeline can be developed as the user chooses. Since the focus of this project is not efficient software rendering solutions, a basic first-pass solution was developed that runs entirely on the PPE. The pipeline is depicted below.

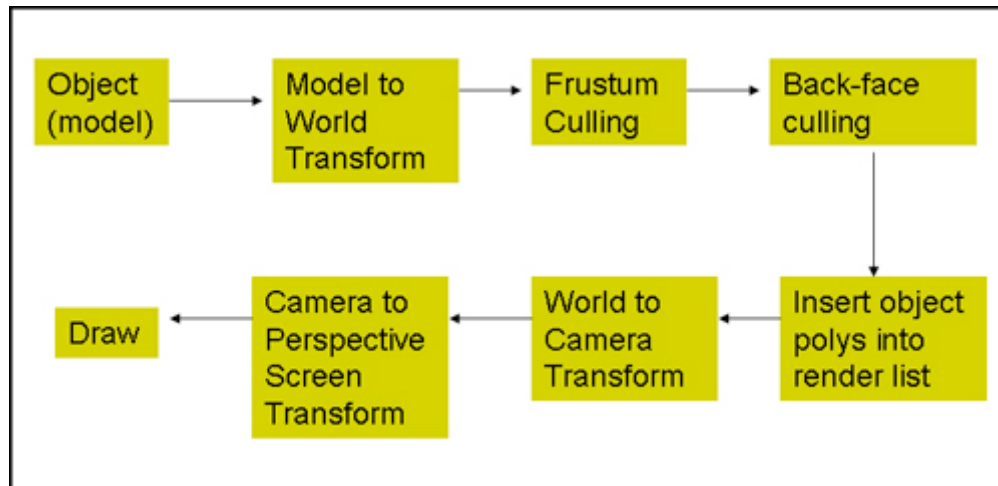


Figure 2: Software Rendering Pipeline

The engine developed for this project provides a camera which can move around in the scene via user input through either the PS3 controller or a USB keyboard.

3.4 Producing Particles

Particle systems are composed of multiple individual particles. Generating particles, emitting new ones, updating existing ones, and removing old ones are some of the challenges facing any particle system implementation. This section describes the steps taken and approach used toward achieving the goals of the project.

One of the primary concerns in console development is memory management. This is especially true when dealing with a system that can generate hundreds of thousands and millions of particles that are born, live, and die in very short time spans. The first consideration for the particle system, then, is how to handle memory use and allocation.

Before determining how to use memory for particles, it is important to investigate all the needs of the system. First, the system must support some large maximum number of state-preserving particles. Particles consist of physics and rendering data, as well as information about their current state. Sorting the particles requires additional memory, but that is a topic taken up in the Visibility Sorting section below. The particle system will also

consist of one or more particle emitters, which carry their own information. Emitters retain information about start and stop times, location and direction for emitting particles, etc... The number of emitters, though, is insignificant compared to the number of particles and so have far less memory concerns.

Memory to meet the particle requirements of this project is allocated entirely statically. Since the project goal is to produce and update many particles simultaneously, it follows that a straightforward implementation is to simply pre-allocate the maximum possible number of particles in memory. This includes statically allocating the space for state, physics, and rendering data.

Since emitters are each independent of other emitters, it is important that they remain entirely separate. This is not trivial, however, since particles are all allocated from a single, static pool (array). That makes the particle system behave more like a memory manager, allocating blocks of particles to each emitter and managing that system. A more robust system would need to handle all the aspects of that management, but for the purposes of this project, it is known in advance how emitters are allocated and that does not change at runtime, eliminating the particle management requirements from the particle system.

Emitters also function as a simplified memory manager. They must emit new particles from their pool, update existing particles, and “kill” particles whose time to live (TTL) has expired. In order to continue emitting new particles, the emitter must recycle dead particles as live ones. The implementation for this project simply walks through the particle array and, if it needs to emit new particles, uses the first available dead or unused particle. If all particles are living, the emitter simply does not try to emit more.

After allocating memory for particles based on the needs of the system, the next step is to decide how to handle producing and updating particles during the life of the emitter. The basic outline of steps is as follows:

1. Emit new particles
2. Update physics for all living particles
3. Update particle colors
4. Prepare unsorted particles for sorting

5. Sort particles
6. Transform sorted particles and insert into render list

Updating particle colors is done in this project to demonstrate the sortedness of the particles. This can be seen in the images in the Visibility Sorting section below. Of these steps, the most expensive is setting the initial velocity. For the fountain effect, this involves setting angles and generating an initial velocity in a cone. This requires two calls to the random number generator, two sin function calls, and one cos function call. The physics update for live particles is handled via forward Euler integration.

For depth sorting, only the particle's z-value is used. In order to minimize the amount of data being passed to the SPEs, only the z-value and the particle's index in the master array are used. This is not a panacea, however, as it creates some additional overhead and cache unfriendly operations, which are explained in the Visibility Sorting section below.

It is also important to realize that some components of this project are not accurate simulations for all the ways that a pipeline could function for this type of particle system. For one thing, particle data would probably be processed through the pipeline in this and possibly other stages on the SPEs and then sent directly to the GPU for rendering. In this project, however, PS3 Linux has no access to the GPU and so particle data is transferred via DMA operations back into main memory.

3.5 Visibility Sorting

Visibility (depth) sorting is the primary goal of the particle system implementation in this project. Properly sorted particles are needed when, for instance, non-commutative alpha blending is used and particles need to be properly drawn back to front (Sylvan, 2007, 10). The result of a particle system with proper depth sorting is shown in Figure three below. This is a fountain which emits particles that are initially blue and change color over time to become white. Notice in the last frame that the white particles falling closer to the camera obscure the blue particles behind them. These are not alpha-blended particles, but the properly sorted particles are clearly visible.

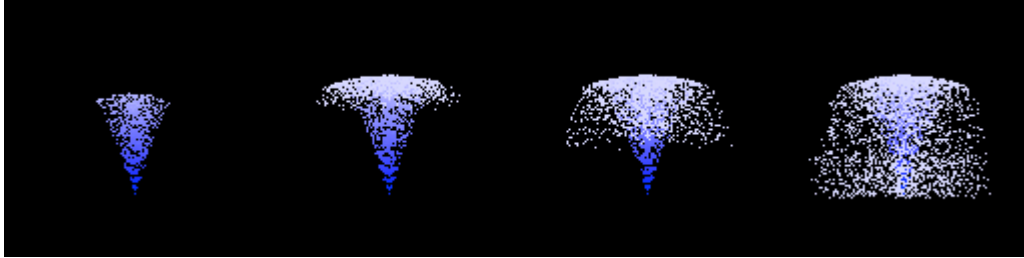


Figure 3: Particle Fountain Over Time

The visibility sorting problem using PS3 Linux is not the same as the problem facing previous researchers. In those cases, updates and sorting were handled directly on the GPU using shaders. This is not possible using PS3 Linux. Since the GPU has full access to its own memory, the challenge was to produce an efficient sorting algorithm which could be parallelized and exploit the strengths of the GPU. This resulted in the use of either Odd-Even Merge sort or Bitonic Merge sort.

In PS3 Linux, performing sorting using the SPEs is not the same challenge as handling everything on the GPU. In fact, any algorithm used on the SPEs which accesses more than 256KB of data faces the same challenge. This is the challenge of out-of-core algorithms. This is a type of algorithm that operates on a set of data which is too large to fit in memory all at once (Wikipedia, 2008). Since sorting networks like Odd-Even Merge sort need full scattered access to the data, these are no longer an option that can be applied to the entire set of data at once.

The solution for this project is a bucket sort combined with another sort for the second phase. This project investigates both Quick sort and Odd-Even Merge sort for the second sorting component. The PPE uses its fully scattered read access to the particle data in main memory to produce a single large block of particles which are only roughly sorted based on the near and far planes of the camera frustum and put into buckets. This requires an $O(n)$ walk through the data to determine the bucket sizes and a second walk to actually place items in the appropriate buckets. Once this block of data is setup, the PPE then divides the work by the number of available SPEs (up to six in this case), and dispatches the buckets of

work. Each SPE does an in-place Quick sort (or Odd-Even Merge sort) on the particles and outputs the sorted data to a destination address in memory.

The result is a single block of memory containing the sorted items. Once all the sorting jobs are complete on the SPEs, the sorted list can be inserted in the correct order into the rendering pipeline. Since the sorted items are an index value and a z-value, another cache-unfriendly operation must be undertaken to perform a fully scattered read of the particle data to insert particles into the rendering pipeline.

Repositioning the camera above the fountain effect and looking straight down at it, the following two figures show an unsorted set of particles (Figure 4) and a sorted set (Figure 5). It is important to note that the blue particles visible in Figure 4 are at the base of the fountain effect and should be mostly occluded by the white and gray particles closer to the camera, as they are in Figure 5.

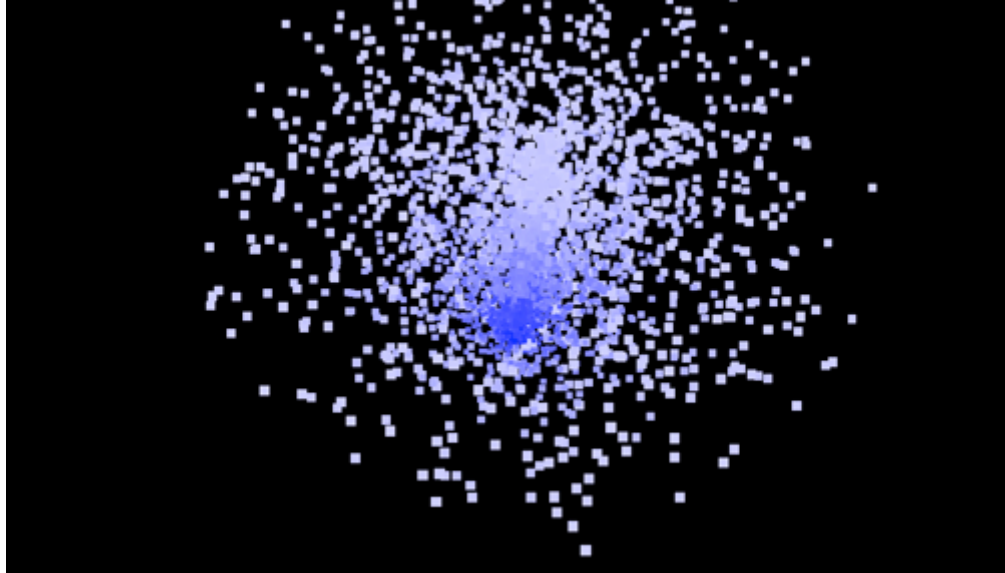


Figure 4: Unsorted Particles

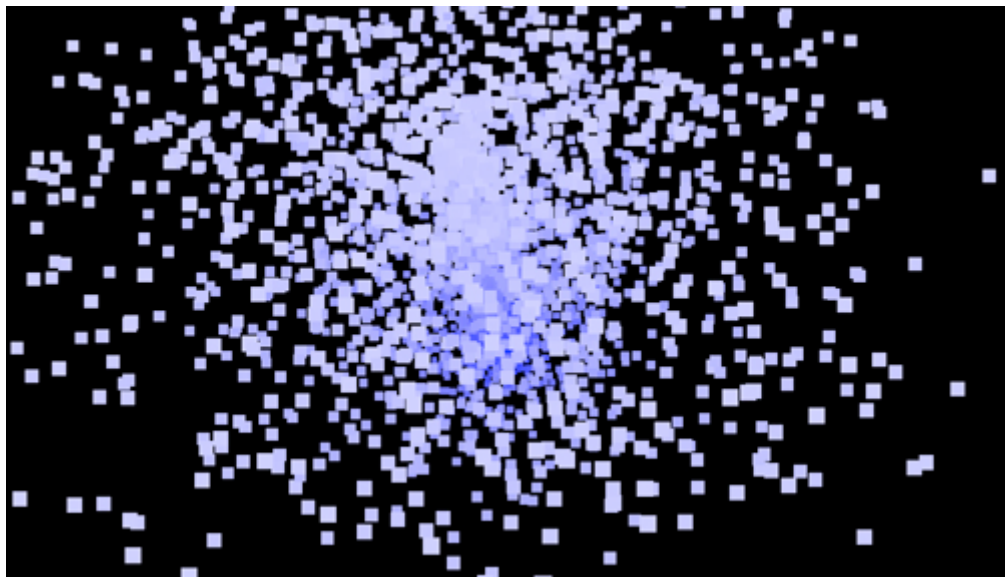


Figure 5: Sorted Particles

Chapter 4: Results and Analysis

This chapter describes the data that was collected, how it was collected, and provides an analysis of the results.

4.1 Particle Systems and Rendering

As explained in the Methodology chapter, rendering particles on screen under PS3 Linux is done entirely in a software rendering engine. Therefore, in generating results for testing the particle systems, the actual rendering phase is removed and only the sorting is investigated.

Emitting new particles and updating the physics simulation is done entirely on the PPE for two reasons. First, due to time constraints, that aspect of the simulation was not implemented on the SPEs. Second, the primary feature under investigation is the use of parallelized visibility sorting.

4.2 Data Collection

The primary means of data collection in this project is an invasive profiler. The profiler measures start and end times, writing out results to a log file, which is then interpreted and analyzed in an outside tool like Excel.

The items being sorted are an index value and a z-value. The z-value for all items is in camera space and falls inside the view frustum. The index is a single unsigned integer which is the index of the particle in the master array. This reduces the amount of information sent to SPEs to a single 32-bit floating point value and a 32-bit unsigned integer for each particle.

In order to facilitate the bucket sort aspect of the algorithm, it is important to know something about the data in advance. In this case, since particle emitters can be anywhere in the world, a common frame of reference is needed. For this purpose, during simulation the particle is transformed into camera space. If the z-value lies within the frustum, it is added to the unsorted list.

Visibility sorting is divided into three phases involving both the PPE and SPEs. The first phase is to loosely sort the data into buckets on the PPE. The second phase is to divide the work and dispatch work chunks to available SPEs. The third phase is the join phase, where the PPE ensures all SPEs have completed their work and all particles are sorted before rendering.

As described in the Visibility Sorting section of the Methodology chapter, the PPE performs two $O(n)$ walks through the unsorted data. The first pass determines bucket sizes to ensure a tightly packed array of data. The second pass actually sorts the data into buckets.

Two passes are required because of memory use. Regardless of how many buckets are used, overlap will occur and more than one element will need to be stored in a bucket. At the same time, the entire set of buckets is stored in a single array in memory. To avoid having to resize the array or rearrange data as buckets grow, the first pass determines the eventual layout of the array by counting the number of elements that will fall into each bucket. After this pass, each bucket is assigned an address for its first element in that array and every new element can be inserted into a bucket in the second pass, knowing that exactly the right number of elements will be placed into each bucket.

The second phase is where the PPE dispatches the work to the available SPEs. After the data has been sorted into buckets, the total work is chunked into sizes appropriate for SPEs. The maximum size of a single DMA transfer is 16KB. Since each item to sort is 64 bits, the maximum number of items that can be sent at once to an SPE is 2048. If more than this maximum number of items needs to be sorted, the data must be divided between available SPEs. DMA transfers less than 16KB must be 1, 2, 4, 8, or 16 bytes, or a multiple of 16 bytes (Arevalo, 2008, 115). To support this, the PPE divides the number of particles to sort by the number of available SPEs, and then adjusts the number to be a multiple of 16 bytes. For the last work chunk, the data is padded to meet the DMA requirements.

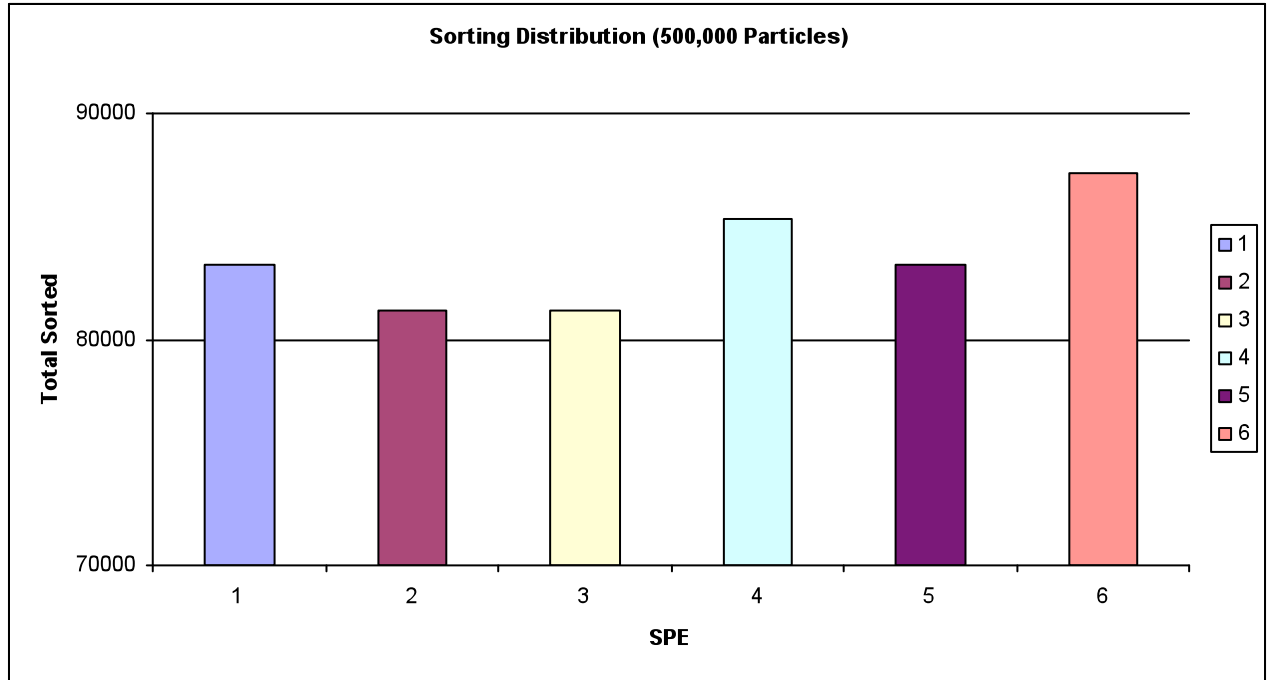


Figure 6: Sorting Distribution

New work chunks are pushed into an SPE’s mailbox queue based on the current workload for that SPE. The PPE polls each SPE and dispatches work to the least busy SPE in turn. In the above figure, the distribution for 500,000 particles is distributed roughly evenly between all six available SPEs.

The final phase is joining, where the PPE waits for the remainder of the work to complete on the current sorting pass before proceeding.

The data collected follows these three phases. In the tables below, The “Bucketize” column refers to the first phase. The “Dispatch” column refers to the second phase. This phase includes the time for the PPE to dispatch all work, which also includes waiting on some of the work to complete on the SPEs. The reason is that the mailboxes have limited queue depths. As a result, the PPE cannot queue all the jobs at once. Instead, it queues up all the work that it can and waits for SPEs to begin completing jobs so that it can queue more. Finally, once all jobs have been dispatched, the PPE waits in the final join phase, the “Join” column, for the last jobs to complete.

The first sorting method employed during the project was Quick sort. The results for this are in the following three tables. Note that most tables have a “PPE only” row, which indicates that the SPEs were not used at all; rather all the sorting work was accomplished entirely on the PPE.

Table 1: 10,000 Particles (Quick sort)

SPE Count	Bucketize (PPE)	Dispatch	Join	Total (sec)
PPE only	0.001597	0.049126	0.0	0.050723
1	0.002479	0.082562	0.020750	0.105791
2	0.002477	0.021674	0.081673	0.105824
6	0.002510	0.000135	0.083208	0.085853

Table 2: 100,000 Particles (Quick sort)

SPE Count	Bucketize (PPE)	Dispatch	Join	Total (sec)
PPE only	0.025675	3.571234	0.0	3.596909
1	0.026202	4.202879	0.224157	4.453238
2	0.029306	2.479905	0.231385	2.740596
6	0.026281	0.550106	0.166081	0.742468

Table 3: 500,000 Particles (Quick sort)

SPE Count	Bucketize (PPE)	Dispatch	Join	Total (sec)
6	0.131589	3.456319	0.276752	3.864660

The second sorting method is Odd-Even Merge sort. This is used to sort the data entirely, using as many passes through the data as necessary to finally sort the entire array. The following three tables indicate the results using this method.

Table 4: 8192 Particles (Odd-Even Merge sort)

SPE Count	Bucketize (PPE)	Dispatch	Join	Total (sec)
PPE only	0.002184	0.032770	0.0	0.034954
1	0.002095	0.013091	0.013007	0.028193
2	0.002035	0.000084	0.013032	0.015151
6	0.002061	0.000097	0.006040	0.008198

Table 5: 131,072 Particles (Odd-Even Merge sort)

SPE Count	Bucketize (PPE)	Dispatch	Join	Total (sec)
PPE only	0.035071	1.040217	0.0	1.075288
1	0.035111	0.382309	0.012272	0.429692
2	0.034862	0.185430	0.012326	0.232618
6	0.034882	0.055730	0.012198	0.102810

Table 6: 524,288 Particles (Odd-Even Merge sort)

SPE Count	Bucketize (PPE)	Dispatch	Join	Total (sec)
6	0.133539	0.247358	0.012078	0.392975

Another possibility is to perform fewer than the required number of passes through the data to achieve an acceptable level of sortedness, but not have a completely sorted array as a result. This was measured using two and four passes through the data with Odd-Even Merge sort, as seen in tables 7 and 8 below.

Table 7: 131,072 (Odd-Even Merge sort 2 passes)

SPE Count	Bucketize (PPE)	Dispatch	Join	Total (sec)
PPE only	0.037267	0.414750	0.0	0.452017
1	0.037109	0.136975	0.004341	0.178425
2	0.037200	0.066363	0.004365	0.107928
6	0.037111	0.019549	0.004234	0.060894

Table 8: 131,072 (Odd-Even Merge sort 4 passes)

SPE Count	Bucketize (PPE)	Dispatch	Join	Total (sec)
PPE only	0.037135	0.608311	0.0	0.64544
1	0.037119	0.224510	0.007193	0.268822
2	0.037328	0.108684	0.007224	0.153236
6	0.037256	0.032576	0.007159	0.076991

The DMA operations to send/receive work chunks to/from each SPE are not double-buffered, which means that the SPU stalls waiting for DMA operations to complete. This is sub-optimal and performance could be improved. Using double-buffering, the SPU could sort data on the current buffer while DMA transfers are completed on the second buffer. The distinction between the times spent on each SPE for DMA transfers versus sorting is explored in Table 9.

Table 9: SPE Workload (2048 Particles)

Method	DMA In	Sort	DMA Out	Total (sec)
Quick sort	0.000041	0.083624	0.000018	0.083683
Odd-Even Merge sort	0.000051	0.006201	0.000043	0.006295
O-E Mergesort (4 pass)	0.000053	0.003396	0.000021	0.003470
O-E Mergesort (2 pass)	0.000053	0.001524	0.000022	0.001599

4.3 Analysis

Two separate sorting methods are used in this project, both Quick sort and Odd-Even Merge sort. The two main parts of the sorting process are bucket sorting data and further sorting the buckets using one of these sorting methods. This process is done entirely on the PPE, without using a single SPE, as a baseline measurement. The process is carried out again using one SPE, then two, and finally all six to accomplish the second sorting phase.

In the case of Quick sort, the PPE actually outperforms the single SPE through all 100,000 particles, as seen in the figure below. One important point about Quick sort is that the pivot was not selected in the middle of the array. It is also the case that somewhat sorted data is being sent to Quick sort, which could result in worse performance. In either case, performance could possibly improve, but another approach is used instead, Odd-Even Merge sort.

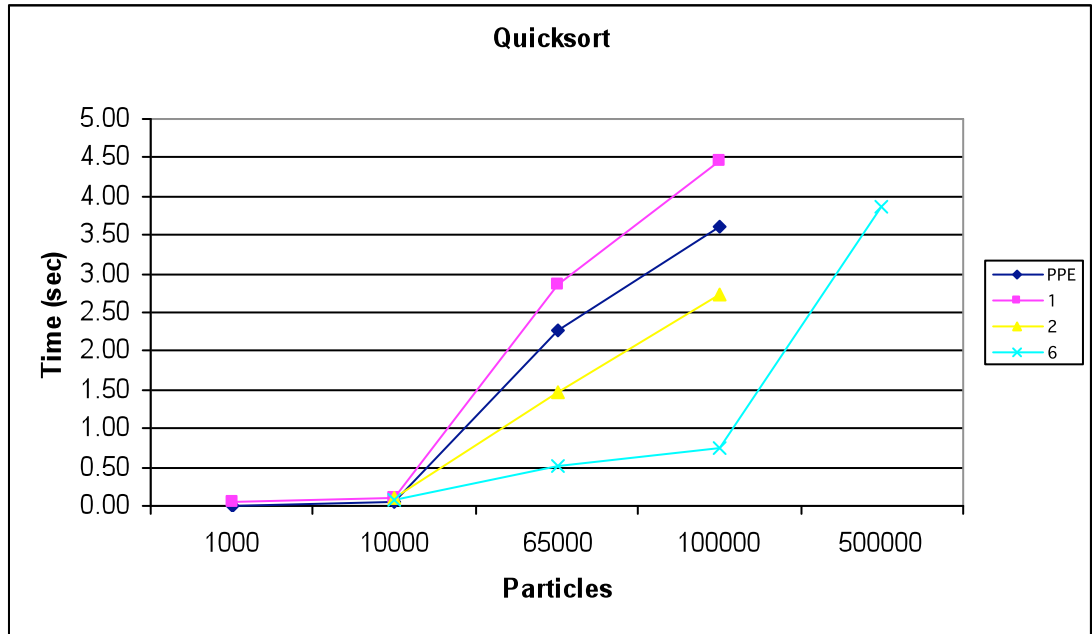


Figure 7: Quicksort

The second approach is to use Odd-Even Merge sort instead of Quick sort for the second sorting phase. This method offers two main advantages over Quick sort. First, its performance is independent of the data being sorted or of the level of sortedness of the input data. Second, the sortedness of the data never decreases in any pass through the data. This means that the number of passes through the data can be controlled to achieve the desired level of sortedness over time without fully sorting the data every frame. One caveat with Odd-Even Merge sort is that the number of elements to sort must be a power of two, which explains the difference in particle numbers in the following graphs.

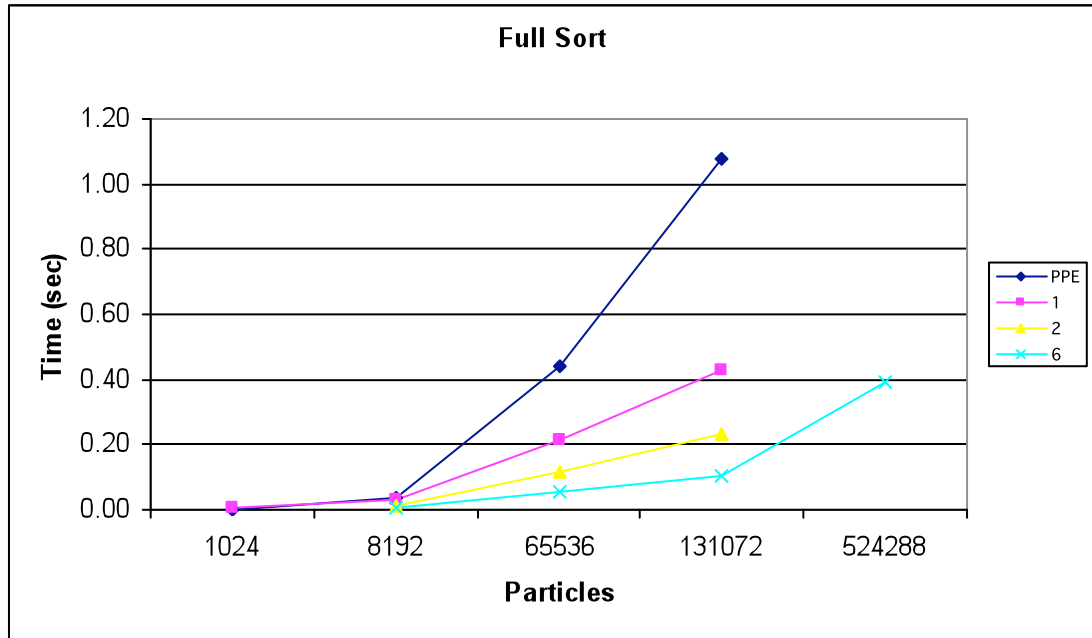


Figure 8: Odd-Even Merge sort

Performing a full sort on the data with Odd-Even Merge sort is much more efficient than Quick sort. However, it still takes about 0.1 seconds to fully sort 131,072 particles using all six SPEs.

Bringing the number of passes down to four helps to close the gap between using all six SPEs and using only the PPE or just one or two SPEs. However, six SPEs still take about the same amount of time to sort 131,072 particles, as seen in the following graph.

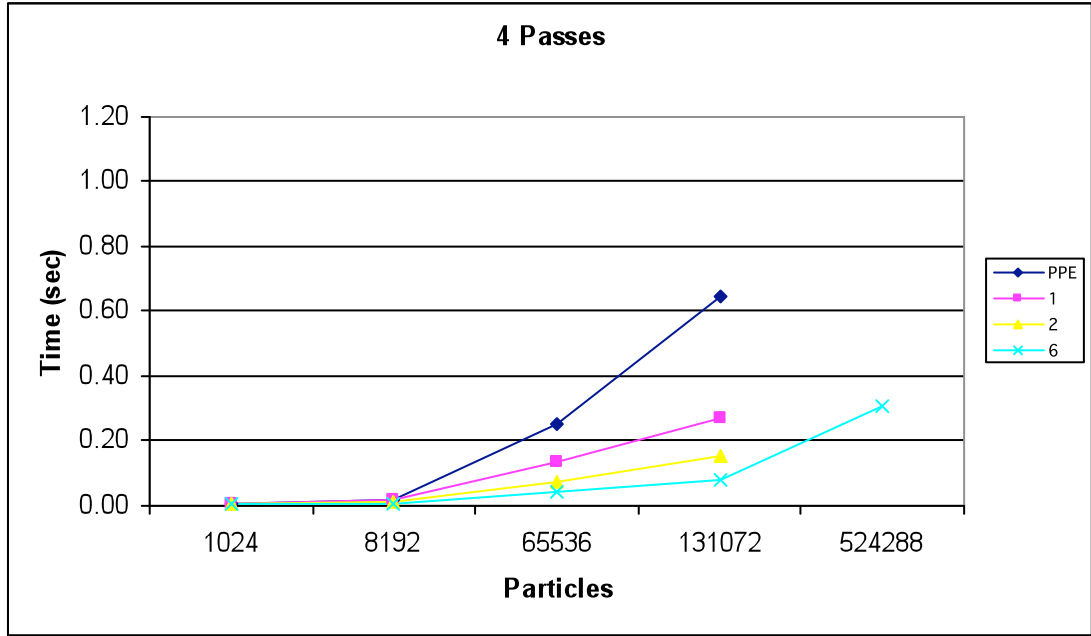


Figure 9: Odd-Even Merge sort (4 passes)

Reducing the number of passes to two produces yet another gain in performance, further bridging the gaps and highlighting another point. Again, as seen in the graph below, performance using all six SPEs is only improved slightly over using four passes.

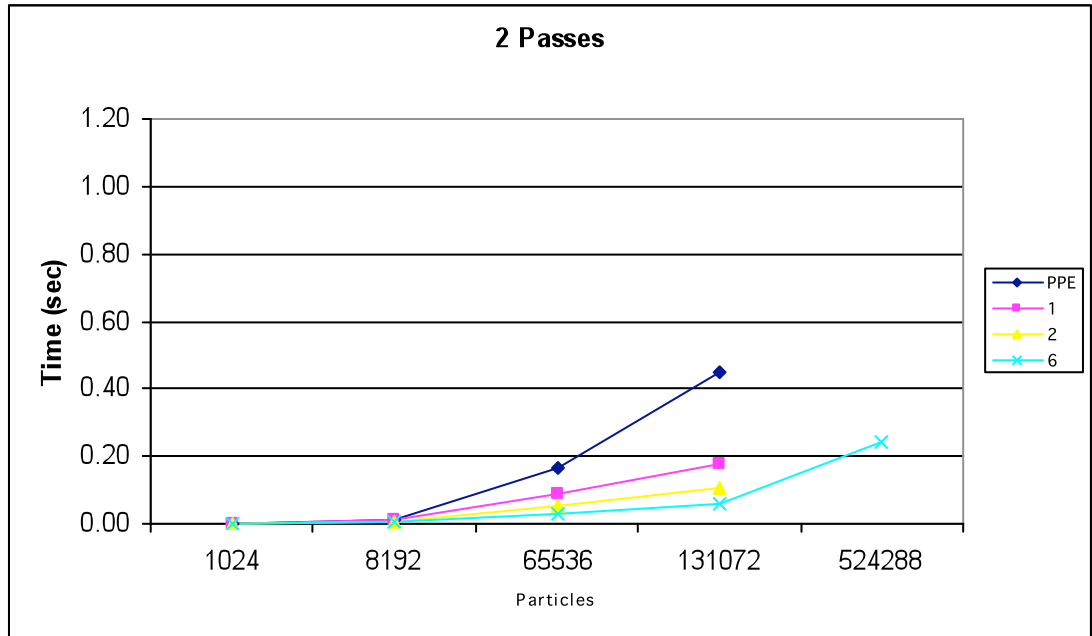


Figure 10: Odd-Even Merge sort (2 passes)

As can be seen in the tables in the previous section, roughly 0.03 seconds is spent by the PPE putting data into buckets for 131,072 particles. With Odd-Even Merge sort, the actual time spent sorting using six SPEs goes from 0.07 seconds for a full sort to 0.04 seconds for four passes and down to 0.02 seconds for two passes. Thus, controlling the number of passes can improve performance markedly, but the first phase bucket sorting using the PPE becomes a bottleneck.

This highlights one drawback of the method used in this project. That is, it still relies on the PPE for computational work. Ideally, all work should be done on the SPEs and the PPE should serve to manage communication and synchronization. It would not be unreasonable to have the PPE setup the work chunks and distribute them to the SPEs. On the other hand, it is undesirable to have the PPE handling the first pass bucket sorting of the data.

Another improvement for overall performance of the algorithm and the quality of the sortedness over time would be to exploit temporal coherency. That is, as particles are emitted and become sorted, their sortedness changes very little from frame to frame. This could be used to advantage, since resorting data is excess work. One way to do this is to perform fewer passes over the data each frame, such as the tests described earlier using two

passes with Odd-Even Merge sort, but to sort the actual particles so that the overall sortedness is improved every frame. That way, over a period of a few frames, the resulting particles become well-sorted without having to achieve the result in the span of a single frame.

Chapter 5: Conclusion

5.1 Summary

The Cell Broadband Engine processor is a revolutionary new CPU architecture powering an incredible next-gen game console in the PlayStation 3. Along with this great power comes great responsibility. The responsibility is for developers to find efficient ways to harness the power and capabilities in the Cell processor.

This thesis presents two methods for properly depth-sorting large-scale particle systems using only the Cell processor. A two-phase sorting process is used by a combination of the PPE and the available SPEs. The PPE bucket sorts the data in a rough first pass and then divides the particles into work chunks, which are further sorted via Quick sort or a controlled number of passes using Odd-Even Merge sort on the SPEs.

The results demonstrate clearly that using the SPEs to handle the “heavy lifting” computational work for any aspect of a game engine is beneficial. They also show that offloading computational work alone is not enough to produce optimal solutions.

5.2 Future Research

The key to this challenge and a possible launch point for further research is to explore out-of-core sorting algorithms, such as Column Sort (Chaudhry, 2001). This may provide not only the out-of-core solution, but also a parallel solution suited to use across multiple SPEs. It does have some limitations with memory, but could prove useful nonetheless.

Reference

Books

Arevalo, Abraham, et al. 2008. *Programming the Cell Broadband Engine: Examples and Best Practices*. Redbooks.

Articles

Kipfer, Peter and Rudiger Westermann. 2005. Improved GPU Sorting. In GPU Gems 2. ed Matt Pharr and Randima Fernando. Addison-Wesley.

Ilmonen, Tommi, Tapio Takala, and Juha Laitinen. 2006. Collision Avoidance and Surface Flow for Particle Systems Using Distance/Normal Grid. Paper presented at WSCG 2006, January 30 – February 3, in Plzen, Czech Republic.

Kipfer, Peter, Mark Segal, and Rudiger Westermann. 2004. UberFlow: A GPU-Based Particle Engine. Paper presented at SIGGRAPH 2004, August 08-12, in Los Angeles, California, USA.

Knott, Dave, Kees van den Doel, and Dinesh K. Pai. 2003. Particle System Collision Detection using Graphics Hardware. Paper presented at SIGGRAPH 2003, July 27-31, in San Diego, California, USA.

Kolb, A., L. Latta, and C. Rezk-Salama. 2004. Hardware-based Simulation and Collision Detection for Large Particle Systems.

Sylvan, Sebastian. 2007. Particle System Simulation and Rendering on the Xbox 360 GPU. Master's Thesis. Chalmers University of Technology.

Latta, Lutz. 2004. Building a Million-Particle System.
http://www.gamasutra.com/features/20040728/latta_01.shtml

Blachford, Nicholas. 2005. Cell Architecture Explained Version 2.
http://www.blachford.info/computer/Cell/Cell0_v2.html

Day, Michael, Ted Maeurer, and Robert Todd. 2006. Cell Software Programming Model.
http://www.power.org/resources/devcorner/cellcorner/CellTraining_Track1

Bartlett, Jonathan. 2007. Programming High-performance Applications on the Cell BE Processor, Part 1: An Introduction to Linux on the PlayStation 3. <http://www-128.ibm.com/developerworks/power/library/pa-linuxps3-1/index.html>

Lang, H.W. 2008. Odd-Even Merge sort. <http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/networks/oemen.htm>

Wikipedia. 2008. Out-of-core Algorithim. http://en.wikipedia.org/wiki/Out-of-core_algorithm

Chaudhry, Geeta, Thomas H. Cormen, Leonard F. Wisniewski. 2001. Columnsort Lives! An Efficient Out-of-Core Sorting Program. <http://citeseer.ist.psu.edu/chaudhry01columnsort.html>