FPGA Particle Graphics Hardware

John Sachs Beeckler

Department of Electrical & Computer Engineering

McGill University, Montreal

A thesis submitted to McGill University in partial fulfillment

of the requirements of the degree of Master of Engineering.

January 23, 2006

Your file Votre référence
ISBN: 978-0-494-24941-3
Our file Notre référence
ISBN: 978-0-494-24941-3

Acknowledgments

I thank my supervisor, Professor Warren J. Gross, for his time and help. I thank CMC Microsystems for donating FPGA design resources and equipment to McGill University. I thank Altera Corporation for giving me the opportunity to present my work. I thank Tsuyoshi Hamada, Naohito Nakasato, and Dr. Ebisuzaki of the RIKEN Institute for sponsoring my visit. Finally, I thank my family and friends.

Abstract

Particle graphics simulations are well suited for modeling phenomena such as water, cloth, explosions, fire, smoke, and clouds. They are normally realized in software, as part of an interactive graphics application. Their use in such applications is limited by the computational burden and resource competition they create. This thesis presents the design and implementation of a reconfigurable hardware particle graphics system for accelerating real-time particle graphics effects: The Particle Pipe. We explore the design process, implementation issues, limitations, challenges, and new possibilities of using FPGAs for the acceleration of real–time particle graphics. The Particle Pipe has been synthesized to an operating frequency of 130 MHz and has the potential for an increase in performance of two orders of magnitude over software methods and one order of magnitude over GPU methods.

Résumé

Les simulations graphiques utilisant des particules sont bien adaptées pour la modelisation de phenomenes tels que l'eau, les textiles, les explosions, le feu, la fumée et les nuages. Ces simulations sont habituellement realisées dans le contexte d'un logiciel interactif graphique. Leur utilité dans ces applications est limitée par le la lourde exigence calculatoire et la competition crée pour le partage des ressources informatiques. Cette thèse presente la conception d'un systeme pour accélérer, en temps-réel, la simulation graphique de particules basés sur la logique programmable: La Pipeline de Particules. La Pipeline de Particules a été synthétisée pour une vitesse de 130 MHz et pourrait potentiellement offrir une augmentation de performance de deux ordres de magnitudes pour des system graphiques particulairs.

Contents

List of Tables

List of Figures

Acronyms

AA: Axis-Aligned

API: Application Programming Interface

CPU: Central Processing Unit

DAC: Digital to Analog Converter

DDR: Double Data Rate (Synchronous Random Access Memory)

DSP: Digital Signal Processor

FIFO: First In First Out

FPGA: Field Programmable Gate Array

GPU: Graphics Processing Unit

PC: Personal Computer

PCI: Peripheral Component Interconnect

RAM: Random Access Memory

RGB: Red Green Blue

SDR: Single Data Rate (Synchronous Random Access Memory)

SDRAM: Synchronous Random Access Memory

SRAM: Static Random Access Memory

VGA: Video Graphics Adapter

Chapter 1

Introduction

1.1 Reconfigurable Hardware in Standard Computer Systems

There will soon come a day when most general–purpose computers are equipped with some form of reconfigurable hardware. Computers will be expected to have a standard FPGA card, or reconfigurable fabrics and chips incorporated in their graphics cards and mother boards. What is most interesting, is the possibility of these FPGAs to be used not simply as a flexible or convenient alternative to custom ICs, but as application–programmable hardware elements in the system. Once application–programmable FPGAs have such a presence in standard computer hardware and system interfaces, many new types of applications can be accelerated and enhanced in completely new ways, being designed to make use of custom, unique, and application–specific hardware designs for *run–time* realization in standard reconfigurable hardware.

Application software will no longer be developed separate from hardware design, but instead, together and in parallel with custom supporting hardware units for run–time realization in FPGAs, providing very specific functionality, as opposed to the general functionality provided by standard hardware. Furthermore, hardware support will not be limited to one design or unit per application, but application

software having access to reconfigurable resources, may be accompanied by any number of, or entire sets of task–specific hardware designs, each realized only when needed, targeting different tasks and stages of the application.

1.2 Acceleration with Fixed Hardware and Reconfigurable Hardware

Reconfigurable hardware acceleration is not subject to the same costs and limitations that is fixed hardware acceleration. The use of fixed hardware acceleration is limited by the cost of the physical hardware, and even more importantly, the need to justify its presence in standard computer hardware. There is a finite amount of fixed hardware that can be present in any single machine. Hardware designs cannot have a standard presence in general–purpose computers, unless justified by a common need or usefulness for the majority of users. On the other hand, hardware designs meant for run–time realization in reconfigurable resources do not exist when not being used, as they are only actually realized in the FPGA when in use! Therefore, their presence or inclusion with applications is not subject to the same constraints. Only users who actually use the application will ever realize the hardware design in the FPGAs or their computers. Although the number of designs which can be *simultaneously* used is limited by a computer's FPGAs, the total number of reconfigurable hardware designs a computer and its applications can have access to is virtually unlimited. Fixed hardware acceleration in standard computer systems, is only realistic for common, universal, or generally important tasks and problems. It is impossible or impractical to have a standard presence of hardware specifically targeting one application or a use of narrow scope, unless that use is common or universally important. Once general–purpose computers have standardized FPGA resources and interfaces, reconfigurable hardware acceleration will become realistic and possible for almost any application.

We see that acceleration by fixed hardware and acceleration by reconfigurable hardware have important differences, and that the universal presence of standard-

ized FPGA hardware in general–purpose computer systems will have a great impact on applications, the design methodology of applications, and the acceleration options available to applications. Reconfigurable hardware will change the way in which hardware acceleration is used, and the types of applications and uses that it is available for. With reconfigurable hardware, it will become possible for virtually any task or application to have any number of possible hardware accelerators for use at different times, during different stages, for different problems. Hardware acceleration will be a realistic and practical option and tool regardless of how unique the task or application is, infrequently it is used, narrow the scope of application is, or how many users of the application there are. Hardware acceleration will no longer be constrained by general usefulness, or a need for generality. With reconfigurable hardware, accelerators can be designed specific to one task, providing one specialized service to a single application.

1.3 Reconfigurable Acceleration for Computer Graphics

Computer graphics is an example of a field with potential for many changes and new possibilities to be created by the use of reconfigurable hardware acceleration in standard computer systems. This is due to both the intense, real–time, computational demands of computer graphics applications, and particularly the desire and need for unique, application–specific, non–standard, specialized tasks, or in the specific case of computer graphics, graphical and artistic effects.

Fixed hardware acceleration already has a large presence in the field of computer graphics. In many ways the current state of computer graphics, and in particular, real–time graphics applications, is defined by the current hardware acceleration technologies present in standard general–purpose systems. However, the use of reconfigurable hardware would make possible and allow the use of new hardware designs and acceleration techniques currently unexplored. Most obviously, reconfigurable hardware would make possible and practical the implementation of

hardware units providing specialized, customized, and specific services, each designed especially to support one particular effect, at any moment in an application. The techniques and possibilities of such use of reconfigurable hardware accelerators is not currently possible with general–purpose graphics hardware. We believe that these new possibilities and design methodologies could potentially revolutionize computer graphics. The new possibilities of reconfigurable hardware and its use in general–purpose computer systems, could create a new application software and FPGA accelerator co–development paradigm, with the potential to greatly influence and change application areas such as real–time computer graphics.

Current graphics hardware is designed to provide general functionality and services useful to many different types of graphics applications. With FPGAs, we can start to make use of hardware which specifically implements exactly one particular effect, calculation, or service, specialized and customized, targeting one specific use, one scene of one game, one effect in one scene, and so on. This thesis is an experiment, a design and implementation case study, of the use of reconfigurable hardware to create custom reconfigurable hardware accelerators for a specific graphics technique and effect: *particle graphics*. We explore the design process, implementation issues, benefits, limitations, and new possibilities of using FPGAs for the customized run–time and real–time acceleration of this technique. We present a new approach to creating real–time particle graphics effects: *The Particle Pipe*, a hardware coprocessor system meant for run–time realization in reconfigurable logic.

1.4 Organization

[1]Section 2 of this paper presents an introduction to the problem of particle graphics, discussing the use and application, implementation, and challenges and limitations of particle graphics. In Section 3, the Particle Pipe design is studied in detail, together with important related concepts and issues. A complete test system

[1]Content from Sections 2 and 3 was previously published in [1].

implementation of the Particle Pipe is presented in Section 4, followed by discussion and analysis of performance results. Finally, Section 5 contains conclusions and discussions of implementation issues, performance potential, practical benefits and limitations, and the design methodology of using reconfigurable hardware for run–time application acceleration in general–purpose computer systems.

Chapter 2

Particle Graphics

2.1 Introduction to Particle Graphics

Particle graphics simulations are well suited for modeling phenomena such as water, cloth, explosions, fire, smoke, and clouds [2]. Dynamic, physical simulations of large groups of individually simple particles can create graphical models of objects and phenomena that are otherwise difficult to render and model realistically. Figure 2.1 shows some examples of particle graphics effects. In these simulations, systems of simple elements such as point masses, with minimal physical properties, structure, and rendered detail, evolve together, interacting with an environment, influenced by forces, and subject to a set of rules designed to produce desired effects. The general properties and evolution of the system are also determined by randomly varying initial conditions, or more precisely, the stochastic properties of those initial conditions [3][4]. The visual ensemble of such a great group of particles, its behavior and evolution, appearance, interaction with an environment, and inherent random variation, can all exhibit a great degree of complexity and detail, attempting to properly resemble the detail and randomness of the natural world.

These graphical particle simulations are generally implemented in software, as a set of effects embedded in a graphics application. The embedding of particle graphics effects in real–time, interactive applications, such as video games, presents a difficult challenge. As discussed in Section 2.3, the size, complexity, and overall

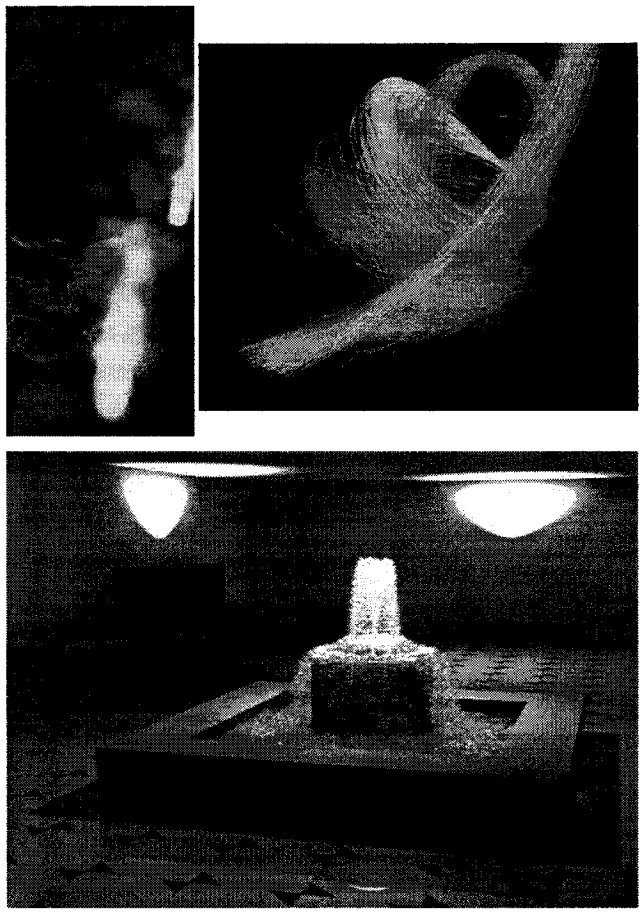

Figure 2.1: Scenes from David McAllister's particle system API.

use of particle graphics in video games are currently severely limited by the computational burden that particle graphics imposes on host applications, competing for valuable computer system resources [5].

2.2 General Implementation of Particle Graphics

The typical implementation of a particle graphics system, shown in Figure 2.2, involves the simulation state data for all particles of a system being loaded sequentially from memory, updated, rendered, and then finally written back to memory. Between frames, new particles can be created and initialized with randomly varying initial conditions. Dead particles, particles which have been active for some time or have reached a certain state, can be deactivated and replaced by newly initialized particles. For each particle processed, a set of forces is calculated for that particle

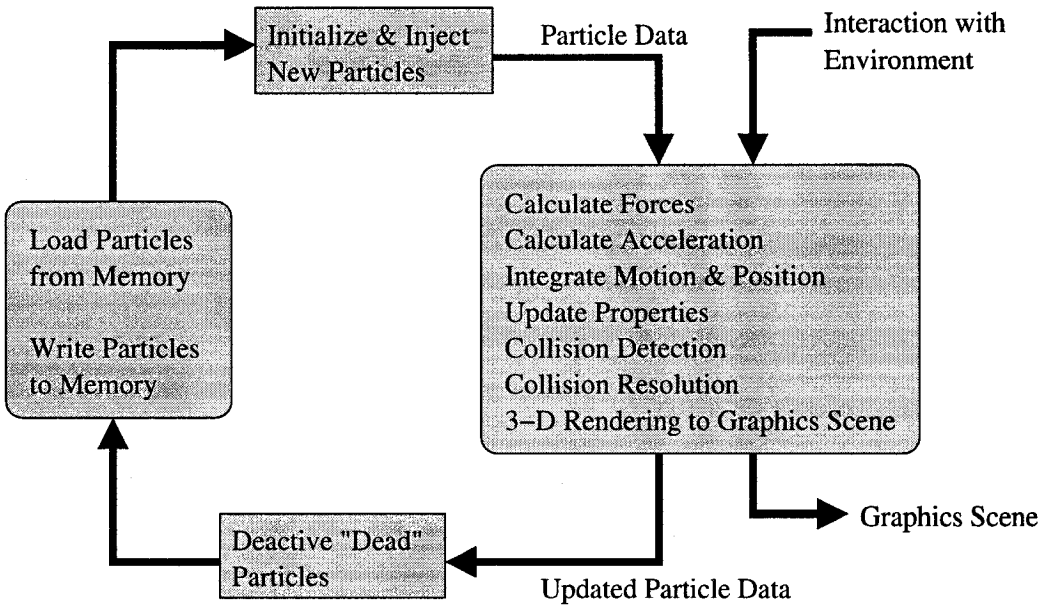

Figure 2.2: General flow of particle graphics simulation.

using its current state data and data describing the simulation environment. These forces might include gravitational and electrical forces, vortex and wind forces, liquid current forces, viscosity and friction forces, explosive forces, spring forces, and anything the designer of a particle graphics effect invents to create the desired system behavior, properties, and effects [5]. From these forces an acceleration for each particle is calculated. Each particle's motion is integrated, typically using a Euler integration step, giving it new, updated velocity and position vectors. Subsequently, collision detection and collision resolution are performed for each particle, allowing them to collide and interact with the simulation environment. Finally, the particles are rendered graphically into a visual scene. A particle could be rendered very simply as a single colored pixel, or as a streak, small texture, or anything else. Since the visual detail of the particle system essentially comes from the massive collection of distinct particles, each with its our dynamic behavior, even the most simple rendering scheme, such as a colored pixel, can be sufficient.

It is worth noting that particle graphics effects, the type of large particle systems used for graphics in real–time applications, are usually first–order. By first–order, it is meant that normal particles within a simulation do not interact with other normal particles in that simulation. Even those described in [8] can be considered as first–

order in the sense that the number of inter–particle interactions is not proportional to n^2. There are projects involving reconfigurable hardware for second–order particle calculations for off–line scientific simulation [9] [10]. However in this thesis, we are interested in particle simulations intended for real–time graphics effects, and those which we are considering will be limited to first–order systems. In these first–order systems, particles will generally not interact with each other, but interact with the system and environment, allowing the entire system of particles to be updated and processed with one single pass through the particle data. However, as described in [8], particles may interact with special force carrying particles, implemented outside of the normal particle system framework, and thus considered to be part of the simulation environment.

Looking at Figure 2.2 we can make a few key observations about the the nature of the problem and task of implementing real–time particle graphics simulations. First, many particle simulations, with some exceptions such as particle effects relying on depth sorting for alpha blending, can be ideally implemented such that the entire population of simulation particles may be updated in one single pass through the particle data. Second, if we limit the simulation interactions to those between particles and the environment (non particles), or particles with other special, force carrying particles, then not only can the simulation be updated and processed with one single pass through the particle data, but the processing and updating of each normal particle can and will be performed completely independently of all other normal particles. This means the tasks, work, and calculations to be performed in such a particle graphics system, are completely *parallel* and *pipelined* in nature. There is a clear, unidirectional, and constant flow of data, allowing acceleration by a parallel or pipelined hardware architecture.

2.3 Problems and Challenges that Limit the Use of Software Particle Systems

The embedding of particle graphics effects in live, interactive graphical applications, such as video games, is where the true challenge lies. As opposed to off–line applications, a particle system in a video game must be calculated and rendered with real–time constraints. In addition, the management and execution of particle systems in live, interactive graphics applications, can only be allocated a very limited portion of a computers system's available resources. These interactive graphics applications, are demanding and typically make full use of computer resources and hardware. Due to the requirements of managing a large particle system, and the inability of these applications to devote major portions of computer resources to a particle system, software implemented particle systems in real–time applications are severely limited in size, number and complexity of effects, rendering complexity, and interaction with an environment [11]. Software–based particle systems in games are currently limited to about 10,000 particles per frame [11].

2.4 Software Based Implementation

Particle systems in computer graphics are for the most part a software task. The great strength of software is it's flexibility. A software particle engine can be built into a graphics toolkit, and made to be completely flexible and customizable, able to create a wide variety of systems and effects. This flexibility is critical, and makes software implementations desirable whenever possible. Unfortunately, using current single processor computers, it is not possible to embed a large software particle system into a real-time application without creating critical competition for system resources to the application's expense.

One could argue that microprocessors are getting faster every day and that what is not possible with software today will be possible tomorrow. However, the problem at hand is not a stand-alone task. We are talking about embedding particle

graphics into a fully demanding software application. We have to assume that if tomorrow's microprocessors are more powerful, then tomorrow's applications will take full advantage of that power and leave us again facing the same problem: *How to add a massive particle system to an application, without competing with or burdening that application.*

2.5 Programmable GPU Based Implementation

Recently, as described in [11][12], powerful techniques have been developed which make use of programmable floating–point graphics hardware to accelerate particle systems. These techniques make use of the graphics processor to support and accelerate particle graphics. The approach is to create a set of double-buffered streams of data, using the CPU and main memory, containing particle position data, particle velocity data, and even a depth map for collision detection. These streams are fed from the CPU to the graphics processor, one pass for each "transformation" that needs to be applied to the particle data. For each pass, an output data stream is created from an input stream by software executing on the graphics processor in the form of a "pixel shader" program. The CPU creates a particle data stream in main memory, feeds it to the graphics processor which has been programmed to perform some calculation on the particles in that stream, and then obtains a stream containing data with updated values. These GPU supported systems [11] are reported to enable the implementation of systems as large as 512x512 particles, while sharing the graphics processor with other tasks. Currently this approach is not capable of implementing collisions with objects of arbitrary geometry, forces being associated with particles themselves, or any kind of 2^{nd} order effect. This technique alone, without an application, is able to create particle systems with as many as one million particles, but the number and complexity of effects are limited.

The work in [11] has succeeded in moving parts of the work involved in managing a particle system to graphics hardware, when it can be conveniently rendered without being constrained by CPU to graphics hardware communication limits.

However, the particle system is not isolated from the CPU and main memory. It continues to require CPU preparation and work at each stage of the process, thus creating a burden, limiting the size and extent of particle graphics in full featured applications. Additionally, the technique requires multiple passes or streams of data for each update cycle of a particle system, and could potentially conflict with other GPU uses. Finally, it appears that a particle system implemented in this way is not as flexible as a pure software implementation.

2.6 New Particle Graphics

What would happen if it were possible to embed huge particle graphics systems into graphics applications without significantly burdening system and application performance, taking away from or compromising some other aspect of the application? The use of particle based simulations for modeling all kinds of objects and phenomena would change drastically. The scope of use of particle graphics could expand. In general, **dynamic physical simulations on a fine–grained, particle level could become an essential part of modeling.** All particle systems could be made larger, more detailed, and have much more complex and flexible behaviors. Live rendered scenes could contain not one or two, but numerous, simultaneous particle system effects. One single scene could model a number of objects using particles without a significant decrease in quality, speed, or compromising another aspect of the host application. What is currently only possible with pre–calculated and pre–rendered, off–line applications could become a reality for live, interactive applications.

2.7 An Ideal Solution for Particle Graphics

What characteristics does an ideal solution for particle graphics have? What properties must a system or solution have to realize these goals? The implementation of a system designed to support and accelerate particle graphics in real–time applica-

tions, must attempt to isolate work, calculations, and memory accesses required to simulate, render, and manage particle graphics from the main computer resources which are needed for other tasks. The particle graphics implementation cannot burden or create unacceptable competition for CPU time, memory access, or graphics hardware usage. Secondly, although isolated from the main system, a good implementation must allow for an efficient, flexible, well defined, and adequate method of interaction between application and particles. A good particle system needs to be a fully interacting and colliding member of its environment. Finally, it must be flexible and customizable, able to implement any visual and artistic effects desired, including those not yet envisioned [5].

2.8 Reconfigurable Hardware Acceleration for Particle Graphics

Could particle graphics, more specifically the embedding of particle graphics into real–time, interactive graphics applications, benefit from custom hardware acceleration? Can reconfigurable logic be used as a platform to realize a particle graphics hardware acceleration system? How suitable is the problem of particle graphics for hardware acceleration via reconfigurable daughter systems? Work described in [6][7] investigates the use of FPGAs for prototyping news kinds of rendering schemes, with potential use for particle graphics.

Particle graphics would benefit greatly from an implementation which completely contains the particle simulations in a separate daughter system with isolated particle memory and processing hardware. In this way, large particle simulations could be managed without competing for system resources. Managing and computing a particle system is, in many cases, highly parallel. As discussed in Section 2.2, for first–order particle effects (with some exceptions such as systems that require depth sorting for rendering with alpha blending), every particle could be processed completely independently of other normal particle data in one single pass. Although in such systems all particles could theoretically be processed simultaneously, nor-

mal software or GPU–accelerated implementations still process particles sequentially, and often in several passes. Upon investigating in Figure 2.2 the sequence of tasks which must be performed for each particle processed, one finds that it is ideal for a pipelined hardware structure. There is a single, unidirectional and constant flow of data, which is easily divided into separate, independent stages. For these reasons, particle graphics has great potential for dramatic acceleration by custom reconfigurable hardware designs.

Chapter 3

The Particle Pipe

This thesis is an exploration, case study, and investigation into the design and use of a custom, application–specific, reconfigurable hardware accelerator for a specific application and task. The application and task targeted by this thesis is the creation of real–time particle graphics effects. In the following sections, our design, a custom particle graphics acceleration system for run–time implementation in an FPGA, will be described in detail. The particle graphics acceleration system designed is based upon a highly configurable, particle update pipeline, referred to as the Particle Pipe. The Particle Pipe and the test system built around it provides a complete and working proof–of–concept implementation, giving us insight into the issues, problems, potential benefits and new possibilities of incorporating custom, reconfigurable hardware accelerators together with software applications.

3.1 Particle Pipe System Overview

The Particle Pipe system, shown in Figure 3.1, is a self–contained hardware co–processing system, which completely contains, manages, executes, and renders particle graphics simulations for an application running on the host machine. The system is intended for implementation in an FPGA with access to its own dedicated RAM, and able to communicate with the system's main processor. A Particle Pipe system is primarily comprised of the following major components: the Particle

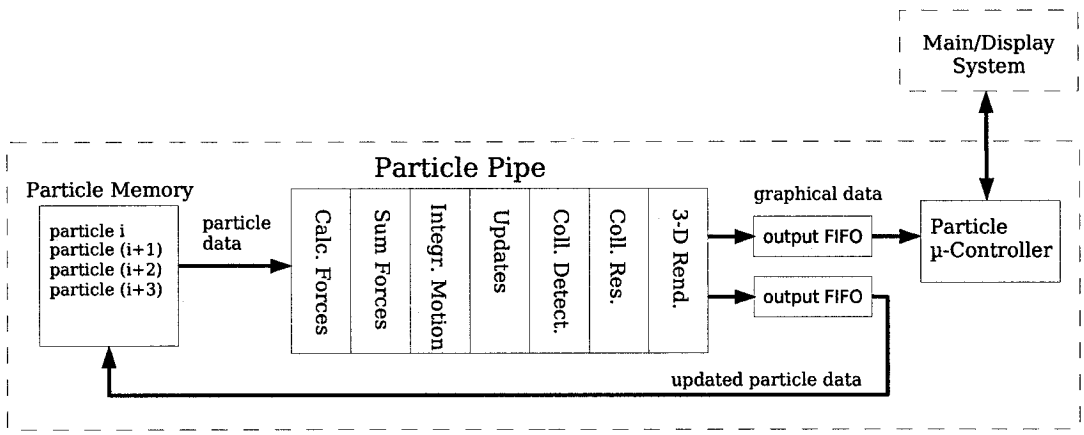

Figure 3.1: FPGA hardware particle graphics accelerator.

Pipe, a particle memory, and a system controller.

The Particle Pipe system will fully contain a large pool of particles, with which any number of simulations can be simultaneously created. It carries out all execution and tasks related to the particle simulations under the control of application software. At each frame or simulation step, the Particle Pipe system will either provide software with all or a select portion of simulation results and current data. This data will be either sent directly to graphics hardware for rendering, or back to application software for integration into application graphics. These results may include contents such as

- 3–D rendered graphical data for visible particles,
- all or select parts of the current particle state data,
- collision information.

The Particle Pipe itself, shown in the center of Figure 3.1, is a fully pipelined particle update processor. Every clock cycle a new particle's data set is accepted as input. Particle data sets travel together synchronously, down the pipeline, from one latch stage to the next. At any moment, all of the registers in one latch stage, are filled with data for one particle. As each particle's data moves down the pipe, it passes in and out of many functional units, which perform all the operations and

tasks needed for the particle graphics simulations. The Particle Pipe includes the following four major systems:

1. The Force System,
2. Integration and Updates,
3. Collision Detection and Response,
4. Graphical Rendering.

Figure 3.2 shows the basic operation of a particle simulation within the Particle Pipe system. First, in Figure 3.2a, sending commands to the particle controller, application software sets up and controls properties, parameters, and options for creating the desired particle simulations. As needed by the simulations, the particle controller will continuously create and initialize new particles in particle memory between frames, introducing them into running simulations with initial conditions created according to the specified simulation properties. This can be seen in Figure 3.2b. The controller will also set and continuously control parameter registers in the Particle Pipe, thus controlling its functionality and the properties of the simulations it implements. This occurs in response to application requests and commands to the controller specifying simulation parameters and properties. As shown in Figure 3.2c, for each simulation frame, all particle data is loaded sequentially from particle memory, and fed one particle per clock cycle, into the Particle Pipe. The Particle Pipe is a fully pipelined hardware unit, which performs all simulation operations on each particle data set. Ideally, upon every clock cycle one particle data set will begin its execution, entering the input end of the pipeline, and one updated particle data set will complete its processing, output from the opposite end of the pipe together with its rendered graphical data. As the updated particle data sets are output from the pipe, they are stored back into particle memory, while the graphical results are sent to the host system for integration into application graphics and display. In preparation for the next frame, the particle controller repeats the actions illustrated

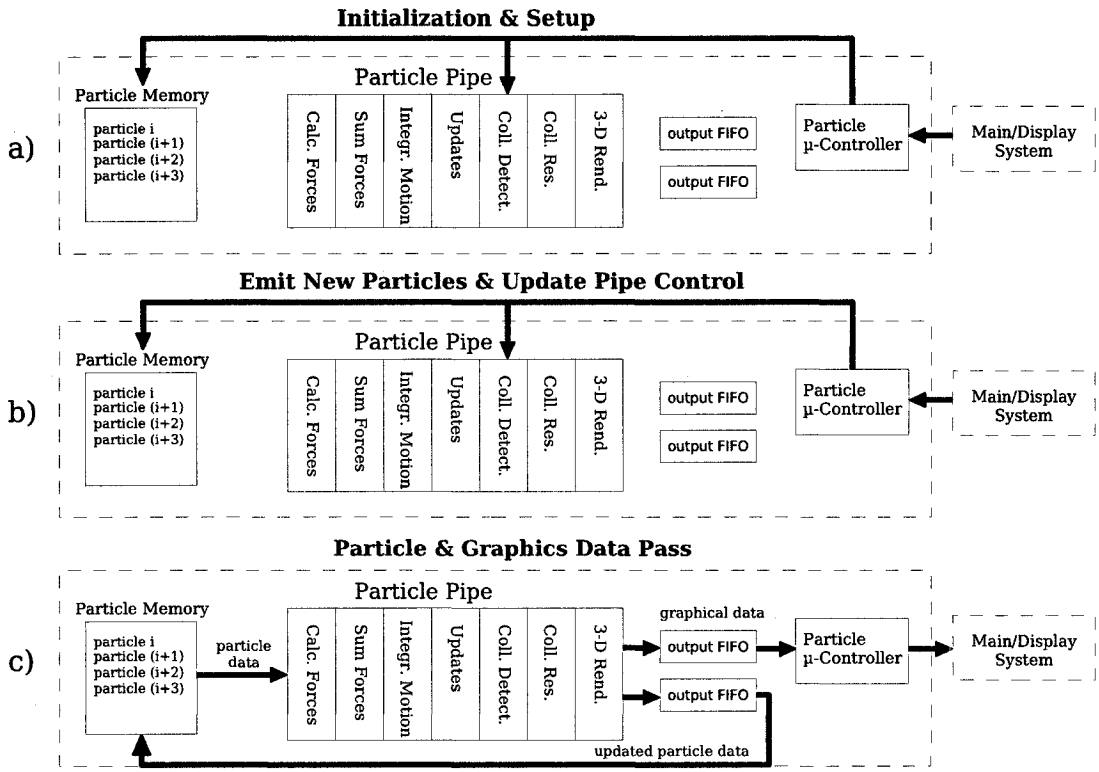

Figure 3.2: General Particle Pipe system operation.

in Figure 3.2b, initializing new particles and updating pipe parameters. Finally, the next frame proceeds with a new flow of particle and graphics data as in 3.2c.

3.2 Particle Memory

The Particle Pipe system needs high–bandwidth and exclusive access to a RAM device, dedicated to containing a large pool of all available particles for simulations. This memory, the particle memory shown in Figure 3.1, should be part of the FPGA system used to implement the Particle Pipe system and needs to be separate or isolated from main system memory. The pipe design is capable of inputting and outputting one particle data on each FPGA clock cycle. Therefore, to use the pipe to its full potential, a particle memory would need to provide read and write access rates given by

$$R_{pmem} = (b_p \times 2 \times f_{pipe}) \,, \tag{3.1}$$

where b_p is the width of one particle's data set in bits, f_{pipe} is the frequency of the Particle Pipe clock in Hz, and R_{pmem} is the required access rate in units of bits per second. Fortunately, all accesses to particle memory, with the exception of the initialization of new particles, are made in a regular, sequential order. This means that the bursting modes of RAM devices can be fully exploited to help achieve the required access rate.

Particle data is stored in particle memory as one large packed array. A particle's data set, or it's entry in particle memory must include all the data fields needed to create any of the simulations. These fields must at least include a position vector, velocity vector, color, life count, and a type field. The position and velocity vectors are 3 dimensional vectors represented in the pipe's fixed–point format. The color field can be whatever the targeted system uses to represent colors, but should correspond to the format used by any color related effects or functional units included in the Particle Pipe. The life count is an integer which is set to some value when a new particle is initialized, and can be, depending on the type of simulation, decremented on every pass through the pipe. When a particle's life count reaches zero, it can be considered "dead" or inactive, and will no longer contribute to the simulation. The particle memory entries occupied by inactive particles then become available for the initialization and creation of new particles. Finally, there is a special field in the particle data, the *type* field. The Particle Pipe system contains one giant array of particles in memory which will be repeatedly passed through the Particle Pipe. Although there is a single array, or pool, of particles, this pool will be used to create any number of distinct simulations, running concurrently. The type field will identify a particle's data set as belonging to one simulation, and will determine the functionality of the Particle Pipe, on that particle's data set at every latch stage, as it moves though the pipe.

3.3 Pipe Fixed–Point Data Format

The VHDL hardware design for the Particle Pipe is structured in such as way that many parameters and options concerning the structure and functionality of the Particle Pipe can be configured at FPGA compile time by changing a corresponding set of configuration constants in a package which is globally visible throughout the design. The pipe design will then be self–configured to implement these changes. One important aspect of the pipe design which is configurable via these constants is the fixed–point format used within the pipe.

There are several factors influencing the choice of an optimum fixed–point representation. Primarily, the fixed–point format used in the pipe will limit the precision and range possible for particle simulations. Another factor influencing the decision is the width of particle memory. It is best for the data set of each particle to fit exactly in an integer number of words, for the memory device used. If the pipe frequency is relatively slow when compared to the the particle memory access time, then particle data sets can be stored in multiple words of memory. However, if the pipe frequency and memory access time are comparable, then each particle data entry will need to fit in as few words as possible.

The Particle Pipe contains numerous fixed–point additions, subtractions, multiplications and divisions, all of which are pipelined. These circuits, become more and more complex with larger bit widths. Also, FPGAs contain dedicated hardware multiplier circuits, which the Particle Pipe design uses to implement many high speed multiplications without using reconfigurable fabric. Altera Stratix FPGAs and Xilinx Virtex FPGAs both have hardwired resources for implementing large numbers of 18x18 bit multiplications. For these reasons, an 18 bit (4:14) fixed–point format is used in the pipe. This format makes the best use of dedicated FPGA multiplier circuits, while still allowing particle data sets to fit perfectly into a low multiple of 32 bits, making optimal use of particle memory. This format also provides the necessary precision to implement a large class of effects.

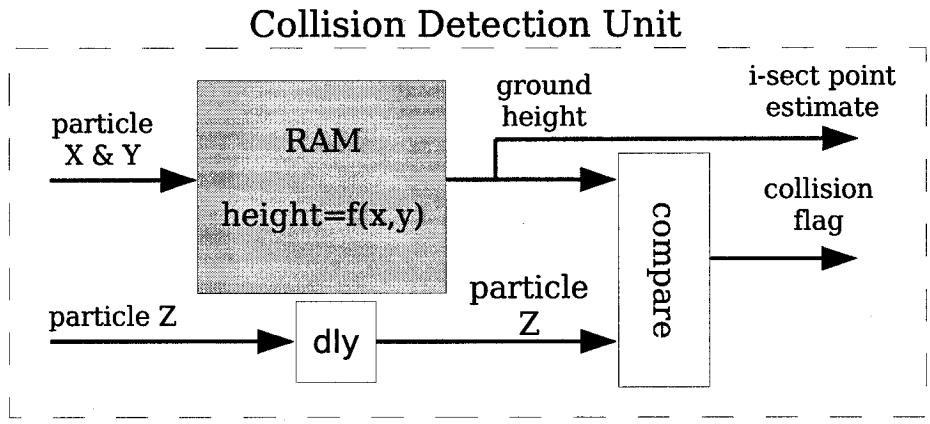

Figure 3.3: RAM based height map collision detection unit.

3.4 Pipe Modularity and Extensibility

The Particle Pipe is organized in a sequence of sections and function units. There is a section for each major operation or class of tasks that need to be performed for a particle graphics simulation. These sections do not refer to one latch stage of the pipeline containing one particle's data set. Rather, they are macroscopic operations, or related groups of operations, such as the "Force System". A very important property of the Particle Pipe design is that the interfaces between blocks and sections are consistent and well defined. The building block modules which make up sections, the function units, have consistent interfaces making them replaceable. By selecting, designing, and replacing new sections and function units which provide desired operations and behavior while satisfying the defined interfaces, a Particle Pipe can be easily and flexibly customized and extended for one particular use. This *FPGA-compile-time* reconfigurability of the pipe is an important feature, since flexibility and extendability are key aspects of any usable particle graphics system.

Highly specialized pipelines can be designed containing any function unit or section that meets the interface requirements. As discussed in [11], various operations could even be implemented as programmable RAM–based lookup tables. One such example is the height map lookup table collision detection unit shown in Figure 3.3. In this unit, particles can be collided against a programmable height

map contained in a lookup table. For each particle, a height value is obtained using an index formed from the particle's (x, y) coordinates, and compared against the particle's z coordinate. If the comparison indicates a collision, the height value obtained from the lookup table can be combined with the particle's (x, y) coordinates to form a collision point estimate. The collision may then be resolved by the normal collision response hardware discussed in Section 3.6.6.

3.5 Pipe Functionality Control

How is the functionality of the pipe controlled? Recall that the Particle Pipe system contains a giant pool of particles for making a number of concurrent simulations. Furthermore, within a single simulation, there may be several distinct kinds of particles with different properties, each subject to different forces and rules. At any moment, the hundreds of latch stages within the Particle Pipe each contain the data of different particles. The pipe needs to perform different operations on each particle's data set, at each latch stage, depending on the *type field* of the particle which is there.

Each functional unit or block in the pipeline is enabled, disabled, and customized by its own set of parameter registers. Before a particle enters a function unit or block, a set of values for the unit's parameter registers, specific to the type of particle entering the function unit, is selected from a table using the particle's *type field* as a table index. This is depicted in Figure 3.4. These selected parameters then synchronously pass through the function unit, from latch stage to latch stage, together with the particle data. At any instant, one single function unit or block, contains the particle data of numerous different particles, each at a different latch stage, and each at a different stage of execution. Each data set is accompanied by its own parameter set, corresponding to its *type field*, determining the functionality of the pipe on that particle.

On the application side, to create simulations, application software defines a number of groups of particles, or "particle types". These are sets of settings and

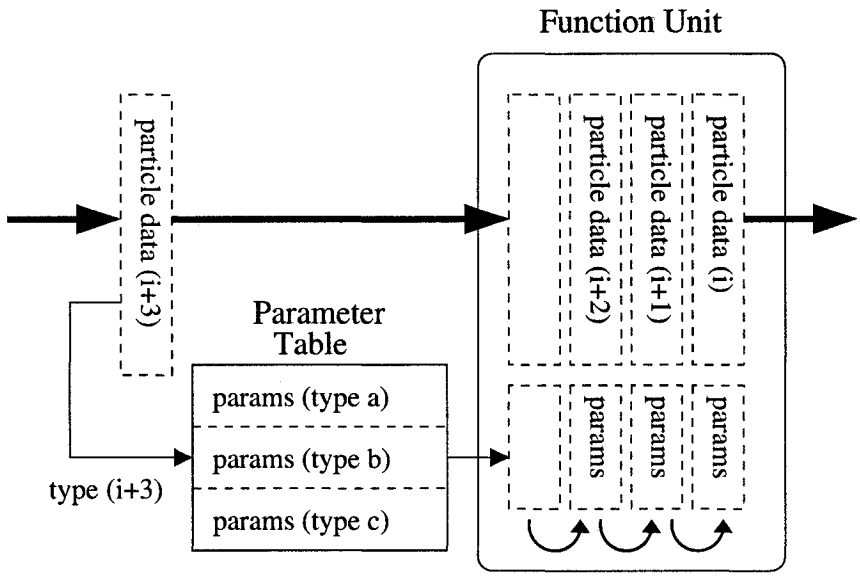

Figure 3.4: Pipe parameter selection.

parameters, along with the groups of particles associated with them. By sending commands to the Particle Pipe controller, application software fills in the values of the parameter tables for each function unit of the pipe, specifying its configuration for each particle type. For example, if there is a uniform force block in the pipeline, its job is to add a specified vector to each particle's sum of forces. Such a block might have two configuration registers. One would be an enable bit, to determine if this block should be enabled for a particle, and another would contain the force vector to be added. Now, imagine a simulation in which there are 3 types of particles. Particles of type 'a' will not experience this force at all. Particles of type 'b', will experience the $(1.0, 0.0, 0.0)$ force vector, and particles of type 'c', will experience the $(0.0, -2.2, 0.0)$ force vector. To accomplish this, the parameter tables for this function unit should be initialized such that the enable bits for particle types 'a','b', and 'c' are 0, 1, and 1 respectively. This enables the unit for particles of type 'b' and 'c', while disabling it for particles of type 'a'. The force vector entries in the parameter tables could be filled in as $(0.0, 0.0, 0.0)$, $(1.0, 0.0, 0.0)$, and $(0.0, -2.2, 0.0)$, indicating that particles of type 'b' should have the $(1.0, 0.0, 0.0)$ vector added to their total force, while particles of type 'c' will experience the $(0.0, -2.2, 0.0)$ force vector. The type values 'a','b', and 'c', implemented as the integer values 0, 1, and

type	enable	F_x	F_y	F_y
0 ('a')	0	(0.0,	0.0,	0.0)
1 ('b')	1	(1.0,	0.0,	0.0)
2 ('b')	1	(0.0,	-2.2,	0.0)

Table 3.1: Example parameter table for simple force unit.

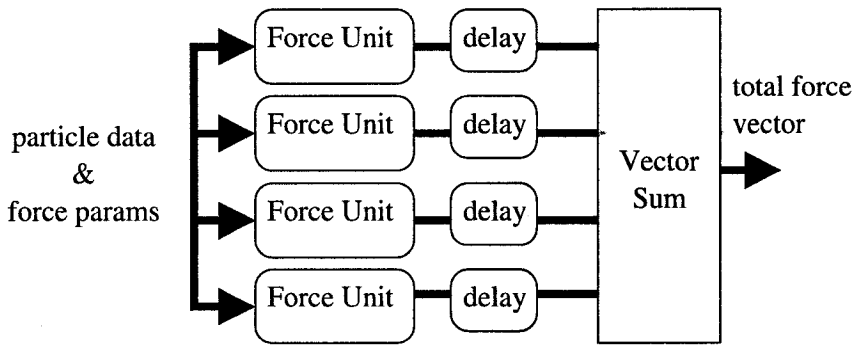

Figure 3.5: Force system.

2, could then be used as indices to the following parameter table for the simple force unit example shown in Table 3.1.

3.6 Particle Pipe Systems

3.6.1 Forces

The force system, shown in Figure 3.5, contains a set of force units in parallel. The interface to each force unit, its inputs and outputs are identical, and the total latch stage latency each unit is declared in a package of constants. Due to this well defined structure, the force system can easily by modified to include any custom set of force units. New force units can be made without knowledge of the pipe design, as long as they provide the required interface. Simple VHDL generation scripts can be used to to include new force units and select existing ones to customize a custom pipe design before FPGA compilation.

Each force unit receives as input all current particle data, together with a set of type–selected force parameters, and outputs a resulting 3–D force vector in the pipe's fixed–point format. Each force unit's output vector is delayed for synchro-

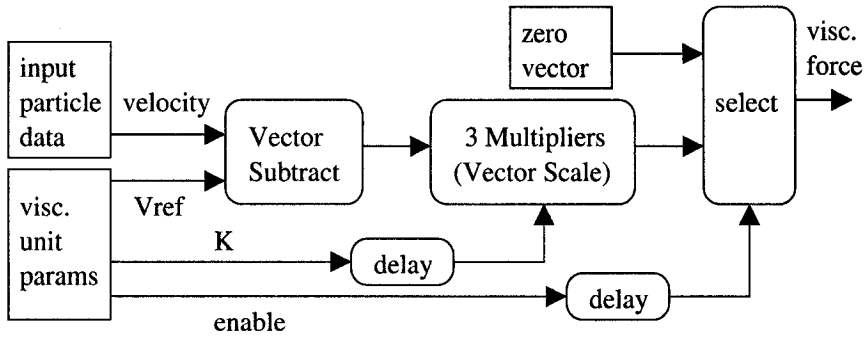

Figure 3.6: Viscosity force unit.

nization and summed to one total force vector.

Examples of easily implementable force units include uniform forces, viscosity forces, vortex forces, attractive and repulsive forces, spring forces, "random nudge" forces, and many more. In Figure 3.6 we take a detailed look at the implementation of a viscosity force unit. The unit calculates a general viscous force using:

$$\vec{f}_{visc} = k_{visc}(\vec{v}_{ref} - \vec{v}_{particle}) \ , \tag{3.2}$$

where $\vec{f}_{visc}$ is the viscous force result, k_{visc} is the scalar viscosity factor, $\vec{v}_{ref}$ is the reference velocity, analogous to the velocity vector of the fluid in which the particle is immersed, and $\vec{v}_{particle}$ is the velocity vector of the particle.

3.6.2 Force-to-Acceleration

An acceleration vector must be obtained from the total force vector. This is done in the force–to–acceleration stage using the following well known relationship:

$$\vec{a} = \frac{1}{m}\vec{f}_{total} \ . \tag{3.3}$$

First, one hardware division obtains the $\frac{1}{m}$ term from the particle's mass value. That term is then multiplied with each component of the total force vector to obtain the particle's acceleration. A less flexible particle system may be implemented without this stage entirely, thus eliminating one inversion and 3 fixed–point multiplications from the pipe design. This can be done by forcing all particles to have the same mass

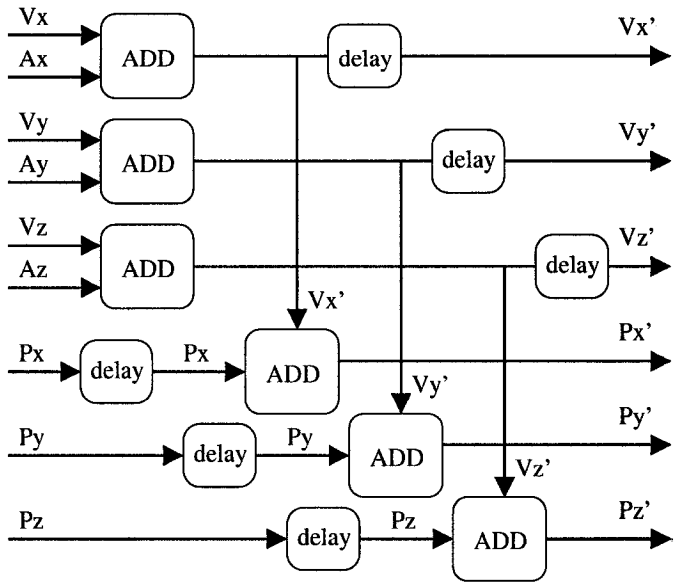

Figure 3.7: Motion integration stage.

and making the proper choice of units. If it is not practical to include a mass term in the particle data sets, particle mass may be treated as a type–selected parameter in the acceleration stage. Each particle type may be assigned a type mass, and this value stored in the pipe parameter tables.

3.6.3 Integration of Motion

Each particle's path of motion, position and velocity over time, needs to be integrated and updated according the following differential equation, which is actually equivalent to Equation 3.3:

$$\frac{d^2\vec{r}}{dt^2} = \frac{\vec{f}}{m}, \tag{3.4}$$

where $\vec{r}$ is the particle's position vector, t is time, $\vec{f}$ is the force vector, and m is particle mass.

A particle system should use the simplest integration method possible. This is the Euler step method. Verlet integration is also a possibility for some particle graphics implementations [11], but the Euler step integration method is by far the best choice for hardware due to its simplicity and explicit use of particle velocity, needed in other parts of the pipe. Figure 3.7 shows the Particle Pipe's motion in-

tegration circuit. The position and velocity vectors are updated as follows, with 2 fixed–point vector additions in parallel:

$$\begin{aligned} \vec{v}_{new} &= \vec{v} + \vec{a} \,, \\ \vec{r}_{new} &= \vec{r} + \vec{v} \,. \end{aligned} \tag{3.5}$$

3.6.4 Update Properties

Particle graphics systems have many properties that can be changed and updated using simple rules, implementing important features and effects. These include such optional and type–configurable update rules as:

- decrementing particle *life count*,
- fading particle colors by a *color step*,
- *killing* particles which satisfy some condition such as energy or position beyond a given value,
- *interpolating* between colors based on another value such as time, energy, or life.

3.6.5 Collision Detection

The particles need a well defined, flexible, and manageable way of colliding and interacting with the geometry of the virtual word in which the simulation resides. The following solution, while having the obvious problem of being unable to implement collision with arbitrary or relatively complex geometries, detects and resolves collisions with environments of simple geometry efficiently and conveniently. As shown in Figure 3.8, the collision detection system is comprised of a parallel collection of collision detection units. Each collision detection unit detects and reports collision information with one type of geometry. For example, the plane collision detection unit detects collisions of particles with a plane, and reports information about that collision if detected. The collision detection system shares the same modular approach that the force system does, making the inclusion of new units for custom

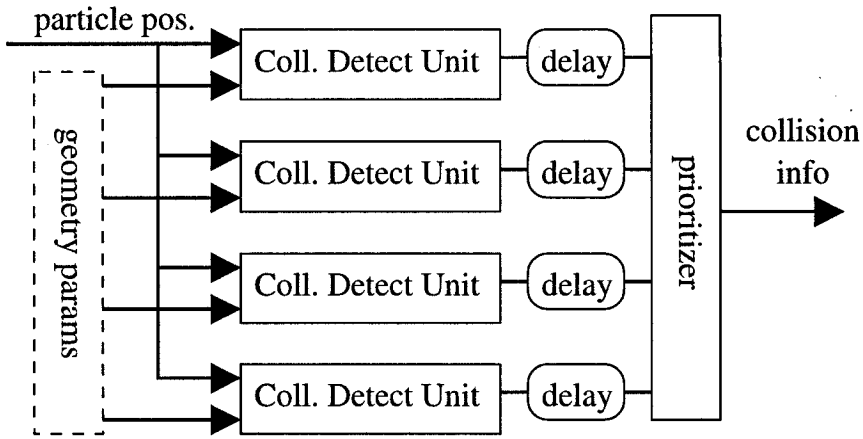

Figure 3.8: Collision detection system.

geometries easy. Inputs to the collision detection units include the particle position and a set of type–selected parameters, defining the collision geometry. Each collision detection unit output contains a collision flag indicating whether or not a collision was in fact detected, an estimate for the point of intersection, a surface normal vector at the intersection point, and surface friction and bounce factors. In our example of the plane detection unit, parameter registers would define the exact orientation and location of the plane, and on what side of that plane should the particles collide. They also provide the surface properties, bounce and friction factors which will be used to respond to a detected collision. Each collision detection unit detects for a basic geometry. Collision detection for slightly more complex shapes can be achieved by approximating the desired shape with a hierarchy of several basic detection units.

Similar to the force system, Figure 3.8 shows the collision detection units in parallel. The collision detection unit outputs, collision information sets, are each delayed and synchronized. Finally, one set of collision information is selected from the detection unit outputs and passed forward to the collision response stage.

3.6.6 Collision Response

Figure 3.9 shows a simplification of the collision response system. If in fact there was a collision detected, and the collision flag received as part of the input colli-

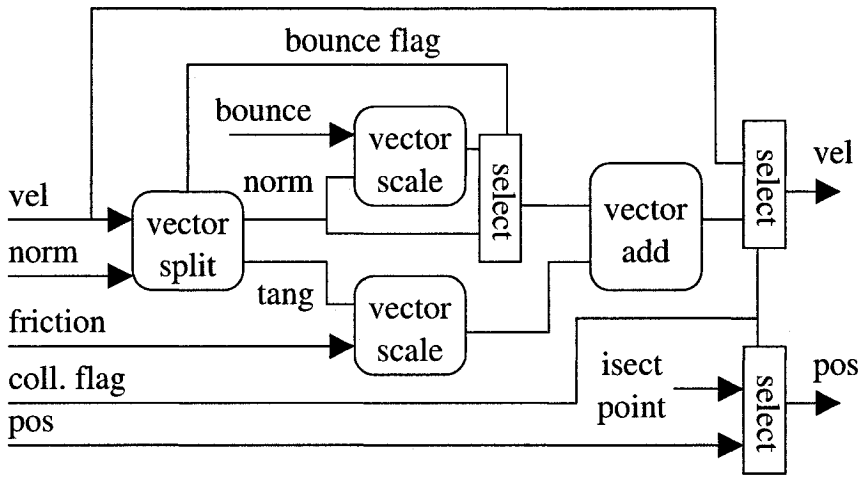

Figure 3.9: Simplified collision response system.

sion information was set, the collision response system will need to perform the following tasks:

1. Replace the particle position by the intersection point estimate.
2. Calculate the component of particle velocity tangent to the collision surface.
3. Calculate the component of particle velocity normal to the collision surface.
4. Scale the tangential particle velocity by the surface friction factor.
5. If the projection of the particle velocity on the surface normal is negative, the particle must be "bounced" off the surface by multiplying the normal particle velocity by the bounce factor.
6. Combine the updated normal and tangential particle velocity components to form a new total velocity vector for the particle.

The first operation in the collision response system, shown in Figure 3.9, breaks the particle velocity into a normal velocity vector and tangential velocity vector (relative to the surface) using the following relationships:

$$\begin{aligned} \vec{v}_{norm} &= (\vec{v} \cdot \hat{n}_{surface})\hat{n}_{surface} \\ \vec{v}_{tang} &= \vec{v} - \vec{v}_{norm} \ . \end{aligned} \tag{3.6}$$

First the dot–product of the particle velocity and surface normal is computed. The result of the dot–product is then used to scale a delayed surface normal vector, which produces the normal velocity component in vector form. Then, the result of the scale, the normal velocity vector, is subtracted from a delayed version of the original particle velocity, producing the tangential velocity vector.

During the second section of collision response, the tangential velocity vector is scaled by the friction factor, and in parallel, the normal velocity vector is scaled by the bounce factor:

$$\begin{aligned} \vec{v}_{tang} &\leftarrow k_{friction}\vec{v}_{tang} \\ \vec{v}_{norm} &\leftarrow k_{bounce}\vec{v}_{norm} \ . \end{aligned} \tag{3.7}$$

The surface friction factor would usually be a fixed–point number between zero and one, while the surface bounce factor should be a negative number to create the bounce, a reversing of the normal velocity component.

Recall that the dot product of the original particle velocity vector and the surface normal has already been computed during the first section of collision response. The sign bit of that result is delayed so that it can be used here to determine whether or not the particle should be bounced. If the sign bit is set, the bounced normal velocity just calculated is selected, otherwise, a delayed version of the original normal velocity is selected.

Next the selected normal velocity, either bounced or not bounced, is combined with the scaled tangential velocity, to create a new total velocity vector which is the proper response to a potential collision. Finally, shown at the end of the collision response stage in Figure 3.9, if the collision flag was set indicating that there was in fact a collision, the particle velocity is replaced with this new collided velocity, and the particle position is replaced with the intersection position estimate, otherwise, the original position and velocity is used.

3.6.7 Rendering

A simplification of the pipe's rendering system is shown in Figure 3.10. Rendering is the last stage of the Particle Pipe. At this stage, particle data has been completely

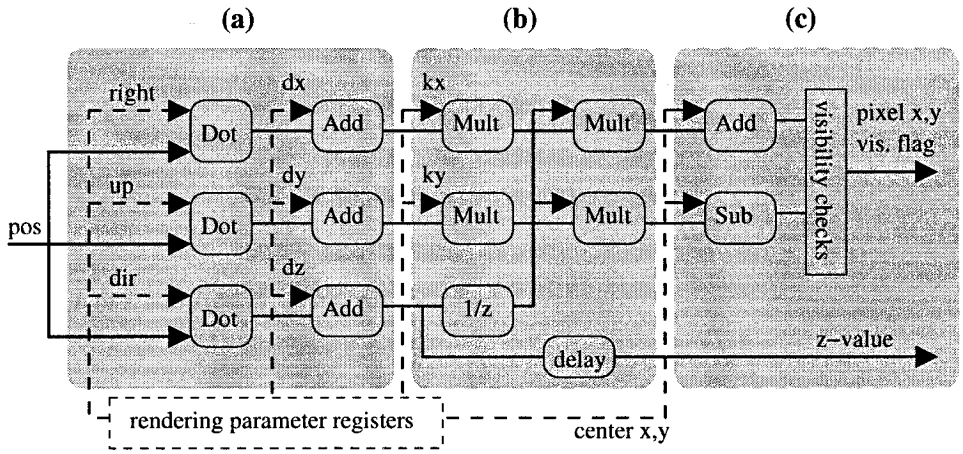

Figure 3.10: Simplified rendering system.

updated. Graphical information is calculated using the newly updated particle data together with rendering parameters, initialized and updated continuously by the particle controller. The graphical information calculated and output for each particle includes:

- a visibility flag,
- screen pixel coordinates,
- a frame buffer address or index,
- a color value,
- a *z–buffer* depth value.

As shown in Figure 3.1, after the rendering stage, the rendered graphical information and particle data are output together from the pipe. Particle data will be set back to particle memory, and rendering results will be send to the graphics system for display.

The first task in rendering is to find a set of *view coordinates* for the particles. Simulations exists in *world space*, and the particle position vectors are in world space coordinates. In Figure 3.10a, To convert particle position coordinates from

world space to view space the following transform is applied:

$$\begin{aligned} x_{view} &= \vec{r}_{world} \cdot \vec{right} + d_x \\ y_{view} &= \vec{r}_{world} \cdot \vec{up} + d_y \\ z_{view} &= \vec{r}_{world} \cdot \vec{dir} + d_z \ . \end{aligned} \tag{3.8}$$

where $\vec{right}$, $\vec{up}$, and $\vec{dir}$ are three vectors defining the "camera's" orientation in world space coordinates. The vector $\vec{d}$ represents the location of the world origin in view space coordinates. These values are held in parameter registers which software uses to control the view throughout a simulation.

After having obtained view space coordinates for the particle position, the view space position vector needs to be projected onto a 2–D surface, the viewing screen. This is accomplished in Figure 3.10b using the following relationships:

$$\begin{aligned} x_{screen} &= k_{x-scale}\, x_{view} \frac{1}{z_{view}} \\ y_{screen} &= k_{y-scale}\, y_{view} \frac{1}{z_{view}} \ . \end{aligned} \tag{3.9}$$

The previous two formulas provide coordinates relative to the center of the screen. Generally, pixel coordinates are most conveniently specified relative to the top left corner of the screen with positive values going down. In Figure 3.10c the following formulas are used to transform the screen coordinates into more useful pixel coordinates:

$$\begin{aligned} x_{pixel} &= x_{screen} + x_{center} \\ y_{pixel} &= y_{center} - y_{screen} \end{aligned} \tag{3.10}$$

Finally, the visibility of the particle is determined by comparing the view space z value to a minimum value, and checking that x_{pixel} and y_{pixel} are within the valid range.

Chapter 4

System Implementation

In order to provide proof–of–concept verification of the Particle Pipe, to test and verify that the Particle Pipe implemented is indeed functioning and capable of creating real particle graphics effects, a fully functioning test system was made. This test system, shown in Figure 4.1, is a working Particle Pipe system implementation capable of creating and testing simple, low–performance particle graphics systems. The following section will discuss in detail the design and implementation of that Particle Pipe test system, how it performed, what was learned from it, and how it could potentially be improved to create a high–performance particle graphics system.

4.1 Hardware Library, Configuration, and Simulation

The Particle Pipe together with its supporting hardware logic was designed as a modular, parametrized, and configurable VHDL hardware library, with scripts for automatic generation and configuration of particular Particle Pipe design instances. This is an important feature because reconfigurable hardware for particle graphics needs to be used to create many custom hardware accelerators, each specifically designed and customized to implement a particular set of effects. Data formats, par-

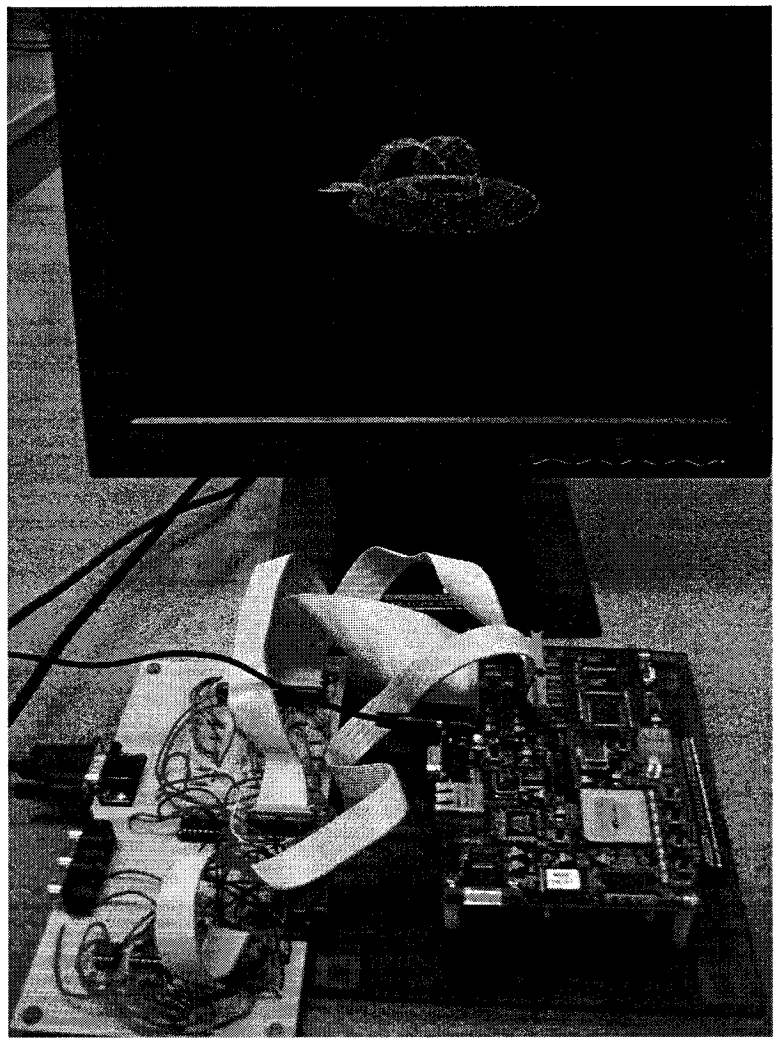

Figure 4.1: Particle Pipe test system in action.

ticle data set members, functional units included at different stages of the pipe and their functionality can all be configured automatically prior to FPGA synthesis by scripts. These generation scripts, in addition to creating the desired hardware configuration, also generate the necessary software header files providing the Particle Pipe controller with a software interface to Particle Pipe hardware and bus systems. Of particular importance is the microcontroller's interface to the pipe parameter bus system, by which the controller sets and controls the pipe parameter tables throughout simulations. These tables, and the parameter bus interface are highly dependent on and specific to the particular pipe configuration created, and therefore need to be automatically generated by the same configuration system.

Figure 4.2: Particle graphics simulation screen shot.

A functional 'C' software model was created in parallel with the development of the Particle Pipe VHDL hardware design. It is a bit accurate, functionally equivalent software model of the Particle Pipe, that can be used as a numerical and graphical test bench for testing and verifying Particle Pipe configurations, designing and verifying new hardware units, and developing and experimenting with new effects, graphically displaying interactive particle simulations such as the one shown in Figure 4.2.

This structured design approach allows flexible and efficient customization and extension of Particle Pipe design instances without extensive knowledge or exhaustive understanding of the Particle Pipe. In this way, Particle Pipe systems can be designed and modified in much the same way that traditional software design can be done using the framework and functionality of existing works and libraries.

4.2 System Configuration

The first step in implementing the Particle Pipe system was to generate a Particle Pipe configuration. The original goal was to create a particle graphics system ca-

pable of recreating an effect similar to the water fountain generated by software written at the beginning, when experimenting and learning about particle graphics effects. This is a relatively simple effect, shown generated by the simulator in Figure 4.2, and generated by the actual Particle Pipe test system in Figure 4.1. A set of Particle Pipe function units and other configuration options units was chosen. The software pipe simulator was then used to test and experiment with that configuration, verifying that it was capable of generating the desired effects. Some of the pipe configuration options selected were:

- Force
  - 3 uniform force vectors (for gravity and wind)
  - viscosity force unit
  - force vector summation
- Motion
  - massless force–to–acceleration
  - Euler motion integrator
- Updates
  - life decrementing
  - particle death (deactivation) at zero life
  - color fading
- Collision
  - 2 planar collision detection units
  - collision prioritizer
  - collision resolution unit with friction and bounce
- Rendering

- 8–bit color values
- 18–bit depth values
- world–to–view transform
- 3–D to 2–D projection
- pixel coordinate conversion
- video and depth buffer address generation

An 18–bit fixed–point format was chosen for the Particle Pipe's internal fixed–point number representation. The was done to best utilize the Stratix FPGA's hard-wired DSP units, which can be used for either 56, 18x18 multipliers circuits or 14, 36x36 multiplier circuits.

The particle data sets were configured to include the following fields:

- velocity vector: 3, 18–bit fixed–point values,
- position vector: 3, 18–bit fixed–point values,
- color: 8 bits
- life: 11 bits,
- type: 1 bit.

With these chosen formats, a packed particle data set is 128 bits, which can be read from or written to particle memory in exactly 4 accesses to a 32–bit wide RAM device.

All fixed–point numbers in the Particle Pipe, as well as in the parameter table system will then use the same 18–bit format. Generation and configuration scripts create a Particle Pipe hardware design configured to the specifications, also generating a pipe parameter table and bus system specific to the chosen data formats and pipe configuration. Finally, source code is generated or configured, defining the software interfaces used by the Nios microcontroller to interface to the pipe and its parameter table system, particle loading and storing hardware, graphics data output buffer, and particle memory for the initialization of new particles.

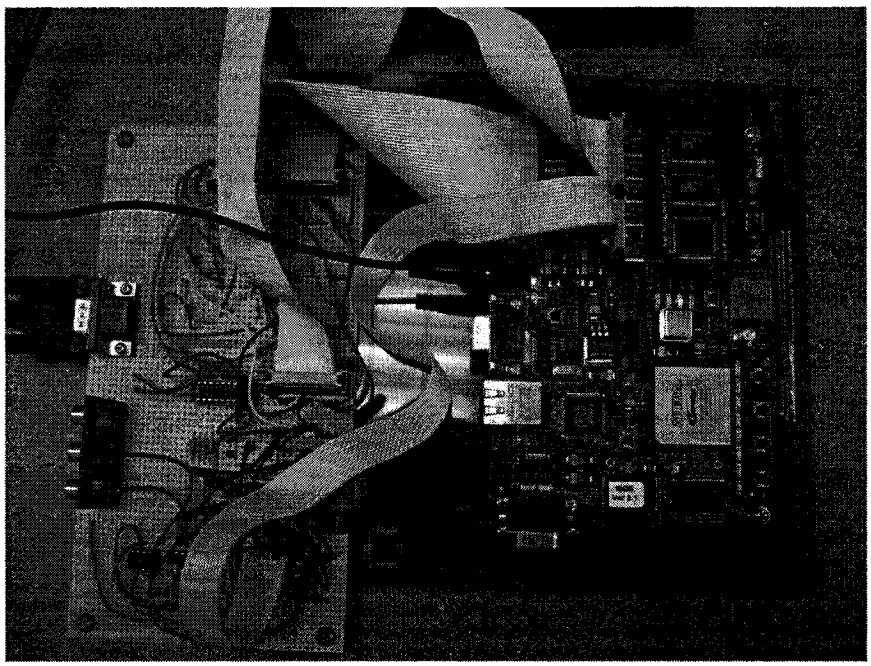

Figure 4.3: Nios FPGA development board with custom VGA interface.

4.3 Development Board

To implement the Particle Pipe test system, the Altera Nios Development Kit, featuring the Stratix EP1S40 FPGA device, was used as a development platform. The FPGA has 41,250 logic elements, 44,860 registers, 56 hardwired 18x18 multiplier cores, and about 3.4 million memory bits [13]. The FPGA is connected to 1 megabyte of 32–bit wide SRAM, and has separate, independent access to 16 megabytes of 32–bit wide SDR SDRAM. The board also has 8 megabytes of FLASH, Ethernet hardware, and two serial ports. To implement microprocessor systems with graphical output capability, the board was enhanced with custom video output hardware, as shown in Figure 4.3.

4.4 Soft–Core Microcontrollers

In recent years FPGAs have become dense enough and fast enough to allow the use of reconfigurable microprocessor systems as components in FPGA chip designs. An on–chip microcontroller can be a flexible and powerful tool for chip and sys-

tem design, especially for the testing and verification of new hardware designs by "systems–on-chip". Some soft–core microcontrollers are open–source, and have been designed to be FPGA–independent, even synthesizable as ASICs. Others, namely systems offered by the manufacturers of FPGAs, are specific to the FPGA technology they target, and are highly optimized for speed and area, using a minimum of FPGA logic and resources. FPGA soft–core microcontroller systems are generally very flexible. A designer is now able to quickly and easily use FPGAs to implement custom systems with special bus topologies, multiple processors, many types of advanced peripherals, and even make use of custom microprocessor instructions and co–processor interfaces, to test and support custom hardware designs. Soft–Core microcontrollers can be synthesized on some of the most inexpensive FPGAs, and reach frequencies as high as 150 MHz [14]. The Nios–II microcontroller has even been reported [14] to use as little as 600 logic elements of an Altera FPGA.

4.5 Nios Microcontroller System

Given a hardware accelerator such as the Particle Pipe and all of its supporting hardware, there is a need for a programmable, flexible, and intelligent control system to manage and coordinate the system as a whole, to perform software tasks such as initialization, implement procedural tasks, move data around, communicate with a host, and perform testing and other non–critical tasks which do not merit hardware designs. The Particle Pipe test system, summarized in Figure 4.4, is a fully functioning, self–supporting system–on–chip, capable of testing itself, implementing simulations, processing and displaying graphics, and communicating with a host computer. It includes all major aspects of a complete, stand–alone system to support working proof–of–concept particle graphics demonstrations using the Particle Pipe.

The Nios soft–core microcontroller was used as the backbone, the *system controller*, in the test system. The Nios has flexible tools for designing reconfigurable 32–bit microcontroller systems with pipelined, multimaster bus architectures. The

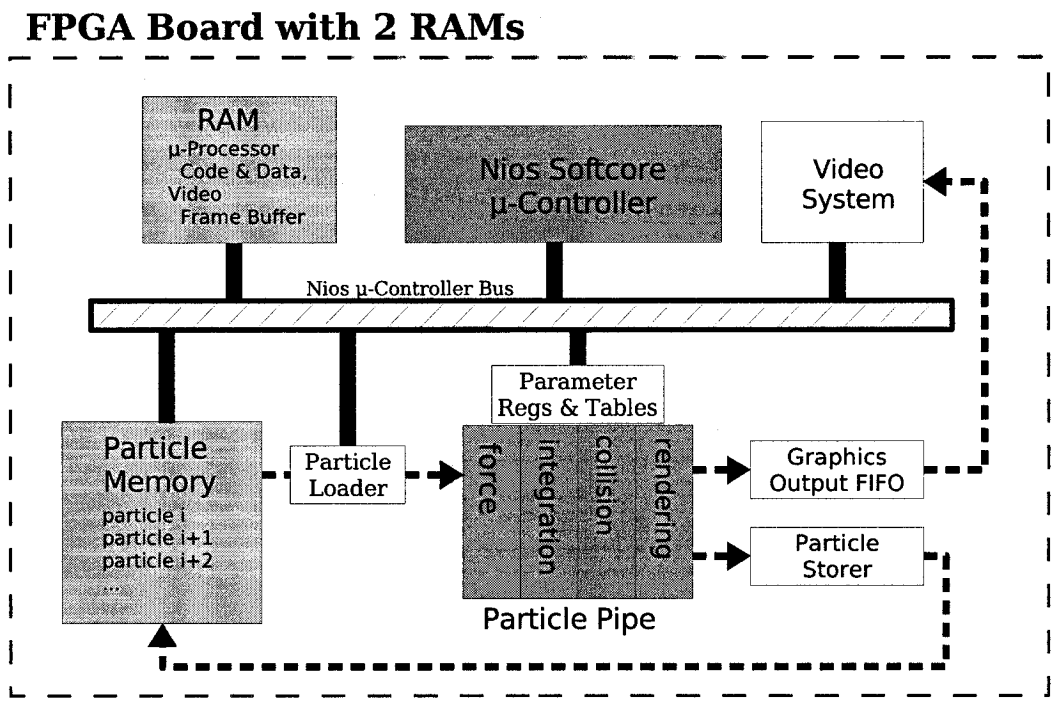

Figure 4.4: Particle Pipe test system architecture overview.

Nios bus system also has a feature key to a Particle Pipe system: simultaneous non–conflicting bus transfers.

Non–Conflicting bus transfers may take place simultaneously due to the slave–side arbitration scheme used in the the Nios' on–chip bus, shown in Figure 4.5. Nios bus systems designed with more than a single bus master present, are not implemented in the traditional bus architecture. Master–slave pairs are formed as needed by making individual bus connections between each bus master and all of its designated slave peripherals. When a bus slave peripheral can be accessed by, or is a slave to more than a single master, access to that peripheral is arbitrated by bus logic at the peripheral. In this way, multiple bus transfers between non–conflicting master–slave pairs may occur simultaneously. This is an flexible way to implement the bus systems required by the Particle Pipe test system.

The development board used provides the FPGA with access to two separate RAM devices. One of those devices, the smaller SRAM is used as Nios system

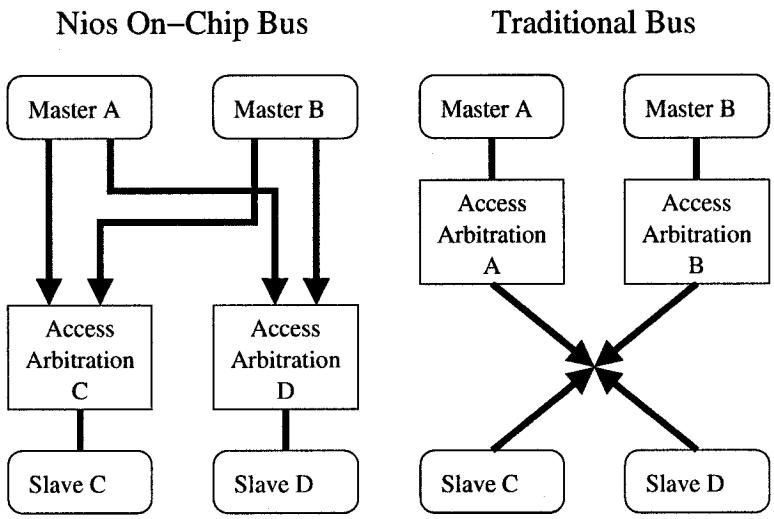

Figure 4.5: Slave side arbitration in the Nios' on–chip bus.

memory, holding microprocessor code and program data. A section of it is reserved for use as a video frame buffer, accessed by both the Nios microcontroller and the video controller. With the use of two separate on–chip bus systems and two independent RAM devices, shown in Figure 4.4, transfers on the microcontroller bus system will not conflict or compete with streaming particle data on the particle data bus between particle memory and the Particle Pipe. The Nios microcontroller does, however, also have access to the particle data bus. Between simulation passes when there is no streaming particle data, the Nios microcontroller will access particle memory to initialize new particles.

Looking at Figure 4.4, one can see that the Nios has access to its own microcontroller bus system, the particle data bus, and the pipe parameter bus. The Nios' own microcontroller bus allows it to access the RAM used for microprocessor code, program data, and a video frame buffer. It also allows the microprocessor to configure and control the various hardware units supporting the pipeline through each unit's configuration and control slave interface. These units include the particle loading and storing hardware, and the pipe's output buffers, and the video system's configuration port. The Nios has access to particle memory for initialization of new particles and testing purposes. The pipe parameter bus allows it to access the pipe's parameter tables during simulations and initialization. Finally, the video controller

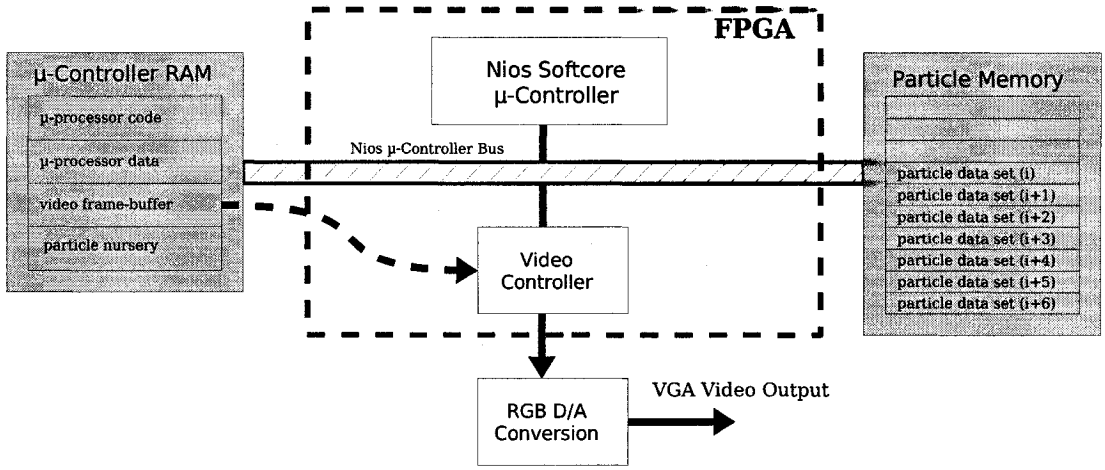

Figure 4.6: Video enhanced Nios soft–core microcontroller system.

is also a master of the microcontroller bus, giving it access to the video frame buffer located in microprocessor data memory.

The Nios microcontroller communicates to a host computer, downloading programs and reporting back system information. A boot–loader and default demonstration program are stored in the on–board FLASH device.

4.6 Video System

A VGA video system, shown in Figure 4.6, was made for the Nios microcontroller system and board. First, a video–speed triple digital–to–analog converter was made for generating and driving the analog RGB (red, green, and blue) signals for the VGA monitor. In a first attempt, a video triple–DAC IC was used, but analog signal problems such as "cross–talk" and "ground–bounce" caused by the nature of the connecting wires caused false latching of input values at the DAC. This was replaced by three passive DACs made from resistor networks which worked quited well.

In order to form a proper VGA video signal based on the contents of a frame buffer in RAM, an on–chip video controller peripheral was designed and added to the Nios microcontroller system. The video controller is a bus master which pre–

fetches data from a frame buffer in the Nios data RAM, and converts that data into a properly timed stream of digital RGB values, which it sends out of the FPGA to the analog–to–digital conversion hardware, along with horizontal and vertical syncing pulses. This process is illustrated in Figure 4.6. To create an image on the VGA monitor, the Nios has only to allocate a portion of its data RAM for use as a frame buffer, initialize the video controller peripheral, telling it where to find the frame buffer and what its dimensions are, then fill that frame buffer with the image to be displayed. The Nios can use a double–buffered frame buffer scheme by allocating memory from its data RAM for two frame buffers, and continuously swapping them from frame to frame. In such as double–buffered video scheme, the Nios would always configure the video hardware to stream video data from one buffer while writing pixel data to the other, swapping buffers upon completion of each frame.

4.7 Parameter Bus System

As introduced in Section 3.5, the functionality of each Particle Pipe function unit, its operation on the particle data sets at each latch stage of the pipe, is controlled by a set of parameter registers, containing values selected according to each particle's type field. This selection of parameter values based on particle type occurs in the parameter selection stages preceding each module, section, or function unit of the pipeline. Each such parameter selection stage uses the particle type as an index to a parameter table, holding the parameter values for each particle type. These parameters must be initialized by the Particle Pipe controller at the beginning of a simulation, and modified between each simulation frame. One simulation frame, of course, corresponds to one complete pass of all particle data through the pipeline. The values which are used to initialize these tables, and how they are updated and modified throughout a simulation, determines the operation of the Particle Pipe, which together with the initialization properties of new particles, determines the resulting simulations and effects.

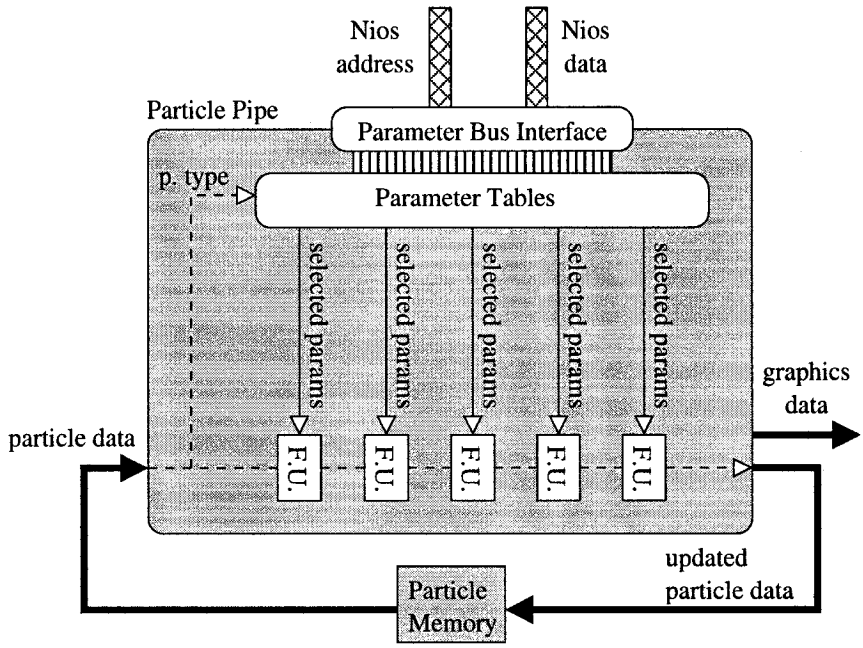

Figure 4.7: Parameter tables and bus system.

How then does the controller set and update these values? The parameter bus interface, shown in Figure 4.7, is the microprocessor interface which gives the controller access to all the parameter tables in the Particle Pipe. All the parameter tables of the Particle Pipe are connected by a special bus system, the pipe parameter bus, allowing them all to be read from and written to. The pipe parameter bus is mapped, through the parameter bus interface, to a reserved section of the Nios microprocessor's address space. The address mappings and data structures used by the parameter tables is completely determined by the pipe configuration details, the contents and structure of the Particle Pipe instantiated. That is why it is necessary for the parameter bus itself, as well as the software interfaces used by the Nios microcontroller, to be generated by the pipe's configuration scripts. By using the addresses and data structures defined in software header files generated by the pipe configuration scripts, microprocessor code running on the Nios microcontroller can read from and write to the Particle Pipe parameter tables, initializing them and changing their contents between simulation frames to create the desired simulation behavior and effects. The mapping of the parameter bus onto the Nios bus does not create bus competition or conflict with other data transfers in the system such as

movement of graphical data to and from the frame buffer, or most importantly, the flow of particle data to and from particle memory.

4.8 Particle Data Flow

The flow of particle data represents the biggest challenge faced when implementing a Particle Pipe system. The Particle Pipe is designed to be capable of receiving one input particle data set and outputting one updated particle data set together with its graphical data upon every clock cycle. In the case of our test system implementation, data formats were chosen such that each particle data set fits into exactly 128 bits, or 4, 32–bit words. For each particle that goes through the pipeline, 4, 32–bit words of particle data must be read from and written to particle memory. To help achieve the highest possible throughput of particle data, it is necessary to:

- Use special hardware for the loading, storing, and transfer of particle data,
- Store particle data in a dedicated RAM device, isolated from other bus transactions,
- Transfer particle data using an isolated and dedicated particle data bus,
- Attempt to fully exploit bus and RAM device high–throughput bursting modes by designing buffered hardware which accesses particle memory in bursts of sequential accesses.

Figures 4.4 and 4.8 show how the particle data flows through the particle test system in its own data path. It is streamed from particle memory into the pipe by the particle loading hardware. From the pipe's output it enters the pipe output FIFO buffers, from where it will be written back to particle memory by the particle storing hardware. This whole process of streaming data through the Particle Pipe, occurs independently and in isolation, without direct participation by the system microcontroller and its bus. Therefore, it occurs without contending or competing with

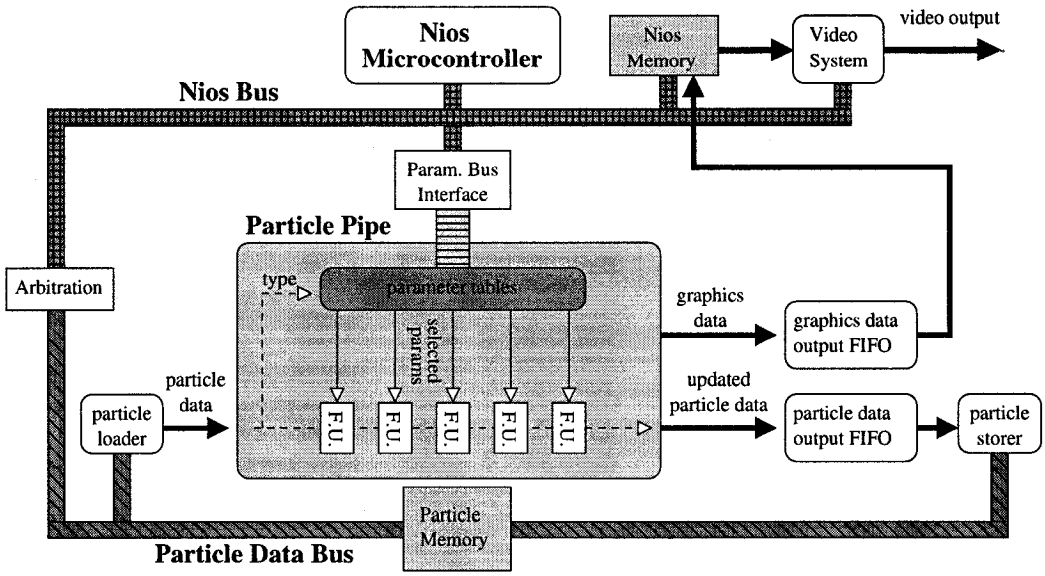

Figure 4.8: Test system architecture overview.

the bus traffic of the microcontroller's video system, particle initialization system, or parameter bus system.

If the system microcontroller were used to load particle data, feed the pipe with an input particle stream, and store the output particle stream back to memory, the throughput of the Particle Pipe would be severely limited. Instead, a custom hardware loading unit streams data out of particle memory from a specific address range specified during initialization, and feeds the data to the input end of the Particle Pipe at a specified rate. Since the width of the particle data sets, and therefore the width of the pipe's input port, is 128 bits or four, 32–bit words, the particle loader will fetch all four words of one particle then apply them together to the pipe's input port forming a valid input cycle. When the loading hardware is not capable of providing the pipe with particle data at its throughput rate of one data set per pipe clock cycle, or when it is necessary to decrease the rate of input data flow to avoid overflowing the pipe's output buffers, invalid "blank entries" are sent through the pipe, effectively stalling it without interfering with the processing of the valid data stream. The particle loader itself receives commands from, and is configured and controlled by the Nios microprocessor through its command and configuration port on the Nios' bus. The Nios controller will specify from what range of particle memory

to stream the particle data, and at what rate it should be streamed. It is necessary for the Nios controller to control and adjust the rate at which the input particle data is streamed to the pipeline to prevent the pipe from "flooding" the particle storer, outputting updated particle data at a rate faster than it can be stored back to particle memory.

As the particle data moves through the pipe, updated particle data together with graphics data for visible particles will stream out of the output end of the pipe. This output data must be buffered, before it can be handled properly. The particle data and graphics data need to be buffered separately because they are processed separately at difference rates and by different mechanisms. Therefore, as shown in Figure 4.8, the output end of the Particle Pipe feeds into two FIFO buffers, one for the particle data and one for the graphics data. These pipe output buffers contain logic and signals for tracking information about number of "valid" and "active" particles output from the pipeline, and making this information available to the Nios microcontroller over its control bus. The state of these FIFOs can be accessed by both the particle storing hardware to control the rate of streaming data though the pipe, and the Nios microcontroller.

Only when a particle in the pipeline is determined to be visible in the pipe's rendering stage, will valid graphics data be output from the pipe together with particle data. For "invalid" or stalled cycles, inactive particles, and particles that were determined to be invisible by the rendering hardware, the graphics data output is flagged as invisible and will not enter the graphics data output buffer. Similarly, during "invalid" or stalled cycles, or when particles have an inactive state, they will not enter the particle data output buffer. Each buffer keeps track of the number of valid cycles or data sets that were presented at its input, the number of data sets that have been read out of the FIFO, the number of data sets currently in the FIFO, and its empty or full status. This information and these signals are used by the particle storer and the Nios controller to determine how and when to processes data from the FIFOs, as well as how to control the particle data input rate.

The Particle storing hardware continuously writes data from the pipe's particle data output buffer, back to particle memory in the address range specified by the Nios microcontroller during initialization. Since inactive particles are not stored back to particle memory, as all the valid and active particles are stored back sequentially to the same block of particle memory from which they were read, the spaces or entries in particle memory occupied by inactive particles will be shifted to the top of particle memory. At the end of each simulation pass, by knowing how many "active" particle data sets were stored back to particle memory, the microcontroller knows at what address the "inactive" and available space begins and how large it is. This block of available, inactive space at the top of particle memory, can be used between frames for the initialization and injection of new particles. Any remaining space in this range, which is not written over with newly initialized particle data, must be cleared to an inactive state as it still contains data from particles left over from the previous simulation frame. This processes is illustrated in Figure 4.10.

4.9 Graphics Output Display

The Particle Pipe outputs graphical data for visible and valid particles during the streaming of particle data. The graphics output for each visible particle will include pixel coordinates ("x" and "y" coordinates for the frame buffer), as well as a color value and a z–buffer depth value for each pixel. This rendering information enters the graphics data output buffer, just as particle data enters the particle data output buffer. The Nios microcontroller can check the state of the graphics output buffer, and when necessary, may read and display graphical data from it. The output of the graphics output buffer has special circuitry to allow the microcontroller to read data from it in a number of forms allowing more efficient processing and rendering of the simulation graphics. It may read a hardware–formed frame buffer index or address directly, together with color and depth values. The Nios reads these pixels as they become available, output from the pipe, and writes them to the video frame buffer in Nios microcontroller memory. Eventually, when the frame is complete,

they will be read out by the video controller, converted to a timed video signal and displayed on the monitor, as in Figure 4.6.

4.10 System Operation and Control Flow

Demonstration code running on the test system's microcontroller, uses the Particle Pipe to make a simple particle graphics effect resembling a fountain of water, shown in Figure 4.1. Here we describe in detail the interaction and operation of the different system components, how they work together, and how the system microcontroller synchronizes and controls the system to create and display a real particle graphics simulation. The general operation of the system is summarized in Figure 4.9.

In preparation, the microcontroller must reset and initialize the video system. After reseting the video hardware, the Nios allocates three contiguous blocks of memory from its data RAM area, each large enough to hold a frame buffer in the form of one color value per pixel. In order to increase the efficiency of our simple microcontroller video system, we decrease the amount of video traffic on the bus, as well as the size of the frame buffers that must be cleared to black between each frame. This is done by commanding the video controller to enter a mode which only displays a 256 by 256 pixel area in the center of the standard 640 by 480 pixel VGA video stream. The system can then use much smaller 64 kilobyte frame buffers, which are more easily managed by the 80 MHz microcontroller bus system. Two of the three frame buffers allocated will be used to implement a double–buffered video scheme. The two buffers will be swapped between frames such that during the preparation of each frame, one frame is written to by the microcontroller, while the other is read from by the video controller for video content. After frame preparation has completed, the microcontroller sends a command to the video controller, giving it a new frame buffer address to read from, pointing to the newly prepared buffer. The third buffer is used to hold a depth buffer, or "Z–buffer", storing depth values for each pixel in the preparation buffer. When a new pixel or particle is to be written

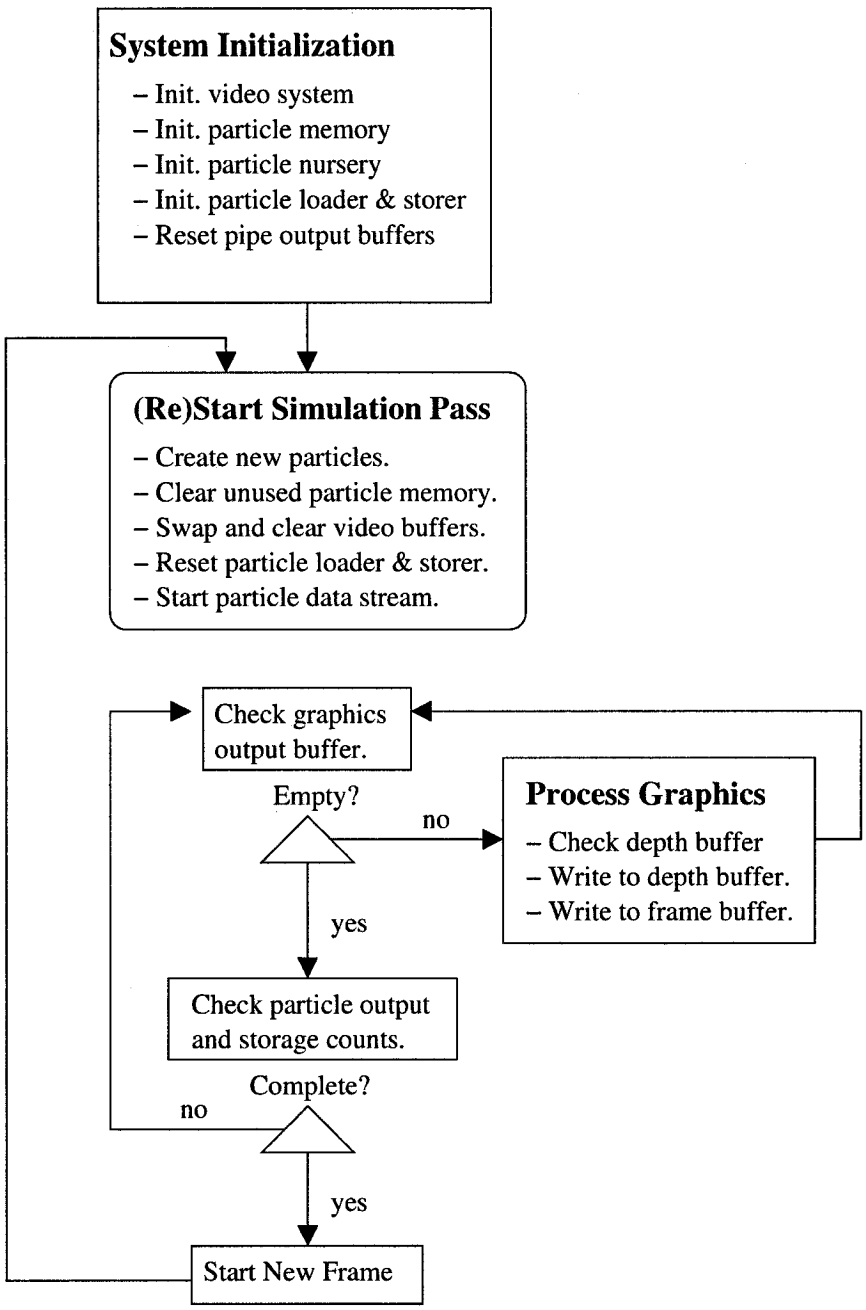

Figure 4.9: System operation and flow.

to the frame buffer, its depth value is compared against the depth value already present at that location. Only if the new pixel's depth value is less than the existing depth value will it be drawn. If it is to be drawn, its color value will be written to the preparation color buffer, and depth written to the depth buffer.

To reset and initialize the Particle Pipe, the pipe output FIFOs must be emptied and the pipe parameter tables must be initialized. The pipe parameter tables are

mapped to a dedicated region of the microcontroller's address space by the pipe parameter bus interface, and can be read from and written to directly. The data structures, and tables of data structures which make up the Particle Pipe's parameter tables, as well as how these parameter table structures map to the microcontroller's memory are defined in the microcontroller's software header files, together with the routines that software executing on the microcontroller will use to access them. These header files, and the interfaces, structures, and mappings they contain were generated by the pipe configuration scripts, together with the hardware they correspond to. The pipe parameter tables are filled with parameter values that determine the simulation properties and the pipe's operation on the different particle types in the desired simulations.

The particle pool, the collection of all particle data sets in particle memory, needs to be initialized before the simulation. All particles in the pool should be cleared to an inactive state, which will be recognized by the pipe as "dead" or inactive. Inactive particles are not updated or rendered by the pipe, nor do they enter the particle data output buffer to be stored back to particle memory. The spaces, or entries these inactive particles occupy are available for use by new particles as they are emitted. It suffices for the microcontroller to simply zero out the particle memory, because this will ensure that the "life" field in each packed particle data set will have the inactive value of zero.

Before the simulation begins, in preparation for the initialization and injection of new particles, which will occur between frames throughout the life of a simulation, the microcontroller prepares a collection of initialized and packed particle data sets. This collection of prepared new particles, the particle nursery, is shown in Figure 4.10. The fields of these particles are initialized with random values conforming to the specified stochastic properties of each field, for each particle type. All fields are assigned random values with the desired mean and standard deviation. These values are then packed together to the format used in the Particle Pipe and particle memory.

The loading and storing hardware, hardware controllers responsible for stream-

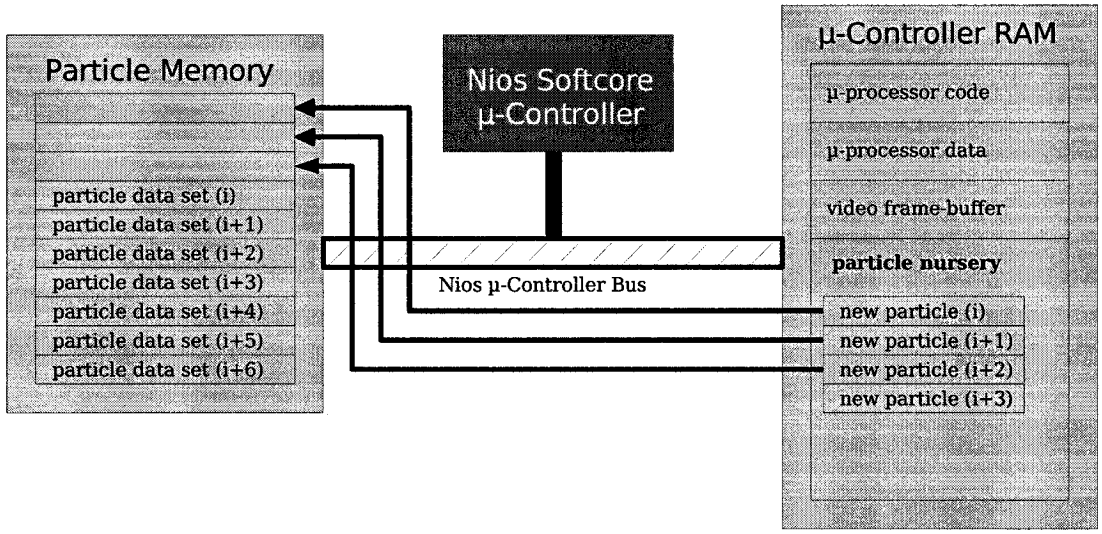

Figure 4.10: New particle initialization using the particle nursery.

ing data from particle memory, through the pipe, and back into particle memory, are initialized. The particle loader and storer need to be set with values specifying the address and size of the particle pool. The loader must also be set with a delay value, for insuring that the pipe input does not overrun the pipe output. Also, counters in the storer and the particle output buffers need to be reset. These are the counters that track the number of valid and inactive data sets seen at the output of the pipe.

Finally to begin the simulation, the microcontroller enables the video controller and commands the particle loader to begin streaming particle data. The particle loader will read data sets from the specified range of particle memory until complete, feeding the packed particles to the pipe's input. As the updated particle data sets complete their processing and journey through the pipe, they are output from the other end. Each clock cycle, the pipe's output will be marked as "valid" if there is truly a particle data set completing, or "invalid" if there was no particle data set input to the pipe on the corresponding input cycle, effectively a "stall slot". Valid output particles will either be active or inactive, and active particles may also be visible, in which case they are accompanied by valid graphical data. There is hardware to track during each simulation pass, the number of valid outputs seen by the output buffers and the number of active particles stored back to particle memory

by the storer. The particle data output buffer is constantly read from and emptied by the storing hardware, which stores the active particles back to memory, from a specified starting address upward.

As the particle data is streaming during the simulation pass, visible particles will create graphics data that will accumulate in the graphics data output buffer. The Nios microcontroller continuously checks the state of that buffer, and processes any graphics data that it contains. To process the graphics data, entries of the buffer are read out by the microcontroller. For each entry, the microcontroller can read coordinates, a color value, a depth value, and a preformed frame buffer index or address. For each pixel or particle read from the buffer, the depth value is checked against the depth value at that pixel location in the depth buffer. If the depth value already present in the depth buffer is less than the pixel currently being processed, nothing else is done and that pixel is abandoned since it is obscured by the content already in the frame buffer. Otherwise, the depth buffer is updated with the pixel's depth value, and the color value is written to the preparation frame buffer.

The microcontroller continues checking, emptying, and processing the graphics output data, throughout the simulation pass. When it finds that the buffer is empty, it will check if the data pass is complete. This is done by reading the counter values tracking the number of valid output particles. When this number is equal to the size of the particle pool, or the number of particles that the loader was instructed to load, then all particle data has been loaded and processed by the pipe. If the particle data output buffer has already been emptied by the storer, then the data pass has truly completed.

Now that the particle data has been updated, newly inactivated or "dead" particles have created an empty region at the top of particle memory. Inactive particles output from the pipe, are counted but are not buffered or stored back to particle memory. This means that an entry in particle memory for each inactivated particle will float to the top, as all active particle data is shifted down in memory, replacing inactive entries. The inactive space at the top of particle memory is used for new particles, and any space remaining after the injection and initialization of new par-

ticles, must be cleared to the inactive state as it still contains data from other shifted particles. As illustrated in Figure 4.10, new particles are copied from the particle nursery in microcontroller data RAM, and written to the empty region in particle memory.

Finally, the simulation frame has completed, all particle data has been updated, graphical output data has been written to the preparation frame buffer, new particles have been created, and we are ready to swap the preparation and display frame buffers. The video controller is instructed to read from the frame buffer just prepared, displaying the graphical data from the simulation frame just completed, and the other frame buffer, together with the depth buffer are cleared.

The pipe's parameter tables can now be updated according to the desired simulation properties, perhaps changing the 3–D rendering view orientation. The particle loading and storing hardware, and the output buffers are reset. A new simulation frame begins as the particle data starts streaming once again.

Chapter 5

Results and Conclusions

5.1 Particle Test System Results

The Particle Pipe test system, is a complete working system, designed for the purpose of testing, verifying, and demonstrating the functionality and concepts of the Particle Pipe. Above all, as a working example implementation, it provides a functioning and interactive *proof–of–concept* for the use of pipelined FPGA hardware designs to accelerate and enhance particle graphics techniques in real–time applications.

The Particle Pipe test system was implemented using an Altera EP1S40 Stratix FPGA. Table 5.1 shows FPGA resource utilization of the fitted design in that device. The Particle Pipe alone was synthesized for that device to a maximum operating frequency of 130 MHz. The complete test system, including the Nios microcontroller, buses, controllers, and other components, operates at a system frequency of 80 MHz. Recalling that the Particle Pipe design is capable of updating and rendering one particle data set upon each pipe clock cycle, the Particle Pipe itself has a potential throughput of processing 130 million particles per second. This corresponds to simulations and effects with 2.1 million particles in each frame at a frame rate of 60 Hz, or 4.3 million particles per frame at 30 Hz.

These throughputs, of course, depend on a particle memory capable of providing the required access rates, and a system integration capable of processing and

	Logic Cells	Registers	18x18 Mult.
Avail. in FPGA	41,250	44,860	56
Total System Design	40,764	32,215	29
Nios μ-Controller	6,397	2,424	1
P. Pipe with I-Faces	32,962	29,156	28
P. Pipe	30,283	27,853	28
Force System	5,157	4,962	3
Total Collision System	15,417	14,577	12
Coll. Detection	2,472	2,301	0
Coll. Response	12,699	12,031	12
Integrate Motion	624	582	0
Rendering	8,382	7,034	13

Table 5.1: FPGA utilization of test system.

displaying the generated graphical data. Given a particle data set width of 128 bits, to achieve the pipe's maximum throughput, the read and write access rates required of particle memory are:

$$(128\ \frac{bits}{particle})(130\ MHz)(1\ \frac{particle}{cycle}) = 2.08\ \frac{gigabytes}{sec}\ . \tag{5.1}$$

This can be realized with standard PC-2100, PC-2700, and PC-3200 DDR-SDRAM memory modules, which respectively provide 2.133, 2.667, and 3.2 gigabytes per second of memory bandwidth [15]. Only 1.28 gigabytes per second are required to fully utilize the Particle Pipe in the 80 MHz test system. This would correspond to a peak performance of 1.3 million particle per frame at a frame rate of 60 Hz, or 2.6 million particles per frame at 30 Hz.

The video processing and display capabilities of the test system are very simple and low in performance compared to standard computer hardware. The inability of the test system as a whole to efficiently process and remove graphical data from the Particle Pipe's output buffer severely limits the speed and size of simulations implemented on the test system. As data is streamed from particle memory through the pipe and back into particle memory by the particle loading and storing hardware, the test system relies on the Nios microcontroller to process the graphics output buffer. Processing graphical data means integrating it into a display system. The

Memory Region	Write Access Rate	Copy Access Rate
Particle Memory	0.7235 bytes/clk	0.1365 bytes/clk
Graphics Memory - Video OFF	0.6142 bytes/clk	0.2806 bytes/clk
Graphics Memory - Video ON	0.6146 bytes/clk	0.2771 bytes/clk

Table 5.2: Memory throughputs observed.

test system's simple VGA display system does not have an efficient method for writing pixels to color and depth buffers, or clearing those buffers. Combined with low particle memory access rates, this limits performance to well below what the Particle Pipe is truly capable of.

The primary factor severely limiting the performance of the test system, and preventing utilization of the Particle Pipe at a performance closer to its potential, is the low access rate achieved to particle memory by the system. The particle loading and storing hardware is unable to load and store particle data from memory efficiently, and therefore starves the Particle Pipe, failing to provide it will a sufficient input stream, or empty its output buffers in time. Since the loading and storing hardware is unable to provide the pipe with a sufficient input stream, the pipe is stalled, processing null data for most of the time during test system operation.

Table 5.2 shows the results of performance tests which measured the system's ability to access blocks of memory in both the SDRAM device, used by the test system as particle memory, and the SRAM device, used by the test system for microcontroller and video memory. The performance tests were performed by using dedicated timing hardware to profile both software accesses to memory regions using the Nios microcontroller, and accesses made by special bus hardware created to generate continuous sequences of transfers to particle memory. The copy rate to particle memory, the type of access required by the Particle Pipe, has been limited to 0.1365 bytes per clock cycle. With this in mind, considering that the Particle Pipe at maximum throughput would require a copy rate of 128 bits per cycle, particle memory starvation is responsible for limiting Particle Pipe utilization to 0.85% of its true potential. During simulations on the test system, the Particle Pipe is actually stalled and not in use for 99.15% of all cycles.

The extremely low particle memory access rate achieved by the loading and storing hardware is the result of the simple design of the particle data system as a whole. As a first implementation of the Particle Pipe, and primarily intended for the testing and verification of the concept and operation of the Particle Pipe, the test system was designed and organized in the most simple and direct way possible. The particle data loading and storing units were designed as two separate controllers, each a master on the particle data bus, implemented using the Nios' Avalon on–chip bus system. The particle storing hardware blindly attempts to empty the particle data output buffer, while the loading hardware loads particle data at a specified rate. Coordination and arbitration between these two bus masters, attempting to simultaneously access particle memory is resolved by the Avalon bus arbitration logic. As the loading and storing units attempt to simultaneously read and write from the single particle memory device, access is shared between the two with alternating permission. The result of this simple solution is that what should be two sequential, pipe–lined, high–speed bursts, one read stream and one write stream, becomes alternating random accesses, by two different masters. The bus transfers are not pipelined, the SDRAM controller cannot reach a bursting mode of operation, and the access rates seen by the particle pipe are extremely slow.

A particle data system *could* be designed explicitly to maximize the performance of the particle memory, since as we have seen, access to particle memory is the major factor limiting Particle Pipe performance. For the design of a high–performance Particle Pipe system, The particle loading and storing hardware should be merged into a single design, capable of exclusively bursting sequential data from particle memory, while all pipe output is buffered for some duration. Then, the pipe output should be exclusively bursted back to particle memory, while the pipe is either not feed with input data, or feed from a read buffer. In order to maximize particle memory performance, it is absolutely necessary that the loading and storing units, as well as the interconnection or bus between with the memory controller and the pipe is pipelined. The transfer of data should be separated from the transfer of address and control information, to allow multiple simultaneous pending read oper-

Frame Rate	Video	Num. of P.	P. Data Pass Time	Total Frame Time
30 Hz	disabled	45,000 particles	98.14%	29.83 Hz
30 Hz	enabled	41,000 particles	89.04%	29.71 Hz
60 Hz	disabled	22,000 particles	96.09%	59.86 Hz
60 Hz	enabled	18,000 particles	77.72%	59.45 Hz

Table 5.3: Particle test system performance at 75 MHz.

ations. If this is done, then a Particle Pipe system could expect to have access rates to particle memory at the performance limits of either the memory device itself, or the clock frequency and width of the system bus.

The Particle Pipe test system provides a proof–of–concept demonstration of FPGA implemented particle graphics, and a platform for testing the functionality of the Particle Pipe design.

Table 5.3 contains results summarizing the actual performance and capabilities of the particle test system. The results shown are the actual performances observed from the test system running demonstration code, generating the effect shown in Figure 4.1, while targeting a given frame rate. The largest particle simulation implementable on the test system contains 45,000 particles, at a frame rate of 30 Hz. The video system, when enabled, reduced the simulation size to 41,000 particles.

5.2 High Performance Implementation

A true, high performance Particle Pipe system could be implemented using a platform such as the Altera PCI Development Kit, shown in Figure 5.1. This platform is a PCI card, featuring the EP1S60 Stratix FPGA, and 256 megabytes of DDR SDRAM. The EP1S60 FPGA has about 60,000 logic elements, which is 20,000 more logic elements than were available in the test system's EP1S40 FPGA. The 256 megabytes of PC333 DDR SDRAM can operate at 333 MHz (2x 166 MHz) providing 2.667 gigabytes per second of memory bandwidth. Recalling equation 5.1, we see that 2.667 gigabytes per second of memory bandwidth would provide more than enough particle memory access to support a 130 MHz Particle Pipe at

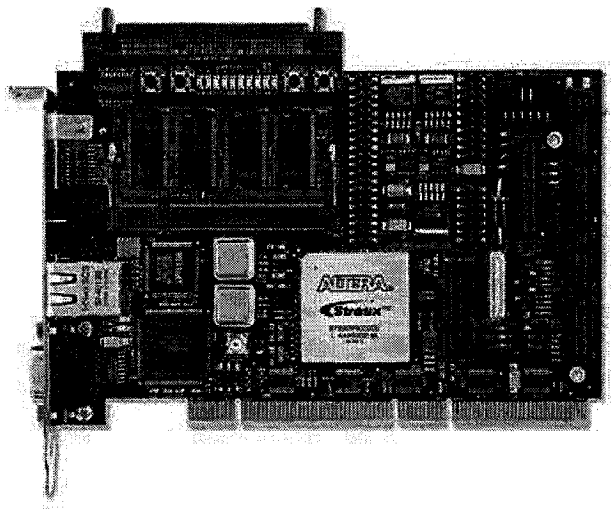

Figure 5.1: Stratix PCI high-speed development board (from [13]).

full throughput. In addition, a 32–bit PCI interface at 33 MHz can burst at 132 megabytes per second, and a 64–bit PCI interface at 64 MHz can burst at 528 megabytes per second [16]. The Particle Pipe to host connection using this platform would therefore be fast enough to handle the graphics output of the Particle Pipe.

Using this kind of a platform to implement a Particle Pipe system, the rendering section in the Particle Pipe FPGA hardware may not be necessary. The 3–D projection and rendering operations can be accomplished by the host computer's normal graphics hardware. This will make available much more of the FPGA's reconfigurable logic and resources to the actual particle simulation hardware. In Table 5.1, we see that about 20% of the reconfigurable logic was used on the rendering section of the pipeline. This would enable the implementable of much more interresting and complex pipelines and simulations.

5.3 Particle Pipe Operation Rate

A reasonable question to ask about Particle Pipe performance, when trying to compare it to normal software or GPU methods is, "What would be the equivalent performance of the system in terms of instructions–per–second, or operations–per–

Design Entity	Number in Particle Test System
Fixed–Point Add/Sub & Limit	39
Fixed–Point Multiply & Limit	28
Fixed–Point Divide	1
Fixed–Point 3–D Vector Add/Sub	5
Fixed–Point 3–D Vector Dot Product	4
Fixed–Point 3–D Vector Scale	4

Table 5.4: Instantiation counts of basic fixed–point operations.

second?" This question merits an explanation of a few major concepts. First of all, the Particle Pipe is application–specific hardware, that implements particle graphics simulations in a completely different way than would software running on a CPU or GPU. The concept of "instruction" does not apply at all, since the hardware does not accomplish tasks by executing sequences of instructions. It is a hardwired logic circuit that updates and produces data. The concept of a metric of the rate of instructions or operations possible is specific to the method or the implementation of a task. In fact, it is well know that the instruction execution rates of two CPUs can not at all be used as a metric of performance or comparison, even if the two CPU's are executing the same program compiled by the same compiler! The only way to compare different architectures is to compare their performance on specific tasks.

It is possible, however, to consider the fundamental operations which are being accomplished by a system for a particular problem. How to define the fundamental operations is questionable, especially when the task or problem, as in computer graphics, is not defined by a specific computation or solution, but is to create an effect of some kind, for which there may be many unrelated and individually unique solutions. However, in an effort to quantize how the Particle Pipe hardware is performing, what it is achieving, and what work it is doing, we examine in Table 5.4 the instantiation counts of the major fixed–point math operations in the Particle Pipe. From this, we can see that fundamentally, for every particle updated in each simulation frame there are about 68 *major* fixed–point math operations performed. Using these operations, and considering the maximum operation frequency of the Particle

Pipe, we find that the Particle Pipe has a potential operation rate of

$$(130\ MHz)(1\ \frac{particle}{cycle})(68\ \frac{ops}{particle}) = 8.84\ billion\ \frac{ops}{sec} \tag{5.2}$$

and

$$\frac{(130\ MHz)(1\ \frac{particle}{clk})(68\ \frac{ops}{particle})}{30\ \frac{frames}{sec}} = 294.6\ million\ \frac{ops}{frame}\ . \tag{5.3}$$

Given that the test system is capable of running simulations as large as 45,000 particles per frame at a frame rate of 30 Hz, we can conclude that the *average* operation rate of the test system's Particle Pipe is

$$(68\ \frac{ops}{particle})(45,000\ \frac{particles}{frame})(30\ \frac{frames}{sec}) = 91.8\ million\ \frac{ops}{sec} \tag{5.4}$$

and

$$(68\ \frac{ops}{particle})(45,000\ \frac{particles}{frame}) = 3.06\ million\ \frac{ops}{frame}\ . \tag{5.5}$$

5.4 Method Comparison

Particle graphics effects created in software, GPU methods, and those created by the hardware of a Particle Pipe, could be and are implemented in completely different ways, essentially doing and accomplishing different tasks, creating unique graphical simulations and effects. Although each is unique and implemented in a different way, they have common properties and goals universal to all particle graphics simulations. The only true way to compare different systems, each capable of implementing particle graphics effects in their own way, is to consider the following aspects:

- What is the largest possible particle simulation size, measured in particles per frame?
- The complexity, variety, and flexibility of implemented simulations.
- How do the size and complexity of the simulations effect a host application?

- How practical is the solution?

We have seen that the use of reconfigurable hardware to implement sets of specialized Particle Pipe systems has the capacity for extremely large simulations, far beyond what would be possible with traditional software methods. The performance of the particle test system was limited to relatively small and simple systems, but as discussed in Sections 5.1 and 5.2, if properly implemented using the right platform, the approach definitely has potential for extremely high throughput, of the order of several million particles per frame. These high throughputs, because implemented in an isolated FPGA subsystem, and not in main memory as are both software and GPU particle systems, could be achieved with a minimum burden on host system and application resources. Because each instantiation within a set of a Particle Pipe systems supporting a host application can be completely unique and specialized specifically for the set of effects it generates, reconfigurable hardware offers a great amount of flexibility not available to GPU methods. An FPGA Particle Pipe can be designed especially to implement a special set of relatively complex or eccentric effects. Although in some ways software offers an unequivocal amount of flexibility and possible complexity, as well as ease of design, FPGA design can achieve flexibility and complexity in a different way, while supporting high performance throughputs not achievable by software methods. In particular, if the FPGA design is approached with a "component library" methodology, making use of an existing framework, designing and using components of a library with some predefined structure, it can also be realistic in terms of both difficulty and flexibility for application designers. Furthermore, the complexity of software particle systems *is* limited by the unacceptable performance cost they impose on host applications. Real–time graphics applications running on todays standard computer hardware, simply cannot afford to implement large and complex particle systems [11]. Finally, while software and GPU methods offer practicality in that the required components are already present in general purpose computers, FPGAs and reconfigurable hardware may soon have such a presence. When FPGAs do have a general presence in computer hardware, and a standard interface emerges, then not only particle graphics,

but many new applications stand to benefit or possibly be revolutionized by the use of reconfigurable hardware accelerators.

5.5 Conclusions

In this thesis we have explored the idea of how reconfigurable hardware could be used to enhance and accelerate an application area such as particle graphics. Not just particle graphics, but many different kinds of new applications and tasks never before considered candidates of hardware acceleration could potentially use the presence of FPGA hardware in standard computers to be accelerated and enhanced in ways which are currently not possible or practical. This was done through the case study of the design and implementation of an FPGA particle graphics hardware system, a radical new approach to particle graphics. We have seen that this technique contains the possibility for achieving new tremendous performances, but that these performances will not be achieved without facing implementation challenges. Like all solutions to all problems, FPGA hardware pipelines have their limitations. We have seen that it is not trivial to design a system capable of providing and processing data at the rates required for that high performance, and if possible, will not occur without in turn causing its own performance or system costs. The fixed point format used in a Particle Pipe system is yet another challenge, as it could possibly limit the dynamic range of some simulations. However, if solutions to these challenges and limitations can be found, than FPGA particle graphics pipelines have the promise and potential for making possible what is otherwise not: real–time graphics applications for standard computers with multiple complex and customized particle simulations as large as 4.3 million particles per frame.

Appendix

The following tables show FPGA synthesis and fitting details of the Particle Pipe test system.

	Logic Cells	Registers	18x18 Mult.
Avail. in FPGA	41,250	44,860	56
Total Test System Chip Design	40,764	32,215	29
Nios μ-Controller System	6,397	2,424	1
Nios CPU and Caches	4,422	1590	1
Video System	451	256	0
Particle Pipe with Interfaces	32,962	29,156	28
Pipe Parameter Bus	1,470	852	0
Particle Pipe	30,283	27,853	28
Force System	5,157	4,962	3
Uniform Force	54	54	0
Viscosity Force	1,005	831	3
Total Collision System	15,417	14,577	12
Coll. Detection	2,472	2,301	0
AA-Planar Collision Detection	526	489	0
Coll. Response	12,699	12,031	12
Integrate Motion	624	582	0
Rendering	8,382	7,034	13
Screen Projection	3,756	3,021	4
Fixed–Point Inverter	2,072	1,541	0
World-to-View Transform	2,799	2,277	9
Static Update Unit	130	130	0
Pipe Data Interface	1,209	451	0
Particle Loader	476	175	0
Particle Storer	385	102	0
Particle Data Output FIFO	37	20	0
Graphics Data Output FIFO	37	20	0
Fixed–Point Multiply and Limit	196	145	1
Fixed–Point Divide	2,072	1,541	0
Fixed–Point Add/Subtract	91	84	0
Fixed–Point Vector Dot-Product	842	675	3
Fixed–Point Vector Add/Subtract	279	258	0
Fixed–Point Vector Scale	588	435	3

Table 5: FPGA utilization of test system by design entity.

	Logic Cells	Registers	18x18 Mult.
Avail. in FPGA	100.00%	100.00%	100.00%
Total Test System Chip Design	98.82%	71.81%	51.79%
Nios μ-Controller System	15.51%	5.40%	1.79%
Nios CPU and Caches	10.72%	3.54%	1.79%
Video System	1.09%	0.57%	0.00%
Particle Pipe with Interfaces	79.91%	64.99%	50.00%
Pipe Parameter Bus	3.56%	1.90%	0.00%
Particle Pipe	73.41%	62.09%	50.00%
Force System	12.50%	11.06%	5.36%
Uniform Force	0.13%	0.12%	0.00%
Viscosity Force	2.44%	1.85%	5.36%
Total Collision System	37.37%	32.49%	21.43%
Coll. Detection	5.99%	5.13%	0.00%
AA-Planar Collision Detection	1.28%	1.09%	0.00%
Coll. Response	30.79%	26.82%	21.43%
Integrate Motion	1.51%	1.30%	0.00%
Rendering	20.32%	15.68%	23.21%
Screen Projection	9.11%	6.73%	7.14%
Fixed–Point Inverter	5.02%	3.44%	0.00%
World-to-View Transform	6.79%	5.08%	16.07%
Static Update Unit	0.32%	0.29%	0.00%
Pipe Data Interface	2.93%	1.01%	0.00%
Particle Loader	1.15%	0.39%	0.00%
Particle Storer	0.93%	0.23%	0.00%
Particle Data Output FIFO	0.09%	0.04%	0.00%
Graphics Data Output FIFO	0.09%	0.04%	0.00%
Fixed–Point Multiply and Limit	0.48%	0.32%	1.79%
Fixed–Point Divide	5.02%	3.44%	0.00%
Fixed–Point Add/Subtract	0.22%	0.19%	0.00%
Fixed–Point Vector Dot-Product	2.04%	1.50%	5.36%
Fixed–Point Vector Add/Subtract	0.68%	0.58%	0.00%
Fixed–Point Vector Scale	1.43%	0.97%	5.36%

Table 6: Percent FPGA utilization of test system by design entity.

Bibliography

[1] John Sachs Beeckler and Warren J. Gross, "FPGA Particle Graphics Hardware," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, pp. 85-94, April 2005.

[2] William T. Reeves, "Particle Systems - A Technique for Modeling a Class of Fuzzy Objects," in *Computer Graphics*, 17:3, pp. 359-376, 1983.

[3] Kees van den Doel, Dave Knott, Dinesh K. Pei, "Simulation of Complex Audio–Visual Scenes," in *Presence: Teleoperators and Virtual Environments*, 13(1), pp. 99-111, February 2004.

[4] Meciej Matyka, "Inverse Dynamic Displacement Constraints in Real-Time Cloth and Soft-Body Models," in *Graphics Programming Methods*, pp. 81-91, 2003.

[5] John van der Burg, "Building an Advanced Particle System," in *Game Developer Magazine*, pp. 44-50, March 2000.

[6] Adam Herout and Pavel Zemcik, "Hardware Pipeline for Rendering Clouds of Circular Points," in *Proceedings of the International Conferences in Central Europe on Computer Graphics, Visualization and Computer Vision*, pp. 17-22, February 2005.

[7] Pavel Zemcik, Adam Herout, Ludek Crha, Otto Fucik, Pavel Tupec, "Particle Rendering Engine in DSP and FPGA," in *Proceedings of the Conference and Workshop on the Engineering of Computer-Based Systems*, pp. 361, 2004.

[8] Tommi Ilmonen, Janne Kontkanen, "The Second Order Particle System," in *Proceedings of the Conference and Workshop on the Engineering of Computer-Based Systems*, 11(2), pp. 240-247, 2003.

[9] Navid Azizi, Ian Kuon, Aaron Egier, Ahmad Arabiha, and Paul Chow, "Reconfigurable Molecular Dynamics Simulator," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 197-206, April 2004.

[10] Toshiyuki Fukushige, Makoto Taiji, Junichiro Makino, Toshikazu Ebisuzaki, and Daiichiro Sugimoto, "A highly parallelized special-purpose computer for many-body simulations with an arbitrary central force: Md-grape," *The Astrophysical Journal*, pp. 468-480, 1996.

[11] Lutz Latta, "Building a Million Particle System," in *Proceedings of the Game Developers Conference*, 2004.

[12] Dave Knott, Kees van den Doel, Dinesh K. Pai, "Particle System Collision Detection using Graphics Hardware," in *Proceedings of the SIGGRAPH 2003 Conference on Sketches and Applications*, July 2003.

[13] Altera Corporation, "Altera Stratix Device Family Data Sheet," 2005, http://www.altera.com/literature.

[14] Altera Corporation, "NIOS II Processor Reference Handbook," 2005, http://www.altera.com/literature.

[15] JEDEC, The Standards Resource for the Worlds Semiconductor Industry, "DDR SDRAM Specification," November 2005, http://www.jedec.org.

[16] PCI SIG, "Conventional PCI 3.0 and 3.3: An Evolution of the Conventional PCI Local Bus Specification," November 2005, http://www.pcisig.com/specifications/conventional/.