FPGA Particle Graphics Hardware

John Sachs Beeckler and Warren J. Gross

McGill University
Department of Electrical and Computer Engineering
Montreal, Québec Canada
{johnbeeckler, wjgross}@macs.ece.mcgill.ca

Abstract

Particle graphics simulations are well suited for modeling phenomena such as water, cloth, explosions, fire, smoke, and clouds. They are normal realized in software, as part of an interactive graphics application, such as a video game. Their use in such applications is limited by the computational burden and resource competition they create for a host application. We present the design of a Hardware Particle Machine, for implementation in an FPGA, intended for accelerating real-time particle graphics in applications such as video games. The Particle Machine is a system that completely contains, manages, and executes particle graphics simulations and rendering. The Particle Machine is a system comprised of particle memory, a controller, and the Particle Pipe, a pipelined Particle Update Processor. The Particle Pipe has been synthesized to 130 MHz, on an Altera Stratix FPGA, resulting in a potential throughput of 2.1 million PPF (particles per frame). This throughput is achieved with minimal load on application and main system performance.

1 Introduction

Particle graphics simulations are well suited for modeling phenomena such as water, cloth, explosions, fire, smoke, and clouds [1]. 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. 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. The visual ensemble of such a great group of particles, its behavior and evolution, appearance, interaction with an environment, and inherent random variation, exhibit a great degree of complexity and detail, properly resembling the detail and randomness in nature.

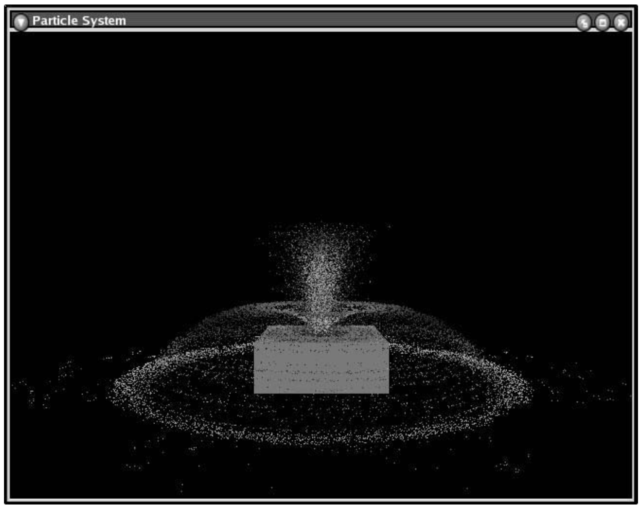

Figure 1. Particle Graphics Simulation

These graphical particle simulations are generally implemented in software, embedded in applications. The embedding of particle graphics in real-time, interactive applications, namely video games, presents a difficult challenge. The size, complexity, and overall use of particle graphics in video games are severely limited by the computational burden that they impose on a host application, competing for valuable resources.

This paper presents an new approach to creating real-time particle graphics: a Hardware Particle Machine. The Hardware Particle Machine is a coprocessor system which features a particle update processor, the Particle Pipe, implemented in reconfigurable logic.

Section 2 of this paper discusses the implementation, challenges, and existing techniques of particle graphics, as well as proposes particle graphics as a candidate for special hardware. Section 3 introduces and discusses the Hardware Particle Machine. The Particle Pipe design and its systems are studied in more detail in Section 4. Finally results and analysis of currently completed work are presented in Section 5, and a conclusion is offered in Section 6.

2 Particle Graphics

2.1 General Implementation

A typical implementation of a particle system involves all the particles of the system being loaded sequentially from memory, updated, rendered, and then finally written back to memory. At each frame, a new particle can be created with randomly varying initial conditions. Dead particles, particles which have been alive for some time or reached some state, can be deactivated and replaced by newly initiated particles. For each particle processed, a set of forces is calculated for a particle using its current data as well as some environmental data. These forces might included gravitational and electrical forces, vortex forces, wind and current forces, viscosity, friction, explosive forces, spring forces, and anything the designer invents to create the desired system behavior and effects [2]. From these forces a particle acceleration is calculated. Next, the particle's motion is integrated using a Euler integration step. This gives a new velocity and position to the particle. Subsequently, collision detection and collision resolution is performed allowing particles to collide and interact with their environment. Finally, the particle is rendered. A particle could be rendered very simply as a single colored pixel, a streak, or anything else. Since the visual detail of the particle system comes from the massive collection of distinct and simple particles, even the most simple rendering scheme, such as a colored pixed for each particle, can be sufficient.

It is worth noting that particle graphics effects, at least the type of large particle systems used for graphics are almost always first order. Even those described in [3] can be considered as first order in the sense that the number of inter-particle interactions is not proportional to n^2. There are some projects involving reconfigurable hardware for 2^{nd} order particle calculations for off-line scientific simulation [4], such as the Grape Project [5]. We, on the other hand, are interested in particle simulations intended for real-time graphics effects. Those which we are considering here are limited to 1^{st} order systems. In other words the particles do not interact with each other. They only interact with the system and environment, allowing the entire system of particles to be updated and processed in one single pass.

2.2 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 of a particle system in an live, interactive application can only be allocated a very limited portion of a computers system resources. These interactive graphical applications, are demanding and typically make full use of available computer resources. Therefore, due to the requirements of managing a large particle system and the inability of a demanding application to devote the majority of its computer system resources to a particle system, large 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 [6]. Software based particle systems in games are currently limited to about 10,000 particles per frame [6].

2.3 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 embedded a large software particle system into a real-time application without competing for resources with the application.

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.*

2.4 Programmable GPU Based Implementation

Recently, powerful techniques have been developed which make use of programmable floating point graphics hardware to accelerate particle systems. This technique was described in [6]. 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 steams 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 performs some calculation on the particles in that stream, and then obtains a stream containing data with updated values. The GPU supported systems [6] 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.

This work 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.

2.5 Bigger and Better Particle Systems

What would happen if it were possible to embed a huge particle graphics system into graphics applications without any significant burden or cost to system performance? The use of particle based simulations for modeling all kinds of objects and phenomena would change drastically. The scope of use of particle graphics would 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 more complex and flexible behavior. **Most importantly, 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. What is currently only possible with pre-calculated and pre-rendered off-line applications, could become a reality for live interactive applications.

2.6 An Ideal Solution for Particle Graphics

What characteristics must an ideal solution for particle graphics have to realize these goals? **Most critically, the implementation must be able to completely isolate all of the work and resources 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 accesses, 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 an interacting and colliding member of its environment. Finally, it must be flexible and customizable.

2.7 Particle Graphics as a Candidate for Hardware Acceleration

Could particle graphics, more specifically the embedding of particle graphics into real-time 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 a reconfigurable co-system?

The problem is completely parallel in nature. For 1^{st} order particle effects, every particle can be processed completely independently of every other particle in one single pass (with a few exceptions such as depth sorting for rendering). Although all particles could theoretically be processed simultaneously, software or GPU accelerated implementations still process particles sequentially. For this reason, custom particle hardware has a great potential for dramatic acceleration via a parallel hardware design

Secondly, upon investigating the sequence of tasks which must be performed for each particle processed, one finds that it is ideal for a pipeline structure. There is a single, unidirectional and constant flow of data, which is easily divided into separate, independent stages.

Particle graphics would benefit from a separate daughter system with isolated particle memory and processing hardware, completely containing the particle simulation. In this

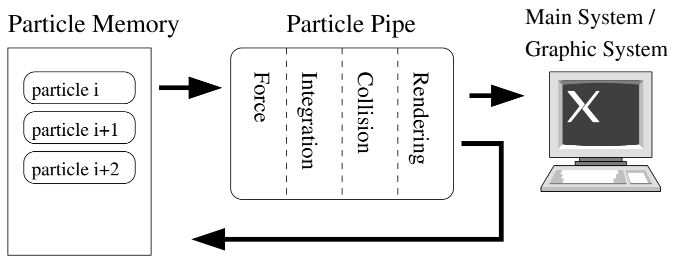

Figure 2. Particle Data Flow

way, a large particle simulation can be managed without competing for system resources.

3 Hardware Particle Machine

3.1 Particle Machine Overview

The Particle Machine, shown in Figure 2, is a self contained system that completely contains, manages, and executes particle graphics simulations. This system could be implemented in reconfigurable logic, on an FPGA card with RAM in a general purpose PC, providing accelerated particle graphics support to application software. The Particle Machine is a system comprised of particle memory, control logic, and the Particle Pipe, a pipelined Particle Update Processor.

The Particle Machine is a system that will fully contain a large pool of particles, with which any number of simulations can be simultaneously created. It carries out all relative execution and tasks under the control of application software. At each frame or simulation step, the Machine will either provide software all or a select portion of simulation results and current data. This data will be sent directly to graphics hardware for rendering, or back to software for integration into application graphics. These results may include contents such as: rendered graphical data for visible particles including z-buffer depth values, all of the current particle data, or selected collision information.

The basic operation of a particle simulation within the Particle Machine is as follows. First, application software sets up and controls properties and parameters for the desired simulations by communicating with the Particle Machine controller. The controller will continuously initialize new particles in particle memory when necessary, and will set and control parameter registers in the Particle Pipe, thus controlling its functionality. This occurs in response to software requests specifying the parameters and properties of simulations. At each frame, all of the particle data is loaded sequentially from particle memory, and fed one particle per clock cycle, into the Particle Pipe. Since the Pipe is fully pipelined, one particle will complete processing on every clock cycle. As each particle in the system is completed, it is output from the other side of the Pipe with new updated data, which is sent back to particle memory, and graphical data.

3.2 Particle Memory

The Particle Machine needs high bandwidth and exclusive access to a large memory, dedicated to containing the entire pool of available particles. This memory, the particle memory shown in Figure 2, is part of the Particle Machine and is separate and isolated from main system memory. The Pipe accepts one particle data set as input and provides another set as output on each clock cycle. Therefore particle memory mush at least be able to provide $(b_{particle} \times 2 \times f_{pipe})\frac{bits}{sec}$ of read and write access, where $b_{particle}$ is the bit width of one particle's data set, and f_{pipe} is the frequency of the Particle Pipe clock. All accesses to particle memory, with the exception accesses for particle initializations, are made in a regular, sequential pattern. This means that 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 at least must 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. In our case we use 16 bits for color. The life count is an integer which is set to some value when a new particle is initialized by the Machine controller. It can be, depending on the simulation parameters, decremented on every pass through the Pipe. When a particle's life count reaches zero, it is considered "dead" or inactive, and will not contribute to the simulation. The particle memory entries occupied by inactive particles are available for new particle initialization. Finally, there is a very special field in the particle data, the type field. The Particle Machine contains one giant array of particles in memory which will be repeatedly passed through the Particle Pipe. Although there is one array, or pool, of particles, this will be use 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 of the Pipe, as it moves though the Pipe.

3.3 Particle Pipe

The Particle Pipe, shown in Figure 2, 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 a simulations. The Pipe includes four major systems:

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

3.3.1 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. A 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 needs use to implement many high speed multiplications without using reconfigurable fabric. Altera Stratix FPGAs and Xilinx Virtex FPGAs both have hardwired resources for implementing 18x18 bit multiplications. For these reasons, an 18 bit (4:14) fixed point format is currently

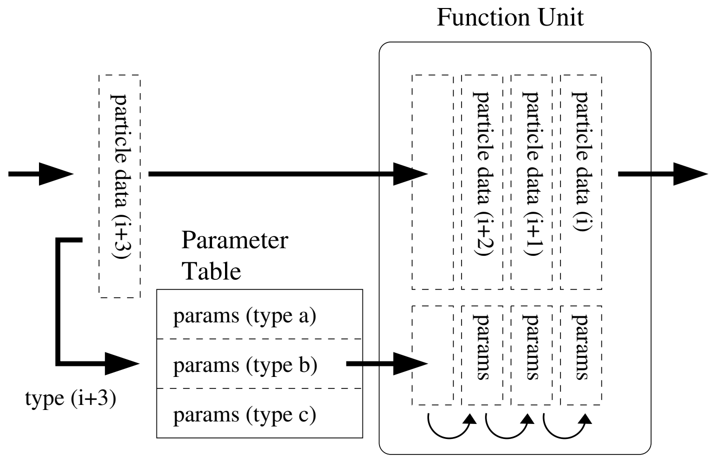

Figure 3. Pipe Parameter Selection

used in our Pipe. This format, most importantly, makes the best use of dedicated multiplier circuits in FPGAs, while still allowing particle data sets to fit perfectly into 64 bits, making optimal use of particle memory.

3.3.2 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 done for a general particle graphics simulation. These sections, do not refer to 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". The interfaces between blocks or sections are consistent and well defined. The building block modules which make up sections, the function units have consistent interfaces making them replaceable. By choosing and designing new sections which provide desired operations while satisfying the interface, a Particle Pipe can be customized for one particular use. This *FPGA-compile-time* reconfigurability of the Pipe is an important feature, since a good particle graphics system must be flexible and customizable.

3.3.3 Pipe Functionality Control

How is the functionality of the Pipe controlled? Recall that the Particle Machine contains a giant pool of particles for making a number of concurrent simulations. What's more, within one simulation, there may be several distinct kinds of particles with different properties, 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 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 the unit, a set of contents for the unit's parameter registers is selected from a table of values, using the particle data's *type field* as a selection index. This is depicted in Figure 3. These selected parameters then 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 hardware, 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*, determines the functionality of the Pipe on that particle data.

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 parameters, along with the groups of particles associated with them. Software fills in the values of the parameter tables for each function unit, 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 vector to each particle's sum of forces. Such a block might have 2 configuration registers. One is an enable bit, which will determine if this block should be enabled for a particle, and another will determine what that force vector is. Now, imagine a simulation in which there are 3 types of particles. Particles of type 1 will not experience this force at all. Particles of type 2, will experience the force $(1.0, 0.0, 0.0)$. Particles of type 3, will experience the force $(0.0, -2.2, 0.0)$. To accomplish this, software will set values in the parameter selection tables for this function unit, such that the enable bits for particle types 1, 2, and 3 are $'0'$, $'1'$, and $'1'$ respectively, and the force vector values for particle types 1 and 2, are $(1.0, 0.0, 0.0)$ and $(0.0, -2.2, 0.0)$.

4 Particle Pipe Systems

4.1 Forces

The force system, shown in Figure 4, 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 received as input all current particle data, together with a set of *type-selected* force parameters. Each force unit output is a 3–D force vector, in the Pipe fixed point format. The force vectors output from each unit are each delayed and summed to one total force vector.

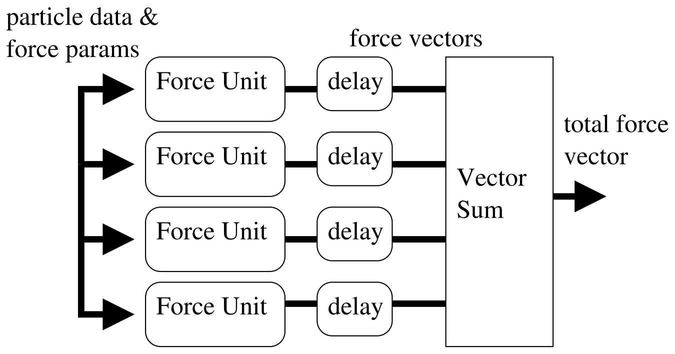

Figure 4. Force System

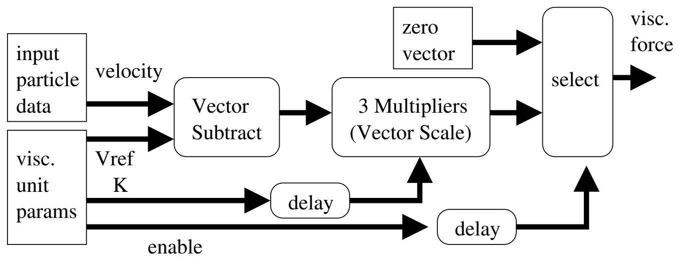

Figure 5. Viscosity Force Unit

Force units which can be included in the force stage include uniform forces, viscosity forces, vortex forces, attractive and repulsive forces, spring forces, "random nudge" forces, and many more. In Figure 5 we take a look at the implementation of a viscosity force unit. The unit calculates a general viscous force using:

$$\vec{f}_{visc} = \vec{k}_{visc}(\vec{v}_{ref} - \vec{v}_{particle}) \ , \tag{1}$$

where $\vec{f}_{visc}$ is the viscous force result, $\vec{k}_{visc}$ is the scalar viscosity constant, $\vec{v}_{ref}$ 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.

4.2 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 relationship:

$$\vec{a} = \frac{1}{m}\vec{f}_{total} \ . \tag{2}$$

First, one hardware division obtains the $\frac{1}{m}$ term, which is then multiplied with each component of the force vector.

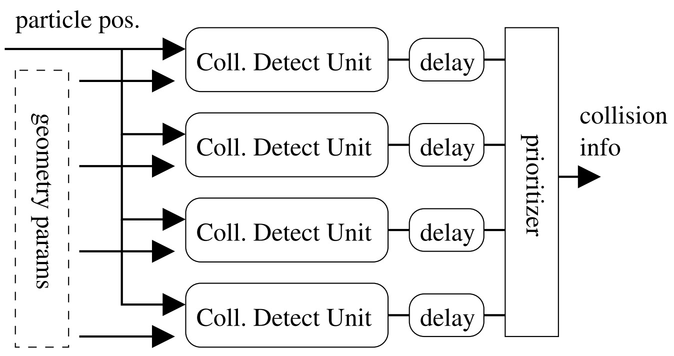

Figure 6. Collision Detection System

4.3 Integrate Motion

Each particle's path of motion, or position over time, needs to be integrated from the following differential equation which is equivalent to Equation 2:

$$\frac{d^2\vec{r}}{dt^2} = \frac{\vec{f}}{m} \ . \tag{3}$$

A particle system should use the simplest integration method possible. This is the Euler step method. Verlet integration is also a possibility for particle graphics [6], but a Euler step integration by far the best choice for hardware due to its simplicity and explicit use of particle velocity, needed in other parts of the Pipe. The position and velocity vectors are updated in parallel as follows:

$$\begin{aligned} \vec{v}_{new} &= \vec{v} + \vec{a} \\ \vec{r}_{new} &= \vec{r} + \vec{v} \ . \end{aligned} \tag{4}$$

4.4 Update Properties

Particle systems have many properties that can be changed and updated using a simple rule to implement an important feature. These include such optional and type-configurable update rules as:

- Decrementing the *life count*,
- Fading particle colors by a *color fade step*,
- Killing particles which satisfy some condition such as energy or position being beyond a certain value,
- Interpolating between particle colors based on some value such as time or energy.

4.5 Collision Detection

Particles needs 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 complex geometries, detects and resolves collisions with environments of simple geometry efficiently and conveniently. As shown in Figure 6, 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 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 exactly what 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 shape with several of the basic detection units available.

Similar to the force system, collision detection units are in parallel, and their outputs, collision information, are delayed long enough to be properly lined up. Finally, one set of collision information is selected from the set of outputs and passed on to the collision response stage.

4.6 Collision Response

Figure 7 shows a simplified version of the collision response system. If in fact there was a collision detected, and the collision flag received as part of the input collision information was set, the collision response system will need to perform the following tasks:

- Replace the particle position by the intersection point estimate.
- Scale the tangential particle velocity by the surface friction factor.
- If the projection of the particle velocity on the surface normal is negative, the particle must be "bounced", by multiplying the normal particle velocity by the bounce factor.

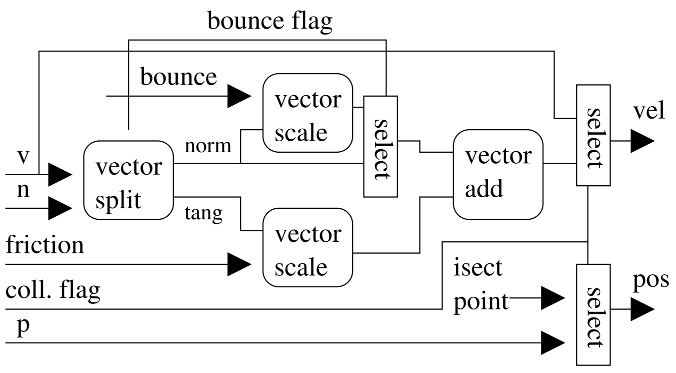

Figure 7. Simplified Collision Response System

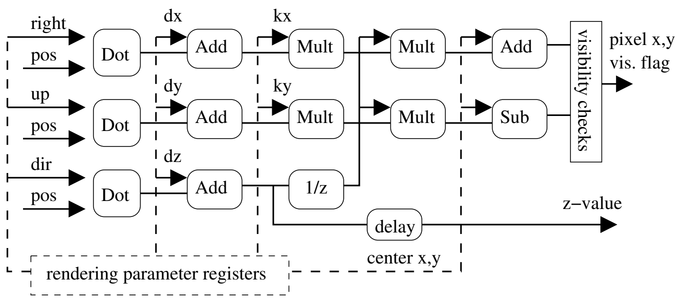

Figure 8. Simplified Rendering System

- Combine the new normal and tangential particle velocities to form a new total velocity vector.

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

$$\begin{aligned} \vec{v}_{norm} &= (\vec{v} \cdot \hat{n}_{surface})\hat{n}_{surface} \\ \vec{v}_{tang} &= \vec{v} - \vec{v}_{norm} \ . \end{aligned} \tag{5}$$

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:

$$\begin{aligned} \vec{v}_{tang} &\leftarrow k_{friction}\vec{v}_{tang} \\ \vec{v}_{norm} &\leftarrow k_{bounce}\vec{v}_{norm} \ . \end{aligned} \tag{6}$$

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 found is used, otherwise, a delayed version of the original normal velocity is used.

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, if the collision flag had been set, 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.

4.7 Rendering

A simplified version of the Pipe's rendering system is shown in Figure 8. Rendering is the last part of the Particle Pipe. At this stage we have all updated particle data. Graphical information is calculated using the new particle data together with rendering parameters set-up and updated continuously by software. The graphical information calculated and output for each particle includes:

- a visibility flag,
- screen pixel coordinates,
- a color value,
- a *z-buffer* depth value.

As shown in Figure 2 at the end of the rendering stage, the rendering results together with the updated particles are output from the Pipe. Particle data will be set back to particle memory, and rendering results will be used by the system.

The first task in rendering is to finding a set of *view coordinates* for the particle. Simulations exists in *world-space* and the particle's position vector contains values in world-space coordinates. To convert the particle's coordinates from world space to view space we apply the following transform:

$$\begin{aligned} x_{view} &= \vec{r}_{world} \cdot \vec{right} + d_x \\ y_{view} &= \vec{r}_{world} \cdot \vec{up} + d_y \\ z_{view} &= \vec{r}_{world} \cdot \vec{dir} + d_z \ . \end{aligned} \tag{7}$$

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 we have 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.

$$\begin{aligned} x_{screen} &= k_{x-scale}\, x_{view} \frac{1}{z_{view}} \\ y_{screen} &= k_{y-scale}\, y_{view} \frac{1}{z_{view}} \end{aligned} \quad (8)$$

The previous 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. Therefore the following formula is used to transform the screen coordinates into more useful pixel coordinates.

$$\begin{aligned} x_{pixel} &= x_{screen} + x_{center} \\ y_{pixel} &= y_{center} - y_{screen} \end{aligned} \quad (9)$$

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.

5 Results

5.1 Functionally Equivalent Software Model for Testing

We have a C software model that was created in parallel with the VHDL hardware design. It is a bit accurate, functionally equivalent model of the Particle Pipe that can be as a numerical test bench for testing and verifying the Particle Pipe HDL design, and testing and experimenting with new function units and Pipe modules. It can be also used for interactive testing and verification of the Pipe hardware design, graphically displaying interactive particle simulations such as the one shown in Figure 1.

5.2 FPGA Resource Utilization and Synthesis Results

We have completed the design and verification of the Particle Pipe VHDL hardware design, with a minimum set of function units and features, sufficient for basic testing and proof of concept. It was synthesized and placed-and-routed for an Altera EP1S40 Stratix FPGA. A complete demonstration system, supporting a working Particle Pipe, integrated with an on–chip Nios microcontroller and video system for graphical display of particle simulations has been developed. The Particle Pipe alone was synthesized to operate at 130 MHz, and the complete system–on–chip demostration system, including the Nios microcontroller system, was synthesized for operation at 80 MHz in the EP1S40 FPGA. Table 1 shows statistics for FPGA resource utilization after synthesis and FPGA fitting.

	Logic cells	Registers	18x18 Mult.
Avail. in FPGA	41,250	44,860	56
Entire Pipe	30,283	27,853	28
Force System	5,157	4,962	3
Coll. Detection	2,472	2,301	0
Coll. Response	12,699	12,031	12
Rendering	8,382	7,034	13
Nios μController	6,397	2424	1

Table 1. FPGA Synthesis Results

5.3 Analysis of Potential and Performance

What is the potential performance that can be expected from a full implementation of the described Particle Graphics Machine with all its components? How useful and applicable is this approach to particle graphics acceleration? How does it compare to all software, and the GPU supported methods?

The Particle Pipe is the heart and engine of the design. It has a potential throughput of 1 particle per clock cycle, and requires a single pass through particle data for one update cycle. Software implementations cannot come close to processing 1 particle per clock cycle with 1 pass per update. Software methods require the execution of large amounts of code for each particle, each assembly instruction of which takes at least a clock cycle. GPU supported particle systems [6] are not an exception, since they actually *are* software methods, relying on software executed by both the CPU and GPU. Pure software and GPU supported implementations compete for main memory access, and in the later case, GPU access. GPU supported implementations are multi-pass, meaning that they require multiple cycles of preparing data steams and processing those streams for a single particle system update.

On the other hand, standard computer hardware operates at frequencies from five to ten times faster than those which we can expect from reconfigurable logic. The application scope of particle system hardware is narrow, leading us to believe that its implementation would be most appropriate as a soft-core hardware accelerator, for use on general purpose, user programmable reconfigurable hardware in a PC, available for the acceleration user applications.

Software methods are flexible and easily customizable, which is critical to graphics effects. If the Particle Machine is implemented as an application programmed, reconfigurable accelerator, it can be designed and customized specifically to accelerate one application, providing a custom set of effects. Therefore, we believe it does offer the necessary flexibility and ability to be extended, with the downside of requiring hardware design for application customization. GPU supported methods seem to be customizable and extensible, but within some limitations. However they have the desirable feature of requiring only software changes for application customization. The strongest advantage to the Hardware Particle Machine implementation is that the system cost and burden created by managing a particle system does not increase with the number of particles as other methods do. This is in theory of course. In practice, the Particle Pipe can probably not be used at its potential due to practical implementation problems. These include problems such as the processing of the graphical data created by the Particle Machine, memory bandwidth and access limitations, and bus communication limits.

The current design is a 130 MHz Particle Pipe with two 64 bit particle memory interfaces. However, the demonstration system implementation runs at 75 MHz, providing the Particle Pipe with one dedicated, 32 bit particle memory, and a a seperate control & parameter bus system. The demonstration system also includes an on–chip Nios microcontroller with its own memory, bus system, and video system for displaying interactive particle graphics simulations created by the Pipe.

For an estimate of the theoretical, potential throughput and performance of the current Pipe design, assume that the Particle Pipe has access to a dedicated particle memory capable of reading and writing one particle data package for every Pipe clock cycle, a is supported by a system capable of receiving and displaying all of its graphical output data with no delays. The total throughput of the Particle Pipe itself is then $\frac{130}{60 frames/sec} \approx 2,166,666$ PPF(particles per frame). The particle simulation is totally contained in the Particle Pipe, and thus does not effect main system performance. As reported in [6], GPU supported particle systems can achieve 250,000 PPF in a full featured application, and pure software only 10,000 PPF.

6 Conclusion

In this paper we presented the design of a Hardware Particle Machine, for implementation in an FPGA, intended for accelerating real-time particle graphics in applications such as video games. The Particle Pipe has been synthesized to 130 MHz, on an Altera Stratix FPGA, resulting in a potential throughput of 2.1 million PPF (particles per frame). This throughput is achieved with minimal load on application and main system performance. We have also implemented a working demonstration system, to create and display interactive particle graphics using the Particle Pipe design.

References

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

[2] John van der Burg. Building an Advanced Particle System. *Game Developer Magazine*, 2000.

[3] Tommi Ilmonen, Janne Kontkanen. The Second Order Particle System. *WSCG Proceedings*, 2003.

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

[5] 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*, 468:51-61, 1996.

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