Graphics Hardware (2004)
T. Akenine-Möller, M. McCool (Editors)

UberFlow: A GPU-Based Particle Engine

Peter Kipfer, Mark Segal, Rüdiger Westermann

Computer Graphics & Visualization, Technische Universität München[†] , ATI Research[‡]

Abstract

We present a system for real-time animation and rendering of large particle sets using GPU computation and memory objects in OpenGL. Memory objects can be used both as containers for geometry data stored on the graphics card and as render targets, providing an effective means for the manipulation and rendering of particle data on the GPU.
To fully take advantage of this mechanism, efficient GPU realizations of algorithms used to perform particle manipulation are essential. Our system implements a versatile particle engine, including inter-particle collisions and visibility sorting. By combining memory objects with floating-point fragment programs, we have implemented a particle engine that entirely avoids the transfer of particle data at run-time. Our system can be seen as a forerunner of a new class of graphics algorithms, exploiting memory objects or similar concepts on upcoming graphics hardware to avoid bus bandwidth becoming the major performance bottleneck.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

1. Introduction

From a conceptual point of view, a particle engine consists of a set of connected components or modules responsible for the creation, manipulation and rendering of particle primitives [Ree83, LT93, BW97, McA00, Sim90]. In computer graphics, particle engines are most commonly used to generate volumetric effects like fire, explosions, smoke or fluids, based on simple physical models. More elaborate particle engines are employed for physically based dynamics simulation (e.g. Smoothed Particle Hydrodynamics [Mon88], Boltzmann models [Boo90, Doo90]). Such systems usually integrate particle-particle interactions as well as more complex physical models to describe particle dynamics.

In current particle engines, the dynamics module is run on the CPU. The rendering module sends particle positions and additional rendering attributes to the GPU for display. This conventional assignment of functional units to processing units reveals the capabilities of early generations of graphics processors. Such processors were solely optimized for the *rendering* of lit, shaded and textured triangles. Nowadays, this design is abandoned in favor of programmable function pipelines that can be accessed via high level shading languages [MGAK03, Mic02]. On current GPUs, fully programmable parallel geometry and fragment units are available providing powerful instruction sets to perform arithmetic and logical operations. In addition to computational functionality, fragment units also provide an efficient memory interface to server-side data, i.e. texture maps and frame buffer objects.

Both in conventional particle engines and in engines that exploit parallel computations and memory bandwidth on the GPU, updated particle positions have to be transferred to client memory before they can be used as input for the geometry engine. Then, bandwidth is becoming a major performance bottleneck, and with the ability to do more operations per time interval on the CPU or the GPU, the bandwidth required will grow substantially. Consequently, for high resolution particle sets the transfer of these sets for rendering purposes has to be avoided.

OpenGL memory objects and similar concepts, i.e. texture access in vertex shaders using PixelShader 3.0, provide this functionality. A memory object is a block of data in graph-

[†] kipfer@in.tum.de, westermann@in.tum.de
[‡] segal@ati.com

ics memory that can be used simultaneously as vertex buffer and render target. This mechanism allows one to direct the output of a fragment program into such objects, and to feed the data into the geometry engine as an array of vertex positions or additional per-vertex attributes. Consequently, the benefits of fragments programs, i.e. random access to textures, parallelism and memory bandwidth, can be combined with the ability to render large sets of geometric data from graphics memory. In this work, we present an efficient realization of such memory objects on the latest ATI card, the Radeon 9800.

To fully take advantage of this mechanism, efficient GPU realizations of algorithms used to perform particle manipulation are essential. In this paper, we present novel approaches to carry out particle simulation in the fragment units of programmable graphics hardware. At the core of these techniques, we present an optimized sorting routine that is far faster than previous approaches. Built upon this implementation, we have implemented inter-particle collision detection and visibility sorting including hundreds of thousands of primitives. In combination with OpenGL memory objects we present the first particle engine that entirely runs on the GPU *and* includes such effects. This is a significant extension of previous approaches, i.e. [KvdDP03, GRLM03], where the GPU was only exploited to either account for collisions with implicitly defined surfaces or to perform the broad phase in collision detection between polygonal objects.

The remainder of this paper is organized as follows. In the following, we first review related work in the field of GPU programming. In Chapter 3 we introduce the concept of OpenGL SuperBuffers, and we describe their realization on recent ATI cards. The use of parallel fragment units for collision detection and visibility sorting is subject of Chapter 4. Here, we also outline the conceptual design of a particle engine on programmable graphics hardware. In Chapter 5 we show timing statistics for different scenarios including collisions and front-to-back sorting. We conclude the paper with a detailed discussion, and we show further results of our approach.

2. Related Work

To take full advantage of new graphics chip technologies, considerable effort has been spent on the development of algorithms amenable to the intrinsic parallelism and efficient communication on such chips. In many examples, programmable GPUs have been explored to speed up algorithms previously run on the CPU. The computational power and memory bandwidth on such processors has been exploited to accelerate the simulation of both local as well as global illumination effects (e.g. [PDC*03, DS03]). Besides its use for the realistic rendering of geometric data, programmable graphics hardware has also been harnessed for the rendering of volumetric data sets [RSEB*00, EKE01, KPHE02]. The acceleration of image based modeling and rendering techniques on GPUs has been considered for example in [YWB03, LMS03, HMG03]. Recently, a number of researchers have demonstrated the benefits of graphics hardware for the implementation of general techniques of numerical computing [HBSL03, LKHW03, SHN03, MA03, KW03, BFGS03].

The results given in many of these examples show, that for compute bound applications as well as for memory bandwidth bound applications the GPU has the potential to outperform software solutions. However, this statement only holds for such algorithms that can be compiled to a stream program, and which then can be processed by SIMD kernels as provided on recent GPUs. On the other hand, execution speed is not the only concern when mapping algorithms to graphics hardware. Another concern is to avoid any data transfer between the CPU and the GPU. As we have to deal with increasing scene and model complexity, bandwidth requirements become an ever more important issue in a graphics application. By trying to keep both the application program and the rendering module on the graphics subsystem, bandwidth limitations can be avoided thus achieving superior frame rates even when execution speed is not significantly higher.

3. OpenGL SuperBuffers

Our method relies on computing intermediate results on the GPU, saving these results in graphics memory, and then using them again as input to the geometry units to render images in the frame buffer. This process requires application control over the allocation and use of graphics memory; intermediate results are “drawn” into invisible buffers, and these buffers are subsequently used to present vertex data or textures to the GPU.

Our implementation exploits a feature of recent ATI graphics hardware that allows graphics memory to be treated as a render target, a texture, or vertex data. This feature is presented to the application through an extension to OpenGL called *SuperBuffers*. The interface allows the application to allocate graphics memory directly, and to specify how that memory is to be used. This information, in turn, is used by the driver to allocate memory in a format suitable for the requested uses. When the allocated memory is bound to an *attachment point* (a render target, texture, or vertex array), no copying takes place. The net effect for the application program therefore is a separation of raw GPU memory from OpenGLs semantic meaning of the data. Thus, SuperBuffers provide an efficient mechanism for storing GPU computation results and later using those results for subsequent GPU computations.

4. GPU-based Particle Engine

Since we aim at developing a particle engine that entirely runs on the GPU, there is a dire need to reinvent modules of the engine in regard to the particular target architecture. Only then can we expect such a system to achieve significantly better performance rates than a software solution. In the following, we describe the most relevant modules responsible for the state of our system. The state is updated on a per time step basis, where a single time step is comprised of the following events:

- Emission
- Collisionless motion of particles
- Sorting of particles
- Pairing of collision partners
- Collision response
- Enforcement of boundary conditions
- Particle rendering

4.1. Emission

Initially, by any suitable scheme particle positions, $r(t)$, are distributed over the 3D domain, and they are encoded in the RGB components of a server-side 2D RGBA texture map of size $n \times n$. As particles are usually released in different time steps of an animation, each gets assigned a time stamp that is stored in the A-component of this texture. The number of unique time stamps depends on the life time of particles in the animation. As we can only use a fixed number of particles, this life time effectively determines the maximum number of particles to be released per time step. A second texture contains for every particle its current velocity, $v(r,t)$. Initially, this velocity is set to a constant value.

To perform any modifications on particle attributes, a view plane aligned quad covering $n \times n$ fragments is rendered. Both textures are bound to that quad, thus enabling a fragment program to access and to modify these attributes. Uniform attributes for all particles can be specified in additional texture coordinates of the quad. Updated values are simultaneously rendered to separate texture render targets using the `ATI_draw_buffer` extension. In the next pass, these targets become the current containers to be processed.

4.2. Collisionless Particle Motion

In the current implementation, each particle is first streamed by its displacement during time interval dt. The displacement is computed using an Euler scheme to numerically integrate quantities based on Newtonian dynamics:

$$v(r,t+dt) = v(r,t) + v_{ext}(r,t) + \frac{F}{m}dt \qquad (1)$$

$$r(t+dt) = r(t) + \frac{1}{2}(v(r,t) + v(r,t+dt))dt \qquad (2)$$

Here v_{ext} is an external wind field used to model particle movement, F is an external force, and m is the particle mass. The wind field is stored in a 3D texture, and it can thus be sampled in the fragment program by using particle positions as texture coordinates.

4.3. Sorting

Once collisionless motion of particles has been carried out, the engine performs either of the following tasks: it resolves inter-particle collisions and renders the particles as opaque primitives *or* it sorts particles according to their distance to the viewer and renders the particles as semi-transparent primitives. Both scenarios rely on the sorting of particles. Therefore, a sorting key to be considered in the sort has to be specified.

In the latter scenario, the sorting key is the distances of particles to the viewer. In the former scenario, particle space is assumed to be divided into the cells of a regular lattice of mesh size g_0, and grid cells are uniquely enumerated in ascending z-, y-, x-order. The index of the cell containing a particle is associated with that particle, and it is used further on as sorting key.

4.3.1. Sorting Keys and Identifiers

Sorting keys are computed once at the beginning of the sort. In a single rendering pass, the fragment program either computes the distance of particle positions to the viewer or it calculates the grid cell floating point index by $x/g_0^2 + y/g_0 + z$. In addition, to every particle a unique identifier is associated. Sorting keys and identifiers are rendered into the R- and G-components of a 2D texture render target, which is then sorted in row-major order. After finishing the sort, containers for particle positions and velocities are rearranged according to the arrangement of identifiers.

Care has to be taken when computing identifiers on a chip supporting limited integer or floating point precision. Recent ATI graphics hardware only provides 24 bit internal floating point format. With a 16 bit mantissa and a 7 bit exponent we get $2^{16} = 65536$ distinct values in the range $[2^{h-1} \dots 2^h]$. As we want to animate far larger particle sets of up to 1 million particles, a different enumeration scheme needs to be developed.

When we let the exponent h take positive values from 1 to 2^4, we get 2^{20} distinct but not equally spaced values. These values can be pre-computed on the CPU and stored in ascending row-major order into a texture map. If we assume 2^{20} particles to be processed per frame, a texture of size $2^{10} \times 2^{10}$ is sufficient to serve as container for these values. To get a unique identifier for each particle, a fragment program simply has to look up the respective identifiers from this container.

At the end of the sorting procedure, from 1D identifiers the initial 2D texture coordinates have to be decoded in order to rearrange particle positions and velocities accordingly. Since we know that values within consecutive sets of

$2^{16}/2^{10}$ rows are equally spaced, for every identifier we only need to determine in which of the $2^4 = 16$ sets it is positioned. This is done by successively subtracting the range $[2^i \dots 2^{i-1}]$ of each set from the identifier until it becomes smaller than zero. Then, the identifier is contained in the i^{th} set, and a final division and modulo operation by the spacing in that set times the number of entries per row yields the appropriate texture address. This address is used to fetch corresponding particle positions and velocities, which are finally output by the fragment program.

4.3.2. Bitonic Sort

To perform the sorting procedure efficiently, we account for the architecture of todays graphics processors. Recent GPUs can be thought of as SIMD computers in which a number of processing units simultaneously execute the same instructions on their own data. GPUs can also be thought of as stream processors, which operate on data streams consisting of an ordered sequence of attributed primitives like vertices or fragments. Considerable effort has been spent on the design of sorting algorithms amenable to the data parallel nature of such architectures. Bitonic sort [Bat68] is one of these algorithms. Bitonic sort first generates a sequence necessary as input for the final merge algorithm. This sequence is composed of two sorted subsequences, where the first is in ascending and the other in descending order. Bitonic merge finally merges both subsequences in order to produce a globally ordered sequence.

Purcell et al. [PDC*03] proposed an implementation of the Bitonic sort on a graphics processor, i.e. the *n*VIDIA GeForceFX graphics accelerator. It was used to sort photons into a spatial data structure, yet providing an efficient search mechanism for GPU-based photon mapping. Comparator stages were entirely realized in a fragment program, including arithmetic, logical and texture operations. The authors report their implementation to be compute limited rather than bandwidth limited, and they achieve a throughput far below the theoretical optimum of the target architecture.

In the following, we present an improved Bitonic sort routine that achieves a performance gain by minimizing both the number of instructions to be executed in the fragment program and the number of texture operations.

4.3.3. The Sorting Pipeline

To explain our implementation, let us put emphasis on some of the characteristics of Bitonic sort when used to sort a 2D texture. As we can see from figure 1, as long as the texture is to be sorted along the rows, in every pass the same instructions are executed for fragments within the same column. Even more precisely, in the k^{th} pass

- the relative address or offset $\triangle r$ of the element that has to be compared is constant
- this offset changes sign every 2^{k-1} columns
- every 2^{k-1} columns the comparison operation changes as well.

The information needed in the comparator stages can thus be specified on a per-vertex basis by rendering column-aligned quad-strips covering $2^k \times n$ pixels. In this way, the output of the geometry units can be directly used as input for the fragment units, thus avoiding most of the numerical computations in the fragment program.

In the k^{th} pass $n/2^k$ quads are rendered, each covering a set of 2^k columns. The constant offset $\triangle r$ is specified as uniform parameter in the fragment program. The sign of this offset, which is either 1 or -1, is issued as varying per-vertex attribute in the first component (r) of one of the texture coordinates. The comparison operation is issued in the second component (s) of that texture coordinate. A less than comparison is indicated by 1, whereas a larger than comparison is indicated by -1. In a second texture coordinate the address of the current element is passed to the fragment program. The fragment program in pseudo code to perform the Bitonic sort finally looks like this:

Row-Wise Bitonic Sort

1 $OP1 = TEX1[r_2, s_2]$
2 $sign = r_1 < 0$? -1 : 1
4 $OP2 = TEX1[r_2 + sign * \triangle r, s_2]$
5 output = $OP1.x * s_1 < OP2.x * s_1$? OP1 : OP2

We should note here that the interpolation of per-vertex attributes during scan-conversation generates a sign value between 1 and -1. Consequently, in line 2 the 1 or -1 values must be reconstructed.

To perform row-wise sorting of texture elements, $\sum_{i=1}^{log(n)} \sum_{j=1}^{i}$ quads have to be generated up front. Each set of quads to be rendered in one pass is stored into a separate display list. Once the texture has been sorted along the rows, another $log(n)$ stages have to be performed. In the i^{th} stage, i additional passes are executed to merge 2^i consecutive rows. Finally, row-wise sorting is performed as described. Since sets of rows are always sorted in ascending order from left to right and from top to bottom, every other set of 2^i rows has to be rearranged to generate a descending sequence required by the Bitonic sort. This operation is implicitly realized in the first pass of each stage.

As one can see from figure 2, the same approach of rendering geometry in order to pass information from the geometry units to the fragment units can be used in the merging of rows as well. The only difference is, that quads now have to be transposed to produce constant values along rows instead of columns.

The final optimization results from the observation that the graphics pipeline as implemented on recent cards is highly optimized for the processing of RGBA samples. As a matter of fact, we pack 2 consecutive entries in each row – including sorting key and identifier – into one single RGBA texture value. This approach is extremely beneficial, because

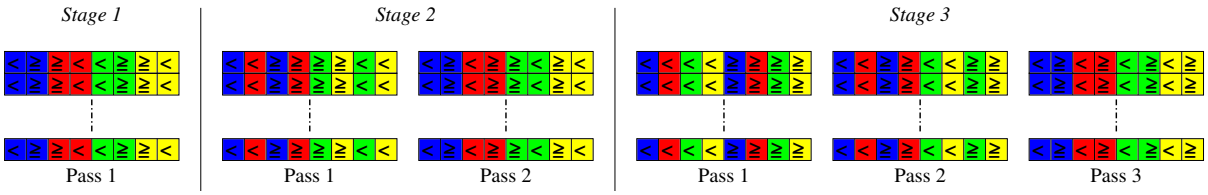

Figure 1: *Bitonic sort of rows of a 2D field. In one row, equal colors indicate elements to be compared. For each element, the operation it has to perform is shown as well.*

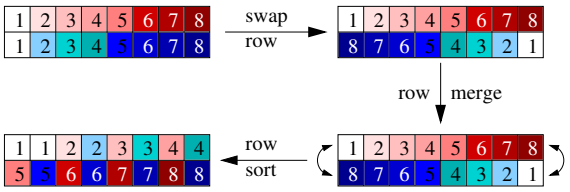

Figure 2: *Bitonic sort of consecutive rows. Every other row is reversed and then merged with its predecessor. Finally, each row is sorted separately, producing a sorted set of elements in every pair of rows.*

it effectively halves the number of fragment operations that have to be performed. In addition, in the first pass of every sorting stage the second operand does not need to be fetched at all. On the other hand, the fragment program only becomes slightly longer because the conditional statement now has to be executed twice. Just at the end of the sorting process is one additional rendering pass required to decode packed texture entries and to rearrange particle positions and velocities according to the arrangement of sorting keys.

4.4. Collision Detection

The collision detection module simultaneously computes for each particle an approximate set of potential collision partners. Only the closest one is kept and used as input for the collision response module. In situations involving multiple collision, resolving these events in parallel can lead to wrong results. Although time-sequential or simultaneous processing of collision events as proposed in [Hah88, Bar89, Mir00] yields correct results, such techniques are not appropriate for the implementation on current graphics architectures.

From the sorted 2D texture, each fragment now fetches the current particle position, and it also fetches a number of particle positions to the left and to the right of this position. Of all these positions the one closest to the current one is kept, and the respective texture coordinate is output to an additional texture render target. Upon completion of collision detection this target is comprised of texture coordinates of the closest potentially colliding partner for every particle.

Although the presented approach provides an effective means for detecting many of the occurred collisions, it has some weaknesses that are worth noting. First, depending on the cell size g_0 many more particles may reside within one cell than can be checked for in the fragment program. Furthermore, because particles are sorted according to cell index only, the closest partner might be the furthest in one row. As a matter of fact, this partner will not be detected. Second, if colliding partners within one cell are arranged in consecutive rows they can not be detected. Third, due to the enumeration of grid cells collisions between particles in adjacent cells can not be detected in general. Fourth, depending on the integration time step and the speed of particles, collisions that occur between successive frames might be missed.

The first drawback can be accounted for by letting the size of grid cells being small enough to only allow for a fixed number of particles in each cell. By considering the same number of potential partners in either row direction, it is guaranteed that collisions within cells are detected.

To overcome the last drawback we essentially sort the set of particles twice. We first build a second set of sorting keys and identifiers from a staggered grid as shown in figure 3. By selecting the closest partner from both grids, only those pairs separated by cell faces in both grids can not be detected. The vast majority of collisions, however, can be determined and resolved.

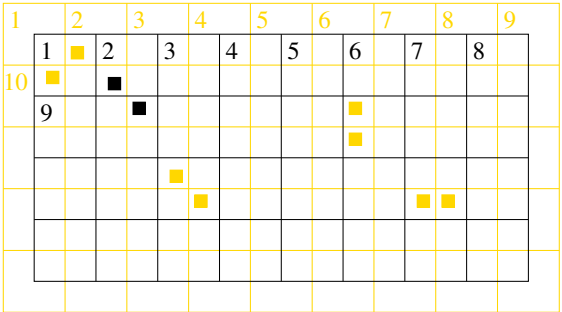

Figure 3: *Initial (orange) and staggered (black) grid used for particle enumeration are illustrated in 2D. For the shown particle pairs, collisions can not be detected in the initial grid. Only collisions between particles colored black will be missed in the staggered grid.*

4.5. Collision Response

The output of the collision detection module is used as input for the fragment program that simulates collision reactions. Of the potentially colliding partners both position and velocity can be accessed thus providing the information required to compute these reactions. Assuming sphere-like particles with a fixed radius, the fragment program computes the distance between both spheres and tests for a collision. If a collision is ascertained, both particles are backed up in time until the first point of contact is reached.

From conservation of linear momentum and energy, new momentum and thus velocity of the current particle is computed. Spherical rigid bodies can be modeled as well by using additional containers for angular momentum and rotation, and by updating these quantities according to external forces and collision impulses. Updated velocities are used to displace the current particle position according to the time interval that was taken for backtracking. Each fragment finally outputs its position and velocity into the respective textures.

4.6. Boundary Conditions

Once the changes in particle positions and velocities due to inter-particle collisions have been computed, particles are tested for collisions with static parts of the scene. Since direct collision detection between large particle and polygon sets is not feasible in general, we first scan-convert the original scene into a signed distance representation. The distance field is stored in a 3D texture map, thus enabling the fragment program to determine the distance of particles to scene geometry by interpolation in this field.

Although this approach is fairly simple to implement and only needs a single dependent texture operation to access the distance field, for high detailed models it comes with a significant overhead in texture memory. On the other hand, the method is well suited for simulating particle flow over height fields. This scenario only requires a 2D RGBA texture map to store surface normal and height. Once a particle falls below the height field, its position is reset to $p_0 + d_0/(d_0 + d_1) * (p_1 - p_0)$. In this formula, p_0, p_1 and d_0, d_1 are the previous and the current particle positions, and their distances to the surface, respectively. At the updated point position, the normal h is interpolated and the reflection vector r to model the collision response is computed.

4.7. Rendering

Once particle positions have been updated and saved in graphics memory, these positions are sent through the GPU again to obtain images in the frame buffer. Therefore, updated particle positions are rendered into a vertex array memory object, which is then processed by the geometry engine in a fixed order. The sorting order that has eventually been computed can thus be maintained. In addition to vertex geometry, attributes like color or texture coordinates can be specified in graphics memory as well.

By evaluating the time stamps of particles in a vertex program, the injection of particles can be controlled and particles can be deleted once their life time is expired. Particles that have not yet been born, i.e. their time stamp is larger than the current time step, are discarded by placing these vertices outside the view frustum. Once the particle time stamp modulo the current time step is equal to zero, all particle attributes are reset thus starting a new life cycle.

5. Performance Evaluation and Discussion

To verify the effectiveness of the described particle engine, we investigate its throughput in a variety of different scenarios. All our experiments were run under WindowsXP/Linux on a Pentium 4 2.0 GHz processor equipped with an ATI 9800Pro graphics card.

The proposed stream model for sorting on GPUs achieves a considerable speed up compared to previous approaches. Besides minimizing the number of texture operations, it exploits both the computational power of geometry and fragment processors. By hard-coding information that is required repeatedly in the sorting stages, computational load in the fragment units can be minimized.

Let us now analyze the performance of the sorting routine by sorting differently sized 2D texture maps. To sort a texture of size $n \times n$, $log(n^2) \cdot (log(n^2) + 1)/2$ rendering passes are required. In each pass, sorting keys and identifiers of respective operands are accessed, compared and rendered. In the table below, we compare different implementations of the Bitonic sort on recent ATI hardware (including the computation of sorting keys and identifiers as well as the rearrangement of particle containers): *FP* implements the entire sort in a fragment program, *GP/FP* exploits geometry units to pass hard-coded information to the fragment program, and *GP/FP Packed* packs consecutive sorting keys and identifiers in one texture element.

Table 1: Timings (fps) for sorting 2D texture maps.

	128^2	256^2	512^2	1024^2
FP	22	3.5	1	0.2
GP/FP	40	10	3	0.4
GP/FP Packed	148	43	10	2

We see a significant performance gain due to the proposed optimization schemes. Overall, the ultimate implementation computes about 600 MPixels per second, thus almost reaching the theoretical optimum of the graphics card. Even more, by restricting the sorting to only a few sets of consecutive

rows we can flexibly select the appropriate frame rate required by the particle engine. Especially in animations where particles are released in time-sequential order this feature enables real-time animations, yet considerably reducing the number of missed collisions. By holding in the same row all particles that are released in one time step, these particles can usually be sorted without sacrificing speed. Although collisions between sets of particles having different time stamps won't be detected, as long as these sets do not mix up entirely, the number of missed collision will be small.

Next, we give timing statistics for the rendering of sets of 256^2, 512^2 and 1024^2 particles. Corresponding screen shots are shown in the color plates below. We analyze the performance of the engine with regard to the simulation of different effects: collisionless particle motion (E1), collisions with a height field (E2), collisionless motion including front-to-back sorting and rendering (E3) and full inter-particle collisions (E4). Inter-particle collision detection includes sorting along texture rows only, and testing a set of 8 particles to the left and to the right of each particle in the fragment program. To compare our system to CPU-based particle engines, (E5) lists the timings for an implementation optimized for CPU processing that uses data-dependent sorting (quick-sort) for front-to-back rendering and is thus equivalent to the GPU experiment (E3).

Table 2: Animation times for large particle sets (fps).

	E1	E2	E3	E4	E5
256^2	640	155	39	133	7
512^2	320	96	8	31	2
1024^2	120	42	1.4	7	0.4

Timings in column E1 essentially show the throughput of the geometry engine combined with OpenGL SuperBuffers. Particles are rendered with associated colors and disabled z-test. A considerable loss in performance can be perceived when using memory objects smaller than 1024^2 – an indication that the graphics card can handle large chunks of data much more efficiently.

As can be seen, even when combining visibility sorting and particle-scene collision detection we are still able to run a real-time animation with about 10 frames per second for a quarter of a million particles. Obviously, animating a million particles puts some load on both the geometry and the fragment subsystem. On the other hand, by restricting the sort to texture rows, we can still perform dynamic simulation of this number of particles with some frames per second.

6. Conclusion

In this paper, we have presented the first particle engine that entirely runs on programmable graphics hardware *and* includes effects such that inter-particle collision and visibility sorting. To fully take advantage of OpenGL memory objects, efficient GPU realizations of algorithms used to perform particle manipulation have been developed. By combining memory objects with floating-point vertex and fragment programs, the system enables real-time animation and rendering of particle dynamics. At run-time, CPU-GPU transfer is completely avoided.

We believe that our work is influential for future research in the field of computer graphics due to several reasons: First, for the first time it has been shown that geometry data can be created, manipulated and rendered on the GPU. By combining vertex and fragment units, it is now possible to simultaneously use the GPU as numerical number cruncher and render server. As this approach allows avoiding any kind of data transfer between the CPU and the GPU, it will significantly speed up applications where numerical computation and rendering of large geometry data is paramount. Second, the particle engine as implemented allows integrating any physical model that requires access to adjacent particles to update particle dynamics. Thus, a variety of grid-less methods to computational simulation of physics based effects can be mapped to graphics hardware. Third, because sorting is at the core of many applications in computer graphics, e.g. occlusion culling, global illumination, unstructured grid rendering, scene graph optimization, the efficient implementation of a sorting algorithm on the render server is extremely beneficial and can be directly used to accelerate a variety of different applications.

References

[Bar89] BARAFF D.: Analytic methods for dynamic simulation of non-penetrating rigid bodies. In *ACM Computer Graphics (Proc. SIGGRAPH '89)* (1989), pp. 223–232. 5

[Bat68] BATCHER K.: Sorting networks and their applications. In *Proceedings AFIPS 1968* (1968). 4

[BFGS03] BOLZ J., FARMER I., GRINSPUN E., SCHRÖDER P.: Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. In *ACM Computer Graphics (Proc. SIGGRAPH '03)* (2003), pp. 917–924. 2

[Boo90] BOON J.: *Lattice Gas Automata: A New Approach to the Simulation of Complex Flows*. Plenum Press, 1990. 1

[BW97] BARAFF D., WITTKIN A.: Physically based modeling: Principles and practice. ACM Siggraph '97 Course Note, 1997. 1

[Doo90] DOOLEAN G. (Ed.): *Lattice Gas Methods for Partial Differential Equations*. Addison Wesley Longman, 1990. 1

[DS03] DACHSBACHER C., STAMMINGER M.:

Translucent shadow maps. In *Proceedings Eurographics Symposium on Rendering 2003* (2003). 2

[EKE01] ENGEL K., KRAUS M., ERTL T.: High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware* (2001). 2

[GRLM03] GOVINDARAJU N., REDON S., LIN M., MANOCHA D.: Cullide: Interactive collision detection between complex models in large environments using graphics hardware. In *Proceedings ACM SIGGRAPH/Eurographics Conference on Graphics Hardware* (2003). 2

[Hah88] HAHN J.: Realistic animation of rigid bodies. In *ACM Computer Graphics (Proc. SIGGRAPH '88)* (1988), pp. 173–182. 5

[HBSL03] HARRIS M., BAXTER W., SCHEUERMANN T., LASTRA A.: Simulation of cloud dynamics on graphics hardware. In *Proceedings ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware* (2003), pp. 12–20. 2

[HMG03] HILLESLAND K., MOLINOV S., GRZESZCZUK R.: Nonlinear optimization framework for image-based modeling on programmable graphics hardware. In *ACM Computer Graphics (Proc. SIGGRAPH '03)* (2003), pp. 925–934. 2

[KPHE02] KNISS J., PREMOZE S., HANSEN C., EBERT D.: Interactive translucent volume rendering and procedural modeling. In *Proceedings IEEE Visualization* (2002). 2

[KvdDP03] KNOTT D., VAN DEN DOEL K., PAI D. K.: Particle system collision detection using graphics hardware. In *SIGGRAPH 2003 Sketch* (2003). 2

[KW03] KRUEGER J., WESTERMANN R.: Linear algebra operators for GPU implementation of numerical algorithms. In *ACM Computer Graphics (Proc. SIGGRAPH '03)* (2003), pp. 908–916. 2

[LKHW03] LEFOHN A., KNISS J., HANSEN C., WHITAKER R.: Interactive deformation and visualization of level set surfaces using graphics hardware. In *Procceedings IEEE Visualization* (2003). 2

[LMS03] LI M., MAGNOR M., SEIDEL H.-P.: Hardware-accelerated visual hull reconstruction and rendering. In *Proceedings of Graphics Interface* (2003), pp. 12–20. 2

[LT93] LEECH J., TAYLOR R.: Interactive modeling using particle systems. In *Proc. 2nd Conference on Discrete Element Methods* (1993). 1

[MA03] MORELAND K., ANGEL E.: The FFT on a GPU. *Proceedings ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware* (2003), 112–119. 2

[McA00] MCALLISTER D.: The design of an api for particle systems, 2000. 1

[MGAK03] MARK W., GLANVILLE R., AKELEY K., KILGARD M.: Cg: A system for programming graphics hardware in a C-like language. In *ACM Computer Graphics (Proc. SIGGRAPH '03)* (2003), pp. 896–907. 1

[Mic02] MICROSOFT: DirectX9 SDK. http://www.microsoft.com/ DirectX, 2002. 1

[Mir00] MIRTICH B.: Timewarp rigid body simulation. In *ACM Computer Graphics (Proc. SIGGRAPH '00)* (2000), pp. 193–200. 5

[Mon88] MONAGHAN J.: An Introduction to SPH. *cpc 48* (1988), 89–96. 1

[PDC*03] PURCELL T., DONNER C., CAMMARANO M., JENSEN H., HANRAHAN P.: Photon mapping on programmable graphics hardware. In *Proceedings ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware* (2003), pp. 41–50. 2, 4

[Ree83] REEVES T.: Particle systems - a technique for modelling a class of fuzzy objects. *ACM Computer Graphics (Proc. SIGGRAPH '83)* (1983). 1

[RSEB*00] REZK-SALAMA C., ENGEL K., BAUER M., GREINER G., T. E.: Interactive volume rendering on standard pc graphics hardware using multi-textures and multi-stage rasterization. In *Eurographics Workshop on Graphics Hardware* (2000), pp. 109–119. 2

[SHN03] SHERBONDY A., HOUSTON M., NAPEL S.: Fast volume segmentation with simultaneous visualization using programmable graphics hardware. In *Procceedings IEEE Visualization* (2003). 2

[Sim90] SIMS K.: Particle animation and rendering using data parallel computation. In *Computer Graphics (Siggraph '90 proceedings)* (1990), pp. 405–413. 1

[YWB03] YANG R., WELCH G., BISHOP G.: Real-time consensus-based scene reconstruction using commodity graphics hardware. In *Proceedings of Pacific Graphics* (2003), pp. 23–31. 2

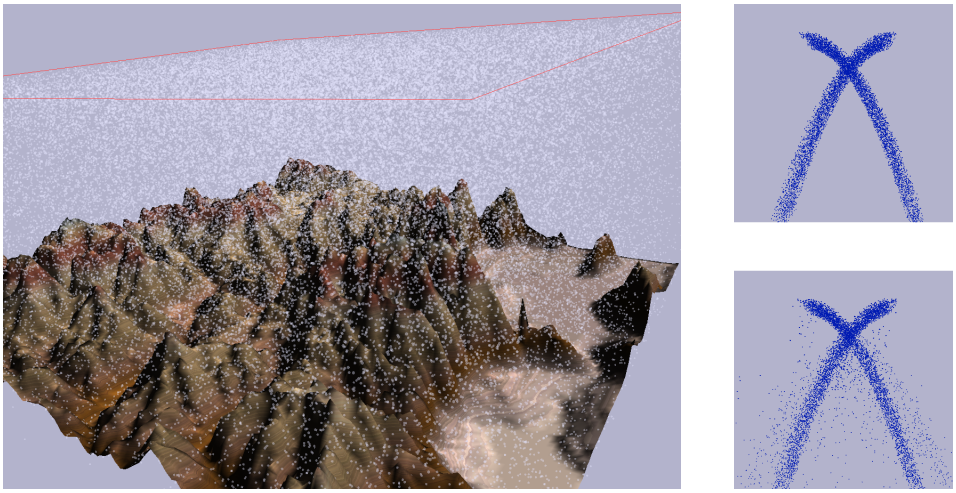

Figure 4:
LEFT IMAGE: *Dense snow falling down on a landscape. Includes detection of ground contact.*
RIGHT IMAGES: *Effect of inter-particle collision (off at the top, on at the bottom).*

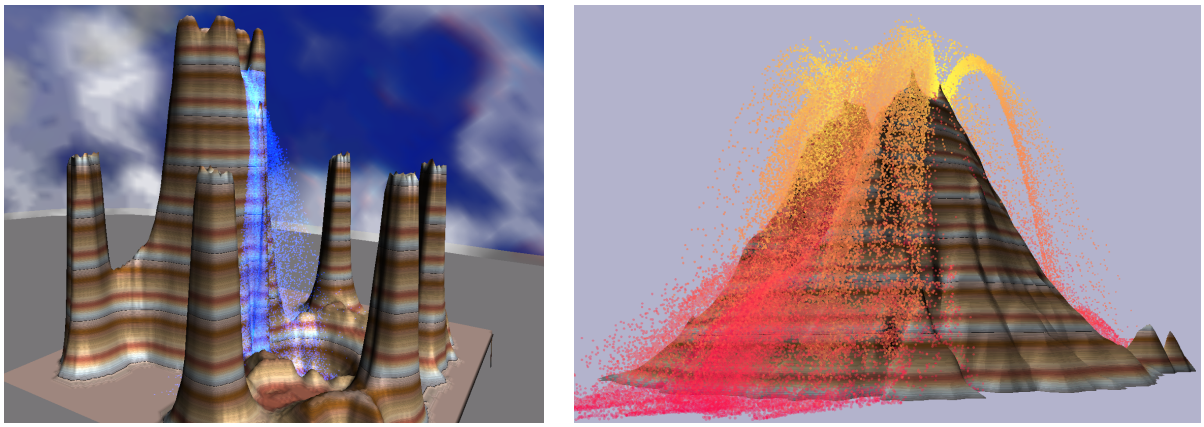

Figure 5: *Tracing large numbers of particles including particle-scene collision detection and front-to-back sorting to model natural phenomena. The point primitives can be rendered using any OpenGL functionality available.*