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

Hardware-based Simulation and Collision Detection for Large Particle Systems

A. Kolb* and L. Latta† and C. Rezk-Salama*

*Computer Graphics and Multimedia Systems Group, University of Siegen, Germany
†2L Digital, Mannheim, Germany

Abstract

Particle systems have long been recognized as an essential building block for detail-rich and lively visual environments. Current implementations can handle up to 10,000 particles in real-time simulations and are mostly limited by the transfer of particle data from the main processor to the graphics hardware (GPU) for rendering.
This paper introduces a full GPU implementation using fragment shaders of both the simulation and rendering of a dynamically-growing particle system. Such an implementation can render up to 1 million particles in real-time on recent hardware. The massively parallel simulation handles collision detection and reaction of particles with objects for arbitrary shape. The collision detection is based on depth maps that represent the outer shape of an object. The depth maps store distance values and normal vectors for collision reaction. Using a special texture-based indexing technique to represent normal vectors, standard 8-bit textures can be used to describe the complete depth map data. Alternately, several depth maps can be stored in one floating point texture.
In addition, a GPU-based parallel sorting algorithm is introduced that can be used to perform a depth sorting of the particles for correct alpha blending.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Graphics processors I.3.5 [Computer Graphics]: Boundary representations I.3.7 [Computer Graphics]: Animation

1. Introduction

Physically correct particle systems (PS) are designed to add essential properties to the virtual world. Over the last decades they have been established as a valuable technique for a variety of applications, e.g. deformable objects like cloth [VSC01] and volumetric effects [Har03].

Dynamic PS have been introduced by [Ree83] in the context of the motion picture Star Trek II. Reeves describes basic motion operations and basic data representing a particle - both have not been altered much since. An implementation on parallel processors of a super computer has been done by [Sim90]. [Sim90] and [McA00] describe many of the velocity and position operations of the motion simulation also used in our PS.

Real-time PS are often limited by the fill rate or the CPU to graphics hardware (GPU) communication. The fill rate is often a limiting factor when there is a high overdraw due to relatively large particle geometries. Using a large number of smaller particles decreases the overdraw and the fill rate limitation looses importance. The bandwidth limitation now dominates the system. Sharing the graphics bus with many other rendering tasks allows CPU-based PS to achieve only up to 10,000 particles per frame in typical real-time applications. A much larger number of particles can be used by minimizing the amount of communication of particle data by integrating simulation and rendering on the GPU.

Stateless PS, i.e. all particle data can be computed by closed form functions based on a set of start values and the current time, have been implemented using vertex shaders (cf. [NVI03]). However, state-preserving PS can utilize numerical, iterative integration methods to compute the particle data from previous values and a dynamically changing environment. They can be used in a much wider range of applications.

While collision reaction for particles is a fairly simple application of Newtonian physics, collision detection can be a rather complex task w.r.t. the geometric representation of the collider object. [LG98] gives a good overview on collision detection techniques for models represented as CSG, polygonal, parametric or implicit surfaces. There are three basic *image-based* hardware accelerated approaches to collision detection based on depth buffers, stencil buffers or occlusion culling. However, all these techniques use the GPU to generate spatial information which has to be read back from the GPU to the CPU for further collision processing.

The technique presented in this paper uses the "stream processing" paradigm, e.g. [PBMH02], to implement PS simulation, collision detection and rendering completely on the fragment processor. Thus a large number of particles can be simulated using the state-preserving approach. The collision detection is based on an implicit, image based object boundary representation using a sequence of depth maps similar to [KJ01]. Several approaches are presented to store one, two or six depth maps in a single texture. Storing the normal vectors for collision reaction is realized using a texture-based normal indexing technique.

The remainder of this paper is organized as follows. Section 2 gives an overview over works related to this paper. The GPU based simulation for large particle systems is described in section 3. Collision detection on the GPU is discussed in section 4. Results and conclusions are given in sections 5 and 6 respectively.

2. Prior work

This section describes prior works related to particle systems (PS) and their implementation using graphics hardware (section 2.1). Additionally, we briefly discuss collision detection approaches (section 2.2), techniques to generate implicit representations for polygonal objects (section 2.3) and the compression of normal vectors (section 2.4).

2.1. Stateless particle systems

Some PS have been implemented with vertex shaders on programmable GPUs [NVI03]. However, these PS are stateless, e.g. they do not store the current positions of the particles. To determine a particle's position a closed form function for computing the current position only from initial values and the current time is needed. As a consequence such PS can hardly react to a dynamically changing environment.

Particle attributes besides velocity and position, e.g. the particle's orientation, size and texture coordinates, have generally much simpler computation rules, e.g. they might be calculated from a start value and a constant factor of change over time.

So far there have been no state-preserving particle systems fully implemented on the GPU.

2.2. Collision detection techniques

The field of collision detection is one of the most active in recent years. Lin and Gottschalk [LG98] give a good overview on various collision detection techniques and a wide range of applications, e.g. game development, virtual environments, robotics and engineering simulation.

There are three basic hardware accelerated approaches based on depth buffers, stencil buffers and occlusion culling. All approaches are image based and thus their accuracy is limited due to the discrete geometry representation.

Stencil buffer and depth buffer based approaches like [BW03, HZLM02, KOLM02] use the graphics hardware to generate proximity, collision or penetration information. This data has to be read back to the CPU to perform collision detection and reaction. Usually, these techniques use the graphics hardware to detect pairs of objects which are potentially colliding. This process may be organized hierarchically to get either more detailed information or to reduce the potentially colliding objects on a coarser scale.

Govindaraju et.al. [GRLM03] utilize hardware accelerated occlusion queries. This minimizes the bandwidth needed for the read-back from the GPU. Again, the collision reaction is computed on the CPU.

2.3. Implicit representation of polygonal objects

Implicit representations of polygonal objects have advantages in the context of collision detection, since the distance of any 3D-point is directly given by the value of the implicit model representation, the so-called *distance-map*.

Nooruddin and Turk [NT99, NT03] introduced a technique to convert a polygonal model in an implicit one using a scanline conversion algorithm. They use the implicit representation to modify the object with 3D morphological operators.

Kolb and John [KJ01] build upon Nooruddin and Turk's algorithm using graphics hardware. They remove mesh artifacts like holes and gaps or visually unimportant portions of objects like nested or overlapping parts. This technique generates an approximate distance map of a polygonal model, which is exact on the objects surface w.r.t. to the visibility of object points and the resolution of the depth buffer.

2.4. Compression of normal vectors

For collision reaction, an efficient way to store an object's normal, i.e. the collision normal, at a particular point on the object's surface is needed. Deering [Dee95] notes, that angular differences below 0.01 radians, yielding some 100k normal vectors, are not visually recognizable in rendering. Deering introduces a normal encoding technique which requires several trigonometric function calls per normal.

In our context we need a normal representation technique which is space and time efficient. Possible decoding of the vector components must be as cheap as possible, while the encoded data must be efficiently stored in textures.

Normal maps store normal vectors explicitly and are not space efficient. Applying an optional DXT-compression results in severe quantization artifacts (cf. [ATI03]).

Sphere maps, cube maps and parabolic maps, commonly used to represent environmental information, may be used to store normal vectors. Sphere maps heavily depend on a specific viewing direction. Cube maps build upon a 3D-index, i.e. a point position in 3-space. Parabolic maps need two textures to represent a whole sphere. Additionally, they only utilize the inscribed circle of the texture.

2.5. Other related works

Green [Gre03] describes a cloth simulation using simple grid-aligned particle physics, but does not discuss generic particle systems' problems, like allocation, rendering and sorting of PS. The photon mapping algorithm described by Purcell et.al. [PDC*03] uses a sorting algorithm similar to the odd-even merge sort presented in section 3.3.3. However, their algorithm does not show the necessary properties to exploit the high frame-to-frame coherence of the particle system simulation.

3. Particle simulation on Graphics Hardware

The following sections describe the algorithm of a state-preserving particle system on a GPU in detail. After a brief overview (section 3.1), the storage (section 3.2) and then the processing of particles is described (section 3.3).

3.1. Algorithm Overview

The particle simulation consists of six basic steps:

1. Process birth and death
2. Update velocities
3. Update positions
4. Sort for alpha blending (optional)
5. Transfer texture data to vertex data
6. Render particles

The state-preserving particle system stores the velocities and positions (step 2. and 3.) of all particles in textures, which are also render targets. In one rendering pass the texture with particle velocities is updated, performing a single time step of an iterative integration. Here acceleration forces and collision reactions are applied. A second rendering pass updates the position textures in a similar way, using the velocity texture. Depending on the integration method it is possible to skip the velocity update pass, and directly integrate the position from accelerations (cf. section 3.3.2).

Optionally, in step 4. the particle positions can be sorted depending on the viewer distance to avoid rendering artifacts. The sorting performs several additional rendering passes on textures that contain the particle-viewer distance and a reference to the particle itself.

Then the particle positions are transferred from the position texture to a vertex buffer and the particles are rendered as point sprites, triangles or quads.

3.2. Particle data storage

The positions and velocities of all active particles are stored in floating point textures using the three color components as x, y and z coordinates. The texture itself is also a render target, so it can be updated with the computed positions and velocities. Since a texture cannot be used as input and output at the same time, we use a pair of these textures and a double buffering technique (cf. figure 1). Depending on the integration algorithm the explicit storage of the velocity texture can be omitted (cf. section 3.3.2).

Other particle attributes like mass, orientation, size, color, and opacity are typically static or can be computed using a simple stateless approach (cf. section 2.1). To minimize the upload of static attribute parameters we introduce particle types. Thus the simulation of these attributes uses one further texture to store the time of birth and a reference to the particle type for each particle (cf. figure 1). To model more complex attribute behavior, simple key-frame interpolation over the age of the particle can be applied.

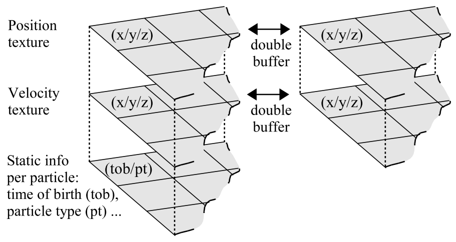

Figure 1: *Data storage using double buffered textures*

3.3. Simulation and rendering algorithm

3.3.1. Process birth and death

Assuming a varying number of short-living particles, the particle system must be able to process the birth of a new particle (*allocation*) and the death of a particle (*deallocation*).

Since allocation problems are serial by nature, this cannot be done efficiently with a data-parallel algorithm on the GPU. Therefore an available particle index is determined on the CPU using either a stack filled with all available indices or a heap data structure that is optimized to always return the smallest available index. The heap data structure guarantees that particles in use remain packed in the first portion of the textures. The following simulation and rendering steps only need to update that portion of data, then. The initial particle data is determined on the CPU and can use complex algorithms, e.g. probability distributions (cf. McAllister [McA00]).

A particle's death is processed independently on the CPU and GPU. The CPU registers the death of a particle and adds the freed index to the allocator. The GPU does an extra pass to determine the death of a particle by the time of birth and the computed age. The dead particle's position is simply moved to invisible areas. As particles at the end of their lifetime usually fade out or fall out of visible areas anyway, the extra clean-up pass rarely needs to be done.

3.3.2. Update velocities and position

Updating a particle's velocity and position is based on the Newtonian laws of motion. The actual simulation is implemented in a fragment shader. The shader is executed for each pixel of the render target by rendering a screen-sized quad. The double buffer textures are alternately used as render target and as input data stream, containing the velocities and positions from the previous time step.

There are several operations that can be used to manipulate the velocity (cf. [Sim90] and [McA00] for more details): global forces (e.g. gravity, wind), local forces (attraction, repulsion), velocity dampening, and collision responses. For our GPU-based particle system these operations need to be parameterized via fragment shader constants.

A complex local force can be applied by mapping the particle position into a 2D or 3D texture containing flow velocity vectors $\vec{\mathbf{v}}_{\text{flow}}$. Stoke's law is used to derive a dragging force:

$$\vec{\mathbf{F}}_{\text{flow}} = 6\pi\eta\, r(\vec{\mathbf{v}}_{i-1} - \vec{\mathbf{v}}_{\text{flow}})$$

where η is the flow viscosity, r the particle radius and $\vec{\mathbf{v}}_{i-1}$ the particle's previous velocity.

The new velocity $\vec{\mathbf{v}}_i$ and position $\mathbf{P}$ is derived from the accumulated global and local forces $\vec{\mathbf{F}}$ using simple Euler integration.

Alternatively, Verlet integration (cf. [Ver67]) can be used to avoid the explicit storage of the velocity by utilizing the position information $\mathbf{P}_{i-2}$. The great advantage is that this technique reduces memory consumption and removes the velocity update rendering pass.

Verlet integration uses a position update rule based only on the acceleration:

$$\mathbf{P}_i = 2\mathbf{P}_{i-1} - \mathbf{P}_{i-2} + \vec{\mathbf{a}}\Delta_i^2$$

Using Euler integration, collision reaction is based on a change of the particle's velocity. Splitting the velocity vector $\vec{\mathbf{v}}_i$ into a normal component $\vec{\mathbf{v}}_i^{\perp}$ and a tangential component $\vec{\mathbf{v}}_i^{\parallel}$ the velocity after the collision can be computed applying friction ν and resilience ε:

$$\vec{\mathbf{v}}_i = (1-\nu)\vec{\mathbf{v}}_i^{\parallel} - \varepsilon\vec{\mathbf{v}}_i^{\perp}$$

To avoid velocities from slowing down close to zero, the friction slow-down should not be applied if the overall velocity is smaller than a given threshold.

Having collider with sharp edges, e.g. a height field, or two colliders close to each other, the collision reaction might push particles into a collider. In this case a caught particle ought to be pushed out in the next simulation step.

To handle this situation, the collision detection is done twice, once with the previous and once with the expected position $\mathbf{P}_i^*$ based on velocity $\vec{\mathbf{v}}_i^*$. This allows differentiating between particles that are about to collide and those having already penetrated (cf. figure 2). The latter must be pushed in direction of the shortest way out of the collider. This direction can be guessed from the normal component of the velocity:

$$\vec{\mathbf{v}}_i = \begin{cases} \vec{\mathbf{v}}_i^* & \text{if } \left(\vec{\mathbf{v}}_i^* \cdot \hat{\mathbf{n}}\right) \geq 0 \quad \text{(heading outside)} \\ \vec{\mathbf{v}}_i^{\perp} - \vec{\mathbf{v}}_i{}^* & \text{if } \left(\vec{\mathbf{v}}_i^* \cdot \hat{\mathbf{n}}\right) < 0 \quad \text{(heading inside)} \end{cases}$$

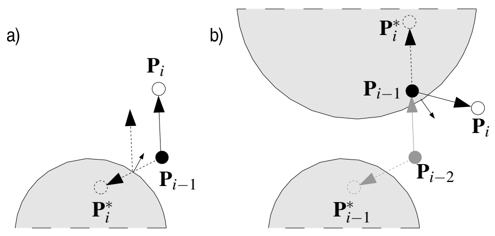

Figure 2: *Particle collision: a) Reaction before penetration; b) Double collision with caught particle and push-back.*

Verlet integration cannot directly handle collision reactions in the way discussed above. Here position manipulations are required to implicitly change the velocity in the following frames.

3.3.3. Sorting

If particles are blended using a non-commutative blending mode, a depth-based sorting should be applied to avoid artifacts.

A particle system on the GPU can be sorted quite efficiently with the parallel sorting algorithm "odd-even merge sort" (cf. Batcher [Bat68]). Its runtime complexity is independent of the data's sortedness. Thus, a check whether all data is already in sequence does not need to be performed on the GPU, which would be rather inefficient. Additionally, with each iteration the sortedness never decreases. Thus, using the high frame-to-frame coherence of the particle order, the whole sorting sequence can be distributed over 20 - 50 frames. This, of course, leads to an approximate depth sortedness, which, in our examples, does not yield any visual artifacts.

The basic principle of odd-even merge sort is to divide the data into two halves, to sort these and then to merge the two halves. The algorithm is commonly written recursively, but a closer look at the resulting sorting network reveals its parallel nature. Figure 3 shows the sorting network for eight values. Several consecutive comparisons are independent of each other and can be grouped for parallel execution (vertical lines in figure 3).

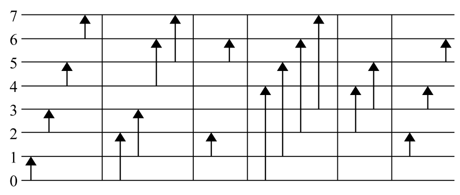

Figure 3: *Odd-Even sorting network for eight values; arrows mark comparison pairs.*

The sorting requires $\frac{1}{2}\log_2^2 n + \frac{1}{2}\log_2 n$ passes, where n is the number of elements to sort. For a 1024×1024 texture this leads to 210 rendering passes. Running all 210 passes each frame is far too expensive, but spreading the whole sorting sequence over 50 frames, i.e. 1 - 2 seconds, reduces the workload to $4-5$ passes each frame.

The sorting algorithm requires an additional texture containing the particle-viewer distance. The distance in this texture is updated after the position simulation. After sorting the rendering step looks up the particle attributes via the index in this texture.

3.3.4. Render particles

The copying of position data from a texture to vertex data is an upcoming hardware feature in PC GPUs. DirectX and OpenGL offer the vertex textures technique (vertex shader 3.0 rsp. `ARB_vertex_shader` extension). Unfortunately there is no hardware supporting this feature at the moment.

Alternatively "über-buffers" (also called super buffers; cf. [Per03]) can be used. This functionality is already available in current GPUs, but up to now it is only supported by the OpenGL API. The current implementation uses the vendor specific `NV_pixel_data_range` extension (cf. [NVI04]).

The transferred vertex positions are used to render primitives to the frame buffer. In order to reduce the workload of the vertex unit, particles are currently rendered as point sprites instead of as triangles or quads. The disadvantage though is that particles are always axis-aligned. To allow a 2D-rotation, texture coordinates are transformed in the fragment shader.

4. Collision detection

In this section we describe the implicit object representation used for collision detection (section 4.1). Furthermore, the normal indexing technique (section 4.2) and various approaches to represent depth maps in textures are introduced (section 4.3).

4.1. Implicit model representation

We use an image-based technique similar to [KJ01] to represent an objects outer boundary by a set of depth maps. These depth maps contain the distance to the object's boundary and the surface normal at the relevant object point.

Each depth map DM_i, $i = 1,\ldots,k$ stores the following information:

1. distance $\mathrm{dist}(x,y)$ to the collider object in projection direction for each pixel (x,y)
2. normal vector $\hat{\mathbf{n}}(x,y)$ at the relevant object surface point
3. transformation matrix $T_{OC\to DC}$ mapping from collider object coordinates to depth map coordinates, i.e. the coordinate system in which the projection was performed
4. z_{scale} scaling value in z-direction to compensate for possible scaling performed by $T_{OC\to DC}$

The object's interior is assigned with negative distance values. Assuming we look from outside onto the object and orthographic projection is used, the distance value $f(\mathbf{P})$ for a point $\mathbf{P}$ is computed using the transformation $T_{OC\to DC}$:

$$f(\mathbf{P}) = z_{scale} \cdot \left(\mathrm{dist}(p'_x, p'_y) - p'_z\right), \qquad (1)$$
$$\text{where } \mathbf{P}' = (p'_x, p'_y, p'_z)^T = T_{OC\to DC}\mathbf{P}$$

$T_{OC\to DC}$ usually also contains the transformation to texture coordinates. Thus fetching the depth value $\mathrm{dist}(p'_x, p'_y)$ is a simple texture lookup at coordinates (p'_x, p'_y).

Taking several depth maps into account, the most appropriate depth for point $\mathbf{P}$ has to be determined. The following definition guarantees that $\mathbf{P}$ is outside of the collider if at least one depth map has recognized $\mathbf{P}$ to be exterior:

$$f(\mathbf{P}) = \begin{cases} \max\{f_i(\mathbf{P})\} & \text{if } f_i(\mathbf{P}) < 0\ \forall i \\ \min\{f_i(\mathbf{P}) \ : \ f_i(\mathbf{P}) > 0\} & \text{else} \end{cases}$$

where $f_i(\mathbf{P})$ denotes the signed distance value w.r.t. depth map DM_i.

Handling several depth maps DM_i, $i = 1,\ldots,k$, $f(\mathbf{P})$ can be computed iteratively:

$$\left.\begin{array}{l} (f(\mathbf{P}) < 0 \wedge f_i(\mathbf{P}) > f(\mathbf{P})) \\ \vee \\ (f_i(\mathbf{P}) > 0 \wedge f_i(\mathbf{P}) < f(\mathbf{P})) \end{array}\right\} \Rightarrow (f(\mathbf{P}) \leftarrow f_i(\mathbf{P})) \qquad (2)$$

where $f(\mathbf{P})$ is initially set to a large negative value, i.e. $\mathbf{P}$ is placed "far inside" the collider.

If $\mathbf{P}' = T_{OC \to DC}\mathbf{P}$ lies outside the view volume for the current depth map or the texture lookup $\mathrm{dist}(p'_x, p'_y)$ results in the initial background value, e.g. no distance to the object can be computed, this value may be ignored as *invalid vote*. Alternatively, if the view volume encloses the complete collider object, the invalid votes can be declared as "far outside". To avoid erroneous data due to clamping of (p'_x, p'_y), we simply keep a one pixel border in the depth map unchanged with background values to indicate an invalid vote.

A fragment shader computes the distance using rule (2) and applies a collision reaction when the final $f(\mathbf{P})$ distance is negative.

Note, that this approach may have problems with small object details in case of an insufficient buffer resolution. Problems can also be caused by local surface concavities. In many cases, these local concavities can be reconstructed with properly placed depth map view volumes (cf. section 5).

4.2. Quantization and indexing of normal vectors

Explicitly storing normal vectors using $8, 16$ or 32 bit per component is sufficient within certain error bound (cf. [ATI03]). Since we store unit vectors, most of the used 3-space remains unused, though. The depth map representation requires a technique which allows the retrieval of normal vectors using a space- and time efficient indexing method. Indices must be encoded in depth maps and the reconstruction of the normal vector in the fragment shader must be efficient.

The normal representation, which is implemented by indexing into a normal index texture, should have the following properties:

1. the complete coordinate space $[0,1]^2$ of a single texture is used
2. decoding and ideally encoding is time efficient
3. sampling of the directional space is as regular as possible

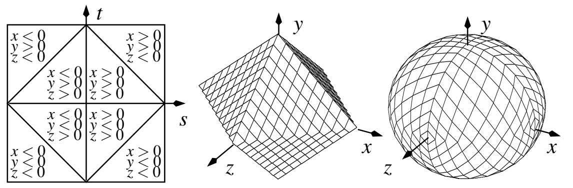

Figure 4: *The eight octants of the L_1-parametrization (left), the octahedron after applying l_1 (middle) and the sampling of the unit sphere (right).*

Cube maps can not be used, since the index to look-up the function value is already a vector with three components. Sphere maps, commonly used as reflection maps, heavily depend on a specific direction, e.g. the viewing direction for the reflection computation. On the other hand, parabolic maps show a very uniform parameterization of the hemi-sphere (cf. Heidrich and Seidel [HS98]), but two of these textures are needed to span the whole sphere.

We propose the following mapping, which is based on the L_1-*norm*: $\|\vec{\mathbf{v}}\|_1 = |v_x| + |v_z| + |v_z|$:

$$l_1(s,t) = \begin{cases} \begin{pmatrix} s \\ t \\ 1-|s|-|t| \end{pmatrix} & \text{if } |s|+|t| \le 1 \\ \begin{pmatrix} \mathrm{sgn}(s)(1-|t|) \\ \mathrm{sgn}(t)(1-|s|) \\ 1-|s|-|t| \end{pmatrix} & \text{if } |s|+|t| > 1 \end{cases} \tag{3}$$

where $s,t \in [-1,1]$. l_1 maps $(s,t) \in [-1,1]^2$ onto the L_1-unit sphere, i.e. the unit octahedron (cf. figure 4). Applying an additional affine transformation, we get a continuous mapping of the standard texture-space $(s,t) \in [0,1]^2$ onto the octahedron. The resolution of the texture-space naturally implies the resolution of the sphere, i.e. the normal space.

It should be pointed out, that the L_1-parametrization proposed above can easily be used to represent any directional data, e.g. reflection maps.

4.3. Depth map representation

Ideally, we want to encode as many depth values and normal vectors as possible into a single texture, thus keeping the amount of data to be transfered and kept in the graphics hardware memory as small as possible.

Throughout our experiments, we have investigated the following depth map formats:

Floating point depth map
The simplest, but most storage ineffient variant uses a floating point texture to store the surface normals uncompressed in the R, G, B-components and the alpha-channel to hold the distance value.

8-bit fixed point depth map
This variant uses a standard *RGBA*-texture with 8 bit per component. Here the R, G-components contain the index into the normal texture, whereas G, A store the depth value, thus having a depth-resolution of 16-bit fixed point. The normal index texture with resolution 256×256 is build using the L_1-parametrization technique described in section 4.2. The *RGB*-components of this texture store the normal vectors, which are looked up using the index stored in the depth map.

16-bit floating point depth map (front-back)
Combining orthographic projection with depth compare function `LESS` generates a *front* depth map. Naturally the depth map taking the inverse z-direction (depth compare

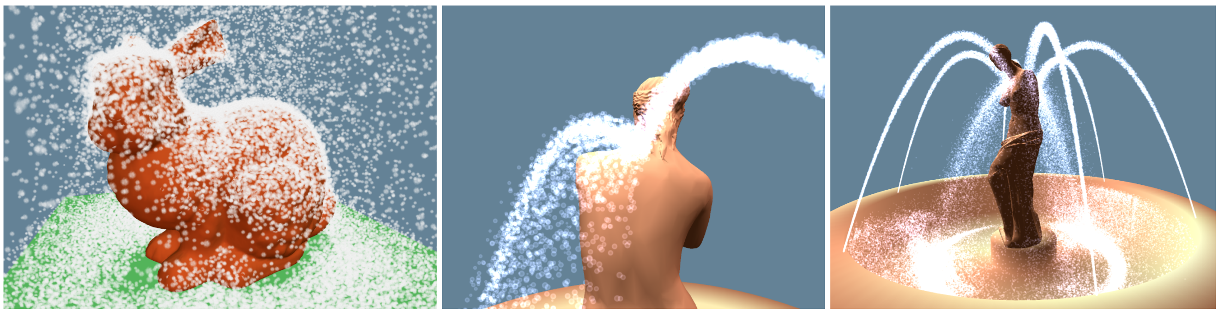

Figure 5: *Sample applications: "bunny in the snow" (left) and "Venus-fountain" (middle and right)*

function `GREATER`) is a usefull *back* depth map counterpart. We can easily represent both of these depth maps in one 16-bit texture. Here the R,G components store the normal texture indices, where two 8 bit indices are packed into a single 16-bit float. The B,A components store the depth value for the front and back map respectively.

8-bit fixed point cubic depth maps ("depth cube")
Another variant is to use cube maps to represent depth maps. In this case perspective projection w.r.t. the view volume center is applied. The depth map representation is analog to the 8-bit fixed point variant.

Generally, the different types of depth maps can be combined for collision detection within a single fragment shader, in order to utilize the advantages of the various types for the specific collider object (cf. section 5.2).

The depth cube variant uses perspective projection, whereas the other variants use orthographic projection only. Using perspective projection during the depth map generation distorts the z-values. To avoid this, the vertices of the collider object are passed w.r.t. the normalized view volume $[-1,1]^3$ to the fragment shader. The shader simply uses this information to compute the depth values $\text{dist}(x,y)$ relative to the center of the view volume.

To compute the distance value for a point $\mathbf{P} \in \mathbb{R}^3$ w.r.t. the depth cube, the transformation $T_{OC \to DC}$ in depth map coordinates (cf. section 4.1) does not contain the perspective projection. $T_{OC \to DC}$ transforms into the normalized view volume $[-1,1]^3$ only, thus picking the corresponding depth value is just a cube map texture lookup

$$\text{dist}(p'_x, p'_y, p'_z), \ \mathbf{P}' = T_{OC \to DC}\mathbf{P}$$

The distance value for a point $\mathbf{P} \in \mathbb{R}^3$ is computed as

$$f(\mathbf{P}) = \left(1 - \frac{\text{dist}(p'_x, p'_y, p'_z)}{\|\mathbf{P}'\|}\right) \left\| \begin{pmatrix} s_x \cdot p'_x \\ s_y \cdot p'_y \\ s_z \cdot p'_z \end{pmatrix} \right\|, \ \mathbf{P}' = T_{OC \to DC}\mathbf{P}$$

where s_x, s_y, s_z are the scaling factors from the normalized view volume $[-1,1]^3$ to the view volume's extends in collider object coordinates.

The placement of the depth cube w.r.t. the collider object specifies the depth compare function for the generation of the depth maps. In the default situation, where the view volume's center is outside the object, the depth compare function `LESS` is used. Otherwise, if the center is inside the collider object, `GREATER` is applied and the fragment shader which computes the distance has to negate the distance value (cf. equation 1).

5. Results

Several tests have been made with different setups for the particle system, e.g. number of particles, sorting, complexity of collider etc. We discuss the general particle system (PS) issues in section 5.1 and describe different aspects on collision dection in section 5.2. Section 5.3 gives some hardware aspects.

The presented particle system was implemented in *Cg* and tested on *NVIDIA Geforce FX 5900 XT* graphics hardware, which represents the first generation of floating-point GPUs.

5.1. Particle simulation

Using a position texture of size 1024×1024, our PS is capable of simultaneously rendering and updating a maximum of 1 million particles at about 10 frames per second. This implementation uses Euler integration, point sprites for rendering, no depth sorting and no collision detection.

In a typical application a particle texture of size 512×512 can be rendered in real-time (about 15 fps) including depth sorting (5 passes per frame) and collision detection (one depth cube). Performance measurement for the fully-featured collision detection is given in the next section. Following the clear trend towards increasing parallelism, a significant performance enhancement is expected with the forthcoming second generation of floating-point GPUs.

Figure 5 shows two sample applications using a quarter million particles. In example "bunny in the snow" each particle is rendered as a snow flake, i.e. the particles velocity is set to zero after a collision has been detected. The collision

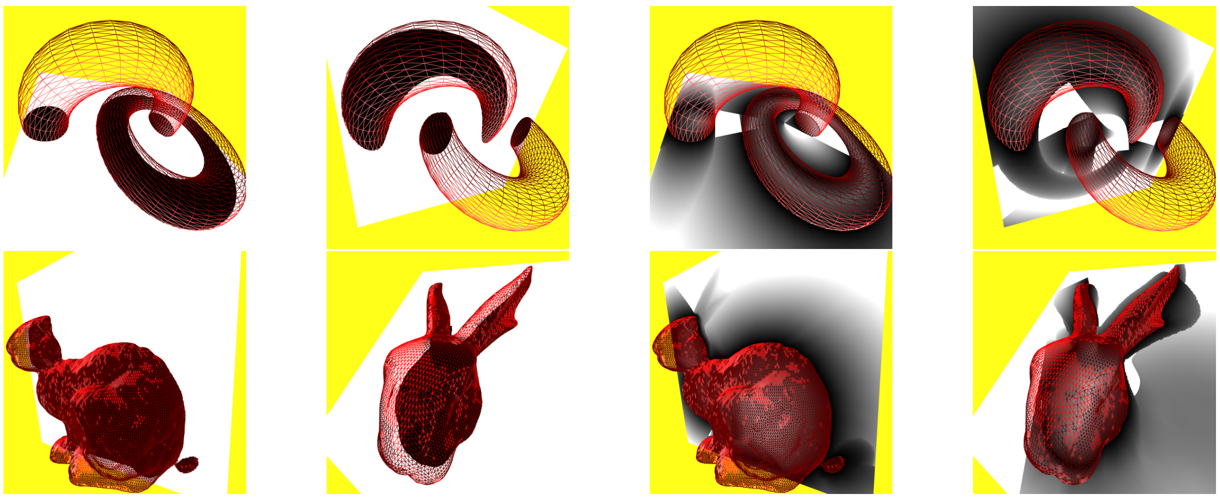

Figure 6: *Visualization of the implicit torii model (top row) and the implicit bunny model (bottom row) along slices: Interior/exterior classification (left) and approximate distance map plotting absolute distance values (right). Additionally, the wireframe model is rendered.*

detection uses one depth cube and one 16-bit front-back texture. The simulation uses depth sorting and runs with 15 fps. The second example, the "Venus fountain", also simulates 512^2 particles. The implicit object boundary is represented using three 16-bit front-back textures and one 8-bit fixed point texture. This examples runs with 10 fps.

5.2. Depth map based collision detection

The main difference between the above presented depth map formats lies in the number of depth maps used and in the projection type. In situations, where collisions occur only from one direction and the collider object has a rather simple shape, a single 8-bit fixed point depth map may result in a proper interior/exterior classification. If there is no restriction to the potential collision direction, the complete collider object has to be reconstructed. Here, either one depth cube or three orthogonal 16-bit front-back textures are used. Concave regions may have to be treated using additional depth maps.

Concerning distance values, depth cubes work well for sphere-like collider objects (cf. figure 6). If the model has many concavities, is strongly twisted or is partly nested, the reconstruction of the distance values based on depth maps leads only to coarse approximations.

In our experiments we use six to 15 depth maps to represent the collider object boundary without restriction to the potential collision direction. Testing a quarter million particles for collision takes $7, 9$ and 12 ms using the 8-bit fixed, the depth cube or the 16-bit front-back format respectively.

We mainly made experiments with rigid objects, thus performing the depth map generation in a preprocessing step. Some tests have been made with deformable objects, forcing the depth map generation to be part of the simulation process. The generation of a complete depth map format with resolution 512^2 takes about $11, 17$ and 26 ms using the 8-bit fixed, 16-bit front-back or the depth cube format respectively. Thus deformable objects should be applied only in combination with a small number of depth maps or particles.

Figure 6 visualizes the depth maps for two models: Two torii and the Stanford bunny. The torii are reconstructed using two depth cubes and two 16-bit front-back textures, giving 16 depth maps in total. The bunny is captured using one depth cube and one 16-bit front-back textures, giving 8 depth maps in total.

5.3. Hardware aspects

We use a standard normal index texture with resolution 256^2. Even though the L_1-parametrization would allow any application specific resolution, the handling of n bit integers or floats in the graphics hardware is hardly possible. Currently we use NVIDIA's pack/unpack functionality, which allows the packing of four bytes in one 32-bit float, for example. We would highly appreciate more functionality of this kind, e.g. to pack $5, 5, 6$-bits in a 16-bit float.

Additionally, improved integer arithmetic and modulo operators would simplify the implementation of various shader functionality, e.g. the parallel sorting.

6. Conclusions and future work

A fully GPU based approach to realize the simulation and collision detection for large particle systems (PS) has been introduced. The simulation of PS is based on the "stream processing" paradigm, using textures to represent all data necessary to implement a state-preserving PS and collison detection using fragment shaders. The collision detection is based on depth maps. These are used to reconstruct an implicit model of the collider objects at the time of collision detection for particles. A novel technique to represent directional data was introduced and applied to store normal vectors using an indexing technique. When rendering the PS a parallel sorting algorithm can be applied to keep a proper particle order for non-commutative blending modes.

The proposed L_1-parametrization should be investigated further, especially its applicability to represent directional data, e.g. reflection maps. Additionally, investigations towards GPU based collision detection using polygons or more complex objects instead of particles should be made.

References

[ATI03] ATI TECHNOLOGIES INC.: *Normal map compression*. Tech. rep., ATI Technologies Inc., 2003. http://www.ati.com/developer/techpapers.html. 3, 6

[Bat68] BATCHER K.: Sorting networks and their applications. In *Spring Joint Computer Conference, AFIPS Proceedings* (1968), pp. 307–314. 4

[BW03] BACIU G., WONG S.-K.: Image-based techniques in a hybrid collision detector. In *IEEE Trans. on Visualization and Computer Graphics* (2003), vol. 9, pp. 254–271. 2

[Dee95] DEERING M.: Geometry compression. In *ACM Proceedings SIGGRAPH* (1995), vol. 14, pp. 13–20. 2

[Gre03] GREEN S.: Stupid opengl shader tricks. http://developer.nvidia.com/docs/IO/8230/GDC2003_OpenGLShaderTricks.pdf, 2003. 3

[GRLM03] GOVINDARAJU N., REDON S., LIN M., MANOCHA D.: Cullide: interactive collision detection between complex models in large environments using graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (2003), Eurographics Association, pp. 25–32. 2

[Har03] HARRIS M.: *Real-Time Cloud Simulation and Rendering*. PhD thesis, Department of Computer Science, University of North Carolina at Chapel Hill, 2003. 1

[HS98] HEIDRICH W., SEIDEL H.-P.: View-independent environment maps. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (1998), ACM Press, pp. 39–45. 6

[HZLM02] HOFF K., ZAFERAKIS A., LIN M., MANOCHA D.: *Fast 3D Geometric Proximity Queries between Rigid and Deformable Models Using Graphics Hardware Acceleration*. Tech. Rep. TR-02-004, University of North Carolina at Chapel Hill, 2002. 2

[KJ01] KOLB A., JOHN L.: Volumetric model repair for virtual reality applications. In *EUROGRAPHICS Short Presentation* (2001), University of Manchester, pp. 249–256. 2, 5

[KOLM02] KIM Y., OTADUY M., LIN M., MANOCHA D.: Fast penetration depth computation using rasterization hardware and hierarchical refinement. In *Proceedings ACM SIGGRAPH/Eurographics symposium on Computer animation* (2002), ACM Press, pp. 23–31. 2

[LG98] LIN M. C., GOTTSCHALK S.: Collision detection between geometric models: a survey. In *Proceedings of IMA Conference on Mathematics of Surfaces* (1998), pp. 37–56. 2

[McA00] MCALLISTER D.: *The Design of an API for Particle Systems*. Tech. rep., Dep. of Computer Science, University of North Carolina at Chapel Hill, 2000. 1, 4

[NT99] NOORUDDIN F., TURK G.: *Simplification and repair of polygonal models using volumetric techniques*. Tech. Rep. GITGVU -99-37, Georgia Institute of Technology, Atlanta, 1999. 2

[NT03] NOORUDDIN F., TURK G.: Simplification and repair of polygonal models using volumetric techniques. *IEEE Trans. on Visualization and Computer Graphics9*, 2 (2003), 191–205. 2

[NVI03] NVIDIA CORPORATION: NVIDIA SDK. http://developer.nvidia.com, 2003. 1, 2

[NVI04] NVIDIA CORPORATION: OpenGL extension EXT_pixel_buffer_object. http://oss.sgi.com/projects/ogl-sample/registry/EXT/pixel_buffer_object.txt, 2004. 5

[PBMH02] PURCELL T., BUCK I., MARK W. R., HANRAHAN P.: Ray tracing on programmable graphics hardware. In *ACM Proceedings SIGGRAPH* (2002), vol. 21, ACM Press, pp. 703–712. 2

[PDC*03] PURCELL T., DONNER C., CAMMARANO M., JENSEN H., HANRAHAN P.: Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (2003), Eurographics Association, pp. 41–50. 3

[Per03] PERCY J.: OpenGL Extensions. www.ati.com/ developer/techpapers.html, 2003. 5

[Ree83] REEVES W.: Particle systems - technique for modeling a class of fuzzy objects. In *ACM Proceedings SIGGRAPH* (1983), vol. 2, pp. 91–108. 1

[Sim90] SIMS K.: Particle animation and rendering using data parallel computation. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques* (1990), ACM Press, pp. 405–413. 1, 4

[Ver67] VERLET L.: Computer experiments on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Physical Review 159* (1967). 4

[VSC01] VASSILEV T., SPANLANG B., CHRYSANTHOU Y.: Fast cloth animation on walking avatars. In *Proc. EUROGRAPHICS* (2001), vol. 20, Eurographics Association, pp. 260–267. 1