Hardware-based Simulation and Collision Detection for Large Particle Systems

Andreas Kolb*  Lutz Latta†  Christof Rezk-Salama*

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

Graphics Hardware, Grenoble, France, August 30th 2004
Overview

- Motivation
- Stateless PS on the GPU
- State-preserving PS on the GPU
- Collision detection
- Results
- Conclusion & future work
Motivation - History & Application

Video games:
- Spacewar (1962): Second video game ever!
- Star Trek II (1983): Planetary fire wall
Motivation - History & Application

Video games:
- Spacewar (1962): Second video game ever!
- Star Trek II (1983): Planetary fire wall

Scientific sample applications:
- Surface Modeling (Szeliski & Tonnesen ’91)
- Collision Detection (Senin etal. ’03)
Prior Work - Stateless PS on the GPU

Stateless simulation: Compute particle data by closed form functions

⇒ no reaction on dynamically changing environment
Prior Work - Stateless PS on the GPU

**Stateless simulation:** Compute particle data by closed form functions

⇒ no reaction on dynamically changing environment

**No storage** of varying data
Prior Work - Stateless PS on the GPU

**Stateless simulation:** Compute particle data by closed form functions

⇒ no reaction on dynamically changing environment

**No storage** of varying data ⇒ simulate in vertex program
Prior Work - Stateless PS on the GPU

**Stateless simulation:** Compute particle data by closed form functions

⇒ no reaction on dynamically changing environment

**No storage** of varying data ⇒ simulate in vertex program

At particle birth

Upload time of birth & initial values to dynamic vertex buffer
Prior Work - Stateless PS on the GPU

**Stateless simulation:** Compute particle data by closed form functions

⇒ no reaction on dynamically changing environment

**No storage** of varying data ⇒ simulate in vertex program

- At particle birth
  - Upload time of birth & initial values to dynamic vertex buffer

- At render time
  - Set global function parameter as vertex program constants
Prior Work - Stateless PS on the GPU

**Stateless simulation:** Compute particle data by closed form functions

⇒ no reaction on dynamically changing environment

**No storage** of varying data ⇒ simulate in vertex program

At particle birth

Upload time of birth & initial values to dynamic vertex buffer

At render time

Set global function parameter as vertex program constants

Render point sprites/triangles/quads with particle system vertex program
State-preserving PS on the GPU

Iterative, time-discrete simulation in fragment program
- Explicit storage of particle data (position, velocity, etc.)
- Reaction on dynamically changing environment
State-preserving PS on the GPU

**Iterative, time-discrete simulation** in fragment program
- Explicit storage of particle data (position, velocity, etc.)
- Reaction on dynamically changing environment

**Stream processing** for dynamic data (position, velocity)
- One or several textures as input stream (read-only)
- One texture as output stream/render target (write-only)

<table>
<thead>
<tr>
<th>Position texture</th>
<th>Velocity texture</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x/y/z)</td>
<td>(x/y/z)</td>
</tr>
<tr>
<td>double buffer</td>
<td>double buffer</td>
</tr>
</tbody>
</table>

Static per particle data, e.g. time of birth (tob), particle type (pt), ...
Algorithm for one time step

1. Process birth and death
2. Velocity operations (forces, particle-object collisions)
3. Position operations
4. Sorting for alpha blending (optional)
5. Transfer position texture to vertex data
6. Rendering
Algorithm for one time step

1. Process birth and death
2. Velocity operations (forces, particle-object collisions)
3. Position operations
4. Sorting for alpha blending (optional)
5. Transfer position texture to vertex data
6. Rendering
Process birth and death

Allocation is a serial problem. Use CPU Heap data structure to get compact index, i.e. tex. coord. Allocation determines initial particle values.Deallocation independently on CPU and GPU. CPU: Re-add freed particle index to allocator. GPU: Move particle out of view volume. In practice, particles fade out or "fall out of view". Clean-up rarely needs to be done.
Process birth and death

**Allocation** is a serial problem $\Rightarrow$ use CPU

**Heap data structure** to get compact index, i.e. tex. coord. range

**Allocation** determines initial particle values
Statepres. PS on the GPU - Allocation

Process birth and death

**Allocation** is a serial problem ⇒ use CPU

**Heap data structure** to get compact index, i.e. tex. coord. range

**Allocation** determines initial particle values

**Deallocation** independently on CPU and GPU
- CPU: Re-add freed particle index to allocator
- GPU: Move particle out of view volume

In practice, particles fade out or “fall out of view” ⇒ clean-up rarely needs to be done
Odd-even merge sort for alpha blending

**GPU-based sorting:** Store particle-viewer distance in texture

**Parallel sorting:** Fixed number of comparisons; $O(n \log_2^2(n))$
Odd-even merge sort for alpha blending

**GPU-based sorting:** Store particle-viewer distance in texture

**Parallel sorting:** Fixed number of comparisons; $O(n \log_2^2(n))$

Sorting network for 8 elements
Odd-even merge sort for alpha blending

GPU-based sorting: Store particle-viewer distance in texture

Parallel sorting: Fixed number of comparisons; $O(n \log_2^2(n))$

![Sorting network for 8 elements](image)

- Every step increases or at least keeps sortedness
- + high inter-frame coherence $\Rightarrow$ partial sorting per frame
Odd-even merge sort for alpha blending

GPU-based sorting: Store particle-viewer distance in texture

Parallel sorting: Fixed number of comparisons; $O(n \log_2^2(n))$

Every step increases or at least keeps sortedness

+ high inter-frame coherence $\Rightarrow$ partial sorting per frame

$1024^2$ particles $\Rightarrow$ 210 sorting passes $\Rightarrow$ spread over 50 frames
Transfer position texture to vertex data
Transfer position texture to vertex data

**Point sprites** most efficient, i.e. only one vertex per particle
Transfer position texture to vertex data

Point sprites  most efficient, i.e. only one vertex per particle

Über-buffer:

- Read & write access to buffer in graphics memory
- Available on current hardware (GFFX, R9xxx)
- OpenGL-only, e.g. `EXT_pixel_buffer_object`

Here: Access position texture as vertex buffer
Transfer position texture to vertex data

Point sprites  most efficient, i.e. only one vertex per particle

Über-buffer:
- Read & write access to buffer in graphics memory

Here: Access position texture as vertex buffer

- Available on current hardware (GFFX, R9xxx)
- OpenGL-only, e.g. EXT_pixel_buffer_object

Vertex texture  as alternative approach
- Access textures from vertex shaders
- Vertex shader actively reads particle positions
- Conceptually available in DirectX (VS3.0) and OpenGL (ARB_vertex_shader/GLSL)
Collision Detection (CD) - HW Approaches

Depth buffer & stencil buffer, e.g. Baciu & Wong ’03

Scene Database  Rendering  Buffer
(spacial clustering)  

Collision Detection  readBack

Simulation and Collision Detection for Large Particle Systems
Collision Detection (CD) - HW Approaches

Depth buffer & stencil buffer, e.g. Baciu & Wong ’03

Scene Database (spatial clustering) → Rendering → Buffer

Collision Detection

readBack

Occlusion queries Govindaraju etal. ’03

Scene Database (spatial clustering) → Occlusion Query → Depth-Buffer

Collision Detection

query result
Collision Detection (CD) - HW Approaches

Depth buffer & stencil buffer, e.g. Baciu & Wong ’03

Scene Database -> Rendering -> Buffer
(spatial clustering)

Collision Detection
readBack

Occlusion queries Govindaraju etal. ’03

Scene Database -> Occlusion Query -> Depth-Buffer
(spatial clustering)

Collision Detection
query result

Collision detection on the GPU?
CD - Implicit Model Representation

Basic concept
Basic concept

**Implicit model:** 3D scalar-valued function $f(P)$:
- $f$ specifies distance to object’s outer boundary
- Signed distance: $> 0 \Rightarrow$ point exterior to object
Basic concept

**Implicit model**: 3D scalar-valued function $f(P)$:
- $f$ specifies distance to object’s outer boundary
- Signed distance: $> 0 \Rightarrow$ point exterior to object

**Image based approach** using depth maps (DM)
- Represent object in depth map textures
- Reconstruct object “on the fly” in fragment program
Basic concept

Implicit model: 3D scalar-valued function $f(P)$:
- $f$ specifies distance to object’s outer boundary
- Signed distance: $> 0 \Rightarrow$ point exterior to object

Image based approach using depth maps (DM)
- Represent object in depth map textures
- Reconstruct object “on the fly” in fragment program

Exact distance map (left) and approximation using one orthogr. projection (right)
Depth Maps (DM)

Collider object information from rendering contains

1. \( \text{dist}(x, y) \): distance to object w.r.t. projection direction
2. Normal vector \( \hat{n}(x, y) \) at the relevant object surface point
3. \( T_{OC \rightarrow DC} \) transforms from object- to DM coordinates
4. \( z_{\text{scale}} \) to compensate for possible \( z \)-scaling by \( T_{OC \rightarrow DC} \)
Depth Maps (DM)

Collider object information from rendering contains

1. \( \text{dist}(x, y) \): distance to object w.r.t. projection direction
2. Normal vector \( \hat{n}(x, y) \) at the relevant object surface point
3. \( T_{OC \rightarrow DC} \) transforms from object- to DM coordinates
4. \( z_{scale} \) to compensate for possible \( z \)-scaling by \( T_{OC \rightarrow DC} \)

Distance measuring for point \( P \in \mathbb{R}^3 \) (in case of orthographic projection):

Map to DC: \[ P' = (p'_x, p'_y, p'_z)^T = T_{OC \rightarrow DC}P \]

Distance value: \[ f(P) = z_{scale} \cdot \left( \text{dist}(p'_x, p'_y) - p'_z \right) \]
Several DMs better approximate the object boundary
Several DMs better approximate the object boundary

Resulting distance value from values $f_1(P), f_2(P), \ldots$

$$f(P) = \begin{cases} 
\max\{f_i(P)\} & \text{if } f_i(P) < 0 \ \forall i \\
\min\{f_i(P) : f_i(P) > 0\} & \text{else}
\end{cases}$$
Several DMs better approximate the object boundary

Resulting distance value from values $f_1(P), f_2(P), \ldots$

$$f(P) = \begin{cases} 
\max \{f_i(P)\} & \text{if } f_i(P) < 0 \ \forall i \\
\min \{f_i(P) : f_i(P) > 0\} & \text{else}
\end{cases}$$
Several DMs better approximate the object boundary.

Resulting distance value from values $f_1(P), f_2(P), \ldots$

$$f(P) = \begin{cases} \max\{f_i(P)\} & \text{if } f_i(P) < 0 \ \forall i \\ \min\{f_i(P) : f_i(P) > 0\} & \text{else} \end{cases}$$
Several DMs better approximate the object boundary

Resulting distance value from values $f_1(P), f_2(P), \ldots$

$$f(P) = \begin{cases} \max\{f_i(P)\} & \text{if } f_i(P) < 0 \ \forall i \\ \min\{f_i(P) : f_i(P) > 0\} & \text{else} \end{cases}$$
Several DMs better approximate the object boundary

Resulting distance value from values $f_1(P), f_2(P), \ldots$

$$f(P) = \begin{cases} \max\{f_i(P)\} & \text{if } f_i(P) < 0 \forall i \\ \min\{f_i(P) : f_i(P) > 0\} & \text{else} \end{cases}$$
Several DMs better approximate the object boundary

Resulting distance value from values $f_1(P), f_2(P), \ldots$

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

Update rule: $$\left\{ \begin{align*} f(P) < 0 & \wedge f_i(P) > f(P) \\ f_i(P) > 0 & \wedge f_i(P) < f(P) \end{align*} \right\} \Rightarrow (f(P) \leftarrow f_i(P))$$
CD - Normal Vector Representation

Desired properties:
CD - Normal Vector Representation

Desired properties:

- space efficient storage (only unit vectors needed!)
  ⇒ use indexing technique into normal-texture
- utilize complete normal-texture
- regular sampling of normal directions
CD - Normal Vector Representation

Desired properties:
- space efficient storage (only unit vectors needed!)
  \[ \Rightarrow \text{use indexing technique into normal-texture} \]
- utilize complete normal-texture
- regular sampling of normal directions

HW-based approaches
1. Cube maps: 3D-index!
2. Parabolic maps: Hemi-sphere & texture partially used

Environmental Cube map

Environmental dual parabolic map
$L_1$-parameterization:
$L_1$-parameterization:

\[
l_1(s, t) = \begin{cases} 
    \begin{pmatrix} 
        s \\
        t \\
        1 - |s| - |t| \\
        \text{sgn}(s)(1 - |t|) \\
        \text{sgn}(t)(1 - |s|) \\
        1 - |s| - |t| 
    \end{pmatrix} & \text{if } |s| + |t| \leq 1 \\
    & \text{if } |s| + |t| > 1 
\end{cases}
\]

Simulation and Collision Detection for Large Particle Systems

---

---
$L_1$-parameterization:

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

maps $[-1, 1]^2$ to $L_1$-unit sphere (octahedron)

Applying $l_1$ to $[-1, 1]^2$ Sampling of directions in 3D
CD - Depth Map Formats

Floating point DM: RGB; A

8-bit fixed point DM: (R; G) = normal-index, B

16-bit fixed DM (front & back): Similar to 8-bit fixed, contains front- & back facing DM

8-bit fixed depth cube: utilize cube map lookup

16-bit xed DM (front & back):

Similar to 8-bit fixed, contains front- & back facing DM

8-bit xed depth cube:

utilize cube map lookup

omni-directional depth map

perspective projection w.r.t. cube center

\[
T_{OC} \rightarrow \frac{1}{k} P_0
\]

determine distance w.r.t. view volume extends

\[
f(P) = \frac{1}{k} \text{dist}(p_0x; p_0y; p_0z)
\]

\[
T_{OC} \rightarrow \frac{1}{k} P_0 = T_{OC} \rightarrow \frac{1}{k} P_0
\]
Floating point DM:  $RGB, A$ store normal & depth value resp.
CD - Depth Map Formats

Floating point DM: $RGB, A$ store normal & depth value resp.

8-bit fixed point DM: $(R, G) = \text{normal-index}$, $BA = \text{store depth value (16 bit)}$
CD - Depth Map Formats

Floating point DM: $RGB, A$ store normal & depth value resp.

8-bit fixed point DM: $(R, G) = \text{normal-index}, \ BA = \text{store depth value (16 bit)}$

16-bit fixed DM (front & back): Similar to 8-bit fixed, contains front- & back facing DM
Floating point DM: \(RGB, A\) store normal & depth value resp.

8-bit fixed point DM: \((R, G)\) = normal-index, \(BA\) = store depth value (16 bit)

16-bit fixed DM (front & back): Similar to 8-bit fixed, contains front- & back facing DM

8-bit fixed depth cube:
- utilize cube map lookup \(\Rightarrow\) omni-directional depth map
- perspective projection w.r.t. cube center
- \(T_{OC\rightarrow DC}\) maps view volume to unit cube \([-1, 1]^3\)
- determine distance w.r.t. view volume extends \(s_x, s_y, s_z\):

\[
f(P) = \left(1 - \frac{dist(p'_x, p'_y, p'_z)}{\|P'\|}\right) \left\|\begin{pmatrix} s_x \cdot p'_x \\ s_y \cdot p'_y \\ s_z \cdot p'_z \end{pmatrix}\right\|, \quad P' = T_{OC\rightarrow DC}P
\]
Results

Performance on NVIDIA Geforce FX 5900 XT

- Only particle simulation: 1024 particles, 10 fps + depth sorting & one depth cube: 512 particles, 15 fps
- Bunny in the snow at 15 fps: 512 particles, depth sorting, one 16-bit fixed front & back DM
- Venus fountain at 10 fps: 512 particles, three 16-bit fixed front & back, one 8-bit fixed DM
Results

Performance on NVIDIA GeForce FX 5900 XT
- Only particle simulation: $1024^2$ particles, 10 fps
- depth sorting & one depth cube: $512^2$ particles, 15 fps
Results

Performance on NVIDIA GeForce FX 5900 XT
- Only particle simulation: $1024^2$ particles, 10 fps
- + depth sorting & one depth cube: $512^2$ particles, 15 fps

Bunny in the snow at 15 fps: $512^2$ particles, depth sorting, one depth cube, one 16-bit fixed front & back DM

Venus fountain at 10 fps: $512^2$ particles, three 16-bit fixed front & back, one 8-bit fixed DM
Hardware Aspects

Normal Representation.

- Normal index texture with resolution $256^2$
- Application specific resolutions require $n$ bit integers
Hardware Aspects

Normal Representation.

- Normal index texture with resolution $256^2$
- Application specific resolutions require $n$ bit integers

**Packing/Unpacking** using NV’s pack/unpack functionality, e.g. packing 8-bits ints in 16-bit int:

```
unpack_2half(pack_4ubyte(...))
```
Normal Representation.
- Normal index texture with resolution $256^2$
- Application specific resolutions require $n$ bit integers

**Packing/Unpacking** using NV’s pack/unpack functionality, e.g. packing 8-bits ints in 16-bit int:
```
unpack_2half(pack_4ubyte(…))
```
More packing functionality would be helpful!
Normal Representation.
- Normal index texture with resolution $256^2$
- Application specific resolutions require $n$ bit integers

Packing/Unpacking using NV’s pack/unpack functionality, e.g. packing 8-bits ints in 16-bit int:

```c
unpack_2half(pack_4ubyte(...))
```

More packing functionality would be helpful!

Other functionality like
- improved integer arithmetic
- improved modulo operators
would help, e.g. for parallel sorting
Conclusion & Future Work

Conclusion
- GPU based approach for large particle systems (PS)
- "stream processing" paradigm for state-preserving PS simulation and collision reaction
- parallel sorting for non-commutative blending
- collision detection based on implicit models
- DM with orthographic & perspective projection
- various formats for efficient DM storage
- $L_1$ parameterization to represent normals

Future Work
- applying $L_1$-parameterization, e.g. as reflection map
- handling linked particles
- GPU based collision detection between (complex) objects
Conclusion

- GPU based approach for large particle systems (PS)
- “stream processing” paradigm for state-preserving PS simulation and collision reaction
- parallel sorting for non-commutative blending
- collision detection based on implicit models
  - DM with orthographic & perspective projection
  - various formats for efficient DM storage
  - $L_1$ parameterization to represent normals

Future Work

- applying $L_1$-parameterization, e.g. as reflection map
- handling linked particles
- GPU based collision detection between (complex) objects
Conclusion & Future Work

Conclusion
- GPU based approach for large particle systems (PS)
- “stream processing” paradigm for state-preserving PS simulation and collision reaction
- parallel sorting for non-commutative blending
- collision detection based on implicit models
- DM with orthographic & perspective projection
- various formats for efficient DM storage
- $L_1$ parameterization to represent normals

Future Work
- applying $L_1$-parameterization, e.g. as reflection map
- handling linked particles
- GPU based collision detection between (complex) objects