Optimized Spatial Hashing for Collision Detection of Deformable Objects

Vision Modeling and Visualization 2003
Matthias Teschner et al.

2 March 2010

Korea University
Computer Graphics Lab.

Analyzed by Po-Ram Kim
Abstract

• We propose a new approach to collision and self–collision detection of dynamically deforming objects that consist of tetrahedrons
• The presented algorithm is integrated in a physically–based environment
  ▪ be used in game engines and surgical simulators
• Using hash function
  ▪ Not always provide a unique mapping of grid cells
  ▪ Optimize the parameter
• The algorithm can detect collisions and self–collisions in environments of up to 20k tetrahedrons in real–time
Introduction

• The detection of collisions and self–collisions of deformable objects based on spatial hashing-1
  ▪ Algorithm classifies all object primitives
    • Object primitives: vertices and tetrahedrons
    • Tetrahedrons → AABB

▪ Using hash function
  • 3D boxes (cells) → 1D hash value
  • Each hash value contains a number of object primitives
  • Self-collision can be detected well
Introduction

• The detection of collisions and self–collisions of deformable objects based on spatial hashing-2
  ▪ Using barycentric coordinates of a vertex with respect to a penetrated tetrahedron
    • To estimate the penetration depth for a pair of colliding tetrahedrons
    • Can be used for collision response
Introduction

- Using a hash function is very efficient
  - Do not need to Spatial Hashing
    - Pre-processing
      - To estimation that the global bounding box and the cell size

- The hash mechanism does not always provide a unique mapping of grid cells to hash table entries
  - the performance decreases
  - To reduce the number of index collisions
    - Optimized the parameters
      - Characteristics of the hash function, hash table size, and the cell size
Introduction

- The paper presents experimental results
  - using physically-based environments for deformable objects with varying geometrical complexity
- 20000 tetrahedrons can be tested for collisions and self-collisions in real-time on a PC
Related Work

• 1. Bounding Box

• 2. Collision Detection using BSP Trees
Related Work

• 3. Collision detection for Bounding Box
• 4. If collision detection is detected for bounding boxes
  ▪ → collision detection for primitives
• Many types of BVs have been investigated
Related Work

• Sphere BV

• AABB BV
  - Efficient collision detection of complex deformable models
    • Journal
    • G. var

between Non-
Related Work

- Physically–based simulation → computational surgery
  - Dynamic Real-Time Deformations using Space & Time Adaptive Sampling
    - SIGGRAPH 2001
    - Gilles Debunne et al.
Related Work

• Cloth modeling
  ▪ Robust treatment of collisions, contact and friction for cloth animation
    • SIGGRAPH ’02
    • R. Bridson et al.
Collision Detection Algorithm

- In a first pass
  - All vertices of all objects are classified with respect to these small 3D cells

- In a second
  - All tetrahedrons are classified with respect to the same 3D cells

- Intersection test
  - Using barycentric coordinates
Collision Detection Algorithm

- Collisions and self–collisions
  - Collisions
    - If
      - A vertex penetrates a tetrahedron
    - Then
      - Collision is detected
  - Self-collisions
    - If
      - A vertex penetrates a tetrahedron
      - The vertex and the tetrahedron belong to the same object
    - Then
      - Self-collisions is detected
Collision Detection Algorithm

• Spatial Hashing of Vertices-1
  ▪ position \((x, y, z)\)
    \(\rightarrow\) integer \((i, j, k)\):
    \[i = \lfloor x/l \rfloor, j = \lfloor y/l \rfloor, k = \lfloor z/l \rfloor\]
  ▪ Example
    • \(P(0.28, 0.72) \rightarrow I(1, 3)\)
    • \(i : 0.28/0.2 = 1.4 \rightarrow 1\)
    • \(j : 0.72/0.2 = 3.6 \rightarrow 3\)
Collision Detection Algorithm

• Spatial Hashing of Vertices-1
  ▪ The hash function
    • Mapping the discretized 3D position \((i, j, k)\) to a 1D index \(h\)
  ▪ The vertex and object information is stored
    • In a hash table at this index \(h\): \(h = \text{hash}(i, j, k)\)

• In a first pass
  ▪ Spatial Hashing of Vertices
  ▪ Compute the AABBs of all tetrahedrons
Collision Detection Algorithm

- Spatial Hashing of Tetrahedrons-2
  - First,
    - The minimum and maximum values describing the AABB of a tetrahedron, are discretized
    - These values are divided by the user-defined cell size and rounded down to the next integer
  - Second,
    - Hash values are computed for all cells affected by the AABB of a tetrahedron
Collision Detection Algorithm

- Spatial Hashing of Tetrahedrons-2
  - All cells are traversed from the discretized minimum to the discretized maximum of the AABB
  - All vertices found at the according hash table index are tested for intersection
Collision Detection Algorithm

• Intersection Test-1
  ▪ If
    • \( p \) and \( t \) are mapped to the same hash index
    • \( p \) is not part of \( t \)
      \[ p : \text{vertex}, t : \text{tetrahedron} \]
  ▪ Then
    • a penetration test has to be performed
Collision Detection Algorithm

• Intersection Test-2(The actual intersection test)
  ▪ First,
    • $p$ is checked against the AABB of $t$
  ▪ second
    • Whether $p$ is inside $t$
      = This test computes barycentric coordinates of $p$ with respect to a vertex of $t$
Parameters

• Optimize all these aspects of the algorithm
  ▪ The characteristics of the hash function
  ▪ The size of the hash table
  ▪ The size of a 3D cell for spatial subdivision
  ▪ The actual intersection test influence the performance of the algorithm
Parameters

Hash Function

- The hash function has to work
  - Vertices of the same object, that are close to each other
  - Vertices of different objects, that are farther away
- Hash function

\[
\text{hash}(x,y,z) = ( x \, p1 \, \text{xor} \, y \, p2 \, \text{xor} \, z \, p3 ) \mod n
\]

where \( p1, p2, p3 \) are large prime numbers in our case 73856093, 19349663, 83492791

- The value \( n \) is the hash table size
Parameters

Hash Table Size

- Larger hash tables
  - reduce the risk of mapping different 3D positions to the same hash index
  - The algorithm generally works faster
  - The performance slightly decreases
    - due to memory management

- If (the hash table size > the number of object primitives)
  - the risk of hash collisions is minimal
Parameters

Hash Table Size

- Performance of the collision detection algorithm for two deformable vessels
- An overall number of 5898 vertices and 20514 tetrahedrons

Figure 3
Parameters

Hash Table Size

- Performance of the collision detection algorithm for 100 deformable objects
- An overall number of 1200 vertices and 1000 tetrahedrons

![Diagram showing collision detection performance](image)

**Figure 4**

Collision detection [ms]

Hash table size
Parameters
Hash Table Size

• **NO** re-initialization of hash table in each simulation step
  ▪ These would reduce the efficiency
  ▪ To avoid this problem
    • each simulation step is labeled with a unique time stamp
  ▪ be performed during the simulation
    • would be comparatively costly for larger hash tables
Parameters

Grid Cell Size

- The grid cell size: used for spatial hashing
  - Influences the number of object primitives
    - Mapping to the same hash index

- In case of larger cells,
  → (cell width size << tetrahedron’s edge length)
    - The number of primitives per hash index increases
      - The intersection test slows down
Parameters

Grid Cell Size

- If (cell size << tetrahedron size)
  → (cell width size << tetrahedron’s edge length)
  - The tetrahedron
    - Covers a larger number of cells
    - has to be checked against vertices in a larger number of hash entries
the grid cell size has a more impact on the performance than hash table size or hash function
Parameters

Intersection Test

• Compare two tests for detecting whether a vertex \( p \) penetrates a tetrahedron \( t \)
  - Barycentric coordinates test
  - Half–space test
    • Checks whether a vertex is in the positive or negative half–space of oriented faces of a tetrahedron
  - Barycentric–coordinate test is faster than the half–space test
    • Using Barycentric coordinates test

\[ \text{\( P \) is a vertex of another tetrahedron.} \]
Parameters

Intersection Test

- Barycentric coordinates test
  - Barycentric coordinates with respect to \( x_0 \)

\[
\beta = (\beta_1, \beta_2, \beta_3)^T \\
p = x_0 + A\beta \\
A = [x_1 - x_0, x_2 - x_0, x_3 - x_0] \\
P = X0 + \beta_1 \cdot V_1 + \beta_2 \cdot V_2 + \beta_3 \cdot V_3 \\
\beta = A^{-1}(p - x_0)
\]
Parameters

Intersection Test

- Barycentric coordinates → Triangle
  \[ \beta = (\beta_1, \beta_2, \beta_3)^T \]
- Barycentric coordinates → Tetrahedron
  - if \[ \beta_1 + \beta_2 + \beta_3 \leq 1 \]
    \[ \beta_1 \geq 0, \beta_2 \geq 0, \beta_3 \geq 0 \]
  - then
    - The vertex is inside the tetrahedron
Time Complexity

• Let $n$ be the number of primitives
  ▪ Primitives: vertices and tetrahedrons

• Time complexity: $O(n^2)$

• The goal of our approach: $O(n)$

• During the first pass takes $O(n)$ time
  ▪ All vertices are inserted into the hash table
**Time Complexity**

• In **the second pass** takes: $O(n \cdot p \cdot q)$
  - $p$ is the average number of cells intersected by a tetrahedron
  - $q$ is the average number of vertices per cell

  - If the cell size is chosen to be proportional to the average tetrahedron size, $p$ is a constant
  - If there are no hash collisions, $q$ is a constant
    • hash collisions: different primitives mapping same hash index

• **Therefore**
  - The time complexity of the algorithm turns out to be linearly dependent on the number of primitives
Results

- The performance is independent from the number of objects
  - It only depends on the number of object primitives

<table>
<thead>
<tr>
<th>setup</th>
<th>objects</th>
<th>tetras</th>
<th>vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
<td>1000</td>
<td>1200</td>
</tr>
<tr>
<td>B</td>
<td>8</td>
<td>4000</td>
<td>1936</td>
</tr>
<tr>
<td>C</td>
<td>20</td>
<td>10000</td>
<td>4840</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td>20514</td>
<td>5898</td>
</tr>
<tr>
<td>E</td>
<td>100</td>
<td>50000</td>
<td>24200</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>setup</th>
<th>ave [ms]</th>
<th>min [ms]</th>
<th>max [ms]</th>
<th>dev [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4.3</td>
<td>4.1</td>
<td>6.5</td>
<td>0.24</td>
</tr>
<tr>
<td>B</td>
<td>12.6</td>
<td>11.3</td>
<td>15.0</td>
<td>0.59</td>
</tr>
<tr>
<td>C</td>
<td>30.4</td>
<td>28.9</td>
<td>34.4</td>
<td>1.25</td>
</tr>
<tr>
<td>D</td>
<td>70.0</td>
<td>68.5</td>
<td>72.1</td>
<td>0.86</td>
</tr>
<tr>
<td>E</td>
<td>172.5</td>
<td>170.5</td>
<td>174.6</td>
<td>1.08</td>
</tr>
</tbody>
</table>
Discussion

• The proposed algorithm
  ▪ Detects whether a vertex penetrates a tetrahedron

• Does NOT detect whether an edge intersects with a tetrahedron
  • The performance of the algorithm would decrease significantly
    ▪ The relevance of an edge test is unclear in case of densely sampled objects
  • It is hard to do collision response in case of penetrating edges
Discussion

• Tetrahedrons are usually mapped to several hash indices
  - Leads to a larger number of elements in the hash table
    - decreasing the performance of the algorithm

• The comparison of the performance with other CD
  - It is difficult
    - RAPID [9], PQP [18], and SWIFT [7]
      - These are NOT optimized for deformable objects
      - They work with data structures
        - That can be pre-computed for rigid bodies
      - But they have to be updated in case of deformable objects
Ongoing Work

- Correct collision response based on our algorithm
  - Our algorithm provides the exact position of a vertex inside a penetrated tetrahedron
  - we can easily derive the penetration depth

- For real-time simulation of deformable objects
  → can be used in game engines or surgical simulators
  - Completed with the collision response (above mentioned)
    - the framework will handle interacting deformable models of up to several thousand tetrahedrons in real-time
Conclusion

• We have introduced
  ▪ Detecting collisions and self–collisions of dynamically deforming objects
  ▪ Origin: computing the global bounding box of all objects and explicitly performing a spatial subdivision
  ▪ Ours: using a hash function that maps 3D cells to a hash table
  ▪ Actual vertex-in-tetrahedron test
    • Using barycentric coordinates
      = Using this information
        : can be used for physically-based collision response
  ▪ optimized the parameters
  ▪ 20k tetrahedrons can be processed in real–time