Simulation in Computer Graphics
Space Subdivision

Matthias Teschner
Outline

– Introduction
– Uniform grid
– Octree and k-d tree
– BSP tree
Model Partitioning

(1) Bounding volumes

(2) Bounding volume tree

(3) Collision detection test
Model vs. Space Partitioning

Model partitioning

Space partitioning
Motivation

- Restrict pairwise object tests to objects that are located in the same region of space
- Only objects or object primitives in the same region of space can overlap
- Efficient broad-phase approach for larger numbers of objects
**Spatial Data Structures**

- Space is subdivided into cells
- Cells maintain references to primitives intersecting the cell
- Data structures have different degrees-of-freedom
- Actual space subdivision is adapted to the scene

![Uniform grid](image1)
![Quadtree/Octree](image2)
![k-d tree](image3)
![BSP tree](image4)
Outline

– Introduction
– Uniform grid
– Octree and k-d tree
– BSP tree
Basic Idea

- Space is divided into cells
- Object primitives are placed into cells
- Object primitives in the same cell are checked for collision
- Pairs of primitives that do not share the same cell are not tested (trivial reject)
Context

- Collision detection for deformable objects
- Tetrahedral meshes
- Uniform 3D grid
- Non-uniform distribution of object primitives and unbounded simulation domain (Hashing)
- Detection of collisions and self-collisions
- Discussion of parameters
Setup

Infinite uniform grid

Hash function:
\[ H(\text{cell}) \rightarrow \text{hash table index} \]

Hash table

Spatial data structure

Representation / implementation
Stage 1

– All vertices are hashed according to their cell
Stage 2

- All tetrahedrons are hashed according to the cells touched by their bounding box
Stage 3

- Vertices and tetrahedrons in the same hash table entry are tested for intersection

A) \rightarrow \text{no collision}

B) \rightarrow \text{collision}

C) \rightarrow \text{self-collision}
Vertex-in-Tetrahedron Test

- Barycentric coordinates more efficient
- They also provide useful collision information
Implementation

– Store all vertices in the hash table
– Compute hash table indices for the bounding boxes of the tetrahedrons
– Do not store the tetrahedrons in the hash table, but check for intersection with all vertices in the respective entry
– Parameters
  – grid cell size, grid cell shape, hash table size, hash function
Parameters

Infinite uniform grid

Hash function: $H(\text{cell}) \rightarrow \text{hash table index}$

Hash table

cell shape  cell size  hash table size
Grid Cell Size

– Cell size should be equal to the size of the bounding box of an object primitive [Bentley 1977]

[Teschner, Heidelberger et al. 2003]
**Hash Table Size**

- Hash collisions reduce the performance
- Larger hash table can reduce hash collisions

---

[Teschner, Heidelberger et al. 2003]

---

*Math diagram* 

**test scenario**
Hash Function

– Should avoid hash collisions
– Should be efficient (has to be computed for all primitives)
  \[ H(x, y, z) = (p_1 \cdot x \ xor p_2 \cdot y \ xor p_3 \cdot z) \mod n \]
– Cell coordinates: \( x, y, z \)
– Large primes: \( p_1, p_2, p_3 \)
– Hash table size: \( n \)
**Performance**

- Linear in the number of primitives
- Independent of the number of objects

<table>
<thead>
<tr>
<th>Objects</th>
<th>Tetras</th>
<th>Vertices</th>
<th>Max time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1000</td>
<td>1200</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>4000</td>
<td>1936</td>
<td>15</td>
</tr>
<tr>
<td>20</td>
<td>10000</td>
<td>4840</td>
<td>34</td>
</tr>
<tr>
<td>2</td>
<td>20514</td>
<td>5898</td>
<td>72</td>
</tr>
<tr>
<td>100</td>
<td>50000</td>
<td>24200</td>
<td>174</td>
</tr>
</tbody>
</table>

Test scenarios
Pentium 4, 1.8GHz
Summary – Uniform Grid

- Space uniformly partitioned into axis-aligned space cells
- Primitives (or their AABBs) are scan-converted to identify intersected space cells
- Hashed storage of cells for non-uniform distribution
- Simple and efficient
Summary – Uniform Grid

– Particularly interesting for deformable objects, $n$-body environments and self-collision
– Parameters significantly influence the performance
– Performance dependent on the number of primitives
– Performance independent of the number of objects
– Technique works with various types of primitives
... Some History

- [Levinthal 1966]
  - 3D grid ("cubing")
  - Analysis of molecular structures
  - Neighborhood search
to compute atom interaction
- [Rabin 1976]
  - 3D grid + hashing, finding closest pairs
- [Turk 1989, 1990]
  - 3D grid + hashing, rigid collision detection
Outline

– Introduction
– Uniform grid
– Octree and k-d tree
– BSP tree
Octree

- Hierarchical structure
- Space partitioning into rectangular, axis-aligned cells
- Octree root node corresponds to AABB of an object
- Internal nodes represent subdivisions of the AABB
- Leaves represent cells which maintain primitive lists
Octree

- Uniform or non-uniform subdivision
- Adaptive to local distribution of primitives
  - Large cells in case of low density of primitives
  - Small cells in case of high density
- Dynamic update
  - Cells with many primitives can be subdivided
  - Cells with less primitives can be merged
k-d Tree – 2-d Example

Diagram showing a 2-dimensional k-d tree with data points and decision nodes. The tree is structured with two dimensions, x and y, and the points are split based on x1 and y1, y2 values.
Collision Query (Range Query)

- Traverse all nodes affected by the intervals of an AABB
- Check all primitives in the leaves for intersection
Outline

- Introduction
- Uniform grid
- Octree and k-d tree
- BSP tree
Binary Space Partitioning Tree BSP

- Hierarchical structure
- Space is subdivided by means of arbitrarily oriented planes
- Generalized k-d tree
- Space partitioning into convex cells
- Discrete-orientation BSP trees DOBSP (finite set of plane orientations)
- Proposed by [Henry Fuchs et al. 1980] to solve the visible surface problem
Collision Detection Example

- BSP trees can be used for the inside/outside classification of closed polygons
Collision Query

– Query point is inside

– Query point is outside
Rendering Example

Scene

Scene partitioning

BSP tree

University of Freiburg – Computer Science Department – 33
BSP Tree for Rendering

- For a given viewpoint
  - Render far branch
  - Render root (node) polygon
  - Render near branch
- Recursively applied to sub-trees
- Back to front rendering
- Example: viewpoint is in 1-
  - Rendering of 1+, 1, 1-
  - Rule recursively applied to 1+ and 1-
  - Viewpoint is in 3+ → rendering of 3, 2b
  - Viewpoint is in 4- → rendering of 2a, 4
Construction

- Keep the number of nodes small
- Keep the number of levels small
- Introduce arbitrary support planes (especially in case of convex objects, where all polygon faces are in the same half-space with respect to a given face)
Summary

– Uniform grid
– Octree and k-d tree
– BSP tree
References