Advanced Computer Graphics

Intersection Acceleration

Matthias Teschner

Computer Science Department
University of Freiburg
Outline

- introduction
- bounding volume hierarchies
- uniform grids
- kd-trees
- octrees
- implementation
Motivation

- a large number of rays has to be checked for intersection with a large number of objects / shapes / primitives
- in 1968, Appel's approach spent up to 95% of the computation time for intersection computations
- use spatial data structures to accelerate the intersection computation
- minimize the number of potential intersection candidates by efficiently rejecting large parts of the scene geometry
Spatial Data Structures

- bounding volumes
  - spheres
  - axis-aligned bounding boxes
  - object-oriented bounding boxes
  - k-dops
- bounding volume hierarchies
- space subdivision
  - uniform grids
  - octrees
  - k-d trees
  - BSP trees
Spatial Data Structures - Examples

- space subdivision with a uniform grid (space oriented)
- bounding volume hierarchy with spheres (object oriented)

[Allen Chang]
Spatial Data Structures - Efficiency

- determined by
  - generation
  - query
  - update
- generation is usually a pre-processing step
- query implements a traversal of the data structure to compute the first intersection of the scene with a ray
- update is only relevant for dynamic scenes
- static scenes: efficiency dominated by the query
- dynamic scenes: efficiency is determined by query and update
Spatial Data Structures - Applications

- e.g., ray tracing, collision detection in animation, neighborhood search in particle-based fluid animation
- generally, objects or primitives are represented in a spatial data structure to accelerate a certain query
- the generation of a particular spatial data structure is quite independent from the actual application
- the query of the data structure differs
  - first ray-primitive intersection in a raytracer
  - all primitive-primitive intersections for collision detection
  - all particles within a certain radius of a particle for fluid animation
Spatial Data Structures - Implementation

- depending on the query, different implementations can be appropriate
- e.g. for uniform grids
  - explicit representation
  - spatial hashing
  - compact hashing
  - index sort
  - z-index sort
- e.g. for bounding volume hierarchies
  - priority queues
  - skip lists
Outline

- introduction
- bounding volume hierarchies
- uniform grids
- kd-trees
- octrees
- implementation
Bounding Volumes

- simple geometric shapes that enclose complex geometry
- if the simple geometry is not hit by the ray, then the complex geometry is also not hit by the ray
- for efficiency
  - intersection test should be fast
  - bounding volume should be tight-fitting
- a) \implies d) improved fitting
- d) \implies a) improved computation time

a) sphere
b) axis-aligned box
c) object-oriented box
d) slabs, k-DOP (discrete orientation polytope)
Bounding Volumes

- intersection
  - if a ray hits all bounding volumes, the enclosed geometry is tested

- union
  - if a ray hits one of the bounding volumes, the geometry is tested

[Allen Chang]
Bounding Volume Hierarchies

- tree with a root node
- each node represents a bounding volume
- the union of all bounding volumes in a tree level encloses the entire geometry
- parts of the geometry (primitives) are stored in leaf nodes

[Allen Chang]
Bounding Volume Hierarchies - Construction

- degrees of freedom
  - top down / bottom up
  - number of children (branching factor)

- goals
  - balanced tree
  - minimal surface of bounding volumes per level
  - minimal overlap of bounding volumes per level

- strategies
  - group primitives close to each other
    (using sorted primitives)
  - split primitives into groups with similar numbers of primitives
Bounding Volume Hierarchy
Ray Traversal - Priority Queue / Heap

- B - bounding volume of the root node
- heap - Min Heap / Priority List

heap = empty; intersection = inf;
if B.intersection(ray)<intersection then
    heap.add(B);
while heap.notEmpty() and
    heap.min.intersection(ray)<intersection do
    cand = heap.min; heap.remove (heap.min);
    if cand.leaf() then
        intersection = cand.minIntersection(ray);
    else
        foreach cand.child do
            if cand.child.intersection(ray)<intersection then
                heap.add(cand.child);
    return intersection;

depth-first, near-to-far
Bounding Volume Hierarchy - Ray Traversal

- if bounding volumes in one layer overlap, the closest intersection of a ray with a primitive is not necessarily in the bounding volume that is entered first

[Kay, Kajiya]
Bounding Volume Hierarchy
Ray Traversal - Priority Queue / Heap

If an intersection would have been found within bounding volume 2, bounding volume 3 would not be processed, as the bounding volumes in this level of the hierarchy do not overlap.

Updating the heap is in $O(\log n)$. The tree-structure of the heap can be linearly represented with an array.

[Allen Chang]
If an intersection would have been found within bounding volume 2, bounding volume 3 still has to be processed, as the bounding volumes in this level of the hierarchy overlap.
Bounding Volume Hierarchy
Ray Traversal - Skip List

If (1) is hit, proceed with (2), otherwise proceed with (null).
If (2) is hit, proceed with (4), otherwise proceed with (3)

[Allen Chang]
Bounding Volume Hierarchy
Ray Traversal - Skip List

(a) Bounding Volume Hierarchy
(b) Skip List Tree
(c) Skip List Traversal

[Allen Chang]
Outline

- introduction
- bounding volume hierarchies
- uniform grids
- kd-trees
- octrees
- implementation
Uniform Grids - Construction

- subdivide 3D space into cubes / cuboids
- associate references to primitives that intersect a cell with this cell
- if a ray hits a cell, only the few primitives inside this cell are tested for intersection

- parameters
  - the cell size is essential for the performance
  - small cells: primitives are spread over many cells
  - large cells: too many primitives in one cell
  - number of cells often proportional to the number of primitives
  - \( M = \rho N \) with 2 \( \leq \rho \leq 10 \)
Uniform Grids - Variants

Basic concept: References to primitives are stored in cells.

To accelerate the traversal, empty cells can be combined to larger areas ($2^i \times 2^i$).

Several grids with varying resolution can be used in parallel.

Grids are commonly combined with bounding volumes to simplify the primitive-cell mapping.

[Allen Chang]
Uniform Grids - Traversal

- unfortunately not as efficient as Bresenham's line algorithm

[Allen Chang]
Uniform Grids - Traversal

\( \partial x, \partial y, \partial z \)
parametric distance along the ray between two grid planes perpendicular to x, y, z (infinite, if a ray is parallel to a principal axis of the grid)

\( dx, dy, dz \)
parametric value of the ray at the next intersection with a grid plane perpendicular to x, y, z (infinite, if a ray is parallel to a principal axis of the grid)

\( i, j, k \)
indices of the current grid cell

\( p_x, p_y, p_z \in \{-1, 1\} \)
increments of cell indices if a grid plane is intersected

Simplified 2D algorithm

\[
\begin{align*}
px & = +1 \\
py & = -1 \\
\text{initialize } & \partial x, \partial y, dx, dy, i, j \\
\text{repeat} & \\
\text{if } & \partial x \leq \partial y \\
\quad & \text{begin} \\
\quad & i := i + px; \\
\quad & dx := dx + \partial x; \\
\quad & \text{end} \\
\text{if } & \partial x \geq \partial y \\
\quad & \text{begin} \\
\quad & j := j + py; \\
\quad & dy := dy + \partial y; \\
\quad & \text{end} \\
\text{until } & \text{intersection in cell } i, j;
\end{align*}
\]
Uniform Grids
Traversal - Initialization

- $dx, dy, dz$ have to be initialized with the first intersections of a ray with planes perpendicular to x, y, z inside the grid

[Cleary]
Uniform Grids - Mailboxing

- avoids multiple tests of one ray with the same object that is associated with different cells
  - each object stores a reference to the last ray that has been tested
- not necessarily useful for cheap intersection tests

Ray k is tested with C. A reference to k is stored with C.

Ray k is tested with A, B. It is not tested with C. A reference to k is stored with A.

Ray k is not tested with A, B, as A and B store a reference to k.

[Cleary]
Uniform Grids - Robust Traversal

- standard traversal discards intersections outside the currently visited cell
  - the object is partially inside the cell, thus it is associated with the cell, but the actual intersection might be outside the cell
- standard traversal can miss intersections close at the cell border due to numerical issues
- $\varepsilon$-band might be used
- robust grid traversal
  - keep the closest intersection for a cell, even if the intersection is outside this cell
  - terminate ray traversal, if the maximum ray parameter of a cell is larger than the ray parameter of the closest intersection
Outline

- introduction
- bounding volume hierarchies
- uniform grids
- kd-trees
- octrees
- implementation
**kd-Trees - Construction**

- recursively subdivide 3D space in half spaces using planes
- planes are perpendicular to the x-, y-, z-axes
- steps
  - subdivide space with respect to the x-axis
  - subdivide the resulting subspaces with respect to y
  - continue with z, x, y, z, x, y, ... until there is only a small number of primitives in each subspace
- balanced tree $\Rightarrow$ minimized depth

[Allen Chang]
**kd-Trees - Traversal**

- check the global bounding box for intersection
- compute the intersection with the plane represented in the root node
- recursively traverse the two children in front-to-back order
- if a node is a leaf, check the primitives
- stop, if a ray-primitive intersection has been found or no further spaces have to be processed

[Pharr, Humphreys]
**kd-Trees - Traversal**

- given a position on the ray, the kd-tree can be traversed to decide the processing order of the subspaces

- if the intersection with a splitting plane is outside the considered cuboid, the processing of half spaces can be accelerated
Outline

- introduction
- bounding volume hierarchies
- uniform grids
- kd-trees
- octrees
- implementation
Octrees - Construction

- start with a global bounding box
- recursively subdivide the box using three planes into eight sub boxes until each sub box contains less than a certain number of primitives

[Allen Chang]
Octrees - Traversal

- e.g.
  - if the global bounding box is hit, process the octree in a BVH manner
  - as sub boxes in the same layer do not overlap, processing can be stopped if an intersection has been found and if the processing order of sub boxes is determined with a priority list
Outline

- introduction
- bounding volume hierarchies
- uniform grids
- kd-trees
- octrees
- implementation
Uniform Grid
Linked List and Dynamic Array

2D grid and linearized representation

Linked list:
- minimal memory overhead
- bad locality
- frequent memory allocations
- single primitives can be inserted

Dynamic array:
- keeps track of capacity and size
- less memory efficient
- improved data locality
- minimized memory allocations
- single primitives can be inserted

[ Lagae]
Uniform Grid - Compact Grid

2D grid and linearized representation

Compact grid:
- two arrays
- useful for dynamic scenes
- generation rather expensive
- generation + traversal more efficient than linked lists or dynamic arrays
- single primitives cannot be inserted or deleted

[Lagae]
Uniform Grid - Compact Grid - Generation

- generate C
- store the number of primitives in each cell of C
  - loop over all primitives and increment the respective value in C
- accumulate the values in C
- generate L
- associate primitive i with cell j: \( L[|--C[j]|]=i \)
  - stores the primitives in reversed order into L
  - after insertion C contains the correct offsets

[Source: Lagae]
Uniform Grid - Hashed Grid

- motivated by a large number of empty cells
- only stores filled grid cells in a hash table

**generation**
- process all primitives
- compute the cell id
- evaluate the hash function for the cell id
- place the primitive into the hash table
  (use a dynamic array per hash table entry)

**traversal**
- compute the cell id that is intersected by a ray
- evaluate the hash function for the cell id
- look-up all relevant primitives from the hash table
Uniform Grid - Hashed Grid

- To avoid hash collisions, perfect hashing can be used.
- E.g., row displacement compression.
  - Identify occupied grid cells.
  - Compute an offset for each row (O) to store all rows with non-overlapping occupied cells in a hash table (H).

- Index into the hash table is $O[\text{row}] + \text{column}$.

[Lagae]
# Uniform Grid - Performance

<table>
<thead>
<tr>
<th>Scene Statistics</th>
<th>Thai Statue</th>
<th>Lucy</th>
<th>Nature</th>
</tr>
</thead>
<tbody>
<tr>
<td>#tri’s</td>
<td>10.00 M</td>
<td>28.06 M</td>
<td>41.35 M</td>
</tr>
<tr>
<td>memory</td>
<td>343.32 MB</td>
<td>963.22 MB</td>
<td>1.39 GB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Grid Statistics</th>
<th>Thai Statue</th>
<th>Lucy</th>
<th>Nature</th>
</tr>
</thead>
<tbody>
<tr>
<td>grid res</td>
<td>302x508x261</td>
<td>485x278x832</td>
<td>906x202x902</td>
</tr>
<tr>
<td># cells</td>
<td>40.04 M</td>
<td>112.18 M</td>
<td>165.08 M</td>
</tr>
<tr>
<td>% empty cells</td>
<td>98.44 %</td>
<td>99.00 %</td>
<td>92.04 %</td>
</tr>
<tr>
<td>avg # tri’s / n-emp cell</td>
<td>29.25</td>
<td>41.50</td>
<td>28.52</td>
</tr>
<tr>
<td>avg # cells / tri</td>
<td>1.83</td>
<td>1.66</td>
<td>9.06</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Render Statistics</th>
<th>Thai Statue</th>
<th>Lucy</th>
<th>Nature</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg # n-emp cells / isect ray</td>
<td>2.38</td>
<td>2.06</td>
<td>12.12</td>
</tr>
<tr>
<td>avg # isect tests / isect ray</td>
<td>73.91</td>
<td>99.28</td>
<td>302.23</td>
</tr>
</tbody>
</table>
Uniform Grid - Performance

- **1024 x 1024 x 1 rays**
- **diffuse shading**

<table>
<thead>
<tr>
<th></th>
<th>Thai Statue</th>
<th>Lucy</th>
<th>Nature</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>compact grid statistics</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mem cells</td>
<td>152.75 MB</td>
<td>427.93 MB</td>
<td>629.72 MB</td>
</tr>
<tr>
<td>mem tri lists</td>
<td>69.78 MB</td>
<td>178.06 MB</td>
<td>1.40 GB</td>
</tr>
<tr>
<td>build time</td>
<td>1.17 s</td>
<td>3.15 s</td>
<td>9.12 s</td>
</tr>
<tr>
<td>render time</td>
<td>1.55 s</td>
<td>1.90 s</td>
<td>10.75 s</td>
</tr>
<tr>
<td>time to image</td>
<td>2.72 s</td>
<td>5.05 s</td>
<td>19.87 s</td>
</tr>
<tr>
<td>memory</td>
<td>222.53 MB</td>
<td>605.98 MB</td>
<td>2.01 GB</td>
</tr>
</tbody>
</table>

| **hashed grid statistics** |             |          |           |
| data density          | 1.56 %      | 1.00 %   | 7.96 %    |
| hash table size       | 967.47 K    | 1.76 M   | 28.82 M   |
| hash table load factor| 64.64 %     | 63.76 %  | 45.57 %   |
| mem domain bits       | 4.77 MB     | 13.37 MB | 19.68 MB  |
| mem offset table      | 517.92 Kb   | 903.50 Kb| 711.73 Kb |
| mem hash table        | 3.69 MB     | 6.73 MB  | 109.94 MB |
| mem cells             | 8.97 MB     | 20.98 MB | 130.31 MB |
| compression ratio     | 1700 %      | 2040 %   | 483.24 %  |
| mem tri lists         | 69.78 MB    | 178.06 MB| 1.40 GB   |
| build time            | 1.76 s      | 4.77 s   | 21.23 s   |
| render time           | 1.43 s      | 1.53 s   | 10.07 s   |
| time to image         | 3.18 s      | 6.30 s   | 31.30 s   |
| memory                | 78.75 MB    | 199.04 MB| 1.52 GB   |
Summary

- spatial data structures minimize the number of potential intersection candidates by efficiently rejecting large parts of the scene geometry
  - bounding volumes
  - bounding-volume hierarchies
  - uniform grids
  - k-d trees
  - octress
- the actual implementation of the data structure significantly influences the performance