Advanced Computer Graphics
Intersection Acceleration

Computer Graphics
Computer Science Department
University of Freiburg

SS 12
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)
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
Various 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) ⇒ d) improved fitting
- d) ⇒ 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
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
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
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.
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).
Bounding Volume Hierarchy
Ray Traversal - Skip List

(a) Bounding Volume Hierarchy
(b) Skip List
(c) Skip List Structure
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

: grid cells identified by Bresenham's DDA

: additional grid cells pierced by ray
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
Uniform Grids
Traversal - Initialization

- \( dx, dy, dz \) have to initialized with the first intersections of a ray with planes perpendicular to \( x, y, z \) inside the grid
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 A, B, C. A reference to k is stored with A, B and C.

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

Robust Grid 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 or
- 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 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 ⇒ minimized depth
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
**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.

If a point on the ray is below with respect to a splitting plane, then the processing order is below-above, otherwise it's above-below.

If \( t_{\text{split}} \) is outside \([t_{\text{min}}, t_{\text{max}}]\), the processing 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
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
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

L[ C[i] ] - first primitive of cell i
C[i+1]-C[i] - number of primitives in cell i

offset of the corresponding primitive set
concatenation of all primitive sets
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
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
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 $O \text{[row]} + \text{column}$
## Uniform Grid Performance

<table>
<thead>
<tr>
<th></th>
<th>Thai Statue</th>
<th>Lucy</th>
<th>Nature</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>scene statistics</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<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>
<tr>
<td><strong>grid statistics</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<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-empty 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>
<tr>
<td><strong>render statistics</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>avg # n-empty cells / intersect ray</td>
<td>2.38</td>
<td>2.06</td>
<td>12.12</td>
</tr>
<tr>
<td>avg # intersect tests / intersect 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><img src="image1.png" alt="Image" /></td>
<td><img src="image2.png" alt="Image" /></td>
<td><img src="image3.png" alt="Image" /></td>
<td></td>
</tr>
</tbody>
</table>

#### compact grid statistics

<table>
<thead>
<tr>
<th></th>
<th>Thai Statue</th>
<th>Lucy</th>
<th>Nature</th>
</tr>
</thead>
<tbody>
<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

<table>
<thead>
<tr>
<th></th>
<th>Thai Statue</th>
<th>Lucy</th>
<th>Nature</th>
</tr>
</thead>
<tbody>
<tr>
<td>data density</td>
<td>1.56 %</td>
<td>1.00 %</td>
<td>7.96 %</td>
</tr>
<tr>
<td>hash table size</td>
<td>967.47 K</td>
<td>1.76 M</td>
<td>28.82 M</td>
</tr>
<tr>
<td>hash table load factor</td>
<td>64.64 %</td>
<td>63.76 %</td>
<td>45.57 %</td>
</tr>
<tr>
<td>mem domain bits</td>
<td>4.77 MB</td>
<td>13.37 MB</td>
<td>19.68 MB</td>
</tr>
<tr>
<td>mem offset table</td>
<td>517.92 Kb</td>
<td>903.50 Kb</td>
<td>711.73 Kb</td>
</tr>
<tr>
<td>mem hash table</td>
<td>3.69 MB</td>
<td>6.73 MB</td>
<td>109.94 MB</td>
</tr>
<tr>
<td>mem cells</td>
<td>8.97 MB</td>
<td>20.98 MB</td>
<td>130.31 MB</td>
</tr>
<tr>
<td>compression ratio</td>
<td>1700 %</td>
<td>2040 %</td>
<td>483.24 %</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>1.76 s</td>
<td>4.77 s</td>
<td>21.23 s</td>
</tr>
<tr>
<td>render time</td>
<td>1.43 s</td>
<td>1.53 s</td>
<td>10.07 s</td>
</tr>
<tr>
<td>time to image</td>
<td>3.18 s</td>
<td>6.30 s</td>
<td>31.30 s</td>
</tr>
<tr>
<td>memory</td>
<td>78.75 MB</td>
<td>199.04 MB</td>
<td>1.52 GB</td>
</tr>
</tbody>
</table>

[Source: Lagae]
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