Spatial Search Data Structures

Corso di dottorato: Geometric Mesh Processing
Problem statement

- Let $m$ be a mesh:
  - Which is the mesh element closest to a given point $p$?
  - Which are the elements inside a given region?
  - Which elements are intersected by a given ray $r$?

- Let $m'$ be another mesh:
  - Do $m$ and $m'$ intersect? If so, where?

- A spatial search data structure helps to answer efficiently to these questions
Motivations

- Picking on a point
- Selecting a region
Motivations\textsuperscript{cntd}

- Ray tracing: shoot a ray for each pixel, see what it hits, possibly recur, compute pixel color
- Involves plenty of ray-objects intersections
Collision detection: in dynamic scenes, moving objects can collide.

How to find out which triangles intersect?
Motivations

- Without any spatial search data structure the solutions to these problems require $O(n)$ time, where $n$ is the numbers of primitives ( $O(n^2)$ for the collision detection)
- Spatial data structure can make it (average) constant
  - ..or average logarithmic
Uniform Grid (1/4)

- **Description:** the space including the object is partitioned in cubic cells; each cell contains references to “primitives” (i.e. triangles)

- **Construction.**
  Primitives are assigned to:

  - The cell containing their feature point (e.g. barycenter or one of their vertices)
  - The cells spanned by the primitives
Spatial Search Data Structure

Uniform Grid (2/4)

- **Closest element** (to point p):
  - Start from the cell containing p
  - Check for primitives inside growing spheres centered at p
  - At each step the ray increases to the border of visited cells

- **Cost**.
  - Worst: $O(\#\text{cells} + n)$
  - Average; $O(1)$
Uniform Grid (3/4)

- **Intersection with a ray:**
  - Find all the cells intersected by the ray
  - For each intersected cell, test the intersection with the primitives referred in that cell
  - Avoid multiple testing by flagging primitives that have been tested (mailboxing)

- **Cost:**
  - Worst: \( O(\#cells + n) \)
  - Aver: \( O(\sqrt[d]{\#cells} + \sqrt[n]{n}) \)
Uniform Grid (4/4)

- **Memory occupation:** $O(\#cells + n)$

- **Pros:**
  - Easy to implement
  - Fast query

- **Cons:**
  - Memory consuming
  - Performance very sensitive to distribution of the primitives.
Spatial Hashing (1/2)

- The same as uniform grid, except that only non empty cells are allocated

**Uniform grid**

**Spatial hashing**

$\text{HASH}(\text{key}(i,j,k))$

$\text{collisions}$

$\ll \#\text{cells}$
Spatial Hashing (2/2)

- **Cost:** same as UG, except that in worst case the access to a cell is $O(\#\text{cells})$ because of collisions

- **Memory occupation:**
  - Worst. : $O(\#\text{cells})$
  - Aver. : $O\left(\frac{\#\text{cells}}{\text{Vol}}\right)^2 \cdot S)$  
    
    $\text{S: surface, Vol: Volume}$

- **Pros:**
  - Easy to implement
  - Fast query if **good hashing** is done
  - Less memory consuming

- **Cons:**
  - Performance **very** sensitive to distribution of the primitives.
Beyond UG

- Uniform grids are input insensitive
- What’s the best choice for the example below?
Hierarchical Indexing of Space

- **Divide et impera strategies:**
  - The space is partitioned in sub regions
  - ..recursively
Hierarchical Indexing of Space

- Divide et impera strategies:
  - The space is partitioned in sub regions
  - ...recursively
Hierarchical Indexing of Space

- Divide et impera strategies:
  - The space is partitioned in sub regions
  - ..recursively
Hierarchical Indexing of Space

- Divide et impera strategies:
  - The space is partitioned in sub regions
  - ...recursively
Basic Facts

- The queries correspond to a visit of the tree
  - The complexity is sublinear in the number of nodes (logarithmic)
  - The memory occupation is linear

- A hierarchical data structure is characterized by:
  - Number of children per node
  - Spatial region corresponding to a node
Spatial Search Data Structure

Binary Space Partition-Tree (BSP) (1/3)

- **Description:**
  - It’s a binary tree obtained by recursively partitioning the space in **two** by a hyperplane.
  - Therefore a node always corresponds to a **convex** region.
**Binary Space Partition-Tree (BSP) (1/3)**

- **Description:**
  - It’s a binary tree obtained by recursively partitioning the space in two by a hyperplane
  - therefore a node always corresponds to a convex region
Binary Space Partition-Tree (BSP) (1/3)

- **Description:**
  - It’s a binary tree obtained by recursively partitioning the space in two by a hyperplane.
  - Therefore a node always corresponds to a convex region.
Binary Space Partition-Tree (BSP) (1/3)

- **Description:**
  - It’s a binary tree obtained by recursively partitioning the space in two by a hyperplane
  - therefore a node always corresponds to a convex region

Spatial Search Data Structure
**Binary Space Partition-Tree (BSP) (1/3)**

- **Description:**
  - It’s a binary tree obtained by recursively partitioning the space in two by a hyperplane.
  - Therefore a node always corresponds to a convex region.
**Binary Space Partition-Tree (BSP) (1/3)**

- **Description:**
  - It’s a binary tree obtained by recursively partitioning the space in two by a hyperplane.
  - Therefore a node always corresponds to a convex region.

---

Spatial Search Data Structure
Binary Space Partition-Tree (BSP) (1/3)

- **Query**: is the point $p$ inside a primitive?
  - Starting from the root, move to the child associated with the half space containing the point
  - When in a leaf node, check all the primitives

- **Cost**:
  - Worst: $O(n)$
  - Aver: $O(\log n)$

The diagram illustrates the BSP tree and how points are queried for location within a primitive.
BSP-Tree For Rendering

- ordering primitives back-to-front

```c
void DrawBackToFront(n,p){
    if(IsLeaf(n))
        Draw(n);
    if( InNegativeHS(p,n) )
        DrawBackToFront(RightChild(n),p);
    else
        DrawBackToFront(LeftChild(n),p);
}
```
BSP-Tree For Rendering

- **Not so fast:** set of polygons not always separable by a plane
Auto-partition:
- use the extension of primitives as partition planes
- Store the primitive used for PP in the node
Building a BSP-Tree

- Building a BSP-tree requires to choose the partition plane

- Choose the partition plane that:
  - Gives the best balance?
  - Minimize the number of splits?
  - ……it depends on the application

- Cost of a BSP-Tree

\[
C(T) = 1 + P(T_L) \cdot C(T_L) + P(T_R) \cdot C(T_R)
\]

Cost of visiting \( T_{L[R]} \)

Probability that \( T_{L[R]} \) is visited if \( T \) has been visited
Building a BSP-Tree: example

\[ C(T) = 1 + P(T_L) \cdot C(T_L) + P(T_R) \cdot C(T_R) \]

\[ C(T) = 1 + |S_L|^{\alpha} + |S_R|^{\alpha} + \beta s \]

\( S_{L[R]} = \text{number of primitives in the left [right] subtree} \)

\( s = \text{number of primitives split by the chosen plane} \)

- \( \alpha, \beta \text{ used for tuning} \)
  - Big alpha, small beta yield a balanced tree (good for in/out test)
  - Big beta, small alpha yield a smaller tree (good for visibility order)
Binary Space Partition-Tree (BSP)

- **Memory occupation:** $O(n)$
  - For each node:
    - $(d+1)$ floating point numbers (in $d$ dimensions)
    - 2 pointers to child node

- Cost of descending the three:
  - $d$ products, $d$ summations (dot product $d+1$ dim.)
  - 1 random memory access (follow the pointer)

- Less general data structures can be faster/less memory consuming
kd-tree

- Kd-tree: k dimensions tree
- È una specializzazione dei BSP in cui i piani di partizione sono ortogonali a uno degli assi principali
- Scelte:
  - L’asse su cui piazzare il piano
  - Il punto sull’asse in cui piazzare il piano
- Vantaggi sui BSP:
  - determinare in quale semispazio risiede un punto costa un confronto
  - La memorizzazione del piano richiede un floating point + qualche bit
kD-Trees: esempio
kD-Trees : esempio
kD-Trees: esempio
kD-Trees: esempio
Costruire un kD-tree

- Dati:
  - axis-aligned bounding box (“cell”)
  - lista di primitive geometriche (triangoli)

- Operazioni base
  - Prendi un piano ortogonale a un asse e dividi la cella in due parti (in che punto?)
  - Distribuire le primitive nei due insiemi risultanti
  - Ricorsione
  - Criterio di terminazione (che criterio?)

- Esempio: se viene usato per il ray-tracing, si vuole ottimizzare per il costo dell’intersezione raggio primitiva
Costruire un kD-tree efficiente per RayCast

- In che punto dividere la cella?
  - Nel punto che minimizza il costo

- Quanto è il costo? Riprendiamo la formula per l’BSP

\[
\text{Cost}(\text{cell}) = 1 + \text{Prob}(\text{left _ cell} | \text{cell}) \cdot \text{Cost}(\text{Left}) + \text{Prob}(\text{right _ cell} | \text{cell}) \cdot \text{Cost}(\text{Right})
\]

![Diagram of kD-tree segmentation](Spatial-Search-Data-Structure)
Sapendo che il raggio interseca la cella \( \text{cell} \), qual’è la probabilità che intersechi la cella \( \text{left\_cell} \)?
Prob\(\left( left\_cell \mid cell \right)\)

\[
\text{Prob}[cell \mid left\_cell] = \frac{\#\text{raggi che intersecano left\_cell}}{\#\text{raggi che intersecano cell}}
\]

Ogni raggio che interseca una cella corrisponde a una coppia di punti sulla sua superficie. Contiamo le coppie di punti sulla superficie delle celle.

\[
\text{Prob}[cell \mid left\_cell] = \frac{\int \int da}{\int \int \sigma_{cell} da} = \frac{\text{Area}(left\_cell)^2}{\text{Area}(cell)^2} = \frac{\text{Area}(left\_cell)}{\text{Area}(cell)}
\]
$cost(left\_cell)$

- Sapendo che il raggio interseca la cella $left\_cell$, qual’è il costo di testare l’intersezione con i triangoli?
- Si approssima con il numero di triangoli che toccano la cella

$Cost(left\_cell) = 4$
Esempio

Come si suddivide la cella qui sotto?
A metà

- Non tiene conto delle probabilità
- Non tiene conto dei costi
Nel punto mediano

- Rende uguali i costi di *left_cell* e *right_cell*
- Non tiene conto delle probabilità
Ottimizzando il costo

- Separa bene spazio vuoto
- Distribuisce bene la complessità
Range Query with kd-tree

- **Query**: return the primitives inside a given box

- **Algorithm**:
  - Compute intersection between the node and the box
  - If the node is entirely inside the box add all the primitives contained in the node to the result
  - If the node is entirely outside the box return
  - If the nodes is partially inside the box recur to the children

- **Cost**: if the leaf nodes contain one primitive and the tree is balanced:
  \[
  O(n^{\frac{1}{d}} + k)
  \]

  \(n\) number of primitives, \(d\) dimension

- \(O(n^{2d})\) possible results
Nearest Neighbor with kd-tree

- **Query:** return the nearest primitive to a given point \( c \)
- **Algorithm:**
  - Find the nearest neighbor in the leaf containing \( c \)
  - If the sphere intersect the region boundary, check the primitives contained in intersected cells
Quad-Tree (2d)

- The plane is recursively subdivided in 4 subregions by a couple of orthogonal planes.
Quad-Tree (2d): examples

- Widely used:
  - Keeping level of detail of images

MIP-map level 0
MIP-map level 1
MIP-map level 2
MIP-map level 3
MIP-map level 4
Quad-Tree (2d): examples

- Widely used:
  - Terrain rendering: each cross in the quadtree is associated with a height value
Oct-Tree (3d)

- The same as quad-tree but in 3 dimensions
Oct-Tree (3d) : Examples

- Processing of Huge Meshes (ex: simplification)
- Problem: mesh do not fit in main memory
- Arrange the triangles in a oct-tree
Oct-Tree (3d) : Examples

- Extraction of isosurfaces on large dataset
  - Build an octree on the 3D dataset
  - Each node store min and max value of the scalar field
  - When computing the isosurface for alpha, nodes whose interval doesn’t contain alpha are discarded
Advantages of quad/oct tree

- Position and size of the cells are implicit
  - They can be explored without pointers (convenient if the hierarchies are complete) by using a linear array where:

\[
\begin{align*}
\text{Children}(i) &= 4i + 1, \ldots, 4(i+1) \\
\text{Parent}(i) &= \left\lfloor \frac{i}{4} \right\rfloor \\
\text{Children}(i) &= 8i + 1, \ldots, 8(i+1) \\
\text{Parent}(i) &= \left\lfloor \frac{i}{8} \right\rfloor
\end{align*}
\]

quadtree

octree
Z-Filling Curves

- Position and size of the cells are implicit
  - They can be indexed to preserve locality, i.e.

\[ (0,0) \rightarrow (1,0) \rightarrow (1,1) \rightarrow (0,1) \rightarrow (0,0) \]

Spatially close \(\rightarrow\) close in memory

Easy conversion between position in space and order in the curve

Just use the 0..1 coordinates as bits

00 01 10 11
Z-Filling Curves

- Position and size of the cells are implicit
  - They can be indexed to preserve locality, i.e.

  "Spatially close → close in memory"

Spatial Search Data Structure
Z-Filling Curves

- Conversion from spatial coordinates to index.
  - Write the coord values in binary
  - Interleave the bits

\[
x = b_0^x b_1^x b_2^x \ldots b_n^x
\]
\[
y = b_0^y b_1^y b_2^y \ldots b_n^y
\]
\[
id = b_0^y b_0^x b_1^y b_1^x b_2^y b_2^x \ldots b_n^y b_n^x
\]
Hierarchical Z-Filling Curves

Spatial Search Data Structure
Bounding Volumes Hierarchies

- If a volume B includes a volume A, it is called *bounding volume* for A
- No object can intersect A without intersecting B
- If two bounding volumes do not overlap, the same hold for the volumes included
The Principle

- What if they do overlap?
- Refine.
Questions!

- What kind of Bounding Volumes?
- What kind of hierarchy?
- How to build the hierarchy?
- How to update (if needed) the hierarchy?
- How to transverse the hierarchy?

All the literature on CD for non-convex objects is about answering these questions.
Cost

\[ T_c = N_v*C_v + N_n*C_n + N_s*C_s \]

\( v \) : visited nodes
\( n \) : couple of bounding volumes tested for overlap
\( s \) : couple of polygons tested for overlap
\( N \) : number of
\( C \) : Cost
BHV - Desirable Properties (2)

- The hierarchy should be able to be constructed in an automatic predictable manner
- The hierarchical representation should be able to approximate the original model to a high degree or accuracy
  - allow quick localisation of areas of contact
  - reduce the appearance of object repulsion
BHV - Desirable Properties

- The hierarchy approximates the bounding volume of the object, each level representing a tighter fit than its parent.
- For any node in the hierarchy, its children should collectively cover the area of the object contained within the parent node.
- The nodes of the hierarchy should fit the original model as tightly as possible.
Sphere-Tree


- Nodes of BVH are spheres.
- Low update cost $C_u$
  - translate sphere center
- Cheap overlap test $C_v$
  \[ D^2 < (R_1 + R_2)^2 \]
- Slow convergence to object geometry
  - Relatively high $N_u$ & $N_v$
Sphere-Tree Construction

Dingliana and O'Sullivan 2000

- Spheres placed around the boxes of a regular oct-tree

Spatial Search Data Structure
Sphere-Tree Construction

- Spheres placed along the Medial-Axis (transform)

Spatial Search Data Structure
Axis-Aligned Bounding Box

[van den Bergen 1997]

- The bounding volumes are axis aligned boxes (in the object coordinate system)
- The hierarchy is a binary tree (built top down)
- Split of the boxes along the longest edge at the median (equal number of polygons in both children)
Axis-Aligned Bounding Box

- The hierarchy of boxes can be quickly updated:
- let $Sm(R)$ be the smallest AABB of a region $R$ and $r_1, r_2$ two regions.

$$Sm(Sm(r_1) \cup Sm(r_2)) = Sm(r_1 \cup r_2)$$

- The hierarchy is updated in $O(n)$ time.
- Note: this is not the same as rebuilding the hierarchy.
If two **convex polyhedra** do not overlap, then there exists a direction $L$ such that their projections on $L$ do not overlap. $L$ is called Separating Axis.

**Separating Axis Theorem:** $L$ can only be one of the following:

- Normal to a face of one of the polyhedra
- Normal to 2 edges, one for each polyhedron
AABB - Overlap

Ex: There are 15 possible axes for two boxes: 3 faces from each box, and 3x3 edge direction combinations.

Note: SA is a normal to a face 75% of the times.

Trick: Ignore the tests on the edges!
Object Oriented Bounding Box

- Better coverage of object than AABB
  - Quadratic convergence
- Update cost $C_u$ is relatively high
  - reorient the boxes as objects rotate
- Overlap cost $C_v$ is high
  - Separating Axis Test tests for overlap of box’s projection onto 15 test axes

[Gotzchalk et al. 1996]
Oriented Bounding Box

[Gottschalk et al. 1996]

AABB

OBB

Spatial Search Data Structure
Building an OBB

- The OBB fitting problem requires finding the orientation of the box that best fits the data
- **Principal Components Analysis:**
  - Point sample the convex hull of the geometry to be bound
  - Find the mean and covariance matrix of the samples
  - The mean will be the center of the box
  - The eigenvectors of the covariance matrix are the principal directions – they are used for the axes of the box
  - The principle directions tend to align along the longest axis, then the next longest that is orthogonal, and then the other orthogonal axis
Principal Component Analysis

\[ c = \frac{1}{3n} \sum_{h=1}^{n} p^h \]

\[ \text{Cov}_{ij} = \frac{1}{3n} \sum_{h=1}^{n} (p^h_i - c_i)(p^h_j - c_j) \]

Cov is symmetric \( \Rightarrow \) eigen vectors form an orthogonal basis

Spatial Search Data Structure
Discrete Oriented Polytope

[Klosowski et al. 1997]

- Convex polytope whose faces are oriented normal to $k$ directions:
  - Overlap test similar to OBB
    - $k/2$ pairs of co-linear vectors
    - $k/2$ overlap tests
  - $k$-DOP needs to be updated in a similar way as the AABB
  - AABB is a 6-DOP
K-Dops examples

6-dop  14-dop  18-dop  26-dop
Discrete Oriented Polytope

[Klosowski et al. 1997]

AABB                                OBB                          6-DOP

Spatial Search Data Structure