**Geometric Hashing** - on the whiteboard

**Binary Space Partitions (BSP)**

**BSP and the painter algorithm**

**Quad trees and R-trees**

---

**BSP tree**

> Given a set of triangles \( S = \{ t_1, \ldots, t_n \} \) in 3D, a BSP \( T \) for \( S \) is an tree where

1. Each leaf stores a triangle \( t \).
2. Each internal (non-leaf) node \( v \) stores a plane \( h_v \) and pointers to two children \( v.\text{right}, v.\text{left} \).
3. All triangles in the subtree \( v.\text{left} \) are fully below \( h_v \), and all triangles in \( v.\text{right} \) are fully on or above \( h_v \).

See further example on the board.

Sometimes we need to split triangles to construct the BSP.

If a (perfect) BSP exist, then for any location of a viewer, we can use the painter algorithm.

Numerous other applications in graphics. (e.g. combine with imposers/billboards)

If the number of triangles above and below \( h_v \) are roughly the same, then the height is \( O(\log n) \).

---

**Quad Trees**

A data simple data structure for geometric objects (e.g. points, houses, an image, 3D scene)

Support efficiently a very wide variety of queries.

Hierarchical Partition of the scene

**QuadTrees**

Assume we are given a red/green picture defined a \( 2^n \times 2^n \) grid. E.g. pixels.

Each pixel is either green or red.

(more general and interesting examples – soon)

Need to represent the shape “compactly”

Need a data structure that could answers multiple types of queries. For example:

1. For a given point \( q \), is \( q \) red or green ?
2. For a given query disk \( D \), are there any green points in \( D \) ?
3. How many green points are there in \( D \) ?
4. Etc etc.

---

**QuadTrees**

Alg \( \text{ConstructQT} \) for a shape \( S \).

- **input** – a node \( v \in T \) and a shape \( S \).
- **Output** – a Quadtree \( T_v \) representing the shape of \( S \) within \( R(v) \).

- If \( S \) is fully green in \( R(v) \), or \( S \) is fully red in \( R(v) \) – then
  - \( v \) is a leaf, labeled Green or Red. Return ;
- Otherwise, divide \( R(v) \) into 4 equal-sized quadrants, corresponding to nodes \( v.NW, v.NE, v.SW, v.SE \).
- Call \( \text{ConstructQT} \) recursively for each quadrant.

---

**3D-Quadtrees**

There are no 3D Quadtrees. We call them Oct-trees

Each node is either a leaf, or is split into 8 equal-volume octants.
QuadTrees

Consider a picture stored on an $2^h \times 2^h$ grid. Each pixel is either red or green.

We can represent the shape "compactly" using a QT.

Height – at most $h$.

Point location operation – given a point $q$, is it black or white
- takes time $O(h)$
- could it be much smaller?

Many other operations are very simple to implement.

QuadTrees for a set of points

Now consider a set of points (red) but on a $2^h \times 2^h$ grid.

Splitting policy: Split until each quadrant contains $\leq 1$ point.

- Build a similar QT, but we stop splitting a quadrant when it contain $\leq 1$ point (or some other small constant).
- Could be easily built by inserting the points one after the other. A leaf is split if it contains 2 points.
- Point location operation – given a point $q$, is it black or white
- takes time $O(h)$ (and less in practice)
- Many other splitting policies are very simple to implement. (e.g. a leaf could contain contains $\leq 17$ points)

QuadTrees for shape

Input: A set $S$ of triangles $S = \{t_1, t_2, \ldots\}$.
- Each leaf $v$ stores a list $v.$TriangleList of all triangles intersecting $R(v)$.
- Splitting policy: Split a quadrant if it intersects more than 5 (say) triangle of $S$.

Note – a triangle might be stored in multiple leaves. Some leaves might store no triangles.

Finding all triangles inside a query region $Q$. We essentially use the function $Report(Q,v)$ from the previous slide (with minor modifications)

Storing the range $R(v)$ of a node

Each node $v$ is associated with a range $R(v)$ – a square. The node $v$ stores (in addition to other info) 4 values

$(\text{MinX}, \text{MinY}), \text{coordinates of the lower left corner of } R(v)$$($\text{MaxX}, \text{MaxY}), \text{coordinates of the upper right corner of } R(v)$

Ray tracing and QuadTrees

Consider a quadrant $R(v)$ with corners $LL=(x_{\text{min}},y_{\text{min}})$ and $UR=(x_{\text{max}},y_{\text{max}})$.

To find if a ray $\mathbf{r} = r_0 + t \cdot \mathbf{d}$ intersects this quadrant
- Find $t_{\text{min}}_x, t_{\text{max}}_x$, where the ray is in the x-span of the quadrant (the vertical slab containing the quadrant). This is easy, since we only need to check the x-component of $\mathbf{d}$. If $t_{\text{min}} > 0$ then this ray does not intersect $R(v)$.
- Find $t_{\text{min}}_y, t_{\text{max}}_y$, where the ray is in the y-span of the quadrant.
- Set $t_{\text{min}}=\min(t_{\text{min}}_x, t_{\text{min}}_y)$
- Set $t_{\text{max}}=\max(t_{\text{max}}_x, t_{\text{max}}_y)$
- The ray is inside the quadrant only for $t \in [t_{\text{min}}, t_{\text{max}}]$

In 3D, we also check $t_{\text{min}}_z, t_{\text{max}}_z$. 

QuadTrees for a set of points

Report(Q,v) {
  // v – a node of a quad tree that stores a set of points (e.g. $S=\{a,b,c,d\}$)
  // v- node of a quad tree that stores a set $S$ of points ($S=\{a,b,c,d\}$)
  Points all data point the points in stored at the subtree rooted at $v$, which also inside $Q$.
  1. If $R(v)$ is disjoint from $Q$, return // no point to report
  2. If $R(v)$ is fully contained in $Q$, print all of $S$ at the subtree rooted at $v$.
  3. Else if $R(v)$ partially overlaps $Q$, {   
    // v is a leaf - check each point in $R(v)$ if inside $Q$ and print if yes.
    Else // v is internal node
    Report(Q, NW(v)) and ..
    Report(Q, SE(v)) and ..
    Report(Q, SW(v)) and ..
    Report(Q, NE(v));
  }
}
Ray tracing and QuadTrees

Now, it is easy to find the first triangle hit by a ray $r$:
- Start from $v=root$. If empty, then continue tracing the ray from the point it leaves the quadrant.
- If $v$ is an internal node, check which of its quadrants is first hit by $r$, and continue recursively.
- If $v$ is a leaf, check each triangle in $v$.

Inserting a new triangle

```c
insert(triangle $t$, node $v$) {
  // Inserting a new triangle $t$ into an existing node $v$ of the Quadtree.
  // $v$ is not necessarily a leaf.
  if $v$ is NULL Error
  if $R(v)$ is disjoint from $t$ (share no points) - Return. Nothing to do.
  if $v$ is not a leaf, then for each child $u$ of $v$, call insert($t$, $u$);
  Else // $v$ is a leaf
    Add $t$ to $v$.TrianglesList
    if number of triangles in $v$.SegmentsList is too long (e.g. >5) Call Split($v$)
}
```

Split($v$)

```c
// Assumption – $v$ is a leaf, but has too many triangles in its list.
// Create 4 children for $v$ (make sure they know which regions they cover.)
for each child $u$ of $v$
  for each segment $s$ in $v$.TrianglesList
    Call insert($s$, $u$)
Empty $v$.TrianglesList
```

Terrain representations and levels-of-details

For every grid point $i,j$, given the elevation $z_{ij}$ (TIN – Triangulated Irregular Network)

- Each triangle approximately fits the surface below it

How to find good triangulation?

- Input - a very large set of points $S=\{(i,j,z_{ij})\}$.
- $z_{ij}$ is the elevation at point $(i,j)$ (latitude and longitude)
- Want to create a surface, consists of triangles, where each triangle interpolates the data points underneath it.
- Idea: Build a QT $T$ for the 2D points.
- (If want triangles: Each quadrant is split into 2 right-hand triangles)
- Assign to each vertex the height of the terrain above it.
- The approximated elevation of the terrain at any point $(x,y)$ is the linear interpolation of its elevated vertices.

QT Split Policy:
- Split a node $v$ if for some date point $(x_i,y_i)\in R(v)$, the elevation of $z_{ij}$ is too far from the the corresponding triangle. If not, leave $v$ as a leaf.
- That is, for any point $(i,j)$ on the plane, the elevation $(i,j,z_{ij})$ is too far from the interpolated elevation.
- Note: A quadrant might contain a huge number of points, but they behave smoothly. E.g. all a the slope of a mountain, but this slope is more or less linear.

Level Of Details

- Idea – the same object is stored several times, but with a different level of details
- Coarser representations for distant objects
- Decision which level to use is accepted `on the fly’ (eg in graphics applications, if we are far away from a terrain, we could tolerate usually large error. E.g., sub pixels error are not noticeable.)
In general, each internal node of level 2. Find the nearest pair of segments (say 7,8). Remove them from level 1, and replace them by a single BB encapsulate both. It corresponds to a node of level 2.

We compute for each segment its bounding box (rectangle).

Continue recursively to partition the subsets, until they are small enough.

R-trees

- Input: A set S of shapes (segments in this example. Triangles in graphics apps)
- Build a tree that could expedite
  - (i) finding the segments intersecting a query region,
  - (ii) answering ray tracing
  - (iii) Emptiness queries, etc

We start the search by visiting the root, then one of its children, one of its grand-children … until we reach a leaf.

Only small portion of the tree could be stored in the main memory.

Consider a very simplistic model of the computer memory – fast main memory, and slow secondary memory (your computer follows this model, probably with more than 2 types of memory, and probably SDD instead of disks, but this model still applies).

Large degree helps

Consider point location operation (find the segment containing a query point)

We start the search by visiting the root, then one of its children, one of its grand-children … until we reach a leaf.

Partitions 2D space into axis-aligned rectangular regions.

Nodes stores partition lines. However each node v corresponds to a region R(v) in the plane. (the reasons that nodes don’t need to store R(v) is that R(v) could be computed from by the path from the root to v and leaves represent input points.

Construction complexity:

\[ T(n) = \frac{(2^k - 1)(2^k)}{2} n \]

\[ T(n) = O(n \log n) \]

Height: \( O(\log n) \)

2D-Trees (and in higher dimension, kD-trees)

- Given a set of points in 2D.
- Bound the points by a rectangle.
- Split the points into two (almost) equal size groups, using a horizontal line, or vertical line, (first horizontal, then vertical, back to horizontal etc)
- (in \( \mathbb{R}^2 \), split by a plane orthogonal to the
  - \( x \)-axis,
  - then orthogonal to \( y \)-axis,
  - then \( z \)-axis,
  - and back to \( x \)-axis etc
- Continue recursively to partition the subsets, until they are small enough.

R-trees in practice. Memory Hierarchy, and advantages of multiple children

- In practice, it is sometimes preferable to create trees with a very large degree (instead of binary trees).

- For example, each internal node, will have between 100 to 500 children

If each node contains about 1000 segments, or keys, then the height (and number of I/Os) is only

\[ \log_{1000} 500 \approx 3.0 \]

\[ \log_{1000} 1000 \approx 3.0 \]

\[ \log_{1000} 10000 \approx 3.0 \]

\[ \log_{1000} 100000 \approx 3.0 \]

\[ \log_{1000} 1000000 \approx 3.0 \]

If the stored items are 1-dimensional (rather than multi-d), then B-trees are used instead of R-trees. They are very convent for insertion/deletion and other operations.
We saw a family of hierarchical trees

```plaintext
Report(Q,v) {
  //
  Q – a query disk, v – node of a quad tree that stores a set S of points (e.g. S={a,b,c,d}).
  Prints all data points the points in stored at the subtree rooted at v, which are also inside Q.
  /*
  1. If R(v) is disjoint from Q - return // no point to report at v's subtree
  2. If R(v) is fully contained in Q – print all of S stored at the subtree rooted at v.
  3. Else // partially overlaps Q. {
      • If v is a leaf – check each point in R(v) if inside Q and print if yes.
      • Else // v internal node
        • Report(Q, NW(v)) and ...
        • Report(Q, NE(v)) and ...
        • Report(Q, SW(v)) and ...
        • Report(Q, SE(v)) }
}
```

Def: Hierarchical tree tree T : R(v) ⊆ R(parent(v)) for every non-root node v.
Where R(v) is the region of node v.
Example, Quadtree, R-tree, KD-tree

Augmenting the tree: We can store at each internal node of v of T additional information that relates to data in the subtree of v. For example:
1. The number of data points in its v's subtree
2. Max,
3. Min,
4. sum-of-RGB
5. etc

Question: Which values should we maintain so we could find the average efficiently?

A word about theoretical guarantees

- All these trees are very efficient for realistic data and queries. Most regions in the tree are either fully inside the query or fully outside.
- All works well in 3D and 4D.
- As far as theoretical guarantees goes, bounds are less striking. Every though, we can prove: In a KD-tree, a query with axis-parallel query region visits at most \( O(\sqrt{n}) \) (in 2D) and \( O(n^{2/3}) \) (in 3D).