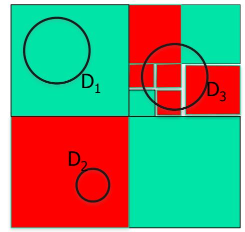
Quadtrees and R-trees

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 $2^h \times 2^h$ grid. E.g. pixels.
Each pixel is either **green** or **red.**

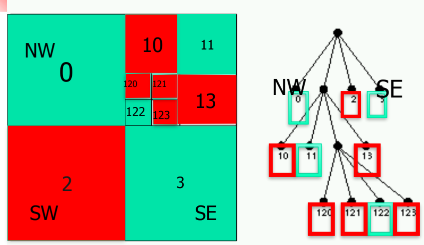
(more general and interesting examples – soon)

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



- Assume we are given a red/green picture defined on a $2^h \times 2^h$ grid of **pixels**.
- Each pixel has as a unique color (Green or Red)
- Every node $v \in \boldsymbol{T}$ is associated with a geometric region ***R(v)*** .
- This is the region that *v is* "in charge of".

Alg **<u>ConstructQT</u>** <u>for a shape S</u>.

•**input** – a node $v \in \boldsymbol{T}$, and a shape *S.*

•**Output** – a Quadtree $\boldsymbol{T}_v$ representing the shape of *S within R(v)*).

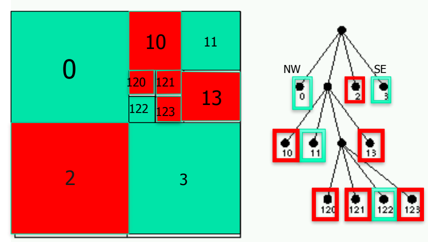
• If *S* is fully green in *R(v),* or S is fully red in *R(v)* – then

• *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 **ConstructQT** recursively for each quadrant.

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)

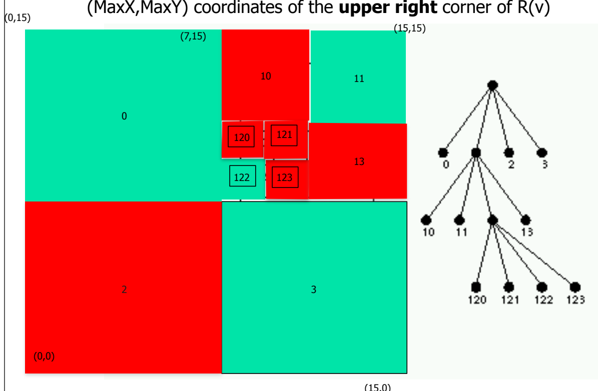
- could it be much smaller ?

Many other operations are very simple to implement.

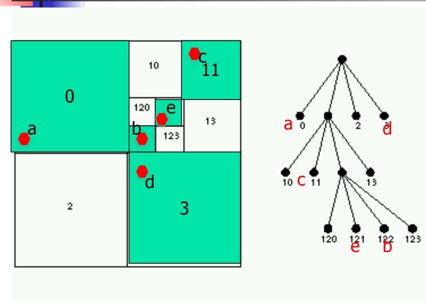
Storing the **range** R(v) of a node

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

(MinX,MinY) – coordinates of the **lower left** corner of R(v)
(MaxX,MaxY) coordinates of the **upper right** corner of R(v)



QuadTree 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 ≤1 point.

Build a similar QT, but we stop splitting a quadrant when it contain ≤1 point (or some other small constant)

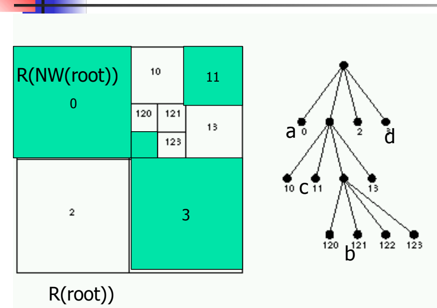
Point location operation – given a point q, is it black or white

– takes time O(h) (and less in practice)

Many other splitting polices are very simple to implement.

(eg. A leaf could contain contains ≤17 points)

Regions of nodes



In general, every node v is associated with a region R(v) in the plane

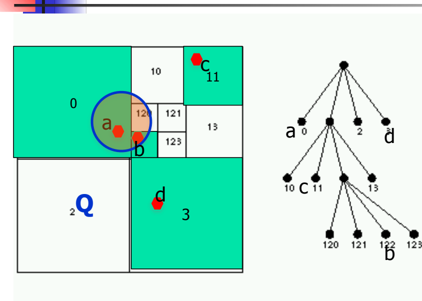
R(root) is the whole region

The smallest area of R(v) is a single pixel.

Let NW(v) denote the North West child of v. (similarly NE, SW, SE)

R(v) = is the union of
R(NW(v)), R(NE(v)) R(SW(v)), R(SE(v))

QuadTrees for a set of points



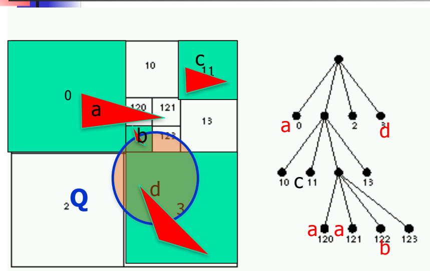
Report(Q,v)

// Q – a query disk

/*report all the points in stored at the subtree rooted at v, which are also inside Q. */

1. If v is NULL – **return**.
2. If R(v) is disjoint from Q – **return**
3. If R(v) is fully contained in Q – report all points in the subtree rooted at v.
4. If v is a leaf – check each point in R(v) if inside Q
5. Else
   - Report(Q, NW(v))
   - Report(Q, NE(v))
   - Report(Q, SW(v))
   - Report(Q, SE(v))

QuadTrees for shape



- Input: A set S of triangles $S=\{t_1 \ldots t_n\}$.
- 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*.

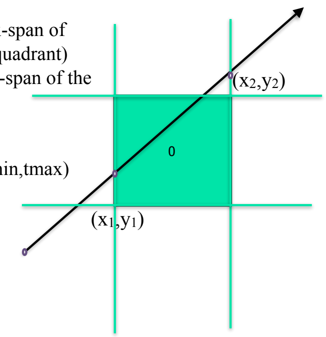
Note – a triangle might be stored in multiple leaves. Some leaves might store no triangles.

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

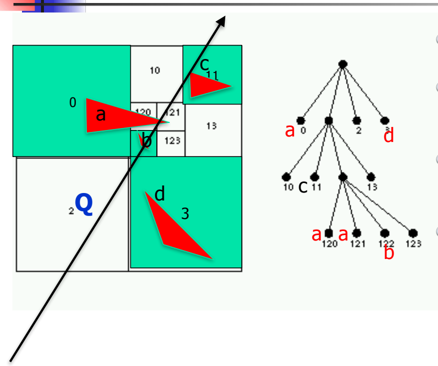
Ray tracing and QuadTrees

- Consider a quadrant with corners LL=(x_1,y_1) and UR=(x_2,y_2).
- To find if a ray r=p+tv intersects this quadrant
  - Find tmin_x, tmax_x, where the ray is in the x-span of the quadrant (the vertical slab containing the quadrant)
  - Find tmin_y, tmax_y, where the ray is in the y-span of the quadrant
  - Set tmin=max(tmin_x, tmin_y)
  - Set tmax=min(tmax_x, tmax_y)
  - The ray is inside the quadrant only for $t \in$ (tmin,tmax)

In 3D, we also check tmin_z, tmax_z



Ray tracing and QuadTrees



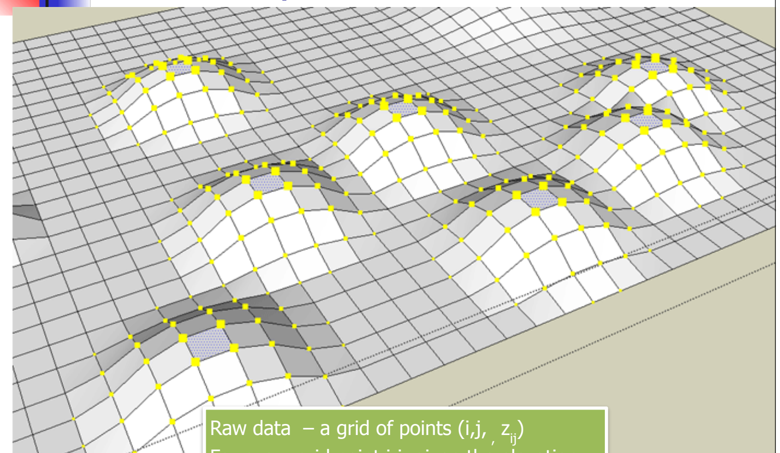
- Now, it is easy to find the first triangle hit by a ray r:
- Start from v=root. If empty, then continue tracing the ray from the point it leaves the quadrant.
- If v is internal node, check which of its quadrants is first hit by r, and continue recursively.
- If v=leaf, check each triangle in v

Inserting a new triangle

insert(triangle $\boldsymbol{t_i}$,node ***v***) {
    // Inserting a new triangle $\boldsymbol{t_i}$ 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 $\boldsymbol{t_i}$ (share no points)– Return. Nothing to do.
    If v is not a leaf, then for each child ***u*** of ***v***, call **insert($\boldsymbol{t_i}$,*u*)**;
    Else // v is a leaf
    Add $\boldsymbol{t_i}$ to ***v*.TrianglesList**
    If number of triangles in **v.SegmentsList** is too long (e.g. >5) Call **Split(v)**
}

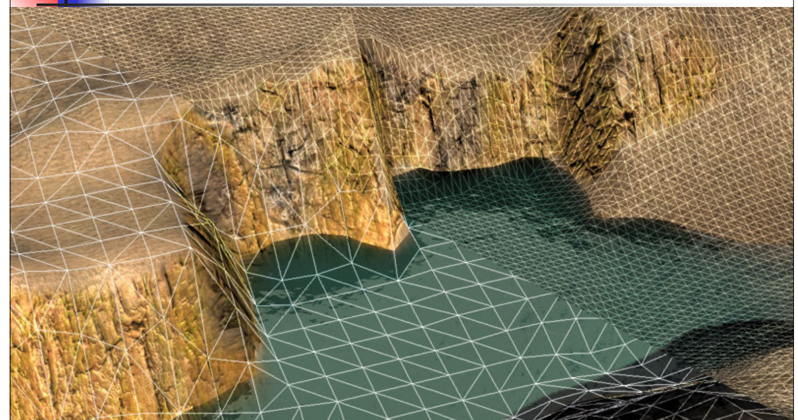
Split(v){
    // Assumption – v is a leaf, but has too many triangles in its list.
    // Create 4 children for v (make sure they know which regions they cover.)
    For each child ***u*** of ***v***
        For each segment **s** in **v.TrianglesList** Call **insert(s, u)**
    Empty **v.TrianglesList**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

Terrain representations



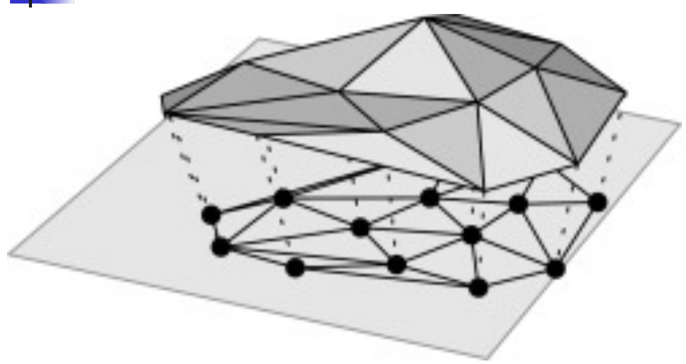
Raw data – a grid of points (i,j, z_{ij})
For every grid point i,j, given the elevation z_{ij}

(TIN – Triangulated Irregular Network)



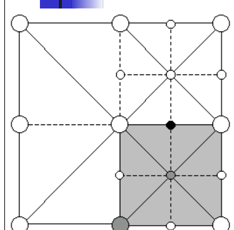
Each triangle approximately fits the surface below it

How to find good a triangulation ?



Each triangle approximately fits the surface below it
(credit SCALGO)

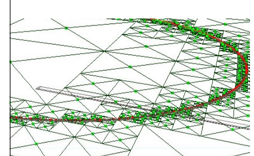
How to find good triangulation ?



- Input – a very large set of points $S=\{ (x_i , y_j , z_{ij}) \}$.
- z_{ij} is the elevation at point (x_i , y_i)
- 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 triangles)
- Assign to each vertex the height of the terrain above it.
- The approximated elevation of the terrain at any point is the linear interpolation of its elevated vertices.

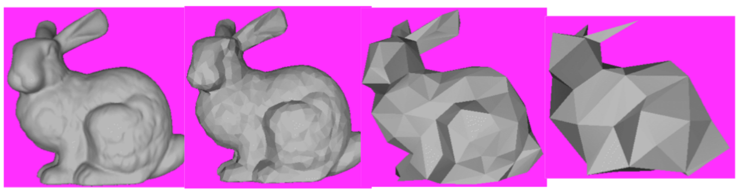
QT Split Policy: Splitting a quadrant into 4 sub-quadrants:

- 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, (x_i ,y_j , z_{ij}) it 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 sloop of a mountain, but this slope is more or less linear.



Level Of Details

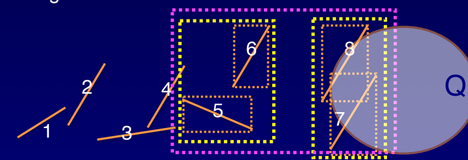
- Idea – the same object is stored several times, but with a different level of details
- Coarser representations for distant objects
- Decision which level to use is accepted `on the fly'
  (eg in graphics applications, if we are far away from a terrain, we could tolerate usually large error)



69,451 polys | 2,502 polys | 251 polys | 76 polys

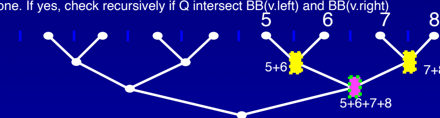
R-trees

- Input: A set S of shapes (segments in this example)
- Prepare a tree that could assists finding the segments intersecting a query region, answering ray tracing etc



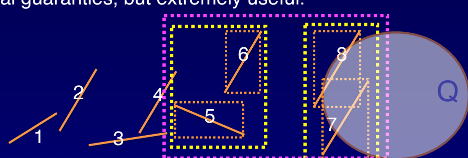
- We compute for each segment its bounding box (rectangle).
- These are the leaves of *T*
- Match pairs of bounding boxes. For example 1-2, 3-4, 5-6, 7-8. For each such pair, compute their bounding boxes. Each node in level 2 is such a box.
- Match these bounding boxes. These are the nodes of level 3.
- Repeat until we are left with one bounding box.
- In general for every node *v*, $BB(v) = BB\left(BB(v.right)\bigcup BB(v.left)\right)$

Once a query region Q is given we determine whether it intersect BB(root)
If not, we are done. If yes, check recursively if Q intersect BB(v.left) and BB(v.right)



R-trees

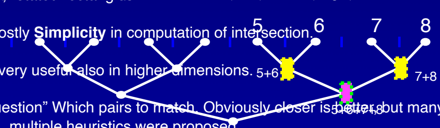
- Input: A set S of shapes (segments in this example)
- Prepare a tree that could assists finding the segments intersecting a query region.
- Fewer theoretical guaranties, but extremely useful.



- We compute for each segment its bounding bounding box (rectangle).
- These are the leaves of *T*
- Match pairs of bounding boxes. For example 1-2, 3-4, 5-6, 7-8. For each such pair, compute their bounding boxes. Each node in level 2 is such a box.
- Match these bounding boxes. These are the nodes of level 3.
- In general for every node *v*, $BB(v) = BB\left(BB(v.right)\bigcup BB(v.left)\right)$

Once a query region Q is given we determine whether it intersect BB(root)
If not, we are done. If yes, check recursively if Q intersect BB(v.left) and BB(v.right)

- Problem in reporting: Many "false alarms" : BBs that intersect Q while their segments don't.
- Should we use axis parallel BB instead of something that could "snag" better? For example, rotated rectangles ?
- Answer: Mostly **Simplicity** in computation of intersection.
- R-trees are very useful also in higher dimensions.
- Other big question" Which pairs to match. Obviously closer is better, but many variants and multiple heuristics were proposed



- Problem in reporting: Many "false alarms" : BBs that intersect Q while their segments don't.
- Should we use axis parallel BB instead of something that could "snag" better? For example, rotated rectangles ?
- Answer: Mostly **Simplicity** in computation of intersection.
- R-trees are very useful also in higher dimensions.
- Other big question" Which Paris to match. Obviously closer is better, But many variants Multiple heuristics