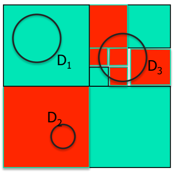
Quadtrees:

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

Support efficiently a very wide variety of queries.

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.

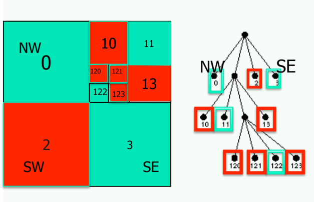
(more general and interesting examples – soon)

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

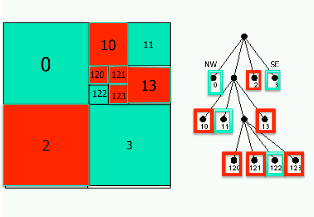


Assume we are given a red/green picture defined a $2^h \times 2^h$ grid. E.g. pixels.

Alg constructQT (input – a shape *R*. Output – a Quadtree corresponding to *R*).

- If *R* is fully green, or *R* is fully red– store as one (leaf) node *v*, labeled Green or Red. //Note: A pixel always have a unique color.
- Otherwise, divide the shape into 4 equal-size quadrants *NW,NE, SW, SE*.
- Call constructQT recursively for each quadrant.
- Create an internal node *v* having 4 children, corresponding to the 4 quadrants. Return *v*.

QuadTrees



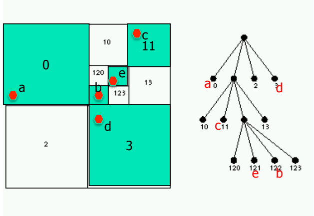
Consider a black/whie picture stored on an $2^h \times 2^h$ grid.

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



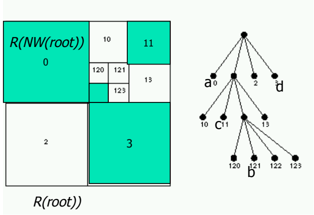
Now consider a set of points (red) but on a $2^h \times 2^h$ grid. ●

Splitting policy: Split until each quadrant contains ≤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



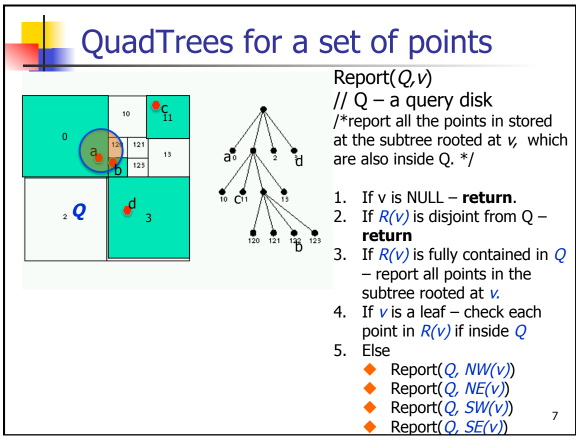
In general, every node v is associated with a region R(v) in the plane

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*)

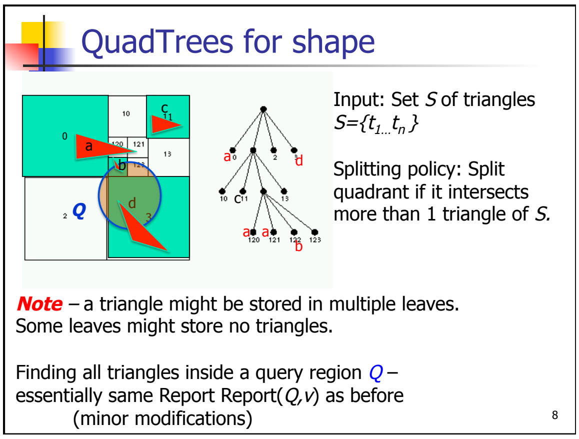
R(v) = is the union of
R(NW(v)), R(NE(v)) R(SW(v)), R(SE(v))



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))
```



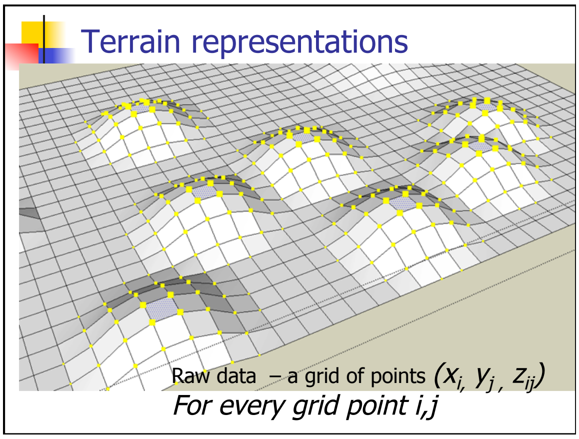
QuadTrees for shape

Input: Set *S* of triangles *S*={t_1…t_n}

Splitting policy: Split quadrant if it intersects more than 1 triangle of *S*.

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

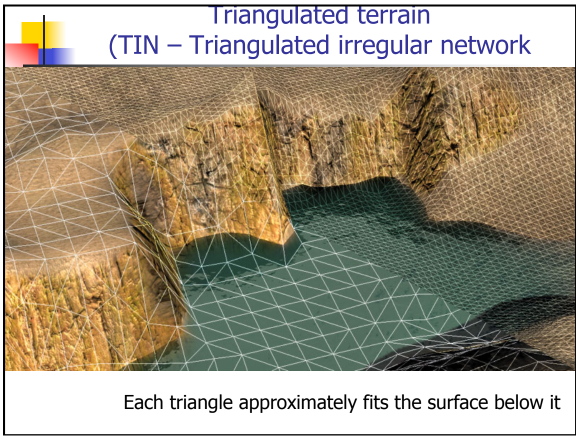
Finding all triangles inside a query region *Q* – essentially same Report Report(*Q*,*v*) as before (minor modifications)



Terrain representations

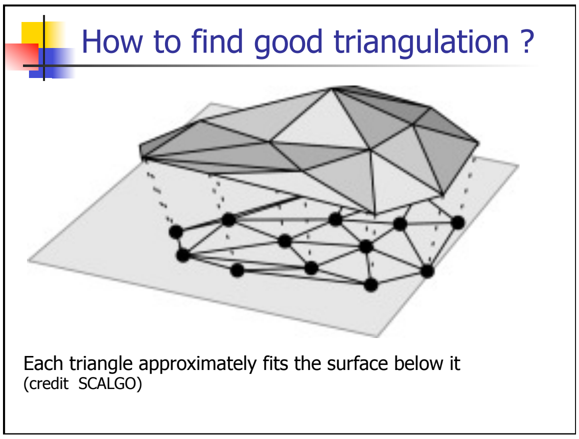
Raw data – a grid of points (x_i, y_j, z_{ij})

For every grid point i,j



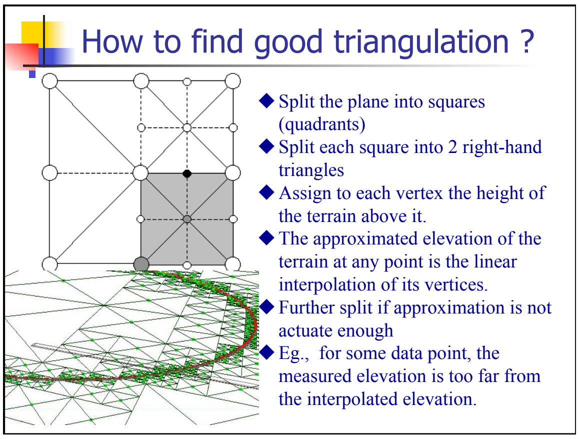
Triangulated terrain (TIN – Triangulated irregular network

Each triangle approximately fits the surface below it



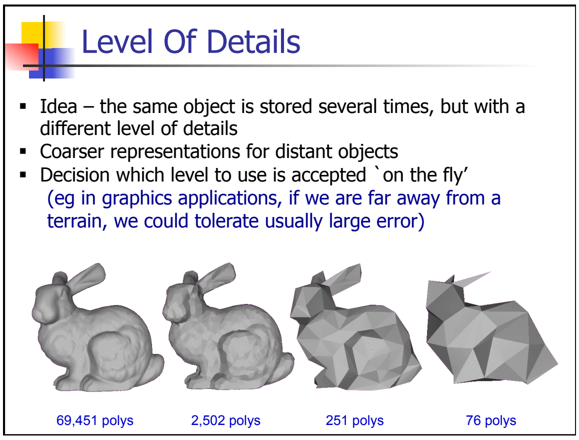
How to find good triangulation ?

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



How to find good triangulation ?

- ◆ Split the plane into squares (quadrants)
- ◆ Split each square 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 is the linear interpolation of its vertices.
- ◆ Further split if approximation is not actuate enough
- ◆ Eg., for some data point, the measured elevation is too far from the interpolated elevation.



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