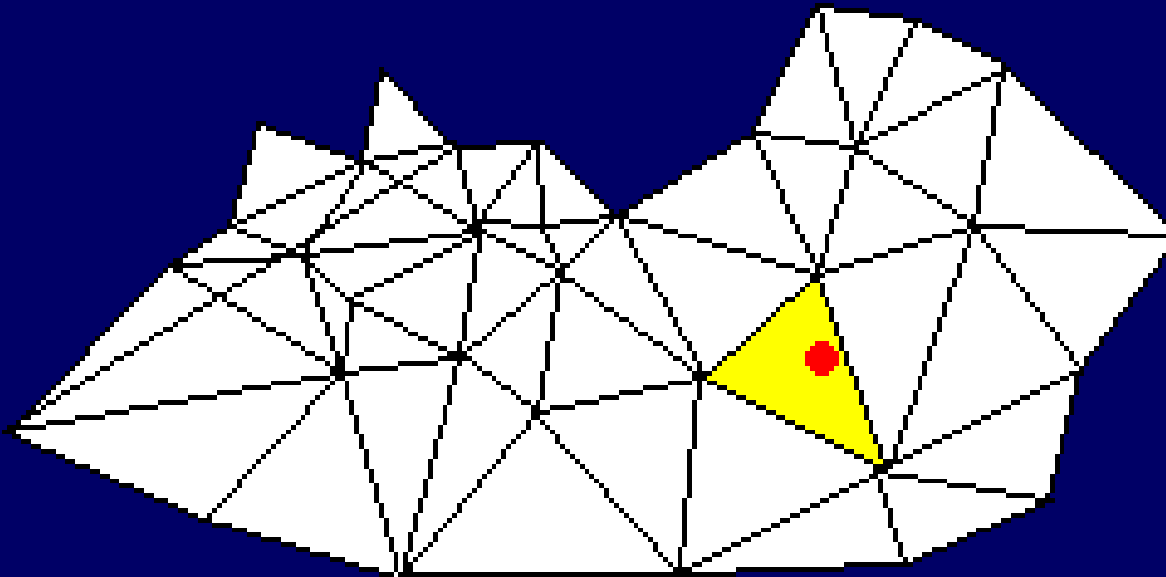


# Point Location in Delaunay Triangulations



# Inspiration:

Graphics software wants to light a model and needs to know which face a light ray hits

You moved the mouse and the windowing system would like to know which window has focus now.

Given latitude and longitude you need to determine which country/city/neighborhood contains your destination

# The problem at hand

Input:

- 1) A set of  $n$  points in  $\mathbb{R}^2$  (beforehand)
- 2) Additional points called query points

The task:

Create a delaunay triangulation using the initial set of points as an offline preprocessing step. Then for each query point, return the triangle which contains it.

And make the data fit in  $O(n)$  bits with query times of  $O(\log n)$ .

# Point location problems

Before going into the specifics there are many variants of this problem and I am only tackling a comparatively simple one. Some of the choices here:

Dimension:

2d? **The windows**

2d surface in 3d? **The graphics mesh**

K-d volumes in n-d space? **Someone probably has a use for this too.**

# Point location problems

Faces:

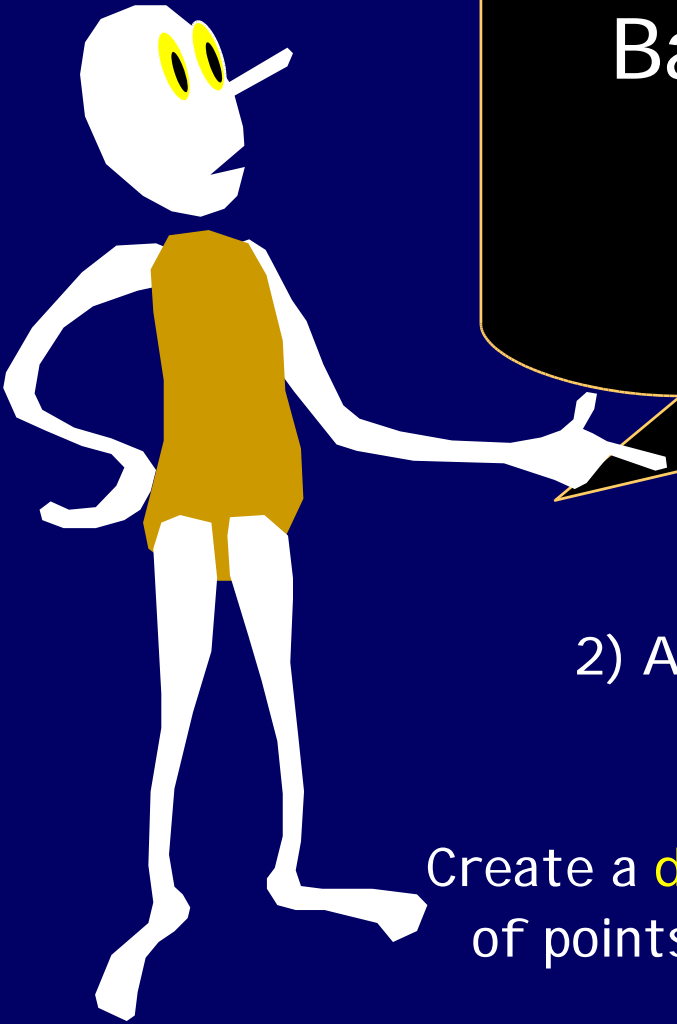
Triangles? Trapezoids? Any polygon?

Input:

Are we initially given the set of polygons to use? Or only the edges? Or the points?

Partial Input:

Sometimes we are allowed to add additional points to aid in making the polygons. Such points are called steiner points and they are generally added to try and create some features which were not present in the original polygon set. For example creating polygons bounded in size or minimum angle.



Back to our case though.  
What is a delaunay  
triangulation?

Input:

- 1) A set of  $n$  points in  $\mathbb{R}^2$
- 2) Additional points called query points

The task:

Create a **delaunay triangulation** using the initial set of points. Then for each query point, return the triangle which contains it.

# Delaunay Triangulations

A triangulation of a set of points is a planar graph which connects all the points and in which all faces are triangles.

Given the same set of points there are usually many different triangulations. The delaunay triangulation is (almost always) uniquely defined as one with a number of special properties...

# Delaunay Triangulation properties

The circumcircle of every triangle contains no other points.

Every line is also contained within some circle which contains no other points.

This triangulation maximizes the minimum angle found in any triangle over all other triangulations.

This triangulation is unique ... except when 4 or more points are co-circular. This is often avoided by ordering the points and defining later points to be outside the circle defined by earlier points. Thus we assume this case does not occur.



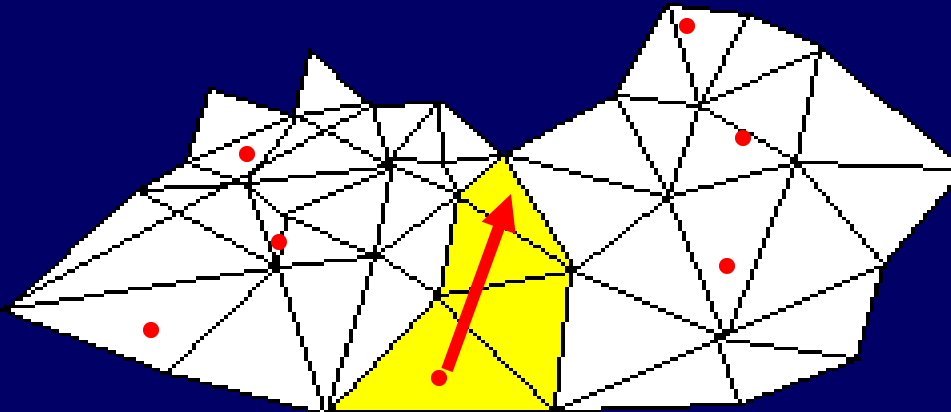
## A brief tour of (some of) the algorithms available to do this task.

- 2) An algorithm which 'walks' through the triangulation
  - Mucke, Saias, Zhu 96
- 2) An algorithm which looks through the history of operations which created the triangulation
  - Boissonat and Teillaud 86
- 3) An algorithm which searches through a DAG of triangles where larger triangles at the top contain smaller child triangles
  - Mulmuley 91
- 4) A hybrid algorithm that does both multi-level refinement like 3 and walking like 1.
  - Devillers 98

# 1) A Walking algorithm

An algorithm using only the final triangulation:

- a) Choose  $n^{1/3}$  random points beforehand and store which triangles they are in.
- b) When given a query point find which of the random points is closest.
- c) Use line-side tests to visit each triangle from the random point to the query point.



# 1) A Walking algorithm

A simple implementation uses  $O(n)$  pointers though Guy's recent paper on graph compression can be directly applied in this case to get  $O(n)$  bits

Query time is data-dependent. Usually close to  $n^{1/3}$  but some pathological cases can be far worse.

This algorithm typically is competitive with the best  $O(\log n)$  algorithms on small problems ( $n < 1,000$ ) because the constants are small.

## 2) Incremental history-based

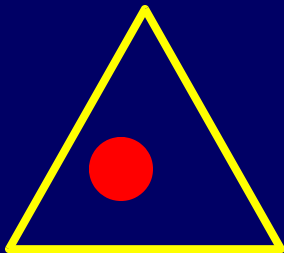
Randomly order the points, then insert the points one by one, each time updating the triangulation so that it is delaunay. We can do this by:

- Locating the triangle containing the point to be added
- Inserting the point and connecting it to nearby vertices
- c) Making 'edge flips' to return the triangulation to being delaunay.

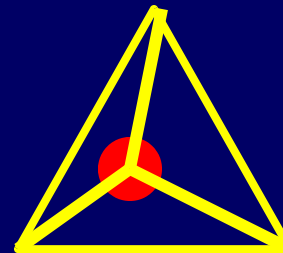
## 2) Incremental history-based

- a) Locating the triangle containing the point to be added ... **save this one for the moment**
- b) Inserting the point and connecting it to nearby vertices

Start with this:



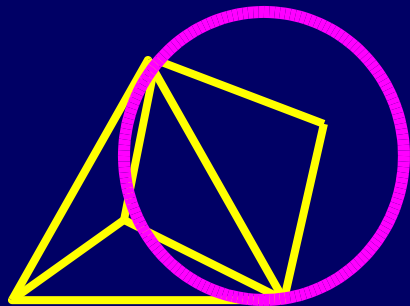
and end with this:



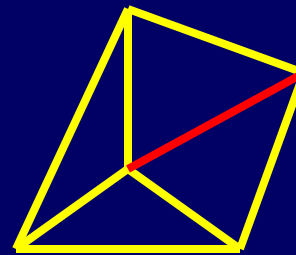
## 2) Incremental history-based

c) Making 'edge flips' to return the triangulation to being delauney.

Not delauney:



edge flip fixes this:



Must test adjacent triangles to any triangle which changes. Initially this is just the three incident on the new point, but edge flips can propagate into the neighboring triangles. Provably terminates.

## 2) Incremental history-based

We can keep a history of all triangles added and all edge flips made in a DAG. Each node will represent a triangle present at some phase of construction.

- When a new point is added the node for its containing triangle will point at 3 child nodes representing the subdivided triangle.
- When an edge flip occurs the two adjacent triangles will each point to the two new triangles.

## 2) Incremental history-based

a) Locating the triangle containing the point to be added

Location is performed by starting with the root node (triangle) and then testing which child triangle the point is contained in. Repeat all the way down the tree, and whatever leaf you wind up with is the containing triangle in the current triangulation.



## 2) Incremental history-based

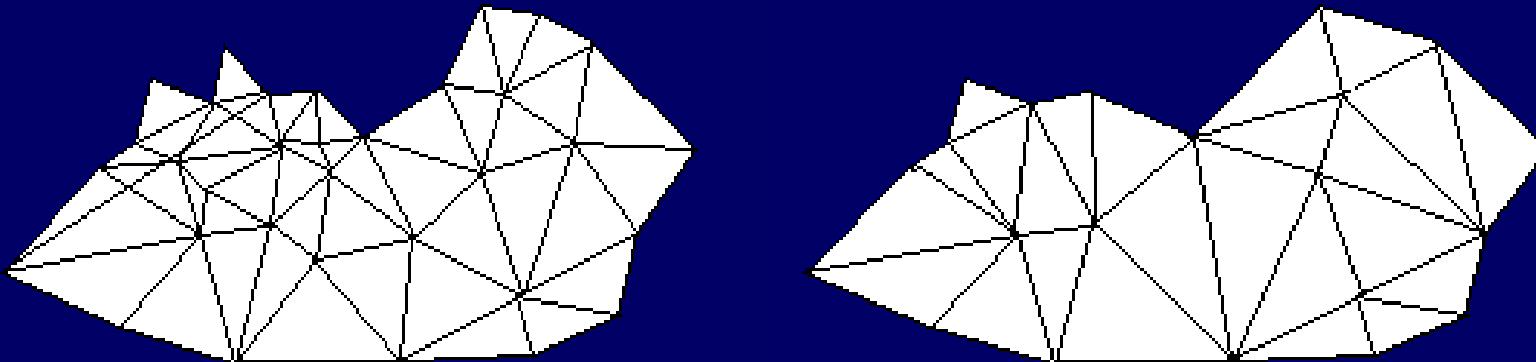
Space is expected  $O(n)$  pointers.

Experimentally this needs about twice as much memory as what would be needed to just store the final triangulation, but can fluctuate depending on the exact insertion order used.

Queries can be performed in expected  $O(\log n)$  time.

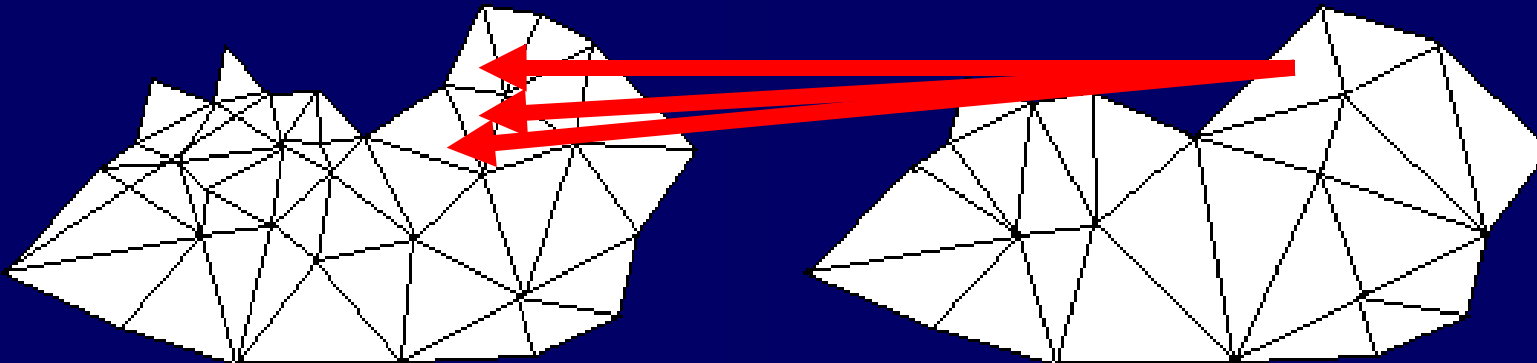
### 3) A Hierarchical Algorithm

This algorithm stores a set of delaunay triangulations, with varying levels of detail. At the lowest level it simply takes all the points and triangulates them. For each successive level it takes only a random sample of points from the current level.



### 3) A Hierarchical Algorithm

Then it links triangles with every triangle they overlap at the lower level. To locate a point in the final triangulation, locate it at the highest level, then search through all the children to find the proper triangle at the next level.



### 3) A Hierarchical Algorithm

Space is again  $O(n)$  pointers

Here however the space is not dependent upon the insertion order used.

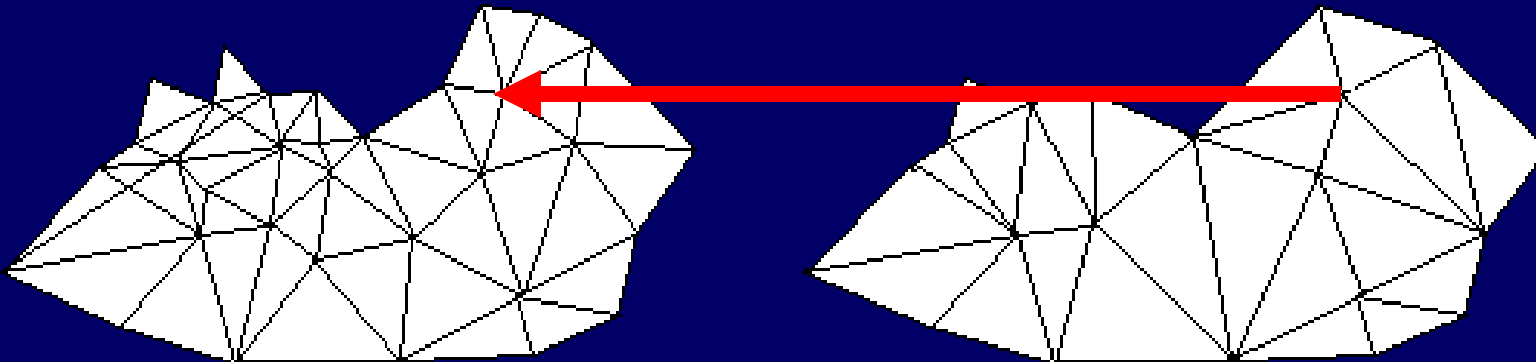
Queries take  $O(\log^2 n)$  time

(and a variant of this algorithm can do it in  $O(\log n)$  )

Its also much easier to add and delete points from this data structure

## 4) The multi-level hybrid

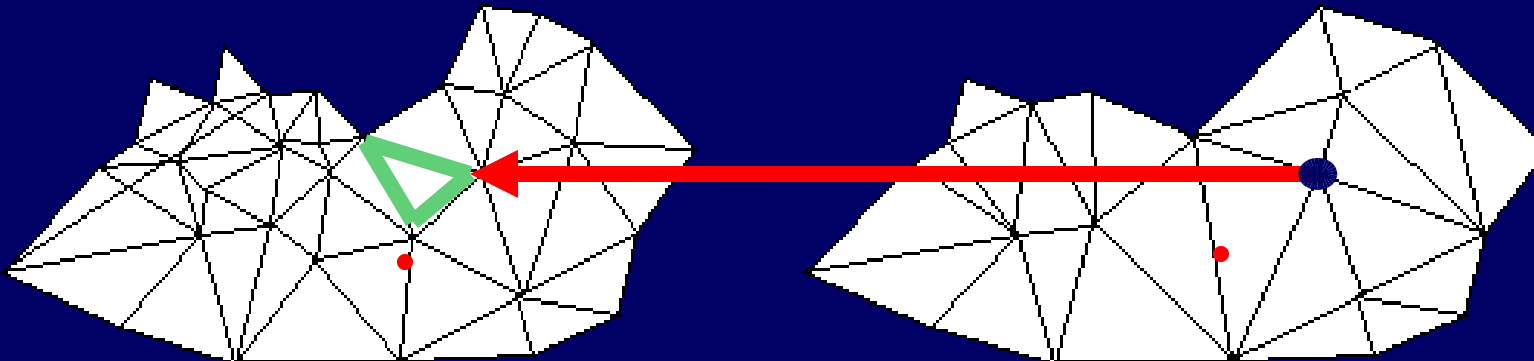
Here again we use random sub-sampling to select progressively smaller subsets of the input points and triangulate them. However rather than linking triangles between levels, the points are linked. The algorithm then proceeds to find the nearest input point to the query at every level.



## 4) The multi-level hybrid

To proceed between levels

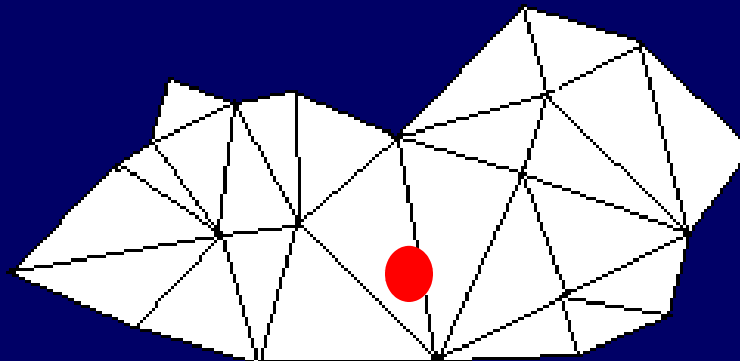
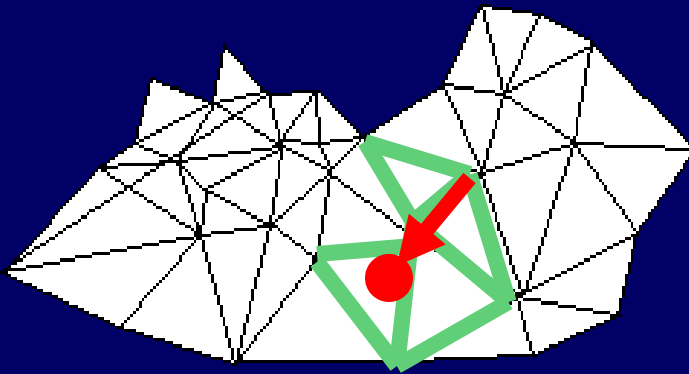
- b) Take the nearest input point to the query in the coarse level and move to an adjacent triangle at the finer level



## 4) The multi-level hybrid

To proceed between levels

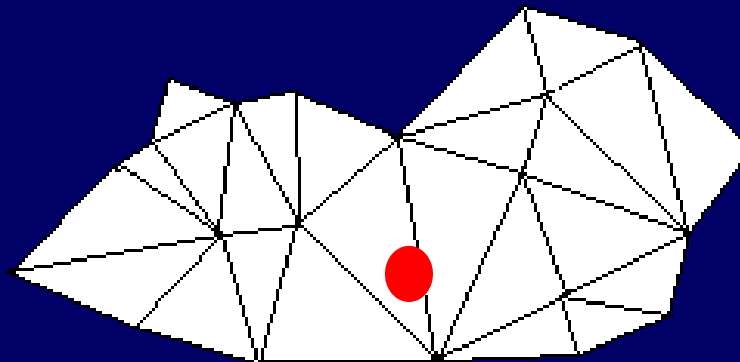
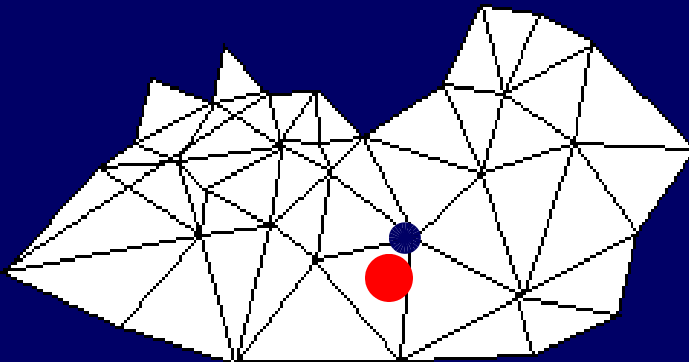
b) Walk to the triangle containing the query as in the first algorithm



## 4) The multi-level hybrid

To proceed between levels

- c) Find the nearest neighbor to query point at this level. It is not necessarily in the current triangle, but very rarely is it many triangles away. At the final level, just use the triangle and skip this step.





## 4) The multi-level hybrid

This algorithm also takes  $O(n)$  pointers.

It winds up being very little more than just storing the full triangulation, because the typical ratio of sizes between levels is 1:40.

The query time is  $O(\log n)$  and by using the random start point technique at the top level makes this algorithm competitive for all problem sizes.

## But how to get to $O(n)$ bits?

The most promising avenue (at the moment) appears to be compressing the data structure used by a pre-existing algorithm in a manner which does not affect the query time. The most likely candidate algorithm would be the last, as it is fast, and not an overly complicated data structure. Graph compression has been studied considerably as well, thus many types of graphs are already known to be compressible to  $O(n)$  bits.

# But how to get to $O(n)$ bits?

Triangulations have known representations as small as 3.67 bits/triangle worst case.

Unfortunately these representations don't support finding neighbors in constant time, so they would ruin the query times.

Another issue is that the multi-level schemes must maintain links between the different levels of triangulation (either triangle to triangle, or point to point) which also must be compressed.

# But how to get to $O(n)$ bits?

Dan Blandford, Guy Blelloch, and Ian Kash published a paper last summer that shows how to compress any 'separable' graph in  $O(n)$  bits while maintaining support for common operations in  $O(1)$  time.

Here separable means roughly that you can make a vertex cut of  $O(n^c)$  vertices for  $c < 1$  such that the graph is divided into two parts, each having at least  $1/3$  of the vertices. Furthermore you must be able to recursively do this on any sub-graph.

## But how to get to $O(n)$ bits?

All planar graphs are separable with only  $O(n^{1/2})$  vertices, thus compressing each level separately would be no trouble. The compression algorithm would label vertices at each level in such a way that neighboring vertices are close in number.

However if the label assignments are not correlated somehow between levels then it becomes impossible to link them in a compressed form (there are  $n^{cn}$  functions from  $cn$  points to  $n$  points, optimal compression wouldn't be good enough)

# But how to get to $O(n)$ bits?

Thus some of the main questions I am working on right now...

If all the triangulation levels and the links between are combined into one large graph, is it (provably) separable?

Or is there a way of finding good vertex cuts which work at every level simultaneously? This would allow the same vertex numbering to be used at each level.

Is there a way to number the vertices such that it doesn't break the graph compression algorithm, and leaves enough similarity between levels that only  $O(n)$  bits can encode all the links?