Scalar Algorithms: Contouring

Visualisation – Lecture 7

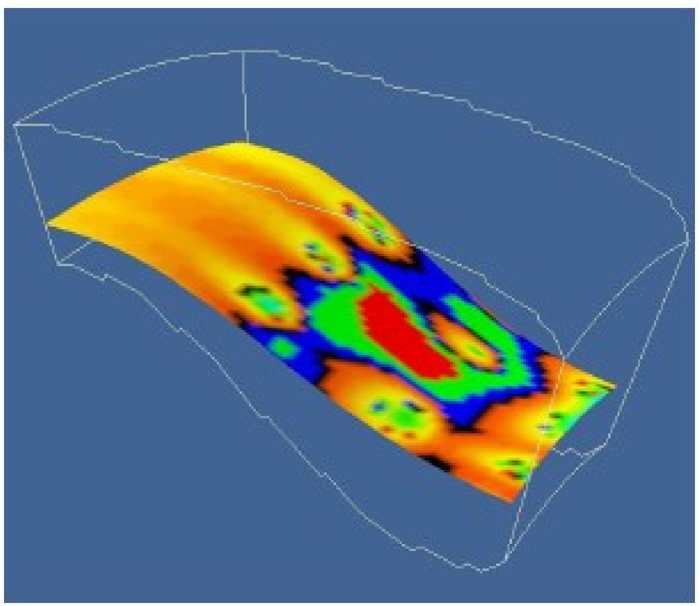
Taku Komura

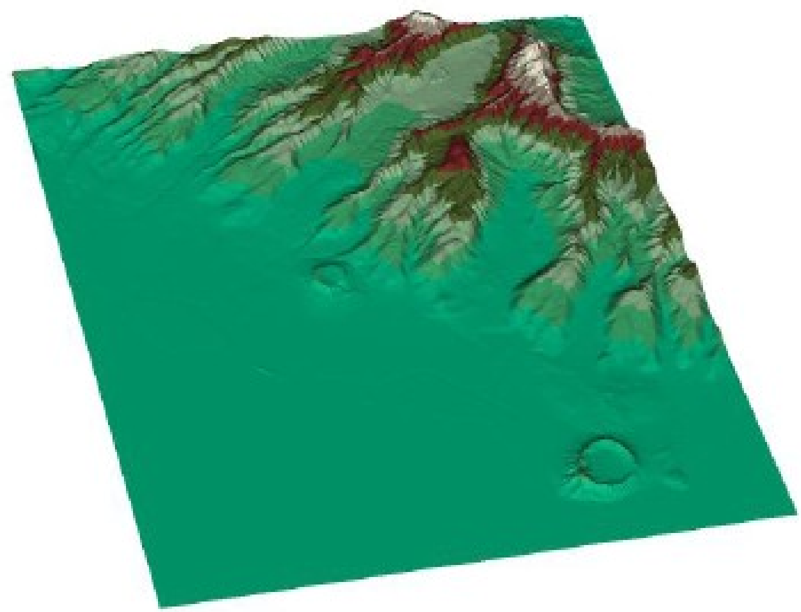
tkomura@inf.ed.ac.uk

Institute for Perception, Action & Behaviour
School of Informatics

Last Lecture ....

- **Colour mapping**
  - distinct regions identified by colour separation
  - transitions shown by colour gradients

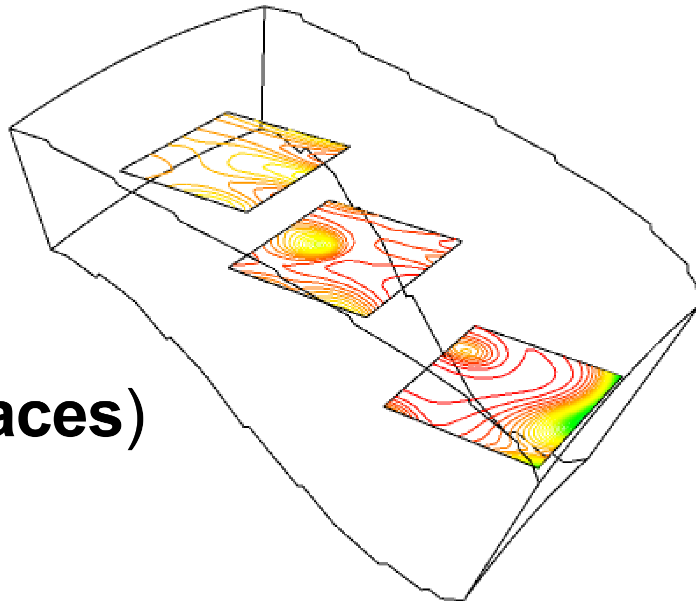




  - **eye separates coloured areas into distinct regions**

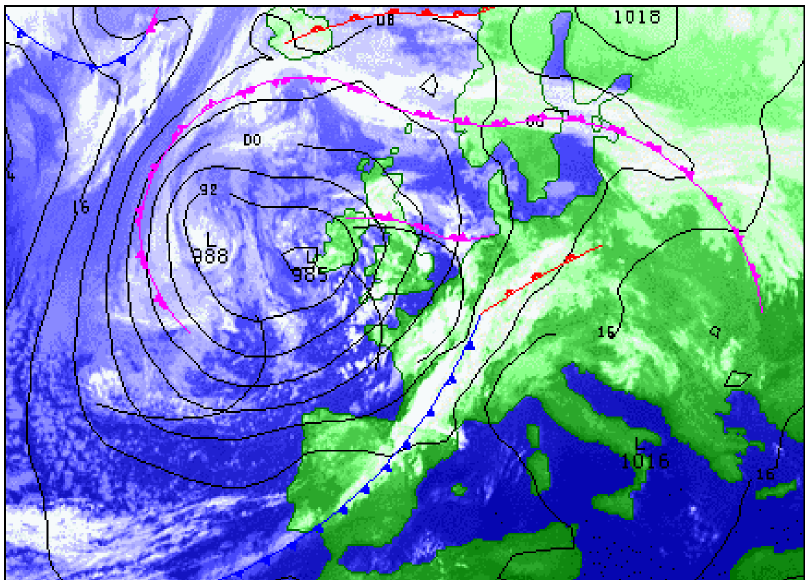
Contouring

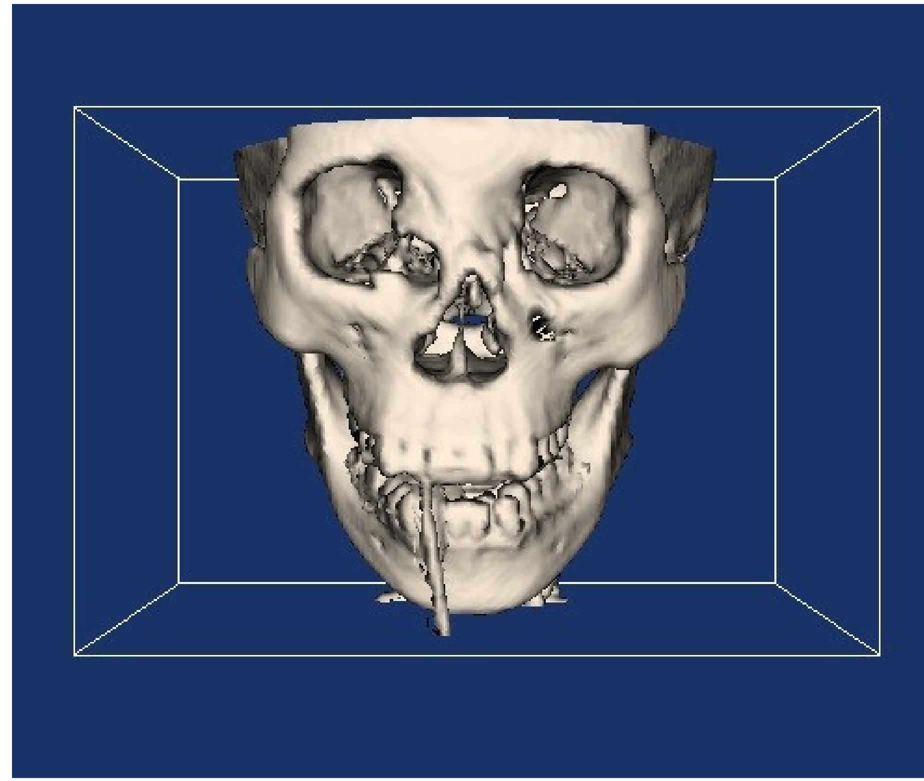
- Contours explicitly construct the **boundary between these regions**



- Boundaries correspond to:
  - **lines in 2D**
  - **surfaces in 3D** (known as **isosurfaces**)
  - of **constant scalar value**

Example : contours

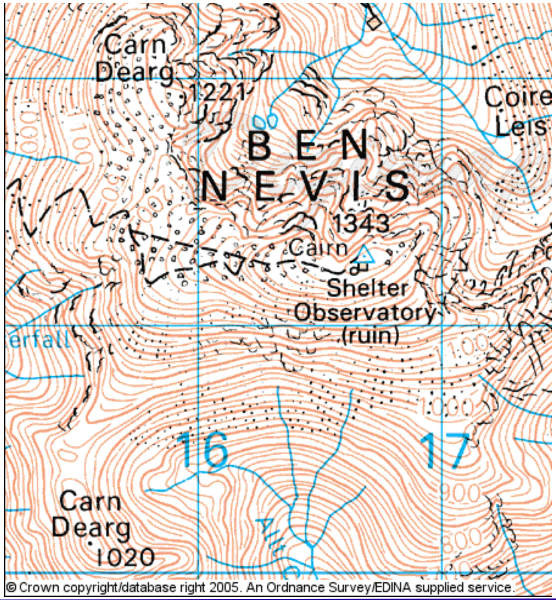




- lines of constant pressure on a weather map (isobars)
- surfaces of constant density in medical scan (isosurface)
  - “iso” roughly means equal / similar / same as

Contours

- Contours **are** boundaries between regions
  - they **DO NOT just** connect points of equal value
  - they **DO also** indicate a **TRANSITION** from a value below the contour to a value above the contour



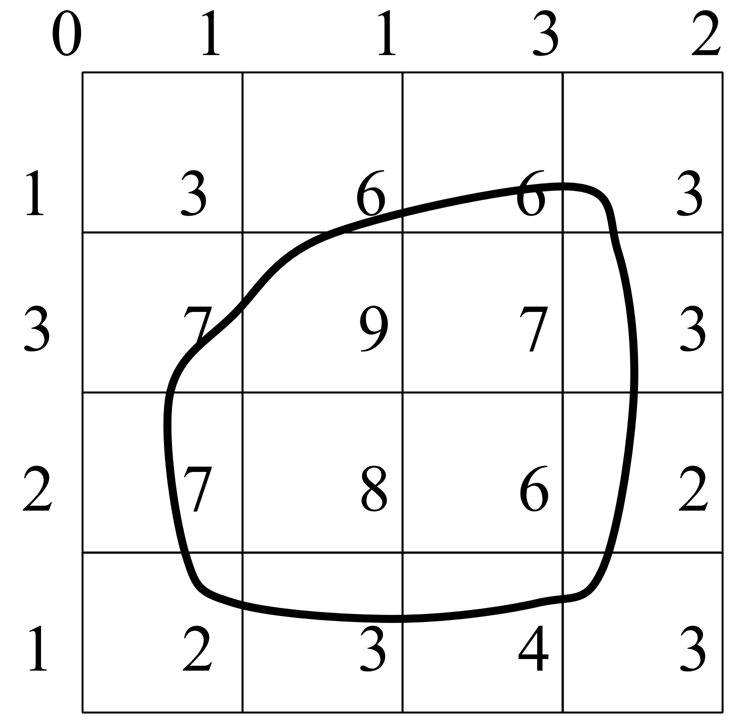
2D contours

- **Data :** 2D structured grid of scalar values

0	1	1	3	2
1	3	6	6	3
3	7	9	7	3
2	7	8	6	2
1	2	3	4	3

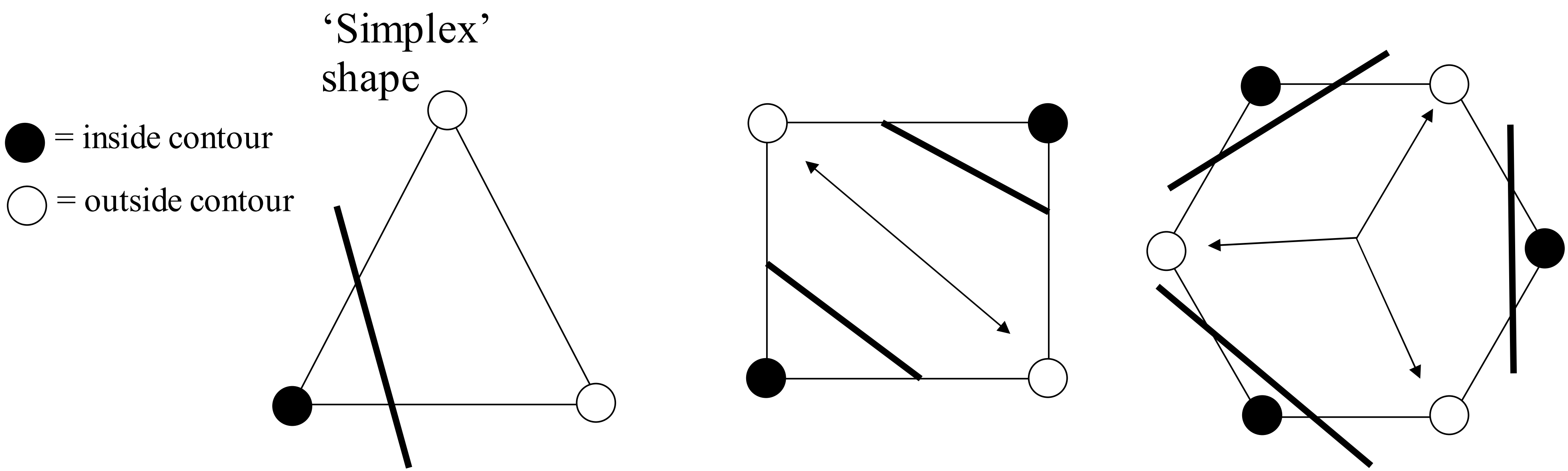
- Difficult to visualise transitions in data
  - use **contour** at specific scalar value to highlight **transition**

2D contours : line generation



- **Select scalar value**
  - corresponds to contour line
    - i.e. contour value, e.g. 5 (right)
- **Interpolate contour line**
  through the grid corresponding to this value
  - must interpolate as **scalar values at finite point locations**
  - true **contour transition may lie in-between point values**
  - **simple linear interpolation** along grid edges

2D contours : ambiguity



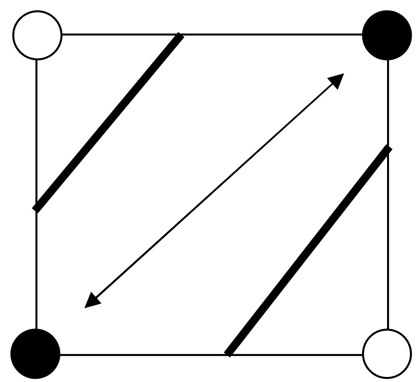
No vertices not connected by a shared edge.
→ **No ambiguous cases.**

Two vertices not connected by a shared edge.
→ **Two ambiguous cases.**

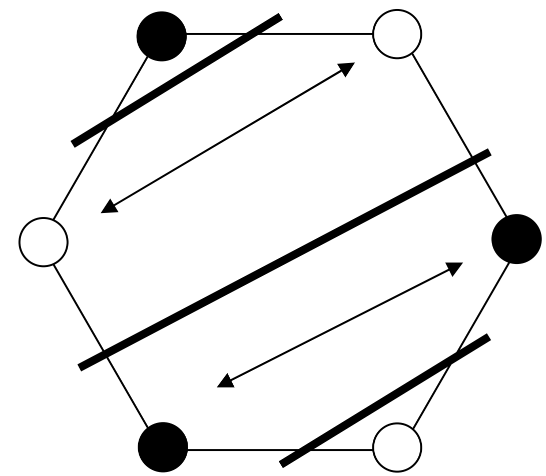
Three vertices not connected by a shared edge.
→ **Three ambiguous cases.**

- Ambiguous cases in contouring
  - **Q :** Are these points connected across the cell (join) or not (break) ?

2D contours : ambiguity



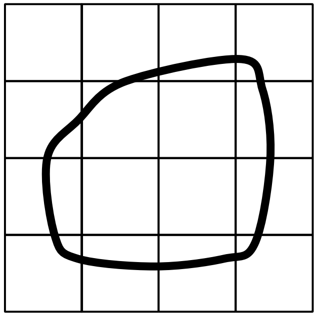
Two vertices not connected by a shared edge.
→ **Two ambiguous cases.**



Three vertices not connected by a shared edge.
→ **Three ambiguous cases.**

- Alternative contouring of same data
  - hence ambiguity

Methods of Contour Line Generation



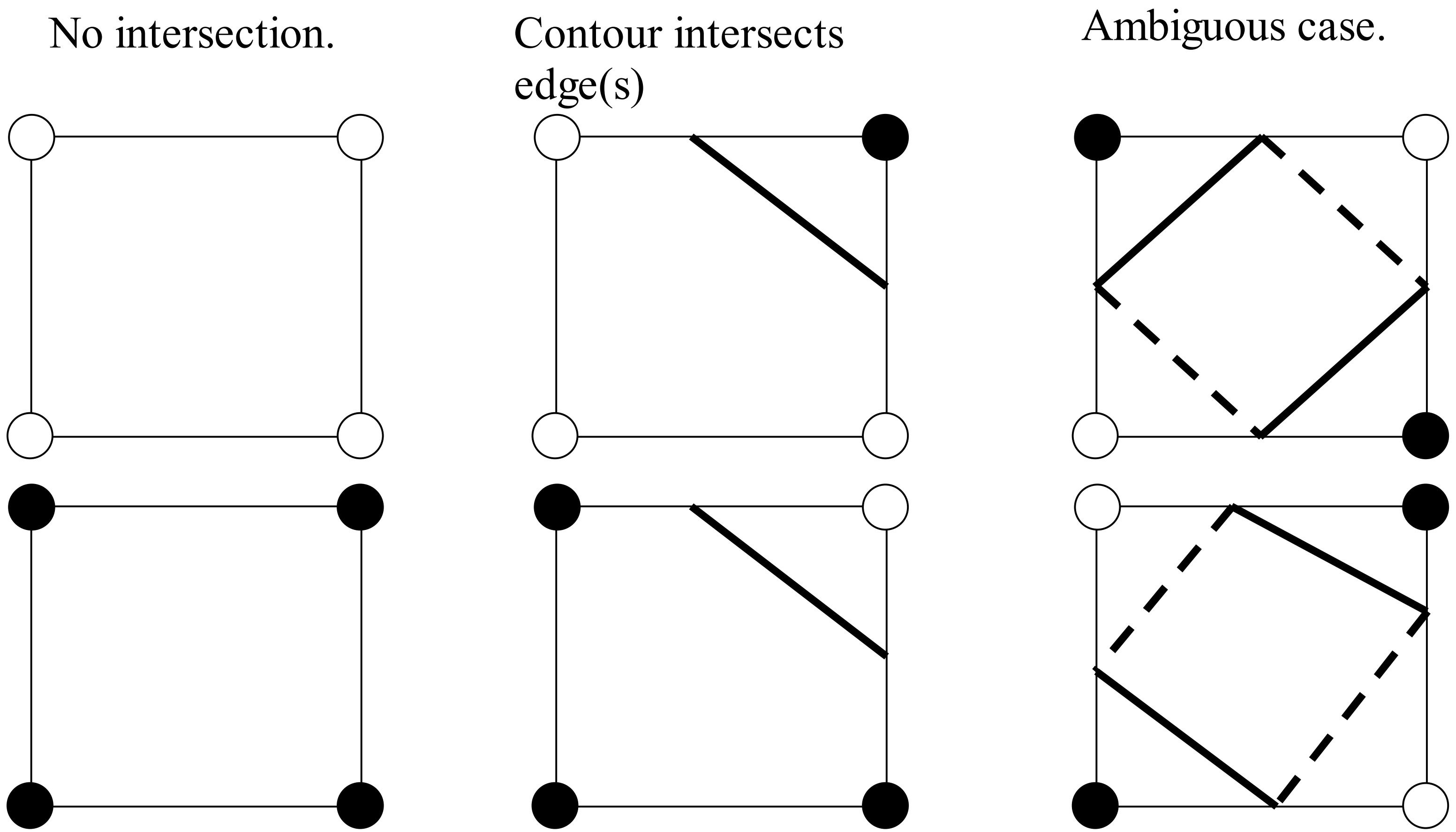
- **Approach 1 : Tracking**
  - find contour intersection with an edge
  - track it through the cell boundaries
    - if it enters a cell then it must exit via one of the boundaries
    - track until it connects back onto itself or exits dataset boundary
  - **Advantages** : produces correctly shaped line
  - **Dis-advantages :** need to search for other contours
- **Approach 2 : Marching Squares Algorithm**
  - works only on structured data
  - contour lines are straight between edges (approximation)

Marching Squares Algorithm

- **Focus :** intersection of contour and cell edges
  - **how the contour passes through the cell**
  - where it actually crosses the edge is easy to calculate

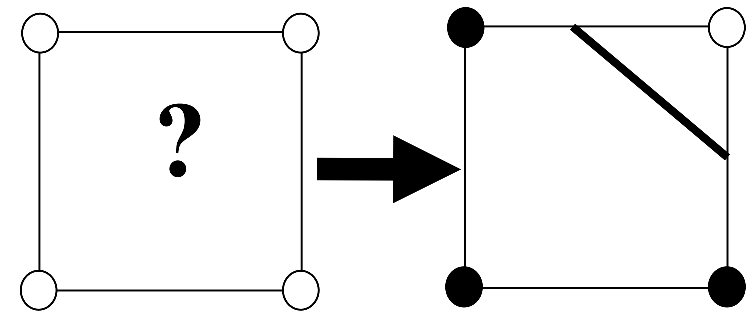
- **Assumption:** a contour can pass through a cell in only a finite number of ways
  - cell vertex is **inside contour if scalar value > contour**
    outside contour if scalar value < contour
  - **4 vertices, 2 states (in or out)**

Marching Squares



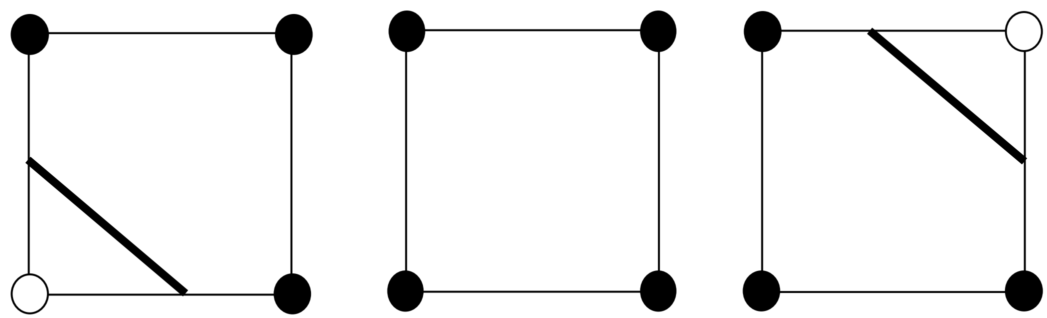
- $2^4 = 16$ possible cases for each square
  - small number so just treat each one separately

MS Algorithm Overview



- **Main algorithm**
  1. Select a cell
  2. Calculate inside/outside state for each vertex
  3. Look up topological state of cell in state table
     - determine which edge must be intersected (i.e. which of the 16 cases)
  4. Calculate contour location for each intersected edge
  5. Move (or **march**) onto next cell
     - until all cells are visited GOTO 2
- **Overall :** contour intersections for each cell

MS Algorithm - notes

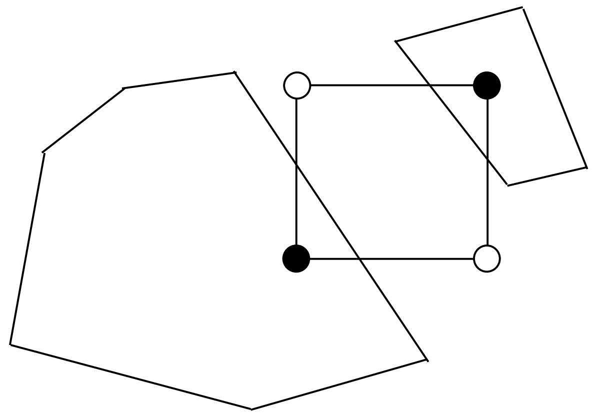


- Intersections for each cell must be merged to form complete contour
  - cells processed independently
  - further **“merging” computation required**
  - disadvantage over tracking (continuous tracked contour)

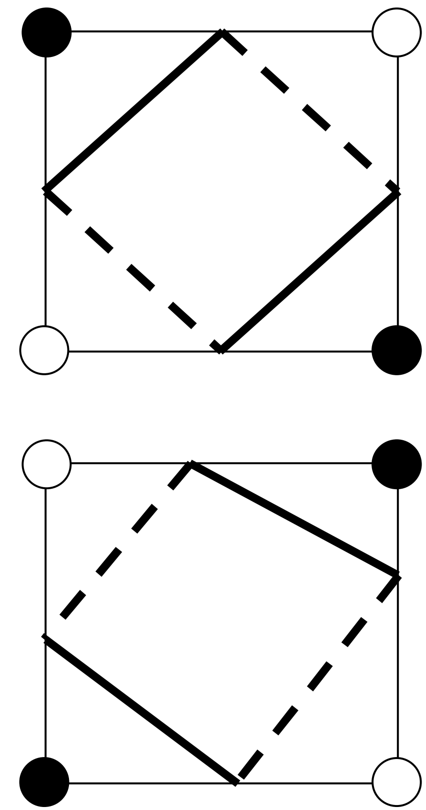
- easy to implement (also to extend to 3D)

MS : Dealing with ambiguity ?

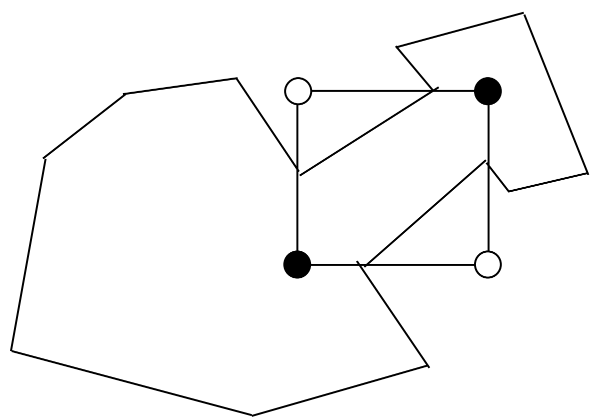
Spli
t



Ambiguous case.

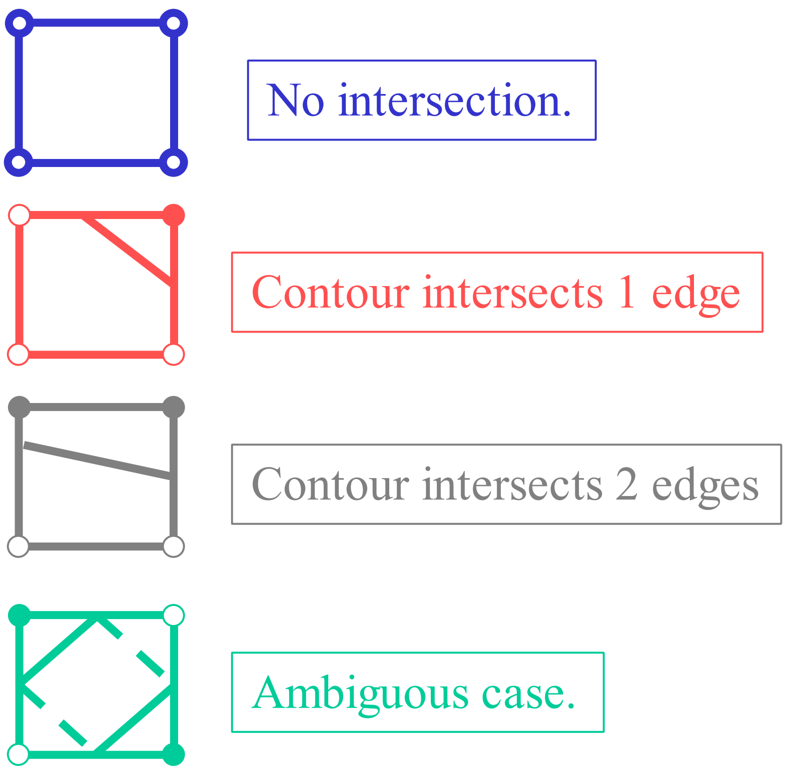


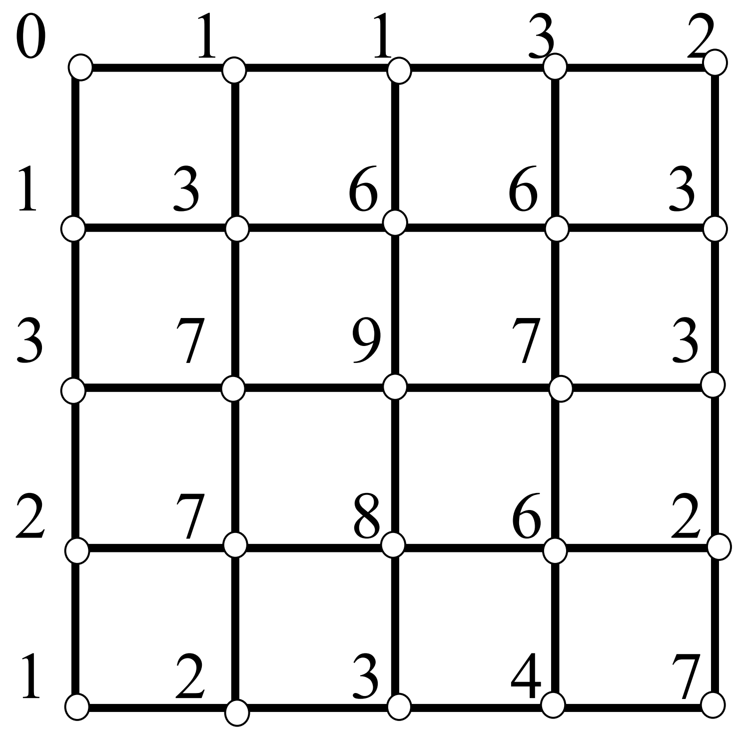
Join



- Choice independent of other choices
  - either valid : both give continuous and closed contour

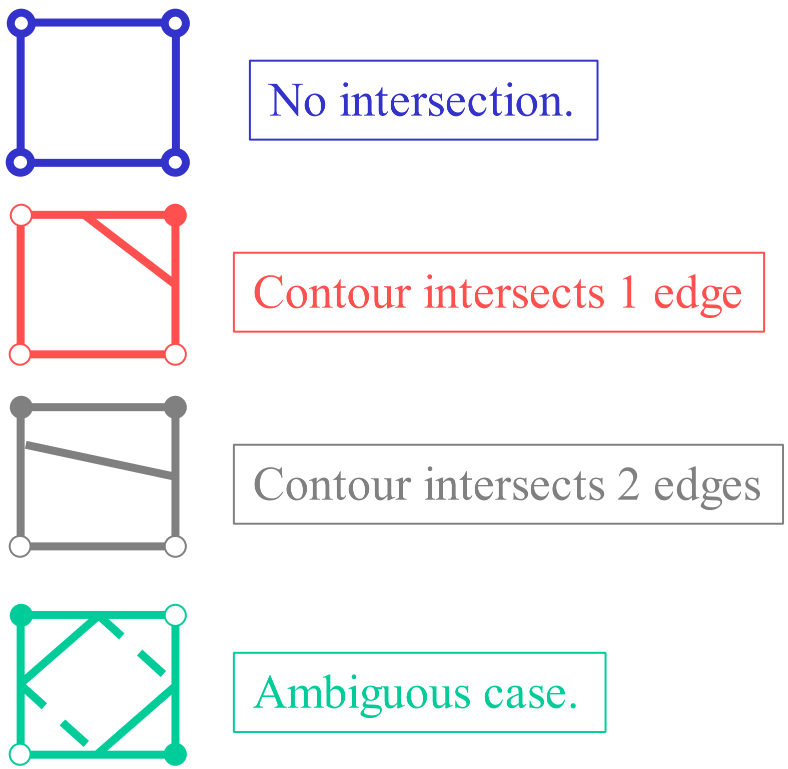
Example : Contour Line Generation

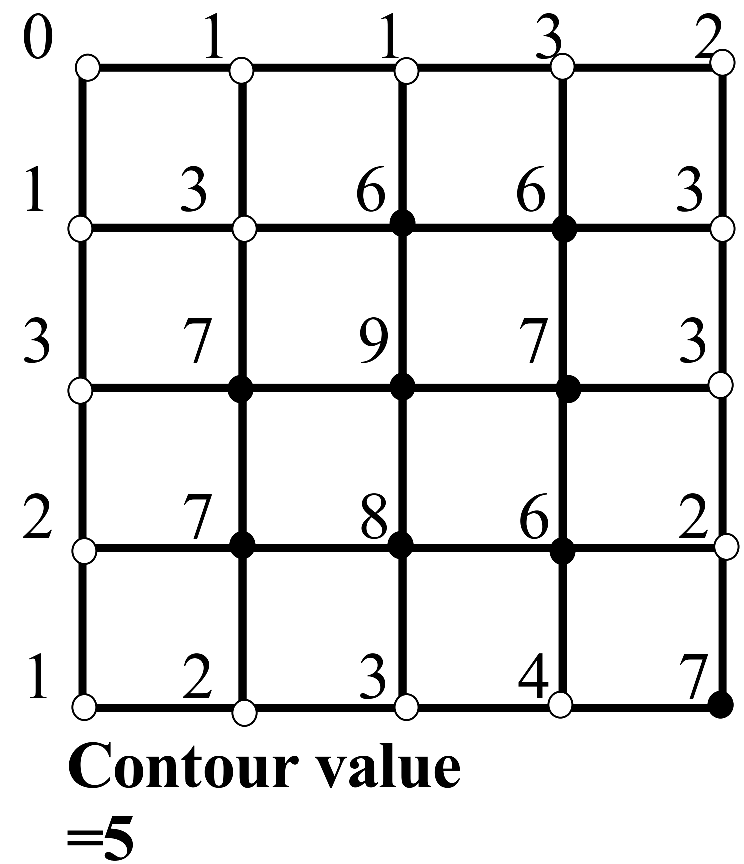




- 3 main steps for each cell
  - here using simplified summary model of cases

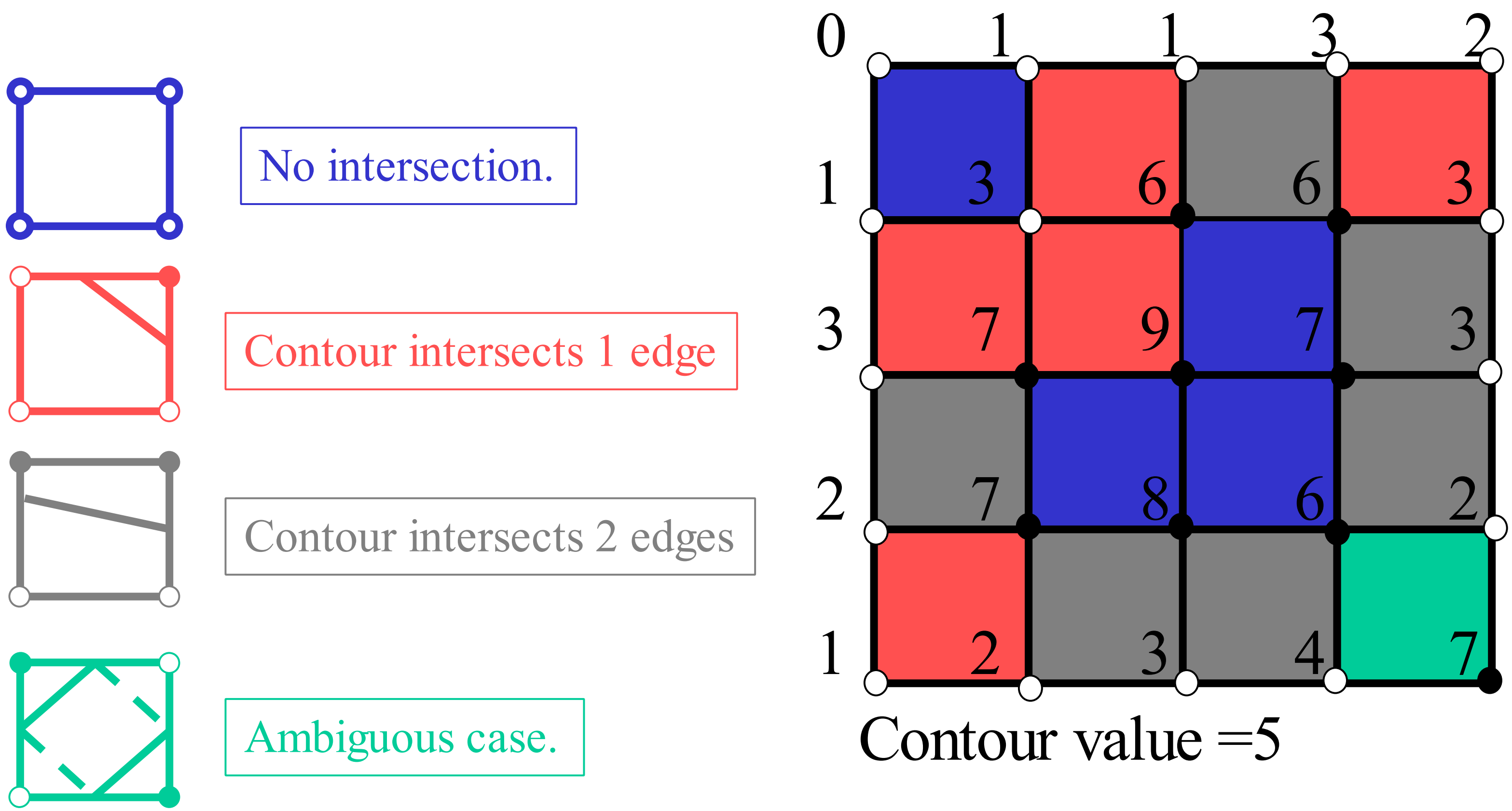
Step 1 : classify vertices





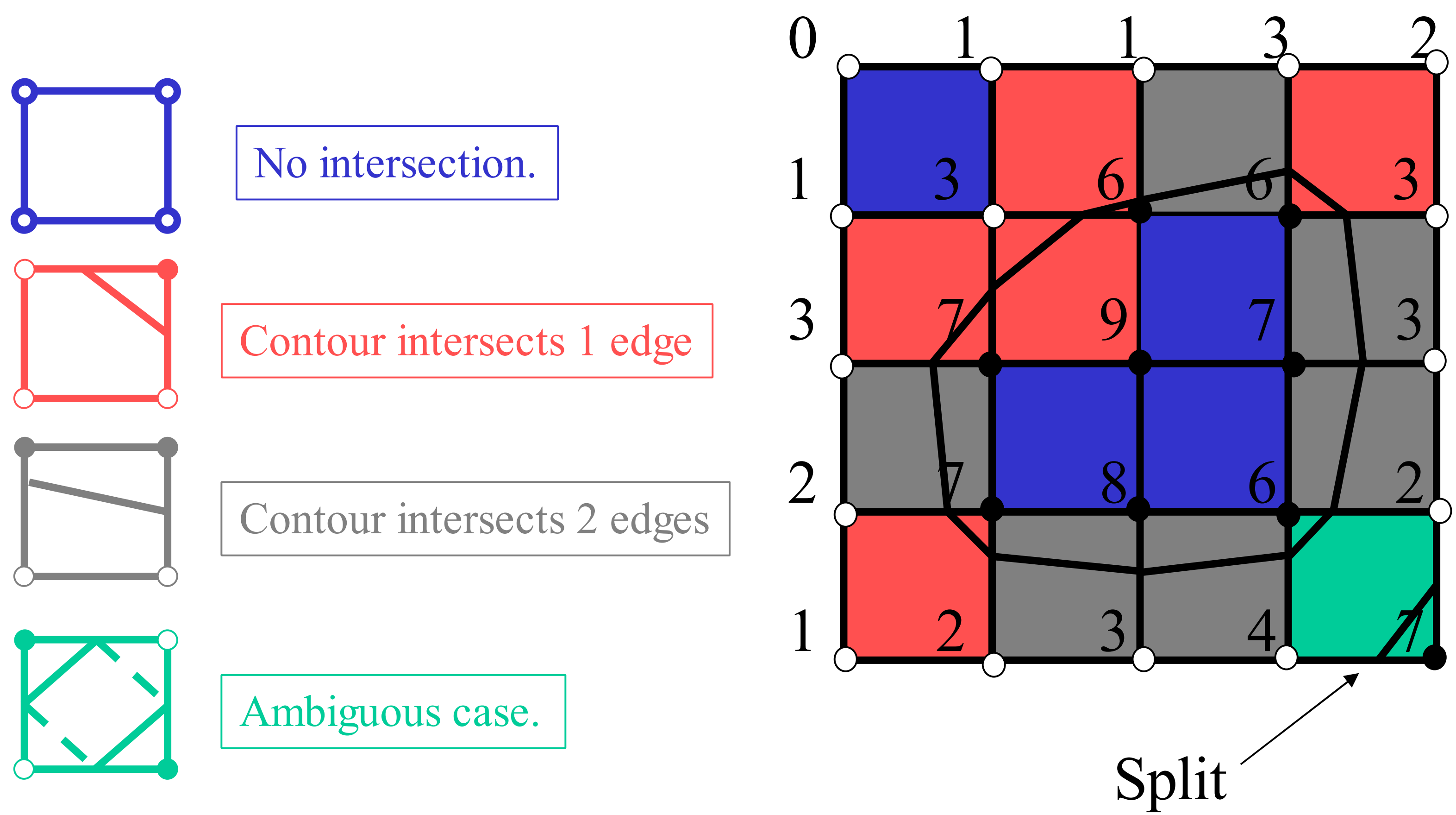
- Decide whether each vertex is inside or outside contour

Step 2 : identify cases



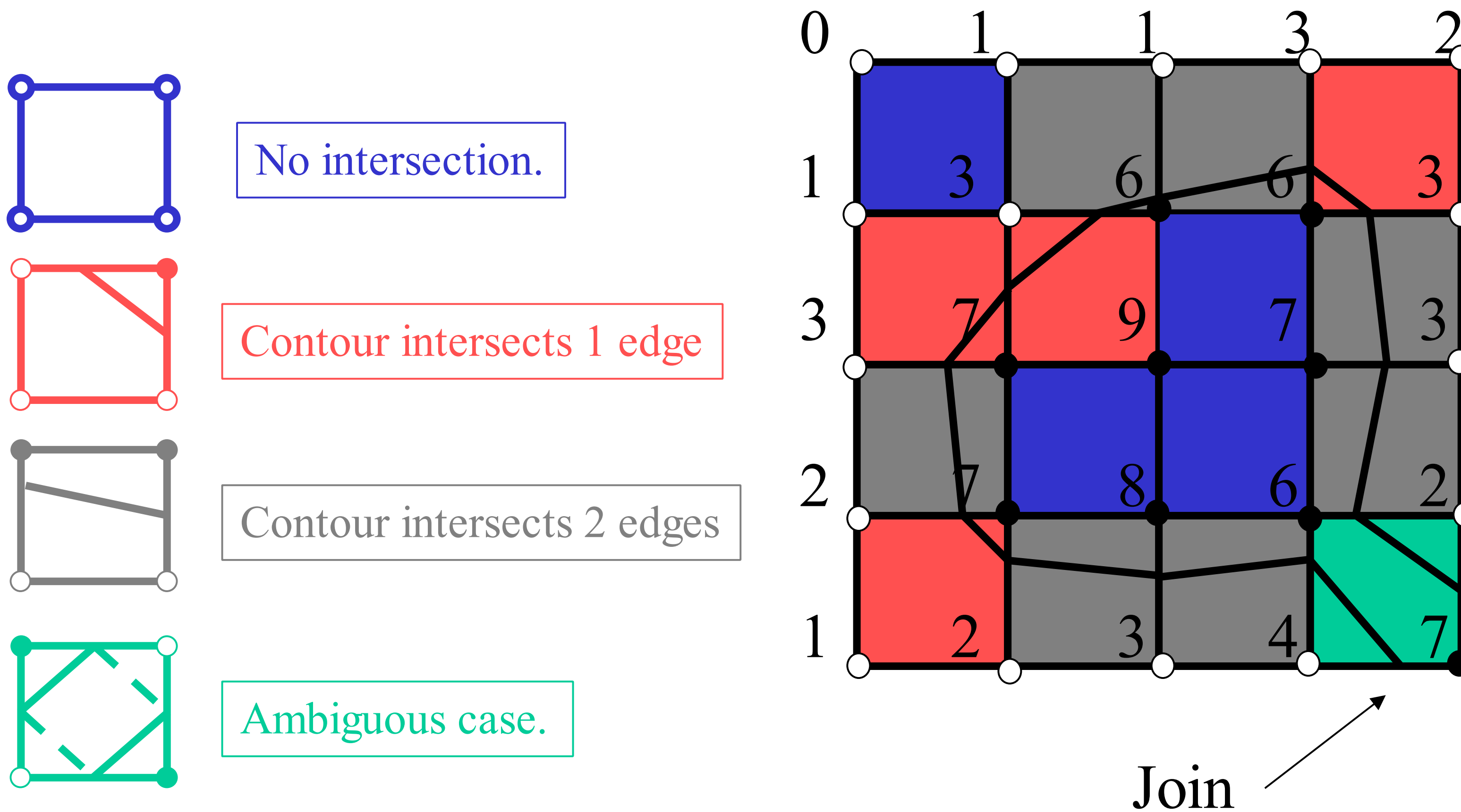
- Classify each cell as one of the cases

Step 3 : interpolate contour intersections



- Determine the edges that are intersected
  - compute contour intersection with each of these edges

Ambiguous contour



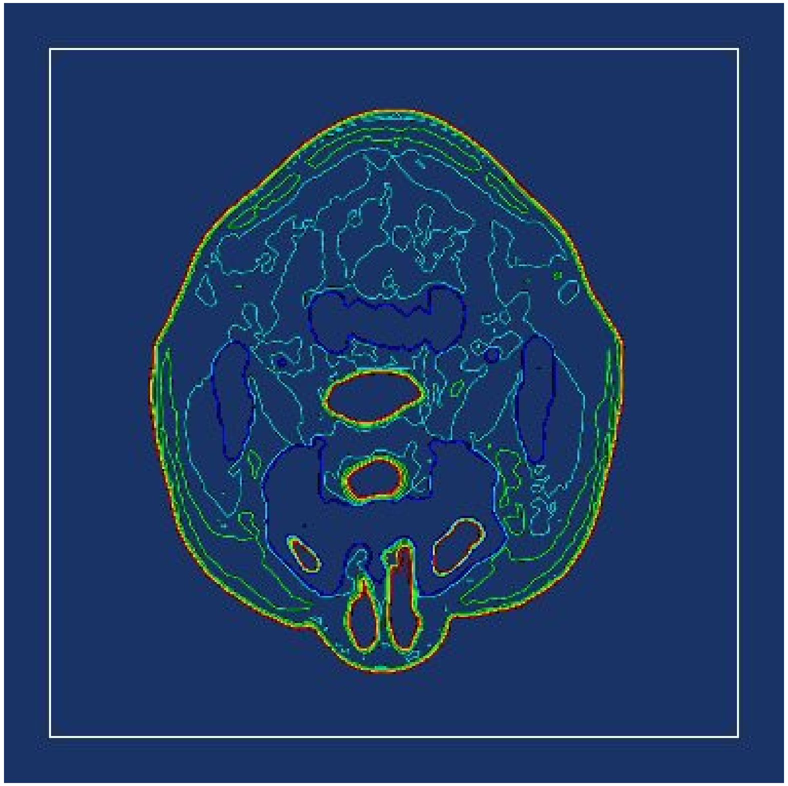
- Finally : resolve any ambiguity
  - here choosing “join” (example only)

Marching Squares Implementation

- Select a cell
  - Calculate inside/outside state for each vertex
  - Create an index by **storing binary state of each vertex in a separate bit**
  - Use **index to lookup topological state of cell in a case table**
  - Calculate contour location (**geometry**) for each edge via interpolation
  - Connect with straight line
- **March** to next cell (order/direction non-important)
- Need to merge co-located vertices into single polyline

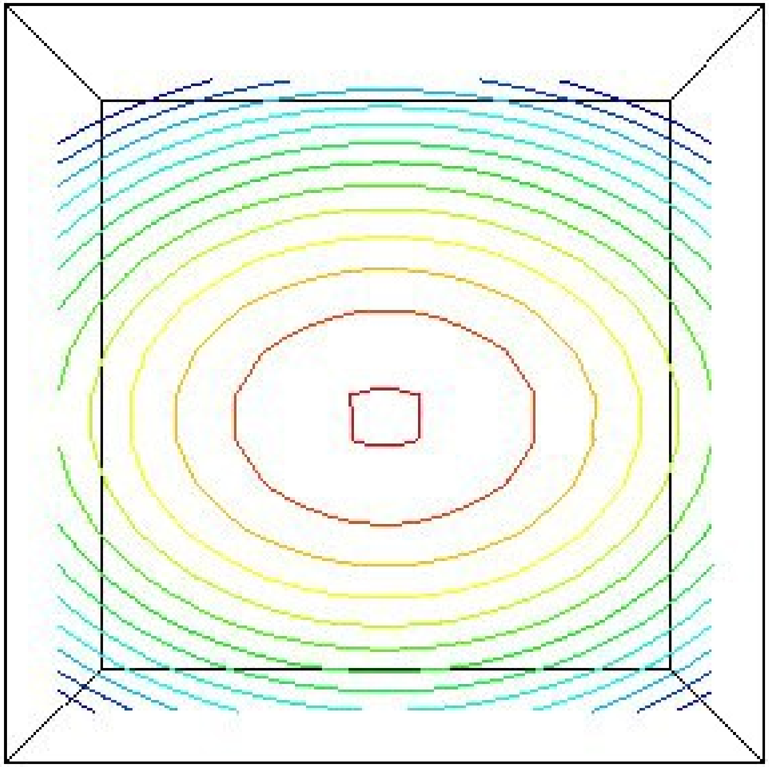
2D : Example contour

A slice through the head



(with colour mapping added)

A Quadric function.



3D surfaces : marching cubes

- Extension of Marching Squares to **3D**
  - **data** : 3D regular grid of scalar values
  - **result :** 3D surface boundary instead of 2D line boundary
  - 3D cube has 8 vertices → $2^8 = 256$ cases to consider
    - use symmetry to reduce to 15
- **Problem : ambiguous cases**
  - cannot simply choose arbitrarily as choice is determined by neighbours
  - poor choice may leave hole artefact in surface

Marching Cubes - cases

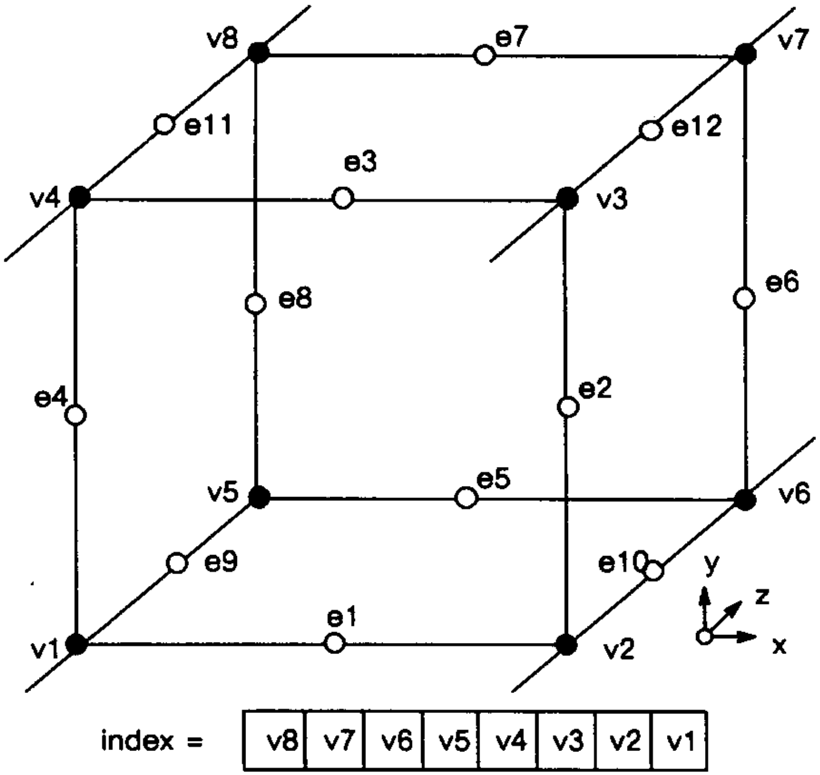


Figure 4. Cube Numbering.

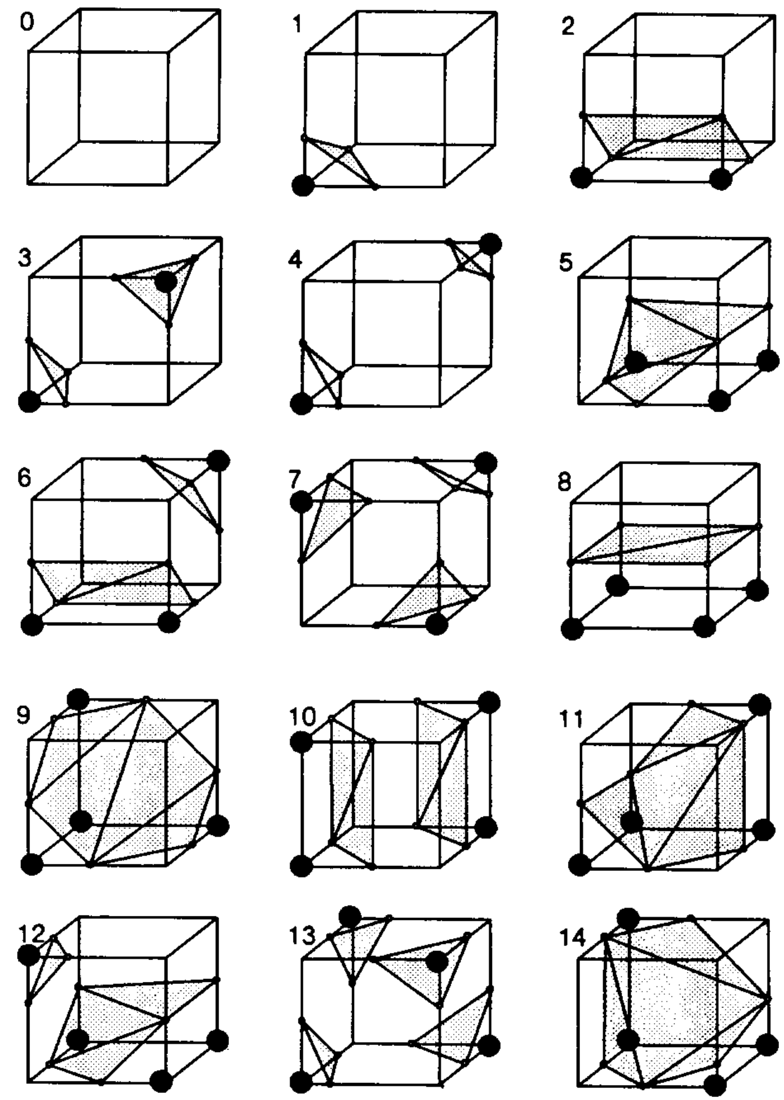
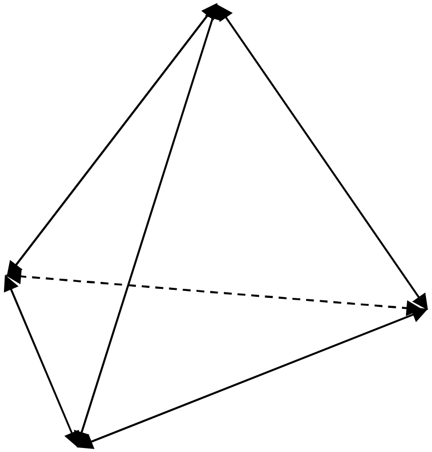


Figure 3. Triangulated Cubes.

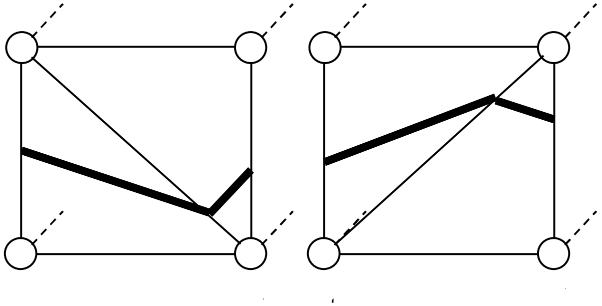
- Ambiguous cases
  - 3,6,10,12,13 – split or join ?

Solution to ambiguous cases



- **Marching Tetrahedra**
  - use tetrahedra instead of cubes
  - **no ambiguous cases**
  - but **more polygons** (triangles now)
  - need to choose which diagonal of cube to split to form tetrahedra
    - constrained by neighbours or bumps in surface

isosurface = 2.5

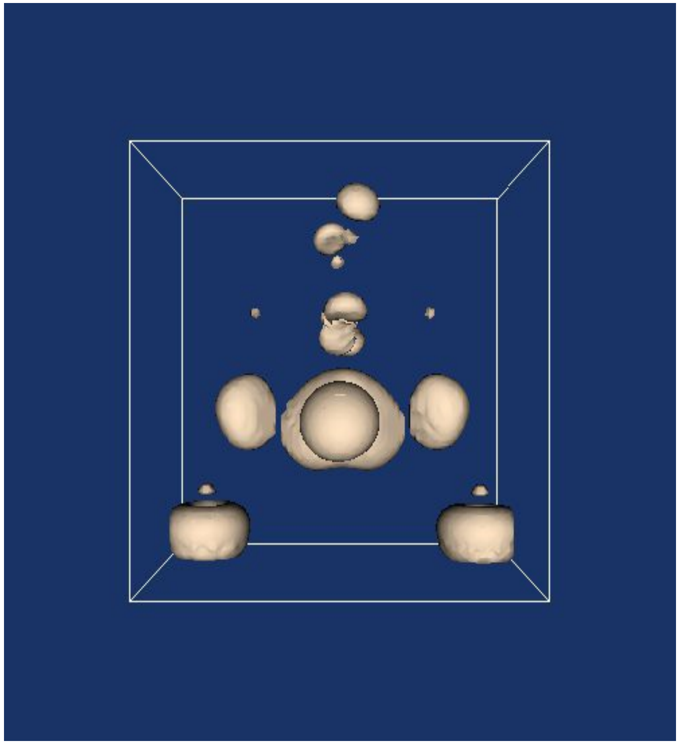


Alternative solutions

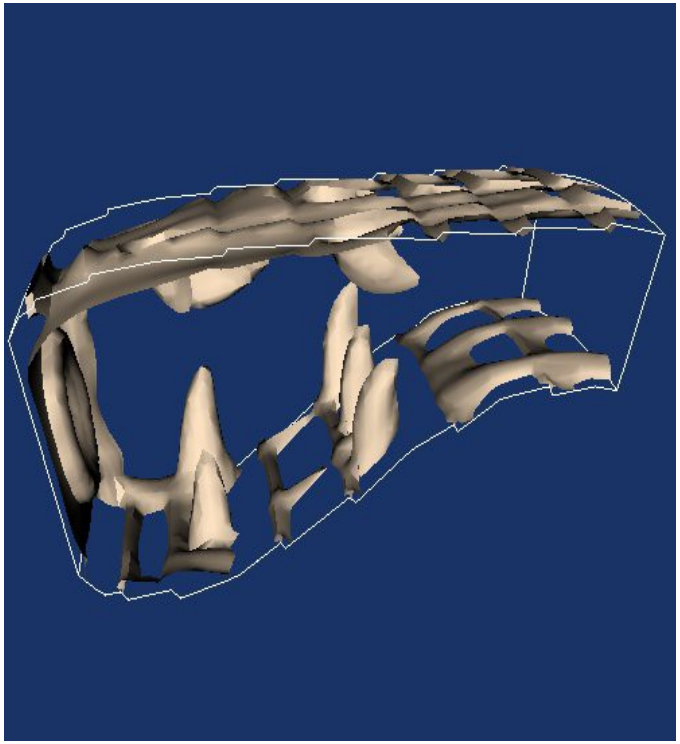
- **Analysis of neighbours** [Neilson '91]
  - decide whether to split or join
  - analysis of scalar variable across face

Results : isosurfaces examples

isosurface of Electron potential



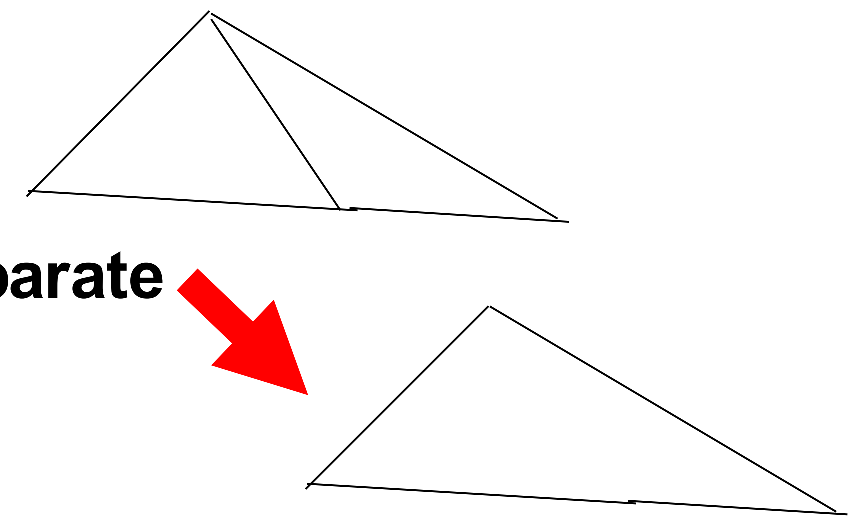
isosurface of flow density



- white outline shows bounds of **3D data grid**
- surface = **3D contour** (i.e. isosurface) **through grid**
- **method :** Marching Cubes

Problems with Marching Cubes

- Generates **lots of polygons**
  - 1-4 triangles per cell intersected
  - many unnecessary
    - e.g. co-planar triangles
    - lots of work extra for rendering!
  - As with marching squares **separate merging required**
    - need to perform explicit search



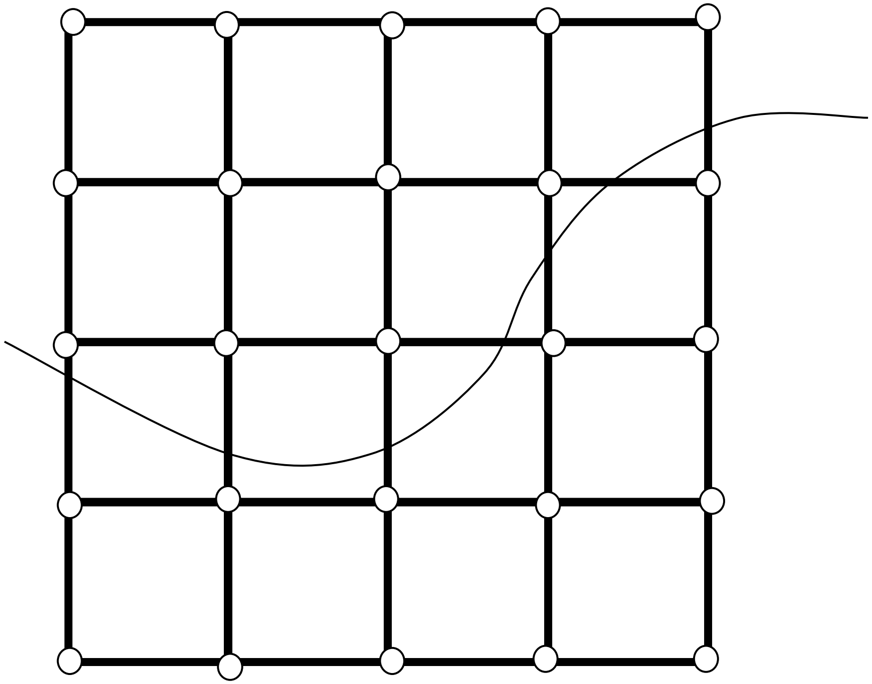
Dividing Cubes Algorithm

- **Marching cubes**
  - often **produces more polygons than pixels** for given rendering scale
  - **Problem :** causes **high rendering overhead**

- **Solution :** Dividing Cubes Algorithm
  - draw **points instead of polygons** *(faster rendering)*
  - Need 1: efficient **method to find points on surface**

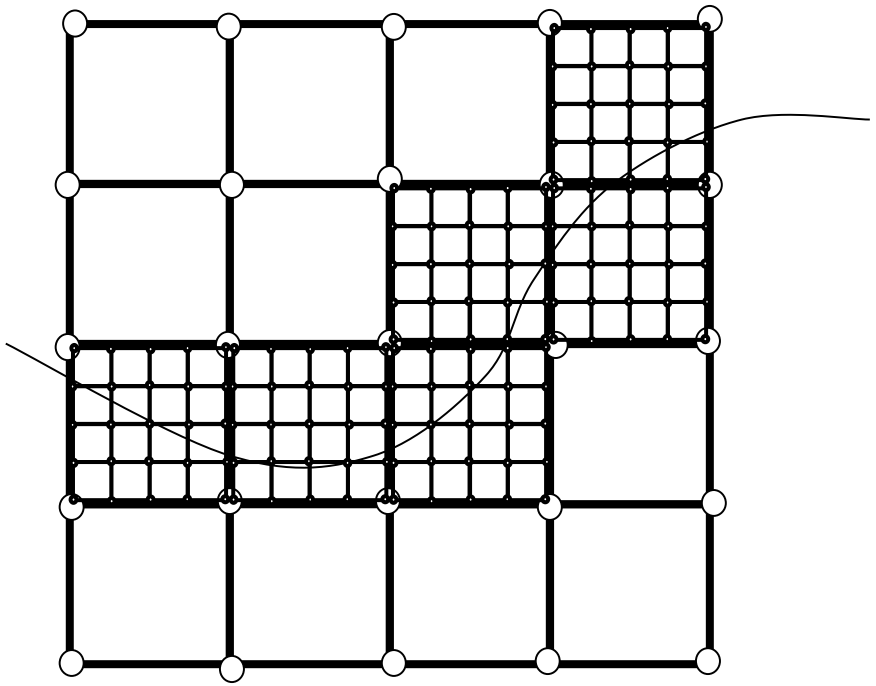
    2: method to **shade points**

Example : 2D divided squares for 2D lines



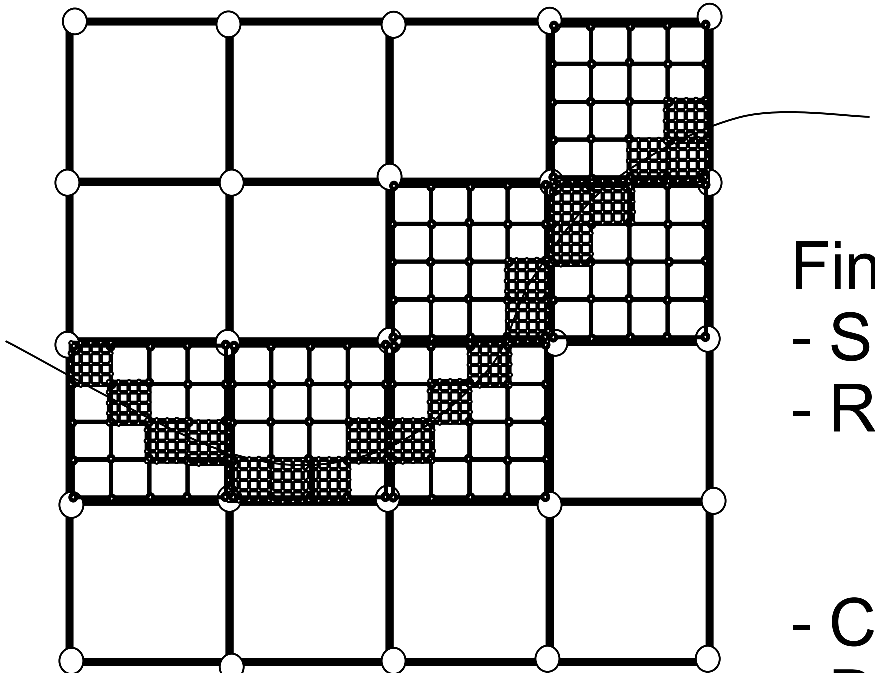
Find pixels that intersect contour
- Subdivide them

2D “Divided Cubes” for lines



Find pixels that intersect line
- **Subdivide them** (usually in 2x2)
- **Repeat recursively**

2D "Divided cubes" for lines

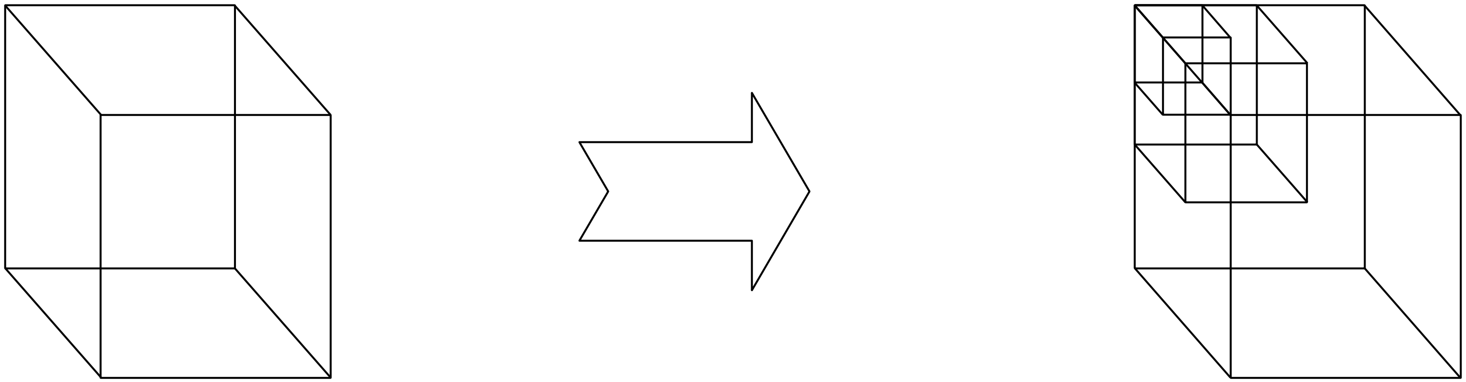


Find pixels that intersect line
- Subdivide them
- Repeat recursively
  until screen resolution reached

- Calculate mid-points
- Draw line

Extension to 3D

- Find **voxels** which intersect **surface**
- Recursively subdivide
  - When to stop?
- Calculate **mid-points of voxels**
- Project **points and draw pixels**



Drawing divided cubes surfaces

- **surface normal** for lighting calculations
  - interpolate **from voxel corner points**
- **problem with camera zoom**
  - ideally dynamically re-calculate points
  - not always computationally possible
- **smooth looking surface**
  - represented by **rendered point cloud**

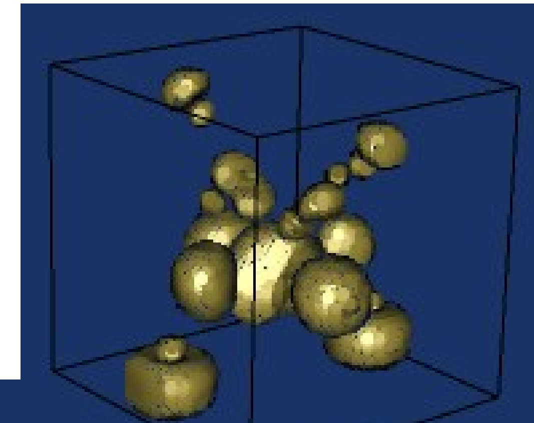
Dividing Cubes : Example

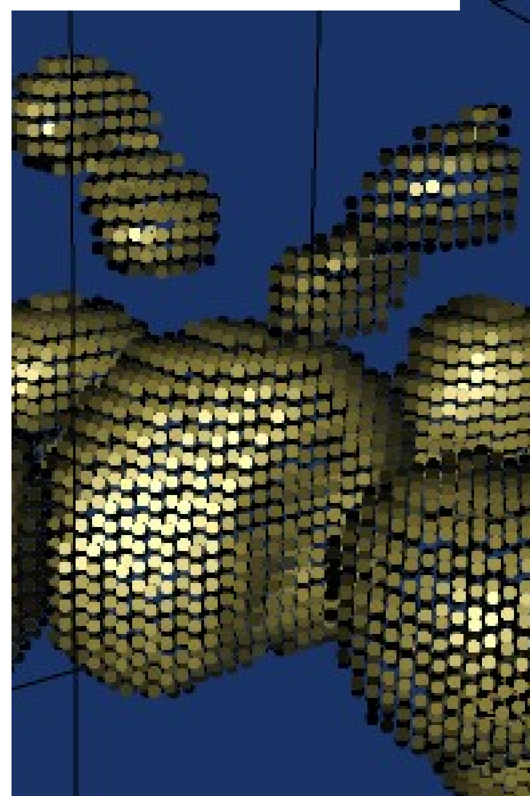
50,000 points

when sampling less than screen resolution structure of surface can be seen

Problem : algorithm is patented

see : `dcubes.tcl`





Contouring available in VTK

- Single object : `vtkContourFilter`
  - can accept any dataset type (cf. pipeline multiplicity)
  - input **tetrahedra cells**
    - *Marching tetrahedra* used
  - input **structured points**
    - *Marching cubes* used
  - input **triangles**
    - *marching triangles* used *[Hilton '97]*
- Also `vtkMarchingCubes` object
  - only accepts structured points, slightly faster
  - problems with patent issues (MC **is** patented)

Summary

- **Contouring Theory**
  - 2D : **Marching Squares Algorithm**
  - 3D : **Marching Cubes Algorithm** [Lorensen '87]
    - marching tetrahedra, ambiguity resolution
    - limited to regular structured grids
  - 3D Rendering : **Dividing Cubes Algorithm** [Cline '88]

- **Contouring Practice**
  - examples and objects in **VTK**

Next lecture : *Advanced Data Representation*