Volume Illumination, Contouring

Computer Animation and Visualisation – Lecture 12

Taku Komura

Institute for Perception, Action & Behaviour
School of Informatics

Overview

- Volume Illumination

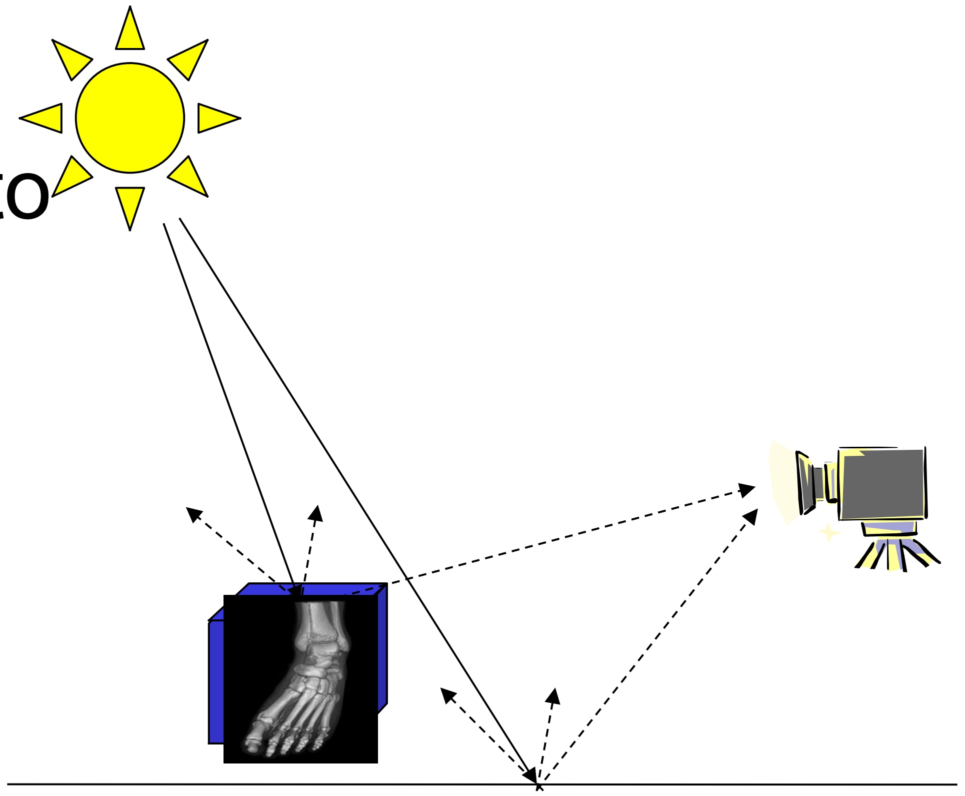
- Contouring
    - Problem statement
    - Tracking
    - Marching squares
    - Ambiguity problems
    - Marching cubes
    - Dividing squares

Light Propagation in Volumes

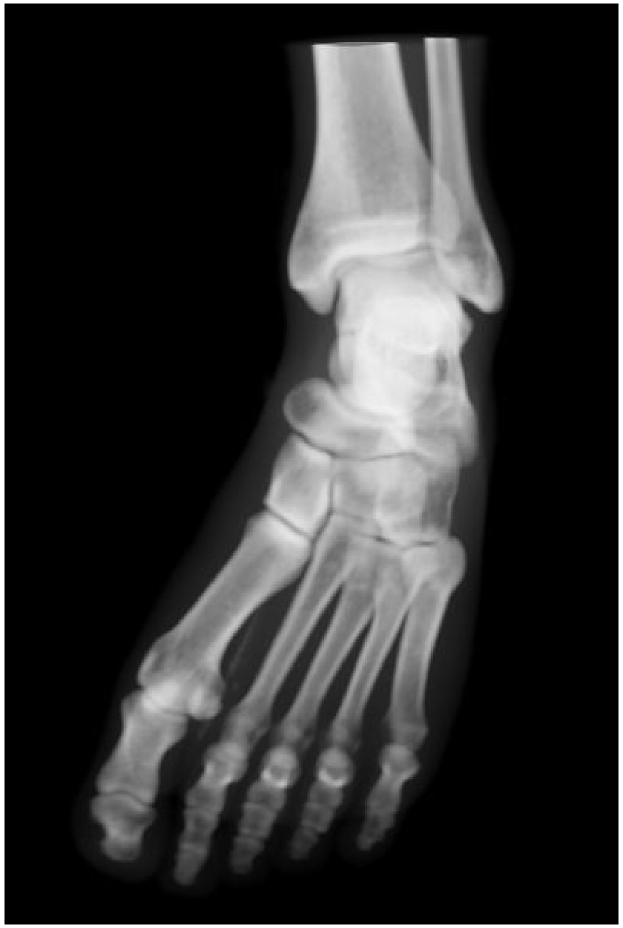
- **Lighting in volume**
  - Beyond light **transmission and emission**, objects can also:
    - **reflect light**
    - **scatter light** into different directions

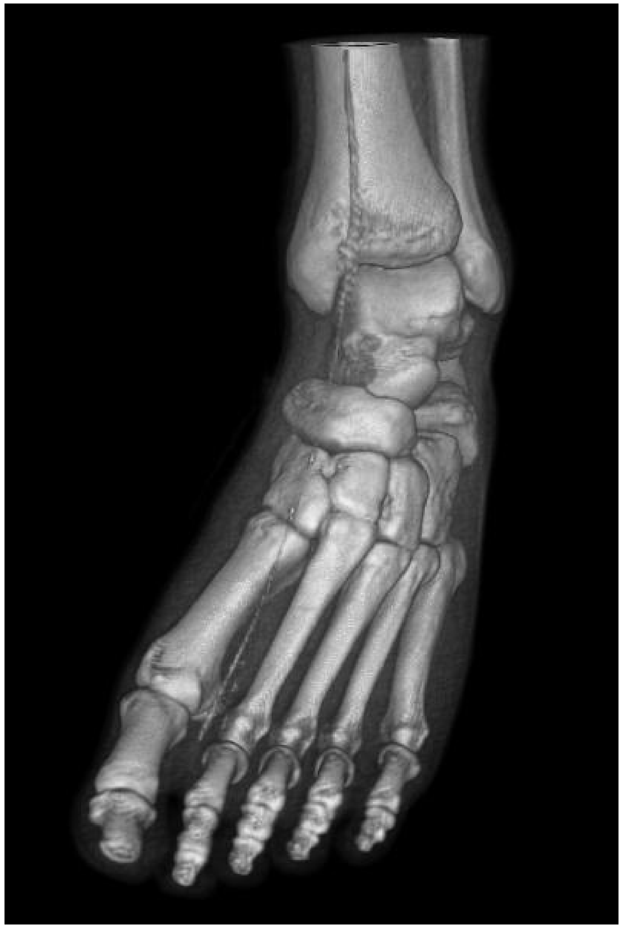
Volume Illumination

- Why do we want to illuminate volumes?
- **illumination helps us to better understand 3D structure**
  - displays visual cues to **surface orientation**
  - highlight significant **gradients within volume**



Result : illuminated iso-surface

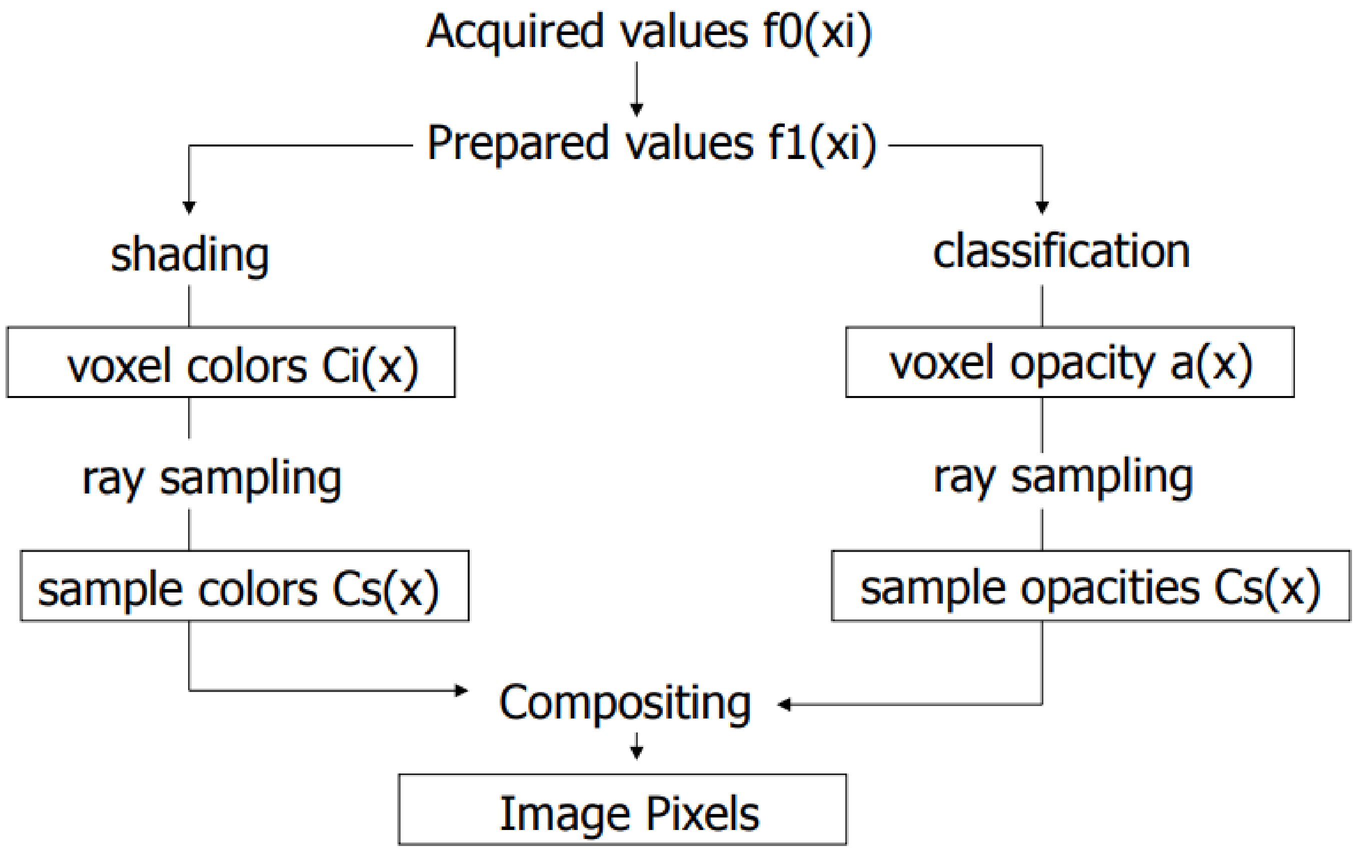

MIP technique


Shaded embedded iso-surface.

- Surface normals recovered from depth map of surface

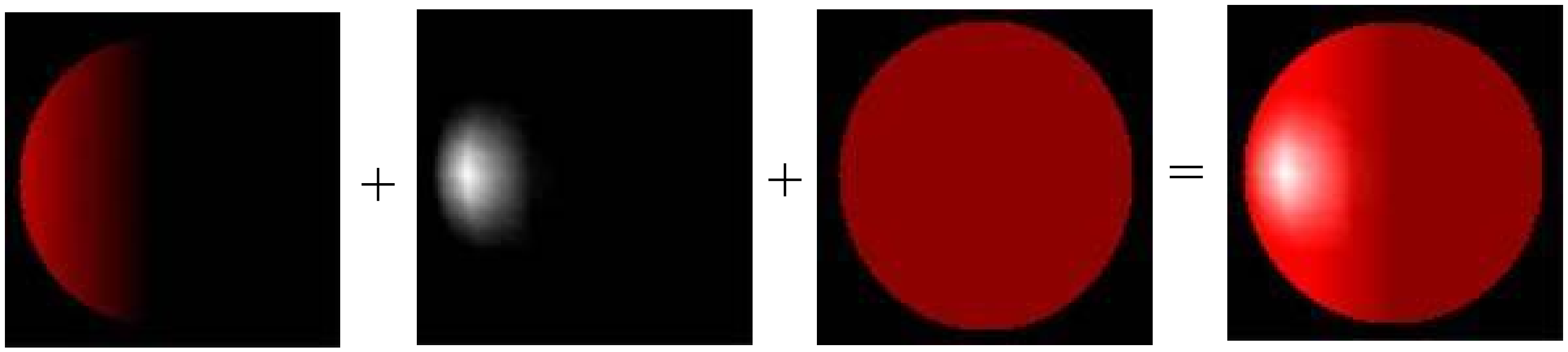
Volume Illumination: Pipeline

- Compute the colour transfer function / Phong illumination
- Compute the opacity using the opacity transfer function and gradient of the voxel values
- Combine the two using the composite method



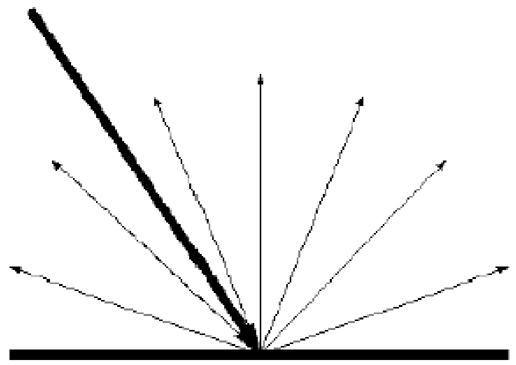
Phong Illumination Model

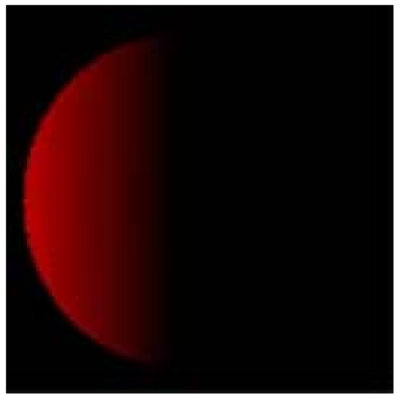
- Simple 3 parameter model
  - The sum of 3 illumination terms:
    - **Diffuse :** non-shiny illumination and shadows
    - **Specular :** bright, shiny reflections
    - **Ambient :** 'background' illumination



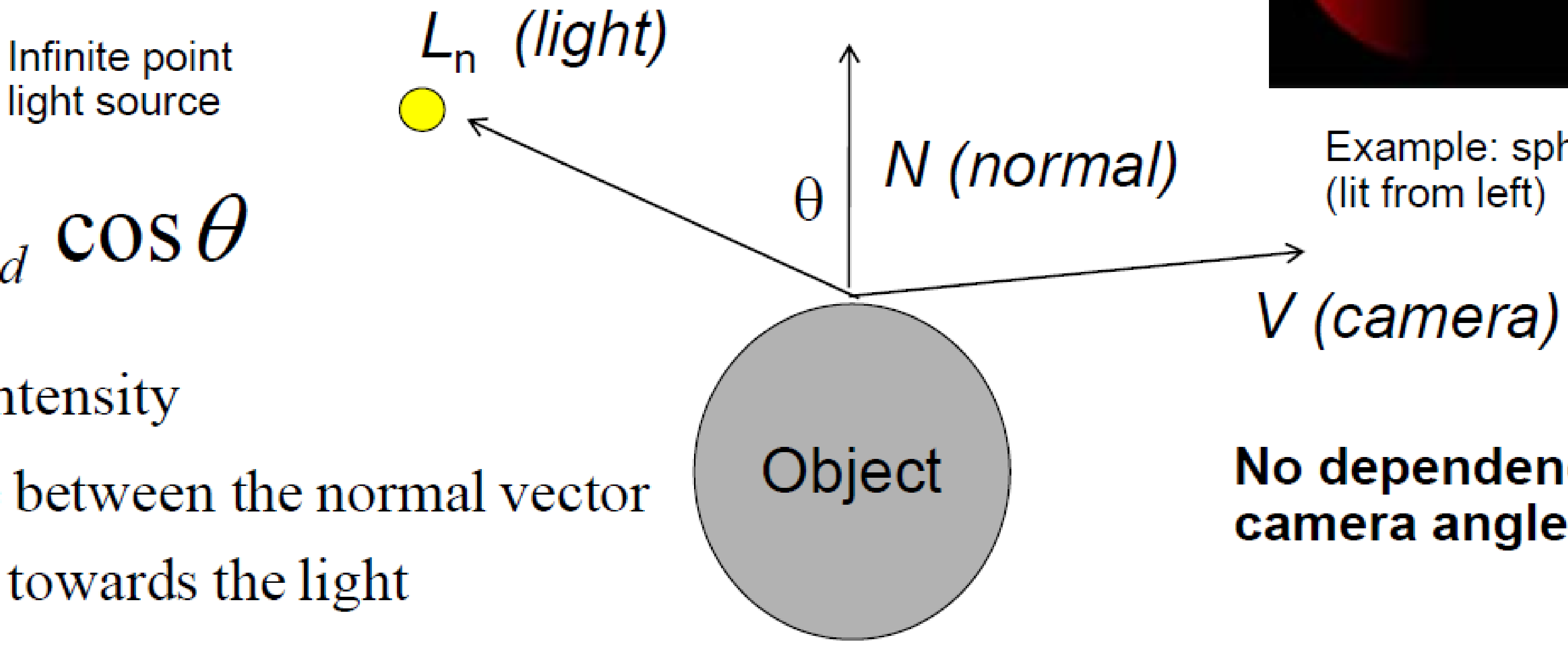
Diffuse (directional) + Specular (highlights) + Ambient (color) = Rc

Diffuse Reflection





Example: sphere (lit from left)



$$I = I_p k_d \cos\theta$$

I_p : Light Intensity

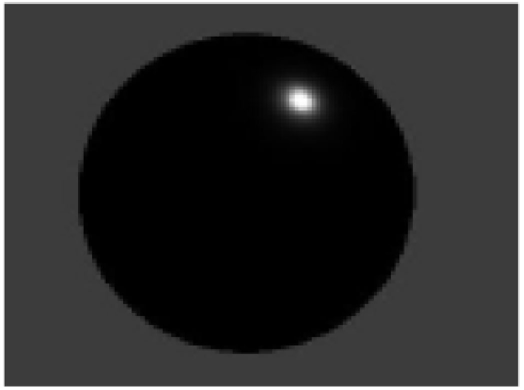
θ : the angle between the normal vector direction towards the light

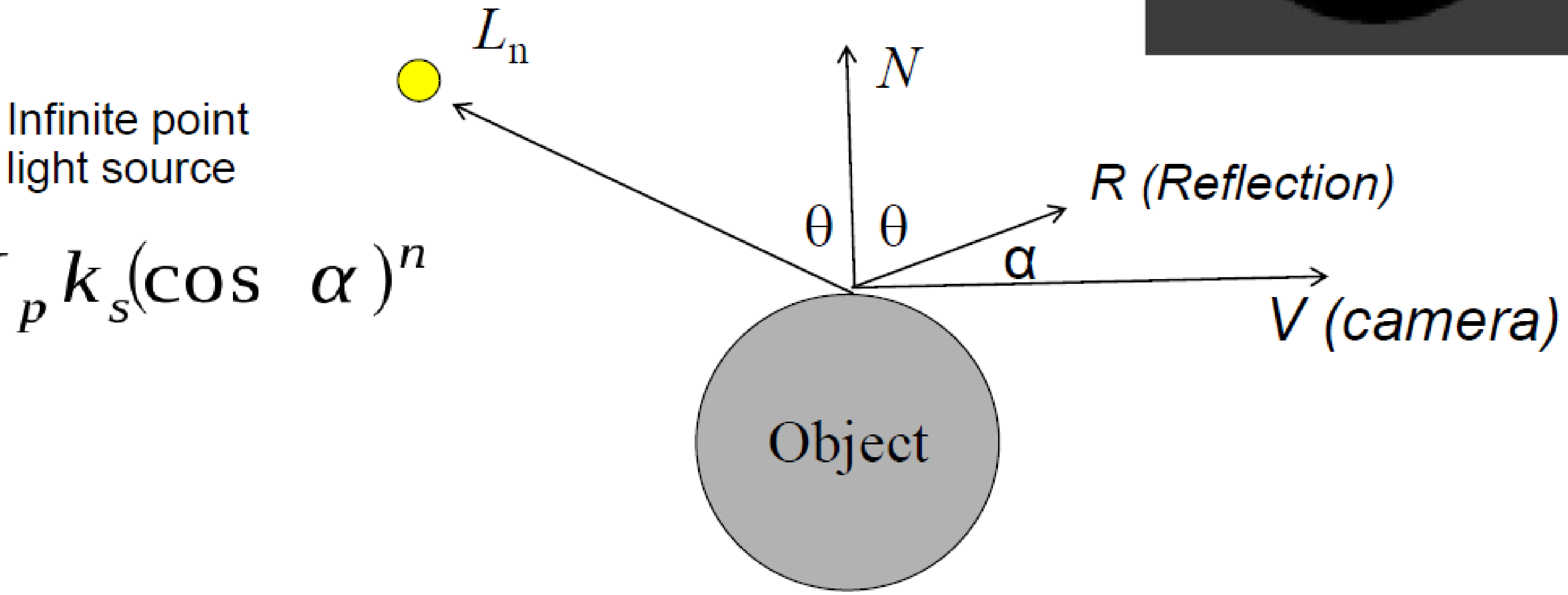
k_d : diffuse reflectivity

No dependence on camera angle!

Specular Reflection

- Direct reflections of light source off shiny object
  - –specular intensity n = shiny reflectance of object
  - –Result: **specular highlight on object**





$$I = I_p k_s (\cos \alpha)^n$$

No dependence on object color.

Ambient Lighting

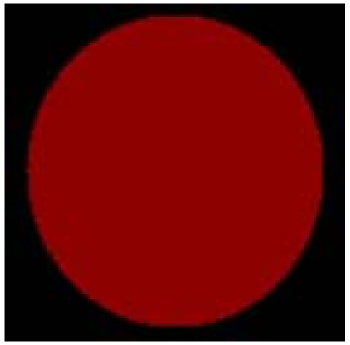
Simple approximation to complex 'real-world' process

Result: **globally uniform color for object**

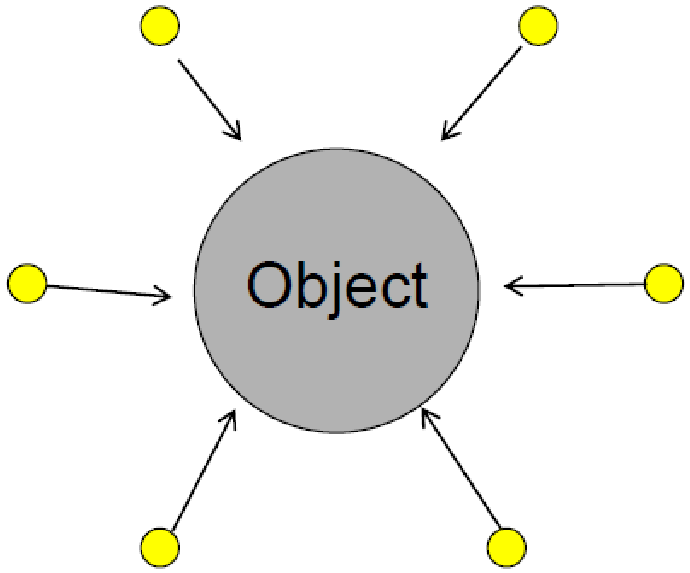
I = resulting intensity

I_a = light intensity

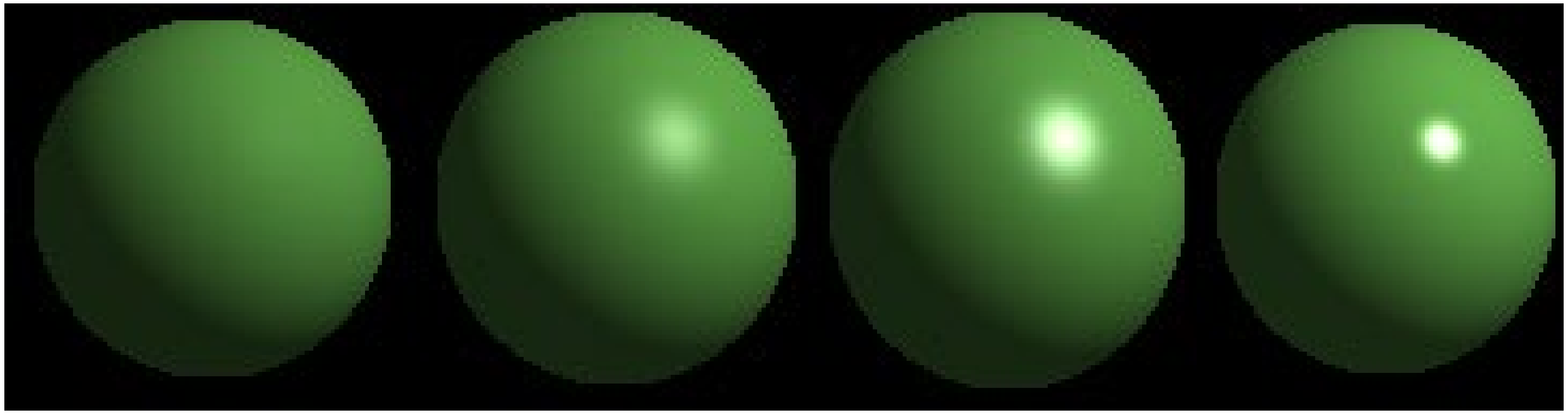
k_a = reflectance


Example: sphere

$$\boxed{I = k_a I_a}$$



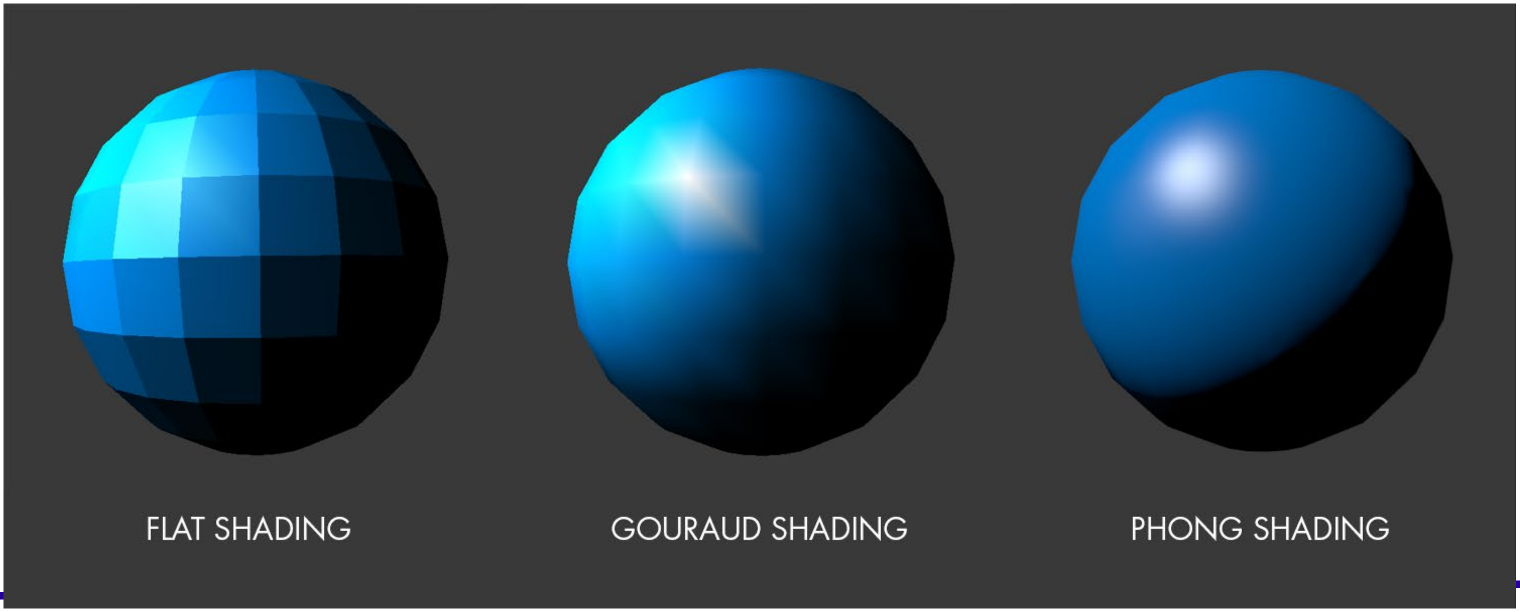
Combining Diffuse and Specular Reflections



Shading Models

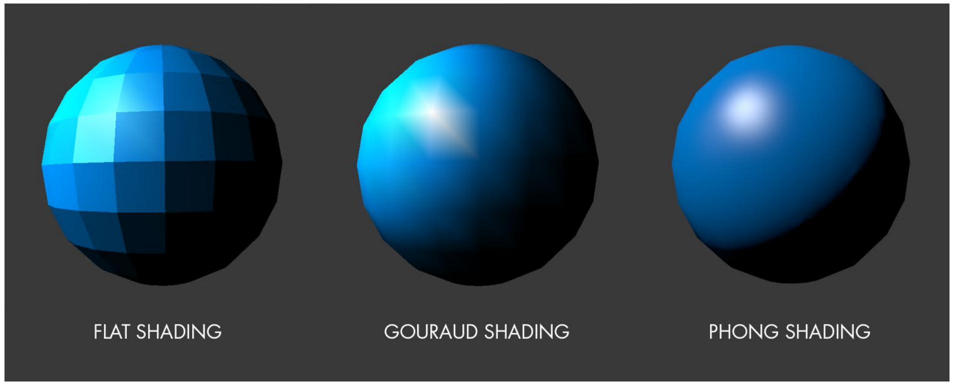
How we compute the colour of a mesh

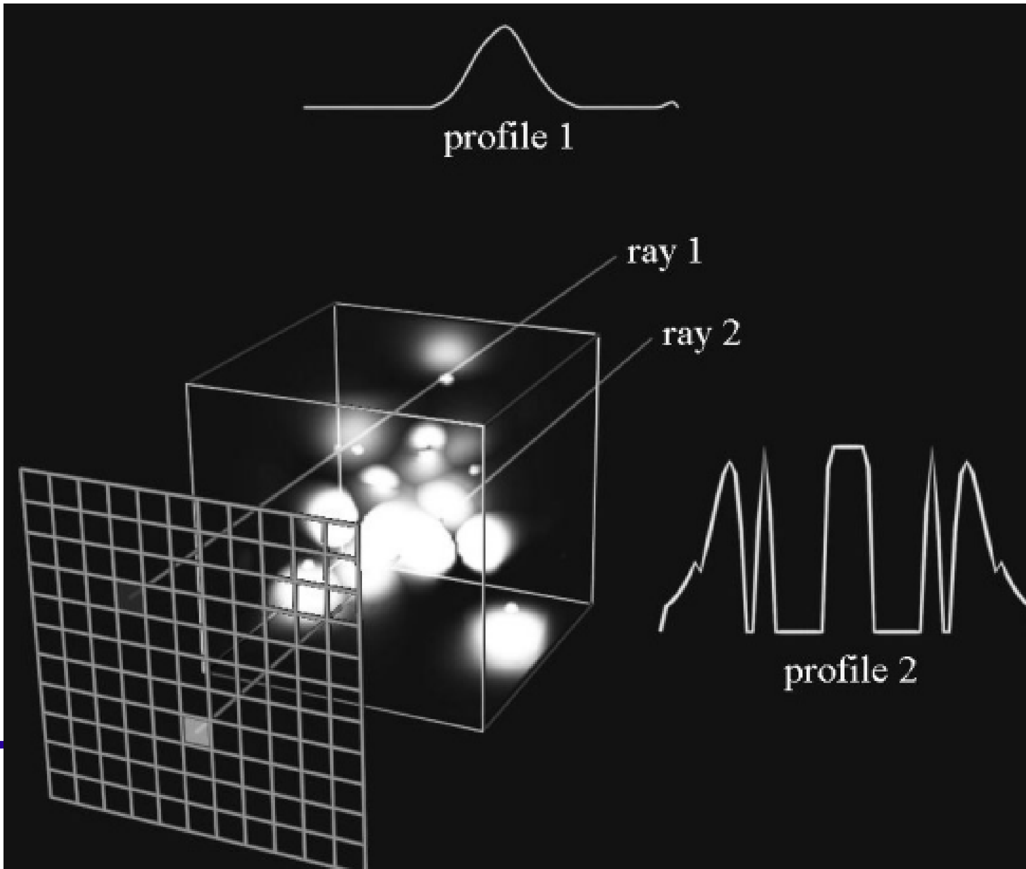
- **Flat**: Apply same normal and color to the whole polygon.
- **Gouraud**: Apply same normal to whole polygon, but interpolate the color over the vertices (Vertex Shader).
- **Phong**: Interpolate normal and color across surface of the polygon (Fragment Shader).



Shading Volume

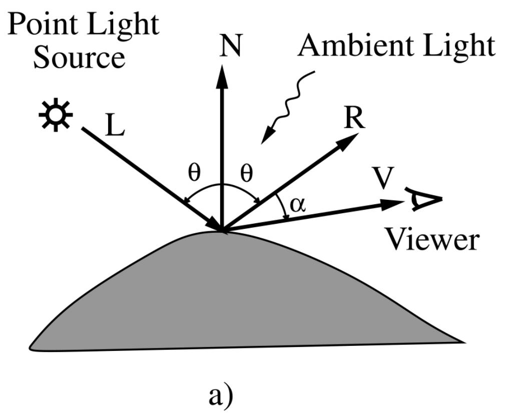
- As we are using an image-order volume rendering, we should go with Phong shading
- Computing the colour at each sample in ray by Phong illumination

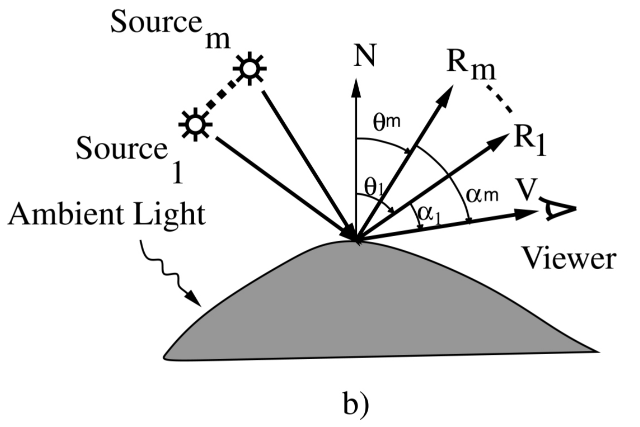




Shading an Embedded iso-surface

- Classify volume with the transfer function
- For calculating the colour of every voxel, use regular specular/diffuse surface shading
- **Remember** lighting requires illumination direction
  - camera model (position)
  - surface orientation
  - **need** to calculate and store **surface normal**





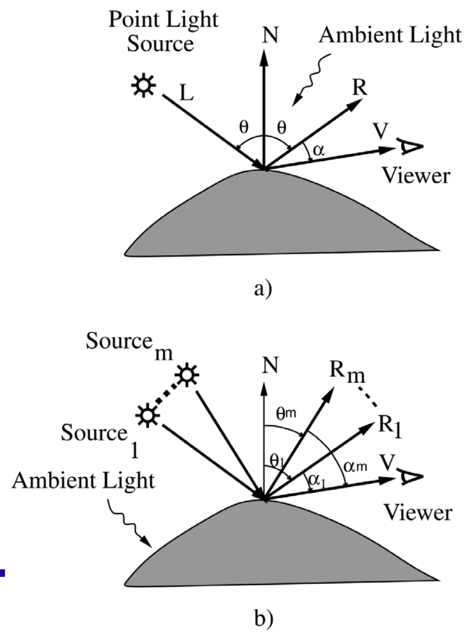
Estimating the Normal Vector

We can use the gradient of the voxel intensity values as the normal vector

$$\nabla I = (I_x, I_y, I_z) = \left(\frac{\delta}{\delta x} I, \frac{\delta}{\delta y} I, \frac{\delta}{\delta z} I\right)$$

$$\frac{\delta}{\delta x} I = \frac{I(x+1,y,z) - I(x-1,y,z)}{2}$$

$$N = \nabla I / \|\nabla I\|$$



Computing the Opacity

- We first use the opacity values computed by the lookup table/opacity transfer function
  - Based on what we wish to see (setting them low for things we do not want to see, and high for those we wish to see)

- Then scale using the gradient vector

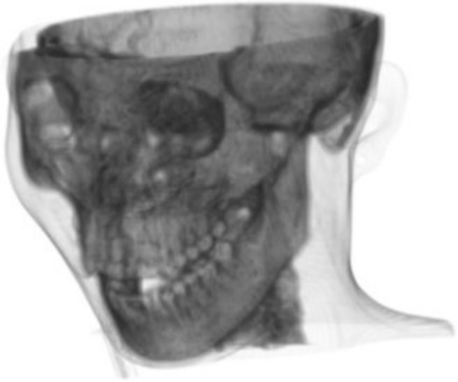
$$\alpha = \|\nabla I\| \alpha_v$$

α : the final opacity

$\|\nabla f(x)\|$: gradient of the volume data

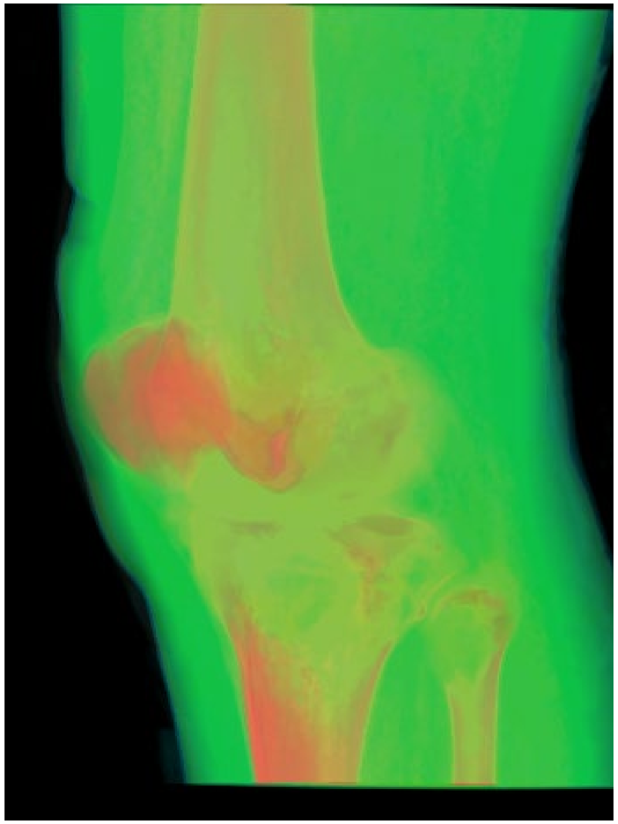
α_v: opacity by the transfer function

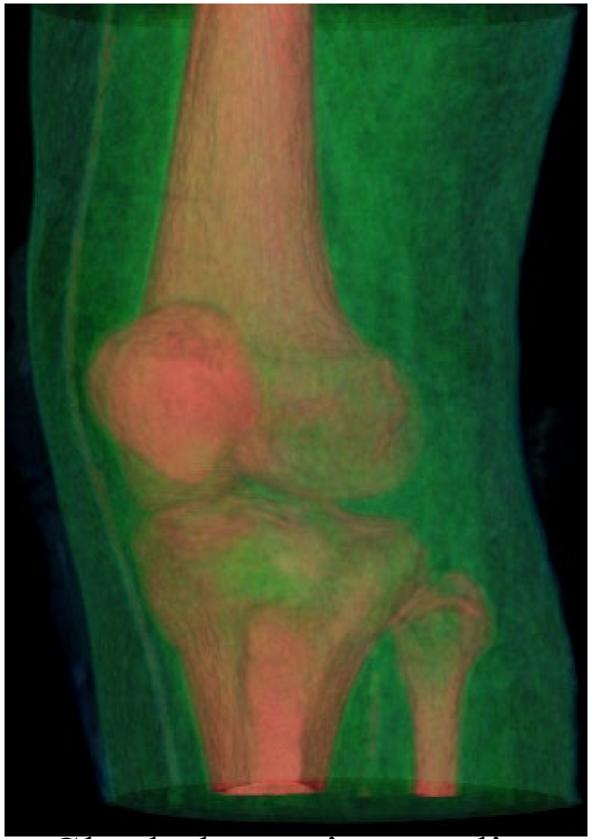
- This emphasizes the boundary of tissues



Illuminating Opacity (Scalar) Gradient

- **Illuminate “scalar gradient”** instead of iso-surface
  - requirement : estimate and store gradient at every voxel


Composite


Shaded opacity gradient
(shades changes in opacity)

Illumination : storing normal vectors

- Visualisation is **interactive**
  - **compute normal vectors for surface/gradient once**
  - **store normal**
  - **perform interactive shading calculations**
- **Storage :**
  - 256^3 data set of 1-byte scalars ~16Mb
  - normal vector (stored as floating point(4-byte)) ~ 200Mb!
  - **Solution** : quantise direction & magnitude as small number of bits

Illumination : storing normal vectors

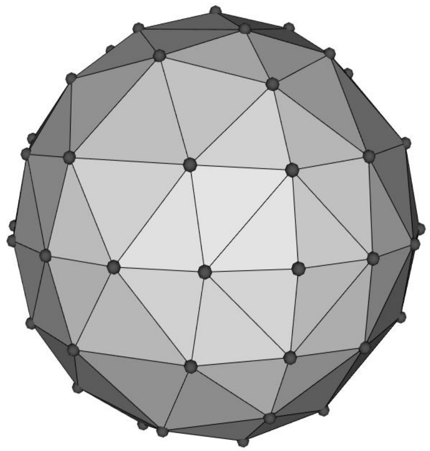
- Quantize vector direction into one of N directions on a sub-divided sphere

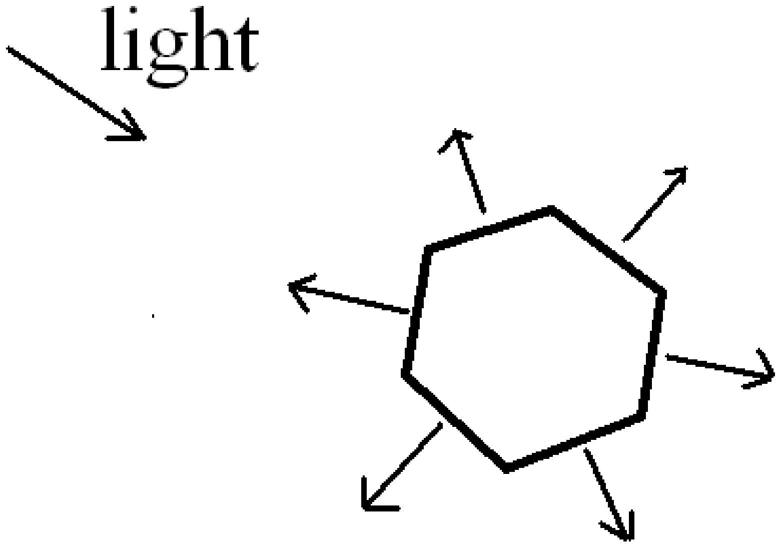
Subdivide an octahedron into a sphere.

Number the vertices.

Encode the direction according to the nearest vertex that the vector passes through.

For infinite light sources, only need to calculate the shading values once and store these in a table.





Volume Illumination: Summary

- Computing the color using the lighting can greatly helps us to understand the 3D structure
- Illumination calculation needs to be done per voxel
- Normals can be computed using the gradient of the intensity
- Opacity scaled by the gradient helps to visualize the boundaries between the tissues

Overview

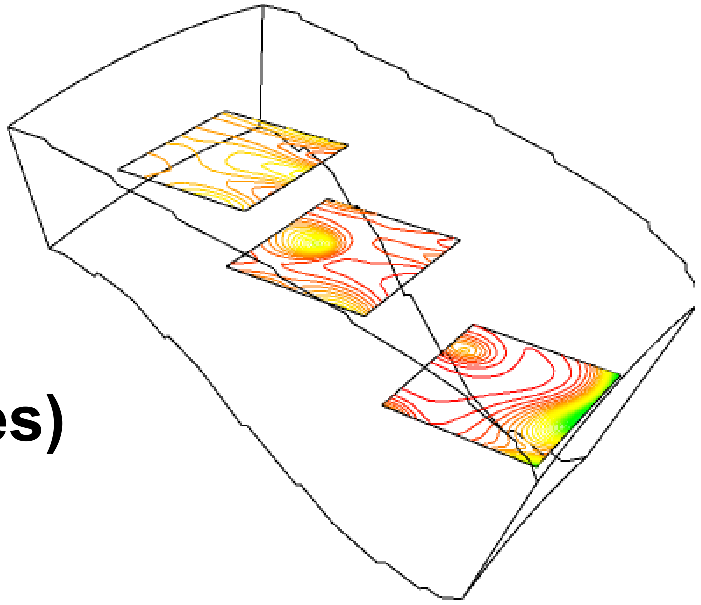
- Volume Illumination

- **Contouring**
  - Problem statement
  - Tracking
  - Marching squares
  - Ambiguity problems
  - Marching cubes
  - Dividing squares

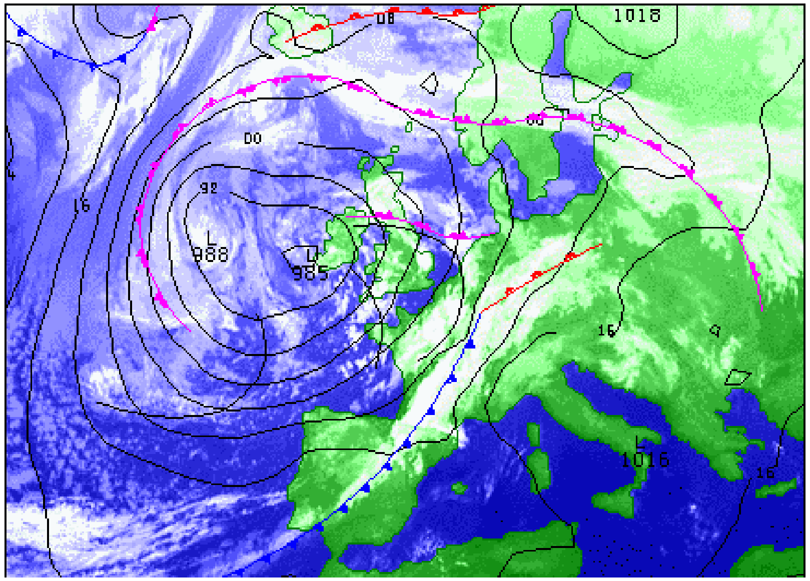
Contouring

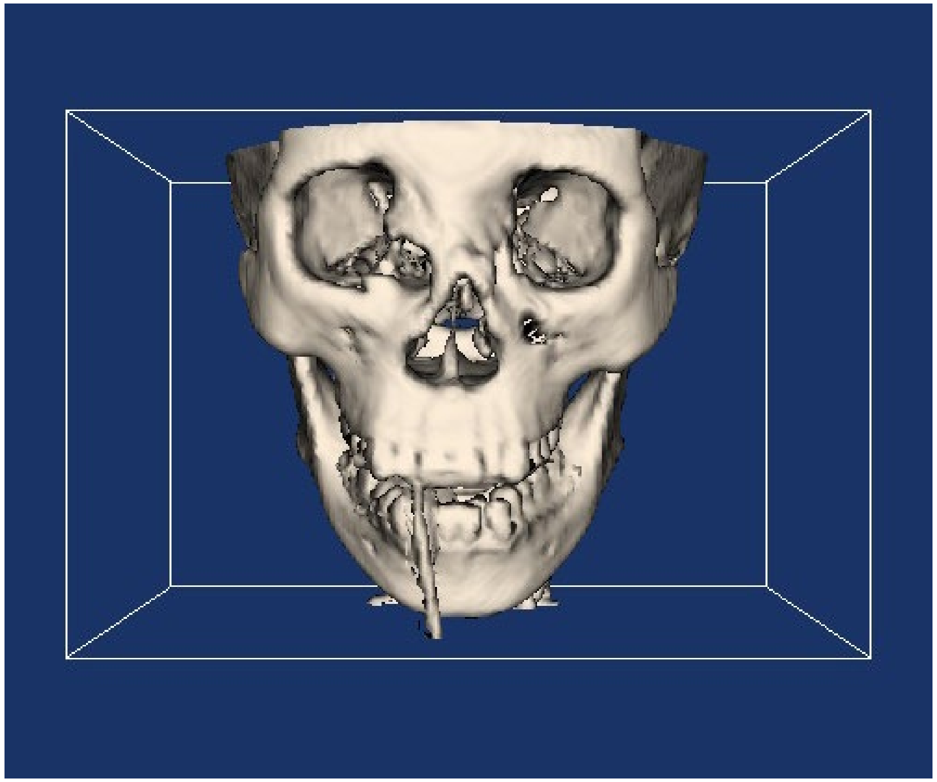
- Contours explicitly *construct* the **boundary between regions with values**

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



Example : contours

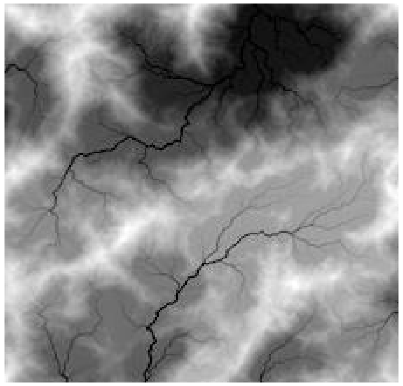


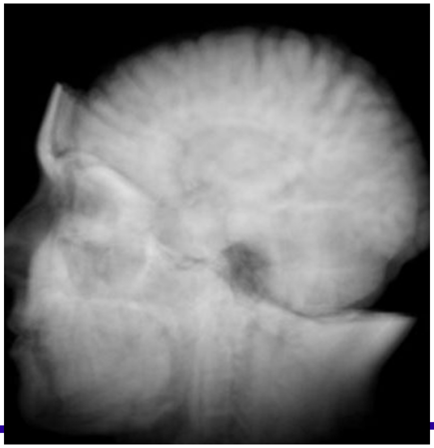


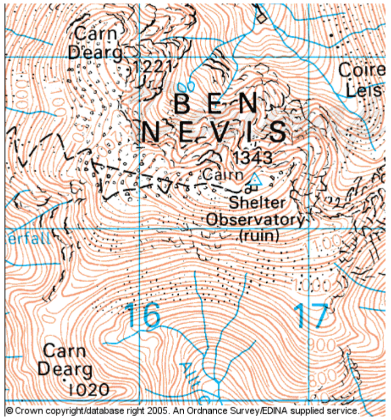
- lines of constant pressure on a weather map (isobars)
- surfaces of constant density in medical scan (isosurface)
  - “iso” roughly means equal / similar / same as

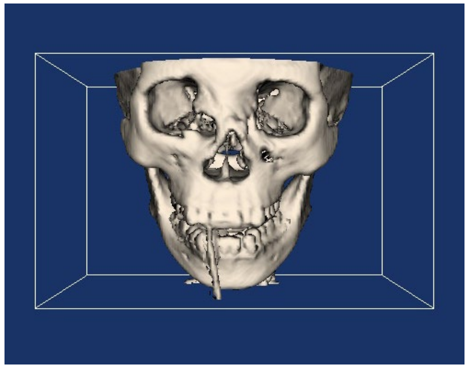
Contouring

- Input : 2D or 3D grid with scalar values at the nodes
- Output : Contours (polylines, polygons) that connect the vertices with the same scalar value



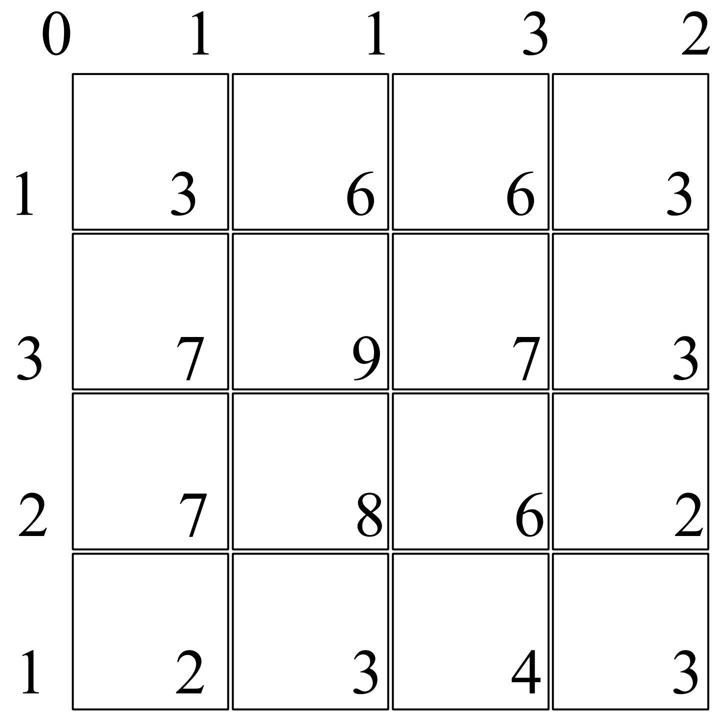






2D Contour

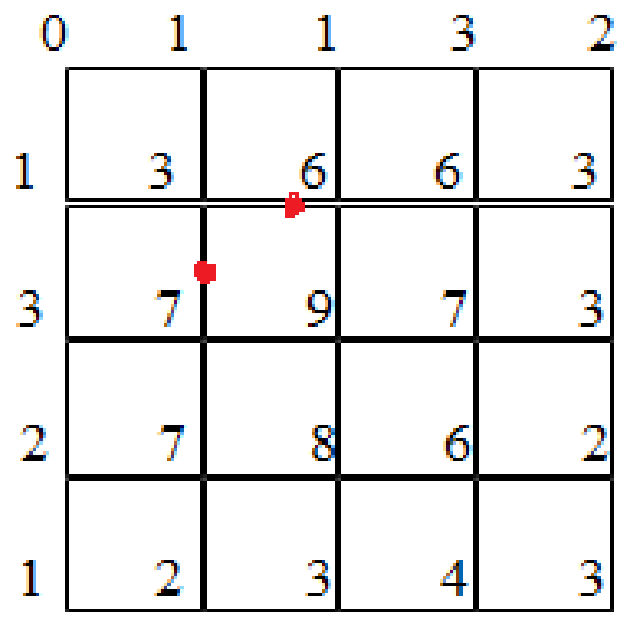
- **Input Data:** 2D structured grid of scalar values



- Difficult to visualise transitions in data
  - use **contour** at specific scalar value to highlight **transition**
- What is the contour of 5**?**

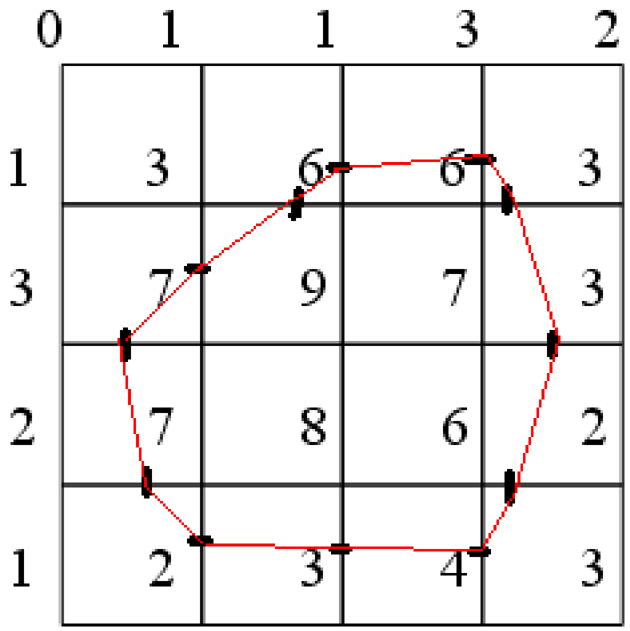
Method 1: Tracking

- **Procedure**
  - 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
  - If it is known to be only one contour, stop
  - *otherwise*
  - Check every edge



Method 1: Tracking

- Procedure
  - 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
  - If it is known to be only one contour, stop
  - *otherwise*
  - Check every edge



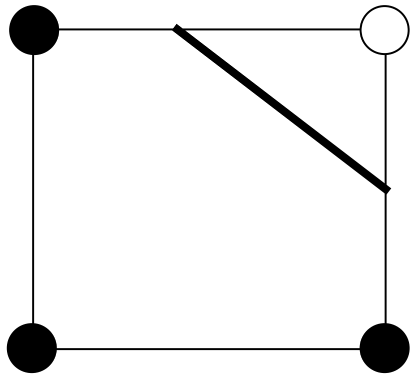
Issues: A sequential process

-> hard to parallelize

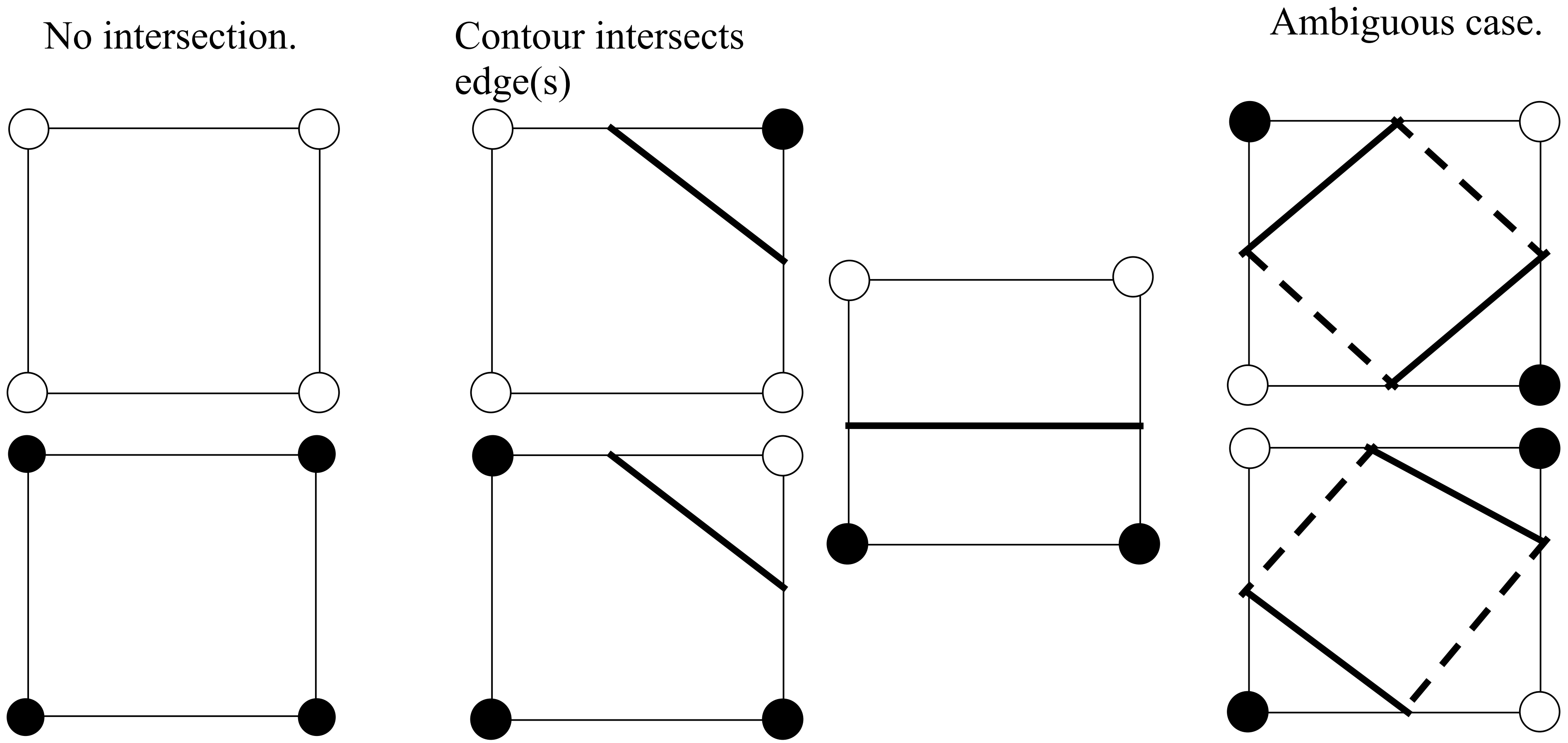
Method 2: Marching Squares (Cube) Algorithm

- Focus : intersection of contour and cell edges
  - how the contour passes through the cell
- Assumption: a contour can pass through a cell in only a finite number of ways
  - A vertex is inside contour if scalar value > contour

    outside contour if scalar value < contour
  - 4 vertices, 2 states (in or out)

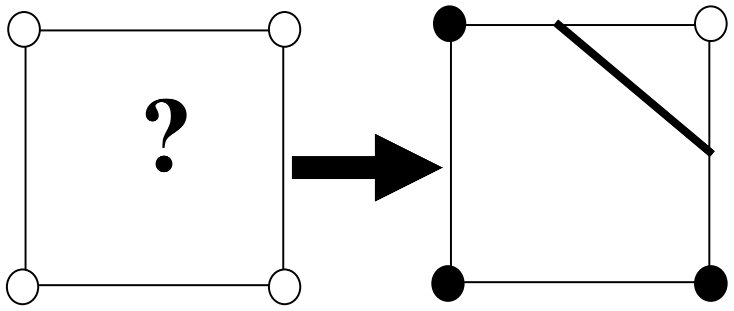


Marching Squares



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

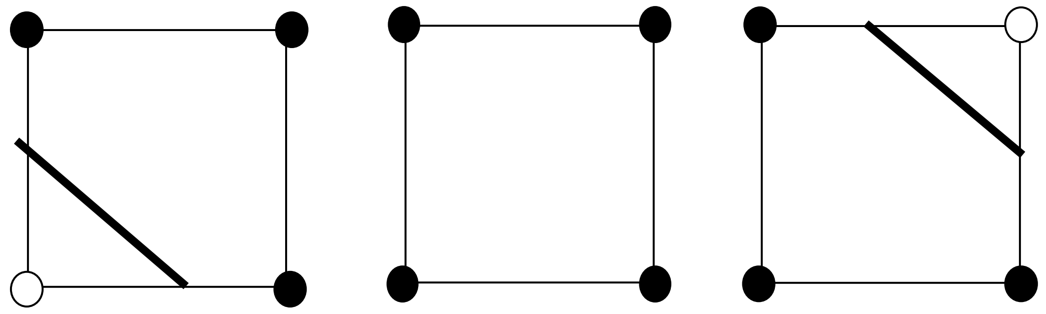
MS Algorithm Overview



- **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**

MS Algorithm - notes

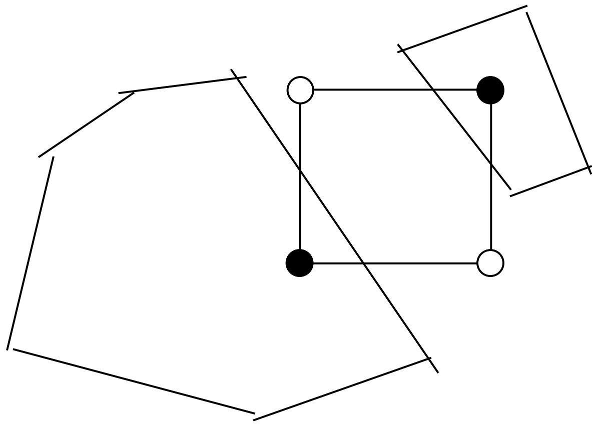


- Intersections for each cell must be merged to form a complete contour
  - cells processed independently
  - further **“merging” computation required**
  - disadvantage over tracking (continuous tracked contour)

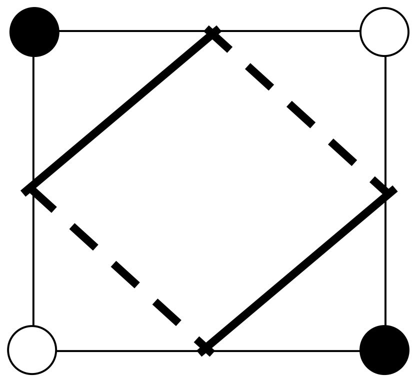
- Easy to implement (also to extend to 3D)
- Easy to parallelise

MS : Dealing with ambiguity ?

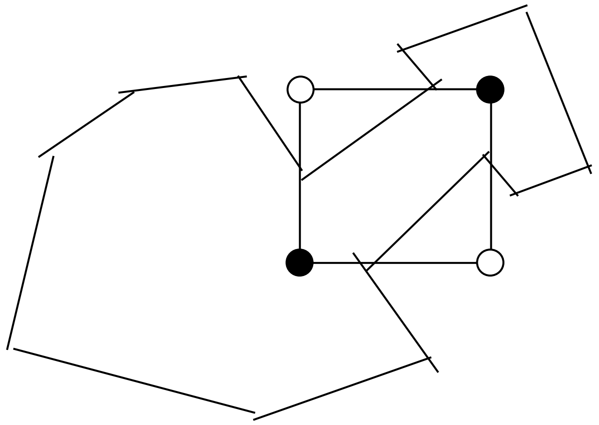
Split

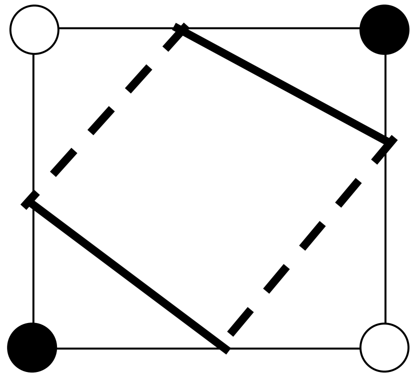


Ambiguous case.



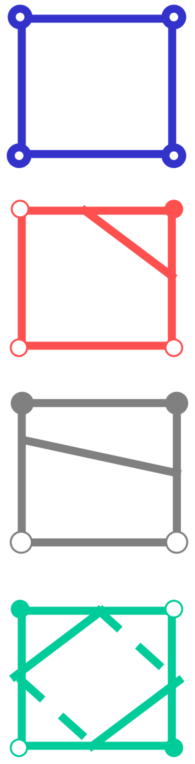
Join





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

Example : Contour Line Generation

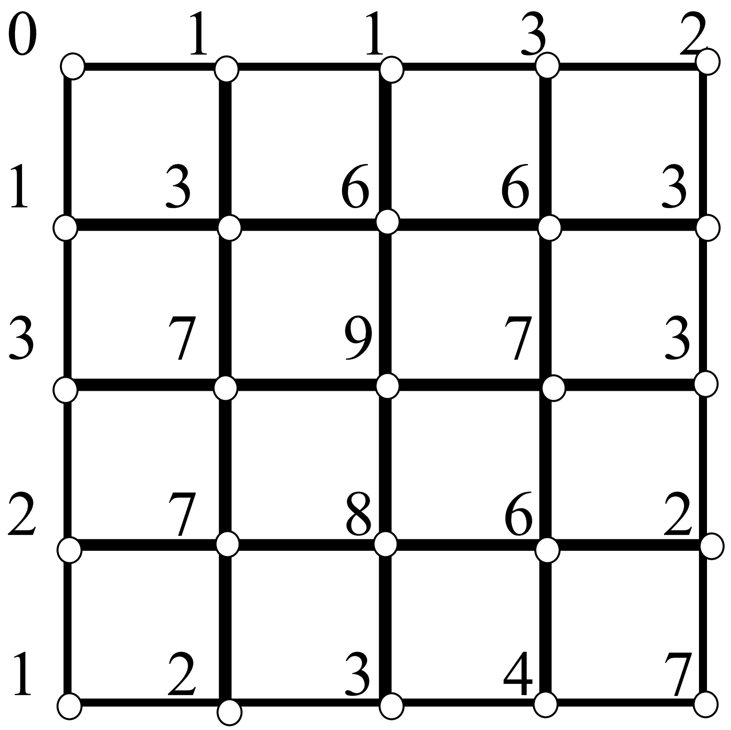


No intersection.

Contour intersects 1 edge

Contour intersects 2 edges

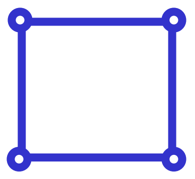
Ambiguous case.



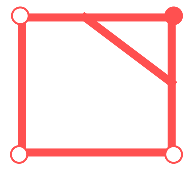
0	1	1	3	2
1	3	6	6	3
3	7	9	7	3
2	7	8	6	2
1	2	3	4	7

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

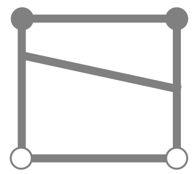
Step 1 : classify vertices



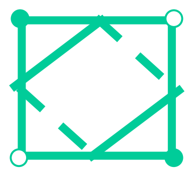
No intersection.



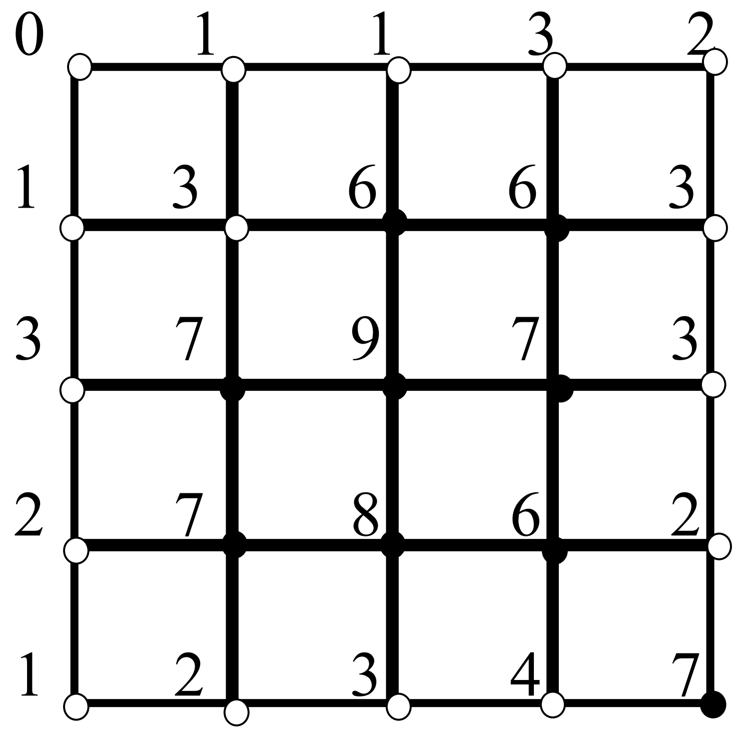
Contour intersects 1 edge



Contour intersects 2 edges



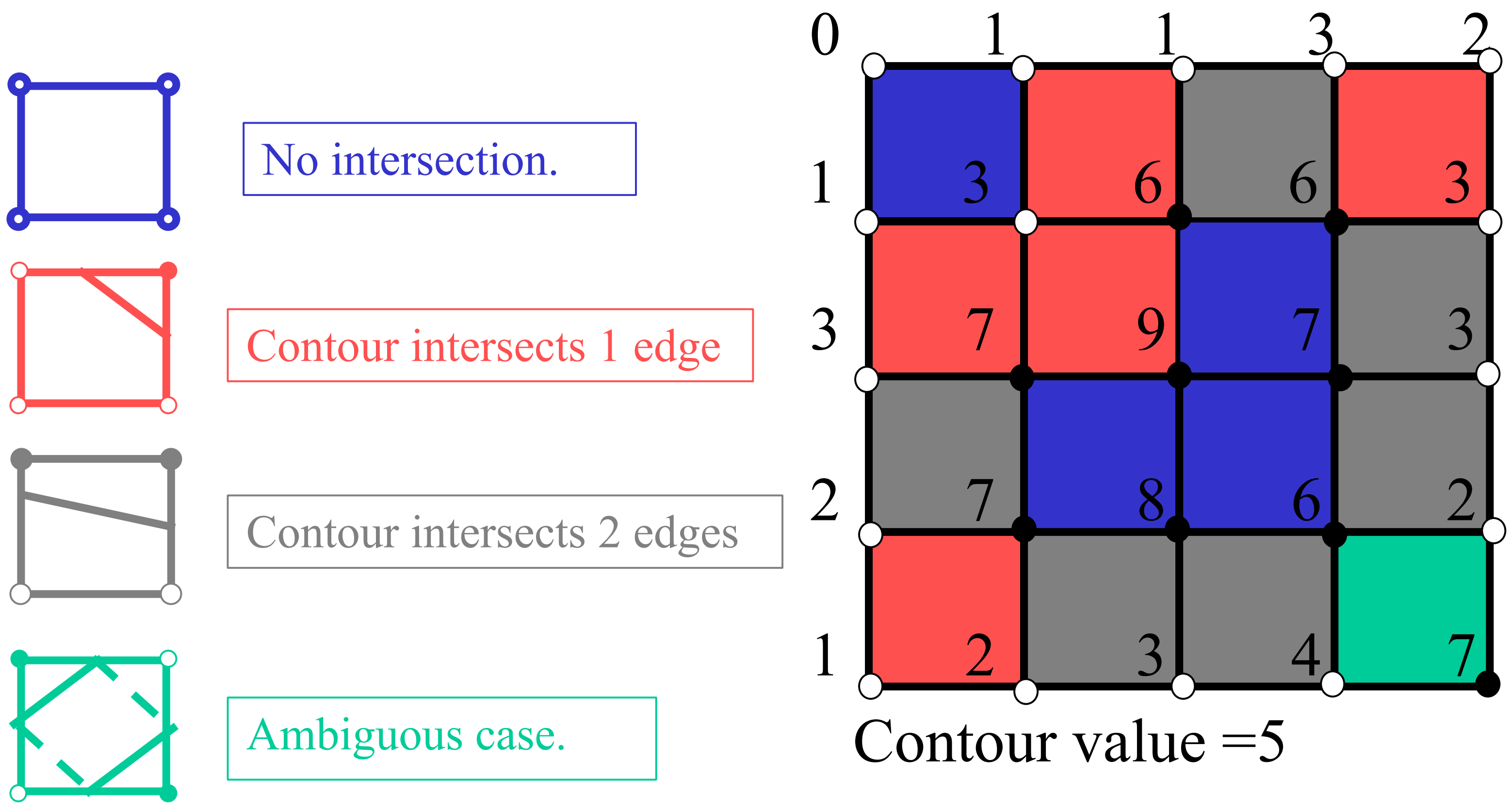
Ambiguous case.



Contour value =5

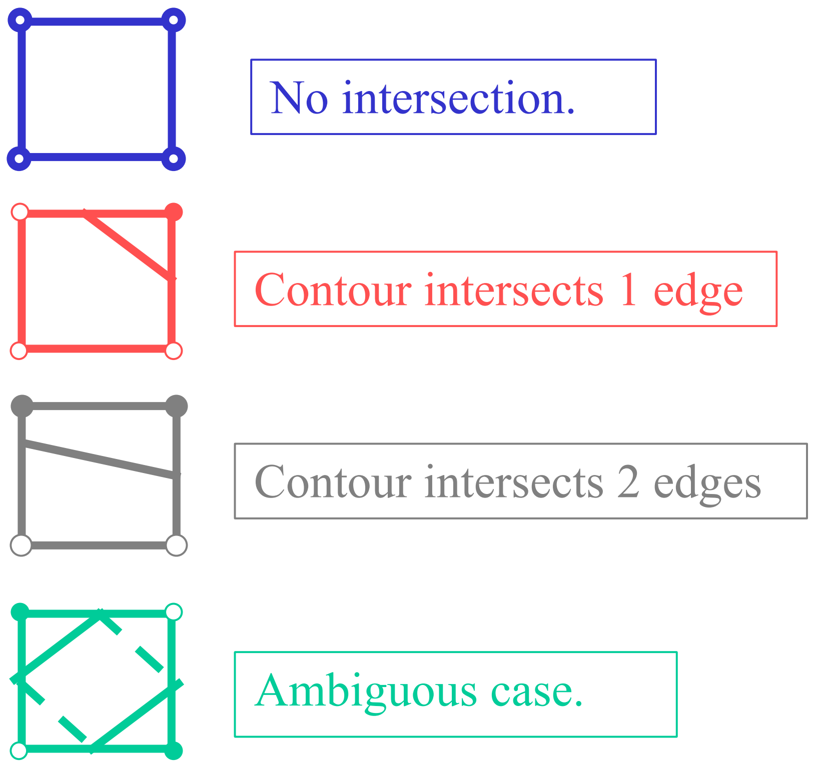
- Decide whether each vertex is inside or outside contour

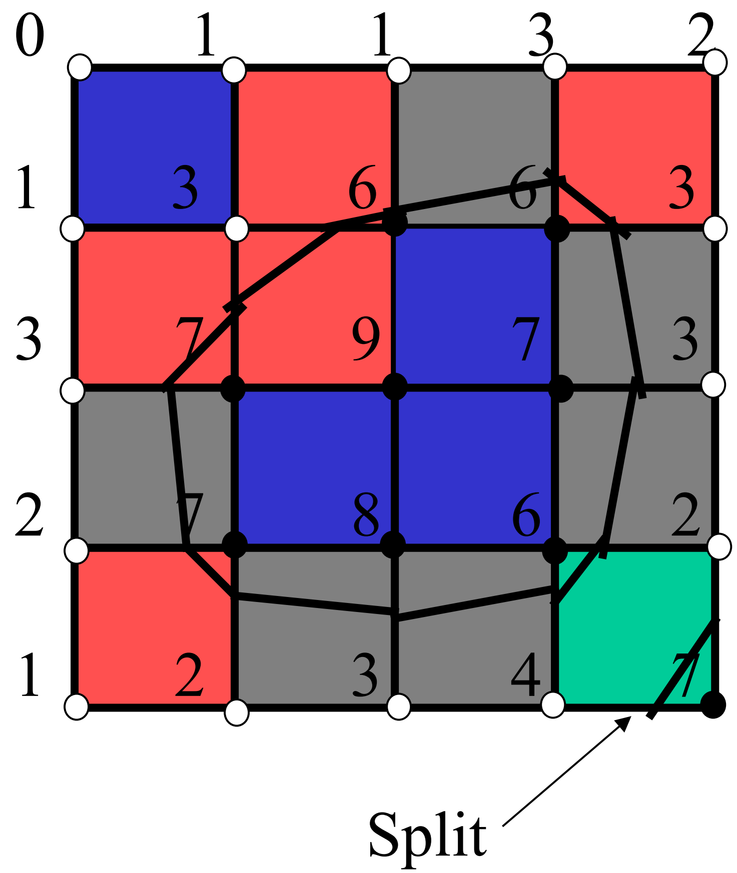
Step 2 : identify cases



- Classify each cell as one of the cases

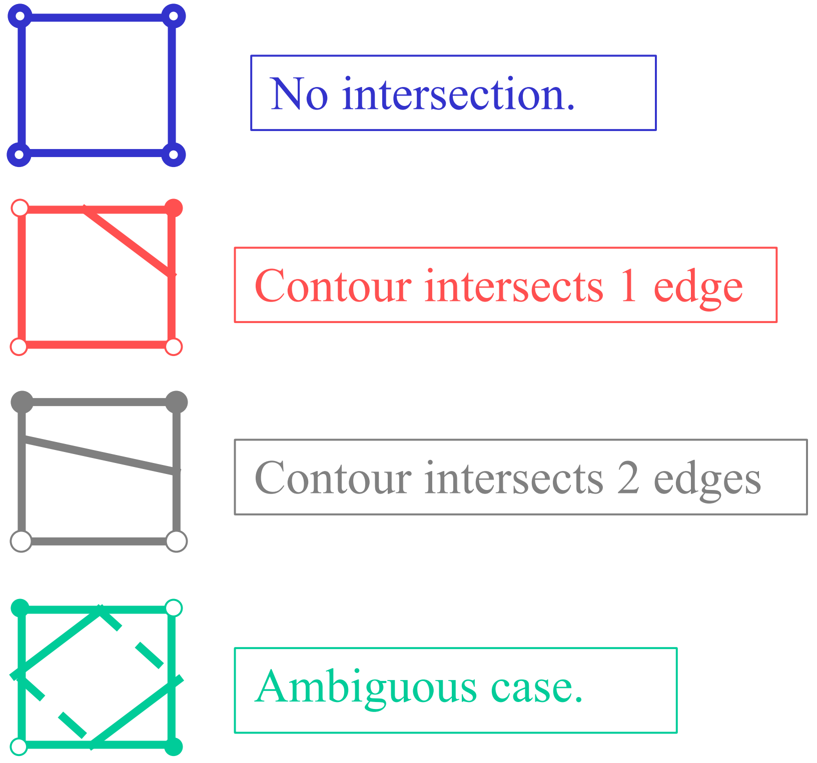
Step 3 : interpolate contour intersections

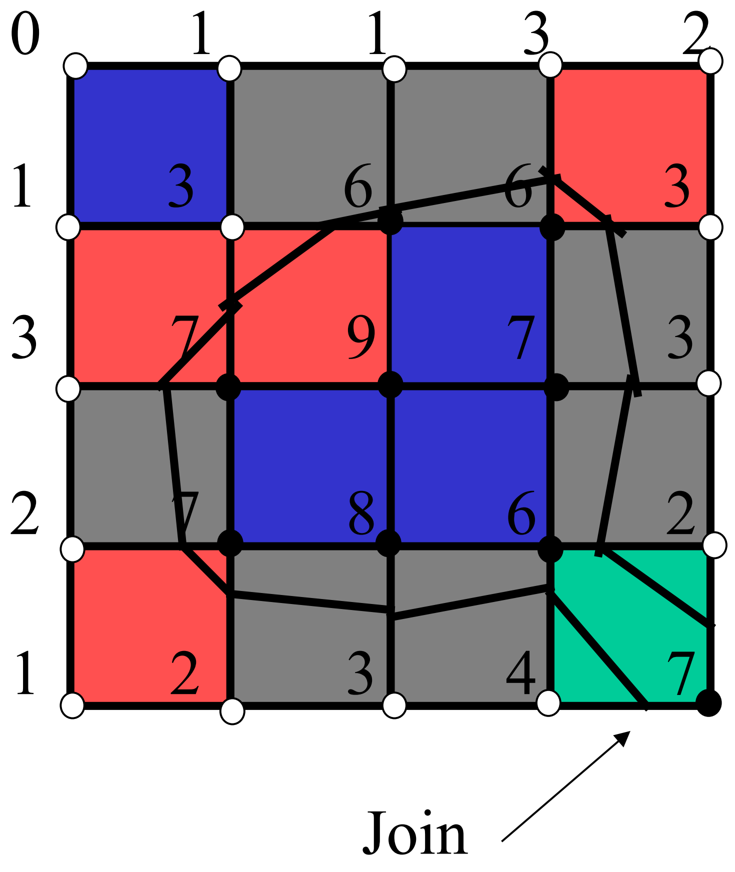




- Determine the edges that are intersected
  - compute contour intersection with each of these edges

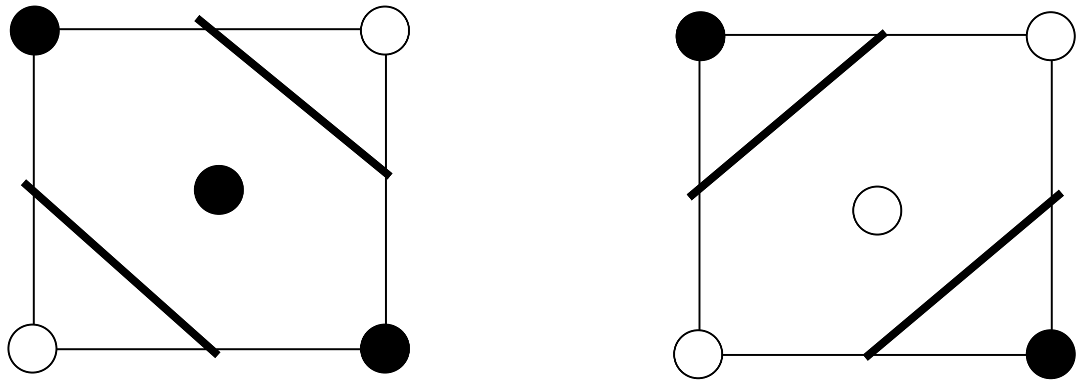
Ambiguous contour





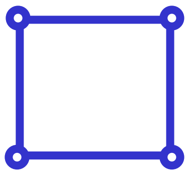
- Finally : resolve any ambiguity
  - here choosing “join” (example only)

MS : Dealing with ambiguity ? One solution

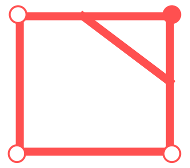


- Calculate the value at the middle of the square by interpolation
- Check if it is under or above the threshold value
- Choose the pattern that matches

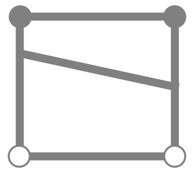
Ambiguous contour



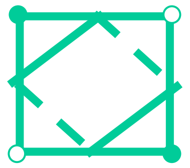
No intersection.



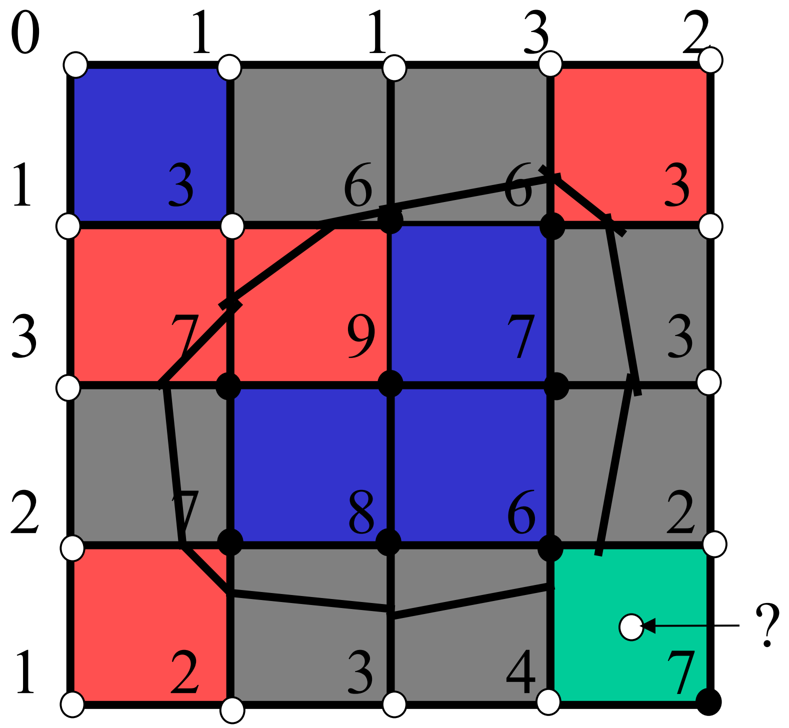
Contour intersects 1 edge



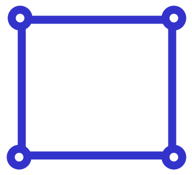
Contour intersects 2 edges



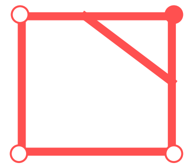
Ambiguous case.



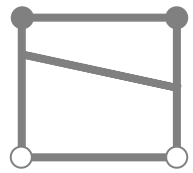
Step 3 : interpolate contour intersections



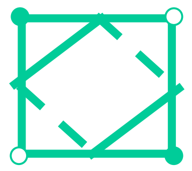
No intersection.



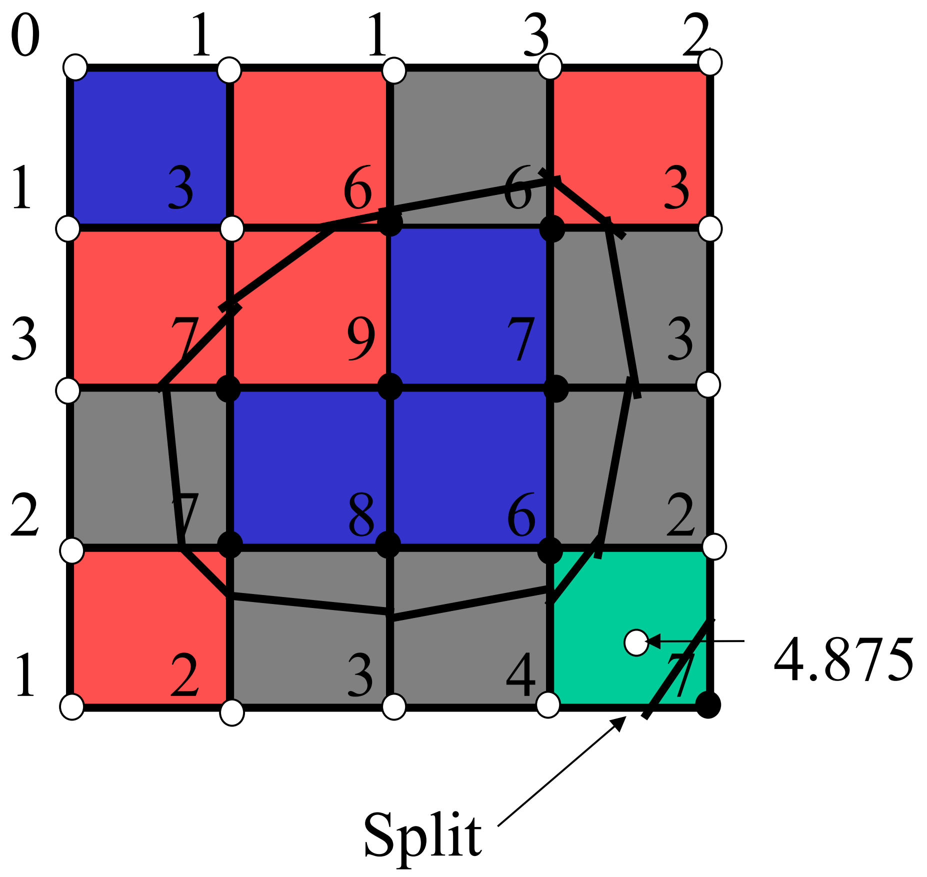
Contour intersects 1 edge



Contour intersects 2 edges

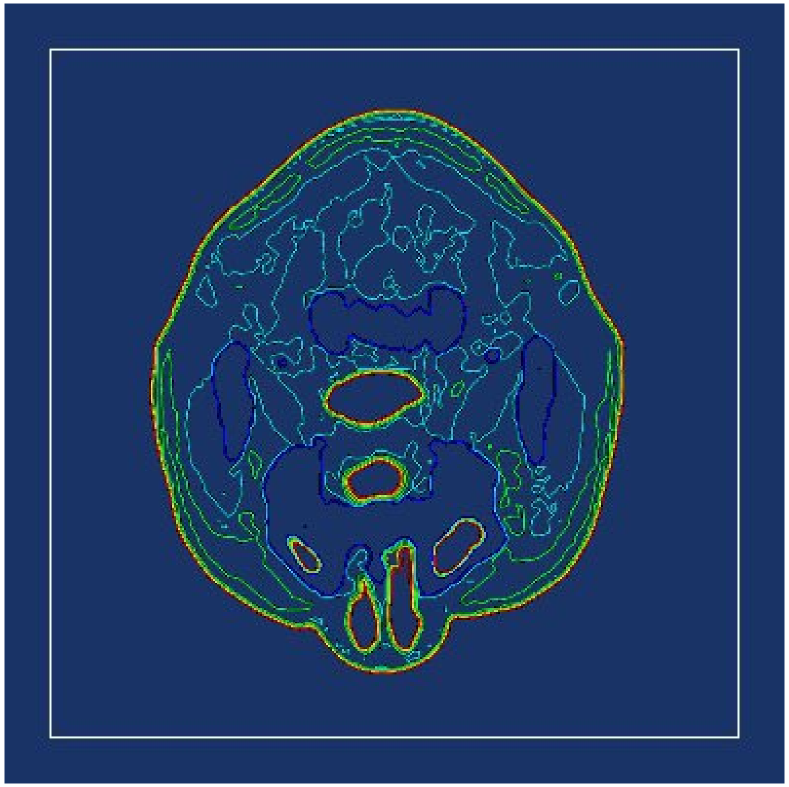


Ambiguous case.



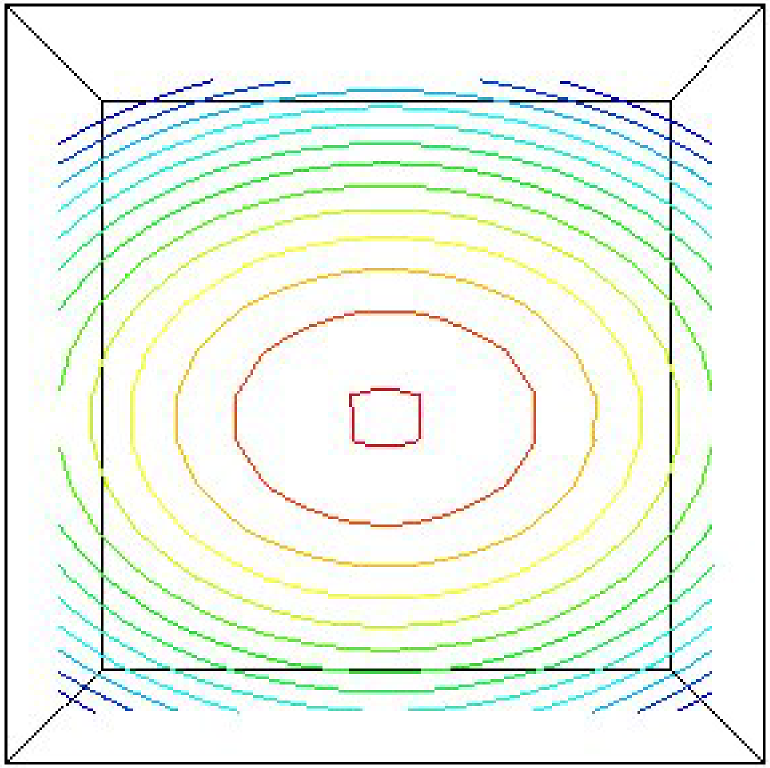
2D : Example contour

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, reflection, rotation 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

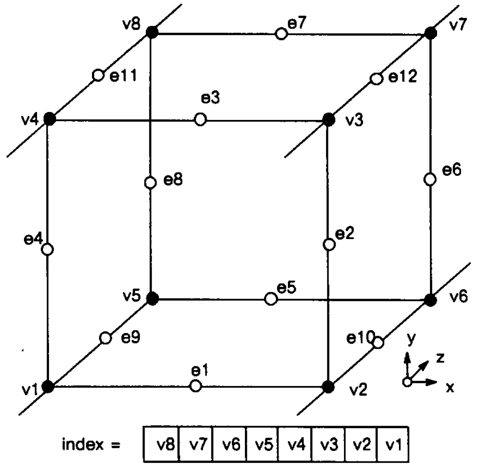


Figure 4. Cube Numbering.

Marching Cubes - cases

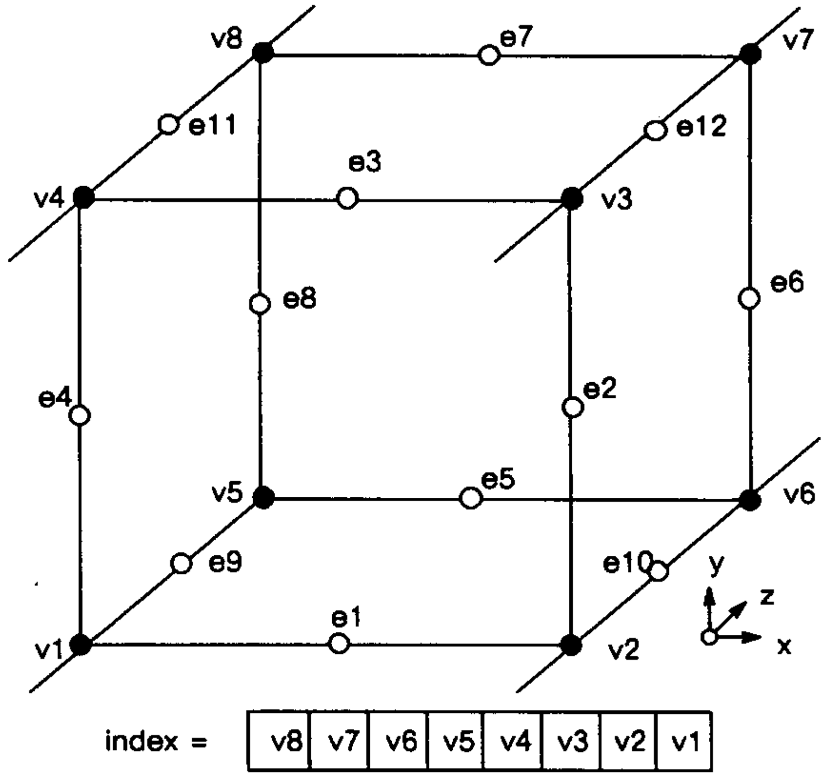


Figure 4. Cube Numbering.

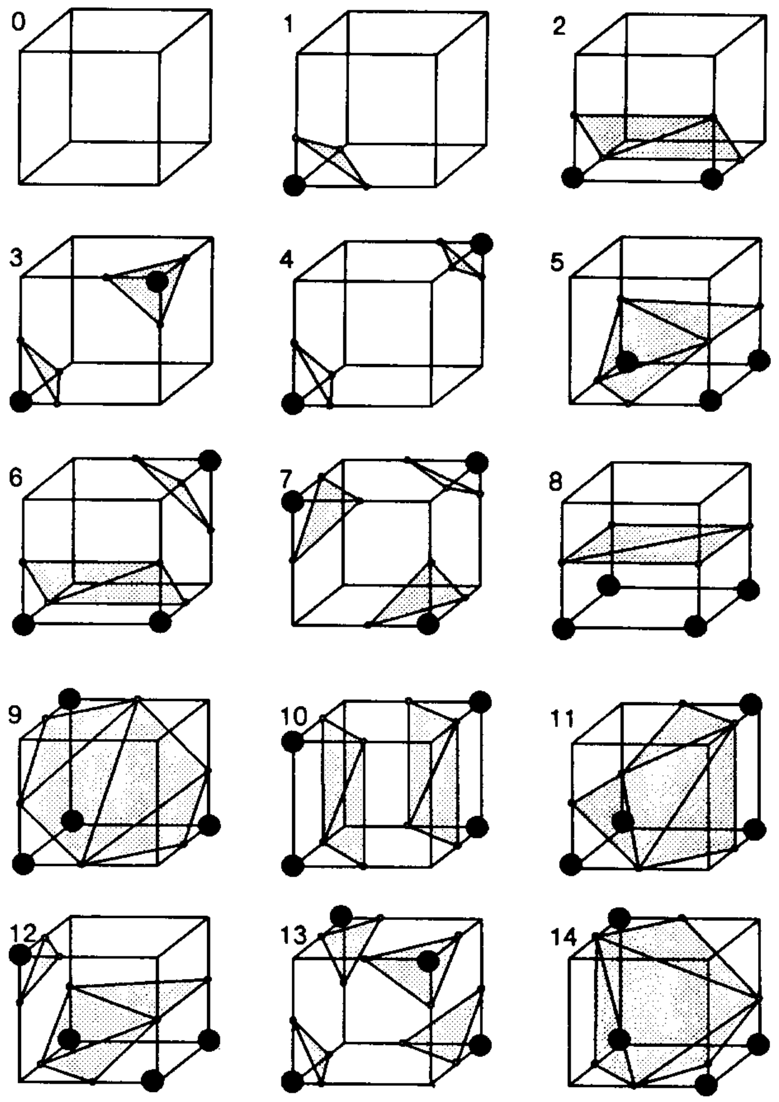
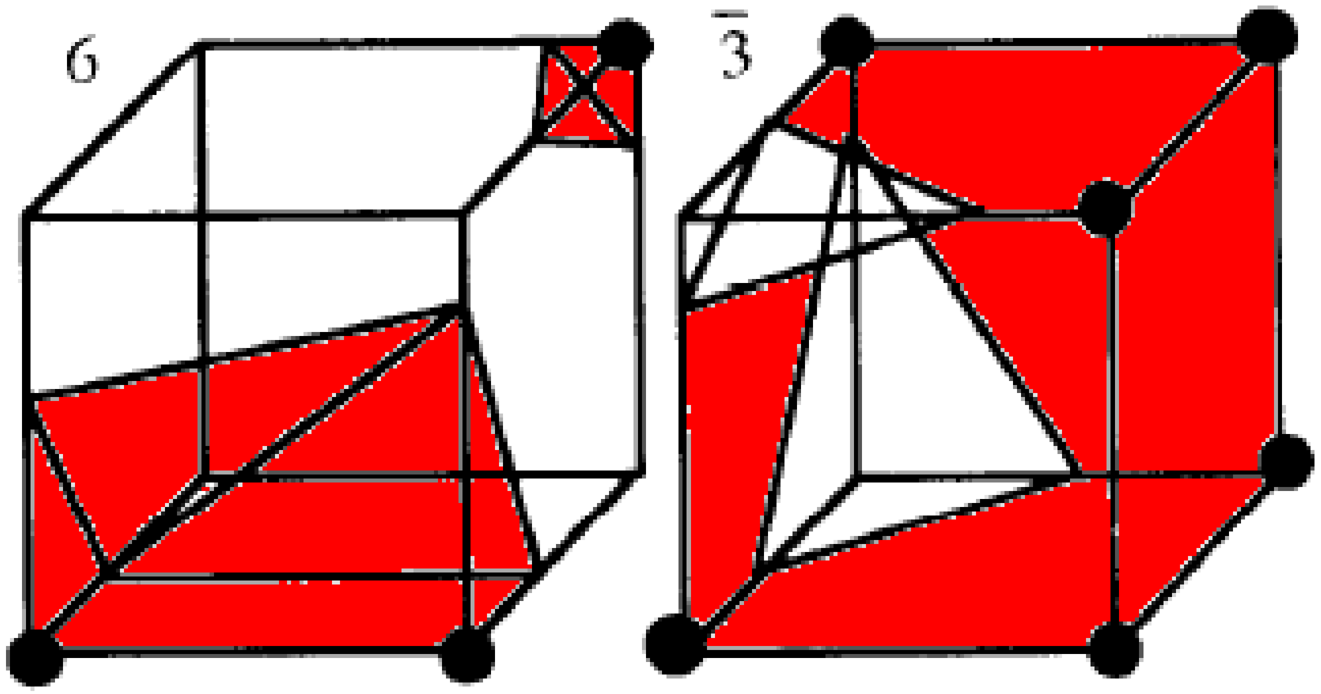


Figure 3. Triangulated Cubes.

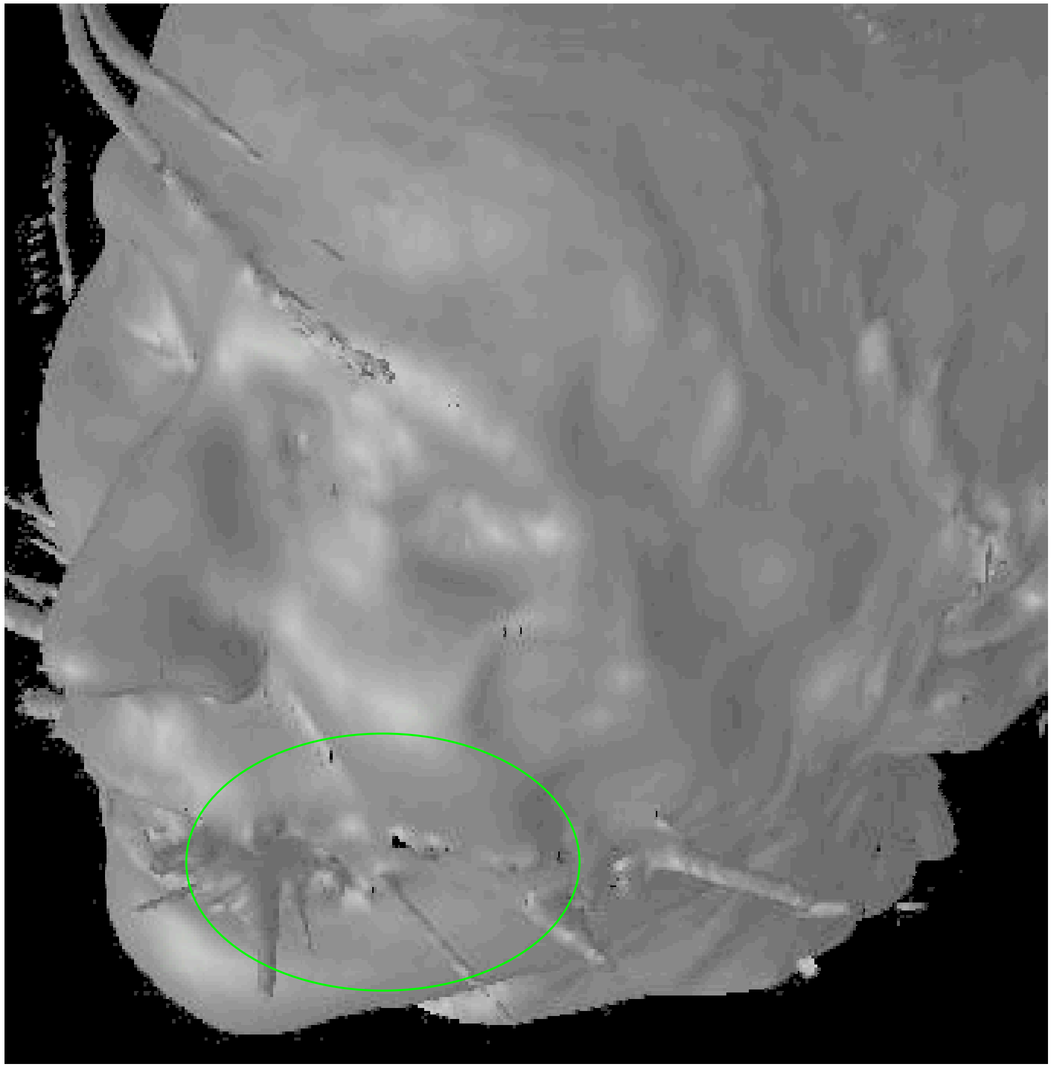
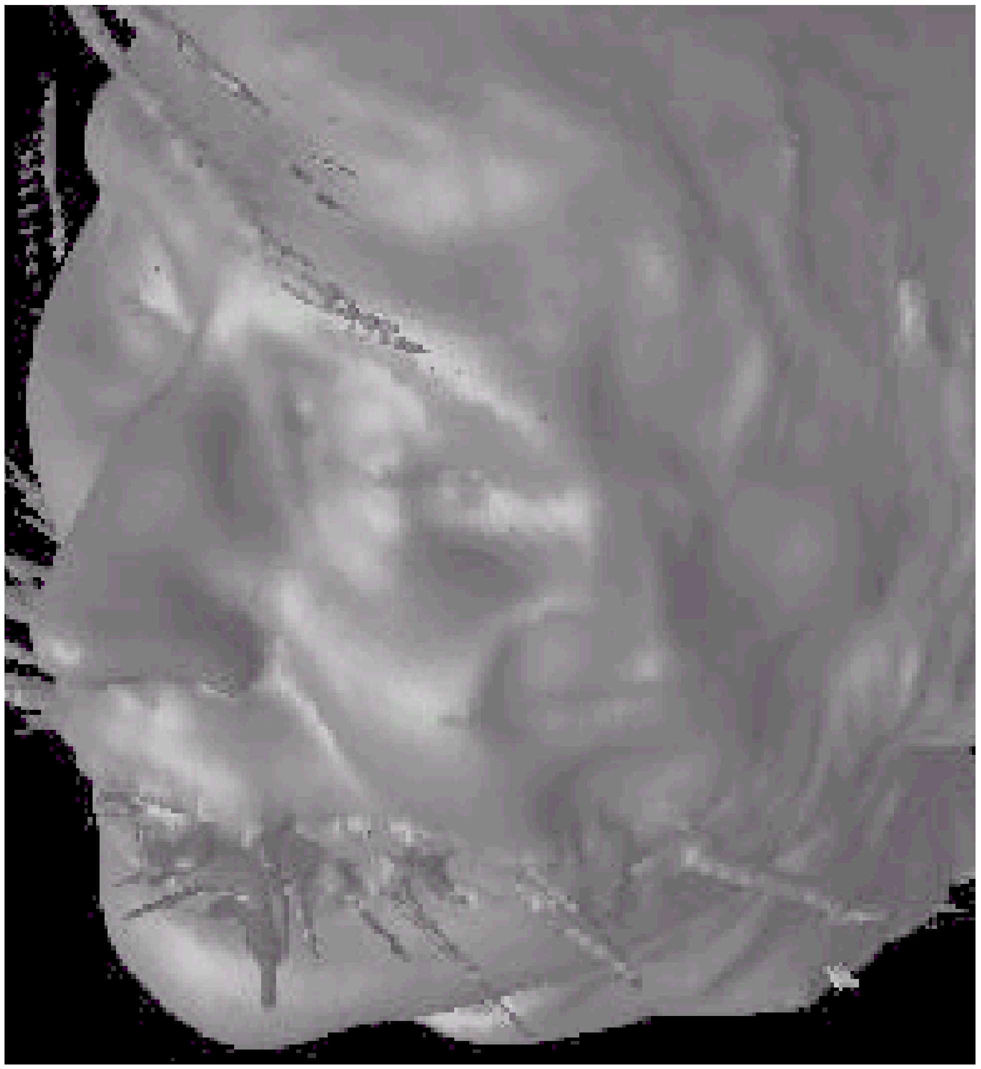
- Ambiguous cases
  - 3,6,10,12,13 – split or join ?

Example of bad choices

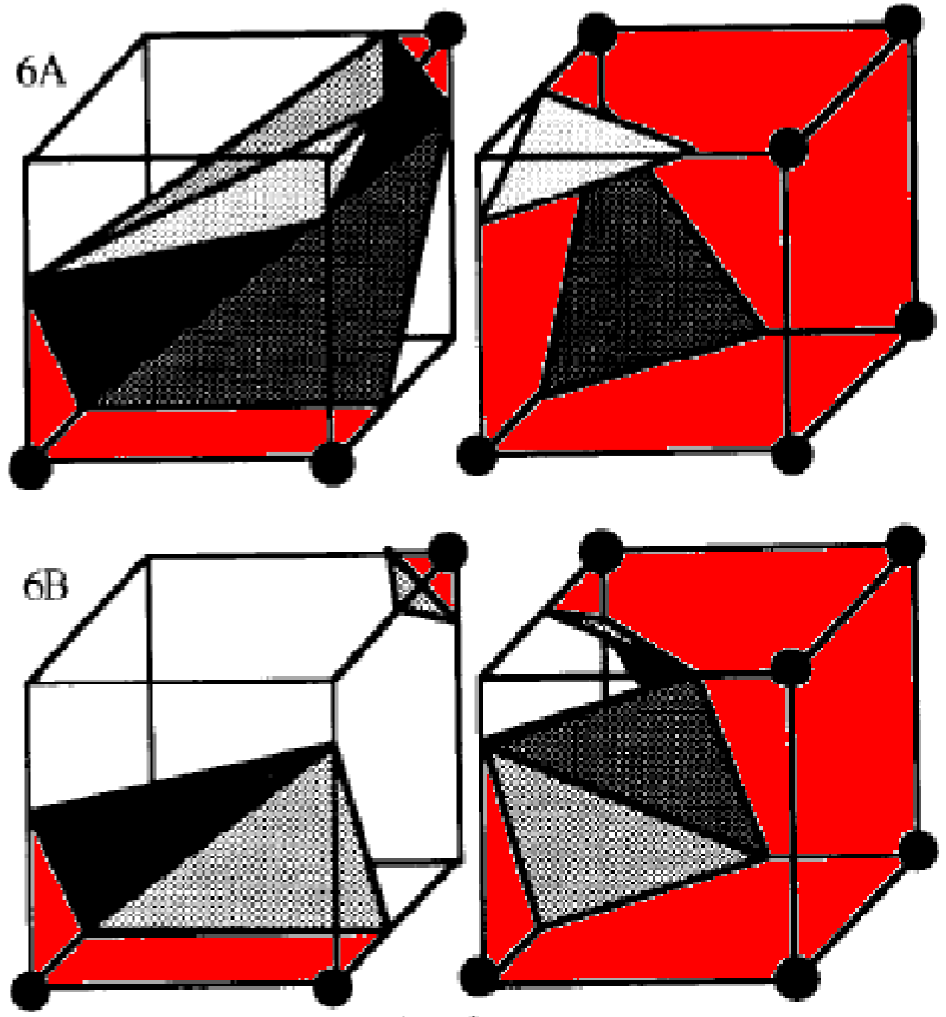


- The dark dots are the interior
- There are edges which are not shared by both cubes
- Need to make sure there is no contradiction with the neighbors

Cracks eliminated

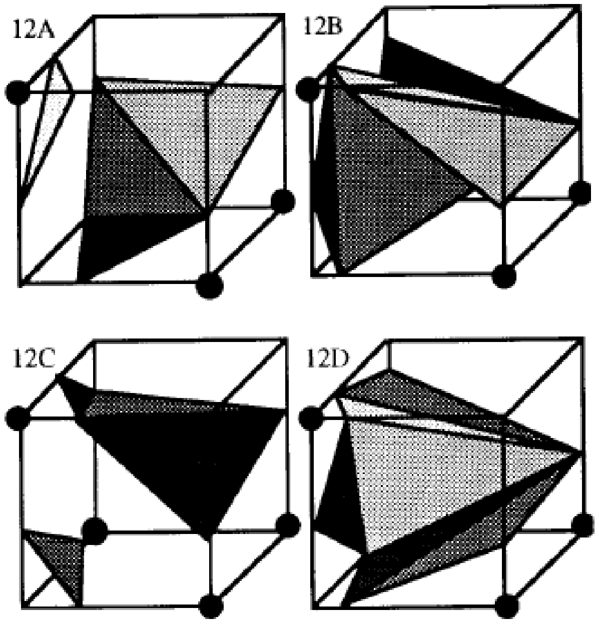
Other two possible triangulations

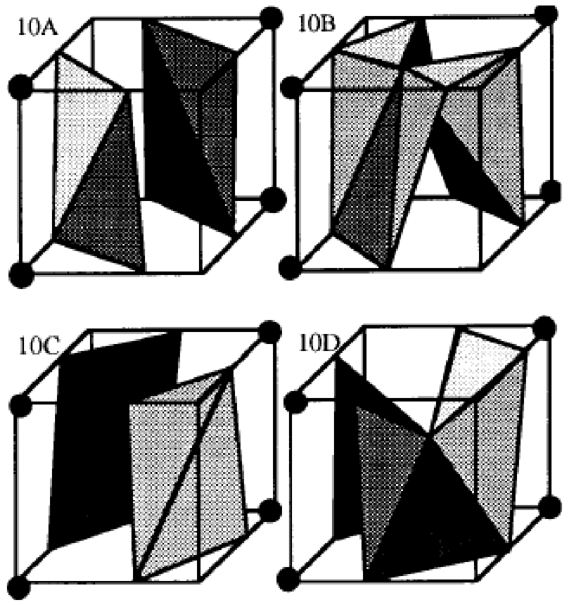


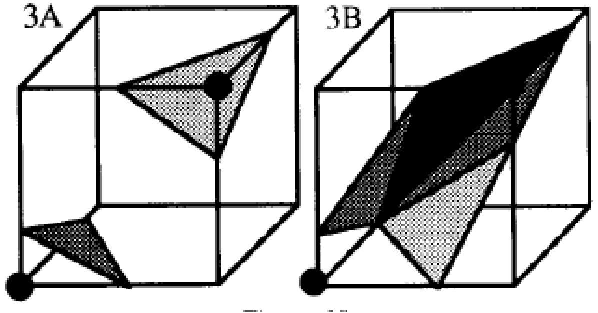
- Need to decide how the faces are intersected by the contours

Adding more patterns

- Adding more patterns for 3,6,10,12,13 [Nielson '91]
- Compute the values at the middle of the faces and the cubes
- Selecting the pattern that matches







Marching Cubes by CUDA

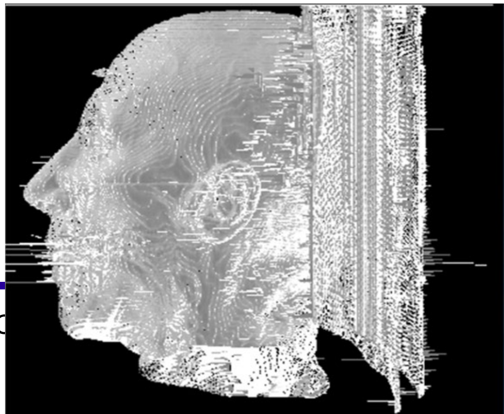
- http://www.youtube.com/watch?v=Y5URxpX8q8U

Dividing Cubes Algorithm

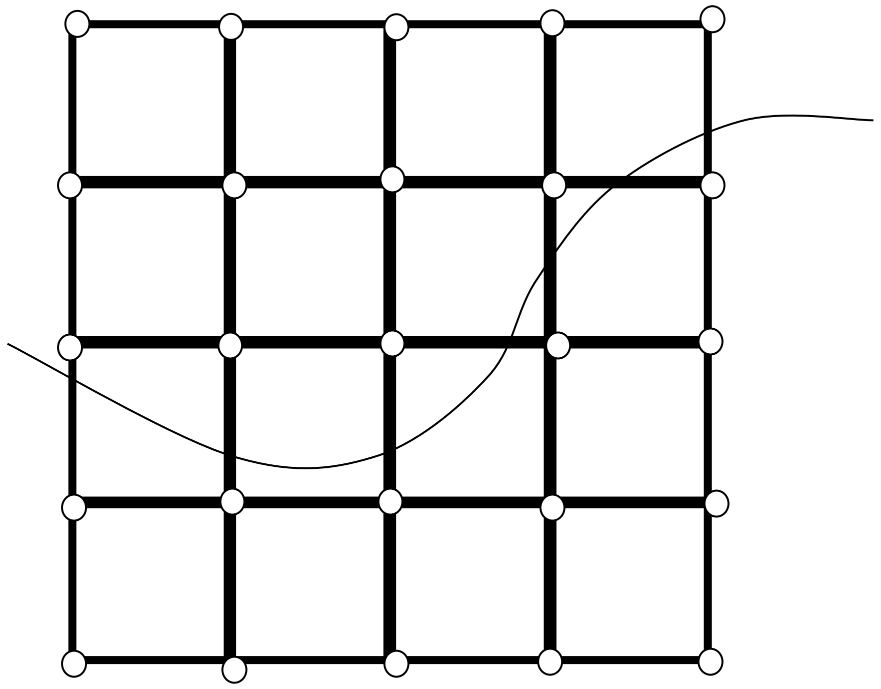
- **Marching cubes : Problem**
  - often **produces more polygons than pixels** for given rendering scale
  - **Problem :** causes **high rendering overhead**

- **Solution : Dividing Cubes Algorithm**
  - **Colouring pixels instead of rendering polygons** ***(faster rendering)***
  - **Need 1: efficient method to find which pixels to colour**

    2: method to shade points

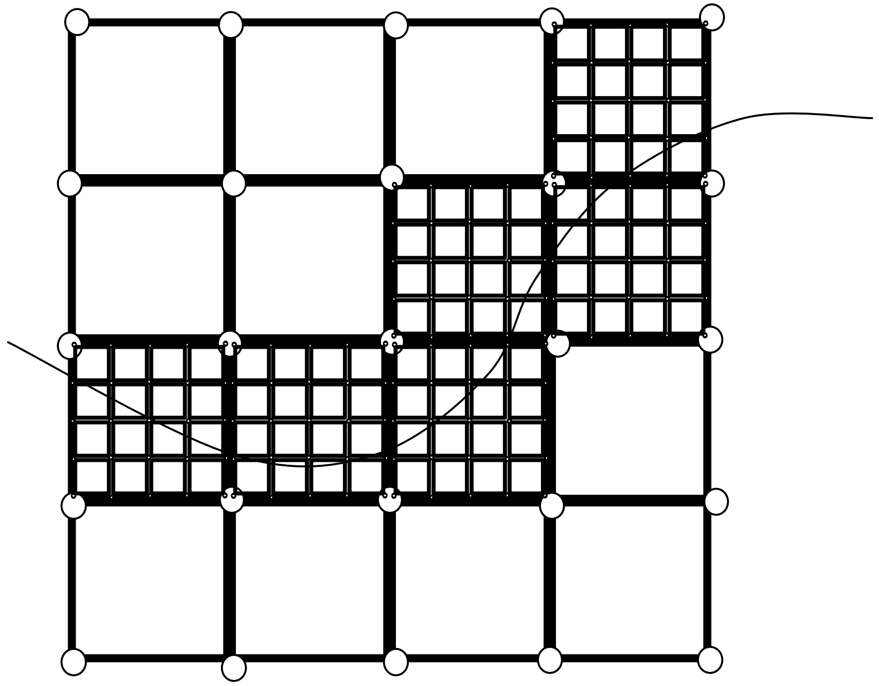


Example : 2D divided squares for 2D lines



Find pixels that intersect contour
- Subdivide them

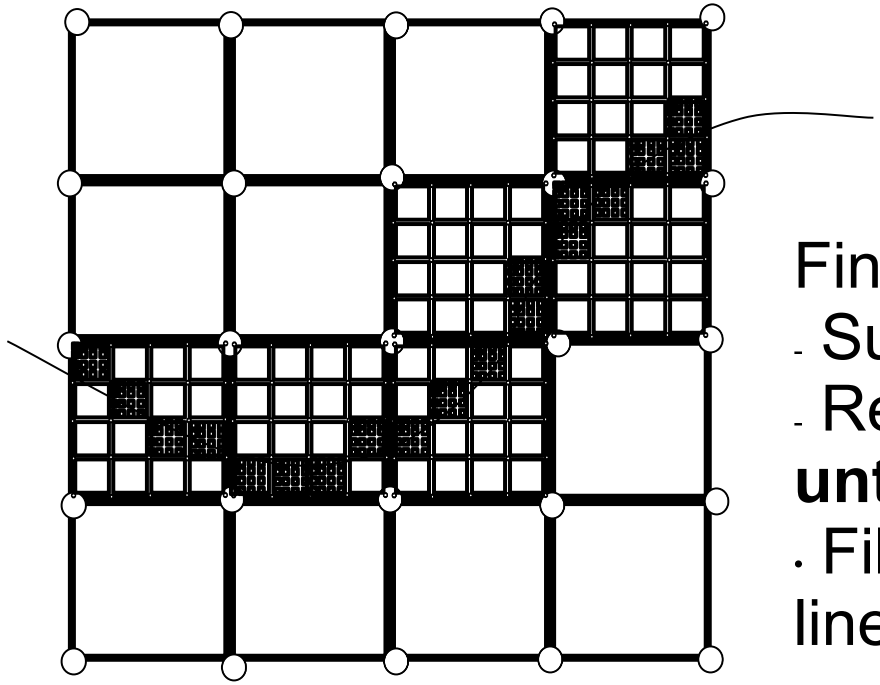
2D “divided squares” for lines



Find pixels that intersect line

- **Subdivide them** (usually in 2x2)
- **Repeat recursively**

2D “divided squares” for lines



Find pixels that intersect line

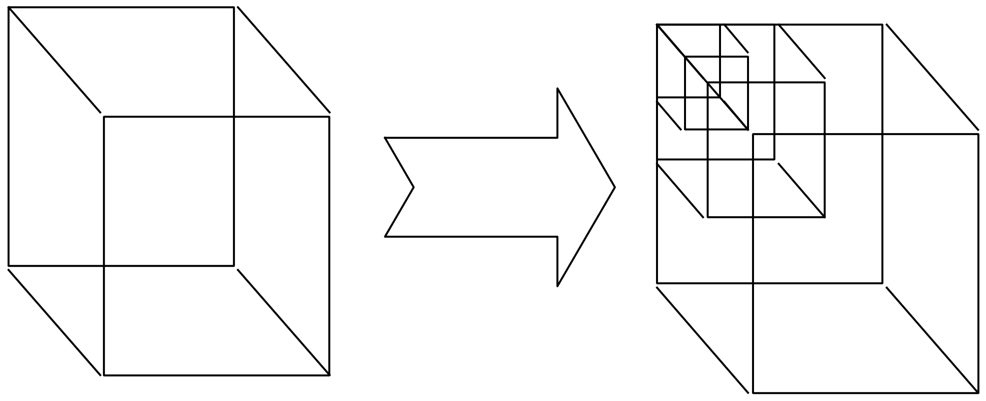
- Subdivide them
- Repeat recursively

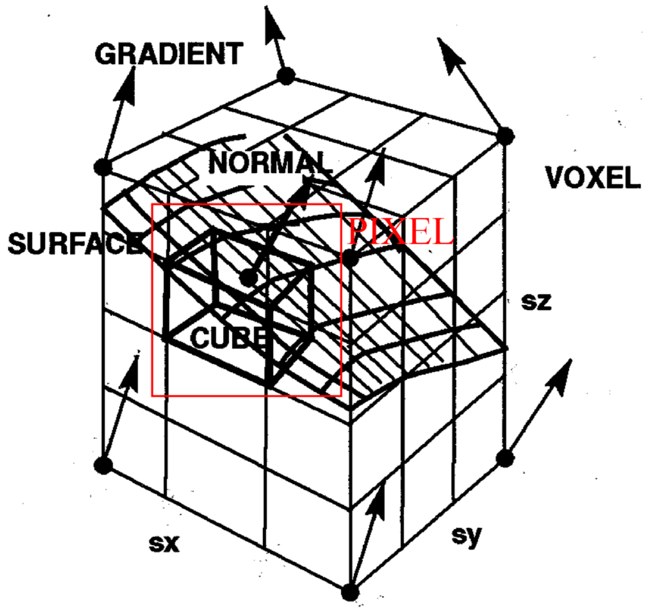
until screen resolution reached

- Fill in the pixel with the color of the line

Extension to 3D

- Find **voxels** which intersect **surface**
- Recursively subdivide the voxels that intersect the contour
  - Until the voxel fits within a pixel
- Calculate **mid-points of voxels**
- Calculate the color of the pixel by shading

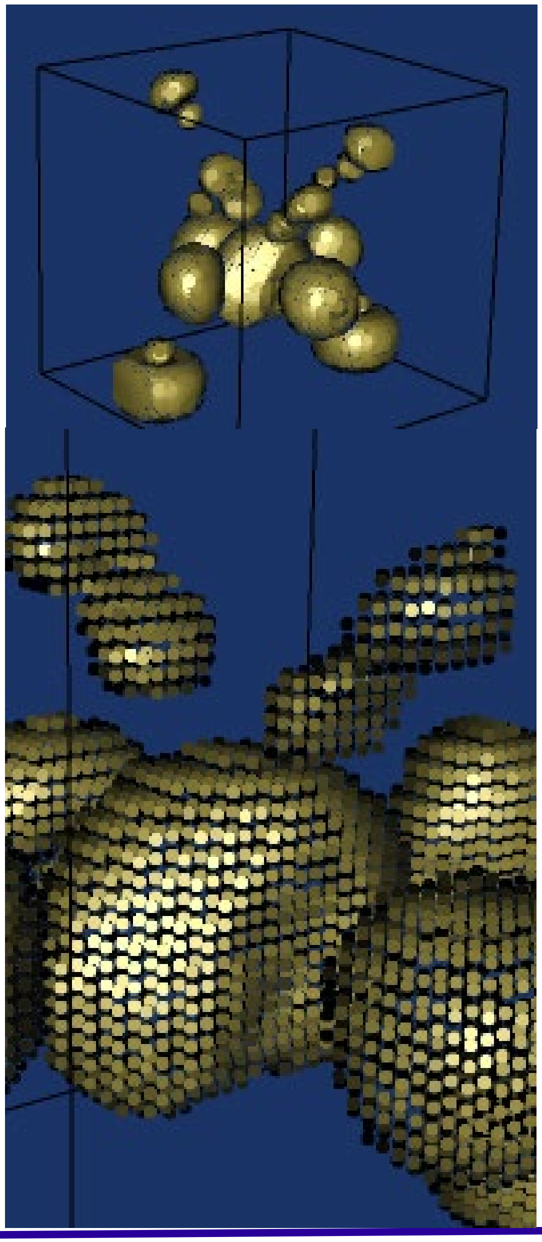




Dividing Cubes : Example

50,000 points

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



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]

Readings

- G.M. Nielson, B Hamann, “The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes”
- W.E. Lorensen, H.E. Cline, “Marching Cubes: A high resolution 3D surface construction algorithm”
- H.E. Cline, W.E. Lorensen and S. Ludke, „Two algorithms for the three-dimensional reconstruction of tomograms”