Computer Graphics

-- Implicit ⇔ Explicit

Junjie Cao @ DLUT

Spring 2019

http://jjcao.github.io/ComputerGraphics/

Conversion

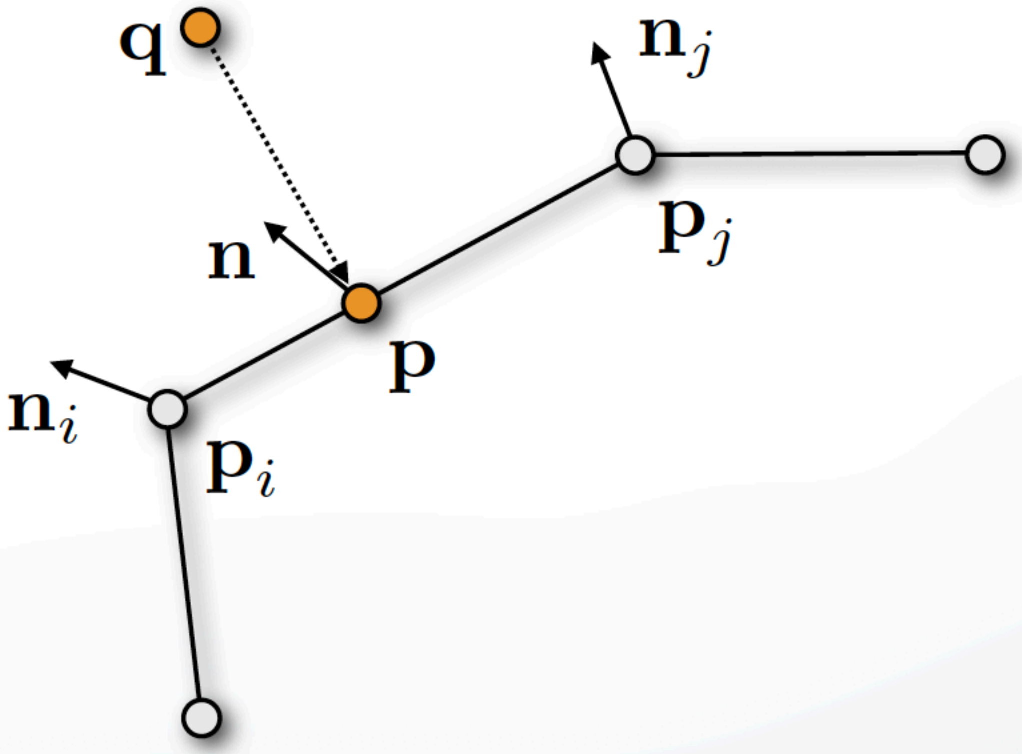
- Explicit to Implicit
  - Compute signed distance at grid points
  - Compute distance point-mesh
  - Fast marching
- Implicit to Explicit
  - Extract zero-level iso-surface $F(x, y, z) = 0$
  - Other iso-surfaces $F(x, y, z) = C$
  - Medical imaging, simulations, measurements, …

Signed Distance Computation

- Find closest mesh triangle
  - Use spatial hierarchies (octree, BSP tree)
- Distance point-triangle
  - Distance to plane, edge, or vertex
  - http://www.geometrictools.com
- Inside or outside?
  - Based on interpolated surface normals

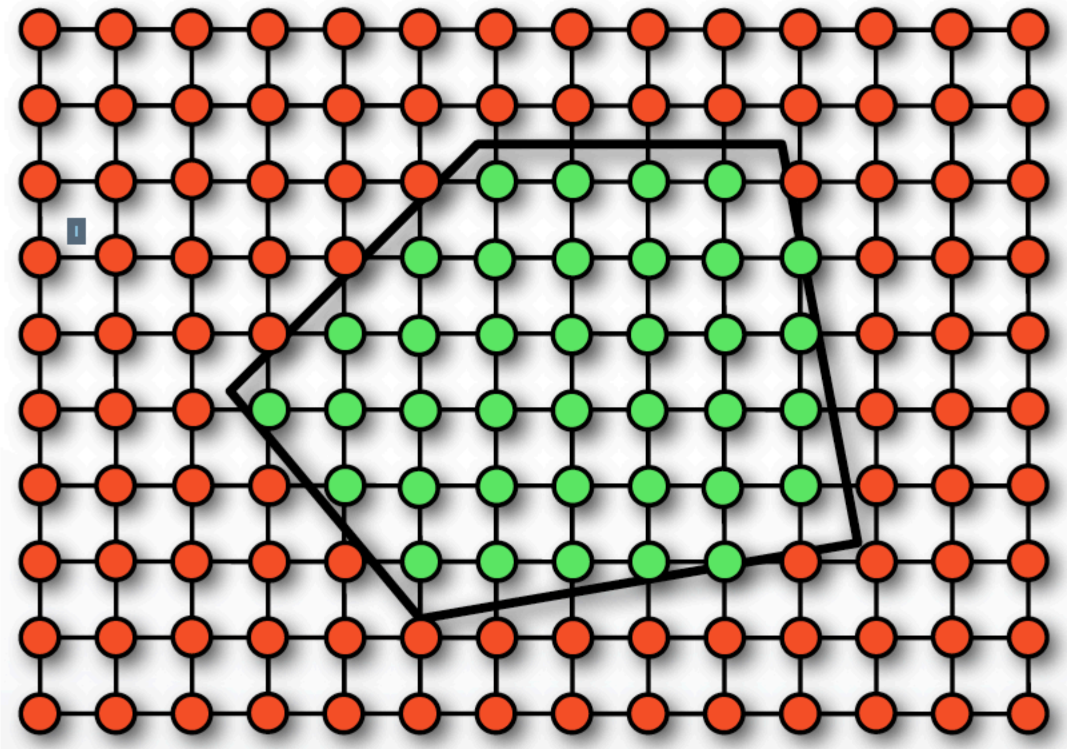
Signed Distance Computation

- Closest point $\mathbf{p} = \alpha \mathbf{p}_i + (1 - \alpha)\mathbf{p}_j$
- Interpolated normal $\mathbf{n} = \alpha \mathbf{n}_i + (1 - \alpha)\mathbf{n}_j$
- Inside if $(\mathbf{q} - \mathbf{p})^\top \mathbf{n} < 0$



Fast Marching Techniques

- Initialize with exact distance in mesh's vicinity
- Fast-march outwards
- Fast-march inwards

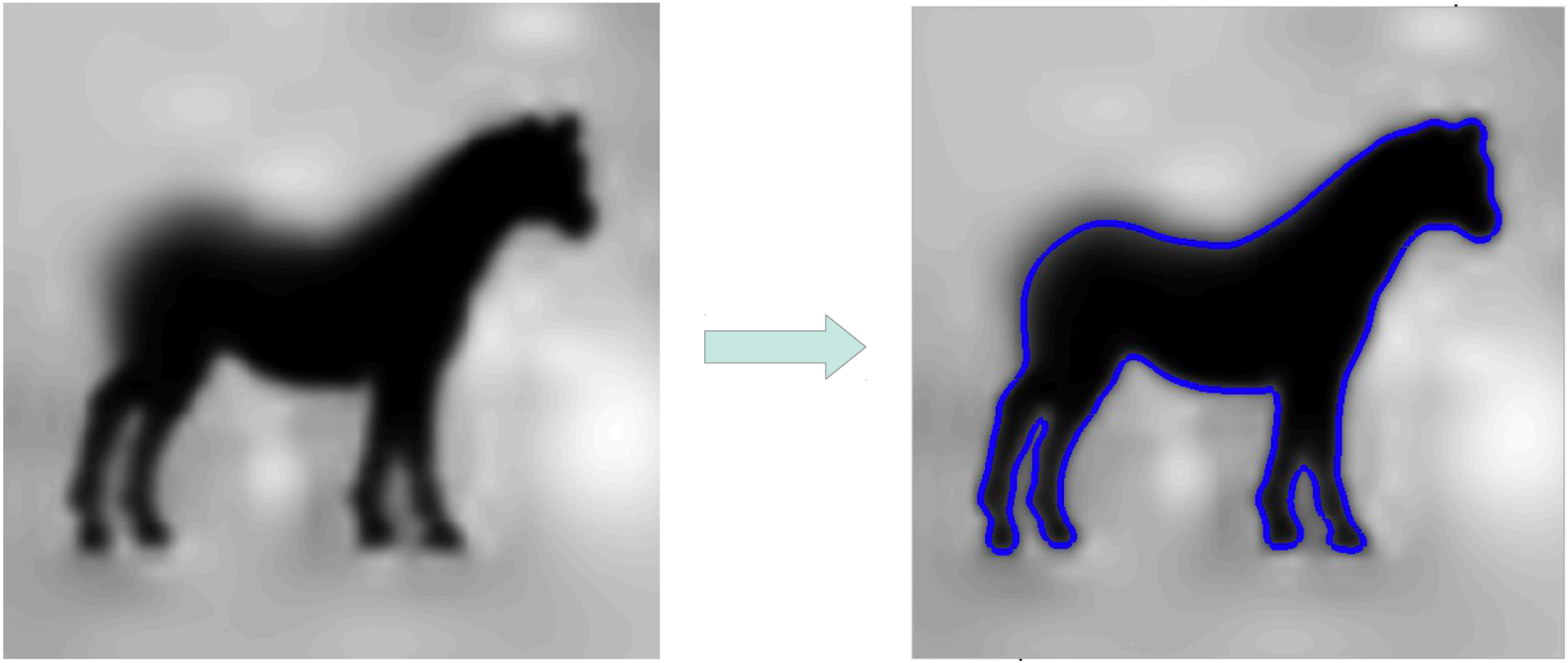


Schneider, Eberly, "Geometric Tools for Computer Graphics", Morgan Kaufmann, 2002
Sethian, "Level Set and Fast Marching Methods", Cambridge University Press, 1999

Conversion

- Explicit to Implicit
  - Compute signed distance at grid points
  - Compute distance point-mesh
  - Fast marching
- Implicit to Explicit (Polygonization of Implicit Surfaces)
  - Extract zero-level iso-surface $F(x, y, z) = 0$
  - Other iso-surfaces $F(x, y, z) = C$
  - Medical imaging, simulations, measurements, …

Recall: Final step of Poisson reconstruction

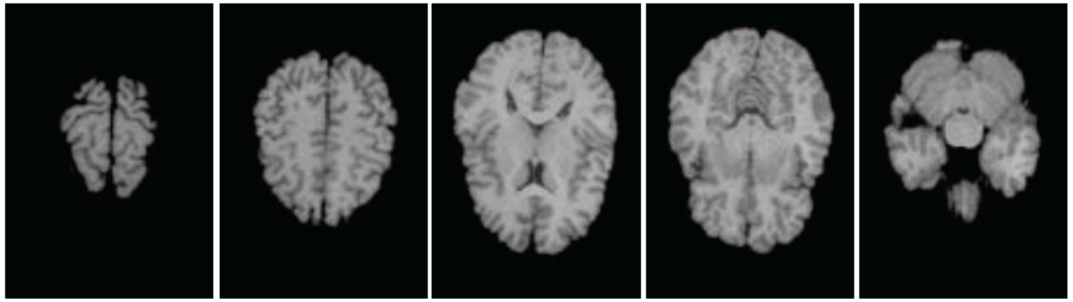


Density Function

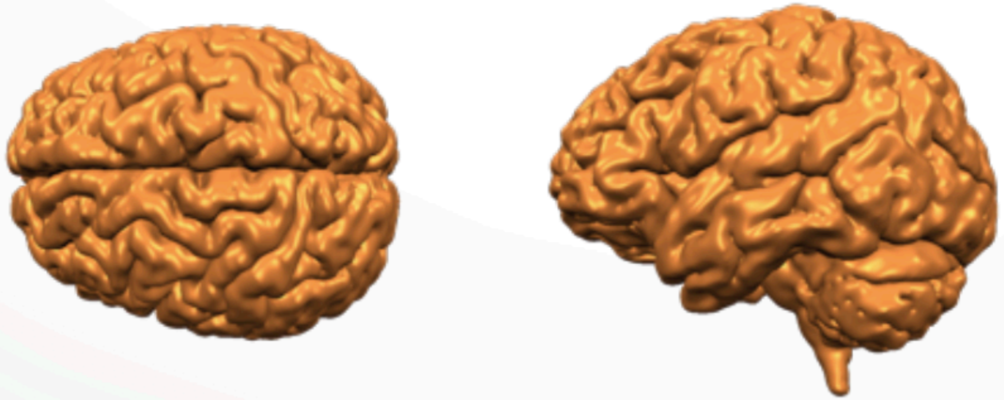
Isosurface

Medical Reconstruction

- Algorithm for isosurface extraction from
- medical scans (CT, MRI)

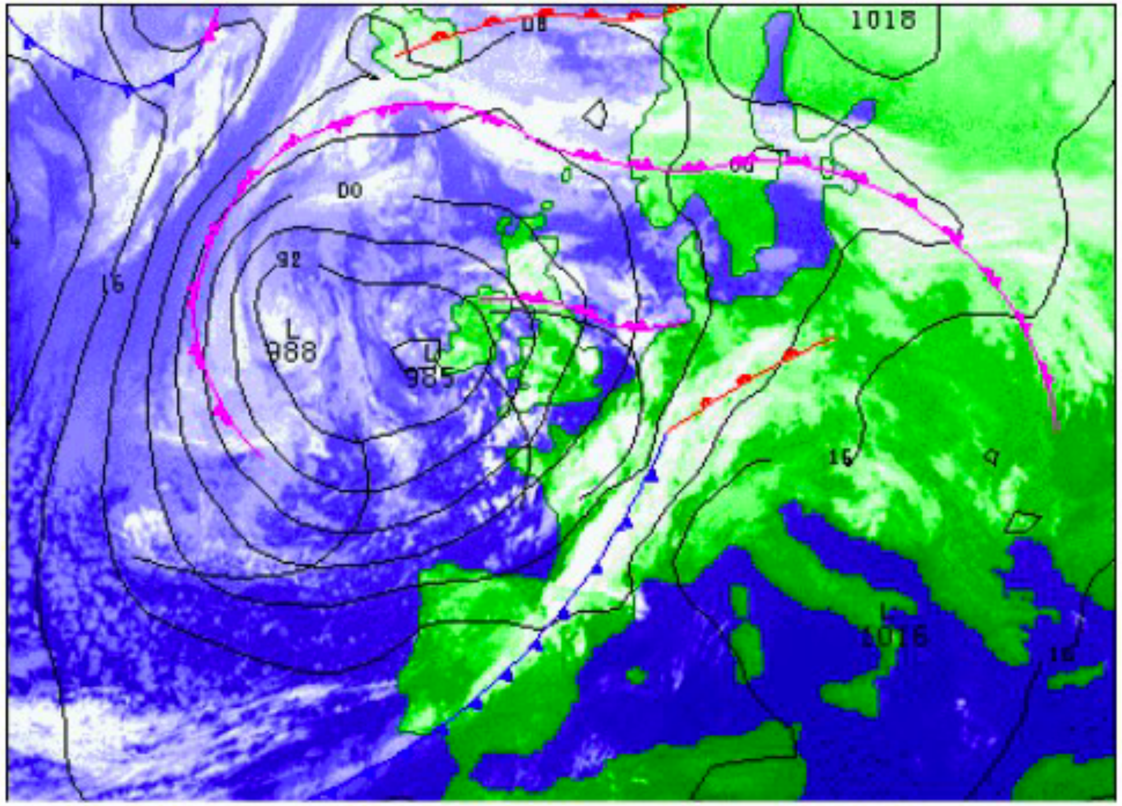


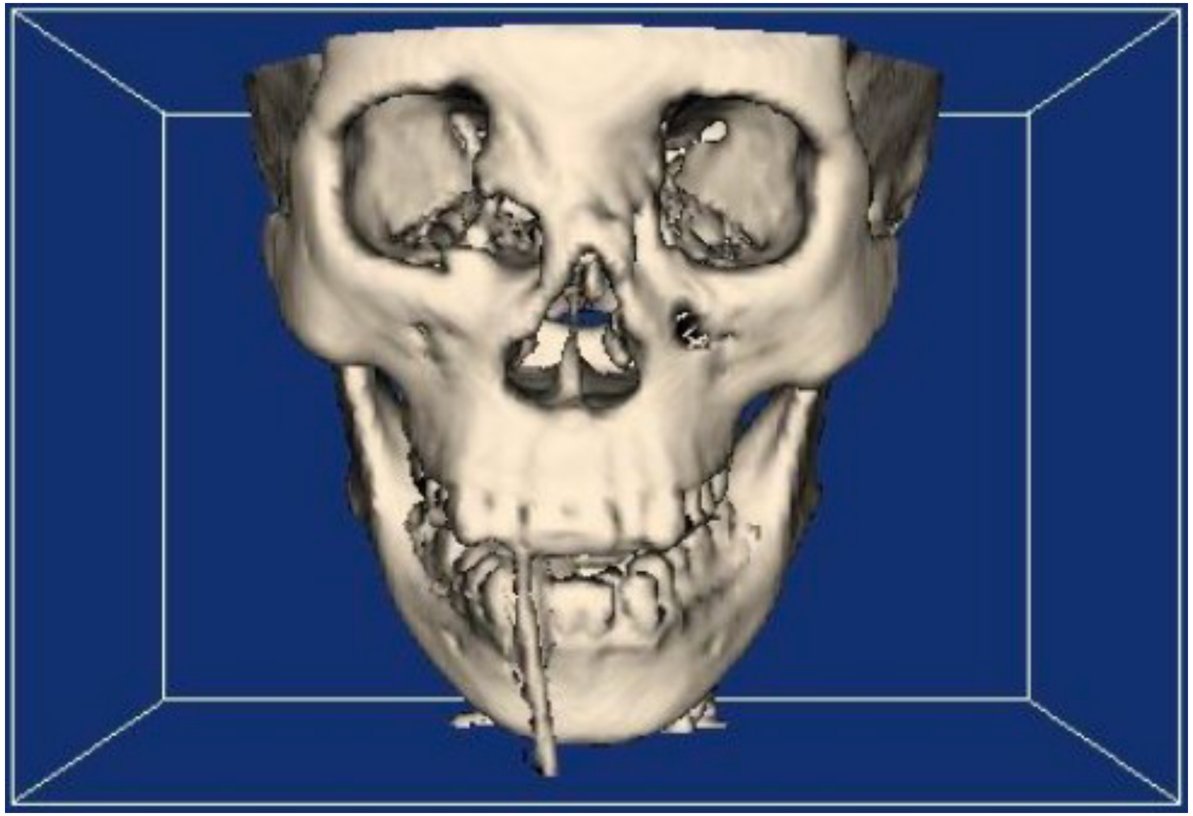
Density Function from MRI Scans



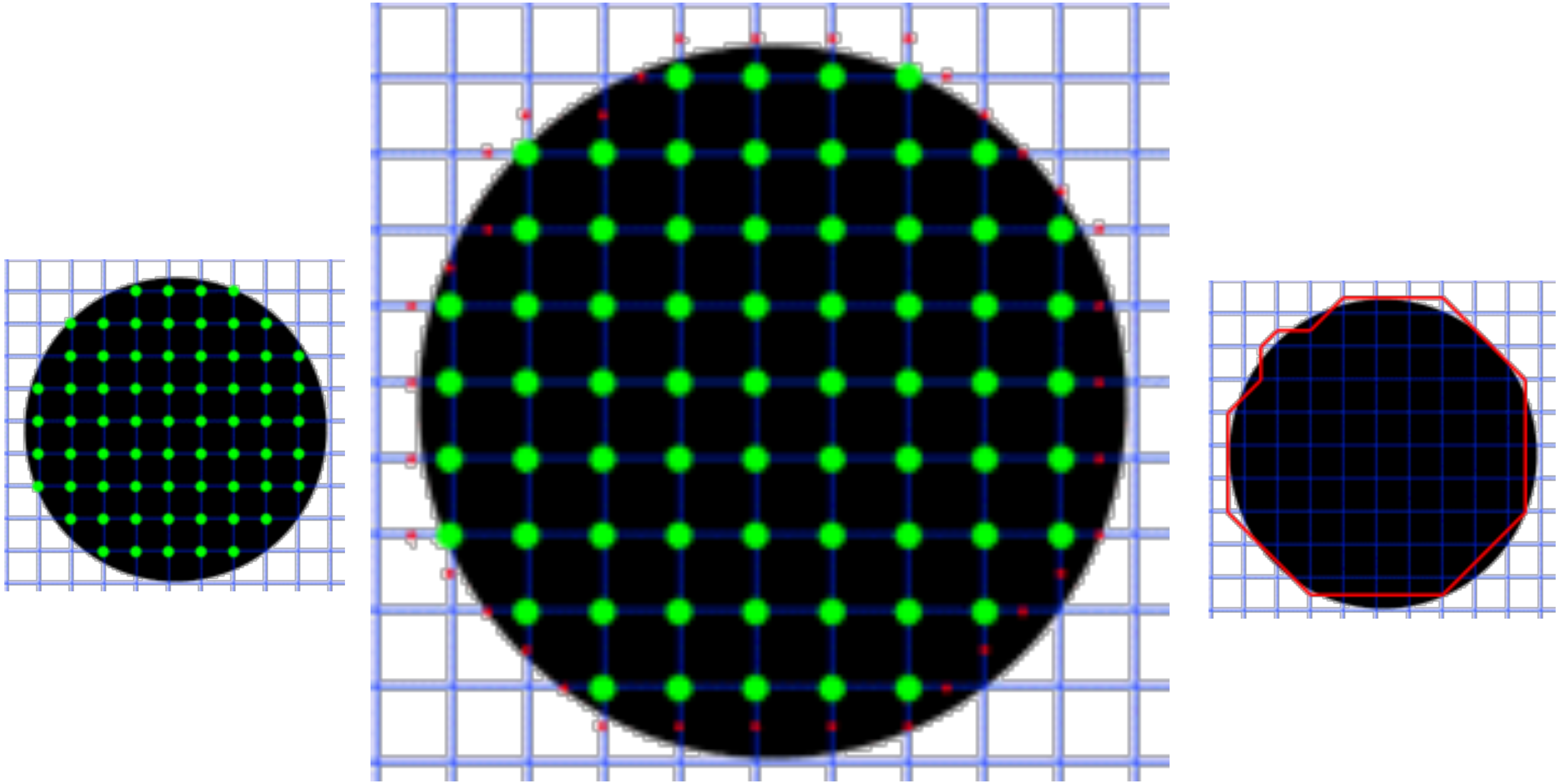
Level Set

- c-Level set: The set of points where a function takes a constant value c
  - Isocontour: Level set of a 2D function
  - Isosurface: Level set of a 3D function





Marching Squares

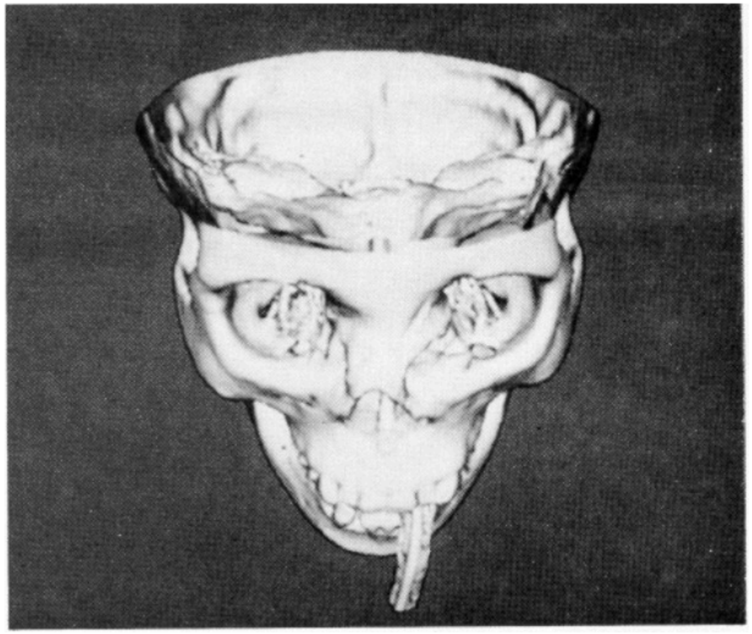


- Any red point is the midpoint of some edge!
- Resulting “circle” is bad

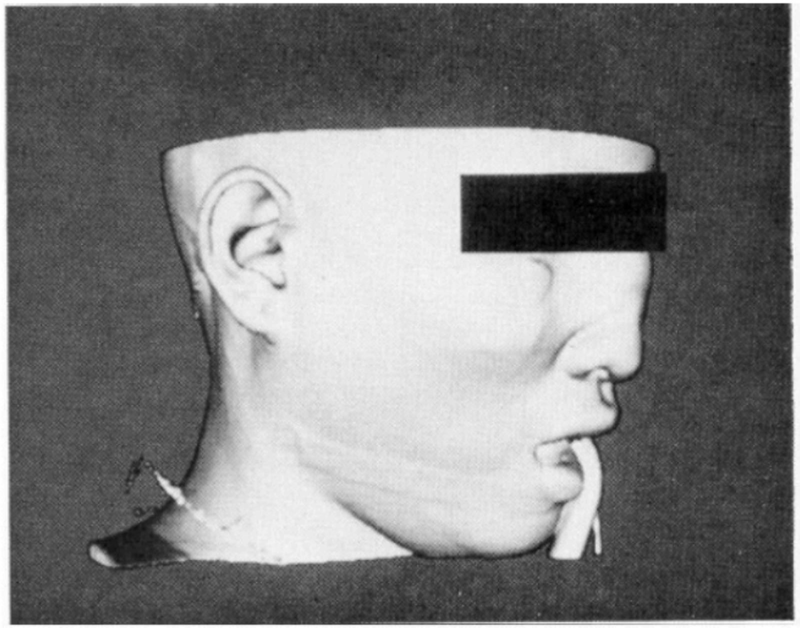
Marching Cubes

- Also known as
  - 3D Contouring / Tessellation of implicit surfaces
  - Polygonising a scalar field / Surface Reconstruction

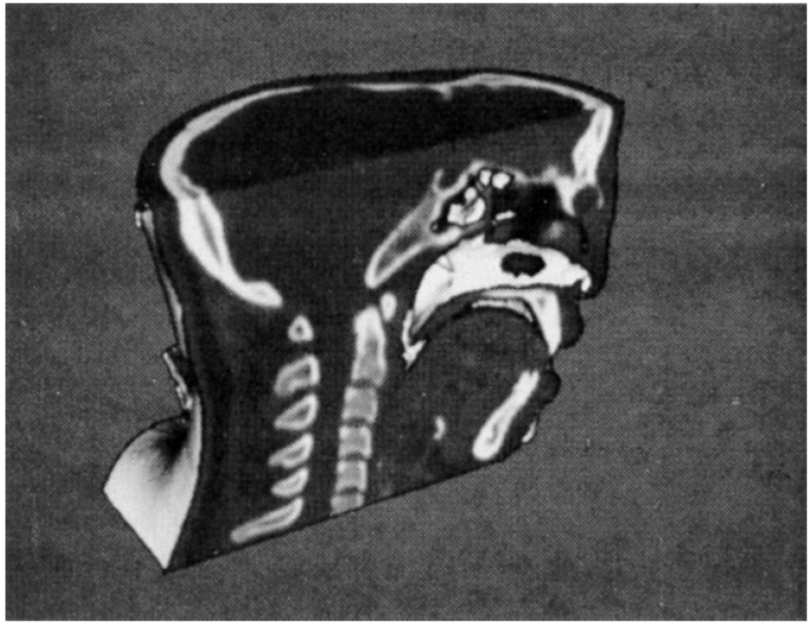
Different level sets of CT scan



Bone surface



Soft tissue surface

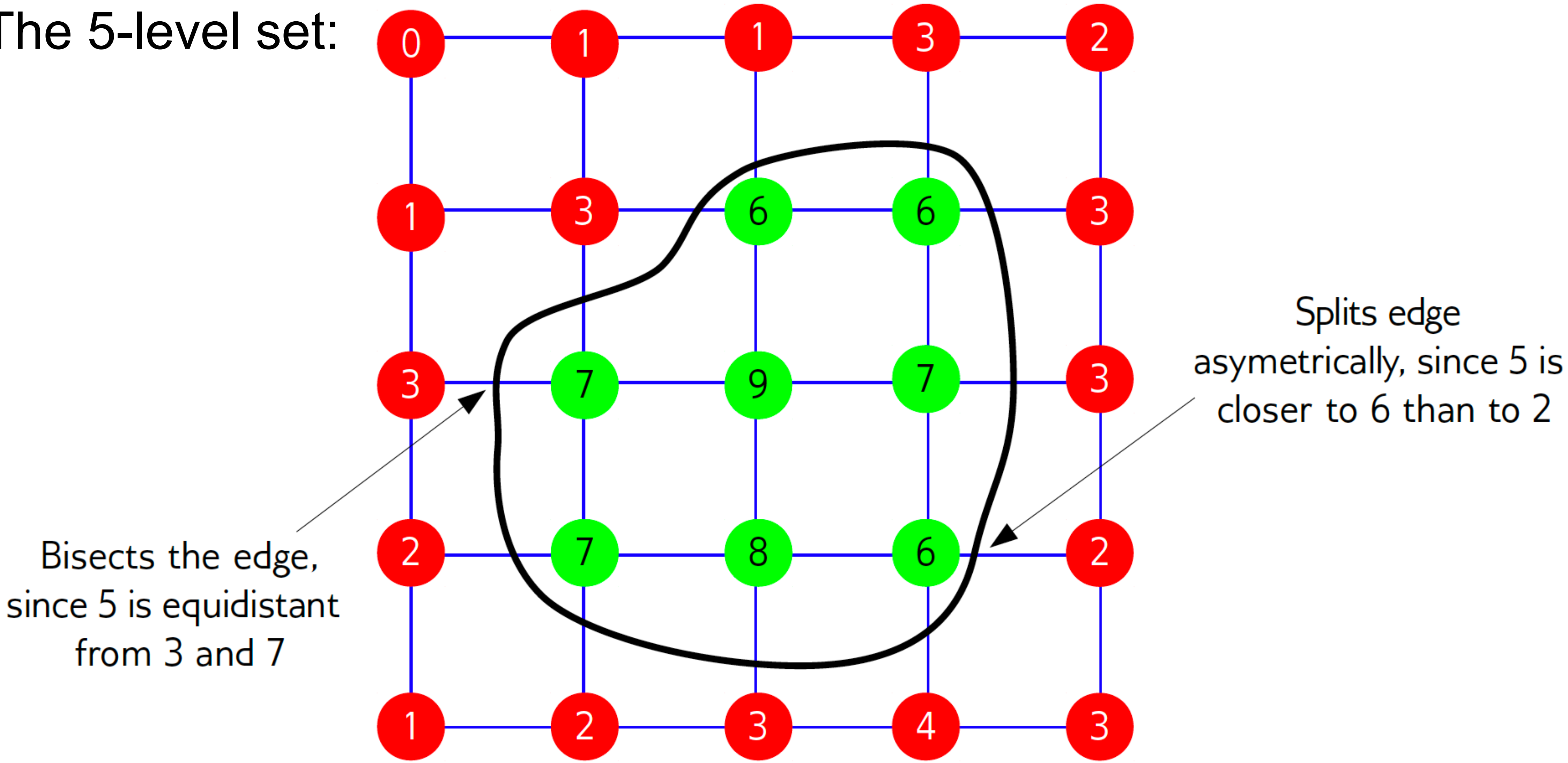


Alignment with original volumetric data

Lorensen and Cline, “Marching Cubes: A High Resolution 3D Surface Reconstruction Algorithm”, SIGGRAPH '87

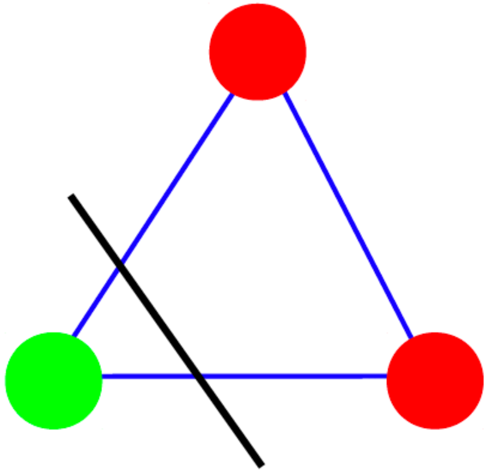
Marching Square

- The 5-level set:

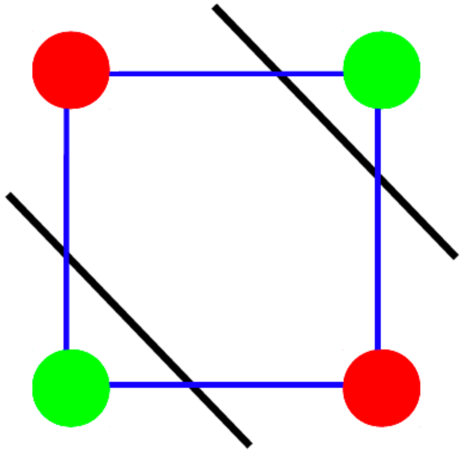


Isocontours: Ambiguity

- Where is the contour?



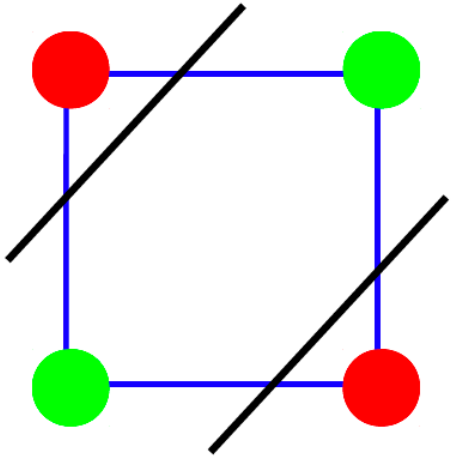
Triangular cell:
No ambiguities



“Split” green (inner) region

or

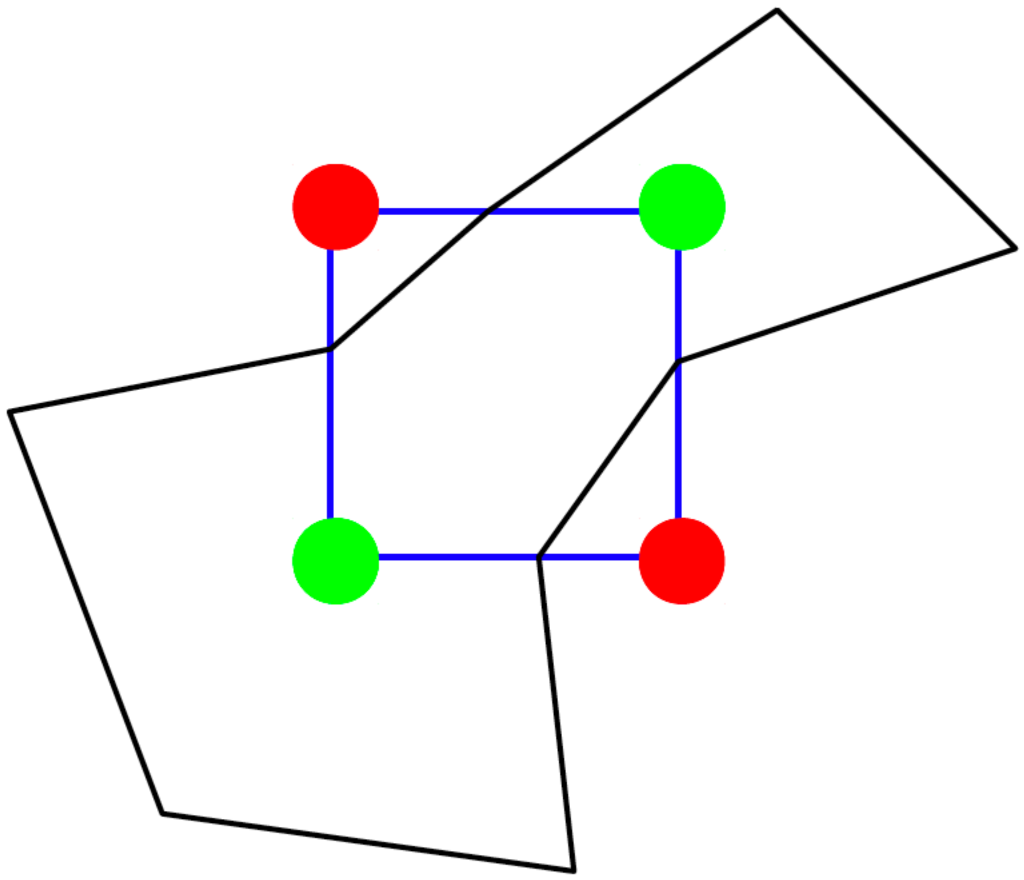
Square cell:
2 ambiguous cases



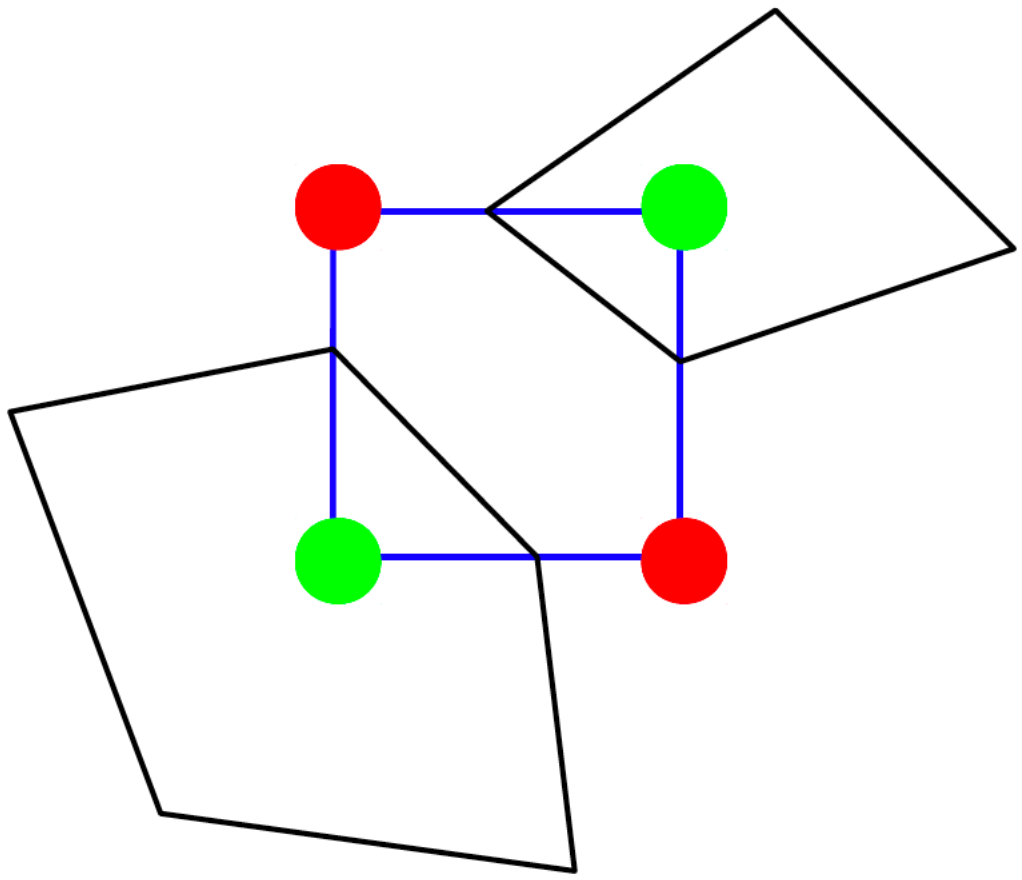
“Join” green (inner) region

Isocontours: Ambiguity

- Where is the contour?



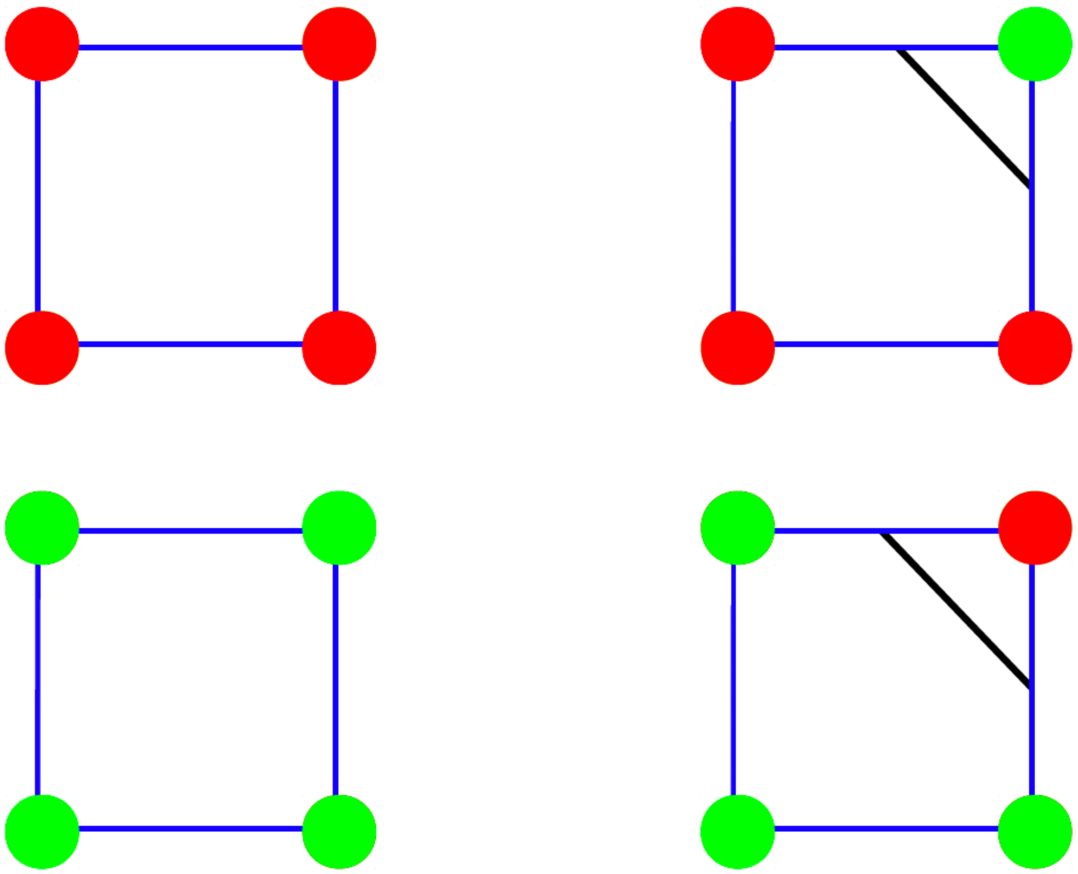
Join

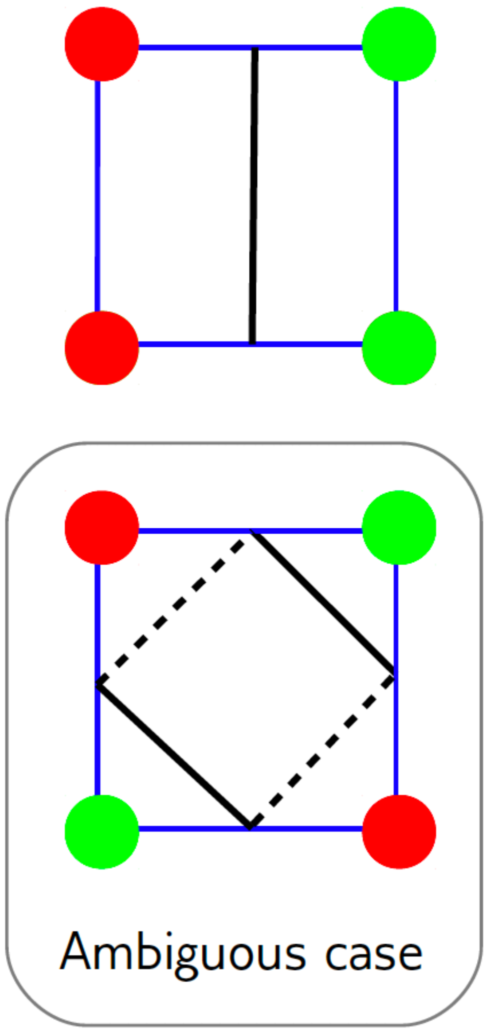


Split

Isocontours: Cell Configurations

No intersections | 1 vertex different | 2 vertices different





$2^4 = 16$ different possibilities, reducible to just 6 distinct cases after factoring out symmetries

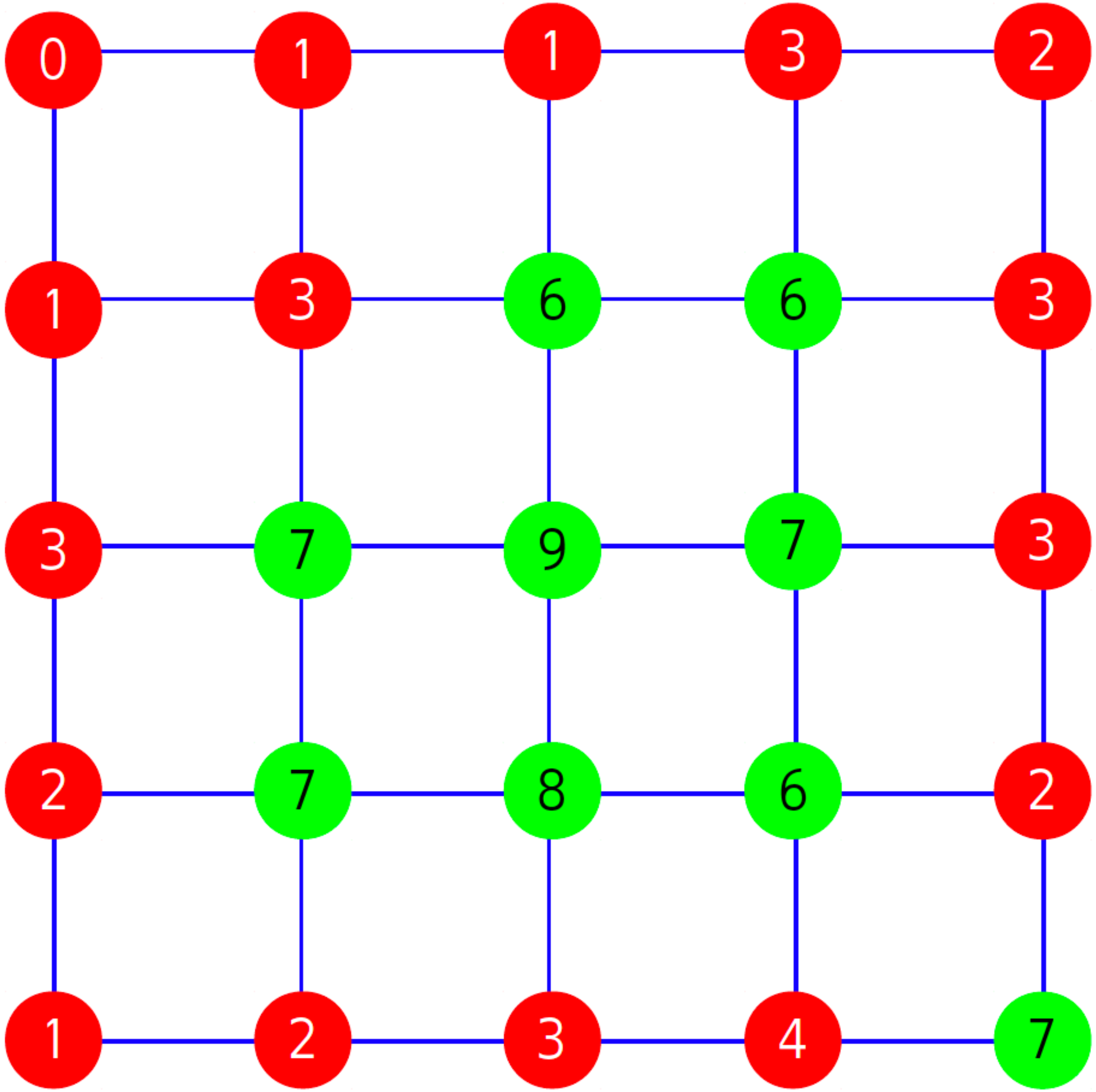
Marching Squares Algorithm

- Select a starting cell
- Calculate inside/outside state for each vertex
- Classify cell configuration
  - Determine which edges are intersected
- Find exact locations of edge intersections
- Link up intersections to produce contour segment(s)
- Move (or “march”) into next cell and repeat
  - … until all cells have been visited

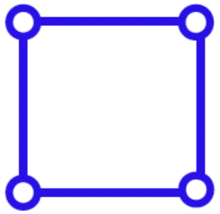
Example : Contour Line Generation

- Find 5-contour of function represented by its values at vertices of a uniform grid

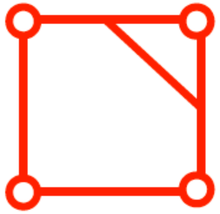
- Step 1: Classify vertices



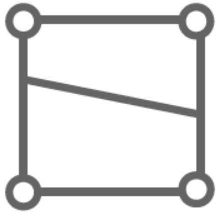
Step 2 : classify cells



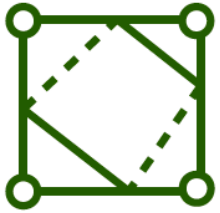
No intersections



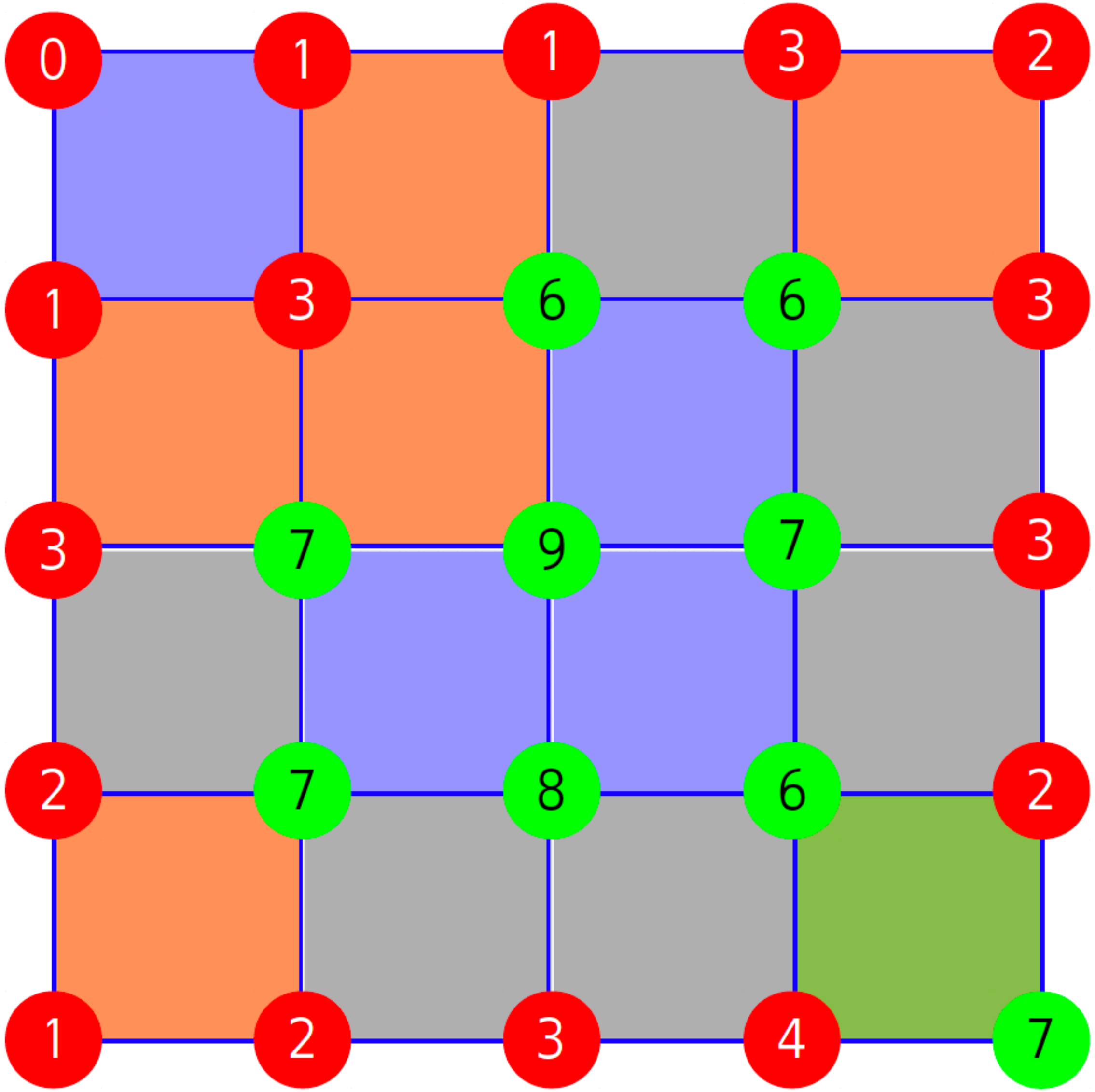
Adjacent edges



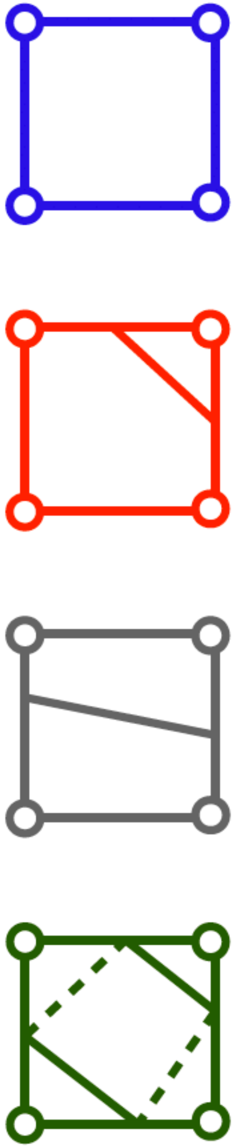
Opposite edges



Ambiguous



Step 3 : interpolate contour intersections

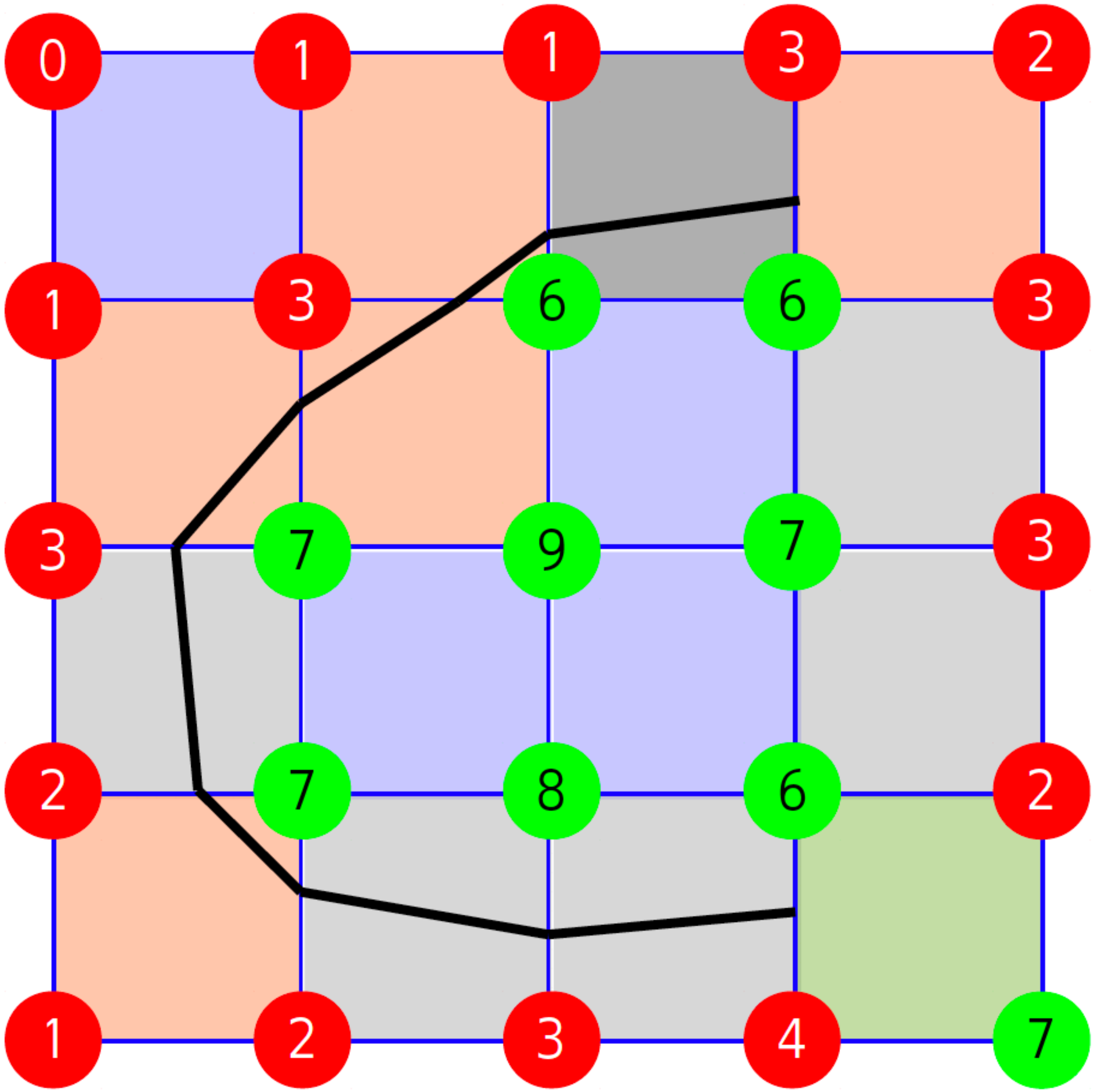


No intersections

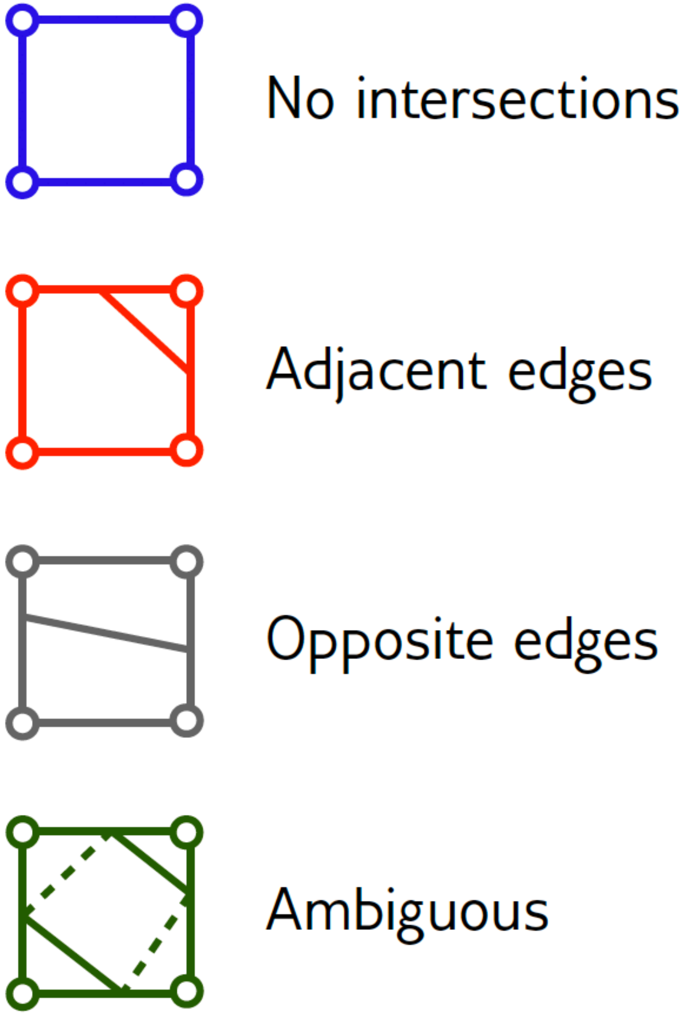
Adjacent edges

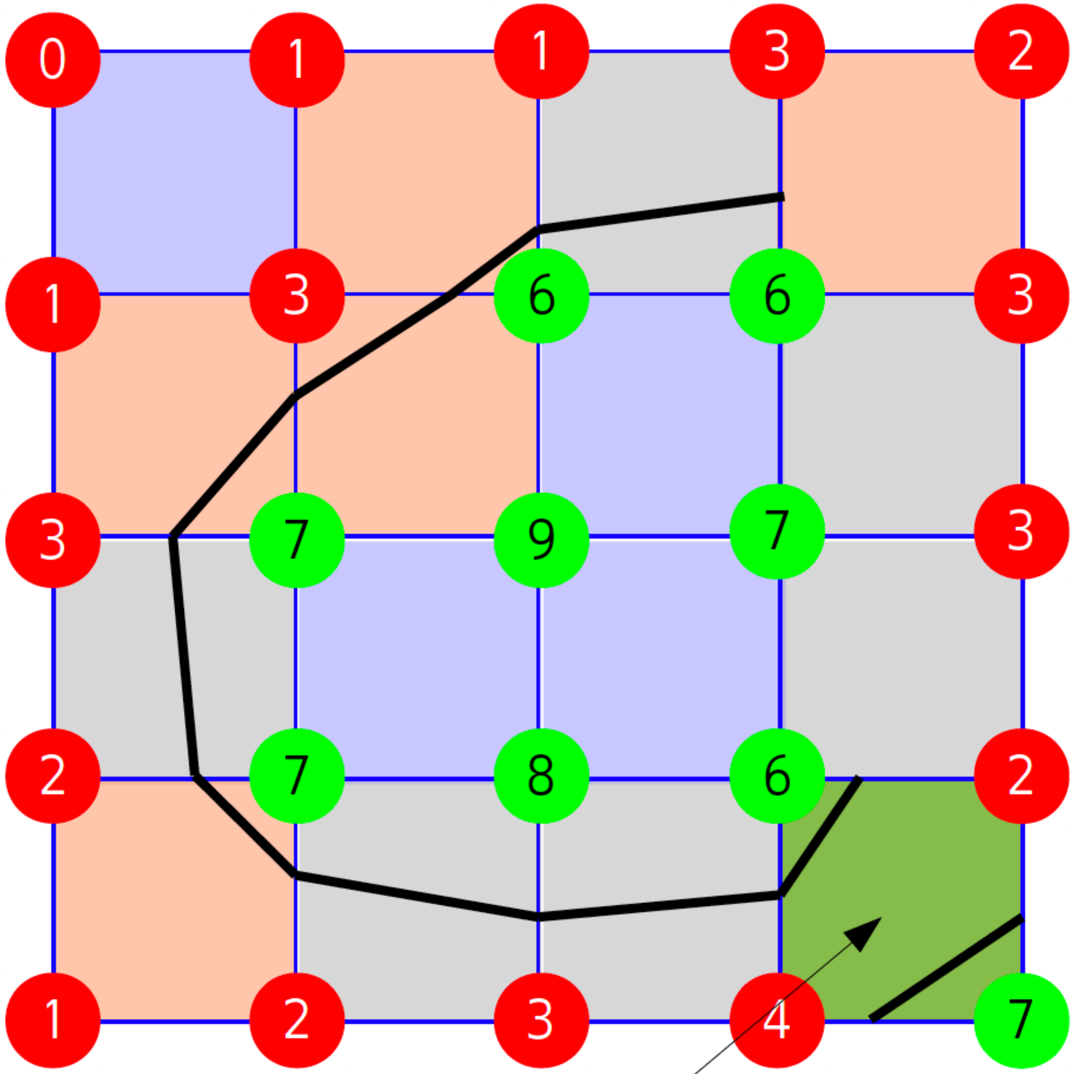
Opposite edges

Ambiguous



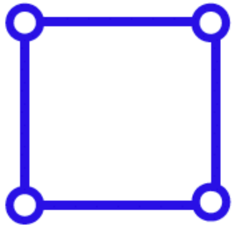
Step 3 : interpolate contour intersections



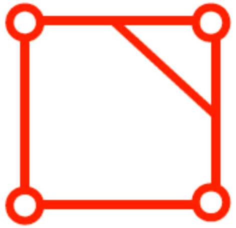


Arbitrarily choose to split here, instead of join. We could also have gone the other way.

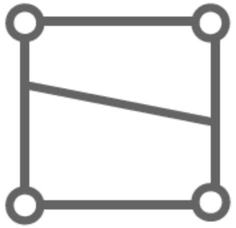
Step 3 : interpolate contour intersections



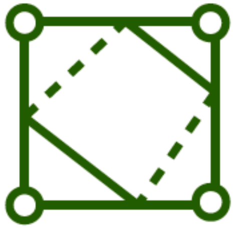
No intersections



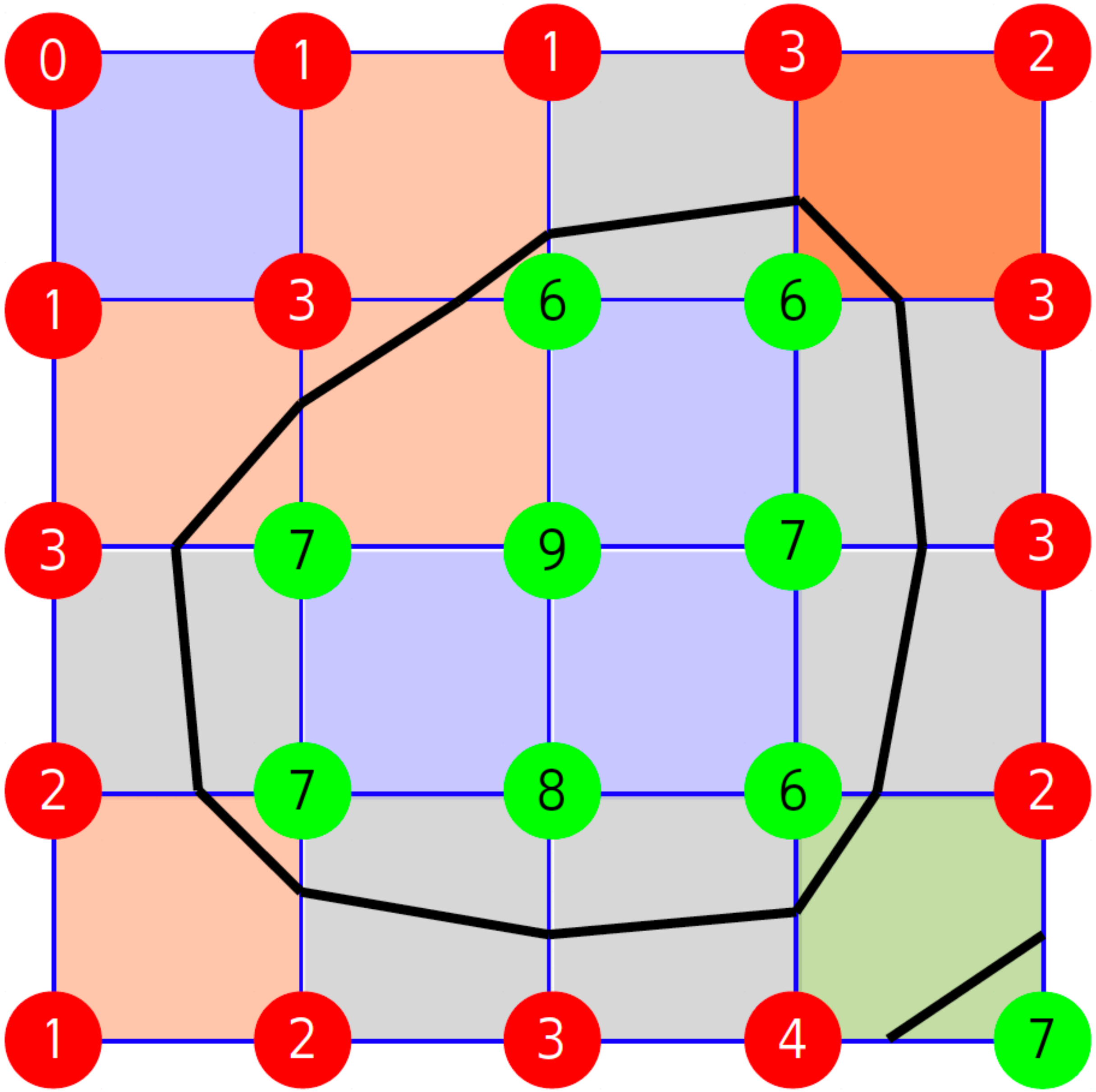
Adjacent edges



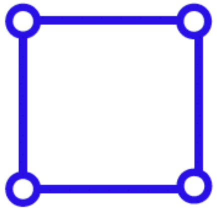
Opposite edges



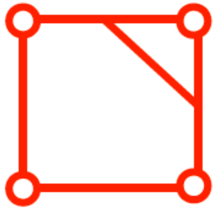
Ambiguous



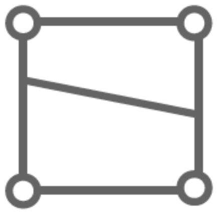
Resolving ambiguities



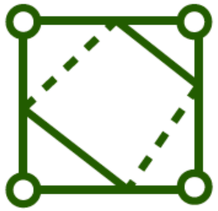
No intersections



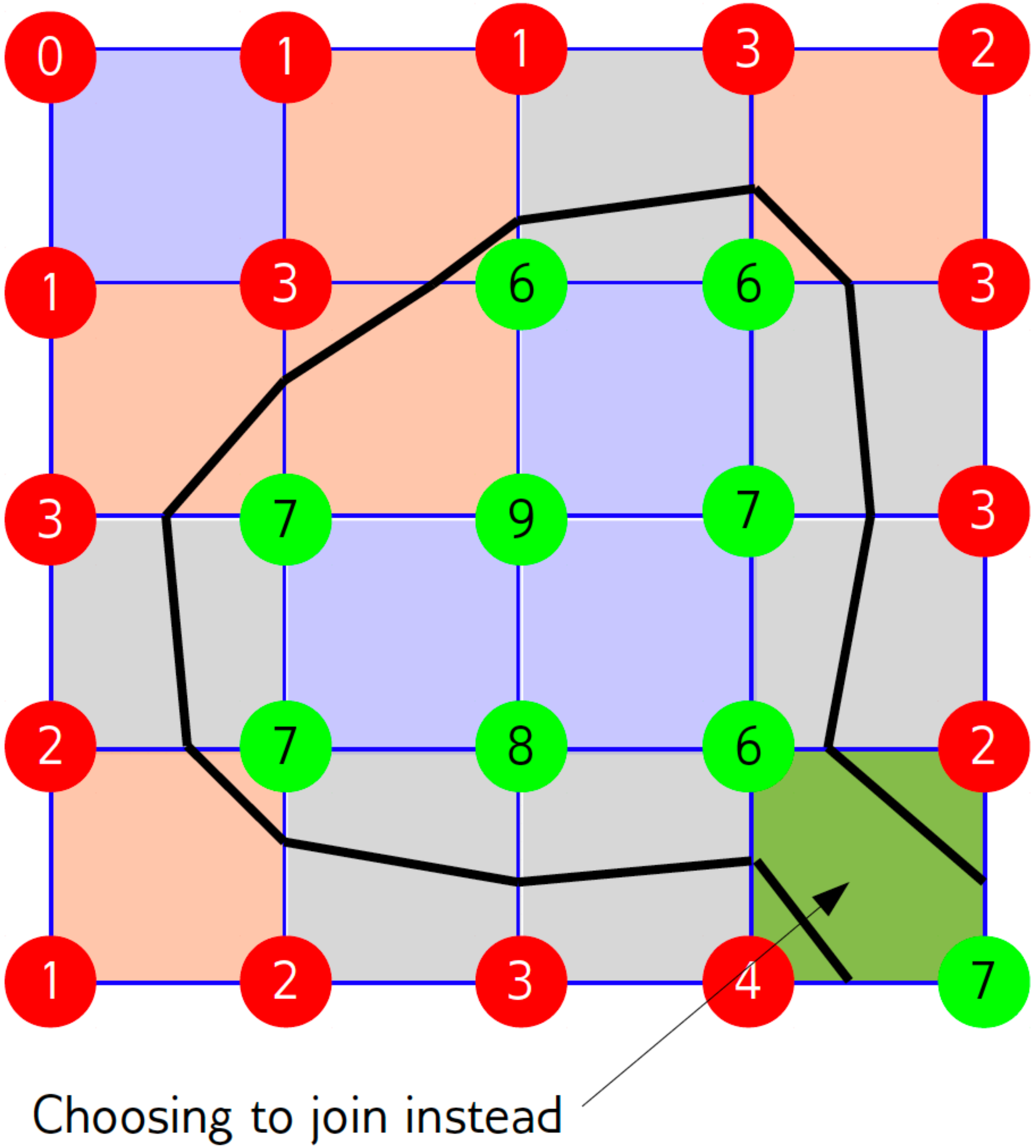
Adjacent edges



Opposite edges

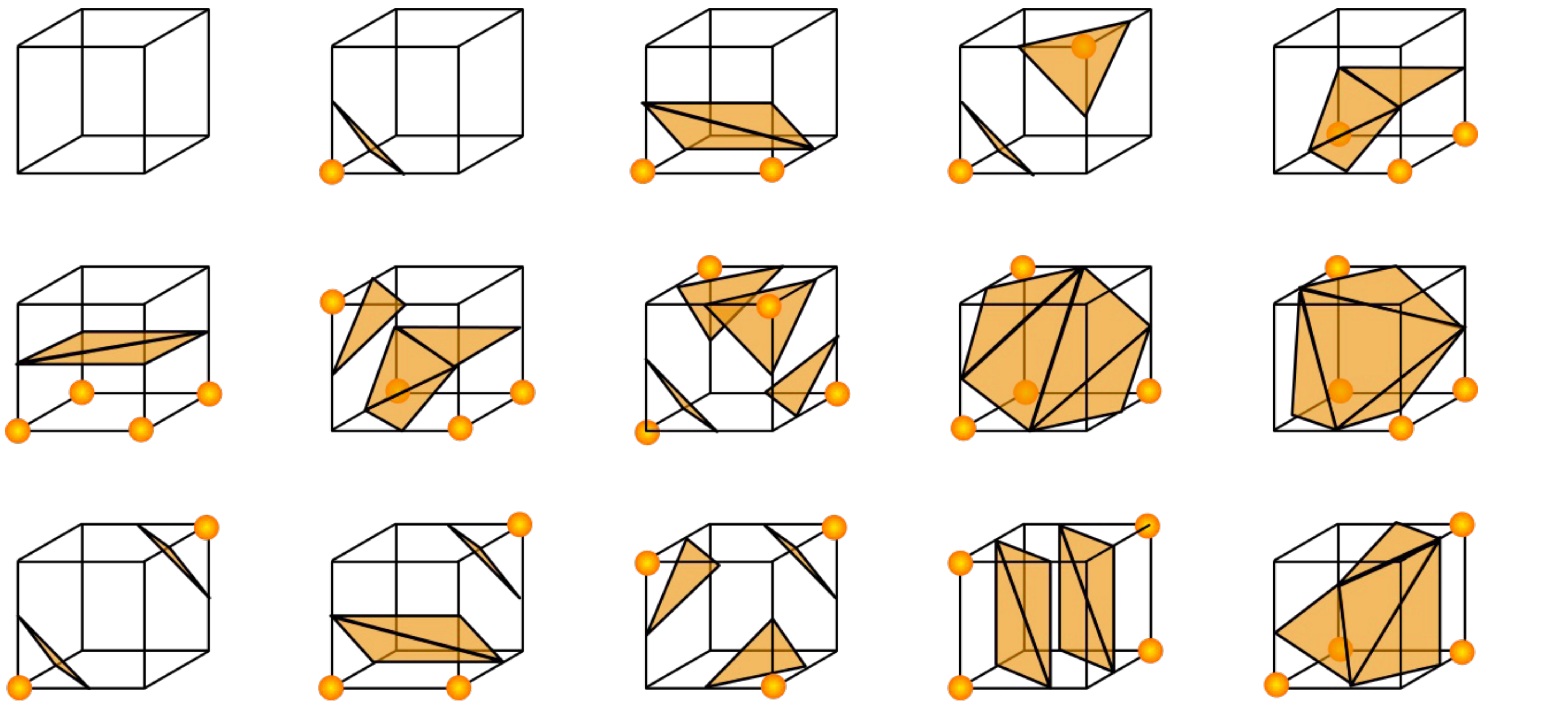


Ambiguous



In 3D: Marching Cubes

- Exactly the same algorithm, but cells are now cubes (15 distinct configurations) and output is triangles (or a polygon mix)



Montani et al., “A modified look-up table for implicit disambiguation of Marching Cubes”, Visual Computer 1994

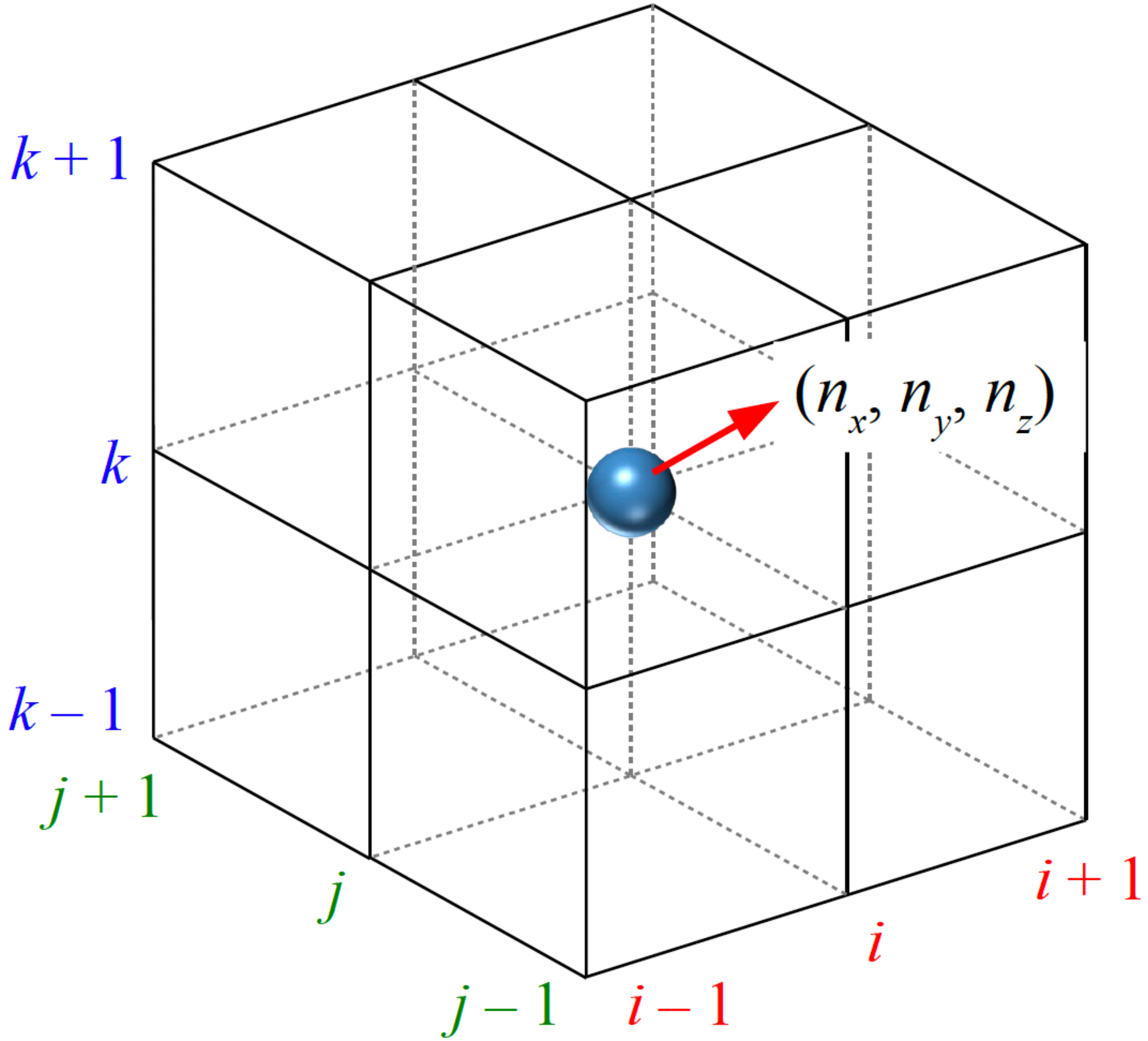
Marching Cubes: Estimating Normals

- We could estimate normals from the generated mesh, but the density function has more information
- Recall: The normal to the surface is the gradient of the density function

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)$$

- We will estimate the gradient from the grid of values

Normals at Cube Vertices



Discrete approximation to the gradient at the blue cube vertex

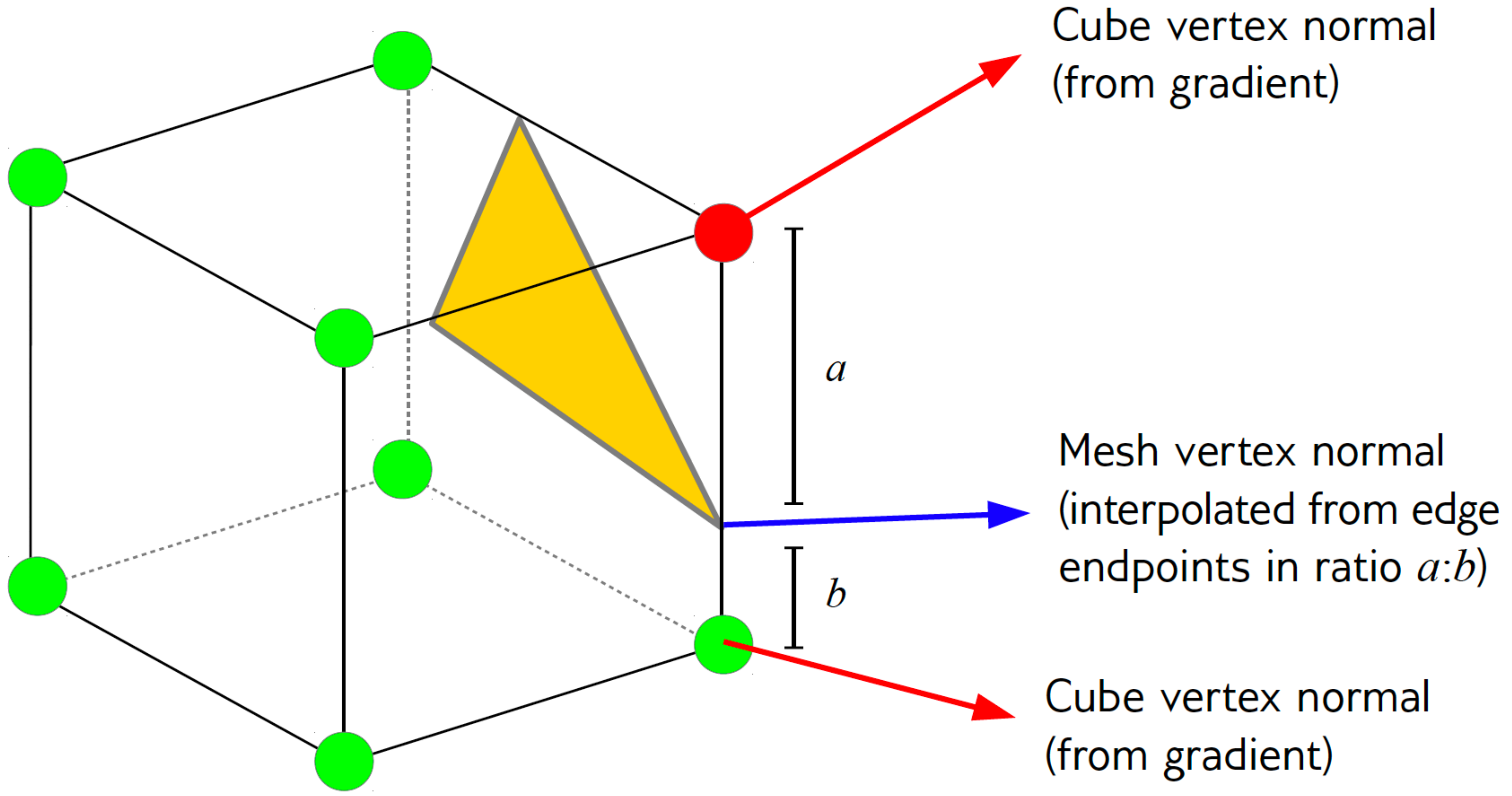
$$n_x = \frac{f(i+1, j, k) - f(i-1, j, k)}{2\Delta x}$$

$$n_x = \frac{f(i, j+1, k) - f(i, j-1, k)}{2\Delta y}$$

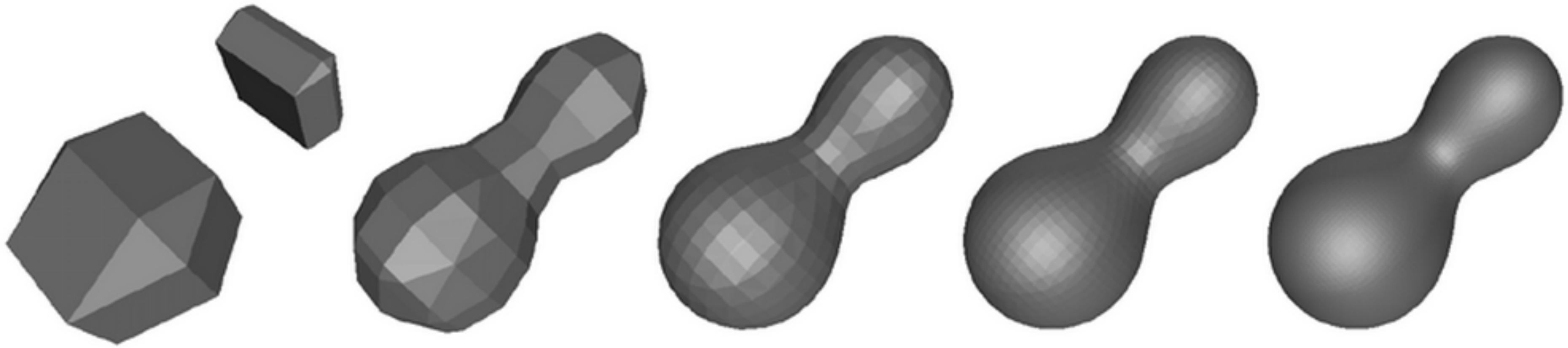
$$n_y = \frac{f(i, j, k+1) - f(i, j, k-1)}{2\Delta z}$$

(Better approximations are possible)

Normals at Mesh Vertices



Grid Resolution



Grid size=10
70 Facets

Grid size=5
220 Facets

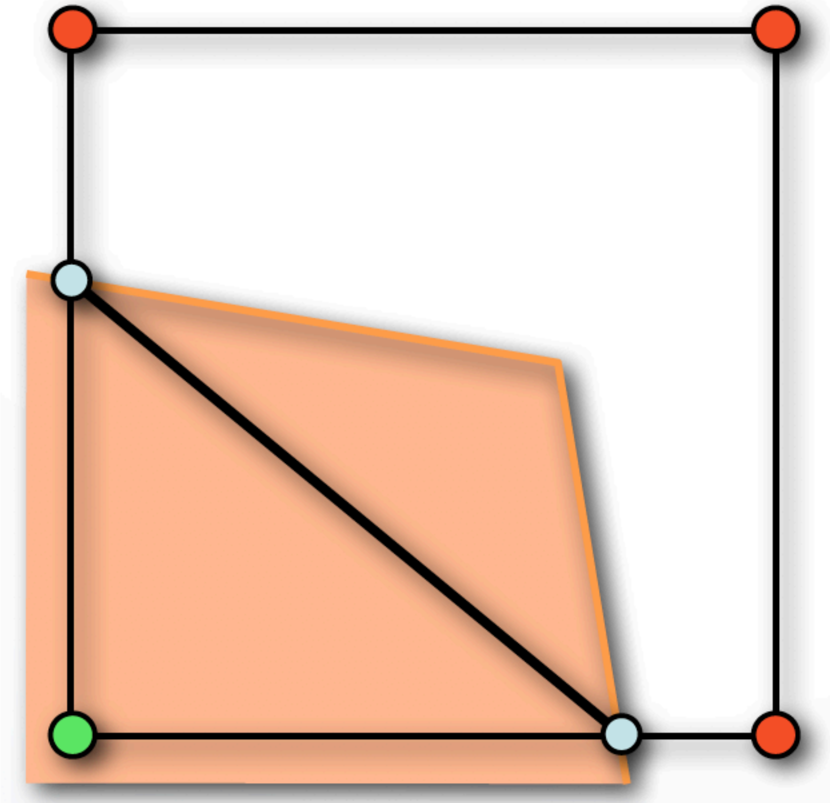
Grid size=2
1700 Facets

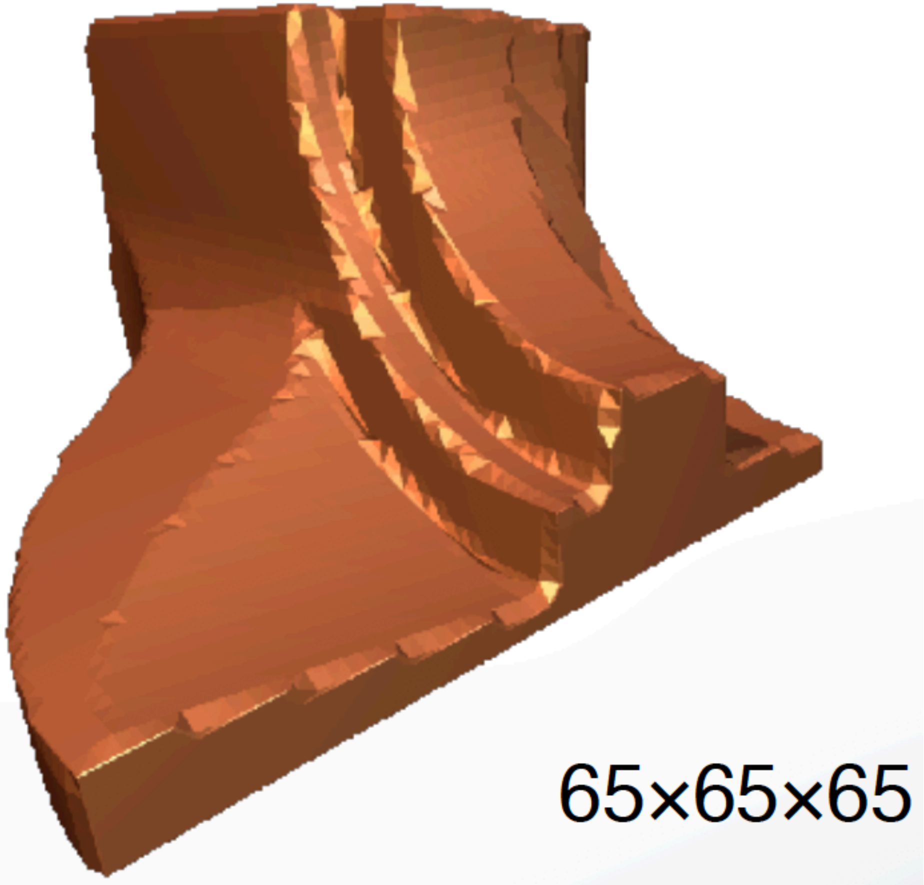
Grid size=1
6800 Facets

Grid size=0.5
27000 Facets

Marching Cubes

- Sample points restricted to edges of regular grid
- Alias artifacts at sharp features



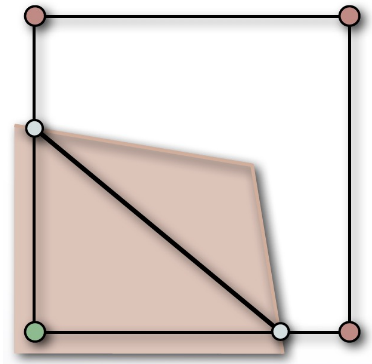


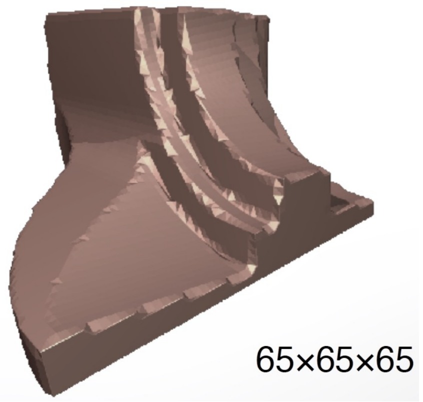
Increasing Resolution

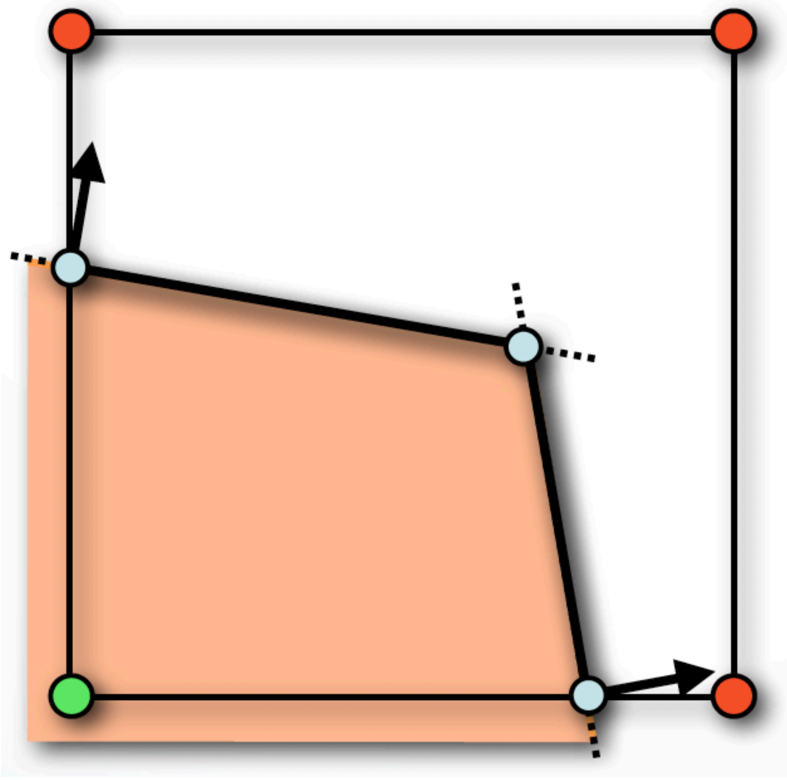
Does not remove alias problems!

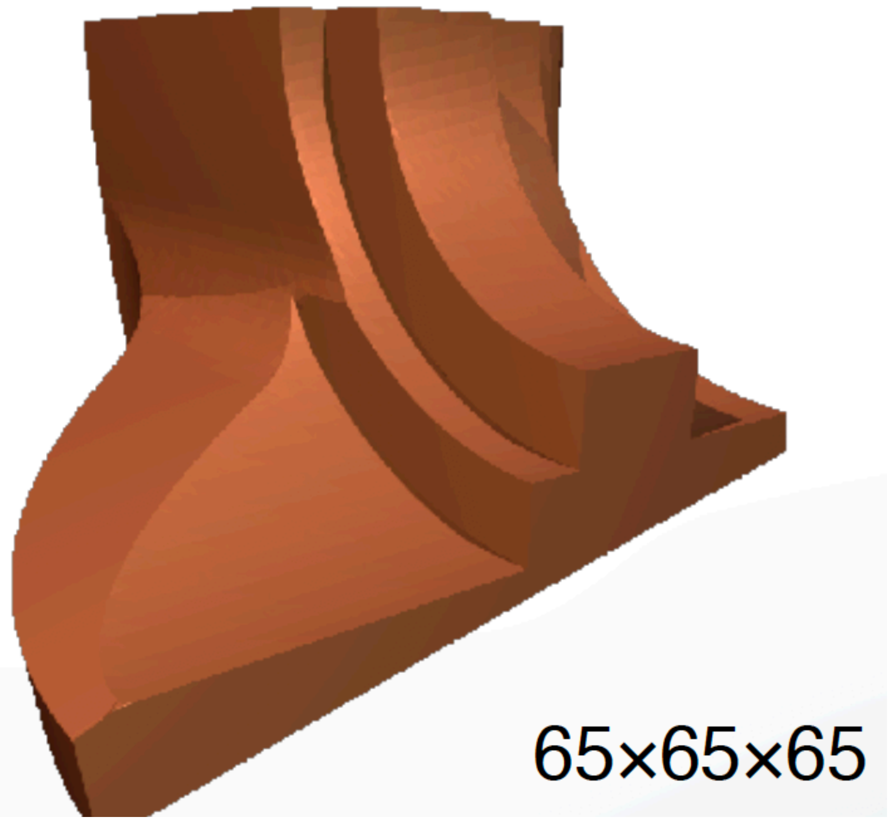
Extended Marching Cubes

- Locally extrapolate distance gradient
- Place samples on estimated features





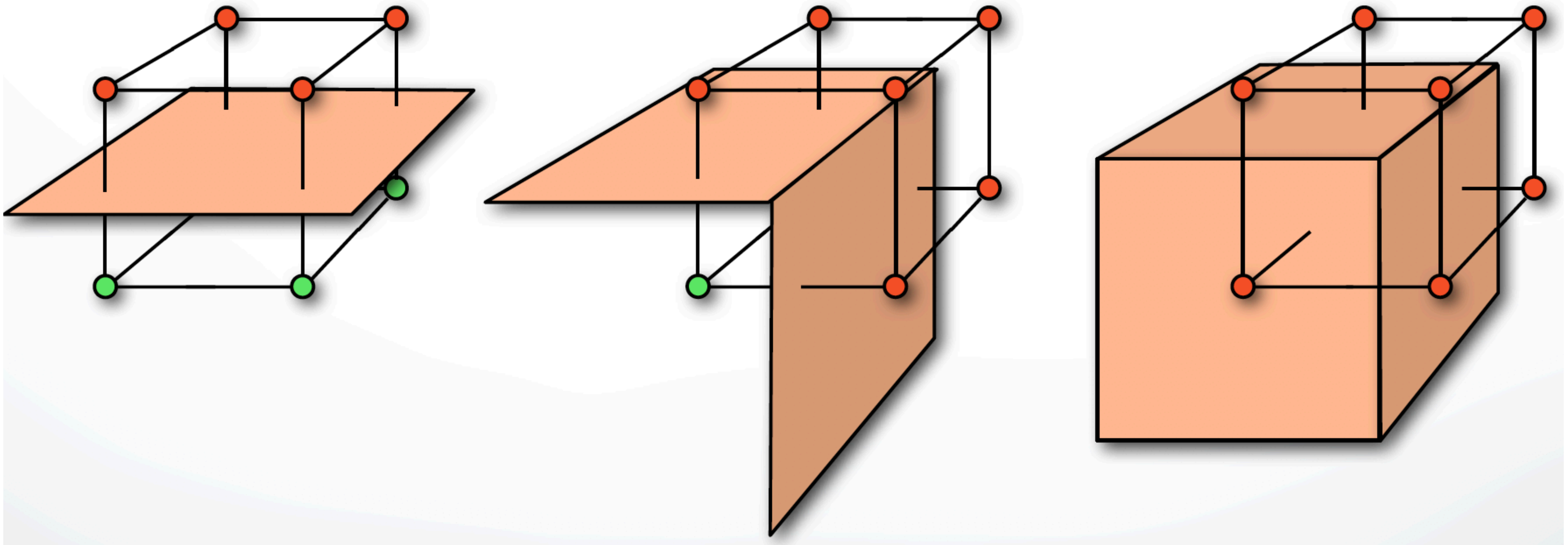




Kobbelt et al., “Feature Sensitive Surface Extraction from Volume Data”, SIGGRAPH 2001

Extended Marching Cubes

- Feature detection
  - Based on angle between normals
  - Classify into edges / corners



Extended Marching Cubes

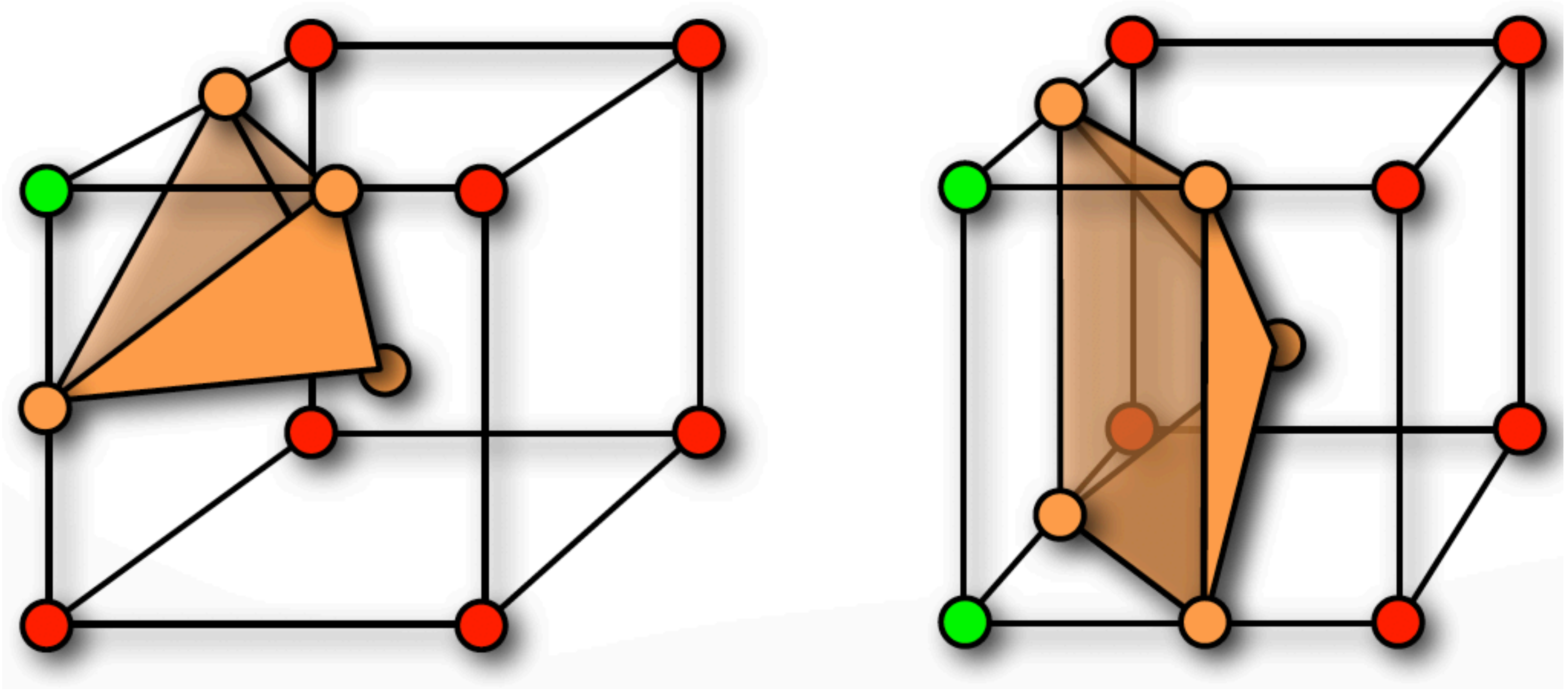
- Feature sampling
- Intersect tangent planes (s_i, n_i)

$$\begin{pmatrix} \vdots \\ \mathbf{n}_i \\ \vdots \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \vdots \\ \mathbf{n}_i^T \mathbf{s}_i \\ \vdots \end{pmatrix}$$

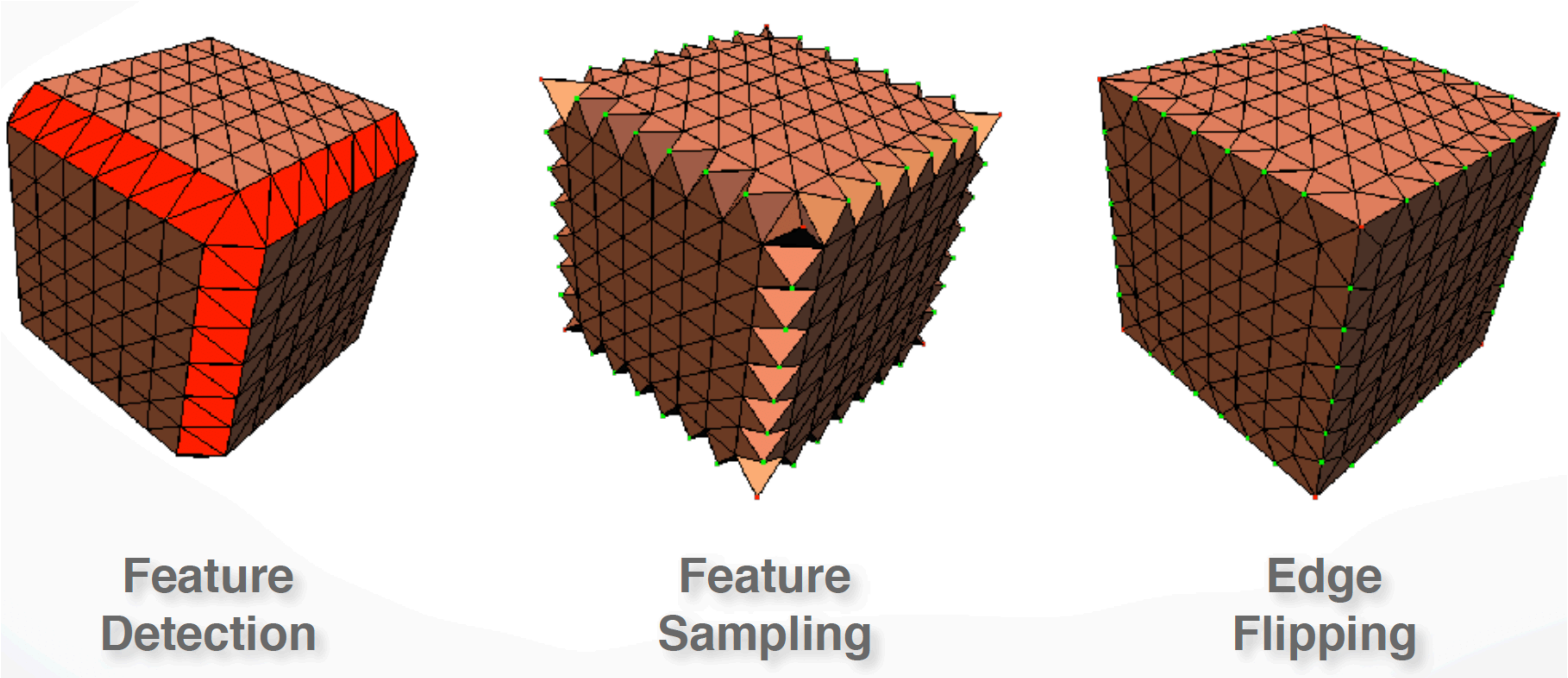
- Over- or under-determined system
- Solve by SVD pseudo-inverse

Extended Marching Cubes

- Feature sampling
- Intersect tangent planes (s_i, n_i)
- Triangle fans centered at feature point



Extended Marching Cubes

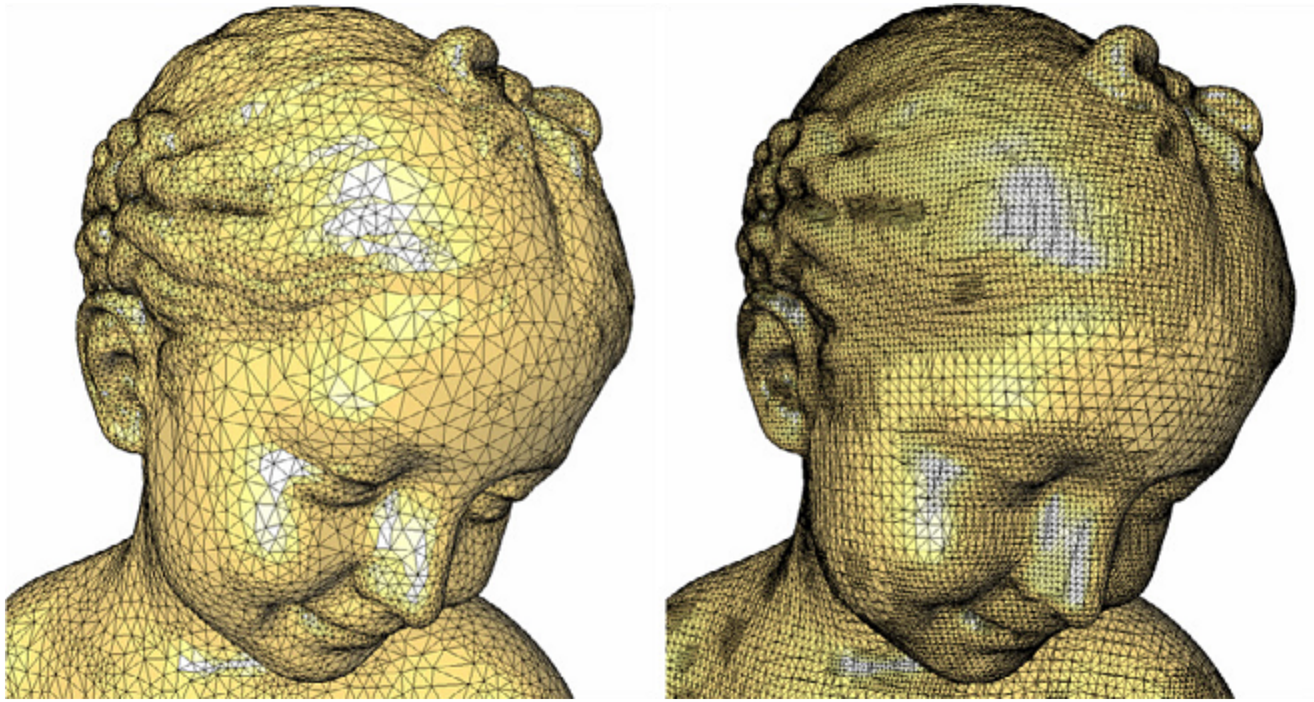


Marching Cubes: Pros and Cons

- Pros:
  - Local computations only, so needs very little working memory & easy to parallelize
  - Simple to implement
- Cons:
  - produces lots of triangles (-> mesh decimation)
  - Degenerate triangles (→ remeshing)
  - No principled approach to resolve ambiguities
  - MC does not preserve features
- EMC preserves features, but…
  - about 10% more triangles
  - 20-40% computational overhead

Cons & Pros

- Even more intelligent forms of marching cubes, which **adapt their cube resolution** to match local surface complexity, produces **pretty low quality** meshes.



- The right mesh was made with adaptive marching cubes while the left mesh was made with a much more advanced algorithm (see Voronoi-based Variational Reconstruction of Unoriented Point Sets).
- Nevertheless marching cubes is useful for its **simplicity**. Implicit functions occur a lot in computer graphics and other fields, and rendering them is often the most intuitive way to work with them

Hex ⇔ Binary

- Hex in c/c++: begin with 0x
- Hex ⇔ Binary
  - 0x000 ⇔ 0000 0000 0000 (12 zeros)
  - 0xfff ⇔ 1111 1111 1111

- Bitwise Inclusive OR Operator: |

```
unsigned short a = 0x5555; // pattern 0101 ...
unsigned short b = 0xAAAA; // pattern 1010 ...
cout << hex << ( a | b ) << endl; // prints "ffff" pattern 1111 ...
```

- Bitwise Exclusive OR Operator: ^

```
unsigned short a = 0x5555; // pattern 0101 ...
unsigned short b = 0xFFFF; // pattern 1111 ...
cout << hex << ( a ^ b ) << endl; // prints "aaaa" pattern 1010 ...
```

Vertex States

- For each of the 8 vertices: either inside or outside of the surface. So 2^8=256 possible vertex states
- 2 of these are trivial, where all points are inside or outside
- account for **symmetries**, there are really only **14 unique** configurations in the remaining 254 possibilities.
- A 8 bit index is formed where each bit corresponds to a vertex state
  - only vertex 3 was **below** the isosurface, cubeindex would equal 0000 1000 or 8.

```
cubeindex = 0;
if (grid.val[0] < isolevel) cubeindex |= 1;
if (grid.val[1] < isolevel) cubeindex |= 2;
if (grid.val[2] < isolevel) cubeindex |= 4;
if (grid.val[3] < isolevel) cubeindex |= 8;
if (grid.val[4] < isolevel) cubeindex |= 16;
if (grid.val[5] < isolevel) cubeindex |= 32;
if (grid.val[6] < isolevel) cubeindex |= 64;
if (grid.val[7] < isolevel) cubeindex |= 128;
```

Edge Intersection State Table

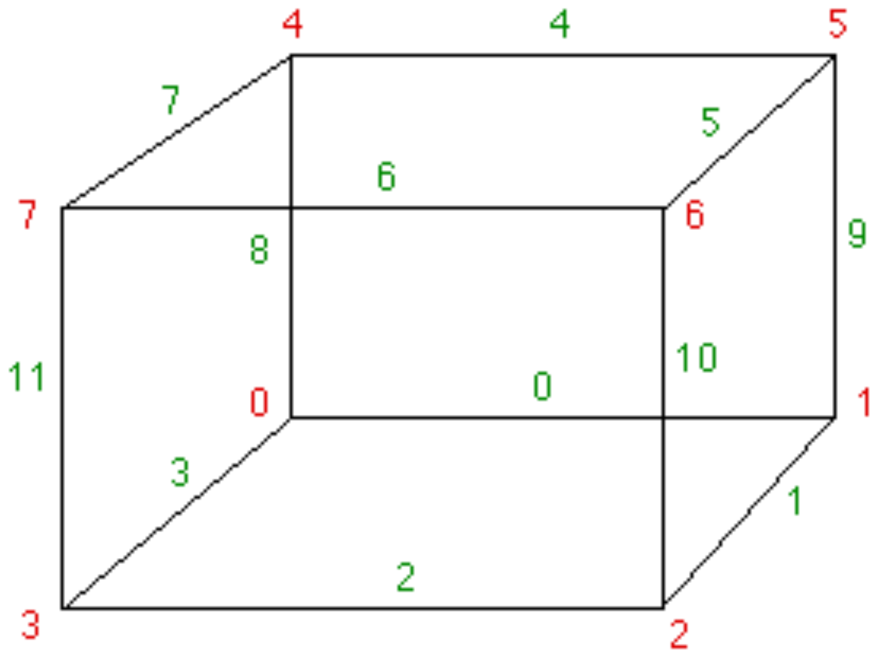
- **Vertex states is index of Edge intersection state table**
  - For any edge, if one vertex is inside of the surface and the other is outside of the surface then the edge **intersects** the surface
  - There are 12 edges. For each entry in the table, **if edge #n is intersected, then bit #n is set to 1**

```
//0x000 ⇔ 000000000000 (12 zeros)
//0xfff ⇔ 111111111111
int edgeTable[256]={ 0x000, 0x109, 0x203, 0x30a, 0x406,
0x50f, …, 0x2fc, 0xdfc, 0xcf5, 0xfff, 0xef6, 0x9fa, …, 0x203,
0x109, 0x000};
```

- only vertex 3 was **below** the isosurface, cubeindex would equal 0000 1000 ⇔ 8.
- edgeTable[8] = 0x80c ⇔ 1000 0000 1100. It means that edge 2,3, and 11 are intersected by the isosurface.

Triangle Connection Table

- For each of the possible vertex states listed in *edgeTable* there is a specific triangulation of the edge intersection points.
- triTable lists all of them in the form of 0-5 edge triples with the list terminated by the invalid value -1.
  - {1, 8, 3, 9, 8, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}: means 2 triangles (2 edge triples)



Triangle Connection Table

It means edge 1, 3, 10 & 11

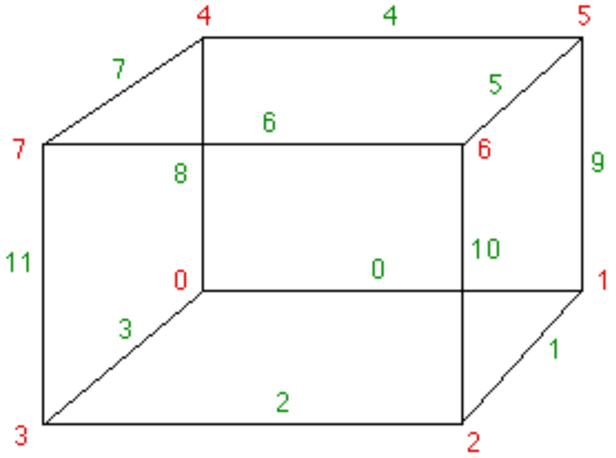
// For example:

// edgeTable [3] = 0x30a ⇔ 1100001010 ⇔ 778

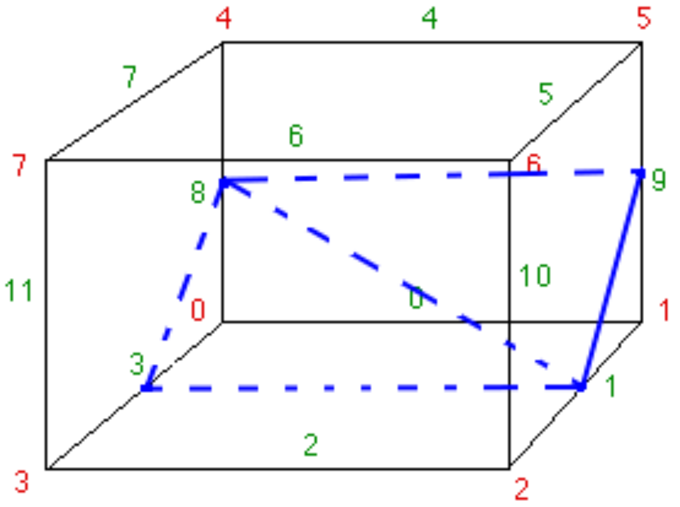
// triTable[3] list the 2 triangles formed when **corner[0] & corner[1] are inside** of the surface, but the rest of the cube is not.

0x0ca ⇔ 0011001010

```
int triTable[256][16] =
{
	{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
	{0, 8, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
	{0, 1, 9, - … -1, -1, -1},
	{1, 8, 3, 9 … -1, -1},…
}
```



Edge index
Vertex index



Triangle Connection Table

// For example:

// vertex state: **3** ⇔ 0000 0011

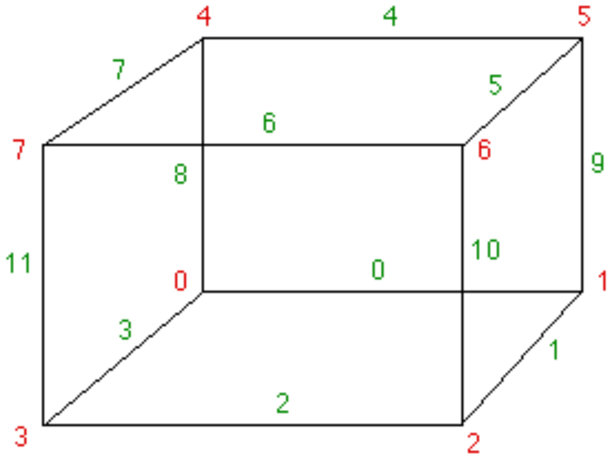
It means edge 1, 3, 8 & 9

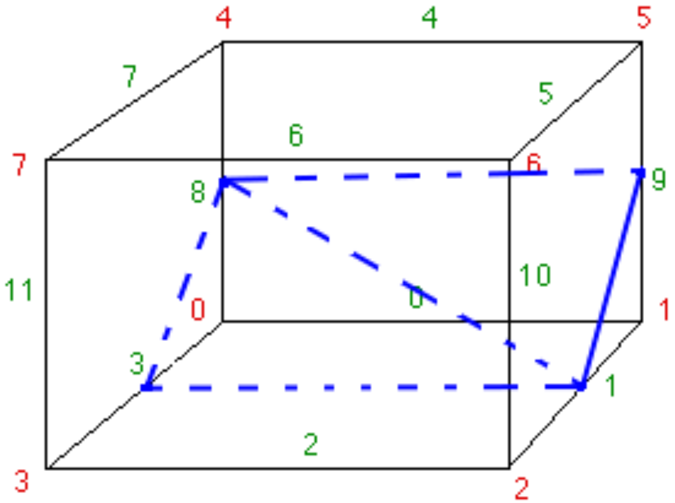
// edgeTable [**3**] = 0x30a ⇔ 0011 0000 1010;

// triTable[**3**] list the 2 triangles formed when **corner[0] & corner[1] are inside** of the surface, but the rest of the cube is not.

```
int triTable[256][16] =  {
        {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1},
        {0, 8, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1},
        {0, 1, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1},
        {1, 8, 3, 9 ... , -1, -
1},…
}
```

Edge index
Vertex index





intersection points

- by linear interpolation

$$P = P_1 + (\mathrm{isovalue} - V_1)\ (P_2 - P_1) \ /\ (V_2 - V_1)$$

Source code

References

- http://paulbourke.net/geometry/polygonise/
- **vtkMarchingCubes**
- CGAL: Poisson_reconstruction_function
- Matlab: Marching Cubes by Peter Hammer

- http://graphics.stanford.edu/~mdfisher/MarchingCubes.html
- **Andrew Nealen: CS 523: Computer Graphics : Shape Modeling**