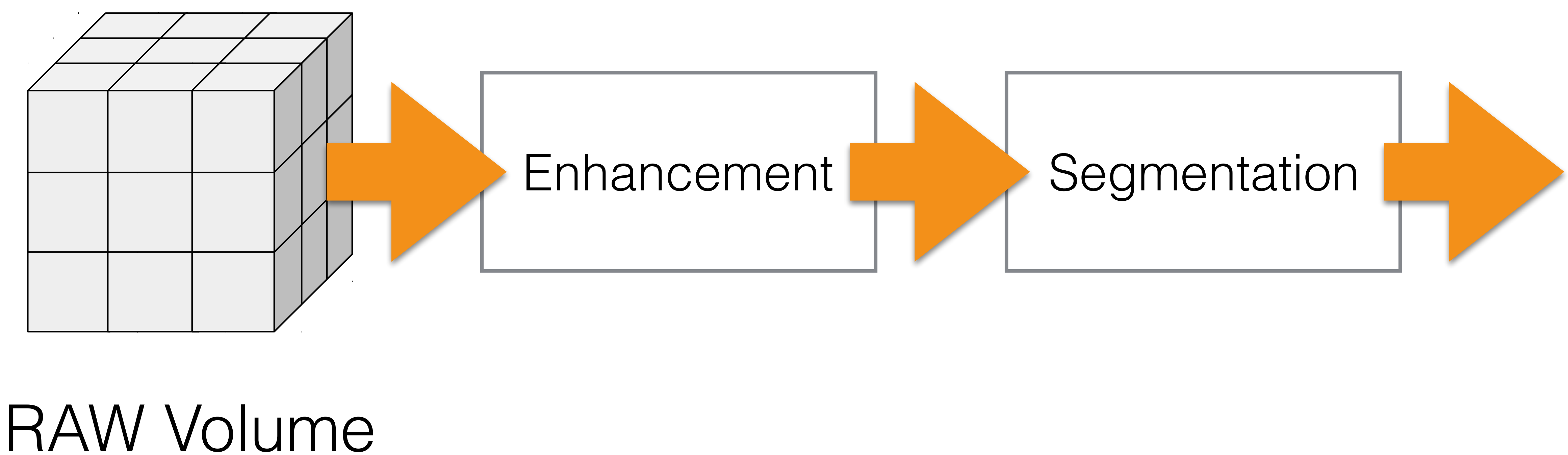
3D from Volume: Part III

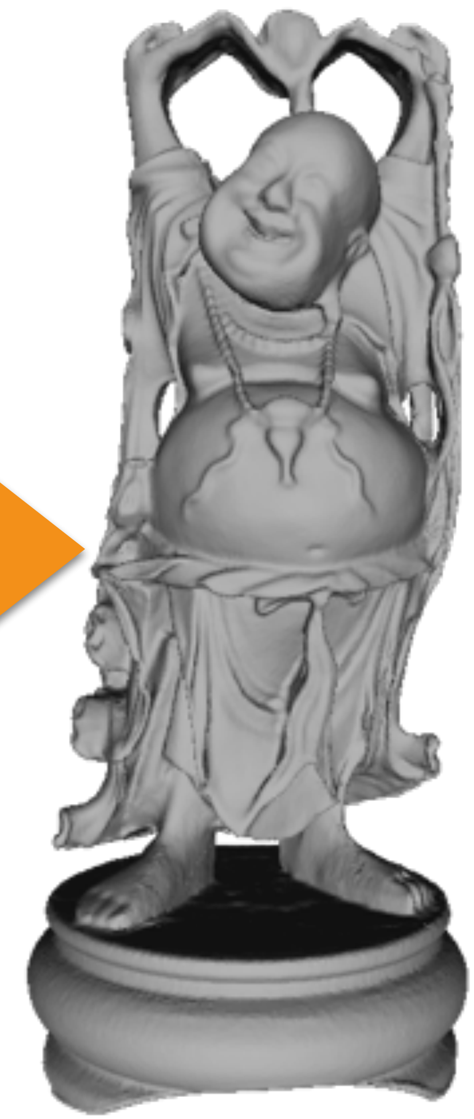
Dr. Francesco Banterle,
francesco.banterle@isti.cnr.it
banterle.com/francesco

The Processing Pipeline



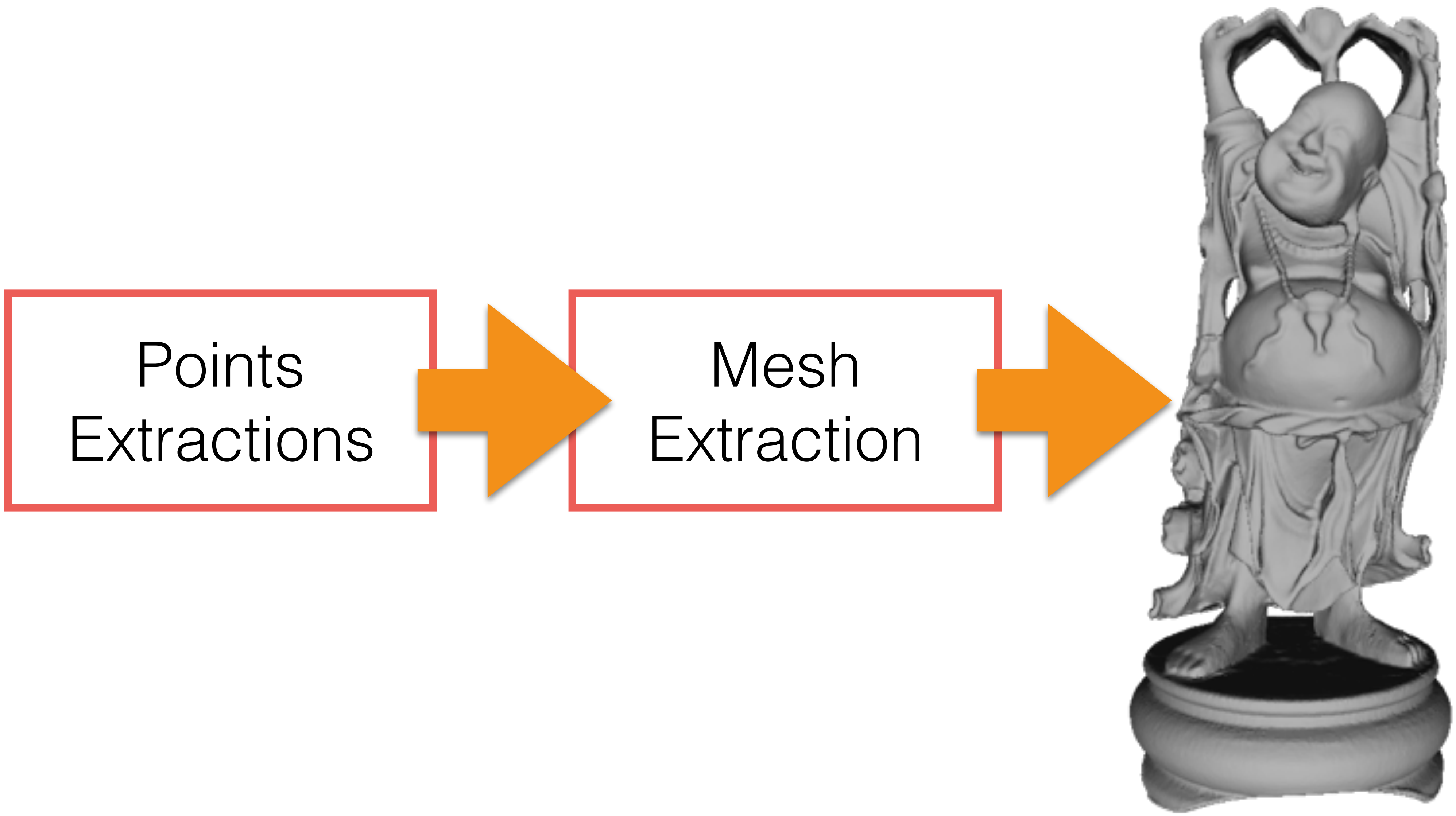
The Processing Pipeline

Points Extractions → Mesh Extraction →



3D Mesh

The Processing Pipeline



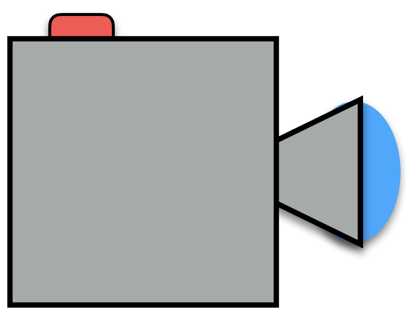
3D Mesh

3D Visualization

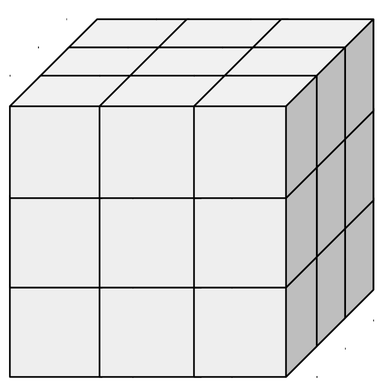
Volume Visualization

- We need to pre-visualize the 3D model that we are going to create.
- Pre-visualization is typically fast (no need to create a 3D model) and helps the segmentation process.

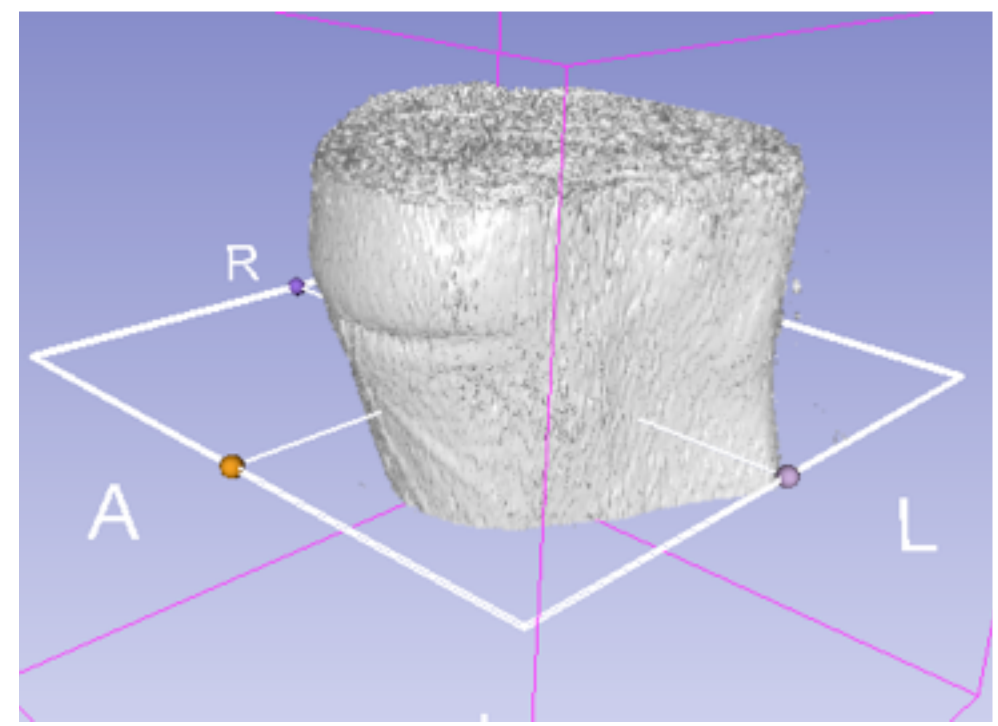
Volume Visualization



Camera



Volume



Camera Model: Pinhole Camera

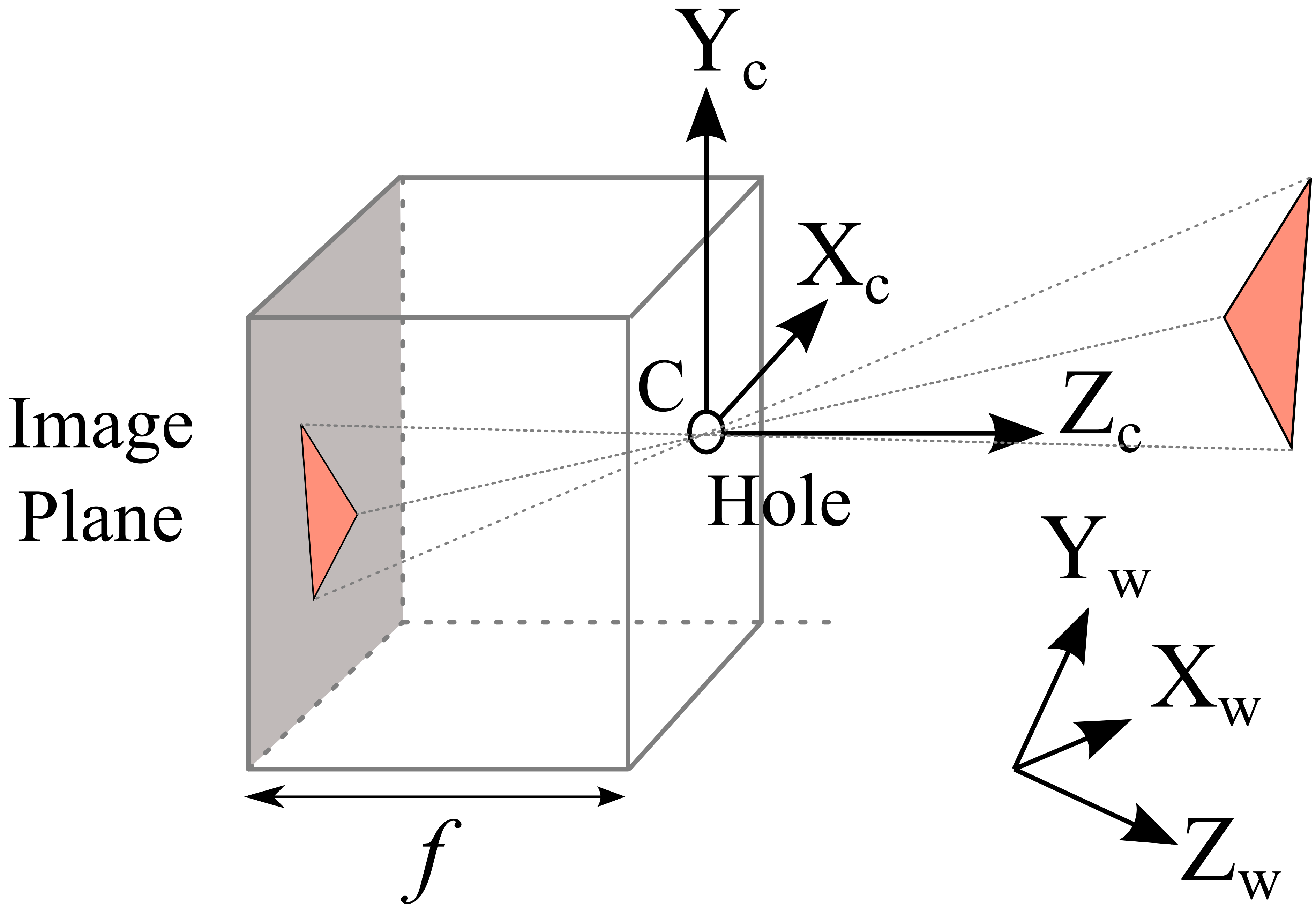


image-space

world-space

Camera Model

- Perspective projection:

$$\mathbf{M} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad \mathbf{m}' = \begin{cases} x' = -f \cdot \frac{x}{z} \\ y' = -f \cdot \frac{y}{z} \\ z' = -f \end{cases}$$

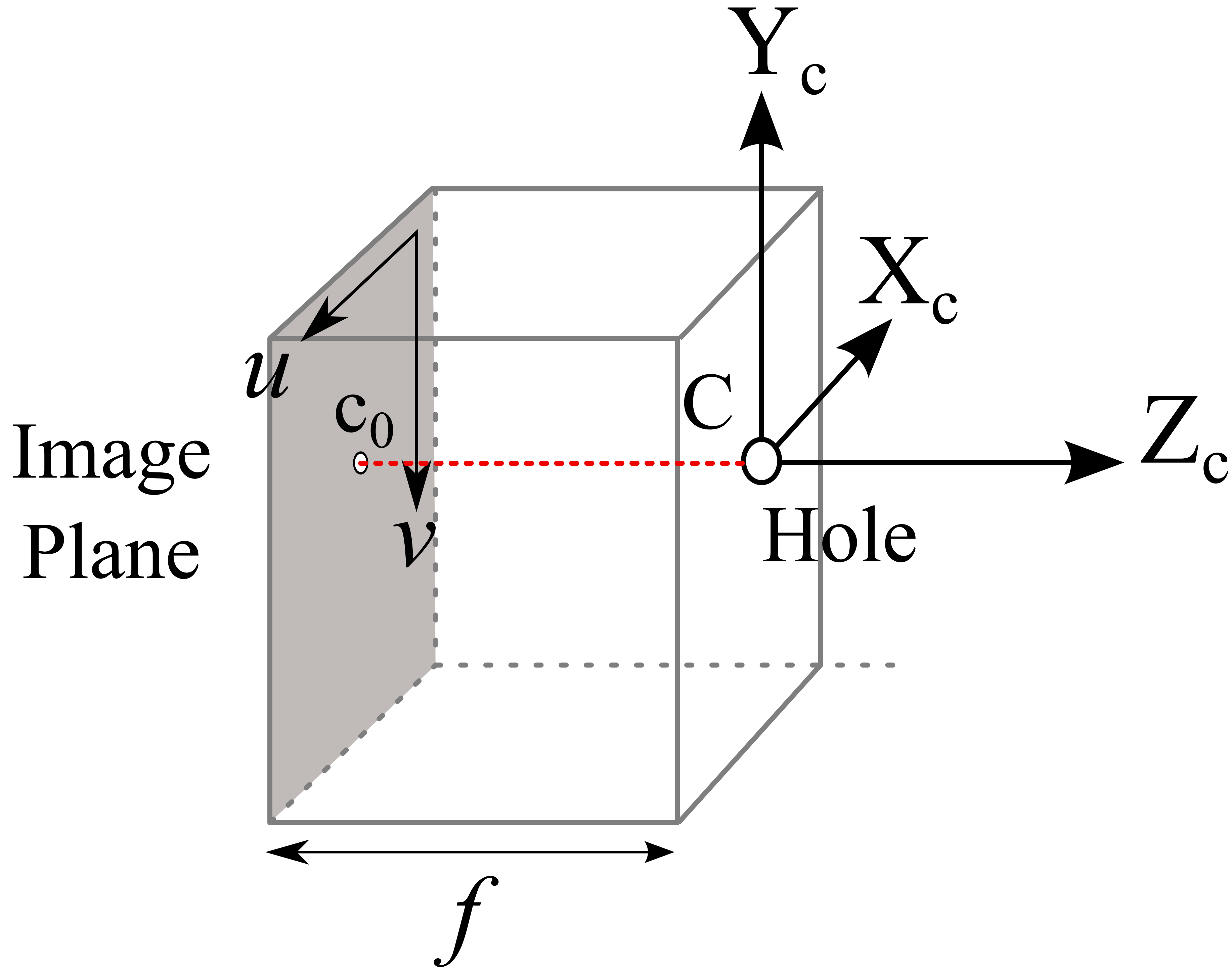
Camera Model

- Perspective projection:

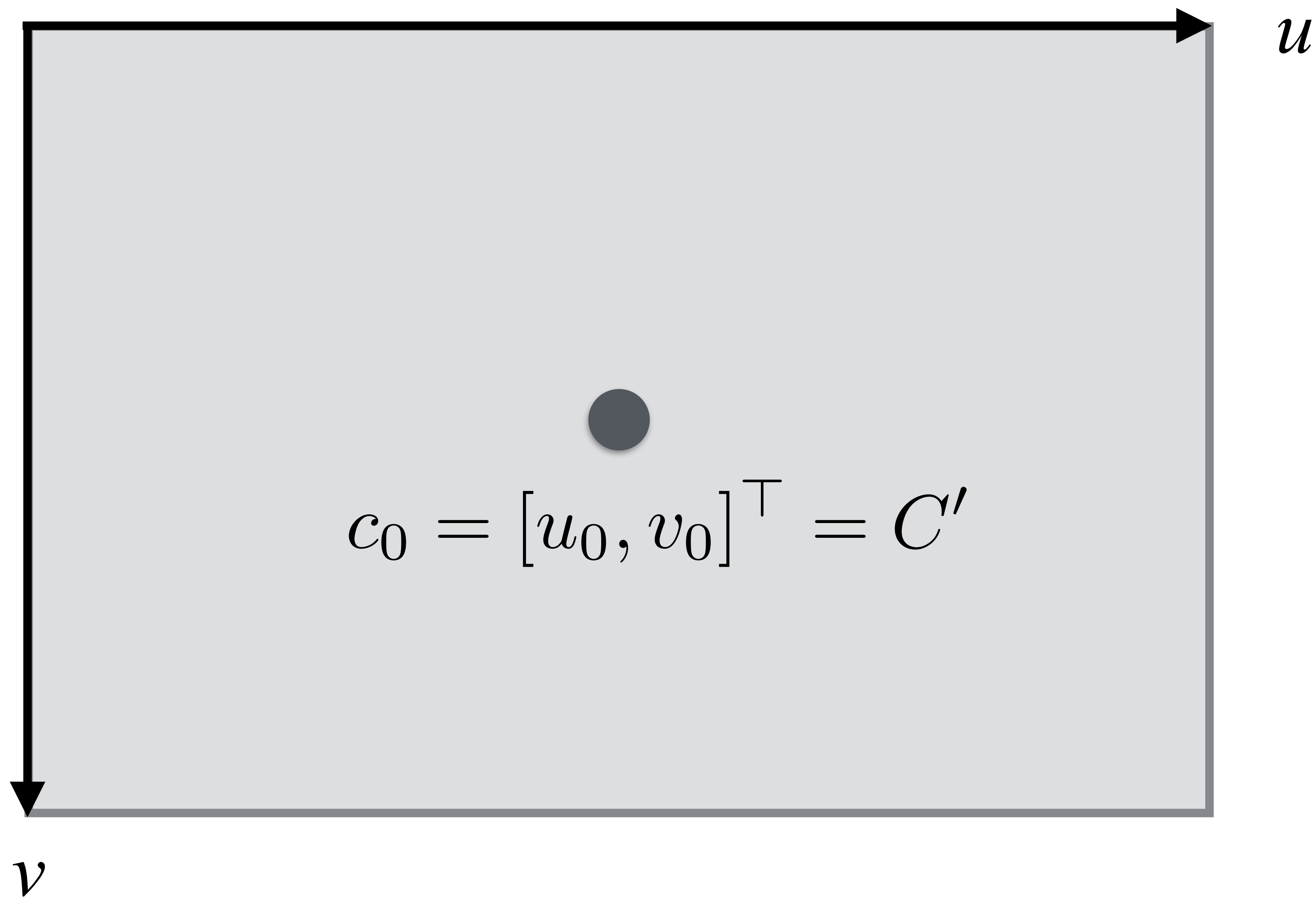
$$\mathbf{M} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad \mathbf{m}' = \begin{cases} x' = -f \cdot \frac{x}{z} \\ y' = -f \cdot \frac{y}{z} \\ z' = -f \end{cases}$$

Homogenous coordinates

Camera Model: Pinhole Camera



Camera Model: Image Plane



- Pixels have different height and width; i.e., (k_u, k_v).
- c_0 is called the principal point.

Camera Model: Intrinsic Parameters

- If we take all into account, we obtain:

$$\mathbf{m}' = \begin{cases} x' = -k_u \cdot f \cdot \frac{x}{z} + u_0 \\ y' = -k_v \cdot f \cdot \frac{y}{z} + v_0 \\ z' = -f \end{cases}$$

$$\mathbf{m}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

Camera Model: Intrinsic Parameters

- This can be expressed in a matrix form with a non-linear projection:

$$\mathbf{m} = P \cdot \mathbf{M} \qquad \mathbf{m}' = \mathbf{m}/\mathbf{m}_z$$

$$P = \begin{bmatrix} -fk_u & 0 & u_0 & 0 \\ 0 & -fk_v & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} = K[I|\mathbf{0}] \qquad K = \begin{bmatrix} -fk_u & 0 & u_0 \\ 0 & -fk_v & v_0 \\ 0 & 0 & 1 \end{bmatrix}$$

Camera Model: Extrinsic Parameters

- They define the pose of the camera; i.e., its orientation and position in the world-space.

- This is defined as geometry matrix G:

$$G = \begin{bmatrix} R & \mathbf{t} \\ 0 & 1 \end{bmatrix}$$

- R is a 3x3 rotation matrix (orthogonal matrix with determinant 1) —> 3 angles: yaw, pitch, and, roll

- **t** is translation vector (3 components)

Camera Model

- The full camera model including the camera pose is defined as

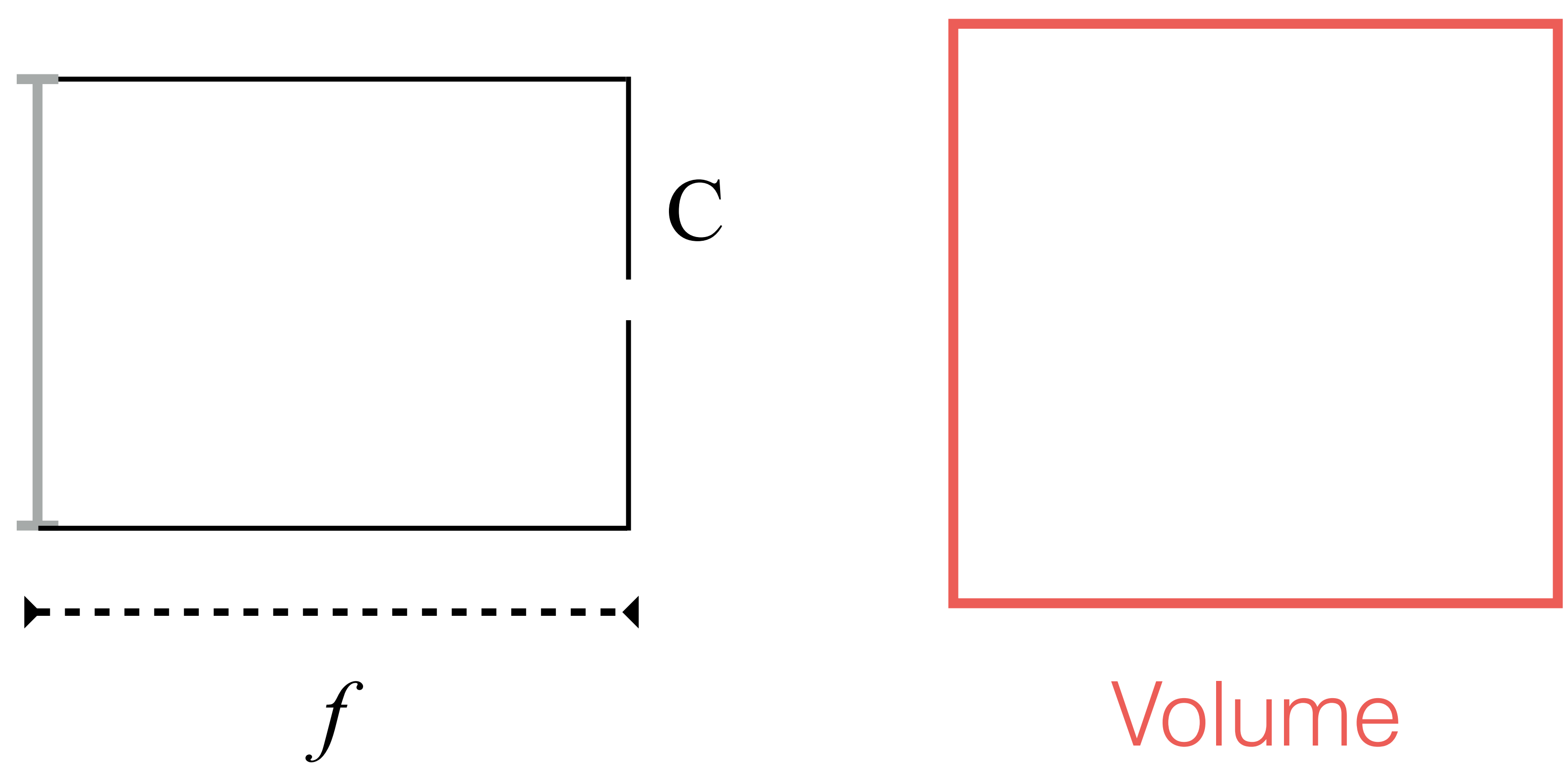
$$P = K[I|\mathbf{0}]G = K[R|\mathbf{t}]$$

$$\mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} \quad R = \begin{bmatrix} \mathbf{r}_1^\top \\ \mathbf{r}_2^\top \\ \mathbf{r}_3^\top \end{bmatrix}$$

- P is 3x4 matrix with 11 independent parameters!

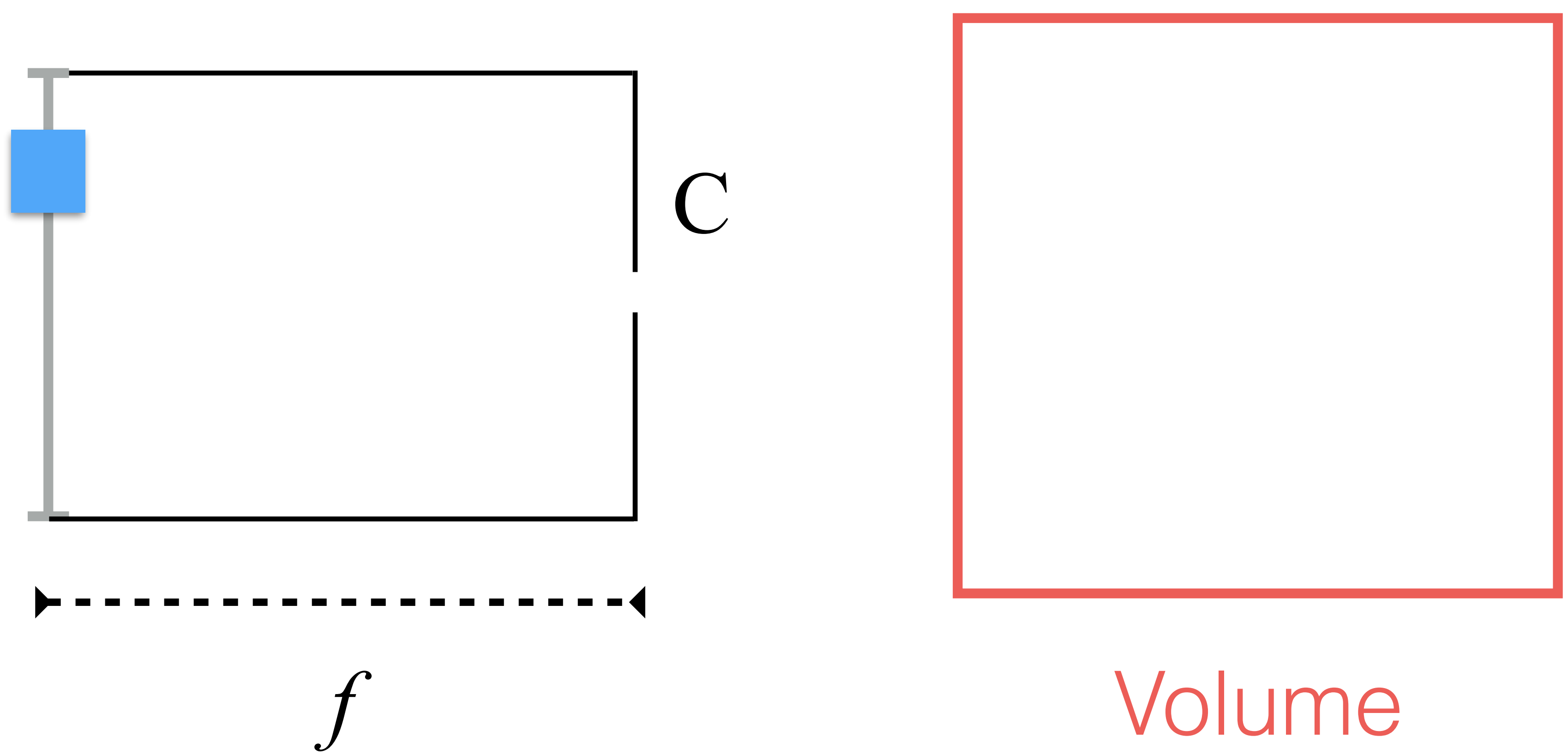
Rendering

- The idea is to create for each pixel a ray (origin and direction) that is going to intersect the volume.
- We will color the pixel if its ray intersect the volume. Otherwise the pixel will be set to zero.



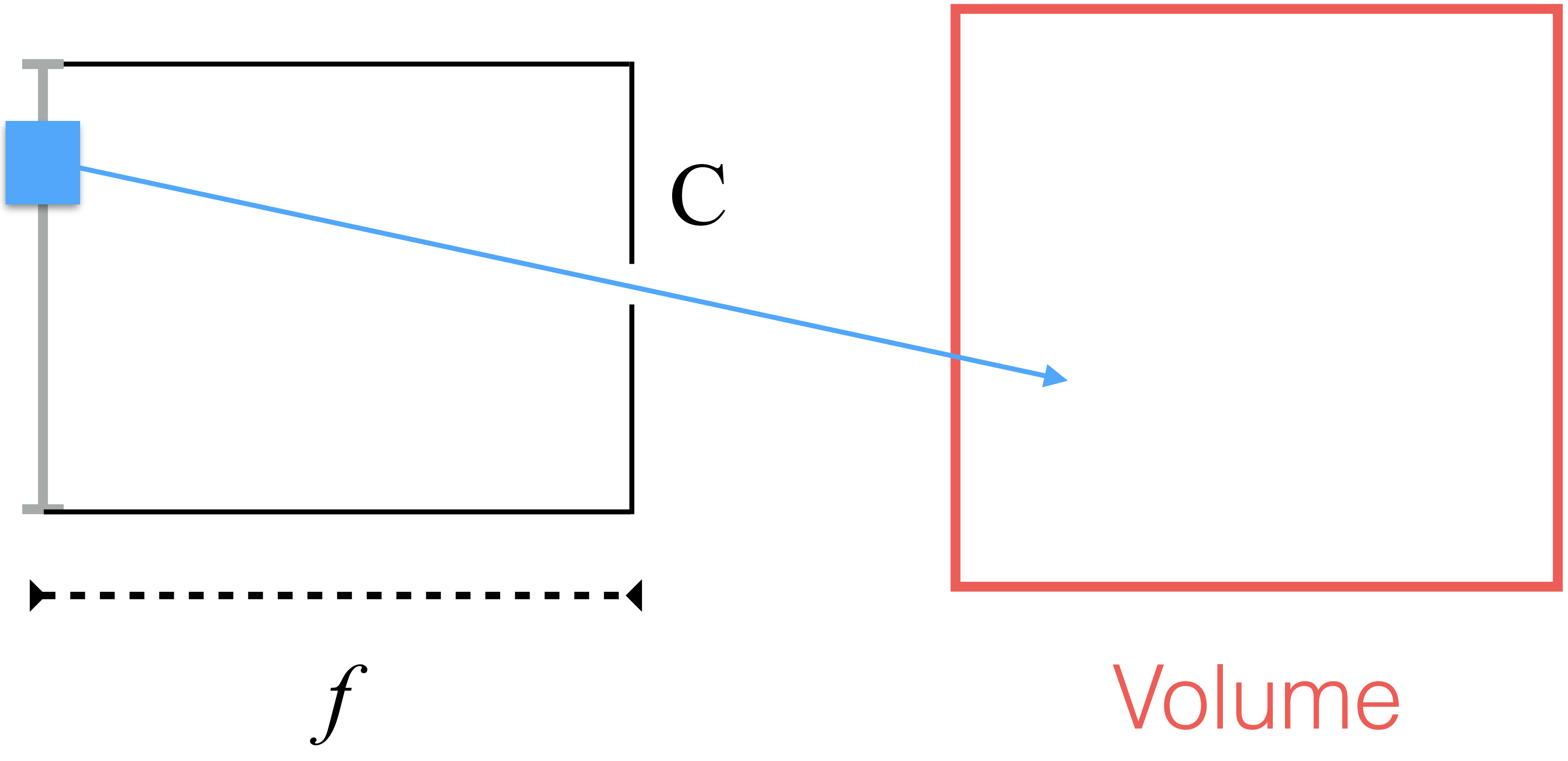
Rendering

- The idea is to create for each pixel a ray (origin and direction) that is going to intersect the volume.
- We will color the pixel if its ray intersect the volume. Otherwise the pixel will be set to zero.

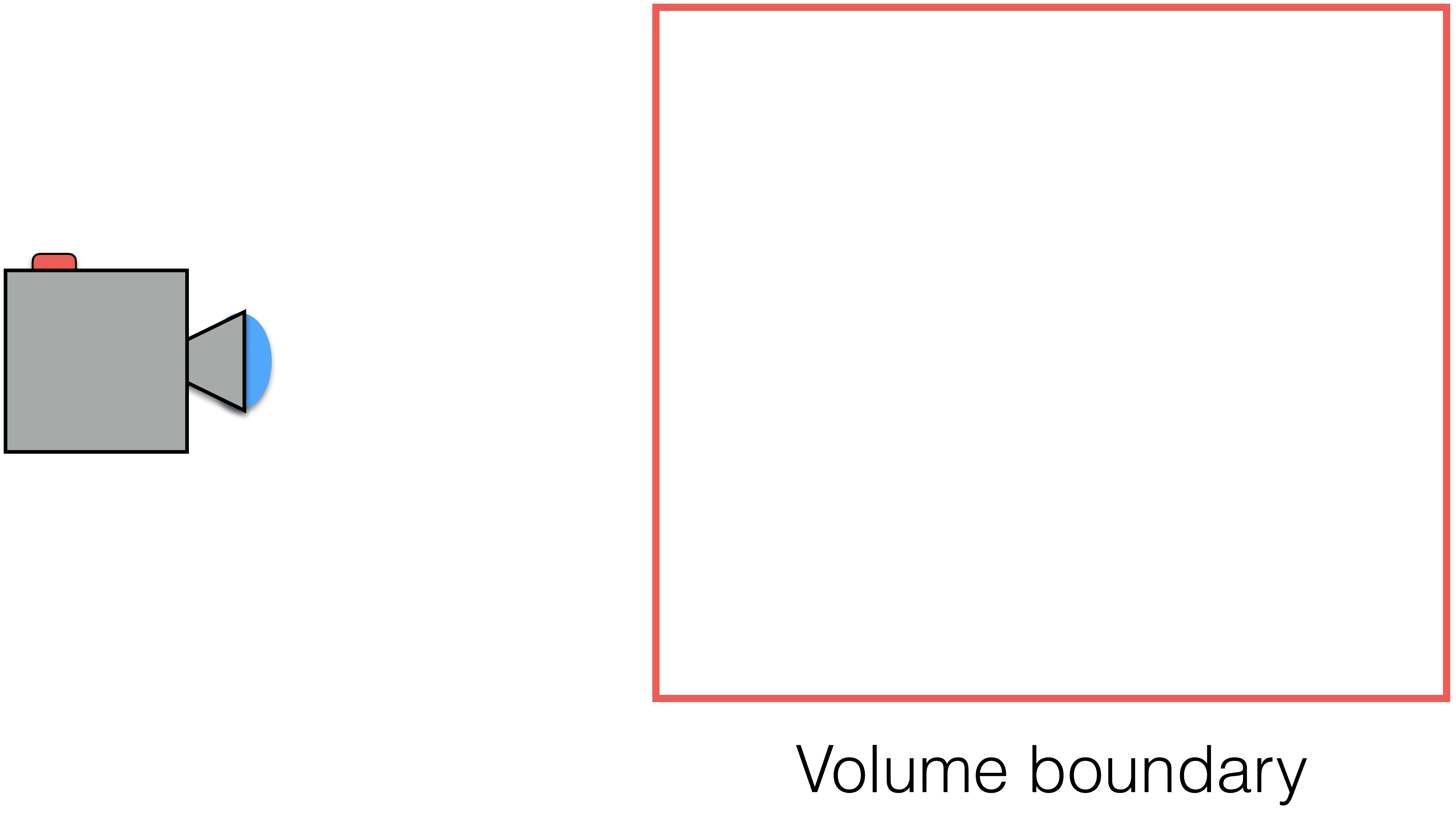


Rendering

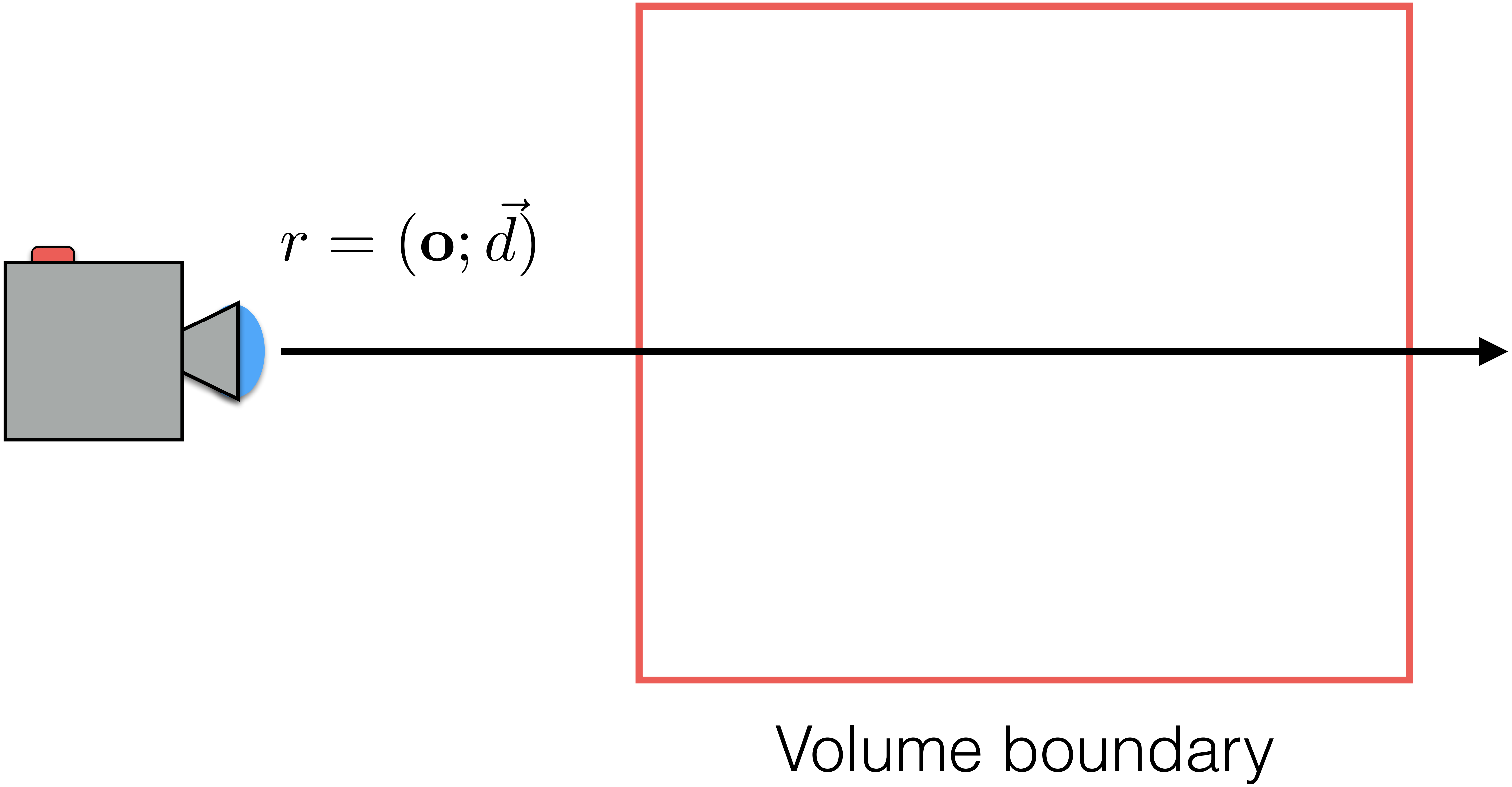
- The idea is to create for each pixel a ray (origin and direction) that is going to intersect the volume.
- We will color the pixel if its ray intersect the volume. Otherwise the pixel will be set to zero.



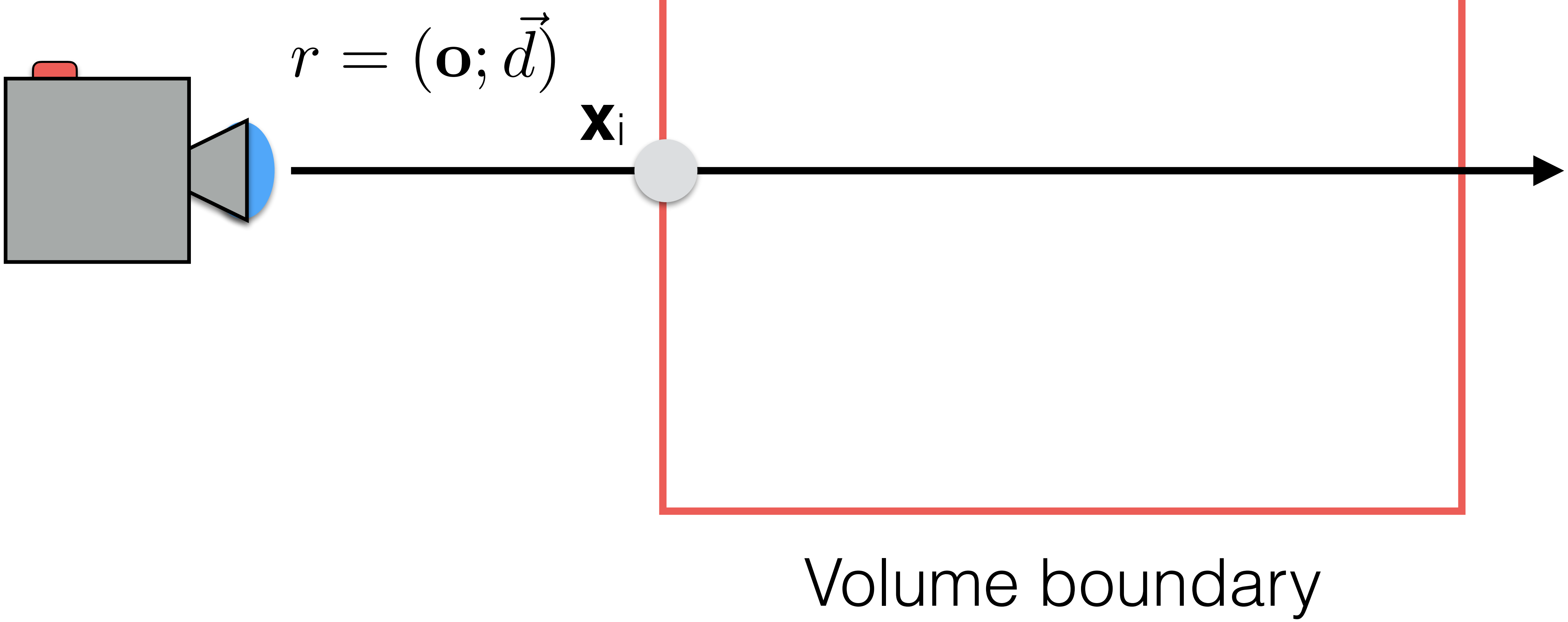
Volume Rendering: Ray-Marching



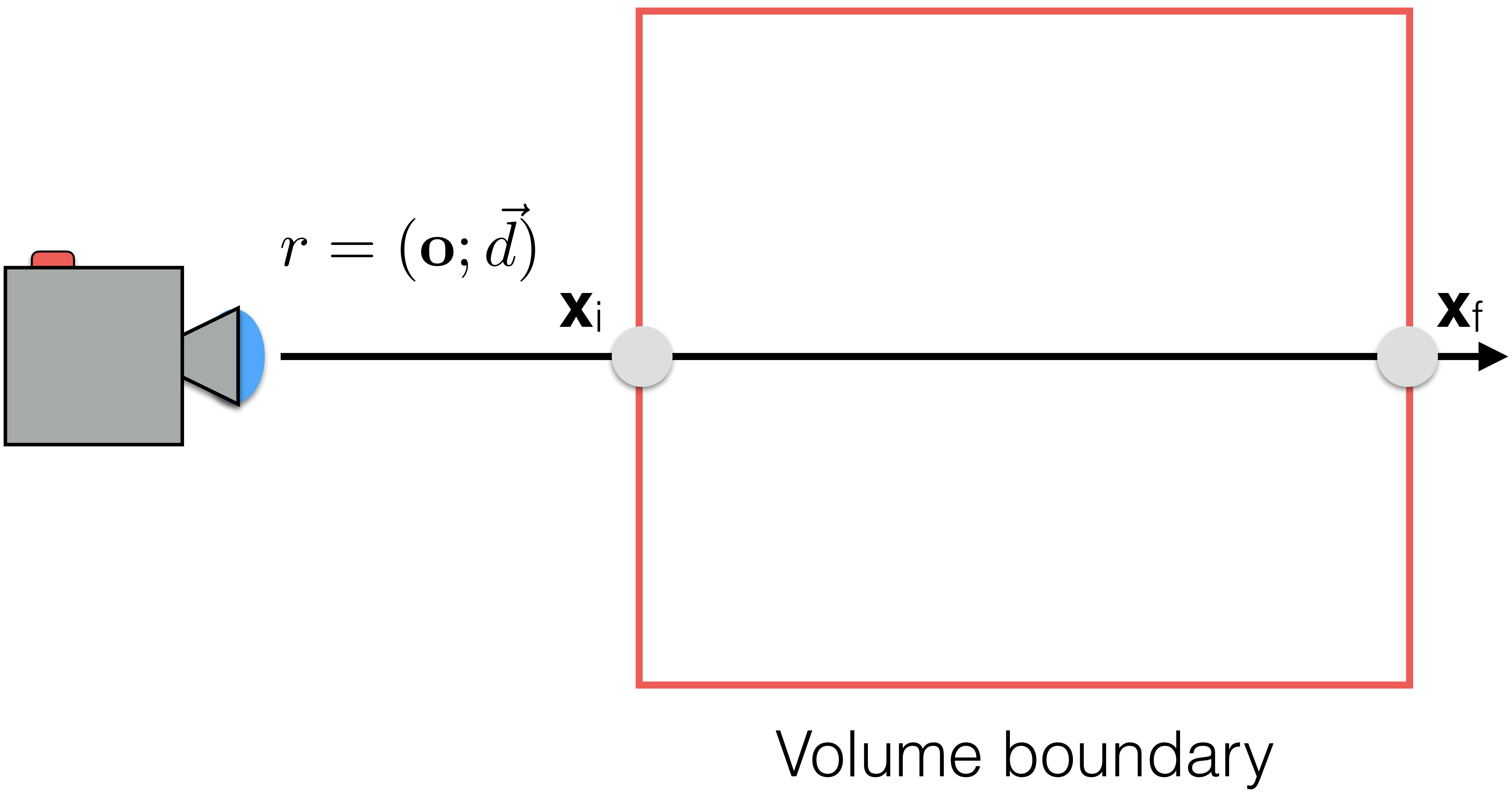
Volume Rendering: Ray-Marching



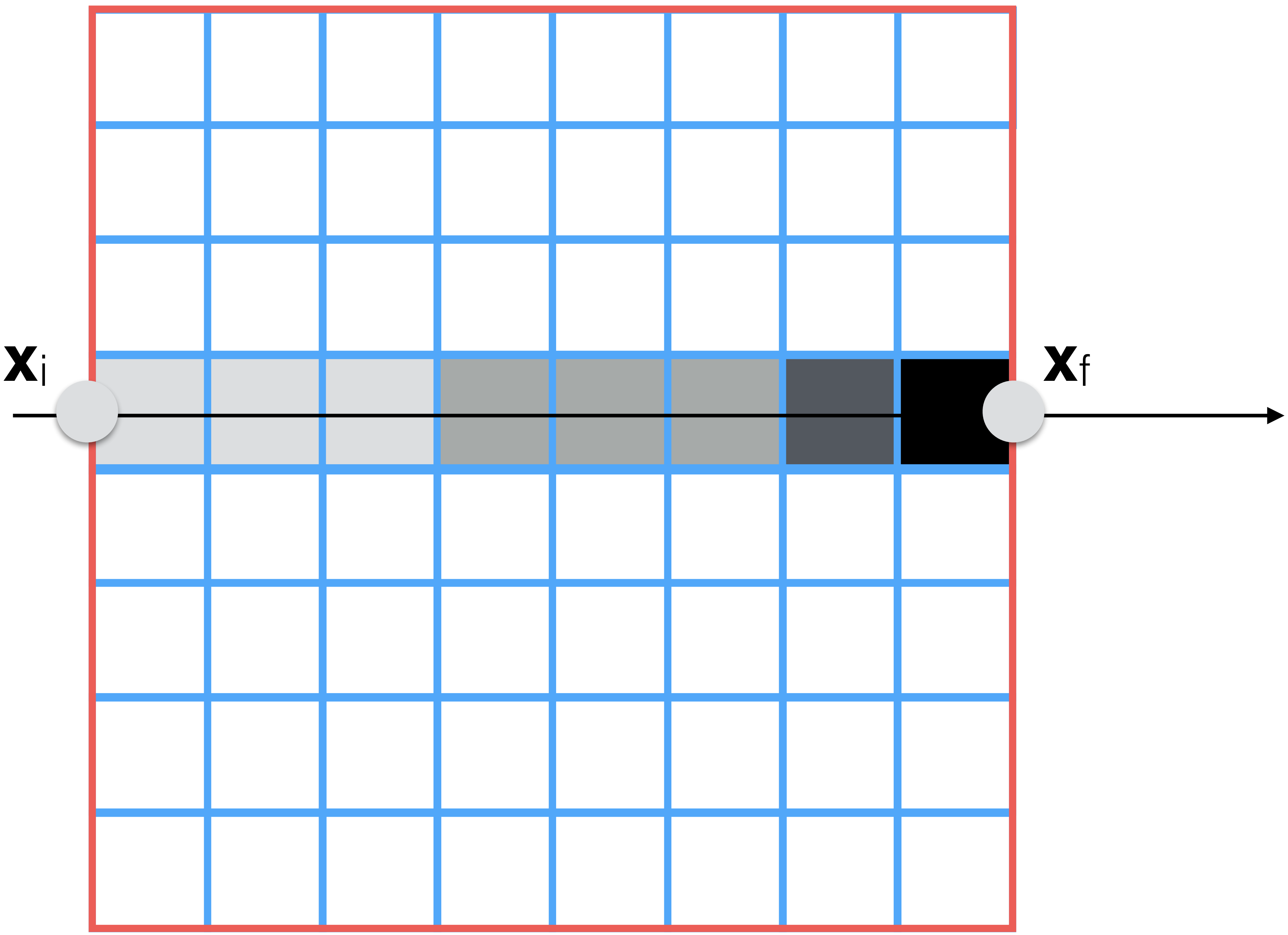
Volume Rendering: Ray-Marching



Volume Rendering: Ray-Marching



Volume Rendering: Ray-Marching

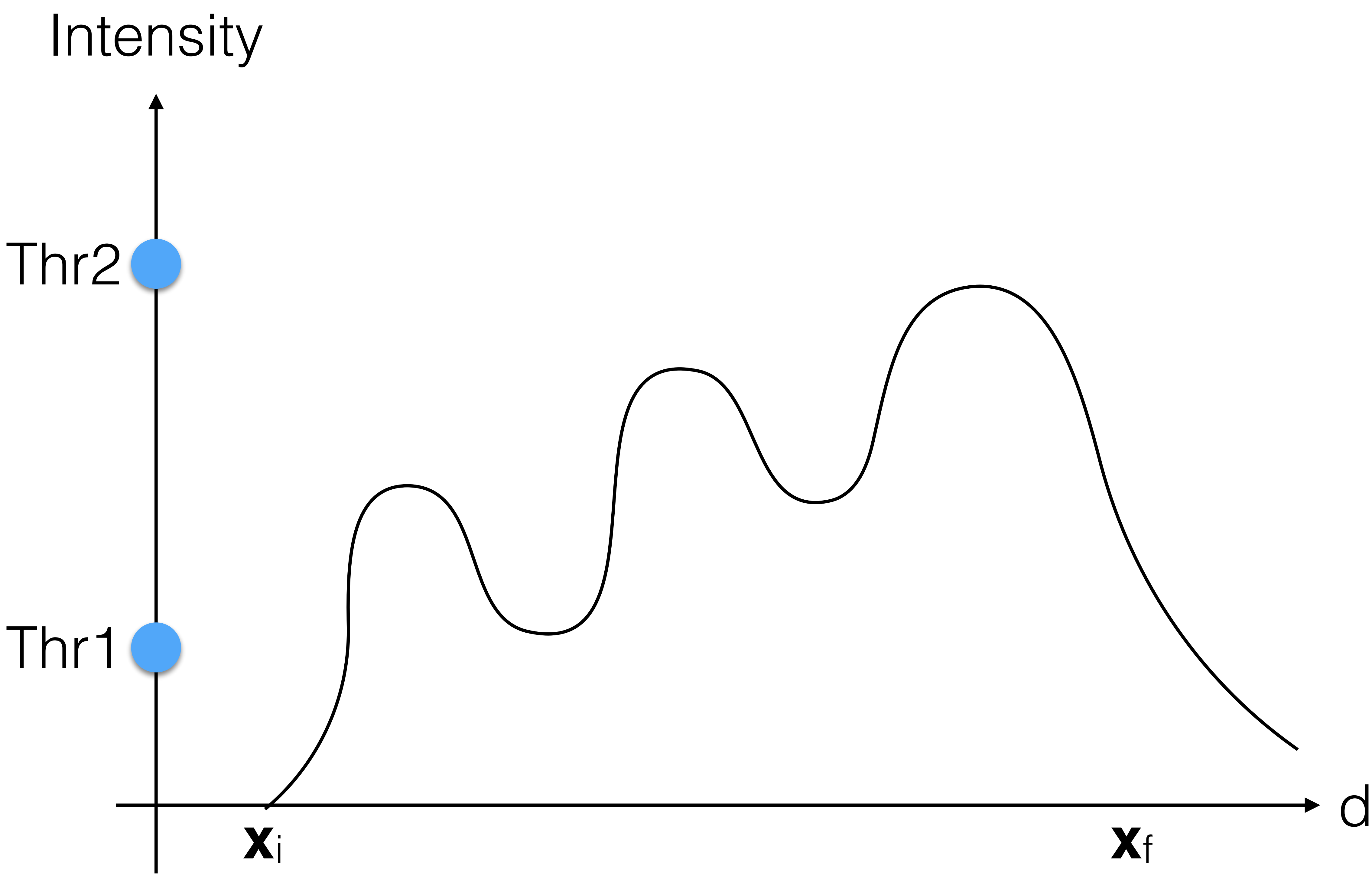


Volume Rendering: Ray-Marching

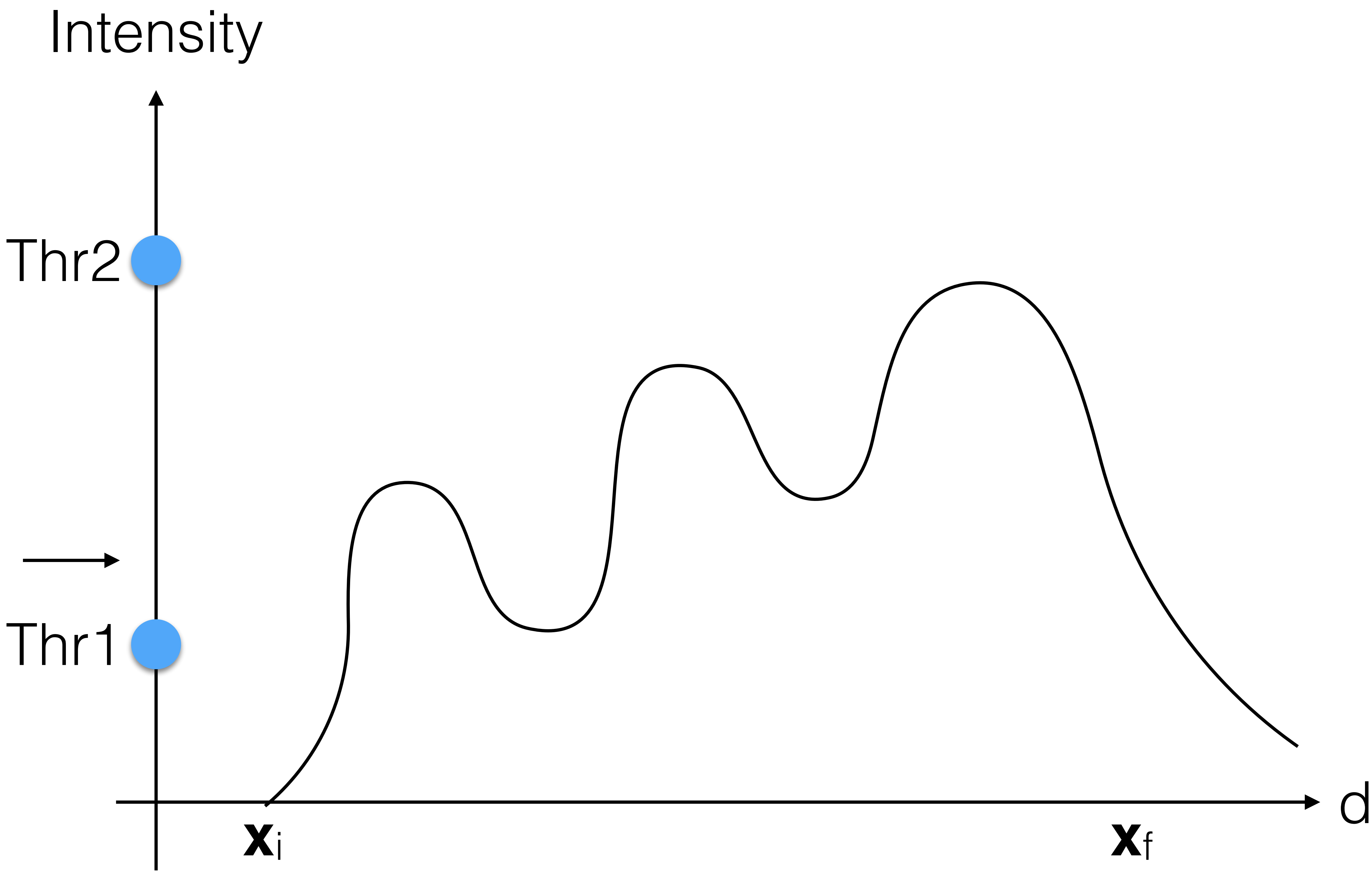
$$I[u,v] = \int_{x_i}^{x_f} T\Bigg(V(\mathbf{o}[u,v] + \vec{d}[u,v] \cdot t)\Bigg) dt$$

T is called the transfer function

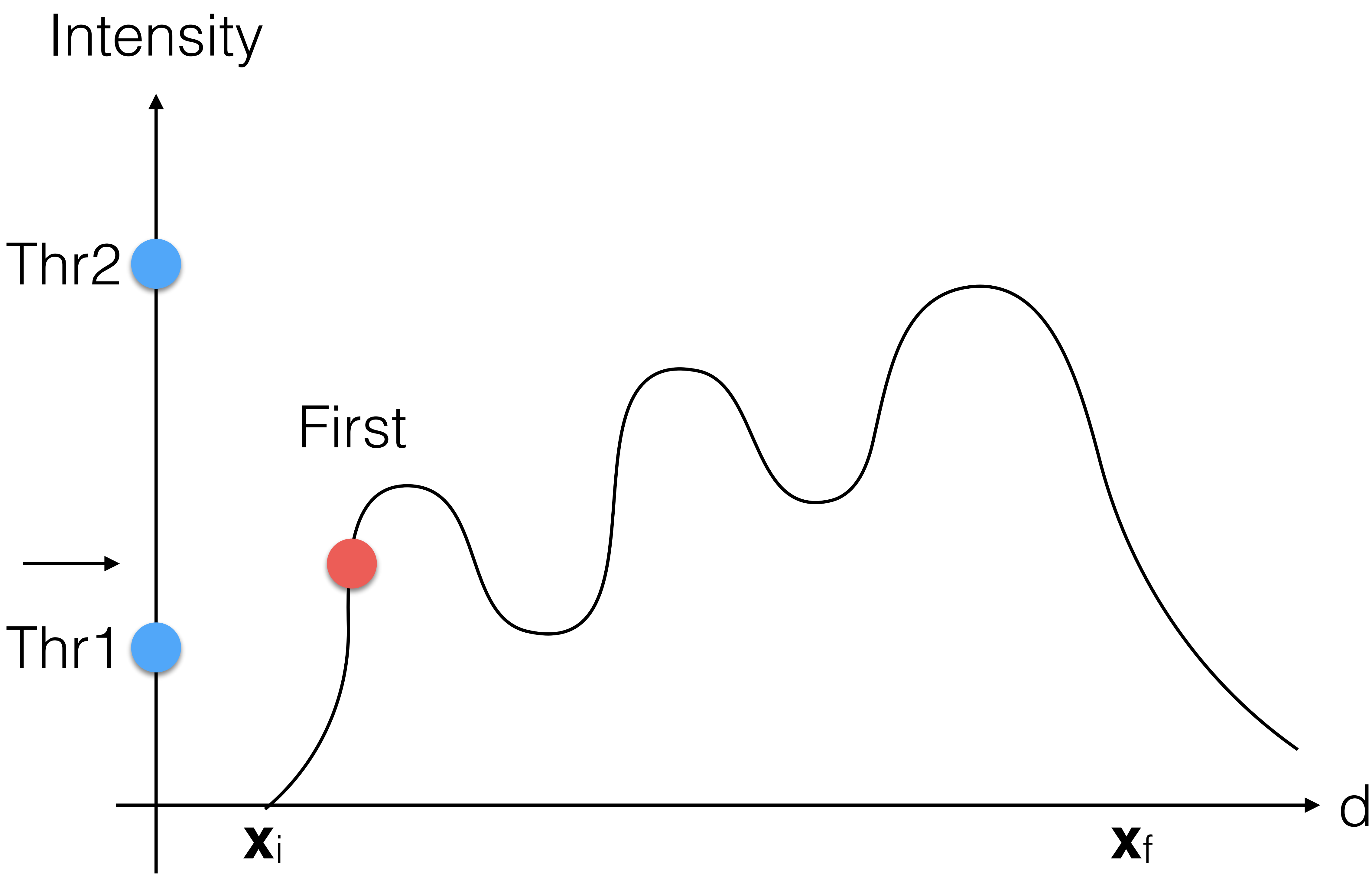
Volume Rendering: Ray-Marching



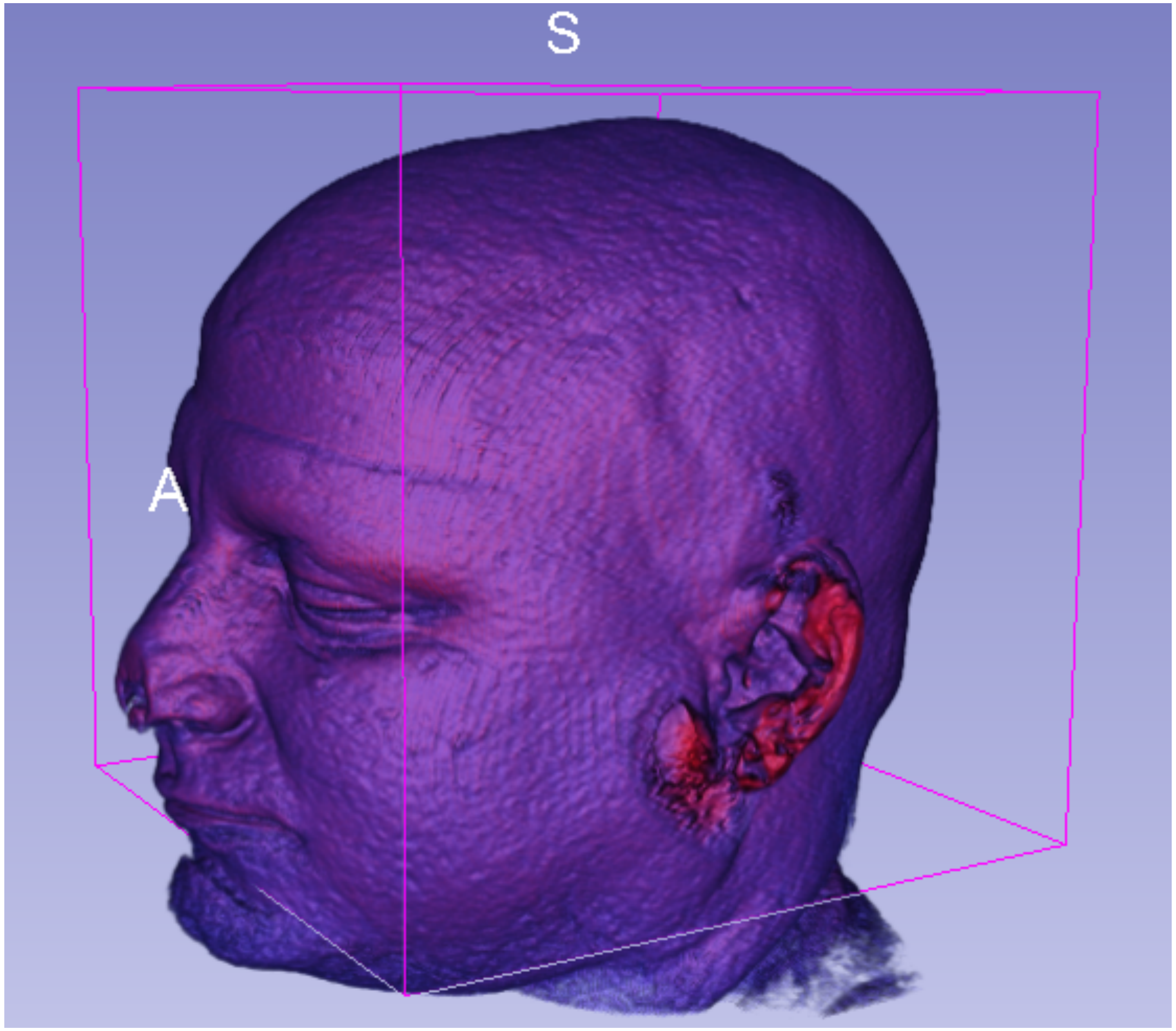
Volume Rendering: Ray-Marching



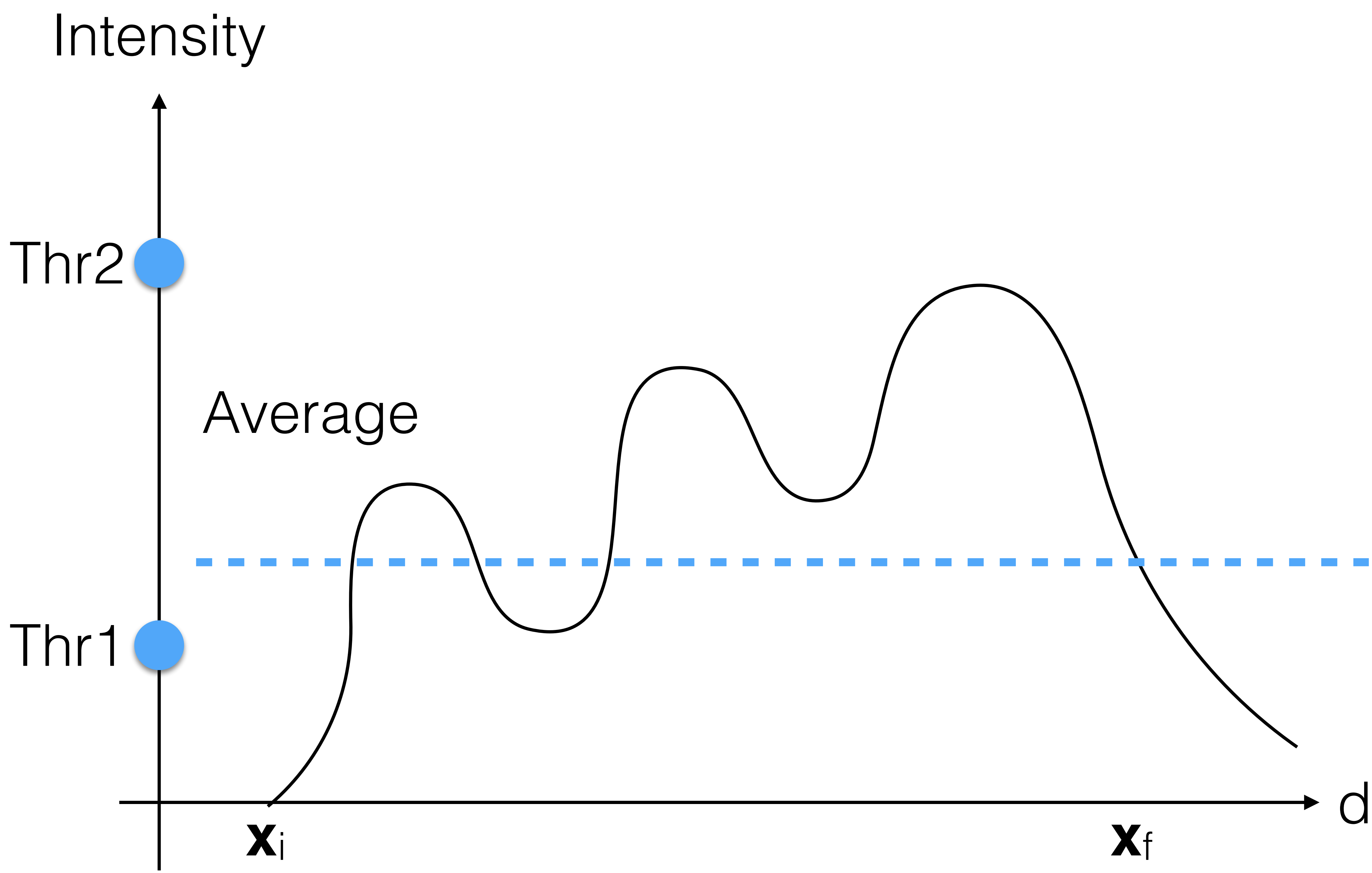
Volume Rendering: Ray-Marching



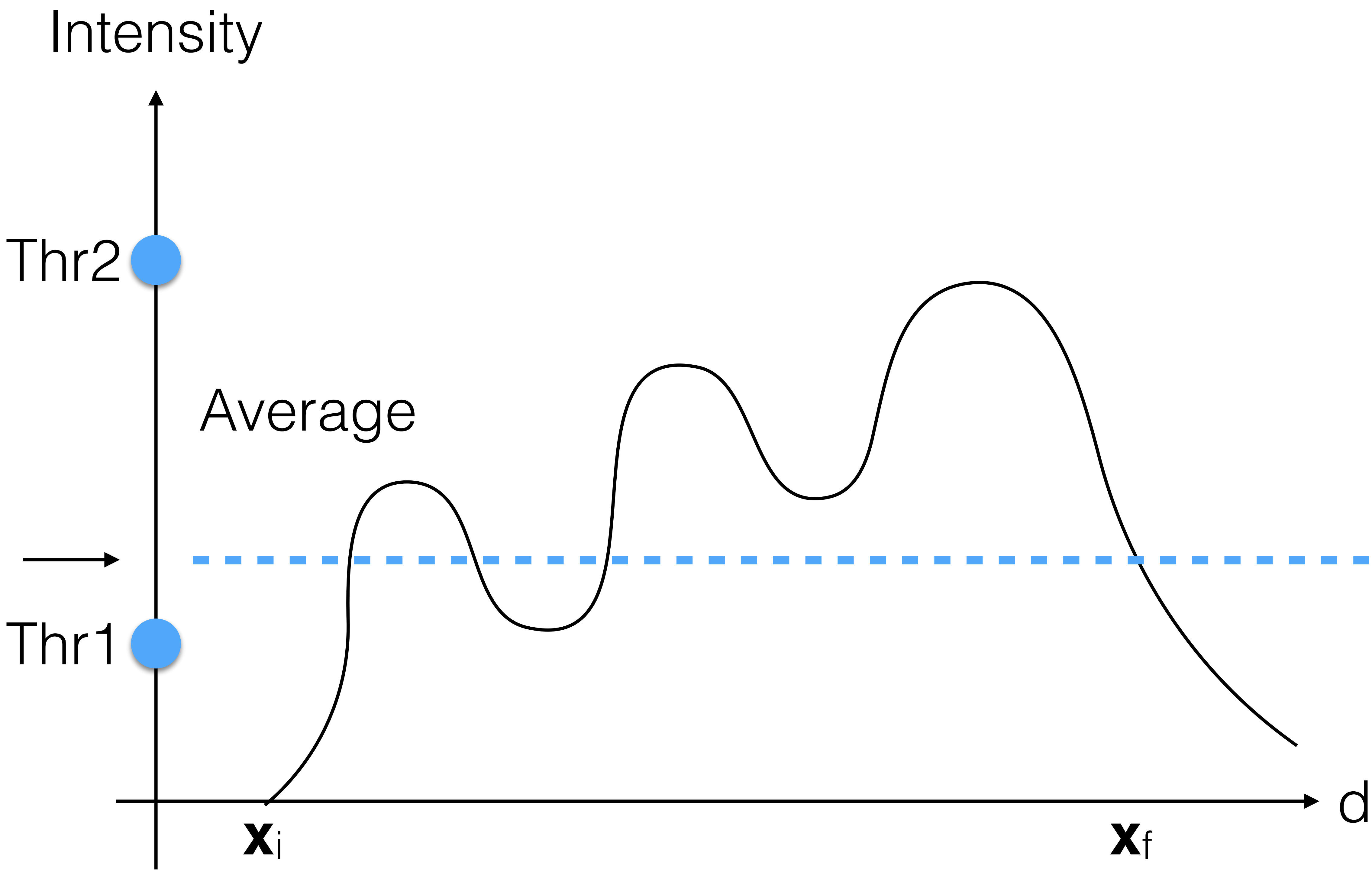
Volume Rendering: Ray-Marching Example



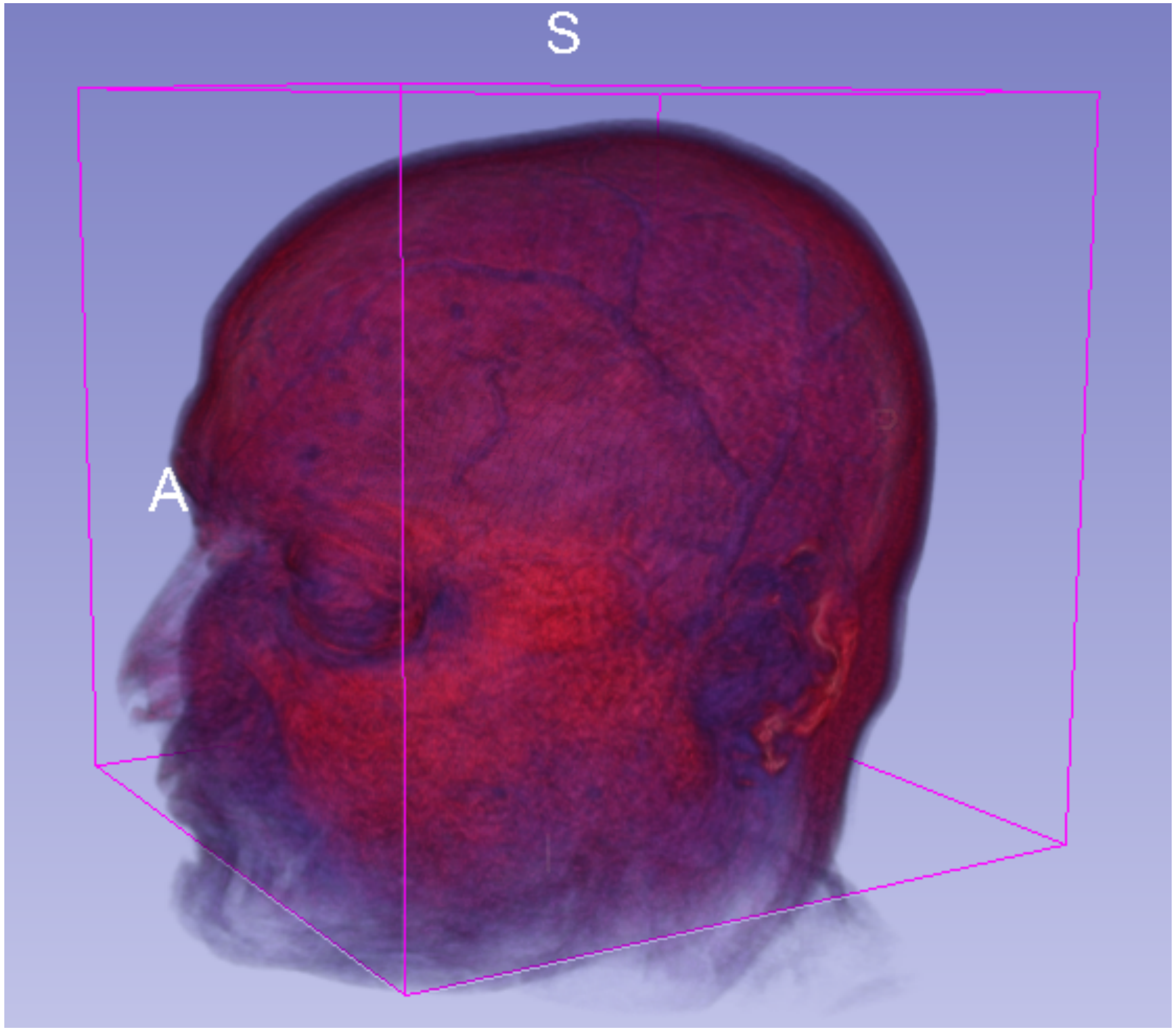
Volume Rendering: Ray-Marching



Volume Rendering: Ray-Marching



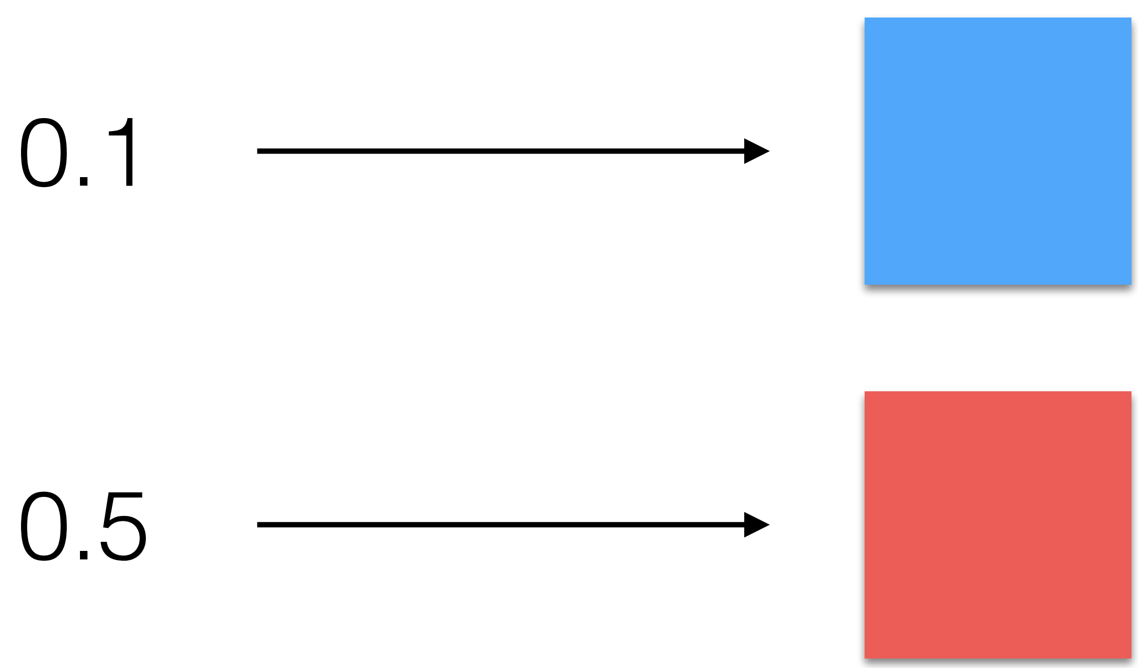
Volume Rendering: Ray-Marching Example



hang on!
Volumes have intensity values not colors…

Volume Rendering: Color Mapping

- To improve visualization intensity values are mapped to colors:



- In between values are linearly interpolated.

Volume Rendering: Let there be light

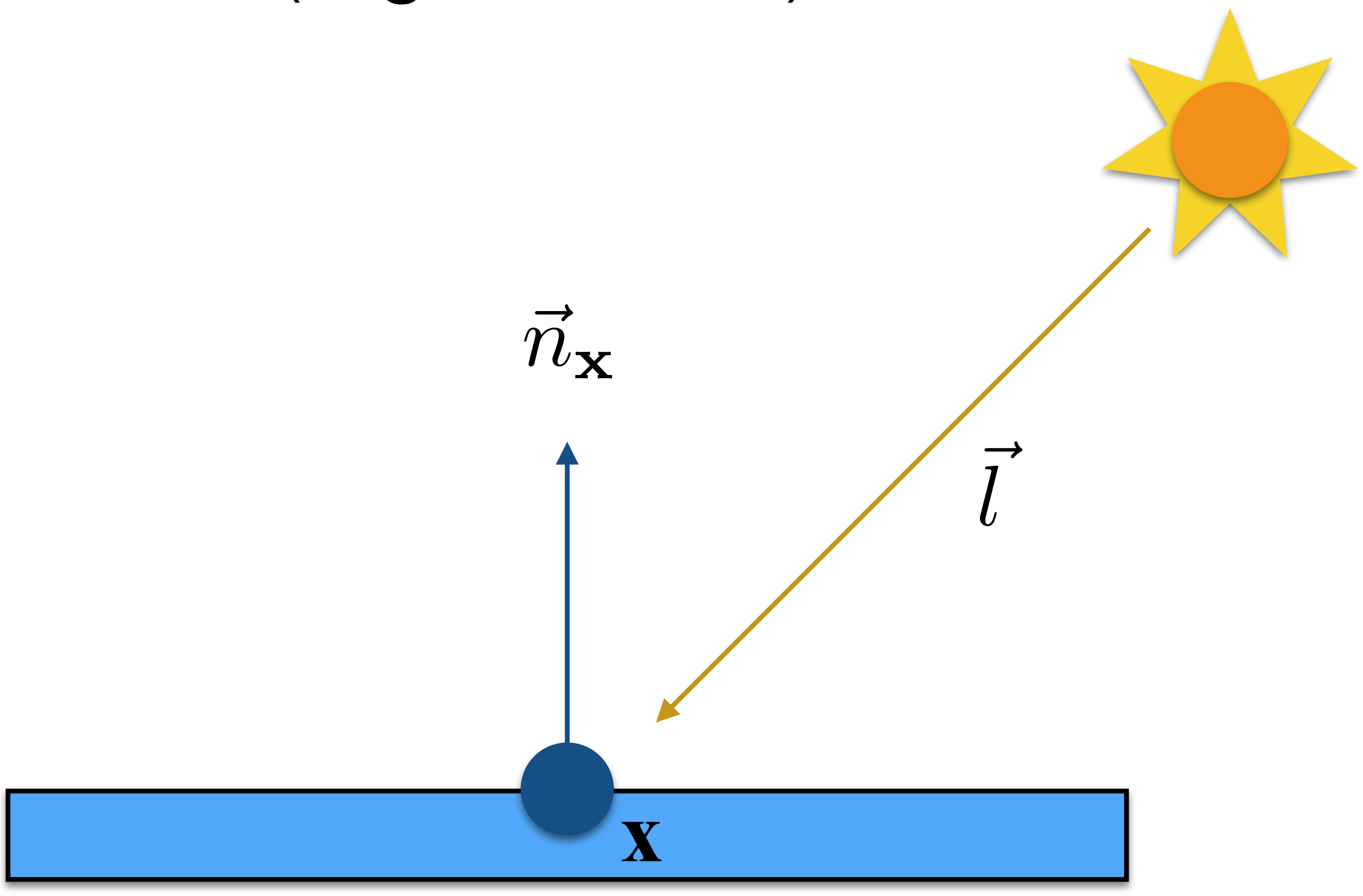
- We can improve quality by adding light sources.
- There are local (taking into account that light bounces around) and global models.
- For the sake of simplicity, we are interested in local models only!

Volume Rendering: Let there be light

- A local model is a function computing radiance (L); i.e., the value for coloring the pixel using only local geometry information:
  - Point's position.
  - Point's normal.
  - Optical properties of the material at its position. The intensity value of the volume (or its color encoding) in our case.
  - Light source's position.

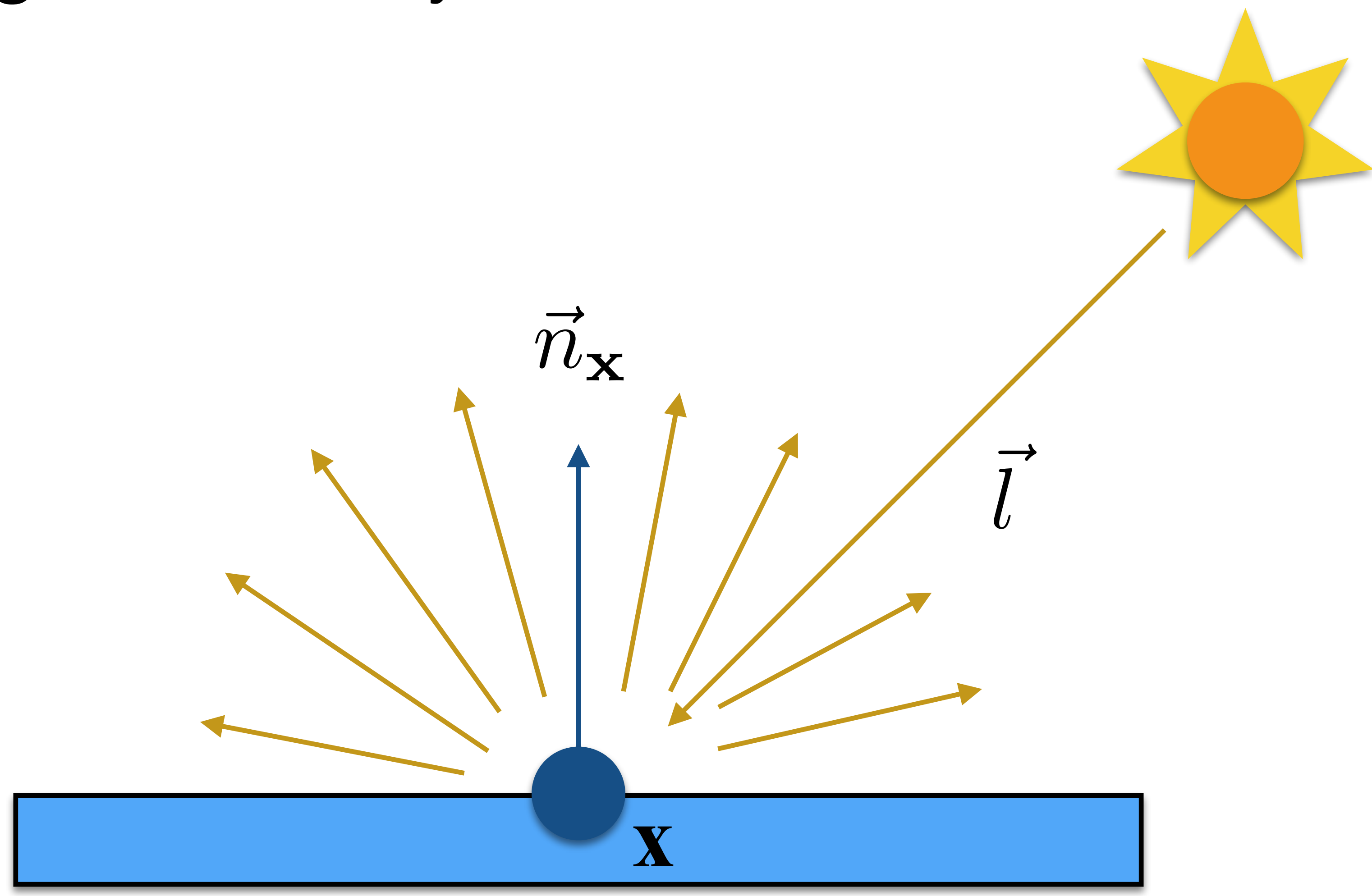
Volume Rendering: Let there be light

- A simple model assumes that the light source is placed at infinite (e.g., the sun):



Volume Rendering: Let there be light

- A simple local model is the diffuse model that assume light is locally reflected in all directions:



Volume Rendering: Let there be light

- The model is defined as

$$L(\mathbf{x}) = \frac{\lambda}{\pi} \cdot \max(-\vec{n}_{\mathbf{x}} \cdot \vec{l}, 0)$$

- Note that:
  - $\vec{n}_{\mathbf{x}}$ needs to be normalized.
  - $\vec{l}$ needs to be normalized.

Volume Rendering: Let there be light

- The model is defined as

Radiance

$$\boxed{L(\mathbf{x})} = \frac{\lambda}{\pi} \cdot \max(-\vec{n}_{\mathbf{x}} \cdot \vec{l}, 0)$$

- Note that:
  - $\vec{n}_{\mathbf{x}}$ needs to be normalized.
  - $\vec{l}$ needs to be normalized.

Volume Rendering: Let there be light

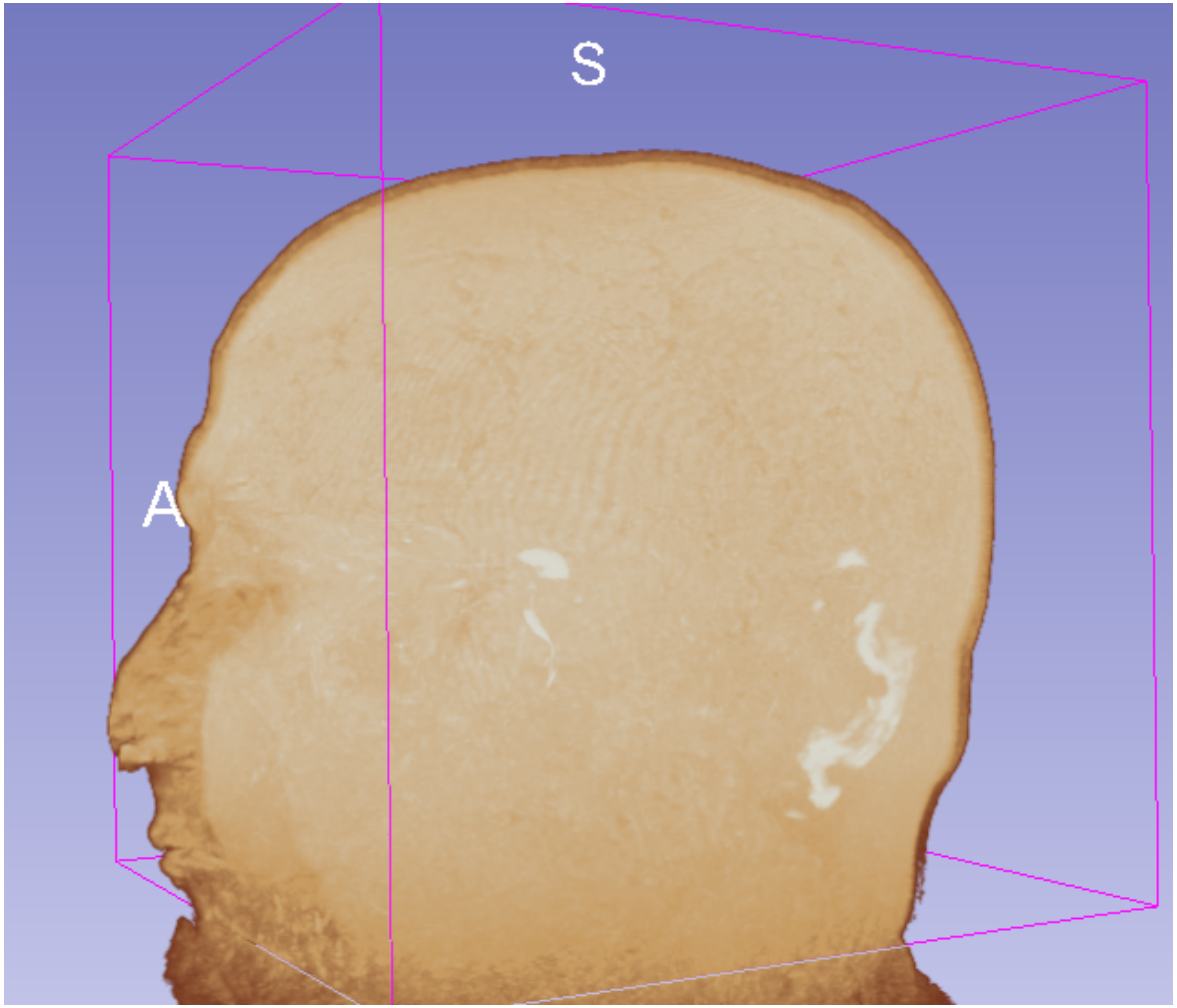
- The model is defined as

Radiance Albedo/Intensity

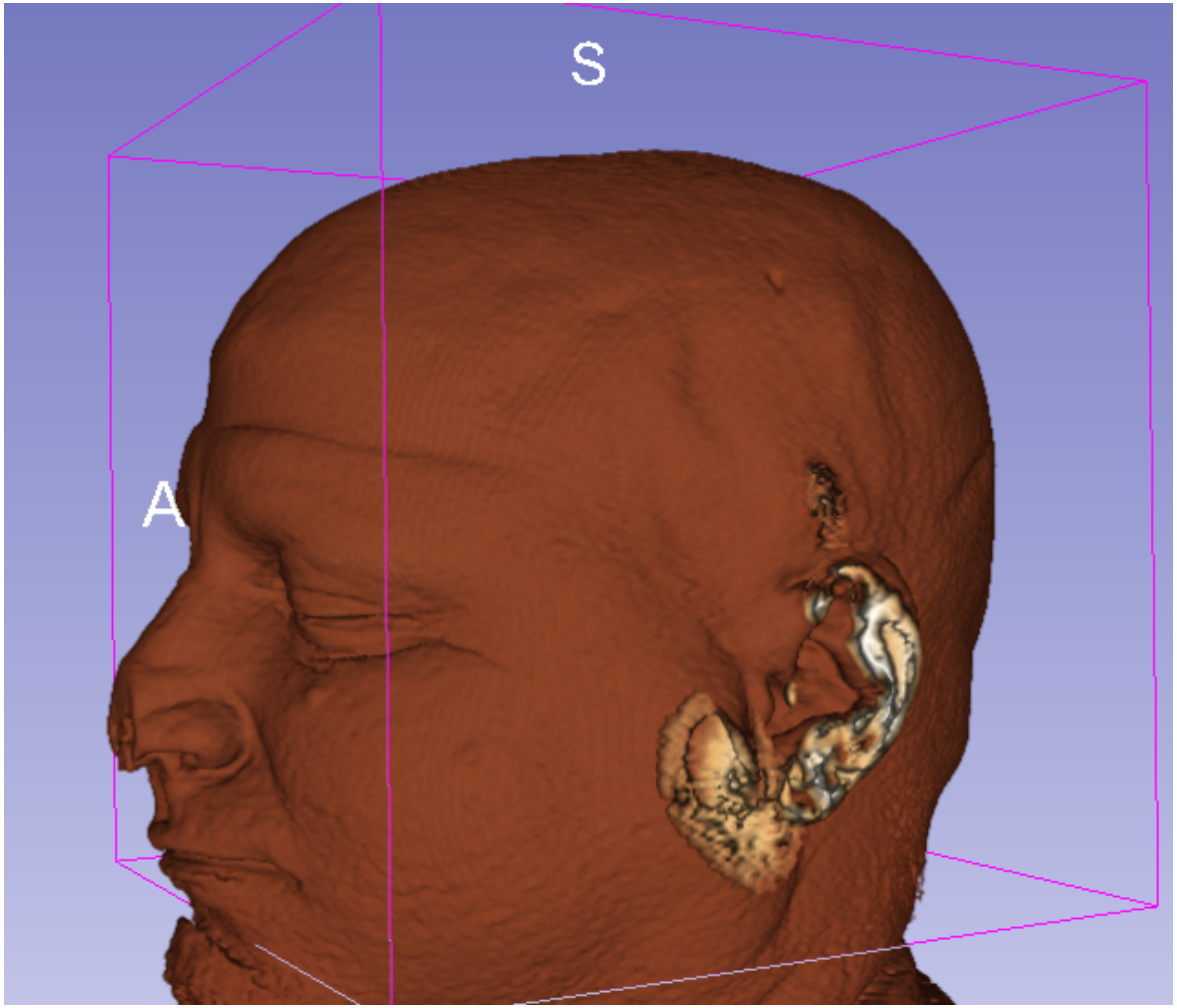
$$L(\mathbf{x}) = \frac{\lambda}{\pi} \cdot \max(-\vec{n}_{\mathbf{x}} \cdot \vec{l}, 0)$$

- Note that:
  - $\vec{n}_{\mathbf{x}}$ needs to be normalized.
  - $\vec{l}$ needs to be normalized.

Volume Rendering: Let there be light



Volume Rendering: Let there be light



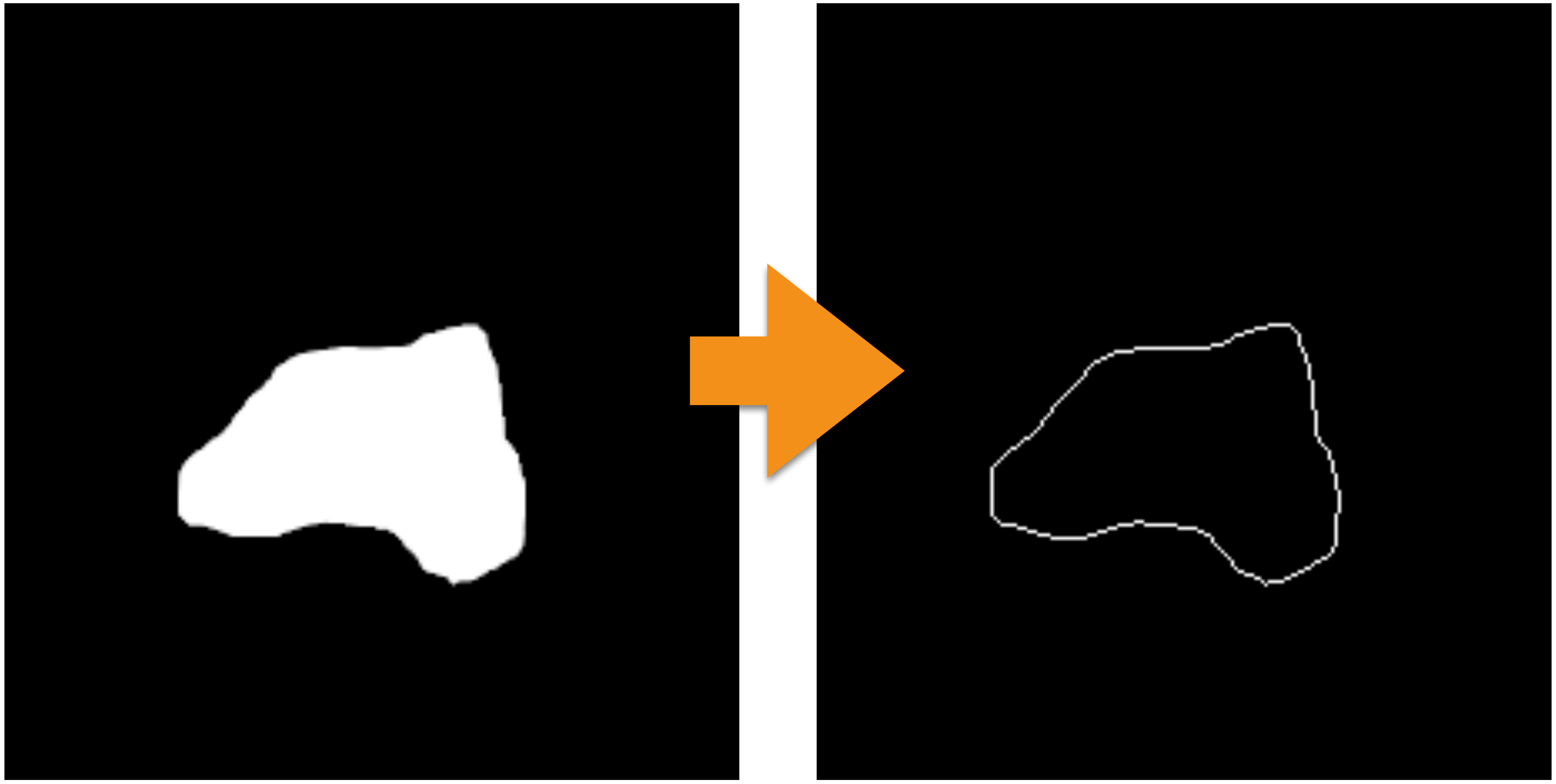
Volume Rendering

- It is a very simple and easy to implement method.
- It is computationally expensive.
  - It works in real-time using a GPU!

3D Points Extraction

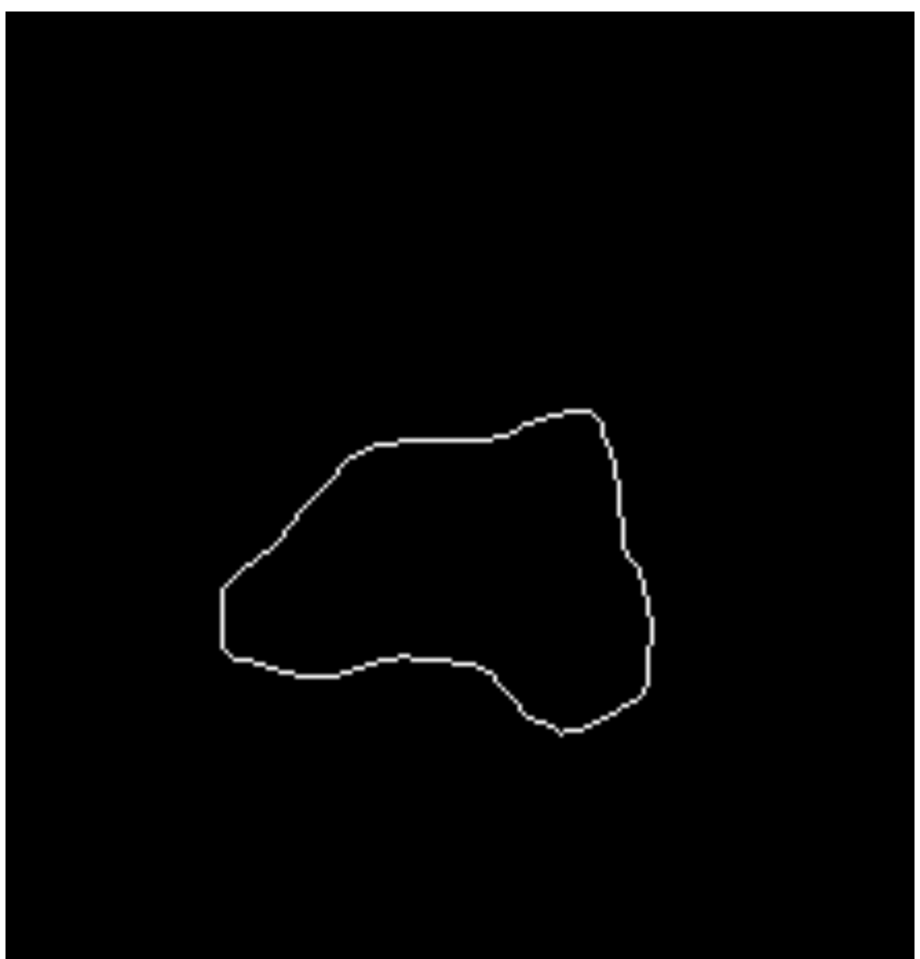
3D Points Extraction

- For each slice of the volume, we compute the edges of the segmented region:



3D Points Extraction

- For each white pixel in the edge with coordinates (u,v) at the i-th slice, we compute its 3D position as



$$m = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} u \cdot k_u \\ v \cdot k_v \\ i \cdot k_w \end{bmatrix}$$

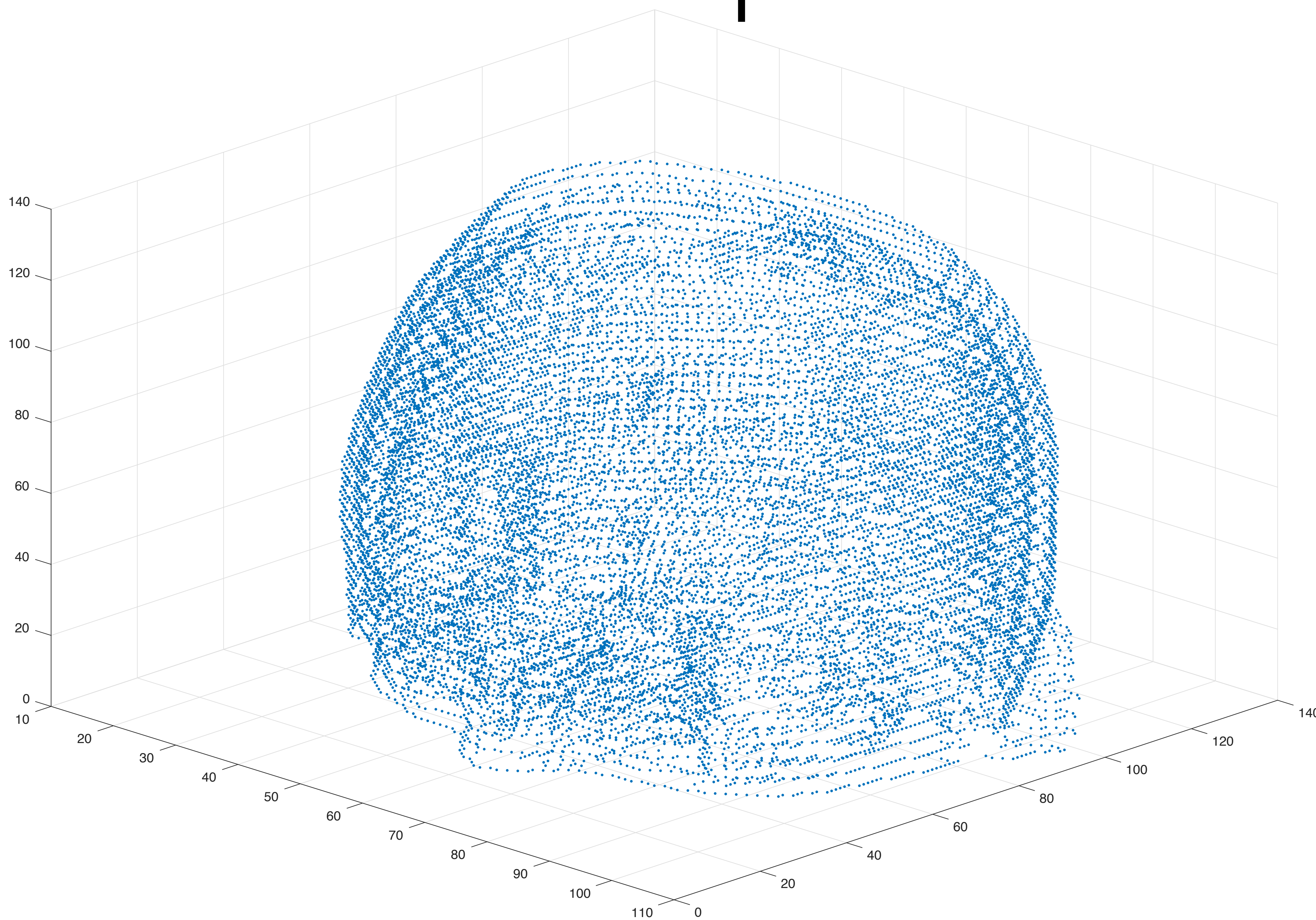
k_u is the pixel's width in mm
k_v is the pixel's height in mm
k_w is the distance between slices in mm

3D Points Extraction

- How do we compute the normal at the point?
- It is simply the negative value of the gradient of the volume in that point!

$$-\vec{\nabla} V$$

3D Points Extraction Example



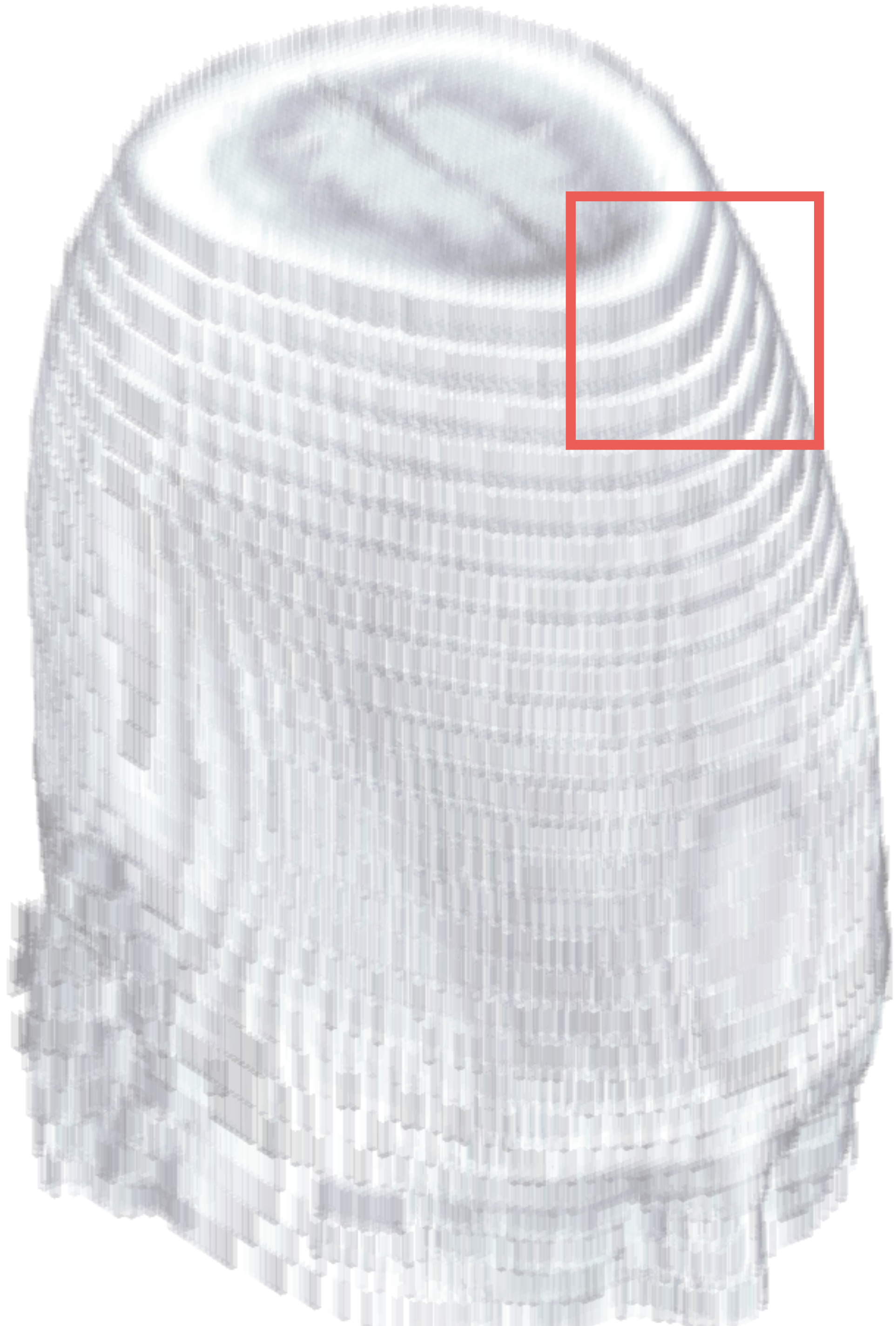
3D Mesh Extraction

A Very Stupid Algorithm:

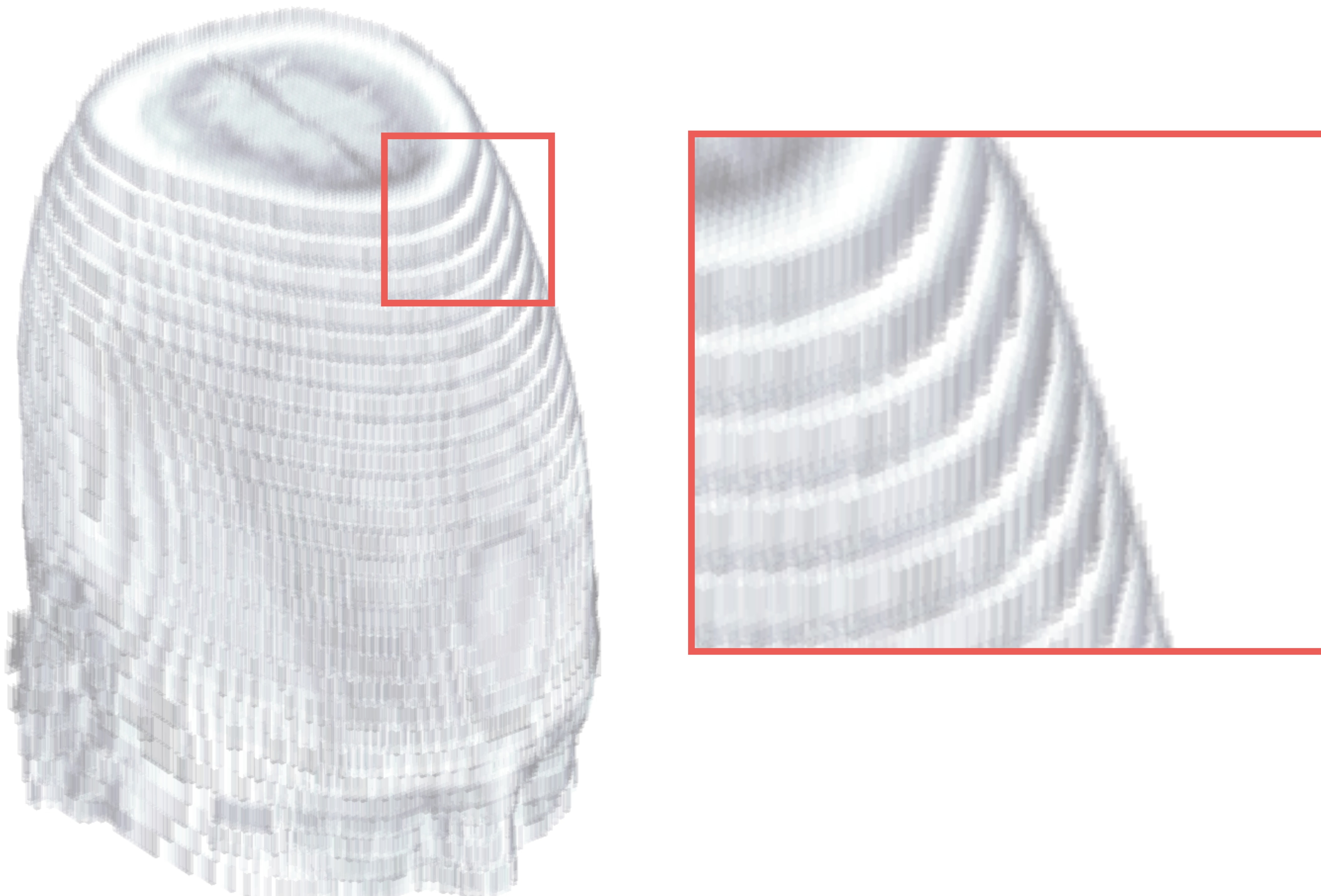
For each extracted point, create a cube…

A Very Stupid Algorithm Example

A Very Stupid Algorithm Example



A Very Stupid Algorithm Example



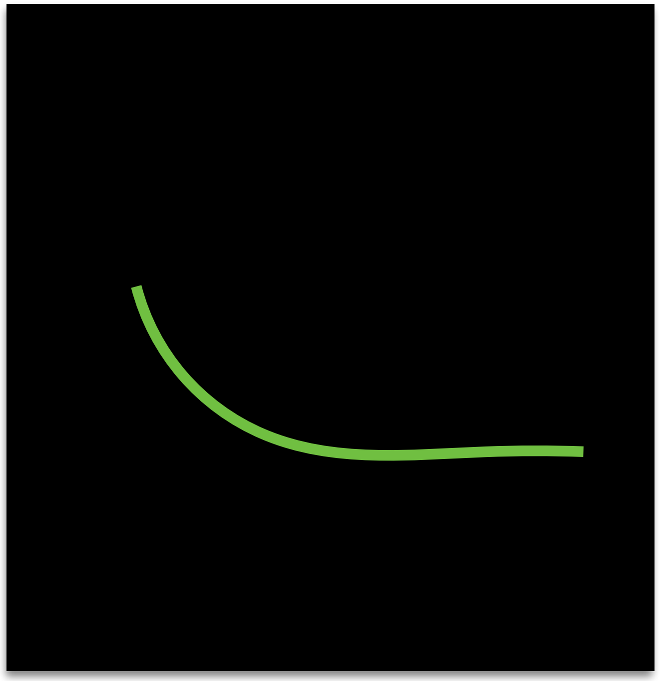
I guess, we can do better than this!

Connecting the dots…

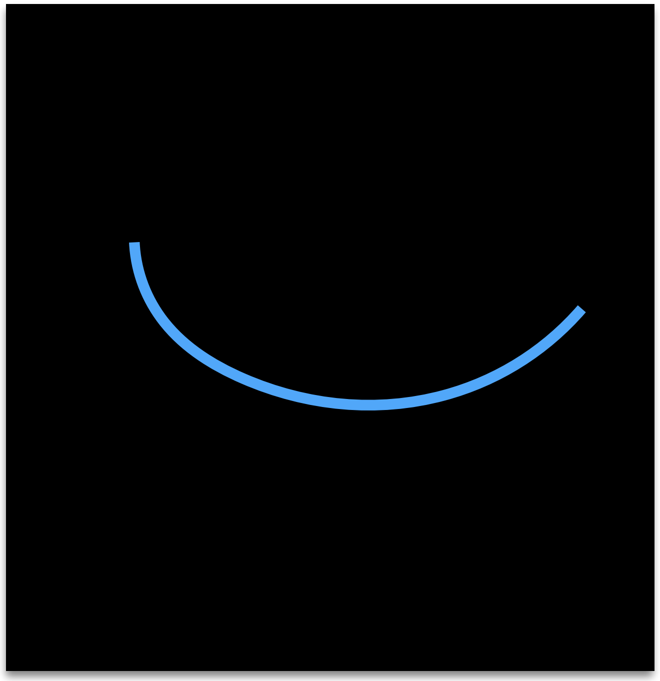
Edges Triangulation

- As the first step, we extract the edges from each slice in the volume.
- We save the connectivity of points belonging to the same edge —> “parametric curve”
    - We may have more curves per slice!

Edges Triangulation

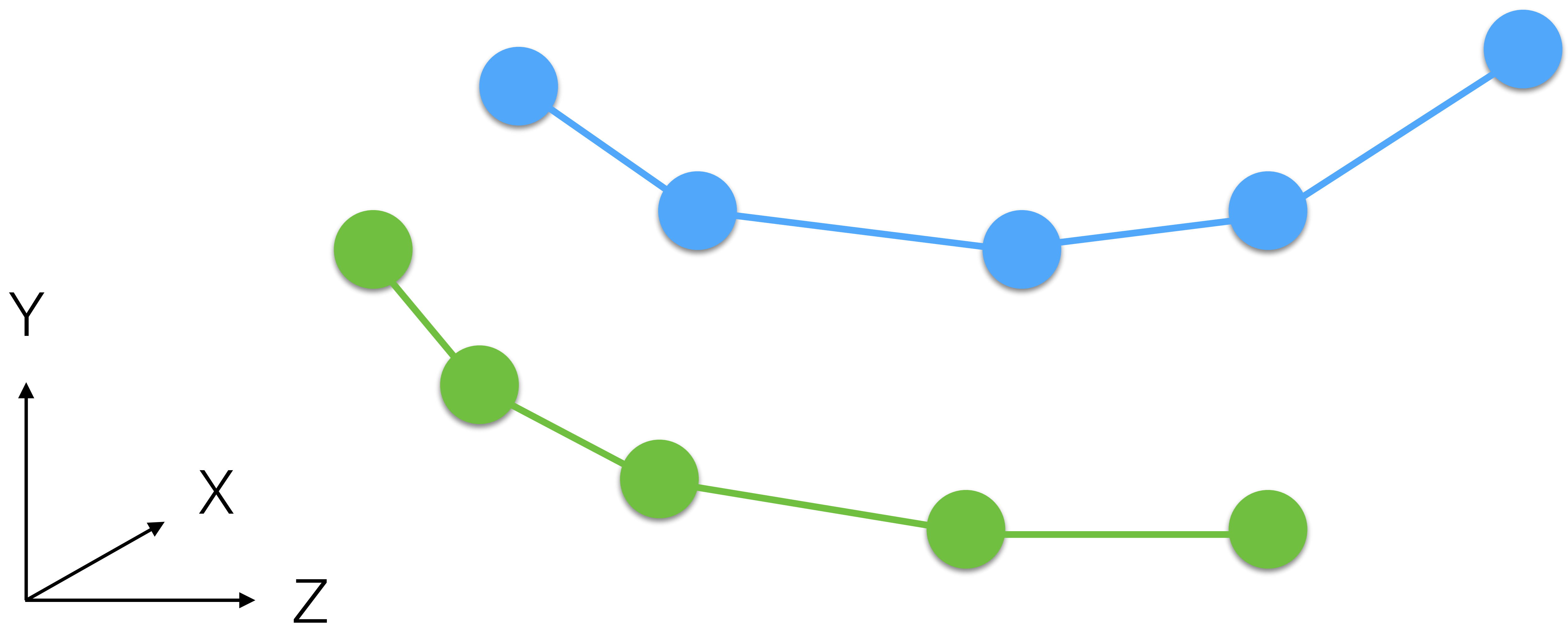


Slice 1

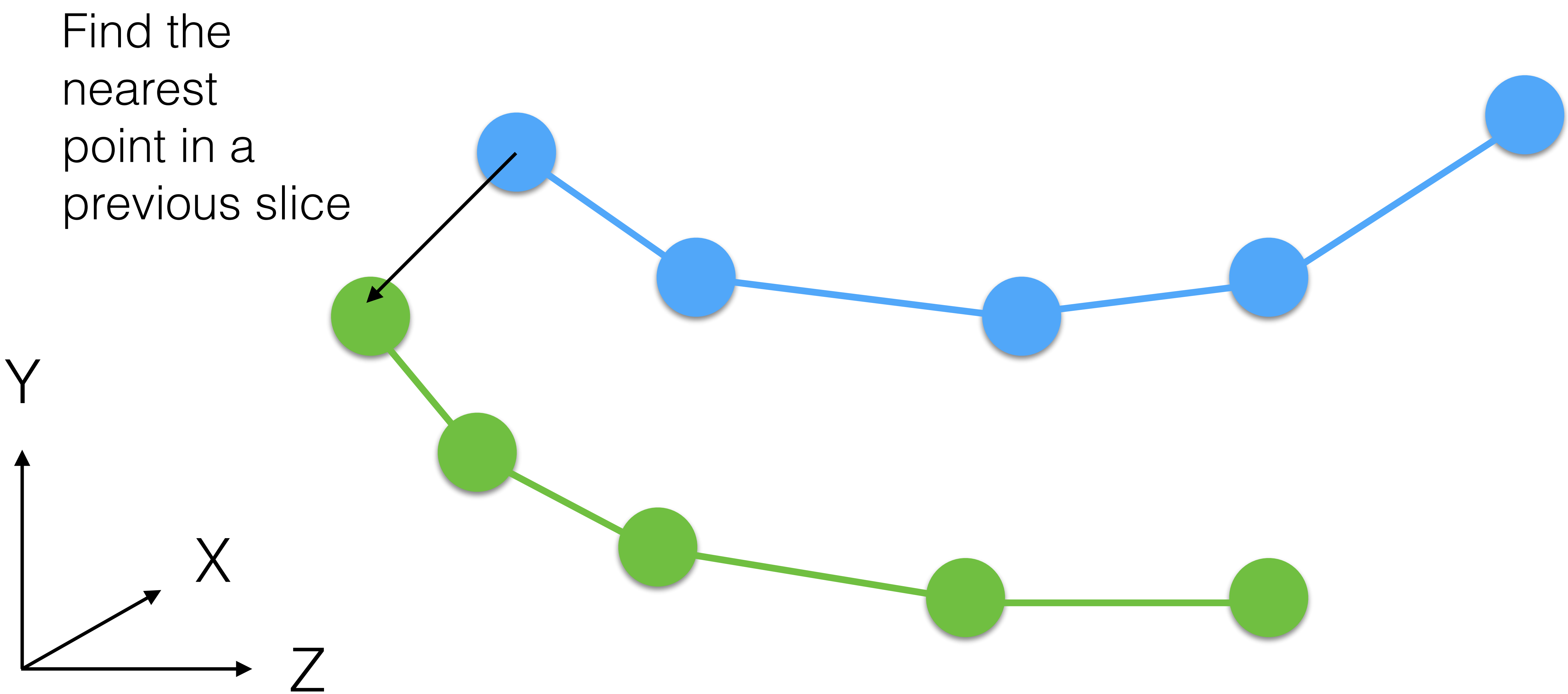


Slice 2

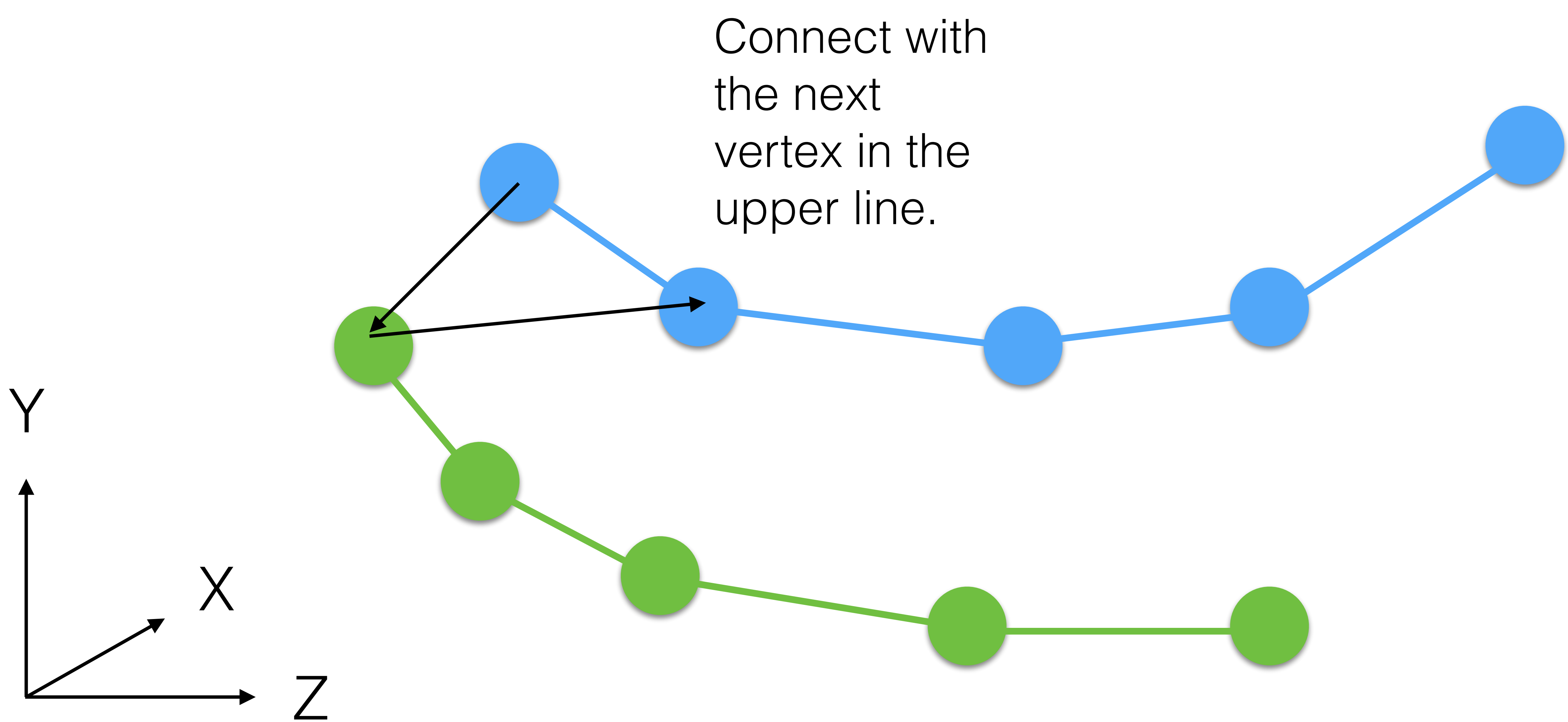
Edges Triangulation



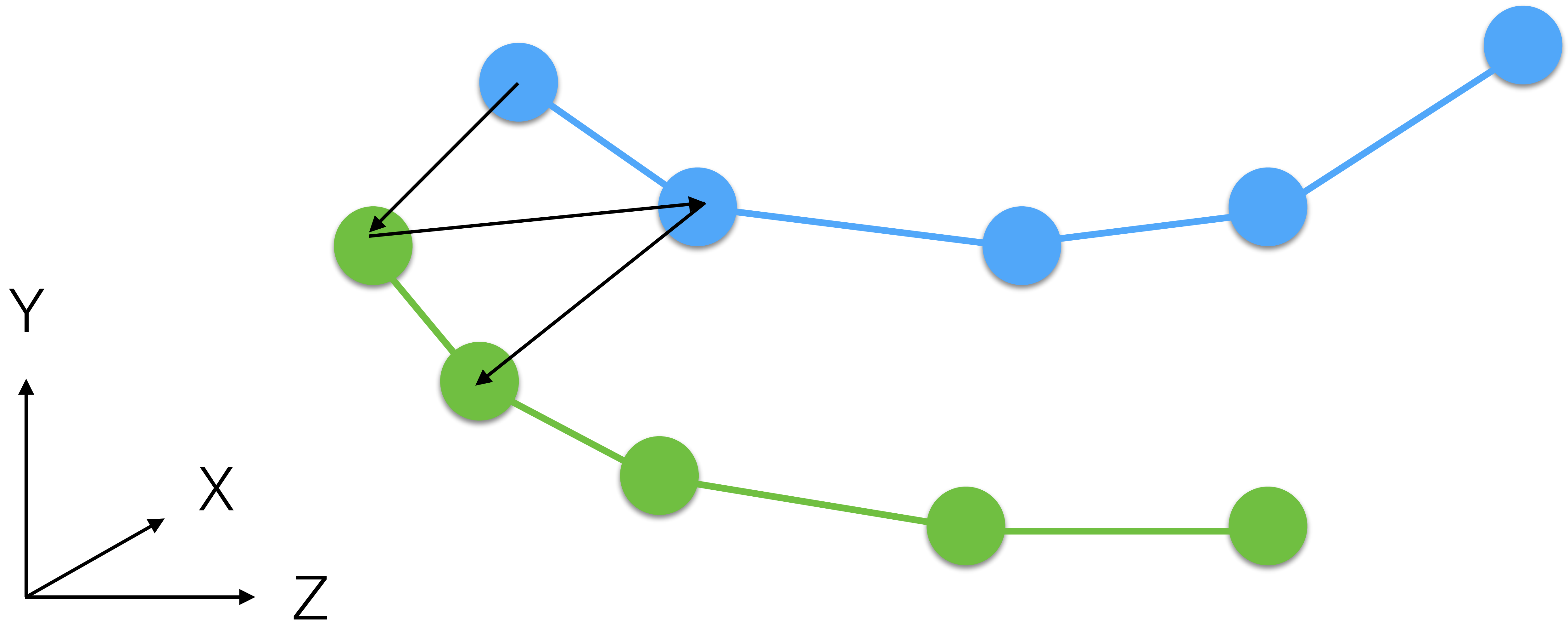
Edges Triangulation



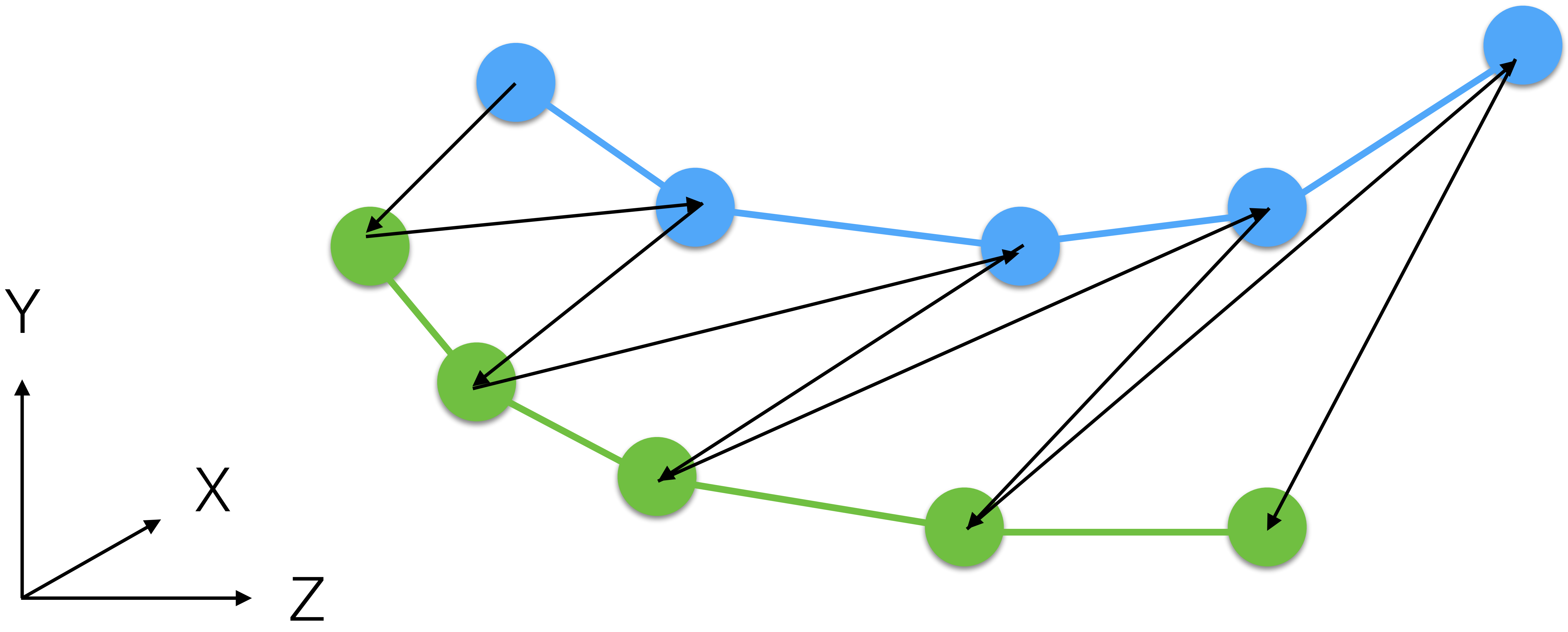
Edges Triangulation



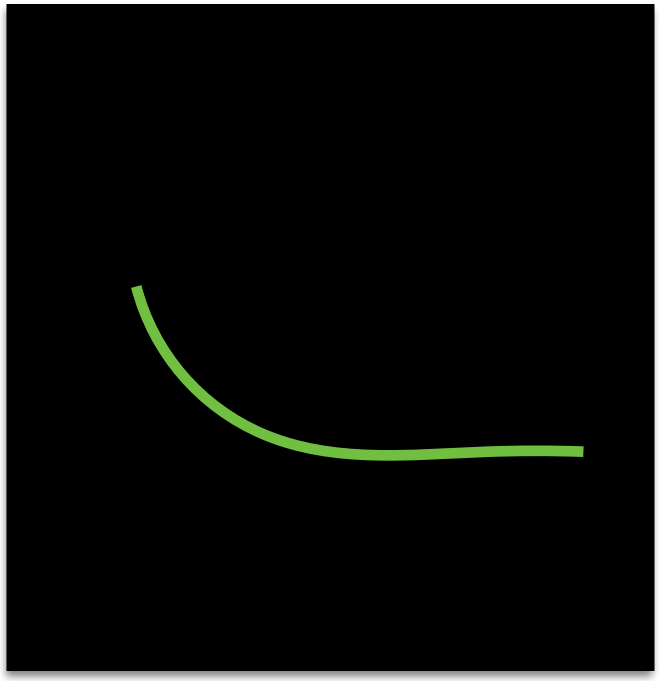
Edges Triangulation



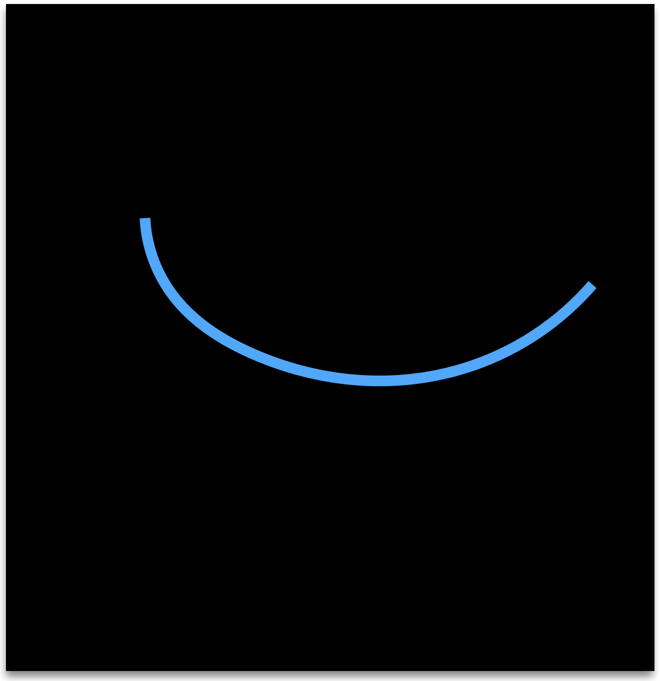
Edges Triangulation



Edges Triangulation: Fail Case

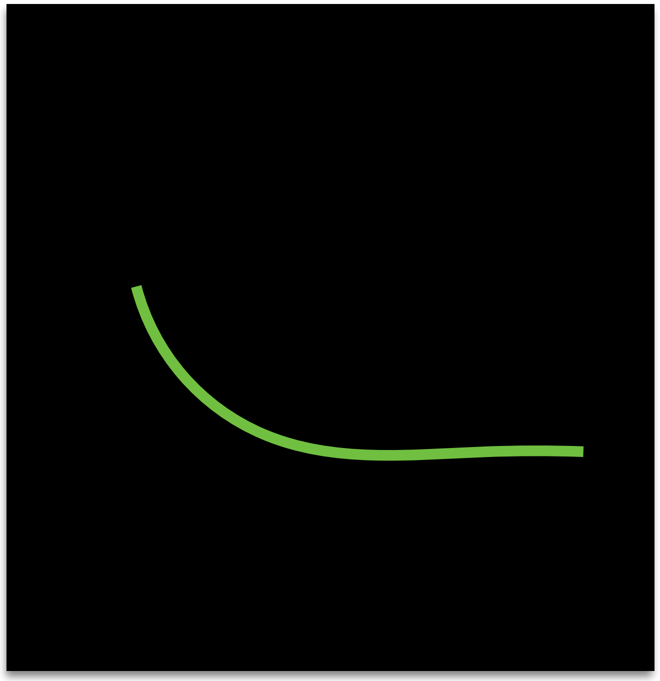


Slice 1

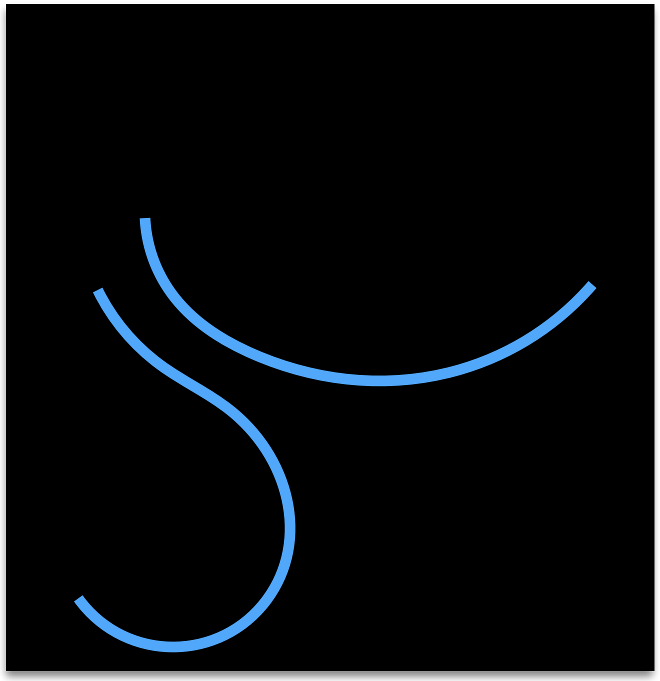


Slice 2

Edges Triangulation: Fail Case

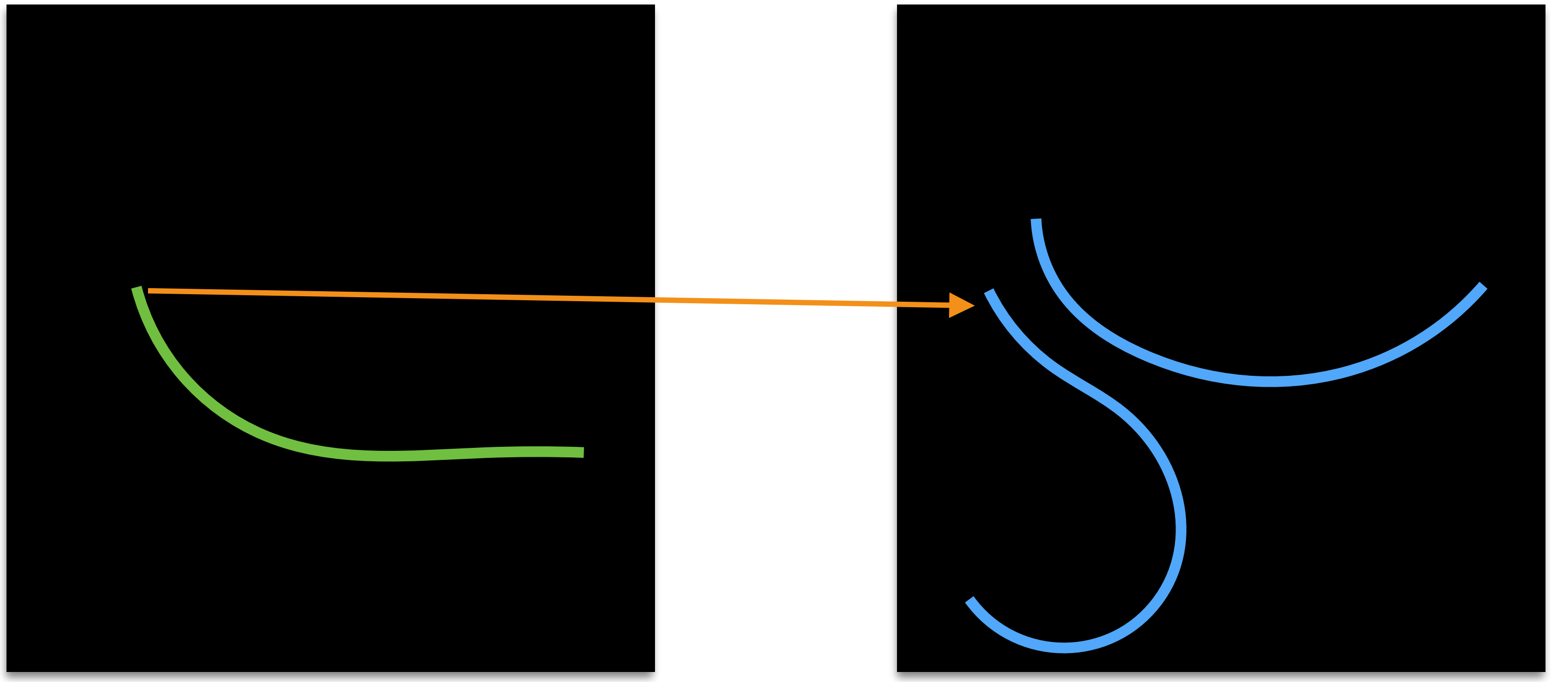


Slice 1



Slice 2

Edges Triangulation: Fail Case



Slice 1 Slice 2

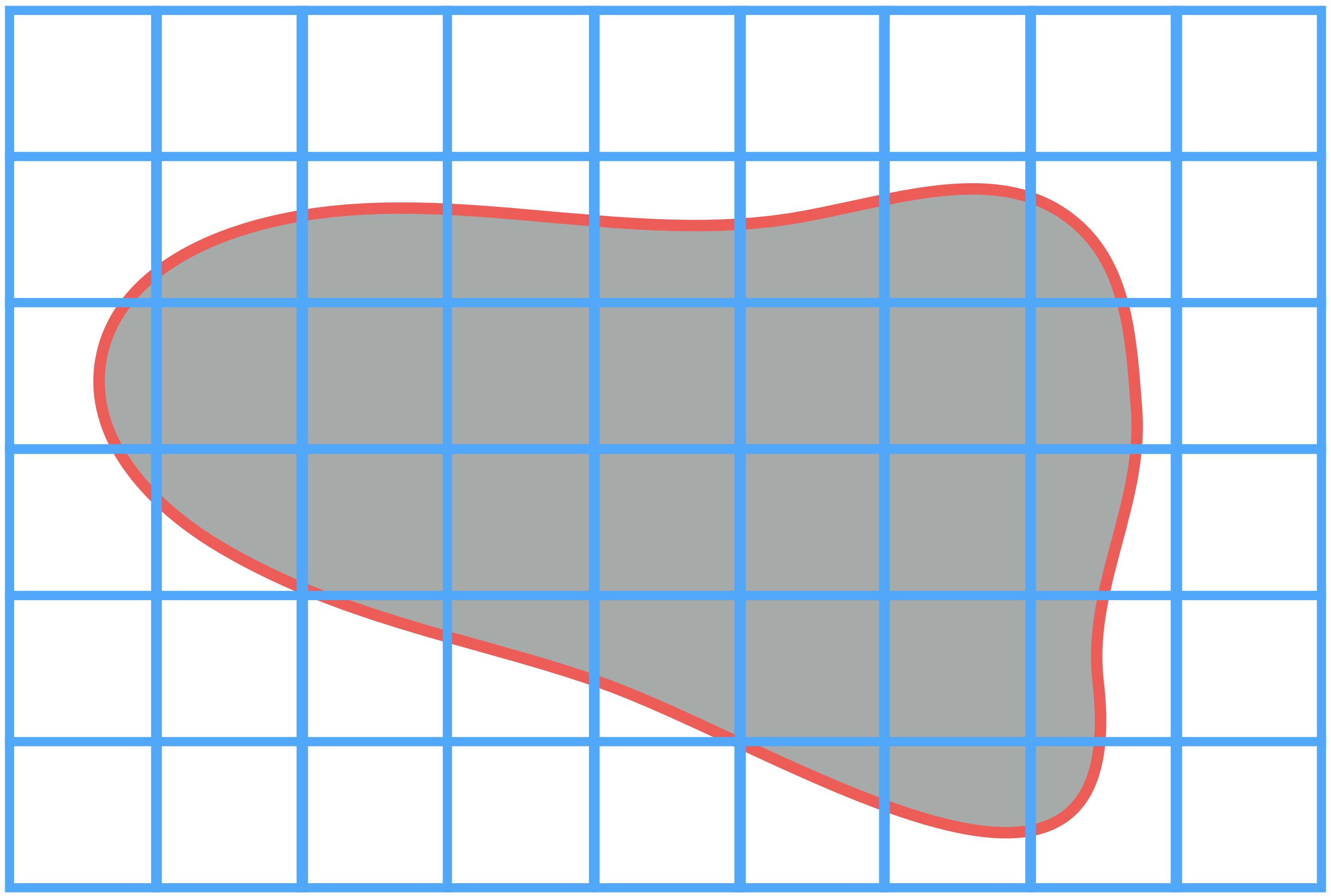
Edges Triangulation

- It works because we have a previously known connectivity.
- It works only for a binary segmentation mask. No multiple objects!
- Quality of triangles is pretty poor!
- We cannot close the mesh, i.e., it is not watertight.

Marching Cubes

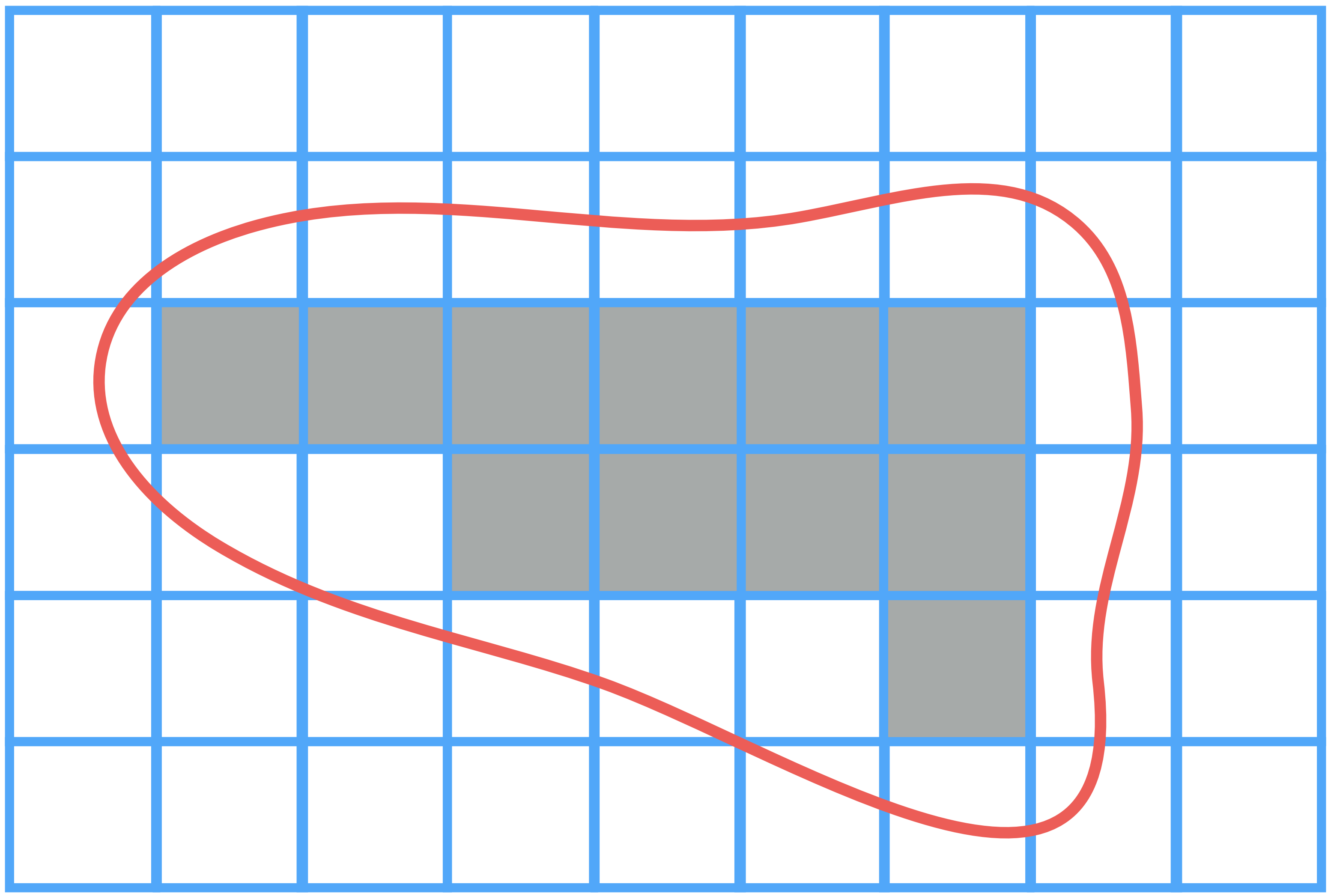
Let’s start in 2D

Marching Squares



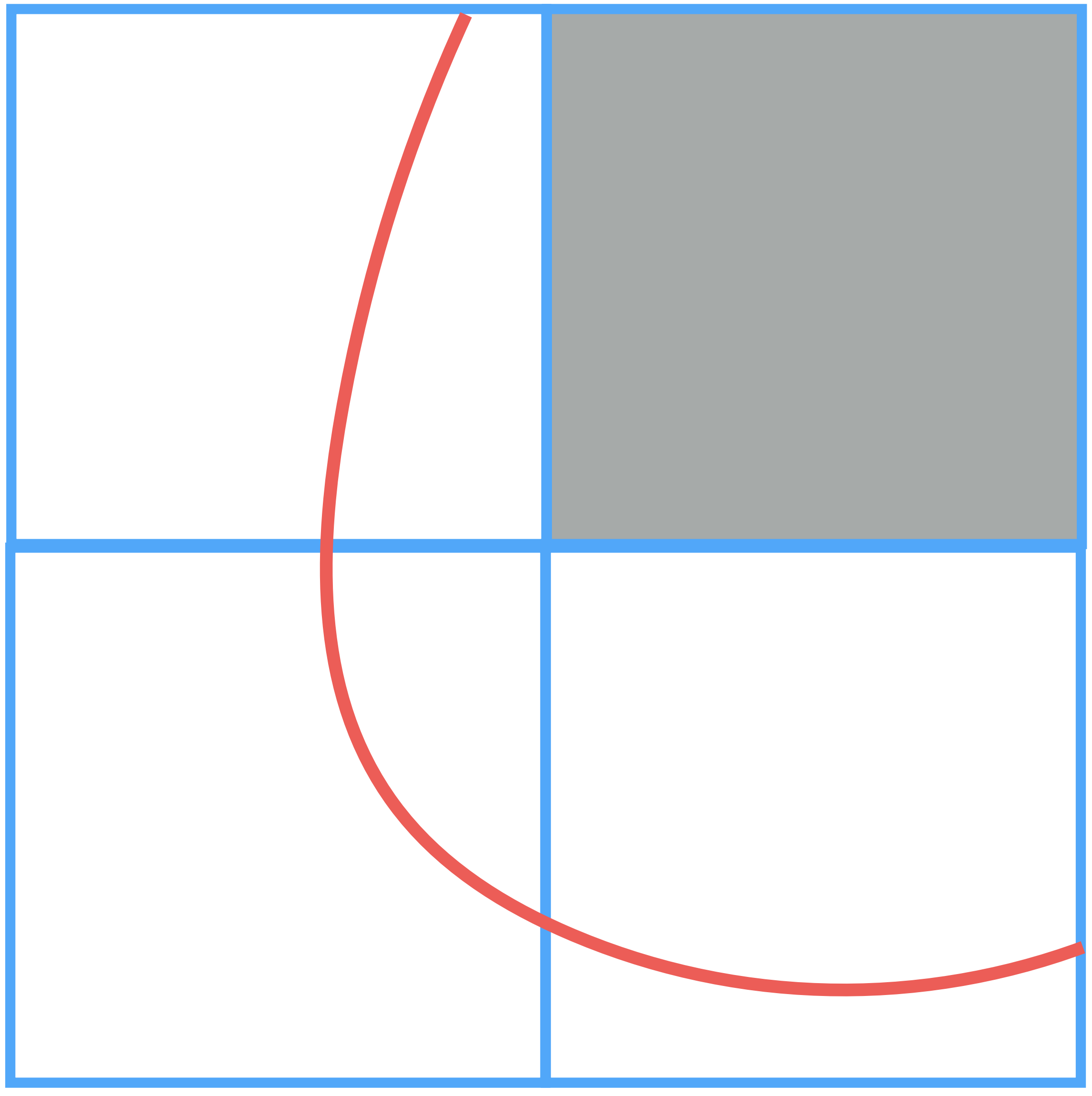
$$f(x, y) = 0$$

Marching Squares



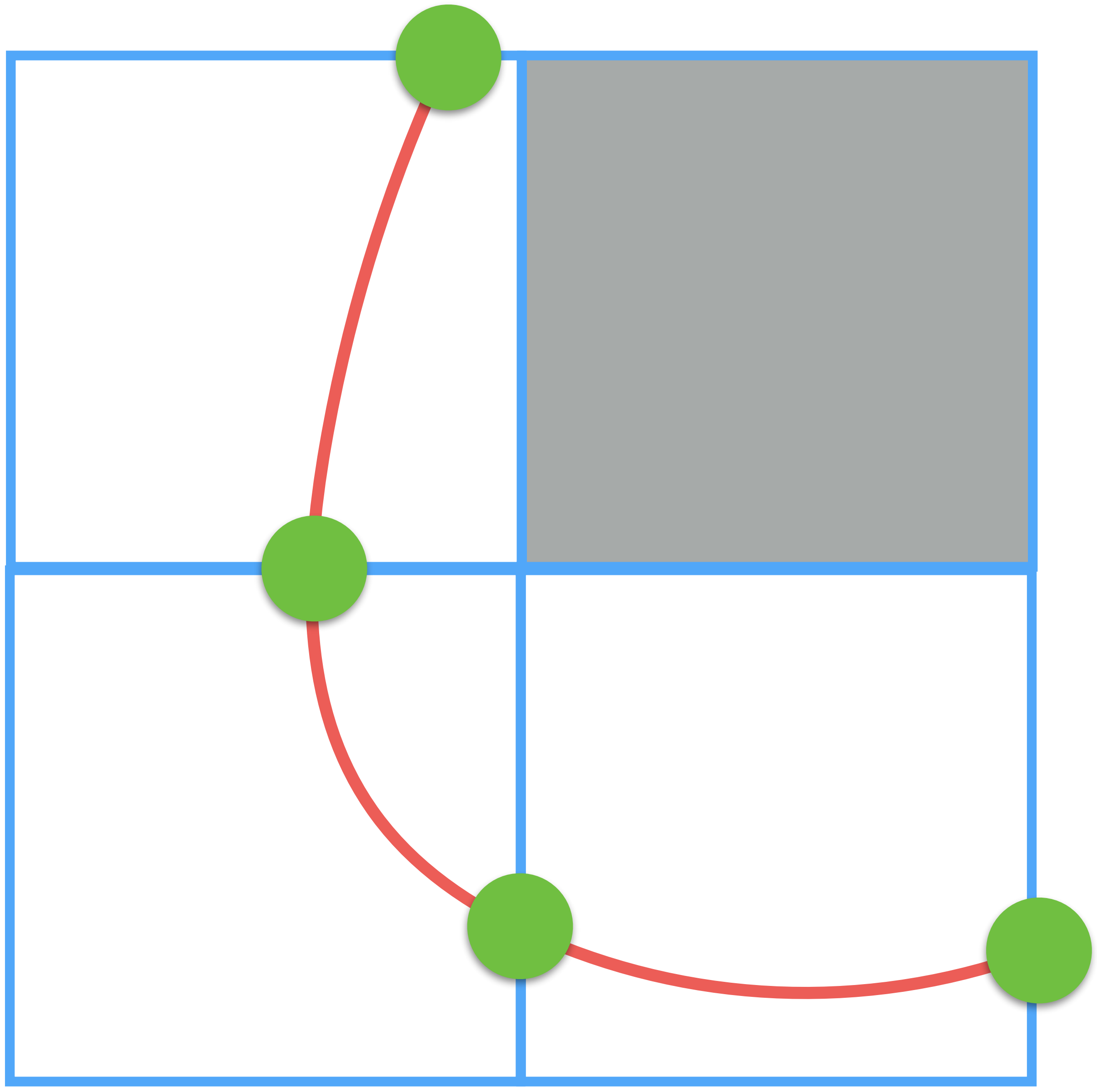
$$f(x, y) = 0$$

Marching Squares

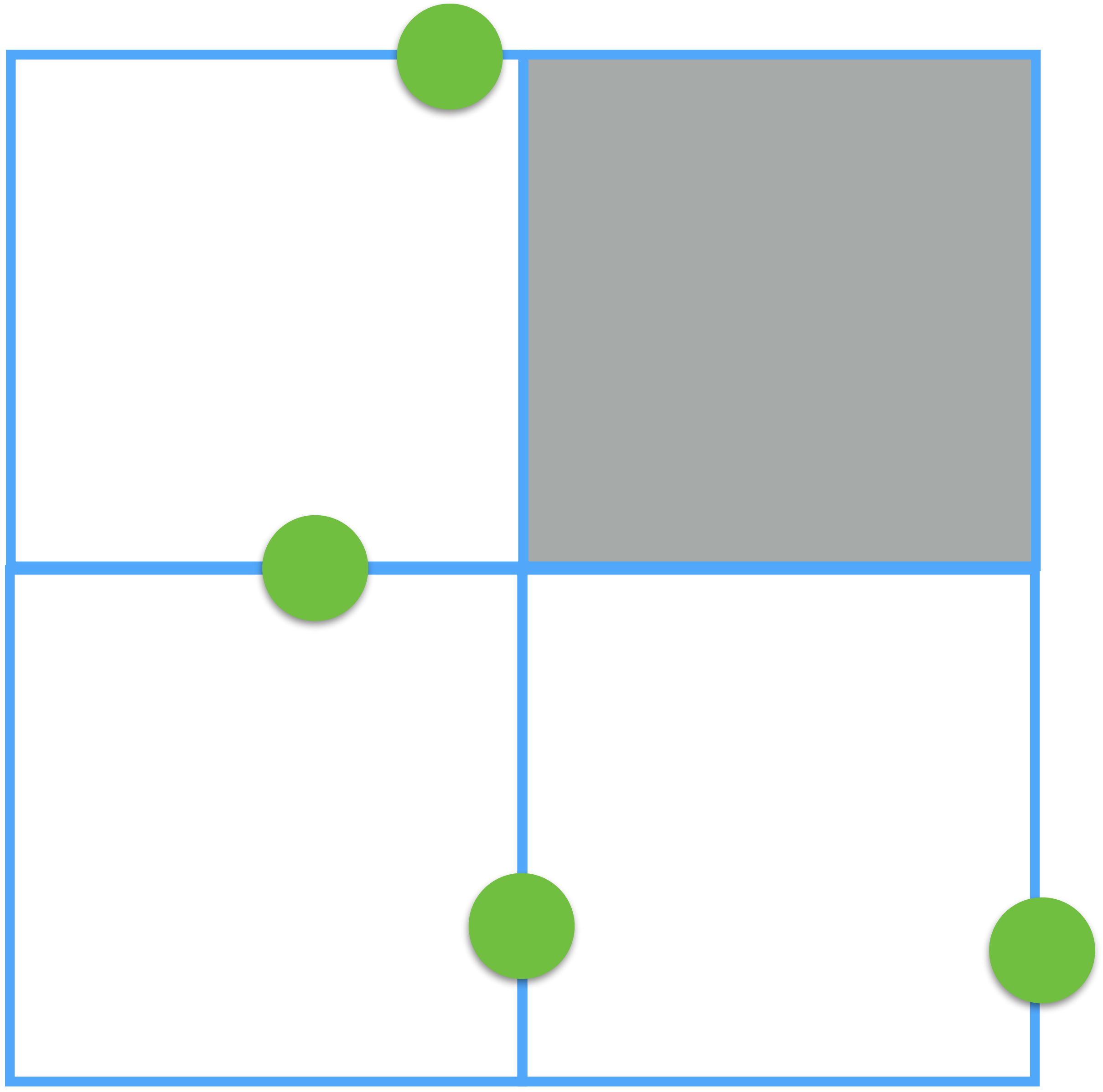


Marching Squares

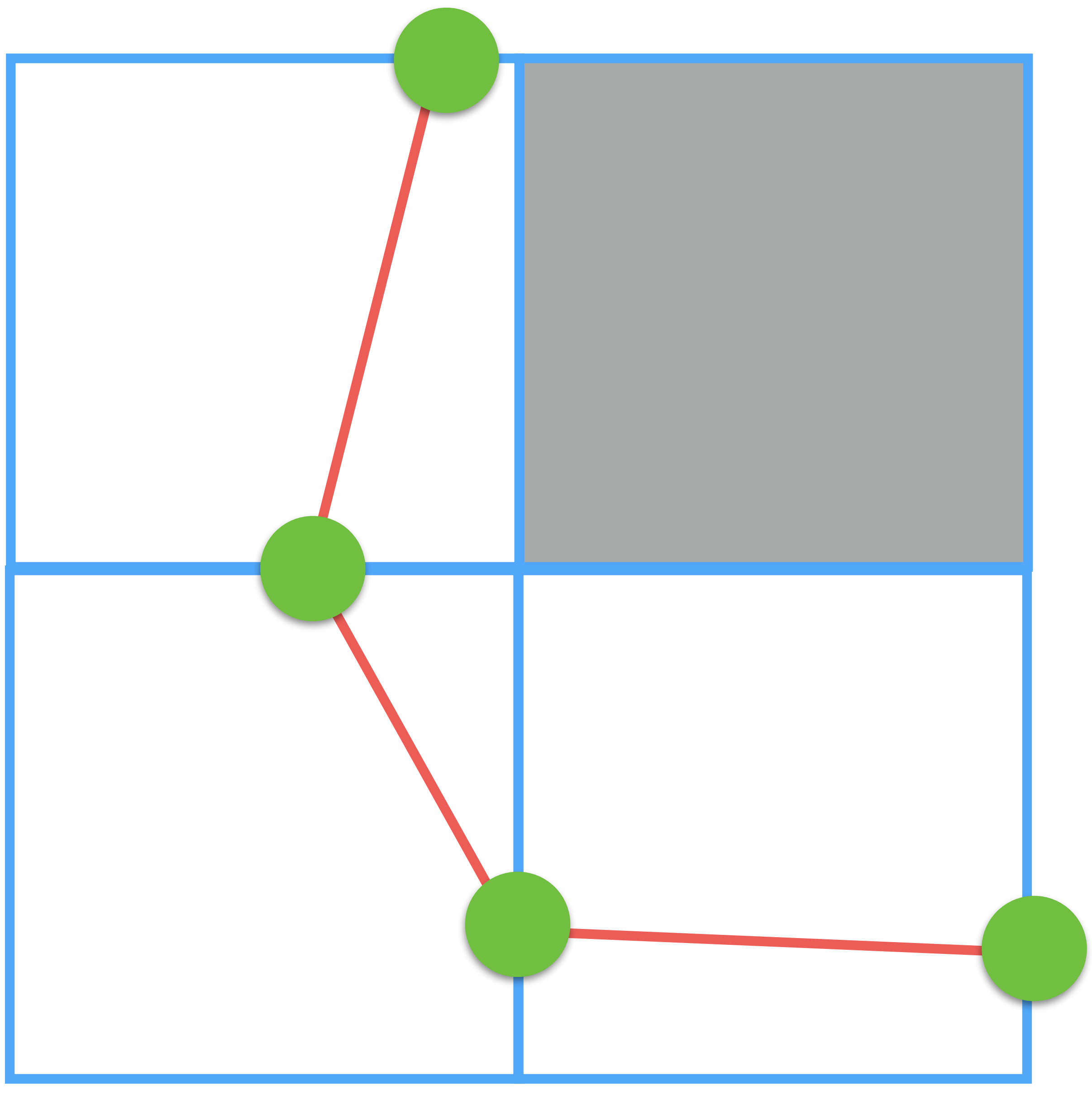
$$f(x, y) = 0$$



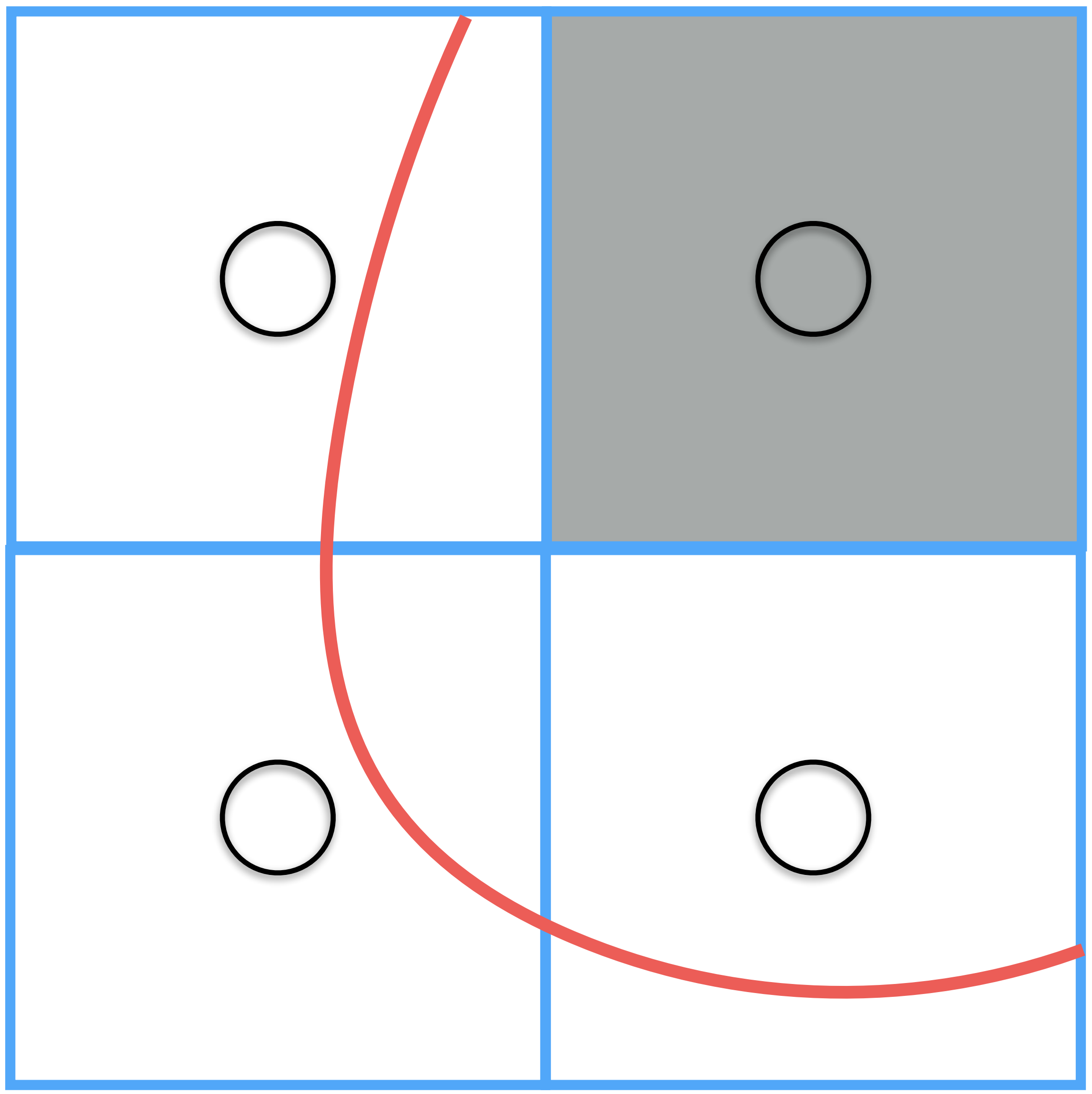
Marching Squares



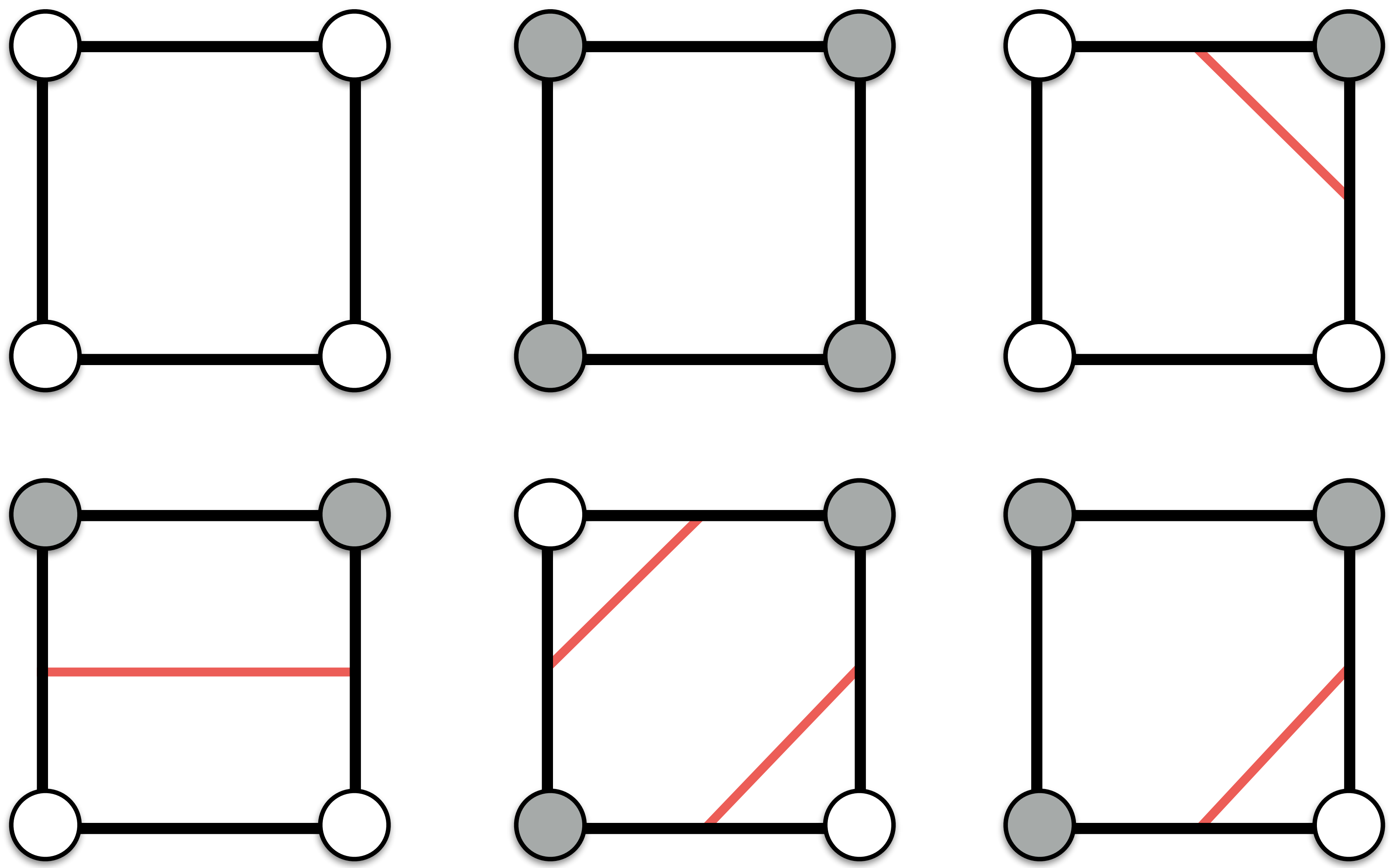
Marching Squares



Marching Squares



Marching Squares: Cases

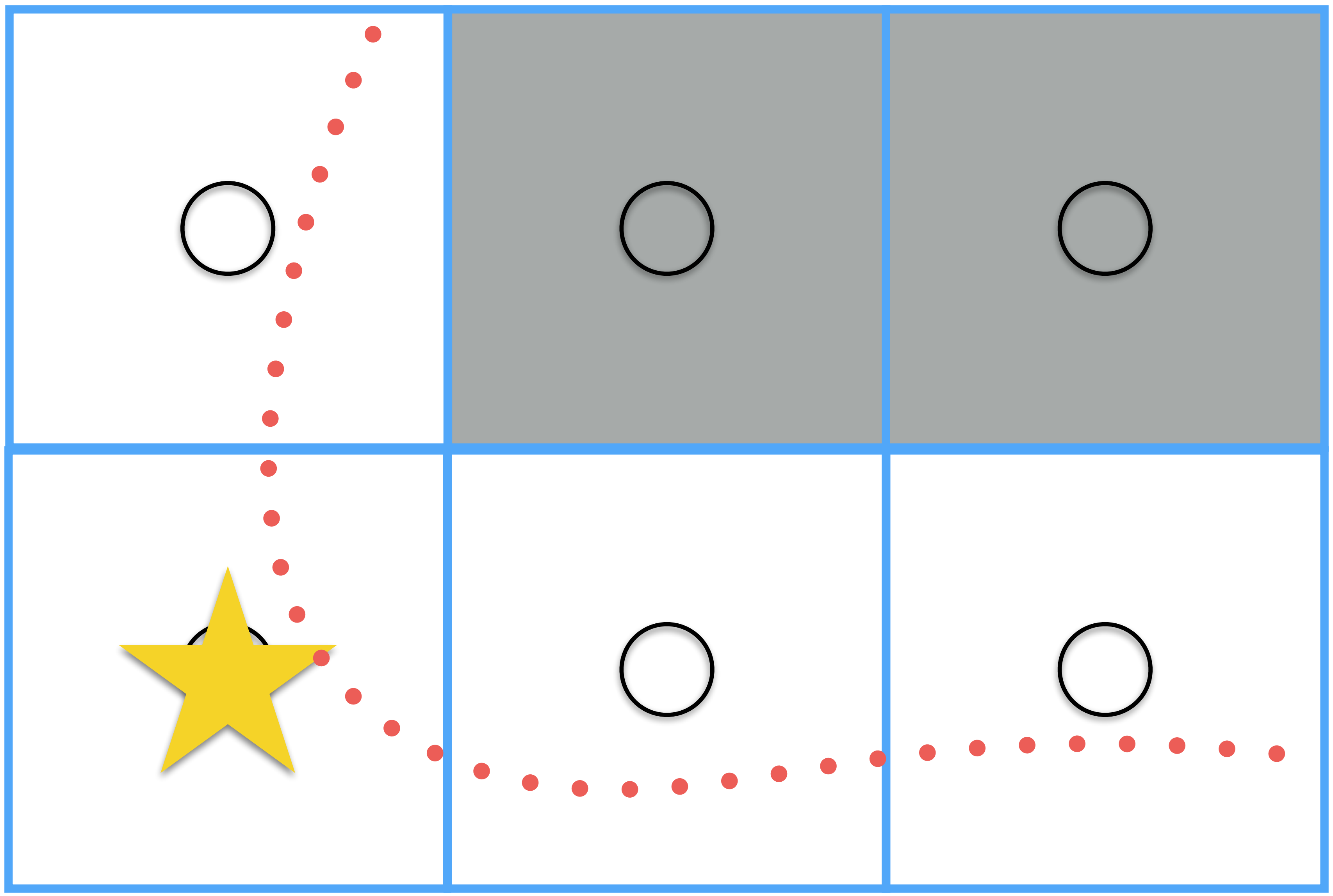


There are in total 16 (2^4) configurations, the other ones can be computed by rotating or reflecting these.

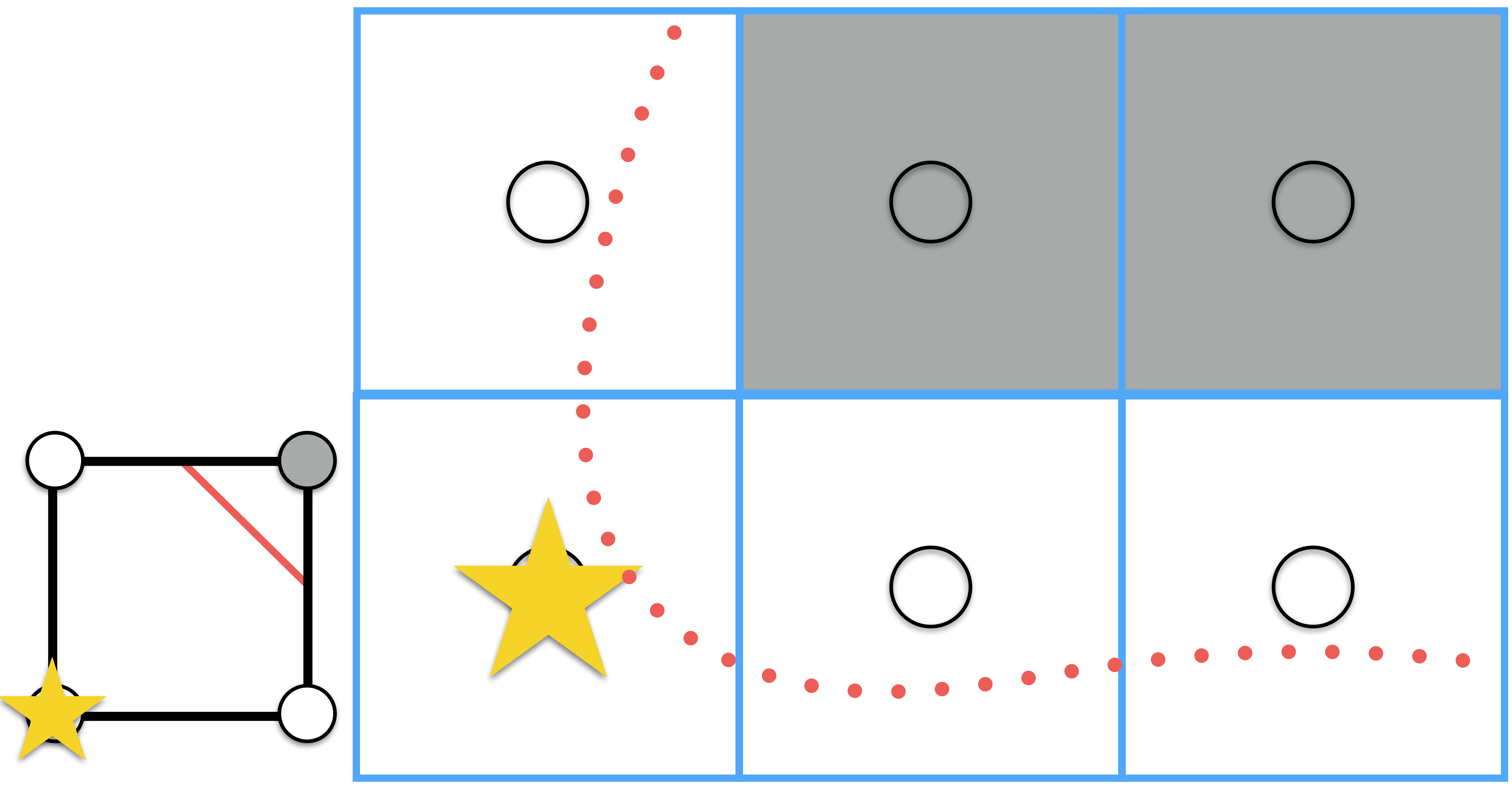
Marching Squares

- 1st pass: For each square (“*we march*”):
  - We determine if it is fully inside (1) or outside the curve (0).
- 2nd pass: For each square:
  - We compute the configuration of the current square.
  - We fetch from the table of configurations our case.
  - We place the line for that case in the current square.

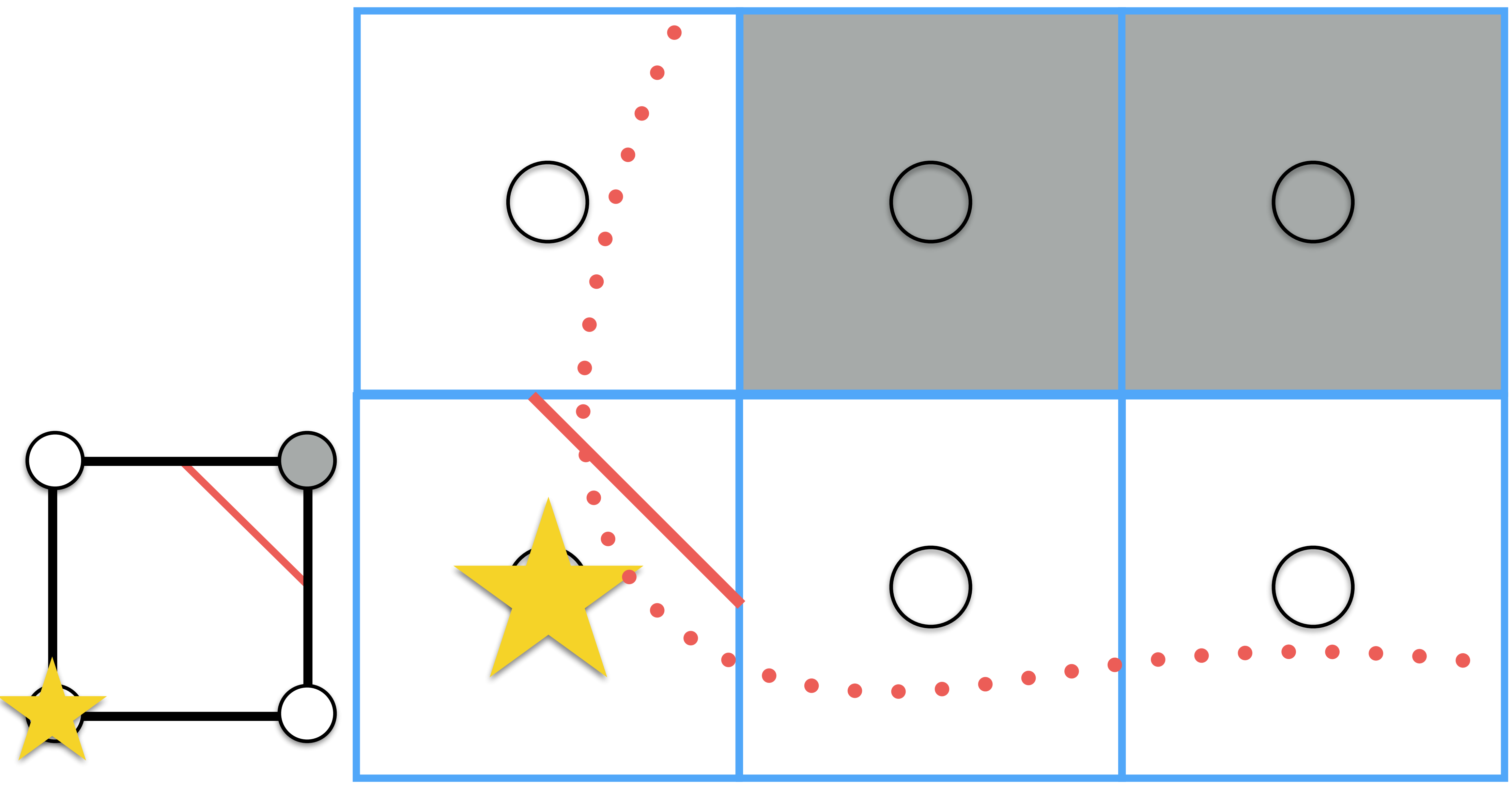
Marching Squares Example



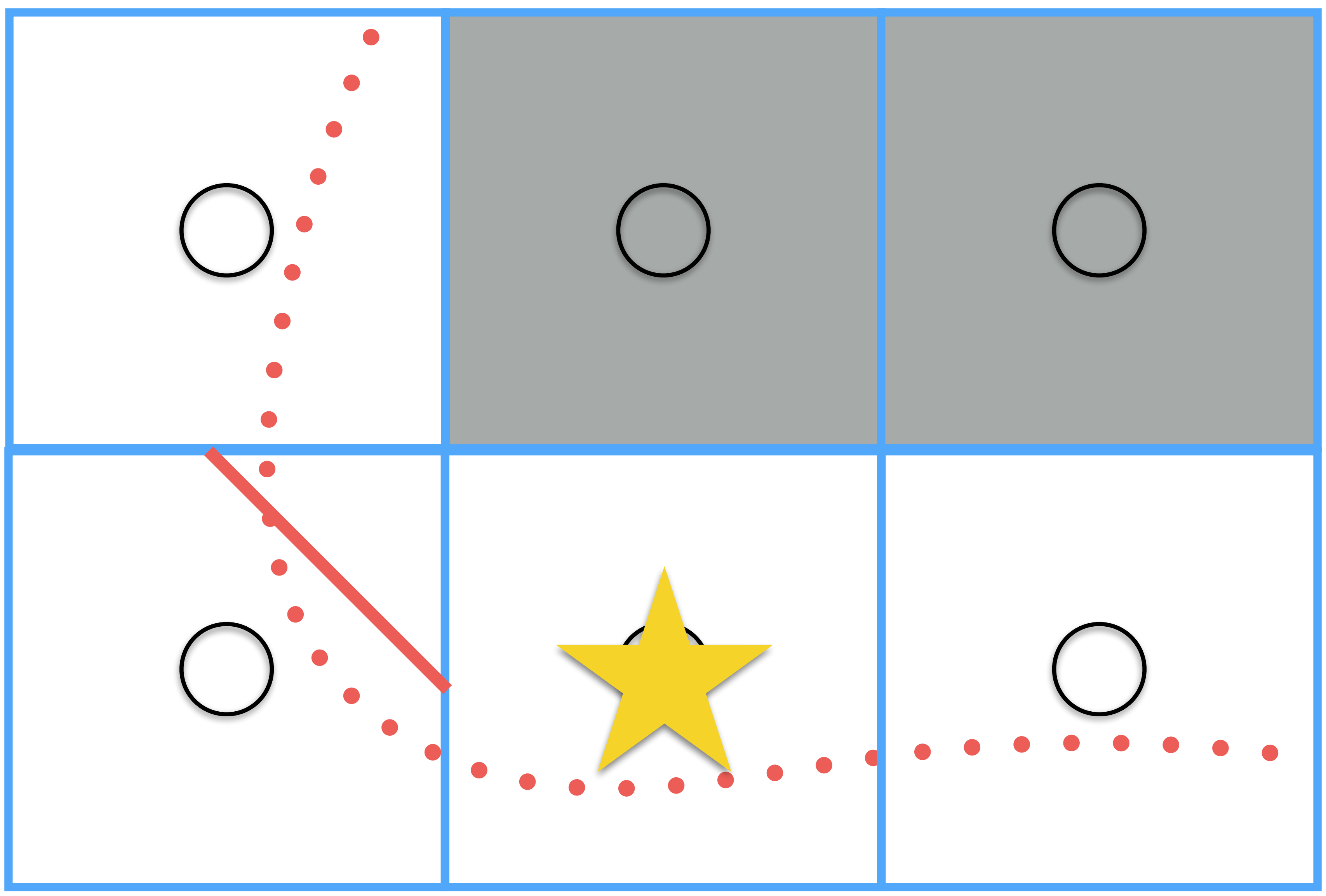
Marching Squares Example



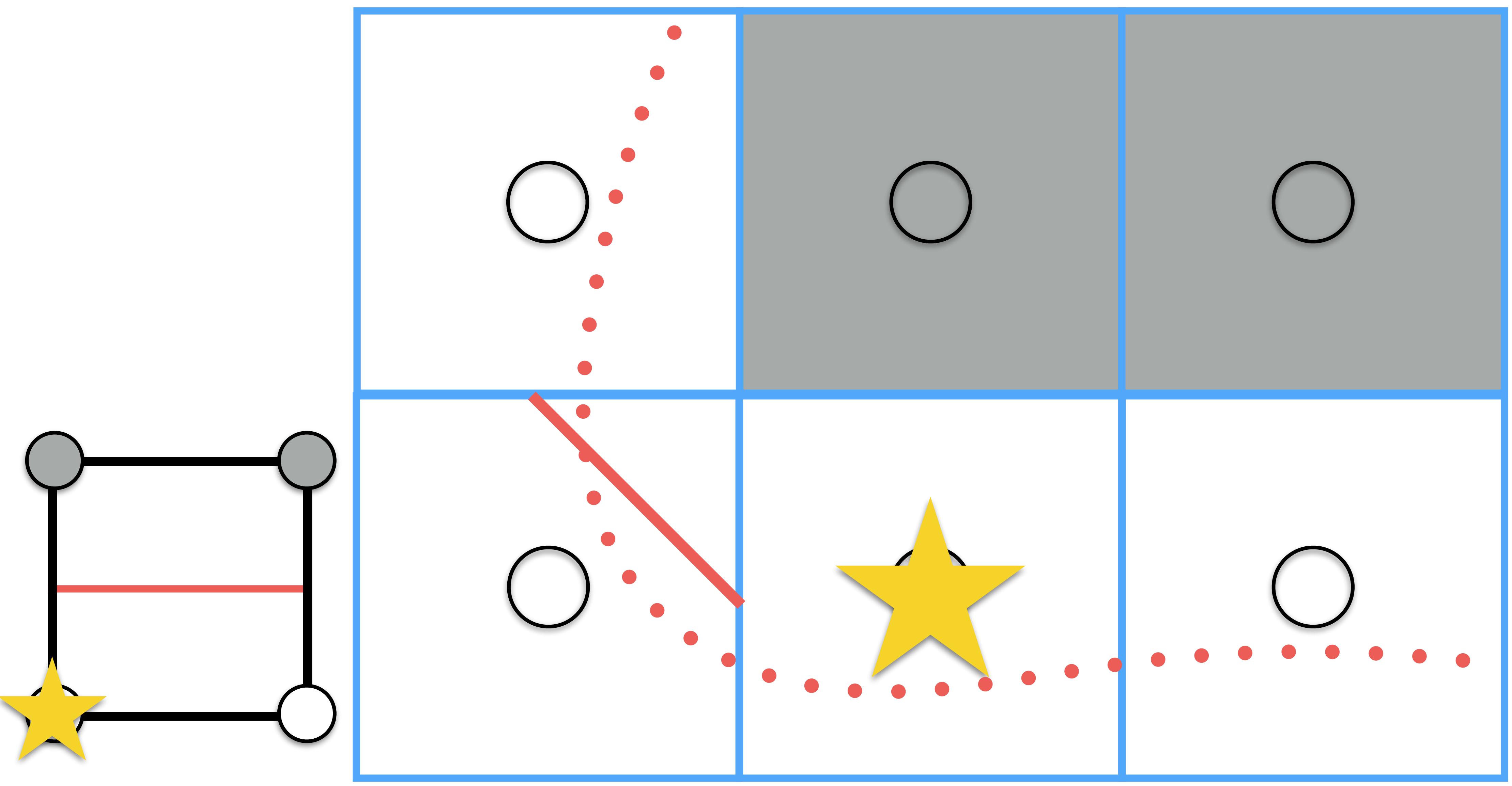
Marching Squares Example



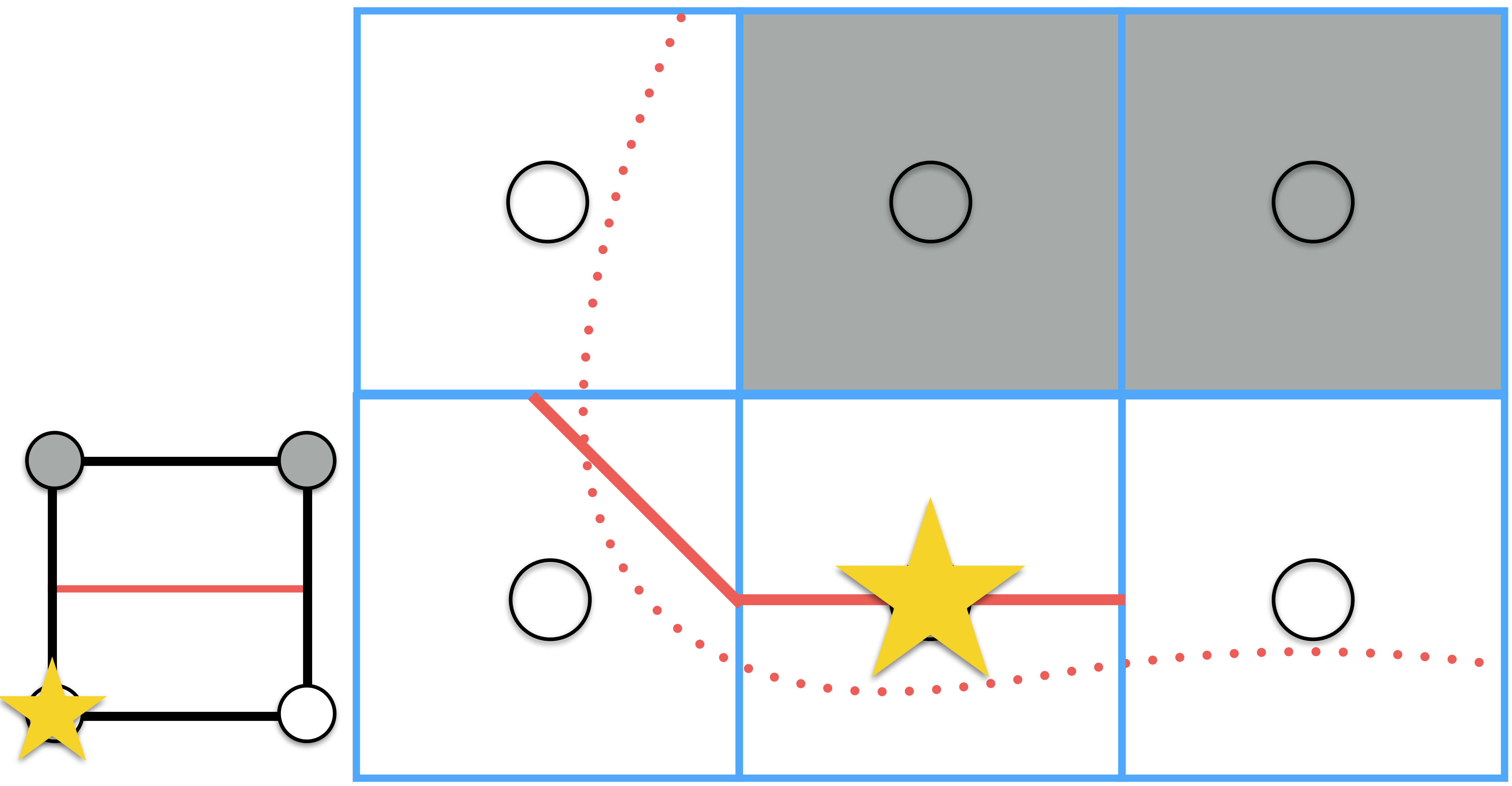
Marching Squares Example



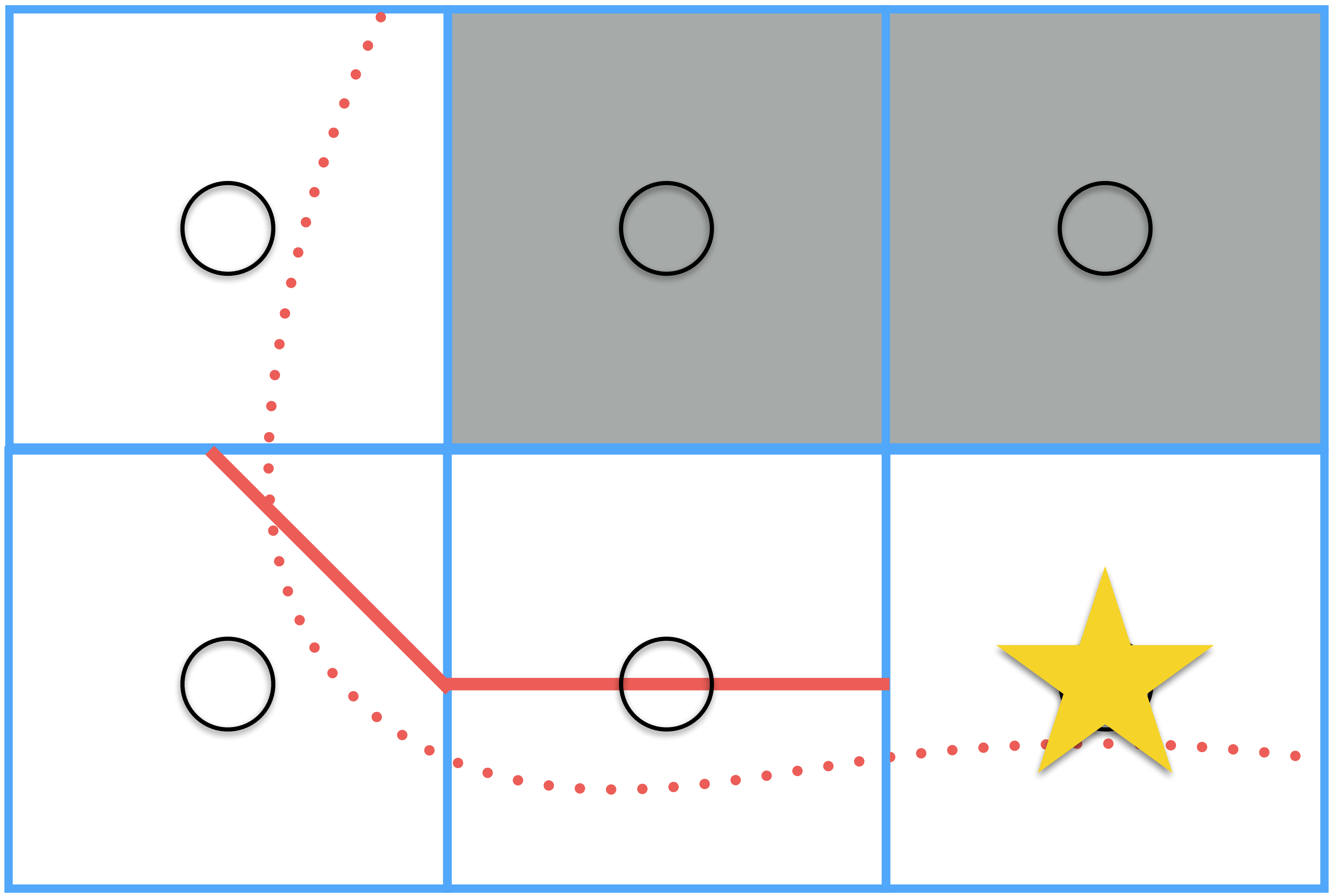
Marching Squares Example



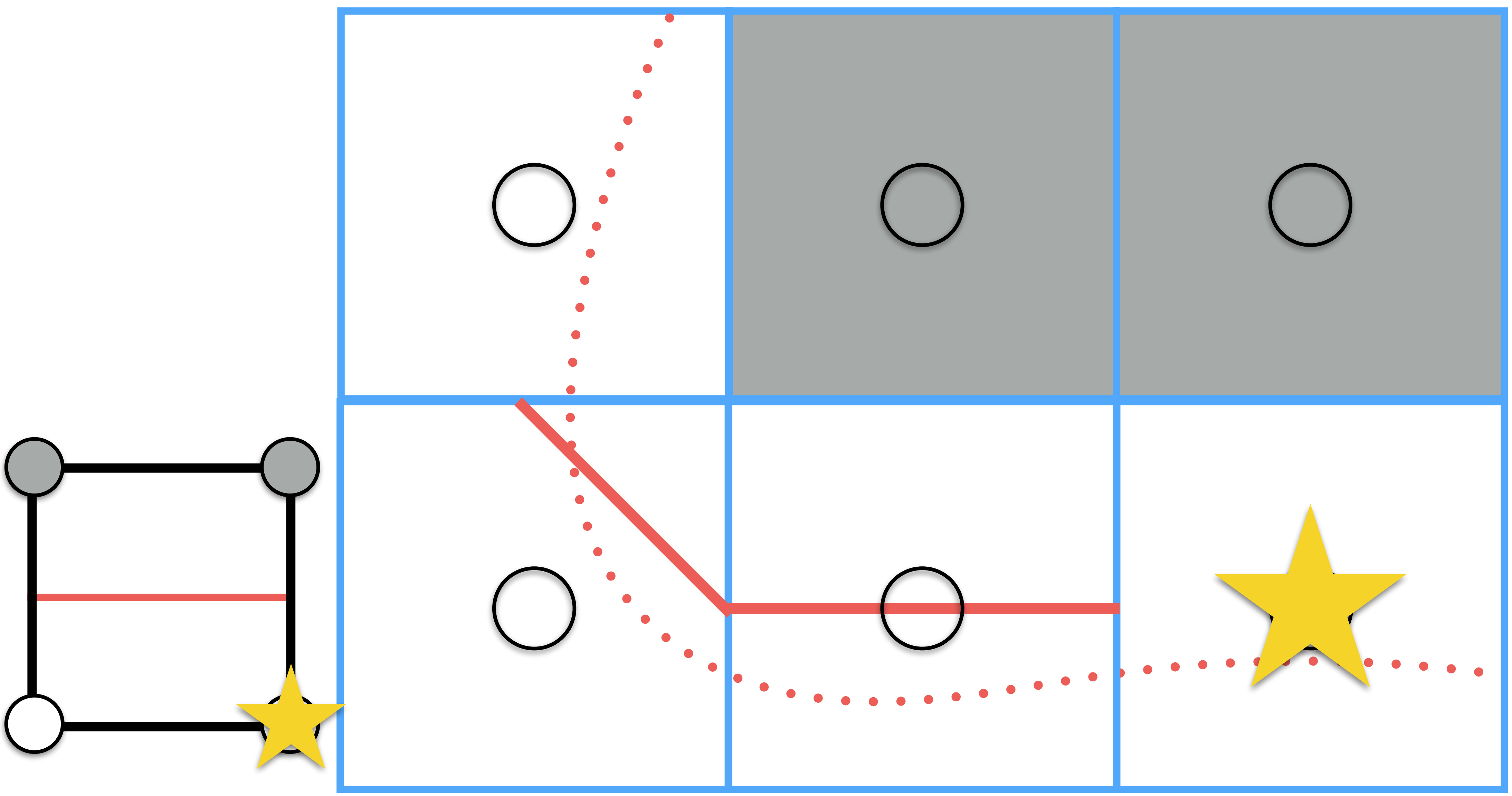
Marching Squares Example



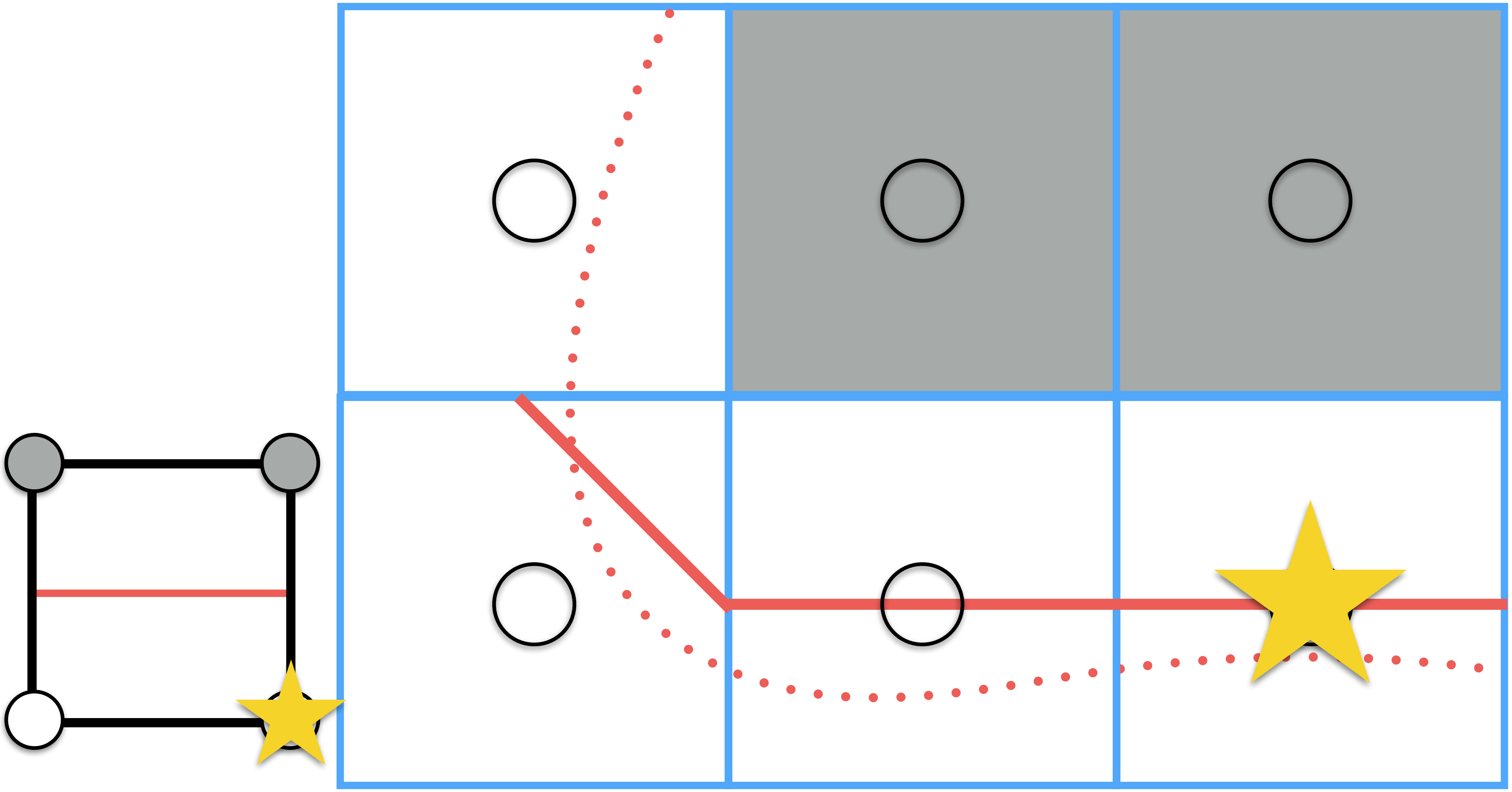
Marching Squares Example



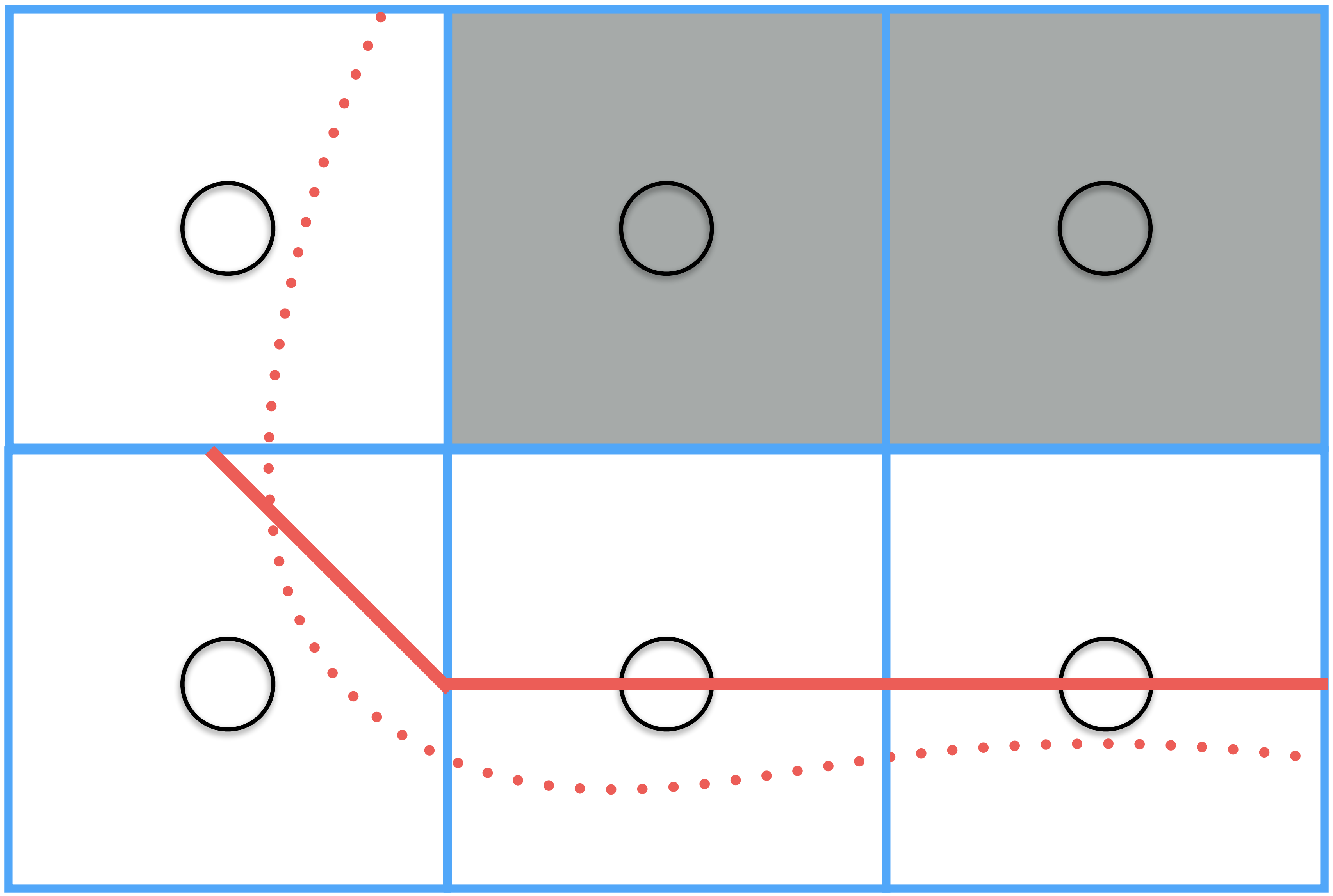
Marching Squares Example



Marching Squares Example



Marching Squares Example



Let's move into the 3D world

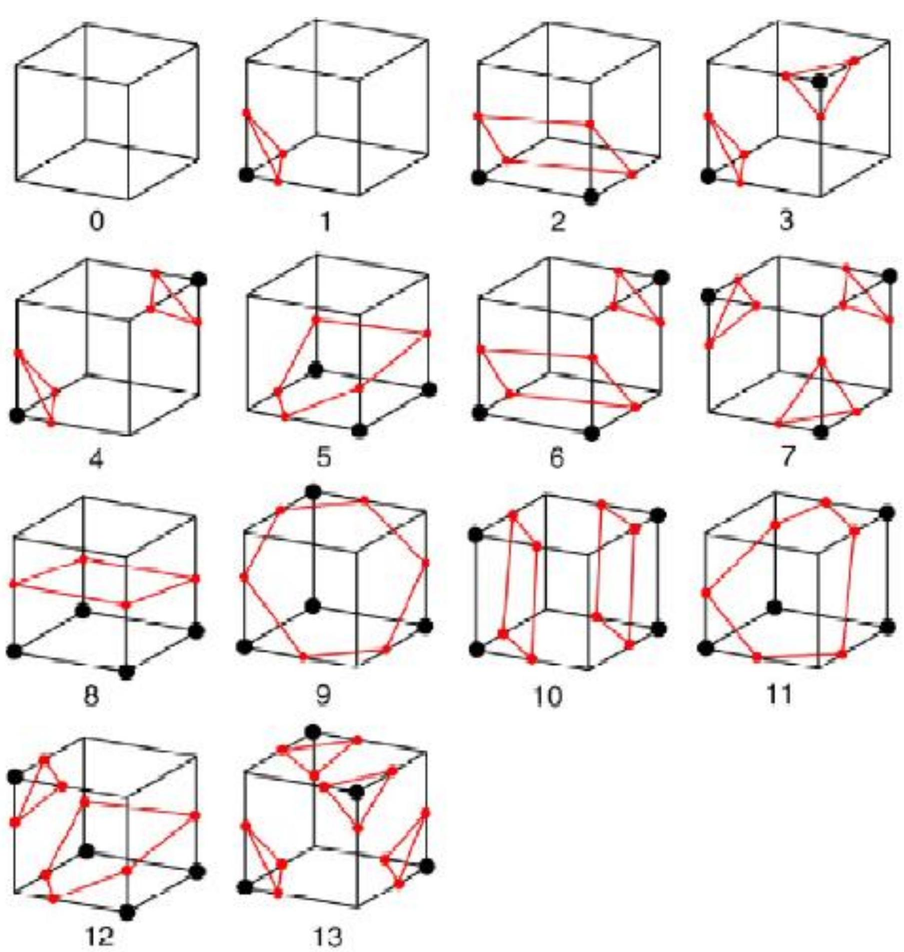
Marching Cubes

- 1st pass: as in the 2D cases, we need to mark which part of the volume is the inside (1) or the outside (0).

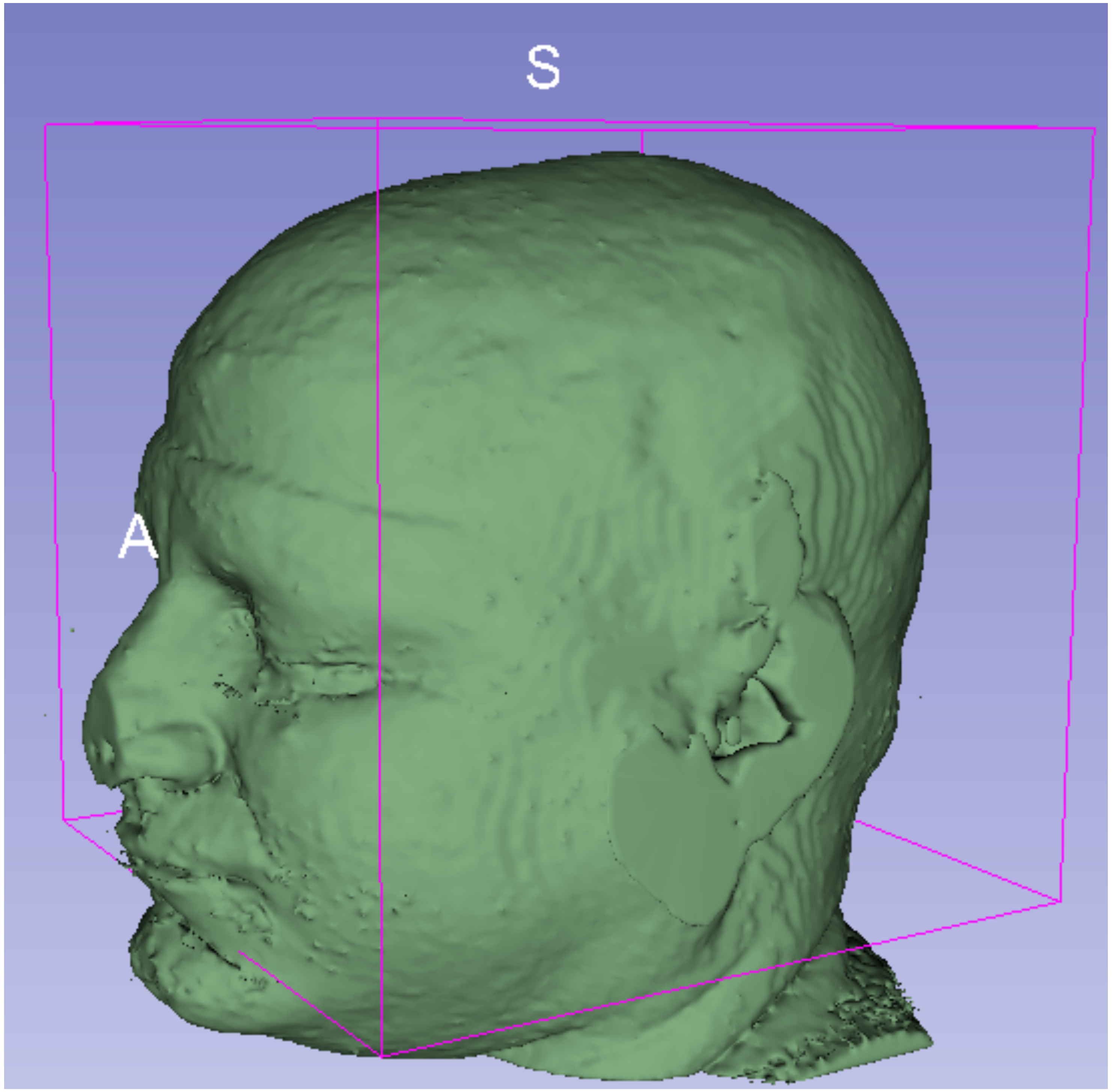
- 2nd pass: for each voxel, we need to find out the current configuration and to look up into a table to place ***triangles***!

Marching Cubes

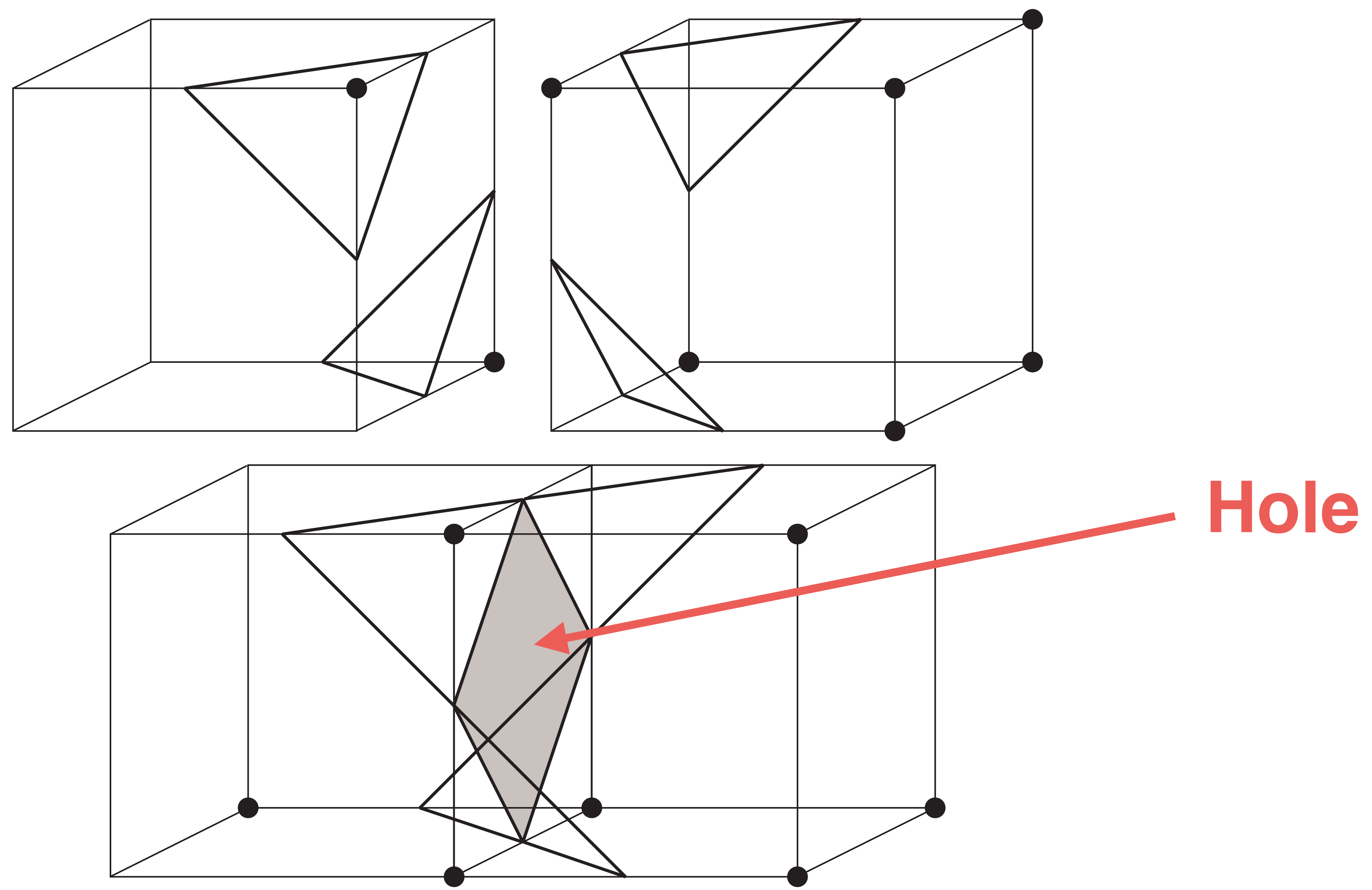
- In 3D the look up table has 256 entries (2^8).
- However, there are only 14 main cases (others are computed by reflecting and/or rotating these):



Marching Cubes



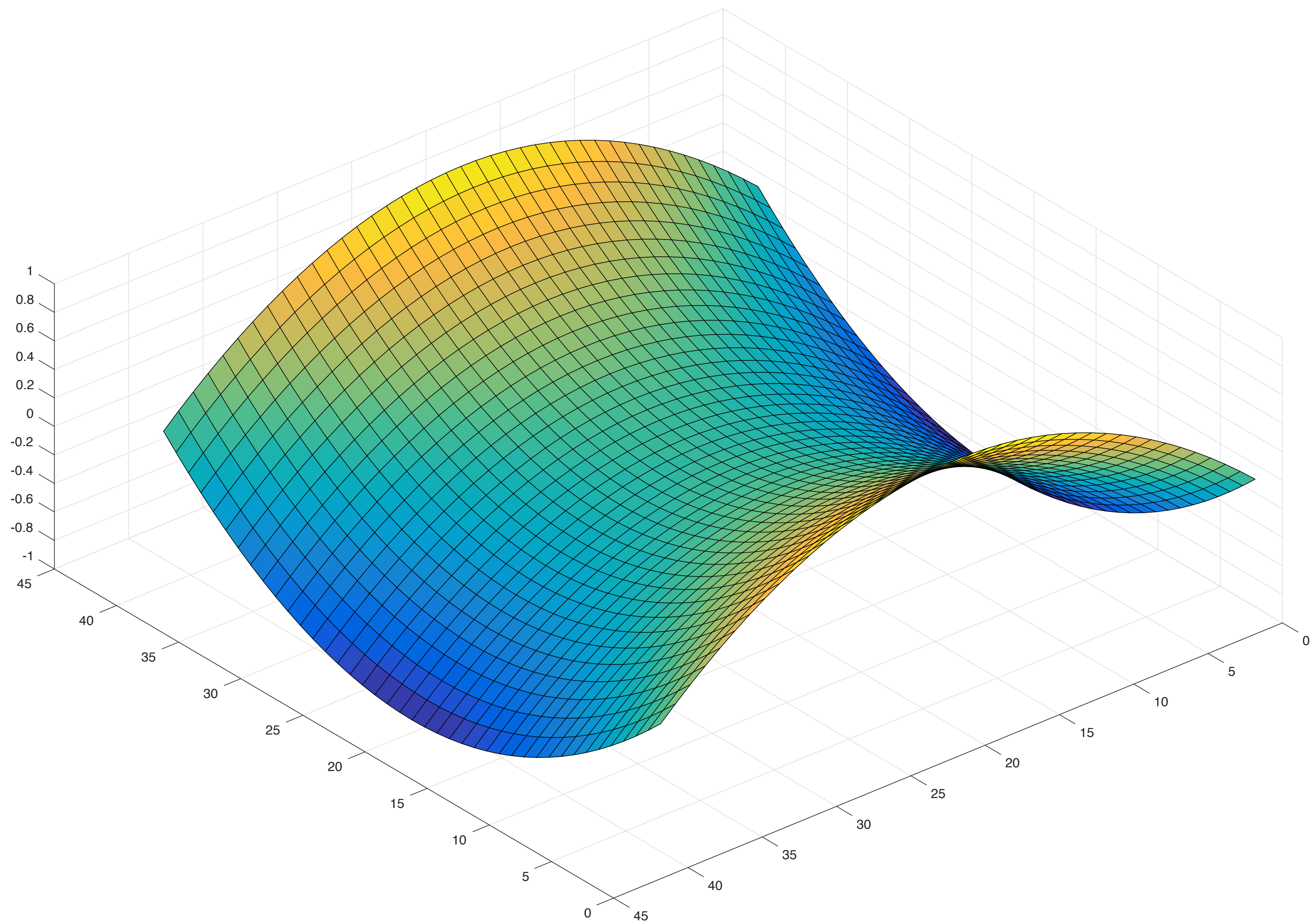
Marching Cubes: Ambiguous Cases



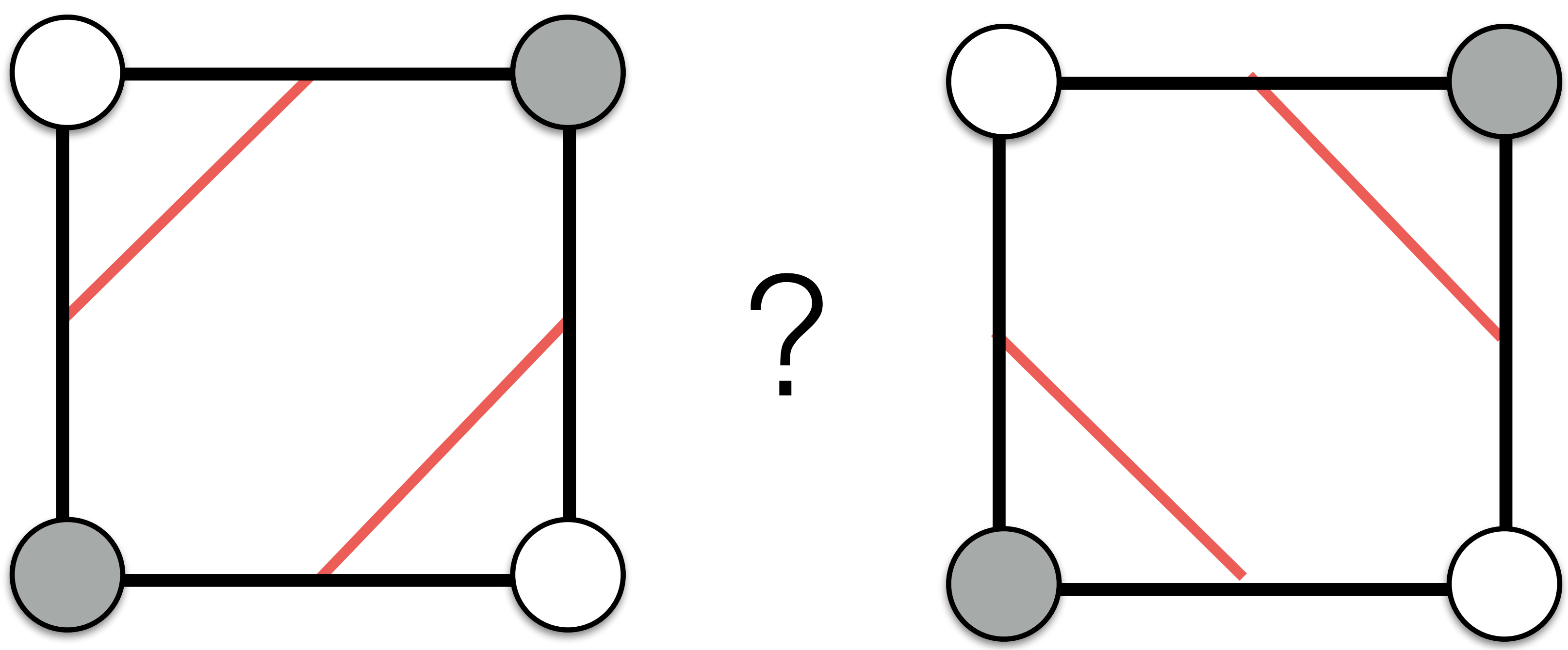
[Cignoni et al. 1999]

Marching Cubes: Ambiguous Cases

- We have ambiguous cases at saddle points.



Marching Cubes: Ambiguous Cases



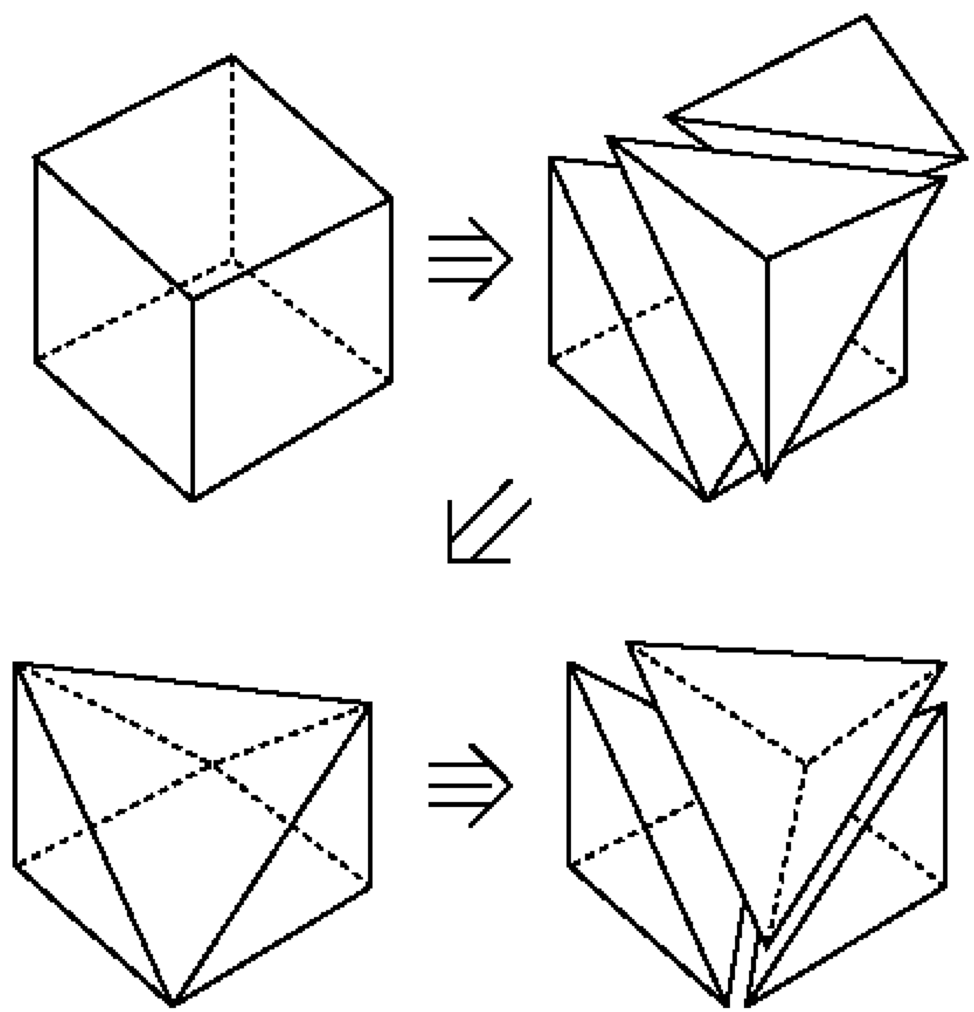
Marching Cubes: Ambiguous Cases

- A typical solution is to compute the saddle point for each face of the a current cube.

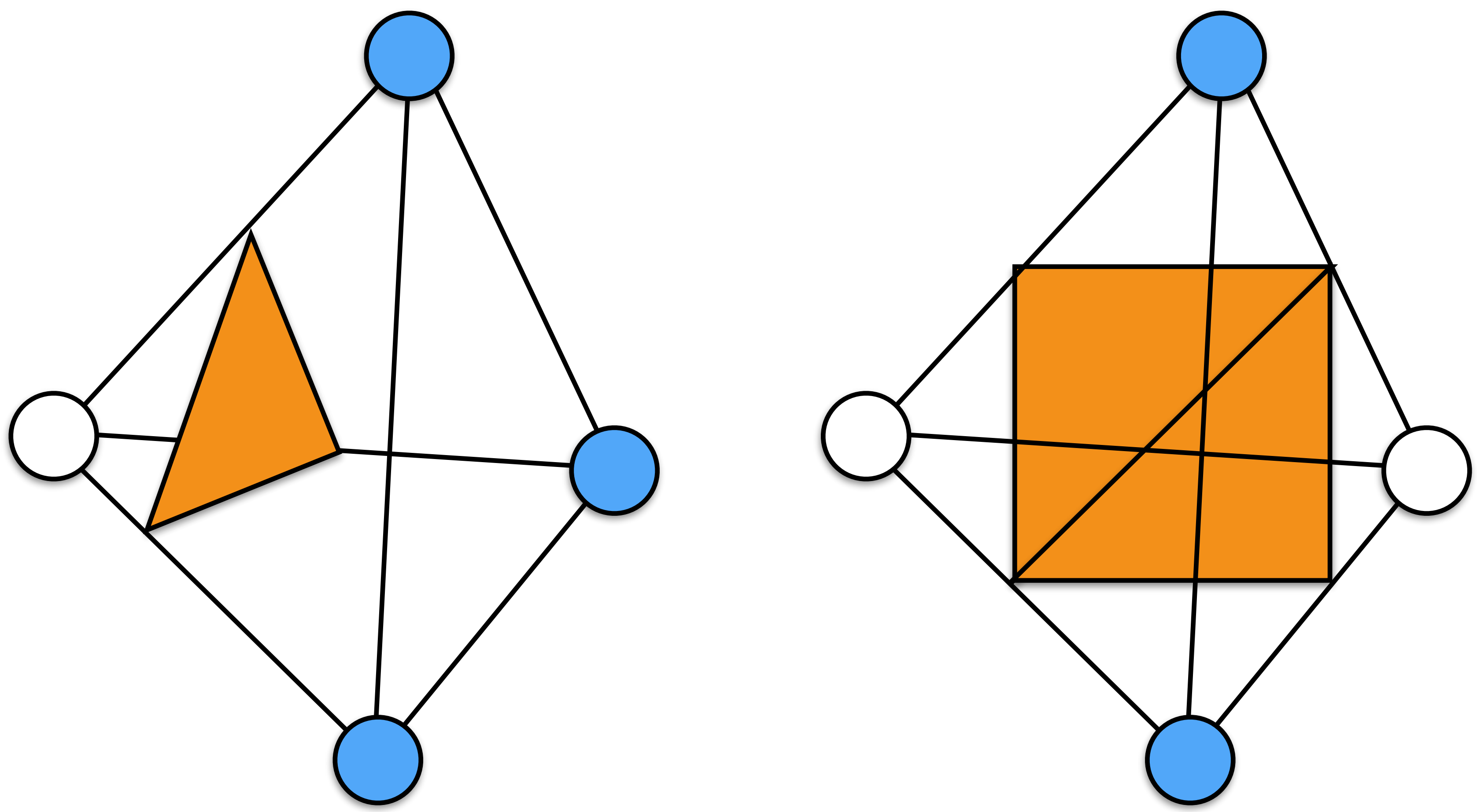
- Based on the sign of each face, we need to extend the existing cases…

Marching Cubes: Ambiguous Cases

- A solution, which avoids ambiguous cases, is to partition each voxel/cell into tetrahedra; e.g. 5 or 6 of them.



Marching Cubes: Ambiguous Cases



Marching Cubes

- Advantages:
  - Easy to understand and to implement.
  - Fast and non memory consuming.
  - Very robust.

Marching Cubes

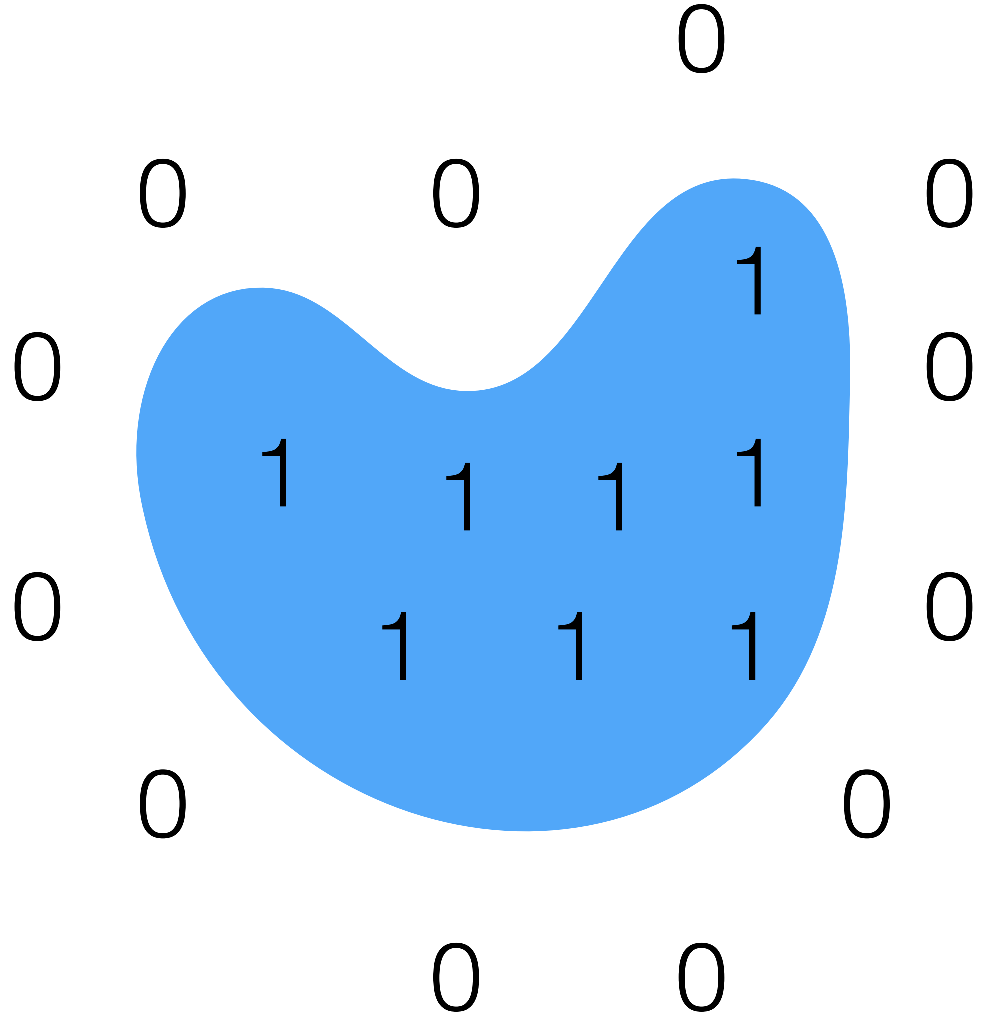
- Disadvantages:
  - Consistency: Guarantee a C0 and manifold result: ambiguous cases.
  - Correctness: return a good approximation of the real surface
  - Mesh complexity: the number of triangles does not depend on the shape of the isosurface (but on the discretization, i.e., number of voxels).
  - Mesh quality: arbitrarily ugly triangles.

Poisson Reconstruction

Poisson Reconstruction

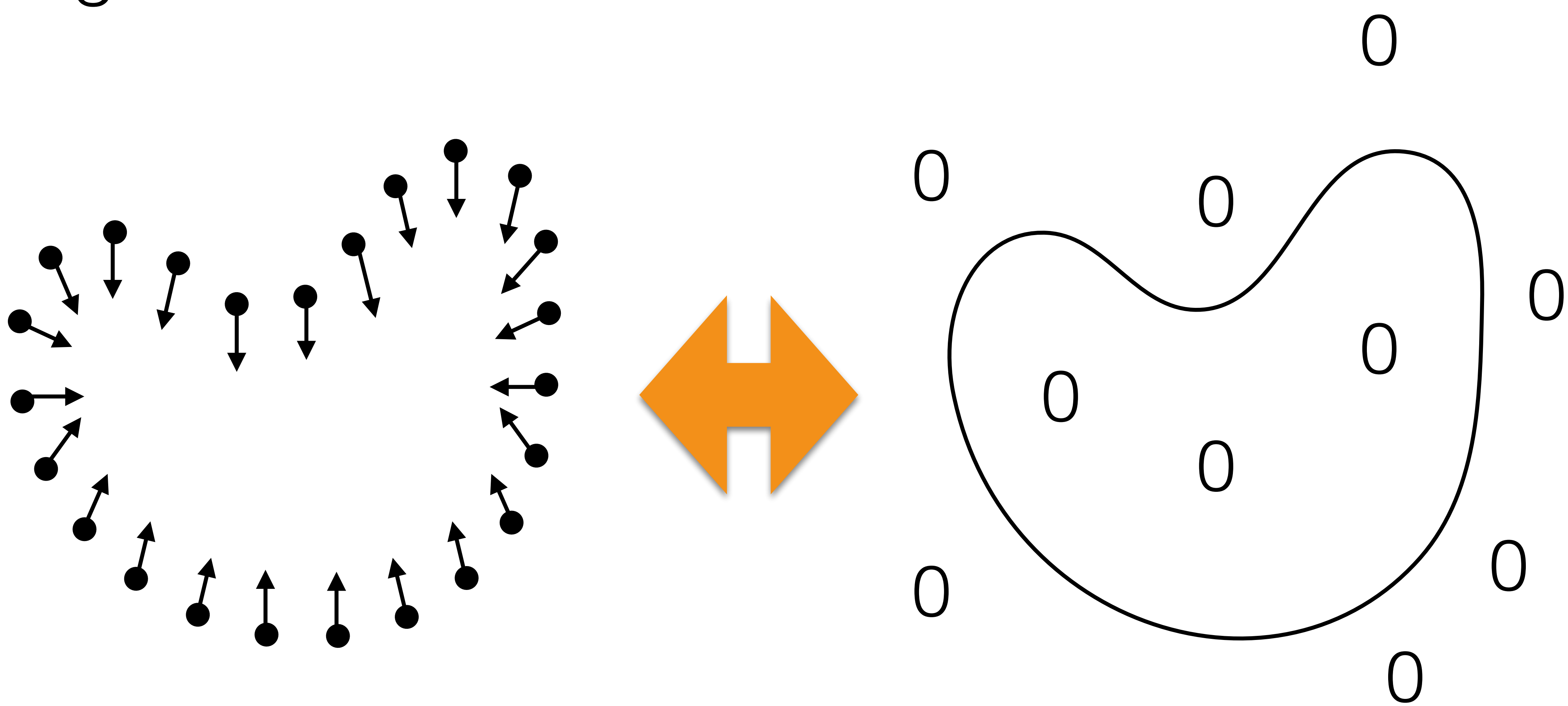
- The idea of this method is to reconstruct the surface of a 3D model by solving for the indicator function of the shape:

$$\chi(\mathbf{p}) = \begin{cases} 1 & \text{if } \mathbf{p} \in M, \\ 0 & \text{otherwise.} \end{cases}$$



Poisson Reconstruction: Gradient Relationship

- There is a relationship between the normal field and gradient of indicator function:



Oriented Points

Indicator function gradient

Poisson Reconstruction: Integration as a Poisson Problem

- Let's represent the points with a normal by a vector field $\vec{V}$.

- We need to find a function χ whose gradients best approximates $\vec{V}$:

$$\min_{\chi} \|\nabla\chi - \vec{V}\|$$

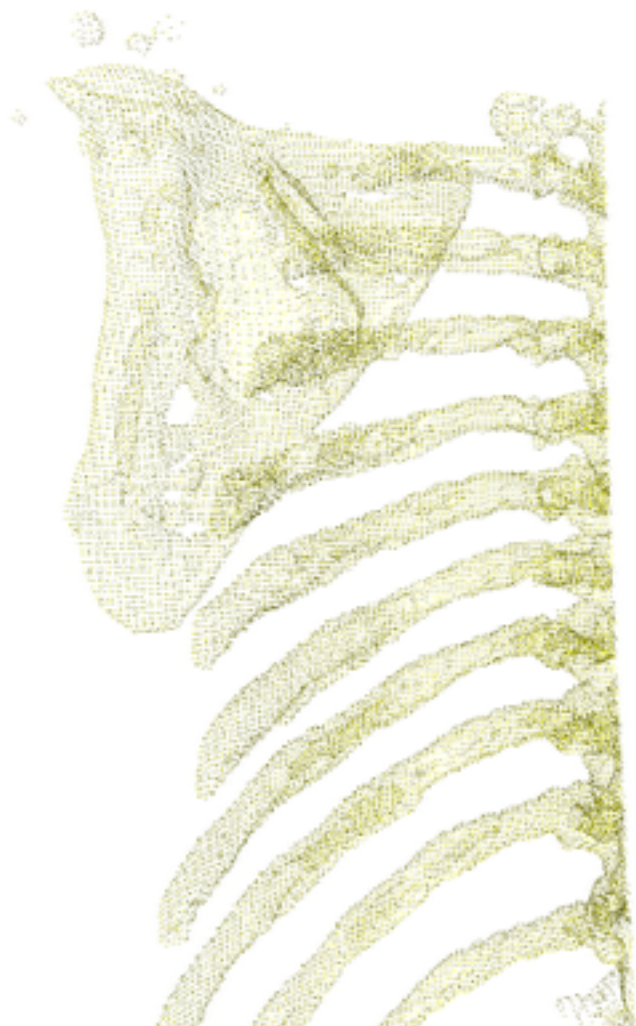
Poisson Reconstruction: Integration as a Poisson Problem

- If we apply the divergence operator, this becomes a Poisson problem:

$$\min_{\chi} \|\nabla \chi - \vec{V}\|$$

$$\nabla \cdot (\nabla \chi) = \nabla \cdot \vec{V} \leftrightarrow \Delta \chi = \nabla \cdot \vec{V}$$

Poisson Reconstruction Example



Poisson Reconstruction Example

Poisson Reconstruction

- Precise and robust.
- Computationally slow, it depends on the resolution.
- The Poisson solution needs to close stuff so if there are no enough points in an area weird things will happen.

that’s all folks!

Acknowledgements

- Some images on work by:
  - Dr. Fabio Ganovelli:
    - http://vcg.isti.cnr.it/~ganovell/
  - Dr. Paolo Cignoni:
    - http://vcg.isti.cnr.it/~cignoni/