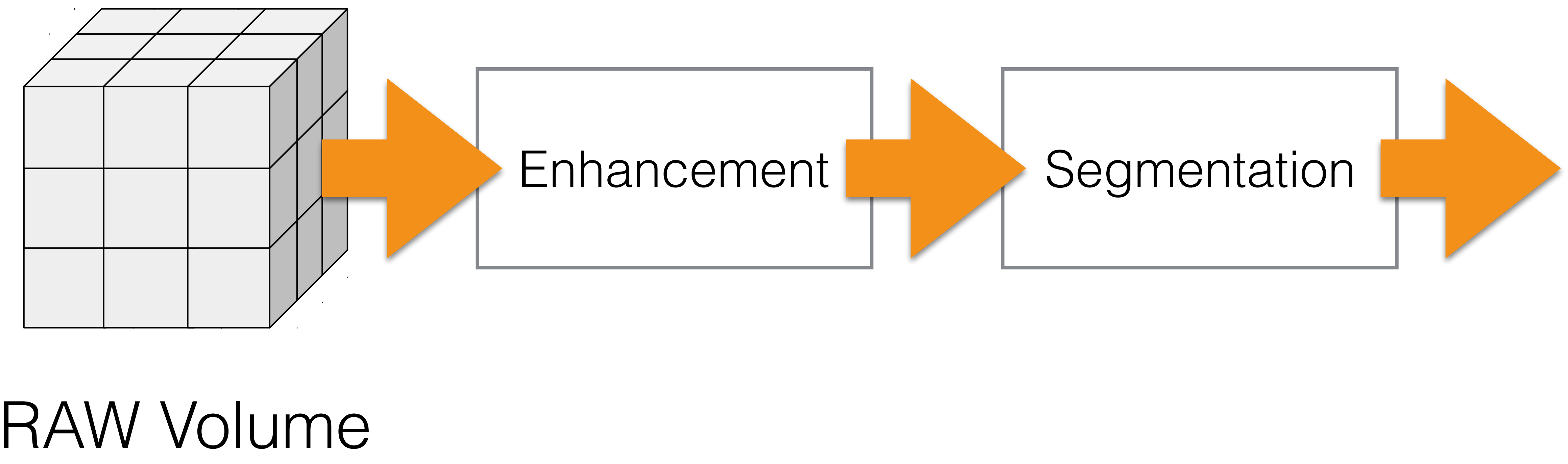
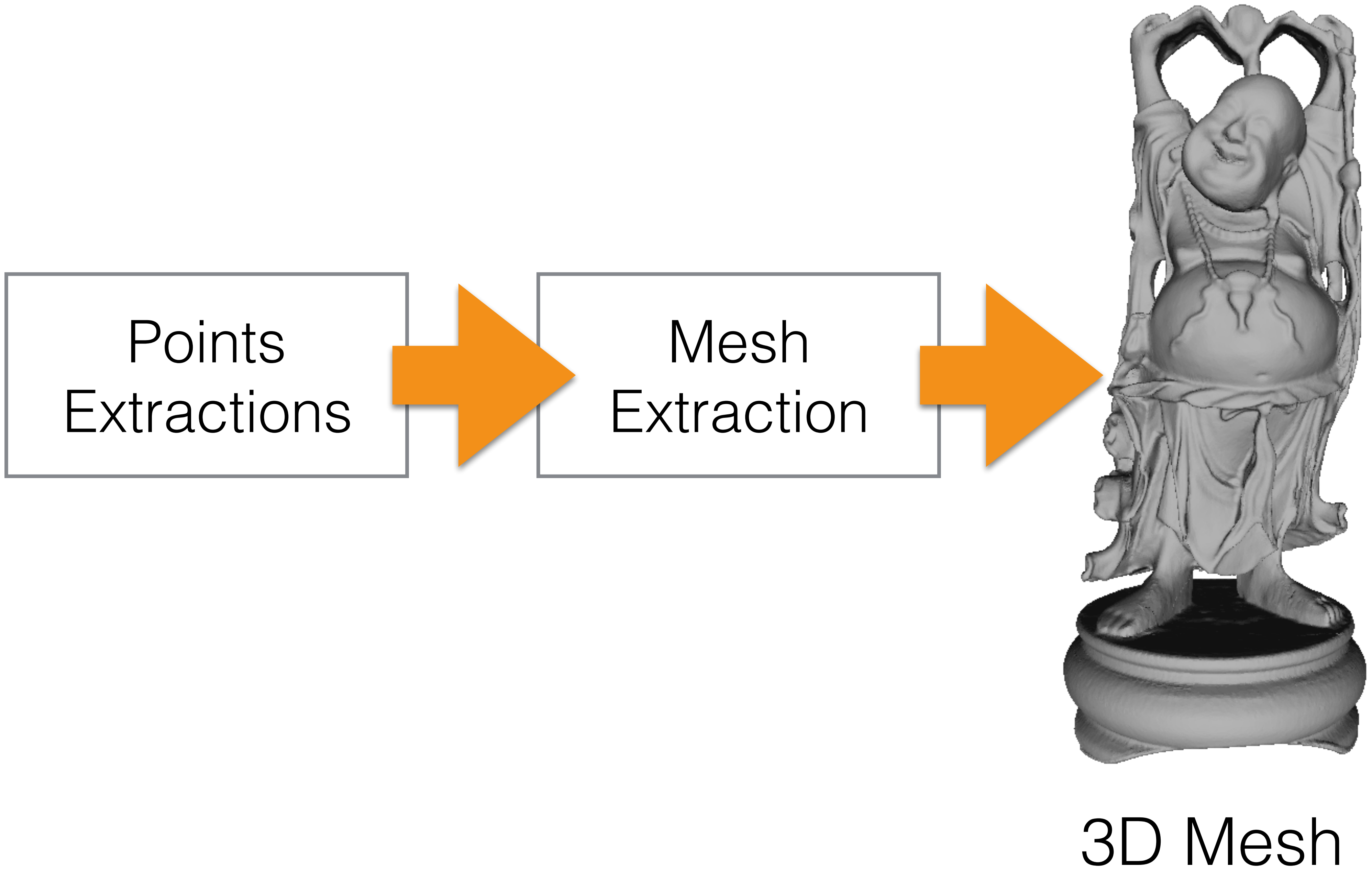
3D from Volume: Part III

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

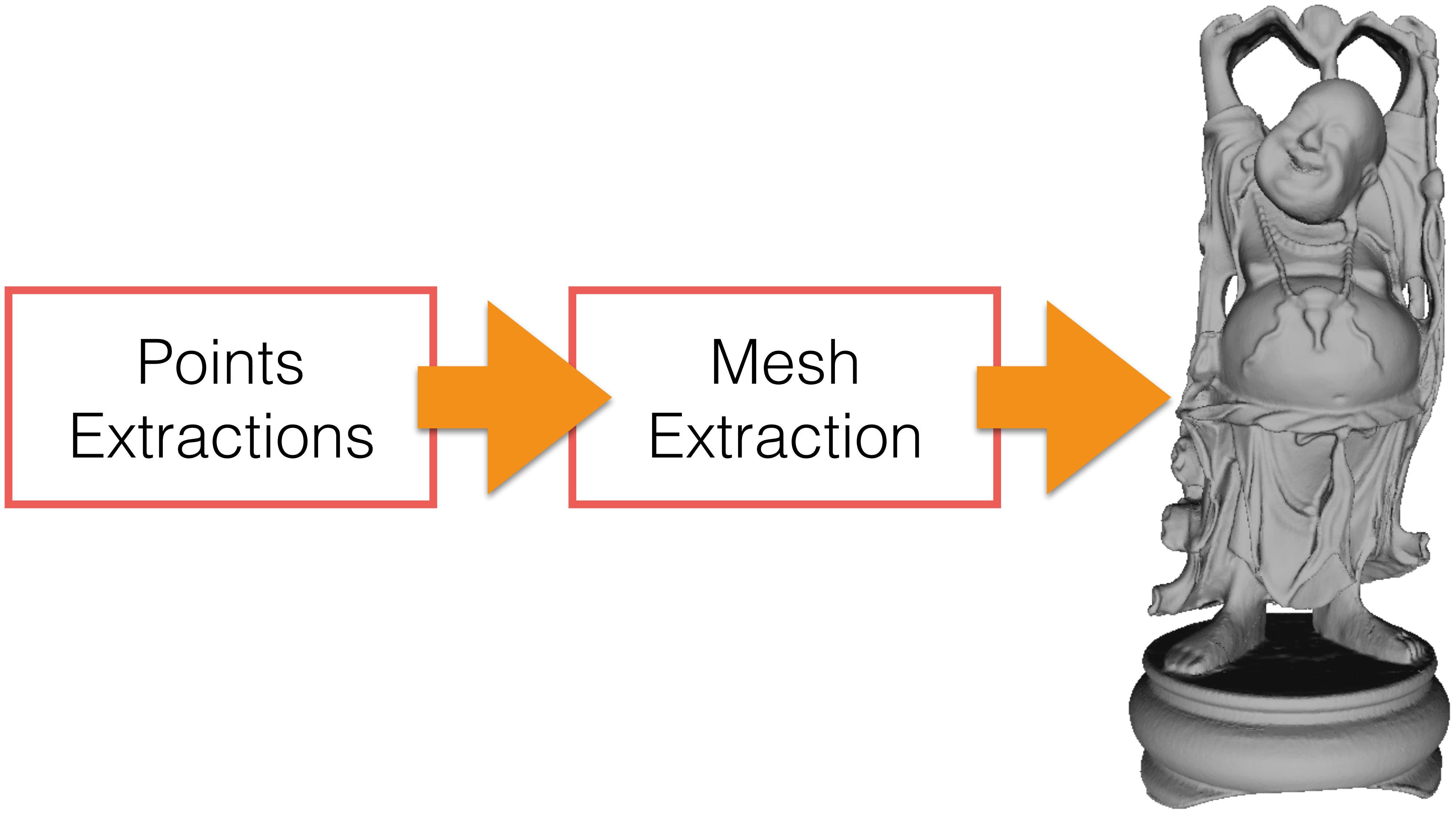
The Processing Pipeline



The Processing Pipeline



The Processing Pipeline

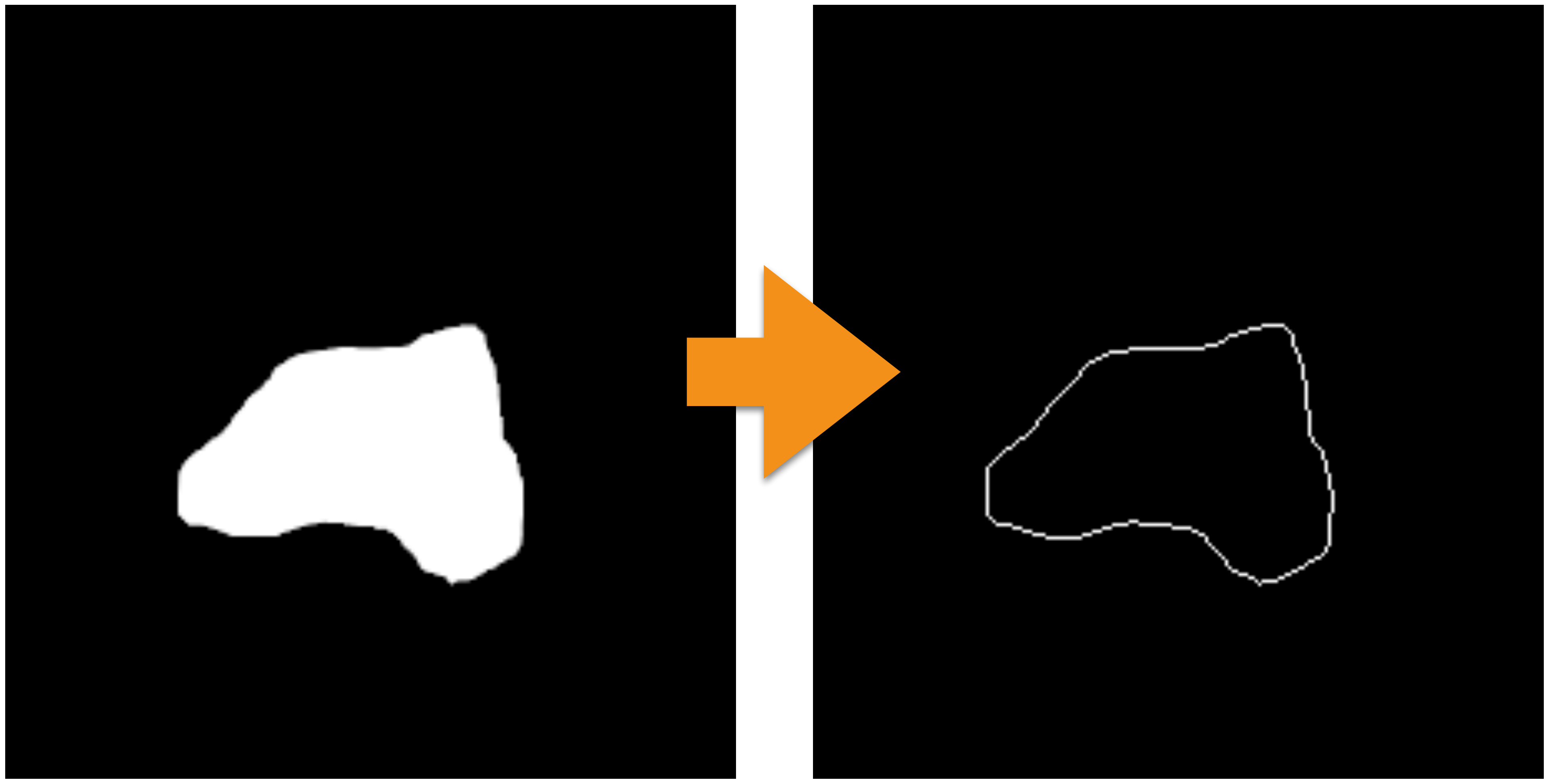


3D Mesh

3D Points Extraction

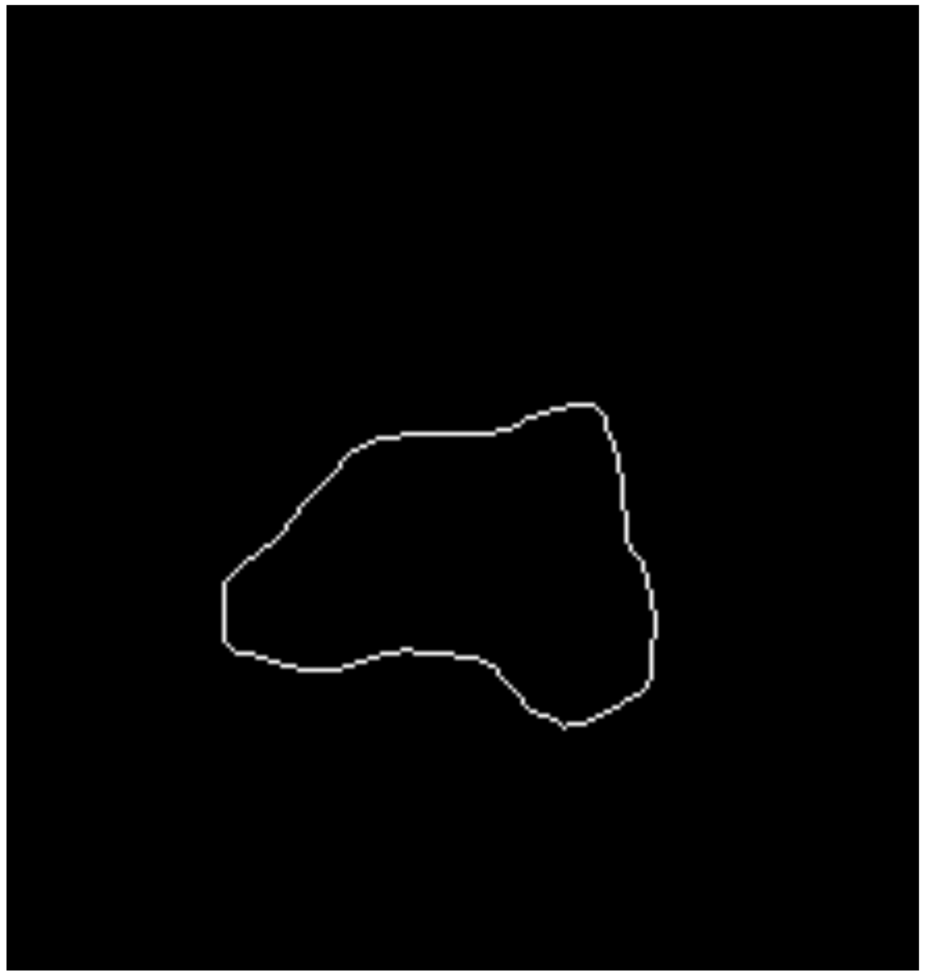
3D Points Extraction

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



3D Points Extraction

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



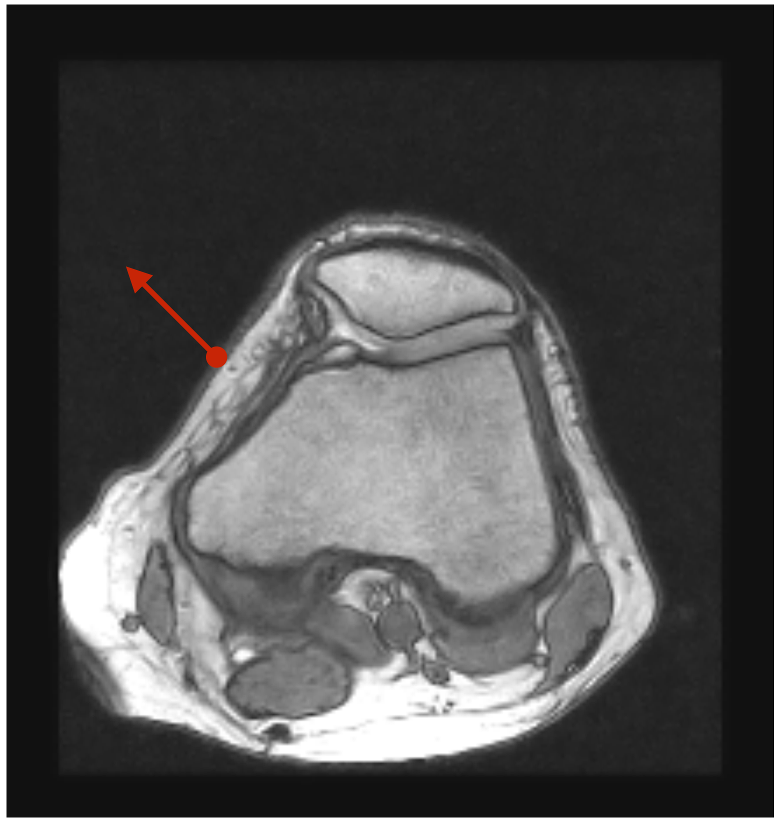
$$m = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} u \cdot k_u \\ v \cdot k_v \\ i \cdot k_w \end{bmatrix}$$

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

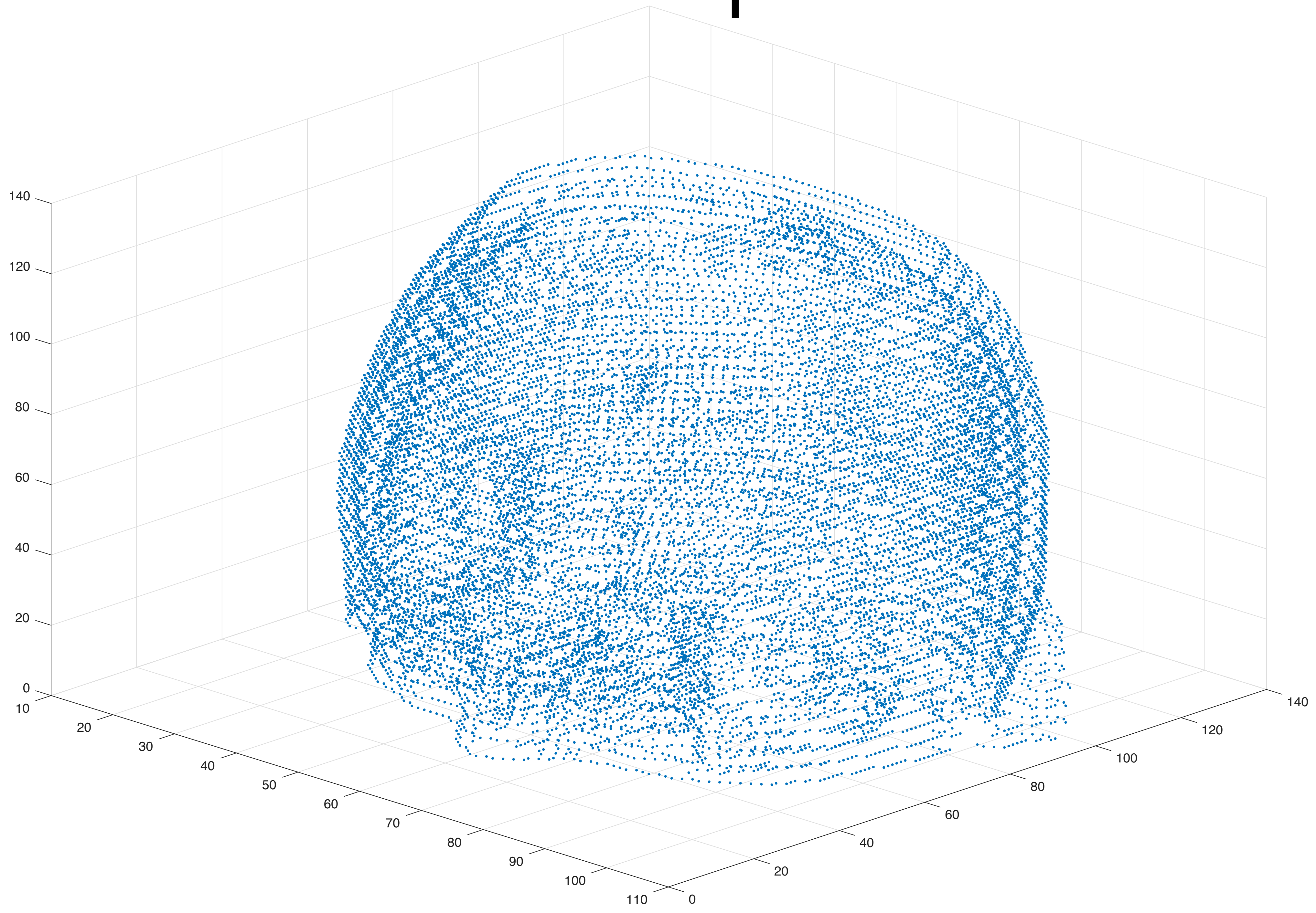
3D Points Extraction

- How do we compute the normal at the point?
- A normal is simply the normalized (i.e., norm 1.0) negative value of the gradient of the volume (not of the mask!) at that point:

$$\vec{n} = -\frac{\vec{\nabla} V}{\|\vec{\nabla} V\|}$$



3D Points Extraction Example

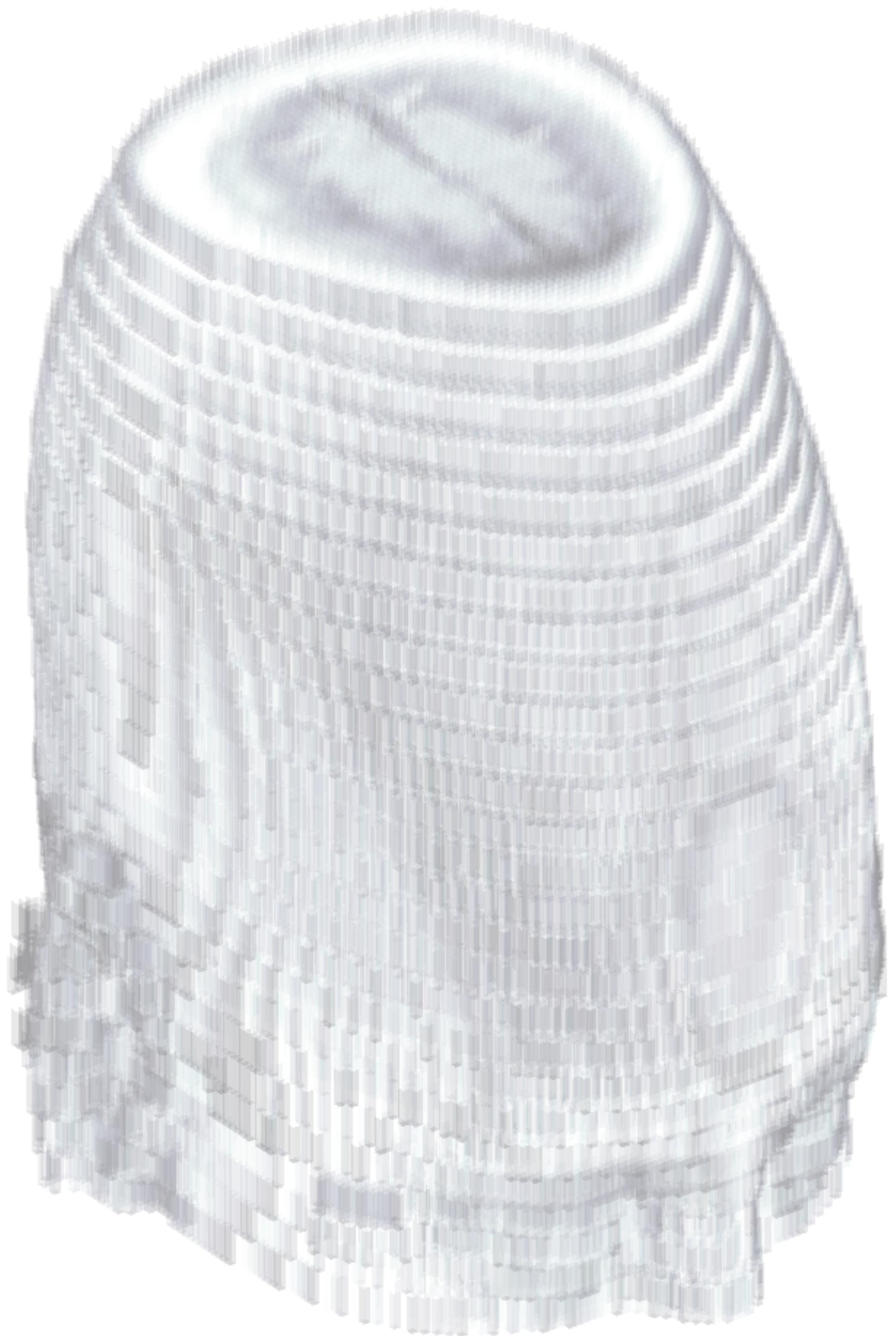


3D Mesh Extraction

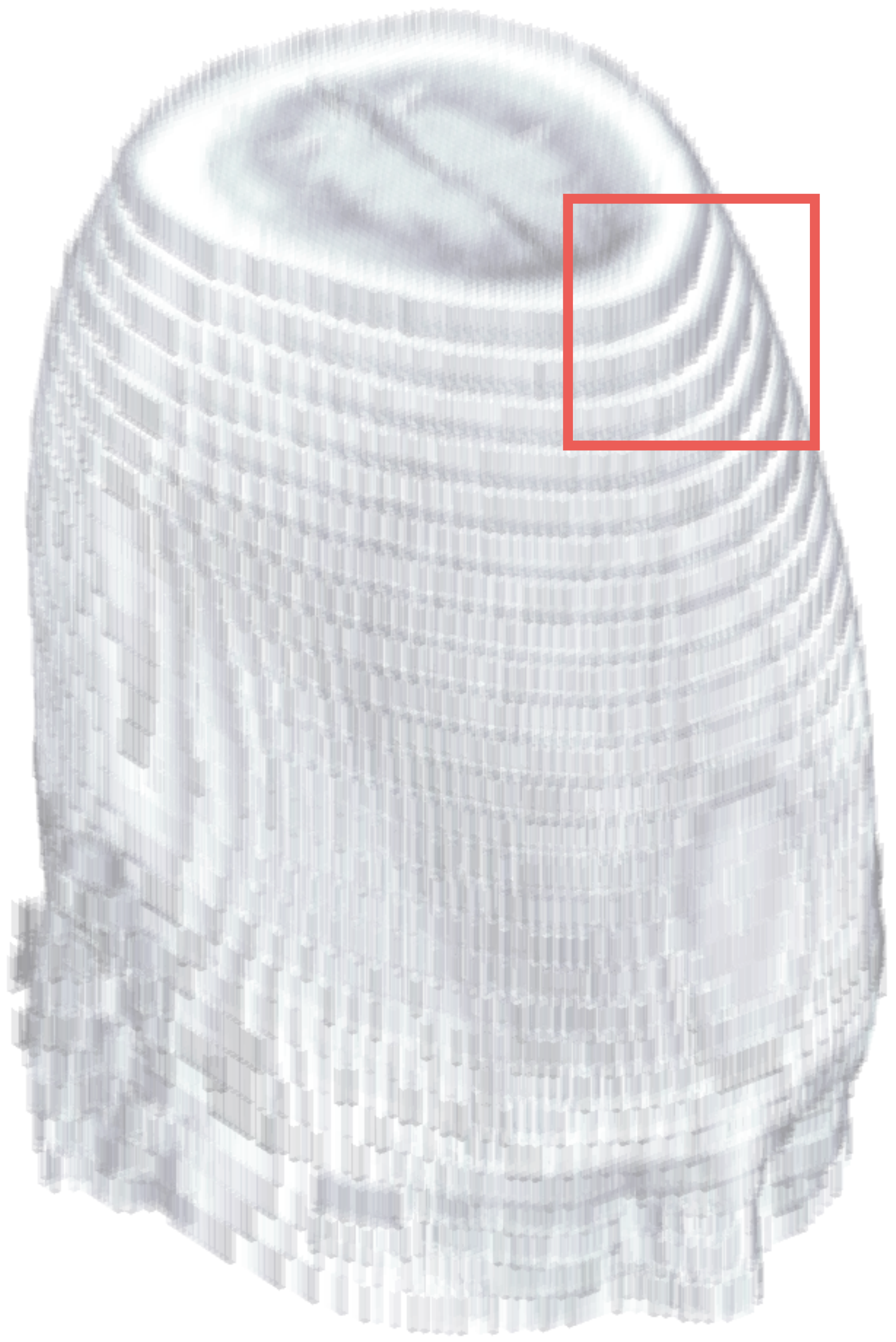
A Very Stupid Algorithm:

For each extracted point, we create a cube…

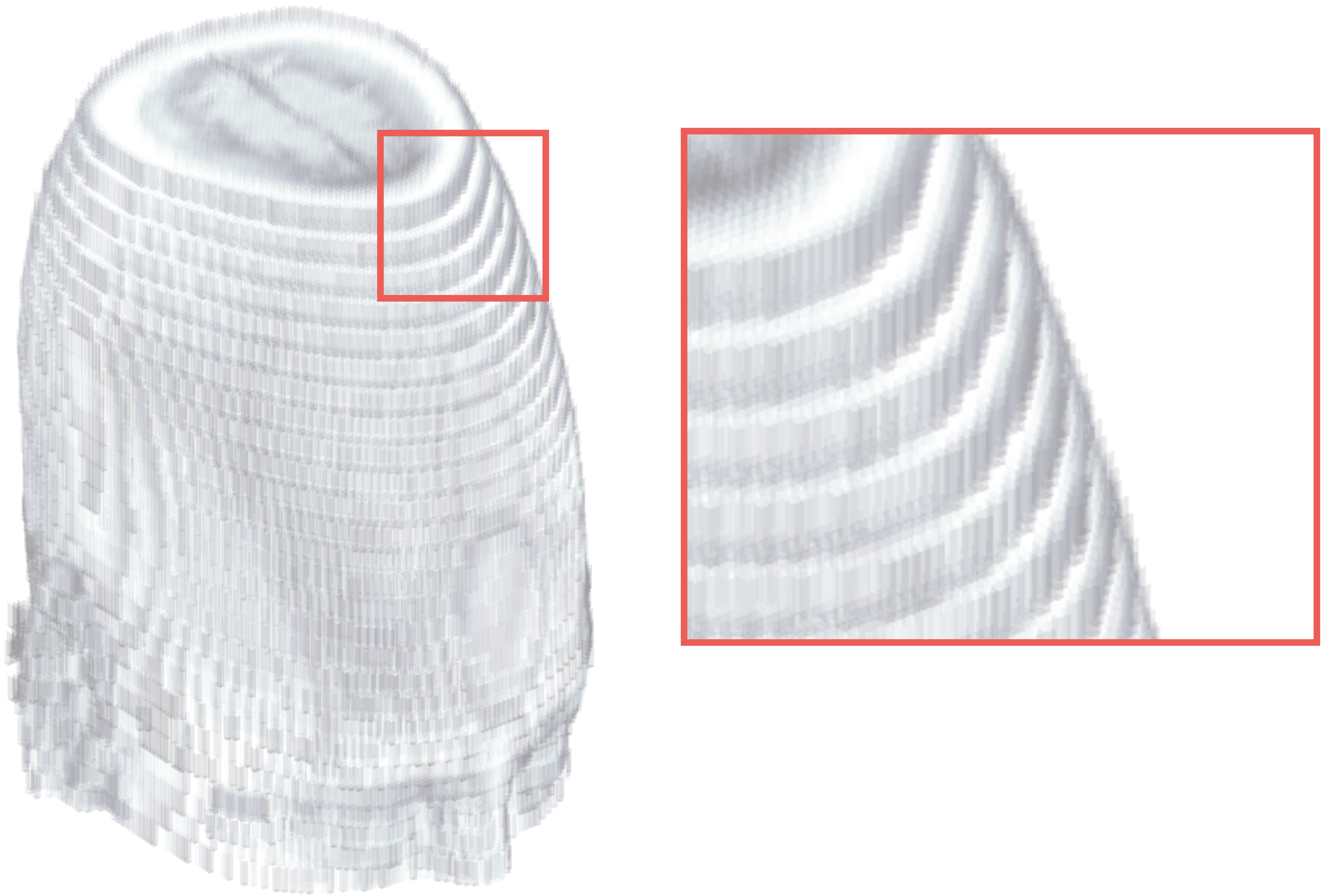
A Very Stupid Algorithm Example



A Very Stupid Algorithm Example



A Very Stupid Algorithm Example



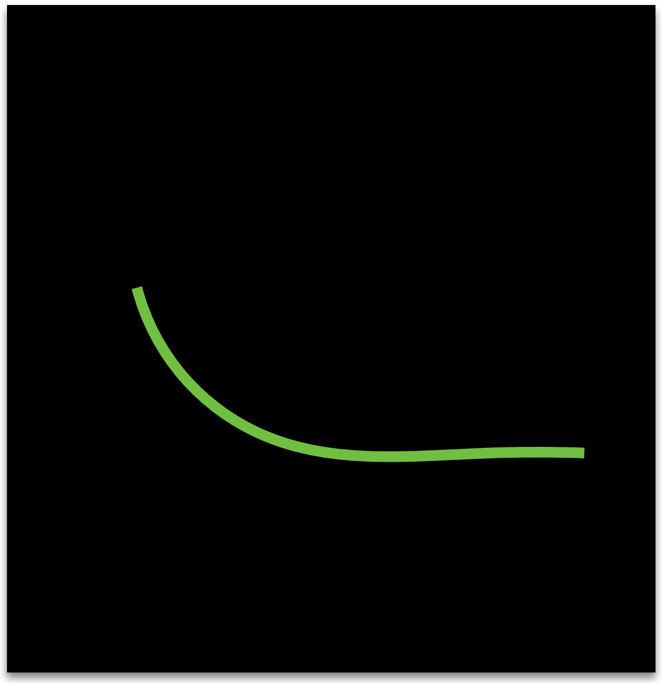
I guess, we can do better than this!

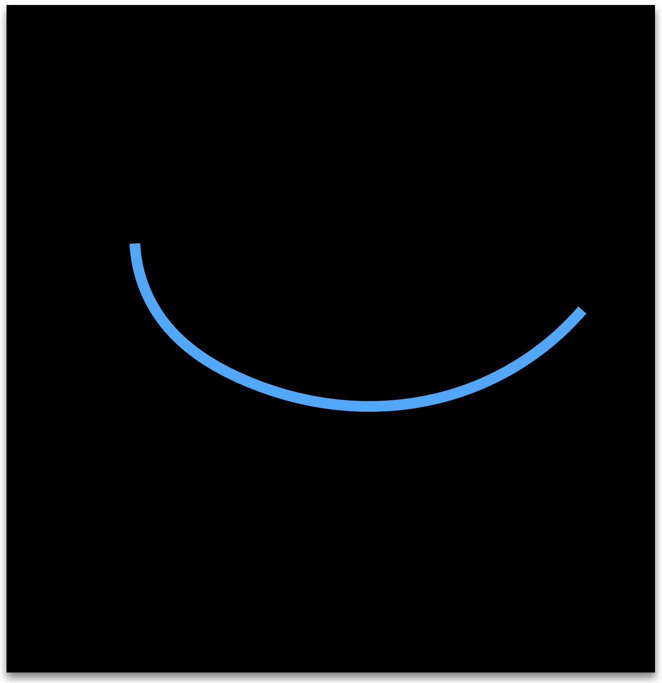
Connecting the dots…

Edges Triangulation

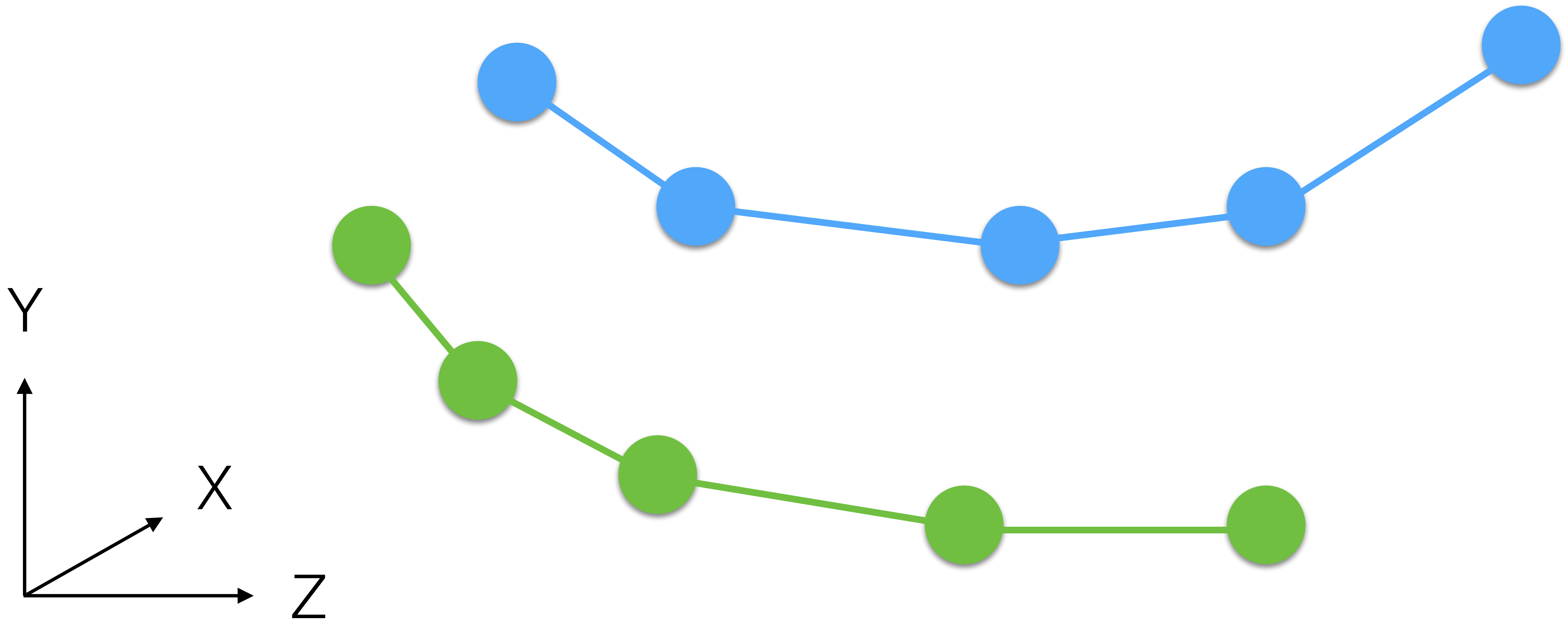
- As the first step, we extract the edges from each slice in the volume.
- We save the connectivity of points belonging to the same edge —> “parametric curve”.

Edges Triangulation: Working Example


Slice 1

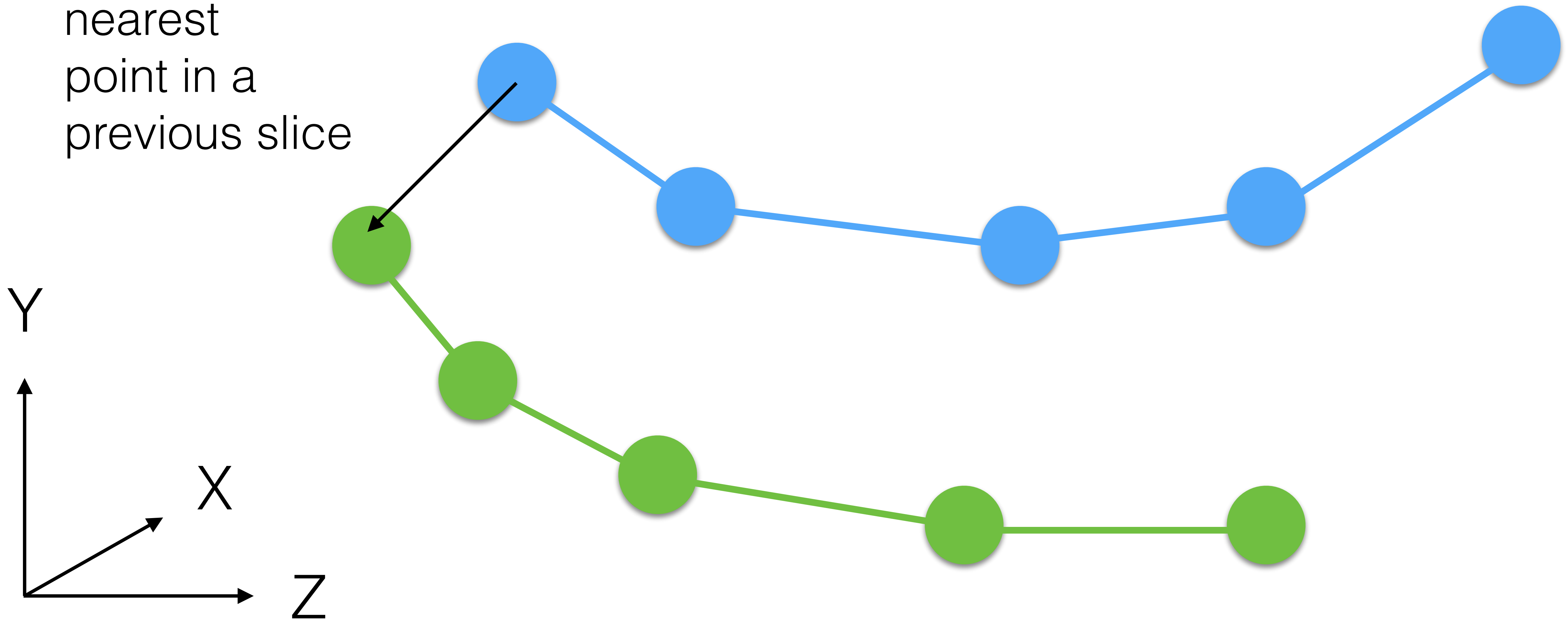

Slice 2

Edges Triangulation: Working Example

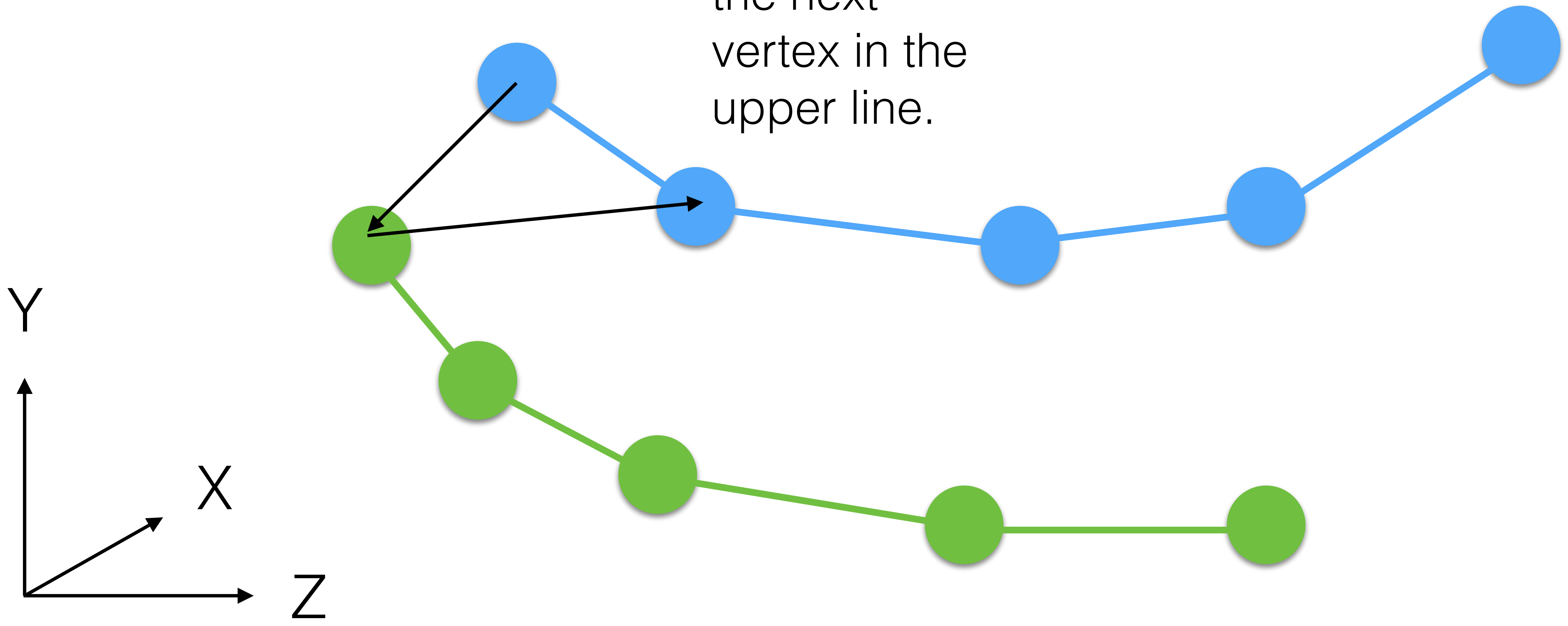


Edges Triangulation: Working Example

Find the nearest point in a previous slice

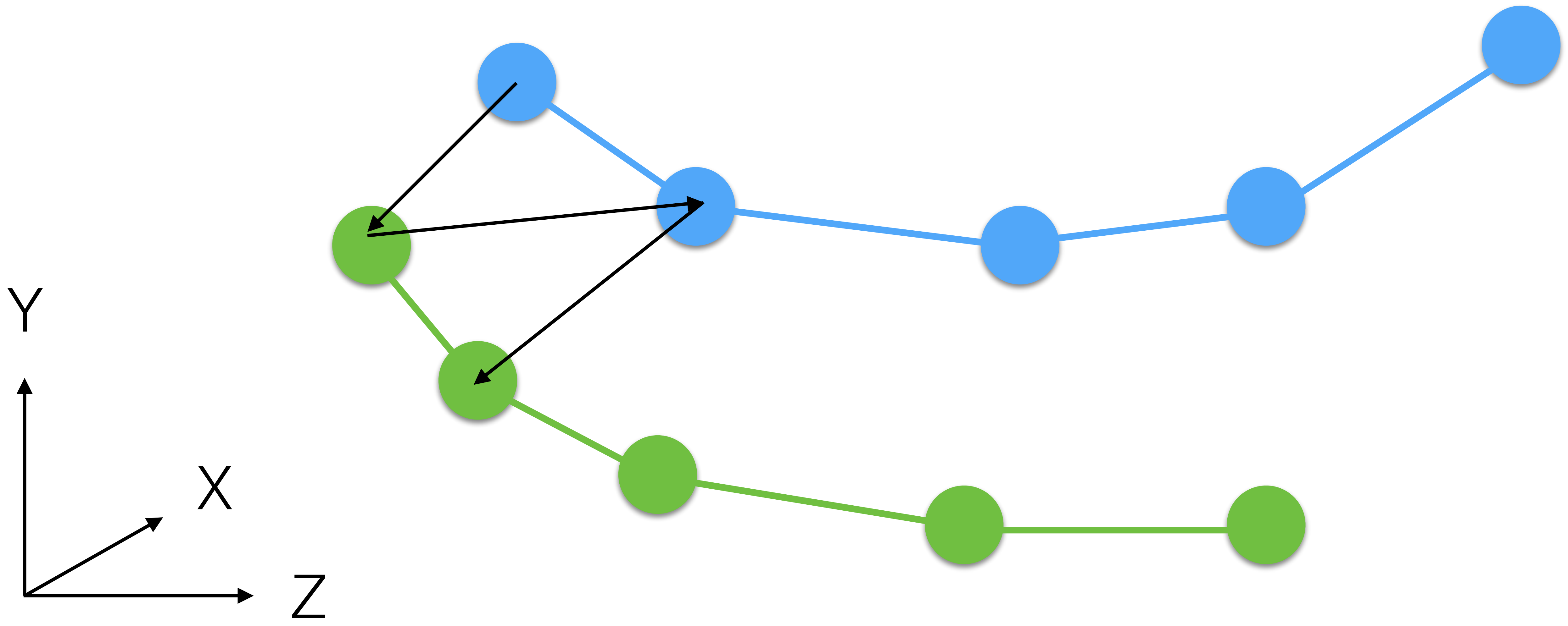


Edges Triangulation: Working Example

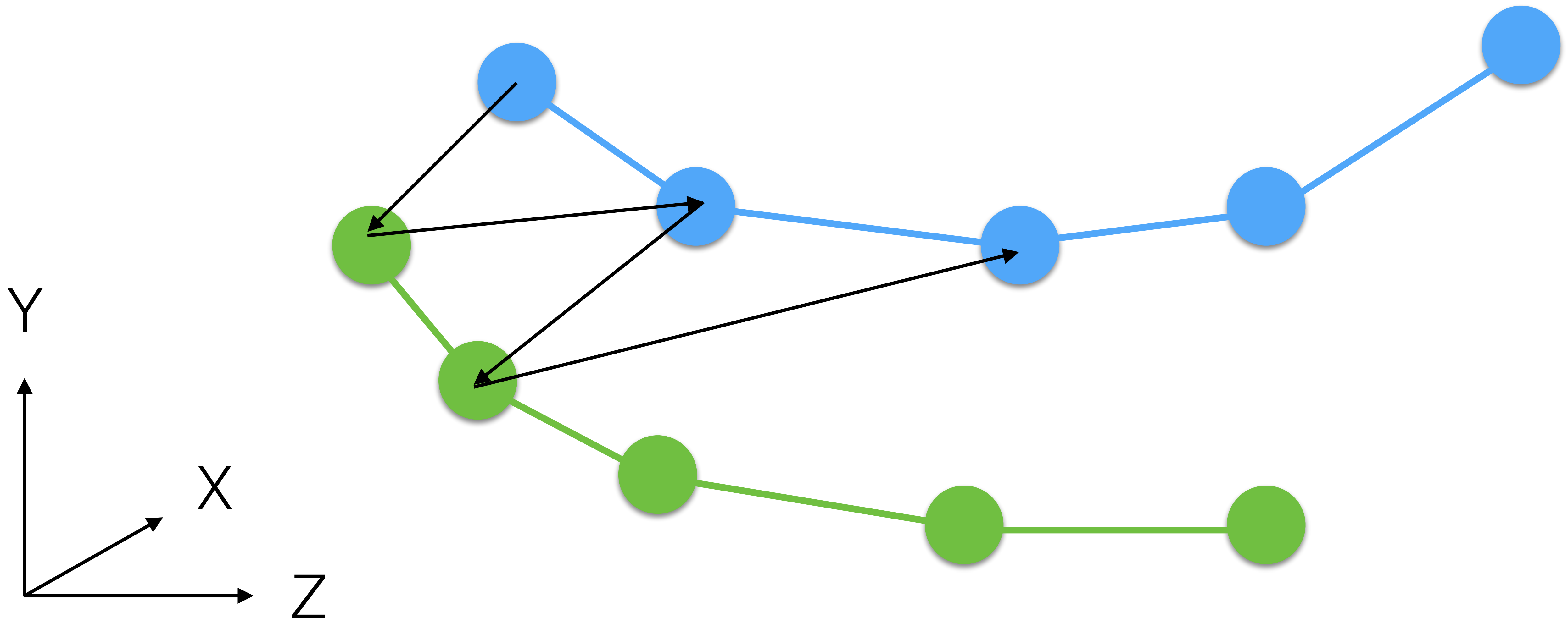


Connect with the next vertex in the upper line.

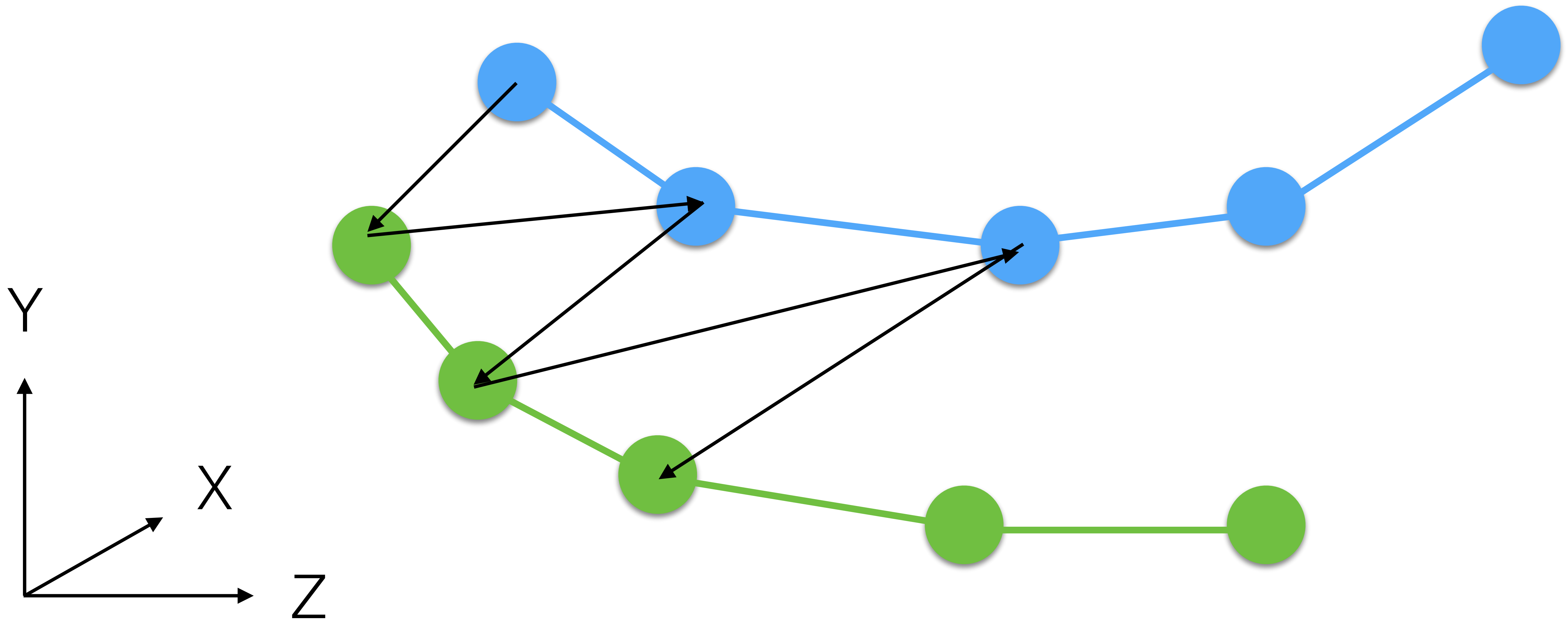
Edges Triangulation: Working Example



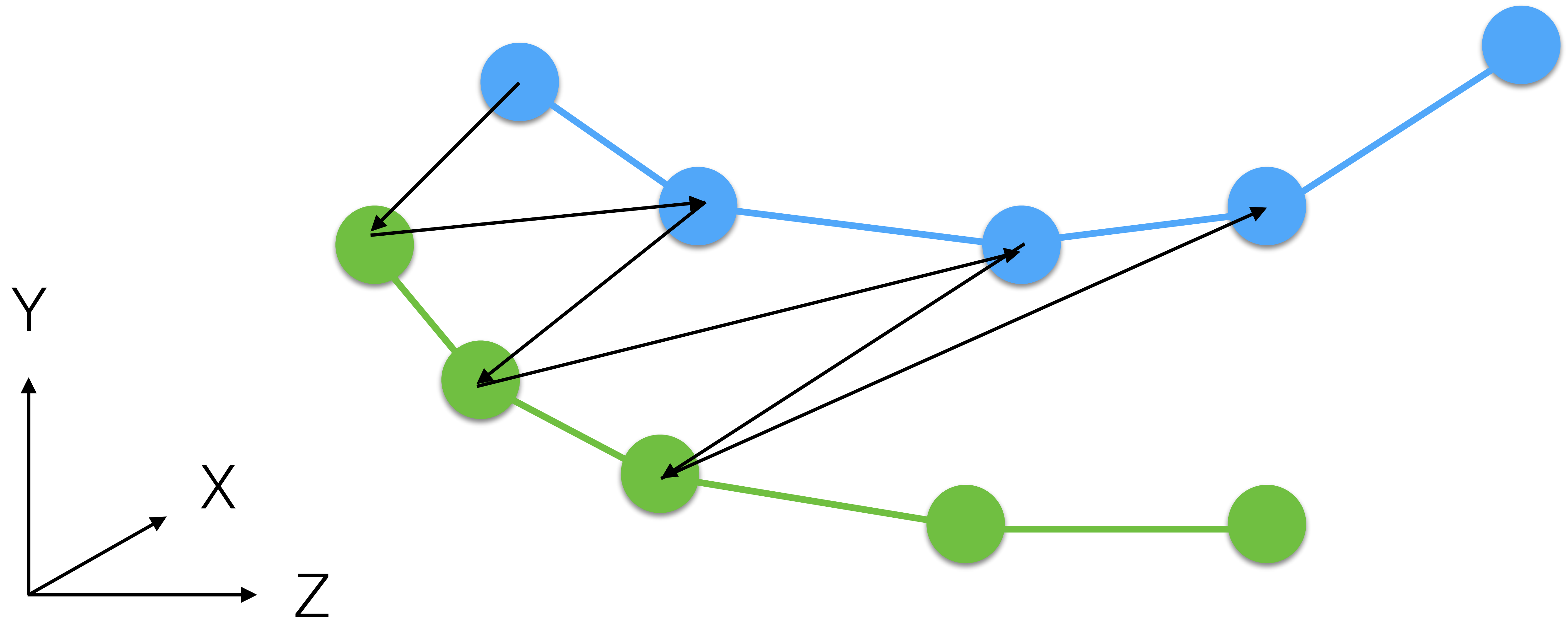
Edges Triangulation: Working Example



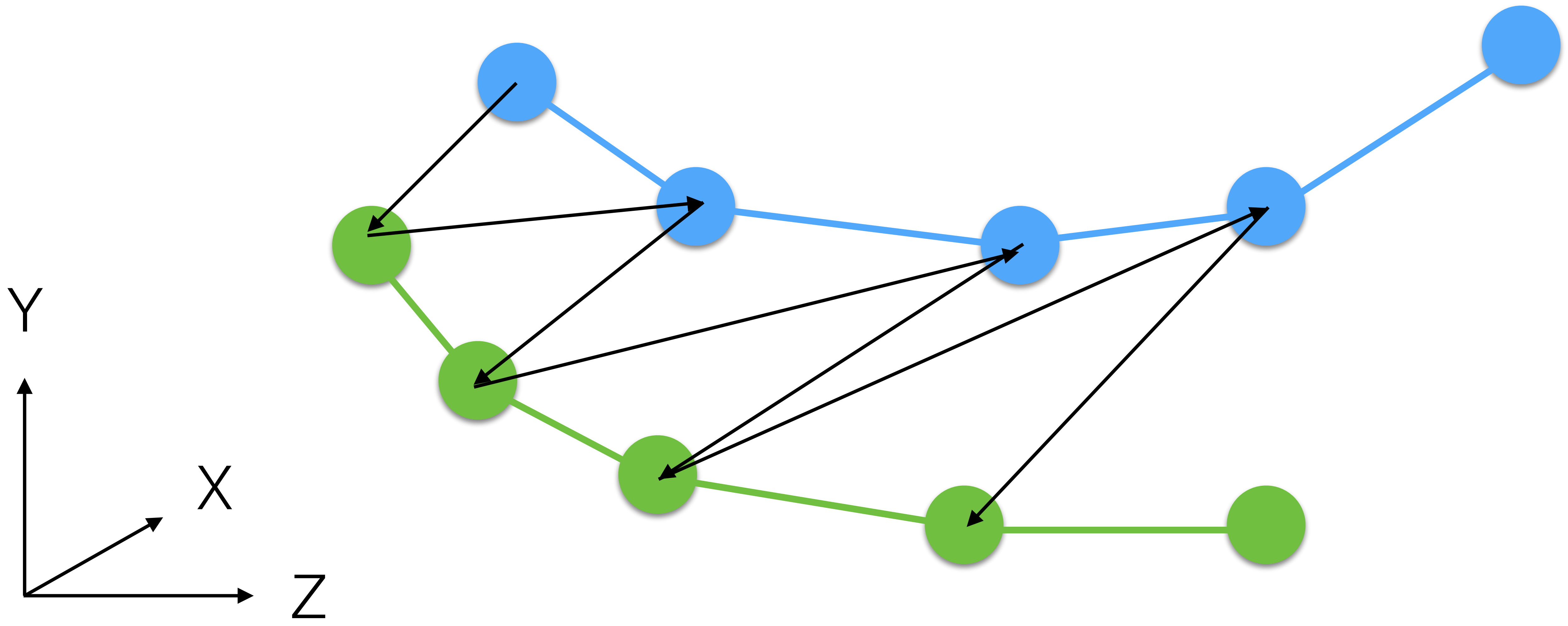
Edges Triangulation: Working Example



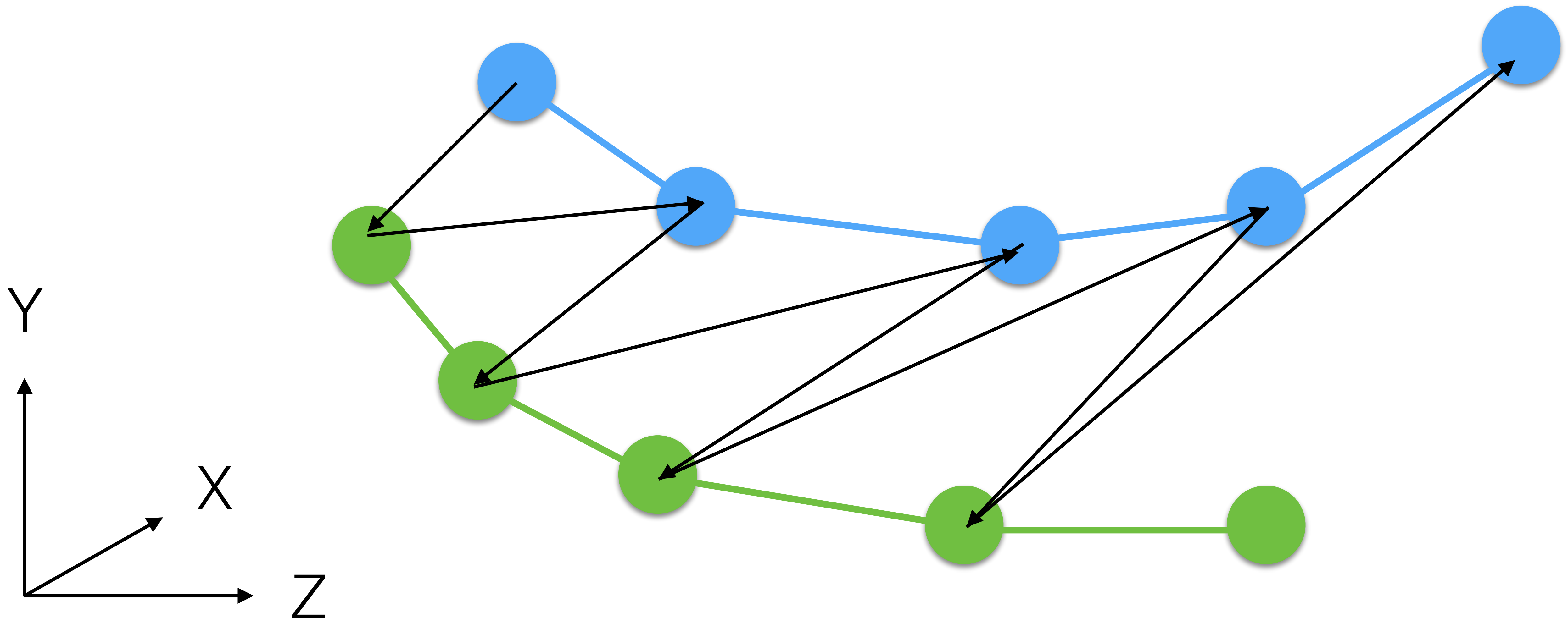
Edges Triangulation: Working Example



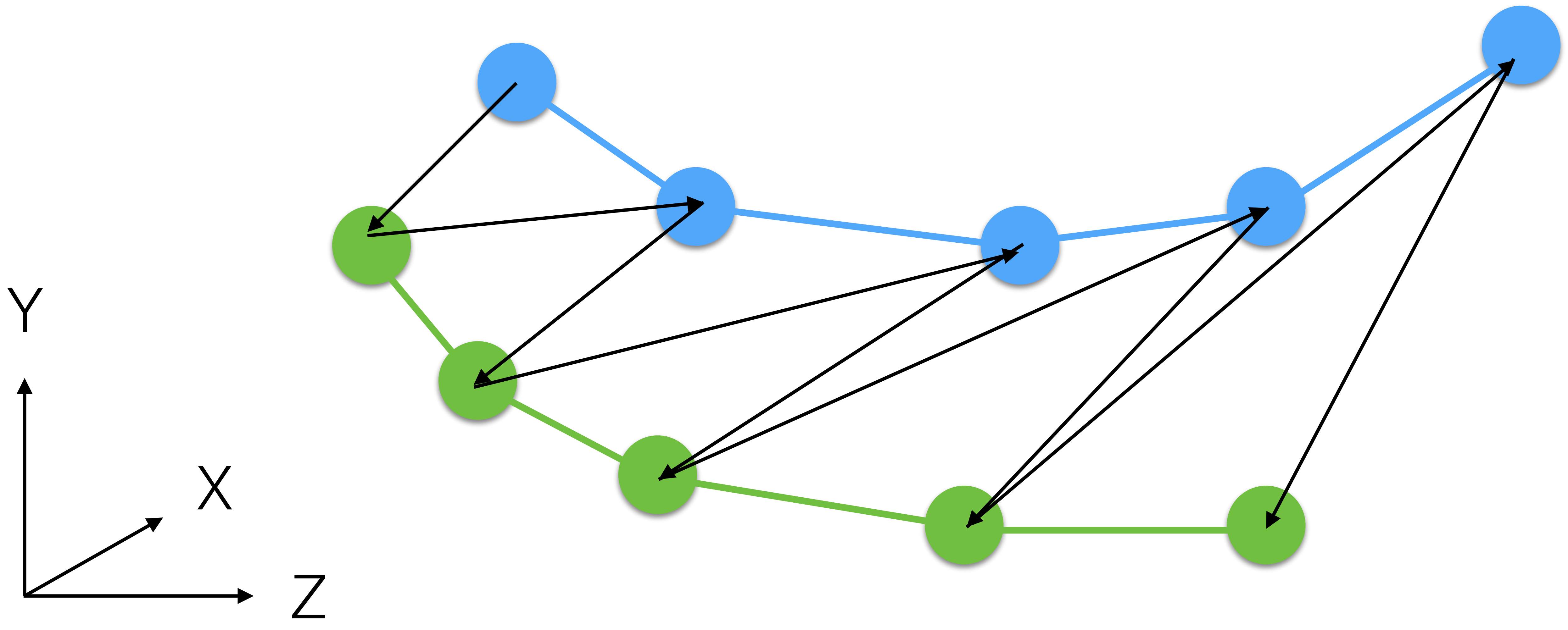
Edges Triangulation: Working Example



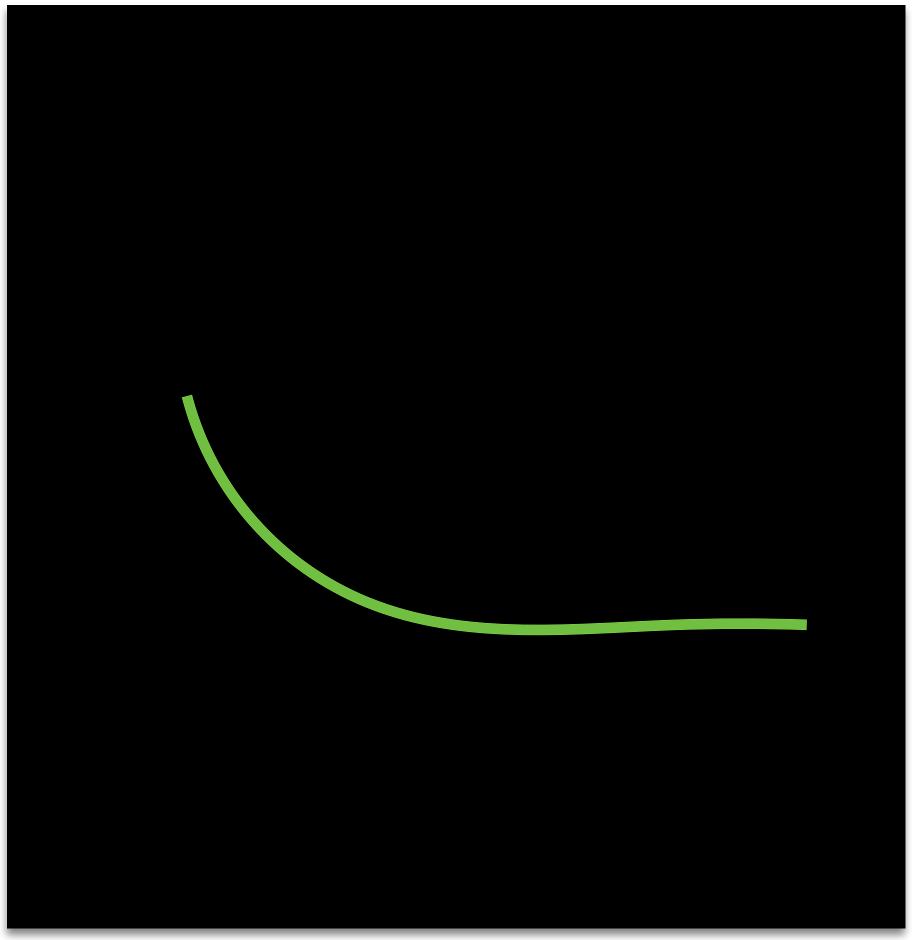
Edges Triangulation: Working Example

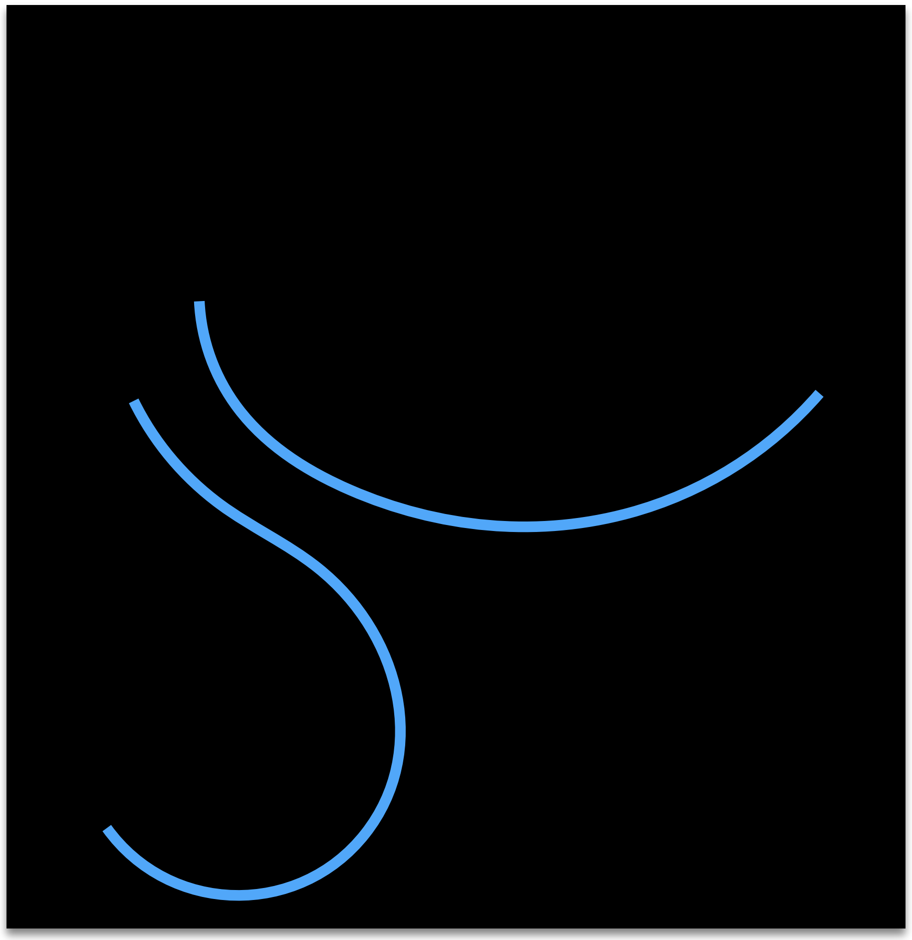


Edges Triangulation: Working Example

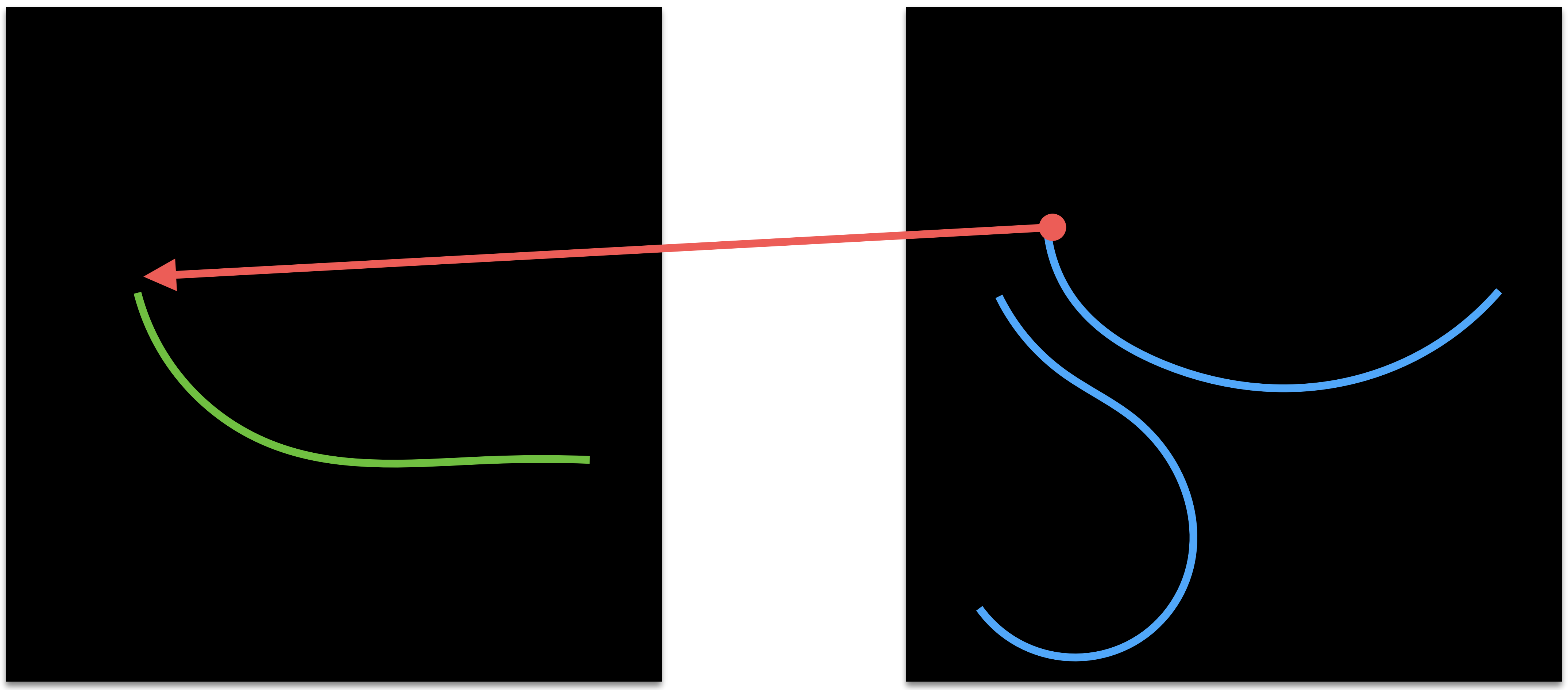


Edges Triangulation: Failure Case


Slice 1


Slice 2

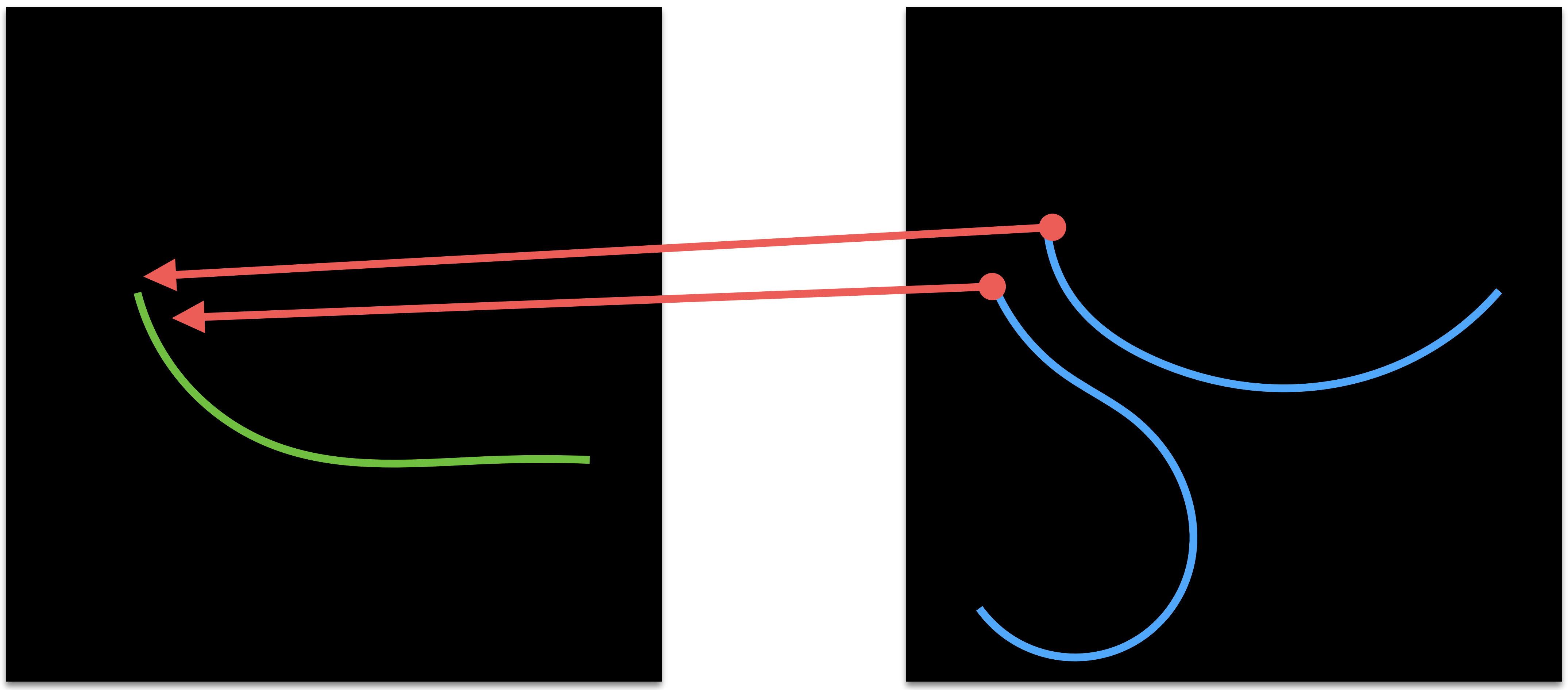
Edges Triangulation: Failure Case



Slice 1

Slice 2

Edges Triangulation: Failure Case


Slice 1

Slice 2

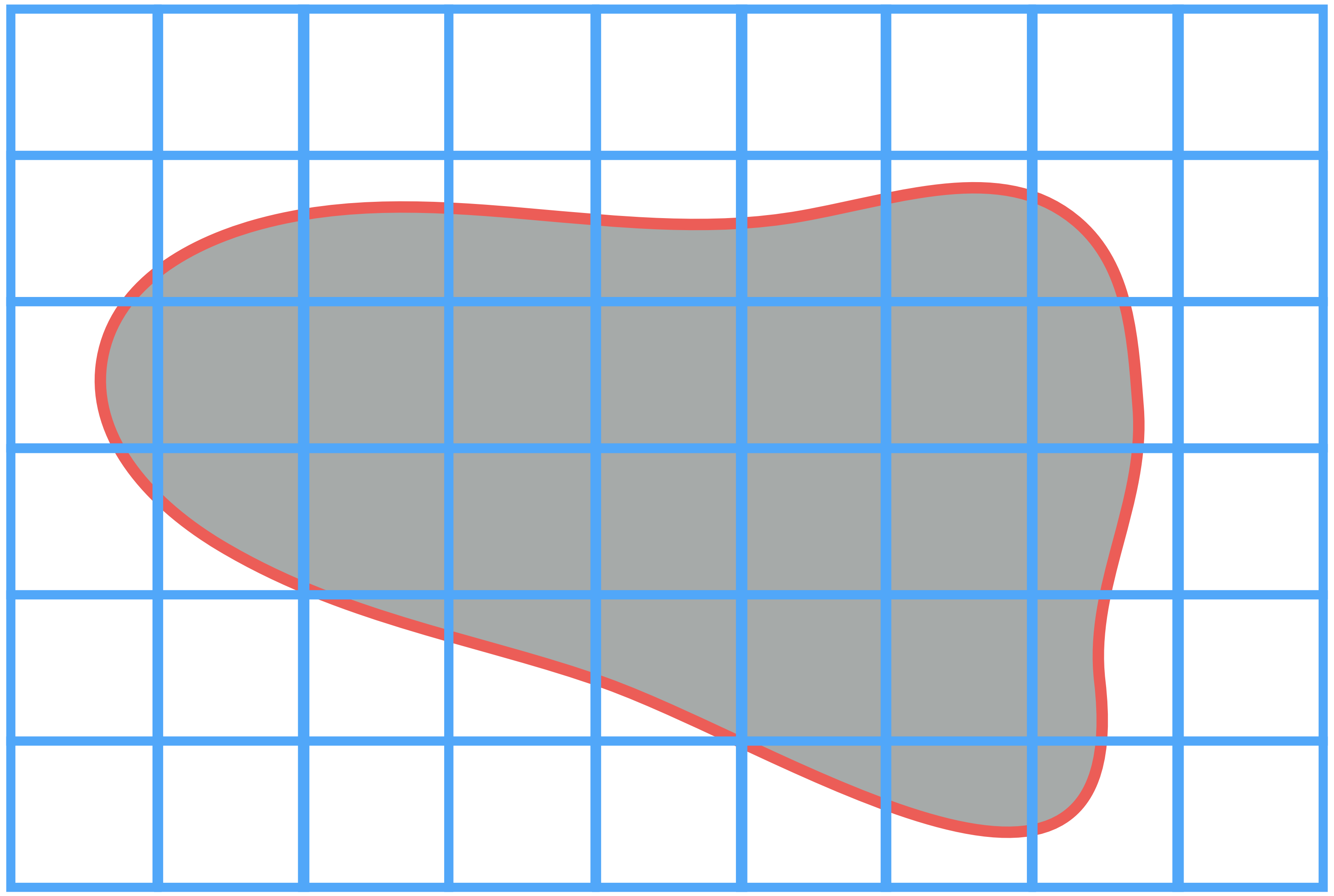
Edges Triangulation

- 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 (top and bottom); i.e., it is not watertight!

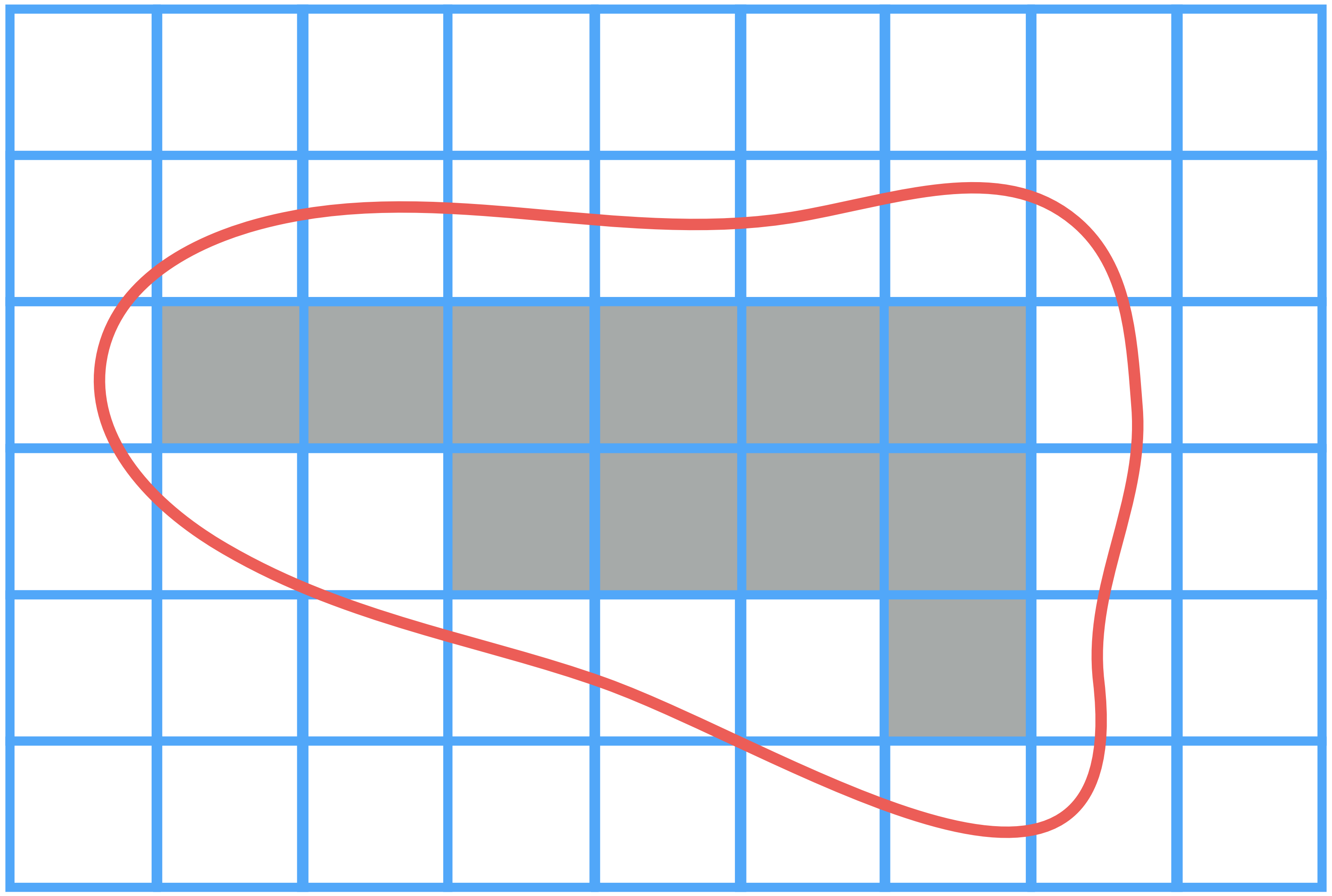
Marching Cubes

Let’s start in 2D

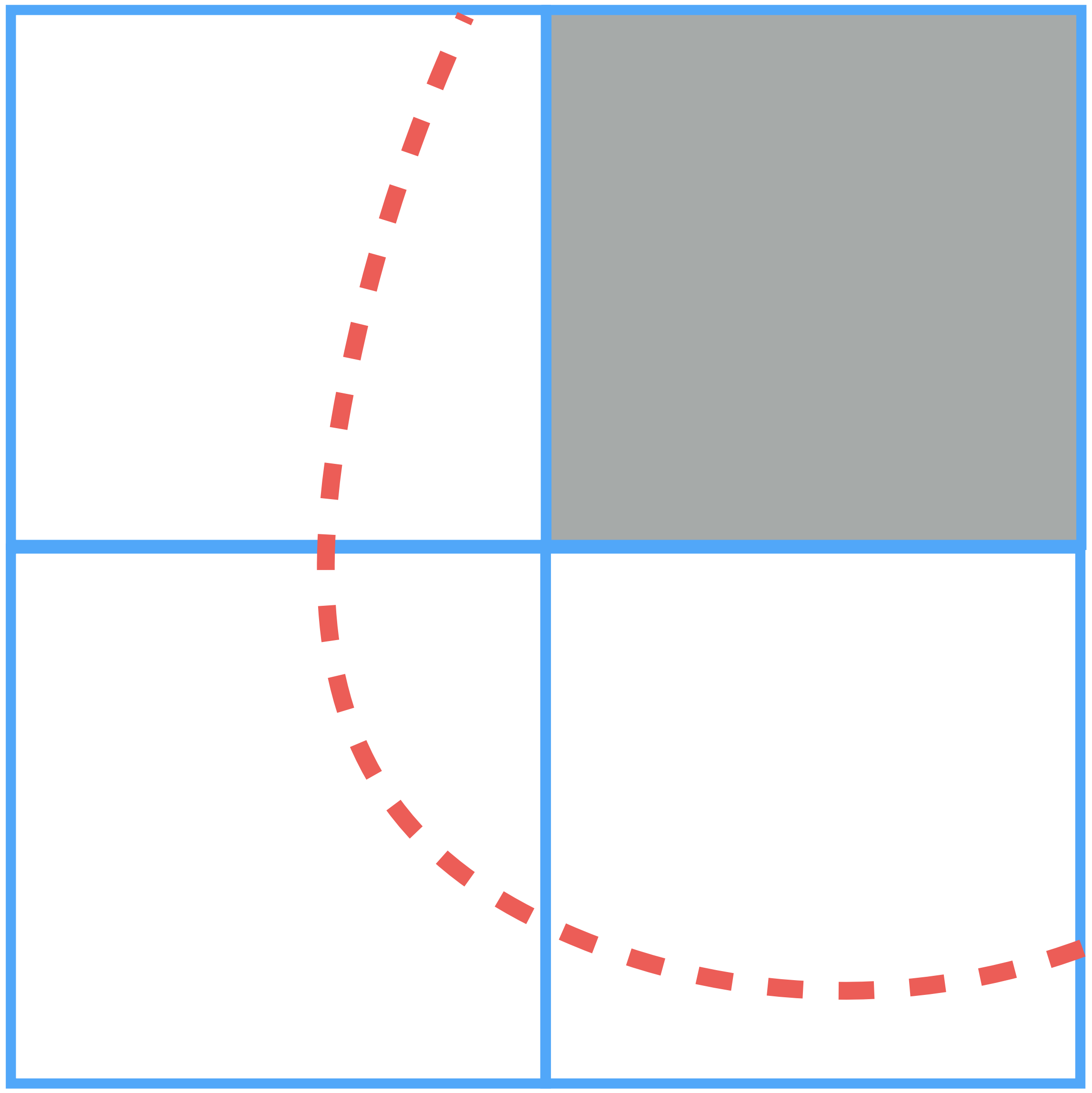
Marching Squares



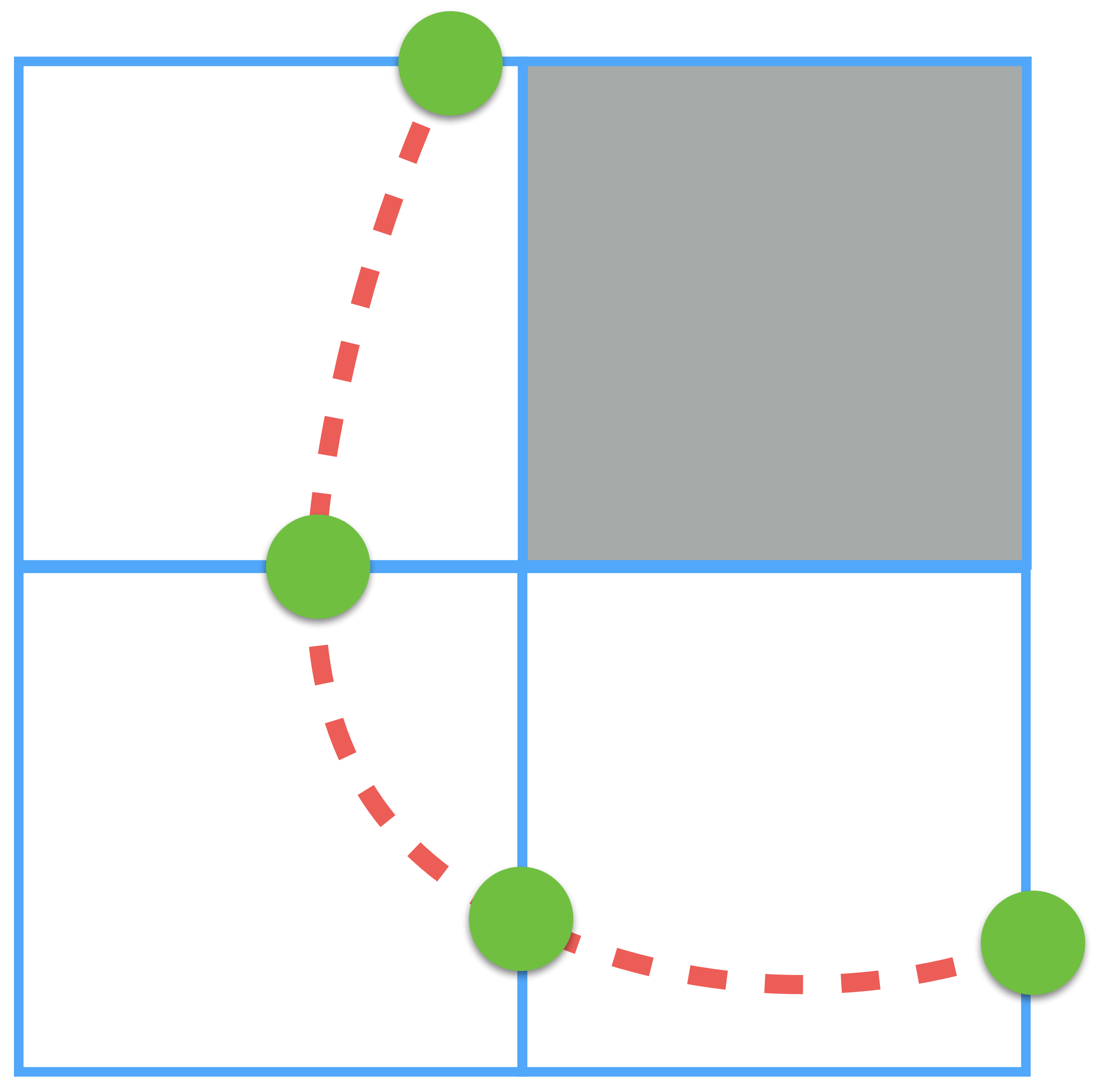
Marching Squares


Segmentation Result in 2D

Marching Squares

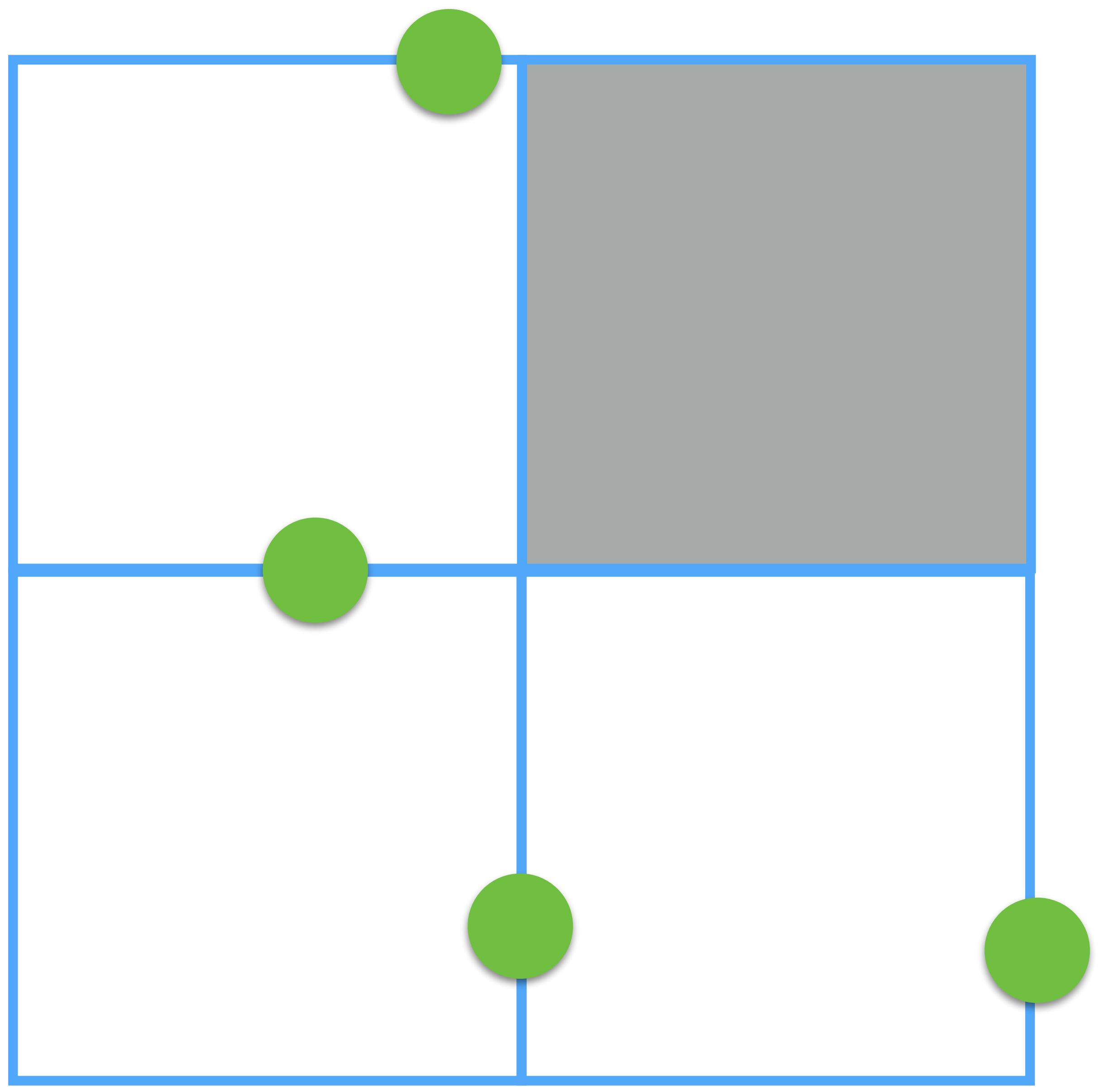


Marching Squares

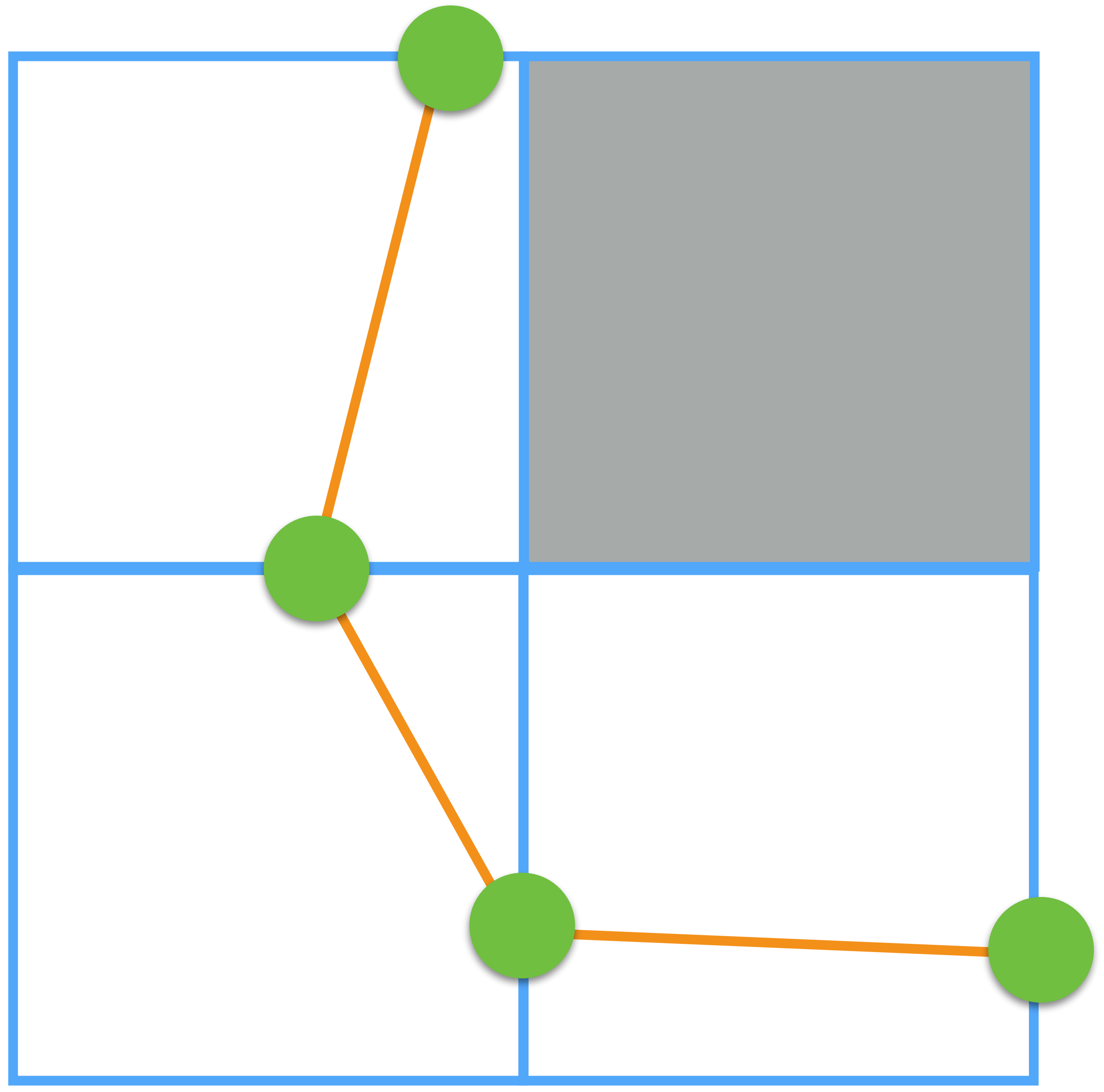


● Ideal intersection point

Marching Squares

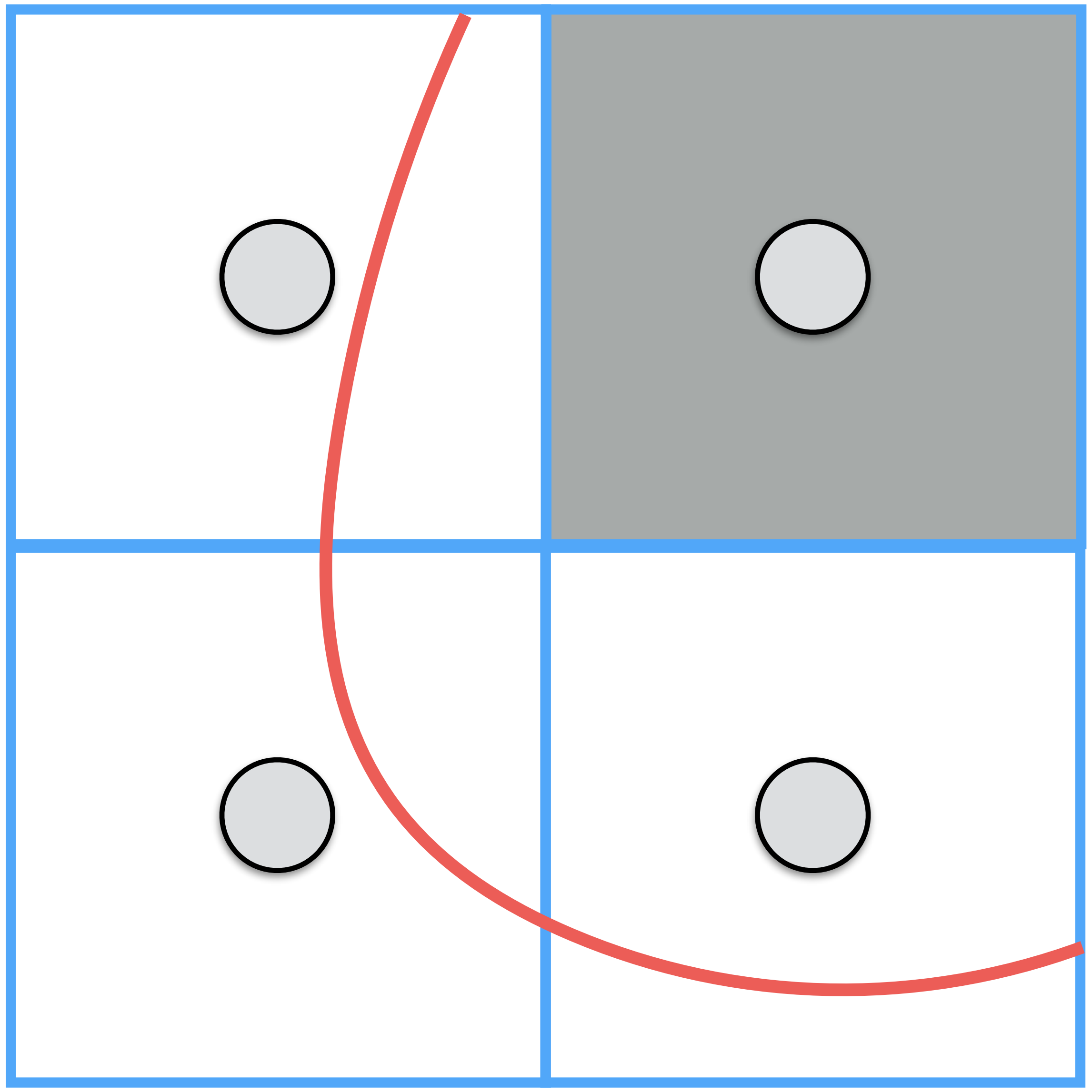


Marching Squares

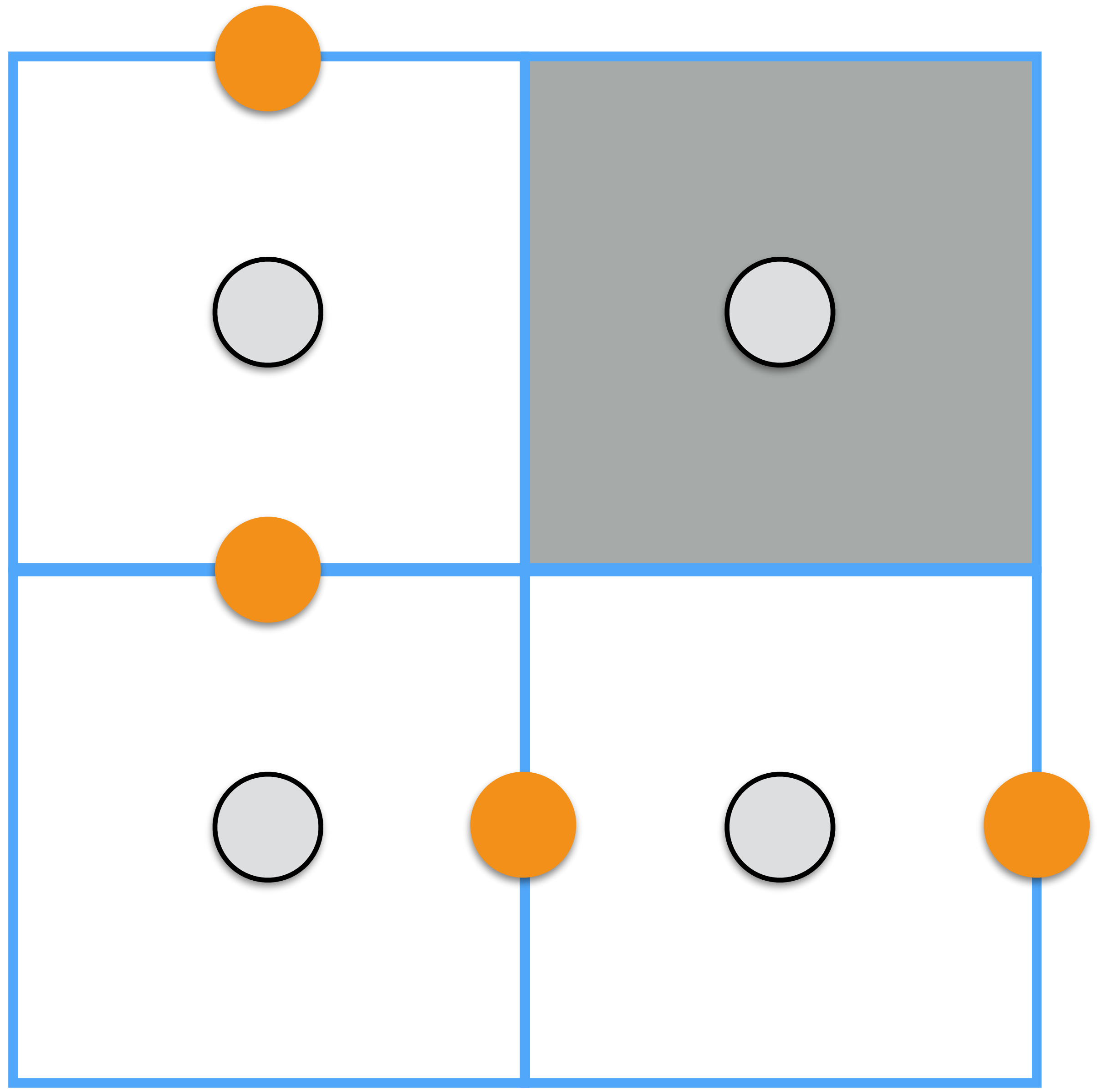


— Ideal piecewise line

Marching Squares

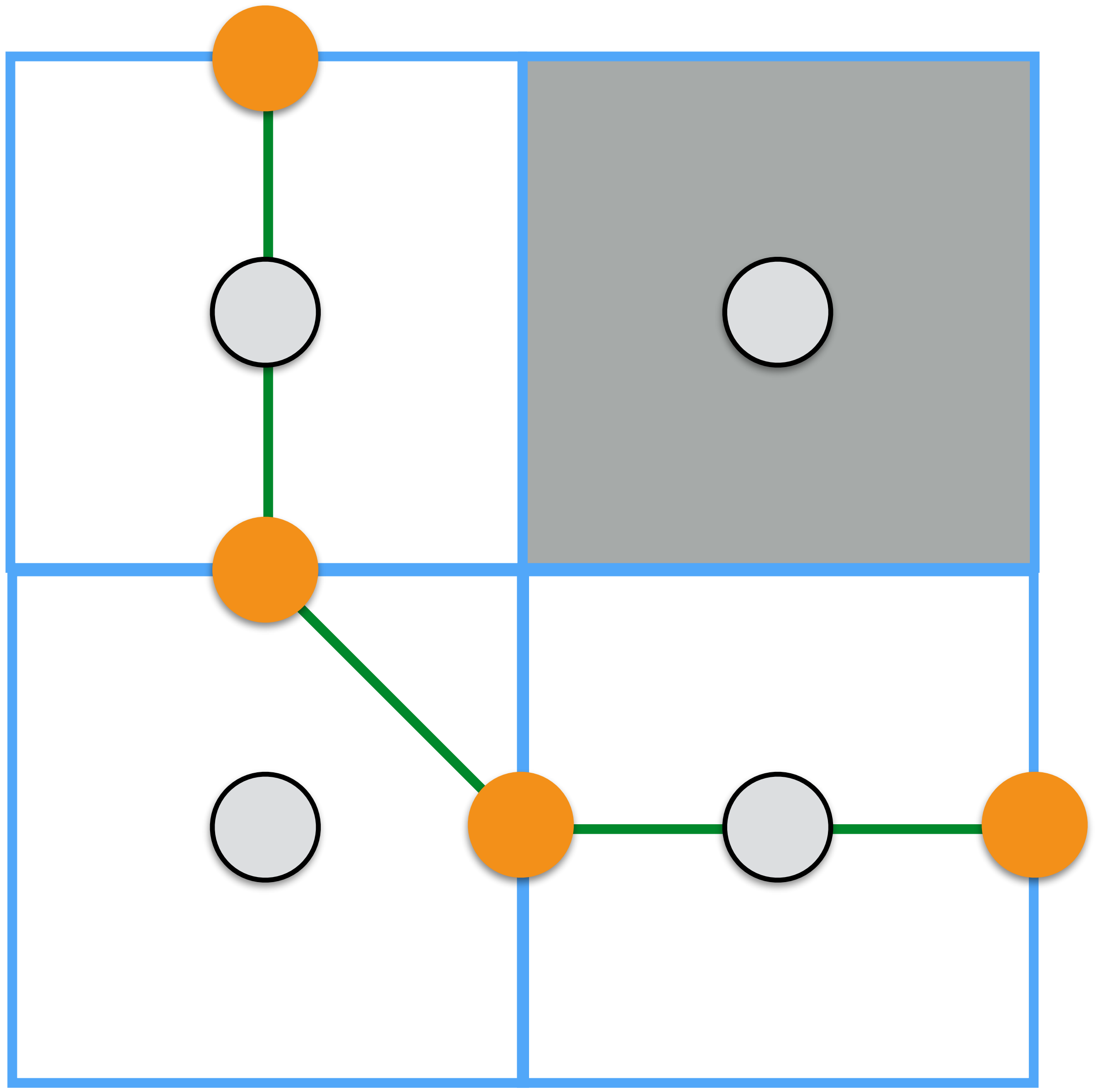


Marching Squares



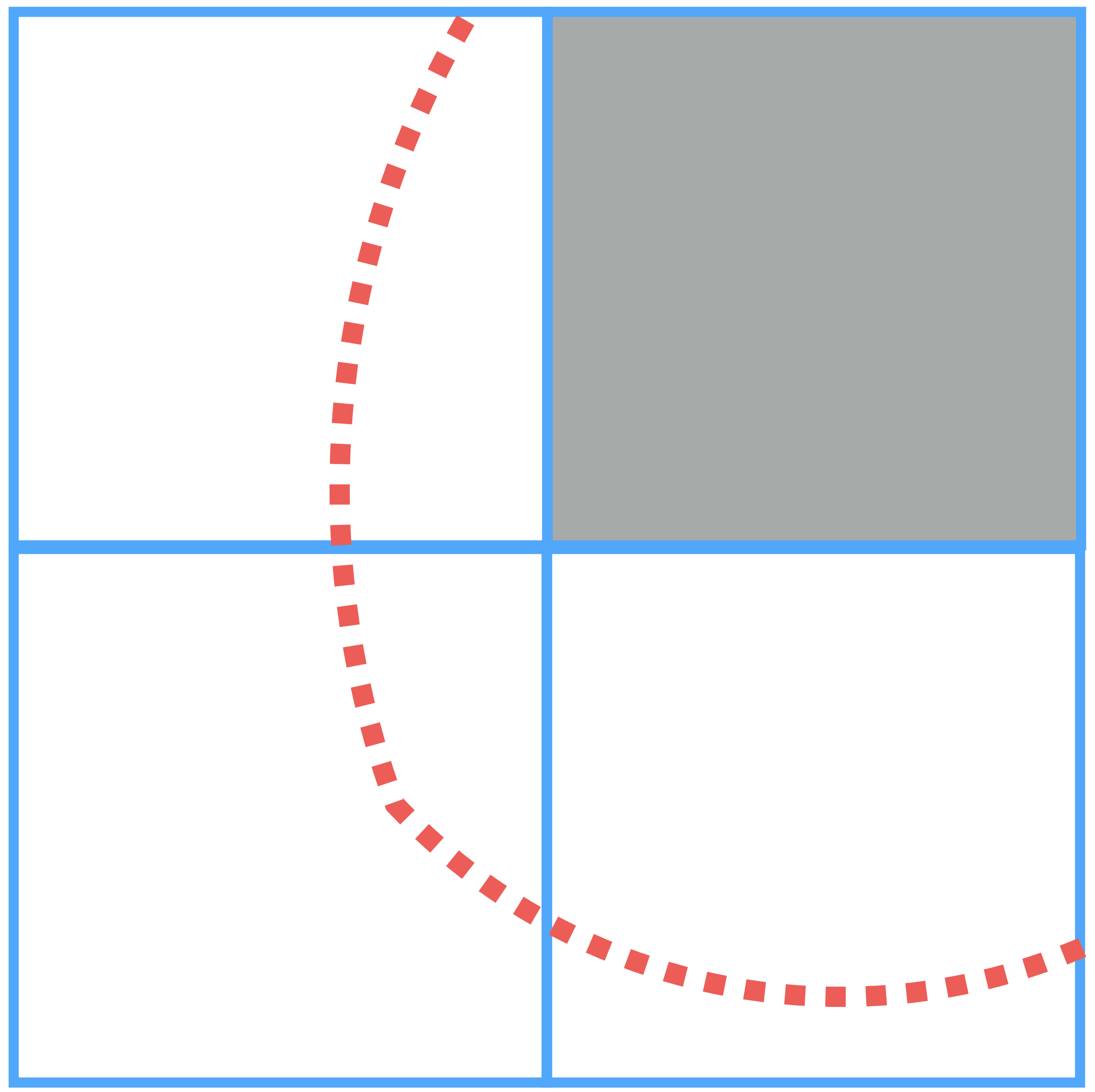
● Best guess when not knowing the original shape of the curve!

Marching Squares



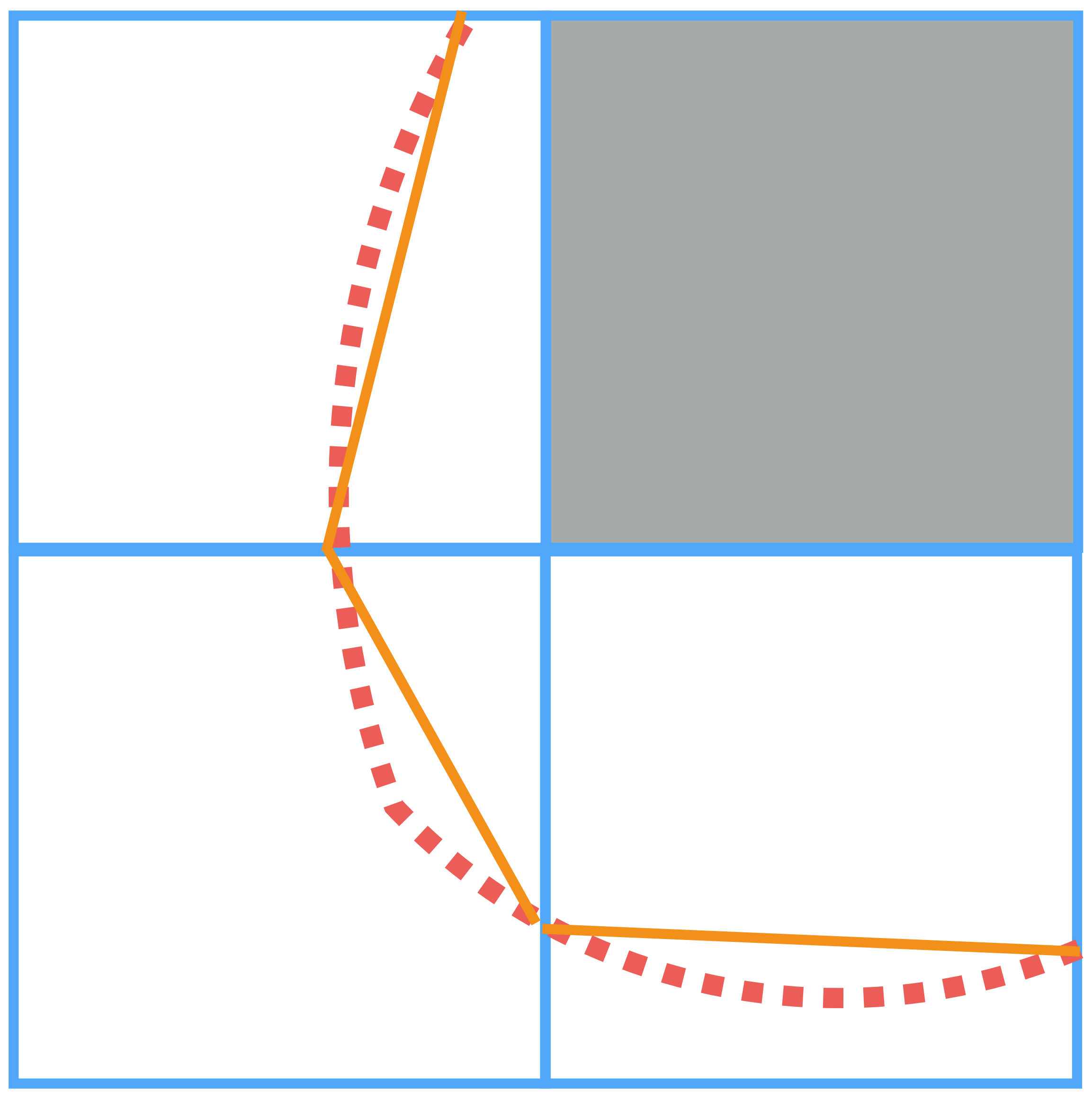
🟥 Marching Squares

Marching Squares



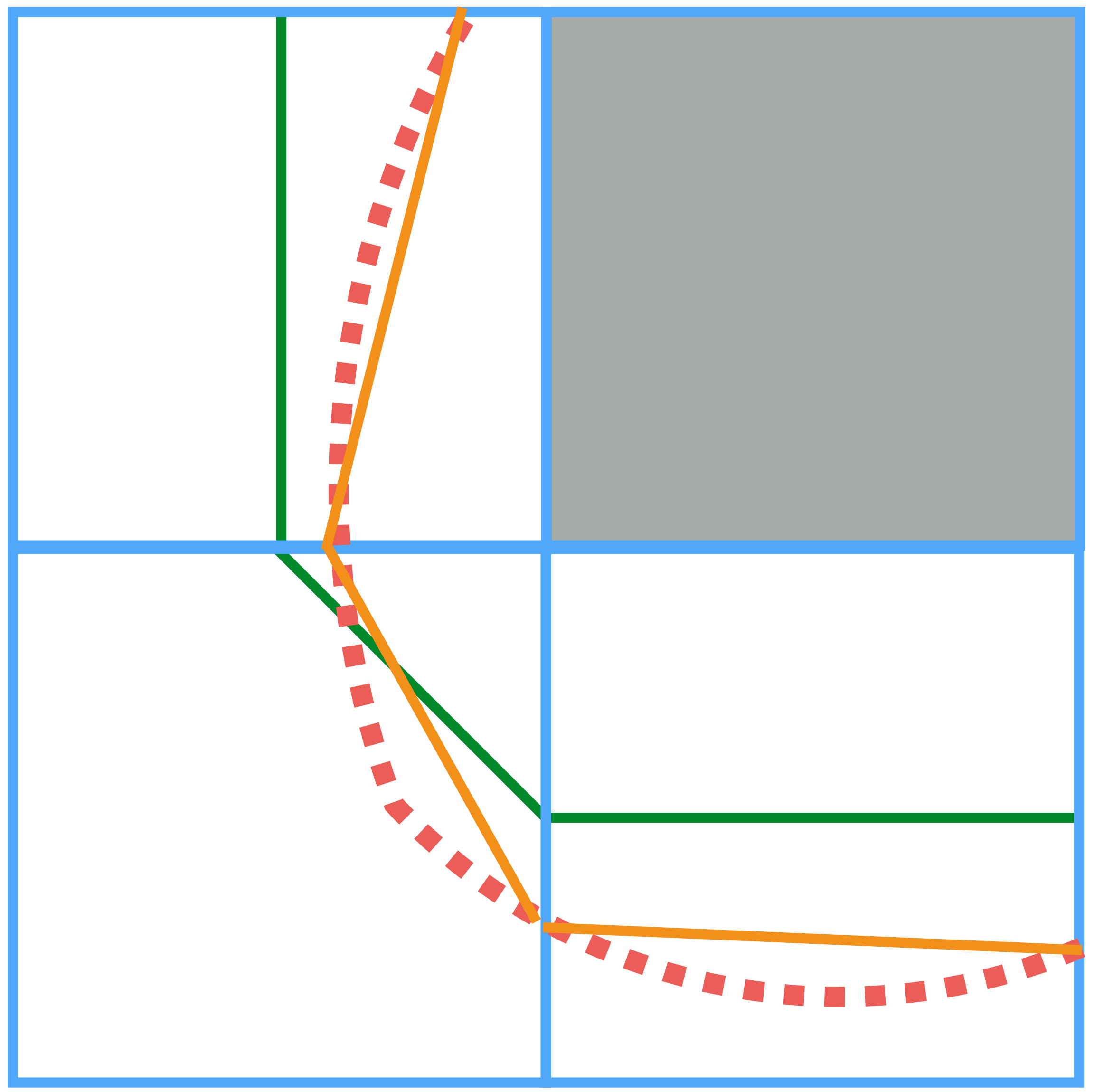
Real boundary
Ideal piece-wise line
Marching squares

Marching Squares



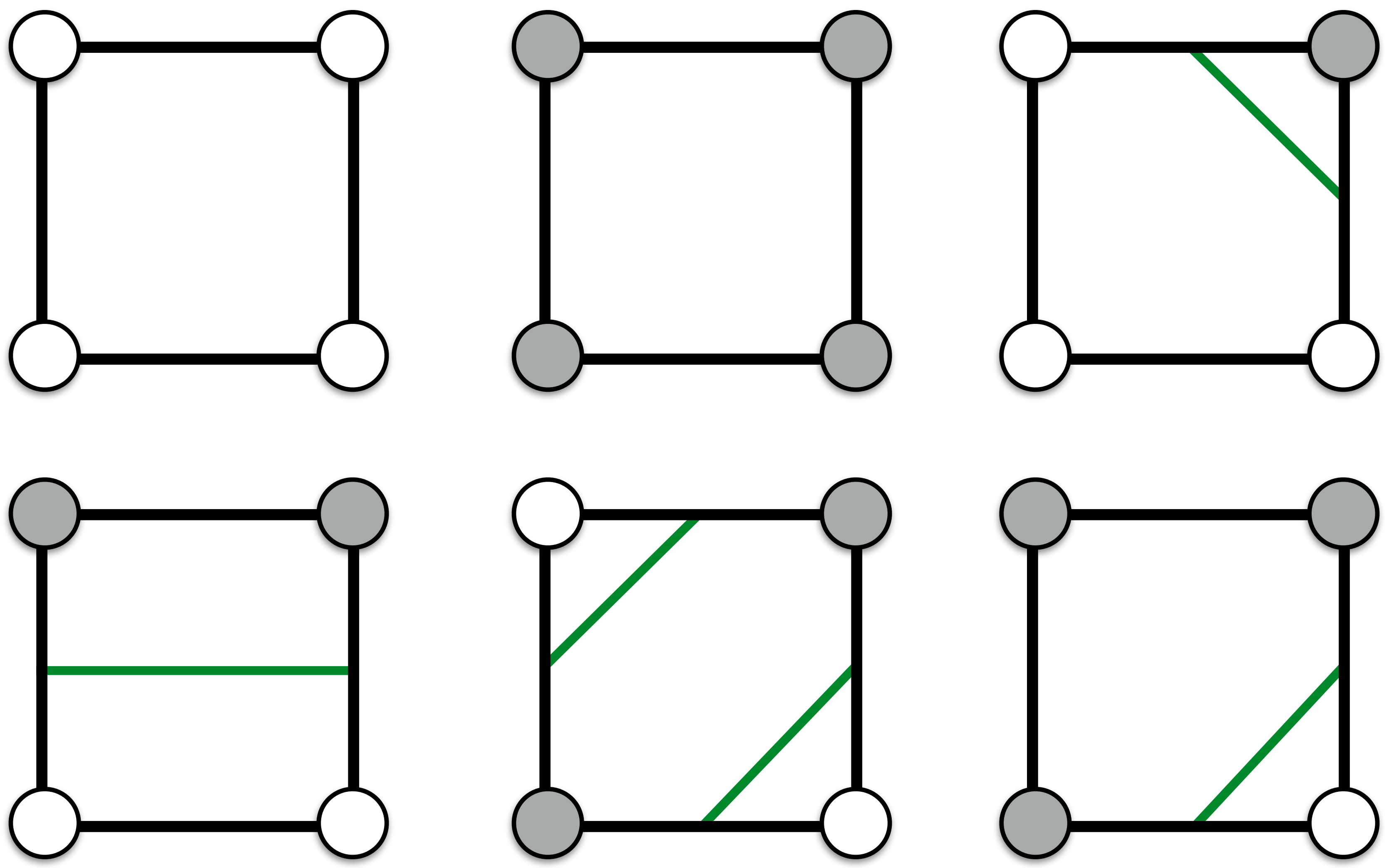
Real boundary
Ideal piece-wise line
Marching squares

Marching Squares



Real boundary
Ideal piece-wise line
Marching squares

Marching Squares: Cases

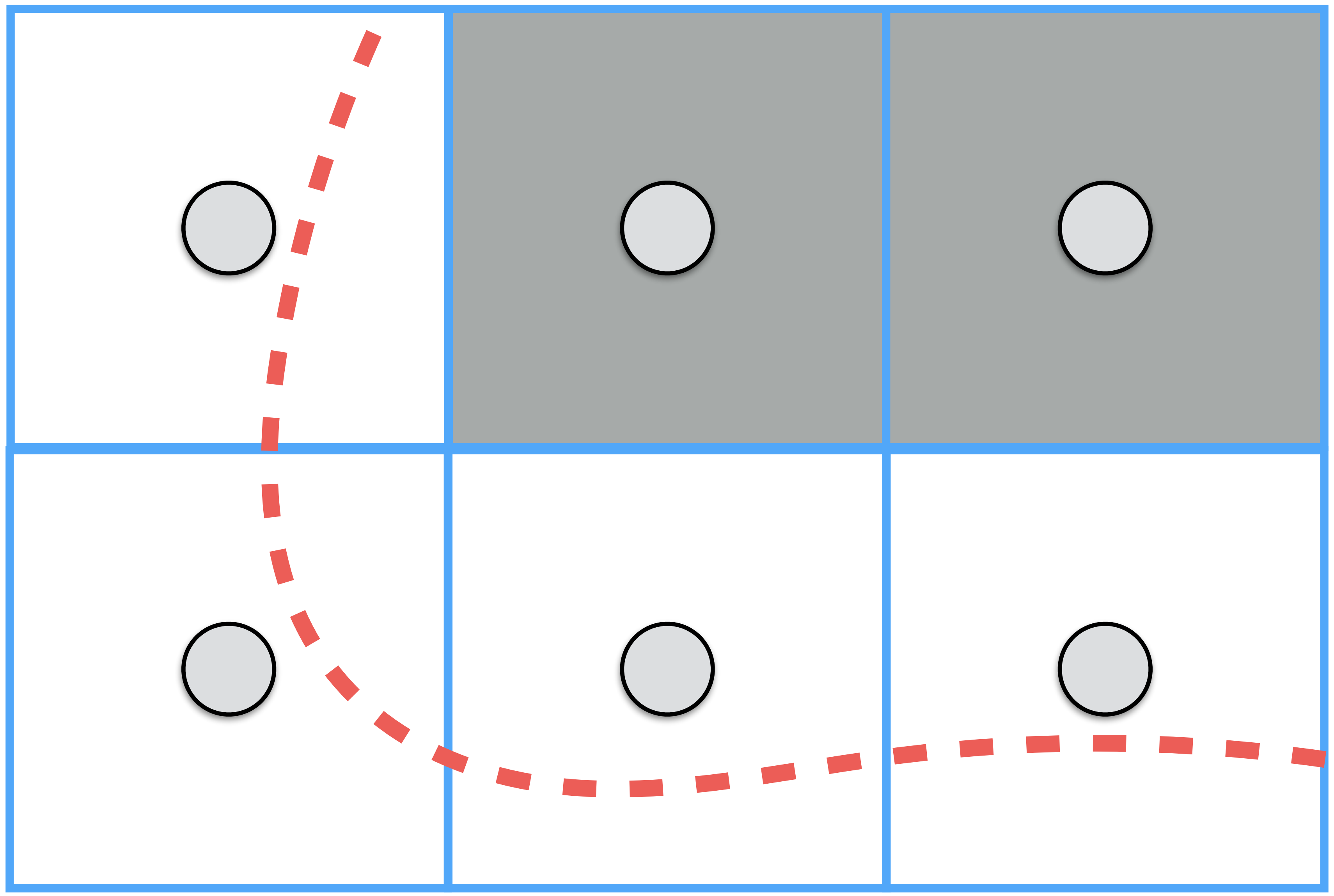


There are in total 16 (2^4) configurations, the other ones can be computed by rotating or reflecting these.

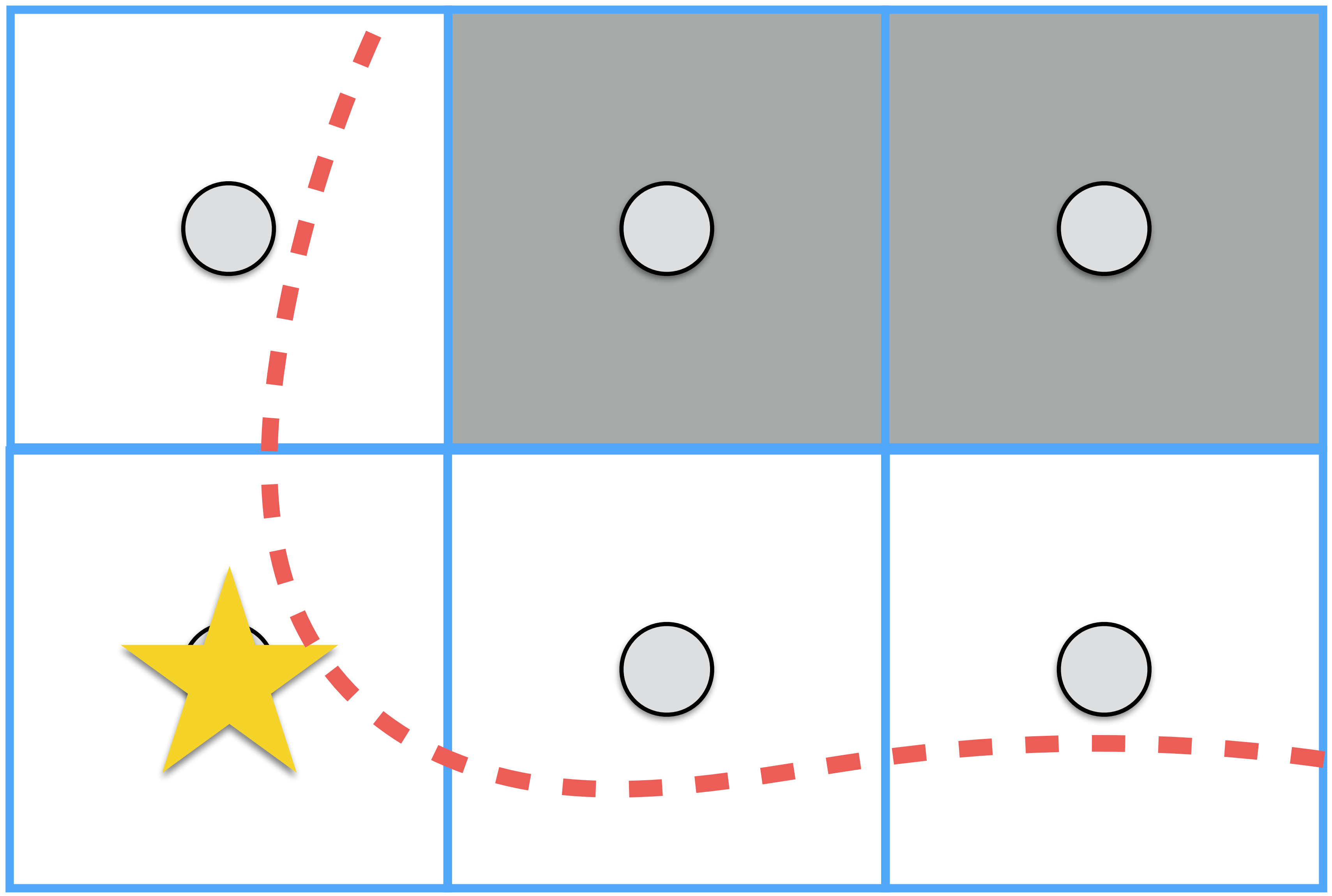
Marching Squares

- 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.

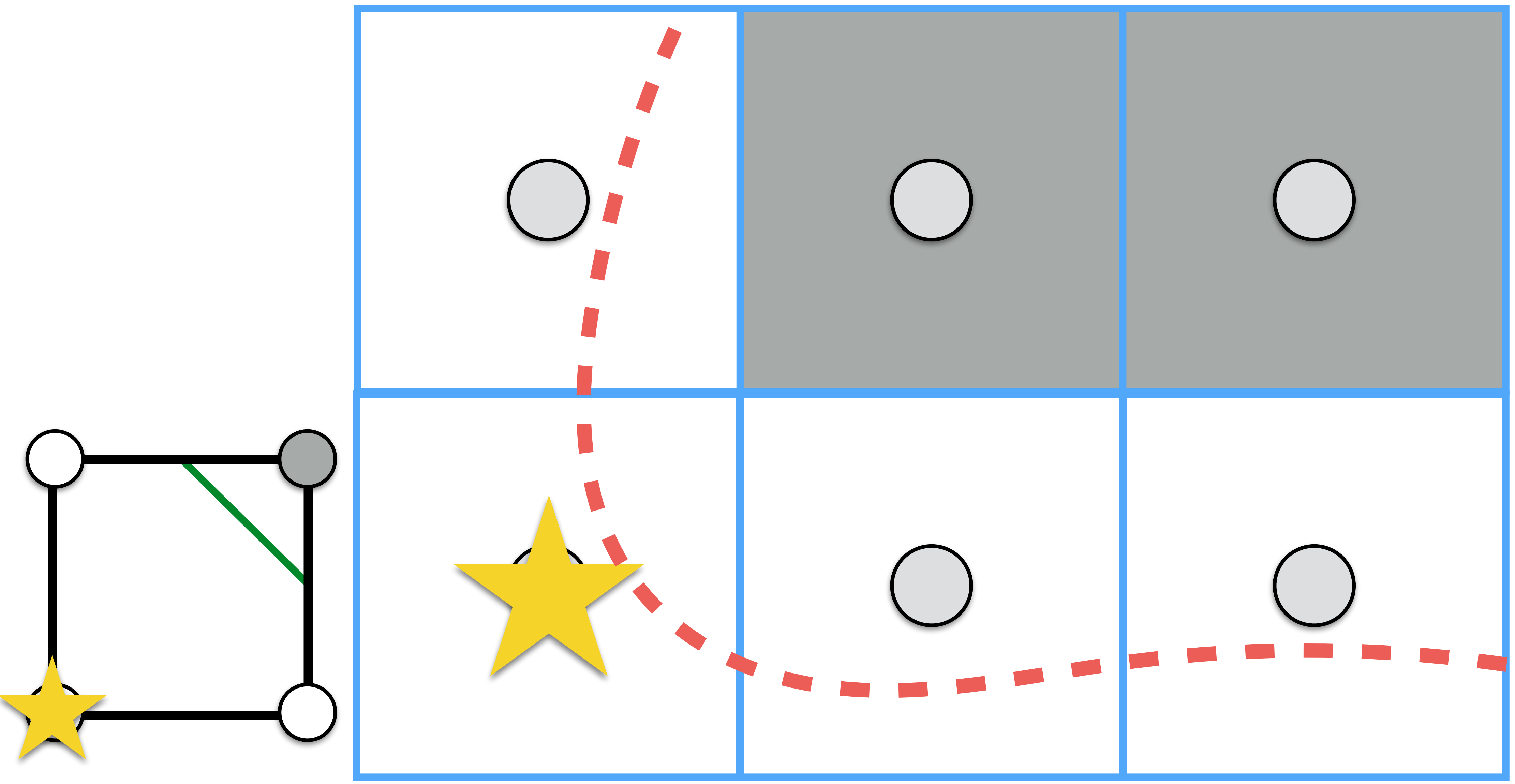
Marching Squares Example



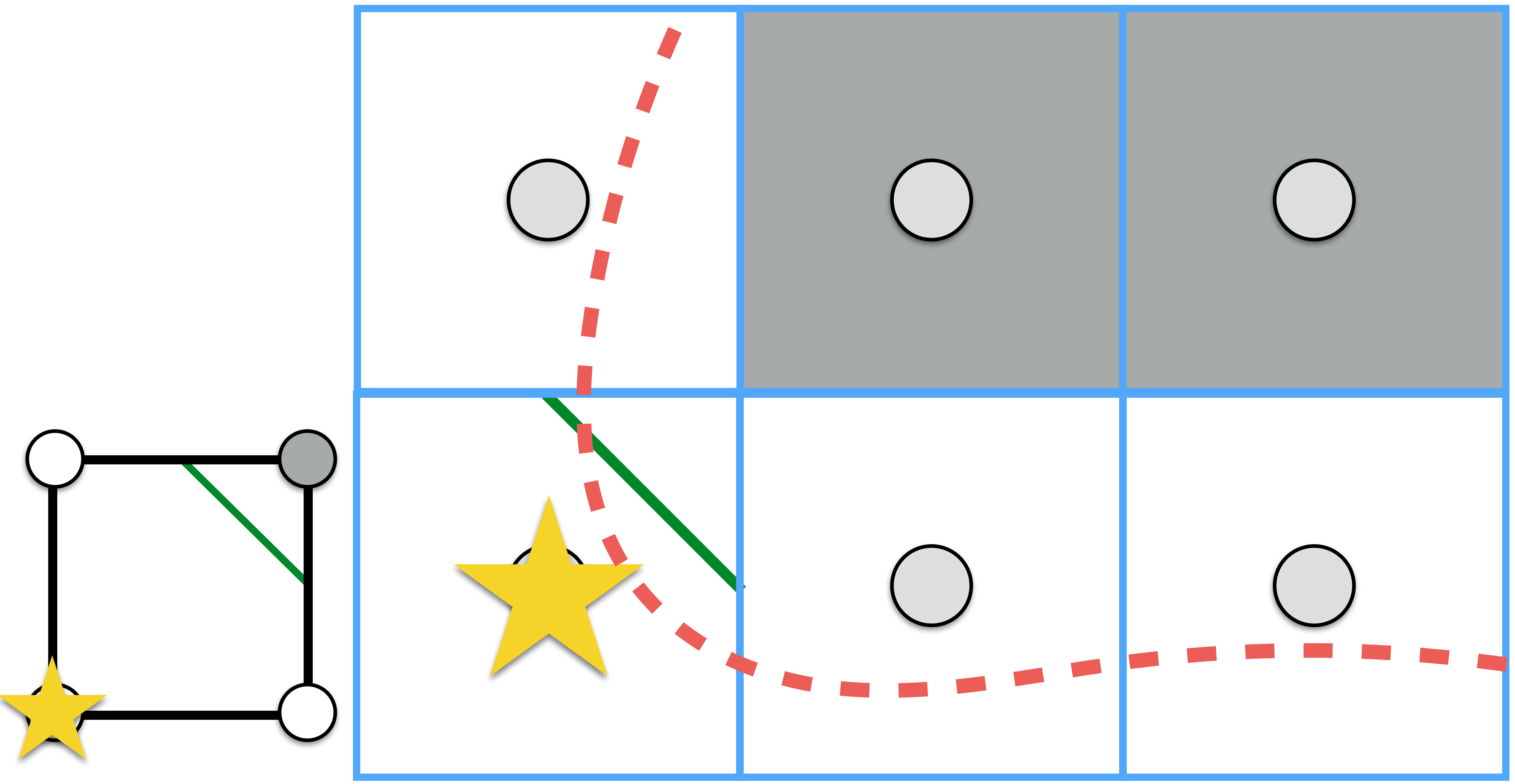
Marching Squares Example



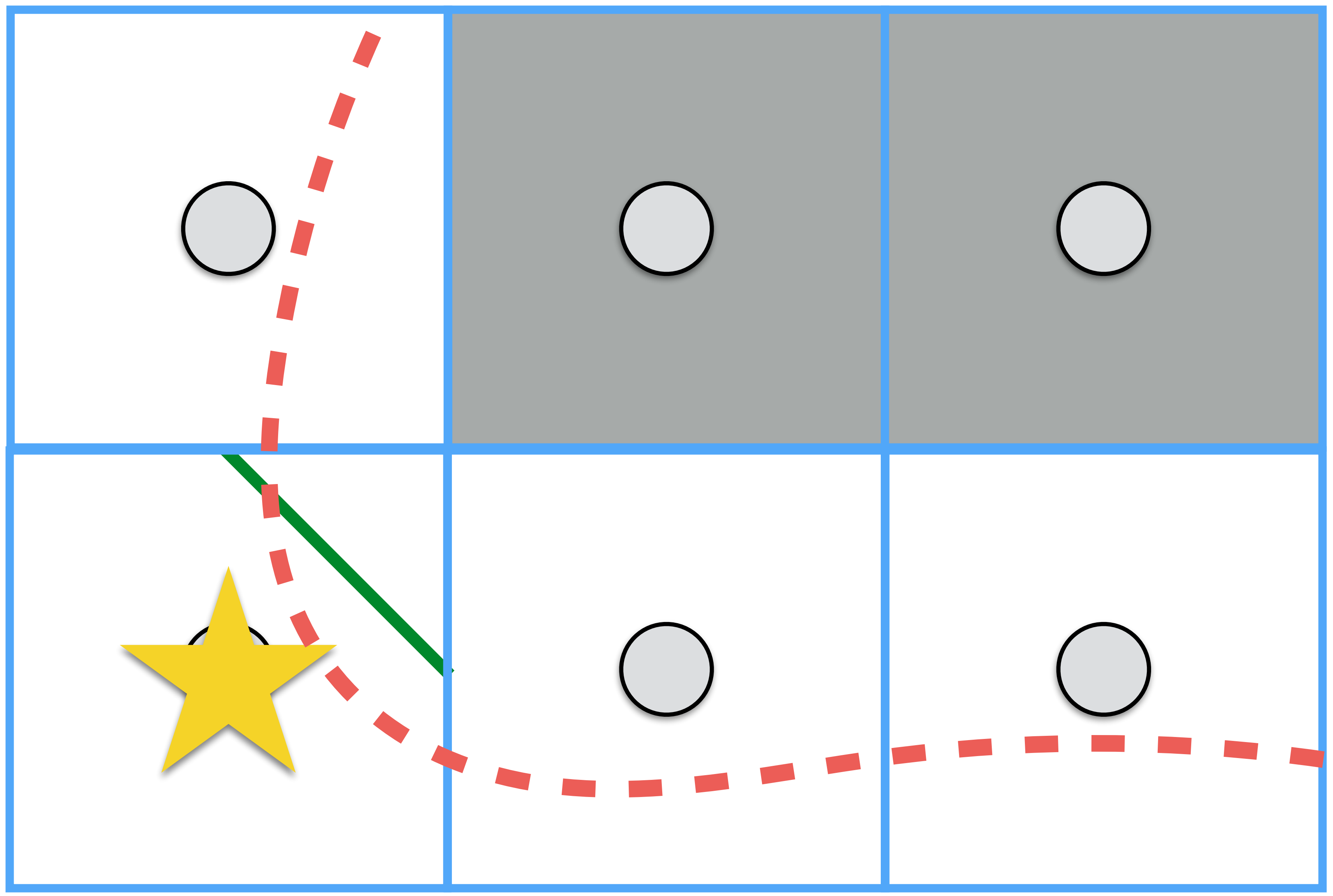
Marching Squares Example



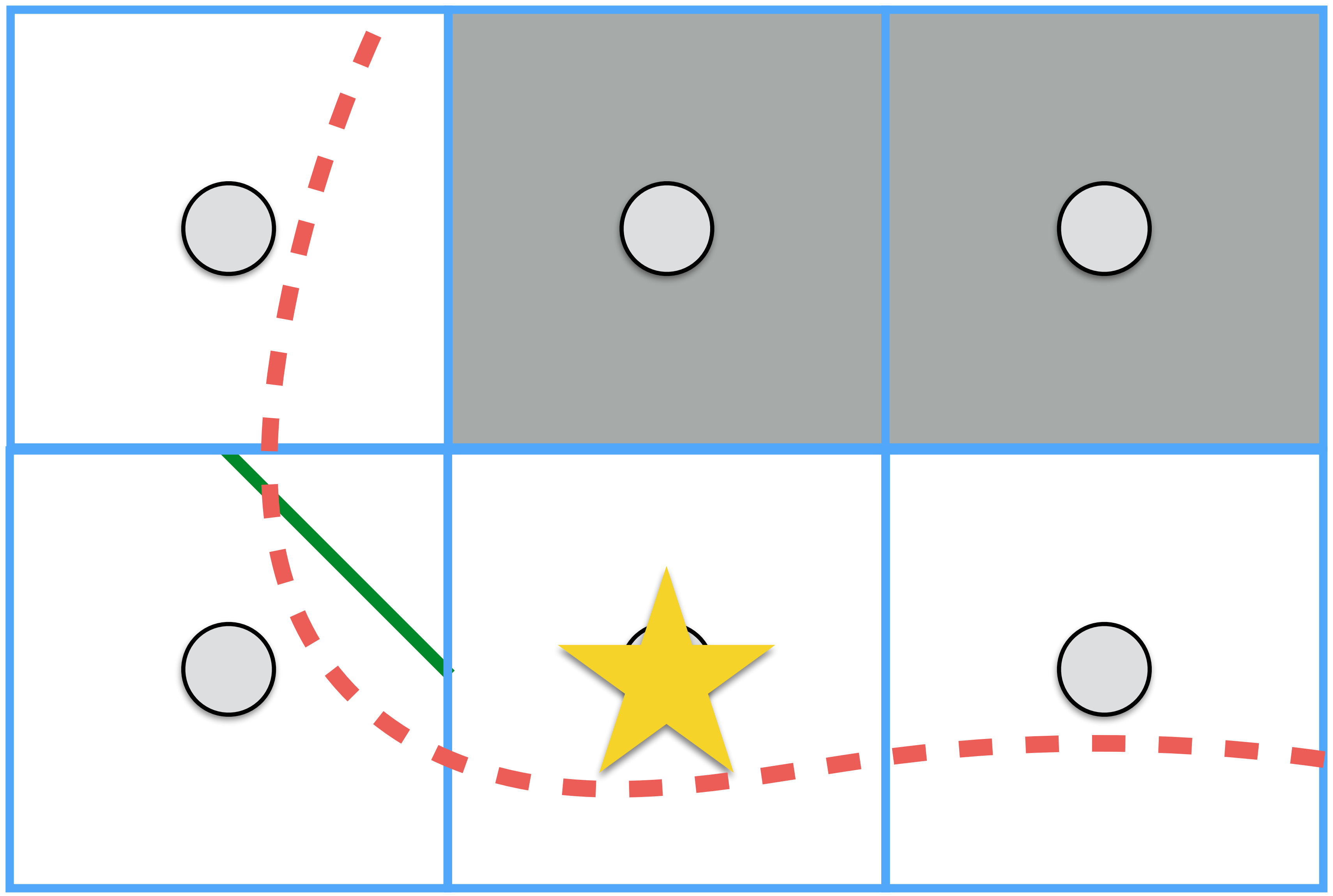
Marching Squares Example



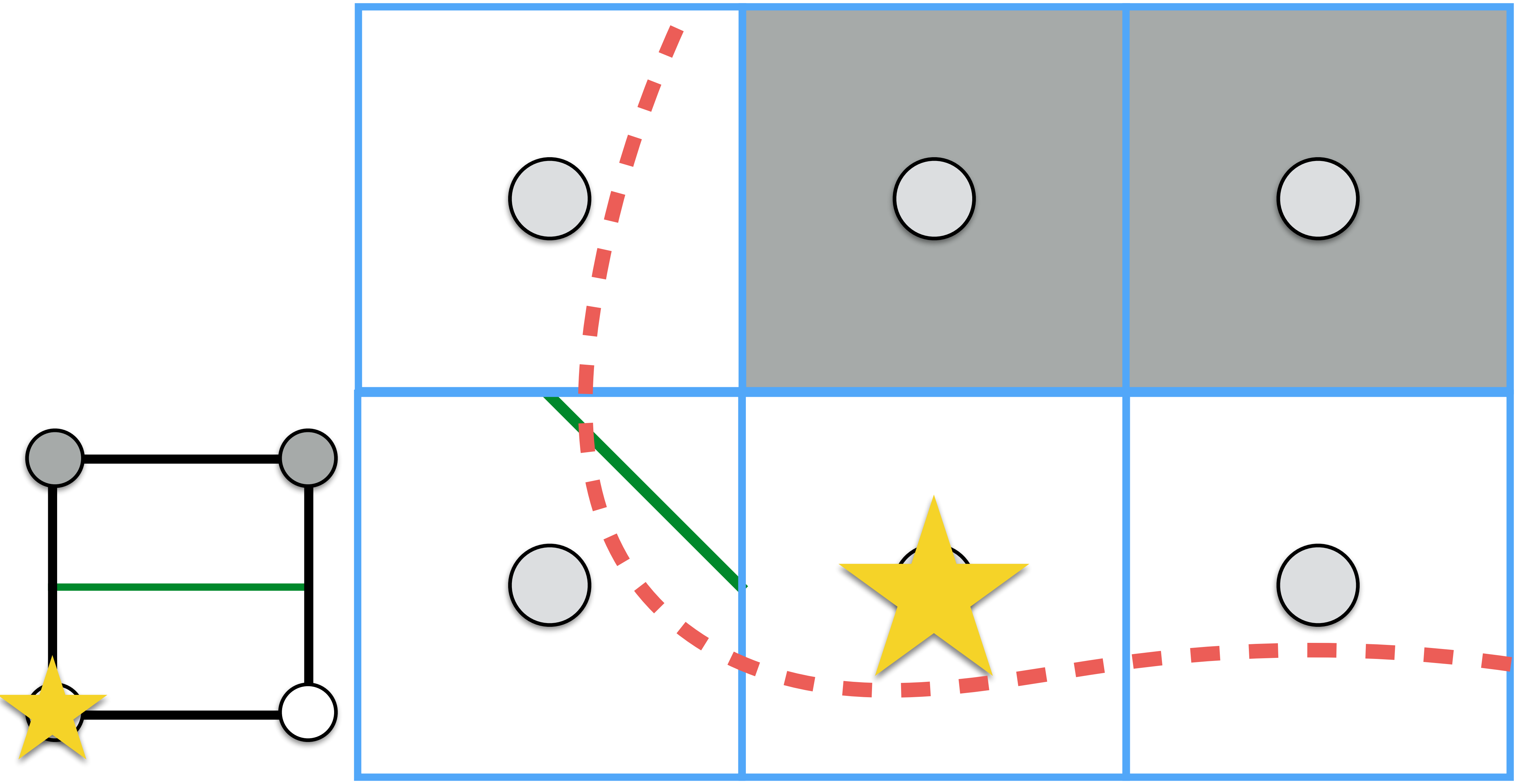
Marching Squares Example



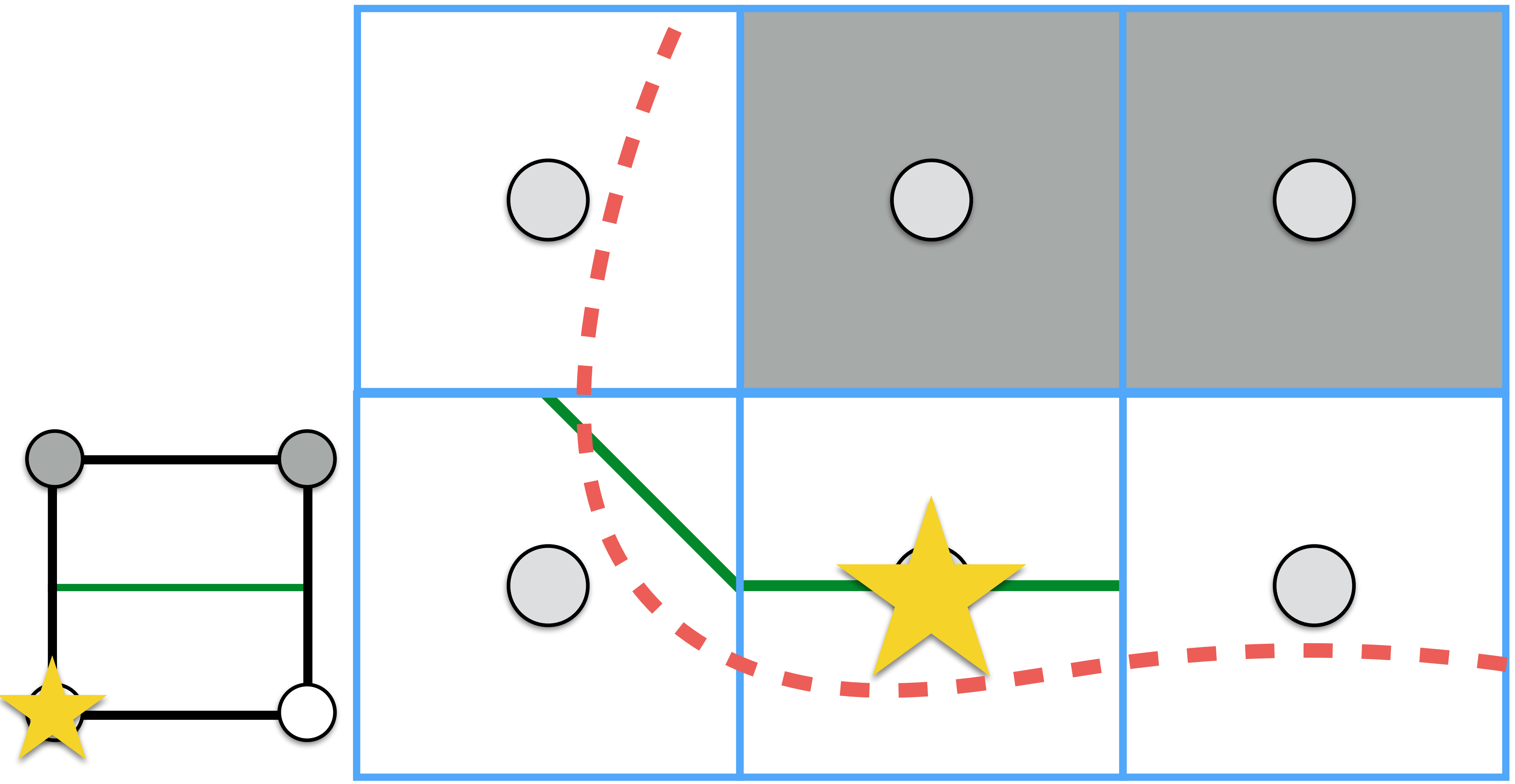
Marching Squares Example



Marching Squares Example



Marching Squares Example

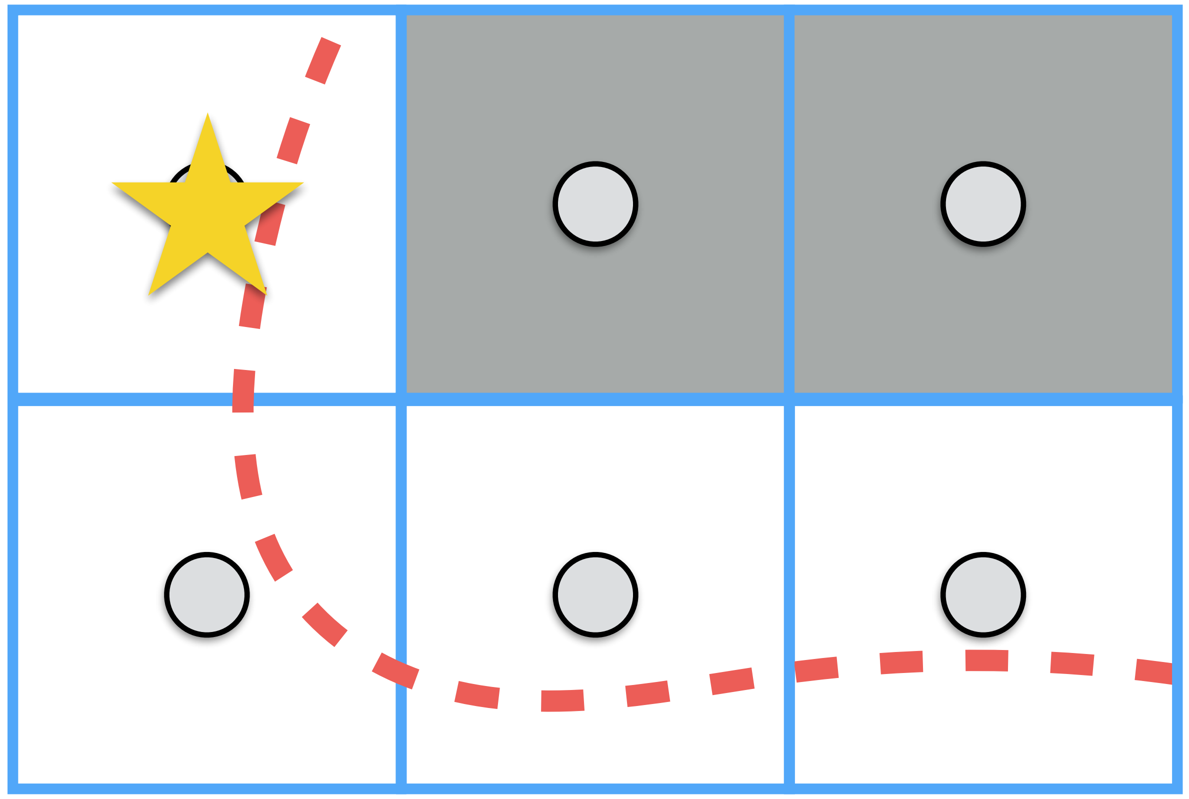


Marching Squares: Boundaries

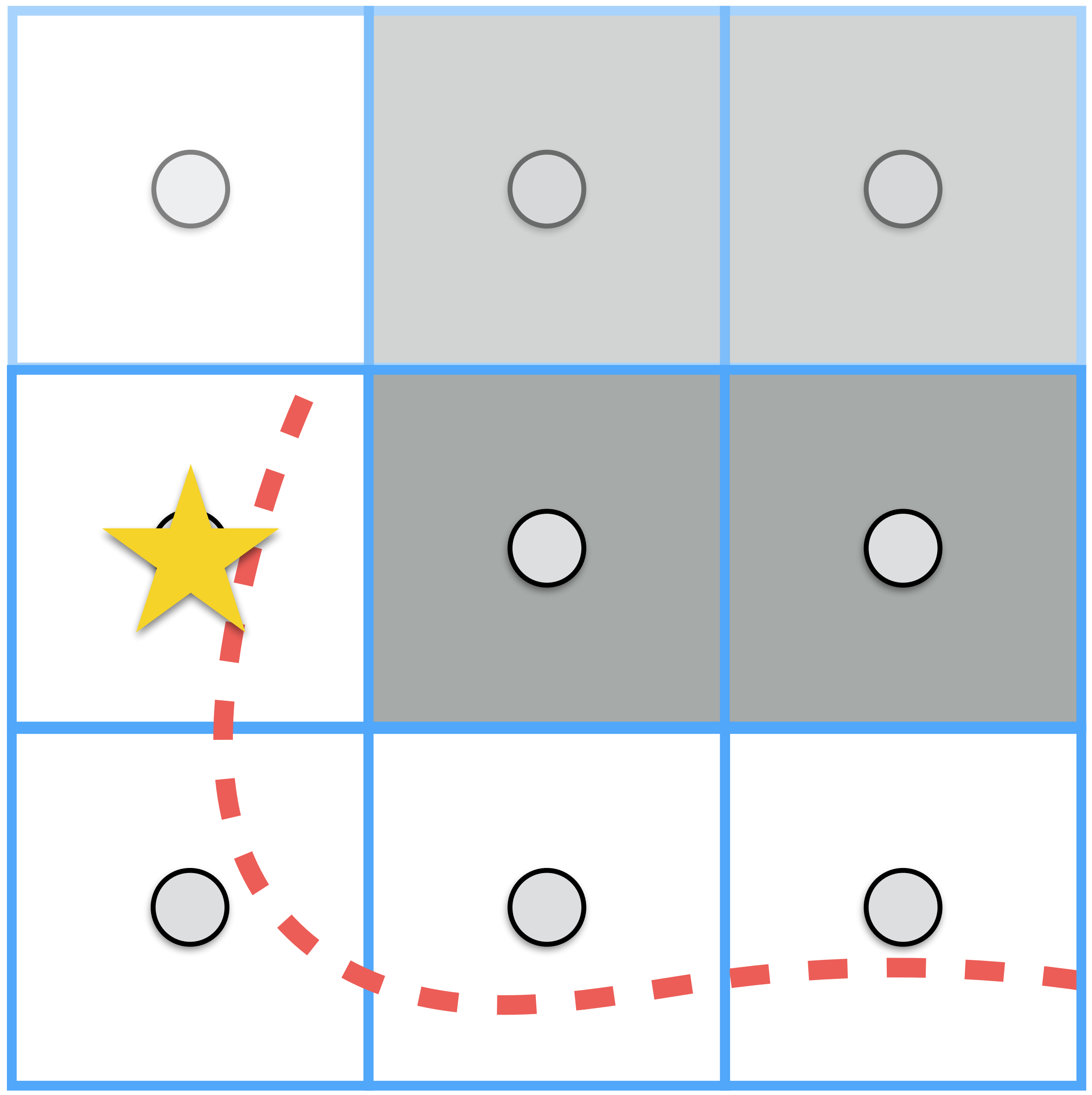
- In theory, the object of our interest should be inside the volume without touching boundaries.

- However, we can have cases where the segmentation is touching boundaries!

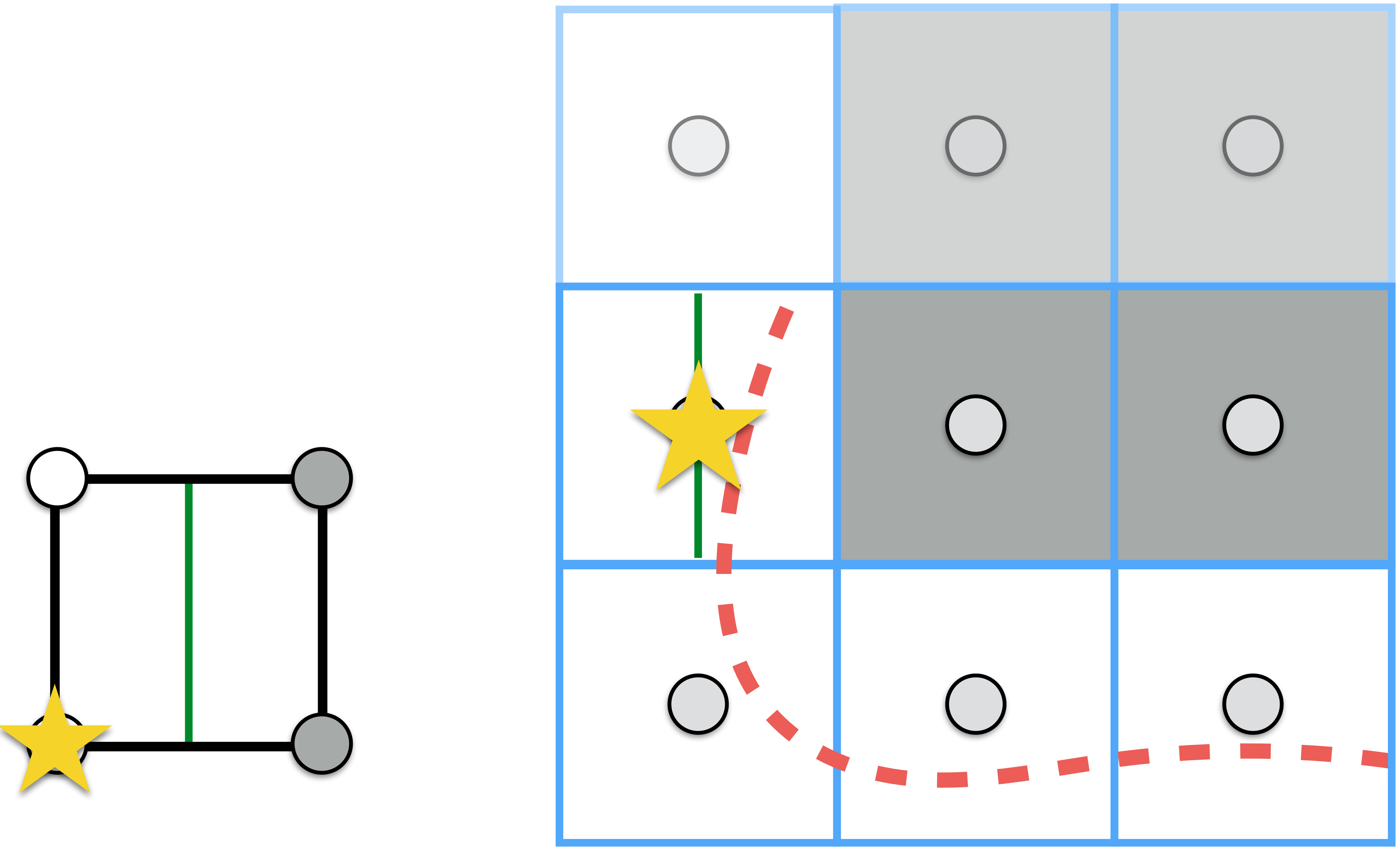
Marching Squares Boundaries Example



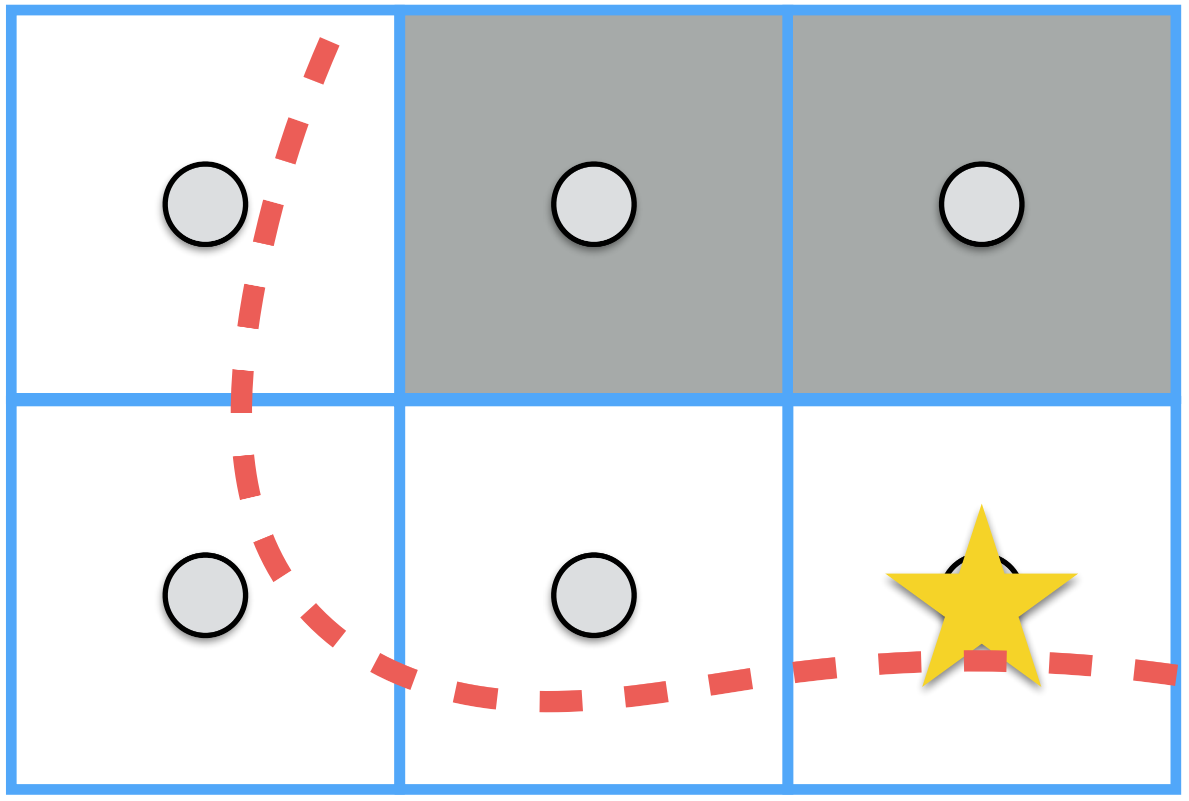
Marching Squares Boundaries Example



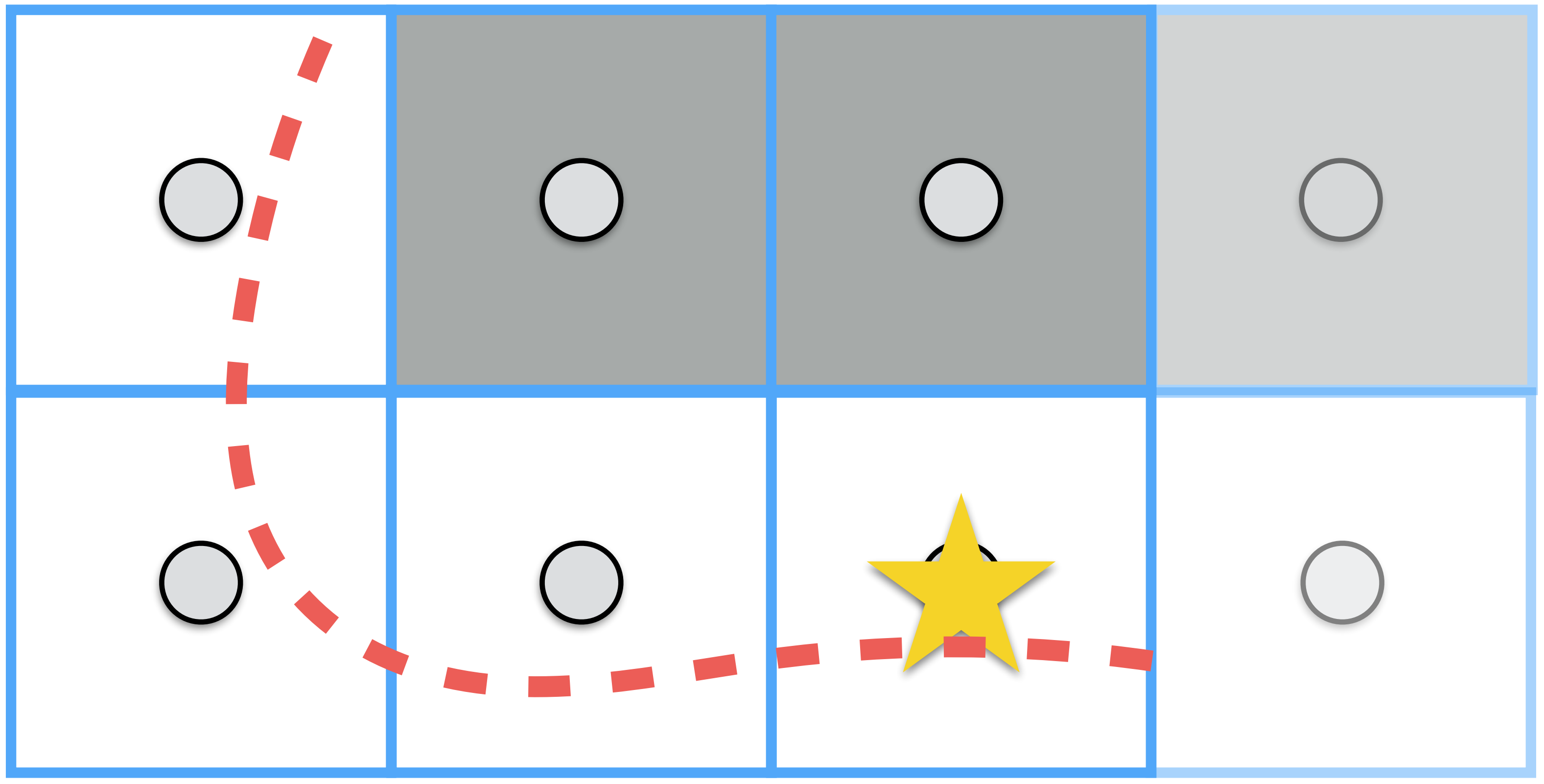
Marching Squares Boundaries Example



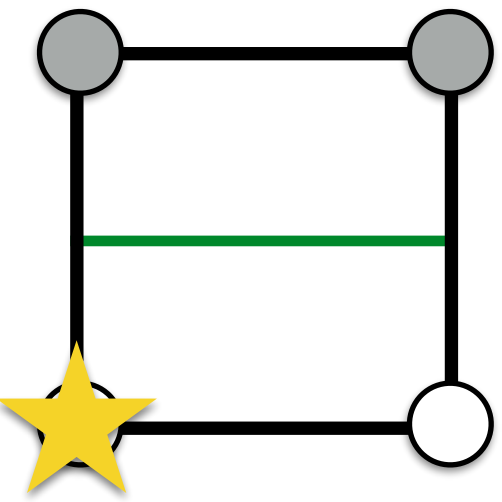
Marching Squares Boundaries Example

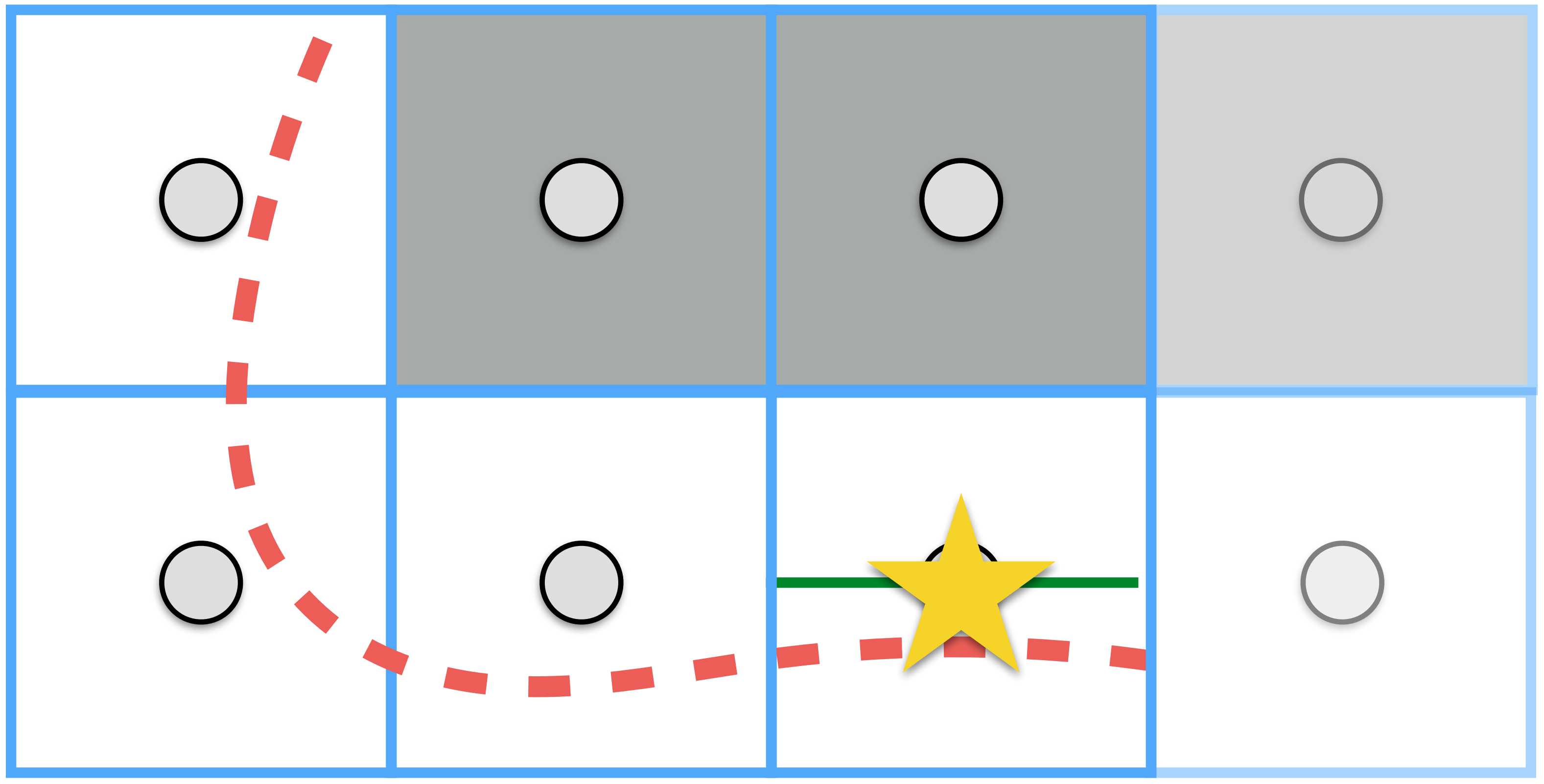


Marching Squares Boundaries Example



Marching Squares Boundaries Example





Marching Squares: Boundaries

- For these cases, we can set different politics:
  - We do not process boundaries, so we cut out part of the information
  - We replicate information from previous scan

Let’s move into the 3D world

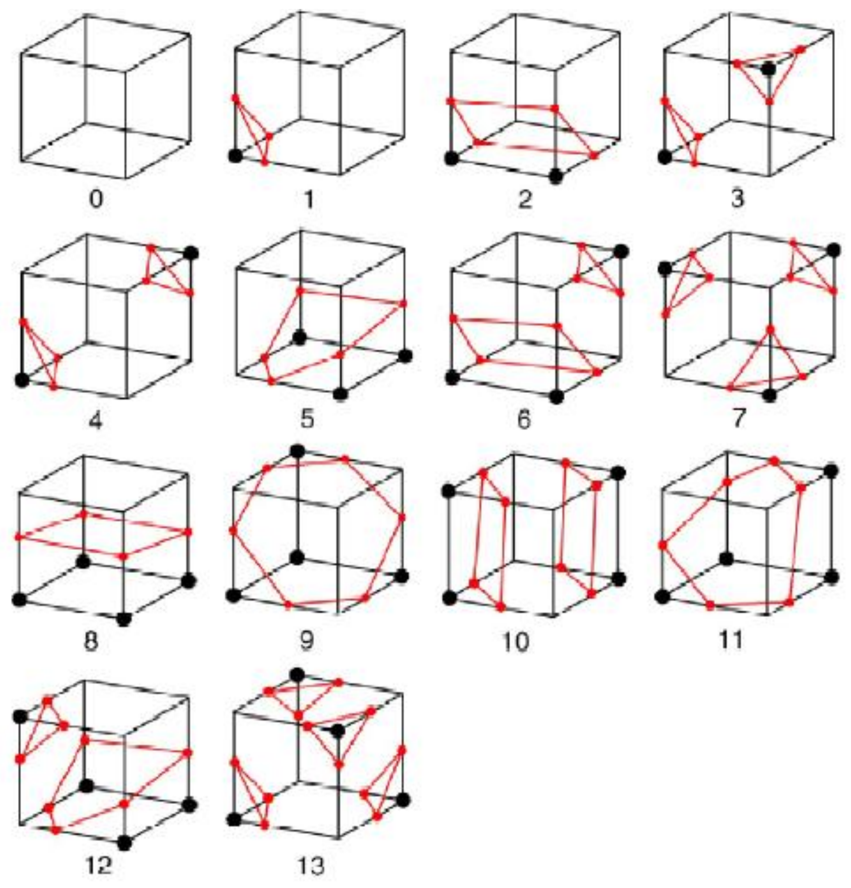
Marching Cubes

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

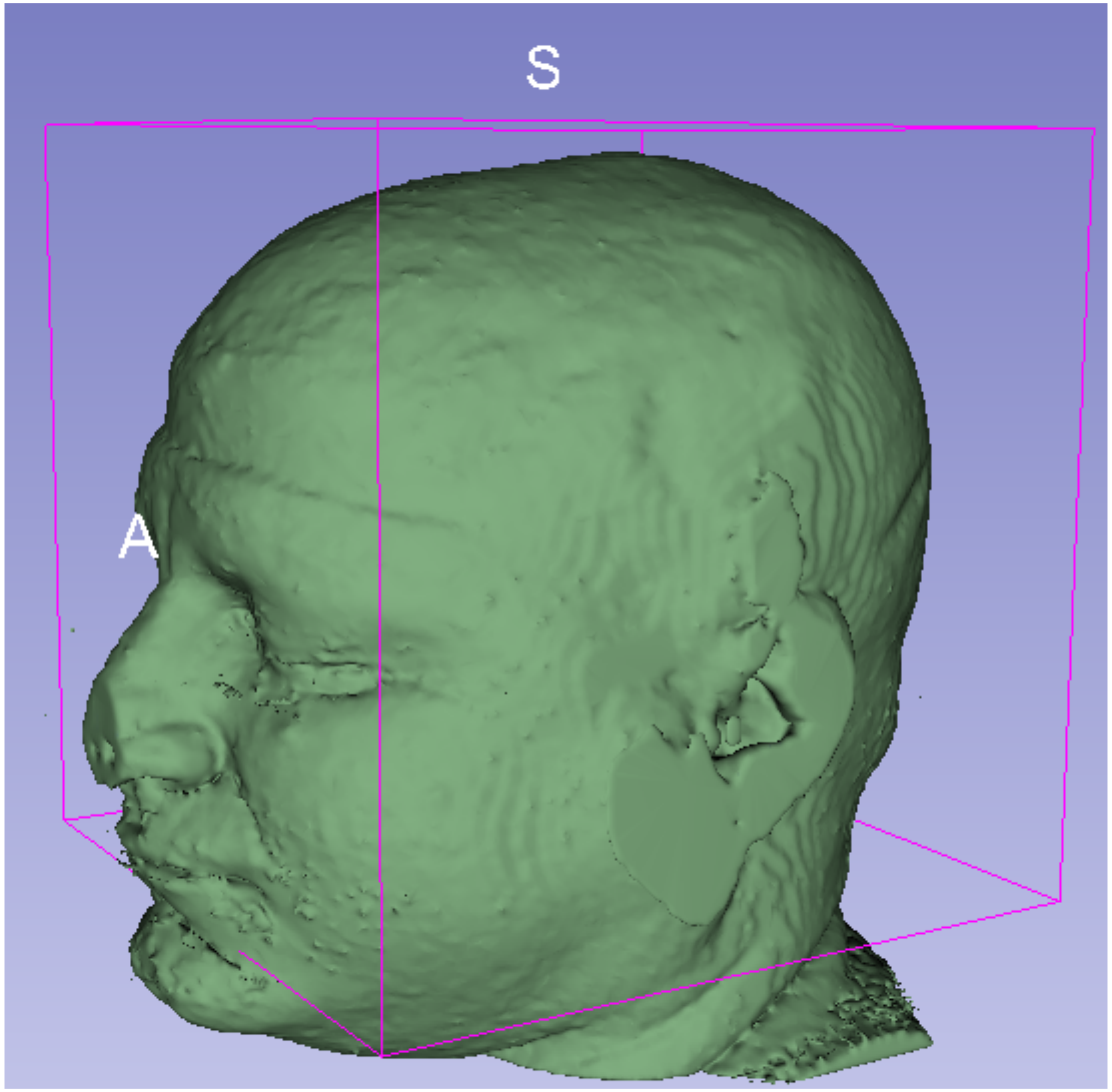
- **2nd pass**: for each voxel, we need to find out the current configuration and to look up into a table to place ***triangles***!

Marching Cubes

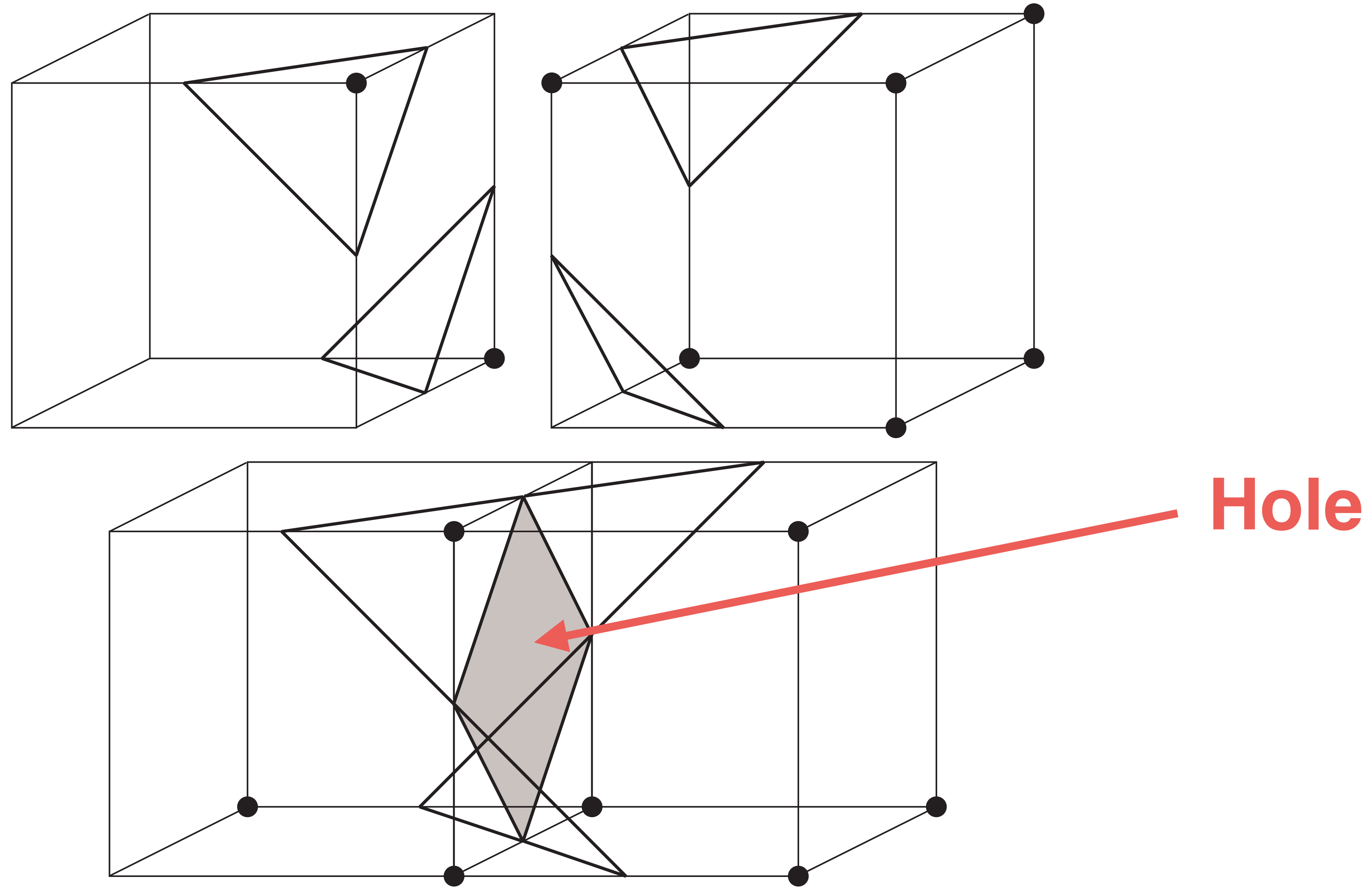
- In 3D the look up table has 256 entries (2^8).
- However, there are only 14 main cases (others are computed by reflecting and/or rotating these):



Marching Cubes



Marching Cubes: Ambiguous Cases

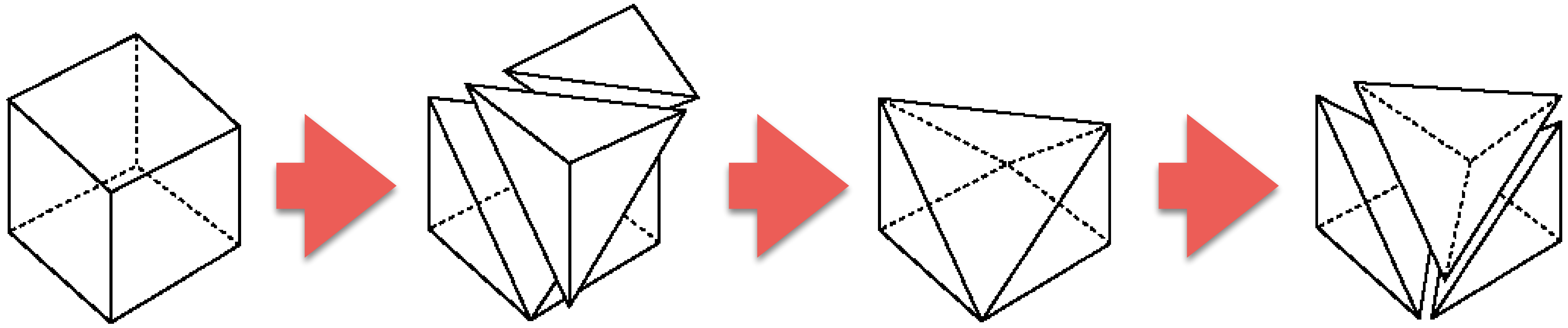


[Cignoni et al. 1999]

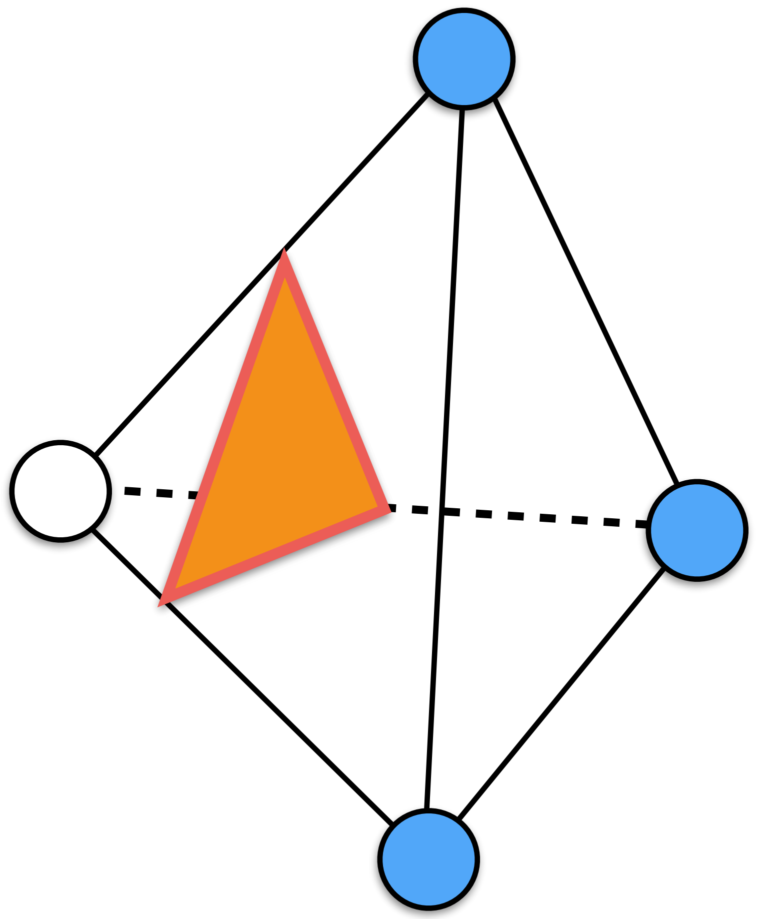
Marching Cubes: Ambiguous Cases

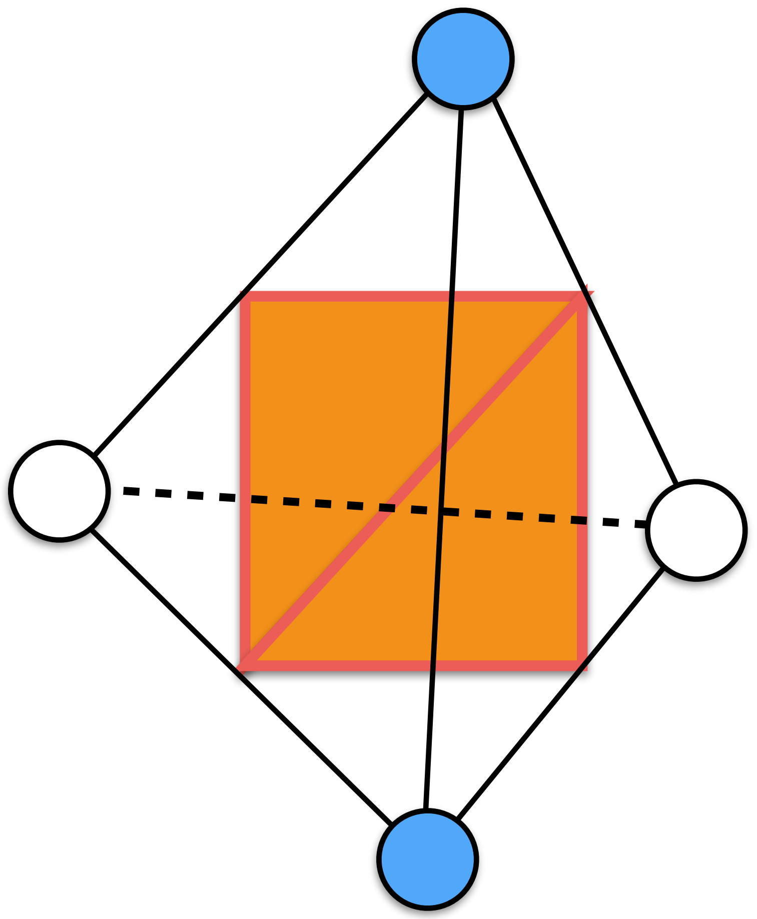
- A solution, which avoids ambiguous cases, is to partition each voxel/cell into tetrahedra; e.g., 5 or 6 of them.

- For each tetrahedra, we compute a configuration based on the segmentation, and then we create triangles according to it.



Marching Cubes:
Examples of Tetrahedra configurations





Marching Cubes: Ambiguous Cases

- Another solution is to extend the table of cubes configuration.
- For each cubes, we have an extra step where we have a table with fixes for certain configurations.

Marching Cubes

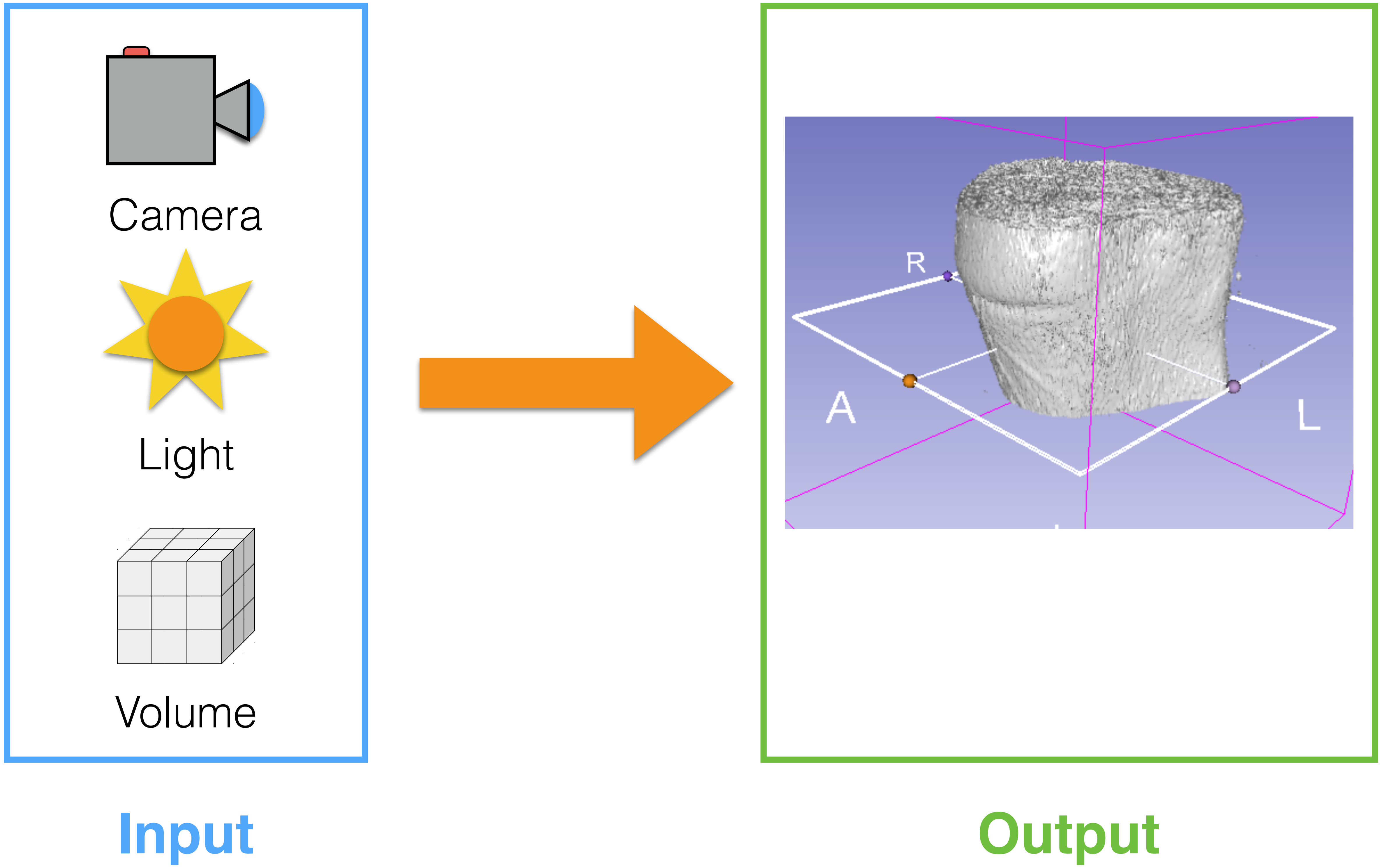
- Advantages:
  - Easy to understand and to implement
  - Fast and non memory consuming
- Disadvantages:
  - Consistency: C_0 and manifold result?
    - Ambiguous cases!
  - Mesh complexity: the number of triangles does not depend on the shape but on the discretization, i.e., number of voxels!
  - Mesh quality: arbitrarily ugly triangles

3D Visualization

Volume Visualization

- We need to pre-visualize the 3D model that we are going to create. This process is called ***rendering***.
- Pre-visualization is:
  - fast: no need to create a 3D model
  - it helps the segmentation process

Volume Visualization



Volume Visualization

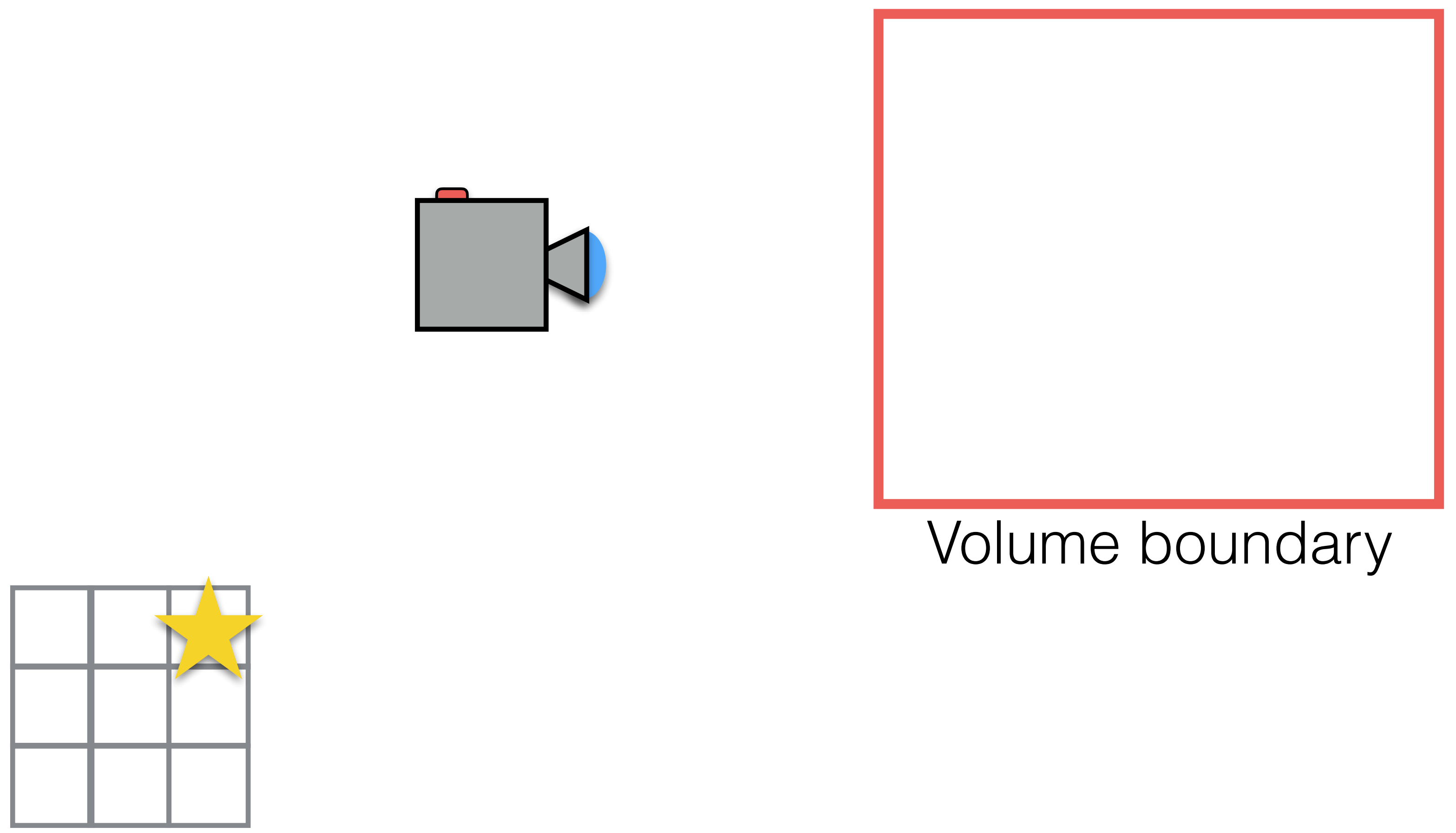
- Given a “virtual camera” and a 3D volume (e.g., from a CAT or MRI), we want to generate an image, i.e., called ***rendered image***.
- What do we need?
    - A virtual camera
    - A virtual light source
    - How to mix voxels’ colors

Rendering

- We need to color pixels (in the image plane) using the volume information; i.e., intensity values.
- For each pixel, we create a ray (i.e., a line):
    - If the ray intersects the volume, then we collect intensity values from it; i.e. we integrate it!
    - Otherwise the pixel will be set to zero or fully transparent!

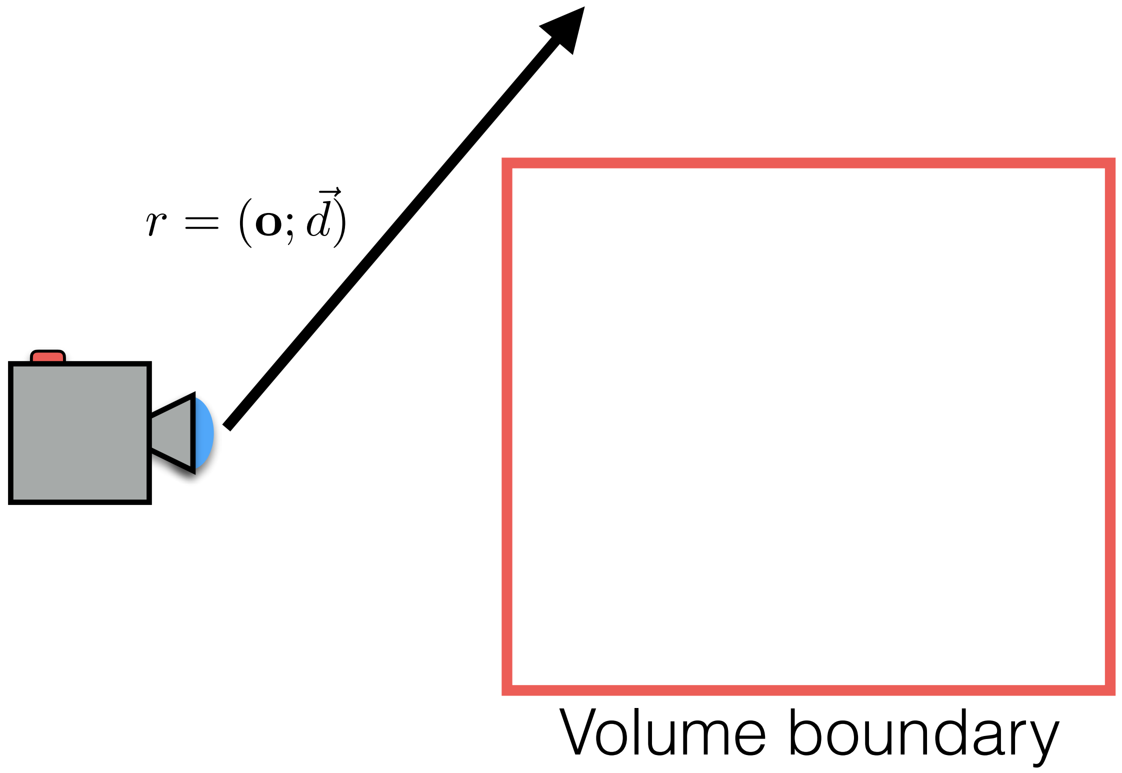
Volume Rendering: Ray-Marching

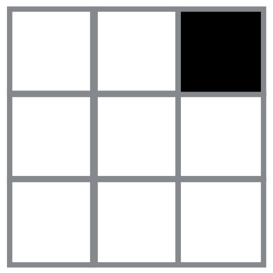
- Let's start our rendering at a given pixel (see the star):



Volume Rendering: Ray-Marching

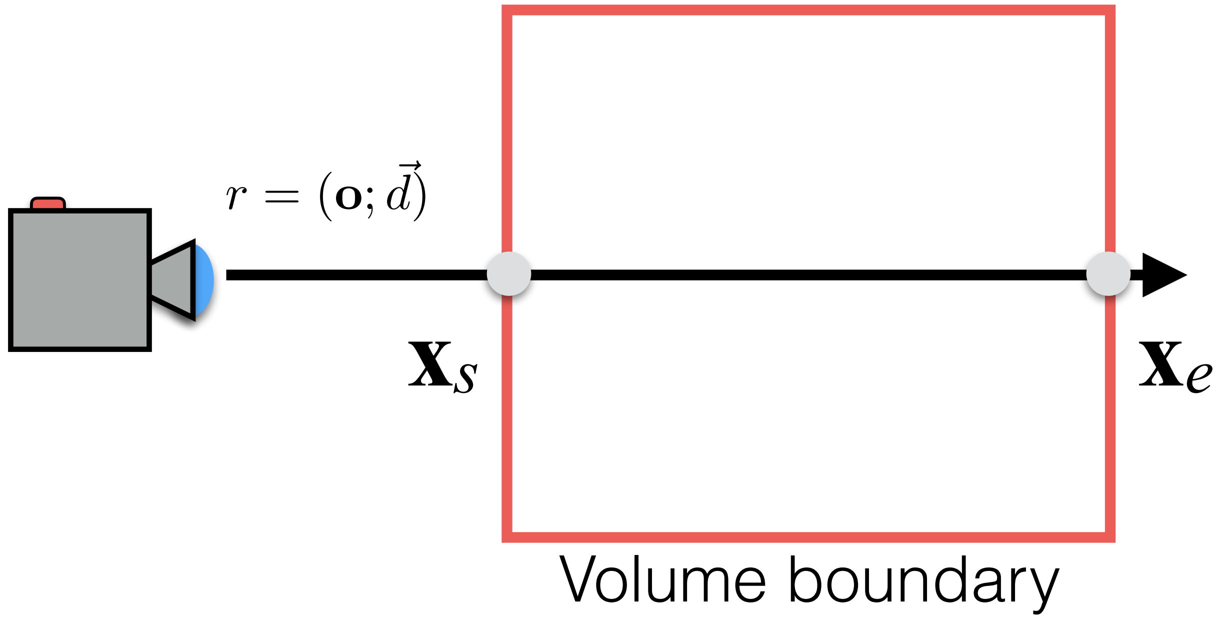
- If the ray misses the volume:

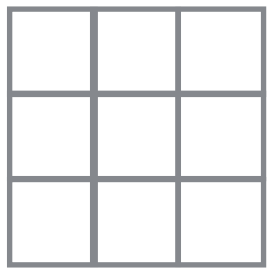




Volume Rendering: Ray-Marching

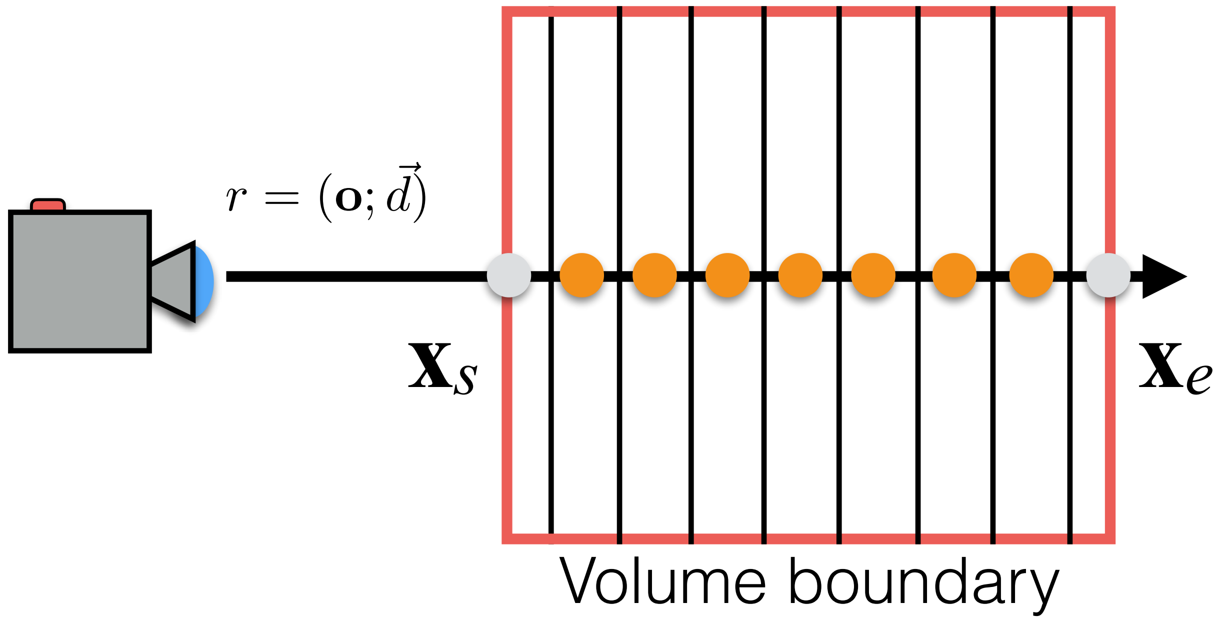
- If the ray hits the volume:



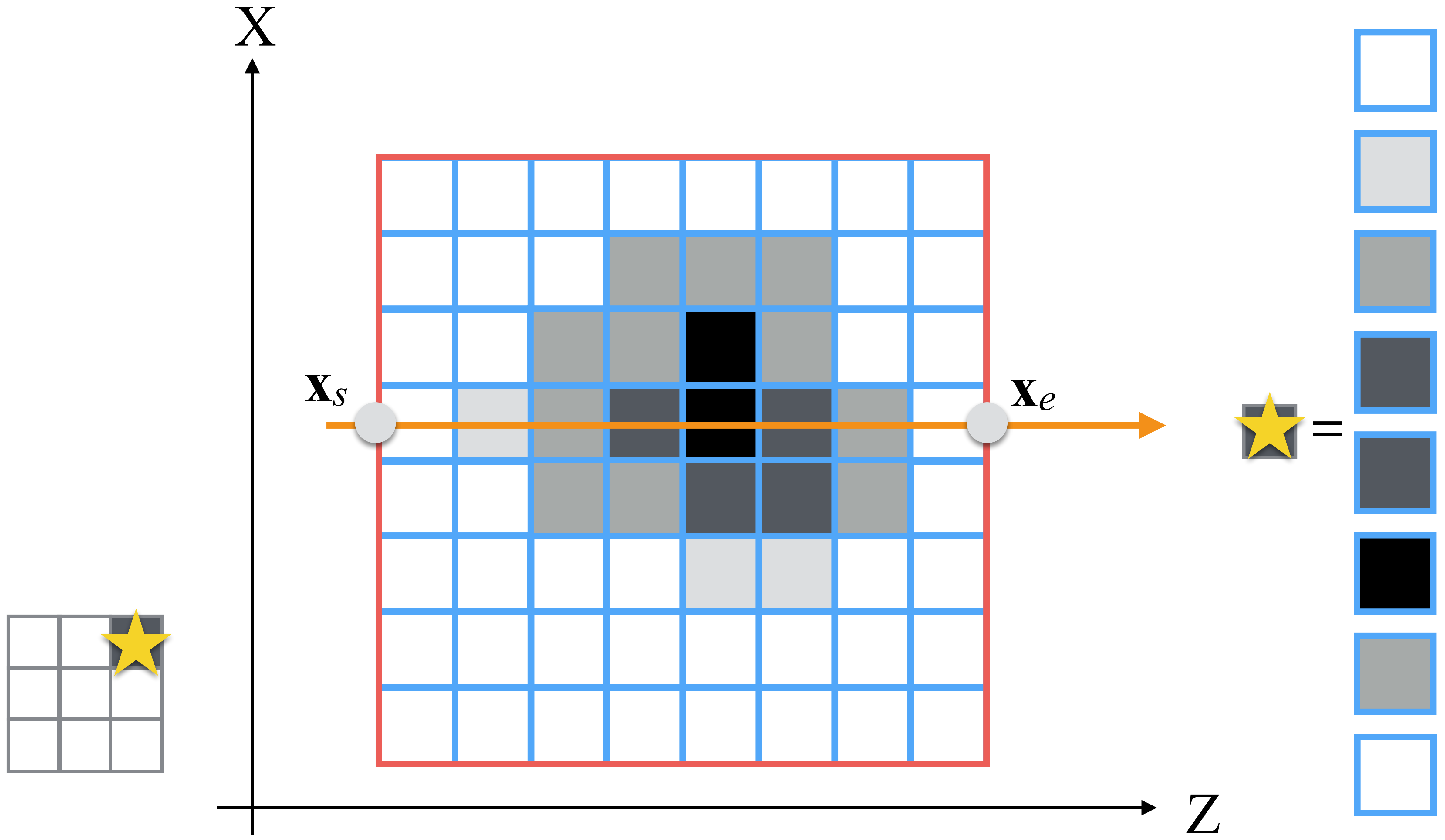


Volume Rendering: Ray-Marching

- Then, we integrate inside it with a step equal to the resolution of the volume:



Volume Rendering: Ray-Marching



Volume Rendering: Ray-Marching

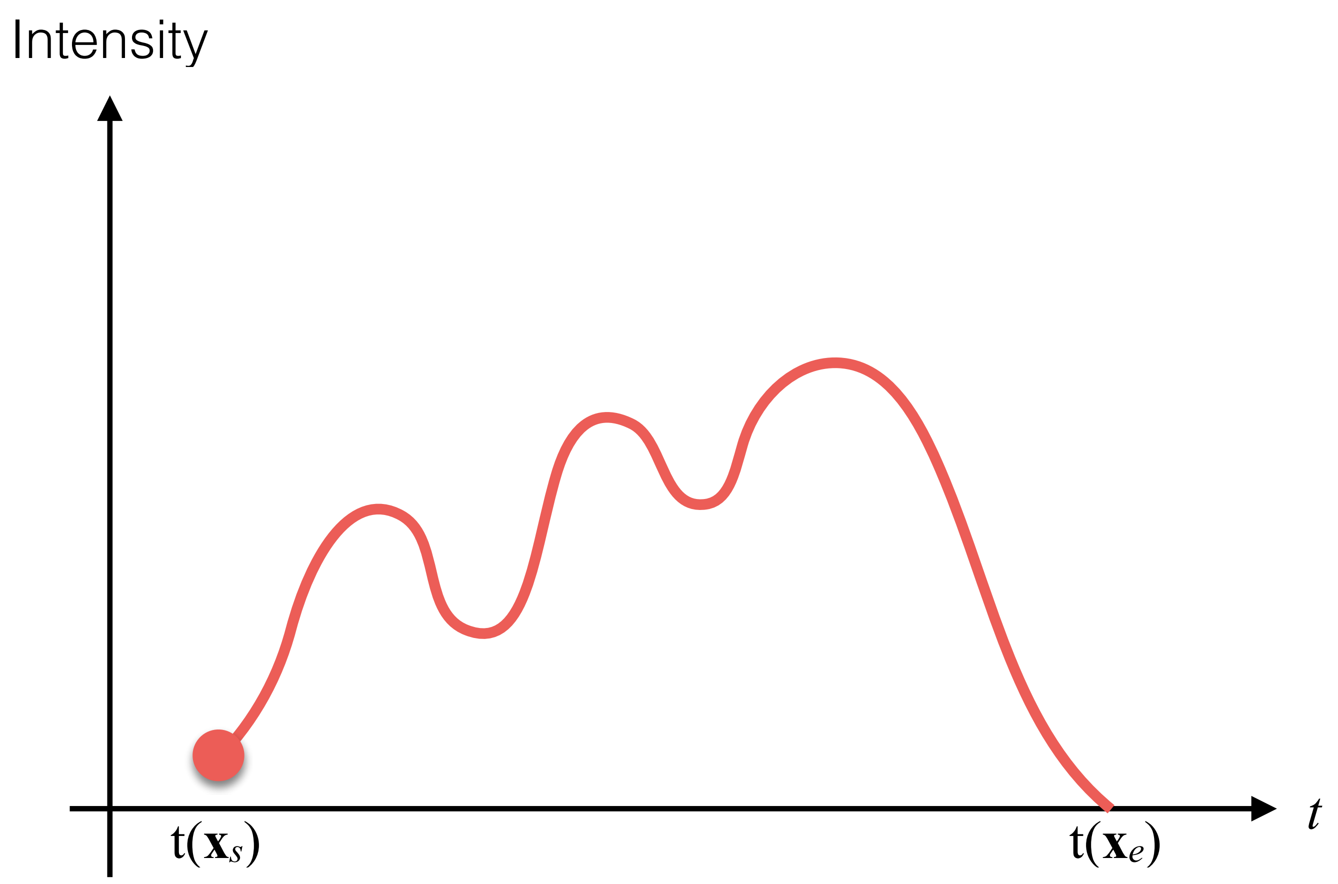
- In other words, we define a ***rendering equation*** as:

$$I[u, v] = \int_{t(\mathbf{x}_s)}^{t(\mathbf{x}_e)} T\Bigg(V\left[\mathbf{o} + \vec{d}[u, v] \cdot t\right] \Bigg) dt$$

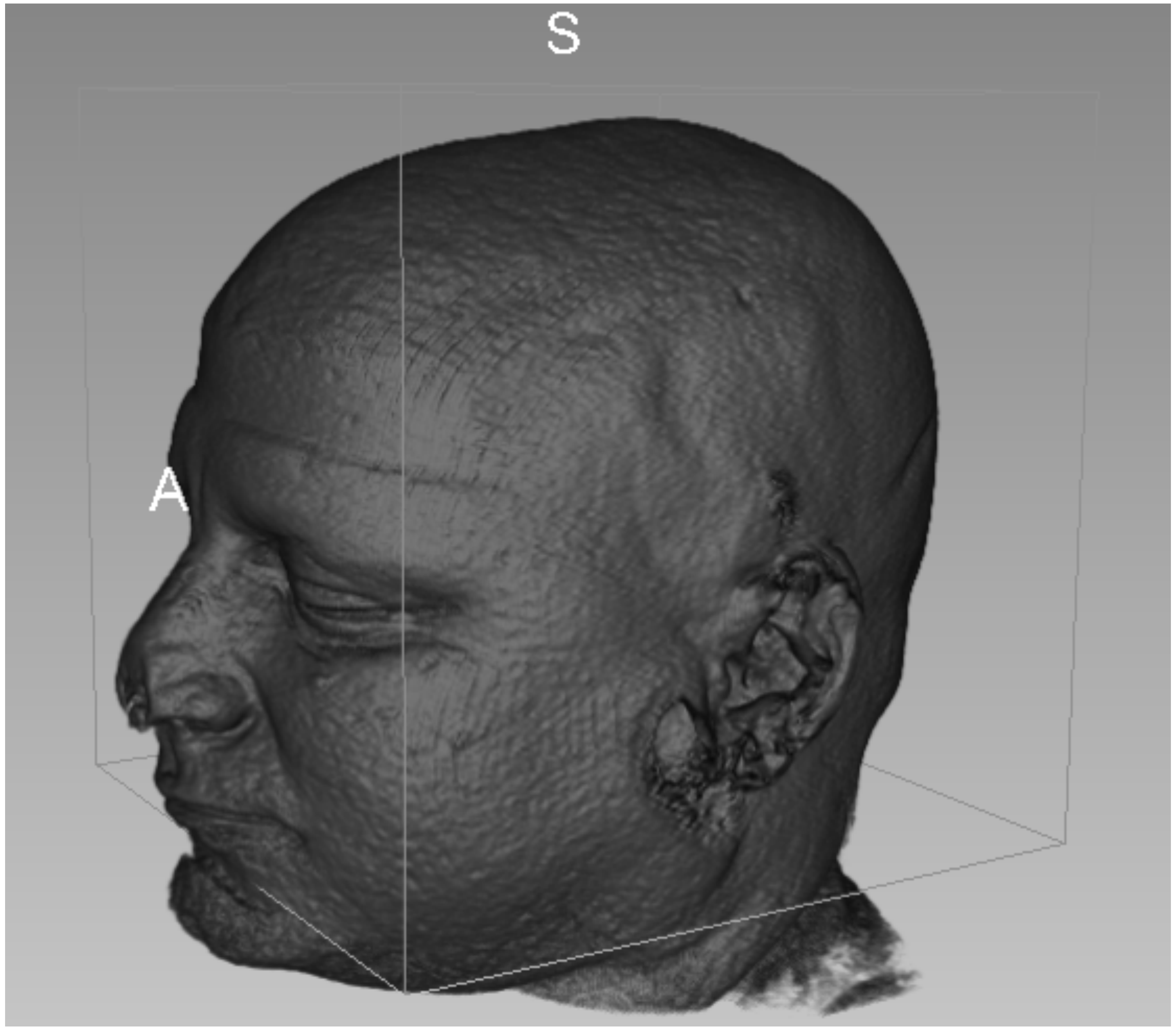
T is called the ***transfer function***
to highlights volume features.

Volume Rendering: Ray-Marching

- To determine the outside surface, we stop the integration at the first non zero value (over a threshold):

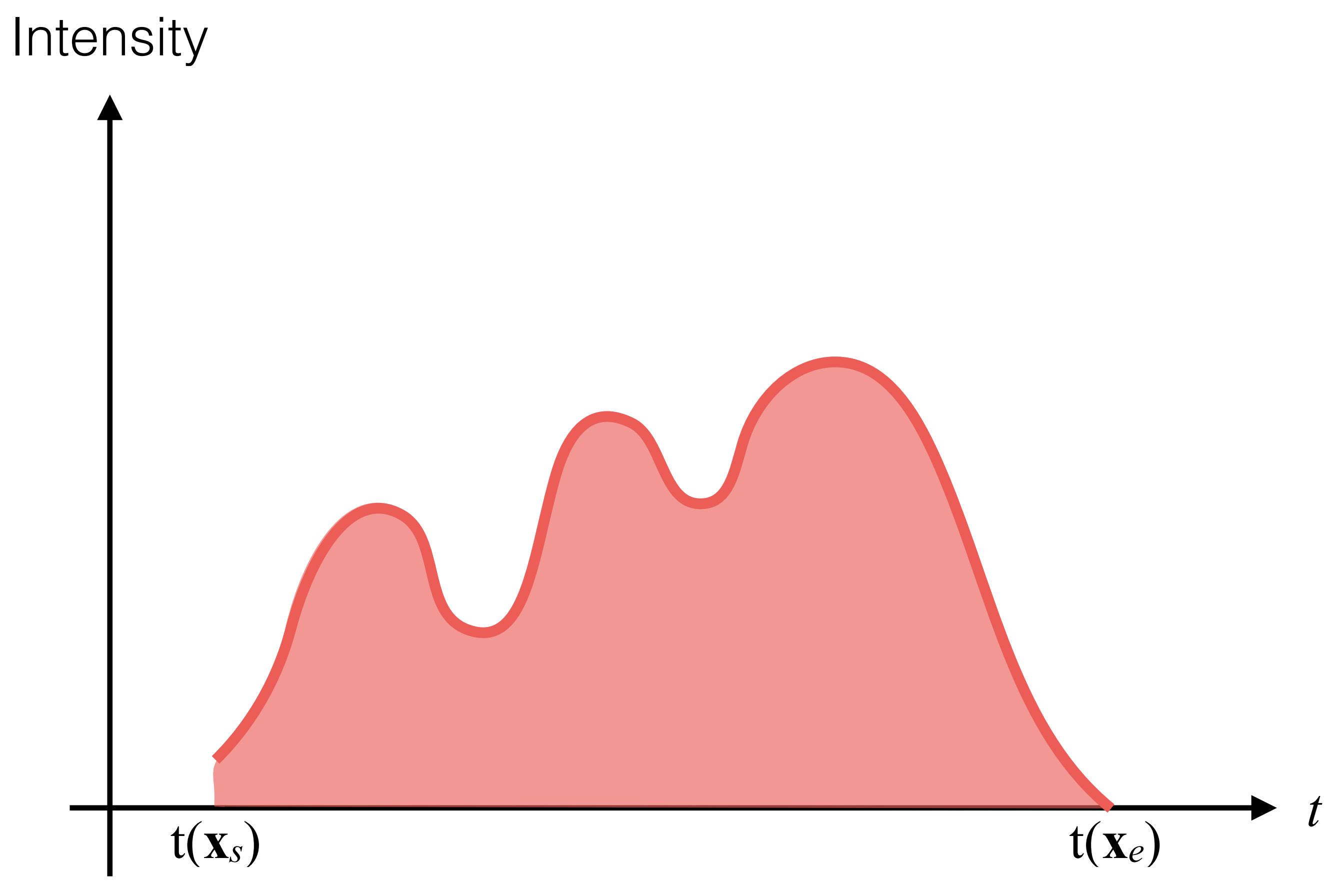


Volume Rendering: Ray-Marching Example

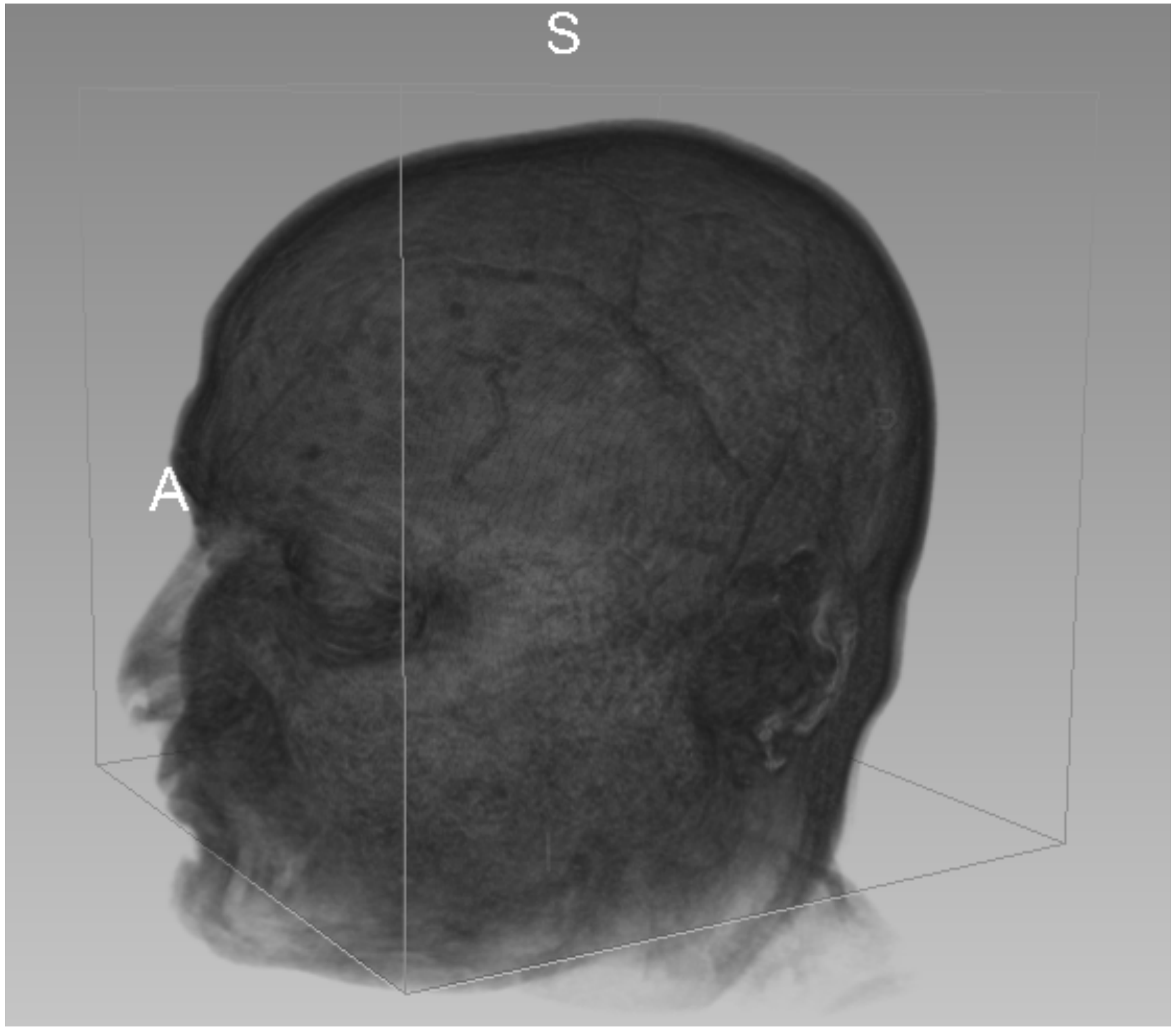


Volume Rendering: Ray-Marching

- To see all features inside the volume, we integrate along the ray:

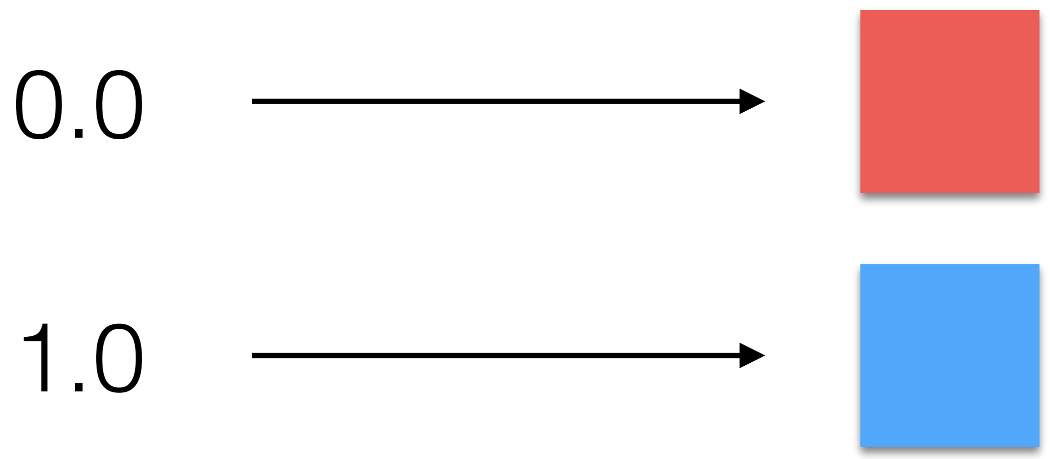


Volume Rendering: Ray-Marching Example

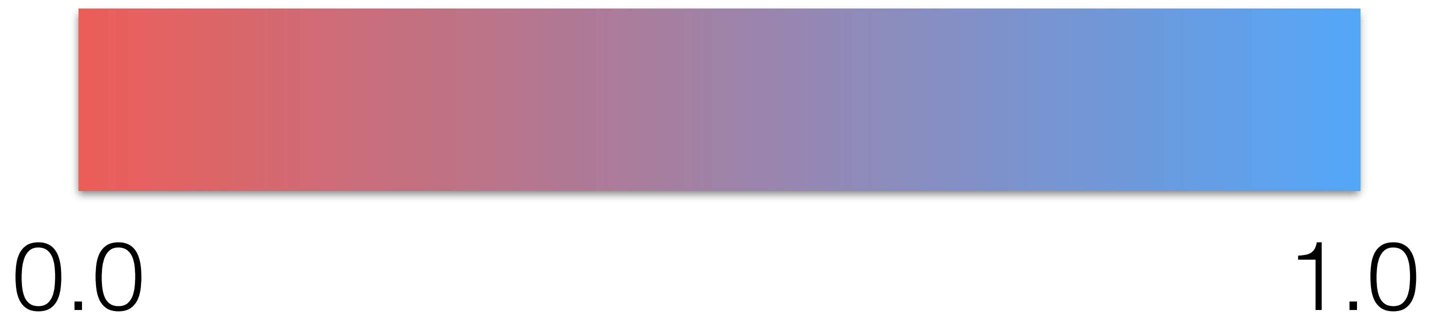


Volume Rendering: Color Mapping

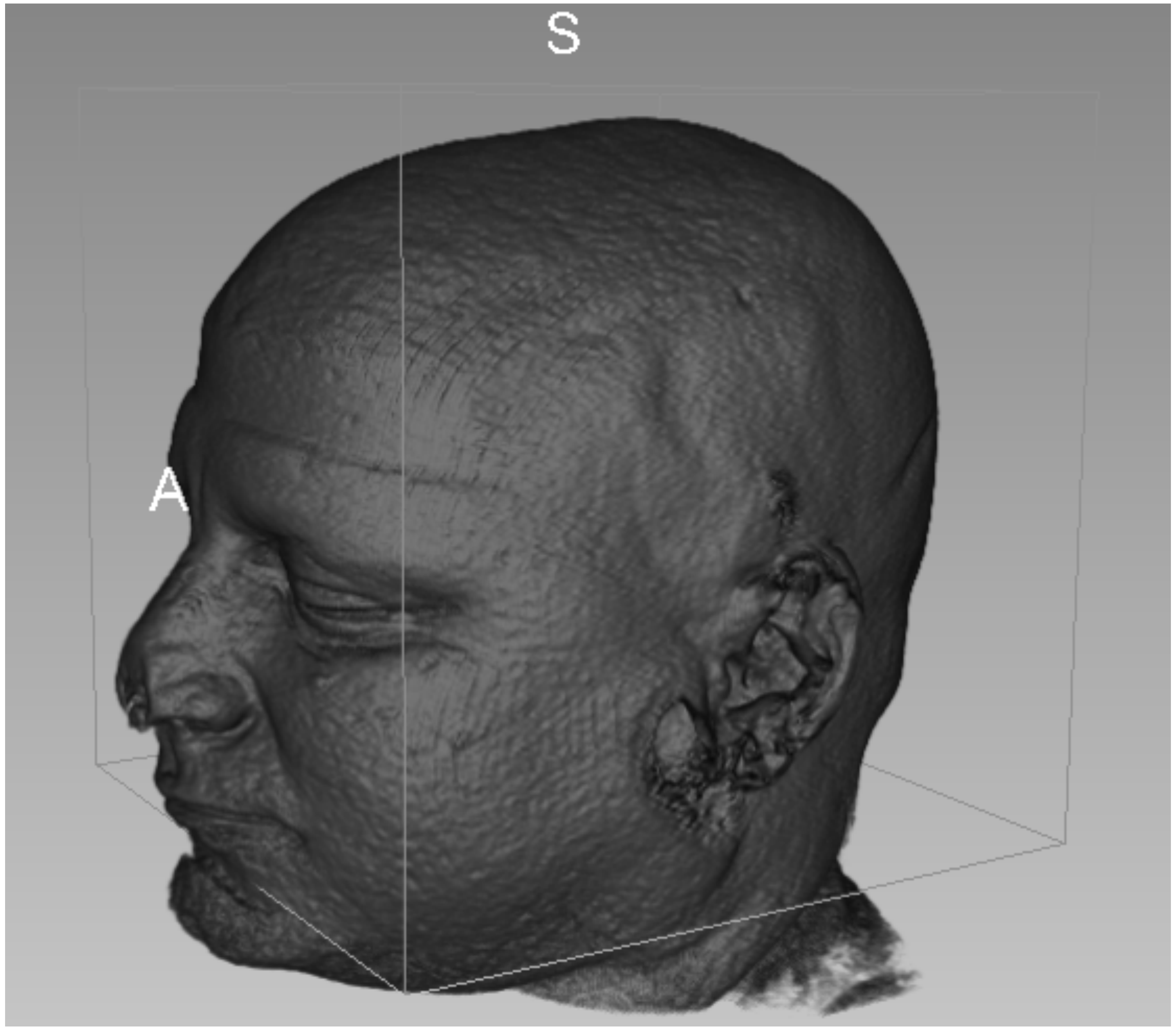
- To improve visualization intensity values are mapped to colors:



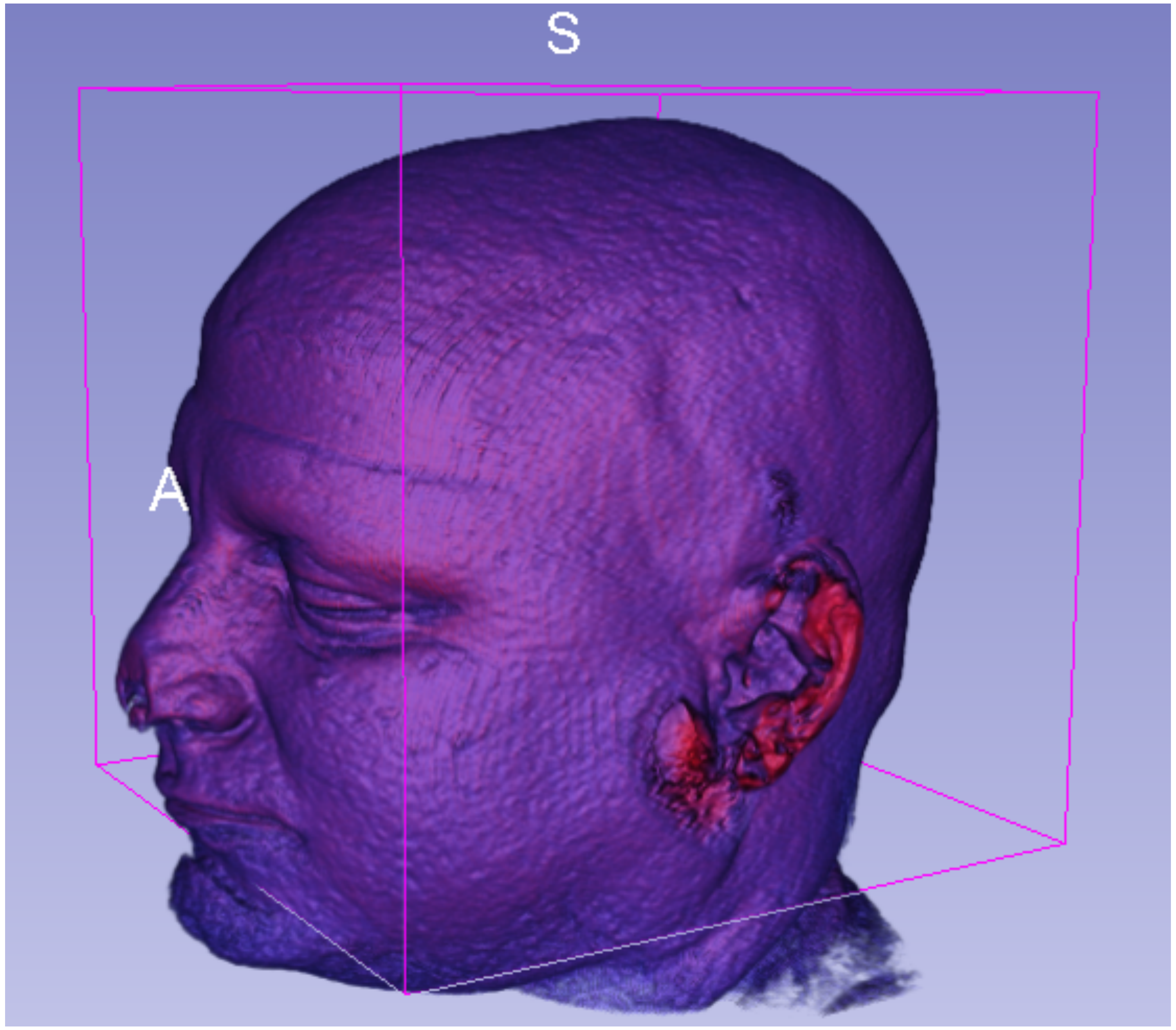
- In between values are linearly interpolated:



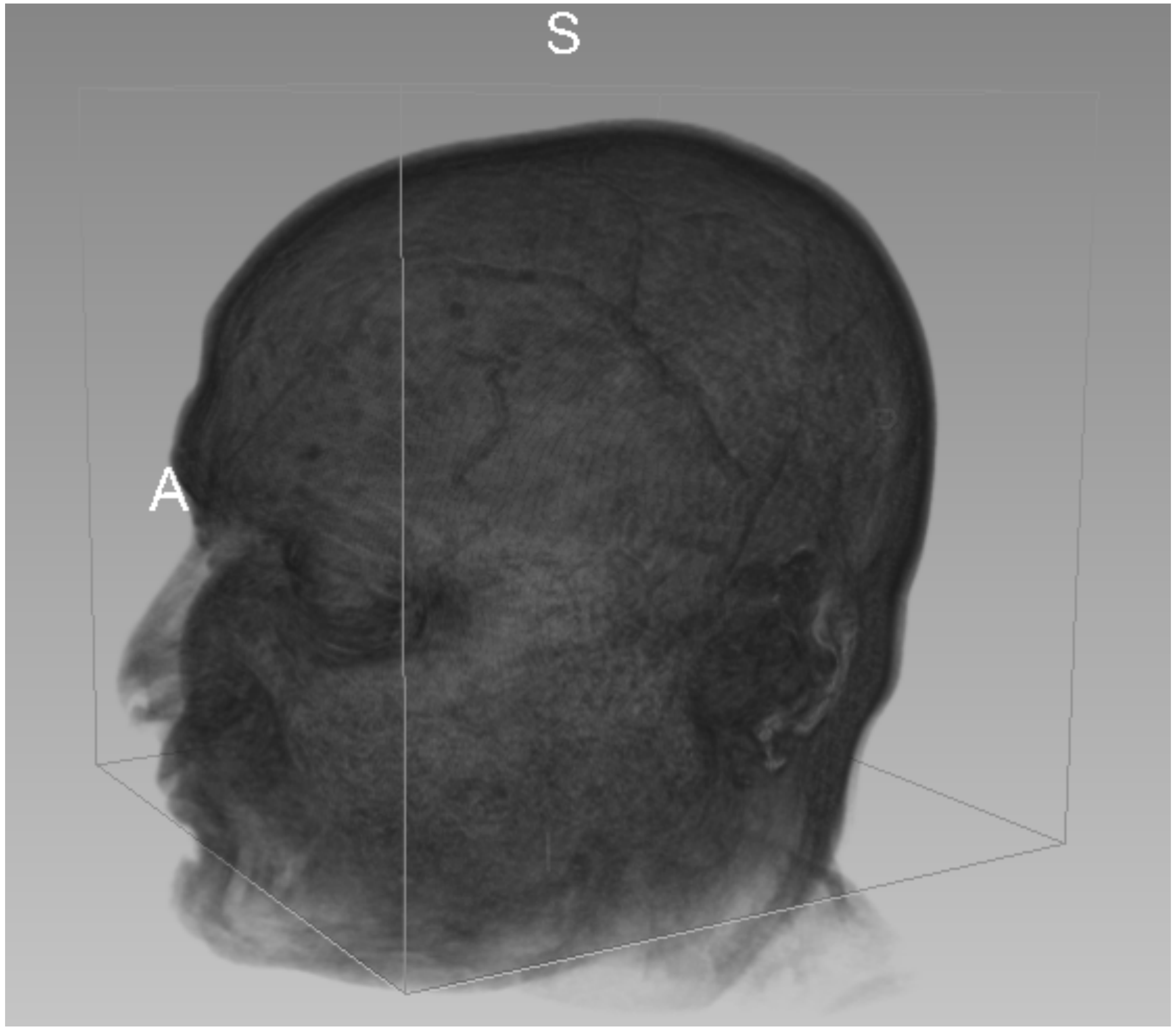
Volume Rendering: Color Mapping



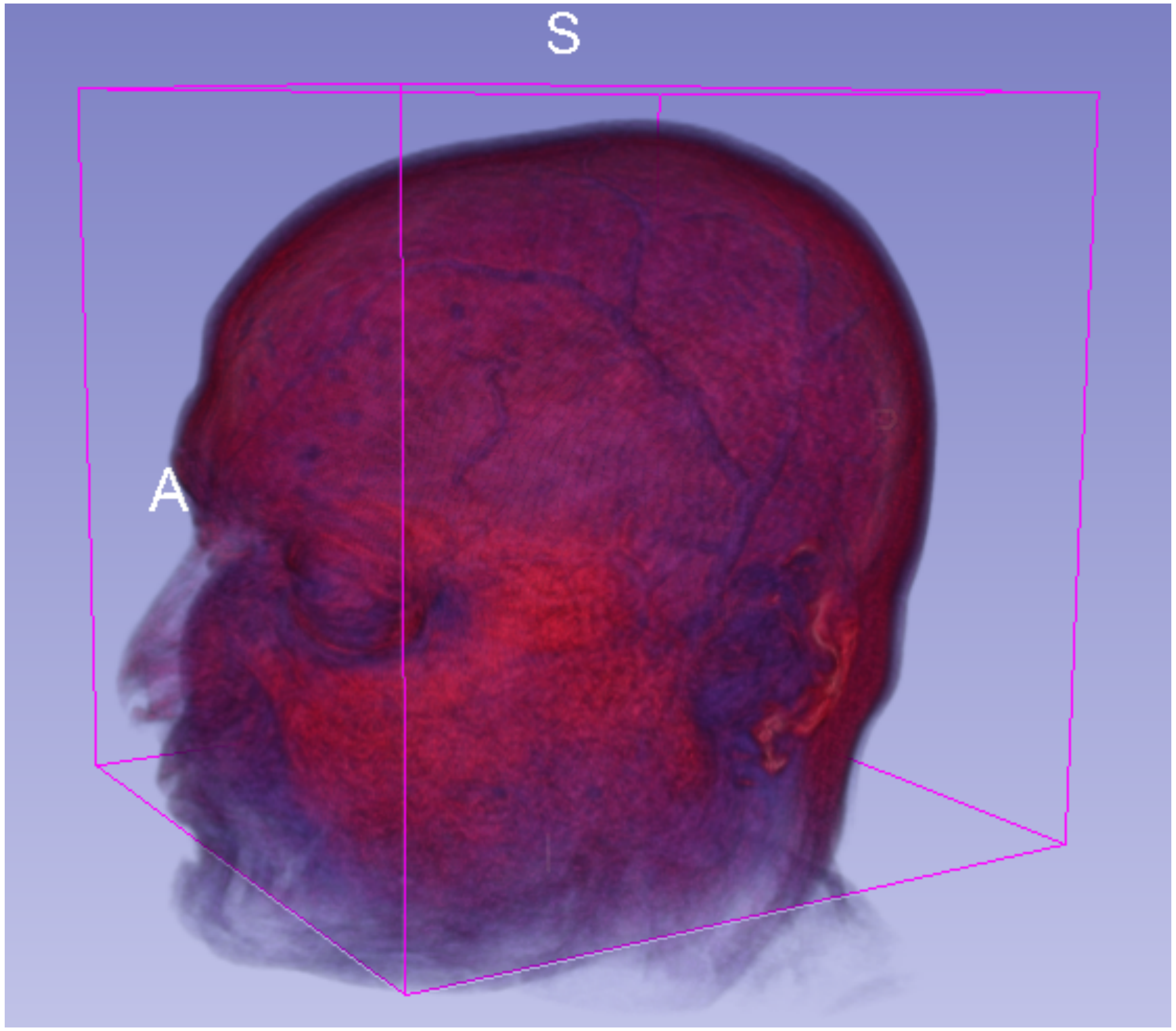
Volume Rendering: Color Mapping



Volume Rendering: Color Mapping



Volume Rendering: Color Mapping



Volume Rendering: Let There Be Light

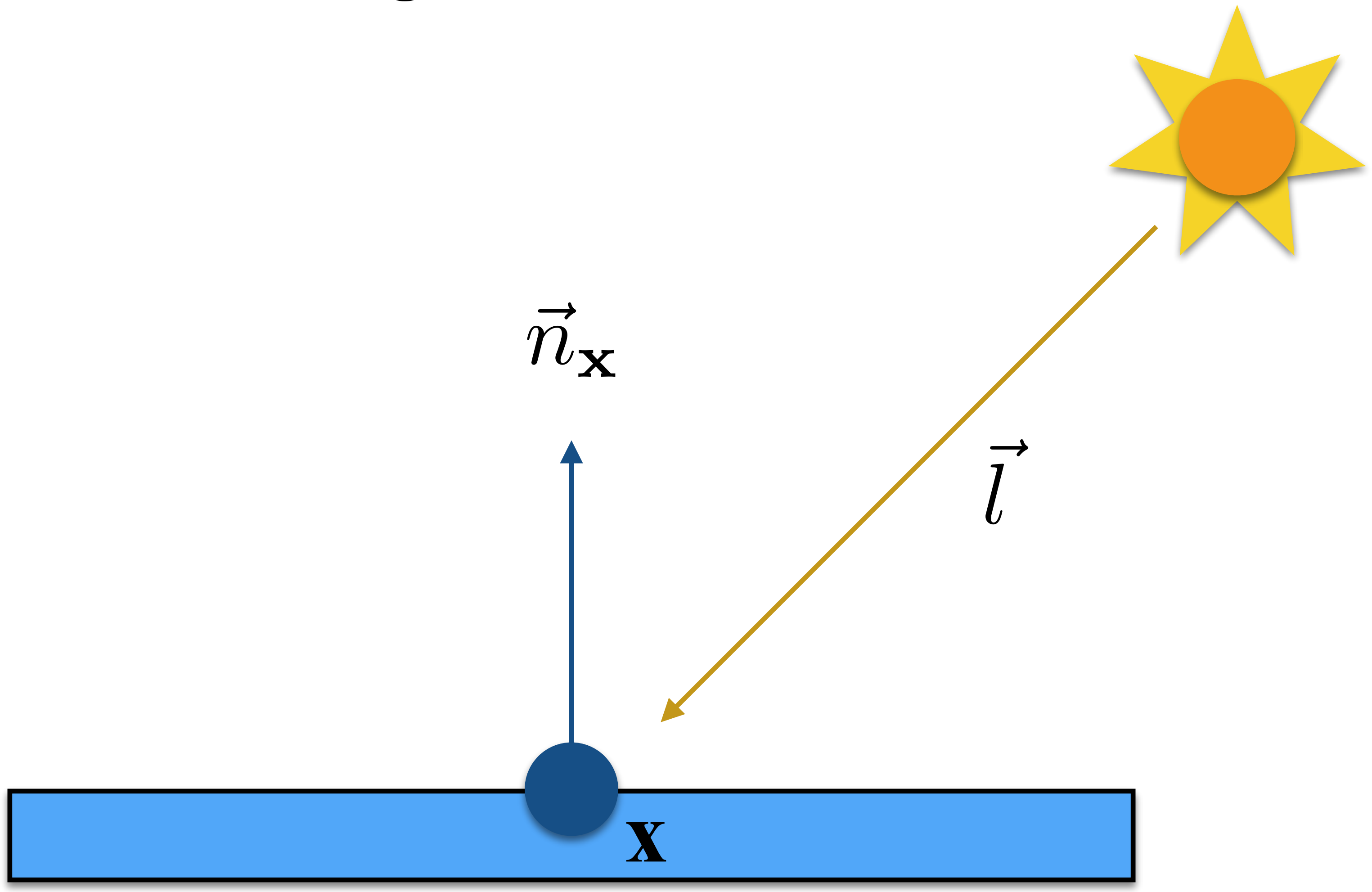
- We need to light each voxel by a light source.
- There are local (taking into account that light bounces around) and global models.
- For the sake of simplicity, we are interested in local models only!

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.

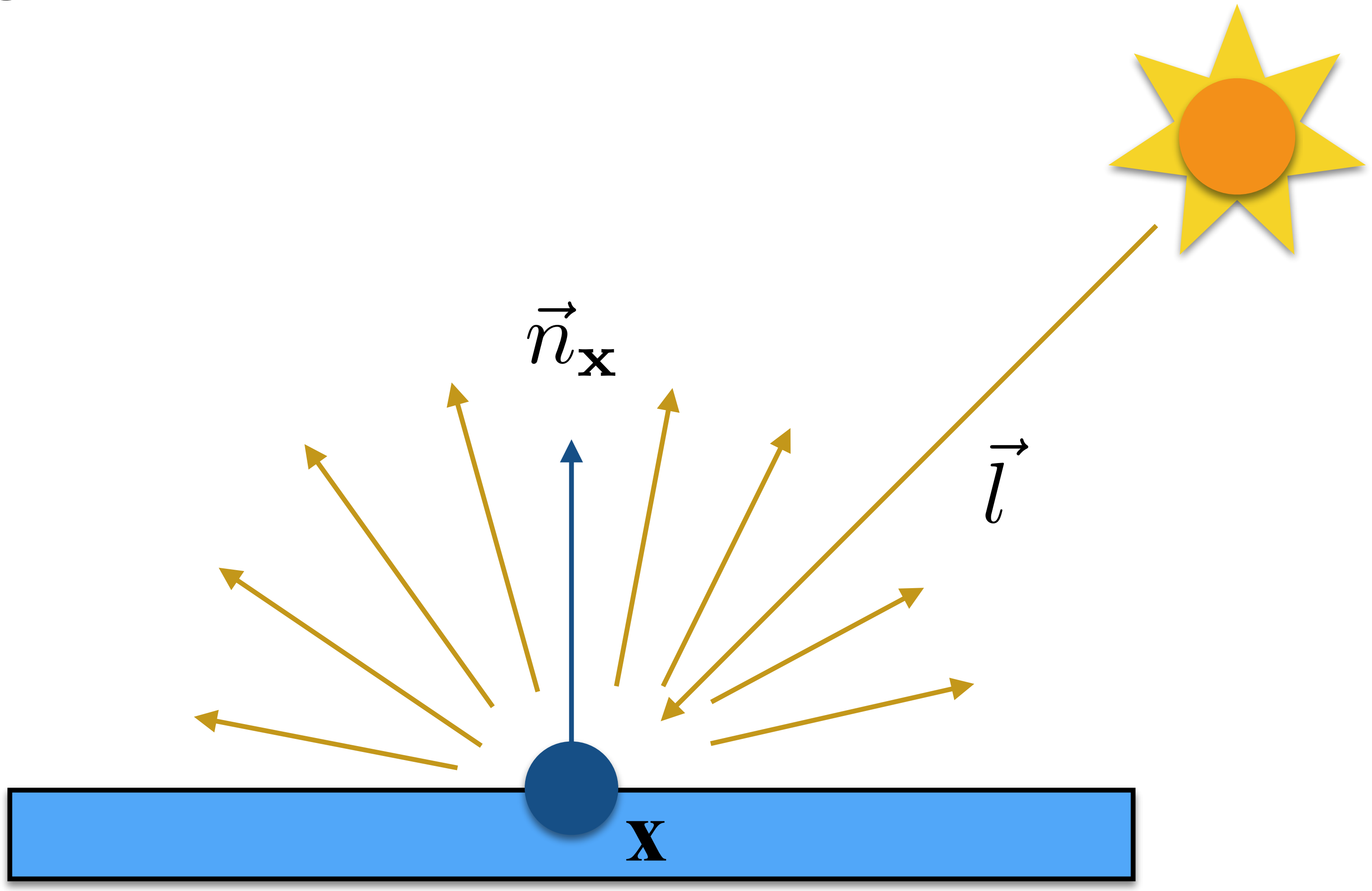
Volume Rendering: Let there be light

- 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:
  - $\vec{n}_{\mathbf{x}}$ is normalized.
  - $\vec{l}$ is normalized.

$$\vec{n}_{\mathbf{x}} = -\frac{\vec{\nabla} V(\mathbf{x})}{\|\vec{\nabla} V(\mathbf{x})\|}$$

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:
  - $\vec{n}_{\mathbf{x}}$ is normalized.
  - $\vec{l}$ is normalized.

$$\vec{n}_{\mathbf{x}} = -\frac{\vec{\nabla} V(\mathbf{x})}{\|\vec{\nabla} V(\mathbf{x})\|}$$

Volume Rendering: Let there be light

- The model is defined as

Radiance Albedo/Intensity

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

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

$$\Rightarrow \quad \vec{n}_{\mathbf{x}} = -\frac{\vec{\nabla} V(\mathbf{x})}{\|\vec{\nabla} V(\mathbf{x})\|}$$

Volume Rendering: Let there be light

- In our case, this model is slightly modified into:

$$L(\mathbf{x}) = \frac{V(\mathbf{x})}{\pi} \max\left(-\vec{n_{\mathbf{x}}} \cdot \vec{l}, 0\right)$$

- Note that:
  - $\vec{n}_{\mathbf{x}}$ is normalized
  - $\vec{l}$ is normalized.
  - $V(\mathbf{x})$ is the volume intensity or color coded intensity at position $\mathbf{x}$.

Volume Rendering: Let there be light

- How does this affect the rendering equation?

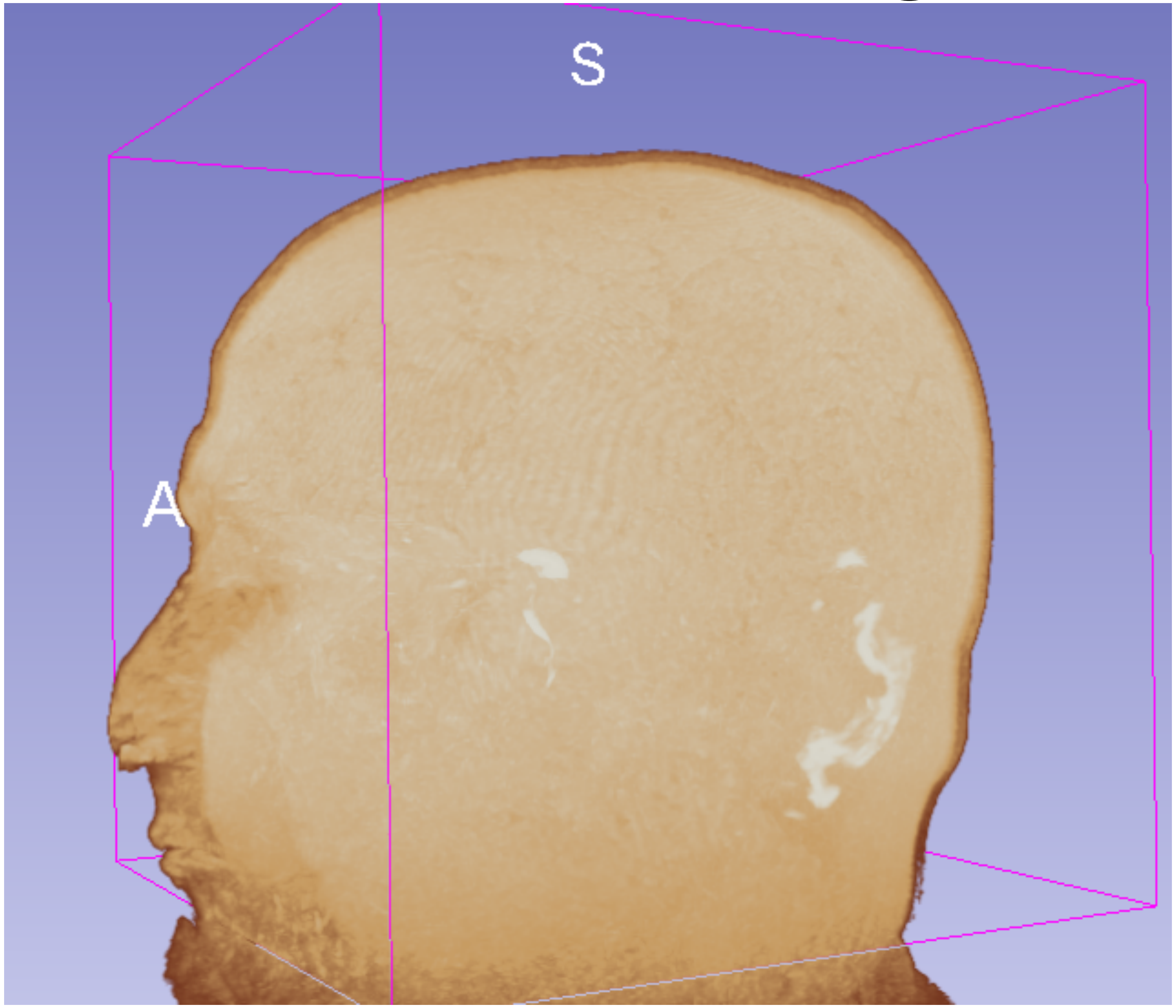
- It changes from:

$$I[u,v] = \int_{t(\mathbf{x}_s)}^{t(\mathbf{x}_e)} T\Big(V\left[\mathbf{o} + \vec{d}[u,v] \cdot t\right]\Big) dt$$

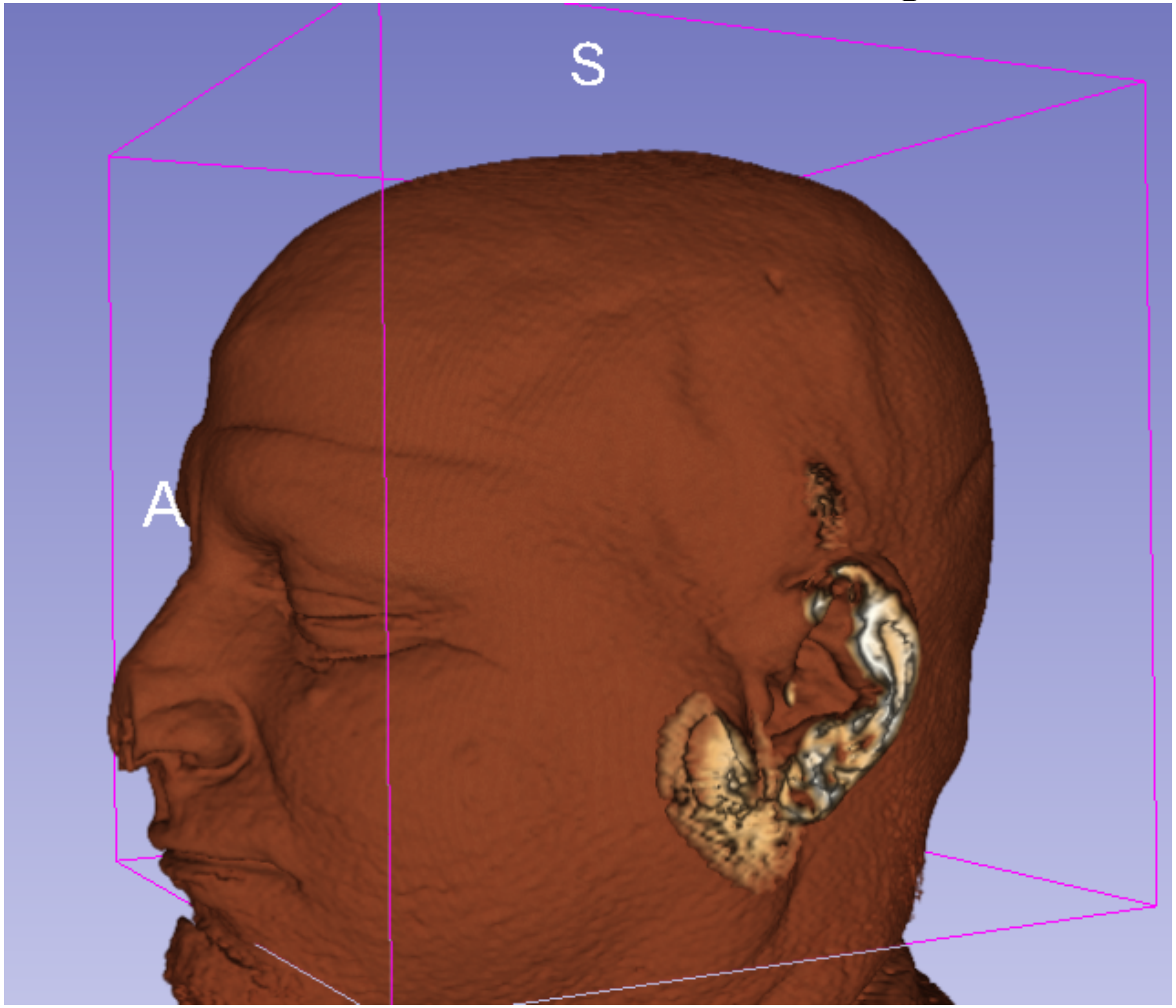
- to:

$$I[u,v] = \int_{t(\mathbf{x}_s)}^{t(\mathbf{x}_e)} T\Big(L(\mathbf{o} + \vec{d}[u,v] \cdot t)\Big) dt$$

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!

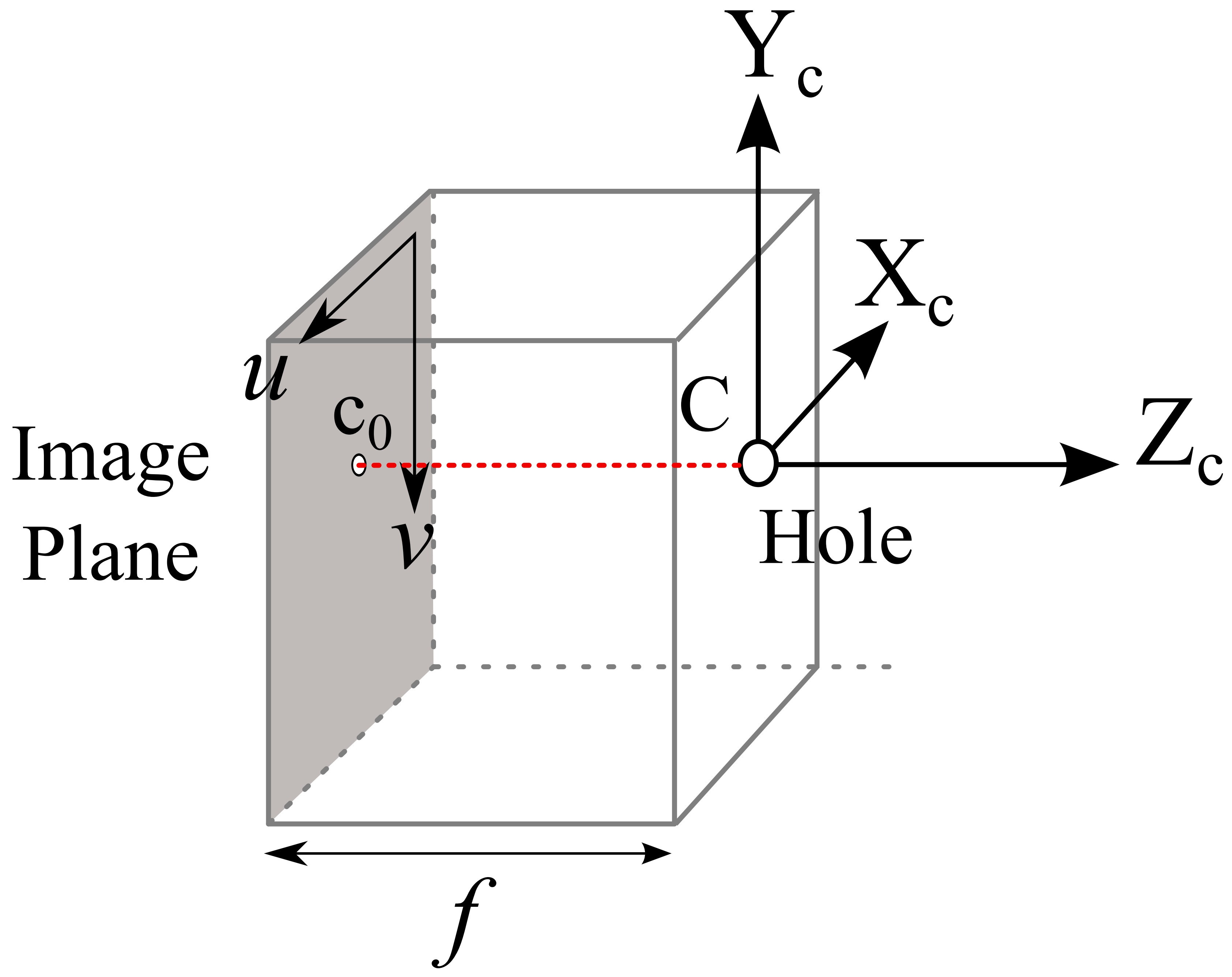
that’s all folks!

Acknowledgements

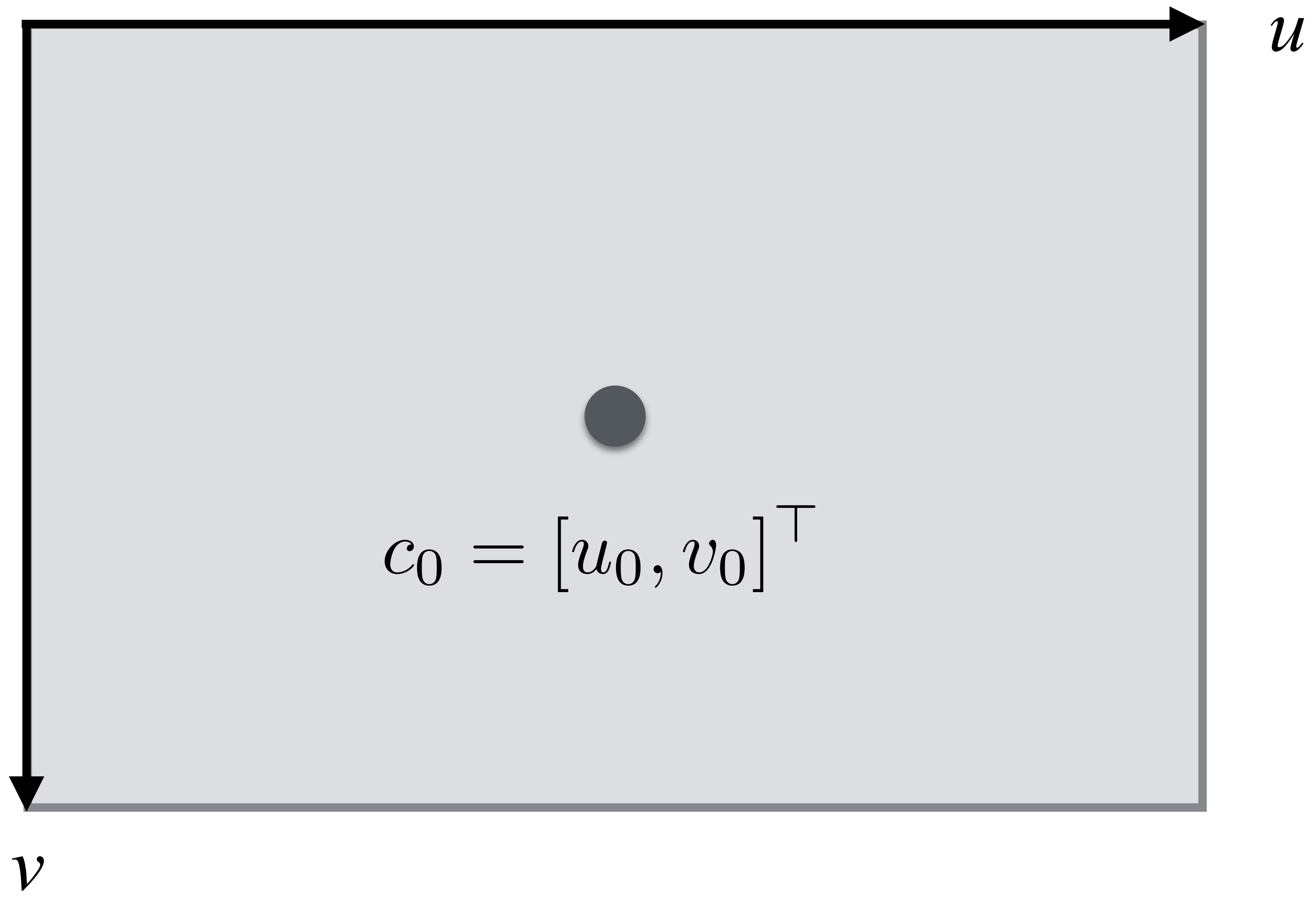
- Some images on work by:
  - Dr. Fabio Ganovelli:
    - http://vcg.isti.cnr.it/~ganovell/
  - Dr. Paolo Cignoni:
    - http://vcg.isti.cnr.it/~cignoni/

Appendix A: The Pin-hole Camera Model

Camera Model: Pinhole Camera



Camera Model: Image Plane



- Pixels are not square: height and width; i.e., (k_u, k_v).
- c_0 is the projection of C (the optical center) and its is called the principal point.

Camera Model: Pinhole Camera

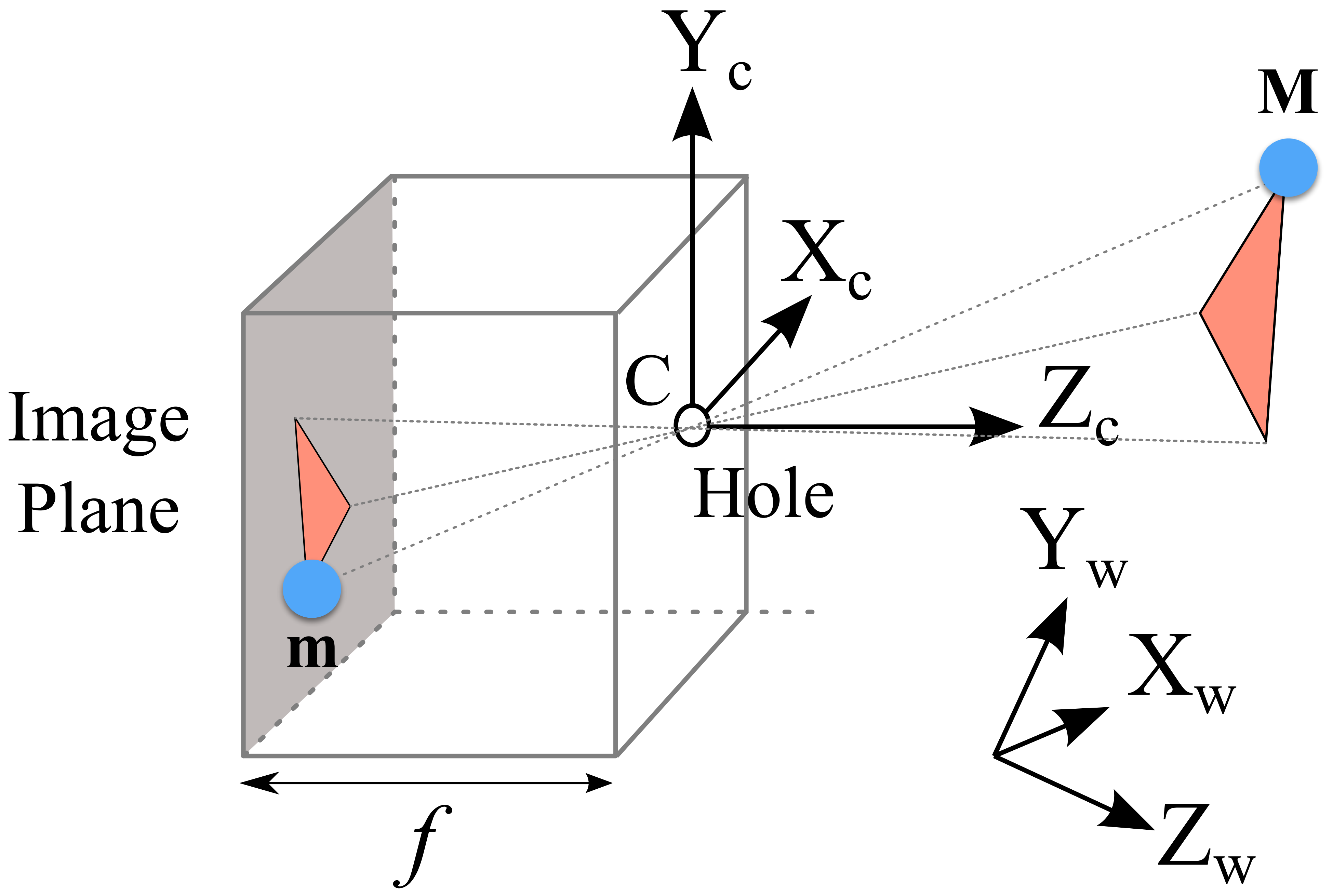


image-space

world-space

Camera Model

- $\mathbf{M}$ is a point in the 3D world, and it is defined as:

$$\mathbf{M} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- $\mathbf{m}$ is a 2D point, the projection of $\mathbf{M}$. $\mathbf{m}$ lives in the image plane UV:

$$\mathbf{m} = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

Camera Model

- By analyzing the two triangles (real-world and projected one), the following relationship emerges:

$$\frac{f}{z} = -\frac{u}{x} = -\frac{v}{y}$$

- This means that:

$$\begin{cases} u = -\frac{f}{z} \cdot x \\ v = -\frac{f}{z} \cdot y \end{cases}$$

Camera Model: Intrinsic Parameters

- If we take all into account of the optical center, and pixel size we obtain:

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

- If we put this in matrix form, we obtain:

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

$$\mathbf{m}z = P \cdot \mathbf{M}$$

Camera Model: Extrinsic Parameters

- Note that K is called ***intrinsic matrix*** and has all projective properties of the camera.

- We need to define how the camera is placed (i.e., rotation and translation). This is described by the ***extrinsic matrix*** G:

$$G = \begin{bmatrix} R & \mathbf{t} \\ 0 & 1 \end{bmatrix} \qquad \mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} \quad R = \begin{bmatrix} \mathbf{r}_1^\top \\ \mathbf{r}_2^\top \\ \mathbf{r}_3^\top \end{bmatrix}$$

- R is a 3x3 rotation matrix, which is an orthogonal matrix with determinant 1.

- $\mathbf{t}$ is translation vector with three components.

Appendix B:
From Pixels to Rays

Rendering: Ray Creation

- We need to create a ray r with an origin and a direction:
  - Origin is set to C; the center of the virtual camera:

$$\mathbf{o} = \mathrm{C}$$

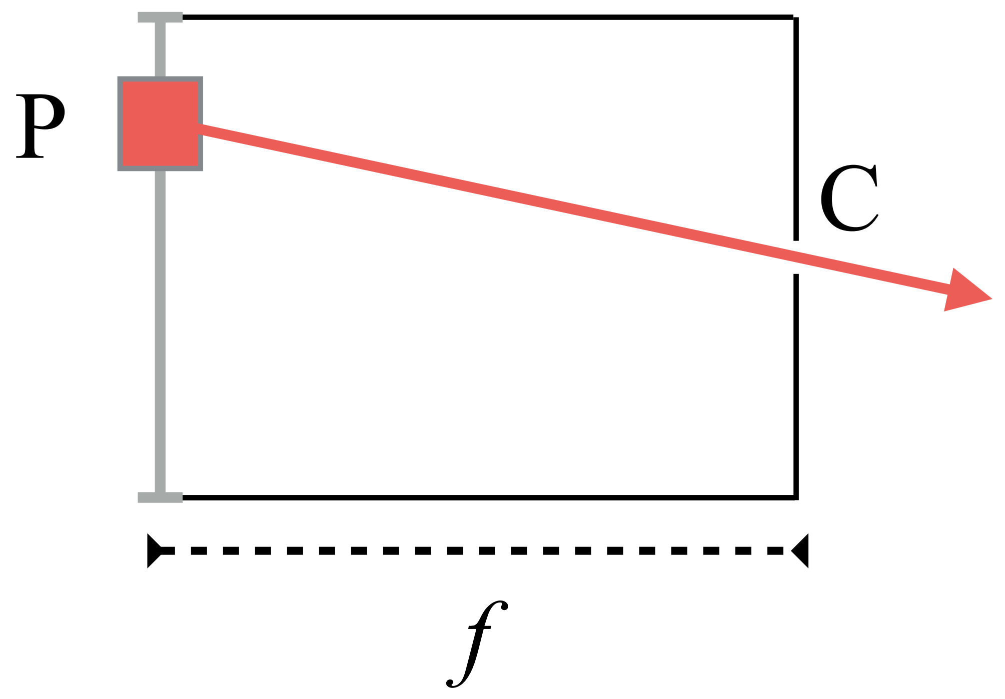
  - This is because the ray has to pass through it!

Rendering: Ray Creation

- Given a pixel coordinates (u, v), we need to compute the 3D point P inside the camera by inverting:

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

- knowing z is set to f.



Rendering: Ray Creation

- Therefore, the point P is:

$$P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{(u-u_0)}{k_u} \\ \frac{(v-v_0)}{k_v} \\ -f \\ 1 \end{bmatrix}$$

- and, the ray direction is simply computed as:

$$\vec{d} = \frac{C - P}{\|C - P\|}$$

Camera Model

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

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

- P is 3x4 matrix with 11 independent parameters!

Appendix C: Ray-Volume Boundary Intersection

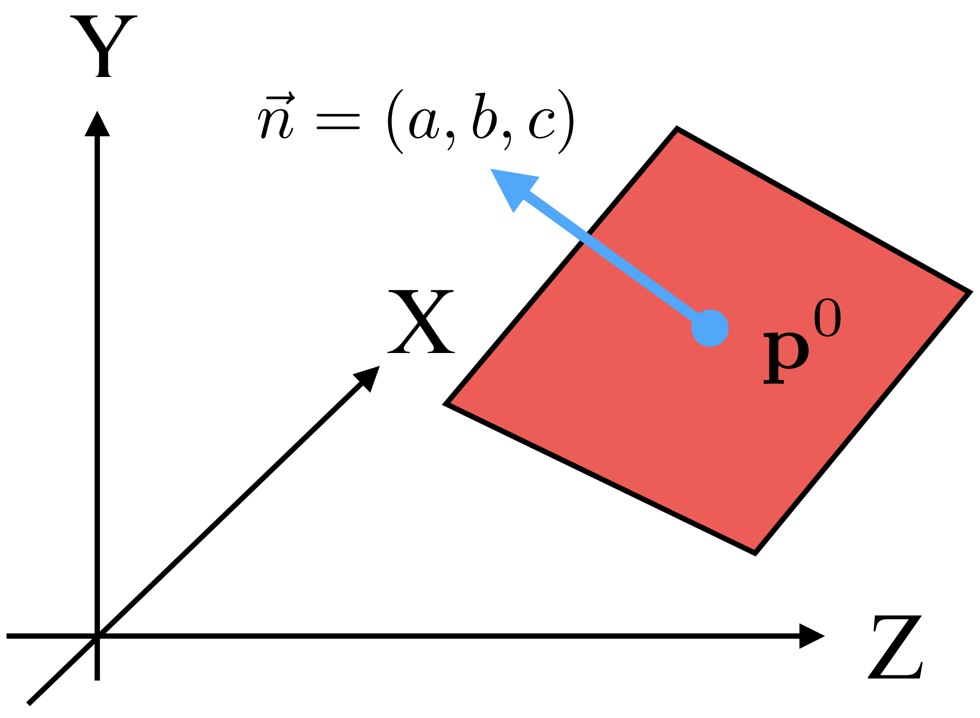
Ray-Box Intersection

- As the first step, we need to find the intersection ray-box. The volume boundary is just a box!
- We know that a box has six faces; i.e., planes:
    - We need to check intersection against six planes

Rendering: Ray-Plane Intersection

- A plane is defined by its normal (a, b, c) and a shift parameter (D):

$$a \cdot x + b \cdot y + c \cdot z + D = 0$$



$$D = -a \cdot p_x^0 - b \cdot p_y^0 - c \cdot p_z^0$$

Rendering: Ray-Plane Intersection

- We need to solve the system:

$$\begin{cases} \mathbf{p}(t) = \mathbf{o} + \vec{d} \cdot t \qquad t > 0 \\ a \cdot p_x + b \cdot p_y + c \cdot p_z + D = 0 \end{cases}$$

Its solution is

$$\vec{v} = \mathbf{p}^0 - \mathbf{o}$$

$$t = \frac{\vec{v} \cdot \vec{n}}{\vec{n} \cdot \vec{d}} \qquad (\vec{n} \cdot \vec{d}) > 0$$