Fall 2018

CSCI 420: **Computer Graphics**

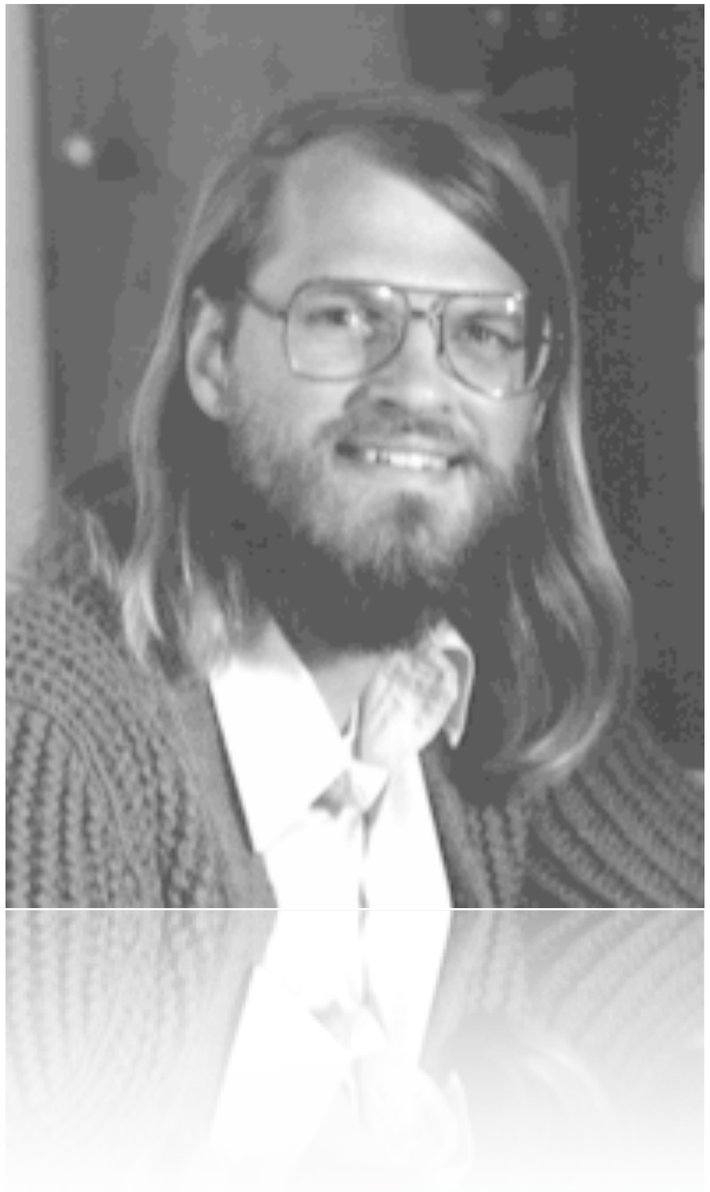
6.2 Bump Mapping
& Clipping

Hao Li

http://cs420.hao-li.com

Bump Mapping

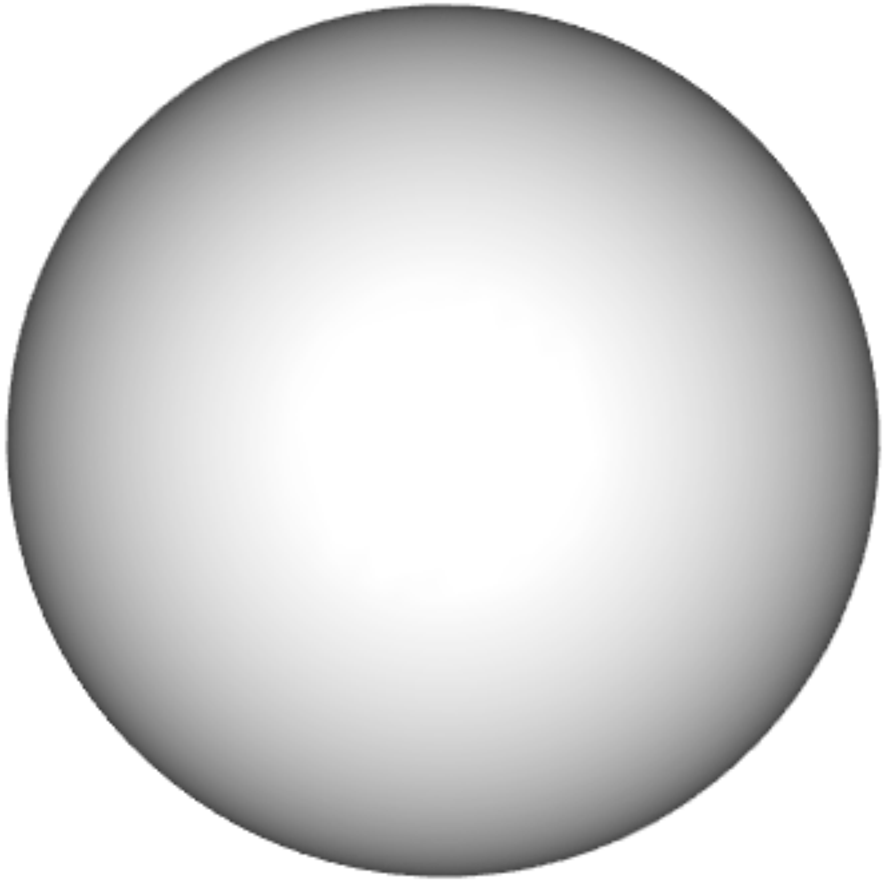
A long time ago, in 1978



… bump mapping was born

courtesy by ZBrush

For Meshes



vertex normal interpolation

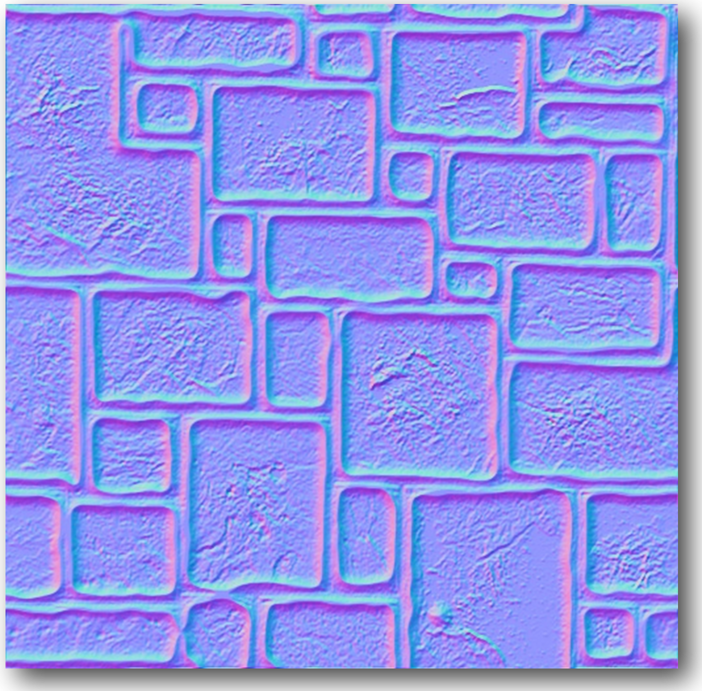
↓

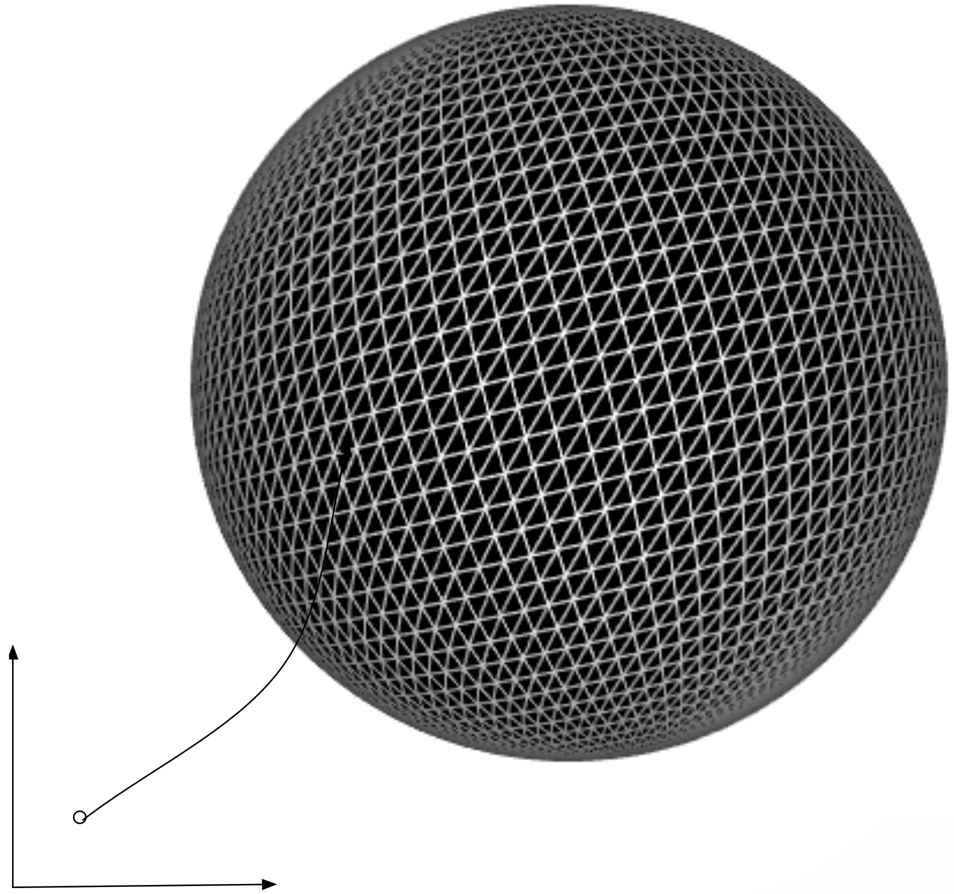
smooth shading

What about
accessing **textures** to modify **surface normals**...

Goal

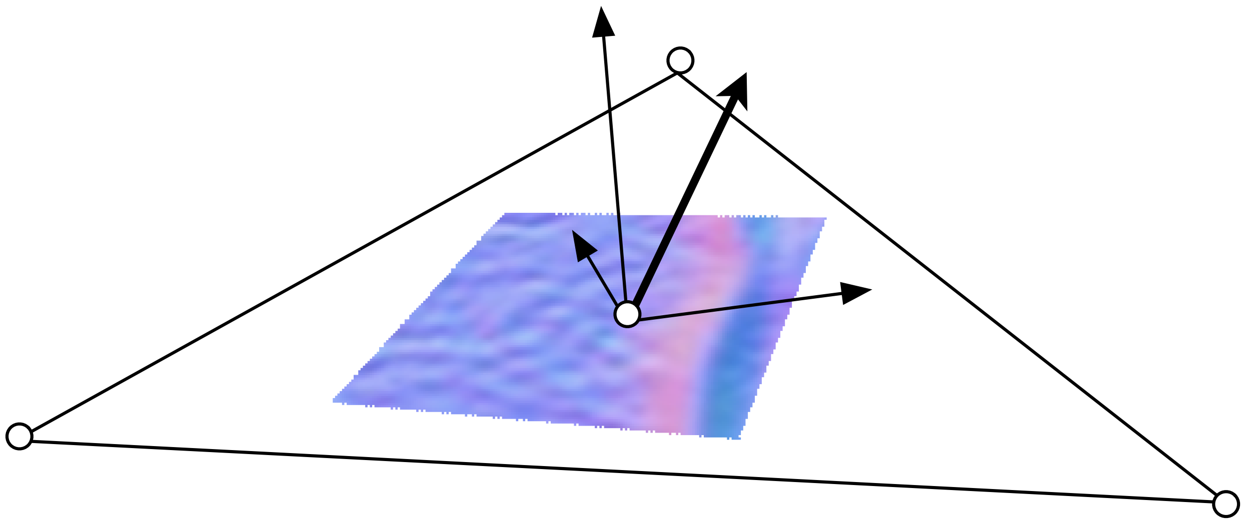
Use **bump map normals** given a **parametrized mesh**



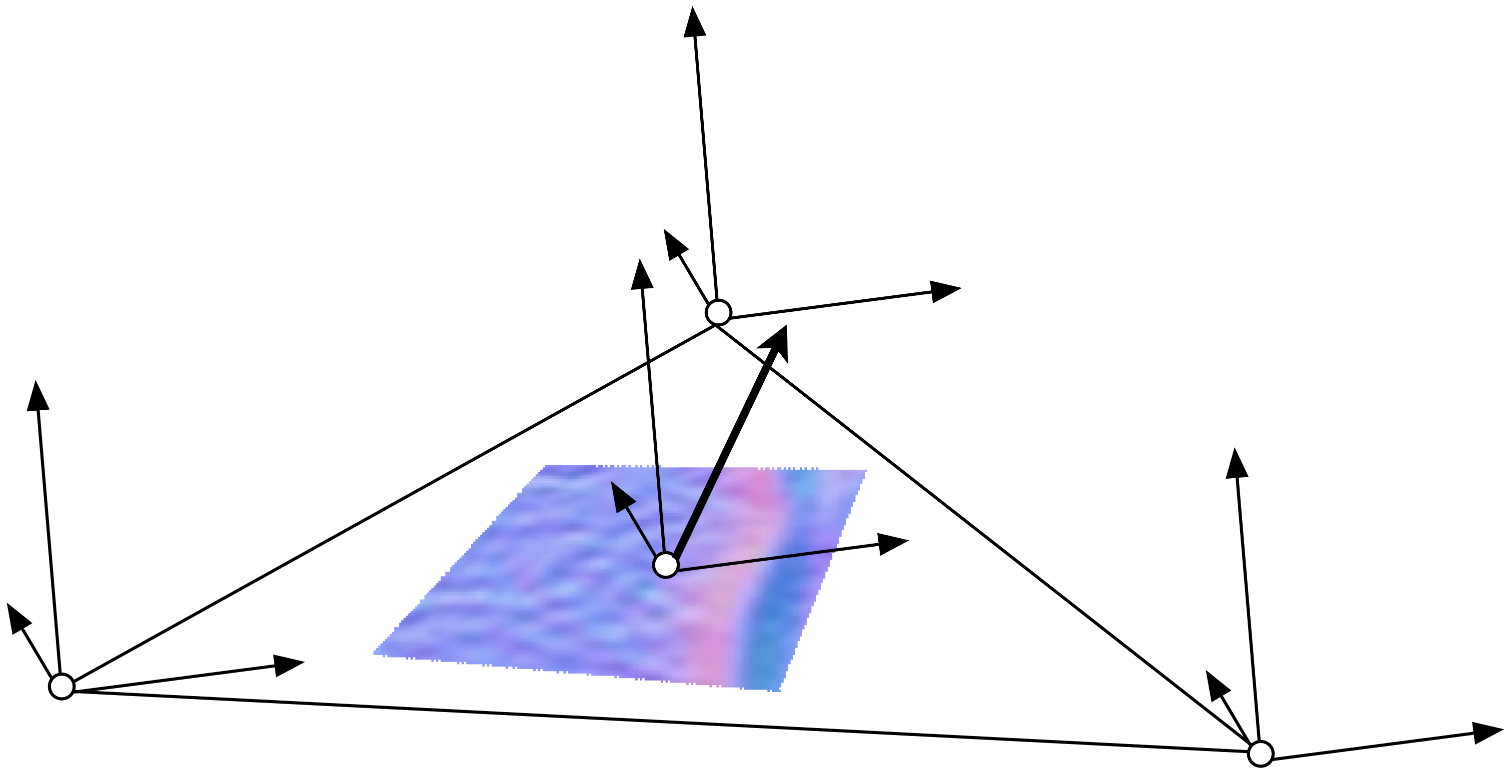


$$\mathbf{u} = \begin{bmatrix} u \\ v \end{bmatrix} \in \mathbb{R}^2$$

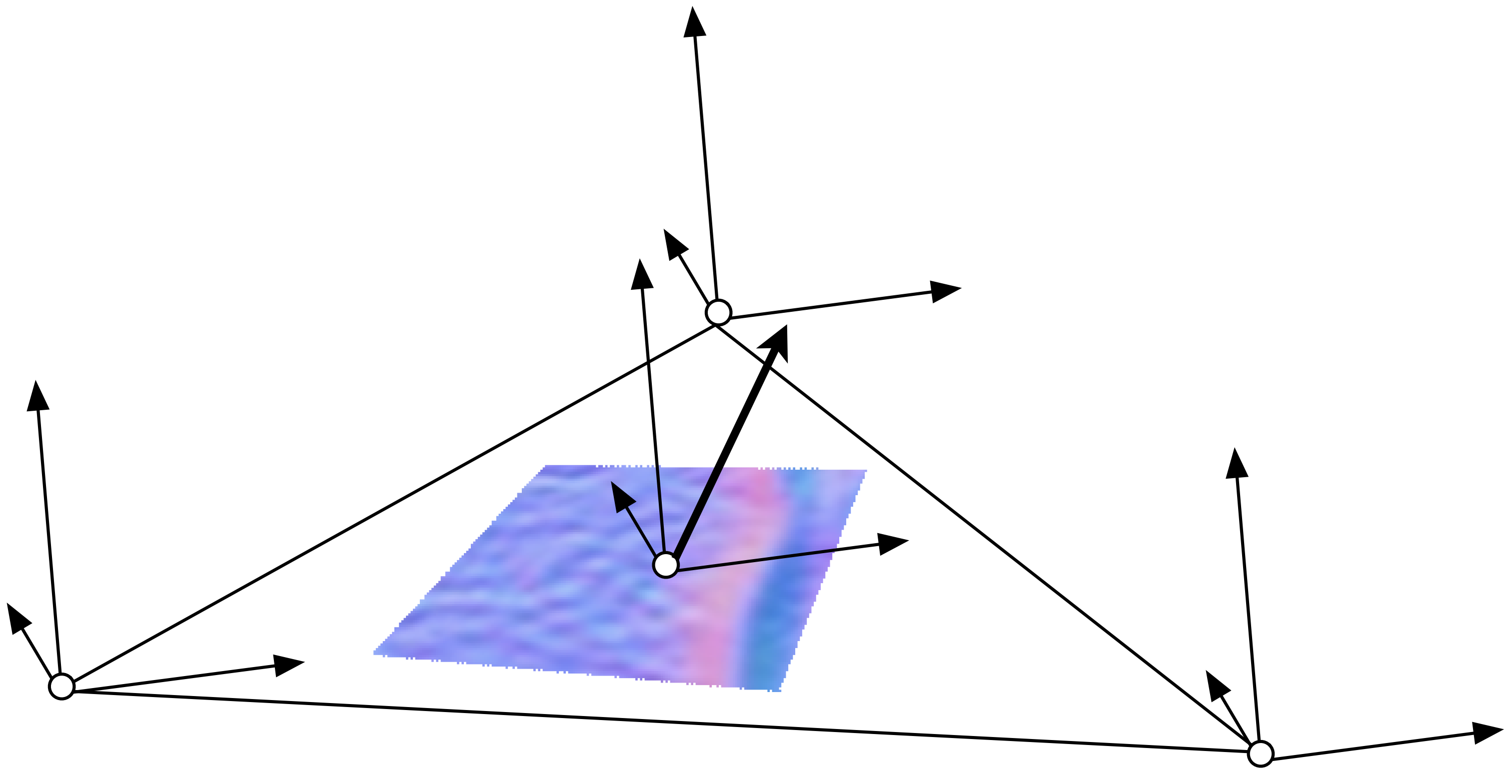
Bump map normals are defined in a **local coordinate frame** inside a triangle



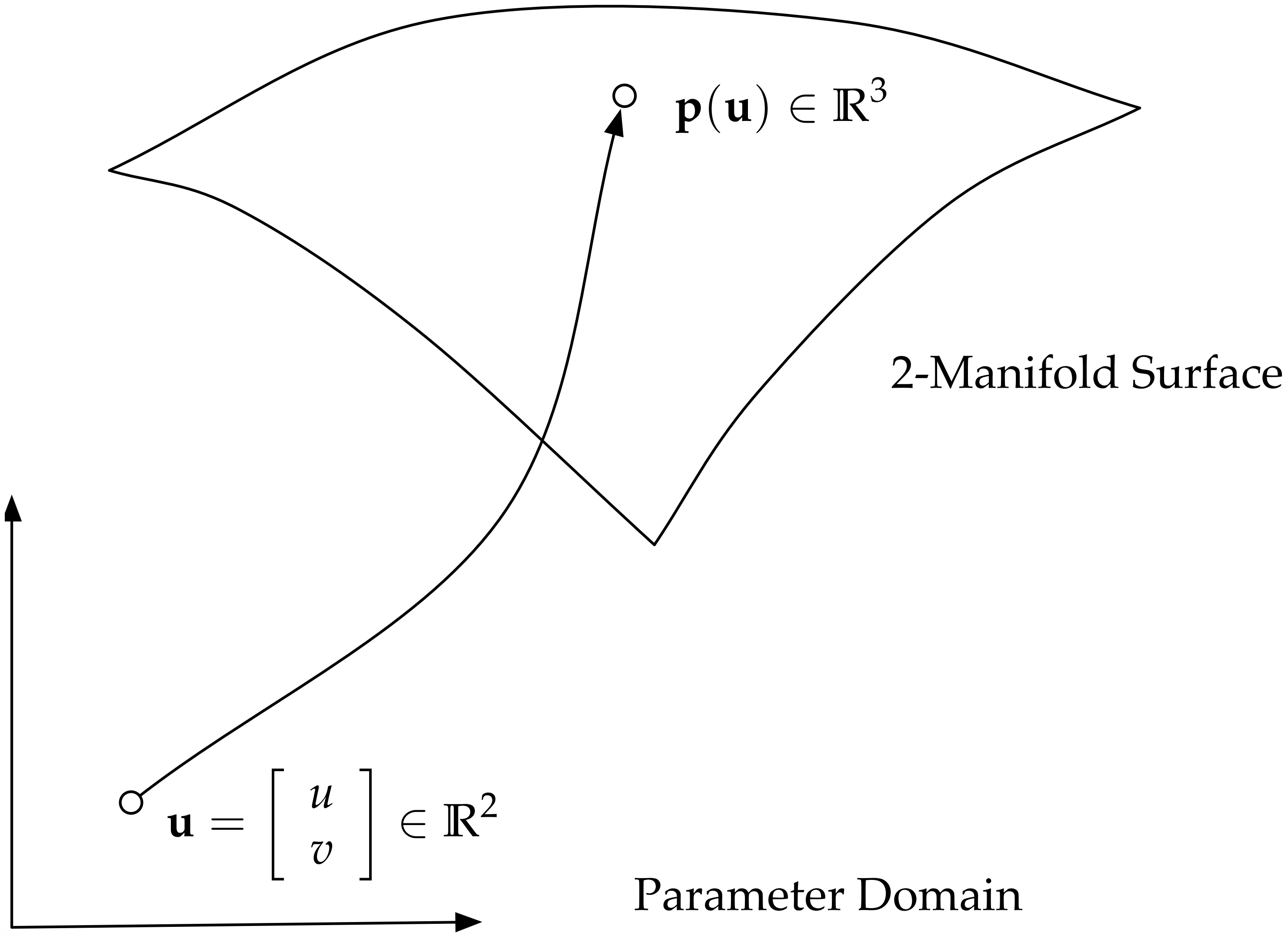
We have **positions, normals** and **parameters** of the triangle corners



How do we obtain coordinate frame?



Some Differential Geometry



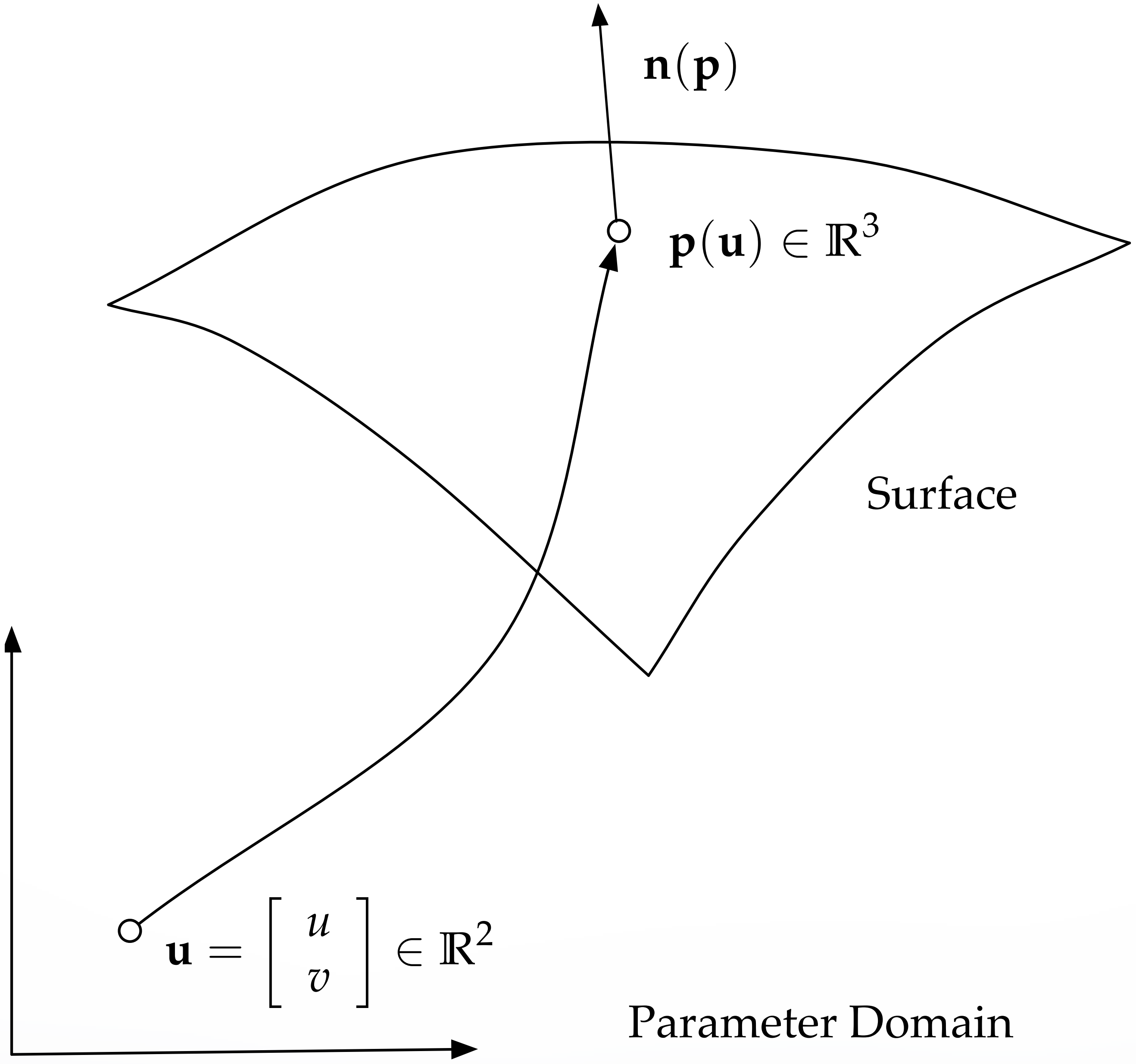
$\mathbf{p}(\mathbf{u}) \in \mathbb{R}^3$

2-Manifold Surface

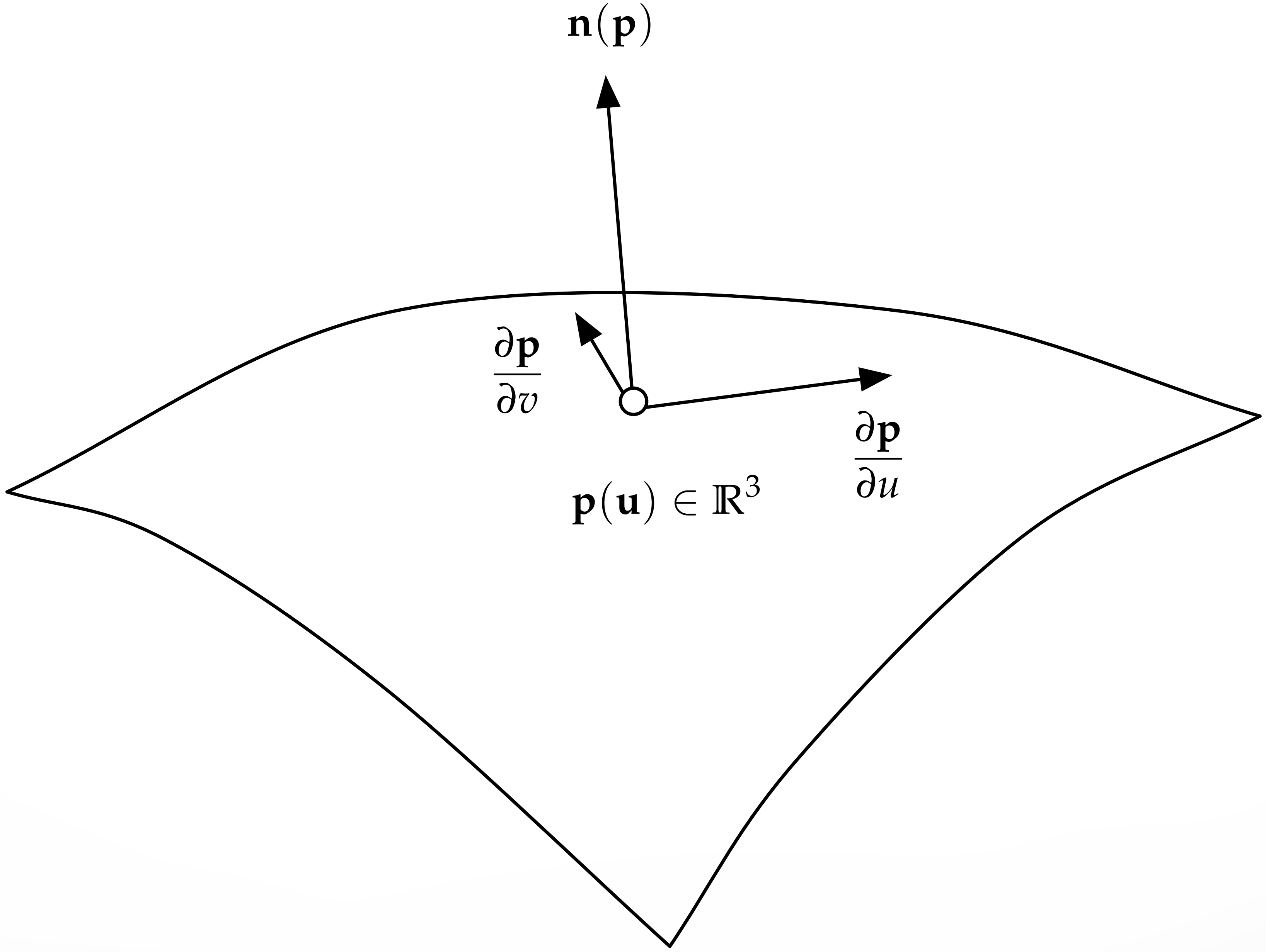
$\mathbf{u} = \begin{bmatrix} u \\ v \end{bmatrix} \in \mathbb{R}^2$

Parameter Domain

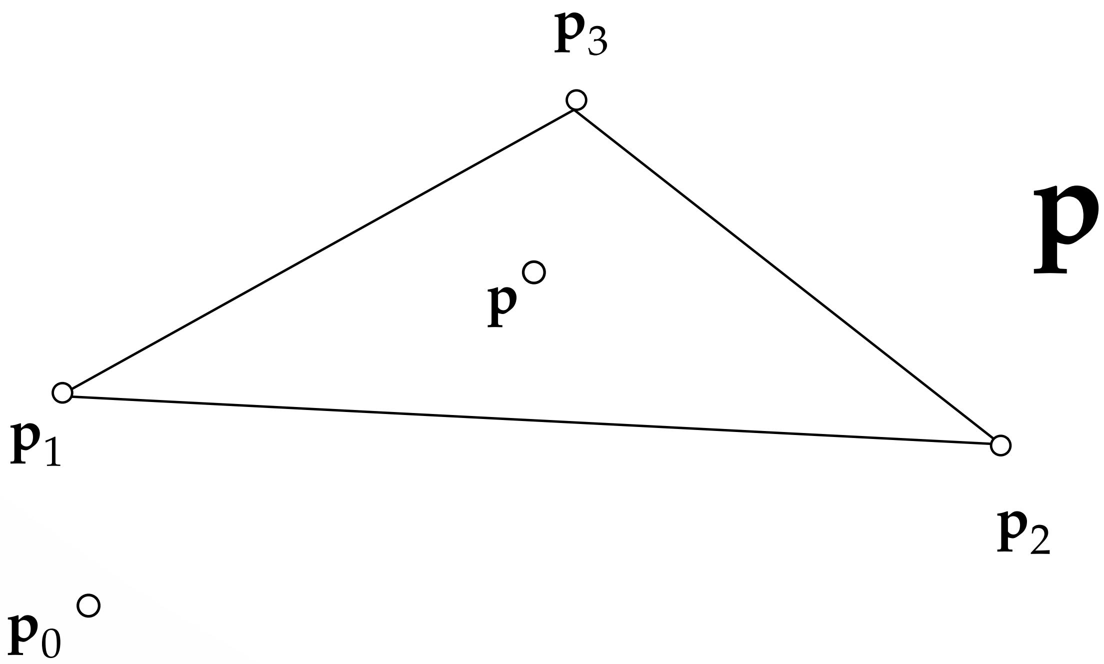
Surface normals for shading



Surface normals obtained from tangent space



Tangent vectors inside triangles



$$\mathbf{p}_i = \mathbf{p}_0 + u_i \frac{\partial \mathbf{p}}{\partial u} + v_i \frac{\partial \mathbf{p}}{\partial v}$$

Fully determined from positions and parameters

we are not interested in $\mathbf{p}_0$

$$\mathbf{p}_2 - \mathbf{p}_1 = (u_2 - u_1)\frac{\partial \mathbf{p}}{\partial u} + (v_2 - v1)\frac{\partial \mathbf{p}}{\partial v}$$

$$\mathbf{p}_3 - \mathbf{p}_1 = (u_3 - u_1)\frac{\partial \mathbf{p}}{\partial u} + (v_3 - v1)\frac{\partial \mathbf{p}}{\partial v}$$

2x2 Matrix Inversion

$$\mathbf{p}_2 - \mathbf{p}_1 = (u_2 - u_1)\frac{\partial \mathbf{p}}{\partial u} + (v_2 - v1)\frac{\partial \mathbf{p}}{\partial v}$$

$$\mathbf{p}_3 - \mathbf{p}_1 = (u_3 - u_1)\frac{\partial \mathbf{p}}{\partial u} + (v_3 - v1)\frac{\partial \mathbf{p}}{\partial v}$$

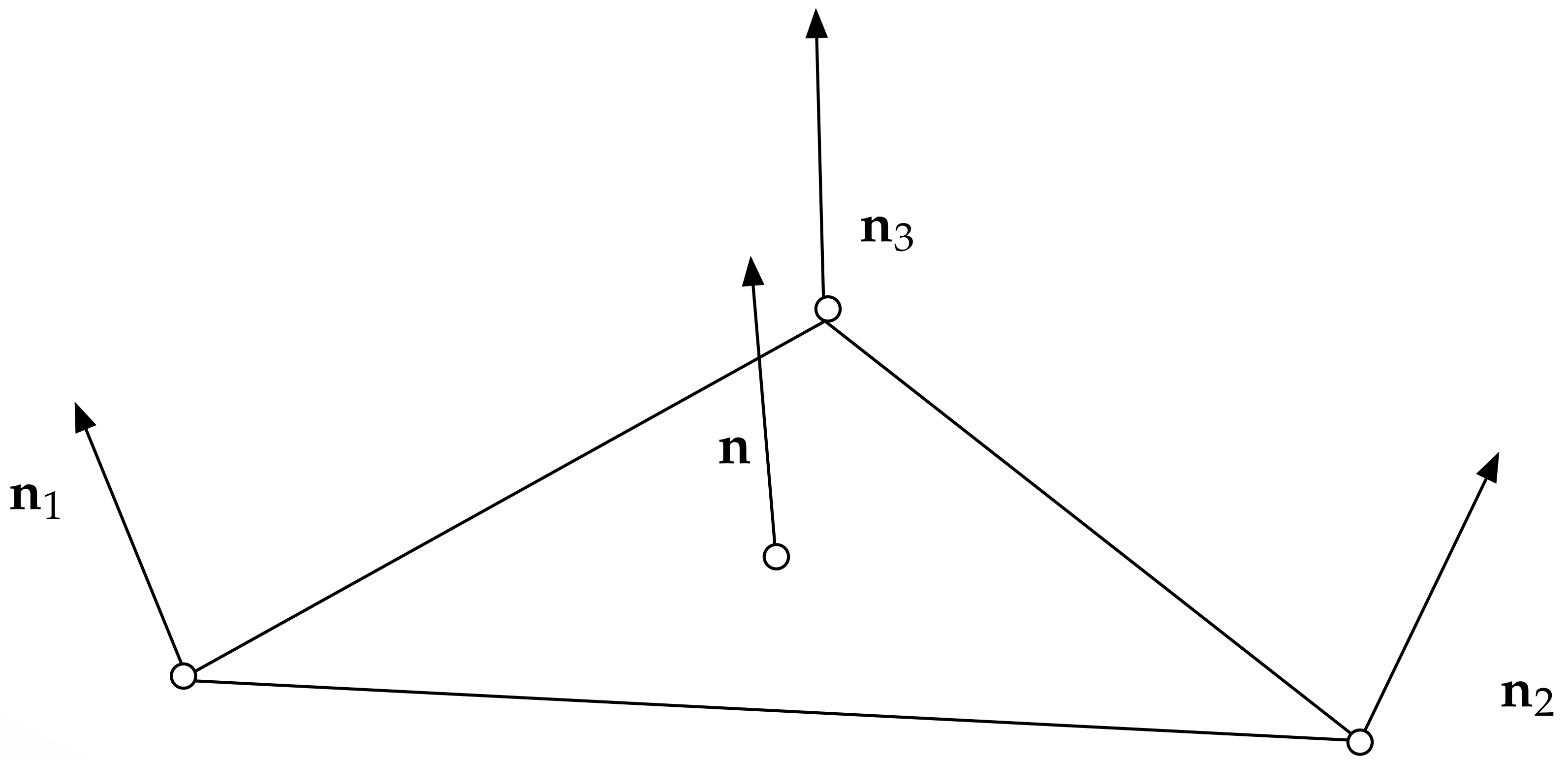
$$\downarrow$$

$$\begin{bmatrix} \mathbf{p}_2 - \mathbf{p}_1 & \mathbf{p}_3 - \mathbf{p}_1 \end{bmatrix} = \begin{bmatrix} \frac{\partial \mathbf{p}}{\partial u} & \frac{\partial \mathbf{p}}{\partial v} \end{bmatrix} \begin{bmatrix} (u_2 - u_1) & (u_3 - u_1) \\ (v_2 - v_1) & (v_3 - v_1) \end{bmatrix}$$

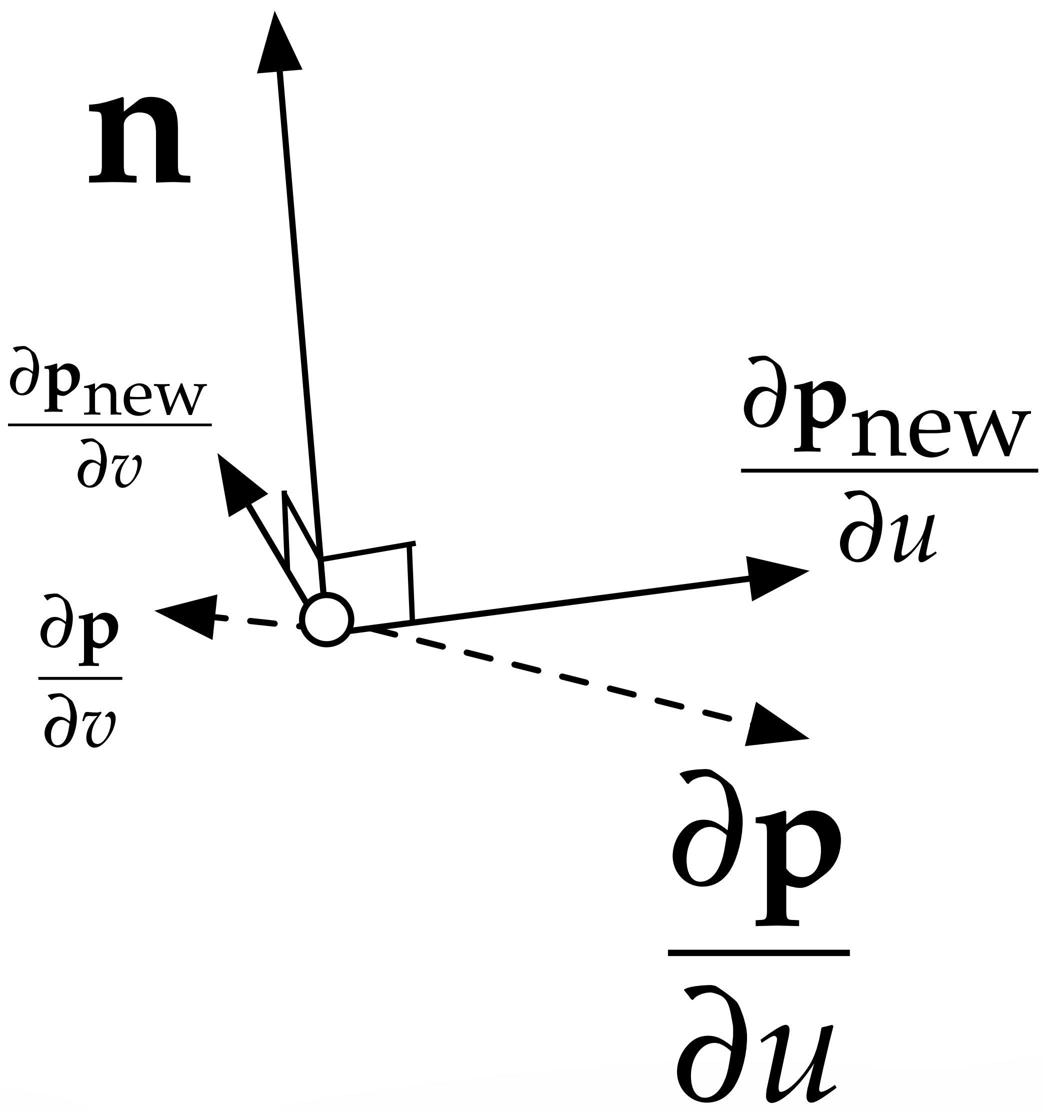
correct if mesh is planar

Normals Interpolation (see Phong Shading)

$$\mathbf{n} = \alpha_1\mathbf{n}_1 + \alpha_2\mathbf{n}_2 + \alpha_3\mathbf{n}_3 \quad \text{from} \quad \mathbf{p} = \alpha_1\mathbf{p}_1 + \alpha_2\mathbf{p}_2 + \alpha_3\mathbf{p}_3$$



Tangent vectors orthogonal to normal



We now have an **inexpensive way** to add **geometric details**

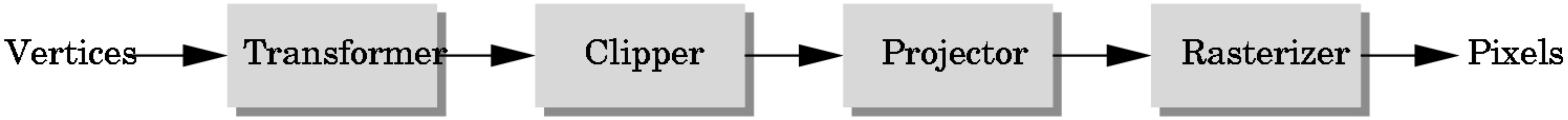
Other bump mapping techniques exist

Further Readings

- “Simulation of Wrinkled Surfaces” [Blinn 1978]
- “Real-Time Rendering” [Akenine-Möller and Haines 2002] p.166 – 177

Clipping

The Graphics Pipeline, Revisited



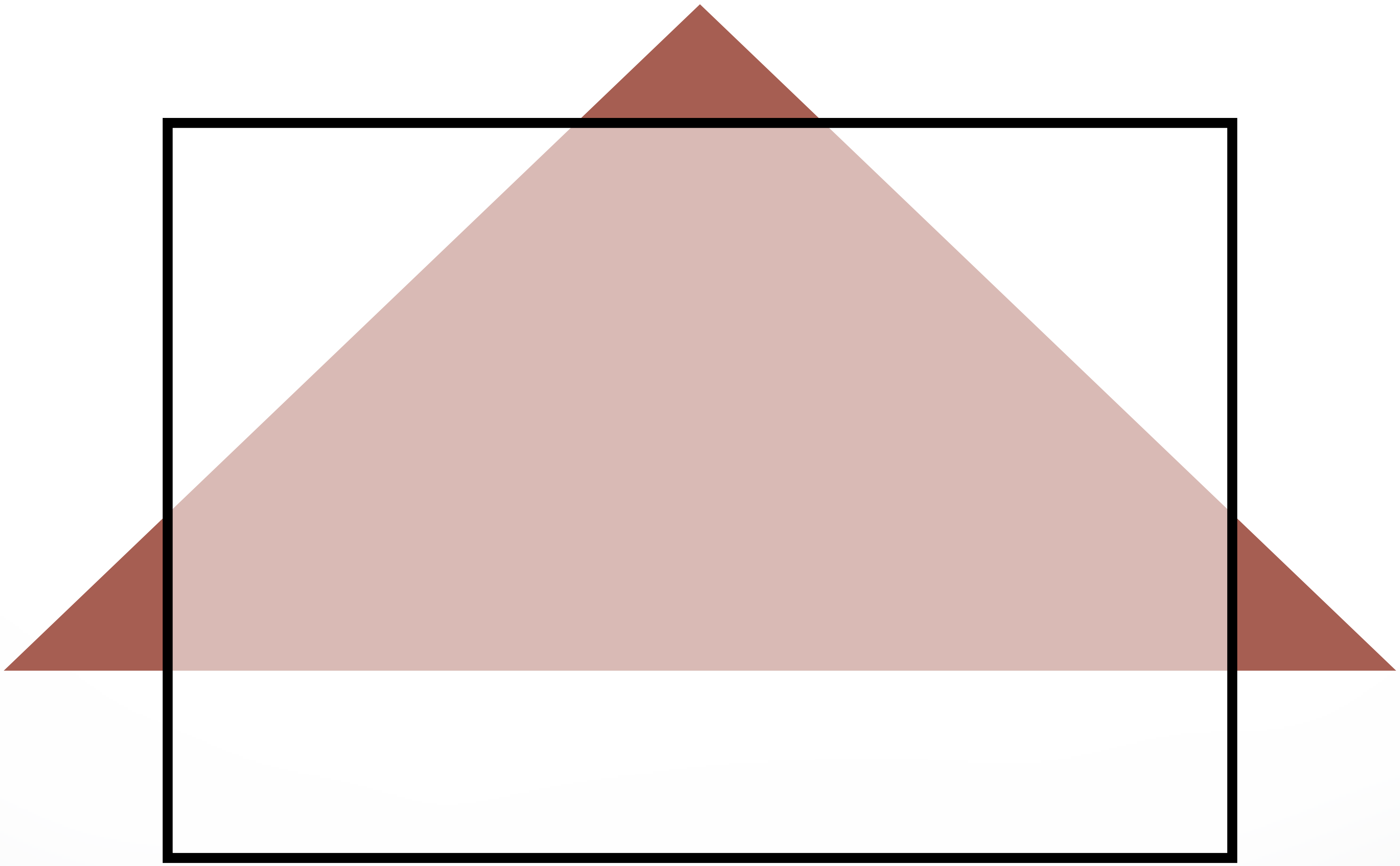
Vertices → Transformer → Clipper → Projector → Rasterizer → Pixels

- Must eliminate objects that are outside of viewing frustum
- Clipping: object space (eye coordinates)
- Scissoring: image space (pixels in frame buffer)

  - most often less efficient than clipping
- We will first discuss 2D clipping (for simplicity)

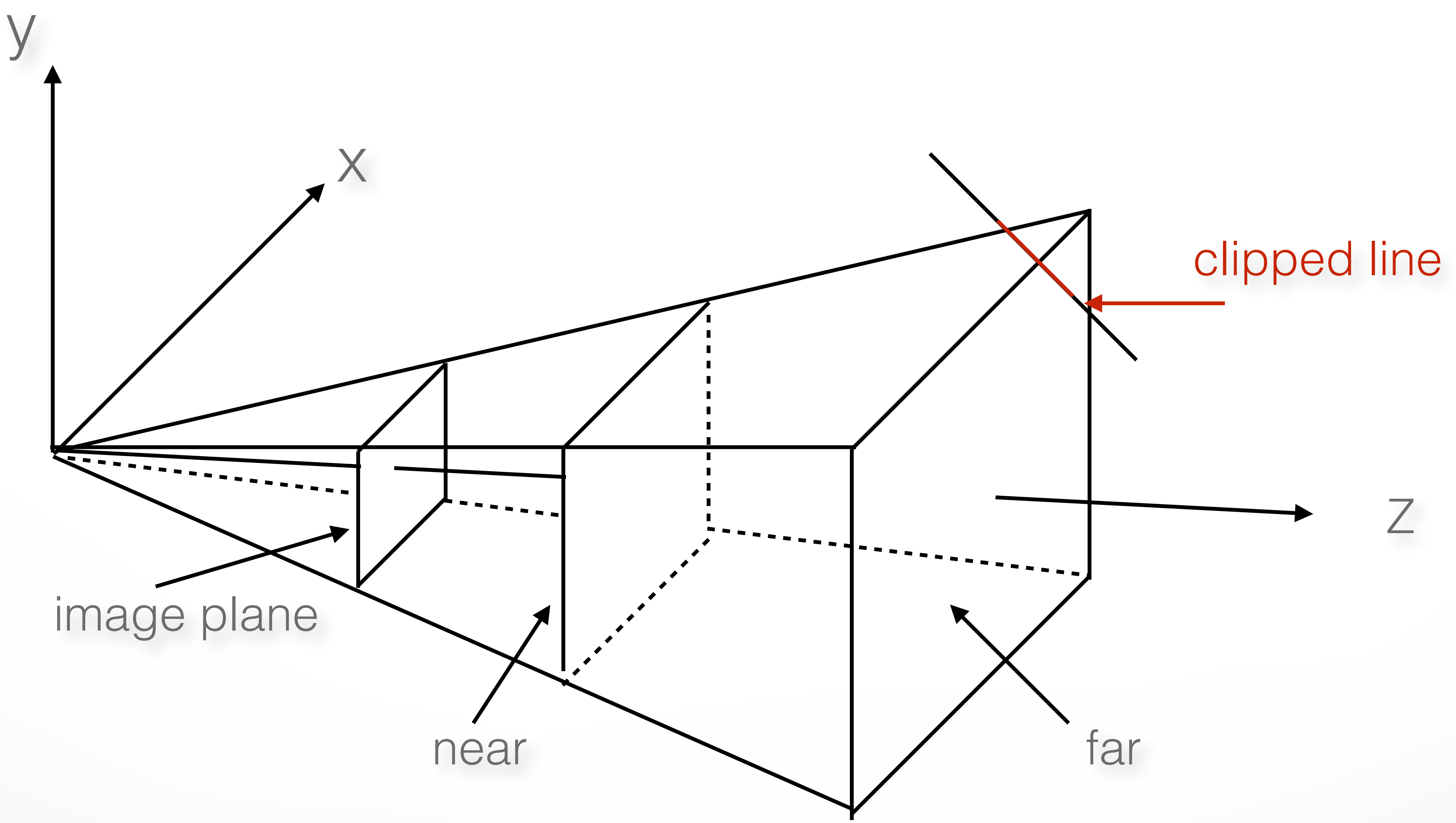
  - OpenGL uses 3D clipping

2D Clipping Problem



Clipping Against a Frustum

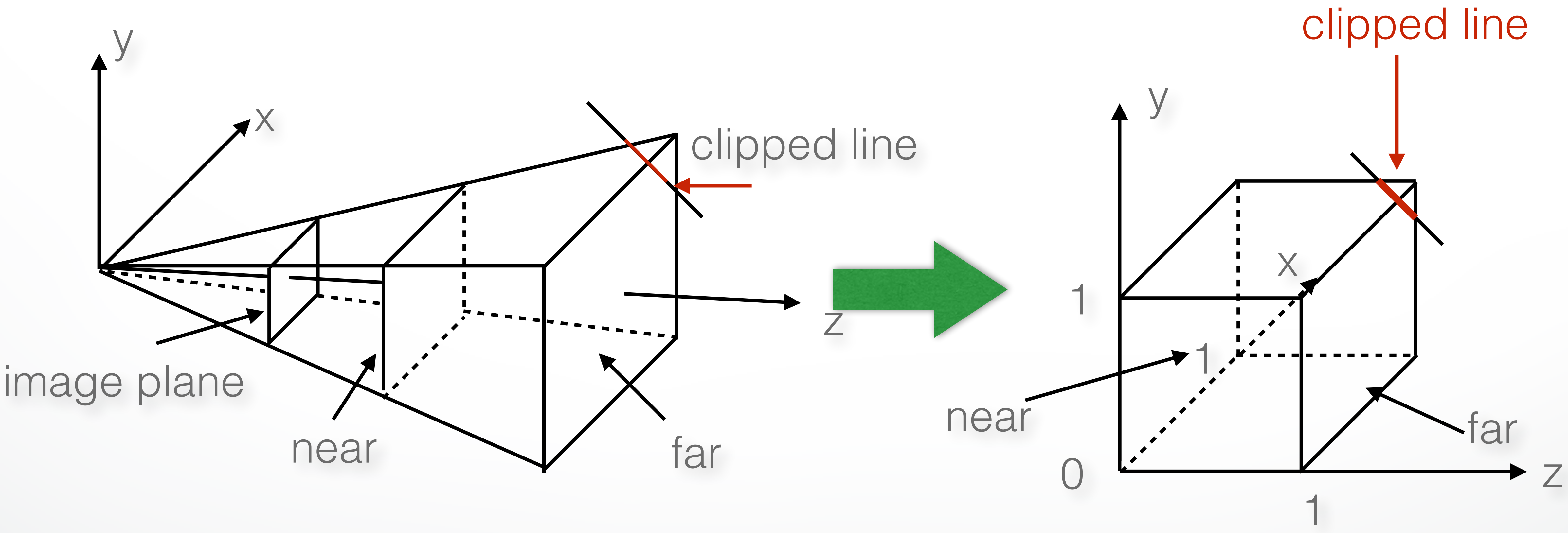
- General case of frustum (truncated pyramid)



- Clipping is tricky because of frustum shape

Perspective Normalization

- Solution:
  - Implement perspective projection by perspective normalization and orthographic projection
  - Perspective normalization is a homogeneous transformation



The Normalized Frustum

- OpenGL uses $-1 \leq x,y,z \leq 1$ (others possible)
- Clip against resulting cube
- Clipping against arbitrary (programmer-specified) planes requires more general algorithms and is more expensive

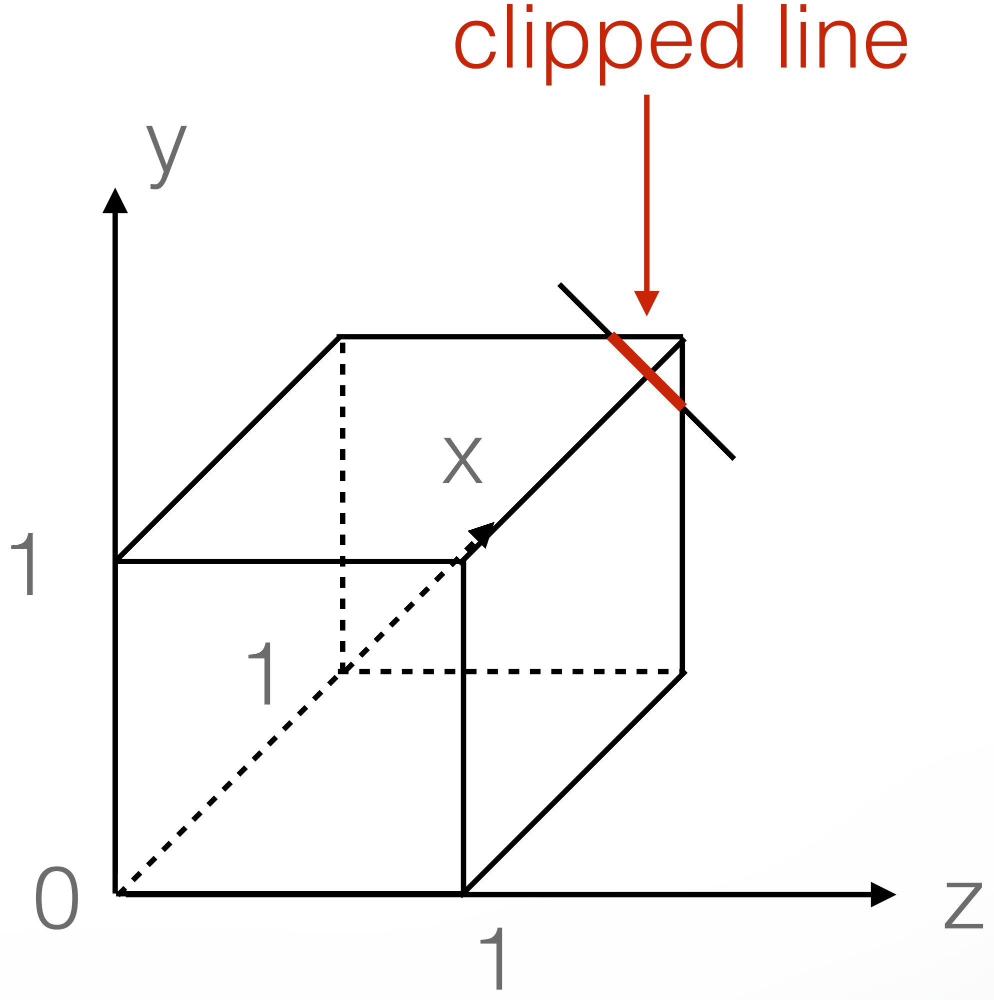
The Viewport Transformation

- Transformation sequence again:
  1. Camera: From object coordinates to eye coords
  2. Perspective normalization: to clip coordinates
  3. Clipping
  4. Perspective division: to normalized device coords
  5. Orthographic projection (setting $z_p = 0$)
  6. Viewport transformation: to screen coordinates

- Viewport transformation can distort
  - Solution: pass the correct window aspect ratio to gluPerspective

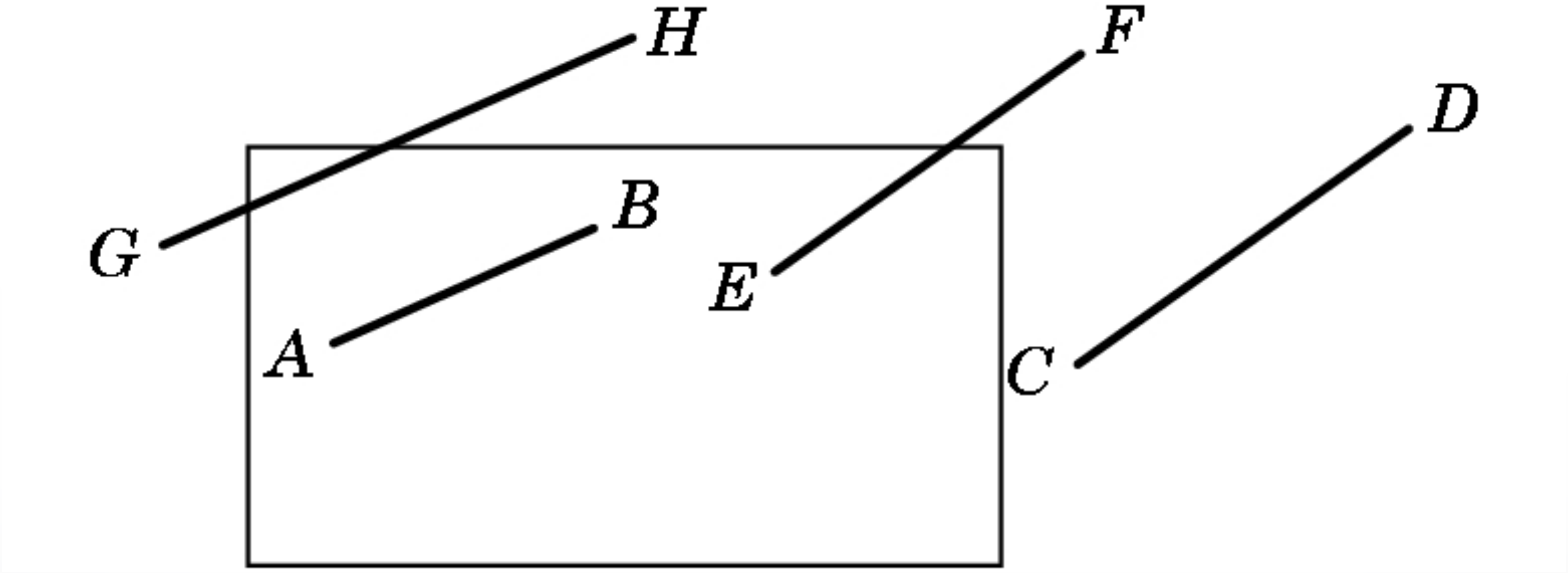
Clipping

- General: 3D object against cube
- Simpler case:
  - In 2D: line against square or rectangle
  - Later: polygon clipping



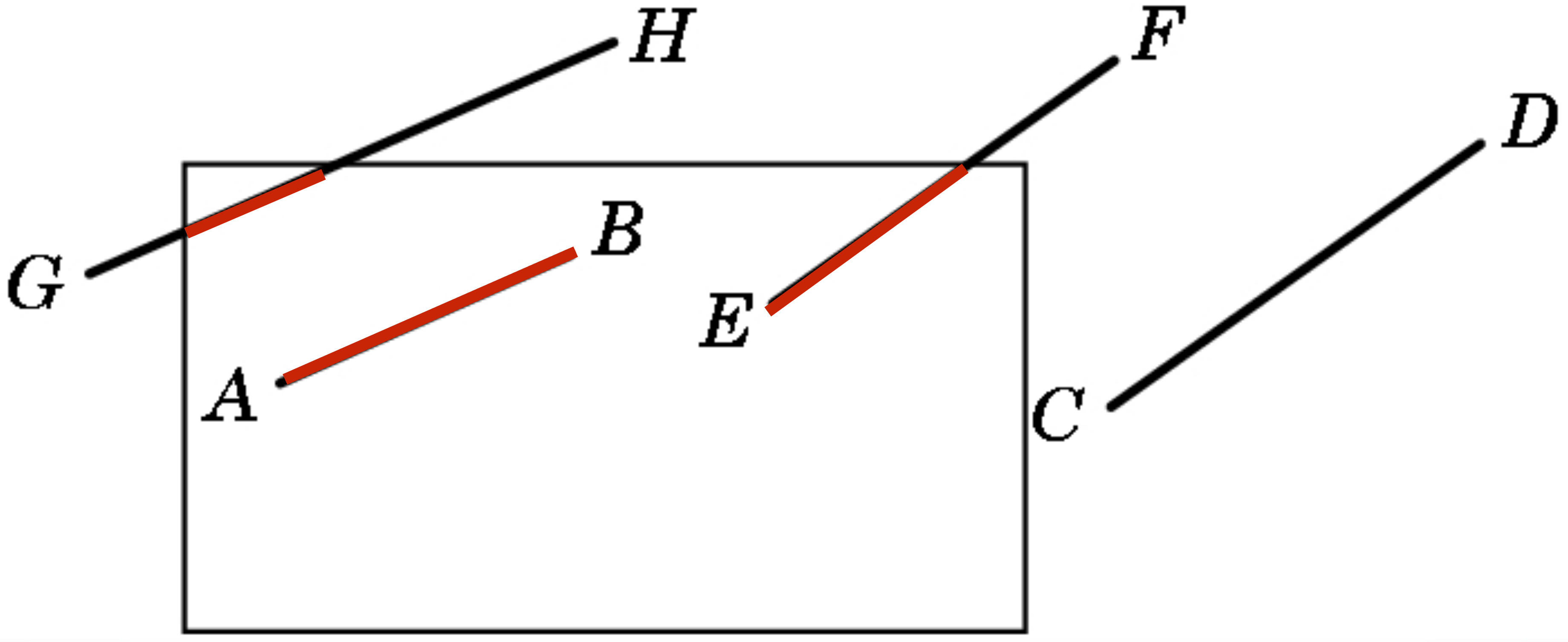
Clipping Against Rectangle in 2D

- Line-segment clipping: modify endpoints of lines to lie within clipping rectangle



Clipping Against Rectangle in 2D

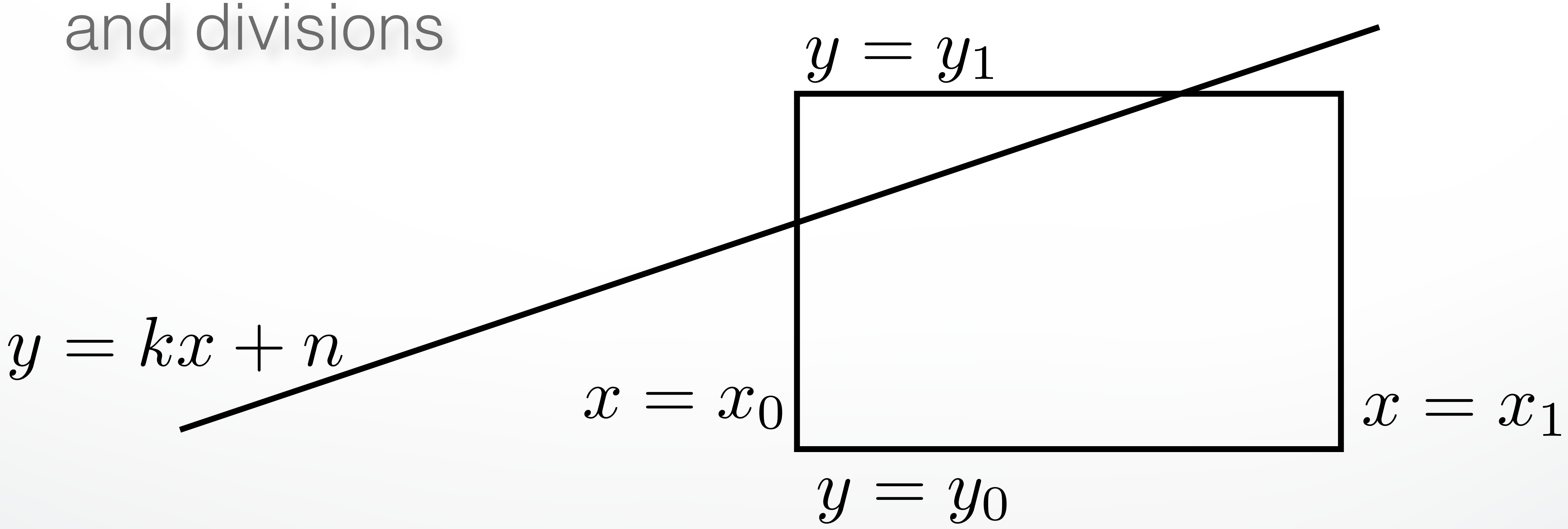
- The result (in red)



Clipping Against Rectangle in 2D

- Could calculate intersections of line segments with clipping rectangle
  - expensive, due to floating point multiplications and divisions

- Want to minimize the number of multiplications and divisions

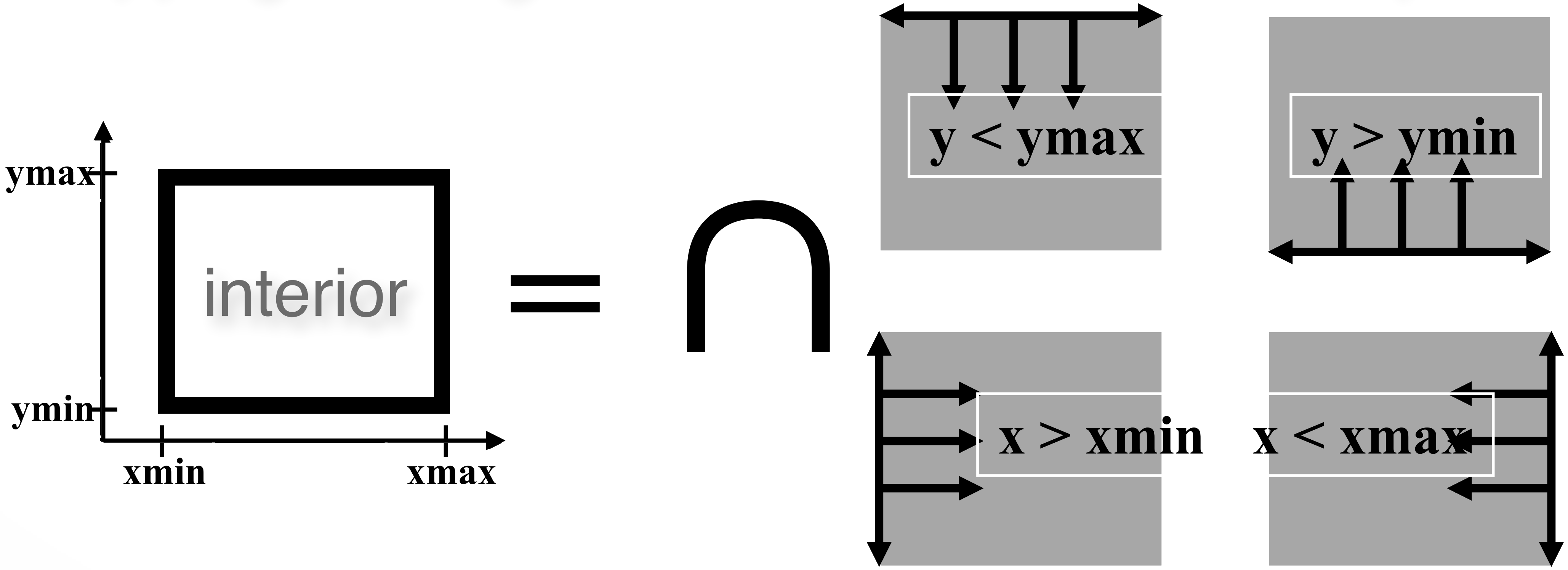


Several practical algorithms for clipping

- Main motivation:
  Avoid expensive line-rectangle intersections
  (which require floating point divisions)
- Cohen-Sutherland Clipping
- Liang-Barsky Clipping
- There are many more
  (but many only work in 2D)

Cohen-Sutherland Clipping

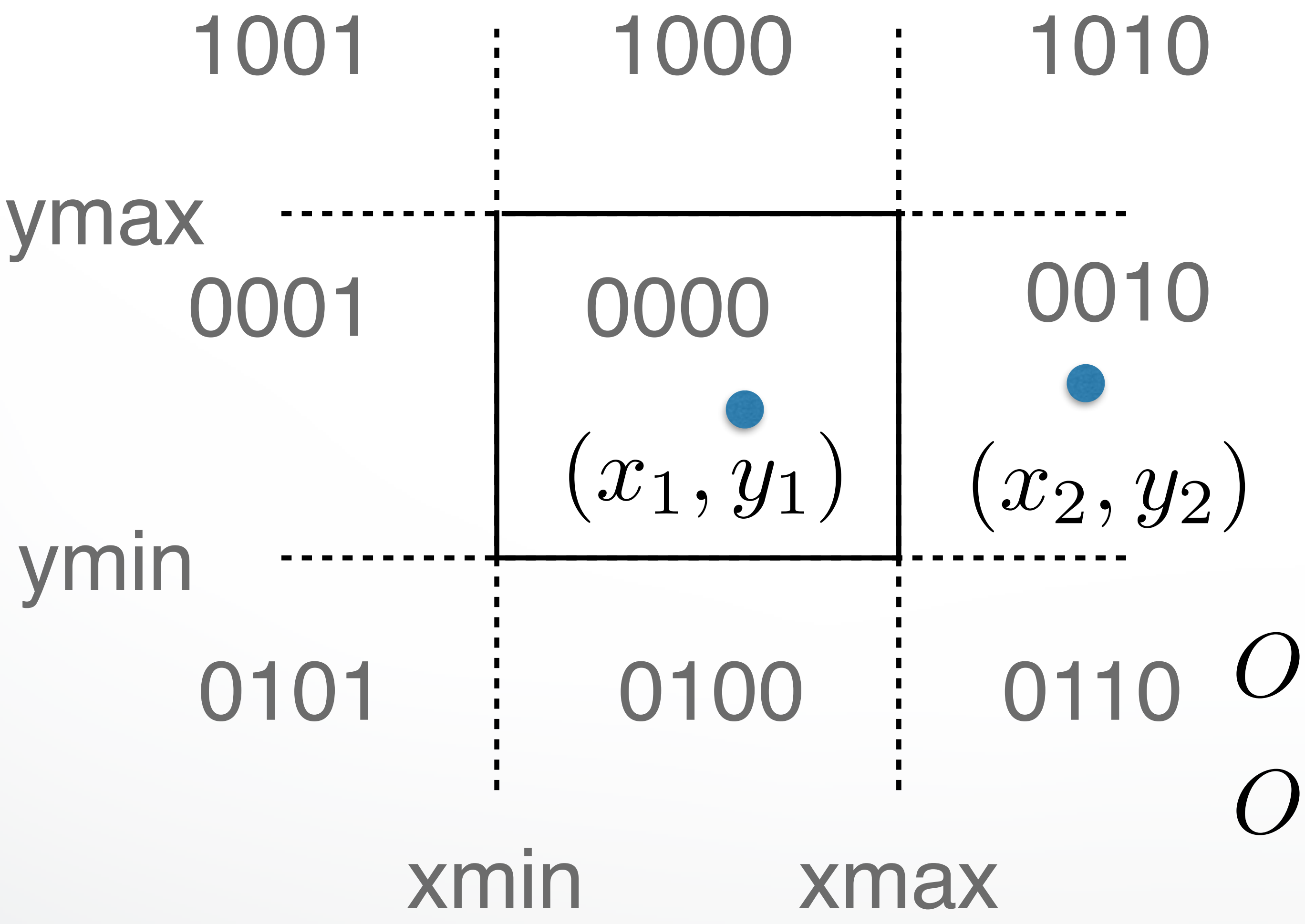
- Clipping rectangle is an intersection of 4 half-planes



- Encode results of four half-plane tests
- Generalizes to 3 dimensions (6 half-planes)

Outcodes (Cohen-Sutherland)

- Divide space into 9 regions
- 4-bit outcode determined by comparisons (TBRL)



$$b_0 : y > y_{max}$$

$$b_1 : y < y_{min}$$

$$b_2 : x > x_{max}$$

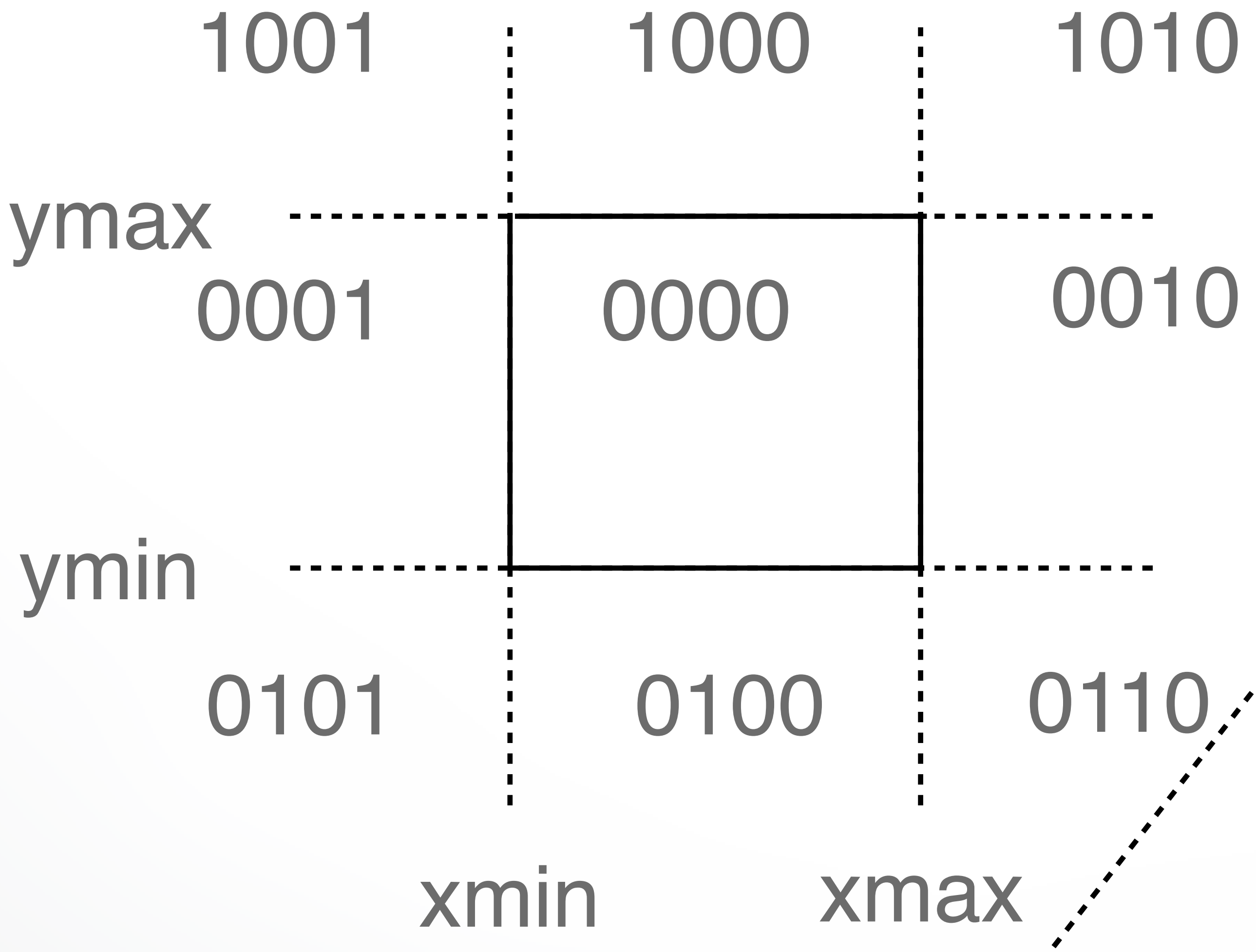
$$b_3 : x < x_{min}$$

$$O_1 = outcode(x_1, y_1)$$

$$O_1 = outcode(x_2, y_2)$$

Cases for Outcodes

- Outcomes: accept, reject, subdivide



$O_1 = O_2 = 0000$: accept entire segment

$O_1 \,\&\, O_2 \neq 0000$: reject entire segment

$O_1 = 0000, O_2 \neq 0000$: subdivide

$O_1 \neq 0000, O_2 = 0000$: subdivide

$O_1 \,\&\, O_2 = 0000$: subdivide

bitwise AND

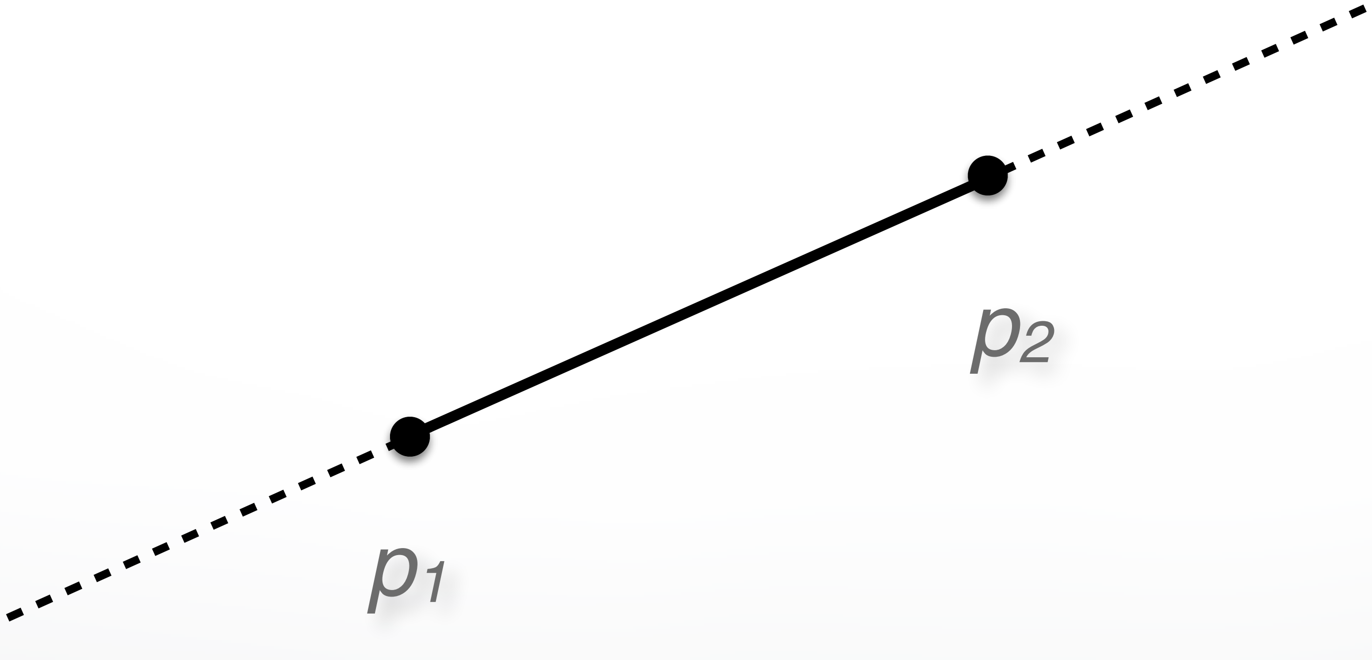
Cohen-Sutherland Subdivision

- Pick outside endpoint ($o \neq 0000$)
- Pick a crossed edge ($o = b_0 b_1 b_2 b_3$ and $b_k \neq 0$)
- Compute intersection of this line and this edge
- Replace endpoint with intersection point
- Restart with new line segment
  - Outcodes of second point are unchanged
- This algorithms converges

Liang-Barsky Clipping

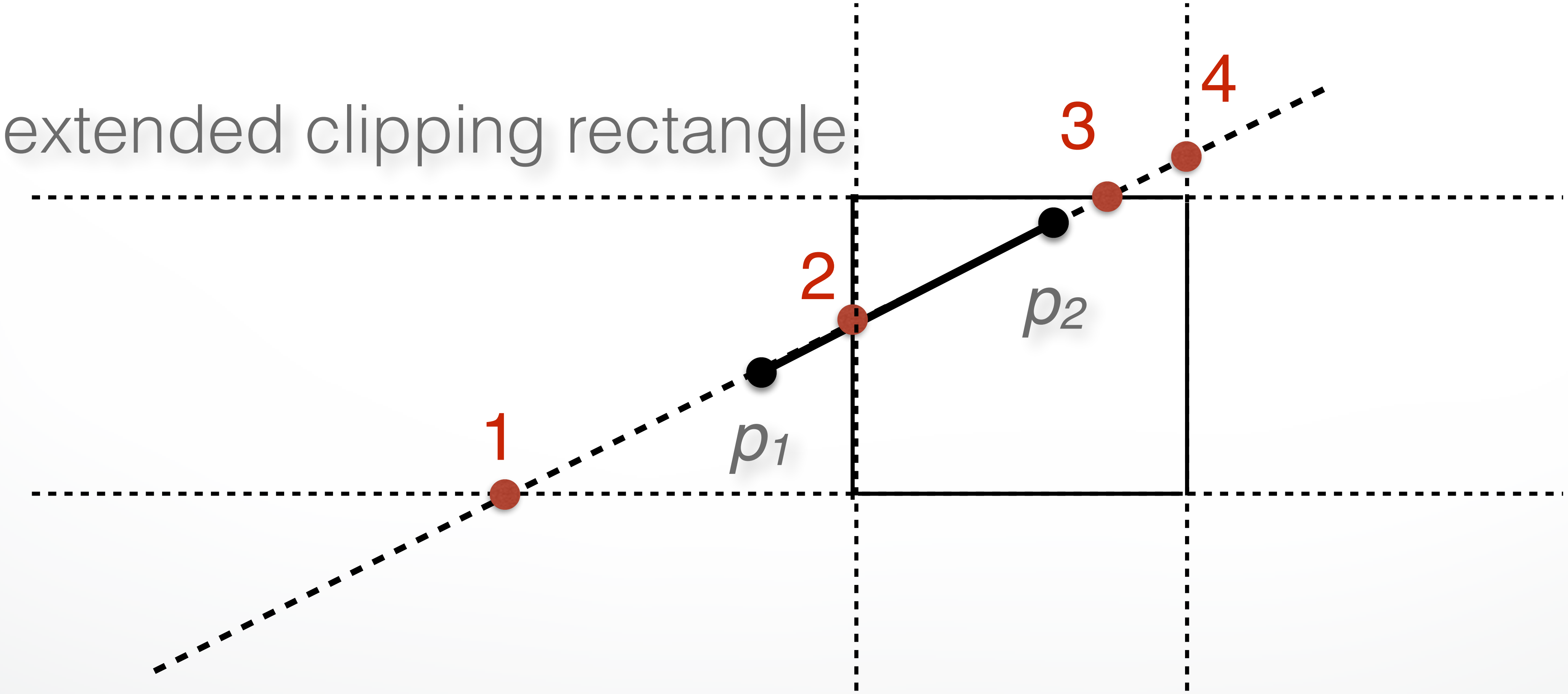
- Start with parametric form for a line

$$
\begin{aligned}
p(\alpha) &= (1-\alpha)p_1 + \alpha p_2, \qquad 0 \le \alpha \le 1 \\
x(\alpha) &= (1-\alpha)x_1 + \alpha x_2 \\
y(\alpha) &= (1-\alpha)y_1 + \alpha y_2
\end{aligned}
$$

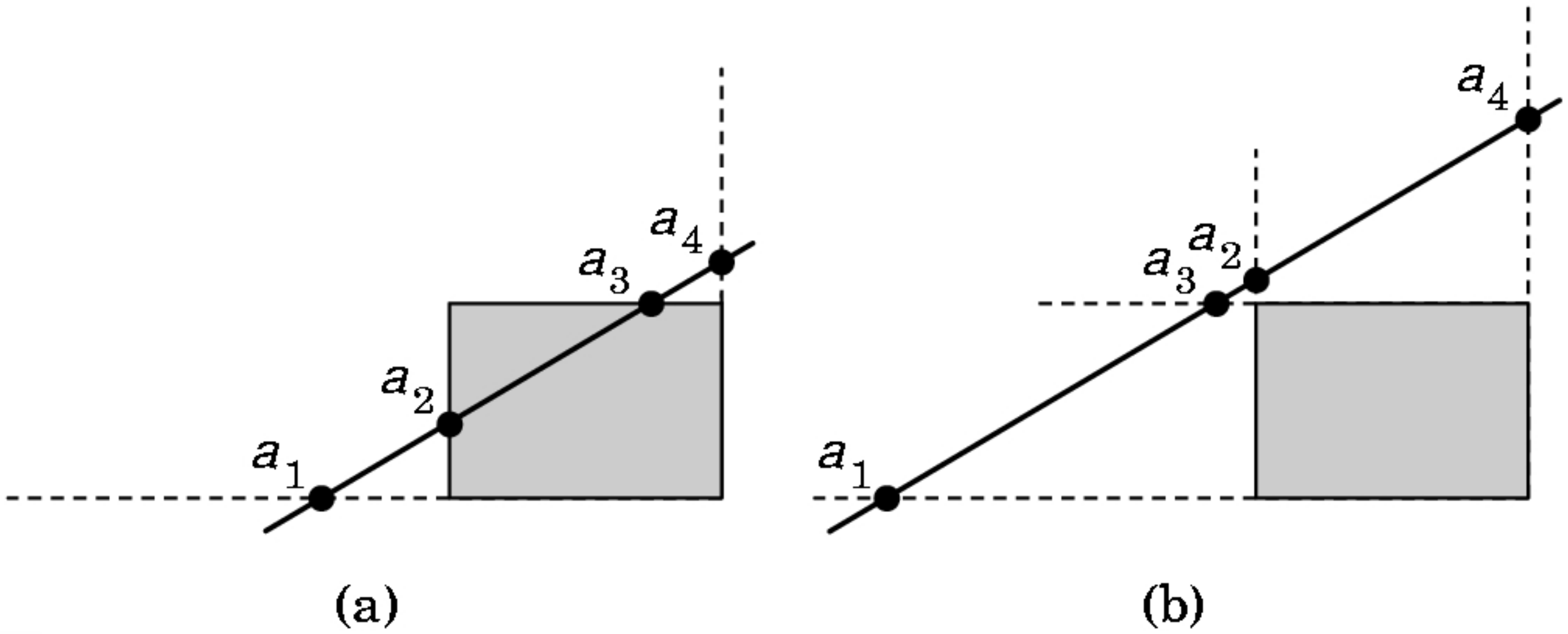


Liang-Barsky Clipping

- Compute all four intersections 1,2,3,4 with extended clipping rectangle
- Often, no need to compute all four intersections



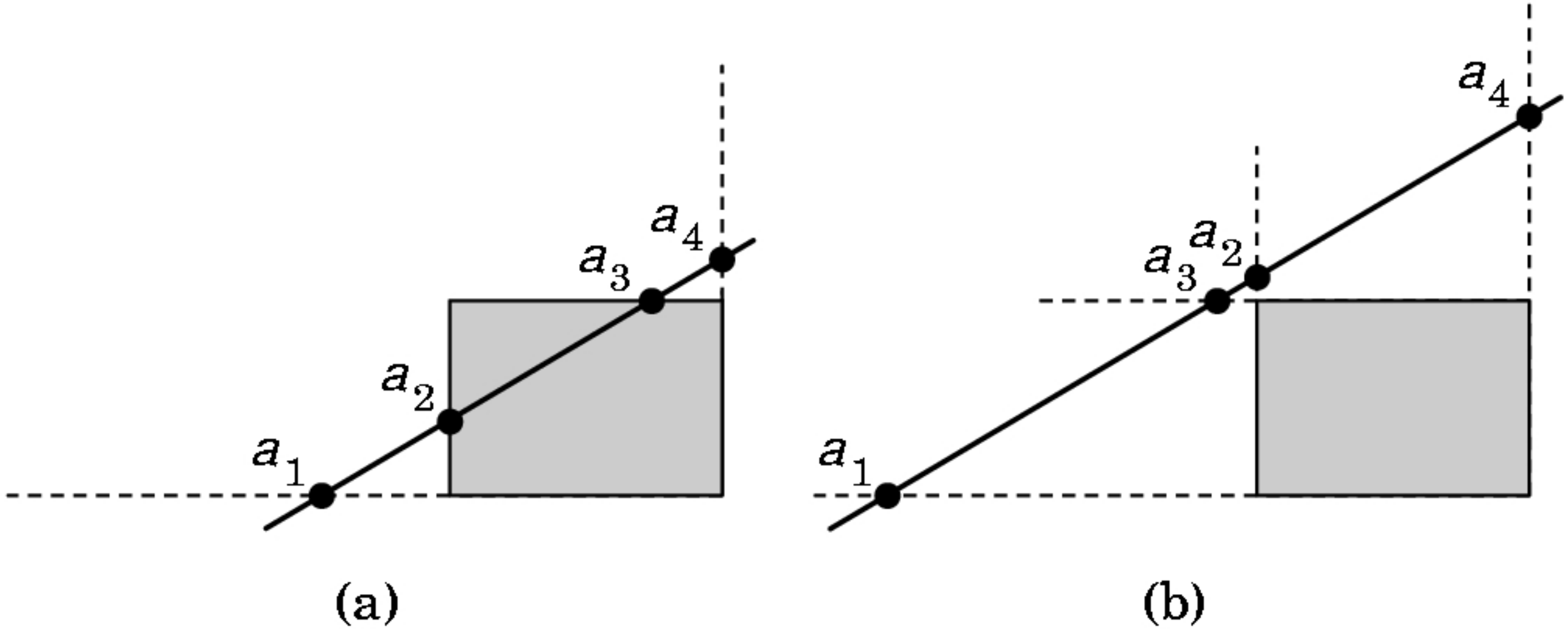
Ordering of intersection points



(a) (b)

- Order the intersection points
- Figure (a): $1 > \alpha_4 > \alpha_3 > \alpha_2 > \alpha_1 > 0$
- Figure (b): $1 > \alpha_4 > \alpha_2 > \alpha_3 > \alpha_1 > 0$

Liang-Barsky Idea



- It is possible to clip already if one knows the order of the four intersection points !

- Even if the actual intersections were not computed !

- Can enumerate all ordering cases

Liang-Barsky efficiency improvements

- Efficiency improvement 1:
  - Compute intersections one by one
  - Often can reject before all four are computed

- Efficiency improvement 2:
  - Equations for α_3, α_2

$$y_{\max} = (1 - \alpha_3)y_1 + \alpha_3 y_2$$

$$x_{\min} = (1 - \alpha_2)x_1 + \alpha_2 x_2$$

$$\alpha_3 = \frac{y_{\max} - y_1}{y_2 - y_1} \qquad \alpha_2 = \frac{x_{\min} - x_1}{x_2 - x_1}$$

  - Compare α_3, α_2 without floating-point division

Line-Segment Clipping Assessment

- Cohen-Sutherland
  - Works well if many lines can be rejected early
  - Recursive structure (multiple subdivisions) is a drawback

- Liang-Barsky
  - Avoids recursive calls
  - Many cases to consider (tedious, but not expensive)
  - In general much faster than Cohen-Sutherland

Outline

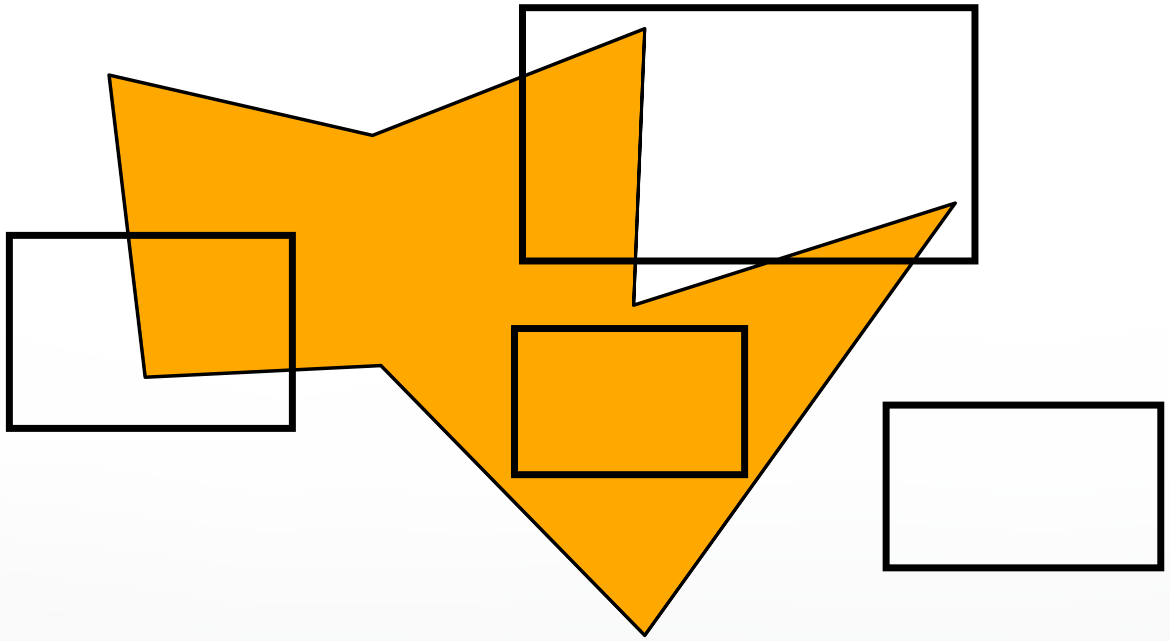
- Line-Segment Clipping
  - Cohen-Sutherland
  - Liang-Barsky

- Polygon Clipping
  - Sutherland-Hodgeman

- Clipping in Three Dimensions

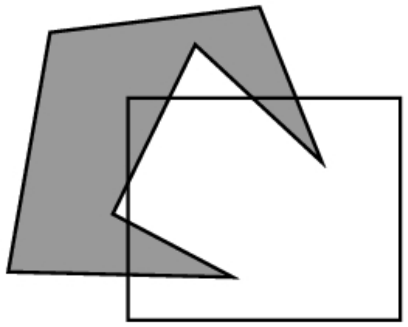
Polygon Clipping

- Convert a polygon into one or more polygons
- Their union is intersection with clip window
- Alternatively, we can first tesselate concave polygons (OpenGL supported)

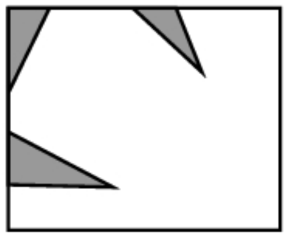


Concave Polygons

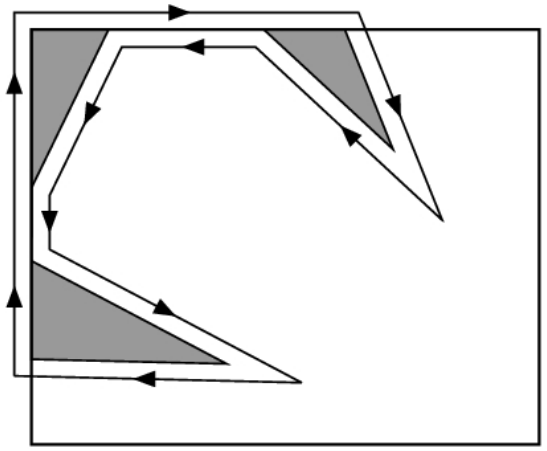
- Approach 1: clip, and then join pieces to a single polygon
  - often difficult to manage



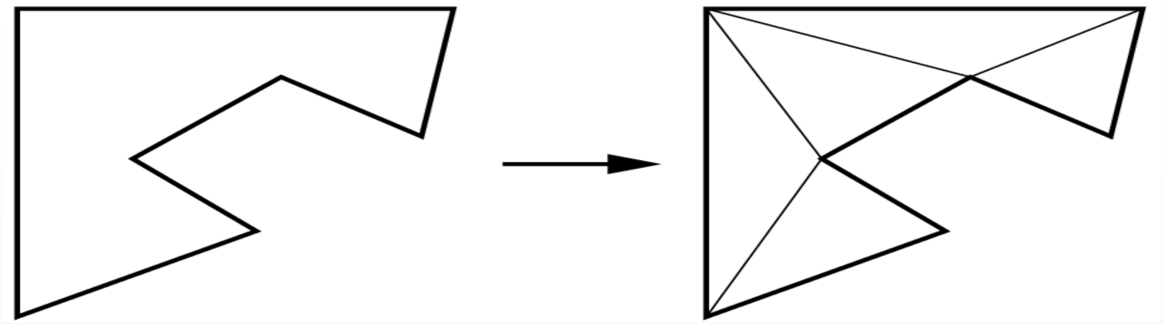
(a)



(b)

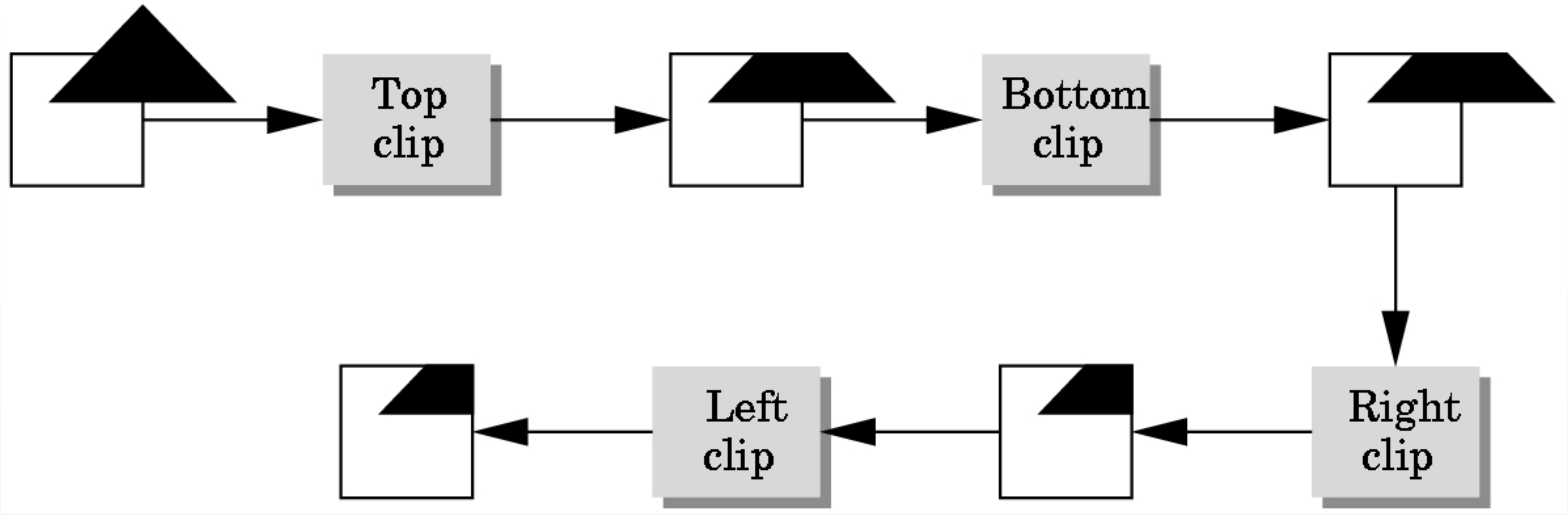


- Approach 2: tesselate and clip triangles
  - this is the common solution



Sutherland-Hodgeman (part 1)

- Subproblem:
  - Input: polygon (vertex list) and single clip plane
  - Output: new (clipped) polygon (vertex list)
- Apply once for each clip plane
  - 4 in two dimensions
  - 6 in three dimensions
  - Can arrange in pipeline

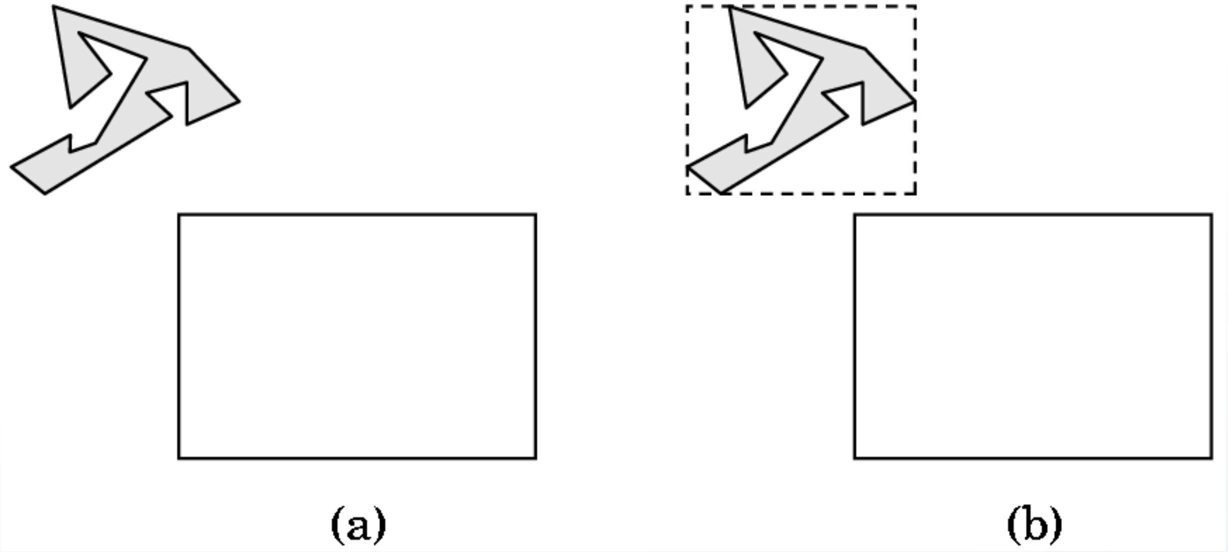


Sutherland-Hodgeman (part 2)

- To clip vertex list (polygon) against a half-plane:
  - Test first vertex. Output if inside, otherwise skip.
  - Then loop through list, testing transitions
    - In-to-in: output vertex
    - In-to-out: output intersection
    - out-to-in: output intersection and vertex
    - out-to-out: no output
  - Will output clipped polygon as vertex list
- May need some cleanup in concave case
- Can combine with Liang-Barsky idea

Other Cases and Optimizations

- Curves and surfaces
  - Do it analytically if possible
  - Otherwise, approximate curves / surfaces by lines and polygons
- Bounding boxes
  - Easy to calculate and maintain
  - Sometimes big savings



Outline

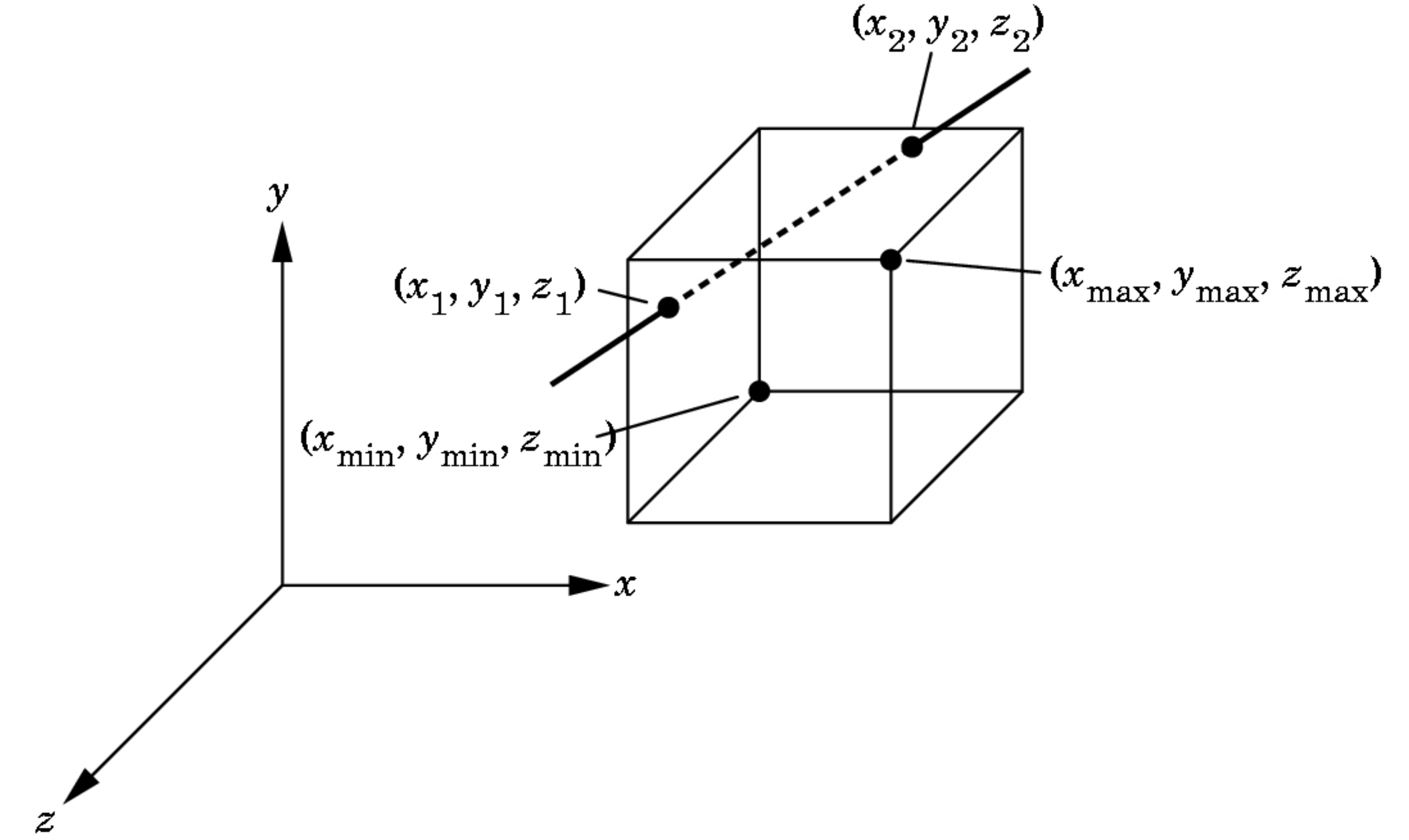
- Line-Segment Clipping
  - Cohen-Sutherland
  - Liang-Barsky

- Polygon Clipping
  - Sutherland-Hodgeman

- Clipping in Three Dimensions

Clipping Against Cube

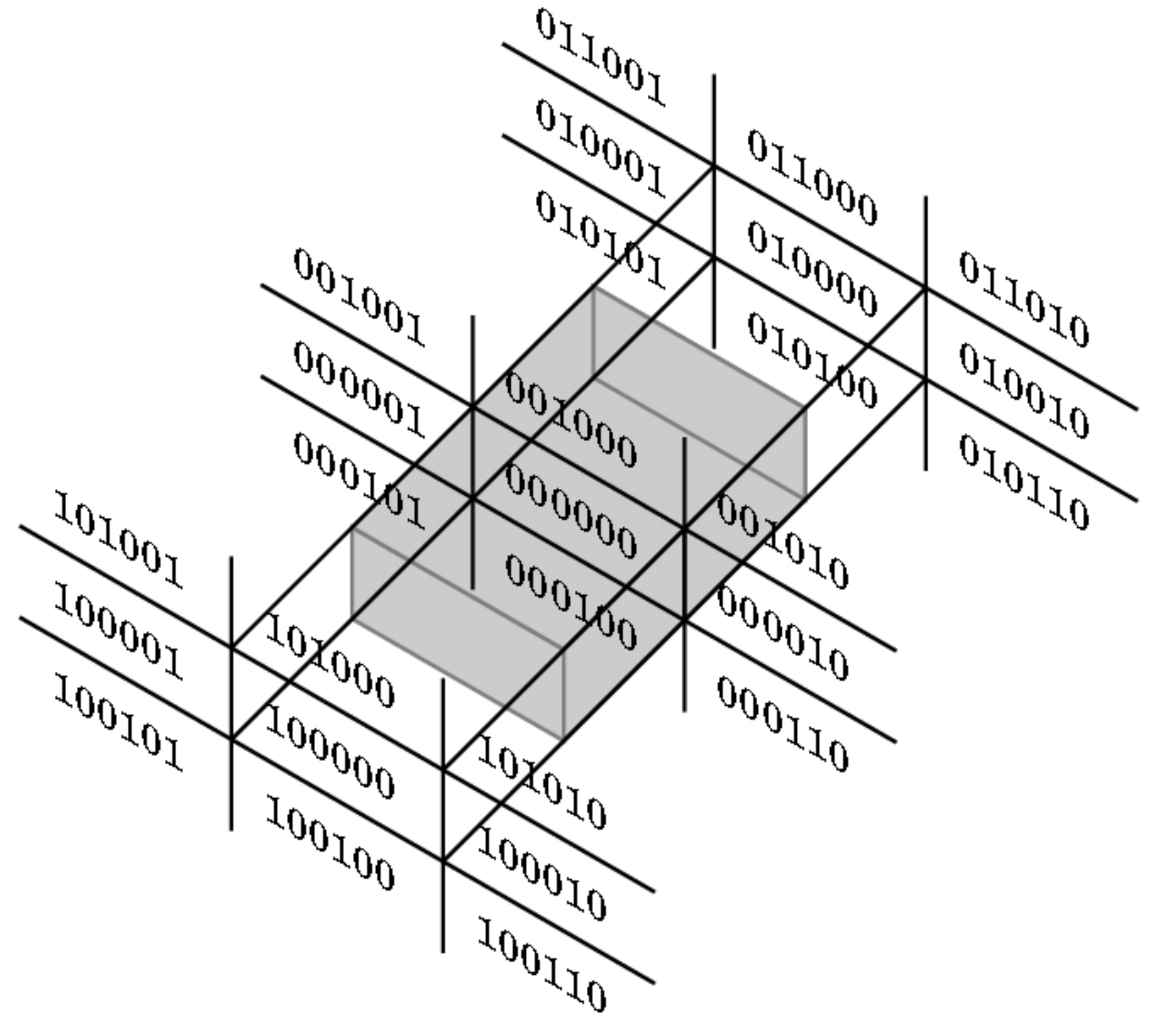
- Derived from earlier algorithms
- Can allow right parallelepiped



Cohen-Sutherland in 3D

- Use 6 bits in outcode
  - b_4: $z > z_{max}$
  - b_5: $z < z_{min}$

- Other calculations as before



Liang-Barsky in 3D

- Add equation $z(\alpha) = (1 - \alpha)z_1 + \alpha z_2$
- Solve, for $\mathbf{p}_0$ in plane and normal **n**:

$$p(\alpha) = (1 - \alpha)p_1 + \alpha p_2$$
$$n \cdot (p(\alpha) - p_0) = 0$$

- Yields

$$\alpha = \frac{n \cdot (p_0 - p_1)}{n \cdot (p_2 - p_1)}$$

- Optimizations as for Liang-Barsky in 2D

Summary: Clipping

- Clipping line segments to rectangle or cube
  - Avoid expensive multiplications and divisions
  - Cohen-Sutherland or Liang-Barsky
- Polygon clipping
  - Sutherland-Hodgeman pipeline
- Clipping in 3D
  - essentially extensions of 2D algorithms

Next Time

- Scan conversion
- Anti-aliasing
- Other pixel-level operations

http://cs420.hao-li.com

Thanks!