# Computer Graphics

## (Implementation of a Renderer)

Thilo Kielmann
Fall 2003
Vrije Universiteit, Amsterdam
kielmann@cs.vu.nl

http://www.cs.vu.nl/~graphics/



# Outline for today

- Tasks of a Renderer
- Implementing Transformations
- Clipping
- Hidden-Surface Removal
- Scan conversion of lines and polygons
- Antialiasing



# Major Tasks of a Renderer (1/2)

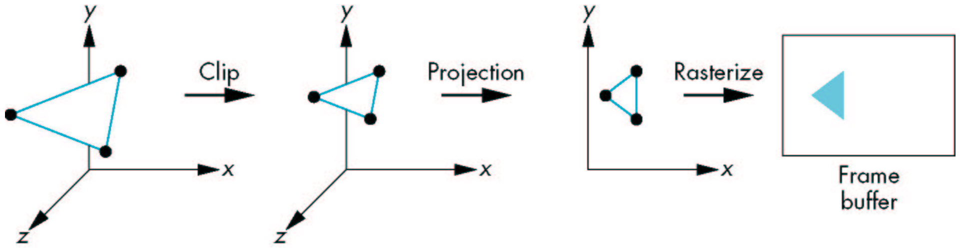

1. Modeling: Objects $\longrightarrow$ Vertices
   maybe clip away some objects
2. Geometric Processing:
   normalization, clipping, hidden-surface removal, shading



# Major Tasks of a Renderer (3/4)


3. Rasterization: 2D-objects $\longrightarrow$ pixels
4. pixels $\longrightarrow$ aliasing, problems displaying colors

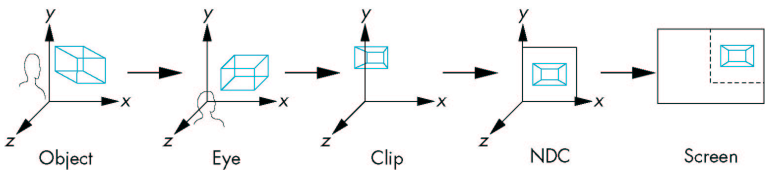
## Basic Implementation Strategies




1. Object-oriented:
   `for ( each_object ) render(object);`
2. Image-oriented:
   `for ( each_pixel ) assign_a_color(pixel);`



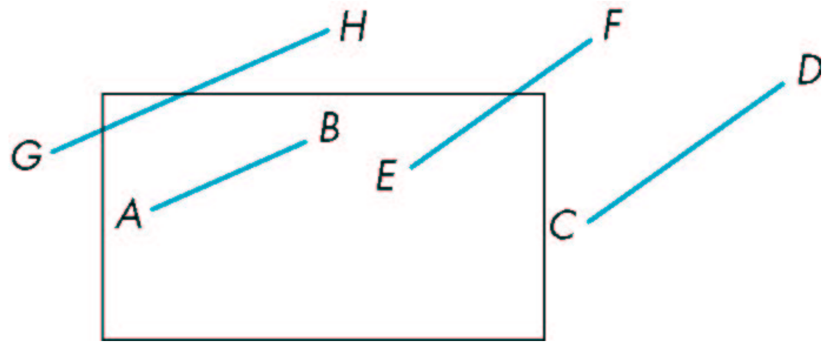
## Transformation Sequence




1. Object (world) coordinates
2. Eye (camera) coordinates
3. Clip coordinates
4. Normalized device coordinates: $x, y, z = \pm 1$
5. Window (screen) coordinates: $x_p = x, y_p = y, z_p = 0$
   (also take viewport into account, if needed)



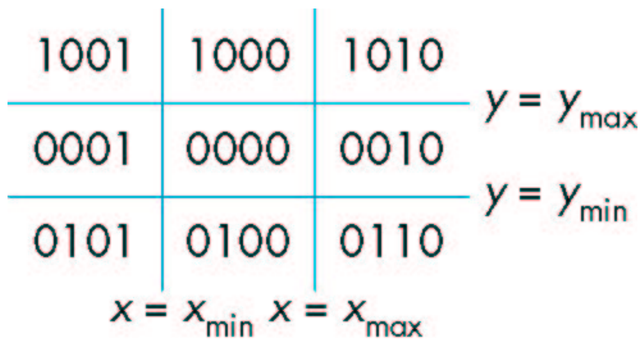
## Line-Segment Clipping (start with 2D)




Accept or reject line segments
Shorten line segments.



## Cohen-Sutherland Clipping



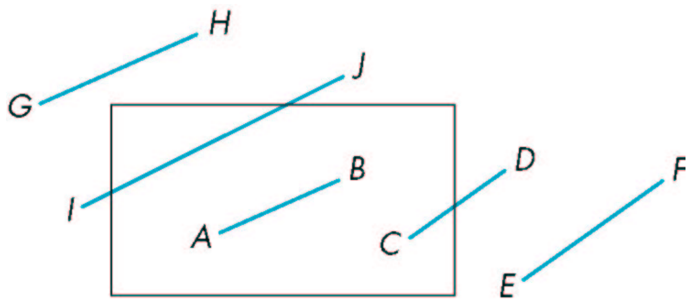

Outcodes for each point.
segment: $o_1 = outcode(x_1, y_1) \quad o_2 = outcode(x_2, y_2)$

# Cohen-Sutherland Clipping (2)

1. $(o_1 = o_2 = 0)$, both endpoints inside: rasterize
2. $(o_1 \neq 0, o_2 = 0)$ (or vice versa),
   one point inside, one outside:
   compute 1 or 2 intersections, shorten
3. $(o_1 \& o_2) \neq 0)$, both endpoints on same outside: discard
4. $(o_1 \& o_2 = 0)$, both endpoints on different outsides:
   compute all intersections



# Cohen-Sutherland Clipping (3)

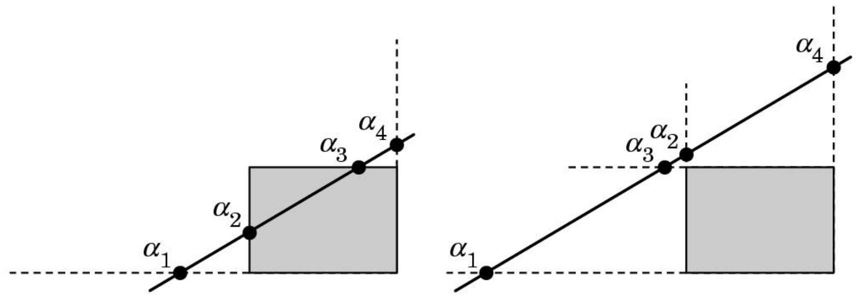



compute intersections using explicit form $y = mx + h$
needs special case for vertical lines
$\Longrightarrow$ parametric form for lines generally used
(also outside clipping)



# Liang-Barsky Clipping

$p(\alpha) = (1 - \alpha)p_1 + \alpha p_2$




**(a)** $1 > \alpha_4 > \alpha_3 > \alpha_2 > \alpha_1 > 0$

**(b)** $1 > \alpha_4 > \alpha_2 > \alpha_3 > \alpha_1 > 0$



# Liang-Barsky Clipping (2)

Example: clipping against $y_{max}$:

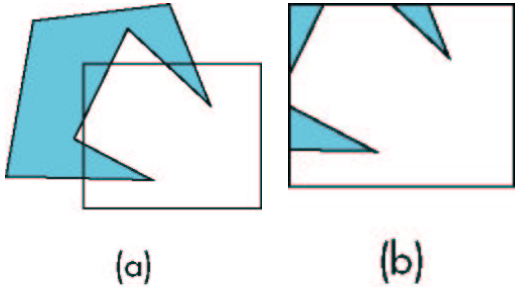
$\alpha = \frac{y_{max} - y_1}{y_2 - y_1}$

avoid floating-point division:
$\alpha(y_2 - y_1) = \alpha \Delta y = y_{max} - y_1 = \Delta y_{max}$

express all tests based on $\Delta y_{max}$ and $\Delta y$

# Polygon Clipping

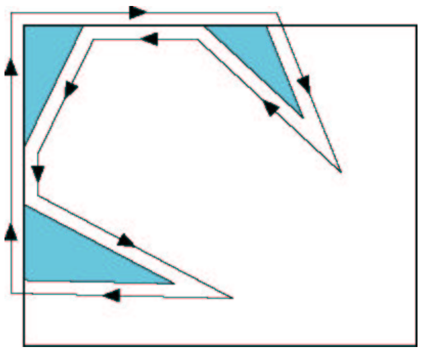
Problem: concave polygons




Clipping line-by line, problem with concave polygons may increase the number of polygons (problem in the pipeline)



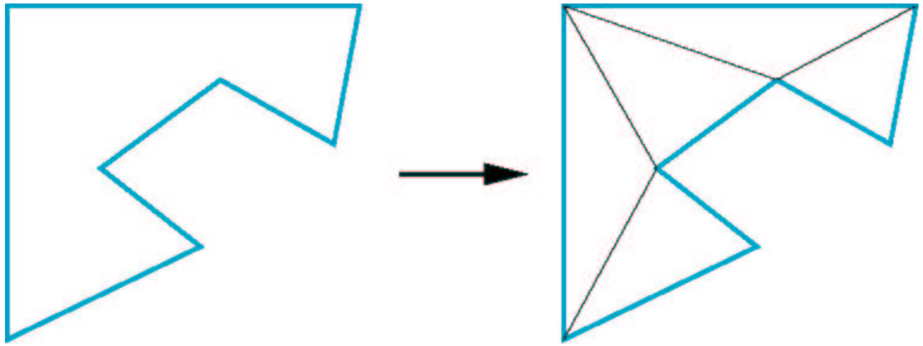
# Concave Polygons (2)



Creating a single polygon may create other problems later (for example when displaying colored lines)



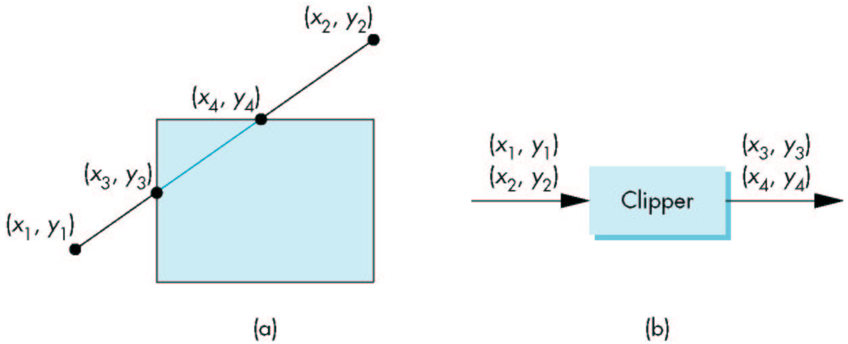
# Tesselation of Concave Polygons



Either forbid or tesselate concave polygons
OpenGL: GLU library does tesselation

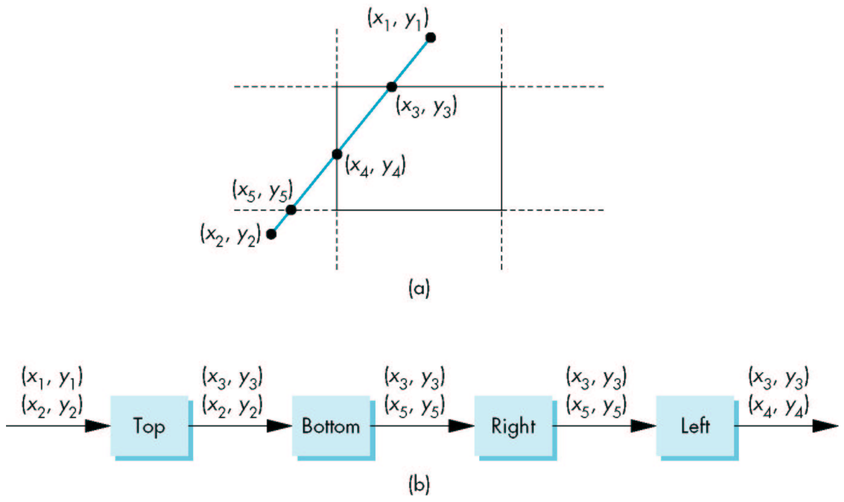


# Clipper as a Black Box



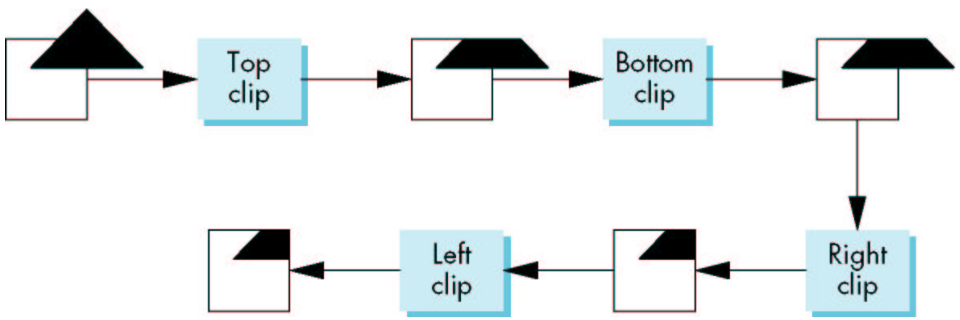

Enables embedding in pipeline architecture.

# Pipeline Clipping



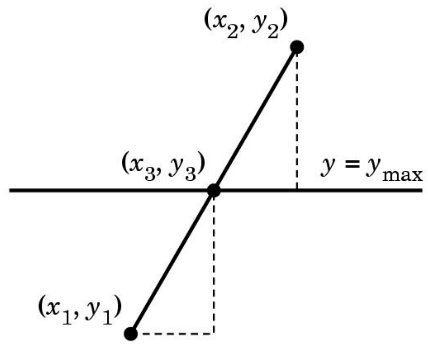



# Example of Pipeline Clipping






# Example: Clipping at the Top

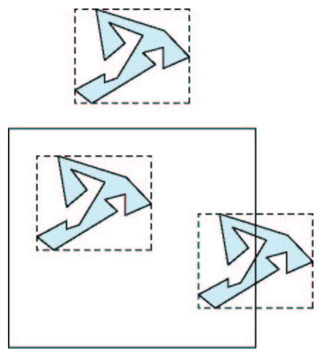



Using similar triangles:

$x_3 = x_1 + (y_{max} - y_1)\frac{x_2 - x_1}{y_2 - y_1} \qquad y_3 = y_{max}$

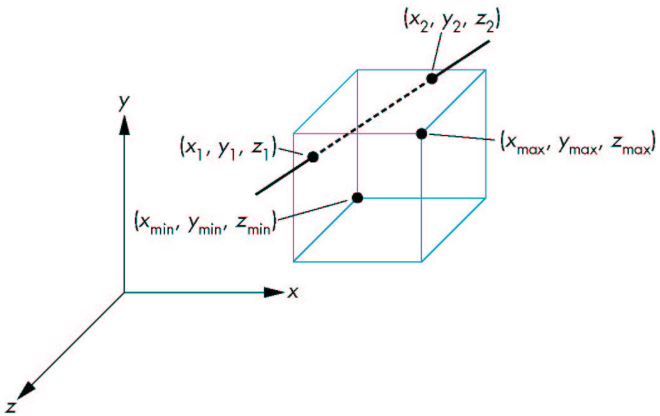


# Clipping with Bounding Boxes



bounding boxes are often kept automatically with objects

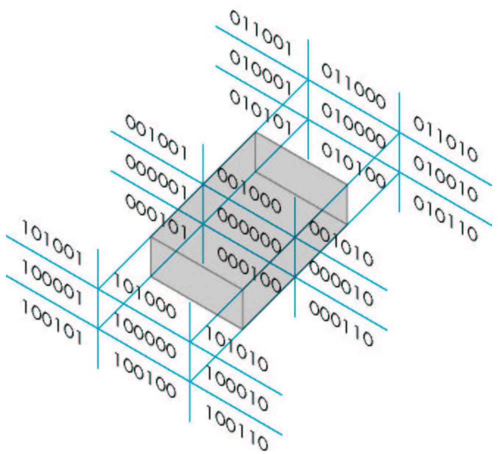
# Clipping in 3D




just use parallelepiped instead of the clipping rectangle



# Cohen-Sutherland in 3D






# Liang-Barsky in 3D

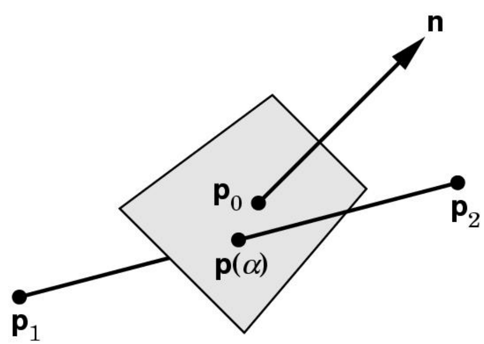
just extend the points to three dimensions

and compute 6 intersections instead of 4



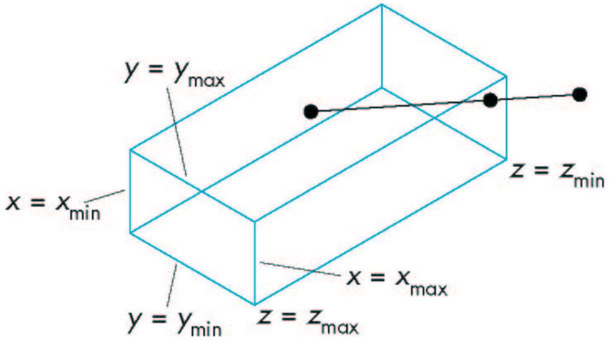
# Plane-Line Intersection

In 3D: clip lines against planes or planes against planes




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

# Clipping for Orthographic Viewing

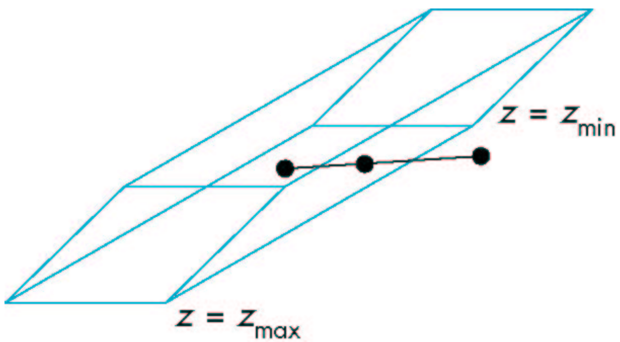



General case needs 6 multiplications and 1 division

With orthographic viewing, each intersection needs 1 division



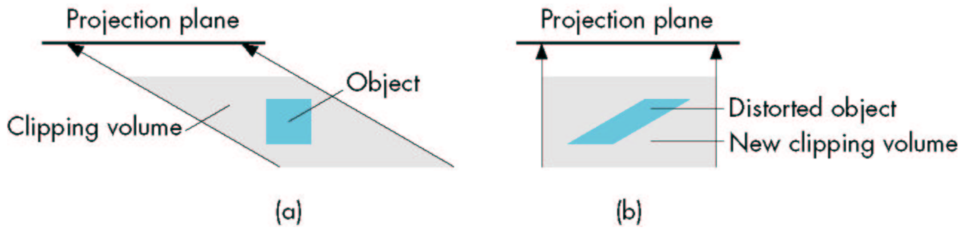
# Clipping for Oblique View




We need the full computation again. **Do we?**



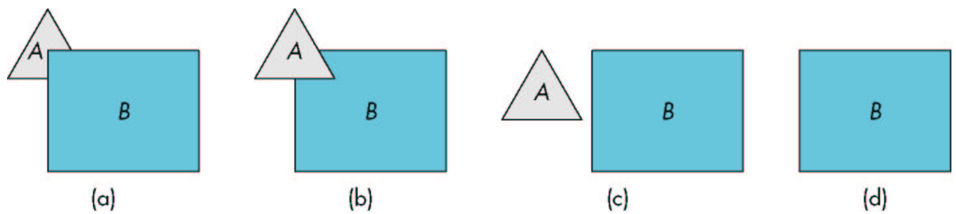
# Remember: Pre-Distortion of Objects




. . . also pays off for perspective projections



# Hidden-Surface Removal

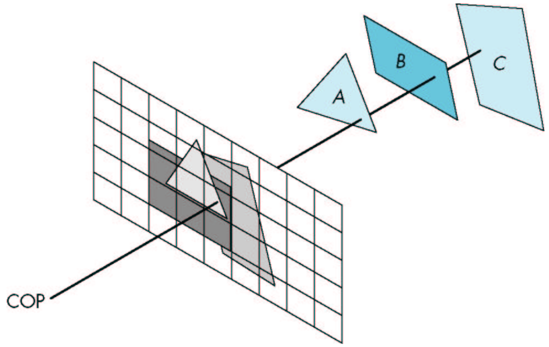



**Object-space approach:**

with $k$ objects, compare each with $k-1$ objects:

$O(k^2)$ checks

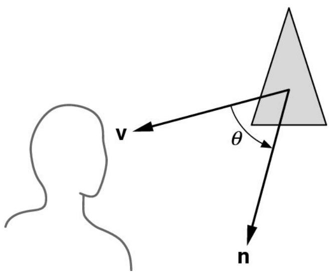
# Hidden-Surface: Image-Space Approach




Test $n \times m$ rays, $O(k)$ checks



# Back-Face Removal: "Culling"




Eliminate back-facing (invisible) polygons first

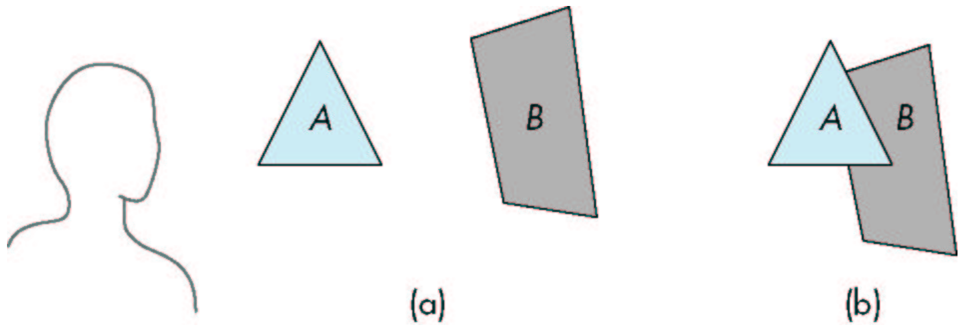
Test: $-90 \leq \theta \leq 90 \qquad \cos\theta \geq 0 \qquad n \cdot v \geq 0$

In OpenGL:

```
glEnable(GL_CULL_FACE)
glCullFace( GL_FRONT / GL_BACK / GL_FRONT_AND_BACK )
```



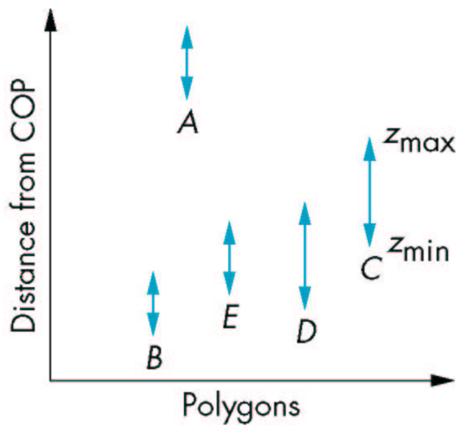
# Depth Sort / Painter's Algorithm




1. sort all polygons by depth
2. draw all polygons, beginning with largest depth

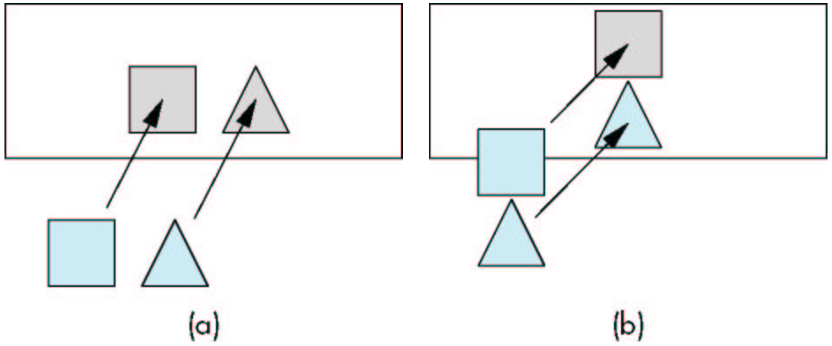


# Sorting Polygons by $z$-Extent




Render $A$ first, but the others?

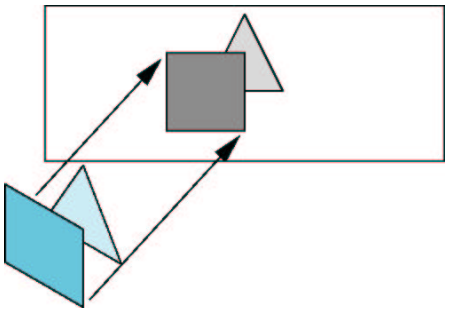
## Check for Overlap in $x$ and $y$




If either $x$ or $y$ do not overlap, we can render in any order.



## Further test with Overlap

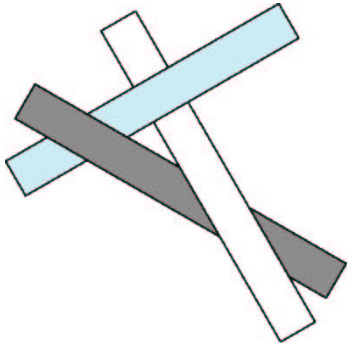


Test if one polygon completely is on one side of the plane defined by the other one.

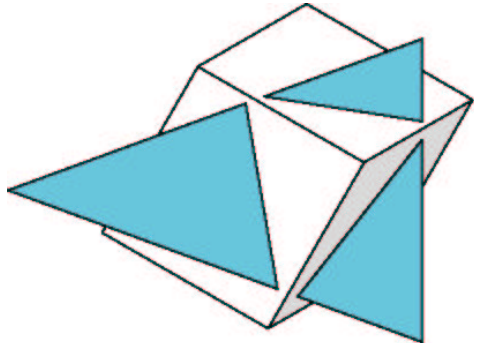
Only works with parallel view – advantage of NDC after perspective normalization!



## Problems. . .



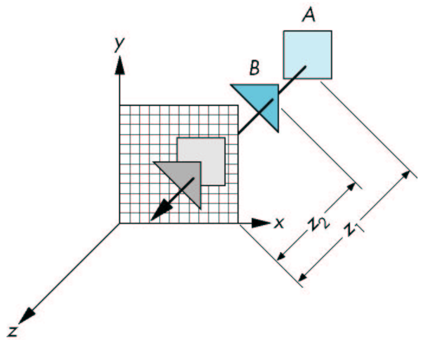
cyclic overlap



piercing polygons



## The $z$-Buffer Algorithm (used in OpenGL)




Keep depth information in $z$-Buffer in same size as frame buffer and resolution of depth information, e.g. `float`.

# The $z$-Buffer Algorithm (2)

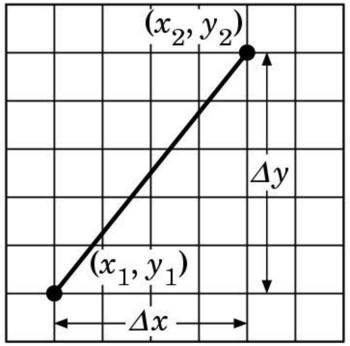
1. Set all pixels to background color
2. Init all $z$-Buffer entries to maximum depth
3. For each object, for each pixel

   if depth(pixel(object)) $<$ depth in $z$-Buffer:
   draw new pixel, update $z$-Buffer

Caution:
user has to reset the depth information for new image!



# Scan Conversion

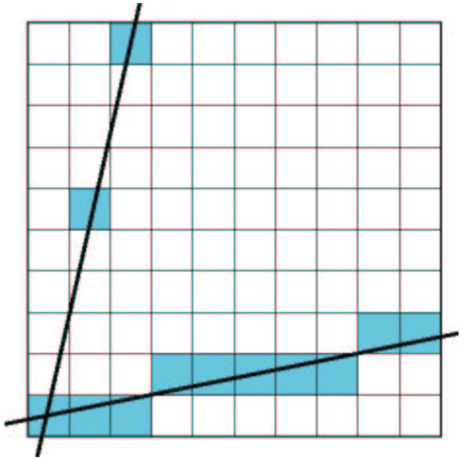



$$y = mx + h$$
$$m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x}$$

Setting pixels according to line segments.
assume: `write_pixel(int x, int y, int value)`
pixels at middle of screen coordinates, e.g. (42.5,30.5)



# DDA Algorithm

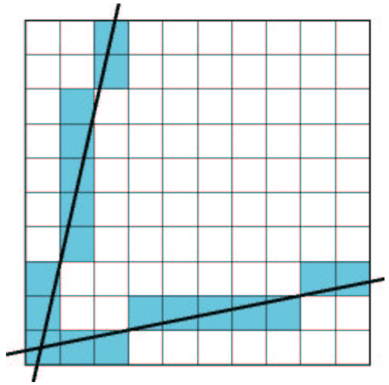


```
for ( x=x1, x <= x2, x++){
   y += m;
   write_pixel(x, round(y),
               color);
}
```

(DDA = Digital Differential Analyzer)

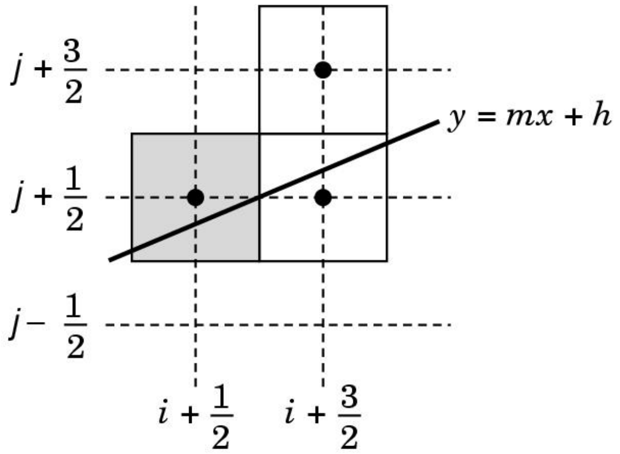


# Select the Loop on Slope



$m \leq 1 \rightarrow$ loop over x
$m > 1 \rightarrow$ loop over y

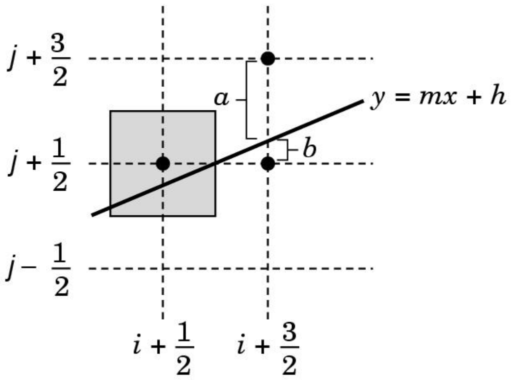
# Bresenham's Algorithm




Incrementally: choose the closest pixel ($0 \leq m \leq 1$ !!!)



# Bresenham: Decision Variable




$d = a - b$, sign of $d$ makes the decision between the two possible pixels



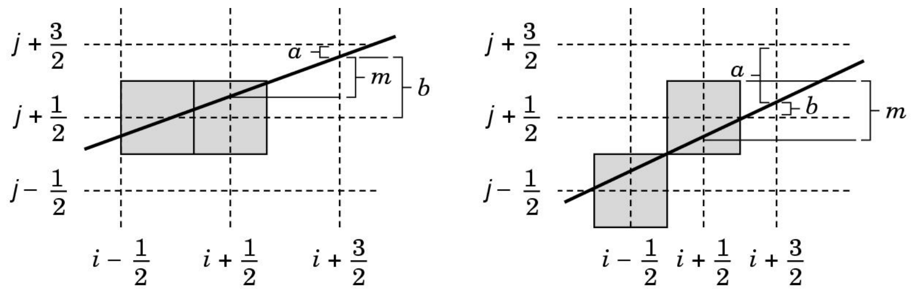
# Making Bresenham's Algorithm Efficient

Computation of $a$ and $b$ requires floating-point arithmetic.

Make it integer (redefine $d$):
$d = (x_2 - x_1)(a - b) = \Delta x(a - b)$
"stretch" the values to integers, we only need the sign!



# Bresenham Incrementally




$$d_{k+1} = d_k + \begin{cases} 2\Delta y & \text{if} d_k > 0; \\ 2(\Delta y - \Delta x) & \text{otherwise} \end{cases}$$

Just a sign test and an addition per pixel!

# Polygon Filling

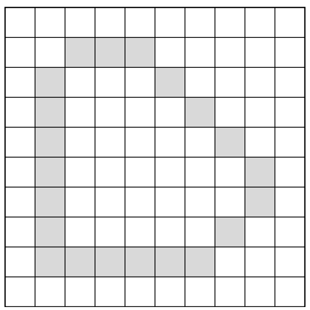


**Odd-even test:**

$p$ is inside the polygon if a ray from $p$ to infinity passes an *odd* number of line segments



# Flood Fill

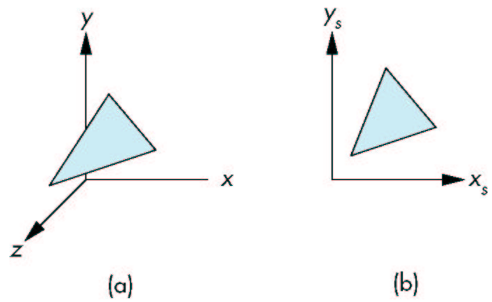


```
flood_fill ( int x, int y ){
   if ( read_pixel(x,y) == WHITE ){
      write_pixel(x,y,BLACK);
      flood_fill(x-1,y);
      flood_fill(x+1,y);
      flood_fill(x,y-1);
      flood_fill(x,y+1);
   }
}
```

Draw the borders of a polygon (with Bresenham),
then find a *seed* point inside,
and color all neighbors until we reach the borders.



# Scan Conversion with the $z$-Buffer

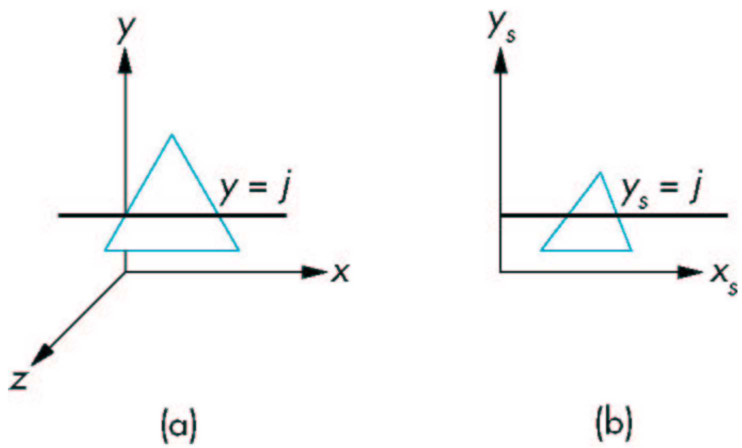



Left: Normalized device coordinates (including depth)
Right: Projection to screen coordinates

Idea: Combine scan conversion, hidden-surface removal, texture, and shading

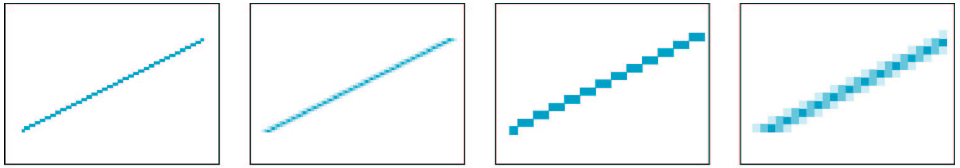


# Scan Conversion with the $z$-Buffer (2)




Use back-projection to NDC to decide the pixel's color.

# Antialiasing

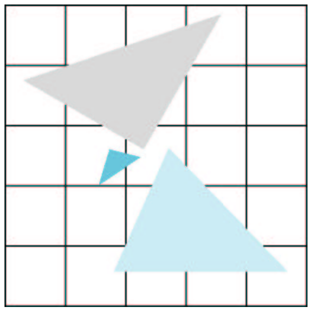


(a) (b) (c) (d)

Color the pixels according to the % of overlap with the idealized line



# Polygons Sharing a Pixel



Pixel color depends on display order of the polygons.
Or: do color averaging (blending, see Chapter 7)



# Display Issues

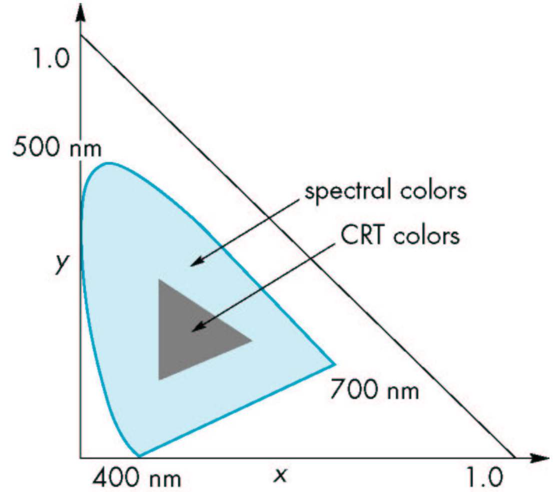
- Graphics APIs focus on **device independence**
- We normally care about writing images fast enough into the frame buffer – and the graphics adaptor transfers this onto the screen.
- But: different displays (monitors etc.) do have different properties. (e.g., different pixel size with the same resolution)



# Color Systems

- RGB color is designed following how color is produced
  . . . but not following how color is **perceived**
- Example: a yellow color $(0.8, 0.6, 0.0)$
  with two different devices, the color may look different although both times 80% red and 60% green and 0%blue gets mixed
- This might be partially compensated by a conversion matrix:
  $C_2 = M \cdot C_1$, with $C_1$ and $C_2$ being RGB colors.

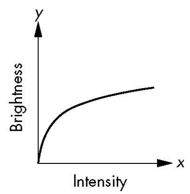
# Visible Colors






# Gamma Correction

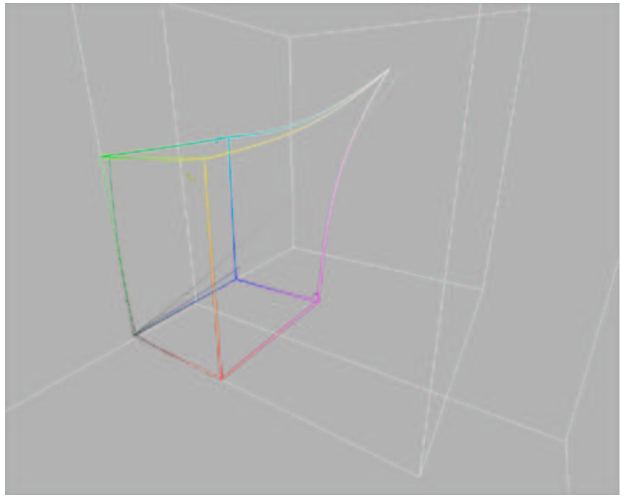
The intensity of a CRT is related to the voltage $V$ as in:
$logI = c_0 + \gamma \log V$ ($c_0$ and $\gamma$ are CRT properties)




The **perceived** brightness goes logarithmic with the intensity.
This effect might be compensated, e.g. with a lookup table.

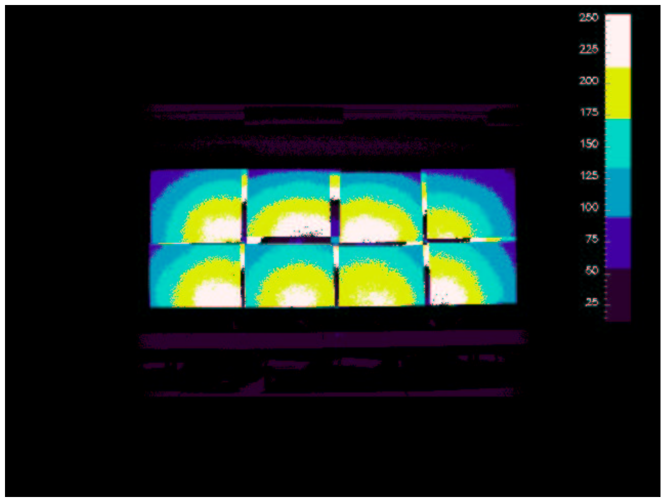


# Color Cube of the ICWall Beamers





# White Distribution of the ICWall Beamers



# Summary

Tasks of a Renderer:

- Transformations
- Clipping
- Hidden-Surface Removal
- Scan Conversion
- Device issues (antialiasing, colors)
- Next week: Curves and Surfaces