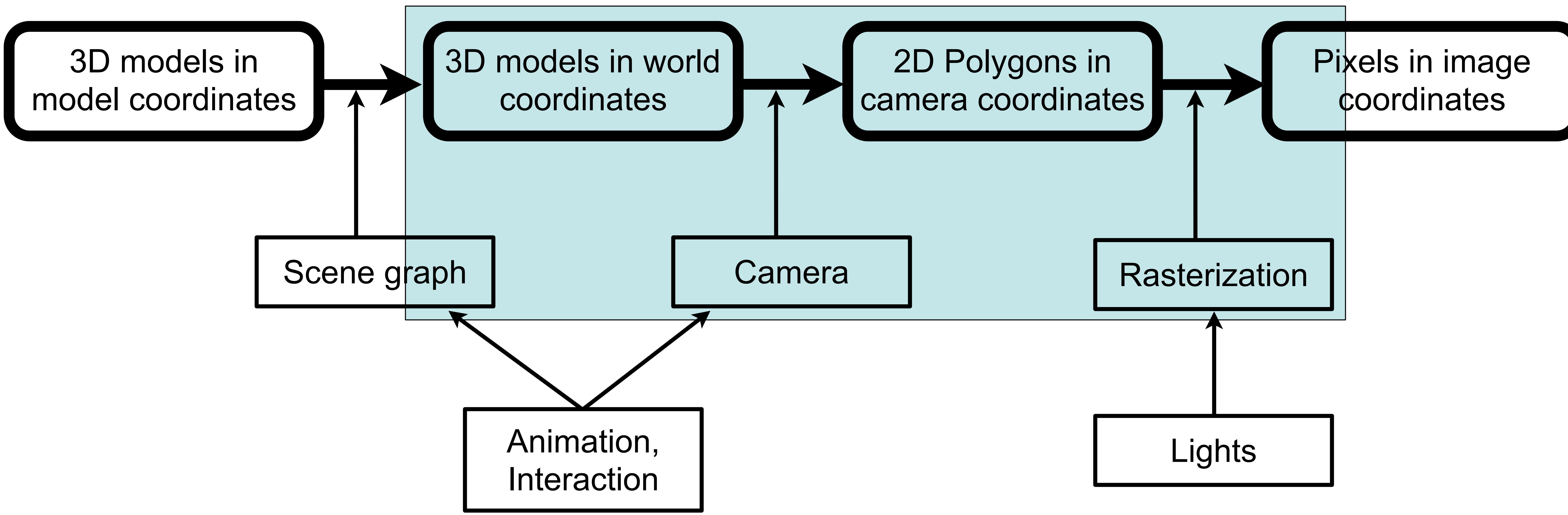
Computer Graphics 1

Chapter 3 (May 26th, 2011, 2-4pm):
The 3D camera and associated problems

The 3D rendering pipeline (our version for this class)

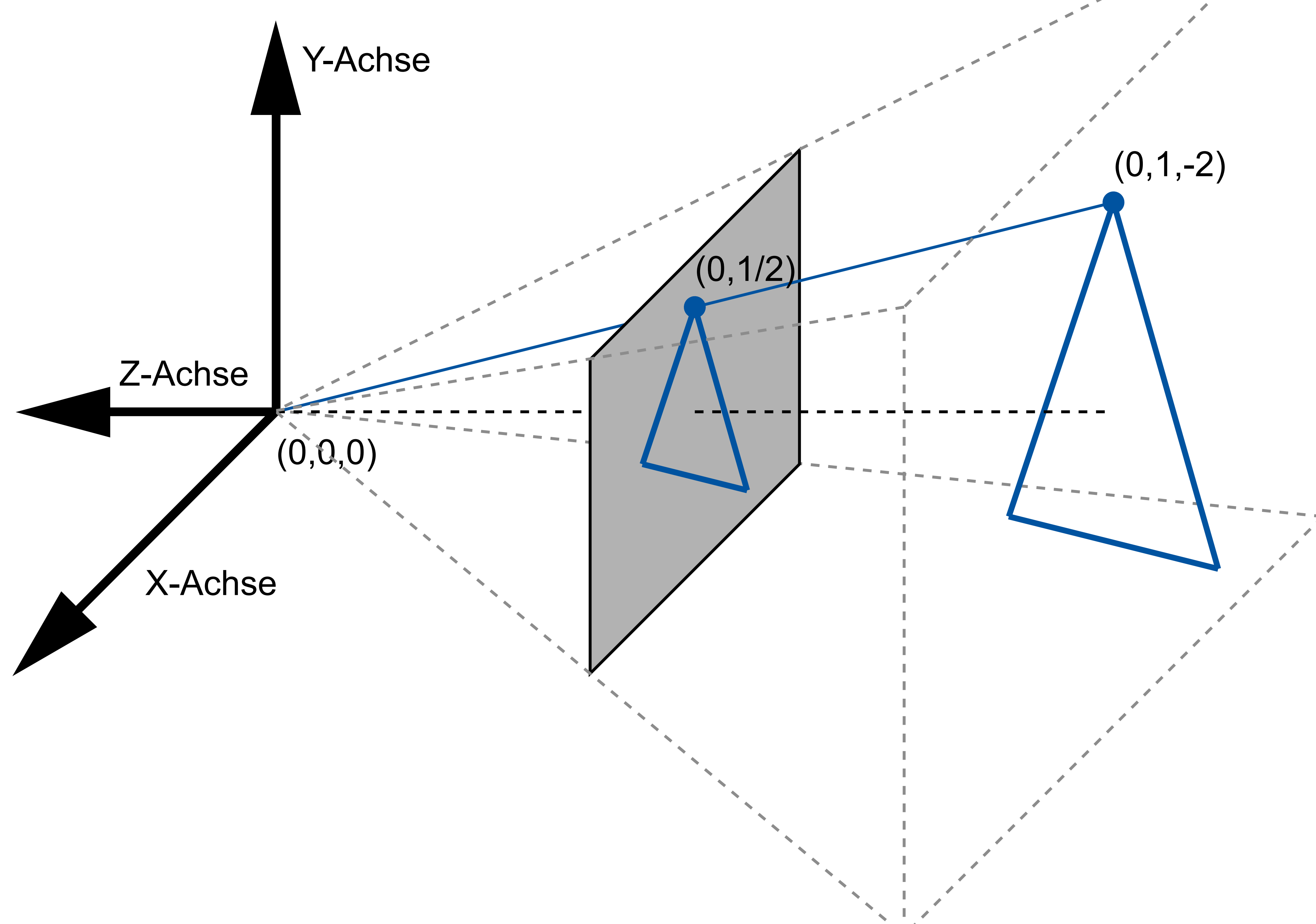


Chapter 3 - The 3D camera and associated problems

- 3D camera model
- Optimizations for the camera
- How to deal with Occlusion
- Rasterization
  - clipping
  - drawing lines
  - filling areas

The mathematical camera model

- Perspective projection
- The Camera looks along the **negative Z axis**
- Image plane at Z=-1
- 2D image coordinates
  - $-1 < x < 1$,
  - $-1 < y < 1$
- Two steps
  - projection matrix
  - perspective division



Projection Matrix (one possibility)

- X and Y remain unchanged
- Z is preserved as well
- 4th (homogeneous) coordinate w != 1

$$\begin{pmatrix} x_{sicht} \\ y_{sicht} \\ z_{sicht} \\ w_{sicht} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ -z \end{pmatrix}$$

- Transformation from world coordinates into view coordinates
- This means that this is not a regular 3D point
  - otherwise the 4th component w would be = 1

- View coordinates are helpful for culling (see later)

Perspective Division

$$\begin{pmatrix} x_{bild} \\ y_{bild} \\ z_{bild} \\ w_{bild} \end{pmatrix} = \frac{1}{w_{sicht}} \begin{pmatrix} x_{sicht} \\ y_{sicht} \\ z_{sicht} \\ w_{sicht} \end{pmatrix} = \begin{pmatrix} x_{sicht} / w_{sicht} \\ y_{sicht} / w_{sicht} \\ z_{sicht} / w_{sicht} \\ w_{sicht} / w_{sicht} \end{pmatrix} = \begin{pmatrix} x / -z \\ y / -z \\ -1 \\ 1 \end{pmatrix}$$

- Divide each point by its 4th coordinate w

- Transformation from view coordinates into image coordinates

- since w = -z and we are looking along the negative Z axis, we are dividing by a positive value
- hence the sign of X and Y remain unchanged
- points further away (larger absolute Z value) will have smaller x and y
  - this means that distant things are smaller
  - points on the optical axis will remain in the middle of the image

Controlling the Camera

- So far we can only look along negative Z
- Other camera positions and orientations:
  - Let C be the transformation matrix that describes the camera‘s position and orientation in world coordinates
  - C is composed from a translation and a rotation, hence can be inverted
  - transform the entire world by C^{-1} and apply the camera we know ;-)

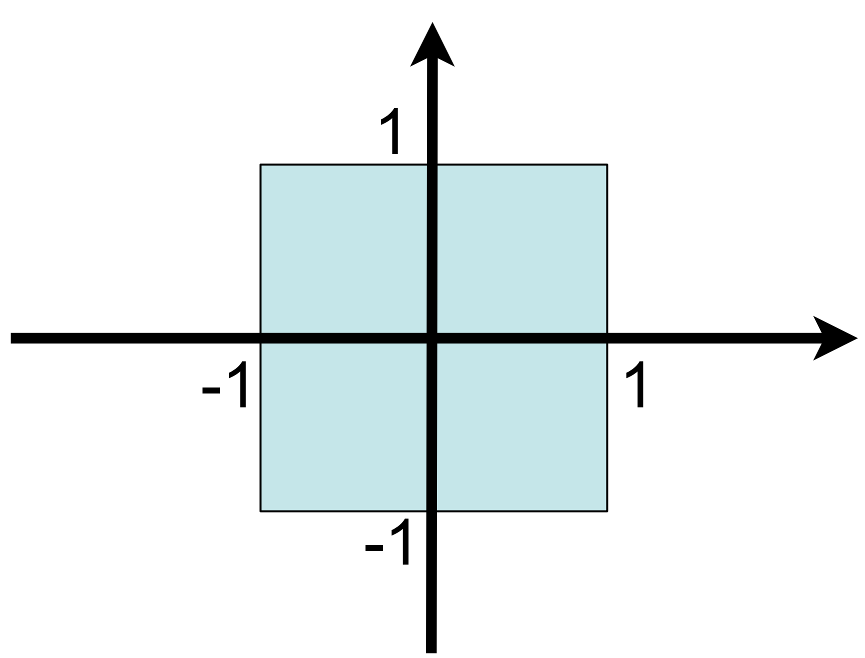
- Other camera view angles?
- If we adjust this coefficient
  - scaling factor will be different
  - larger abs value means __________ angle.
  - could also be done in the division step

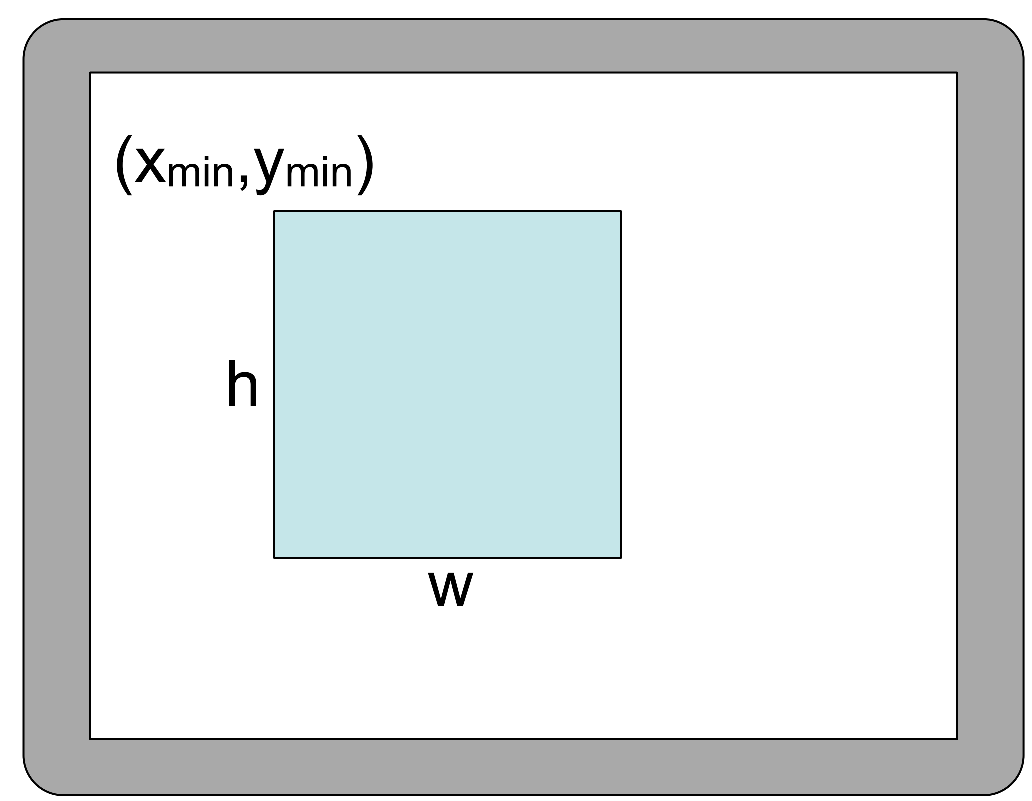
$$\begin{pmatrix} x_{sicht} \\ y_{sicht} \\ z_{sicht} \\ w_{sicht} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ -z \end{pmatrix}$$

From image to screen coordinates

- Camera takes us from world via view to image coordinates
- $-1 < x_{image} < 1$, $-1 < y_{image} < 1$

- In order to display an image we need to go to screen coordinates
  - assume we render an image of size (w,h) at position (x_{min}, y_{min})
  - then $x_{screen} = x_{min} + w(1+x_{image})/2$, $y_{screen} = y_{min} + h(1-y_{image})/2$





Chapter 3 - The 3D camera and associated problems

- 3D camera model
- Optimizations for the camera
- How to deal with Occlusion
- Rasterization
  - clipping
  - drawing lines
  - filling areas

Optimizations in the camera: Culling

- view frustum culling
- back face culling
- occlusion culling

http://en.wikipedia.org/wiki/File:At_the_drafting_race_from_The_Powerhouse_Museum_Collection.jpg

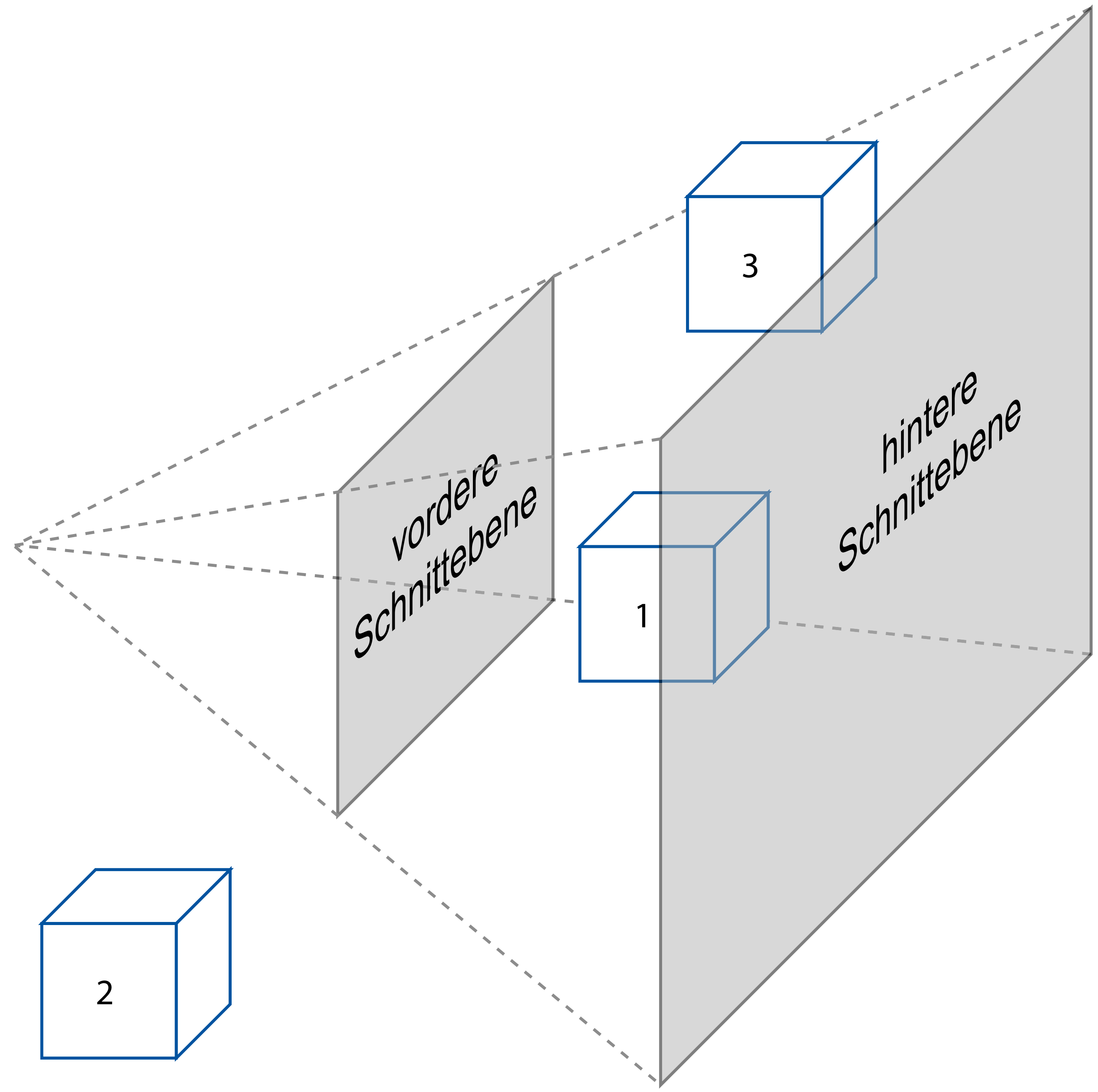
View Frustum culling

- Goal: Just render objects within the viewing volume (aka view frustum)
- Need an easy test for this...

- Z-Axis: bewteen 2 clipping planes
- $z_{near} > z_{view} > z_{far}$ (remember: negative z)

- X- and Y-Axis: inside the viewing cone
- $-w_{view} < x_{view} < w_{view}$
- $-w_{view} < y_{view} < w_{view}$

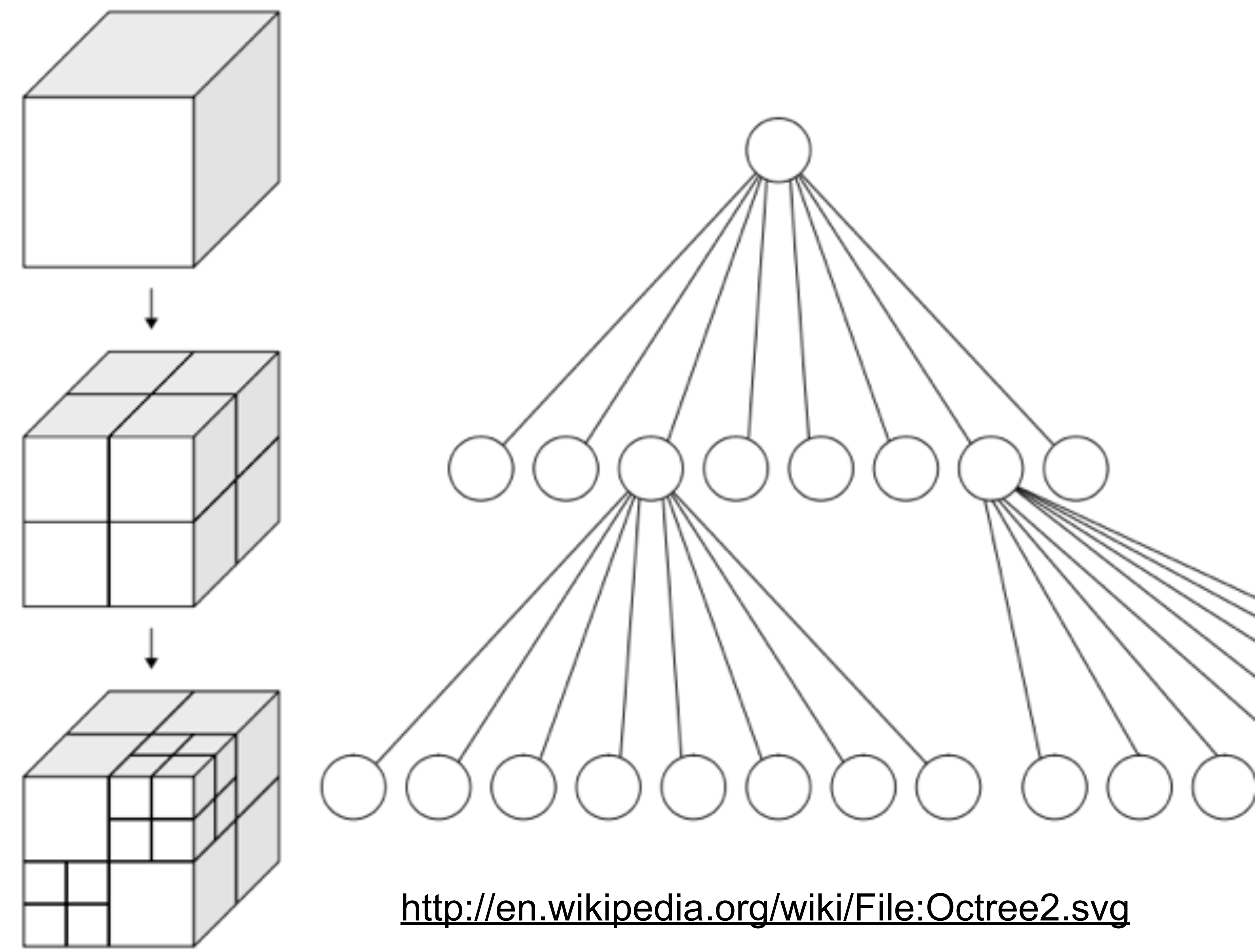
- Two simple comparisons for each axis!



$$\begin{pmatrix} x_{bild} \\ y_{bild} \\ z_{bild} \\ w_{bild} \end{pmatrix} = \frac{1}{w_{sicht}} \begin{pmatrix} x_{sicht} \\ y_{sicht} \\ z_{sicht} \\ w_{sicht} \end{pmatrix} = \begin{pmatrix} x_{sicht} / w_{sicht} \\ y_{sicht} / w_{sicht} \\ z_{sicht} / w_{sicht} \\ w_{sicht} / w_{sicht} \end{pmatrix} = \begin{pmatrix} x / -z \\ y / -z \\ -1 \\ 1 \end{pmatrix}$$

Octrees speed up View Frustum Culling

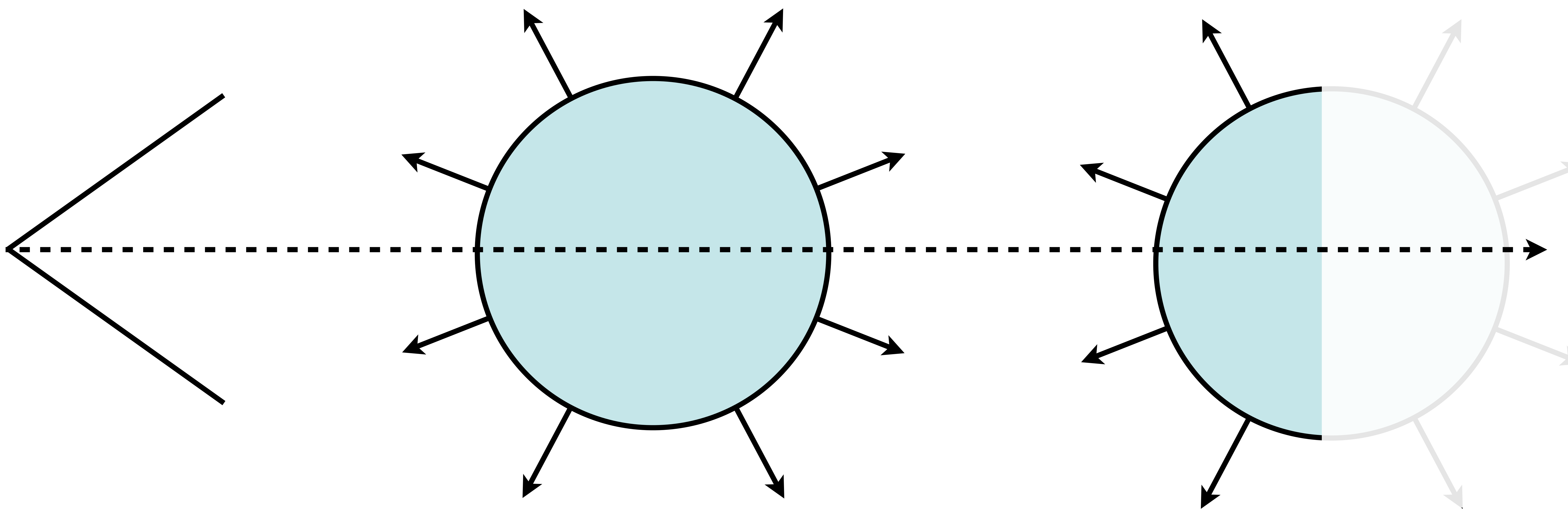
- Naive frustum culling needs O(n) tests
  - where n = number of objects
- Divide entire space into 8 cubes
  - see which objects are inside each
- subdivide each cube again
  - do recursively until cube contains less than k objects
- Instead of culling objects, cull cubes
- Needs O(log n) tests



http://en.wikipedia.org/wiki/File:Octree2.svg

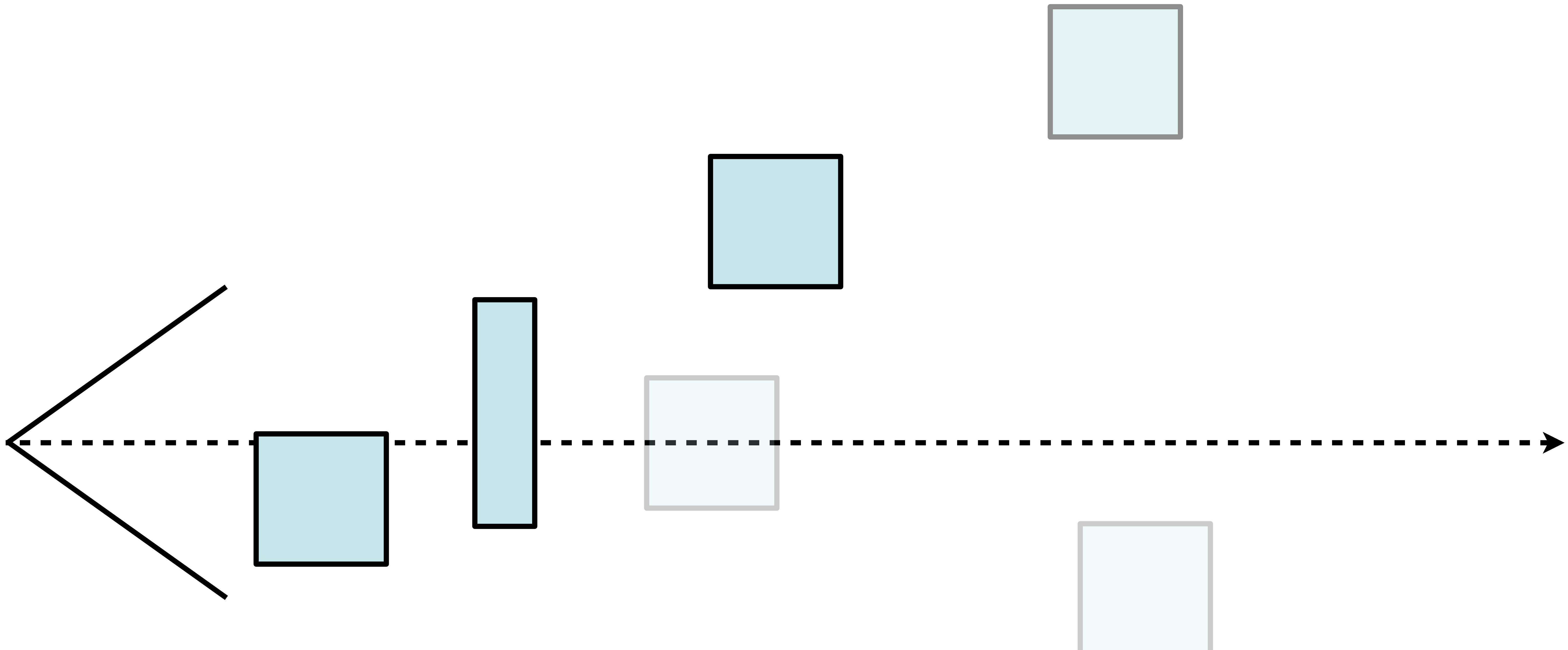
Back-face culling

- Idea: polygons on the back side of objects don‘t need to be drawn
- Polygons on the back side of objects face backwards
- Use the Polygon normal to check for orientation
  - if angle < 90° between optical axis and normal then polygon is facing backwards
  - if angle < 90° => cos(angle) > 0 => scalar product > 1 for vectors of length 1



Occlusion culling

- Idea: objects that are hidden behind others don‘t need to be drawn
- efficient algorithm using an occlusion buffer, similar to a Z-buffer



Chapter 3 - The 3D camera and associated problems

- 3D camera model
- Optimizations for the camera
- How to deal with Occlusion
- Rasterization
  - clipping
  - drawing lines
  - filling areas

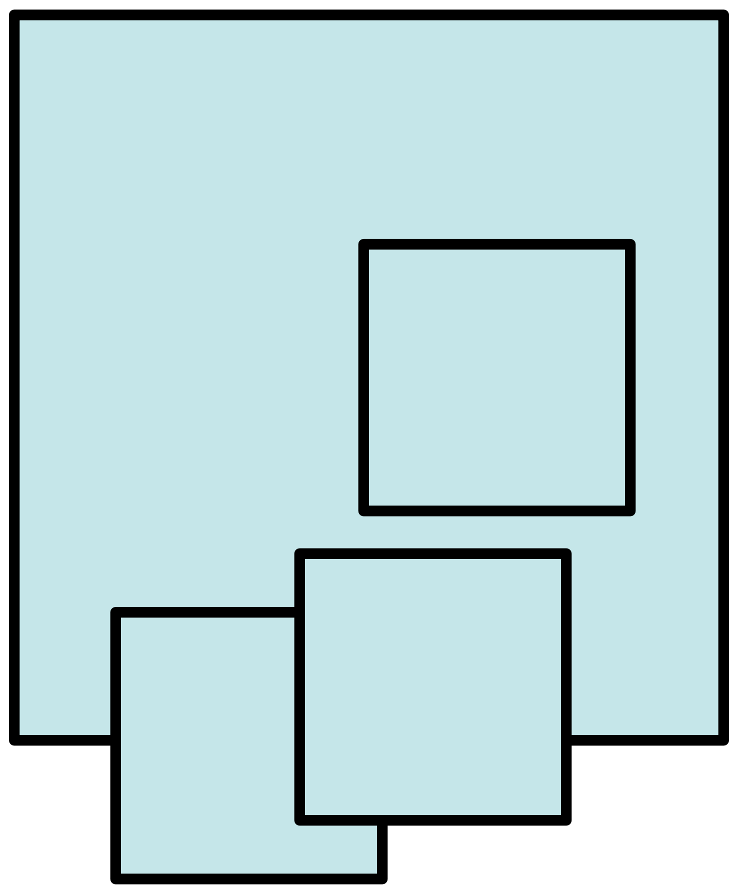
Occlusion: The problem space in general

- Need to determine which objects occlude which others
- want to draw only the frontmost (parts of) objects

- Culling worked at the object level, now look at the polygons

- More general: draw the frontmost polygons
  - ..or maybe parts of polygons?

- Occlusion is an important depth cue for humans
  - need to get this really correct!



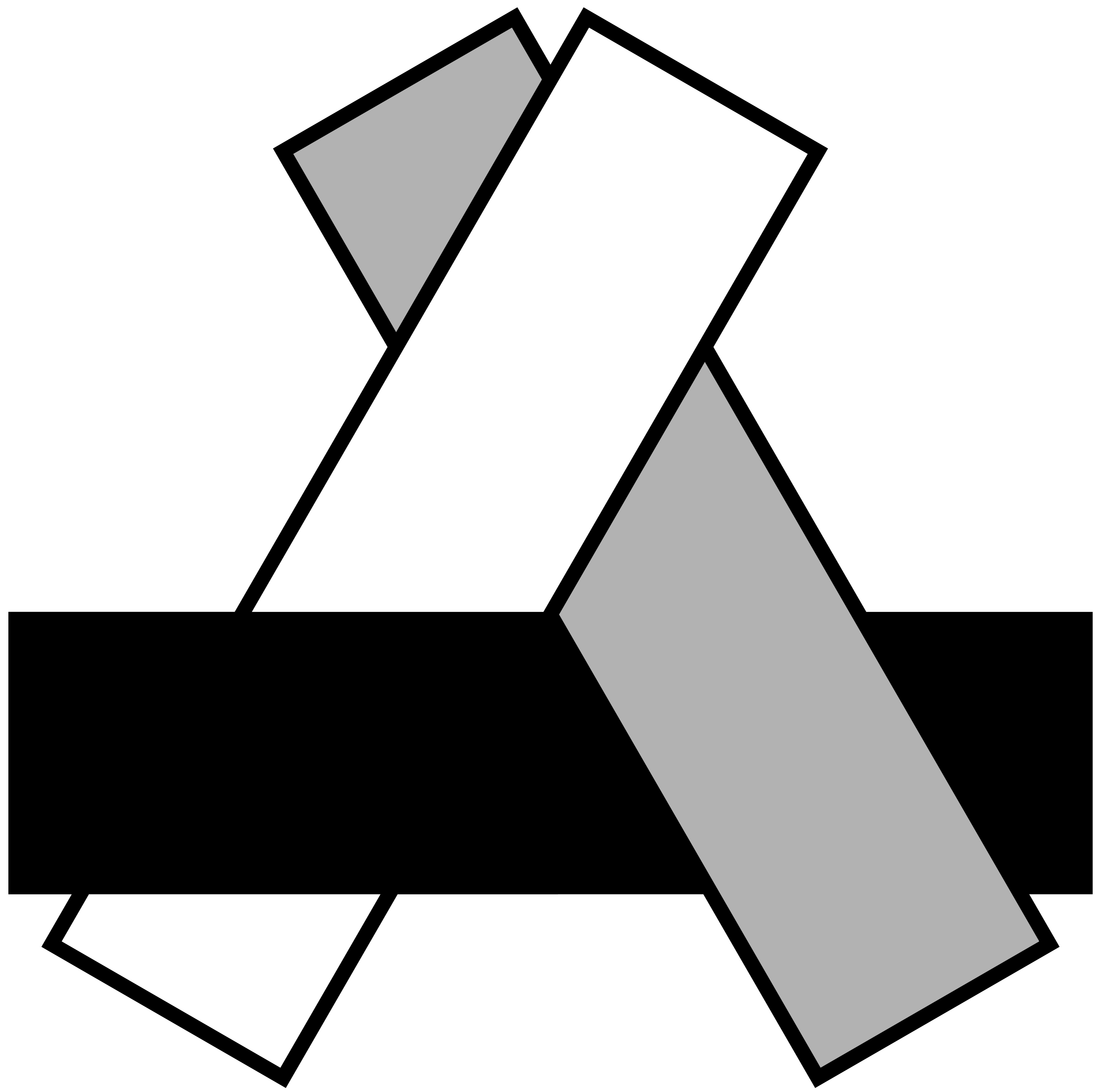
Occlusion: depth-sort

- Regularly used in 2D vector graphics

- Sort polygons according to their z position in view coordinates
- Draw all polygons from back to front
- Back polygons will be overdrawn
- Front polygons will remain visible

- Problem 1: self-occlusion
  - not a problem with triangles ;-)

- Problem 2: circular occlusion
  - think of a pin wheel!

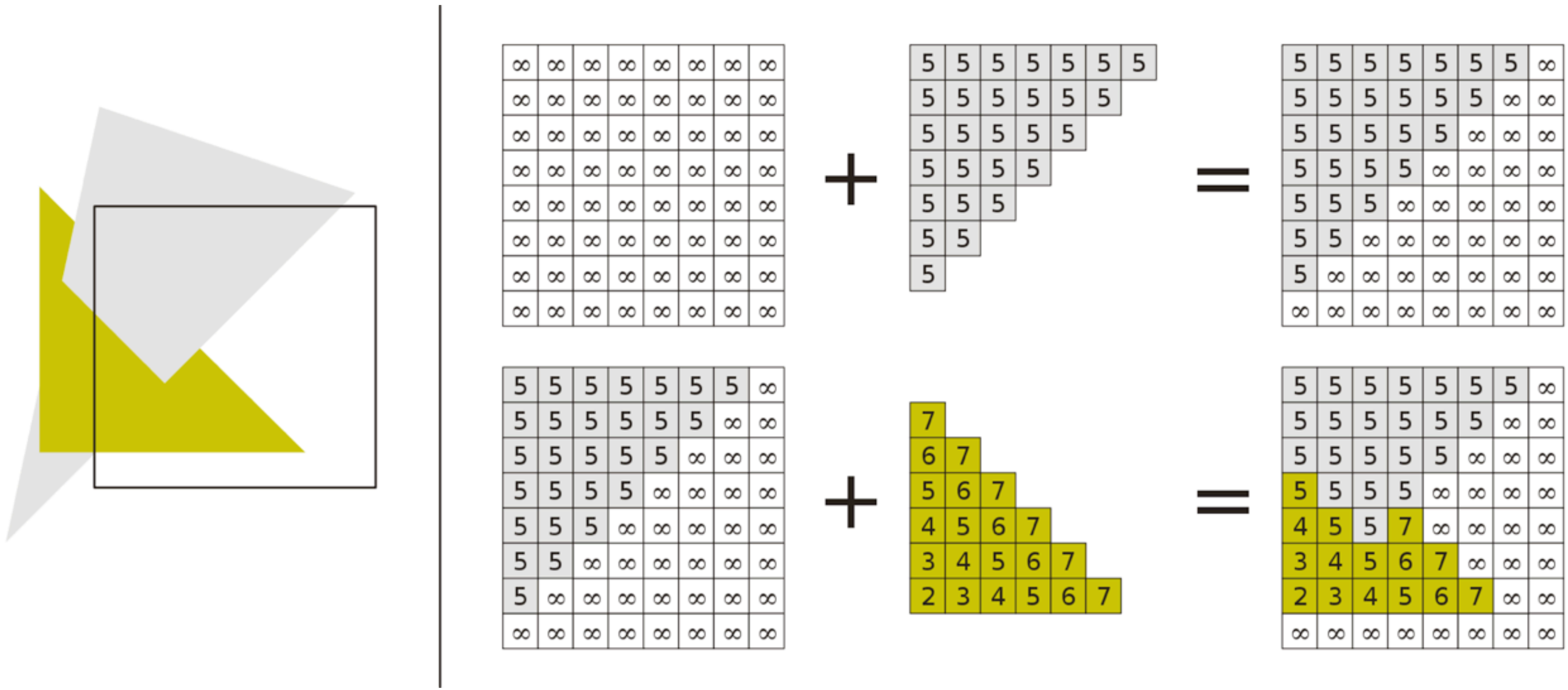


Occlusion: Z-Buffer

- Idea: compute depth not per polygon, but per pixel!
- Approach: for each pixel of the rendered image (frame buffer) keep also a depth value (Z-buffer)
- Initialize the Z-buffer with x_{far} which is the furthest distance we need to care about
- loop over all polygons
  - Determine which pixels are filled by the polygon
  - for each pixel
    - compute the z value (depth) at that position
    - if z > value stored in Z-buffer (remember: negative Z!)
      - draw the pixel in the image
      - set Z-buffer value to z

http://de.wikipedia.org/w/index.php?title=Datei:Z-buffer.svg

Z-Buffer example



Z-Buffer: tips and tricks

- Z-Buffer normally built into graphics hardware
- limited precision (e.g., 16 bit)
  - potential problems with large models
  - set clipping planes wisely!
  - never have 2 polygons in the exact same place
  - otherwise typical errors (striped objects)

- Z-Buffer can be initialized partially to something else than x_{far}
  - at pixels initialized to x_{near} no polygons will be drawn
  - use to cut out holes in objects
  - then rerender objects you want to see through these holes

Chapter 3 - The 3D camera and associated problems

- 3D camera model
- Optimizations for the camera
- How to deal with Occlusion
- Rasterization
  – clipping
  – drawing lines
  – filling areas

Rasterization: the problems

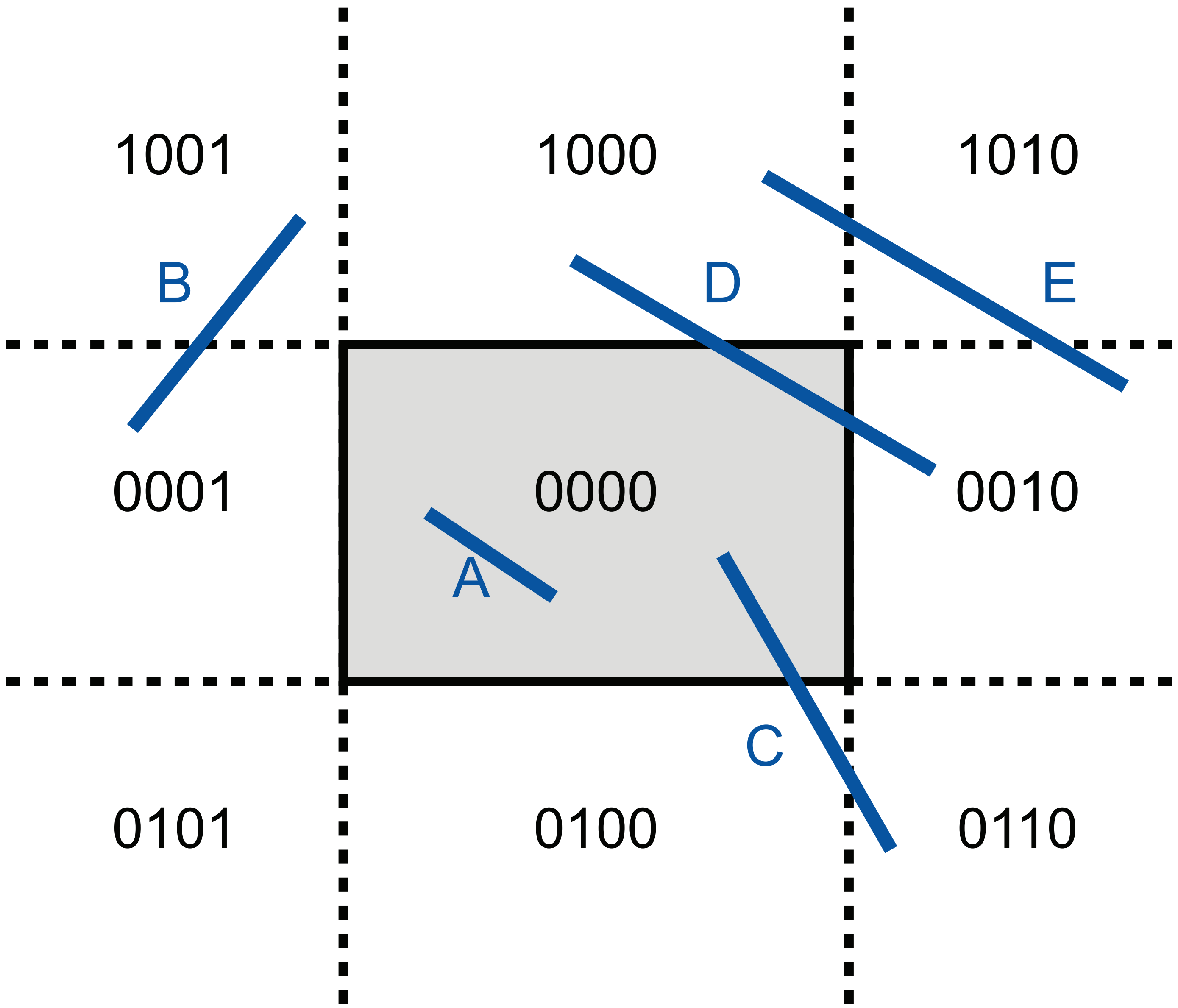
- Clipping: Before we draw a polygon, we need to make sure it is completely inside the image
  - if it already is: OK
  - if it is completely outside: even better ;-)
  - if it intersects the image border: need to do clipping!

- Drawing lines: How do we convert all those polygon edges into lines of pixels?

- Filling areas: How do we determine which screen pixels belong to the area of a polygon?

- Part of this will be needed again towards the end of the semester in the shading/rendering chapter

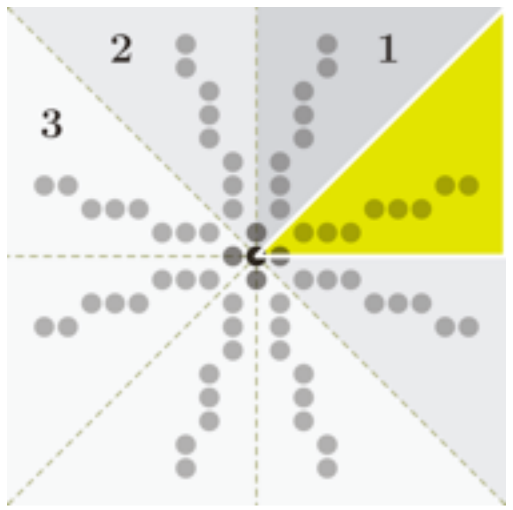
Clipping (Cohen & Sutherland)



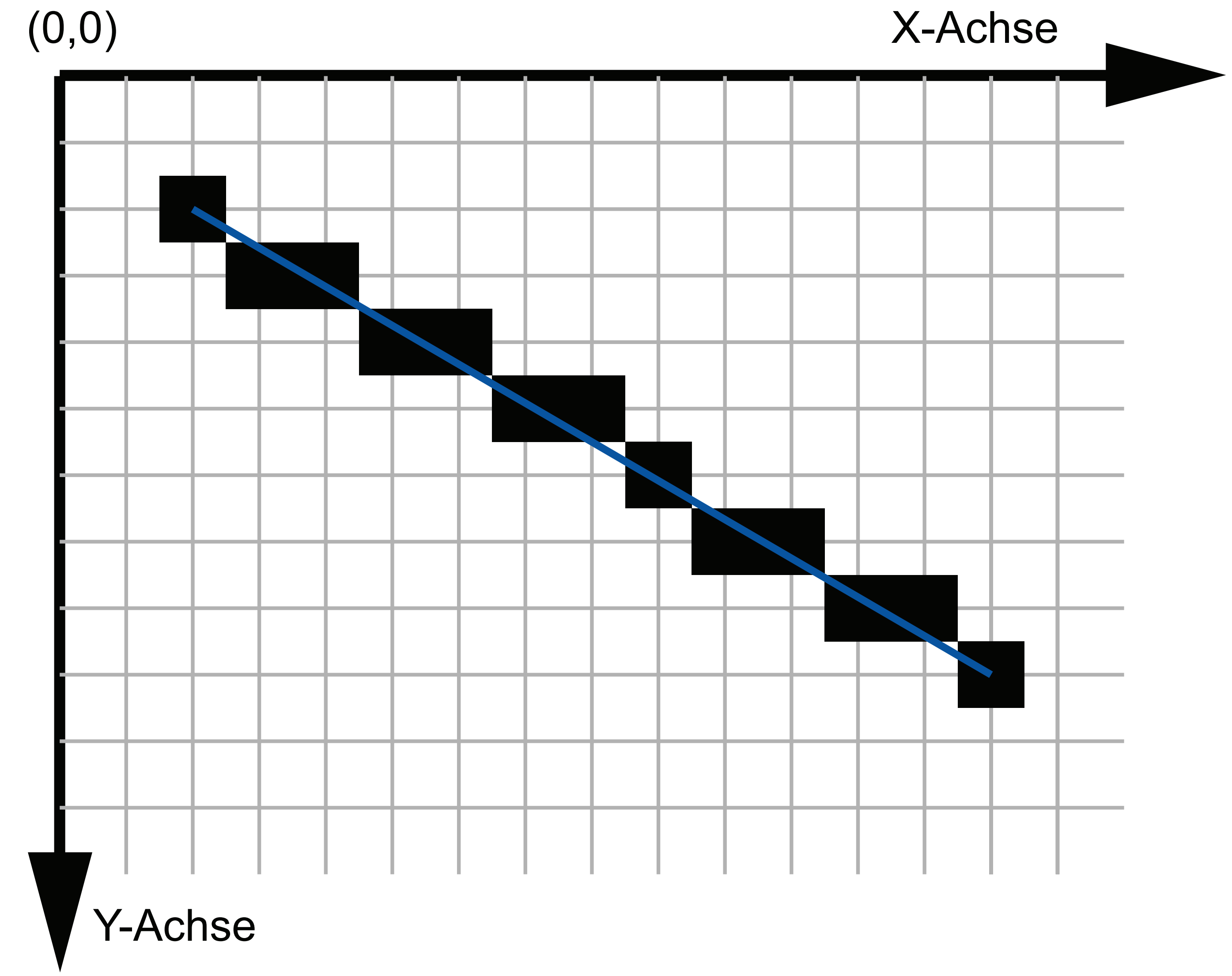
- Clip lines against a rectangle
- For end points P and Q of a line
  - determine a 4 bit code each
  - 10xx = point is above rectangle
  - 01xx = point is below rectangle
  - xx01 = point is left of rectangle
  - xx10 = point is right of rectangle
  - easy to do with simple comparisons
- Now do a simple distinction of cases:
  - P OR Q = 0000: line is completely inside: draw as is (Example A)
  - P AND Q != 0000: line lies completely on one side of rectangle: skip (Example B)
  - P != 0000: intersect line with all reachable rectangle borders (Ex. C+D+E)
    - if intersection point exists, split line accordingly
  - Q != 0000: intersect line with all reachable rectangle borders (Ex. C+D+E)
    - if intersection point exists, split line accordingly

Drawing a line: naive approach

- Line from (x_1,y_1) to (x_2, y_2), Set $dx := x_2 - x_1$, $dy := y_2 - y_1$, $m := dy/dx$
- Assume $x_2 > x_1$, otherwise switch endpoints
- Assume $-1 < m < 1$, otherwise exchange x and y



For x from 0 to dx do:
    setpixel $(x_1 + x, y_1 + m * x)$
od;

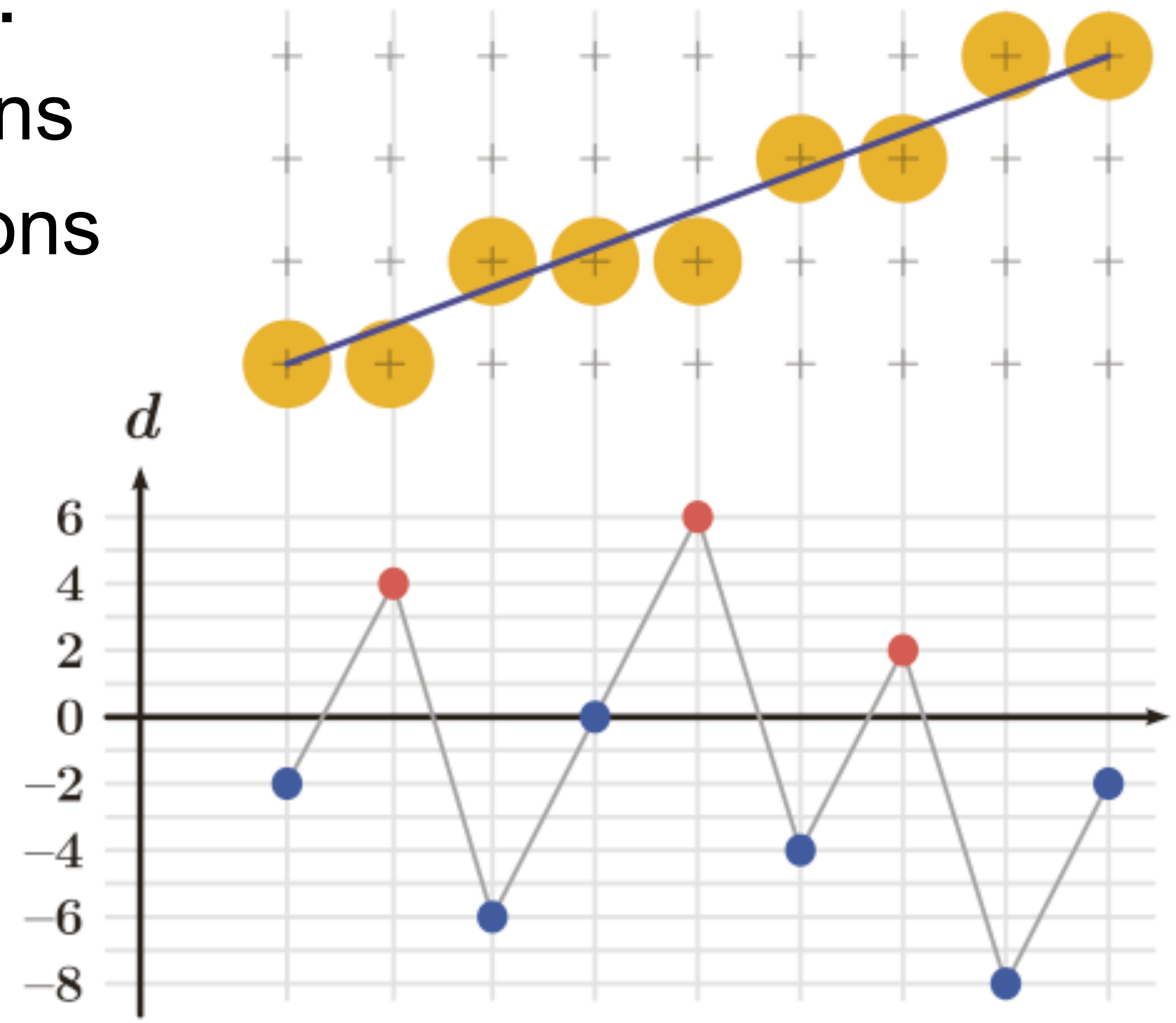


- In each step:
  – 1 float multiplication
  – 1 round to integer

top figure from http://de.wikipedia.org/w/index.php?title=Datei:Line_drawing_symmetry.svg

Drawing a line: Bresenham‘s Algorithm

- Idea: go in incremental steps
- Accumulate error to ideal line
  - go one pixel up if error beyond a limit
- Uses only integer arithmetic
- In each step:
  - 2 comparisons
  - 3 or 4 additions

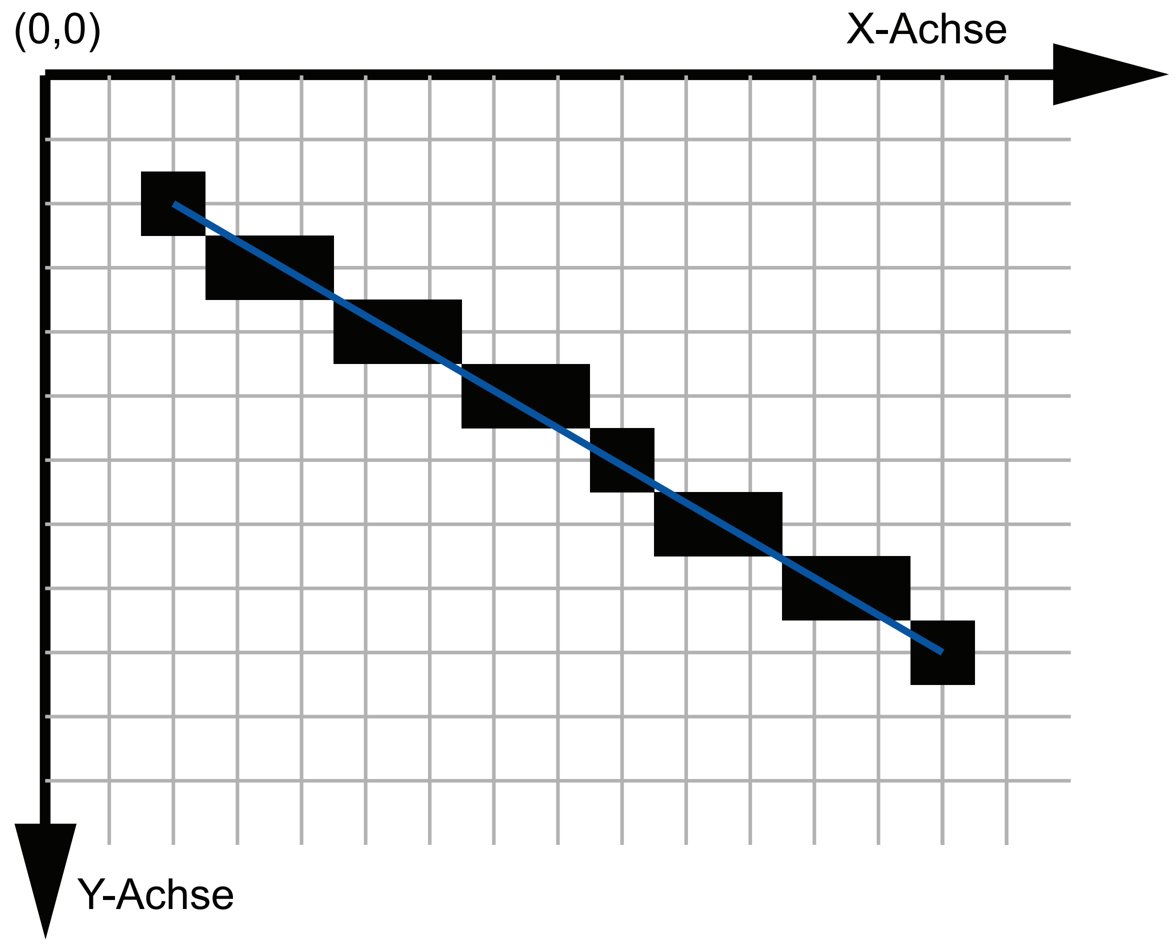


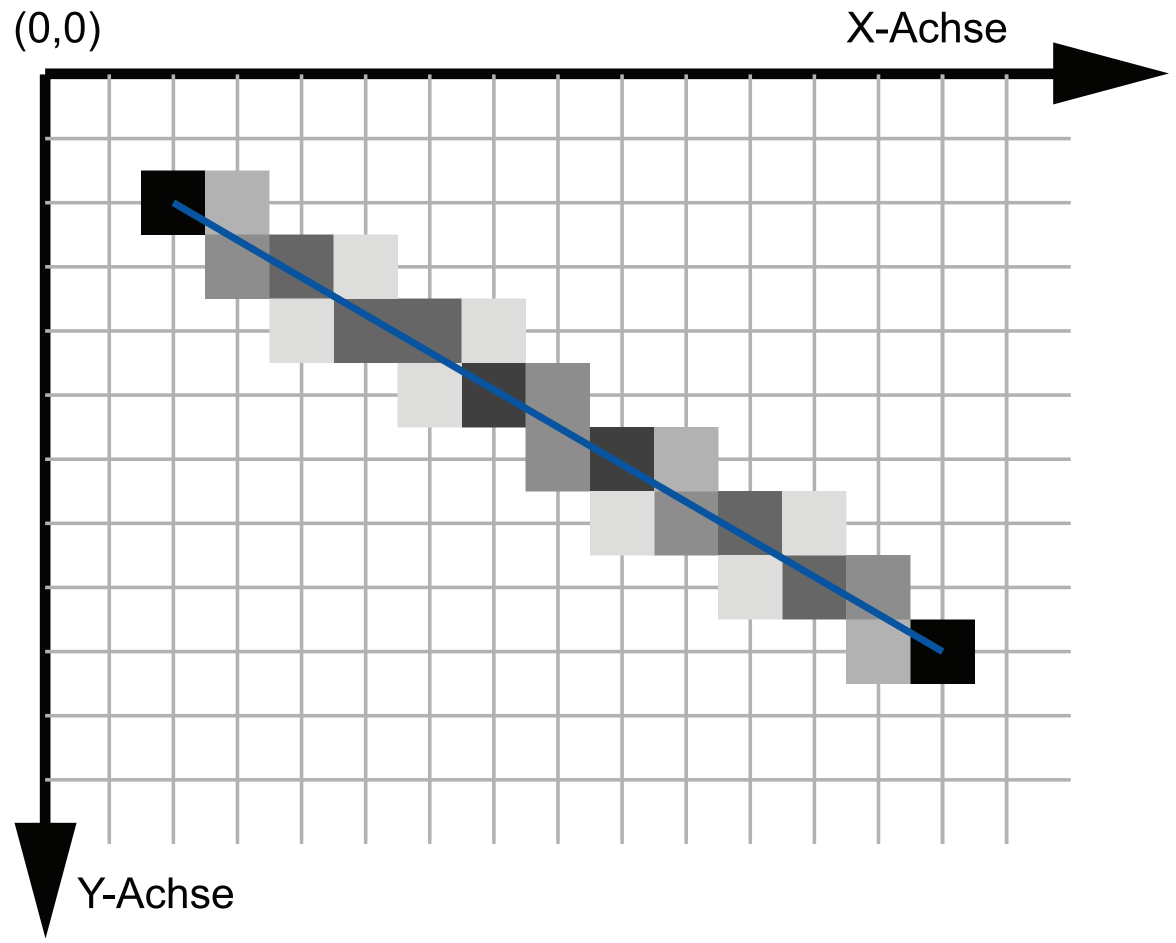
http://de.wikipedia.org/w/index.php?title=Datei:Bresenham_decision_variable.svg

```
dx := x2-x1; dy := y2-y1
d := 2*dy – dx; DO := 2*dy;
dNO := 2*(dy - dx)
x := x1; y := y1
setpixel (x,y)
fehler := d
WHILE x < x2
      x := x + 1
      IF fehler <= 0 THEN
            fehler := fehler + DO
      ELSE
            y := y + 1
            fehler = fehler + dNO
      END IF
      setpixel (x,y)
END WHILE
```

Antialiased lines

- Problem: Bresenham‘s lines contain visible steps (aliasing effects)
- Opportunity: we can often display greyscale
- Idea: use different shades of grey as different visual weights
  - instead of filling half a pixel with black, fill entire pixel with 50% grey



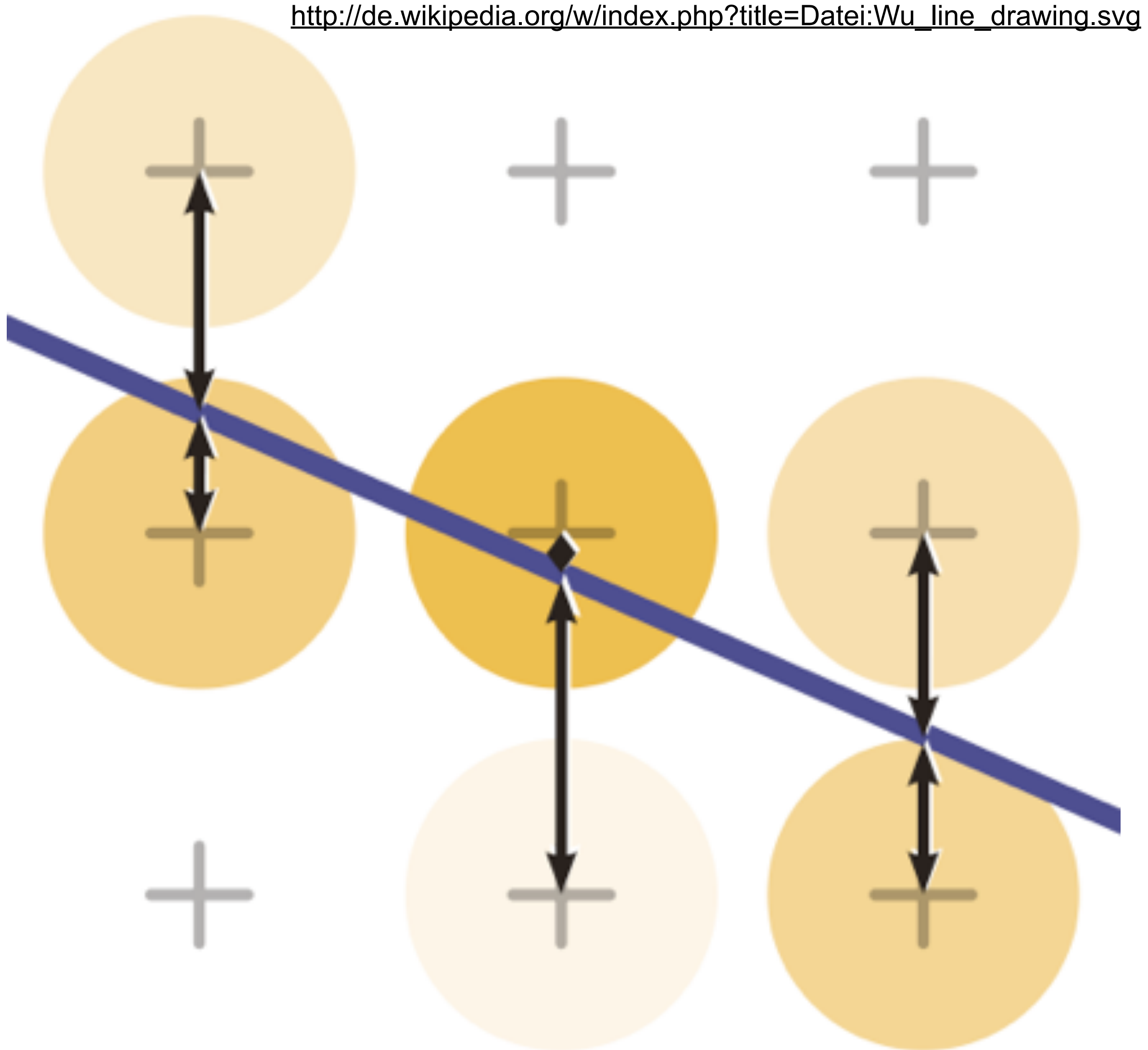


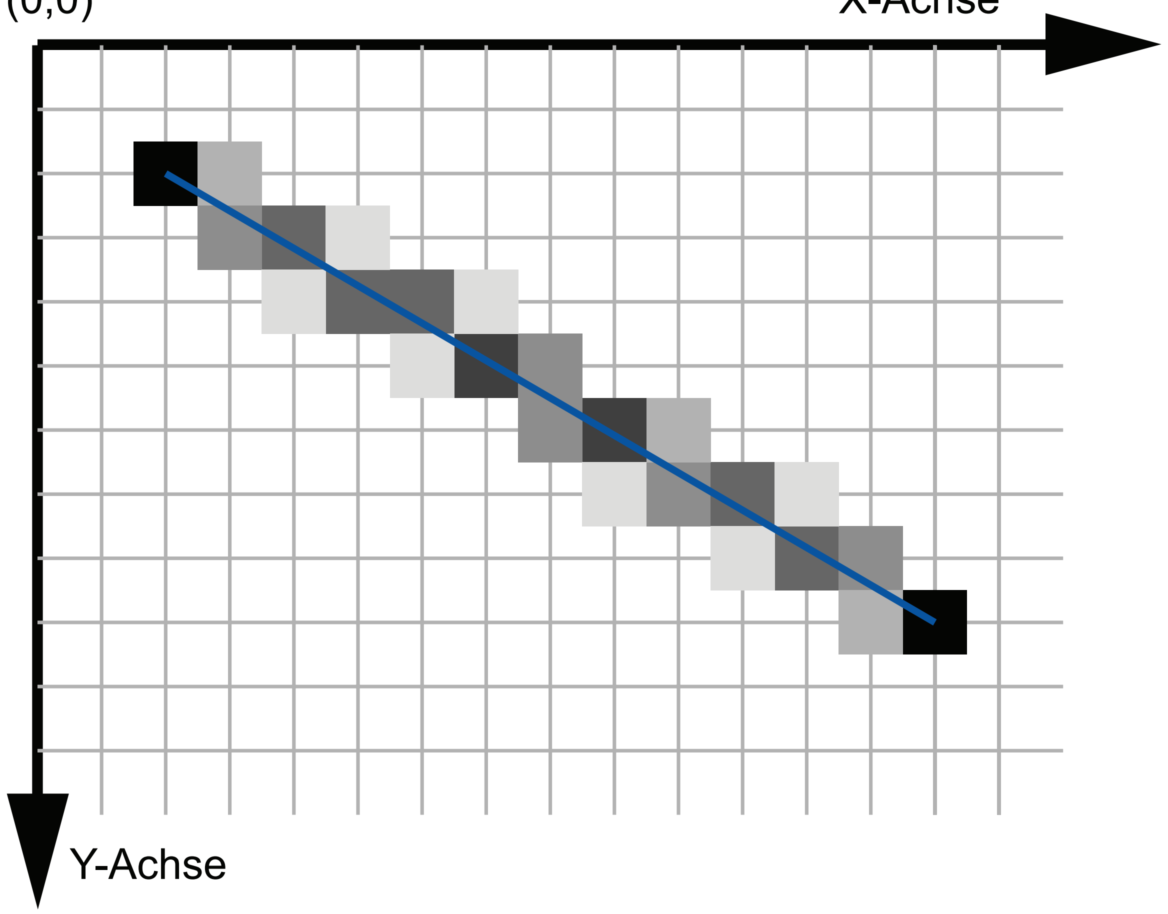
- Different algorithms exist
  - Gupta-Sproull for 1 pixel wide lines
  - Wu for infinitely thin lines

http://de.wikipedia.org/w/index.php?title=Datei:Wu_line_drawing.svg

Wu‘s antialiasing approach

- Loop over all x values
- Determine 2 pixels closest to ideal line
  - slightly above and below
- Depending on distance, choose grey values
  - one is perfectly on line: 100% and 0%
  - equal distance: 50% and 50%
- Set these 2 pixels

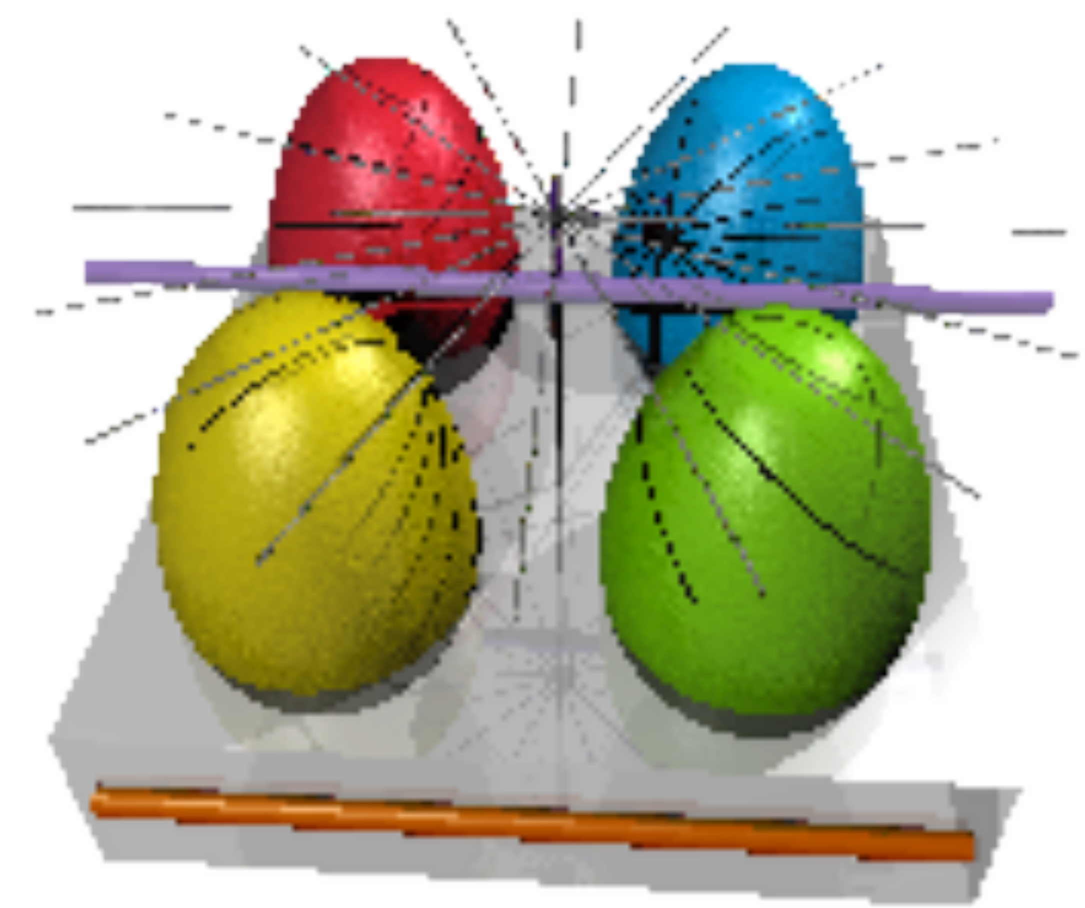


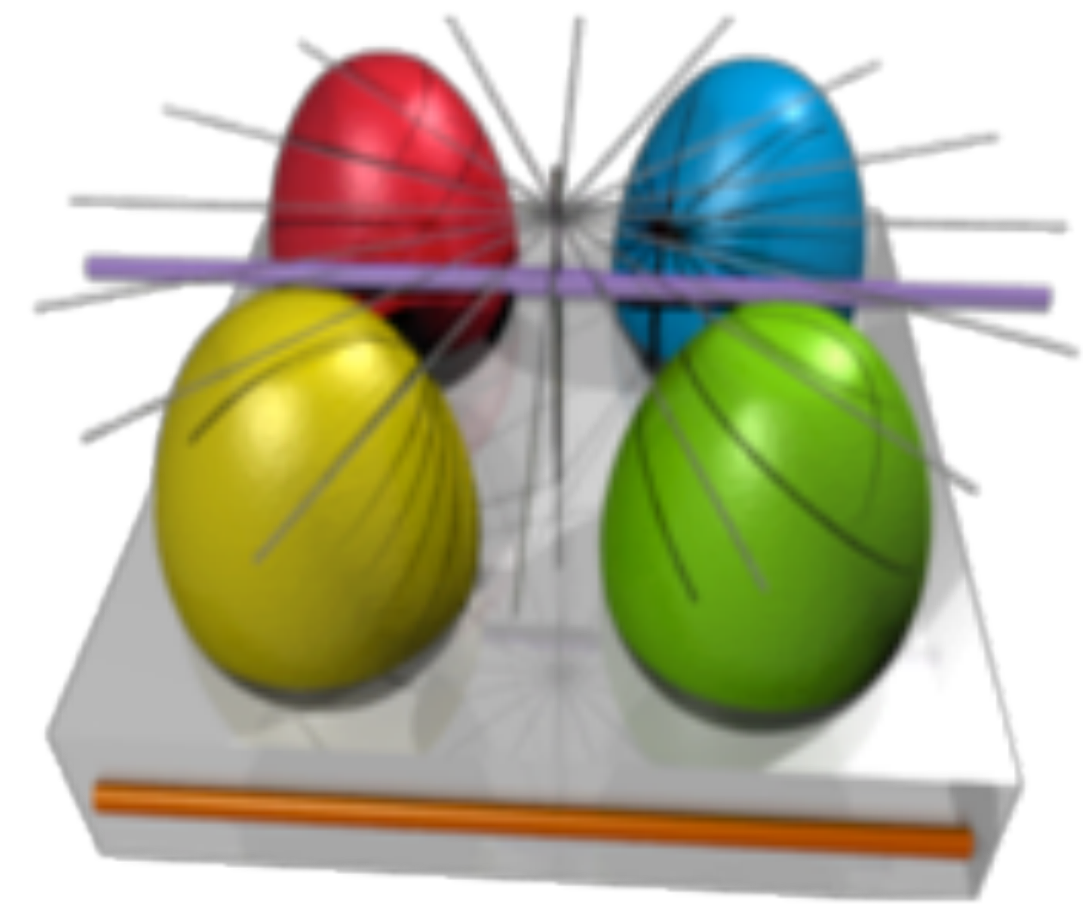


Antialiasing in General

- Problem: hard edges in computer graphics
- Correspond to infinitely high spatial frequency
- Violate sampling theorem (Nyquist, Shannon)
  - reread 1st lecture „Digitale Medien“

- Most general technique: Supersampling
- Idea:
  - render an image at a higher resolution
    - this way, effectively sample at a higher resolution
  - scale it down to intended size
  - interpolate pixel values
    - this way, effectively use a low pass filter





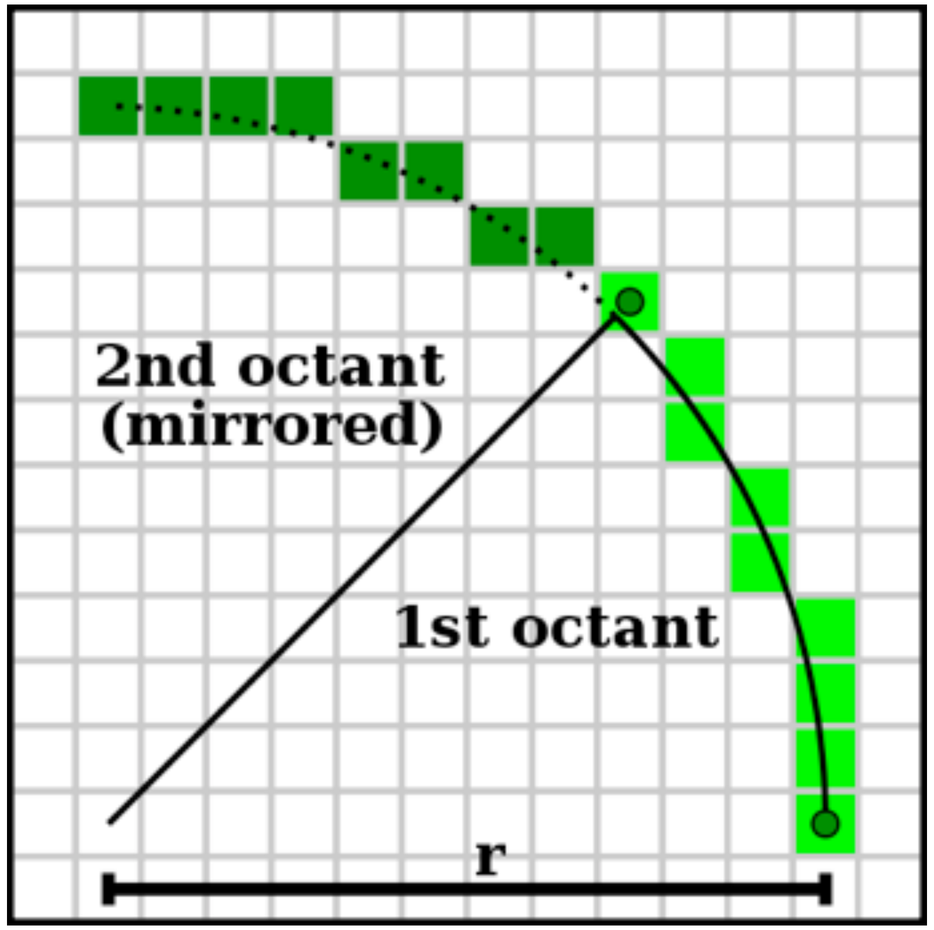
http://de.wikipedia.org/w/index.php?title=Datei:EasterEgg_anti-aliasing.png

Line drawing: summary

- With culling and clipping, we made sure all lines are inside the image
- With algorithms so far we can draw lines in the image
  – even antialiased lines directly
- This means we can draw arbitrary polygons now (in black and white)

- All algorithms extend to color
  – just modify the setpixel (x,y) implementation
  – choice of color not always obvious (think through!)
  – how about transparency?

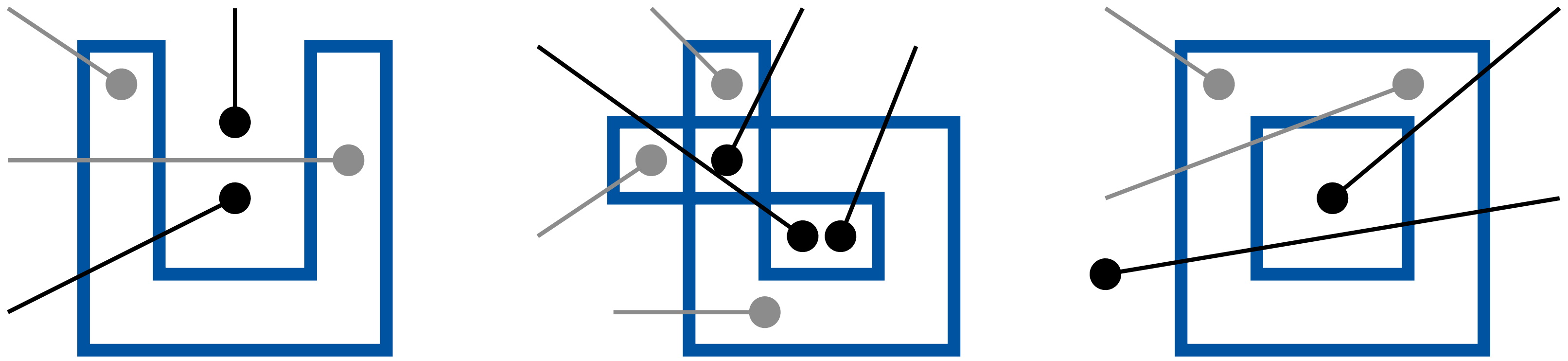
- All these algorithms implemented in hardware
- Other algorithms exist for curved lines
  – mostly relevant for 2D graphics



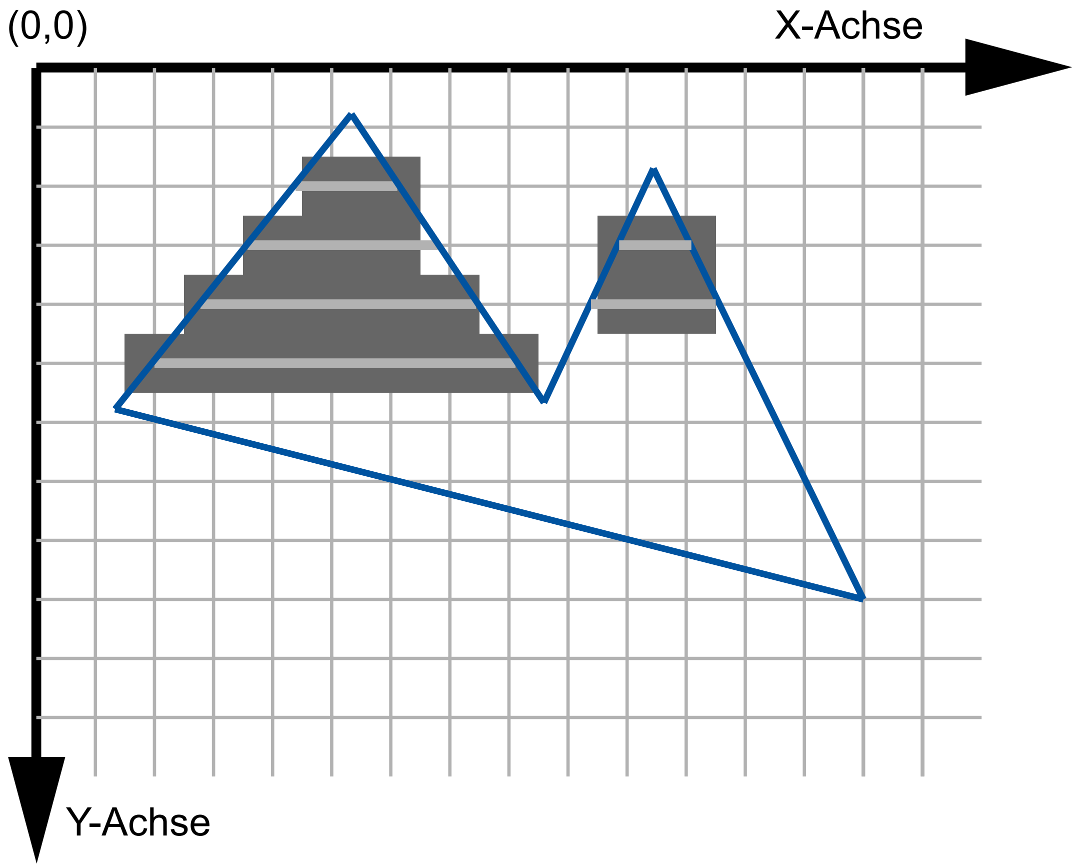
http://en.wikipedia.org/wiki/File:Bresenham_circle.svg

Filling a Polygon: Scan line algorithm

- Define parity of a point in 2D:
  - send a ray from this point to infinity
  - direction irrelevant (!)
  - count number of lines it crosses
  - if 0 or even: even parity (outside)
  - if odd: odd parity (inside)



- Determine polygon area (x_{min}, x_{max}, y_{min}, y_{max})
- Scan the polygon area line by line
- Within each line, scan pixels from left to right
  - start with parity = 0 (even)
  - switch parity each time we cross a line
  - set all pixels with odd parity



Rasterization summary

- Now we can draw lines and fill polygons
- All algorithms also generalize to color
- Ho do we determine the shade of color?
  - this is called shading and will be discussed in the rendering section