Computer Graphics 1

Ludwig-Maximilians-Universität München
Summer semester 2020

Prof. Dr.-Ing. Andreas Butz
lecture additions by Dr. Michael Krone, Univ. Stuttgart

https://commons.wikimedia.org/wiki/File:Stanford_bunny_qem.png

Chapter 5 – 3D Camera & Rasterization

- Classical Viewing Taxonomy
- 3D Camera Model
- Optimizations for the Camera
- How to Deal with Occlusion
- Rasterization
  - Clipping
  - Drawing lines
  - Filling areas

Classical Views of 3D Scenes

- As used in arts, architecture, and engineering
  - Traditional terminology has emerged
  - Varying support by 3D graphics SW and HW
- Assumptions:
  - Objects constructed from flat faces (polygons)
  - Projection surface is a flat plane
    - Nonplanar projections also exist in special cases
- General situation:
  - Scene consisting of 3D objects
  - Viewer with defined position and projection surface
  - *Projectors (Projektionsstrahlen)* are lines going from objects to the projection surface
- Main classification:
  - Parallel projectors or converging projectors

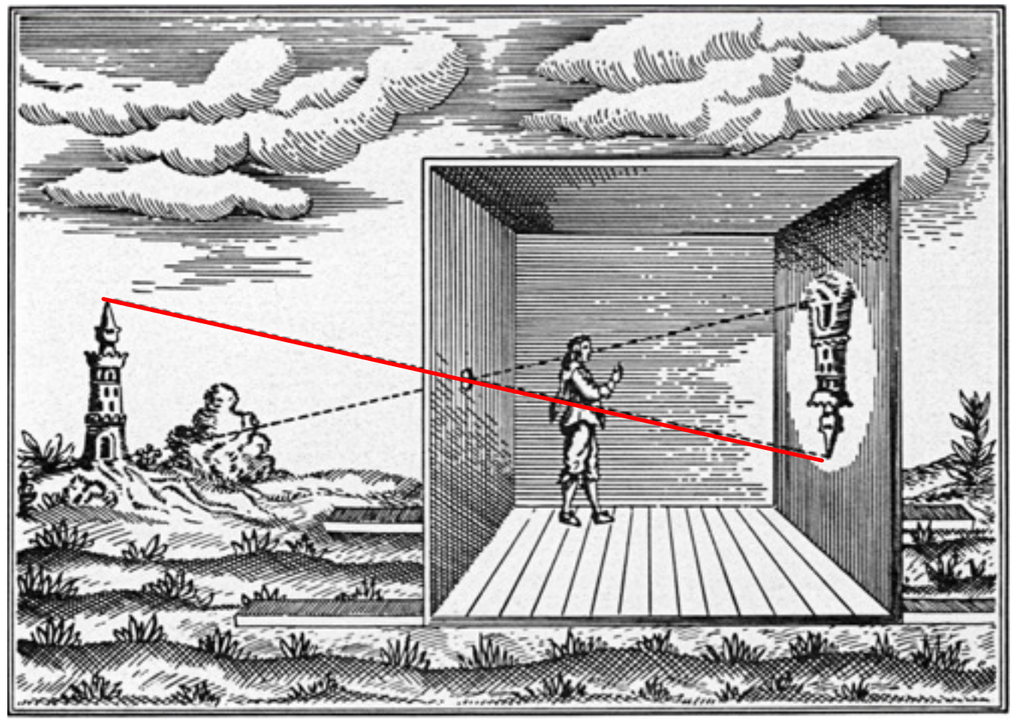
http://www.semioticon.com/seo/P/images/perspective_1.jpg

http://www.techpin.com/2008/08/page/18/

Classical Views of 3D Scenes

- As used in arts, architecture, and engineering
  - Traditional terminology has emerged
  - Varying support by 3D graphics SW and HW
- Assumptions:
  - Objects constructed from flat faces (polygons)
  - Projection surface is a flat plane
    - Nonplanar projections also exist in special cases
- General situation:
  - Scene consisting of 3D objects
  - Viewer with defined position and projection surface
  - *Projectors (Projektionsstrahlen)* are lines going from objects to the projection surface
- Main classification:
  - Parallel projectors or converging projectors

http://www.semioticon.com/seo/P/images/perspective_1.jpg



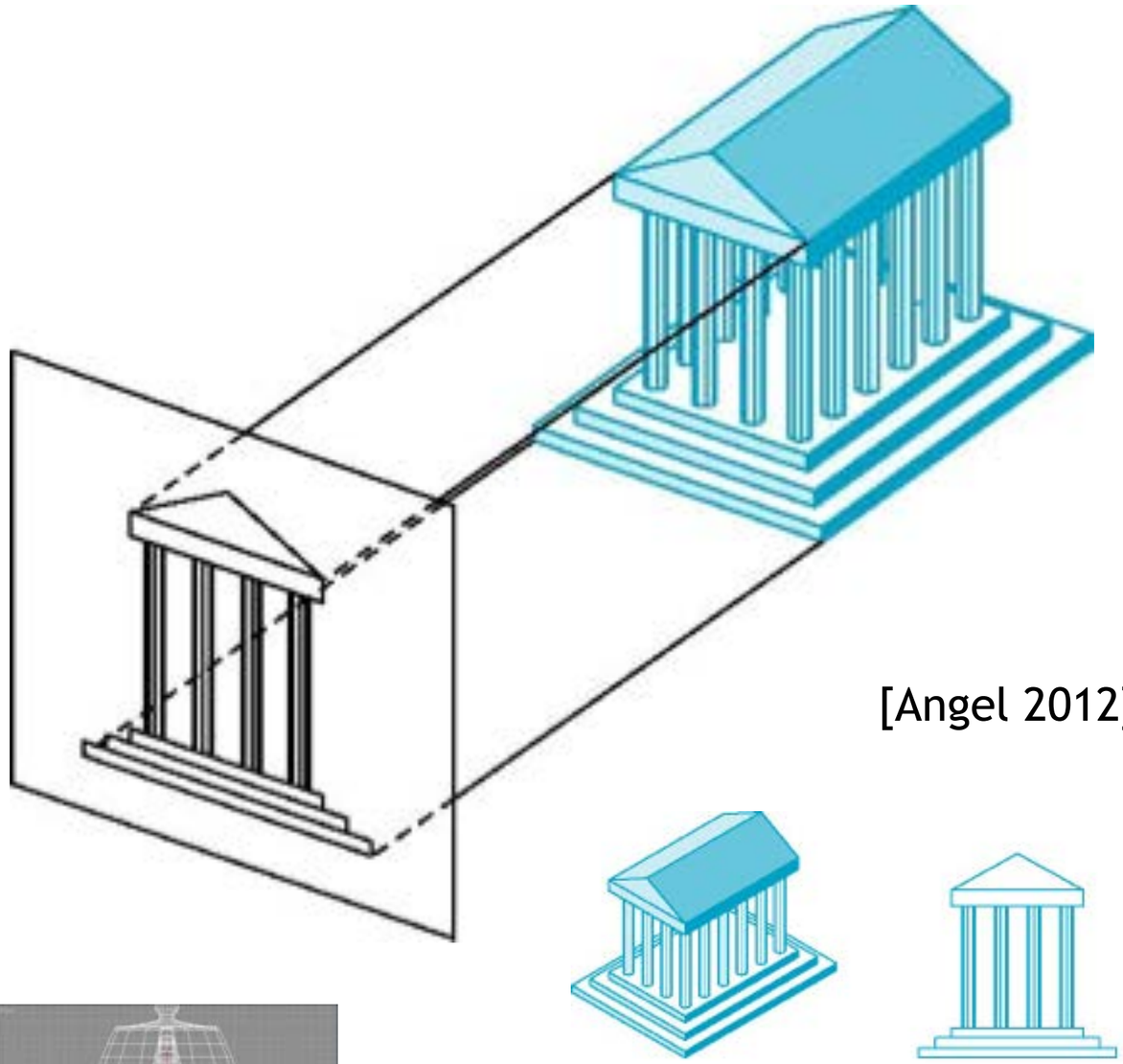
http://cobweb.cs.uga.edu/~maria/classes/2016-CompPhoto/

Taxonomy of Views

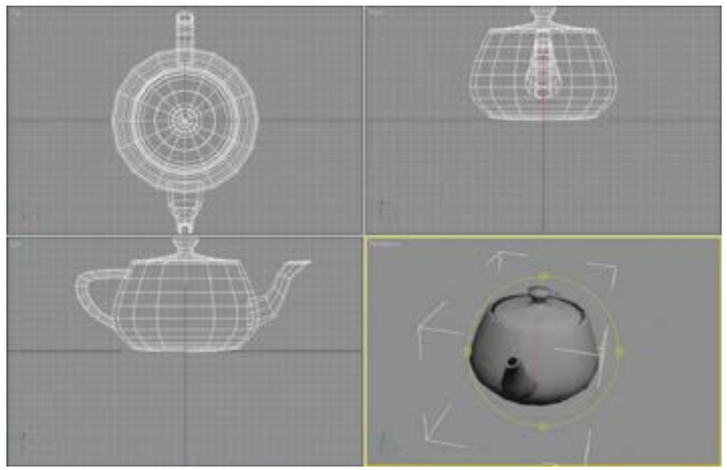
[Angel 2012]

Orthographic Projection

- Projectors are orthogonal to the projection plane
- In the “pure” case, projection plane is parallel to a coordinate plane
  - top/front/side view
  - Often used as a multi-view combination
  - Often together with overview (e.g. isometric view)
- Advantage:
  - No distortions
  - Can be used for measurements

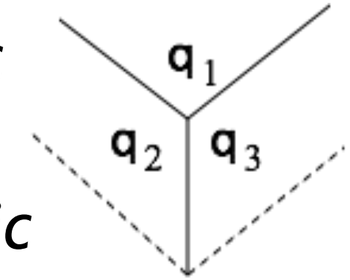


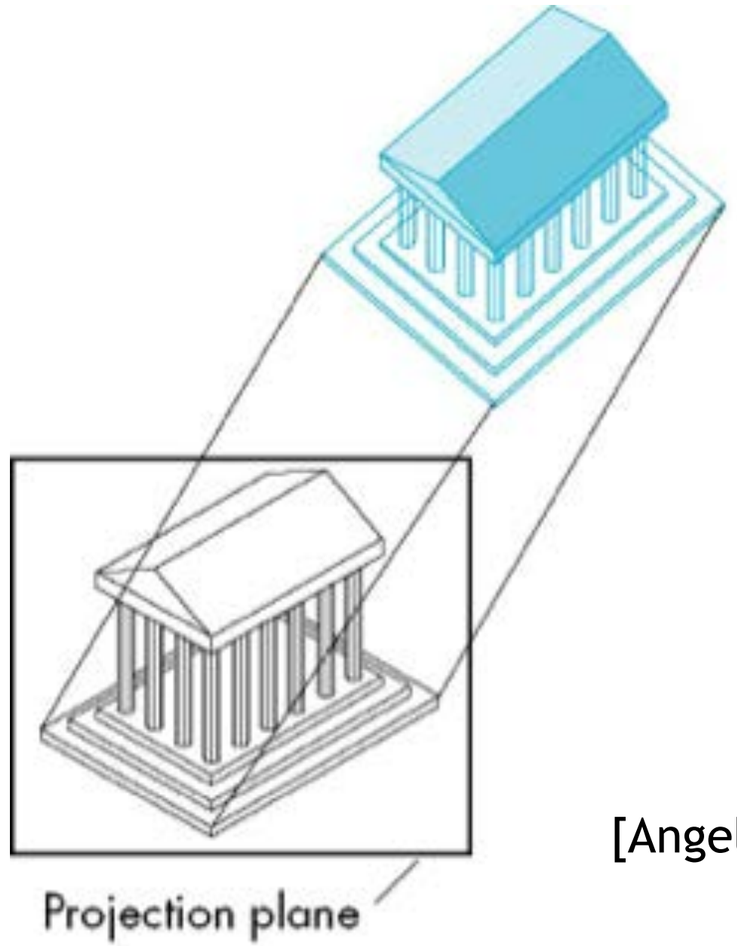
[Angel 2012]



Axonometric Projections

- Using orthographic projection, but with arbitrary placement of projection plane
- Classification of special cases:
  - Look at a corner of a projected cube
  - How many angles are identical?
    - None: *trimetric*
    - Two: *dimetric*
    - Three: *isometric*

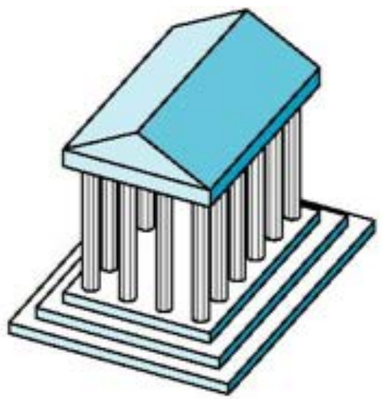




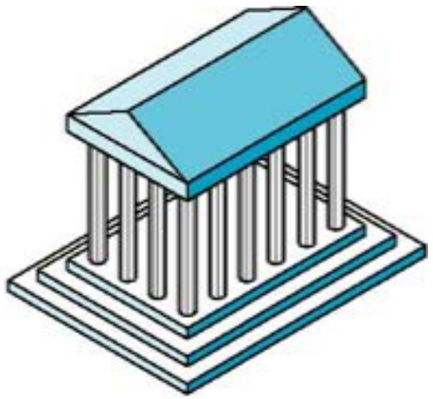
[Angel 2012]

- Advantage:
  - Preserves lines
  - Somehow realistic
- Disadvantage:
  - Angles not preserved

Dimetric

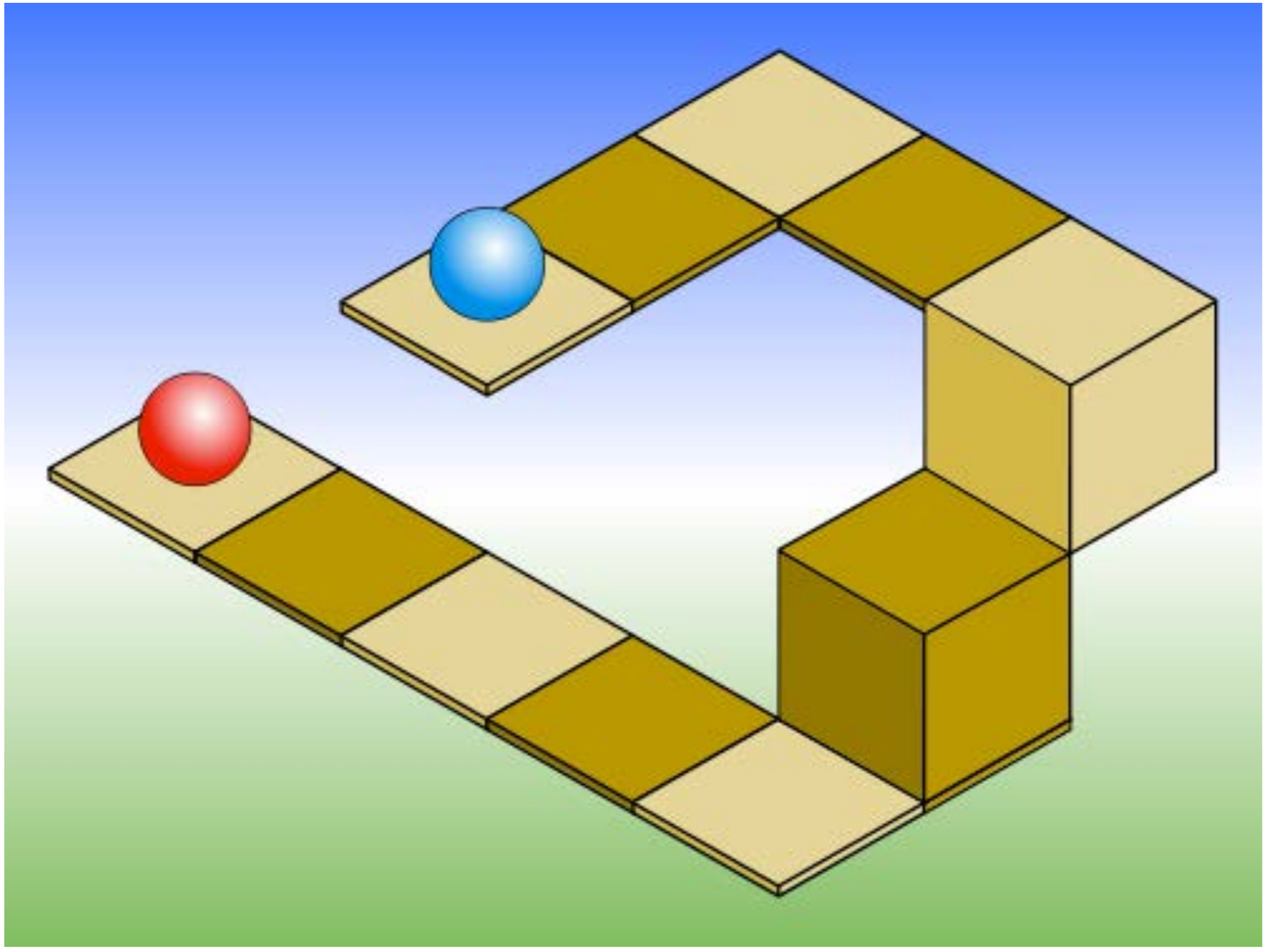


Trimetric



Isometric

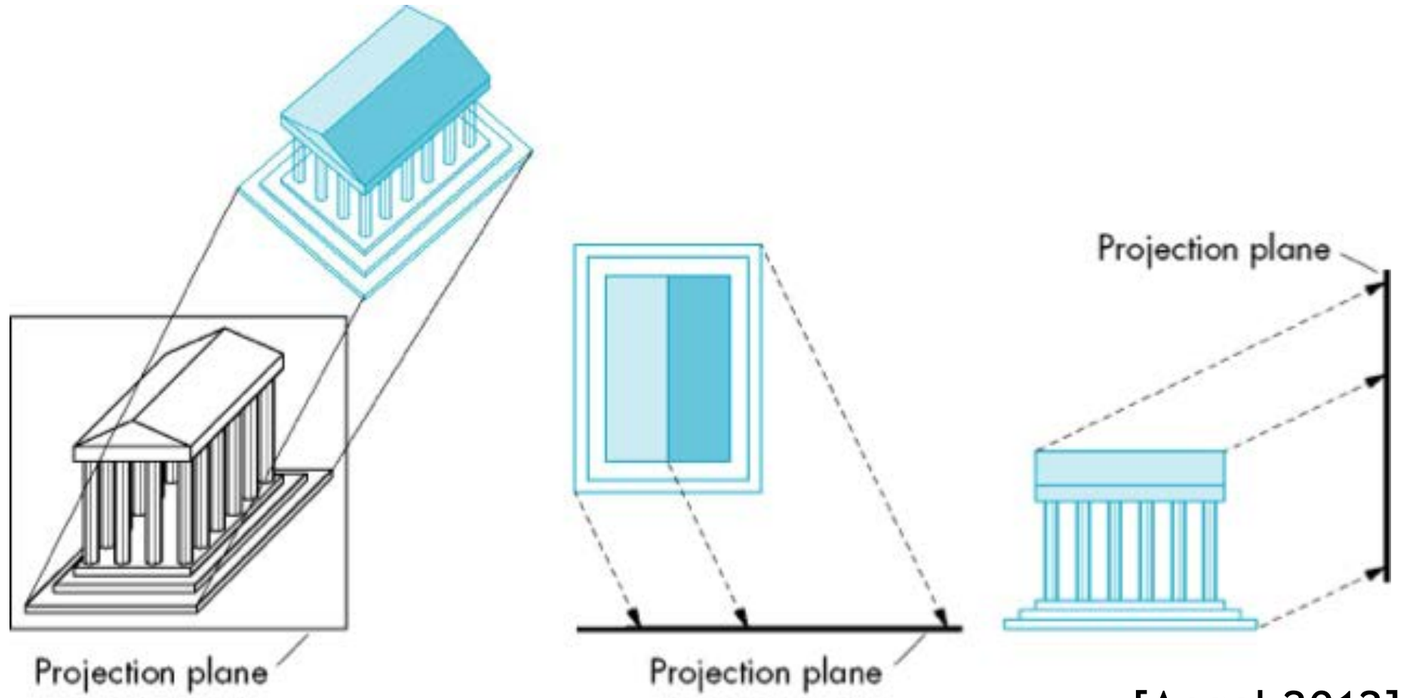
Optical Illusions in Isometric Projections



Source:
Wikipedia

Oblique Projection *(Schiefe Parallelprojektion)*

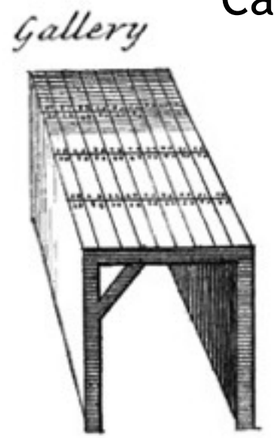
- Projectors are not orthogonal to projection plane
  - Usually projection plane parallel to one coordinate plane
- Traditional subclasses:
  - *Cavalier perspective*
    - Constant angle (usually 30° or 45°) between direction of projectors (DoP) and projection plane
    - No foreshortening
  - *Cabinet perspective*
    - Constant angle (30°/45°/63.4°) between DoP and projection plane
    - Foreshortening (Verkürzung) (of depth) by factor 0.5

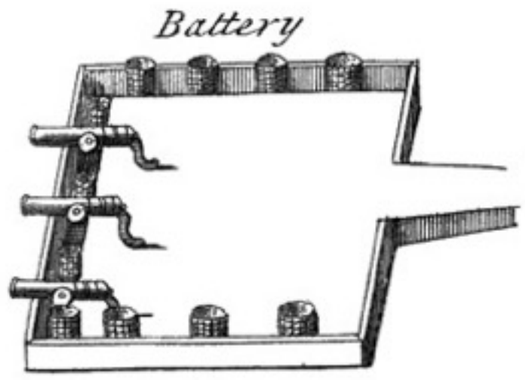


[Angel 2012]

Cavalier projection

Cabinet projection

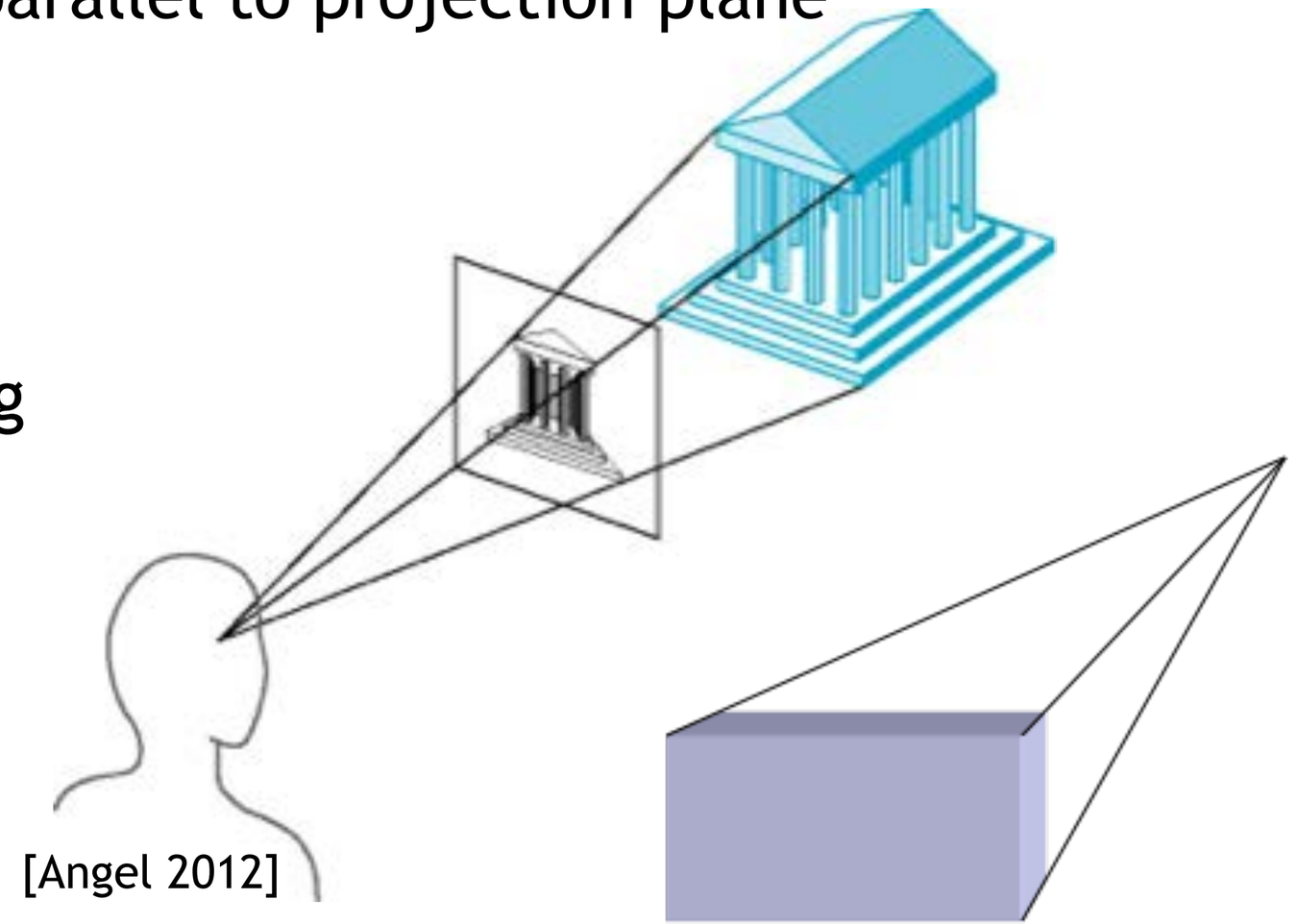




Images: Wikipedia

Perspective Projection *(Perspektivische Projektion)*

- Projectors converge at *center of projection* (cop)
- Parallel lines appear to converge in a *vanishing point* (Fluchtpunkt)
  - *Exception*: Lines that are parallel to projection plane
- Advantage:
  - Very realistic
- Disadvantage:
  - Non-uniform foreshortening
  - Only few angles preserved



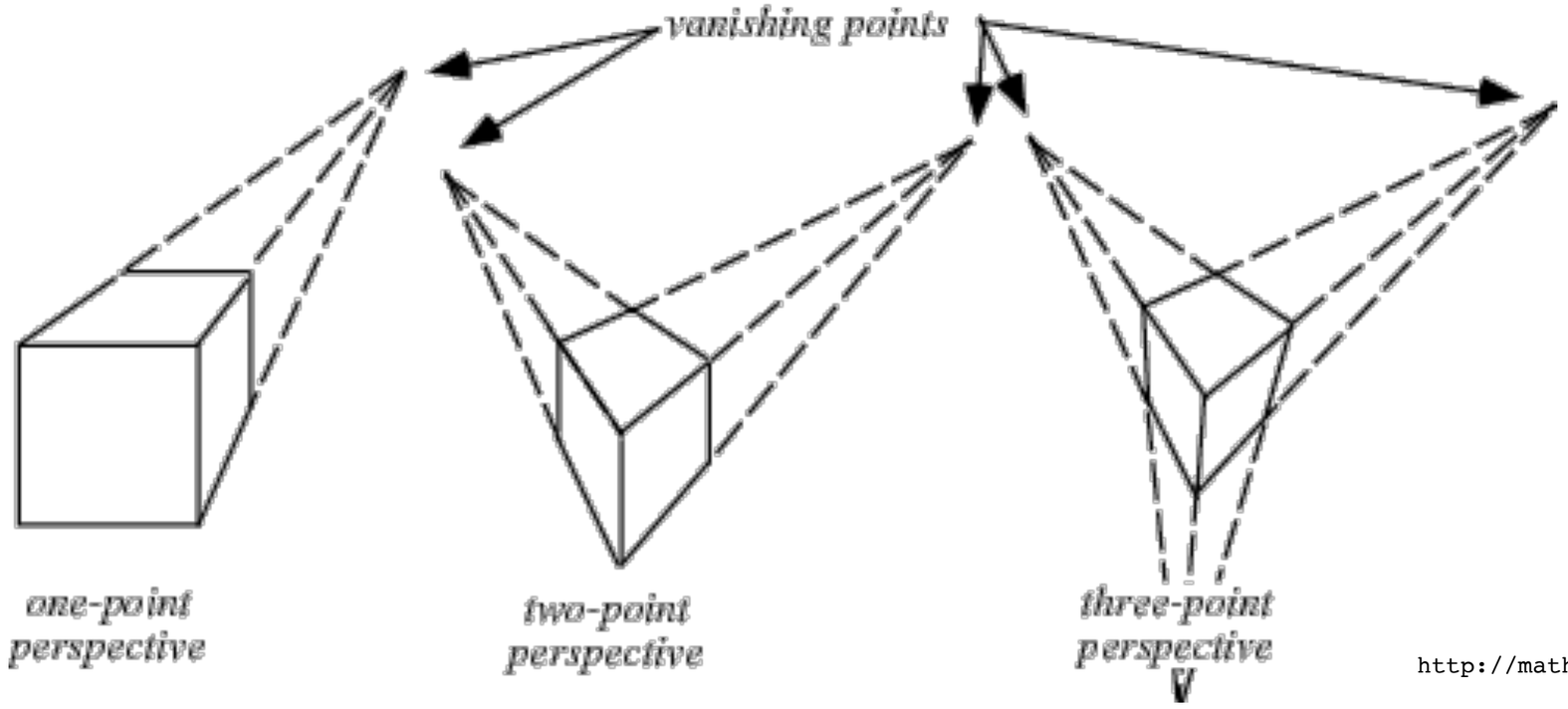
[Angel 2012]

Number of Vanishing Points in Perspective Projection

One point

Two points

Three points



http://mathworld.wolfram.com/Perspective.html

How to Realize Projection in Three.js?

- Parallel / Orthographic projections:
  → `THREE.OrthographicCamera( left, right, top, bottom, near, far);`

```
var camera = new THREE.OrthographicCamera(w/-2, w/2, h/2, h/-2, 1, 1000);
scene.add(camera);
```

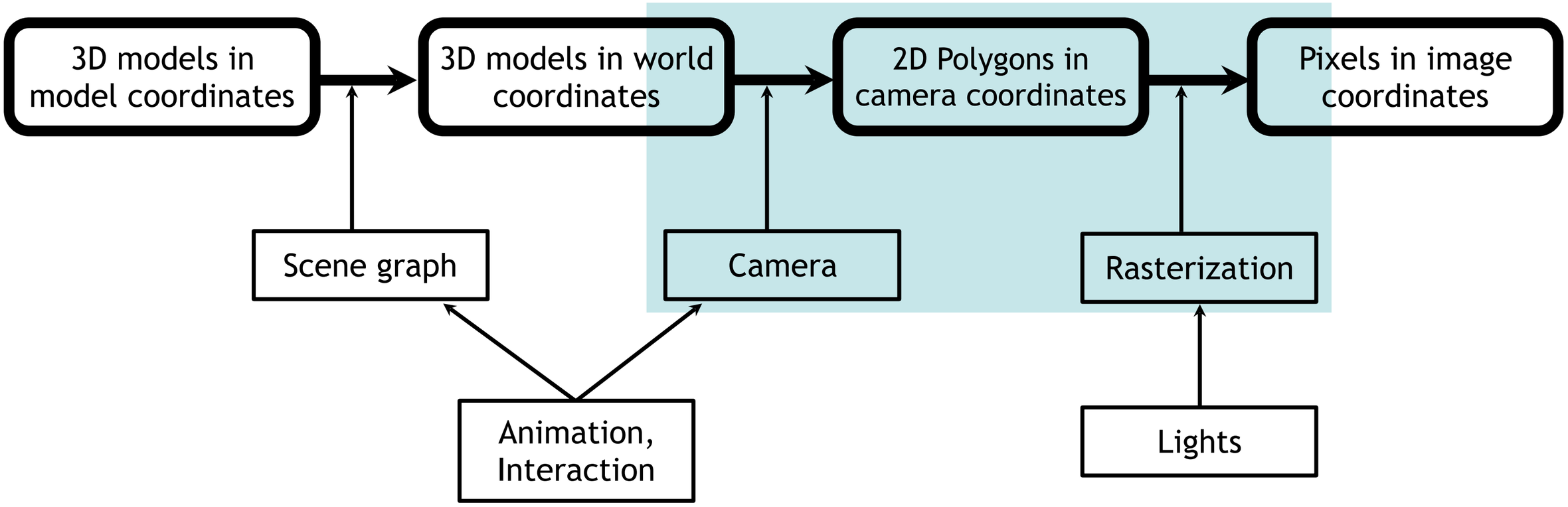
- Perspective projections:
  → `THREE.PerspectiveCamera( field of view (angle), aspect ratio, near, far);`

```
var camera = new THREE.PerspectiveCamera(45, w/h, 1, 1000);
scene.add(camera);
```

Chapter 5 – 3D Camera & Rasterization

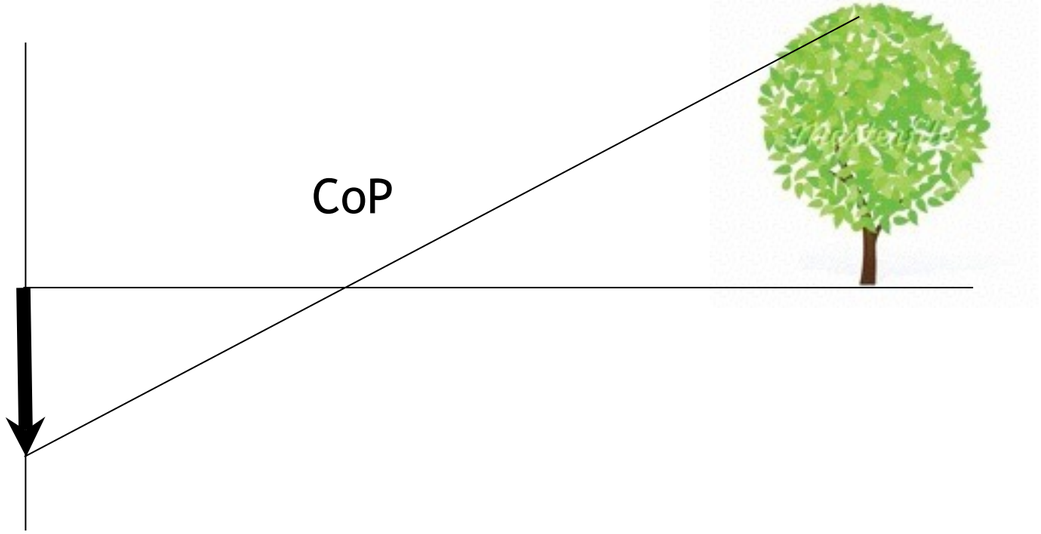
- Classical Viewing Taxonomy
- 3D Camera Model
- Optimizations for the Camera
- How to Deal with Occlusion
- Rasterization
  - Clipping
  - Drawing lines
  - Filling areas

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

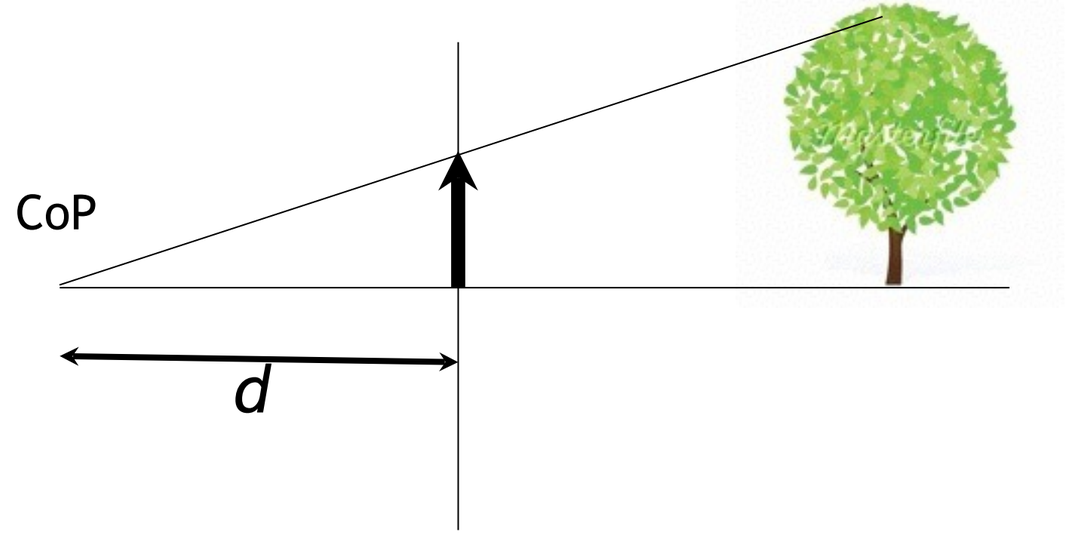


Perspective Projection and Photography

- In photography, we usually have the center of projection (CoP) between the object and the image plane
  - Image on film/sensor is upside down
- In CG perspective projection, the image plane is in front of the camera!



Photography

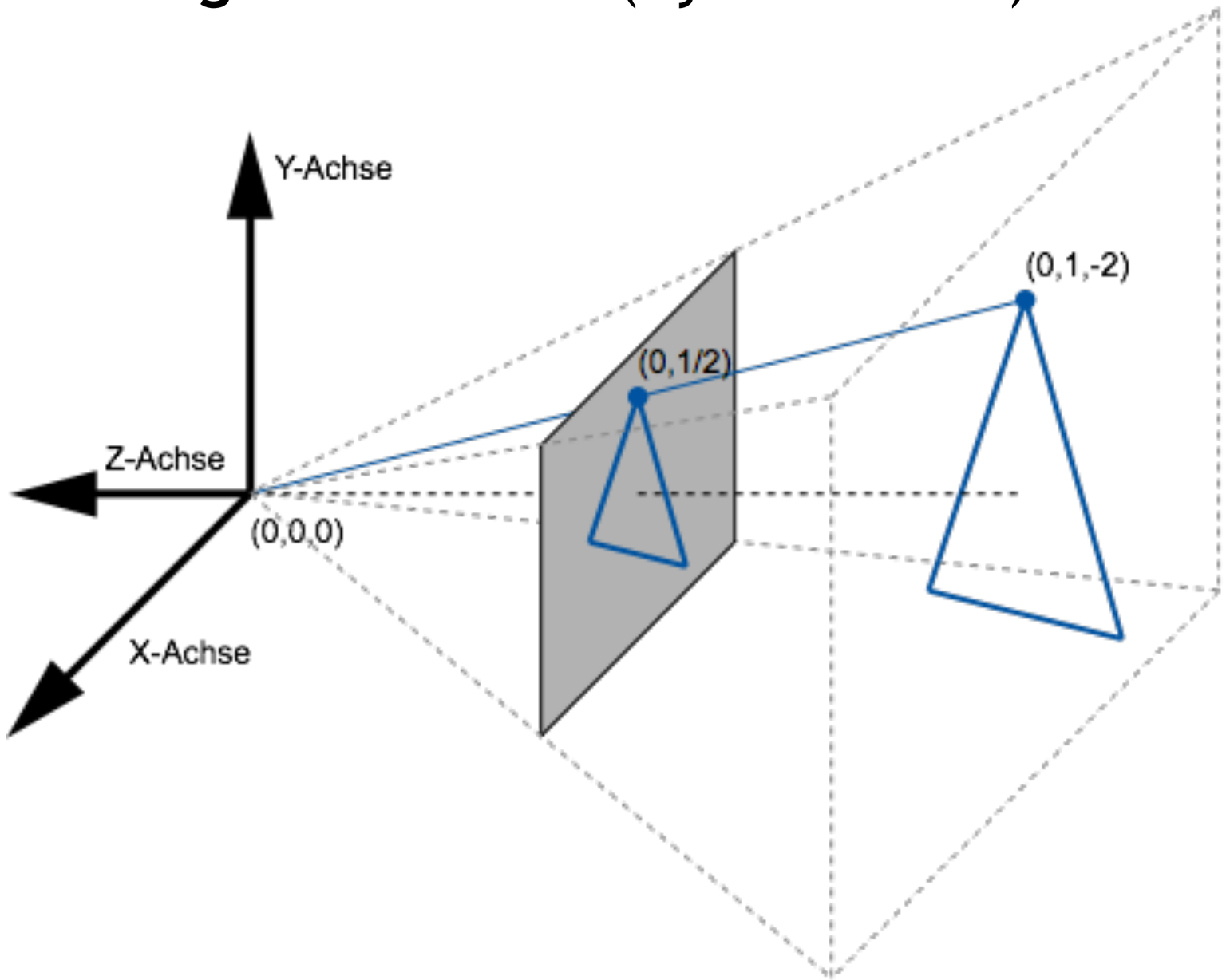


CG Perspective Projection

The Mathematical Camera Model for Perspective Projection

- The camera looks along the ***negative Z axis*** (by definition)
- 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)

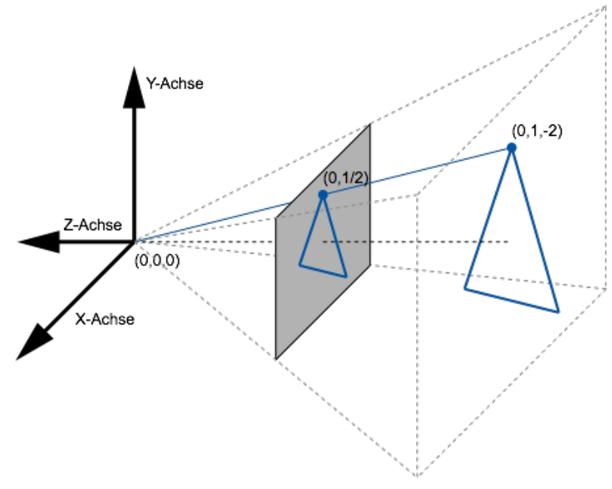
Perspective Division

- Divide each point by its 4th coordinate w

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

- 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 will get 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 absolute value means _____ angle
  - Could also be done in the division step

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

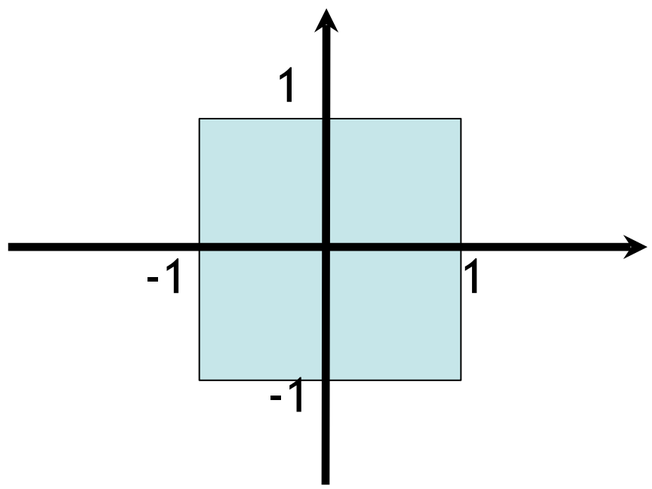
From Image to Screen Coordinates

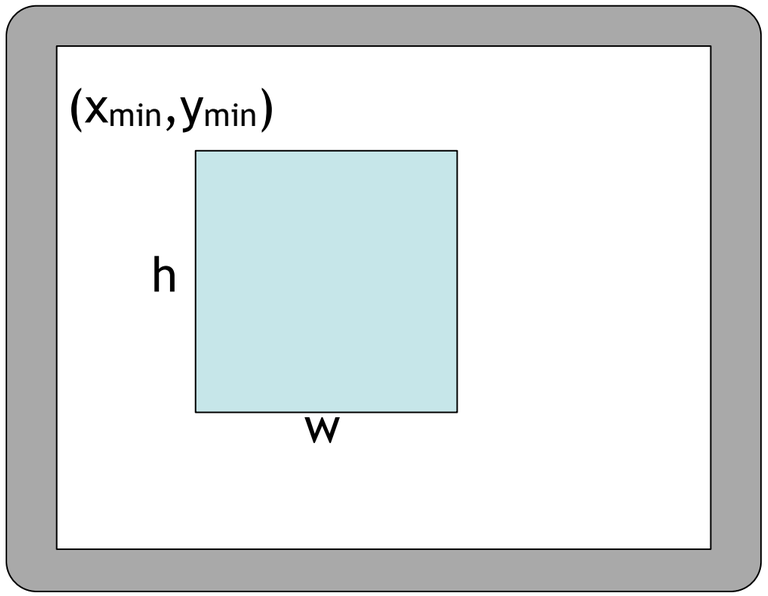
- Camera takes us from world via view to image coordinates

  $-1 < x_{image} < 1,$

  $-1 < y_{image} < 1$

- 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 $x_{screen} = x_{min} + w \cdot \frac{1+x_{image}}{2}, y_{screen} = y_{min} + h \cdot \frac{1-y_{image}}{2}$





Chapter 5 – 3D Camera & Rasterization

- Classical Viewing Taxonomy
- 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

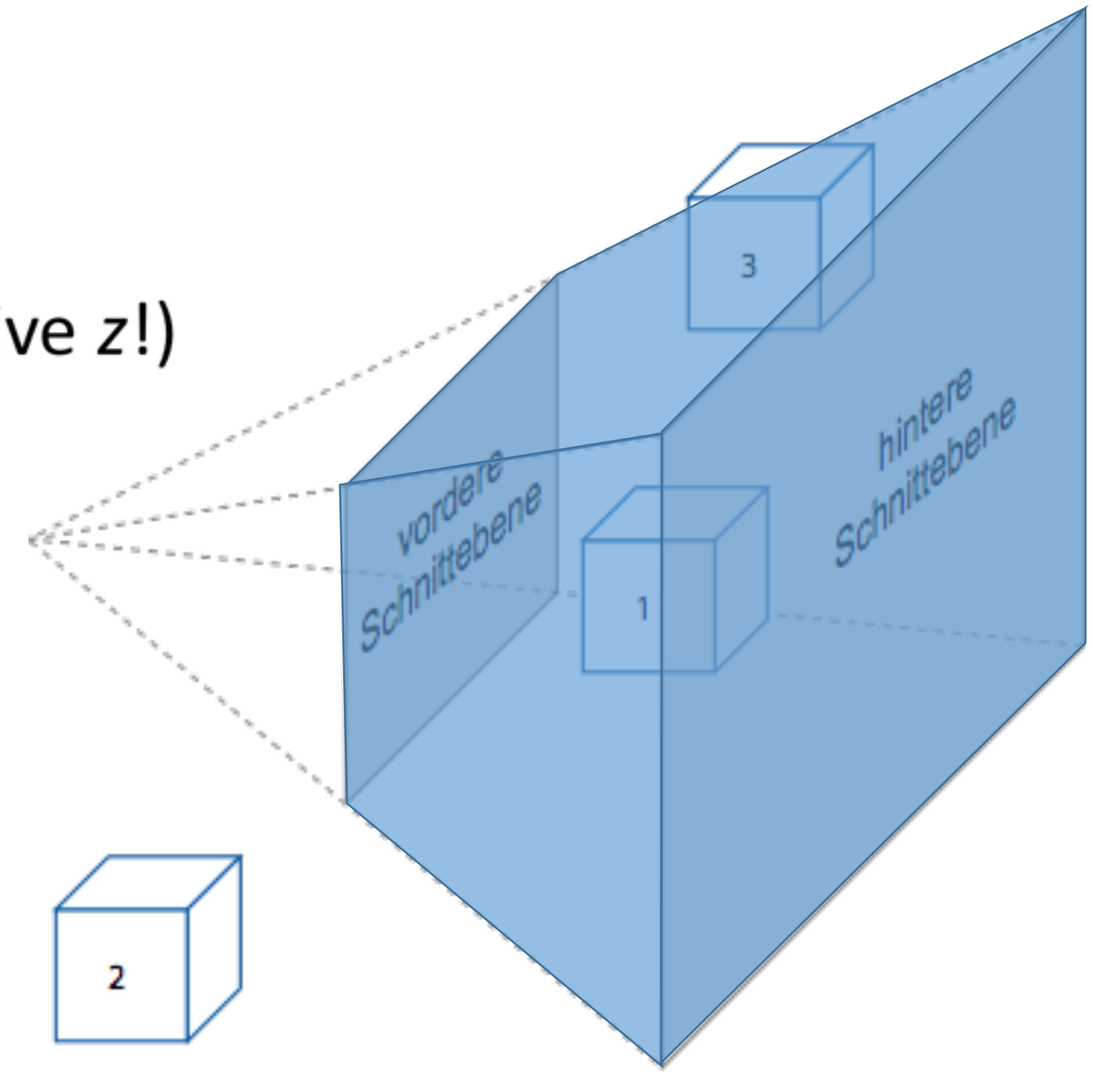
View Frustum Culling

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

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

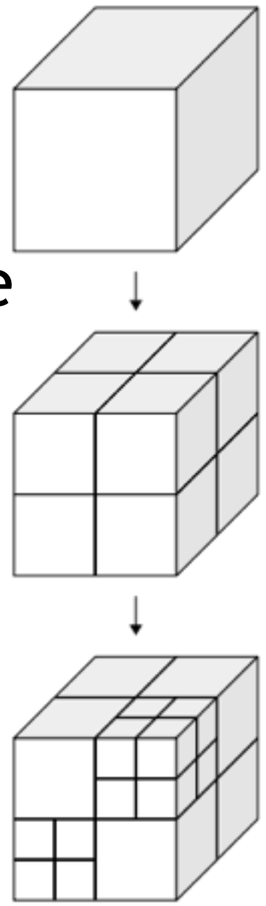
- X- and Y-Axis: inside the viewing cone
- $-w_{view} < x_{view} < w_{view}$
- $-h_{view} < y_{view} < h_{view}$

→ Two simple comparisons for each axis!

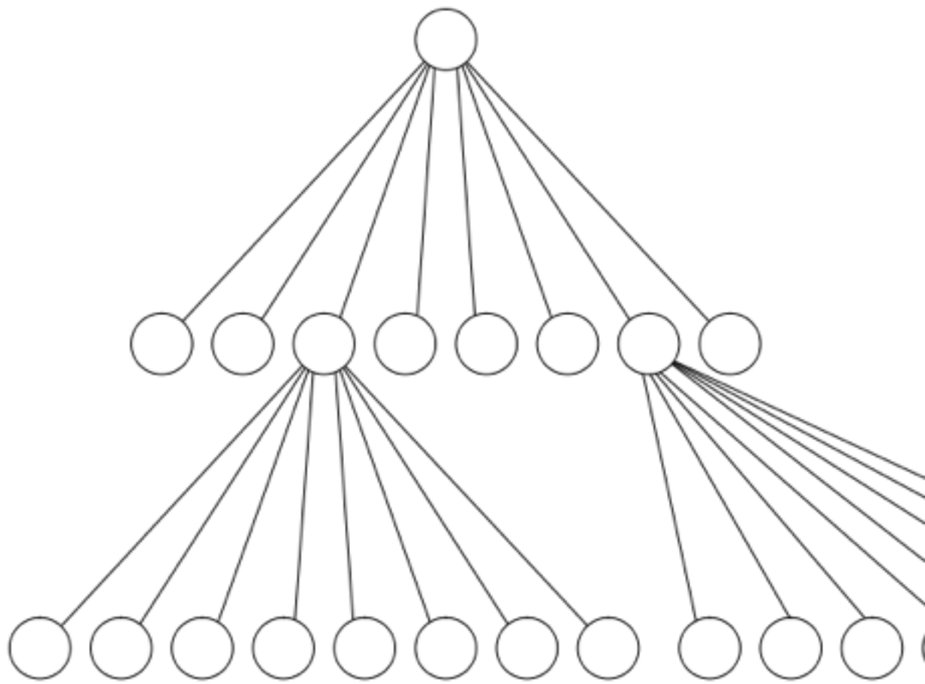


Optimization: Octrees Speed up View Frustum Culling

- Naive frustum culling needs *O(n)* tests
  - Where *n* = number of objects



- Divide entire space into 8 cubes
  - See which objects are inside each cube
- Subdivide each cube again
  - Repeat recursively until cube contains less than k objects



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

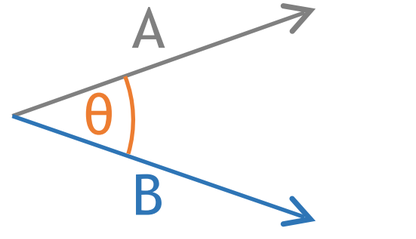
- Instead of culling objects, cull cubes
- Needs *O(log n)* tests

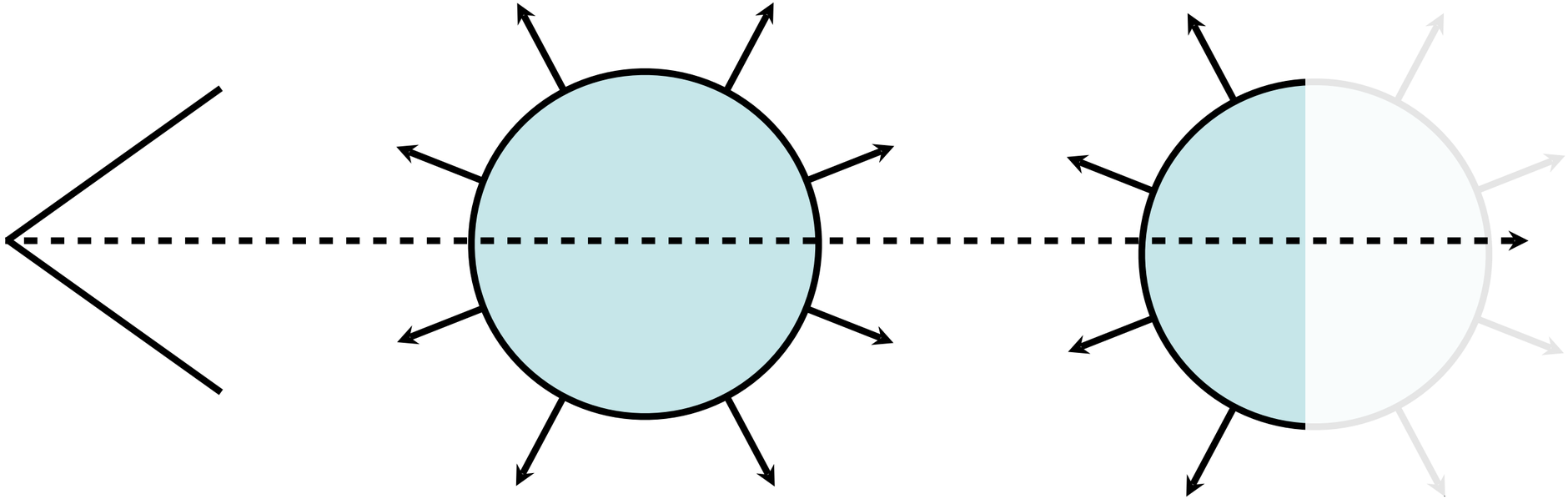
- *There are more space-partitioning data structures (kd-/BSP-tree, bounding volume hierarchy,…)*

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
  - Normals are often stored in face mesh structure,
  - Otherwise can be computed as cross product of 2 triangle edges
  - Normal faces backwards if angle with optical axis is < 90° (i.e. scalar product > 0)

$$A \cdot B = |A| \cdot |B| \cos\theta$$

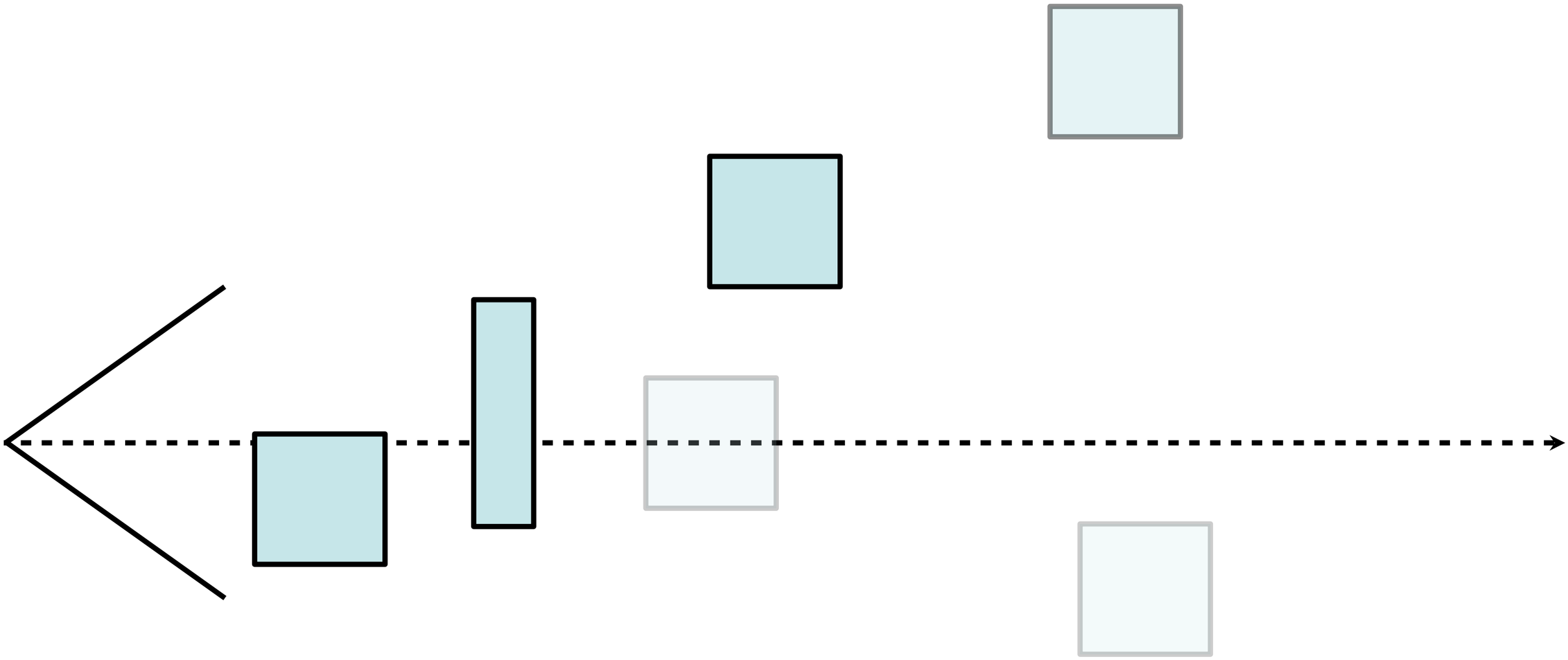




→ *Alternative: Use vertex order (winding, clockwise or counter-clockwise)!*

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 5 – 3D Camera & Rasterization

- Classical Viewing Taxonomy
- 3D Camera Model
- Optimizations for the Camera
- How to Deal with Occlusion
- Rasterization
  - Clipping
  - Drawing lines
  - Filling areas

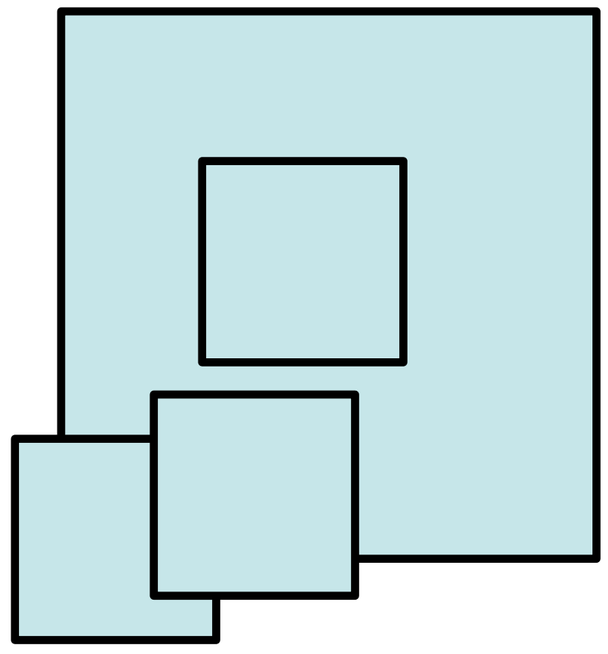
Occlusion: The Problem Space in General

- Need to determine which objects occlude which others objects
- 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?

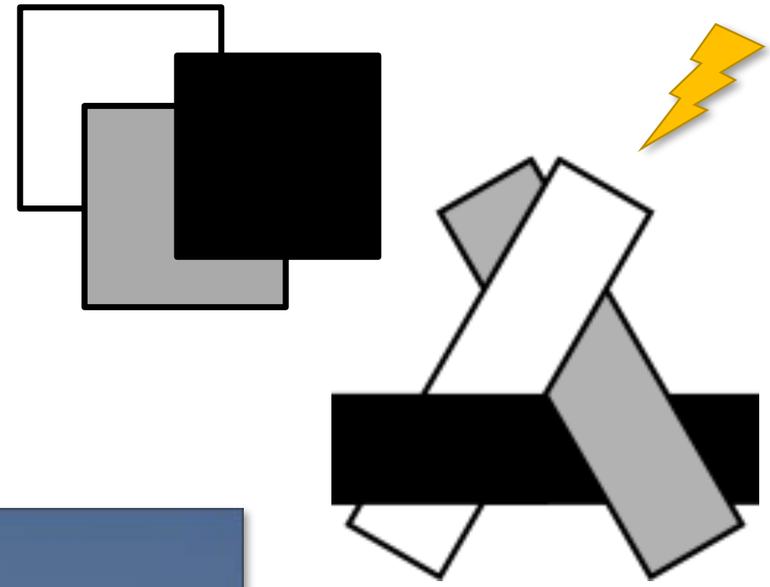
- Occlusion is an important depth cue for humans
  - Need to get this really correct!



Occlusion – Simple Solution: Depth-sort + ordered rendering

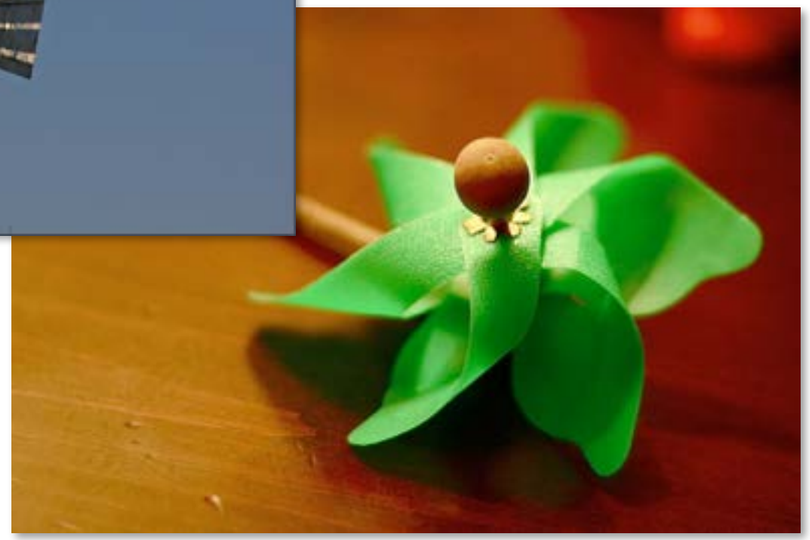
- Regularly used in 2D vector graphics
- Sort polygons according to their z position in view coordinates
- Draw all polygons from back to front
  - Back polygons will be overdrawn
  - Front polygons will remain visible

→ "Painter's algorithm"



- Problem 1: Self-occlusion
  - Not a problem with triangles

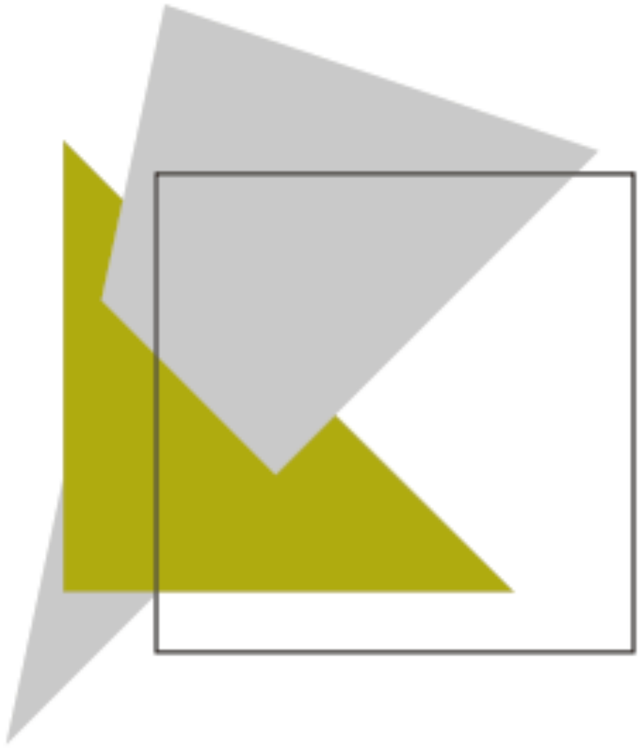
- Problem 2: Circular occlusion
  - Think of a pin wheel!



http://view.stern.de/de/original/1732381/Windrad-Gozo-Malta-Windbrunnen-windrad-Blau-Technik--Verkehr.jpg

Occlusion – Better Solution: Z-Buffer

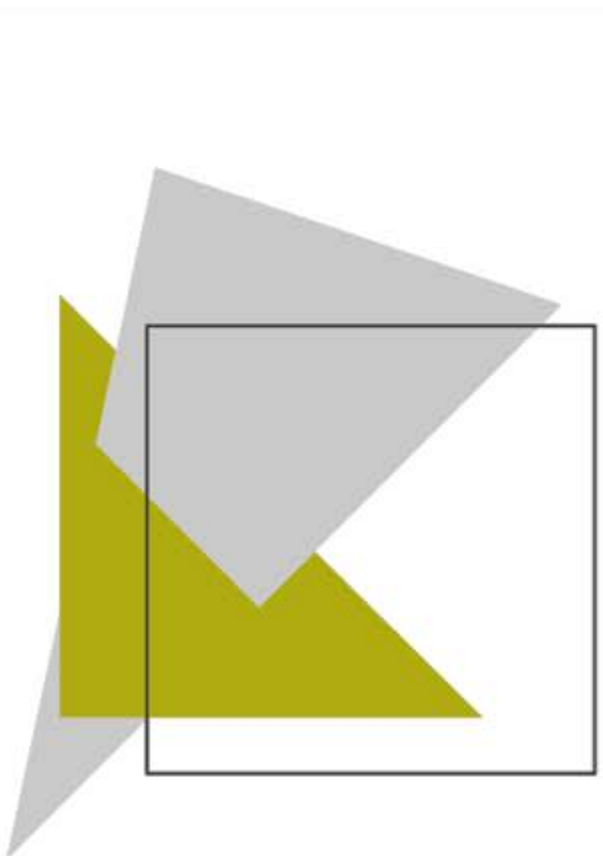
- ***Idea:*** Compute depth not per polygon, but per pixel!
- Approach: for each pixel of the rendered image (frame buffer) keep also a depth value (Z-buffer)
- Initialize the Z-buffer with z_{far}, which is the far clipping plane and hence the furthest distance we need to care about
- Loop over all 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

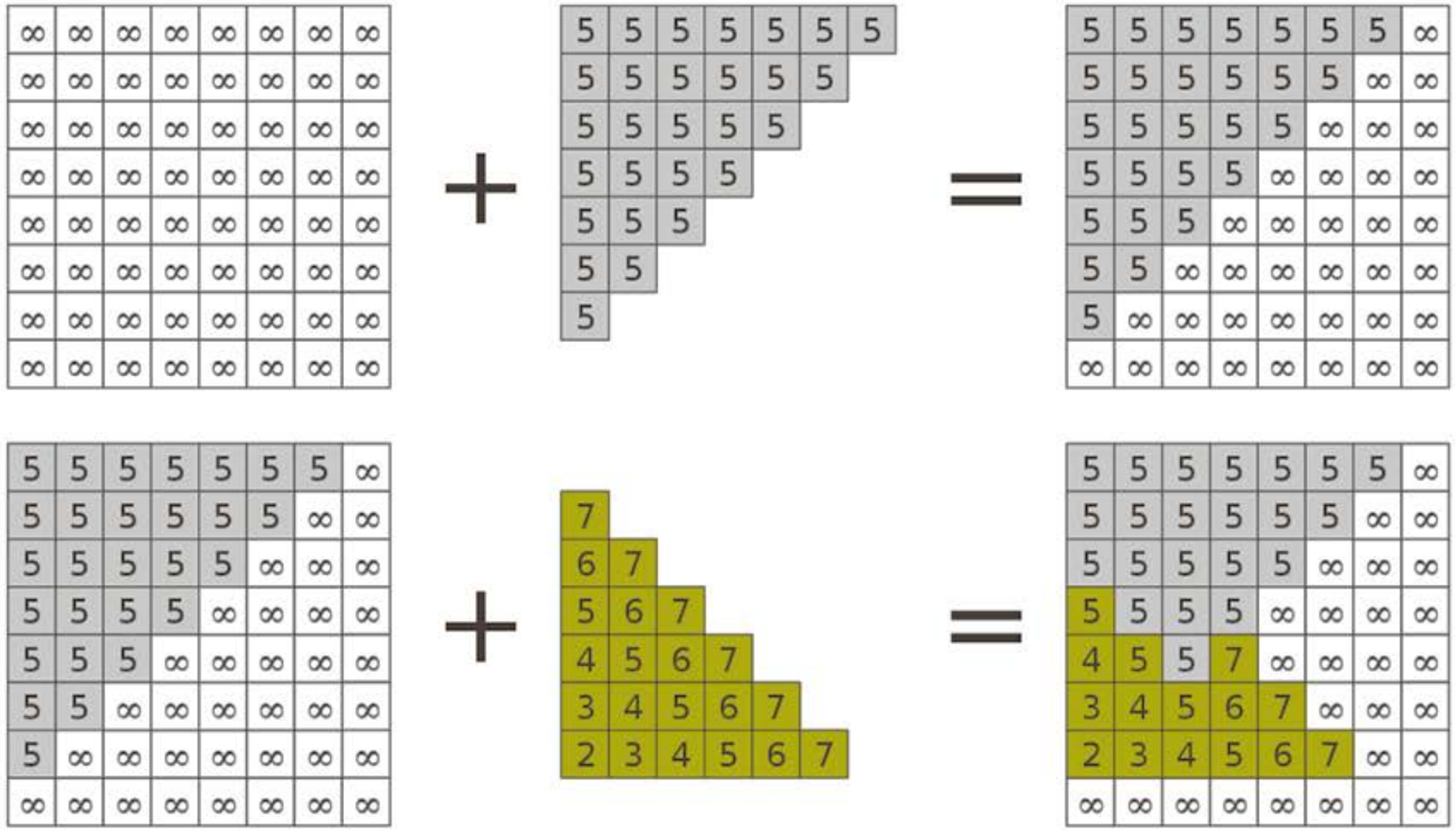


http://de.wikipedia.org/w/index.php?title=Datei:Z-buffer.svg

Z-Buffer Example

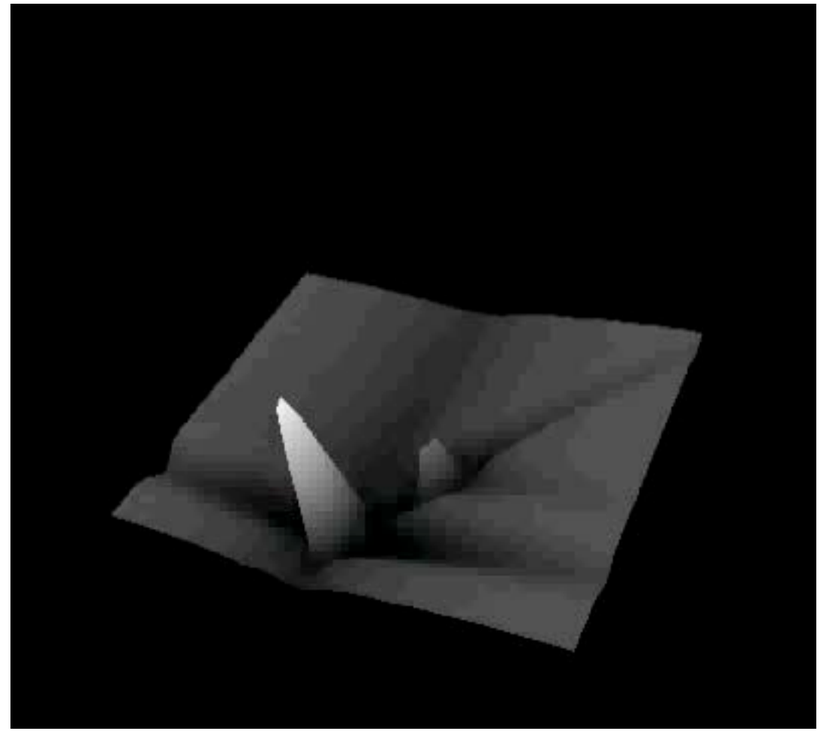
→ In contrast to OpenGL, this example uses positive z-values (and thus tests for z <= Z-buffer-value)!





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)



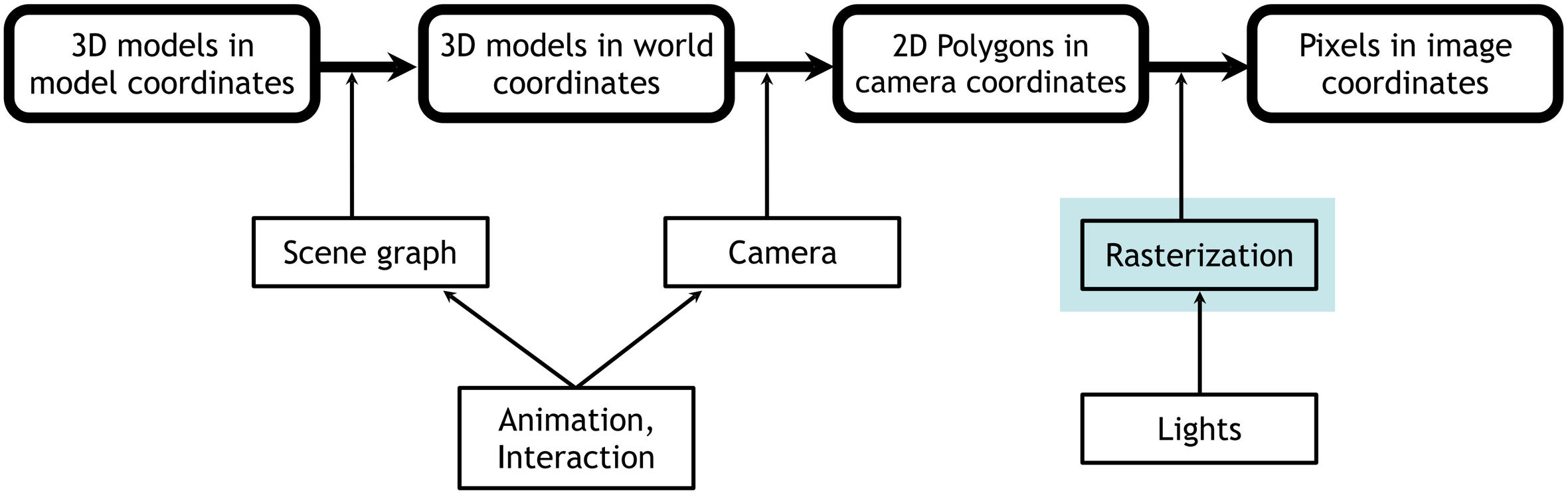
http://www.youtube.com/watch?v=TogP1J9iUcE

- 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 re-render the objects you want to see through these holes

Chapter 5 – 3D Camera & Rasterization

- Classical Viewing Taxonomy
- 3D Camera Model
- Optimizations for the Camera
- How to Deal with Occlusion
- Rasterization
  - Clipping
  - Drawing lines
  - Filling areas

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

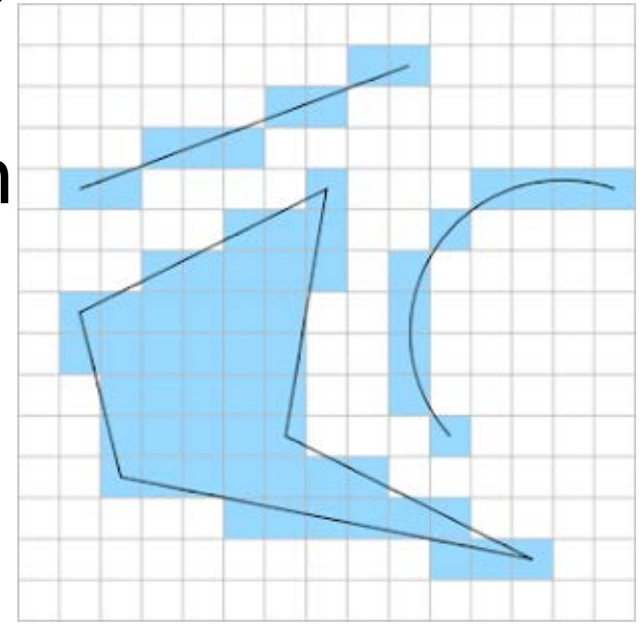


Rasterization: The Problems

- ***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?

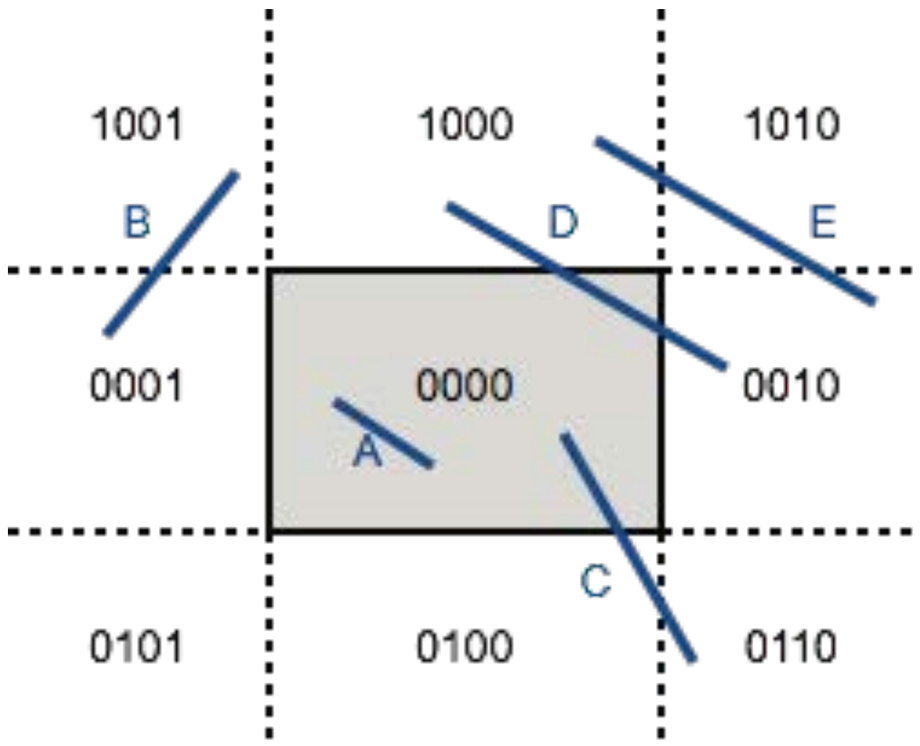
- ***This is done automatically by the GPU!***
- *Part of this will be needed again later in the semester for the shading/rendering chapter!*



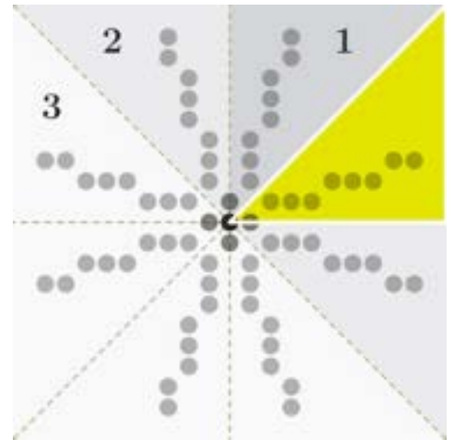
http://iloveshaders.blogspot.de/2011/05/how-rasterization-process-works.html

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 (bit-wise operations=:
  - **`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 (Ex. 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



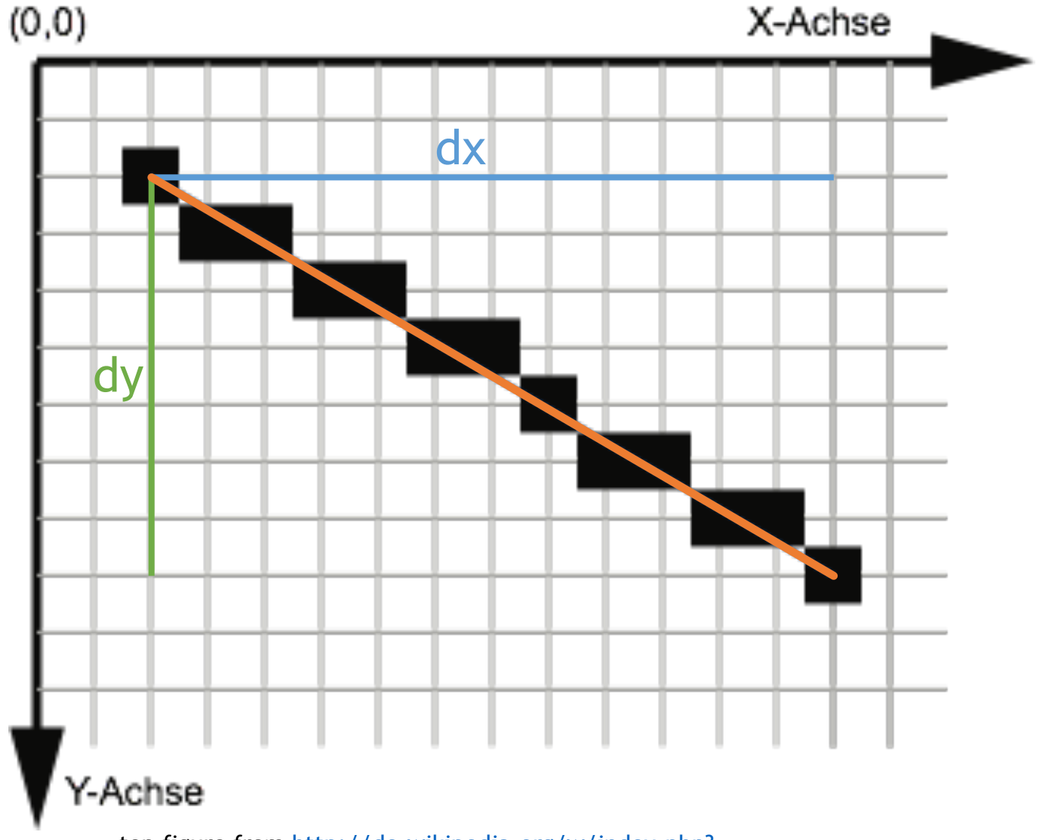
Drawing a Line: Naïve Approach



- 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 swap x and y
  - Loop over y and compute x

```
for (x from 0 to dx) {
    setPixel (x1 + x, y1 + m * x)
}
```

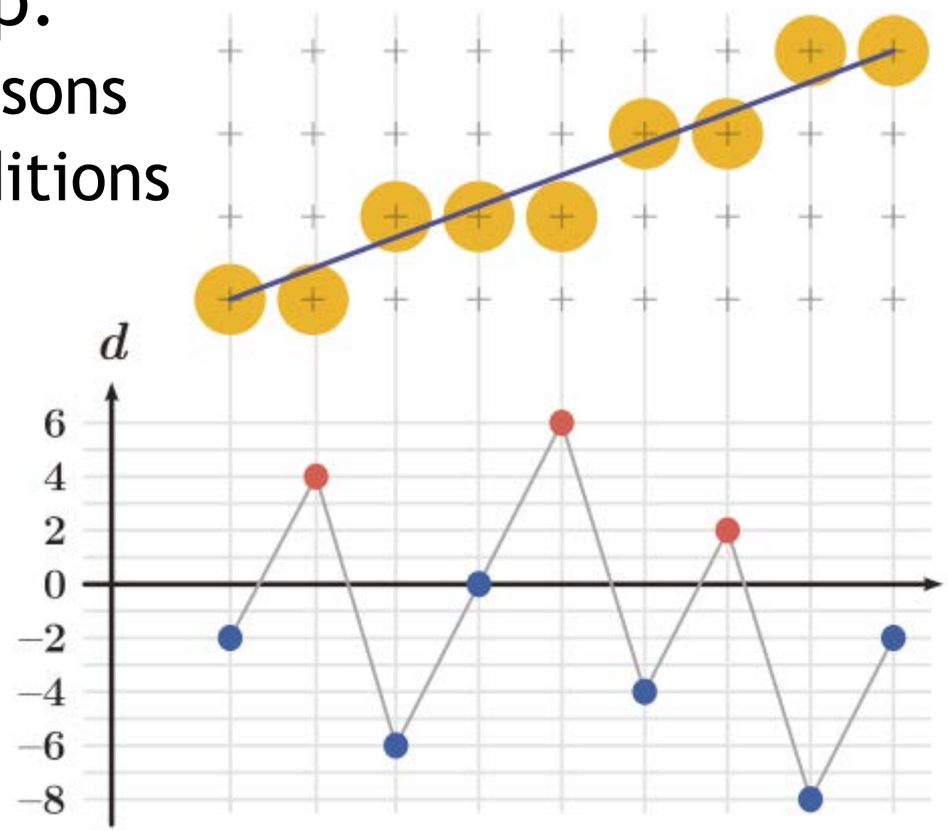
- In each step:
  - 1 float multiplication, 2 additions
  - 1 round to integer



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
  - 2 or 3 additions



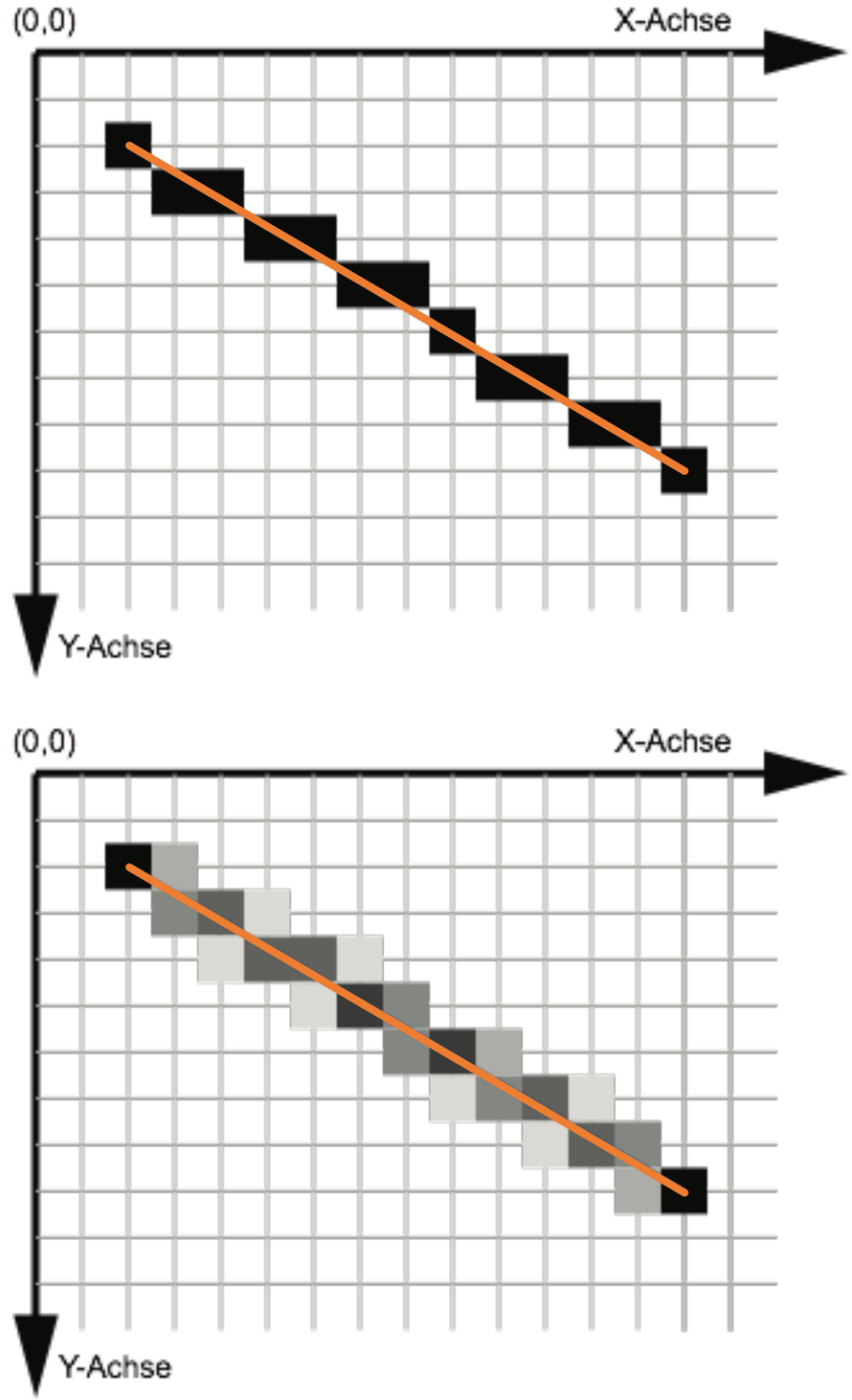

```
dx  := x2-x1;
dy  := y2-y1;
d   := 2*dy – dx;
DO  := 2*dy;
dNO := 2*(dy - dx);
x := x1;
y := y1;
setpixel (x,y);
fehler := d;
WHILE ( x < x2 ) {
        x := x + 1;
        IF ( fehler <= 0 ) {
                fehler := fehler +
DO
        } ELSE {
                y := y + 1;
                fehler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

Antialiased Lines

- ***Problem:*** Bresenham‘s lines contain visible steps (aliasing effects)
- Opportunity: we can usually display at least 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

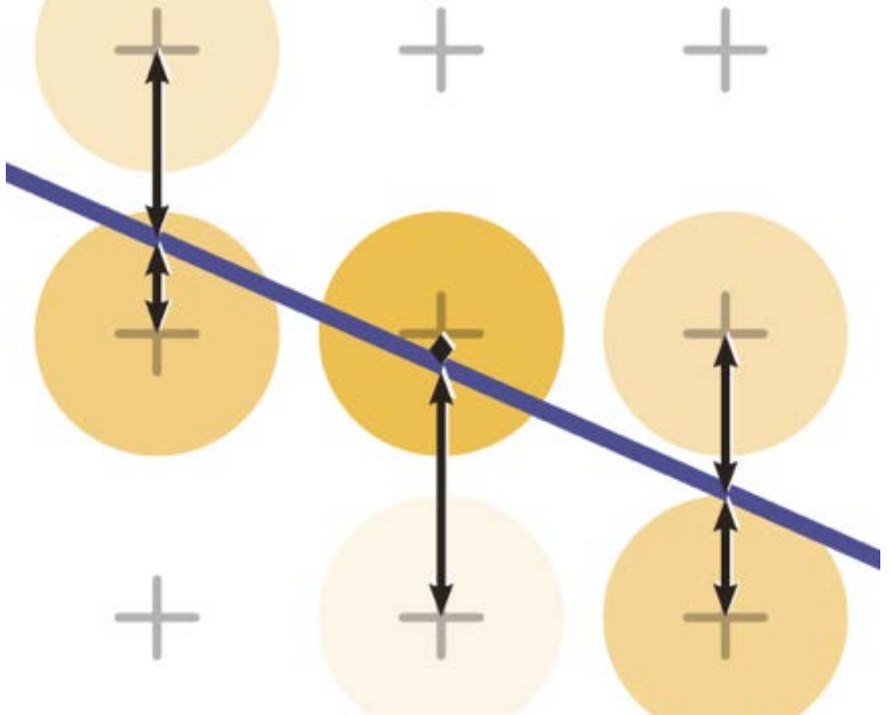
- Different algorithms exist
  - Gupta-Sproull for 1 pixel wide lines
  - Wu for infinitely thin lines

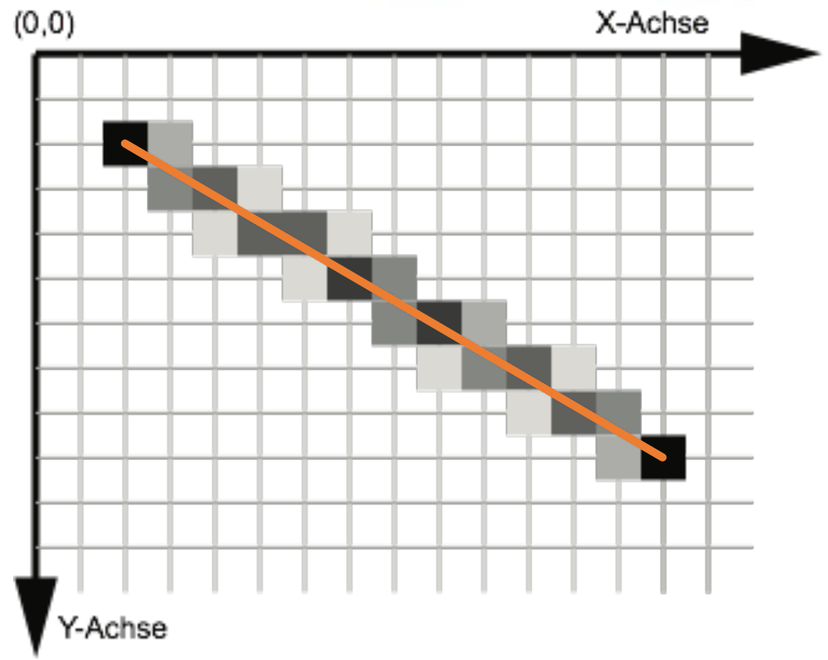


Wu‘s Antialiasing Approach

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

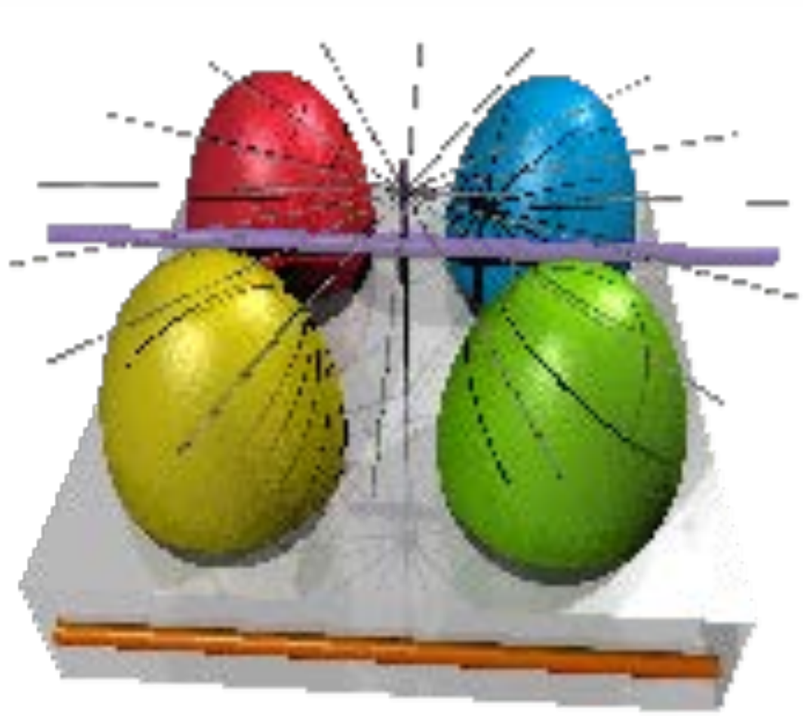
http://de.wikipedia.org/w/index.php?title=Datei:Wu_line_drawing.svg





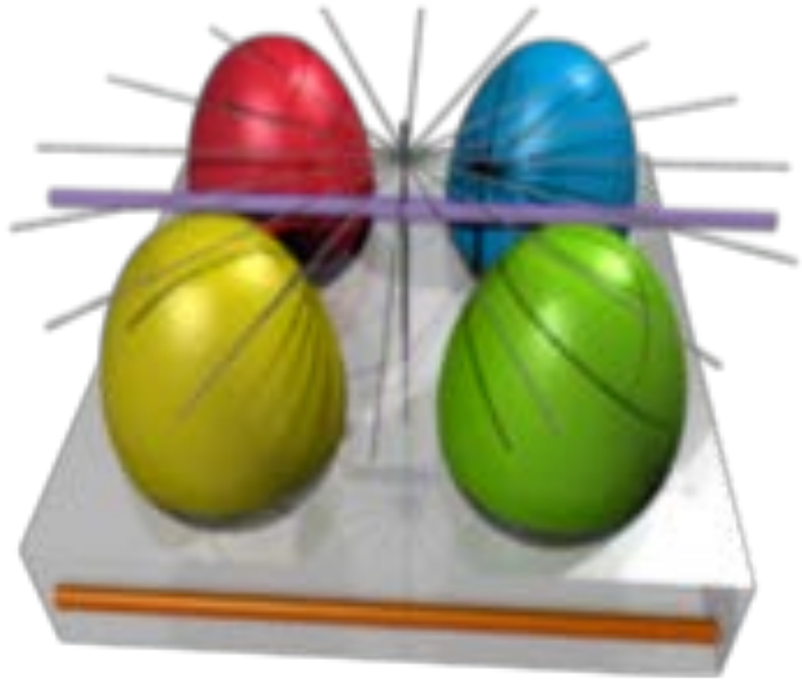
Antialiasing in General

- ***Problem:*** Hard edges in computer graphics
- Correspond to infinitely high spatial frequency
- Violate sampling theorem (Nyquist, Shannon)
  - Re-read 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

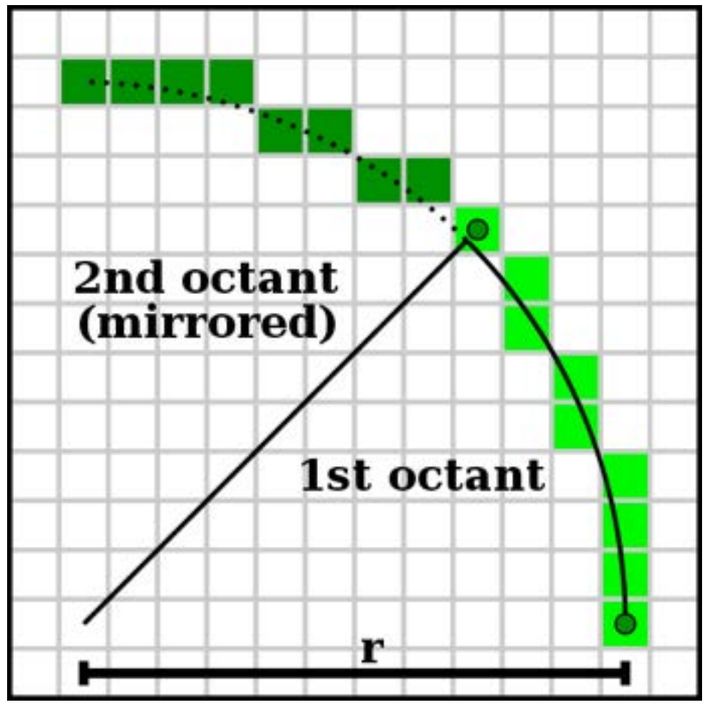
→ *Supported in OpenGL*



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

Line Drawing: Summary

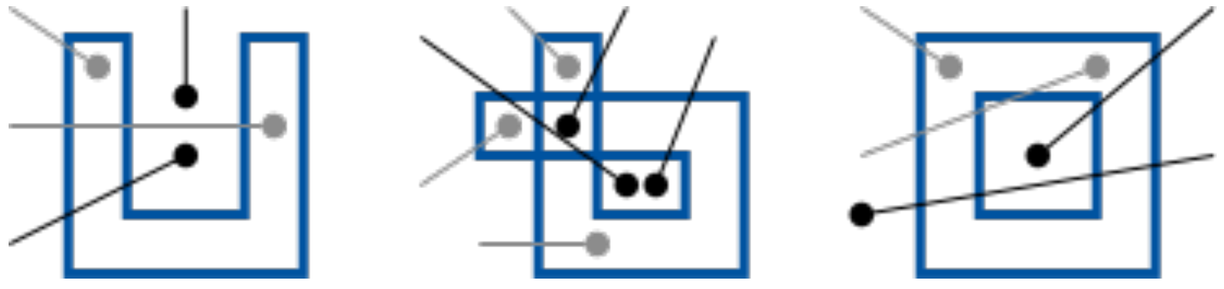
- With culling and clipping, we made sure all lines are inside the image
- With algorithms so far we can draw lines in the image
  - Even antialiased lines directly
- This means we can draw arbitrary polygons now (in black and white)
- All algorithms extend to color
  - Just modify the `setPixel(x,y)` 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



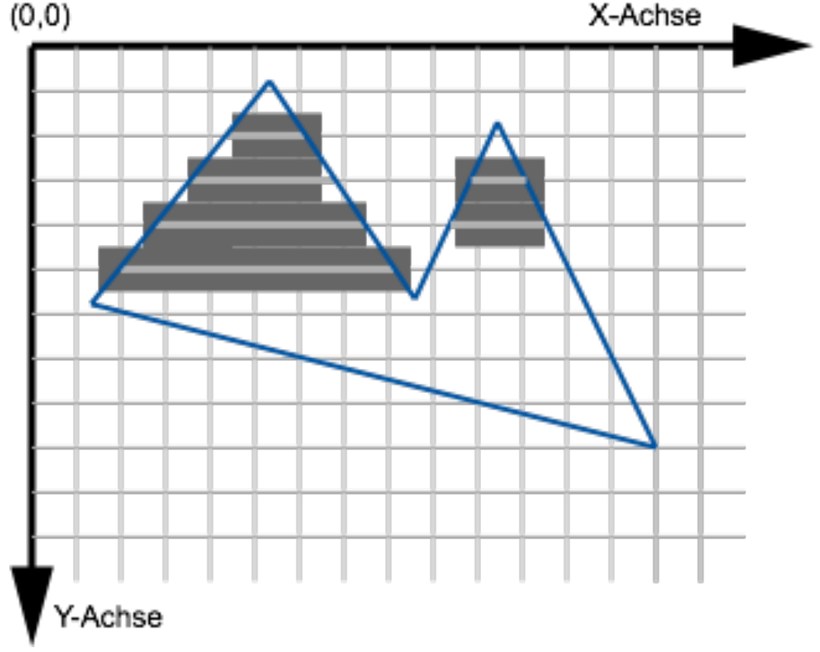
http://en.wikipedia.org/wiki/File:Bresenham_circle.svg

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