Chapter 5 - 3D Camera & Optimizations, Rasterization

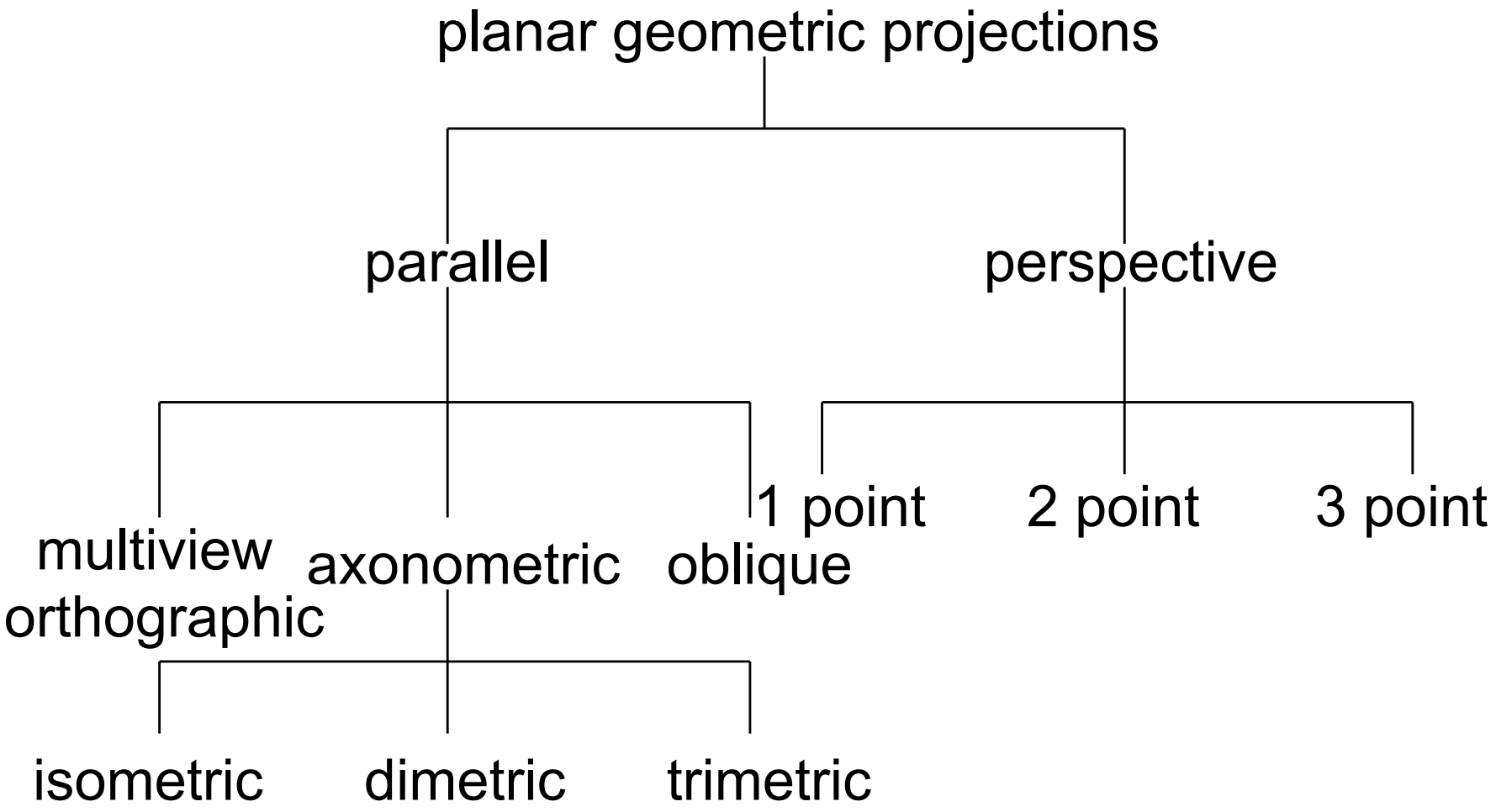
- Classical Viewing Taxonomy
- 3D Camera Model
- Optimizations for the Camera
- How to Deal with Occlusion
- Rasterization
  - Clipping
  - Drawing lines
  - Filling areas

Based on material from:
E. Angel and D. Shreiner : Interactive Computer Graphics.
6th ed, Addison-Wesley 2012

Classical Views of 3D Scenes

- As used in arts, architecture, and engineering
  - Traditional terminology has emerged
  - Varying support by 3D graphics software and hardware
- Assumptions:
  - Objects constructed from flat principal faces (polygons)
  - Projection surface is a flat plane
    - Nonplanar projections are important: Maps for long distance navigation for instance
- General situation:
  - Scene consisting of 3D objects
  - Viewer with defined position and projection surface
  - *Projectors (Projektionsstrahlen)* are lines going from objects to projection surface
- Main classification:
  - Parallel projectors or converging projectors

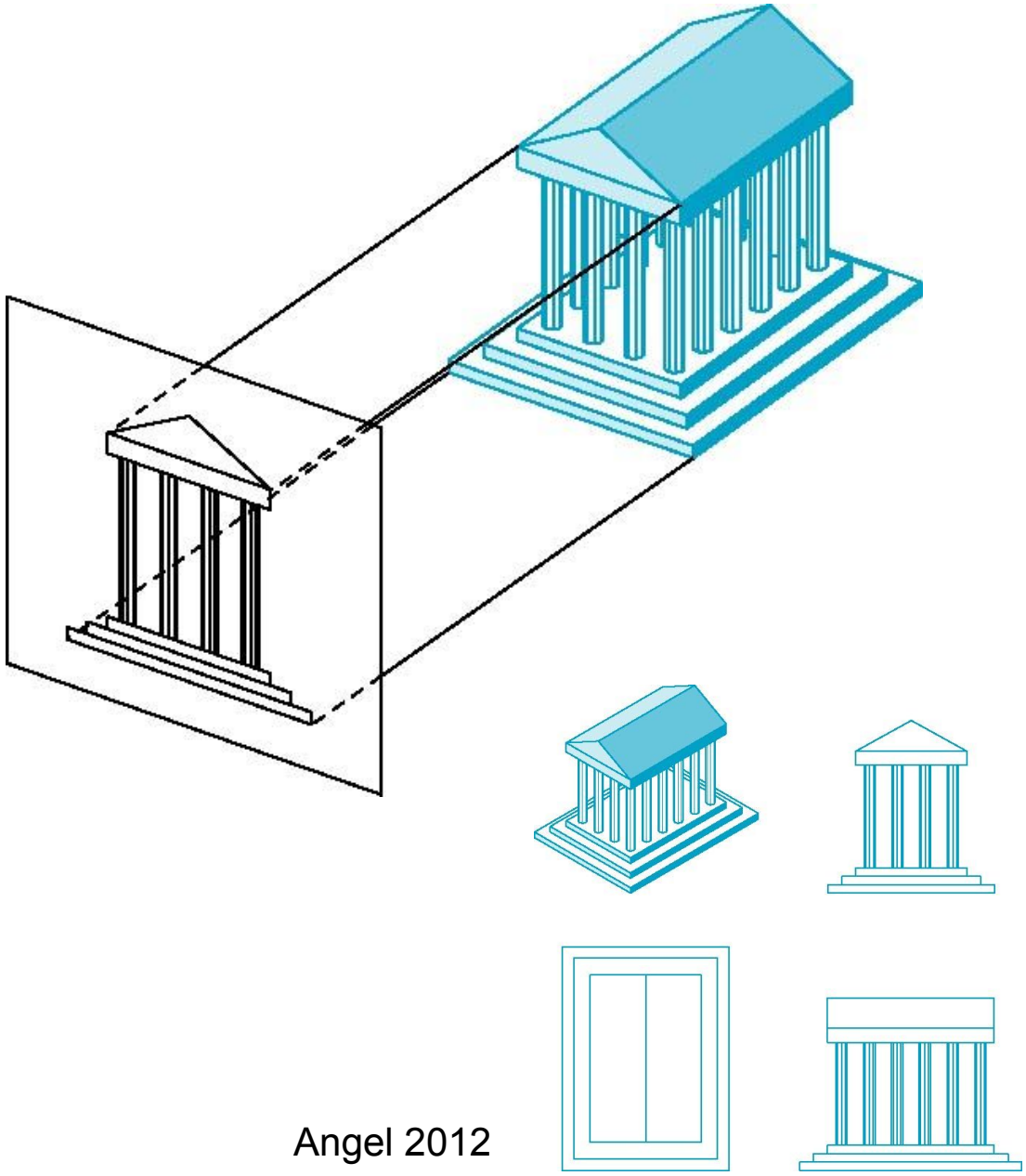
Taxonomy of Views



Angel 2012

Orthographic Projection

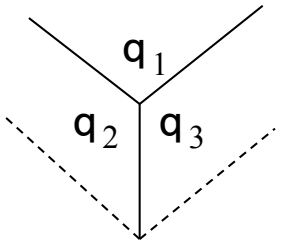
- Projectors are orthogonal to the projection plane
- In the “pure” case, projection plane is parallel to a coordinate plane
  - top/front/side view
  - Often used as a multi-view combination
  - together with overview (e.g. isometric view)
- Advantage:
  - No distortions
  - Can be used for measurements

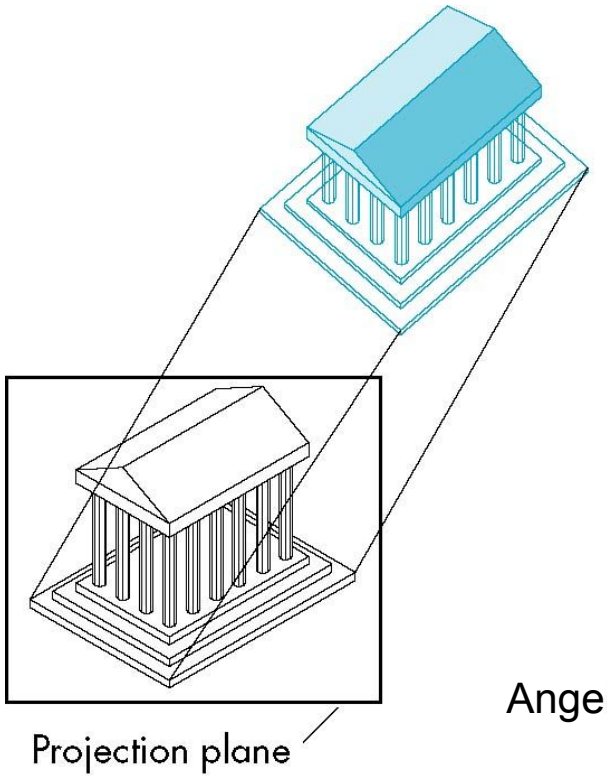


Angel 2012

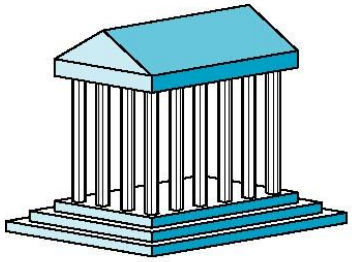
Axonometric Projections

- 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*
- Advantage:
  - Preserves lines
  - Somehow realistic
- Disadvantage:
  - Angels not preserved

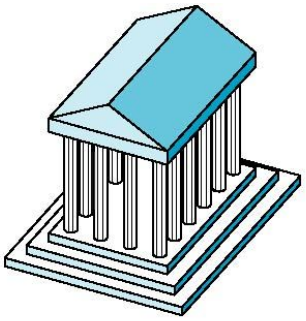




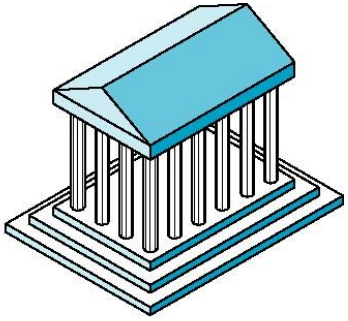
Angel-2012



Dimetric

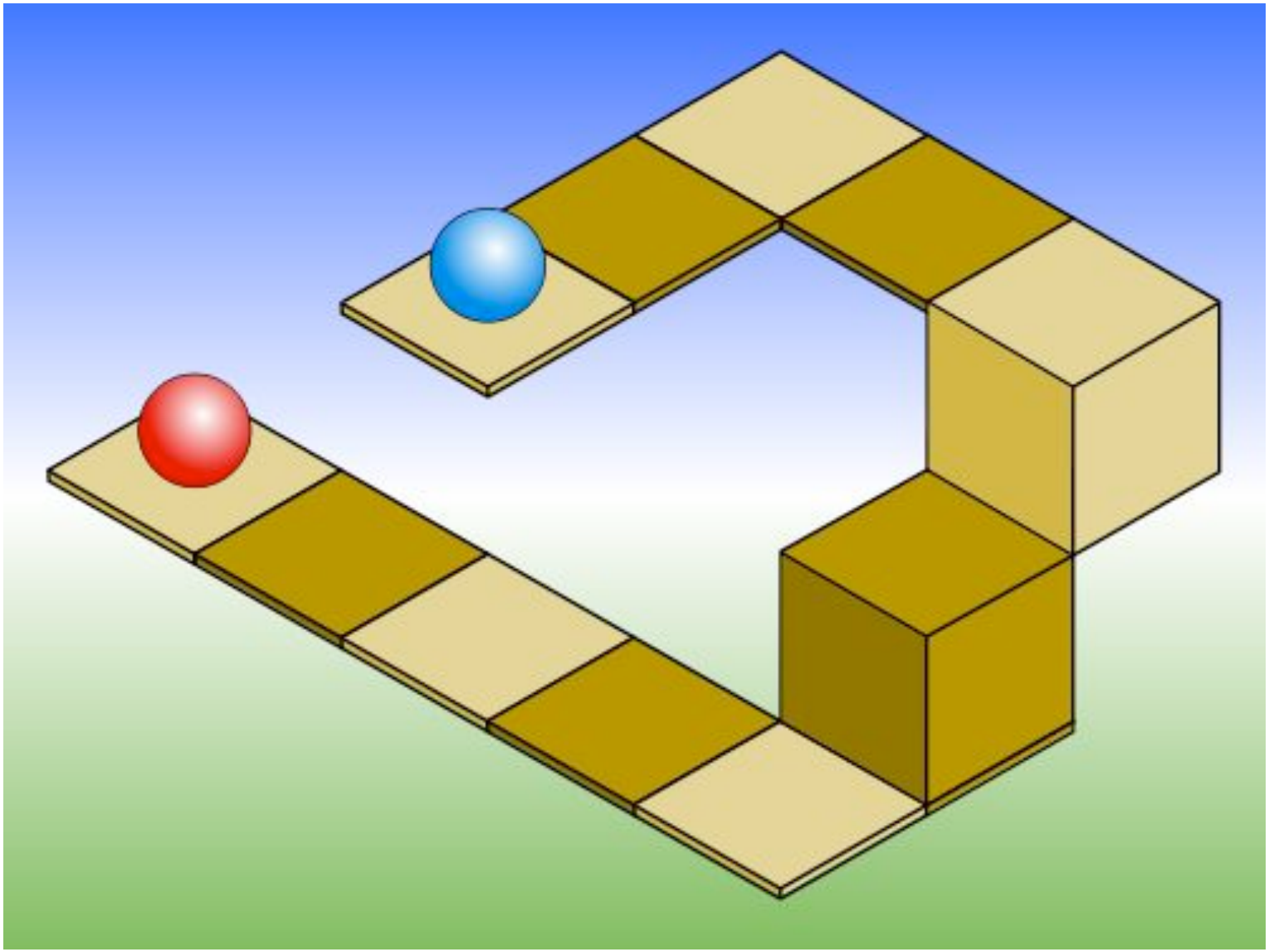


Trimetric



Isometric

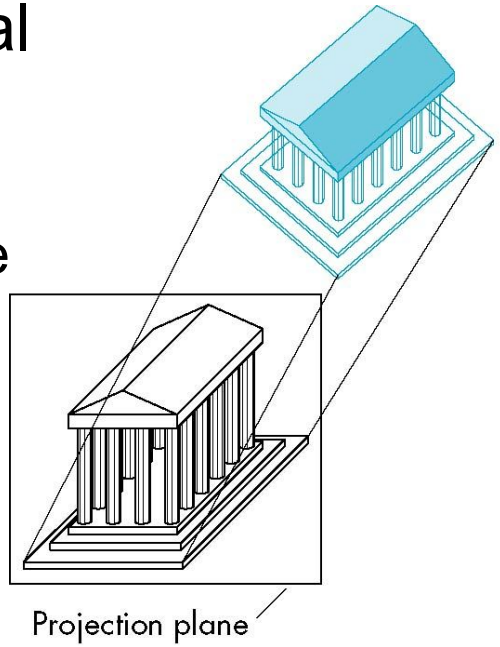
Optical Illusions in Isometric Projections



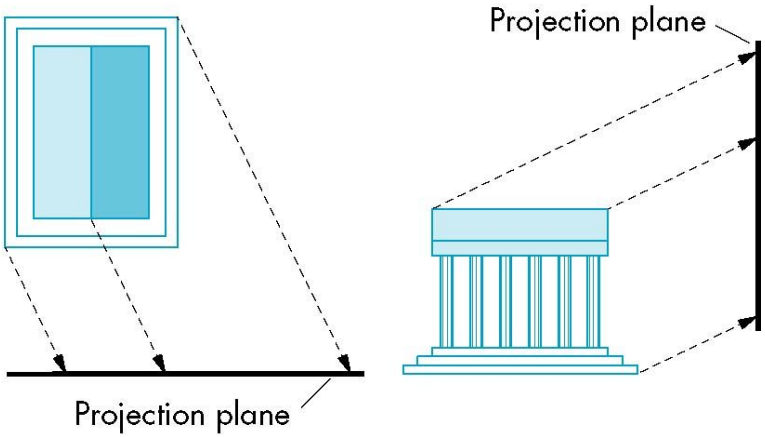
Source:
Wikipedia

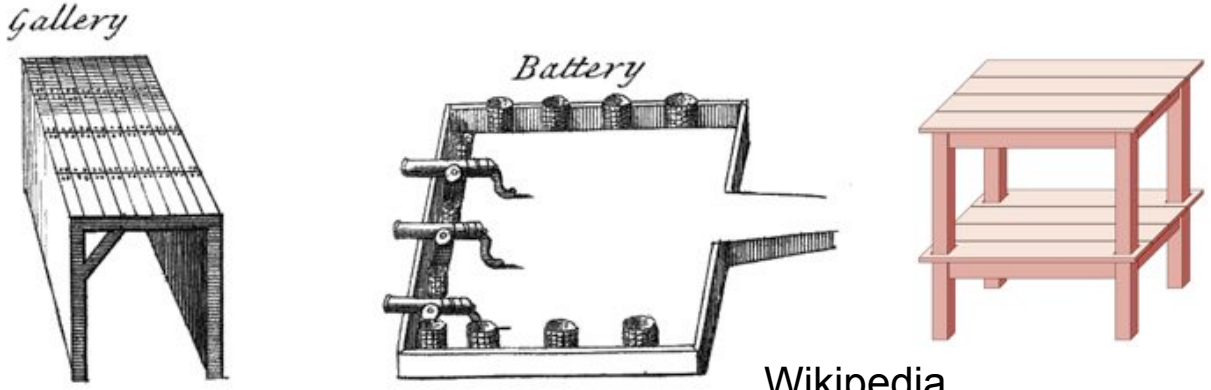
Oblique Projection *(Schiefe Parallelprojektion)*

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



Angel 2012

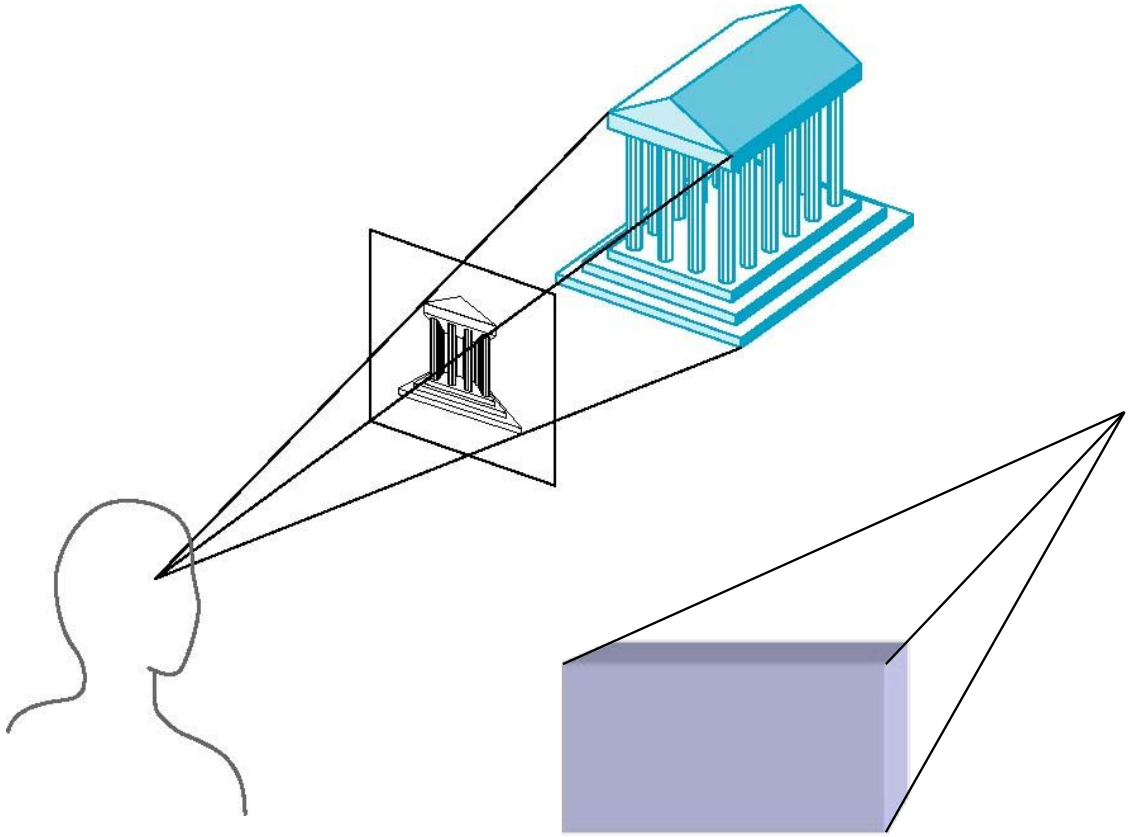




Wikipedia

Perspective Projection *(Perspektivische Projektion)*

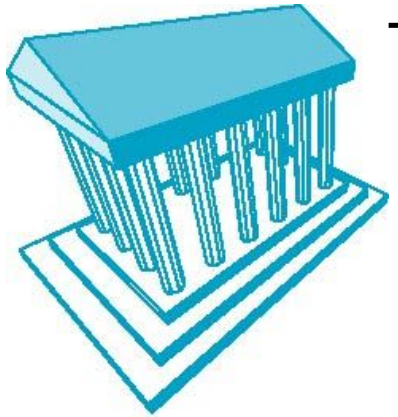
- Projectors converge at *center of projection (cop)*
- Parallel lines (not parallel to projection plane) appear to converge in a *vanishing point (Fluchtpunkt)*
- Advantage:
  - very realistic
- Disadvantage:
  - non-uniform foreshortening
  - only few angles preserved



Number of Vanishing Points

One point

Two points



Three points

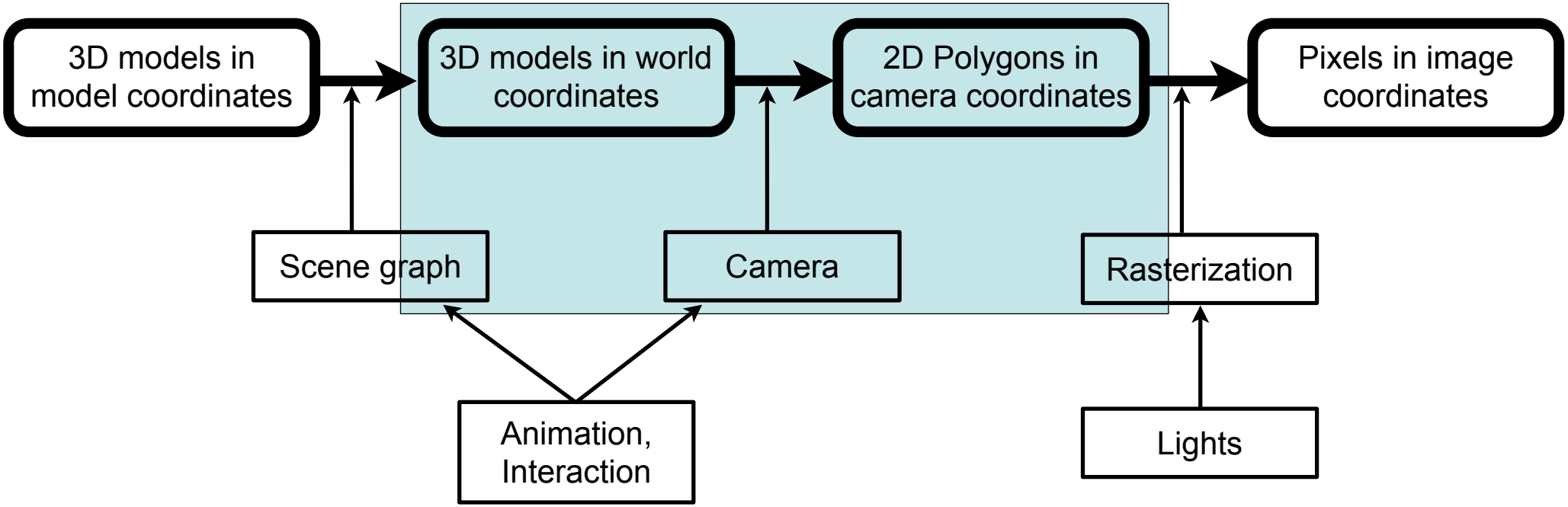
How to Realize Projection in OpenGL?

- Parallel projections:
  - Orthographic projections, including axonometric projections:
    - `gluLookAt()`, projection matrix, affine transformations
  - Oblique transformations:
    - not directly supported
    - can be simulated by using a shearing operation
- Perspective projections:
  - Directly supported in OpenGL (two ways)
  - Mathematically easy to map onto existing concepts

Chapter 5 - 3D Camera & Optimizations, Rasterization

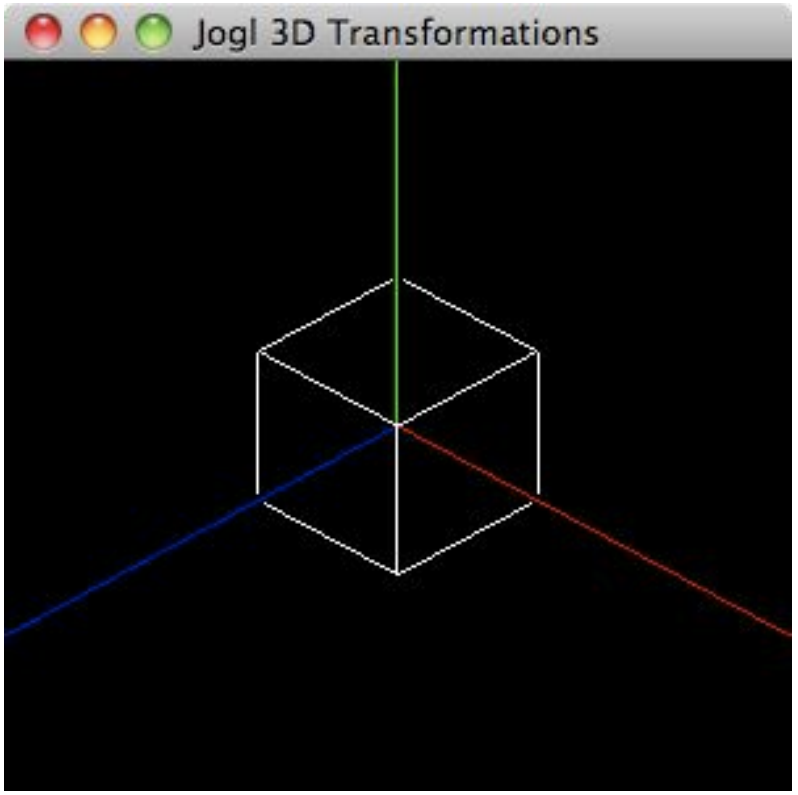
- Classical Viewing Taxonomy
- 3D Camera Model
- Optimizations for the Camera
- How to Deal with Occlusion
- Rasterization
  - clipping
  - drawing lines
  - filling areas

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



Isometric View of a Cube with OpenGL

```
glu.gluLookAt(0.5, 0.5, 0.5, 0, 0, 0, 0, 1, 0);


gl.glScaled(0.25, 0.25, 0.25);
gl.glColor3d(1, 1, 1); //draw in white

gl.glBegin(GL2.GL_LINE_LOOP); // draw front side
   gl.glVertex3d(-1, -1, 1);
   gl.glVertex3d(1, -1, 1);
   gl.glVertex3d(1, 1, 1);
   gl.glVertex3d(-1, 1, 1);
gl.glEnd();
...// draw back side and connecting planes
```



Q: What is the default projection used in OpenGL?
Q: How did we choose an isometric view?

Default Orthographic Projection: (Conceptual) Matrix

- We can restrict ourselves to looking down the negative z axis
  - Camera position (and axonometric view) have already been dealt with by coordinate transformations
  - Modelview matrix, `lookAt()` function
- Therefore, projection matrix (conceptually) simply wipes out the z value
  - In practice, it is better to keep the z value for optimizations (see later)

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Conceptual Matrix

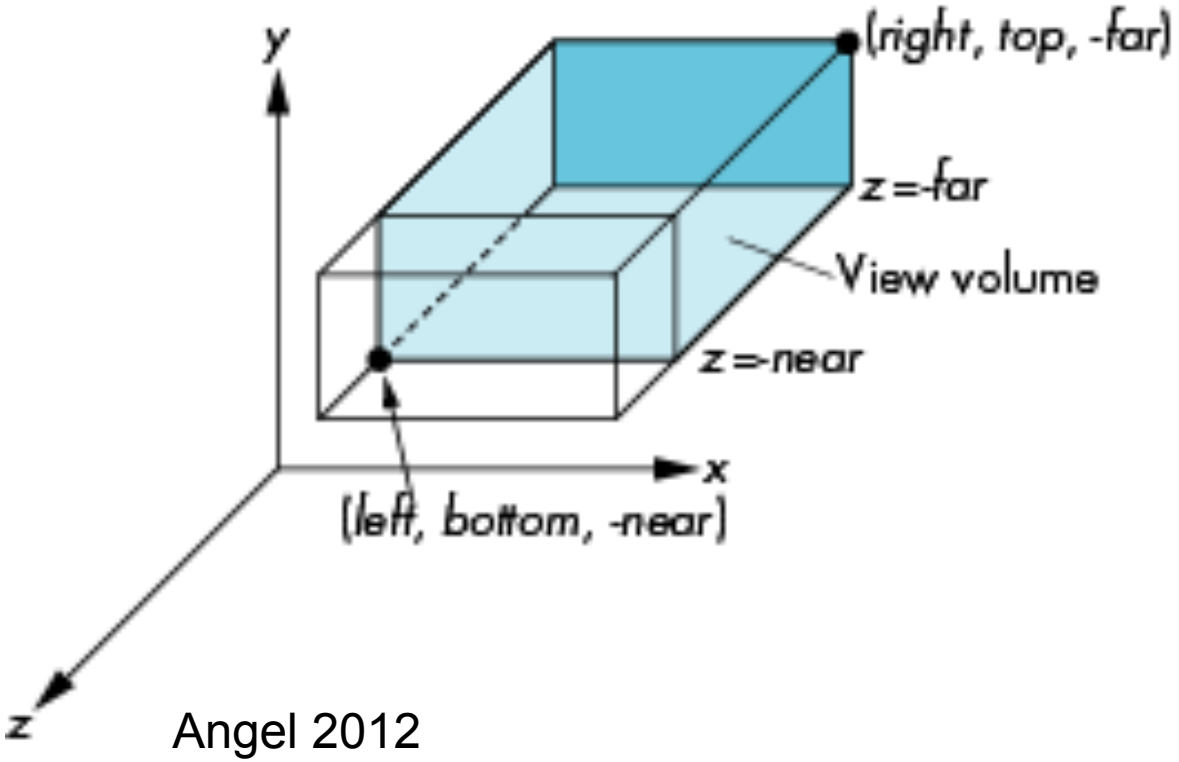
```
Projection matrix (Ortho)
  1.000   0.000   0.000   0.000
  0.000   1.000   0.000   0.000
  0.000   0.000   1.000   0.000
  0.000   0.000   0.000   1.000
```

JOGL Printout

Q: Why did we have to scale the cube to small proportions?
Q: Why was the camera position using coordinate values of 0.5 (and not 1)?

Orthographic Projection: Viewing Volume

- Analogous to camera lens:
  - Only part of the scene is represented, rest is “clipped”
  - Orthographic projection: Rectangular viewing volume
  - In OpenGL, defined by `Ortho*` function



Angel 2012

Default values in OpenGL (no `Ortho*` call):

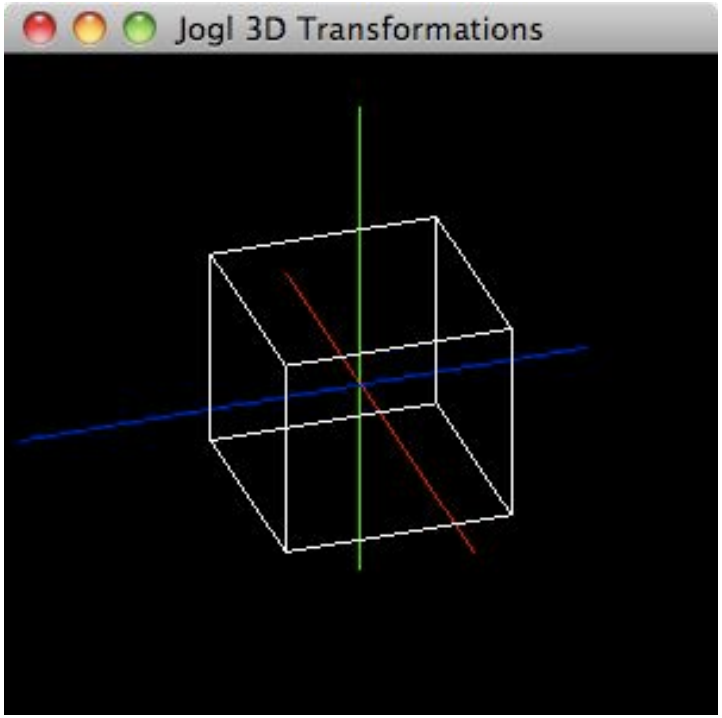
left = bottom = near = –1
right = top = far = +1

Normalization in Orthographic Projection

```
glu.gluLookAt(3, 2, 1, 0, 0, 0, 0, 1, 0);
… and the cube is unit cube around origin

gl.glOrthof(-3, 3, -3, 3, 0, 100);
```

- User-specified viewing volume is translated into default volume
  - Applying affine transformations, i.e. matrix multiplications
  - Translation of volume center to origin *(T)*
  - Scaling to side lengths of 2 *(S)*

$$ST = \begin{pmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{2}{near-far} & -\frac{near+far}{far-near} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

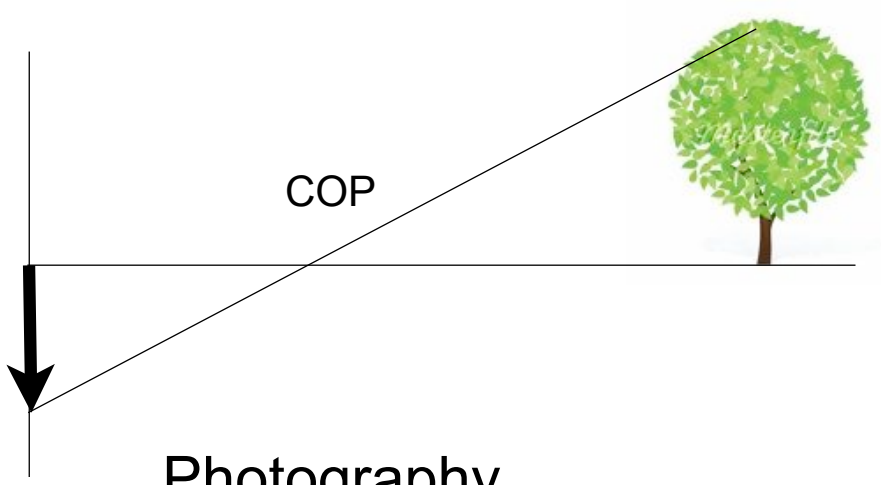



```
Projection matrix (Ortho)
  0.333   0.000  -0.000   0.000
  0.000   0.333  -0.000   0.000
  0.000   0.000  -0.020  -1.000
  0.000   0.000  -0.000   1.000
```

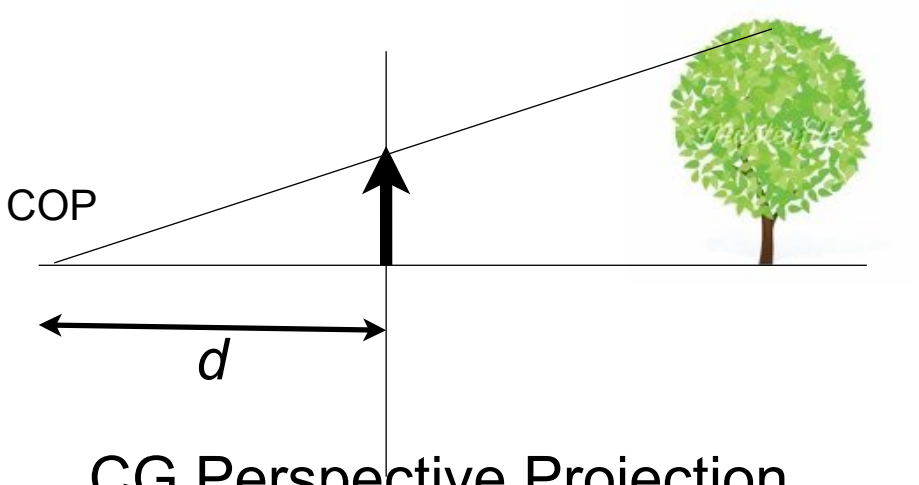
Q: How do we call this kind of view?

Perspective Projection and Photography

- In photography, we usually have the *center of projection (cop)* in between the object and the image plane
  - Image is upside down
- In most cases of CG perspective projection, the image plane is in front of the camera!

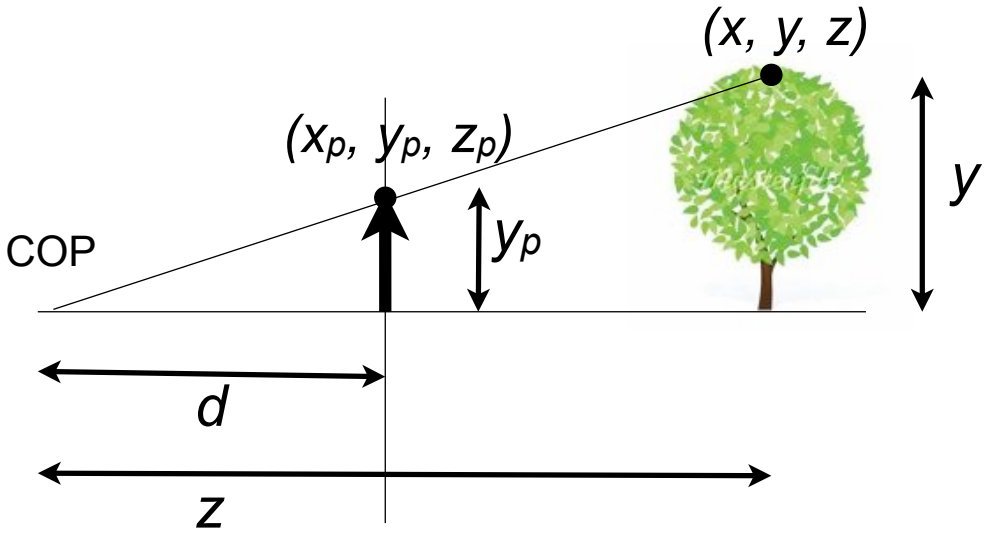


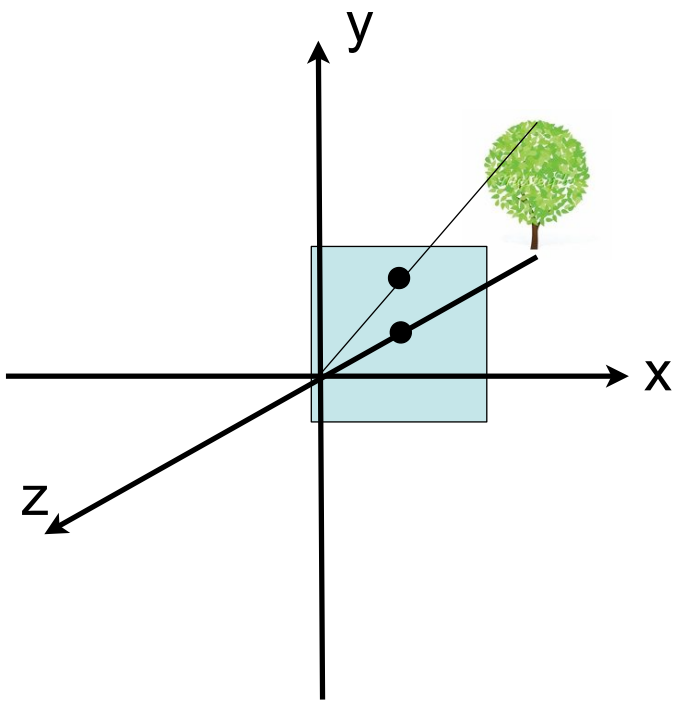
Photography



CG Perspective Projection

Simple Perspective Projection





Let *(x, y, z)* be an arbitrary point in space (e.g. the tree top).
Let *(x_p, y_p, z_p)* be the projection of this point to the image plane (parallel to x/y).
Let *d* be the distance of the image plane from the origin (in *z* direction).
d has always a negative value!
$z_p = d$
From projection geometry:
$y / z = y_p / d$

$$y_p = \frac{y}{z/d} \qquad x_p = \frac{x}{z/d}$$

Non-uniform foreshortening

Homogeneous Coordinates Revisited (see Ch. 3)

- Usage of a representation of coordinate-positions with an extra dimension
  - Extra value is a *scaling factor*
- 3D position (x, y, z) is represented by (x_h, y_h, z_h, h) such that

$$x = \frac{x_h}{h}, \quad y = \frac{y_h}{h}, \quad z = \frac{z_h}{h}$$

- Simple choice for scaling factor h is the value 1
  - In special cases other values can be used
- Given a scaled position (x_h, y_h, z_h, h),
  we get the original position back by dividing all components by h,
  i.e. as

  (x_h / h, y_h / h, z_h / h, 1)
  "divide all components by the fourth component"

Matrix Representation of Simple Perspective

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix}$$

What happens if we apply this matrix to an arbitrary point *p* (in homogenous coordinates)?

$$p = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

$$Mp = \begin{pmatrix} x \\ y \\ z \\ z/d \end{pmatrix}$$

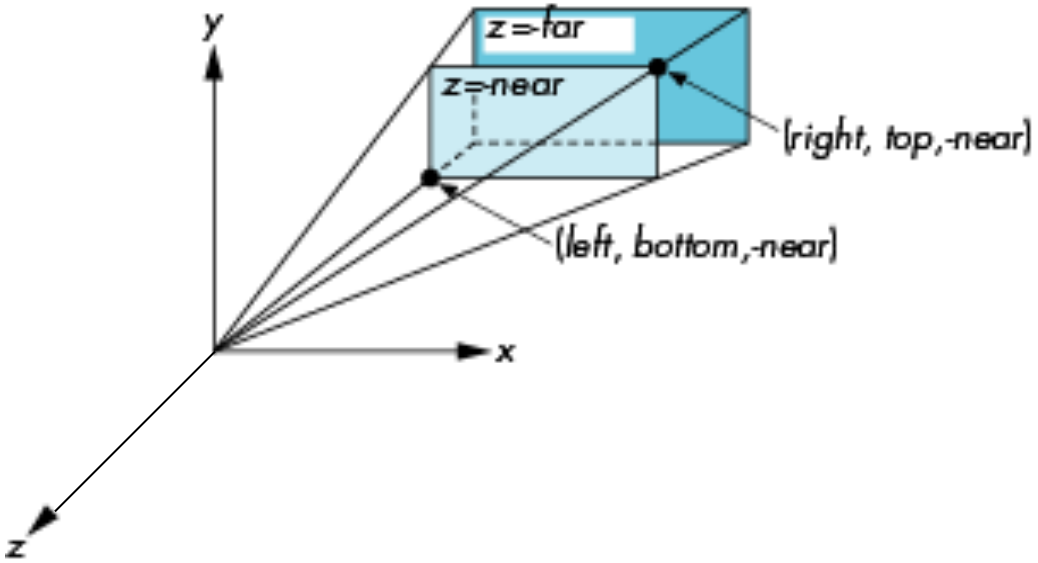
Scaling up to the original 3D space from this representation: Divide the all components by the fourth.

So the point *p* is mapped to:

$$\begin{pmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ \frac{z}{z/d} \\ \frac{z/d}{z/d} \end{pmatrix} = \begin{pmatrix} x_p \\ y_p \\ z_p \\ 1 \end{pmatrix}$$

Perspective Transformations

- It is possible to map also perspective transformations to matrix multiplications of coordinates
- We need a final *perspective division* to recover from scaling
- Additionally, a viewing volume *(frustum, Pyramidenstumpf)* is defined, and normalization is applied, as in orthographic projection



OpenGL functions for perspective transformation:

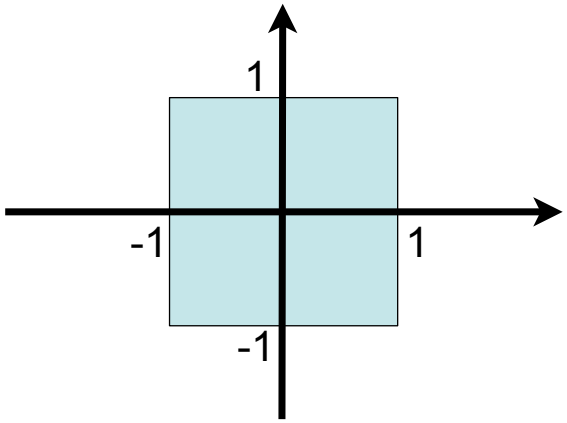
```
glFrustum
 (left,right,bottom,top,near,far)

glPerspective
 (fovy, aspect, near, far)
```

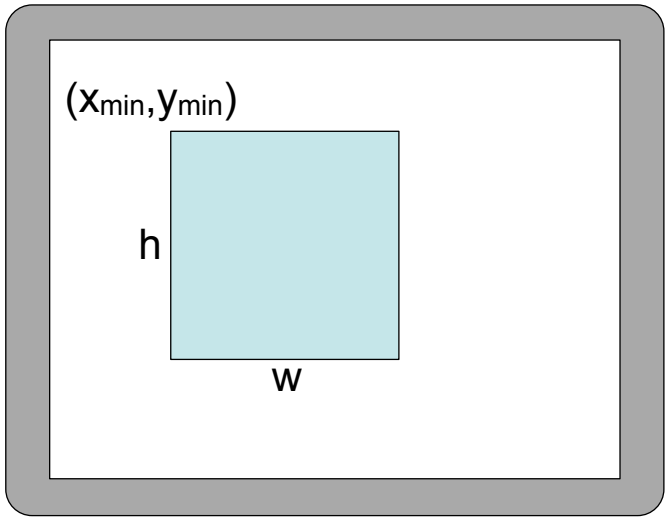
fovy: Angle of view between top and bottom
aspect: height/width ratio

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 5 - 3D Camera & Optimizations, 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

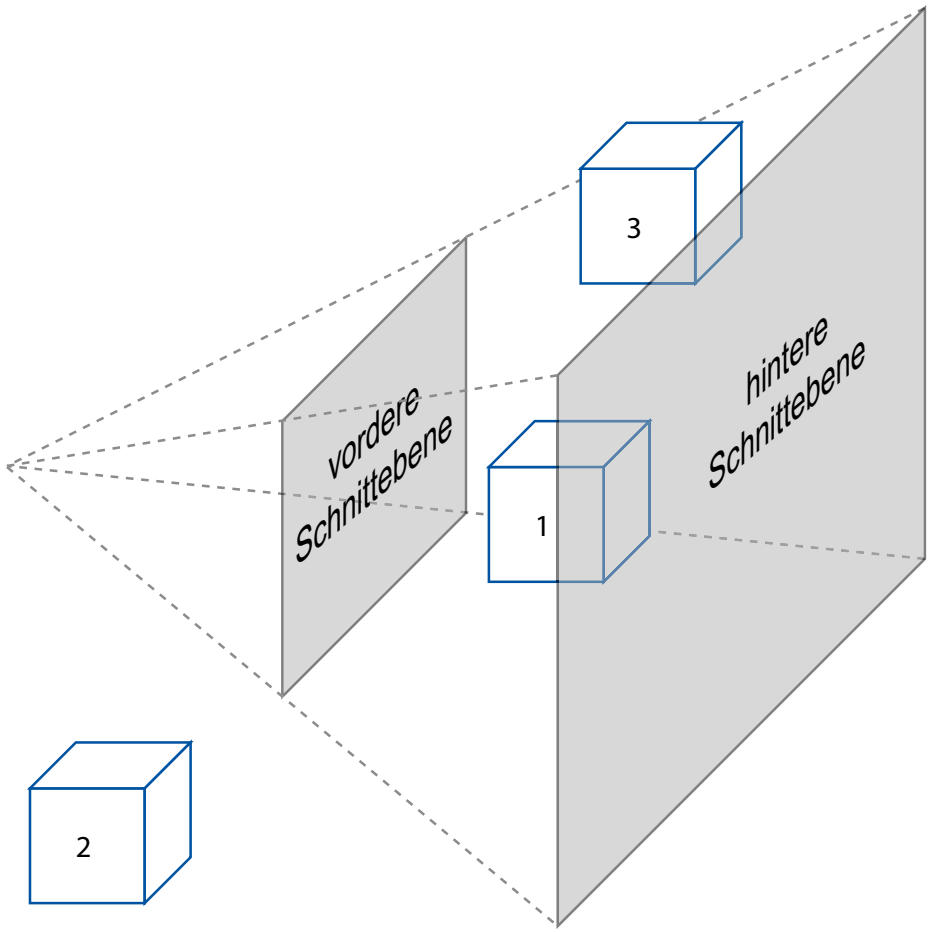
View Frustum Culling

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

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

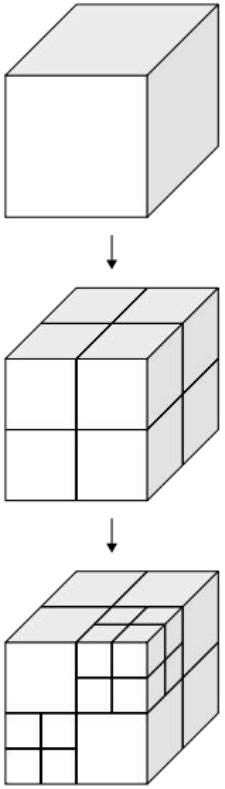
- X- and Y-Axis: inside the viewing cone
- $-w_{view} < x_{view} < w_{view}$
- $-w_{view} < y_{view} < w_{view}$

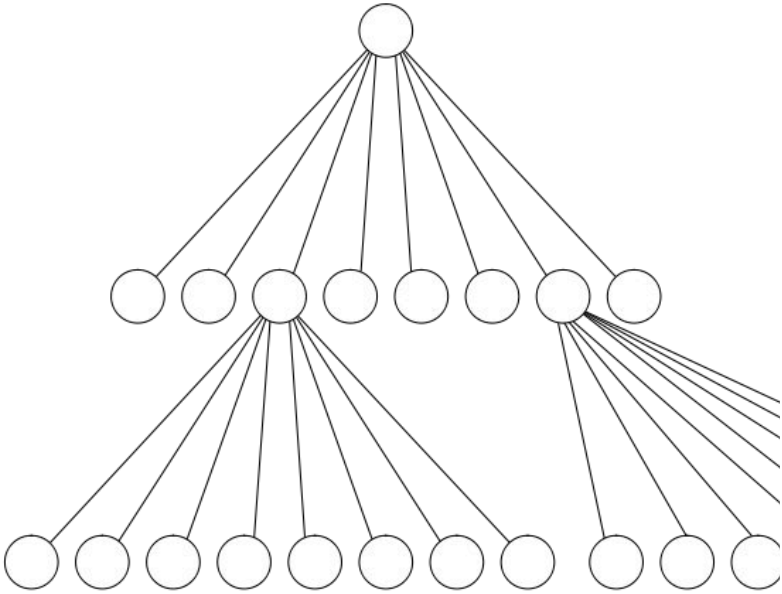
- Two simple comparisons for each axis!



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
  - Repeat recursively until cube contains less than k objects
- Instead of culling objects, cull cubes
- Needs O(log n) tests

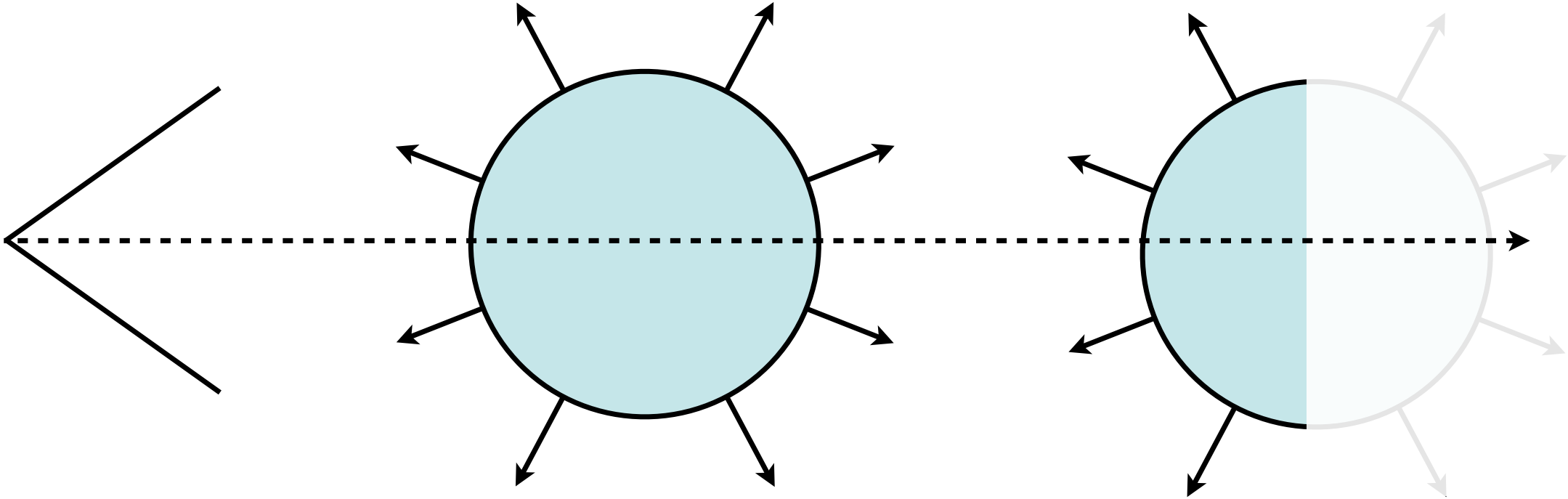




http://en.wikipedia.org/wiki/File:Octree2.svg

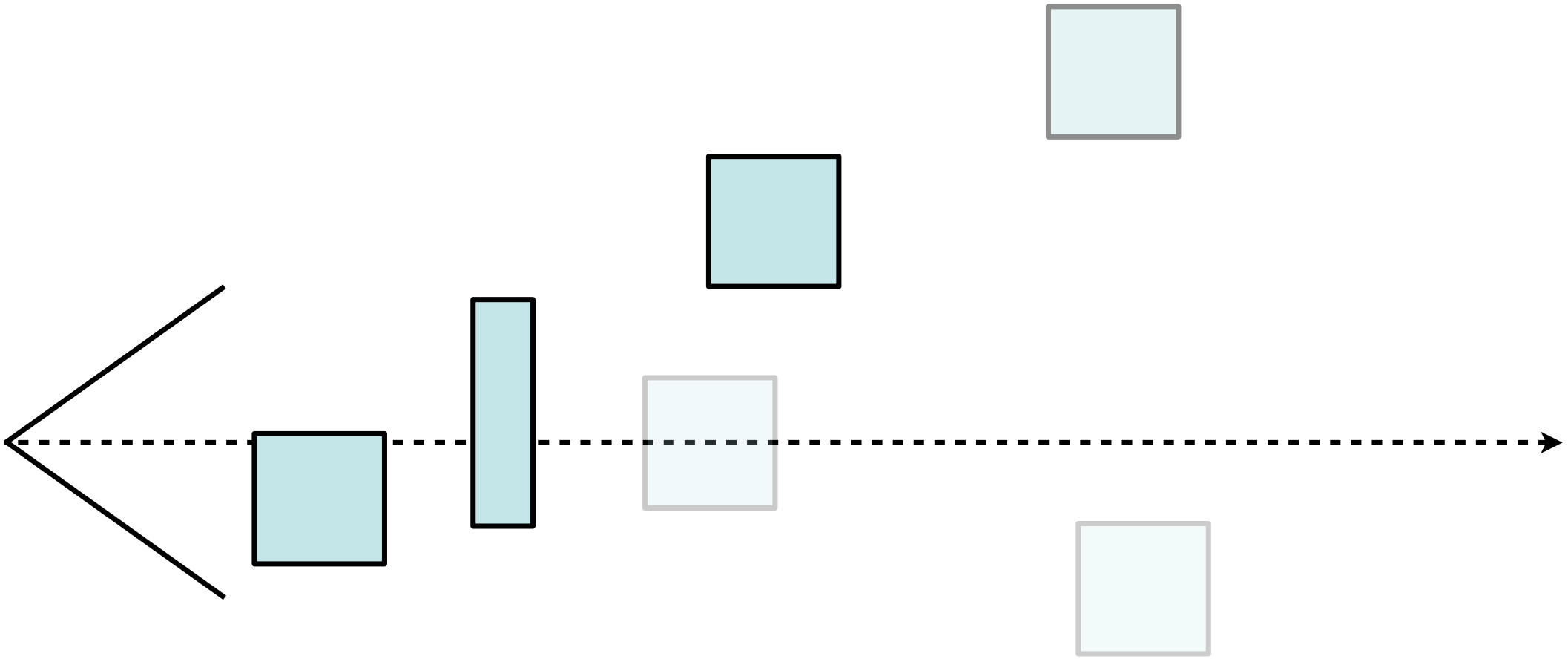
Back-face Culling

- 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
  - See earlier chapter, often normals are stored in face mesh structue



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 & Optimizations, Rastering

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

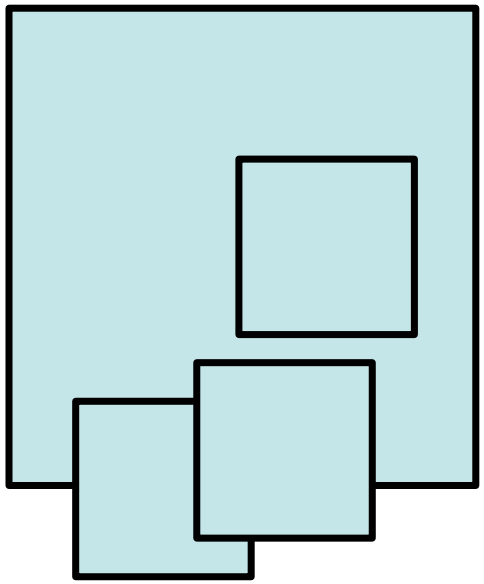
Occlusion: The problem space in general

- Need to determine which objects occlude which others
- 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!



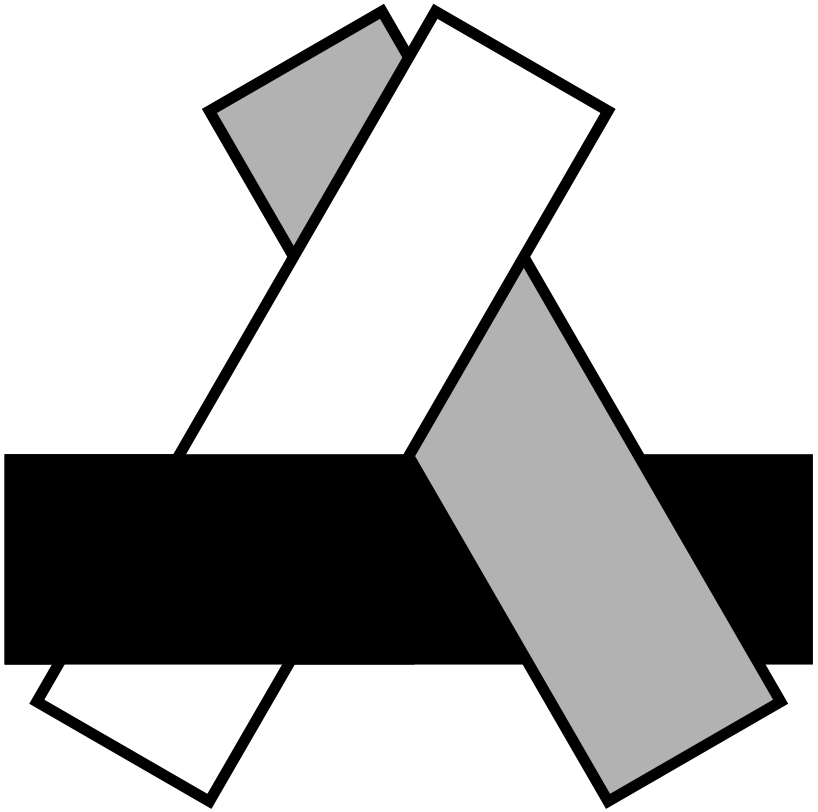
Occlusion: depth-sort

- 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!

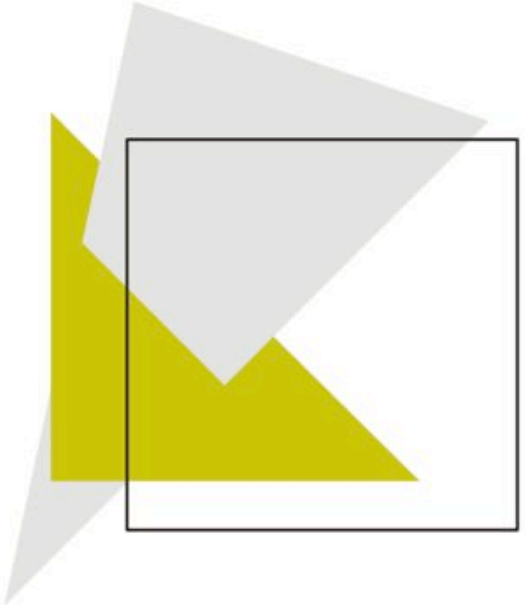


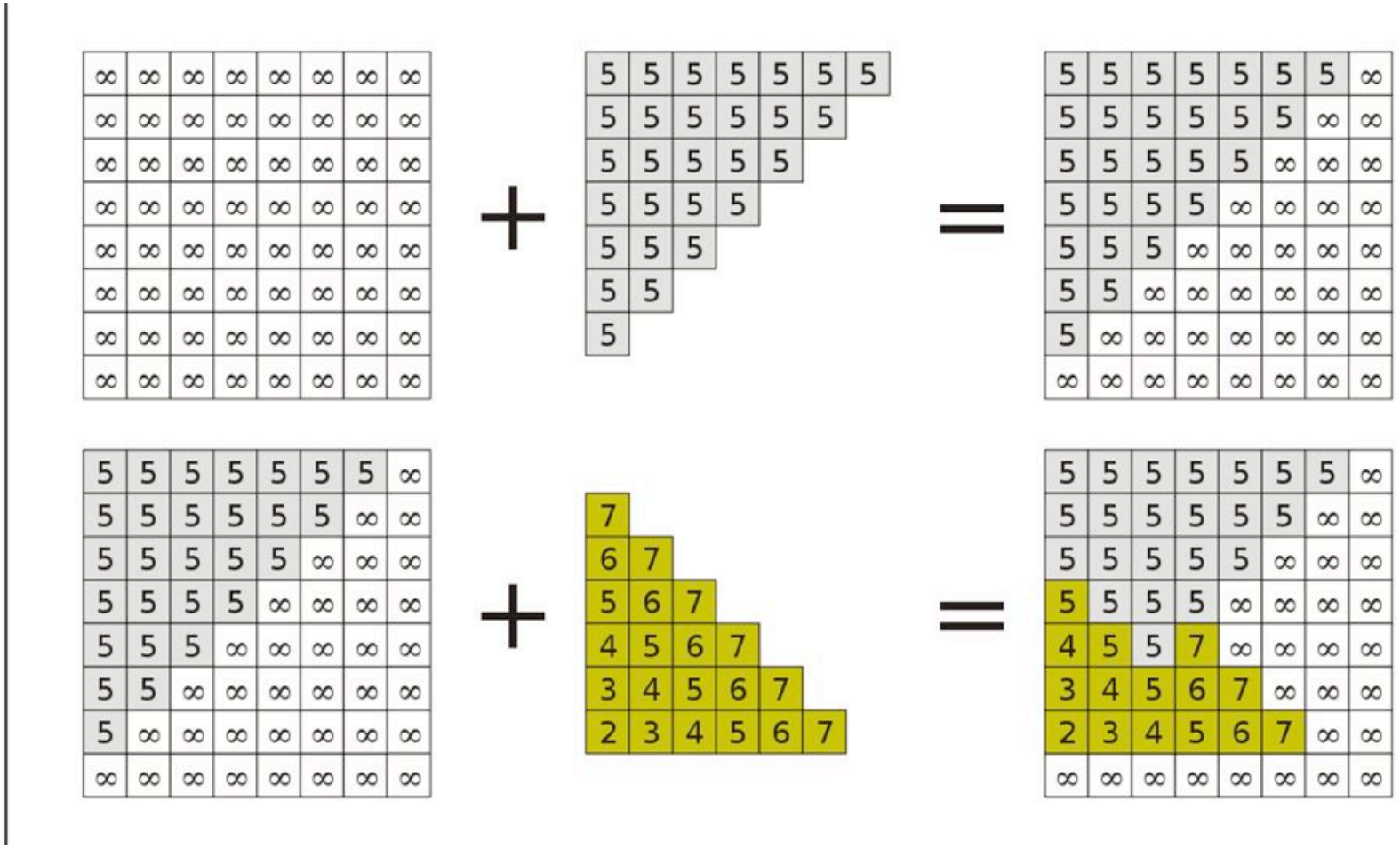
Occlusion: Z-Buffer

- 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

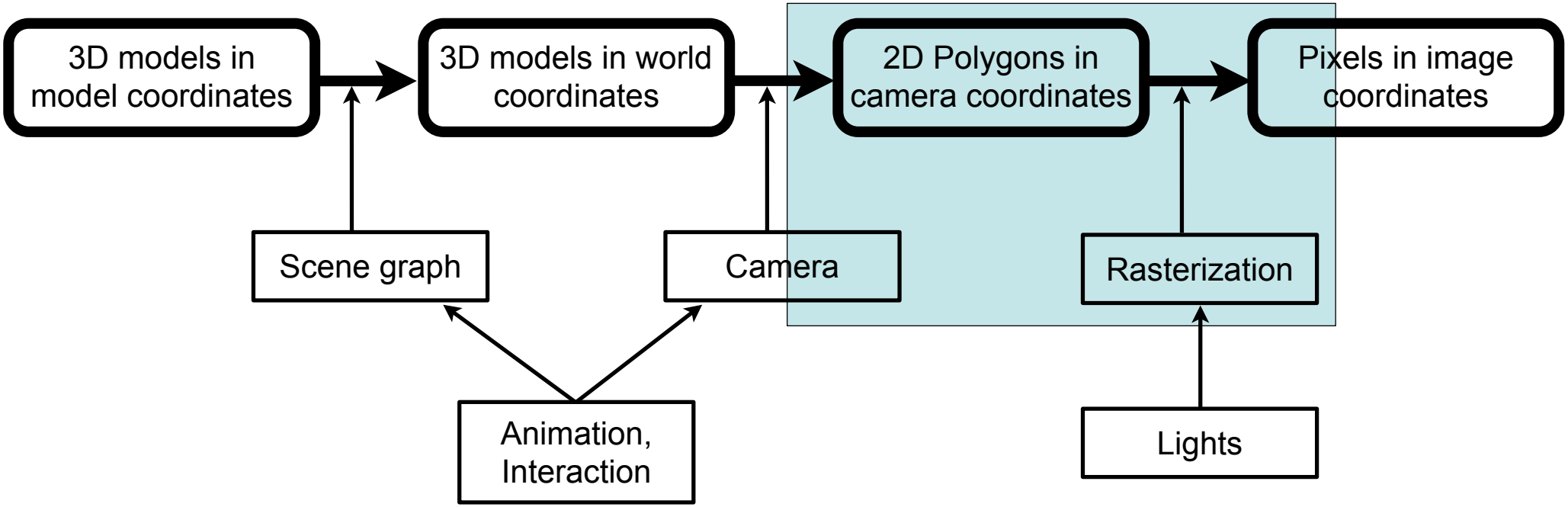
- Z-Buffer normally built into graphics hardware
- Limited precision (e.g., 16 bit)
  - potential problems with large models
  - set clipping planes wisely!
  - never have 2 polygons in the exact same place
  - otherwise typical errors (striped objects)

- 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 5 - 3D Camera & Optimizations, 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!

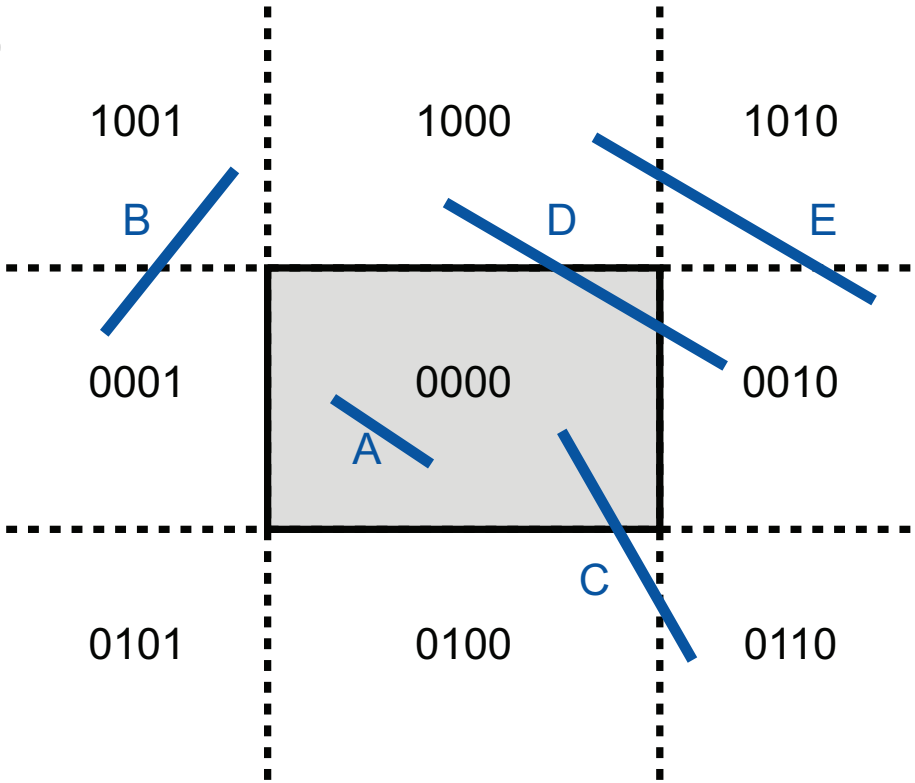
- Drawing lines: How do we convert all those polygon edges into lines of pixels?

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

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

Clipping (Cohen & Sutherland)

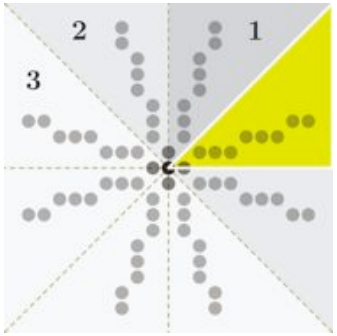
- Clip lines against a rectangle
- For end points P and Q of a line
  - determine a 4 bit code each
  - 10xx = point is above rectangle
  - 01xx = point is below rectangle
  - xx01 = point is left of rectangle
  - xx10 = point is right of rectangle
  - easy to do with simple comparisons



- 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: 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 exchange x and y

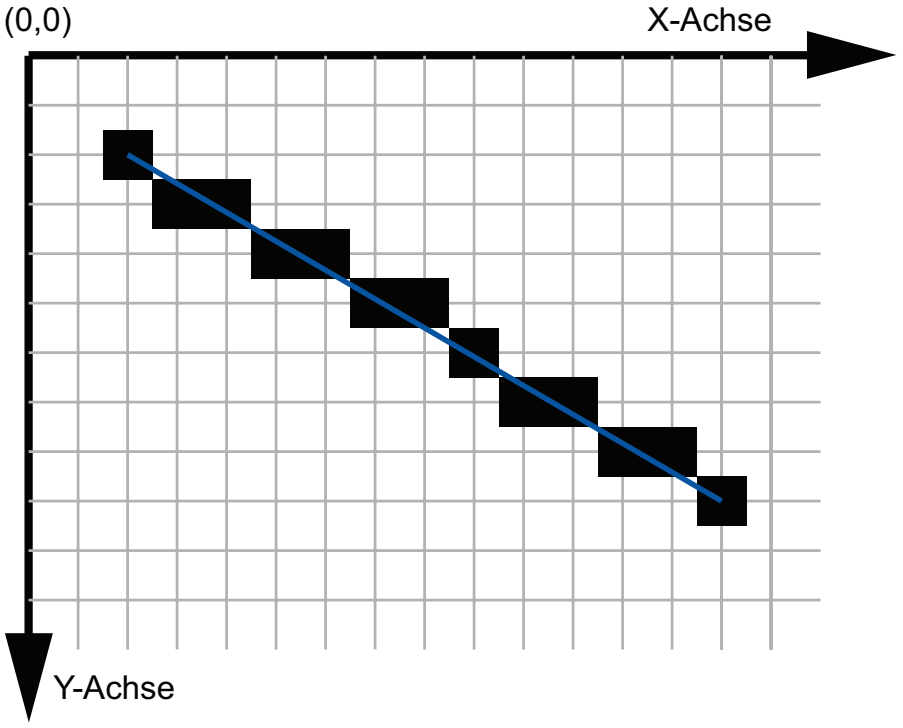


For x from 0 to dx do:

  setpixel (x_1 + x, y_1 + m * x)

od;

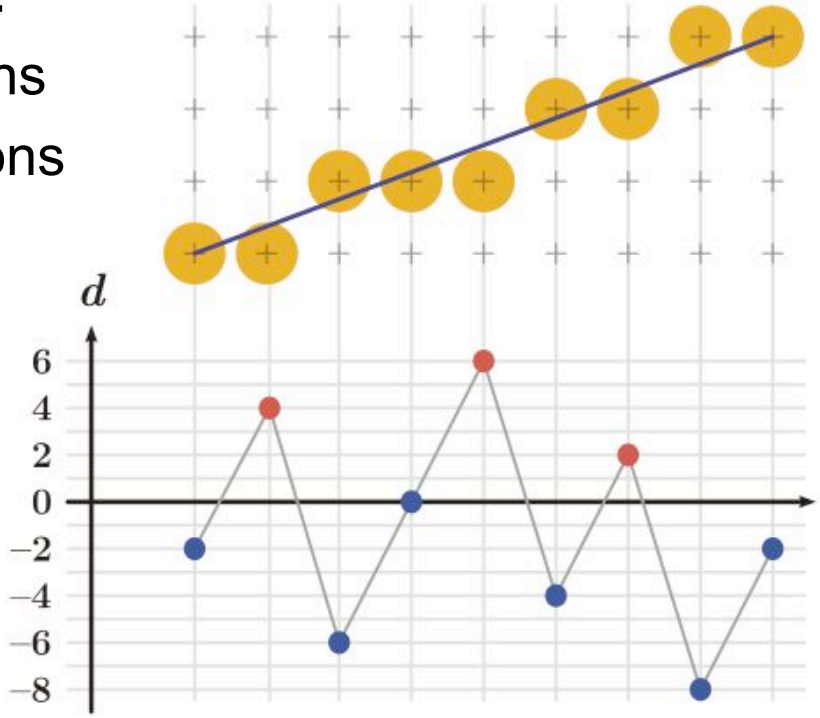
- In each step:
  - 1 float multiplication
  - 1 round to integer



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



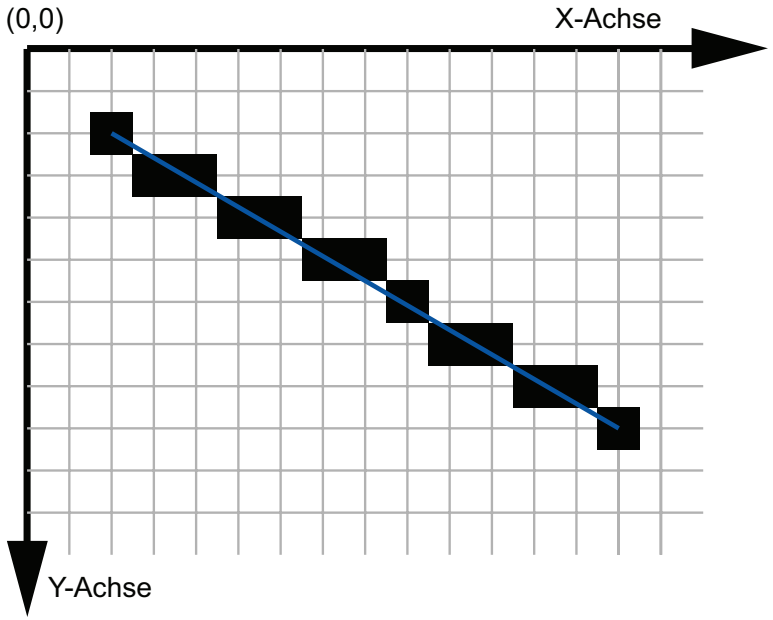
http://de.wikipedia.org/w/index.php?title=Datei:Bresenham_decision_variable.svg

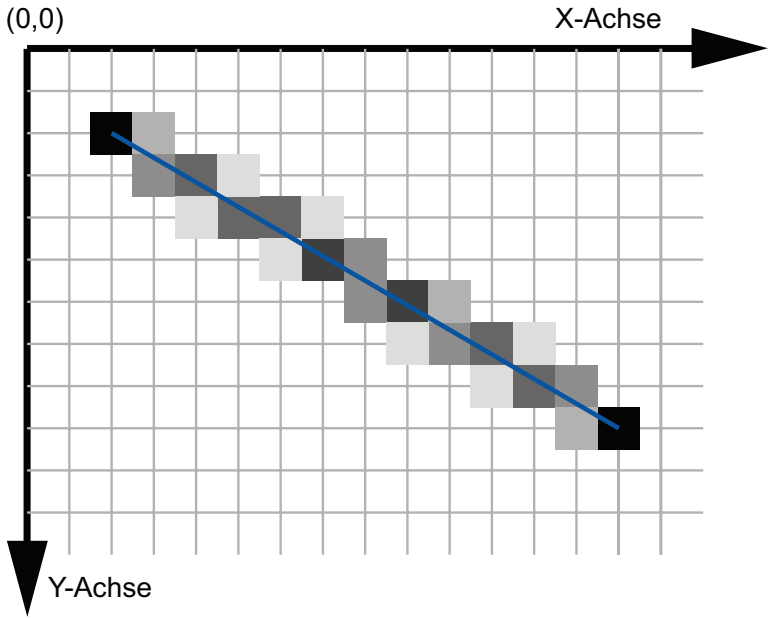
```
dx := x2-x1; dy := y2-y1
d := 2*dy – dx; DO := 2*dy;
dNO := 2*(dy - dx)
x := x1; y := y1
setpixel (x,y)
fehler := d
WHILE x < x2
      x := x + 1
      IF fehler <= 0 THEN
            fehler := fehler + DO
      ELSE
            y := y + 1
            fehler = fehler + dNO
      END IF
      setpixel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

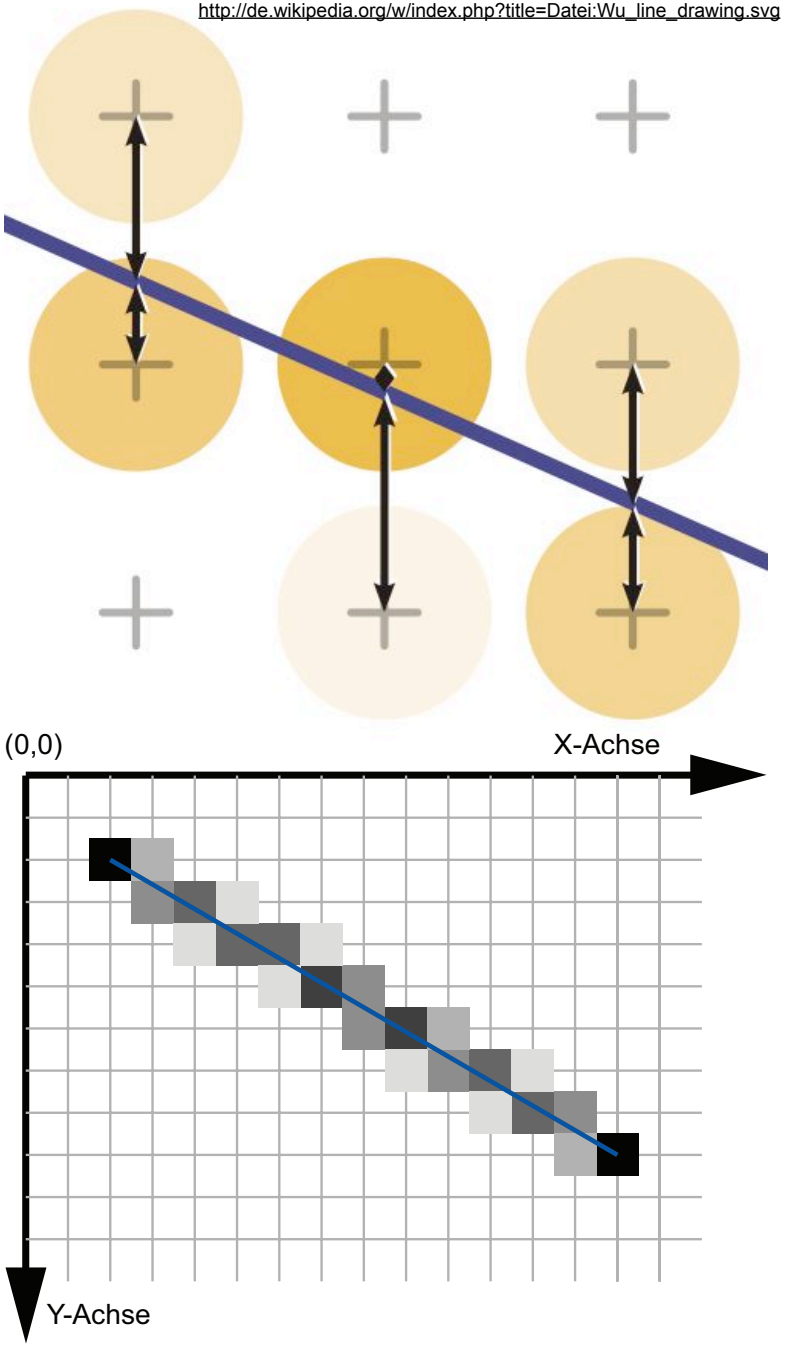




Wu‘s Antialiasing Approach

- 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

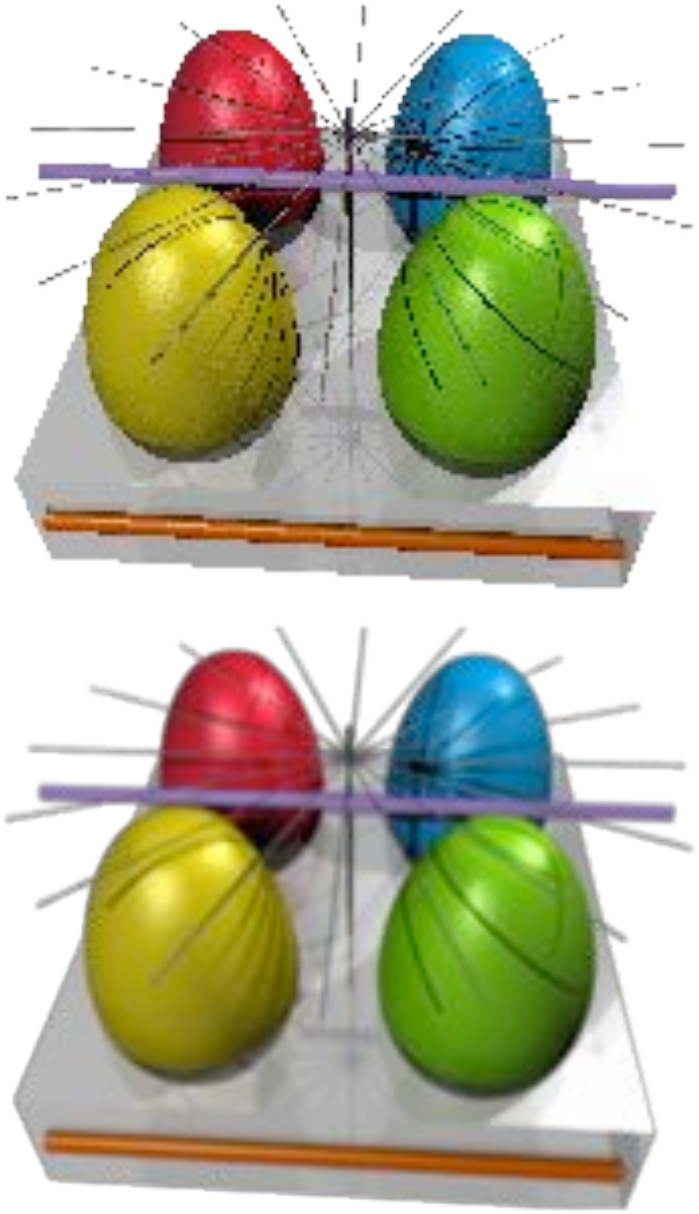
http://de.wikipedia.org/w/index.php?title=Datei:Wu_line_drawing.svg



Antialiasing in General

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



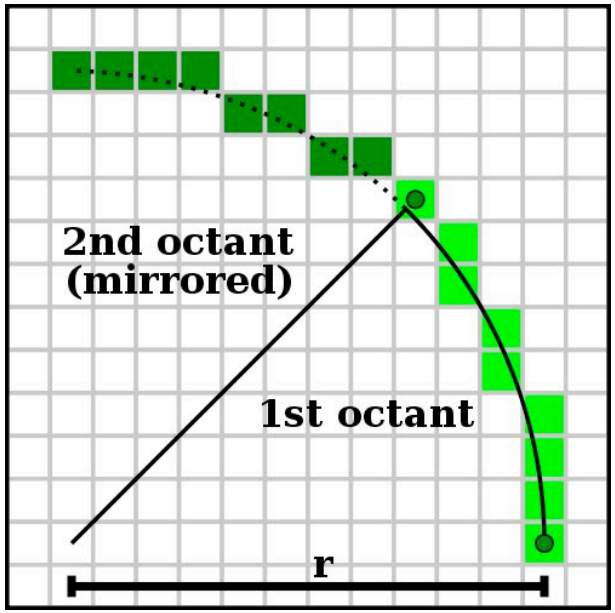
http://de.wikipedia.org/w/index.php?title=Datei:EasterEgg_anti-aliasing.png

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?

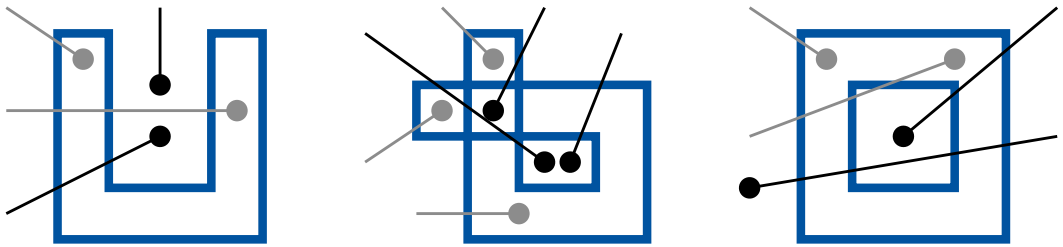
- All these algorithms implemented in hardware
- Other algorithms exist for curved lines
  - mostly relevant for 2D graphics



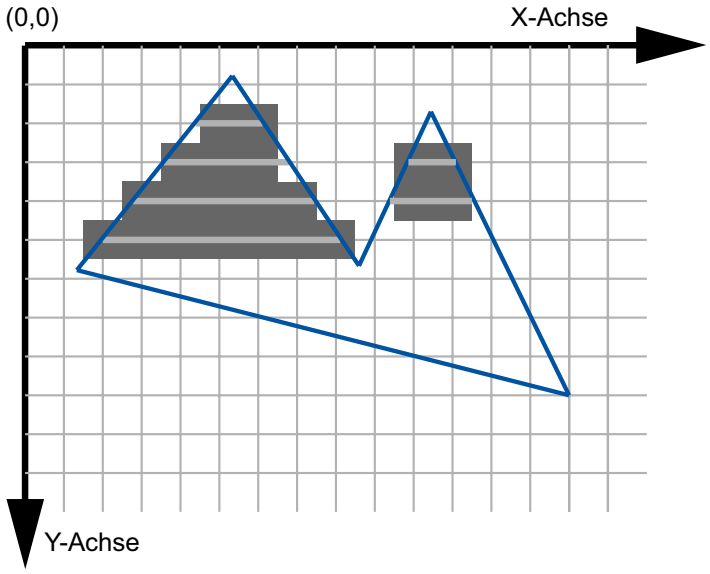
http://en.wikipedia.org/wiki/File:Bresenham_circle.svg

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