Computer Graphics

-Raster Algorithms and Depth Handling-

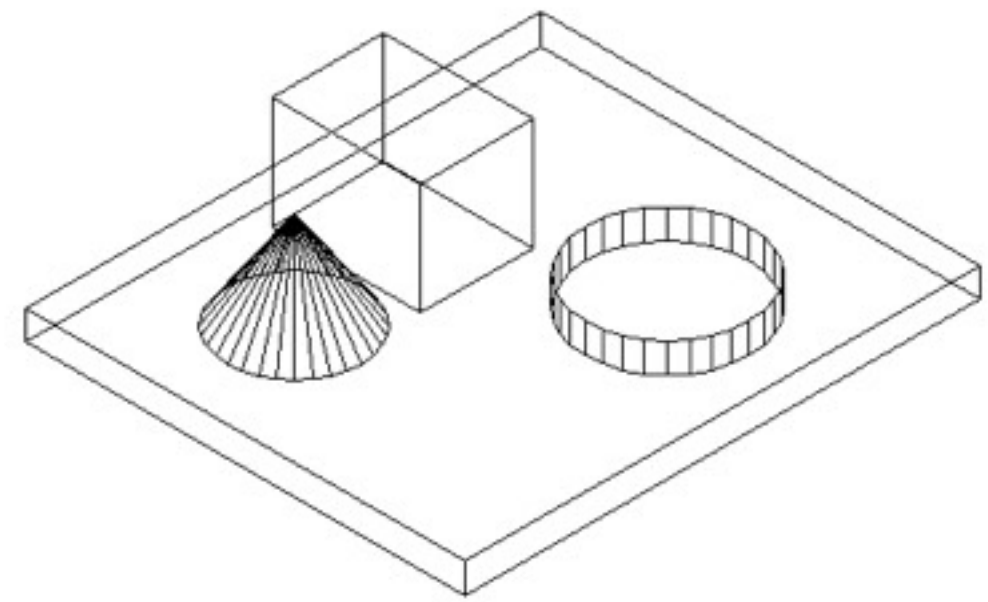
Oliver Bimber

Course Schedule

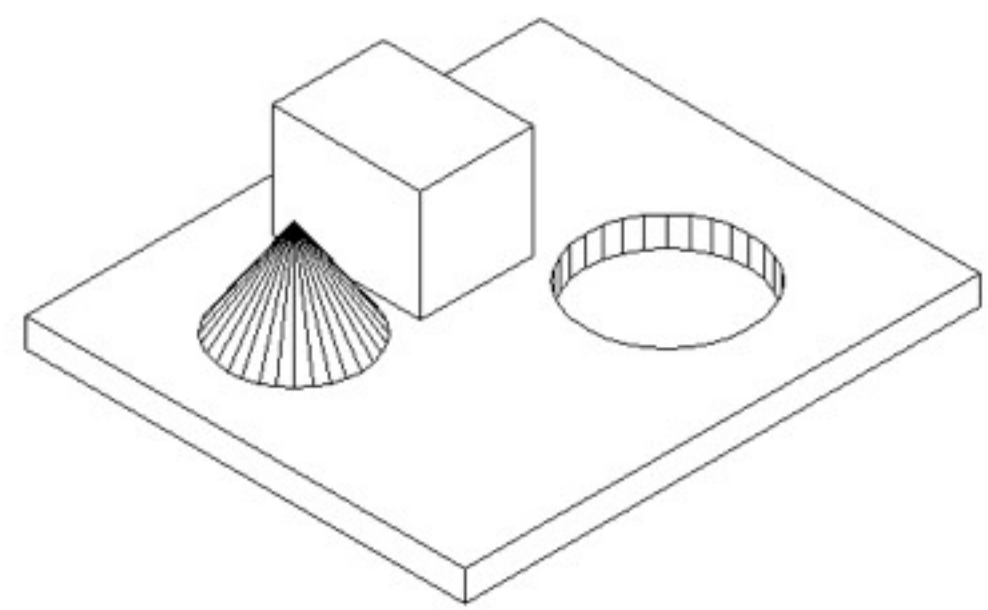
Type	Date	Time	Room	Topic	Comment
C1	01.03.2016	13:45-15:15	HS 18	Introduction and Course Overview	Conference
C2	15.03.2016	13:45-15:15	HS 18	Transformations and Projections	Easter Break
C3	05.04.2016	13:45-15:15	HS 18	Raster Algorithms and Depth Handling	
C4	12.04.2016	13:45-15:15	HS 18	Local Shading and Illumination	
C5	19.04.2016	13:45-15:15	HS 18	Texture Mapping Basics	
C6	26.4.2016	13:45-15:15	HS 18	Advanced Texture Mapping & Graphics Pipelines	
C7	03.05.2016	13:45-15:15	HS 18	Intermediate Exam	
C8	09.05.2016	17:15-18:45	HS 18	Global Illumination I: Raytracing	
C9	10.05.2016	13:45-15:15	HS 18	Global Illumination II: Radiosity	Conference / Holiday
C10	31.05.2016	13:45-15:15	HS 18	Volume Rendering	
C11	07.06.2016	13:45-15:15	HS 18	Scientific Data Visualization	
C12	14.06.2016	13:45-15:15	HS 18	Curves and Surfaces	
C13	21.06.2016	13:45-15:15	HS 18	Basics of Animation	
C14	28.06.2016	13:45-15:15	HS 18	Final Exam	
C15	04.10.2016	13:45-15:15	TBA	Retry Exam	

Hidden Surface Removal

- So far, we know how to transform vertices, surfaces (e.g., triangles) and normals, and how to project them
- But how do we ensure that surfaces in the front occlude surfaces in the back?
- If triangles are rendered without considering their depth order, we refer to the result as wire-frame model
- Otherwise we refer to it as surface model
- Depth relative to camera position can be used to determine the depth order



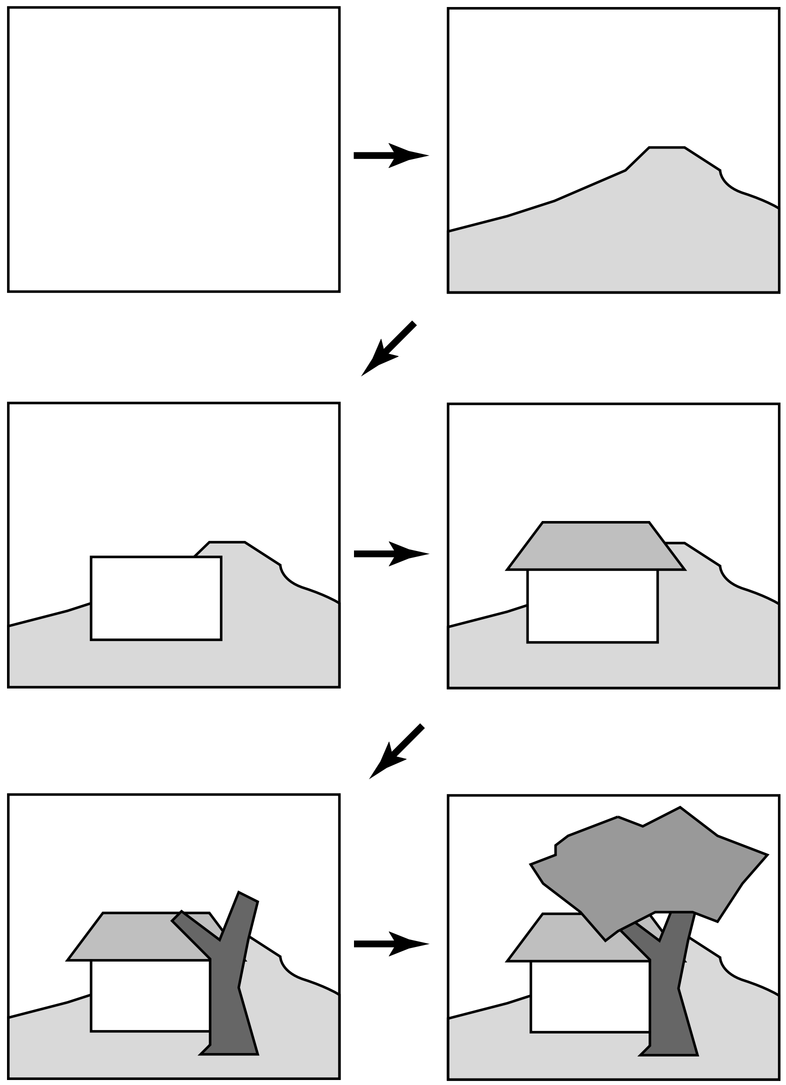
without depth-handling

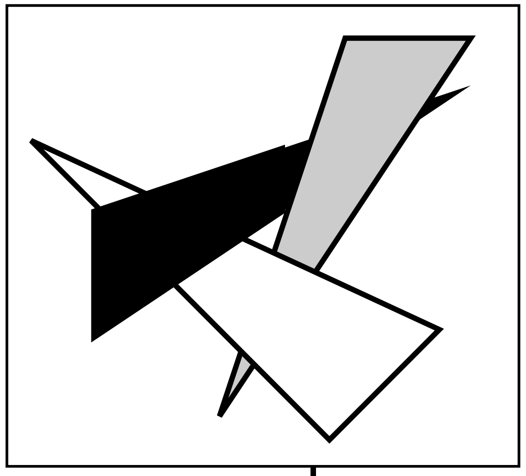


with depth-handling

Painter‘s Algorithm

- Let‘s assume that this depth order is known (can be pre-sorted)
- We can simply render the surfaces in the same order (from back to front)
- The problem is, that the relative order of surfaces is not always well defined (e.g. cycles)
- Cycles occur if a global back-to-front ordering of surfaces is not possible for a particular position

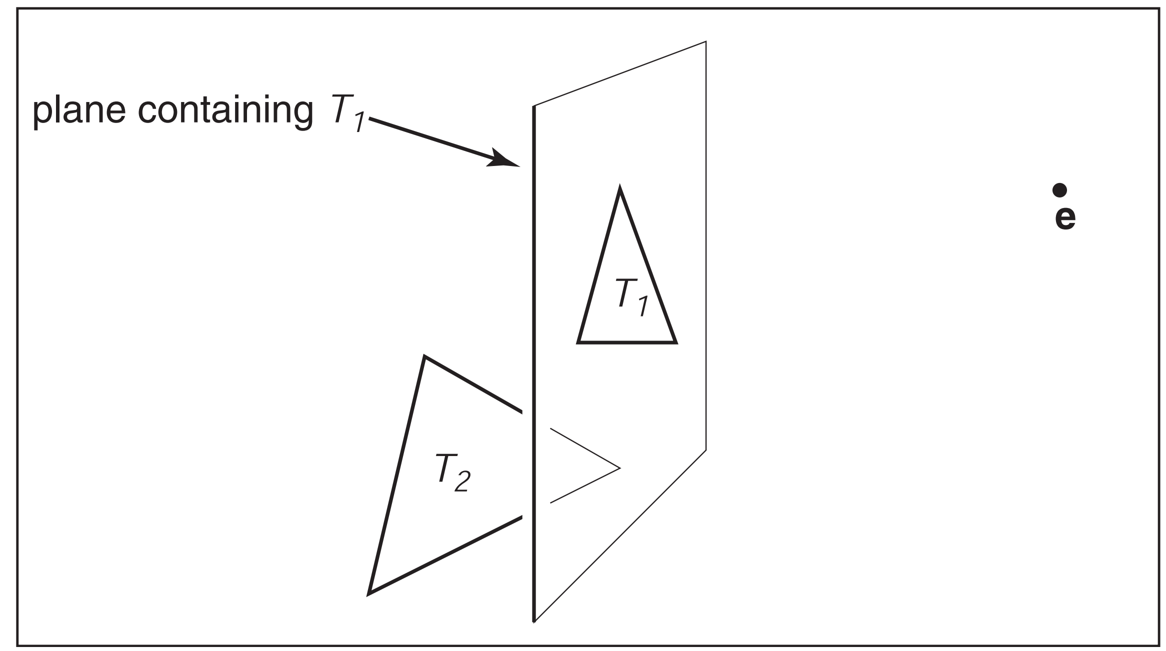




cycle

BSP Trees

- Binary Space Partitioning (BSP) trees can help (you might know them from your algorithms class)
- They will work only if no surface crosses the plane of another surface
- This condition, however, can be relaxed through a pre-processing step
- If a triangle passes through a plane of another triangle, we simply cut this triangle into multiple triangles to satisfy our condition

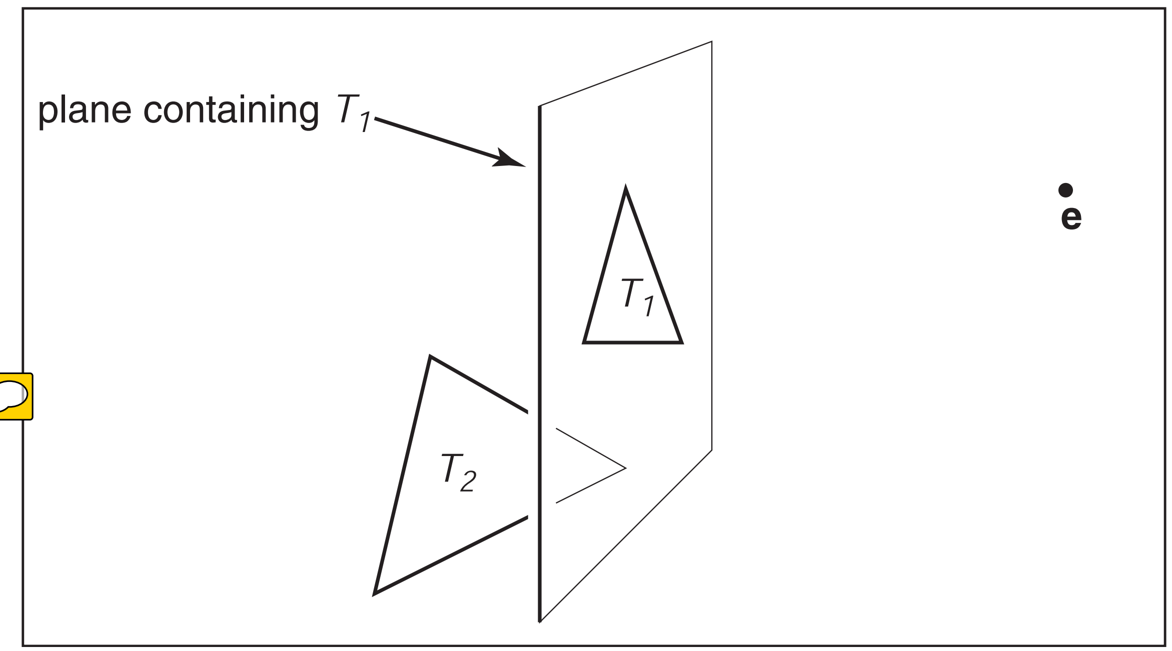


BSP Trees

- The three vertices of a triangle are located on the same plane that can be represented by its implicit plane equation
- Let‘s consider only two triangles (T1 and T2) with their plane equations (f1 and f2), and the known camera point (e)
- Let‘s also assume that all points of T2 are on the f1(p)<0 side of T1
- Then we can find the correct order for any eye point (e):
  - if f1(e)<0 then T1,T2
  - else T2,T1

$$f(p) = ((b-a) \times (c-a)) \cdot (p-a) = 0$$

(a,b,c) are points on plane,
cosine of plane normal and vector on plane must be 0



$f(p) = 0$ point is on plane

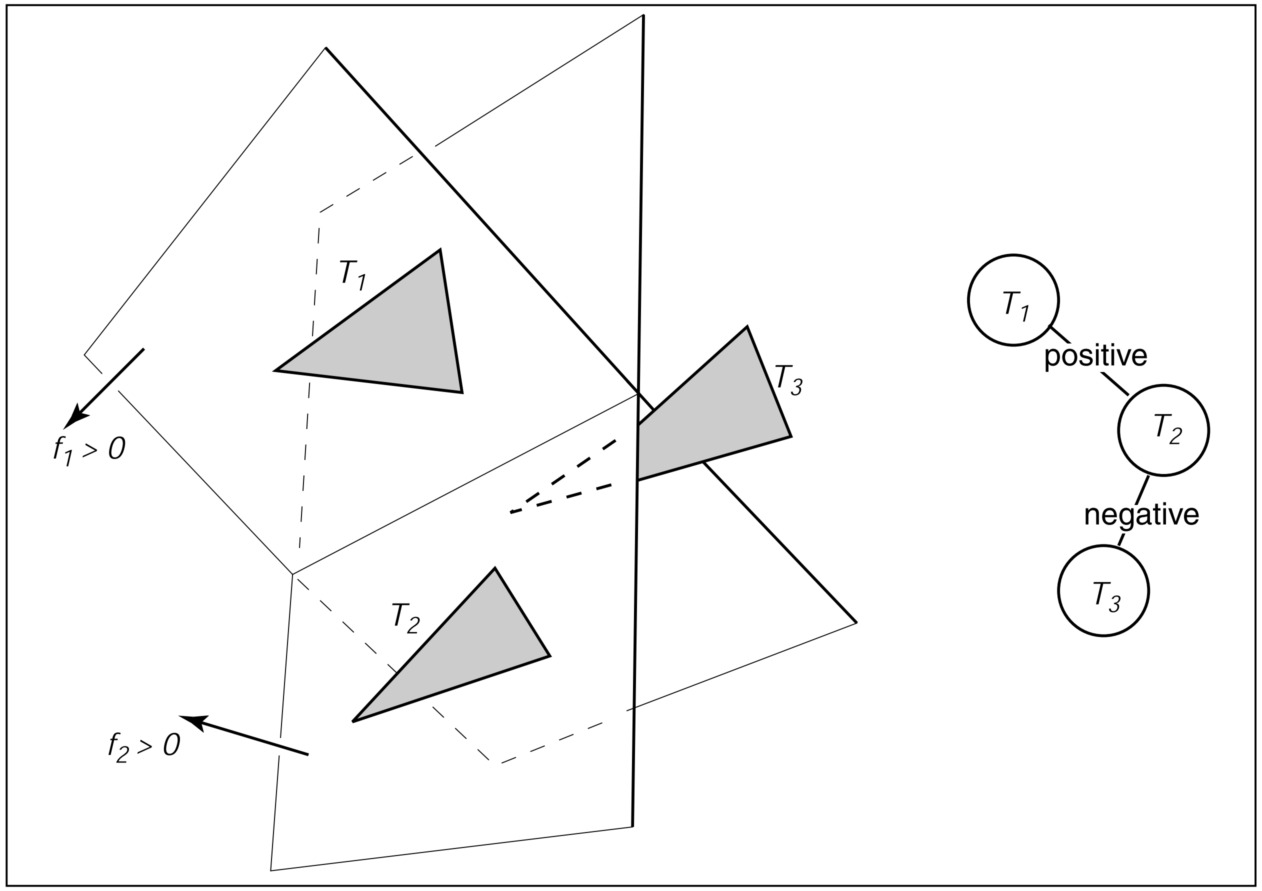
$f(p^+) > 0$ point is on one side of plane

$f(p^-) < 0$ point is on other side of plane

BSP Trees

- This can be generalized using a binary tree structure and recursion
- We pick one reference triangle (T1) as root and start building our tree structure by dividing all other triangles in two groups: the ones on one side and the ones on the other side
- Unfortunately, which side is which depends on the order of the triangle vertices when computing the plane equation (there are two possibilities)
- We need to compare against a common reference point
- Using (e) as reference makes sense

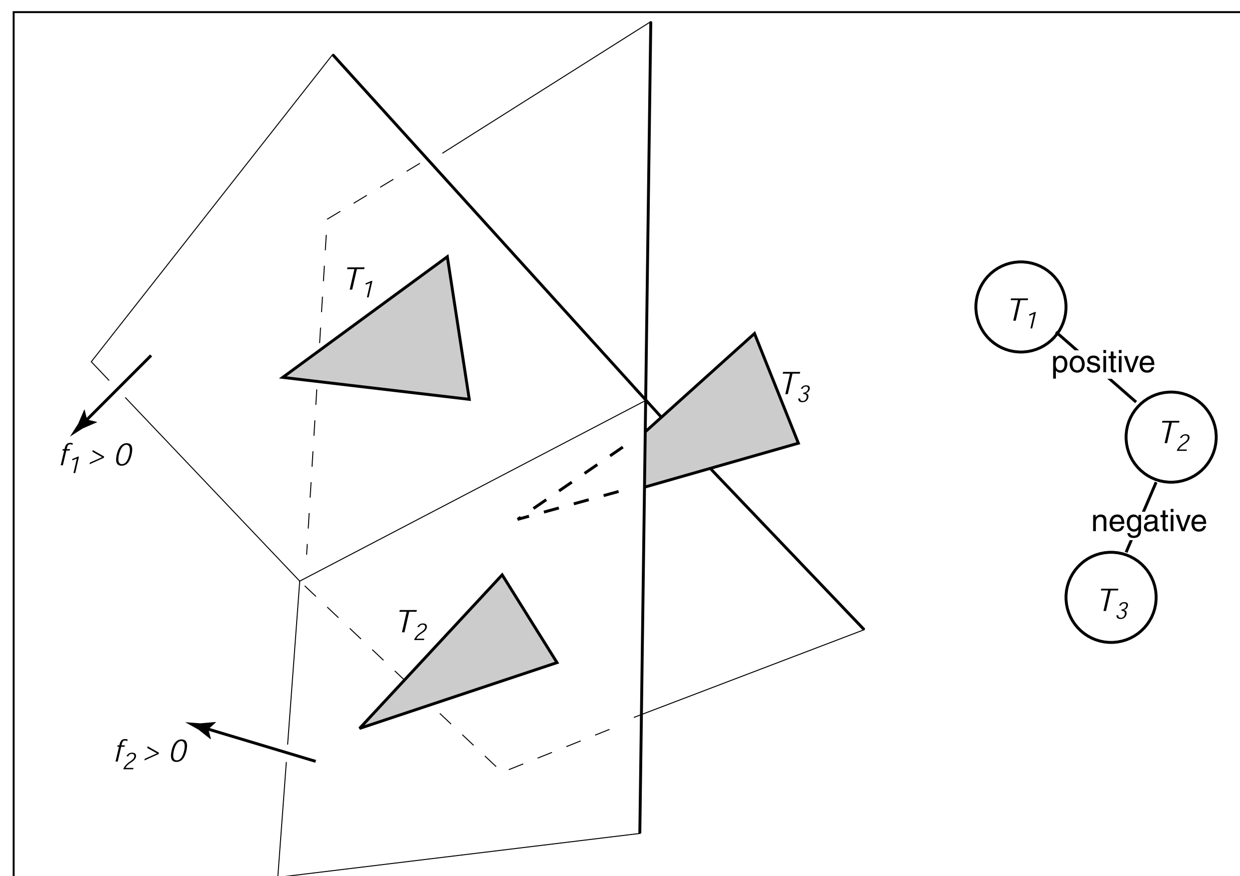
```
function render(bsptree tree, point e)
if (tree.empty) then
    return
if (f_tree.root(e)<0) then
    render(tree.plus,e)
    draw(tree.triangle)
    render(tree.minus, e)
else
    render(tree.minus,e)
    draw(tree.triangle)
    render(tree.plus,e)
```



BSP Trees

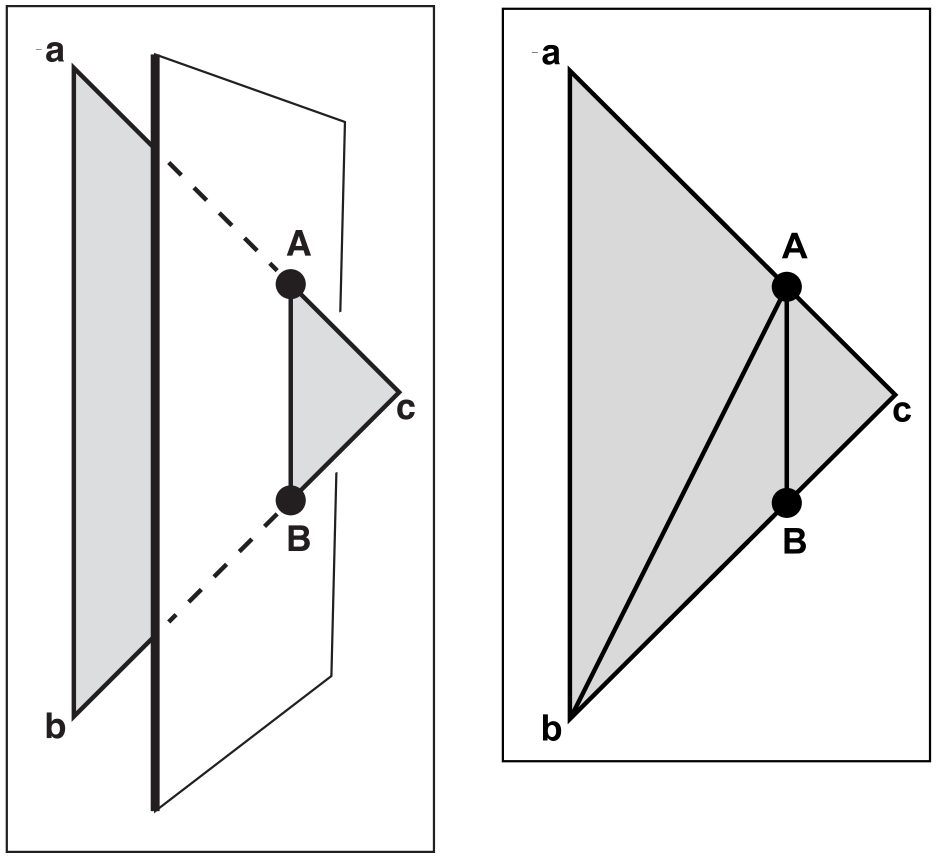
- With respect to each reference triangle (at root):
  - We render first all triangles that are on opposite site of e
  - Then we render the reference triangle
  - Then we render all triangles that are on same side as e
- This is repeated recursively
- The advantage is, that this technique works for any viewpoint e
- Therefore, the BSP tree can be pre-computed for static scenes, and traversed for moving cameras during runtime
- Good for static sceneries in games

```
function render(bsptree tree, point e)
if (tree.empty) then
    return
if (f_tree.root(e)<0) then
    render(tree.plus,e)
    draw(tree.triangle)
    render(tree.minus, e)
else
    render(tree.minus,e)
    draw(tree.triangle)
    render(tree.plus,e)
```



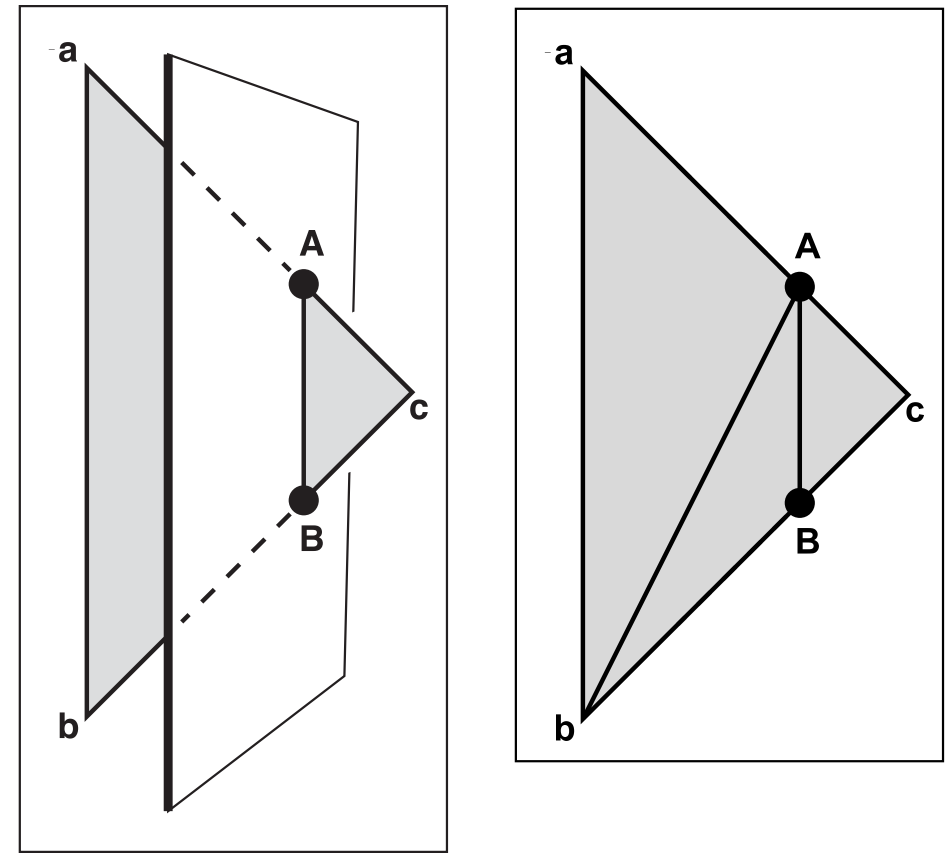
BSP Trees

- But we assumed that triangles do not intersect planes of other triangles, so far
- This is a fairly uncommon situation for normal 3D scenes
- If one triangle intersects a dividing plane of another triangle, then one vertex will be on one side, while the others will be on the other side
- If this is detected during building the BSP tree, we have to find the two intersection points (A,B) and cut the triangle into three
- All three triangles are now on clear sides



BSP Trees

- How do we compute the intersection points A and B?
- Parameterize the lines ac and bc, and compute the line intersection with the implicit plane equation



$$l(t) = a + t(c - a)$$

parametric line equation

$$f(p) = ap_x + bp_y + cp_z + d = np + d = 0$$

alternative implicit pane equation (n is the plane normal and d the distance to the origin)

$$n \cdot (a + t(c - a)) + d = 0$$

solving f(l(t)) for t...

$$t = -\frac{n \cdot a + d}{n \cdot (c - a)}$$

...leads to the intersection at scale t...

$$A = a + t_A(c - a)$$
$$B = b + t_B(c - b)$$

...and to the actual points A and B

Recap: Perspective Projection

- The value (f) is the distance to the far clipping plane of our viewing frustum, and (n) the distance to its near clipping plane
- Here, the homogeneous coordinate plays an important role
- If we allow the homogeneous coordinate to take up any value (not just 0 or 1), then it can be used to encode how much the other three coordinates must be scaled after a perspective transform
- Dividing these three coordinates by the homogeneous coordinate is called homogenization or perspective division
- As for the orthographic case, the resulting value in the z component is not used yet - it will be used for hidden surface removal since it preserves depth order

Perspective transform matrix M_p

$$\boxed{\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ (n+f)z - fn \\ \boxed{z} \end{bmatrix}$$

homogeneous coordinate

$$\begin{bmatrix} nx \\ ny \\ (n+f)z - fn \\ z \end{bmatrix} \boxed{/z} = \begin{bmatrix} \frac{n}{z}x \\ \frac{n}{z}y \\ n+f-\frac{fn}{z} \\ 1 \end{bmatrix}$$

homogenization or perspective division

$$M_p^{-1} = \begin{bmatrix} \frac{1}{n} & 0 & 0 & 0 \\ 0 & \frac{1}{n} & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -\frac{1}{fn} & \frac{n+f}{fn} \end{bmatrix}$$

Recap: Perspective Projection

- The value (f) is the distance to the far clipping plane of our viewing frustum, and (n) the distance to its near clipping plane
- Here, the homogeneous coordinate plays an important role
- If we allow the homogeneous coordinate to take up any value (not just 0 or 1), then it can be used to encode how much the other three coordinates must be scaled after a perspective transform
- Dividing these three coordinates by the homogeneous coordinate is called homogenization or perspective division
- As for the orthographic case, the resulting value in the z component is not used yet - it will be used for hidden surface removal since it preserves depth order

Perspective transform matrix M_p

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ (n+f)z - fn \\ \boxed{z} \end{bmatrix}$$

homogeneous coordinate

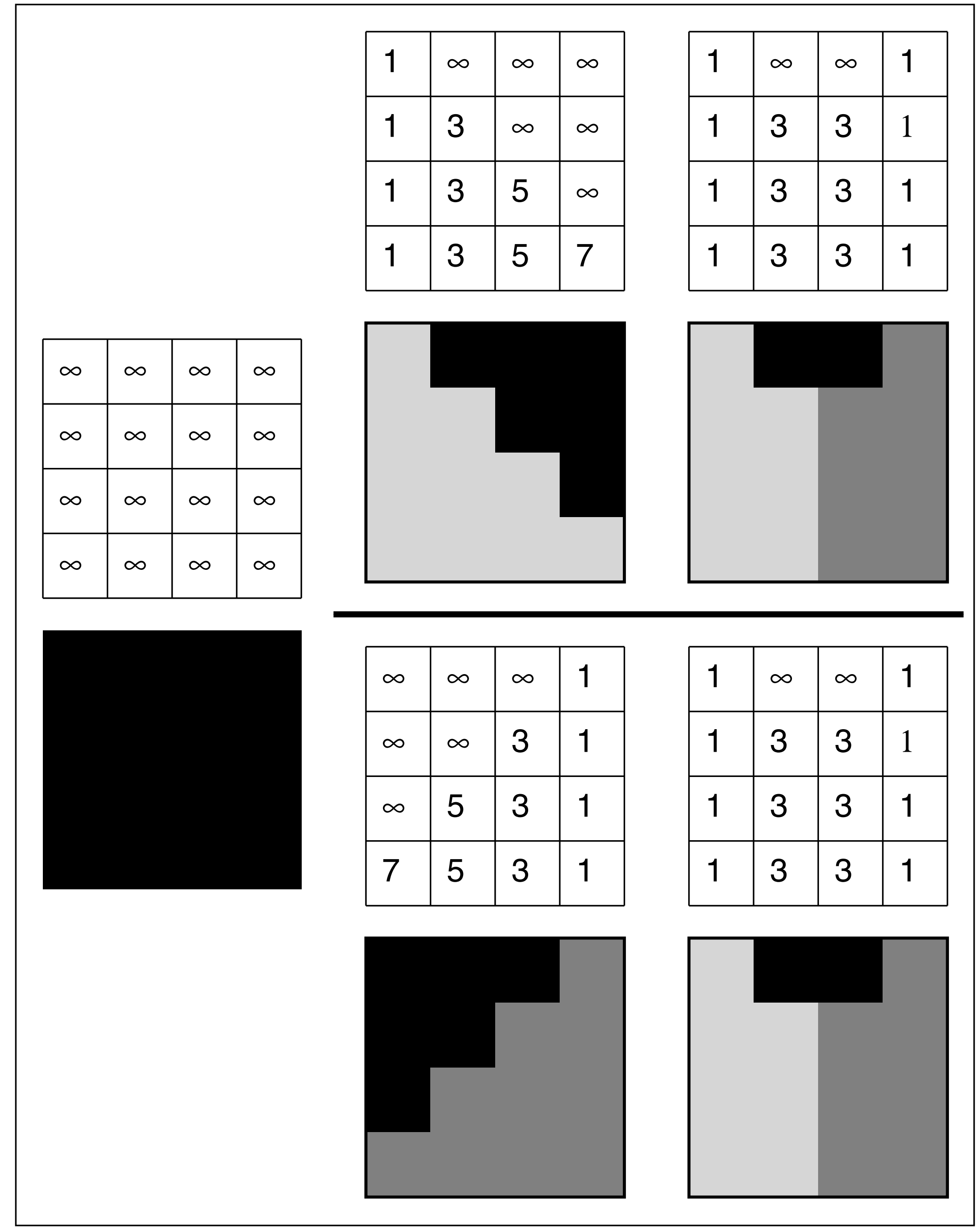
$$\begin{bmatrix} nx \\ ny \\ (n+f)z - fn \\ z \end{bmatrix} \boxed{/z} = \begin{bmatrix} \frac{n}{z}x \\ \frac{n}{z}y \\ \boxed{n + f - \frac{fn}{z}} \\ 1 \end{bmatrix}$$

homogenization
or perspective division

$$M_p^{-1} = \begin{bmatrix} \frac{1}{n} & 0 & 0 & 0 \\ 0 & \frac{1}{n} & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -\frac{1}{fn} & \frac{n+f}{fn} \end{bmatrix}$$

Z-Buffer

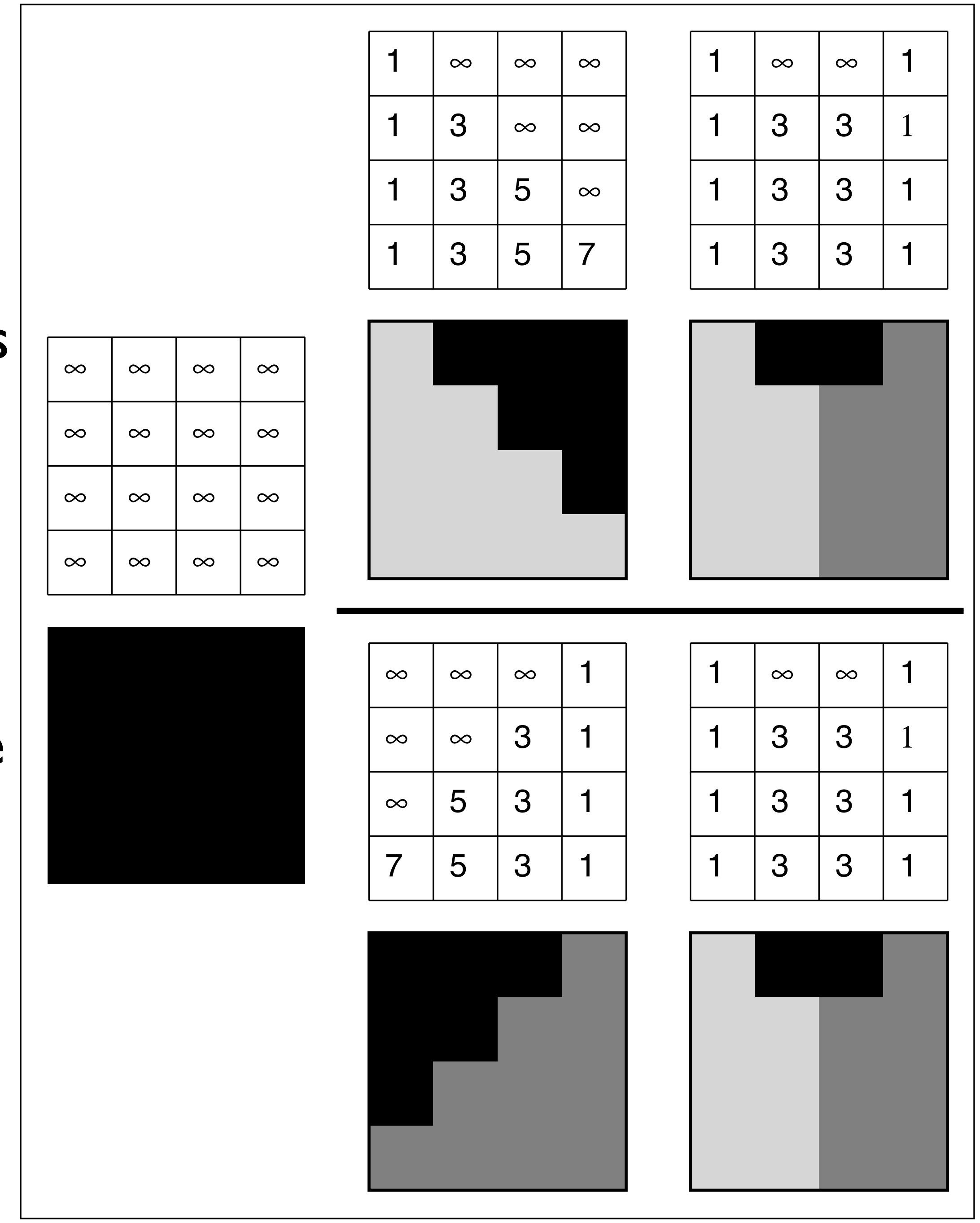
- BSP trees are not efficient for dynamic scenes (but for static ones!)
- For depth-handling of dynamic scenes, the most common technique that is being used is Z-buffering (or depth-buffering)
- The Z-buffer algorithm is extremely simple and usually hardware supported
- Each pixels stores the depth of the closest triangle that has been rasterized so far (infinity/ large value at an initial stage)
- If a new triangle has a lower depth value at a particular pixel, the pixel content and its Z value are replaced



two rastered triangles and corresponding values in Z-buffer

Z-Buffer

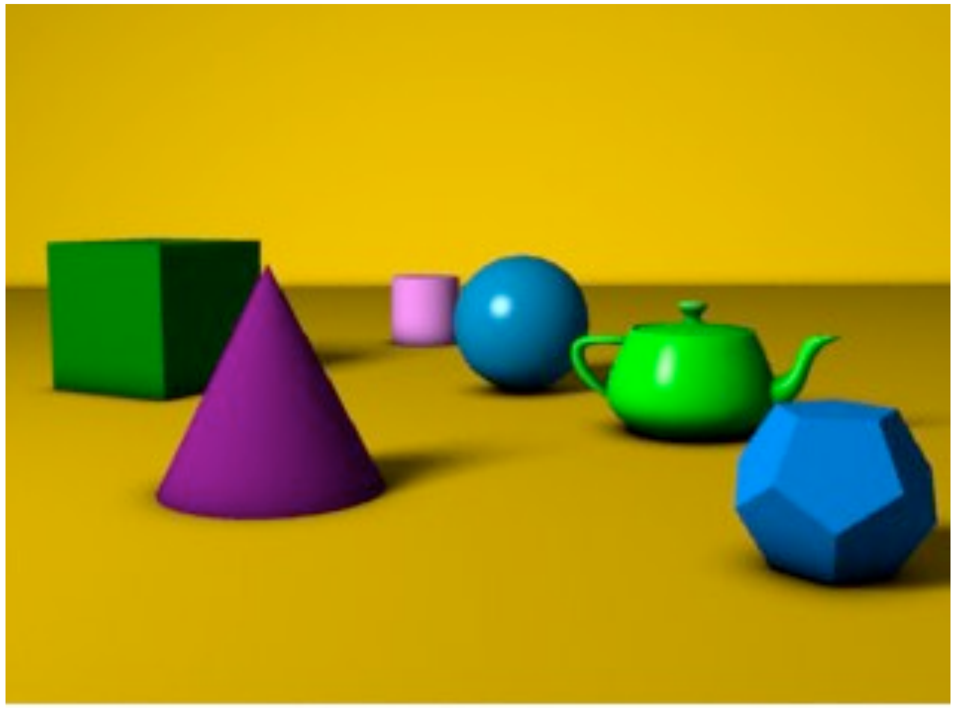
- Thus, a rastered image is stored in different buffers (more details in graphics pipeline class)
    - Frame-buffer for color, Z-Buffer for depth, ...
- In practice, the Z-buffer on graphics cards is today up to 32 bit (historically 8 bit)
- The precision of the Z-buffer plays a dominant role for rendering
- If the distance between two depth values falls below the precision resolution of the Z-buffer, errors will occur (called Z-buffer fighting)
- The entire Z-buffer resolution is used in between the near and far clipping planes of the virtual camera (see transformation and projections)
- Thus it makes sense to choose them well (dynamically derived from scene‘s bounding volume - not constant!)



two rastered triangles and corresponding values in Z-buffer

Z-Buffer

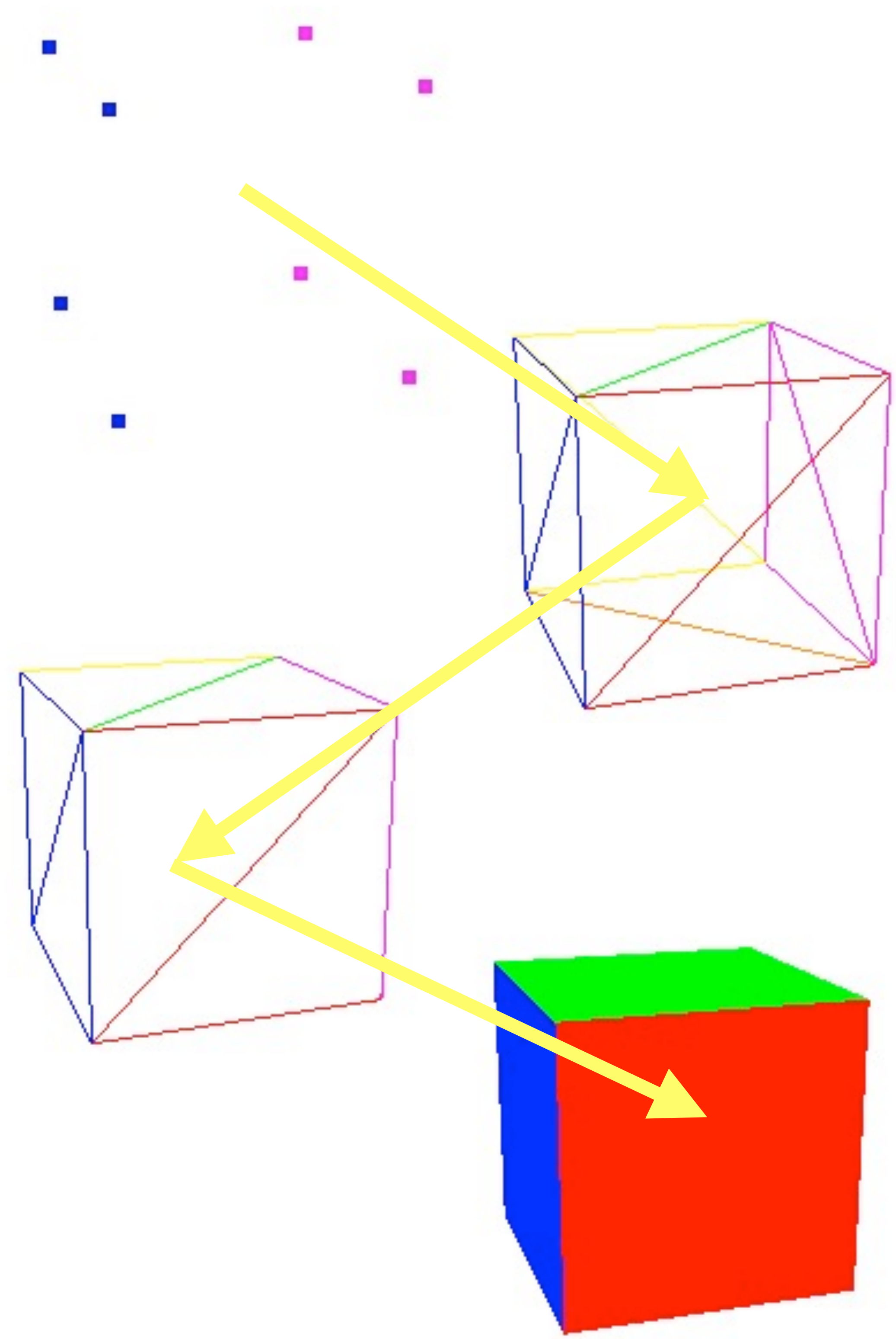
- In fact, not the Z components of the scene geometry are used directly, but the Z components after projection (we mentioned in the transformation and projection class, that these components preserve depth order)
- Thus, these components are indeed used - since they are important for depth handling, as mentioned earlier
- The only question is, how do we derive the Z values for pixels that are not exact projections of vertices?
- This leads us to the rasterization process itself



intensity shaded Z-buffer (dark=close/ small values, bright=far/large values)

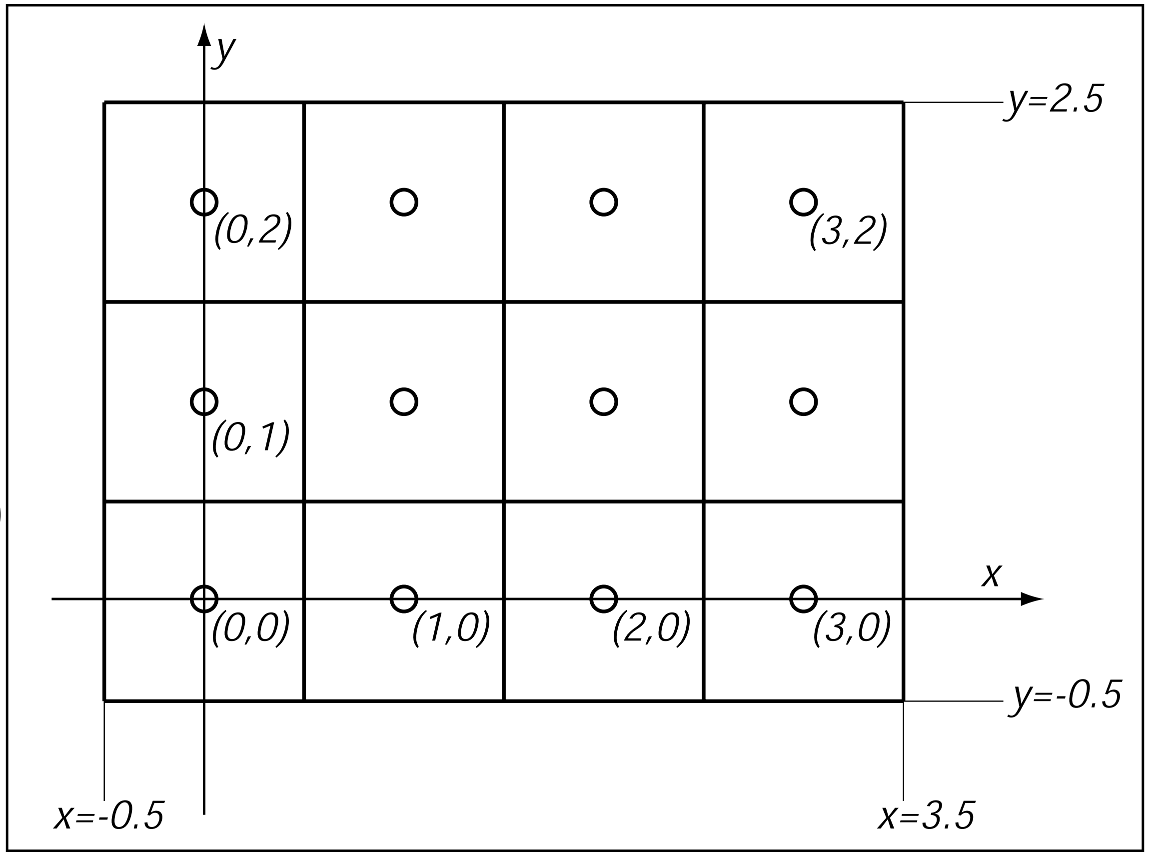
Rasterization

- In fact, we can only compute the 2D projections (in pixel coordinates) of transformed 3D points (in world coordinates), their transformed normals, and their Z-value that preserves depth order
- But this does not give us a useful image yet
- At least, we want to connect the vertices of visible triangles by line segments
- But how do we compute which pixels have to be turned on?
- And even better: we wish to also fill the pixels that belong to the visible area of a triangle



Rasterdisplays

- This process is in general called rasterization
- Today, graphics is displayed on raster displays (tomorrow, they might become holographic)
- Raster displays consist of a discrete grid of pixels
- For some displays, each pixel consists of even smaller elements -sometimes called RGB sub-pixels (eg. for LCD/CRT displays, but not for DLP displays)
- When projecting scene points, they will not necessarily map exactly to the discrete pixel positions - what do we do?
- How do we display other points on lines and surfaces? Discretizing them in 3D space and projecting the result would be an inefficient option



Drawing Lines

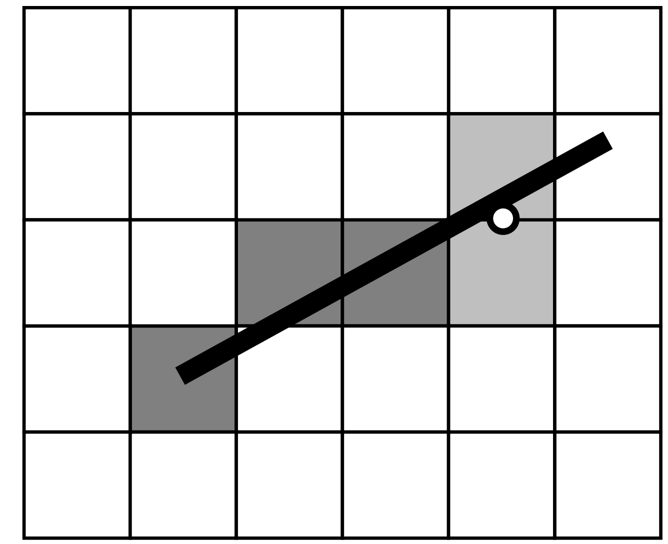
- A better solution is to project 3D vertices only (don‘t necessarily round them to the nearest pixel) and then fill in the remaining pixels in between them on the display grid

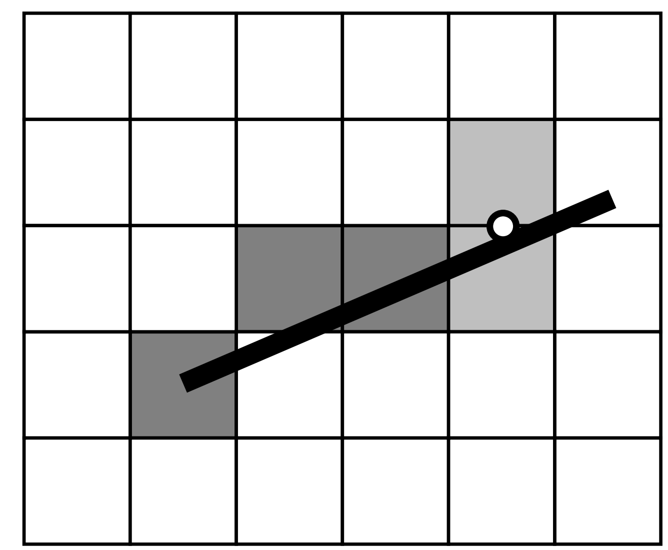
- The most common way of drawing lines, for example, is the midpoint algorithm (it draws the thinnest line possible without gaps)

- The following example is for m=0..1 (i.e. lines with a slope of 0-45 deg) - but similar constructs can be made for all other three slope ranges

  - We assume that $x_0<x_1$, otherwise we swap

  - By rasterizing the line from left to right, there are only two possibilities: draw next pixel at same height as previous, or one higher





$$f(x,y) = (y_0 - y_1)x + (x_1 - x_0)y + x_1 y_0 = 0$$

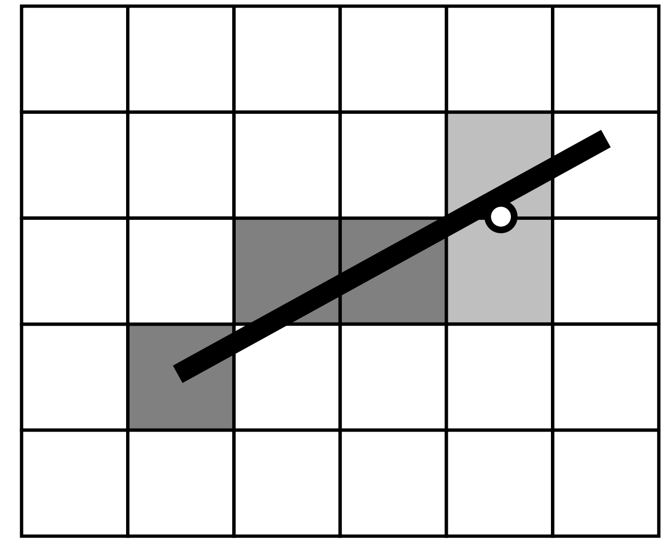
implicit line equation:
(x_0,y_0) is start point, (x_1,y_1) is end point

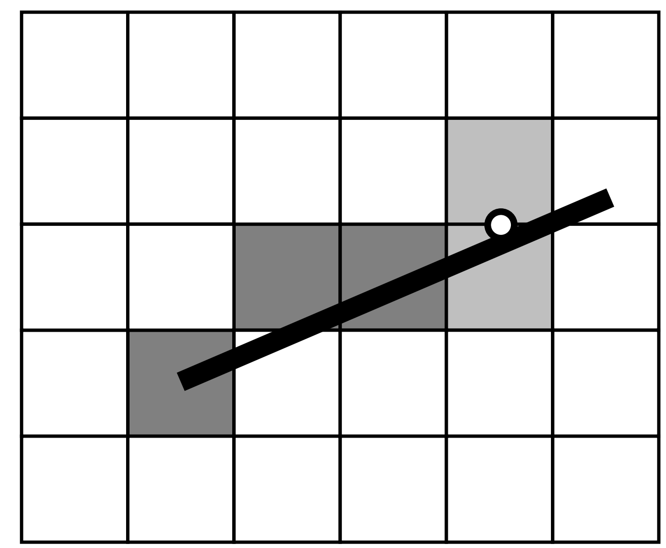
$$m = \frac{y_1 - y_0}{x_1 - x_0}$$

slope

Drawing Lines

- There will always exactly one pixel in each column (no gaps) but possibly multiple pixels in one row (remember: example is for m=0..1)
- How do we determine whether to select the same height or one higher for the next pixel?
  - The two candidate pixels are at (x+1,y) and (x+1,y+1)
  - The midpoint between the two candidates is (x+1,y +0.5)
  - If the line passes below midpoint then select lower candidate, else select upper candidate





$$f(x,y) = (y_0 - y_1)x + (x_1 - x_0)y + x_1 y_0 = 0$$

implicit line equation:
(x_0,y_0) is start point, (x_1,y_1) is end point

$$m = \frac{y_1 - y_0}{x_1 - x_0}$$

slope

Drawing Lines

- Another possibility is to use a parametric line equation
- For a slope range of m=-1,1 (i.e., -45 - +45 deg) we can proceed as follows (similar for the other case)
    - For all x from x_0 to x_1, compute t, compute y and round y to nearest pixel
- Since t goes from 0 (at the start) to 1 (at the end), it can also be used to interpolate different colors for intermediate pixels if the start and end colors are given

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_0 + t(x_1 - x_0) \\ y_0 + t(y_1 - y_0) \end{bmatrix}$$

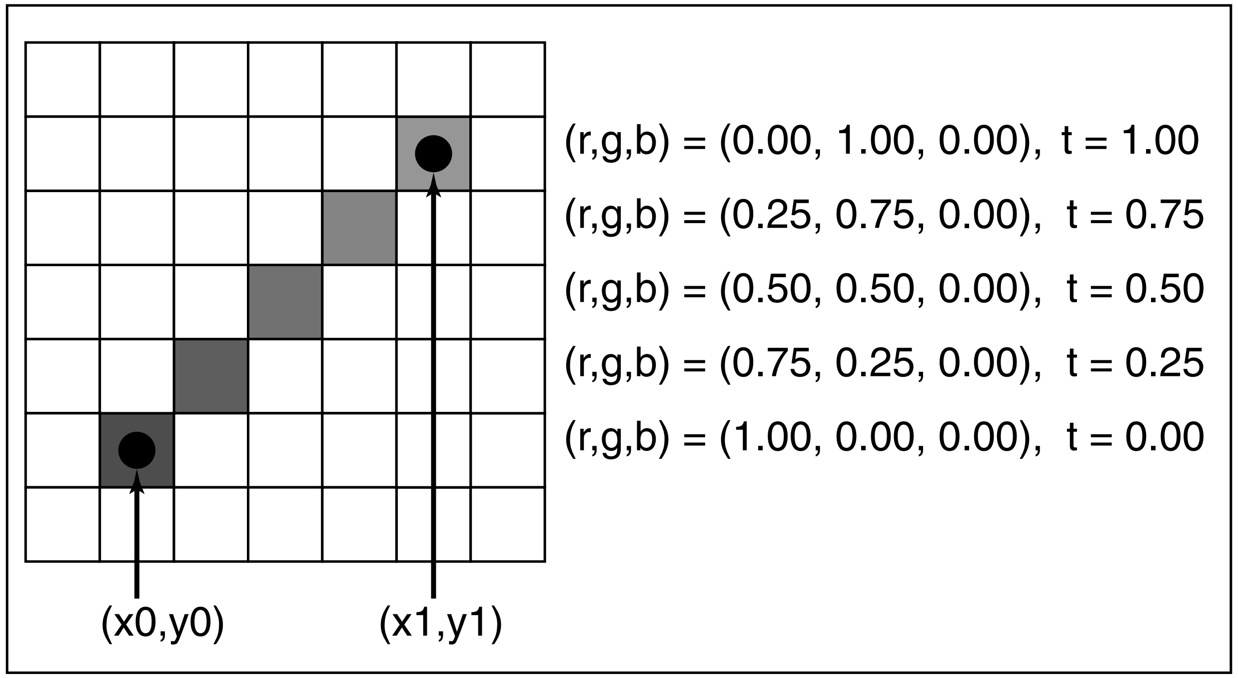
parametric line equation with parameter t

$$t = \frac{x - x_0}{x_1 - x_0}$$

parameter for given x

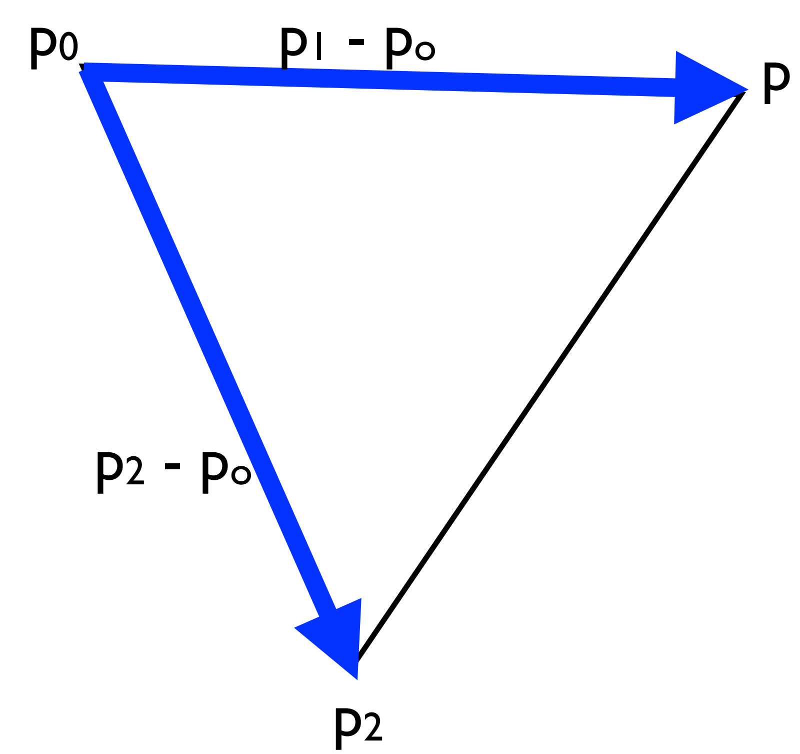
$$c_t = (1-t)c_0 + tc_1$$

example for interpolating colors (c_x are RGB vectors)



Rasterizing Triangles

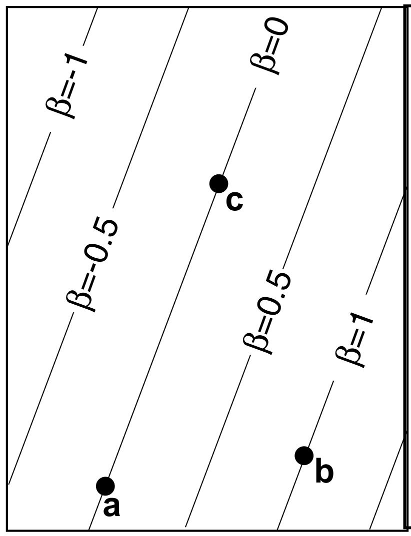
- These were only two example for drawing lines (others exist, such the Bresenham algorithm)
- How are triangles rasterized?
- We use interpolation in barycentric coordinates
  - They are the signed scaled distances from the lines through a triangle (see example for beta on the left)
  - They are also proportional to the areas of the three sub-triangles they span (see example on the right)

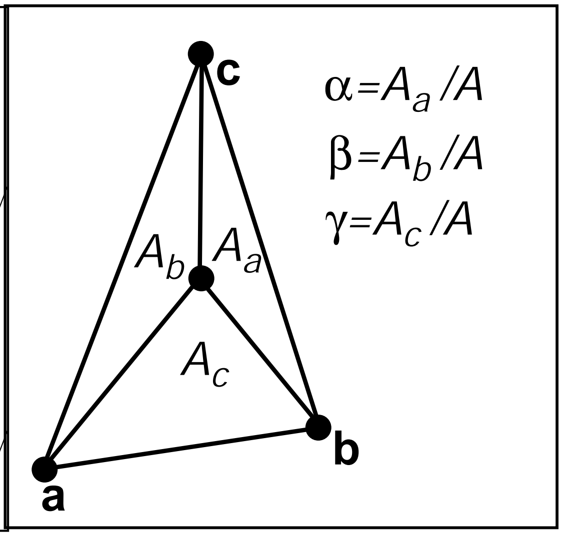


$$p = p_0 + \beta(p_1 - p_0) + \gamma(p_2 - p_0)$$

$$p = (1 - \beta - \gamma)p_0 + \beta p_1 + \gamma p_2$$

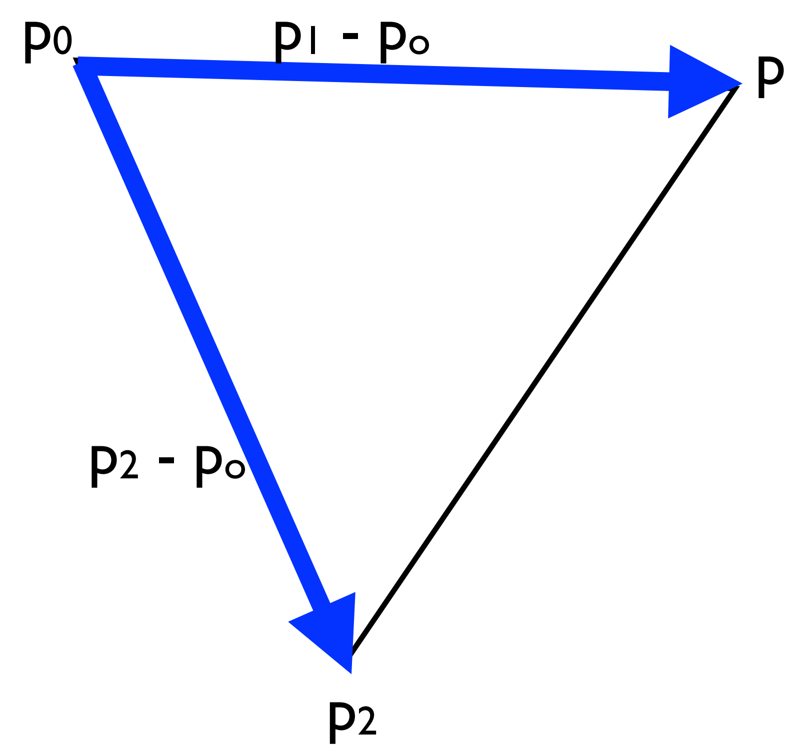
$$p = \alpha p_0 + \beta p_1 + \gamma p_2, \alpha = (1 - \beta - \gamma)$$





Rasterizing Triangles

- If the vertices have different properties, such as X,Y screen coordinates (from projection), Z-values (for Z-buffering), RGB colors, normal vectors, etc., they all can be interpolated for points inside the triangle through their barycentric coordinates



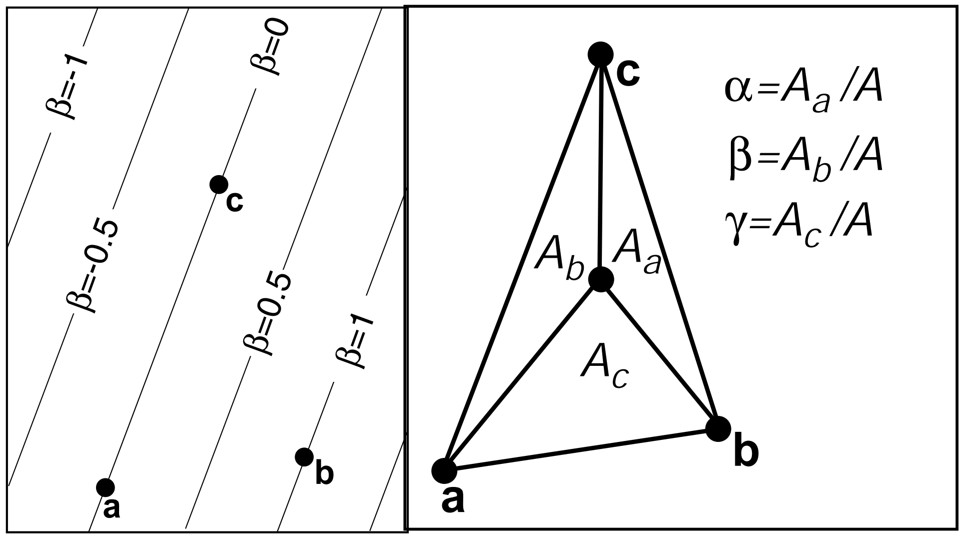
- Brute force:

for all x
  for all y
    compute coordinates
    if barycentric coordinates in interval 0..1 (for a triangle)
      interpolate parameters using barycentric coordinates
      daw pixel with interpolated parameters
    endif
  endfor
endfor

$$p = p_0 + \beta(p_1 - p_0) + \gamma(p_2 - p_0)$$

$$p = (1 - \beta - \gamma)p_0 + \beta p_1 + \gamma p_2$$

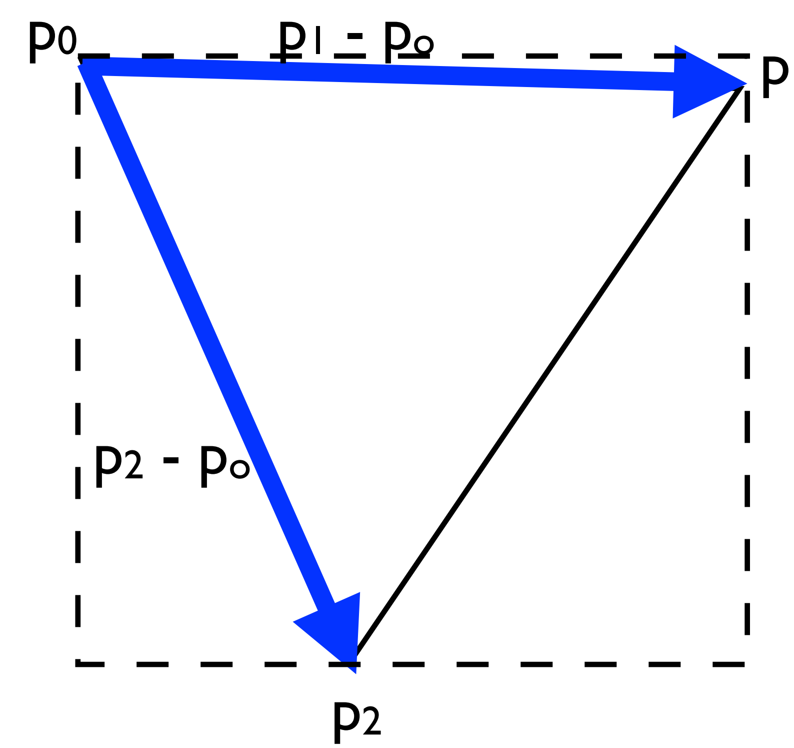
$$p = \alpha p_0 + \beta p_1 + \gamma p_2, \alpha = (1 - \beta - \gamma)$$

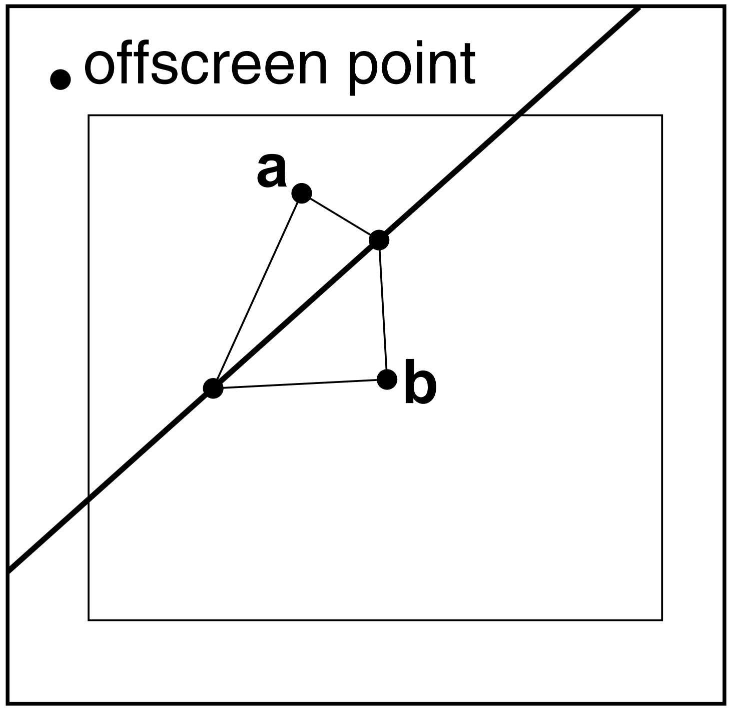


- Rasterizing the whole screen for each triangle is not efficient

Rasterizing Triangles

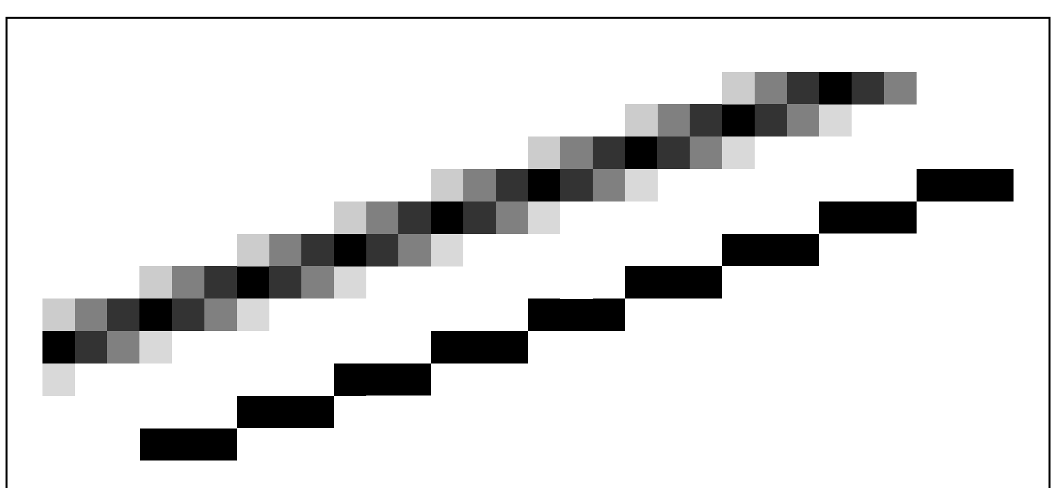
- A better way is to compute the bounding box for each triangle and only rasterize the triangle in the bounding box
- Alternatively, one can compute the bounding box of all triangles on screen, rasterize all pixels inside this single area, but check barycentric coordinates of each triangle (inside/outside) for each individual pixel
- But how do we deal with adjacent triangles (i.e., what about shared edges)?
  - One edge-pixels should be awarded to exactly one triangle
  - Define a fixed off-screen point and compare on which side of the edge line this point is with
  - The edge is awarded to the triangle whose third vertex is on the same side of the edge line as the off-screen point

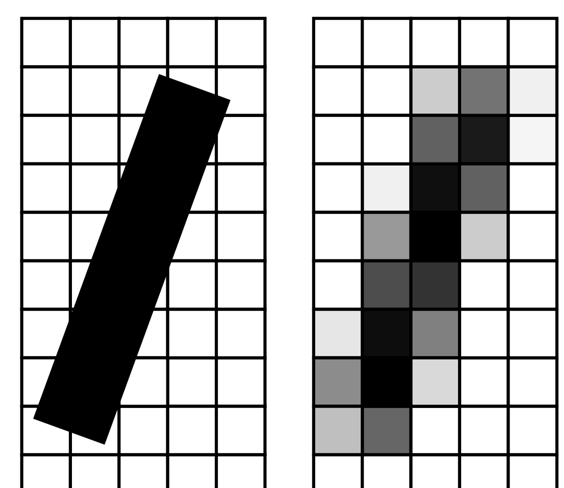




Aliasing

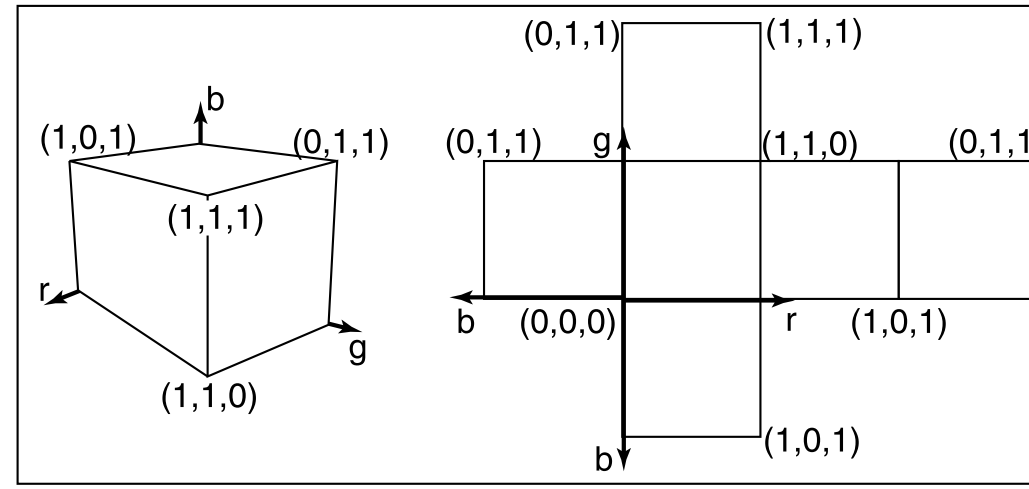
- One remaining risk is, that edge lines might go through off-screen point
- Solution: use a second off-screen point in case the first one is intersected
- One problem with all of these rasterization techniques, is that lines or edges of triangles appear jaggy
- Solution: antialiasing
- A simple solution would be to apply a box filter and convolution to smoothen the edges
- But there are several advanced antialiasing techniques (even hardware supported by your graphics card)



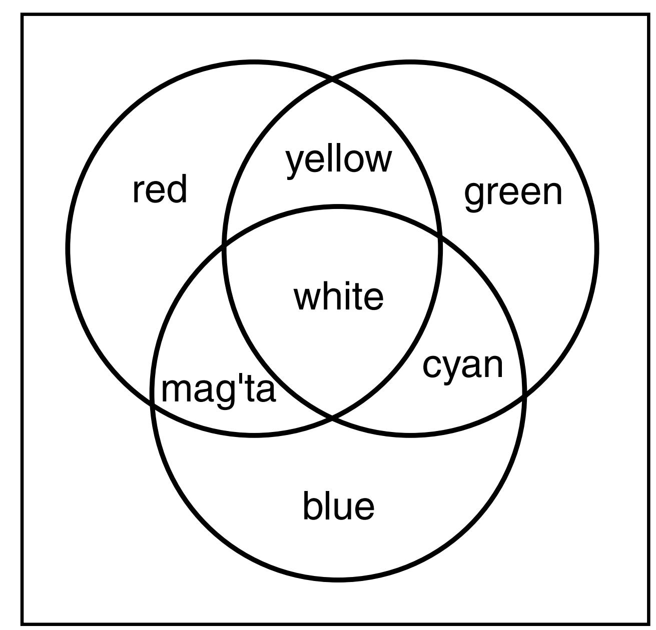


Rendering and Displaying Colors

- Many different parameterization schemes for color exist (called color spaces)
- In computer graphics, the most common one is the RGB space, since it is used by most raster displays
- RGB stands for Red, Green, Blue - the three primary colors used by such displays for additive color mixing
- Normal graphics cards allow 256 (i.e., 8 bit) tonal values for each color channel, and there form 2^{24} different colors can be addressed
- However, if and how they are displayed depends on the display!



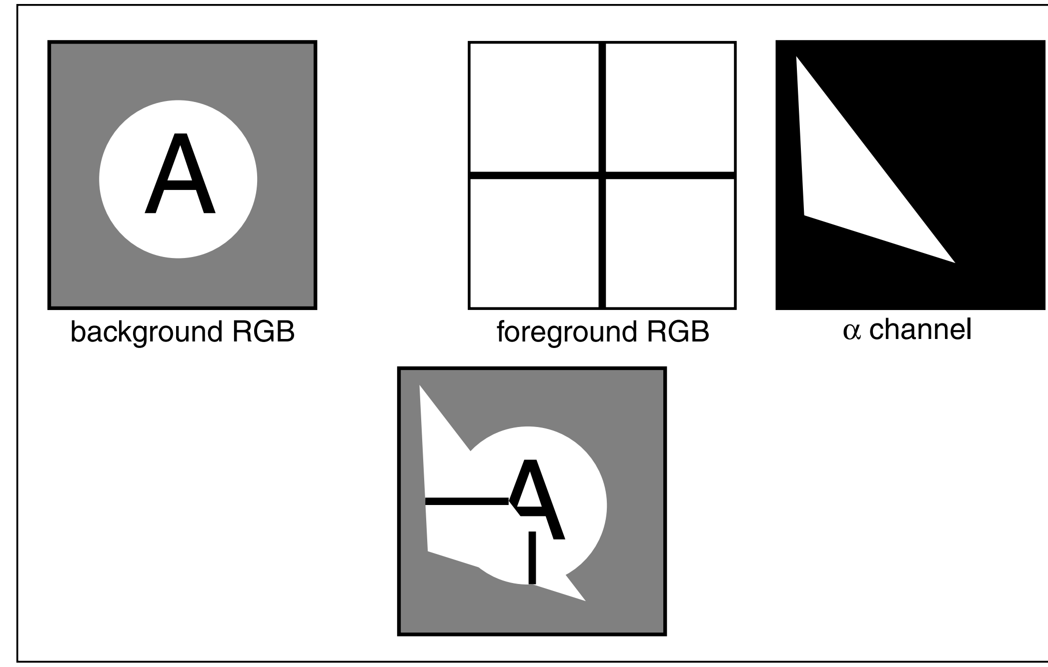
RGB color space (RGB cube)



additive color mixing

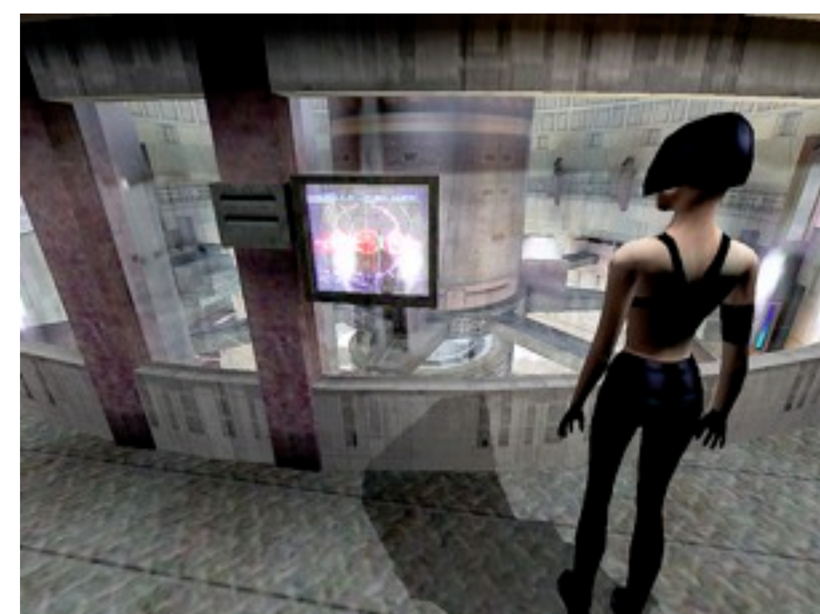
Alpha Blending

- Besides the three color channels (RGB) an additional channel is usually supported by common rendering pipelines (actually, there are several more as we will see later)
- It is used for blending together differently rasterized portions (i.e., allows to simulate simple transparency effects)
- It is called alpha channel and stors one normalized weight (alpha) per pixel
- For example, to blend the overlapping region of an opaque background with a semi-transparent foreground, their colors are weighted and added during rasterization
- There are several different blending functions
- If alpha is 0 or 1, we simply stencil our portions instead of blending them



$$c = \alpha c_f + (1-\alpha) c_b$$

c_f is the color of the foreground and c_b is the color of the background



...more cool graphics (with alpha blending)

Course Schedule

Type	Date	Time	Room	Topic	Comment
C1	01.03.2016	13:45-15:15	HS 18	Introduction and Course Overview	Conference
C2	15.03.2016	13:45-15:15	HS 18	Transformations and Projections	Easter Break
C3	05.04.2016	13:45-15:15	HS 18	Raster Algorithms and Depth Handling	
C4	12.04.2016	13:45-15:15	HS 18	Local Shading and Illumination	
C5	19.04.2016	13:45-15:15	HS 18	Texture Mapping Basics	
C6	26.4.2016	13:45-15:15	HS 18	Advanced Texture Mapping & Graphics Pipelines	
C7	03.05.2016	13:45-15:15	HS 18	Intermediate Exam	
C8	09.05.2016	17:15-18:45	HS 18	Global Illumination I: Raytracing	
C9	10.05.2016	13:45-15:15	HS 18	Global Illumination II: Radiosity	Conference / Holiday
C10	31.05.2016	13:45-15:15	HS 18	Volume Rendering	
C11	07.06.2016	13:45-15:15	HS 18	Scientific Data Visualization	
C12	14.06.2016	13:45-15:15	HS 18	Curves and Surfaces	
C13	21.06.2016	13:45-15:15	HS 18	Basics of Animation	
C14	28.06.2016	13:45-15:15	HS 18	Final Exam	
C15	04.10.2016	13:45-15:15	TBA	Retry Exam	

Thank You!