- limitations: . a pinhole-camera as used in image computation is missing some features of real-world-lens cameras
  . e.g.: depth-of-field, vignetting (fix with darker overlay), flare of the sun

Rasterization

"How can we render 2D or 3D graphic very time-efficient?"

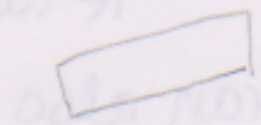
Definition:
- given a primitive form (like a 2D line, circle, polygon), which pixels on a raster display are covered by this primitive → extension: which part of pixel
- it's used for realtime-rendering (e.g. for computer games)

Line Rasterization:

- assume: . pixels are points on 2D-integer-grid
  . endpoints of lines at pixel coordinates
  . slope (steigung) of lines is ≤ 1
  . line thickness is one pixel
- functions: . define lines with startpoint (x_0, y_0) & endpoint (x_e, y_e) through function $y = mx + b$ with $m = \frac{dy}{dx}$
  . algorithm: brute-force, that means iterate over x-values (x_0 to x_e) then calculate y
  . problem: m and y_i are floating-point values and multiplications are expensive
- digital differential analyzer (dda): . same definition of lines
  . algorithm: incremental, that means we increment $x + 1$ and $y + m$
  → avoiding multiplication
  → but rounding errors accumulate
- bresenham: . define lines implicitly: $F(x,y) = dy\,x - dx\,y + dx\,B = 0$
  → if $F(x,y) < 0$ point above line, $F(x,y) > 0$ below
  . algorithm: midpoint formulation, measure ther vertical distance of midpoint M to line $d_{i+1} = F(M_{i+1}) = F(x_i + 1, y_i + \frac{1}{2})$
  → increment $x + 1$ and if $d_i > 0$ also increment y

· to eliminate fractions and create integer bresenham multiply by 2 $F(x_0, y_0) = 2(ax_0 + by_0 + c)$ → $d_{start} = 2a + b$

→ very useful algorithms because you don't need Multiplication, division or floating-point-values

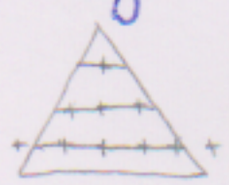
- thick lines: · pixel replication 
  · moving pen
  · filling area with block 
- additional: · bresenhams algo can also be used for circles

Polygon Rasterization

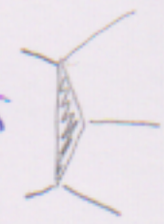
"How can we rasterize polygons e.g. triangles?"

- there are two approaches: tessellate the polygons into triangles or use scan-conversion

- Triangle Rasterization
  - let's start with simplest polygon shape
  - brute-force: this is a way of rasterizing a triangle, where you iterate over each pixel in the corresponding bounding box and check if that pixel is inside the triangle
  - incremental: implicit edge functions to describe the triangle, if every $F_i(x,y) \leq 0$ the point is inside the triangle
  - scan conversion: create horizontal lines from the left edge to the right edge, then fill in every pixel that lies on these lines (fill can also be called fragment)

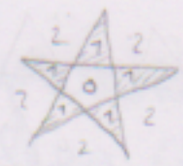


  - problems: · if the edges of the triangles are not straight lines after being filled there can be gaps
    · also how to deal with edges (if $F_i(x,y) \leq 0$ we have too many pixels, if $F_i(x,y) < 0$ we have too (little)



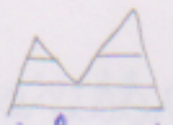
- Polygon Rasterization
  - one option is now to tessellate the polygon into triangles and then use options from above
  - another option is to check if pixel is directly in polygon by using one of these inside-outside-tests

- even-odd-rule: count the number of edges that a ray starting at point P crosses
  if the number is odd, the point is inside



- winding-number-rule: you need to know the direction of the edge you're crossing,
  add 1 if its going upwards, -1 if downwards
  if the number is ≠ 0, the point is inside

- you can also use scan-conversion for polygons:
  here you then need to check if the line is inside the polygon or not → could use scanline algo (skipped)



Clipping