Computer Graphics

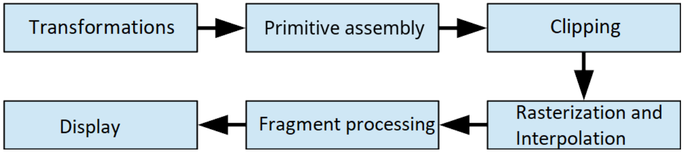
Ágoston Sipos
`siposagoston@inf.elte.hu`

Eötvös Loránd University
Faculty of Informatics

2025-2026. Fall semester

Table of contents

Graphics pipeline

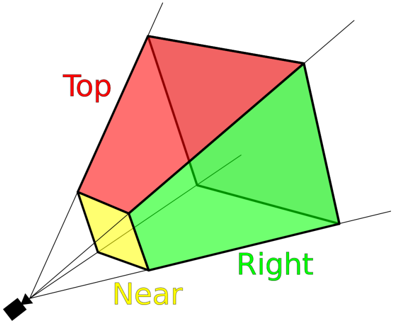


Incremental image synthesis

- Incremental principle
- We looked at the graphics pipeline in terms of transformations and we also looked at a solution in screen space to the visibility problem
- Now we look at the topic of clipping and rasterization

Clipping

- We saw that we need the clip for two reasons:
  - Filtering out degenerate cases (see e.g. last lecture's central projection)
  - We don't want to calculate unnecessarily (we should not waste resources on what we cannot see)
- During clipping, we filter out geometric elements outside the viewing frustum

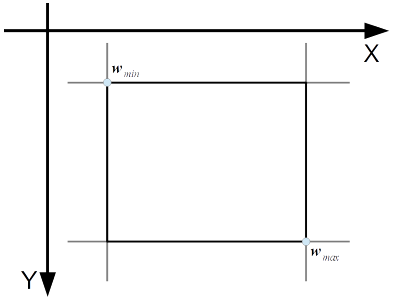


Clipping in 2D

- We examine the clipping of simple geometric elements (points and segments), now in the plane
- We need a region, *on which* we clip – from the simple (axis-aligned rectangle) to the general case (convex polygon)
- First we clip points, then we try to clip segments using the results from the point clipping

Clipping points with an axis-aligned rectangle

- We need to decide for a point $\mathbf{x} = [x, y]^T$ whether it is inside or outside the clipping region
- In the simplest case, our clipping region is an axis aligned rectangle
- We can easily represent it, with the two endpoints of its diagonal: $\mathbf{w}_{min} = [x_{min}, y_{min}]^T$, $\mathbf{w}_{max} = [x_{max}, y_{max}]^T$



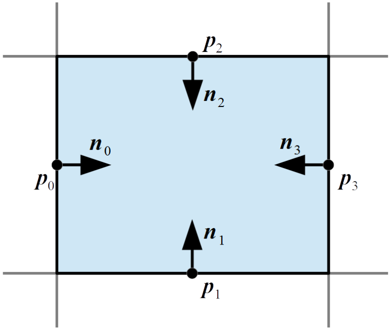
Clipping points with an axis-aligned rectangle

- Then **x** is inside the axis aligned rectangle if

$$x \in [x_{min}, x_{max}] \wedge y \in [y_{min}, y_{max}]$$

Clipping points with convex quadrilateral

- Let our clipping region be a convex quadrilateral
- We can represent it as an intersection of four half-planes: with half-planes defined by the bounding lines and their normals pointing inward
- We can then represent our clipping region as a point-normal pairs $(\mathbf{p}_i, \mathbf{n}_i)$, $i = 0, .., 3$, where all normals should be pointing inward



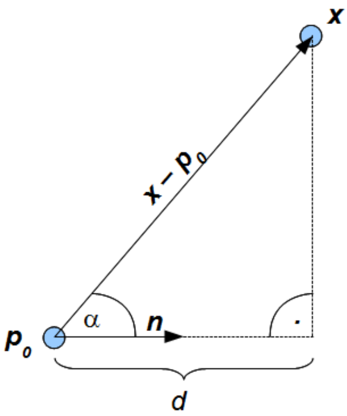
Clipping points with convex quadrilateral

- Then $\mathbf{x} = [x, y]^T$ is inside of the clipping region, if

$$\langle \mathbf{x} - \mathbf{p}_i, \mathbf{n}_i \rangle \geq 0 \ , \ \ i = 0, 1, 2, 3$$

- In this case every half-plane represented by $(\mathbf{p}_i, \mathbf{n}_i)$ contains $\mathbf{x}$ (either it is on the bounding line or in the direction of the normal)
- That is, we need to check only the sign of the signed orthogonal projections of the vectors $\mathbf{x} - \mathbf{p}_i$ on $\mathbf{n}_i$ normals

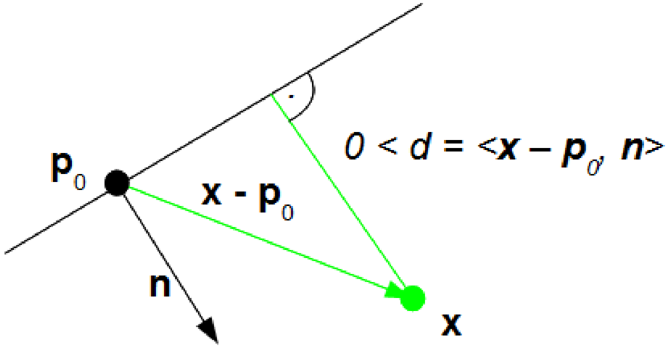
Signed orthogonal projection



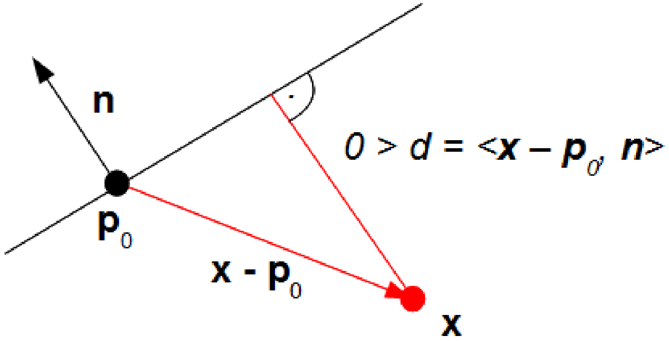
$$\cos\alpha = d \,/\, |\boldsymbol{x} - \boldsymbol{p}_0|$$

$$d = |\boldsymbol{x} - \boldsymbol{p}_0| \cos\alpha$$
$$= \langle \boldsymbol{x}-\boldsymbol{p}_0, \boldsymbol{n} \rangle, \ |\boldsymbol{n}| = 1$$

Signed orthogonal projection – point inside the half-plane


$\mathbf{p}_0$
$0 < d = \langle \mathbf{x} - \mathbf{p}_0, \mathbf{n} \rangle$
$\mathbf{x} - \mathbf{p}_0$
$\mathbf{n}$
$\mathbf{x}$

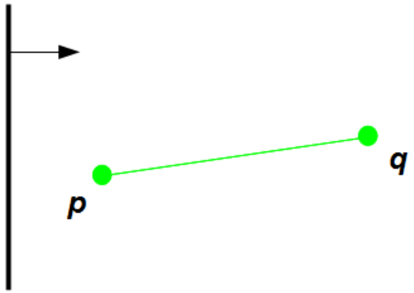
Signed orthogonal projection – point outside the half-plane



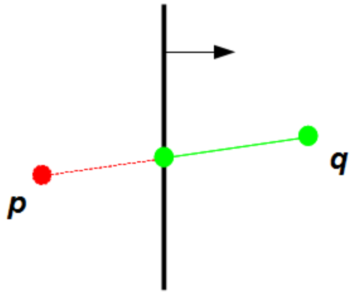
Clipping segment with half-plane

- Clip the two endpoints of the segment
  - **If** both are inside, **then** we keep the segment
  - **If** both are outside, **then** we discard the segment
  - **If** one is inside and the other is outside, **then** we keep the one inside, we replace the one outside with the intersection point of the segment and the half-plane

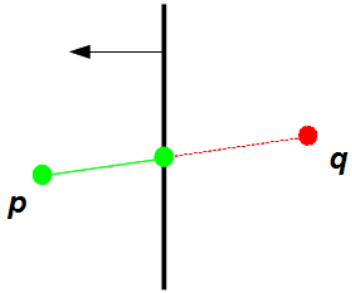
Clipping segment with half-plane



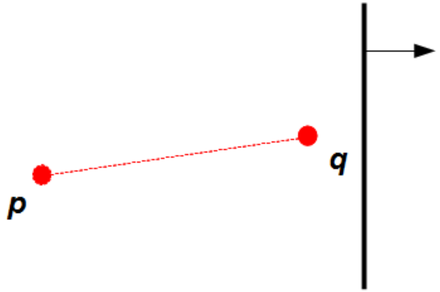
Clipping segment with half-plane – new endpoint!



Clipping segment with half-plane – new endpoint!



Clipping segment with half-plane



Clipping segment with convex quadrilateral

For every bounding half-plane:

- Clip the endpoints of the segment with the half-plane
  - **If** both are inside, **then** we do not modify the endpoint and continue
  - **If** both are outside, **then** we discard the segment and return
  - **If** one is inside the other is outside, **then** we keep the point inside, we replace the other with the intersection point of the segment and the half-plane and continue

Clipping segment with an axis-aligned rectangle

- In the third case (when an intersection has to be calculated), in the axis aligned case our task is simple
- For example, consider clipping with the left boundary $\Rightarrow$ then we need to find t for which the x-coordinate of $\mathbf{x}(t)$ is x_{min}
- That is from $x_{min} = x_1 + t(x_2 - x_1)$ we get $t = \frac{x_{min}-x_1}{x_2-x_1}$
- Then the coordinates of the new end point $\mathbf{x}(\frac{x_{min}-x_1}{x_2-x_1}) = \begin{bmatrix} x_{min} \\ y_1 + \frac{x_{min}-x_1}{x_2-x_1}(y_2 - y_1) \end{bmatrix}$

Clipping segment with convex quadrilateral

- As previously seen, the endpoints must be clipped on the current half-plane
- If a new end point needs to be calculated, we simply need to substitute the (current) parametric equation of the segment into the implicit equation of the line defining the two half-planes.
- That is, for the current $i \in \{0, 1, 2, 3\}$ solve the equation

$$\langle \mathbf{x}(t) - \mathbf{p}_i, \mathbf{n}_i \rangle = 0$$

  where, with $t \in [0, 1]$

$$\mathbf{x}(t) = \mathbf{x}_1 + t(\mathbf{x}_2 - \mathbf{x}_1)$$

- Then the intersection parameter is $t = \frac{\langle \mathbf{p}_i - \mathbf{x}_1, \mathbf{n}_i \rangle}{\langle \mathbf{x}_2 - \mathbf{x}_1, \mathbf{n}_i \rangle}$

Clipping segment with convex polygon

- Same as with convex quadrilateral
- The only difference is that there are not four, but more (or only three) bounding lines
- Direction of normals should be consistent

Clipping segments

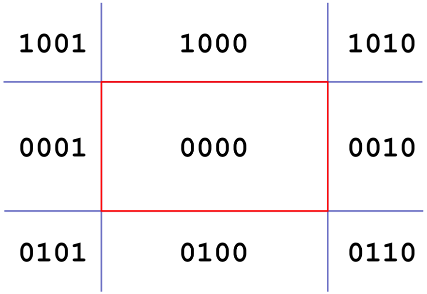
- According to the above, a segments must be clipped with every half-plane
- But there are many possibilities (four half-planes for quadrilaterals, two endpoints – any endpoint can fall anywhere)
- With the Cohen-Sutherland algorithm, we can decide which sides should we clip with

Cohen–Sutherland algorithm

- Used for clipping with axis-aligned rectangles (but also works on convex quadrilaterals)
- Assign a 4-bit (so-called TBRL) code to each of the endpoints:
  - T $=$ 1 if the point is above the window, otherwise 0
  - B $=$ 1 if the point is below the window, otherwise 0
  - R $=$ 1 if the point is to the right of the window, otherwise 0
  - L $=$ 1 if the point is to the left of the window, otherwise 0

Cohen–Sutherland algorithm

code = Top, Bottom, Right, Left

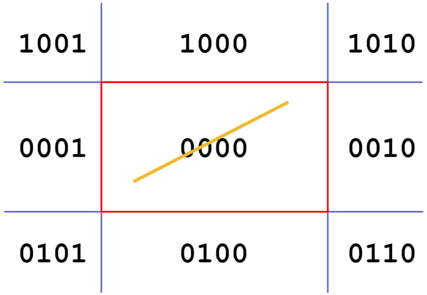


Cohen-Sutherland algorithm

- In the axis aligned case it's easy to calculate the TBRL code: `code` $= (y > y_{max}, y < y_{min}, x > x_{max}, x < x_{min})$
- Let the TBRL of the two endpoints $\texttt{code}_a$ and $\texttt{code}_b$
  - **If** $\texttt{code}_a$ OR $\texttt{code}_b == 0$ **then** both endpoints are inside the window $\Rightarrow$ we keep the segment
  - **If** $\texttt{code}_a$ AND $\texttt{code}_b$!= 0 **then** both endpoint are outside the window along a common side $\Rightarrow$ we discard the segment
  - **Otherwise**: we have to cut the segment (in TBRL code, the 1s tell us which side it can be on). We recalculate the code of the new endpoint and compare them again as above.

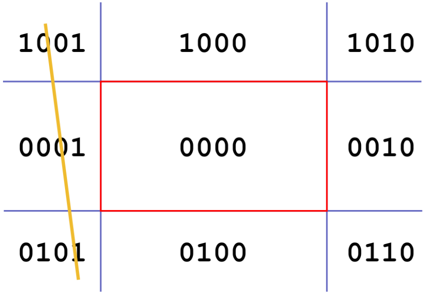
Cohen-Sutherland algorithm

$\texttt{code}_a$ OR $\texttt{code}_b == 0$:



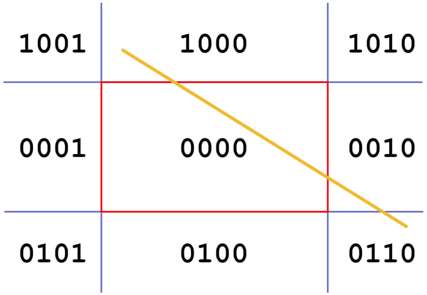
Cohen-Sutherland algorithm

$\texttt{code}_a$ AND $\texttt{code}_b$!= 0:



Cohen-Sutherland algorithm

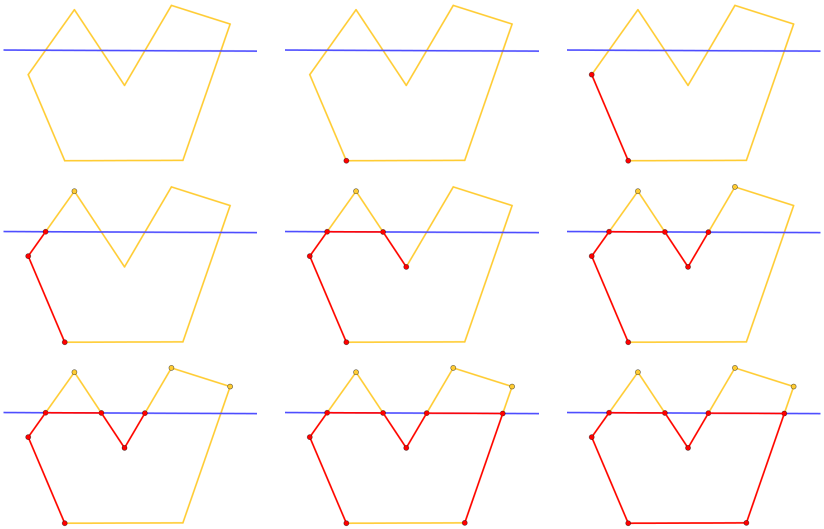
Otherwise:



Clipping polygon with convex polygon

- A polygon can be stored with an array of its vertices if the traversal order is fixed
- Let our clipping region be a convex polygon
- Let's see how we can clip a polygon with convex polygon

Sutherland-Hodgman polygon clipping



Sutherland-Hodgman polygon clipping

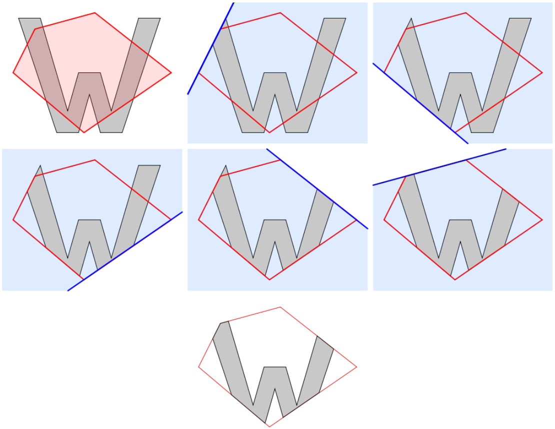
- Clipping with one side: Let **p**[] be the array of vertices of the polygon that we clip, and **q**[] will be the array of the output polygon (i.e. the clipped polygon)
- Let n be the number of vertices, and assume that $\mathbf{p}[0] = \mathbf{p}[n]$
- Let's go along the edges of the polygon to be cut:
  - **If** $\mathbf{p}[i]$ and $\mathbf{p}[i+1]$ are inside **then**
    $\Rightarrow$ add $\mathbf{p}[i]$ point to the **q**[] output polygon
  - **If** $\mathbf{p}[i]$ is inside and $\mathbf{p}[i+1]$ is outside **then**
    $\Rightarrow$ add $\mathbf{p}[i]$ to **q**[] output polygon and add the intersection of the segment defined by $\mathbf{p}[i], \mathbf{p}[i+1]$ and the line of the current side to **q**[]
  - **If** $\mathbf{p}[i]$ is outside and $\mathbf{p}[i+1]$ is inside **then**
    $\Rightarrow$ add the intersection of the side's line and the segment defined by $p[i], p[i+1]$ to **q**[]
  - **If** $p[i]$ and $\mathbf{p}[i+1]$ are outside **then** $\Rightarrow$ *SKIP*
- This is repeated for all sides of the clipping polygon (the result of the clipping with the previous side is the input for the next clipping)

Sutherland-Hodgman polygon clipping

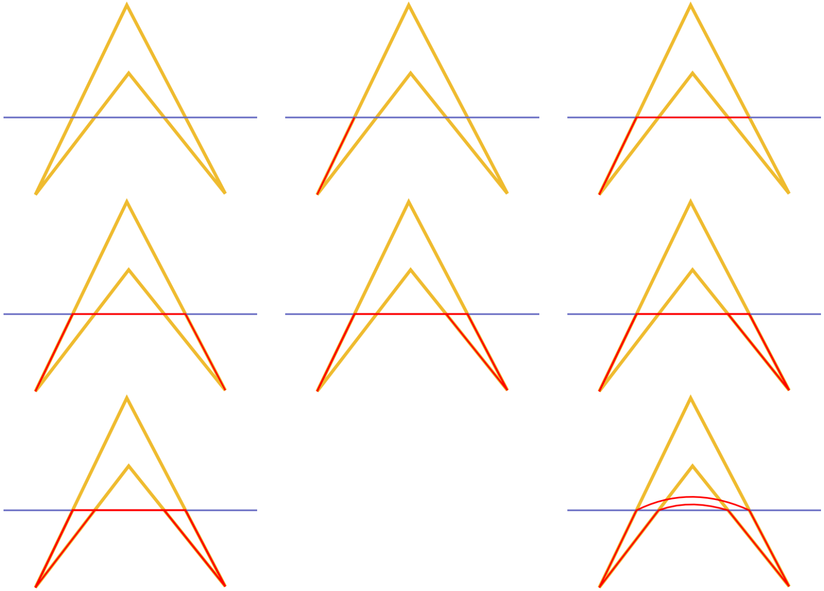
```
PolygonClip(in p[n], out q[m], in line) {
  m = 0;
  for( i=0; i < n; i++) {
    if (IsInside(p[i])) {
      q[m++] = p[i];
      if (!IsInside(p[i+1]))
        q[m++] = Intersect(p[i], p[i+1], line);
    } else {
      if  (IsInside(p[i+1]))
        q[m++] = Intersect(p[i], p[i+1], line);
    }
  }
}
```

Sutherland-Hodgman polygon clipping

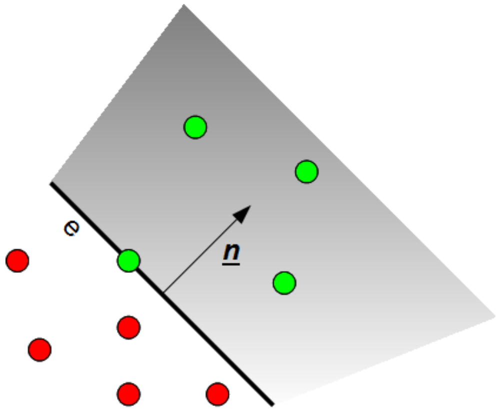
The clipping is done on all edges of the clipping polygon



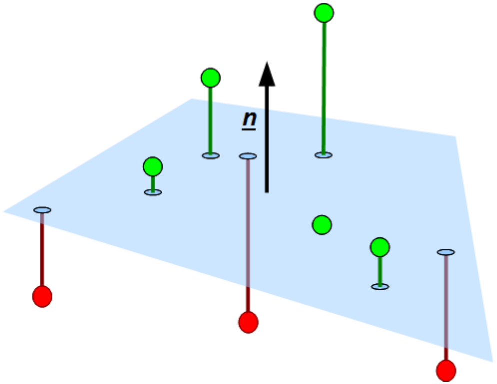
Sutherland-Hodgman problem: clip a concave polygon



Clipping in 2D

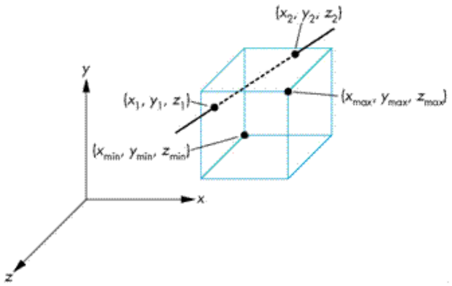


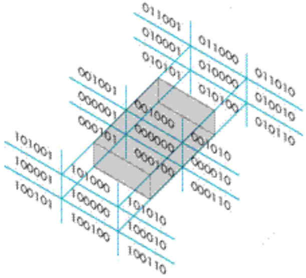
Clipping in 3D



Clipping in 3D

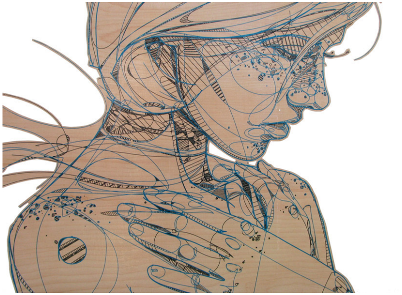
- It works the same way as before, but the implicit equations of $\langle \mathbf{x} - \mathbf{p}, \mathbf{n} \rangle \geq 0$ now define a *half-spaces*
- In space, it is even more worthwhile to clip with axis aligned boxes
- Cohen-Sutherland bitcodes are now 6 in length: in front, behind, top, bottom, right, left





Segment drawing

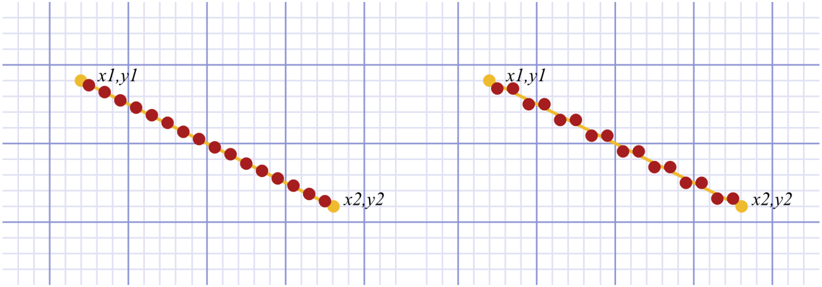
- One of the most used primitive
- It is crucial that we can draw them well
- Even better, if it's fast



Jason Thielke, jasonthielke.com

How do we draw segments?

- The two endpoints are given.
- How can we connect them?
- We only have miniature rectangles (we call them pixels).



Representing segments (again)

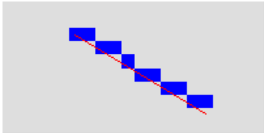
- Endpoints: $(x_1, y_1), (x_2, y_2)$
- Assume it is not vertical: $x_1 \neq x_2$.
- Segment equation:

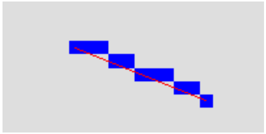
$$\begin{aligned} y &= mx + b, x \in [x_1, x_2] \\ m &= \frac{y_2 - y_1}{x_2 - x_1} \\ b &= y_1 - mx_1 \end{aligned}$$

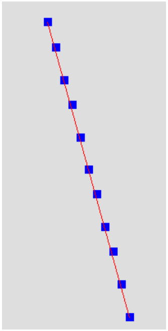
Naive algorithm

```
def line1(x1,y1,x2,y2, draw):
  m = float(y2-y1)/(x2-x1)
  x = x1
  y = float(y1)
  while x<=x2:
    draw.point((x,y))
    x += 1
    y += m
```

Naive algorithm







- due to rounding, the calculation are "off" by half a pixel
- `draw.point((x,y))` $\rightarrow$ wants `int` values, conversion is slow
- `m = float(y2-y1)/(x2-x1)` not accurate
- `y += m` $\rightarrow$ the error accumulates in y
- only works correctly with $|m| < 1$

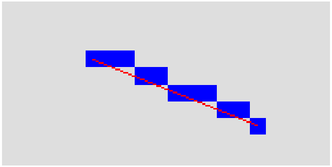
Improving the algorithm 1.

```python
def line2(x1,y1,x2,y2, draw):
  m = float(y2-y1)/(x2-x1)
  x = x1
  y = y1
  e = 0.0
  while x<=x2:
    draw.point((x,y))
    x += 1
    e += m
    if e >= 0.5:
      y += 1
      e -= 1.0
```

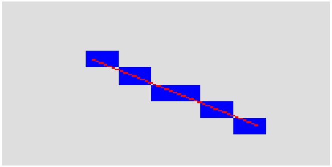
Improving the algorithm 1.

- Good: Always „hits” the endpoints
- Good: It moves more evenly in the y direction.
- Bad: We still use `float` values

Naive:



Improved:



Improving the algorithm 2.

```
def line3(x1,y1,x2,y2, draw):
  x = x1
  y = y1
  e = -0.5⟵
  while x<=x2:
    draw.point((x,y))
    x += 1
    e += float(y2-y1)/(x2-x1)⟵
    if e >= 0.0:⟵
      y += 1
      e -= 1.0
```

Improving the algorithm 3.

```
def line4(x1,y1,x2,y2, draw):
  x = x1
  y = y1
  e = -0.5*(x2-x1)⟵
  while x<=x2:
    draw.point((x,y))
    x += 1
    e += y2-y1⟵
    if e >= 0.0:
      y += 1
      e -= (x2-x1)⟵
```

Improving the algorithm 4.

```
def line5(x1,y1,x2,y2, draw):
  x = x1
  y = y1
  e = -(x2-x1)⟵
  while x<=x2:
    draw.point((x,y))
    x += 1
    e += 2*(y2-y1)⟵
    if e >= 0:
      y += 1
      e -= 2*(x2-x1)⟵
```

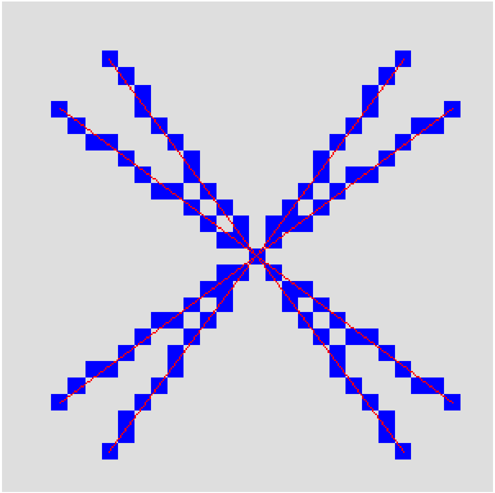
Improving the algorithm 4.

- This is the *Bresenham* algorithm (one of its special case)
- We accumulate the error separately in `e`
- We are not using `float` values
- It only works for $|m| \leq 1$
- And only if $x_1 < x_2$ and $y_1 < y_2$
- It can be generalized for segments with arbitrary slopes.

Bresenham algorithm

- The plane should be divided into eighths, each is a separate case.
- (The examples were: right and down)
- We have to decide, whether $|x_2 - x_1|$ or $|y_2 - y_1|$ is larger (where the segment is steeper).
- If $|y_2 - y_1|$ is larger, swap $x_i \leftrightarrow y_i$, we also use the swapped values during draw!
- If $x_1 > x_2$, then swap: $x_1 \leftrightarrow x_2$, $y_1 \leftrightarrow y_2$.
- We increase the e error with $|y_2 - y_1|$ in every step
- We step along y based on the sign of $y_2 - y_1$

Bresenham algorithm



Complete *Bresenham* algorithm 1.

```
def Bresenham(x1,y1,x2,y2, draw):
    steep = abs(y2-y1)>abs(x2-x1)
    if steep:
        x1, y1 = y1, x1
        x2, y2 = y2, x2
    if x1>x2:
        x1, x2 = x2, x1
        y1, y2 = y2, y1
    Dy = abs(y2-y1)
    if y1<y2:
        Sy = 1
    else:
        Sy = -1
```

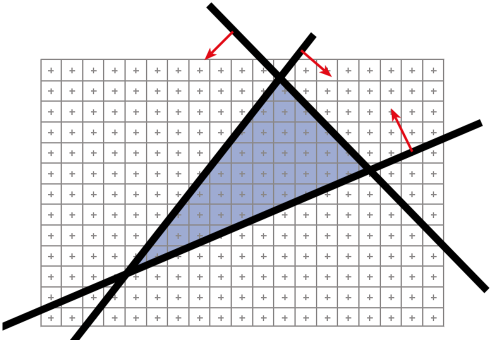
Complete *Bresenham* algorithm 2.

```
x = x1
y = y1
e = -(x2-x1)
while x<=x2:
  if steep:
    draw.point((y,x))
  else:
    draw.point((x,y))
  x += 1
  e += 2*Dy
  if e >= 0:
    y += Sy
    e -= 2*(x2-x1)
```

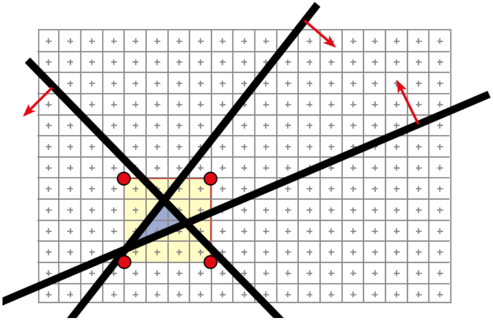
Triangle rasterization

- We can clip the sides of the triangle – now we fill it!
- If we give the vertices in a specific traversal order, then we can specify the half-planes (we can control the edge directions) $\rightarrow$ because, if (t_x, t_y) is the direction vector of the side, then $(-t_y, t_x)$ will be the normal
- For every pixel of the screen check if it is on the correct half-plane specified by the sides of the triangle!

Triangle rasterization



Triangle rasterization – smarter



Triangle rasterization

- It could be done even smarter, but: in practice, this brute-force approach can be used well!

Triangle rasterization

- We don't want to fill it with a fixed value, we want to interpolate the values from the vertices.
- Uses: color (Gouraud shading), texture coordinates, normal vectors
- Let a point in the triangle $p = \alpha p_1 + \beta p_2 + \gamma p_3$, with α, β, γ barycentric coordinates.
- We can then interpolate any other value in the same way:

$$c = \alpha c_1 + \beta c_2 + \gamma c_3$$

- This is the *Gouraud interpolation* (or barycentric interpolation)

Triangle filling 1.

```
for all x:
   for all y:
      $\alpha, \beta, \gamma$ = barycentric(x, y)
      if $\alpha \in [0,1]$ and $\beta \in [0,1]$ and $\gamma \in [0,1]$:
         c = $\alpha$c_1 + $\beta$c_2 + $\gamma$c_3
         draw.point((x, y), c)
```

Barycentric coordinates

- Barycentric coordinates can be calculated using the following formulas:

$$\begin{aligned}
f_{01}(x,y) &= (y_0 - y_1)x + (x_1 - x_0)y + x_0 y_1 - x_1 y_0 \\
f_{12}(x,y) &= (y_1 - y_2)x + (x_2 - x_1)y + x_1 y_2 - x_2 y_1 \\
f_{20}(x,y) &= (y_2 - y_0)x + (x_0 - x_2)y + x_2 y_0 - x_0 y_2
\end{aligned}$$

- Then the barycentric coordinates belonging to the x, y point:

$$\begin{aligned}
\alpha &= f_{12}(x,y)/f_{12}(x_0,y_0) \\
\beta &= f_{20}(x,y)/f_{20}(x_1,y_1) \\
\gamma &= f_{01}(x,y)/f_{01}(x_2,y_2)
\end{aligned}$$

Triangle filling 2.

```
x_min = min(floor(x_i))
x_max = max(ceiling(x_i))
y_min = min(floor(y_i))
y_max = max(ceiling(y_i))
for y in [y_min..y_max]:
  for x in [x_min..x_max]:
    $\alpha = f_{12}(x,y)/f_{12}(x_0,y_0)$
    $\beta = f_{20}(x,y)/f_{20}(x_1,y_1)$
    $\gamma = f_{01}(x,y)/f_{01}(x_2,y_2)$
    if $\alpha>0$ and $\beta>0$ and $\gamma>0$:
      c = $\alpha$c_1 + $\beta$c_2 + $\gamma$c_3
      draw.point((x,y),c)
```

Triangle filling 2.

- Speed up: it is unnecessary to examine every x, y point, it is enough to go through the rectangle containing the triangle.
- Incremental theorem:
  - It's still slow, we don't exploit the fact that we go in order on x and y.
  - What are these f-s?
  - All have the $f(x, y) = Ax + By + C$ form.
  - Then $f(x + 1, y) = f(x, y) + A$, and
  - $f(x, y + 1) = f(x, y) + B$
- Implementation: *homework*

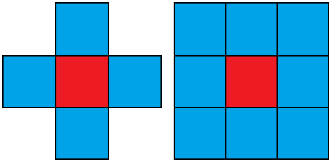
Flood-fill

- Suitable for filling any polygon with an already rasterized contour.
- Input: rasterized image + one of its points.
- Brute-force: from the specified starting point, we work recursively:
    - Is the color of the current point the same as the color of the starting point?

      No we stop

      Yes color it, and
    - we start again for every neighbor.

Flood-fill – neighbors



- Four neighbors: up, down, right, left
- Eight neighbors: the previous four + the corners
- Recursion is very rough: in practice there are smarter algorithms $\rightarrow$ active edge list etc.