

Számítógépes Grafika

8. előadás: vágás, raszterizálás

Bán Róbert

`rob.ban@inf.elte.hu`

Eötvös Loránd Tudományegyetem
Informatikai Kar

2025-2026. tavaszi félév

Tartalom

Emlékeztető

Vágás

Vágás 2D-ben

Vágás 3D-ben

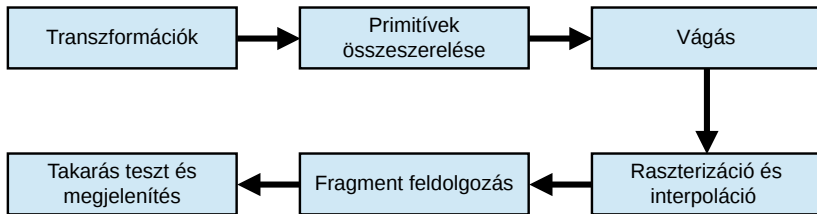
Raszterizálás

Szakasz raszterizálása

Háromszög raszterizálása

Poligon raszterizáció

Grafikus szerelőszalag

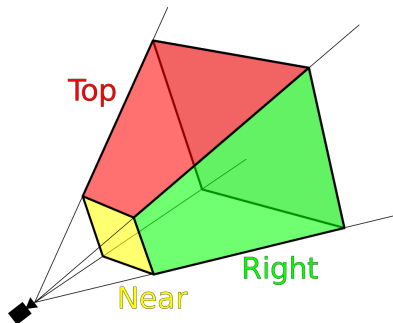


Inkrementális képszintézis

- ▶ Inkrementális elv
- ▶ A transzformációk szemszögéből végignéztük a grafikus szerelőszalagot illetve a láthatósági probléma egy képtér alapú megoldását is megnéztük
- ▶ Most pedig a vágás és raszterizáció témakörével fogunk foglalkozni

Vágás

- ▶ Láttuk, hogy a vágásra két okból is szükségünk lesz:
 - ▶ Degenerált esetek kiszűrése (ld. pl. múlt óra középpontos vetítés)
 - ▶ Ne számoljunk feleslegesen (amit úgyse látunk, ne számoljuk sokat)
- ▶ Vágás során a nézeti csonkagúlán kívül geometriai elemeket szűrjük ki

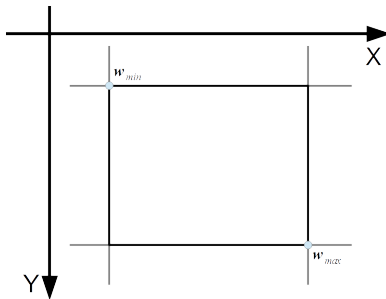


Vágás 2D-ben

- ▶ Egyszerű geometriai elemek (pontok és szakaszok) vágását vizsgáljuk most a síkban
- ▶ Kell egy tartomány *amire* vágunk – az egyszerűtől (tengelyillesztett téglalap) az általános felé (konvex poligon) haladunk
- ▶ Először pontok vágásával foglalkozunk, majd az itteni eredményeket felhasználva próbálunk meg szakaszokat vágni

Pontok vágása tengelyillesztett téglalapra

- ▶ El kell döntenünk egy $\mathbf{x} = [x, y]^T$ pontról, hogy a vágási tartományon belül vagy kívül van-e
- ▶ A legegyszerűbb esetben a vágási tartományunk egy tengelyekkel párhuzamos oldalú téglalap
- ▶ Ezt egy átlójának két végpontjával könnyen reprezentálhatjuk:
 $\mathbf{w}_{min} = [x_{min}, y_{min}]^T$, $\mathbf{w}_{max} = [x_{max}, y_{max}]^T$



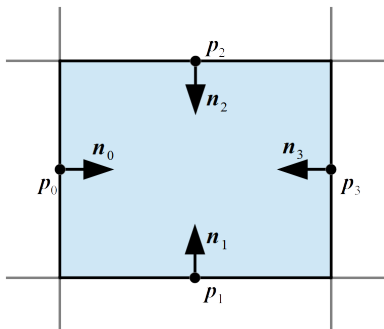
Pontok vágása tengelyillesztett téglalapra

- ▶ Ekkor az x pontosan akkor van a tengelyillesztett téglalapon belül van, ha

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

Pontok vágása konvex négyszögre

- ▶ Legyen most a vágási tartományunk egy konvex négyszög
- ▶ Ez felírható négy félsík metszeteként: a határoló egyeneseinek befelé irányított normálvektorai által meghatározott félterekkel
- ▶ Azaz ekkor a vágási tartományunkat $(\mathbf{p}_i, \mathbf{n}_i)$, $i = 0, \dots, 3$ pont-normális párral tudjuk reprezentálni, ahol most legyen minden normális befelé irányított



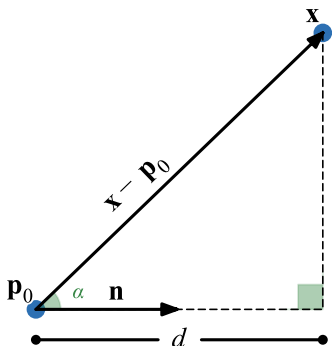
Pontok vágása konvex négyszögre

- ▶ Ekkor $\mathbf{x} = [x, y]^T$ pontosan akkor van a vágási tartományon belül, ha

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

- ▶ Ebben az esetben minden $(\mathbf{p}_i, \mathbf{n}_i)$ által meghatározott félsík tartalmazza $\mathbf{x}$ -et (vagy a határoló egyenesen van, vagy pedig attól normális irányban)
- ▶ Azaz egyszerűen az $\mathbf{x} - \mathbf{p}_i$ vektorok $\mathbf{n}_i$ normálisokra vett előjeles merőleges vetületeinek előjelét kell vizsgálni

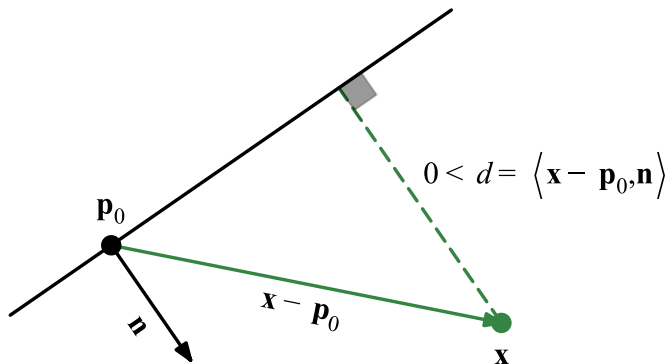
Előjeles merőleges vetület



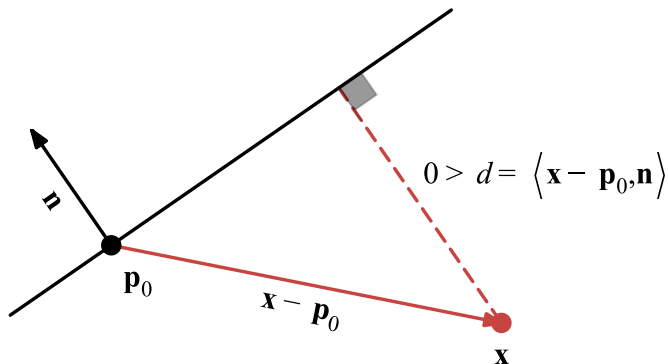
$$\cos \alpha = \frac{d}{|\mathbf{x} - \mathbf{p}_0|}$$

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

Előjeles merőleges vetület – pont a félsíkon belül



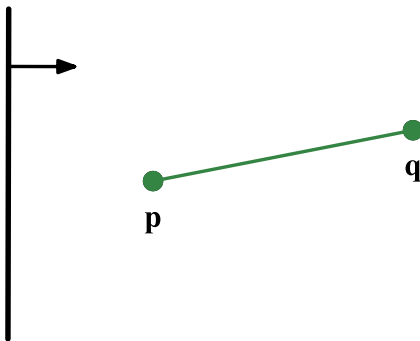
Előjeles merőleges vetület – pont a félsíkon kívül



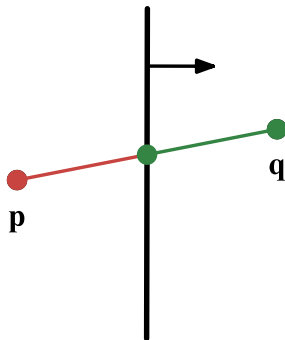
Szakasz vágása félsíkra

- ▶ Vágjuk a szakasz két végpontját a félsíkra
 - ▶ **Ha** mindkettő belül van, **akkor** megtartjuk a szakaszt
 - ▶ **Ha** mindkettő kívül van, **akkor** eldobjuk a szakaszt
 - ▶ **Ha** az egyik kívül a másik pedig belül van, **akkor** megtartjuk a bent lévő végpontot, a kiesőt pedig lecseréljük a szakasz és a félsík határolóegyenesének metszéspontjára

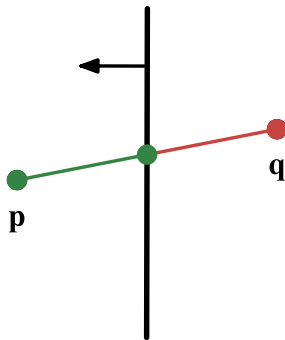
Szakasz vágása félsíkra



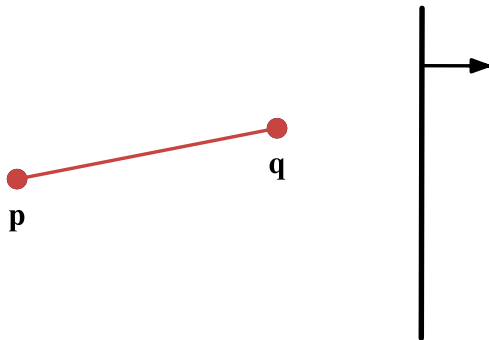
Szakasz vágása félsíkra – új végpont!



Szakasz vágása félsíkra – új végpont!



Szakasz vágása félsíkra



Szakasz vágása konvex négyszögre

Minden egyes határoló félsíkra:

- ▶ Vágjuk a szakasz két végpontját a félsíkra
 - ▶ **Ha** mindkettő belül van, **akkor** nem módosítjuk a végpontot és megyünk tovább
 - ▶ **Ha** mindkettő kívül van, **akkor** eldobjuk a szakaszt és befejeztük
 - ▶ **Ha** az egyik kívül a másik pedig belül van, **akkor** megtartjuk a bent lévő végpontot, a kiesőt pedig lecseréljük a szakasz és a félsík határolóegyenesének metszéspontjára és megyünk tovább

Szakasz vágása tengelyillesztett téglalapra

- ▶ Amikor a harmadik **Ha** teljesül (azaz amikor metszéspontot kell számítani), tengelyillesztett esetben egyszerű a dolgunk
- ▶ Például tekintsük a bal oldali határra történő vágást $\Rightarrow$ ekkor kell egy olyan t , amelyre $\mathbf{x}(t)$ x -koordinátája x_{min}
- ▶ Azaz $x_{min} = x_1 + t(x_2 - x_1)$ -ből $t = \frac{x_{min} - x_1}{x_2 - x_1}$
- ▶ Ekkor az új végpont koordinátái

$$\mathbf{x}\left(\frac{x_{min} - x_1}{x_2 - x_1}\right) = \begin{bmatrix} x_{min} \\ y_1 + \frac{x_{min} - x_1}{x_2 - x_1}(y_2 - y_1) \end{bmatrix}$$

Szakasz vágása konvex négyszögre

- ▶ A korábban látottaknak megfelelően kell a végpontokat vágni az aktuális félsíkra
- ▶ Ha új végpontot kell számolni, akkor egyszerűen be kell helyettesíteni a szakasz (aktuális) parametrikus egyenletét az adott félsíkot határoló egyenes implicit egyenletébe
- ▶ Azaz, meg kell oldani az aktuális $i \in \{0, 1, 2, 3\}$ -re az

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

egyenletet, ahol $t \in [0, 1]$ mellett

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

- ▶ A metszéspont paramétere ekkor $t = \frac{\langle \mathbf{p}_i - \mathbf{x}_1, \mathbf{n}_i \rangle}{\langle \mathbf{x}_2 - \mathbf{x}_1, \mathbf{n}_i \rangle}$

Szakasz vágása konvex sokszögre

- ▶ Ugyanúgy működik, mint konvex négyszögre
- ▶ Az egyetlen különbség, hogy nem négy, hanem több (vagy csak három) határoló egyenes van
- ▶ A normálisok irányítása legyen konzisztens

Szakaszok vágása

- ▶ A fentiek szerint tehát a szakaszt minden egyes félsíkra vágni kell
- ▶ Viszont elég sokféle lehetőség van (négy félsík négyszögeknél, két végpont – bármelyik végpont eshet bárhová)
- ▶ A Cohen-Sutherland algoritmussal rendszerezni tudjuk, hogy mely oldalakra kell vágást végezni

Cohen-Sutherland vágás

- ▶ Tengelyillesztett téglalapokra vágásnál használják (de működik konvex négyszögekre is)
- ▶ Rendeljük hozzá egy-egy 4 bites (úgynevezett TBRL) kódot a két végponthoz:
 - ▶ $T = 1$ ha a pont az ablak felett van, különben 0
 - ▶ $B = 1$ ha a pont az ablak alatt van, különben 0
 - ▶ $R = 1$ ha a pont az ablaktól jobbra van, különben 0
 - ▶ $L = 1$ ha a pont az ablaktól balra van, különben 0

Cohen-Sutherland vágás

code = Top, Bottom, Right, Left

1001	1000	1010
0001	0000	0010
0101	0100	0110

Cohen-Sutherland vágás

- ▶ Tengelyillesztett esetben könnyen számítható a TBRL kód:
 $\text{code} = (y > y_{\max}, y < y_{\min}, x > x_{\max}, x < x_{\min})$
- ▶ Legyen a két végpont TBRL kódja code_a és code_b
 - ▶ **Ha** $\text{code}_a \text{ OR } \text{code}_b == 0$ **akkor** mindkét végpont az ablakon belül van $\Rightarrow$ megtartjuk a szakaszt
 - ▶ **Ha** $\text{code}_a \text{ AND } \text{code}_b \neq 0$ **akkor** mindkét végpont az ablakon kívül van egy közös oldal mentén $\Rightarrow$ eldobjuk a szakaszt
 - ▶ **Különben:** vágnunk kell a szakaszt (a TBRL-beli 1-esek mondják meg, hogy melyik oldalra lehet). Újraszámítjuk a vágott végpont kódját és ismét összehasonlítjuk őket a fentieknek megfelelően.

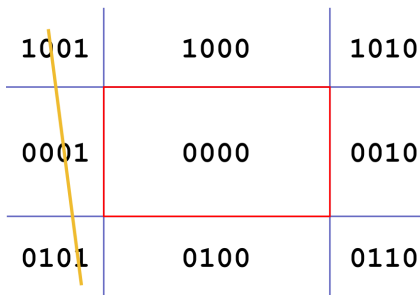
Cohen-Sutherland vágás

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

1001	1000	1010
0001	0000	0010
0101	0100	0110

Cohen-Sutherland vágás

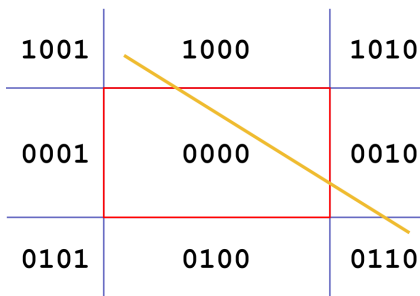
$\text{code}_a \text{ AND } \text{code}_b \neq 0$:



1001	1000	1010
0001	0000	0010
0101	0100	0110

Cohen-Sutherland vágás

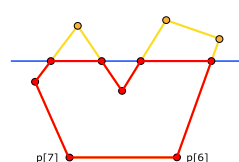
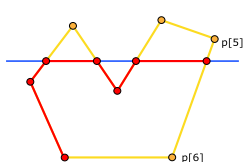
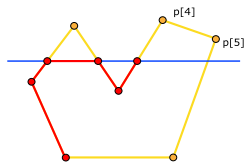
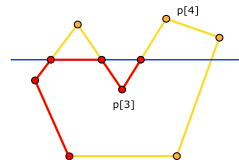
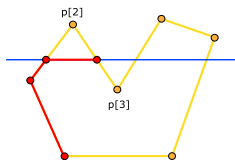
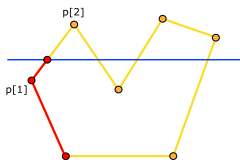
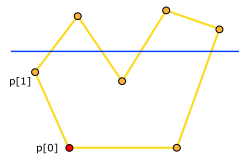
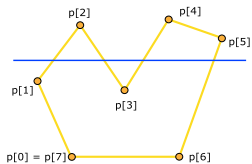
Különben:



Poligon vágása konvex poligonra

- ▶ Egy poligont eltárolhatunk a csúcspontjainak tömbjével, ha a bejárési irányt rögzítjük
- ▶ Legyen a vágási tartományunk egy konvex poligon
- ▶ Nézzük meg, hogy miképp tudunk egy poligont egy konvex poligonra vágni

Sutherland-Hodgman poligon vágás



Sutherland-Hodgman poligon vágás

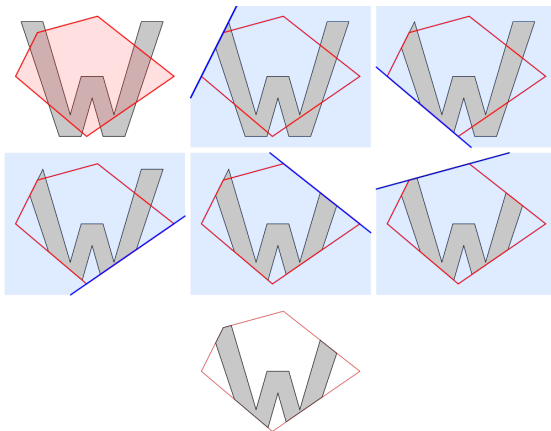
- ▶ Egy oldalra vágás: legyen $\mathbf{p}[]$ a vágandó poligon csúcspontjainak tömbje, a $\mathbf{q}[]$ pedig a kimeneti poligon tömbje (azaz a vágott poligoné)
- ▶ Legyen n a csúcspontok száma, továbbá tegyük fel, hogy $\mathbf{p}[0] = \mathbf{p}[n]$
- ▶ Menjünk végig a vágandó poligon élein:
 - ▶ **Ha $\mathbf{p}[i]$ és $\mathbf{p}[i+1]$ belül van akkor**
 $\Rightarrow$ adjuk hozzá a $\mathbf{p}[i]$ pontot a $\mathbf{q}[]$ kimeneti poligonhoz
 - ▶ **Ha $\mathbf{p}[i]$ belül és $\mathbf{p}[i+1]$ kívül van akkor**
 $\Rightarrow$ adjuk hozzá $\mathbf{p}[i]$ -t a $\mathbf{q}[]$ kimeneti poligonhoz és adjuk hozzá $\mathbf{q}[]$ -hoz a $\mathbf{p}[i], \mathbf{p}[i+1]$ által meghatározott szakasz és az aktuális oldal egyenesének metszéspontját is
 - ▶ **Ha $\mathbf{p}[i]$ kívül és $\mathbf{p}[i+1]$ belül van akkor**
 $\Rightarrow$ adjuk hozzá $\mathbf{q}[]$ -hoz a $\mathbf{p}[i], \mathbf{p}[i+1]$ által meghatározott szakasz és az aktuális oldal egyenesének metszéspontját
 - ▶ **Ha $\mathbf{p}[i]$ és $\mathbf{p}[i+1]$ is kívül van akkor $\Rightarrow$ SKIP**
- ▶ Ezt ismételjük el a vágó poligon minden oldalára (az előző oldal vágásának az eredménye a következő vágás bemenete)

Sutherland-Hodgman poligon vágás

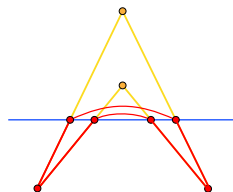
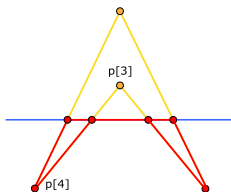
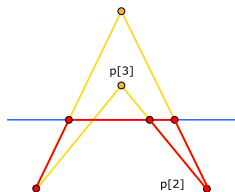
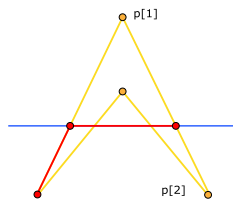
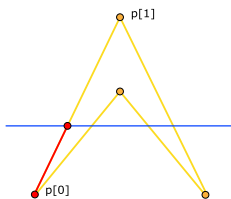
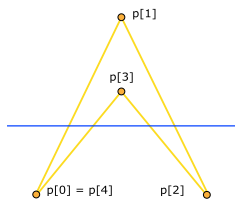
```
PolygonClip(in p[n], out q[m], in line) {  
    m = 0;  
    for( i=0; i < n; i++) {  
        if (IsInside(p[i])) {  
            q[m++] = p[i];  
            if (!IsInside(p[i+1]))  
                q[m++] = Intersect(p[i], p[i+1], line);  
        } else {  
            if (IsInside(p[i+1])) 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

Sutherland-Hodgman poligon vágás

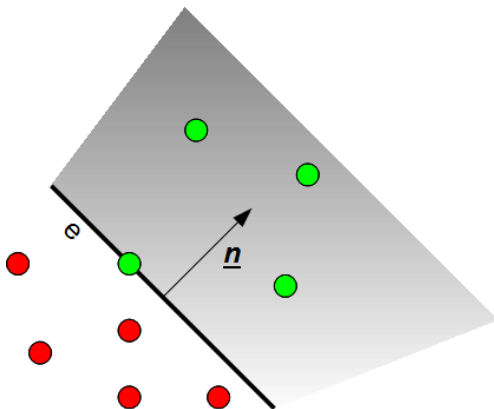
A vágást elvégezzük a vágó poligon minden élére



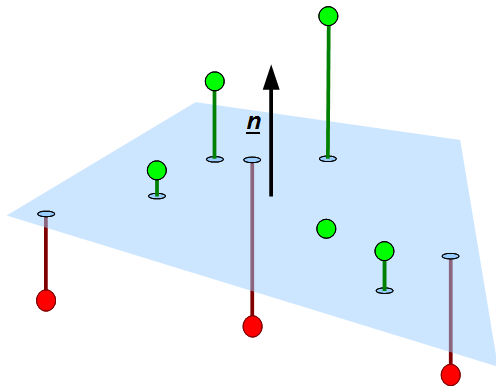
Sutherland-Hodgman probléma: konkáv vágandó poligon



Vágás 2D-ben

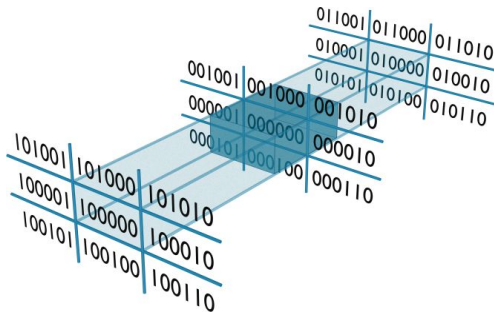


Vágás 3D-ben



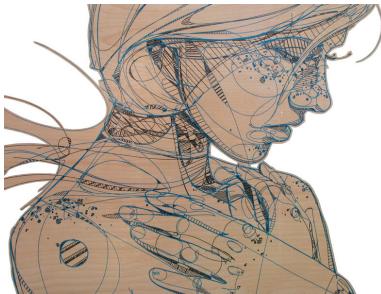
Vágás 3D-ben

- ▶ Lényegében ugyanúgy működik, mint korábban, csak most az $\langle \mathbf{x} - \mathbf{p}, \mathbf{n} \rangle \geq 0$ típusú implicit egyenletek *féltereket* határoznak meg
- ▶ Térben még inkább megéri tengelyillesztett téglatestekre vágni
- ▶ A Cohen-Sutherland bitkódok most már 6 hosszúak: in front, behind, top, bottom, right, left



Szakasz rajzolás

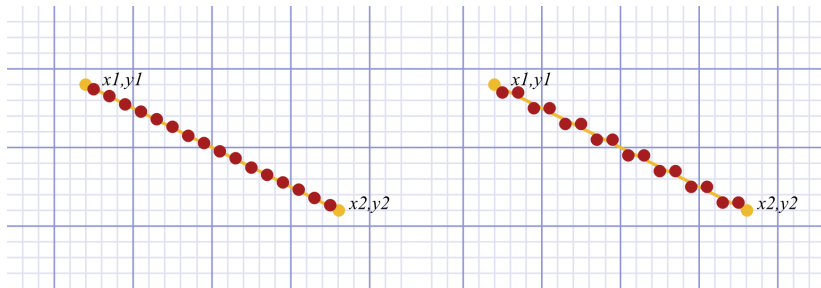
- ▶ Az egyik leggyakrabban használt primitív
- ▶ Fontos, hogy szépen tudjuk rajzolni
- ▶ Még jobb, ha gyorsan is



Jason Thielke, jasonthielke.com

Hogyan rajzolunk szakaszt?

- ▶ Adott a két végpont.
- ▶ Hogyan tudjuk összekötni őket?
- ▶ Csak miniatűr téglalapjaink vannak (amiket pixelnek nevezünk).



Szakasz megadása (sokadszor)

- ▶ Végpontok: $(x_1, y_1), (x_2, y_2)$
- ▶ Tfh. nem függőleges: $x_1 \neq x_2$.
- ▶ Szakasz egyenlete:

$$y = mx + b, x \in [x_1, x_2]$$

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

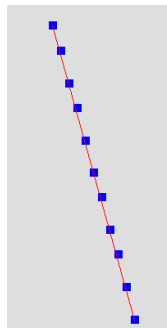
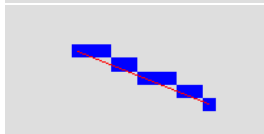
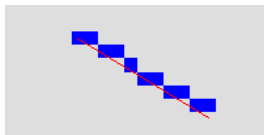
$$b = y_1 - mx_1$$

Naiv algoritmus

```
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
```

Naiv algoritmus

- ▶ kerekítések miatt egy fél pixellel „el van csúszva” a számolás
- ▶ `draw.point((x,y))` $\rightarrow$ int-eket vár, lassú a konverzió
- ▶ $m = \text{float}(y_2 - y_1) / (x_2 - x_1)$ nem pontos
- ▶ $y += m \rightarrow$ a hiba gyűlik y-ban
- ▶ csak $|m| < 1$ -re működik helyesen



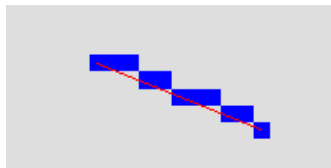
Javítsuk az algoritmust 1.

```
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
```

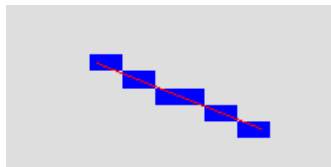
Javítsuk az algoritmust 1.

- ▶ Jó: Mindig „eltalálja” a végpontokat
- ▶ Jó: Egyenletesebben lép az y irányban.
- ▶ Rossz: Még mindig használunk float-okat

Naiv:



Javítás:



Javítsuk az algoritmust 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
```

Javítsuk az algoritmust 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) ←
```

Javítsuk az algoritmust 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) ←
```

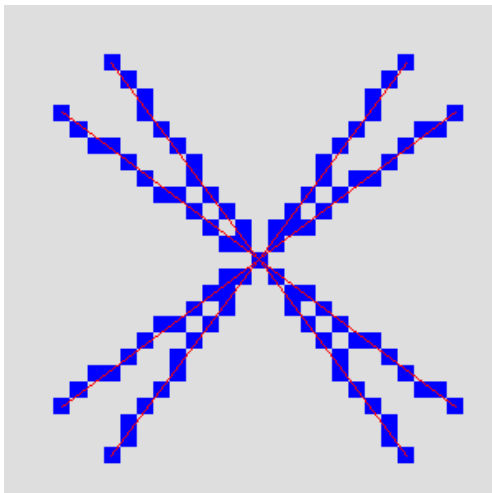
Javítsuk az algoritmust 4.

- ▶ Ez a *Bresenham* algoritmus (egyik speciális esete)
- ▶ Külön gyűjtjük a hibát e -ben
- ▶ Nem használunk `float`-okat
- ▶ Csak $|m| \leq 1$ meredekségű szakaszokra működik, ha $x_1 < x_2$ és $y_1 < y_2$
- ▶ Tetszőleges meredekségre általánosítható

Bresenham algoritmus

- ▶ Nyolcadokra kéne felbontani a síkot, mindegyik külön eset.
- ▶ (Előzők végig: jobbra-le)
- ▶ El kell döntenünk, hogy $|x_2 - x_1|$ vagy $|y_2 - y_1|$ a nagyobb (merre meredekebb a szakasz).
- ▶ Ha $|y_2 - y_1|$ a nagyobb, cseréljük fel $x_i \leftrightarrow y_i$, és rajzolásnál is fordítva használjuk!
- ▶ Ha $x_1 > x_2$, akkor csere: $x_1 \leftrightarrow x_2$, $y_1 \leftrightarrow y_2$.
- ▶ Az e hibatagot $|y_2 - y_1|$ -nal növeljük minden lépésben
- ▶ y -nal $y_2 - y_1$ előjele szerint haladunk.

Bresenham algoritmus



Teljes *Bresenham* algoritmus 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
```

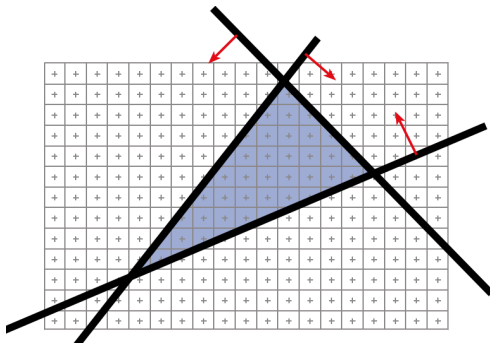
Teljes *Bresenham* algoritmus 2.

```
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)
```

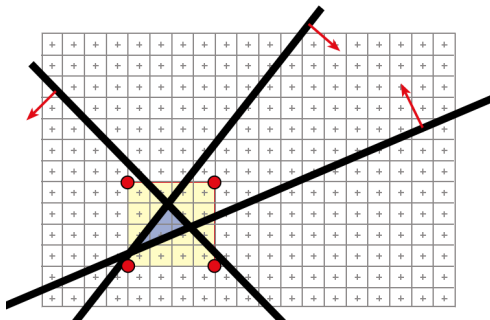
Háromszög raszterizáció

- ▶ A háromszög oldalait tudjuk vágni – most töltsük ki a belsejét!
- ▶ Ha egy meghatározott bejárési irányban adtuk meg az összes háromszög csúcsát, tudunk félsíkokat adni (tudjuk irányítani az éleket) $\rightarrow$ u.i. ha (t_x, t_y) az irányvektora az oldalnak, akkor $(-t_y, t_x)$ egy normális lesz
- ▶ Minden pixelre menjünk végig a képernyőn és nézzük meg, hogy a háromszög oldalai által meghatározott síkok jó oldalán van-e!

Háromszög raszterizáció



Háromszög raszterizáció – okosabban



Háromszög raszterizáció

- ▶ Lehetne még okosabban is csinálni, de: gyakorlatban ez a brute-force megközelítés nagyon jól alkalmazható!

Háromszög raszterizáció

- ▶ Nem egy rögzített értékkel akarunk kitölteni, a hanem a csúcsokban adott értékeket akarjuk interpolálni.
- ▶ Felhasználásai: szín (Gouraud-árnyalás), textúra koordináták (jövő hét...), pozíció, normálvektorok (Phong-árnyalás)
- ▶ Legyen a háromszög egy pontja $\mathbf{p} = \alpha\mathbf{p}_1 + \beta\mathbf{p}_2 + \gamma\mathbf{p}_3$, az α, β, γ baricentrikus koordinátákkal adott.
- ▶ Ekkor bármilyen más mennyiséget is végig tudunk interpolálni ugyanígy:

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

- ▶ Ez az úgy nevezett *Gouraud-interpoláció* vagy *baricentrikus interpoláció*.

Háromszög kitöltés 1.

```
for all x:
  for all y:
     $\alpha, \beta, \gamma = \text{barycentric}(x, y)$ 
    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)
```

Baricentrikus koordináták

- ▶ A baricentrikus koordináták számíthatók a következő képletek segítségével:

$$f_{01}(x, y) = (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0$$

$$f_{12}(x, y) = (y_1 - y_2)x + (x_2 - x_1)y + x_1y_2 - x_2y_1$$

$$f_{20}(x, y) = (y_2 - y_0)x + (x_0 - x_2)y + x_2y_0 - x_0y_2$$

- ▶ Ekkor az x, y ponthoz tartozó baricentrikus koordináták:

$$\alpha = f_{12}(x, y) / f_{12}(x_0, y_0)$$

$$\beta = f_{20}(x, y) / f_{20}(x_1, y_1)$$

$$\gamma = f_{01}(x, y) / f_{01}(x_2, y_2)$$

Háromszög kitöltés 2.

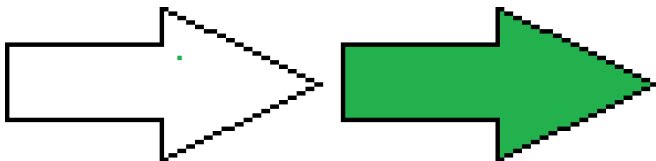
```
x_min = min(floor(x_i))
x_max = max(ceiling(x_i))
y_min = min(floor(y_i))
y_max = max(ceiling(y_i))
for y in [y_min..y_max]:
    for x in [x_min..x_max]:
         $\alpha = f_{12}(x, y) / f_{12}(x_0, y_0)$ 
         $\beta = f_{20}(x, y) / f_{20}(x_1, y_1)$ 
         $\gamma = f_{01}(x, y) / f_{01}(x_2,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

Háromszög kitöltés 2.

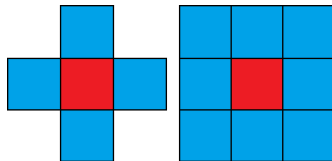
- ▶ Gyorsítás: felesleges minden x, y -t vizsgálni, elég a háromszöget tartalmazó téglalapon végig menni.
- ▶ Inkrementálissá tétel:
 - ▶ Most még lassú, nem használjuk, ki hogy sorban megyünk x -en, y -on.
 - ▶ Milyenek is ezek az f -ek?
 - ▶ Mind $f(x, y) = Ax + By + C$ alakú.
 - ▶ Ekkor $f(x + 1, y) = f(x, y) + A$, ill.
 - ▶ $f(x, y + 1) = f(x, y) + B$
- ▶ Megvalósítás: *házi feladat*

Flood-fill – elárasztásos kitöltés

- ▶ Tetszőleges, már raszterizált körvonalú poligon kitöltésére alkalmas.
- ▶ Bemenet: raszter kép + annak egy pontja
- ▶ Brute-force: a megadott pontból kiindulva rekurzívan haladunk:
 - ▶ Az aktuális pont színe megegyezik a kiindulási pont színével?
 - nem megállunk
 - igen átszínezzük, és
 - ▶ minden szomszédra újratekdjük az algoritmust.



Flood-fill – szomszédságok



- ▶ Négy szomszéd: fent, lent, jobbra, balra
- ▶ Nyolc szomszéd: az előző négy + a sarkak
- ▶ Rekurzió nagyon durva: gyakorlatban ennél okosabb algoritmusok is vannak → aktív éllista stb.