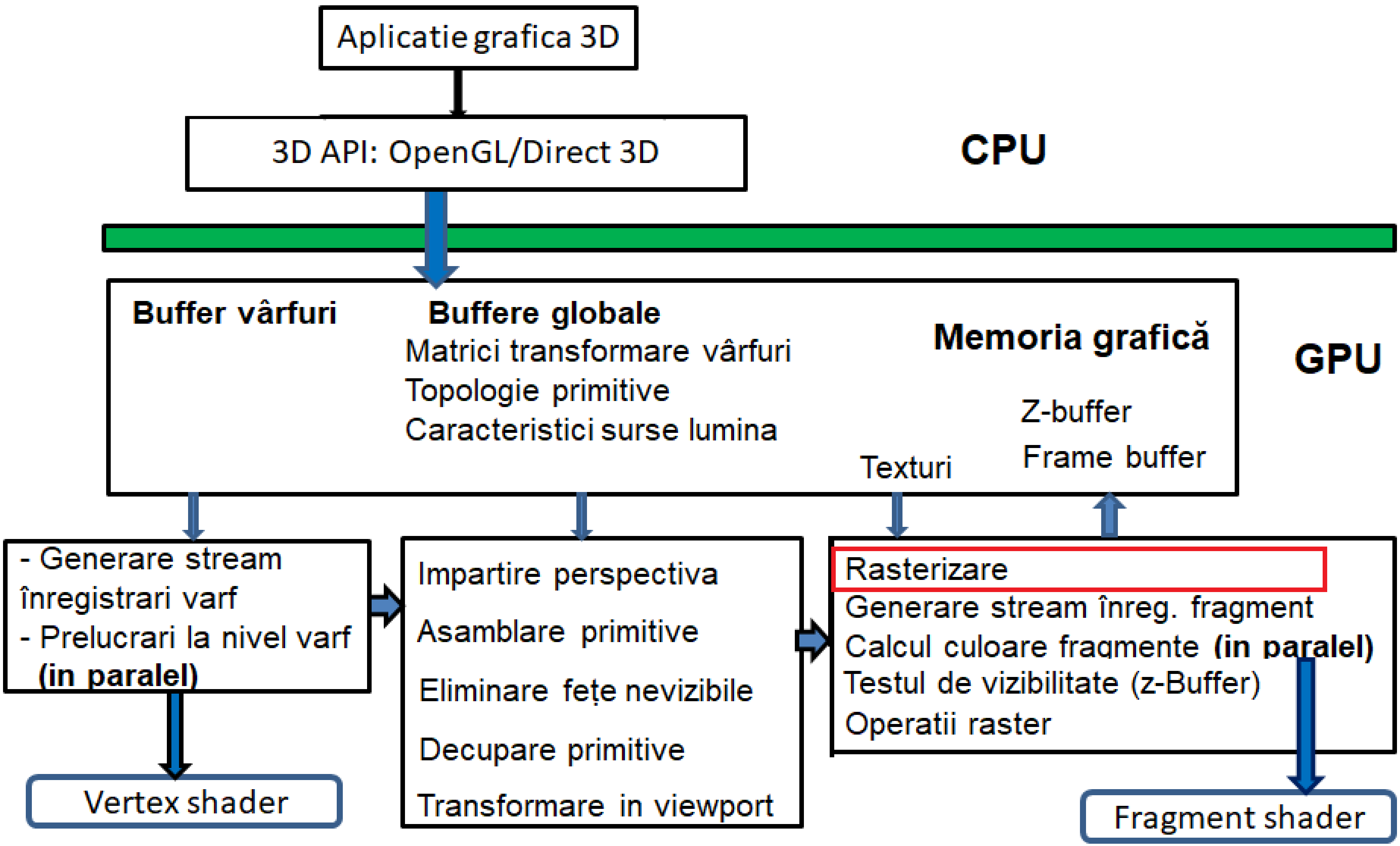
Rasterizarea primitivelor grafice

Prof. univ. dr. ing. Florica Moldoveanu

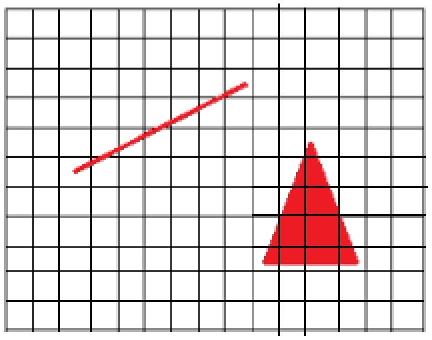
Curs *Elemente de Grafică pe Calculator* – UPB, Automatică şi Calculatoare
2020-2021

Rasterizarea in banda grafica



Rasterizarea (1)

- Primitive grafice acceptate in versiunile noi ale bibliotecii OpenGL(3.0→) sunt:
  - **punctul**, definit prin coordonatele sale (x,y,z);
  - **linia**, definita prin coordonatele extremitatilor sale, (x1,y1,z1) si (x2,y2,z2); specificate in aceasta ordine, **linia este un vector cu originea in (x1,y1,z1) si varful in (x2,y2,z2);**
  - **suprafaţa triunghiulară**, definita prin coordonatele 3D ale varfurilor sale.
- Suprafața de afişare este un spațiu discret, alcatuit din celule de afisare adresate prin coordonate intregi, (0,0) <= (x,y) <= (xmax,ymax).
- Primitivele grafice sunt definite intr-un plan analogic: coordonatele punctelor de pe traseul unui vector sau ale punctelor interioare unei suprafete triunghiulare sunt numere reale.



- **Rasterizarea:**
  - operatia de discretizare a primitivelor grafice: descompunerea lor in fragmente care se afiseaza in pixelii suprafetei de afisare;
  - are ca rezultat o aproximare in spatiul discret a primitivelor definite analitic.

Rasterizarea (2)

Algoritmii de rasterizare executati pe GPU integreaza:

1. Calculul coordonatelor (x,y) ale pixelilor in care se afiseaza fragmentele.
2. Calculul coordonatei z a fiecarui fragment.
3. Interpolarea atributelor asociate varfurilor primitive (iesiri din Vertex shader): culoare, coordonate textura, ş.a. Valorile obtinute prin interpolare la nivel de fragment sunt intrari pentru fragment shader.
4. Calculul culorii fiecarui fragment, în fragment shader.
5. Testul de vizibilitate (adancime) a fiecarui fragment si actualizarea culorii pixelului:

   daca z< Z-buffer[y][x] atunci
   { Z-buffer[y][x] = z
   *actualizeaza culoarea pixelului (x,y) in buffer-ul imagine folosind culoarea fragmentului
   }

Deoarece in algoritmii de rasterizare este important modul de calcul al coordonatelor (x,y), vom abstractiza operatiile din pasii 2, 3, 4 şi 5 în functia **putpixel (int x, int y);**

Rasterizarea vectorilor (1)

La rasterizarea unui vector:

1. **Pixelii (x,y) in care se afiseaza fragmentele de pe traseul vectorului sunt determinati astfel incat sa fie cei mai apropiati de adresele punctelor de pe vectorul analitic.**

2. Coordonata z a fiecarui fragment se obtine prin interpolarea coordonatelor z ale varfurilor vectorului:

- in prezentarea algoritmului z-buffer s-a explicat interpolarea liniara a coordonatelor z;
- rezultate mai realiste (indiferent de tipul proiectiei) se obtin prin interpolarea perspectiva.

3. Culoarea fiecarui fragment se obtine prin interpolarea culorilor varfurilor vectorului (prin calcul incremental) – culoarea interpolata este intrare pentru fragment shader; nu este uzual sa se calculeze culoarea unui fragment de vector in “fragment shader”.

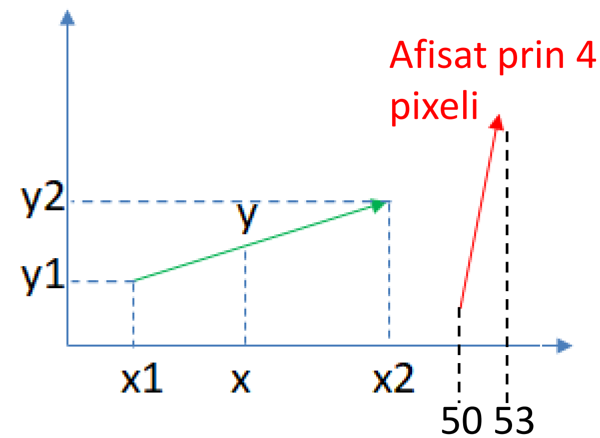
➢ Algoritmii de rasterizare a vectorilor se deosebesc prin pasul 1.

Rasterizarea vectorilor (2)

Ecuatia unui vector: y = m*x + b

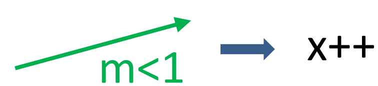
m = (y2-y1)/(x2-x1) şi b=y1 - m*x1

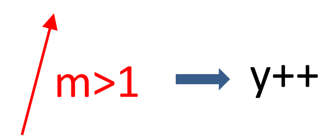
(x1,y1), (x2,y2) sunt adrese de pixeli



Consideram urmatorul algoritm de rasterizare a unui vector:

```
int x, x1,y1,x2,y2; float m, b, y;
putpixel(x1,y1);
for(x=x1+1; x<x2; x++)
  { y=m*x+b ;
    putpixel(x, (int)(y+0.5));//fragmentul se afiseaza in pixelul cu adresa cea mai apropiata de (x,y)
   }
putpixel(x2,y2);
```





Dezavantaje algoritm:

- Nu se tine cont de panta dreptei: vectorii cu panta mare (ex. cel rosu) sunt aproximati prin câţiva pixeli!
- Calculul fiecarei adrese de pixel contine operatii cu numere reale

Algoritmul Digital Differential Analyser (DDA)

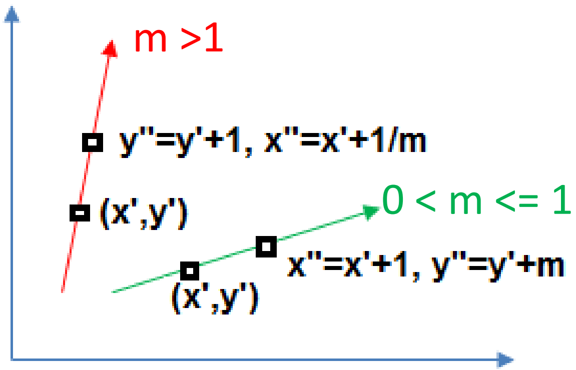
- Ține cont de panta vectorului
- Coordonatele pixelilor de pe traseul vectorului se obtin printr-un calcul incremental (eficient)

Fie (x', y') si (x'',y'') - 2 puncte succesive de pe vector

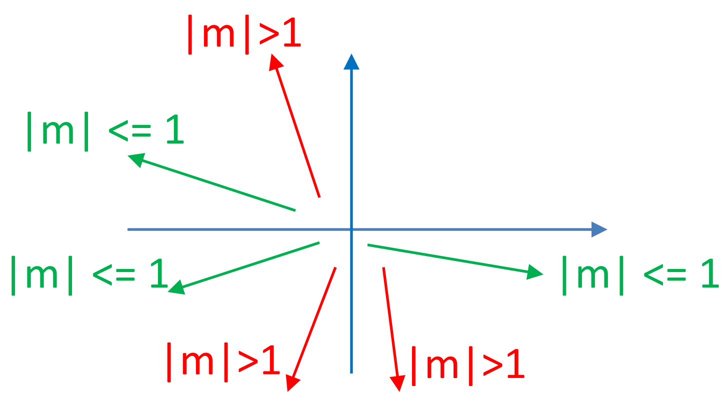
$$(y'' - y') / (x'' - x') = (y2 - y1) / (x2 - x1) = m$$

|m <1| : se incrementeaza x : x'' = x' +1 → y'' = y' + m

| m >1| : se incrementeaza y: y'' = y' + 1 → x'' = x' +1/m




```
void DDA(int x1, int y1, int x2, int y2)
{double m, absm, rx, ry; int x, y;
//vectorul se genereaza de la (x1,y1) la (x2,y2)
 if(x1==x2) //vector vertical
   { if(y1>y2) {y=y1; y1=y2; y2=y;}
     for(y=y1; y<=y2; y++)  putpixel(x1,y);
    return;
 }
```



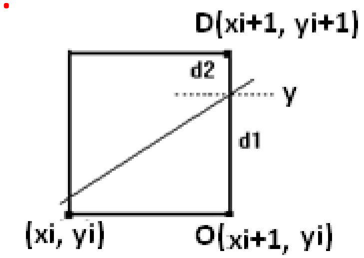
Algoritmul DDA(2)

```
if(y1==y2) //vector orizontal
  {if(x1>x2)
    {x=x1; x1=x2; x2=x;}
   for(x=x1; x<= x2; x++)  putpixel(x,y1);
   return;
  }
 m=(double)(y2-y1)/(x2-x1); absm=abs(m);
 if(absm<=1 && x1>x2  ||  absm>1 &&  y1>y2)
     {x=x1; x1=x2; x2=x; y=y1; y1=y2; y2=y;}
 putpixel(x1, y1);
 if( absm<=1)
     for(x=x1+1, ry=y1; x<x2; x++)
       {ry+=m;  putpixel(x, (int)(ry+0.5)); } // y = y + m
else
     {m=1/m;
      for(y=y1+1, rx=x1; y<y2; y++)
       {rx+=m;  putpixel((int)(rx+0.5), y); } // x = x +1/m
      }
 putpixel(x2, y2);
}
```

Dezavantaj: calcule cu numere reale

Algoritmul Bresenham (1)

- Contine numai operatii cu numere intregi.
- Coordonatele pixelilor de pe traseul vectorului se obtin prin calcul incremental.
- **Este definit pentru vectori din primul octant: vectori cu panta 0<m<=1**
- Pentru fiecare valoare a lui x, de la xmin_vector la xmax_vector, se alege acel punct al spaţiului discret care este mai apropiat de punctul de pe vectorul teoretic.



- Fie m=(y2-y1)/(x2-x1) panta vectorului,
- (x_i, y_i) ultimul punct de pe traseul vectorului în spaţiul discret
- d1 distanţa de la punctul de pe vectorul teoretic, (x_i+1, y), la punctul $O(x_i+1,y_i)$
- d2 distanţa de la punctul de pe vectorul teoretic la punctul $D(x_i+1,y_i+1)$.
- O si D sunt adrese de pixeli (puncte ale spatiului discret)

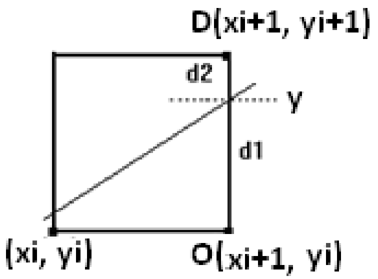
➢ Următorul punct al spatiului discret ales pentru aproximarea vectorului va fi:

  - O dacă d1<d2, D în caz contrar.
  - Dacă d1=d2 se poate alege oricare dintre cele două puncte.

Algoritmul Bresenham(2)

- Exprimăm diferenţa d1-d2:

  $y=m*(x_i+1)+b$ este ordonata punctului de pe vectorul teoretic

  $d1= y - y_i= m*(x_i+1)+b-y_i$

  $d2= y_i + 1-y= y_i+1-m*(x_i+1)-b$

  $d1- d2= 2*m*(x_i+1)-2*y_i+2*b -1$

- Se înlocuieşte m cu dy/dx apoi se înmulţeşte în ambele părţi cu dx. Rezultă:

  $t_i= (d1-d2)*dx= 2*dy*(x_i+1)-2*dx*y_i+2*b*dx-dx = 2*dy*x_i - 2*dx*y_i + 2*b*dx - dx + 2*dy$

- t_i reprezintă eroarea de aproximare în pasul i: pe baza ei se alege urmatorul punct al spatiului discret

  Notăm cu (x_{i+1},y_{i+1}) punctul care se va alege în pasul i.

- Expresia erorii de aproximare în pasul i+1 este:

  $t_{i+1} = 2*dy*x_{i+1} - 2*dx*y_{i+1} + 2*b*dx - dx +2*dy$

Algoritmul Bresenham(3)

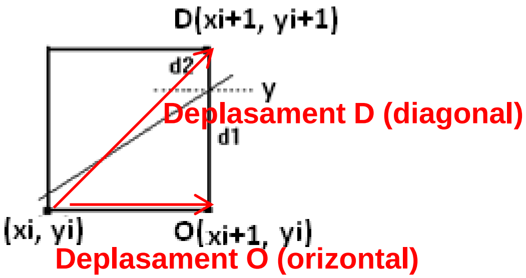
t_i=(d1-d2)*dx = 2*dy*x_i - 2*dx*y_i + 2*b*dx-dx + 2*dy, (dx>0)

(1) Dacă t_i <= 0 (d1<=d2), se alege punctul O, deci $x_{i+1} = x_i+1$ şi $y_{i+1} = y_i$

Rezultă:

t_{i+1} = 2*dy*(x_i+1) - 2*dx*y_i + 2*b*dx - dx +2*dy

sau **$t_{i+1} = t_i + 2*dy = t_i + c1$**



(2) Dacă t_i > 0, se alege punctul D, deci $x_{i+1} = x_i + 1$ şi $y_{i+1} = y_i +1$

Rezultă:

t_{i+1} = 2*dy*(x_i+1) - 2*dx*(y_i+1) + 2*b*dx - dx +2*dy

sau **$t_{i+1} = t_i + 2*dy - 2*dx = t_i + c2$**

- Valoarea variabilei de test se obtine prin calcul incremental: adunarea unei constante intregi

Eroarea de aproximare pentru primul pas. Se inlocuiesc in t_i : x_i = x1 şi y_i = y1:

t_1 = 2*dy*x1 - 2*dx*y1 + 2*dy – dx + 2*dx(y1 - (dy/dx)*x1)

sau **t_1 = 2*dy-dx**

Algoritmul Bresenham(4)

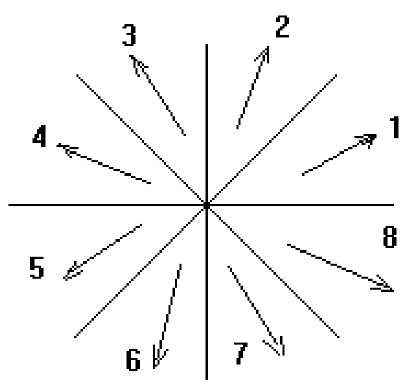
Implementare in C:

```
void Bres_vect(int x1, int y1, int x2, int y2)
{ //pentru vectori cu panta cuprinsă între 0 şi 1
  int dx, c1, c2, x, y, t;
  dx=x2-x1;
  c1=(y2-y1)<<1;  // 2*dy
  c2=c1-(dx<<1);  // 2*dy – 2*dx
  t=c1-dx;  // 2*dy - dx
  putpixel(x1, y1);
  for(x=x1+1, y=y1; x<x2; x++)
   { if(t<0) t+= c1; //deplasament  O
      else  { t+= c2; y++;} // deplasament D
    putpixel(x,y);
   }
  putpixel(x2,y2);
}
```

Algoritmul Bresenham- generalizare(1)

Generalizarea algoritmului Bresenham pentru vectori de orice panta

- Vectorii definiţi în planul XOY pot fi clasificaţi, pe baza pantei, în **opt clase geometrice**, numite **“octanţi“**
- Un vector care apartine unui octant O are 7 vectori simetrici in ceilalti 7 octanti



dx=x2 – x1 şi dy=y2 – y1

octantul 1: dx>0 şi dy>0 şi dx>=dy;

octantul 2: dx>0 şi dy>0 şi dx<dy;

octantul 3: dx<0 şi dy>0 şi abs(dx)<dy;

octantul 4: dx<0 şi dy>0 şi abs(dx)>=dy;

octantul 5: dx<0 şi dy<0 şi abs(dx)>=abs(dy);

octantul 6: dx<0 şi dy<0 şi abs(dx)<abs(dy);

octantul 7: dx>0 şi dy<0 şi dx<abs(dy);

octantul 8: dx>0 şi dy<0 şi dx>=abs(dy);

Algoritmul Bresenham –generalizare(2)

Intr-un pas al algoritmului generalizat sunt posibile urmatoarele deplasamente:

+h, deplasamentul orizontal spre dreapta (în sensul crescător al axei x): x++

-h, deplasamentul orizontal spre stânga (în sensul descrescător al axei x): x--

+v, deplasamentul vertical în sus (în sensul crescător al axei y): y++

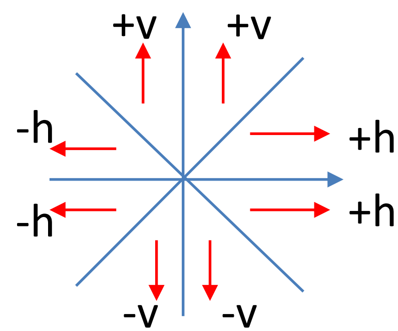
-v, deplasamentul vertical în jos (în sensul descrescător al axei y): y--

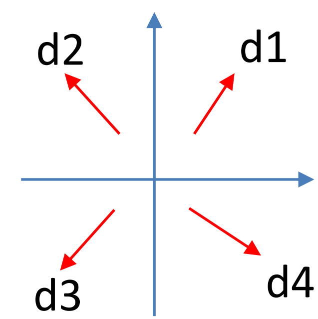
d1, deplasamentul diagonal dreapta-sus: x++; y++

d2, deplasamentul diagonal stânga-sus: x--; y++

d3, deplasamentul diagonal stânga-jos: x--; y--

d4, deplasamentul diagonal dreapta-jos: x++; y--



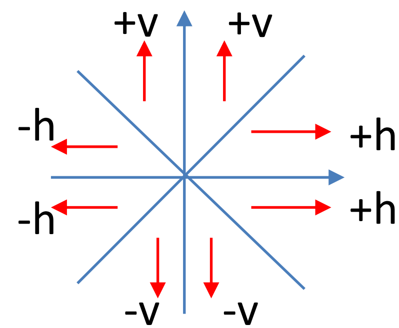


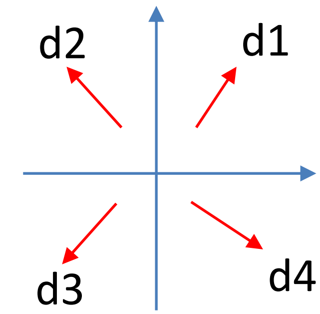
Octant	1	2	3	4	5	6	7	8
Deplas O	+h	+v	+v	-h	-h	-v	-v	+h
Deplas D	d1	d1	d2	d2	d3	d3	d4	d4

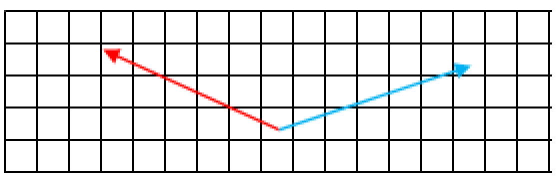
Corespondenţa între deplasamentul ales într-un pas al algoritmului Bresenham (punctul O sau punctul D) şi deplasamentele echivalente pentru vectorii simetrici din ceilalti octanti:

Algoritmul Bresenham –generalizare(2)

Octant	1	2	3	4	5	6	7	8
Deplas O	+h	+v	+v	-h	-h	-v	-v	+h
Deplas D	d1	d1	d2	d2	d3	d3	d4	d4







Vectorul care trebuie generat:

(x1 =10, y1=3) – (x2=5, y2 = 5)

dx = -5; dy =2; abs(dx)>dy →Vectorul face parte din octantul 4

Vectorul simetric din octantul 1

(x1=10, y1=3) – (x2=15, y2=5)

dx=5; dy =2

In fiecare pas al algoritmului Bresenham generalizat:

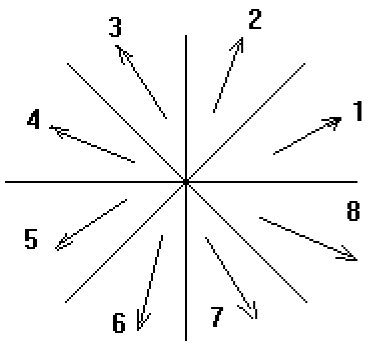
- Se determina tipul deplasamentului pentru urmatorul pixel al vectorului simetric din oct. 1
- Se calculeaza coordonatele punctului de pe vectorul care trebuie generat:
    - Pentru deplasament O: x--
    - Pentru deplasament D: x--; y++

Algoritmul Bresenham – generalizare(3)

Algoritmul Bresenham generalizat - implementare in C

```
void Bres_general(int x1, int y1, int x2, int y2)
{ int x, y, i, oct, dx, dy, absdx, absdy, c1, c2, t;
  if(x1==x2) //vertical
  { if(y1>y2){y=y1; y1=y2; y2=y;}
     for(y=y1; y<=y2;y++)
       putpixel(x1,y);
     return;
  }
 if(y1==y2) //orizontal
   {if(x1>x2) {x=x1; x1=x2; x2=x;}
     for(x=x1; x<= x2; x++)
       putpixel(x,y1);
    return;
   }
```

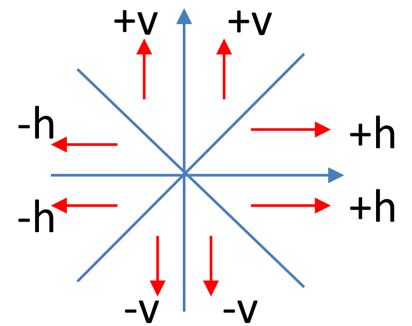
Algoritmul Bresenham – generalizare (4)

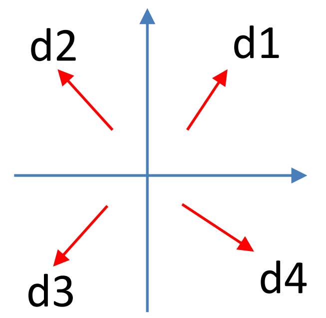


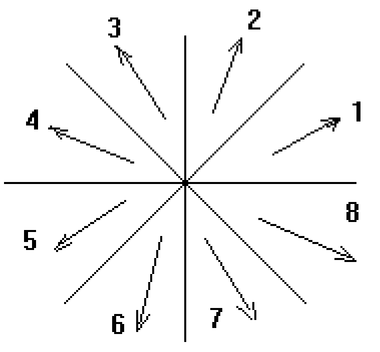

```
dx= x2-x1; dy= y2-y1; absdx= abs(dx); absdy= abs(dy);
if(dx>0)//oct=1,2,7,8
   { if(dy>0)// oct=1,2
      if(dx>=dy) oct=1; else oct=2;
     else
      if(dx>=absdy) oct=8; else oct=7;
   }
else//3,4,5,6
   { if(dy>0)// oct=3,4
      if(absdx>=dy) oct=4; else oct=3;
     else
      if(absdx>=absdy) oct=5; else oct=6;
   }
// Numărul de paşi la execuţia algoritmului generalizat este maxim(abs(dx), abs(dy))
// Dacă abs(dy) > abs(dx), se inversează rolul variabilelor absdx şi absdy în calculul constantelor c1 şi c2
if(absdy>absdx) //  adresele de pe traseul vectorului se obtin prin incrementarea lui y
    {x=absdx; absdx=absdy;absdy=x;}
  c1= absdy<<1;  c2= c1-(absdx<<1); t= c1-absdx;
```

Algoritmul Bresenham – generalizare (5)

```
putpixel(x1,y1);
 for(i=1, x=x1, y=y1; i<absdx; i++)
 { if(t<0) // deplasament O
    {t+=c1;
     switch(oct)
     {case 1:  case 8: x++; break; // +h
      case 4:  case 5: x--;break; // -h
      case 2:  case 3: y++;break; //+v
      case 6:  case 7: y--;break; // -v
     }
    }
   else
    {t+=c2 // deplasament D
     switch(oct)
     {case 1: case 2: x++; y++; break; // d1
      case 3: case 4: x--; y++; break; // d2
      case 5: case 6: x--; y--; break; // d3
      case 7: case 8: x++; y--; break; // d4
     }
    }
   putpixel(x,y);
 } putpixel(x2,y2);
}
```







Generarea vectorilor de orice panta

Exercitiu

Fie vectorul (x1=5, y1 =5) → (x2=1, y2 =15)

1. Din care octant face parte vectorul?
2. Care este numarul de pixeli prin care este aproximat vectorul?
3. Care este vectorul simetric cu el din octantul 1?

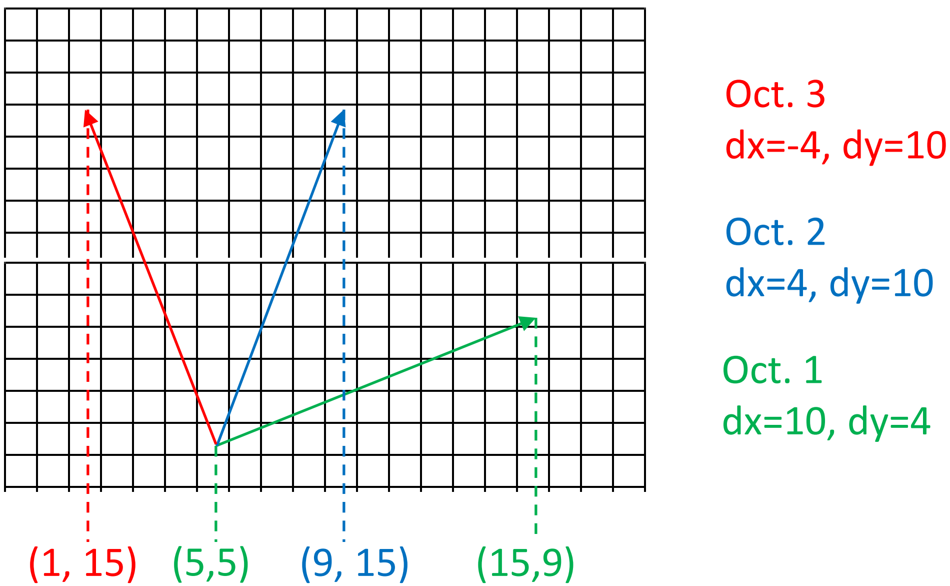
Raspunsuri:

1. Octantul 3:

   dx<0, dy>0, abs(dx)<dy

2. 11 pixeli: se genereaza prin incrementarea lui y

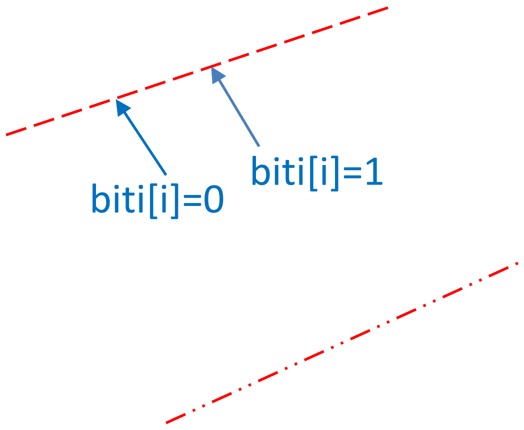
3. (x1=5, y1 =5) → (x2=15, y2 =9)



Generarea liniilor intrerupte

```
void Bres_gen(int x1, int y1, int x2, int y2, int şablon)
{  // şablon: defineste tipu de linie intrerupta - intreg pe 16 biti cu valori 0 si 1
  // se suprapune şablonul  pe traseul vectorului: se afiseaza numai acei pixeli ai vectorului pentru care
     // bitul sablonului = 1


   int val, bit, biţi[16] ;
  // se extrag biții şablonului în vectorul biţi
   for(int i=0, val=1; i<16; i++)
   { if (şablon & val) biţi[i] = 1; else biţi[i] = 0;
     val = val << 1;
   }
   ..................................
  if(biţi[0]]) putpixel(x1,y1);
  for(i=1, x=x1, y=y1, bit=1; i<absdx; i++)
 { .......................................
    if(biţi[(bit++) % 16]) putpixel(x,y); // se repeta sablonul din 16 in 16 pixeli
 }
 if(biţi[bit % 16) putpixel(x2,y2);
}
```

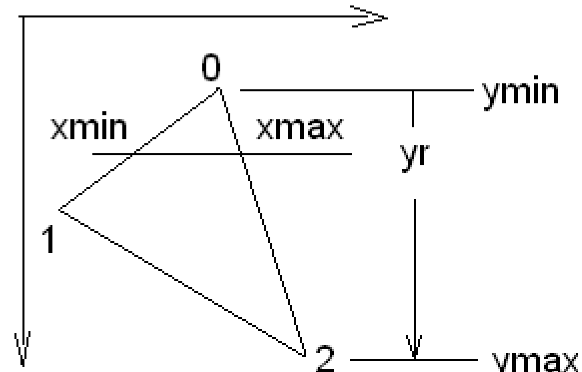


Rasterizarea suprafetelor triunghiulare

La rasterizarea unei suprafete triunghiulare

1. Pentru determinarea fragmentelor interioare suprafetei se intersecteaza suprafata triunghiului cu liniile raster (imagine) de la ymin_triunghi la ymax_triunghi;
2. Fiecare segment dintr-o linie raster, care este interior suprafetei triunghiulare, este descompus in fragmente.
3. Coordonatele z ale fragmentelor se obtin prin calcul incremental (vezi curs “algoritmul z-buffer”).
4. Calculul culorii fiecarui fragment este efectuat in programul “fragment shader”.
5. Sunt afisate fragmentele care trec testul de vizibilitate.

Rasterizarea suprafetelor triunghiulare - algoritmul -



Bool SupTri (int x0, int y0, int x1, int y1, int x2, int y2)

{ 1. Se sorteaza varfurile triunghiului a.î. varful cu
   coordonata y minima sa fie (x0,y0);

2. Daca triunghiul este degenerat (varfurile sunt coliniare), return false;

3. Se orienteaza conturul in sens trigonometric (v0-v1-v2: sens trigonometric);

4. pentru (yr = ymin-triunghi; yr<= ymax-triunghi; yr++)

   - se calculeaza segmentul din linia y=yr, care este interior suprafetei triunghiului

   - se memoreaza extremitatile sale in 2 tablouri, XMIN, XMAX:

   XMIN[yr] = extr_stanga; XMAX[yr] = extr_dreapta;

5. Se afiseaza pixelii segmentelor interioare (de la ymin_triunghi la ymax_triunghi);

   return true;

}

Pasii 1 si 2

1. Se sorteaza varfurile triunghiului a.i. varful 0 sa aiba cordonata y minima

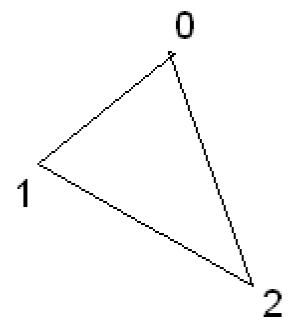
```
if( y1 < y0) { t= y1; y1 = y0; y0 =t; t= x1; x1 = x0; x0 =t;} //y0<y1

if( y2 < y0) { t= y2; y2 = y0; y0 =t; t= x2; x2 = x0; x0 =t;} //y0<y2
```

2. Daca triunghiul este degenerat (varfurile sunt coliniare), return false

$$\det = \begin{vmatrix} x1 - x0 & y1 - y0 \\ x2 - x0 & y2 - y0 \end{vmatrix}$$

$$= (x1 - x0)(y2 - y0) - (x2 - x0)(y1 - y0)$$

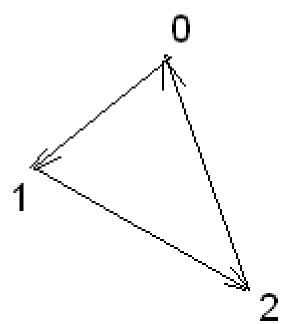


if(det==0) → varfurile sunt coliniare

Pasul 3: orientarea conturului

3. Se orienteaza conturul in sens trigonometric (v0-v1-v2: sens trigonometric)

$dx0 = x1 - x0$; $dx1 = x2 - x0$; $dy0 = y1 - y0$; $dy1 = y2 - y0$;

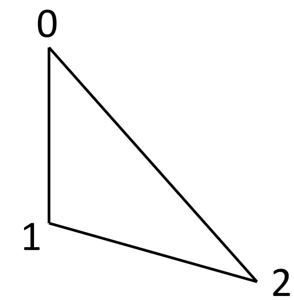


Pentru un contur orientat trigonometric:

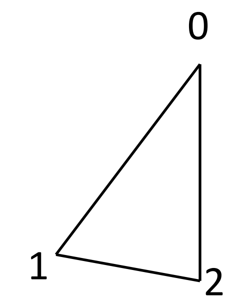
$x1 < x0$, $x2 > x0$

$dx0 < 0$, $dx1 > 0$, $dy0 > 0$, $dy1 > 0$

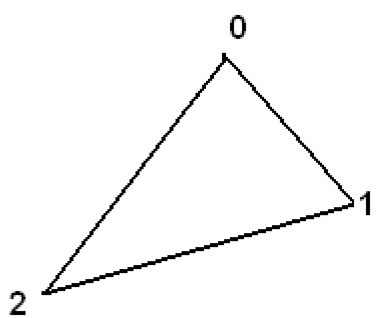
det = $dx0*dy1 - dx1*dy0$ **< 0**



$x1=x0$, $dx0=0$, $dx1>0$
det = $- dx1*dy0 < 0$



$x2=x0$, $dx1=0$, $dx0<0$
det = $dx0*dy1 < 0$



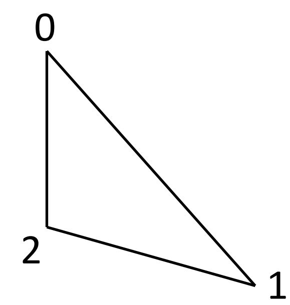
Pentru un contur orientat anti-trigonometric:

$x1 > x0$, $x2 < x0$

$dx0 > 0$, $dx1 < 0$, $dy0 > 0$, $dy1 > 0$

det = $dx0*dy1 - dx1*dy0$ **> 0**

Se inverseaza varful 1 cu varful 2



$x2=x0$, $dx1=0$, $dx0>0$
det = $dx0*dy1 > 0$

Pasul 4: calculul extremitatilor segmentelor interioare(1)

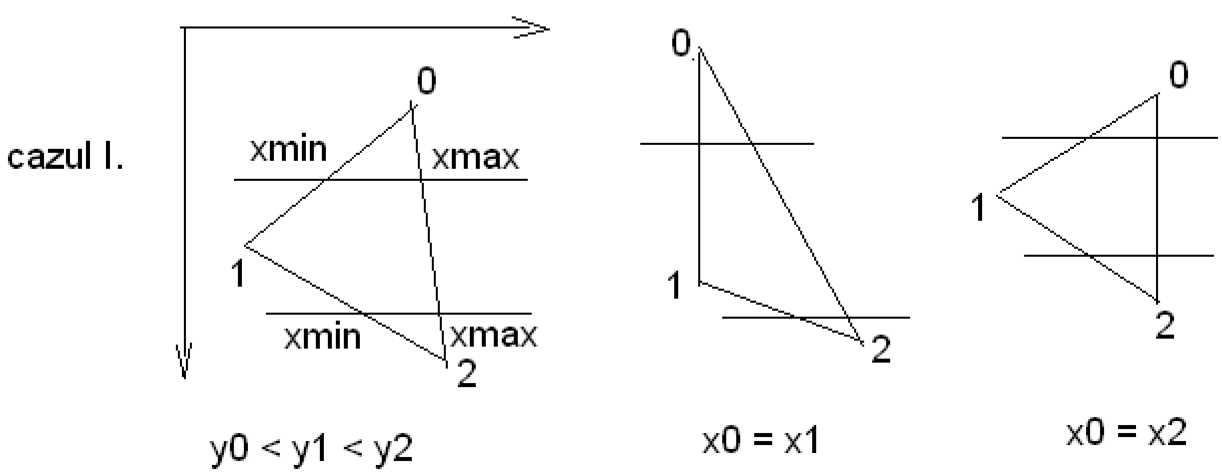
4. Se calculeaza extremitatile segmentelor interioare suprafetei triunghiului

- Coordonatele xmin, xmax ale extremitatilor segmentelor interioare suprafetei triunghiului se memoreaza in 2 tablouri:

  int XMIN[H], XMAX[H]; // H este numarul de linii imagine

- Calculul extremitatilor este efectuat in functiile **CalculXmin()** si **CalculXmax()**;

Cazuri:



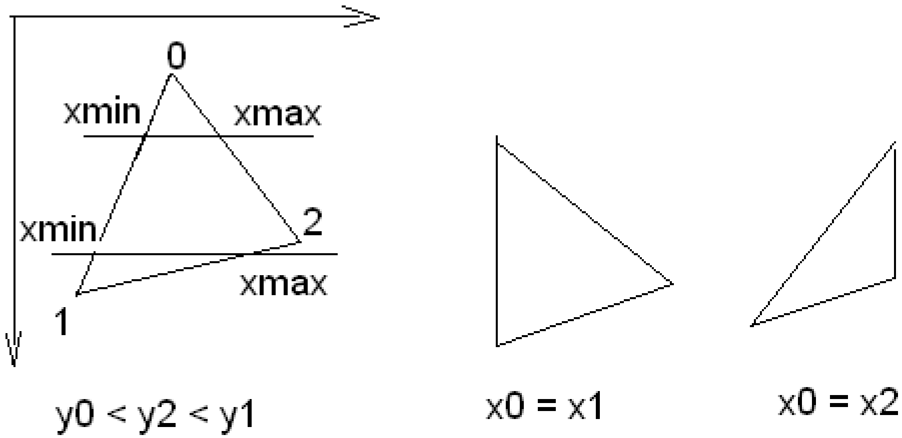
CalculXmin(x0, y0, x1, y1);

CalculXmin(x1, y1, x2, y2);

CalculXmax(x0, y0, x2, y2);

Pasul 4: calculul extremitatilor segmentelor interioare (2)

cazul II.

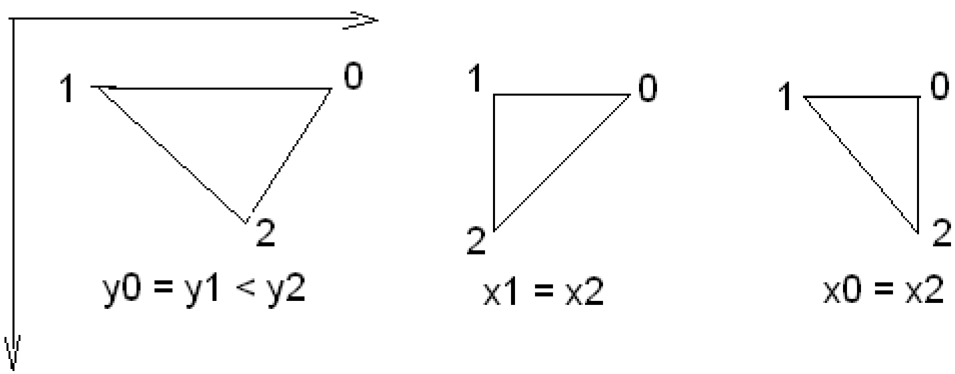


CalculXmin(x0, y0, x1, y1);

CalculXmax(x0, y0, x2, y2);

CalculXmax(x2, y2, x1, y1);

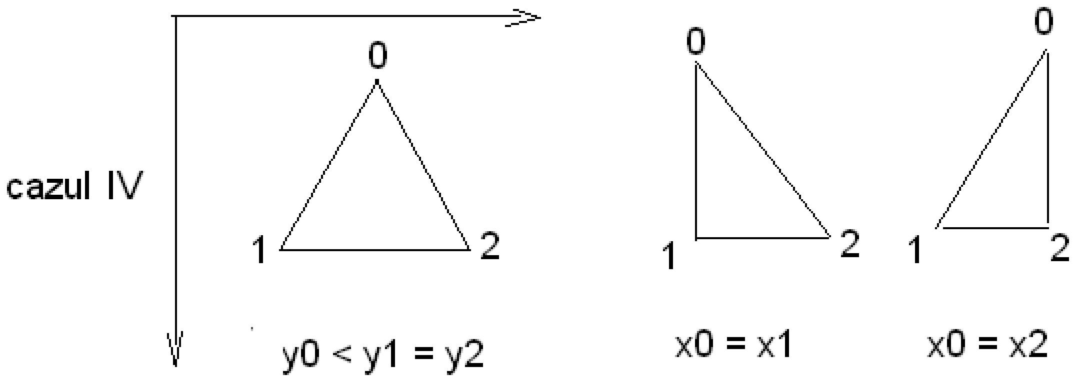
cazul III.



CalculXmin(x1, y1, x2, y2);

CalculXmax(x0, y0, x2, y2);

Pasul 4: calculul extremitatilor segmentelor interioare (3)



CalculXmin(x0, y0, x1, y1);

CalculXmax(x0, y0, x2, y2);

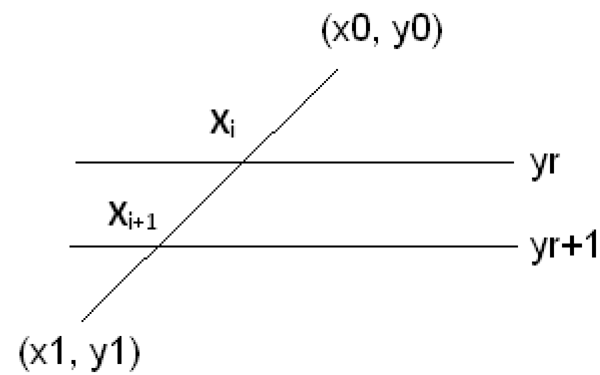
Calculul coordonatelor xmin, xmax

m = (y1 - y0)/ (x1 – x0) - panta laturii

(yr +1 – yr)/ (x_{i+1} – x_i) = m → $x_{i+1} = x_i + (1/m)$

dx = 1.0/m - constanta a laturii (x0,y0) - (x1,y1)

$x_{i+1} = x_i + dx$ → calcul incremental



Pasul 4: calculul extremitatilor segmentelor interioare (4)

```
int  XMIN[H], XMAX[H];
void Calcul Xmin(int x0, int y0, int x1, int y1)
{ // y0 < y1
 // calculeaza toate intersectiile dintre
//latura (x0,y0) – (x1,y1) si liniile raster
  int y;
  if( x0==x1)// latura verticala
   {for(y =y0; y<=y1;y++)
               XMIN[y] = x0;
    return;
   }
float dx = (float)(x1 – x0) / (y1 – y0);
XMIN[y0] = x0; XMIN[y1] = x1;
 for(y = y0 +1; y<y1; y++)
      XMIN[y] = XMIN[y-1] + dx;
}
```

```
void CalculXmax(int x0, int y0, int x1, int y1)
{ - similara cu functia CalculXmin
  - memoreaza punctele succesive de pe latură
    in tabloul XMAX
}
```

Pasul 5: afisarea pixelilor suprafetei triunghiulare

5. Se afiseaza pixelii interiori triunghiului, intre xmin si xmax pe fiecare linie raster

```
ymax = y1; if (y2>y1 )ymax = y2;

for(int y= y0; y<= ymax; y++) // pt fiecare linie raster care intersecteaza suprafata triunghiului

    for( int x = XMIN[y]; x <= XMAX[y]; x++)

        putpixel(x, y); // afisare fragment interior suprafetei triunghiului
```