2

Linii şi Suprafeţe Ascunse: Metodele Buffer-ului de Adâncime şi a Subdivizării Ecranului

Acest capitol este al doilea din cele trei care prezintă algoritmi pentru îndepărtarea liniilor şi suprafeţelor ascunse. Acest capitol prezintă două metode cu aplicabilitate mai generala şi care pot trata cazuri pe care capitolul precedent nu le putea.

1 Introducere prezintă diferenţele faţa de metodele cunoscute până acum şi ce limitări se impun.

2 Metoda Subdivizării Ecranului (Algoritmul Warnock) este o tehnică *"divide and conquer"* care colorează zone dreptunghiulare ale ecranului atunci când s-a ajuns la concluzia că întreaga zonă va fi colorată identic. Metoda subdivide ecranul în mod sistematic, până când condiţia este îndeplinită; uneori această zonă se reduce la un pixel.

3 Metoda Buffer-ului de Adâncime este simplă din punct de vedere conceptual, dar consumă mult timp de calcul şi/sau memorie. Această metodă poate să trateze orice scenă, oricât de complicată ar fi.

1 Introducere

În capitolul precedent am văzut în ce constă îndepărtarea suprafeţelor ascunse şi câteva metode cu limitările lor. În acest capitol se prezintă două metode generale şi puternice, metoda subdivizării ecranului şi metoda buffer-ului de adâncime. Acestea sunt valabile doar pentru display-uri raster şi deci nu pot fi utilizate pe dispozitive vectoriale cum ar fi plotter-ele cu peniţă. Ambele metode se bazează pe faptul că o imagine poate fi descompusă în părţile sale atomice, pixelii. La acest nivel, singura primitivă grafică necesară este `SetPix`. Nu este necesar să calculăm intersecţii de linii sau poligoane, ca la metoda sortării în adâncime. Totul se poate reduce la testul dacă un pixel dat este sau nu în interiorul unui anumit poligon din scenă. Dacă da, trebuie determinată distanţa de la acest punct al

poligonului până la ecran sau la planul de proiecţie. În final, un singur pixel este setat pe culoarea potrivită.

După cum se va vedea, ambele metode necesită ca obiectele să fie mărginite de poligoane plane. (Metoda buffer-ului de adâncime poate fi utilizată pentru afişarea oricărei scene, chiar şi cu obiecte mărginite de suprafeţe curbe. În asemenea cazuri, metoda este adăugată ca postprocesor la metoda de generare.) S-a impus această restricţie pentru a ramâne în domeniul cursului.

Trasarea prin calculul şi afişarea tuturor pixelilor este o metodă sigură dar costisitoare. Pentru a reduce costurile, algoritmul subdivizării ecranului, prezentat mai întâi, încearcă să evite mersul atât de departe în descompunerea unei imagini. El nu utilizează primitive de trasare de linii sau poligoane, doar pixelul şi dreptunghiul, care este un caz special de poligon.

Algoritmul nu trasează un poligon oarecare, dar trebuie să fie în măsură să decidă dacă un poligon din scenă se suprapune peste un dreptunghi sau peste un pixel. Este nevoie să ştim dacă două segmente de dreaptă se intersectează, dar nu ne interesează punctul de intersecţie. (Există şi versiuni ale algoritmului care determină şi utilizează punctele de intersecţie, dar ele nu intră în categoria metodelor de subdivizare a ecranului "pure". Ele se situează între acestea şi cele de sortare în adâncime. Ele mai trebuie să poată desena un poligon plin de formă arbitrară.)

Algoritmul subdivizării ecranului poate fi modificat pentru trasarea numai a conturului poligoanelor. El ar fi un pseudo-algoritm de eliminare a liniilor ascunse şi este valabil doar pe display-uri raster.

Deci, pe parcursul acestui capitol, obiectele din spaţiu vor fi considerate ca având suprafeţe poligonale plane. În afara acestei condiţii, nu se mai impune nici o altă restricţie în privinţa complexităţii scenei grafice. Obiectele pot să se întrepătrundă sau să se acopere ciclic.

2 Metoda Subdivizării Ecranului (Algoritmul Warnock)

Metoda subdivizării ecranului a fost prezentată de John Warnock şi deseori este numită algoritmul Warnock. Ea poate fi utilizată pentru afişarea de scene în care obiectele sunt mărginite de suprafeţe poligonale plane. Scenele pot avea orice complexitate. În această privinţă, algoritmul este la fel de puternic ca şi algoritmul Z-buffer, prezentat mai târziu, fără să necesite un buffer de adâncime, dar este mai complicat. Versiunea algoritmului pictorului prezentat în capitolul precedent nu trata obiecte care să se întrepătrundă sau care să se acopere ciclic. Aici nu există asemenea restricţii.

Poligoanele care descriu suprafeţele obiectelor pot avea orice formă. Totuşi, vom lua în considerare doar triunghiul. Aceasta nu este o restricţie în privinţa complexităţii scenei grafice, întrucât, după cum s-a văzut, orice poligon poate fi descompus în triunghiuri. Motivul limitării doar la triunghi constă în sistematizarea structurii de date care descrie obiectele. Dacă, dintr-un motiv oarecare, nu dorim să descompunem poligoanele în triunghiuri, se poate implementa un test mai general de acoperire, dar aceasta nu afectează ideea de bază a Algoritmului Warnock.

În unele privinţe, metoda este întrucâtva legată de ideea care stă la baza versiunii fără buffer a algoritmului Z-buffer, descris în următoarea secţiune. Prin această metodă, modul în care se colorează o zonă dată a ecranului se stabileşte pentru zona cea mai mică posibilă, un

pixel. Individual, pentru fiecare zona de dimensiunea unui pixel, se calculează adâncimea în centrul pixelului pentru toate poligoanele care trec prin acel pixel.

În algoritmul Warnock nu se ia decizia neapărat la nivelul pixelului. Algoritmul încearcă să determine culoarea pentru cea mai mare zonă posibilă, în particular un dreptunghi care conţine mai mulţi pixeli. Dacă ajunge la concluzia că un întreg dreptunghi se poate umple cu una şi aceeaşi culoare, o face. Acest lucru nu prea e posibil pentru dreptunghiuri mari, dar mult mai posibil pentru dreptunghiuri mai mici şi întotdeauna pentru pixeli.

Forma de bază a algoritmului trasează dreptunghiuri umplute cu o culoare solidă, dar acestea pot fi oricât de mici, chiar şi de dimensiunea unui pixel. Se porneşte cu un dreptunghi de mărimea ecranului şi se verifică dacă poate fi umplut. Dacă dreptunghiul este gol sau conţinutul său aparţine unui singur poligon, este umplut cu culoarea de fond sau cu culoarea acelui poligon. Dacă nu este cazul, dreptunghiul este micşorat prin subdivizarea lui în patru dreptunghiuri şi se reiau testele pentru fiecare dreptunghi. Procesul poate continua până când dreptunghiul ajunge de mărimea unui pixel. În acest moment, întotdeauna se poate cunoaşte culoarea de afişat.

Există implementări ale acestui algoritm care trasează nu numai dreptunghiuri, ci şi poligoane de diferite forme. Ele diferă de algoritmul de bază prin aceea că un dreptunghi poate fi umplut şi atunci când conţinutul său aparţine doar în parte unui poligon. Dreptunghiul este umplut în parte cu culoarea de fond, iar cealaltă parte cu poligonul decupat relativ la acel dreptunghi. Această modalitate încearcă să evite subdivizările. Acest lucru este însă în contradicţie cu ideea de bază, aceea de a elimina decuparea de poligoane.

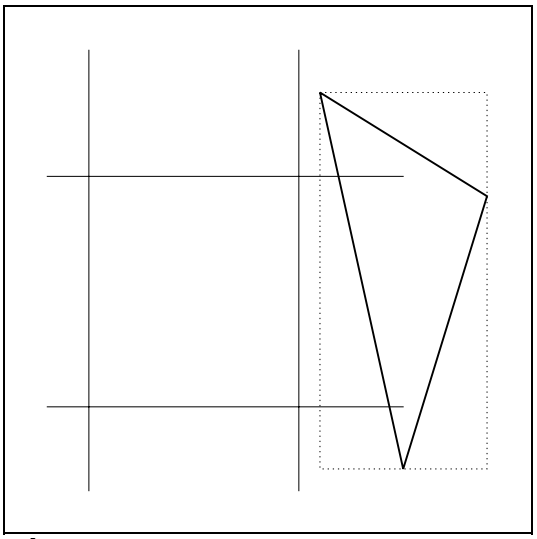

Figura 2.1

Vom descrie algoritmul de bază în detaliu. Mai târziu vom contura versiuni care conţin decuparea şi umplerea de poligoane. Aceasta necesită calcule suplimentare care nu ne aduc nici un câştig în cazul scenelor mai complicate.

Mai întâi vom utiliza pseudocodul pentru a descrie versiunea de bază, fără să mai calculăm intersecţiile dintre dreptunghiuri şi triunghiuri. În pseudocod doar determinăm dacă există sau nu suprapunerea. Poligoanele testate - în cazul nostru triunghiurile - sunt proiecţii perspectivă ale poligoanelor care compun scena grafică.

```
Pentru un dreptunghi dat:
   - dacă dreptunghiul are mărimea unui pixel, verifică toate
     triunghiurile dacă conţin centrul pixelului.
        - dacă nici un triunghi nu conţine centrul pixelului,
          setează pixelul pe culoarea de fond.
        - dacă unul sau mai multe triunghiuri conţin centrul
          pixelului, setează pixelul pe culoarea triunghiului
          cel mai apropiat de ecran în acel punct.
   - dacă dreptunghiul este mai mare decât un pixel, verifică
     dacă nu există triunghiuri care să acopere dreptunghiul.
        - dacă nu există triunghiuri care să se suprapună, umple
```

```
            dreptunghiul cu culoarea de fond.
        - dacă unul din triunghiuri acoperă în întregime
          dreptunghiul şi este mai apropiat de celelalte
          triunghiuri care acoperă dreptunghiul, umple
          dreptunghiul cu culoarea triunghiului.
          Altfel subdivide dreptunghiul.
end.
```

Se observă că se face un test simplu de adâncime, fie atunci când dreptunghiul devine de mărimea unui pixel, fie în cele patru colţuri ale dreptunghiului. Nu se baleiază întreaga scenă la nivel de pixel; ori de câte ori se găseşte că un dreptunghi are o anumită culoare, este afişat, fără a mai fi subdivizat.

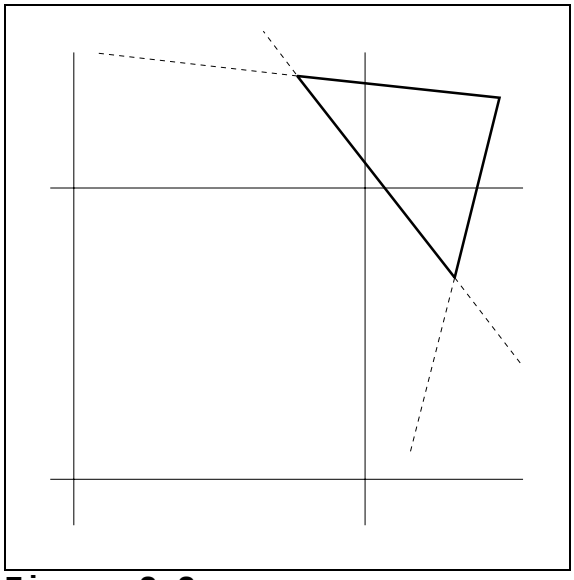

Figura 2.2

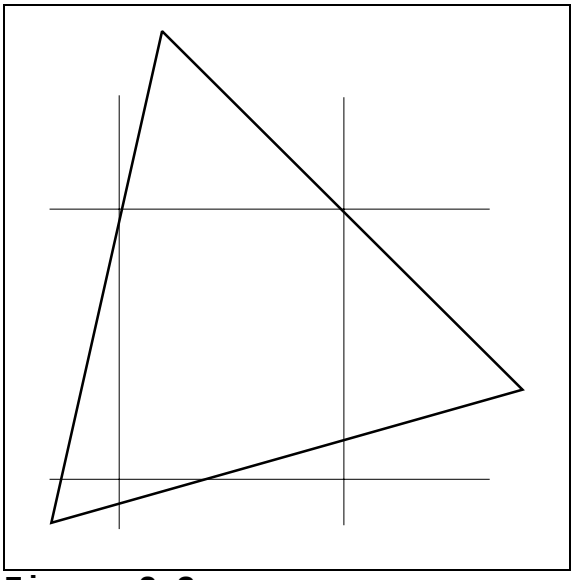

Figura 2.3

Algoritmul necesită o serie de criterii analitice pentru a determina dacă un triunghi se suprapune peste un dreptunghi dat şi dacă da, dacă dreptunghiul se încadrează complet în triunghi sau nu. Pentru aceasta, presupunem că avem următoarele coordonate pentru triunghi şi dreptunghi:

- vertex-urile triunghiului:
  $P_1 = (x_1,y_1)$
  $P_2 = (x_2,y_2)$
  $P_3 = (x_3,y_3)$
- limitele zonei:
  x_l - marginea stânga
  x_r - marginea dreapta
  y_b - marginea de jos
  y_t - marginea de sus

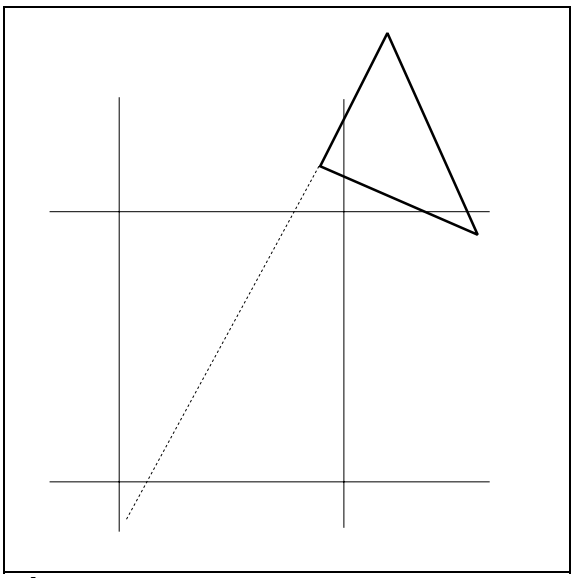

Figura 2.4

Se poate utiliza testul minimax pentru a vedea dacă triunghiul este în întregime în exteriorul uneia din margini. Aplicăm mai întâi acest test, întrucât este mai puţin costisitor. Calculăm:

$$x_{min} = min(x_1, x_2, x_3)$$
$$x_{max} = max(x_1, x_2, x_3)$$
$$y_{min} = min(y_1, y_2, y_3)$$
$$y_{max} = max(y_1, y_2, y_3)$$

Dacă vreuna din condiţiile de mai jos este îndeplinită, nu avem suprapunere:

$$x_{max} < x_l$$
$$x_r < x_{min}$$
$$y_{max} < y_b$$
$$y_t < y_{min}$$

Testele de mai sus verifică dacă un triunghi, cum este cel din Figura 2.1, nu se suprapune. Triunghiurile care nu sunt eliminate de acest test încă mai pot fi disjuncte faţă de dreptunghi. Dorim să ştim aceasta întrucât, dacă nu avem suprapunere, putem evita alte subdivizări. Deci trebuie să efectuăm şi alte teste.

Următorul test este unul de intersecţie latură-dreptunghi. Acesta poate detecta unele cazuri în care nu există suprapunere. Segmentul dintre punctele (x_1, y_1) şi (x_2, y_2) are ecuaţia

$$f(x,y) = (x-x_1)(y_2-y_1)-(y-y_1)(x_2-x_1)$$

Dacă $f(x_l, y_b)$, $f(x_l, y_t)$, $f(x_r, y_b)$, $f(x_r, y_t)$ au toate acelaşi semn, atunci latura (x_1, y_1) (x_2, y_2) nu intersectează dreptunghiul. Se aplică acest test pentru fiecare din cele trei laturi ale triunghiului. Dacă nici una din laturi nu intersectează dreptunghiul, atunci triunghiul fie nu se suprapune (Figura 2.2), fie conţine întregul dreptunghi (Figura 2.3). Pentru distingerea acestor situaţii este necesar un test de interior de triunghi, descris în capitolul precedent.

Dacă unul din teste indică o intersecţie, nu e neapărat necesar ca triunghiul să se suprapună peste dreptunghi (Figura 2.4). Sunt necesare alte teste mai costisitoare. Cazurile posibile sunt prezentate în Figura 2.5.

Poziţiile a şi b pot fi detectate verificând dacă vertex-urile triunghiului se află în interiorul dreptunghiului. Dacă (x,y) este un vârf al triunghiului, testăm:

$$x_l < x$$
$$x < x_r$$
$$y_b < y$$
$$y < y_t$$

Dacă toate condiţiile sunt îndeplinite atunci vertex-ul (x,y) este în interiorul dreptunghiului. Se testează în acest fel toate cele trei vârfuri, până când găsim unul în interior. În acest caz avem suprapunere.

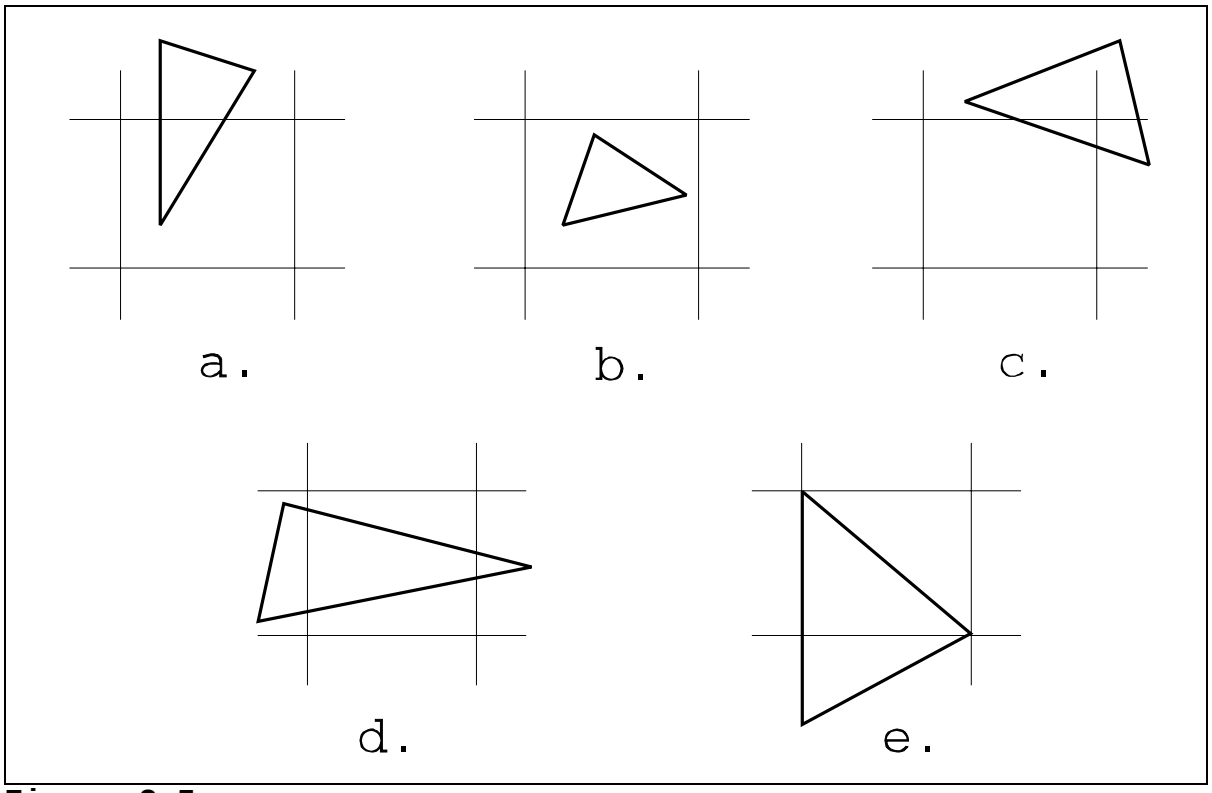

Figura 2.5

Cazurile c, d şi e nu pot fi detectate prin acest test. Cazul c se rezolvă printr-un test de interior de triunghi. Pentru cazul d efectuăm testul de intersecţie de laturi descris în capitolul 1 (care nu este identic cu testul extins de mai sus). Acest test detectează şi cazurile a şi c dar nu şi cazul b. Cazul e este una din situaţiile de "coincidenţă" în care o latură a triunghiului coincide cu o latură a dreptunghiului iar alta cu diagonala dreptunghiului sau cu altă latură a lui. Determinarea acestor situaţii este costisitoare. Ea poate fi evitată prin calcul în virgulă flotantă şi prin modificarea coordonatelor triunghiului astfel încât să nu mai fie cu valorile întregi ale coordonatelor ecranului. În continuare nu vom mai trata aceste cazuri, întrucât ele nu se întâlnesc frecvent.

Această versiune a algoritmului trebuie să determine, pentru un triunghi şi un dreptunghi date, dacă sunt disjuncte, dacă se intersectează sau dacă dreptunghiul este în interiorul triunghiului. Algoritmul mai trebuie să determine adâncimea unui triunghi într-un anumit punct. Întrucât scena este proiectată în perspectivă, adâncimea se calculează ca intersecţia unei drepte paralele cu axa Oz cu planul triunghiului după transformare. Aceasta înseamnă că mai întâi se aplică o transformare perspectivă asupra tuturor vertex-urilor care compun scena. Valorile (x,y) ale proiecţiilor triunghiului sunt utilizate pentru determinarea suprapunerii, iar coordonata z pentru calcule de adâncime.

Pentru a evita transformările repetate ale proiecţiilor vertex-urilor în coordonate ecran, vom scala toate perechile (x,y) proiectate perspectivă cu valorile adecvate.Este vorba de o transformare simplă din coordonate utilizator în coordonate ecran, prezentată mai jos. Prin această transformare mai putem determina mărimea obiectelor.

Valorile adâncimii se calculează cu formula prezentată la sortarea geometrică. Structura înregistrării pentru fiecare triunghi va conţine coeficienţii ecuaţiei planului triunghiului, pentru a nu relua calculele.

Dacă `ax + by + cz - d = 0` este ecuaţia unui plan, adâncimea `z` într-un punct al ecranului (x_s, y_s) este

$$z = \frac{-ax_s - by_s + d}{c}$$

Atunci când dreptunghiul devine de mărimea unui pixel, calculăm adâncimea pentru toate triunghiurile care conţin acel pixel. Cel mai apropiat triunghi este cel cu z-ul cel mai mic (lhs).

Când dreptunghiul este mai mare decât un pixel, pentru fiecare triunghi care îl conţine calculăm adâncimea în cele patru colţuri ale dreptunghiului. Pentru fiecare triunghi care îl intersectează, calculăm adâncimea planului său în cele patru colţuri. Dacă unul din triunghiurile care conţin dreptunghiul are toate valorile de adâncime mai mici decât toate celelalte, acest triunghi este în mod sigur cel mai apropiat de ecran.

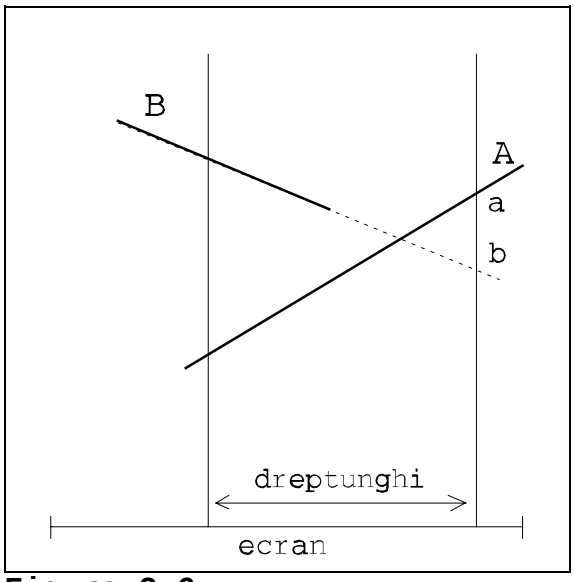

Figura 2.6

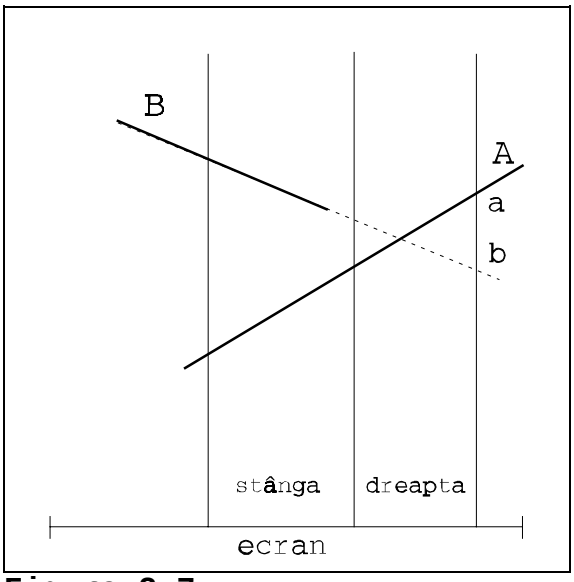

Figura 2.7

Aceasta este o condiţie suficientă, însă nu şi necesară. În Figura 2.6 avem ecranul şi două triunghiuri prezentate îngroşat. Adâncimea se calculează în colţuri, prezentate cu linie punctată. Triunghiul A care conţine dreptunghiul este mai aproape decât triunghiul B, dar algoritmul nu poate verifica aceasta pentru că punctul b este mai aproape decât a. Pentru aceasta, dreptunghiul trebuie divizat. După divizare (vezi Figura 2.7), situaţia este concludentă pentru dreptunghiul din stânga. În partea dreaptă există doar un singur triunghi, deci ambele dreptunghiuri pot fi umplute.

Îmbunătăţiri ale Algoritmului Putem evita testele repetate de suprapunere utilizând liste înlănţuite, în limbaje gen Pascal sau C. Putem reduce numărul de triunghiuri care trebuie comparate cu fiecare dreptunghi în procesul de divizare, pe baza informaţiilor acumulate pe nivele anterioare ale divizării. Dacă un triunghi conţine un dreptunghi, el va conţine şi toate părţile în care acesta este divizat. Dacă un triunghi este în afara unui dreptunghi, el va fi în afara tuturor subdiviziunilor acestuia. Deci, pentru dreptunghiurile rezultate în urma divizării, această informaţie nu mai trebuie calculată din nou, trecând prin toate testele. Cum se

realizează aceasta?

Vom utiliza trei liste de pointeri la triunghi. Una va conţine toate triunghiurile disjuncte; o vom numi Dlist. Alta va conţine toate triunghiurile care conţin dreptunghiul, numită Slist. A treia va conţine toate triunghiurile care intersectează dreptunghiul - Ilist. Aceste liste trebuie actualizate pe parcursul procesului de divizare. Se observă că Dlist şi Slist pot doar să crească, iar Ilist să se micşoreze pe măsură ce subdiviziunile sunt tot mai mici.

Vom extrage triunghiurile din Ilist şi le vom adăuga la Dlist sau Slist odată cu rafinarea subdivizării. Triunghiurile din Dlist nu se testează deloc; ele nu ne mai interesează. Aceasta poate accelera semnificativ procesul în cazul scenelor complexe, cu multe triunghiuri. Slist şi Dlist au comportare de stivă: ceea ce intră la început poate fi extras la sfârşit. Atunci când procesăm un dreptunghi, mai întâi mutăm un număr, d, de triunghiuri din Ilist în Dlist, şi un alt număr, s, din Ilist în Slist. Atunci când acest dreptunghi este umplut, fie direct, fie prin subdivizări, înainte de a reveni în ierarhie, în Ilist vor reveni mai întâi primele s triunghiuri din vârful lui Slist şi primele d din Dlist. Numai Ilist nu se comportă ca o stivă. Elementele care trebuie extrase din Ilist pot avea orice poziţie în această listă.

Subdivizarea unui dreptunghi se poate realiza prin patru apeluri recursive. Fiecare apel recursiv poate utiliza variabile locale care să conţină numărul de elemente care se transferă în Slist şi Dlist. Înaintea revenirii se transferă înapoi în Ilist numărul de triunghiuri corespunzător fiecărei stive. O altă posibilitate, pe care o vom utiliza în continuare, este să folosim pointeri locali spre începutul lui Dlist, respectiv Slist, aşa cum arată la începutul procesării dreptunghiului. Înainte de a încheia procesul, toate elementele dintre vârf şi fiecare dintre aceşti pointeri vor fi transferate înapoi în Ilist.

La început, când primul dreptunghi este primul ecran, Ilist va conţine toate triunghiurile care compun scena, iar celelalte două liste vor fi vide. Vom arăta modul în care se întreţin listele într-un caz general, undeva pe parcursul procesului de divizare.

Pentru un dreptunghi dat, se efectuează testul de suprapunere peste toate triunghiurile din Ilist. Cele care conţin dreptunghiul sunt mutate în Slist iar cele disjuncte în Dlist. Apoi se calculează adâncimea fiecărui triunghi din Slist şi a planului fiecărui triunghi din Ilist. Dacă se poate umple dreptunghiul fără a mai fi necesare divizări suplimentare, toate triunghiurile mutate în Dlist şi Slist pentru acest dreptunghi se duc înapoi în Ilist. Aceasta ne garantează faptul că, după procesarea dreptunghiului, listele vor fi în aceeaşi stare ca şi înainte. Altfel facem o subdivizare şi avansăm un nivel de recursivitate.

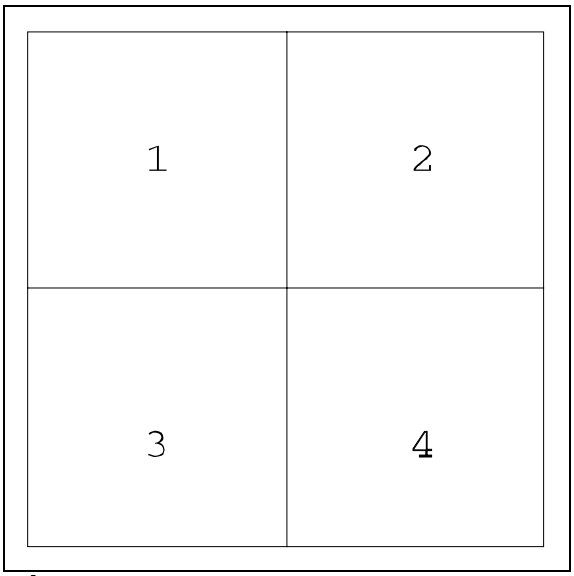

Figura 2.8

Vom utiliza următoarea convenţie. Subdiviziunile unui dreptunghi dat se procesează în ordinea 1, 2, 3, 4, ca în Figura 2.8. Subdiviziunile dreptunghiului 2 sunt identificate prin 21, 22, 23, 24, deci dreptunghiul marcat în Figura 2.9 este 412. Dreptunghiul originar de la care se porneşte este identificat printr-un spaţiu - el precede toţi ceilalţi identificatori. Pointerii locali către vârful lui Slist şi Dlist la începutul procesului sunt Spointer şi Dpointer; la aceste nume vom adăuga numele dreptunghiului. Deci atunci când procesăm

dreptunghiul 412 vom avea Spointer412 şi Dpointer412. Listele sunt simplu înlănţuite şi se termină cu nil, indicat aici printr-un 0.Elementele noi se inserează în faţă.

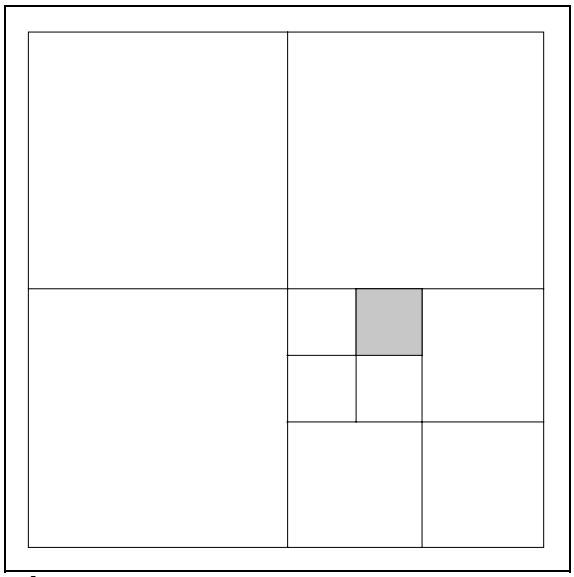

Figura 2.9

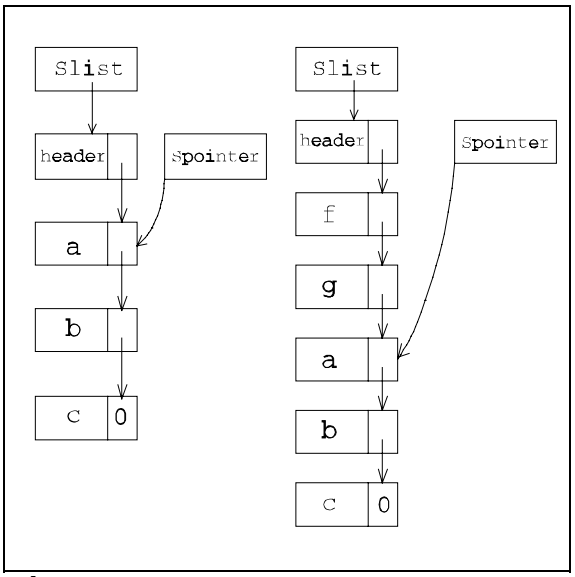

Figura 2.10

În Figura 2.10 avem un exemplu de Slist. Spointer indică primul element. După inserarea a două noi elemente, Spointer va indica "mai jos", deşi nu este modificat. Lista este identificată printr-un header, pentru a putea extrage elemente. El este reprezentat în Figura 2.10, dar în continuare îl vom ignora.

În Figura 2.11 avem ecranul, cu trei triunghiuri A, B, şi C. La început listele vor fi:

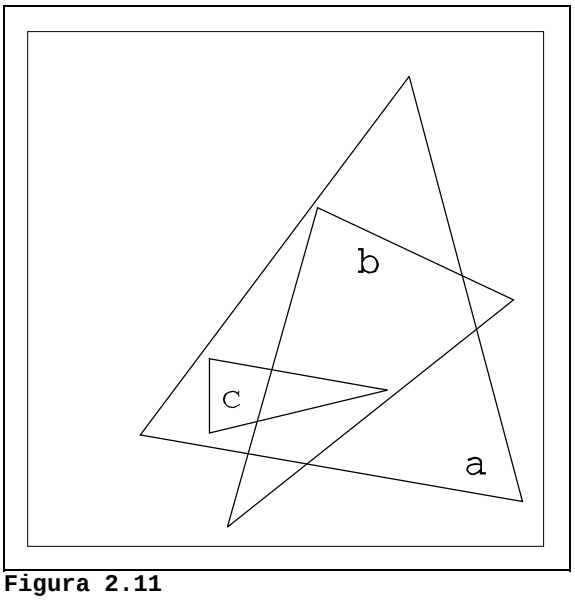

Figura 2.11

```
Spointer: 0
Dpointer: 0
Ilist → A → B → C → 0
Slist → 0
Dlist → 0
```

La procesarea dreptunghiului 1 triunghiurile b şi c trec în Dlist, ele fiind disjuncte (vezi Figura 2.12)

```
Spointer1: 0
Dpointer1: 0
Ilist → a → 0
Slist → 0
Dlist → b → c → 0
```

După procesarea dreptunghiului 1, listele vor arăta la fel ca la început, întrucât procesul mută înapoi elementele din Slist şi Dlist. Trecem direct la dreptunghiul 4. Listele nu se modifică,

întrucât toate triunghiurile îl intersectează (vezi Figura 2.13).

```
Spointer4: 0
Dpointer4: 0
Ilist → a → b → c → 0
Slist → 0
Dlist → 0
```

La procesarea dreptunghiului 41 triunghiul a trece în Slist.

```
Spointer41: 0
Dpointer41: 0
Ilist → b → c → 0
Slist → a → 0
Dlist → 0
```

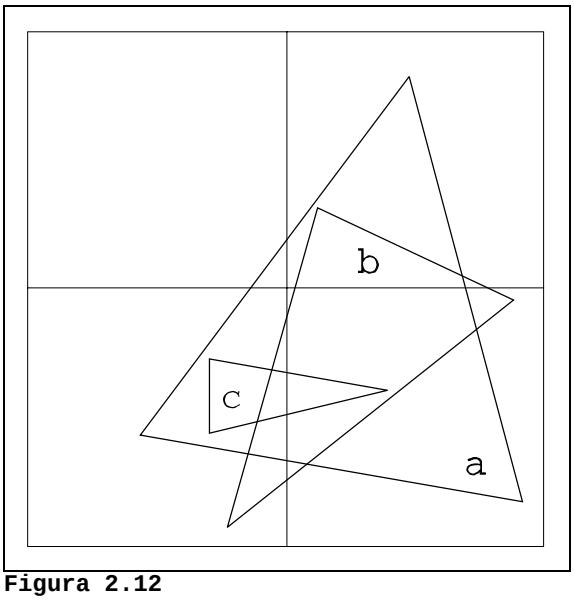

Figura 2.12

În Figura 2.14 avem dreptunghiul 41 mărit. La procesarea lui 412 triunghiul b trece în Slist, iar triunghiul c în Dlist.

```
Spointer 412: a
Dpointer 412: 0
Ilist → 0
Slist → b → a → 0
Dlist → c → 0
```

Dar dreptunghiul 412 poate fi umplut, deci nu mai sunt necesare alte subdiviziuni. După umplere, triunghiurile mutate în Slist şi Dlist la 412 sunt aduse înapoi în Ilist. Tot ceea ce se află în Slist de la vârf până la a exclusiv şi tot ceea ce se află în Dlist de la vârf până la 0 exclusiv se mută înapoi în Ilist. Deci, înaintea procesării dreptunghiului 413, listele vor fi din nou:

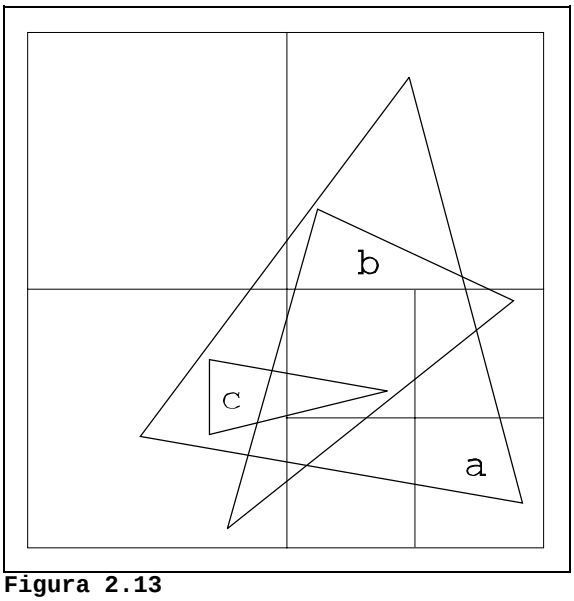

Figura 2.13

```
Ilist → b → c → 0
Slist → a → 0
Dlist → 0
```

413 primeşte informaţia că triunghiul a îl conţine. Se mută b în Slist şi se face divizarea. Toate subdivizările vor porni cu listele:

```
Ilist → c → 0
Slist → b → a → 0
Dlist → 0
```

4131 nu modifică aceste liste, întrucât triunghiul c încă îl mai intersectează. 41313 va adăuga triunghiul c la Slist:

```
Spointer41313: b
Dpointer41313: 0
Ilist → 0
Slist → c → b → a → 0
Dlist → 0
```

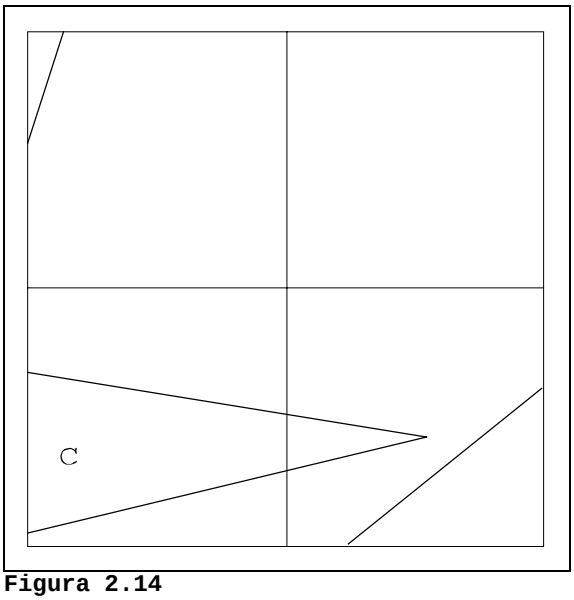

Figura 2.14

Se umple dreptunghiul, se aduce triunghiul c înapoi la Ilist şi se revine cu un nivel de recursivitate.

Am prezentat câţiva paşi în detaliu şi am presupus că triunghiurile care conţin dreptunghiul nu sunt niciodată mai apropiate decât cele care îl intersectează, astfel încât am făcut mai mulţi paşi în recursivitate pentru a arăta strategia de întreţinere a listelor. Desigur, se putea întâmpla ca un dreptunghi să poată fi umplut chiar dacă avem triunghiuri care îl intersectează; aceasta ar scurta procesul de subdivizare.

Se mai poate reduce numărul de triunghiuri luate în considerare, printr-un test de faţetă ascunsă. Acesta se implementează simplu considerând faţetele ascunse ca triunghiuri disjuncte. La procesarea primului dreptunghi - întregul ecran - aceste triunghiuri vor fi plasate în Dlist, pentru a nu mai fi luate în considerare.

Descriere în Pseudocod O descriere în completă în C sau Pascal ar fi foarte lungă. Preferăm să schiţăm ideile esenţiale în pseudocod şi să dăm câteva indicaţii utile.

Această versiune a algoritmului porneşte cu toate triunghiurile în lista înlănţuită Ilist. Pe parcursul prelucrării, ele vor fi distribuite între Ilist, Slist şi Dlist şi aduse înapoi, după cum s-a văzut mai sus. Deci avem nevoie de o rutină care să extragă unul sau mai multe triunghiuri dintr-o listă şi să le insereze în altă listă.

```
const num_vertex = { număr de vertex-uri }

type  point = record  x, y, z : real
              end;

  ptriunghi = ^triunghi;
  triunghi  = record
                    a, b, c    : integer;
                    pa, pb, pc : real;
                    next       : ptriunghi;
              end;

  t_poziţie=(conţine,disjunct,intersecţie);

var vertex: array [0..num_vertex] of point;

procedure DrawRec(xl, xh, yl, yh, color : integer);
begin
{ desenează dreptunghiul specificat prin coordonate ecran, cu
```

```
  culoarea "color" }
end;

function TestRecTri(r1,r2:point; ptr:ptriunghi) : t_poziţie;
begin
{ Compară dreptunghiul precizat prin două colţuri opuse, r1 şi
  r2, cu triunghiul indicat de "ptr".
  Efectuează testul de faţetă ascunsă, caz în care consideră
  triunghiul ca disjunct.
  Returnează una din cele trei poziţii posibile. }
end;

procedure Fill(xh, xl, yh, yl : integer);
var  midx, midy : integer;
     r1, r2     : point;
     poz        : t_poziţie;
     Spointer,
     Dpointer,
     ptr        : ptriunghi;

procedure MoveTo (list, ptr : ptriunghi);
begin
{ mută triunghiul indicat de "ptr" din Ilist în lista indicată
  de  "list" }
end;

begin
   if (xl = xh) and (yl = yh) then
   begin {pixel}
      { Cu "ptr" indicând, pe rând, toate triunghiurile din
        Slist, le calculează adâncimea în (xl,yl); cu "ptr"
        indicând, pe rând, fiecare triunghi din Ilist, testează
        dacă (xl,yl) este în interior; dacă da, calculează
        adâncimea.

        Dacă punctul aparţine unui triunghi din aceste liste,
        setează pixelul pe culoarea celui mai apropiat triunghi.

        Dacă nu aparţine nici unui triunghi, setează pixelul
        (xl,yl) pe culoarea de fond. }
   end
   else begin
      Spointer := { vârful Slist }
      Dpointer := { vârful Dlist }
      r1.x := xl; r1.y := yl;
      r2.x := xh; r2.y := yh;
      { "ptr" parcurgând Ilist: }
         poz := TestRecTri(r1 , r2 , ptr);
         case poz of
         conţine     : MoveTo(Slist, ptr);
         disjunct    : MoveTo(Dlist, ptr);
         intersecţie : { avans "ptr" }
         end {case};
      { Cu ptr parcurgând triunghiurile din Slist, calculează
```

```
        adâncimea triunghiurilor în cele 4 colţuri ale
        dreptunghiului xl,xh,yl,yh.
        Dacă unul din triunghiuri este mai apropiat în toate cele
        4 colţuri, atunci ptr parcurge Ilist şi calculează
        adâncimea planelor triunghiurilor în cele 4 colţuri.
        Dacă nici una din ecuaţiile planelor nu dă punct mai
        apropiat:
      }
      DrawRec (xl, xh, yl, yh, ptr^.color);
      { altfel: }
        midx := (xl + xh) div 2;
        midy := (yl + yh) div 2;
        if (midy < yh) then
           Fill(xl, midx, midy + 1, yh);
        if (midx < xh) and (midy < yh) then
           Fill(midx + 1, xh, midy + 1, yh);
        Fill(xl, midx, yl, midy);
        if (midx < xh) then
           Fill(midx + 1, xh, yl, midy);

      { mută toate triunghiurile de la vârful Slist până la
        Spointer înapoi în Ilist; la fel din Dlist. }
    end;
end { Fill };
```

Consideraţii de Implementare: Transformarea Coordonatelor Algoritmul utilizează coordonatele din *spaţiul imaginii* (2D) şi coordonatele din *spaţiul obiect* (3D). Parametrii algoritmului precizează colţurile dreptunghiului de pe ecran, deci vor fi de tip întreg. Astfel controlul apelurilor recursive este simplu. Numerele întregi dau o specificare precisă mărimii dreptunghiurilor, lucru necesar pentru a evita scrierea aceloraşi pixeli de mai multe ori. Subdiviziunile se determină în mod riguros atunci când limitele inferioare şi superioare ale dreptunghiului coincid. Dacă s-ar lucra în numere reale, acest lucru ar fi mult mai dificil.

Comparaţiile dreptunghi-triunghi trebuie efectuate în spaţiul coordonatelor imaginii. Întrucât descrierea obiectelor se face, de obicei, în coordonate utilizator, este necesară transformarea lor în sistemul de coordonate ale ecranului. Nu le vom rotunji la numere întregi, ci vom trata coordonatele dreptunghiului ca numere reale pe parcursul comparaţiei. Aceasta evită o serie de ambiguităţi la testul de interior şi la testele de intersecţie, după cum s-a arătat anterior.

Aceasta se realizează prin parcurgerea tabelei vertex-urilor şi transformarea lor înainte de umplere. Dăm mai jos un exemplu numeric. Să presupunem că o scenă este definită în coordonatele utilizator:

```
wxl = -2
wyl = -1.5
wxh = 2
wyh = 1.5
```

Pentru comoditate, de obicei, originea sistemului de coordonate este în centru. Această fereastră se va afişa pe tot ecranul, cu coordonatele ecran 0 - 639 pe x, 0 - 479 pe y. Pentru transformarea tuturor vertex-urilor:

```
for i := 1 to num_vertex do begin
   vertex[i].x := vertex[i].x * 160 + 320;
   vertex[i].y := vertex[i].y * 160 + 240
end;
```

(Coordonatele z nu trebuie transformate deoarece ele sunt utilizate doar pentru comparaţii de adâncime. Ecuaţiile planelor trebuie calculate pentru aceste coordonate transformate. Dacă se doreşte o proiecţie perspectivă, vertex-urile trebuie transformate perspectivă înaintea acestei transformări. În nici un caz, parametrii de iluminare nu pot fi calculaţi pornind de la vertex-urile transformate. Ei trebuie calculaţi anterior pe baza coordonatelor utilizator originale şi memoraţi în înregistrările triunghiului, altfel trebuie păstrat un set de vertex-uri netransformate.)

Găsirea celui mai Apropiat Triunghi Înconjurător Pentru orice dreptunghi dat pe parcursul subdivizării, este necesar să găsim triunghiul înconjurător care are cele mai mici valori de adâncime în cele 4 colţuri. Aceasta nu e atât de simplu pe cât pare.

Dacă Slist nu este vidă, este uşor de găsit în Slist un triunghi care are cea mai mică adâncime într-un anumit punct, de exemplu (xl,yl). Avem nevoie doar de un pointer la acest triunghi, ptr0.

În continuare efectuăm un test extins. Găsim triunghiul din Slist *şi* Ilist care are cea mai mică adâncime într-un colţ dat (pentru triunghiurile din Ilist aplicăm ecuaţia planului) şi obţinem un pointer la el. Efectuăm acest test extins pentru toate cele 4 colţuri ale dreptunghiului dat şi obţinem ptr1, ptr2, ptr3, ptr4.

Dacă cei 4 pointeri sunt egali cu ptr0, atunci ptr0 indică cel mai apropiat triunghi înconjurător şi dreptunghiul poate fi umplut cu acea culoare. Dacă nu avem egalitate, nu se fac alte teste. În acest caz este necesară divizarea.

O Versiune care Evită Unele Subdivizări Am menţionat înainte că există versiuni ale algoritmului lui Warnock care evită unele subdivizări succesive, dar ele necesită umplerea doar a unor porţiuni ale dreptunghiurilor. Din punctul nostru de vedere, aceasta încalcă spiritul metodei, dar dăm mai jos o descriere a unui asemenea algoritm:

- dacă nici un triunghi nu se suprapune cu dreptunghiul, umple dreptunghiul cu culoarea de fond
- dacă un singur triunghi se suprapune şi este conţinut în întregime, umple dreptunghiul cu culoarea de fond, apoi desenează triunghiul
- dacă un singur triunghi se suprapune şi este parţial conţinut în dreptunghi, umple dreptunghi cu culoarea de fond şi desenează doar porţiunea din triunghi care este în interior
- dacă un singur triunghi se suprapune şi înconjoară dreptunghiul, îl umple cu culoarea triunghiului
- dacă mai multe triunghiuri se suprapun şi unul dintre ele înconjură dreptunghiul şi este mai apropiat decât toate celelalte, umple dreptunghiul cu culoarea triunghiului

- în toate celelalte situaţii subdivide dreptunghiul.

3 Metodele Buffer-ului de Adâncime

Metodele Buffer-ului de Adâncime sunt cele mai puternice şi mai generale tehnici de îndepărtare a suprafeţelor ascunse. Există mai multe implementări şi majoritatea lor necesită un spaţiu tampon imens. Ele pot afişa realist chiar şi cele mai complexe scene, indiferent dacă obiectele sunt mărginite de suprafeţe curbe sau plane şi ele se pot întrepătrunde sau acoperi reciproc.

Aceste metode utilizează descrierile imaginilor şi obiectelor pentru efectuarea calculelor şi testelor şi obţin informaţia necesară la nivel de pixel. Deşi aceste metode pot prelucra şi obiecte cu suprafeţe curbe, pentru a nu intra în aspectele matematice ne vom limita la suprafeţe compuse din poligoane plane. (Mai târziu vom aborda tehnicile de umbrire, care pot să facă să apară ca suprafeţe curbe, chiar şi suprafeţele plane.) Vom prezenta două versiuni ale algoritmului buffer-ului de adâncime. Mai întâi vom vedea unele caracteristici şi aspecte preliminare.

Aceste metode se bazează pe algoritmul modificat de umplere a unui poligon. Obiectele de afişat sunt descrise prin specificarea poligoanelor care le mărginesc, cel mai convenabil prin definirea câte unei reţele de poligoane pentru fiecare obiect. Obiectele pot avea orice formă - nu există limitări la obiecte convexe. Suprafeţele pot să se întrepătrundă şi să se acopere ciclic.

Dacă în scenă apar şi poligoane concave, ele sunt specificate astfel încât pe baza primelor trei puncte să se poată determina corect orientarea, aşa cum s-a arătat în capitolul 1. Chiar dacă nu se respectă acest lucru, algoritmul este operaţional, întrucât calculul adâncimii nu necesită cunoaşterea orientării unui plan, dar înaintea lui nu se poate efectua îndepărtarea faţetelor ascunse. Vom presupune că poligoanele au fost descrise corect pentru această operaţie sau că ea s-a executat deja şi poligoanele sunt marcate dacă ele sunt faţete ascunse. IFA poate să înjumătăţească timpul de calcul.

Ambele versiuni ale algoritmului vor fi explicate pentru o proiecţie perspectivă, aceasta fiind cel mai des utilizată în grafica pe calculator. Metoda se simplifică mult dacă se aplică unei proiecţii ortografice. Vom utiliza transformarea perspectivă în adâncime, care păstrează imformaţia de adâncime pentru fiecare punct transformat.

Atunci când privim un poligon în spaţiu, vedem proiecţia acestuia. Imaginea lui pe ecran este făcută vizibilă prin setarea pe o anumită culoare a pixelilor din interiorul proiecţiilor marginilor poligonului. Se poate întâmpla ca mai multe poligoane, chiar şi total disjuncte în spaţiu, şă aibă proiecţiile suprapuse pe ecran: aceiaşi pixeli aparţin proiecţiilor mai multor poligoane. Într-un asemenea caz noi vedem doar poligonul cel mai apropiat de ecran. Cu alte cuvinte, setăm pixelii pe culoarea celui mai apropiat în acel punct. În esenţă, calculul distanţei trebuie efectuat pentru fiecare pixel al proiecţiei unui poligon.

3.1 Algoritmul Z-Buffer

Ideea de bază a *metodei Z-buffer* derivă din cele prezentate mai sus. Proiecţia unui poligon este baleiată linie cu linie şi fiecare linie pixel cu pixel. Pentru fiecare pixel se

calculează distanţa de la planul de vedere la acel punct al poligonului care se proiectează în pixel. Această distanţă se memorează în Z-buffer, numit şi buffer de adâncime. Valoarea memorată se compară cu adâncimile punctelor altor poligoane care se proiectează în acelaşi pixel.

În cel mai bun caz Z-buffer-ul are atâtea celule câţi pixeli sunt pe ecran. Fiecare celulă trebuie să conţină o *valoare de adâncime*, adică un număr real. Este uşor de imaginat că un asemenea buffer de adâncime va fi mult mai mare decât tot tamponul de cadre - o condiţie greu de îndeplinit de către sistemele grafice. Un display de înaltă rezoluţie de 1000×1000 de pixeli necesită un Z-buffer de un milion de numere reale, fiecare pe patru octeţi. Totuşi, există versiuni ale algoritmului care utilizează un Z-buffer mai mic.

Dorim să memorăm în Z-buffer distanţa până la punctul poligonului cel mai apropiat de ecran. Pentru a găsi minimul, iniţializăm tot Z-buffer-ul cu un număr mai mare, suficient ca să depăşească orice distanţă la orice punct al oricărui poligon. Atunci când se scanează proiecţia unui poligon, pentru fiecare pixel se calculează distanţa punctului corespunzător al poligonului şi se memorează, dacă este mai mică decât valoarea corespunzătoare din Z-buffer. În acelaşi timp, în tamponul de cadre se setează pixelul pe culoarea poligonului. Atunci când se finalizează procesul pentru toate poligoanele, în Z-buffer se va găsi adâncimea pentru cel mai apropiat poligon în acel punct şi în tamponul de cadre se va găsi culoarea acelui poligon. Cu alte cuvinte, tamponul de cadre va conţine toate părţile vizibile ale scenei.

La implementarea algoritmului vom utiliza proprietăţile geometrice ale liniilor, planelor şi proiecţiilor perspectivă, pentru a reduce calculele, care par a fi destul de extinse.

Descrierea Algoritmului Vom prezenta algoritmul Z-buffer pentru un singur poligon, dintr-o scenă compusă din mai multe poligoane. Operaţiile descrise trebuie efectuate pentru fiecare poligon din scenă.

Întrucât vom lucra doar cu adâncimi relative, Z-buffer-ul va fi iniţializat cu 1, care este cea mai mare adâncime în volumul de vedere.

Descrierea unei scene prin una sau mai multe reţele de poligoane nu conduce la ordinea de parcurgere secvenţială a poligoanelor. Un singur poligon din scenă este prelucrat după cum urmează.

a. **Transformarea Perspectivă în Adâncime** Vertex-urile poligonului se transformă în volumul de vedere după relaţiile:

$$x_t = x \cdot \frac{d}{d+z}$$
$$y_t = y \cdot \frac{d}{d+z}$$
$$z_t = z \cdot \frac{1}{d+z}$$

Poligonul astfel obţinut este tot 3D şi îl vom numi *poligon transformat*. Dacă urmează o umbrire sau iluminare, este necesar să salvăm vertex-urile netransformate, deci valorile transformate vor fi plasate într-o a doua tabelă.

b. **Ecuaţia Planului** Trebuie să determinăm ecuaţia planului căruia îi aparţine poligonul transformat. Luăm primele trei vertex-uri, (x_1, y_1, z_1), (x_2, y_2, z_3) şi (x_3, y_3, z_3) şi calculăm:

$$A = y_1(z_2-z_3)+y_2(z_3-z_1)+y_3(z_1-z_2)$$
$$B = z_1(x_2-x_3)+z_2(x_3-x_1)+z_3(x_1-x_2)$$
$$C = x_1(y_2-y_3)+x_2(y_3-y_1)+x_3(y_1-y_2)$$
$$D = x_1(y_2z_3-y_3z_2)+x_2(y_3z_1-y_1z_3)+x_3(y_1z_2-y_2z_1)$$

Cele patru valori, A, B, C, D determină ecuaţia planului în care se află poligonul transformat:

$$Ax+By+Cz-D = 0$$

Aceşti coeficienţi se memorează împreună cu poligonul transformat.

c. **Proiecţie Ortografică** Poligonul transformat se proiectează în planul de vedere. Această proiecţie nu necesită calcule - se ignoră coordonata z a fiecărui vertex. Vom numi proiecţia *poligon proiectat*.

d. **Clipping** Poligonul proiectat încă mai este definit în coordonate utilizator. Poziţia lui pe ecran depinde de fereastra specificată în planul de vedere. Presupunem că această fereastră este `(wxl, wyl, wxh, wyh)`. Mai trebuie precizat un viewport. Dacă scena nu se încadrează complet în fereastră, trebuie efectuat un clipping (2D). Scanarea poligonului proiectat se va rezuma doar la partea decupată. Se poate întâmpla ca un poligon să fie eliminat complet de această operaţie şi să nu mai rămână nimic de scanat.

e. **Transformarea de Vizualizare** Se face fără rotunjire, pentru că în pasul f se va utiliza un algoritm de umplere a poligoanelor. Dacă algoritmul de umplere acceptă coordonate utilizator pentru vertex-urile poligonului, se poate sări peste acest pas.

f. **Baleiere** Acesta este pasul cel mai complicat din proces. Pentru fiecare punct al poligonului care corespunde unui pixel trebuie să găsim distanţa până la planul de vedere. Numărul acestor puncte depinde de modul în care se vede poligonul pe ecran, adică de poziţia lui faţă de planul de vedere.

Deci, în procesul de scanare, trebuie ţinut cont de proiecţia finală a poligonului, în coordonate dispozitiv, decupată dacă e cazul. Parcurgem ecranul pixel cu pixel (adică în coordonate ecran absolute) şi pentru fiecare pixel determinăm adâncimea corespunzătoare.

Pentru un pixel dat pe ecran, mai întâi determinăm coordonatele reale corespunzătoare acestui pixel din transformarea de vizualizare. Aceasta se face printr-o transformare inversă din coordonate normalizate în coordonate reale. Pentru simplificare, presupunem că portul de vedere (viewport) este întregul ecran. Dacă se specifică un alt viewport, atunci trebuie să ţinem cont de acesta. Presupunem că coordonatele ecran absolute sunt între 0 şi xmax de la stânga la dreapta şi de la 0 la ymax de jos în sus. Avem nevoie de xmax şi ymax pentru transformarea din coordonate ecran în coordonate normalizate. Atunci pixelul (x,y) de pe ecran va corespunde punctului (x_w,y_w) în coordonate utilizator:

$$x_w = wxl + x\frac{wxh-wxl}{xmax}$$
$$y_w = wyl + y\frac{wyh-wyl}{ymax} \qquad \textbf{(1)}$$

Notăm

$$q = \frac{wxh-wxl}{xmax}$$

Pentru a găsi adâncimea punctului (x_w,y_w) trebuie să găsim punctul de pe poligonul transformat a cărui proiecţie este (x_w,y_w). Coordonata z a acestui punct este adâncimea relativă a punctului (x_w,y_w). Întrucât proiecţia poligonului transformat în planul de vedere este ortografică, pentru coordonata z calculăm intersecţia planului în care este situat poligonul transformat cu dreapta care trece prin (x_w,y_w), paralelă cu z.
Pentru aceasta se înlocuiesc x_w şi y_w în ecuaţia planului `Ax+By+Cz-D=0`. Obţinem

$$z(x_w,y_w) = \frac{-Ax_w-By_w+D}{C} \qquad \textbf{(2)}$$

Vom numi aceasta *adâncimea pixelului* (x,y). Întrucât parcurgem proiecţia poligonului pe ecran pixel cu pixel, x va avea pasul 1. Dacă x creşte cu 1, x_w creşte cu q. De aici rezultă

$$z(x_w+q,y_w) = \frac{-A(x_w+q)-By_w+D}{C} = z(x_w,y_w) - q\cdot\frac{A}{C}$$

Se observă că adâncimea unui pixel se modifică cu o constantă faţă de pixelul precedent (ecuaţia 3).

$$adâncime(x+1,y) = adâncime(x,y) - q\cdot\frac{A}{C} \qquad \textbf{(3)}$$

O relaţie asemănătoare s-ar putea dezvolta şi pentru pasul pe y când se trece la următoarea linie de scanare, dar câştigul este minor.
Pentru baleierea poligonului proiectat procedăm în felul următor. Mai întâi calculăm valoarea `q·A/C` (constantă pentru întregul poligon). Apoi se baleiază proiecţia poligonului cu algoritmul de umplere cunoscut, dar, în loc să se traseze o linie orizontală între doi pixeli, calculăm adâncimea pixelului de start cu formula (2) şi apoi adâncimea fiecărui pixel cu formula (3). Fiecare valoare calculată se compară cu cea corespunzătoare acelui pixel din Z-buffer şi dacă este mai mică, se memorează în Z-buffer şi acel pixel se setează pe culoarea poligonului. Întreaga proiecţie a poligonului trebuie baleiată în acest fel.

Procesul se reia de la pasul a pentru următorul poligonul, până când sunt prelucrate toate poligoanele.

Acest algoritm este cel mai simplu algoritm *universal* de îndepărtare a suprafeţelor ascunse. Singurul dezavantaj este că necesită un volum enorm de memorie pentru buffer. Secţiunea următoare prezintă o modalitate de aplicare a algoritmului fără a fi necesar asemenea spaţiu de memorare.

3.2 Algoritmul Liniei de Scanare

Acest algoritm se poate implementa sub mai multe forme. Vom descrie două dintre acestea. Prima utilizează un Z-buffer de dimensiunea unei linii de scanare. A doua nu necesită buffer de adâncime dar presupune mai multe calcule. Alte variaţiuni se bazează pe proprietăţile geometrice ale planelor, necesită mai puţine calcule dar sunt mai complicate ca algoritm, trebuie să întreţină structuri de date mai complexe şi sunt mai dificil de programat.

Numele algoritmului provine probabil din faptul că este mai convenabil să parcurgem liniile de scanare atunci când procesăm pixeli individuali.

Versiunea cu Buffer Această versiune utilizează un Z-buffer de mărimea unei linii de scanare, numit "*scan line buffer*". Este similar algoritmului Z-buffer de bază. Indiferent de structura de date care reprezintă poligoanele unei scene, trebuie să existe o modalitate de parcurgere secvenţială a tuturor poligoanelor. Aceasta va ajuta la ordonarea poligoanelor astfel încât să faciliteze prelucrarea ulterioară. Pentru ceea ce urmărim, aceasta este ordinea după valoarea maximă a lui y a proiecţiei poligonului. Pentru aceasta, le parcurgem poligon cu poligon ca mai jos (vezi Figura 2.15).

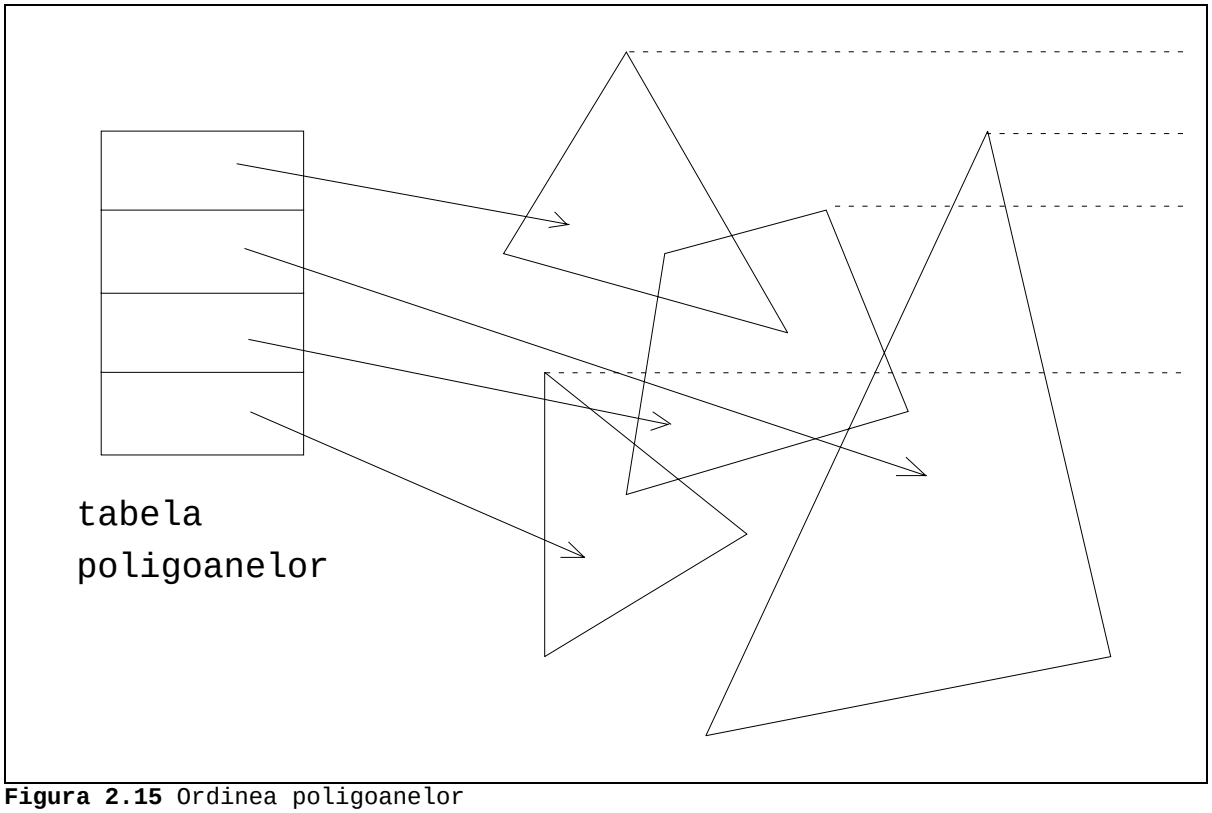

Figura 2.15 Ordinea poligoanelor

Pentru ordonarea corectă a poligoanelor trebuie parcurşi 5 paşi:

a. Transformarea perspectivă în adâncime a tuturor vertex-urilor
b. Calculul planului poligonului transformat
c. Proiecţie ortografică în planul de vedere - se ignoră informaţiile de adâncime
d. Clipping

e. Transformarea în coordonate ecran a tuturor vertex-urilor poligonului decupat, cu ajustarea cu 0.5 a coordonatei y (ca să nu fie valori întregi).

După cei cinci paşi, adăugăm acest poligon la o structură de date separată, care conţine poligoanele proiectate şi decupate, în coordonate ecran; fie această structură SPOL.

Va trebui să determinăm valorile y maximă şi minimă pentru fiecare poligon. ymax îl vom folosi drept cheie de sortare. Nu vom rearanja structura de date, dar vom utiliza pointeri pentru ordonarea descrescătoare. Apoi construim un tabel (îl vom numi tabel de poligoane) care va conţine nu numai pointerul, dar şi y_{max}, y_{min} şi coeficienţii ecuaţiei planului fiecărui poligon.

ymax	ymin	ABCD	pointer

Odată creat acest tablou, algoritmul se continuă după cum urmează. Se menţin doi indici, INPOL şi OUTPOL. Aceştia indică în tabel domeniul poligoanelor care sunt intersectate de linia de scanare curentă. La decrementarea liniei de scanare se modifică INPOL prin căutarea de y_{max} mai mare decât valoarea liniei de scanare, iar pentru OUTPOL căutăm valori y_{min} mai mari decât valoarea liniei de scanare. Nu este necesar să căutăm în tot tabloul. Poligoanele posibil de inclus sunt toate sub INPOL, iar cele de exclus sunt între INPOL şi OUTPOL.

Pentru fiecare linie de scanare, algoritmul iniţializează SLB cu 1 - adâncimea relativă maximă. Se actualizează INPOL şi OUTPOL. Poligoanele intersectate sunt cele cu indicele între OUTPOL şi INPOL.

Se ia un poligon individual din SPOL. În continuare se procedează asemănător cu prelucrarea unei linii de scanare de la algoritmul de umplere a poligoanelor.

Două vertex-uri consecutive formează o latură. Comparăm cele două valori y cu linia de scanare. Dacă una este mai mare şi una mai mică, linia de scanare intersectează latura şi trebuie calculat x-ul intersecţiei şi se rotunjeşte. Aceste valori se sortează crescător într-o tabelă pentru x. După prelucrarea tuturor laturilor unui poligon, avem un număr par de intrări în tabela x. În majoritatea aplicaţiilor, se lucrează numai cu poligoane convexe care nu se intersectează, de obicei doar triunghiuri. În acest caz, vom avea doar două valori x.

Pentru toţi pixelii dintre prima şi a doua intersecţie se calculează adâncimea relativă. Se compară aceasta cu valoarea existentă în poziţia corespunzătoare din SLB şi dacă este mai mică, se memorează şi se setează poziţia corespunzătoare din memoria video pe culoarea poligonului respectiv. Se continuă până la prelucrarea tuturor perechilor de intersecţii ale acestui poligon.

În acelaşi mod se prelucrează şi celelalte poligoane care intersectează linia de scanare curentă. Când toate poligoanele dintre OUTPOL şi INPOL au fost prelucrate, memoria video conţine toate părţile vizibile ale scenei pentru acea linie de scanare. Se repetă pentru toate liniile de scanare, de sus în jos.

Descriere Formală a Versiunii SLB

```
Setează toată memoria video pe culoarea de fond;

Creează SPOL, cu toate poligoanele, transformate în perspectivă,
proiectate, decupate, transformate în coordonate ecran;
Creează tabela_poligoanelor, cu pointeri în SPOL, sortată după
ymax;

for scan := ymax downto 0 do begin
   modifică INPOL şi OUTPOL conform noii valori a lui scan;

   setează tot SLB pe 1;

   for i := OUTPOL to INPOL do begin
      { poligonul i are vertex-uri de la 1 la Ki şi Ki + 1 = 1,
        ciclic }
      for J := 1 to Ki do begin
         calculează x-ul intersecţiei laturii de la vertex[J] la
            vertex[J+1] cu linia de scan;
         rotunjeşte x;
         sortează x-urile în tabela_x;
      end {for J};

      { avem un număr par de intersecţii: 2*n }
      for J := 1 to n do begin
         for X := tabela_x[J*2-1] to tabela_x[J*2] do
         begin
            { procesează perechea J }
            calculează adâncime(x,scan) cu coeficienţii ecuaţiei
               planului din tabela_poligoanelor[i];
            if adâncime(x,scan) < buff[X]
            then
               memorează adâncimea şi setează pixelul pe culoarea
                  poligonului;
         end {for X}
      end {for J}
   end {for i}
end {for scan}.
```

Versiunea fără Buffer Această versiune este ceva mai complicată. În locul unui buffer al liniei de scanare (SLB), se utilizează o *tabelă a liniei de scanare* (SLT - *scan line table*), mai mică decât un SLB.

Versiunea descrisă în continuare sare peste decuparea de la pasul d şi efectuează un clipping simplificat pe parcursul calculelor pentru fiecare linie de scanare. Din această cauză, nu mai este necesară crearea structurii de date SPOL.

Acţiunea are loc pe linii de scanare succesive. Începutul este ca şi la versiunea anterioară. Vertex-urile unui poligon suferă o transformare perspectivă, i se calculează coeficienţii ecuaţiei planului şi se memorează un pointer la acest poligon într-o tabelă de poligoane, sortată după ymax, descrescator. Deci tabela poligoanelor conţine doar valorile ymax şi ymin în coordonate utilizator, transformate perspectivă. Celelalte vertex-uri ale poligonului se ignoră în continuare.

Valoarea liniei de scanare se transformă în coordonate utilizator, pentru actualizarea tabelei poligoanelor. Pentru o valoare dată a liniei de scanare se actualizează pointerii INPOL şi OUTPOL, iar poligoanele intersectate se află între INPOL şi OUTPOL. Se calculează x-ul intersecţiilor liniei de scanare cu laturi şi se transformă în coordonate ecran. Acestea se sortează crescător într-o tabelă a liniei de scanare (SLT). Împreună cu valorile x, se memorează în SLT indicele poligonului în tabela poligoanelor.

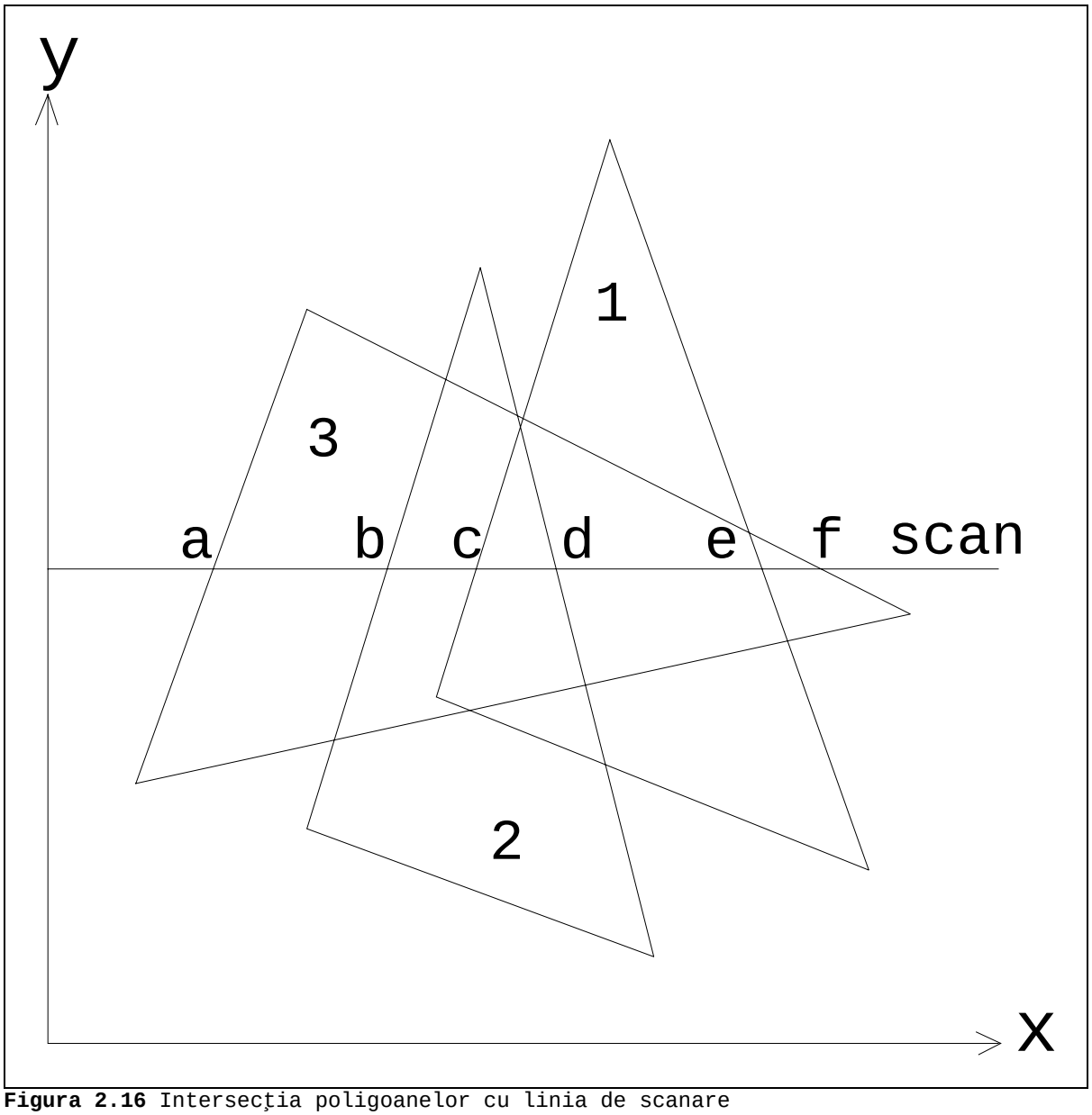

Figura 2.16 Intersecţia poligoanelor cu linia de scanare

Clipping-ul simplificat se efectuează pe măsură ce se introduc aceste valori. Dacă cele două capete ale unei laturi sunt situate de o parte şi de alta a liniei de scan (în coordonate utilizator), atunci se calculează x-ul intersecţiei şi se rotunjeşte. Dacă este înafara ecranului (presupunem că este de la 0 la 639), atunci, dacă `x < 0` se memorează 0 în tabel; dacă `x > 639` se memorează 639. Decuparea după y nu este necesară, întrucât liniile de scan nu iau valori înafara ecranului.

După procesarea tuturor poligoanelor intersectate, SLT conţine intersecţiile liniei de

scan curente cu laturile tuturor poligoanelor, împreună cu pointerii la poligoanele intersectate. Întotdeauna vom avea un număr par de asemenea pointeri la acelaşi poligon. În Figura 2.16 linia de scan are 6 intersecţii: a, b, c, d, e şi f.

SLT va conţine cele 6 valori, împreună cu pointerii în tabela poligoanelor (vezi Tabelul II).

Tabelul II

x (sortate):	a	b	c	d	e	f
ptr la poligon:	3	2	1	2	3	1
fanion 1:	off	off	off	on	on	on	off
fanion 2:	off	off	on	on	off	off	off
fanion 3:	off	on	on	on	on	off	off
nr.fanioane:	0	1	2	3	2	1	0

După completarea SLT parcurgem pixelii de la stânga la dreapta, pe linia de scanare. Se avansează până se ajunge la o intersecţie. Pe baza pointerului corespunzător se setează un fanion asociat poligonului, indicând faptul că acest poligon este activ. Indicatorul "activ" va fi pus pe *off* când se întâlneşte din nou un x aparţinând acestui poligon. Dacă există mai multe inersecţii, este posibil să fie setat de mai multe ori.

Pentru orice pixel dat vor fi active un număr de fanioane. Se disting trei cazuri:

1. Nr.fanioane = 0; pixelul nu este afectat.
2. Nr.fanioane = 1; se setează pixelul pe culoarea poligonului activ.
3. Dacă este mai mare decât 1, calculăm adâncimile relative ale punctelor corespunzătoare din toate poligoanele active, se compară şi se setează pixelul pe culoarea celui mai apropiat poligon activ.

Aceste operaţii se repetă pentru toate liniile de scanare, scanare, de sus în jos. După procesarea ultimei linii de scan, ecranul conţine toate părţile vizibile ale scenei.

Descriere în Pseudocod

```
Setează pagina video pe culoarea de fond;

creează tabela_poligoanelor, sortată după ymax;

for scan := high_y downto low_y do begin
   { scan, high_y şi low_y în coordonate utilizator! }
   actualizează INPOL şi OUTPOL;
   for i := OUTPOL to INPOL do begin
      { poligonul i are vertex-urile de la 1 la Ki şi Ki + 1 = 1,
        ciclic }
      for J := 1 to Ki do begin
         calculează x-ul intersecţiei laturii de la vertex[J] la
            vertex[J+1] (transformate);
         { x în coordonate ecran }
         decupează x;
```

```
            memorează x şi i în SLT, sortat;
         end { for J };
      end { for i };

      { setează o linie de pixeli }
      setează toate fanioanele pe off;
      for X := low_x to high_x do begin
         if (o intrare din SLT = X) then
            inversează fanionul corespunzător;
         if (nr_fanioane = 0) then nimic;
         if (nr_fanioane = 1) then
            setează pixelul pe culoarea poligonului activ;
         if (nr_fanioane > 1) then begin
            calculează adâncimile relative ale tuturor poligoanelor
               active, pe baza coeficienţilor planelor;
            setează pixelul pe culoarea celui mai apropiat poligon;
         end { if };
      end { for X }
end {for scan};
```

Evident, ambii algoritmi efectuează o serie de acţiuni şi operaţii aritmetice de mai multe ori, în loc să memoreze rezultatele în structuri de date adecvate, pentru a fi reutilizate. Câteva aspecte de ineficienţă:

1. Transformarea perspectivă a poligoanelor se face la crearea listei de poligoane şi, apoi, de fiecare dată când poligonul este intersectat de linia de scan.

2. Determinarea intersecţiilor laturilor unui poligon cu linia de scan se face prin testarea tuturor laturilor, în loc să fie păstrate sortate într-un tablou, ca şi la algoritmul de umplere a poligonului.

3. Intersecţiile liniei de scan cu laturile se calculează pentru fiecare linie de scan, în loc să fie memorate şi să se adune o constantă dedusă din panta acelei laturi.

4. Valorile adâncimilor poligoanelor se recalculează pentru fiecare punct al poligonului, deşi, pentru pixeli adiacenţi, ele diferă printr-o constantă care s-ar putea calcula o singură dată, ca la Z-buffer.

Dacă se au în vedere aceste aspecte, se elimină multe calcule. Algoritmul se complică însă foarte mult şi trebuie manipulate mult mai multe structuri de date.