

Eliminarea partilor nevizibile ale scenelor 3D din imagini

Prof. univ. dr. ing. Florica Moldoveanu

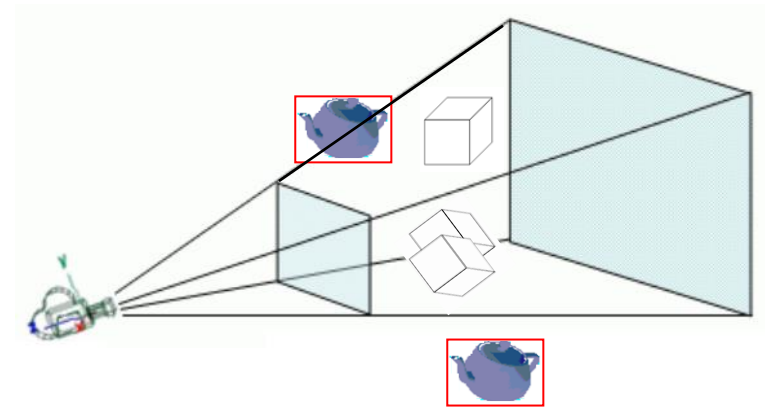
Eliminarea partilor nevizibile ale scenelor 3D, din imagini (1)

Vizibilitatea obiectelor dintr-o scena 3D, într-o imagine, depinde de:

- Pozitia observatorului si directia in care priveste
- Pozitionarea obiectelor din scena unul față de celalalt
- Volumul vizual

Partile unei scene 3D nevizibile într-o imagine sunt:

- Părți aflate total sau parțial în afara volumului vizual
- Fețe auto-obturate: fețele unui obiect obturate de celelalte parti ale aceluiasi obiect
- Părți obturate (ascunse) de obiecte ale scenei, aflate în față lor în raport cu observatorul



Eliminarea părților nevizibile ale scenelor 3D, din imagini (2)

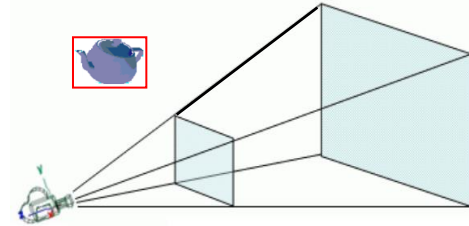
- ❖ Eliminarea părților dintr-o scenă 3D care nu sunt vizibile într-o imagine este efectuată la mai multe niveluri:

1. Eliminarea obiecte sau grupuri de obiecte: grupuri de primitive - Object culling:

1.1. Frustum Culling - eliminarea din banda grafică a grupurilor

de primitive aflate complet în afara volumului vizual: poate fi

efectuată pe CPU, de bibliotecă folosită de aplicația grafică 3D (motorul grafic).



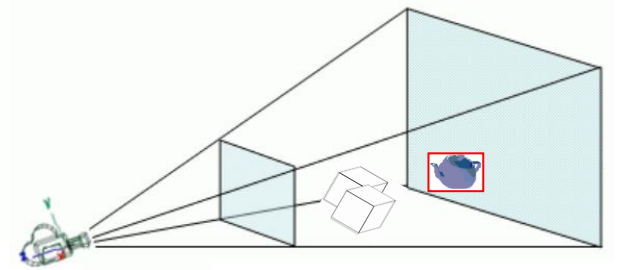
1.2. Occlusion Culling – eliminarea din banda grafică

a grupurilor de primitive care sunt complet obturate de

alte grupuri de primitive (obiecte): poate fi

efectuată pe CPU, prin algoritmi implementați într-o bibliotecă folosită de aplicația grafică

3D (motorul grafic).



Eliminarea părților nevizibile ale scenelor 3D, din imagini (3)

2. Eliminarea fețelor auto-obturate ale obiectelor - *Back face culling*: →

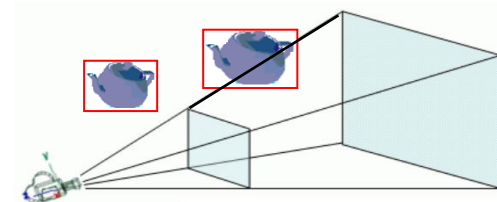


➤ este efectuată de GPU.

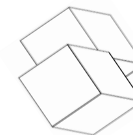
3. Eliminarea părților nevizibile ale primitivelor grafice:

3.1. Eliminarea primitivelor/părților de primitive aflate în afara

volumului vizual, prin operația de **decupare (clipping)** efectuată de GPU.



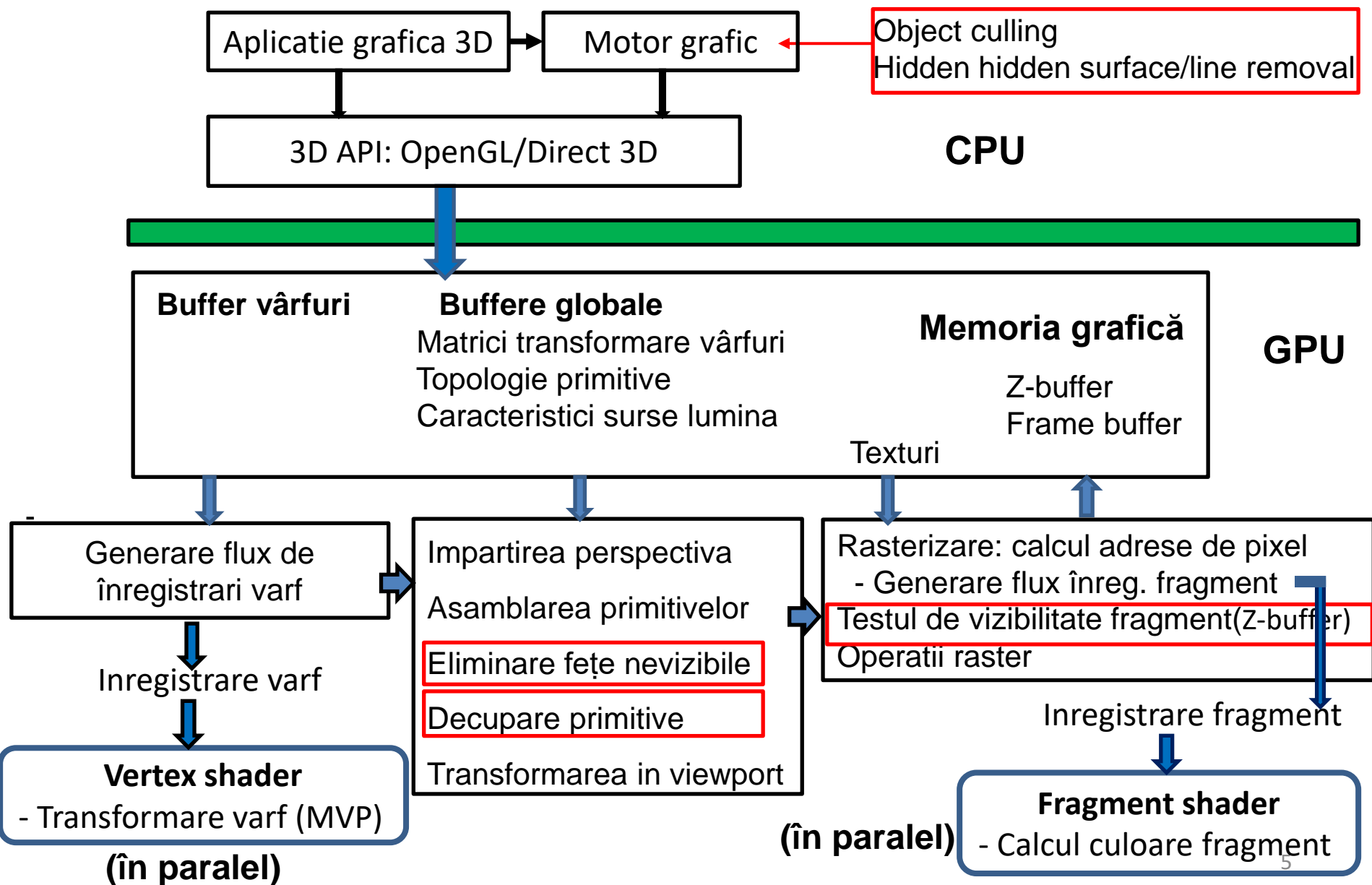
3.2. Eliminarea părților obturate (ascunse) de alte primitive grafice: ***hidden surface/hidden line removal***: poate fi efectuată prin algoritmi implementați în motorul grafic folosit de aplicație: algoritmul BSP, algoritmul Pictorului, ș.a. →



4. Eliminarea fragmentelor nevizibile

➤ este efectuată de GPU: fragmentele de primitive obturate de alte fragmente (aflate mai aproape de observator) sunt suprascrise la momentul rasterizării (algoritmul Z-buffer)

Eliminarea partilor nevizibile in banda grafica

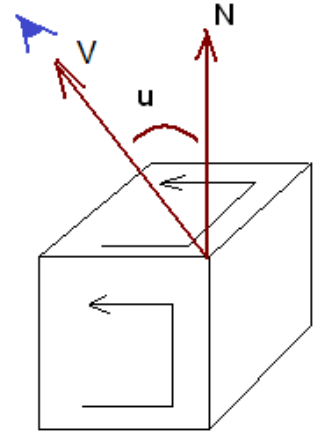


Eliminarea fețelor auto-obturate pentru poliedre convexe (Back face culling)(1)

- ❖ Poate reduce substantial numarul de poligoane rasterizate.
- ❖ Clasificarea fetelor unui obiect in “fete din fata (vizibile)” si “fete din spate” se face pe baza pozitiei obiectului fata de observator

Algoritmul se bazeaza pe urmatoarele presupuneri:

1. Observatorul si obiectul sunt raportati la acelasi sistem de coordonate.
2. Conturul fiecărei fețe a obiectului este orientat in sens trigonometric atunci cand obiectul este vazut din exterior; conditia asigura ca normala fiecărei fete vazuta din exterior este orientata catre spatiul exterior obiectului.
3. Observatorul este situat in afara obiectului.



O față este vizibila daca : $0 \leq u < 90^\circ$ sau $0 < \cos(u) \leq 1$ sau $\cos(u) > 0$

u: unghiul dintre normala la față si vectorul orientat catre observator

Produsul scalar: $N \cdot V = |N| * |V| * \cos(u) \rightarrow \cos(u) = N \cdot V / (|N| * |V|) = N_u \cdot V_u$

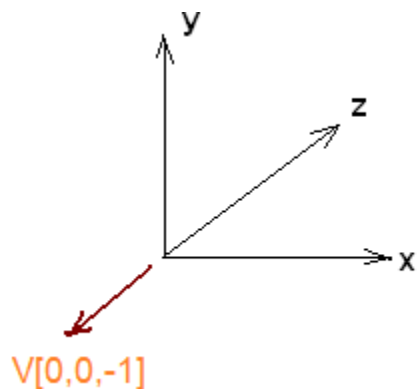
Conditia de vizibilitate: $N_u \cdot V_u > 0$ (produsul scalar al versorilor)

Determinarea fetelor auto-obturate pentru poliedre convexe(2)

Determinarea fețelor auto-obturate poate fi efectuată:

1. În sistemul coordonatelor globale (în care este definit și observatorul – camera virtuală)
2. În sistemul de coordonate observator.
3. În sistemul coord. de decupare, transformând poziția observatorului cu matricea $P*V$. **În acest**

caz:



Sistemul coordonatelor de decupare

$$N_u \bullet V_u = [n_x, n_y, n_z] \bullet [0, 0, -1] = -n_z$$

Condiția de vizibilitate devine: $n_z < 0$
(N_u și V_u au aceeași orientare)

După transformarea de proiecție poziția observatorului este la infinit, pe axa z negativă.

Vectorul orientat către observator este același pentru orice punct al unui obiect.

Determinarea fetelor auto-obturate pentru poliedre convexe(3)

Calculul normalei la o față a unui poliedru

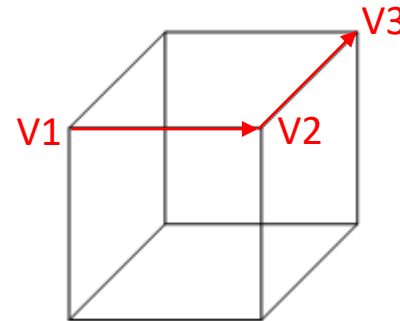
Fie $V1(x1, y1, z1)$, $V2(x2, y2, z2)$, $V3(x3, y3, z3)$,
3 varfuri succesive ale conturului feței.

Normala la față se poate obtine calculand produsul vectorial: $(V1-V2) \times (V2-V3)$

$$\mathbf{N} = (V1-V2) \times (V2-V3) = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ x2-x1 & y2-y1 & z2-z1 \\ x3-x2 & y3-y2 & z3-z2 \end{vmatrix} =$$

$$[(z3-z2)*(y2-y1) - (z2-z1)*(y3-y2)]*\mathbf{i} - [(x2-x1)*(z3-z2) - (x3-x2)*(z2-z1)]*\mathbf{j} \\ + [(x2-x1)*(y3-y2) - (x3-x2)*(y2-y1)]*\mathbf{k}$$

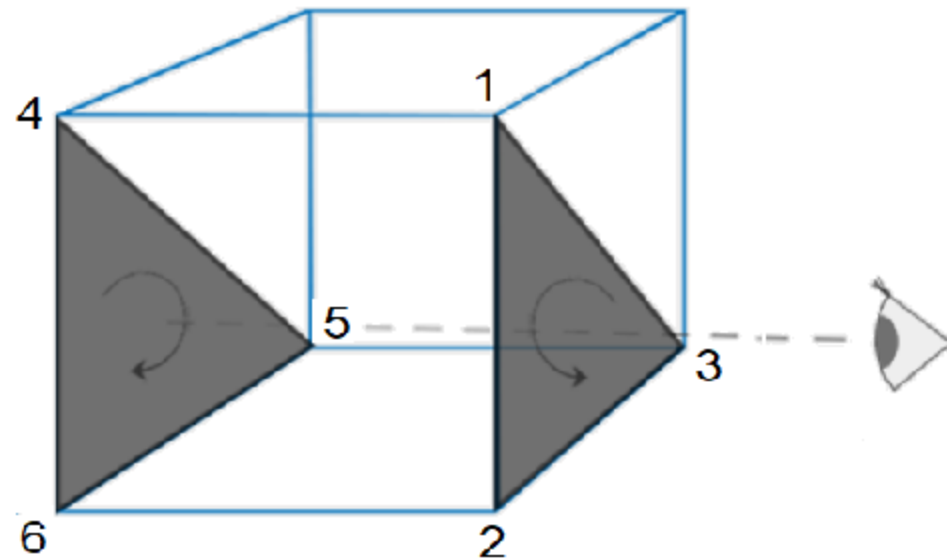
unde \mathbf{i} , \mathbf{j} , \mathbf{k} sunt versorii directiilor axelor sistemului de coordonate carteziane 3D



Determinarea fetelor nevizibile pe GPU (1)

- Presupunem ca **pentru toate fetele unui cub s-a definit în aplicatie orientarea trigonometrica a conturilor lor**, atunci and sunt vazute din exteriorul cubului: 1,2,3; 4,5,6, etc.
- Dupa transformarea de proiectie, in raport cu pozitia observatorului, ordinea varfurilor fețelor proiectate este:

- trigonometrica pentru fețele “din fata” (vizibile); ex. 1,2,3;
- anti-trigonometrica pentru fetele “din spate”; ex. ordinea 4,5,6 din pozitia observatorului este anti-trigonometrica – corespunde feței vazute din interior



Determinarea fetelor nevizibile pe GPU (2)

OpenGL da posibilitatea programatorului sa specifice care este orientarea fetelor “din față” și care fațe trebuie sa fie eliminate (sunt nevizibile).

1. Pentru activarea operatiei de eliminare a fetelor nevizibile (“face culling”) se apeleaza:

`glEnable(GL_CULL_FACE);` //implicit, operatia este dezactivata

2. Pentru a specifica orientarea fetelor “din față”: trigonometrica (GL_CCW)/sensul acelor de ceas (GL_CW):

`glFrontFace(GL_CCW/GL_CW);`

3. Pentru a specifica fețele care se dorește a fi eliminate: din față / din spate/toate

`glCullFace(GL_FRONT/ GL_BACK/GL_FRONT_AND_BACK);`

În cazul **GL_FRONT_AND_BACK** nu vor fi afișate fețe, ci numai alte primitive: puncte, linii.

Exemplu: dacă toate fețele unui obiect sunt orientate trigonometric, apelând

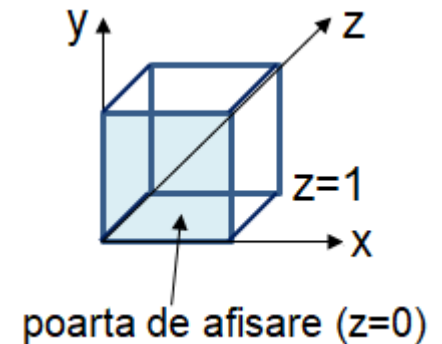
`glFrontFace(GL_CCW)` și `glCullFace (GL_BACK)` sau `glFrontFace(GL_CW)` și `glCullFace (GL_FRONT)`

se elimină fețele ale caror proiecții nu sunt orientate trigonometric.

Algoritmul z-buffer(1)

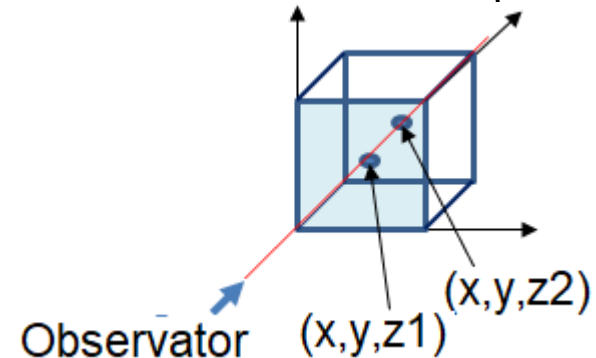
- Algoritm de eliminare a fragmentelor nevizibile ale primitivelor grafice.
- Executat de GPU, fiind integrat in procesul de rasterizare a primitivelor

- ❖ Primitivele rasterizate au coordonatele raportate la spatiul ecran
- ❖ Observatorul este la infinit, pe axa z negativa
- ❖ Asupra primitivelor se efectueaza proiectie ortografica
in planul XOY (poarta de afisare)



Daca doua fragmente aflate pe acelasi proiector, rezultate din rasterizarea a doua primitive diferite, au coordonata z diferita:

- Fragmentul avand coordonata z mai mica va fi afisat în pixelul (x,y).



Algoritmul z-bufer(2)

- Primitivele grafice (triunghiuri, linii) în care a fost descompusă scena 3D sunt rasterizate în ordinea în care au fost transmise din programul de aplicație (nu sunt sortate).
- Buffer-ul imagine (frame buffer) este actualizat pe măsura rasterizării primitivelor, astfel:
 - dacă (fragmentul curent se proiectează în pixelul (x,y) și
 $z_{\text{fragment}} < z_{\text{fragment afișat în pixelul } (x,y)}$)
atunci
actualizează culoarea pixelului (x,y) în bufer-ul imagine, la culoarea fragmentului curent

Rezultă: este necesar să se memoreze coordonatele z ale fragmentelor afișate în pixelii imaginii



Z-buffer: - tablou bidimensional cu număr de elemente egal cu numărul de pixeli ai suprafeței de afișare
- în memoria grafică

Algoritmul z-buffer(3)

Algoritmul z-buffer (integrat in procesul de rasterizare):

- *Initializeaza buffer-ul imagine (frame buffer) la culoarea de fond
- *Initializeaza Z-buffer la coordonata z maxima ($z=1$)

Pentru fiecare fragment $f(x,y,z)$ rezultat din rasterizarea unei primitive

- * calculeaza culoarea fragmentului (in fragment shader)

//testul de vizibilitate (adancime)

daca $z < Z\text{-buffer}[y][x]$ atunci

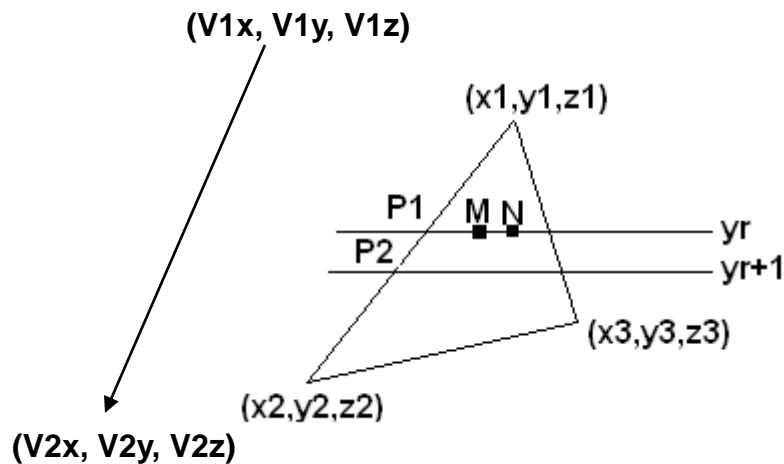
{ $Z\text{-buffer}[y][x] = z$

*actualizeaza culoarea pixelului (x,y) in buffer-ul imagine la culoarea fragmentului f

}

Algoritmul z-bufer(4)

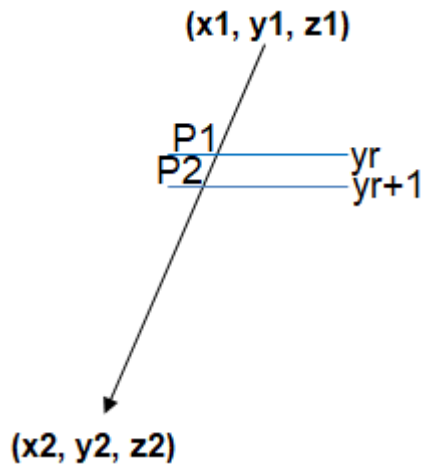
Calculul coordonatei z a unui fragment



Coordonatele z ale fragmentelor rezultate la rasterizarea vectorilor si a suprafetelor triunghiulare se obtin prin calcul incremental, in algoritmul de rasterizare.

Algoritmul z-bufer(5)

Calculul coordonatei z la rasterizarea vectorilor



$$x = x_1 + t(x_2 - x_1)$$

$$y = y_1 + t(y_2 - y_1) \quad \text{Ecuatiile parametrice ale vectorului}$$

$$z = z_1 + t(z_2 - z_1)$$

y_r, y_{r+1} : ordonatele a 2 linii raster (ecran) care intersecteaza vectorul

$$y(P_1) = y_r = y_1 + t_{p_1}(y_2 - y_1) \rightarrow t_{p_1} = (y_r - y_1) / (y_2 - y_1)$$

$$z(P_1) = z_1 + (y_r - y_1)(z_2 - z_1) / (y_2 - y_1) = z_1 + (y_r - y_1) * K,$$

K este o constanta la rasterizarea vectorului

$$z(P_2) = z_1 + (y_{r+1} - y_1) * K = z_1 + (y_r - y_1) * K + K$$

$Z(P_2) = z(P_1) + K \rightarrow$ coordonatele z ale fragmentelor de pe vector se obtin prin calcul incremental.

Algoritmul z-bufer(6)

Calculul coordonatei z la rasterizarea poligoanelor

$P1(xP1, yr, zP1), P2(xP2, yr+1, zP2)$

m- panta laturii $(x1,y1,z1) - (x2,y2,z2)$

$xP2 = xP1 + 1/m$ (vezi curs - rasterizare)

$A*x + B*y + C*z + D = 0$ ec planului poligonului

$zP1 = (-A*xP1 - B*yr - D)/C$

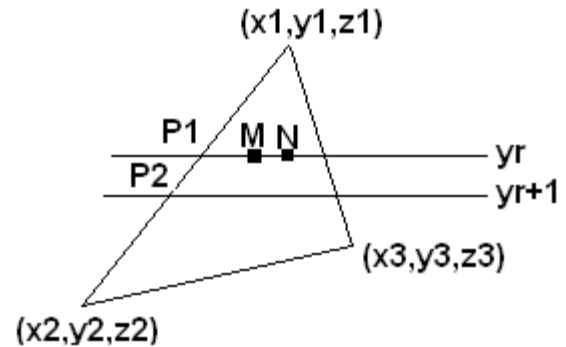
$zP2 = (-A*xP2 - B*(yr + 1) - D)/C = (-A(xP1 + 1/m) - B*yr - B - D)/C = zP1 + (-A/m - B) \rightarrow$

$zP2 = zP1 + K1$

$M(xM, yr, zM), N(xM + 1, yr, zN)$

$zN = (-A*(xM + 1) - B*yr - D)/C = zM + (-A/C) \rightarrow \mathbf{zN = zM + k2}$

Observatie: atunci cand un vector se suprapune pe suprafata unui poligon pot aparea defecte datorita modului diferit in care se calculeaza coordonatele z ale fragmentelor care se afiseaza in aceiasi pixeli!



Algoritmul z-bufer(7)

Aprecieri asupra algoritmului z-Buffer

- 1) Numarul de comparatii de valori z pentru un pixel = numarul de fragmente care se proiecteaza in acel pixel.

Timpul de calcul tinde sa devina independent de numarul de poligoane: in medie, numarul de pixeli acoperiti de un poligon este invers proportional cu numarul de poligoane.

- 2) Bufferul Z este un buffer de numere intregi: pentru o precizie buna, este necesara o reprezentare pe 32 biți a valorilor din z-buffer.
- 3) “Depth fighting”: doua fragmente care se afiseaza in acelasi pixel, cu z diferit in spatiul observator (z-numere reale), au acelasi z in Z-buffer (z- numere intregi) → culoarea pixelului depinde de ordinea rasterizarii primitivelor din care fac parte cele 2 fragmente.
- 4) “Depth complexity”: **numarul de suprascrieri ale unui pixel in buffer-ul imagine**, la generarea unui cadru imagine (incluzand calcul culoare fragment, utilizare texturi)
 - poate fi redusa prin tehnica “deffered rendering”: se calculeaza culoarea numai pentru fragmentele vizibile in pixelii imaginii.