

---

# Компютърна графика

---

Определяне на видими  
стени и обекти

# Права

## ■ Векторно (параметрично) уравнение на права

- правата се задава с
  - радиус-вектор  $\mathbf{P}$
  - направляващ вектор  $\mathbf{d}$
- произволна точка от правата се определя от

$$\mathbf{L} = \mathbf{P} + t\mathbf{d}$$

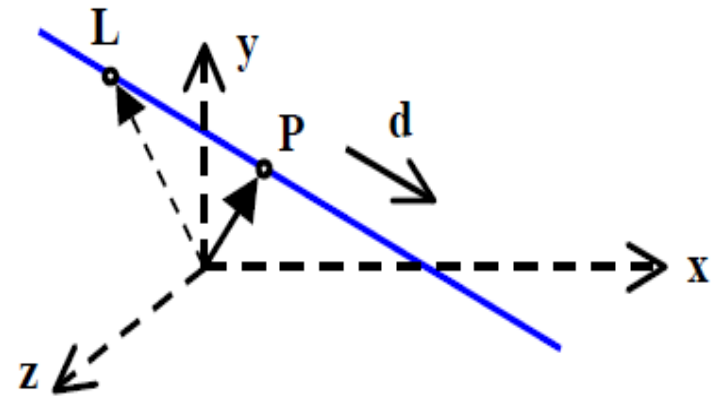
- връзка с декартови координати в 2D

$$x = P_x + t.d_x$$

$$y = P_y + t.d_y$$

- ако  $d_x \neq 0$ , то  $t$  може да се елиминира

$$y = (P_y - (d_y/d_x).P_x) + (d_y/d_x).x \Rightarrow y = a.x + b$$



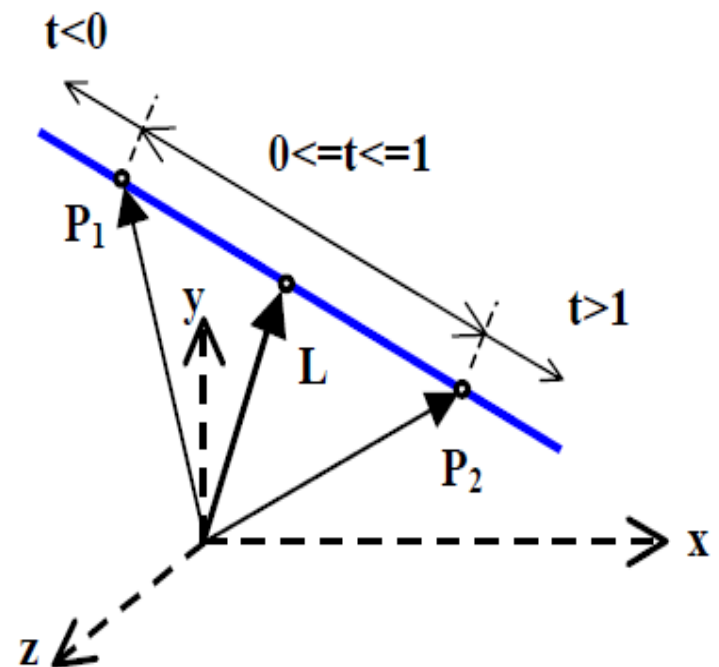
# Прави

- Отсечки от прави
  - отсечката е зададена с две точки  $P_1$  и  $P_2$
- Векторът, определящ посоката на правата, е

$$\mathbf{d} = P_2 - P_1$$

- Уравнението на правата е

$$\mathbf{L} = P_1 + t(P_2 - P_1)$$



# Права

## ■ Точка на пресичане на две отсечки в 2D

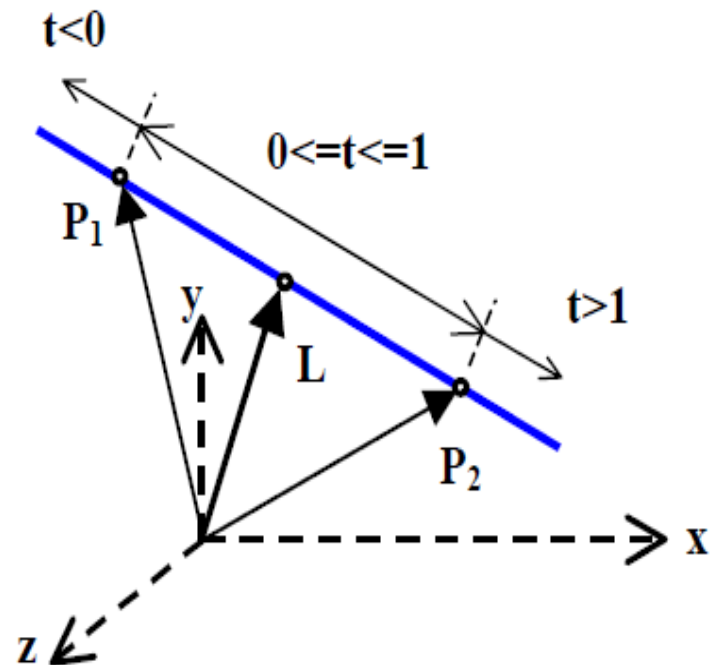
- отсечките са от  $P_{11}$  до  $P_{12}$  и от  $P_{21}$  до  $P_{22}$

$$L_1 = P_{11} + t_1(P_{12} - P_{11})$$

$$L_2 = P_{21} + t_2(P_{22} - P_{21})$$

- в точката на пресичане  $L_1 = L_2$ 
  - за да се определи точката на пресичане се определят  $t_1$  и  $t_2$ 
$$x_{11} + t_1(x_{12} - x_{11}) = x_{21} + t_2(x_{22} - x_{21})$$
$$y_{11} + t_1(y_{12} - y_{11}) = y_{21} + t_2(y_{22} - y_{21})$$

- ако  $0 \leq t_1 \leq 1$  и  $0 \leq t_2 \leq 1$ ,  
то отсечките имат точка на пресичане



# Равнина

## ■ Векторно уравнение на равнина

- равнината е зададена с три точки  $P_1$ ,  $P_2$  и  $P_3$

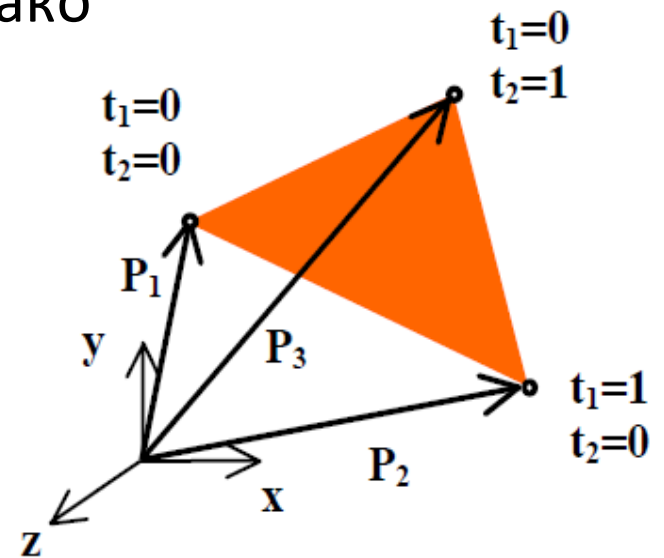
$$\mathbf{P} = P_1 + t_1 (P_2 - P_1) + t_2 (P_3 - P_1)$$

- Точка  $P$  лежи вътре в триъгълника, ако

$$t_1 > 0 \text{ и}$$

$$t_2 > 0 \text{ и}$$

$$t_1 + t_2 < 1$$



# Равнина

- Равнина, зададена с точка и нормала
  - ако  $\mathbf{n}$  е нормала към равнината, а  $\mathbf{p}$  е свободен вектор в равнината, то

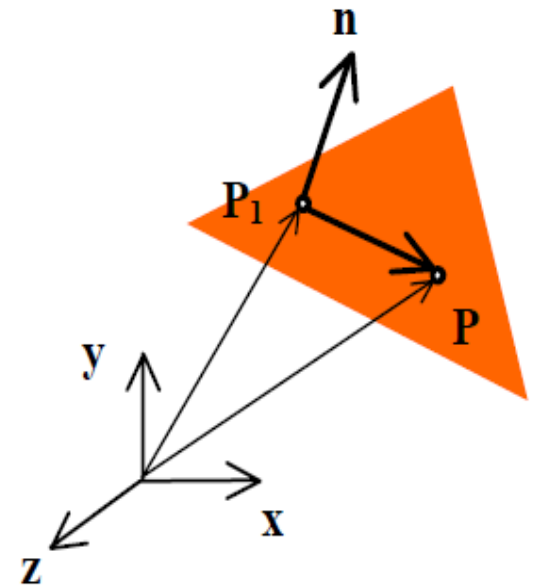
$$\mathbf{n} \cdot \mathbf{p} = 0$$

- Ако равнината е зададена с три точки  $P_1$ ,  $P_2$  и  $P_3$ , то нормала към равнината може да се определи от векторното произведение на двете отсечки

$$\mathbf{n} = (P_2 - P_1) \times (P_3 - P_1)$$

или

$$\mathbf{n} = \|P_2 - P_1\| \cdot \|P_3 - P_1\| \cdot \sin \alpha$$



# Равнина

- Ако равнината е зададена с нормала и точка от равнината (радиус-вектор)  $P_1$ , а  $P$  произволна точка от равнината, то

$$\mathbf{p} = \mathbf{P} - \mathbf{P}_1 \quad \mathbf{n} \cdot (\mathbf{P} - \mathbf{P}_1) = 0 \quad \mathbf{n} \cdot \mathbf{P} = \mathbf{n} \cdot \mathbf{P}_1 = s \quad (s \text{ е скалар})$$

$$\mathbf{n} \cdot \mathbf{P} = s$$

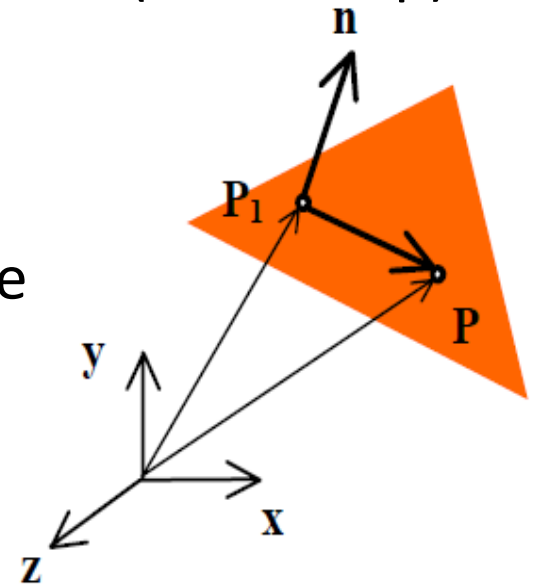
- Преминаване към аналитично уравнение

$$\mathbf{n}_x \cdot x + \mathbf{n}_y \cdot y + \mathbf{n}_z \cdot z - s = 0$$

или

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

където  $A = \mathbf{n}_x$ ,  $B = \mathbf{n}_y$ ,  $C = \mathbf{n}_z$ ,  $D = -\mathbf{n} \cdot \mathbf{P}_1$



# Равнина

## ■ Точка на пресичане на права и равнина

- равнината е зададена с

$$\mathbf{n} \cdot \mathbf{P} = s$$

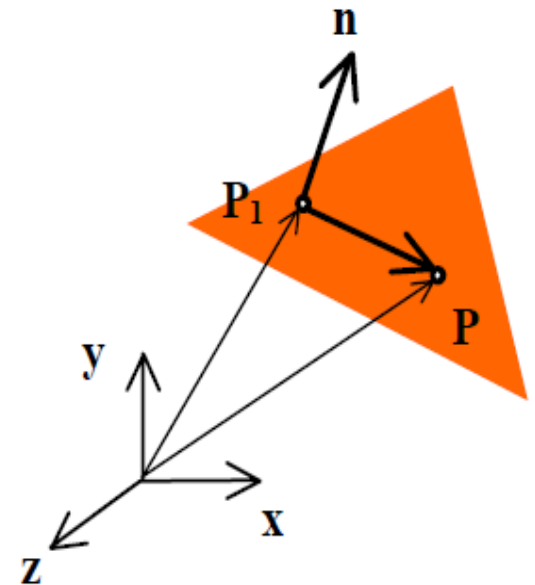
- правата е зададена с

$$\mathbf{P} = \mathbf{P}_1 + t\mathbf{d}$$

## ■ В точката на пресичане

$$\mathbf{n} \cdot (\mathbf{P}_1 + t\mathbf{d}) = s$$

$$t = (s - \mathbf{n} \cdot \mathbf{P}_1) / \mathbf{n} \cdot \mathbf{d}$$





# Равнина

## ■ *Права на пресичане на две равнини*

- двете равнини са зададени с

$$\mathbf{n}_1 \cdot \mathbf{P} = s_1 \quad \text{и} \quad \mathbf{n}_2 \cdot \mathbf{P} = s_2$$

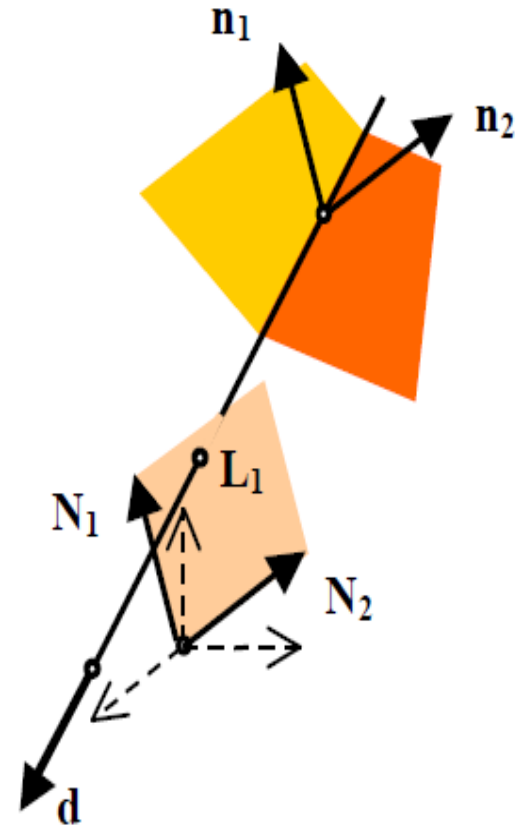
- правата на пресичане е перпендикулярна на  $\mathbf{n}_1$  и на  $\mathbf{n}_2$
- посоката на правата на пресичане се определя от вектора

$$\mathbf{d} = \mathbf{n}_1 \times \mathbf{n}_2$$

- търси се точка от правата, например  $L_1$
- използват се два радиус-вектора, които образуват равнина перпендикулярна на  $\mathbf{d}$  и минаваща през т.О:

$$\mathbf{N}_1 = \mathbf{n}_1, \quad \mathbf{N}_2 = \mathbf{n}_2$$

- точката  $L_1$  лежи в тази равнина



# Равнина

- Уравнението на пресечницата е

$$L = L_1 + t(\mathbf{n}_1 \times \mathbf{n}_2)$$

- $L_1$  може да се представи като

$$L_1 = a \cdot \mathbf{N}_1 + b \cdot \mathbf{N}_2 \quad (a \text{ и } b \text{ са скалари})$$

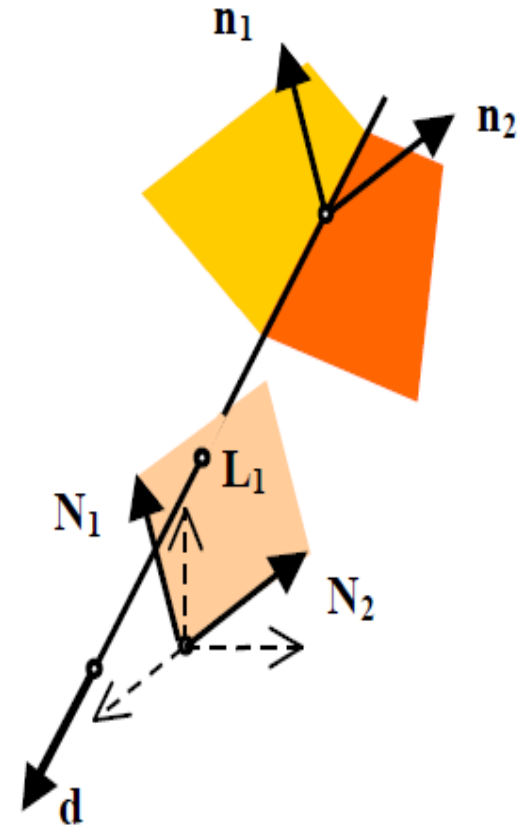
или  $L_1 = a \cdot \mathbf{n}_1 + b \cdot \mathbf{n}_2 \quad / \cdot \mathbf{n}_1 \quad / \cdot \mathbf{n}_2$

- след умножение с двете нормали

$$L_1 \cdot \mathbf{n}_1 = a |\mathbf{n}_1|^2 + b \mathbf{n}_1 \cdot \mathbf{n}_2 = s_1$$

$$L_1 \cdot \mathbf{n}_2 = a \mathbf{n}_1 \cdot \mathbf{n}_2 + b |\mathbf{n}_2|^2 = s_2$$

- решава се спрямо  $a$  и  $b$  и след заместване се определя  $L_1$



# Прави и равнини

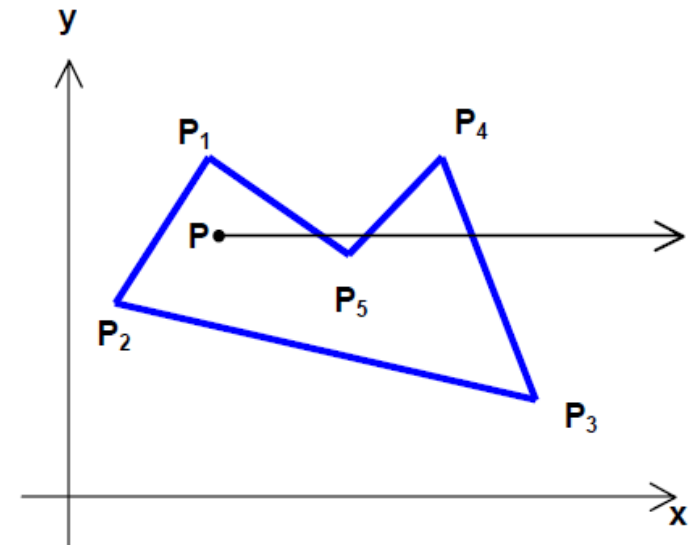
## ■ Тест за принадлежност на точка в произволен многоъгълник

### ■ Дадени са

- многоъгълник с координатите на върховете си

$$P_k \text{ (} k=1, 2, \dots, n, n+1; P_{n+1}=P_1 \text{)}$$

- точка  $P(x_p, y_p)$



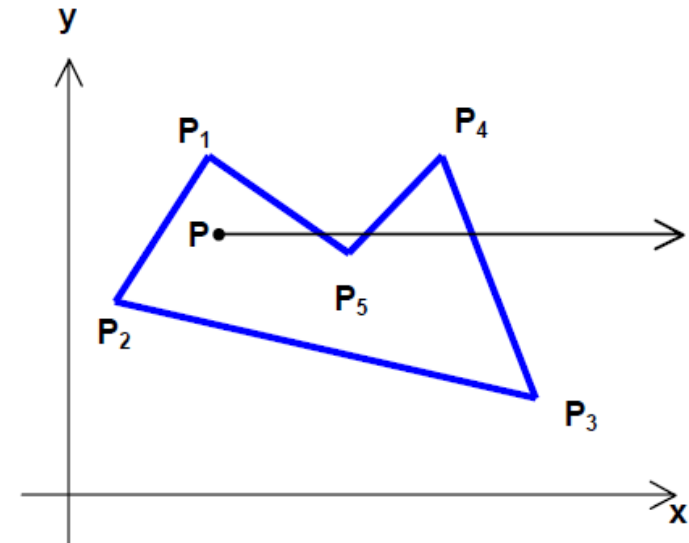
# Прави и равнини

## ■ Тест 1

- Уравнение на лъч през т.Р  
успореден на абсцисата

$$P + t_1 \cdot i$$

- $i$  е единичен вектор по оста  $x$



- **Точката е вътре в многоъгълника ако лъча пресича нечетен брой пъти страните на многоъгълника**
  - за всяка страна се определя има ли пресечна точка с лъча

# Прави и равнини

## ■ Тест 2

- Пренасят се т.Р и многоъгълника в началото на КС

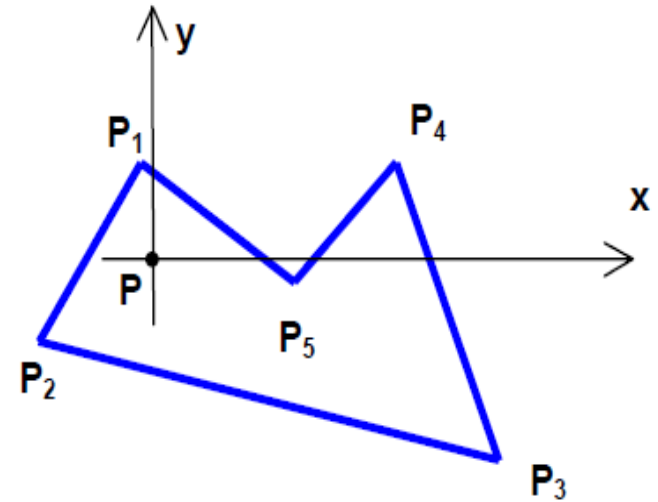
- трансляция с  $T(-x_p, -y_p)$

- **При нечетен брой точки на пресичане на страните на многоъгълника с новата абсциса, точката е вътре в многоъгълника**

- Точка на пресичане на ръб с положителната посока на оста  $x$  има само, ако двете крайни точки на ръба са от различни страни на оста  $x$  и

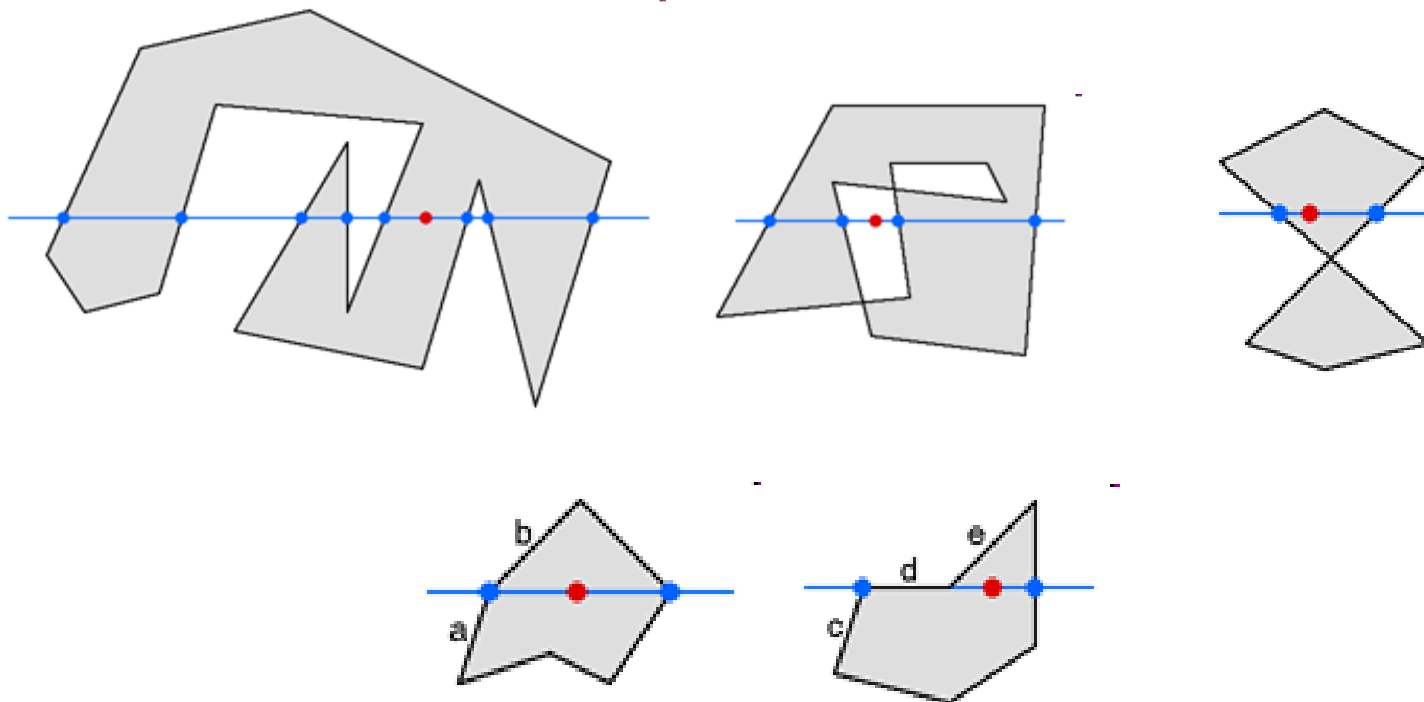
- двете крайни точки са отдясно на оста  $y$  или

- едната крайна точка е отляво, а другата – отдясно на оста  $y$  и точката на пресичане има координата  $x > 0$



# Прави и равнини

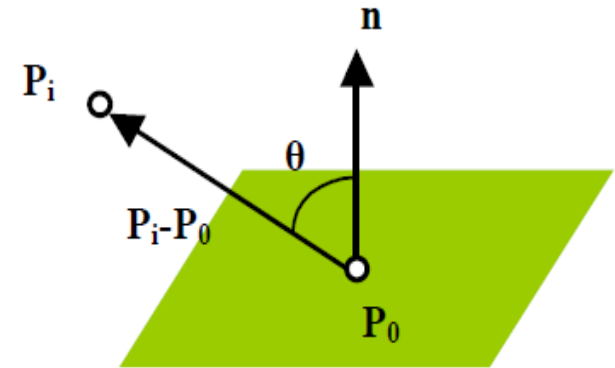
*Тест за принадлежност на точка в произволен  
МНОГОЪГЪЛНИК*



# Прави и равнини

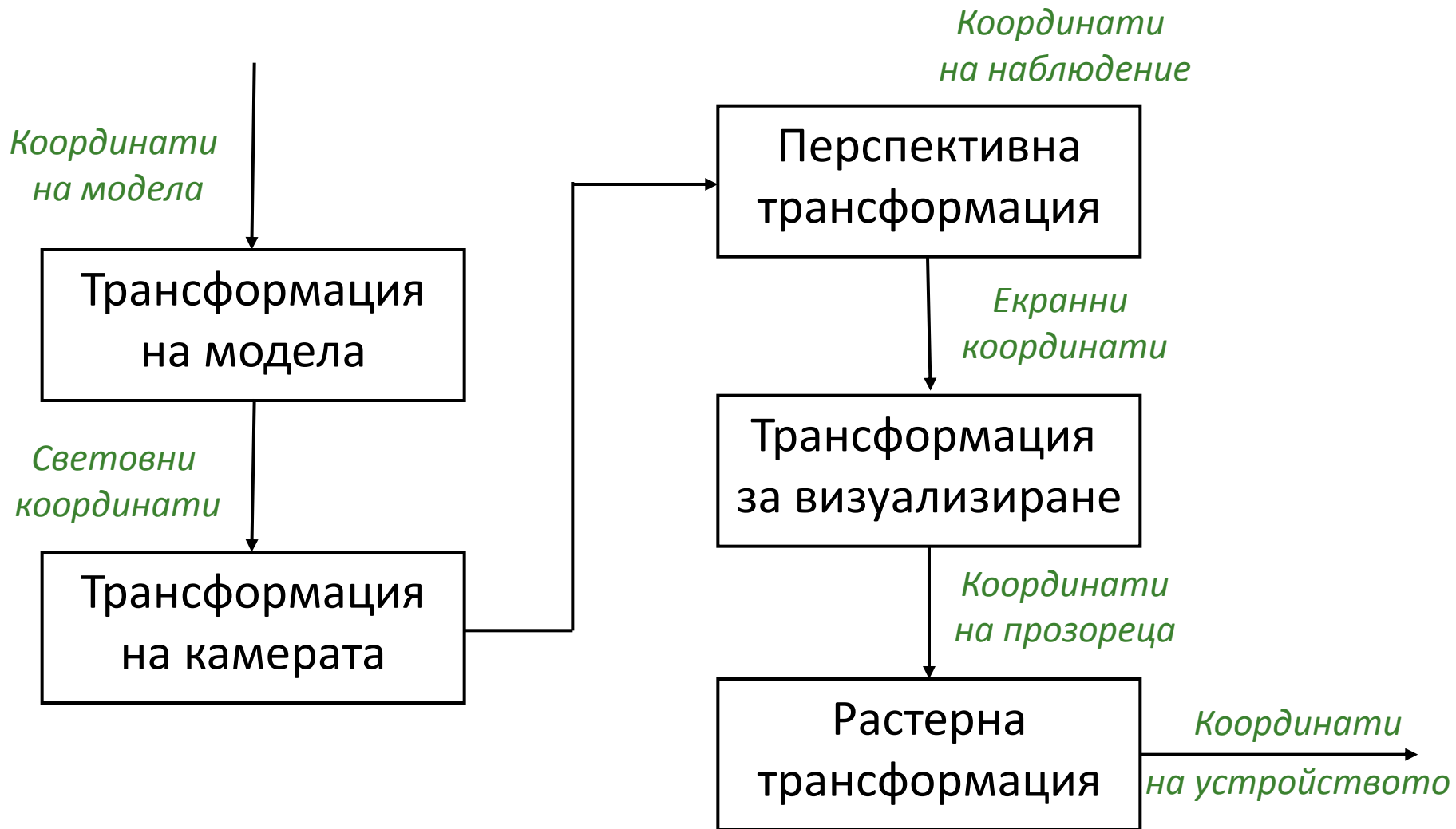
## ■ *Разделяне на пространството от равнина*

- дадена е равнина с точка  $P_0$  и нормала  $n$  и две точки  $P_1$  и  $P_2$
- да се определи дали двете точки лежат от една и съща страна на равнината



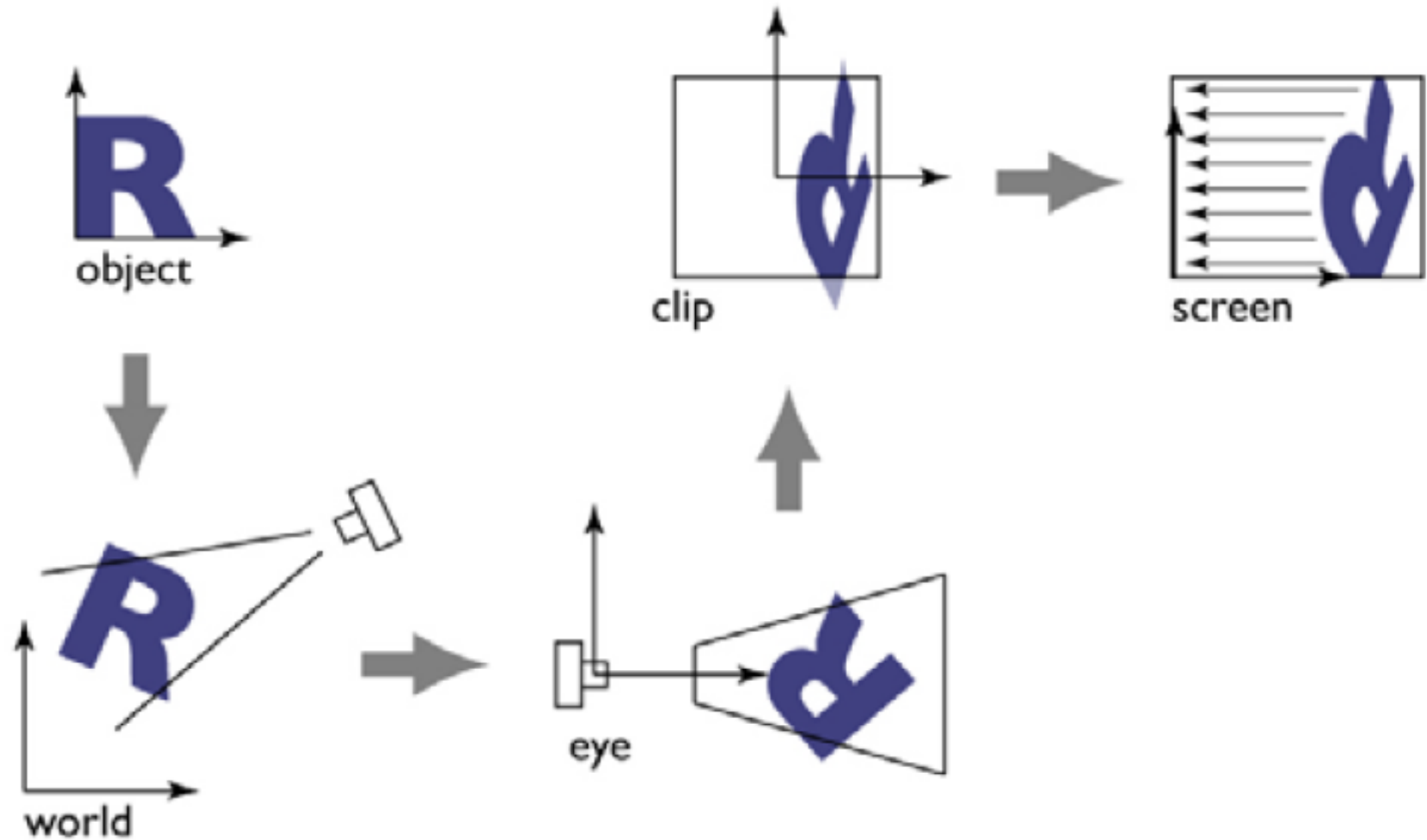
- Използва се знака на скалярно произведение
  - ако  $n \cdot (P_i - P_0) > 0$ , то точката  $P_i$  лежи в това полупространство, в което е нормалата  $n$  ( $P_i \neq P_0$ ,  $P_0$  е точка от равнината)
  - ако  $n \cdot (P_1 - P_0)$  и  $n \cdot (P_2 - P_0)$  имат еднакви знаци, то  $P_1$  и  $P_2$  лежат от една и съща страна на равнината

# Основен графичен конвейер

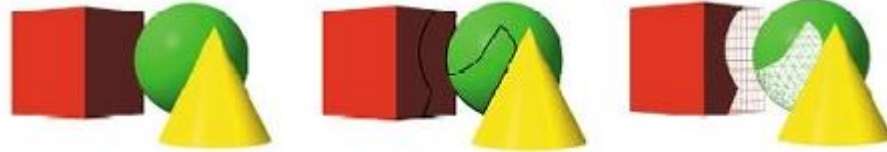




# Основен графичен конвейер

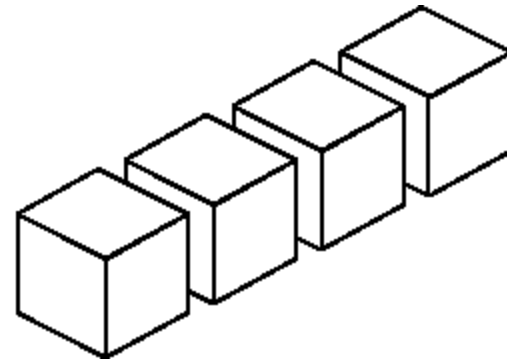
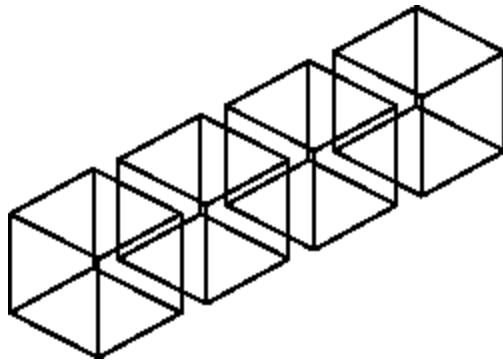


# Видими обекти / скрити обекти



## ■ Цел

- да се определи какво се вижда от дадена сцена при определена позиция на наблюдение
  - не се рендират примитиви, които не са видими
  - отхвърлят се примитиви, които не променят сцената и генерираното изображение



## ■ Кога?

- колкото е възможно по-рано в графичния конвейер

# Видими обекти / скрити обекти

## ■ Термини

### □ *culling*

- отхвърлят се примитиви или групи от примитиви
- *операция за оптимизация на рендирането*

### □ *clipping*

- примитив се разделя на видима и невидима част
- *необходима операция*

### □ *testing*

- отхвърля се фрагмент (потенциален пиксел)
- *необходима операция*

# Видими обекти / скрити обекти

- **Форми на закриване/невидимост**
  - **Изрязване с визуален обем** (View-frustum culling)
    - невидим, защото е извън визуалния обем
  - **Скриване** (Backface culling)
    - невидим, защото е вътре в затворена фигура
  - **Закриване** (Occlusion culling)
    - невидим, защото е закрит
  - **Незначителност** (Importance culling)
    - (почти) невидим, защото проекцията му е твърде малка в сравнение с останалите елементи

# Видими обекти / скрити обекти

- **Форми на закриване/невидимост**
  - ***Хардуерна имплементация (GPU)***
    - автоматизирано
    - много ефективно (много малко забавяне)
    - изпълнява се късно в графичния конвейер и разглежда само един примитив в даден момент
  - ***Софтуерна имплементация (CPU)***
    - алгоритми и структури от данни за изпълнението
    - по-малко ефективно
    - изпълнява се рано в графичния конвейер и разглежда цели групи примитиви

# Видими обекти / скрити обекти

- **Форми на закриване/невидимост**
  - **Умерени (*Conservative*)**
    - примитив се отхвърля само ако е сигурно че трябва да се отхвърли
    - понякога се визуализира обект, който не е видим
  - **Не умерени (*Non Conservative*)**
    - примитивите се отхвърлят според евристики
    - понякога НЕ се визуализира (частично) обект!

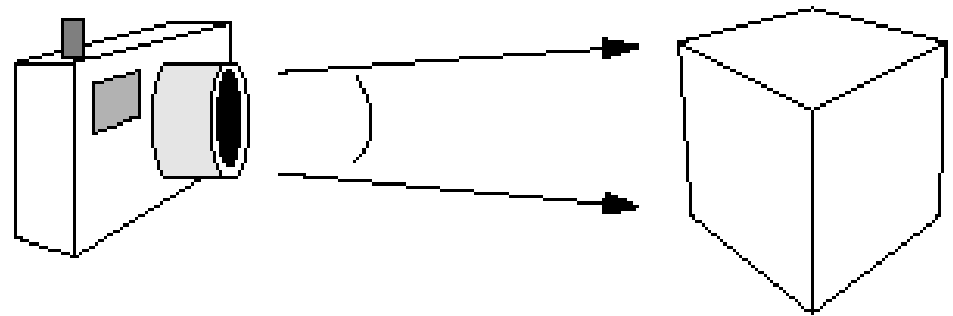
# Видими обекти / скрити обекти

- Две основни групи алгоритми за откриване на видими повърхности
  - **Методи в пространството на обектите**  
*(Object Space Methods)*
    - сравняват се обекти и части от обекти в сцената за да се определи кои са видими
  - **Методи в пространството на изображението**  
*(Image Space Methods)*
    - видимостта се определя точка по точка за всеки пиксел в проекционната равнина

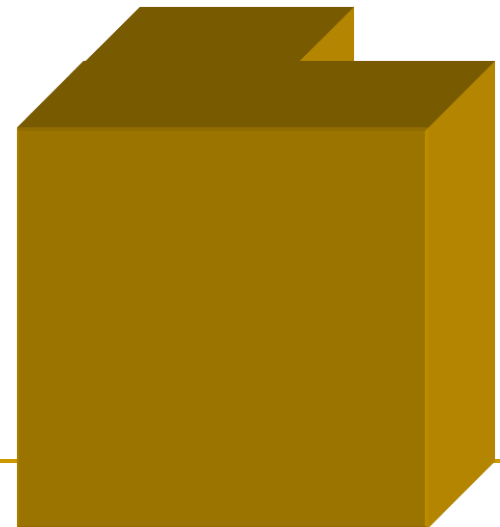
# Скрити повърхности

- Отстраняване на невидими стени на изпъкнали тела

- *Backface culling*
- *Back-Face Detection*
- *Self-Hidden Surfaces*



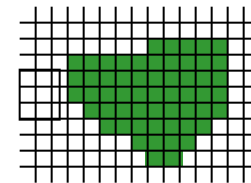
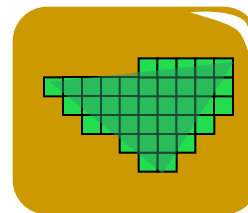
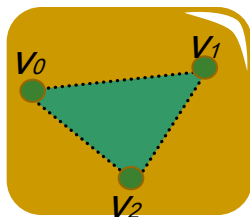
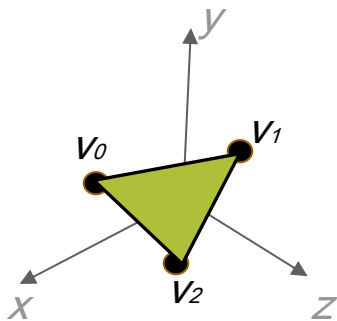
- Хардуерна имплементация
- Софтуерна имплементация





# Скрити повърхности

## ■ Хардуерна имплементация



само след тази стъпка!



Възли  
(Vertices)  
(точки в  $R^3$ )

изчисления с  
възли

Проектирани  
възли  
(точки в  $R^2$ )

set-  
up

Растрериране  
на точки

set-  
up

Растрериране на  
триъгълници

set-  
up

Растрериране  
на сегменти

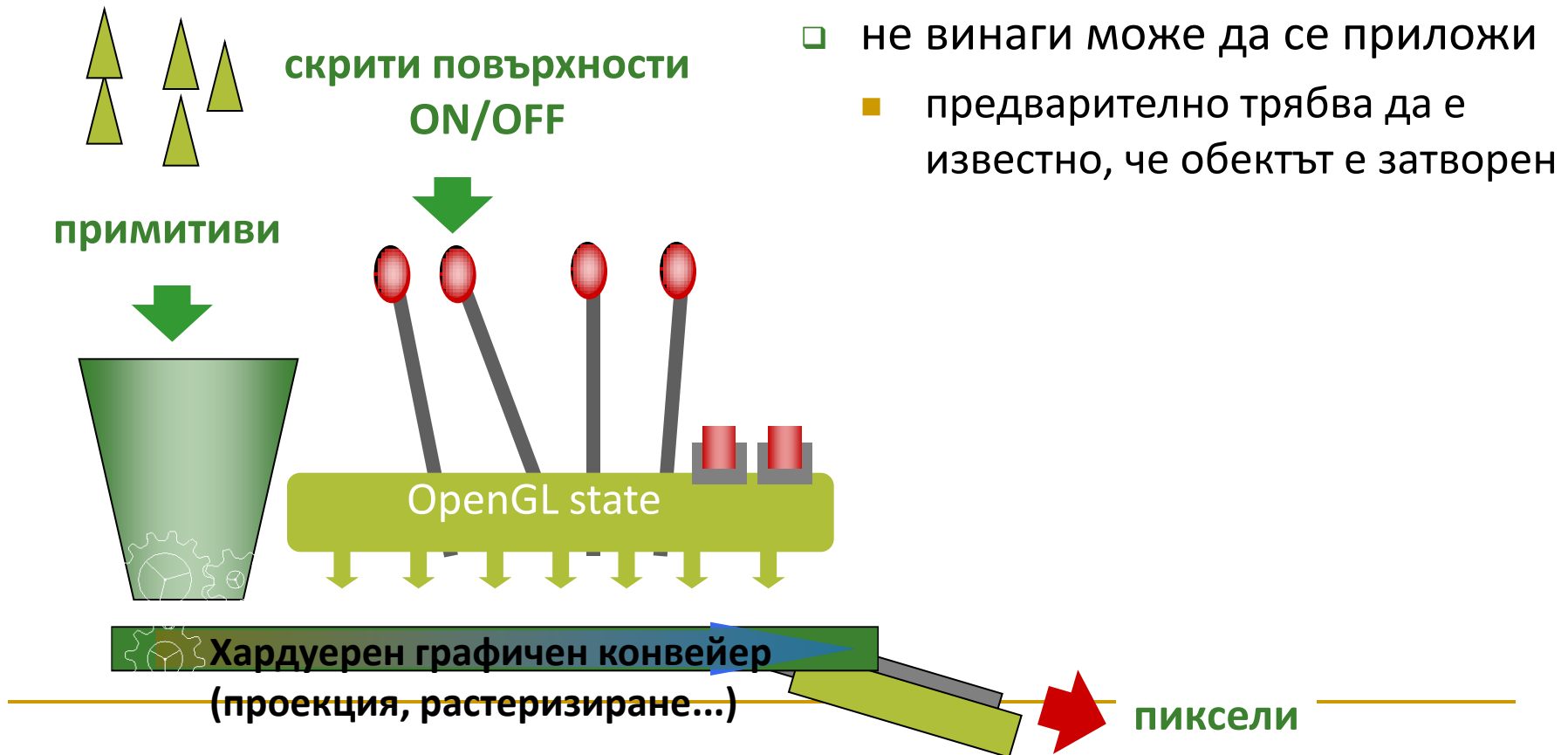
фрагменти  
(кандидат пиксели)

изчисления с  
фрагменти

Пиксели  
(Frame buffer)

# Скрити повърхности

## ■ Хардуерна имплементация



# Скрити повърхности

## ■ Хардуерна имплементация

### □ OpenGL state

- функции за промяна на състоянието

```
glEnable (GL_CULL_FACE) ;  
glDisable (GL_CULL_FACE) ;
```

- променят само състоянието, но не екрана

- функции за определяне дали се отхвърлят предни или задни скрити стени

```
glCullFace (GL_FRONT) ;  
glCullFace (GL_BACK) ;
```

- **недостатък: работи само за изпъкнали обекти**

# Скрити повърхности

## ■ *Софтуерна имплементация*

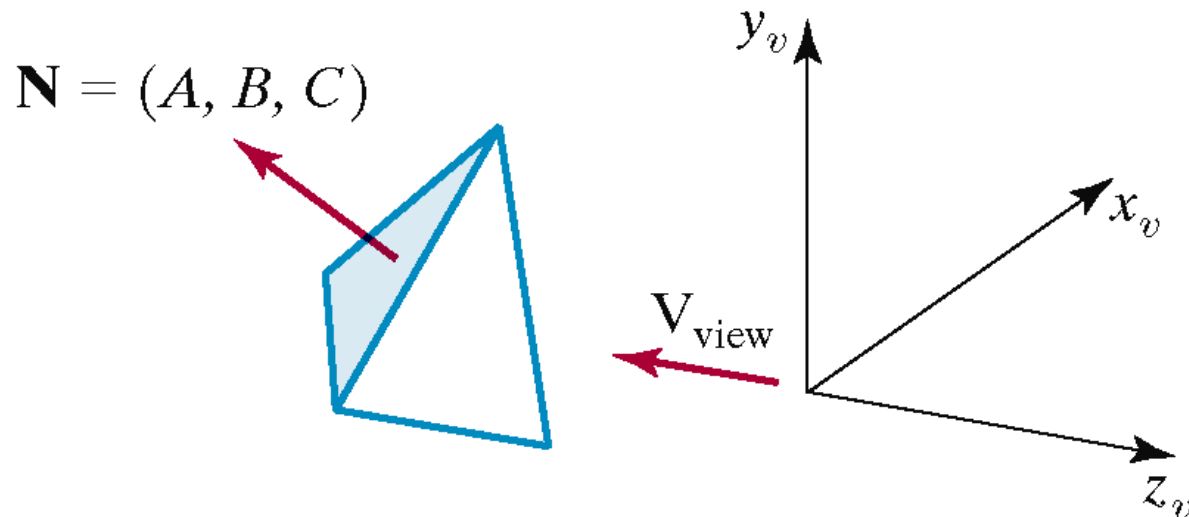
- определяне и отхвърляне на задната стена на твърди тела, които са изпъкнали многостени
- най-простият подход
  - отстраняват се от по-нататъшно разглеждане част от стените на обекти в сцената
- Дадена е т.  $P(x, y, z)$
- Даден е многоъгълник в равнина, определена с параметрите  $A, B, C, D$
- Точката е зад равнината на многоъгълника ако

$$Ax + By + Cz + D < 0$$

# Скрити повърхности

## ■ Софтуерна имплементация

- Нека е дадена дясно ориентирана координатна система с посока на наблюдение по отрицателната посока на оста  $z$
- Ако  $z$  компонента на нормалата на многоъгълника е по-малка от 0, то повърхнината не се вижда



# Скрити повърхности

## ■ Софтуерна имплементация

### ■ Вектор на наблюдение

- при перспективна проекция с център  $C$

$$\mathbf{V} = \mathbf{C} - \mathbf{P}$$

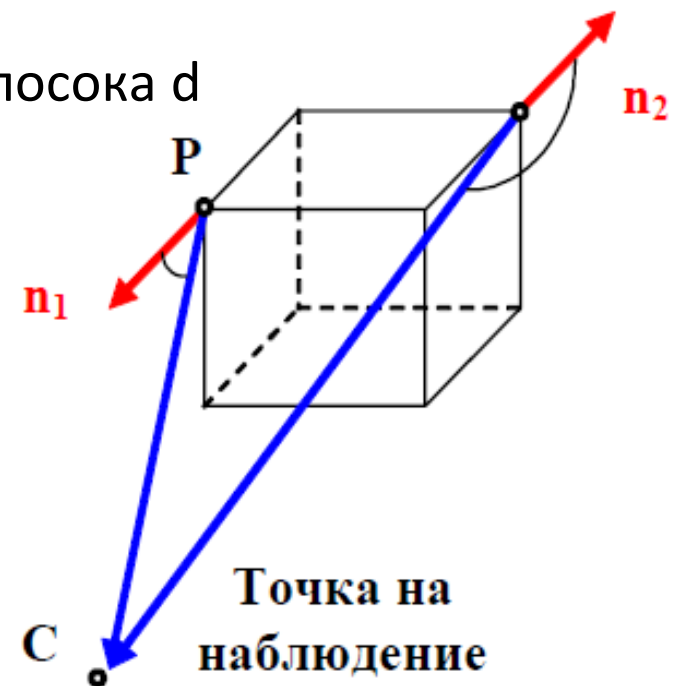
- при паралелна проекция, зададена с посока  $d$

$$\mathbf{V} = \mathbf{P} + t \cdot \mathbf{d}$$

### ■ Ако

$$\mathbf{n} \cdot \mathbf{V} > 0$$

то стената е видима



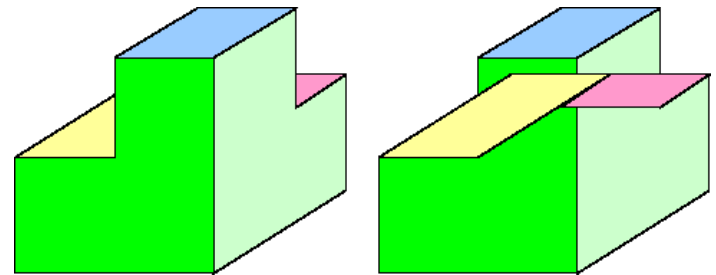
# Скрити повърхности

## ■ *Софтуерна имплементация*

- обикновено се елиминират около половината от стените на многоъгълниците от по-нататъшни тестове на видимостта

## ■ Необходими са други подходи за по-сложните ситуации

- само-закриване
- частично закриване на един обект от друг

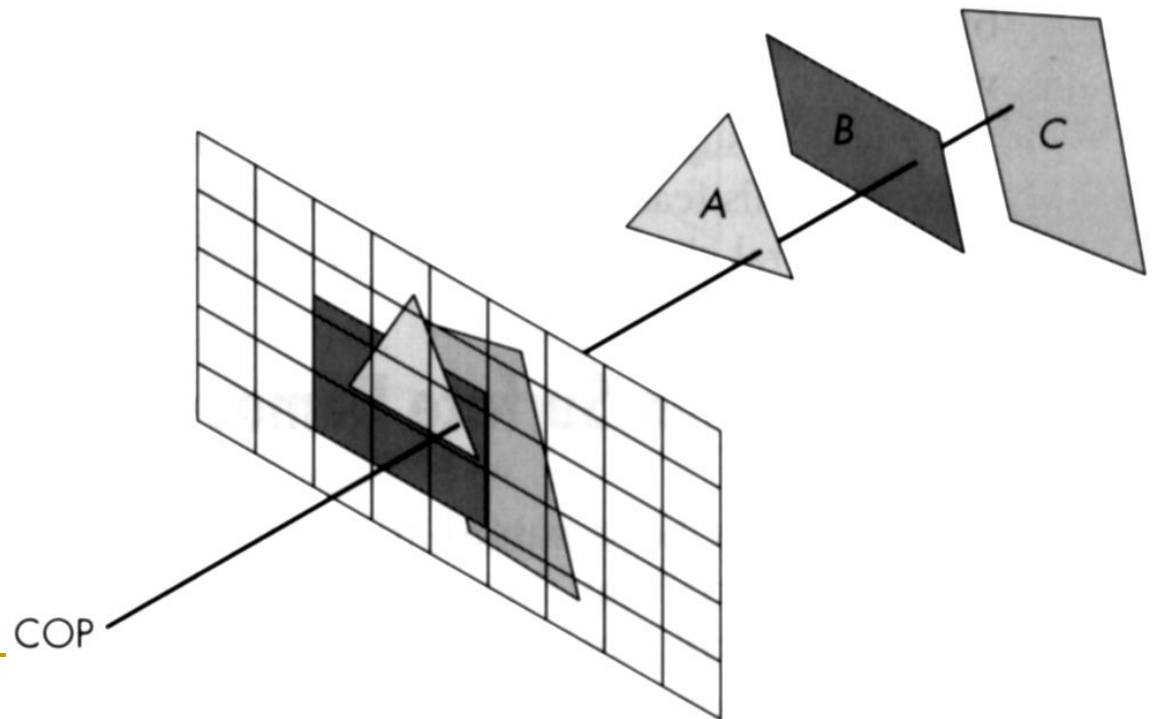


# Закрити повърхности

## ■ *Occlusion culling*

- много важна форма на определяне на невидимост
- могат да бъдат отхвърлени голяма част от примитивите

- по-близо разположените до камерата обекти закриват по-отдалечените



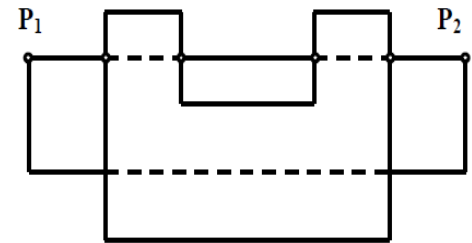


# Закрити повърхности

- Два подхода
  - премахване на цели стени
    - триъгълници, групи от триъгълници, полигони
  - премахване на части от стени
- Историческо развитие на методите
  - *Премахване на скрити линии*
  - *Алгоритъм на художника*
  - *Метод на z-буфер*

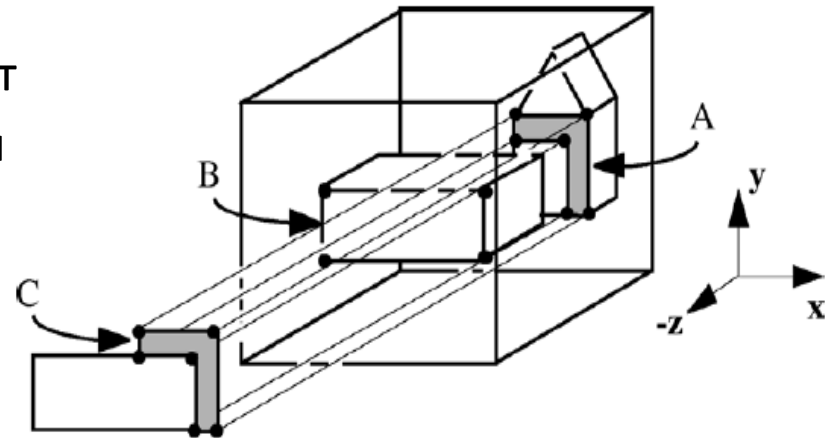
# Закрити повърхности

- **Премахване на скрити линии** (Roberts, 1963)
  - сравнява се всеки ръб във всеки обект и се елиминират невидимите ръбове или части от ръбове
    - стени, които скриват само части от ръб го делят на няколко изцяло видими или изцяло невидими отсечки



# Закрити повърхности

- **Премахване на скрити линии** (Roberts, 1963)
  - подобен подход се прилага и за скрити стени
    - всеки многоъгълник се изрязва с проекцията на всички останали многоъгълници, разположени пред него
    - елиминират се невидимите части от стената и се създават нови видими многоъгълници
  - недостатък
    - сложността е  $O(n^2)$ 
      - всеки обект се сравнява с всички ръбове



Polygon A is clipped by B which is in front of it. A new sub-polygon, C, is created.

# Закрити повърхности

## ■ Алгоритъм на художника

- визуализират се цели стени, но се изчертават в *правилен ред*
- създава се ред за визуализиране на стените
  - всеки многоъгълник се наслагва върху предишните
  - гарантира се коректно визуализиране при всякаква разделителна способност
- многоъгълниците се сортират по *дълбочина* на разположението им
  - от най-отдалечения към най-близкия до позицията на камерата



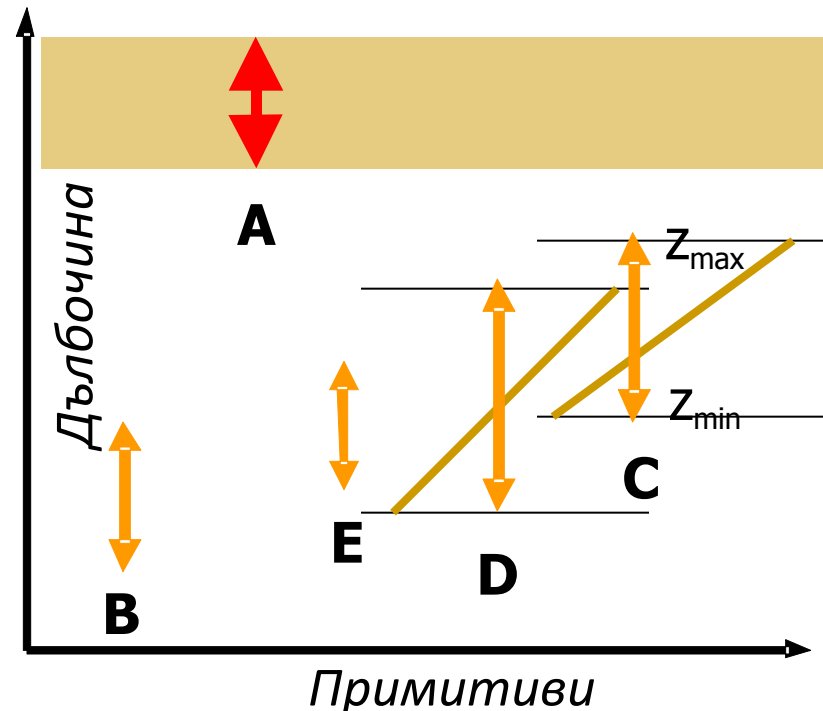
# Откриване на видими повърхности

## ■ Алгоритъм на художника

- 1) Отстраняват се задните стени
- 2) Стените се сортират в намаляващ ред на z координатите им
- 3) Всяка стена от сортирания списък се проектира на екрана и се рендира като запълнен многоъгълник

## □ Проблеми

- примитивите имат max и min дълбочина
- съществува ли “правилен” ред?
- първо C или първо D?



# Откриване на видими повърхности

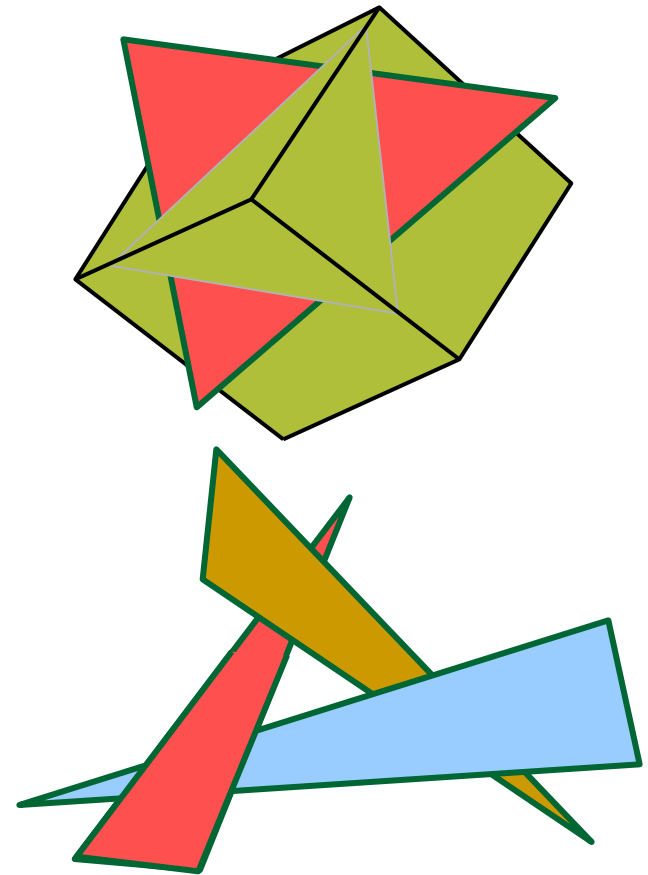
## ■ Алгоритъм на художника

### □ Проблеми

- взаимно закриващи се обекти

### □ Недостатъци

- изчислителна сложност
  - сортиране  $\rightarrow (n \log (n))$ 
    - в компютърната графика повече от линейна сложност води до проблеми
- само софтуерна имплементация! (CPU)
  - не се интегрира в хардуерния конвейер
- не се паралелизира

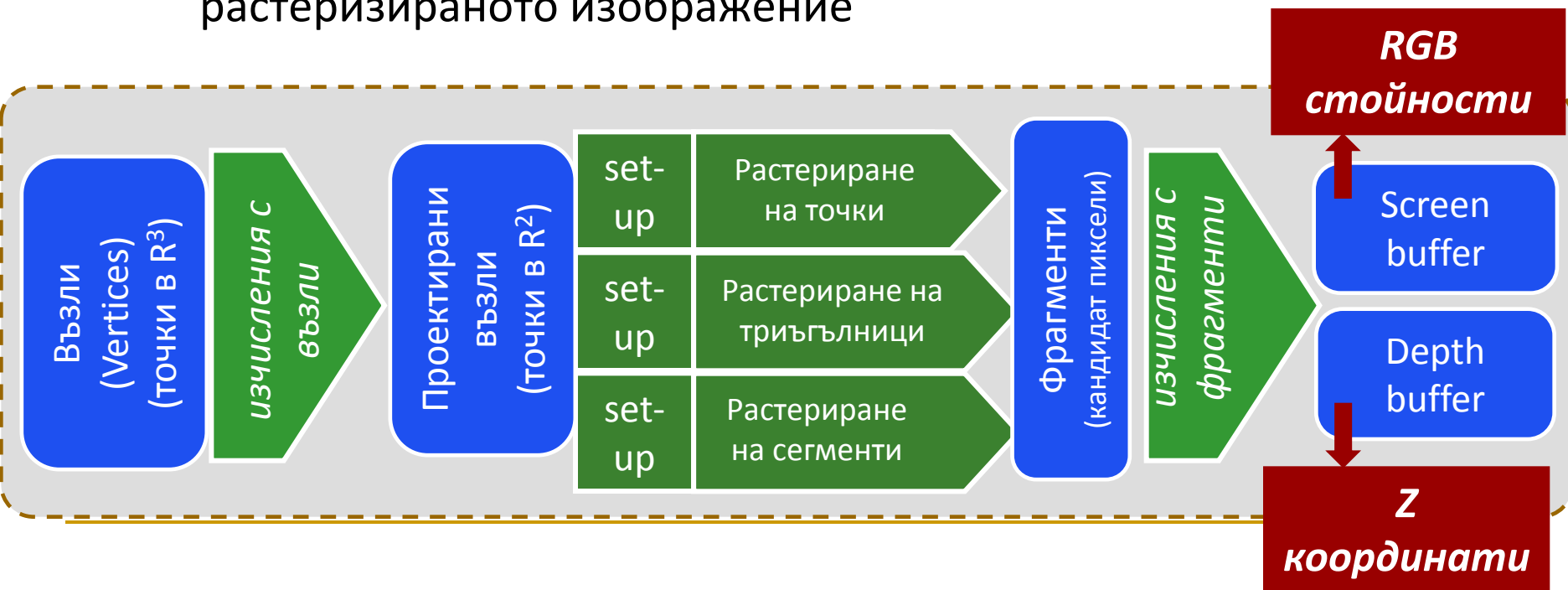


# Метод на z-буфер

## ■ Алгоритъм на z-буфер

### □ *Depth-Buffer Method*

- Сравнява стойностите на z-координатите за всеки пиксел в растеризираното изображение

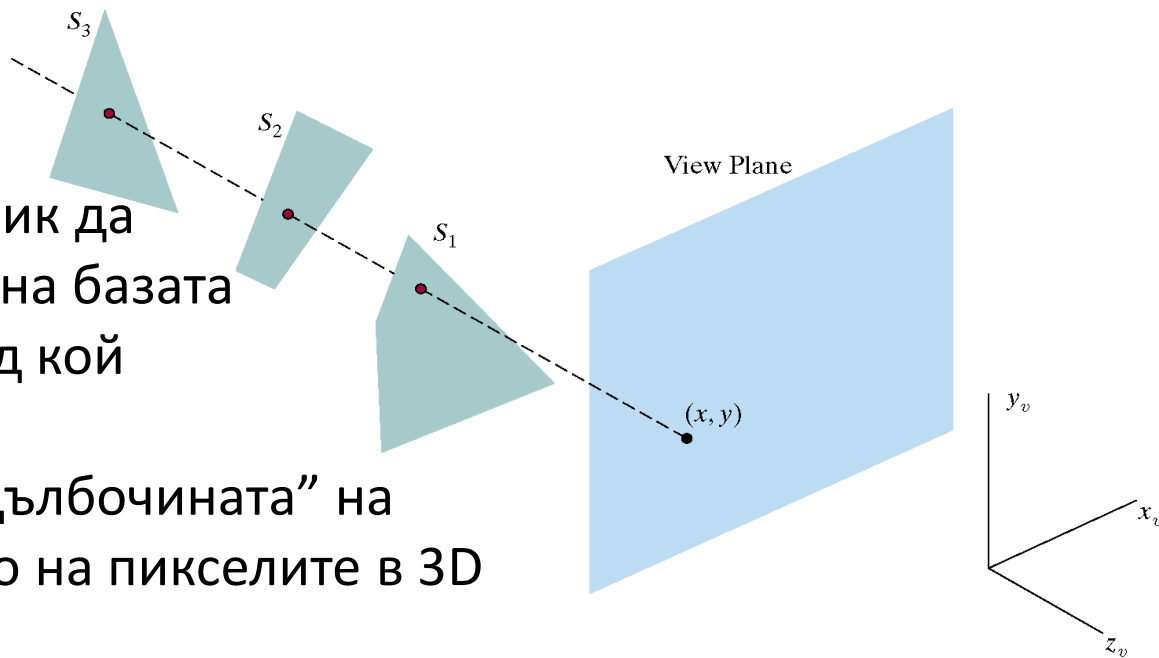


# Метод на z-буфер

## ■ Алгоритъм на z-буфер

### □ Цел

- да се определи кой многоъгълник да се визуализира на базата на това кой пред кой е разположен
- анализира се “дълбочината” на разположението на пикселите в 3D пространството
- z-координатите се изчисляват лесно
  - алгоритъмът е бърз





# Метод на z-буфер

- За всеки многоъгълник от сцената
  - растеризира се многоъгълника
  - определя се z-координата за всеки пиксел от екрана в рамката за визуализиране
  - ако z-координатата е по-голяма (т.е. обекта е по-близко до позицията на наблюдение), то се заменя цвета на пиксела с новата стойност за многоъгълника
- Обикновено се прилага за сцени, които съдържат само многоъгълници

# Метод на z-буфер

- За да се приложи алгоритъм със z-буфер се използват
  - frame buffer (screen buffer)
    - съдържа стойностите с цвета за всеки пиксел
  - z-buffer
    - съдържа текущата стойност на z-координатата за всеки пиксел
- Двата буфера имат еднакви размери
- Не се изисква анализ на пресичането на обекти един с друг
- Не се изисква сортиране на обектите
  - реда на обработването на стените няма значение
- Алгоритъмът изисква допълнителна памет за z-буфера

# Метод на z-буфер

## ■ Алгоритъм на z-буфер

### 1. Инициализация на z-буфер и frame буфер

- за всеки пиксел  $(x, y)$  от рамката на екрана

`frameBuff(x, y) = backgroundColor`

`depthBuff(x, y) = 1.0`

# Метод на z-буфер

## ■ Алгоритъм на z-буфер

2. Обработка се всеки многоъгълник в сцената един по един

- проектира се многоъгълника на екрана
- за всеки проектиран възел от многоъгълника с координати  $(x, y)$  за изчислява дълбочината  $z$  (ако не е известна)
- Ако  $z < \text{depthBuff}(x, y)$  то се определя цвета на многоъгълника за тази позиция и се задават нови стойности

$$\text{depthBuff}(x, y) = z$$

$$\text{frameBuff}(x, y) = \text{surfColour}(x, y)$$

# Метод на z-буфер

- **Пример** (стойността за всеки пиксел е z-координатата)

64	64	64	64	64	64	64	64	64	
64	64	64	64	64	64	64	64	64	
64	64	64	64	64	64	64	64	64	
64	64	64	64	64	64	64	64	64	
64	64	64	64	64	64	64	64	64	
64	64	64	64	64	64	64	64	64	
64	64	64	64	64	64	64	64	64	
64	64	64	64	64	64	64	64	64	
64	64	64	64	64	64	64	64	64	
64	64	64	64	64	64	64	64	64	

+

5	5	5	5	5	5	5			
5	5	5	5	5	5				
5	5	5	5	5					
5	5	5	5						
5	5	5							
5	5								
5									

=

5	5	5	5	5	5	5	64		
5	5	5	5	5	5	64	64		
5	5	5	5	5	64	64	64		
5	5	5	5	64	64	64	64		
5	5	5	64	64	64	64	64		
5	5	64	64	64	64	64	64		
5	64	64	64	64	64	64	64		
64	64	64	64	64	64	64	64		
64	64	64	64	64	64	64	64		
64	64	64	64	64	64	64	64		

5	5	5	5	5	5	5	64		
5	5	5	5	5	5	64	64		
5	5	5	5	5	64	64	64		
5	5	5	5	64	64	64	64		
5	5	5	64	64	64	64	64		
5	5	64	64	64	64	64	64		
5	64	64	64	64	64	64	64		
64	64	64	64	64	64	64	64		
64	64	64	64	64	64	64	64		
64	64	64	64	64	64	64	64		

+

7									
6	7								
5	6	7							
4	5	6	7						
3	4	5	6	7					
2	3	4	5	6	7				

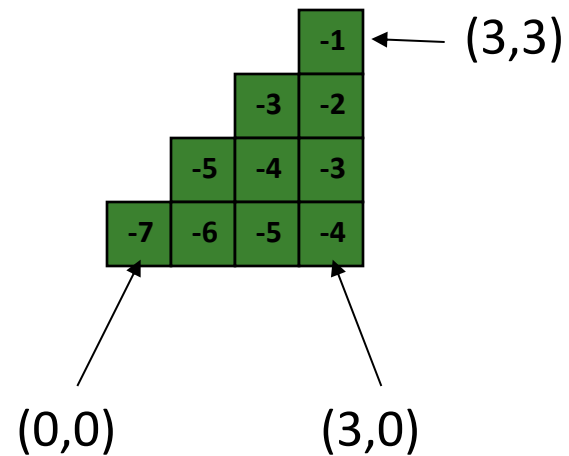
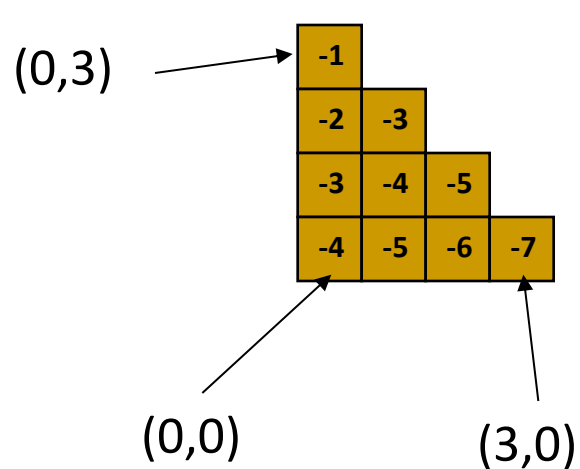
=

5	5	5	5	5	5	5	64		
5	5	5	5	5	5	64	64		
5	5	5	5	5	64	64	64		
5	5	5	5	64	64	64	64		
4	5	5	7	64	64	64	64		
3	4	5	6	7	64	64	64		
2	3	4	5	6	7	64	64		
64	64	64	64	64	64	64	64		
64	64	64	64	64	64	64	64		
64	64	64	64	64	64	64	64		

# Метод на z-буфер

## ■ Пример

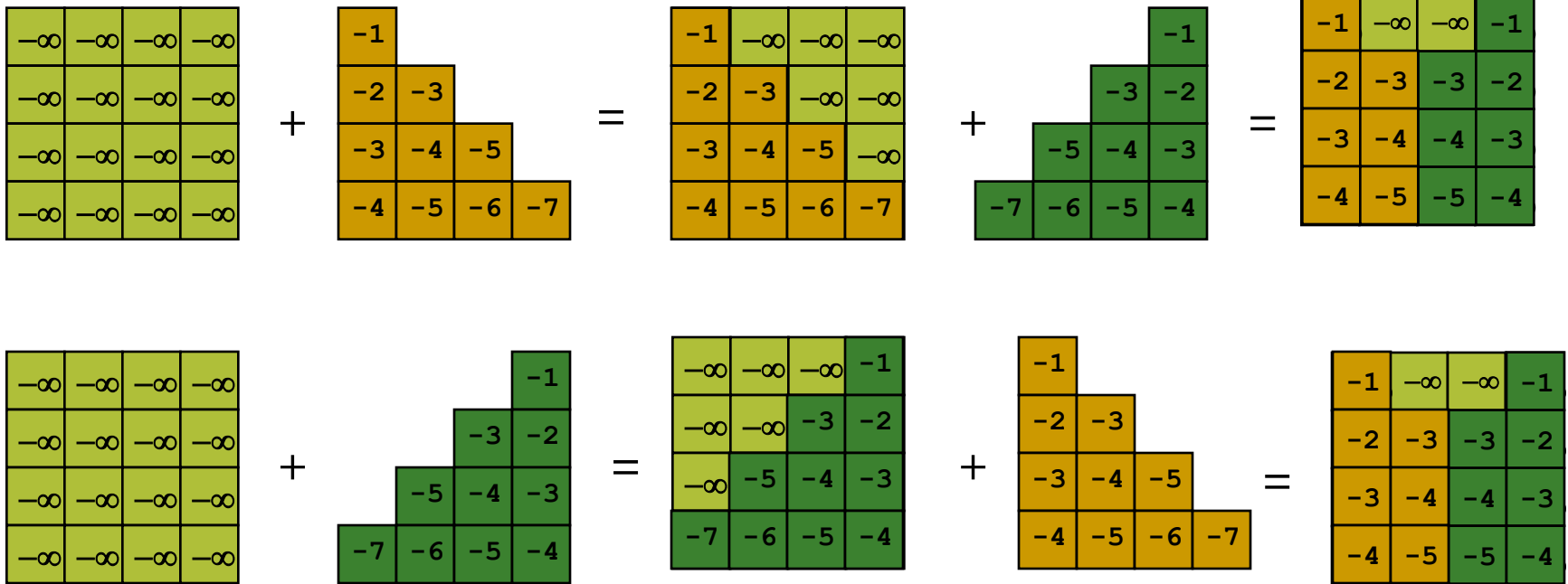
- Има ли значение реда на обработване?



# Метод на z-буфер

## ■ Пример

- Има ли значение реда на обработване?



# Метод на z-буфер

```
allocate z-buffer;

// Initialize color and depth
for each pixel (x,y)
    color [x][y] = backgroundColor;
    zBuffer[x][y] = farPlane;

// Draw polygons (in any order)
for each polygon p
    for each pixel (x,y) in p // Rasterize polygon
         $p_z$  = p's z-value at (x,y); // Interpolate polygon's z-value
        if  $p_z > zBuffer[x][y]$  // If new depth is closer:
            color[x][y] = newColor; // Write new color & new
depth
            zBuffer[x][y] =  $p_z$ ;
```



# Метод на z-буфер

## ■ *Използване на depth buffer в OpenGL*

### 1. Инициализиране на буфер

```
glutInitDisplayMode(GLUT_DEPTH)
```

### 2. Разрешаване на depth buffering

```
glEnable(GL_DEPTH_TEST)
```

### 3. Изтриване на depth buffer

□ когато се изтрива и color buffer в callback функцията display

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
```

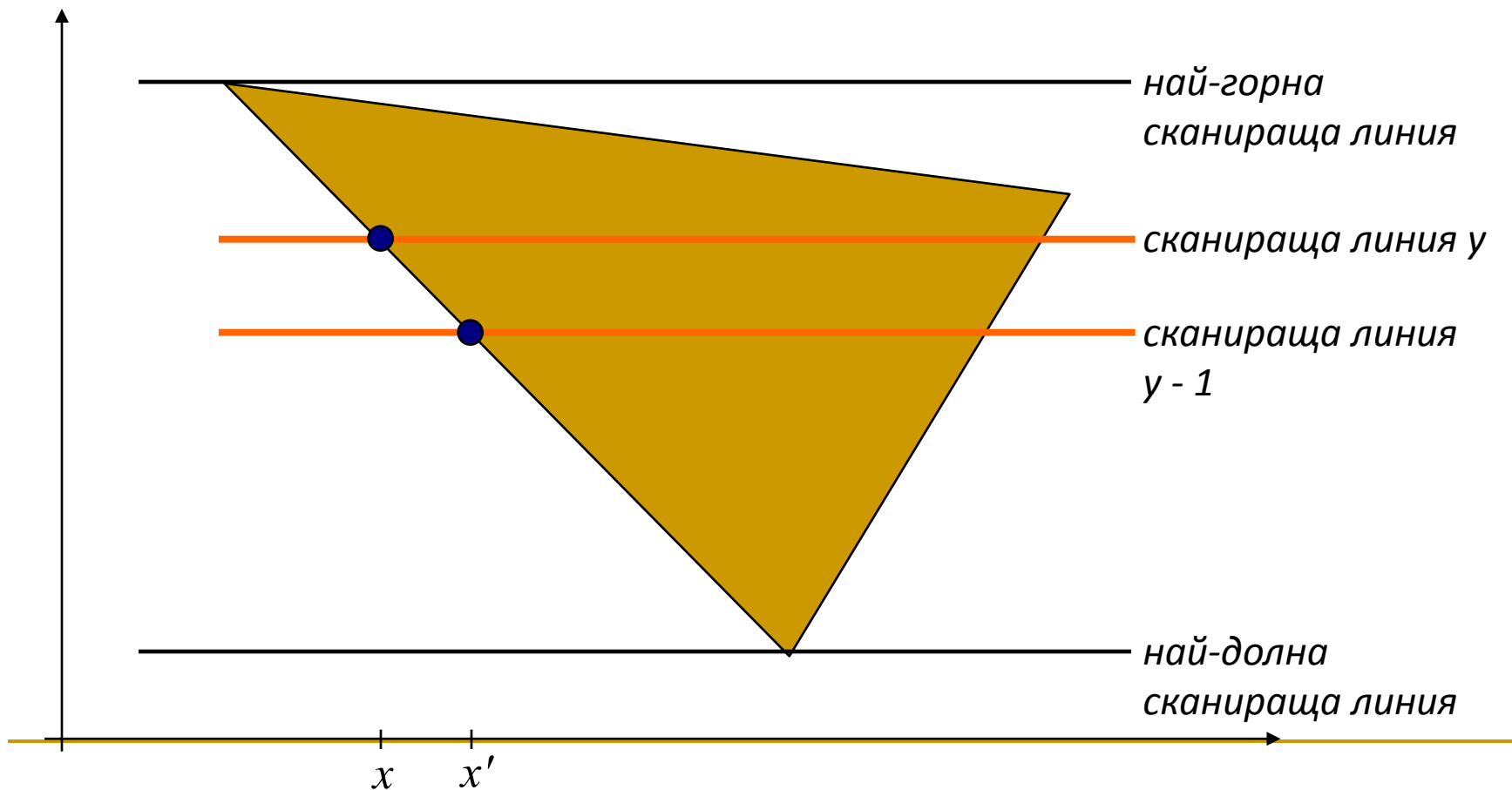
# Метод на z-буфер

## ■ *Изчисляване на дълбочина*

- Итеративни изчисления със сканираща линия
  - за всеки многоъгълник в сцената най-напред се разглежда най-горния възел
  - след това рекурсивно се изчисляват стойностите на x-координатите надолу по левия ръб на полигона

# Метод на z-буфер

## ■ Изчисляване на дълбочина



# Метод на z-буфер

## ■ Изчисляване на дълбочина

- За всяка точка от многоъгълника z-координатата се определя от уравнението на равнината

$$z = \frac{-Ax - By - D}{C}$$

- За всяка сканираща линия съседните x-координати и съседните y-координати се различават с  $\pm 1$

$$z' = \frac{-A(x+1) - By - D}{C}$$

$$z' = z - \frac{A}{C}$$

# Метод на z-буфер

## ■ Изчисляване на дълбочина

- Стойностите на x-координатата на началната точка от всяка сканираща линия се определя от предишната

$$x' = x - \frac{1}{m}$$

- където  $m$  е наклона

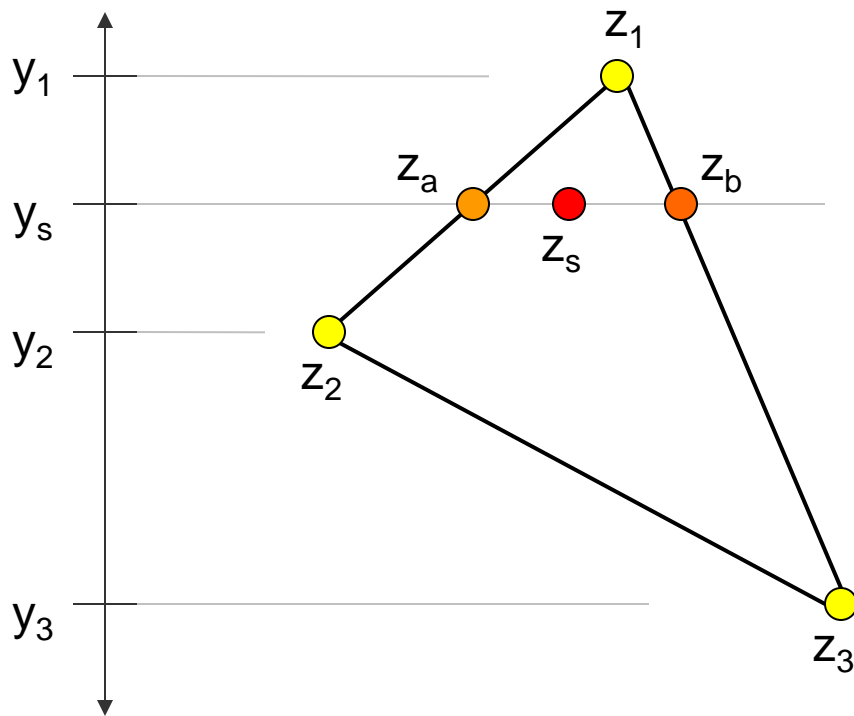
- Стойностите на z-координатата по ръба се изчисляват с интерполация

$$z' = z - \frac{A/m + B}{C}$$

- ако ръбът е вертикален:  $m = \infty$ , то  $z' = z + \frac{B}{C}$

# Метод на z-буфер

## ■ Изчисляване на дълбочина



$$z_a = z_1 + (z_2 - z_1) \frac{y_1 - y_s}{y_1 - y_2}$$

$$z_b = z_1 + (z_3 - z_1) \frac{y_1 - y_s}{y_1 - y_3}$$

$$z_s = z_b + (z_a - z_b) \frac{x_b - x_s}{x_b - x_a}$$

# Метод на z-буфер

## ■ *Предимства*

- ❑ приложим при всички случаи на скриване, закриване, самозакриване
- ❑ резултатът не зависи от реда на обработване на примитивите
- ❑ бърз, лесно се имплементира
  - не се сортират обекти
- ❑ подходящ за хардуерна имплементация
  - бърза памет за съхраняване на z-буфера
- ❑ лесно се прилага при осветеност и сенки

# Метод на z-буфер

## ■ Недостатъци

- изисква допълнителна памет за z-буфера
  - целочислени стойности за дълбочината
  - алгоритъм на сканиращата линия
- данните са споделени
  - четенето и записва изискват синхронизация
- възможно е “нащърбване” при дискретизирането (aliasing)
  - налага се използване на супер-резолюция
- не е подходящ при прозрачни обекти



# Метод на A-буфер

## ■ A-буфер

- anti-aliased, area-averaged, accumulation buffer

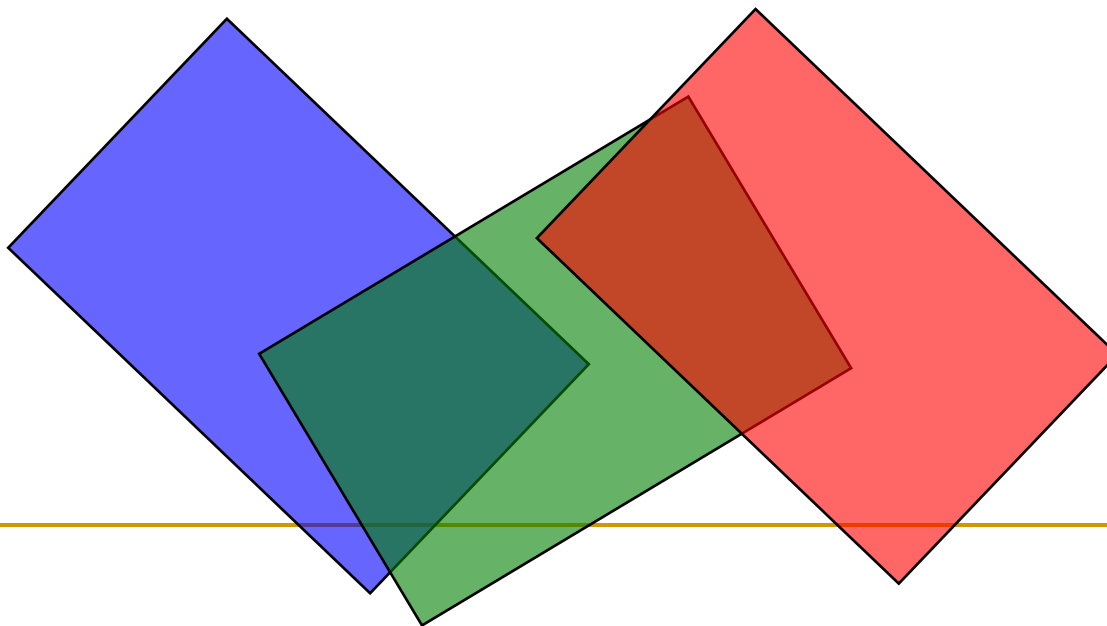
## ■ Разширение на метода на z-буфер

- Разработен от Lucasfilm Studios за системата за графична визуализация **REYES**
- **Renders Everything You Ever Saw**



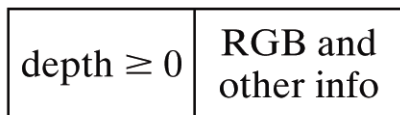
# Метод на A-буфер

- Разширението на метода на z-буфер е в посока на допускане на прозрачност
- Основна структура за данни в алгоритъма на A-buffer
  - *accumulation buffer*

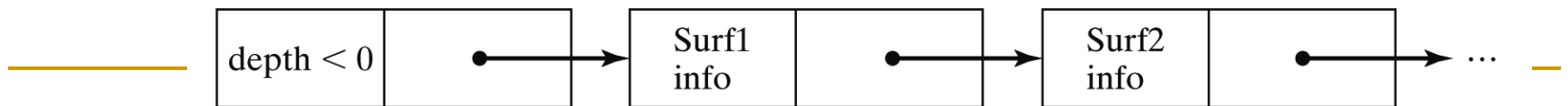


# Метод на A-буфер

- Ако дълбочината е по-голяма от 0, то в A-буфера се съхранява z-координатата за съответния пиксел от екрана
  - както и при z-буфер
- Ако дълбочината е по-малка от 0, то в A-буфера се съхранява указател към свързан списък с данни за визуализиране на повърхнината



(a)



(b)

# Метод на A-буфер

- Информацията за повърхнината в A-буфера включва
  - стойности на RGB компонентите на цвета
  - параметър за непрозрачност
  - дълбочина (z-координата)
  - процентно покритие на многоъгълника
  - идентификатор на повърхността
  - други параметри за визуализиране на многоъгълника
- Алгоритъмът с A-буфер е същия както при z-буфер
  - стойностите на дълбочината и прозрачността се определят за определяне на окончателния цвят на пиксела

# Метод на сканиращата линия

- Метод за определяне на видимите повърхности в пространството на изображението
- Изчисляват се и се сравняват стойностите на z-координатите в сцената по протежение на различни сканиращи линии
- Използват се две таблици
  - таблица на ръбовете (*edge table*)
  - таблица на стените (*surface facet table*)

# Метод на сканиращата линия

- **Таблицата на ръбовете** съдържа
  - координати на крайните точки на отсечки в сцената
  - обратния наклон на всяка отсечка
  - указател към таблицата на стените
- **Таблицата на стените** съдържа
  - коефициентите на равнината, в която лежи стената
  - характеристики на материал на стената
  - други данни за повърхността на стената
  - възможно е да има указатели към таблицата с ръбовете

# Метод на сканиращата линия

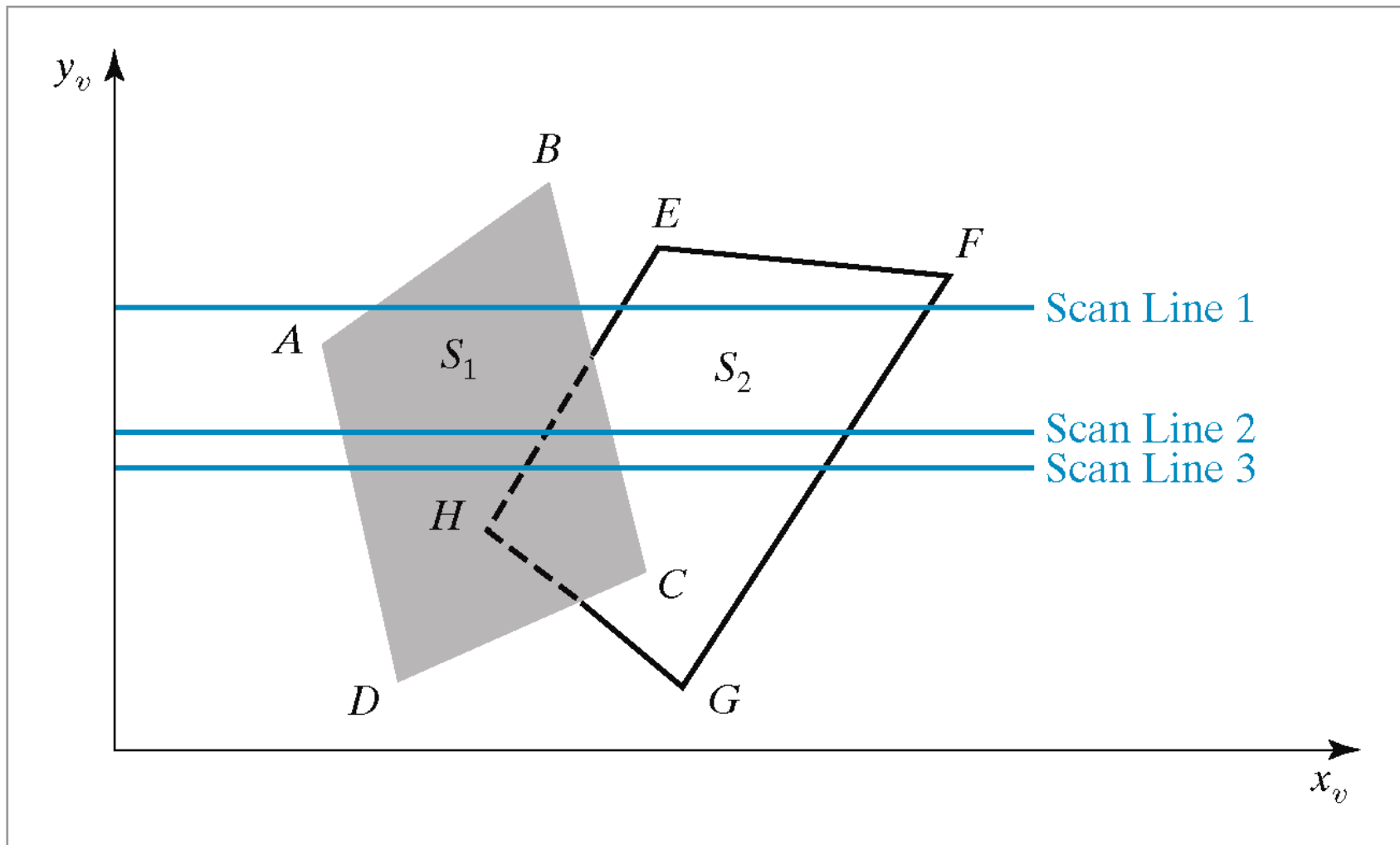
- За да се ускори определянето на пресечни точки за дадена сканираща линия се формира активен списък на ръбовете за всяка сканираща линия по време на обработването ѝ
  - активният списък ръбове съдържа само тези ръбове, които пресичат сканиращата линия в ред на намаляване на  $x$
- За всяка стена се задава флаг, който определя дали дадена точка по протежение на сканиращата линия е вътре или извън многоъгълника на стената

# Метод на сканиращата линия

- Пикселите по протежение на всяка сканираща линия се обработват от ляво надясно
  - при пресичане отляво със стената флагът на стената се вдига
  - при пресичане отдясно със стената флагът се сваля
- Изчисления за дълбочината на разположение се извършват само ако флаговете на повече от една стена са вдигнати за определена точка от сканиращата линия



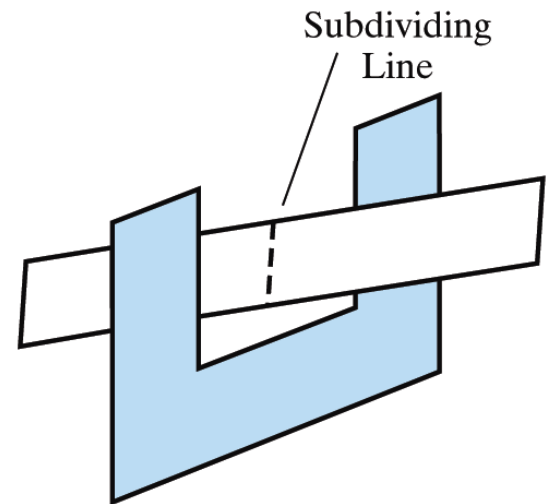
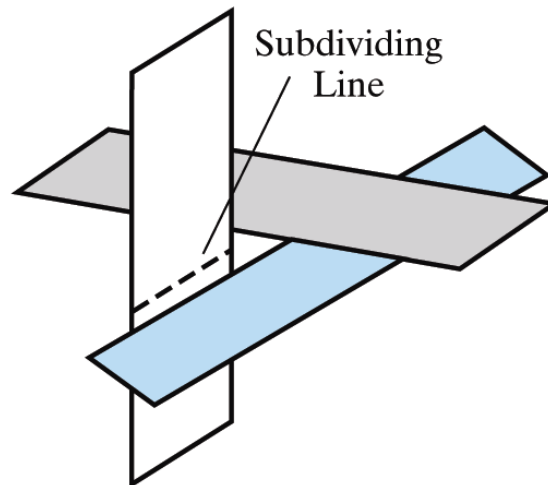
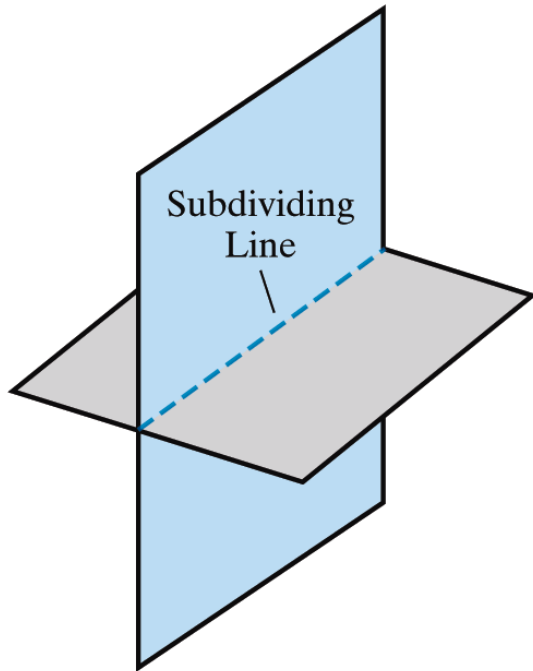
# Метод на сканиращата линия



# Метод на сканиращата линия

## ■ Ограничения

- методът не се справя със случаите на пресичане на многоъгълници или при циклично припокриване
  - такива многоъгълници следва да бъдат разделени



# Видими обекти / скрити обекти

- **Редуциране на сложността и ускоряване на изчисленията**
  - използва се съгласуваност в различни аспекти
    - **Обекти**
      - ако обектите са разделени, то се сравняват обекти, не полигони
    - **Полигони**
      - ако полигон се променя плавно, то се модифицира инкрементално
    - **Отсечки**
      - отсечка променя видимостта само ако пресича друга отсечка или полигон
    - **Сканираща линия**
      - множество видими обекти не се променя значително по протежение на сканираща линия
    - **Област**
      - групи съседни пиксели често принадлежат на един и същи видим полигон
    - **Дълбочина**
      - съседни части от една повърхнина обикновено имат близки стойности за дълбочината
    - **Кадри**
      - в анимация съседните кадри обикновено са много подобни

# КРАЙ

---

Следваща тема:

Геометрично моделиране  
на 2D и 3D обекти