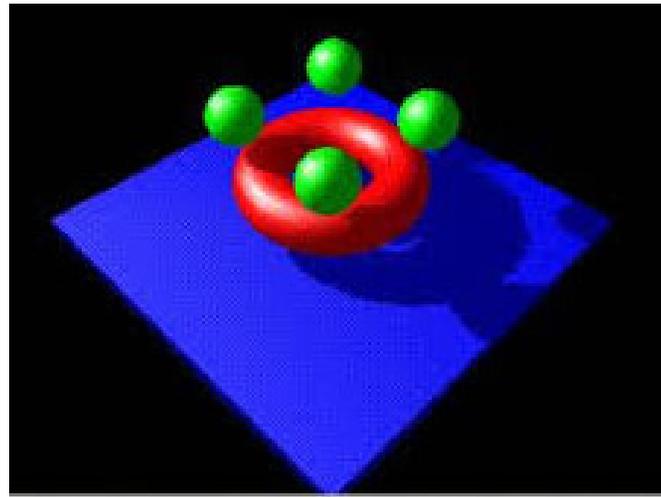


aula 20  
Retornando ao Realismo  
IC/UFF – 2017



**Aura**

e quanto tiver mais de um

Objeto na cena?

Mais Algoritmos....

Para tratar a oclusões por outros objetos



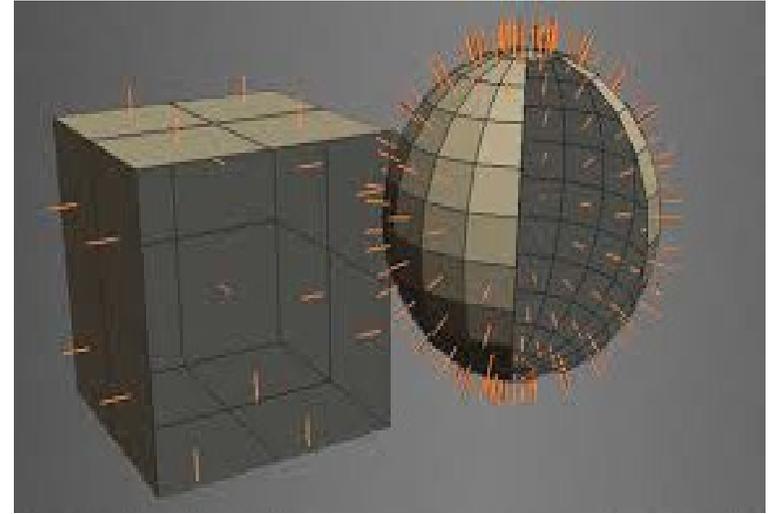
# Técnicas de visibilidade

*Back face culling* (método de Roberts ou teste da normal)

*Priority fill* ou *painter's algorithm*

*Z- buffer*  
(*min Max*)

*Ray casting*  
(*Ray tracing simplificado*  
ou *aproximado*)



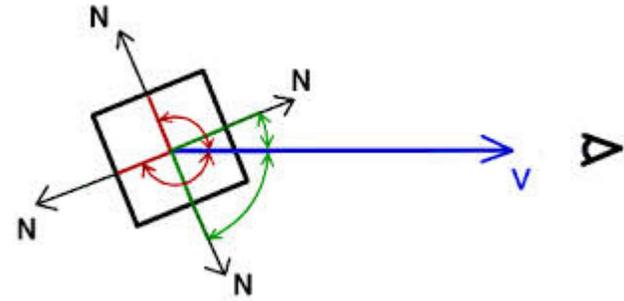
# *Back face culling*



*Em CG **back-face culling** determina quando a face de um objeto será visível.*

*Esse processo torna o **rendering** mais eficiente pois reduz o número de polígonos a ser desenhado*

## *Back face culling*



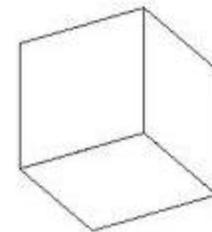
Idéia básica:

**Remover faces traseiras dos objetos em relação ao observador**

Adequadas para objetos convexos.

OBS :

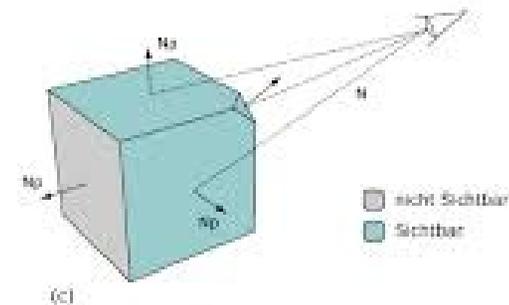
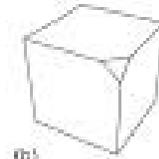
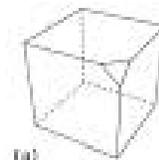
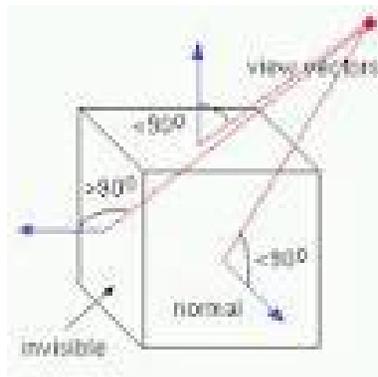
Ser **não convexo**  $\neq$  ser **côncavo**

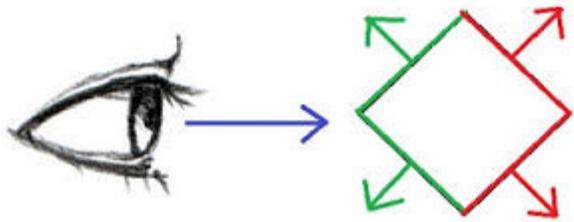


# Algoritmo no espaço do objeto

Usa-se a **direção que as normais** às faces fazem com a direção de visualização.

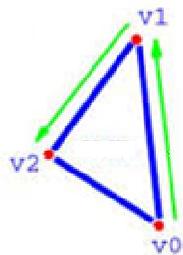
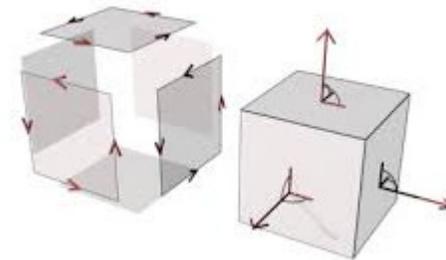
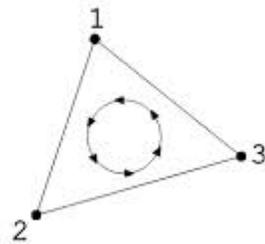
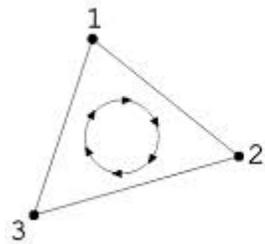
Entre **-90** graus e **90** graus a **face é visível** pelo observador (ou a face é de frente) .





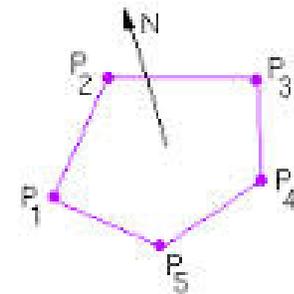
# 1-Obtêm a normal às faces

Através do cálculo do **produto vetorial** de dois vetores da face: a ordem dos vértices é importante!



$$N = (V_1 - V_0) \times (V_2 - V_0)$$

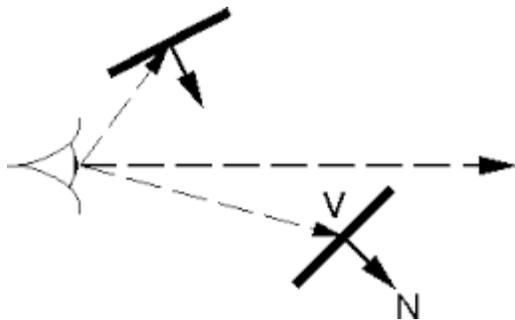
$$(V_1 - V_0) \times (V_2 - V_0) = -(V_2 - V_0) \times (V_1 - V_0)$$



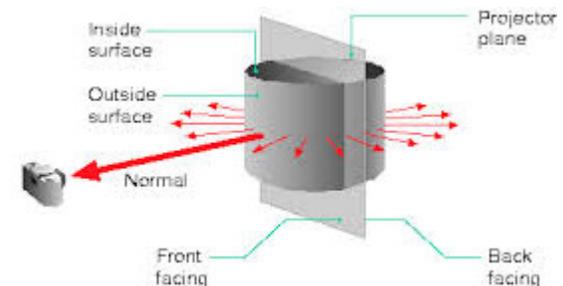
2 - Define-se o vetor da direção de visão

### 3- Verifica-se o ângulo!

Através do **produto interno** entre as normais e a direção de visão, (não é preciso calcular o ângulo) apenas ver se o resultado **é maior que zero** → ângulo entre  $-90^\circ$  e  $90^\circ$  !



$$(V_0 - P) \cdot N \geq 0$$



# Passando a considerar as projeções

Isto é , supomos que voce já tenha decidido como vai passar de 3D para 2D.

Qual será a direção de vista da cena!!

ALGORITMOS NA FORMA **RASTER** (tratam pixels a pixel!)

ou

semi raster (pode tratar grupo de pixels!)

# *Painter's algorithm*

**Painter's algorithm**, ou **priority fill**, é uma das soluções mais simples para o problema de Visibilidade não resolvido pelo método anterior.

Na projeção de cena 3D para o plano do vídeo 2D é necessário **decidir que faces são visíveis ou escondidas** ( hidden ) .

O nome "painter's algorithm" se refere a técnica usada por pintores : primeiro pintam detalhes mais longes da cena e depois os cobrem com as partes mais próximas.

O **painter's algorithm** desenha os polígonos da cena pela sua distância (depth): dos mais longes para os mais próximos (**farthest to closest**).

Cobrindo assim as partes não mais visíveis — ou seja o visibility problem é resolvido com algum custo extra (the cost of having painted invisible areas).

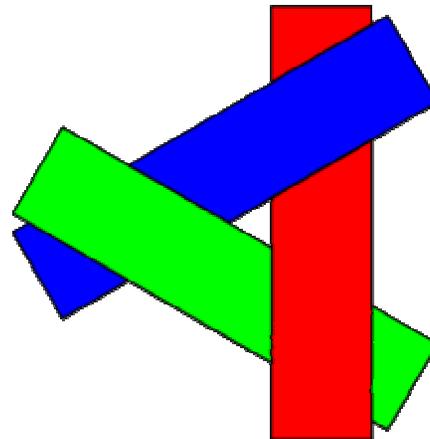
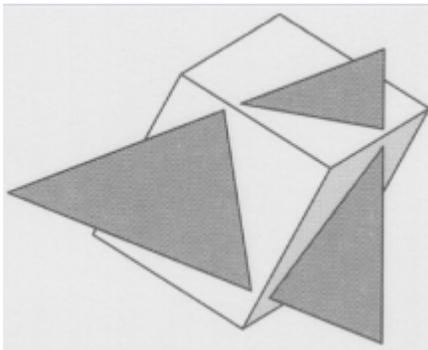
A ordem usada é chamada **depth order**. **Essa ordenação tem uma boa propriedade**: if one object obscures **part** of another **then it is painted after** the object that it obscures. **RESOLVE ALGUNS CASOS DE PARCIAIS**

# *Painter's algorithm*

*Possibilidade de falha → quando parte MAIORES de uma face se sobrepoem a outra → solução divisão da face (Newell's Algorithm).*

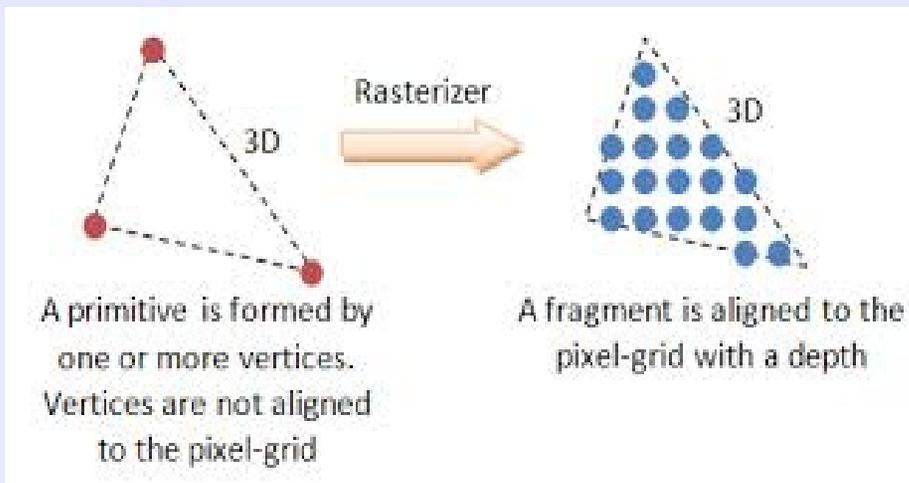
Essa falha do algoritmo levou ao desenvolvimento do método de

**z-buffer ou depth buffer**



## HÁ ALGORITMOS NA FORMA **VETORIAL** E **RASTER**

**RASTER:** o objeto em 3D é tratado na forma final quando já “*discretizado*” em pixels.



### **Rasterisation**

(ou **rasterization**)

converte uma imagem descrita como vector format para a forma de pixels ( dots ) para representação em video, printer ou storage in a bitmap file format.

# **z-buffer algorithm**

Idéia básica: testar a distância (z - depth) de cada superfície para determinar a mais próxima (visible surface).

Considera um array :  $z\ buffer(x, y)$  para cada pixel  $(x, y)$  .

Esse array é inicializado com maximum depth.  
Após isso o algorithm segue como:

# **z-buffer algorithm**

**for each polygon P**

**for each pixel (x, y) in P**

**compute z\_depth at x, y**

**if z\_depth < z\_buffer (x, y) then**

**set\_pixel (x, y, color) = intensidade de P em (x,y)**

**z\_buffer (x, y) = z\_depth**

**Vantagem do z-buffer:**

**sempre funciona e é de simples implementação!**

## **z-buffer** *algorithm*

Considerando o quando um ponto é opaco ou transparente.

Conceito de canal alfa ou composição de transparência:

**Alpha compositing:** processo de combinar a imagem com o fundo criando a aparência de **partial** or **full transparency**.

# Idéia de translúcidos – modelo RGB $\alpha$

Considere 2 polígonos, um **vermelho=R (red, 1 , 0 , 0, 0.5 )**, e o outro **azul=B(blue, 0 , 0 , 1, 0.5 )** renderizáveis em um fundo **verde=G(green background (0 , 1 , 0 , 0))**.

Ambos **50% transparentes**. Se o **V(red)** estiver na frente de todos, depois o **azul (blue)** e o **verde** for o fundo (**green background**).

No final deve-se ter **50% R, 25% G** e **25% B** (Renderizando de traz para a frente as percentagens da cada cor):

**Green background.** (0 , 1 , 0 )

**Polígono blue :** (0 , 0.5 , 0.5 ) – conta 50% da cor sobre o fundo!

**Polígono red:** (0.5 , 0.25 , 0.25 ) – conta 50% da cor sobre outras!

**z-buffer** *algorithm com canal alfa!*

OU

## **Alpha-blending + the Z-buffer**

**Given:** A list of polygons  $\{P_1, P_2, \dots, P_n\}$  and a background

**Output:** A COLOR which displays the intensity of the polygon surfaces.

**Initialize:** z-depth and z-buffer(x,y) , -buffer(x,y)=max depth; and  
COLOR(x,y)=background at (x,y)

**Begin:**

# **z-buffer *algorithm com canal alfa!***

```
for(each polygon P in the polygon list)
do{
  for(each pixel(x,y) that intersects P)
  do{
    Calculate z-depth of P at (x,y)
    If (z-depth < z-buffer[x,y])
    then{
      z-buffer[x,y]=z-depth;
      COLOR(x,y)=Intensity of P at(x,y);
    }
    #considerando  $\alpha$ :
    Else if (COLOR(x,y).opacity < 100%)
    then{ COLOR(x,y)=Superimpose COLOR(x,y) in front of Intensity of P at(x,y); }
    #End consideração do  $\alpha$ :
  }
}
display COLOR array.
```

## *Masking Technique ou mim Max*

Muito bom para o Hidden lines de curvas.  
Isso é eliminar linhas invisíveis de superfícies

É um ALGORITMO NA FORMA **RASTER**

**depende**

**Da direção de vista da cena!!**

**e**

**de voce já ter passado de 3D para 2D.**

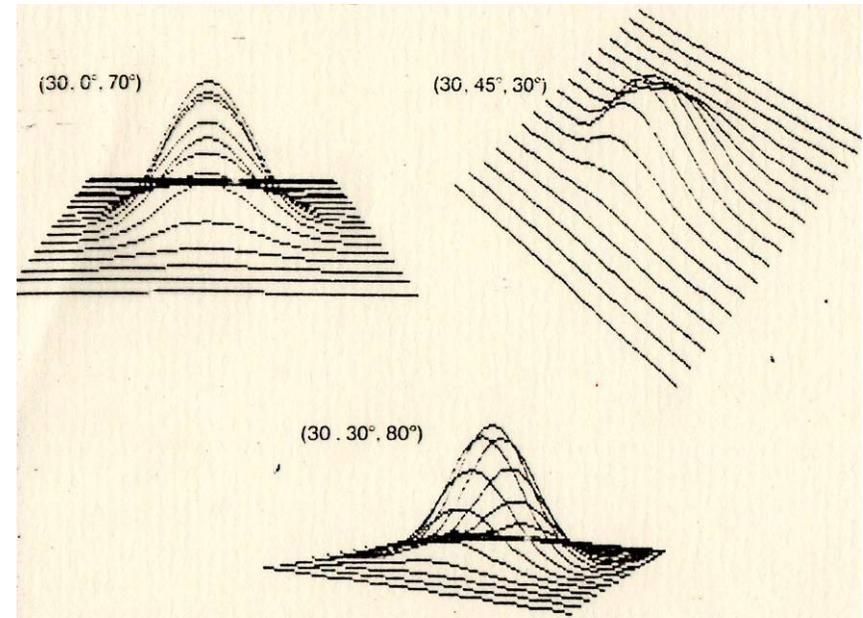
# Imagine que foi gerada uma superfície

A partir de uma série de curvas.

E que voce já tem a projeção dela a partir de um certo ponto de vista.

Ou sua projeção de determinada direção

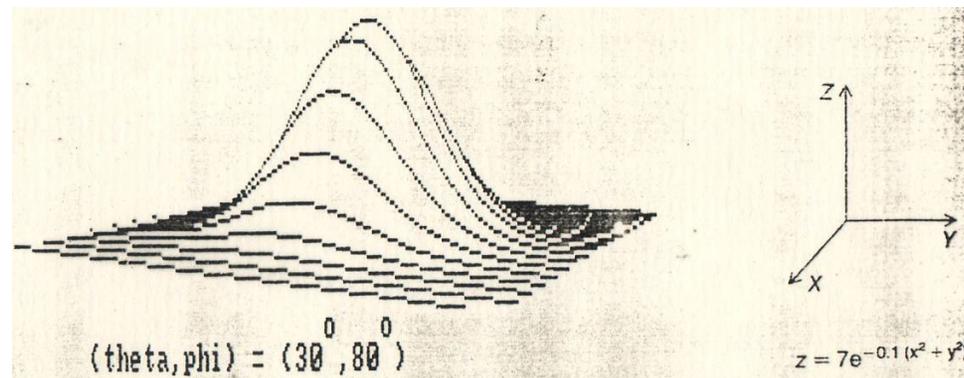
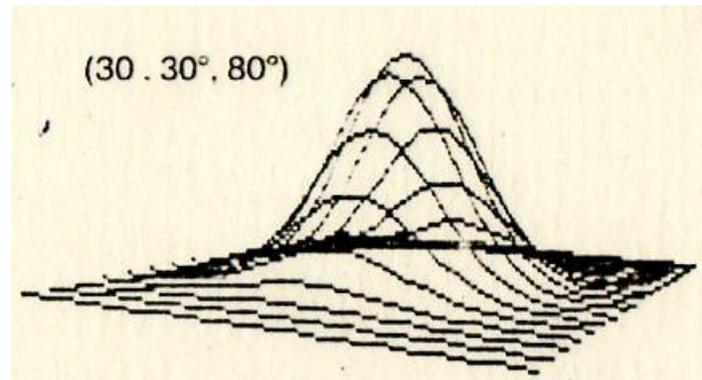
Ou seja ela já é descrita por uma série de linhas em 2D de determinada direção.



Mesma superfície representada por um conjunto de curvas e Vista de diversos pontos de vista

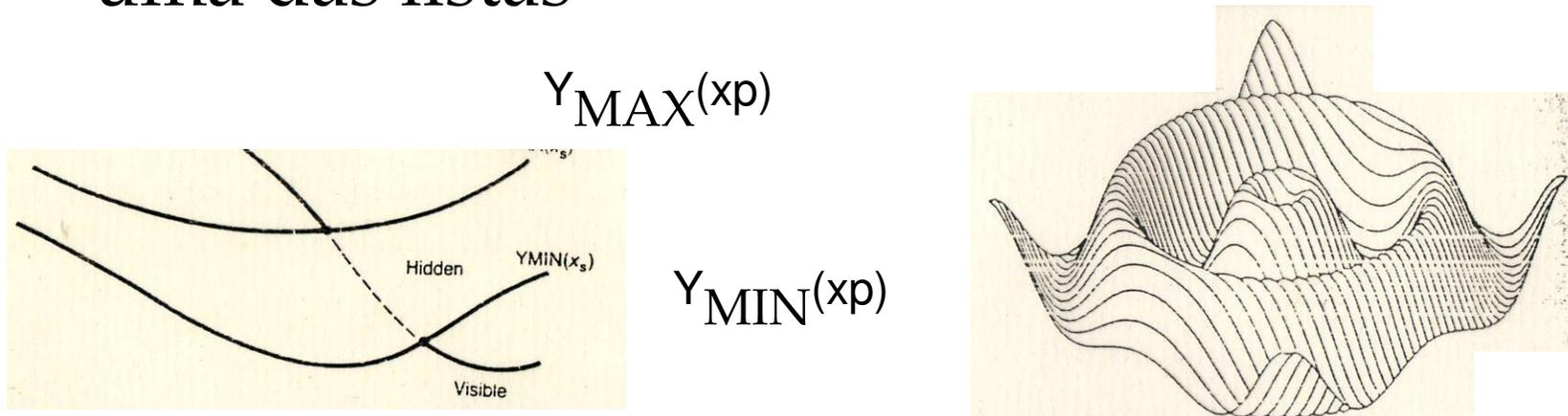
# Como eliminar as linhas que são obscurecidas por partes da superfície mais a frente?

Hidden lines por mascaramento ou lista de limites verticais superiores e inferiores de cada passo (pixel) horizontal



# O conceito da técnica de mascaramento

Para cada pixel ou passo de  $n$  pixels é feito 2 listas de coordenadas verticais  $Y_{MAX}(xp)$  e  $Y_{MIN}(xp)$  e só se desenha se algo ao ser projetado para esse  $xp$  estiver atualizando uma das listas



# O número de pixel usado, ou o passo

Pode ser uma função da curvatura da superfície ou curva.

Mais curvatura menor passo!!

E como se obtém a curvatura?

# O que é curvatura?

curvatura

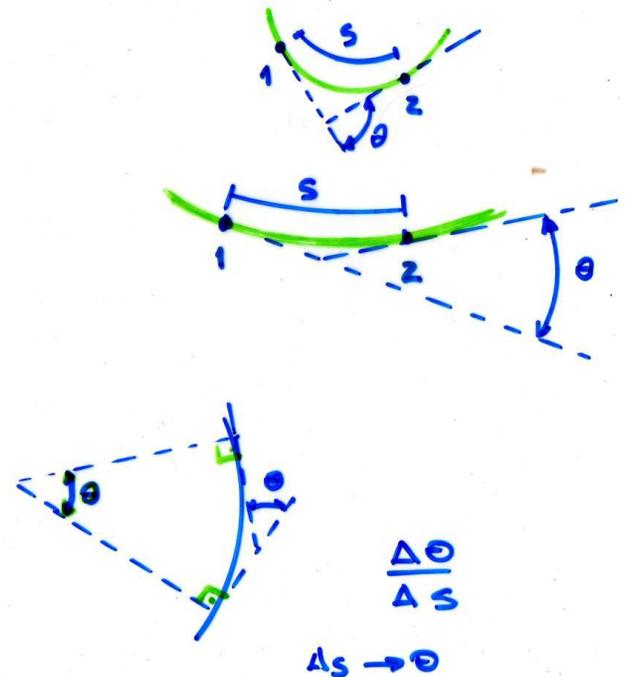
derivada do ângulo  
formado por 2 Tangentes  
à curva em relação ao  
comprimento do arco  
entre essas 2 Tangentes

$$\frac{d\theta}{ds}$$

em um círculo  $ds = R d\theta$

logo curvatura  $\frac{1}{R}$

para os círculos



## Ray tracing *simplificado ou aproximado* *ou*

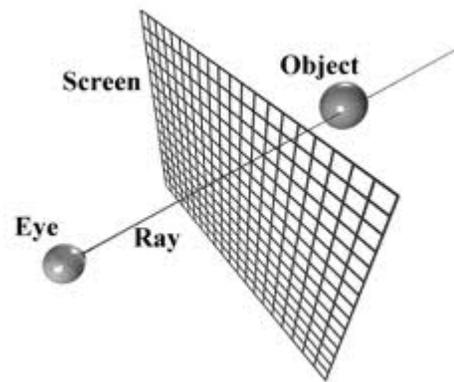
**Ray casting** lança raios a partir do observador de forma a perceber a distância dos objetos que compõem a cena.

Os raios são emitidos **a partir do observador**, (no sentido inverso do que acontece na natureza), para reduzir recursos computacionais (pois a maior parte dos raios de luz que partem da fonte não chegam ao observador).

# Ray casting

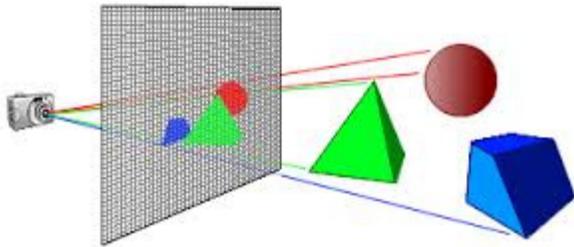
Supõe-se um raio do olho do observador passando por **cada ponto da tela** a ser desenhada. O ponto da tela receberá a cor do objeto que for atingido na cena pelo raio.

O calculo das interseções é o ponto chave do algoritmo.



# Ray casting

permite remover as superfícies escondidas utilizando as informações obtidas a partir das primeiras intersecções encontradas pelos raios lançados a partir do observador.



# Ray tracing (rastreamento)

Método recursivo, onde recorre ao lançamento de raios secundários a partir das interseções dos raios primários com os objetos.

Ray casting é apropriado para a renderização de jogos 3D em tempo-real.

Durante a viagem do raio pode acontecer: absorção, reflexão ou refração. A superfície pode refletir toda ou apenas uma parte do raio numa ou mais direção. A soma das componentes absorvidas, refletidas e refratadas tem que ser igual ao inicial.

Duvida em como calcular as

Interseções?

## Bibliografia:

- D. F. Rogers, J. A. Adams. Mathematical Elements for Computer Graphics, 2dn Ed. , Mc Graw Hill, 1990
- E. Azevedo, A. Conci, [Computação Gráfica: teoria e prática](#), [Campus](#) ; - Rio de Janeiro, 2003
- J.D.Foley,A.van Dam,S.K.Feiner,J.F.Hughes. Computer Graphics- Principles and Practice, Addison-Wesley, Reading, 1990.
- Y. Gardan. Numerical Methods for CAD , MIT press, Cambridge, 1985.
- A. H. Watt, F. Policarpo - [The Computer Image](#) , Addison-Wesley Pub Co (Net); 1998
- [https://noppa.oulu.fi/noppa/kurssi/521493s/luennot/521493S\\_3-d\\_graphics\\_vi.pdf](https://noppa.oulu.fi/noppa/kurssi/521493s/luennot/521493S_3-d_graphics_vi.pdf)
- <http://graphics.stanford.edu/papers/rad/>