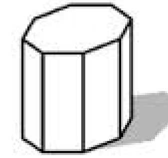


# Realismo Visual

Introdução

Aulas 11 e 12

UFF - 2019

Capitulo 5- livro texto de **computacao grafica**

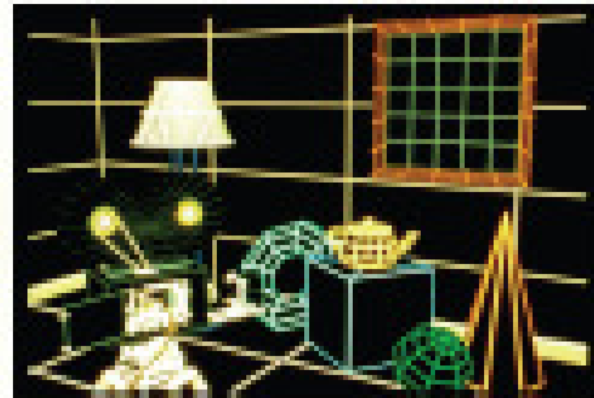
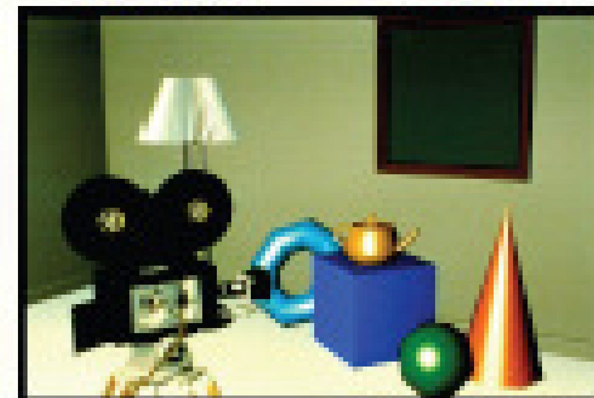
# Objetivos

Melhorar o entendimento das cenas e objetos criados

Possibilidade de representação de dados, objetos e cenas complexas

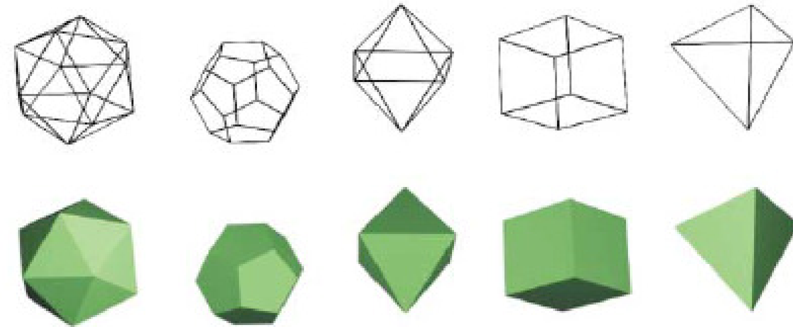
Realismo até o nível desejado da forma adequada para a aplicação

(real time x perfeição física da cena)

# Nível adequado do realismo

Remoção de partes invisíveis do objeto
   (linhas, superfícies e oclusões por
   outros objetos)

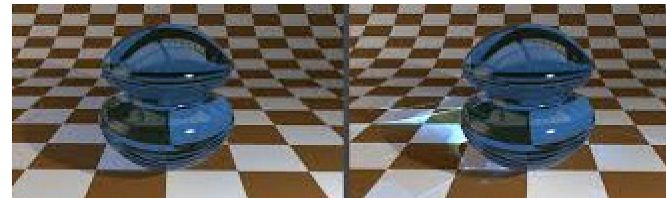


Sombreamento das diversas superfícies
   ou *Shading* :

   reflexão difusa,

   reflexão especular

Demais níveis de detalhes:



   Sombras (*shadows*)

   Reflexão,

   Transparências,

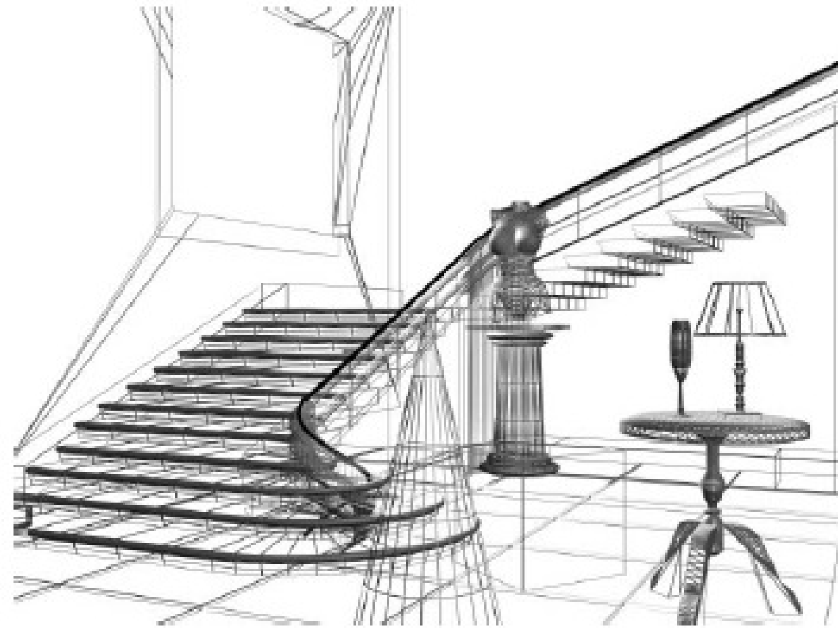
   Refração,

   Texturas

## *Wire frame* : adequado para posicionamentos e desenho, mas não realístico

Todas as linhas são mostradas.

Passo seguinte do realismo eliminar **partes da cena que não são vistas quando objetos opacos são vistos de determinada direção.**



# Tratamento de *hiddens* ou *Hidden Line/surface problem*

Eliminação de linhas:
caso particular da

definição de que faces ou superfícies são ocultas por outras do objeto ou cena.

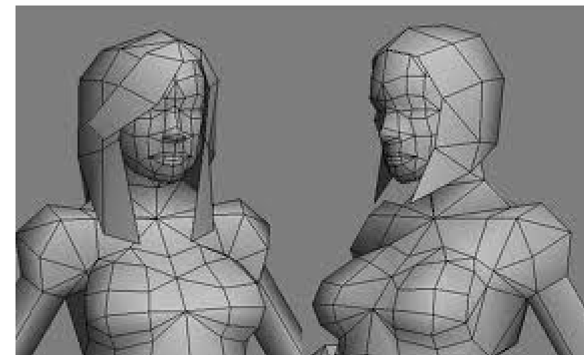
# Técnicas de visibilidade

*Back face culling*

*Priority fill* ou *painter's algorithm*

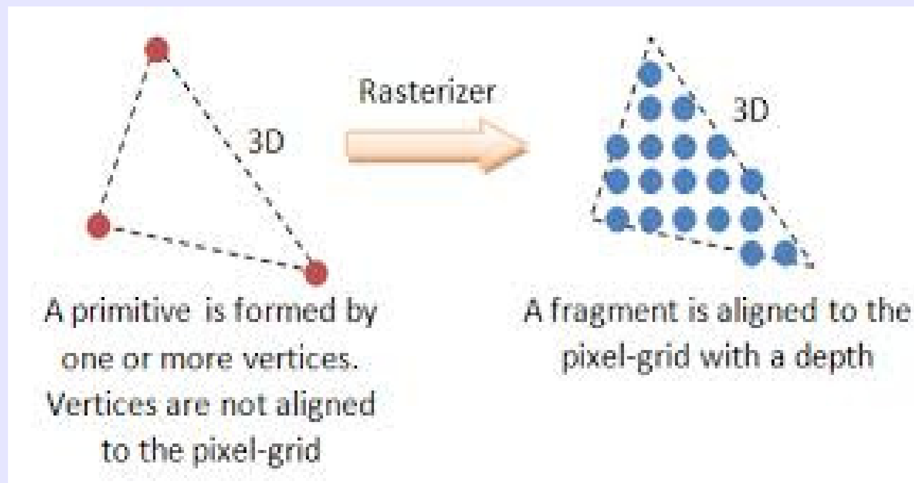
*Z- buffer*

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



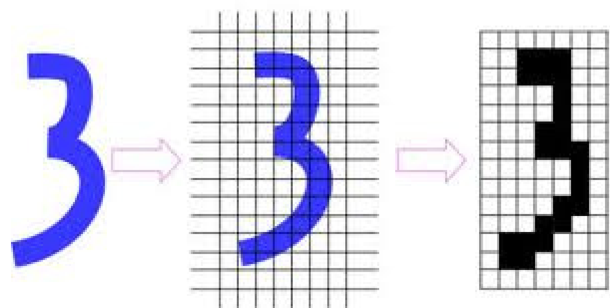
## HÁ ALGORITMOS NA FORMA **VETORIAL** E **RASTER**

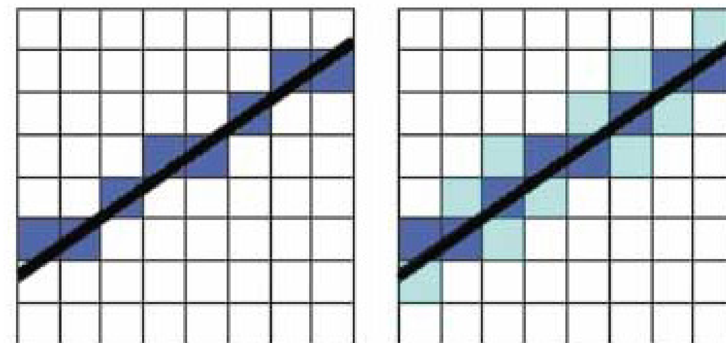
**RASTER:** o objeto em 3D é tratado na forma final quando já “*discretizado*” em pixels.



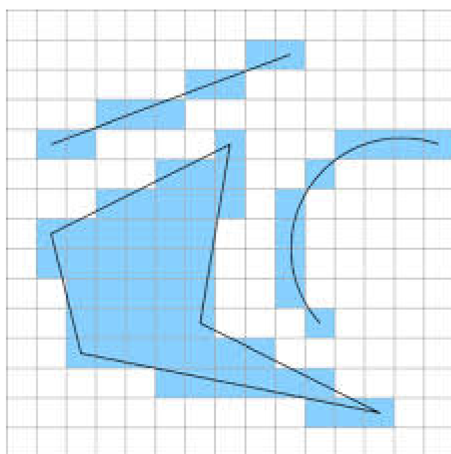

**Rasterisation**
(ou **rasterization**)
converte uma imagem descrita como ***vector format*** para a forma de ***pixels*** ( ***dots*** ) para representação no video, para armazenamento no formato de ***bitmap*** .

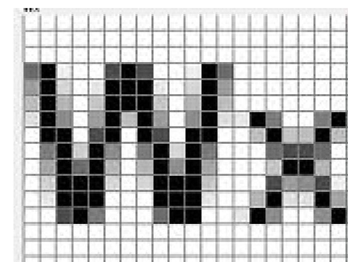
## *Alising → antialising*





*Rasterizar* = Usar a malha de pixels para descrever os objetos!





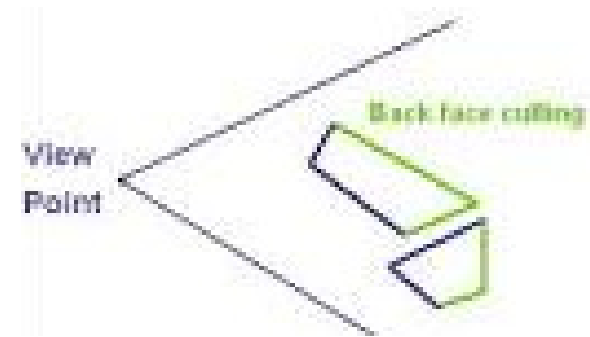
## *Back face culling, método de Roberts* ou teste da normal

Algoritmo posiciona o **objeto** e o **observador** no mesmo sistema de coordenadas (SRU ou WC).

Não considera projeções ou perspectivas inicialmente.

Isso entra em uma outra etapa no processo de visualização (**pipeline**)

# *Back face culling*




Demo: em javascript:

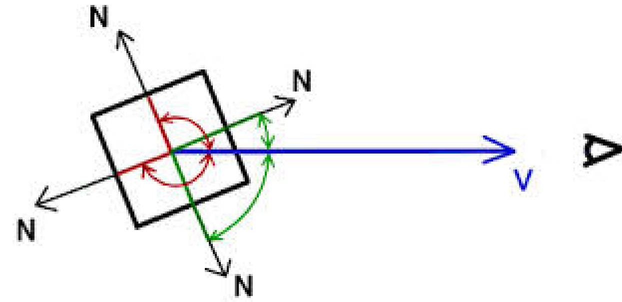
http://echolot-1.github.io/back-face-culling-demo/

echolot-1/**back-face-culling-demo**

*Em CG **back-face culling** determina quando **a face** de um objeto será **visivel de um ponto de vista**.*

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

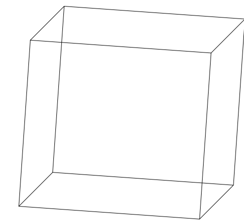
# *Back face culling*




Idéia básica:

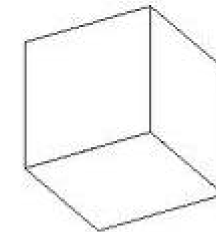
**Remover faces traseiras dos objetos em relação ao observador**

Adequadas para objetos convexos.



OBS :

Ser **não convexo** ≠ ser **côncavo**

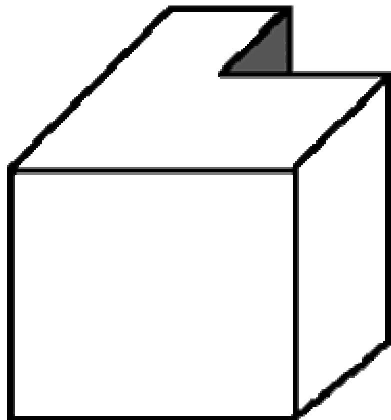


# Objetos convexos

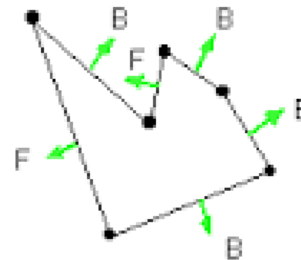
**Definição:**

Formado por faces convexas.

*i.e.* Formado por polígonos convexos: nos quais a **ligação entre quaisquer 2 pontos** internos nunca passa por uma parte externa a face:



não convexo

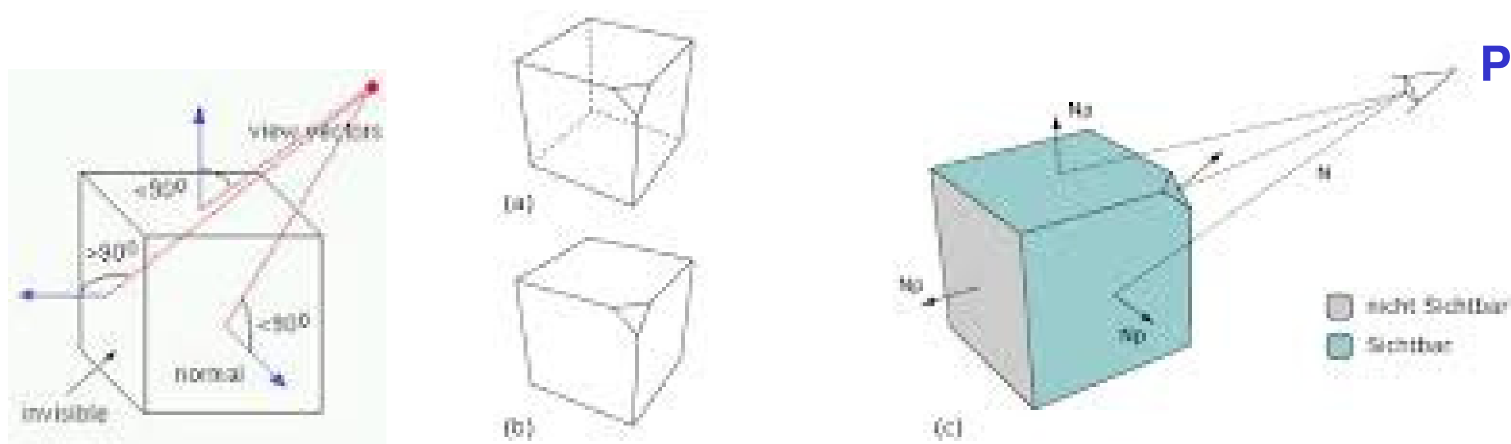


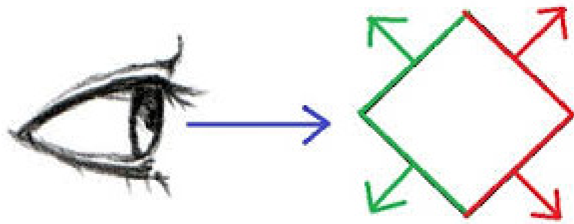

não convexo

# Algoritmo posiciona o objeto e o observador no mesmo sistema de coordenadas (SRU ou WC)

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

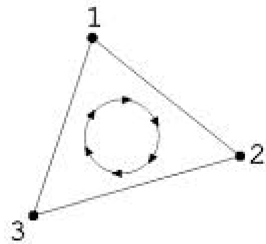
Entre **-90** graus e **90** graus a **face é visível** pelo observador (ou a face é de frente) .

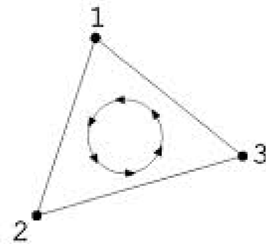


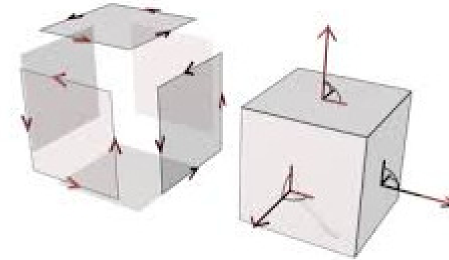


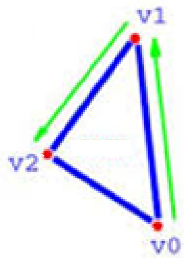
# 1-Obtêm a normal às faces

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

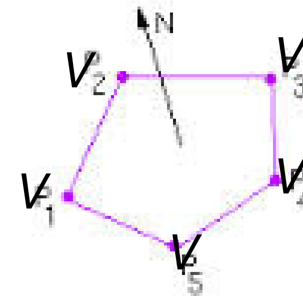





$$N = (V_1 - V_0) \times (V_2 - V_0)$$




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

## 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 ° e 90 ° !

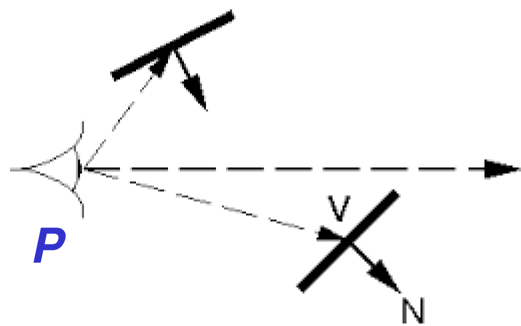
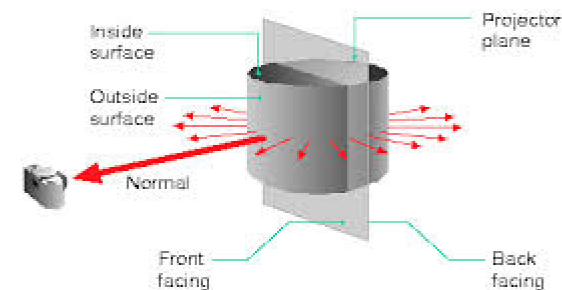



figure 206
Inside and
outside surface

p




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

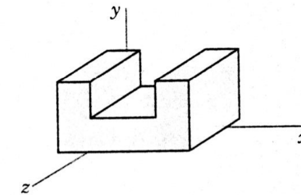
# Algoritmo

4- Só desenha a face se ele é visível !

**OBS- Se for visível ai se preocupa em projetar o objeto de 3D para 2D e em posicioná-lo no sistema de coordenadas do dispositivo .**

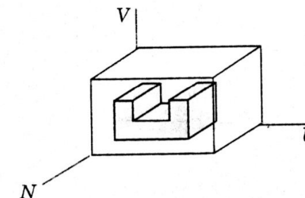
# Viewing pipeline / Ações para ver uma cena

Modelagem dos objetos que compõem a Cena –SRO)

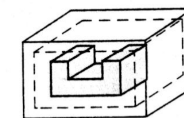



não convexo

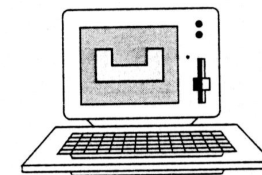
Sua posição no SRU (World Coordinates - WC), sua visão de maneira realística por um observador .




Sua vista em perspectiva e projeção em 2D , se a face for visível .



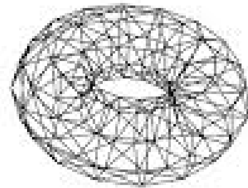
E posicionamento na window ou no canvas de desenho (DC - SRD) se a face for visível .

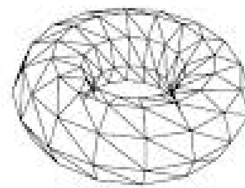


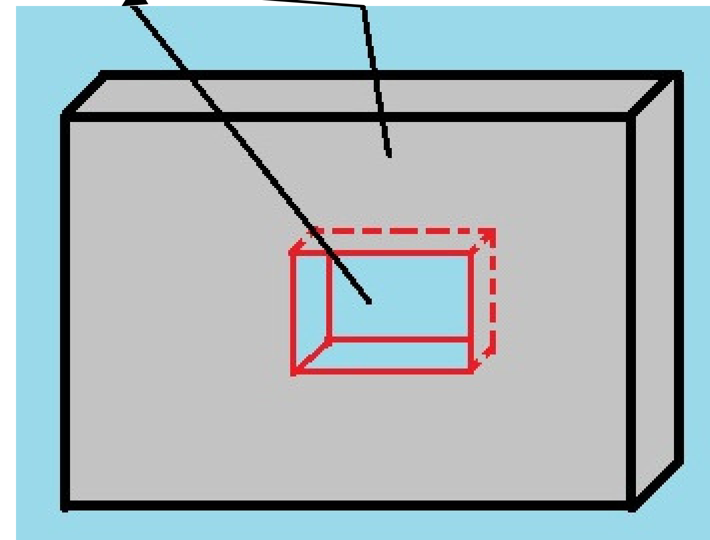
# Fórmula de Euler V – A + F = 2

*Genus* G de um objeto : menor **número de furos** que trespassam o objeto.

**Genus** G=**1**







**não convexo**

***Qual o genus*** de uma tubulação ?

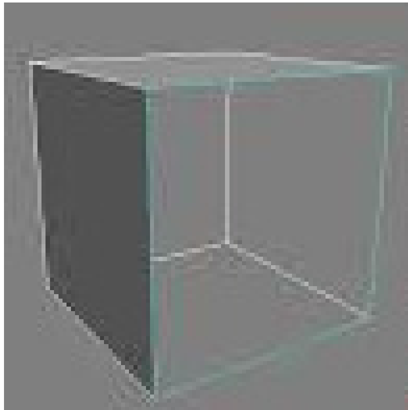
**não convexo**

Resposta: Veja o vídeo no Breno onde ele mostra isso por deformação!
Segue o link do vídeo no youtube: http://youtu.be/QkcryL4f6hE

# Fórmula de Euler :V – A + F = 2

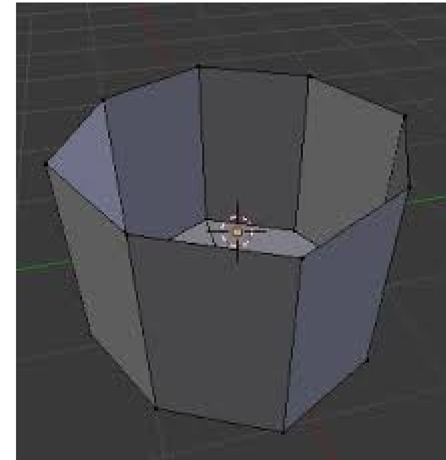
***Buracos H*** : menor **número de furos** que não trespassam ou loops fechados de faces.

**não convexo**



***Buracos*** H=**1**

**não convexo**

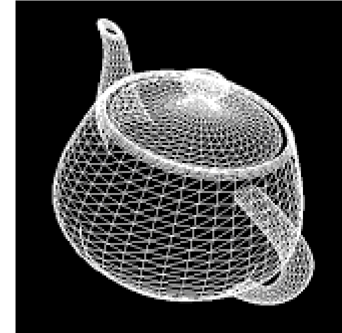


# Formula de Euler→**Euler-Poincaré**:

**Componentes separáveis** ou partes conectadas: C

formula de Euler - Poincaré: V-A+F-H = 2(C-G)

H=1 e G=?



***Utah teapot***

**não convexo**

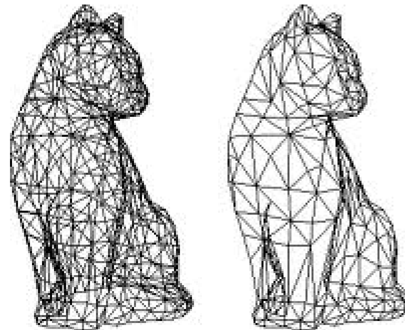
Um **teapot** não é uma
**chaleira** ! Nunca é
usado para por água no
fogo e a ferver!
Mas para por agua fervendo e fazer chá

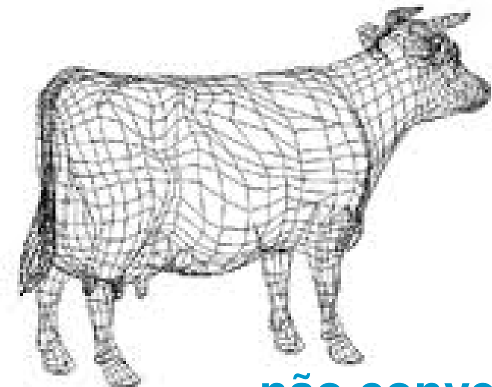
$V - A + F = 2$

# Importante da modelagem correta para o de uso do objeto adequadamente

Já definir se há **buracos H**, ou furos **trespassantes G** ou **partes conectadas C**, na modelagem inicial do objeto é mais complexo.

**não convexo**





**não convexo**

Qual o ***Geno*** de um corpo humano para uma modelagem que o tratasse por dentro, como para uma endoscopia?

# Painter's algorithm

**Painter's algorithm**, ou **priority fill**, é uma das soluções mais simples para o problema de Visibility em 3D CG.

Na projeção de cena 3D para o plano do video 2D é nescessario **decidir que faces são visiveis ou escondidas (**hidden) .

O nome "painter's algorithm" se refere a técnica usado por pintores : primeiro pintam coisas mais longes da cena de depois as cobrem com as partes mais próximas.

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

Cobrindo assim as parte invisiveis — ou seja o ***visibility problem*** é resolvido com algum custo extra (o custo de ter pintado areas desnecessarias).

A ordem usada é chamada *depth order*. ***Esssa ordenação tem uma boa propriedade***: se um objecto obscurece **parte de outro** en**tão ele é pintado depois do que vai** obscurecer.

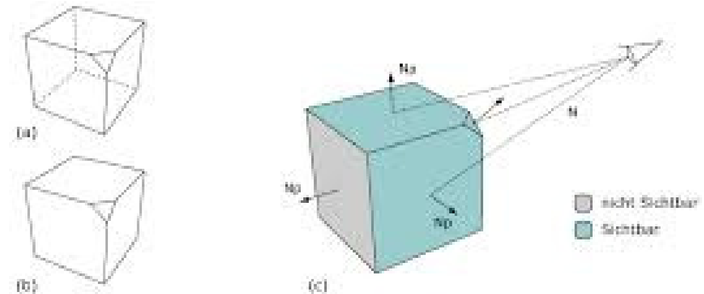
# *Painter's algorithm*

Como a distância da Face pode ser computada?

*1- Pelo cálculo da distância média dos Vértices da Face ao observador P (Xp,Yp,Zp)*

*2- Fazendo uma interpolação da distancia dos vertices (ViX, ViY, ViZ) ao observador P (Xp,Yp,Zp)*

*(ViX, ViY, ViZ) - (Xp,Yp,Zp) para cada vertice i Da face*



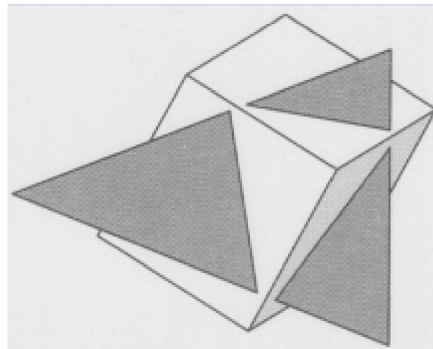
# *Painter's algorithm*

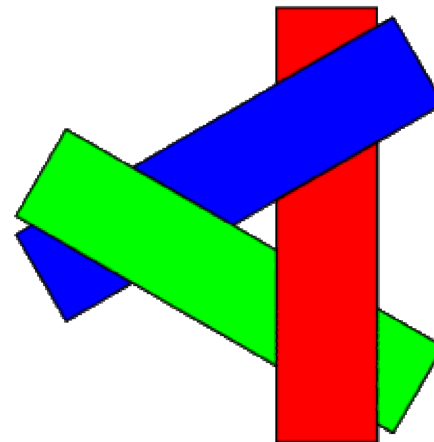
Pelo distância computada , há

*Possibilidade de falha → quando parte 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**





Como a distância da Face pode ser computada?

3- *Cálculo da distância de cada ponto da FACE ao observador.*

Essa é a idéia básica do **z-buffer algorithm** :

testar a distância (z – depth) de cada ponto da cena para determinar a face mais próxima do observador (visible surface).

Considera um array de todos os pixels a serem pintados:  z buffer(x, y) para cada pixel (x, y) .

Esse array é inicializado com “maximum depth”.

Após isso o algoritimo segue como:

# z-buffer algorithm

```
for cada pixel (x, y) da cena z_buffer (x, y) = maximum
    depth
for cada face P da cena
    for cada pixel (x, y) de cada face P
            compute z_depth de (x, y)
            if z_depth < z_buffer (x, y) then
            set_pixel (x, y, color)= cor de P em (x,y)
    z_buffer (x, y) = z_depth
```

**Vantagem do z-buffer:**

**sempre funciona e é de simples**

# **z-buffer** *algorithm* Com **canal alfa**

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 **transparencias em diversos níveis**.

# Idéia de translúcidos – modelo RGBα

Considere 2 polígonos, um **vermelho=(1 , 0 , 0, 0.5 ),** e o outro **azul=(** 0 , 0 , 1, 0.5 **)** *renderizados* em um fundo **verde=(0 , 1 , 0 , 0).**

Ambos **50% transparentes**. Se o **V(red)** estiver na frente do verde, entre 0,5 do verde na cor.

Se o **V(red)** depois o **azul (blue)** e depois o **verde fica 0,5 da cor que já está no azul transparente**.

No final deve-se ter 100% R, 25% G e 50% B (Rederizando de traz para a frente):

Fundo Verde nada Transparente. (0 , 1 , 0 )

Poligono **azul=(**0 , 0.5 , **1**) – conta 50% da cor sobre o fundo!

Poligono **vermelho=(1** , 0.5 , 0 ) – conta 50% da cor do fundo verde!

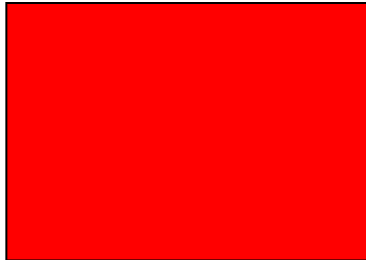
Poligono **vermelho=(** 1 , 0.25 , 0.5 ) – conta 50% da cor sobre o fundo azul transparente !

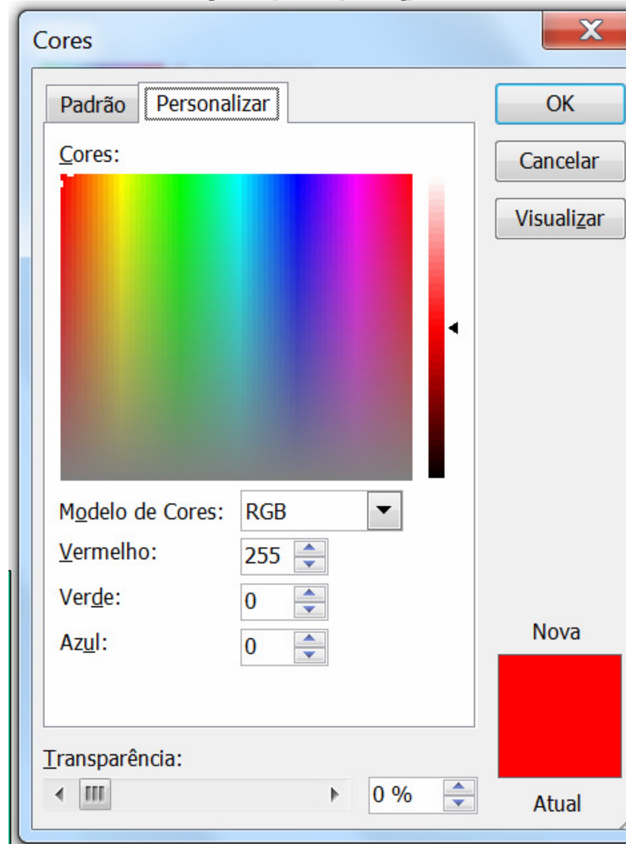
# Uma Cor transparente no fundo

Branco= (1,1,1)

Se cada cor com 1 byte:
Branco= (255,255,255)

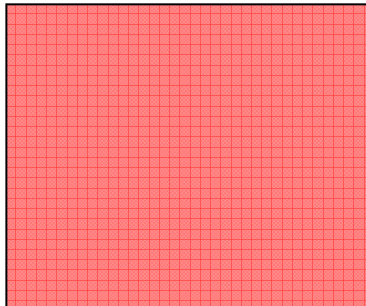
**vermelho=(1** , 0 , 0 )



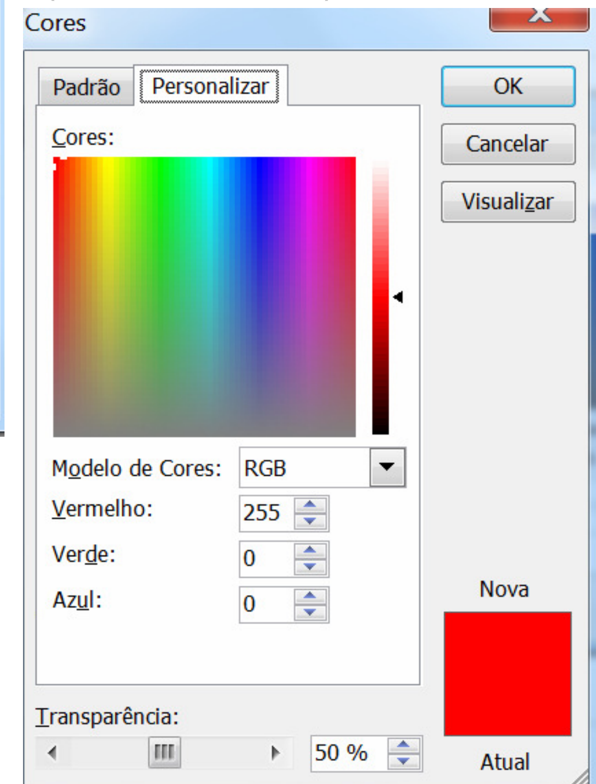



Vermelho maximo:
= (255,0,0)

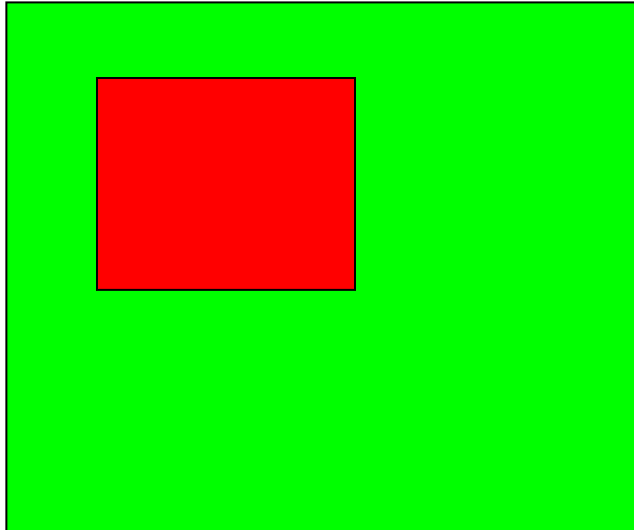
Vermelho 50% transparente:= (255,127,127)

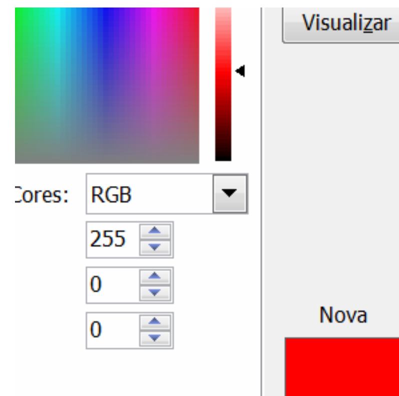


**vermelho=(**1 , 0.5 , 0.5 )



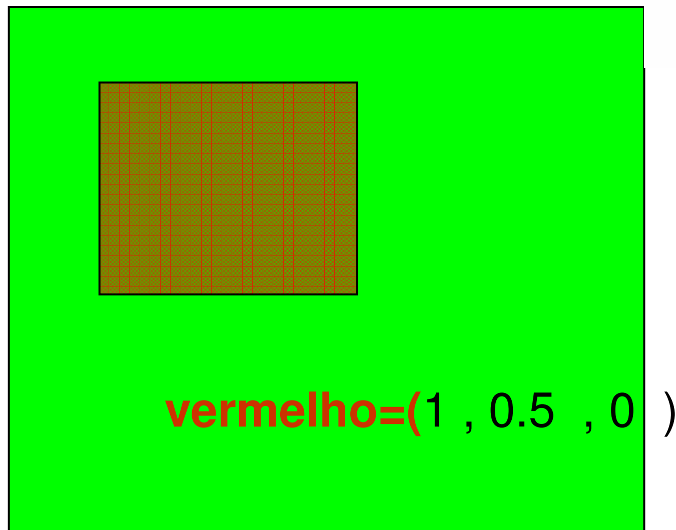
# 2 cores com transparencia

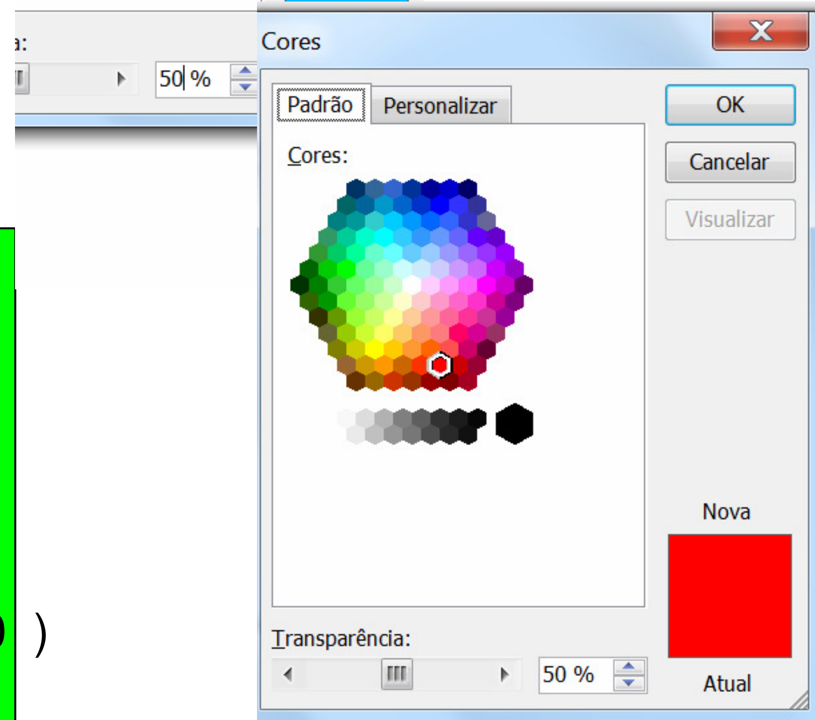


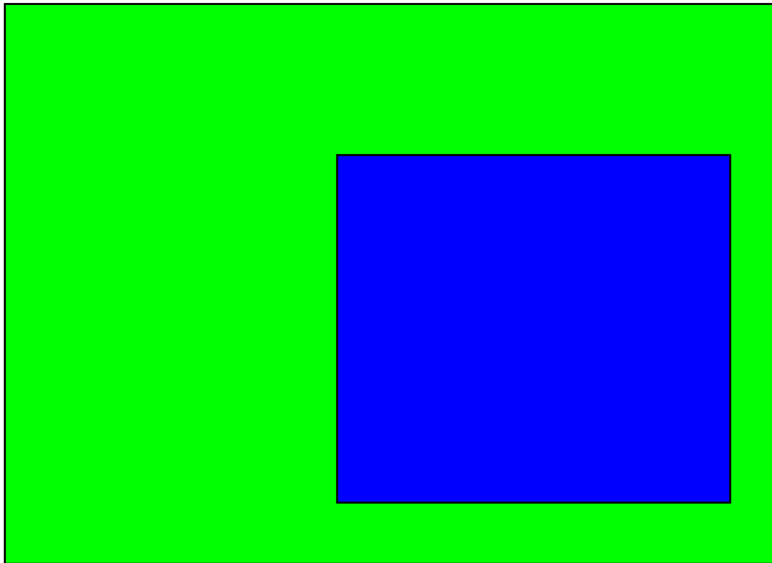



vermelho=(1 , 0 , 0 ,0 ),
sobre qualquer fundo,
sempre é vermelho

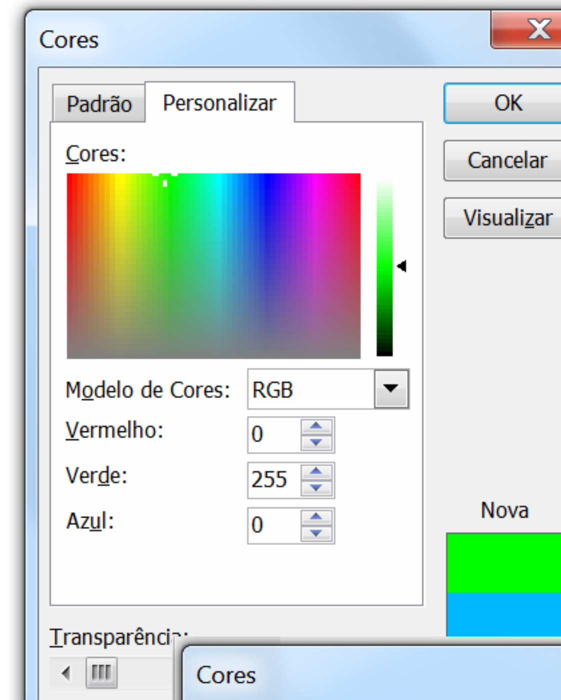
vermelho=(1 , 0 , 0 ,0,5 ),
sobre fundo verde recebe 0,5
Do verde

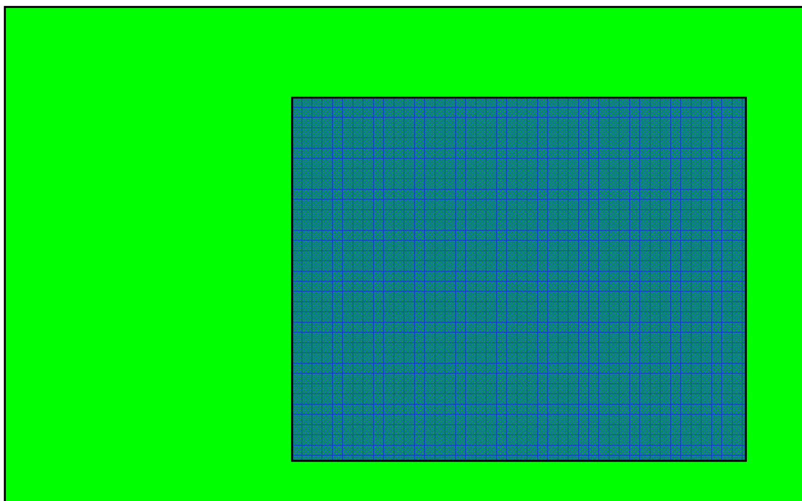


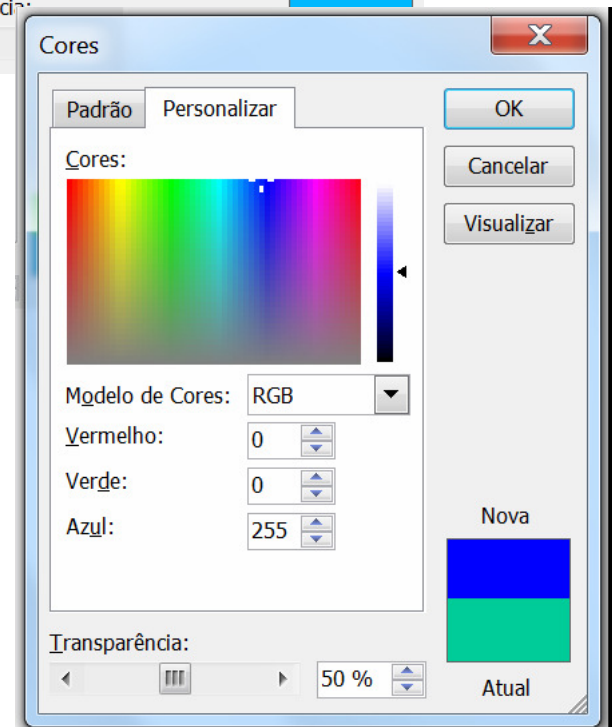





azul=( 0 , 0 , 1, 0 )

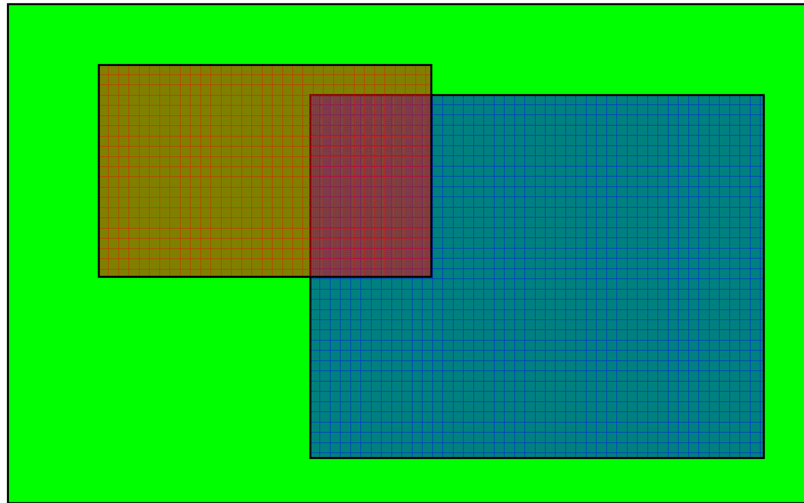





azul=( 0 , 0 , 1, 0.5 ) antes de combinar com o fundo

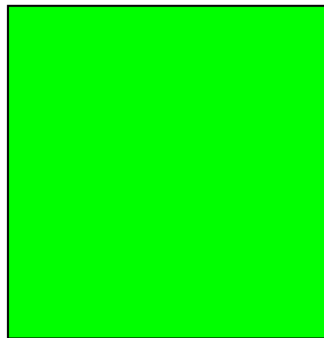
azul=( 0 , 0,5 , 1) depois de combinar com o fundo



**vermelho=(1** , 0 , 0 ,0,5 ),
Em 1 byte or canal de cor = (255,0,0)
**vermelho** transparente sobre fundo verde recebe 0,5 do fundo (255,127,0)

Tom de azul transparente **azul=(** 0 , 0 , 1, 0.5 **), sobre fundo verde e**m 1 byte or canal de cor = (0,255,127)

Vermelho sobre essa cor de fundo do azul transparente: Resultado : (255, 127, 63)



Fundo Verde nada Transparente. (0 , 1 , 0, 0 )

Em 1 byte or canal de cor = (0, 255,0)

# **z-buffer** *algorithm com* canal alfa!
# ou
# **Alpha-blending + the Z-buffer**

**input:** uma lista de **Faces {P1, P2, .....Pn}** e uma cor de FUNDO ou **background**

**Output:** uma COLOR que descreve a intensidade da Face.

**Inicialize:** COLOR(x,y), z-depth e z-buffer(x,y):
z-depth =z-buffer(x,y)=max-depth;
COLOR(x,y)=**background**

**Begin:**

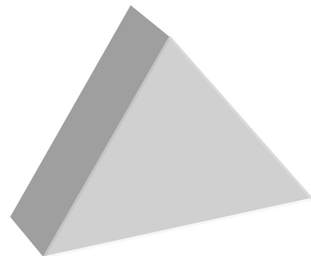
# **z-buffer** *algorithm com* canal alfa!

```
For ( cada face P na lista de Faces)
Do {
    for ( cada  pixel(x,y) de uma face P)
    do {
        Calcule z-depth de P em (x,y)
        If (z-depth < z-buffer[x,y])
        then{
                z-buffer[x,y]=z-depth;
         COLOR(x,y)=Intensidade da cor de P em (x,y);
                }
        considerando α (transparencia) :
        Else if (COLOR(x,y). α < 0%)
        then { COLOR(x,y)= calcule COLOR(x,y) em função da Intensity de  P(x,y); }
        #End consideração do α:
          }
    }
Pinte  o COLOR(x,y) .
```

# Terceira parte do trabalho 1

**Entrega: 04/06/2019 - terça**

**Desenhar a figura 3D do seu grupo como se fosse um objeto fosco , isso é sem as arestas das faces não visiveis e fazer ela ser a tela final onde você fornece o resultado do seu teste de QI. Voce escolhe o melhor algoritmo para sua figura.**



## Ray tracing *simplificado ou aproximado ou*

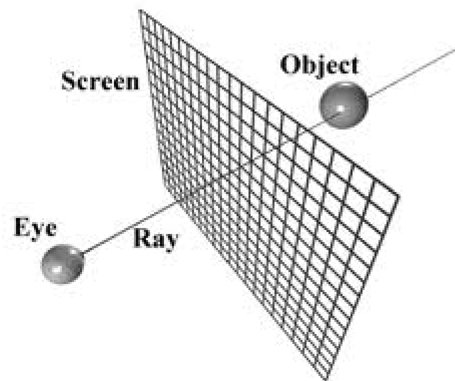
**Ray casting** lança raios a partir do observador de forma a perceber a distância dos objetos que compõem a cena.

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.

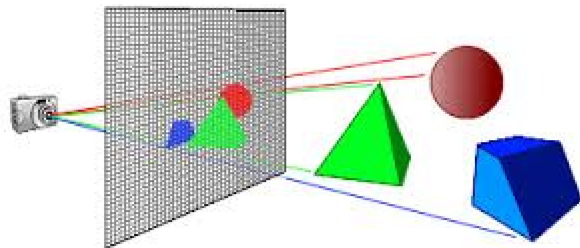
O calculo das interseções é o ponto chave do algoritmo.



# Ray casting

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

Veremos mais detalhes depois de falar sobre o sombreamento nas próximas aulas.



# 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.