

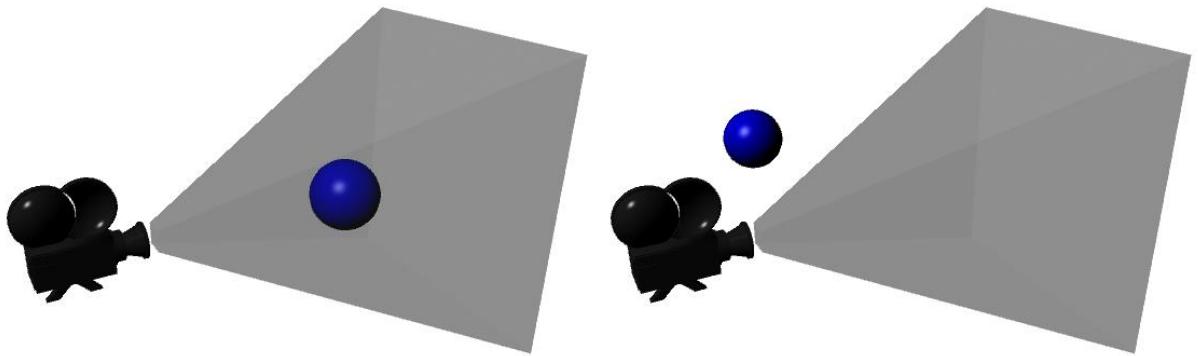
# Optimización

TC2008B Modelación de  
Sistemas Multiagentes con  
Gráficas Computacionales

# Optimización

En gráficas computacionales es necesario conocer aquellas superficies visibles y aquellas que no lo son para:

- ❖ Evitar dibujar las superficies no visibles.



# Superficies no visibles

¿Por qué sería una superficie no visible?

Por eficiencia, es necesario evitar dibujar este tipo de superficies.

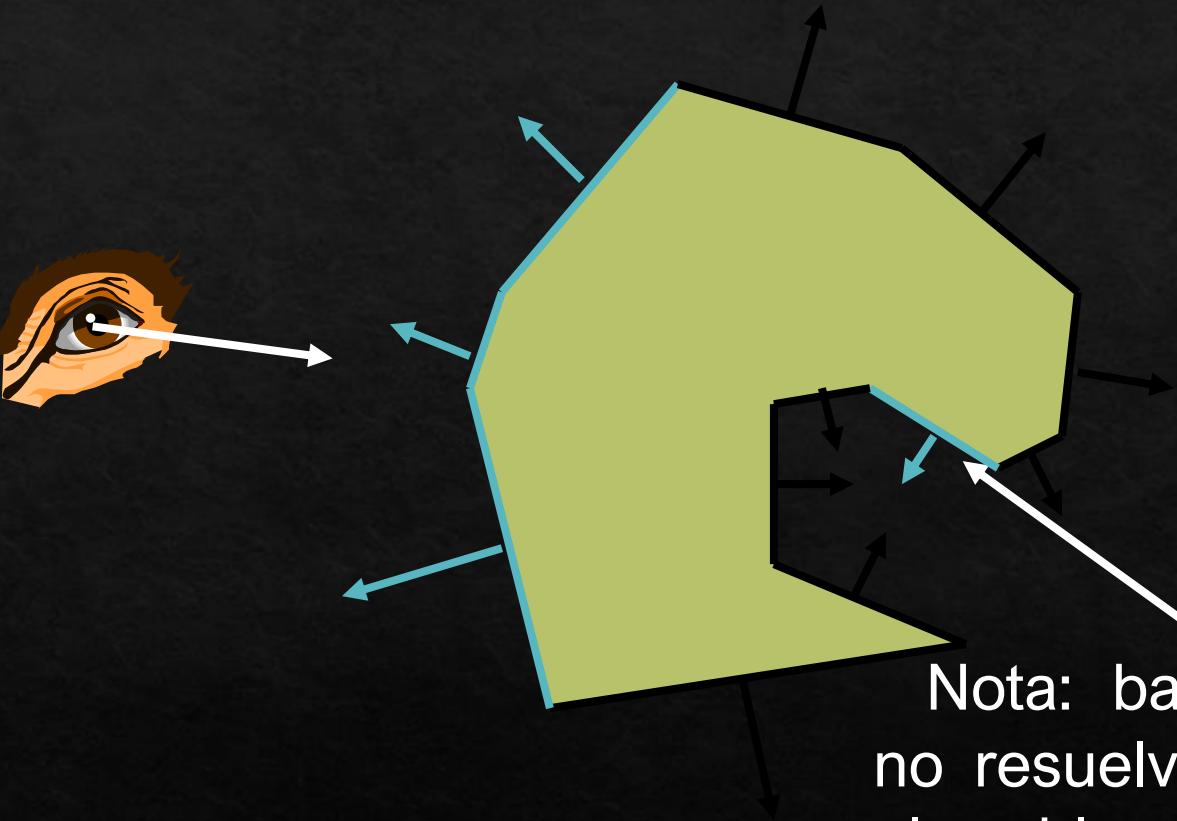
Polígono “ve” hacia atrás (back facing).

Polígono fuera del campo de vista.

Polígono ocultado por objeto(s) más cercanos.

Demasiado pequeños para ser vistos.

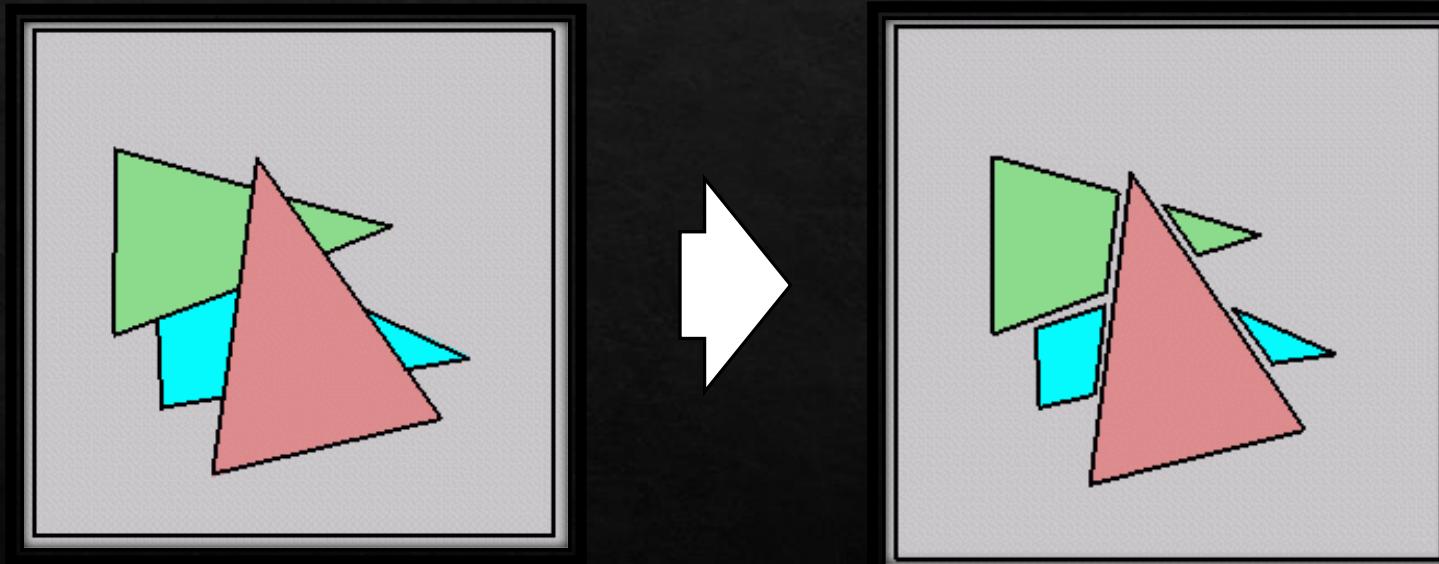
# Back face Culling



Nota: back face culling  
no resuelve por si mismo  
el problema de superficies  
ocultas!

# Oclusión

- ❖ En una escena se pueden encimar polígonos:



# Oclusión

## Algoritmo del Pintor

- ❖ La idea es pintar los polígonos tal como lo hace un pintor:
  - ❖ Dibujar los polígonos de atrás hacia adelante.
- ❖ ¿Funcionará Siempre?
  - ❖ No:
    - ❖ Polígonos que se intersectan.
    - ❖ Ciclos sin orden de visibilidad.



# Algoritmos analíticos de visibilidad

De finales de los años 60 a finales de los 70 hubo intenso interés en encontrar algoritmos eficientes para *quitar superficies ocultas*.



Dos de ellos:

Árboles BSP (*Binary Space-Partitioning*).

Algoritmo de Warnock.

# Árboles BSP

- ❖ Árbol BSP: organiza el espacio (*es partición*) en un árbol binario:
  - ❖ **Preproceso:** distribuye en un árbol binario los objetos de la escena.
  - ❖ **Runtime:** recorrer este árbol equivale a recorrer los objetos de atrás hacia adelante.
  - ❖ **Idea:** divide el espacio recursivamente en medios-espacios escogiendo *planos de división*:
    - ❖ Estos planos pueden tener orientación arbitraria.

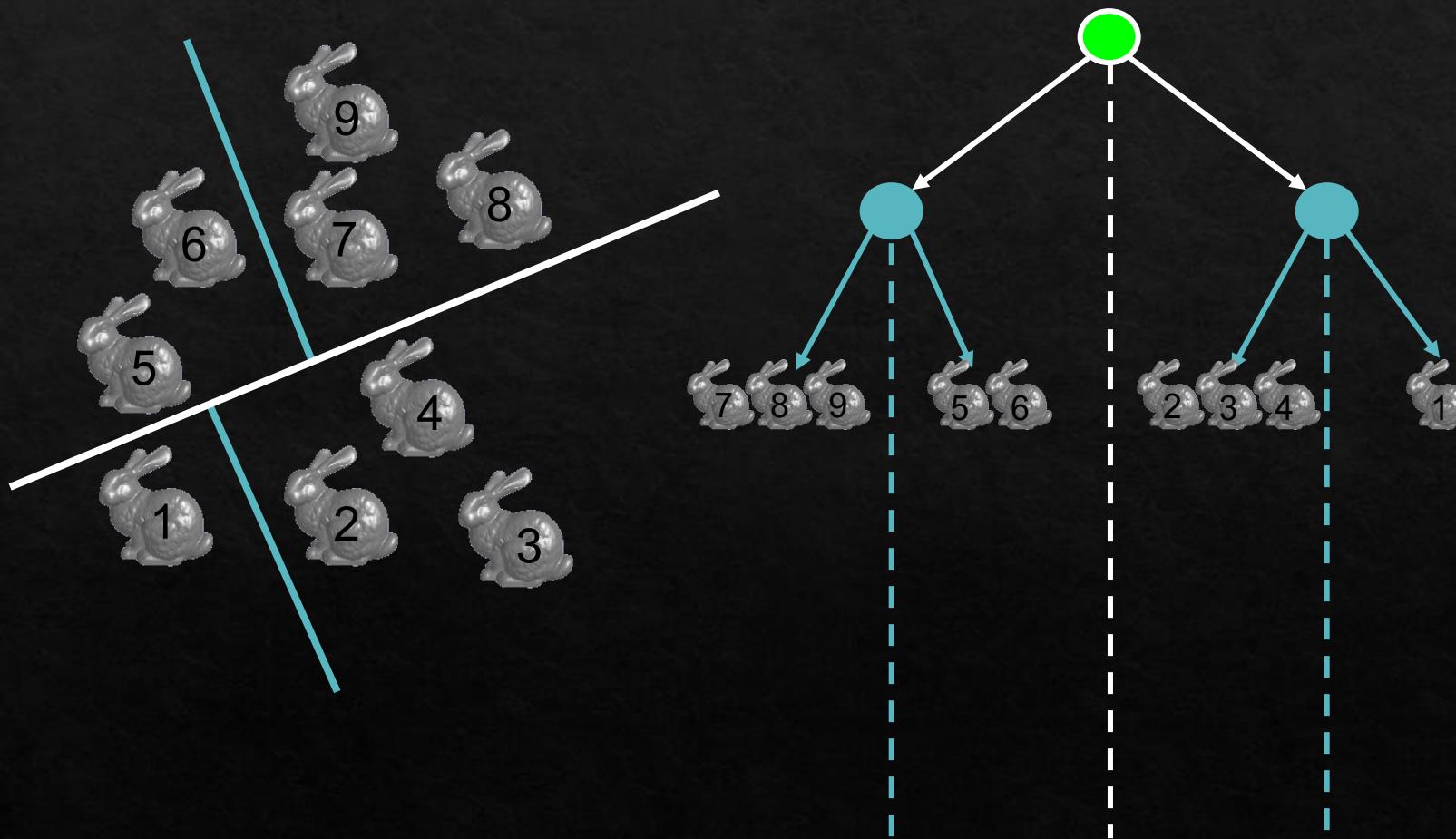
# Árboles BSP: Pre proceso



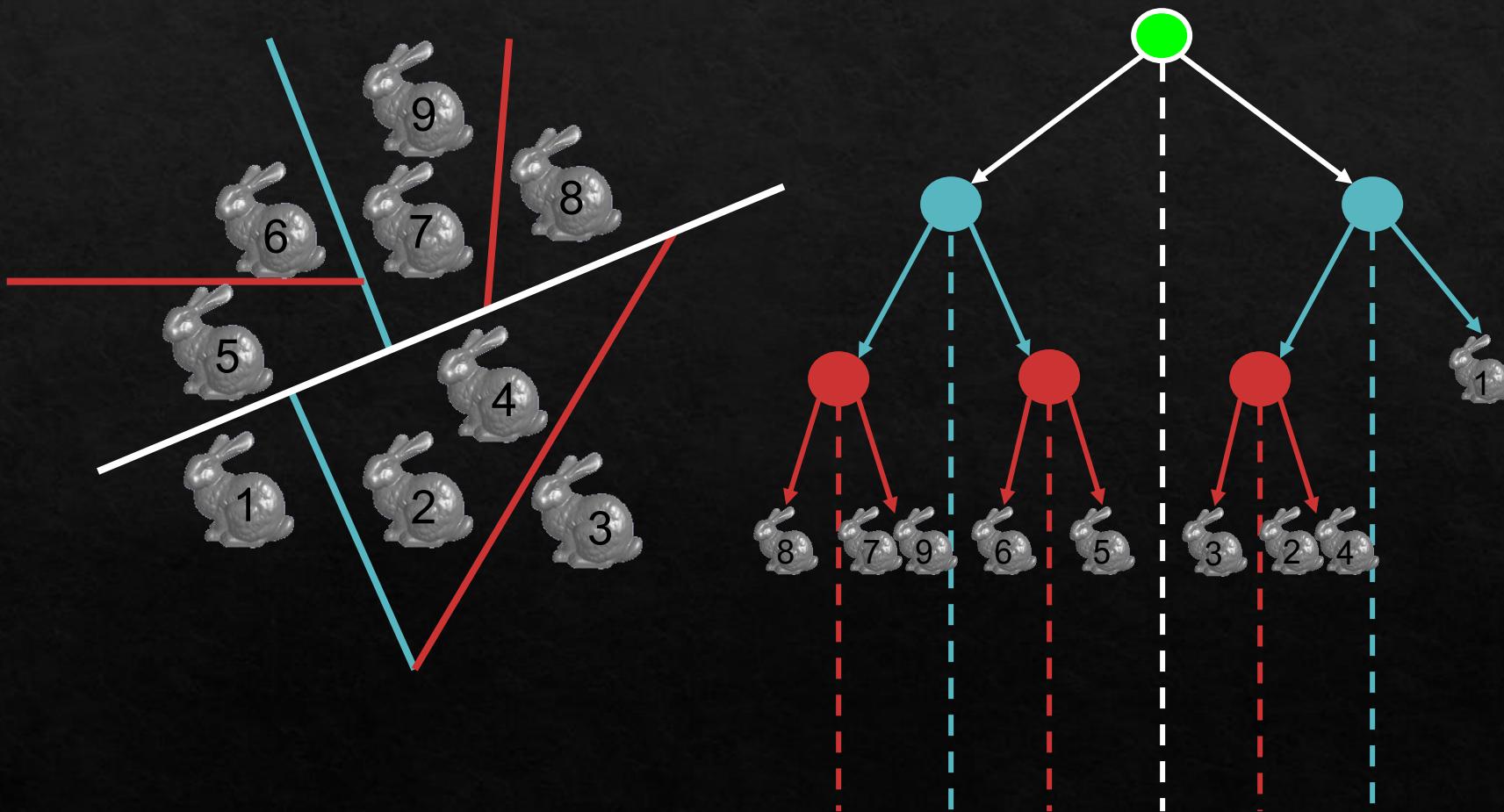
# Árboles BSP: Pre proceso



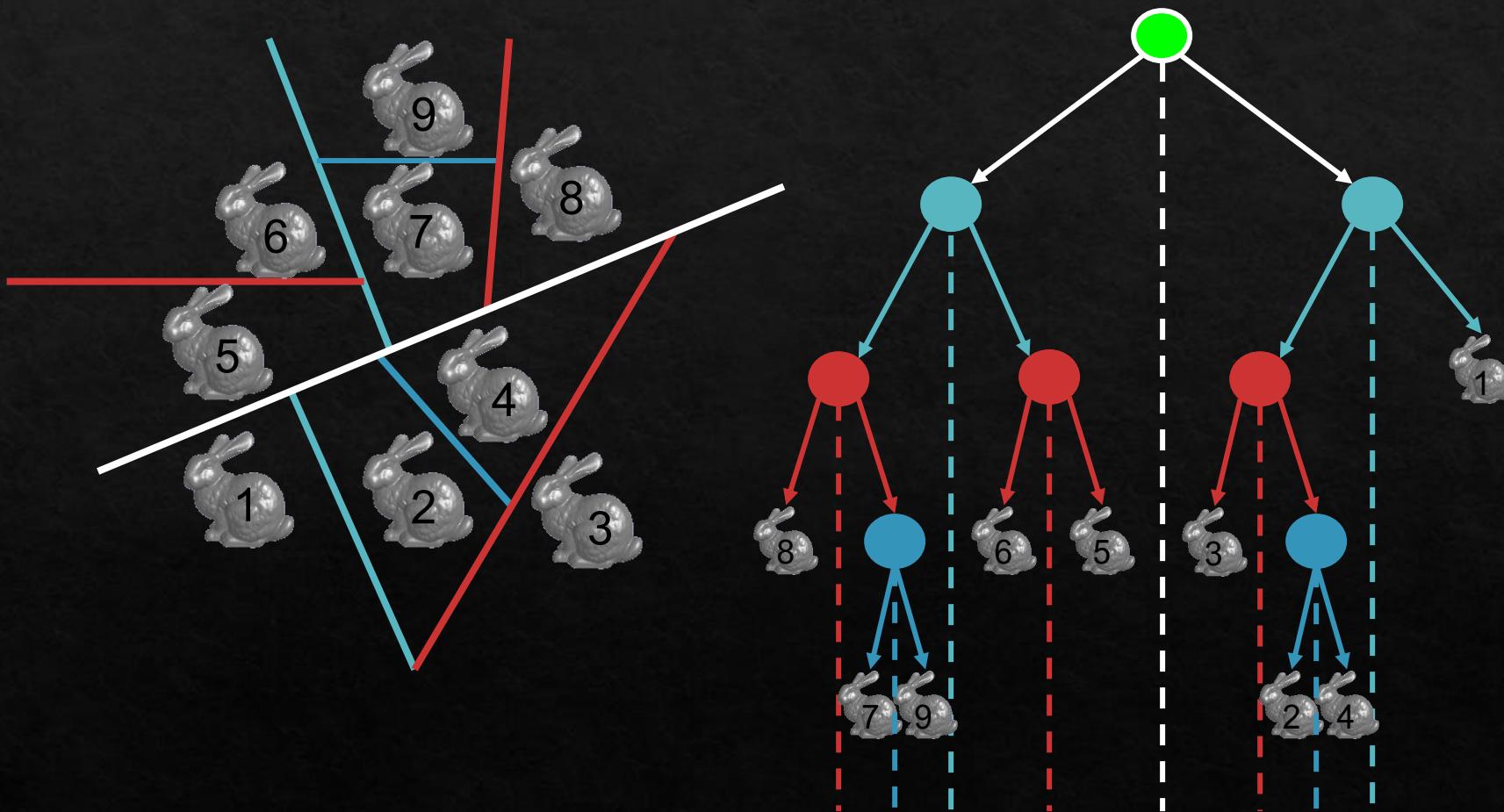
# Árboles BSP: Pre proceso



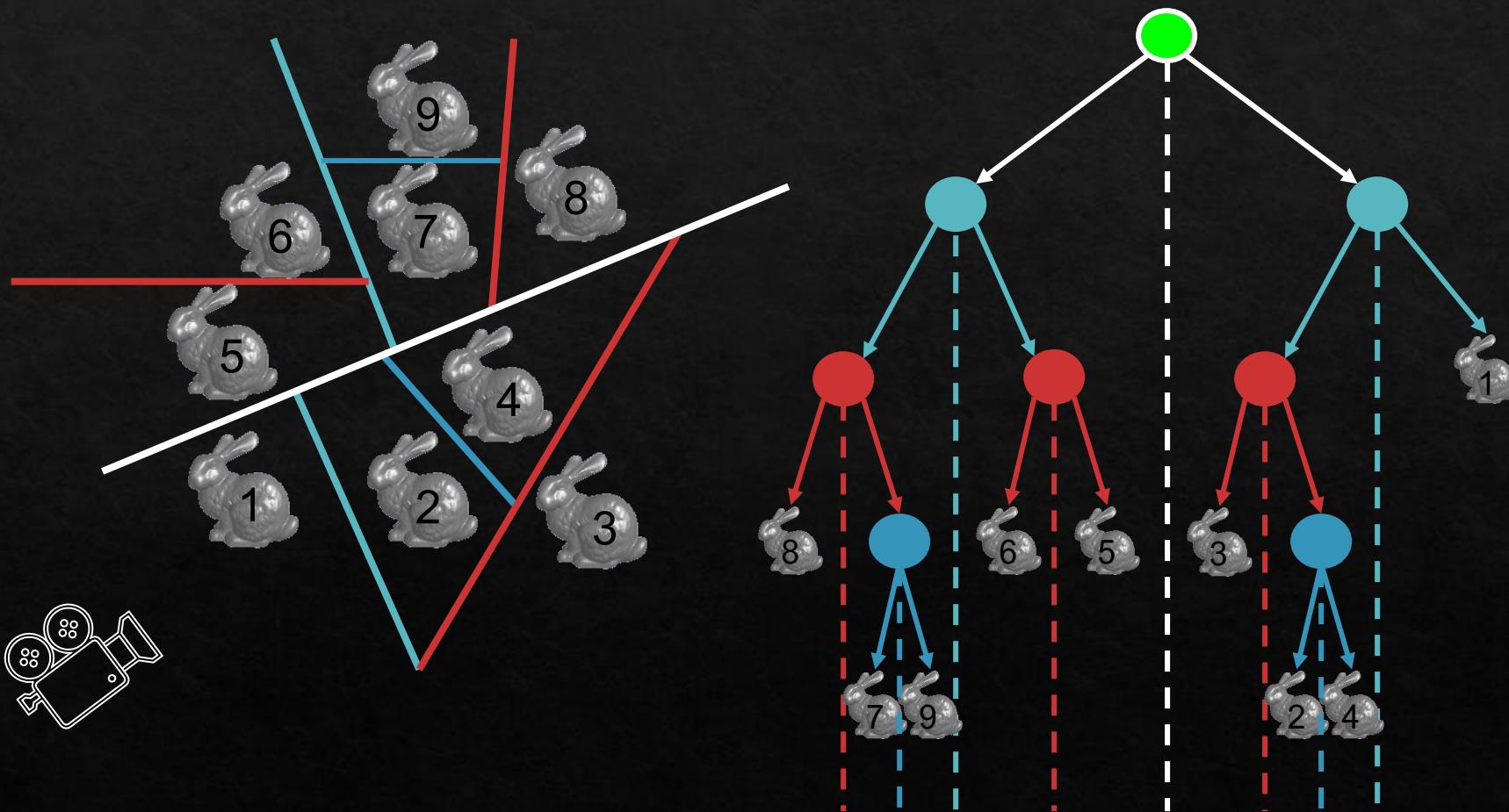
# Árboles BSP: Pre proceso



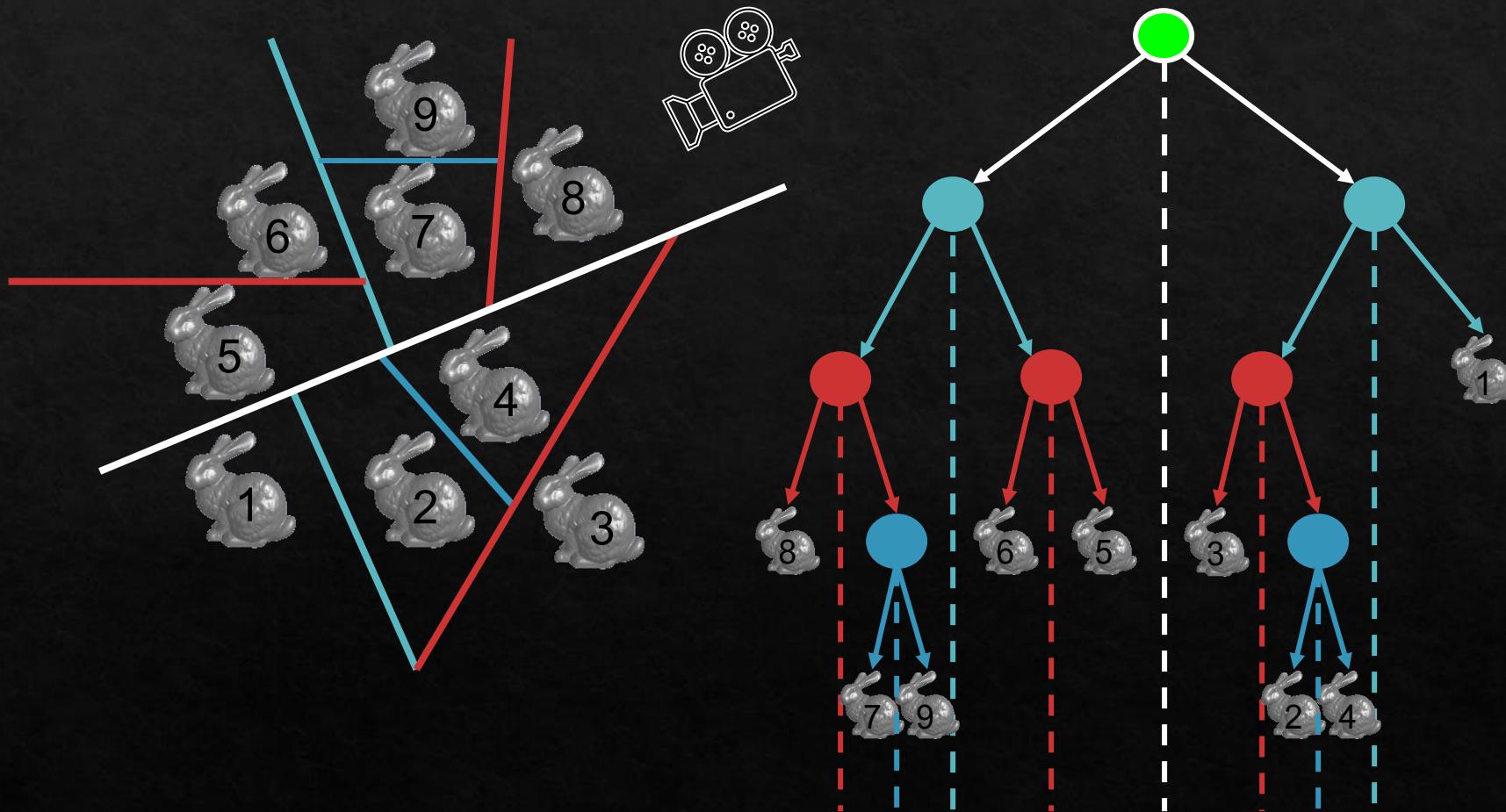
# Árboles BSP: Pre proceso



# Árboles BSP: Runtime

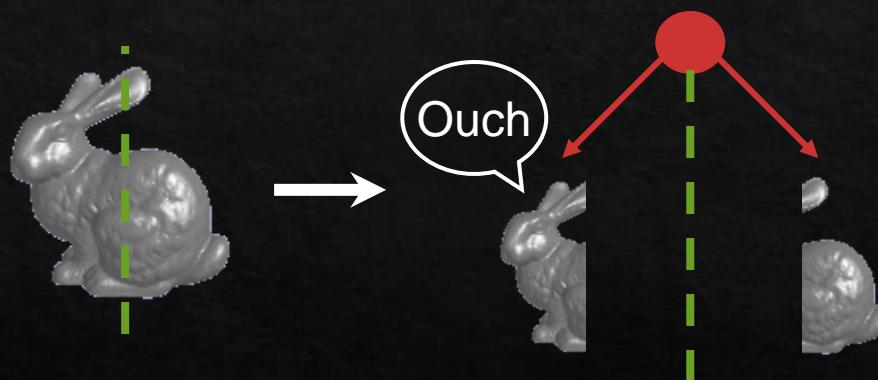


# Árboles BSP: Runtime



# Árboles BSP: Desventajas

- ❖ Pero ¿si un plano de división atraviesa un objeto?
  - ❖ Divide el objeto, da la mitad a cada nodo:

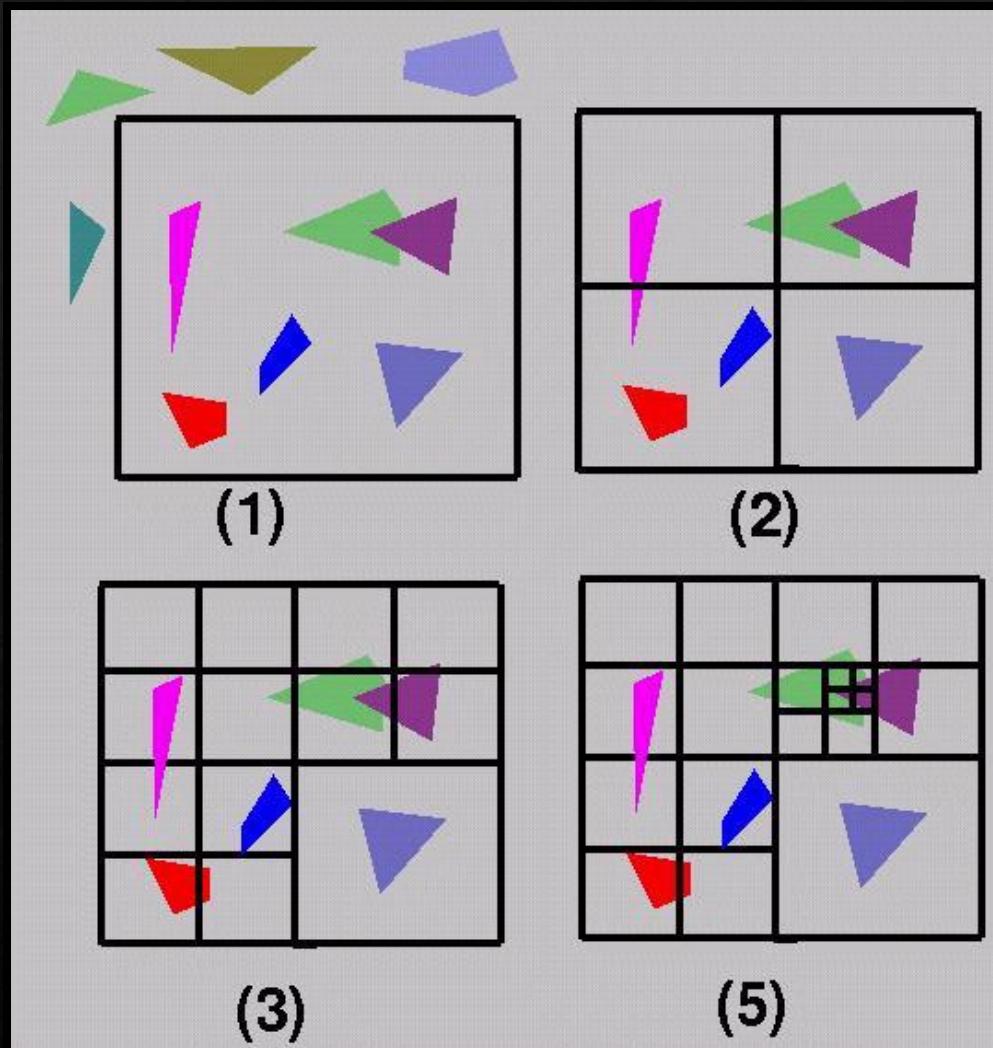


- ❖ Peor caso: pueden crear hasta  $O(n^3)$  objetos!

# Algoritmo de Warnock (1969)

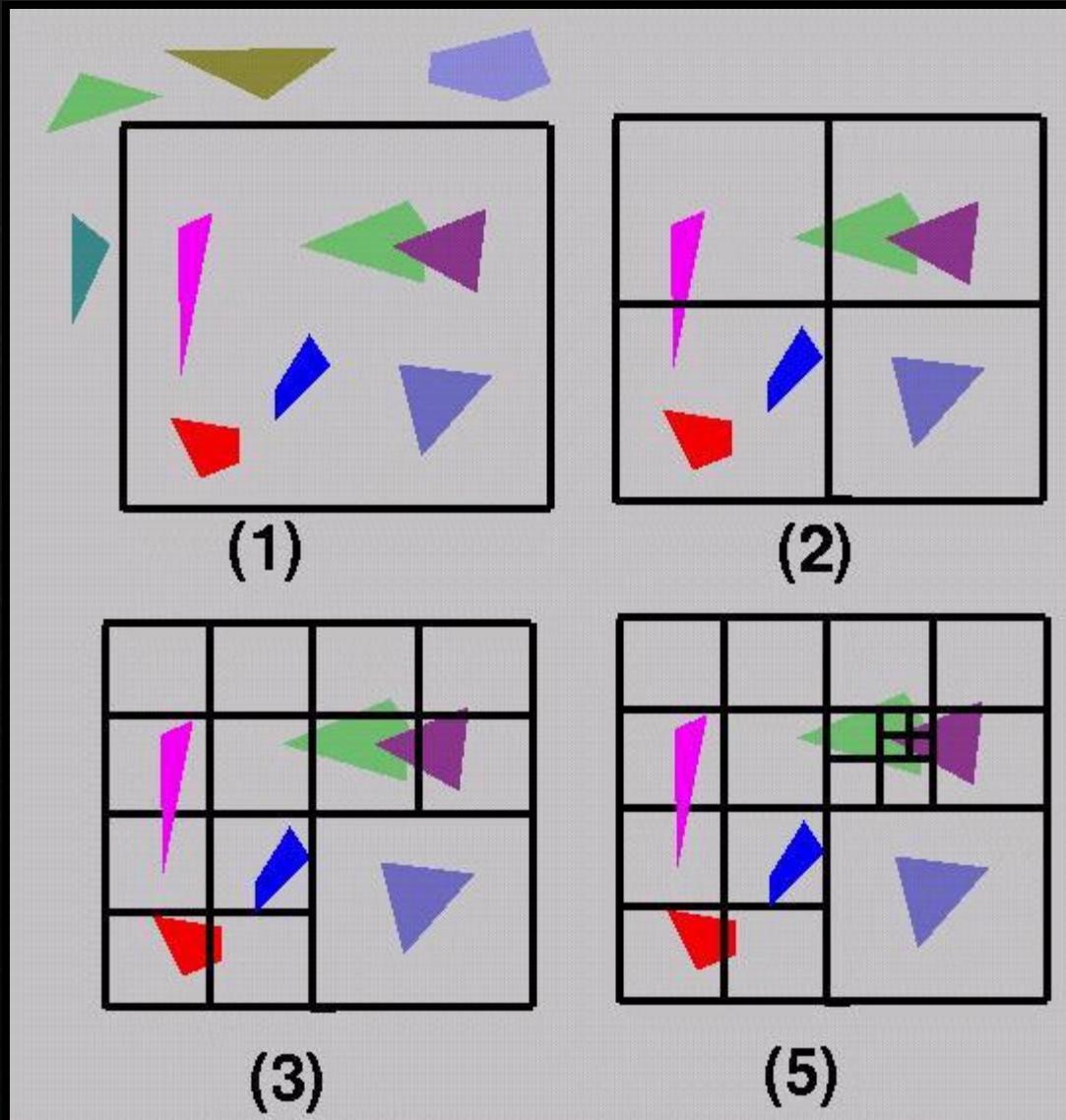
Elegante esquema basado en un enfoque general:

- ❖ *Si la situación es demasiado compleja, **subdivide**.*
- ❖ Comienza con un puerto de vista (viewport) raíz y una lista de todas las primitivas.
- ❖ Recursivamente:
  - ❖ Recorta objetos al viewport.
  - ❖ Si el número de objetos en el viewport es 0 o 1, visibilidad es trivial.
  - ❖ De otra manera, subdivide en viewports más pequeños, distribuye las primitivas entre ellos y prosigue recursivamente.

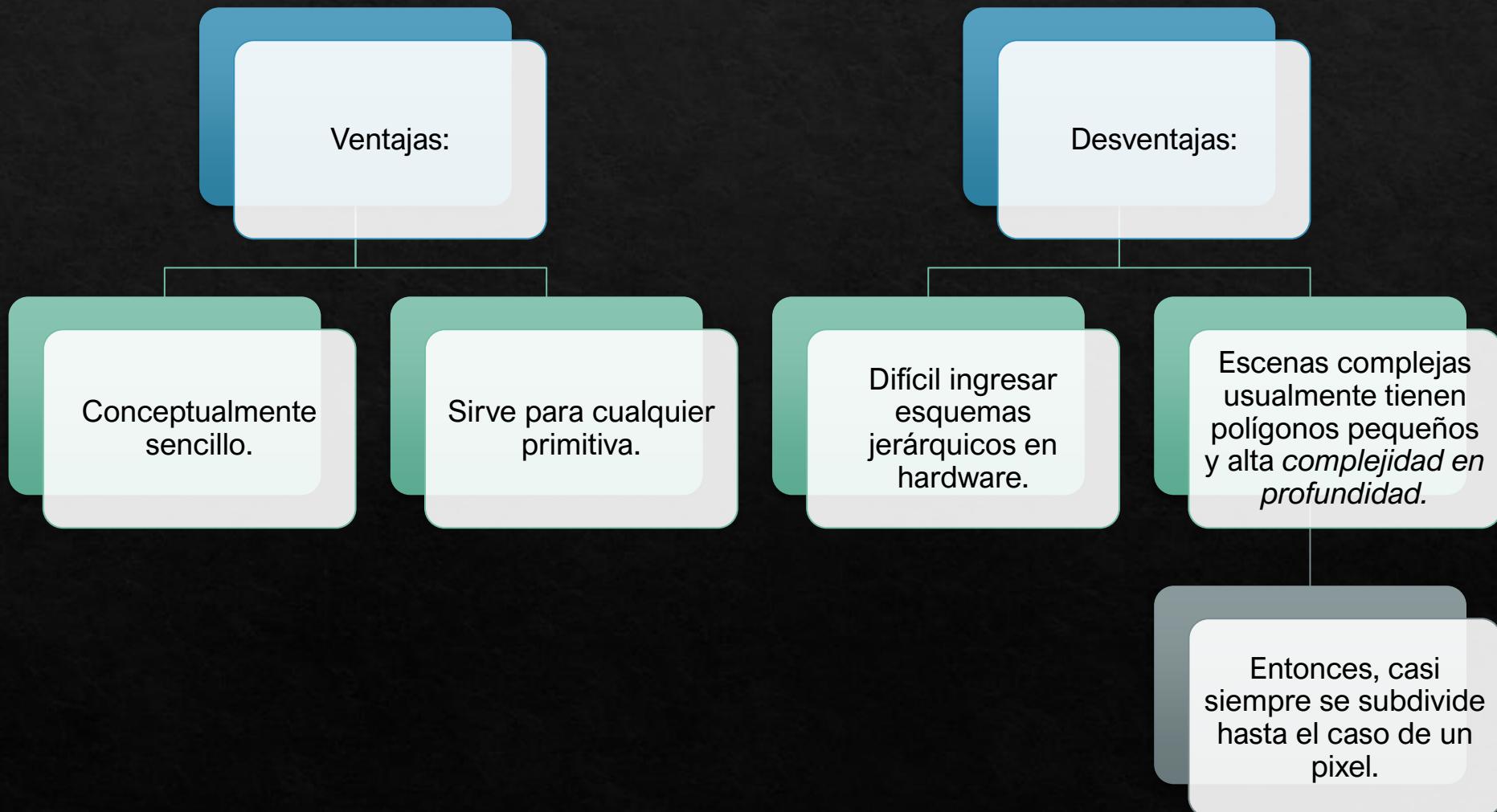


# Algoritmo de Warnock

- ❖ ¿Cuál es la condición de terminación?
- ❖ ¿Cómo determinar la superficie visible correcta en este caso?



# Algoritmo de Warnock

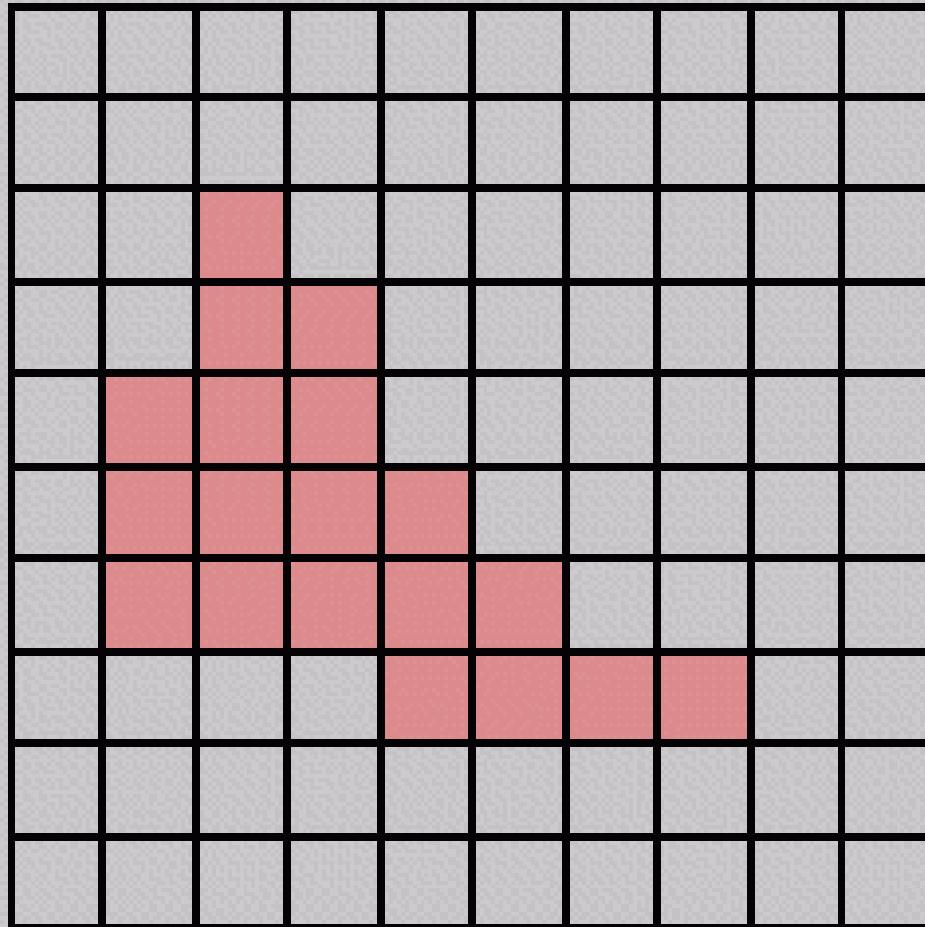


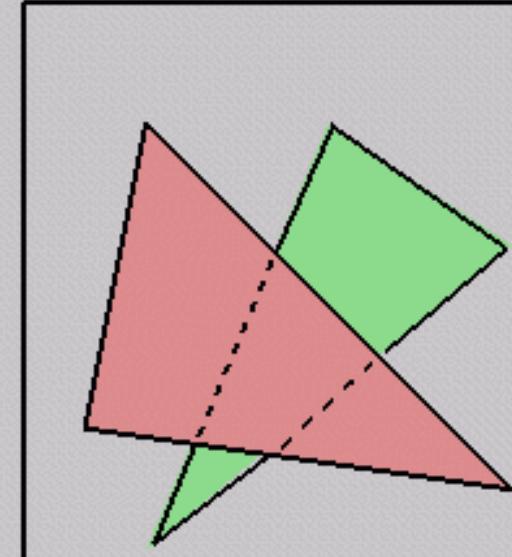
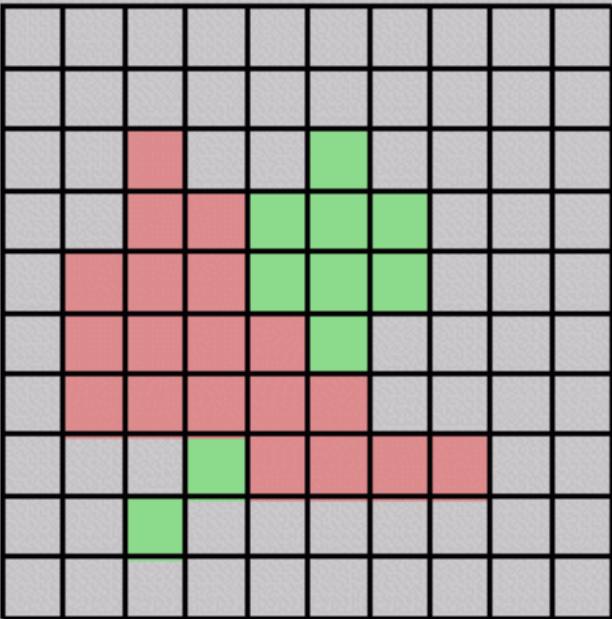
# Algoritmo de Z-Buffer

- ❖ Árboles BSP y algoritmo de Warnock se propusieron cuando la memoria era cara (económicamente):
  - ❖ Ejemplo: el primer framebuffer de 512x512 costaba más de USD \$50,000.
- ❖ Ed Catmull (mid-70s) propuso un enfoque nuevo y radical: ***z-buffering***.
- ❖ Idea: resolver visibilidad ***independientemente en cada pixel***.

# Algoritmo de Z-Buffer

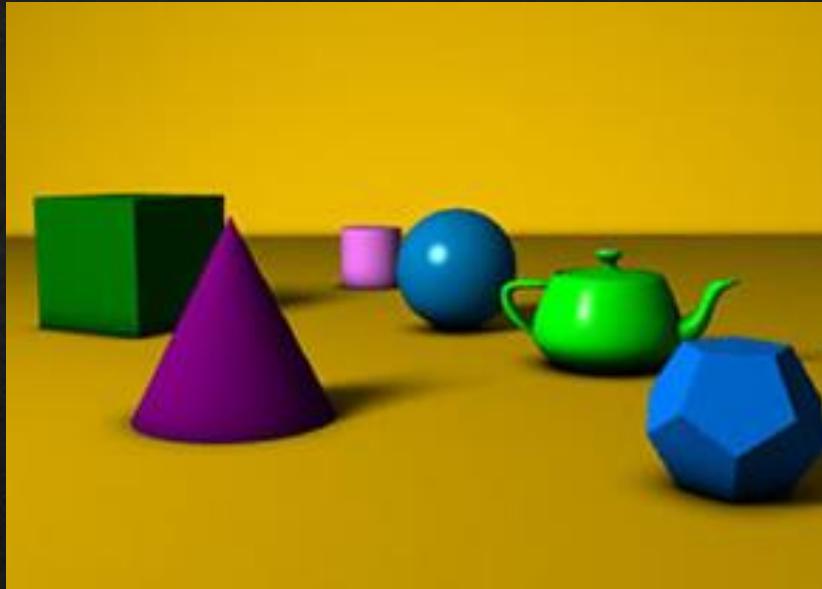
Ya sabemos cómo rasterizar polígonos en una imagen discretizada en pixeles:





# Algoritmo de Z-Buffer

*Si varias primitivas ocupan el mismo pixel, ¿cuál le da el color?*



## Algoritmo de Z-Buffer

Idea: retener información de profundidad (Z en coordenadas del ojo) a través de la transformación de proyección.

# Algoritmo de Z-Buffer

- ❖ Agrega al framebuffer un buffer de profundidad (*Z-buffer*) que guarda valor de z en cada pixel:
  - ❖ Al principio del cuadro, inicializa las profundidades de los pixeles a  $\infty$ .
  - ❖ Al rasterizar interpola Z a lo largo del polígono y guarda el pixel en el Z-buffer.
  - ❖ Deja de escribir un pixel si su valor Z es mayor que el valor Z ya guardado allí.

# Z-Buffer Ventajas



Simple.



Fácil de implementar en hardware.



Polígonos pueden procesarse en orden arbitrario.



Maneja fácilmente  
interpenetración de  
polígonos.

Posibilita rasterizar parámetros  
de sombreado (normal a la  
superficie) y sombrear solo los  
fragmentos visibles.

# Z-Buffer Desventajas



“Mucha” memoria (e.g. 1280x1024x32 bits).



Loop interno Read-Modify-Write requiere memoria rápida.



No permite antialiasing.

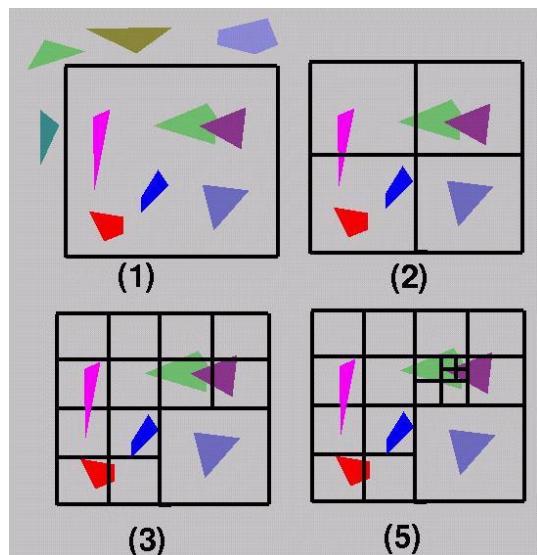
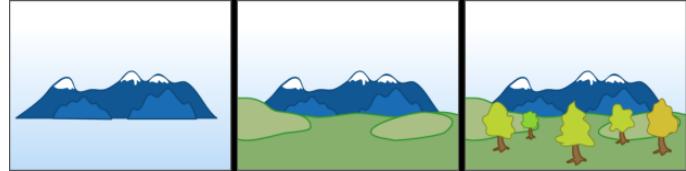


Difícil simular polígonos translúcidos.



Problemas de precisión (centelleo que empeora en proyección de perspectiva).

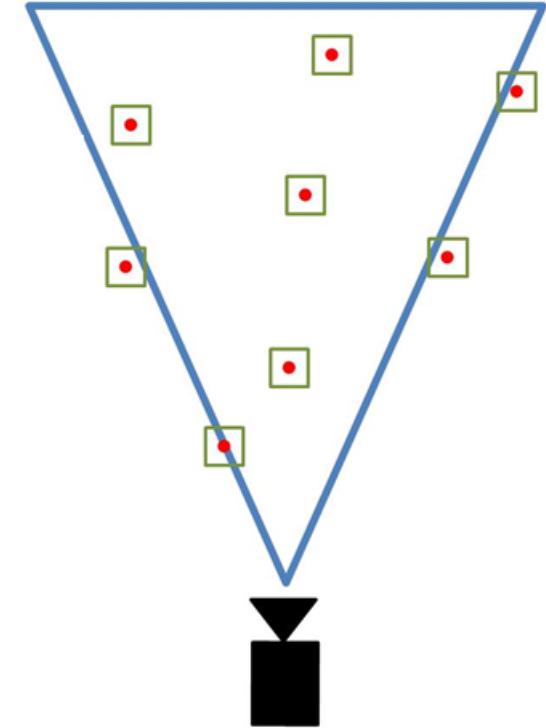
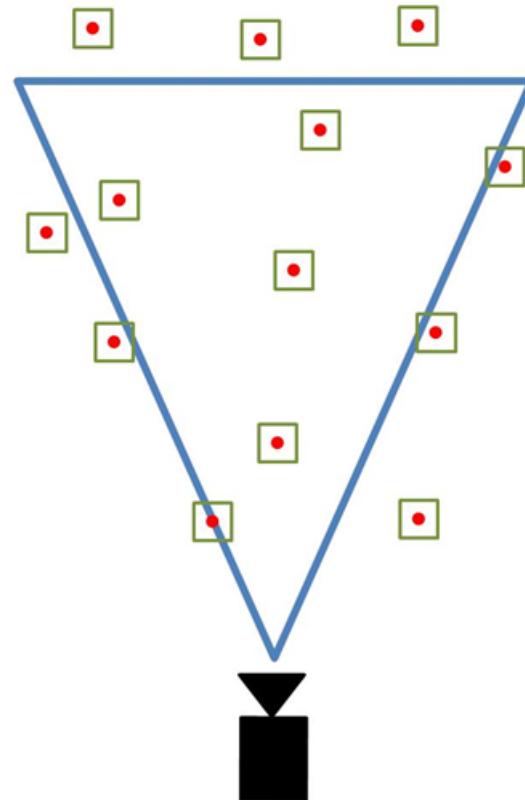
# Repaso



- ❖ Hasta ahora hemos visto eliminación:
  - ❖ Polígono “ve” hacia atrás (***backface culling***).
  - ❖ **Polígono ocluido** por objeto(s) más cercanos:
    - Algoritmo del Pintor
    - Árboles BSP
    - Algoritmo de Warnock
    - Algoritmo de Z buffer
- ❖ Frustum = pirámide recortada.

## View Frustum Culling (VFC)

La siguiente técnica consiste en no dibujar lo que está fuera del **view frustum** para ahorrar recursos computacionales

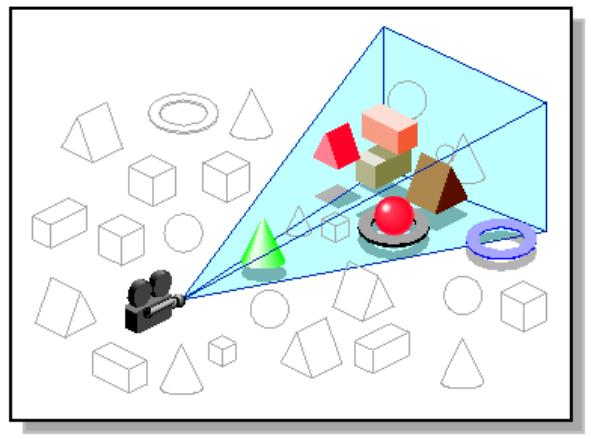
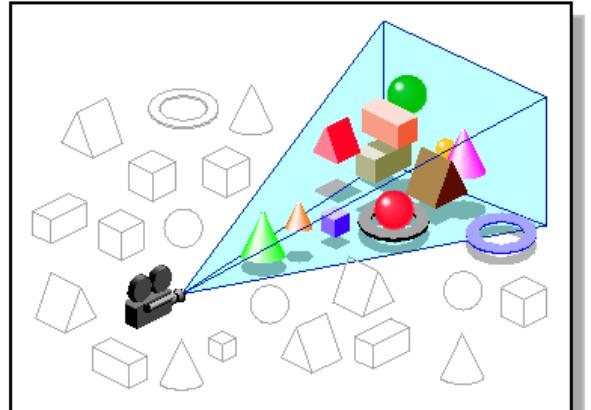
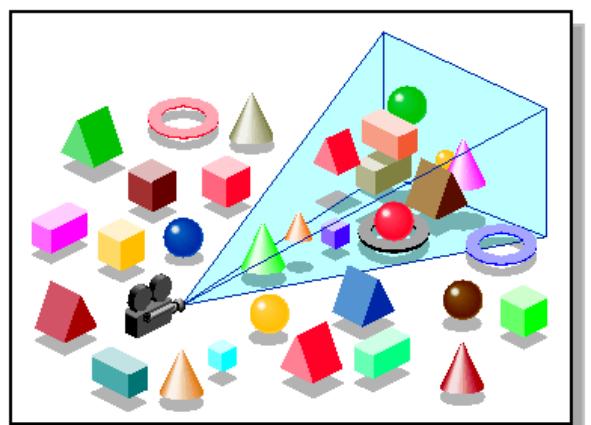




James H. Clark (fundó SGI) 1976

# Objetivo

- ❖ Eliminar **rápidamente** partes grandes de la escena que no se verán en la imagen final:
- ❖ No de manera exacta, pero una solución conservadora y rápida de cuáles primitivas pudieran ser visibles:
  - ◆ **Z-buffer & recorte** darán la solución exacta.
- ❖ Este estimado es el **Conjunto Potencialmente Visible**, o **PVS**.



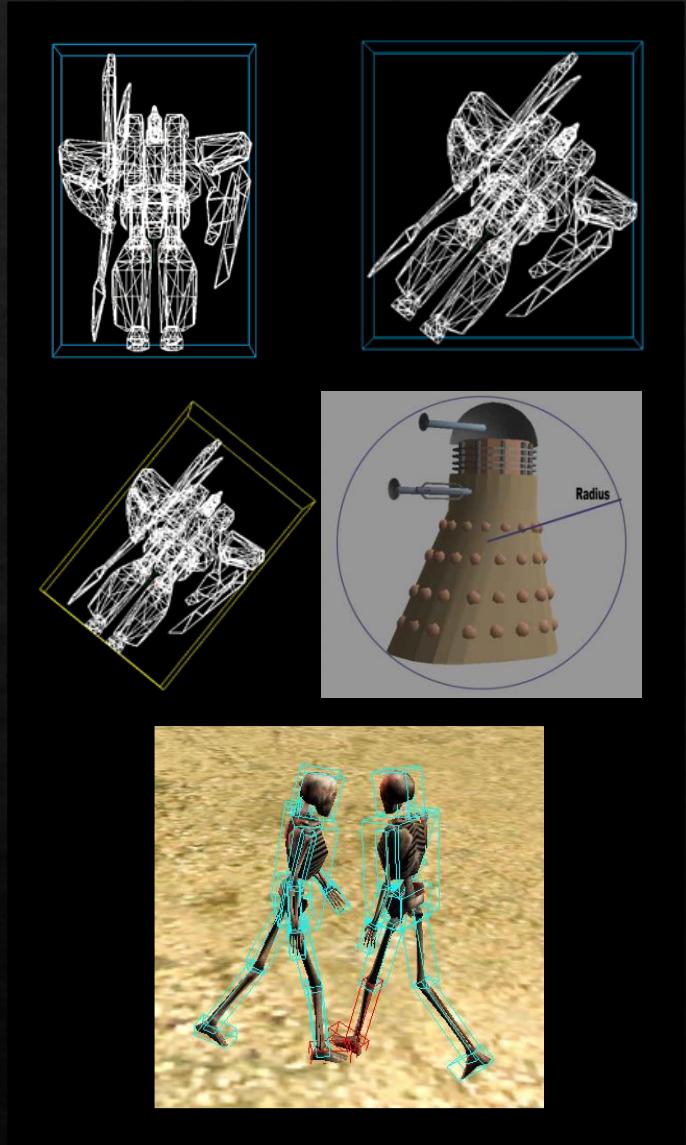
# View-Frustum Culling

- Agrupar primitivas en bloques.
- Simplifica los objetos con su **volumen acotante**.
- Antes de dibujar las primitivas del bloque, prueba su **Volumen Acotante (BV)** contra el **frustum**:
  - Si el bloque está por completo fuera del **frustum**, no dibuja ninguna primitiva.
  - Si el bloque intersecta el **frustum** agrega el objeto al **Conjunto potencialmente visible (PVS)**.
  - Dibuja el **PVS**.

# View-Frustum Culling

¿De qué forma deben ser los **BV**?

- Esferas y cajas acotantes (**BBs**) alineadas con los ejes (**AABBs**): fáciles de calcular, pruebas baratas.
- BB orientadas (**OBBs**) eliminan la necesidad de recalcular una nueva caja acotante cuando la primitiva que contiene es rotada.
- Muchas propuestas, pero se sigue usando sobre todo esferas o **AABBs**.



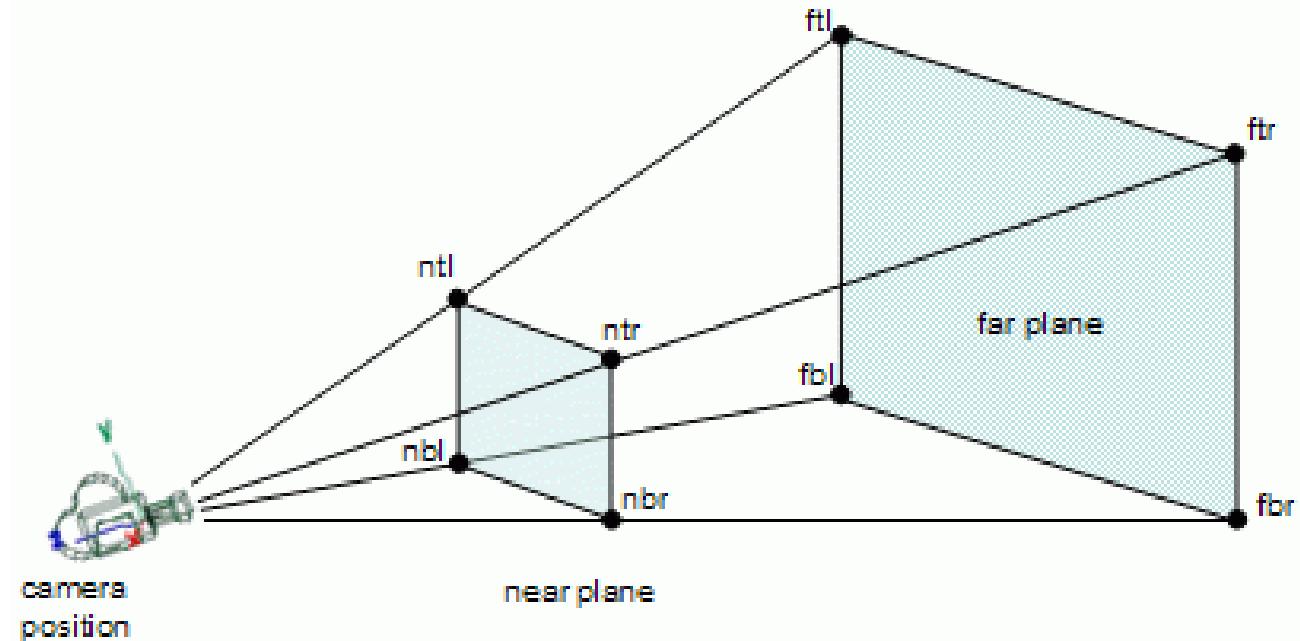
# View-Frustum Culling

Técnicas de **VFC**:

- ❖ Geométrico (**GVFC**).
- ❖ Radar (**RVFC**).
- ❖ Clip Space (**CSVFC**).

## VFC Geométrico

- ◊ Este enfoque consiste en calcular los seis planos del **frustum**.
- ◊ Verificar intersecciones entre el **BB** y los planos.



# VFC Geométrico

Inicialmente hay que calcular:

$$fc = p + d * \text{farDist}$$

$$ftl = fc + (\text{up} * H_{\text{far}}/2) - (\text{right} * W_{\text{far}}/2)$$

$p$  – posición de la cámara.

$d$  – dirección de la cámara.

$\text{nearDist}$  – distancia de la cámara al plano cercano.

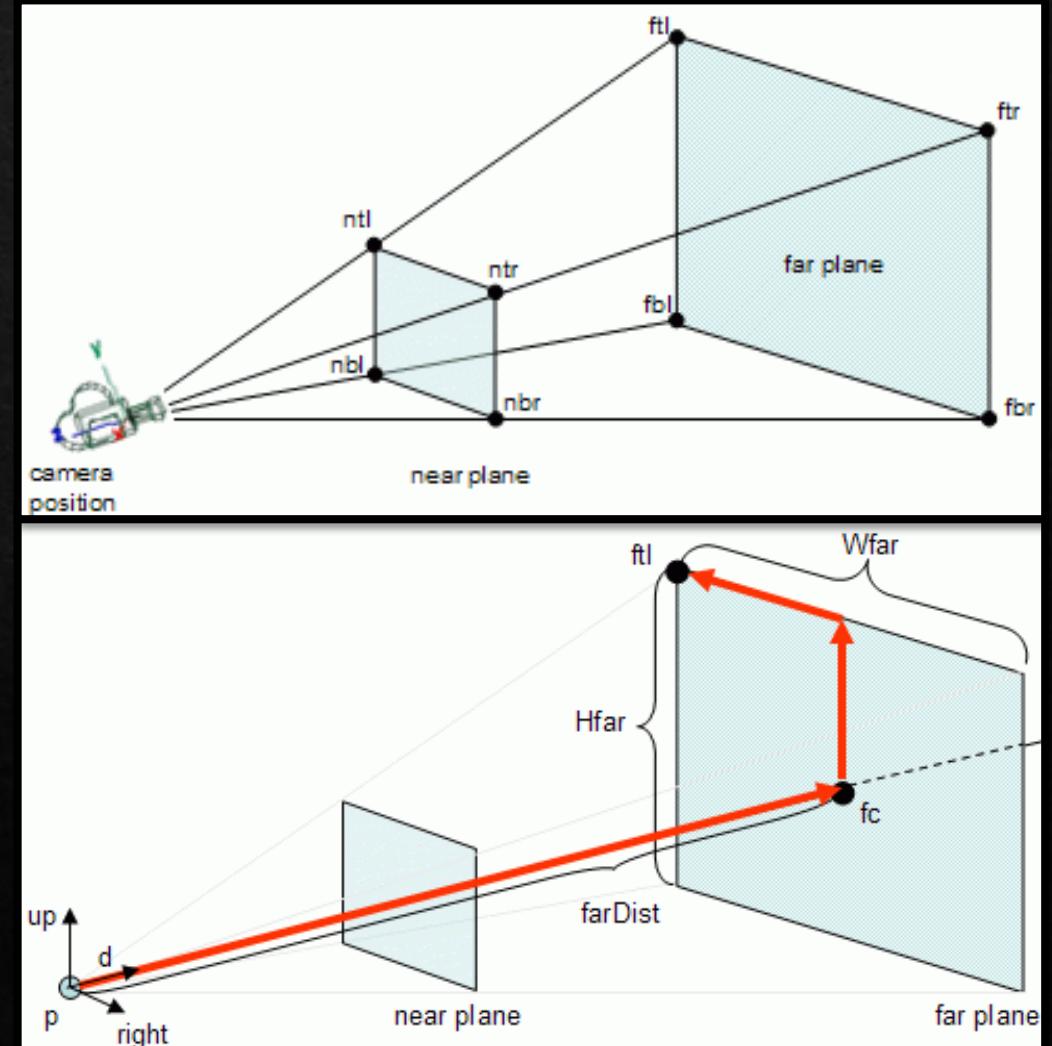
$H_{\text{near}}$  – altura del plano cercano.

$W_{\text{near}}$  – ancho del plano cercano.

$\text{farDist}$  - distancia de la cámara al plano lejano.

$H_{\text{far}}$  - altura del plano lejano.

$W_{\text{far}}$  - ancho del plano lejano.



# VFC Geométrico

$$ftr = fc + (up * Hfar/2) + (right * Wfar/2)$$

$$fbl = fc - (up * Hfar/2) - (right * Wfar/2)$$

$$fbr = fc - (up * Hfar/2) + (right * Wfar/2)$$

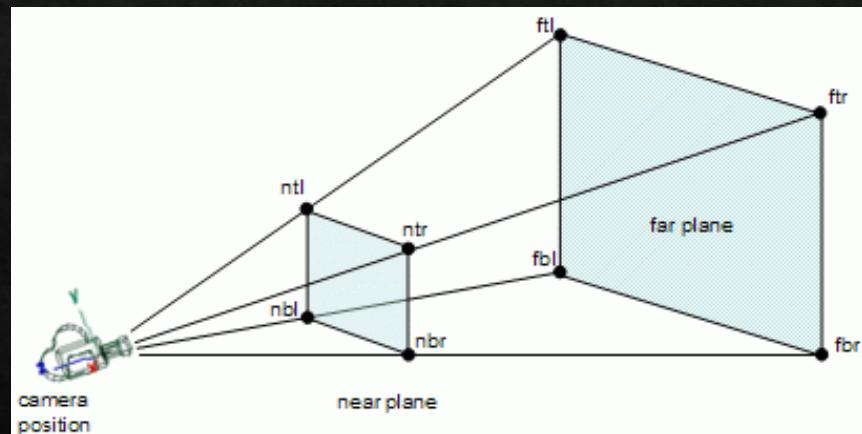
$$nc = p + d * \text{nearDist}$$

$$ntl = nc + (up * Hnear/2) - (right * Wnear/2)$$

$$ntr = nc + (up * Hnear/2) + (right * Wnear/2)$$

$$nbl = nc - (up * Hnear/2) - (right * Wnear/2)$$

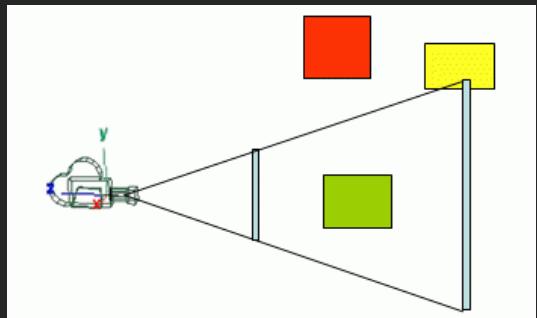
$$nbr = nc - (up * Hnear/2) + (right * Wnear/2)$$



# VFC Geométrico

- ❖ Una vez que tenemos los puntos del plano, ya podemos obtener su ecuación:
  - ❖ La Ecuación de un plano:
$$Ax + By + Cz + D = 0$$
  - ❖ Sean  $p_0, p_1, p_2$  tres puntos de un plano, se procede a calcular los coeficientes A, B, C y D:
    1. Calcular los vectores  $\bar{v} = p_1 - p_0$  y  $\bar{u} = p_2 - p_0$ .
    2. Calcular  $\bar{n} = \bar{v} \times \bar{u}$ .
    3.  $A = \bar{n}_x, B = \bar{n}_y, C = \bar{n}_z$ .
    4.  $D = -\bar{n} \cdot p_0$ .

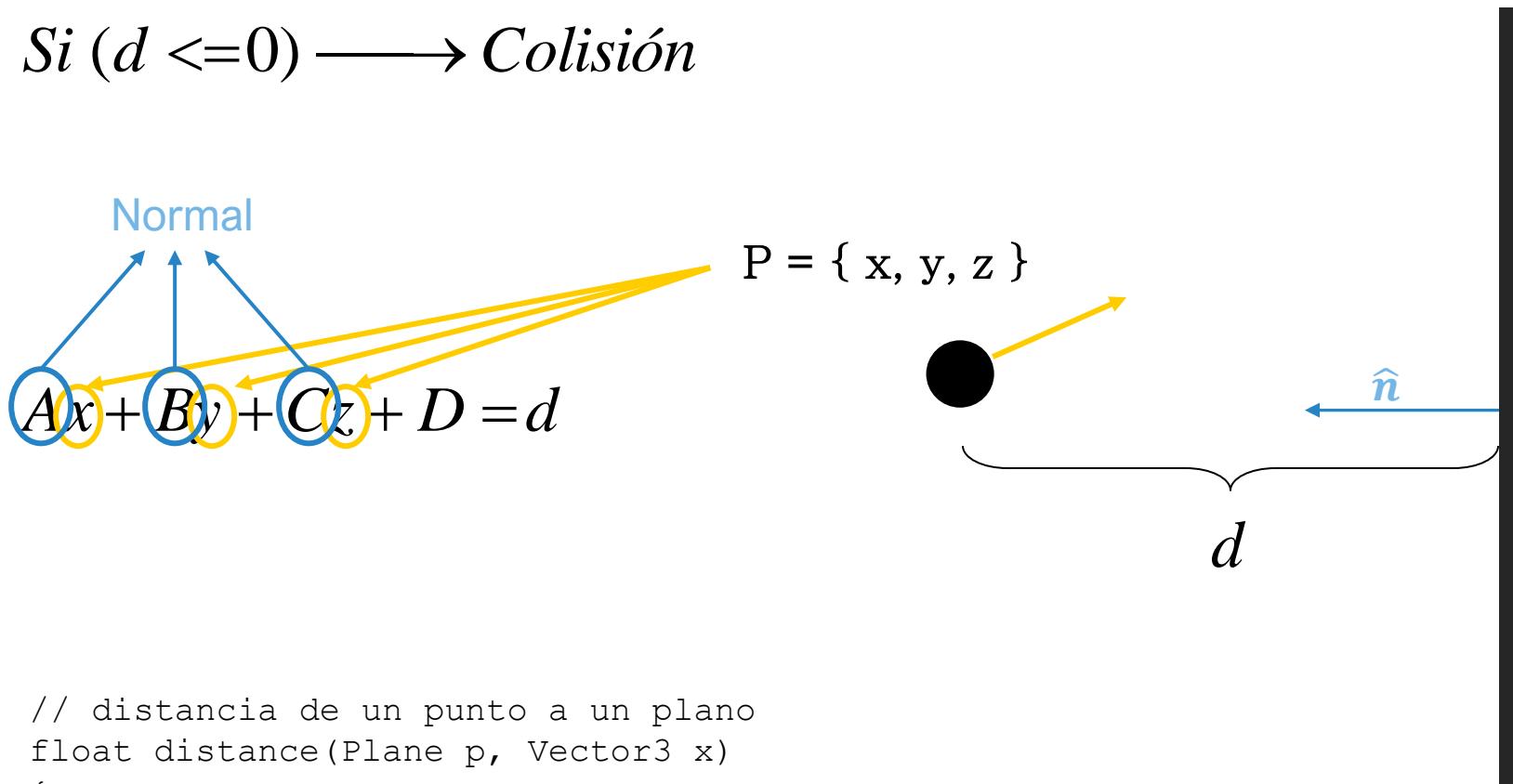
# VFC Geométrico



Finalmente cada uno de los puntos que conforman el **BB** es probado con los seis planos usando la ecuación de distancia de un punto al plano.

# Distancia de un punto a un plano

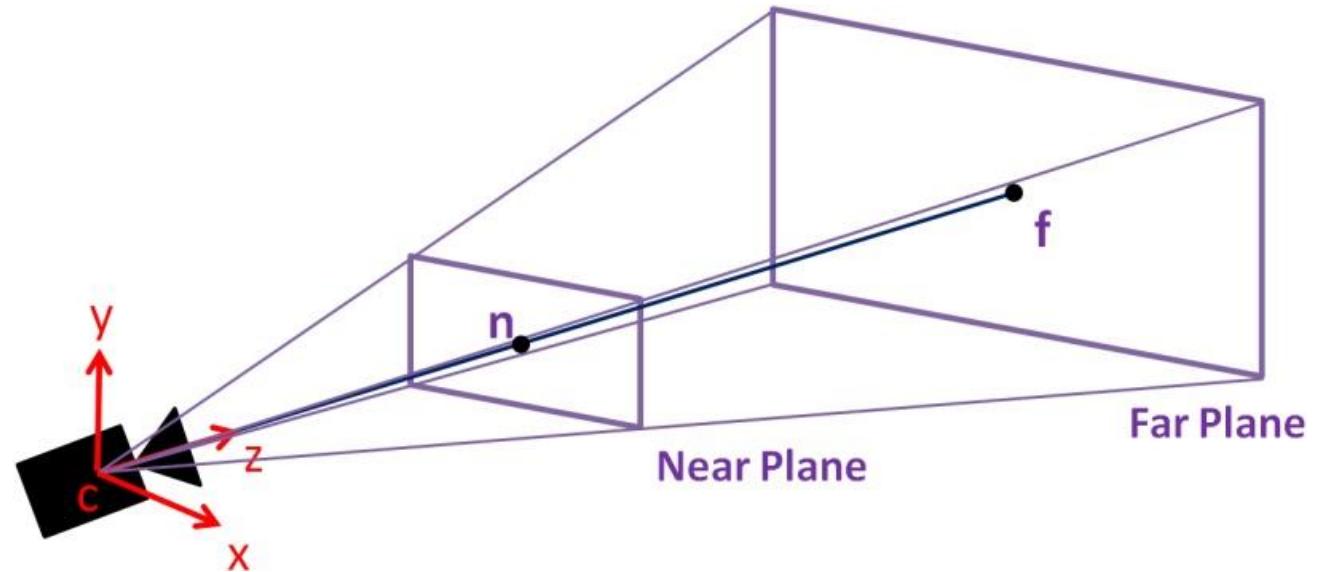
*Si ( $d \leq 0$ )  $\longrightarrow$  Colisión*



```
// distancia de un punto a un plano
float distance(Plane p, Vector3 x)
{
    return ( dot(p.normal, x) + p.distance );
}
```

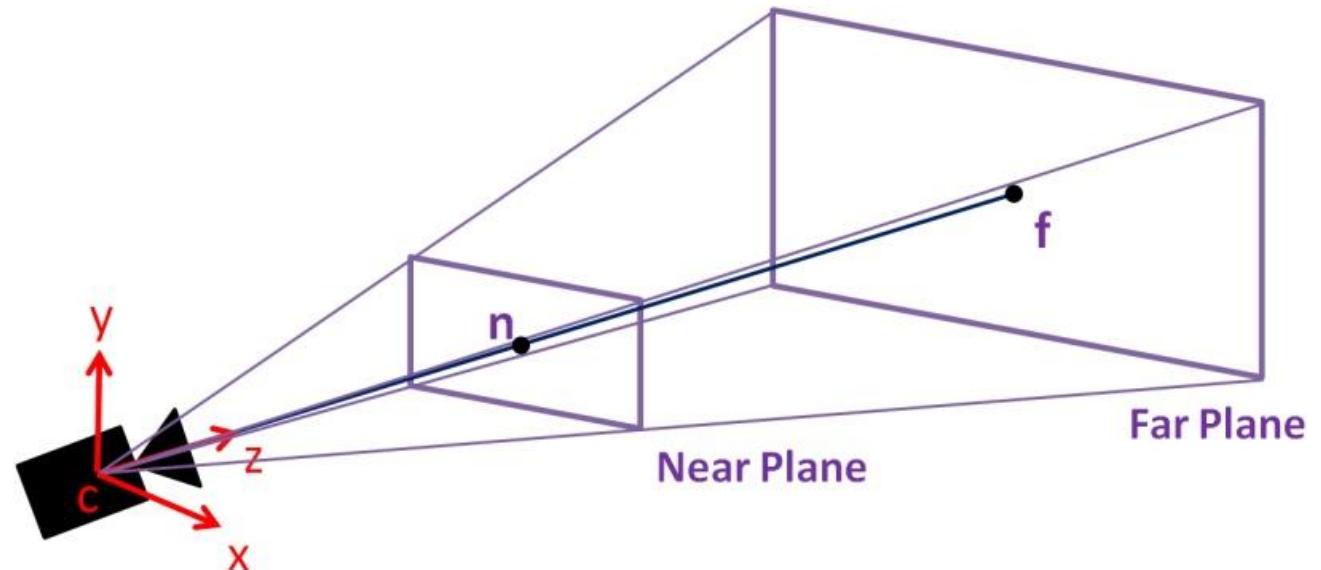
# Radar VFC

- ❖ Es más simple que el **VFC Geométrico**.
- ❖ Reduce el número de comparaciones necesarias para determinar si un objeto está fuera del **view frustum** a sólo tres.



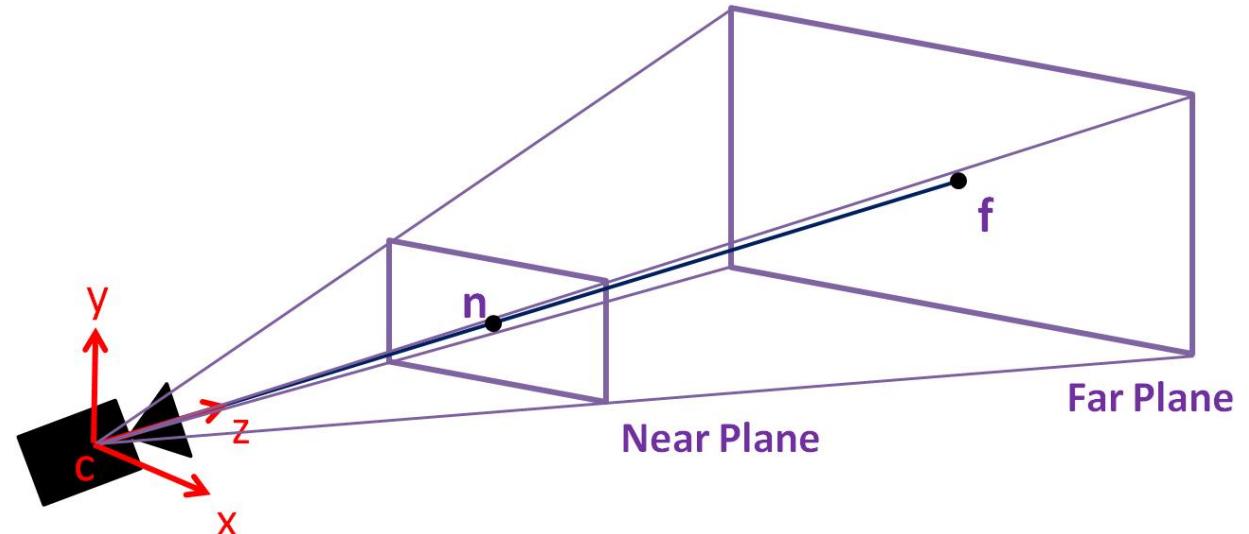
# Radar VFC

- ❖ El **BB** puede ser aproximado a una esfera o a un **punto**.
- ◆ **Radar VFC** compara si los objetos están o no en el rango de vista y está basado en los puntos referenciales de la cámara.



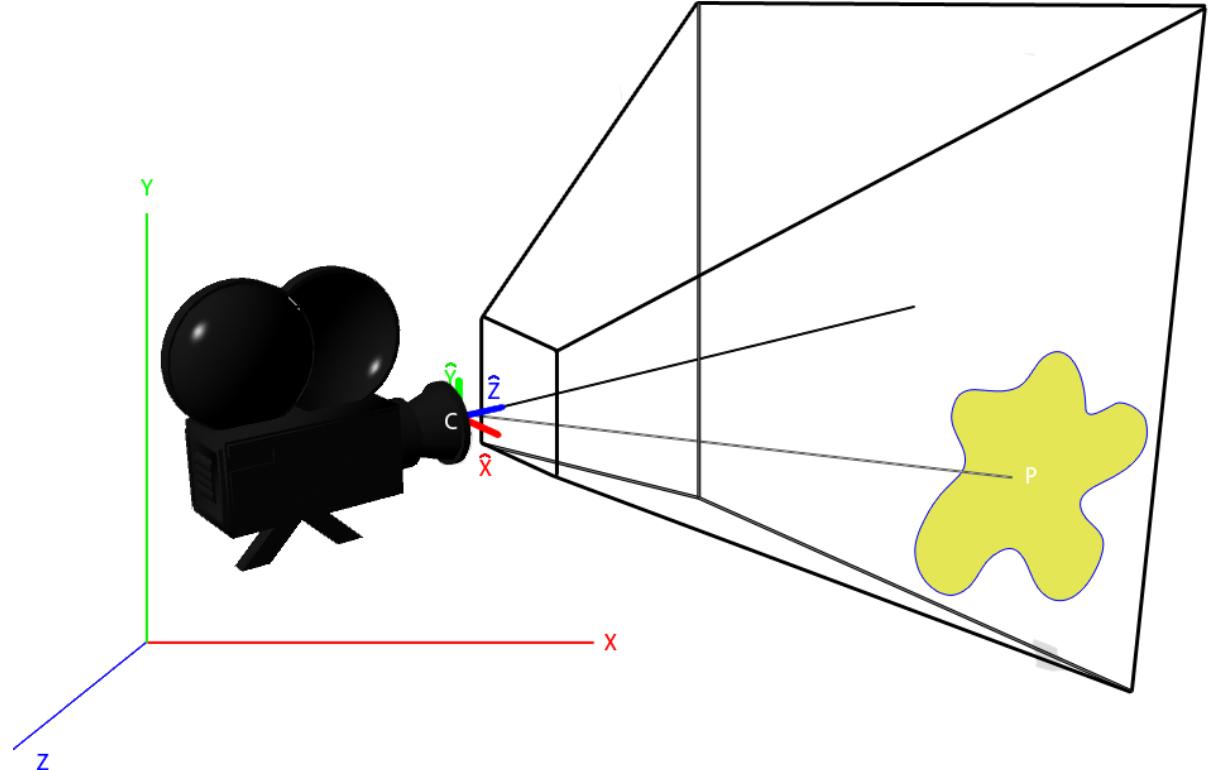
# Radar VFC

El referencial de la cámara  
está definido por los tres  
vectores unitarios  $(\hat{x}, \hat{y}, \hat{z})$ .



## Radar VFC

- ❖ La idea es encontrar las coordenadas del **BB** en el referencial y después usar estos datos para determinar si el punto está dentro o fuera el **view frustum**.



# Radar VFC

- ◆ El primer paso es hallar el referencial de la cámara, es decir, los vectores unitarios  $(\hat{x}, \hat{y}, \hat{z})$ :

$$\hat{z} = \frac{\bar{d}}{\|\bar{d}\|} = \left( \frac{d_x}{\sqrt{d_x^2 + d_y^2 + d_z^2}}, \frac{d_y}{\sqrt{d_x^2 + d_y^2 + d_z^2}}, \frac{d_z}{\sqrt{d_x^2 + d_y^2 + d_z^2}} \right)$$

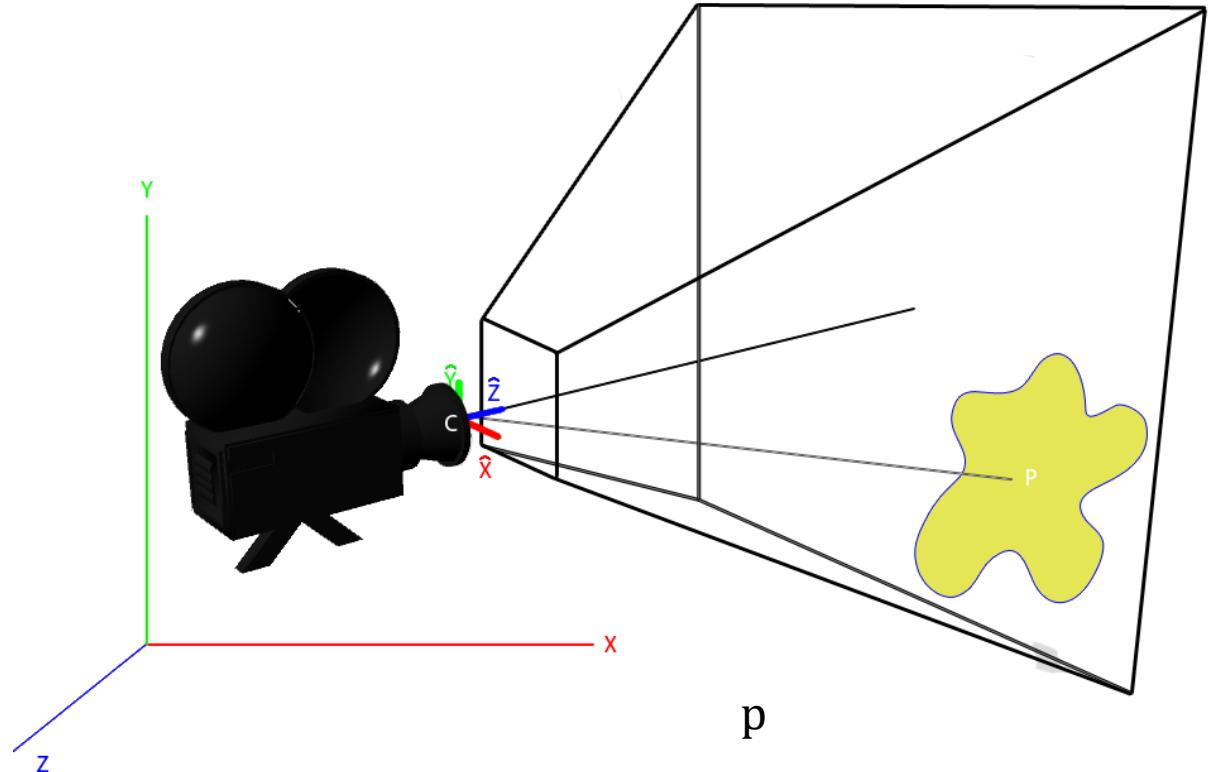
$$\hat{x} = \|\hat{z} \times \hat{u}\|$$

$$\hat{y} = \|\hat{x} \times \hat{z}\|$$

## Radar VFC

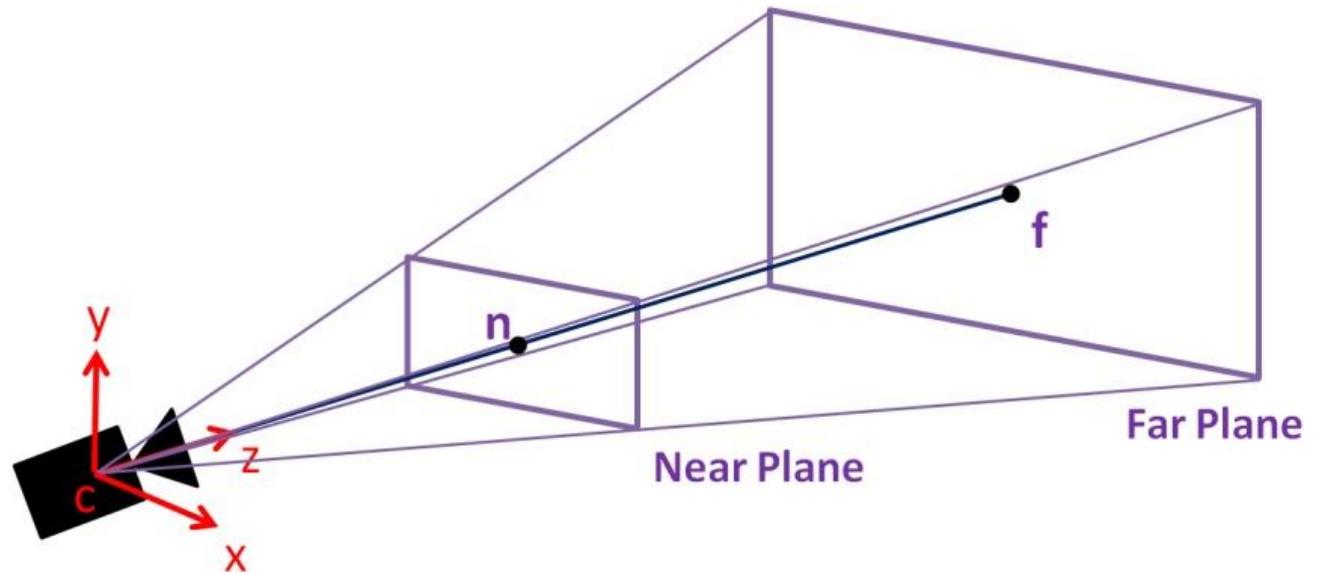
Después se calcula el vector  $\bar{v}$  formado por el **BB** y la posición de la cámara (c):

$$\bar{v} = p - c$$



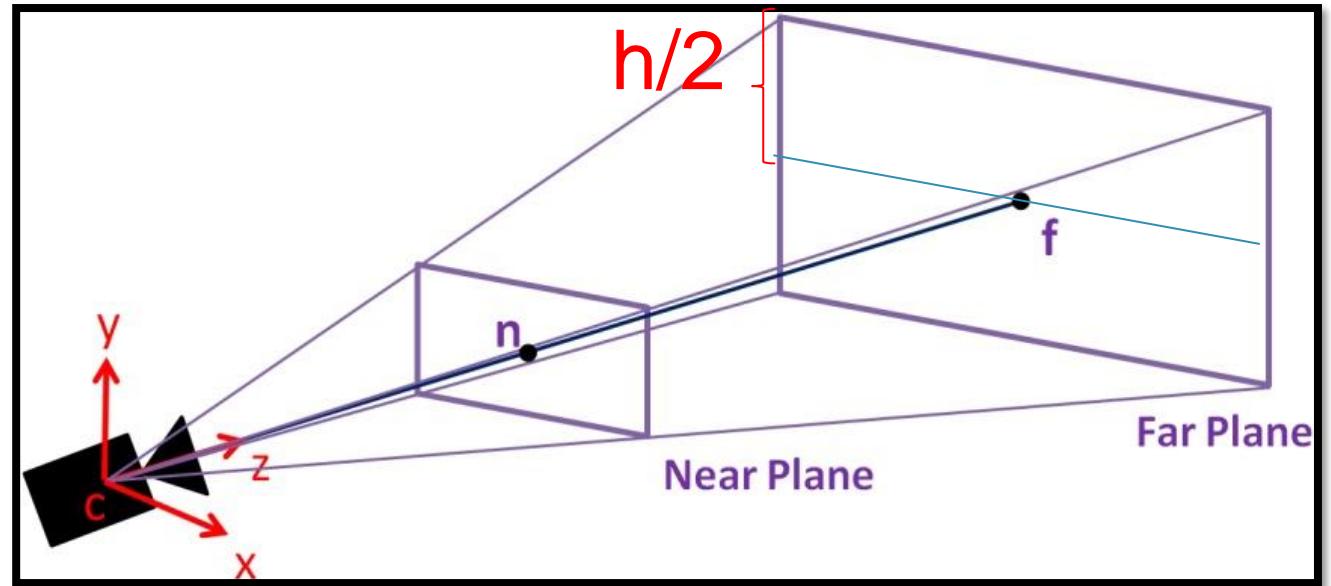
## Radar VFC

- ◆ Posteriormente se calcula la proyección de  $\bar{v}$  en el referencial de la cámara, es decir, se proyecta  $\bar{v}$  sobre los vectores unitarios  $(\hat{x}, \hat{y}, \hat{z})$ .
- ◆ La primera proyección (usar producto punto) se lleva a cabo con el vector  $\hat{z}$ :
- ◆ Si  $proy < nearD \text{ || } proy > farD$  el objeto está fuera del **frustum**.



## Radar VFC

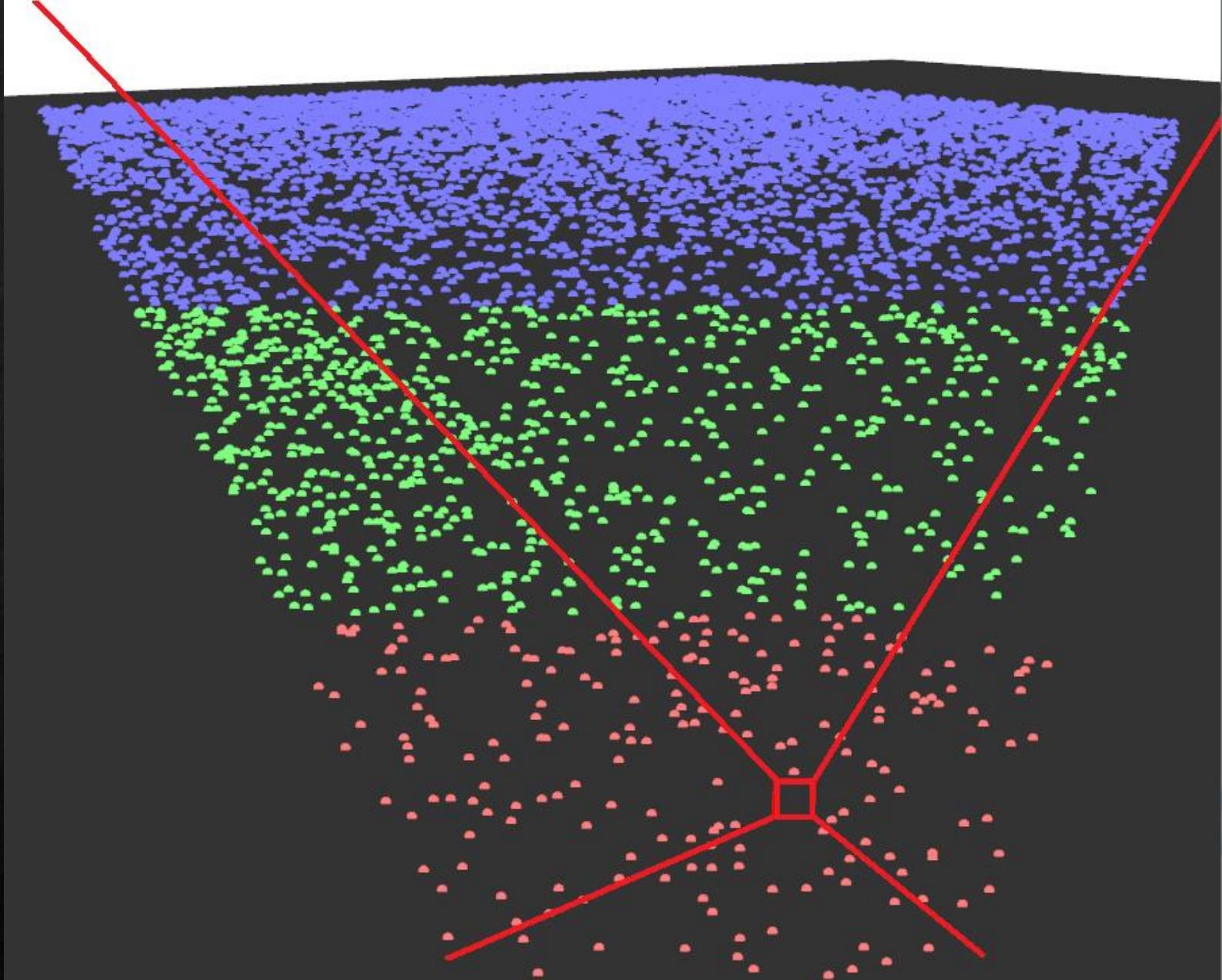
- ❖ Si la condición anterior no se cumple, entonces el **BB** es proyectado en el vector unitario  $\hat{y}$ :
- ❖ Si  $proy < -\frac{h}{2} \text{ || } proy > \frac{h}{2}$  el objeto está fuera del **frustum**.



# Radar VFC

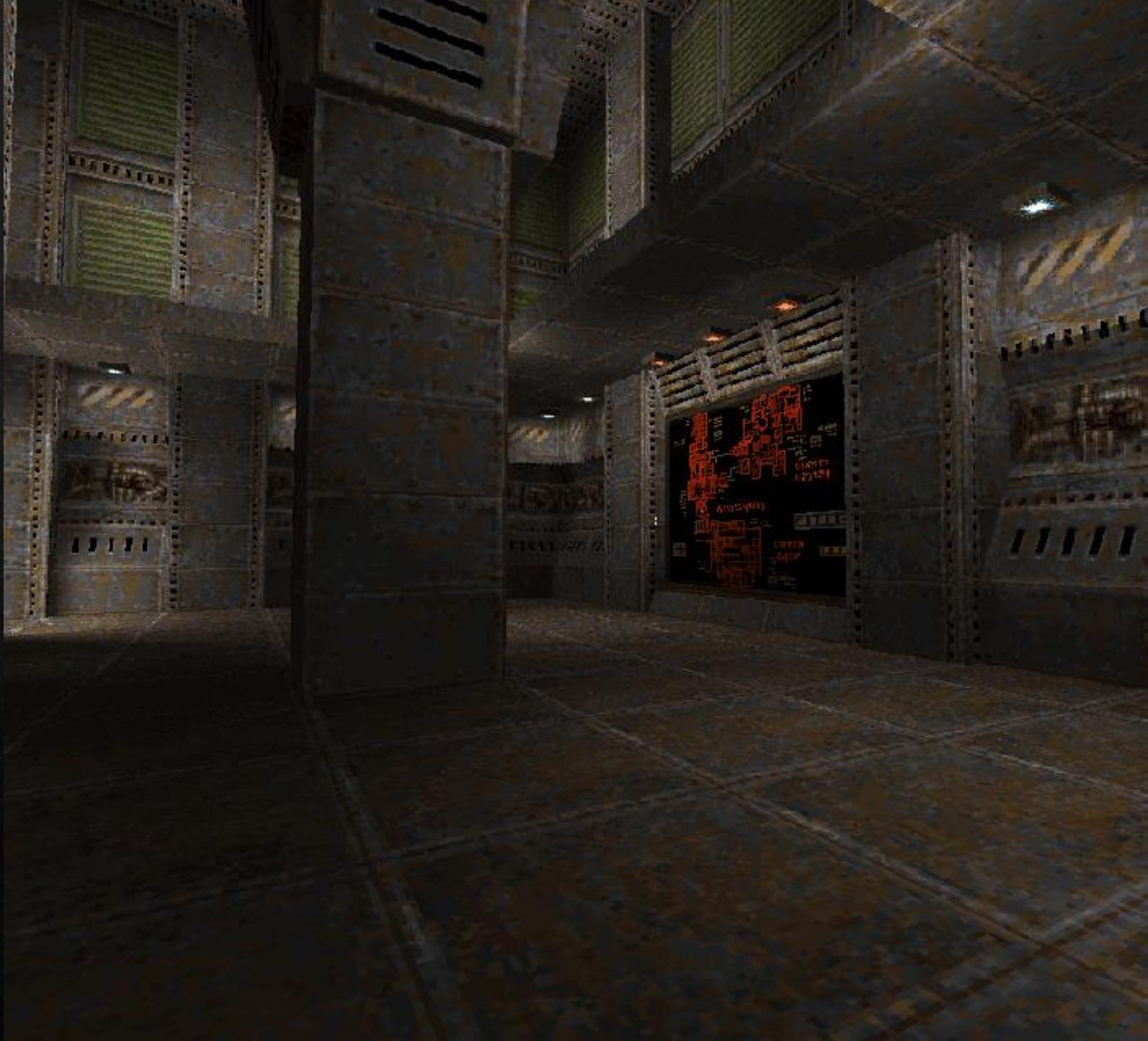
- ❖ Finalmente si la condición anterior no se cumple, el BB es proyectado en el vector  $\hat{x}$ .
- ❖ Si  $proy < -\frac{w}{2}$  ||  $proy > \frac{w}{2}$  el BB está fuera del frustum.

Radar VFC



# View Frustum Culling

Estas técnicas son ideales para escenas exteriores, ¿qué ocurre en interiores?



# Celdas y Portales



Meta: caminar a través de modelos arquitectónicos (edificios, ciudades, sótanos).



Éstos se dividen naturalmente en:

Cuartos, alcobas, corredores...



Portales transparentes conectan estas celdas:

Puertas, entradas, ventanas...



Nota: celdas sólo ven otras celdas a través de portales.

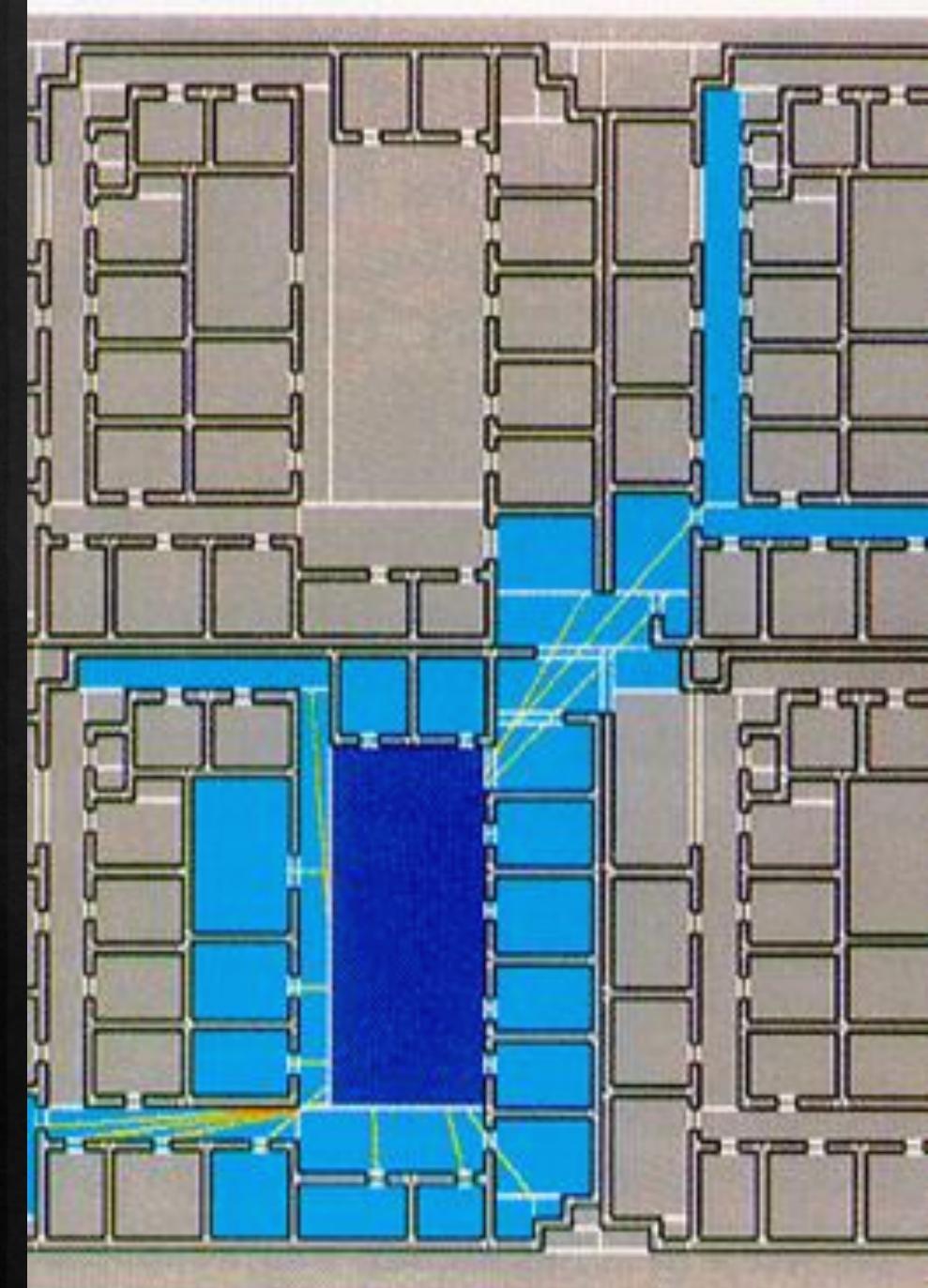
# Celdas y Portales

❖ Ejemplo:

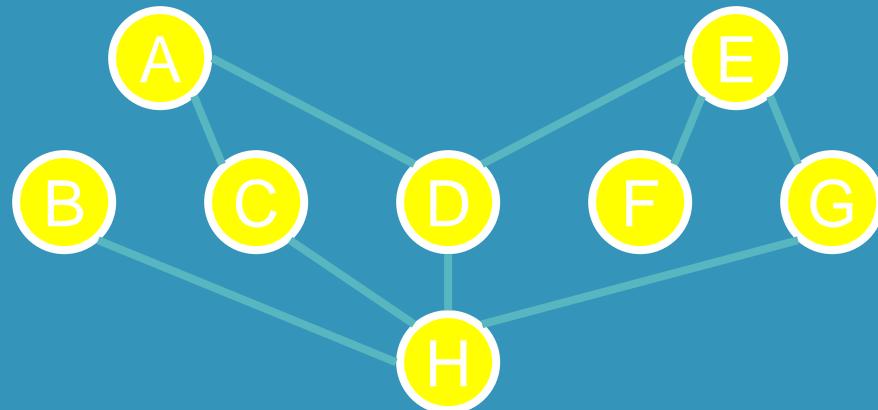
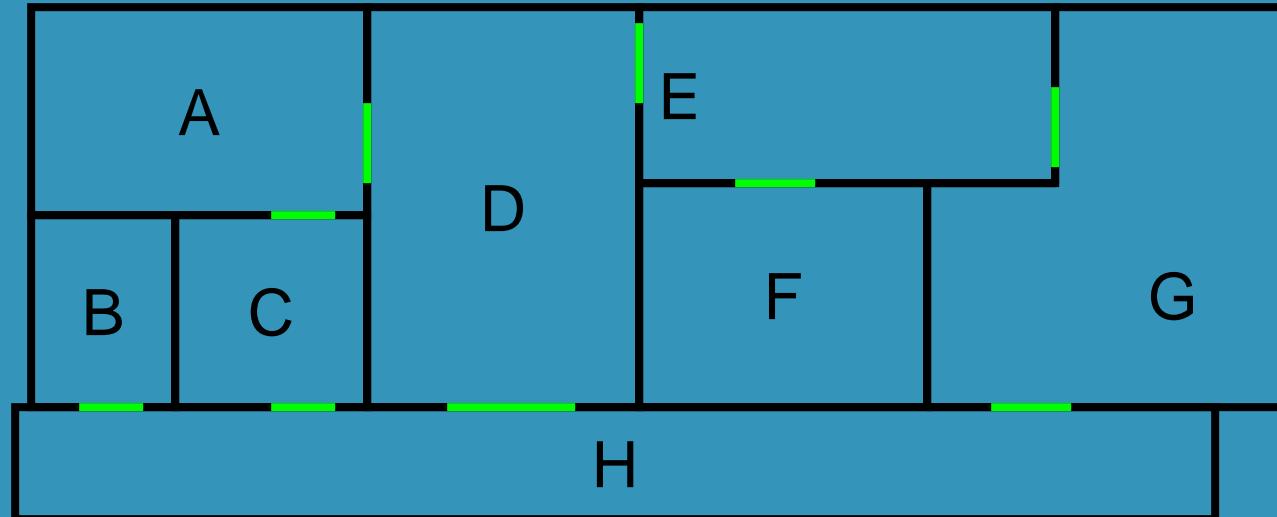


# Celdas y Portales

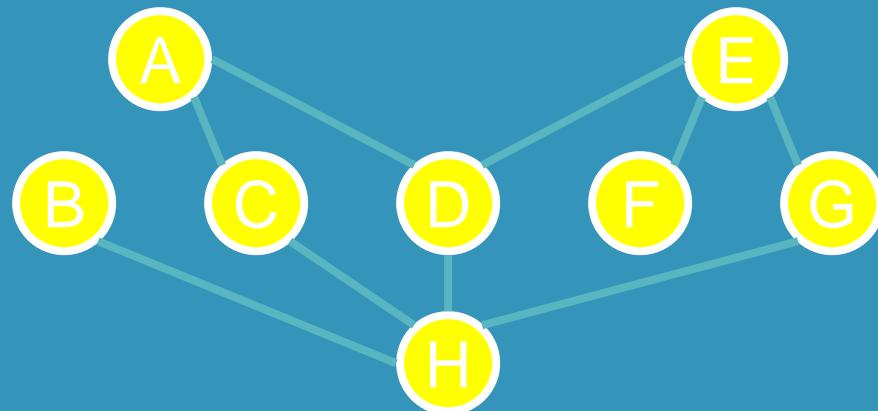
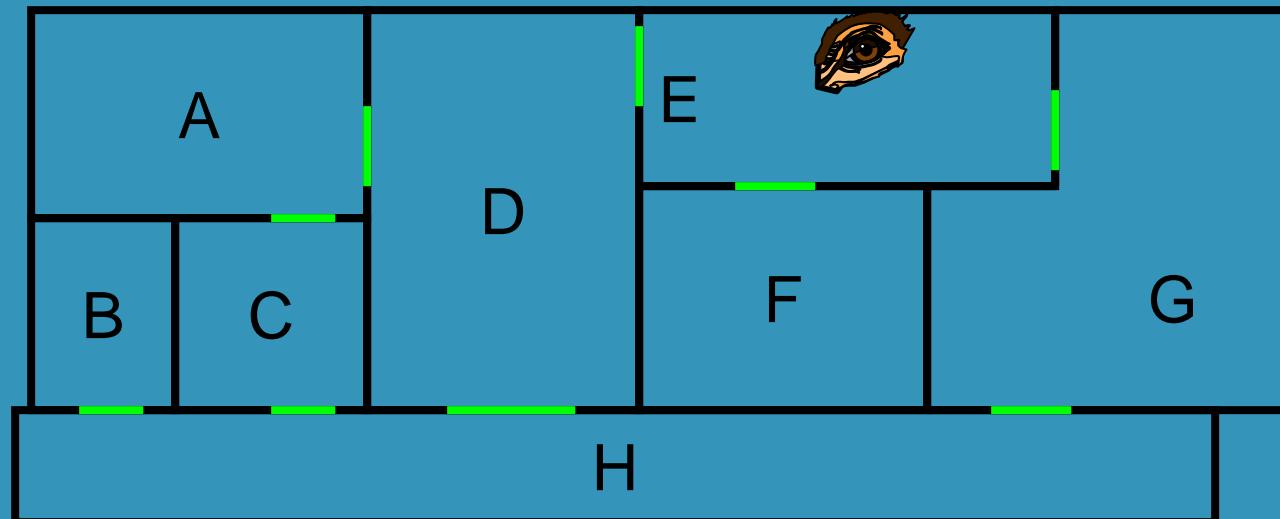
- ❖ Idea:
  - ❖ Celdas son la unidad básica del Conjunto potencial Visible (PVS).
  - ❖ Crea un **grafo de adyacencia** de las celdas.
  - ❖ Comenzando con la celda que contiene al ojo, recorre el grafo, dibujando celdas visibles.
  - ❖ Una celda es visible sólo si puede verse a través de una secuencia de portales:
    - ❖ La visibilidad de celdas se reduce a verificar que las secuencias de portales mantienen una **línea de vista**.



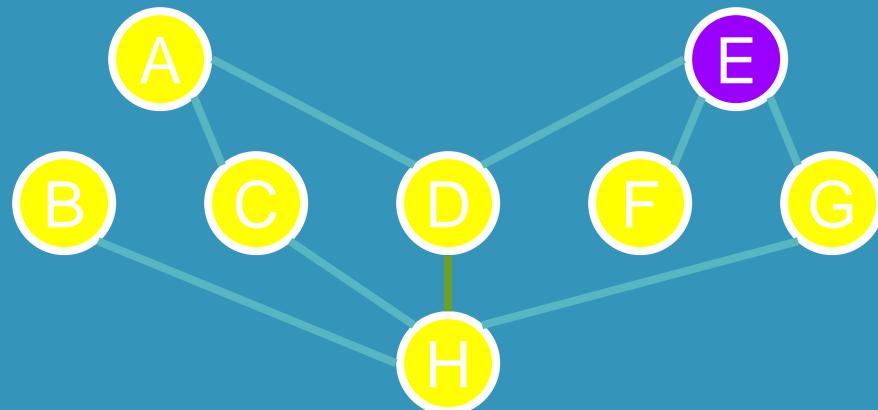
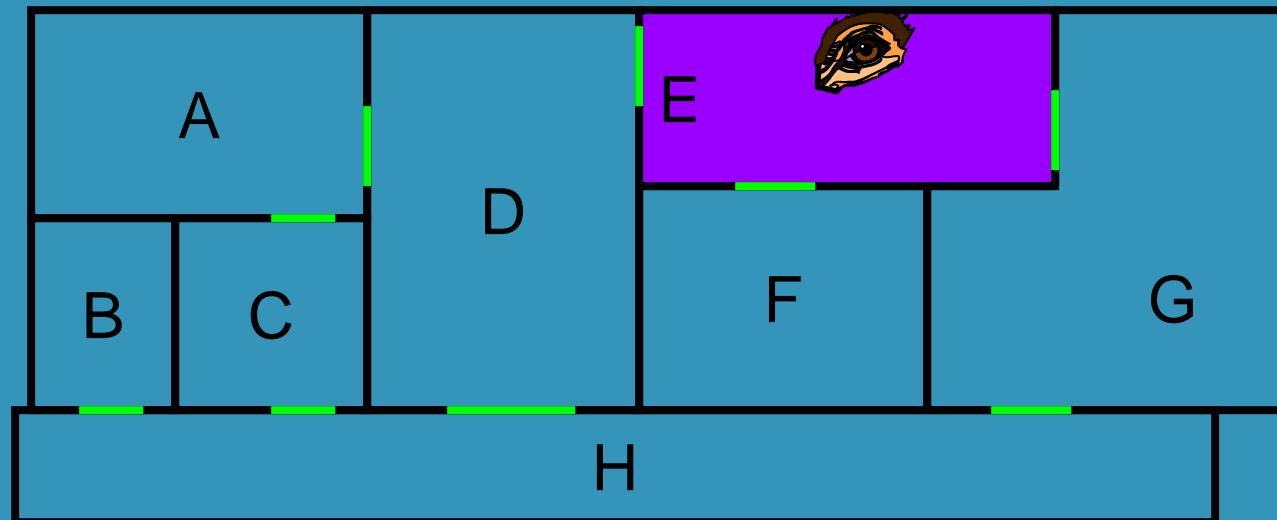
# Celdas & Portales



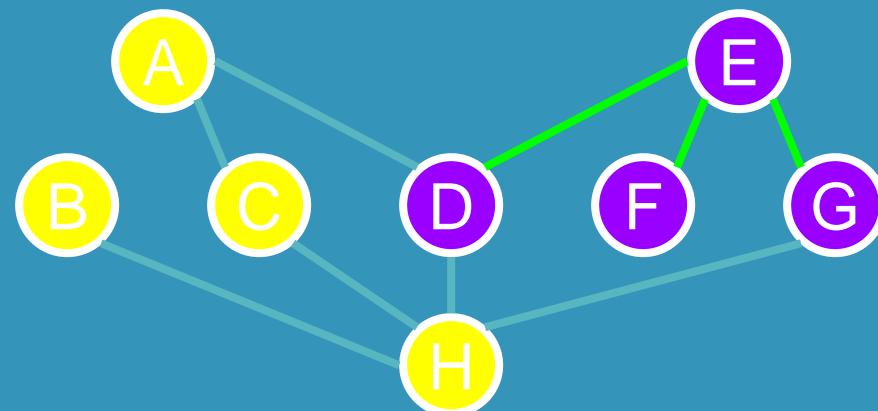
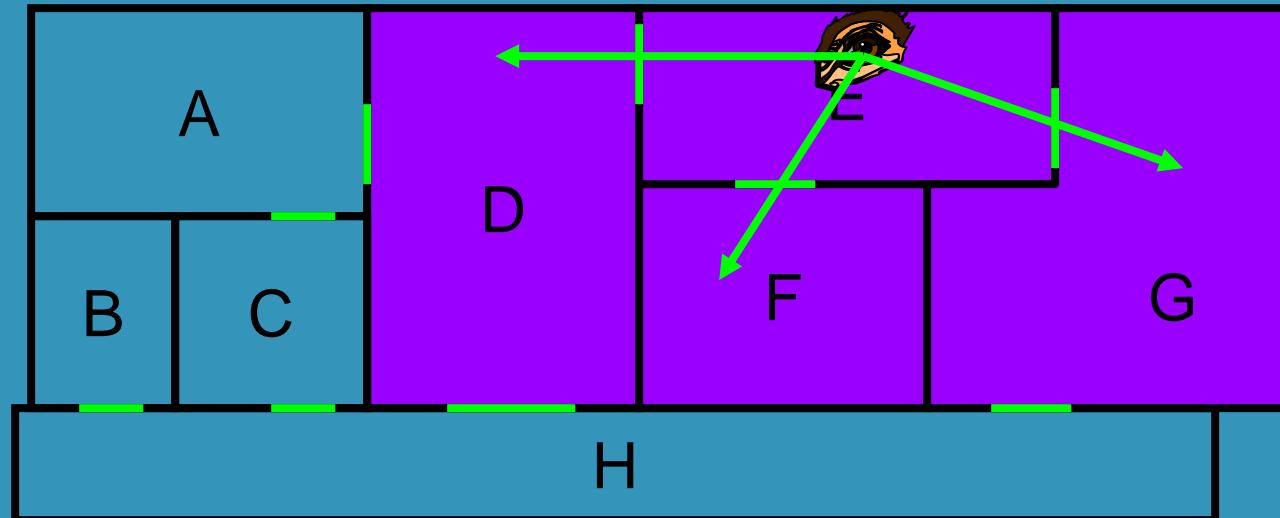
# Celdas & Portales



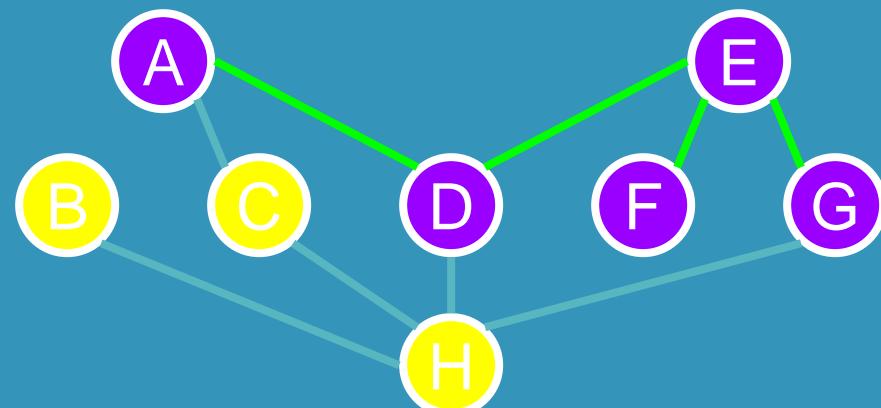
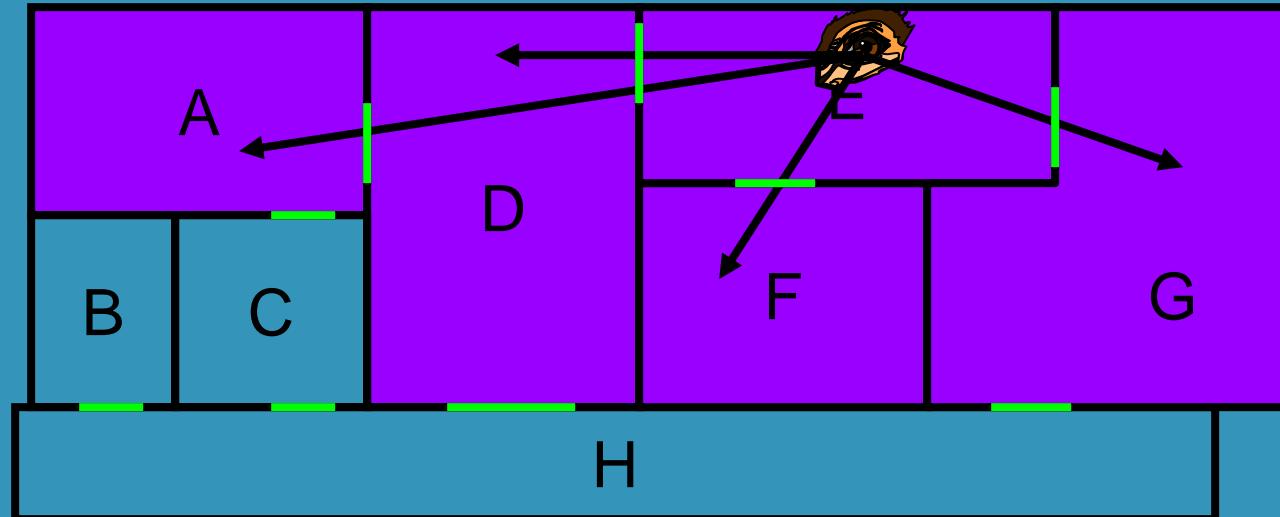
# Celdas & Portales



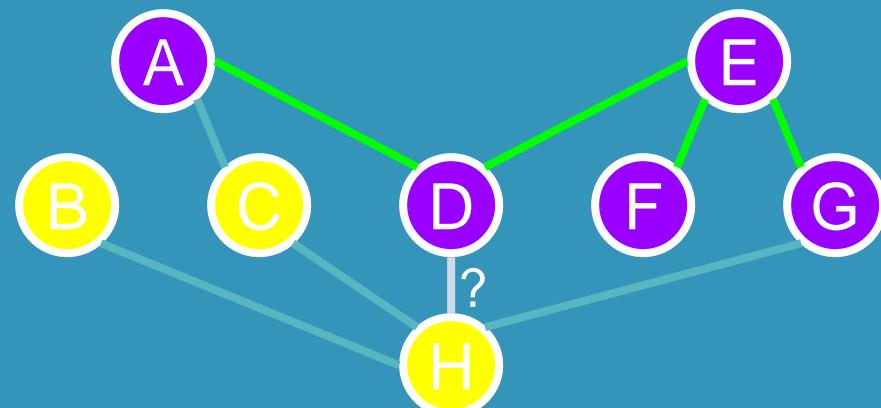
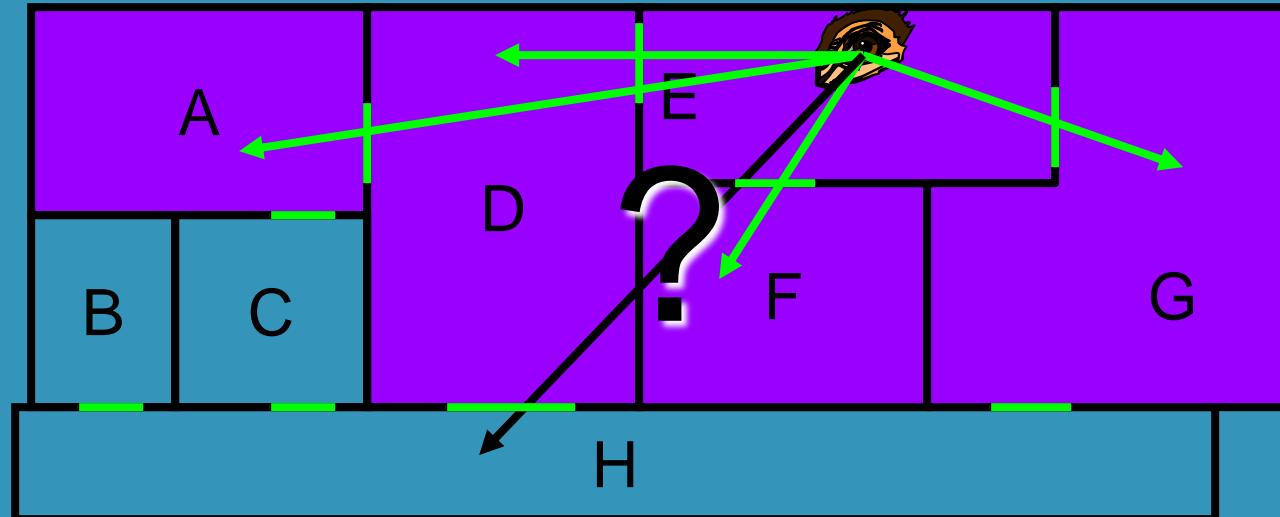
# Celdas & Portales



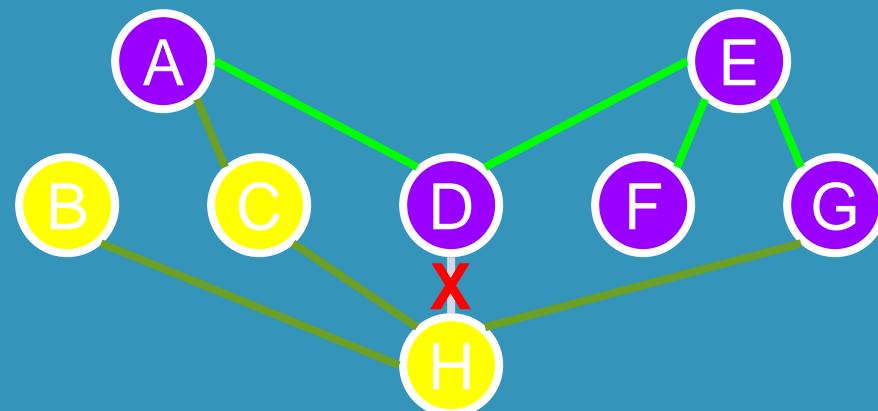
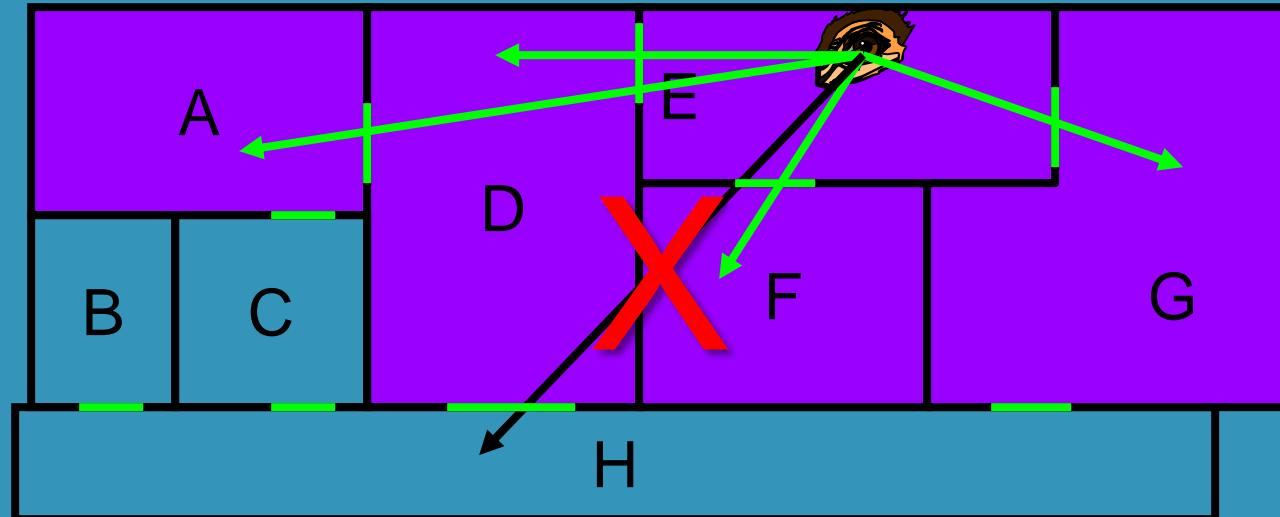
# Celdas & Portales



# Celdas & Portales

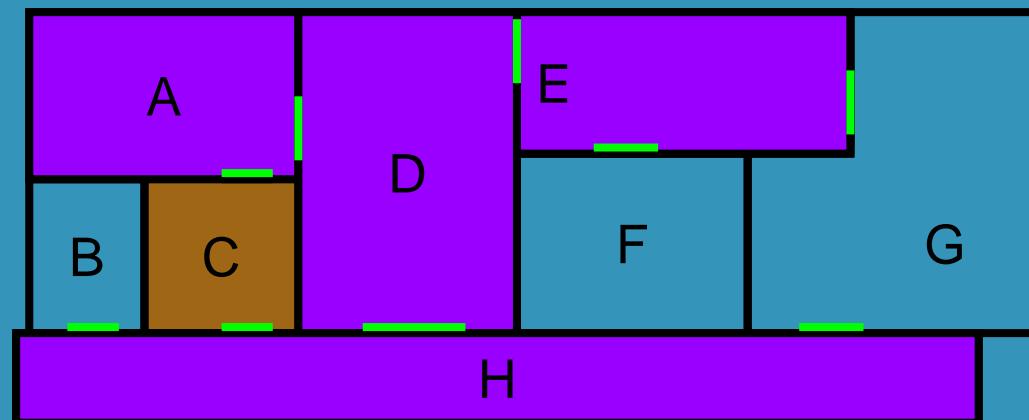


# Celdas & Portales



# Celdas & Portales

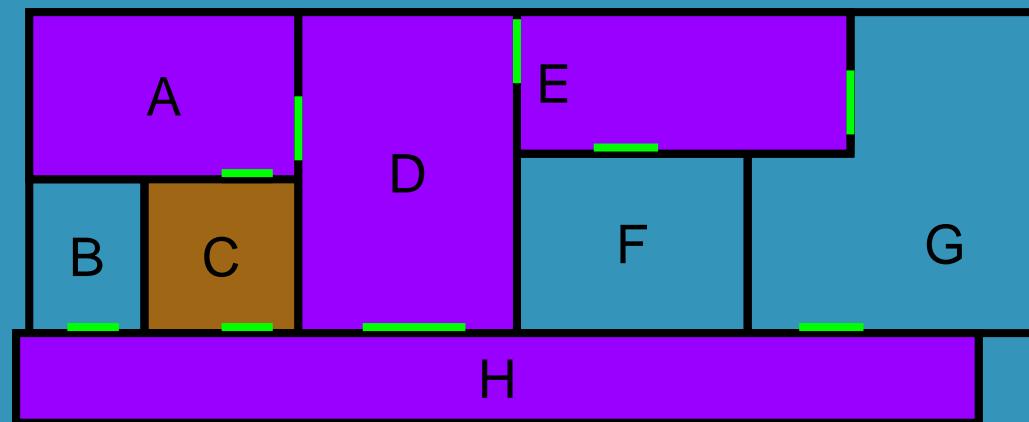
- ❖ Solución **independiente de la vista**: encuentra todas las celdas visibles desde una celda dada:



*C sólo puede ver A, D, E, y H.*

# Celdas & Portales

- ❖ Solución **independiente de la vista**: encuentra todas las celdas visibles desde una celda dada:



*C nunca verá F.*

# Celdas & Portales

- ❖ Preguntas:
  - ❖ ¿Cómo detectar si una celda es visible dado un punto de vista?
  - ❖ ¿Cómo detectar visibilidad entre celdas independientemente del punto de vista?
- ❖ Lo principal:
  - ❖ Estos problemas se reducen a visibilidad ojo-portal y portal-portal respectivamente.

## Técnicas de Nivel de Detalle

# Nivel de Detalle (LOD)

- ❖ Nivel de Detalle (**LOD**) es una importante herramienta para mantener la interactividad:
  - ❖ Se enfoca en fidelidad vs. rendimiento.
  - ❖ Se usa en conjunto con:
    - ❖ Rendering paralelo.
    - ❖ Occlusion culling.
    - ❖ Image-based rendering.

# Nivel de Detalle (LOD)

El problema:

Conjuntos de datos geométricos demasiado complejos para ser renderizados a tasas interactivas.

Una solución:

Simplificar la geometría poligonal de objetos pequeños o distantes.

Esto es LOD.

- También se llama: simplificación poligonal o geométrica, reducción de mallas, decimación, modelado multiresolución.

# Nivel de Detalle (LOD Tradicional)

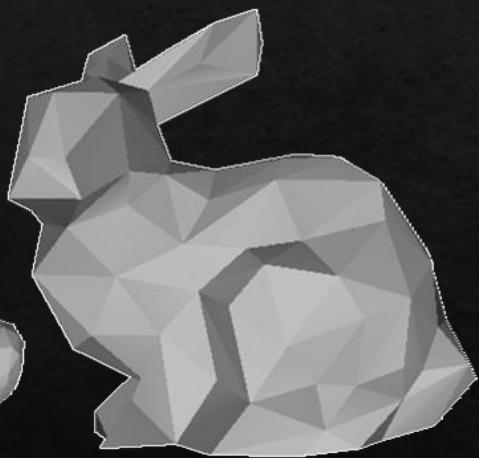
- ❖ Crea **niveles de detalle** de los objetos:



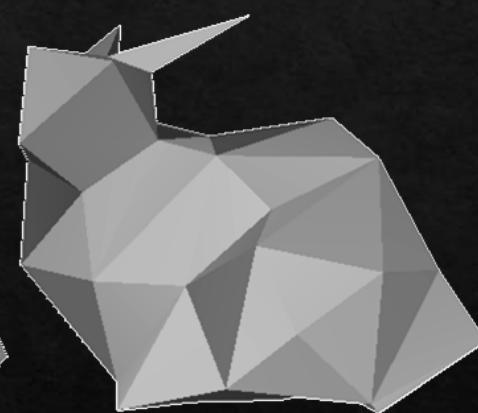
69,451 polys



2,502 polys



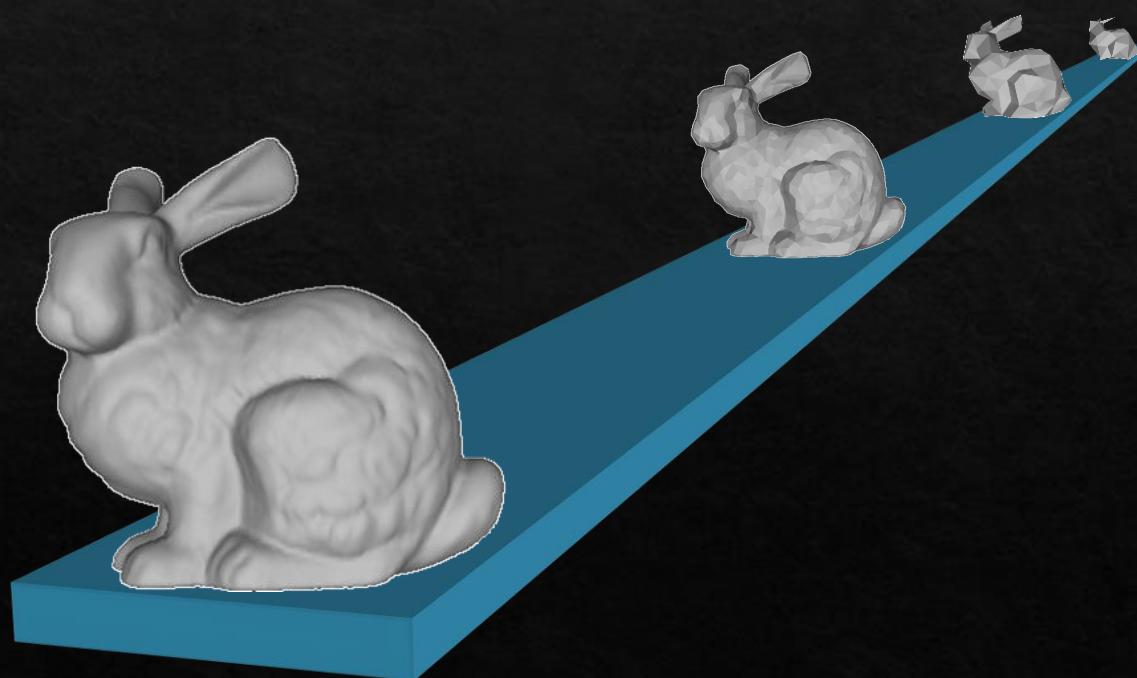
251 polys



76 polys

# Nivel de Detalle (LOD Tradicional)

Objetos distantes usan menos resolución:



# Nivel de Detalle

- ❖ ¿Cómo representar y generar versiones más sencillas de un modelo complejo?
- ❖ ¿Cómo evaluar la fidelidad de los modelos simplificados?
- ❖ ¿Cuándo usar qué LOD de un objeto?



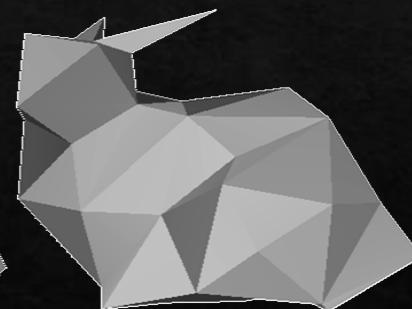
69,451 polys



2,502 polys



251 polys



76 polys

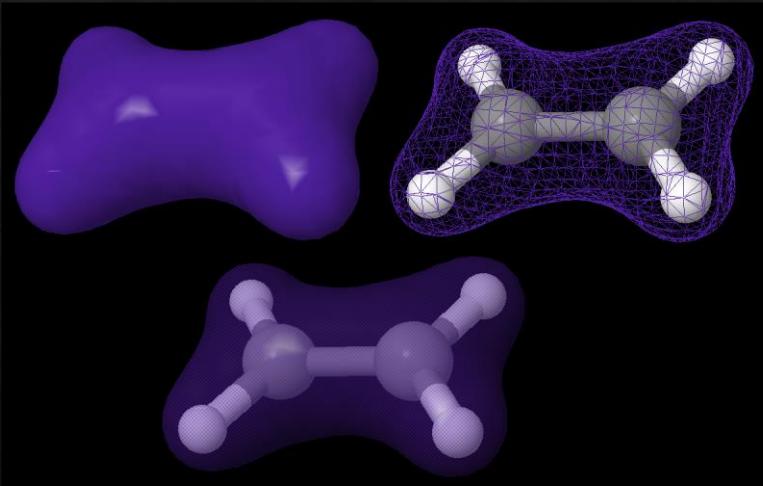
# LOD Discreto: Ventajas

- ❖ Programación sencilla, desacopla simplificación del render:
  - ❖ Creación del LOD no necesita ser en tiempo real.
  - ❖ Rendering sólo escoge el LOD durante la ejecución.
- ❖ Va bien con el hardware gráfico moderno:
  - ❖ Fácil compilar cada LOD en triangle strips, display lists, vertex arrays, ...
  - ❖ Éstos se dibujan 3 a 5 veces más rápido que triángulos desorganizados.

# LOD Discreto: Desventajas

- ¿Por qué usar otra cosa?
- Problemas con simplificación drástica.
- Casos problemáticos:
  - ⑥ Volar sobre terrenos.
  - ⑥ Isosuperficies volumétricas.
  - ⑥ Scans de datos muy detallados.
  - ⑥ Modelos CAD muy grandes.

# Casos problemáticos



Isosuperficies volumétricas:  
representan puntos de  
valor constante (presión,  
temperatura) en un  
espacio volumétrico.

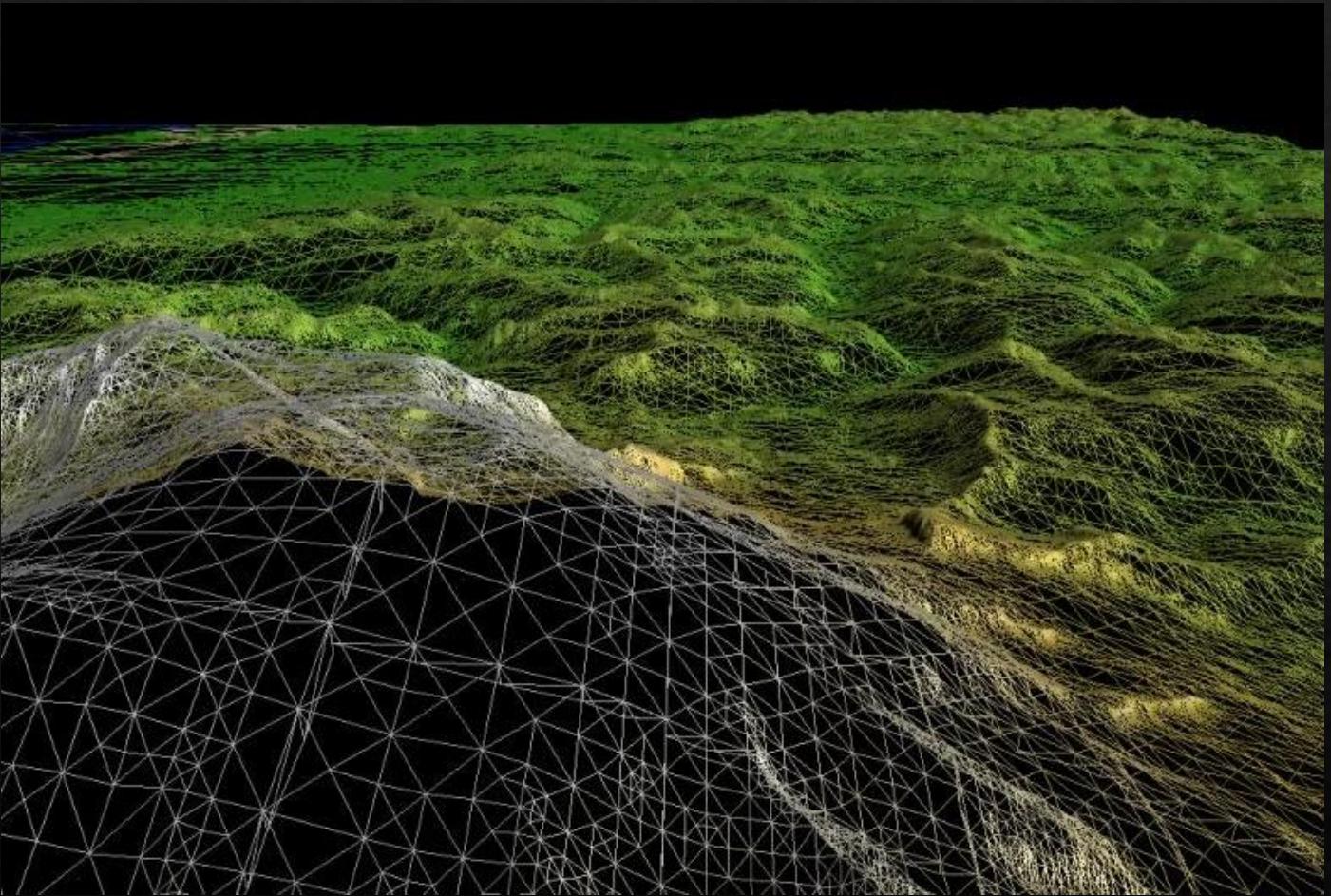


Scans de datos  
muy detallados.

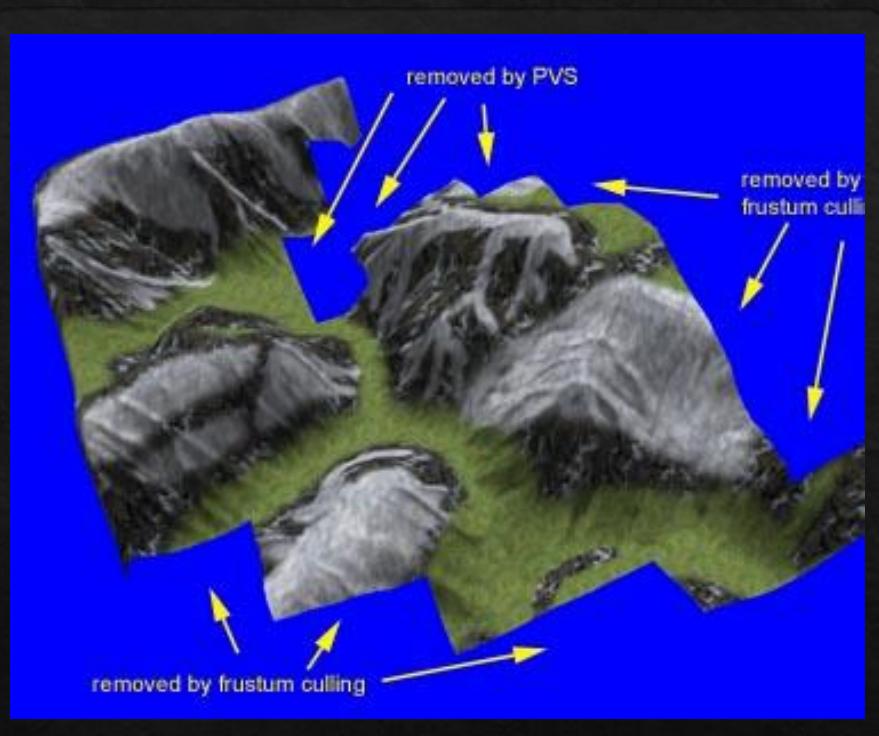
# LOD Continuo

A diferencia del enfoque discreto:

- ◊ No crea niveles de detalle individuales en un pre proceso, sino una estructura de datos de la cual un nivel de detalle deseado puede extraerse **durante la ejecución**.



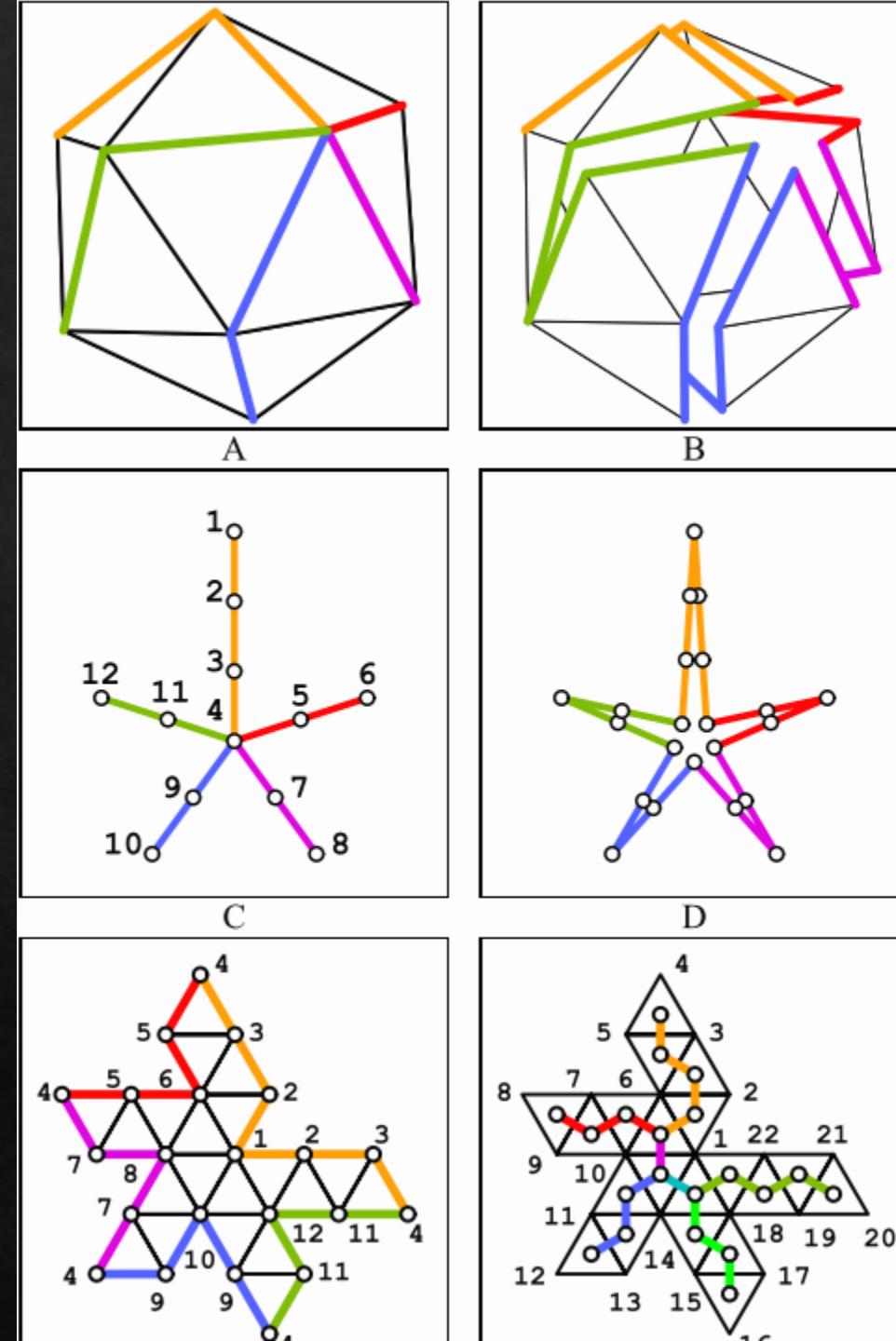
# LOD Continuo: Ventajas



- ❖ Mayor granularidad → mejor fidelidad.
  - ❖ LOD se especifica exactamente, no de unas pocas opciones creadas previamente, por lo que los objetos no usan mas polígonos de los necesarios, lo que libera polígonos para otros objetos.
- ❖ Mayor granularidad → transiciones más suaves.
  - ❖ Cambiar entre LODs tradicionales causa “popping” mientras que LOD continuo ajusta el detalle gradualmente. Además permite fácilmente hacer geomorphing para que se noten aun menos estos efectos.

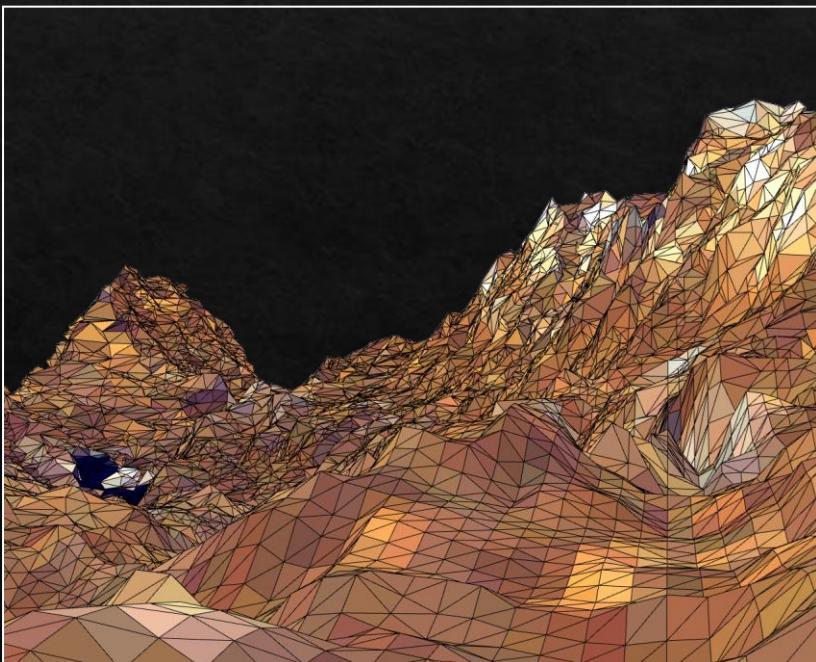
# LOD Continuo: Ventajas

- Soporta transmisión progresiva:
  - Mallas Progresivas.
  - Compresión progresiva Forest Split.
- Lleva a usar los parámetros de vista para seleccionar la mejor representación, por lo que el mismo objeto puede estar representado a diversos niveles de detalle.

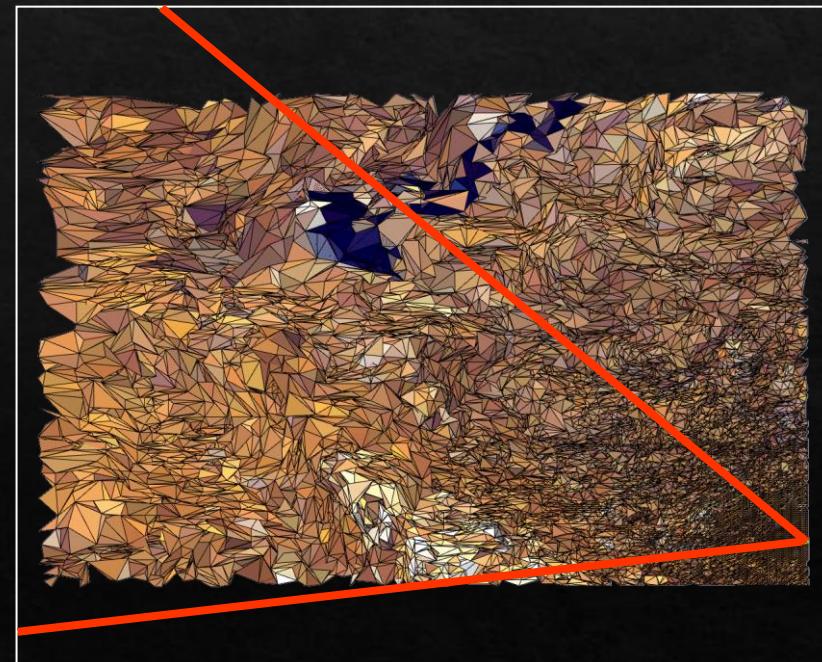


# LOD Dependiente de la Vista

- ❖ Muestra partes cercanas de un objeto a mayor resolución que las lejanas:



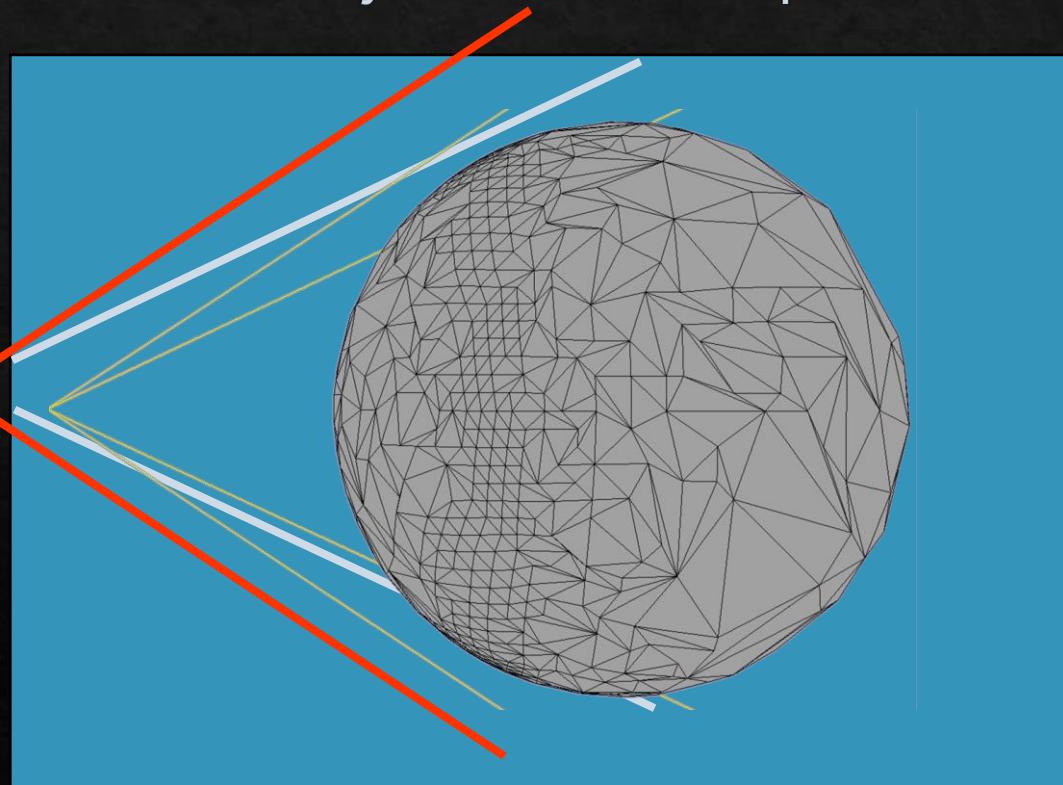
Desde el ojo



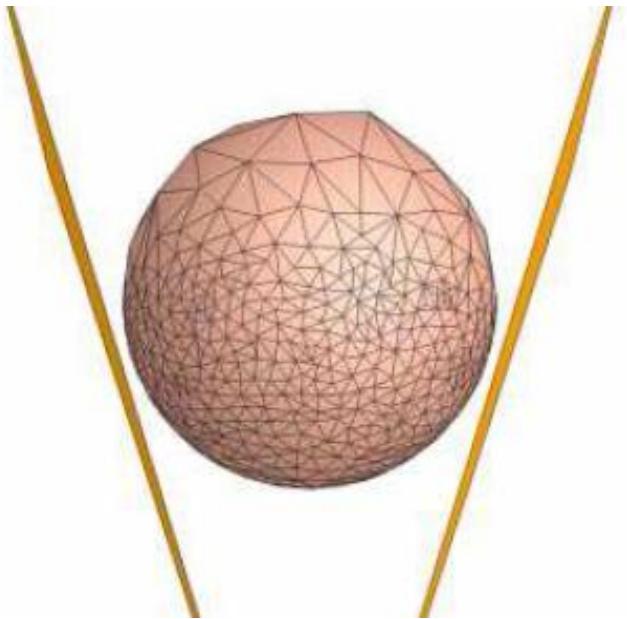
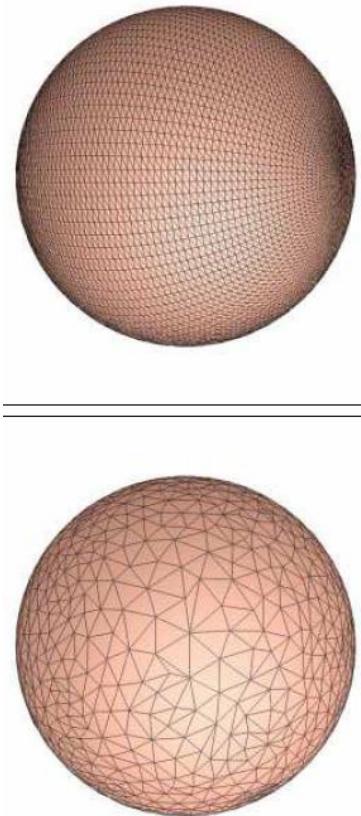
Desde arriba

# LOD Dependiente de la Vista

- ❖ Muestra silueta con mayor resolución que el interior:



LOD  
Dependiente  
de la Vista



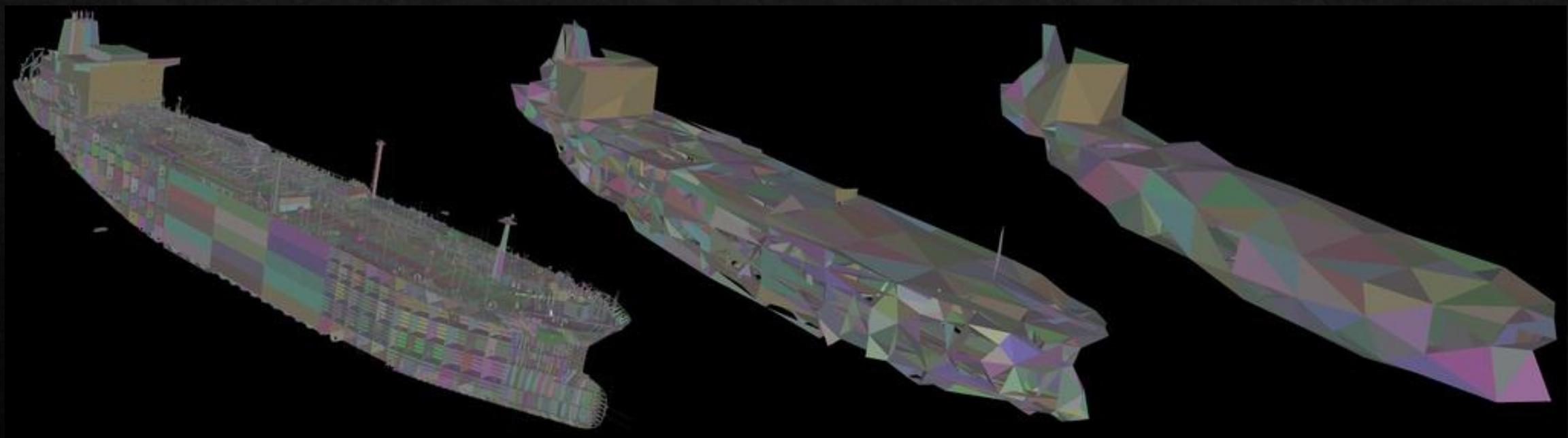
# LOD dependiente de la vista: Ventajas

- ❖ Aún mayor granularidad.
- ❖ Permite simplificar drásticamente objetos grandes:
  - ❖ Modelo de estadio.
  - ❖ Volar sobre terreno.



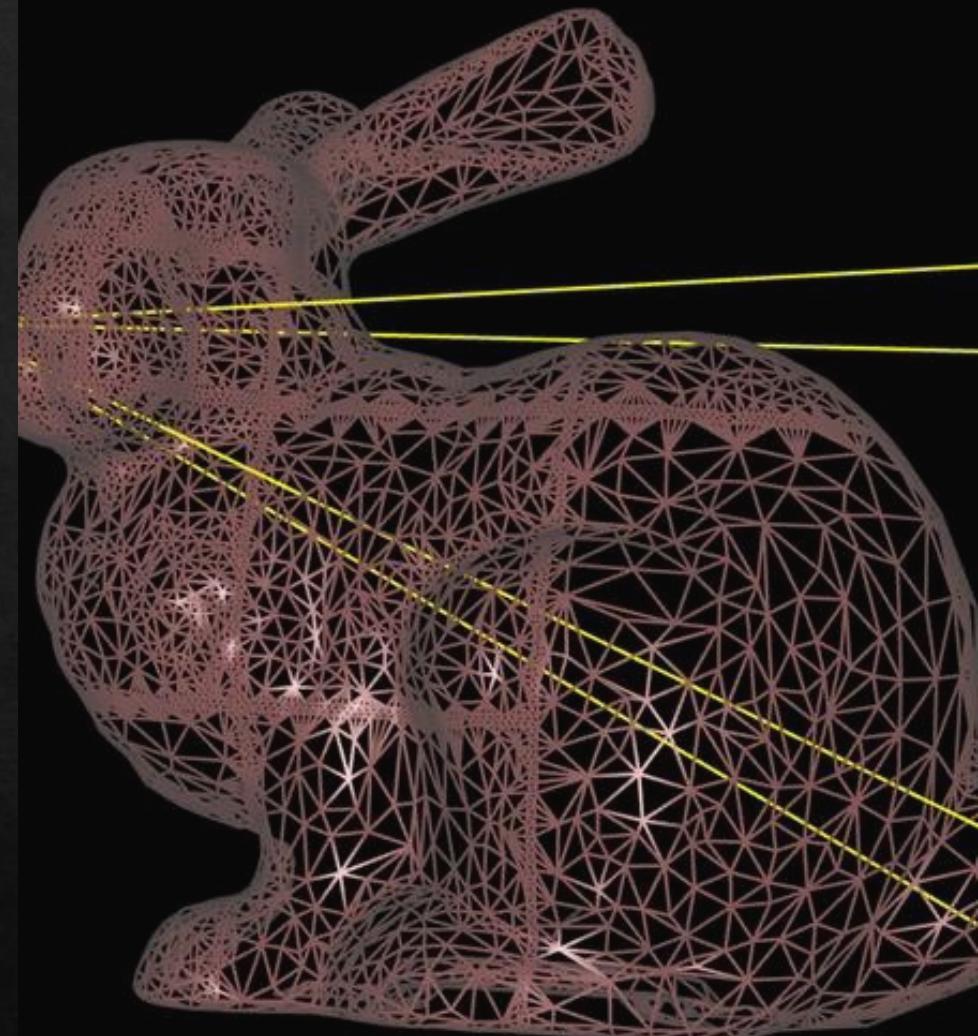
# LOD Jerárquico

- ❖ Mientras que LOD Dependiente de Vista resuelve el problema de objetos grandes, LOD Jerárquico puede resolver el problema de objetos pequeños:
  - ❖ Junta objetos en colecciones de objetos.
  - ❖ A distancia suficiente simplifica las colecciones, no los objetos individuales.



# LOD Jerárquico

- ❖ LOD Jerárquico funciona bien junto con LOD Dependiente de Vista:
  - ❖ Maneja toda la escena como un solo objeto a ser simplificado de manera dependiente de vista
- ❖ LOD Jerárquico también funciona con los esquemas de LOD Discreto tradicionales:
  - ❖ *Impostores*



# Impostor

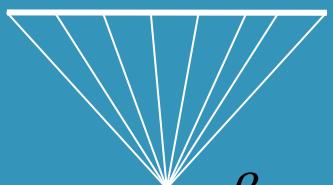
## ❖ Definición:

- ❖ Imagen de un objeto complejo que es mapeado como textura a un rectángulo (puede ser otro polígono).
- ❖ Usualmente imagen de textura y valor alfa.
- ❖ La resolución de textura es importante, da vida al Impostor.

$$texres = screenres \frac{objsize}{distance}$$

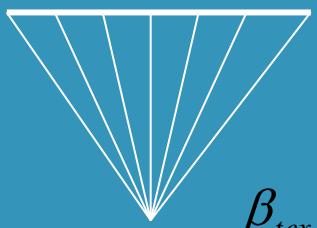
$$\beta_{tex} > \beta_{scr}$$

Screen resolution

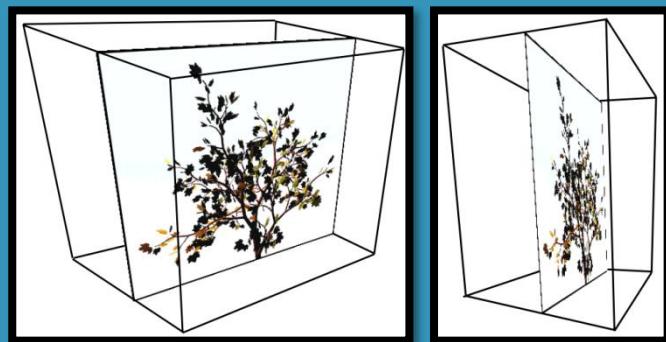


$$\beta_{scr}$$

Texture resolution



$$\beta_{tex}$$





Impostores



