

# Estructuras de Datos Espaciales y Técnicas de Aceleración

Tomas Akenine-Möller  
Department of Computer Engineering  
Chalmers University of Technology

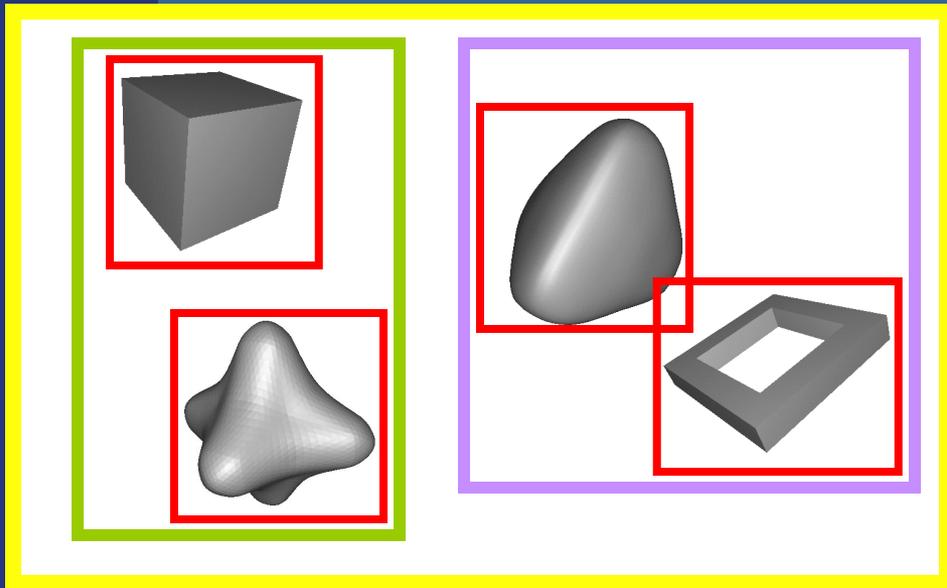
# Estructuras de datos espaciales

- ¿Qué son?
  - Son estructuras de datos que organizan la geometría en 2D, 3D o más dimensiones.
  - El objetivo es lograr un procesamiento más rápido.
  - Necesarias en la mayoría de las "técnicas de aceleración"
    - Acelerar rendering en tiempo real
    - Acelerar los test de intersección
    - Acelerar la detección de colisiones
    - Acelerar ray tracing e iluminación global
- En los juegos son ampliamente utilizadas.
- Las herramientas para la generación de animaciones también las utilizan.

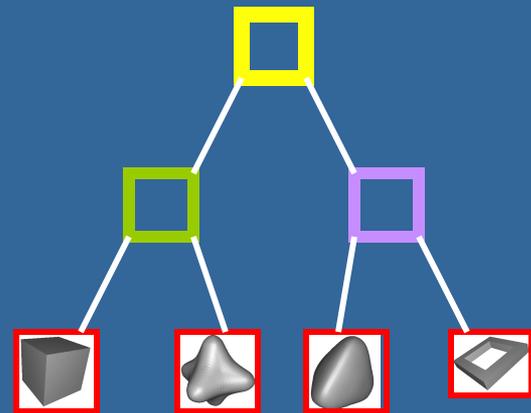
# ¿Cómo funcionan?

- Organizan la geometría jerárquicamente

En espacio 2D

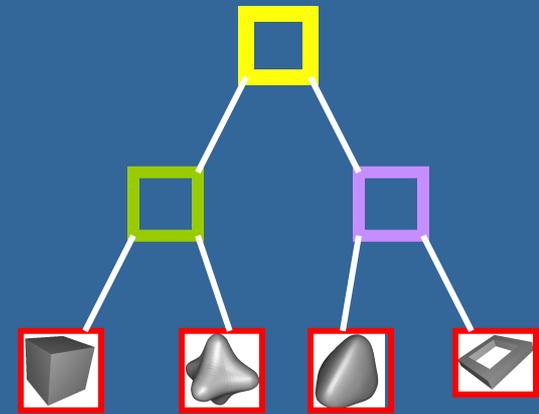
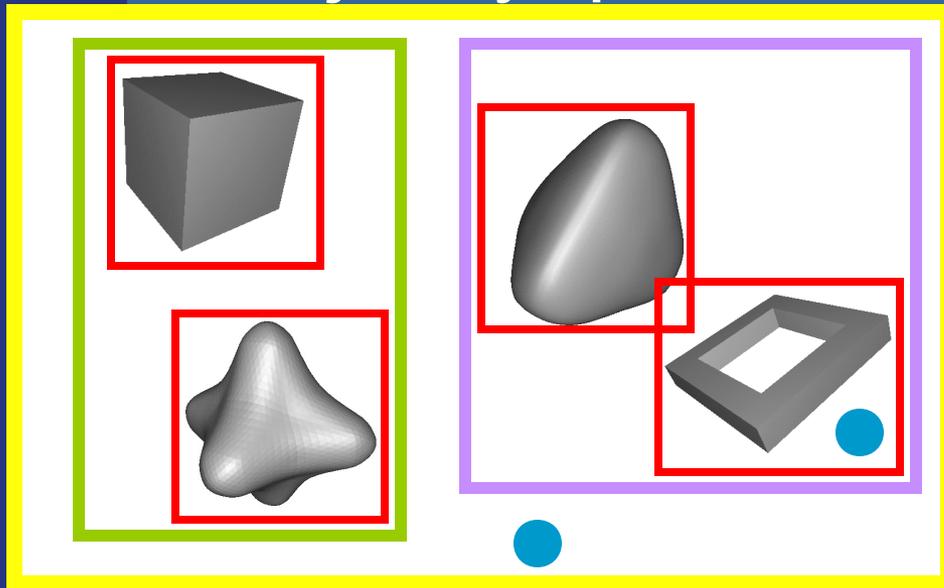


Estructura de datos



# Para qué sirve? Un ejemplo

- Asumamos que hacemos clic sobre un objeto y queremos saber sobre cual fue

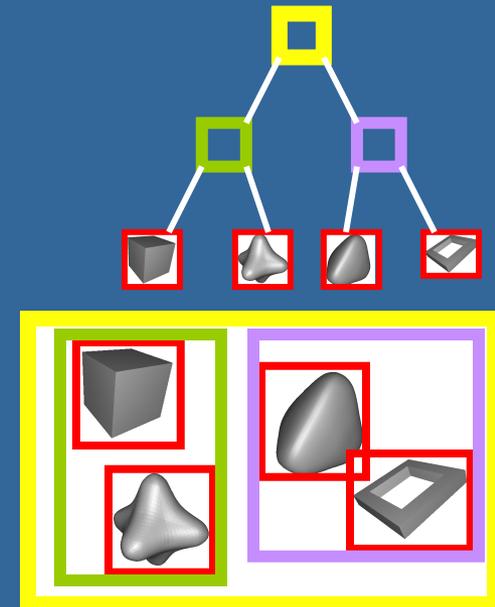
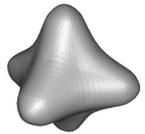


●  
clic!

- 1) Testear primero la raíz
  - 2) Descender recursivamente
  - 3) Terminar el recorrido cuando sea posible
- En general  $O(\log n)$  en lugar de  $O(n)$

# Jerarquía de Volúmenes Acotantes Bounding Volume Hierarchy (BVH)

- Volúmenes acotantes más comunes (BVs):
  - Esfera
  - Caja (AABB and OBB)
- Los BV no son parte de la imagen generada -- más bien, encierra a un objeto
- La estructura es un árbol k-ario
  - Las hojas tienen la geometría
  - Los nodos internos tienen a lo más k hijos
  - Los nodos internos tienen BVs que encierran toda la geometría de su subarbol



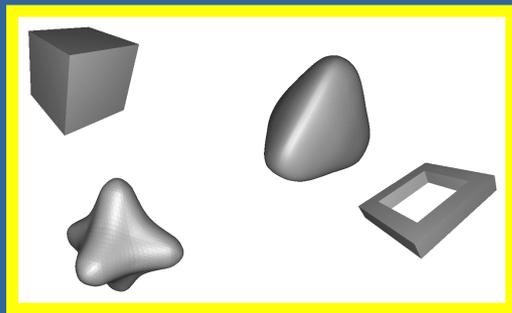
# Algunos hechos acerca de los árboles

- *La altura del árbol*,  $h$ , es el camino más largo de la raíz a las hojas
- *Un árbol balanceado* tiene todas las hojas a altura  $h$  o  $h+1$
- La altura de un árbol balanceado con  $n$  nodos:  $\text{floor}(\log_k(n))$
- Un árbol binario ( $k=2$ ) es el más simple
  - $k=4$  y  $k=8$  es bastante común en computación gráfica

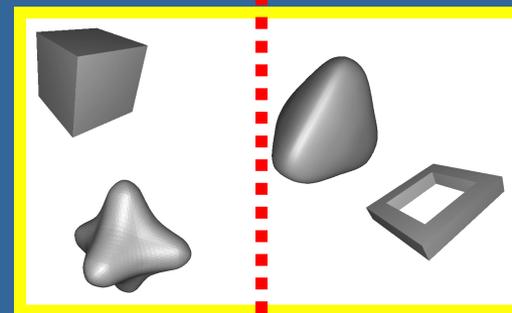
# Cómo crear una BVH?

## Ejemplo: BV=AABB

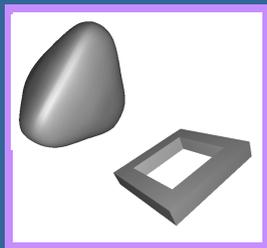
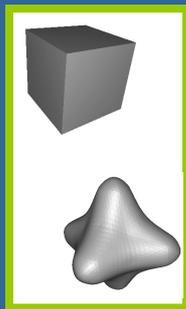
- Hallar caja mínima y separar por el eje más largo



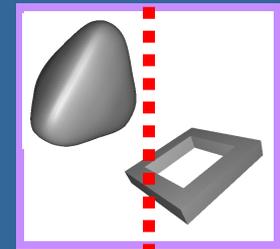
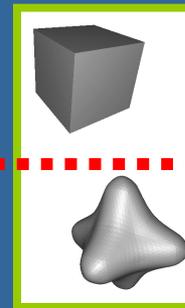
x es más largo



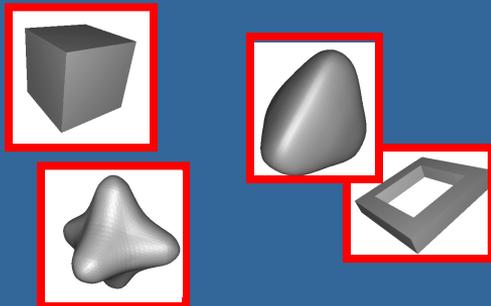
Hallar cajas mínimas



Separar a lo largo de los ejes más largos



Encontrar cajas mínimas



Es un método TOP-DOWN  
Es más complejo para otros BVs

# Criterio de parada para el método Top-Down

- Es necesario detener la recursión...
  - Cuando BV está vacío
  - O cuando sólo una primitiva (e.g. triángulo) está dentro del BV
  - O cuando  $<n$  primitivas están dentro del BV
  - O cuando se alcanzó el nivel de recursión  $l$
- Es un criterio similar para árboles BSP y para octrees

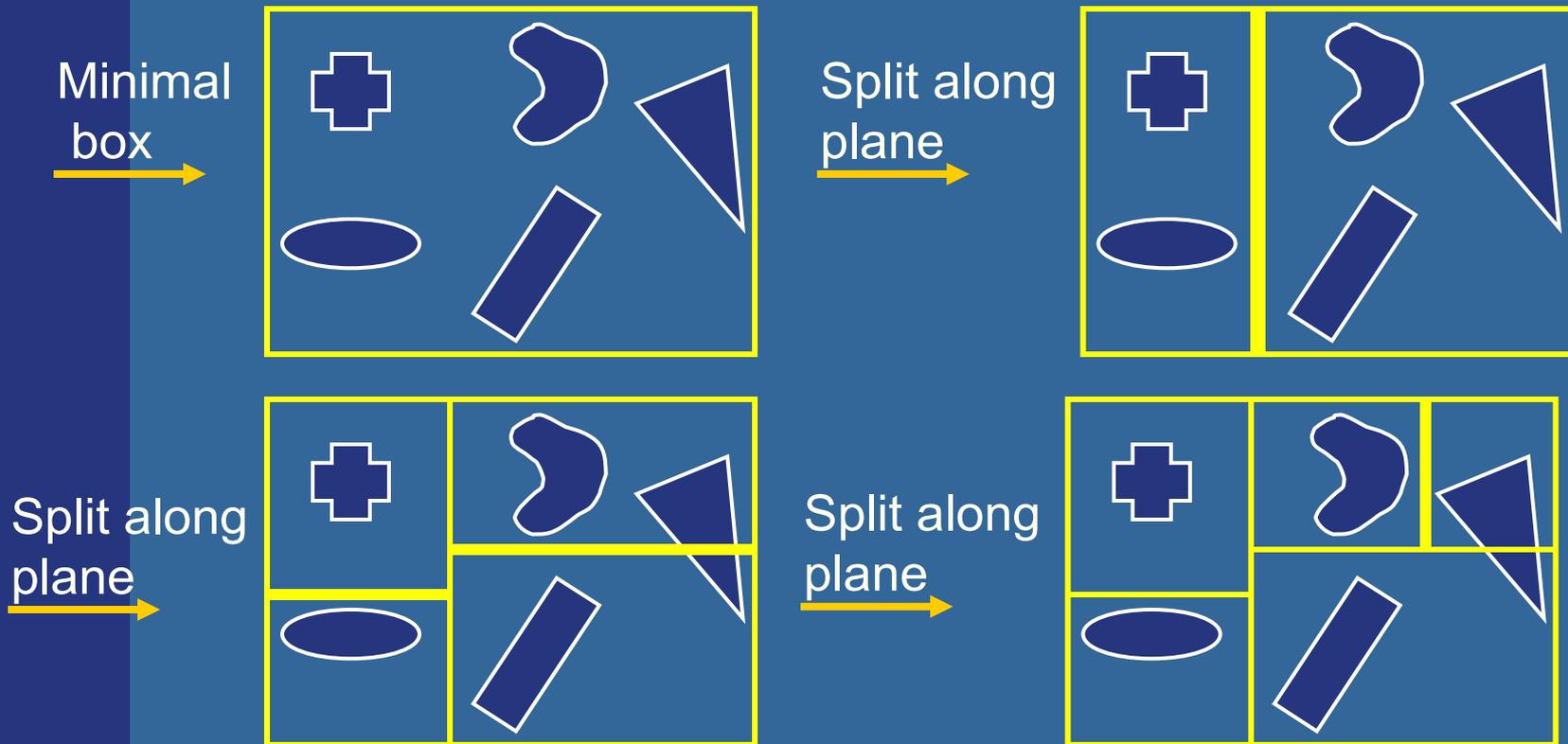
# Binary Space Partitioning (BSP) Trees

## Árboles BSP

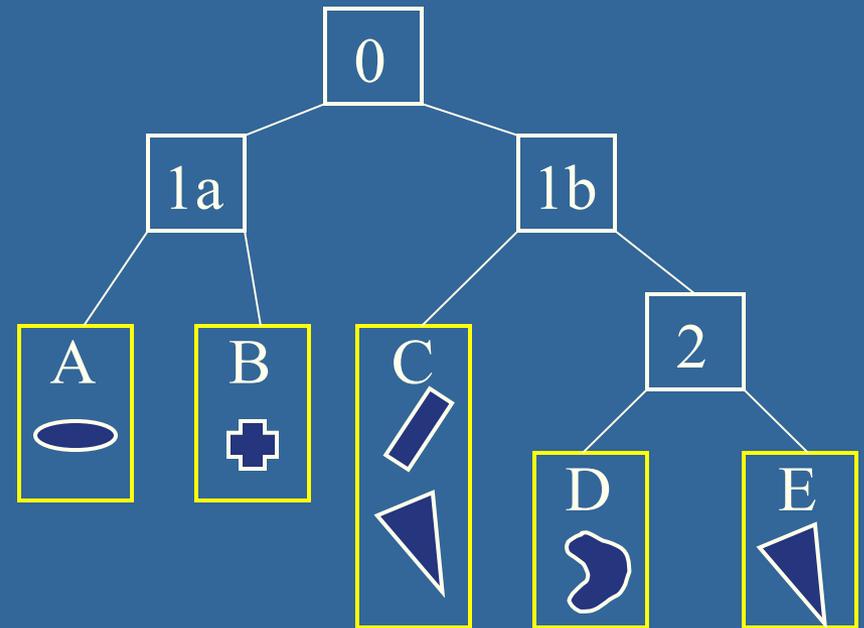
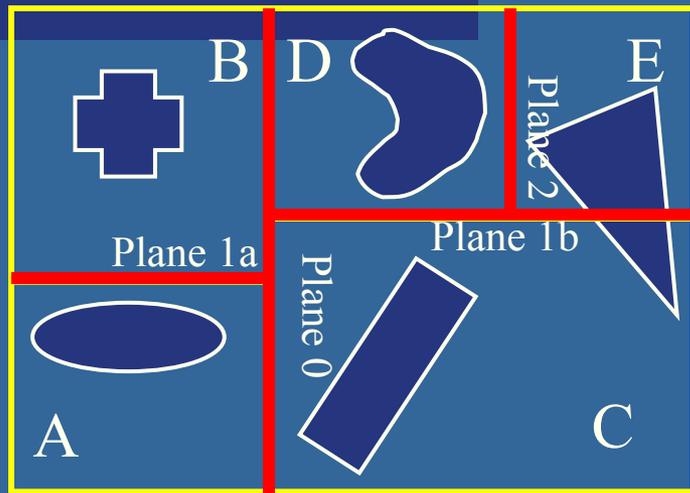
- Dos tipos diferentes:
  - Alineado a los ejes
  - Alineado a los polígonos
- Idea general:
  - Dividir el espacio con un plano
  - Ordenar geoméricamente dentro del espacio al que pertenece
  - Hacerlo recursivamente
- If traversed in a certain way, we can get the geometry sorted along an axis
  - Exacto cuando es alineado a los polígonos
  - Aproximado cuando es alineado a los ejes

# Árbol BSP alineado a los ejes (1)

- Sólo pueden haber planos separadores alineados a x,y o z



# Árbol BSP alineado a los ejes (2)

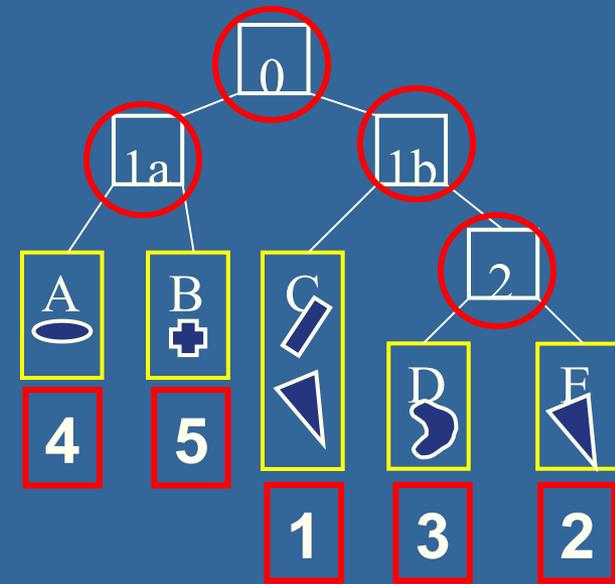
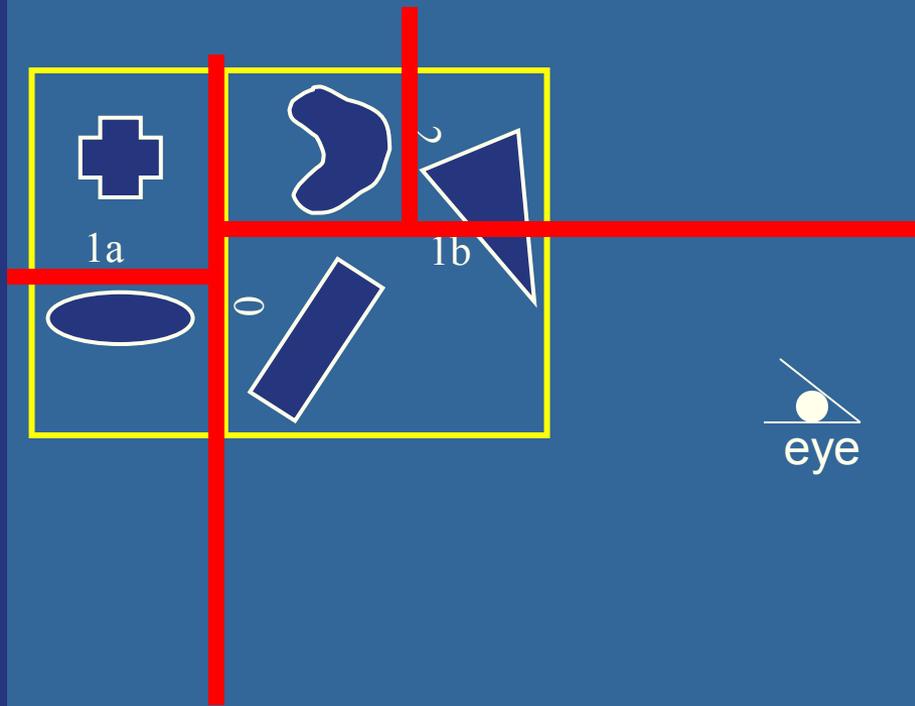


- Cada nodo interno contiene un plano separador
- Las hojas contienen la geometría
- Diferencias con BVH
  - Encierra todo el espacio y provee de sorting
  - La BVH puede ser construida de cualquier forma (no sort)
  - BVHs pueden utilizar cualquier tipo deseable de BV

# Árbol BSP alineado a los ejes

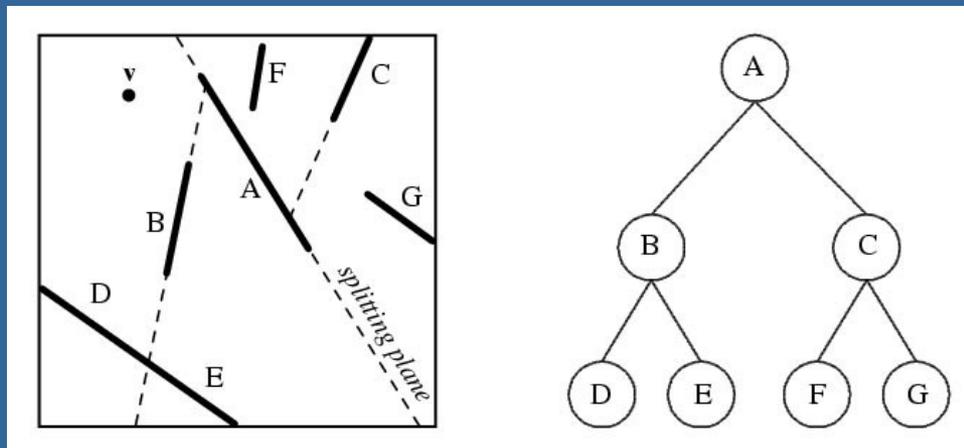
## Algoritmo de sort en bruto

- Testear los planos contra el punto de vista
- Testear recursivamente desde la raíz
- Ordenar de adelante hacia atrás. ck



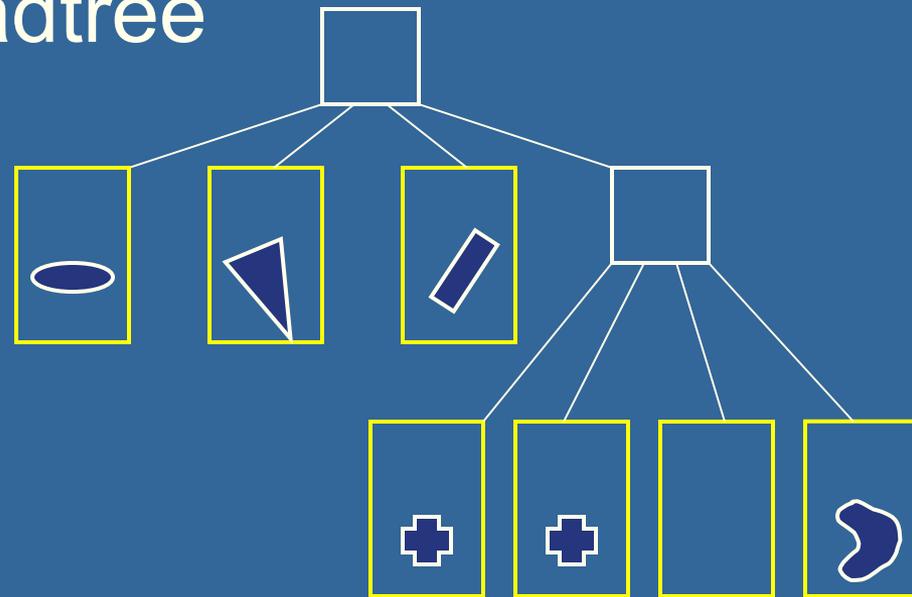
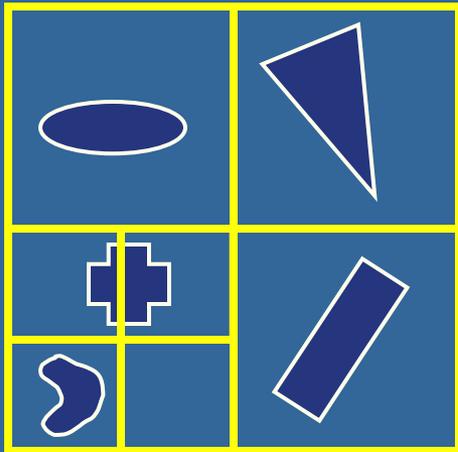
# Árbol BSP alineado a los polígonos

- Se lo menciona en otra parte del curso
- Permite un ordenamiento exacto
- Muy similar al BSP alineado a los ejes
  - El plano de corte está ahora localizado en los planos de los triángulos



# Octrees (1)

- Similar a los árboles BSP alineados a los ejes
- Se explica con quadtree



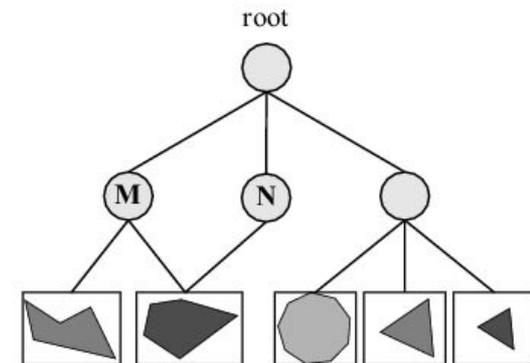
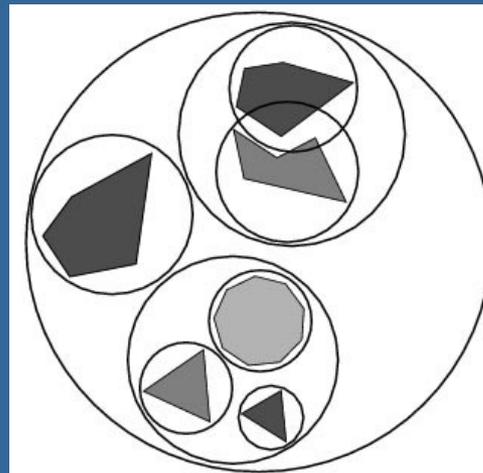
- En 3D es una caja, con 8 hijos

## Octrees (2)

- Caro de reconstruir (ídem BSPs)
- Leer acerca de loose octrees en el libro
  - Es una relajación del octree para evitar problemas
- Octrees pueden ser usados para
  - Acelerar ray tracing
  - Acelerar el picking
  - Para las técnicas de Culling
  - No son muy usados en tiempo real
    - Una excepción son los loose octrees

# Grafos de escena

- BVH es la estructura más comunmente usada
  - Simple de comprender
  - Código simple
- Sin embargo, sólo almacena geometría
  - Rendering es más que geometría
- El grafo de escena es un BVH extendido con:
  - Luces
  - Texturas
  - Transformaciones
  - Y más



# Técnicas de aceleración

- Las estructuras de datos espaciales son usadas para acelerar el rendering y las diferentes consultas
- Por qué mayor velocidad?
- ¡Hardware gráfico 2x más rápido en 6 meses!
- Alcanzaría con esperar
- ¡NO!
- Nunca estaremos satisfechos
  - Resolución de pantalla: 3000x1500
  - Realismo: iluminación global
  - Complejidad geométrica: ¡no hay límite superior!

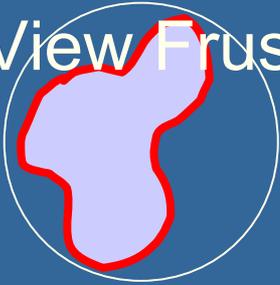
# Qué se va a tratar ahora

- Técnicas de culling
- Rendering con nivel de detalle (Level-of-detail rendering (LODs))
- “Cull” significa “seleccionar de un grupo”
- En contexto gráfico: no procesar datos que no contribuyen a la imagen final

# Varias técnicas de culling

(Los objetos rojos son salteados)

Sección de vista  
(View Frustum)



■ detalle

Cara de atrás  
(backface)



portal



oclusión

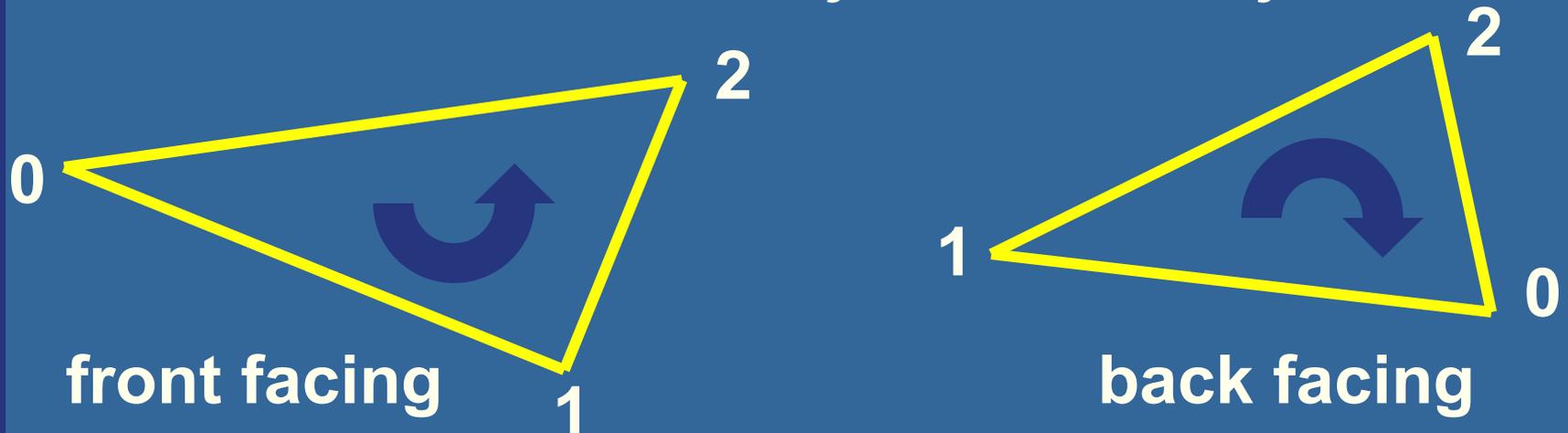


# Backface Culling

- Técnica simple para eliminar polígonos que no miran hacia el espectador
- Son usados en:
  - Superficies cerradas (esfera, cubo, toro)
  - or cuando sabemos que las backfaces no van a ser vistas (ejemplo: paredes en una habitación)
- 2 métodos (espacio de pantalla y de vista)
- Qué etapas de beneficiar? Rasterización, pero también geometría  
(donde los test son hechos)

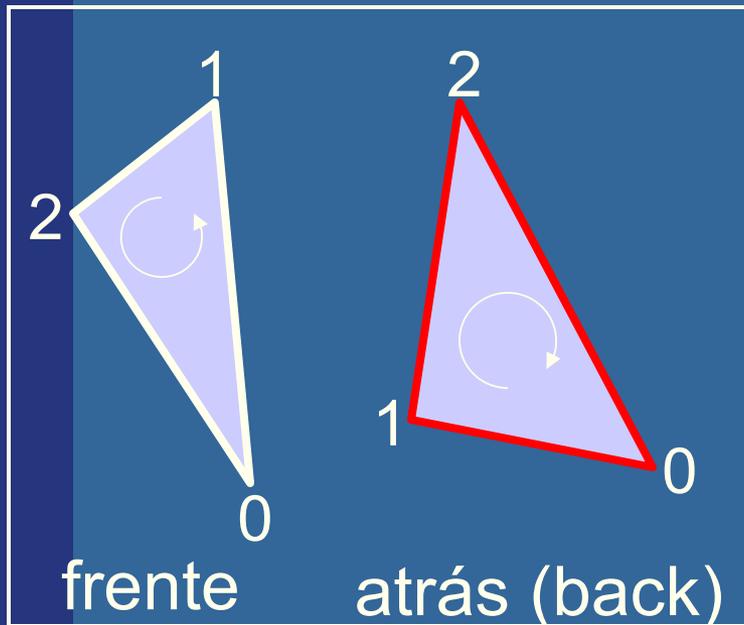
# Backface culling (continuación)

- A menudo implementada en la API
- OpenGL: `glCullFace (GL_BACK) ;`
- ¿Como determinar qué caras no se ven?
- 1º, deben ser polígonos orientados de forma consistente, e.j., contrareloj

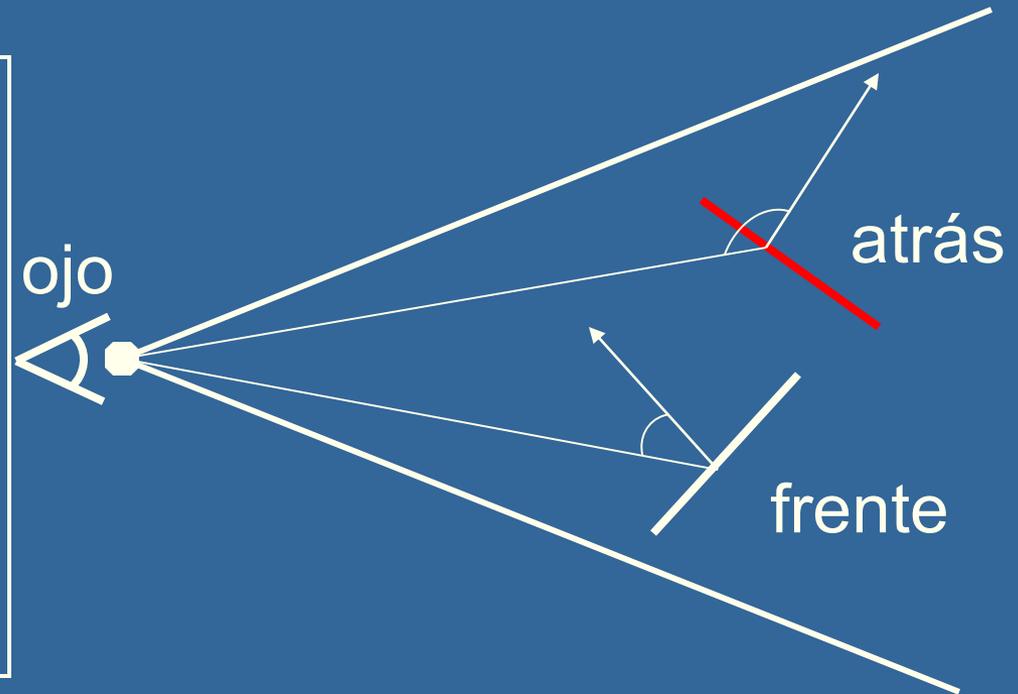


# Cómo hacer cull a backfaces

- Dos maneras en diferentes espacios:



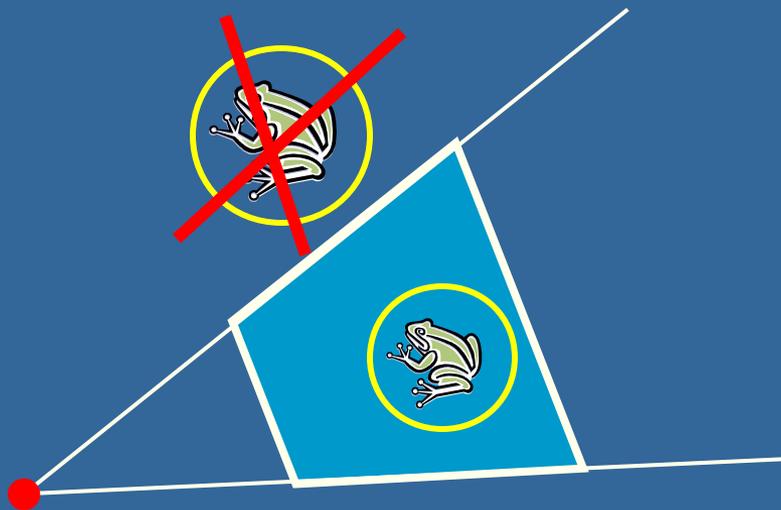
Espacio de pantalla



Espacio de vista

# Culling de la sección de vista (View-Frustum Culling)

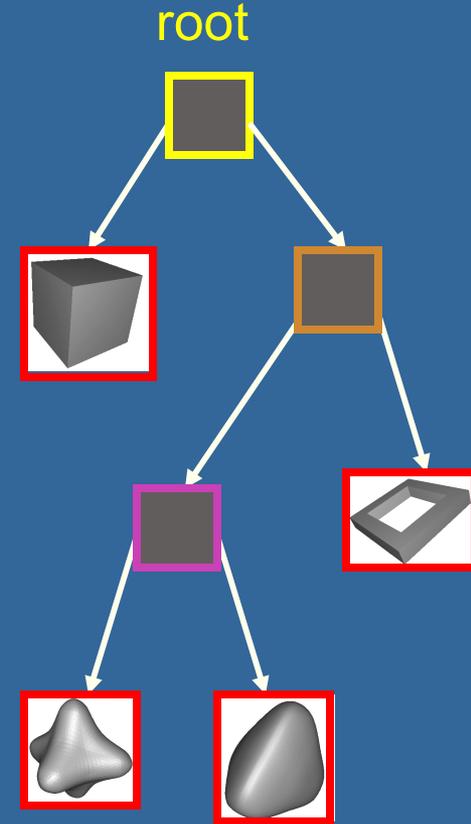
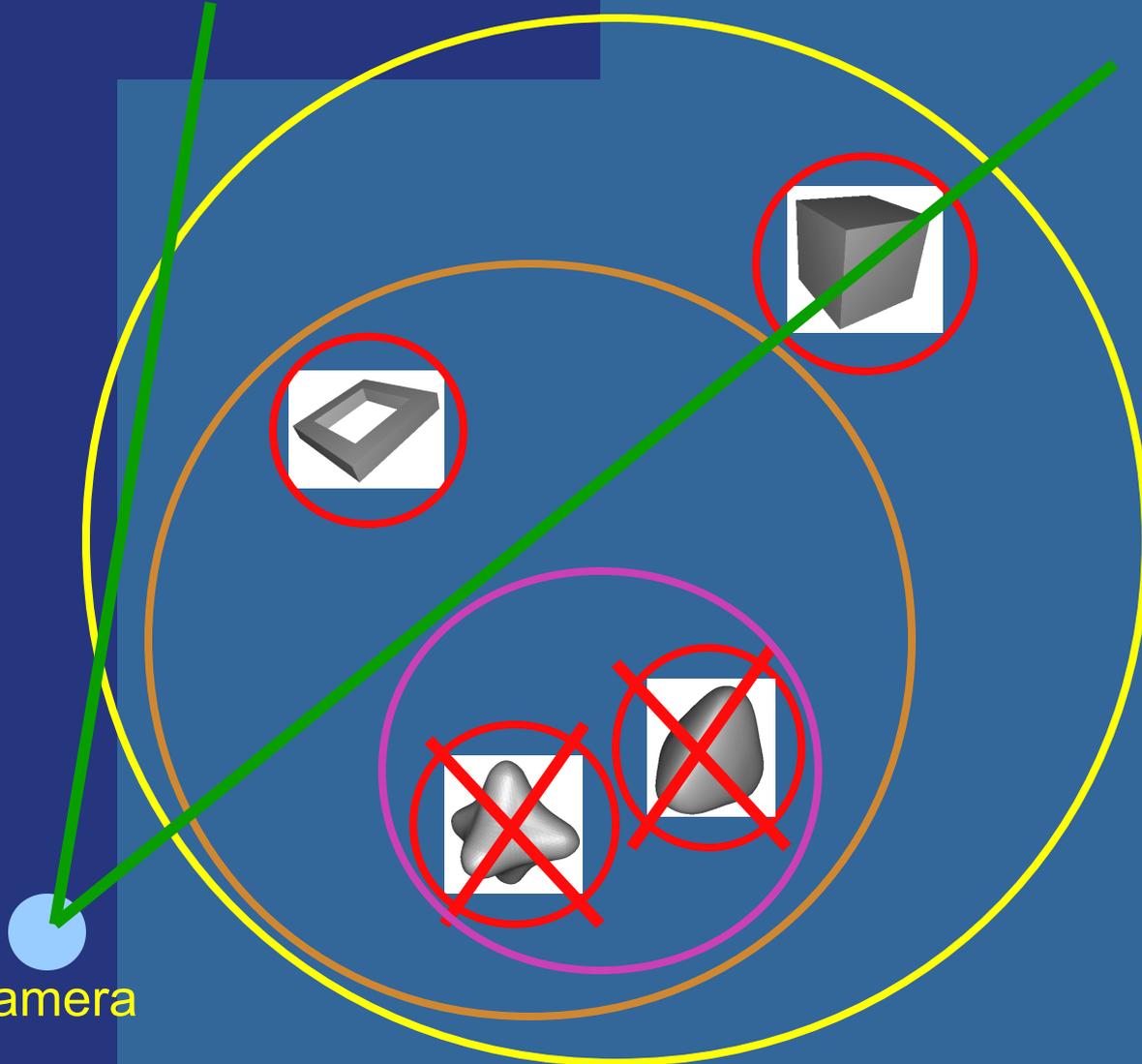
- Encerrar cada grupo “natural” de primitivas en un volumen simple (esfera, caja, etc.)
- Si un volumen acotante (bounding volume (BV)) está fuera de la sección de vista, su contenido también está fuera (no es visible)



# ¿Podemos acelerar la sección de vista aun más?

- Sí, si se utiliza una aproximación jerárquica, como una estructura de datos espaciales (BVH, BSP, scene graph)
- ¿Qué etapas se benefician?
  - Geometría y rasterización
  - El bus entre CPU y Geometría

# Ejemplo de Vista Jerárquica Frustum Culling



# Portal Culling

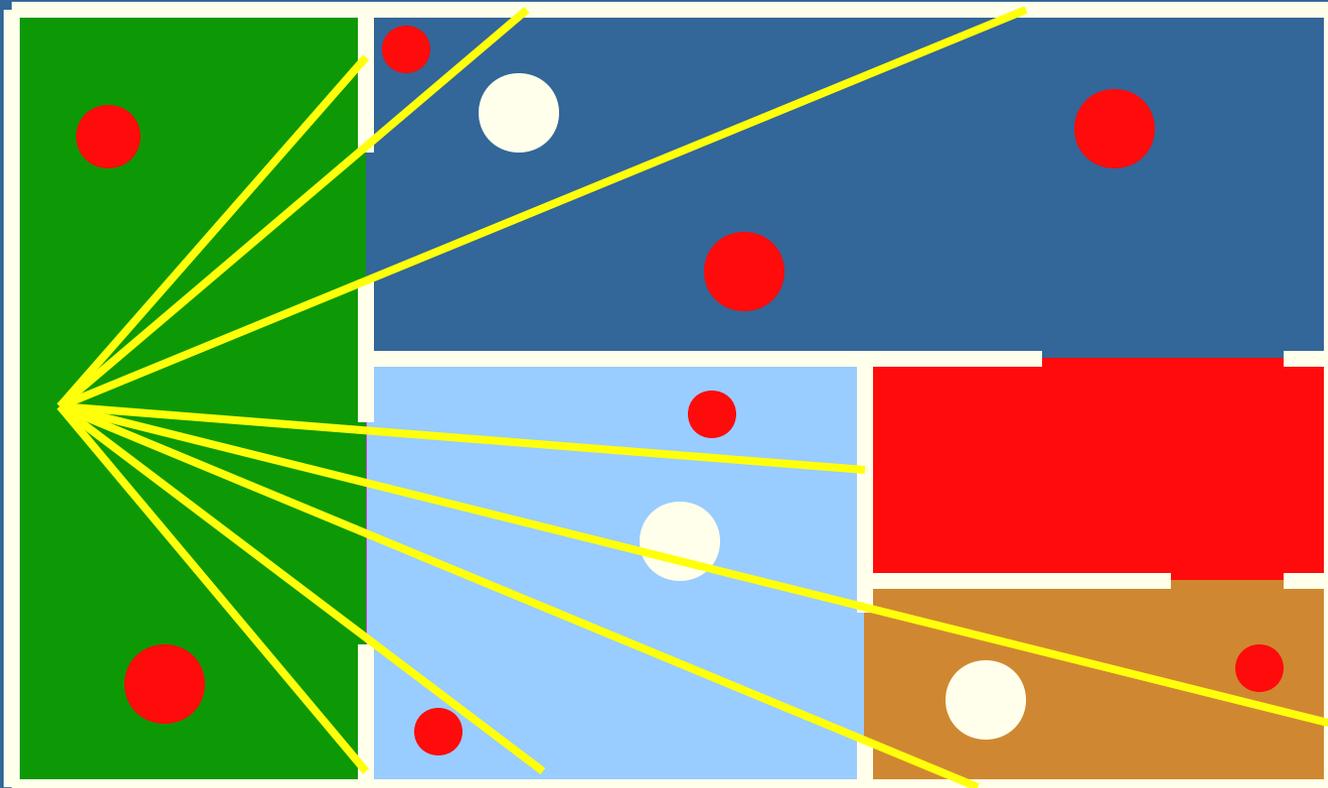
Images courtesy of David P. Luebke and Chris Georges



- Promedio: Son selec. (culled) 20-50% de los polígonos en la vista
- Aceleración: de un poco hasta 10 veces

# Ejemplo de Portal culling

- Un edificio visto desde arriba
- Círculos son objetos a ser renderizados



# Algoritmo de Portal Culling (1)

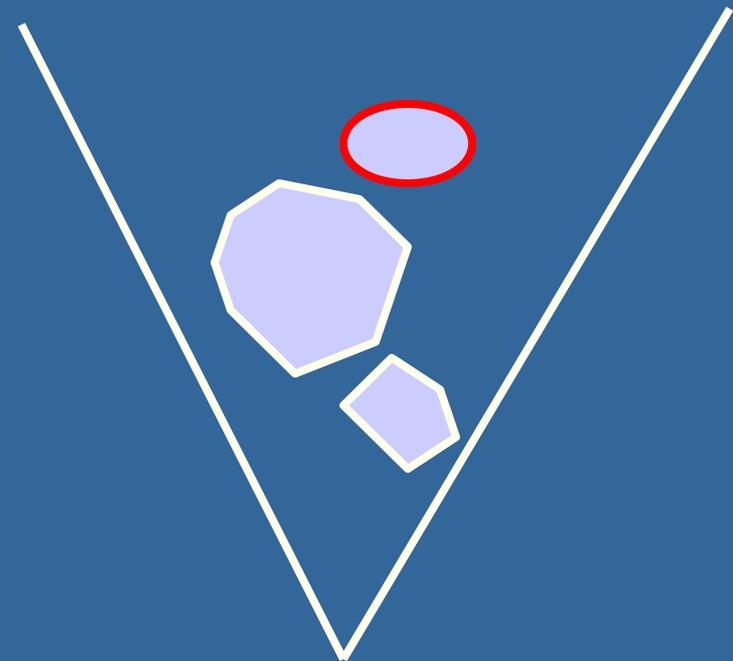
- Dividir en celdas con portales (hacer un grafo)
- Por cada frame:
  - Localizar la celda del espectador e iniciar 2D AABB para la pantalla completa
  - \* Renderizar la celda actual con View Frustum culling con respecto al AABB
  - Recorrer las celdas más cercanas (a través de los portales)
  - Intersecar AABB & AABB de los portales recorridos
  - Goto \*

# Portal Culling Algorithm (2)

- Cuándo terminar:
  - Cuando la actual AABB es vacía
  - Cuando no tenemos tiempo de renderizar una celda (“lejana” para el espectador)
- También: marcar objetos renderizados

# Culling de oclusión (Occlusion Culling)

- Idea principal: Objetos que están por completo detrás de otros pueden ser culled
- Problema difícil de resolver eficientemente
- Mucha investigación en esta área



# Ejemplo

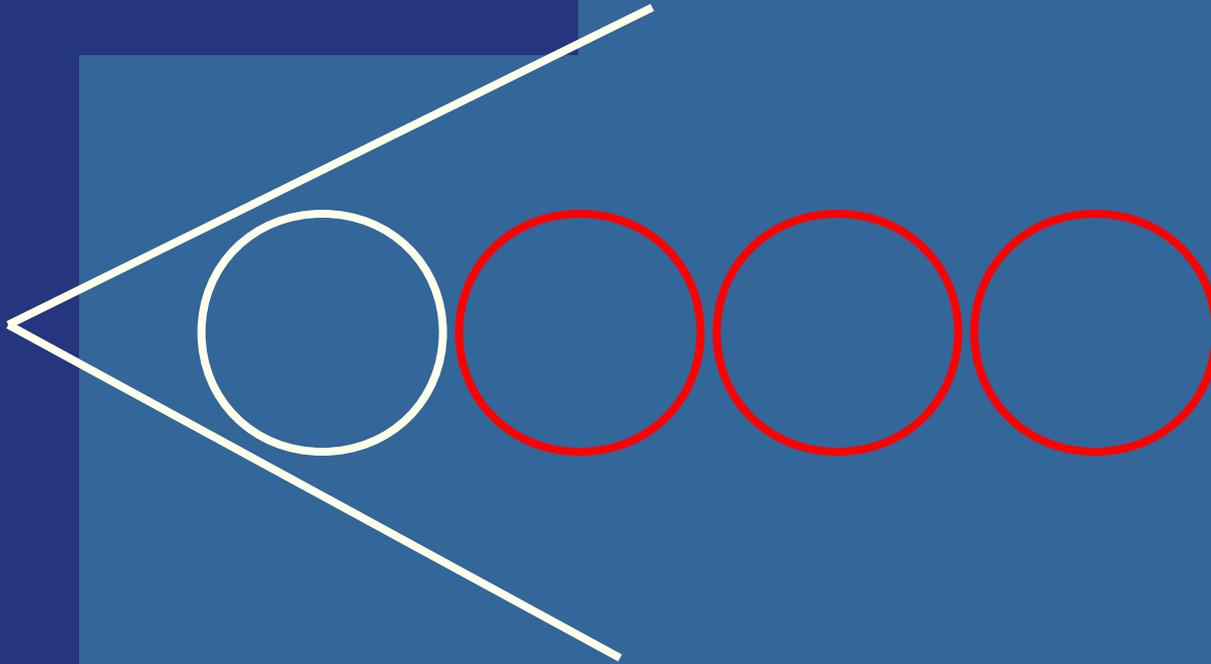
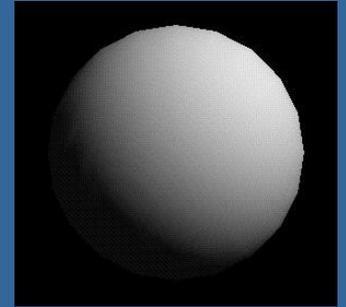


Imagen final



- Notar que “Portal Culling” es un tipo de occlusion culling

# Algoritmo de culling de oclusión

Utilizar algún tipo de representación de la oclusión  $O_R$

Por cada objeto  $g$  hacer:

```
if( not Ocluido( $O_R, g$ ))
```

```
    render( $g$ );
```

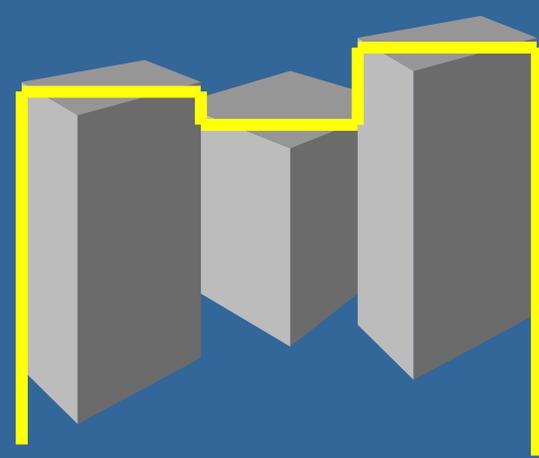
```
    update( $O_R, g$ );
```

```
end;
```

```
end;
```

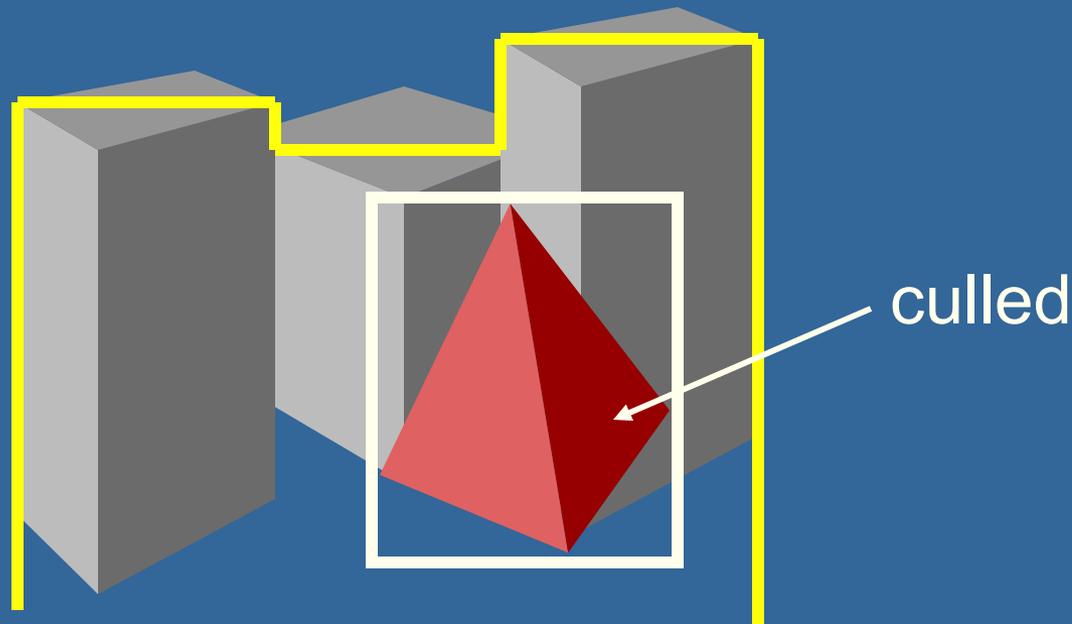
# Horizontes de Oclusión: Un algoritmo simple

- Objetivo: paisaje urbano
  - Oclusión densa
  - Espectador está aprox. a 2 metros sobre el piso
- Algoritmo:
  - Procesar escena de adelante a atrás utilizando un quad tree
  - Mantener un horizonte constante a trozos
  - Cull objetos contra el horizonte
  - Agregar el contorno de los objetos visibles al horizonte



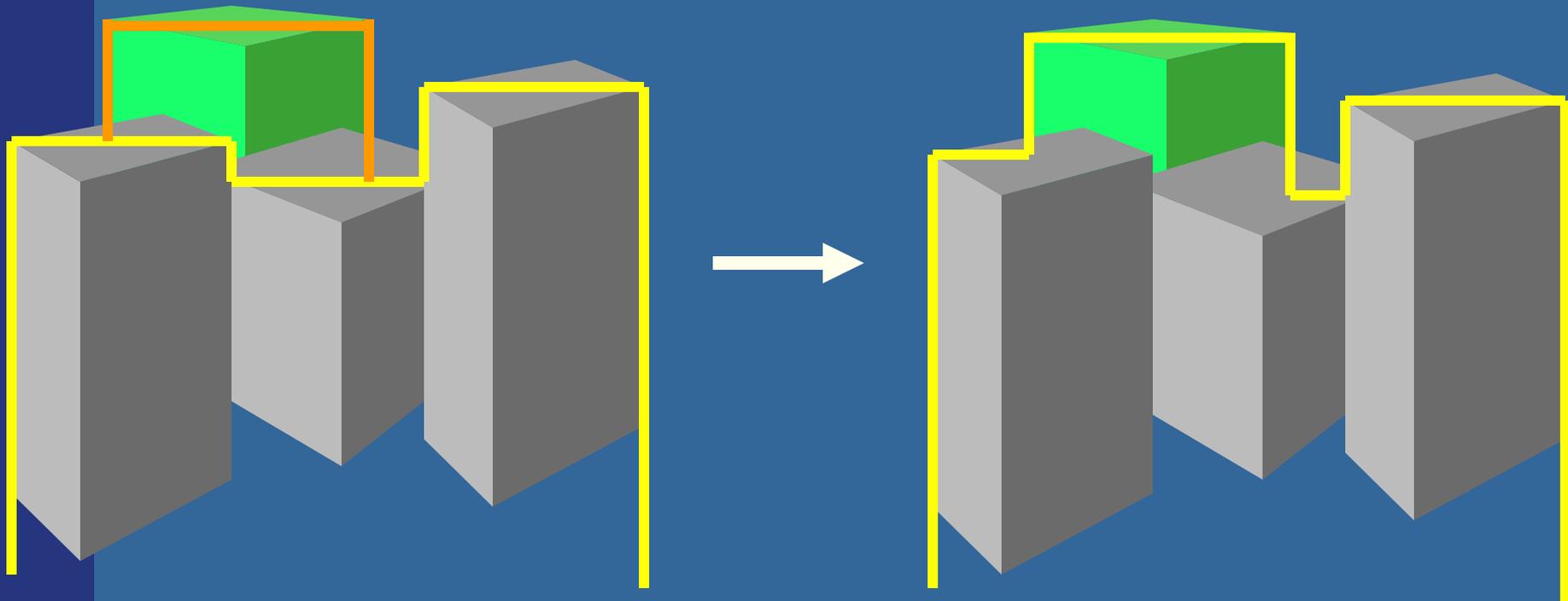
# Test de oclusión con horizontes de oclusión

- Para procesar al tetraedro (que está detrás de los objetos grises):
  - Encontrar una caja de proyección alineada a los ejes
  - Comparar contra el horizonte de oclusión

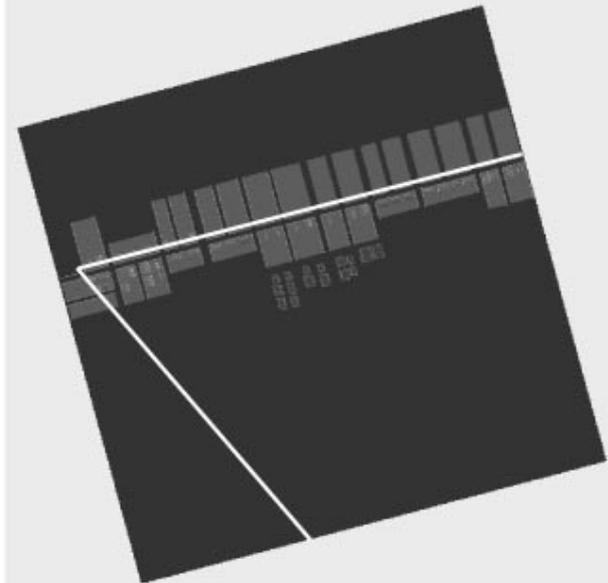


# Actualizar horizonte

- Cuando un objeto es considerado visible:
- Agregar su “poder de oclusión” a la representación de la oclusión



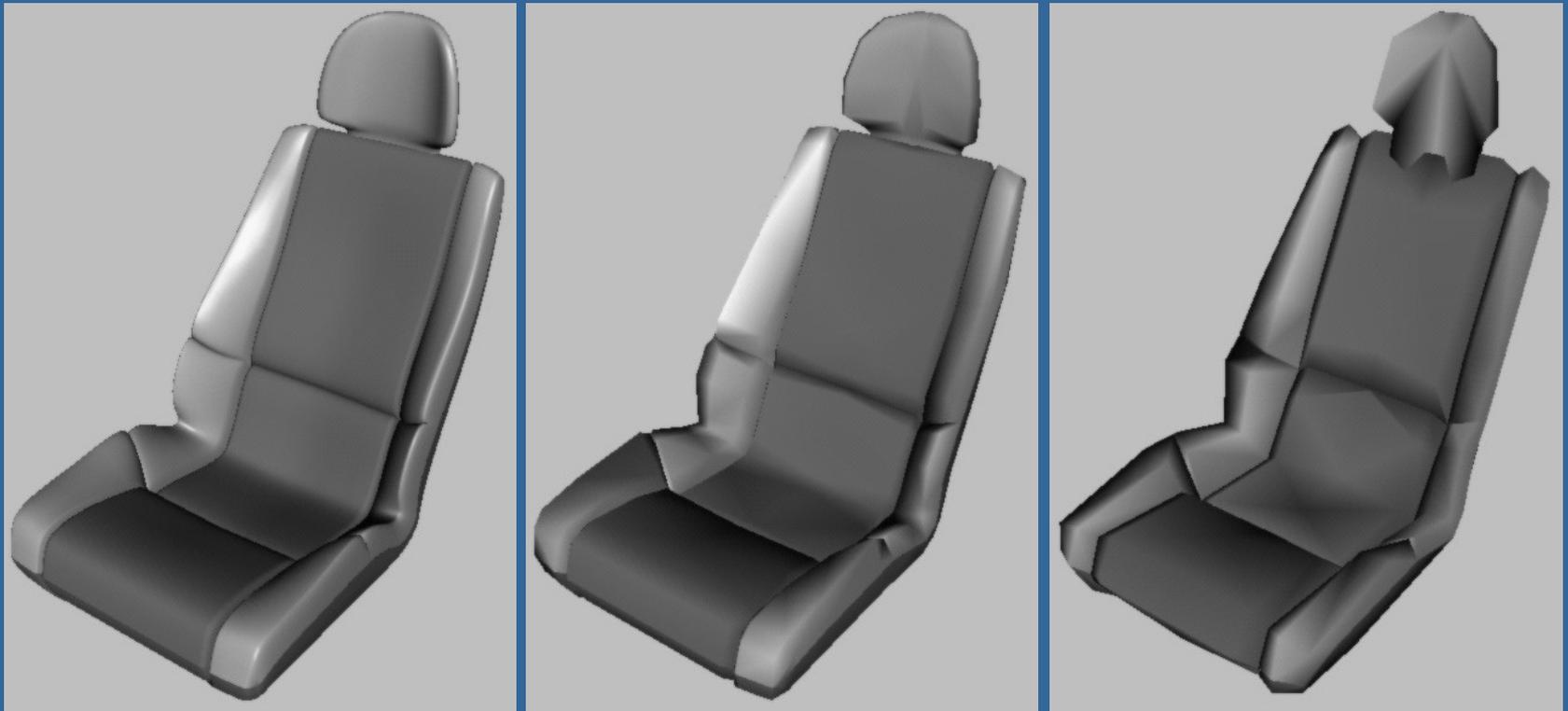
# Ejemplo:



Leer el capítulo del libro

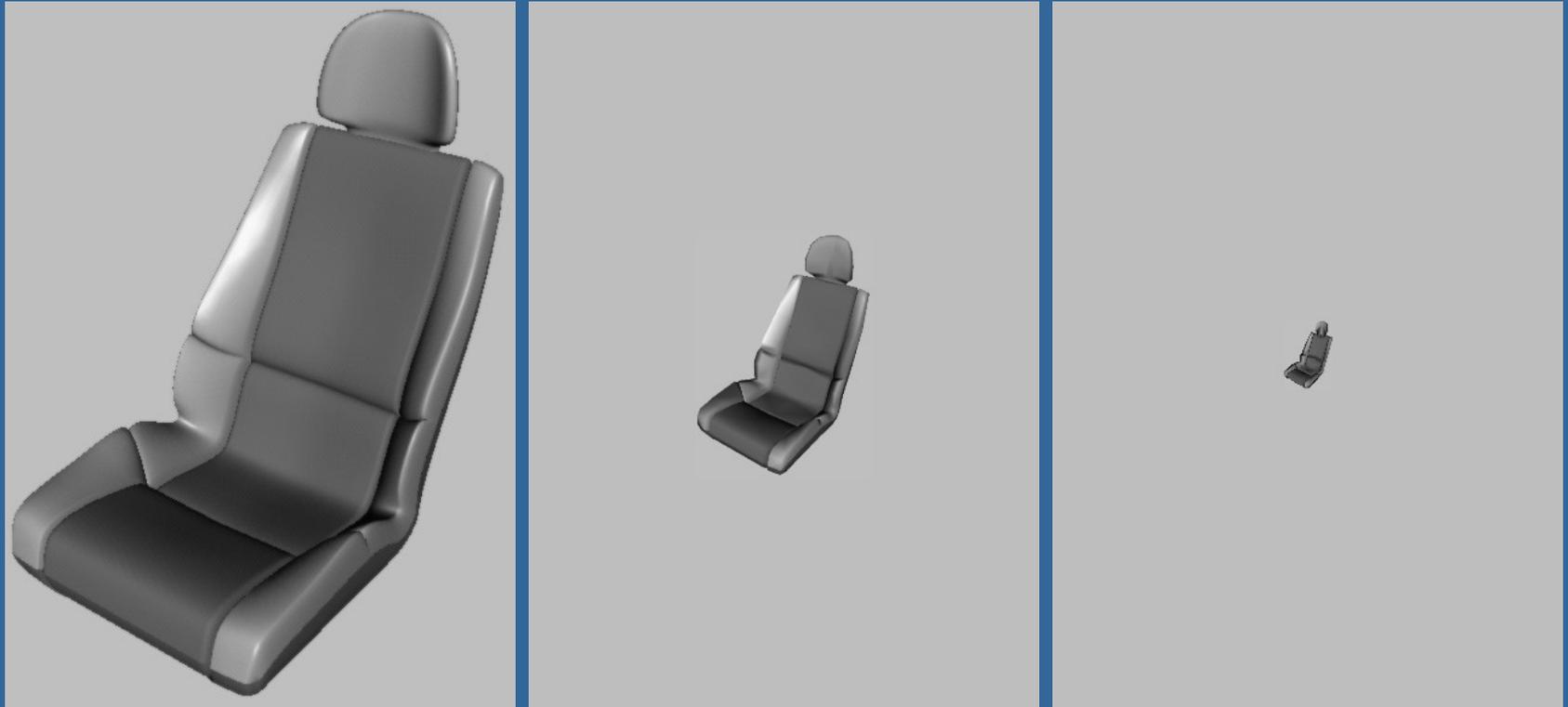
# Rendering de nivel de detalle (Level-of-Detail Rendering)

- Utilizar niveles de detalle diferentes a distancias de usuario diferentes
- Más triángulos cuanto más cerca del espectador



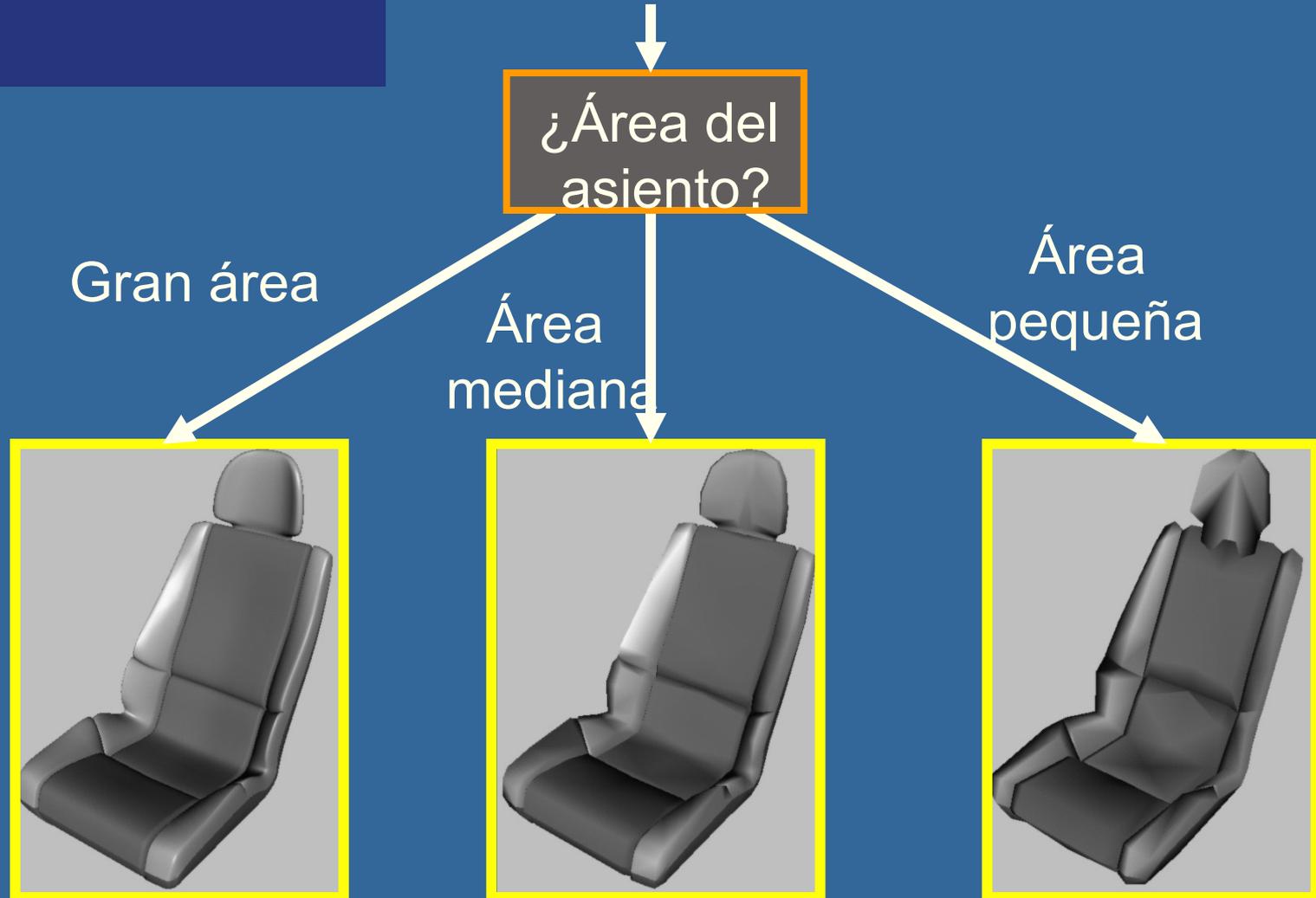
# LOD rendering

- Casi no hay diferencia, pero mucho más rápido



- Usar área de proyección del BV para seleccionar el LOD apropiado

# Grafo de escena con LODs



# LOD rendering para objetos lejanos

- Cuando el objeto está muy alejado, reemplazarlo con un cuadrángulo de algún color
- Cuando el objeto está *realmente muy alejado*, no renderizarlo (detail culling)!
- Utilizar área proyectada de BV para determinar cuándo saltarlo