

CC3501

Detección de

Colisiones

Eduardo Graells-Garrido
Primavera 2023



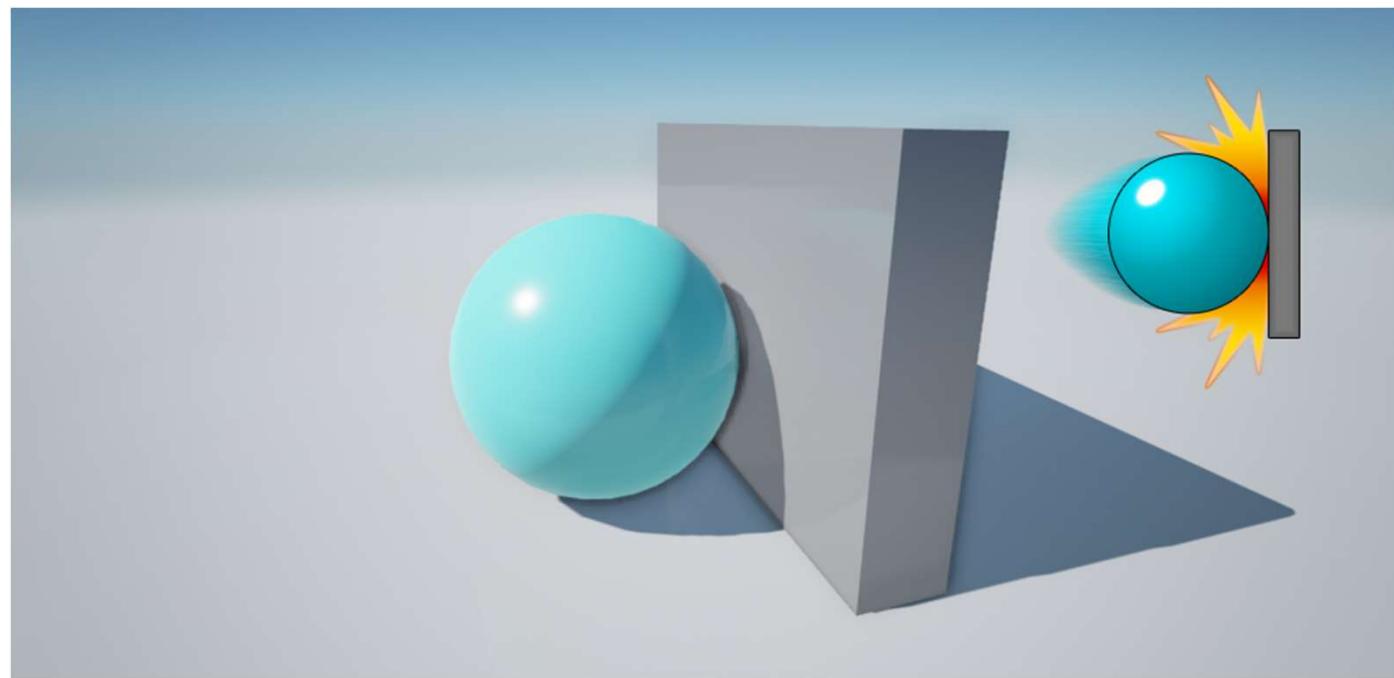
Hermann and Dorothea (1981), Wolfgang Lettl

Introducción

Las colisiones añaden realismo a una escena, ya que permiten que los objetos interactúen.

Cuando una colisión es detectada, el sistema hace que los objetos involucrados reaccionen de manera similar a como lo harían en la realidad.

[DEMO](#)

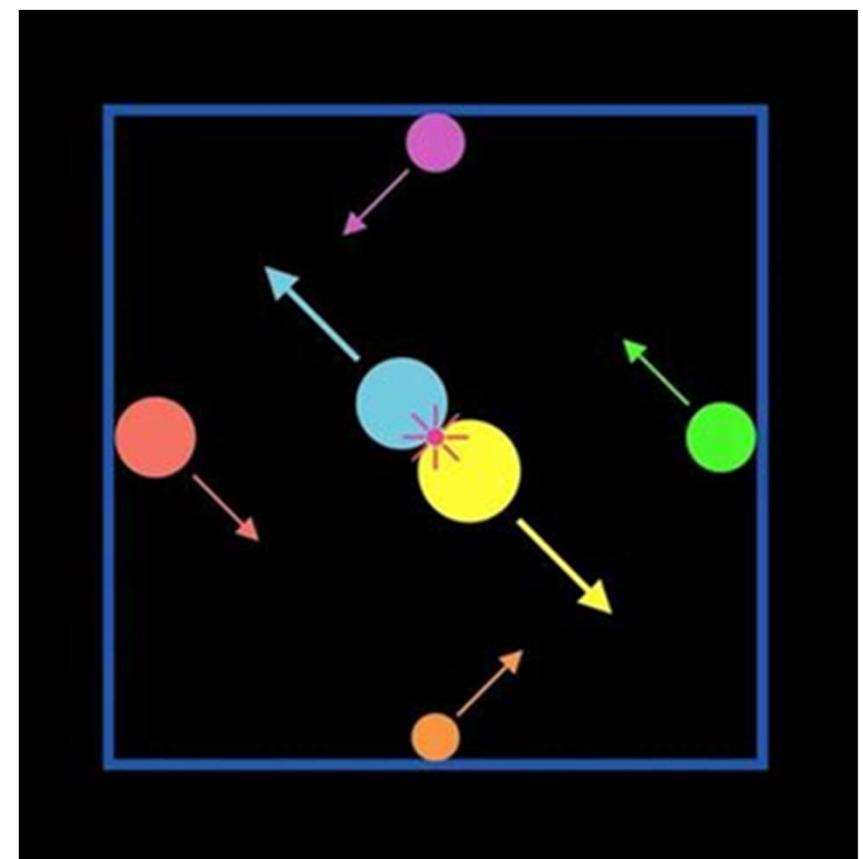
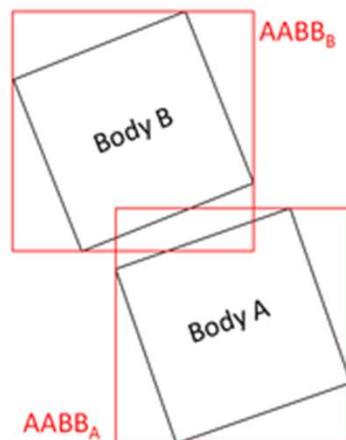
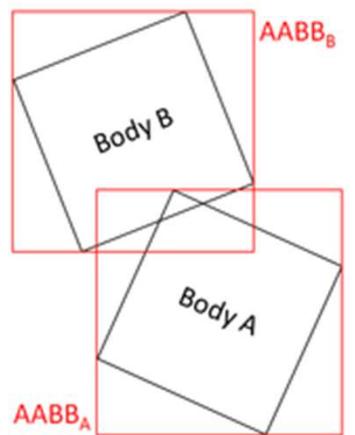


Introducción

Este proceso suele tener dos etapas:

Primero, **detectar si hay una colisión**, es decir, saber si dos objetos chocaron.

Segundo, **determinar la respuesta física de la colisión**, es decir, definir qué sucede con los objetos luego del choque.

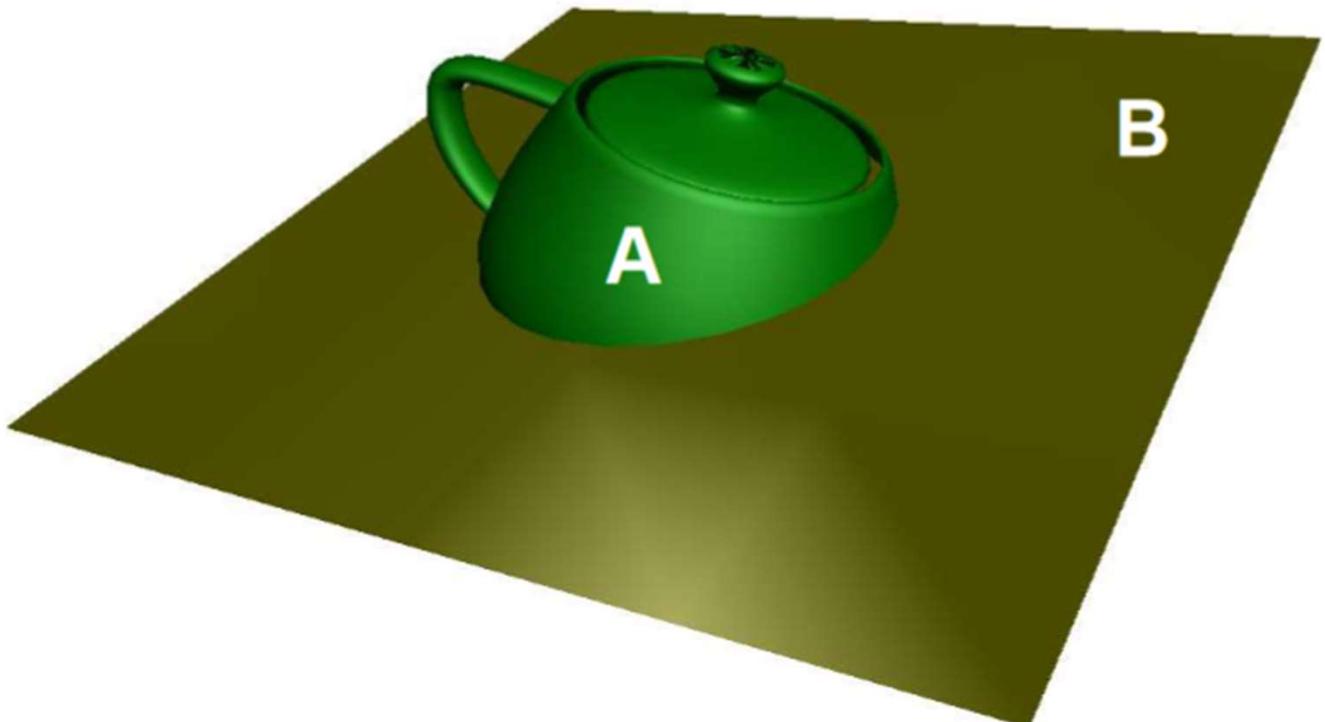


Colisión Discreta

Queremos saber si A y B chocan.

Para ello comparamos sus posiciones.

A o B pueden moverse, sin embargo, **el cálculo lo hacemos para un instante del tiempo**. Una *snapshot*.

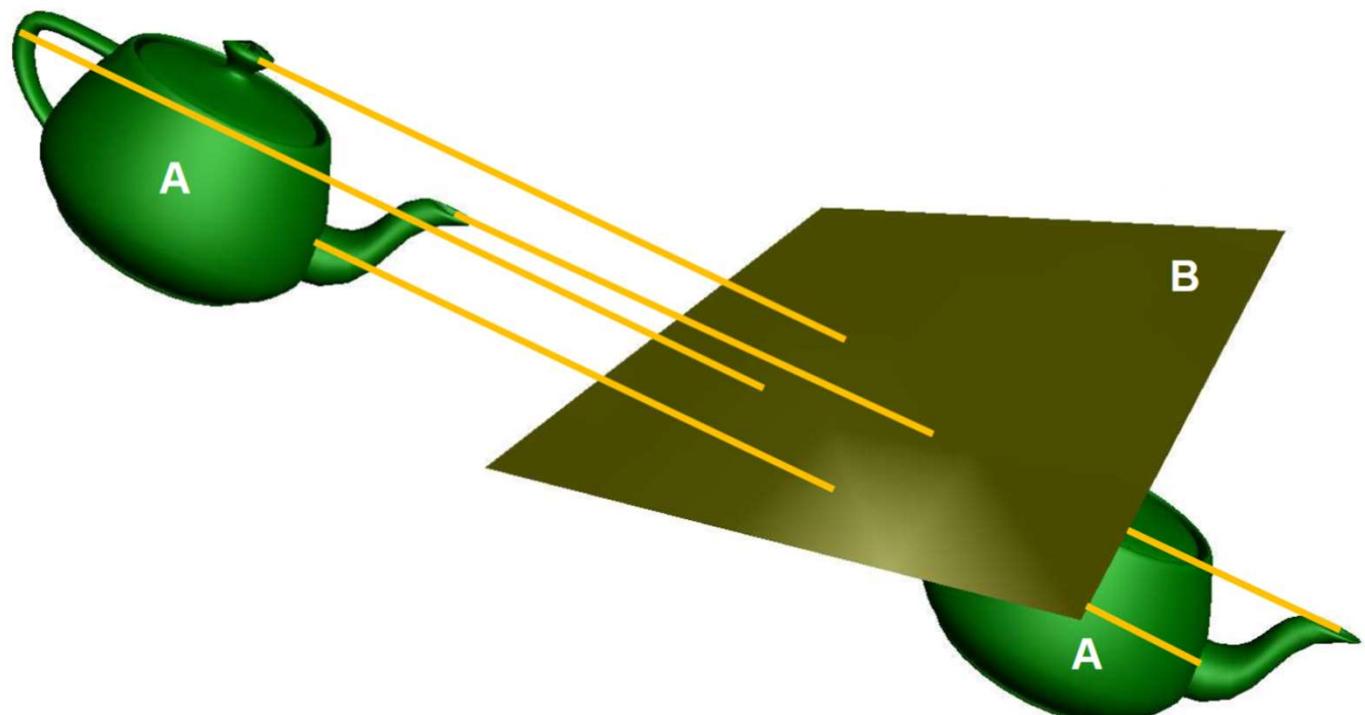


Colisión Continua

En este caso A es un objeto que se mueve, que durante su movimiento se interseca con B.

Este caso es más difícil que el anterior, puesto que la graficación podría hacerse antes y después de la colisión.

Por esto, usualmente un sistema de simulación de física tiene un reloj independiente del rendering.



¿Cómo detectar colisiones?

1. Hay que **rechazar colisiones** de manera **muy rápida**.
2. Hay que utilizar **representaciones jerárquicas** (ya veremos por qué) para **rechazar colisiones rápidamente**.
3. Para el caso **discreto**: **comparar las aristas del objeto A con todas las caras del objeto B**.
4. Para el caso **continuo**: crear *pseudo-aristas* que conectan los puntos del objeto A en los instantes de tiempo involucrados en la simulación, y luego comparar las *pseudo-aristas* de A con todas las caras del objeto B.

Pero antes...
Consultas Geométricas

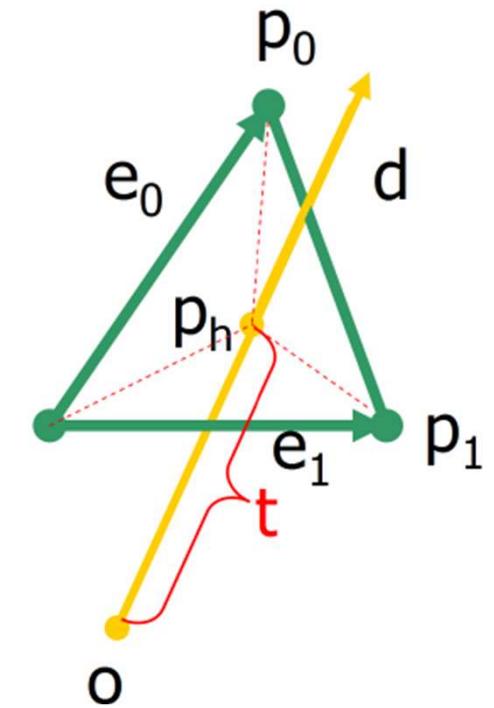
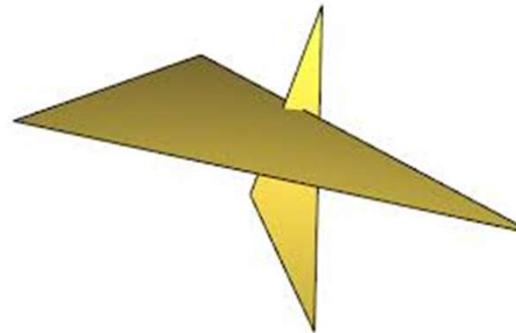
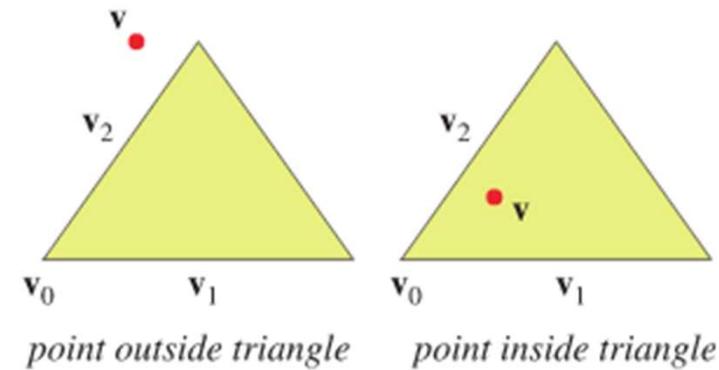
Consultas Geométricas

Tenemos **geometrías diferentes**

(puntos, líneas, triángulos, superficies, volúmenes) y queremos saber:

- **¿Se intersectan?** Si es así, ¿cómo? ¿dónde?
- **¿Una contiene a otra?** Si es así, ¿cuánto?

A diferencia de una operación geométrica (como subdivisión o simplificación), **una consulta geométrica no interviene a un objeto**, sino que produce una **respuesta**.



Ejemplo: el punto más cercano a una línea

El problema es encontrar el punto de una malla que es más cercano a un punto consultado.

Propuesto: derivar la ecuación.

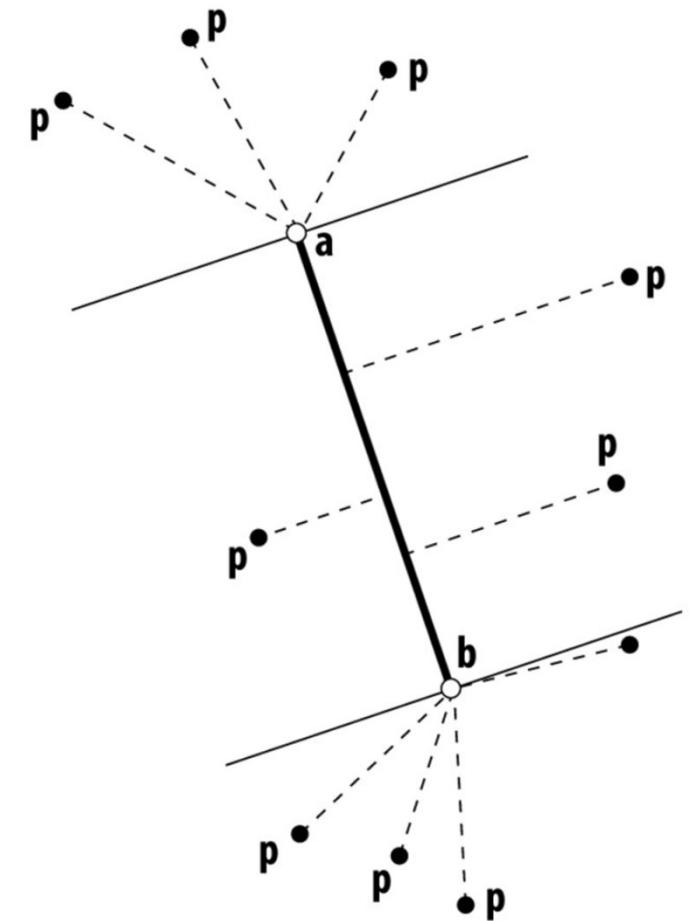
El punto más cercano dentro de un segmento de línea

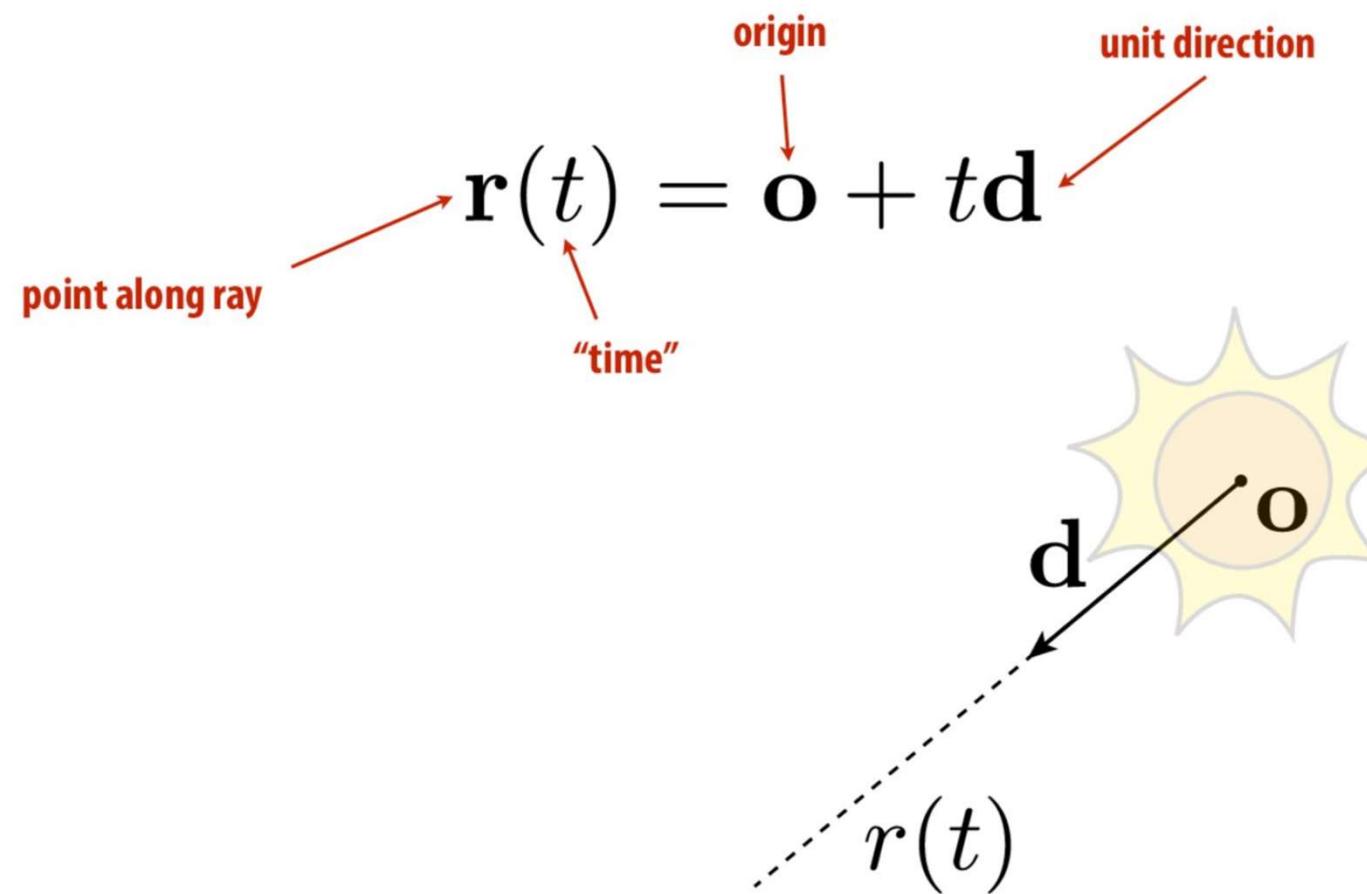
Hay dos casos:

- 1) Está más cerca de un punto interior;
- 2) Está más cerca de uno de sus extremos.

Un algoritmo puede ser:

- 1) Encontrar el punto más cercano a la línea;
- 2) Verificar si su proyección está dentro del segmento
(podemos **parametrizar** el segmento entre 0 y 1);
- 3) Elegir el punto de acuerdo al valor de t (< 0 ; $0 \leq t < 1$; ≥ 1).





El punto más cercano en un triángulo

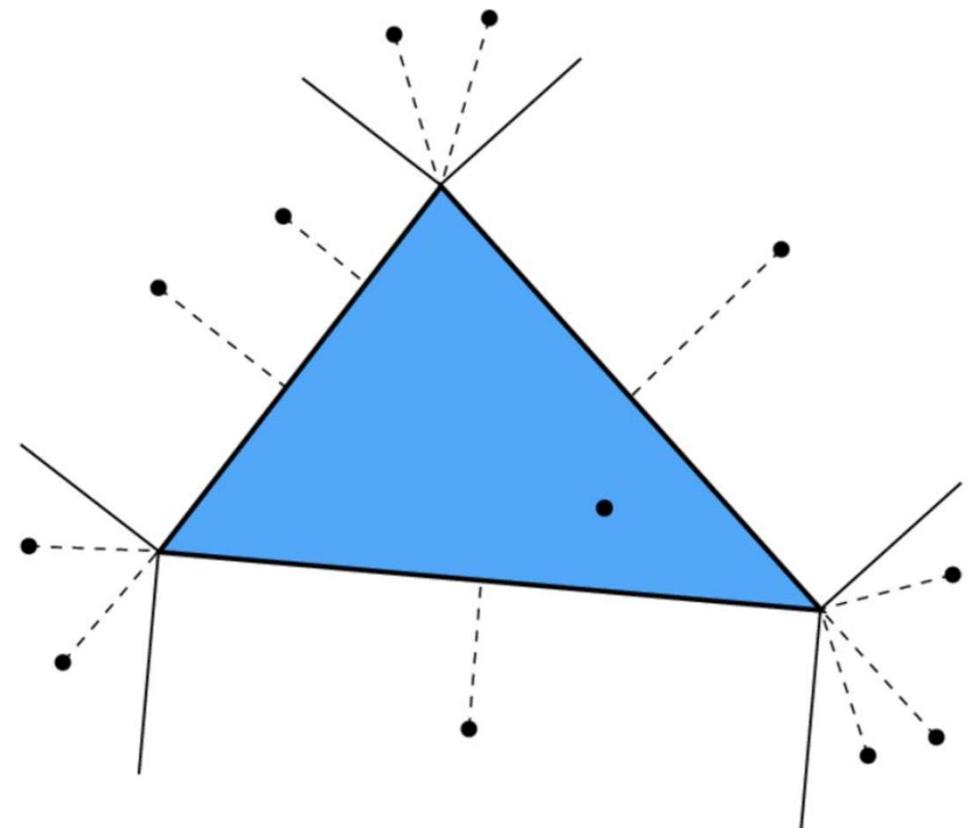
¿Cuáles son todas las posibilidades para punto más cercano a un punto de consulta p?

Si el punto está afuera del triángulo, calcular el punto más cercano en cada una de sus aristas. De los tres, elegir el más cercano.

Propuesto: ¿qué hacer cuando el punto está adentro del triángulo?

Para 3D, primero hay que proyectar el punto de consulta al plano del triángulo.

Luego se puede hacer un test con coordenadas baricéntricas. ¿Se les ocurre otro?

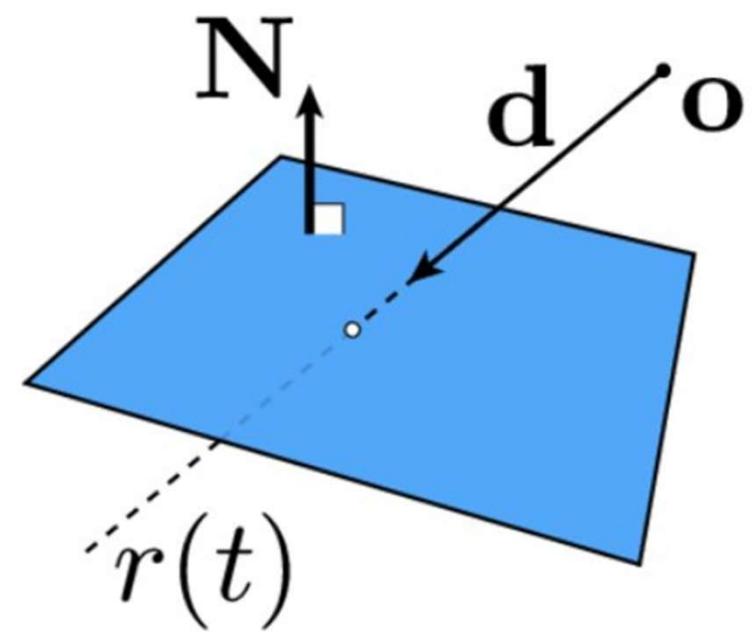


$$\mathbf{N}^T \mathbf{x} = \mathbf{c}$$

$$\mathbf{N}^T \mathbf{r}(t) = \mathbf{c}$$

$$\mathbf{N}^T (\mathbf{o} + t\mathbf{d}) = \mathbf{c}$$

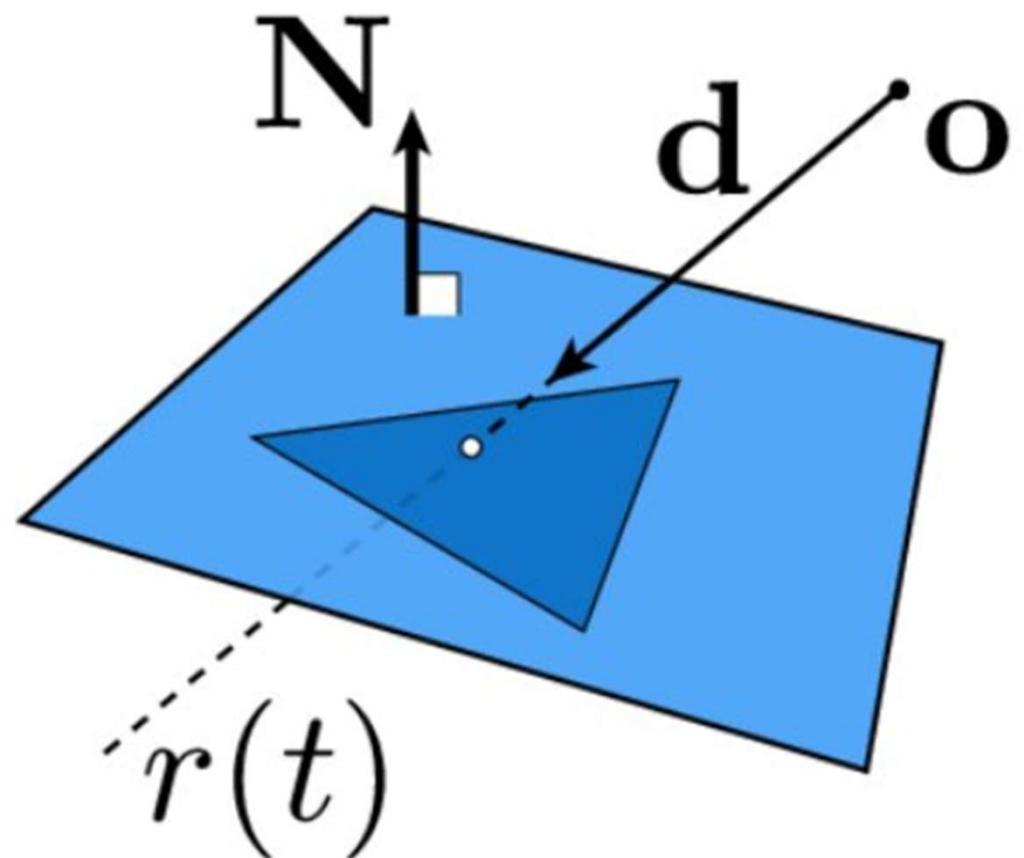
$$r(t) = \mathbf{o} + \frac{c - \mathbf{N}^T \mathbf{o}}{\mathbf{N}^T \mathbf{d}} \mathbf{d}$$



Intersección Rayo-Triángulo

El triángulo está en un plano. Por tanto la operación es la misma, el resultado dependerá de si el punto $r(t)$ está dentro del triángulo.

Una manera de resolver ese problema es calculando las coordenadas baricéntricas de $r(t)$. Si todas las coordenadas son positivas, el punto está dentro del triángulo.



¿Cómo encontrar el punto más cercano en una malla?

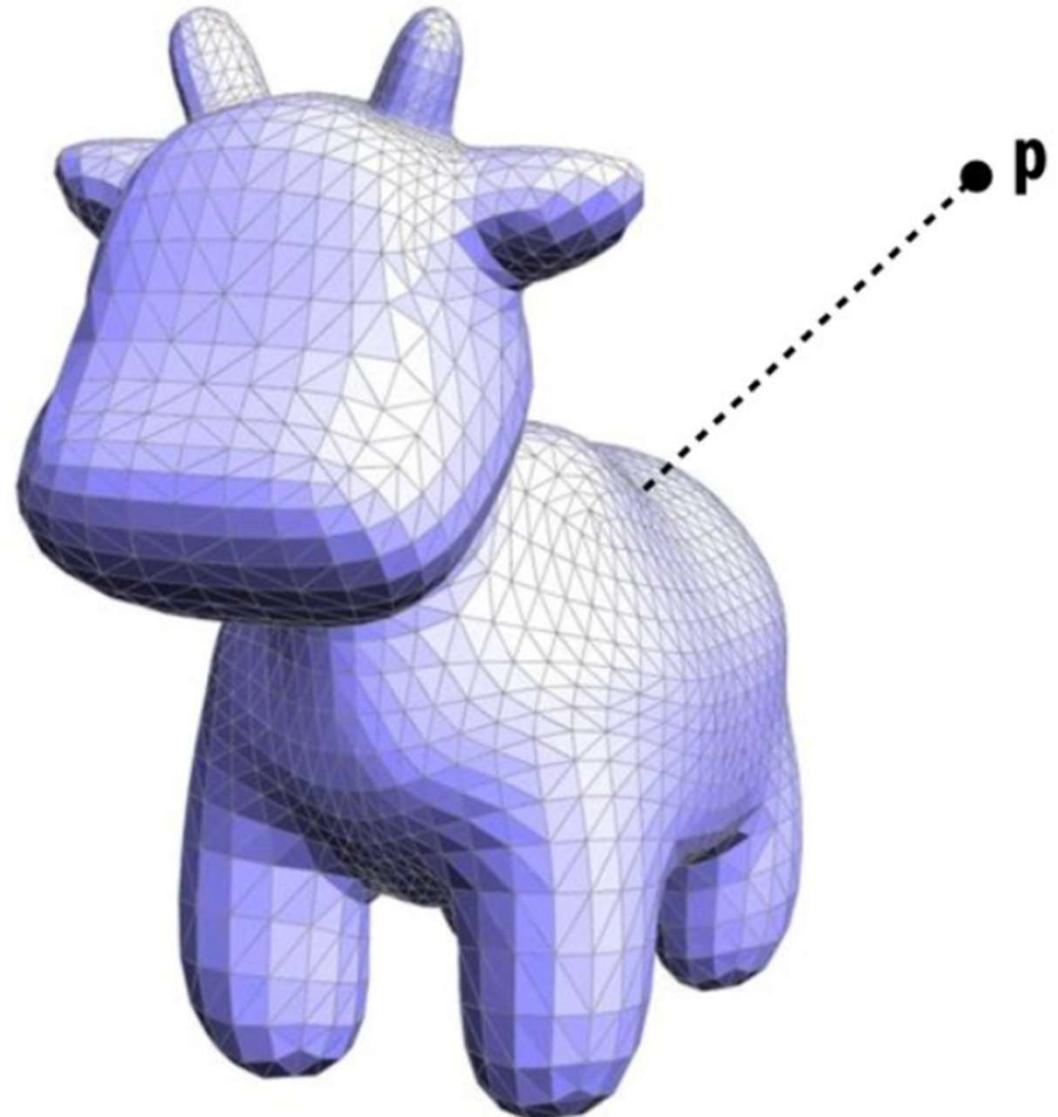
Conceptualmente es sencillo:

- Aplicar la consulta anterior a cada triángulo
- Quedarnos con el triángulo y el punto correspondiente que estén más cercanos a p

¿Cuánto cuesta hacer esto?

¿Qué pasa si tenemos millones de triángulos?

Necesitamos estructuras de datos que permitan responder esto de manera eficiente.



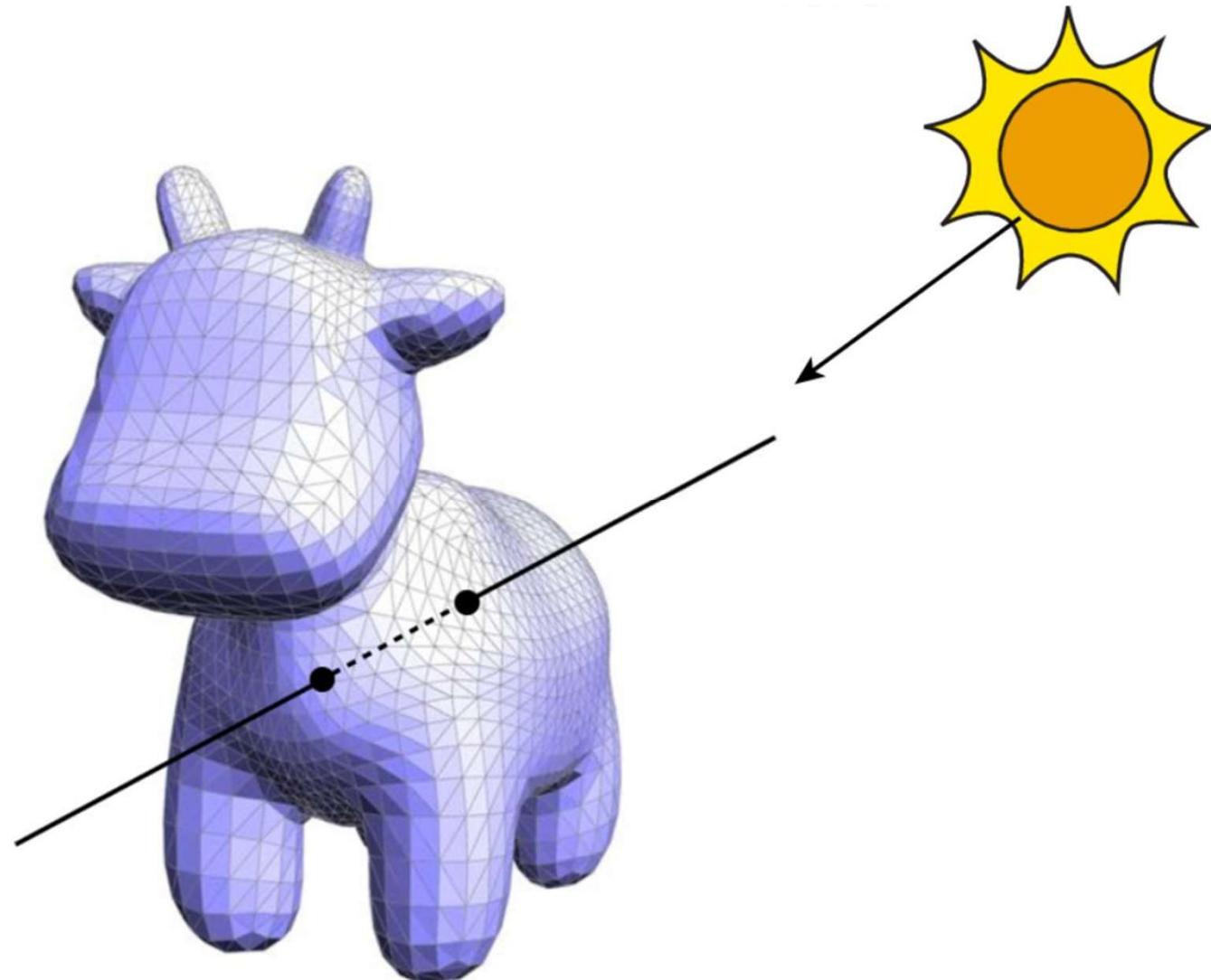
Intersección entre rayo y malla

Un rayo es una línea con orientación que comienza en un punto. **¡Como las fuentes de iluminación!**

Queremos saber los puntos de una superficie que son atravesados por un rayo. ¿Por qué?

- Geometría: test de orientación de malla, qué está dentro y qué está afuera.
- Graficación: visibilidad, ray tracing (recordemos que hay superficies con transparencia y la luz tiene distintas maneras de reflejarse en una superficie)
- Animación: detección de colisiones.

¡La intersección no es única!





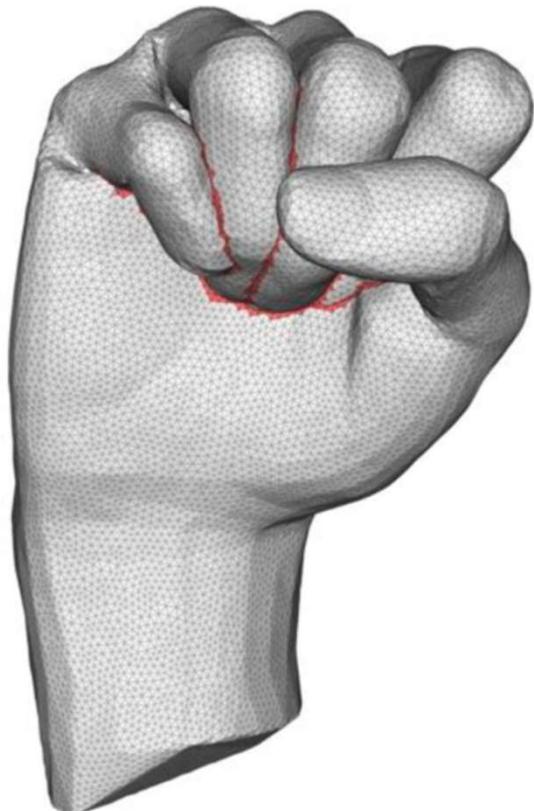
Demo “Brigade 3”, una de las primeras muestras de ray tracing en tiempo real (¡hace 10 años!). Y lo lograron con implementaciones eficientes de test de intersección de rayos. <https://www.youtube.com/watch?v=aKqxonOrI4Q>

Intersección malla-malla

¿Cómo sabemos si una malla se intersecta a sí misma?

¿Cómo sabemos si una malla intersecta a otra?

¿Cómo sabemos si hubo una colisión y cuál es su efecto?



Intersección Triángulo-Triángulo

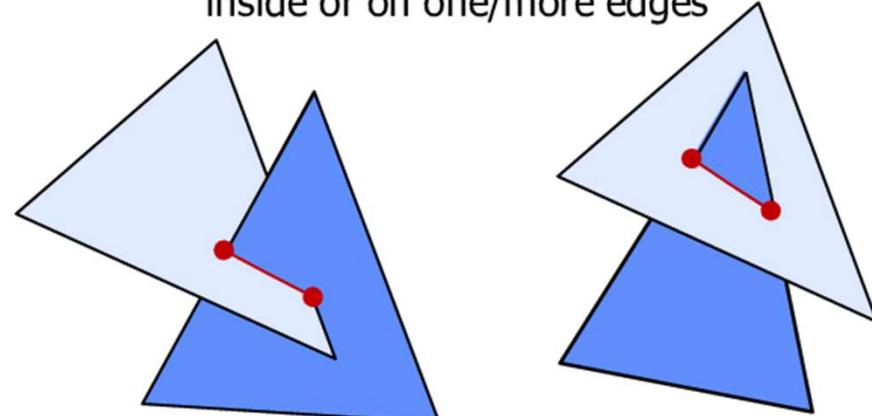
Una manera básica: **intersectar las aristas de un triángulo con el plano del otro**. Si es así, hacer cálculo de intersección de planos.

Cuando los triángulos son coplanares, el resultado no es un segmento, es un polígono.

Cuando tenemos triángulos en movimiento, se puede definir el prisma de movimiento (estela) que deja (o tendrá) un triángulo, y tratar el problema como una intersección de poliedros.

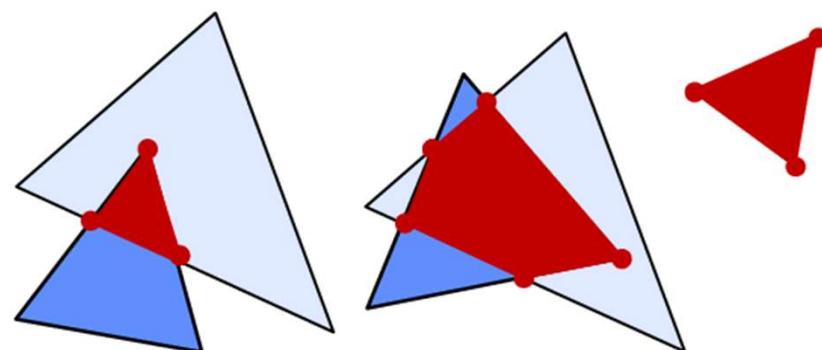
Non Coplanar (3D)

2 intersection points, either inside or on one/more edges

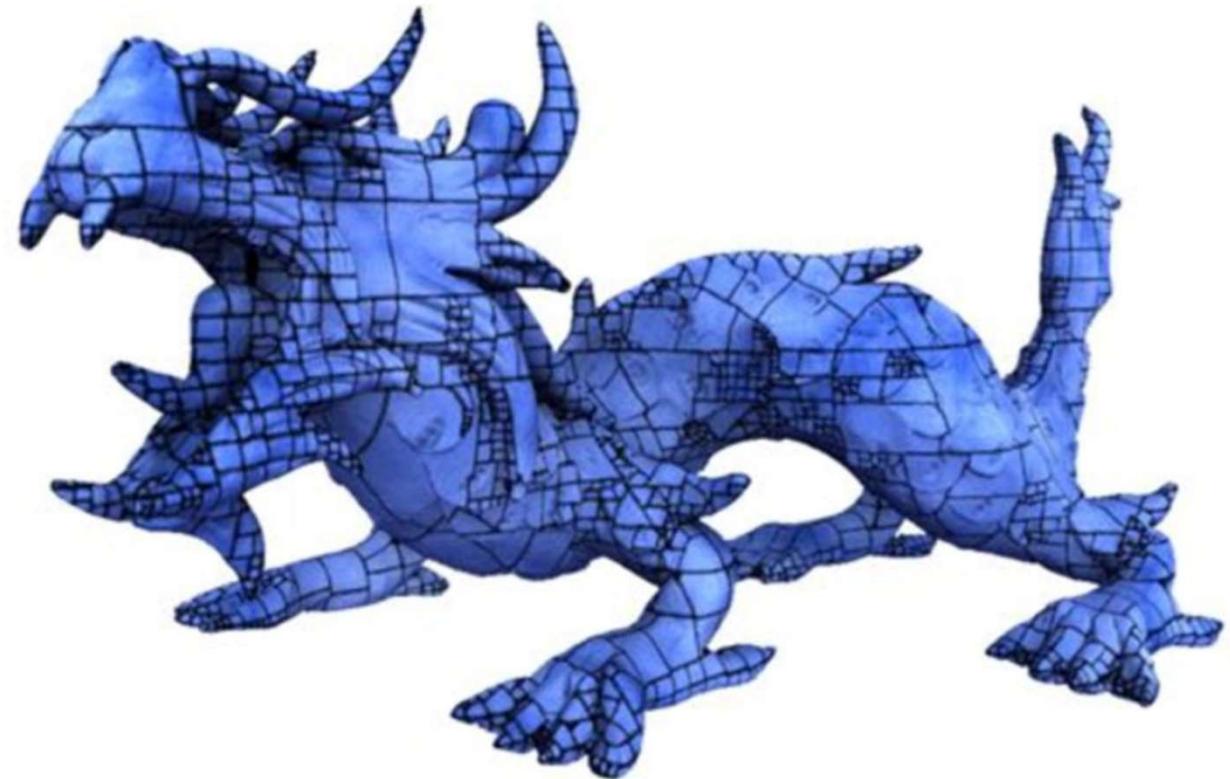
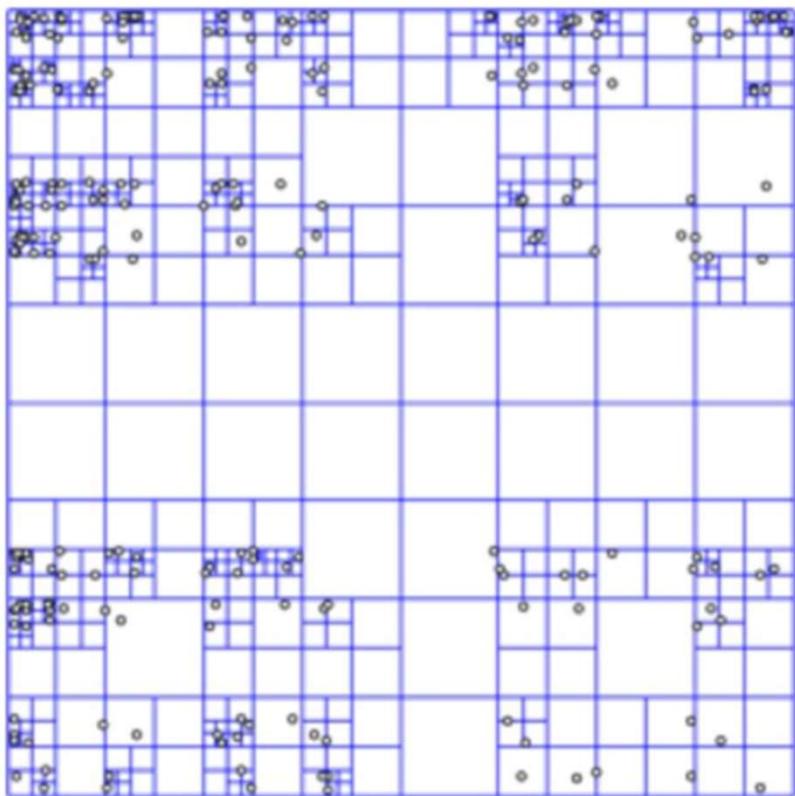


Coplanar (2D)

An intersection *polygon*, could be a whole triangle if fully inside



Representaciones Jerárquicas



¿Cómo se aceleran las operaciones? No necesariamente con otras fórmulas, sino cambiando la unidad de consulta.

Detección Rápida

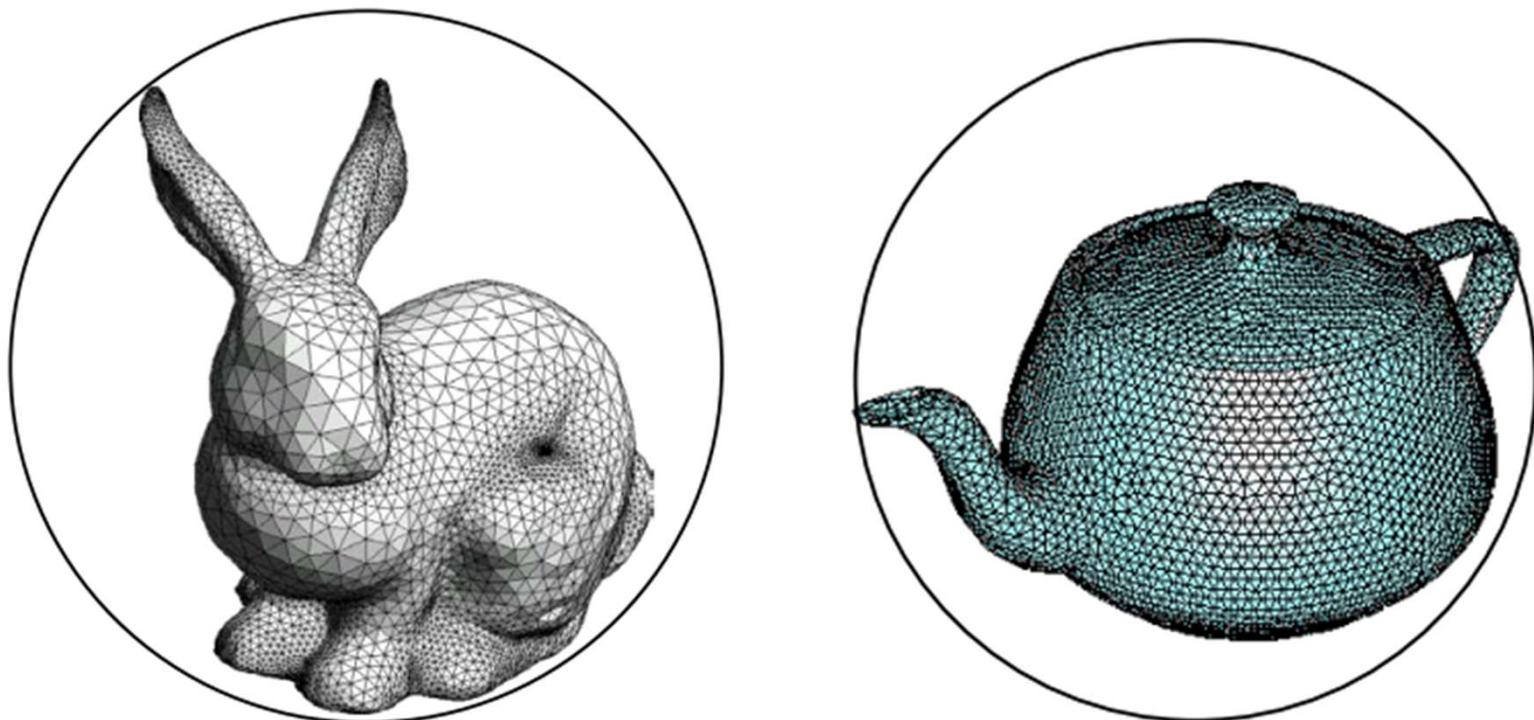
Los objetos de una escena podrían ser geométricamente complejos, por lo que realizar un análisis de colisiones exacto será computacionalmente costoso.

Una solución aproximada es **usar una geometría proxy para cada objeto** (esferas, prismas rectangulares, etc.).

Es más fácil hacer la detección de colisiones entre formas simples que entre formas complejas.

Descarte **MUY** Rápido

Como parte del pre-procesamiento de un objeto, se calcula **la esfera más pequeña que lo contiene completamente**. Se conoce como *bounding sphere*.



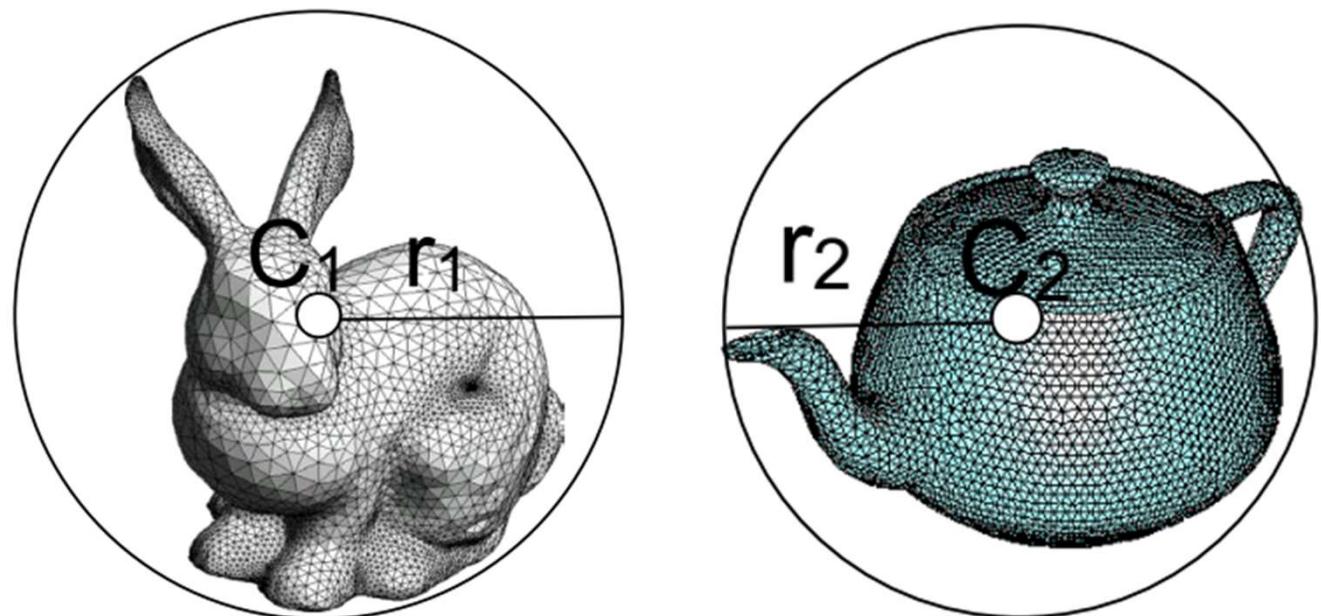
Colisiones de Esferas

Si tenemos dos esferas con centros C_1 y C_2 , de radios r_1 y r_2 .

Las esferas se *intersectan* si $\|C_1C_2\| < r_1 + r_2$

Este tipo de test presenta muchos falsos positivos. Sin embargo, no tiene falsos negativos: si las esferas contenedoras no se intersectan, los objetos no pueden intersectarse (*por definición!*).

Así, es un test muy rápido que nos permite descartar interacción.



$$X_{max_1} > X_{min_2} \text{ } \&\& \text{ } Y_{max_1} > Y_{min_2}$$
$$\text{ } \&\& \text{ } X_{max_2} > X_{min_1} \text{ } \&\& \text{ } Y_{max_2} > Y_{min_1}$$

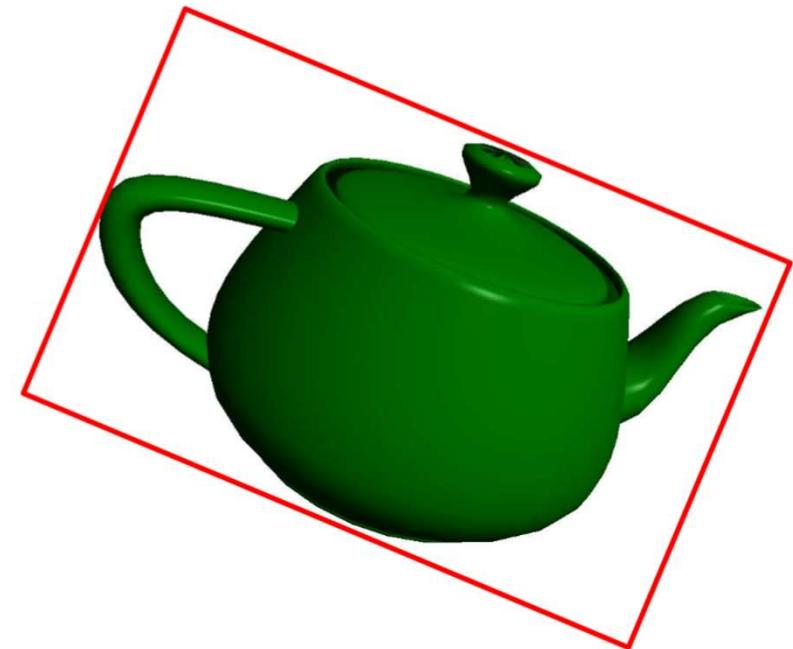
Bounding Boxes

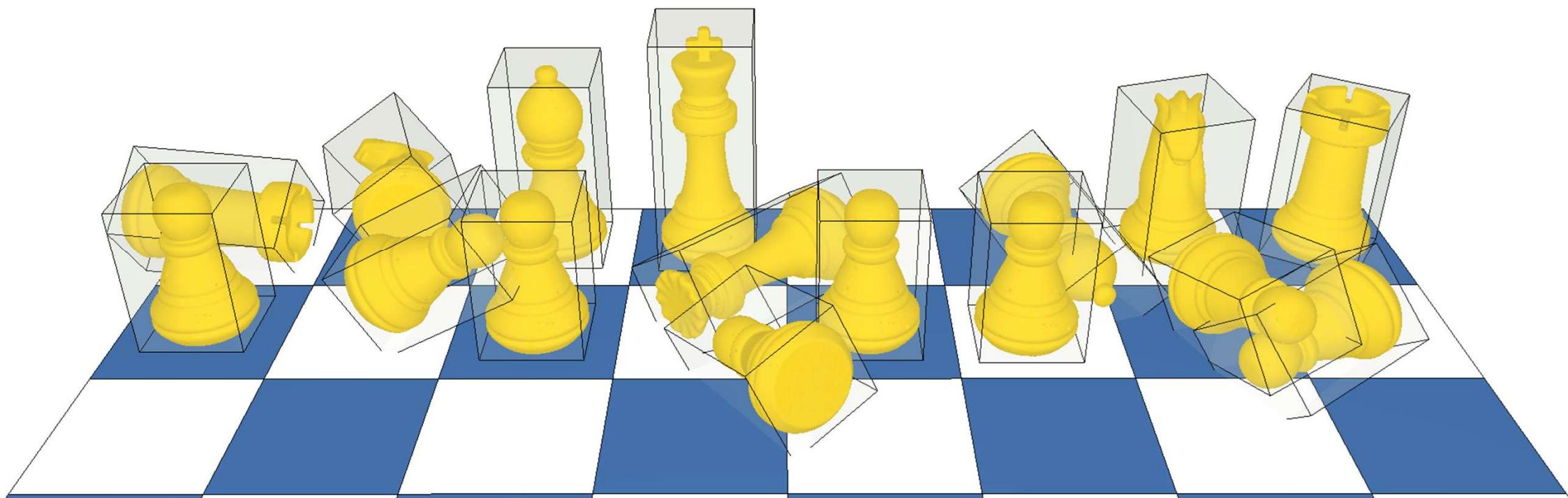
El mismo ejercicio se puede hacer con prismas rectangulares o **cajas**.

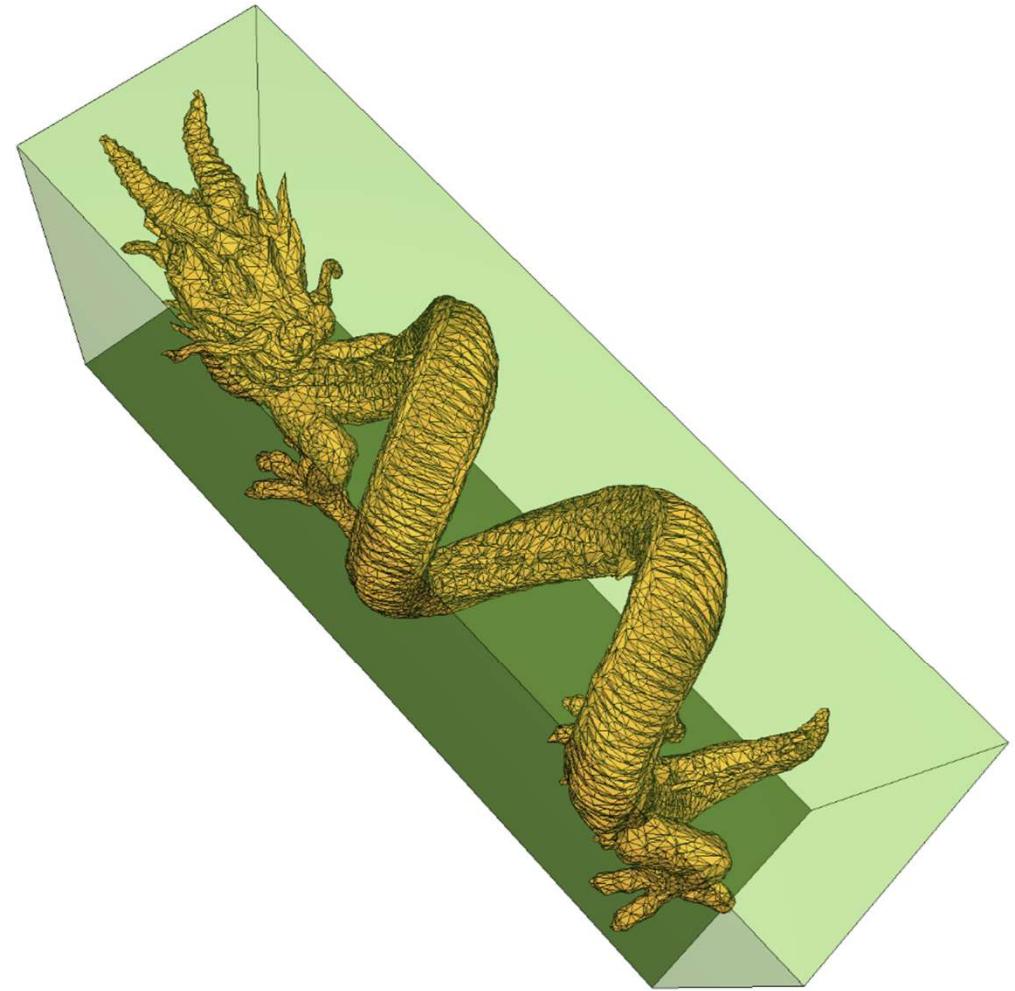
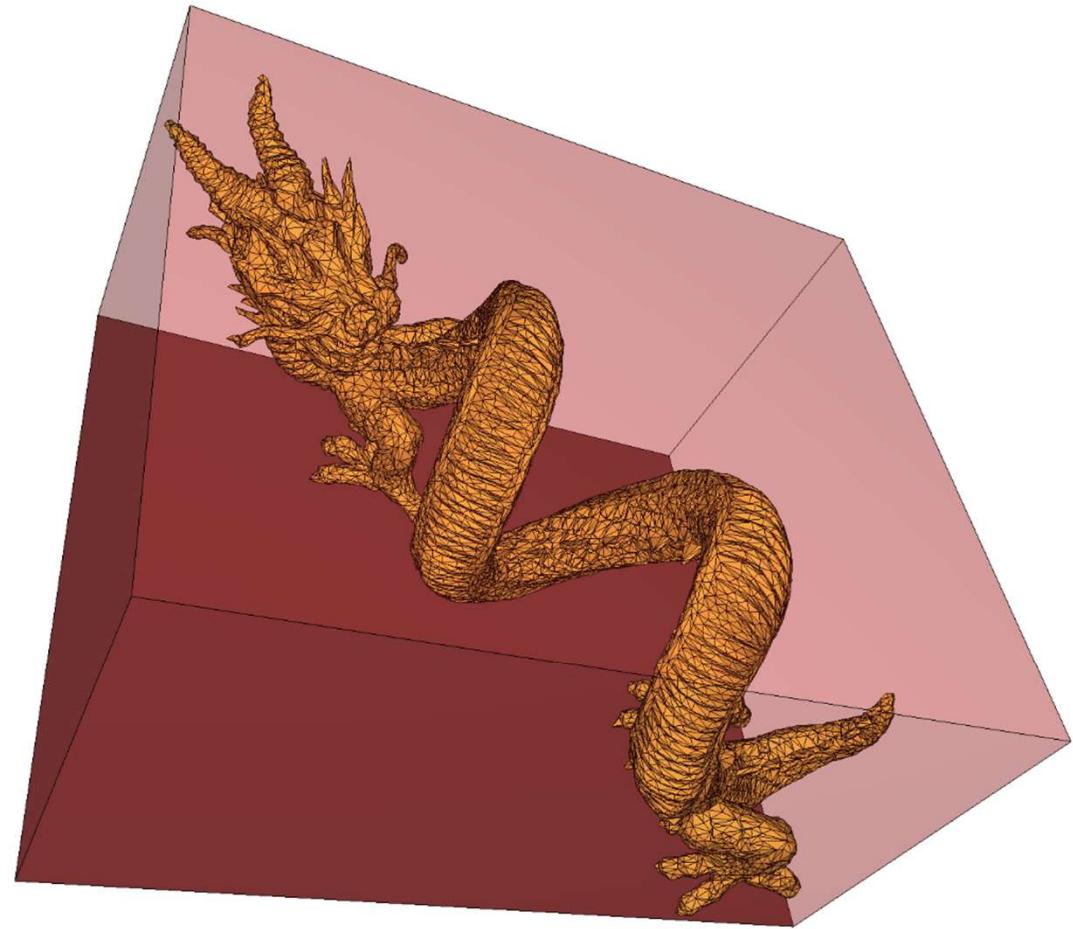
Existen dos tipos de cajas:

AABB: Axis-Aligned Bounding Box, donde las aristas de cada caja están alineadas con los ejes de la escena.

AOBB: Arbitrary-Oriented Bounding Box, donde la caja de orienta en el sistema de coordenadas del objeto, permitiendo que el ajuste sea mayor (pero a un mayor costo de comparación).





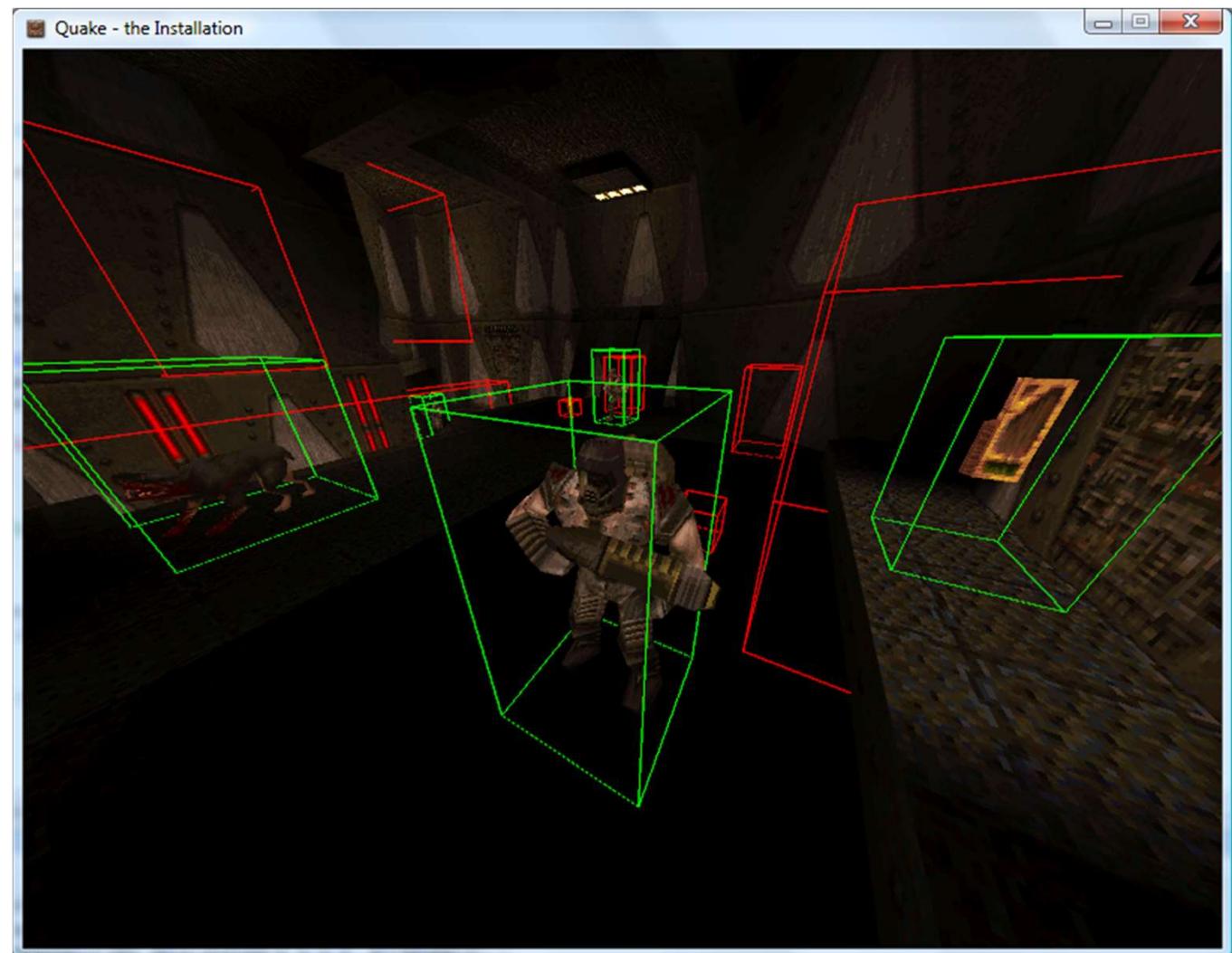


En ocasiones no es necesaria una colisión exacta.

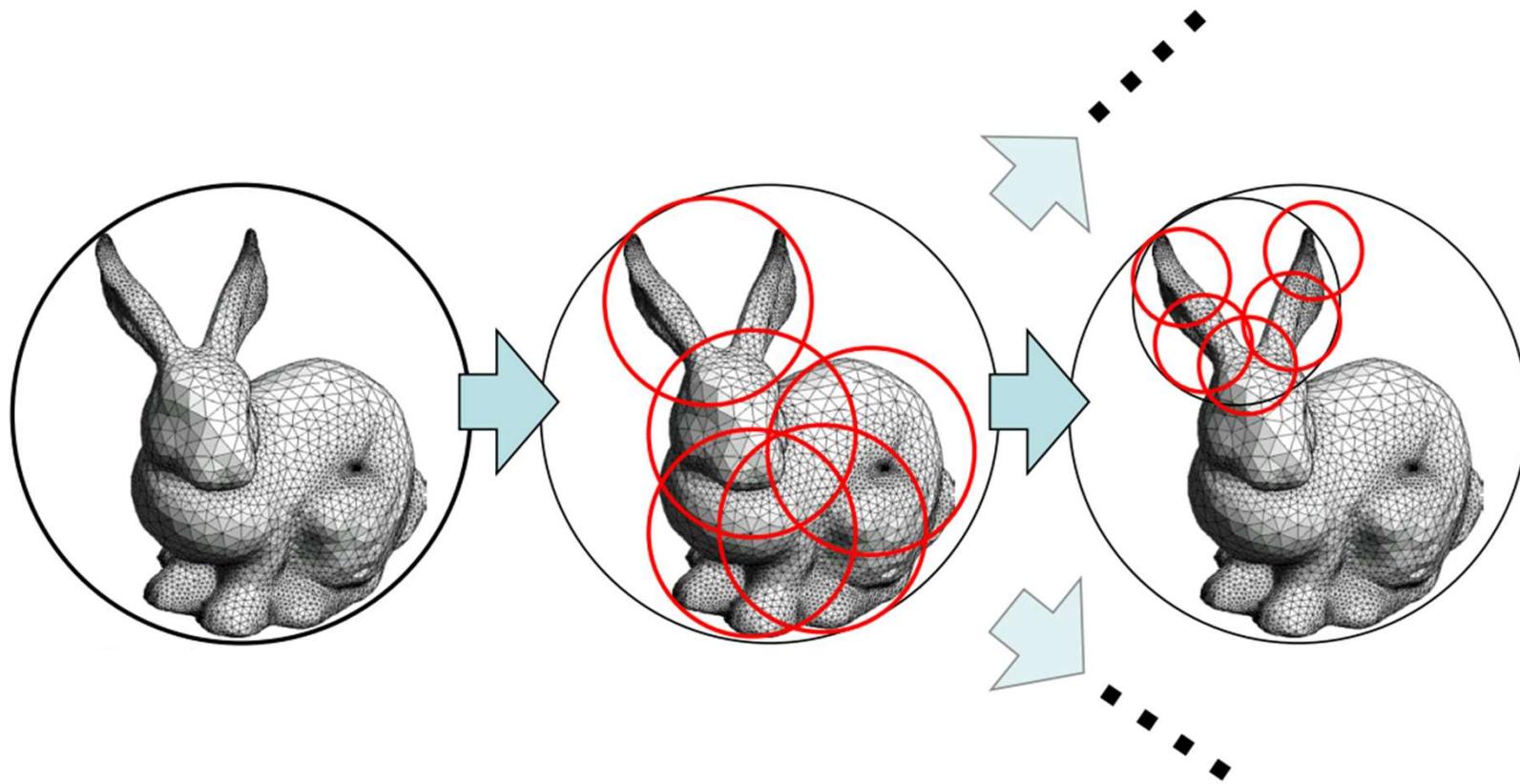
Por ejemplo, en un videojuego se puede activar la opción de **guardar la partida** al **entrar a un área específica** (un *save point*)

Esa colisión no necesita física y puede hacerse de manera básica.

<https://sbcode.net/threejs/obb/>



Modelación Jerárquica para Mayor Precisión



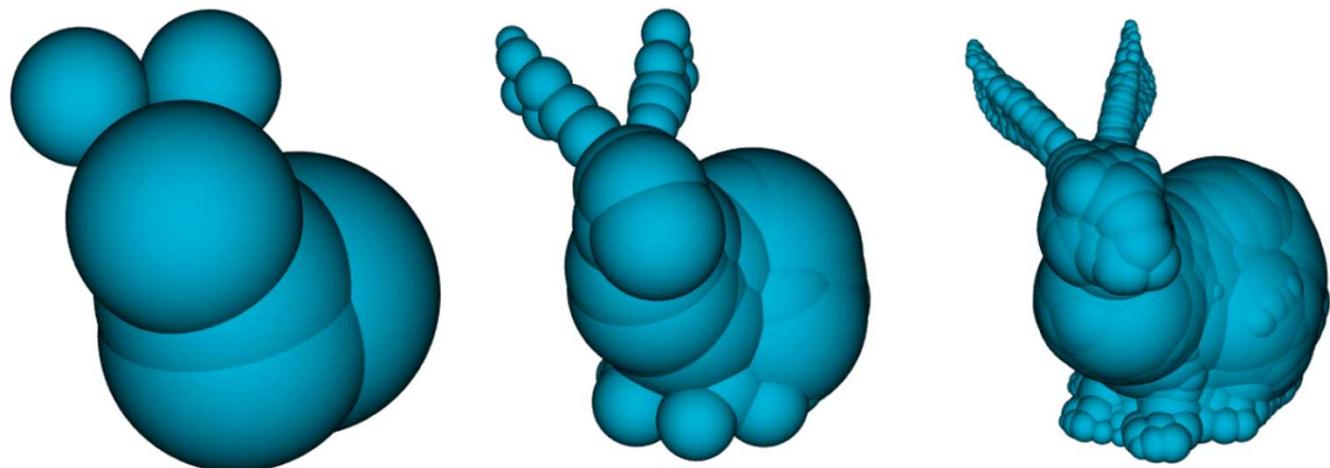
Representación Jerárquica

Se crea un árbol donde el nodo raíz es la esfera contenedora. Luego se subdivide el modelo 3D en distintas esferas (puede haber estrategias para hacer esa división).

Luego cada una de esas esferas puede volver a ser dividida.

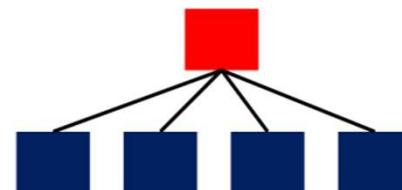
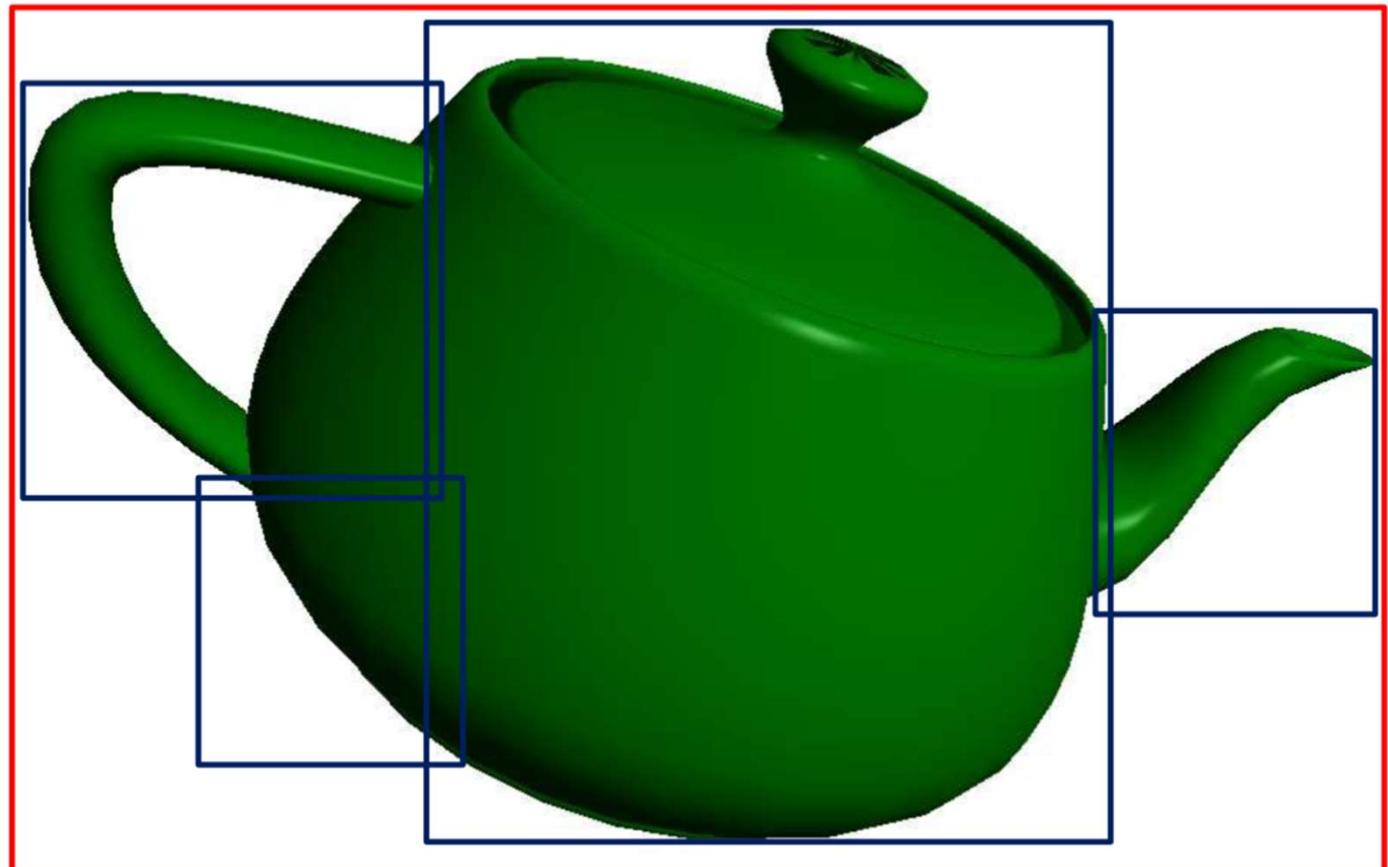
El cálculo de intersecciones se hace hasta que se han probado todos los niveles del árbol.

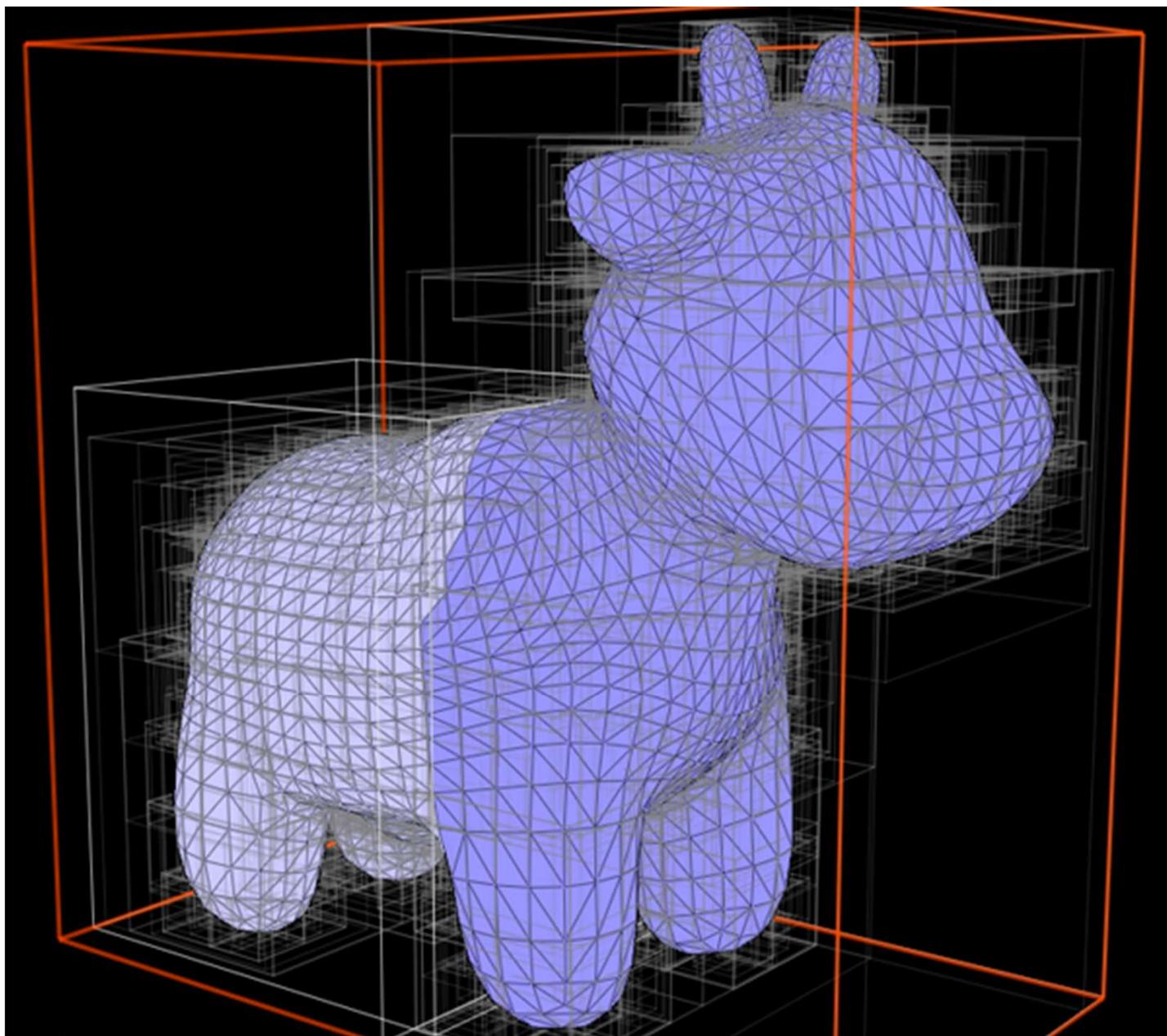
¡Se puede implementar como una función recursiva!

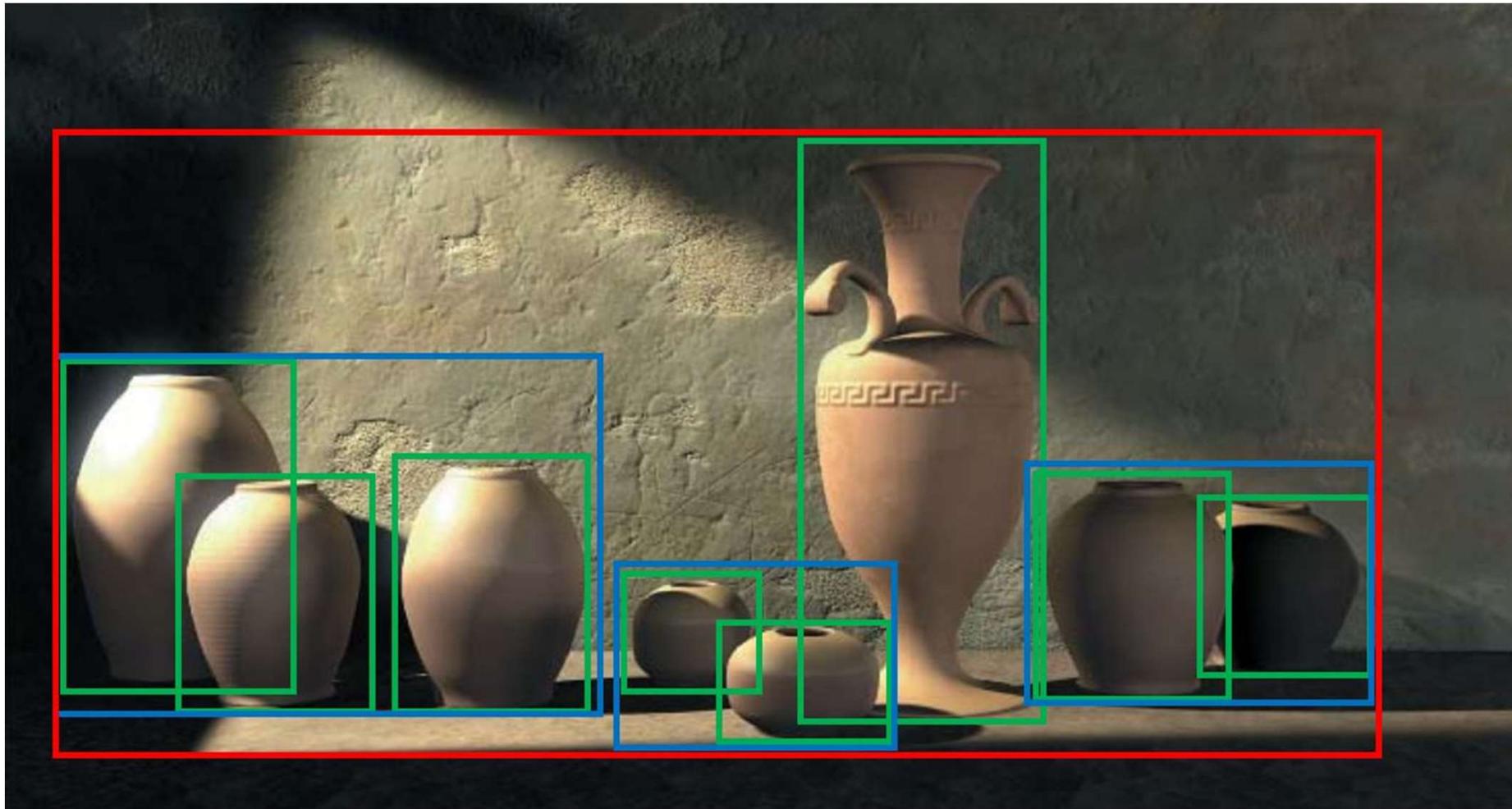


Cajas en vez de Esferas

El modelamiento jerárquico también se puede hacer por cajas.







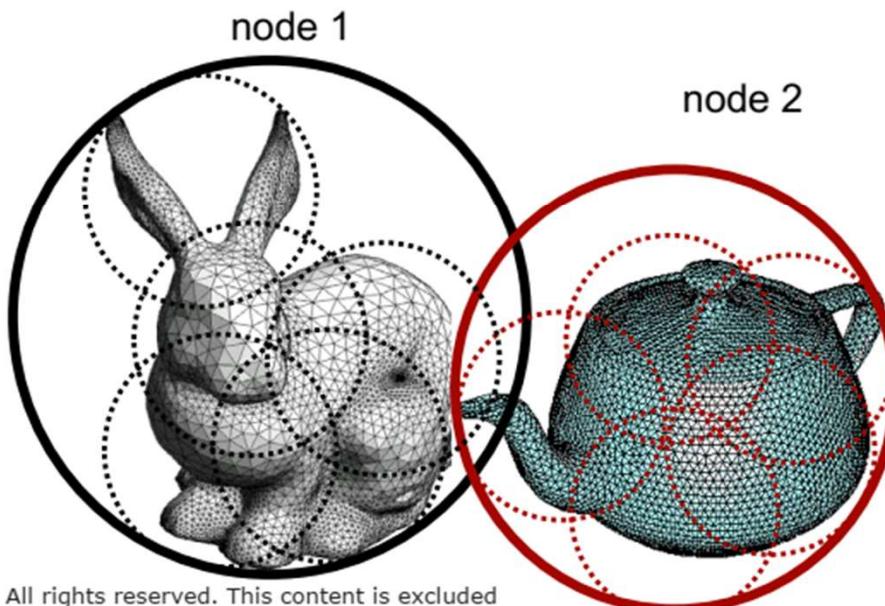
<https://jacco.ompf2.com/2022/04/13/how-to-build-a-bvh-part-1-basics/>

```
boolean intersect(node1, node2)
    // no overlap? ==> no intersection!
    if (!overlap(node1->sphere, node2->sphere))
        return false

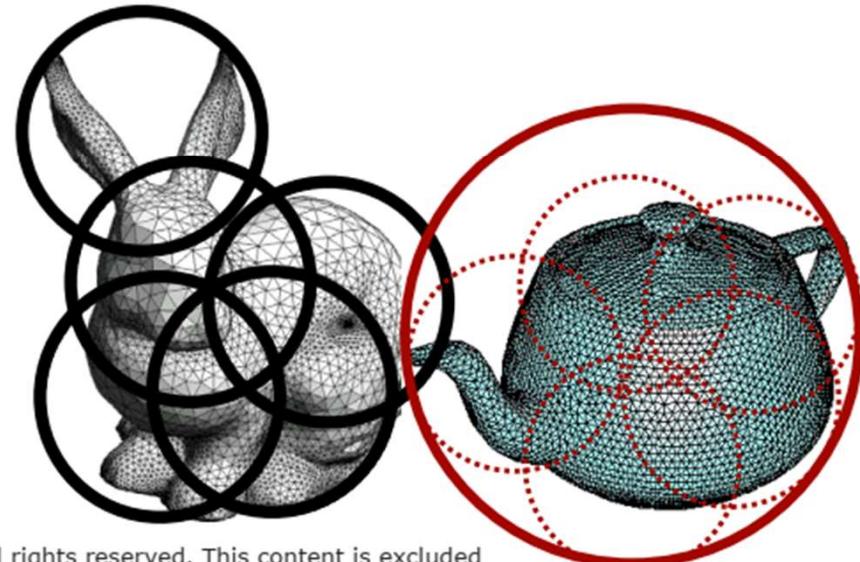
    // recurse down the larger of the two nodes
    if (node1->radius()>node2->radius())
        for each child c of node1
            if intersect(c, node2) return true
    else
        for each child c of node2
            if intersect(c, node1) return true

    // no intersection in the subtrees? ==> no intersection!
    return false
```

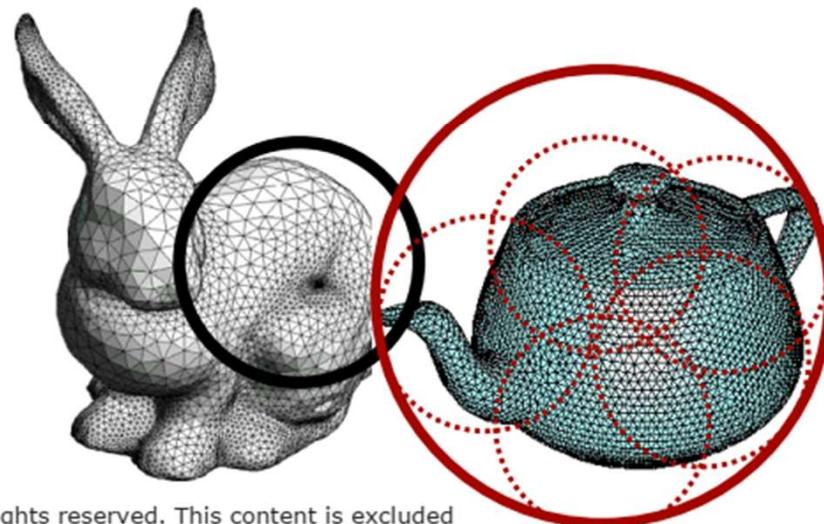
```
boolean intersect(node1, node2)
    if (!overlap(node1->sphere, node2->sphere)
        return false
    if (node1->radius() > node2->radius())
        for each child c of node1
            if intersect(c, node2) return true
    else
        for each child c of node2
            if intersect(c, node1) return true
    return false
```



```
boolean intersect(node1, node2)
    if (!overlap(node1->sphere, node2->sphere))
        return false
    if (node1->radius()>node2->radius())
        for each child c of node1
            if intersect(c, node2) return true
    else
        for each child c of node2
            if intersect(c, node1) return true
return false
```

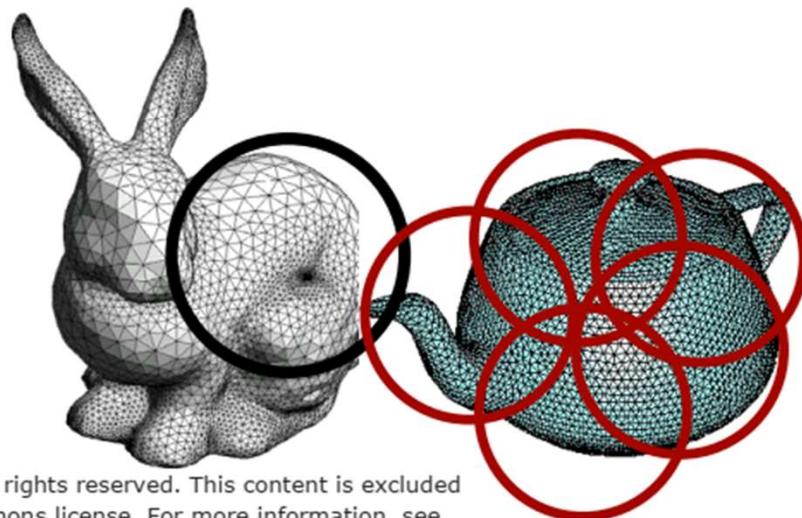


```
boolean intersect(node1, node2)
    if (!overlap(node1->sphere, node2->sphere))
        return false
    if (node1->radius() > node2->radius())
        for each child c of node1
            if intersect(c, node2) return true
    else
        for each child c of node2
            if intersect(c, node1) return true
    return false
```



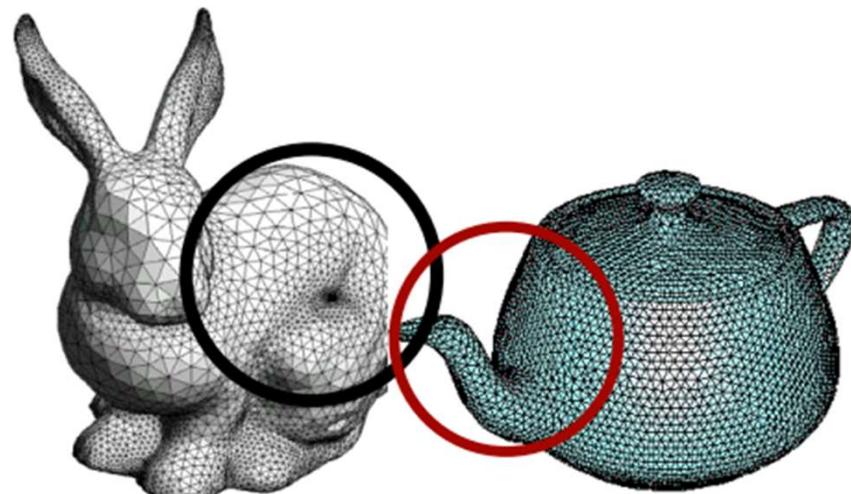
© Gareth Bradshaw. All rights reserved. This content is excluded from our Creative Commons license. For more information, see

```
boolean intersect(node1, node2)
    if (!overlap(node1->sphere, node2->sphere)
        return false
    if (node1->radius ()>node2->radius ())
        for each child c of node1
            if intersect(c, node2) return true
    else
        for each child c of node2
            if intersect(c, node1) return true
    return false
```

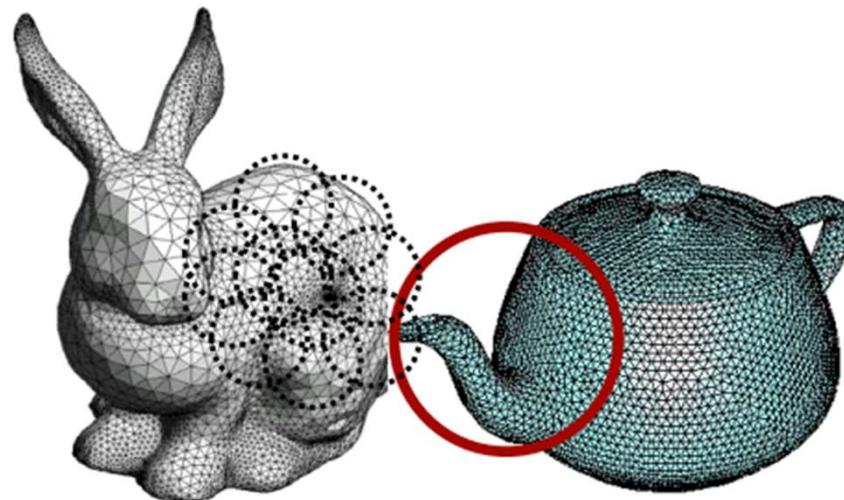


© Gareth Bradshaw. All rights reserved. This content is excluded from our Creative Commons license. For more information, see

```
boolean intersect(node1, node2)
    if (!overlap(node1->sphere, node2->sphere)
        return false
    if (node1->radius()>node2->radius())
        for each child c of node1
            if intersect(c, node2) return true
    else
        for each child c of node2
            if intersect(c, node1) return true
    return false
```



```
boolean intersect(node1, node2)
    if (!overlap(node1->sphere, node2->sphere)
        return false
    if (node1->radius()>node2->radius())
        for each child c of node1
            if intersect(c, node2) return true
    else
        for each child c of node2
            if intersect(c, node1) return true
    return false
```



```
boolean intersect(node1, node2)
    if (!overlap(node1->sphere, node2->sphere)
        return false

    // if there is nowhere to go, test everything
    if (node1->isLeaf() && node2->isLeaf())
        perform full test between all primitives within nodes

    // otherwise go down the tree in the non-leaf path
    if ( !node2->isLeaf() && !node1->isLeaf() )
        // pick the larger node to subdivide, then recurse
    else
        // recurse down the node that is not a leaf

return false
```

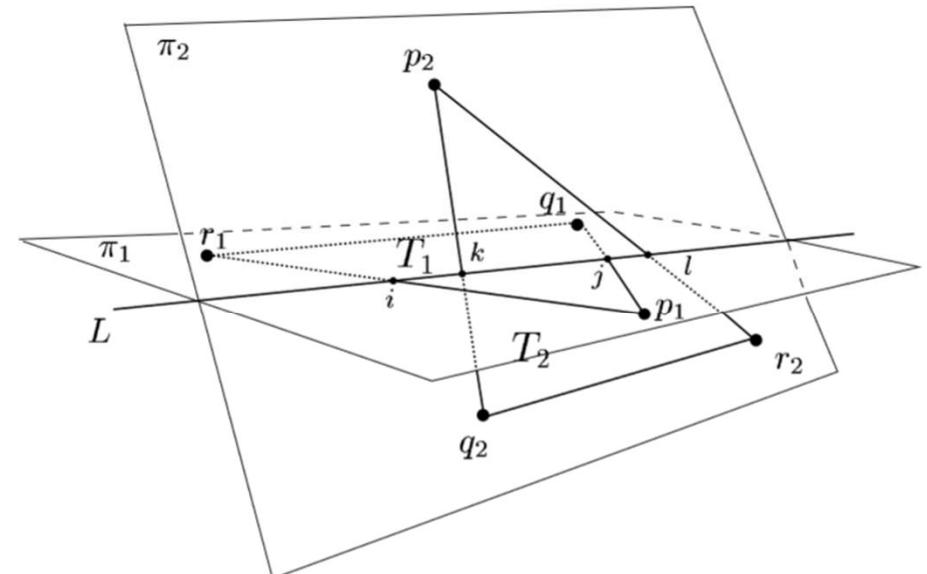
Faster Triangle-Triangle Intersection Tests

Olivier Devillers , Philippe Guigue

Thème 2 — Génie logiciel
et calcul symbolique
Projets Prisme

Rapport de recherche n° 4488 — Juin 2002 — 17 pages

[Faster Triangle-Triangle Intersection Tests](#)



¿Preguntas?

En la siguiente clase veremos lo que sigue a la colisión: motores de física y cuerpo rígido.