Modelos y Simulación para Videojuegos II

PROVISORIO

Unidad 5

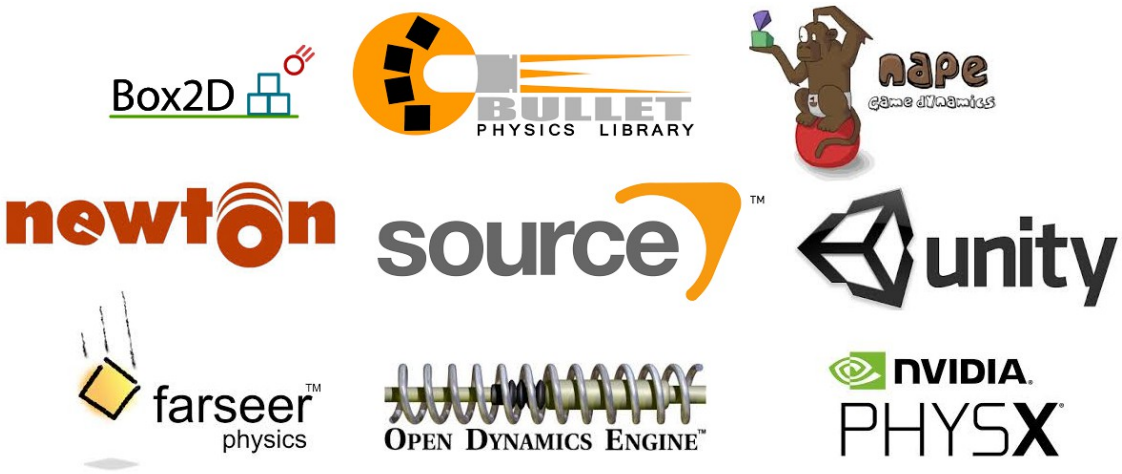
Physics Engines

Ing. Mariano Banquiero

Contenido

Introducción.

En esta unidad veremos los lineamientos básicos de un motor de física orientado a videojuegos. No intenta ajustarse a las especificaciones de ningún motor de física en concreto si no que desde el punto de vista académico se verán los lineamientos generales y los componentes básicos de todo sistema de simulación basado en física.



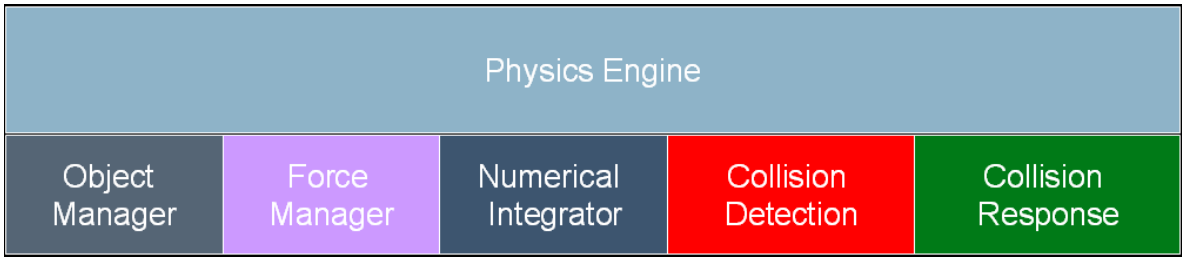
Algunos de los motores de físicas más conocidos.

Como vimos en la unidad anterior, algunos motores de física son de uso general y pueden simular cuerpos rígidos, partículas, y permiten conectar cuerpos rígidos entre si para simular objetos mas complicados como un ragdoll. Otros en cambio son mas bien específicos y están orientados a simular con mayor exactitud un sistema físico concreto, como puede ser un simulador de vuelo o el simulador de física de un videojuego de futbol.

Los principales componentes de un motor de física orientado a videojuegos son:

- Object manager.
- Force manager.
- Numerical Integrator.
- Collision Detection.
  - Broad Phase
  - Narrow Phase

- Collision Response.



Principales componentes de un motor de física genérico.

En el transcurso de esta unidad vamos a estudiar con mayor cada uno de estos módulos.

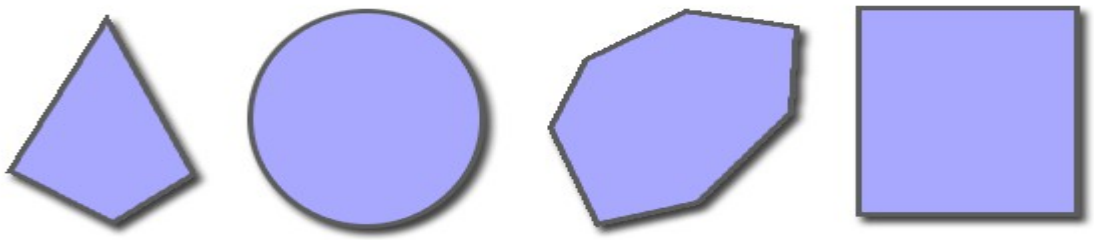
1-Object Manager.

Es el encargado de instanciar, inicializar y eliminar los distintos objetos que el motor es capaz de simular. Es también responsable de mantener las distintas conexiones o enlaces entre los objetos y las restricciones en los movimientos, en el caso que el motor soporte cuerpos articulados.

Algunos de los objetos que usualmente soportan los motores de física son:

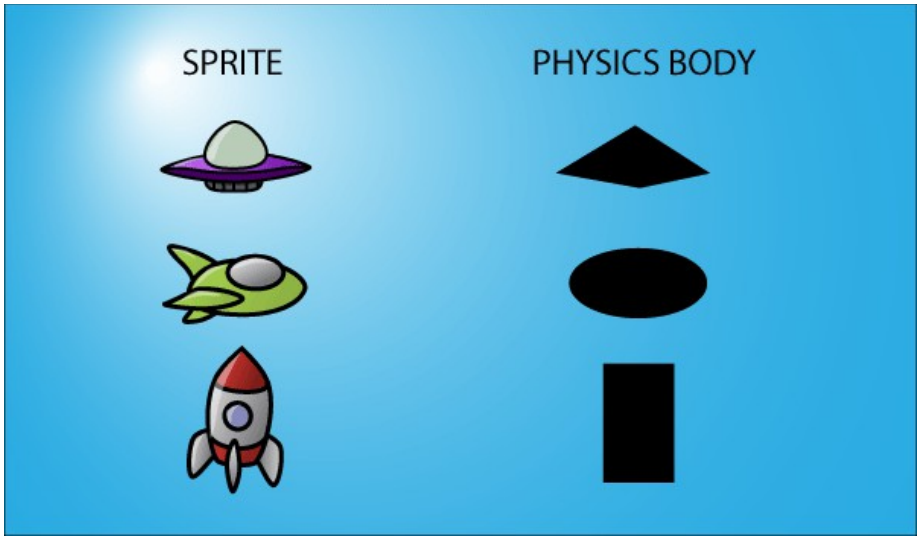
- Partículas.
- Cuerpos rígidos.
- Cuerpos rígidos articulados.

El object manager es encargado también de mantener los distintos atributos geométricos y físicos que soporta el modelo lógico. Entre los atributos geométricos esta la forma del objeto, en el caso que el objeto sea un cuerpo rígido, la forma suele ser alguna primitiva geométrica simple, como esferas, cajas, y polígonos. Muchos motores de física están limitados solo a polígonos convexos, mientras que otros soportan un mayor numero y complejidad de formas geométricas básicas y combinaciones entre ellas. Además puede que el motor soporte objetos de 2D o de 3D. Los motores que soportan 3d son muchísimo más complejos para la complejidad adicional de las rotaciones en el espacio además de la dificultad de determinar las intersecciones en 3 dimensiones.



Shapes. Las formas geométricas habituales suelen incluir cajas, esferas y polígonos.

Las propiedades cinemáticas son la posición, velocidad y aceleración. Y por último las propiedades dinámicas o mass properties que son la masa, el centro de masas y el tensor de inercia para el caso de los cuerpos rígidos. Muchas veces el centro de masas y el tensor de inercia se pueden computar algorítmicamente por el motor, o bien se pueden definir por el programador como una variable de diseño. Esto es debido a que la forma geométrica del objeto puede ser una mera aproximación de la forma real del cuerpo y puede que no tenga una distribución uniforme de masas.



La forma geométrica es una aproximación de la forma real del cuerpo.

A la colección o conjunto de objetos se los suele llamar escena o mundo (world). Algunos motores también soportan el concepto de layers o capas en la escena. Usualmente los objetos solo pueden colisionar entre si mientras pertenezcan al mismo layer.

2-Force Manager.

Es el módulo encargado de aplicar las distintas fuerzas a los objetos que componen la escena. Las fuerzas a su vez pueden ser directas o indirectas. Las directas son las que se aplican a partir de una acción del usuario. Por ejemplo el usuario puede con el cursor mover un determinado objeto. O bien pulsando una tecla puede encender la turbina de una nave y el force manager tendrá que aplicar la fuerza correspondiente al objeto o conjunto de objetos que representan la nave.

Por otra parte las fuerzas indirectas son las que se aplican a todos los objetos globalmente, como por ejemplo la gravedad. Otro ejemplo de fuerza indirecta es la acción del viento.

3- Integrador Numérico.

El integrador es el responsable de resolver las ecuaciones del movimiento para cada objeto. El motor usualmente va iterar por todos los objetos de la escena y para cada objeto computar la posición, velocidad y orientación (en caso de que se trate de cuerpos). Para realizar estos cálculos el integrador deberá poder conocer todas las fuerzas que actúan sobre cada objeto para poder determinar las fuerzas resultantes, a partir de las cuales usando la ley de Newton se podrá calcular la aceleración. La aceleración se deberá integrar en cada paso para obtener la nueva velocidad y esta última integrar nuevamente para obtener finalmente la posición que es la que se va a utilizar para poder dibujar el objeto en el lugar preciso de la pantalla.

El proceso puede parecer sencillo, pero se complica cuando hay pares de fuerzas que interactúan entre sí. Por ejemplo un par de objetos unidos por un resorte, resulta en un sistema de fuerzas que interactúan entre sí de tal forma que el cómputo de la fuerza aplicada a uno de los objetos, modifica la fuerza aplicada al otro objeto y viceversa.

Si partimos de la segunda ley de Newton:

$$\vec{F} = m\vec{a}$$

si tenemos en cuenta que la aceleración representa la tasa de cambio de la velocidad con respecto al tiempo, es decir la derivada de la velocidad con respecto al tiempo, podemos re-escribir la ecuación como :

$$\vec{F} = m\frac{dv}{dt}$$

que a su vez se puede escribir así, despejando el término dv :

$$dv = \frac{\vec{F}}{m}dt \Rightarrow \Delta v = \frac{\vec{F}}{m}\Delta t \qquad (1)$$

En esta última expresión se reemplazo los diferenciales "d" por los "delta", teniendo en cuenta que para intervalos pequeños de tiempo, es cierto que :

$$\frac{\Delta v}{\Delta t} \sim \frac{dv}{dt}$$

La expresión (1) es de gran importancia práctica, pues nos brinda una forma de calcular el incremento de le velocidad a partir de un incremento de tiempo. Lo

que la ecuación (1) nos dice es que un pequeño incremento en el tiempo del valor de Δt va a producir un pequeño incremento en la velocidad llamado Δv y nos da una fórmula precisa para calcularlo que es multiplicar el valor de Δt por el valor de la fuerza neta (dato que nos va a proveer el force manager) y dividirlo por la masa, (dato que nos va a proveer el object manager).

En la simulación en cada paso del game loop se avanza una cierta cantidad de tiempo llamada elapsed_time que es igual a Δt y con este valor es posible computar Δv. A este proceso se lo llama integración numérica.

La velocidad en un instante dado se computa sumando el cambio de velocidad Δv al valor que tenía anterior, es decir :

$$\vec{v} \leftarrow \vec{v} + \frac{\vec{F}}{m}\Delta t$$

Ahora tenemos que repetir este mismo proceso para obtener la posición, es decir la primera vez que integramos obtenemos la velocidad y la segunda vez la posición:

$$\vec{s} \leftarrow \vec{s} + \vec{v}\Delta t$$

Es importante notar que estas fórmulas son solo aproximaciones, ya que el *dv* es una aproximación de Δv. Cuanto más pequeño es el intervalo de tiempo Δt, más precisa va a ser la aproximación. Sin embargo el Δt usualmente esta fijo en un valor que depende de los fps, por ejemplo a 30 fps el valor de Δt es de 30/1000 segundos. Durante ese lapso de tiempo la velocidad puede estar variando, pero la ecuación la toma como fija y a partir de ahí hay un determinado error que se puede ir propagando con el tiempo. Afortunadamente hay varios métodos numéricos para atacar este problema y permitir resolver el problema de la integración con muchísima precisión, mucha más de la que usualmente se precisa en el ámbito de un video juego.

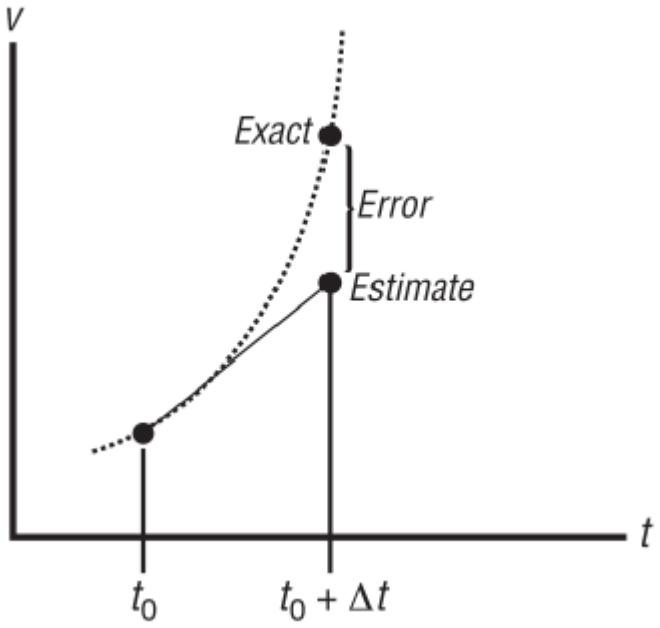
Método de Euler.

El método de integración de Euler es el más simple que se puede aplicar, y de hecho muchos motores de física sencillos utilizan este método como integrador.

```
// Explicit Euler
x += v * dt;
v += (1/m * F) * dt
```

En la primer línea actualizamos la posición multiplicando la velocidad actual por la variación del tiempo, es decir integramos la velocidad. En la segunda línea hacemos lo mismo con la velocidad, integramos la aceleración.

El método de Euler es sumamente sencillo de implementar como podemos ver, sin embargo como veremos a continuación introduce mucho error, especialmente cuanto la velocidad cambia de manera no lineal entre step y step.



El método de Euler da una aproximación lineal. La diferencia con el real es el error en la estimación.

Para comprobarlo podemos hacer un ejemplo con el movimiento de un objeto que cae por acción de la gravedad. Dicho cuerpo se mueve en caída libre y podemos usar las fórmulas de la cinemática para comparar el valor obtenido por integración de Euler contra el valor real. Recordemos que las fórmulas de la cinemática son exactas y nos dan una descripción del movimiento de los cuerpos, al menos en las condiciones ideales de aceleración constante, ausencia de fricción, etc.

La ecuación de la cinemática nos dice que en cada instante la posición x vendrá dada por esa expresión :

$$x = \frac{1}{2}gt^2$$

En nuestro caso vamos a suponer que el cuerpo empieza en x=0, la gravedad es de 10 m/s2 y que a medida que cae x va aumentando. La posición inicial es cero al igual que la velocidad inicial. Vamos a suponer que el elapsed_time es de un segundo, es decir dt = 1.

En el primer paso, cuando ha transcurrido un segundo de animación, la ecuación de la cinemática nos dice que la posición exacta del cuerpo cayendo será de :

$$x1 = 0.5 * 10 * 0^2 = 0$$

Por otra parte, al aplicar la integración de Euler,

$$x1 = x0 + v0 = 0 + 0 = 0$$
$$v1 = v0 + a \cdot dt = 0 + 10 * 1 = 10$$

Ahora en el segundo paso, la ecuación cinemática:

$$x2 = 0.5 * 10 * 1^2 = 5$$

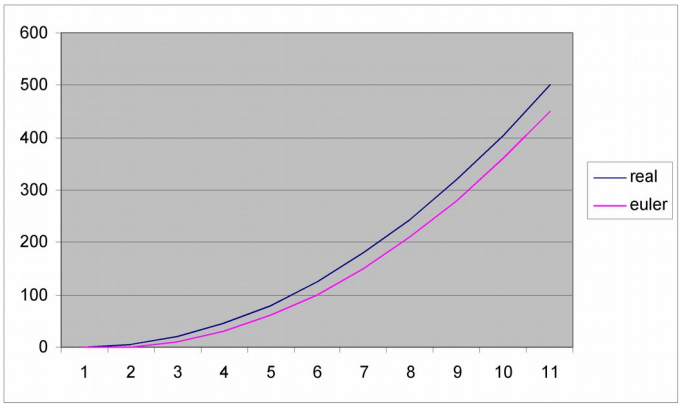
Y al aplicar Euler,

$$x2 = x1 + v1 = 0 + 10 = 10$$

Ya vemos que tenemos una diferencia entre la posición computada por la integración de Euler y el valor real que es el de la ecuación de caída libre.

Podemos hacer una tabla con los distintos valores, computados según la integración de Euler y el valor real:

Tiempo	X Exacto	EULER V	EULER X
0	0	0	0
1	5	10	0
2	20	20	10
3	45	30	30
4	80	40	60
5	125	50	100
6	180	60	150
7	245	70	210
8	320	80	280
9	405	90	360
10	500	100	450



Sympletic Euler.

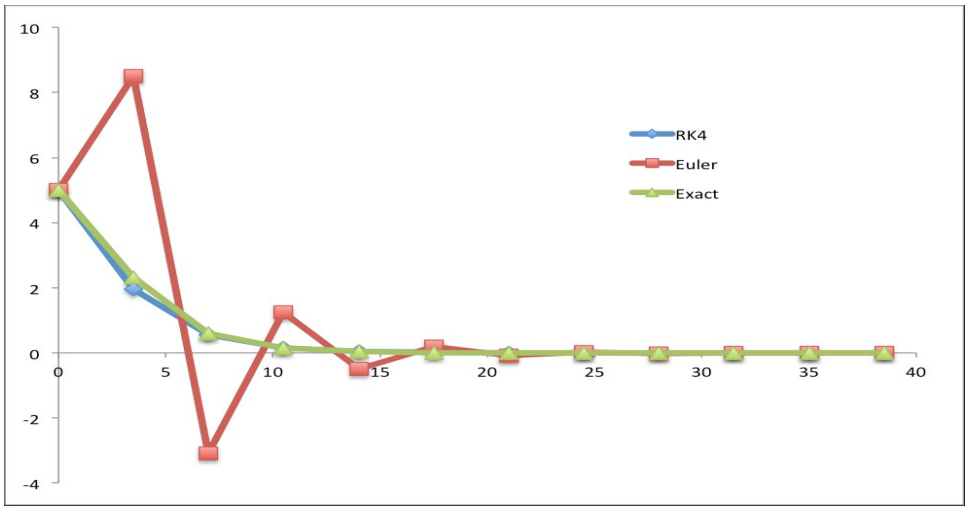
Una forma de mejorar el método de Euler simple es el llamado Euler sympletic o semi-implícito, que consiste en el siguiente algoritmo:

```
// Sympletic Euler
v += (1/m * F) * dt
x += v * dt;
```

Tenemos que notar que lo único que hicimos fue cambiar de orden las líneas. Primero integramos la aceleración para computar el nuevo valor de la velocidad. Y ahora integramos la posición usando el valor nuevo de la velocidad. El error de este método es similar al anterior, la diferencia es que se esta forma de integrar preserva la energía total del sistema por lo que se comporta más estable. El análisis de la estabilidad y conservación de la energía escapa del propósito de este apunte.

Mejores Métodos de Integración.

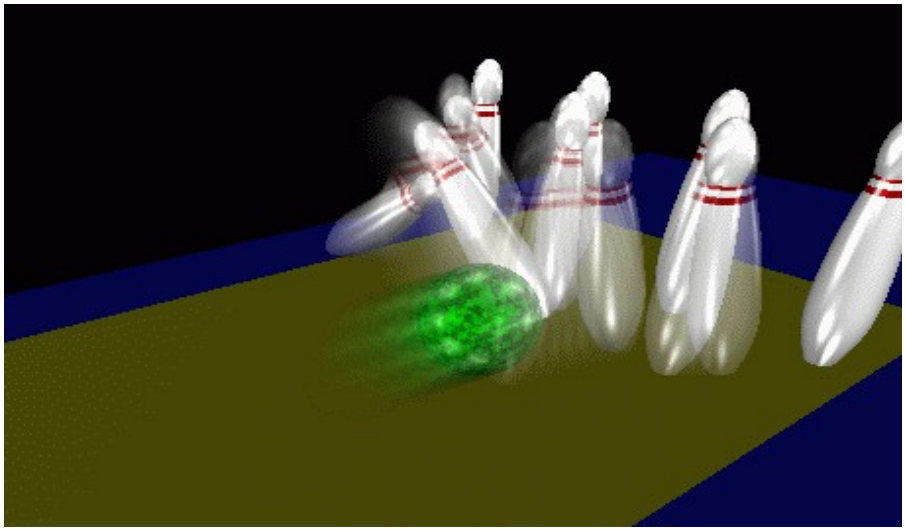
El método de Euler solo toma datos del paso anterior para extrapolar los resultados al paso siguiente. Existen otros métodos de integración que se basan en tomar información de pasos anteriores adicionales, de tal forma de lograr una mejor aproximación. Unos de los más utilizados el método de Runge - Kutta, que en su versión más utilizada usa los valores de 4 paso anteriores logrando una excelente aproximación.



Método de Runge - Kutta RK4 comparción con Euler y el valor exacto.

4- Detección de Colisiones.

El movimiento de los objetos en el escenario hace que estos sean susceptibles de chocar entre sí. Para que la simulación sea realista tenemos que ser capaces de detectar la colisión para poder evitarla o eventualmente corregir la trayectoria de los cuerpos luego de producirse el impacto. La detección de colisiones es una parte fundamental en el motor de física.

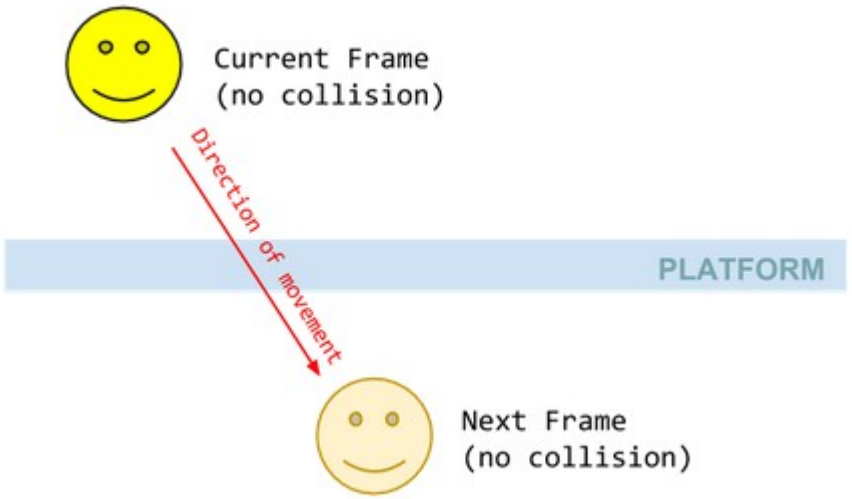


La detección de colisiones juega un rol fundamental en el realismo de las simulaciones.

La detección de colisiones es un problema de computación geométrica, se trata de determinar si dos objetos colisionan entre sí y si es ese el caso, determinar que punto o que puntos exactos son los que entran en contacto. Usualmente a esta colección de puntos se los llama contact manifold. Además de la posición de los puntos que entran en contacto, suele ser necesario la velocidad de los mismos.

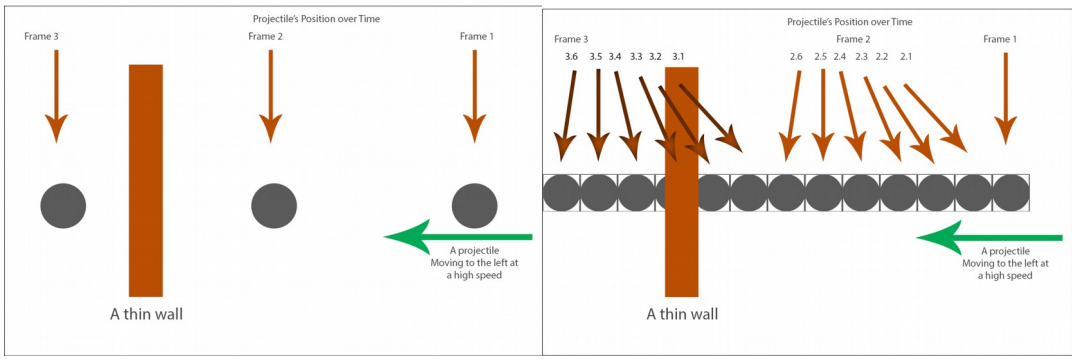
Nota : En matemáticas manifold (variedad) significa un espacio abstracto geométrico, como una curva, pero no tiene nada que ver con el significado que se le da en los motores de físicas. En concepto de contact manifold es como si fuese un conjunto de contactos.

El problema es mucho más complejo de lo que parece a simple vista, inclusive con cuerpos simples como esferas y cajas, la cantidad de chequeos que hay que hacer para resolver el problema por fuerza bruta es enorme. Además puede haber objetos moviéndose muy rápido y atravesando cientos de otros objetos por completo (especialmente los muy delgados) en el lapso de un solo paso de la simulación. Esto significa que un sistema robusto de detección de colisiones no puede basarse solamente en la posición actual de los objetos, ya que de un frame a otro un objeto suficientemente rápido puede atravesar por completo uno suficientemente delgado.



Detección solo basada en la posición: se pierde colisiones.

Tampoco se puede basar solo en la separación entre los objetos y la velocidad relativa, ej: si los objetos se encuentran a cierta distancia y acercándose. Por otra parte los cuerpos extensos, si son cóncavos, pueden colisionar entre si en varios puntos al mismo tiempo. Además pueden haber mas de 2 cuerpos colisionando simultáneamente. Un sistema robusto de detección de colisiones tiene que ser capaz de lidiar con estos problemas eficientemente. Se pueden atacar varias de estas cuestiones puntualmente, por ejemplo determinar si un cierto objeto atraviesa una pared o un piso, como casos particulares en un sistema simple de detección de colisiones. Sin embargo una solución general requiere determinar con precisión si un objeto va a colisionar en un futuro cercano contra cualquier otro basándose las velocidades relativas de todos los objetos en su entorno. Esto es un problema geométrico sumamente complejo y los motores de físicas que soportan este tipo de algoritmos se llaman "continuous" collision detection (CCD), indicando que la detección de colisiones se hace en forma continua, a diferencia de las implementaciones simples (que son las que vamos a ver) que solo lo hacen en forma discreta en cada paso de la simulación.



CCD en Unity3d. La figura de la izquierda muestra como un proyectil que se mueve muy rápido atraviesa un pared delgada. En la fig. derecha se muestra como el método de continuos collisson detection permite resolver el problema.

Métodos de detección de colisiones.

La base para todo módulo de detección de colisiones son los algoritmos de colisión geométricos entre formas básicas, siendo que la mayor parte de los objetos que modele el motor de física serán formas básicas o combinaciones entre ellas. Así surgen los diferentes métodos para obtener la colisión entre figuras como círculos, cajas, polígonos entre sí. Como vimos anteriormente el algoritmo no solo tiene que ser capaz de determinar si dos objetos colisionan, si no también cuales son los puntos que entran en contacto, y a que velocidad.

Performance

Cada algoritmo de colisión requiere efectuar cálculos matemáticos, principalmente cálculos geométricos sobre inclusiones de puntos, rectas y polígonas. Para realizar estos cálculos es necesario utilizar distintas instrucciones de la CPU. A continuación se listan en forma genérica algunas de las más comunes:

Costo computacional	Instrucciones
Bajo	Sumar, Multiplicar, Potencia
Medio	División, Logaritmo
Alto	Raíz cuadrada, operaciones trigonométricas: coseno, seno, tangente, arcoseno, etc.

El costo de una operación depende de muchos factores, como la arquitectura del procesador, el resto de los componentes de la PC que se utiliza para medir, la carga de trabajo, etc. Pero en líneas generales una instrucción de raíz cuadrada siempre será más costosa que una de multiplicación.

Es por esto que muchas veces la forma matemática mas sencilla o correcta de realizar un testeo de colisión puede no ser la indicada a la hora de implementarla en una aplicación en tiempo real. A veces es preferible utilizar un algoritmo de colisión que solo utilice sumas y multiplicaciones en vez de uno que efectúa muchos cálculos trigonométricos.

Las distintas estrategias de colisión que se explicarán en este apunte estarán orientadas a alcanzar la máxima performance posible. Veremos a continuación algunos de los algoritmos de colisión más comunes.

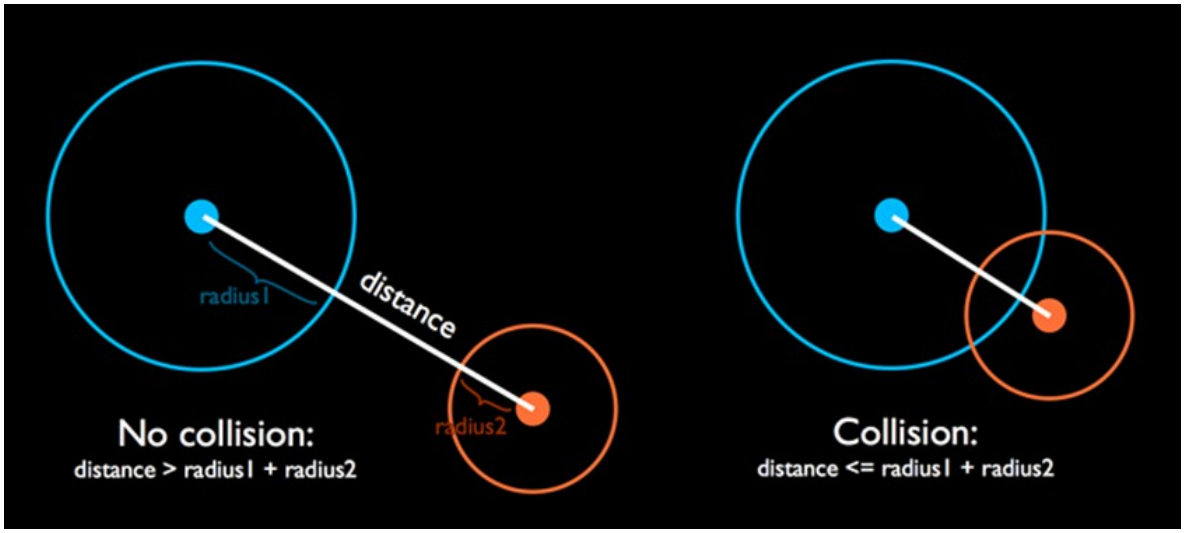
Colisión Círculo - Círculo.

La colisión más simple que se puede hacer es entre 2 círculos. Para que 2 círculos entren en colisión la distancia entre sus centros tiene que ser menor a la suma de los radios, es decir :

$$dist < radio1 + radio2 \Rightarrow \|p1 - p2\| < radio1 + radio2$$

Calcular la distancia requiere efectuar una raíz cuadrada, y eso en términos de costo computacional es muy costoso. Conviene modificar la ecuación para evitar el calculo de la raíz, elevando ambos miembros al cuadrado obtenemos esta fórmula equivalente pero que no precisa mas que multiplicaciones y sumas:

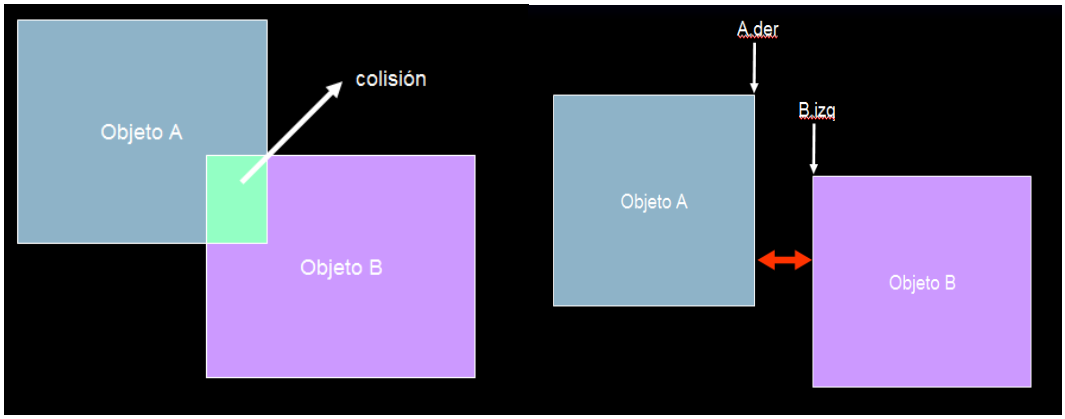
$$(p1_x - p2_x)^2 + (p1_y - p2_y)^2 < r_1^2 + 2 \cdot r_1 \cdot r_2 + r_2^2$$



Colisión entre círculos.

Colisión entre Cajas.

La colisión entre cajas también es sencilla cuando estas están alineadas a los ejes, es decir cuando no están rotadas. En ese caso se puede verificar comparando por separados los ejes X e Y, para determinar si hay un solape (indicando en verde en la figura) entre ambas figuras.



Colisión entre cajas alineadas a los ejes.

Puede resultar más intuitivo realizar el test inverso, es decir verificar si no colisionan, para ello podemos ir descartando por eje. Por ejemplo, si la posición derecha del objeto A es menor a la posición izquierda del objeto B indica que hay una separación (o GAP) entre ambos y por ende no pueden estar colisionando:

```
if(A.der<B.izq)
   return NO_COLISION;
```

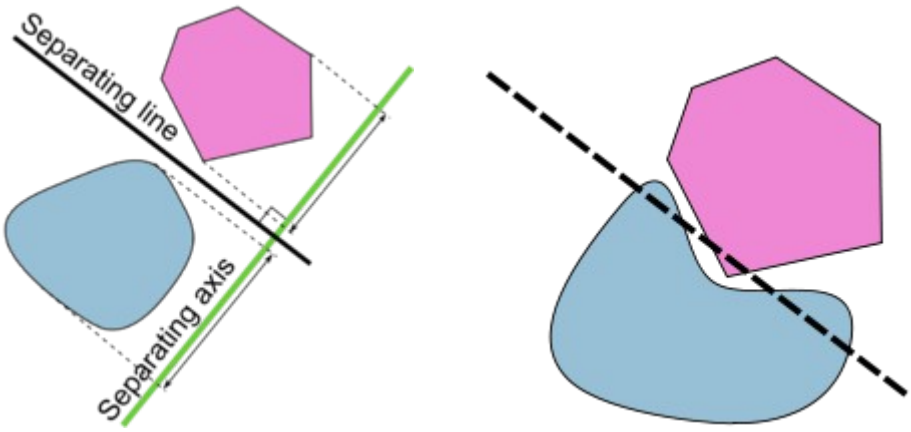
El caso contrario (que la posición derecha del objeto A sea mayor o igual a la posición izquierda del objeto B) no indica colisión. Lo que hay que hacer es descartar todo el resto de las posibilidades. El código completo para una función que detecte la colisión entre 2 rectángulos podía ser:

```
boolean collideRectRect(A, B)
{
   if(A.der < B.izq || A.izq > B.der || A.arr < B.aba || A.aba > B.arr)
      return false;
   else
      return true;
}
```

Colisión entre polígonos. Algoritmo SAT.

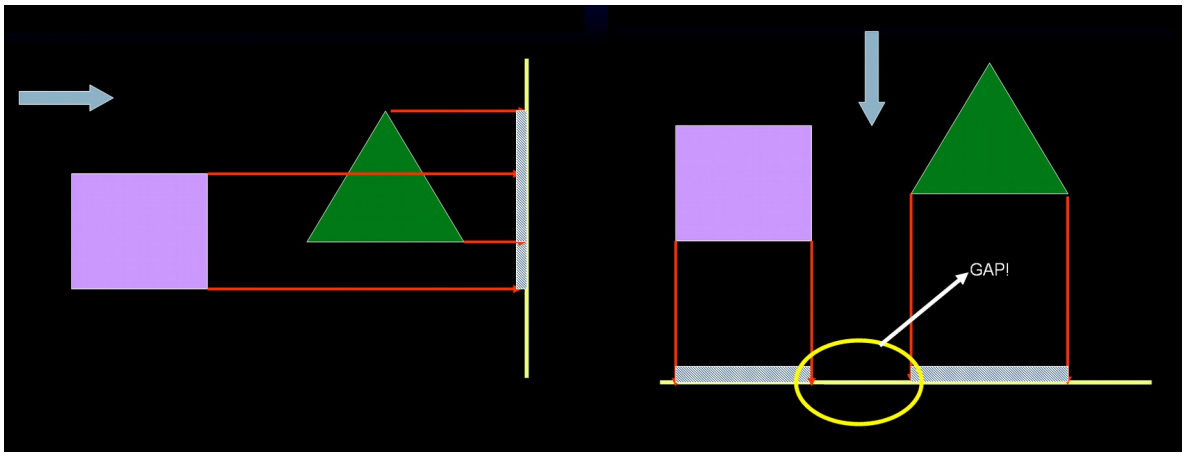
El algoritmo anterior solo funciona para rectángulos que están alineados a los ejes, si alguno de los rectángulos estuviese en cierta orientación ya no se podría aplicar. Sin embargo, se puede aplicar el mismo concepto de buscar si existe una separación entre ambos cuerpos que implique no se estén tocando y por ende descartar la colisión.

El algoritmo esta basado en un resultado matemático llamado Seperation Axis Teorem (SAT) que dice 2 cuerpos convexos están separados (y por ende no hay colisión) si y solo si existe en una línea (o un plano en 3d) que los separa. A dicha línea se la llama línea de separación.



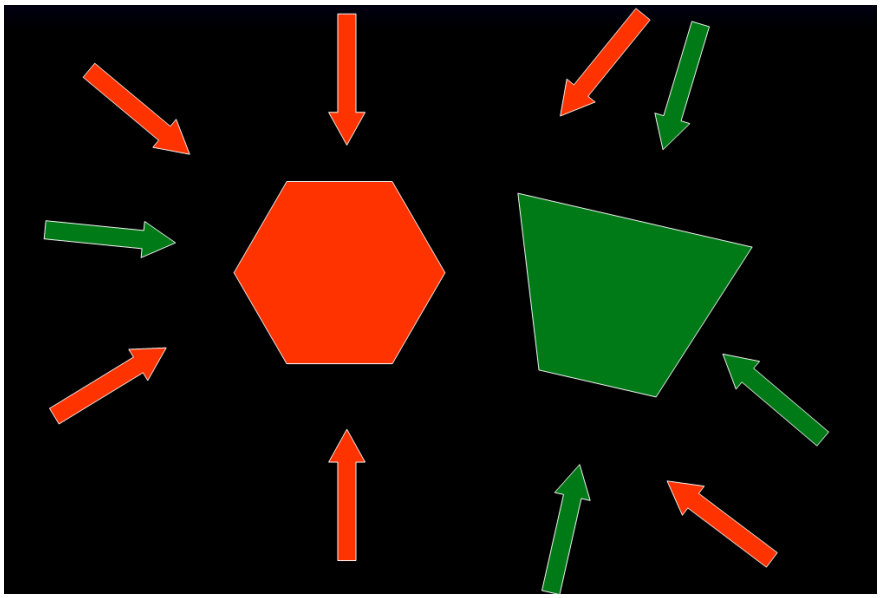
SAT. Dos convexos que no se tocan sii tiene una línea de separación. A la derecha se ve que el resultado no se cumple si los cuerpos son cóncavos.

Haciendo uso de este resultado la idea es ir chequeando la existencia o no de esta línea de separación. Para ello hay que notar que si uno proyecta ambos cuerpos en la dirección de la línea de separación, sobre un eje imaginario alejado de ambos cuerpos, (dibujado en verde en la figura) se puede ver que si existe separación entre los convexos tiene que aparecer una separación entre las proyecciones de los mismos.



Gap entre las proyecciones cuando se proyecta sobre el eje de separación.

Podemos hacer una analogía visual imaginando que alumbramos los cuerpos con una linterna y observamos su sombra sobre una pared de fondo. Si notamos que hay una separación en las sombras proyectadas es porque los cuerpos no se están tocando. El recíproco no es cierto y si notamos que las sombras se superponen no quiere decir que los cuerpos se estén tocando. En ese caso y de manera similar al algoritmo del rectángulo, tenemos que seguir probando todos los ejes posibles. Afortunadamente el teorema también indica que la línea de separación tiene que ser perpendicular a uno de los lados de uno de los polígonos que intervienen. Es decir que solo tenemos que probar un eje por cada lado de cada polígono (teniendo en cuenta que puede haber direcciones repetidas), la máxima cantidad de lados a probar es la suma de los lados de los polígonos.



La cantidad de direcciones a testear es menor o igual a la suma de los lados de los polígonos.

Colisiones en 3d. Colisión a nivel de Triángulos.

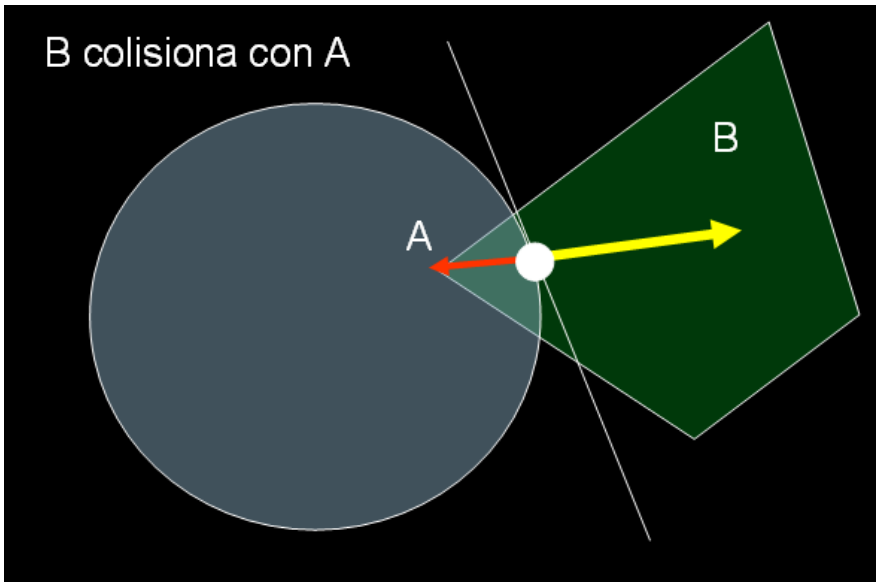
Teniendo en cuenta que el universo 3d está compuesto por una sopa de triángulos, la forma más precisa de realizar la detección 3D es realizar chequeos a nivel de cada triángulo de cada modelo del escenario. Sin embargo los algoritmos de chequeo de intersección entre triángulos en 3d son ciertamente costosos a nivel computacional, y además las gran cantidad de triángulos que usualmente componen un modulo 3d hace que este tipo de chequeo de colisión sea impracticable.

Elementos de una colisión.

Una vez que detectamos que 2 cuerpos entran en colisión es preciso determinar que puntos exactos están en contacto, que velocidades relativas tienen, entre otros datos que luego van a ser de utilidad en la siguiente etapa de collision response.

Los elementos más importantes de una colisión son:

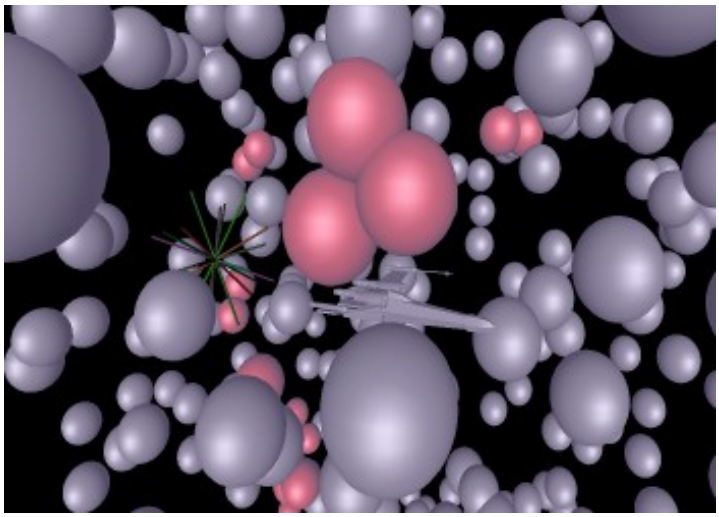
- el punto de colisión o punto de contacto.
- la penetración, que indica cuanto entra un cuerpo en otro.
- la normal en el punto de contacto.



Elementos de una colisión: punto de contacto, normal y penetración.

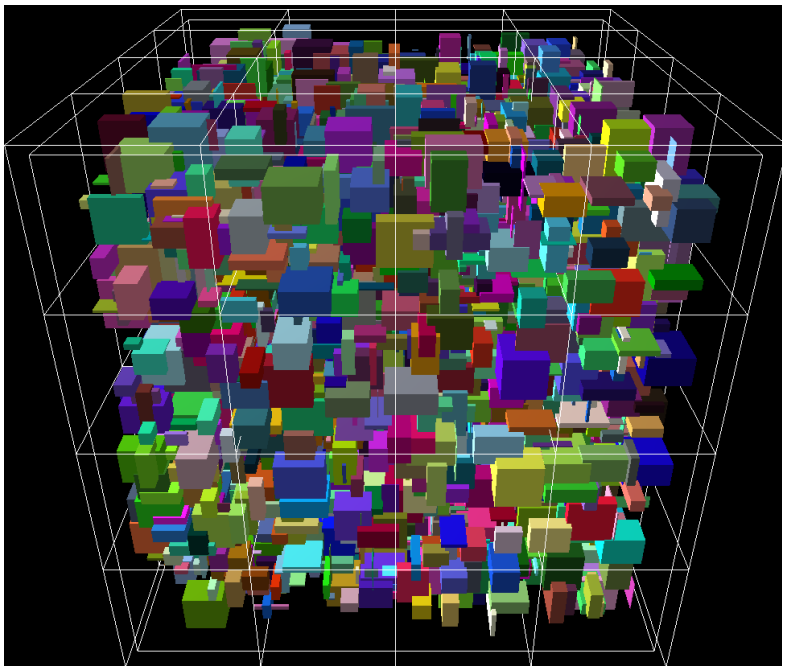
Broad Phase.

En principio cada cuerpo que se está moviendo puede colisionar con cualquier otro cuerpo en la escena (ya sea que se este moviendo también o que esté estático). Eso implica que la cantidad de colisiones a testear sería del orden de n x (n-1) /2, siendo n la cantidad de objetos en la escena. La fórmula se deduce del hecho que por cada objeto tendríamos que chequear las colisiones con el resto de los objetos (no hace falta chequear colisión consigo mismo) y la división por 2 surge del hecho que las colisiones son conmutativas, si el objeto A colisiona con B, el B colisiona con A y no hace falta chequearlo otra vez. Esta fórmula cuadrática indica que la cantidad de testeos a realizar crece mucho más rápido que la cantidad de objetos en la escena con lo cual el problema de detección de colisiones se puede tornar demasiado complejo si la escena es lo suficientemente grande.



La detección de colisiones puede resultar muy compleja si la cantidad de objetos en la escena es grande.

Para atacar este problema se suele dividir la detección de colisiones en 2 sub-estapas o fases, una llamado broad phase o fase aproximada, donde a groso modo se determina que pares de objetos pueden llegar a colisionar entre si, y una fase mas precisa, llamada narrow phase donde a partir de los candidatos surgidos de la primer fase, se determina precisamente si colisionan o no y en tal caso los elementos de la colisión.



Broad Phase. busca que partes de objetos pueden colisionar.

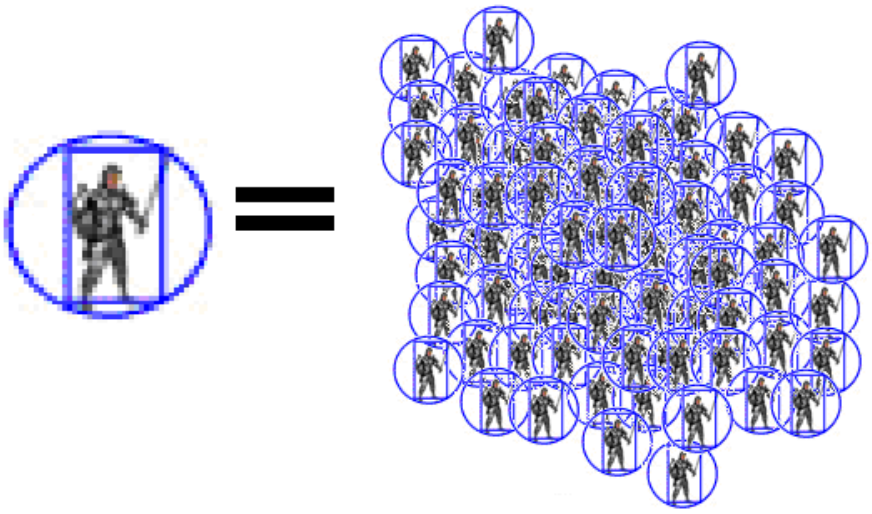
Broad Phase entonces es una primera aproximación. La idea es descartar rápidamente pares de objetos que se encuentran lo suficientemente alejados entre

si como para que no se sea posible que estén en contacto por más rápido que se muevan. Para resolver esta fase se suelen usar los llamados algoritmos de subdivisión del espacio. Todos estos algoritmos se basan en subdividir el espacio de la escena en un conjunto de celdas (usualmente disjuntas entre si), de tal forma que los objetos que pertenecen a una celda se sepa que no pueden colisionar con los que están en una celda diferente.

De esta forma una vez construida esta subdivisión en celdas, la cantidad de testeos que hay que chequear es mucho menor, pues solo se testean entre si objetos que están dentro de la misma celda. Lo que cambia en cada uno de los distintos algoritmos que vamos a ver, es la forma precisa de dividir el espacio de la escena en un conjunto de celdas.

Los algoritmos de subdivisión son usados frecuentemente en videojuegos para optimizar la performance de numerosas etapas del mismo. Por ejemplo son fundamentales para la etapa de render, al permitir identificar rápidamente que modelos son potencialmente visibles desde el punto de vista de la cámara. También se usan en IA para determinar rápidamente caminos entre dos partes del escenario. En síntesis son estructuras de datos que permiten manejarse con la escena de una forma mucho más organizada. Haciendo una analogía con un sistema administrativo, estas estructuras son como los índices de la base de datos. En un videojuego la "base de datos" es el escenario, los "datos" son los distintos modelos, y estas estructuras son como los índices que nos permiten acceder con mayor performance.

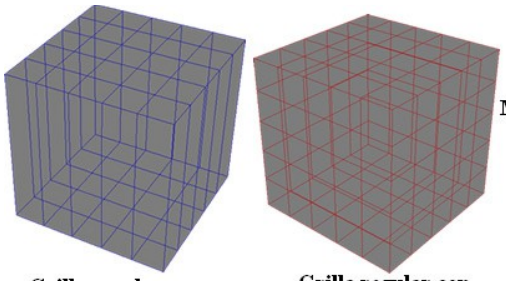
Por lo dicho anteriormente el enfoque para introducir estos conceptos va a ser general y no específico de la etapa de colisiones. Hay que tener en cuenta que luego el uso de estas estructuras será diferente según el caso, si es que se va a usar para acelerar un chequeo de intersecciones o se va a usar para determinar si un modelo es visible. También para facilitar la compresión de los conceptos se usan ejemplos simples con pocos modelos, para que sea más realista la situación puede resultar útil considerar que un modelo en verdad puede ser un conjunto de cientos de modelos y que para resumir la explicación se usa un solo modelo.



Cada modelo es en la práctica un conjunto de cientos de modelos.

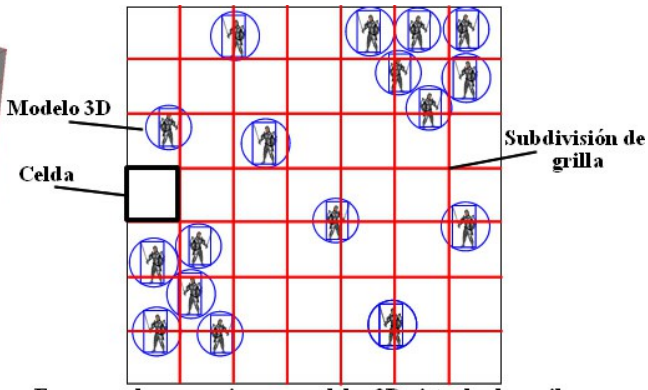
Grilla regular

La forma más simple de dividir el escenario es utilizando todas celdas del mismo tamaño. Este método se llama grilla regular y consiste en dividir el escenario en una grilla, tabla o matriz compuesta por celdas de tamaño fijo que delimitan sectores imaginarios en todo el escenario. La grilla puede ser de dos dimensiones sobre el nivel del piso (X, Z) o de tres dimensiones, subdividiendo también el espacio en el eje Y.



Grilla regular con subdiviciones en eje X y Z

Grilla regular con subdiviciones en los tres ejes

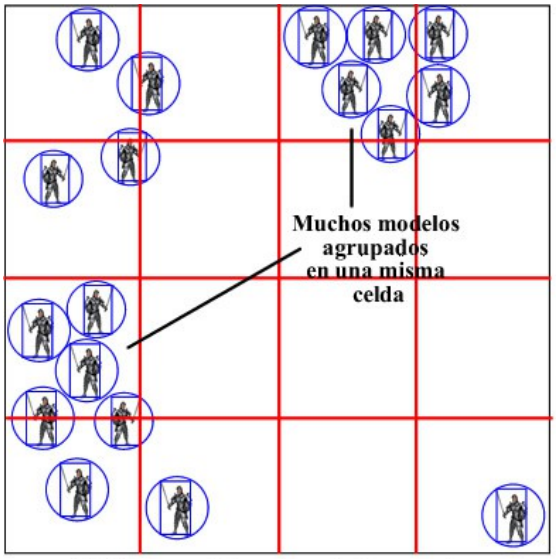


Esquema de escenario con modelos 3D visto desde arriba, con una grilla reglar

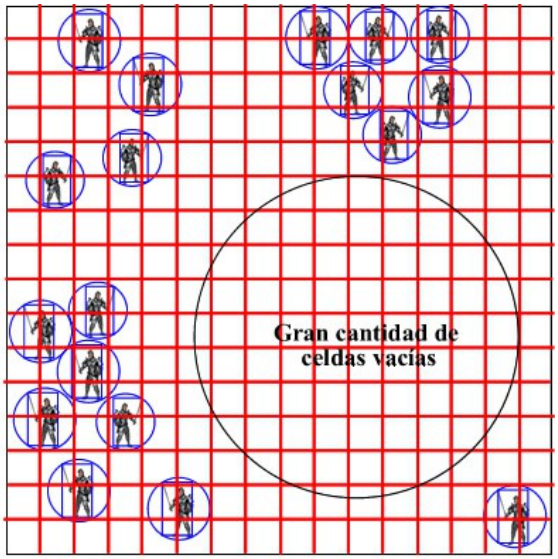
Tamaño de celda y uso de memoria

A la hora de construir la grilla es fundamental determinar cual será el tamaño de la celda. Una celda demasiado grande abarcará muchos modelos en su interior. En el caso extremo, el propio escenario entero es una celda, con lo cual la técnica se reduce a la estrategia de fuerza bruta. Por el otro lado, un tamaño de celda demasiado chico podría hacer que un objeto ocupara varias celdas y eso repercutiría en la cantidad de testeos.

Un escenario muy grande con un tamaño de celda chico puede resultar en un enorme gasto de memoria para las áreas que no contienen ningún modelo.

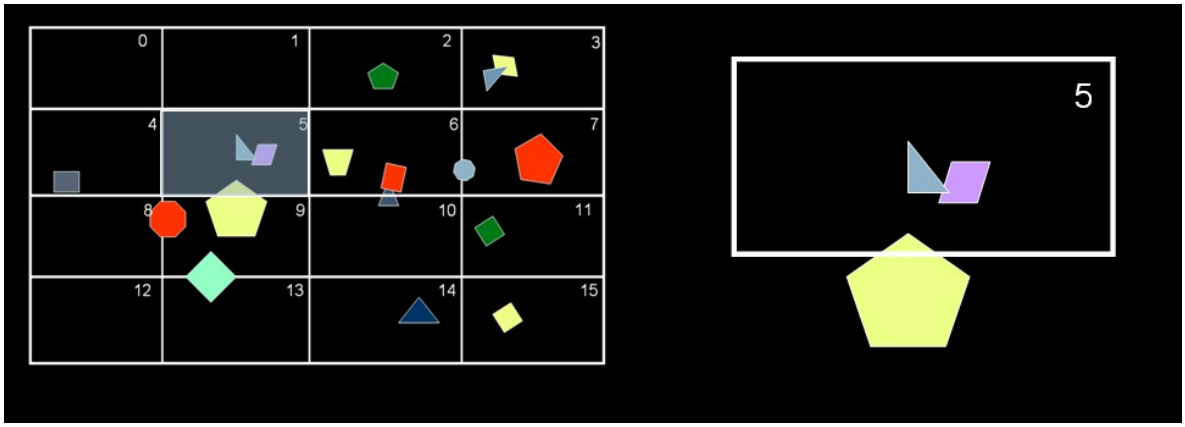


Grilla con celdas demasiado grandes



Grilla con celdas demasiado pequeñas

Una grilla ideal debería contar con celdas de tamaño pequeño para las zonas muy concentradas de modelos (clusters de modelos) y celdas de mayor tamaño para las zonas vacías. Las siguientes tres técnicas son refinamientos de ésta para lograr una solución más eficiente.



Ejemplo de escenario dividido usando una grilla regular. A la derecha contenido de la celda 5.

Como en la grilla regular todas las celdas tienen el mismo tamaño no hace falta mayor información que el tamaño de la celda y del escenario para generar y mantener toda la grilla. Por ejemplo con el número de celda se puede calcular exactamente en que posición de la escena se encuentra. Una estructura de datos para almacenar una grilla regular podría ser esta :

```
struct celda
{
   int cant_obj;
   int lst_objetos[MAX_OBJ];
} CELDAS[MAX_CELDAS];
```

Quadtree y Octree

Concepto y construcción

Octrees y Quadtrees son estructuras de datos que evolucionan a partir de la grilla regular. Su rasgo distintivo es la capacidad de adecuarse a la geometría particular de un escenario mediante la construcción de una grilla adaptativa. Un Octree es un proceso de construcción de grilla adaptativa que abarca los tres ejes cartesianos, mientras que un Quadtree solo abarca dos ejes (generalmente los ejes X y Z). La utilización de uno u otro radica en la naturaleza de la aplicación y el escenario en particular. En el resto de esta sección utilizaremos solo el Octree como herramienta de ejemplo, siendo sencillo llevar luego los conceptos a solo dos dimensiones.

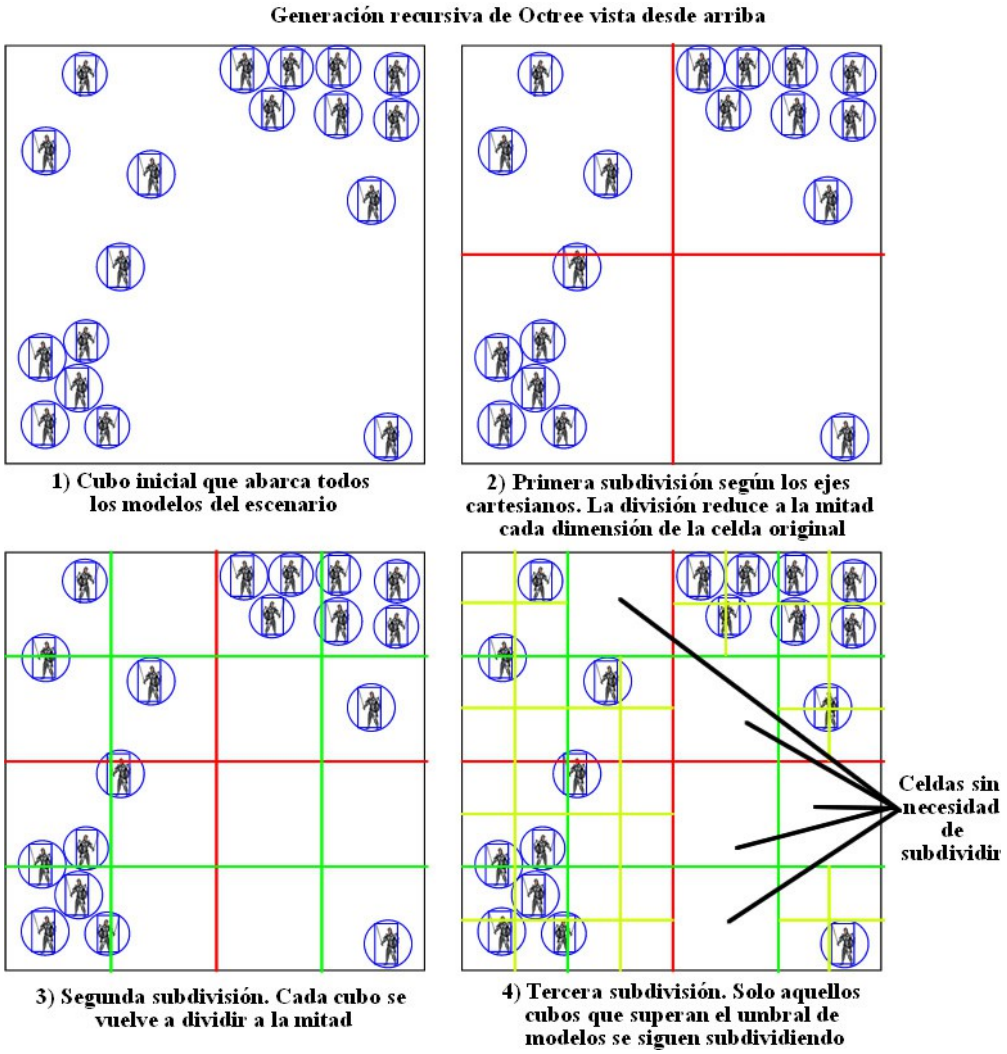
Esta grilla adaptativa no posee todas sus celdas del mismo tamaño, aunque todas son múltiplos entre si del mismo diámetro. La construcción de la grilla se realiza en forma jerarquía e iterativa mediante el siguiente proceso:

1. El escenario es considerado un gran cubo con sus dimensiones expandidas hasta abarcar los puntos más extremos de todos los modelos 3D.

2. Este cubo lógico es partido a la mitad por sus tres ejes, x y z; dejando al cubo original dividido en 8 nuevos sub-cubos o celdas, cada una con sus dimensiones reducidas a la mitad.

3. Cada sub-cubo es a su vez vuelto a dividir a la mitad por sus tres ejes, generando otros 8 sub-cubos nuevos. Este proceso continua para los nuevos subcubos de forma recursiva.

4. Antes de que un cubo comience a realizar una nueva subdivisión, se verifica la cantidad de modelos de todo el escenario que caben dentro de este nuevo cubo disminuido. Para ello se realizan testeos Cubo-Modelo (Bounding Box-Bounding Box) con todos los modelos que ya cabían dentro del cubo mayor que dio origen al cubo actual. Si la cantidad de modelos incluidos en el cubo supera un determinado umbral, se procede a realizar una nueva subdivisión por los tres ejes. De lo contrario el cubo queda como está, con los modelos incluidos dentro de él.



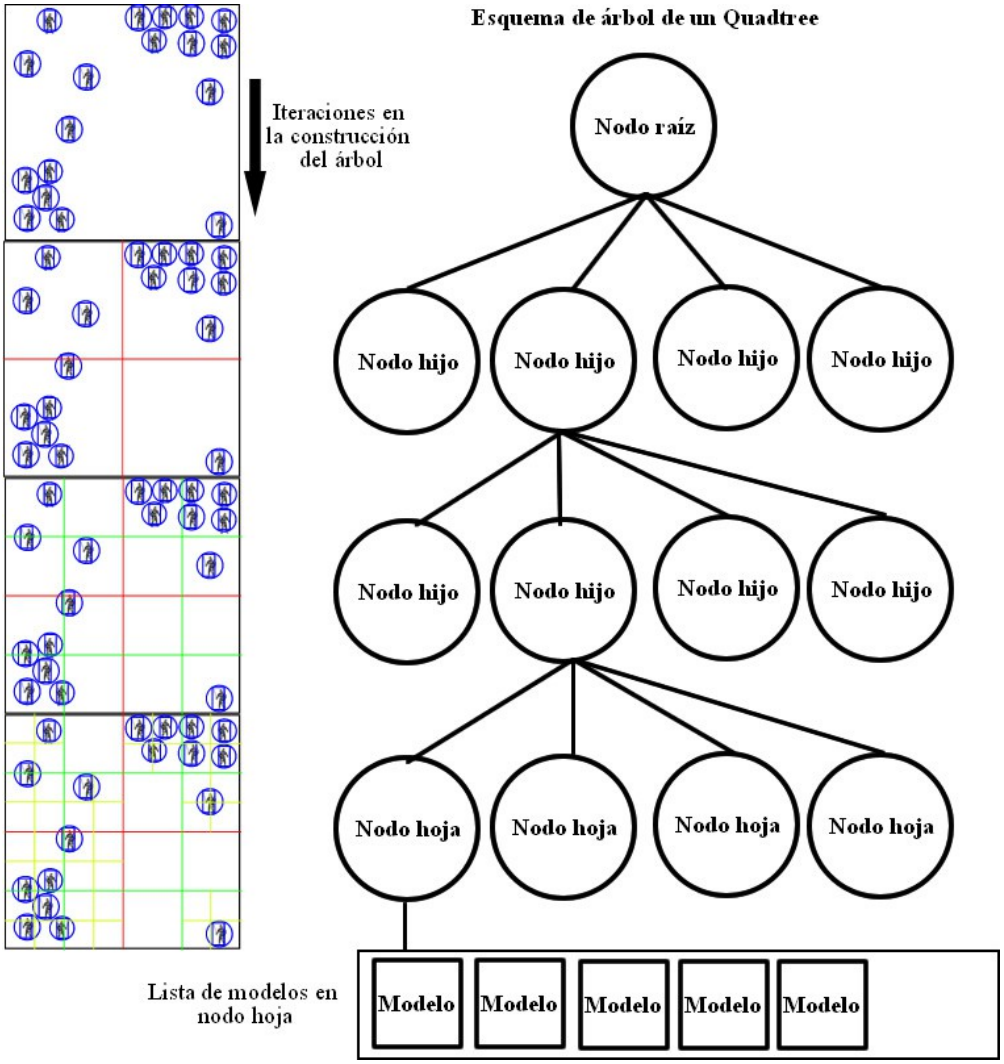
La construcción de esta grilla conforma una estructura de datos de árbol en la que existe un nodo raíz que representa el cubo inicial del escenario, que posee ocho nodos hijos. Cada hijo a su vez puede subdividirse (o no) y poseer ocho hijos más. Los hijos que ya no tienen más subdivisiones se denominan hoja y son los que poseen una lista con todos los modelos contenidos en el volumen de ella.

La estructura de datos para un nodo podría ser la siguiente:

```
//Estructura de Nodo
class OctreeNode
{
	OctreeNode[] children = new OctreeNode[8];
```

```
        Model[] models = null;
}
```

Tanto la raíz como los nodos intermedios del árbol tendrán un array con ocho nodos hijos que representan las subdivisiones que se realizaron por sus tres ejes (8 hijos = 2^3), y tendrán el array de modelos en null. Los nodos hoja en cambio no tendrán nodos hijo pero si tendrán cargada la lista de modelos. La cantidad de modelos que habrá en cada nodo hoja dependerá de la forma en que se construyó el árbol.



La concentración de modelos del escenario determina que tanto se seguirá subdividiendo un nodo. Por lo tanto en aquellos lugares en los que halla más modelos habrá más subdivisiones y en aquellos lugares vacíos habrá muy pocas subdivisiones. Esto produce que el tamaño de las celdas sea más chico donde realmente se necesita ese nivel de precisión, manteniendo celdas de gran tamaño para abarcar las áreas de poca información.

Umbral de corte

En la construcción del árbol se mencionó que era necesario fijar un umbral de corte. Este límite restringe la profundidad máxima del árbol e influye en la cantidad de testeos que será necesario realizar.

El umbral puede ser fijado por los siguientes dos parámetros:

- Cantidad de modelos máximos tolerados en una celda
- Profundidad máxima del árbol

El primer parámetro significa que si la cantidad de modelos que son contenidos en un nodo particular del árbol supera cierto número, el algoritmo proseguirá con la subdivisión recursiva. En caso de no superar este límite, el nodo es considerado una hoja y los modelos contenidos son almacenados en la lista de modelos. El valor de este número dependerá de la aplicación en particular pero algunas consideraciones deben ser tenidas en cuenta:

- Si el valor es muy grande, cada celda contendrá una gran cantidad de modelos reduciendo la eficiencia del método.
- Si el valor es muy chico la cantidad de subdivisiones puede ser muy grande pueden haber objetos que ocupen varias hojas en el árbol situación que hay que tratar de evitar.

La profundidad máxima también debe ser restringida. Este número indica que el árbol no construido no podrá contener más de N niveles. Si un nodo ya alcanzó esa profundidad máxima, no se sigue subdividiendo sino que se convierte en nodo hoja. Este umbral de profundidad es necesario para evitar un crecimiento desmedido del árbol.

Construcción offline

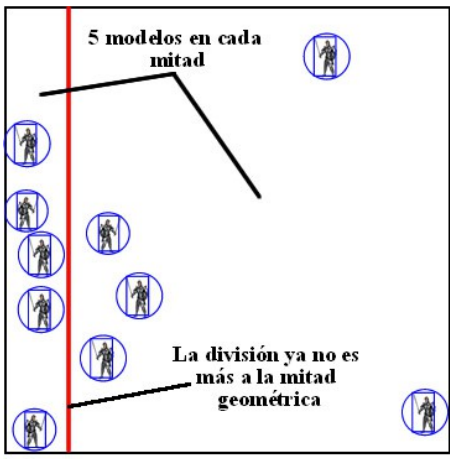
La construcción de una estructura de datos de Octree/Quadtree deja de ser tan sencilla y rápida como lo era la técnica de grilla regular. Es por esto que este algoritmo, y los siguientes que serán explicados, son generalmente ejecutados en forma offline, sus resultados son almacenados en un archivo de datos y luego esa información es utilizada directamente por la aplicación en tiempo real, sin tener que hacer demasiados cálculos adicionales.

No existe un formato estándar en el mercado para almacenar Octrees o Quadtrees, como si lo hay en otros medios como imágenes y sonido. Dado que el tipo de información a almacenar en cada nodo puede variar, cada aplicación suele construir su propio formato de almacenamiento para esta estructura.

KD-Tree : Balancear el árbol

La técnica de KD-Tree es una evolución directa de los Octrees/Quadtrees que tiene como objetivo lograr un árbol balanceado sobre el escenario de modelos. Para lograr esto propone que las divisiones por los tres ejes X, Y, Z no se realizan a la mitad espacial del tamaño de cada nodo sino a la “mitad de densidad” de modelos ubicados en ese nodo. Es decir, si una celda tiene una longitud en el espacio tridimensional de 10x10x10, en un Octree terminaría generando ocho

nodos hijos de longitud 5x5x5 cada uno (siempre y cuando se cumpla el umbral de subdivisión establecido). Lo que propone la estrategia de KD-Tree es que la dimensión de los nuevos nodos hijos no sea fija, sino que sea variable en función a la distribución de modelos que se encuentran contenidos. Si ese mismo nodo inicial tuviera 6 modelos y lo quisiéramos dividir a la mitad del eje Y pero según la densidad de modelos y no según la mitad espacial, deberíamos encontrar una distancia de corte tal que separe el nodo en dos mitades que contengan 3 modelos cada una.



Nodo dividido por el eje Y en función de la densidad de modelos

Al dividir siempre por la mitad en función de la densidad de los modelos, la cantidad de modelos en cada celda es similar y el árbol resultante se asemeja mucho más a un árbol balanceado. Al tener un árbol balanceado la performance de acceso a la estructura para realizar clasificaciones es mucho mayor.

Como ahora todos los nodos no tienen el mismo tamaño, es necesario cambiar la estructura de datos para almacenar los valores exactos por los cuales se ha producido el corte espacial en cada eje:

```
//Estructura de Nodo
class KdTreeNode
{
        KdTreeNode[] children = new KdTreeNode[8];
        Model[] models = null;

        //cortes de los tres ejes
        float xCut;
        float yCut;
        float zCut;
}
```

Construcción

El árbol generado con la técnica de Kd-Tree posee varias ventajas frente a un Octree/Quadtree pero su construcción se vuelve bastante más complejo:

1. El escenario es considerado un gran cubo con sus dimensiones expandidas hasta abarcar los puntos más extremos de todos los modelos 3D.

2. Este cubo lógico es partido a la mitad por cada eje según la densidad de modelos presentes en el mismo:

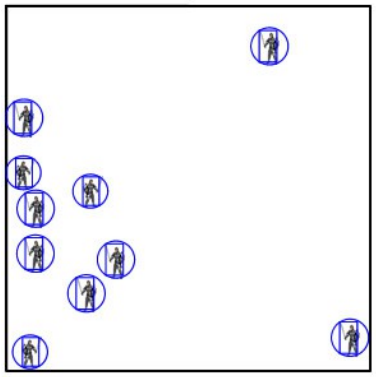
    a. Se toma el eje X y se cuenta la cantidad total de modelos que hay en el nodo. Se busca una línea de corte en este eje que permita dividir los modelos en dos mitades iguales. En caso de ser impares una mitad puede quedar con un modelo más que la otra o alguno de los modelos puede pertenecer a ambas mitades.

    b. Se repite este mismo proceso para los ejes Y y Z.

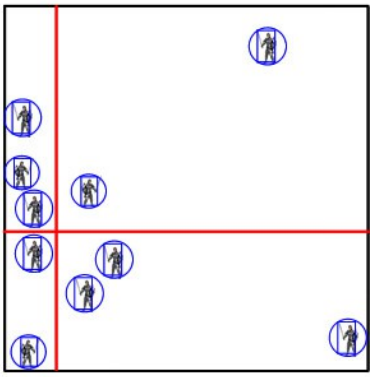
    c. Se almacena en el nodo padre las distancias de corte elegidas para cada uno de los ejes.

3. Cada sub-cubo es a su vez vuelto a dividir por sus tres ejes según la densidad de modelos. El proceso continúa en forma recursiva hasta alcanzar el umbral de corte establecido, al igual que en la técnica de Octree/Quadtree.

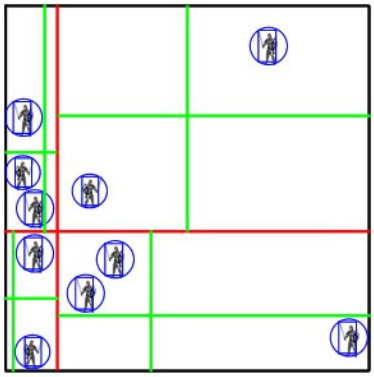
Esquema de creación de un KD-Tree visto desde arriba



1) Cubo inicial que abarca todos los modelos del escenario



2) Primera iteración. El nodo es subdividido según la densidad de modelos



3) Tercera iteración. Cada nodo hijo es vuelto a dividir según la densidad de modelos

Encontrando la mejor subdivisión

Para que esta estrategia funcione correctamente es necesario encontrar, para cada eje de cada nodo, el mejor plano de corte. El procedimiento de dividir un espacio en dos mitades se denomina “Split” y el criterio que se utiliza para encontrar donde realizar esta división es el “Best fit”. El objetivo es encontrar la distancia óptima por la cual separar las dos mitades de un nodo, según alguno de los tres ejes de corte. Como la construcción del Kd-Tree es offline, no existen restricciones de tiempo real sobre la misma por lo tanto puede dedicarse bastante procesamiento de cálculo para encontrar el mejor plano de corte.

Una de las técnicas más utilizadas se denomina “Divide y conquistarás” (divide and conquer), y consiste en los siguientes pasos:

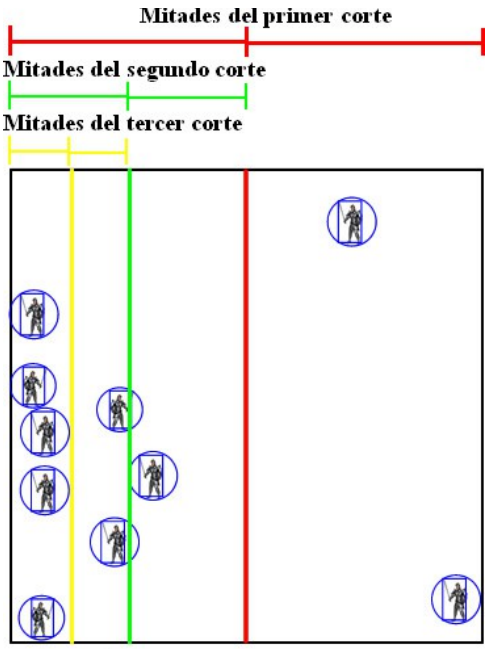
1. Dado un nodo con un cierto volumen y un eje por el cual realizar la partición, dividir el nodo por la mitad espacial y contar la cantidad de modelos que quedó en cada mitad:

    a. Si ambas mitades poseen la misma cantidad de modelos entonces ya hemos encontrado el mejor plano de corte para este eje y terminamos el proceso.

    b. Si existen diferencias en ambas mitades, se procede con el siguiente paso, excepto que haya una diferencia de uno cuando el total de modelos del nodo era impar.

2. En la mitad generada anteriormente que mayor cantidad de modelos tiene, volver a realizar un corte a la mitad y tomar este nuevo plano con el plano central de corte del nodo. Contar la cantidad de modelos que quedaron en cada una de las mitades. Si son distintas con el punto 1.b continuar en forma recursiva este proceso.

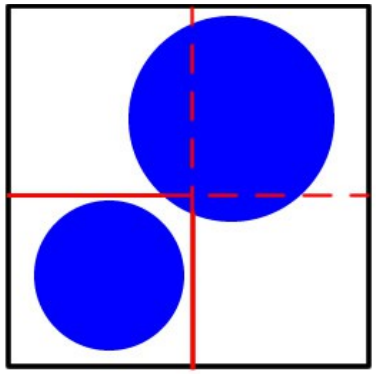
3. El proceso debe continuar hasta encontrar un plano de corte que separa el nodo en dos mitades con igual cantidad de modelos (o una mitad con un modelo más si son impares). Pero también debe poseer un número máximo de iteraciones para cada eje, porque hay ciertas configuraciones de modelos que no podrán ser divididos correctamente a la mitad con esta técnica.



Método Divide-and-Conquer para encontrar el mejor plano de corte

Limitación de los cortes

Existen ciertas configuraciones de modelos para los cuáles no existe un plano de corte alineado a los ejes que permita separar esos modelos en dos mitades de igual cantidad. El Kd-Tree al igual que un Octree y Quadtree realiza cortes por sus tres ejes en forma alineada a los mismos, es decir que todos los cortes se realizan en forma vertical, horizontal o longitudinal. Esto produce que ciertos modelos no puedan ser separados correctamente:



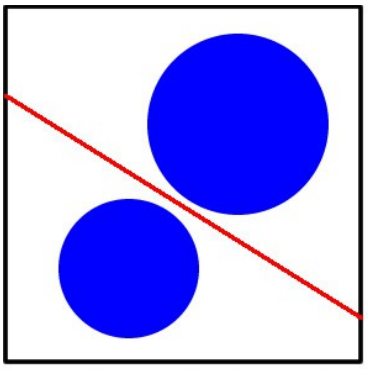
Modelos que no pueden ser divididos mediante cortes rectos

Cuando este caso sucede no será posible encontrar un plano de corte que genere dos mitades iguales. Por lo tanto habrá que optar por tener una mitad con dos modelos y otra con ninguno, o compartir uno de los modelos en ambas mitades.

Esta imposibilidad con ciertas configuraciones de modelos hace que el KD-Tree no pueda ser un árbol perfectamente balanceado.

BSP Tree.

El BSP-Tree termina la línea de evolución iniciada con la grilla regular e intenta alcanzar como resultado un árbol completamente balanceado de rápido acceso. La táctica anterior podía obtener resultados bastantes buenos pero se veía limitada con ciertas configuraciones de modelos porque solo podía efectuar cortes rectos por los ejes. En un árbol BSP los cortes que se pueden hacer para dividir el volumen de un nodo en dos mitades no tienen ninguna restricción, pueden hacerse cortes en diagonal con cualquier dirección. Matemáticamente si dos convexos no intersectan entre sí, entonces siempre existirá un plano que pueda separarlos, por lo tanto tenemos garantizado encontrar un corte que siempre separa en mitades equitativas el volumen de nuestros nodos.



Solución de modelos inseparables mediante planos no paralelos a los ejes

BSP son las iniciales de “Binary Space Partitioning” que indican que el espacio de nuestro escenario ahora es dividido en forma binaria. Ya no contamos más con esquema de grilla con celdas sino que tenemos mitades, que a su vez se dividen en dos mitades, cada una vuelta a dividir y así sucesivamente. El árbol deja de tener 4/8 hijos como en las técnicas anteriores y pasa a convertirse en un árbol binario. Cada nodo del árbol tendrá una estructura similar a la siguiente:

```
//Estructura de Nodo
class BspNode
{
	BspNode positiveChild;
	BspNode negativeChild;
	Model[] models = null;

	//plano de corte
	float nx;
	float ny;
	float nz;
	float d;
}
```

Dado que es un árbol binario se necesitan dos referencias a sus hijos: positive-negative o front-back. Estas referencias se obtienen como resultado de clasificar del testeo de colisión entre un plano y un modelo. Además es necesario almacenar en cada nodo el plano de corte, dado que es variable para cada uno, por lo tanto necesitamos guardar los valores de la ecuación del plano <N,D>.

Al igual que un Octree o Kd-Tree el árbol está compuesto por nodos intermedios que no tienen ningún modelo y nodos hojas que contienen la lista de modelos y a su vez no tienen ningún hijo.

Construcción

El algoritmo BSP introduce mucha más flexibilidad en la forma de dividir nuestro escenario en secciones pero esta libertad extra también complica en forma significativa el proceso de construcción del árbol.

Los pasos a seguir para de generación del árbol varían un poco a los utilizados en las técnicas anteriores. En un Octree o KD-Tree luego de dividir un nodo por los tres ejes se generaban 4/8 nodos hijos. En árbol BSP en cambio siempre realizaremos divisiones binarias. El concepto de eje ya no existe porque estamos pariendo el espacio con una inclinación arbitraria que incluye siempre a los tres ejes.

Los a pasos a seguir son:

1. Se arranca con el nodo raíz, que identifica el volumen de todo el escenario, y con la lista completa de modelos existentes que se quieren clasificar en el árbol.

2. Se crean dos listas temporales de modelos: lista positiva y lista negativa.

3. Se selecciona una plano de corte (su elección será analizada luego). Por cada modelo del nodo se realiza un testeo de colisión Plane-Bounding Box en el que se pueden obtener tres resultados:

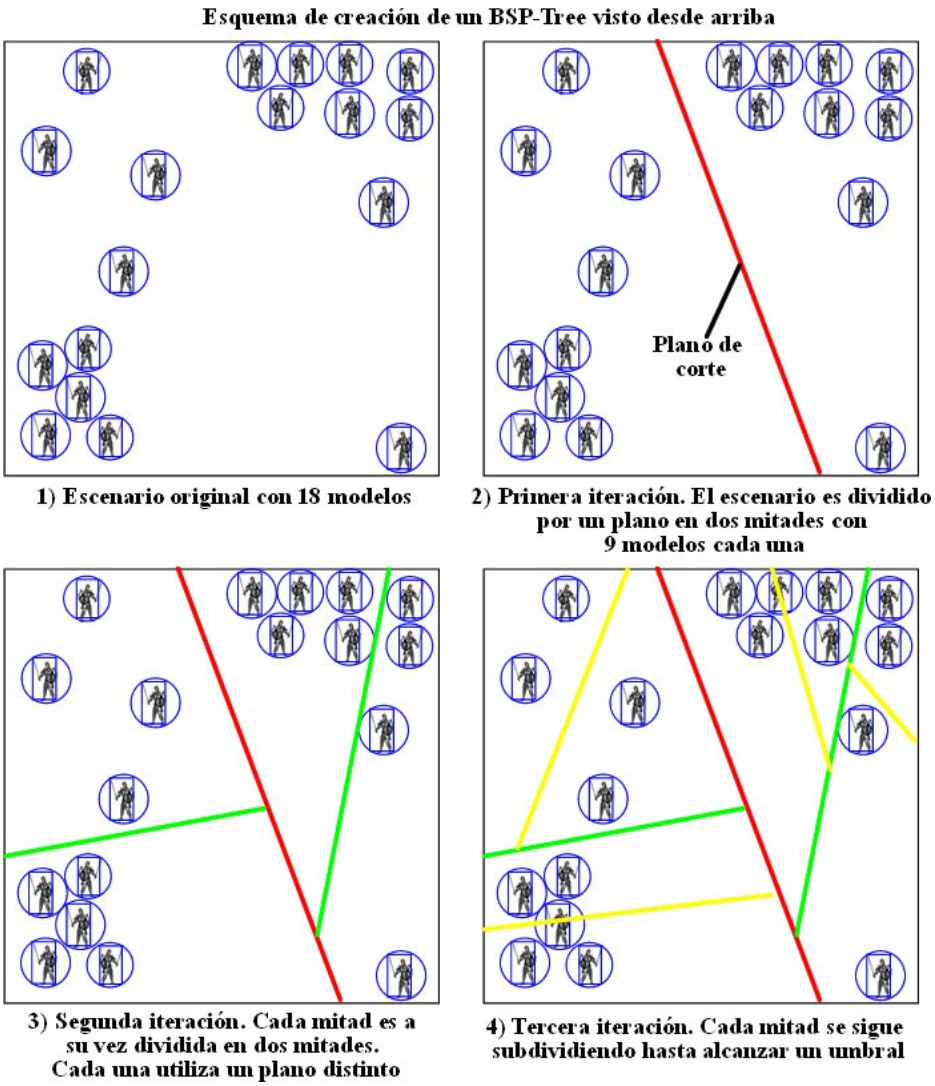
    a. El modelo se encuentra completamente en el lado positivo del plano (o front-side). En este caso el modelo es almacenado en la lista temporal positiva creada anteriormente.

    b. Si el modelo se encuentra completamente en el lado negativo del plano (o back-side) se almacena en la lista temporal negativa de modelos.

    c. Si el modelo se encuentra atravesando el plano se adjunta tanto a la lista positiva como a la negativa. Otra opción sería partir el modelo en dos (split) que será analizada luego.

4. El proceso se ejecuta dos veces en forma recursiva. La primera vez se invoca utilizando solo los modelos que quedaron en la lista positiva, mientras que la segunda vez se invoca utilizando la lista negativa. De esta forma los modelos van decantando según la mitad a la que corresponden.

5. El proceso continúa generando los nodos de todo el árbol hasta que se alcanza un umbral de corte, que puede ser la cantidad de modelos máxima tolerada por un nodo o que ya no quede ningún modelo por clasificar.



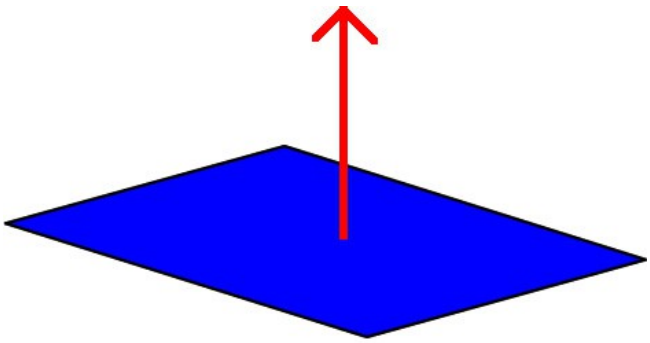
Si se eligen buenos planos de corte el resultado final del algoritmo de construcción es un árbol binario muy bien balanceado con una excelente performance de búsqueda.

Plano de corte

El gran potencial de la técnica de BSP nunca será alcanzado si no se eligen buenos planos de corte que terminen generando un árbol bien balanceado. Dado que ahora tenemos libertad total de elegir la dirección de corte que queramos la forma de detectar la mejor subdivisión no es tan sencilla.

Para determinar como separar un espacio en dos es necesario crear un plano, por lo tanto es necesario especificar dos cosas:

- N: es la normal al plano que indica la dirección en la que se hará el corte.
- D: es el desplazamiento del plano respecto del origen de coordenadas, en la dirección N.

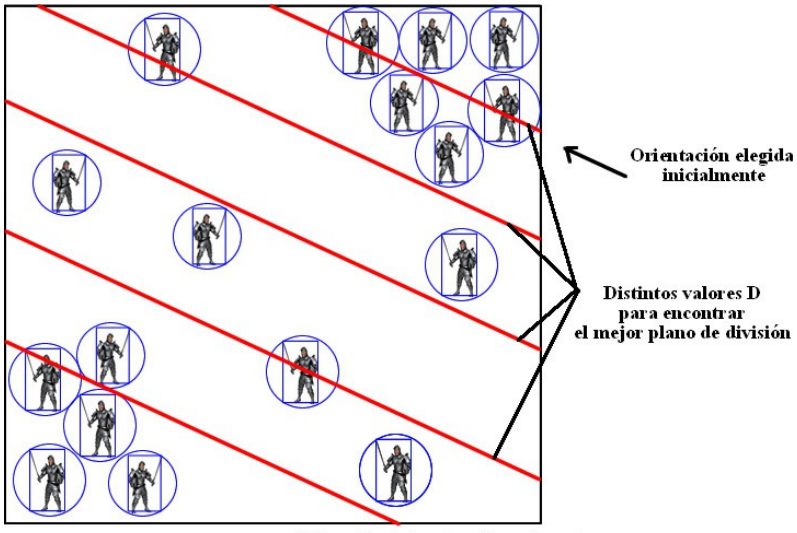


Vector normal del plano

Jugar con ambos elementos nos da una cantidad infinitas de posibilidades así que por lo general se suele fijar el valor de un vector normal y se encuentra el mejor valor de D para esa inclinación que divida en dos mitades equitativas el nodo. El valor N puede ser obtenido de varias formas:

- Crear un vector en forma aleatoria.
- Obtener el vector normal de algún triángulo de algún modelo. Este suele ser el método más utilizado.

Una vez que se ha establecido una dirección se puede utilizar el método "divide-and-conquer" visto anteriormente para calcular el valor de D que mejor divide el espacio.

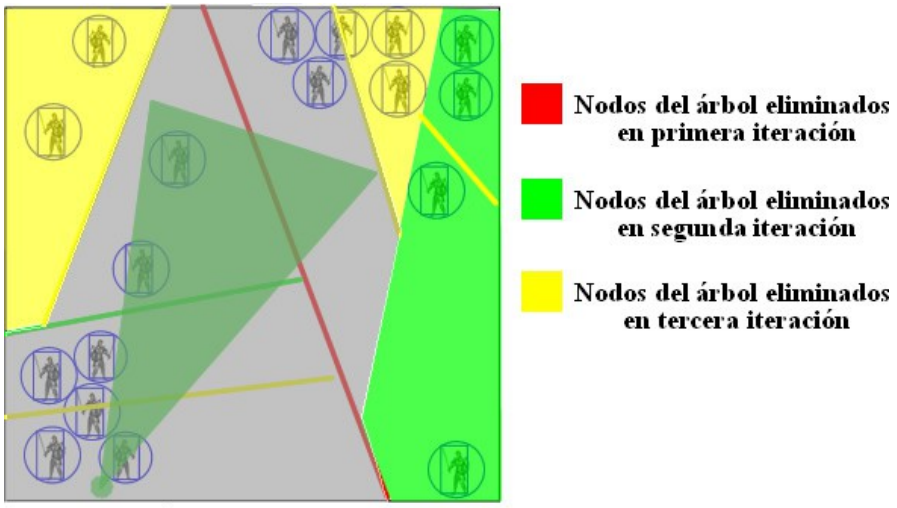


Obtención del mejor plano de corte

Dado que el proceso de construcción del BSP se realiza en forma offline pueden elegirse varios vectores normales distintos para cada iteración y luego seleccionar el que mejor resultados brindó. Incluso pueden construirse varios árboles BSP y luego elegir aquél que más balanceado resultó.

Utilización en gráficos.

Las estructuras de subdivisión del escenario que mencionamos, tienen un uso intensivo en numerosas aplicaciones de la computación gráfica en general y en los videojuegos en particular. La misma estructura de datos que sirve para acelerar las búsqueda de colisiones en el motor de física, también se usa para acelerar el motor de gráficos. Al testear intersecciones entre los objetos y el frustum de la cámara, la idea es determinar rápidamente que objetos son visibles o no desde el punto de vista del jugador, para renderizar sólo aquellos que son potencialmente visibles. Es decir no dibujar nada que este fuera del alcance de la cámara. La mayoría de estas técnicas evolucionaron a partir de la necesidad de optimizar el rendimiento de los motores de gráficos en la época que no había placa de video y la velocidad de renderización era demasiado lenta para dibujar todos los modelos. De no aplicar este tipo de optimizaciones juegos como el doom o el quake no hubiesen sido posibles en aquella época. Hoy en día las mismas estructuras se siguen usando para lograr más modelos o más complejos a la misma velocidad, para soportar objetos transparentes, y para otras aplicaciones dentro del video juego como la inteligencia artificial y la física como estamos viendo en este apunte.



Utilización de Frustum en un BSP

Historia del BSP: Quake 1

Existe una variación de la técnica de BSP denominada Leafy-BSP que se hizo muy popular tras su utilización en el videojuego Quake 1, desarrollado por Id Software en 1996 con John Carmack a la cabeza del desarrollo. Quake 1 fue el primer videojuego que renderizaba gráficos realmente en tres dimensiones en tiempo real. Juegos anteriores como Doom o Wolfestein eran denominados “Engines 2.5D” dado que no eran realmente 3D, sino que utilizaban algunos trucos visuales para engañar al ojo. Quake 1 fue el primero juego en poder manejar gráficos tridimensionales en tiempo real a una velocidad decente e inicio toda una

revolución en el mundo de los videojuegos e incluso en el mundo del hardware, impulsando las primeras tarjetas aceleradoras de video.

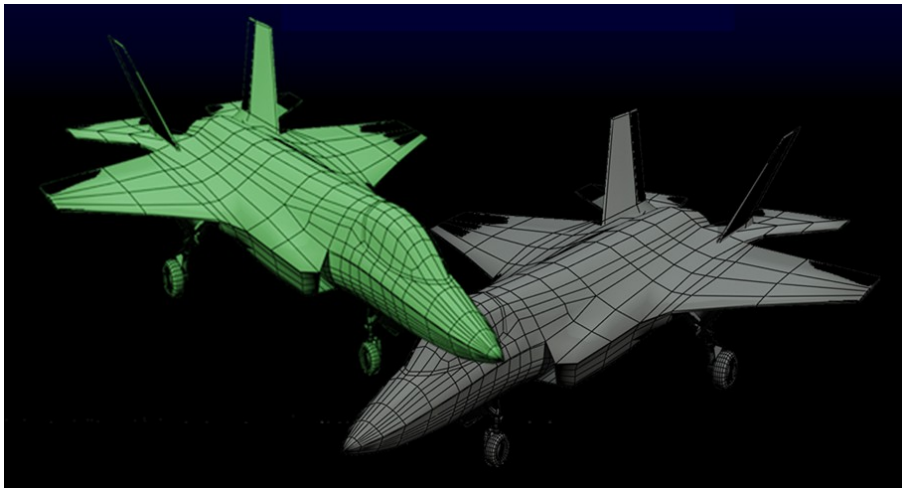
Una de las herramientas que hizo posible la manipulación de gráficos en tres dimensiones en una aplicación en tiempo real fue el uso de BSP como estructura de datos principal para administrar la geometría de cada nivel del juego.

Narrow Phase.

Una vez que el Broad Phase se complete y tengamos una lista de pares de objetos candidatos a colisionar pasamos a la fase final, el Narrow Phase donde se verifica si existe o no colisión, y en el caso de que haya, los elementos de la misma para que puedan ser usados en la etapa final, el collision response.

En esta etapa se pueden usar los distintos algoritmos de intersección entre figuras como los que vimos al principio de la unidad, como por ejemplo SAT para colisión entre polígonos convexos.

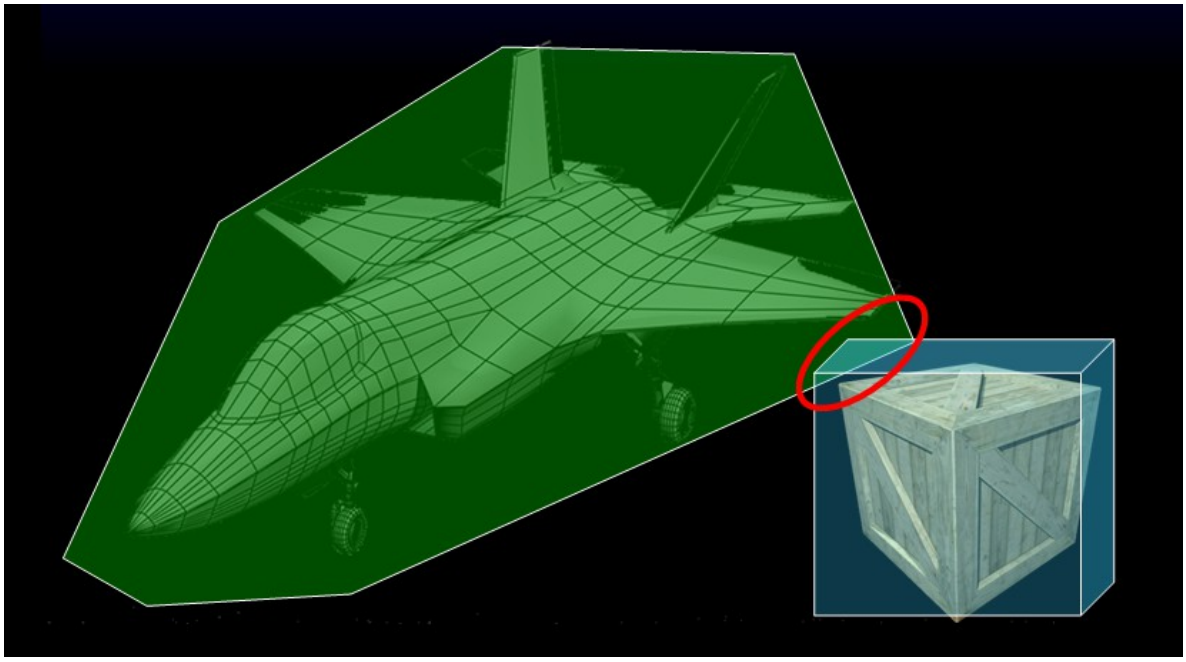
Sin embargo cuando las modelos son complicados (especialmente en los motores de física que trabajan en 3d), el test de intersección exacto puede ser impracticable. Imaginemos un modelo de un avión compuesto de miles de triángulos y queremos ver si colisiona contra otro avión también compuesto por otros tantos miles de triángulos. Para determinar la colisión exacta deberíamos testear triángulo por triángulo de cada uno de los modelos. Muchas veces esto no se puede hacer en el poco lapso de un paso de update, por lo que se busca una solución intermedia: un volumen de simplificación.



Detectar la colisión exacta requiere chequear una enorme cantidad de triángulos entre si.

Una solución consiste en reemplazar el modelo exacto del objeto, por otro más simple que lo contenga en su totalidad y que sea mucho más fácil por intersecciones, como pueden ser un caja, una esfera o un cuerpo convexo. A este tipo de volúmenes se los llama volúmenes de simplificación. Los más utilizados son

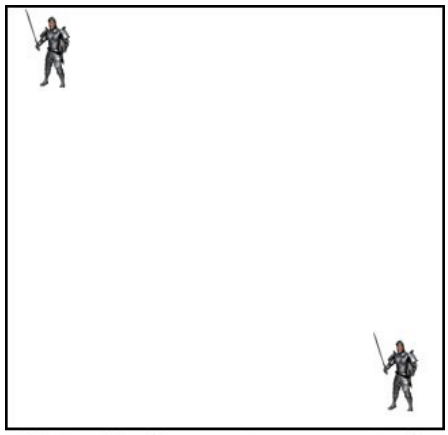
- Bounding Sphere
- Bounding Ellipsoid
- Axis Aligned Bounding Box
- Oriented Bounding Box
- Convex Hull



Test de intersección entre cuerpo convexo y caja.

Simplificación de modelos

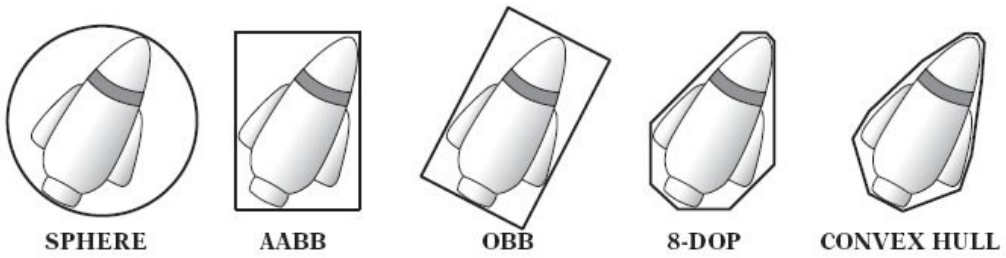
Realizar una detección de colisiones triángulo por triángulo de todos los modelos de un escenario es la forma más precisa pero también la menos eficiente. Muchas veces enfocarse en calcular colisiones entre dos modelos triángulo por triángulo es innecesario. Por ejemplo, si dos modelos se encuentran en rincones completamente opuestos del escenario probablemente ningún triángulo colisione.



Objetos ubicados en extremos opuestos de un escenario. No existe ninguna colisión de triángulos entre ellos

Lo idea sería poder descartar en forma rápida y liviana (computacionalmente) aquellos modelos con muy pocas probabilidades de colisionar. Una forma de lograr esto es mediante la simplificación de modelos. En lugar de considerar un modelo 3D como un conjunto de triángulos desparramados por el espacio, se

confecciona un volumen simple que lo englobe, como una caja o una esfera. El objetivo es que este volumen posea una geometría bien sencilla, que haga que los cálculos de colisión sean ágiles.



Distintos volúmenes simplificados para englobar un modelo 3D

La forma de proceder en la detección de colisiones con estos volúmenes sería la siguiente:

1. Testear el objeto colisionador contra los volúmenes simplificados de los demás objetos.
2. Si no hay colisión, no hace falta seguir verificando sus triángulos interiores.
3. Si hay colisión con su volumen simplificado, no podemos asegurar que sea realmente una colisión. Esto es porque el volumen simplificado suele ocupar más espacio real que la suma de los triángulos del modelo. En este caso debe seguir realizando testeos a nivel de triángulos o aceptar un cierto nivel de error en el cálculo de colisiones.

A continuación se detallan algunos de los volúmenes más utilizados.

Axis-aligned Bounding Box (AABB)

El axis-aligned bounding box (AABB) es una de los volúmenes simplificados más utilizados. Es una caja rectangular de seis caras que se caracteriza por tener sus caras orientadas de forma tal que sus normales son siempre paralelas a los ejes cartesianos. La mejor característica del AABB es su rápido chequeo de superposición, que implica simplemente la comparación directa de los valores individuales de las coordenadas.

La forma más simple de representar este volumen es mediante su punto máximo y mínimo:

```
class AABB
{
	Point minVertex;
	Point maxVertex;
}
```

Axis-aligned Bounding Box aplicado a un modelo 3D

Ambos puntos representan las esquinas opuestas de la caja. Dos cajas de este tipo se superponen entre sí solo si sus tres ejes se superponen. El procedimiento para detectar colisión es el siguiente:

```
int TestAABBAABB(AABB a, AABB b)
{
        // No hay intersección si están separados en al menos un eje
        if (a.max.x < b.min.x || a.min.x > b.max.x) return 0;
        if (a.max.y < b.min.y || a.min.y > b.max.y) return 0;
        if (a.max.z < b.min.z || a.min.z > b.max.z) return 0;
        // Superposición en todos los ejes significa que hay colisión
        return 1;
}
```

El procedimiento anterior posee un costo computacional muy bajo y permite rechazar en forma temprano aquellos volúmenes de modelos que no colisionan entre sí.

Para calcular el mínimo Bounding Box de un modelo 3D se siguen los siguientes pasos:

1. Iterar sobre todos los vértices de todos los triángulos del modelo
2. Por cada vértice almacenar los máximos valores de X, Y y Z, cada uno por separado.
3. Realizar lo mismo para los mínimos valores de X, Y y Z.

4. Confeccionamos el punto máximo y mínimo del volumen en base a estos valores. El centro de la caja es la mitad de la diferencia entre ambos extremos.

Los valores de Bounding Box deben actualizarse cada vez que el modelo realiza una variación de su posición en el espacio. Para esto no es necesario volver a recalcular ambos valores extremos de la caja. Es suficiente con desplazar el centro del volumen y luego aplicar la variación al valor máximo y mínimo de la figura.

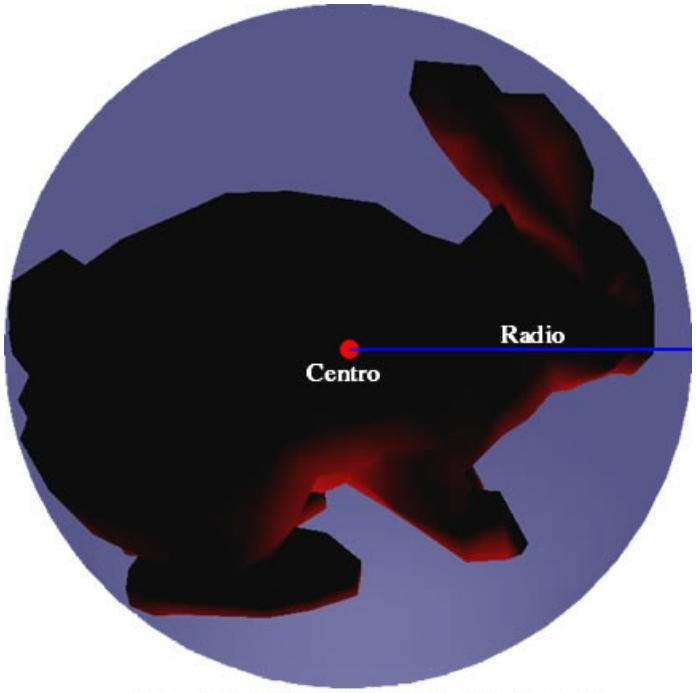
Esto no es válido cuando el modelo realiza rotaciones a lo largo del escenario. Este volumen de simplificación es dependiente de la orientación del modelo. En caso de que esta rotación se vea modificada habrá que aplicar una estrategia de rotación de Bounding Box u optar por volver a recalcular el volumen.

Bounding Sphere

La esfera es otro volumen de simplificación muy común. Al igual que los AABB, las esferas poseen un test de intersección muy liviano. Además tienen el beneficio de ser independientes de la rotación del modelo. Simplemente hay que trasladar su centro a la nueva posición.

Una esfera se define en términos de un centro y su radio:

```
class Sphere
{
        Point center;
        float r;
}
```



Bounding Sphere aplicada a un modelo 3D

Al tener solo cuatro componentes (3 floats par su centro y uno para el radio) la esfera es el volumen mas eficiente en cuanto memoria requerida.

El test de superposición entre dos esferas es muy simple. Se calcula la distancia euclidiana entre los centros de las dos esferas y se compara contra la suma de sus radios:

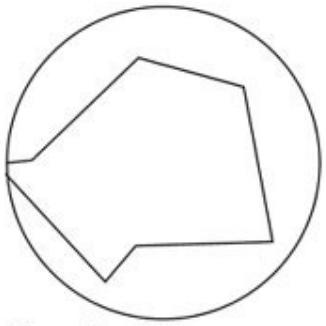
$$S_1 : C = (x_1, y_1, z_1), R1$$
$$S_2 : C = (x_2, y_2, z_2), R2$$
$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} <= R1 + R2$$

Algorítmicamente, para evitar realizar una operación costosa de raíz cuadrada, se comparan directamente las distancias al cuadrado:

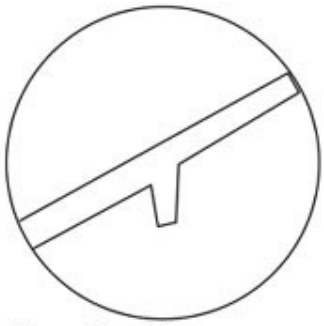
```
int TestSphereSphere(Sphere a, Sphere b)
{
        // Calcular distancia al cuadrado entre los centros
        Vector d = a.c - b.c;
        float dist2 = Dot(d, d);
        // Hay interseccion si la distancia al cuadrado es menor o igual a la suma de los //
        radios al cuadrado
        float radiusSum = a.r + b.r;
        return dist2 <= radiusSum * radiusSum;
}
```

A pesar que el test de intersección de esferas tiene algunas operaciones aritméticas de más que la intersección de AABB, también tiene menos ramificaciones (cláusulas IF) y requiere capturar meno datos de memoria. En las arquitecturas modernas de CPU, el test de esferas superpuestas es ligeramente más rápido que el de AABB.

La principal desventaja de una esfera frente a un AABB es que muchas veces agrega demasiado espacio desperdiciado al volumen simplificado:



Bounding Sphere que se adapta bastante bien al modelo



Bounding Sphere con mucho volumen desperdiciado

Por lo tanto, la elección entre un volumen de simplificación y otro dependerá de la naturaleza del modelo.

Cálculo de un Bounding Sphere

Calcular la mínima esfera que envuelve a un modelo no es una tarea trivial. Existen diversos métodos para encarar el problema. Una aproximación simple puede ser obtenida a partir del Bounding Box del modelo, a través del siguiente procedimiento:

1. Obtener el Axis-aligned Bounding Box del modelo, de la forma que se detalló anteriormente.
2. El punto medio del AABB se considera el centro de la esfera.
3. El radio de la esfera es la distancia que existe entre el centro y uno de los vértices extremos de la AABB.

El procedimiento anterior logra una aproximación fácil de calcular pero que no siempre puede ser la mínima esfera que engloba a ese modelo.

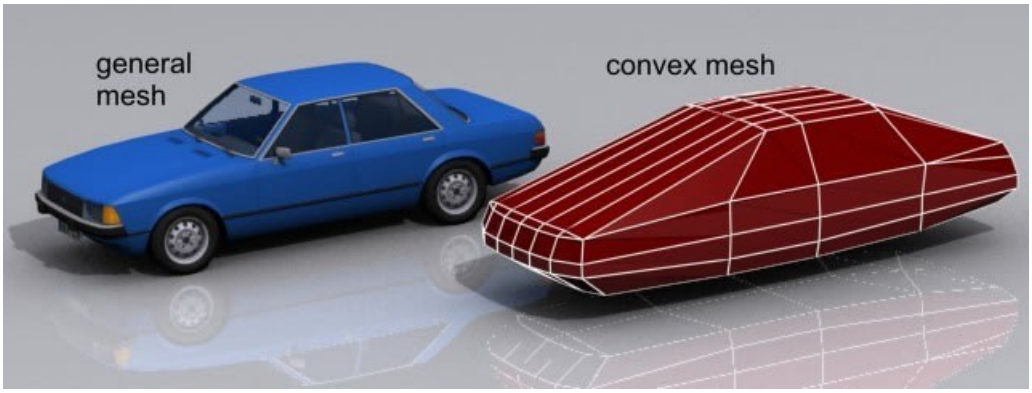
Otro procedimiento más costoso pero que suele otorgar mejores resultados es el siguiente:

1. Obtener el Axis-aligned Bounding Box del modelo, de la forma que se detalló anteriormente.
2. El punto medio del AABB se considera el centro de la esfera.
3. Se calcula la distancia desde el centro hacia todos los vértices originales de la malla.
4. La mayor distancia encontrada se considera el radio de la esfera.

Así y todo este segundo procedimiento puede no obtener la mínima esfera que envuelve a un cuerpo. Alternativas más complejas utilizan cálculos estadísticos, como desviación estándar y covarianza, y cálculos matriciales con autovalores.

Convex-Hull

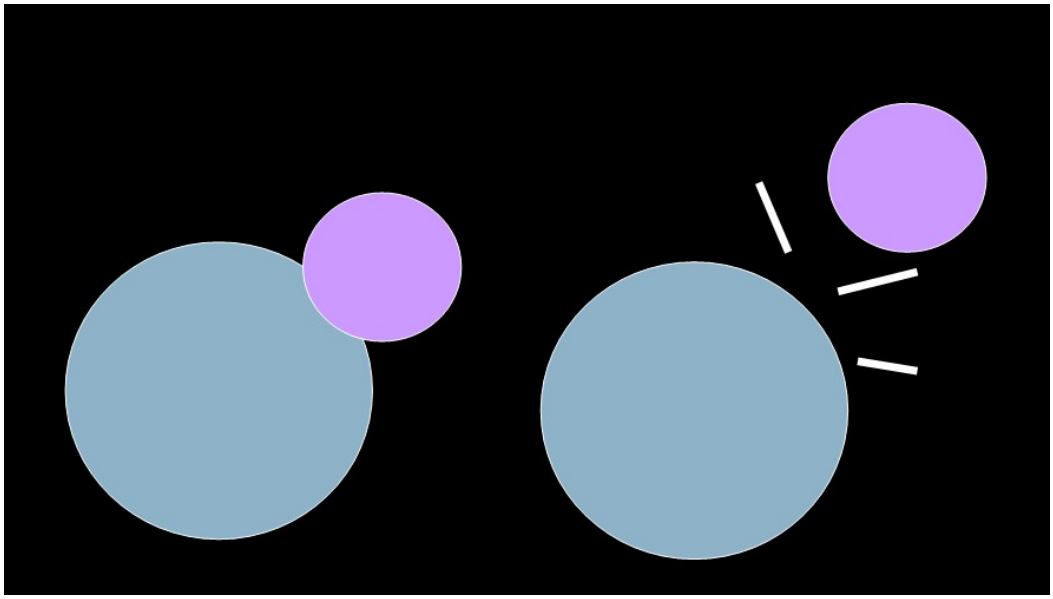
Un Convex-Hull es el menor polígono convexo que contiene a un modelo 3D. Su geometría permite un volumen simplificado suficientemente apretado, que logra muy buena precisión en la detección de colisiones, pero a costa de un incremento computacional significativo respecto de los otros dos volúmenes vistos.



5- Collision Response: Impulse Resolution.

Una vez que el sistema de detección de colisiones determinó que objetos están colisionando, es tiempo que el sistema de resolución de la colisión determine que hacer con los objetos en cuestión. La etapa de collision response recibe como datos los distintos pares de objetos que están colisionando más los elementos de la colisión (punto de contacto, penetración y normal, etc) y decide que hacer con los cuerpos. La idea es que hay resolver la colisión para o bien evitarla o bien que en los próximos frames los objetos dejen de estar en situación de colisión. Una forma es evitar que la colisión antes de que esta se produzca. Si en un frame anterior los objetos no estaban colisionando y en el frame actual están en colisión, se puede volver atrás la posición de los objetos en un estado intermedio donde no estén en colisión.

Otro método de collision response, es de impulse resolution o resolución por impulso, que es el que vamos a desarrollar en este apunte. El método de impulse resolution fue popularizado por su uso en el motor de física box2d y se basa en generar un impulso (cambio instantáneo en la velocidad) a los cuerpos que entran en colisión, en el sentido contrario al impacto, de tal forma que en próximos frames tiendan a separarse y salir del estado de colisión.



Impulse Resolution. Una vez detectado que dos cuerpos colisiones el motor genera un impulso que tiende a separar los objetos para evitar la colisión en frames posteriores.

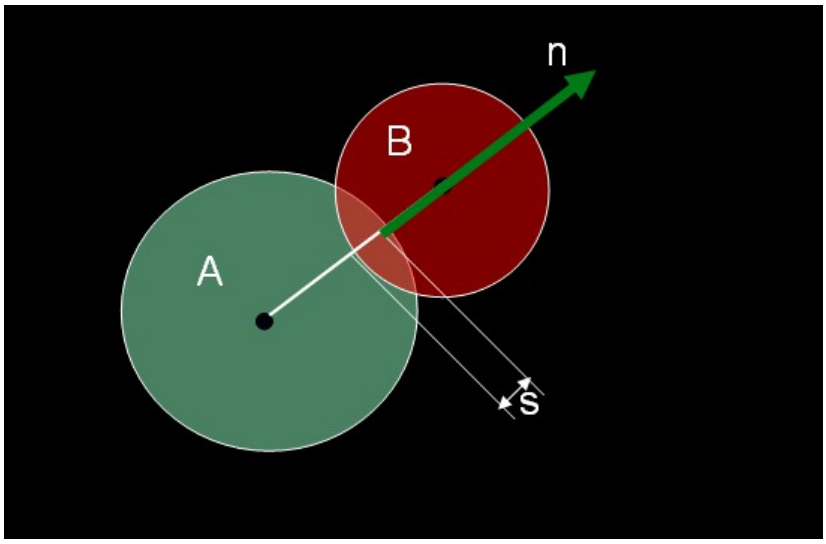
El método asume que durante el impacto entre los dos objetos la fuerzas más importantes son las fuerzas producto de la colisión, por lo cual ignora el resto de las mismas al menos por ese instante. En base a esto computan las velocidades resultantes luego del impacto. Como la velocidad de los cuerpos cambia instantáneamente de un frame a otro, se llama impulse (impulso) resolution a esta técnica.

Uno podría pensar que sería más fácil modificar directamente la posición de los objetos que entran en colisión para evitar la misma. En la práctica las simulaciones físicas complejas requieren que se trabaje con la velocidad y no con la posición de los objetos. Tratar de manipular directamente la posición de los objetos es muy difícil además que lleva a ecuaciones que no son lineales y cuya resolución numérica no es estable y viola los principios de conversación (energía, momento, etc). En general es mejor trabajar con la velocidad o con la aceleración (es decir generando fuerzas que luego se integraran en frames posteriores). La mayor parte de los motores de física trabajan con la velocidad o con la aceleración para manipular los objetos.

Impulse Resolution.

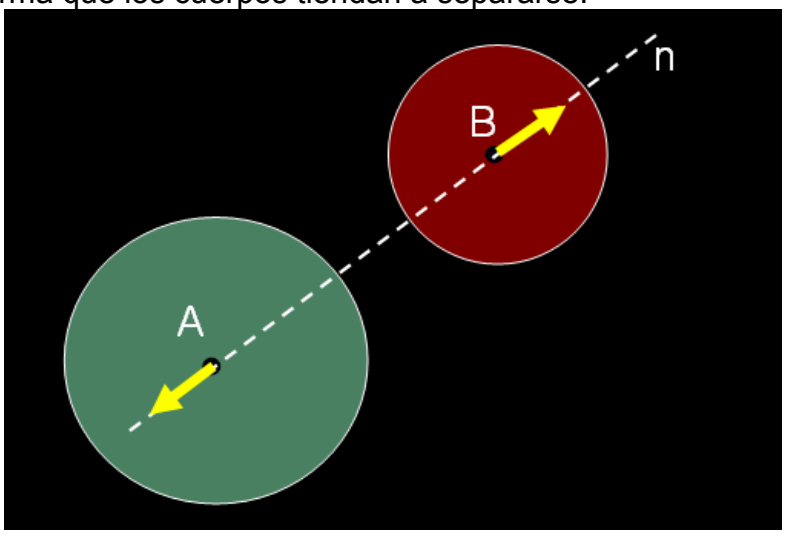
Queda claro (al menos intuitivamente) que si 2 cuerpos rígidos colisionan entre sí, se va a producir un choque y ambos saldrán "disparados" de tal forma de no quedar más en colisión, ya que dos cuerpos no pueden ocupar la misma posición en el espacio. También queda claro que el cuerpo con menos masa tiene que sentir mucho mas el efecto de la colisión que el cuerpo con más masa. Por ejemplo si una esfera de 1000k de masa colisiona frontalmente contra esfera de 1k, es de esperar que esta última salga disparada a toda velocidad mientras que la primera apenas note el impacto. Otro concepto que conocemos de nuestra experiencia diaria es que el impacto también depende de la velocidad (en el sentido de "vector de velocidad") y por ejemplo no es lo mismo un impacto frontal en la ruta, que un impacto con el auto de adelante. El impulse resolution tiene que tener en cuenta todos estos aspectos para que el resultado sea realista.

Supongamos los cuerpos A y B entran en contacto en un determinado frame. La etapa de detección de colisiones nos provee con la información del punto exacto de colisión, la penetración y la normal en el punto de impacto tal como se muestra en la figura:



Elementos de la colisión entre A y B.

El impulse resolution va a generar una impulso en la dirección de la normal a la colisión, y en sentido contrario a la velocidad "relativa" de cada uno de los cuerpos. El mismo impulso se va a aplicar a ambos cuerpos (es una consecuencia directa de la ley de acción y reacción de Newton), pero en sentidos opuestos, de tal forma que los cuerpos tiendan a separarse:



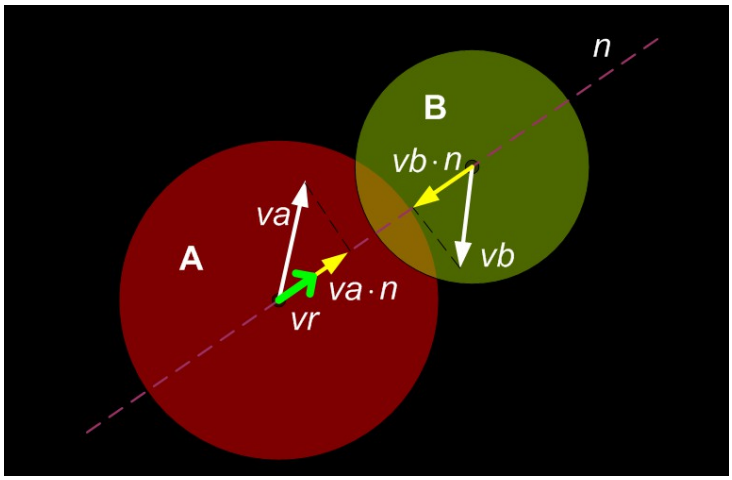
Impulse resolution. El mismo impulso se aplica a ambos cuerpos en sentidos opuestos.

Antes de seguir con la derivación de la "fórmula" del impuso, tenemos que definir un par de conceptos que vamos a utilizar para la misma: la velocidad relativa y el coeficiente de restitución.

Velocidad Relativa.

La velocidad relativa entre dos cuerpos A y B es una medida de cuan rápido se están acercando o alejando entre ellos. Para computar la velocidad relativa hay que tener en cuenta las componentes de las velocidades de los cuerpos sobre la línea que une sus centros de gravedad. Dicha línea representa la normal de la colisión y es la dirección que tendrá el impulso que resolverá la misma. Matemáticamente hay que proyectar los vectores de velocidad de ambos objetos sobre el vector normal de la colisión y luego hacer la resta vectorial.

$$vr = (\vec{vb} - \vec{va}) \cdot \vec{n}$$



Velocidad relativa entre A y B.

Coeficiente de Restitución

Imaginemos una pelota rebotando en el piso. Sabemos por la experiencia que en cada rebote la pelota alcanzará menos altura, hasta que eventualmente no rebotará más. Esto es debido a que parte de la energía cinética de la pelota se pierde en cada impacto (generando calor) hasta que eventualmente toda la energía es transformada en calor (una forma de energía que no produce trabajo).

Si es una pelota de goma es posible que tarde mucho tiempo antes de dejar de rebotar, mientras que una pelota de madera quizá rebote una sola vez. En el primer caso se dice que el cuerpo tiene un elevado coeficiente de restitución.

Pelota rebotando. En cada rebote se pierde parte de la energía y por eso la pelota alcanza cada vez menos altura.

El coeficiente de restitución es una medida del grado de elasticidad de un cuerpo, los cuerpos elásticos tienden a rebotar mucho mejor (sin perder velocidad luego del rebote) que los cuerpos que no son elásticos.

Matemáticamente el coeficiente de restitución (se suele escribir con la letra e) se expresa mediante la fórmula $\vec{v'} = -e\vec{v}$. El signo menos es debido a que la velocidad cambia de sentido luego del choque.

Cálculo del impulso.

Para corregir la colisión vamos a generar un impulso en la dirección normal n (que la conocemos, pues viene dada en los elementos de la colisión) y de una magnitud *j*, que no la conocemos y que tenemos que calcular.

El impulso es un cambio instantáneo en la velocidad y teniendo en cuenta que los cuerpos A y B tienen que separarse en los siguientes frames el cambio en sus velocidades tiene diferente signo (si es positivo en A, tiene que ser negativo para el B). Escribiendo eso nos queda:

$$\vec{v_a'} = \vec{v_a} + \frac{j}{m_a}\vec{n} \quad (1) \qquad \vec{v_b'} = \vec{v_b} - \frac{j}{m_b}\vec{n} \quad (2)$$

Notemos que el impulso en A y en B tienen:

- la misma dirección n, la normal de la colisión.
- sentido opuesto
- inversamente proporcional a las masas.

Restando las ecuaciones (1) y (2)

$$\vec{v_a'} - \vec{v_b'} = \vec{v_a} - \vec{v_b} + \frac{j}{m_b}\vec{n} + \frac{j}{m_b}\vec{n} \quad (3)$$

Teniendo en cuenta que la velocidad luego de la colisión tiene que cambiar de sentido (por el rebote) y aplicando el coeficiente de restitución que introducimos anteriormente:

$$\vec{v_a'} - \vec{v_b'} = -e(\vec{v_a} - \vec{v_b}) \quad (4)$$

Reemplazando (4) en (3) y operando :

$$-e(\vec{v_a} - \vec{v_b}) = (\vec{v_a} - \vec{v_b}) + (\frac{j}{m_a} + \frac{j}{m_b})\vec{n}$$

$$-(1+e)(\vec{v_a} - \vec{v_b}) = (\frac{j}{m_a} + \frac{j}{m_b})\vec{n} \quad (5)$$

multiplicando ambos miembros por $\vec{n}$ y recordando que $\vec{n}$ es un vector normalizado lo cual implica que $\vec{n} \cdot \vec{n} = 1$, tenemos que

$$-(1+e)(\vec{v_a} - \vec{v_b})\vec{n} = (\frac{j}{m_a} + \frac{j}{m_b}) \quad (6)$$

Despejando j obtenemos

$$j = \frac{-(1+e)(\vec{v_a} - \vec{v_b})\vec{n}}{(\frac{1}{m_a} + \frac{1}{m_b})} \quad (7)$$

Esta última ecuación permite calcular la magnitud del impulso lineal, muchas veces se la llama "linear collision response", pues solo tiene en cuenta la velocidad lineal de los cuerpos rígidos. En el caso de partículas funciona correctamente, pero en el caso de cuerpos rígidos, para obtener un mayor realismo es preciso tener en cuenta los efectos de la rotación. En este último caso hay que notar que los puntos que entran en contacto tienen además de la velocidad lineal del cuerpo rígido, dada por la velocidad del centro de masas, una velocidad tangencial, producto de que el cuerpo rígido esta girando sobre el eje que pasa por su centro de masas.

Resolviendo la ecuación con esos cambios llegamos a la fórmula del impulso con efecto angular :

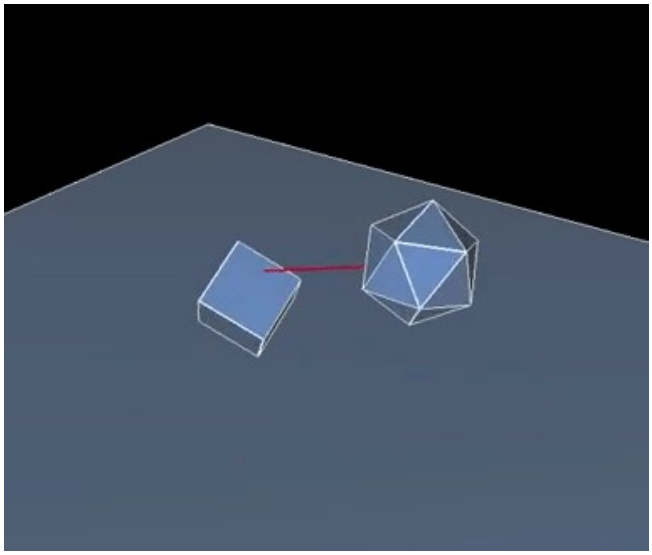
$$j = \frac{-(1+e)((V^A - V^B) * t)}{\frac{1}{mass^A} + \frac{1}{mass^B} + \frac{(r^A \times t)^2}{I^A} + \frac{(r^B \times t)^2}{I^B}}$$

Más allá de la complicación de la deducción de la fórmula, luego la implementación es muy sencilla, como suele suceder muchas veces en física. Y en unas pocas líneas de código se puede computar un cambio preciso en la velocidad que resuelva la colisión.

6- Constraints.

Introducción.

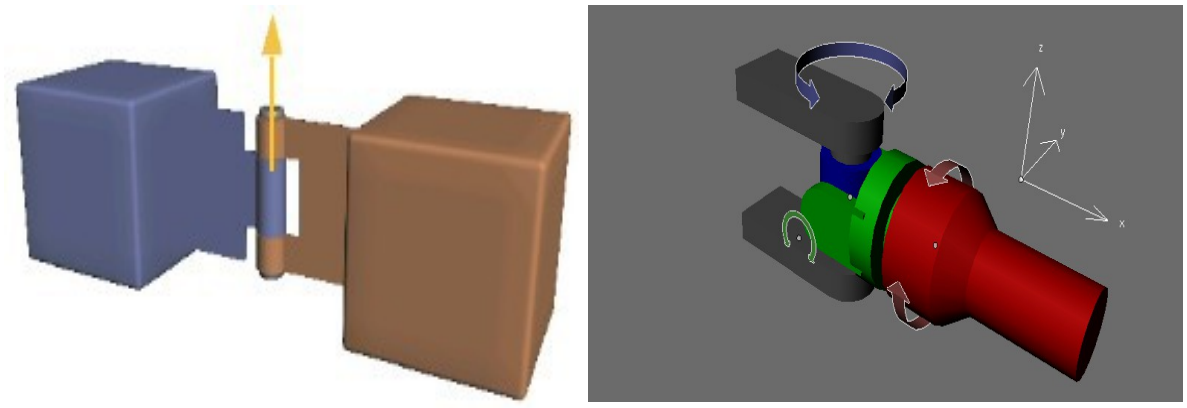
En muchas situaciones de la vida real los objetos físicos no se pueden mover libremente debido a que existen cierto tipo de restricciones que limitan los grados de libertad del sistema. Por ejemplo 2 cuerpos unidos por una soga. La existencia de la soga limita la capacidad de los objetos de separarse más allá de longitud de la soga que los mantiene unidos.



Ejemplo de constraint: dos cuerpos unidos por una soga.

En 3 dimensiones los cuerpos tienen 6 grados de libertad (6 DOF) por sus siglas en inglés "degrees of fredom"), que representan su habilidad de traslación en cada uno de los ejes del espacio: X, Y y Z (3 DOF) más su capacidad de rotar sobre un eje arbitrario (3 DOF adicionales). Cuando existe un constraint, la capacidad de moverse se ve restringida y por el cuerpo tiene menos grados de libertad.

Desde el punto de vista de una simulación se pueden lograr diversos efectos interesantes al limitar los grados de libertad de los cuerpos rígidos. Los constraints se usan para simular cuerpos articulados entre sí como esqueletos, máquinas simples y ragdolls, para simulación de tejidos y ropas, entre otros.



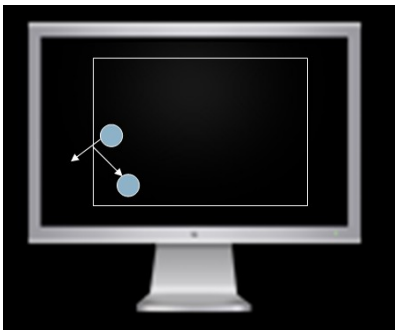
Se pueden usar constraints para simular articulaciones y así generar máquinas simples.

Tipos de Constraints.

Existen diversos tipos de constraints dependiendo la forma en que limitan los grados de libertad en el movimiento de los cuerpos rígidos. Vamos a estudiar los principales tipos de constraints usados en el contexto de videojuegos.

Boundary Constraints.

Supongamos una bola moviéndose libremente por la pantalla. Mientras este dentro de los límites de la misma la bola se puede mover libremente (en este caso en 2 dimensiones) por cualquiera de los 2 ejes X e Y.



Sin embargo, como no queremos que la bola se vaya de pantalla, una solución es chequear en cada paso si la posición en X o en Y de la bola esta fuera de los límites establecidos y en dicho caso hacerla "rebotar", cambiando la velocidad acordemente.

Por ejemplo si la posición X es menor a cero (el límite izquierdo de la pantalla) invertimos la componente X de velocidad , de tal forma que la bola salga rebotada.

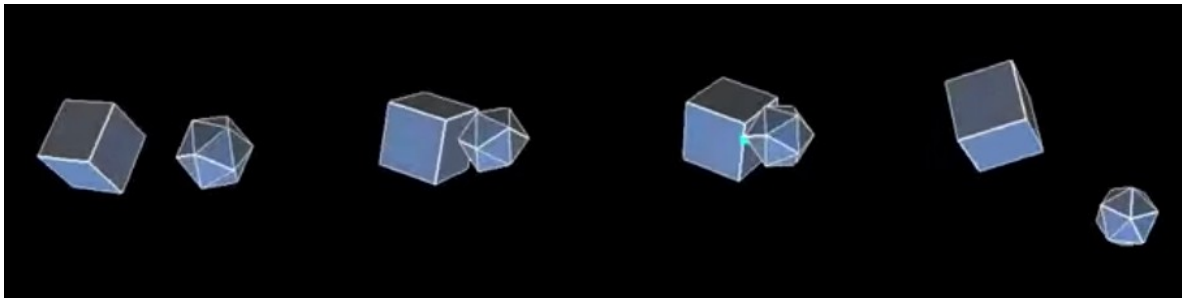
El pseudocódigo para hacer eso podría ser:

```
if (x < 0)
{
    x = 0
    if (velX < 0)
        velX = velX * -1
}
```

Este tipo simple de restricción se denomina boundary constraint o restricción de frontera. Se usa habitualmente en simulaciones muy simples para impedir que los objetos salgan de la pantalla o del área de simulación.

Penetration Constraints.

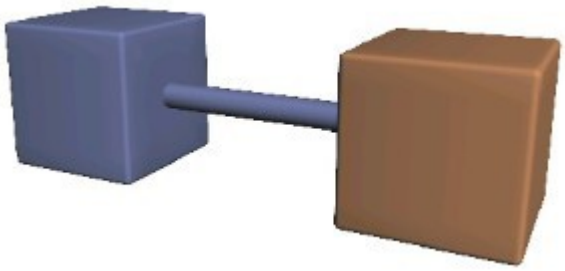
Otra restricción que conocemos es la que impide que dos cuerpos rígidos ocupen en el mismo lugar del espacio al mismo tiempo. Cuando dos cuerpos rígidos colisionan, como vimos en la etapa de collision response, el motor de física debe generar un impulso de tal forma que los objetos se separen entre si, para evitar la colisión.



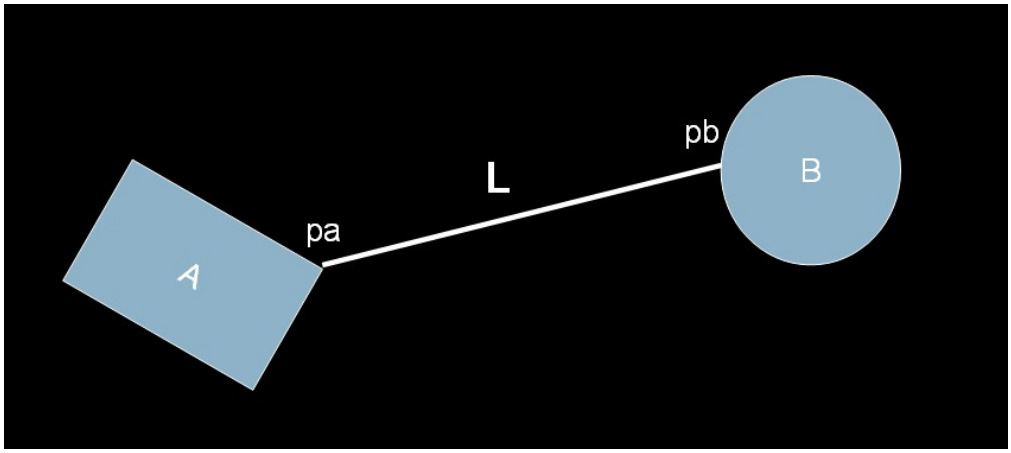
Distance Constraints.

Una de las más comunes restricciones que se pueden definir entre 2 cuerpos rígidos es la restricción de distancia. En este tipo de restricción dos puntos (uno de cada cuerpo rígido) llamados pivotes están limitados a permanecer a una distancia fija entre ellos.

Muchas veces los pivotes corresponden al centro de gravedad de ambos cuerpos. Podemos imaginar esta restricción como una barra ficticia sin masa y que no se puede estirar ni comprimir, y que esta unida a ambos cuerpos por los puntos pivotes.



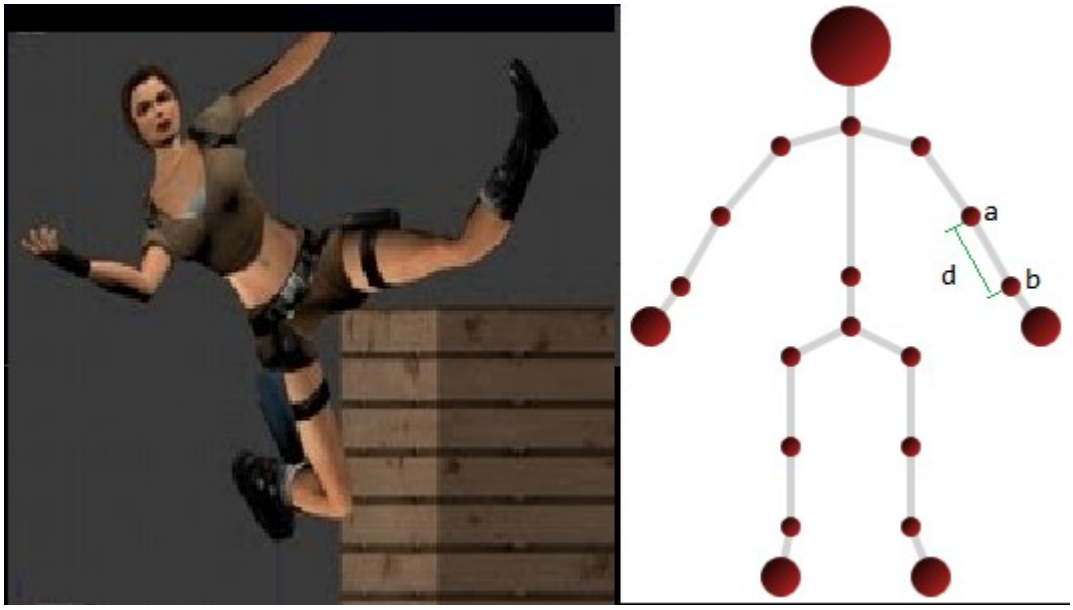
Una variación de esta restricción es permitir que la distancia sea inferior o igual a cierto valor. Es decir los cuerpos pueden estar tan cerca como se quieran, pero no se pueden alejar más allá de la cierta distancia. Podemos pensarla como si existiese una soga ficticia sin masa, y que no se puede estirar.



Elementos de una Restricción de distancia. Los pivotes pa, pb y la distancia L.

Existen muchos sistemas de cuerpos rígidos relacionados entre sí que se pueden simular con restricciones de distancia. Un ejemplo clásico es el péndulo.

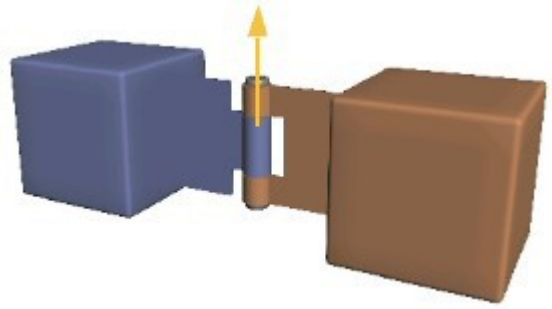
Las distintas partes de un esqueleto tipo "ragdoll" se pueden implementar usando distintas restricciones de distancia entre las articulaciones. Las partes del ragdoll se mantienen unidas durante la simulaciones gracias a las restricciones de distancia.



Ragdoll. Se puede implementar usando restricciones de distancia entre cada articulación.

Hinge Constraints.

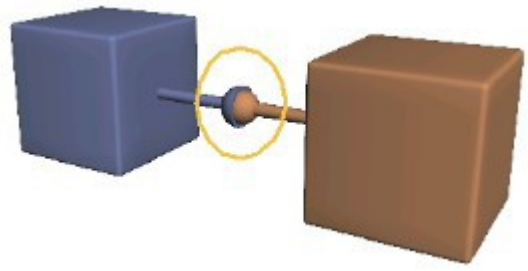
Este tipo de restricción permite que los cuerpos conectados giren libremente alrededor de un eje fijo que actúa de forma similar a una bisagra de una puerta.



Muchas veces puede haber restricciones adicionales como el ángulo máximo y mínimo que limitan aun más el movimiento de ambos cuerpos conectados.

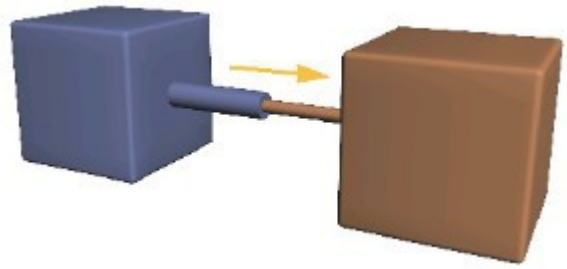
Ball-and-Socket Constraints.

Es similar al anterior, pero permite que los cuerpos giren libremente alrededor de un punto que actúa como pivote.



Slider Constraints.

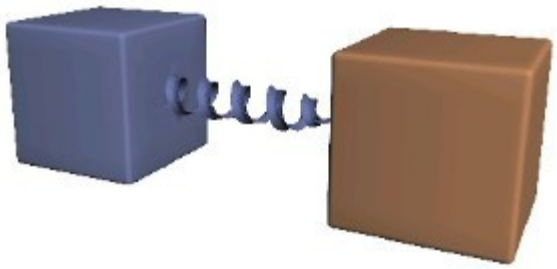
Permiten mover a los cuerpos conectados por un determinado eje, de forma similar a un pistón. Muchas veces se puede limitar también la distancia mínima y máxima de movimiento del conjunto.



Spring Constraints.

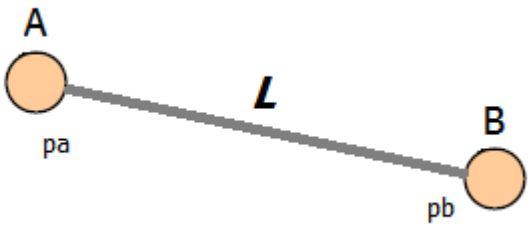
Los cuerpos se comportan como si estuviesen unidos por un resorte. Luego se pueden configurar las propiedades del resorte (coeficiente del resorte) entre

otros parámetros. Este tipo de constraint esta basado en la ley de Hooke para resortes. Junto con las restricciones de distancia es una de las más utilizadas en videjuegos.



Implementando Constraints.

Matemáticamente las restricciones se representan como una o varios ecuación (o inecuaciones) que limitan los grados de libertad del sistema. Por ejemplo una simple restricción de distancia entre 2 cuerpos rígidos puede expresarse con la siguiente ecuación.



$$\|\vec{p_a} - \vec{p_b}\| = L$$

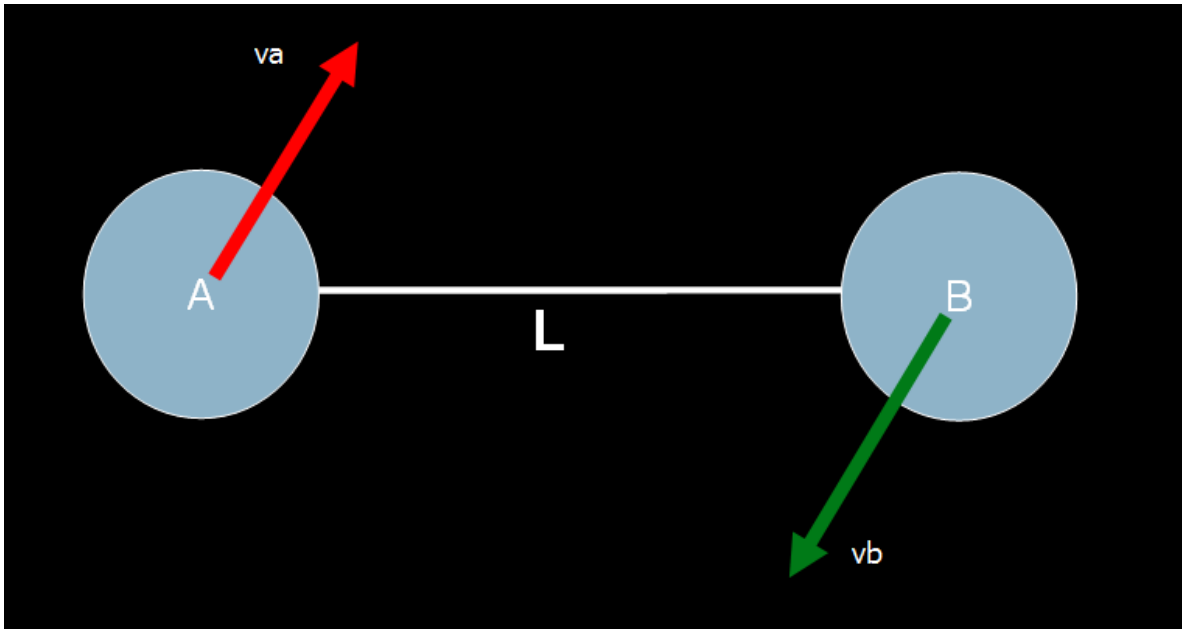
Que indica que la distancia el puntos pa (en el cuerpo a) y el punto pb (en el cuerpo b) tiene que ser exactamente igual a L.

Uno podría pensar en chequear la distancia entre los cuerpos como se hacía con las boundary constraint, y si no se cumple la ecuación de distancia, modificar las posiciones de los cuerpos rígidos para que la distancia se mantenga fija.

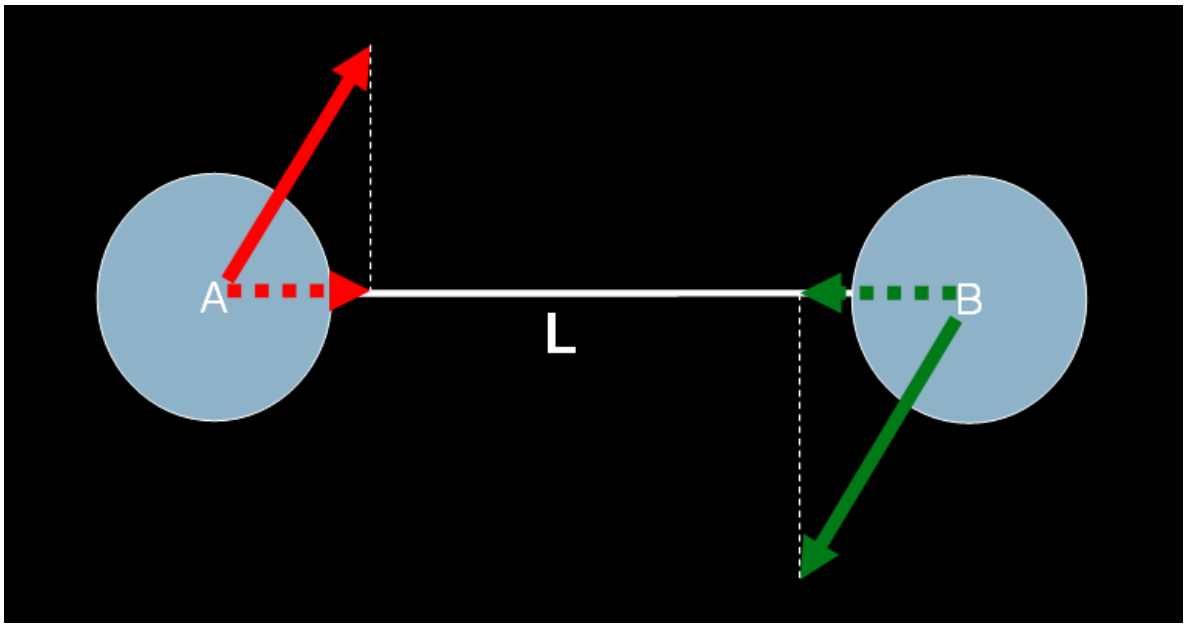
Sin embargo, hacer eso, repercute negativamente en la estabilidad de la simulación, al violar los principios físicos de conservación de la energía y del momento lineal. La mayor parte de los motores de físicas evitan a toda costa modificar directamente la posición de los cuerpos rígidos y trabajan con las velocidades o con las fuerzas.

La resolución matemática de una restricción de distancia es muy similar a la forma en que el motor de física trabaja con los colisiones en el etapa de collision response.

Imaginemos 2 los cuerpos A y B inicialmente están a exactamente a la distancia L cumpliendo con la restricción.



Sin embargo los cuerpos tienen cierta velocidad, que hará que en pasos siguientes de la simulación tiendan a acercase o separarse y la distancia no se mantendrá constante. Lo que nos interesa es la velocidad relativa de los cuerpos a lo largo de la línea de la restricción, ya que esa será la medida de cuanto se están acercando o alejando sobre los puntos de la restricción.



Algebraicamente computamos la velocidad relativa así:

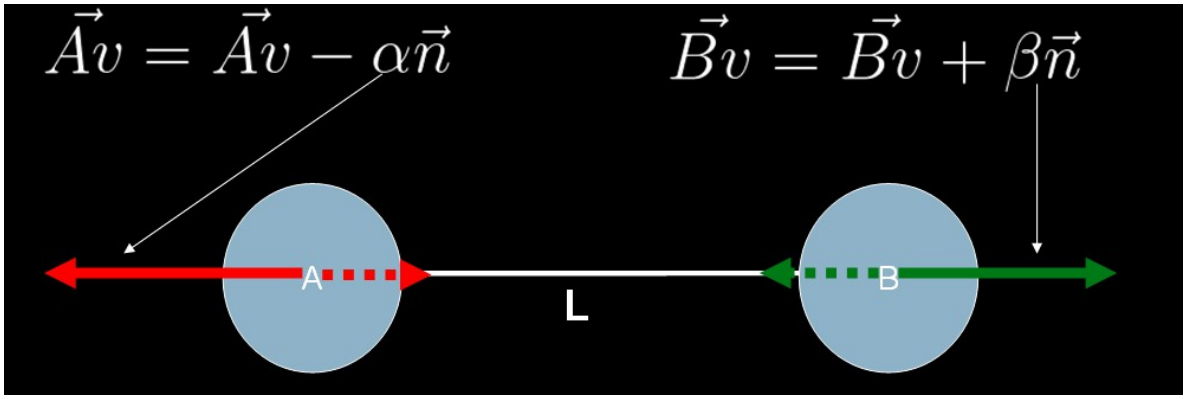
$$\vec{n} = \frac{\vec{b} - \vec{a}}{\left\| \vec{b} - \vec{a} \right\|}$$

$$\left.\begin{aligned} va &= \vec{A}v \cdot \vec{n} \\ vb &= \vec{B}v \cdot \vec{n} \end{aligned}\right\} \quad \vec{vr} = (va - vb)\vec{n}$$

Los vectores punteados (rojo y verde) representan la velocidad sobre la línea de la restricción, y como se ve en la figura anterior, de no tomar ninguna medida, los cuerpos A y B tenderán a acercarse en los siguientes pasos de la simulación.

Para corregir esta situación tenemos que aplicar un impulso en la dirección contraria de tal forma de compensar dichas velocidades para restablecer la distancia L.

Algebraicamente:



Nos queda calcular el parámetro alfa y beta que representa la intensidad aplicada a cada fuerza. Queda claro que esa intensidad debe depender de la velocidad del cuerpo y de su masa. Si el cuerpo se mueve muy rápido el parámetro deberá ser mayor para pode compensar la distancia rápidamente en los siguientes pasos. Existen distintos métodos para determinar exactamente que valor de alfa y beta utilizar, algunos más sofisticados (y costosos computacionalmente) que otros.

Una forma simple y rápida de determinar estos valores es usar la distancia actual L' en los cálculos. Lo ideal seria que la distancia siempre sea igual L, pero si no se cumple, los cuerpos estarán a una cierta distancia L' .

La diferencia en valor absoluto entre L y L' es una medida de "cuan mal" está la restricción. Cuanto mayor sea ese valor, menos se cumple la restricción y habrá que compensar con mayor intensidad. Una diferencia mínima significa que la restricción "casi" se cumple y hasta se puede llegar a ignorar.

La siguiente ecuación utiliza este concepto para determinar el valor de alfa:

$$\alpha = (va - vb - \frac{|L - L'|}{dt})(\frac{ma}{ma + mb})0.3$$

El 0.3 es un valor de diseño que representa que se va a corregir solo un 30% del error en la distancia. La práctica demuestra que no es conveniente intentar corregir el 100% del error en un solo paso, si no que conviene distribuirlo en varios pasos de simulación para que la misma se mas fluida y evitar así cambios bruscos de velocidad. Podemos interpretar esto como si fuese un resorte en lugar de una cuerda, que permite cierto estiramiento antes de regresar a su posición de equilibrio.