**Colisiones**

En el mundo real todos los objetos reaccionan e interactúan entre sí, ya sea al contacto o por medio de otras fuerzas: chocan, se deslizan, rebotan, se rompen, se adhieren y muchos otros comportamientos se presentan a cada interacción. Los videojuegos con la necesidad de mostrar comportamientos similares a los vistos en el mundo real utilizan diferentes tipos de métodos para emular o copiar estas conductas.

Con cada tecnología, llámese lenguaje o engine, la manera de solucionar el comportamiento de los objetos en colisión es muy diferente, pero en todo momento sigue los mismos principios.

Definición:

Colisión se refiere a detectar la intersección de dos o más objetos. Decimos que dos objetos colisionan cuando uno de ellos se sobrepone a otro en este momento debemos disparar una “señal” y tratar dicha colisión en consecuencia, impidiendo el movimiento si es un sólido, restando vida si es un enemigo, etc. En los videojuegos el objetivo es simular la física del mundo real, en tiempo real y con estabilidad, siempre y cuando la simulación resultante sea satisfactoria para los jugadores del juego.

**Métodos de colisiones**

Dependiendo el juego, existen muchos métodos de detección de colisiones. Se utilizan diferentes métodos ya que algunos requieren más requisitos y procesamientos que otros y por lo tanto unos son más precisos que otros. Por más que los juegos sean muy complejos, siempre se busca simplificar lo más posible para mejorar el rendimiento y las colisiones no son una excepción.

**Para juegos en 2 dimensiones:**

Los algoritmos para la detección de colisión en los juegos 2D dependen del tipo de formas que pueden colisionar (por ejemplo Rectángulo con Rectángulo, Rectángulo con Círculo, Círculo con Círculo, etc). En general, usted tendrá una forma genérica simple que cubre la entidad conocida como "hitbox".

La detección de colisiones en los juegos se lleva a cabo en dos pasos:

Primero se determina que par de figuras necesitan someterse a prueba de colisión y que método se va a usar. Se le conoce como etapa general (broad phase).

Luego se determinan los resultados de colisión para cada par identificado en el primer paso dependiendo el método necesario. Se lo conoce como etapa detallada (narrow phase).

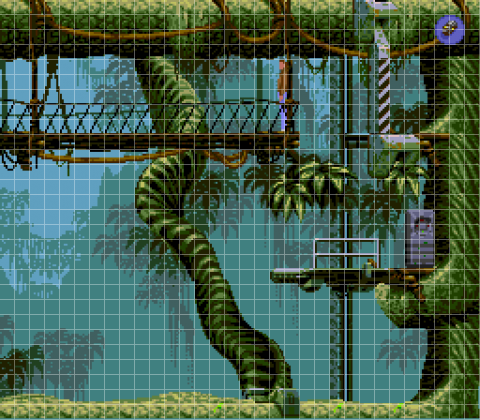
**Broad phase. Etapa general.**

Lo primero y principal es definir qué tipo de detección de colisiones vamos a utilizar para nuestro juego. Para ello tenemos que ver que tan preciso o realista debe ser la simulación que vamos a realizar. No es lo mismo un juego donde hay plataformas planas y pocos tipos de movimientos que un juego que tenga plataformas inclinadas o curvas y unas físicas más detalladas.

Hay diferentes categorías de juegos en 2D respecto a la detección de colisiones, a continuación se describirán sus detalles e implementación y también sus ventajas y desventajas. Hay 4 formas principales, de la más simple a la más compleja son:

**#1 Tile-based. Totalmente basado en celdas**

El movimiento del personaje se limita a las celdas (tiles), por lo que nunca puede estar a medio camino entre dos celdas. Las animaciones pueden ser utilizadas para crear la ilusión de movimiento suave, pero en lo que se refiere a la lógica del juego, el jugador siempre está en la parte superior de una celda específica. Esta es la forma más fácil de poner en práctica un juego de plataformas, pero impone fuertes restricciones sobre el control de personaje, lo que es inadecuado para los juegos de acción plataformeros. Es, sin embargo, popular con rompecabezas y juegos de plataformas "cinematográficas".



*Ejemplos de juegos que han usado este estilo: Prince of Persia, Toki Tori, Lode Runner, Flashback*

**Cómo funciona**

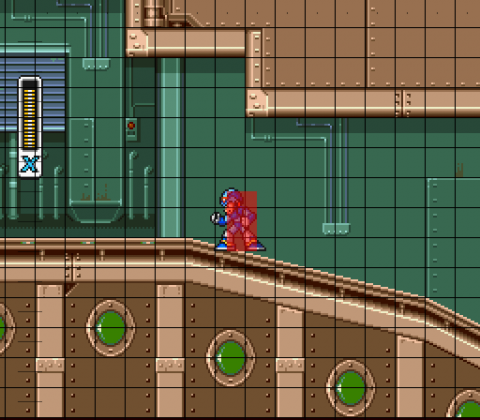
El mapa es una cuadrícula de celdas de igual tamaño, cada uno almacena información, como si se tratara de un obstáculo o no, qué imagen a utilizar, qué tipo de sonido de pasos usar, y así sucesivamente. El jugador y otros personajes están representados por un conjunto de uno o más celdas que se mueven juntas. En Lode Runner, por ejemplo, el jugador es una sola celda. En Toki Tori, el jugador es de 2 × 2 celdas. En Flashback, lo cual es inusual debido a las celdas de menor tamaño, el jugador es de dos celdas de ancho y cinco celdas de altura (ver imagen superior) al ponerse de pie, pero sólo tres azulejos de altura al agacharse.

En este tipo de juego, el jugador rara vez o prácticamente nunca se moverá diagonalmente, pero si lo hace, el movimiento se puede realizar en dos etapas separadas. Asimismo, es posible solo mover una celda a la vez, pero el movimiento multi-celda se puede hacer en forma de múltiples pasos de una celda si es necesario. Este tipo de juego solo permite movimientos de desplazamientos básicos, no se pueden hacer saltos en arco o diagonales.

Las ventajas de este sistema incluyen la simplicidad y precisión. Dado que los juegos son más deterministas, los fallos son mucho menos probables, y la experiencia de juego es más controlada, con menos necesidad de ajustar los valores en función de las circunstancias. La implementación de ciertas mecánicas se torna muy fácil, todo lo que tienes que hacer es comprobar si la celda del jugador y las celdas del escenario están alineadas en la forma específica que permita una acción determinada. En principio, este sistema no permite pasos de menos de una celda, pero que puede ser mitigado en varias formas diferentes. Por ejemplo, las celdas pueden ser un poco más pequeñas que el jugador (por ejemplo, un jugador es de 2 × 6 celdas).

**#2 Tile Based (Smooth). Parcialmente basado en celdas**

La Colisión todavía está determinada por un mapa de celdas (tilemap), pero los personajes se pueden mover libremente por el mundo. Esta es la forma más común implementada en las consolas de 8 bits y 16 bits, y sigue siendo popular hoy en día, ya que sigue siendo fácil de implementar y hace que la edición de niveles sea más simple que las técnicas más sofisticadas. Los mejores juegos de plataformas de acción de todos los tiempos se basan en este tipo.

****

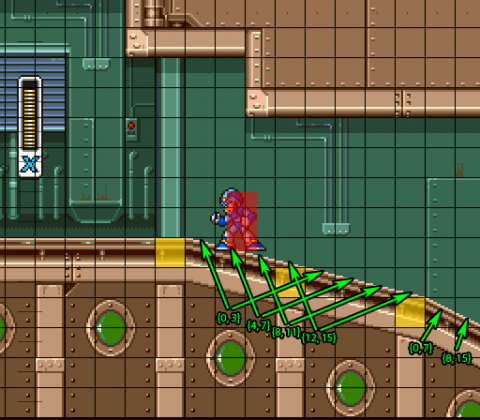
*Ejemplos: Sonic the Hedgehog, Mega Man, Super Metroid, Contra, Metal Slug, y prácticamente cualquier juego de la era de16-bit.*

**Cómo funciona**

La información del mapa se almacena en la misma forma que con la técnica anterior, la diferencia es en cómo los personajes interactúan con el escenario. Las cajas de colisiones del personaje es ahora un Axis-Aligned Bounding Box (AABB, es decir, un rectángulo que no se puede girar) más adelante se explican los tipos de hitbox. En muchos casos, el propio sprite del personaje es más grande que el hitbox lógico, ya que esto hace una experiencia visual más agradable y un juego más justo. En la imagen superior, se puede ver que el sprite de X es un rectángulo de dos celdas de ancho, pero su hitbox es rectangular con una sola celda de ancho.

En este tipo de juegos se permiten elementos muy comunes de los juegos 2D:

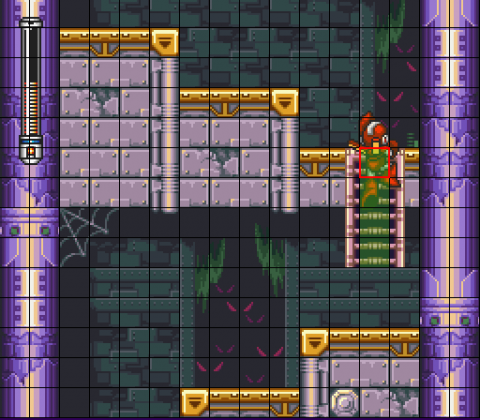
**Pendientes:**

****

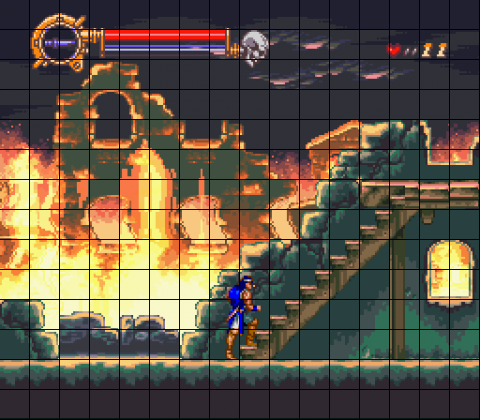
**Plataformas de un solo sentido**

****

**Escaleras verticales**

****

**Escalones**

****

**Otros tipos de interacciones especiales**

****

**#3 Bitmask. Máscara de bits**

Similar al anterior, pero en lugar de utilizar celdas de gran tamaño, se usa una imagen para determinar la colisión para cada píxel. Esto permite juegos más detallados, pero aumenta significativamente la complejidad, el uso de memoria, y requiere algo parecido a un editor de imágenes para crear niveles. También a menudo implica que las celdas no se utilizarán para los efectos visuales, y por lo tanto pueden requerir mucho trabajo, para cada nivel individual. Debido a estos problemas, se trata de una técnica relativamente poco común, pero puede producir resultados de mayor calidad que los enfoques basados en celdas. También es adecuado para entornos dinámicos, como los escenarios destructibles en Worms ya que se puede "dibujar" en la máscara de bits para cambiar el escenario.



*Ejemplos: Worms, Talbot’s Odysse.*

**Cómo funciona.**

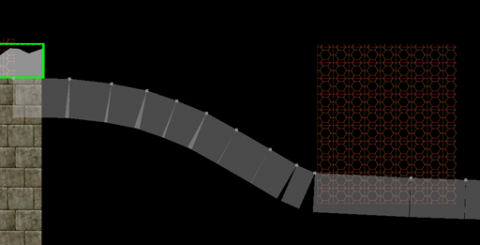
La idea básica es muy similar al algoritmo de las celdas, simplemente puede considerar cada píxel a ser una celda, aplicar el mismo algoritmo exacto, y todo va a funcionar. Las pendientes, escaleras, escalones y escenarios con particularidades especiales ahora no se programan de la misma manera que antes, en este caso tienen métodos mucho más complejos.



**#4 Vectorial**

Esta técnica utiliza datos vectoriales (líneas o polígonos) para determinar los límites de las zonas de colisión. Muy difícil de implementar correctamente, sin embargo, es cada vez más popular debido a la difusión de los motores de física, como Box2D, que son adecuados para la aplicación de esta técnica. Se ofrece beneficios similares a la técnica de máscara de bits, pero sin mayor sobrecarga de la memoria, y el uso de una manera muy diferente de los niveles de edición.





*Ejemplos: Braid, Limbo.*

**Cómo funciona**

Hay dos maneras generales de utilizar este método:

Resolver el movimiento y las colisiones manualmente, similar al método de máscara de bits, pero utilizando ángulos de polígonos para calcular la deflexión y obtener pendientes adecuadas.

O utilizar un motor de física (por ejemplo Box2D)

Obviamente, la segunda es más popular tanto porque es más fácil y porque le permite hacer muchas otras cosas con la física en el juego. Por desgracia, hay que tener mucho cuidado cuando se usa este método, para evitar que el juego se sienta como genérico, sin interés en la física de plataformas.

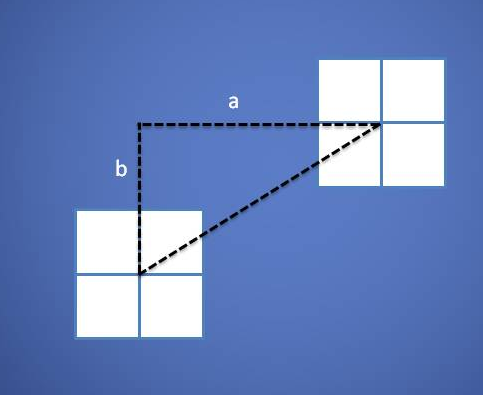
**Narrow phase. Etapa detallada.**

Se pueden detectar las colisiones de muchas formas, pero hay 3 que son los más comunes en los juegos en 2D:

**Bounding Areas- Colisión de área rectangular:**

El caso más común de este tipo y el más usado es mediante rectángulos determinados por coordenadas, que marcan el área de colisión. Es un método muy sencillo que depende del cuadrado natural que rodea a cualquier imagen en 2 dimensiones, comúnmente llamado “Bounding Box” (Cajas delimitadoras). Los rectángulos deben estar alineados para que funcione este método, o sea, no tiene que haber rotación.

La manera más sencilla para determinar si dos objetos se tocan, es si estos dos cuadrados definidos con posición y ancho lo hacen. Hay dos condiciones que tienen que cumplirse para definir una colisión. Siguiendo el siguiente diagrama:



* La distancia “a” es menor o igual a la suma de la mitad de la anchura (paralelo al eje X) de ambos objetos. ( En este caso particular la suma de estas cantidades es igual al acho total de uno de los objetos)
* La distancia “b” es menor o igual a la suma de la mitad de la altura (paralelo al eje Y) de ambos objetos.( En este caso particular la suma de estas cantidades es igual a la altura total de uno de los objetos

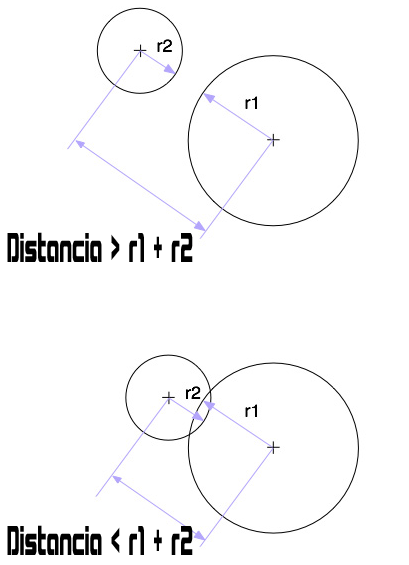
Parece bastante sencillo, pero el tener el registro de estas distancias en tiempo real puede ser algo muy complicado en un juego con muchos elementos en pantalla. Este método es la base para muchos otros que se han generado y parten de tener conocimiento claro de la geometría y la posición.

Ejemplo práctico:

<http://jsfiddle.net/knam8/>

**Bounding Areas- Colisión de área circular:**

Otra forma simple para la detección de colisiones es entre dos círculos. Este algoritmo funciona tomando los puntos centrales de los dos círculos y el radio de cada círculo. Si la distancia entre el punto central de los dos círculos es menor a la suma de los dos radios de los círculos, entonces hay colisión.



Hay muchos sprites a los cuales es más eficiente utilizar una colisión circular. Este método es muy útil si el sprite además de tener una forma circular, también cambia de ángulo.

Ejemplo práctico:

<http://jsfiddle.net/gQ3hD/2/>

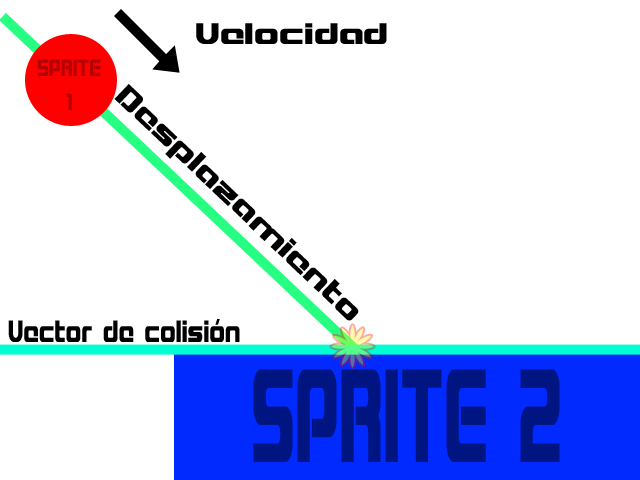
**Ventajas y desventajas de la colisión calculada por áreas:**

Ventajas: es la técnica más simple, requiere pocos cálculos matemáticos y por lo tanto utiliza pocos requisitos.

Desventajas: Al ser tan simple, la detección puede fallar si el área es muy pequeña o si se mueve demasiado rápido. En esos casos los sprites pueden atravesarse sin alcanzar a detectar una colisión. Para solucionar eso, se implementa el siguiente método.

**Ray casting – Vector de desplazamiento:**

Una técnica más fiable para sprites pequeños o sprites que se mueven muy rápido es de Ray casting. Este método detecta la intersección del vector de desplazamiento o trayectoria generado por los dos sprites. Cuando el vector del desplazamiento toca el vector que define el segundo sprite, hay colisión.



Hay dos conceptos importantes utilizados por este método: A priori o a posteriori.

Detección de colisiones a priori o a posteriori.

Puede detectar colisiones antes de que ocurran (a priori) o después (a posteriori). Si detecta colisiones antes de que ocurra, debe predecir dónde los sprites estarán en un futuro. Si detecta colisiones después de que ocurren, normalmente necesitará separar los sprites que han colisionado. Ninguno de las dos formas es notablemente mejor o más simple que el otro.

**Teorema de separación de ejes SAT (Separating Axis Theorem):**

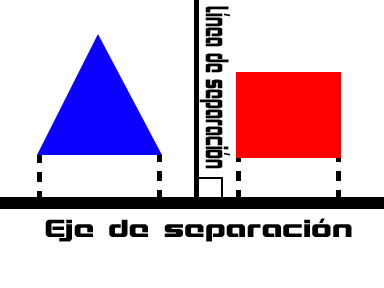
Este es un algoritmo de colisión que puede detectar una colisión entre dos polígonos convexos. Es más complicado de implementar que los métodos anteriores, pero es más eficiente y certero. Dependiendo la complejidad de los objetos, el rendimiento cambia y por eso se utilizan métodos para optimizar la detección de colisiones. Los algoritmos que utilizamos son de utilidad para cualquier juego que requiere una rápida detección de colisiones y proporciona más que un simple resultado booleano.

El teorema dice lo siguiente

"Dado dos formas convexas, no existe una línea en la que sus proyecciones serán separadas si y sólo si no se intersectan."

La línea donde las formas tienen proyecciones disjuntas se llaman **eje de separación** (separating axis).

Una **línea de separación** es la línea que puede trazarse entre las dos formas, sin tocar ninguna forma. Aquí se explica para juegos en 2D, pero se puede hacer en 3D también (en 3D, se llama el teorema plano de separación).



Funcionamiento del Teorema SAT:

El método SAT, como todos los de detección de colisiones, pone a prueba para ver si hay un espacio entre las dos formas, o si las dos formas se están tocando (es decir, colisionan o no).

SAT funciona así: imagine que tiene una luz, usted toma su linterna y alumbra las dos figuras. Primero, alumbra desde la izquierda. En el ejemplo se puede ver que la sombra generada en la “pared” no tiene ningún espacio o separación. En este caso puede haber colisión, pero todavía no se puede afirmar.



Si en el ejemplo anterior si hubiera una separación en la sombra generada, entonces se puede afirmar que las dos formas no se tocaban. Como en el caso anterior no hay una separación, entonces se tiene que hacer otra prueba para saber si hay colisión o no certeramente.



Cuando se alumbra por la parte superior, y mira en el suelo, se puede ver un lugar donde la luz golpea el suelo. Si las formas chocaban, usted no debería ver ninguna esa separación en el suelo. Por lo tanto, las formas no están colisionando.

¿Qué ocurre cuando hay rotación de objetos?

El método SAT tiene una manera sencilla de comprobar esto. Sólo tienes que comprobar dependiendo de la cantidad de lados que tengan las figuras. En el ejemplo anterior (con el triángulo y cuadrado), hay que hacer 3 (triángulo tiene 3 lados) mas 4 (cuadrado tiene 4 lados) = 7 comprobaciones. Para un hexágono y decágono, sería 6 (hexágono) 10 (decágono) = 16 comprobaciones. Esto puede sonar como mucho, pero hay que recordar que si alguna vez ve la luz entre los objetos, puede detener allí y asegurar de que no están colisionando.

**Ventajas y desventajas del SAT:**

Ventajas:

Es rápido: Utiliza matemáticas vectoriales bastante básicas y se puede descartar que haya colisión en tan solo una prueba, tan pronto como se detecta una separación. Hay eliminación de cálculos innecesarios.

Es preciso.

Desventajas:

Sólo funciona con polígonos convexos.

No se puede decir en qué parte especifica se están colisionando las figuras.

Ejemplo interactivo <http://www.sevenson.com.au/actionscript/sat/>

<http://www.metanetsoftware.com/technique/diagrams/tutA_demo.html>

Para mayor información del teorema SAT y colisiones que funcionan mediante la proyección de sus lados visitar:

<http://gamedevelopment.tutsplus.com/tutorials/collision-detection-using-the-separating-axis-theorem--gamedev-169>

<http://www.dyn4j.org/2010/01/sat/>

<http://www.metanetsoftware.com/technique/tutorialA.html>

**Pixel Perfect - Colisión calculada pixel por pixel:**

Los objetos ocupan un área rectangular, pero tienen una máscara (sprite de 1 bit) que define que píxeles son visibles. Primero se realiza una detección de colisión de área, luego, si hubo colisión, se realiza una detección pixel por pixel entre los pixeles superpuestos de la máscara de ambos objetos. Si existen dos pixeles superpuestos, y ambos son visibles, entonces hay una colisión.



Este método es el más preciso para juegos 2D, pero el más pesado de todos también. Este método no se usa si se requiere tener un juego bien optimizado.

**Juegos con gráficos en 3 dimensiones, que utilizan un sistema de colisiones y hit boxes en 2 dimensiones:**

Muchos juegos que tienen gráficos en 3D, siguen utilizando colisiones en 2D para optimizar el rendimiento y facilitar los códigos de programación. El 2D requiere muchísimo menos procesamiento matemático que el 3D. El punto negativo de esto, es que se pueden provocan múltiples bugs visuales y de jugabilidad.

League of Legends es un buen ejemplo de este sistema. Aunque es el juego con mayor cantidad de jugadores activos en todo el mundo, funciona con un engine propio muy obsoleto y con un rendimiento pobre en comparación con cualquier otro engine actual. El juego visualmente está hecho con modelos 3d, pero su programación está totalmente basada en 2D, tanto las colisiones, como las habilidades, impactos, etc.





Como la cámara siempre tiene un ángulo y una perspectiva que muestra todo desde arriba, se pensó que no era necesario programar un eje z. En varias ocasiones, vemos a nuestro personaje saltar, volar o atravesar paredes, pero en realidad sus colisiones solo están apareciendo o desapareciendo.

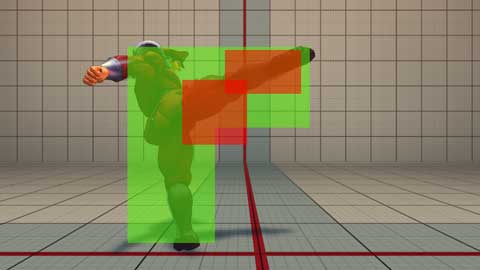
A causa de esta simpleza, se conocen un sinfín de bugs visuales que afectan el gameplay. Muchas veces ves que tu ataque golpea al adversario, pero el ataque pasa de largo y no le hace nada o lo contrario, ves esquivas una habilidad y recibes el impacto de todas formas. Lamentablemente, han pasado más de 4 años de la salida de League of Legends, pero si engine sigue siendo el mismo y los jugadores se han acostumbrado a jugar con estas fallas visuales y el mal rendimiento del engine.



Consejo si juegas League of Legends: siempre es más efectivo esquivar cualquier habilidad hacia arriba.

No necesariamente el juego tiene que ser visto desde arriba para poder usar este método. Esta técnica también es usada en juegos llamados “2.5D”,o sea, juegos hechos con gráficos en 3 dimensiones, pero con una cámara lateral, llamada side scrolling.

Por ejemplo, Street Fighter 4:



**Para juegos en 3 dimensiones:**

Las colisiones de los juegos en tres dimensiones cambian mucho dependiendo la necesidad, en un mismo juego puede haber colisiones muy complejas como también colisiones muy simples. Todo con el objetivo de optimizar lo más posible el rendimiento del juego.

**Bounding Volumes- Colisión calculada por volúmenes:**

Las colisiones en los juegos en 3D, se hacen con mallas 3D formadas por polígonos, que pueden ser simples cubos, como también pueden ser modelos muy detallados que simulen a la perfección el modelo original del personaje o estructura que lleve la colisión.

El primer ejemplo es de Counter Strike 1.6: es un juego que requiere colisiones precisas, pero al ser un juego antiguo no se podían usar muchos polígonos para sus colisiones a causa de no afectar negativamente el rendimiento. Por eso se optimizó lo más posible para que las colisiones sean básicas y no se afecte mucho el gameplay.



Por otro lado, en los juegos con el fin de optimizar más el rendimiento, no se le da la misma importancia a la colisión de un personaje que a la colisión del entorno, como así también hay partes del entorno que tienen una colisión más detallada y otras que tienen colisiones más simples o nulas por el hecho de que no interactúan con nada o interactúan muy poco.

Los juegos que simulan lo más posible la física real, son los que tienen las colisiones más complejas. El mejor ejemplo son los últimos juegos de Pro Evolution Soccer o FIFA, estos juegos usan la misma maya 3d de los objetos como colisión, así es perfectamente exacta y el contacto entre jugadores o la pelota es casi perfecto.

En otros juegos, incluso las telas de la ropa tienen sus propias colisiones para dar más realismo al juego.

**David Emiliano Palleres - Tecnicatura en desarrollo de video juegos – Universidad de Mendoza**