UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

Tecnicatura en diseño y programación de videojuegos

UNL VIRTUAL

Modelos y algoritmos para videojuegos I

Unidad 8

Docentes
Sebastián Rojas Fredini
Patricia Schapschuk

CONTENIDOS

8. CRASH!

8.1. Estás colisionando

En este preciso instante estás colisionando. ¿Contra qué? ¿Qué es una colisión? ¿Para qué sirve?

No, no nos estamos haciendo los graciosos. El mundo físico no tendría sentido si los objetos no pudiesen interactuar entre sí mediante el contacto.

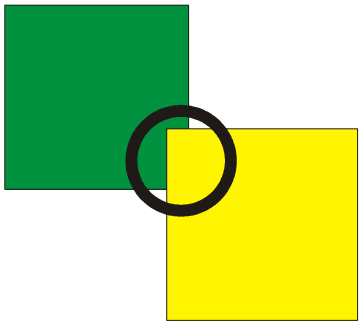
La computadora frente a la que estás sentado, por ejemplo, se encuentra en contacto con la mesa, que evita que la máquina se caiga al suelo. Lo mismo sucede con vos y tu silla. Existe un contacto entre ambos cuerpos, que son impenetrables y reaccionan ante ese contacto.

En esta unidad no entraremos en detalles sobre colisiones, contacto y otros conceptos que se verán en la segunda parte de la materia, pero sí veremos unos conceptos básicos para poder empezar a desarrollar nuestros videojuegos.

8.1.1. Colisión

En este campo de estudio utilizaremos la palabra colisión –por lo menos momentáneamente– como sinónimo de choque. Es decir, para referirnos al suceso según el cual dos objetos que se encuentran separados se acercan para ocupar el mismo espacio físico, lo cual es imposible, produciéndose así una colisión. Dicho de otro modo, una colisión es aquello que se produce cuando un objeto entra en contacto con el otro.

En el ámbito de los videojuegos, a diferencia de lo que ocurre en la realidad, es usual que dos objetos ocupen el mismo espacio. Sin embargo, el encuentro de sendos objetos también se denomina *colisión*, y ésta persistirá mientras algunas partes de los objetos sigan coincidiendo en el mismo espacio físico, es decir, mientras se encuentren encimadas.



En esta figura se aprecian dos objetos con la zona de colisión remarcada.

Ahora, la pregunta más obvia, y que seguramente todos ya se respondieron, es: ¿Para qué sirven las colisiones? Justamente para que los objetos interactúen entre sí en el mundo virtual.

Imaginemos un caballero de armadura oxidada que recorre los salones de un castillo, enfrentando los esbirros del caballero negro. En dichos corredores hay ítems con forma de manzanas que permiten que el caballero de armadura oxidada –llamado Lancelot- recupere un poco de su salud. Ahora ¿cómo sabemos si Lancelot se paró sobre una manzana o no? Sencillo: detectando colisiones.

Si en nuestro mundo el objeto que representa al caballero colisiona con el que representa a la manzana, significa que Lancelot juntó la manzana y debemos recuperar su salud.

Lancelot también puede saltar sobre la cabeza de sus enemigos y así derrotarlos. Aquí necesitamos otra vez hacer uso de las colisiones para saber si los objetos que representan a Lancelot y a un esbirro están colisionando.

Otro ejemplo claro de colisión de Sir Lancelot es la que se produce contra el piso. ¿Cómo sabemos que Lancelot se encuentra en el suelo? Sencillamente calculando las colisiones del objeto que representa a Lancelot y el objeto que representa al suelo.

Esto sucede todo el tiempo en los videojuegos que estamos acostumbrados a ver diariamente, aunque no de la misma manera.

Existen variadas alternativas a la hora de detectar colisiones. En general, éstas se ven diferenciadas por la figura de colisión que usan y el algoritmo de particionamiento espacial.

En los próximos parágrafos veremos un poco más en detalle en qué consisten ambas cosas.

8.1.2. Figuras de colección colisión

En la sección anterior dijimos que dos personajes o ítems en nuestro videojuego colisionaban cuando los objetos que los representaban se encimaban.

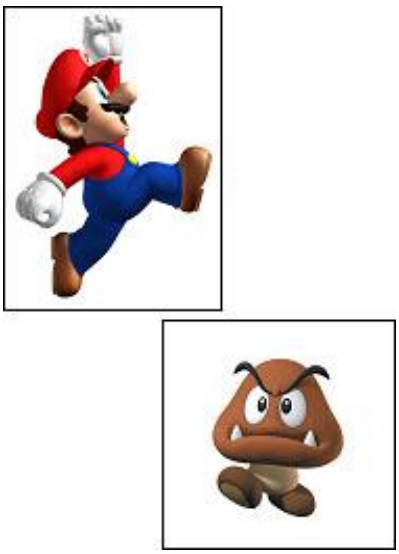
Ahora ¿por qué hablamos de objetos que los representan y no de los personajes mismos? La distinción surge de lo que se denomina *figura de colisión*. En esta materia limitaremos la discusión a colisiones en 2D.

Imaginemos el siguiente ejemplo: el famoso fontanero está a punto de caer sobre un indefenso goomba.

Como dijimos anteriormente, nos interesa saber cuándo ambos objetos colisionan para saber así cuando Mario lo pisó.

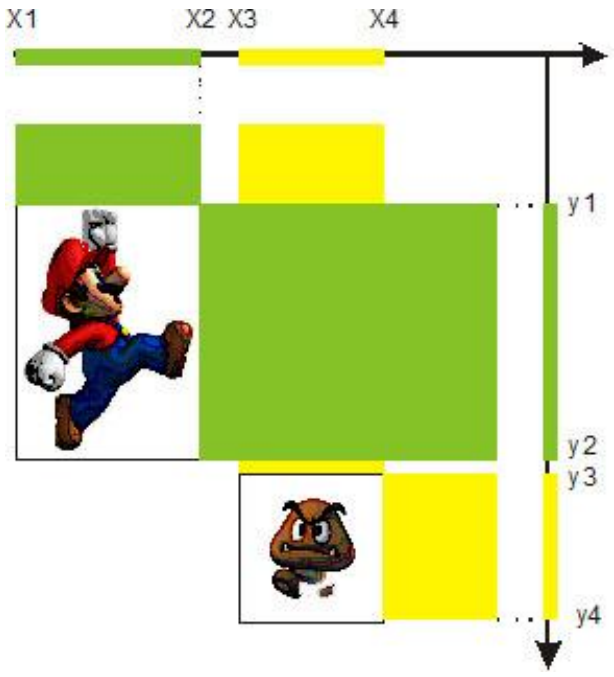
Ahora, la situación es la siguiente: ¿Cuáles son los objetos entre los que tenemos que calcular la colisión? Si bien Mario y el goomba no son rectangulares, para nosotros ambos objetos son imágenes y, por lo tanto, rectangulares.

De modo que, cuando programamos, nuestros objetos son los siguientes:



Nosotros conocemos el ancho y el alto de cada una de las imágenes. Es toda la información que tenemos a priori de la imagen para ver si dos sprites están colisionando. Entonces, en este caso, calcular la colisión se reduce a calcular la colisión entre dos rectángulos, lo cual es muy sencillo.

Veamos un primer caso:



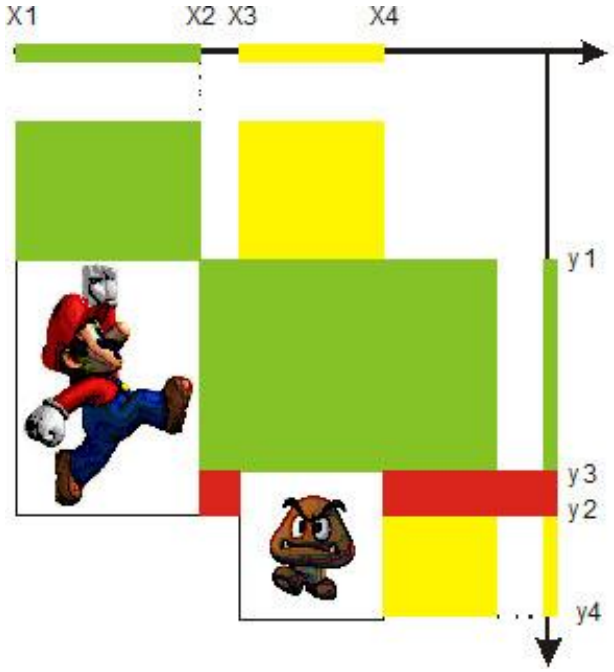
Lo primero que hacemos es descomponer las coordenadas de cada imagen en cada uno de los ejes. Es decir, la imagen de Mario empieza en la coordenada *x1* y termina en la coordenada *x2,* en el eje *x.*

Por otro lado, si miramos la coordenada, Mario ocupa desde *y*1 a *y*2. Noten que en este ejemplo utilizamos un eje *y* positivo hacia abajo.

Para el goomba sucede lo mismo. El mismo ocupa de *x*3 a *x*4 en *x* y desde *y*3 hasta *y*4 en *y*. Hasta ahora, sólo hemos proyectado las imágenes sobre el eje *x* e *y*.

En este primer caso no hay colisión y vemos que sus coordenadas no se solapan sobre ninguno de los dos ejes.

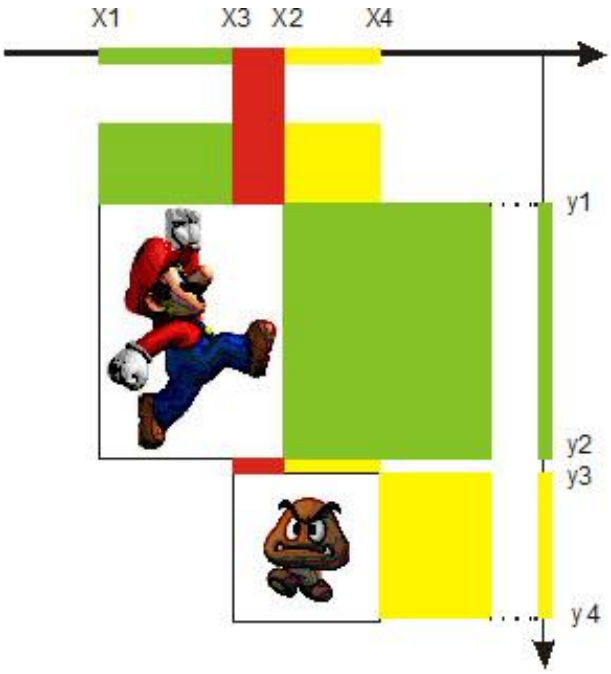
Veamos, ahora, un segundo caso:



Aquí, Mario se encuentra más abajo, por lo que vemos que en la descomposición sobre el eje *y* hay una intersección entre ambas imágenes.

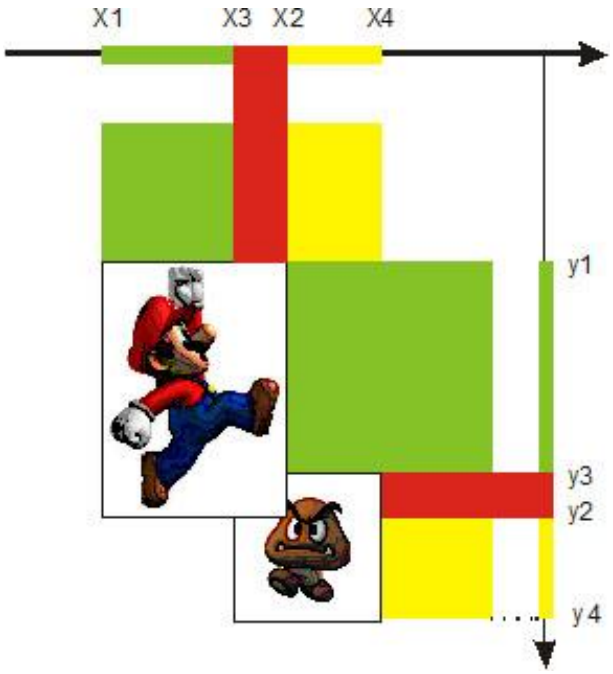
Es decir *y*3 < *y*2. Por otro lado, sobre el eje *x* no hay superposición y se sigue manteniendo que *x*2 < *x*3.

El siguiente caso es análogo al anterior:



Sólo hay solapamiento en las coordenadas *x* de ambas imágenes, en este caso *x*3 < *x*2. Sin embargo, tampoco hay colisión, como se puede apreciar, entre ambas imágenes.

Veamos ahora qué sucede cuando existe colisión:



En este caso, tanto en *y* como en *x* hay un solapamiento. Además, podemos observar que, efectivamente, las imágenes están colisionando. Es decir, la condición que tenemos aquí es que $y3<y2$ y que $x3<x2$.

Dicho de otra manera, si las imágenes se solapan en ambas coordenadas, las mismas estarán colisionando.

En el caso de nuestro ejemplo, las colisiones son analizadas siempre desde el mismo lado, pero esto también sucedería si las imágenes se encontrasen invertidas.

Por lo tanto, la condición, en general, para que dos imágenes se encuentren colisionando en cada uno de los ejes es:

En el eje *x*:

- xa_max= la coordenada mayor de la imagen A en x
- xa_min= la coordenada menor de la imagen A en x
- xa= la posición de la imagen A en x
- xb= la posición de la imagen B en x

Si $xa<xb$

Si $xa\_max > xb\_min$ -> Colisión en *x*

Si $xa>xb$

Si $xb\_max > xa\_min$ -> Colisión en *x*

Si $xa=xb$

Colisión en *x*

En el eje *y*:

- yb_max= la coordenada mayor de la imagen en y
- yb_min= la coordenada menor de la imagen en y
- ya= la posición de la imagen A en y
- yb= la posición de la imagen B en y

Si *ya*<*yb*

Si *ya*_max > *yb*_min -> Colisión en *y*

Si *ya*>*yb*

Si *yb*_max > *ya*_min -> Colisión en *y*

Si *ya*=*yb*

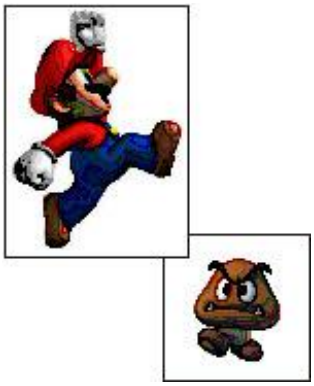
Colisión en *y*

Si luego existe colisión en ambos ejes, ambas imágenes se encontrarán colisionando. En este caso, la figura de colisión es la imagen misma. Esta clase de figura se suele denominar *bounding box.*

8.2. Otras figuras de colisión

En la sección anterior vimos cómo detectar colisiones usando la imagen como figura de colisión, a saber, dos rectángulos.

Esto, en la práctica, no siempre es bueno, ya que puede ocurrir que los personajes dibujados en las imágenes, a pesar de que las mismas estén encimadas, no se hayan solapado.



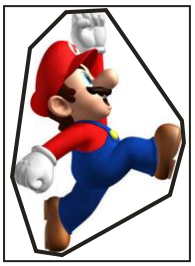
La figura ejemplifica esta situación. Si calculamos la colisión por los rectángulos de la imagen, podríamos pensar que Mario efectivamente pisó al Goomba. Sin embargo, es posible observar que no es así. Para estos casos, donde visualmente la diferencia es muy obvia, se utilizan otras figuras de colisión.

A continuación, conoceremos dos de las más comunes.

8.2.1. Polígono convexo

En este caso, sobre la imagen, se define a mano un polígono que será utilizado para realizar las colisiones. Aquí, por supuesto, cada imagen deberá contener información sobre su figura de colisión.

Por ejemplo, en el caso de Mario, podríamos definir el polígono de colisión de la siguiente manera:



Por lo tanto, ya no debemos calcular intersecciones entre rectángulos, sino sobre polígonos convexos, cambiando el algoritmo radicalmente. Esto, si bien es costoso desde el punto de vista computacional, es mucho más preciso y se ajusta mucho más a la imagen real del personaje que cuando usamos rectángulos.

8.2.2. Colisión a nivel de Píxel

Es la técnica más costosa en entornos 2D y en la práctica no suele presentar diferencias notables con respecto al uso de polígonos convexos. Consiste en utilizar la imagen como figura de colisión. Es decir, la colisión se realiza píxel a píxel. Por ejemplo, en el caso de Mario, si el píxel es blanco, no hay personaje; si el píxel es de otro color, significa que hay una parte del personaje, y por lo tanto, se puede colisionar contra ese píxel.

8.3. Técnica de colisión

Las figuras de colisión que vimos anteriormente no son mutuamente excluyentes, sino todo lo contrario.

Generalmente se utiliza una colisión a nivel de imagen, que nos da una idea burda sobre si las imágenes están en contacto o no. Si esta prueba de colisión da *falsa,* significa que las imágenes no están colisionando. En cambio, si la prueba de colisión de *bounding boxes* da *verdadera,* podemos conformarnos con eso o refinar un poco más la colisión.

Por ejemplo, podríamos probar si hay colisiones de los polígonos convexos de las imágenes, lo cual mejoraría mucho la detección. Dicha prueba sólo se realiza sobre las imágenes que posiblemente están colisionando, debido a que pasaron la primera prueba de colisión. De esta forma, sólo se realiza la prueba más costosa sobre las candidatas que tienen posibilidad de estar en contacto.

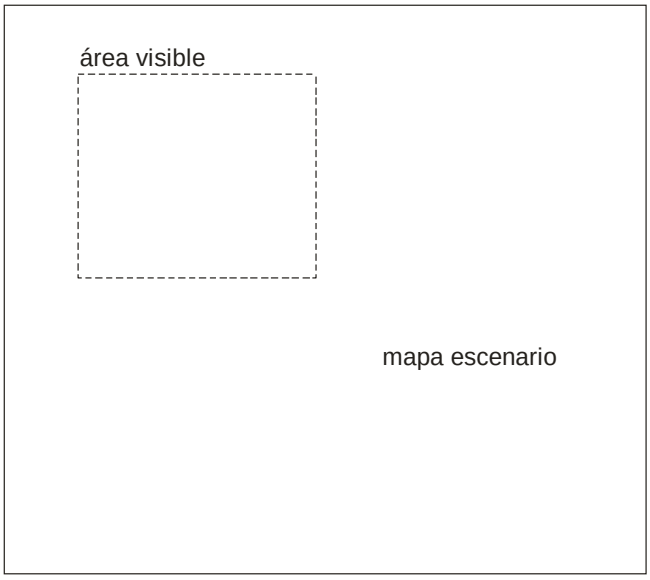
8.4. Algoritmos de particionamiento espacial

En el parágrafo anterior vimos de manera muy general cómo se detectan las colisiones en un videojuego.

No obstante, para saber si las imágenes se encuentran colisionando, no nos queda otra alternativa que probar las imágenes todas contra todas para comprobar si colisionan o no.

Esto no es conveniente en términos de rendimiento computacional, por lo que se desarrollaron algoritmos que permiten saber de antemano si un personaje puede llegar a colisionar con otro.

Veamos un ejemplo:



La línea punteada indica el área que podemos ver en la pantalla de todo el mapa de juego.

Supongamos que nuestro personaje se encuentra siempre dentro del área visible. Esto significa que sólo tiene oportunidad de colisionar con otros elementos que se encuentren dentro de esa área. De modo que no necesitamos evaluar colisiones contra objetos que no están en ese momento en pantalla, por más que sean parte del nivel.

Esta técnica ya presenta una mejora sustancial contra la forma *inocente* de encarar el problema, ya que reducimos considerablemente la cantidad de chequeos que debemos hacer.

La técnica mencionada anteriormente puede parecer una solución óptima, pero el problema es que no sabemos de antemano qué enemigos se encuentran en la zona visible, por lo que tenemos que resolver antes esta cuestión.

Para hacerlo, se utilizan también los algoritmos de particionamiento espacial, que no nos resuelve el problema, pero lo traslada hacia otro lado.

De manera que, para saber qué enemigos se encuentran en el área visible, debemos hacer una prueba de colisión entre el rectángulo que representa el área mencionada y los enemigos, lo cual implica hacer una prueba de colisión de contra todos. Si bien en la práctica el problema no se soluciona como se describió anteriormente, a través del ejemplo podemos darnos una idea de sobre qué se basan las técnicas de particionamiento espacial y para qué se utilizan. El objetivo es siempre reducir el espacio de búsqueda para colisiones, y en particular, esto se logra dividiendo el espacio en el cual existen las entidades.

A continuación veremos una de las técnicas más populares en los mundos bidimensionales.

8.4.1. Quadtrees

Veamos un ejemplo del videojuego *Zelda* para SNES.

Si tomamos una zona de juego como la mostrada en la imagen y si nuestro personaje se encuentra en la parte inferior de la pantalla, notaremos que es imposible que colisione con enemigos que se encuentran en la parte superior (si, por ejemplo, dividimos la pantalla en dos horizontalmente).

Lo mismo sucederá si dividimos la pantalla en dos verticalmente: si nuestro personaje se encuentra del lado izquierdo es imposible que colisione con enemigos que estén a la derecha (se excluyen aquellos que se encuentran en el medio).

De esta forma, podríamos dividir la pantalla en cuatro, como se muestra a continuación.

Con esta división, si nuestro personaje se encuentra en el cuadrado 1, sólo necesitaremos chequear colisiones con los enemigos que estén en el mismo cuadrado, ya que el resto, al encontrarse en zonas disjuntas, no puede estar en contacto con nuestro personaje.

Como vemos, en la zona 3 se encuentra la mayoría de los enemigos, por lo que el proceso de colisión será costoso allí. Por lo tanto, deberemos refinar un poco más la subdivisión espacial en esa zona Y para ello, volveremos a aplicar la misma lógica, subdividiendo el cuadrado 3 en 4 subregiones, de la siguiente manera:

De esta forma, si nuestro personaje se encuentra en la región 3.a, sólo necesitaremos chequear colisiones con los objetos de la región 3.a. Por lo tanto, procediendo de esta manera, la cantidad de colisiones que se necesiten evaluar será mucho menor que antes.

Este proceso se puede repetir recursivamente y realizar tantas subdivisiones como se deseen.

Generalmente, se utilizan árboles como estructuras de datos para representar dichas subidivisiones. De ahí el motivo del nombre Quadtree (Quad = cuatro; Tree = árbol).

En esta materia no entraremos en detalles sobre la implementación, la cual quedará reservada para cursos más avanzados.

BIBLIOGRAFÍA

Alonso, Marcelo; Finn, Edward. *Física: Vol. I: Mecánica*, Fondo Educativo Interamericano, 1970.

Altman, Silvia; Comparatone, Claudia; Kurzrok, Liliana. *Matemática/ Funciones*. Buenos Aires, Editorial Longseller, 2002.

Botto, Juan; González, Nélida; Muñoz, Juan C. *Fís Física*. Buenos Aires, Editorial tinta fresca, 2007.

Díaz Lozano, María Elina. *Elementos de Matemática*. Apuntes de matemática para las carreras de Tecnicaturas a distancia de UNL, Santa Fe, 2007.

Gettys, Edward; Séller, Frederick. J.; Skove, Malcolm. *Física Clásica y Moderna*. Madrid, McGraw-Hill Inc., 1991.

Lemarchand, Guillermo; Navas, Claudio; Negroti, Pablo; Rodriguez Usé, Ma. Gabriela; Vásquez, Stella M. *Física Activa*. Buenos Aires, Editorial Puerto de Palos, 2001.

Sears Francis; Zemansky, Mark; Young, Hugh; Freedman, Roger. *Física Universitaria*. Vol. 1, Addison Wesley Longman, 1998.

SFML Documentation, http://www.sfml-dev.org/documentation/

Stroustrup, Bjarne. The Design and Evolution of C++. Addison Wesley, Reading, MA. 1994.

Stroustrup, Bjarne. *TheC++ Programming Language.* Addison Wesley Longman, Reading, MA. 1997. 3° ed.

Wikipedia, http://www.wikipedia.org