

Más que SDL

La fabrica de éxitos

SDL suministra los componentes gráficos para construir un juego. Desafortunadamente, en la programación de juegos hay algo más que un motor gráfico. Este mes Steven Goodwin analizará algunas de las cosas que SDL no hace y demostrará como se pueden hacer.

POR STEVEN GOODWIN

SDL es un API (Application Program Interface) multimedia. La mayoría de sus funcionalidades se pueden llevar a cabo dentro del subsistema de gráficos, pero también tiene bibliotecas para sonido, palanca de juegos y hebras. Sin embargo no soporta la detección de colisiones. Esto no es una sorpresa ya que caería fuera de su alcance. En el campo de las 3D, OpenGL tampoco suministra una infraestructura para la detección de colisiones. Y escribir un juego sin detección de colisiones es como comer curry sin cerveza ¡A la postre es insatisfactorio!

Ojala que llueva café

Las colisiones son como el clima: No se trata de una sola cosa. El código de colisión para comprobar que el jugador no se da bruces con una pared es diferente del código del jugador dándose de bruces con un enemigo. Y eso es diferente de la colisión del jugador cogiendo una llave. Se cubrirán varios métodos de detección de la colisión, cómo funcionan, porqué ese método particular es pertinente, y donde ayudará a implementarlo SDL.

Moverse desde la posición A a la posición B no es tan simple como pudiera pensarse. Y desde luego que no es tan simple como cuando salió el primer

juego de plataformas *Manic Miner*, pero los principios básicos son los mismos. Se supone que el jugador comienza el nivel en un lugar seguro (libre de colisiones) y se requiere que permanezca en una posición segura al final de cada cuadro.

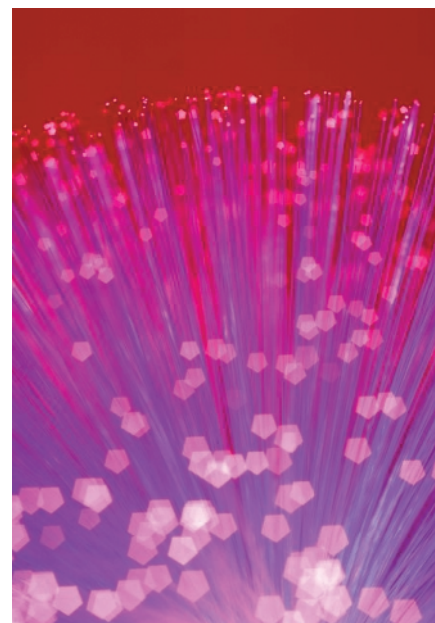


Figura 1: El jugador y su cuadro de límites.

Si no se puede encontrar una nueva posición segura, entonces hay que volver a la posición original (suponiendo que esta posición aun sea segura), o matar al jugador. Esto es una decisión de diseño del juego, no es una decisión técnica. Como los juegos son cada vez más complejos, con más y más objetos móviles, este problema ha llegado a ser más difícil. Cuando los juegos con autentica 3D llegaron a estar disponibles, esta complejidad aumentó en varias órdenes de magnitud. Incluso hoy, los juegos comerciales tienen que hacer trampas con sus sistemas de colisión para hacerlos jugables en un PC, o una consola, razonablemente modernos porque las matemáticas implicadas son muy complejas.

Cinco a uno

Para el sistema de colisión *en movimiento* se comprobará un pixel cada vez, buscando colisiones y reseteando a la ultima posición segura conocida si esto no es posible. Éste es el método que toman la mayoría de los juegos de este tipo.



Aunque se *comprobará* un pixel cada vez, se *moverán* varios pixels en el transcurso de un solo cuadro. Esto permitirá cambiar la velocidad de caminando a arrastrándose y moverse por el nivel de una forma mucho más realista.

Verificar varias posiciones en un cuadro puede parecer un derroche, pero es más fiable que verificar solamente que el punto final es valido (Véase Consideraciones del proyectil) y más flexible que escribir código específico para escu-

Consideraciones del proyectil

El problema en mover objetos varios pixeles en un cuadro se llama *consideraciones del proyectil*. Imagine una bala moviéndose muy rápidamente hacia la izquierda. Ahora imagine al jugador moviéndose muy rápidamente hacia la derecha. En el mundo real, la bala golpearía al jugador. En el juego, si solamente se considera la validez de la posición final ¡No se puede!

La bala podría terminar totalmente a la izquierda del jugador después de un solo cuadro, y no habría ocurrido ninguna colisión. Para evitar esto, se debe escribir un código especial que comprueba toda la trayectoria de la bala. Como a la mayoría de nosotros nos interesa más escribir juegos que ecuaciones matemáticas, se utilizara la fuerza bruta para comprobar cada punto a lo largo de la trayectoria.

driñar la ruta buscando obstáculos. La función que hace esto se llama *plyCheckCollisions*.

Para determinar si cualquier posición en particular es segura, debemos comparar la posición del jugador contra cada bloque existente para ver si se solapan. Para simplificar este problema, no se considerará los datos de la imagen del jugador, es decir su contorno exacto.

Miraremos solamente el cuadro que lo rodea, conocida como *bounding box* (cuadro de límites). Si esta cuadro choca con el cuadro contenedor de cualquier bloque, no puede moverse a esta posición.

Esta no es un compromiso tan infrecuente, puesto que cualquier problema en el código de colisión es tan molesto para el jugador que juega, como para el programador que lo programa. El jugador podría meterse en un hueco, podría ejecutarse una animación donde el personaje se rasca la nariz y después se encuentra que un pixel en particular está en colisión con el mundo y ya no puede escapar. Así que especificamos al artista que el gráfico del jugador debe permanecer dentro de un área fija, y no se tomará en consideración nada exterior del cuadro durante la detección de la colisión. Por la misma razón, mantenemos

un cuadro de límites de tamaño fijo a lo largo del juego.

La función para verificar las colisiones funciona de la siguiente manera: Comienza procesando el cuadro de límites del jugador actual y entonces solicita detalles de casi todos los objetos dentro del rectángulo (estamos usurpando la estructura *SDL_Rect*, aunque SDL en sí mismo no tiene soporte para

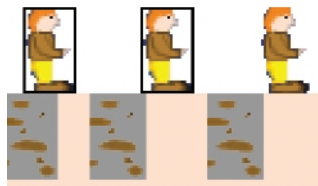


Figura 2: El cuadro de límites no refleja siempre los gráficos.

colisión). Dije *casi* porque el primer argumento para *exGetRectCollision* se utiliza como parámetro 'ignorado'. Esto es porque si se comprueba *todo* en el mundo contra el cuadro de límites del jugador siempre habrá como mínimo un objeto en en colisión: ¡El jugador! Como esto es inútil, se ignora explícitamente.

La *ListOfCollisions* ofrecerá más información acerca del estado de las colisiones: Cuantas colisiones ha habido, que objetos han sido afectados (bloques, enemigos, y así) e incluso la posición x, y donde ha ocurrido la colisión. Esta última información permitirá, por ejemplo, determinar si una flecha golpea en la cabeza o en la pierna. Entonces se puede usar esto para modificar el transcurso del juego. Teniendo una rutina de colisión de propósito general tal como esta, se puede enfocar el esfuerzo en otras áreas. Por el momento sólo esta-

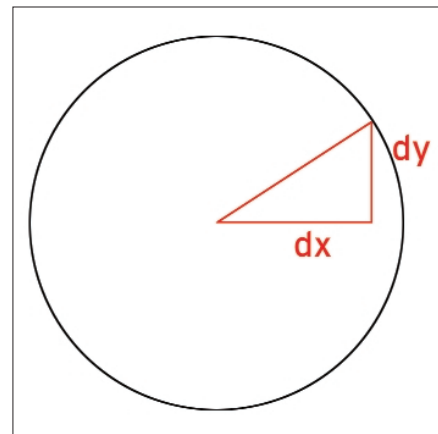


Figura 3: Comprobación de proximidad.

mos interesados en si la nueva posición es segura, o no.

Dentro de la función *exGetRectCollision* se puede optimizar el código "comprobar cada bloque en el mundo" para comprobar solamente los bloques con los que se pueden colisionar (una mejora en la velocidad que beneficiara a todo el juego; otra razón para usar un código de colisión de propósito general).

La función *exGetRectCollision* funciona de la misma forma que la función 'repair' que se vio el mes pasado. En esta etapa tampoco se hará caso de las colisiones enemigas, puesto que para esto se realizará un tipo de comprobación diferente y sería un desperdicio realizar esta prueba dos veces.

Si, por ejemplo, el personaje está saltando y colisiona con algo, no puede quedarse parado en medio del aire, hay que pensar en dejarlo caer. O si el personaje está saltando arriba y a la izquierda y choca ¿se debería intentar saltar recto hacia arriba, ignorando el movimiento hacia la izquierda o comenzar a caer? Todas estas son decisiones sobre el transcurso del juego y se implementa de forma muy simple utilizando la misma función *exGetRectCollision* para cada una de las posibles nuevas posiciones. La función *plyUpdJumping* muestra que decisiones he realizado para el componente de saltos.

La misma función *exGetRectCollision* se puede usar para otra parte del movimiento de colisión: la detección de un choque con el suelo. Se debe averiguar si el jugador está sobre algo sólido, o no. Para esto se puede usar la misma función, pero con un rectángulo de colisión distinto. Principalmente, una

Listado 1: ¿Choca o no?

```
01 SDL_Rect PlayersBoundingBox;
02 EX_COLLISION_LIST ListOfCollisions;
03 plyGetPlayerRect(pPlyObj, &PlayersBoundingBox);
04 exGetRectCollision(pPlyObj, &PlayersBoundingBox, &ListOfCollisions);
```

Listado 2: Desde *exCheckCollision* en *collision.c*

```
01 tx = pCollisionRc->x / TheGame.iTileWidth;
02 ty = pCollisionRc->y / TheGame.iTileHeight;
03 TestTileCollision(pCollisionRc, tx, ty, pCollisionList);
04 if (pCollisionRc->x%TheGame.iTileWidth)
05 TestTileCollision(pCollisionRc, tx+1, ty, pCollisionList);
06 if (pCollisionRc->y%TheGame.iTileHeight)
07 TestTileCollision(pCollisionRc, tx, ty+1, pCollisionList);
08 if ((pCollisionRc->x%TheGame.iTileWidth) && (pCollision
    Rc->y%TheGame.iTileHeight))
09 TestTileCollision(pCollisionRc, tx+1, ty+1, pCollisionList);
```

pequeña cuadro justo debajo de los pies, que es la anchura del jugador. Si hay una colisión, hay que alegrarse, ya que el jugador se encuentra seguro sobre algo.

Si no fuese así, habrá que cambiar al estado de caída y procesar la colisión de la manera normal como descenso del jugador. La detección de suelo resalta una limitación con el método de colisión por medio de el cuadro de límites, pues un pixel pudo chocar con una superficie, pero no lo hacen los gráficos, según se muestra en la Figura 2.

A la medianoche

Es mucho más fácil detectar qué objetos quiere coger el jugador. Aunque nuestra rutina de uso general de la colisión funcionaría no es siempre el mejor método, ya que es innecesario realizar detección de colisión. La recogida se acciona por proximidad, tan pronto como el jugador está dentro, por ejemplo, de la mitad del ancho de un bloque (o 16 pixeles) se puede recoger el objeto.

El método matemático para comprobar la distancia es el que usaba Pitágoras: “El cuadrado de la hipotenusa es igual a la suma de los cuadrados de los catetos”. Esto dejará que se realice una comprobación circular, buscando cualquier

Listado 3: Matemáticas de los días del colegio

```
01 dx = pObj1->x - pObj2->x;
02 dy = pObj1->y - pObj2->y;
03 iDistance = sqrt( dx*dx +
    dy*dy
04 );
05 if (iDistance < iProximity)
06 { /* ... coger está en el
    rango
07 ... */ }
```

Listado 4: Pitágoras hecho fácil.

```
01 dx = pObj1->x - pObj2->x;
02 dy = pObj1->y - pObj2->y;
03 if (iDistanceSquared <
04 iProximity*iProximity)
05 { /* ... coger está en el
    rango
06 ... */ }
```

Listado 5: Matriz de la superficie de colisión

```
01 if (SDL_LockSurface(pCurrentSurface) < 0)
02 {
03 fprintf(stderr, "No puede bloquear la superficie: %s\n",
    SDL_GetError());
04 return NULL;
05 }
06 pCollisionData = (Uint8 *)malloc(pCurrentSurface->w *
    pCurrentSurface-
07 >h);
08 /* pCurrSurface->pixels ahora apunta a datos de pixel validos para
    leer/escribir */
09 SDL_UnlockSurface(pCurrentSurface);
```

Listado 6: Consultar la paleta

```
01 if (iBytesPerPixel == 1)
02 {
03 pixel = *(Uint8 *)pSurfacePtr;
04 r = pCurrentSurface->format->palette->colors[pixel].r;
05 g = pCurrentSurface->format->palette->colors[pixel].g;
06 b = pCurrentSurface->format->palette->colors[pixel].b;
07 }
```

objeto dentro de un radio específico desde la posición actual del jugador.

Debido a que la raíz cuadrada es lenta y solo funciona en punto flotante y con números de doble precisión, habrá que ahorrar tiempo... evitándola. No se tiene verdadero interés en conocer la respuesta a la ecuación, sólo su resultado. ¿Está dentro del radio de proximidad? Se puede elevar al cuadrado cada lado de la ecuación obteniendo el mismo resultado.

Odisea espacial

En un principio, también se puede utilizar el cuadro de límites de colisión cuando se choca con el enemigo. Sin embargo, puesto que se puede tener una amplia gama de enemigos diferentes (y solo se dispone del tamaño de un bloque para almacenarlos) puede que sea un poco injusto si se representan con unos gráficos tan pequeños que podrían matar al jugador a una distancia de 30 pixeles.

De manera parecida, también es injusto si la animación provoca que el enemigo se encoja a la mitad de su tamaño original, pero todavía se considera la colisión del enemigo como cuando estaba a tamaño completo. Por lo tanto se va a necesitar una forma de

colisión que este basada en pixeles y eso va a necesitar que se investigue a fondo en SDL para determinar que pixeles son transparentes. Lamentablemente, no es tan simple como pudiera parecer.

Dulce perfección

La colisión basada en pixels en realidad es realmente fácil en teoría: Comprobar cada pixel de la imagen y si alguno de ellos es opaco (es decir, no es transparente) se ha encontrado una colisión. SDL, a pesar de toda su potencia, no proporciona una sencilla función GetPixel. De hecho, no proporciona ninguna función GetPixel. Así que habrá que escribir una.

El primer paso en el proceso es *bloquear* la superficie. El bloqueo es un método donde la imagen de la superficie (ya esté en memoria de vídeo o de sistema) se copia a un nuevo búfer en la memoria del sistema. Entonces se pueden leer, escribir o cambiar los pixels en este búfer, todo lo que se necesite y cuando se desbloquee la superficie, estos nuevos datos se copian nuevamente dentro de la superficie original.

Mientras la superficie esté bloqueada, no se podrán hacer operaciones de transferencia de bloques (blit) desde o hacia ella. Es el copiado de estos datos a y

desde la memoria lo que convierte el bloqueo de la superficie en una operación muy costosa, por tanto los píxeles no se deben escribir uno a uno. Si se está trabajando con algún algoritmo especial de mapa de bits (como fractales, por ejemplo) entonces se procesan un gran número de píxeles y se escriben todos juntos en la superficie bloqueada. Para la colisión (que es una circunstancia excepcional) se bloqueará la superficie una vez al comienzo del juego, se copiarán los datos necesarios dentro de lo que se llamará la superficie de colisión y después se desbloqueará la superficie inmediatamente.

La superficie de colisión será una simple matriz, donde un elemento en la matriz tendrá una equivalencia directa con un píxel en la imagen. Un valor de de cero en la matriz significará “sin colisión”, mientras que un uno significará “colisión”. Para simplificar, cada elemento en la matriz será un byte, aunque en el futuro se puede ahorrar memoria reduciendo esto a un solo bit.

Los datos de píxel (en `pCurrentSurface->pixels`) se almacenan en el mismo formato que en la superficie. Como ya se ha visto (en las partes 1 y 2 de esta serie), cada superficie debe crearse con un bit de profundidad diferente, así que se necesita comprender como leer diferentes formatos de píxel.

Aquí hay dos maneras. Antes de nada, hay que saber cuál es el formato del píxel:

```
iBytesPerPixel = 2
pCurrentSurface->
format->BytesPerPixel;
pSurfacePtr = (Uint8
```

Listado 9: Comparando color.

```
01 if (iBytesPerPixel == 2) pixel = *(Uint16
02 *)pSurfacePtr;
03 if (iBytesPerPixel == 3) pixel = *(Uint32
04 *)pSurfacePtr)&0xfffff;
05 if (iBytesPerPixel == 4) pixel = *(Uint32
06 *)pSurfacePtr;
07
08 if (SDL_MapRGB(pCurr->format, r, g, b) ==
09 pCurrentSurface->format->colorkey)
10 *pCollisionData = 0; /* este píxel es transparente */
11 else
12 *pCollisionData = 1; /* este píxel no lo es! */
13 pCollisionData++;
```

Listado 7: Normalizando

```
01 Uint32 pixel;
02 SDL_PixelFormat *fmt = pCurrentSurface->format;
03 pixel = *(Uint32 *)pSurfacePtr; /* Leyendo de una superficie
04 de 4-bytes-per-pixel */
05 r = ((pixel & fmt->Rmask) >> fmt->Rshift) << fmt->Rloss;
06 g = ((pixel & fmt->Gmask) >> fmt->Gshift) << fmt->Gloss;
07 b = ((pixel & fmt->Bmask) >> fmt->Bshift) << fmt->Bloss;
```

Listado 8: Leyendo bytes

```
01 if (iBytesPerPixel == 2) pixel = *(Uint16
02 *)pSurfacePtr;
03 if (iBytesPerPixel == 3) pixel = *(Uint32
04 *)pSurfacePtr)&0xfffff;
05 if (iBytesPerPixel == 4) pixel = *(Uint32
06 *)pSurfacePtr;
```

```
*)pCurrentSurface->
pixels;
```

Con un bit por píxel, se está usando una superficie paletizada y se tiene que consultar la paleta para conseguir los componentes rojo, verde y azul. Esos componentes están fácilmente disponibles en la superficie, como se muestra en el Listado 6.

Con los formatos de píxel empaquetado (2, 3 o 4 bytes por píxel) se tiene que realizar un poco de aritmética a nivel de bit para aislar los componentes individuales del RGB, y se *normalizan* de modo que cada uno esté dentro de la gama 0 a 255. El Listado 7 muestra este código.

La rutina de conversión del listado 7 es idéntica para todos los formatos empa-

quetados. La superficie tiene su propia combinación de máscaras, de cambios y de parámetros de pérdida para alcanzar esta unidad. El único código adicional (mostrado en el Listado 8) es necesario para leer el correcto número de bytes desde el puntero de la superficie.

Habiendo recogido los datos RGB, ahora se puede usar la función `SDL_MapRGB` para crear una superficie compatible en color y compararla con la clave de color, escribiendo el resultado en la matriz de colisión.

Después de leer un píxel, se debe mover sobre el siguiente. Aunque esto puede parecer como un bucle fácil, es aquí que la segunda manera toma la delantera. Es decir de la *tonalidad*, o *intervalo*, de la superficie.

Cantando canciones disparatadas

Cuando se crea una superficie, no es siempre del tamaño que se solicita. Las superficies en memoria de vídeo, por ejemplo, suelen ser una potencia de 2 (128, 256, 512) y por tanto la pantalla de 640x480 podría potencialmente ser de 1024x512 píxeles. Puesto que el tamaño real está bajo el control de la tarjeta gráfica hay muy poco que se pueda hacer sobre esto.

Al usar las funciones de transferencia de bloque estándares, SDL estimará automáticamente esto. Pero puesto que se he puentado SDL, se debe estar enterado de qué está sucediendo bajo el

capó, puesto que cada línea puede tener un suplemento de 384 píxeles. El valor `pCurrentSurface->pitch` mantiene el tamaño (en bytes) de cada línea de la superficie. Por lo tanto se debe aumentar el indicador de pixel como corresponde. Véase el Listado 10.

Nuestros datos de colisión (porque son nuestros, almacenados en nuestra memoria) no tienen el problema de tonalidad, y por tanto cada octeto se puede almacenar secuencialmente. Como referencia, la función se llama `exCUBuildCollisionSurface` y se encuentra dentro de `collision.c`.

Contonéalo (pero solo un poco)

Ahora hay que escribir la función `exCUBoundingBox2Pixel`, para determinar si hay píxeles letales enemigos dentro del cuadro de límites del jugador. Otra vez, la oferta de SDL no da soporte para esto, así que toca leer por completo todos los píxeles manualmente. Sin embargo, hay dos atajos que se pueden tomar. El primero es que se puede comprobar primero el cuadro de límites de cada objeto, antes de hacer otra cosa. Si éstos no se interceptan, entonces la prueba de colisión de pixel

magistral es una pérdida de tiempo, y se puede salir pronto de la función.

La segunda optimización es comprobar solamente el área de píxeles donde ambas cuadros de límites se solapan, es decir, el conjunto unión. SDL no proporciona ninguna función para procesar esta unión, pero se puede hacer bastante fácilmente considerando cada caso por separado: ¿El enemigo está encima o debajo del jugador? ¿El enemigo está a la izquierda o a la derecha del jugador?

Habiendo encontrado el área para comprobar de $(x1,y1)$ a $(x2,y2)$. Ahora se debe encontrar el primer pixel en nuestros datos de colisión que pertenezca a la posición $(x1,y1)$ (Listado 11) y después iterar a través de cada pixel uno por uno. El haber estipulado una anchura constante de 640 para los gráficos también tiene aquí una prima adicional. Puesto que nuestros datos de la colisión se relacionan directamente con los gráficos de la superficie, cada línea de datos de la colisión es siempre de 640 píxeles, lo que hace fácil moverse a partir de una línea a la siguiente.

El bucle de detección de colisión (Véase el Listado 12) es muy parecido al

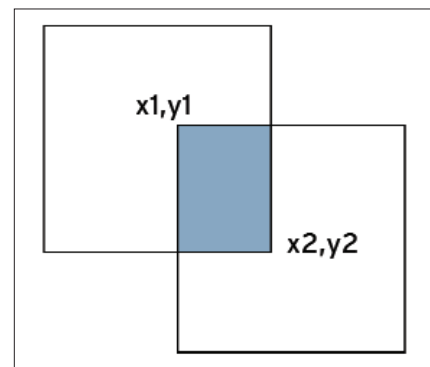


Figura 4: Un conjunto unión.

código para crear una superficie de colisión, mientras que el procesado de $x1, y1$ pide el código prestado del código `exDrawTile`; ambos deben ser fáciles de seguir. Véase el archivo `collision.c`.

Una solución alternativa es crear una superficie de 8 bits con `SDL_CreateRGBSurface` y escribir en ella los datos de colisión, quizás con una transferencia de bloque (blit), o una combinación de bloqueo-desbloqueo. Entonces, para ahorrar tiempo de procesador, se puede bloquear la superficie para toda la duración del juego (porque no se transferirá nada hacia o desde él). Siempre hay más de una solución a un problema. A modo de ejercicio intente [1] y vaya a la carpeta `sources/sdl`.

Listado 10: Incrementando el puntero de pixel.

```
01 pSurfacePtr = (Uint8 *)pCurrentSurface->pixels;
02 for(y=0; y<pCurrentSurface->h; y++)
03 {
04 for(x=0; x<pCurrentSurface->w; x++)
05 {
06 /* ... lee los datos del pixel como arriba .. */
07 pSurfacePtr += iBytesPerPixel;
08 }
09 /* Move onto the next line */
10 pSurfacePtr -= pCurrentSurface->w * iBytesPerPixel; /* rebobinando
11 al principio de la línea */
12 pSurfacePtr += pCurrentSurface->pitch; /* la tonalidad ya está en
bytes */
```

Listado 11: Encontrando x1 e y1

```
01 pColData = pCollisionData;
02 pColData += (iRegion%TheGame.iNumTileWidth) * TheGame.iTileWidth;
03 pColData += (iRegion/TheGame.iNumTileWidth) * TheGame.iTileHeight *
640;
04 pColData += x1;
05 pColData += y1 * 640;
```

Listado 12: Bucle de detección de colisión el puntero de pixel.

```
01 for(y=y1;y<y2;y++)
02 {
03 for(x=x1;x<x2;x++)
04 {
05 if (*pColData)
06 {
07 /* Colisión encontrada!!
08 Almacénela!!! */
09 return TRUE;
10 }
11 pColData++;
12 }
13 pColData -= (x2-x1);
14 pColData += 640;
15 }
```

RECURSOS

[1] <http://www.bluedust.com/pub/>