

Moviendo “Sprites” por la pantalla

Cash and Carry

En la primera parte de este tutorial vimos como cargar una imagen en una superficie, y como transferir un bloque (blit) sobre la pantalla. Este mes, mostraremos cómo esas operaciones básicas de carga y transferencia de bloque se pueden utilizar para componer la pantalla completa del juego.

POR STEVEN GOODWIN



Nuestra pantalla de título en el número 1 de Linux Magazine, publicada en la página 28, consistió en una superficie y una transferencia de bloque (blit). Pero con una superficie y varias transferencias de bloque, podemos hacer mucho más ¿Cómo? Bien, pues si tomamos una sola superficie y señalamos un área de 32 por 32 píxeles. A cada área la llamaremos *ladrillo*, o *región*. Podemos entonces transferir un bloque desde cada ladrillo a lugares específicos en la superficie de la pantalla para construir una imagen completa. Como podemos transferir en bloque varias veces desde un ladrillo (y a varios lugares diferentes), utilizaremos

menos memoria que en la pantalla de bienvenida, aunque cubriremos la misma área. La mayoría de las plataformas de juegos usan este método. Incluso en los juegos de estrategia en tiempo real (como Command & Conquer y Warcraft) usan este sistema, aunque sus gráficos sean isométricos en apariencia.

Al hacer el tamaño del ladrillo uniforme a través de todo el juego, nuestros gráficos pueden presentarse en lugares predecibles de la superficie de la imagen haciendo que el código de control sea entonces mucho más fácil de escribir. También estipularemos que cada superficie deba ser 640 píxeles de ancho, dando un número constante de ladrillos (20) en

cada línea. Esta es una decisión arbitraria para hacer el trabajo del artista más uniforme: mientras más opciones proponga, ¡mas sitio habrá para el error! El código para dibujar un solo ladrillo en la pantalla será entonces como el mostrado en el Listado 1.

Aunque cada ladrillo será alineado sobre un espacio limitado de 32x32,



Figura 1: Un grupo de bloques de muestra.



Figura 2: A la izquierda, *Explorer Dug* desprovisto de un alpha. A la derecha, con un alpha de 128.

nuestra función `exDrawTile` ha de ser escrita para dibujar ladrillos en cualquier posición arbitraria, permitiendo que sea reutilizada por el código renderizador para el jugador y su enemigo. Determinar que ladrillo se dibuja y en que lugar se colocará, es la responsabilidad de un bucle y una simple matriz de índice de ladrillos. Podemos hacerlo con una breve función como la del Listado 2.

Puesto que no tenemos ningún medio de estirar nuestros ladrillos, cualquiera de las cosas grandes en nuestros niveles (como la puerta de salida) tendrá que ser construida combinando varias ladrillos más pequeños. Por el contrario, cualquier imagen que sea más pequeña que 32x32 solo ocupará una porción del ladrillo y tendremos que utilizar píxeles transparentes para ocultar el resto.

Abriendo puertas

La transparencia se maneja con una técnica conocida como *afinado del color*. Por cada superficie que requiera trans-

parencia especificamos un color simple que, en lugar de ser transferido en bloque, será ignorado. No existe un color específico que tenga que usarse para este propósito. Tampoco debería haberlo. Puesto que todos los juegos son diferentes y cada gráfico en el juego utilizará un sistema de colores distinto, debemos elegir un color clave específico. Sin embargo, un color cuidadosamente elegido será, normalmente, suficiente para todos los gráficos del juego.

Las mejores opciones son los colores muy brillantes, extremos, los que, por lo general, nadie usaría. En *Explorer Dug* usaremos el verde brillante ¡Incluso el verde fluorescente! Con unos componentes RGB de 0, 255, 0 es muy improbable que alguien pueda ‘accidentalmente’ usarlo como parte de un gráfico auténtico, incluso para la hierba. Si los artistas quieren un verde brillante seguro que 0,254,0 es suficientemente bueno, con tal de que tengamos bastante resolución con la profundidad de color elegida (Véase el cuadro “Claridad de Color”). He utilizado siempre una clave de color verde intenso en mis juegos, y sin embargo nunca he tenido un problema.

Hemos elegido un color, ahora tenemos que decirle a la superficie cual es

```
SDL_SetColorKey(pGeneralTileGfx,
SDL_SRCCOLORKEY, SDL_MapRGB(U
pGeneralTileGfx->format,
0, 255, 0));
```

Esta función toma la superficie y le aplica una clave de color verde brillante. La clave de color tiene que ser dada en el mismo formato que la superficie y por tanto necesitamos usar la función `SDL_MapRGB` (la cuál pasa el formato de la superficie), para convertir los componentes de color RGB. Si estamos usando color de 8 bits (p.e. una superficie paletizada), entonces esta función buscará el resultado más cercano a los valores dados de RGB en su tabla. Especificando después la clave de color para la superficie, cada transferencia de bloque para esta superficie ignorará todos los píxeles de ese color, tratándolos como transparentes. Desde luego que puede cambiar la clave de color en cualquier punto durante su aplicación o desviarla completamente con:

Perdiendo el Juego

Para ahorrar espacio en la revista, las variables usadas en los recortes del código normalmente son abreviaturas del código fuente actual. Por ejemplo, notará que la variable global `theGame` no está presente en la mayoría de los listados que ofrecemos.

```
SDL_SetColorKey(pGeneralTileGfx,
0, 0, 0);
```

Esto posibilita reemplazar la clave de color para gráficos concretos (o incluso para regiones de ladrillos específicas) en el juego sin tener que crear una superficie separada. Desafortunadamente, SDL no facilita una forma fácil de retomar la clave de color actual, pero puede alcanzarla directamente en la estructura de la superficie para conseguirla como en el Listado 3.

El valor `old_colour_key` mantiene el color en un formato que la superficie pueda entender y así no necesita la función `SDL_MapRGB` cuando re-apliquemos la clave a la superficie.

Made in Japan

Además de hacer un solo color transparente, a veces queremos hacer transparente la superficie completa. No totalmente, pero si parcialmente. Esto dará a la imagen una apariencia fantasmagórica, de tal manera que podamos ver al personaje y al escenario detrás de él. Para hacer esto necesitamos aplicar un canal alpha. Los efectos especiales hacen un uso exhaustivo de canales alpha; explosiones, humo y lluvia usan

Listado 1: Dibujando un ladrillo

```
01 void exDrawTile(SDL_Surface
 *pTile, int iRegion, int x,
 int y)
02 {
03 SDL_Rect src, dest;
04 src.x = (iRegion %
 iNumTileWidth) * iTileWidth;
05 src.y = (iRegion /
 iNumTileWidth) * iTileHeight;
06 src.w = iTileWidth;
07 src.h = iTileHeight;
08 dest.x = x;
09 dest.y = y;
10 dest.w = iTileWidth; /* No son
 necesarios actualmente, ya que
 son ignorados */
11 dest.h = iTileHeight;
12 if (SDL_BlitSurface(pTile,
 &src, pScreen, &dest) < 0)
13 fprintf(stderr, "Blit error!
%s", SDL_GetError());
14 }
```

Listado 2: Bucle

```
01 int iNumTileHeight = 20,
 iNumTileHeight = 15;
02 /* rellenar una pantalla de
 640x480 */
03 int iTileWidth = 32,
 iTileHeight = 32;
04 for(ty=0; ty<iNumTileHeight;
 ty++)
05 for(tx=0; tx<iNumTileWidth;
 tx++)
06 exDrawTile(pTileSurface,
 map_data[tx + ty *
 iNumTileWidth], U
07 tx*iTileWidth ty*iTileHeight);
```

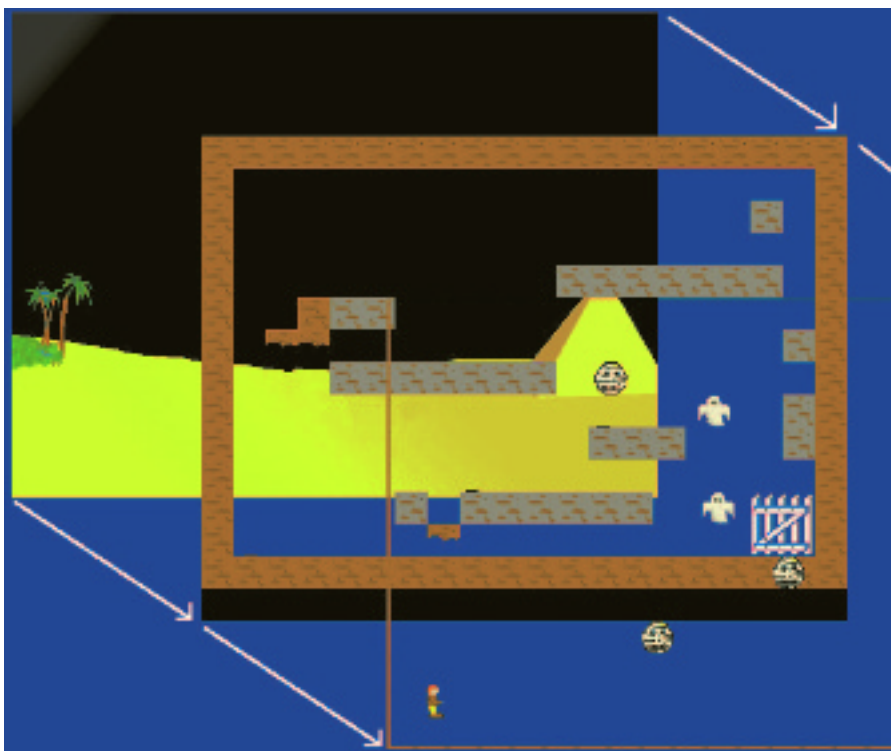


Figura 3: Apilado de las superficies.

varias texturas sobrepuestas, cada una con su propio componente alpha. Usaremos alpha para un propósito mucho más simple: desdibujar al jugador al comienzo del juego, para indicar que es imbatible. El componente alpha de una superficie se mantiene

Claridad de Color

Si la superficie a la que se ha aplicado la clave de color ha sido creada con `SDL_LoadBMP`, usará color de 24 bits. Esto significa que el componente verde tiene 8 bits de resolución, 254 y 255 deben ser colores diferentes y no hay problema. Por otra parte si la superficie solo tiene color de 16 bits, entonces es probable que el verde solo ocupe 5 o 6 bits. Esta pérdida de precisión significa que 254 y 255 son el mismo color puesto que ambos se representan internamente como $31 \ll 255 \gg 3 == 254 \gg 3$. Para estar absolutamente seguro hemos de verificar el formato de la superficie manualmente. Lamentablemente esta prueba sucede durante el juego, pero nuestros gráficos son creados fuera del juego donde esta información no está disponible. En realidad, no importa, siempre y cuando el componente verde de los colores reales no exceda 247 (suponiendo una precisión de 5 bits) estaremos sobre terreno seguro.

como un simple valor de 8 bits, que el rango de `SDL_ALPHA_TRANSPARENT (0)` a `SDL_ALPHA_OPAQUE (255)`. Por tanto cualquier valor entre estos dos es válido, aunque internamente 128 se trata como un caso especial y por eso se procesa más rápidamente. Es posible utilizar las claves de color y las superficies alpha al mismo tiempo.

Como con la clave de color, SDL no proporciona una forma de recuperar el valor alfa de una superficie sin referirse a la variable directamente, como en el Listado 4.

También es posible especificar una cantidad distinta de alpha para cada pixel individual, lo cual nos permitirá crear un borde mucho mas brumoso alrededor del personaje. De todos modos, esto es muy lioso, ya que la mayoría de los formatos de fichero no incluyen esta información. El formato XCF de GIMP lo hace, pero actualmente no hay muchas librerías que apoyen este formato. En vez de eso, tenemos que cargar dos imágenes, una contiene la información alpha y otra contiene la información gráfica (intensidad) y unificarlas manualmente mediante la lectura de los pixeles específicos de la superficie alpha y combinándolos con la superficie de intensidad. Esto, sin

embargo, excede nuestro propósito actual.

Ahora Todos Juntos

Ahora estamos listos para combinar las tres partes de nuestra pantalla de juego (telón de fondo, nivel de ladrillos y objetos dinámicos). Esto se puede hacer de un par de maneras. La más obvia, podemos dibujar cada una de las partes por turnos, como en el listado 5. No obstante, para mejorar la velocidad del juego, pondremos una condición adicional: El primer ladrillo (región 0) nunca será renderizado en el área de juego. Nunca. Esto es porque la mayor parte de este área está, de hecho, vacía. Se perdería un montón de tiempo de proceso de la CPU si la mayor parte del código de renderizado involucrado dibujara bloques completamente

Listado 3: Rescate de la clave de color

```
01 Uint32 old_colour_key;
02 old_colour_key =
   pGeneralTileGfx->format->color
   key;
03
   SDL_SetColorKey(pGeneralTileGf
   x,
04 SDL_SRCCOLORKEY,
05
   SDL_MapRGB(pGeneralTileGfx->fo
   rmat, 0, 255, 0));
06 exDrawTile(pGeneralTileGfx,
   12, xpos, ypos);
07
   SDL_SetColorKey(pGeneralTileGf
   x,
08 SDL_SRCCOLORKEY,
09 old_colour_key);
```

Listado 4: Rescate del valor alpha

```
01 Uint8 old_alpha;
02 old_alpha =
   pPlayerGfx->format->alpha;
03 SDL_SetAlpha(pPlayerGfx,
   SDL_SRCALPHA, iAmountOfAlpha);
04 exDrawTile(pPlayerGfx, iFrame,
   xpos, ypos);
05 SDL_SetAlpha(pPlayerGfx,
   SDL_SRCALPHA, old_alpha);
```

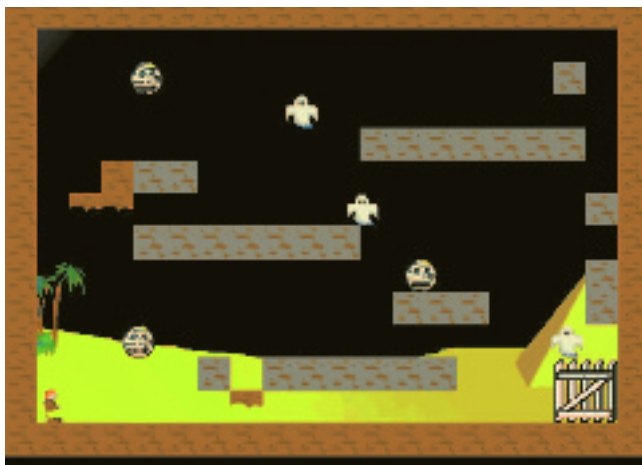


Figura 4: Completando la imagen del juego.

transparentes en la pantalla y, en los juegos, siempre nos alegramos de descubrir optimizaciones de velocidad. La manera más rápida para dibujar algo es no dibujarlo en absoluto, así que sacrificaremos esta minúscula cantidad de memoria al gran dios de la velocidad.

Podemos mejorar la velocidad más aun mediante la combinación del telón de fondo y los ladrillos en una única función de renderizado. Esto se puede hacer muy fácilmente diciendo que si queremos dibujar el primer ladrillo (el único completamente transparente), copiamos realmente (p. e.: transferir un bloque) una porción del telón de fondo sobre la pantalla. Por otra parte, dibujaremos un ladrillo *en vez del* telón de fondo. De este modo, en una pantalla con 300 ladrillos (20x15) dibujaremos 300 ladrillos con datos validos. Mientras que, en la propuesta original, habríamos dibujado 600. La cuestión de “es más rápido dibujar un ladrillo grande (el telón de fondo) que varios pequeños” se la dejaré a las pruebas de rendimiento. Hay muchas variables a considerar en este tipo de optimización y la única forma de saber que método se ejecutará más rápido es probar ambos. Durante el desarrollo encontré que la solución “varios pequeños” usaba menos tiempo de proceso (alrededor del 8% , según *top*) y por eso la adopte.

Usando esta idea, ahora podremos escribir una rutina especial para dibujar un ladrillo plano, que debería (o no debería) copiar la superficie del telón de fondo en lugar de un ladrillo (Véase el Listado 6).

Si hay ladrillos que impliquen transparencias también necesitaremos copiar el telón de fondo antes de dibujarlos. Determinar que ladrillos específicos tienen píxeles transparentes no es difícil, pero es lento. Así que podemos comprobar cualquiera de ellos cuando comienza el juego, o indicarle al artista que solo los ladrillos con 8 o superior (por ejemplo) pueden tener transparencias. Elegiremos la segunda opción.

Borrando y rebobinando

Ahora somos capaces de dibujar la pantalla una vez; necesitamos ser capaces de dibujarla varias veces. Lo cual nos permitirá animar algunos enemigos moviéndose en la parte superior. El código para esto es muy fácil, pero ha de estar hecho correctamente o tendremos desagradables chismes en la pantalla, o una carga innecesaria para la CPU (Véase el Listado 7).

Por “dinámica” realmente queremos decir dos cosas:

- Objetos que se mueven
- Objetos que permanecen quietos, pero animados

Componer la pantalla implica eliminar cada objeto dinámico de ella, así que nos hemos quedado con el fondo estático otra vez. Esto se puede hacer de una cuantas formas distintas.

Primeramente, podríamos interrogar a cada objeto dinámico para recordar a que se parecía la pantalla antes de haberla dibujado y entonces redibujar ese bloque cuando solicitemos una reposición. A pesar de ser aceptable, esto implica un montón de trabajo en la parte de cada objeto y no estamos considerando que dos (o más) objetos puedan superponerse. Aun así, éste puede ser seguramente el método más eficiente de reposición.

Además podríamos llamar a *exDrawStatic* otra vez. Esto podría funcionar, pero será muy lento. Sin embargo, si la imagen del telón de fondo alguna vez estuvo animada necesitaríamos implementar esta solución.

Listado 5: Dibujando la pantalla del juego

```
01 void exDrawStatic(void)
02 {
03     int tx, ty;
04     int obj;
05     /* Telon de fondo */
06     SDL_BlitSurface(pBackdrop,
07                     NULL, pScreen, NULL);
07     /* Ladrillos */
08
09     for(ty=0;ty<iNumTileHeight;ty++)
10
11     for(tx=0;tx<iNumTileWidth;tx++)
12
13     exDrawTile(pGeneralTileGfx,
14               map_data[tx + ty *
15               iNumTileWidth], tx, ty);
16     /* Objetos dinámicos- por
17     ejemplo, enemigos */
18
19     for(obj=0;obj<iNumEnemies;obj++)
20
21     exDrawTile(pEnemyTileGfx,
22               Enemy[obj].Graphic,
23               Enemy[obj].xpos,
24               Enemy[obj].ypos);
25 }
```

Listado 6: Dibujado de un ladrillo plano

```
01 void exDrawLevelTile(int tx,
02                      int ty)
03 {
04     SDL_Rect src;
05     int iRegion;
06     iRegion = exGetTileRegion(tx,
07                               ty);
08     src.x = tx * iTileWidth;
09     src.y = ty * iTileHeight;
10     src.w = iTileWidth;
11     src.h = iTileHeight;
12     if (iRegion == 0 && pBackdrop)
13     SDL_BlitSurface(pBackdrop,
14                     &src, pScreen, &src);
15     if (iRegion)
16     exDrawTile(pGeneralTileGfx,
17               iRegion, src.x,U
18               src.y);
19 }
```

Doble Búfer

Bajo X11, podemos conseguir gráficos suaves porque nunca escribimos directamente en la pantalla: escribimos en un búfer separado y SDL transfiere los bloques a la pantalla. Sin embargo algunos controladores escriben directamente en la pantalla. Si estamos usando uno de estos controladores y no puede dibujarlo todo lo bastante rápido, lo vera “deshilachado”.

Esto ocurre porque se puede ver la mitad del último cuadro y la mitad del cuadro actual, al mismo tiempo. En esas situaciones puede emplear un doble búfer. Esta técnica conlleva dos búferes de pantalla, uno que dibuja sobre el monitor y otro que dibuja nuestro juego internamente. Esto significa que nunca se escribirá el mismo cuadro que está siendo mostrado.

Emplear doble búfer en SDL es muy fácil. Primero necesitamos ajustar el modo de vídeo:

```
pScreen = SDL_SetVideoMode(
    iWidth, iHeight, 16,
    SDL_HWSURFACE | SDL_DOUBLEBUF);
```

Esto añade dos búferes a la superficie de la pantalla, un búfer frontal y un búfer trasero. Podemos escribir en esta superficie como normal con la función `SDL_BlitSurface` y SDL solo escribirá dentro de un búfer: en el trasero. Entonces, en lugar de actualizar la pantalla con `UpdateRect`, intercambiamos los búferes de pantalla usando:

```
SDL_Flip(pScreen);
```

Esto realizará una copia de la nueva imagen desde el búfer trasero al búfer delantero y estaremos listos para escribir el siguiente cuadro. Es el búfer delantero el que entonces se muestra en la pantalla.

La tercera opción es hacer uso de la función `exDrawLevelTile` y redibujar el ladrillo por debajo de cada objeto dinámico usando código existente. Esto es un punto intermedio entre ambas opciones razonable.

```
tile_x = Enemy[obj].xpos /
    iTileWidth;
tile_y = Enemy[obj].ypos /
    iTileHeight;
exDrawLevelTile(tile_x, tile_y);
```

Si los enemigos se extienden sobre dos a más ladrillos, entonces cada uno de esos ladrillos necesitará ser redibujado. Detectar esto es bastante fácil puesto que nos hemos cerciorado de que cada ladrillo del juego

tiene el mismo tamaño: 32x32. Así que si la posición del enemigo no está en el límite de los 32 pixels (p.e. 0, 32, 64, 96, 128, etc), debe superponerse por lo menos a más de un ladrillo. Podemos ampliar el código anterior para incluirlo en el Listado 8.

Para eso necesitamos un reciclador C, el símbolo de porcentaje (%) es el operador del modulo y devuelve el resto del calculo (en nuestro caso `xpos/Ancho`). Así que si aparece un resto es que estamos solapando el ladrillo vecino.

Desde aquí podemos dibujar y actualizar la pantalla completa. También podemos llamarla repetidamente en un bucle para mover los enemigos alrededor de la pantalla.

Capta el mensaje

SDL utiliza un paradigma conocido como *programación orientada a eventos*. Esto funciona de la misma manera que lo hace X Window, Microsoft Windows y la mayoría de la programación para entornos gráficos de usuario (GUI para los amigos). También está algo relacionado con la programación de redes. En una aplicación “tradicional” para consolas, el programa comienza, hace algo y termina. Un software altamente interactivo no pueden trabajar de este modo, por culpa de la etapa “esperar una entrada(input)”. Mientras esperamos una *entrada*, no puede suceder nada y si ocurre algún tipo de *entrada* que no es la que esperamos (como una *entrada* de ratón, mientras esperamos una pulsación del teclado) seremos incapaces de procesarla. Además, la rutina de *entrada* está diciendo que está “bloqueada”, lo cual impide que el resto de la acción (como las actualizaciones de pantalla o las animaciones) ocurra hasta que se reciba la *entrada* esperada. Los juegos basados en turno, como el ajedrez, funcionan bien con el bloqueo de *entrada*, pero los juegos de acción como el nuestro no lo hacen, así que tendremos que usar el paradigma de programación orientado a eventos.

Con la programación orientada a eventos, en lugar de preguntar, cuando queremos, a Linux por una *entrada* específica, nuestro programa se pone en un bucle y continuamente pregunta “hay alguna *entrada*, hay alguna *entrada*”.

Entonces, cuando llega la *entrada* (que puede ser de cualquier tipo), la procesaremos y continuaremos con nuestro

Listado 7: Dibujado de la pantalla

```
01 exDrawStatic();
02 while(TheGame.bIsRunning)
03 {
04 exRepairScreen(); /* Quitar
    todos los objetos dinámicos */
05 exUpdateLevel(); /* Actualizar
    los objetos dinámicos */
06 exDrawDynamic(); /* Dibujar
    los objetos dinámicos que
    regresan a la pantalla */
07 SDL_Update(pScreen, 0,0,0,0);
08 }
```

bucle. Preguntando de nuevo “hay alguna *entrada*, hay alguna *entrada*”. Este proceso también es conocido como encuesta (polling). Fijémonos que se llama “orientada a eventos” y no “orientada a *entrada*” porque Linux puede decirnos varias cosas (como que la ventana se ha redimensionado o se ha cerrado) y no solamente que el teclado o el ratón envían una entrada de datos. Bajo SDL nos da eventos cuando:

- Hay señal de entrada desde el teclado, el ratón o la palanca de juegos
- Nuestra ventana se activa o se desactiva
- Nuestra ventana se redimensiona (solo ocurre si se llama a `SetVideoMode` desde `SDL_VIDEORESIZE`)
- Nuestra ventana se cierra, en

Listado 8: Detectado de los límites del ladrillo

```
01 if (Enemy[obj].xpos %
    iTileWidth)
02 exDrawLevelTile(tile_x+1,
    tile_y);
03 if (Enemy[obj].ypos %
    iTileHeight)
04 exDrawLevelTile(tile_x,
    tile_y+1);
05 if ((Enemy[obj].xpos %
    iTileWidth) &&
    (Enemy[obj].ypos %
    iTileHeight))
06 exDrawLevelTile(tile_x+1,
    tile_y+1);
```

Listado 9: Un manejador de eventos bastante simpático para Explorer Dug

```
01 SDL_Event ev;
02 while(SDL_PollEvent(&ev) >= 0)
03 {
04     /* ev.type representa el tipo
05     de mensaje que recibimos */
06     /* Maneja el mensaje y
07     continúa la encuesta para más
08     eventos */
09     if (ev.type == SDL_QUIT) /*
10         Debemos manejar esto */
11     break;
12     /* Otros manejadores de
13     mensajes van aquí */
14 }
```

cuyo caso debemos salir del bucle de eventos y cerrar el programa

- Nuestra ventana necesita redibujarse (en ese caso debemos llamar a *SDL_UpdateRect*)

Estos eventos son puestos en una cola internamente y nos los dan cuando preguntamos por ellos, uno por uno con la función *SDL_PollEvent*.

Este bucle funciona muy mal ¿Por qué? Porque no hay noción del tiempo. Primero, con un lazo tan apretujado, Linux tiene muy poco tiempo para procesar algo. Es el equivalente moderno del ¡10 GOTO 10! Este problema se puede resolver fácilmente introduciendo un corto retraso con la función *SDL_Delay* que vimos el mes anterior. Esperando un milisegundo cada vez que ejecutamos el bucle no significa mucho para nuestro juego (seguimos sintiendo que el teclado y el ratón responden), pero para el Sistema Operativo será la diferencia entre trabajar bien y detenerse completamente.

El segundo problema con el tiempo es que el juego se ejecutara tan rápido como pueda. De una forma bastante divertida, pero eso no es lo que queremos. Los juegos deben funcionar tan rápido como sea posible, pero no al máximo. Si nuestro monitor refresca su pantalla 70 veces en un segundo, entonces tendremos 70 imágenes diferentes cada segundo. Si nuestro monitor refresca a 85 Hz tendremos 85 imá-

genes. Este es un límite natural provocado porque la función *SDL_UpdateRect* espera hasta el siguiente ciclo de redibujado del monitor antes de permitir que nuestro juego continúe.

Para hacer esta temporización uniforme tenemos que manejarla nosotros mismos introduciendo bloqueos de cuadro, donde el ritmo de presentación de cuadros se taponan hasta pararlo si exceder un valor específico. Los juegos comerciales de 3D (como Unreal) siempre marcan 60 marcos por segundo, ya que esto está bastante cerca del índice de refresco de una televisión o de un monitor y hace que el juego parezca que va fino.

60 cuadros por segundo (fps) significa que debemos terminar el proceso de cada cuadro en menos de 16,666 milisegundos. Desde luego que se puede conseguir, si bloqueamos nuestro juego a 60 fps.

Si conseguimos producir el cuadro actual más aprisa que esto, esperaremos un rato (otra vez, con la función de *SDL_Delay*) hasta que han expirado nuestros 16,666 milisegundos, si no continuaremos con el cuadro siguiente. Sin embargo, cada vez que completemos un ciclo de nuestro bucle esperaremos siempre 1 ms, lo mínimo para cerciorarnos de que el procesador, y otros aplicaciones (tales como los demonios del servidor) consiguen un momento de respiro.

Podemos medir tiempo con la función *SDL_GetTicks*. Ésta devuelve el número de milisegundos desde que el programa fue iniciado y puede ser colocada en cualquier parte del código de actualización del dibujo para averiguar cuánto tiempo tomó el cuadro anterior. Esto debe estar en torno a algunos milisegundos.

El resultado de *SDL_GetTicks* se almacena en un *Uint32* lo que significa que el contador de tiempo se reiniciará después de alrededor de 4 mil millones de milisegundos o, en otras palabras, 49 días. En nuestro juego, esto solamente sucederá si usted juega 49 días seguidos.

Esto es inverosímil, y los efectos secundarios (negativos) serán probablemente mínimos. Pero en los juegos en línea con acceso masivo de multi-jugadores tales como Ultima Online, esta acontecimiento es la norma, no la excepción. Así que si está utilizando

Listado 10: Un manejador de eventos aun más simpático para Explorer Dug

```
01 SDL_Event ev;
02 Uint32 prev_time, curr_time;
03 Uint32 period, delta_time;
04 period = 1000/60; /* 16
05     milisegundos entre cuadros */
06 TheGame.bIsRunning = TRUE;
07 prev_time = SDL_GetTicks();
08 while(SDL_PollEvent(&ev) >= 0
09     && TheGame.bIsRunning)
10 {
11     exUpdateInterface(&ev); /*
12     procesado de eventos del
13     teclado/palanca etc */
14 do {
15     curr_time = SDL_GetTicks();
16     /* Comprobar si han pasado 49
17     días */
18     if (curr_time > prev_time)
19     delta_time =
20     curr_time-prev_time;
21     else
22     delta_time = 0xffffffff -
23     (prev_time-curr_time) + 1;
24     SDL_Delay(1);
25 }
26 while(SDL_PollEvent(&ev) >= 0
27     && delta_time < period);
28 prev_time = curr_time;
29 if (ev.type == SDL_QUIT)
30 break;
31 exRepairScreen();
32 exUpdateLevel();
33 exDrawDynamic();
34
35     SDL_UpdateRect(TheGame.pScreen
36     , 0,0,0,0);
37 }
```

esta función ya está advertido sobre el tema y de los pasos convenientes para evitar el problema, como hacemos en el Listado 10.

Ahora tenemos un lazo de evento en su sitio, y estamos listos para procesar cualquier evento que recibamos, sea del ratón, el teclado o, incluso, de una palanca de juegos. Pero por lo que se refiere a abarcar completamente esos acontecimientos, y lo más importante, cómo los utilizaremos para controlar nuestro juego, tendremos que esperar hasta el mes próximo... ■