Diseño de juegos bajo Linux

Todo bajo Control

Este mes todo está bajo control. Y Steven Goodwin controla un montón. Veremos cómo el jugador controla el juego y cómo el juego controla los personajes.

POR STEVEN GOODWIN

El mes pasado vimos cómo SDL proporciona toda la entrada a nuestro juego a través de un bucle de eventos con *SDL_PollEvent*. Esta función cumplimenta nuestra variable *SDL_Event* con información útil sobre el evento siguiente. Ahora necesitamos interpretar estos datos.

Mensaje en una botella

Lo primero que hay que advertir es que *SDL_Event* no está implementado como estructura, sino como una unión, porque cada elemento es mutuamente exclusivo, no tiene sentido obtener información sobre el puntero del ratón para un evento que reporta la pulsación de una tecla. Por consiguiente, la lectura de parte de una estructura que no corresponda con el tipo de evento producirá invariablemente datos inutilizables y no el contenido del ultimo mensaje del ratón, por ejemplo.

Solamente necesitamos manejar los eventos en los que tenemos interés; todo lo demás será ignorado, SDL no intentará ni supondrá qué hacer con el evento. En nuestro juego la pantalla siempre se actualiza 60 veces por segundo, así que no necesitamos manejar el mensaje de *SDL_VIDEOEXPOSE*, por ejemplo.

Cada estructura del evento contiene una miríada de información que pertenece al evento. Por ejemplo, *SDL_MouseButtonEvent* indica qué botón de ratón fue pulsado, la posición x e y del puntero del ratón y qué dispositivo del ratón era (para sistemas con dos, o más, ratones). Los nombres asociados a las variables dentro de cada estructura se pueden encontrar en las páginas de manual, o en *SDL_events.h*, que generalmente habita en */usr/local/include/SDL*.

Desde aquí podemos crear manejadores de evento para cada mensaje en el que estemos interesados. Esto se hace creando funciones separadas que manejen una clase particular de evento (como la entrada).

Whisky en la Jarra

El control es probablemente la parte más importante de un juego de ordenador. Es la interfaz entre el jugador y el juego. Todos los diseñadores de juegos saben esto y pasarán muchas semanas (incluso meses) retocando el sistema de control hasta que esté "perfecto". SDL no hace este proceso más fácil, sino que proporciona la suficiente información sobre el controlador, sea del ratón, el teclado o una palanca de juegos, para permitirnos la suficiente flexibilidad para crear un buen sistema de control.

La entrada del teclado es la forma de entrada más empleada en este juego, ¡ya que la mayoría de los PCs tiene tendencia a tenerlo! Pero como cada pulsación del teclado tiene 2 partes, un evento de tecla pulsada y un evento de tecla liberada, es necesario escuchar ambos.

Manejar múltiples teclas es muy fácil y la velocidad de auto repetición del teclado no causa problemas adicionales, ya que solo importa si la tecla ha sido pulsada o no. Si aparece auto repetición, una característica importante es que se puede almacenar siempre el tiempo en el

cual la tecla fue presionada utilizando la función *SDL_GetTicks*.

Con la asignación de un puntero *SDL_Event* (pEevnt), se encuentra la tecla correcta con la linea:

```
SDLKey key = pEvent->key.keysym.sym;
```

Esto puede parecer un poco agotador para una simple pulsación de tecla, pero los otros miembros de estas estructuras también son redundantes (Por ejemplo *pEvent ->key.state* incluye la misma información como tipo de evento; pulsado o soltado) o intrascendente (el código de exploración específico del teclado). Prefiero pensar que esto es minucioso y no prolijo.

El valor *sym* equivale a una de las muchas definiciones que indican una tecla específica: Por ejemplo SDLK_LEFT para el cursor izquierdo y SDLK_RIGHT para el cursor derecho. Las teclas alfanuméricas mapean muy bien el ASCII, así que comprobar la tecla ‘0’ es tan simple como debe ser:

```
if (key == '0') /* ...se podia haber usado SDLK_0 aquí... */
```

No obstante, recomiendo usar las definiciones de *SDL_keysyms.h* ya que las letras activan siempre eventos con sus homólogas minúsculas del ASCII (el valor de tecla sería igual a ‘a’ no ‘A’) sin tener en cuenta el bloqueo de mayúsculas o la tecla ‘mayúsculas.

```
if (key == SDLK_a) /* ...Vaya! Efectivamente son minúsculas... */
```

Usaremos este conocimiento recién encontrado para almacenar pulsaciones de tecla de una manera parecida a los juegos, mediante la creación de una matriz especial para indicar que teclas están controlando el personaje. Esto es diferente de que tecla ha sido pulsada porque, cuando dos teclas son pulsadas, queremos la pulsada más recientemente para controlar la dirección. Nuestro código será algo así como el Listado 1.

Escribir el sistema de control de esta manera hace que sea más fácil cambiar la asignación de teclas o añadir un control de palanca de juegos. Adviértase, sin embargo, que el mensaje de ‘tecla liberada’ se perderá si cambia del juego al depurador. SDL no funciona y no está habilitado para escuchar mensajes de tecla liberada (los cuales, ahora van al depurador). Por tanto cuando retorne al juego tomará un evento adicional de tecla pulsada y tecla soltada para restablecer el sistema ¡Esto es más notorio si el personaje continúa moviéndose a pesar de que no se pulsan teclas!

Sin embargo, comprobar todas las teclas de este modo, puede ser levemente incomodo. La abstracción de las teclas de control del juego es una ‘buena cosa’, pero tener que escribir el código para aceptar el nombre de los jugadores para la tabla de puntuaciones máximas cuando el cursor izquierdo ha sido remapeado, digamos, a ‘Z’. en este caso, solamente queremos preguntarle a SDL que tecla concreta se ha pulsado. Podemos hacer esto con la función

```
Uint8 *keystate = SDL_GetKeyState(NULL);
```

El puntero devuelto es una matriz (que no está permitido modificar) que representa cada tecla en el sistema y está referenciado con las macros SDLK_* que

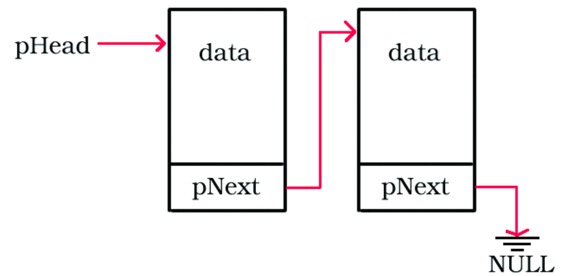

Figura 1: Nuestro juego hasta ahora.

Cuadro 1: Di ‘Patata’

SDL ha hecho que capturar la pantalla sea un asunto muy simple que apenas requiere una función, *SDL_SaveBMP*. Esta es el espejo exacto de la función *SDL_LoadBMP* que hemos usado en la primera parte y toma el puntero de la superficie (que en nuestro caso es la superficie de pantalla) y un nombre de fichero.

```
01 if (ev.type == SDL_KEYDOWN &&
02 ev.key.keysym.sym==SDLK_F12)
03 {
04 static int curr_grab = 0;
05 char GrabName[32];
06 sprintf(GrabName,
07 "DugPic%d.bmp", curr_grab);
08 ++curr_grab;
09 SDL_SaveBMP(TheGame.pScreen,
10 GrabName);
11 }
```

Cuadro 2: Acerca del tiempo

Este mes hemos hecho una adición a la función de actualización. Es un parámetro para controlar el tiempo. Generalmente, cada llamada a los comportamientos que la función de *Update()* realizaría un cuadro valido de la actividad. Sin embargo, en casos extremos, el juego podría haber tomado dos cuadros para procesar la ultima imagen. Esto requeriría que la función de *Update()* se llamara dos veces dentro de un solo cuadro para compensar. Sin embargo, llamar dos veces a *Update()* no permite que el comportamiento se optimice con eficacia. Así que en vez de eso lo llamamos una vez y le pedimos que actualice N cuadros. Los juegos con una granularidad más fina pasarían un número de coma flotante (indicando los segundos transcurridos) a sus funciones de la actualización.

Tabla 1: Retrollamadas

Función	Notas
Start game	Llamada una vez al comienzo del nivel
Reset game	Llamado siempre que el nivel consiga reiniciarse, después de que el jugador muera por ejemplo
Update	Mover o animar al objeto
Draw	Dibujar este objeto en la pantalla
Destroy	Liberar cualquier memoria ocupada al inicio del juego

veremos más tarde. Cualquier valor distinto de cero en esta matriz significa que la tecla está actualmente pulsada.

```
if ( keystate[SDLK_F12] ) ➘
printf("The F12 key is down.");
```

Cualquier tecla que se ha mantenido pulsada durante dos cuadros consecutivos será considerada 'VERDADERO' para ambos cuadros, porque esta función informa del estado de la tecla y no eventos. Normalmente esto no es un problema, pero si se está usando esta característica para realizar una captura de pantalla, obtendrá una imagen por cada cuadro mientras la tecla este pulsada. Véase el Cuadro 1 "Di 'Patata'", para más detalles sobre como realizar una captura de pantalla.

También se puede determinar que modificador de teclas están pulsados (como 'mayúsculas' o 'alt'); pero para esto necesita una función diferente. Véase el Listado 2.

Mi Chica Piruleta

La entrada de la palanca de juegos es muy fácil de programar bajo SDL, que incluso permite múltiples palancas de juego sin ningún esfuerzo extra. En nuestro ejemplo, se supondrá que hay una palanca de juegos, pero en un juego más completo es deseable dejar al jugador que seleccione que palanca de juegos utilizará y que personaje controlará.

Listado 1: Pulsaciones de teclas

```
01 void exUpdateInterface(const
   SDL_Event *pEvent)
02 {
03 if (pEvent->type ==
   SDL_KEYDOWN)
04 {
05 SDLKey key = pEvent-
   >key.keysym.sym;
06 if (key == SDLK_LEFT)
07 {
08 iMovement[EXI_MOVE_LEFT] =
   TRUE;
09 iMovement[EXI_MOVE_RIGHT] =
   FALSE;
10 }
11 /* ... etcétera ... */
```

Listado 2: Modificadores de teclas

```
01 if (SDL_GetModState() & KMOD_SHIFT) printf("Una de las teclas de
   mayúsculas se ha presionado.");
02 if (SDL_GetModState() & KMOD_LSHIFT) printf("Se ha presionado la
   tecla de mayúsculas izquierda.");
```

Al contrario que el control de teclado, SDL no inicializa la palanca de juegos automáticamente cuando se llama a *SDL_Init*. Hay tres pasos adicionales: Inicializar el subsistema de la palanca de juegos (que no es distinto del subsistema de vídeo o el temporizador), poner en marcha los eventos de la palanca de juegos (si no el bucle de eventos no informará sobre los eventos de la palanca de juegos) y abrir un puerto de palanca de juegos para la lectura. Véase el Listado 3.

Si se tienen que manejar varias palancas de juegos, entonces la función *SDL_NumJoysticks* informará del número de palancas de juegos conectadas al sistema. El parámetro para la función *SDL_JoystickOpen*, como se puede esperar, indica cual de las palancas de juegos hay que abrir.

Una vez que se tiene un puntero *SDL_Joystick* se pueden consultar las capacidades y los parámetros de la palanca. Esto es importante porque cada palanca de juegos es distinta: Algunos tienen controles tipo HAT (conmutador digital de 8 vías, otros tienen 'track balls' y otros tienen más botones de lo que se pueda imaginar. Las palancas de juegos para simulación aérea suelen incluir todos los anteriores.

Determinar las capacidades de la palanca de juegos le permitirá saber, por ejemplo, si el jugador tendrá que reasignar los botones de la palanca de juegos al teclado. Nosotros solo necesitamos control a izquierda, derecha y botón de salto, así que cualquier palanca de juegos disponible debe ser suficientemente buena para *Explorer Dug*.

Refiriendonos de nuevo a la tabla comentada anteriormente hay tres eventos principales en los que estamos interesados: *SDL_JOYAXISMOTION*, *SDL_JOYBUTTONDOWN* y *SDL_JOYBUTTONUP*. Puesto que solamente hay una palanca de mando en nuestro juego, podemos ignorar cualquier elemento de la estructura del evento y suponer que cualquier evento de la palanca de juego debe haber venido de la palanca cero (porque es la única que hemos abierto).

Entonces podemos usar *pEvent->jaxis.value* (que oscila entre -32768 y 32767, indicado totalmente a la izquierda, totalmente a la derecha) para controlar el juego. Véase el Listado 4.

La banda muerta es un tema interesante. Todas las palancas de juego se mueven, que yo sepa. Pero las palancas de juego también se mueven cuando nadie las está tocando. Tiemblan ligeramente y por tanto generan eventos de palanca de juegos espúreos en cada cuadro. Puesto que esto es una característica del hardware no podemos evitar que suceda, pero podemos limitar los efecto que esto tiene en nuestro juego. Para hacerlo, crearemos una banda muerta alrededor de la parte central de la palanca. Cualquier movimiento que ocurra en esta área será ignorado y tratado como si la palanca de juegos enviara 0,0. Fuera de la banda muerta, se tratará a la palanca normalmente. Aquí he usado una banda muerta de 8000, que es un número razonable para este tipo de juegos, aunque en un juego

Listado 3: Inicialización de la palanca de juegos

```
01 SDL_Joystick *pJoyStick;
02 SDL_InitSubSystem(SDL_INIT_JOYSTICK);
03 SDL_JoystickEventState(SDL_ENABLE);
04 pJoyStick = SDL_JoystickOpen(0);
```

profesional, este valor debe ser configurable.

Jumpin' Jack Flash

Como los dispositivos del teclado y el ratón, las palancas de juegos también se pueden leer sin el bucle de eventos. Para hacerlo hace falta llamar a la función *SDL_JoystickUpdate* que lee el hardware, antes de usar una de las funciones del listado 5 para consultar los datos.

Ahora hemos recuperado la entrada, deseamos controlar algo en el juego y por eso, necesitamos algún objeto del juego …

Gente corriente

cada objeto del juego, ya sea el jugador, un enemigo, o una puerta de salida, tiene un número de elementos comunes. Todos tienen una posición de comienzo, todos tienen gráficos y todos "hacen cosas". Sin embargo, cada objeto en el juego tendrá una posición de comienzo distinta, gráficos distintos, y todos hacen "cosas" distintas. Para implementar un buen armazón debemos poder identificar esos elementos y aislarlos con eficacia del motor común del juego.

Datos comunes para todos los objetos:

- Posición de comienzo
- Una superficie conteniendo gráficos

Código único para cada tipo de objeto:

- Como inicializar el objeto
- Como actualizarlo, p. e. dirección y velocidad de la trayectoria.
- Como dibujar el objeto
- Como destruir el objeto

Necesitamos estas distinciones porque, por ejemplo, aunque podemos describir cada objeto en términos de una superficie, cada objeto en el juego no utilizará esta superficie de la misma manera. Algunos objetos consistirán en un ladrillo (los enemigos), algunos usarán varios (la puerta de salida) y otros se desvanecerán en algún momento (el jugador).

Realmente esta lista no está completa y no dudaremos en introducir nuevas características durante el desarrollo del juego, pero es un buen punto de partida. El método que he elegido para implementar estas características utiliza retro-llamadas (callback) individuales a las funciones, donde cada una tiene un propósito específico.

Sin embargo, la implementación exacta de cómo trabaja esa función se controla por el código que es específico para cada tipo de objeto. Cada uno de esos tipos se denomina comportamiento. Y cada función de retro-llamada manejará como se comporta ese objeto en una situación en particular. Nuestro juego inicial de retro-llamadas se muestran en la Tabla 1. Esta lista imita nuestro bucle de juego general realmente bien: Comenzamos el juego, procesamos un bucle de actualización del dibujo y entonces salimos. No tenemos ninguna distinción entre el nivel y el juego, puesto que cada nivel es un mini-juego en sí mismo. La lista también demuestra que tenemos una jerarquía de dos niveles de datos: configuración y estado. Los datos de la configuración se crean en el comienzo del juego e incluyen cosas como la superficie SDL o su posición de comienzo. Por el contrario, los datos del estado incluyen las cosas que cambian durante el juego, como la posición o la dirección actual.

Esto perfila la información que necesitamos para nuestra estructura EX_OBJECT. Cada objeto del juego utilizará esta estructura genérica, almacenada como parte de una lista enlazada, con las funciones de retro-llamada púdicamente ocultas dentro de otra estructura (EX_OBJ_VTABLE), como se ve en el Listado 6.

Aquí hemos creado la configuración más usada y las variables de estado dentro de la estructura EX_OBJECT y marcado las otras con punteros nulos. Será responsabilidad de cada comportamiento asignar la memoria para estos

Listado 4: Movimiento de la palanca de juegos

```
01 if (pEvent->type ==
   SDL_JOYAXISMOTION)
02 {
03 if (pEvent->jaxis.axis == 0)
   /* X-axis; assume
04 stick 0 */
05 {
06 int DeadBand = 8000;
07
   TheGame.iface.iMove[EXI_MOVE_L
   EFT] = FALSE;
08
   TheGame.iface.iMove[EXI_MOVE_R
   IGHT] = FALSE;
09 if (pEvent->jaxis.value <
   -DeadBand)
10
   TheGame.iface.iMove[EXI_MOVE_L
   EFT] = TRUE;
11 else if (pEvent->jaxis.value >
   DeadBand)
12
   TheGame.iface.iMove[EXI_MOVE_R
   IGHT] = TRUE;
13 }
14 }
```

Listado 5: Consultado la palanca de juegos

```
01 SDL_JoystickGetAxis(SDL_Joystick *joystick, int axis);
02 SDL_JoystickGetHat(SDL_Joystick *joystick, int hat);
03 SDL_JoystickGetBall(SDL_Joystick *joystick, int ball, int
04 *dx, int *dy);
05 SDL_JoystickGetButton(SDL_Joystick *joystick, int
06 button);
```

Listado 6: EX_OBJECT

```
01 typedef struct sOBJECT {
02 /* Setup - los datos no
   cambian */
03 int init_x, init_y; /*
   Posición de inicio */
04 SDL_Surface *pGfx; /* apunta a
   una superficie existente */
05 void *pBhvSetupData;
06 /* Estado - consigue el
   reinicio */
07 int x, y; /* Posición actual
   */
08 void *pBhvStateData;
09 /* Admin */
10 EX_OBJ_VTABLE VTable;
11 /* Lista de componentes
   enlazados */
12 struct sOBJECT *pNext;
13 } EX_OBJECT;
```

Listado 7: Enemigo caminando

```
01 typedef struct sDHE_STATE {
02 int iDirection; /* La dirección en que camina, -1 para izquierda, +1
   para derecha */
03 int iAnimFrame; /* Marco actual de la animación, mapeado directamente
   a una región */
04 int iAnimDir; /* La animación ping-pong entre cuadros
   0->1->2->3->2->1->0. será -1 or +1, dependiendo de la dirección */
05 } EX_DHE_STATE;
```

punteros y crear estructuras que el propio comportamiento pueda comprender. Tomando el ejemplo de un enemigo que camina a ciegas de izquierda a derecha (véase el listado 7). Este estúpido enemigo horizontal (EEH) inicializará su estado en cada reset de los datos de configuración que hemos definido como tal como se ve en el listado 8.

Como lenguaje, C no facilita ningún mecanismo para parar su cambio de información de configuración durante el ciclo de actualización (donde solamente debería ser cambiado el estado), ni le previene de datos del estado que cambian durante el ciclo de la creación (donde no debería cambia nada). Por lo tanto tenemos que confiar en el sentido común y en los programadores competentes. Pero hay una característica del lenguaje que podemos utilizar para centrar el desarrollo.

La Tabla La palabra VTable es la abreviatura de 'virtual table' y se ha cogido prestado de C++. Es una manera de llamar a funciones específicas en objetos específicos, sin tener que especificarlos en tiempo de compilación. Esto permite un cierto nivel de abstracción. Cada objeto hace esto estableciendo los punteros a las funciones privadas como en el listado 9. Declaramos cada una de las funciones (dheStart, dheUpdate y dheDraw) como estáticas para hacer sus nombres invisibles a cualquiera fuera del fichero actual. Esto asegurara que solo podemos llamar a estas funciones referenciando a VTable, de este modo:

```
pObject->VTable.Start(pObject);
```

Para prevenir referencias a NULL o a otros punteros de función inválidos, prepararemos un conjunto de retro-llamadas predeterminadas bajo demanda. Cada objeto podrá entonces reemplazar (o sobrecargar) esos punteros de funciones con los suyos propios.

Este método puede parecer redundante, ya que requiere 2 instancias de *pObject*, pero normalmente es más eficiente que añadir conmutadores de estado en torno a cada llamada a Start, Update o Draw. Esto también escala bien si añadimos nuevas funcionalidades a los objetos a través de su VTable. Siempre podremos usar una pequeña función de envoltorio para prevenir el error casual cuando una VTable de un objeto se esta usando, pero se pasa el puntero de otra.

```
void exDrawObject(EX_OBJECT ↩
*pObj)
{
pObj->VTable.↩
Draw(pObj);
}
```

Listado 8: Enemigo horizontal mudo

```
01 typedef struct sDHE_SETUP {
02 int x1,y1,x2,y2; /* grados
   para el enemigo */
03 int iInitialDir; /* -1 para
   izquierda, +1 para derecha */
04 int iSpeed;
05 } EX_DHE_SETUP;
```

Tu me llevas

¡Pero porque cada EEH utiliza el mismo código, no significa que utiliza los mismos datos! Tenemos ya una estructura para la información de la configuración, así que si tenemos una manera de cambiar estos datos iniciales entonces podemos introducir mucha variedad en nuestro juego, basándolo en los datos. Bien, eso se hace antes los que se dice, ya que podemos ampliar nuestra susodicha función *Bhv_CreateDumbHorizontal* para incluir un juego de parámetros, tales como velocidad y dirección.

A fin de hacer estos parámetro generales para todos los comportamientos, los crearemos como cadenas de caracteres y dejaremos que el comportamiento los analice en busca de algo significativo. De esta manera podríamos también leerlos directamente de un archivo de texto para hacer los niveles de juego personalizables de una forma muy simple.

Todas los objetos basados en comportamiento trabajan usando exactamente los mismos métodos; aunque algunos requerirán más código que otros. El jugador, por ejemplo, requerirá manejar colisiones con el mundo, manejar las recolecciones y el contacto con los enemigos. Programaremos un manejador el próximo mes. ■

Listado 9: Ajustando punteros

```
01 BOOL Bhv_CreateDumbHorizontal(EX_OBJECT *pObj)
02 {
03 /* ... otras initializaciones de cosas ... */
04 pObject->VTable.Start = dheStart;
05 pObject->VTable.Update = dheUpdate;
06 pObject->VTable.Draw = dheDraw;
07 /* ... */
08 }
09 static void dheStart(EX_OBJECT *pObj) { /* ... hace cosas ... */ }
10 static void dheUpdate(EX_OBJECT *pObj) { /* ... hace cosas ... */ }
11 static void dheDraw(EX_OBJECT *pObj) { /* ... hace cosas ... */ }
```