Game designing under Linux

Taking Control

This month is all about control. And Steven Goodwin has lots of it. We look at how the player controls the game and how the game controls the characters.

BY STEVEN GOODWIN

We saw last month [2] how SDL provides all input to our game through an event loop with *SDL_PollEvent*. This function fills our *SDL_Event* variable with useful information about the next event. We now need to interpret this data.

Message in a bottle

The first thing to notice is that *SDL_Event* is not implemented as structure, but as a union, because each element is mutually exclusive – there is no point having information about the mouse pointer for an event that describes a key press. As a consequence, reading part of a structure that doesn't match the event type will invariably produce unusable data, and not the contents of the last mouse message, for example.

We only need to handle those events we have an interest in – everything else will be ignored, SDL will not try and guess what to do with the event. In our game we always update the screen 60 times a second, so do not need to handle the SDL_VIDEOEXPOSE message, for example.

Each event structure holds a myriad of information pertaining to the event. For instance, *SDL_MouseButtonEvent* indicates which mouse button was pressed, the x and y location of the mouse pointer and which mouse device it was (for those systems with two, or more, mice). The names of the member variables within each structure can be found from the man pages, or *SDL_events.h*, which usually lives in */usr/local/include/SDL*.

From here we can create event handlers for each message we're interested in. We do this by creating separate functions that handle one particular class of event (like input).

Whiskey in the Jar

Control is probably the most important part of a computer game. It is the interface between the player and the game. Every game designer knows this and will spend many weeks (even months) tweaking the control system until it is "just so". SDL doesn't make this process any easier, but it does provide enough information about the controller, be it from the mouse, keyboard or a joystick, to allow us enough flexibility to create a good control system.

Keyboard input is the most prominent form of input for this game, since most PC's tend to have one! But because each key press comes in two parts, a key down event and a key up event, we need to listen for both.

Handling multiple keys is very easy, and the keyboard's auto-repeat speed causes no additional problems, since we're concerned only whether the key is down, or not. If auto-repeat becomes an important feature we can always store the time at which the key was pressed by utilising the *SDL_GetTicks* function.

Given an *SDL_Event* pointer (pEvent) we can find the correct key with the line:

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

This may appear a little long-winded for a simple key press, but the other members of these structures are either redundant (pEvent->key.state, for example, includes the same information as the event type – *pressed* or *released*) or unimportant (the scancode for your specific keyboard). I prefer to think of this as thorough, and not verbose.

The *sym* value equates to one of many defines that indicate the specific key: SDLK_LEFT for the left cursor and SDLK_RIGHT for the right cursor, for instance. The alphanumeric keys map very nicely to ASCII, so checking for the '0' key is as simple as it should be:

```
if (key == '0') /* ...we could➘
 have used SDLK_0 here... */
```

Listing 1: Key presses

```
void exUpdateInterface(const ➘
SDL_Event *pEvent)
{
 if (pEvent->type == SDL_KEYDOWN)
  {
  SDLKey key = pEvent-➘
>key.keysym.sym;
  if (key == SDLK_LEFT)
   {
   iMovement[EXI_MOVE_LEFT] = ➘
TRUE;
   iMovement[EXI_MOVE_RIGHT] = ➘
FALSE;
   }
 /* ... and so on ... */
```

Listing 2: Key modifiers

```
if (SDL_GetModState() & KMOD_SHIFT)    printf("One of the shift keys has
been pressed.");
if (SDL_GetModState() & KMOD_LSHIFT)  printf("The left shift key has
been pressed.");
```

Listing 3: Joystick initialization

```
SDL_Joystick  *pJoyStick;
 SDL_InitSubSystem(SDL_INIT_JOYSTICK);
 SDL_JoystickEventState(SDL_ENABLE);
 pJoyStick = SDL_JoystickOpen(0);
```

However, I recommend using the defines from *SDL_keysyms.h*, as the letters always trigger events with their lower case ASCII counterparts (the *key* value would equal 'a', not 'A'), regardless of the caps lock or shift key.

```
if (key == SDLK_a) /* ...Yeah!
 It's really lower case... */
```

We shall use this newfound knowledge to store the key presses in a game-friendly manner, by creating a special array to indicate which keys are controlling the character. This is different from which key has been pressed because, when two keys are down, we want the most recently pressed key to control the direction. Our code looks something like Listing 1:

Writing the control system in this manner makes it much easier to change the key assignments, or add joystick control. Be warned, however, that the 'key up' message will get lost if you switch from game to debugger. SDL isn't running, and is not able to listen for the key up messages (which now go to the debugger). So when you return to the game it will take an extra key-down, and key-up event to reset the system. This is most notable if your character continues to move despite no keys being pressed!

Checking for *all* keys in this manner, however, can be slightly cumbersome. Abstracting the game control keys is a 'good thing', but imagine having to write the code to accept the players name for the high score table where the left cursor had been remapped to 'Z', say. In this case, we just want to ask SDL which particular keys are down. We can do that with the function,

```
Uint8 *keystate = SDL_
GetKeyState(NULL);
```

The pointer returned is an array (which you are not allowed to modify) that represents each key in the system, and is referenced with the SDLK_* macros we saw earlier. Any non-zero value in this array means that the key is currently down.

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

Because this function reports the key state, and not events, any key that has been held down for two consecutive frames will be considered 'TRUE' for both frames. Normally this is not a problem, but if you were using this feature to perform a screen grab, you'd get one image every frame whilst the key was down. See Box 1 "Hold Me Now", for details on how to take a screen grab.

You can also determine which of the *key modifiers* (like ALT, or SHIFT) are pressed – but that requires a different function. See Listing 2.

Box 1: Hold me Now

SDL has made screen grabs a very simple affair requiring just one function, *SDL_SaveBMP*. This is an exact mirror of the *SDL_LoadBMP* function we've used since part 1, and takes the surface pointer (which in our case is the screen surface) and a filename.

```
if (ev.type == SDL_KEYDOWN &&
ev.key.keysym.sym==SDLK_F12)
 {
 static int curr_grab = 0;
 char GrabName[32];
 sprintf(GrabName,
"DugPic%d.bmp", curr_grab);
 ++curr_grab;
 SDL_SaveBMP(TheGame.pScreen,
GrabName);
 }
```

My Girl Lollipop

Joystick input is very easy to program under SDL, it even supports multiple joysticks without any extra effort. In our examples, we shall assume one joystick, but in a more complete game you'd want to let the player select which joystick they used, and which character it would control.

Unlike keyboard control, joysticks are not automatically initialized by SDL when we call *SDL_Init*. There are three additional steps: you must initialize the joystick subsystem (which is no different to the video, or timer, subsystems), switch on joystick events (otherwise the event loop will not report joystick messages), and open a joystick port for reading. See Listing 3.

If you are supporting several joysticks, then the *SDL_NumJoysticks* function will report the number of joysticks attached to your system. The parameter to the *SDL_JoystickOpen* function, as you've probably guessed, represents which of the joysticks to open.

Once you have an SDL_Joystick pointer you can query the capabilities and parameters of the stick. This is important because each joystick is different: some have hat controls (8-way

Listing 4: Joystick motion

```
 if (pEvent->type == SDL_JOYAXISMOTION)
  {
  if (pEvent->jaxis.axis == 0) /* X-axis; assume
stick 0 */
   {
   int DeadBand = 8000;
   TheGame.iface.iMove[EXI_MOVE_LEFT] = FALSE;
   TheGame.iface.iMove[EXI_MOVE_RIGHT] = FALSE;
   if (pEvent->jaxis.value < -DeadBand)
    TheGame.iface.iMove[EXI_MOVE_LEFT] = TRUE;
   else if (pEvent->jaxis.value > DeadBand)
    TheGame.iface.iMove[EXI_MOVE_RIGHT] = TRUE;
   }
  }
```

Listing 5: Query joystick

```
SDL_JoystickGetAxis(SDL_Joystick *joystick, int axis);
SDL_JoystickGetHat(SDL_Joystick *joystick, int hat);
SDL_JoystickGetBall(SDL_Joystick *joystick, int ball, int
*dx, int *dy);
SDL_JoystickGetButton(SDL_Joystick *joystick, int
button);
```

Table 1: Callbacks

Function	Notes
Start game	Called once at the start of the level
Reset game	Called whenever the level gets reset, after the player dies for example
Update	Move or animate the object
Draw	Draw this object to the screen
Destroy	Free any memory we've allocated in start game

digital switches), some have track balls and some have more buttons than you could possibly imagine! Flight simulator joysticks usually have all of the above!

Determining the capabilities of the joystick will let you know if the player will have to reassign joystick buttons to the keyboard, for example. We only need left and right control, and a jump button, so any available joystick should be good enough for *Explorer Dug*.

Referring back to table one above there are three main events that we're interested in, *SDL_JOYAXISMOTION*, *SDL_JOYBUTTONDOWN* and *SDL_JOYBUTTONUP*. Since there's only one joystick in our game we can ignore the *which* element of the event structure and assume that any joystick event must have come from stick zero (because that's the only one we've opened).

We can then use *pEvent->jaxis.value* (which varies between -32768 and 32767, indicating full left, to full right) to control the player, please see Listing 4.

Listing 6: EX_OBJECT

```
typedef struct sOBJECT {
 /* Setup - data doesn't change */
 int    init_x, init_y; /* Start position */
 SDL_Surface  *pGfx; /* points to an existing surface */
 void *pBhvSetupData;
 /* State - gets reset */
 int x, y; /* Current position */
 void *pBhvStateData;
 /* Admin */
 EX_OBJ_VTABLE VTable;
 /* Linked list component */
 struct sOBJECT  *pNext;
 } EX_OBJECT;
```

The deadband is an interesting point. All joysticks move, that we know. But joysticks also move when you're not touching them! They all jitter slightly, and so we'll get spurious joystick events each frame. Since this is a hardware feature, we can not stop it from happening, but we can limit the effects it has on our game. To do so, we create a deadband around the central portion of the joystick. Any movement within this area is ignored, and treated as if the joystick were reporting 0,0. Outside the deadband, we treat the stick as normal. I've used a hard-coded deadband of 8000 here, which is a reasonable number for this type of game, although in a professional release this value should be customisable.

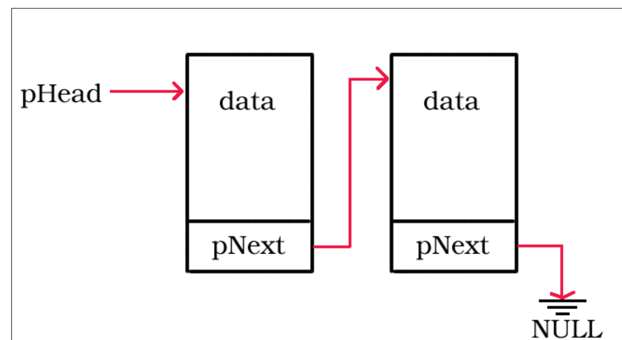

Figure 1: Our game so far

Jumpin' Jack Flash

Like the keyboard and mouse devices, the joysticks can also be read without the event loop. To do so requires you to call the *SDL_JoystickUpdate* function that reads the hardware, before using one of the functions in Listing 5 to query the data:

Now we've retrieved the input, we want to control something in the game and so, we need some game objects…

Common People

Every object in the game, whether it is the player, an enemy, or an exit gate, has a number of common elements. They all have a start position, they all have graphics and they all "do stuff". However, every object in the game will have a *different* start position, *different* graphics, and all do *different* "stuff". In order to implement a good framework we must be able to identify those elements and isolate them effectively from the common game engine.

Common data, for all objects:
- Start position
- A surface containing graphics

Unique code, for each type of object:
- How to initialise the object
- How to update them, e.g. speed and direction of travel
- How to draw the object
- How to destroy the object

We need this distinction because, for example, although we can describe each object in terms of a surface, not every object in the game will use this surface in the same way. Some objects will consist of one tile (the enemies), some will use several (the exit gate), and some will be faded in over time (the player).

This list is certainly not complete, and we'll no doubt introduce new features during the lifetime of the game, but it is a good starting point. The method I've chosen to implement these features utilize individual callback functions, where each has a specific purpose.

However, the precise implementation of *how* that function works is controlled by code that is specific to each type of object. Every such type is called a *behaviour*. And each callback function will handle how that object *behaves* in any particular situation. Our initial set of callbacks are shown in table 1. This list mimics our general game loop quite

Listing 7: Walking enemy

```
typedef struct sDHE_STATE {
 int iDirection; /* the direction he's walking, -1 for left,+1 for right */
 int iAnimFrame; /* Current animation frame, maps directly to a region */
 int iAnimDir;  /* The animation ping-pongs between frames 0->1->2->3->2->1->0. Will be -1 or +1, depending on direction */
 } EX_DHE_STATE;
```

nicely: we start the game, process an update-draw loop, and then exit. We have no distinction between level and game, since each level is a mini-game in itself. The list also demonstrates that we have a two-level hierarchy of data: setup and state. Setup data is created at the start of the game and includes things like the SDL surface or its start position. In contrast, the state data includes things that *change* during the game, like the current position or direction.

This outlines the information we need for our EX_OBJECT structure. Every object in the game will use this generic structure, stored as part of a linked list, with the callback functions nicely hidden inside another structure (EX_OBJ_VTABLE), as shown in Listing 6.

Here we've created the commonly used setup and state variables inside the EX_OBJECT structure, and marked the others with void pointers. It will be the responsibility of each behaviour to allocate memory for these pointers, and create structures that the behaviour itself can understand. Taking the example of an enemy that walks blindly from left to right. See Listing 7. This dumb horizontal enemy (DHE) will initialize its state on every reset from the setup data that we have defined as such Listing 8.

As a language, C does not provide any mechanism to stop you changing setup information during the update cycle (where you should only be changing the state), nor does it prevent you from changing state data during the render cycle (where you should not be changing anything). We therefore have to rely on common sense and polite programmers! But there is one language feature we can use to focus development.

The Table

The word VTable is short for 'virtual table', and is borrowed from C++. It is a way of calling specific functions on specific objects, without having to specify them at compile time. This allows for a certain level of abstraction. Each object does this by setting up the pointers to private functions like Listing 9:

We declare each of the functions (dheStart, dheUpdate and dheDraw) as *static* to make its name invisible to anything outside the current file. This ensures we can only call these functions by dereferencing the VTable, thus:

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

To prevent us dereferencing NULL, or otherwise invalid, function pointers, we prepare a set of default callbacks upon creation. Each object can then replace (or overload) these function pointers with its own.

This method may appear redundant, since it requires two instances of *pObject*, but it is usually more efficient than adding *switch* statements around every call to Start, Update, or Draw. It also scales well if we add new functionality to the object through its VTable. We can always use a small wrapper function to prevent the occasional bug where one object's VTable is used, but is passed the pointer of another.

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

Driven By You

Just because each DHE uses the same code, doesn't mean it uses the same data! We already have a structure for the setup information, so if we have a way of changing this initial data then we can introduce a great deal of variety into our game, making it *data-driven*. Well, that's no sooner said than done, as we can extend our *Bhv_CreateDumbHorizontal* function above to include a set of parameters, such as speed and direction.

In order to make these parameters general across all behaviours, we'll create them as string of characters and let the behaviour parse them into something meaningful. In this way we could also read them directly from a text file to make customisable game levels a very simple prospect.

All behaviour-based objects work using exactly the same methods – although some will require more code than others. The player, for example, will require us to handle collisions with the world, handle the pickups, and handle contact with the enemies. We'll get a handle on that next month! ■

Box 2: About Time

We've made one addition to the update function this month. It is a parameter to control time. Usually, each call to the behaviours Update() function would perform one frames worth of activity. However, in extreme cases the game could have taken two frames to process the last image. This would require the Update() function to get called twice within a single frame to compensate.

However, calling Update() twice doesn't allow the behaviour to optimise itself effectively. So instead we call it once and ask it to update N frames. Games that have a finer granularity would pass a floating point number (indicating seconds elapsed) to their update functions.

Listing 8: Dumb horizontal enemy

```
typedef struct sDHE_SETUP {
 int x1,y1,x2,y2; /* extents for
enemy */
 int iInitialDir; /* -1 for left,
+1 for right */
 int iSpeed;
 } EX_DHE_SETUP;
```

Listing 9: Setting pointers

```
BOOL Bhv_CreateDumbHorizontal(EX_OBJECT *pObj)
{
 /* ... other initialisation stuff ... */
 pObject->VTable.Start = dheStart;
 pObject->VTable.Update = dheUpdate;
 pObject->VTable.Draw = dheDraw;
 /* ... */
}
static void dheStart(EX_OBJECT *pObj)  { /* ... do stuff ... */ }
static void dheUpdate(EX_OBJECT *pObj) { /* ... do stuff ... */ }
static void dheDraw(EX_OBJECT *pObj)  { /* ... do stuff ... */ }
```