Moving Sprites on Screen

Load and Blit

In the first part of this tutorial we saw how to load an image into a surface, and blit it onto the screen. This month, we'll show how those basic operations of load and blit can be used to compose the entire game screen.

BY STEVEN GOODWIN

Our title screen in issue 33 (p66) consisted of one surface and one blit. But with one surface, and several blits, we can do much more. How? Well, we take a single surface and mark out areas of 32 by 32 pixels. Each area is called a *tile*, or a *region*. We can then blit each tile to specific places on the screen surface to build up a complete image. Because we can blit one tile many times (and in many different places), we will use less memory than the welcome screen, although it covers the same area. Most platform games use this method. Even real-time strategy games (like Command & Conquer and Warcraft) use this system, although their graphics are isometric in appearance.

By making the tile size uniform across the whole game, our graphics can be laid out in predictable places on the surface image making the code that controls them much easier to write. We shall also stipulate that each surface should be 640 pixels wide, giving a consistent number of tiles (20) on each line. This is an arbitrary decision to make the artists work more uniform: the more options you support, the greater the room for error! The code to draw a single tile on screen is then as illustrated in Listing 1.

Although each tile will be aligned onto a 32x32 boundary, our exDrawTile function has been written to draw tiles at any arbitrary position, allowing it to be reused by the render code for the player, and his enemy's. Determining which tile is drawn, and where to place it, is the responsibility of a loop, and a simple array of tile indices. We can do that with a short function as in Listing 2.

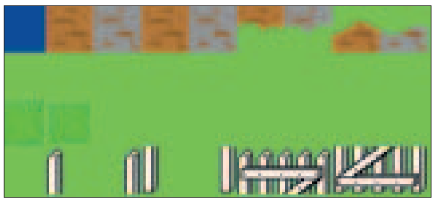
Figure 1: A sample tile set

Since we have no means to stretch our tiles, any large features in our levels (such as the exit gate) will have to be built by combining several smaller tiles. Conversely, any images that are smaller than 32x32 will only occupy a portion of the tile, and we shall have to utilise transparent pixels to hide the fact.

Break on Through

Transparency is handled with a technique known as *colour keying*. For each surface that requires transparency we specify a single colour that, instead of being blitted, will be ignored. There is no specific colour that has to be used for this purpose. Nor should there be. Since all games are different, and every graphic in the game will use a different set of colours, you should choose a colour key specific to you. However, one carefully chosen colour will usually suffice for all graphics in the game.

The best choices are very bright, extreme, colours that no one would, generally, have a use for. In *Explorer Dug* we shall use bright green. Fluorescent green, even! With RGB components of 0, 255,0 it is highly unlikely that anyone would 'accidentally' use it as part of a genuine graphic, even grass. If artists want a bright green colour then 0, 254, 0 is just as good, provided you have enough resolution with your chosen colour depth (see boxout "Clarity of Colour"). I have always used a bright green colour key in my games, and never yet had a problem.

Having chosen a colour, we now have to tell the surface which one it is!

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

This function takes the surface, and applies a colour key of bright green to it. The colour key has to be given in the same format as the surface and so we need to use *SDL_MapRGB* function (which is passed the format of the surface), to convert the RGB colour components. If you're using 8-bit colour (i.e. a palletised surface), then this function will search for the closest match to the given RGB values in its table. After specifying the colour key for the surface,

Listing 1: Drawing a tile

```
void exDrawTile(SDL_Surface *pTile, int iRegion, int x, int y)
{
SDL_Rect src, dest;

     src.x = (iRegion % iNumTileWidth) * iTileWidth;
     src.y = (iRegion / iNumTileWidth) * iTileHeight;
     src.w = iTileWidth;
     src.h = iTileHeight;

     dest.x = x;
     dest.y = y;
     dest.w = iTileWidth;    /* Not actually needed,
                                since they're ignored */
     dest.h = iTileHeight;

     if (SDL_BlitSurface(pTile, &src, pScreen, &dest) < 0)
        fprintf(stderr, "Blit error! %s", SDL_GetError());
}
```

Missing the Game

To conserve space in the magazine, the variables used in the code snippets are usually shortened versions of the actual source code. You will notice that the global variable *theGame*, for example, is not present in the most of listings given.

every blit from this surface will ignore all pixels of that colour, treating them as transparent. You can, of course, change the colour key at any point during your application, or turn it off entirely with:

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

This makes it possible to replace the colour key for specific graphics (or even specific tile regions) in the game without having to create a separate surface. Unfortunately, SDL doesn't provide an easy way of retrieving the current colour

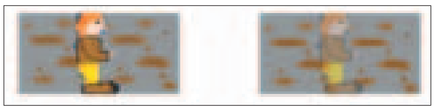
Figure 2: On the left, Explorer Dug lacks an alpha. On the right, he has an alpha of 128

key, but you can reach directly into the surface structure to get it as in Listing 3.

The *old_colour_key* value holds the colour in a format that the surface understands, and so does not need the *SDL_MapRGB* function when we re-apply the key to the surface.

Big in Japan

In addition to making a single colour transparent, we sometimes want to make the entire surface transparent. Not totally, but partially. This will give the image a ghost-like appearance, as we can see the character, *and* the scenery behind him. To do this we need to apply an *alpha channel*. Special effects make extensive use of alpha channels; explosions, smoke and rain all use several overlaid textures, each with its own alpha component. We shall be using alpha for a much simpler purpose: fading the player in at the start of the game, to indicate that they are invincible.

The alpha component of a surface is held as a single 8-bit value that ranges from *SDL_ALPHA_TRANSPARENT* (0) to *SDL_ALPHA_OPAQUE* (255). Any value in-between is also valid, although 128 is treated as a special case internally, and so processes faster. It is possible to use colour key surfaces and alpha surfaces at the same time.

Like the colour key, SDL provides no way of retrieving the alpha value of a surface without referencing the variable directly, as in Listing 4.

It is also possible to specify different amounts of alpha for each individual pixel, which allows us create a much smoother edge around the characters. However, this is quite tricky, since most file formats do not include this information. The GIMPs XCF does, but there's currently no library support for this format. Instead, we have to load two images, one containing alpha information and another containing graphic (intensity) information, and join them together manually by reading specific pixels from the alpha surface, and combining them with the intensity surface. This, however, exceeds our current purpose.

All Together Now

We are now ready to combine all three parts of our game screen (backdrop, level tiles, and dynamic objects). This can be done in a couple of ways. Most obviously, we could draw each of the parts in turn, as in Listing 5.

However, to improve the game speed, we shall make one additional stipulation: the first tile (region 0) will never be rendered to the game area. Ever. This is because most of the tiled area is, in fact, empty. It would waste a lot of CPU time if most of the render code involved drawing completely transparent blocks onto the screen – and in games, we're always glad to discover speed optimisations. The fastest way to draw something is not to draw it at all, so we'll sacrifice this tiny amount of memory to the greater god of speed.

We can further improve speed by combining the backdrop and tiles into one render function. This can be done very

Listing 2: Loop

```
int iNumTileHeight = 20, iNumTileWidth = 15;
  /* fill a 640x480 screen */
int iTileWidth = 32, iTileHeight = 32;
  for(ty=0; ty<iNumTileHeight; ty++)
    for(tx=0; tx<iNumTileWidth; tx++)
       exDrawTile(pTileSurface, map_data[tx + ty * iNumTileWidth],↵
       tx*iTileWidth  ty*iTileHeight);
```

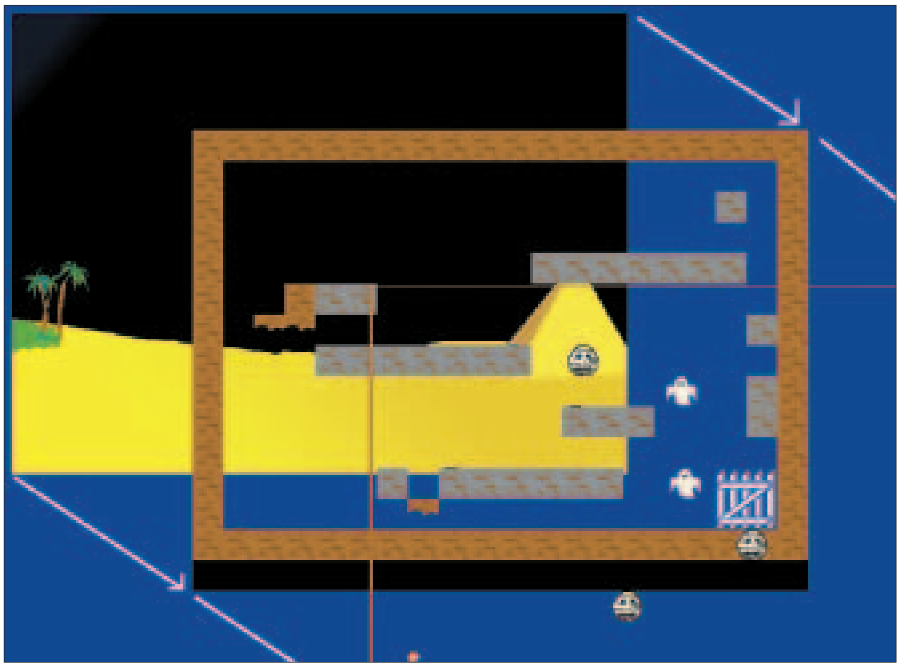
Figure 3: Stacking the surfaces

simply by saying that if we want to draw the first tile (the completely transparent one), we actually copy (i.e. blit) a portion of the backdrop to the screen. Otherwise, we draw the tile *instead* of the backdrop. In this way, a screen with 300 tiles (20x15) will draw 300 tiles worth of data. Whereas, in the original suggestion, we'd be drawing 600. The question of "is it quicker to draw one large (backdrop) tile than several small ones" I shall leave to the benchmarking tests. There are many variables to consider with this type of optimisation, and the only way to know which method will execute faster is to try them both. During development I found the "several small ones" solution used less processing (about 8%, according to *top*) and so adopted it.

Using this idea, we can now write a special routine to draw a level tile, which may (or may not) copy the backdrop surface in lieu of a tile (see Listing 6).

If there are tiles that involve transparencies we will need to copy the backdrop before drawing it, also. Determining which specific tiles have transparent pixels is not difficult – but it is slow. So we can either check for them when the game starts, or stipulate to the artists that only tiles numbered 8 and above (for example) can have transparencies. We shall choose the second option.

Erase and Rewind

Now we are able to draw the screen once – we need to be able to draw it several times. Which will allow us to animate some enemies moving around on top of it. The code for this is very easy, but must be done correctly as to prevent unsightly artefacts on screen, or an unnecessary burden on the CPU (see Listing 7).

By 'dynamic' we actually mean two things:

- Objects that move
- Objects that remain still, but animate

Repairing the screen involves removing every dynamic object from it, so we've left with the static background again. This can be done in a couple of different ways.

Firstly, we could ask each dynamic object to remember what the screen looked like before it drew itself, and then redraw that block when we request a repair. Although good, this involves a lot of work on the part of each object, and

Listing 3: Retrieving the colorkey

```
Uint32 old_colour_key;

      old_colour_key = pGeneralTileGfx->format->colorkey;
      SDL_SetColorKey(pGeneralTileGfx,
                     SDL_SRCCOLORKEY,
                     SDL_MapRGB(pGeneralTileGfx->format, 0, 255, 0));

      exDrawTile(pGeneralTileGfx, 12, xpos, ypos);

      SDL_SetColorKey(pGeneralTileGfx,
                     SDL_SRCCOLORKEY,
                     old_colour_key);
```

Listing 4: Retrieving the alpha value

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

Clarity of Color

If the surface you are color keying has been created with SDL_LoadBMP, chances are it will use 24-bit colour. This means that the green component has 8 bits of resolution, 254 and 255 will be different colors, and there's no problem. On the other hand, if the surface is only 16-bit color, then green will probably only occupy 5 or 6 bits. This loss of precision means that 254 and 255 are the same since they will both be represented as 31 internally (255>>3 == 254>>3). To be *absolutely* safe you need to check the surface format manually. Unfortunately this test occurs in-game, but our graphics are created off-line where this information is unavailable. Realistically speaking, however, as long as the green component of genuine colors does not exceed 247 (which assumes 5 bits of precision), you should be safe.

Listing 5: Drawing the game screen

```
void exDrawStatic(void)
{
int tx, ty;
int obj;
  /* Backdrop */
  SDL_BlitSurface(pBackdrop, NULL, pScreen, NULL);
  /* Tiles */
  for(ty=0;ty<iNumTileHeight;ty++)
    for(tx=0;tx<iNumTileWidth;tx++)
    exDrawTile(pGeneralTileGfx, map_data[tx + ty *
iNumTileWidth], tx, ty);
  /* Dynamic objects - for example, enemies */
  for(obj=0;obj<iNumEnemies;obj++)
    exDrawTile(pEnemyTileGfx, Enemy[obj].Graphic,
Enemy[obj].xpos, Enemy[obj].ypos);
}
```

Listing 6: Drawing a level tile

```
void exDrawLevelTile(int tx, int ty)
{
SDL_Rect src;
int iRegion;

  iRegion = exGetTileRegion(tx, ty);

  src.x = tx * iTileWidth;
  src.y = ty * iTileHeight;
  src.w = iTileWidth;
  src.h = iTileHeight;

  if (iRegion == 0  && pBackdrop)
    SDL_BlitSurface(pBackdrop, &src, pScreen, &src);

  if (iRegion)
    exDrawTile(pGeneralTileGfx, iRegion, src.x,➥
    src.y);
}
```

Double Buffer

Under X11, we can achieve smooth graphics because we never write directly to the screen: we write to a separate buffer and then SDL blits *that* to the screen. However, some drivers *do* write directly to the screen. If you're using such a driver, and are unable to draw everything quick enough, you will see "tearing".

This occurs because you are able to see half of the last frame, and half of the current frame at the same time. In these situations you can employ *double buffering*. This technique involves two screen buffers, one that gets drawn to the monitor, and one that we draw our game into. This means we will never be writing to the same frame that's being displayed.

Employing double buffering in SDL is very easy. First, we need to set-up the video mode:

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

This adds two buffers to the screen surface, a front buffer, and a back buffer. We can write to this surface as normal with the *SDL_BlitSurface* function, and SDL will only write into one buffer – the back one. Then, instead of updating the screen with *UpdateRect*, we flip the screen buffers using:

```
SDL_Flip(pScreen);
```

This will make a copy the new image from the back buffer to the front buffer, and we can get ready to write the next frame. It is the front buffer that is then displayed on screen.

doesn't consider the possibilities that two (or more) objects might overlap. However, this can often be the most efficient repair method.

We could also call *exDrawStatic* again. This would work, but be very slow. However, if the backdrop image were to ever animate we would need to implement this solution.

The third option is to make use of the *exDrawLevelTile* function, and redraw the tile underneath every dynamic object using existing code. This is a reasonable trade-off.

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

If the enemy extends over two or more tiles, then each of these tiles will need to be redrawn. Detecting this is fairly easy since we've made sure that *every* tile in the game is the same size: 32x32. So, if the position of the enemy isn't on a 32-pixel boundary (i.e. 0, 32, 64, 96, 128, etc), it must overlap at least one more tile. We can extend the above code to include Listing 8.

For those in need of a C refresher, the percent symbol (%) is the modulus operator, and returns the remainder from the calculation (in our case *xpos / Width*). So, if there is any remainder present we must be overlapping the neighbouring tile.

From here we can draw and update the entire screen. We can also call it repeatedly in a loop to move the enemies around the screen.

Get The Message

SDL uses a paradigm known as *event-driven programming*. It works in the same way as X Window, Microsoft Windows and most forms of GUI programming. It

Listing 7: Screen draw

```
exDrawStatic();

while(TheGame.bIsRunning)
  {
  exRepairScreen();  /* Remove all the old dynamic objects */
  exUpdateLevel();   /* Update the dynamic objects */
  exDrawDynamic();   /* Draw the dynamic objects back to the screen */

  SDL_Update(pScreen, 0,0,0,0);
  }
```

is also somewhat akin to network programming. In a ‘traditional’ console application, the program starts, does something, and exits. Sometimes, it starts, waits for input, does something else, and then exits. A lot of highly interactive software can not work in this manner because of the ‘wait for input’ phase. Whilst we are waiting for input, nothing else can happen, and if some form of input occurs that we are not expecting (like mouse input, whilst waiting for a key press) we will be unable to process it. In addition, the input routine is said to be ‘blocking’, which prevents anything else (like screen updates or animations) from happening until the requested input has been received. Turn-based games, like chess, work well with blocking input, but action games like ours don’t, so we have to use the event-driven paradigm.

With event-driven programming, instead of asking Linux for *specific* input when we want it, our program sits in a loop and continually asks “is there any input, is there any input”.

Then, when the input arrives (of whatever type it may be), we process it and continue with our loop. Once again asking, “is there any input, is there any input”. This process is also known as *polling*. Note that it is called *event*-driven, and not *input*-driven because Linux can tell us about several things (like whether the window has been resized or closed), and not just mouse or keyboard input. Under SDL, we are given events when:

- There is some input from the keyboard, mouse, or joystick
- Our window is activated, or deactivated
- Our window is resized (only occurs if SetVideoMode was called with *SDL_VIDEORESIZE*)
- Our window is closed, in which case we must exit the event loop and close the program
- Our window needs redrawing (in which case we must call *SDL_Update Rect*)

These events are queued up internally and given to us when we ask for them, one at a time, with the *SDL_PollEvent* function.

This loop works. Badly! Why? Because there is no concept of time. Firstly, with

Listing 8: Detecting tile boundaries

```
 if (Enemy[obj].xpos % iTileWidth)
   exDrawLevelTile(tile_x+1, tile_y);

 if (Enemy[obj].ypos % iTileHeight)
   exDrawLevelTile(tile_x, tile_y+1);

 if ((Enemy[obj].xpos % iTileWidth) && (Enemy[obj].ypos % iTileHeight))
   exDrawLevelTile(tile_x+1, tile_y+1);
```

Listing 9: A fairly nice event handler for Explorer Dug

```
SDL_Event ev;

  while(SDL_PollEvent(&ev) >= 0)
    {
    /* ev.type represents the type of message we've received */
    /* Handle the message, and continue polling for more events */
    if (ev.type == SDL_QUIT) /* We MUST handle this */
      break;
    /* Other message handlers go here */
    }
```

LIsting 10: A nicer event handler for Explorer Dug

```
SDL_Event ev;
Uint32 prev_time, curr_time;
Uint32 period, delta_time;
  period = 1000/60;       /* 16 milliseconds between frames */
  TheGame.bIsRunning = TRUE;
  prev_time = SDL_GetTicks();

  while(SDL_PollEvent(&ev) >= 0 && TheGame.bIsRunning)
    {
    exUpdateInterface(&ev); /* process events for keyboard/stick etc */
  do {
    curr_time = SDL_GetTicks();
    /* Check for 49 day wrap-around */
    if (curr_time > prev_time)
      delta_time = curr_time-prev_time;
    else
      delta_time = 0xffffffff - (prev_time-curr_time) + 1;
        SDL_Delay(1);
     }

  while(SDL_PollEvent(&ev) >= 0 && delta_time < period);
    prev_time = curr_time;

    if (ev.type == SDL_QUIT)
      break;
      exRepairScreen();
      exUpdateLevel();
      exDrawDynamic();
      SDL_UpdateRect(TheGame.pScreen, 0,0,0,0);
     }
```

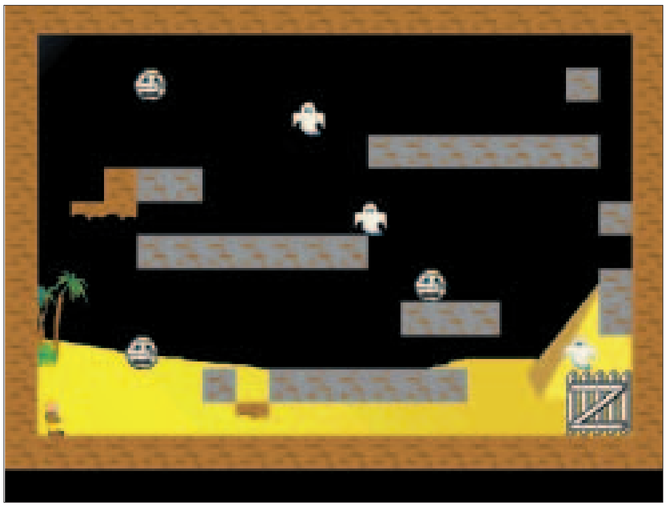

Figure 4: Complete game image

such a tight loop, Linux has very little time to process anything else. It’s the modern equivalent of *10 GOTO 10*! This problem can be solved easily by introducing a short delay with the *SDL_Delay* function we saw last month. Waiting for a millisecond each time around the loop doesn’t mean a lot to our game (the keyboard and mouse still feel responsive), but to the OS it’s the difference between working well, and grinding to a halt.

The second problem with time is that the game will run as fast as it can. Funnily enough, this isn’t what we want! Games should run as fast as possible – but no faster! If our monitor refreshes its display 70 times a second, then we will get 70 different images every second. If our monitor refreshes at 85 Hz we will get 85 images. This is a natural limit that occurs because the *SDL_UpdateRect* function waits until the next monitor redraw cycle before allowing our game to continue.

To make this timing consistent we have to handle it ourselves by introducing *frame lock*, where the frame rate is capped to stop it from exceeding a specific value. Commercial 3D games (like Unreal) always target 60 fps, as this is fairly close to the refresh rate of a television or monitor and makes the game look smooth.

60 fps means we must finish processing each frame in under 16.666 milliseconds. That’s certainly possible to achieve, so we shall frame lock our game to 60fps.

If we succeed in producing the current frame quicker than this, we’ll wait a while (again, using the *SDL_Delay* function) until our 16.666 milliseconds have expired, otherwise we’ll continue with the next frame. Every time around our event loop, however, we will *always* wait for 1 ms, just to make sure the processor, and other applications (such as server daemons) get a chance to breath.

We can measure time with the *SDL_GetTicks* function. It returns the number of milliseconds since the program was started, and can be placed either side of the update-draw code to work out how long the last frame took. This should be in the order of milliseconds.

The result of *SDL_GetTicks* is stored in a *Uint32* which means the timer will wrap after around 4 billion milliseconds – or 49 days. In our game, this will only happen if you play for 49 days continuously!

This is unlikely, and the (negative) side effects are likely to be minimal. But in massively multi-player online games such as Ultima Online, this occurrence is the norm, not the exception. So if you are using this function you be aware of the issues, and take suitable steps to avoid the problem, as we do in Listing 10.

We now have an event loop in place, and are ready to process any event we receive, be it from the mouse, the keyboard or even a joystick. But as for the full scope of what those events are, and, more importantly, how to use them to control our game, will have to wait until next month… ■

SELLING OUT FAST!

For more information see:
www.linux-magazine.com/Backissues