Using SDL with OpenGL

by Ian Romanick

1 Introduction

OpenGL provides platform-independent access to accelerated 3D rendering. However, it takes more than just 3D rendering capabilities to display graphics on the user's screen. OpenGL relies on a set of platform-dependent window system interface layers to bind OpenGL rendering to the rest of the system. For example, X Windows based systems use GLX to associate a window with an OpenGL rendering context. Microsoft's Windows and Apple's OS X have similar interface layers.

These interfaces lie at the boundary of the platform-independent OpenGL API and the necessarily platform-dependent window system API. To provide the rich and varied set of functionality on each platform, these window system interface layers are also, by necessity, platform dependent. There exist a few wrapper libraries that provided a limited set of functionality and hide these platform dependencies. Two well known examples are GLUT and SDL.

This chapter will introduce some SDL for the purpose of interfacing with OpenGL. It is by no means a comprehensive tutorial on SDL. Instead, this chapter focuses providing the basics of three key topics.

1. Creating a window suitable for OpenGL rendering.
2. Event loop.
3. Timing.

1.1 Creating a Drawing Surface

Before using any part of SDL, the library must be initialized. This gives the library an opportunity to allocate internal data structures, connect with operating system resources, and perform other internal bookkeeping.

```
int SDL_Init(Uint32 flags);
```

The *flags* parameter to `SDL_Init` selects the subsystems that will be initialized. This is a bitwise-or of flag values for each subsystem. Only two subsystems, the video subsystem and the timer subsystem, will be used in this chapter. Figure 1 shows the use of `SDL_Init` to initialize these subsystems. If the initialization is successful, zero will be returned. Otherwise -1 will be returned.

Figure 1 SDL initialization.

```
if (SDL_Init(SDL_INIT_VIDEO | SDL_INIT_TIMER) < 0)
    /* ... error path ... */

atexit(SDL_Quit);
```

The call to `atexit` adds the function `SDL_Quit` to the list of functions that are automatically called when the program terminates. This is a common idiom with SDL. The guarantees that all of the resources used by SDL itself will be released when the program terminates, even if program is terminated by crashing.

Creating the actual drawing surfaces consists of two primary steps. First, the desired attributes of the drawing surface must be specified. Once the surface is fully described, it is created. The function `SDL_GL_SetAttribute` specifies a minimum acceptable value for a specific attribute.

```
int SDL_GL_SetAttribute(SDL_GLattr attr, int value);
```

`SDL_SetVideoMode` creates the drawing surface[1]. The *`width`* and *`height`* parameters specify the dimensions of the surface. The *`bpp`* parameter is not used for OpenGL surfaces and should be set to 0.

The *`flags`* parameter is a bitwise-or of flag values, much like the *`flags`* parameter to `SDL_Init`. For OpenGL surfaces, `SDL_OPENGL` must be set. If `SDL_FULLSCREEN` is set, the window will be "fullscreen." This usually means that the video mode will be changed to match the dimensions of the window. If `SDL_RESIZABLE` is set, the window will be created with appropriate decorations so that the user can resize the window. When this happens, SDL sends resize events to the application. These events will be covered shortly.

The call to `SDL_SetVideoMode` can fail for a variety of reasons, not the least of which is inability of the system to satisfy all of the requested minimum values. For example, requesting 64 bits of red is sure to fail on any current generation system.

```
SDL_Surface *SDL_SetVideoMode(int width, int height, int bpp, Uint32 flags);
```

Figure 2 demonstrates requesting at least 8 bits of red, green, and blue in the color buffer and 24 bits in the depth buffer. Line 3 also requests that the surface be double buffered. Line 5 calls `SDL_SetVideoMode` to create an 800x600 window that can be used for OpenGL rendering.

Figure 2 Creating an SDL drawing surface.

```
1 SDL_GL_SetAttribute(SDL_GL_RED_SIZE, 8);
2 SDL_GL_SetAttribute(SDL_GL_GREEN_SIZE, 8);
3 SDL_GL_SetAttribute(SDL_GL_BLUE_SIZE, 8);
4 SDL_GL_SetAttribute(SDL_GL_DEPTH_SIZE, 24);
5 SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);
6
7 screen = SDL_SetVideoMode(800, 600, 0, SDL_OPENGL);
8 if (screen == NULL)
9     /* ... error path ... */
```

1.2 Events

Most graphical applications, including games, are driven by an event loop. As the name suggestions, this is a loop in the main execution path of the code that processes a series of events. Events, in this context, are occurrences outside the program that affect its behavior. These take the form of user input from the keyboard or mouse, timers expiring, messages from the operating system, and so on. SDL has a very flexible mechanism for receiving events.

SDL provides two functions to receive events. `SDL_PollEvent` will return immediately. If no event was available, it returns 0. `SDL_WaitEvent`, on the other hand, will block until an event is available. Both functions return 1 if an event is available and fill in the supplied *`SDL_Event`* structure.

```
int SDL_PollEvent(SDL_Event *event);

int SDL_WaitEvent(SDL_Event *event);
```

While `SDL_WaitEvent` is blocking, it may give up the program's CPU time and allow other programs to run. If there are no other programs to run, the operating system will put the CPU in a lower power mode. This is critically very important for mobile systems. Spinning in a loop repeatedly calling `SDL_PollEvent` will drain the battery much faster than blocking in `SDL_WaitEvent`.

A simple event loop is shown in Figure 3. The structure of this loop is important. The code blocks in `SDL_WaitEvent` until a first event is available. Remaining events are processed, without blocking, by calling `SDL_PollEvent`. After all events have been processed, a message is printed at line 15.

The only event processed by this event loop is `SDL_QUIT`, which is typically generated by the user clicking the "window close" icon on the window. When this event is received, a flag is set that signals the event loop to terminate.

The *`SDL_Event`* is a union of structures. Each structure contains a tag, called *`type`*, the tells which type of event it is. This allows code to examine the *`type`* field of the *`SDL_Event`* event union, then

[1] This function is so named for largely historical reasons. SDL has its root in DOS and Linux console environments where there was no window system. On these platforms the call to `SDL_SetVideoMode` would literally set the video mode.

Figure 3 Simple SDL event loop.

```
bool done = false;

while (!done) {
    SDL_Event event;

    SDL_WaitEvent(&event);
    do {
        switch (event.type) {
        case SDL_QUIT:
            done = true;
            break;
        }
    } while (SDL_PollEvent(&event));

    printf("got some events\n");
}
```

select the specific event structure from the union that contains the specific event data. This is used, for example, to determine which key was pressed or released on the keyboard, the location of the mouse pointer, or the direction the joystick is being pushed.

A slightly more complex event loop is shown in Figure 4. Here `SDL_KEYDOWN` and `SDL_KEYUP` events are processed as keys are pressed and released on the keyboard.

Figure 4 SDL keyboard event loop.

```
bool done = false;

while (!done) {
    SDL_Event event;

    SDL_WaitEvent(&event);
    do {
        switch (event.type) {
        case SDL_QUIT:
            done = true;
            break;
        case SDL_KEYUP:
        case SDL_KEYDOWN:
            printf("`%c' was %s\n",
                   event.key.keysym.sym,
                   (event.key.state == SDL_PRESSED) ? "pressed" : "released");
            break;
        }
    } while (SDL_PollEvent(&event));
}
```

1.3 Timing

Graphical applications will typically only update their display in response to some stimulus. In many cases this is input from the user. This is probably sufficient for a graphical editor, for example. However, many other types of applications update their display in response to another type of stimulus: the passage of time. A game, for example, will update the display 60 times per second whether the player is doing anything or not. The event loop presented in the previous section has no way to "wake up" due to the passage of time. This can be very easily added using a timer.

In SDL, timers can be created that will call a function a specific rate. After calling `SDL_AddTimer`, every *`interval`* milliseconds[2] the function specified to *`callback`* will be called. It will be passed *`param`*

[2] The SDL documentation states the the resolution of these timers is 10ms.

as a parameter.

```
SDL_TimerID SDL_AddTimer(Uint32 interval, SDL_NewTimerCallback callback,
    void *param);
```

The function passed as `callback` must fit the prototype shown below. When called, `param` is the value supplied as `param` to `SDL_AddTimer`. `interval` is the current timer interval. The value returned by the callback function will become the new timer interval. If zero is returned, the timer will be canceled. It is common practice to simply return `interval`.

```
typedef Uint32 (*SDL_NewTimerCallback)(Uint32 interval, void *param);
```

Note that the timer mechanism does not generate events. The event loop still can't wake up! However, the callback function can use `SDL_PushEvent` to generate an event. This function inserts a new event in the event stream.

```
int SDL_PushEvent(SDL_Event *event);
```

Figure 5 and Figure 6 show examples of setting a timer and generating an event from the callback. If this code were used with the event loop in the previous section, the `SDL_WaitEvent` call would wake up approximately every 10ms (100 times per second).

Figure 5 Adding a timer.

```
SDL_TimerID timer_id = SDL_AddTimer(10, timer_callback, NULL);
if (timer_id == NULL)
    /* ... error path ... */
```

Figure 6 Generating an event from the timer callback.

```
Uint32 timer_callback(Uint32 interval, void *not_used)
{
    SDL_Event e;

    e.type = SDL_USEREVENT;
    e.user.code = 0;
    e.user.data1 = NULL;
    e.user.data2 = NULL;
    SDL_PushEvent(& e);

    return interval;
}
```