Using SDL in C# Using the SDL.NET Library in Conjunction with TAO.NET

A Tutorial by Terry "Cibressus" Price

Terryp_AT_meldstar_DOT_com

4/10/2005

Foreword

I'm glad you have an interest in learning SDL. Thusly, because I'm not psychic I cannot tell if you will be in an influential position in a few years, therefore, I will take the chance to brainwash you all. Please push for hardware SDL drivers without the DX or GDI abstraction layer.

Introduction

I expect you to have a reasonable understanding of the C# language. Secondly, facility in high school math is a must. And lastly, do not expect this to be the only resource in your path to learning the SDL library; you should attempt to read as much as you possibly can. I've written this tutorial for the Windows32 platform, but similar techniques can be used on any platform. If you would like I give you free permission to port this. All code and this document is licensed under the GNU Public License (GPL).

Libraries and References

We will need 4 libraries.

- SdlDotNet
- System
- System.Drawing
- Tao.Sdl

Each of these libraries can be either found in your default library folder or at the SDL.NET site (link given in appendix). Please add these libraries to your references list.

Remember, windows looks in the application directory first for the DLL then the System32 directory, so you can put the DLL’s at either place, but for convenience I suggest the System32 Directory in your windows folder. We will compile this application as a windows application.

Design - Methods

Our first framework will be very simplistic; it will be designed more or less procedurally. All though this will upset many of the OO freaks out their, it will suffice for now, remember, where not making a game here, were learning how to use SDL. Later on we will create another framework.

So, let's create our class, I'm calling it SimpleExample, but feel free to call it whatever you want. Now within our class let us make some functions.

Constructor

```
public SimpleExample()
```

This function will initialize our program.

Key Handler

```
private void KeyDown(object o, KeyboardEventArgs e)
```

This is an event handler that handles keyboard presses.

Quit Handler

```
private void Quit(object s, QuitEventArgs e)
```

This is an event handler that handles the application wanting to quit. Make sure you don't put any cleanup code in here; it might take an entire loop before the application actually quits.

Application Loop

```
public void Run()
```

This is where all our actual rendering is done, and where our application actually does stuff, we go to this because we need to get into a non static method.

Main

```
public static void Main()
```

This is of course our application entry point. We need to get out of this into a non-static method.

Design – Members

The following is a table of the data members we will need and their function.

Type	**Name**	**Value**	**Usage**
private const int	width	640	The width of our application
private const int	height	480	The height of our application
private cont int	BPP	32	How much memory for color precision for each pixel
private bool	quit	Not Initialized	Does the user want to quit?
private Random	rand	Not Initialized	For random number generation
private Surface	screen	Not Initialized	See description below

The last member probably stood out a bit. So, just what is a surface? A surface is basically an area of memory that you do all your rendering to. So, your monitor could be a surface, a window, or even a texture on a 3d object. But in this tutorial a surface will be a window which we render to.

Initialization

Let's go back to our constructor. The first thing we want to do is subscribe our keyboard event handler. We do this through the += operator on Events.KeyboardDown. This should look something like this.

```
Events.KeyboardDown += new KeyboardEventHandler(this.KeyDown);
```

This should be pretty self explanatory all we are doing is creating a new keyboard event handler that calls KeyDown whenever there is a new KeyboardDown event. We then add that to the event list.

Do the same thing for the quit event, except add it to the Events.Quit instead of Events.Keydown, and it's a QuitEventHandler instead of a KeyboardEventHandler.

```
Events.Quit += new QuitEventHandler(this.Quit);
```

Now we must set our screen surface to be just that, our screen. We do this through the fuction

```
Surface Video.SetVideoModeWindow(int width, int height, bool
frame);
```

The width and height are simply the size of your window. The frame is weither or not your application has a title bar and resize handles arround it. Simply pass in your width, height, and whether or not if you want a frame, in this case I do. Remember that this returns a surface, so that's how we give our surface those properties.

```
screen = Video.SetVideoModeWindow(width, height, true);
```

Now lets just seed our random number generator against time.

```
rand = new Random();
```

Event Handlers

We have two event handlers one for when the user presses a key, and the other for when the user wants to quit. For our first examination lets implement a way for the application to quit when the user presses a key.

```
    private void KeyDown(object s, KeyboardEventArgs e)
    {
        if (e.Key == Key.Escape)
        {
            quit = true;
        }
    }
```

KeyboardEventArgs contains a member called Key which contains the key pressed which caused the event. It is stored in the form of Key. so, we check to see if the key pressed was escape. Key contains a list of keys that could have been pressed, the one that interests us is the Escape key. If the escape key has been pressed then we want to quit.

Our next event handler is our quit event handler. Make sure that you don’t put any cleanup code in here, if you do it may take a entire loop through the execution cycle before it actually quits, and that means that it is possible for the application to try and acess a resource that has already been cleaned up.

```
private void Quit(object s, QuitEventArgs e)
{
      quit = true;
}
```

As you can see here all were doing is setting quit to true. You could do other things like ask if the user is sure he or she wants to quit.

Application Run Loop

```
   public void run()
    {
        while (quit == false)
        {
            while (Events.Poll())
            {
            }
            screen.Fill(Color.FromArgb(rand.Next(255),
                          rand.Next(255),rand.Next(255)));
            System.Threading.Thread.Sleep(100);
            screen.Unlock();
            screen.Flip();
        }

    }
```

This is our application loop. It is quite literally a loop. The logic goes something like this. While we don’t want to quit, check if we want to quit, then do our rendering.

Our first function

```
public bool Events.Poll()
```

In SDL whenever an event occurs like mouse movement or the use minimizes the application it is stored on a queue this function removes the top event, dispatches it to the event handler if there is one, and returns true if there are more events on the stack. Thusly, we need to poll while there are more events on the queue. One the queue is empty, it returns false, and exits the while loop.

Before we do any rendering we must lock our screen. This prevents the screen from drawing halfway through and us getting a bad flickering effect. To do this use the Surface.Lock() function, and to unlock it use the Surface.Unlock() function. In this application our surface is screen, so it’ll look something like this.

```
screen.Lock();
//rendering
screen.Unlock()
```

Now were ready to do some rendering. Let's fill the screen in a random color. We only need one function in SDL to do that.

The Surface data type contains a method called

```
public void Fill(Drawing.Color color)
```

So, lets do that to our screen surface.

```
screen.Fill(Color.FromArgb(rand.Next(255),
            rand.Next(255),rand.Next(255)));
```

All we are doing is using the System.Drawing.Color.FromArgb() function to convert a random RGB value with a max of 255,255,255 and a min of 0,0,0 from bytes to Drawing.Color.

We then wait 100 MS with the sleep function. To avoid giving our selves a epileptic seizure.

```
System.Threading.Thread.Sleep(100);
```

Before we can start rendering the next frame we need to flip the back buffer with the function

```
public void surface.flip()
```

Remember that this has to be done on our surface so in this example it'll be

```
screen.Flip();
```

In the old days before back buffers, all the rendering and drawing had to be done in one area of memory, as a result you would either display as you drew or not display as you drew, either way gave you a bad flickering effect, because it was no longer a smooth transition. As computers memories increased, their was so much memory that now the computer could hold two screen at the same time. So now, as the computer draws to one screen, it displays another, the "back side". When that screen is done drawing it "flips" the screen and draws on the screen that was just displayed while displaying the one it was drawing to. As a result flickering was reduced because there is a smooth transition between one frame and another, and you no longer have to show nothing or something drawing.

To Test

```
public static void Main(string[] args)
{
    SimpleExample t = new SimpleExample();
    t.Run();
}
```

This is of course our main function, all we are doing is instantiating our class and calling the run method. Compile and run, and revel at the shiny flashiness of your new program.

Afterword

I hope this gave you some good insight into how SDL worked, I encourage you stick with it for a week, and read lots of other resources, some of which can be found in the appendix. You can view the code at www.ktfi.org/~cibressus/sdl.rar

Apendix

www.libsdl.org
www.gamedev.net
www.msdn.com
www.robloach.net
www.cibressus.info
www.cs-sdl.sourceforge.net

Adknowledgments

Thanks to of course, my self, Rob Loach, Chris “ColdAcid” Charabauk, Gamedev.net the midnight crew of the #gamedev chatroom on afternet and of course the Yellow Dart.