SDL.NET Programming Tutorial by Example

(c) 2008 Egon A. Rath

Please note that this Tutorial is not for the absolute beginner. I assume that you already can code fluent in C# and that you are able to understand code when reading them. I will not explain every single line of code when it should be clear what it's doing. Also please note that i am not a native speaker, so there are for sure many grammatical errors in this text (which are maybe corrected in the future when i find them ;-)

Table of Contents:

Overview

What is SDL.NET?

SDL is a Cross Platform Library for creating Games. It was developed during 1999-2001 by Sam Lantiga while he was working for Loki Software, a Company specialised in Porting

Windows Games to Linux. It provides you with the following Functionality:

- Bitmap Blitting
- Audio Output
- Video Playback
- Networking

Where available, Hardware Acceleration is used.

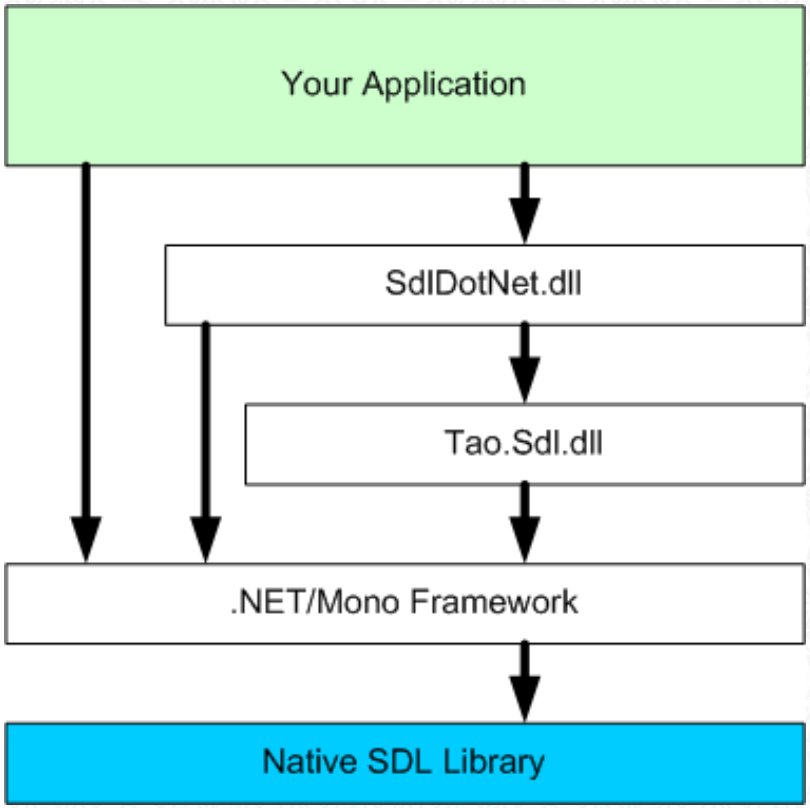
Setting up SDL.NET

To use SDL.NET you need to download the latest Release from it's Project Homepage located at http://cs-sdl.sourceforge.net. The downloaded package consists of a few DLL's with the following purposes:

File	Native/Managed	Description
SdlDotNet.dll	Managed	The actual implementation of the SDL.NET Library
Tao.Sdl.dll	Managed	Wrapper over native SDL
SDL_*.dll	Native	The native DLL's implementing the SDL Library

There are many more DLL's installed for some additional functionality like loading different image's. If you are using Linux you only get the managed Libraries, leaving the Task of obtaining the native SDL Library to the User (obtain it via your preffered Update mechanism). Some may ask why there are two managed DLL's: The Tao.Sdl.dll is a direct wrapper of the C-based SDL Library using Platform Invoke. The SdlDotNet.dll is a Object-Oriented wrapper over the first one and makes working with SDL easier and more comfortable.

The big Picture:

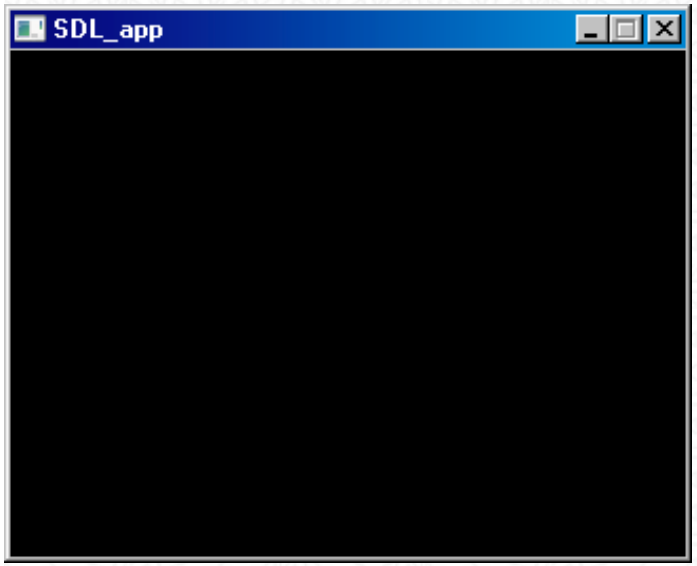


To use SDL.NET make sure you have referenced the two managed DLL's in your Project and the native DLL's (or Shared Objects) are available (On Windows: Directory where your Application resides or a Search Path; On Linux: Configured in /etc/ld.so.conf (or whatever your distribution uses))

Creating your first Window

```
using System;
using SdlDotNet.Graphics;
using SdlDotNet.Core;

public class SDL
{
    public static void Main( string[] args )
    {
        Video.SetVideoMode( 320, 240, 32, false, false, false, true );
        Events.Run();
    }
}
```

When compiled and run, you will see the following Window:



Nothing fancy, but hey, it's your first Window created with SDL. Let's go through the Code to see what's happening (only the essential part's of course, i assume that you are fluent in C#)

```
Video.SetVideoMode( 320, 240, 32, false, false, false, true );
```

Created a new SDL Surface with the specified size and the desired color deepth. We also specify the following boolean parameters:

- Resizable: False - We don't want our Window to be User resizable
- OpenGL Surface: False - We don't want that OpenGL is able to render to our Surface
- Fullscreen: False - We just want a Window managed by the Operating System
- Hardware Surface: True - Of course we want a Surface which resides in the Video Memory of our Graphics Cards. You should always create Hardware Surfaces when possible for the sake of performance

Events.Run();

Starts the Event Loop which fires the following Events:

- User Input Events (Keyboard, Mouse, Joystick)
- Frame Tick Event (per Default fires with the Rate of 1/(Refreshrate of your Screen). In my case

every 1/60 seconds)
- Graphic Events (Video Resize, Video Expose)
- Audio Events (Channel play finished, Sample play finished)
- Other Events (Application Active, Application Quit)

You have probably noticed that the Application does not Quit when the Window is closed. In the next step we will subscribe to the appropiate Event to handle this:

```
using System;
using SdlDotNet.Graphics;
using SdlDotNet.Core;

public class SDL
{
    public static void Main( string[] args )
    {
        Video.SetVideoMode( 320, 240, 32, false, false, false, true );
        Events.Quit += new EventHandler<QuitEventArgs>( ApplicationQuitEventHandler );
        Events.Run();
    }

    private static void ApplicationQuitEventHandler( object sender, QuitEventArgs args )
    {
        Events.QuitApplication();
    }
}
```

Congratulations, you have created your first fully functional SDL Application. Let's move on to get something fancy on your Screen.

Surface Loading and Blitting

Before we can go further, we have to know some Terms and what they mean:

Term	Description
Surface	Something you can draw on and contains Pixel Data. Can be either located in Main Memory or in Video Memory
Blit	Stands for "Block Image Transfer" and means: Copy a source surface to destination surface at the specified position

Now we are going to create a new surface, blit a Background image into it and then Blit a Foreground image above. The following images are being used for our Example:

The Background Image

The Foreground Window

The Result

```
using System;
using System.Drawing;
using SdlDotNet.Graphics;
using SdlDotNet.Core;

public class SDL
{
    private static Surface m_VideoScreen;
    private static Surface m_Background;
    private static Surface m_Foreground;
    private static Point m_ForegroundPosition;

    public static void Main( string[] args )
    {
        m_VideoScreen = Video.SetVideoMode( 320, 240, 32, false, false, false, true );

        LoadImages();

        Events.Quit += new EventHandler<QuitEventArgs>( ApplicationQuitEventHandler );
        Events.Tick += new EventHandler<TickEventArgs>( ApplicationTickEventHandler );
        Events.Run();
    }

    private static void LoadImages()
    {
        m_Background = ( new Surface( @"D:\temp\DemoBackground.png" )).
Convert( m_VideoScreen, true, false );
```

```
        m_Foreground = ( new Surface( @"D:\temp\DemoForeground.png" )).
Convert( m_VideoScreen, true, false );
        m_Foreground.Transparent = true;
        m_Foreground.TransparentColor = Color.FromArgb( 255, 0, 255 );
        m_ForegroundPosition = new Point( m_VideoScreen.Width/2 - m_Foreground.Width/2,
                                          m_VideoScreen.Height/2 - m_Foreground.Height/2 );
    }

    private static void ApplicationTickEventHandler( object sender, TickEventArgs args )
    {
        m_VideoScreen.Blit( m_Background );
        m_VideoScreen.Blit( m_Foreground, m_ForegroundPosition );
        m_VideoScreen.Update();
    }

    private static void ApplicationQuitEventHandler( object sender, QuitEventArgs args )
    {
        Events.QuitApplication();
    }
}
```

What are we doing in this Code:

1. Creating the Video Surface
2. Creating the Background Surface from a PNG File
3. Creating the Foreground Surface from a PNG File
4. Blitting all together every Frame

```
m_Background = ( new Surface( @"D:\temp\DemoBackground.png" )).Convert
( m_VideoScreen, true, false );
```

We load the File "DemoBackground.png" from Disk and create a new, compatible Surface. This is necessary because the loaded File may not have the same Format (Pixel Deepth) as our Display Surface. Without creating a compatible Surface we would end up with a automatic conversion every time we blit the surface to the target - which would decrease performance dramatically. Fortunately the Surface Class has a Method "Convert" which creates the needed Surface with additional Parameters:

- Hardware Surace: True - Create the new Surface in Video Memory
- Alpha Blending: False - We do not want Alpha Blending (more on this later)

You can load Images from a huge number of different File formats: BMP, PNM, XPM, LBM,

PCX, GIF, JPEG, TGA and PNG. I would recommend you to use PNG because it has some advantages over the other formats: Support for Alpha Channels and relatively small file sizes.

```
m_Foreground.Transparent = true;
```

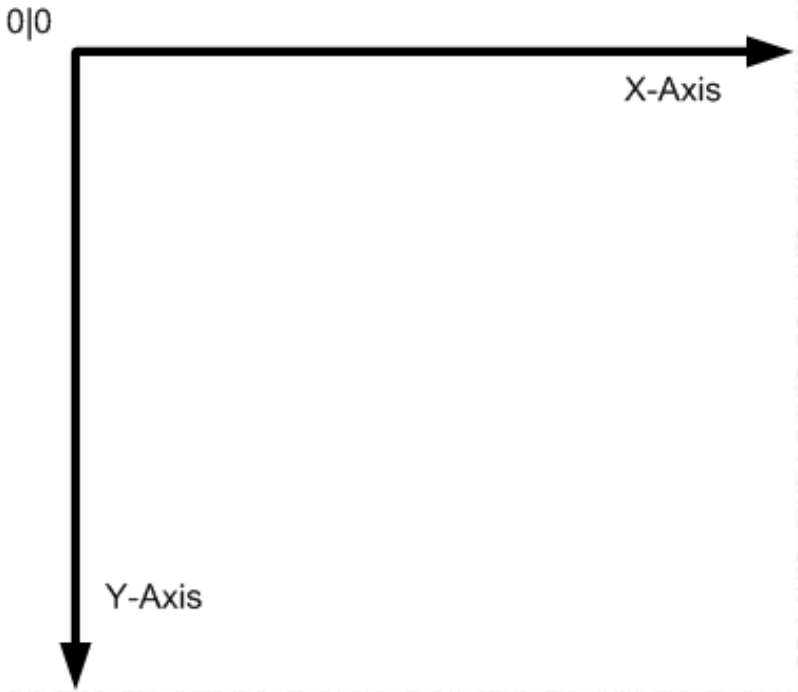
We specify that the Surface has Transparency Enabled (Alpha). Only if this option is set you can use the following statement:

```
m_Foreground.TransparentColor = Color.FromArgb( 255, 0, 255 );
```

which states that when Blitting the Surface, the Color R=255, G=0, B=255 is transparent. As you can see from the above images, the Foreground has a Magenta Background which is not shown in the Final image.

```
m_VideoScreen.Blit( m_Foreground, m_ForegroundPosition );
```

Blits the Foreground Surface to the Video Screen Surface. Additionally we specify the Position at which to Blit. Please note that the Coordinate System of SDL starts at the upper left corner:

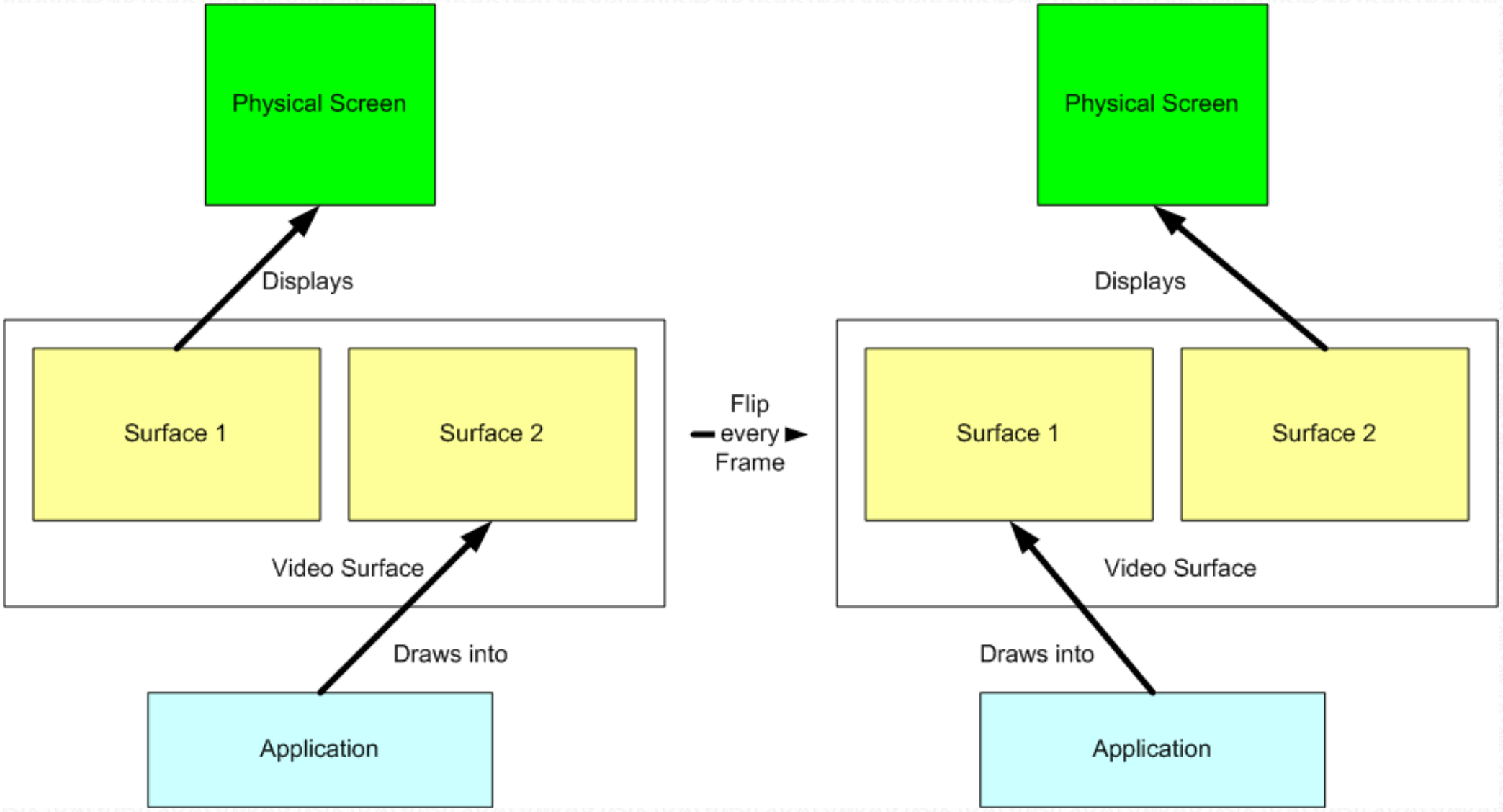


The last Thing we have to do every Frame is to update our Video Surface to show up the changes

we made. We do this with:

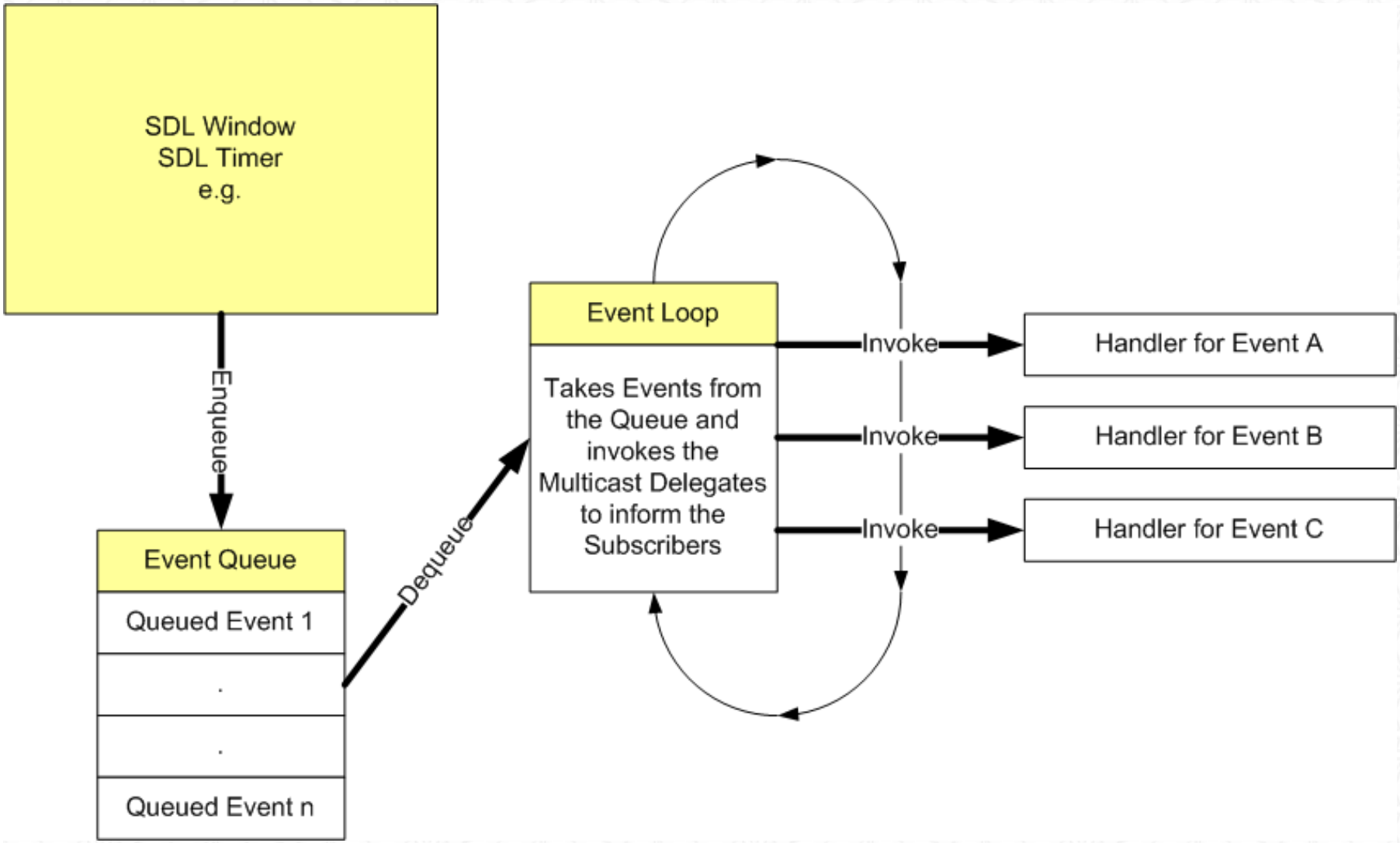
```
m_VideoScreen.Update();
```

When Double Buffering is enabled (Default) we end up with two Surfaces in the Background - from which only one is shown on the Screen and the other one is used as a Back Buffer to draw into. Without Double Buffering we would encounter Flickering on the Screen when Drawing to our Surface.



The Event Queue

If you developed Graphics User Interfaces before (like Windows Forms or GTK#) you probably know the mechanics of the so called Event Queue. If not, take a look at the following picture, think about it and read the description below



Whats happening inside the Event System is the following:

- Every time something happens (like the User is pressing a Mouse Button or a Key, a Timer Tick occured or the Window has been resized) a Event is generated by SDL and stored in the

Event Queue. This Data Structure is just a plain Queue which stores the Information.
- The Event Loop dequeues on Event from the Queue, Generates a proper Arguments class instance (like TickEventArgs) and invokes the Multicast Delegate, so your Application can respond to the Event. Then the Event Loop moves on to the next Item in the Queue.

In SDL.NET the Event Loop is implemented as a static Class. You have to execute Events.Run() to start the Loop processing the Enqueued Events. There are a huge amount of different Events available for subscription, but for now it's enough to know the following ones (because that's the only ones we used until now)

- Tick
  - Is Fired every time a Tick has occured. The Tick is like a Timer in your Application which is periodically called according to the desired Frame Rate. The Events Class has a Property called "FPS" which sets the desired Framerate. If you set FPS for example to 25 you end up with your Tick Event triggered 40 times per Second (1000ms / 25fps = 40 times). Please note that the Tick Event is fired exactly all 40ms. If your Handler Method takes 25ms, the Event Loop will wait 15ms before fireing the Event again.
- Quit
  - Is Fired when the User closes the SDL Window.

Additionally to the Events there are a few Methods in the Events Class which can be used in your Application. The most important is probably "QuitApplication()" which generates a Quit Event and enqueues it in the Event Queue (the Application will close). See the Documentation for further Details on the Events Class.

Color Keying, Alpha and Alpha Channels

When you are Blitting Surfaces, most of the time you probably don't want to blit the entire rectangular surface. Instead you want some parts transparent or semi transparent (e.g. for Sprites). This is where the above things take into account:

- Color Keying
  - Specifies a color which is ignored during Blitting operations. We used this technique before in the "Hello World!" Example to Blit only the Text Element of the Image and not the Background.
- Alpha
  - Specifies how Transparent or Opaque the entire Surface is.
- Alpha Channel
  - The Alpha Channel is something special compared to the other Methods. Normally, Images

have three Colors for each Pixel: Red, Green and Blue (RGB-Colorspace). Some Image Formats can have an additional "Alpha-Channel" which is essentially a Layer with a deepth of 1 byte per pixel. This byte represents how transparent the according pixel is (0 for opaque till 255 for transparent). The Alpha Channel is also known as "Per-Pixel Alpha"

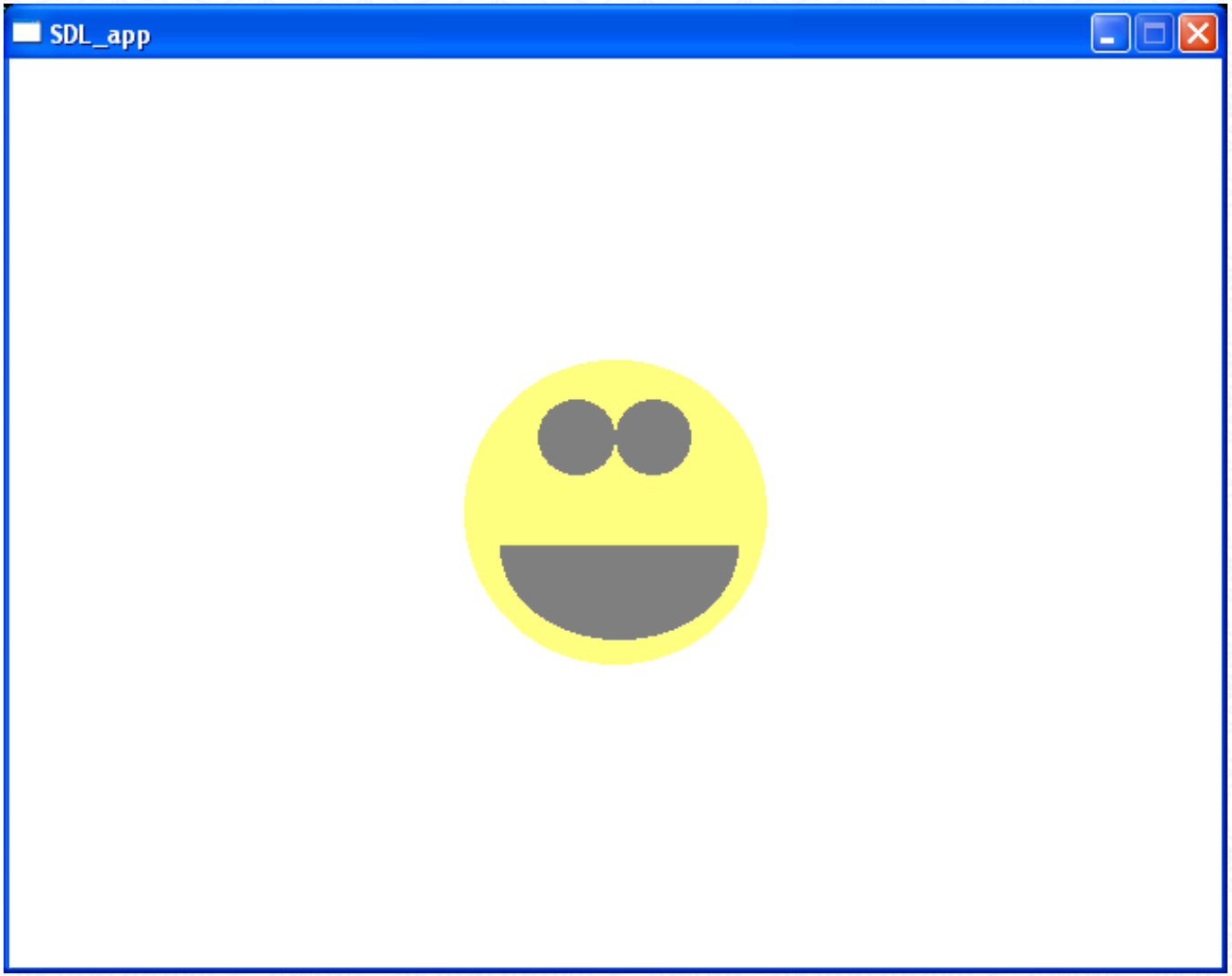
Now that you know the Terms, let's use them. Because we've used Color Keying before i will start with Alpha.

The following Image should be placed on white background and the magenta background color should not be visible. Additionally we want the surface 50% transparent:

```
using System;
using System.Drawing;
using SdlDotNet.Graphics;
using SdlDotNet.Core;

public class AlphaDemo
{
    private Surface m_VideoSurface;
    private Surface m_Sprite;

    public static void Main( string[] args )
    {
        new AlphaDemo();
    }

    public AlphaDemo()
    {
        m_VideoSurface = Video.SetVideoMode( 640, 480, false, false, false, true );
        m_Sprite = new Surface( @"SmileyNoAlphaChannel.png" ).Convert
```

```
( m_VideoSurface, true, true );
        m_Sprite.Alpha = 128;
        m_Sprite.AlphaBlending = true;
        m_Sprite.Transparent = true;
        m_Sprite.TransparentColor = Color.FromArgb( 255, 0, 255 );
        Events.Quit += new EventHandler<QuitEventArgs>( delegate( object
sender, QuitEventArgs args )
        {
            Events.QuitApplication();
        } );
        Events.Fps = 25;
        Events.Tick += new EventHandler<TickEventArgs>( delegate( object
sender, TickEventArgs args )
        {
            m_VideoSurface.Fill( Color.White );
            m_VideoSurface.Blit( m_Sprite, new Point( m_VideoSurface.Width/2 -
m_Sprite.Width/2,
                m_VideoSurface.Height / 2 - m_Sprite.Height / 2 ));
            m_VideoSurface.Update();
        } );
        Events.Run();
    }
}
```

When we run the Application the following Screen is shown:



Looks familiar, eh? But we have a few new lines of code:

```
m_Sprite.Alpha = 128;
m_Sprite.AlphaBlending = true;
```

specifies that this Surface has an Alpha of 128. Because 0 is opaque and 255 is fully transparent we end up with a transparence of 50%. Additionally we have to tell SDL that we want Alpha on this Surface by setting AlphaBlending to true. The next thing we will take a look at is the Alpha Channel. But first a little comparison of an image with, and another without an Alpha Channel:

This image has no Alpha channel. The Borders are hard because there is no blending with the Background

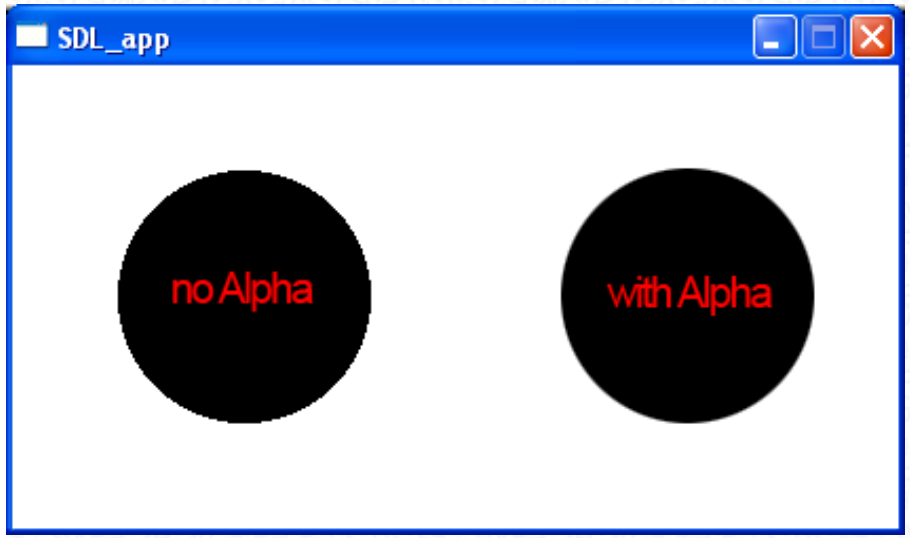
This image has an Alpha Channel. You can see that some of the Pixels from the Border are blended with the Background.
If you want a smooth integration of your Sprites with the Background, use images with a Alpha Channel

Let's see how the two different images look in a little Application:

```
using System;
using System.Drawing;
using SdlDotNet.Graphics;
using SdlDotNet.Core;

public class AlphaDemo
{
    private Surface m_VideoSurface;
    private Surface m_SpriteNoAlphaChannel;
    private Surface m_SpriteWithAlphaChannel;

    public static void Main( string[] args )
    {
        new AlphaDemo();
    }

    public AlphaDemo()
```

```
    {
        m_VideoSurface = Video.SetVideoMode( 420, 220, false, false, false, true );
        m_VideoSurface.AlphaBlending = true;
        m_SpriteNoAlphaChannel = new Surface( @"DotWithoutAlphaChannel.png" ).
Convert( m_VideoSurface, true, true );
        m_SpriteNoAlphaChannel.Transparent = true;
        m_SpriteNoAlphaChannel.TransparentColor = Color.White;
        m_SpriteWithAlphaChannel = new Surface( @"DotWithAlphaChannel.png" );

        Events.Quit += new EventHandler<QuitEventArgs>( delegate( object
sender, QuitEventArgs args )
        {
            Events.QuitApplication();
        } );
        Events.Fps = 25;
        Events.Tick += new EventHandler<TickEventArgs>( delegate( object
sender, TickEventArgs args )
        {
            m_VideoSurface.Fill( Color.White );
            m_VideoSurface.Blit( m_SpriteNoAlphaChannel, new Point( 10, 10 ));
            m_VideoSurface.Blit( m_SpriteWithAlphaChannel, new Point( 220, 10 ));
            m_VideoSurface.Update();
        } );
        Events.Run();
    }
}
```

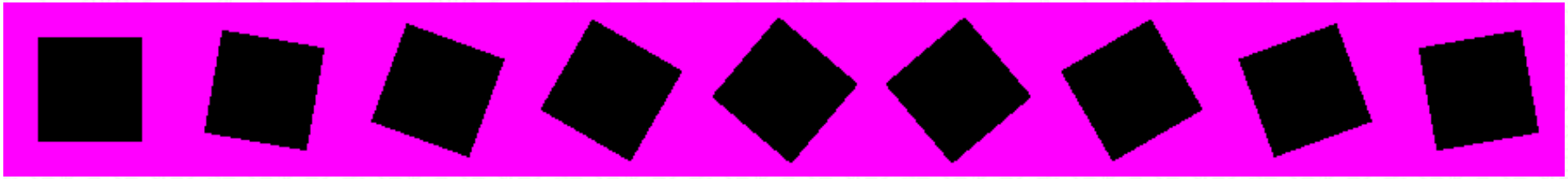
The resulting Screen looks like:



Because there is nothing new in the above source i will not explain it anymore. Please note that you can't convert the Surface with Alpha Channel information into the same Format as the Video Surface because you will loose the Alpha Information in this case. But that should be now problem because of the fact that Alpha Channels are rarely used in Games.

Clip Blitting and Sprite Sheets

When using a huge amount of Surfaces like when doing Animations it's pretty unhandy to use a single file for every sprite. That's the reason for so called Spritesheets, which are essentially nothing more than a huge Image with all sprites in it. As an example, let's consider a rotating cube (ok, you can use the Methods for rotating a sprite, but it's just an example). We will use the following Spritesheet:

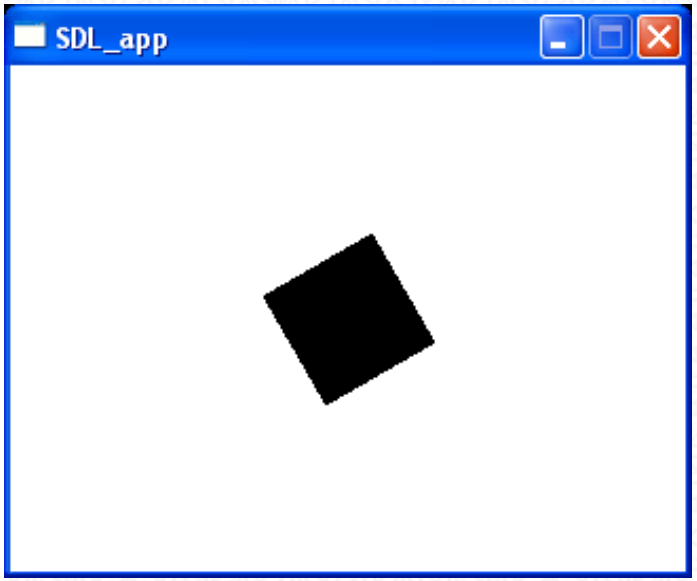


Every Sprite has a Dimension of 100x100 Pixels. We will load the entire Image into a Surface and Clip Blit every single Part of it onto our Video Surface.

```
using System;
using System.Drawing;
using System.Runtime.InteropServices;
using SdlDotNet.Graphics;
using SdlDotNet.Core;


public class SpriteSheetDemo
{
    private Surface m_VideoSurface;
    private Surface m_SpriteSheet;
    private int m_SpriteSheetOffset = 0;

    public static void Main( string[] args )
    {
        new SpriteSheetDemo();
    }

    public SpriteSheetDemo()
```

```
    {
        m_VideoSurface = Video.SetVideoMode( 320, 240, false, false, false, true );
        m_SpriteSheet = new Surface( @"SpriteSheet.png" ).Convert( m_VideoSurface,
true, true );
        m_SpriteSheet.Transparent = true;
        m_SpriteSheet.TransparentColor = Color.FromArgb( 255, 0, 255 );

        Events.Quit += new EventHandler<QuitEventArgs>( delegate( object
sender, QuitEventArgs args )
        {
            Events.QuitApplication();
        } );
        Events.Fps = 25;
        Events.Tick += new EventHandler<TickEventArgs>( delegate( object
sender, TickEventArgs args )
        {
            if( m_SpriteSheetOffset == 900 )
                m_SpriteSheetOffset = 0;

            m_VideoSurface.Fill( Color.White );
            m_VideoSurface.Blit( m_SpriteSheet, new Point( m_VideoSurface.Width/2 -
50, m_VideoSurface.Height / 2 - 50 ),
                new Rectangle( new Point( m_SpriteSheetOffset, 0 ), new Size( 100, 100 )));
            m_VideoSurface.Update();

            m_SpriteSheetOffset += 100;
        } );
        Events.Run();
    }
}
```

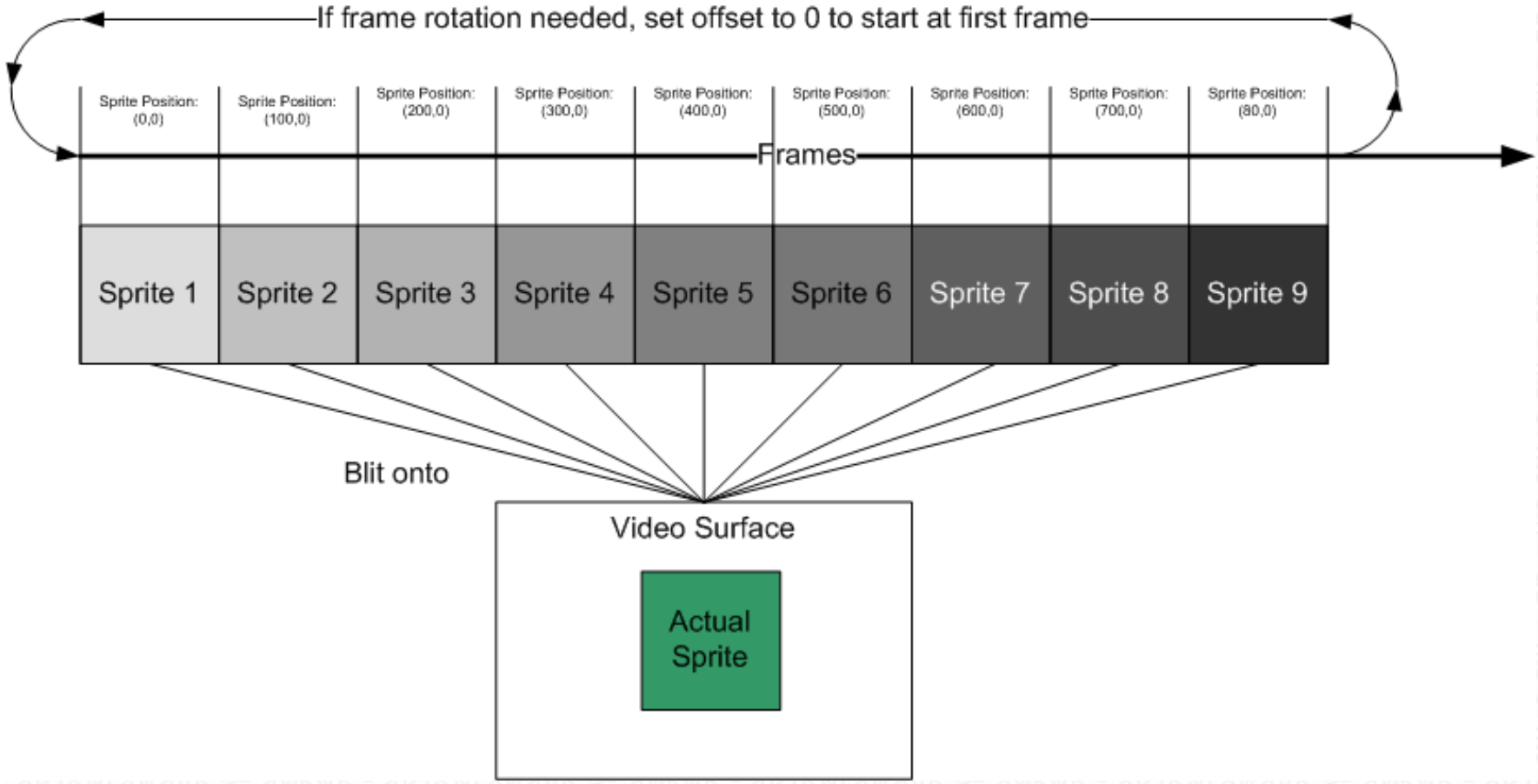
When you run the application you will see a Window with a white Background and a Rotating Cube:



Let's take a look at what's going on inside our Application at a High level:

- We load our Spritesheet into a Hardware Surface (when possible)
- At every Frame, we increase a offset variable with the with value of every sprite and generate a new Rectangle which reflects the actual sprite inside the Spritesheet.
- We blit this Rectangular Area to our Video Surface
- If the offset has reached the with of our Spritesheet, reset them to zero to start with the first sprite again.

If things are not clear yet, take a look at the picture below which visually shows what happens:



When you intend to use a Spritesheet, i would recommend you to create a Sprite Manager Class which can manage the Process of retrieving the Surface from the Image by using the Sprite's Name. For this to work, create a Textual Representation of the Sprites within the Spritesheet:

```
#
# Sprite Description Map
#
# Name;Offset X;Offset Y;Width;Height
sprite1;0;0;100;100
sprite2;100;0;100;100
sprite3;200;0;100;100
sprite4;300;0;100;100
sprite5;400;0;100;100
sprite6;500;0;100;100
sprite7;600;0;100;100
sprite8;700;0;100;100
```

```
sprite9;800;0;100;100
```

The Map describes which sprite is at which position in the Sheet and what's his dimension. Your Manager class has the sole role of returning you the wanted image from a given Sprite's Name (The actual implementation is left to the reader)

Maybe you have asked yourself how to create a Spritesheet. I would recommend you to use the Graphic Editor of your choice (for example Adobe Photoshop, GraphicsGale or Cosmigo ProMotion) and create the Sprites each after another. Then combine them in a new Image. You have to take care of a few things when doing this (will make your life a lot more easier):

- Align all Sprites on a common Baseline (per Line)
- Group Sprites which have a relation together (e.g. don't mix up the Sprites of an animation with some other content)
- Choose a good Background!
  - You can do this by inverting the Sprite's Image and choose one of this colors.

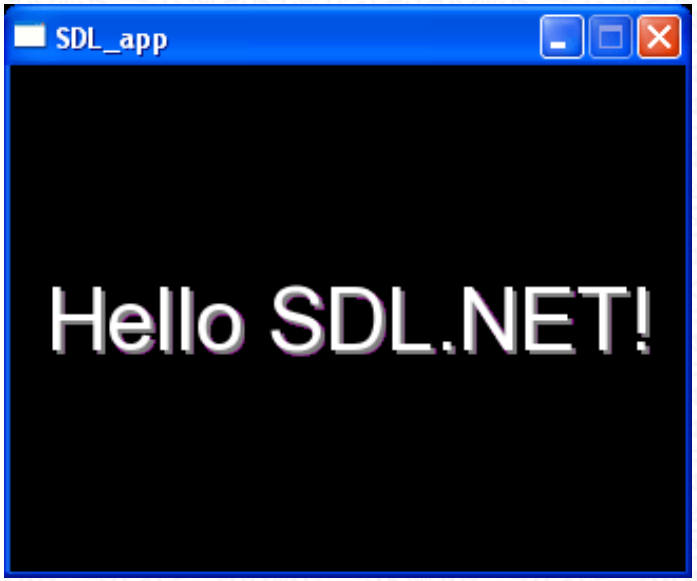
Normal Sprite *Inverted Sprite* *Sprite with Background*

Credit goes to Grim from the Sprite Database for these Images

True Type Fonts

Normally you not only need Sprites and Bitmaps in your Application but also Text in various Forms. There are two different methods on how to use Fonts in a SDL Application: TrueType and Bitmap Fonts. In this section i will cover TrueType fonts and how to use them. TrueType fonts are always rendered into a Surface and then blitted.

We will create a little Application which shows two rendered Text Surfaces, one containing the actual Text and the other one the shadow:




```
using System;
using System.Drawing;
using System.Runtime.InteropServices;
using SdlDotNet.Graphics;
using SdlDotNet.Core;

public class FontDemo
{
    private Surface m_VideoSurface;
    private Surface m_FontForegroundSurface;
    private Surface m_FontShadowSurface;

    public static void Main( string[] args )
    {
        new FontDemo();
    }

    public FontDemo()
    {
        m_VideoSurface = Video.SetVideoMode( 320, 240, false, false, false, true );

        SdlDotNet.Graphics.Font font = new SdlDotNet.Graphics.Font( @"Arial.
ttf", 42 );

        // Create the Font Surfaces
        m_FontForegroundSurface = font.Render( "Hello SDL.NET!", Color.White );
        m_FontShadowSurface = font.Render( "Hello SDL.NET!", Color.White, Color.
FromArgb( 255, 0, 255 )).Convert( m_VideoSurface, false, true );
```

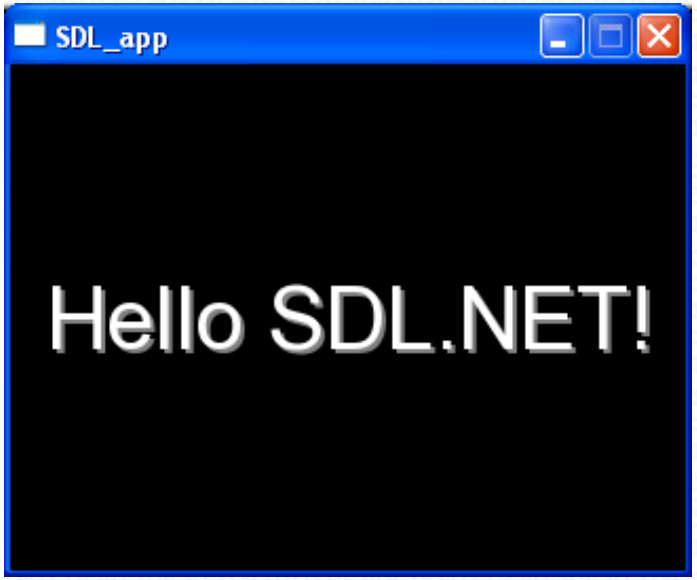
```
        m_FontShadowSurface.Alpha = 128;
        m_FontShadowSurface.AlphaBlending = true;
        m_FontShadowSurface.Transparent = true;
        m_FontShadowSurface.TransparentColor = Color.FromArgb( 255, 0, 255 );

        Events.Quit += new EventHandler<QuitEventArgs>( delegate( object
sender, QuitEventArgs args )
        {
            Events.QuitApplication();
        } );
        Events.Fps = 25;
        Events.Tick += new EventHandler<TickEventArgs>( delegate( object
sender, TickEventArgs args )
        {
            m_VideoSurface.Fill( Color.Black );

            m_VideoSurface.Blit( m_FontShadowSurface,
                new Point( m_VideoSurface.Width/2 - m_FontShadowSurface.Width/2 +2,
                           m_VideoSurface.Height/2 - m_FontShadowSurface.Height/2 +2 ));

            m_VideoSurface.Blit( m_FontForegroundSurface,
                new Point( m_VideoSurface.Width / 2 - m_FontForegroundSurface.Width / 2,
                           m_VideoSurface.Height / 2 - m_FontForegroundSurface.
Height / 2 ));

            m_VideoSurface.Update();
        } );
        Events.Run();
    }
}
```

When you take a close look at the image you will probably notice the Magenta colored artifacts on the shadow. This is because of the way we use create the shadow and on the other side by the effect that the renderer always creates antialiased surfaces:

- We create a Surface with the rendererd Font in. This image has per-Pixel Alpha and is antialised, so the text looks good on every background.
- We create another Surface with the rendered Font in. Because this will be the shadow, we have to specify a Background color and convert the Image to use per Surface Alpha instead of per-Pixel.
- We have to make the Background Transparent and set the Alpha of the Surface
- Because the Text is antialised we get artifacts (not all pixels around the text have the same color)

The sole reason i've showed you this example was because i wanted you to show that you can create effects by using the Alpha ;-) If you really want True Type Fonts and shadows, use

the following method:




```
SdlDotNet.Graphics.Font font = new SdlDotNet.Graphics.Font( @"Arial.ttf", 42 );
m_FontForegroundSurface = font.Render( "Hello SDL.NET!", Color.White );
m_FontShadowSurface = font.Render( "Hello SDL.NET!", Color.FromArgb( 128, 128, 128 ));
```

This three lines of code are also the only ones which are really new for us:

1. Create a new Instance of the Font Class. You can either provide the Filename or a byte[] containing the File Data as a constructor parameter. The second parameters specifies the Font size in Pixels.
2. Render a Text with the given Fore- and Background Color into a new Surface

Keyboard Events

Nearly all Games need to handle Key presses in some way - like moving around a character or pausing the Game. In SDL Key-Presses are handled by the Event Queue which fires a Event every time a Key is pressed. There are two different Events associated with this:

1. KeyboardDown
2. KeyboardUp

as the Name implies, the first on is called when the User presses a Key, the second one when the

Key has been released. But let's go straight into a little example:

```
using System;
using System.Collections.Generic;
using System.Drawing;
using SdlDotNet.Graphics;
using SdlDotNet.Input;
using SdlDotNet.Core;

public class KeyboardTest
{
    private static Surface m_VideoScreen;
    private static Surface m_CursorSpriteSheet;
    private static Surface m_BackSurface;
    private static bool[] m_CursorKeys = new bool[4];
    private static int m_BackOffsetX = 0, m_BackOffsetY = 0;

    public static void Main( string[] args )
    {
        SetVideoDriver();

        m_VideoScreen = Video.SetVideoMode( 500, 500, 32, false, false, false, true, true );
        LoadImages();

        Events.Quit += new EventHandler<QuitEventArgs>( ApplicationQuitEventHandler );
        Events.Tick += new EventHandler<TickEventArgs>( ApplicationTickEventHandler );
        Events.KeyboardDown += new EventHandler<KeyboardEventArgs>( KeyboardEventHandler );
        Events.KeyboardUp += new EventHandler<KeyboardEventArgs>( KeyboardEventHandler );
        Events.Run();
    }

    private static void SetVideoDriver()
    {
        if( System.Environment.OSVersion.Platform == PlatformID.Win32NT )
            System.Environment.SetEnvironmentVariable( "SDL_VIDEODRIVER", "directx" );

        Console.Out.WriteLine( "Using Video Driver: {0}", Video.VideoDriver );
        Console.Out.WriteLine( "Hardware surfaces : {0}", VideoInfo.HasHardwareSurfaces );
        Console.Out.WriteLine( "Hardware blits    : {0}", VideoInfo.HasHardwareBlits );
    }

    private static void LoadImages()
    {
        m_CursorSpriteSheet = new Surface( @"ArrowsSheet.png" ).Convert
( m_VideoScreen, true, true );
        m_CursorSpriteSheet.Transparent = true;
```

```
        m_CursorSpriteSheet.TransparentColor = Color.FromArgb( 255, 0, 255 );

        m_BackSurface = new Surface( @"Background.png" ).Convert( m_VideoScreen,
true, true );
    }

    private static void KeyboardEventHandler( object sender, KeyboardEventArgs args )
    {
        switch( args.Key )
        {
            case Key.UpArrow:
                m_CursorKeys[(int)CursorKeys.Up] = args.Down;
                break;

            case Key.DownArrow:
                m_CursorKeys[(int)CursorKeys.Down] = args.Down;
                break;

            case Key.LeftArrow:
                m_CursorKeys[(int)CursorKeys.Left] = args.Down;
                break;

            case Key.RightArrow:
                m_CursorKeys[(int)CursorKeys.Right] = args.Down;
                break;

            case Key.Escape:
                Events.QuitApplication();
                break;
        }
    }

    private static void ApplicationTickEventHandler( object sender, TickEventArgs args )
    {
        List<Point> arrowDestPoints = new List<Point>();
        List<Rectangle> arrowSourceRects = new List<Rectangle>();

        if( m_CursorKeys[(int)CursorKeys.Up] )
        {
            arrowDestPoints.Add( new Point( m_VideoScreen.Width/2 - 50,
m_VideoScreen.Height/2 - 150 ));
            arrowSourceRects.Add( new Rectangle( 100, 100, 100, 100 ));
            if( m_BackOffsetY > 0 )
                m_BackOffsetY -= 5;
        }

        if( m_CursorKeys[(int)CursorKeys.Down] )
```

```
        {
            arrowDestPoints.Add( new Point( m_VideoScreen.Width/2 - 50,
m_VideoScreen.Height/2 + 50 ));
            arrowSourceRects.Add( new Rectangle( 0, 100, 100, 100 ));
            if( m_BackOffsetY < m_BackSurface.Height - m_VideoScreen.Height )
                m_BackOffsetY += 5;
        }

        if( m_CursorKeys[(int)CursorKeys.Left] )
        {
            arrowDestPoints.Add( new Point( m_VideoScreen.Width/2 - 150,
m_VideoScreen.Height/2 - 50 ));
            arrowSourceRects.Add( new Rectangle( 0, 0, 100, 100 ));
            if( m_BackOffsetX > 0 )
                m_BackOffsetX -= 5;
        }

        if( m_CursorKeys[( int ) CursorKeys.Right] )
        {
            arrowDestPoints.Add( new Point( m_VideoScreen.Width / 2 + 50,
m_VideoScreen.Height / 2 - 50 ));
            arrowSourceRects.Add( new Rectangle( 100, 0, 100, 100 ));
            if( m_BackOffsetX < m_BackSurface.Width - m_VideoScreen.Width )
                m_BackOffsetX += 5;
        }

        m_VideoScreen.Blit( m_BackSurface, new Point( 0, 0 ), new Rectangle
( m_BackOffsetX, m_BackOffsetY, m_VideoScreen.Width, m_VideoScreen.Height ));
        for( int index = 0; index < arrowDestPoints.Count; index ++ )
        {
            m_VideoScreen.Blit( m_CursorSpriteSheet, arrowDestPoints
[index], arrowSourceRects[index] );
        }
        m_VideoScreen.Update();
    }

    private static void ApplicationQuitEventHandler( object sender, QuitEventArgs args )
    {
        Events.QuitApplication();
    }

    private enum CursorKeys : int
    {
        Up = 0,
        Down = 1,
        Left = 2,
        Right = 3
```

```
    }
}
```

seems a little bit hughe at the first look, but there are many things we do in this example.
It implements a Surface which can be scrolled using the Cursor Keys:

What we are doing at a glance is the following:

- When the User presses a Key the Handler is called which checks if it's a Cursor Key. If this condition is met, we set a flag in a bool[4] to save the State (true=pressed,false=not pressed/released)
- In the Tick Method we check which keys are pressed and build List of Points (the Positions where to Blit the Arrows to) and another List with the Source Rectangles for the Sprite Sheet.
- We set a X- and Y-Offset for the Background Surface according to the pressed Keys
- Finally we blit the Background Surface and the needed Arrows on the Screen

The interesting Object in this example is the KeyboardEventArgs parameter in our Event Handler. This class contains the necessary informations to handle the Keypress:

- Down
  - Is the Key down (True; Pressed) or up (False; Released)

- Key
  - The Key which was pressed. The possible Keys are defined in the "Key" Enumeration
- Scancode
  - The raw Code sent from the Keyboard when a Key is pressed or released. If the Key Enumeration does not contain the Key you want to check for, you can use the Scancode instead. Please note that not all Keyboards may have the same Keys, so don't use anything special when possible.
- Mod
  - The Modifiers which are currently pressed additionally (Alt, Ctrl, Shift...). The modifications are defined in the "ModifierKeys" enumeration and are OR'ed together in the Property.

There are more properties, but those above are the most important probably. Eventually you have noticed that the Event is only triggered when the Key is pressed and released, but not when the Key is hold down. This is because Key-Repeating is disabled by default in SDL. Use Keyboard.EnableKeyRepeat(delay,rate) to enabled this feature.

Sometimes it would be nice to know outside of the Keyboard Handler if a Button is pressed or not. Instead of setting a Variable somewhere in your Application, you can use the Method "IsKeyPressed" from the "KeyboardState" class. As the sole parameter you have to give one of the members of the Key Enumeration and it will give you back a bool (true=pressed, false=not pressed). It's a static Method so you don't have to create an instance for it's use.

Even if it has nothing to do with Keypresses, there is a line of code above which you haven't seen yet:

System.Environment.SetEnvironmentVariable("SDL_VIDEODRIVER", "directx");

This sets the Environment Variable SDL_VIDEODRIVER to the value "directx". This Environment sets the Output Driver SDL uses to put Graphics onto the Screen. In this case we will use DirectX (DirectDraw) when it's available. But theres another Section on this later.

Mouse Events

Working with the Mouse is (from the aspect of the Events) very similar to the Keyboard. A Mouse Event is triggered on the following conditions:

- The Mouse is beiing moved around in the Window (MouseMotion Event)
- One of the Mouse's Button is pressed or released (MouseButtonDown and MouseButtonUp)

I will show you an example which's solely puropos is to present the User with a White Background Window where he can draw Freeform Lines. We will also use a custom Cursor (Hand Shaped). The result will loke similar to this:

The Mouse cursor we use is the following one:

Using a different Cursor is nothing fancy. There are only a few steps involved in doing this:

1. Loading the new Cursor in a Surface and setting it's parameters (Transparency)
2. Disabling the Default Cursor
3. Every Frame, blit the Cursor Surface at the Position where the Mouse currently is (as the last operation, because we want the Cursor the Topmost Visual Object)

Let's see how this is implemented in real code:

```
using System;
using System.Collections.Generic;
using System.Drawing;
using SdlDotNet.Graphics;
using SdlDotNet.Input;
using SdlDotNet.Core;

public class KeyboardTest
{
    private static Surface m_VideoScreen;
    private static Surface m_DrawingSurface;
    private static Surface m_Cursor;
    private static Point m_CursorPosition = new Point();
    private static Point m_LastCursorPos = new Point();
    private static bool m_ButtonDown = false;

    public static void Main( string[] args )
    {
        m_VideoScreen = Video.SetVideoMode( 800, 600, 32, false, false, false, true, true );
        m_VideoScreen.Fill( Color.White );
        LoadImages();

        Events.Quit += new EventHandler<QuitEventArgs>( ApplicationQuitEventHandler );
        Events.Tick += new EventHandler<TickEventArgs>( ApplicationTickEventHandler );

        Mouse.ShowCursor = false;
        Events.MouseMotion += new
EventHandler<MouseMotionEventArgs>( ApplicationMouseMotionEventHandler );
        Events.MouseButtonDown += new
EventHandler<MouseButtonEventArgs>( ApplicationMouseButtonEventHandler );
        Events.MouseButtonUp += new
EventHandler<MouseButtonEventArgs>( ApplicationMouseButtonEventHandler );
        Events.Run();
    }

    private static void LoadImages()
    {
        m_Cursor = new Surface( @"Cursor.png" ).Convert( m_VideoScreen, true, true );
        m_Cursor.Transparent = true;
        m_Cursor.TransparentColor = Color.FromArgb( 255, 0, 255 );

        m_DrawingSurface = Video.CreateRgbSurface( m_VideoScreen.Width,
m_VideoScreen.Height, 32, 0, 0, 0, 0, true );
        m_DrawingSurface.Fill( Color.White );
    }
```

```
    private static void ApplicationMouseButtonEventHandler( object
sender, MouseButtonEventArgs args )
    {
        if( args.Button == MouseButton.PrimaryButton )
            m_ButtonDown = args.ButtonPressed;
        else if( args.Button == MouseButton.SecondaryButton )
            m_DrawingSurface.Fill( Color.White );
    }

    private static void ApplicationMouseMotionEventHandler( object
sender, MouseMotionEventArgs args )
    {
        m_CursorPosition = args.Position;
        m_CursorPosition.X -= 6; // Because the Visual Pointer of the Cursor is 6
pixels inside

        if( m_ButtonDown )
        {
            IPrimitive line = new SdlDotNet.Graphics.Primitives.Line
( m_LastCursorPos, m_CursorPosition );
            m_DrawingSurface.Draw( line, Color.Red );
        }

        m_LastCursorPos = m_CursorPosition;
    }

    private static void ApplicationTickEventHandler( object sender, TickEventArgs args )
    {
        m_VideoScreen.Blit( m_DrawingSurface );
        m_VideoScreen.Blit( m_Cursor, m_CursorPosition );
        m_VideoScreen.Update();
    }

    private static void ApplicationQuitEventHandler( object sender, QuitEventArgs args )
    {
        Events.QuitApplication();
    }
}
```

What's the logical flow of the Application:

- We subscribe to the Events associated with the Mouse
- We disable the default Mouse Button
- Whenever the User presses a Button, we set a bool Variable to have it's current state available for other operations

    - If it's the secondary Button (On most Mice the Right Button) we clear the Drawing Surface
- Whenever the Mouse moves - and the primary Button is pressed - we draw a Line from the last known Mouse Position to the actual one.

If you took a look at the MouseMotionEventArgs Object passed to the Event Handler you may have noticed that there are properties called "RelativeX" and "RelativeY". These properties represent the same thing as we did manually: Representing at which Point the Mouse was the last time the Handler was called.

Playing Sounds and Music

In every serious game you are confronted with music of different kinds like Sound Effects and Background Music. In SDL.NET you have a few choices on how to play sound in general:

- The Sound Class
    - Especially useful if the Sample you want to play is intended to be a sound effect (short playtime)
- The Music/MusicPlayer Class
    - Mostly for playing longer samples like Background Music

SDL is able to load some common music file formats which are: ogg, mp3, wav, mod and mid

Let's go straight into a example which will do the following things:

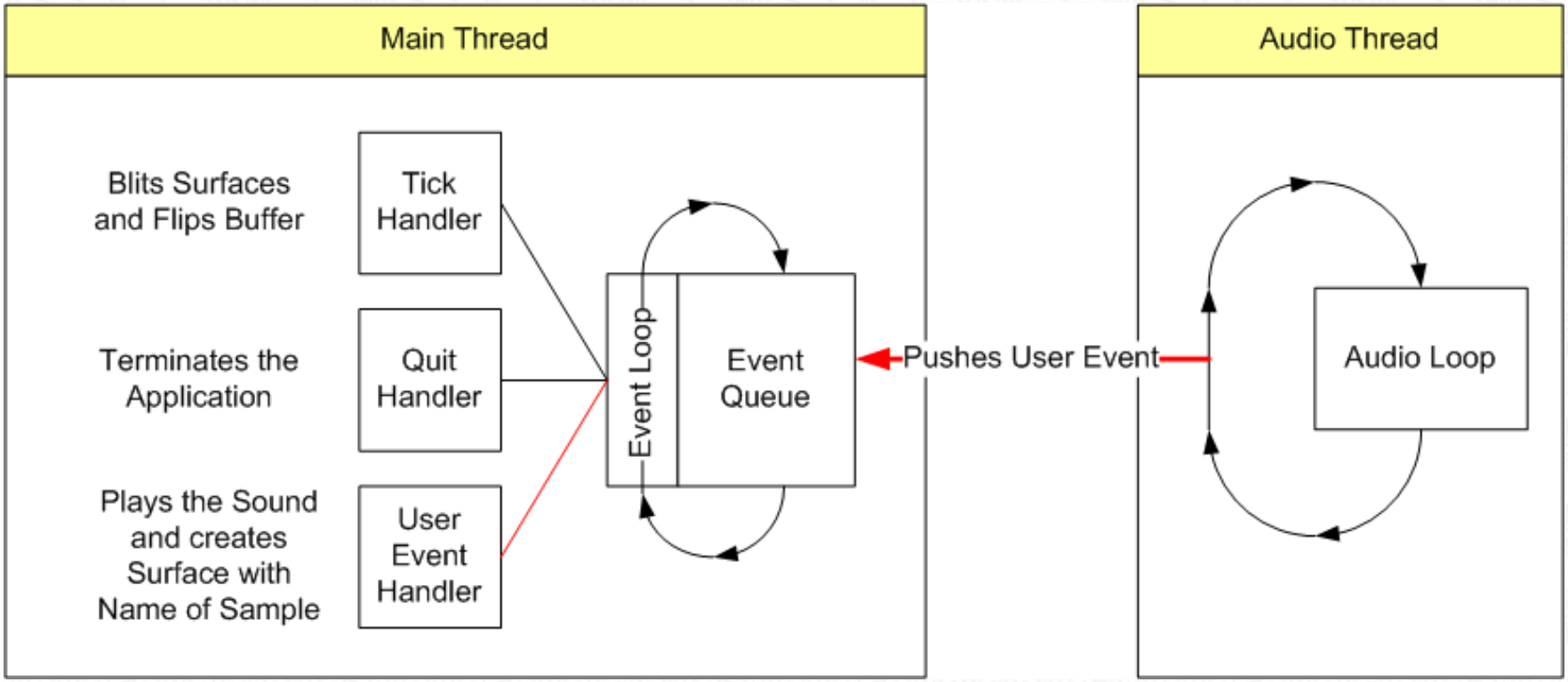
1. Creates a SDL window which shows what's currently playing
2. Plays a rather long Background Music Sample
3. Plays a total of four samples in a loop

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Threading;
using SdlDotNet.Graphics;
using SdlDotNet.Input;
using SdlDotNet.Core;
using SdlDotNet.Audio;

using Font = SdlDotNet.Graphics.Font;

public class AudioTest
{
```

```
    private static Surface m_VideoScreen;
    private static Surface m_FileNameSurface;
    private static Font m_Font = new Font( "Arial.ttf", 60 );
    private static bool m_Terminate = false;

    public static void Main( string[] args )
    {
        m_VideoScreen = Video.SetVideoMode( 800, 600, 32, false, false, false, true, true );

        Events.Quit += new EventHandler<QuitEventArgs>( ApplicationQuitEventHandler );
        Events.Tick += new
EventHandler<TickEventArgs>( ApplicationTickEventHandler );

        Events.UserEvent += new EventHandler<UserEventArgs>( ApplicationUserEventHandler );

        Thread audioThread = new Thread( new ThreadStart( AudioPlaybackThread ));
        audioThread.Start();

        Events.Run();
    }

    private static void AudioPlaybackThread()
    {
        // Play background Music
        Music bgMusic = new Music( "back.ogg" );
        MusicPlayer.Volume = 30;
        MusicPlayer.Load( bgMusic );
        MusicPlayer.Play();

        // Start playing sound Effects
        List<string> audioFiles = new List<string>( new string[] { "three.ogg", "two.
ogg", "one.ogg", "fight.ogg" } );
        int cnt = 0;
        while( ! m_Terminate )
        {
            UserEventArgs userEvent = new UserEventArgs( audioFiles[cnt++] );
            if( cnt >= audioFiles.Count )
                cnt = 0;

            Events.PushUserEvent( userEvent );

            SdlDotNet.Core.Timer.DelaySeconds( 2 );
        }
    }

    private static void ApplicationUserEventHandler( object sender, UserEventArgs args )
```

```
    {
        // Play Ogg File
        if(( args.UserEvent as string ).EndsWith( "ogg" ))
        {
            Sound snd = new Sound( args.UserEvent as string );
            m_FileNameSurface = m_Font.Render(( args.UserEvent as string ).Replace( ".
ogg", String.Empty ), Color.White );
            snd.Play();
        }
    }

    private static void ApplicationTickEventHandler( object sender, TickEventArgs args )
    {
        m_VideoScreen.Fill( Color.Black );
        if( m_FileNameSurface != null )
            m_VideoScreen.Blit(  m_FileNameSurface, new Point( m_VideoScreen.Width/2
- m_FileNameSurface.Width/2,
                m_VideoScreen.Height / 2 - m_FileNameSurface.Height / 2 ));
        m_VideoScreen.Update();
    }

    private static void ApplicationQuitEventHandler( object sender, QuitEventArgs args )
    {
        m_Terminate = true;
        Events.QuitApplication();
    }
}
```

If you go throught the example, you will probably encounter a few things new to you. We start a new Thread which loops through a generic List of strings and pushes a User Event onto the Queue every time the sample needs to be played. This is necessary because all SDL operations should occur in the same Thread as the Event Queue is running on. Visualize this:



Next to the default Events SDL provides to you (QuitEvent, TickEvent, e.g) there is also one called "UserEvent" which is only fired by the User itself - and is used to execute some functionality on the Main Thread in our case. We supply the Name of the file to play as the User-defined Object contained within the UserEventArgs object passed to the handler. The Handler then Plays the sample and creates a surface with the Name of the Sample. This surface is then Blitted onto the Screen in the Tick Event Handler. But let's move on to the parts really necessary to play audio:

The "Music" Class encapsulates a File for playing by the "MusicPlayer" Class. Please note that the "Play" Methods for audio are non blocking. What's necessary to Play a Music Sample:

1. Load the Sample into a Music Instance by passing it's name (or data in a byte[]) to it's constructor
2. Load the Music Instance into the MusicPlayer (Load Method)
3. Set up necessary Parameters like the Volume
4. Execute the Play Method

It's nearly the same for a (shorter) sample using the Sound Class:

1. Load the Sample into a Sound Instance by passing it's name (or data in a byte[]) to it's constructor

2. Execute the Play Method

There are many more methods and properties in all of these classes. You can use them for example to limit the playing of the Sample to some Timespan or to fade in/out the Sample - please consult the Documentation for further details.

Another interesting Class you may use in your Projects is the "Mixer" Class. With it's functionality you can access the Mixer (Soundcard) Device to do some interesting things:

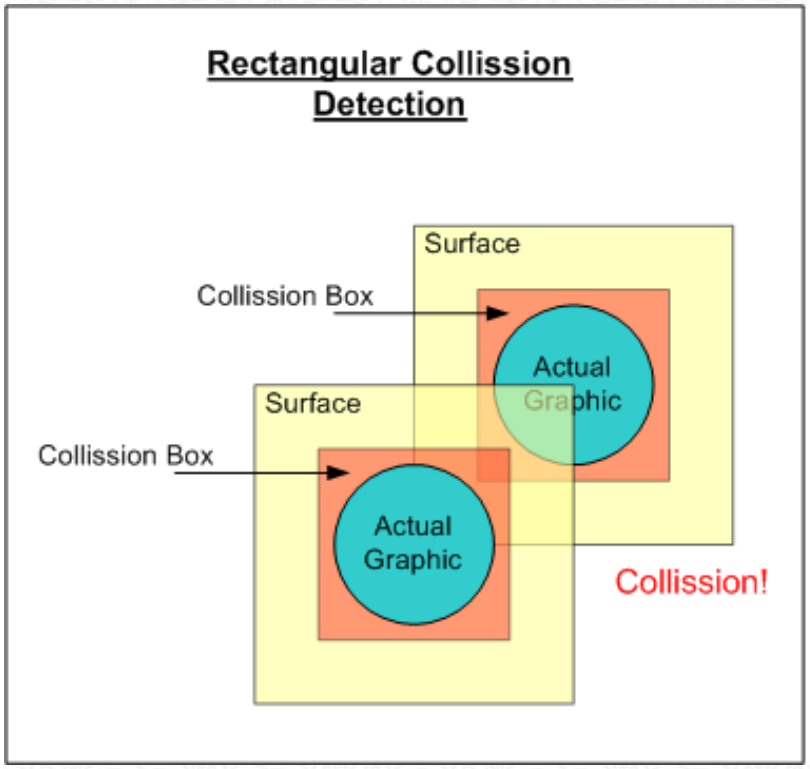
- Set overall Volume of all Sound Output (Mixer.SetAllChannelsVolume)
- Pause/Resume all Output (Mixer.Pause, Mixer.Resume)
- Set the Position of the Audio Output (if your card supports Surround): Mixer.SetPosition
- .... consult the Documentation for more details. It's relatively well documented.

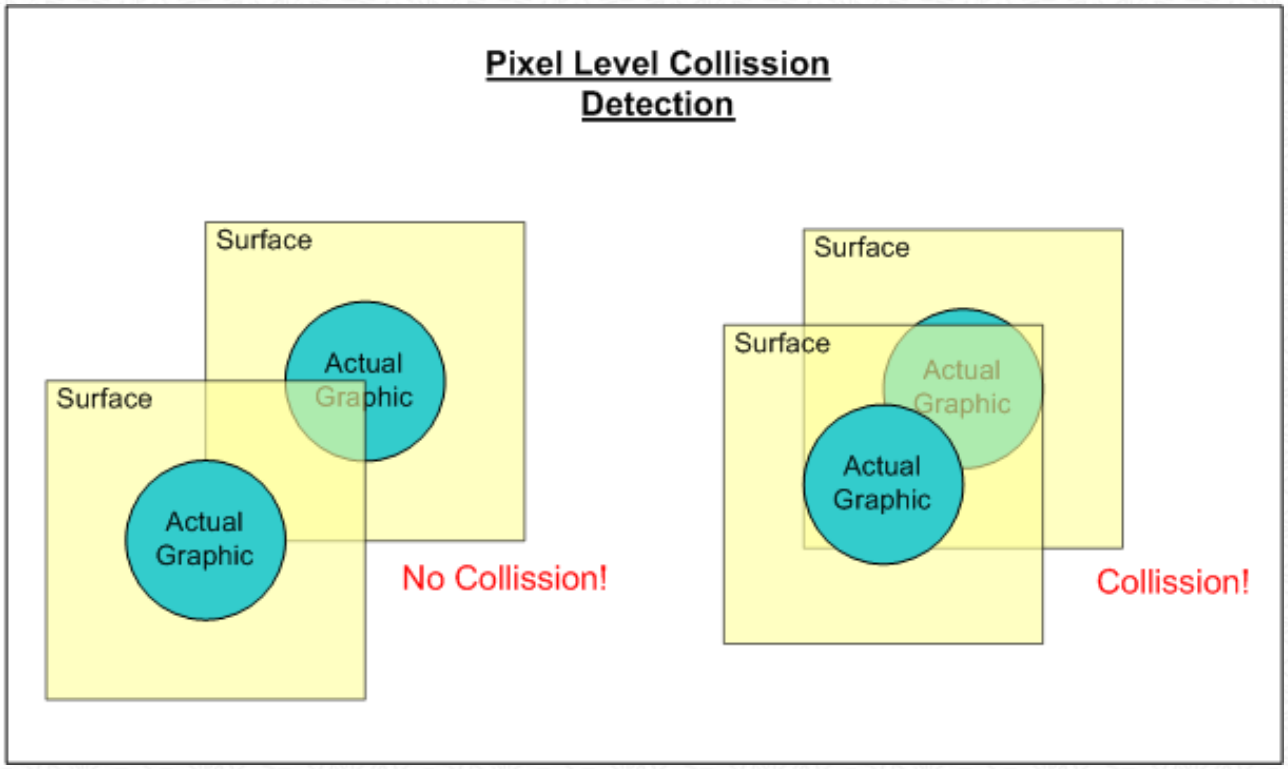
Collision Detection

In most games a essential part is to do collision detection - which means to check if two graphical objects are "touching". There are two different approaches on doing this:

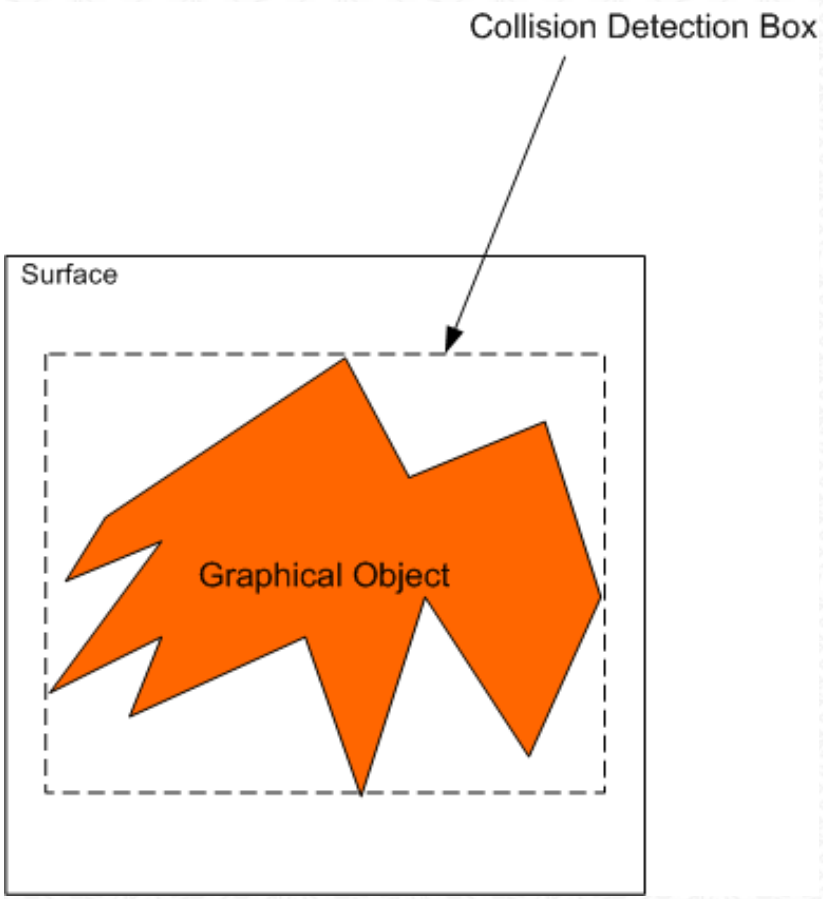
- Rectangular Collision Detection
  - Imagine a virtual Rectangle around the Graphical Object which is checked if it intersects with a Rectangle from another object
- Per Pixel Collision Detection
  - Check if the actual Graphical Object intersects with another. This is much more computing intensive than doing the Rectangular Check and is seldomly used in games.

The difference between this two methods is shown by the following illustration:





Lets start with Rectangular Collision Detection: Each Surface in your Application contains one Graphical Object. Imagine a virtual Rectangle with the dimension of the graphical object:



When doing the Collision Detection you only check if two Rectangles intersect with each other.
Let's show a little code:

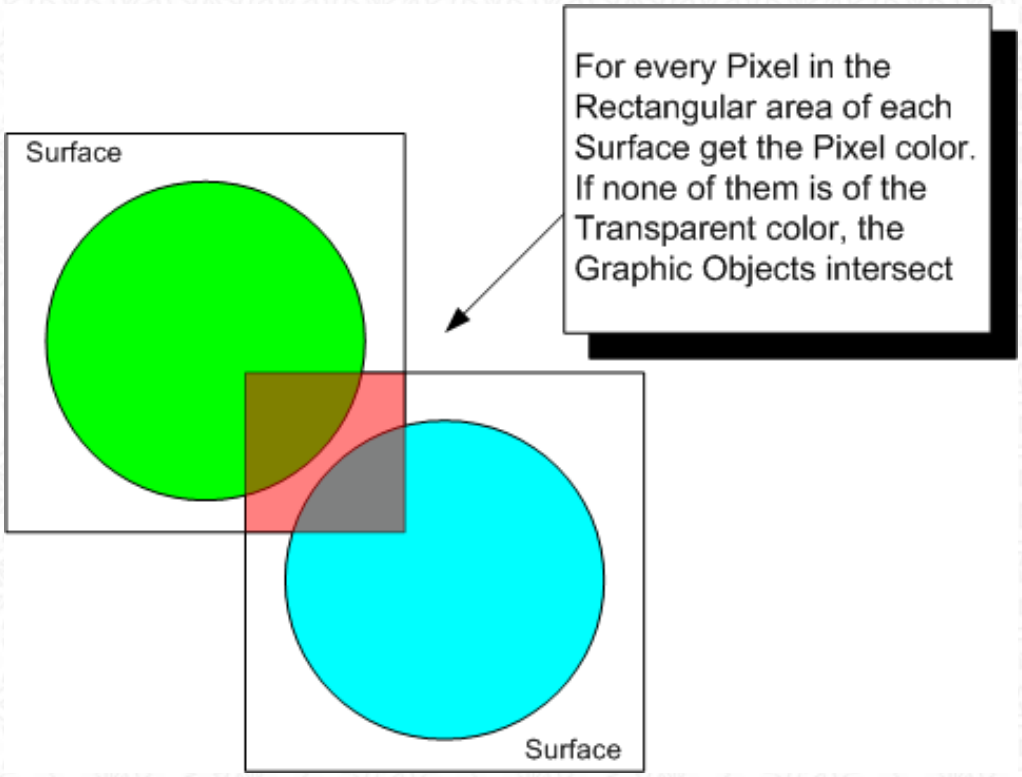
```
private Sprite GetBoundingBoxCollisionStatus()
{
    if( m_SelectedSprite == null )
        return null;

    foreach( Sprite sprite in m_SpriteCollection )
    {
        if( sprite == m_SelectedSprite )
            continue;

        if( m_SelectedSprite.Rectangle.IntersectsWith( sprite.Rectangle ))
            return sprite;
    }

    return null;
```

```
}
```

This Method checks the currently selected Sprite's Rectangular Box with each other of a Sprite collection. If a intersection is found, it returns the colliding surface, otherwise null is returned.

A more precise solution is to do per pixel collision detection. The steps necessary to do this are:

- Check if the Rectangular Boxes intersect
  - If they do we need to perform the check
  - If not, there can be no collision
- Calculate the Rectangle to perform the detection



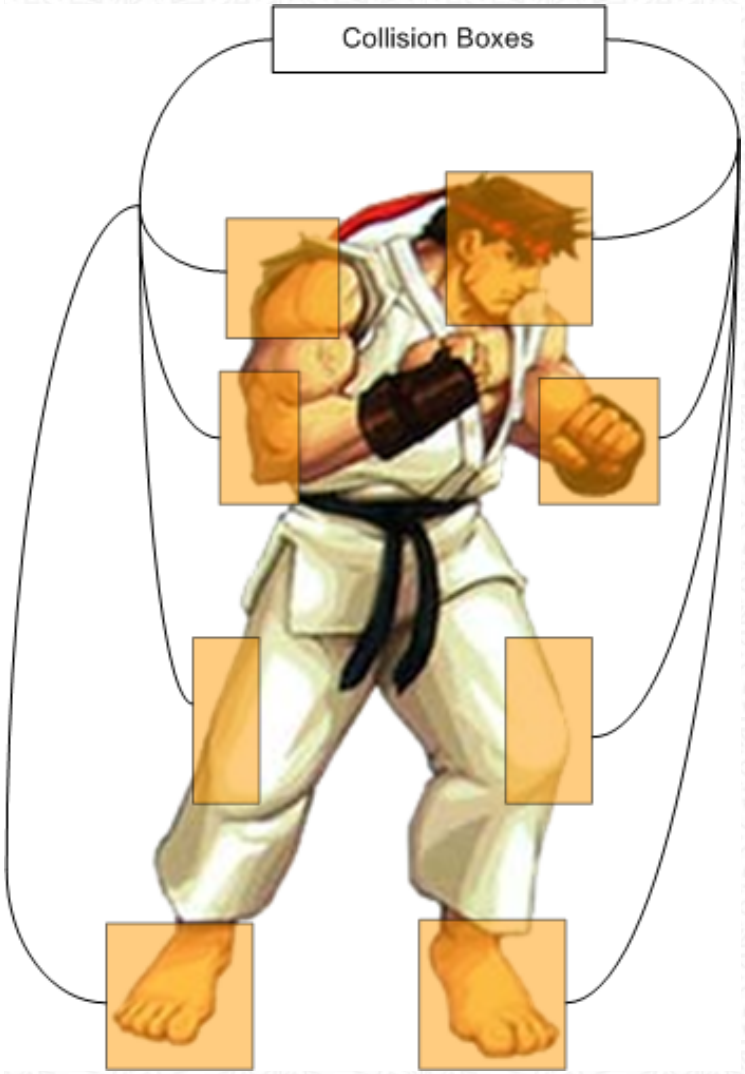
- For Each Pixel in the Rectangle, grab the color of each surface and compare their value. If none of them are the color used for transparency, then the graphical objects intersect.

In code, it looks like the following:

```
private Sprite GetPixelCollisionStatus()
```

```
{
    Sprite sprite;
    // Check if we have to check at all
    if(( sprite = GetBoundingBoxCollisionStatus()) == null )
        return null;

    // Get interesecting Rectangle
    int bottom = Math.Min( sprite.Y + sprite.Height, m_SelectedSprite.Y +
m_SelectedSprite.Height );
    int top = Math.Max( sprite.Y, m_SelectedSprite.Y );
    int right = Math.Min( sprite.X + sprite.Width, m_SelectedSprite.X +
m_SelectedSprite.Width );
    int left = Math.Max( sprite.X, m_SelectedSprite.X );
    m_CheckBox = new Rectangle( new Point( left, top ), new Size( Math.Abs( right -
left ), Math.Abs( bottom - top )));

    // Performing collision Test
    sprite.Surface.Lock();
    m_SelectedSprite.Surface.Lock();
    for( int x = 0; x < m_CheckBox.Width; x ++ )
    {
        for( int y = 0; y < m_CheckBox.Height; y ++ )
        {
            Point p1 = new Point( m_CheckBox.X - sprite.X +x, m_CheckBox.Y - sprite.Y +y );
            Point p2 = new Point( m_CheckBox.X - m_SelectedSprite.X +x, m_CheckBox.Y
- m_SelectedSprite.Y +y );
            if( ! sprite.Surface.GetPixel( p1 ).Equals( m_TransparentColor )
&& ! m_SelectedSprite.Surface.GetPixel( p2 ).Equals( m_TransparentColor ))
            {
                sprite.Surface.Unlock();
                m_SelectedSprite.Surface.Unlock();
                return sprite;
            }
        }
    }
    sprite.Surface.Unlock();
    m_SelectedSprite.Surface.Unlock();

    return null;
}
```

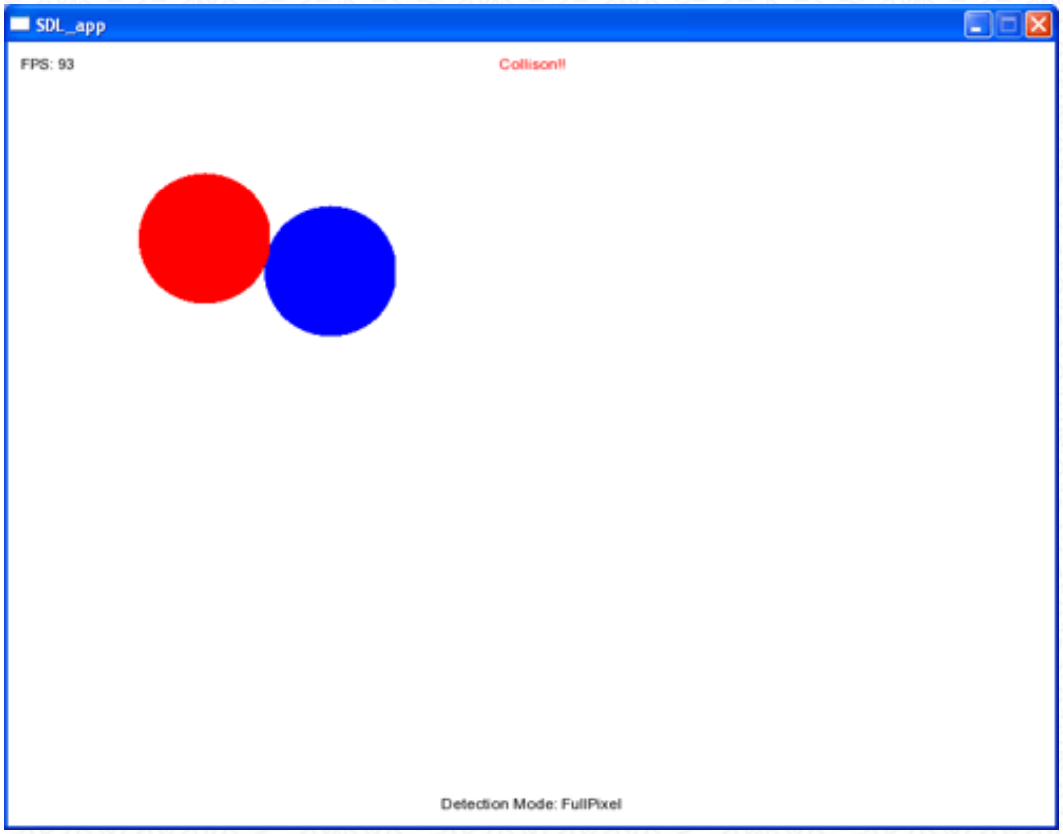
This method is rarely used because it takes much processing power. Instead in most games there are a few Collision Rectangles associated with each surface (which are defined at the design time of the Graphic). In a fighting game for example, the following collision rectangles may be used:



Every time you check for a collision, just perform a Rectangular Check on all these Rectangles.
It should be precise enough for most games and doesnt slow down your Application.

Because i haven't supplied you with a full running example (because it's rather large), just download it here: Collision.cs

When you run it, you get a Window with two Balls you can move around with the Mouse.
Pressing enter switches the Collision Detection Mode from Rectangular to Per Pixel:



Getting Joystick Input

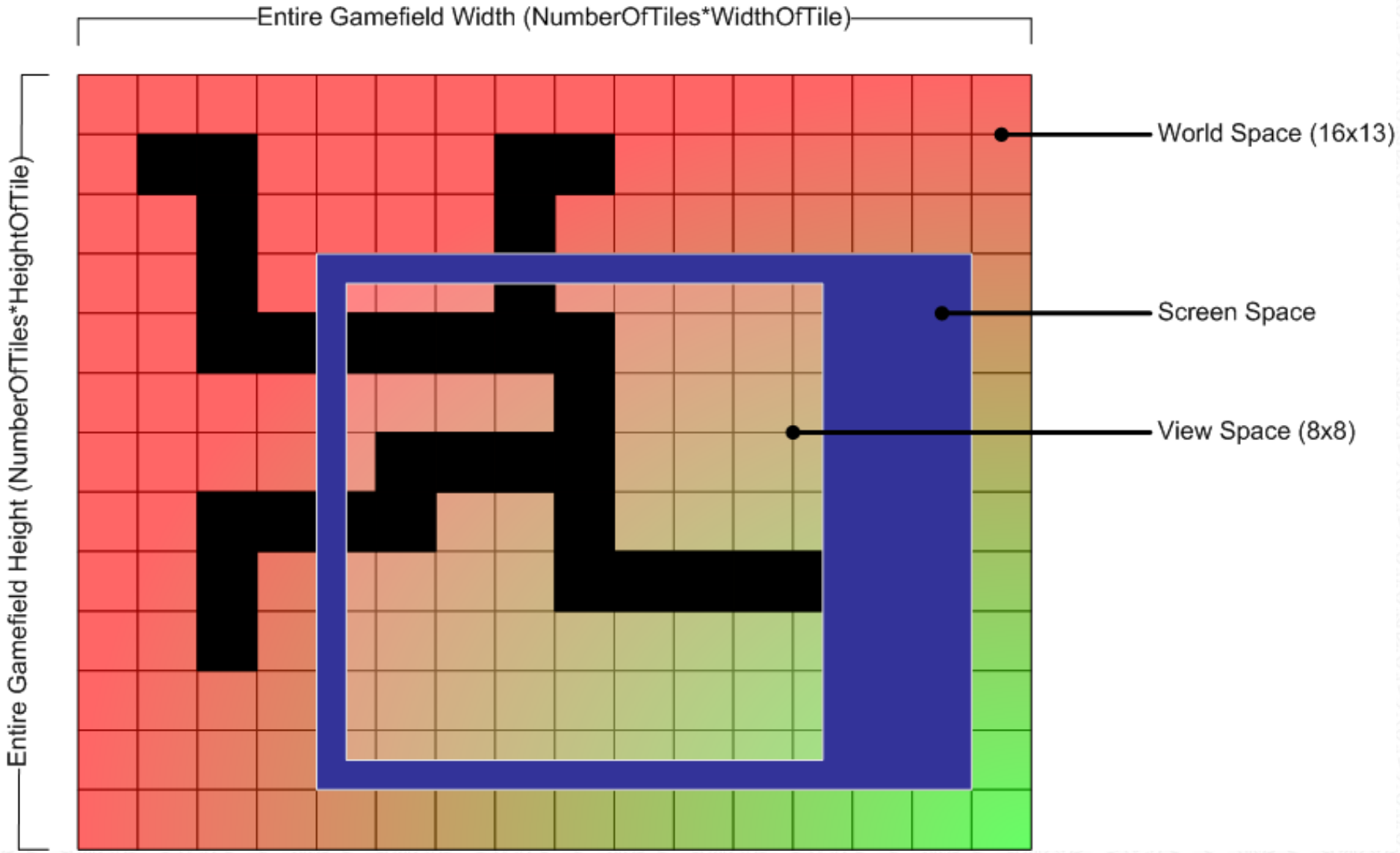
This chapter is missing due to the fact that i currently do not own a Joystick like Device (neither a Joystick nor a Gamepad)

Tiling

For some kind of games the concept of tiling is perfectly suited for their needs. But first of all a little introduction in what Tiles are:

- Your game field is built from a number of regions which each contain a different graphic to represent something (like grass, tree or water)
- Using this approach simplifies the creation of you game world, because you can write tools for letting create your gameworld from an artist
- Your game field can grow virtually to any size you need, because only the description of the tiles is stored permanently. You just build the visible thing when needed.

Take a look at the following illustration:



The large section in the Background is your Game field, which consists of a huge amount of tiles. Normally you just load a set of Tiles from a Graphics Sheet and store them somewhere. Then you have a File which describes your Gamefield in some way:

X	Y	Tile	Comment

0	0	20	A tile representing a water surface
1	0	23	A tile representing a coast part
23	15	733	A tile representing a Building
23	17	760	A tile representing a sign in the front of a Building

Your store this information somewhere in a Datastructure suited to your needs. When you are displaying something to the user, you just have to calculate which tiles are visible, build a new surface by combining the correct tiles and blitting it to the screen.

There are some standard terms you may need:

- World Space
  - It's the entire gamefield. Normally it has a width of x-tiles and a height of y-tiles
- Screen Space
  - The entire User visible screen. Includes menus, action buttons and the:
- View space
  - Your Window into the gamefield.

To calculate which tiles are visible in the View space use somthing like this:

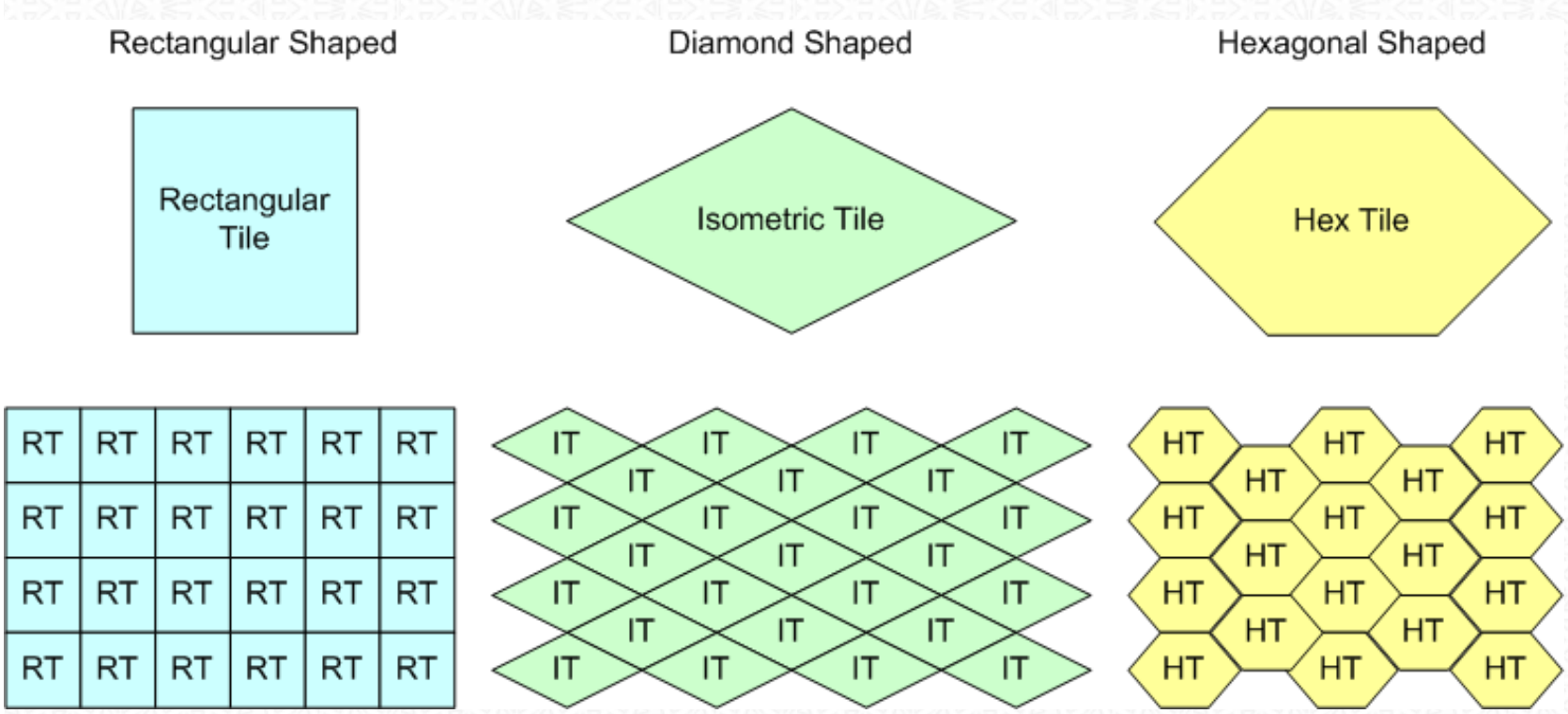
```
public Surface GetViewSpace( Point offset, Size viewSize )
{
    Stopwatch sw = new Stopwatch();
    sw.Start();

    Surface viewspace = new Surface( viewSize );

    Point startTile = new Point( offset.X / 48, offset.Y / 48 );
    Point endTile = new Point(( offset.X + viewSize.Width ) / 48, ( offset.Y +
viewSize.Height ) / 48 );
    int offsetX = offset.X % 48;
    int offsetY = offset.Y % 48;

    for( int y = startTile.Y, yCnt = 0; y <= endTile.Y; y ++, yCnt ++ )
    {
        for( int x = startTile.X, xCnt = 0; x <= endTile.X; x ++, xCnt ++ )
        {
            viewspace.Blit(
                m_Tiles[y,x].Surface,
                new Point( offsetX >= 0 ? ( 48 - offsetX ) + (( xCnt -1 ) * 48 ) : ( 48
- offsetX ) * ( xCnt * 48 ), offsetY >= 0 ? ( 48 - offsetY ) + (( yCnt -1 ) * 48 ) : ( 48
```

```
- offsetY ) * ( yCnt * 48 )),
                new Rectangle( 0, 0, 48, 48 ));
        }
    }

    sw.Stop();
    Console.Out.WriteLine( "took: {0}", sw.Elapsed );
    return viewspace;
}
```

What are we doing here:

- Get the starting Tile by dividing your Viewspace offset in pixels by the width of the Tile (startTile and endTile). The top left of your Gamefield has pixel position 0/0 and the bottom right:
  - Number of Tiles in X Dimension * Width of Tile
  - Number of Tiles in Y Dimension * Height of Tile
- Because most of the time you can't display a entire Tile at the edges of the Viewspace, calculate the offset into the left and top tiles you have to display (offsetX and offsetY)
  - We have to do this because if you display only Full Tiles your Scrolling would jump with the Width/Height of your Tiles. In most games we would like to have smooth scrolling
- For every Tile needed to display, blit it on the target surface at the correct position.
  - If the Tile to Blit is on the left Edge, only Blit the Portions calculated before (offsetX)
  - If the Tile to Blit is on the top Edge, only Blit the Portions calculated before (offsetY)
  - If it's anywhere else, blit the entire Tile
    - Position: X = (Number of Tiles already blitted fully in X Dimension * Width of Tile) + (WidthOfTile - offsetX)
    - Position: Y = (Number of Tiles already blitted fully in Y Dimension * Height of Tile) + (HeightOfTile - offsetY)

Please note that this code only works for rectangular Tiles. There are many more you can choose of. The most important are:



Examples of games which use the different Tiles:

Rectangular: The classic Sim City Game (Maxis, 1989)

Hexagonal: Battle Worlds: Kronos (King Art Games, Not Released as of Spring 2008)

Isometric: Anno 1602 (Max Design Software, 1998)

Download the entire Sourcecode for a example here

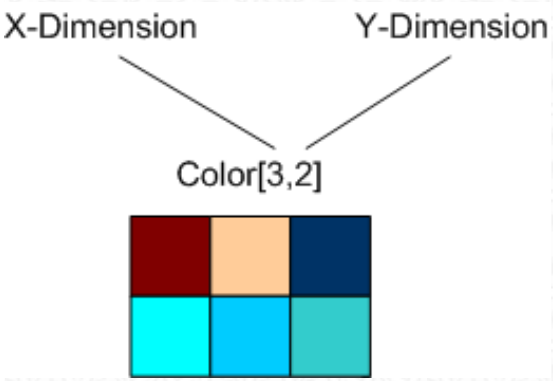
Direct Pixel Manipulation

Even if SDL is a Blitting Engine, you sometimes want to set a individual Pixel of a surface to some color - if you ask when: Let's say you write a Graphics Application in SDL like Pavel Kanzelsberger has done with it's Pixel Graphic Tool (Adobe Photoshop Clone; Runnable everywhere where SDL is available)

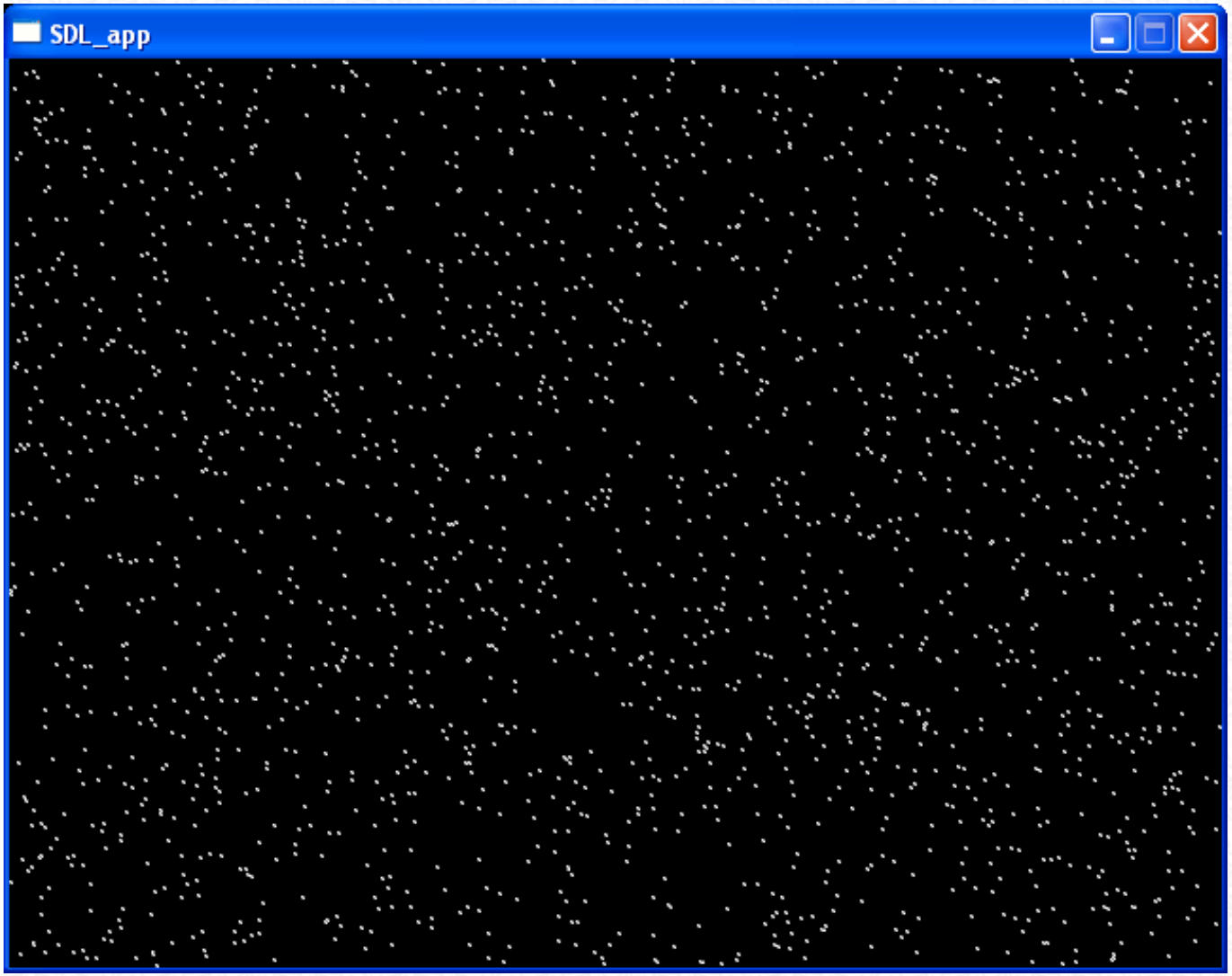
There are two Methods available in the Surface class which accomplish setting and getting the Color of Pixels:

- Color GetPixel(Point p)
  - Returns the System.Drawing.Color of the Pixel at Position p in the Surface
- void SetPixels(Point p, Color[,] color)
  - Sets the Colors contained in the jagged Array color at Position p (Top left corner)

The format of the jagged array is the following:



Below an example of how to do simple Pixel setting (Will create a snowing like Surface)



Code:

```
using System;
using System.Drawing;
using SdlDotNet.Graphics;
using SdlDotNet.Core;

public class DirectPixel
{
    private static Surface m_VideoScreen;

    public static void Main( string[] args )
```

```
    {
        m_VideoScreen = Video.SetVideoMode( 640, 480 );
        Events.Tick += new EventHandler<TickEventArgs>( TickHandler );
        Events.Quit += delegate( object sender, QuitEventArgs qa )
        {
            Events.QuitApplication();
        };
        Events.Run();
    }

    public static void TickHandler( object sender, TickEventArgs args )
    {
        m_VideoScreen.Fill( Color.Black);

        Color[,] colorBlock = new Color[2,2]
            {{ Color.White, Color.Gray },
            { Color.Gray, Color.White }};

        Random rnd = new Random();
        for( int cnt = 0; cnt < 2000; cnt ++ )
        {
            m_VideoScreen.SetPixels(
                new Point( rnd.Next( m_VideoScreen.Width -1 ), rnd.Next
( m_VideoScreen.Height -1)),
                colorBlock );
        }

        m_VideoScreen.Update();
    }
}
```

Please note that setting and getting Pixels are relatively slow operations and should avoided whenever possible.

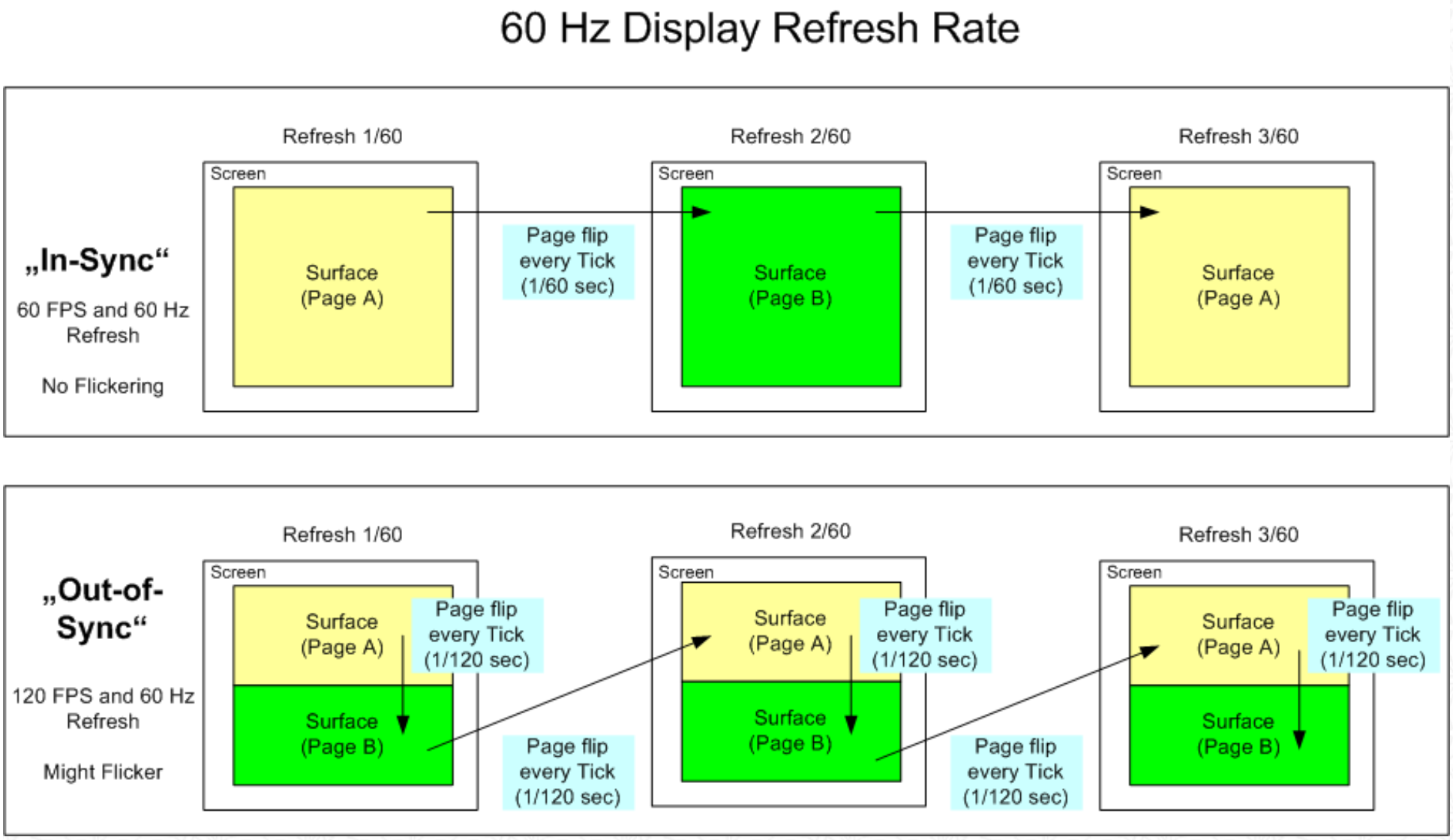
Regulating the Frame Rate and Frame independant Movement

Per default, your Application runs at the same Tick Speed as your Display's current Refresh rate (60 Hz on a typical LCD) - so your Tick Event Handler is called 60 times per second. You can adjust this rate using the following Properties:

- Events.TargetFps
  - This is the Framerate you would like to have (get/set)
- Events.Fps

- This is the Framerate you actually have. If you use the Setter, it will actually set Events.TargetFps

In my opinion it's the best to leave this at the current Refresh Rate of your screen. Why? Because when you set it to anything else there might be the problem that the Surface is flipped between a Screen Refresh and that can produce slight flickering:



Because the screen updates faster than our eyes, we won't see the Surface Flip directly, but we notice them as flickering.

If you are moving Sprites around the Screen (probably you will do) then you will mostly run into the following scenario:

- If you set the Target FPS to something fixed (like 60 fps) and the Target System cannot archive this, your Application will slow down and eventually will run out of sync.
- If you didn't limited the Framerate the game would run either too fast (if the Target System has more power than yours) or too slow.

The solution to these Problems is to use Frame independant movement. That's the outline for it:

1. You want your Sprite to move for example at 200 pixels per second
2. To archive this, you have to calculate the amount of pixels to move per tick:
    - 1 / 1000 * TicksSinceLastTickEvent * MovementPerSecondInPixels

If you run your Application at 60 FPS, the Tick Event will called every 16,6 ms (1000 ms / 60 fps). Assume that your Handler will take 5 ms to complete, so there's 11.6 ms until the next Tick Event is fired. This difference is used to calculate the amount of pixels to move. Let's move on directly into a example:

The Application will move a Surface ("Hello, FPS!") from left to right and back. With the Cursor Keys Up and Down you can set the Target Framerate - and you will see that the movement is always at the same speed.

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Threading;
using SdlDotNet.Graphics;
using SdlDotNet.Input;
using SdlDotNet.Core;
using SdlDotNet.Audio;

using Font = SdlDotNet.Graphics.Font;

public class AudioTest
{
    private static Surface m_VideoScreen;
    private static Surface m_TextSurface;
    private static Font m_FpsFont;
    private static int m_CurrentPos = 0;
    private static int m_MoveMulti = 1;
    private static float m_MoveLeftOver = 0;

    public static void Main( string[] args )
    {
        m_VideoScreen = Video.SetVideoMode( 800, 600, 32, false, false, false, true, true );

        Initialize();

        Events.TargetFps = 60;
        Events.Quit += new EventHandler<QuitEventArgs>( ApplicationQuitEventHandler );
        Events.Tick += new
EventHandler<TickEventArgs>( ApplicationTickEventHandler );

        Keyboard.EnableKeyRepeat( 200, 2 );
        Events.KeyboardDown += new
EventHandler<KeyboardEventArgs>( ApplicationKeyboardDownEventHandler );

        Events.Run();
    }

    private static void Initialize()
    {
        m_FpsFont = new Font( "Arial.ttf", 14 ); // Font for displaying FPS counter
        m_TextSurface = ( new Font( "Arial.ttf", 26 )).Render( "Hello, FPS!", Color.White );
```

```
    }

    private static void ApplicationKeyboardDownEventHandler( object
sender, KeyboardEventArgs args )
    {
        if( args.Key == Key.DownArrow )
            Events.TargetFps --;
        else if( args.Key == Key.UpArrow )
            Events.TargetFps ++;
    }

    private static void ApplicationTickEventHandler( object sender, TickEventArgs args )
    {
        int offsetPerSecond = 200; // We want to move 50 pixels every second, not more,
not less

        m_VideoScreen.Fill( Color.Black );

        // Blit FPS Display Surface to Screen
        Surface fpsCounterSurface = m_FpsFont.Render( String.Format( "FPS: {0},
WANTED: {1}", Events.Fps, Events.TargetFps ), Color.Yellow );
        m_VideoScreen.Blit( fpsCounterSurface, new Point( 10, 10 ));

        // Calculating the amount of pixels to move
        float offset = 1.0f / 1000 * args.TicksElapsed * offsetPerSecond;
        // If there is still somewhere to move add it to the offset
        if( m_MoveLeftOver < 1 )
        {
            offset += m_MoveLeftOver;
            m_MoveLeftOver = 0;
        }

        // If the offset is still below the 1 pixel threshold, set them as the leftover and
        // set the offset to 0
        if( offset < 1 )
        {
            m_MoveLeftOver += offset;
            offset = 0;
        }

        // Calculate position where to blit the text surface to
        if( m_CurrentPos >= m_VideoScreen.Width - m_TextSurface.Width )
            m_MoveMulti = -1;
        else if( m_CurrentPos <= 0 )
            m_MoveMulti = 1;
        m_CurrentPos += ( int ) ( offset * m_MoveMulti );
```

```
        // Actual blitting
        m_VideoScreen.Blit( m_TextSurface, new Point( m_CurrentPos, m_VideoScreen.
Height / 2 - m_TextSurface.Height / 2 ) );
        m_VideoScreen.Update();
    }

    private static void ApplicationQuitEventHandler( object sender, QuitEventArgs args )
    {
        Events.QuitApplication();
    }
}
```

There's essentially a single line which calculates the movement offset:

```
float offset = 1.0f / 1000 * args.TicksElapsed * offsetPerSecond;
```

But the next question comes to mind immediatly. What happens, when the offset is below 1 because the Target Framerate is too high? Don't worry, we also have a solution for this one: If it's below 1, we store this offset and add it at the next Event Handler call to the offset (until we have more than 1 pixel to move):

- Check if there is anything to move from the last time
  - If yes, add them to the offset and set the used variable to zero.
- Is the Offset >= 1?
  - No: Store it in the Variable for the Next time and set the Offset to zero
  - Yes: Do the Blitting with this Offset

The End (Download the entire Tutorial as one large Archive)