SpriteLibrary

The Big Documentation File

Tim Young – 2016,2017

SpriteLibrary is a library that allows you to build graphical games using C# and windows forms. It gives you a relatively small, but fairly powerful set of tools for animating objects, dealing with collisions, and getting graphical objects to move around. You do need to know how to program in C#, but the SpriteLibrary should make the graphical side of things much easier. You can download the latest version of the sprite library, and this documentation, from:

https://git.solidcharity.com/timy/SpriteLibrary

http://www.codeproject.com/Articles/1085446/Using-Sprites-Inside-Windows-Forms

You can also find the code documentation (properties, methods, constructors, etc.) at:

http://tyounglightsys.ddns.info/SpriteLibrary

Overview

This is a simple graphics library for Windows Forms. It allows you to build simple games (2D, arcade style, types where you have numerous small things happening scattered around on the screen) in their simple infrastructure without needing to learn a lot of extra coding. Sadly, there is a bit of a different mindset needed to make an arcade-style game, with lots of things in motion simultaneously. So you will need to develop a different programming thought-process to handle dealing with time-based events.

The SpriteLibrary does not have code to work with joysticks or other controllers; there is nothing keeping you from making games that use those, but only the pieces for interacting with the mouse and keyboard have been built into SpriteLibrary. When I get a laptop with a touch-screen, I may update the code to work with that too. But for now, it is mainly just a keyboard / mouse thing.

Windows Forms have been around for quite some time, but it is still not very easy to make graphical games in WinForms. There are many complex libraries, and other systems for making games in, and using those extensive systems is how they recommend most people do their games. SpriteLibrary is a simple sprite engine for use within Windows Forms, with the intent of keeping things simple. While you can make a complete game with this library, it is not a very polished system in and of itself. It is mainly to get people started in graphical programming. Probably the first comment I will get from this is, "why not use a real gaming library?" The point of this library is for simple games, for an entry into graphical programming. After someone uses this, then they will probably build the desire to work with DirectX, OpenGL, XNA, Unity, etc.

This `SpriteController` class allows you to take a `PictureBox` and turn it into a gaming field. You give it a background to draw on, and then plunk your sprites on it. All the work of animating and moving the sprites is taken care of. It also contains a simple system for determining if keys have been pressed (since that is one of the big things that stymies fledgling programmers). You can use this library as an example for how to make your own sprite-controller, or use this one.

C# WinForms are not built to do complex graphics easily, but it does not take much to spark the interest of a fledgling programmer. And using a class like this is a fairly simple way to get started.

Background

I recently had the opportunity to start to teach a nephew some programming. His mindset was the same as my mindset eons ago, when I started thinking about programming. “How can I make a game?” I like C# and Visual Studio. It is very easy for someone who does not know much about programming, yet wants to make something that works. It has excellent built-in documentation, and can be as complex, or as simple, as you need it to be. But it does not do graphics easily, unless you are very simplistic (one image in a box), or go to OpenGL, DirectX, Unity, UnrealEngine, or one of those fairly complex systems. Granted, there is a lot of documentation and tutorials to make those “simple.” But, using this class is probably even simpler. This is my attempt for putting graphical game creation into the hands of fledgling C# programmers.

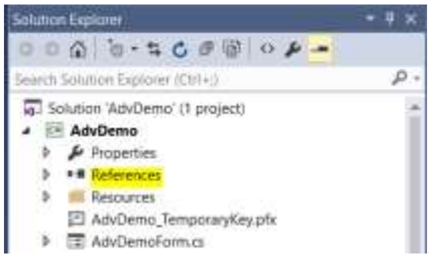
Using the Code – The Quick Guide to starting

Building the Library

After you download the source code for the library, you need to open the project and "build" it. You may want to change the “Target Framework” to the .net version you prefer. If you do not know what to choose, leave it as is for now. This option is under the project preferences under “Application.” It should match a .net version that you have installed on your computer. Different operating systems allow different versions of .net. Using too new a version may keep your program from running on older computers. At the same time, if you use too old of a version, it may not work too well.

To build, click on the “build” menu and then “Build.” Down at the bottom of Visual Studio, it should tell you that it has been built. The library cannot be executed on its own; you need to make a project (or download and run the demo) to see it in action. The resulting DLL file should be in the projects *SpriteLibrary/SpriteLibrary/Bin/Release* directory.

Adding a Reference

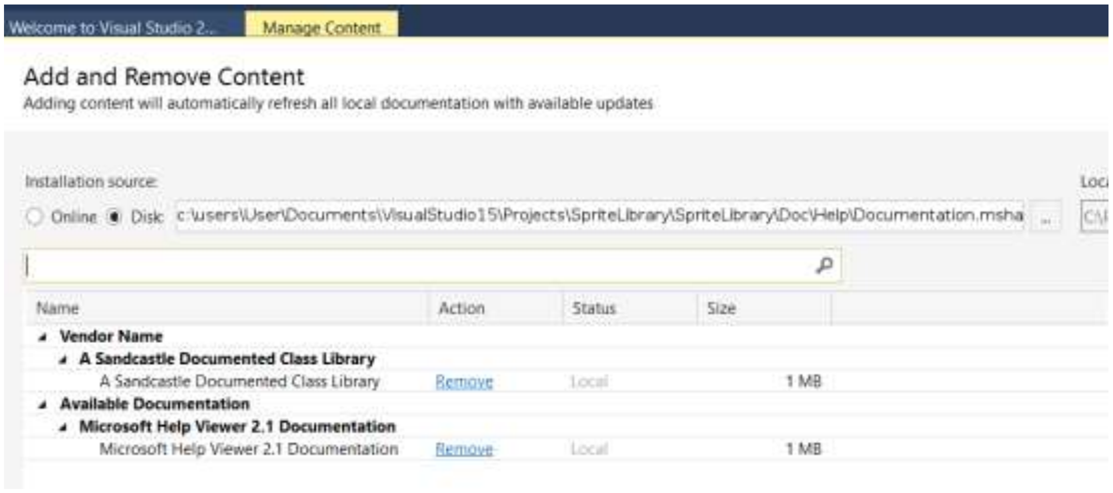


In your project (make a new project if you do not have one already), right-click “References” in the “Solution Explorer” and “Add Reference.” Go down to “Browse” and find the *SpriteLibrary* DLL. If you have built it (see above), it should be in your *projects/SpriteLibrary/SpriteLibrary/Bin/Release* directory. If you develop a game and use the built-in ClickOnce installer, this DLL will be automatically installed, along with your package, now that you have added it as a reference. So you should only need to do this once per project.

Installing the documentation

In the Doc\Help directory of the project is a .msha file, which has the SpriteLibrary documentation on it. If you install this file (instructions below), and if you are set to “Launch help in Help Viewer” (not the online help), then, when you press F1 on anything relating to the SpriteLibrary, then context-sensitive help for those items will pop up. If you prefer the online documentation, you can manually go to http://tyounglightsys.ddns.info/SpriteLibrary/doc to find the same documentation (you can browse this documentation; Visual Studio does not have it set up to merge another website into their online visual studio repository.)

To install the msha file and configure Visual Studio for accessing the online help, open Visual Studio and go to “Help” -> “View Help.” The Microsoft Help Viewer should come up, usually with the “Manage content” tab selected. If it is not selected, open the “manage content” tab.



Add and Remove Content

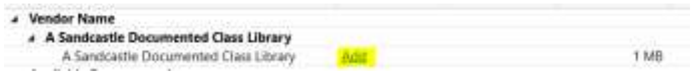
Adding content will automatically refresh all local documentation with available updates

Installation source

○ Online ● Disk c:\users\User\Documents\VisualStudio15\Projects\SpriteLibrary\SpriteLibrary\Doc\Help\Documentation.msha

Name	Action	Status	Size
Vendor Name			
A Sandcastle Documented Class Library			
A Sandcastle Documented Class Library	Remove	Local	1 MB
Available Documentation			
Microsoft Help Viewer 2.1 Documentation			
Microsoft Help Viewer 2.1 Documentation	Remove	Local	1 MB

Then, select “Disk” and browse to your SpriteLibrary project, finding the Documentation.msha file under the Doc\Help directory. Once you have found that, click the “Add” button next to the new documentation file.

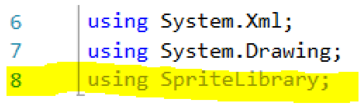


Name	Action	Status	Size
Vendor Name			
A Sandcastle Documented Class Library			
A Sandcastle Documented Class Library	Add		1 MB

Once that has been added, you still need to press “Update” in the bottom right corner of the window. At this time, SpriteLibrary has not been digitally signed, and a notification of that will pop up during the install.

Using the Sprite Library

You need to add a “`using SpriteLibrary;`” at the top of your main form, and in any code file (class, form, etc) where you reference the pieces of the `Sprite` Library (the `Sprites`, the `SpriteController`, etc.)



```
using System.Xml;
using System.Drawing;
using SpriteLibrary;
```

Initializing the Sprite Library

In the form that you want to use for a game, you will want to create a `SpriteController`. You will need to define it first as a variable. Something like:

```
SpriteController MySpriteController;
```

You may notice that we do not yet instantiate it. We need to have a `PictureBox` associated with the `SpriteController` when we make it. So, after the form is loaded and the `InitializeComponents();` has been run, then we can instantiate the `SpriteController`. In this example, we have a `PictureBox` named `MainDrawingArea`. You can have multiple sprite controllers per game, but you only want to have one per `PictureBox`. So you will want to store the variable, and pass it to the appropriate places that your program needs.

```
MainDrawingArea.BackgroundImageLayout = ImageLayout.Stretch;

MySpriteController = new SpriteController(MainDrawingArea);
```

We set the background layout to `stretch`. This is important for being able to resize the window and for a number of other things. If your sprite controller is not working from the very get-go, verify that the BackgroundLayout is set to stretch.

Adding your First Sprite

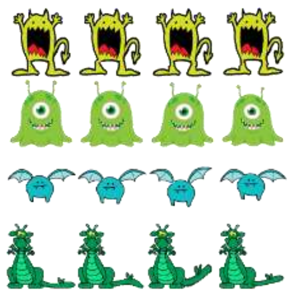
First, you must have a sprite to add. Sprites are basically a series of frames on one image. Each frame is one picture, and is the same size (for example, 100x100). And there may be many frames in the image. The resulting image might be 400 x 100, or 200 x 200. The sprite will display each frame, one at a time. So you might have an image of your main character standing there. The next frame will be with his leg slightly extended. The next has his leg out. When we see it animate, the leg moves out as if your adventurer is walking. This sprite controller assumes the image will already have the transparency set so you can see the background behind the sprite, which basically means you are using png files with transparency.

There are a few ways to create your sprite and load an animation. My example programs, which you can download from the URL at the top of this document, have us reading in the sprite from a sprite-sheet in the project resources. But they can be loaded in from any image. The reason I use resource files is because the resource files are included along with the package in the click-once deployment. So it is a good place to put them if you are hoping to give your project to someone else. Loading a sprite looks something like this:

```
Sprite JellyMonster = new Sprite(new Point(0, 100), MySpriteController,
                            Properties.Resources.monsters, 100, 100, 200, 4);

JellyMonster.SetName(“jelly”);
```

In this example, we are making a sprite named `JellyMonster` by pulling the animation out of the second row of the image "Properties.Resources.monsters". (0,0 is the first row, 0,100 is the second row.) We pass it the sprite controller, and then the "monsters" image file. We specify that the image we are pulling out is of the size 100 x 100. We use an animation speed of 200ms per frame, and we pull 4 frames out of the image. When we print the sprite, we can grow, or shrink the sprite on the `PictureBox`. It does not need to remain at 100x100. That is just the size of the individual frame in the sprite sheet image.



The last step we do in the above code is to name the sprite. We name our master sprites, and then clone them when we want to have a bunch of them. We usually will not have the named sprites display on the screen without cloning (though you can). The main reason we do this is so that we can destroy sprites at our leisure, and make as many of them again whenever we want. When you destroy a clone, it is easy to make more from the master. But destroying the master means you need to start from scratch. It is very efficient to clone sprites, but it takes a lot more effort to generate new ones from scratch.

You clone a sprite by doing something like:

```
Sprite newsprite = SpriteController.DuplicateSprite("spritename");
```

Telling Your Sprite To Do Something

Now that we have a sprite, we can put it somewhere. Once it is placed on the background, it will automatically start displaying the first animation (animation 0). An animation is a series of pictures which the sprite displayer shows one after the other to make it look like it is walking, running, etc. You can have multiple animations for each sprite (walk left, walk right, fall left, fall right, die, etc.) These animations are referenced by number. This number is made in the order of which you added the animations to the sprite. You can tell a sprite to change animations, or to animate once (leaving the sprite displaying the final frame of the animation). Here is some code for how the Sprite is configured in the `ShootingDemo`.

```
JellyMonster.AutomaticallyMoves = true;
JellyMonster.CannotMoveOutsideBox = true;
JellyMonster.SpriteHitsPictureBox += SpriteBounces;
JellyMonster.SetSpriteDirectionDegrees(180);
JellyMonster.PutBaseImageLocation(new Point(startx, starty));
JellyMonster.MovementSpeed = 30;
```

We tell the Sprite that it will automatically move. We tell it that it cannot go outside the bounds of the `picturebox`. And then we tell it where it starts, and which direction it moves. The += line is an event. `SpriteBounces` is a function that gets executed when the Sprite hits the `PictureBox`. (See "Sprite Events" below).

Sprite Events

You can add events to sprites. An event is a C# term. You can google for "C# events" to understand more about them. An event is triggered from some outside (or inside) force, and, when that event happens, the code on the event gets executed.

For example, you can add some code that is run when one sprite hits another sprite. In the `ShootingDemo`, we add that event to the monster sprites. Many times a second, the sprites check to see if they have hit another sprite, and if they do, they execute the code in that event. There are events for when sprites hit the edge of the `picturebox`, or if they have exited the `picturebox`. There is even an event that fires off before a sprite moves to a new location. You can use that one to adjust, or cancel the movement location.

You need to create an event, and add that event to the sprite. Events are cloned with the sprites, so you can add events to the parent Sprite, and those events will work for all the cloned sprites.

Payload

Each sprite has a `payload` that is of an empty class, "`SpritePayload`." This means you can store virtually anything there. This is in case you want to add extra attributes to your sprites. If you want to track the health of different sprites, some attributes for the sprite AI, or other things, you can create a class and store that data in the `Sprite.payload`. You will want to make a class that overrides the `SpritePayload` that contains the values you want to store:

```
public class MonsterPayload : SpriteLibrary.SpritePayload
    {
        public int Health = 1;
    }
```

You then make a new payload, set the data, and store it in the `Sprite.payload`. If you have multiple types of payload in different sprites, you may want to verify that the sprite payload is the type you are expecting before you try to use it.

```
If(mySprite.payload != null && mySprite.payload is MonsterPayload) { dosomething;}
```

What is happening behind the scenes

What is a sprite?

The main thing about the SpriteLibrary, as you might be able to guess, is how it deals with Sprites. A Sprite is a series of animations, some properties, and some code to go along with it.

You can have a Sprite that is a mouse. It has a few different pictures which are displayed, one after the other. By cycling through these images, it looks like the mouse is running. We can have multiple sets of images, so you can have your mouse running from left to right, and from top to bottom. Each different set of images is called an animation.

But you also need to have it move across the screen. Each sprite can be told what to do, whether setting it to move from left to right, or giving it a series of points on the screen that it moves from one to the other. The sprite tends to move smoother if you use a move-to or give it a direction. If you try to move it by putting its x and y coordinates to positions, it tends to look jerky. That is because the sprite movement routines take into consideration the imprecise nature of the timing system, and makes it appear to be moving smoothly.

Sprites also have events. An event is something that happens to the sprite, which you can write code for. For example, you can add a "On_Click" event that will let you run some code when someone clicks on your sprite. You want to use this if you want to create a menu, with "forward" / "back" sprites, or "play game" / "Exit" sprites. More about these events can be found below in the Events section.

Named Sprites

The SpriteLibrary can be used anyway you can get it to work. But it was designed to work by making master sprites, and then using copies of those master sprites whenever you need a new one.

The reason for this is mainly for memory conversation. If you are always generating new sprites from scratch, the SpriteLibrary will end up storing all their images multiple times. When you use a master sprite, all sprites duplicated off of that one use the one set of images. This makes it very memory efficient.

When you create a new sprite, you should set the sprite name to something that you will remember. That way you can create as many duplicates of that sprite, and destroy those duplicate sprites as you need to.

```
OneSprite = new Sprite(new Point(0, 0), MySpriteController, Properties.Resources.shot, 50, 50,
200, 4);
OneSprite.SetSize(new Size(30, 30));
OneSprite.SetName("shot");
```

Duplicating sprites

When you are ready to make a copy of, and use a named sprite, you can make a new copy of that sprite.

```
Sprite newsprite = MySpriteController.DuplicateSprite("shot");
//We figure out where to put the shot
Point where = Spaceship.PictureBoxLocation;
int halfwit = Spaceship.VisibleWidth / 2;
halfwit = halfwit - (newsprite.VisibleWidth / 2);
int halfhit = newsprite.VisibleHeight / 2;
where = new Point(where.X + halfwit, where.Y - halfhit);
newsprite.PutPictureBoxLocation(where);
//We tell the sprite to automatically move
newsprite.AutomaticallyMoves = true;
```

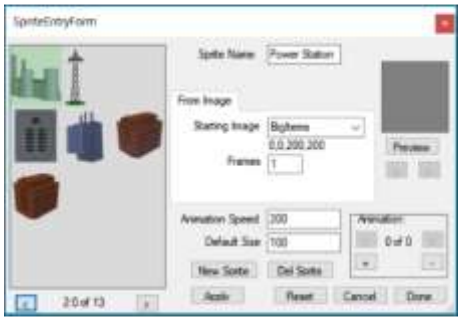
```
//We give it a direction, up
newsprite.SetSpriteDirectionDegrees(90);
//we give it a speed for how fast it moves.
newsprite.MovementSpeed = 20;
```

Each of these duplicates has their own location, direction, speed, and the like. When you duplicate a sprite, none of those properties are copied. You need to manually set those. A duplicated sprite only retains the original images (animations), any events you have set, and the animation speeds for each animation.

The SpriteDatabase

A SpriteDatabase is another way to define named sprites. If your program only has a few Sprites, than using a database may be a bit of overkill. But if you have a lot of sprites, or you are using multiple PictuerBoxes, you may want to consider setting up a SpriteDatabase.

The database allows you to build a file that defines all your sprites. If you instantiate your SpriteController with the Database, or you add it using SetSpriteDatabase, the SpriteController will reference the database when you asked for a named sprite. The nice thing about this, is that it spreads out the creation of sprites. In the AdventureDemo example, it took four or five seconds for the game to start while the SpriteController was creating the initial sprites. If a Database was used, the sprites would only be created when first referenced, which means that the load-time is virtually un-noticeable.



Sprite Name: Power Station
Starting Image: BigItems
0,0,200,200
Frames: 1
Animation Speed: 200
Default Size: 100
0 of 0
2:0 of 13

The SpriteDatabase has the additional benefit of letting you define your sprites through a GUI. The OpenEditWindow function will allow you to browse through the images which you have already added as resources to your project. You can highlight an item to turn into a sprite, specify the number of frames, and even preview the sprite.

Resources are read-only, so you use this function by instantiating the database by giving it a filename. Then you use the OpenEditWindow function of the database. When you are done defining your sprites, you can add the file as a resource to your program, and then instantiate the database using the resource name instead of the filename.

Once you have sprites defined in the database, any SpriteController that knows about the database will be able to return those sprites. For example, if we have a sprite named “Power Station” defined in the SpriteController, we can do a:

```
Sprite newsprite = MySpriteController.DuplicateSprite(“Power Station”);
```

And we will get a sprite back. The SpriteController will look up the sprite definition and create a named sprite based off of that template, and then it will return a copy of that named sprite for your use.

You do not need to both link SpriteControllers and use a SpriteDatabase, you should either do one or the other. If you want to use the sprite definition system of the SpriteDatabase, then do not bother with linking SpriteControllers. If you prefer to manually define your named Sprites, then do not bother with a SpriteDatabase.

Linked SpriteControllers

Another way to share sprite templates between SpriteControllers is by linking SpriteControllers using the LinkControllersForSpriteTemplateSharing function. Once your Controllers have been instantiated, you can link them to each-other. You only need to link them one way; they are automatically linked bi-directionally. If you need to destroy one of the controllers, be sure to unlink them first. You can have some very odd behavior if you do not.

What linked SpriteControllers do, is that a named Sprite can be accessed from either controller. The primary reason to do this, is to make it so you do not need to load lots and lots of the same sprites for multiple controllers.

One place where this would have been useful was in the AdvDemo program, where there was a playing-field that you ran around on, and a battle Picture-box where the sprites were seen much larger. That program had a lot of sprites, and took a long time to load them. It would take twice as long, if I chose to load the sprites into both of them. By linking the controllers, it would allow me to define the sprites once, but access them in multiple controllers.

You do not need to both link SpriteControllers and use a SpriteDatabase, you should either do one or the other. If you want to use the sprite definition system of the SpriteDatabase, then do not bother with linking SpriteControllers. If you prefer to manually define your named Sprites, then do not bother with a SpriteDatabase.

The timing system

SpriteLibrary needs to erase and draw the sprites every time a sprite moves or animates. So SpriteLibrary creates a timer that runs code many times a second. It checks to see if anything needs to be erased, moved, or re-drawn, and it does that.

How the graphics works

The sprite controller keeps track of the individual rectangles that need to be re-drawn on the screen. It draws the sprites on the background image, and then tells the picturebox to invalidate the areas of the background image which had been changed. Redrawing the entire picturebox is a time-consuming thing, which is why we try to only re-draw the changed portions.

The process starts by “erasing” the areas where the sprites are. It retains a copy of the background image, and copies the portions of that image to the background, thus overwriting the sprite with the original background. Then the sprites move to their new location, and re-draw themselves on the background. Finally, both the location where the sprite was, and the location where the sprite was moved to, are both invalidated.

In Depth

Time

The Sprite controller uses a System.Windows.Forms.Timer. This timer is notoriously un-precise, but it is very easy to set up initially. It tries to fire off every 10 milliseconds, but it can fire off incredibly slowly if you have long pieces of code. You want all your functions to run as quickly as possible to avoid things looking jerky.

Most programs you will make using the sprite library will begin by tapping into the DoTick Event. Every time the sprite controller is ready to pass control back to your program, it will call the DoTick event. You want to see if you should be doing anything, and then exiting the do-tick function.

How to program a delay.

If you want to wait a little bit, you might think you could do a Thread.Sleep. This has the side-effect that it blocks the sprite-controller and the user-interface. Neither one of those can do anything while the thread is sleeping. Instead, you should have a DateTime variable that is set to the time you want to delay to, and have something like this in your DoTick:

```
If(DelayUntil > DateTime.UtcNow) return;
```

You can then set your delay with something like:

```
DelayUntil = DateTime.UtcNow + TimeSpan.FromMilliseconds(100);
```

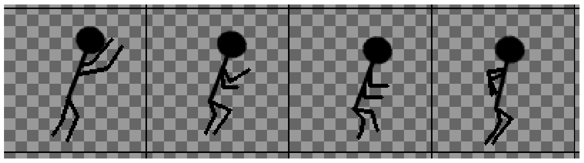
This sort of delay allows everything to continue to function. The spritecontroller continues to check for events, the User Interface still updates, etc. And, since the timer is not precise, a counter that counts down from 1000 may have different amounts of time pass.   We use the DateTime.UtcNow, instead of DateTime.Now, mainly in case someone is playing your game when daylight-savings happens. DateTime.Now will lose, or gain an hour. That can do strange things to your game.

Sprites

Drawing sprites

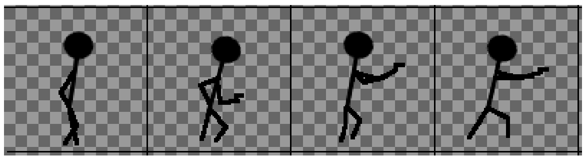
Sadly, it is hard to give a lot of advice on how to draw sprites. This is mainly because the act of drawing has such personal preferences associated with it. I tend to use GIMP. Most everyone I know of who has made games with SpriteController has drawn sprites using different tools. If you google for Sprite Creation Tools, you will find a lot of them.

The SpriteLibrary is going to look for multiple “frames”, which are individual images. These are usually easiest to create in one image, one after the other. For example:



This is a 4-frame sprite of a man jumping to the right. The sprite, when you define it, starts reading from the left side and moves to the right.

Here is another 4-frame sprite, this one of the same man, drawing a gun and shooting.



In the actual sprite itself, you do not want to have the lines going around and in-between the spaces. I show them here, simply to help explain how a sprite is constructed. In Gimp, I have a layer for my "grid", and I can hide that layer when I go to save my sprite file.

Because these sprites work best if they have transparency, you will most likely want to save the file as .png. PNG files handle transparency well (jpg and gif do not).

Because the sprite controller can rotate and flip sprites for you, you only need to make sprites facing one direction, so long as you do not mind them looking the same if they are flipped left vs right. If your adventurer has a sword belted on one side of his body, and not the other, then you may want him to look differently when he walks left than when he walks right. I am a lazy sprite maker. You see my adventurer above? He is a stick-man, and he was pretty easy to make. Half the work in a game is drawing animations. Really. Making animations is a lot of work, even if you have something like the SpriteLibrary to make animating things easier.

Instantiating sprites

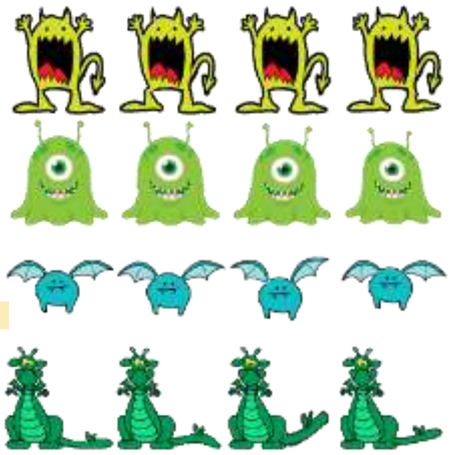
Now that you have some sprite images, you need to turn them into sprites within your program. I will call this "instantiation", though it can also be called "constructing."

There are a good number of sprite-instantiation functions. For greater information on "where" to instantiate your sprites, and how to design a game, see the section on Program Design down below.

Usually, however, you want to instantiate your sprites at the beginning of a program. If you have a lot of sprites and do not need them all immediately, you can try to instantiate them on demand. Either way, what you want to do is to make one "named sprite", which is the master sprite for that image. Then, every time you want to use a sprite of that type, you duplicate the master sprite and put the copy of it onto the screen. This is because the initial instantiation of the sprite takes a long time, but duplicating the sprite happens very quickly. It is also much more memory efficient this way.

The SpriteController will remember your sprites for you. After you instantiate, and name, the sprite, you can then ask the SpriteController for the sprite named [whatever] and you get that sprite. Or, you ask for a duplicate of the sprite named [whatever], and you get a copy of that which is ready to be placed on your screen.

Here is an example. We are have this image in our resources:



Each of the images is 100x100. We want to read in the second set of images, the jelly-monster.

```
Sprite JellyMonster = new Sprite(new Point(0, 100),
         MySpriteController,
         Properties.Resources.monsters,
         100, 100, 200, 4);
JellyMonster.SetName("jelly");
```

In this example, we are making a sprite named `JellyMonster` by pulling the animation out of the second row of the image "Properties.Resources.monsters". (0,0 is the first row, 0,100 is the second row.) We pass it the sprite controller, and then the "monsters" image file from within the resources. We specify that the image we are pulling out is of the size 100 x 100. We use an animation speed of 200ms per frame, and we pull 4 frames out of the image.

When we print the sprite, we can grow, or shrink the sprite on the `PictureBox`. It does not need to remain at 100x100. That is just the size of the individual frame in the sprite sheet image.

The last step we do in the above code is to name the sprite. We name our master sprites, and then clone them when we want to have a bunch of them. Destroying the master means you need to start from scratch. It is very efficient to duplicate sprites, but it takes a lot more effort to generate new ones from scratch.

You duplicate a sprite by doing something like:

```
Sprite newsprite = SpriteController.DuplicateSprite("spritename");
```

Sprites, when they are duplicated, retain all the main settings of the original. BUT, they do not retain the position on the screen or any "changes" you made to the sprite (animation speed adjustments, what animation you are viewing, etc. Your named sprites should not be on the screen, remember? But even so, the location is not duplicated.) The sprite has the animations (the images), an animation speed, and can have functions for if it collides with another sprite. All of these things are duplicated from the master sprite.

The things that are not duplicated, are things which are usually unique to a sprite. The location, speed, direction, and which animation is currently displaying. Those things are all separate. That way we could have fifteen jelly-monsters running around the map, chasing after some poor adventurer. Those jelly-monsters need to have different locations, speeds, directions, etc. But, if one of them collides with our adventurer, we want the same “KillAdventurer” function to run.

Rotating and flipping sprites

When you instantiate a sprite, or add an animation to an existing sprite, you can have your sprite flip or rotate. You can give a sprite a custom rotation while it moves around, but that takes a bit more drawing power. If you are only going to have a few different rotations, you may want to have different animations, each one at a different rotation.

For example, if you have an adventurer walking right, we can create an animation with the image flipped horizontally. This will make him look like he is walking left instead of right. You can do this by using the AddAnimation (to your named sprite), giving it the number of an animation to duplicate, and either telling it the angle to rotate, or telling it to flip horizontally or vertically.

Here is the function you call to make a sprite that is rotated.

```
public void AddAnimation(
         int AnimationToCopy,
         int RotationDegrees
)
```

The jelly-monster does not really look good rotated, but we could try:

```
JellyMonster.AddAnimation(0,180);
```

This would have him basically jiggling on his head. What we really would want to do for the jelley-monster, would be to flip him horizontally using:

```
public void AddAnimation(
         int AnimationToCopy,
         bool MirrorHorizontal,
         bool MirrorVertical
)
```

The actual code to do that would be:

```
JellyMonster.AddAnimation(0, true, false);
```

One thing to be aware of with rotating and flipping. The sprites remain bound by the original size. If your original size was a rectangle and not a square, rotating the sprite will result in the object changing sizes.

If you do not want to create rotated or flipped sprites, you can rotate and flip them using their original shape. This does result in a little bit slower drawing and extra memory usage over time (C# does clean the memory usage up, so this works fine for many games). You can use the Sprite.Rotation(int) function to rotate a sprite. And you can use the Sprite.MirrorHorizontally and Sprite.MirrorVertically Booleans to flip a sprite.

Moving Sprites

There are a few ways to have sprites move. Whenever possible, you want to use a movement function instead of simply placing a sprite at the new location. Even if the sprite location is moving a pixel at a time, you will find that manually moving the sprite will look jerky. The sprite movement functions that come with the sprite are set up on a timer such that they adjust to minor delays, so that movement always looks smooth.

The two main ways to do movement are to use a MoveTo function, or set a direction and speed. The MoveTo functions can move to a specific point on the screen, or move to another sprite.

Here is an example of how to tell a JellyMonster to move to a specific point. The sprite will move at the specified speed until it reaches the given point. When it reaches that point, the SpriteArrivedAtEndPoint function will trigger. And, the Boolean, SpriteReachedEndPoint, will be true.

```
JellyMonster.AutomaticallyMoves = true;
JellyMonster.MoveTo(new Point(100,100));
JellyMonster.MovementSpeed = 30;
```

Here is an example for telling the JellyMonster to move in a direction:

```
JellyMonster.AutomaticallyMoves = true;
JellyMonster.CannotMoveOutsideBox = true;
JellyMonster.SpriteHitsPictureBox += SpriteBounces;
JellyMonster.SetSpriteDirectionDegrees(180);
JellyMonster.PutBaseImageLocation(new Point(startx, starty));
JellyMonster.MovementSpeed = 30;
```

Destroying sprites

Once you are done with a sprite, you should destroy it by calling the sprite.Destroy() function, which is usually followed by setting the variable to null:

```
JellyMonster.Destroy();
JellyMonster = null;
```

Destroying a sprite will erase it from the screen, and then remove all traces of it from the SpriteLibrary. You should be careful about storing sprites in variables that persist after the sprite is destroyed. Most functions will realize that a sprite has been destroyed and work appropriately. But, if you use a sprite that has been “destroyed”, there can be some rather odd things that can happen.

What I mean by this is, if I have a Sprite variable named “JellySprite” and I do something like: JellySprite.Destroy(); Then, in a different part of the program I can still do something like: JellySprite.ChangeAnimation(3); JellySprite can still point to the sprite, even though the sprite has been destroyed. This is something C# does. The act of “Destroying” a sprite will go through the process of getting rid of it from the screen and SpriteLibrary, but you still need to set the variable to null. Once the variable is set to null, C# knows it can do “garbage collection” and finish getting rid of the variable.

Sprite Display Order

Some sprites will print on top of other sprites. When two sprites exist in the same space, one of them will invariably be on top. You ultimately have control over which one is on top.

Sprites have a Z value, which determines which sprite prints on top of which sprite. Usually, this is enough. But there will be some sorts of games where you will want to use an alternative approach to determining which sprite draws on top of another. For example, you might want to have a tower defense game where, as the sprites go farther down the Y axis, they print before those which are closer to the top of the screen.

The SpriteController has a function to sort the sprites. You can experiment with sorting them differently, but here is the default value, sorting strictly by the Zvalue:

```
SpriteComparisonDelegate = delegate (Sprite first, Sprite second)
         { return first.Zvalue.CompareTo(second.Zvalue); };
```

Animations

Creating a sprite

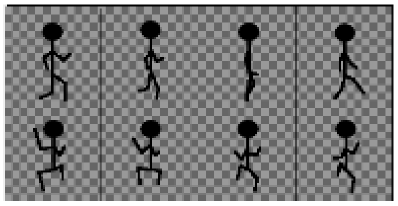
A Sprite is an animation. It is a series of images that are displayed one after the other. Usually, those images are related to each-other, such that it looks like it is moving. If you only have one image, that one image keeps repeating; the same frame over and over.

One SpriteLibrary user complained about terrible response-time to his program. It turned out that he had his single-frame sprites trying to refresh every 1ms. The program was spending all it's time trying to re-draw the same image. He changed it so his sprites had a duration of 1000ms, and that made all the difference in the world to his program.

When you instantiate a named sprite, you want to make sure you set all the defaults for animation speeds. Those speeds will be passed down to all duplicates of the named sprite.

Multiple Animations in a Sprite

A Sprite can have multiple animations. For example, you may have an adventurer who has animations for walk left, walk right, jump left, jump right, climb up, fall down, etc. They are all the one adventurer, so they are all in one sprite. You can tell the sprite to display animation 0, and it looks like he is walking right. Tell it to do animation 1, and it looks like he is walking left.



The SpriteLibrary is written in C#, and so the indexes start at Zero. The first animation is 0, the second is 1, etc.

I usually have all the frames for one sprite in one image, but you do not need to do that. You can pull frames from multiple images for different animations. BUT, every animation needs to come from one image. At the time of the writing of this documentation, I do not have a sprite, or animation creation method that pulls frames from multiple images.

Duration of sprite animations

The duration that one frame is displayed is tracked in milliseconds. Please be aware that it is mostly impossible for the system to animate faster than 15ms. It might appear to do so, but what happens is that some of the frames will be skipped so that it appears to animate fast. I have seen people attempt to build sprites that animate insanely fast, and all that happens is that two-thirds of their frames are skipped. It is best to assume that 50 to 100ms is “fast.” That is ten to twenty images a second, which is plenty fast for an animation. The problem with this speed is that it does take a toll on the processor. A bunch of sprites animating at 50ms will quickly make your game run slowly on an older computer.

200-300 is a nice speed for an animation. It does not take as much of a toll on the processor, and you can handle many sprites animating at that speed. A lot will depend on how many sprites you need to make your game work. You may need to adjust the animation speed of things once your game has been created if you find that it is running too slowly.

When I programmed the sprite controller, I did skimp a bit on drawing optimally. The right way to do it would be to occasionally draw partial sprites when two of them overlap. Instead, what I did was to tell the system to redraw all sprites that overlapped, in the order they needed to be drawn. This means that, when you have a lot of overlapping sprites, it can sometimes do a lot of rendering. And, if a sprite is completely hidden by other sprites, that hidden sprite is drawn completely, and then overwritten by the sprites on top of them. And, this all happens every time something redraws.

All this to say, the duration of any animation may be something to consider. You can override the speed of an individual frame, making it animate faster or slower than the rest of the frames in the sequence. But usually you will have the same duration for each frame in the animation. Most of the time, I just draw something and change the duration until it looks right. But, if you have a lot of sprites, and a lot of overlap, you may need to adjust your duration a little bit to make things work.

Single-frame sprites

Single-frame sprites can be very useful. Often menu-buttons, scenery, or stationary objects will not animate. You should use a duration of 1000 or thereabouts for these sprites.

Events

Part of the power of the sprite library is the event system. An event is triggered by something in the program; two sprites collide, someone clicks on a sprite, or something like that. (search the internet for "C# event" to read up on this concept if it is new to you) You can create an event and attach it to a sprite, or the sprite controller, and then have your code run when the event occurs.

An example is if you have an explosion sprite, and want the sprite to be destroyed once the explosion has completed. You can add an event to SpriteAnimationComplete, and tell the sprite to animate once. Then, when the sprite has finished with its animation, the code you have created will execute.

How events work

The sprite-library is always checking for things, like sprites colliding, sprites exiting the bounds of the picture-box, sprites being destroyed, and the like. When the sprite-controller finds that something has happened, it will try to execute the code associated with that event. If no such code exists, the sprite controller continues on.

When to use events

You want to use events whenever you want something triggered instead of happening at a set time.

For example:

You can take a torpedo and send it one direction. You can have it set to explode when it hits something (SpriteHitsSprite).

You can take a ball and have it "bounce" every time it hits the bounds of the picture-box. Set an event for when it hits the edge (SpriteHitsPictureBox), check to see what direction it had been moving, and change the direction accordingly.

You can take a sprite and tell it to MoveTo a bunch of places, and then have it run some code when it has reached it's end-point (SpriteArrivedAtEndPoint). You may want to do this if you are having an adventurer jump over something to trigger the fact that he just landed.

You can tell a sprite to do something when a sprite has finished animating. This is excellent for an explosion that you want to have disappear once the explosion animation has completed. For this to work you want to set the sprite to AnimateOnce, and then have a SpriteAnimationComplete event.

One not-so-obvious use is the CheckBeforeMove event; you can use this to adjust the position of a sprite that is automatically moving, or to check to see if a sprite has collided with something in particular. In one of my demos (RunningDemo), I use this to see if the sprite is standing on the ground, or if it needs to fall. The CheckBeforeMove event happens when the sprite thinks it knows where it is supposed to move to, but sends the location to the event. You can change it and even cancel the move event (make the movement stop).

How to add an event

Events can be added to sprites at any time.

```
ExplosionSprite.SpriteAnimationComplete += ExplosionCompletes;

public void ExplosionCompletes(object sender, EventArgs e)
{
    Sprite tSprite = (Sprite)sender;
    tSprite.Destroy();
}
```

In this example, ExplosionSprite is the name of the sprite, and we are telling the sprite to destroy itself once the sprite has finished animating. You can add the event to the sprite when you first create the sprite and name it (see the section on naming sprites). Events are duplicated, along with all the properties, when a sprite is duplicated from a named sprite. It usually makes most sense to put the event on the named sprite, so you do not need to do it over and over as you create and destroy sprites later on.

If you add events to a sprite that is based off a named sprite, the named sprite is not changed, and that event only occurs on that one sprite.

You do not need to remove events from sprites when destroying sprites. Those are properly cleaned up for you.

Movement

There are a number of ways to move a sprite, but some work a lot better than others. Because the sprite-controller (and C# in general) handles time in an inconsistent manner, it looks a lot better to use the sprite “SetDirection” and “MoveTo” functions. To get a sprite to move, however, you not only need to use one of those, but you also need to define the speed, and set “AutomaticallyMoves = true.” Here is an example of defining a sprite and starting it moving.

```
NewSprite.AutomaticallyMoves = true;
NewSprite.SetSpriteDirectionDegrees(180); //Direction in degrees. 0 = right, 180 = left
NewSprite.MovementSpeed = speed;
```

One issue with the sprite-controller is that it resets things every time you tell it to do the same thing again. So, it is better to check to see if the sprite is already moving the direction you want it to be moving, or check to see if it is already doing the animation you want, before telling it to move that direction or do that animation. Telling it to do it over and over usually results in it just sitting there.

```
If(NewSprite.AutomaticallyMoves == false)
{
    NewSprite.AutomaticallyMoves = true;
    NewSprite.SetSpriteDirectionDegrees(180); //Direction in degrees. 0 = right, 180 = left
    NewSprite.MovementSpeed = speed;
}
```

Direction

You can get and set the sprite direction. This direction is not precise. The sprite actually stores the direction it is moving in a "Vector". So if you give it a direction, it computes a vector and moves in that general direction. If you set a direction of 45 and then ask the sprite what direction it is moving, it might return a direction of 45.230. If you do a comparison between that and 45, C# will say that it is not going at a 45-degree angle. In short, when comparing directions, allow a little bit of slop.

```
double degrees = NewSprite.GetSpriteDegrees();
if(degrees > 43 && degrees < 47)
{
    //assume we are going about 45 degrees.
}
```

Speed

Speed is how many pixels the sprite travels in a specific amount of time. This means that the relative speed changes with the size of your background. If your background image is 800x600, then a speed of 10 will look faster than if your background is 1024x768. You will need to figure out what speed is appropriate for your game, or whatever you are making.

MoveTo

There are a number of MoveTo functions. You can move to one point, move to a list of points, or move to a sprite. They all work well in conjunction with the `SpriteArrivedAtEndPoint` event.

When you moveto one point, the sprite will move to that position and then stop moving. When it reaches the destination, it will fire off the SpriteArrivedAtEndPoint event, as well as set the SpriteReachedEndPoint variable to true.

When you MoveTo multiple points, the sprite will move from one point to the next, to the next. When it reaches each waypoint (each point along the way except for the ending point), the SpriteArrivedAtWaypoint event is triggered. When the Sprite reaches the end, the SpriteReachedEndPoint event is triggered, and the SpriteReachedEndPoint variable is set to true.

When you MoveTo a sprite, the target adjusts every time that the sprite moves. It tries to get the center of the moving sprite to hit the center of the target sprite. This is not "optimal." Instead of moving to where the sprite will be, it moves to where it is now. So the sprite that is doing the MoveTo should be moving faster than the destination sprite if you want to eventually hit it. There are no "waypoints" involved, but the SpriteReachedEndPoint function is called when the sprite finally hits the target sprite. And, the SpriteReachedEndPoint is set to true.

If at any time you do another move-to, the old move-to information is replaced with the new. You cannot simply add another waypoint. You replace the entire move-to destination with a different one. And, in doing that, none of the SpriteReached.. events get triggered.

SpriteReachedEndPoint: This variable is somewhat useful, but has caused a fair bit of confusion. This value is basically "false" during the operation of a move-to. If you never tell a sprite to move, then it is always "true." As soon as you tell it to move, the value remains false until it has stopped moving. This value starts “true”, but, if you are telling it to move somewhere, it is “false” until it gets to that spot (or the movement is canceled)

The PictureBox Background

The background is actually a critical part of the sprite-controller. Because the controller takes portions of the background to write over the sprites to “erase” them, the background must be solid. A transparent, or empty sprite controller will give you all sorts of grief. And, always have your background image set before creating the SpriteController. (see replacing a background below for what you need to do if you do not set it beforehand). I like to set the BackgroundImageLayout at the same time, as it makes sure that I get it correct. The Sprite Controller needs it to be ImageLayout.Stretch. Also, you will save yourself some grief if you make a duplicate of a resource instead of using the resource itself.

```
MainDrawingArea.BackgroundImage = new Bitmap(Properties.Resources.Background);
MainDrawingArea.BackgroundImageLayout = ImageLayout.Stretch;
MySpriteController = new SpriteController(MainDrawingArea);
```

The background size drastically affects the speed of the sprites. Sprites move in pixels per amount of time. If you have a huge background that is squished into a small PictureBox, then a low speed may not even be noticeable. But, if the background image is very small, the sprite may go zipping across a lot faster than you would expect.

The background size also affects memory usage and update speeds. For your first program that uses a sprite controller, use something that is in the 800x600 range. (600x600 or 800x800 if you want a square.) You can adjust it from there. The SpriteController can handle things that are fairly large, or fairly small.

Replacing the Background

Replacing the background image is actually a lot more complex than you might imagine. Once you use the below code, it can be done without any problem. But you need to do it this way, or it just goofs up in a number of small ways.

You need to tell the sprite controller that you are replacing the background image, and you need to change the image to that image as well. Because the Images are actually pointers to memory where the image sets, changes to one image will affect the other image. This goofs things up, so what we do is duplicate the image twice, and tell the sprite controller to use one of the copies and then set the background to be the other one of the two copies. Finally, we tell the PictureBox to invalidate itself. That does everything that is needed.

```
void ReplaceBackground(Image NewBackground)
        {
            if (MyController == null) return;
            if (NewBackground == null) return;

            Image OneImage = new Bitmap(NewBackground);
            MyController.ReplaceOriginalImage(OneImage);

            Image TwoImage = new Bitmap(NewBackground);
            pb_map.BackgroundImage = TwoImage;
            pb_map.Invalidate();
        }
```

Sound

So, sound is one of the things that still does not work well. The problem is that C# does not "mix" different sounds well. There are other libraries and systems that I might use at some point in time. But, for now you cannot play multiple sounds using the SpriteLibrary.

The PlayAsync function tried to do this a little bit, but it ends up still playing one sound at a time.

PictureBoxes

It is easiest if you use one PictureBox for the whole game. Creating and destroying PictureBoxes means you need to create and destroy the SpriteControllers that go with them. This is actually both messy, and very hard to do. If you can, design your game with only one SpriteController. You can have part of your window that is "menu" area, and part that is for displaying the score. When your game is over, replace the background and display a "you have won / lost" sprite.

In my AdventureDemo game, I tried to use multiple PictuerBoxes. This seemed like a good idea at the time, but I quickly decided that I should have found a way to do it with just the one. Well, it would have been OK to do it with two of them, but not to destroy them and open a new window with a new set of PictureBoxes.

Part of it is simply the inefficiency of needing to load all the sprites every time you create a new PictureBox. The other bit of it is the pain of trying to destroy the old controller.

One very odd bit of behavior I had, at one time, was if I exited the game in the middle of a battle, my party sprites would continue to get "hit" by the enemies, even though their window had been closed and the enemies had been "destroyed." I tried to get rid of a lot of stuff, but simply closing the window does not actually get rid of all the variables that had been part of that.

So, to sum all of this up. Try to only have one window, and the picture-boxes on that one window. If you need to have multiple PictureBoxes on one window, that is not too much of a problem. But, do not try to open new windows with new PictureBoxes and new sprite-controllers. That can end up with some rather odd behavior.

Menus

It is not too difficult to make a nice-looking menu using sprites. The general concept is to make a bunch of sprites that are "buttons." Each of these sprites should have an OnClick event. When the player clicks on that menu item, the code in your OnClick event will fire off.

You can also use a MouseHover (to pull up a tool-tip or something)

Many people will use the “MouseEnter” and “MouseLeave” to change the animation of a menu sprite to glow, pulse, or do something when the mouse is over the button. It is just very nice to have the button react to the mouse so the user knows which button they are on at any given time.

If all your menu sprites use the same animation scheme, it might look something like: Animation 0 is the menu item by itself. Animation 1 is it with the mouse over it (glowing, etc).

Then, all your buttons can share the same mouse-enter, and mouse-leave functions.

```
MenuSprite.MouseEnter += spriteMouseEnter;
MenuSprite.MouseLeave += spriteMouseLeave;

void spriteMouseEnter(Sprite MenuSprite, SpriteEventArgs e)
{
    MenuSprite.ChangeAnimation(1);
}

void spriteMouseLeave(Sprite MenuSprite, SpriteEventArgs e)
{
    MenuSprite.ChangeAnimation(0);
}
```

Because you may sometimes have some transparent areas surrounding a menu-button, you may also want to try out the MouseEnterTransparent, MouseLeaveTransparent, and MouseHoverTransparent functions. These work the same way, except that, instead of triggering when the mouse is over the sprite rectangle, it only triggers if it is over a non-transparent area of the sprite.

Keypresses

Handling keypresses is always an interesting thing. The SpriteController has some code to do this for you, but you can also roll your own.

C# has a strange issue. If you have a key pressed and then the form loses focus, the form will not register a “key up” event for that key. This means it is entirely possible for you to release a key and for C# to think that the key is still pressed.

The main code for handling keypresses is SpriteController.IsKeyPressed(Key).

Below is some code. We define a SpriteController, and give it a “DoTick” function. That function is called many times a second by the SpriteController. To keep things sane, we exit out of the function if less than 100 milliseconds has passed. (We do this by storing the time we last did something, and then subtracting that from the current time and comparing the milliseconds.)

Finally, we check for a few particular keypresses and act upon them.

```
SpriteController MySpriteController;
DateTime LastMovement = DateTime.Now; //Used to give a slight delay in checking for keypress.

public partial class ShootingField : Form
```

```
{
    MainDrawingArea.BackgroundImage = Properties.Resources.Background;
    MainDrawingArea.BackgroundImageLayout = ImageLayout.Stretch;
    MySpriteController = new SpriteController(MainDrawingArea);
    MySpriteController.DoTick += CheckForKeyPress;
}

private void CheckForKeyPress(object sender, EventArgs e)
{
    bool left = false;
    bool right = false;
    bool space = false;
    bool didsomething = false;
    TimeSpan duration = DateTime.Now - LastMovement;
    if (duration.TotalMilliseconds < 100)
        return;
    LastMovement = DateTime.Now;
    if (MySpriteController.IsKeyPressed(Keys.A) || MySpriteController.IsKeyPressed(Keys.Left))
    {
        left = true;
    }
    if (MySpriteController.IsKeyPressed(Keys.D)||MySpriteController.IsKeyPressed(Keys.Right))
    {
        right = true;
    }
    if (left && right) return; //do nothing if we conflict
    if (left)
    {
        if (LastDirection != MyDir.left)
        {
            Spaceship.SetSpriteDirectionDegrees(180);
            //We want to only change animation once.  Every time we change
            //the animation, it starts at the first frame again.
            Spaceship.ChangeAnimation(0);
            LastDirection = MyDir.left;
        }
        didsomething = true;
        Spaceship.MovementSpeed = 15;
        Spaceship.AutomaticallyMoves = true;
    }
    if (right)
    {
        if (LastDirection != MyDir.right)
        {
            Spaceship.SetSpriteDirectionDegrees(0);
            Spaceship.ChangeAnimation(0);
            LastDirection = MyDir.right;
        }
        didsomething = true;
        Spaceship.AutomaticallyMoves = true;
        Spaceship.MovementSpeed = 15;
    }
    if(!didsomething)
    {
        LastDirection = MyDir.stopped;
        //No keys pressed.  Stop moving
        Spaceship.MovementSpeed = 0;
    }
}
```

One of the things that the above code does, which is probably not immediately apparent, is that it does not repeatedly tell something to do what it is already doing. It tracks the "LastDirection", which is the direction the spaceship was traveling previously. If we are already going the same direction that our key-presses are telling us we should be going, the code leaves the sprite alone. The sprite has a direction and speed, and will continue to go that direction at that speed until the key is released, or something stops it. The ShootingDemo, from which the above code was pulled, has the SpaceShip set to not be able to go outside the bounds of the PictureBox. So it will stop at the edge.

The reason we do not tell the SpaceShip to move every time is because it takes a little bit of time for it to get started. There is just a little bit of housecleaning that needs to be done before it starts off on its way. If you tell it to move again, it starts the housecleaning again. It will barely move if you tell it to move the same direction every 10th of a second. And the animation stuff looks terrible if you tell it to restart the movement every 10th of a second.

The right way to do it is to tell it which way to go, and only change the direction when something acts on the sprite to change the direction.

Code for resizing the picturebox

You do not need much code for resizing. What you want to do is to have the picturebox anchored to the window so that the picturebox will resize when the window does. The sprite library should handle the resizing of the image within the picturebox quite nicely. You may want to put a limit on how small, or how large the window can go. But the easiest way to do that is by setting the Form.MaximumSize and Form.MinimumSize.

Program design (needs filling out)

This section is geared towards helping describe how you may want to design a game that is built with SpriteLibrary. This is a suggestion only. It will probably help you to read this section, as it does contain a fair bit of hard-won thought. BUT, every developer must make a lot of their own decisions. There is nothing magical about this.

One Form

It is a lot simpler to program a game if it is all within one form. While you can develop games using multiple forms, things can get very messy. The SpriteLibrary was designed for use with one form, and bouncing between different forms can result in some very strange behavior. In my AdventureDemo (http://www.codeproject.com/Articles/1110409/Using-SpriteLibrary-to-develop-a-Role-Playing-Game), I used multiple forms. That was a bad idea. If I closed the main game window during a battle, the game exited and the game-menu popped up. Then, the sprites continued to fight, even though they were technically gone. The party continued to take damage, and sometimes die. It was a little disconcerting.

The game is a lot simpler to deal with if you only have one form. This means you draw your menu on the same PictureBox that you have your game running on. You can change the PictureBox background, so you have a different looking window when the window is up, or you can leave it be. It does not really matter too much. But, please, try using putting it all in one Form if possible.

Having multiple PictureBoxes works fairly well, particularly with linked sprite-controllers and / or a SpriteDatabase. Both of those allow you to define a sprite once but use it on multiple picture boxes.

Enum for Game Mode

To get everything to run in one PictureBox, it is usually easiest to have an Enum, which tells you which mode you are in. The enum might be something like:

```
public enum GameMode { Menu, Menu_Settings, Menu_HighScores, Playing, Won, Lost }
```

You will have a "Tick" function (more information below), and the first thing in your tick function is to determine what mode you are in, and process things accordingly.

Instantiate Named Sprites (or instantiate on demand)

You often want to load your sprites at the beginning. Sometimes, you will want to make a function, called "GetSprite", which will find a named sprite. If the named sprite does not exist, it will load it and name it. If you do that, you can load and name the sprite the first time you need it. This only works in a few cases; if the process of instantiating a sprite is quick enough that it can be done in the middle of action. Otherwise, you will want to pause long enough at the beginning of the game to load all your sprites.

Function to change mode

You should create a function to change the game mode. We will place a call for this in a moment, but for now, we do something like:

> Clear all sprites: The sprite controller has a process for clearing all sprites that are duplicates of named sprites. Basically, anything on the screen gets removed.
>
> Start Game: Change the background to whatever the game background is. Duplicate all the sprites you need for the game and place them in their starting positions.
>
> Start Menu: Change the background to whatever the menu background image is. Duplicate all the menu sprite buttons and place them in their respective menu spots. (You should have menu buttons for things like "play", "settings", "high-scores", and "exit")
>
> Start Settings: This is usually a special "menu" that has items like: "volume", "speed", etc.
>
> Start Game Won: This is usually a special menu that shows you a "you have won!" message, as well as your score, and any special "win" message. You usually have a "next" button or something to go on to the menu from there.
>
> Start Game Lost: This is usually a special menu that shows you a "you have Lost!" message, as well as your score, and any special "lost" message. You usually have a "next" button or something to go on to the menu from there.

Game Tick

The Game Tick function is where most of the work on your game goes. The Tick happens every few milliseconds. When you create your SpriteController, you need to add your Tick function with something like: MyController.DoTick += GameTick;

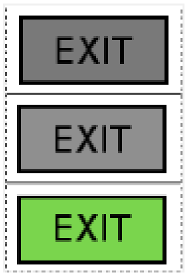
You usually check first to see if you are switching your mode to something else:

```
    If(newmode != currentmode) ChangeMode(newmode);
```

And then, you have an if (or switch) block to process things depending on which mode you are in:

```
    If(currentmode == GameMode.Playing) {
        CheckForKeypresses();
        CheckForChangeInLevel();
    }
```

Making a Game Menu



Your Menu is usually set up by creating a bunch of functions, one for each MenuButtonSprite. A MenuButtonSprite is basically a sprite that does something when you click it.

For an example, we will use an "exit" sprite button. This button will look something like the one on the left. We load the sprite as three different animations. The first one, at the top, we will use for when we show the menu. The second menu item is just a little bit lighter in color, and we will use that when we mouse-over the menu item. And we will use the green one when we click on it.

We will want to have some functions for this button. Let's call them ExitClick, ExitMouseOver, and ExitMouseLeave.

When we instantiate this menu button, we will do something like the following:

```
Sprite ExitButton = Sprite(new Point(0, 0), MyController, Properties.Resources.Exit, 100, 100, 
1000, 1);
ExitButton.AddAnimation(new Point(0, 50), Properties.Resources.Exit, 100, 100, 1000, 1);
ExitButton.AddAnimation(new Point(0, 100), Properties.Resources.Exit, 100, 100, 1000, 1);
ExitButton.Click += ExitClick;
ExitButton.MouseOver = ExitMouseOver;
ExitButton.MouseLeave = ExitMouseLeave;
ExitButton.SetName("ExitButton");
```

Now, let's make our different functions.

```
public void ExitClick(object sender, SpriteEventArgs e)
{
    If(sender is Sprite)
    {
         Sprite ExitSprite = (Sprite)sender;
         Sender.ChangeAnimation(2);
         NewGameMode = GameMode.Exiting;
     }
}

public void ExitMouseOver(object sender, SpriteEventArgs e)
{
    If(sender is Sprite)
    {
         Sprite ExitSprite = (Sprite)sender;
         Sender.ChangeAnimation(1);
    }
}
public void ExitMouseLeave(object sender, SpriteEventArgs e)
{
    If(sender is Sprite)
    {
         Sprite ExitSprite = (Sprite)sender;
         Sender.ChangeAnimation(0);
    }
}
```

Conclusion

I hope you find the SpriteLibrary helpful for creating games. It is not an all-powerful gaming system, but does do a fair bit of work for you. If you have questions for how to use it, you can leave messages on the CodeProject site:

http://www.codeproject.com/Articles/1085446/Using-Sprites-Inside-Windows-Forms