Inside SpriteWorld 2

by Tony Myles,
Karl Bunker
and Vern Jensen

version 2.2

Table of Contents

General Introduction

SpriteWorld is a collection of routines that you can use to implement smooth animation in your applications. SpriteWorld was designed with an eye towards the style of animation seen in arcade games in particular. You can use SpriteWorld to…

- perform smooth multi-layered animation

- perform collision detection
- create animations from color icon or pict resources
- use custom bit blitting routines for off screen and/or on screen drawing
- synchronize animation with millisecond accuracy

- perform simple about box animations or write full blown arcade games

History

SpriteWorld 1.0 was written by Tony Myles; his most recent version appeared in April, 1994. Unfortunately, Myles has been unable to support SpriteWorld since then, and hence has not addressed several bugs and limitations that have been reported. In late 1994, I (Karl Bunker) started modifying SpriteWorld in order to use it in a game I was developing. After this first round of revamping, I began collaborating with Vern Jensen, and between us we developed a version of SpriteWorld that we felt was worthy of being released as an update or alternative to Myles' original. We have made some rather extensive changes to Myles' code, but SpriteWorld should still be considered Tony Myles's creation. The wonderfully elegant structure of SpriteWorld, with its World/Layer/Sprite/Frame hierarchy is Tony's, and many of the most crucial sections of the code are still largely his. Our contribution has been to fix things up, add some things, and to update the documentation. For a more detailed comparison of this version of SpriteWorld with Myles', see the "What's New" file included as part of the documentation package. This version of SpriteWorld is distributed with Tony Myles' permission.

Acknowledgments

Tony Myles' Acknowledgments
SpriteWorld would not have come to exist if it were not for the work of many people in the Macintosh game development community. Most notably John Calhoun, and the late Duane Blehm. Thanks to Apple's Developer Support Center for tech notes and sample code. Thanks to Ed Harp for our friendship, and our late night explorations into animation techniques. Thanks to David F. Johnson for just being who you are. Thanks to Ben Sharpe for some great blitter code. Thanks to all the people at Pharos for their support and help.

Karl & Vern's Acknowledgments
Our main indebtedness, of course, is to Tony Myles. Our work has been to build on the excellent foundation that he provided. It is important to note that any bugs found in this version of SpriteWorld must **not** be considered Tony Myles' responsibility.

A terrific contribution to SpriteWorld 2 was made by Christofer Åkersten (chris@basesoft.se). He wrote the BlitPixieAllBit routines, providing a depth-independent 68K blitter that will be especially useful to those working with screen depths less than 8 bits.

Likewise, Brian Roddy (formerly with Apple) made the excellent contribution of several blitters for SpriteWorld 2. These include a multiple-depth PPC blitter, special blitters for translucency and lighting effects, and blitters to perform rotating and scaling on Sprites.

Anders Björklund contributed the excellent DoubleRect blitter that eliminates the seams from scrolling animations. In addition, he was of great assistance by tracking down and fixing some bugs in the AllBit blitters, and made various other contributions as well.

Stefan Sinclair has not only created a version of SpriteWorld for C++ and X-Windows, but he is active on the SpriteWorld mailing list and on comp.sys.mac.programmer.games, answering SpriteWorld questions and helping people out in general.

Peter Lewis (peter@stairways.com.au) wrote the Pascal interface files, and also contributed some code and made many excellent suggestions which have enhanced SpriteWorld considerably.

Thanks also to Peter Lewis and Stairways Software <http://www.stairways.com/> for the software contribution.

Thanks to Greg Galanos and Metrowerks, for the donation of 2 copies of CodeWarrior.

Thanks to Onyx Technology, Inc. <http://www.onyx-tech.com/> for donating a copy of their debugging tool Spotlight.

Thanks to Tim Carroll of Apple DTS for his help with the PPC 8 bit blitter.

And a big thanks to Cary Farrier of Apple, and Apple Inc. in general for their assistance with this project.

Support

If you have a question about SpriteWorld or need help fixing a problem in your program, you can send your question to the SpriteWorld mailing list (see the SpriteWorld FAQ for more information) or post a message to comp.sys.mac.programmer.games.

If you have a question that isn't appropriate for those groups, or if you have suggestions, bug reports, or just general comments about SpriteWorld, you can contact Vern Jensen at:

Jensen@loop.com

Karl would also like to hear from you if you are just saying thanks or announcing the release of a game that uses SpriteWorld. You can reach him at:

KarlBunker@aol.com

If you write to both of us, place the second address in either the "To:" or "Cc:" field of your message, so we both know that the other one received your message. (i.e. don't use the "Bcc:" field.)

The “official” source for the latest versions of SpriteWorld 2, its documentation and demos is the Web page:

http://users.aol.com/SpriteWld2/

and its corresponding ftp site:

ftp://users.aol.com/SpriteWld2/

If you think you have found a bug in SpriteWorld, try to modify one of the simple demo programs so that it reproduces the bug, and then send that code to Vern. Remember that in order to achieve speed, there is little or no error checking in SpriteWorld’s animation routines. (There is more error checking in the initialization routines.) Therefore, by feeding incorrect parameters to a SpriteWorld routine, you can easily to make it do Bad Things. Don’t be too quick to assume that such misbehavior is the result of a bug in SpriteWorld.

Important:

SpriteWorld is intended for programmers. It is not a “game construction kit” that non-programmers can use to create games. If you encounter problems in your efforts to use SpriteWorld, you should do either or both of two things: 1) Solve the problem(s) yourself. 2) Ask for help in a Mac programmer conference or Usenet newsgroup; comp.sys.mac.programmer.games is recommended. You can also join the SpriteWorld mailing list, which was created specifically for this purpose. (See the SpriteWorld FAQ for more information.) Both Vern and Karl read the SpriteWorld mailing list, and will try to answer your question if nobody else can.

To use SpriteWorld in your own project, start with one of the smaller demo programs provided as part of the SpriteWorld package. Slowly add in your own code and your own Sprite graphics. Refer to the other demo programs for more information and sample code. If your project breaks, go back a few steps to isolate the problem. Important: replace everything that affects the look and feel of your program with your own code. Otherwise, a new breed of "me too" games will be released, which is the last thing any of us wants. Things to replace include all the graphics, as well as things such as the ScrollingWorldMoveProc (if you're making a scrolling game), since even that affects the feel of a game.

Distribution

Tony Myles originally released SpriteWorld as freeware, and naturally that policy is continued with this version. Programs created using SpriteWorld may be distributed as freeware, shareware, or commercially. You aren't required to give notice that you used SpriteWorld in the credits to your program, though you're certainly welcome to do so. Beyond that, Vern and I would very much appreciate being informed about any SpriteWorld-using releases.

The SpriteWorld package itself may be distributed by any means, provided that it is distributed and presented as freeware.

Disclaimer

The package is provided “as is”, without guarantees of any kind. None of its authors can be held responsible any for damage or loss of any kind that may occur from using it.

Contribute!

Obviously, SpriteWorld is already a collaborative endeavor. You can contribute too, and help to make it even better. Some things that would enhance SpriteWorld are:

- Alternate formats for the sprites. For example, an RLE blitter, and a way to save Sprite images in RLE format, which would result in faster loading of Sprites and lower memory requirements. Or maybe a way to save the raw data from a Sprite's GWorld to a resource, which would speed up loading time and save memory. (Although it would probably need some sort of compression, since otherwise it would take up a lot more disk space.)
- Better demos. Put together something that's fun and shows off SpriteWorld's capabilities, and maybe we'll include in the package. A "real" game demo would be nice.
- We're interested in anything that would make SpriteWorld better, but it should fit into the existing structure and focus of SpriteWorld. We don't want to add more general-purpose routines to the package, because we have to keep its size and scope manageable.

Known Bugs

Currently, SpriteWorld does not work very well with multiple monitors. I used to say that SpriteWorld worked just fine on multiple monitors as long as you didn't use direct-to-screen drawing, but I have now received reports that problems occur even when direct-to-screen drawing is not used. For instance, if a SpriteWorld is created from a window on one monitor, and then that window is moved to another monitor, its contents will no longer be drawn. Since neither Karl nor I have access to multiple monitor systems, we unfortunately can not do anything about the problem for the time being. If you need multiple monitor support, it seems your only option is to use the SWStdWorldDrawProc as your screenDrawProc, and not allow the window to be moved onto a different monitor than it was created on.

Introduction to Sprites

“Sprite” is a technical term meaning an animated object that appears on the computer’s screen and may move around or exhibit other interesting behavior. A good example is an arcade game in which you pilot a space ship against hordes of alien invaders. In this case the ship and the invaders can be considered Sprites.

In terms of SpriteWorld’s implementation, a Sprite is basically a data structure that contains a series of the graphic images or Frames of the Sprite, a rectangle that specifies where on the screen the Sprite is to be drawn, and various other parameters that specify how far it should move and in what direction, as well as when it should move and be drawn. SpriteWorld provides a suite of routines you can use to create Sprites, and specify all of their animation characteristics.

Sprites can have one or more Frames. The current Frame of a Sprite is the image that will be drawn when the animation is rendered on the screen. If a Sprite has multiple Frames, its screen image can be animated by advancing the current Frame in sequence. At the same time the screen position at which the Sprite’s current Frame will be drawn can be adjusted, thus giving the illusion of movement.

Sprites also support the notion of collisions. As a Sprite moves around on the screen, it may come in contact with another Sprite. This is called a collision. SpriteWorld provides routines to detect collisions and act upon them. By default nothing will happen; the Sprites will harmlessly pass through each other. On the screen the Sprite images will smoothly overlap one another.

SpriteWorld uses Sprites to drive the animation. Each frame of the animation is built by processing the Sprites and then drawing them. When the Sprites are processed, their new positions are calculated based on their installed movement procedures, and their current Frame is changed based on their Frame advance parameters. In general the Sprites are used as a mechanism to shield you from all the gory details of the animation.

Introduction to SpriteWorld

Unlike the Amiga and other game machines, the Macintosh has no Sprite animation hardware built-in. Sprite animation must therefore be implemented in software. SpriteWorld is an attempt to implement a Sprite-based animation architecture on the Macintosh.

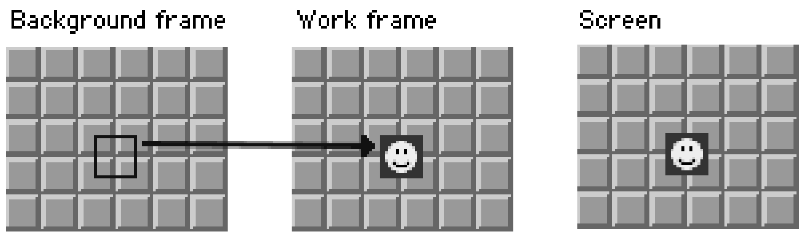
SpriteWorld achieves its smooth animation using a frame differential technique. This means that for each frame of the animation, only the areas of the screen that have changed are actually drawn. This technique uses a double buffering scheme, meaning that two offscreen areas, one to keep a fresh copy of the background and the other to serve as a work area, are used to render the animation before it is drawn on screen. This calls for a three step process to building a frame of the animation.

- A section of the background corresponding to the Sprite’s old location is copied from the background frame to the offscreen work frame. This step erases the Sprite from its old position in the work frame, preparing it for the Sprite’s new location.
- The current Frame of the Sprite is then drawn in the Sprite’s current position in the offscreen work area.

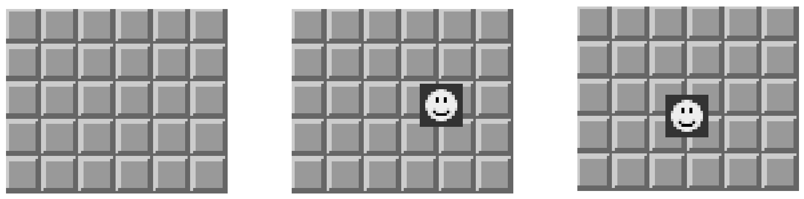
(Note the various uses of the word “frame”: A “Frame” (capitalized) refers to a single graphic image of a Sprite; each Sprite may have several Frames, each one representing a different stage in the Sprite’s appearance. A “frame” of animation refers to a single iteration of the entire screen image, comparable to a frame of a movie. And just to make things more confusing, the two main offscreen areas are sometimes referred to as the “background frame” and the “work frame”.)

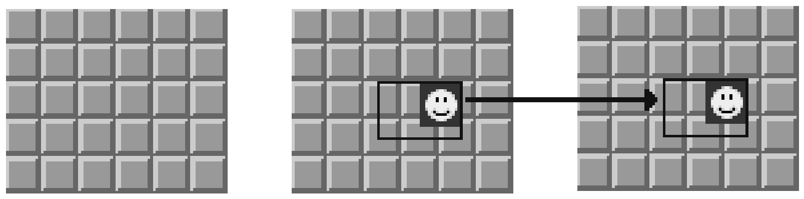
• The union of the Sprite’s last position and its current position is then copied from the work area to the corresponding position on the screen, effectively erasing the Sprite from its old position on screen, and drawing the Sprite in its new position simultaneously.

This simple animation technique is commonly used by games and animation applications on the Macintosh. SpriteWorld’s implementation of this technique employs several improvements and optimizations for smooth overlapping of Sprites, skipping inactive Sprites, etc.



Step 1 : The Sprite is erased from its old position in the work frame by copying a section of the background frame.


Step 2 : The Sprite is drawn to its new position in the work frame.


Step 3 : The union rect of the Sprite's old position and its new position is copied from the work frame to the screen. This erases the old Sprite from the screen and draws the new one in a single step.

When creating animations using SpriteWorld you will deal primarily with four data structures: SpriteWorlds, SpriteLayers, Sprites, and Frames. These four structures have a containment relationship in that SpriteWorlds contain any number of SpriteLayers, which contain any number of Sprites, which contain one or more Frames.

SpriteWorlds

SpriteWorlds provide a context for the animation to take place. The context provided by a

SpriteWorld is essentially the graphics environment for the animation both on and off screen. Everything in the animation happens under the domain of a SpriteWorld. Your program can create any number of SpriteWorlds.

A SpriteWorld contains a frame for the offscreen background area, the offscreen work area, and the screen itself. The frames for the background and the work area are offscreen GWorlds whose size you determine when you create the SpriteWorld. The screen frame is just a data structure for accessing the screen's GWorld or its video RAM. A SpriteWorld also maintains a list of the SpriteLayers that are taking part in the animation. There are routines for adding and removing Layers from a world. There can be any number of Layers in a given SpriteWorld.

In terms of the actual drawing, the SpriteWorld is responsible for erasing and redrawing the Sprite offscreen and updating the Sprite onscreen. You can install custom routines to handle these functions by writing a custom pixel blitter. By default SpriteWorld uses QuickDraw's CopyBits routine for all drawing operations. Built into SpriteWorld are a set of high speed pixel blitters called BlitPixie, and an option for compiled Sprites.

SpriteLayers

SpriteLayers are used to maintain groups of related Sprites. Using SpriteLayers you can animate sets of Sprites in separate overlapping planes, creating the illusion that the Sprites are passing in front of and/or behind other Sprites. When drawing occurs, each Sprite in each Layer, and each Layer in the world, is drawn consecutively, one overlapping the other.

The first Layer is drawn first, the last drawn last, so that the Sprites in each Layer overlap each other in a consistent order.

Aside from the animation, this layering facility is also used by the collision detection mechanism. If you arrange your Sprites in logical layers, e.g. the good guys in one layer and the bad guys in another, then detecting collisions is simply a matter of checking one layer of Sprites against another. You can also detect collisions between Sprites residing within a single layer.

Sprites

Sprites are the star of the show. Any animation you create will consist of one or more Sprites. These Sprites move about the screen and do interesting things according to parameters you can specify using the routines provided. You can specify the timing, direction, and distance a Sprites moves at any one time. By installing a move routine, your Sprites can exhibit extremely complex behavior such as simulated gravitational forces.

A Sprite contains one or more Frames. As a Sprite moves it may change which Frame is to be currently drawn, producing the illusion of animation. You can specify the timing of these Frame changes, and by installing a Frame routine, you can perform more sophisticated frame animation such as rotating a space ship when certain key is pressed.

Sprite movement and Sprite Frame changes can also easily be linked together. For example, in an animation of a walking figure the position of the Sprite on the screen can be controlled by which Frame image is current, thus avoiding the “sliding feet” syndrome so common in animations of walking.

When one Sprite overlaps another, a collision has occurred. By installing a collision routine the Sprite can take action when a collision is detected, such as playing an explosion sound.

Frames

Frames are used to maintain the individual graphic images of a Sprite. Each Frame consists of a data structure, an offscreen GWorld in which the actual image is stored, and a mask.

The SpriteWorld Libraries and Source Code

In order to use SpriteWorld in your programming project, you must include the SpriteWorld source code in your project. All of the files in the Sources folder (contained in the SpriteWorld Files folder) must be added to your project except for BlitPixieAllBit.c and BlitPixieDoubleRect. These files must only be added if you want to take advantage of the AllBit or DoubleRect blitters. As of the 2.1 release, Scrolling.c is now required in all projects, regardless of whether your applications make use of the scrolling routines or not.

The SpriteWorld Libraries have been removed as of the 2.1 release, since the libraries seem to break with every other version of CodeWarrior that is released. In addition, it's a good idea to use the source code itself in your projects, because then you can enter SpriteWorld functions when debugging.

The SpriteWorld 2 source code is made freely available as part of the package. You may want to study this to gain a better understanding of some SpriteWorld functions, or to copy and modify some functions for use in your own code. Studying the source code will also enable you to make more advanced uses of SpriteWorld; for example, implementing “procedural” Sprites in which the Sprite image is modified in response to user interaction.

The SpriteWorld “package”:

Depending on how you obtained SpriteWorld, you may not have all the folders mentioned below. The folders that make up the complete SpriteWorld 2 package are:

Documentation

SpriteWorld Examples

SpriteWorld Files

Complete documentation for SpriteWorld is in the “Documentation” folder.

The “SpriteWorld Examples” folder holds several demo programs using SpriteWorld. Reading through the source code to these will be an important part of learning how to use SpriteWorld.

The SpriteWorld Libraries, source code, headers, and the source to some ancillary utility routines used by the demo programs are all in the folder “SpriteWorld Files”.

For Think C users, the quickest way to get started is to copy “SpriteWorld Files” into your main Symantec/Think folder, or an alias into the “Aliases” folder within your main Symantec/Think folder. This will make the Libraries, header files and source files available to the demos. CodeWarrior users can copy “SpriteWorld Files” into their “Metrowerks CodeWarrior” folder; CodeWarrior currently doesn't support aliases.

Using SpriteWorld

Getting Started

Performing animation using SpriteWorld involves 4 **core** steps (in **bold**), and 2 initialization or housekeeping steps:

- Initialize the SpriteWorld package.
- **Create the various pieces: the SpriteWorld, the SpriteLayers, the Sprites, and the Frames.**
- **Assemble the various pieces: add the Frames to the Sprites, add the Sprites to the SpriteLayers, add the SpriteLayers to the SpriteWorld.**
- **Set the various movement and Frame advance parameters that define a Sprite's behavior.**
- **Drive the animation using SWProcessSpriteWorld and SWAnimateSpriteWorld in a tight loop. Collision detection may optionally be performed here.**
- The last step is to ask the SpriteWorld package to clean up. This disposes of all the SpriteLayers, Sprites, and Frames that were created earlier.

While the animation is running you may add or remove individual Sprites or entire SpriteLayers, change a Sprite's movement and Frame advance characteristics, switch the order of SpriteLayers in the SpriteWorld, and perform any number of other manipulations. Thus an animation can be a simple "set up and forget" proposition, or an intensely dynamic, complex and user-interactive programming endeavor.

Creating an animation

Before SpriteWorld can be used it must first be initialized with a call to SWEnterSpriteWorld. SWEnterSpriteWorld performs some checks to see if SpriteWorld can run, and then sets up some internal data structures. You must call SWEnterSpriteWorld before calling any other SpriteWorld routine.

```
err = SWEnterSpriteWorld();
```

Once the SpriteWorld package is initialized you can start creating the pieces that will make up your animation. The central piece to any animation is the SpriteWorld. If your application has a window in which you would like to display the animation you can easily create a SpriteWorld by calling SWCreateSpriteWorldFromWindow.

```
err = SWCreateSpriteWorldFromWindow(&spriteWorldP, gameWindowP,
                                &worldRect, &backRect, maxDepth);
```

For even the simplest animation you must create at least one SpriteLayer. This is accomplished by calling the SWCreateSpriteLayer function. There is no limit to the number of SpriteLayers that you might use in an animation.

```
err = SWCreateSpriteLayer(&spriteLayerP);
```

An easy way to create a Sprite with a set of Frames is by calling SWCreateSpriteFromCicnResource.

```
err = SWCreateSpriteFromCicnResource(spriteWorldP,
                            &newSpriteP,                          NULL,
                            kCIconResourceID,
                            kNumberOfFrames,
                            kFatMask);
```

Assembling The Pieces

Before an animation can be run the pieces that you have created must be assembled. This is accomplished by adding the Sprites to the SpriteLayers, and adding the SpriteLayers to the SpriteWorld.

```
      // repeat this call for each Sprite
SWAddSprite(spriteLayerP, newSpriteP);

      // repeat this call for each SpriteLayer
SWAddSpriteLayer(spriteWorldP, spriteLayerP)
```

Defining Sprite Behavior

Before and possibly during the animation you will want to define the movement behavior of your Sprites. The following code snippet covers most of the basic behavioral parameters you will be dealing with.

```
      // set the Sprite’s initial location (horiz, vert)
SWSetSpriteLocation(newSpriteP, 100, 100);

      // set how often a Sprite moves in milliseconds
SWSetSpriteMoveTime(newSpriteP, 30);

      // set the Sprite’s movement direction/distance in pixels
SWSetSpriteMoveDelta(newSpriteP, 10, 5);

      // set the Sprite’s moveProc to define its motion behavior
SWSetSpriteMoveProc(newSpriteP, myBounceSpriteMoveProc);

      // set how often the Frame changes in milliseconds
SWSetSpriteFrameTime(newSpriteP, 1000/30);

      // set the range of Frames to be cycled through
SWSetSpriteFrameRange(newSpriteP, 0, 10);
```

Driving The Animation

Once everything is in place the animation is driven by repeated calls to SWProcessSpriteWorld and SWAnimateSpriteWorld. SWProcessSpriteWorld runs through each Sprite installed in each SpriteLayer in the SpriteWorld and advances the position of the Sprite's destination rectangle according to each Sprite's movement characteristics. SWAnimateSpriteWorld draws each Sprite that needs to be drawn on the screen. Rigorous checking is done on each Sprite to determine if it really needs to be drawn since a considerable time savings is gained by skipping even one small Sprite.

```
      // core animation loop
while (animationIsRunning)
{
            // move the Sprites to their new positions
      SWProcessSpriteWorld(spriteWorldP);

            // render a frame of the animation
      SWAnimateSpriteWorld(spriteWorldP);
}
```

Movement routines

In order for a Sprite to move, it must have a movement routine, or moveProc. This is a user-defined routine which is called by SpriteWorld when it is time for a Sprite to be moved. The moveProc you define can make use of such SpriteWorld routines as SWOffsetSprite or SWMoveSprite. SWBounceSprite and SWWrapSprite can also be used in a moveProc to produce a bouncing or wrap-around Sprite motion. A moveProc is installed with SWSetSpriteMoveProc:

```
SWSetSpriteMoveProc(mySpriteP, MyMoveProc);
```

The moveProc, in turn, has this form:

```
SW_FUNC void MyMoveProc(SpritePtr srcSpritePtr)
```

Frame routines

Frame routines, or frameProcs, are analogous to moveProcs. If a frameProc routine is installed for a Sprite, then that routine will be called by SpriteWorld when it is time for the Sprite to change its Frame. However, although a Sprite can't move without a moveProc, it can change Frames without a frameProc. By setting a Sprite's Frame range with SWSetSpriteFrameRange, the Sprite can be set to automatically cycle through the specified set of Frames. A frameProc will be used for allow more complex forms of Frame-changing, or to trigger some other action when a Sprite's Frame changes. A frameProc is installed with SWSetSpriteFrameProc:

```
SWSetSpriteFrameProc(mySpriteP, MyFrameProc);
```

The frameProc, in turn, has this form:

```
SW_FUNC void MyFrameProc(SpritePtr srcSpriteP,
                             FramePtr curFrameP,                     long
                           *frameIndex);
```

Detecting Collisions

Since collision detection is such an application specific problem, SpriteWorld employs a collision detection mechanism that is very versatile, but simple and easy to use.

The SWCollideSpriteLayer function is used to check the Sprites in the source SpriteLayer against the Sprites in the destination SpriteLayer for collisions. In order to check for collisions between the Sprites of a single SpriteLayer, you must pass the same SpriteLayer as the source and the destination.

```
    // see if our ship has collided with any enemies
SWCollideSpriteLayer(mySpriteWorldP,
                   shipSpriteLayerP,
                   enemySpriteLayerP);


    // we may want to see if any of the enemies
    // have collided with each other
SWCollideSpriteLayer(mySpriteWorldP,
                   enemySpriteLayerP,
                   enemySpriteLayerP);
```

When a collision is detected the collision routine, if any, of the source Sprite is called. For anything useful to happen you must install a collision routine, or collideProc, in the Sprites you expect to be involved in collisions. A collideProc is installed with SWSetSpriteCollideProc:

```
SWSetSpriteCollideProc(shipSpriteP, myCollideProc);
```

The collideProc has this form:

```
SW_FUNC void MyCollideProc(SpritePtr srcSpriteP,
                   SpritePtr dstSpriteP,              Rect*
                  sectRect);
```

A collision is defined as the condition that occurs when the rectangle that defines the current screen location of a Sprite intersects the corresponding rectangle of another Sprite. This may or may not mean that the actual images of the Sprites as they appear on screen overlap. Therefore it is up to the collision routine you provide to determine a more precise definition of a collision for your Sprites if necessary. The SpriteWorld routines SWRadiusCollision, SWPixelCollision and SWRegionCollision can help with this.

Sprite Masks and Self-Masking

Programmers preparing to use SpriteWorld will probably be familiar with the concept of masks. A mask is a black and white image in which the black pixels indicate which parts of the Sprite image should actually be drawn—which parts are not background, in other words. Masks are included as part of cicn resources; Sprites created from PICT resources may or may not require an additional PICT for the mask. When creating a Sprite with SWCreateSpriteFromPictResource, SWCreateSpriteFromSinglePict, or SWCreateSpriteFromSinglePictXY, if you pass the same resource ID for both the Sprite

PICT and the mask PICT, SpriteWorld will create a mask from the Sprite’s own image. This is called self-masking. When this is done, any pixels in the Sprite image that are not pure white will become part of the mask. If you wish a color other than white to be used as the Sprite's "transparent" color, you can call SWSetTransparentColor to change it to any RGB color. Then all pixels in the Sprite's image that do not match that color will become part of the mask.

As is described in the documentation to SpriteWorld’s Sprite-creation routines, SpriteWorld allows two types of mask to be created: a region mask and a pixel mask. There is also a “fat” mask, in which both types are created. It should be noted that the initialization of a Sprite will be considerably slower if a fat mask is requested, as compared to only a pixel mask. If only a pixel mask is made, then you must use a custom blitter such as BlitPixie to draw the Sprite, rather than CopyBits (unless your Sprite is rectangular and has no mask at all). Also, if only a pixel mask is created, you can’t use any functions that require a QuickDraw region, such as SWRegionCollision.

SpriteWorld DrawProcs

The business of animation is primarily one of copying images, and SpriteWorld provides you with a variety of different methods of copying (or “drawing”; the terms can be used interchangeably here) to fit different situations. Some of these methods are discussed in more detail later in this file. Each drawing procedure is in the form of a “drawProc”. The routine SWSetSpriteDrawProc installs a drawProc for a particular Sprite. Likewise, SWSetSpriteWorldOffscreenDrawProc sets the drawProc SpriteWorld uses for copying between the offscreen work frame and the offscreen background frame. SWSetSpriteWorldScreenDrawProc sets the drawProc used for drawing to the screen. Here is a list of the drawProcs available in SpriteWorld:

SWStdSpriteDrawProc	CopyBits; the default drawProc for Sprites.
SWStdWorldDrawProc	CopyBits; the default offscreen and screen drawProc.
BlitPixieAllBitMaskDrawProc	Depth-independent BlitPixie with a mask.
BlitPixie8BitMaskDrawProc	BlitPixie with a mask; optimized for 8 bit graphics.
BlitPixieAllBitPartialMaskDrawProc	
BlitPixie8BitPartialMaskDrawProc	BlitPixie with a “partial” mask.
BlitPixieAllBitRectDrawProc	
BlitPixie8BitRectDrawProc	BlitPixie with no mask (draws rectangular areas only).
BPAllBitInterlacedMaskDrawProc	
BP8BitInterlacedMaskDrawProc	Interlaced BlitPixie with a mask. Only every other horizontal line is drawn.
BPAllBitInterlacedPartialMaskDrawProc	
BP8BitInterlacedPartialMaskDrawProc	Interlaced BlitPixie with a partial mask.
BPAllBitInterlacedRectDrawProc	
BP8BitInterlacedRectDrawProc	Interlaced BlitPixie with no mask.

CompiledSprite8BitDrawProc	For Compiled Sprites.

Note that only SWStdWorldDrawProc and the “RectDrawProcs” can be used as the screenDrawProc and offscreenDrawProc. The RectDrawProcs can also be used for drawing Sprites, but only if the Sprite is rectangular. The MaskDrawProcs and CompiledSprite8BitDrawProc are used for Sprites with irregular shapes. When creating a Sprite, the appropriate maskType must be defined for the type of drawProc you intend to use with that Sprite. See SWSetSpriteDrawProc for more information.

The various “BlitPixie” drawProcs make use of custom blitters included in SpriteWorld. These BlitPixie routines are generally much faster than CopyBits. The BlitPixie8Bit routines are optimized for 8 bit graphics, and exist in two versions: one written in 68K assembly language and one in C specifically optimized for PPC-native compiling. Another set of BlitPixie routines are also available; these are the BlitPixieAllBit routines. The BlitPixieAllBit routines are depth-independent, meaning they will work for all bit depths, from 1 through 32. (Note, however, that the PPC version only works in 8-bits or higher.) If your animation is only going to appear at 8 bits depth, then the BlitPixie8Bit routines should be used, as these are somewhat faster than BlitPixieAllBit. One point should be noted about using BlitPixieAllBit at 16 and 32 bits depth: if you install BlitPixieAllBitMaskDrawProc for a Sprite, the drawProc that will actually be used is BlitPixieAllBitPartialMaskDrawProc. This is because due to the bit-wise nature of 16 and 32 bit pixels, BlitPixieAllBitPartialMaskDrawProc is required. (The PartialMask drawProcs are discussed below.)

The BPInterlaced drawProcs are versions of BlitPixie in which every other horizontal line of an image is skipped, and left black. This provides much better speed than standard BlitPixie, at the cost of a darker, coarser-looking image. This can be used to allow your animation to run acceptably on lower-end machines that normally wouldn’t be fast enough. To achieve the best speed when interlacing, **all** of the drawProcs: the offscreenDrawProc, screenDrawProc, and the drawProcs of all Sprites should be one of the BPInterlaced drawProcs.

The PartialMask drawProcs are primarily intended for drawing masked tiles (see the file “SpriteWorld - Tiling”), but they can be used for Sprites as well. These drawProcs are significantly slower than the BlitPixieMask drawProcs, however. The standard BlitPixieMask drawProcs require that the background of a Sprite—the part that doesn’t correspond to the mask image—be white. The PartialMask drawProcs don’t have this constraint. Thus they can be used when the mask of a Sprite (or tile)—and the part you want drawn—corresponds to only a part (hence the “Partial”) of the whole Sprite image.

BlitPixie and Drawing Directly to the Screen

As noted above, SpriteWorld allows you to optionally use custom blitters called BlitPixie for copying images, rather than the standard CopyBits. BlitPixie can be used to move images from one offscreen GWorld to another, and also to draw images to the screen. BlitPixie is considerably faster than CopyBits, and setting up your SpriteWorld to use it will greatly improve the speed of the animation. BlitPixie’s core routines are written in 68K assembly language for the 68K version of SpriteWorld, and in C for the PPC native version.

As discussed above, the SpriteWorld call:

```
SWSetSpriteDrawProc(mySpriteP, BlitPixie8BitMaskDrawProc);
```

will cause that Sprite to be copied from the Sprite’s Frame image to the offscreen work frame using BlitPixie rather than the default CopyBits.

Likewise, the SpriteWorld call:

```
SWSetSpriteWorldOffscreenDrawProc(mySpriteWorldP,
                                  BlitPixie8BitRectDrawProc);
```

will cause SpriteWorld to use BlitPixie when transferring sections of the background image from the background frame to the work frame.

Both of these involve image transfers from one offscreen GWorld to another, and using BlitPixie for these should be no more “dangerous” than using CopyBits.

For one more speed boost, you can use the SpriteWorld call:

```
SWSetSpriteWorldScreenDrawProc(mySpriteWorldP,
                                  BlitPixie8BitRectDrawProc);
```

SpriteWorld will then use BlitPixie for the final step of rendering the animation: copying the Sprite’s image from the work frame to the screen. When this is done, image information is placed directly into the video RAM, bypassing the usual toolbox calls for drawing on the screen. This can provide faster animation, but it should be used with some caution.

For a long time Apple Inc.’s official position was that drawing directly to the screen in this way is not officially supported, and is not guaranteed to work with future hardware or system software. More recently, Apple has stated that direct-to-screen drawing will continue to be supported for the foreseeable future, providing certain guidelines are followed. SpriteWorld 2 follows those guidelines that are its responsibility; other guidelines are the responsibility of the programmer (you), and are described below.

In general, the major problem with direct-to-screen drawing is simply that it’s difficult to guarantee that it has been done correctly and will work for all end users. Drawing to the screen is much more sensitive to differences in user’s hardware and system setups than most code, so it’s harder to test.

Guidelines for drawing directly to the screen:

• Ideally, if you use direct-to-screen animation in your application, the user should be able to turn it off and use CopyBits for screen drawing if problems occur.

• You should hide the mouse cursor for as long as your direct-to-screen animation is running, or not allow it to be shown in the field where the animation is happening. Otherwise, graphical “artifacts” will appear around the cursor if it is placed over a Sprite. It is recommended that you use ShieldCursor to hide the cursor; certain third party video drivers watch for this call and use it to maintain compatibility with direct-to-screen drawing.

• Be sure that the animation stops (or switches to CopyBits for screen drawing) when your application is in the background. Otherwise your animation may be drawn over the contents of other application’s windows.

• If the window that SpriteWorld is using for its animation is movable be sure to use SWWindowMoved.

• Make sure that the window SpriteWorld is using fits on the screen. BlitPixie only clips its drawing to the window, not to the screen, so if your window is hanging off the edge of the screen BlitPixie will likely write to random memory, causing a crash. If you want your window to be able to hang off the edge of the screen, you can call SWWindowMoved after creating the SpriteWorld, but before updating the window. This will ensure that BlitPixie clips its drawing to both the window and the screen.

Compiled Sprites

For the greatest possible speed on 68K Macs at 8 bits depth, SpriteWorld can “compile” Sprites. After a Sprite has been created (using one of the `SWCreateSprite`... calls), it can be compiled using

```
SWCompileSprite(mySpriteP);
```

Once this has been done, you can set the Sprite’s drawing procedure with

```
SWSetSpriteDrawProc(mySpriteP, CompiledSprite8BitDrawProc);
```

This will cause the Sprite to be drawn to the offscreen work frame using code compiled specifically for that Sprite. Animation using compiled Sprites can be as much as about 40% faster than with BlitPixie.

Some technical notes about compiled Sprites:

• When a Sprite is compiled, the mask image of the Sprite is used to create a series of 68K machine language instructions. This package of instructions is then attached to the Frame data structure of each Frame of the Sprite. When the Sprite is drawn using CompiledSprite8BitDrawProc, these instructions are executed, accessing the Frame’s image and quickly blitting only the masked, or non-transparent, parts of the image to the destination work frame. This process is more correctly called “mask compiling”, but using that term would probably cause confusion, even if technically correct.

• Because the code used in compiled Sprites is 68K machine language, BlitPixie would be faster than compiled Sprites in the Power PC native version of SpriteWorld, so compiled Sprites are not implemented in the PPC native version. You can still call SWCompileSprite in a PPC native SpriteWorld, but it will do nothing, and setting a Sprite’s drawProc to CompiledSprite8BitDrawProc will actually set it to BlitPixie8BitMaskDrawProc.

• If you are going to clone a Sprite, you can compile the Sprite either before or after cloning it. Either way, you only need to compile one Sprite in the clone “family”; the compiled data will be available for all the clones.

• Compiling takes a little time and may add slightly to the initialization time of your program (assuming you compile the Sprites at initialization time), depending on the number of Sprites and Frames being compiled, their size, the complexity of the mask, and the speed of the platform. Compiling also requires some additional memory.

• For the optimum combination of speed and safety in a 68K project with 8 bit graphics, use CompiledSprite8BitDrawProc for the Sprites’ drawing procedure, BlitPixie8BitRectDrawProc as the offscreen draw procedure, and SWStdWorldDrawProc (i.e., CopyBits) as the screen draw procedure. For maximum speed, use CompiledSprite8BitDrawProc for Sprites, and BlitPixie8BitRectDrawProc for both the offscreen and screen draw procedure.

Assertions

SpriteWorld uses assertions in its code to help prevent crashes. Assertions are "sanity checks" that make sure all the parameters and variables each function uses are what they are assumed to be. For instance, SWDrawTile might make sure the SpriteWorldPtr passed to the function is not NULL, make sure there is a TileMap installed in the SpriteWorld, make sure that TileMap is locked, and it might also verify that all the other parameters are within their proper range.

The idea behind assertions is that if you, the programmer, forget to do something you should have done (such as locking the SpriteWorld or a Sprite before running the animation), SpriteWorld will simply report the problem and quit, rather than causing a crash.

Since it takes time to do assertions, the traditional practice is to leave them on during the development cycle of your program, and turn them off shortly before releasing your finished product. That way, the finished product will run as fast as possible. To turn off assertions, you need to open up SWCommonHeaders.h (in the SpriteWorld Files -> Headers folder) and find the SW_ASSERT section. Then simply change this definition from 1 to 0:

```
#define SW_ASSERT_ON          1
```

All assertions will be compiled out, allowing SpriteWorld to run as fast as possible. It should be noted that assertions don't slow SpriteWorld down very much (they're simply if statements added to most functions), so you could even leave them turned on in your final product if you want. It's always nicer to get a neat dialog box reporting a problem than to have the program crash.

Assertions should not be confused with errors. The error codes returned by the various SpriteWorld functions are not predictable, and may return an error under certain situations but not under others. For instance, you may have enough memory to run your game in 256 colors, but when running it in millions, there may not be enough memory to create the SpriteWorld, in which case an error code will be returned. You should always check your error codes. The difference between error codes and assertions is that assertions always indicate a serious bug in your program that needs to be fixed. If the assertion hadn't been triggered, your program would most likely have crashed. Error codes, on the other hand, usually don't indicate a bug in your code, but rather a user-controlled condition that needs changing, such as the screen depth, or a resource missing from your resource file.

When reporting assertion failures, SpriteWorld looks for ALRT and DITL resources 128. It uses these to display an alert dialog box that tells you where the assertion failure occurred. The ALRT and DITL resources needed for this may be copied from just about any of the demos that come with SpriteWorld. If they are not present in your program and an assertion failure occurs, SpriteWorld will beep and quit the program.

SpriteWorld Reference

The following section serves as a reference to the routines SpriteWorld provides. This documentation is divided into five parts; the first four correspond to the four core data structures of SpriteWorld: SpriteWorlds, SpriteLayers, Sprites, and Frames. The final part lists SpriteWorld utility functions and Sprite Compiler functions. Within each section, the routines are listed in alphabetical order.

Note:

• The routines are categorized according to which structure is **modified**. For example, SWAddFrame is considered a Sprite routine, because a Sprite is being changed by having a Frame added to it.

• There are many routines in the SpriteWorld code that aren’t documented here. For the most part, these should be considered “internal” and used with caution, if at all.

SpriteWorld Routines

SWAddSpriteLayer

This function will add a SpriteLayer to a SpriteWorld.

```
void SWAddSpriteLayer(SpriteWorldPtr spriteWorldP,
                                SpriteLayerPtr spriteLayerP);
```

spriteWorldP	A SpriteWorld to which the Layer will be added.
spriteLayerP	A SpriteLayer to add.

DESCRIPTION

The SWAddSpriteLayer function is used to add a previously created SpriteLayer to a SpriteWorld. A world can contain any number of Layers. Once a Layer is added to a world, it becomes an active part of the animation. Any Sprites in the Layer will be processed and drawn when the next Frame of the animation is rendered.

SWAnimateSpriteWorld

This function will render a frame of the animation using the frame differential technique.

```
void SWAnimateSpriteWorld(SpriteWorldPtr spriteWorldP);
```

spriteWorldP	The SpriteWorld to be animated.

DESCRIPTION

The SWAnimateSpriteWorld function is used to render a frame of the animation by drawing all the Sprites in the specified SpriteWorld in their new positions. You will typically call this function right after SWProcessSpriteWorld in your main animation loop. This function marks all the Sprites as no longer in need of drawing, so that next time around if the Sprite has not been moved or otherwise changed in any way, it will not be drawn again unnecessarily.

SEE ALSO

SWProcessSpriteWorld

SWChangeWorldRect

This function changes the size and location of a SpriteWorld's worldRect.

```
OSErr SWChangeWorldRect(SpriteWorldPtr spriteWorldP,             Rect*
                                newWorldRect,                            Boolean
                                changeOffscreenAreas)
```

spriteWorldP	The SpriteWorld.
newWorldRect	The new size and location of the worldRect
changeOffscreenAreas	A Boolean value specifying if you want to change the size of the offscreen areas to match the new worldRect size.

DESCRIPTION

This function changes the size of a SpriteWorld's worldRect (the visible area on the screen), allowing you to make it smaller or larger than it was when the SpriteWorld was first created. This can be useful for situations where you need a large worldRect in one case, such as a game map that fills the entire screen, but need a smaller worldRect in another case, such as the actual game where stats take up part of the screen, allowing less space for the main animation. This function can also be used to allow the user to resize the game's window while the animation is running. This is particularly useful for scrolling games, where the window isn't a fixed size.

It is important to note that SWChangeWorldRect can only make the worldRect as large as the offscreen areas of the SpriteWorld. So if you need to switch between several different worldRect sizes, you need to pass an offscreenRect that's the largest of those sizes to SWCreateSpriteWorldFromWindow when first creating the SpriteWorld. In addition, if you do not use SWCreateSpriteWorldFromWindow, but use a "hand-made" SpriteWorld, you should not attempt to make the worldRect larger than it was when you first created the SpriteWorld's windowFrame. Doing so could result in a crash, or other strange behaviour.

You may call SWChangeWorldRect as many times as you wish, making the worldRect any size and moving it to any location within the source window, as long as it is never made larger than the SpriteWorld's offscreen areas. If you wish to undo the effects of your calls to SWChangeWorldRect, you may call SWRestoreWorldRect, which will return the worldRect to its original size and position when the SpriteWorld was first created.

In addition to changing the SpriteWorld's windRect variable, this function also resizes the SpriteWorld's visScrollRect to match the new worldRect's width and height. And if you pass true as the changeOffscreenAreas parameter, it will also change the SpriteWorld's work and background frames so they think they are smaller than they actually are. (It does this by changing their frameRect variables.) You would generally pass true as this parameter for non-scrolling animations, so that functions such as SWUpdateSpriteWorld only need to copy the used portion when copying the back frame to the work frame, and so you only fill the used portion when drawing your background. However, if you have a scrolling animation, you should pass false, so that the entire offscreen area remains available for the animation, and to ensure that the offscreen areas' frameRects are still evenly divisible by your tiles' width and height.

After calling this function, the SpriteWorld's windRect variable will be set to the worldRect you passed to this function. In addition, the SpriteWorld has the originalWindRect and originalBackRect variables which contains the size and location of the worldRect and offscreen areas when the SpriteWorld was first created. These variables are used by SWRestoreWorldRect, but can also be used by you if you wish. For instance, you can calculate the maximum size of your window from the originalBackRect variable and pass this value to the toolbox function GrowWindow when the user resizes your window.

After calling this function, you should call SWDrawTilesInBackground (if tiling is used) and

SWUpdateSpriteWorld or SWUpdateScrollingSpriteWorld to update the newly exposed portion should the windRect be made larger. If the windRect is made smaller, these calls are not necessary, but won't hurt anything. You can determine whether the windRect was made smaller or larger by comparing the width and height of the spriteWorldP->windRect before calling SWChangeWorldRect with its width and height after calling SWChangeWorldRect. You might also want to call SWResetTilingCache before calling SWDrawTilesInBackground, if you changed the background since SpriteWorld last drew tiles in that area.

This function should be called outside of your main animation loop. That is, it should not be called from any DrawProc, MoveProc, FrameProc, etc., and should not be called between calls to SWProcessSpriteWorld and SWAnimateSpriteWorld. (Rather, it should be called before SWProcessSpriteWorld or after SWAnimateSpriteWorld.)

For an example of this function, see the source code for the Modified Scrolling Demo, contained in the Scrolling Demo folder.

SEE ALSO

SWRestoreWorldRect

SWCopyBackgroundToWorkArea

This function copies SpriteWorld’s background frame to its work frame.

```
void SWCopyBackgroundToWorkArea(SpriteWorldPtr spriteWorldP);
```

spriteWorldP	The SpriteWorld owning the background frame and work frame.

DESCRIPTION

A SpriteWorld’s background frame holds a clean copy of the background for the animation. The work frame is where the background and Sprites are mixed before copying to the screen. SWCopyBackgroundToWorkArea copies the entire background frame to the work frame. By calling SWCopyBackgroundToWorkArea and then SWUpdateWindow (which copies the work frame to the window), you can update the worldRect of the SpriteWorld without drawing any Sprites. You might want to do this for the opening display of a game, for example.

SEE ALSO

SWUpdateWindow

SWUpdateSpriteWorld

SWSetPortToBackground

SWCreateSpriteWorldFromWindow

This function will create a SpriteWorld automatically for an existing window.

```
OSErr SWCreateSpriteWorldFromWindow(
                                  SpriteWorldPtr* spriteWorldP,
                                  CWindowPtr srcWindowP,                Rect*
                                worldRect,                              Rect*
                                backRect,                               short
                                maxDepth);
```

spriteWorldP	A newly created SpriteWorld is returned in this parameter.
srcWindowP	A window in which the new SpriteWorld will exist.
worldRect	The dimensions of the SpriteWorld.
backRect	The dimensions of the offscreen areas.
maxDepth	The maximum depth of the offscreen GWorlds

DESCRIPTION

The SWCreateSpriteWorldFromWindow function is used to create a new SpriteWorld for an existing window. Given an existing window, this function will create a new SpriteWorld in the window, using the dimensions you specify in coordinates local to the window. If you specify NULL for a worldRect parameter, the full dimensions of the window will be used as the worldRect. The maximum-depth GDevice of the worldRect will used for all the GWorlds created for this SpriteWorld.

For fastest animation, you should place the left side of your worldRect (in global coordinates) on an even long word boundary. In 8-bits (256 colors), this would be every 4th pixel, starting at screen column 0. For other bit depths, you should align to every 32nd, 16th, 8th, and 2nd pixel when in 1-bit, 2-bits, 4-bits, and 16-bits respectively. If you don't use a worldRect, make sure the left side of your window is aligned to an even long word boundary.

The backRect parameter is provided so you can make the offscreen area larger than the worldRect. You will only need to do this if you are using the scrolling routines; otherwise, pass NULL to use a backRect the same size as the worldRect.

The coordinates you use for functions such as SWMoveSprite are local to the worldRect of the SpriteWorld. Thus, if worldRect is the same as the window's dimensions, coordinates for Sprites will be the same as local coordinates in that window. To conserve memory, you should define a worldRect no larger that what is needed for the animated portion of your window. SWCreateSpriteWorldFromWindow will return an error if any memory allocation fails, or if srcWindowP is a WindowPtr rather than a CWindowPtr.

The maxDepth parameter allows you to limit the depth of the offscreen GWorlds that SpriteWorld creates. If this parameter is zero, then all offscreen GWorlds will be the same depth as the “parent” window. Normally you will pass zero for maxDepth, as having the GWorlds at a different depth from the window will severely hamper the speed of SpriteWorld. However, for less-demanding animation, the option provided by maxDepth can be very useful. If the end user's monitor is set to 16 or 32 bits depth, but you only require 8 bits for your animation's images, your choices would normally be to a) require that the monitor be switched to 8 bits, or b) require a large memory allocation for your program in order to accommodate 16 or 32 bit offscreen GWorlds. The maxDepth parameter provides a third alternative. By passing a maxDepth of 8, the offscreen GWorlds will be created at 8 bits depth. If this is done, BlitPixie can still be used as the spriteDrawProc and offscreenDrawProc, but CopyBits will have to be used for the screenDrawProc. CopyBits will be transferring images between PixMaps of different depth, and this slows it down considerably. As noted, this option will only be useful for less-demanding animation.

SEE ALSO

SWSetSpriteWorldScreenDrawProc

SWDisposeSpriteWorld

This will dispose of an existing SpriteWorld and its contents, releasing the memory it occupies.

```
void SWDisposeSpriteWorld(SpriteWorldPtr *spriteWorldP);
```

spriteWorldP	A SpriteWorld to be disposed.

DESCRIPTION

The SWDisposeSpriteWorld function is used to dispose of a SpriteWorld previously created using SWCreateSpriteWorld or SWCreateSpriteWorldFromWindow. The memory occupied by the SpriteWorld, as well as the background and work frames, and all contained SpriteLayers, Sprites and Frames will be released. Note that only Layers that have been added to the SpriteWorld, and only Sprites that have been added to one of those Layers will be disposed of. Thus, any Sprites that have been removed from their Layer with SWRemoveSprite, or that have never been added to a Layer, will not be disposed of.

There is no particular need to call SWDisposeSpriteWorld if your application is about to quit; the system will free up the application’s heap space when you quit. The only caveat is that if you don’t call SWDisposeSpriteWorld, and you have called SWSyncSpriteWorldToVBL to activate VBL syncing, you **must** call SWSyncSpriteWorldToVBL again with a value of false before you quit.

NOTE: SWDisposeSpriteWorld will not dispose any TileMaps that are installed in the SpriteWorld being disposed. They are left for you to dispose using SWDisposeTileMap.

SEE ALSO

SWDisposeLayer

SWDisposeSprite

SWSyncSpriteWorldToVBL

SWEnterSpriteWorld

This function initializes the SpriteWorld package .

```
OSErr SWEnterSpriteWorld(void);
```

DESCRIPTION

The SWEnterSpriteWorld function is used to initialize the SpriteWorld package.

SWEnterSpriteWorld performs some checks to see is SpriteWorld can run, and then sets up some internal data structures. You must call SWEnterSpriteWorld before calling any other SpriteWorld routine.

SWEnterSpriteWorld returns an error code if initialization fails or if it can't run on this machine, otherwise it returns noErr.

SWExitSpriteWorld

This function shuts down the SpriteWorld package.

```
void SWExitSpriteWorld(void);
```

DESCRIPTION

The SWExitSpriteWorld function is used to shut down the SpriteWorld package. At this point no further calls should be made to any SpriteWorld routines until SWEnterSpriteWorld is called again. There is no need to call SWExitSpriteWorld if your application is about to quit, since the system will free up the application's heap space when you quit.

SWFlagRectAsChanged

This function tells SpriteWorld that a portion of the background has been changed.

```
OSErr SWFlagRectAsChanged(SpriteWorldPtr spriteWorldP,             Rect*
                                theChangedRect);
```

spriteWorldP	The SpriteWorld with the backFrame.
theChangedRect	A pointer to the rect that has been changed in the backFrame.

DESCRIPTION

Occasionally, you may want to access the backFrame of a SpriteWorld during the animation in order to change a part of the background. After changing the backFrame in such a way, you will want the change to be immediately reflected on the screen. You can do this by calling SWUpdateSpriteWorld, but this would be unnecessarily slow if only a small part of the background has been changed. SWFlagRectAsChanged will force only the rect you specify to be updated to the workFrame and the screen. The flagged rect will be updated the next time SWAnimateSpriteWorld is called. SWFlagRectAsChanged allocates a few bytes of memory, and returns an error in the unlikely event that the allocation fails.

This function is not for use with scrolling SpriteWorlds. If you change the background of a scrolling SpriteWorld, you should call SWFlagScrollingRectAsChanged.

SPECIAL CONSIDERATIONS

Note that the coordinates of theChangedRect must be local to the worldRect of the

SpriteWorld; depending on how you set up the SpriteWorld, the worldRect may not be the same as the window's portRect.

SEE ALSO

SWFlagScrollingRectAsChanged

SWSetPortToBackground

SWFlagScrollingRectAsChanged

This function tells SpriteWorld that a portion of the scrolling background has been changed.

```
OSErr SWFlagScrollingRectAsChanged(SpriteWorldPtr spriteWorldP,    Rect*
                                   theChangedRect);
```

spriteWorldP	The scrolling SpriteWorld.
theChangedRect	A pointer to the rect that has been changed in the scrolling world.

DESCRIPTION

This function is basically the same as SWFlagRectAsChanged, but for use with scrolling SpriteWorlds. During the next call to SWAnimateScrollingSpriteWorld, each changedRect will be copied from the backFrame to the workFrame. In addition, all idle sprites in each rect will be redrawn, since they will have been erased when the rect was copied.

The rectangle you pass can be anywhere in your "virtual world". However, due to SpriteWorld's scrolling technique, you may get unexpected results if you pass a rectangle that is not within the bounds of the current visScrollRect.

Generally you won't have a need to use this function, since any changes you make to a scrolling background by modifying it directly aren't permanent (since they are erased when you scroll away and back), so you will generally need to find a different way of changing the background in the first place (such as by using different tiles). However, if you need to modify it directly for some special effect (as the Wrapping Illustration does), you can use this function to "clean up" after yourself.

SEE ALSO

SWFlagRectAsChanged

SWGetNextSpriteLayer

This function will return the next SpriteLayer from a SpriteWorld.

```
SpriteLayerPtr SWGetNextSpriteLayer(SpriteWorldPtr spriteWorldP,
                                    SpriteLayerPtr curSpriteLayerP);
```

spriteWorldP	A SpriteWorld from which to get the SpriteLayer.
curSpriteLayerP	A SpriteLayer previously returned from this function, pass NULL to get the first SpriteLayer.

DESCRIPTION

The SWGetNextSpriteLayer function is used to iterate through the Layers in world. The Layer following the current one you specify is returned. When the are no more Layers in the world, NULL is returned.

SWGetSpriteWorldVersion

This function returns the version number of SpriteWorld.

```
unsigned long SWGetSpriteWorldVersion(void);
```

DESCRIPTION

SWGetSpriteWorldVersion returns a an unsigned long value in binary coded decimal indicating the current version number of SpriteWorld. The high word indicates the major version number of SpriteWorld, and the low word indicates any beta number attached to the version. For example, 0x02010000 = SpriteWorld version 2.0.1.

If a beta number is returned, then that version of SpriteWorld should not be considered intended for general release. Visit the SpriteWorld web page for the latest version.

SWLockSpriteWorld

This function locks a SpriteWorld in preparation for animation.

```
void SWLockSpriteWorld(SpriteWorldPtr spriteWorldP);
```

spriteWorldP	A SpriteWorld to be locked.

DESCRIPTION

The SWLockSpriteWorld function is used to lock the SpriteWorld including all the SpriteLayers, Sprites, and Frames contained within. Before a SpriteWorld can be animated, and before anything is drawn to the background frame, the SpriteWorld must first be locked.

SPECIAL CONSIDERATIONS

Note that if you add a new Sprite or Frame to a running animation, it must be locked before it is animated. Any Frames, Sprites or Layers that were not attached to the SpriteWorld when SWLockSpriteWorld is called will not be locked. Thus, as a rule, calling

SWLockSpriteWorld should be done only after assembling all the elements of your animation with SWAddSprite and SWAddSpriteLayer.

∆ WARNING ∆

Drawing a picture or pattern to the background frame must be done **after** the SpriteWorld is locked.

SEE ALSO

SWLockSpriteLayer

SWLockSprite

SWProcessSpriteWorld

This function processes all the Sprites in a SpriteWorld, updating their positions, resetting their timers, calling their custom move and Frame procs, etc.

```
void SWProcessSpriteWorld(SpriteWorldPtr spriteWorldP);
```

spriteWorldP	A SpriteWorld to be processed.

DESCRIPTION

The SWProcessSpriteWorld function is used to perform all the automatic processing to every Sprite in the SpriteWorld. This includes updating the Sprite’s positions, resetting their movement and Frame change timers, and calling their custom move and Frame routines if any. This function, in conjunction with SWAnimateSpriteWorld, drives the animation.

This function does no drawing, it only processes all Sprites in terms of their movement and Frame changing characteristics using the parameters you specify when setting up the Sprite.

SEE ALSO

SWAnimateSpriteWorld

SWRemoveSpriteLayer

This function will remove a SpriteLayer from a SpriteWorld.

```
void SWRemoveSpriteLayer(SpriteWorldPtr spriteWorldP,
                            SpriteLayerPtr spriteLayerP);
```

spriteWorldP	A SpriteWorld from which the SpriteLayer is to be removed.
spriteLayerP	A SpriteLayer to remove.

DESCRIPTION

The SWRemoveSpriteLayer function is used to remove a SpriteLayer from a SpriteWorld. This is done when you want to remove an entire Layer of Sprites from the animation. The Sprites in the Layer that is removed will not be processed or drawn when the next frame of the animation is rendered.

SPECIAL CONSIDERATIONS

Removing the Layer will not erase the Sprites where they are on the screen. If you wish the Sprites in the Layer to disappear and the animation to continue, you must first set each Sprite’s visibility to FALSE, render a frame of the animation, and then remove the Layer from the world. Alternatively, you could call SWUpdateSpriteWorld after removing the Layer.

SEE ALSO

SWSetSpriteVisible

SWRestoreWorldRect

This function restores the SpriteWorld's worldRect to its original size and location when the SpriteWorld was first created.

```
OSErr SWRestoreWorldRect(SpriteWorldPtr spriteWorldP)
```

spriteWorldP	The SpriteWorld whose worldRect you wish to restore.

DESCRIPTION

This function undoes the effects of any calls to SWChangeWorldRect by restoring the SpriteWorld's worldRect to its original size and location when the SpriteWorld was first created. It also restores the offscreen areas to their original size. After calling this function, you should call SWDrawTilesInBackground (if tiling is used), and SWUpdateSpriteWorld if your game is a non-scrolling game, or SWUpdateScrollingSpriteWorld if your game is a scrolling game, before resuming the animation.

If calling SWUpdateScrollingSpriteWorld, you may pass false as the updateScreen parameter, since the scrolling routines automatically update the screen each frame anyway. You may also wish to call SWResetTilingCache before calling SWDrawTilesInBackground, if you think the newly exposed portion of the background may have been changed since SpriteWorld last drew tiles in that area.

SWSetPortToBackground

This function will set the current port to the GWorld of SpriteWorld’s background frame.

```
void SWSetPortToBackground(SpriteWorldPtr spriteWorldP);
```

spriteWorldP	The SpriteWorld owning the background frame.

DESCRIPTION

SWSetPortToBackground sets the current port to the background frame. You will want to do this to draw the background image for SpriteWorld’s use. You can easily get the rect of the background frame with:

```
spriteWorldP->backRect
```

SPECIAL CONSIDERATIONS

SWSetPortToBackground will only perform its function if the SpriteWorld has been locked. The GWorld of the background frame will not be locked if the SpriteWorld is not locked.

SWSetPortToWindow

This function will set the current port to the “parent” window of the SpriteWorld.

```
void SWSetPortToWindow(SpriteWorldPtr spriteWorldP);
```

spriteWorldP	The SpriteWorld using the window.

DESCRIPTION

SWSetPortToWindow sets current port to the window the SpriteWorld is using. You will want to use this if you are doing any drawing to the window that isn’t handled by SpriteWorld.

SWSetPortToWorkArea

This function will set the current port to the GWorld of SpriteWorld’s work frame.

```
void SWSetPortToWorkArea(SpriteWorldPtr spriteWorldP);
```

spriteWorldP	The SpriteWorld owning the work frame.

DESCRIPTION

SWSetPortToWorkArea sets the current port to the offscreen work frame. The work frame is where the background and Sprites are mixed before copying to the screen. Normally you won't need to call SWSetPortToWorkArea, since SpriteWorld does all the necessary drawing in the work area for you, but if you need to do additional drawing in this area for some reason, then this function can be used.

SPECIAL CONSIDERATIONS

SWSetPortToWorkArea will only perform its function if the SpriteWorld has been locked. The GWorld of the work frame will not be locked if the SpriteWorld is not locked.

SWSetPostEraseCallBack

This function sets a CallBack function to be called after erasing the Sprites offscreen each frame.

```
OSErr SWSetPostEraseCallBack(SpriteWorldPtr spriteWorldP,
                                   CallBackPtr callBack)
```

spriteWorldP	The SpriteWorld that will call the CallBack.
callBack	The CallBack to be installed in the SpriteWorld.

DESCRIPTION

This function allows you to specify a CallBack function that is to be called by SWUpdateSpriteWorld and SWAnimateSpriteWorld (or SWUpdateScrollingSpriteWorld and SWAnimateScrollingSpriteWorld) immediately after erasing the Sprites from their old position offscreen, but before drawing them in their new position. This allows you to perform special effects by drawing "underneath" the Sprites each frame.

Keep in mind, however, that when you draw something into the work area, you may end up drawing on top of idle Sprites. This shouldn't be a problem in a non-scrolling animation, since the idle Sprites are automatically redrawn whenever they are copied to the screen (which will happen if another Sprite overlaps them), although it will matter in a scrolling animation, since the entire screen is updated every frame.

When your CallBack is called, the port is already set to the work area, so there is no need for you to call SWSetPortToWorkArea before drawing offscreen. Your CallBack should be defined like this:

SW_FUNC void MyCallBack(SpriteWorldPtr spriteWorldP);

To "remove" a CallBack so that it is no longer called, simply call SWSetPostEraseCallBack again, passing it a value of NULL as the CallBackPtr. If you don't want SWUpdateSpriteWorld or SWUpdateScrollingSpriteWorld to call your CallBack, simply don't install your CallBack until after calling that function, or you can remove the CallBack and then install it again afterwards.

SEE ALSO

SWSetPostDrawCallBack

SWSetPostDrawCallBack

This function sets a CallBack function to be called after drawing the Sprites offscreen each frame.

```
OSErr SWSetPostDrawCallBack(SpriteWorldPtr spriteWorldP,
                                        CallBackPtr callBack)
```

spriteWorldP	The SpriteWorld that will call the CallBack.
callBack	The CallBack to be installed in the SpriteWorld.

DESCRIPTION

This function allows you to specify a CallBack function that is to be called by SWUpdateSpriteWorld and SWAnimateSpriteWorld (or SWUpdateScrollingSpriteWorld and SWAnimateScrollingSpriteWorld) immediately after drawing the Sprites offscreen, but before updating the screen. This allows you to perform special effects by drawing something offscreen before the screen gets updated. For instance, the Wrapping Illustration uses a postDrawCallBack to draw its color-coded rects offscreen before the screen gets updated. This avoids the flicker that would be caused if the rects were drawn directly to the screen each frame.

When your CallBack is called, the port is already set to the work area, so there is no need for you to call SWSetPortToWorkArea before drawing offscreen. Your CallBack should be defined like this:

SW_FUNC void MyCallBack(SpriteWorldPtr spriteWorldP);

To "remove" a CallBack so that it is no longer called, simply call SWSetPostDrawCallBack again, passing it a value of NULL as the CallBackPtr. If you don't want SWUpdateSpriteWorld or SWUpdateScrollingSpriteWorld to call your CallBack, simply don't install your CallBack until after calling that function, or you can remove the CallBack and then install it again afterwards.

SEE ALSO

SWSetPostEraseCallBack

SWSetSpriteWorldMaxFPS

This function controls how often the SpriteWorld is processed and its animation rendered on the screen.

```
void SWSetSpriteWorldMaxFPS(SpriteWorldPtr spriteWorldP,           short
                                framesPerSec);
```

spriteWorldP	A SpriteWorld.
framesPerSec	The rate at which the animation will be processed and rendered. A framesPerSec value of zero or less tells SpriteWorld to place no limit on the frame rate, allowing the animation to run at maximum speed.

DESCRIPTION

Normally, you will want to limit how fast your animation runs, so that it won't go too fast on faster Macs. You could do this using SWSetSpriteMoveTime and SWSetSpriteFrameTime for each Sprite you create, but SWSetSpriteWorldMaxFPS provides an alternate approach.

SWSetSpriteWorldMaxFPS controls how often SWProcessSpriteWorld actually scans through the Sprites in the SpriteWorld looking for Sprites that are due for updating. One advantage of this over setting each Sprite's moveTime and frameTime separately is that SpriteWorld will spend less time scanning through its list of Layers and Sprites. This will leave more CPU time available for other tasks.

SpriteWorld keeps track of time in milliseconds, so framesPerSec values between 1 and 1000 are meaningful, if not necessarily achievable on your hardware. Values for framesPerSec of zero or less are special values for SWSetSpriteWorldMaxFPS, and tell SpriteWorld to place no limit on the frame rate, so that the animation will run at the fastest speed possible. By default, SpriteWorld places no limit on animation speed.

SPECIAL CONSIDERATIONS

SWSetSpriteMoveTime and SWSetSpriteFrameTime are codependent upon SWSetSpriteWorldMaxFPS. A Sprite can't be moved, or its Frame changed, any more frequently than the SWSetSpriteWorldMaxFPS setting allows, regardless of the SWSetSpriteMoveTime or SWSetSpriteFrameTime settings. As a rule, the best way to control the speed of Sprites is to set a “base” speed with SWSetSpriteWorldMaxFPS. Then, if you want a particular Sprite to move more slowly than the base speed use SWSetSpriteMoveTime. To make a Sprite move faster, increase the distance it travels each time it is moved.

SWSyncSpriteWorldToVBL will also affect the frame rate if it is turned on; in that case the maximum frame rate will be equal to the refresh rate of the monitor.

Keep in mind that regardless of how quickly and frequently SWProcessSpriteWorld processes the SpriteWorld and advances the positions (and/or Frames) of Sprites, the changes can't actually **appear** on the screen any faster than the refresh rate of the monitor. This is typically 60-75 Hz, depending on the model of monitor.

Naturally, there is no guarantee that any given Mac will be able to run your animation at the framesPerSec value you specify. SWSetSpriteWorldMaxFPS only sets an upper limit to the animation speed.

To make sure that your animation doesn't run too fast on hardware faster than yours, you can follow this procedure: Once you have the animation looking the way you want it on your machine, start running it with successively lower framesPerSec values. If the animation starts running slower, you will know that the frame rate is being controlled by SpriteWorld, rather than limited by the speed of your Mac.

When SWSetSpriteWorldMaxFPS is used, SpriteWorld sets a flag:
spriteWorldP->frameHasOccurred to true whenever SWProcessSpriteWorld is called **and** it updates the positions of the Sprites. If SWProcessSpriteWorld is called and it isn't time to update the positions of the Sprites (i.e., it isn't time for a frame), then SWProcessSpriteWorld will set the flag to false and exit. If you want some action performed once and only once for each frame of animation, you can use this flag as follows:

```
SWProcessSpriteWorld(mySpriteWorldP);

SWAnimateSpriteWorld(mySpriteWorldP);

if (mySpriteWorldP->frameHasOccurred)

{
        UpdateScore();
    }
```

SEE ALSO

SWSetSpriteMoveTime

SWSetSpriteFrameTime

SWProcessSpriteWorld

SWSyncSpriteWorldToVBL

SWSetSpriteWorldOffscreenDrawProc

This function will set a SpriteWorld's offscreen drawing routine to the one you specify.

```
OSErr SWSetSpriteWorldOffscreenDrawProc(
                            SpriteWorldPtr spriteWorldP,
                            WorldDrawProcPtr offscreenProc);
```

spriteWorldP	A SpriteWorld whose offscreen drawing routine will be set.
offscreenProc	A new offscreen drawing routine.

DESCRIPTION

The SWSetSpriteWorldOffscreenDrawProc function is used to specify a new offscreen drawing routine for the given SpriteWorld. This routine is used to erase a Sprite in the offscreen work area by copying the appropriate rectangle from the background frame. The default offscreen drawing routine calls CopyBits.

The options for offscreenProc currently available are:

`SWStdWorldDrawProc`	CopyBits
BlitPixieAllBitRectDrawProc	
BlitPixie8BitRectDrawProc	BlitPixie
BPAllBitInterlacedRectDrawProc	
BP8BitInterlacedRectDrawProc	Interlaced BlitPixie

An error code is returned if a BlitPixie drawProc is specified and the depth of the offscreen work area is not correct for the drawProc.

SEE ALSO

SWSetSpriteWorldScreenDrawProc

SWSetSpriteWorldScreenDrawProc

This function will set a SpriteWorld's screen drawing routine to the one you specify.

```
OSErr SWSetSpriteWorldScreenDrawProc(
                                SpriteWorldPtr spriteWorldP,
                                DrawProcPtr screenDrawProc);
```

spriteWorldP	A SpriteWorld whose screen drawing routine will be set.
screenDrawProc	A new screen drawing routine.

DESCRIPTION

The SWSetSpriteWorldScreenDrawProc function is used to specify a new screen drawing routine for the given SpriteWorld. This routine is used to put the updated Sprite image onto the screen. The standard screen draw routine calls CopyBits.

The options for screenDrawProc currently available are:

SWStdWorldDrawProc	CopyBits
BlitPixieAllBitRectDrawProc	
BlitPixie8BitRectDrawProc	BlitPixie
BPAllBitInterlacedRectDrawProc	
BP8BitInterlacedRectDrawProc	Interlaced BlitPixie

An error code is returned if a BlitPixie drawProc is specified and the depth of the monitor is not correct for the drawProc, or if a BlitPixie drawProc is specified and the depth of the offscreen work frames does not match the depth of the monitor (they may not match if the maxDepth parameter was non-zero when SWCreateSpriteWorldFromWindow was called).

SEE ALSO

SWSetSpriteWorldOffscreenDrawProc

SWCreateSpriteWorldFromWindow

SWSwapSpriteLayer

This function will swap two SpriteLayers in a SpriteWorld.

```
void SWSwapSpriteLayer(SpriteWorldPtr spriteWorldP,
                                SpriteLayerPtr srcSpriteLayerP,
                                SpriteLayerPtr dstSpriteLayerP);
```

spriteWorldP	A SpriteWorld.
srcSpriteLayerP	The Layers to swap.
dstSpriteLayerP	

DESCRIPTION

SWSwapSpriteLayer swaps the position of two SpriteLayers, so that the one whose Sprites were formerly drawn behind the other’s is now in front.

SWSyncSpriteWorldToVBL

This function causes SpriteWorld to wait for the monitor’s vertical retrace before drawing to the screen.

```
OSErr SWSyncSpriteWorldToVBL(SpriteWorldPtr spriteWorldP,          Boolean
                             vblSyncOn);
```

spriteWorldP	A SpriteWorld.
vblSyncOn	If true, a VBL task interrupt is installed, and SpriteWorld will wait for the vertical retrace before drawing to the screen. If false, and a VBL task has previously been installed, it is removed.

DESCRIPTION

In some cases, you can achieve better looking animation by synchronizing the screen-drawing to the monitor’s vertical retrace. When Sprites have a lot of horizontal motion, they may appear “torn”—divided horizontally into an upper and lower section that are offset from one another. This is caused by the monitor’s refresh occurring midway through the drawing of the Sprite, so the Sprite’s old position is mixed with the new one.

The cure for “tearing” is to synchronize with the monitor’s refresh, and this is done by installing a vertical retrace interrupt task. The vertical retrace, or VBL (for Vertical BLanking) task is called when the monitor’s electron gun jumps from the lower right to the upper left of the screen to begin another refresh. If animation is drawn to the screen quickly after this occurs, it will be finished before the refresh reaches the Sprites’ positions.

If SWSyncSpriteWorldToVBL is called with the vblSyncOn parameter set to true, a VBL task will be installed, and SWAnimateSpriteWorld will be set to wait for this task to “fire” before it draws the Sprites to the screen. SWSyncSpriteWorldToVBL only needs to be called once, while you are setting up your SpriteWorld. By default, SpriteWorld does not perform VBL synchronization. An error is returned by SWSyncSpriteWorldToVBL if the SlotVInstall or SlotVRemove calls fail.

SPECIAL CONSIDERATIONS

As noted, VBL synchronizing can improve the appearance of some animations, but there are a number of special considerations that must be taken into account:

Once a VBL task has been installed, it must be removed before the application quits, or the system will crash. SpriteWorld will remove the VBL task when you call SWDisposeSpriteWorld, and you can remove it yourself at any time by calling SWSyncSpriteWorldToVBL with a vblSyncOn parameter of false. The problem and danger lies with abnormal exits from your application, such as command-option-escape forced quits. For this reason, you may want to leave out VBL synchronizing for most of the development period of your application. More advanced programmers may want to patch the ExitToShell trap so that it calls SWSyncSpriteWorldToVBL.

When an animation is pushing the limits of a Mac (i.e., with slower Macs), VBL synchronizing may provide little or no improvement, since the screen drawing will be too slow to stay ahead of electron gun. By the same token, you may find that with VBL synchronizing on, you still get a tearing effect on Sprites when they are near the top of the screen (where they have the least time to finish drawing before the electron gun overtakes them).

When VBL synchronizing is active, SpriteWorld will spend a certain percentage of time in an empty loop, waiting for the VBL task to fire. This may be a liability if there are other CPU-intensive tasks that need attention in your main loop.

With VBL synchronizing active, SWProcessSpriteWorld will not be called any more frequently than the refresh rate of the monitor, regardless of any SWSetSpriteWorldMaxFPS, SWSetSpriteMoveTime or SWSetSpriteFrameTime settings. Monitors generally have a refresh rate between 60 and 75 cycles per second.

Because the VBL task is specific to a particular monitor, you should not allow an animation’s window to be dragged to a new monitor on a multi-monitor system.

SEE ALSO

SWProcessSpriteWorld

SWSetSpriteWorldMaxFPS

SWUnlockSpriteWorld

This function will unlock a SpriteWorld.

```
void SWUnlockSpriteWorld(SpriteWorldPtr spriteWorldP);
```

spriteWorldP	A SpriteWorld to be unlocked.

DESCRIPTION

The SWUnlockSpriteWorld function is used to unlock the SpriteWorld, including all the SpriteLayers, Sprites, and Frames contained within. After animation has completed you may want to unlock the SpriteWorld. This SpriteWorld must not be used for animation while in this unlocked state. If you wish to animate the SpriteWorld again, you must first call SWLockSpriteWorld.

SWUpdateSpriteWorld

The function will draw the current frame of the animation, completely redrawing the on-screen worldRect.

```
void SWUpdateSpriteWorld(SpriteWorldPtr spriteWorldP,              Boolean
                                updateWindow);
```

spriteWorldP	A SpriteWorld to be updated.
updateWindow	If true, the updated SpriteWorld will be drawn to the screen; if false, the SpriteWorld will be updated to the offscreen workFrame, but not

to the window.

DESCRIPTION

The SWUpdateSpriteWorld function is used to build the current frame in the offscreen workFrame, and copy the entire frame to the window. You will typically call this function when the window in which your animation is running receives an update event, or before starting the animation, or whenever the background image has been changed. If you have a huge number of active sprites, nearly filling the worldRect, you may find that you get better animation by calling SWUpdateSpriteWorld in place of SWAnimateSpriteWorld.

The updateWindow parameter allows you to control whether the updated SpriteWorld is drawn to the screen. By setting this to false, you can perform the actual updating of the window yourself (by copying spriteWorldP->workFrameP to spriteWorldP->windowFrameP), perhaps with a special effect, such as a screen-wipe.

SWUpdateWindow

This function copies SpriteWorld’s work frame to its window frame.

```
void SWUpdateWindow(SpriteWorldPtr spriteWorldP);
```

spriteWorldP	The SpriteWorld owning the work frame and window frame.

DESCRIPTION

A SpriteWorld’s work frame is where the background and Sprites are mixed before copying to the screen. Normally in animation, only those parts of the work frame that contain changed or moved Sprites are copied to the window. `SWUpdateWindow` copies the entire work frame to the window. However, unlike `SWUpdateSpriteWorld`, `SWUpdateWindow` does not process a frame of animation beforehand. Thus it can serve as a quick way to respond to an update event.

SEE ALSO

SWUpdateSpriteWorld

SWWindowMoved

This function updates the data used by SpriteWorld’s direct-to-screen blitter.

```
void SWWindowMoved(SpriteWorldPtr spriteWorldP);
```

spriteWorldP	The SpriteWorld owning the window frame.

DESCRIPTION

If one of the BlitPixieRectDraw routines is being used as a SpriteWorld’s screenDrawProc, then certain information SpriteWorld uses for this drawProc must be updated if the window is moved. By calling SWWindowMoved immediately after responding to an inDrag mouse click with DragWindow, you can use direct-to-screen blitting in a movable window. If you fail to call SWWindowMoved, the window's contents will be drawn to the window's old position.

If the window is dragged off the side of the screen so that part of the window's boundaries extend past the screen's boundaries, SWWindowMoved automatically clips the window's frameRect, which the blitters use when drawing to the screen. This means that everything will still work fine even if the window is dragged part way (or even all the way) off the screen. (See the Large Background Scrolling demo in the SpriteWorld Demos package for more information.)

SPECIAL CONSIDERATIONS

If a window is being used with a direct-to-screen blitter and it has a title bar, you should take into account that the window might be collapsed by the user with WindowShade. This can be done by checking (*theWindow->visRgn)->rgnBBox in the event loop.

SEE ALSO

SWWindowFrameMoved

Layer Routines

SWAddSprite

This function will add an existing Sprite to a SpriteLayer.

```
OSErr SWAddSprite(SpriteLayerPtr spriteLayerP,
                  SpritePtr newSpriteP);
```

spriteLayerP	An existing SpriteLayer.
newSpriteP	A Sprite to be added to the specified SpriteLayer.

DESCRIPTION

The SWAddSprite function is used to add an existing Sprite to the end of a SpriteLayer, causing it to be drawn above any Sprites previously added to that Layer. An error code is returned if the Sprite is already part of a SpriteLayer. (It must be removed before it can be added to another Layer.)

SEE ALSO

SWInsertSpriteBeforeSprite

SWInsertSpriteAfterSprite

SWCollideSpriteLayer

This function will check for collisions between two SpriteLayers.

```
void SWCollideSpriteLayer(SpriteWorldPtr spriteWorldP,
                          SpriteLayerPtr srcSpriteLayerP,
                          SpriteLayerPtr dstSpriteLayerP);
```

spriteWorldP	The SpriteWorld containing the SpriteLayers.
srcSpriteLayerP	A SpriteLayer containing one or more Sprites.
dstSpriteLayerP	Another SpriteLayer containing one or more Sprites.

DESCRIPTION

The SWCollideSpriteLayer function is used to check the Sprites in the source SpriteLayer against the Sprites in the destination SpriteLayer for collisions. In order to check for collisions between the Sprites of a single SpriteLayer, you must pass the same SpriteLayer as the source and the destination. SWCollideSpriteLayer will generally be called in the event loop of your program, after SWProcessSpriteWorld and SWAnimateSpriteWorld.

When a collision is detected the collision routine, if any, of the **source** Sprite is called. For anything useful to happen you must install a collision routine in the Sprites you expect to be involved in collisions.

A collision is defined as the condition that occurs when the rectangle that defines the current screen location of a Sprite intersects the corresponding rectangle of another Sprite. This may or may not mean that the actual images of the Sprites as they appear on the screen overlap. Therefore it is up to the collision routine you provide to determine a more precise definition of a collision for your Sprites if necessary. SWRadiusCollision, SWPixelCollision and SWRegionCollision can help with this.

If you have set an overall frame rate for the SpriteWorld with SWSetSpriteWorldMaxFPS, then SWCollideSpriteLayer will only check for collisions if SWProcessSpriteWorld has actually updated the Sprites' positions. That is, even if you call SWCollideSpriteLayer in the event loop of your program, actual collision checking will only happen once per frame of animation. This can be helpful in preventing the same collision from being detected multiple times in spite of the fact that your collision routine sets up a condition that should "turn off" the collision (such as removing one of the colliding Sprites with SWRemoveSpriteFromAnimation).

SEE ALSO

SWSetSpriteCollideProc

SWRadiusCollision

SWRegionCollision

SWPixelCollision

SWSetSpriteWorldMaxFPS

SWCountNumSpritesInLayer

This function will return the number of Sprites in a Layer.

```
short SWCountNumSpritesInLayer(SpriteLayerPtr srcSpriteLayerP);
```

srcSpriteLayerP	An existing SpriteLayer.

DESCRIPTION

This function will count the number of Sprites in a Layer and return that number as a short.

SWCreateSpriteLayer

This function will create a new SpriteLayer.

```
OSErr SWCreateSpriteLayer(SpriteLayerPtr *spriteLayerP);
```

spriteLayerP	A newly created SpriteLayer.

DESCRIPTION

The SWCreateSpriteLayer function is used to create a new SpriteLayer. This function allocates memory for a new SpriteLayer and returns a pointer to the new Layer in the spriteLayerP parameter.

SWDisposeAllSpritesInLayer

This function will remove all Sprites from an existing SpriteLayer.

```
void SWDisposeAllSpritesInLayer(SpriteLayerPtr srcSpriteLayerP);
```

srcSpriteLayerP	An existing SpriteLayer.

DESCRIPTION

This function will remove and dispose all Sprites from an existing SpriteLayer. If you need the Sprites erased from the animation before they are removed, use SWRemoveSpriteFromAnimation.

SEE ALSO

SWRemoveAllSpritesFromLayer

SWDisposeSpriteLayer

This function will dispose of an existing SpriteLayer, releasing the memory it occupies.

```
void SWDisposeSpriteLayer(SpriteLayerPtr *spriteLayerP);
```

spriteLayerP	A SpriteLayer to be disposed.

DESCRIPTION

The SWDisposeSpriteLayer function is used to dispose of a SpriteLayer previously created using SWCreateSpriteLayer. This function releases the memory occupied by the SpriteLayer.

∆ WARNING ∆

If you wish the animation in which your SpriteLayer is taking part to continue, you must first remove the SpriteLayer from the SpriteWorld by calling SWRemoveSpriteLayer. Disposing of a SpriteLayer that is part of an active animation will most likely result in a crash when the next frame of the animation is rendered.

SWFindSpriteByPoint

This function returns the SpritePtr of the Sprite (if any) whose Frame rectangle contains the supplied point.

```
SpritePtr SWFindSpriteByPoint(SpriteLayerPtr spriteLayerP,
                                  SpritePtr startSpriteP,           Point
                                testPoint);
```

spriteLayerP	The SpriteLayer to search.
startSpriteP	The Sprite from which to start the reverse-order search. NULL to search from the last Sprite.
testPoint	The point to test.

DESCRIPTION

The SWFindSpriteByPoint function searches the given SpriteLayer for the Sprite last-most in the SpriteLayer (and thus top-most in drawing order) which contains testPoint. NULL is returned if no Sprite contains testPoint.

SPECIAL CONSIDERATIONS

Note that the coordinates of the Sprite are local to the worldRect of the SpriteWorld; you will have to take this into account if the worldRect is different from the window’s portRect, and testPoint is in coordinates local to the window.

SWGetNextSprite

This function will return the next Sprite from a SpriteLayer .

```
SpritePtr SWGetNextSprite(SpriteLayerPtr spriteLayerP,
                            SpritePtr curSpriteP);
```

spriteLayerP	An existing SpriteLayer.
curSpriteP	A Sprite previously returned from this function, pass NULL to get the first Sprite.

DESCRIPTION

The SWGetNextSprite function is used to iterate through the Sprites in a SpriteLayer. The Sprite following the current one you specify is returned. When there are no more Sprites in the SpriteLayer, NULL is returned.

SWInsertSpriteAfterSprite

This functions inserts a new Sprite in a Layer immediately after an existing Sprite, causing the new Sprite to be drawn above the existing Sprite should the Sprites overlap during the animation.

```
OSErr SWInsertSpriteAfterSprite(SpritePtr newSpriteP,
                                  SpritePtr dstSpriteP);
```

newSpriteP	The Sprite being inserted into the Layer.
dstSpriteP	The previously-existing Sprite.

DESCRIPTION

SWInsertSpriteAfterSprite can be used to add a Sprite to a Layer at a specific point in the Layer’s list of Sprites. Specifically, this routine adds the new Sprite directly after a given Sprite that is already in the Layer. The order of Sprites in a Layer can be important, because it determines the order in which the Sprites are drawn, and this in turn determines which Sprite will be drawn on top of the other when two Sprites overlap. The first Sprite in a Layer is drawn first, and appears bottom-most in an overlap situation, and the last Sprite is top-most. If “spriteA” is currently being drawn underneath “spriteB”, and you want it to be drawn on top, you can do this:

SWRemoveSprite(spriteA);

SWInsertSpriteAfterSprite(spriteA, spriteB);

An error code is returned if the newSpriteP is already part of a SpriteLayer when you call this function. (You can only add Sprites that are not yet part of a SpriteLayer.)

SEE ALSO

SWInsertSpriteBeforeSprite

SWInsertSpriteBeforeSprite

This functions inserts a new Sprite in a Layer immediately before an existing Sprite, causing the new Sprite to be drawn below the existing Sprite should the Sprites overlap during the animation.

```
OSErr SWInsertSpriteBeforeSprite(SpritePtr newSpriteP,
                                 SpritePtr dstSpriteP);
```

newSpriteP	The Sprite being inserted into the Layer.
dstSpriteP	The previously-existing Sprite.

DESCRIPTION

SWInsertSpriteBeforeSprite can be used to add a Sprite to a Layer at a specific point in the Layer's list of Sprites. Specifically, this routine adds the new Sprite directly before a given Sprite that is already in the Layer. The order of Sprites in a Layer can be important, because it determines the order in which the Sprites are drawn, and this in turn determines which Sprite will be drawn on top of the other when two Sprites overlap. The first Sprite in a Layer is drawn first, and appears bottom-most in an overlap situation, and the last Sprite is top-most. If "spriteA" is currently being drawn over "spriteB", and you want it to be drawn underneath, you can do this:

SWRemoveSprite(spriteA);

SWInsertSpriteBeforeSprite(spriteA, spriteB);

An error code is returned if the newSpriteP is already part of a SpriteLayer when you call this function. (You can only add Sprites that are not yet part of a SpriteLayer.)

SEE ALSO

SWInsertSpriteAfterSprite

SWLockSpriteLayer

This functions locks a SpriteLayer in preparation for animation.

```
void SWLockSpriteLayer(SpriteLayerPtr spriteLayerP);
```

spriteLayerP	A SpriteLayer to be locked.

DESCRIPTION

The SWLockSpriteLayer function is used to lock a SpriteLayer including all the Sprites and Frames contained within. Before a SpriteLayer can be animated it must first be locked.

SPECIAL CONSIDERATIONS

You must be careful when adding a Layer to an animation that is already running. The new Layer must be locked before it can be animated.

SWPauseSpriteLayer

This function will pause a SpriteLayer so it is not processed by SWProcessSpriteWorld.

```
OSErr SWPauseSpriteLayer(SpriteLayerPtr spriteLayerP);
```

spriteLayerP	The SpriteLayer to be paused.

DESCRIPTION

Call this function if you want to pause a SpriteLayer so that the Sprites in that Layer are not moved and their Frames are not changed by SWProcessSpriteWorld. The Sprites in that layer will still be visible; they simply won't be moved. Call SWUnpauseSpriteLayer to undo the effect.

SWRemoveAllSpritesFromLayer

This function will remove all Sprites from an existing SpriteLayer.

```
void SWRemoveAllSpritesFromLayer(SpriteLayerPtr srcSpriteLayerP);
```

srcSpriteLayerP	An existing SpriteLayer.

DESCRIPTION

This function will remove all Sprites from an existing SpriteLayer. If you need the Sprites erased from the animation before they are removed, use SWRemoveSpriteFromAnimation.

SEE ALSO

SWDisposeAllSpritesInLayer

SWRemoveSprite

This function will remove a Sprite from its Layer.

```
OSErr SWRemoveSprite(SpritePtr oldSpriteP);
```

oldSpriteP	The Sprite you want removed from the Layer it is currently in.

DESCRIPTION

The SWRemoveSprite function is used to remove a Sprite from the SpriteLayer it is currently in. Typically, you would use SWRemoveSpriteFromAnimation if you wish to remove a Sprite from the animation once you are done with it. However, if you want to move a Sprite from one layer to another, you should call SWRemoveSprite, followed by a call to SWAddSprite, SWInsertSpriteBeforeSprite, or SWInsertSpriteAfterSprite, to add it to the new layer.

An error code will be returned if the Sprite is not currently a part of any Layer.

SPECIAL CONSIDERATIONS

Removing a Sprite from a Layer will not erase the Sprite where it is on the screen. If you want the Sprite to disappear and the animation to continue, you must first set the Sprite's visibility to FALSE, render a frame of the animation, and then remove the Sprite from the Layer. SWRemoveSpriteFromAnimation provides an easier way to remove Sprites from a running animation.

As a rule, SWRemoveSprite should not be used from within a moveProc, frameProc, or collisionProc, except to remove the **source** Sprite of that routine, or a Sprite from another Layer. You should not remove Sprites other than the source Sprite that are in the same layer as the source Sprite. The same goes for the collisionProc; if both Sprites in collision are from the same Layer, only one may be removed by your collideProc.

SEE ALSO

SWRemoveSpriteFromAnimation

SWSwapSprite

This function will swap two Sprites in a SpriteLayer.

```
OSErr SWSwapSprite(SpritePtr srcSpriteP,
                            SpritePtr dstSpriteP)
```

srcSpriteP	The Sprites to swap.
dstSpriteP	

DESCRIPTION

SWSwapSprite swaps the position of two Sprites in a Layer, so that the Sprite which was formerly drawn behind the other is now in front. An error code is returned if both Sprites are not in the same Layer.

SWUnlockSpriteLayer

This function will unlock a SpriteLayer .

```
void SWUnlockSpriteLayer(SpriteLayerPtr spriteLayerP);
```

spriteLayerP	A SpriteLayer to be unlocked.

DESCRIPTION

The SWUnlockSpriteLayer function is used to unlock a SpriteLayer , including all the Sprites and Frames contained within. This SpriteLayer must not be used for animation while in this unlocked state. If you wish to animate the SpriteLayer again, you must first call SWLockSpriteLayer.

SWUnpauseSpriteLayer

This function will unpause a SpriteLayer so it is once again processed by SWProcessSpriteWorld.

```
OSErr SWUnpauseSpriteLayer(SpriteLayerPtr spriteLayerP);
```

spriteLayerP	The SpriteLayer to be unpaused.

DESCRIPTION

Call this function to undo the effects of SWPauseSpriteLayer.

SEE ALSO

SWPauseSpriteLayer

Sprite Routines

SWAddFrame

This function will add a Frame to an existing Sprite.

```
void SWAddFrame(SpritePtr srcSpriteP, FramePtr newFrameP);
```

srcSpriteP	An existing Sprite.
newFrameP	A new Frame to be added to the Sprite.

DESCRIPTION

The SWAddFrame function will add a Frame to an existing Sprite, making it the last Frame of that Sprite. This Frame may also be added to other Sprites so that they may share

Frames, thus saving memory. The function will return an error if you try to add more Frames than the Sprite can contain. (You specify how many Frames a Sprite can contain when creating it with SWCreateSprite, or if you use a function like SWCreateSpriteFromPictResource, the number of frames will be determined automatically.)

SWBounceSprite

This function can be used as part of a Sprite's movement routine to make the Sprite bounce around the screen.

```
Boolean SWBounceSprite(SpritePtr srcSpriteP);
```

srcSpriteP	A Sprite being moved.

DESCRIPTION

This function will check to see if a Sprite has gone outside its moveBoundsRect, and if so, will "bounce" it back inside that rectangle. It also changes the sprite's delta, so if the Sprite hits, for example, the left side, the sprite's horizMoveDelta will be changed from negative to positive. A Boolean value is returned indicating whether the Sprite was bounced or not; it will be true if it was bounced, false if it was not.

This function can be used as part of a moveProc; the simplest example being:

```
SW_FUNC void MySpriteMoveProc(SpritePtr spriteP)
{
	Boolean	wasBounced;
	SWOffsetSprite(spriteP, spriteP->horizMoveDelta,
		spriteP->vertMoveDelta);
	wasBounced = SWBounceSprite(spriteP);
}
```

SEE ALSO

SWSetSpriteMoveBounds

SWWrapSprite

SWCloneSprite

This function will create a duplicate of an existing Sprite.

```
OSErr SWCloneSprite(SpritePtr cloneSpriteP,
                              SpritePtr *newSpriteP,              void*
                            spriteStorageP);
```

cloneSpriteP	An existing Sprite to be cloned.
newSpriteP	The newly created Sprite.
spriteStorageP	Pointer to memory in which the Sprite structure will be stored. If NULL, SpriteWorld will allocate the needed memory.

DESCRIPTION

The SWCloneSprite function will create a duplicate of an existing Sprite. The Frame set of the Sprite is not duplicated, instead both Sprites share the same Frame set. Naturally, this can save a lot of memory if you have many Sprites that look the same.

Generally, you will want to load one copy of each Sprite (the "master" Sprites), and clone these whenever you need to add a Sprite to the animation. Thus, the master Sprites are never added to the animation themselves; just their clones. This enables you to freely remove and dispose any Sprites that are a part of the animation without having to worry that you'll accidentally dispose the master Sprite that the clones were made from.

When you lock a Sprite, all Sprites sharing the same set of Frames will also be locked. This means that when you lock a master Sprite or any of its clones, all other clones of that Sprite will also be locked, including any clones you make of it in the future. This is because the only part of a Sprite that must be locked is the Frames, and because the same set of Frames are shared between the master Sprite and its clones, locking one Sprite will lock all Sprites in that "family".

SEE ALSO

SWFastCloneSprite

SWCloneSpriteFromTile

This function will create a Sprite made up of one or more Tile Frames.

```
OSErr SWCloneSpriteFromTile(SpriteWorldPtr spriteWorldP,
                             SpritePtr* newSpriteP,                void*
                            spriteStorageP,                        short
                            firstTileID,                           short
                            lastTileID)
```

spriteWorldP	The SpriteWorld containing the Tiles.
newSpriteP	The newly created Sprite.
spriteStorageP	Pointer to memory in which the Sprite structure will be stored. If NULL, SpriteWorld will allocate the needed memory.
firstTileID	The first tileID to be added to the Sprite.
lastTileID	The last tileID to be added to the Sprite.

DESCRIPTION

This function creates a new Sprite and adds to it all the Tile Frames from firstTileID to lastTileID. This function is very fast since it uses the same Frames the Tiles use; it does not create copies of them. Even so, it is perfectly safe to dispose of the original Tiles the

Sprite was cloned from while still keeping the Sprite in the animation, since SpriteWorld is smart enough to not dispose the Tile Frames unless they are no longer in use.

Generally this function is used to allow tiles to move. For example, in Super Mario Bros. for the NES, when Mario jumps and hits a question block, the block moves up a few pixels and then back down as Mario's hand slams into it. Normally this wouldn't be possible for Tiles, but by creating a Sprite with the same Frame or Frames as the Tile, and placing it over the real Tile while changing the ID of the real Tile to a blank space, you can allow the Tile to move. Once it's done moving, simply dispose the Sprite and change the real Tile back to the proper ID.

The current frame index of the Sprite is automatically set by this function if possible. That is, in the example above of a Sprite created from tileIDs 5, 6, and 7, if the current tile image for tileID 5 were 7, the current frame index of the Sprite would be set to 2, since that is the frameIndex for the Sprite that is the same as the current image for tileID 5. This is particularly useful for animated Tiles, since it allows you to convert a Tile into a Sprite, where the Sprite has the same image as the animated Tile did.

If you added Tiles that use partial masks to the new Sprite, the Sprite's drawProc should be a partialMaskDrawProc. If the Tiles you added have no mask at all, you may use a rectDrawProc for speed. Otherwise, use one of the standard maskDrawProcs.

SWCloseSprite

This function will close an existing Sprite.

```
void SWCloseSprite(SpritePtr deadSpriteP);
```

deadSpriteP	A Sprite to be closed.

DESCRIPTION

This function is equivalent to SWDisposeSprite, except that it does not attempt to dispose the memory taken up by the Sprite's SpriteRec. It will dispose all of the Sprite's Frames, unless those Frames are shared by undisposed clone Sprites. Because it does not dispose the memory taken up by the Sprite's SpriteRec, this function should be called instead of SWDisposeSprite when you want to dispose a Sprite that was created using your own memory area (instead of allocating memory in the heap for the SpriteRec), if you don't want that memory area to be disposed when you get rid of the Sprite.

∆ WARNING ∆

You should not close a Sprite that is part of a running animation, or SpriteWorld will crash when it tries to process a Sprite that no longer exists. First remove the Sprite with SWRemoveSprite, then close it.

SEE ALSO

SWDisposeSprite

SWCreateSprite

This function will create a new Sprite with no Frames.

```
OSErr SWCreateSprite(SpritePtr *newSpriteP,                       void*
                                 spriteStorageP,                   short
                                 maxFrames);
```

newSpriteP	A newly created Sprite.
spriteStorageP	Pointer to memory in which the Sprite structure will be stored. If NULL, SpriteWorld will allocate the needed memory.
maxFrames	A value that indicates the maximum number of Frames to be contained in the Sprite.
userData	User defined information to be associated with the Sprite.

DESCRIPTION

The SWCreateSprite function will create and initialize a new Sprite with an empty Frame set. For the Sprite to be of any use, you must create and add some Frames. Normally, you will create Sprites with one of the SWCreateSprite… routines listed below. For information on “manually” creating Frames, see the SpriteWorld source code.

SWCreateSpriteFromCicnResource

This function will create a new Sprite complete with a set of Frames created from a series of color icon (‘cicn’) resources.

```
OSErr SWCreateSpriteFromCicnResource(
                                   SpriteWorldPtr destSpriteWorld,
                                   SpritePtr *newSpriteP,          void*
                                 spriteStorageP,                   short
                                 cIconID,                          short
                                 maxFrames,                        short
                                 maskType);
```

destSpriteWorld	The SpriteWorld that will ultimately contain the Sprite. This is passed so that the same GDevice used elsewhere in the SpriteWorld can be used to create the Frames for the Sprite.
newSpriteP	The newly created Sprite.
spriteStorageP	Pointer to memory in which the Sprite structure will be stored. If NULL, SpriteWorld will allocate the needed memory.
cIconID	The resource id of a color icon (‘cicn’) resource from which the first Frame of the Sprite will be created.
maxFrames	A value that indicates the maximum number of Frames to be contained in the Sprite, and the number of color icon resources available to create them.
maskType	A value that indicates what type(s) of mask should be created for the Sprite. For a description of these flags and their meaning see below.

DESCRIPTION

The SWCreateSpriteFromCicnResource function will create and initialize a new Sprite, and create a set of Frames for the Sprite from color icon resources in sequence starting with the specified cIconID. The number of Frames to be created is specified by the maxFrames parameter. The color icon resources for the Frames are expected to have sequential id numbers. An error code is returned if any memory allocation fails, or any of the color icon resources in the sequence cannot be found.

The destination SpriteWorld is passed as a parameter so that the SpriteWorld's GDHandle can be used when creating the GWorld(s) for this Sprite.

The maskType parameter can currently be one of the values described here:

kNoMask	A value indicating that no mask should be created for the Frames of the Sprite. (A rect blitter will be used.)
kRegionMask	A value indicating that a QuickDraw region (RgnHandle) should be created for possible use as a mask for the Frames of the Sprite.
kPixelMask	A value indicating that an offscreen GWorld should be created, and used as a mask for the Frames of the Sprite. This GWorld will be the same bit depth as the Frame image and is suitable for use with a custom mask blitter, such as those provided with SpriteWorld.
kFatMask	This value is equivalent to kRegionMask + kPixelMask. This results in a Frame that contains both of the above types of masks. This is useful if your application switches between using QuickDraw and a custom drawing routine at runtime.

SEE ALSO

SWCreateSpriteFromPictResource

SWCreateSpriteFromSinglePict

SWCreateSpriteFromSinglePictXY

SWCreateSpriteFromPictResource

This function will create a new Sprite complete with a set of Frame created from a series of picture ('PICT') resources.

```
OSErr SWCreateSpriteFromPictResource(
                             SpriteWorldPtr destSpriteWorld,
                             SpritePtr *newSpriteP,                 void*
                           spriteStorageP,                          short
                           pictResID,                               short
                           maskResID,                               short
                           maxFrames,                               short
                           maskType);
```

destSpriteWorld	The SpriteWorld that will ultimately contain the Sprite. This is passed so that the same GDevice used elsewhere in the SpriteWorld can be

used to create the Frames for the Sprite.

newSpriteP	The newly created Sprite.
spriteStorageP	Pointer to memory in which the Sprite structure will be stored. If NULL, SpriteWorld will allocate the needed memory.
pictResID	The resource id of a picture (‘PICT’) resource from which the first Frame of the Sprite will be created.
maskResID	The resource id of a picture (‘PICT’) resource from which the first mask of the Frames of the Sprite will be created. This can be the same as pictResID if no part of the Sprite image is pure white (see the section “Sprite Masks and Self-Masking”).
maxFrames	A value that indicates the maximum number of Frames to be contained in the Sprite, and the number of picture resources available to create them.
maskType	A value that indicates what type(s) of mask should be created for the Sprite. For a description of these flags and their meaning see SWCreateSpriteFromCicnResource.

DESCRIPTION

The SWCreateSpriteFromPictResource function will create and initialize a new Sprite, and create a set of Frames for the Sprite from picture resources in sequence starting with the specified `pictResID`. The number of Frames to be created is specified by the maxFrames parameter. The picture resources for the Frames are expected to have sequential id numbers. An error code is returned if any memory allocation fails, or any of the picture resources in the sequence cannot be found.

The destination SpriteWorld is passed as a parameter so that the SpriteWorld’s GDHandle can be used when creating the GWorld(s) for this Sprite.

Note: If you are going to use one of SpriteWorld's mask blitters to draw the Sprite, such as the BlitPixieAllBitMaskDrawProc or the CompiledSprite8BitDrawProc, the background portion of the Sprite's image must be white. This is because the blitter can go faster by assuming the unmasked portion of a Sprite's image is white. The only exception to this is when loading self-masking Sprites that have a background other than white, and you have called SWSetTransparentColor, setting the transparent color to the same color as your Sprite's background. (See SWSetTransparentColor for more info.)

SEE ALSO

SWCreateSpriteFromCicnResource

SWCreateSpriteFromSinglePict

SWCreateSpriteFromSinglePictXY

SWCreateSpriteFromSinglePict

This function will create a new Sprite complete with a set of Frames from a single, multi-frame picture (‘PICT’) resource.

```
OSErr SWCreateSpriteFromSinglePict(
                                   SpriteWorldPtr destSpriteWorld,
                                   SpritePtr *newSpriteP,           void*
                                 spriteStorageP,                    short
                                 pictResID,                         short
                                 maskResID,                         short
                                 frameDimension,                    short
                                 borderWidth,                       short
                                 maskType);
```

destSpriteWorld	The SpriteWorld that will ultimately contain the Sprite. This is passed so that the same GDevice used elsewhere in the SpriteWorld can be used to create the Frames for the Sprite.
newSpriteP	The newly created Sprite.
spriteStorageP	Pointer to memory in which the Sprite structure will be stored. If NULL, SpriteWorld will allocate the needed memory.
pictResID	The resource id of a picture (‘PICT’) resource from which the Frames of the Sprite will be created. If frameDimension is zero, this must also be the resource id of a ‘nrct’ resource (see below).
maskResID	The resource id of a picture (‘PICT’) resource from which the masks of the Frames of the Sprite will be created. This can be the same as pictResID if no part of the Sprite image is pure white (see the section “Sprite Masks and Self-Masking”).
frameDimension	If all the Frame images in the picture are the same height and width, they can be arranged in a horizontal or vertical strip. This parameter tells SpriteWorld the width of each Frame if the strip is horizontal, or the height if it is vertical. This parameter must be 0 if you are using a ‘nrct’ resource (see below).
borderWidth	The number of unused pixels between Frames (if any).
maskType	A value that indicates what type(s) of mask should be created for the Sprite. For a description of these flags and their meaning see SWCreateSpriteFromCicnResource.

DESCRIPTION

The SWCreateSpriteFromSinglePict function will create and initialize a new Sprite, and create a set of Frames for the Sprite from a picture resource with the specified `pictResID`. This picture resource will typically contain several images which represent the Frames of your animated Sprite. There are two distinct ways of using this routine. In the first described below, the images of the Frames can be placed anywhere in the picture, in any order, and can be differing sizes. In the second method, the Frame images must all fit within Rects of the same size, and must be arranged in order in a horizontal or vertical strip. The frameDimension parameter tells SpriteWorld which of the two methods you want to use.

In the “any size, any order” method of using this routine, in addition to the picture resource, you must also create a ‘nrct’ resource with the same ID number as the PICT. The nrct resource is similar to a ‘STR#’ resource, except that the records it contains are Rect data structures rather than Pstrings. In each record of the nrct resource you will give the coordinates of the rectangle within the picture that encloses one Frame of your Sprite. The number of Rects you enter in the nrct resource determines the number of Frames to the

Sprite. You must pass zero for the frameDimension parameter if you are using this method.

With the “horizontal or vertical strip” method, you don’t have to create a ‘nrct’ resource. Instead, the Frame images are arranged horizontally or vertically in the picture. All the Frame images must fit within the same size bounding Rect. We’ll assume that the strip is horizontal for the remainder of this paragraph. In that case, the height of the picture is used by SpriteWorld as the height of each Frame’s Rect, and the frameDimension parameter provides the width. Within the picture, the first Frame image must start at the top-left of the picture, and each successive image must be to the right of that.

The borderWidth parameter to this function specifies the number of unused pixels between frames. You can pass 0 if your Frames are right next to each other, or another number such as 1, if your Frames have a 1-pixel border between them. Usually, if you use a border between Sprites, you will want to make the border black so the boundaries between the Frames will be obvious while you work on the picture.

Arranging the images in a horizontal strip may seem the most obvious approach, but there is an advantage to the vertical strip option. Both CopyBits and BlitPixie are optimized to work with rectangles that have a left origin that is an even multiple of four. For the best speed, you could make sure that the width of each Frame, including the size of the border between each Frame, is a multiple of four. If you arrange the images in a vertical strip, you don’t have to concern yourself with this calculation, as all the Frames will have a left origin of zero. SpriteWorld will determine whether the images are arranged vertically or horizontally by checking whether the PICT is wider than it is high.

An error code is returned by SWCreateSpriteFromSinglePict if any memory allocation fails, if either of the picture resources cannot be found, or if the frameDimension parameter is zero and no nrct resource is found.

The destination SpriteWorld is passed as a parameter so that the SpriteWorld’s GDHandle can be used when creating the GWorld(s) for this Sprite.

Note: If you are going to use one of SpriteWorld's mask blitters to draw the Sprite, such as the BlitPixieAllBitMaskDrawProc or the CompiledSprite8BitDrawProc, the background portion of the Sprite's image must be white. This is because the blitter can go faster by assuming the unmasked portion of a Sprite's image is white. The only exception to this is when loading self-masking Sprites that have a background other than white, and you have called SWSetTransparentColor, setting the transparent color to the same color as your Sprite's background. (See SWSetTransparentColor for more info.)

SEE ALSO

SWCreateSpriteFromCicnResource

SWCreateSpriteFromPictResource

SWCreateSpriteFromSinglePictXY

SWCreateSpriteFromSinglePictXY

This function will create a new Sprite complete with a set of Frames from a single, multi-frame picture (‘PICT’) resource. With this routine, the Frame images can be arranged in rows and columns.

```
OSErr SWCreateSpriteFromSinglePictXY(
                            SpriteWorldPtr destSpriteWorld,
                            SpritePtr *newSpriteP,               void*
                           spriteStorageP,                         short
                           pictResID,                              short
                           maskResID,                              short
                           frameWidth,                             short
                           frameHeight,                            short
                           horizBorderWidth,                       short
                           vertBorderHeight,                       short
                           maxFrames,                              short
                           maskType);
```

destSpriteWorld	The SpriteWorld that will ultimately contain the Sprite. This is passed so that the same GDevice used elsewhere in the SpriteWorld can be used to create the Frames for the Sprite.
newSpriteP	The newly created Sprite.
spriteStorageP	Pointer to memory in which the Sprite structure will be stored. If NULL, SpriteWorld will allocate the needed memory.
pictResID	The resource id of a picture (‘PICT’) resource from which the Frames of the Sprite will be created.
maskResID	The resource id of a picture (‘PICT’) resource from which the masks of the Frames of the Sprite will be created. This can be the same as pictResID if no part of the Sprite image is pure white (see the section “Sprite Masks and Self-Masking”).
frameWidth	The width of the Frame images, in pixels.
frameHeight	The height of the Frame images.
horizBorderWidth	The width of the horizontal separation between each Frame image in the PICT. See below for more information.
vertBorderHeight	The height of the vertical separation between each row of Frame images in the PICT. See below for more information.
maxFrames	The total number of valid Frame images in the PICT. If zero is passed, SWCreateSpriteFromSinglePictXY will calculate this number from the pict.
maskType	A value that indicates what type of mask should be created for the Sprite. For a description of these flags and their meaning see SWCreateSpriteFromCicnResource.

DESCRIPTION

The SWCreateSpriteFromSinglePictXY function will create and initialize a new Sprite, and create a set of Frames for the Sprite from a picture resource with the specified `pictResID`. This picture resource will typically contain several images which represent the Frames of your animated Sprite. This routine is similar to SWCreateSpriteFromSinglePict; the major difference being that with this routine, the Frame images in the PICT can be arranged in rows and columns. The following requirements apply to the arrangement of the Frame images in the PICT:

• The Frame images can be laid out in any number of rows and columns. The images will be read from left to right and top to bottom. The first image must begin at the top-left of the PICT.

• The Frame images in a row must be separated by a border whose width you specify in the horizBorderWidth parameter. This value is user-defined to allow optimum alignment of the Frame images. Both CopyBits and BlitPixie are fastest when the left side of the source rect is an even multiple of four. When assembling a graphic of Frame images, you can adjust the horizontal separation of the Frame images to achieve this alignment, and then set the horizBorderWidth parameter accordingly.

• Each row of Frame images will be separated by a border that you specify in the vertBorderHeight parameter. You can pass any value you wish, including 0.

• If you pass zero as the maxFrames parameter, SWCreateSpriteFromSinglePictXY will calculate the number of Frames as follows:

```
maxFrames = (PICTwidth/frameWidth) * (PICTheight/frameHeight);
```

This calculation will be incorrect if the valid Frame images end before the bottom-right of the PICT, or if the accumulated horizontal or vertical border spaces adds up to more than the width or height, respectively, of a Frame image.

An error code is returned by SWCreateSpriteFromSinglePictXY if any memory allocation fails, if either of the picture resources cannot be found, or if the bottom-right of the PICT is reached before the requested number of Frame images have been read.

The destination SpriteWorld is passed as a parameter so that the SpriteWorld’s GDHandle can be used when creating the GWorld(s) for this Sprite.

Note: If you are going to use one of SpriteWorld's mask blitters to draw the Sprite, such as the BlitPixieAllBitMaskDrawProc or the CompiledSprite8BitDrawProc, the background portion of the Sprite's image must be white. This is because the blitter can go faster by assuming the unmasked portion of a Sprite's image is white. The only exception to this is when loading self-masking Sprites that have a background other than white, and you have called SWSetTransparentColor, setting the transparent color to the same color as your Sprite's background. (See SWSetTransparentColor for more info.)

SEE ALSO

SWCreateSpriteFromCicnResource

SWCreateSpriteFromPictResource

SWCreateSpriteFromSinglePict

SWDisposeSprite

This function will dispose of an existing Sprite, releasing the memory it occupies.

```
void SWDisposeSprite(SpritePtr *deadSpriteP);
```

deadSpriteP	A Sprite to be disposed.

DESCRIPTION

The SWDisposeSprite function will dispose of a Sprite. It will also dispose of the Sprite’s Frames, unless those Frames are shared by undisposed clone Sprites. If the Sprite does not use memory in the heap for the SpriteRec (that is, if you did not pass NULL as the spriteStorageP parameter when creating the Sprite), then you should not dispose it with SWDisposeSprite, but should use SWCloseSprite instead.

∆ WARNING ∆

You must not call SWDisposeSprite on a Sprite that is part of a running animation, or SpriteWorld will crash when it tries to process a Spite that no longer exists. SWRemoveSpriteFromAnimation will remove the Sprite from its Layer, and then optionally dispose of it.

SEE ALSO

SWCloseSprite

SWRemoveSprite

SWRemoveSpriteFromAnimation

SWFastCloneSprite

This function will create a duplicate of an existing Sprite, sharing the frameArray of the master Sprite in order to save time and memory.

```
OSErr SWFastCloneSprite(SpritePtr masterSpriteP,
                               SpritePtr *newSpriteP,                 void*
                             spriteStorageP);
```

masterSpriteP	An existing Sprite to be cloned.
newSpriteP	The newly created Sprite.
spriteStorageP	Pointer to memory in which the Sprite structure will be stored. If NULL, SpriteWorld will allocate the needed memory.

DESCRIPTION

This function is identical to SWCloneSprite, except that the clone "shares" the frameArray of the master Sprite, and therefore a new frameArray does not have to be created, saving a little time and memory. There is a side-effect to this, however: if you modify the frameArray of either the master Sprite or any of the clones that were created with this function, both the master Sprite and all the clones created by this function will be affected. For instance, if you call SWRemoveFrame to remove a Frame from the master Sprite, that Frame will also be removed from all the clones that were created with this function. Since you generally won't need to modify the Frames of a Sprite after creating it, this shouldn't be a problem.

Another difference between this function and SWCloneSprite is that this function does not increment the useCount variable of each Frame used by the Sprite (also done to save time). One side effect of this is that if you dispose the master Sprite from which you made the clone, all the Frames will be disposed, since they don't "know" that another Sprite is using them. This could cause problems if the clone was currently being used in the animation. However, in most cases, you won't dispose the master Sprite until the animation is over, so this shouldn't be a problem either.

In general, this function is "riskier" than SWCloneSprite, and only slightly faster. (SWCloneSprite is very fast as it is.) You should only use this function in situations where you need to clone lots of Sprites at once and need something faster than SWCloneSprite, and where the above-mentioned side-effects won't bother you. If you don't understand the side-effects, you shouldn't use this function.

SWInsertFrame

This function will insert a Frame in an existing Sprite.

```
void SWInsertFrame(SpritePtr srcSpriteP, FramePtr newFrameP,        long
                           frameIndex);
```

srcSpriteP	An existing Sprite.
newFrameP	A new Frame to be added to the Sprite.
frameIndex	The index of where the new frame is to be inserted.

DESCRIPTION

SWInsertFrame adds a Frame to a Sprite, but unlike SWAddFrame, it allows you to specify the index of the new Frame's location in the Sprite's frameArray, instead of adding the Frame to the end of the existing Frames . All Frames above and equal to frameIndex will be moved up one Frame, allowing the new Frame to be placed in the empty location. Keep in mind that the frameIndex values start at 0, so if you want to insert a Frame into the 4th position, you should use a frameIndex of 3. An error code will be returned if there is no room to add the new Frame, or if frameIndex is out of bounds.

SWIsPointInSprite

This functions determines whether a given point is within a given Sprite.

```
Boolean SWIsPointInSprite(SpritePtr srcSpriteP,                     Point
                           testPoint);
```

srcSpriteP	The Sprite to test.
testPoint	The point to test.

DESCRIPTION

The SWIsPointInSprite returns TRUE if the point is within the Sprite’s Frame rectangle, and FALSE otherwise.

SPECIAL CONSIDERATIONS

Note that the coordinates of the Sprite are local to the worldRect of the SpriteWorld; you will have to take this into account if the worldRect is different from the window’s portRect, and testPoint is in coordinates local to the window.

SWIsSpriteInRect

This functions determines whether a given Sprite is within a given rectangle.

```
Boolean SWIsSpriteInRect(SpritePtr srcSpriteP,                    Rect*
                                  testRect);
```

srcSpriteP	The Sprite to test.
testRect	The rectangle to test.

DESCRIPTION

The SWIsSpriteInRect returns TRUE if any part of the Sprite is within the rectangle, and FALSE otherwise.

SPECIAL CONSIDERATIONS

Note that the coordinates of the Sprite are local to the worldRect of the SpriteWorld; you will have to take this into account if the worldRect is different from the window's portRect, and testRect is in coordinates local to the window.

SWIsSpriteFullyInRect

This functions determines whether a given Sprite is entirely within a given rectangle.

```
Boolean SWIsSpriteFullyInRect(SpritePtr srcSpriteP,               Rect*
                                  testRect);
```

srcSpriteP	The Sprite to test.
testRect	The rectangle to test.

DESCRIPTION

The SWIsSpriteFullyInRect returns TRUE if a Sprite's destRect is entirely enclosed within the rectangle, and FALSE otherwise.

SPECIAL CONSIDERATIONS

Note that the coordinates of the Sprite are local to the worldRect of the SpriteWorld; you will have to take this into account if the worldRect is different from the window's portRect, and testRect is in coordinates local to the window.

SWLockSprite

This functions locks a Sprite in preparation for animation.

```
void SWLockSprite(SpritePtr srcSpriteP);
```

srcSpriteP	A Sprite to be locked.

DESCRIPTION

The SWLockSprite function will lock the Sprite including all the Frames contained within. Before a Sprite can be animated it must first be locked.

SPECIAL CONSIDERATIONS

You must be careful when adding a Sprite to an animation that is already running. The new Sprite must be locked before it can be animated.

SWMoveSprite

This function will move a Sprite's current position to an absolute horizontal, and vertical coordinate.

```
void SWMoveSprite(SpritePtr srcSpriteP,                          short
                                horizLoc,                          short
                                vertLoc);
```

srcSpriteP	A Sprite to be moved.
horizLoc	A value indicating the absolute horizontal coordinate to which the Sprite will be moved.
vertLoc	A value indicating the absolute vertical coordinate to which the Sprite will be moved.

DESCRIPTION

The SWMoveSprite function is used to move a Sprite's current position to an absolute horizontal, and vertical coordinate. The Sprite's last position is remembered so that when the animation is rendered the Sprite will be properly erased from its last known position.

If the Sprite has a movement routine installed, it will not be called by this function.

SPECIAL CONSIDERATIONS

Note that the horizLoc and vertLoc coordinates of the Sprite are local to the worldRect of the SpriteWorld, not the portRect of the "parent" window.

SWOffsetSprite

This function will offset the Sprite's current position to a relative horizontal, and vertical coordinate.

```
void SWOffsetSprite(SpritePtr srcSpriteP,                            short
                           horizDelta,                               short
                           vertDelta);
```

srcSpriteP	A Sprite to be moved.
horizDelta	A value indicating the relative horizontal coordinate by which the Sprite will be offset.
vertDelta	A value indicating the relative vertical coordinate by which the Sprite will be offset.

DESCRIPTION

The SWOffsetSprite function is used to offset a Sprite's current position to a relative horizontal, and vertical coordinate. The Sprite's last position is remembered so that when the animation is rendered the Sprite will be properly erased from its last known position.

If the Sprite has a movement routine installed, it will not be called by this function.

SWPixelCollision

This function tests whether the pixel masks of two 8-bit Sprites are intersecting, and returns TRUE if any part of the Sprites overlap.

```
Boolean SWPixelCollision(SpritePtr srcSpriteP, dstSpriteP);
```

srcSpriteP	
dstSpriteP	The two Sprites to be tested

DESCRIPTION

SWPixelCollision can be called as part of a Sprite's collideProc to determine if the actual images of the two Sprites are overlapping. The collideProc will be called whenever the destination rects of the Sprites overlap; SWPixelCollision can be used as a secondary test. Because it uses the Sprites' pixel masks, this function will be valid for any Sprites, however convoluted their shape. In order for this function to work, the Sprites must have been created using kPixelMask or kFatMask, and the SpriteWorld must be 8-bit depth. SWPixelCollision is somewhat faster than SWRegionCollision (though slower than SWRadiusCollision), and doesn't require that the Sprite have a region mask.

Δ WARNING Δ

If the srcSpriteP's depth is not 8-bit, this function will return 0, since this function requires an 8-bit mask. It is up to you to make sure you are in 8-bit depth before calling this function, and to use an alternate routine such as SWRegionCollision if you are not. Otherwise, a collision may not be reported even if there is one.

SEE ALSO

SWCollideSpriteLayer

SWSetSpriteCollideProc

SWRadiusCollision

SWRegionCollision

SWRadiusCollision

This function tests whether the outlines of two circular Sprites are intersecting, returning TRUE if they are.

```
Boolean SWRadiusCollision(SpritePtr srcSpriteP, dstSpriteP);
```

srcSpriteP	
dstSpriteP	The two Sprites to be tested

DESCRIPTION

SWRadiusCollision can be called as part of a Sprite's collideProc to determine if the (presumed) circular outlines of the two Sprites are overlapping. The collideProc will be called whenever the destination rects of the Sprites overlap; SWRadiusCollision can be used as a secondary test. Because it uses a radius calculated from the width of the Sprite's destination rects, this routine is intended for circular Sprites that fill the Frame's rect. This routine is much faster than SWRegionCollision or SWPixelCollision.

SEE ALSO

SWCollideSpriteLayer

SWSetSpriteCollideProc

SWRegionCollision

SWPixelCollision

SWRegionCollision

This function tests whether the regions of two Sprites are intersecting, and returns TRUE if any part of the Sprites overlap.

```
Boolean SWRegionCollision(SpritePtr srcSpriteP, dstSpriteP);
```

srcSpriteP	
dstSpriteP	The two Sprites to be tested

DESCRIPTION

SWRegionCollision can be called as part of a Sprite's collideProc to determine if the actual images of the two Sprites are overlapping. The collideProc will be called whenever the destination rects of the Sprites overlap; SWRegionCollision can be used as a secondary test. Because it uses regions, this function will be valid for any Sprites, however convoluted their shape. In order for this function to work, the Sprites must have been created using kRegionMask or kFatMask.

SEE ALSO

SWCollideSpriteLayer

SWSetSpriteCollideProc

SWRadiusCollision

SWPixelCollision

SWRemoveFrame

This function will remove a Frame from an existing Sprite.

```
void SWRemoveFrame(SpritePtr srcSpriteP, FramePtr oldFrameP);
```

srcSpriteP	An existing Sprite.
oldFrameP	A Frame to be removed from the Sprite.

DESCRIPTION

The SWRemoveFrame function will remove a Frame from an existing Sprite. You will probably never want to do this, since you can simply dispose of a Sprite and its Frames automatically when your animation is finished.

SWRemoveSpriteFromAnimation

This function erases a Sprite from the animation, removes it from its Layer, and optionally disposes of it.

```
void SWRemoveSpriteFromAnimation(SpriteWorldPtr spriteWorldP,
                                  SpritePtr spriteP,                 Boolean
                                disposeOfSprite);
```

spriteWorldP	The SpriteWorld containing the Sprite.
spriteP	The Sprite to be removed.
disposeOfSprite	If true, dispose of the Sprite after making it invisible and removing it from its Layer. If false, just make the Sprite invisible and remove it from its Layer.

DESCRIPTION

SWRemoveSpriteFromAnimation provides an easy way to get rid of a Sprite that is part of a running animation. Without SWRemoveSpriteFromAnimation, you would first have to call SWSetSpriteVisible to make the Sprite invisible, then call SWAnimateSpriteWorld so the Sprite would actually be erased from the screen, and finally call SWRemoveSprite to remove it from its Layer. SWDisposeSprite would also have to be called if you wanted to permanently dispose of the Sprite to free up the memory it uses. SWRemoveSpriteFromAnimation does all of this for you.

When you call this function, the Sprite is immediately removed from its current Layer and added to the spriteWorldP->deadSpriteLayerP, a special Layer in the SpriteWorld that is used just for Sprites that need to be erased from the animation and then removed. During the next call to SWAnimateSpriteWorld, any Sprites that are in this Layer are erased. After the Sprite has been erased, SWProcessSpriteWorld will automatically remove it from the deadSpriteLayerP, and optionally dispose of it. The fact that the Sprite is immediately removed from its original Layer as soon as you call this function means that you will not have to worry about accidentally bumping into this Sprite again before it is removed.

The deadSpriteLayerP is not added to the SpriteWorld until SWAnimateSpriteWorld is called, and it is removed when the function finishes. This means that if you cycle through all the SpriteLayers in a MoveProc or CollideProc, you will not encounter any Sprites that are in the deadSpriteLayer. The only time this might happen is if you install a postEraseCallBack or postDrawCallBack that cycles through all the Layers of a SpriteWorld.

SPECIAL CONSIDERATIONS

SWRemoveSpriteFromAnimation should not be used from within a moveProc, frameProc, or collisionProc, except to remove the **source** Sprite of that routine, or a Sprite from another Layer. You should not remove Sprites other than the source Sprite that are in the same Layer as the source Sprite.

SEE ALSO

SWRemoveSprite

SWDisposeSprite

SWSetCurrentFrame

This function will set a Sprite’s current Frame to the one you specify.

```
void SWSetCurrentFrame(SpritePtr srcSpriteP, FramePtr newFrameP);
```

srcSpriteP	An existing Sprite.
newFrameP	A Frame previously added to the Sprite.

DESCRIPTION

The SWSetCurrentFrame function will set the Sprite’s current Frame to the one specified by the newFrameP parameter. The current Frame will be rendered in the animation at the Sprite’s current location. The newFrameP must be a Frame that is already part of the Sprite. If it is not, an error code will be returned. If you wish to set the current Frame to a

Frame that is not part of the Sprite, you should first add the Frame to the Sprite using either SWAddFrame or SWInsertFrame.

SEE ALSO

SWSetCurrentFrameIndex

SWSetCurrentFrameIndex

This function will set a Sprite's current Frame using the specified index into the Sprite's Frame list.

```
void SWSetCurrentFrameIndex(SpritePtr srcSpriteP,                    short
                           frameIndex);
```

srcSpriteP	An existing Sprite.
frameIndex	An index into the Sprite's Frame list.

DESCRIPTION

The SWSetCurrentFrameIndex function will set a Sprite's current Frame using the specified index into the Sprite's Frame list. The current Frame will be rendered in the animation at the Sprite's current location. If you think the current index may already be what you want, you can check it before calling this function, thus saving a little time if it is already what you want:

```
if (srcSpriteP->curFrameIndex != theIndex)
      SWSetCurrentFrameIndex(srcSpriteP, theIndex);
```

SEE ALSO

SWSetCurrentFrame

SWSetSpriteCollideProc

This function sets a Sprite's collision routine, to be called when the Sprite is involved in a collision with another.

```
void SWSetSpriteCollideProc(SpritePtr srcSpriteP,
                            CollideProcPtr collideProc);
```

srcSpriteP	An existing Sprite.
collideProc	A new collision routine.

DESCRIPTION

The SWSetSpriteCollideProc function is used to specify a collision routine, to be called when the Sprite is involved in a collision with another. This routine may do some further processing to determine if the Sprites have actually collided, and if so perform some action such as playing an explosion sound.

A collideProc routine has the form:

```
SW_FUNC void MyCollideProc(SpritePtr srcSpriteP,
                                         SpritePtr dstSpriteP,
                                         Rect* sectRect);
```

srcSpriteP	A Sprite from the source SpriteLayer.
srcSpriteP	A Sprite from the destination SpriteLayer.
sectRect	A rectangle defining the area of overlap between the two Sprites.

SEE ALSO

SWCollideSpriteLayer

SWSetSpriteDrawProc

The function will set a Sprite's drawing routine to the one you specify.

```
OSErr SWSetSpriteDrawProc(SpritePtr srcSpriteP,
                                DrawProcPtr drawProc);
```

srcSpriteP	An existing Sprite.
drawProc	A new drawing routine for the Sprite.

DESCRIPTION

The SWSetSpriteDrawProc function will set the drawing routine the Sprite uses to render itself in the offscreen work area of the SpriteWorld. The Sprite's default drawing routine uses CopyBits.

The options for drawProc currently available are:

SWStdSpriteDrawProc	CopyBits (use kRegionMask or kFatMask when creating the Sprite).
BlitPixieAllBitMaskDrawProc	
BlitPixie8BitMaskDrawProc	BlitPixie with a mask (use kPixelMask or kFatMask when creating the Sprite).
BlitPixieAllBitPartialMaskDrawProc	
BlitPixie8BitPartialMaskDrawProc	BlitPixie with a mask when the un-masked background of the Sprite is not white. (use kPixelMask or kFatMask when creating the Sprite).
BlitPixieAllBitRectDrawProc	
BlitPixie8BitRectDrawProc	BlitPixie with no mask needed (use this for rectangular Sprites only).

BPAllBitInterlacedMaskDrawProc	
BP8BitInterlacedMaskDrawProc	Interlaced BlitPixie with a mask. Only for use when the offscreenDrawProc and the screenDrawProc are BPInterlacedDrawProc.
BPAllBitInterlacedPartialMaskDrawProc	
BP8BitInterlacedPartialMaskDrawProc	Interlaced BlitPixie with a partial mask.
BPAllBitInterlacedRectDrawProc	
BP8BitInterlacedRectDrawProc	Interlaced BlitPixie with no mask needed (use this for rectangular Sprites only).
CompiledSprite8BitDrawProc	Compiled Sprites (use kPixelMask or kFatMask when creating the Sprite).

An error code is returned if BlitPixie or CompiledSprite8BitDrawProc is specified and the bit depth of the SpriteWorld is not 8 bits, if the mask of the Sprite is inappropriate for the requested drawProc, or if CompiledSprite8BitDrawProc is requested and the Sprite has not been compiled.

SWSetSpriteFrameAdvance

This function will set the value by which the current Frame index of the Sprite will be automatically incremented.

```
void SWSetSpriteFrameAdvance(SpritePtr srcSpriteP,                short
                                 frameAdvance);
```

srcSpriteP	An existing Sprite.
frameAdvance	The value by which the current Frame index will be automatically incremented.

DESCRIPTION

The SWSetSpriteFrameAdvance function allows you to specify the value by which the current Frame index of the Sprite will be automatically incremented. This will usually be one or zero. The Sprite’s current Frame index will be automatically incremented when the Sprite is processed by the SWProcessSpriteWorld function. The Frame index is a number referring to the Frame images that belong to a Sprite; the first Frame corresponds to an index value of 0. When a Sprite is created, the frameAdvance is set to 1 by default.

SEE ALSO

SWSetSpriteFrameAdvanceMode

SWSetSpriteFrameRange

SWSetSpriteFrameAdvanceMode

This function sets the mode of automatic frame advancement for a Sprite.

```
void SWSetSpriteFrameAdvanceMode(SpritePtr srcSpriteP,
                                 AdvanceType advanceMode);
```

srcSpriteP	An existing Sprite.
advanceMode	The mode of frame advancement; either kSWWrapAroundMode or kSWPatrollingMode.

DESCRIPTION

This function allows you to specify the mode of automatic frame advancement for a Sprite. When the sprite's Frame index is automatically advanced by SWProcessSpriteWorld using the value set by SWSetSpriteFrameAdvance, if the frame index is advanced past the first or last frame of the Sprite, it can either wrap around, or change direction and go the other way. You can control which action will occur by passing either kSWWrapAroundMode or kSWPatrollingMode as the advanceMode parameter.

As an example, if the Sprite's Frame advancement is set to 1, and the Sprite has four frames, this is the order that the frames would be processed for each mode:

kSWWrapAroundMode: 0-1-2-3-0-1-2-3

kSWPatrollingMode: 0-1-2-3-2-1-0-1-2-3

SEE ALSO

SWSetSpriteFrameAdvance

SWSetSpriteFrameProc

This function sets the routine to be called when the Sprite’s current Frame is advanced.

```
void SWSetSpriteFrameProc(SpritePtr srcSpriteP,
                          FrameProcPtr frameProc);
```

srcSpriteP	An existing Sprite.
frameProc	A routine to be called when the current Frame is advanced.

DESCRIPTION

The SWSetSpriteFrameProc function is used to specify a routine to be called when the current Frame of the Sprite is to be advanced. This routine could do some additional processing in order to determine which Frame of the Sprite should be made current.

A frameProc routine has the form:

```
SW_FUNC void (*FrameProcPtr)(SpritePtr srcSpriteP,
                             FramePtr curFrameP,
                             long *frameIndex);
```

srcSpriteP	A Sprite being processed by SWProcessSpriteWorld.
curFrameP	The current Frame of the Sprite.
frameIndex	A value indicating the index of the current Frame. Your function may change this value indicating a different Frame to be made current.

The frameIndex variable is simply a pointer to the sprite's curFrameIndex variable. Adjusting this value is the same as adjusting the srcSpriteP->curFrameIndex variable. You must use this index to change the sprite's frame; changing the sprite's curFrameP variable directly will not work.

SWSetSpriteFrameRange

This function specifies the range of Frame indexes within which the current Frame of the Sprite will be advanced.

```
void SWSetSpriteFrameRange(SpritePtr srcSpriteP,                    short
                           firstFrameIndex,                          short
                           lastFrameIndex);
```

srcSpriteP	An existing Sprite.
firstFrameIndex	A value indicating the index of the first Frame in the range.
lastFrameIndex	A value indicating the index of the last Frame in the range.

DESCRIPTION

The SWSetSpriteFrameRange function is used to specify the range of Frame indexes within which the current Frame of the Sprite will be advanced. The Frame index is a number referring to the Frame images that belong to a Sprite; the first Frame corresponds to an index value of 0. This function allows you use a subset of a Sprite’s Frames to be animated. The current Frame of the Sprite will be automatically advanced when the Sprite is processed by SWProcessSpriteWorld. When a Sprite is created, firstFrameIndex is set to 0 and lastFrameIndex is set to maxFrames-1, so by default the Sprite will cycle through its full set of Frames.

This function checks to make sure the Sprite's curFrameIndex is within the new frame range. If not, it calls SWSetCurrentFrameIndex to change it.

SEE ALSO

SWSetSpriteFrameAdvance

SWSetSpriteFrameTime

This function sets the time interval between automatic advances of the Sprite’s current Frame.

```
void SWSetSpriteFrameTime(SpritePtr srcSpriteP,                    long
                           timeInterval);
```

srcSpriteP	An existing Sprite.
timeInterval	A value indicating the millisecond time interval between Frame advances. A value of 0 indicates the Frame advance should happen as often as possible. A value of -1 indicates the Frame advance should never happen.

DESCRIPTION

The SWSetSpriteFrameTime function is used to set the time interval between automatic advances of the Sprite's current Frame. To advance the current Frame as quickly as possible pass zero into the timeInterval parameter. The current Frame of the Sprite will be automatically advanced when the Sprite is processed by SWProcessSpriteWorld after the specified time interval has passed. When a Sprite is created, the time interval is set to -1 by default. In order for the Sprite to change Frames, you must set the time interval with SWSetSpriteFrameTime, or install a frameChangeProc with SWSetSpriteFrameProc.

SPECIAL CONSIDERATIONS

Note that SWSetSpriteWorldMaxFPS places a limit on how often the animation will actually be rendered, and thus places a lower limit on the time interval between Frame changes. To give SWSetSpriteFrame complete control over Sprite's Frame changes, do not use SWSetSpriteWorldMaxFPS to set the speed of the animation.

SEE ALSO

SWSetSpriteFrameProc

SWSetSpriteFrameAdvance

SWSetSpriteWorldMaxFPS

SWSetSpriteLocation

This function will set a Sprite's current and last known position to an absolute horizontal, and vertical coordinate.

```
void SWSetSpriteLocation(SpritePtr srcSpriteP,                     short
                           horizLoc,                                 short
                           vertLoc);
```

srcSpriteP	A Sprite to be moved.
horizLoc	A value indicating the absolute horizontal coordinate to which the Sprite's current and last known position will be set.
vertLoc	A value indicating the absolute vertical coordinate to which the Sprite's current and last known position will be set.

DESCRIPTION

The SWSetSpriteLocation function is used to set a Sprite’s current and last known position to an absolute horizontal, and vertical coordinate. Since the Sprite’s last known position is set as well, the Sprite will not be erased from wherever it was before this function was called. You will typically use this function before the animation starts or when introducing a new Sprite into the animation, to set the Sprite initial position.

If the Sprite has a movement routine installed, it will not be called by this function.

SPECIAL CONSIDERATIONS

Note that the horizLoc and vertLoc coordinates of the Sprite are local to the worldRect of the SpriteWorld, not the portRect of the “parent” window.

SWSetSpriteMoveBounds

This function will set a Sprite’s movement boundary rectangle.

```
void SWSetSpriteMoveBounds(SpritePtr srcSpriteP,                    Rect
                           *moveBoundsRect);
```

srcSpriteP	An existing Sprite.
moveBoundsRect	A rectangle describing the Sprite’s movement boundary.

DESCRIPTION

The SWSetSpriteMoveBounds function is used to specify a movement boundary rectangle for a Sprite. Enforcement of this movement boundary if left to the Sprite’s movement routine provided by you. You may want to use this rectangle as an area around which the Sprite might bounce, or wrap, or some other movement behavior.

SWSetSpriteMoveDelta

This function sets the values by which the Sprite’s current position should be offset.

```
void SWSetSpriteMoveDelta(SpritePtr srcSpriteP,                     short
                          horizDelta,                               short
                          vertDelta);
```

srcSpriteP	An existing Sprite.
horizDelta	A value by which the current horizontal position should be offset.
vertDelta	A value by which the current vertical position should be offset.

DESCRIPTION

The SWSetSpriteMoveDelta function is used to specify the values by which the Sprite’s current position should be offset. The horizontal and vertical deltas indicate the direction and distance the Sprite should be moved when the Sprite's moveProc is called by SWProcessSpriteWorld. It is the responsibility of the moveProc to actually add the values in srcSpriteP->horizMoveDelta and srcSpriteP->vertMoveDelta to the Sprite's position. If there is no moveProc (installed with SWSetSpriteMoveProc), the Sprite will not move.

SEE ALSO

SWSetSpriteMoveProc

SWSetSpriteMoveProc

This function sets a Sprite's movement routine to be called when the Sprite is automatically moved.

```
void SWSetSpriteMoveProc(SpritePtr srcSpriteP,
                            MoveProcPtr moveProc);
```

srcSpriteP	An existing Sprite.
moveProc	A movement routine to be called when the Sprite is moved.

DESCRIPTION

The SWSetSpriteMoveProc function is used to specify a routine to be called when the Sprite is automatically moved. The Sprite will be automatically moved when it is processed by the SWProcessSpriteWorld function.

A moveProc routine has the form:

```
SW_FUNC void MyMoveProc(SpritePtr srcSpritePtr)
```

In your moveProc, you can change the position of the Sprite by calling such routines as SWMoveSprite or SWOffsetSprite. You can also directly change the Rect: srcSpriteP—>destFrameRect; this would be faster, though only very slightly. For example, in order to add a Sprite's deltas to its position, you could use this code:

```
srcSpriteP->destFrameRect.right += srcSpriteP->horizMoveDelta;

srcSpriteP->destFrameRect.bottom += srcSpriteP->vertMoveDelta;

srcSpriteP->destFrameRect.left += srcSpriteP->horizMoveDelta;

srcSpriteP->destFrameRect.top += srcSpriteP->vertMoveDelta;

srcSpriteP->needsToBeDrawn = true;
```

Note the last line of this code. If you use your own code to move a Sprite, you must set the needsToBeDrawn flag yourself. If you use any of SpriteWorld's internal routines to move a Sprite, such as SWMoveSprite, SWOffsetSprite or SWSetSpriteLocation, this flag is set automatically by SpriteWorld. Also note that the code above should only be used to offset a Sprite's position, not to set it to an absolute location, since this will not work properly if the Sprite's current Frame has a hotSpot other than 0,0. To move a Sprite to an absolute location, use SWMoveSprite.

The Sprites are moved starting with the first Sprite in the first layer, proceeding to the last Sprite in the last layer. Since the first layer is the one that is also drawn first, the

Sprites that appear at the “bottom” of the animation (underneath all the other Sprites) are the ones that will be moved first. Usually, the order in which the Sprites are moved won’t matter to your application. However, if it is important that certain Sprites are moved before others, you may want to make your own routine to move them all; this routine can then be called either immediately before or after SWProcessSpriteWorld. If you do use such a routine for moving all the Sprites, then you will not want to install moveProcs for any of them.

SWSetSpriteMoveTime

This function sets the time interval between movements of the Sprite.

```
void SWSetSpriteMoveTime(SpritePtr srcSpriteP,                      long
                           timeInterval);
```

srcSpriteP	An existing Sprite.
timeInterval	A value indicating the millisecond time interval between movements of the Sprite.

DESCRIPTION

The SWSetSpriteMoveTime function is used to specify a millisecond time interval between movements of the Sprite. The Sprite will be automatically moved when it is processed by the SWProcessSpriteWorld function after the specified time interval has passed.

SPECIAL CONSIDERATIONS

Note that SWSetSpriteWorldMaxFPS places a limit on how often the animation will actually be rendered, and thus places a lower limit on the time interval between movements of the Sprite. To give SWSetSpriteMoveTime complete control over a Sprite’s movement, do not set the animation speed with SWSetSpriteWorldMaxFPS.

SEE ALSO

SWSetSpriteWorldMaxFPS

SWSetSpriteVisible

This function sets the visibility of a Sprite.

```
void SWSetSpriteVisible(SpritePtr srcSpriteP,                       Boolean
                           isVisible);
```

srcSpriteP	An existing Sprite.

isVisible	A Boolean value specifying the visible state of the Sprite.

DESCRIPTION

The SWSetSpriteVisible function is used to specify whether a Sprite that taking part in the animation should actually by drawn. This result of calling this function is reflected when the animation is rendered using SWAnimateSpriteWorld.

SWUnlockSprite

This function will unlock a Sprite .

```
void SWUnlockSprite(SpritePtr srcSpriteP);
```

srcSpriteP	A Sprite to be unlocked.

DESCRIPTION

The SWUnlockSprite function will unlock a Sprite , including all the Frames contained within. The Sprite must not be used for animation while in this unlocked state. If you wish to animate the Sprite again, you must first call SWLockSprite.

∆ WARNING ∆

This routine can be dangerous, as the Sprite will be unlock regardless of whether its Frames are used by more than one Sprite (as they will be if the Sprite is a clone).

SWUpdateSpriteFromPictResource

The function will redraw the internal graphic image of a Sprite, using a PICT resource.

```
OSErr SWUpdateSpriteFromPictResource(SpritePtr theSpriteP,          short
                                  pictResID);
```

theSpriteP	An existing Sprite to be updated.
pictResID	The resource id of a picture (‘PICT’) resource from which the first Frame of the Sprite was created, or the ID of a single, multi-frame picture.

DESCRIPTION

The SWUpdateSpriteFromPictResource function redraws the pict or picts of a Sprite’s Frames to the Sprite’s GWorld(s). While a SpriteWorld is being initialized and multiple Sprites are being created, you may want your application to display a “splash” graphic. If this graphic uses a different palette from the one used by the Sprites, The color of the Sprites will be mapped incorrectly, because the screen GDHandle and its clut are also used by SpriteWorld's GWorlds. Once the initialization is done and the application's palette is correctly set for the Sprites' graphics, you can quickly redraw the Sprites' picts to their GWorlds with SWUpdateSpriteFromPictResource. This routine might also be used to restore Sprites to their original appearance after they have been modified by drawing directly to the Sprite's GWorlds.

SWWrapSprite

This function can be used as part of a Sprite's movement routine to make the Sprite wrap from one side of the screen to the other.

```
Boolean SWWrapSprite(SpritePtr srcSpriteP);
```

srcSpriteP	A Sprite being moved.

DESCRIPTION

This function will check to see if a Sprite has gone entirely outside of its moveBoundsRect, and if so, it will wrap it to the other side. When the Sprite is wrapped, the old position of the Sprite is not erased, so it is important that when the Sprite moves outside its moveBoundsRect that it is not visible on the screen. A Boolean value is returned indicating whether the Sprite was wrapped or not; it will be true if it was wrapped, false if it was not.

This function can be used as part of a moveProc; the simplest example being:

```
SW_FUNC void MySpriteMoveProc(SpritePtr spriteP)
{
      Boolean     wasWrapped;
      SWOffsetSprite(spriteP, spriteP->horizMoveDelta,
            spriteP->vertMoveDelta);
      wasWrapped = SWWrapSprite(spriteP);
}
```

SEE ALSO

SWSetSpriteMoveBounds

SWBounceSprite

Frame Routines

SWCopyFrame

This function will create a new Frame and copy the image from the old Frame into it.

```
OSErr SWCopyFrame(SpriteWorldPtr destSpriteWorldP,
                  FramePtr oldFrameP,
                  FramePtr *newFrameP,                              Boolean
                copyMasks)
```

`destSpriteWorldP`	The SpriteWorld that will ultimately contain the Frame. This is passed so that the same GDevice used elsewhere in the SpriteWorld can be used to create the new Frame.
`oldFrameP`	The Frame that will be copied.
`newFrameP`	Receives a pointer to the new Frame.
`copyMasks`	Specifies whether you want to copy the masks from the old Frame to the new Frame.

DESCRIPTION

SWCopyFrame creates a new Frame the same size as the old Frame and copies the old Frame's image into the new Frame, allowing you to modify the new Frame's image without changing the original version. If you don't need to modify the image, but simply want to add the same image to several Sprites, you should use SWAddFrame or SWInsertFrame, not SWCopyFrame.

The destSpriteWorldP specifies the SpriteWorld that the new Frame will end up being used in. The oldFrameP is the Frame to be copied, and the newFrameP variable receives a pointer to the new Frame once it is created. The same type of mask that was in the original Frame is created for the new Frame, so if the original Frame had only a pixel mask, then only a pixel mask will be created for the new Frame.

The copyMasks parameter allows you to specify whether the mask image should be copied from the old Frame into the new Frame. If you pass a value of false, the new Frame's mask will still be created, but will remain uninitialized, possibly containing random data. You might want to do this if you are going to change the new Frame's mask anyway, so having SWCopyFrame copy the old mask would be a waste of time, since you would be replacing it. One example of this would be if you were to change the Frame's image and then call SWUpdateFrameMasks to update the mask for the new image. However, if you are not going to be replacing the Frame's mask, you should pass true for the copyMasks parameter.

Make sure to lock the new Frame before using it, either by calling SWLockFrame(newFrameP), or by adding the Frame to a Sprite and then locking the Sprite.

SWCreateFrame

This function will create a new Frame structure.

```
OSErr SWCreateFrame(GDHandle theGDH,
                    FramePtr* newFrameP,                            Rect*
                  frameRect,                                        short
                  depth)
```

theGDH	The GDH to be used when creating the Frame's GWorld.
newFrameP	A FramePtr variable that receives the address of the new Frame.
frameRect	The dimensions of the Frame.
depth	The depth of the Frame.

DESCRIPTION

This function creates a new Frame structure containing a GWorld the size of the frameRect parameter. No masks are created, so this function is ideal for creating extra offscreen buffers, such as an offscreen area used to display the "stats" portion of your game. TheGDH parameter specifies which GDevice should be used when creating the GWorld for this Frame. Normally, you would simply use the GDevice of your SpriteWorld (spriteWorldP->mainSWGDH). Here's an example that creates a new 8-bit Frame (256 colors) that is 200x400 pixels large:

SetRect(myRect, 0, 0, 200, 400);

err = SWCreateFrame(spriteWorldP->mainSWGDH, &newFrameP, &myRect, 8);

An error code is returned if the function fails. To access the Frame's image, use this:

SetGWorld(newFrameP->framePort, nil);

You can then draw into the Frame's GWorld as you would any other GWorld. However, you must make sure to lock the Frame first by calling SWLockFrame(newFrameP). The Frame's dimensions are stored in the Frame's frameRect variable. You can also use SpriteWorld's blitters to copy between Frames once the Frames are locked. Here's an example that copies the contents of our Frame into the SpriteWorld's work Frame:

BlitPixie8BitRectDrawProc(newFrameP, spriteWorldP->workFrameP, newFrameP->frameRect, newFrameP->frameRect);

For more information on calling SpriteWorld's blitters directly, or for information about how to set up the "stats" area of your game, see the SpriteWorld Tips and Tricks document.

SWCreateWindowFrame

This function will create a new Frame structure for use with a window.

```
OSErr SWCreateWindowFrame(FramePtr* newFrameP,                     Rect*
                          frameRect,                               short
                          maxHeight)
```

newFrameP	A FramePtr variable that receives the address of the new Frame.
frameRect	The dimensions of the Frame.
maxHeight	Useful for allowing the windowFrame to be expanded later. See below.

DESCRIPTION

This function creates a new Frame structure for use with a window. This does not create a new GWorld; the Frame structure is simply used to store information about the window that is used by SpriteWorld. Make sure to set the port to your window before calling this function, as the Frame is created using information about the current port.

Normally you won't need to call this function yourself, since SWCreateSpriteWorldFromWindow calls it for you, but this could be useful if you wanted to set up a stats area of a window, and want to be able to use SpriteWorld's blitters to draw to that area. By calling SWCreateWindowFrame, you can create a Frame structure that can be passed to SpriteWorld's blitters, allowing you to use those blitters to draw in the stats area of your screen.

Simply pass the address of an unused FramePtr variable as the newFrameP parameter. The frameRect parameter specifies the rectangle, in coordinates local to your window, of your windowFrame.

The maxHeight parameter is typically used internally by SpriteWorld, but can be useful to you as well in some situations. It allows you to specify the maximum height of the windowFrame, should it be resized with SWWindowFrameMoved later. (Normally, you can't make it larger than its original size.) If you plan on making the windowFrame taller than its initial height later on in the program, you can specify the maximum height you'll ever need to make it with this parameter. Otherwise, just pass 0 as the maxHeight parameter. The window's width can be resized to any size later, regardless of what you pass as the maxHeight parameter.

Do not use SWLockFrame to lock a window Frame. Use SWLockWindowFrame instead. (And make sure to lock it before using it!) When you are done with the window Frame, call SWDisposeWindowFrame, not SWDisposeFrame.

SEE ALSO

SWWindowFrameMoved

SWDisposeFrame

This function will dispose of an existing Frame, releasing the memory it occupies.

```
void SWDisposeFrame(FramePtr *deadFrameP);
```

deadFrameP	A Frame to be disposed.

DESCRIPTION

The SWDisposeFrame function will dispose of a Frame. Frames are used to hold the images for Sprites, Tiles, and the background and work areas of a SpriteWorld.

SEE ALSO

SWDisposeSprite

SWDisposeWindowFrame

This function will dispose of an existing window Frame, releasing the memory it occupies.

```
void SWDisposeWindowFrame(FramePtr *deadFrameP);
```

deadFrameP	A windowFrame to be disposed.

DESCRIPTION

When disposing a window Frame, call this function, not SWDisposeFrame.

SWLockFrame

This functions locks a Frame in preparation for animation.

```
void SWLockFrame(FramePtr srcFrameP);
```

srcFrameP	A Frame to be locked.

DESCRIPTION

Few programmers will have reason to use this routine. You can lock all of a Sprite’s Frames with SWLockSprite, or all the Layers, Sprites and Frames in a SpriteWorld with SWLockSpriteWorld.

SWLockWindowFrame

This functions locks a Frame created with SWCreateWindowFrame.

```
void SWLockWindowFrame(FramePtr windowFrameP);
```

windowFrameP	A windowFrame to be locked.

DESCRIPTION

Call this function to lock a Frame created with SWCreateWindowFrame.

SWSetFrameHotSpot

This functions sets a Frame's hotSpot.

```
void SWSetFrameHotSpot(FramePtr srcFrameP,                         short
                       hotSpotH,                                   short
                       hotSpotV)
```

srcFrameP	The Frame whose hotSpot you want to set.
hotSpotH	The horizontal coordinate of the hotSpot.

hotSpotV	The vertical coordinate of the hotSpot.

This function sets a Frame's "hotSpot", which is similar in many respects to a cursor's hotSpot. A Frame's hotSpot determines how a Sprite will be positioned when that Frame is the Sprite's current Frame. Normally, functions such as SWMoveSprite or SWSetSpriteLocation position the Sprite's top-left corner over the specified row and column. However, when a hotSpot is used, the hotSpot is positioned over the specified row and column. For instance, if the hotSpot for the Sprite's current Frame is 25,25, and the Sprite is moved to 100,100, the Sprite's top-left corner will be at 75,75, placing the Frame's hotSpot directly over 100,100. By default, each Frame's hotSpot is 0,0.

HotSpots are particularly useful for when a Sprite has Frames of differing sizes, since you can use hotSpots to specify in which directions the Frame should expand or shrink, rather than having everything relative to the Sprite's top-left corner. Normally, if a Sprite needs hotSpots, you would cycle through all of a Sprite's Frames right after creating the Sprite, calling SWSetFrameHotSpot to set the hotSpot for each Frame.

Important: When you look at a Sprite's destFrameRect or oldFrameRect, you must remember that these rects have had the hotSpot for the Sprite's current Frame subtracted from them. For instance, if you move a 50x50 Sprite to 100,100, and the current Frame's hotSpot is 25,25, the destFrameRect will be set to (75, 75, 125, 125). If you use the top-left corner of this rect in a subsequent call to SWMoveSprite, you will run into problems, because SWMoveSprite moves the Sprite's hotSpot to the specified location, so a call such as this would move the Sprite's hotSpot to its current top-left location:

```
SWMoveSprite(&mySpriteP, mySpriteP->destFrameRect.left,
             mySpriteP->destFrameRect.top);
```

If you want to know where the Sprite's hotSpot is on the screen, you can add the current Frame's hotSpot to the destFrameRect's top-left corner:

```
Point hotSpot;

hotSpot.h = mySpriteP->destFrameRect.left + mySpriteP->curFrameP->hotSpotH;
hotSpot.v = mySpriteP->destFrameRect.top + mySpriteP->curFrameP->hotSpotV;
```

This point can then be used with SWMoveSprite, since that function moves a Sprite's hotSpot (not the top-left corner) to the specified row and column. For example, this would move the Sprite 5 pixels to the right of where it was previously:

```
SWMoveSprite(mySpriteP, hotSpot.h + 5, hotSpot.v);
```

This is equivalent to calling SWOffsetSprite(mySprite, 5, 0). No conversion is necessary when using SWOffsetSprite, since rather than specifying where the hotSpot is to be moved to, you simply specify how far the Sprite should be moved from its previous position.

SWUnlockFrame

This functions unlocks a Frame.

```
void SWUnlockFrame(FramePtr srcFrameP);
```

srcFrameP	A Frame to be unlocked.

DESCRIPTION

This function unlocks a Frame; the various Handles in the Frame data structure, including the GWorld pixMapHandle are unlocked. The Frame must not be used for animation while in this unlocked state. If you wish to animate the Sprite using the Frame again, you must first call SWLockFrame or SWLockSprite.

SPECIAL CONSIDERATIONS

Few programmers will have reason to use this routine, as there is rarely any reason to unlock a single Frame.

∆ WARNING ∆

This routine can be dangerous, as the Frame will be unlocked regardless of whether it is used by more than one Sprite (as it will be if the Sprite is a clone), or whether the Frame's GWorld is used by more than one Frame (as it will be if the Sprite was created with SWCreateSpriteFromSinglePict).

SWUnlockWindowFrame

This function unlocks a Frame created with SWCreateWindowFrame.

```
void SWUnlockWindowFrame(FramePtr windowFrameP);
```

windowFrameP	A windowFrame to be unlocked.

DESCRIPTION

Call this function to unlock a Frame created with SWCreateWindowFrame.

SWUpdateFrameMasks

Updates the masks of a Frame based on the Frame's current image.

```
OSErr SWUpdateFrameMasks(SpriteWorldPtr destSpriteWorldP,
                         FramePtr srcFrameP)
```

destSpriteWorldP	The SpriteWorld the Frame will be used in.
srcFrameP	The Frame to be updated.

DESCRIPTION

The SWUpdateFrameMasks function updates any masks that are a part of that Frame, using the Frame's image to determine what the mask should be. That is, all non-white pixels in the Frame's framePort will be turned into the Sprite's mask. This allows you to modify the image in a Frame's framePort and have the mask reflect your changes. However, this function is rather slow, so you should avoid using it while the animation is running. A faster method is to modify the Sprite's mask directly (if you are using a pixelMask). See the SpriteWorld Tips and Tricks file for more information.

SWWindowFrameMoved

This function updates the data used by SpriteWorld’s direct-to-screen blitter.

```
void SWWindowFrameMoved(FramePtr windowFrameP,                      Rect
                      *frameRect);
```

windowFrameP	The window frame that was moved.
frameRect	The size and location of the windowFrame in the window

DESCRIPTION

If you have created a WindowFrame from a movable window, and use one of SpriteWorld's blitters to update the contents of that WindowFrame, then you should call SWWindowFrameMoved to update the blitter information for that WindowFrame whenever its window is moved.

This function is essentially the same as SWWindowMoved, except that you must provide an additional parameter: you must pass a frameRect to the function specifying the windowFrame's size and location in coordinates local to the window. This rect is the same as the rect you pass to SWCreateWindowFrame when you first created the window Frame that is now being moved.

You can also use SWWindowFrameMoved to change the location of the windowFrame within the window, even if the window itself hasn't moved. You may also change its size, making the windowFrame's width longer or shorter, and you may also make the windowFrame's height shorter than it was when it was first created, but you may not make it taller than it was originally unless you passed a maxHeight parameter allowing you to do this to SWCreateWindowFrame when first creating the windowFrame. See the documentation for SWCreateWindowFrame for more information on the maxHeight parameter.

Utility, Sprite Compiler and Miscellaneous Routines

SWAnimate8BitStarField

This function will erase the stars from their old positions on the screen and draw them in their new positions.

```
void SWAnimate8BitStarField(SpriteWorldPtr spriteWorldP,
                                  StarArray *starArray,                        short
                                numStars,
                                  unsigned short backColor);
```

`spriteWorldP`	The SpriteWorld in which the stars should be drawn.
`starArray`	The array containing the star data.
`numStars`	The number of stars in the starArray.
`backColor`	The color index of the background (usually black).

DESCRIPTION

This function provides a fast and easy way to add a moving star field background to your non-scrolling animation, such as those used in games like Space Junkie, Solarian II, and Swoop. This function is quite fast since the stars are drawn only once, directly to the screen.

This function avoids drawing stars on top of any Sprites in the SpriteWorld, so the Sprites will appear to be in front of the star field. To accomplish this, the routine checks to make sure that the pixel below each star is the same as backColor before the star is drawn. This ensures that the star is not drawn on top of a Sprite. This also means that the stars will not be drawn over **anything** that is not the background color, whether a Sprite or not.

The StarArray is a structure that is defined in BlitPixie.h:

```
typedef struct StarArray
{
short      horizLoc;          // Current horizontal position of the star
short      vertLoc;           // Current vertical position of the star
short      oldHorizLoc;       // Horizontal position of star the previous frame
short      oldVertLoc;        // Vertical position of star the previous frame
short      horizSpeed;        // To be used by the user to move the star
short      vertSpeed;         // To be used by the user to move the star
unsigned short color;         // Current color of the star
Boolean    needsToBeErased;   // If drawn last frame, then it needs to be erased.
short      userData           // Reserved for user
}StarArray;
```

You should define your StarArray like this:

```
StarArray          myStarArray[kNumStars];
```

You need to set up all of the variables in the StarArray before the animation starts (to set each star’s starting position, color, speed, etc.). Then you can animate the stars by putting a statement like this in the main loop of your animation:

```
SWAnimate8BitStarField(spriteWorldP, myStarArray, kNumStars, kBackColor);
```

It is up to you to move each star by changing its horizLoc and vertLoc. However, before you move each star, you must save its old position like so:

```
myStarArray[starNumber].oldHorizLoc = myStarArray[starNumber].horizLoc;
myStarArray[starNumber].oldVertLoc = myStarArray[starNumber].vertLoc;
```

It is important to do this, because SWAnimate8BitStarField erases each star from its oldHorizLoc and oldVertLoc before it draws it in its new horizLoc and vertLoc.

The horizSpeed and vertSpeed variables are provided for your use so that you can make different stars move different speeds. It is up to you to add these values to the horizLoc and vertLoc of the star. However, you may not always want to use these variables to keep track of the star's speed. Such would be the case in a game like Lunatic Fringe or Space Madness, where all the stars move the same speed and direction, and the speed and direction of the stars change depending on where the ship is going.

The color variable is the color index of the star. Since SWAnimate8BitStarField is only for use in 8-bit mode, you may use any value between 0 and 255 for the color field. Normally, 0 will be white and 255 will be black, and the numbers in between are the various colors in your palette. You would generally set the color of the star before the animation starts and then leave it that way, although you can change the color of the star while the animation is running if you want.

The needsToBeErased field is used internally by SpriteWorld to keep track of whether the star was drawn the previous frame, and therefore whether the star needs to be erased or not. The only time you should use this variable is before the animation starts, when you should set it to false, since the star hasn't been drawn yet.

The last parameter to the function, backColor, indicates the color index of the background. This color is used to erase the stars. Also, each star will not be drawn unless the destination pixel is already the same color as the backColor, to ensure that the stars are not drawn on top of sprites. (SpriteWorld actually checks the destination pixel of the work area, instead of the screen, since reading directly from VRAM can slow down the next write to VRAM on some systems. Since the work area is a mirror of what is on the screen, SpriteWorld can use it to determine whether the star will be drawn on top of a Sprite or not.) If your background is black, then you should pass 255 as the backColor.

BlitPixie8BitFlipSprite

This function horizontally flips the Frames of an 8 bit Sprite.

```
void BlitPixie8BitFlipSprite(SpritePtr srcSpriteP);
```

srcSpriteP	The Sprite to be flipped.

DESCRIPTION

BlitPixie8BitFlipSprite “flips” a Sprite's Frame images horizontally, so that a right-facing character faces left, and vice-versa. This function assumes (and does not check for) 8 bit pixels.

SPECIAL CONSIDERATIONS

• BlitPixie8BitFlipSprite requires that the Sprite have a pixel mask, or no mask at all. It may have a region mask as well, and the region mask will be flipped correctly if both types of mask are present, but if there is **only** a region mask, the mask will not be flipped, and the flipped Sprite will not be drawn correctly by CopyBits. If there is only a pixel mask, or no mask, then BlitPixie8BitFlipSprite is reasonably fast, and can probably be used in mid-animation without a noticeable pause. If a region mask is present, the routine will take considerably longer.

• Note that because cloned Sprites share the same Frame images, if you flip one clone Sprite, all members of the clone "family" will also be flipped. As a rule, flipping won't be appropriate for cloned Sprites.

• Compiled Sprites must be compiled again after flipping.

• The Sprite must be locked before calling BlitPixie8BitFlipSprite.

BlitPixie8BitGetPixel

This function will return the 8 bit pixel value at a given point in a given Frame.

```
unsigned char BlitPixie8BitGetPixel(FramePtr srcFrameP,             Point
                                    thePoint);
```

srcFrameP	The Frame to get the pixel value from.
thePoint	The Point location of the pixel.

DESCRIPTION

BlitPixie8BitGetPixel provides a fast way to determine the color index value located at a given point in a Frame. The Frame can be a Sprite Frame, or the SpriteWorld's backFrame, workFrame, or windowFrame. This function assumes (and does not check for) 8 bit pixels.

SPECIAL CONSIDERATIONS

If the Frame specified is the windowFrame (mySpriteWorldP->windowFrameP), the pixel will be read directly from the screen. Due to caching issues, the act of reading from VRAM (the screen) can significantly slow down the next write to VRAM on some systems.

SEE ALSO

BlitPixie8BitSetPixel

BlitPixie8BitSetPixel

This function will set a single pixel in a given Frame value to an 8 bit color index value.

```
void BlitPixie8BitSetPixel(FramePtr srcFrameP,                      Point
                                    thePoint,
                                      unsigned char theColor);
```

srcFrameP	The Frame to draw the pixel in.
thePoint	The Point location for the pixel.
theColor	The 8 bit color index value.

DESCRIPTION

BlitPixie8BitSetPixel provides a fast way to set an individual pixel to a particular color index value. The Frame can be a Sprite Frame, or the SpriteWorld’s backFrame, workFrame, or windowFrame. This function will check whether thePoint is within the frame’s frameRect, and do nothing if it is not. It assumes (but does not check for) 8 bit pixels.

SEE ALSO

BlitPixie8BitGetPixel

SWClearStickyError

This function clears SpriteWorld’s sticky error value.

```
void SWClearStickyError(void);
```

DESCRIPTION

The SpriteWorld sticky error value holds the first non zero error result of any SpriteWorld function that has been called since the sticky error value was cleared with SWClearStickyError. SWClearStickyError is called by SpriteWorld when you create a SpriteWorld with SWCreateSpriteWorldFromWindow.

SEE ALSO

`SWStickyError`

SWCompileSprite

This function “compiles” a Sprite’s mask image, allowing the Sprite to be drawn using `CompiledSprite8BitDrawProc`.

```
OSErr SWCompileSprite(SpritePtr srcSpriteP);
```

srcSpriteP	A Sprite to be compiled.

DESCRIPTION

SWCompileSprite uses the mask image of an 8-bit Sprite to create a series of 68K machine language instructions. This package of instructions is then attached to the Frame data structure of each Frame of the Sprite. When the Sprite is drawn using CompiledSprite8BitDrawProc, these instructions are executed, accessing the Frame’s image and quickly blitting only the masked, or non-transparent, parts of the image to the destination work frame. Drawing Sprites with CompiledSprite8BitDrawProc is notably faster on 68K Macs than BlitPixie8BitMaskDrawProc.

An error code is returned if the bit depth of the SpriteWorld is not 8 bits, if the Sprite was not created using kPixelMask or kFatMask, or if memory allocation fails.

SEE ALSO

SWSetSpriteDrawProc

SWSetCleanUpSpriteWorld

This function keeps track of a SpriteWorld and removes its VBL-syncing task in the case of an assertion failure or unexpected error.

```
void SWSetCleanUpSpriteWorld(SpriteWorldPtr spriteWorldP);
```

`spriteWorldP`	The SpriteWorld that has VBL-syncing turned on and needs to be cleaned up should an unexpected error occur.

DESCRIPTION

Whenever SpriteWorld detects an assertion failure, it reports the error and quits. This could cause a crash if a SpriteWorld had the syncToVBL option turned on when this happened. This function keeps track of one SpriteWorld that needs to be cleaned up should an assertion failure occur. SpriteWorld will then automatically remove the VBL task from this SpriteWorld if an assertion occurs.

The SpriteWorld specified by this function will also be cleaned up if the FatalError() routine included in SWApplication.c detects an error before FatalError() makes the program quit.

There is currently no way to flag more than one SpriteWorld at a time as needing to be cleaned up should an unexpected error occur. Generally, you shouldn't need more than one SpriteWorld at a time that has the VBL-syncing option turned on, so this shouldn't be a problem.

SWSetStickyIfError

This function set the contents of SpriteWorld’s sticky error value if and only if the passed value is not zero.

```
void SWSetStickyIfError(OSErr errNum);
```

`errNum`	The error value.

DESCRIPTION

Primarily intended as an internal routine, SWSetStickIfError sets SpriteWorld’s sticky error value. If the value passed in errNum is zero, the contents of the sticky error are not changed.

SEE ALSO

SWClearStickyError

SWStickyError

SWSetTransparentColor

This function changes the transparent color that is used when loading self-masking sprites.

```
void SWSetTransparentColor(const RGBColor *theColor);
```

`theColor`	An RGBColor indicating the new transparent color.

DESCRIPTION

When a Sprite is loaded and a separate mask is not provided, the mask is generated based on the Sprite's image: all non-white pixels are converted into part of the Sprite's mask, while white pixels remain transparent. However, by calling this function, you can change the transparent color to anything, so that all pixels not matching your new color are converted into the Sprite's mask when the mask is created. This allows you to use the color white as part of your Sprite's image, and use a different color to indicate transparent sections.

For instance, to load a Sprite where all black pixels should be transparent, you would do this before making the call to load the Sprite:

```
RGBColor    myColor;

myColor.red = myColor.green = myColor.blue = 0x0000;

SWSetTransparentColor(&myColor);
```

Once you have changed the transparent color, <u>it remains changed until you change it again</u>! This means that if you change the transparent color to a non-white value for one Sprite, you must change it back to white again if you want to load a self-masking Sprite whose transparent portions are white. This is done by passing an RGBColor with red, green, and blue components of 0xFFFF to this function.

Using a transparent color other than white adds an additional delay to the loading process, since SpriteWorld must not only load the Sprite's image and create the mask, but it must also change all pixels matching the transparent color to white, in order to make the image compatible with SpriteWorld's blitters, which require all transparent parts of a Sprite's image to be white in order to work properly.

SWStickyError

This function returns the contents of SpriteWorld’s sticky error value.

```
OSErr SWStickyError(void);
```

DESCRIPTION

The sticky error value holds the first non zero error result of any SpriteWorld function that has been called since the sticky error value was cleared with SWClearStickyError. SWStickyError will inform you if an error was returned by any of the SpriteWorld routines you have called since the sticky error value was cleared. This allows you to make a series of SpriteWorld calls without checking each one individually to see if it returned an error. Whenever **any** SpriteWorld function results in an error, the error number will be preserved by SpriteWorld until it is cleared by SWClearStickyError (or overwritten by some subsequent non-zero error result).

It's generally a good idea to call this function just before starting your animation to make sure no errors occurred during setup, even if you usually check the error codes returned by each function, since it's possible you missed a function that returns an error code. (For instance, the various SWSetDrawProc functions return error codes, but most people never check them.)

SEE ALSO

SWClearStickyError

SpriteWorld Data Structures, etc.

```
///-----------------------------------------------------------------------------------------
//           sprite world error constants
///-----------------------------------------------------------------------------------------

enum
{
kSystemTooOldErr = 100,      // < System 7.0
kMaxFramesErr,               // attempt to exceed maximum number of frames for a sprite
kInvalidFramesIndexErr,      // frame index out of range
kNotCWindowErr,              // attempt to make a SpriteWorld from non-color WindowPtr
kNilParameterErr,            // nil SpritePtr, FramePtr, etc.
kWrongDepthErr,   // 105     // invalid pixel size for attempted function
kWrongMaskErr,               // invalid mask type for attempted function
kOutOfRangeErr,              // tileID, tileMap, or other value out of bounds
kTilingNotInitialized,       // tiling hasn't been initialized
kTilingAlreadyInitialized,   // tiling already initialized; can't be initialized again
kNullTileMapErr,  // 110     // no TileMap has ever been created/loaded
kTileMapNotLockedErr,        // the TileMap is not locked and can't be used until it is
kAlreadyCalledErr,           // the function was already called and can't be called again
kSpriteNotCompiledErr        // the sprite must be compiled before drawProc can be set
kBadParameterErr, // 114     // a parameter that was passed to the function is invalid
kSpriteAlreadyInLayer,       // the Sprite is already in a Layer, and can't be added again
kNilFrameErr,                // the Frame this function acts on is NIL
kNotLockedErr                // a structure that must be locked is not
};


///-----------------------------------------------------------------------------------------
//           sprite world data structure
///-----------------------------------------------------------------------------------------

struct SpriteWorldRec
{
SpriteLayerPtr headSpriteLayerP;  // head of the sprite layer linked list
SpriteLayerPtr tailSpriteLayerP;  // tail of the sprite layer linked list
SpriteLayerPtr deadSpriteLayerP;  // where SWRemoveSpriteFromAnimation puts Sprites

UpdateRectStructPtr headUpdateRectP; // used by SWFlagRectAsChanged

FramePtr extraBackFrameP;         // used when adding a background behind tiles
FramePtr backFrameP;              // frame for the background
FramePtr workFrameP;              // work, or "mixing" frame
FramePtr windowFrameP;            // frame for drawing to the screen

DrawProcPtr offscreenDrawProc;    // callback for erasing sprites offscreen
DrawProcPtr screenDrawProc;       // callback for drawing sprite pieces onscreen
DoubleDrawProcPtr doubleRectDrawProc; // callback for updating screen when scrolling

CallBackPtr postEraseCallBack;    // called after erasing sprites
CallBackPtr postDrawCallBack;     // called after drawing sprites

Rect    windRect;                 // holds windowFrameP->frameRect for easier access
Rect    backRect;                 // holds backFrameP->frameRect for easier access
Rect    originalWindRect;         // used by SWRestoreWindRect
Rect    originalBackRect;         // used by SWRestoreWindRect

Rect    visScrollRect;            // rect that is copied to screen when scrolling
Rect    oldVisScrollRect;         // visScrollRect from last frame
Rect    offscreenScrollRect;      // same as visScrollRect, but local to offscreen
short   horizScrollRectOffset;    // offset from offscreenScrollRect to visScrollRect
short   vertScrollRectOffset;     // offset from offscreenScrollRect to visScrollRect
short   horizScrollDelta;         // horizontal scrolling delta
short   vertScrollDelta;          // vertical scrolling delta
Rect    scrollRectMoveBounds;     // move bounds for visScrollRect

WorldMoveProcPtr worldMoveProc;   // pointer to the scrolling world move procedure
SpritePtr    followSpriteP;       // pointer to the "follow sprite", or NULL

TileMapStructPtr *tileLayerArray;       // an array of all the tileMap layers
short        lastActiveTileLayer;       // the last active tile layer
Boolean      tilingIsInitialized;       // has the tiling been initialized yet?
Boolean      tilingIsOn;          // are the tiling routines turned on?
short        **tilingCache;       // two-dimensional tiling cache
short        numTilingCacheRows; // number of rows in tilingCache array
short        numTilingCacheCols; // number of cols in tilingCache array
FramePtr     *tileFrameArray;     // array of tile framePtrs
short        *curTileImage;       // array specifying the current frame of each tile
short        maxNumTiles;         // number of elements in tileFrameArray
short        tileWidth;           // width of each tile
short        tileHeight;          // height of each tile
long         numTilesChanged;     // number of rects in changedTiles array to update
Rect         *changedTiles;       // array of rects of tiles that changed
long         changedTilesArraySize;// number of elements in changedTiles array
TileChangeProcPtr tileChangeProc;// pointer to tile frame changing procedure
TileRectDrawProcPtr tileRectDrawProc; // pointer to the function that draws tiles in a rect
DrawProcPtr  tileMaskDrawProc;    // drawProc for drawing masked tiles in tile layers
DrawProcPtr  partialMaskDrawProc;       // drawProc for drawing partialMask tiles above
sprites

GDHandle     mainSWGDH;           // GDH of SpriteWorld's window
short        pixelDepth;          // SpriteWorld's depth

short        fpsTimeInterval;     // milliseconds per frame of animation (1000/fps)
unsigned long runningTimeCount;   // running total time in milliseconds
UnsignedWide lastMicroseconds;    // value of previous Microseconds() call
unsigned long timeOfLastFrame;    // time (from runningTimeCount) of last frame
VBLTaskRec   vblTaskRec;          // extended VBLTask record
Boolean      usingVBL;            // is the VBL task installed?
Boolean      frameHasOccurred;    // Has the SpriteWorld been processed?

short        pad1;

long         userData;            // reserved for user
```

```
};


///--------------------------------------------------------------------------------------
//            sprite layer data structure
///--------------------------------------------------------------------------------------

struct SpriteLayerRec
{
SpriteLayerPtr nextSpriteLayerP; // next sprite layer
SpriteLayerPtr prevSpriteLayerP; // previous sprite layer

SpritePtr     headSpriteP;        // head of sprite linked list
SpritePtr     tailSpriteP;        // tail of sprite linked list

long          userData;           // reserved for user
short         tileLayer;          // the tile layer this sprite layer is under
Boolean       isPaused;           // does this layer get processed by SWProcessSpriteWorld?
};


///--------------------------------------------------------------------------------------
//            sprite data structure
///--------------------------------------------------------------------------------------

struct SpriteRec
{
SpriteLayerPtr parentSpriteLayerP;      // the sprite layer this sprite is currently in
SpritePtr     nextSpriteP;        // next sprite in that layer
SpritePtr     prevSpriteP;        // previous sprite in that layer
SpritePtr     nextActiveSpriteP;  // next active sprite in the SpriteWorld
SpritePtr     nextIdleSpriteP;    // next idle sprite in the SpriteWorld

Boolean       isVisible;          // draw the sprite?
Boolean       needsToBeDrawn;     // sprite has changed, needs to be drawn
Boolean       needsToBeErased;    // sprite needs to be erased onscreen
char          pad1;
Rect          destFrameRect;      // frame destination rectangle
Rect          oldFrameRect;       // last frame destination rectangle
Rect          deltaFrameRect;     // union of the sprite's lastRect and curRect
DrawProcPtr   frameDrawProc;      // callback to draw sprite
short         tileDepth;          // the tile layers this sprite is under
short         pad2;

Rect          clippedSourceRect;  // source sprite frame rect after clipping
Rect          destOffscreenRect;  // sprite's dest rect after clipping and offset
Rect          oldOffscreenRect;   // the destOffscreenRect from the previous frame
Boolean       destRectIsVisible;  // is destOffscreenRect visible on screen?
Boolean       oldRectIsVisible;   // was oldOffscreenRect visible on screen?
char          pad3;

Boolean       frameArrayIsShared; // is the frameArray shared with another sprite?
FramePtr      *frameArray;        // array of frames
FramePtr      curFrameP;          // current frame
long          numFrames;          // number of frames
long          maxFrames;          // maximum number of frames
long          frameTimeInterval;  // time interval to advance frame
unsigned long timeOfLastFrameChange; // time (from runningTimeCount) frame was last changed
long          frameAdvance;       // amount to adjust frame index
long          curFrameIndex;      // current frame index
long          firstFrameIndex;    // first frame to advance
long         lastFrameIndex;     // last frame to advance
FrameProcPtr frameChangeProc;    // callback to change frames

long         moveTimeInterval;   // time interval to move sprite
unsigned long timeOfLastMove;    // time (from runningTimeCount) sprite was last moved
short        horizMoveDelta;     // horizontal movement delta
short        vertMoveDelta;      // vertical movement delta
Rect         moveBoundsRect;     // bounds of the sprite's movement
MoveProcPtr  spriteMoveProc;     // callback to handle movement

CollideProcPtr spriteCollideProc;// callback to handle collisions

GWorldPtr    sharedPictGWorld;   // if common GWorld used for frames, here it is
GWorldPtr    sharedMaskGWorld;   // same for mask
RemovalType  spriteRemoval;      // whether to remove sprite, and if so how
AdvanceType  frameAdvanceMode;   // mode of automatic frame advancement

unsigned long translucencyLevel; // used by translucent blitter
unsigned long rotation;          // used by rotation blitter
short        scaledWidth;        // used by scaling blitter
short        scaledHeight;       // used by scaling blitter

long         userData;           // reserved for user
};


///-------------------------------------------------------------------------------------
//           frame data structure
///-------------------------------------------------------------------------------------

struct FrameRec
{
GWorldPtr    framePort;          // GWorld for the frame image
PixMapHandle framePixHndl;       // handle to pix map (saved for unlocking/locking)
PixMapPtr    framePix;           // pointer color pix map (valid only while locked)

char*        frameBaseAddr;      // base address of pixel data (valid only while locked)
unsigned long frameRowBytes;     // number of bytes in a row of the frame
short        leftAlignFactor;    // used to align rect.left to the nearest long word
short        rightAlignFactor;   // used to align rect.right to the nearest long word
Boolean      isFrameLocked;      // has the frame been locked?
Boolean      isWindowFrame;      // is this a window frame?
short        pad2;

Rect         frameRect;          // source image rectangle
short        hotSpotH;           // horizontal hot point for this frame
short        hotSpotV;           // vertical hot point for this frame
Point        offsetPoint;        // image offset factor relative to destination rectangle
RgnHandle    maskRgn;            // image masking region
GWorldPtr    maskPort;           // GWorld for the mask image
PixMapHandle maskPixHndl;        // handle to pix map (saved for unlocking/locking)
PixMapPtr    maskPix;            // pointer to color pix map (valid only while locked)
char*        maskBaseAddr;       // base address of mask pixel data (valid while locked)
Boolean      tileMaskIsSolid;    // used by SWDrawTilesAboveSprite
Boolean      sharesGWorld;       // shares GWorld with other frames

unsigned short useCount;         // number of sprites using this frame
short        worldRectOffset;    // non-whole-byte offset for 1-bit blitter
unsigned short numScanLines;     // number of scan lines
unsigned long* scanLinePtrArray; // array of pointers to each scanline

PixelCodeHdl pixCodeH;           // handle to compiled sprite data
```

```
      BlitFuncPtr   frameBlitterP;      // procPtr to compiled sprite data

      long          userData;           // reserved for user
      };


      ///--------------------------------------------------------------------------------
      //            Star Array data structure
      ///--------------------------------------------------------------------------------

      typedef struct StarArray
      {
      short            horizLoc;        // Current horizontal position of the star
      short            vertLoc;         // Current vertical position of the star
      short            oldHorizLoc;     // Horizontal position of star the previous frame
      short            oldVertLoc;      // Vertical position of star the previous frame
      short            horizSpeed;      // To be used by the user to move the star
      short            vertSpeed;       // To be used by the user to move the star
      unsigned short   color;           // Current color of the star
      Boolean          needsToBeErased;        // If drawn last frame, it needs to be erased.
      short            userData;        // Reserved for user
} StarArray;
```

SpriteWorld Function Prototypes

You might find it useful to print out this section of Inside SpriteWorld, or to save it as a separate document, for use as a “quick reference guide”.

SpriteWorld Routines

```
void SWAddSpriteLayer(SpriteWorldPtr spriteWorldP,
                              SpriteLayerPtr spriteLayerP)

void SWAnimateSpriteWorld(SpriteWorldPtr spriteWorldP)

OSErr SWChangeWorldRect(SpriteWorldPtr spriteWorldP,            Rect*
newWorldRect,                 Boolean changeOffscreenAreas)

void SWCopyBackgroundToWorkArea(SpriteWorldPtr spriteWorldP)

OSErr SWCreateSpriteWorldFromWindow(
                              SpriteWorldPtr* spriteWorldP,
                              CWindowPtr srcWindowP,            Rect*
worldRect,                    Rect* backRect,                   short
maxDepth)

void SWDisposeSpriteWorld(SpriteWorldPtr *spriteWorldP)

OSErr SWEnterSpriteWorld(void)

void SWExitSpriteWorld(void)

OSErr SWFlagRectAsChanged(SpriteWorldPtr spriteWorldP,          Rect*
theChangedRect)

OSErr SWFlagScrollingRectAsChanged(SpriteWorldPtr spriteWorldP,   Rect*
theChangedRect)
```

```
SpriteLayerPtr SWGetNextSpriteLayer(SpriteWorldPtr spriteWorldP,
                                SpriteLayerPtr curSpriteLayerP)
unsigned long SWGetSpriteWorldVersion(void)
void SWLockSpriteWorld(SpriteWorldPtr spriteWorldP)
void SWProcessSpriteWorld(SpriteWorldPtr spriteWorldP)
void SWRemoveSpriteLayer(SpriteWorldPtr spriteWorldP,
                                SpriteLayerPtr spriteLayerP)
OSErr SWRestoreWorldRect(SpriteWorldPtr spriteWorldP)
void SWSetPortToBackground(SpriteWorldPtr spriteWorldP)
void SWSetPortToWindow(SpriteWorldPtr spriteWorldP)
void SWSetPortToWorkArea(SpriteWorldPtr spriteWorldP)
OSErr SWSetPostEraseCallBack(SpriteWorldPtr spriteWorldP,
                                CallBackPtr callBack)
OSErr SWSetPostDrawCallBack(SpriteWorldPtr spriteWorldP,
                                CallBackPtr callBack)
void SWSetSpriteWorldMaxFPS(SpriteWorldPtr spriteWorldP,          short
framesPerSec)
OSErr SWSetSpriteWorldOffscreenDrawProc(
                                SpriteWorldPtr spriteWorldP,
                                WorldDrawProcPtr offscreenProc)
OSErr SWSetSpriteWorldScreenDrawProc(
                                SpriteWorldPtr spriteWorldP,
                                DrawProcPtr screenDrawProc)
void SWSwapSpriteLayer(SpriteWorldPtr spriteWorldP,
                                SpriteLayerPtr srcSpriteLayerP,
                                SpriteLayerPtr dstSpriteLayerP)
OSErr SWSyncSpriteWorldToVBL(SpriteWorldPtr spriteWorldP,         Boolean
vblSyncOn)
void SWUnlockSpriteWorld(SpriteWorldPtr spriteWorldP)
void SWUpdateSpriteWorld(SpriteWorldPtr spriteWorldP,             Boolean
updateWindow)
void SWUpdateWindow(SpriteWorldPtr spriteWorldP)
void SWWindowMoved(SpriteWorldPtr spriteWorldP)
```

Layer Routines

```
OSErr SWAddSprite(SpriteLayerPtr spriteLayerP,
                                SpritePtr newSpriteP)
void SWCollideSpriteLayer(SpriteWorldPtr spriteWorldP,
                                SpriteLayerPtr srcSpriteLayerP,
                                SpriteLayerPtr dstSpriteLayerP)
short SWCountNumSpritesInLayer(SpriteLayerPtr srcSpriteLayerP)
OSErr SWCreateSpriteLayer(SpriteLayerPtr *spriteLayerP)
void SWDisposeAllSpritesInLayer(SpriteLayerPtr srcSpriteLayerP)
void SWDisposeSpriteLayer(SpriteLayerPtr *spriteLayerP)
```

```
SpritePtr SWFindSpriteByPoint(SpriteLayerPtr spriteLayerP,
                              SpritePtr startSpriteP,          Point
testPoint)
SpritePtr SWGetNextSprite(SpriteLayerPtr spriteLayerP,
                              SpritePtr curSpriteP)
OSErr SWInsertSpriteAfterSprite(SpritePtr newSpriteP,
                              SpritePtr dstSpriteP)
OSErr SWInsertSpriteBeforeSprite(SpritePtr newSpriteP,
                              SpritePtr dstSpriteP)
void SWLockSpriteLayer(SpriteLayerPtr spriteLayerP)
OSErr SWPauseSpriteLayer(SpriteLayerPtr spriteLayerP)
void SWRemoveAllSpritesFromLayer(SpriteLayerPtr srcSpriteLayerP)
OSErr SWRemoveSprite(SpritePtr oldSpriteP)
OSErr SWSwapSprite(SpritePtr srcSpriteP,
void SWUnlockSpriteLayer(SpriteLayerPtr spriteLayerP)
OSErr SWUnpauseSpriteLayer(SpriteLayerPtr spriteLayerP)
```

Sprite Routines

```
void SWAddFrame(SpritePtr srcSpriteP, FramePtr newFrameP)
Boolean SWBounceSprite(SpritePtr srcSpriteP)
OSErr SWCloneSprite(SpritePtr cloneSpriteP,
                              SpritePtr *newSpriteP,           void*
spriteStorageP)
OSErr SWCloneSpriteFromTile(SpriteWorldPtr spriteWorldP,
                              SpritePtr* newSpriteP,           void*
spriteStorageP,               short firstTileID,               short
lastTileID)
void SWCloseSprite(SpritePtr deadSpriteP)
OSErr SWCreateSprite(SpritePtr *newSpriteP,                    void*
spriteStorageP,               short maxFrames)
OSErr SWCreateSpriteFromCicnResource(
                              SpriteWorldPtr destSpriteWorld,
                              SpritePtr *newSpriteP,           void*
spriteStorageP,               short cIconID,                   short
maxFrames,                    short maskType)
OSErr SWCreateSpriteFromPictResource(
                              SpriteWorldPtr destSpriteWorld,
                              SpritePtr *newSpriteP,           void*
spriteStorageP,               short pictResID,                 short
maskResID,                    short maxFrames,                 short
maskType)
```

```
OSErr SWCreateSpriteFromSinglePict(
                                SpriteWorldPtr destSpriteWorld,
                                SpritePtr *newSpriteP,             void*
spriteStorageP,                 short pictResID,                   short
maskResID,                      short frameDimension,              short
borderWidth,                    short maskType)

OSErr SWCreateSpriteFromSinglePictXY(
                                SpriteWorldPtr destSpriteWorld,
                                SpritePtr *newSpriteP,             void*
spriteStorageP,                 short pictResID,                   short
maskResID,                      short frameWidth,                  short
frameHeight,                    short horizBorderWidth,            short
vertBorderHeight,               short maxFrames,                   short
maskType)

void SWDisposeSprite(SpritePtr *deadSpriteP)

OSErr SWFastCloneSprite(SpritePtr masterSpriteP,
                                SpritePtr *newSpriteP,             void*
spriteStorageP)

void SWInsertFrame(SpritePtr srcSpriteP, FramePtr newFrameP,       long
frameIndex)

Boolean SWIsPointInSprite(SpritePtr srcSpriteP,                    Point
testPoint)

Boolean SWIsSpriteInRect(SpritePtr srcSpriteP,                     Rect*
testRect)

Boolean SWIsSpriteFullyInRect(SpritePtr srcSpriteP,                Rect*
testRect)

void SWLockSprite(SpritePtr srcSpriteP)

void SWMoveSprite(SpritePtr srcSpriteP,                            short
horizLoc,                       short vertLoc)

void SWOffsetSprite(SpritePtr srcSpriteP,                          short
horizDelta,                     short vertDelta)

Boolean SWPixelCollision(SpritePtr srcSpriteP, dstSpriteP)

Boolean SWRadiusCollision(SpritePtr srcSpriteP, dstSpriteP)

Boolean SWRegionCollision(SpritePtr srcSpriteP, dstSpriteP)

void SWRemoveFrame(SpritePtr srcSpriteP, FramePtr oldFrameP)

void SWRemoveSpriteFromAnimation(SpriteWorldPtr spriteWorldP,
                                SpritePtr spriteP,                 Boolean
disposeOfSprite)

void SWSetCurrentFrame(SpritePtr srcSpriteP, FramePtr newFrameP)
void SWSetCurrentFrameIndex(SpritePtr srcSpriteP,                 short
frameIndex)

void SWSetSpriteCollideProc(SpritePtr srcSpriteP,
                                CollideProcPtr collideProc)

OSErr SWSetSpriteDrawProc(SpritePtr srcSpriteP,
                                DrawProcPtr drawProc)

void SWSetSpriteFrameAdvance(SpritePtr srcSpriteP,                short
frameAdvance)

void SWSetSpriteFrameAdvanceMode(SpritePtr srcSpriteP,
                                AdvanceType advanceMode)

void SWSetSpriteFrameProc(SpritePtr srcSpriteP,
                                FrameProcPtr frameProc)

void SWSetSpriteFrameRange(SpritePtr srcSpriteP,                  short
firstFrameIndex,                short lastFrameIndex)

void SWSetSpriteFrameTime(SpritePtr srcSpriteP,                   long
timeInterval)

void SWSetSpriteLocation(SpritePtr srcSpriteP,                    short
horizLoc,                       short vertLoc)

void SWSetSpriteMoveBounds(SpritePtr srcSpriteP,                  Rect
*moveBoundsRect)

void SWSetSpriteMoveDelta(SpritePtr srcSpriteP,                   short
horizDelta,                     short vertDelta)

void SWSetSpriteMoveProc(SpritePtr srcSpriteP,
                                MoveProcPtr moveProc)

void SWSetSpriteMoveTime(SpritePtr srcSpriteP,                    long
timeInterval)

void SWSetSpriteVisible(SpritePtr srcSpriteP,                     Boolean
isVisible)

void SWUnlockSprite(SpritePtr srcSpriteP)

OSErr SWUpdateSpriteFromPictResource(SpritePtr theSpriteP,        short
pictResID)

Boolean SWWrapSprite(SpritePtr srcSpriteP)
```

Frame Routines

```
OSErr SWCopyFrame(SpriteWorldPtr destSpriteWorldP,
                                FramePtr oldFrameP,
                                FramePtr *newFrameP,              Boolean
copyMasks)

OSErr SWCreateFrame(GDHandle theGDH,
                                FramePtr* newFrameP,              Rect*
frameRect,                      short depth)
```

```
OSErr SWCreateWindowFrame(FramePtr* newFrameP,                    Rect*
frameRect,                      short maxHeight)

void SWDisposeFrame(FramePtr *deadFrameP)

void SWDisposeWindowFrame(FramePtr *deadFrameP)

void SWLockFrame(FramePtr srcFrameP)

void SWLockWindowFrame(FramePtr windowFrameP)

void SWSetFrameHotSpot(FramePtr srcFrameP,                        short
hotSpotH,                       short hotSpotV)

void SWUnlockFrame(FramePtr srcFrameP)

void SWUnlockWindowFrame(FramePtr windowFrameP)

OSErr SWUpdateFrameMasks(SpriteWorldPtr destSpriteWorldP,
                                FramePtr srcFrameP)

void SWWindowFrameMoved(FramePtr windowFrameP,                    Rect
*frameRect)
```

Utility, Sprite Compiler and Miscellaneous Routines

```
void SWAnimate8BitStarField(SpriteWorldPtr spriteWorldP,
                                StarArray *starArray,             short
numStars,                       unsigned short backColor)

void BlitPixie8BitFlipSprite(SpritePtr srcSpriteP)

unsigned char BlitPixie8BitGetPixel(FramePtr srcFrameP,           Point
thePoint)

void BlitPixie8BitSetPixel(FramePtr srcFrameP,                    Point
thePoint,                       unsigned char theColor)

void SWClearStickyError(void)

OSErr SWCompileSprite(SpritePtr srcSpriteP)

void SWSetCleanUpSpriteWorld(SpriteWorldPtr spriteWorldP)

void SWSetStickyIfError(OSErr errNum)

void SWSetTransparentColor(const RGBColor *theColor)

OSErr SWStickyError(void)
```