Draft 1.2 6/20/93

Inside SpriteWorld

by Tony Myles

I apologize for any out of date information and incomplete areas in this documentation. It is a constant battle keeping it in synch with the development of SpriteWorld itself. I am working on it as feverishly as possible.

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 color 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 drawing on and off screen
- synchronize animation on millisecond intervals
- perform simple about box animations or write full blown arcade games

To use SpriteWorld it may help to have at least a passing familiarity with…

- QuickDraw, especially offscreen GrafPorts, CopyBits, and regions
- the Time Manager

Acknowledgements and References

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 some very enlightening tech notes and sample code. Thanks to Brigham Stevens for his expertise in writing directly to the screen in as safe a manner as possible. 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, and for his great work on the upcoming C++ version of SpriteWorld. Thanks to all the guys (Paula too!) at Pharos for their support and help way back when. Special thanks to all the folks who have been sending me e-mail with their comments, criticisms, congratulatory remarks, etc. More to come.

I has been brought to my attention that it would be appropriate to make reference to related work here. The source code to Glypha and Glider by John Calhoun is what got me started, as well as Duane Blehm's work with regions as seen in Cairo Shootout. Other packages you will want to check out are Juri Munkki's Vector Animation Toolkit, Eli Bishop's Sprite Manager, and Ingemar Ragnemalm's Sprite Animation Toolkit. These have all been helpful to me in one way or another.

Introduction to Sprites

What is a sprite?

"**Sprite**" is a technical term meaning an animated object that appears on the computer's screen and may move around or exhibit other interesting behaviour. 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 little aliens can be considered sprites.

What a sprite really is in terms of SpriteWorld's implementation, 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 have this notion of a current frame. The current frame of a sprite is the image that will be drawn when the the animation is rendered on the screen. By advancing the current frame in sequence the sprite will appear to be animated. At the same time the screen position at which the sprite's current frame will be drawn can be adjusted 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 completely unaware of anything. 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 movement parameters, 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

Why SpriteWorld?

Unlike the Amiga and other game machines, the Macintosh has no sprite animation hardware built-in. Sprite animation therefore, must to be implemented in software. SpriteWorld is an attempt to implement a sprite-based animation architecture on the Macintosh. This has been done before, but it was not done right, or it was not released to the general developer community.

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 is copied to the offscreen work image. The area of this section is calculated by taking the union of the sprite's last position and its current position.
- The current frame of the sprite is then drawn in the sprite's current position in the offscreen work area. The result is the sprite resting on the piece of background copied in the previous step.
- This piece 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.

When creating animations using SpriteWorld you will deal primarily with four simple 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.

<architecture diagram here>

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.

A SpriteWorld contains a frame for the offscreen background area, the offscreen work area, and the screen itself. You can choose to create these frames yourself, or have them created automatically depending on the circumstances of your animation.

A SpriteWorld also maintains a list of the sprite layers 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 world.

In terms of the actual drawing the SpriteWorld is responsible for erasing the sprite offscreen and drawing the sprite onscreen. You can install custom routines to handle this by writing a custom pixel blitter. By default SpriteWorld uses QuickDraw's CopyBits routine for all drawing operations.

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, and 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 properly.

Aside from the animation, this layering facility is also used by the collision detection mechanism. If you arrange your sprites in logical layers, ie. the good guys in one layer, 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 custom move routine, your sprites can exhibit extremely complex behaviour 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 custom frame routine, you can perform more sophisticated frame animation such as rotating a space ship when certain key is pressed.

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 simply contains an offscreen GrafPort in which the actual image is stored, and a mask.

Using SpriteWorld

During the early development of SpriteWorld the emphasis was put on the speed of the animation, sacrificing ease of use, and features. Once the animation reached an acceptable level of speed, the feature set was enhanced. Once the features reached an acceptable level, ease of use was emphasized. The result of all this is an animation architecture that hopefully maintains a good balance between these three concerns in their proper proportions. In this section we will attempt to describe how you can easily make use of SpriteWorld to create fast, smooth animation.

Getting Started

Performing animation using SpriteWorld involves 4 **core** steps, and 3 initialization or housekeeping steps…

- Initialize the SpriteWorld package.
- **Create the various pieces, ie. the SpriteWorld, the SpriteLayers, the Sprites, and the**

Frames.

- **Assemble the various pieces, ie. 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 behaviour.**
- **Drive the animation using SWProcessSpriteWorld and SWAnimateSpriteWorld in a tight loop. Collision detection may optionally be performed here.**
- When the animation is finished, you must dispose of all the pieces (SpriteWorlds, SpriteLayers, Sprites, and Frames) that were created earlier.
- The last step is to ask the SpriteWorld package to clean up.

While the animation is running you may add or remove individual Sprites or entire SpriteLayers.

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,

      &windowRect);
```

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);
```

The easiest way to create a Sprite with a set of Frames is by calling SWCreateSpriteFromCIconResource.

```
err = SWCreateSpriteFromCIconResource(&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 sprite layer
SWAddSpriteLayer(spriteWorldP, spriteLayerP);
```

While an animation is running it is possible that you may want to add or remove a Sprite or SpriteLayer on the fly. SpriteWorld does support this but you should be aware that if you remove an active Sprite from the animation it will simply be left where it is on the screen and will not be erased. Typically you will want to erase the Sprite first by making it invisible which is accomplished by making a call to SWSetSpriteVisible.

```
      // make this sprite invisible
SWSetSpriteVisible(newSpriteP, false);
```

Defining Sprite Behaviour

Before and possibly during the animation you will want to define the movement behaviour of your Sprites. The following code snippet cover most of the basic behavioural 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
SWSetSpriteMoveInterval(newSpriteP, 30);
```

```
	// set the sprite's movement direction/distance
SWSetSpriteMoveDelta(newSpriteP, 10, 5);

	// more to be added here…
```

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 the 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);
}
```

Detecting Collisions

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

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(shipSpriteLayerP, enemySpriteLayerP);

	// we may want to see if any of the enemies
```

```
        // have collided with each other
SWCollideSpriteLayer(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 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 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.

SpriteWorld Reference

This section serves as a reference to the routines SpriteWorld provides. There are routines for manipulating each of the four core data structures, SpriteWorlds, SpriteLayers, Sprites, and Frames. Some utility routines, for working with color icon and pict resources, are also provided.

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 is initialization fails, other wise 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. It should be called when the application is finished using SpriteWorld to balance the original call to `SWEnterSpriteWorld`. At this point no further calls should be made to any SpriteWorld routines until `SWEnterSpriteWorld` is called again.

SWCreateSpriteWorld

Use This function will create a new SpriteWorld, allowing you to specify the frames to be used for the screen, background, and work area.

```
OSErr SWCreateSpriteWorld(SpriteWorldPtr *spriteWorldP,
                          FramePtr screenFrameP,
                          FramePtr backFrameP,
                          FramePtr workFrameP);
```

spriteWorldP	A newly created SpriteWorld is returned in this parameter.
screenFrameP	A frame used to represent the screen.
backFrameP	A frame to be used for the offscreen background area.
workFrameP	A frame to be used for the offscreen work area.

DESCRIPTION

The SWCreateSpriteWorld function is used to create a new SpriteWorld manually. If you choose to create the screen, background, and work frames yourself, you should use this function to create the SpriteWorld. You might do this if you wish to set up the frames in a some special way.

SWCreateSpriteWorldFromWindow

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

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

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.

DESCRIPTION

The SWCreateSpriteWorldFromWindow function provides is used to create a new SpriteWorld for an existing window in an easier and more automatic fashion than is possible using SWCreateSpriteWorld. 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.

SWDisposeSpriteWorld

This will dispose of an existing SpriteWorld, 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 SWCreateSpriteFromWindow. The memory occupied by the SpriteWorld, as well as the screen, background, and work frames will be released.

Δ WARNING Δ

The SWDisposeSpriteWorld function will also dispose of the screen, background, and work frames used by the SpriteWorld. You should not dispose of them yourself after calling this function even if you did create them yourself.

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.

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 the sprite's visibility to false, render a frame of the animation, and then remove the layer from the world.

SEE ALSO

SWSetSpriteVisibility

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.

SWLockSpriteWorld

This function will lock 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 it must first be locked.

SPECIAL CONSIDERATIONS

You must be careful when adding a SpriteLayer, Sprite, or Frame to an animation that is already locked and running. Everything must be locked before it can be animated.

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 will 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.

SWSetSpriteWorldEraseProc

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

```
void SWSetSpriteWorldEraseProc(SpriteWorldPtr spriteWorldP,
                               DrawProcPtr eraseProc);
```

spriteWorldP	A SpriteWorld whose erase routine will be set.
eraseProc	A new erase routine.

DESCRIPTION

The SWSetSpriteWorldEraseProc function is used to specify a new erase routine for the given SpriteWorld. This routine is used to erase a sprite in the offscreen background area. The standard erase routine calls CopyBits.

The routine you pass must be of type DrawProcPtr which is defined like so…

```
typedef void (*DrawProcPtr)(FramePtr srcFrameP,
                            FramePtr dstFrameP,
                            Rect *srcRect,
                            Rect *dstRect);
```

srcFrameP	A Frame containing the source image.
dstFrameP	A Frame containing the destination image.
srcRect	A rectangle defining the area of the source image to be copied from.
dstRect	A rectangle defining the area of the destination image to be copied into.

SWSetSpriteWorldDrawProc

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

```
void SWSetSpriteWorldDrawProc(SpriteWorldPtr spriteWorldP,
                              MaskDrawProcPtr drawProc);
```

spriteWorldP	A SpriteWorld whose draw routine will be set.
drawProc	A new draw routine.

DESCRIPTION

The SWSetSpriteWorldDrawProc function is used to specify a new draw routine for the given SpriteWorld. This routine is used to draw a sprite on the screen by copying the sprite's image from the offscreen work frame to the screen frame. The standard draw routine calls CopyBits.

The routine you pass must be of type MaskDrawProcPtr which is defined like so…

```
typedef void (*MaskDrawProcPtr)(FramePtr srcFrameP,
                                FramePtr dstFrameP,
                                Rect *srcRect,
                                Rect *dstRect,
                                RgnHandle maskRgn);
```

srcFrameP	A Frame containing the source image.
dstFrameP	A Frame containing the destination image.
srcRect	A rectangle defining the area of the source image to be copied from.
dstRect	A rectangle defining the area of the destination image to be copied into.
maskRgn	A region defining the area to which the image copy should be clipped.

SWUpdateSpriteWorld

The function will draw the current frame of the animation in reponse to an update event.

```
void SWUpdateSpriteWorld(SpriteWorldPtr spriteWorldP);
```

spriteWorldP	A SpriteWorld to be updated.

DESCRIPTION

The SWUpdateSpriteWorld function is used to build the current frame in the offscreen work area, 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.

SWProcessSpriteWorld

This function processes all the Sprites in a SpriteWorld, updating there positions, resetting there timers, calling there 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 there movement and frame change timers, and calling there custom move and frame routines if any. This function, in conjunction with SWAnimateSpriteWorld, drives the animation.

This function does no drawing, it simply processes a sprite in terms of its movement and frame changing characteristics using the parameters you specify when setting up the sprite.

SEE ALSO

SWAnimateSpriteWorld

SWAnimateSpriteWorld

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

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

spriteWorldP A 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.

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.

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.

SWAddSprite

This function will add an existing Sprite to a SpriteLayer.

```
void 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 a SpriteLayer.

SWRemoveSprite

This function will remove a sprite from a layer.

```
void SWRemoveSprite(SpriteLayerPtr spriteLayerP,
                    SpritePtr oldSpriteP);
```

spriteLayerP	An existing SpriteLayer.
oldSpriteP	A Sprite to be removed from the specified SpriteLayer.

DESCRIPTION

The SWRemoveSprite function is used to remove a Sprite from a SpriteLayer . This is done when you want to remove the sprite from the animation. The sprite that is removed will not be processed or drawn when the next frame of the animation is rendered.

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 .

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 the are no more Sprites in the SpriteLayer, NULL is returned.

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 Sprite or Frame to an animation that is already locked and running. Everything must be locked before it can be animated.

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.

SWCollideSpriteLayer

This function will check for collisions between two SpriteLayers.

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

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.

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 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.

SEE ALSO

SWSetSpriteCollideProc.

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.
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.

SWCreateSpriteFromCIconResource

This function will create a new sprite complete with a set of frames created from a series of color icon ('cicn') resources.

```
OSErr SWCreateSpriteFromCIconResource(SpritePtr *newSpriteP,
                                      void* spriteStorageP,
                                      short cIconID,
                                      short maxFrames,
                                      short frameFlags);
```

newSpriteP	The newly created sprite.
spriteStorageP	Pointer to memory in which the sprite structure will be stored.
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.
frameFlags	A value that indicates how the frames for the sprite should be created. For a description of what these flags are their meaning see below.

DESCRIPTION

The SWCreateSpriteFromCIconResource 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 frame flags parameter can currently be one of three values described here...

kNoMask	A value indicating that no mask should be created for the frames of the sprite.
kRegionMask	A value indicating that a QuickDraw region (RgnHandle) should be

created, and used as a mask for the frames of the sprite.

kPixelMask	A value indicating that an offscreen GrafPort should be created, and used as a mask for the frames of the sprite. This GrafPort will be the same bit depth as the frame image and is suitable for use with a custom mask blitter.
kFatMask	This value is equivalent to kRegionMask + kPixelMask. This results in a "fat" frame that contains both of the above types of masks. This is useful if your application switches between using QuickDraw and custom drawing routine, at runtime.

SWCreateSpriteFromPictResource

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

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

newSpriteP	The newly created sprite.
spriteStorageP	Pointer to memory in which the sprite structure will be stored.
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.
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.
frameFlags	A value that indicates how the frames for the sprite should be created. For a description of what these flags are their meaning see SWCreateSpriteFromCIconResource.

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 cIconID. 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.

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.

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.

SPECIAL CONSIDERATIONS

Since these Sprites share their frames you must be careful when disposing of these Sprites not to dispose of the frame set twice.

SWDisposeSprite

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

```
void SWDisposeSprite(SpritePtr deadSpriteP, Boolean disposeFrames);
```

deadSpriteP A Sprite to be disposed.

disposeFrames A boolean value indicating whether or not the frames of the Sprite should be automatically disposed along with the Sprite.

DESCRIPTION

The SWDisposeSprite function will dispose of a Sprite, and optionally dispose of the frames used by the Sprite.

The disposeFrame parameter is typically used when you have a number of Sprites that

share the same frames. In order to avoid disposing of the frames twice you will want to dispose of the first Sprite that owns the frames by passing true into disposeFrames, subsequent Sprites should be disposed of by passing false into disposeFrames.

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 locked and running. Everything must be locked before it can be animated.

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.

SWSetSpriteDrawProc

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

```
void 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.

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 new frame to an existing Sprite. This frame may also be added to other Sprites so that they may share frames, thus saving memory.

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.

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.

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.

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. The Sprite's current frame index will be automatically incremented when the Sprite is processed by the SWProcessSpriteWorld function.

SEE ALSO

SWSetSpriteFrameRange

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. This 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.

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 SWSetSpriteFrame 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 0 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.

SEE ALSO

SWSetSpriteFrameAdvance

SWSetSpriteFrameProc

This function set the routine to be called when the Sprite's current 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.

The routine you pass must be of type FrameProcPtr which is defined like so…

```
typedef 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.

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.

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.

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

Describe special considerations here.

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 complex movement behavior

SWSetSpriteMoveDelta

This function sets the values by which the Sprite's current position will be automatically offset.

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

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

DESCRIPTION

The SWSetSpriteMoveDelta function is used to specify the values by which the Sprite's

current position will be automatically offset. The horizontal and vertical deltas indicate the direction and distance the Sprite will be moved when the Sprite is processed by SWProcessSpriteWorld.

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.

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

SWBounceSpriteMoveProc

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

```
void SWBounceSpriteMoveProc(SpritePtr srcSpriteP,
                            Point *spritePoint);
```

srcSpriteP	A Sprite being moved.
spritePoint	The current location of the Sprite.

DESCRIPTION

The SWBounceSpriteMoveProc function is provided for use as a movement routine for a Sprite. When this function is installed as a Sprite's movement routine, the Sprite will bounce around inside the area defined by its movement boundary rectangle.

SEE ALSO

SWSetSpriteMoveBounds

SWWrapSpriteMoveProc

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

```
void SWWrapSpriteMoveProc(SpritePtr srcSpriteP,
                          Point *spritePoint);
```

srcSpriteP	A Sprite being moved.
spritePoint	The current location of the Sprite.

DESCRIPTION

The SWWrapSpriteMoveProc function is provided for use as a movement routine for a Sprite. If this function is installed as a Sprite's movement routine, and the Sprite moves outside the area defined by its movement boundary rectangle it will be wrapped to the other side.

SEE ALSO

SWSetSpriteMoveBounds

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 futher processing to determine if the Sprites have actually collided, and if so perform some action such as playing an explosion sound.

SEE ALSO

SWCollideSpriteLayer.

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.

The collision routine you provide must of type CollideProcPtr which is define as…

```
typedef void (*CollideProcPtr)(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.

FunctionName

Short description here.

```
Boolean Function (short param1,
                  long param2);
```

param2	Describe the parameters here.
param2	Describe the parameters here.

DESCRIPTION

Describe the function here.

SPECIAL CONSIDERATIONS

Describe special considerations here.

Summary of SpriteWorld

```
///-----------------------------------------------------------------------------
//    sprite world type definitions
///-----------------------------------------------------------------------------

typedef struct SpriteWorldRec SpriteWorldRec;
typedef SpriteWorldRec *SpriteWorldPtr, **SpriteWorldHdl;


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

```
struct SpriteWorldRec
{
      SpriteLayerPtr headSpriteLayerP;    // head of the active layer linked list

      FramePtr screenFrameP;                    // frame for drawing to the screen
      FramePtr backFrameP;                        // frame for drawing from the
background
      FramePtr loadFrameP;                        // frame for drawing to the loader

      DrawProcPtr eraseDrawProc;                // callback for erasing sprites offscreen
      MaskDrawProcPtr screenDrawProc;     // callback for drawing sprite pieces onscreen

      long userData;                                          // reserved for user
};


///-----------------------------------------------------------------------------
//    sprite layer type definitions
///-----------------------------------------------------------------------------

typedef struct SpriteLayerRec SpriteLayerRec;
typedef SpriteLayerRec *SpriteLayerPtr, **SpriteLayerHdl;


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

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

      SpritePtr headSpriteP;                    // head of sprite linked list
      short numSprites;                           // number of sprites

      Boolean isVisible;
      Boolean needsToBeDrawn;
      Boolean needsToBeUpdated;

      long userData;                                        // reserved for user
};
```

```
///-----------------------------------------------------------------------------
//    time task data structure
///-----------------------------------------------------------------------------

typedef struct SWTimeTaskRec SWTimeTaskRec;
typedef SWTimeTaskRec *SWTimeTaskPtr, **SWTimeTaskHdl;

struct SWTimeTaskRec
{
      TMTask timeTask;
      Boolean taskIsPrimed;
      Boolean taskHasFired;
};


///-----------------------------------------------------------------------------
//    sprite type definitions
///-----------------------------------------------------------------------------

typedef struct SpriteRec SpriteRec;
typedef SpriteRec *SpritePtr, **SpriteHdl;

typedef void (*DrawProcPtr)(FramePtr srcFrameP,
                                                    FramePtr dstFrameP,
                                                    Rect *srcRect,
                                                    Rect *dstRect);
typedef void (*MaskDrawProcPtr)(FramePtr srcFrameP,
                                                          FramePtr dstFrameP,
                                                          Rect *srcRect,
                                                          Rect *dstRect,
                                                          RgnHandle maskRgn);
typedef void (*FrameProcPtr)(SpritePtr srcSpriteP,
                                                     FramePtr curFrameP,
                                                     long *frameIndex);
typedef void (*MoveProcPtr)(SpritePtr srcSpriteP, Point *spriteLoc);
typedef void (*CollideProcPtr)(SpritePtr srcSpriteP,
                                                          SpritePtr dstSpriteP,
                                                          Rect* sectRect);
```

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

struct SpriteRec
{
      SpritePtr nextSpriteP;                        // next sprite
      SpritePtr prevSpriteP;                        // previous sprite

            // drawing fields
      Boolean isVisible;                                  // draw the sprite?
      Boolean needsToBeDrawn;                       // sprite has changed, needs to be
drawn
      Boolean needsToBeUpdated;                     // sprite needs to be updated
offscreen
      Rect destFrameRect;                                 // frame destination rectangl
      Rect oldFrameRect;                                  // last frame destination
rectangle
      Rect deltaFrameRect;                                // union of the sprite's last
and curRect
      RgnHandle screenMaskRgn;                      // mask region for drawing sprite t
screen
      DrawProcPtr frameDrawProc;                    // callback to draw sprite

            // frame fields
      TimeTaskRec frameTimeTask;                    // frame advance time task
      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
      long frameAdvance;                                  // amount the 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

            // movement fields
      TimeTaskRec moveTimeTask;                     // movement time task
      long moveTimeInterval;                        // time interval to move sprite
      short horizMoveDelta;                         // horizontal movement delta
      short vertMoveDelta;                                // vertical movement delta
      Rect moveBoundsRect;                                // bounds of the sprite's
```

```
    movement
        MoveProcPtr spriteMoveProc;             // callback to handle movement

            // collision detection fields
        Rect collideRect;                                   // collision detection
    rectangle
```

```
        CollideProcPtr spriteCollideProc;   // callback to handle collisions

        long userData;                                          // reserved for user
};


///-------------------------------------------------------------------------------
//    frame type definitions
///-------------------------------------------------------------------------------

typedef struct FrameRec FrameRec;
typedef FrameRec *FramePtr, **FrameHdl;


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

struct FrameRec
{
        union
        {
                CGrafPtr colorGrafP;                // color port for the frame image
                GrafPtr monoGrafP;                  // mono port for the frame image
        } framePort;

        union
        {
                PixMapPtr pixMapP;                  // color pix map (valid only while locked
                BitMapPtr bitMapP;                  // mono bit map (valid only while locked)
        } framePix;

        unsigned long *frameBaseAddr; // base address of pixel data
        unsigned long frameRowBytes;  // number of bytes in a row of the frame
        unsigned long frameRowLongs;  // number of long words in a row of the frame
        short leftAlignFactor;              // used to align the left to long word
        short rightAlignFactor;             // used to align the right to long word
        Boolean isFrameLocked;              // has the frame been locked?
        Boolean isColor;                          // is this a color frame?

        Rect frameRect;                           // source image rectangle
        Point offsetPoint;                        // image offset factor relative to
destination
```

```
        RgnHandle maskRgn;                          // image masking region

        union
        {
              CGrafPtr colorGrafP;                  // color port for the mask image
              GrafPtr monoGrafP;                    // mono port for the mask image
        } maskPort;

        union
        {
              PixMapPtr pixMapP;                    // pointer to color pix map
              BitMapPtr bitMapP;                    // pointer to mono bit map
        } maskPix;

        unsigned long *maskBaseAddr;  // base address of mask pixel data
        Boolean isMaskLocked;               // has the mask been locked?
};
```