SpriteWorld 2 Tips and Tricks

Contents:

Introduction

This file assumes you have some familiarity with SpriteWorld 2. It offers some ideas for less-obvious ways of working with SpriteWorld and extending its capabilities. SpriteWorld users should feel free to offer contributions to this file. Send submissions to Vern Jensen at:
Jensen@loop.com

For up to date information about SpriteWorld 2, and to download the latest version of SpriteWorld 2, visit the official Web site at:
http://users.aol.com/spritewld2/

Tips and Tricks

Attaching a data structure to a Sprite

Often, you will need to keep track of more information relating to a Sprite than can be stored in the userData field of a sprite. To do this, you will first make a data structure containing the additional variables. Then you need to “attach” this data structure to your sprite. Essentially, there are two ways to do this.

The first, perhaps a little easier to understand, is to put a pointer to a data structure into the userData field of a sprite. You can either declare the data structure as a global, or allocate memory for it as a pointer with NewPtr(). You’ll also want to create a data type corresponding to your data structure:

```
// the typedef
typedef struct MyDataStruct
{
```

```
   short  direction;
   Boolean isMoving;
} MyDataStruct, *MyDataStructPtr;

// the global variable declaration
MyDataStruct  gMyDataStruct;
```

Then you put the pointer to the structure into the sprite:

```
mySpriteP->userData = (long)&gMyDataStruct;
```

Once this is done, you can access the data structure by way of the SpritePtr:

```
((MyDataStructPtr)mySpriteP->userData)->direction = kUp;
```

The second method is to define a data structure, and make the **SpriteRec** (not the SpritePtr) the first element in that structure.

```
// the typedef
typedef struct MyDataStruct
{
SpriteRec  mySprite;
short  direction;
Boolean isMoving;
} MyDataStruct, *MyDataStructPtr;
```

Once this is done, you can again access the data structure by way of the SpritePtr -- because they both point to the same physical memory address. Because the SpritePtr is the address of the SpriteRec, and your data structure begins with that SpriteRec, the other elements of your data structure will inhabit the memory directly following the memory inhabited by the SpriteRec. To access your data structure in this case, you cast the type of the SpritePtr to the “MyDataStructPtr” type that you defined. When you use this method, you have to create the sprite correctly, so that its SpriteRec will in fact inhabit the memory location you want it to. To make the memory used by the data structure permanent and stationary, it is allocated with NewPtr():

```
tempSpritePtr = (MyDataStructPtr)NewPtr(sizeof(MyDataStruct));

err = SWCreateSpriteFromPictResource(
 mySpriteWorldP, // required by SpriteWorld 2
 mySpriteP,  // this can be, but doesn't have to be, a global variable
 tempSpritePtr, // this tells SW to use the pointer memory you allocated
 kSpritePicResID,
 kSpriteMaskResID,
 kNumFrames,
 kFatMask);
```

If the mySpriteP variable is a global, then you’ll be able to access the sprite at any time, rather than just in routines like a collideProc that have the SpritePtr passed to them. In any case, now you can access your data structure any time you can access the SpritePtr, simply by casting the SpritePtr’s type:

((MyDataStructPtr)(mySpriteP))->direction = kUp;

An advantage to this latter method of attaching data to a Sprite is that when the Sprite is disposed of with SWDisposeSprite(), the memory allocated for the additional data will also be released. SWDisposeSprite calls DisposePtr((Ptr)theSpriteP), which releases the SpriteRec memory and the rest of the data structure.

==========

Modifying a Sprite's image

At times you may want to modify a Sprite's image directly, rather than loading a new Sprite from a resource. To accomplish this, you simply have to make sure the Sprite is locked (or the Frame that you are going to modify is locked), and then call SetGWorld to set the port to the Sprite's framePort. Here's an example:

SetGWorld(mySpriteP->curFrameP->framePort, nil);

This will set the port to the Sprite's current Frame. To set the port to a Frame using a Frame index, use this:

frameIndex = 2;
SetGWorld(mySpriteP->frameArray[frameIndex]->framePort, nil);

This will set the port to the third Frame in the Sprite. (Remember that the frameIndex starts at 0.) The frameArray variable is an array containing pointers to all the Frames contained in a Sprite. To get the dimensions and position of the Frame in the GWorld, use this:

frameRect = mySpriteP->frameArray[frameIndex]->frameRect;

Keep in mind that if you draw outside this rectangle, you may draw over images for other Frames contained in the Sprite, if the Frames were loaded using SWCreateSpriteFromSinglePict. This is because that function loads all Frames into a single "shared" GWorld. The frameRect variable for a particular Frame tells you where that Frame's image is located in the GWorld.

Once you have set the port and know the rectangle for the Frame's image, you can use standard QuickDraw calls (or even SpriteWorld's blitters) to change the Frame's image. For instance, this would fill the first Frame of a Sprite with a blue rectangle:

SetGWorld(mySpriteP->frameArray[0]->framePort, nil);
ForeColor(blueColor);
PaintRect(&mySpriteP->frameArray[0]->frameRect);

Keep in mind that once you change a Sprite's image, you also should change the mask. This can be accomplished with the SWUpdateFrameMasks function, or a much faster method would be to modify the mask directly, if you have a pixel mask. See the section below for more information.

==========

Modifying a Sprite's pixelMask

One of the nice things about using a pixelMask for your Sprite is that you can modify the mask directly, simply by setting the port to the maskPort, and using standard QuickDraw calls to change the image. This is because the pixelMask is simply a GWorld the same size and depth as the Sprite's image that contains a B&W picture representing the Sprite's mask. One thing you should be aware of, however, is that the B&W image will be inverted if the depth of the Sprite is 8-bits (256 colors) or less. That is, white = on, black = off. But when running in thousands of colors or more, the standard black = on, white = off is the case.

To modify a Sprite's mask, you first must make sure the Frame your are modifying is locked. (It will be if the Sprite is locked.) Then you should set the port to the Frame's maskPort, set the ForeColor to either whiteColor of blackColor, and start drawing. Here's an example that draws a circle in the mask port of the first Frame of a Sprite after erasing the previous contents of the mask:

```
        // mySpriteP->frameArray[0] gives us a pointer to the first Frame in the Sprite
SetGWorld(mySpriteP->frameArray[0]->maskPort, nil);

    // Mask image is inverted when in 8-bit or less
if (gSpriteWorldP->pixelDepth <= 8)
{
    ForeColor(whiteColor);
    BackColor(blackColor);
}
else
{
    ForeColor(blackColor);
    BackColor(whiteColor);
}

myRect = mySpriteP->frameArray[0]->frameRect;          // get size of frame
EraseRect(&myRect);    // Fill rect using background color
PaintOval(&myRect);    // Draw circle using foreground color
```

The third line from the end gets the rectangle containing both the position and size of the mask in the maskPort. If you draw outside this rectangle, you could end up drawing into other masks if you are modifying a Sprite that was created using a function such as SWCreateSpriteFromSinglePict, since all the masks are stored in a single GWorld. In this case, the frameRect for the Frame you access will specify the exact position of the mask image for that Frame in the maskPort.

It is important to make sure the Sprite's image, contained in the framePort (mySpriteP->frameArray[0]->framePort) matches the Sprite's mask. That is, those pixels in the Sprite's framePort that correspond to the "off" pixels in the maskPort should be white. If they are not white, the Sprite will not be drawn correctly. This is because SpriteWorld's standard blitters can go faster by assuming that all unmasked pixels in the Sprite's image are white. In short, all I'm saying is that if you modify the mask, make sure you modify the image to match it, changing unmasked pixels in the Sprite's image to white.

==========

Forcing a Sprite to be redrawn

Occasionally, you may wish to force an idle sprite to be redrawn. One example would be if you changed the Sprite's drawProc to a drawProc that draws the sprite differently. This is done in Shark Attack; when the fish are shot, they flash white, which is accomplished by using a special drawProc to draw them. If the fish were idle, or didn't move that particular frame, it would be necessary to force the sprite to be drawn, since simply changing the Sprite's drawProc will not do this. (And the fish wouldn't flash white if it weren't drawn once the new drawProc was installed.) To force a Sprite to be redrawn, simply set the needsToBeDrawn variable of that Sprite to true. Here's an example:
mySpriteP->needsToBeDrawn = true;

Normally you won't need to do this, since SpriteWorld automatically sets the Sprite's needsToBeDrawn flag whenever the Sprite is moved or the current Frame image is changed. However, in special cases, such as the one mentioned above, you may find it necessary. Another example of a situation where you may need to force a sprite to be redrawn would be if you changed that sprite's image. For an example of this, take a look at the UpdateDiamondMeter function in Scrolling Demo.c.

==========

Getting the mouse coordinates when a worldRect is used

There are times when you may need to get the mouse's current position in your SpriteWorld, so that you can make a Sprite follow the mouse (as is done in the Tiling Demo), or so you can see if the mouse was over a Sprite when the button was clicked. Normally, this process is quite straightforward, since the coordinates returned by GetMouse() are already local to your window. However, if a worldRect is used, an additional operation will need to be made before these coordinates are local to your SpriteWorld, since your SpriteWorld's origin is not the top-left corner of the window.

To convert the mouse coordinates that are local to your window into coordinates that are local to your SpriteWorld with a worldRect, you simply must subtract the offset between your worldRect's top-left corner and the window's top-left corner from the mouse coordinates. Here's an example:

```
GetMouse(&mousePoint);

        // Subtract any offset from worldRect to window if a worldRect is used
mousePoint.h -= gSpriteWorldP->windRect.left;
mousePoint.v -= gSpriteWorldP->windRect.top;
```

This will work because the SpriteWorld's windRect variable contains the rectangle, in coordinates local to the window, of the SpriteWorld's area in that window. In fact, this will work properly even if you don't have a worldRect, because in that case, the windRect.left and windRect.top will be 0, leaving mousePoint unchanged.

==========

How to keep Sprites at a fixed position on the screen while scrolling

Sometimes you may have a Sprite that needs to know the visScrollRect's position for the current frame before the Sprite can move. An example of this would be the Sprite in the upper left-hand corner of the Scrolling Demo. While it appears to remain in a fixed position on the screen, it actually moves offscreen every time the screen scrolls. The Sprite simply moves itself to the current visScrollRect's top-left corner, plus a small offset of about 10 pixels to give it a little space between itself and the corner of the screen.

The only problem is that the visScrollRect is moved after all the Sprites are processed by SWProcessScrollingSpriteWorld. This means that when the Sprites are processed, they do not have the visScrollRect position for the current frame, but rather, they have its position from last frame. To get around this problem, don't give your Sprite a MoveProc, but rather, call the MoveProc yourself immediately after calling SWProcessScrollingSpriteWorld, but before calling SWAnimateScrollingSpriteWorld and any collision routines. This will allow the Sprite to use the current visScrollRect position, rather than the one from the previous frame. For more information on this technique, see the Scrolling Demo Read Me file.

==========

How to keep the mouse from flickering while in front of a scrolling animation

Sometimes you may need to be able to move the mouse in front of a scrolling animation while it is running. Normally, the mouse cursor will flicker or disappear if you do this, since CopyBits hides the mouse while it is drawing the window's contents, and SpriteWorld's blitters will draw right over the mouse cursor, completely erasing it until it is moved. Naturally, neither of these results are acceptable if the user needs to be able to move the mouse in front of the animation. (One example of this would be turn-based RPG type games, where nothing happens until the user clicks the mouse indicating where the player should move.)

One solution to this scenario is to not call SWProcess or SWAnimate until the user clicks the mouse. However, this is an acceptable solution only in games where nothing happens until the user clicks the mouse. For other games, where the animation needs to continue to run even when the user doesn't click the mouse, you can keep the mouse from flickering or disappearing by adding a "fake mouse" sprite to the SpriteWorld in a layer higher than all the other sprites reserved specifically for the mouse sprite. Each frame, you should:

1) Get the mouse's current position (using GetMouse), first making sure your window is the current port, since GetMouse returns coordinates that are local to the current port).

2) Subtract the offset from your SpriteWorld's worldRect to the window if a worldRect is used. (See the section in this file entitled "Getting the mouse coordinates when a worldRect is used")

3) Move your mouse sprite to that location. However, you shouldn't install the MoveProc for this sprite in your SpriteWorld, since it needs the very latest visScrollRect value each frame. Instead, call the sprite's MoveProc directly each frame just after calling SWProcessScrollingSpriteWorld. (See the section called "How to keep Sprites at a fixed position on the screen while scrolling" for more info.)

For a demonstration of this technique, take a look at the Large Background Scrolling demo, available in the SpriteWorld 2.1 Demos package, which is distributed separately from SpriteWorld.

==========

Calling SpriteWorld's blitters directly

Although you can always use CopyBits to do your copying, it's much faster to use one of SpriteWorld's blitters. Although these blitters were designed primarily for drawing and copying Sprites, you can call them directly as well. All you have to do is call the desired DrawProc directly, passing it the appropriate information yourself. For a list of all the DrawProcs, take a look at BlitPixie.h. Each DrawProc has the following parameters:

srcFrameP - the source Frame
dstFrameP - the destination Frame
*srcRect - the source rectangle
*dstRect - the destination rectangle

The srcFrameP and destFrameP are both pointers to a Frame structure. This includes Tile Frames, Sprite Frames, and SpriteWorld's work and background Frames. Here's an example that would copy the contents of the SpriteWorld's background Frame into its work Frame using the BlitPixieAllBitRectDrawProc.

```
BlitPixieAllBitRectDrawProc(spriteWorldP->backFrameP, spriteWorldP->workFrameP, &spriteWorldP->backFrameP->frameRect, &spriteWorldP->workFrameP->frameRect);
```

Here's an example that would copy the image for TileID 2 into a Sprite's second frame (index 1), assuming the Sprite's Frame is big enough to contain the tile image:

```
FramePtr tileFrameP = spriteWorldP->tileFrameArray[2];
FramePtr spriteFrameP = mySpriteP->frameArray[1];

BlitPixieAllBitRectDrawProc(tileFrameP , spriteFrameP , &tileFrameP ->frameRect, &spriteFrameP ->frameRect);
```

The blitter automatically clips the drawing to the dstFrameP's frameRect, so even if the rectangle you pass in the last parameter extends past the destination Frame's boundaries, it will still be clipped properly. However, it is important that the rectangle you pass as the dstRect is the same size as the srcRect, since the dstRect is used to determine how much to copy - if the srcRect is smaller than the dstRect, the blitter will still try to copy the entire dstRect, which will result in random junk being placed in the dstRect.

==========

Storing the name of a DrawProc in a variable

One very useful but little-known trick is that you can store the name of one of SpriteWorld's DrawProcs in a variable, allowing you to write code that is independent of a particular DrawProc. For instance, if you draw directly to the screen, you may want to use a blitter to get the most speed, but you may also want to have the option of using the SWStdWorldDrawProc (CopyBits), should drawing directly to the screen be incompatible with a particular Macintosh. This is easily done when you use a

variable to store the name of your DrawProc, since you can write the rest of your code to use whatever DrawProc is in that variable, and it will work regardless of which DrawProc is used.

For instance, at the beginning of your program, you might test the depth and set the DrawProc variable to BlitPixie8BitRectDrawProc if you are in 256 colors, or to BlitPixieAllBitRectDrawProc if you are not. You also then have the freedom to change the variable later if you decide you need to use CopyBits (SWStdWorldDrawProc) for some reason. For this to work, your variable must be of type DrawProcPtr. Here's an example of setting a variable to a DrawProc and then calling the DrawProc stored in the variable to copy the SpriteWorld's background area to the work area:

```
DrawProcPtr     myDrawProc = BlitPixie8BitRectDrawProc;

(*myDrawProc)(spriteWorldP->backFrameP, spriteWorldP->workFrameP,          &spriteWorldP->backFrameP->frameRect, &spriteWorldP->workFrameP->frameRect);
```

You can also use the variable when setting the DrawProc for a Sprite or for your SpriteWorld:

```
SWSetSpriteDrawProc(mySpriteP, myDrawProc);
SWSetSpriteWorldOffscreenDrawProc(gSpriteWorldP, myDrawProc);
```

This is quite useful, because you can test the depth once at the beginning of your program and determine which mask and rect DrawProcs to use, and later you can simply set your Sprites to use those DrawProcs, rather than having to test the depth each time you create a new Sprite.

==========

Accessing a Sprite's SpriteRec from its DrawProc

Are you creating a custom DrawProc, where you need to be able to access the SpriteRec of the Sprite being drawn? While the SpritePtr isn't passed to the function, you can access it from a global variable, called gCurrentSpriteBeingDrawn. Before any Sprite's DrawProc is called, this variable is set to the Sprite being drawn, so you can read that variable if you wish. But first, you need to gain access to the variable by adding this line to your code:

```
extern SpritePtr                gCurrentSpriteBeingDrawn;
```

Then you can read the gCurrentSpriteBeingDrawn variable from within your DrawProc to gain access to the Sprite that is being drawn.

==========

How to handle the "stats" area of your game

Something SpriteWorld hasn't supported very well in the past are games that need a separate "stats" portion of the screen, such as the right hand section of the screen in Apeiron which displays information such as the current level, number of lives left, etc. Naturally, you could just create a GWorld for your offscreen area and use CopyBits to transfer your images from it to the screen, but CopyBits is slow, and it would be nice to be able to use a blitter to do this. Or you could create a

separate SpriteWorld for your stats area, but this uses up more memory than necessary, since you won't need both a background and work area, unless you plan on animating Sprites in your stats area.

The solution is to call SWCreateWindowFrame to create a special Frame structure just for the stats portion of your Window, and SWCreateFrame to create the offscreen area, if you need one. SWCreateWindowFrame doesn't actually create a GWorld, but simply stores important information about the window in the Frame structure. This data is then used when you call any of SpriteWorld's blitters to draw in your stats area. In fact, without creating this special Frame for your stats area, it would be impossible to use one of SpriteWorld's blitters to draw in that area, since you need to pass a destFrameP to the blitter, and if you pass the SpriteWorld's normal windowFrameP, it won't work, because the blitter will clip everything to that Frame's frameRect.

Below is an example that creates a windowFrame for the stats area that is 40 pixels tall and as wide as the SpriteWorld's worldRect, and is positioned right above the worldRect . An offscreen area is also created to store the contents of the stats area offscreen. This offscreen area is then filled with a red rectangle, and its contents are then copied to the screen using the SpriteWorld's current screenDrawProc. This code example was copied from the Stats.c file of Shark Attack.

```
FramePtr        gStatsWindowFrameP;
FramePtr        gStatsBackFrameP;
...
void    InitStats( void )
{
        Rect            statsRect;
        OSErr           err;

                // put statsRect right above SpriteWorld's worldRect
        statsRect = gSpriteWorldP->windRect;
        statsRect.bottom = statsRect.top;
        statsRect.top -= 40;        // 40 = height of stats area

                // Create the windowFrame
        SetPort(gWindowP);      // Set port to window first - very important!
        err = SWCreateWindowFrame(&gStatsWindowFrameP, &statsRect, 0);
        FatalError(err);

                // Create the offscreen Frame
        OffsetRect(&statsRect, -statsRect.left, -statsRect.top);
        err = SWCreateFrame(gSpriteWorldP->mainSWGDH, &gStatsBackFrameP,
                &statsRect, gSpriteWorldP->pixelDepth);
        FatalError(err);

        SWLockWindowFrame(gStatsWindowFrameP);
        SWLockFrame(gStatsBackFrameP);

        SetGWorld(gStatsBackFrameP->framePort, nil);
        ForeColor(redColor);
        PaintRect(&gStatsBackFrameP->frameRect);

                // Copy offscreen contents to screen
        SetGWorld(gStatsWindowFrameP->framePort, nil);
        (*gSpriteWorldP->screenDrawProc)(gStatsBackFrameP, gStatsWindowFrameP,
                &gStatsBackFrameP->frameRect, &gStatsWindowFrameP->frameRect);
}
```

Complete documentation for the SWCreateFrame and SWCreateWindowFrame routines are available in Inside SpriteWorld.

Don't forget to lock your Frames after creating them, calling SWLockWindowFrame for your windowFrame, and SWLockFrame for any offscreen Frames. Also, to set the port to one of your Frames, simply use: SetGWorld(myFrameP->framePort, nil);

You can get a Frame's size by looking at the myFrameP->frameRect variable. If it is a windowFrame, the frameRect variable indicates the frame's position in coordinates local to the window, so the frameRect's top and left sides may not be 0.

==========

Making the frames of a sprite smoothly come to a stop when the sprite stops moving

Several people have asked how to control the animation of a sprite's frames so that when the sprite (a person) stops moving, the frames smoothly return to a standing position, rather than immediately jumping to the stopping frame. There are many ways one could do this. The method described below is just one way. It may or may not be the way you want to do it, but will hopefully at least give you ideas for how you could implement your own method, if you choose not to use the method described below. Let me say also that this method has not been tested, so it may have problems I have overlooked.

First, I would have the actual movement of the sprite controlled by the frameProc, rather than by a moveProc and moveDeltas. I would set up the frameProc so that it looks at which frame is being rendered, and on that basis decides how much to move the sprite forward. In this way you can have the figure move more naturally; you can control the forward movement so that it never looks like the figure's feet are sliding on the ground. You can then control how fast the figure is moving in two ways: by increasing or decreasing the frameRate, and by changing how much the figure moves at certain frames. The first of these is obvious; as an example of the second, consider a single frame where the figure has one leg extended ahead of the other. If the figure is walking from left to right, this frame might be placed at the same point on the screen as the frame before it; the back foot is "planted", so the left edge of the frame should appear at the same point as the previous frame. If the figure is running, you can change that a little. You could place the frame somewhat to the right of the previous frame, and a little higher on the screen. Then it will look like the figure has both feet off the ground, and is "flying" in mid running stride.

To control the figure's speed, you could attach a variable to the sprite (you could use one of the moveDeltas, since you may not be using them for anything else). The exact amount the figure moves would be different for each frame (as I described above), but the frameProc would look at this variable to see "in general" how fast the figure is moving, and to calculate the actual movement for each particular frame.

To effect acceleration and deceleration, you could attach a second variable. This variable, positive for acceleration and negative for deceleration, would be an amount by which the "speed"

variable should change with each frame. Thus, when the player took their finger off the arrow key, the acceleration/deceleration variable would get a negative value, so that the movement variable is reduced with each frame, and the figure slows down. You might also want to adjust the frameTime of the sprite, so that the frames change faster when the sprite is speeding up, and change slower when the sprite is slowing down.

You want the figure to come to a stop from any point in its stride, and to come to a stop with the feet in a sensible position for stopping (not in mid stride, with one leg extended in front). You could perform a test in the frameProc that says "if the speed variable is below a certain threshold, and the current frame is a good one for stopping at, then stop immediately".

For "fine control" of the sprite's movement like this, you'll need to use something other than simple integer arithmetic to add the acceleration/deceleration variable to the speed variable. One method is to use fixed point variables, another would be to using floating point variables.

This method may be more complex that what you're looking for, and would only be appropriate for certain types of games. One game that comes to mind is Prince of Persia; it has very smooth control of the frame movement, so that the character actually looks like it's running, instead of looking like a sprite with a bunch of frames that are being animated, and don't have any relationship to the actual speed of the sprite . However, it also has its drawbacks; when you jump, the character doesn't jump until the frames get to the right place for the jumping to look natural. The same goes for stopping. This can make the animation look nicer, but also gives you less control. So exactly how you implement your frame-changing scheme will depend largely on the needs of your particular game.

==========

Achieving a 3D effect, part 1 -- making the Sprites that are higher on the screen appear further away by having them drawn behind the Sprites that are lower and “closer”:

For Sprites that are stationary, this is no problem. Just add the “furthest” Sprites to the Layer (or their Layer to the SpriteWorld) first, and they will be drawn first when SpriteWorld renders the screen image. Sprites (and Layers) are drawn in the order in which they were added, so the later ones will be drawn over the earlier ones when there is an overlap.

For Sprites that move, the situation is more complicated. One way to handle this is to sort the Sprites in a Layer after SWProcessSpriteWorld (which updates the positions of all the Sprites) has been called, and before SWAnimateSpriteWorld (which actually draws the Sprites in their new positions) is called. You will want to sort the Sprites in order of their height on the screen, so that the highest Sprites have the earliest positions in the Layer. Most sorting algorithms involve iteratively swapping the positions of a pair of elements in the list being sorted. The SpriteWorld routine SWSwapSprite() can be used to perform this swapping. The sorting algorithms suitable for this situation include selection sort, insertion sort, bubble sort, and Shellsort. Using Quicksort will be more problematical, since Sprites are kept in a linked list rather than an indexed array, but it’s probably doable. Naturally, if you have many Sprites in the Layer, you will want to use the fastest sorting algorithm you can.

Another situation occurs if you have a single moving Sprite and several stationary Sprites, and you want the moving Sprite to maintain a correct 3D position as it moves among the stationary Sprites. An example would be a Sprite-person moving around in a forest of Sprite-trees. The person will have to change which trees it is drawn behind or in front of, depending on its height on the screen. To handle

this situation, first the stationary Sprites must be added to their layer in order of their height, as discussed above. Then the moving Sprite, which is added to the same Layer, can be given a collideProc to be called whenever it “collides” (overlaps) with one of the stationary Sprites. This collideProc would check the height of the moving Sprite relative to the stationary Sprite, and change its position in the Layer accordingly, using SWInsertSpriteBeforeSprite() and SWInsertSpriteAfterSprite():

```
void movingSpriteCollideProc(
    SpritePtr movingSpriteP,
    SpritePtr stationarySpriteP,
    Rect* sectRect)
{
    if ( (movingSpriteP->destFrameRect.bottom >= stationarySpriteP->destFrameRect.bottom &&
        movingSpriteP->prevSpriteP != stationarySpriteP )
    {
        SWRemoveSprite( mySpriteLayerP, movingSpriteP );
        SWInsertSpriteAfterSprite( mySpriteLayerP, movingSpriteP, stationarySpriteP );
    }
    else if ( movingSpriteP->destFrameRect.bottom < stationarySpriteP->destFrameRect.bottom
          && movingSpriteP->nextSpriteP != stationarySpriteP )
    {
        SWRemoveSprite( mySpriteLayerP, movingSpriteP );
        SWInsertSpriteBeforeSprite( mySpriteLayerP, movingSpriteP, stationarySpriteP );
    }
}
```

In either of the situations discussed above (multiple moving Sprites or a single moving Sprite), it is likely that you will need some kind of 3D collision-detection routine. Otherwise there will be nothing to prevent a Sprite from passing **through** an overlapping Sprite as it changes its height on the screen.

==========

Fractional Movement Deltas and Moving a Sprite in Any Arbitrary Direction

For further discussion of the principles involved here, I strongly recommend Tricks of the Mac Game Programming Gurus (Hayden Books, (800) 428-5331, hayden@hayden.com).

For many types of games, you will want a Sprite to be able to move at a constant speed in any given direction, and for its speed to remain constant as it changes direction. To do this, fractional deltas and fractional position holders must be used. There are many ways to approach this problem; I am only giving one example here.

To get started, let’s first define a simple “FixedPoint” data type:

```
typedef short FixedPoint;
```

As you can see, this data type is simply a short. However, we will store its fractional part in the lower 4 bits of the short, allowing an accuracy of 1/16 of a pixel, which is good enough for our purposes. A couple of things should be note about this particular implementation of a fractional data type. First, with PPC-native projects, it will be more efficient to use floating point math. Second, you may need to

use a different data type, such as the Fixed type provided by the Mac Toolbox, if your game is scrolling. The position of a Sprite within the potentially large worldRect of a scrolling game may require more than the 12 bit whole number portion of this FixedPoint type. (Since this number is signed, the maximum value is 0x7FF, and its range in decimal is -2048 to 2047.) For the most part, this data type is very easy to work with. Two FixedPoint variables can be added or subtracted as if they were simple shorts, and the result will be a valid FixedPoint. A FixedPoint can also be divided by or multiplied by a short with no special handling.

A couple of functions are needed for converting our FixedPoint data:

```
/******************** FixedPoint2Short ********************/
short FixedPoint2Short( FixedPoint theFixed )
{
    Boolean         isNegative;
    short           theShort;

    theShort = theFixed;
    isNegative = theShort < 0;

    if ( isNegative )
        theShort = -theShort;

    theShort >>= 4;

    if ( isNegative )
        theShort = -theShort;

    return theShort;
}

/******************** Short2FixedPoint ********************/
FixedPoint Short2FixedPoint( short theShort )
{
    Boolean         isNegative;
    FixedPoint      theFixed;

    theFixed = theShort;
    isNegative = theFixed < 0;

    if ( isNegative )
        theFixed = -theFixed;

    theFixed <<= 4;

    if ( isNegative )
        theFixed = -theFixed;

    return theFixed;
}
```

The Sprite we'll use for our example here is a "bug". (We could have made it a "spaceship", but that's just **too** unoriginal!) We'll say that we want our bug to be able to move in 10 different directions, so we'll create a Sprite with 10 Frames, each one showing the bug facing in a direction rotated 36 degrees from the previous Frame (36x10 = 360 degrees). We'll also stipulate for this example that the direction of movement for our bug is controlled by the Sprite's current Frame index -- when the Frame of the Sprite is changed, a routine is called to change the movement deltas of the bug accordingly.

When creating the Sprite, we'll have to extend it with an external data structure, as discussed earlier in this file. Here is the data structure we'll use:

```
typedef struct BugRecord
{
   SpriteRec     bugSprite;
   FixedPoint     fixedPtPosHoriz;
   FixedPoint     fixedPtPosVert;
   FixedPoint     fixedPtDeltaH;
   FixedPoint     fixedPtDeltaV;
}BugRecord, *BugRecordPeek;
```

Here we create the Sprite (by cloning from a previously created "master" Sprite, in this case), attach its data structure, and start it on its way:

```
/******************** LaunchNewBug ********************/
void LaunchNewBug( Point startPoint, short direction )
{
   Ptr                 sStorage;
   SpritePtr         newBugSprite;
   BugRecord*       bugRecP;
   OSErr               myErr;

   sStorage = NewPtrClear( sizeof(BugRecord) );

   myErr = SWCloneSprite( mMasterBugSprite, &newBugSprite, sStorage );

   bugRecP = (BugRecordPeek)newBugSprite;

   SWSetSpriteDrawProc( newBugSprite, BlitPixie8BitMaskDrawProc );
   SWSetSpriteMoveProc( newBugSprite, BugMoveProc );
   SWSetSpriteMoveTime( newBugSprite, 0 );
   SWSetCurrentFrameIndex( direction );     // 0 to 9
      // our routine for setting the fractional deltas:
   SetDeltasFromFrame( newBugSprite );
   SWSetSpriteFrameTime( newBugSprite, -1 );
   SWSetSpriteVisible( newBugSprite, true );
   SWAddSprite( gBugSpriteLayerP, newBugSprite );

   SWSetSpriteLocation( newBugSprite, startPoint.h, startPoint.v );

      // initialize the fractional position holders
   bugRecP->fixedPtPosHoriz = Short2FixedPoint( startPoint.h );
```

```
    bugRecP->fixedPtPosVert = Short2FixedPoint( startPoint.v );
}
```

Next is the routine for setting the fractional deltas according to the current Frame index. A moving Sprite will have vertical and horizontal “components” to its motion. For example, with a Sprite moving at a 45 degree angle, the horizontal and vertical components are equal. For any given angle of motion, the vertical and horizontal components of motion are proportional to the sine and cosine, respectively, of that angle. The “vertComponent” and “horizComponent” arrays below refer to the vertical and horizontal components of our Sprite’s motion. These arrays have been filled by manually calculating the appropriate values for each frame of our Sprite. We have arbitrarily chosen to multiply the sine and cosine values by 10, so that they can be rounded to whole numbers.

The first Frame of our Sprite faces due left, so the values for vertComponent and horizComponent are obvious: vertComponent[0] is 0 and horizComponent[0] is -10. In the second Frame, the bug is rotated 36 degrees upward, so we whip out our calculator and find that the sine of 36 degrees is .58, which multiplied by 10 is 5.8, and rounded off is 6. We make this negative for the vertComponent array because we know that the vertical delta will have to be negative for an upward-moving Sprite. The cosine of 36 degrees is .81, so -8 is the value for horizComponent for the second Frame. The rest of the values are calculated for the rest of the Frames and entered into the arrays. These values between -10 and 10 are then converted into our actual fractional deltas by converting them to FixedPoint and dividing by “gSpeedValue”. gSpeedValue controls the actual speed of the Sprite. If it has a value of 10 (the same as the arbitrarily chosen multiplier above), then the Sprite will move the equivalent of one pixel in each frame of animation. Decreasing gSpeedValue increases the speed of the Sprite, and increasing it slows down the Sprite. This division can be done as ordinary integer division, because when the FixedPoint number is divided by a short, the lower 4 bits of the FixedPoint variable are filled with the fractional “remainder” of the division, and the result is a valid FixedPoint.

Instead of using the Short2FixedPoint() routine here, a slightly faster and more accurate method would be to manually precalculate the FixedPoint values corresponding to the sine and cosine calculations, and to fill the arrays with these values. This would require converting the decimal results of the trigonometric calculations into our hexadecimal FixedPoint format, and is left as “an exercise for the reader”.

```
/******************** SetDeltasFromFrame ********************/
void SetDeltasFromFrame( SpritePtr bugSpriteP )
{
    BugRecord*        bugRecP;
    static short      vertComponent[] = {0, -6, -9, -9, -6, 0, 6, 9, 9, 6};
    static short      horizComponent[] = {-10, -8, -3, 3, 8, 10, 8, 3, -3, -8};
    short             vertical,
                      horizontal;


    bugRecP = (BugRecordPeek)bugSpriteP;

    vertical = vertComponent[bugSpriteP->curFrameIndex];
    horizontal = horizComponent[bugSpriteP->curFrameIndex];

    bugRecP->fixedPtDeltaV = ( Short2FixedPoint(vertical)/gSpeedValue );
    bugRecP->fixedPtDeltaH = ( Short2FixedPoint(horizontal)/gSpeedValue );
}
```

Finally we have the bug's moveProc. Here the fractional delta values in the Sprite's data structure are added to the Sprite's current position. However, note that the Sprite's current position must also be kept track of in a pair of FixedPoint variables attached to the Sprite. This moveProc is telling the Sprite to move by some fraction of a pixel. Since this isn't actually possible, what happens is that the "theoretical" position of the Sprite is updated by a fraction of a pixel. When this theoretical position accumulates enough fractions so that it differs from the Sprite's real position by some whole number of pixels, then the Sprite is actually moved. Note that the fractional position holders were initialized when the Sprite was first activated.

```
/******************** BugMoveProc ********************/
void BugMoveProc( SpritePtr bugSpriteP )
{
   BugRecord*          bugRecP;
   FixedPoint          currFixedPtPosH,
                       currFixedPtPosV;
   short               moveH,
                       moveV;

   bugRecP = (BugRecordPeek)bugSpriteP;

   currFixedPtPosH = Short2FixedPoint( bugSpriteP->destFrameRect.left );
   currFixedPtPosV = Short2FixedPoint( bugSpriteP->destFrameRect.top );

   bugRecP->fixedPtPosHoriz += bugRecP->fixedPtDeltaH;
   bugRecP->fixedPtPosVert += bugRecP->fixedPtDeltaV;

   moveH = FixedPoint2Short( bugRecP->fixedPtPosHoriz - currFixedPtPosH );
   moveV = FixedPoint2Short( bugRecP->fixedPtPosVert - currFixedPtPosV );

   SWOffsetSprite( bugSpriteP, moveH, moveV );
}

========================
```