SpriteWorld — Tiling

by Vern Jensen

What is Tiling?

Tiling is a method of using tiles, which are small rectangular images that are all the same size, to draw the background of your animation. For instance, one tile might be a wall, another tile some water, and another some grass. Then you can arrange these tiles however you like to form a background complete with walls, grass, and water.

Tiling has many advantages over using a picture or pattern as the background for your game. For one, it saves memory, since you can make a game with multiple levels to explore, without having to create a separate picture for each level; instead, you can simply rearrange the tile layout. Another advantage is that your sprites can interact with the tiles in a way that would be impossible if you used a picture as your background. For instance, you can have walls which your Sprites can not run over, spikes which kill your player, transporter tiles that beam the Sprite to another location, and so on. And with SpriteWorld's ability to have tiles that change images (so you could have a waterfall that actually moves, or a candle that flickers), and the ability to mark sprites to appear underneath tiles (so your Sprite could move behind a tree, etc.), the advantages of tiling become even greater.

Of course, there are some situations in which you may not want to use tiling; it may be a better idea in some circumstances to use a picture as your background instead of tiles. It is up to you to decide whether it would be better to use tiling in your game or not. However, if you are going to make a scrolling game, then tiling is recommended, since using a picture as the background for a scrolling game could take up a lot more memory than tiling would, and would considerably limit the size of the scrolling area. Note, however, that tiling is not limited to scrolling games - it could be very useful in many types of non-scrolling games, such as multi-level platform games.

Getting Started

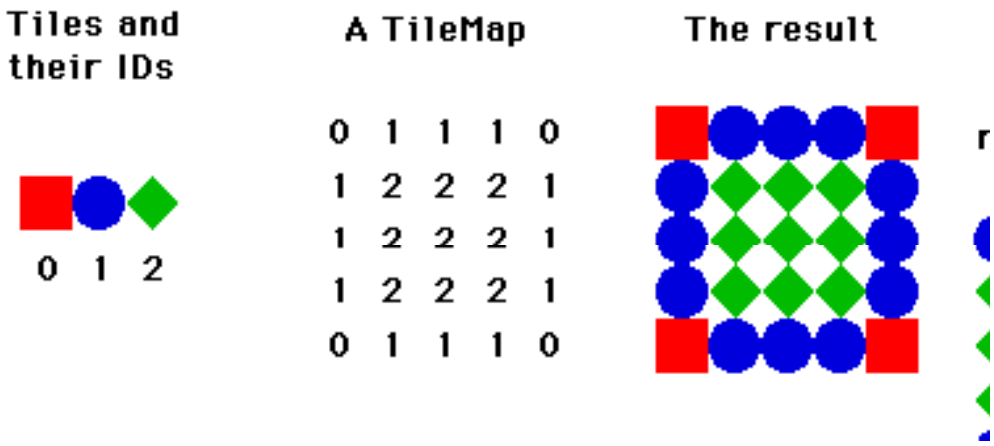
If you are making a scrolling game, then the size of the offscreen area must be evenly divisible by the size of your tiles, so that your tiles fit perfectly in the offscreen area without being clipped. If necessary, you can make the offscreen area slightly larger than what the user sees on the screen in order to fulfill this requirement. See SWCreateSpriteWorld for more information. However, this is not necessary unless you are making a scrolling game.

In order to use the Tiling routines, you must first call SWInitTiling, which allocates memory for the various tiling data structures that SpriteWorld uses. You should generally call SWInitTiling only once in the beginning of your program, after the SpriteWorld has been created. It must be called before you can load any tiles. After calling SWInitTiling, you should either create or load the TileMap (which is a "map" of where the tiles should be drawn), which can be accomplished with either SWCreateTileMap or SWLoadTileMap. Then you should load the tiles, lock the SpriteWorld (which also locks the tiles), and then draw the tiles in the background of the SpriteWorld with SWDrawTilesInBackground. It is not important whether you create the TileMap or load the tiles first, just as long as both are done before the

animation starts.

To create tiles, you simply draw small rectangular images that fit together when put side by side, and store these images in either a CICN or PICT resource. The CICN format is useful for quickly testing a tile, but eventually you will want to store all your tiles in one or more PICT resources, since you can have multiple tiles in a single PICT, which saves memory and speeds up loading. You must decide what size you want each tile to be, and then make all your tiles that size.

When the tiles are loaded, you assign each tile a unique ID, starting at 0. You then use these IDs when setting up your TileMap, which is simply a “map” of where each tile should be drawn. The TileMap is a two-dimensional array that is created by SWCreateTileMap or loaded with SWLoadTileMap.



Converting Sprite Coordinates to Tile Coordinates

Often you will need to convert a Sprite's coordinates to TileMap coordinates in order to see what tile is under a particular part of the Sprite. To do this, simply divide the Sprite's row and column by the tile width and tile height. Here's an example that checks to see what tile is under the top-right pixel of a sprite:

```
tileRow = spriteP->destFrameRect.top / gSpriteWorldP->tileHeight;
tileCol = (spriteP->destFrameRect.right-1) / gSpriteWorldP->tileWidth;
tileID = gTileMap[tileRow][tileCol];
```

Keep in mind that you must subtract 1 from the right or bottom sides of the destFrameRect before doing the division, since the right and bottom sides specify a border enclosing the Sprite; this border is not part of the Sprite's image.

Multiple Tile Layers

SpriteWorld gives you the ability to use more than one tile "layer", meaning you can have more than one TileMap added to the SpriteWorld at once. All layers must be the

same size and use tiles of the same size, and all layers will scroll at the same speed. This means you can't use multiple tile layers to achieve parallax scrolling, where the various tile layers scroll at different speeds. Rather, the multiple tile layers are provided for a different purpose.

Let's consider that a vine hangs down off a platform, and your Sprite walks behind the vine but in front of the background. Normally, you would give the tile a partial mask that indicates the portion of the tile that contains the vine. However, this poses a problem if you wish to place the vine in front of lots of different backgrounds. This means you'd have to draw two versions of each background; one without the vine (for normal use), and one with the vine. And if you have several different vine tiles making it twist and turn differently in each tile, then you have to make multiple versions of each background. Obviously, this would take up too much memory.

The solution is to keep the vine out of the background by moving it into its own tile. The vine tile is then placed in a tile layer higher than the background tile layer, and when the animation is run the vine will be drawn in front of the background. Problem solved - no longer do you have to make multiple versions of each kind of background; you just leave your background alone, and only one tile for each part of the vine is needed.

This solution does not, of course, come without its drawbacks. The drawback here is speed; now as you scroll, not only do the tiles in the bottom layer that are coming into view need to be drawn, but any masked tiles in higher layers also need to be drawn. And as you may know, drawing masked tiles or Sprites is quite a bit slower than drawing unmasked tiles or Sprites.

Another speed penalty is that when you change one of the tile images with SWChangeTileImage or SWDrawTile, all of the tiles in that row and column must be redrawn. So if our vine is hanging over a background and you change the vine to make it sway with the wind, the background is first redrawn and then the vine is redrawn. This means that changing tile images when multiple layers are used is considerably slower than changing images when only one layer is used.

There are, of course, ways of minimizing the slowdown while scrolling. One way is to keep your number of tile layers as low as possible - try to use only 2 or 3 tile layers if you can. (SpriteWorld allows a maximum of 10 tile layers.) However, the main way to keep things going as fast as possible is to place as few masked tiles in the higher layers as possible, since drawing these masked tiles is what makes the scrolling go slower.

There is, however, a small speed benefit to using multiple tile layers: when drawing masked tiles in layers higher than the background layer, the standard tileMaskDrawProc is used, rather than the slower partialMaskDrawProc. This means that if your Sprite moves behind a masked tile in a layer higher than layer 0, the masked tiles in that layer will be drawn over the sprites faster than they would be if you used a partially masked tile in layer 0 to specify which part of the tile should be drawn above the sprite.

To install several tile layers, simply call SWInstallTileMap several times, passing it

each tileMap and the number of the tile layer you wish to install it in. Use SWSetSpriteLayerUnderTileLayer to tell SpriteWorld which Sprite Layers should be under which tile layers as far as drawing order is concerned.

Use tileIDs of -1 in your TileMaps whenever you don't want any tile to be drawn in that location, to let whatever is underneath show through. Generally you should only use tileIDs of -1 in TileMaps that are in layers higher than 0. The only time you may use tileIDs of -1 in tile layer 0 is when a background is added behind your tiles, as is described in the section below. When this is the case, using tileIDs of -1 in layer 0 is perfectly fine, since you have something behind layer 0 which will show through when no tiles are in layer 0.

There may be some cases when you want to place a solid tile (that is, an unmasked tile) in layers higher than layer 0. Normally, this would result in inefficiency, since SpriteWorld would first draw the tiles in the lower layers, and then the solid tile in the higher layer would completely erase the tiles drawn in that row and column earlier. To avoid this, place tileIDs of -2 in that row and column of all TileMaps below the TileMap containing your solid tile. A value of -2 tells SpriteWorld that not only is no tile in that location, but no background should be drawn in that location, even if an extraBackFrame is added behind your tiles. This allows SpriteWorld to not waste time by drawing tiles which will only be erased anyway.

Adding a Background Behind Your Tiles

It is possible to add a background behind your tiles that shows through wherever your bottom-most tileMap (tile layer 0) doesn't have Tiles. Normally, the tileMap should be completely filled with valid tileID values, but when a background is added behind your Tiles, you may use a tileID value of -1 in your bottom tile layer whenever you don't want a Tile in that location so the background can show through.

The background can be any picture or pattern and is repeated automatically, just like the Mac toolbox function FillCRect repeats a pattern to fill a rectangle. This means the background does not need to be as large as your scrolling world or window. The background can be any size, provided it is at least as large as your tile size. Its width and height do not need to be evenly divisible by your tiles' width and height.

To create a background behind your tiles, call the SWCreateExtraBackFrame function. When an extraBackFrame is installed, it will be used to draw the areas of your bottom tile layer that have tileID values of -1. It is also used to draw the background behind masked tiles in the bottom tile layer. This is because when you have a masked tile, and an extraBackFrame is installed, the tile's mask is used to determine which part of the tile should be drawn on top of the background. When this is the case' the SpriteWorld's tileMaskDrawProc is used to draw the tile, rather than the partialMaskDrawProc, since the masks specify the entire visible part of the tile.

How to Make Tiles that Appear Above Sprites

Sometimes you may wish to have Sprites that move under certain parts of your scenery, such as a man walking under a tree, beneath a bridge, or behind a waterfall. Let's say we want the following diamond tile to appear in front of the red background, producing the result on the right:

Now we want any Sprites that move across this tile to appear behind the diamond but in front of the red background, like so:

Now there are two ways of doing this. One way would be to use multiple tile layers and place the red tile in layer 0, and the diamond tile in layer 1, and give the diamond tile a mask so it doesn't completely erase the red tile when it's drawn. However, another way would be to combine the diamond tile and the red tile as shown in the first illustration and to create a partial mask that tells SpriteWorld which part of the tile should be drawn above the Sprite:

By using this method, only one tile layer is needed, but we can still have portions of our background drawn over our Sprites. The mask in this example is called a partial mask because it covers only part of the tile. Normally, SpriteWorld requires that the unmasked portion of a tile or Sprite be white, since this allows SpriteWorld's blitters to go faster. When the unmasked portions are not white, as in the example above, a slower partial mask blitter must be used to draw the tile when it is drawn above a Sprite. Partially masked tiles may only be used in tile layer 0; all other tile layers must have whole masks or no mask at all.

To set the drawProc for drawing partially masked tiles when a Sprite moves over them, call SWSetPartialMaskDrawProc. To set the drawProc for drawing normal masked tiles, where the mask covers the entire tile, and the rest of the tile is white, use SWSetTileMaskDrawProc. You would use this for tiles such as the one below, which would typically be placed in tile layers higher than 0:

I say you would typically place tiles such as the one above in layers higher than 0 because there is one case where you could place it in layer 0: when a background is added behind your tiles. That's because when you have a background behind your tiles, you no longer have to fill each tile in layer 0 completely - you can leave holes, which allows the background to show through in those sections.

It is important to point out that when a background is added behind the tiles, partially masked tiles are no longer allowed in layer 0 - you now must use whole masks, where the unmasked portion of the tile is white. In summary,

1) If you have partially masked tiles in layer 0, and have not added a background behind your tiles, you should call SWSetPartialMaskDrawProc to set the drawProc for drawing the portion of these tiles that appears above Sprites when the Sprites move over those tiles.

2) If you have more than one tile layer, or have added a background behind your tiles, you should call SWSetTileMaskDrawProc to set the drawProc that is used to draw the masked tiles in all layers higher than layer 0, or to draw the masked tiles in layer 0 when a background is added behind your tiles.

It is perfectly fine to use a combination of both approaches. That is, you can have both partial masks in layer 0 and masked tiles in higher layers. Remember, however, that partially masked tiles are only allowed when no background has been added behind your tiles.

Controlling which Sprites are Drawn Behind Tiles

SpriteWorld allows you to control which Sprites should be drawn under tiles and which should be drawn above them. That way, you could have one Sprite move behind a tree while some birds fly above it. Or you could have some cars race under a bridge, while others race across it. To handle this, simply call SWSetSpriteLayerUnderTileLayer. This function tells SpriteWorld that an entire SpriteLayer should appear underneath a certain tile layer.

You can also control which Tiles are drawn above Sprites by the Layer the Tiles are placed in. For instance, if you have a SpriteLayer set under Tile Layer 1, placing Tiles in Layer 0 will cause them to be drawn under the Sprites, but placing the same Tiles in Layer 1 will cause them to be drawn above the Sprites.

In addition, when an extraBackFrame is used, any SpriteLayer set to be drawn under Tile Layer 0 will be drawn under the entire tile layer. However, when an

extraBackFrame is not installed, the Sprites under Tile Layer 0 will only be drawn under the partially masked tiles, but above the rest of the tile. That's because when an extraBackFrame is not installed, the unmasked portion of partially masked tiles take the place of the extraBackFrame for providing the background that is behind layer 0.

Tips for Faster Speed

For maximum speed during the animation, you should separate the groups of tiles that have no masks from those that do, since SpriteWorld can then simply check to see if a group of tiles has a mask, and if not, it can avoid trying to draw those tiles above any sprites. (This only applies to tiles that are loaded together from a single PICT resource.) It will not cause any problems to put tiles that have masks and tiles that don't into the same PICT resource, but it could make things slower than they would otherwise be during the animation.

Also, if you have any tiles in which the entire image of the tile should appear above the Sprites, you should group these tiles together into a single PICT and pass kSolidMask as the maskType for that pict. This will tell SpriteWorld that all of the tiles in that pict are "solid", and can be drawn with the SpriteWorld's offscreenDrawProc when they are drawn above Sprites, instead of having to use the slower tileMaskDrawProc. Using kSolidMask as the maskType also saves memory, since no mask has to be created for any of the tiles in that group. See the MaskType descriptions in SWLoadTilesFromPictResource for more information.

Variables used for tiling

There are certain variables in the SpriteWorldRec that are used for tiling that you should become familiar with. You may wish to access these directly at times, or you may just want to know what they are used for so you have a better understanding of how the tiling routines work. This is not a list of all the variables, but only of the ones that might be useful to you.

`FramePtr`	`*tileFrameArray`	The array where all of the tile images are stored. Each tile is stored in the element of this array corresponding to the tile's ID. (i.e. the image for tile ID 5 is stored in tileFrameArray[5])
`short`	`*curTileImage`	An array specifying the current image of each tileID. See SWChangeTileImage for more information.

Tiling Function Reference

These functions are not listed in alphabetical order, but rather in the order in which

they would most likely be used in an actual program. The following is a list of each function in the order in which they are documented, so that you can find the function you want quickly. They are also listed with the parameters as you would pass them to each function, so you can also look here for the order of the parameters in a particular function.

err = **SWInitTiling**(spriteWorldP, tileHeight, tileWidth, maxNumTiles)

SWExitTiling(spriteWorldP)

err = **SWCreateTileMap**(&tileMapStructP, numTileMapRows, numTileMapCols)

SWDisposeTileMap(&tileMapStructP)

SWLockTileMap(tileMapStructP)

SWUnlockTileMap(tileMapStructP)

SWInstallTileMap(spriteWorldP, tileMapStructP, tileLayer)

err = **SWLoadTileMap**(&tileMapStructP, resourceID)

err = **SWSaveTileMap**(tileMapStructP, resourceID)

err = **SWResizeTileMap**(tileMapStructP, numRows, numCols)

err = **SWLoadTileFromCicnResource**(spriteWorldP, tileID, cIconID, maskType)

err = **SWLoadTilesFromPictResource**(spriteWorldP, startTileID, endTileID, pictResID, maskResID, maskType, horizBorderWidth, vertBorderHeight)

SWDisposeTile(spriteWorldP, deadTileID)

err = **SWCreateExtraBackFrame**(spriteWorldP, &frameRect)

SWDisposeExtraBackFrame(spriteWorldP)

err = **SWSetPortToExtraBackFrame**(spriteWorldP)

SWLockTiles(spriteWorldP)

SWUnlockTiles(spriteWorldP)

SWSetTilingOn(spriteWorldP, tilingIsOnFlag)

err = **SWChangeTileSize**(spriteWorldP, tileWidth, tileHeight)

err = **SWSetTileMaskDrawProc**(spriteWorldP, drawProc)

err = **SWSetPartialMaskDrawProc**(spriteWorldP, drawProc)

SWSetSpriteLayerUnderTileLayer(spriteLayerP, tileLayer)

SWDrawTilesInBackground(spriteWorldP)

SWResetTilingCache(spriteWorldP)

SWDrawTile(spriteWorldP, tileLayer, tileRow, tileCol, tileID)

SWSetTileChangeProc(spriteWorldP, MyTileChangeProc)

SWChangeTileImage(spriteWorldP, tileID, newImage)

SWResetCurrentTileImages(spriteWorldP)

tileID = **SWReturnTileUnderPixel**(spriteWorldP, tileLayer, pixelCol, pixelRow)

didCollide = **SWCheckSpriteWithTiles**(spriteWorldP, srcSpriteP, kSWEntireSprite, &insetRect, startTileLayer, endTileLayer, firstTileID, lastTileID, fixPosition)

SWInitTiling

This function initializes the SpriteWorld's tiling data structures and prepares the SpriteWorld for tiling function calls.

```
OSErr SWInitTiling(SpriteWorldPtr spriteWorldP,                    short
tileHeight,      short tileWidth,                                  short
maxNumTiles)
```

`spriteWorldP`	The SpriteWorld that will be using the tiling.
`tileHeight`	The height of the tile images.
`tileWidth`	The width of the tile images.
`maxNumTiles`	The maximum number of tiles that may be loaded.

Description:

SWInitTiling prepares SpriteWorld for tiling function calls. Since it allocates the memory for several structures that are used by the tiling routines, you need to call SWInitTiling before you use any of the other tiling functions.

The dimensions of the tiles you will be loading, in pixels, are set by tileHeight and tileWidth. The maximum number of tiles you may load is set by maxNumTiles. After setting all of these values with SWInitTiling, you can not change them unless you call SWExitTiling and SWInitTiling again, in which case you would have to reload all the tile images, since they are disposed by SWExitTiling.

SWInitTiling returns an error if any memory allocation fails, or if the tiling has already been initialized for the SpriteWorld. The SpriteWorld must have already been created before this function is called.

See Also:

SWLoadTilesFromPictResource
SWLoadTileFromCicnResource

SWExitTiling

This function will dispose of everything created by SWInitTiling as well as all the tiles that have been loaded, releasing the memory they occupy.

```
void SWExitTiling(SpriteWorldPtr spriteWorldP)
```

`spriteWorldP`	The SpriteWorld containing the tiling data to be disposed.

Description:

The SWExitTiling function is used to dispose of all the tiling data previously created using SWInitTiling. The memory occupied by the various arrays, including all loaded tile images, will be released. You would normally not need to call this function, since it is called automatically by SWDisposeSpriteWorld. However, you may wish to call this if you want to dispose and reinitialize the tiling data while your program is running. This would be necessary if you wanted to use a different size of tiles, for example.

SWCreateTileMap

This function allocates memory for a TileMap that is numTileMapRows high by numTileMapCols wide.

```
OSErr SWCreateTileMap(TileMapStructPtr *tileMapStructP,            short
numTileMapRows,       short numTileMapCols)
```

`*tileMapStructP`	The address of a TileMapStructPtr variable that will be given a pointer to the TileMapStruct that is created.
`numTileMapRows`	The number of rows in the TileMap.
`numTileMapCols`	The number of columns in the TileMap.

Description:

The TileMap is a two-dimensional array that represents the “virtual background” you are creating. Each element in the TileMap array is a short which corresponds to a particular tile image. Your program must fill this array with tileID values corresponding to the tiles you want to appear in the background of your animation. Thus, if your background consists only of a single tile image repeated over and over, you would fill all elements of the TileMap with the tileID number for that tile. Make sure not to place values in the tileMap until after it is created with SWCreateTileMap. SWCreateTileMap automatically sets all elements of the TileMap to 0. Take a look at the Scrolling Demo and the Tiling Demo for examples of how to

create and set up the TileMap.

SWCreateTileMap allocates memory for a TileMap array that is numTileMapRows high by numTileMapCols wide. It also allocates memory for the TileMapStruct, which contains information about the TileMap, such as the number of rows and columns in the TileMap, the address of the TileMap array, and other information used internally by SpriteWorld. The TileMap is then locked, and a pointer to the TileMapStruct is returned to you in the tileMapStructP variable. In order to use the TileMap in a SpriteWorld, you must first install it in the SpriteWorld by calling SWInstallTileMap.

This is the definition of a TileMapStruct:

```
typedef struct TileMapStruct
{
    short          numRows;          // Number of rows
    short          numCols;          // Number of columns
    TileMapPtr     tileMap;          // Address of TileMap array
    Handle         tileMapDataH;     // Handle used by SpriteWorld
    Handle         arrayOfPointersH; // Handle used by SpriteWorld
    Boolean        isLocked;         // Is it currently locked?
} TileMapStruct, *TileMapStructPtr;
```

Since the memory for the TileMapStruct is allocated by SWCreateTileMap, you should not pass the address of a TileMapStruct to this function, but the address of a TileMapStructPtr variable, which will be set to point to the TileMapStruct that is created. You can access the data of the TileMap by using the tileMapStructP->tileMap variable. For example, the statement "tileMapStructP->tileMap[0][5] = 0;" sets row 0, column 5 of the TileMap to zero. You can also assign the tileMapStructP->tileMap variable to your own TileMapPtr variable for easier access, like so:

TileMapPtr myTileMap = myTileMapStructP->tileMap;

You can then use the myTileMap variable to access the TileMap, so the statement "myTileMap[0][5] = 0;" would have the same effect as "myTileMapStructP->tileMap[0][5] = 0;". Keep in mind that this pointer to the TileMap will remain valid only while the TileMap is locked. If you ever unlock the TileMap (via SWUnlockTileMap), make sure to reassign the value in the TileMapStruct to your copy of the TileMapPtr after you lock the TileMap again.

An error code will be returned if there was not enough memory to create the TileMap.

Note: you should pass an unused tileMapStructP to this function; if you pass one that points to a TileMapStruct, it will not get disposed, and you will lose your pointer to it. Instead, first dispose of the old TileMap with the SWDisposeTileMap function.

See Also:

SWLoadTileMap

SWDisposeTileMap

This function disposes a TileMap that was created previously by SWCreateTileMap or SWLoadTileMap.

```
void SWDisposeTileMap(TileMapStructPtr *tileMapStructP)
```

`*tileMapStructP` The address of a pointer to the TileMapStruct containing the TileMap to be disposed.

Description:

This function disposes the TileMapStruct as well as the TileMap itself. The TileMap is the only part of a SpriteWorld that is not automatically disposed by SWDisposeSpriteWorld. You must dispose of any TileMaps yourself. Just make sure that no SpriteWorld is still using the TileMap that was disposed, or it will crash when it tries to process the animation (unless Tiling for that SpriteWorld is turned off).

If you are disposing a TileMap that is currently installed in a SpriteWorld (even if it's not in use), it's a good idea to call SWInstallTileMap(spriteWorldP, NULL) so the SpriteWorld knows the TileMap has been disposed. This is not necessary if you are about to call SWInstallTileMap anyway to install a different TileMap in the SpriteWorld.

SWLockTileMap

This function will lock a TileMap that was previously unlocked via SWUnlockTileMap.

```
void SWLockTileMap(TileMapStructPtr tileMapStructP)
```

`tileMapStructP` A pointer to the TileMapStruct containing the TileMap to be locked.

Description:

This function locks a TileMap, preparing it for use in an animation. Since SWCreateTileMap and SWLoadTileMap automatically lock the TileMap, the only time you will need to call this function is if you previously unlocked the TileMap by using SWUnlockTileMap. Remember that after locking the TileMap, you should reassign the tileMapStructP->tileMap variable to any copies of that variable that you made previously, since the address of the TileMap array may have changed while it was unlocked. See SWCreateTileMap for more information about making copies of the tileMapStructP->tileMap variable.

SWUnlockTileMap

This function will unlock a TileMap, allowing it to move around freely in the heap.

```
void SWUnlockTileMap(TileMapStructPtr tileMapStructP)
```

`tileMapStructP` A pointer to the TileMapStruct containing the TileMap to be unlocked.

Description:

This function unlocks a TileMap, allowing it to move around freely in the heap. You might want to unlock your TileMaps (as well as your SpriteWorlds) when you are disposing a lot of memory, and want the memory manager to keep things nice and compact in the heap. However, the TileMap must be locked again before it can be used in the animation. Also, while the TileMap is unlocked, you must not access the TileMap data by using the TileMapStruct's tileMap variable.

Note: A TileMap will not be unlocked when you call SWUnlockSpriteWorld, even if the TileMap is installed in that SpriteWorld. You must use SWUnlockTileMap whenever you want to unlock a TileMap.

SWInstallTileMap

This function will install a TileMap in a SpriteWorld.

```
void SWInstallTileMap(SpriteWorldPtr spriteWorldP,
                      TileMapStructPtr tileMapStructP,              short
tileLayer)
```

`spriteWorldP`	The SpriteWorld you wish to install the TileMap in.
`tileMapStructP`	A pointer to the TileMapStruct containing the TileMap to be installed, or NULL, to remove a TileMap that is already installed.
`tileLayer`	A value between 0 and 9 specifying the Tile Layer the TileMap should be installed in.

Description:

This function installs a TileMap in the specified tile layer of a SpriteWorld. You need to install at least one TileMap before you can perform certain operations on that SpriteWorld that require a TileMap, such as calling SWDrawTilesInBackground or SWDrawTile. You may install the same TileMap in more than one SpriteWorld if you wish.

In addition to installing a TileMap before the animation begins, this function can be useful for switching between several different TileMaps while the animation is running. For instance, if your game has several different "rooms" that the player can walk between, you can simply call SWInstallTileMap when you want to switch from one room to another. This way you don't have to keep reloading the rooms, and the

program can remember changes made to the TileMap of a particular room even when the player walks into a different room.

When switching from one TileMap to another, remember to make any other function calls that are necessary, such as SWDrawTilesInBackground, and SWSetScrollingWorldMoveBounds, if your game is a scrolling game, and the new TileMap is a different size than the old one. Also make sure to call SWUpdateSpriteWorld after drawing the new TileMap in the background.

If you only have one tile layer, you should pass a value of 0 when calling SWInstallTileMap, since that installs the TileMap in the first layer. For more information about multiple tile layers, see the section entitled "Multiple Tile Layers" as the beginning of this document.

While it is not necessary install each TileMap in as low a layer as possible, it is recommended for speed. For instance, you could install only two TileMaps - one in layer 2, and one in layer 9, and leave the other tile layers empty, but this would be less efficient than if you installed those TileMaps in tile layers 0 and 1. The only time you should have tile layers that are separated by empty layers are when you temporarily "hide" a layer, as described below.

If you wish to "hide" or remove a tile layer, simply call SWInstallTileMap with a value of NULL for the tileMapStructP. This will "uninstall" the TileMap that was installed in that layer. Keep in mind that this does not dispose the TileMap; it simply uninstalls it from the SpriteWorld. If you wish to dispose it, you must call SWDisposeTileMap. After hiding a layer, remember to call SWDrawTilesInBackground to update the background, as well as SWUpdateSpriteWorld or SWUpdateScrollingSpriteWorld to update the screen.

If you remove all the TileMaps installed in a SpriteWorld, functions such as SWDrawTilesInBackground will draw nothing, since there is nothing to draw. This means that your background frame will contain whatever it contained previously, which could be random data, or tiles that were drawn there earlier. Therefore, it is recommended that whenever you hide a tile layer, you make sure there are either other tile layers beneath it with no "holes" in those layers, or make sure there is an extraBackFrame installed in the SpriteWorld, so when holes are encountered in your tile layers, they are filled with the extraBackFrame's contents.

SWLoadTileMap

This function will load a TileMap from a resource and lock it.

```
OSErr SWLoadTileMap(TileMapStructPtr *tileMapStructP,              short
resourceID)
```

`*tileMapStructP` The address of a TileMapStructPtr variable that will be given a

pointer to the TileMapStruct containing the TileMap that was loaded.

`resourceID` The ID of the TMAP resource to be loaded.

Description:

This function will load a TileMap from a TMAP resource in the current resource file. The TileMap will then be locked, and a pointer to the TileMapStruct will be returned to you in the tileMapStructP variable. In order to use the TileMap in a SpriteWorld, you must first install it in the SpriteWorld by calling SWInstallTileMap. For more information about TileMaps, see the documentation for SWCreateTileMap.

An error code will be returned if the requested TMAP resource could not be found, or if there was not enough memory to load the TileMap.

Note: you should pass an unused tileMapStructP to this function; if you pass one that points to a TileMapStruct, it will not get disposed, and you will lose your pointer to it. Instead, first dispose of the old TileMap with the SWDisposeTileMap function.

See Also:

SWCreateTileMap

SWSaveTileMap

This function saves the TileMap as a TMAP resource in the current resource file.

```
OSErr SWSaveTileMap(TileMapStructPtr tileMapStructP,                short
resourceID)
```

`tileMapStructP` A pointer to the TileMapStruct containing the TileMap to be saved.

`resourceID` The resource ID that this TileMap will be saved as.

Description:

This function will save the TileMap as a TMAP resource in the current resource file. If there is already a TMAP resource with the same ID as resourceID, it will be replaced with the new TileMap. SWSaveTileMap is provided so that you can design TileMaps with your own level editor, and then save them to be loaded later with SWLoadTileMap.

An error code will be returned if the TileMapStructP is NULL, if there isn't enough disk space available to save the TileMap, or if there isn't enough memory available to load the old TMAP resource that is going to be replaced with the new one. If this function fails because of a memFullErr, it is because there was already a TMAP resource with the same ID as the one you're trying to save, and there isn't enough memory to load the old TMAP resource in order to delete it from the resource file. When this happens, you can still save the TileMap by calling UniqueID('TMAP') and

saving the TileMap using the resource number returned by UniqueID. That way you won't lose the work you did designing the TileMap, and can then use a utility such as ResEdit to delete the old resource and change the resource ID of the new TileMap to the proper one.

SWResizeTileMap

This function changes the size of an existing TileMap.

```
OSErr SWResizeTileMap(TileMapStructPtr tileMapStructP,              short
newNumTileMapRows,  short newNumTileMapCols)
```

`tileMapStructP`	A pointer to the TileMapStruct containing the TileMap that is going to be resized.
`newNumTileMapRows`	The new number of rows for the TileMap.
`newNumTileMapCols`	The new number of columns for the TileMap.

Description:

This function will change the size of a TileMap, while keeping as much of the TileMap data intact as possible. If the TileMap is made larger, then the new area will be filled with 0's; if the TileMap is made smaller, then the data in the area that was trimmed will be lost. If the TileMap is already the requested size, this function does nothing. Make sure to change the size of your scrollingWorldMoveBounds to reflect the changes to the size of the TileMap, if the TileMap is currently used in a scrolling game. (See SWSetScrollingWorldMoveBounds for more information). Also check to make sure that the visScrollRect is still within those new boundaries.

After this function has been called, the tileMapStructP->tileMap variable will be different, since it will contain the address of the resized TileMap array instead of the old one. This means that if you have made any copies of this variable, you should update them after calling this function. See SWCreateTileMap for more information about making copies of the tileMapStructP->tileMap variable.

This function is mainly intended for use in a level editor, so that if you find that you are running out of room while editing your TileMap, or have too much room, then you can easily change its size, without having to dispose and recreate it, which would erase its data.

An error code is returned if there was not enough memory available to make the change (which might happen even if you make the TileMap smaller than it was), or if the tileMapStructP is NULL. If an error occurs, this function will leave your TileMap the way it was before this function was called.

SWLoadTileFromCicnResource

This function loads a tile from a cicn resource, placing it in the SpriteWorld.

```
OSErr SWLoadTileFromCicnResource(SpriteWorldPtr spriteWorldP,       short
tileID,                          short cicnResID,                   short
maskType)
```

`spriteWorldP`	The SpriteWorld that will receive the tile.
`tileID`	The ID number to be assigned to this tile.
`cicnResID`	The resource id of the cicn resource containing the tile image.
`maskType`	A value that indicates what type of mask should be created for the tile. For a description of these flags and their meaning see SWLoadTileFromPictResource below.

Description:

SWLoadTileFromCicnResource will load a single tile from a cicn resource into the SpriteWorld, assigning the tileID number to that tile. The tileID can be any number from 0 to maxNumTiles-1. MaxNumTiles is a value set with SWInitTiling that tells SpriteWorld how many individual tiles you are going to load. So if you set maxNumTiles to 2, then you may load up to two tiles, assigning them tileIDs 0 and 1.

If there is an existing tile with the same tileID as the tile you are loading, the old one will be disposed and replaced with the new one you are loading. This is useful if you want to change the look of tiles between levels, while having them perform the same functions.

Before you can load any tiles, the tiling data structures must have already been initialized with SWInitTiling. If this function hasn't been called yet, an error code will be returned. An error code is also returned if any memory allocation fails, if the cicn resource cannot be found, or if the tileID is out of range.

See Also:

SWInitTiling
SWLoadTilesFromPictResource

SWLoadTilesFromPictResource

This function loads one or more tiles from a PICT resource, placing them in the SpriteWorld.

```
OSErr SWLoadTilesFromPictResource(SpriteWorldPtr spriteWorldP,      short
startTileID,                    short endTileID,                    short
pictResID,                      short maskResID,                    short
maskType,                       short horizBorderWidth              short
vertBorderHeight)
```

spriteWorldP	The SpriteWorld that will contain the tiles.
startTileID	The ID number of the first tile to be loaded from the PICT.
endTileID	The ID number of the last tile to be loaded from the PICT.
pictResID	The resource id of the picture (‘PICT’) resource containing the tile images.
maskResID	The resource id of the picture (‘PICT’) resource containing the mask images of the tiles. If there are no masks for the tiles this parameter should be zero.
maskType	A value that indicates what type of mask should be created for the tiles. For a description of these flags and their meaning, see below.
horizBorderWidth	The width of the horizontal separation, in pixels, between each tile image in the PICT. See below for more information.
vertBorderHeight	The height of the vertical separation between each tile.

Description:

SWLoadTilesFromPictResource will load a series of tiles from a single PICT resource into the SpriteWorld, assigning each tile a unique ID starting with startTileID and ending with endTileID. The tileIDs can be any number from 0 to maxNumTiles-1. MaxNumTiles is a value set with SWInitTiling that tells SpriteWorld the maximum number of tiles you will load. So if you set maxNumTiles to 10, then you may load up to ten tiles, assigning them tileIDs 0 to 9.

If there is an existing tile with the same tileID as any of the tiles you are loading, the old tile will be disposed and replaced with the new tile of the same ID. This is useful if you want to change the look of tiles between levels, while having them perform the same functions.

Before you can load any tiles the tiling must have already been initialized with SWInitTiling. If it hasn’t, an error code will be returned. The PICT resource from which the tile images are taken can hold any number of tile images, but there are certain requirements as to how the images are laid out in the PICT graphic:

• The dimensions of the tiles are set by SWInitTiling. Naturally, the tile images must conform to these dimensions.

• The tile images can be laid out in any number of rows and columns. The images will

be read from left to right and top to bottom. The first image must begin at the top-left of the PICT.

• The tile images must be separated by a border whose width and height you specify in the horizBorderWidth and vertBorderHeight parameters. These values are user-defined to allow optimum alignment of the tile images. Both CopyBits and BlitPixie are fastest when the left side of the source rect is an even multiple of four. When assembling a graphic of tile images, you can adjust the horizontal separation of the tile images to achieve this alignment, and then set the horizBorderWidth parameter accordingly. You would usually set the vertBorderHeight to the same value as the horizBorderWidth, to make the tiles easy to edit in a drawing program, although it doesn't matter what you set this parameter to as far as speed is concerned.

The mask is used to define what part of the tile should appear above any sprites that are set to be drawn under the tiles. (See the section "How to Make Tiles that Appear Above Sprites" in the beginning of this file for more information.) The maskType parameter can currently be one of these values:

kNoMask — A value indicating that no mask should be created for the tiles. This means that none of the tiles loaded from this PICT will appear above any sprites.

kRegionMask — A value indicating that a QuickDraw region (RgnHandle) should be created for possible use as a mask for the tiles.

kPixelMask — A value indicating that an offscreen GWorld should be created, and used as a mask for the tiles. This GWorld will be the same bit depth as the tile image and is suitable for use with the various BlitPixie blitters.

kFatMask — This value is equivalent to kRegionMask + kPixelMask. This results in a tile that contains both of the above types of masks. This is useful if your application switches between using QuickDraw and a custom drawing routine at runtime.

kSolidMask — This is a special maskType that may only be used for tiles. Passing kSolidMask as your maskType indicates that the entire tile should be drawn above the sprites. If you use this as your maskType, then the SpriteWorld's offscreenDrawProc will be used to draw the tiles loaded from the PICT or CICN whenever any of those tiles need to be drawn above any sprites. This results in faster drawing as well as memory saved, since no mask needs to be created for any of the tiles loaded from the PICT or CICN.

An error code is returned by SWLoadTilesFromPictResource if the tiling data has not been initialized (with a call to SWInitTiling), if any memory allocation fails, if the picture resource cannot be found, if endTileID is equal to or greater than spriteWorldP->maxNumTiles (a value set by SWInitTiling), or if the bottom-right of the PICT is reached before the requested number of tile images have been loaded.

See Also:

SWInitTiling
SWLoadTileFromCicnResource

SWDisposeTile

This function will dispose of an existing tile.

```
void SWDisposeTile(SpriteWorldPtr spriteWorldP,                    short
deadTileID)
```

`spriteWorldP`	The SpriteWorld containing the tile.
`deadTileID`	The ID number of the tile to be disposed of.

Description:

The SWDisposeTile function will dispose of a tile. For the most part, programmers will have no need to use this function. It is not necessary to dispose of a tile before loading a new tile with the same ID; SWLoadTilesFromPictResource and SWLoadTileFromCicnResource will automatically dispose of old tiles as needed. SWExitTiling will dispose of all the tiles that have been loaded. If your application no longer has any use for a series of tiles, you can free up some memory by disposing of them; however, note that if the tiles were loaded with SWLoadTilesFromPictResource, then the memory savings will only be significant if you dispose of all the tiles that were loaded from a given PICT. As long as one tile loaded from a PICT is still in use, the GWorld created to hold that PICT's image will not be disposed of.

Δ WARNING Δ

You must not call SWDisposeTile on a tile that is part of a running animation, or SpriteWorld will crash when it tries to process a tile that no longer exists.

SWCreateExtraBackFrame

This function will create an extraBackFrame that is used to draw a background behind the tiles.

```
OSErr SWCreateExtraBackFrame(SpriteWorldPtr spriteWorldP,          Rect
*frameRect)
```

`spriteWorldP`	The SpriteWorld to contain the extraBackFrame.
`frameRect`	The size of the extraBackFrame.

Description:

SWCreateExtraBackFrame creates a Frame for storing the background image which is displayed behind the tiles, should you wish this feature. Whenever a tile with tileID -1 is encountered in the background tile layer, the appropriate piece of the

extraBackFrame will automatically be drawn in that tile's location. The extraBackFrame can be any size, as long as it is at least as large as your tiles. (So if your tiles are 40x40, the extraBackFrame must be at least 40x40.) The image in the extraBackFrame is repeated like a pattern, so that it fills the background area behind the tiles without needing to be as large as your TileMap.

After calling SWCreateExtraBackFrame, you should lock it by either calling SWLockSpriteWorld, or by calling:

SWLockFrame(spriteWorldP->extraBackFrameP).

The extraBackFrameP will be disposed of by SWDisposeSpriteWorld, or you can dispose it by calling SWDisposeExtraBackFrame. After locking the extraBackFrame, you should draw your background picture or pattern in it by calling SWSetPortToExtraBackFrame, and then drawing the image just as you would draw it in the backFrame of the SpriteWorld.

If you call SWCreateExtraBackFrame to create a new Frame without disposing the extraBackFrame you previously created, the old Frame will automatically be disposed and replaced by the new Frame. If there was not enough memory to create the new Frame, the old Frame will have been disposed, so don't try to use it! An error code will be returned when there is not enough memory to create the Frame.

When an extraBackFrame is installed, tiles which have masks in the bottom layer behave differently than they would otherwise. For more information, see the section at the beginning of the document entitled "Adding a Background behind your Tiles."

SWDisposeExtraBackFrame

This function will create an extraBackFrame that is used to draw a background behind the tiles.

```
void SWDisposeExtraBackFrame(SpriteWorldPtr spriteWorldP)
```

`spriteWorldP` The SpriteWorld containing the extraBackFrame.

Description:

This function disposes the extraBackFrame of a SpriteWorld, effectively removing it from the animation. Note that when there is no extraBackFrame, you should make sure there are no "holes" in your tileMap, since nothing will be drawn in those holes if there are holes, meaning random pieces from other parts of the map could show up there. Also keep in mind that if you dispose the extraBackFrame while an animation is running and wish it to disappear from the screen, you should call SWDrawTilesInBackground and SWUpdateSpriteWorld to redraw the tiles and copy them to the screen.

SWSetPortToExtraBackFrame

This function sets the port to the extraBackFrame so you can draw in it.

```
OSErr SWSetPortToExtraBackFrame(SpriteWorldPtr spriteWorldP)
```

`spriteWorldP` The SpriteWorld containing the extraBackFrame.

Description:

This function disposes the extraBackFrame of a SpriteWorld, effectively removing it from the animation. Note that when there is no extraBackFrame, you should make sure there are no "holes" in your tileMap, since nothing will be drawn in those holes, meaning random pieces from other parts of the map could show up there. Also keep in mind that if you dispose the extraBackFrame while an animation is running and wish it to disappear from the screen, you should call SWDrawTilesInBackground and SWUpdateSpriteWorld to redraw the tiles and copy them to the screen.

An error code is returned if there is no extraBackFrame installed, or if it is not locked.

SWLockTiles

This function locks all tiles that have been loaded.

```
void SWLockTiles(SpriteWorldPtr spriteWorldP)
```

`spriteWorldP` The SpriteWorld containing the tiles to be locked.

Description:

This function is used to lock all the tile images that have been loaded into the SpriteWorld. Since the tiles are automatically locked by SWLockSpriteWorld, you will not normally need to call SWLockTiles. It is provided in case you load more tiles after the SpriteWorld has already been locked, such as a different set of tiles for a different level. You must lock the tiles before you can use them in an animation.

SWUnlockTiles

This function will unlock all tiles that have been loaded.

```
void SWUnlockTiles(SpriteWorldPtr spriteWorldP)
```

`spriteWorldP` The SpriteWorld containing the tiles to be unlocked.

Description:

This function is used to unlock all the tiles that have been loaded into the SpriteWorld. You would normally not need to call this routine, since SWUnlockSpriteWorld automatically unlocks all the tiles as well. It is provided in case you for some reason want to unlock the tiles without unlocking the rest of the SpriteWorld. Note that you must not run an animation that uses tiles while those tiles are unlocked.

SWSetTilingOn

This function notifies SpriteWorld whether tiling is active.

```
void SWSetTilingOn(SpriteWorldPtr spriteWorldP,                    Boolean
tilingIsOnFlag)
```

`spriteWorldP`	The SpriteWorld.
`tilingIsOnFlag`	If true, tiling is turned on for the SpriteWorld; if false, tiling is turned off.

Description:

SWSetTilingOn notifies SpriteWorld whether tiling is currently active in a SpriteWorld's animation. When tiling is active, SpriteWorld automatically updates the portion that scrolls into view during a scrolling animation. Also, tiling must be active in order for sprites to be drawn under tiles.

When you first create a SpriteWorld, the tiling routines are inactive by default, since the tiling has not been initialized yet and is not ready to be used. However, SWInitTiling makes the tiling routines active, so it is not necessary to call SWSetTilingOn at the beginning of your animation.

This function was provided since some games will need to switch back and forth between screens that use tiling and screens that do not. An example of this would be a game that has a title screen with a picture as the background and maybe some sprites moving around , but then uses tiling once the game begins. Therefore, you would want to call SWSetTilingOn with a tilingIsOnFlag value of false before drawing the title screen, and then call SWSetTilingOn with a tilingIsOnFlag value of true before starting the game.

Functions which are for your use, such as SWDrawTile and SWDrawTilesInBackground will not be disabled when tiling is turned off. This function only turns on or off the automatic tile updating SpriteWorld does while scrolling or when sprites move behind masked tiles in a tileMap. If you forget to turn off tiling before going back to your title screen (or any other screen that doesn't use tiling), then you may experience problems such as Sprites being drawn behind tiles that are no longer there!

SWChangeTileSize

This function changes the current tile size.

```
OSErr SWChangeTileSize(SpriteWorldPtr spriteWorldP,
                       short tileWidth,
                  short tileHeight)
```

`spriteWorldP`	The SpriteWorld.
`tileWidth`	The tile width.
`tileHeight`	The tile height.

Description:

This function allows you to change the tile size without having to call SWExitTiling and then SWInitTiling again, which would dispose all of your tiles. Just remember that the tiles in your TileMap must match the width and height that you pass to this function, or the next call to SWDrawTilesInBackground could crash. Also remember that you should call SWDrawTilesInBackground and SWUpdateSpriteWorld after calling this function to draw the new tiles in the background and then copy that to the screen.

An error code is returned if this function fails.

SWSetTileMaskDrawProc

This function sets the drawProc to be used when drawing tiles with "whole" masks.

```
OSErr SWSetTileMaskDrawProc(SpriteWorldPtr spriteWorldP,
                            DrawProcPtr drawProc)
```

`spriteWorldP`	The SpriteWorld.
`drawProc`	The drawProc.

Description:

This function sets the drawProc to be used when drawing masked tiles in tile layers higher than 0, and is also used to draw masked tiles in layer 0 when an extraBackFrame is installed. You should not use a partialMaskDrawProc for this function, since the partialMaskDrawProcs are slower than the standard mask drawProcs. In fact, this function will return an error code if you try to use a known 8-bit partialMaskDrawProc with it. Instead, use the same mask drawProc with this function that you would use for drawing your Sprites.

SWSetPartialMaskDrawProc

This function sets the drawProc to be used when drawing the partially masked portion of a tile so that portion of the tile appears above a Sprite, and the rest of the tile appears behind the Sprite.

```
OSErr SWSetPartialMaskDrawProc(SpriteWorldPtr spriteWorldP,
                               DrawProcPtr drawProc)
```

`spriteWorldP`	The SpriteWorld.
`drawProc`	The drawProc to be used when drawing the partially masked portion of the tiles.

Description:

Not to be confused with SWSetTileMaskDrawProc, this function sets the drawProc to be used when drawing the masked portion of partially masked tiles in tile layer 0 when no extraBackFrame is installed. You may use any of the following drawProcs:

SWStdSpriteDrawProc	CopyBits with a mask region. The default tileMaskDrawProc. (Slow.)
BlitPixie8BitPartialMaskDrawProc	A special BlitPixie drawProc for tiles with partial masks.
BP8BitInterlacedPartialMaskDrawProc	Same as above, but interlaced.
BlitPixieAllBitPartialMaskDrawProc	Depth-independent BlitPixie for depths other than 8 bits.
BPAllBitInterlacedPartialMaskDrawProc	Depth-independent interlaced version.

You should use one of the BlitPixie drawProcs as the tileMaskDrawProc, if possible. Using CopyBits, which is the default drawProc, could really slow things down if you have even just a few Sprites that are under partially masked tiles in your animation.

SWSetSpriteLayerUnderTileLayer

This function tells SpriteWorld that all the Sprites in the specified SpriteLayer should be drawn under the tiles in the specified tile layer.

```
void SWSetSpriteLayerUnderTileLayer(SpriteLayerPtr spriteLayerP,    short
tileLayer)
```

`spriteLayerP`	The SpriteLayer.
`tileLayer`	The tileLayer.

Description:

This function tells SpriteWorld that all Sprites in the specified SpriteLayer should appear under the specified tile layer. The Sprites will also be drawn under all tile layers higher than the specified tile layer. For instance, if you have 3 tile layers, and you set your SpriteLayer under tile layer 1, the Sprites in that Layer will be drawn under both tile layers 1 and 2. (But not under layer 0.) See the section entitled "How to Make Tiles that Appear Above Sprites" for information on how to set up the tiles so they are drawn above the Sprites in your SpriteLayer.

By default, each SpriteLayer is set to be drawn under tile layer 10, and since layer 9 is the highest tile layer possible, this means that each SpriteLayer, if left unchanged, will always be above all the tiles, even if you have made use of all 10 tile layers.

When the Sprites are drawn beneath a tile layer, they are drawn under both masked and unmasked tiles. The exception to this is the tiles in layer 0 when no extraBackFrame is installed; in this case, the Sprites are drawn under the partially masked portion of tiles in the layer, but above the rest of the tiles. However, when an extraBackFrame is installed, the Sprites are drawn under all the tiles in layer 0 - including those with no mask.

When setting SpriteLayers under tile layers, if you set a SpriteLayer to be underneath a lower tileLayer than a previous SpriteLayer, SpriteWorld will ignore its tileLayer value and use the tileLayer value from the previous SpriteLayer. For instance, if one SpriteLayer is set to be under tile layer 1, and the next SpriteLayer in the list is set to be under tile layer 0, its value will be ignored, and the previous tileLayer value - layer 1 - will be used when drawing that SpriteLayer. Otherwise, strange things could happen in the animation where a Sprite is drawn under a particular tile layer, and a higher Sprite is drawn above it, but is drawn under that tile layer. This is why SpriteWorld makes sure each SpriteLayer is under a tile layer equal to or higher than the previous SpriteLayer.

You can take advantage of this as a feature, since it means that you can use tileLayer 0 as a "use the previous tileLayer" value. For instance, if a previous SpriteLayer were under layer 1, and the next SpriteLayer is under layer 0, SpriteWorld will ignore this value, since the second SpriteLayer must be equal to or above the first, and the second SpriteLayer will be under the same tile layer as the previous SpriteLayer - layer 1. This means you can set SpriteLayers to be under layer 0 whenever you want them to be under the same layer as the previous SpriteLayer.

SWDrawTilesInBackground

This function draws the tiles in the SpriteWorld's background.

```
void SWDrawTilesInBackground(SpriteWorldPtr spriteWorldP)
```

`spriteWorldP` The SpriteWorld using tiling.

Description:

This function will draw the tiles in the background at the current location of the visScrollRect. You will typically call this function before calling SWUpdateSpriteWorld. You would generally do this before starting the animation, or whenever the background tile pattern has been changed.

It is important that you become familiar with the SWResetTilingCache function before you use SWDrawTilesInBackground.

The current port is saved when this function is called and restored when it is done.

SWResetTilingCache

This function resets the SpriteWorld's tiling "cache", which is used to help speed up the tiling when only one tile layer is used.

```
void SWResetTilingCache(SpriteWorldPtr spriteWorldP)
```

`spriteWorldP` The SpriteWorld using tiling.

Description:

[Note: The following applies only to animations with only one tile layer, since the tiling cache can not be used with more than one layer. This means that SWResetTilingCache does not need to be used if you have more than one tile layer, although it won't hurt anything if you do call it.]

To speed things up, SpriteWorld has an array to keep track of which tiles have been drawn in the background. When new tiles are drawn in the background during a scrolling animation or by SWDrawTilesInBackground, the new tile values are compared with SpriteWorld's array, and only when the tileIDs are different (indicating that the tile in that location has changed) will the new tile be drawn. This helps speed up scrolling, so that when new tiles scroll into the screen, not every single one of them has to be redrawn. This optimization could also be useful in a non-scrolling platform type game where the player can move between different rooms that each have their own tileMap layout. When drawing the tiles for a new room, SpriteWorld is smart enough to redraw only the tiles that have changed since the previous room, simply by comparing the new tile values with the "cached" tile values it stored the previous time the tiles were drawn.

However, there are times when this optimizing feature can cause problems. One example would be if you loaded a new tile image for a particular tileID. Since SpriteWorld only compares the new tileID values with the cached tiling data, it won't know that the tile image has changed, and might not draw the new tile image when it should, if it detects that the tile has previously been drawn in that location in the

offscreen area.

Another example where this optimizing feature could cause problems would be if you drew something directly into the background that erases the tiles, such as a picture for the title screen of your game. SpriteWorld only keeps track of the tiles that were drawn last in the offscreen areas; it won't know it if you erase them with something else. Then the next time you use a tiling routine to draw the tiles in the background, SpriteWorld may not draw all the tiles, if it thinks that some of the tiles in the new layout have already been drawn in the offscreen area, when in reality you erased those tiles so you could draw the title screen for your game.

You can call SWResetTilingCache to avoid these problems. SWResetTilingCache will "reset" SpriteWorld's tiling cache so that any memory of previously drawn tiles will be erased. This means that all new tiles will be drawn properly in the background, instead of first being checked against SpriteWorld's cache to see if they really need to be drawn.

You should call this function whenever you draw something in the background area that erases tiles that were there previously. You should also call this function any time that you load new tiles with the same IDs as previously existing tiles that were used in the animation. (You might do this to change the look of the tiles for a different level.)

SWDrawTile

This function sets the specified element in the TileMap to the requested tileID and draws the corresponding tile, if it is currently visible on the screen.

```
void SWDrawTile(SpriteWorldPtr spriteWorldP,                    short
tileLayer,    short tileRow,                                    short
tileCol,      short tileID)
```

`spriteWorldP`	The SpriteWorld using tiling.
`tileLayer`	A value between 0 and 9 indicating the Tile Layer the tile should be drawn in.
`tileRow`	The row of the TileMap.
`tileCol`	The column of the TileMap.
`tileID`	The tile ID of the tile to be drawn.

Description:

SWDrawTile sets the specified element in the two-dimensional TileMap array to the value passed in tileID. The TileMap affected is the TileMap that is currently installed in tileLayer. If you have only one Tile Layer, you should pass a tileLayer value of 0.

SWDrawTile will also draw the tile, as well as any tiles under and above it in other Tile Layers. The new tile image will automatically be copied to the screen when the next frame is processed, even in non-scrolling animations. The image drawn is the current image of the tileID. (See SWChangeTileImage for more information.)

You should generally call SWDrawTile to change a value in the tileMap instead of changing the tileMap directly, since changing it directly will not draw the tile if it is visible on the screen. However, it is safe to set the value directly as long as you know for certain that the tile you are changing is not currently visible on the screen. If you're not sure, then calling SWDrawTile is the best idea, since it is smart enough not to draw the tile if it is not visible on the screen. You can set the tile directly by using:

myTileMap[row][col] = tileID;

Since SWDrawTile must redraw the tiles in every layer of the row and column passed to this function, it would be faster, if you are changing the tileIDs of the same row and column in several different layers, to change them first "manually" by setting the tileMap values directly, and then calling SWDrawTile once to get your changes updated on the screen. For instance, if you have a fire animation that is in front of an animated background, and both tiles change in the same frame, it would be faster to set the tileID values directly to change the tiles and call SWDrawTile once to redraw them both, than it would be to call SWDrawTile twice, since both calls would redraw both the background and fire tiles.

Δ WARNING Δ

This function does no error checking on the values you pass to it, so it is up to you to make sure that tileRow and tileCol are within the boundaries of the tileMap.

Also, this function changes the current port and does not set it back when done! This could mess up things like GetMouse, which return the mouse coordinates local to the current port. Therefore, use this function with caution if you use it outside the normal animation loop, where it could mess up your current port.

See Also:

SWChangeTileImage

SWResetCurrentTileImages

SWSetTileChangeProc

This function installs a tile-changing routine to be called each frame of animation.

```
void SWSetTileChangeProc(SpriteWorldPtr spriteWorldP,
                         TileChangeProcPtr tileChangeProc)
```

`spriteWorldP` The SpriteWorld using tiling.

`tileChangeProc`	A new tile-change routine.

Description:

The SWSetTileChangeProc function is used to specify a routine to be called each time SWProcessSpriteWorld processes a frame of animation. This routine may be used to change the images of some or all of the tiles in the TileMap by calling SWChangeTileImage. You can install only one TileChangeProc. To “turn off” a TileChangeProc that you have already installed, simply call SWSetTileChangeProc(spriteWorldP, NULL).

The tile-change routine you provide should be defined like this:

```
SW_FUNC void MyChangeProc(SpriteWorldPtr spriteWorldP);
```

You would normally use the tileChangeProc to control tiles that change images regularly during the animation, such as a flickering candle or a waterfall. It is up to you to keep your own variables to keep track of when it is time to change each tile’s image, since you will probably not want to change the tile images every single frame. To see a demonstration of a tileChangeProc in action, take a look at the Scrolling Demo. For more information on changing a tile’s image, see SWChangeTileImage.

SWChangeTileImage

This function changes the tile image-to-tileID correspondence, so that a given tileID corresponds to a different tile image.

```
void SWChangeTileImage(SpriteWorldPtr spriteWorldP,                short
tileID,              short newImageID)
```

spriteWorldP	The SpriteWorld using tiling.
tileID	The tile number of the tile to change.
newImageID	The number of the tile image which the tileID will now correspond to.

Description:

There are times when you may wish to change a tile’s image in order to animate the tile, or simply to change it temporarily. For instance, you might have a game with a waterfall where the waterfall tiles are constantly changing to make the water look real, or you might want to change all the tiles of a certain type to a different image once the player performs a certain action, such as pushing a button that triggers all the tiles that were previously blocking the player’s path to change so that the player can move over them.

The best way to describe how to use this function would be to provide an example. Let us say that we have a waterfall made out of tiles, where each tile has 5 frames. For this example, we'll also stipulate that these 5 frames were loaded as tiles 0, 1, 2, 3, and 4. To animate the waterfall, you would install this function as your TileChangeProc (See SWSetTileChangeProc for more info):

```
SW_FUNC void MyTileChangeProc(SpriteWorldPtr spriteWorldP)
{
   short curImage;

      // Get the current image used by tileID 0
   curImage = spriteWorldP->curTileImage[0];

      // Change the image
   if (curImage < 4)
      curImage++;
   else
      curImage = 0;

   SWChangeTileImage(spriteWorldP, 0, curImage);
}
```

First we find out which image the tileID currently corresponds to by looking at the curTileImage array of the SpriteWorld. Initially, this array is set up so each tile corresponds to its own image (curTileImage[0] = 0, curTileImage[1] = 1, etc.), but this can be changed with SWChangeTileImage. For example, calling SWChangeTileImage(spriteWorldP, 0, 3) tells SpriteWorld that tileID 0 now corresponds to the image normally used by tile 3. If you were then to look at the curTileImage array, curTileImage[0] would be equal to 3. Then whenever SpriteWorld encounters tileID 0 in the tileMap, it draws the image for tile 3.

Keep in mind that SWChangeTileImage does not change the actual tileID values in the tileMap, but only changes the tileID to tile image correspondence, so that the tileID refers to a different tile image.

Besides changing the curTileImage array, SWChangeTileImage also quickly scans through all the tiles currently visible on the screen, looking for occurrences of the old tile image (in this case, tileID 0), and changing any occurrences to the new image by redrawing the tile. Keep in mind that this could take a while if a lot of tiles need redrawing (that is, if a lot of waterfall tiles are currently visible on the screen).

Going back to the code example above, you will notice that after we determine the current image used by tileID 0, we then choose a new image, and then call SWChangeTileImage to inform SpriteWorld of our changes. It is important to realize that tileID 0 is the only tile in the tileMap whose image is being changed. Tile IDs 1-4 are never used in the tileMap, since they are simply provided as alternate images for tileID 0.

If you do not want to animate a tile, but simply want to change its image, then you do not need to do so in a TileChangeProc; you can simply call SWChangeTileImage directly whenever necessary.

Δ WARNING Δ

This function changes the current port and does not set it back when done. This could mess up things like GetMouse, which return the mouse coordinates local to the current port. Therefore, use this function with caution if you use it outside the normal animation loop, where it could mess up your current port.

See Also:

SWResetCurrentTileImages

SWResetCurrentTileImages

This function resets the elements in the curTileImage array to their original values, so each tileID corresponds to its original image.

```
void SWResetCurrentTileImages(SpriteWorldPtr spriteWorldP)
```

`spriteWorldP` The SpriteWorld containing the tiles.

Description:

SWResetCurrentTileImages resets the tileID to tile image correspondence, so that each tile ID refers to its original image, thus undoing any the effects of any previous calls to SWChangeTileImage. It is important to note, however, that SWResetCurrentTileImages only changes the values in the curTileImage array; it does not change any of the tiles that are visible on the screen by redrawing them, like SWChangeTileImage does.

For this reason, you would most likely call this function before starting a game and between levels, to make sure that the effects of calls to SWChangeTileImage during the previous level or game do not affect the next level or game.

See Also:

SWChangeTileImage

SWReturnTileUnderPixel

This function returns the tileID under pixelCol and pixelRow

```
short SWReturnTileUnderPixel(SpriteWorldPtr spriteWorldP,          short
tileLayer,                   short pixelCol,                       short
pixelRow);
```

spriteWorldP	The SpriteWorld containing the TileMap.
tileLayer	A value between 0 and 9 specifying the tile layer you wish to read.
pixelCol	The pixel column.
pixelRow	The pixel row.

Description:

This function converts a pixel's coordinates to the TileMap's coordinates and returns the tileID of the tile in that location of the tileLayer you specify. Although you could do this quite easily yourself with the code below, this function makes it just a little easier.

```
short row, col;
row = pixelRow / spriteWorldP->tileHeight;
col = pixelCol / spriteWorldP->tileWidth;
tileID = spriteWorldP->tileLayerArray[tileLayer]->tileMap[row][col];
```

This function also makes sure the pixel is within the bounds of the TileMap. If it is not, it returns -1. Likewise, -1 is returned if no tileMap is installed in the specified tileLayer.

SWCheckSpriteWithTiles

Call this function to determine whether a Sprite overlaps certain tiles.

```
Boolean SWCheckSpriteWithTiles(SpriteWorldPtr spriteWorldP,
                               SpritePtr srcSpriteP,
                               SWTileSearchType searchType,         Rect
*insetRect,                    short startTileLayer,                short
endTileLayer,                  short firstTileID,                   short
lastTileID,                    Boolean fixPosition)
```

spriteWorldP	The SpriteWorld containing the TileMap.
srcSpriteP	The Sprite you want to compare with the tiles.
searchType	A constant value specifying which part of the Sprite to check.
insetRect	An optional Rect specifying inset values for the Sprite's rect.
startTileLayer	The first tile layer to check with the Sprite.
endTileLayer	The last tile layer to check with the Sprite.
firstTileID	The tile marking the beginning of the "set" of Tiles to check.
lastTileID	The tile marking the end of the "set" of Tiles to check.
fixPosition	Indicates whether to move the Sprite off of the tile.

Description:

This function provides an easy way to determine whether your Sprite has come in contact with any tile within a certain set of tiles since it last moved. This function is particularly useful to see if your Sprite has run into a wall just after your MoveProc has moved it. If it has, this function can optionally back up your Sprite so it is up against the wall, but no longer in it. However, this function can be used for other purposes as well. For instance, you may use it to determine whether your Sprite has come in contact with tiles that damage the Sprite, or have some other effect, such as causing the Sprite to slow down.

When calling this function, you must tell it which part of the Sprite you want to check with the tiles. You do this by passing a searchType parameter with one of the following values:

kSWEntireSprite	All tiles in contact with the Sprite will be checked.
kSWTopSide	Only the tiles in contact with the very top line of the Sprite will be checked.
kSWRightSide	Only the tiles in contact with the far right side of the Sprite will be checked.
kSWBottomSide	Only the tiles in contact with the bottom side of the Sprite will be checked.
kSWLeftSide	Only the tiles in contact with the left side of the Sprite will be checked.

When passing a searchType other than kSWEntireSprite, only the tiles in contact with a 1-pixel line on the specified edge of the Sprite will be checked. For instance, if you pass kSWTopSide as your searchType, only the tiles in contact with the very top line of your Sprite (srcSpriteP->destFrameRect.top) will be checked.

This function is very fast even when using a searchType of kSWEntireSprite, since only a small number of tileIDs must be checked in the TileMap. No pixel-precise tests are made to determine if the Sprite's mask collides the Tile's mask.

The insetRect parameter allows you to inset each side of the search rect. Normally, the Sprite's destFrameRect is used as the search rect. (The entire rect if kSWEntireSprite is passed to the function, or the appropriate side of the destFrameRect otherwise.) However, sometimes you will not want to check the entire Sprite with the tiles. One example would be a top-down view of a tank game. When a tank runs into a wall, its turret should be able to move over the wall freely, while the actual body of the tank should not pass through the wall.

This is where the insetRect comes in. You can specify inset values for the top, left, bottom, and right sides of a Sprite, and the search rect will be modified by these values before it is used by this function. The top and left values in the rect you pass will be added to the destFrameRect, and the right and bottom values will be subtracted, so passing an insetRect with a value of 10 for each side would shrink the search rect by 10 pixels on each side. The appropriate part of the resulting rect is then used when checking the tiles. (i.e. the entire rect is used if kSWEntireSprite is passed, or the left side is used if kSWLeftSide is passed, etc.) You can pass NULL as the insetRect parameter if you don't need to modify the search rect before it is used.

The startTileLayer and endTileLayer parameters specify the range of tileLayers that you wish to check the Sprite with. So if you pass 0 as your startTileLayer, and 3 as your endTileLayer, layers 0-3 will be checked with your Sprite.

The fixPosition parameter specifies whether you want this function to "fix" the Sprite's position if a collision was detected and a searchType other than kSWEntireSprite was used. If you pass true and a collision is detected, the Sprite will be moved off the tile. This is useful for keeping your Sprite from moving through walls. For instance, if you pass a searchType of kSWTopSide and a collision is found, the Sprite will be moved down just far enough that a collision with that row of tiles is no longer detected.

When the fixPosition parameter is true, the searchType parameter must match the direction the Sprite is moving. For instance, if you use kSWTopSide as your searchType, the Sprite should be moving up. Otherwise, the Sprite will not be repositioned correctly when its position needs fixing.

This function returns true if a tileID is found that is within the tile set specified by the firstTileID and lastTileID parameters. For instance, if you pass a firstTileID of 5 and a lastTileID of 9, the function will return true if a tileID of 5, 6, 7, 8, or 9 is found. Thus by grouping your tiles into logical sets when loading them, you can easily check to see whether your Sprite has run into any tile of a specific set, such as the set of wall tiles, mud tiles, or spike tiles.

This function is safe to use even when part of the Sprite is outside the bounds of the TileMap. It will only check the part of the Sprite that is within the bounds of the TileMap, or do nothing if the Sprite is entirely outside the TileMap's bounds.

Tip: In addition to using the insetRect parameter to make the search rect smaller, you can also use it to make the search rect bigger by passing negative values. For instance, to determine whether a Sprite in a 2D platform game is standing on a wall, you can pass an insetRect with left, top, and right values of 0, and a bottom value of -1. Since negative values expand the search rect instead of shrinking it, a bottom value of -1 will make the search rect's bottom one pixel lower than it would be otherwise. Then pass kSWBottomSide to the function, and this new bottom will be checked with the tiles. If any match your "wall" tile set, your Sprite is standing on a wall!

Tiling Function Prototypes

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

```
OSErr SWInitTiling(SpriteWorldPtr spriteWorldP,                    short
tileHeight,      short tileWidth,                                  short
maxNumTiles)

void SWExitTiling(SpriteWorldPtr spriteWorldP)
```

```
OSErr SWCreateTileMap(TileMapStructPtr *tileMapStructP,            short
numTileMapRows,  short numTileMapCols)

void SWDisposeTileMap(TileMapStructPtr *tileMapStructP)

void SWLockTileMap(TileMapStructPtr tileMapStructP)

void SWUnlockTileMap(TileMapStructPtr tileMapStructP)

void SWInstallTileMap(SpriteWorldPtr spriteWorldP,
                TileMapStructPtr tileMapStructP,                   short
tileLayer)

OSErr SWLoadTileMap(TileMapStructPtr *tileMapStructP,              short
resourceID)

OSErr SWSaveTileMap(TileMapStructPtr tileMapStructP,               short
resourceID)

OSErr SWResizeTileMap(TileMapStructPtr tileMapStructP,             short
newNumTileMapRows,                                                 short
newNumTileMapCols)

OSErr SWLoadTileFromCicnResource(SpriteWorldPtr spriteWorldP,      short
tileID,          short cicnResID,                                  short
maskType)

OSErr SWLoadTilesFromPictResource(SpriteWorldPtr spriteWorldP,     short
startTileID,     short endTileID,                                  short
pictResID,       short maskResID,                                  short
maskType,        short horizBorderWidth                            short
vertBorderHeight)

void SWDisposeTile(SpriteWorldPtr spriteWorldP,                    short
deadTileID)

OSErr SWCreateExtraBackFrame(SpriteWorldPtr spriteWorldP,          Rect
*frameRect)

void SWDisposeExtraBackFrame(SpriteWorldPtr spriteWorldP)

OSErr SWSetPortToExtraBackFrame(SpriteWorldPtr spriteWorldP)

void SWLockTiles(SpriteWorldPtr spriteWorldP)

void SWUnlockTiles(SpriteWorldPtr spriteWorldP)
```

```
void SWSetTilingOn(SpriteWorldPtr spriteWorldP,                    Boolean
tilingIsOnFlag)

OSErr SWSetTileMaskDrawProc(SpriteWorldPtr spriteWorldP,
                DrawProcPtr drawProc)

OSErr SWSetPartialMaskDrawProc(SpriteWorldPtr spriteWorldP,
                DrawProcPtr drawProc)

void SWSetSpriteLayerUnderTileLayer(SpriteLayerPtr spriteLayerP,   short
tileLayer)

void SWDrawTilesInBackground(SpriteWorldPtr spriteWorldP)

void SWResetTilingCache(SpriteWorldPtr spriteWorldP)

void SWDrawTile(SpriteWorldPtr spriteWorldP,                       short
tileLayer,      short tileRow,                                     short
tileCol,        short tileID)

void SWSetTileChangeProc(SpriteWorldPtr spriteWorldP,
                TileChangeProcPtr tileChangeProc)

void SWChangeTileImage(SpriteWorldPtr spriteWorldP,                short
tileID,         short newImageID)

void SWResetCurrentTileImages(SpriteWorldPtr spriteWorldP)

short SWReturnTileUnderPixel(SpriteWorldPtr spriteWorldP,          short
tileLayer,      short pixelCol,                                    short
pixelRow)
```

Boolean SWCheckSpriteWithTiles(SpriteWorldPtr spriteWorldP, SpritePtr srcSpriteP, SWTileSearchType searchType, Rect *insetRect, short startTileLayer, short endTileLayer,short firstTileID, short lastTileID, Boolean fixPosition);