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

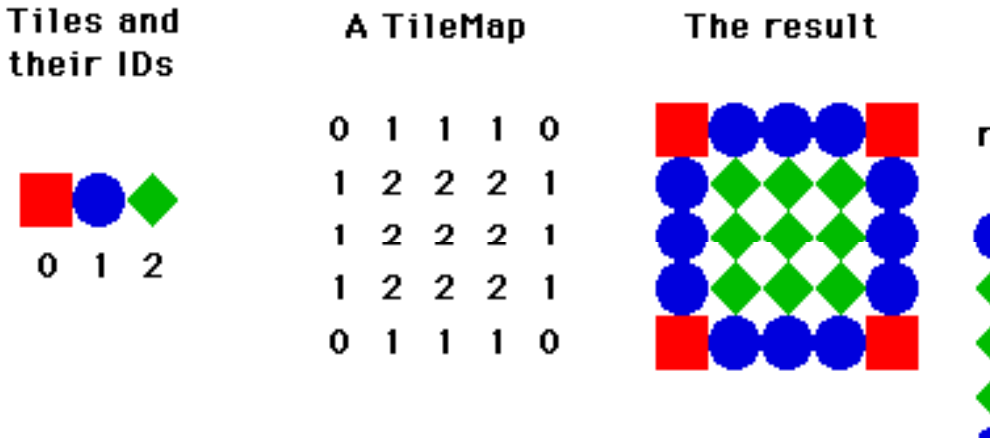
If you are making a scrolling game, then the size of the offscreen area must be evenly divisible by the size of your tiles, so that your tiles fit perfectly in the offscreen area without being clipped. If necessary, you can make the offscreen area slightly larger than what the user sees on the screen 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 allocated by SWCreateTileMap or loaded with SWLoadTileMap.



How to Make Tiles that Appear Above Sprites

SpriteWorld allows you to have tiles that appear in front of any sprites that have been marked to be drawn behind these tiles. That way you could have a Sprite move underneath a bridge, behind a waterfall, under a tree, and so on. Also, you can mark some sprites to appear underneath tiles while other sprites are marked to appear above them. So you could have a man walk behind a tree while some birds fly in front of it. You could also have the same Sprite move behind something, and then later move in front of it.

You define which part of a tile appears in front of the Sprites by making a mask for it. The area of the tile covered by the mask will appear in front of a Sprite that moves across it, while the unmasked portion of the tile will appear behind the Sprite. So if you want the entire tile to appear in front of the Sprite, then mask out the whole tile, and if you want the entire tile to appear behind the Sprite, then use no mask. Only the sprites that are marked to appear under the tiles will be drawn underneath the masked tiles; all other sprites will appear completely above the tiles, regardless of whether the tile has a mask or not. You can mark which sprites should be drawn under the tiles with SWSetSpriteUnderTiles.

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, then 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.

SWInitTiling(spriteWorldP, tileHeight, tileWidth, maxNumTiles)

SWExitTiling(spriteWorldP)

SWCreateTileMap(spriteWorldP, &tileMapP, numTileMapRows, numTileMapCols)

SWResizeTileMap(spriteWorldP, &tileMapP, numRows, numCols)

SWLoadTileMap(spriteWorldP, &tileMapP, resourceID)

SWSaveTileMap(spriteWorldP, resourceID)

SWLockTiles(spriteWorldP)

SWUnlockTiles(spriteWorldP)

SWSetTilingOn(spriteWorldP, tilingIsOnFlag)

SWSetSpriteUnderTiles(srcSpriteP, isUnderFlag)

SWSetTileMaskDrawProc(spriteWorldP, drawProc)

SWLoadTileFromCicnResource(spriteWorldP, tileID, cIconID, maskType)

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

SWDisposeTile(spriteWorldP, deadTileID)

SWDrawTilesInBackground(spriteWorldP)

SWResetTilingCache(spriteWorldP)

SWDrawTile(spriteWorldP, tileRow, tileCol, tileID)

SWSetTileChangeProc(spriteWorldP, MyTileChangeProc)

SWChangeTileImage(spriteWorldP, tileID, newImage)

SWResetCurrentTileImages(spriteWorldP)

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 in 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 can be called.

See Also:

SWLoadTilesFromPictResource
SWLoadTileFromCicnResource

SWExitTiling

This function will dispose of everything created in 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(SpriteWorldPtr spriteWorldP,
                      TileMapPtr *tileMapP,
                      short numTileMapRows,
                      short numTileMapCols)
```

`spriteWorldP`	The SpriteWorld that will be using this TileMap.
`*tileMapP`	The address of a TileMapPtr variable that will be given a pointer to the TileMap that was created.
`numTileMapRows`	The number of rows in the TileMap.
`numTileMapCols`	The number of columns in the TileMap.

Description:

This function will create a TileMap that is numTileMapRows high by numTileMapCols wide. A pointer to the TileMap is saved in the spriteWorldP->tileMap variable. A pointer to the TileMap is also returned to you in the second parameter. You can then use this to access the TileMap as you would access any two-dimensional array; i.e. “tileMapP[row][col] = tileID”. Or, you can use the SpriteWorld’s pointer to the tileMap if you prefer: “spriteWorldP->tileMapP[row][col] = tileID”. You can define a TileMap pointer like so: “TileMapPtr myTileMapP;”. You then pass the address of this pointer to the second parameter of this function, and it will be set to point to the TileMap that was created.

The TileMap 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 initialized with SWCreateTileMap. SWCreateTileMap automatically fills all elements of the TileMap with 0. Take a look at the Scrolling Demo and the Tiling Demo for examples of how to create and set up the TileMap.

An error code will be returned if there was not enough memory to create the TileMap. Note that since this function disposes of any TileMap that previously existed in the SpriteWorld, there will be no TileMap left after this function has been called if it was unable to create a new TileMap (in which case an error code will be returned). For this reason, you should always make sure this function returns noErr (0) before you attempt to use the TileMap it created.

See Also:

SWLoadTileMap

SWResizeTileMap

This function changes the size of an existing TileMap.

```
OSErr SWResizeTileMap(SpriteWorldPtr spriteWorldP,
                      TileMapPtr *tileMapP,
                      short newNumTileMapRows,
                      short newNumTileMapCols)
```

`spriteWorldP`	The SpriteWorld that contains the TileMap.
`*tileMapP`	The address of a TileMapPtr variable that will be given a pointer to the resized TileMap.
`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 the spriteWorldP's 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, and 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 you use the scrolling routines. (See SWSetScrollingWorldMoveBounds for more information). Also check to make sure that the visScrollRect is still within those boundaries.

This function first copies the data from the TileMap into a temporary array, and then disposes the old TileMap. It then creates the new TileMap of the requested size and copies the old TileMap data into it. If it is unable to create the new TileMap because there isn't enough memory, it recreates the old TileMap and copies the original data back into it. Since the original TileMap is disposed and a new one is created, any pointers you have to the old TileMap will become invalid, and should be replaced by the TileMap pointer that is returned by this function. (You can simply pass the address of your TileMap pointer to this function as the second parameter, and it will automatically be updated to point to the resized TileMap).

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 there is no TileMap to change. If an error occurs, this function will leave your TileMap the way it was before this function was called.

SWLoadTileMap

This function will load a tileMap from a resource.

```
OSErr SWLoadTileMap(SpriteWorldPtr spriteWorldP,
                    TileMapPtr *tileMapP,
                    short resourceID)
```

`spriteWorldP`	The SpriteWorld that will be using this TileMap.
`*tileMapP`	The address of a TileMapPtr variable that will be given a pointer to 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. A pointer to the TileMap is placed in spriteWorldP->tileMap, and a pointer to the TileMap is also returned in the second parameter. If there was already a TileMap in the spriteWorldP, it will be disposed and replaced with the new TileMap. Any pointers to the old TileMap will become invalid, and should be replaced with the tileMapP that is returned in the second parameter of this function. For more information about the tileMapP and how to use it to access the TileMap, see the first paragraph in 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. If there was not enough memory to load the TileMap, it is possible that the old TileMap, if there was one, was disposed. If an error occurs, you should check to make sure that the tileMapP variable is not NULL; if it is, then the old TileMap (if there was one) was disposed and can no longer be used.

SWSaveTileMap

This function saves the TileMap currently in the SpriteWorld.

```
OSErr SWSaveTileMap(SpriteWorldPtr spriteWorldP,
                    short resourceID)
```

`spriteWorldP`	The SpriteWorld that will be using this TileMap.
`resourceID`	The resource ID that this TileMap should be saved as.

Description:

This function will save the current 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 there is no TileMap to save, or if there isn't enough memory or disk space to save it. There is some possibility that there might not be enough memory to save the TileMap, since SpriteWorld can not save the TileMap in its normal state, but must copy the two-dimensional array into a one-dimensional array, so it can be saved into a single resource. If there is not enough memory to create the one-dimensional array, then the TileMap can not be saved. However, this is usually not a problem unless you use huge TileMaps; a TileMap that has 100 rows and 100 columns will only take up 20k. (100 x 100 x sizeof(short) = 20,000 bytes, or 20k.)

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 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!

SWSetSpriteUnderTiles

This function marks whether a Sprite should be drawn under the masked portion of the tiles or not.

```
void SWSetSpriteUnderTiles(SpritePtr srcSpriteP,
                           Boolean isUnder)
```

`srcSpriteP`	The Sprite to be changed.
isUnder	A Boolean value indicating whether the Sprite should be drawn below the tiles or not.

Description:

This function allows you to set a Sprite to be drawn "under" the tiles that have been loaded with a mask. This would enable you to add depth to your animations by having some sprites

move behind trees, rocks, etc. while other sprites such as birds fly over the trees and rocks. SWSetSpriteUnderTiles simply sets a variable in the SpriteRec indicating whether the Sprite should be drawn under the tiles or not. Immediately after drawing each Sprite, SpriteWorld checks to see if the Sprite is marked to be drawn under the tiles and if tiling is active (see SWSetTilingOn). If so, it cycles through all the tiles overlapping the Sprite and draws the masked portion of those tiles (if there is a mask for any of the tiles). If there are no tiles with masks that are overlapping the Sprite, then no extra drawing is done.

Naturally it takes SpriteWorld longer to draw sprites when they are under tiles since more drawing must be done. However, you should not notice much of a slowdown unless you have many sprites on the screen at once that are under tiles. So if speed is a concern, you should try to limit the number of sprites you mark to be drawn under tiles, or else make sure that only a small number of those sprites will actually be underneath tiles that have masks at any one time. (You could do this by limiting the number of tiles that have masks, and using these tiles sparingly when you set up your tileMap.)

See Also:

SWSetTileMaskDrawProc

SWSetTileMaskDrawProc

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

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

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

Description:

This function sets the drawProc that will be used when drawing the portion of the tiles that are above sprites. You may use any of the following drawProcs:

SWStdSpriteDrawProc	CopyBits with mask. The default tileMaskDrawProc.
BlitPixie8BitPartialMaskDrawProc	A special BlitPixie drawProc for tiles with 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 tiles in your animation.

See Also:

SWSetSpriteUnderTiles

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
SWSetSpriteUnderTiles
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
SWSetSpriteUnderTiles
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.

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.

SWResetTilingCache

This function resets the SpriteWorld's tiling "cache", which is used to help speed up the tiling.

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

`spriteWorldP` The SpriteWorld using tiling.

Description:

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 tileRow,
                short tileCol,
                short tileID)
```

`spriteWorldP`	The SpriteWorld using tiling.
`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. SWDrawTile will also draw the tile, if its position is currently visible on the screen. 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;

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.

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:

```
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.

See Also:

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 now 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 say 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):

```
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 by SWChangeTileImage. For example, calling SWChangeTileImage(spriteWorldP, 0, 3) tells

SpriteWorld that tileID 0 now corresponds to the image normally used by tileID 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, looking for tileID 0), and changing any occurrences to the new image. 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 you need to.

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:

SWChangeTileImage can be used to change the image used by a particular tileID. SWResetCurrentTileImages can be used to change them all back, so each tile ID refers to its original image.

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