Ritchie's QB64 Sprite Library

Version A0.11 August 2nd, 2011

Thank you for your interest in my sprite library. This document was designed to be used as a tutorial and reference. If you take the time to go through the tutorials you'll find that using it will be much easier to comprehend. The tutorial should only take you an hour or two to go through. All example code in the tutorials has been provided as well, but I do suggest typing in the first example as the tutorial progresses through it. You will learn much more if you do this than simply loading the first example in finished format.

The sprite library is still very much a work in progress, but as you'll see with the Space Invaders and Asteroids games that have been developed with it, it is still very useful in its present form. I encourage all users of the sprite library to show off their work in QB64's forums and highlight any changes, upgrades, modifications and corrections made to the library. If you do make changes to the library please send me a copy of the changes with a brief explanation of what was done. Your changes will be noted in the next release of the library along with credit given to you of course.

You'll find that I've commented every line of the sprite library for easy understanding and reading. Please don't hesitate to dig around in the code and learn from it as well. I'm not the best QB64 programmer by a long shot, but I believe my code will help programmers that are new to QB64.

I would like to take this opportunity to thank Rob (Galleon), the creator of QB64, for his outstanding contribution to the programming world. A special thanks to John Onyon, aka UnseenMachine, for suggesting I look into using sprite sheets and the result of that suggestion bringing this library to life. And a very special thank you to all the forum members of QB64 that are so helpful, especially Clippy for maintaining such a great QB64 documentation Wiki. And finally my wife and kids, for listening to the never ending clickety clack of my keyboard over the past few months.

If you have any questions, comments, suggestions, flames or concerns please don't hesitate to send them my way, either in the QB64 forum or via email at: terry.ritchie@gmail.com

I hope you have as much fun using the library as I did creating it. Have fun and don't forget to share the games you create with the library with the QB64 community. ☺

Sincerely,

Terry Ritchie

Note: This document was written as though the reader has general knowledge and use of QB64, in particular the graphics commands related to QB64. If any of the general QB64 commands are unfamiliar or are proving a task to comprehend, turn to the excellent Wiki provided at the QB64 homepage: **http://qb64.net/wiki/index.php?title=Main_Page**

What is a Sprite?

According to Wikipedia, "A sprite is a two dimensional image or animation that is integrated into a larger scene." Think of a sprite as a small picture, or series of pictures, that can be placed on the computer screen, shown, and then moved to a new location and shown again. This movement, along with the sprite changing through a series of pictures, gives the illusion of animation and motion to a small controllable area. Figure 1 below shows some familiar sprites from the coin operated game Donkey Kong.

Figure 1 – Donkey Kong Sprites

These sprites are arranged in what is commonly known as a "sprite sheet". Each sprite, or character image, on this sprite sheet is 64 pixels wide by 64 pixels high. By placing these sprites at known locations on the screen and timing their appearance just right the illusion of Mario running, jumping, climbing, swinging a hammer and coming to an untimely death can be achieved. Sprites also contain what is known as a transparent layer or color that is ignored by a blitter or sprite engine when placed on the screen. The bright green color seen in figure 1 is the transparent color used for these sprites. When one of the sprites is drawn on the screen the green color is ignored, allowing whatever is behind the sprite to be seen. This allows a sprite to show background scenes and images behind them without destroying the original background information. We'll learn how to grab and manipulate the sprites from a sprite sheet later in this documentation.

Why Use a Sprite Library?

The code required to manipulate sprites can be rather lengthy and tedious to create. Before I wrote the sprite library I would maintain individual images on the hard drive to be loaded and used as sprites. Keeping track of the individual images required lots of code and time to create. Having done this a few times I noticed that there was much similarity in the way the sprites were handled in the various programs I designed. At the urging of a fellow programmer, John Onyon or UnseenMachine as he is known on the QB64 forums, I investigated how sprite sheets could help eliminate many of the graphic files I had to maintain. It was from this investigation that I decided a common method of dealing with sprites would be beneficial as well. By utilizing this sprite library the tedious task of creating the routines to manipulate the sprites have already been written for you. But, for the curious, the sprite library is written in 100% QB64 code and can be viewed and modified at will if desired. In fact I encourage all users of the library to learn from the code as well as modify it to make it better, add new features or improve my code.

The Sprite Library Files.

The sprite library consists of three main files:

- SPRITETOP.BI – a BASIC include file that needs to be located at the very top of your code.
- SPRITE.BI – a BASIC include file that needs to be located at the very bottom of your code.
- SPRITE_NOERROR.BI – a BASIC include file that needs to be located at the very bottom of your code. However, only use this file when you are sure that no errors exist in your code. SPRITE.BI contains error trapping routines that help you to debug your program. SPRITE_NOERROR.BI has had all of the error trapping routines stripped from it, allowing your final compiled project to run faster since these error checks are no longer needed.

We will investigate how to use these files in your code a bit later. For now make sure that the files are in your QB64 folder or a folder of your choosing. A sprite library is useless without first having some sprites to use, so a quick primer on how to create sprites will be covered next.

Creating Sprites.

To use this library your sprites must be arranged on sprite sheets. The library requires that all sprites contained on one sheet be of the same size. As seen in figure 1 the Donkey Kong sprites all have the same dimension of 64 pixels wide by 64 pixels high. If the Donkey Kong game has sprites that exceed this limit there are two options to choose from. The first is to create another sprite sheet containing the larger, same sized sprites. The second would be to “chop” the sprites up and put them back together inside your code. An example of this can be seen in Figure 1. The character Peach, the girl in the dress, is taller than 64 pixels in height. She has three different stances she can pose in while the game is in play. Three of Peach’s sprites are of these three stances and the fourth is her head. In the game’s code you would simply put one of the three Peach stances where it belongs and then put Peach’s head sprite on top.

If the Donkey Kong game has sprites that are smaller, then again there are two options. First, you could center the smaller sprites within the 64x64 pixel area, surrounded by transparent colored pixels. This is a perfectly acceptable solution for static sprites, or sprites that never, or rarely, interact with other sprites. But, because they have an oversized area of nothing around them, collision detection with other sprites is much harder to achieve. The second option, and probably the best overall, is to create another sprite sheet containing the smaller sprites.

How to create a sprite sheet.

At the very least you’ll need to have at your disposal a graphics program that you are comfortable using and it must be able to create .PNG graphic files. In order to use the transparency features of the sprite library it must also be able to support the creation of transparent .PNG files. I use a program called PhotoImpact X3, but any good graphics editor should be up to the task. As you may have already realized the sprite library uses 32bit color .PNG files exclusively.

For those that are artistically challenged (me!) you can still create some pretty impressive sprite sheets. Scattered all over the Internet are sites devoted to collecting sprites and sprite sheets. Rarely, however, are the sprite sheets in a form that you can use right away with the sprite library. You'll still need your trusty graphics program to create sprite sheets that are compatible with the sprite library. One of my favorite places to get sprites is the Sprite Database located at http://sdb.drshnaps.com/index.php. Simply do a Google search for "Sprite Sheet" or "Sprite Graphics" and literally hundreds of resources will be at your fingertips.

The width and height of a sprite sheet needs to be divisible by the size of the sprites it contains. For example, the Donkey Kong sprite sheet contains 64x64 pixel sprites; therefore the sprite sheet must accommodate these perfectly. The size of the sprite sheet in Figure 1 is 512 pixels wide by 192 pixels high, which will accommodate 8 sprites in each row (64 times 8 = 512) and 3 sprites in each column (64 times 3 = 192) for a total of 24 sprites. The sprite sheet could have just as easily been 1536 pixels wide by 64 pixels high creating one long row of 24 sprites (24 times 64 = 1536). The dimension of the sprite sheets you create is completely up to you as long as their width and height is evenly divisible by the size of the sprites it contains. You may have an oddball sized sprite that needs to be on a sheet of its own. This is acceptable as well and the sprite sheet would simply be the same size as the sprite itself. Also, each sprite position on the sprite sheet does not have to contain a sprite. If we only had 23 sprites to use on the Donkey Kong sprite sheet, any one of the sprite positions could have no sprite and this would be perfectly acceptable. In fact, it would probably be a wise idea to include an extra row or two on a sprite sheet for sprites you may wish to add later on.

If you wish to use transparency with your sprites then fill in the remaining pixels with a color that is not used in the actual sprites and instruct your graphics program to use this color as the transparency color. The sprite library will recognize the transparency color and use it when it places sprites on the screen. Sprite sheets must also be saved in 32bit color.

Now that we know how to define and create sprite sheets let's go ahead and load a sprite sheet into memory. But first we'll need to make the basic construct of a program that includes the sprite library files. In the QB64 editor type in the following:

```
'$INCLUDE:'spritetop.bi'



'$INCLUDE:'sprite.bi'
```

Include a few blank lines in between the two statements above. SPRITETOP.BI contains the constant and type declarations needed for the sprite library, and must always be at the top of your program's code. The file SPRITE.BI contains the actual functions and procedures that make up the sprite library's command set and must always be the last line in your program's code. If you're wondering what the extension of .BI stands for it's "BASIC Include file".

If you chose to place the sprite library files in a folder other than QB64 then you'll need to modify the above to lines to accommodate this. For example, I keep library files in a folder called "Libs" inside my QB64 folder. I would need to append this folder name as follows:

```
'$INCLUDE:'Libs\spritetop.bi'
'$INCLUDE:'Libs\sprite.bi'
```

SPRITESHEETLOAD() – defining the sprite sheet and loading it into memory.

The sprite library command SPRITESHEETLOAD is used to load a sprite sheet into memory for later use. SPRITESHEETLOAD has the following syntax:

handle% = SPRITESHEETLOAD(*filename$*, *spritewidth%*, *spriteheight%*, *transparent&*)

Modify the code in your editor to look like this: (there is no need to capitalize the library commands; they will be auto-capitalized by the editor)

```
'$INCLUDE:'spritetop.bi'

DIM dksheet%

dksheet% = SPRITESHEETLOAD("dkong.png", 64, 64, _RGB(0, 255, 0))

'$INCLUDE:'sprite.bi'
```

The Donkey Kong sprite sheet has been included with the sprite library for these examples. The variable dksheet% is a handle, or pointer, that now contains a value that refers to the Donkey Kong sprite sheet. dkong.png is the name of the sprite sheet file contained on the hard drive. 64, 64 informs SPRITESHEETLOAD of the size of each sprite on the sheet and finally _RGB(0, 255, 0) contains the color value of the transparent color found on the sheet, bright green in this case. If your sprite sheet does not contain a transparent color a special constant has been created to let the command know this. Simply substitute the word NOTRANSPARENCY in its place and the command will not use transparency information. SPRITESHEETLOAD will halt program execution and inform you of an error if the filename you specify does not exist.

SPRITENEW() – creating a sprite from the sprite sheet in memory.

The sprite library command SPRITENEW is used to create a sprite from a previously loaded sprite sheet. SPRITENEW has the following syntax:

handle% = SPRITENEW(*sheet%*, *cell%*, *behavior%*)

Modify the code in your editor to look like this: (from now on new lines will be **bold** to show which ones were added)

```
'$INCLUDE:'spritetop.bi'

DIM dksheet%
DIM mario%

dksheet% = SPRITESHEETLOAD("dkong.png", 64, 64, _RGB(0, 255, 0))
mario% = SPRITENEW(dksheet%, 1, SAVE)

'$INCLUDE:'sprite.bi'
```

The SPRITENEW command has been used to create a handle, or pointer, called mario% that points to the first image located on dksheet%. dksheet% is pointing to the sprite sheet we previously loaded informing SPRITENEW of which sprite sheet to use. The number 1 tells SPRITENEW to use the first image in the upper left hand corner of the sprite sheet and SAVE indicates that the sprite should save and restore the contents of the background as it moves around the screen.

SPRITENEW references images on a sprite sheet by a number. The images are numbered starting at 1 in the upper left hand corner and moving right. When the last image is reached the numbering continues in the row below starting at the left hand side again. Figure 2 below shows how our DKONG.PNG sprite sheet is numbered.

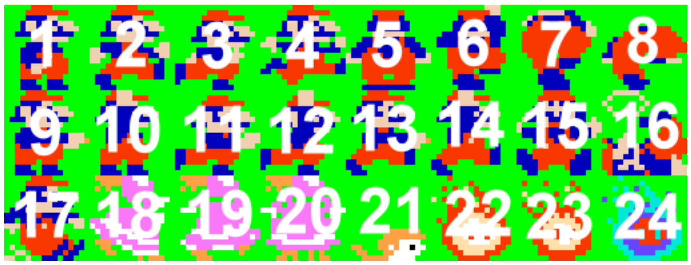

Figure 2 – Sprite Numbering

SPRITEPUT() – placing a sprite on the screen.

Now that we have a sprite sheet in memory to pull sprites from, and a sprite created from the sprite sheet, it's time to place the sprite on the screen. The sprite library command SPRITEPUT is used to place a sprite on the screen and has the following syntax:

SPRITEPUT *x!*, *y!*, *handle%*

Modify the code in your editor to look like this:

```
'$INCLUDE:'spritetop.bi'

DIM dksheet%
DIM mario%
DIM background&

SCREEN _NEWIMAGE(800, 600, 32)
CLS
dksheet% = SPRITESHEETLOAD("dkong.png", 64, 64, _RGB(0, 255, 0))
mario% = SPRITENEW(dksheet%, 1, SAVE)
background& = _LOADIMAGE("backg.png", 32)
_PUTIMAGE (0, 0), background&
SPRITEPUT 399, 299, mario%

'$INCLUDE:'sprite.bi'
```

Mario should now be standing in the center of your screen on a custom background image. Sprites are referenced by their center point in the sprite library. In the code above we created an 800x600 pixel screen with 32bit color. SPRITEPUT was told to put mario% at location 399, 299 on the screen, which placed the center of the mario% sprite at the center of the screen. Of course there is no true center point in a 64x64 pixel sprite when dealing with integer screen values, so the sprite is placed as close to center as possible. If you want sprites that have a true center point, and you need that accuracy, then make your sprites with odd numbered widths and heights.

Remember when we created the mario% sprite we gave it the behavior of SAVE indicating that we want the mario% sprite to save the background? Well, when we used SPRITEPUT to put the mario% sprite on the screen, SPRITEPUT saw this and saved a copy of the background where mario% will reside at. Let's modify the code slightly again to see this in action:

```
'$INCLUDE:'spritetop.bi'

DIM dksheet%
DIM mario%
DIM background&
DIM count%

SCREEN _NEWIMAGE(800, 600, 32)
CLS
dksheet% = SPRITESHEETLOAD("dkong.png", 64, 64, _RGB(0, 255, 0))
mario% = SPRITENEW(dksheet%, 1, SAVE)
background& = _LOADIMAGE("backg.png", 32)
_PUTIMAGE (0, 0), background&
FOR count% = 299 TO 499
    _LIMIT 32
    SPRITEPUT count%, 299, mario%
    _DISPLAY
NEXT count%

'$INCLUDE:'sprite.bi'
```

mario% now moves across the screen at 32 frames per second one pixel at a time without damaging the background! Change the word SAVE in the SPRITENEW command to read DONTSAVE and run the program again. As you can see, this time when mario% moves the background is not taken into consideration and destroyed as mario% moves across the screen. You may be wondering why you would ever want a sprite that does not save the background if this kind of behavior happens. Well, it all boils down to speed. You need to decide which sprites need to save the background and which don't during the design phase of your game. Saving and restoring the background takes a little bit more horsepower and keeping these to a minimum will help to increase the overall speed of your program. Since we're only dealing with one sprite for now, change the SPRITENEW behavior back to SAVE.

Every time the SPRITEPUT command is used on a sprite that has been configured to save the background, SPRITEPUT will restore the background and place the sprite at its new location. When SPRITEPUT is used on a sprite that is not configured to save the background the sprite is simply drawn in the new location. This is why creating sprites that don't save the background yields faster results, but it will be up to you to restore any background data that was lost. This is typically done by clearing the entire screen at the beginning of each frame, redrawing the background information, redrawing the sprites in their new locations and then displaying the results to the user. Depending on the size and number of sprites your game contains this can actually be a faster method than creating only sprites that save their background information. The asteroids game that is included with the sprite library is an example of clearing and redrawing each frame with updated information. The Space Invaders game included with the sprite library uses a combination of both types of sprites.

Animating Sprites

To animate a sprite you simply change its image periodically and the illusion of animation is achieved. There are a number of commands built into the library that can do this allowing for manual changes by the programmer or automatic changes by the library.

SPRITESET() – choosing a different image from the sprite sheet.

SPRITESET allows a sprite's image to be changed by selecting a new image from the sprite sheet that was defined for the sprite when created with the SPRITENEW command. The syntax for the SPRITESET command is as follows:

SPRITESET *handle%*, *cell%*

Modify the code in your text editor again to see how SPRITESET works.

```
'$INCLUDE:'spritetop.bi'

DIM dksheet%
DIM mario%
DIM background&
DIM count%

SCREEN _NEWIMAGE(800, 600, 32)
CLS
dksheet% = SPRITESHEETLOAD("dkong.png", 64, 64, _RGB(0, 255, 0))
mario% = SPRITENEW(dksheet%, 1, SAVE)
background& = _LOADIMAGE("backg.png", 32)
_PUTIMAGE (0, 0), background&
FOR count% = 299 TO 499
    _LIMIT 32
    SPRITEPUT count%, 299, mario%
    _DISPLAY
NEXT count%
SPRITESET mario%, 16
SPRITEPUT count%, 299, mario%
_DISPLAY

'$INCLUDE:'sprite.bi'
```

After marios%'s short journey it seems it was just too much for him and he meets an untimely demise. By using SPRITESET the sprite's image was changed to image number 16 located on the dksheet% sprite sheet. SPRITEPUT was used again to update mario%'s image on screen to reflect the change that SPRITESET made.

SPRITEANIMATESET() – defining a group of images to animate a sprite.

If you examine the Donkey Kong sprite you'll notice that the images have been arranged in such a way that animation cells are next to each other. For instance, images 1, 2 and 3 could rapidly be changed to give Mario the illusion of running. Images 5, 6, 7 and 8 are all of Mario climbing and images 9 through 14 are all different poses Mario can make while swinging a hammer. The grouping of these different animation sequences were placed on the sheet like this on purpose. Using the SPRITEANIMATESET command we can tell a sprite to use these groupings as an animation sequence. The syntax for the SPRITEANIMATESET command is:

SPRITEANIMATESET *handle%*, *startcell%*, *endcell%*

Modify your code once again to look like the code below:

```
'$INCLUDE:'spritetop.bi'

DIM dksheet%
DIM mario%
DIM background&
DIM count%

SCREEN _NEWIMAGE(800, 600, 32)
CLS
dksheet% = SPRITESHEETLOAD("dkong.png", 64, 64, _RGB(0, 255, 0))
mario% = SPRITENEW(dksheet%, 1, SAVE)
SPRITEANIMATESET mario%, 1, 3
background& = _LOADIMAGE("backg.png", 32)
_PUTIMAGE (0, 0), background&
FOR count% = 299 TO 499
    _LIMIT 32
    SPRITENEXT mario%
    SPRITEPUT count%, 299, mario%
    _DISPLAY
NEXT count%
SPRITESET mario%, 16
SPRITEPUT count%, 299, mario%
_DISPLAY

'$INCLUDE:'sprite.bi'
```

No wonder he falls over dead after his short journey! Look at mario% go! The SPRITEANIMATESET command was used to tell the mario% sprite that its current animation sequence starts at image number 1 on the dksheet% sprite sheet and ends at image 3. But there is another command in there we have not talked about yet.

SPRITENEXT() – advancing to the sprite's next animation image.

After you have used the SPRITEANIMATESET command you can manually advance to the next animation cell by issuing a SPRITENEXT command. The syntax for the SPRITENEXT command is:

SPRITENEXT *handle%*

SPRITENEXT will automatically loop around to the first cell in the animation sequence once it has hit the upper limit imposed by SPRITEANIMATESET. There is another command that allows the programmer to manually change animation cells as well.

SPRITEPREVIOUS() – advancing to the sprite's previous animation image.

SPRITEPREVIOUS works exactly like SPRITENEXT except in the opposite direction. SPRITEPREVIOUS will automatically loop around to the last cell in the animation sequence once it has hit the lower limit imposed by SPRITEANIMATESET. The syntax for SPRITEPREVIOUS is:

SPRITEPREVIOUS *handle%*

SPRITEANIMATION() – enable or disable a sprite's auto-animation feature.

There will be times when manually handling the animation of a sprite is needed, but wouldn't it be nice to be able to tell a sprite to animate by itself? This is what SPRITEANIMATION can do for sprites. The syntax for SPRITEANIMATION is as follows:

SPRITEANIMATION *handle%*, *onoff%*, *behavior%*

Make the following modifications to your code. Note that the SPRITENEXT command was removed.

```
'$INCLUDE:'spritetop.bi'

DIM dksheet%
DIM mario%
DIM background&
DIM count%

SCREEN _NEWIMAGE(800, 600, 32)
CLS
dksheet% = SPRITESHEETLOAD("dkong.png", 64, 64, _RGB(0, 255, 0))
mario% = SPRITENEW(dksheet%, 1, SAVE)
SPRITEANIMATESET mario%, 1, 3
SPRITEANIMATION mario%, ANIMATE, FORWARDLOOP
background& = _LOADIMAGE("backg.png", 32)
_PUTIMAGE (0, 0), background&
FOR count% = 299 TO 499
    _LIMIT 32
    SPRITEPUT count%, 299, mario%
    _DISPLAY
NEXT count%
SPRITEANIMATION mario%, NOANIMATE, FORWARDLOOP
SPRITESET mario%, 16
SPRITEPUT count%, 299, mario%
_DISPLAY

'$INCLUDE:'sprite.bi'
```

We get the same results! mario% is still running to his death, but this time a manual update of the sprite's animation cell was not needed. When automatic animation is enabled for a sprite the SPRITEPUT command sees this and advances to the next animation cell automatically. While in automatic animation mode SPRITESET becomes useless and SPRITENEXT and SPRITEPREVIOUS might yield unexpected results.

The reason SPRITESET no longer works is because SPRITEPUT will always go to the next (or previous, we'll talk about that in a minute) animation cell. In our code toward the bottom the automatic animation was turned off to allow SPRITESET to work. If automatic animation had not been turned off, SPRITESET would have set mario% to image number 16, but SPRITEPUT would have just changed it back to the next animation cell. REM out the second SPRITEANIMATION command to see the results for yourself.

If you use SPRITENEXT or SPRITEPREVIOUS while automatic animation is turned on, you'll actually skip two cells. This is because SPRITENEXT and SPRITEPREVIOUS will move to the next cell as intended, but SPRITEPUT will also advance a cell sensing that automatic animation is turned on for the sprite. It's best not to use manual animation commands while automatic animation is turned on for a sprite, but with proper coding you may find this behavior desirable, such as making an animation to appear to run twice as fast.

As you've seen in the example code there are two constants you can use to turn automatic animation on, ANIMATE, and off, NOANIMATE. There are also three behaviors to choose from; FORWARDLOOP which cycles through the cells from lowest value to highest; BACKWARDLOOP which cycles through the cells from highest to lowest and BACKFORTHLOOP which "ping-pongs" between cells from lowest to highest, then highest to lowest, and so on. When you use SPRITEANIMATION to turn off automatic animation for a sprite using NOANIMATE, the behavior value is ignored.

Setting Sprites in Motion

You've already see one way to move a spite around the screen using SPRITEPUT with x and y values that you supply, but there are a few other ways to get sprites moving around.

SPRITETRAVEL() – moving a sprite in a given direction angle and speed.

SPRITETRAVEL can be used to move a sprite in a specific direction, or angle, ranging from 0 to 360 degrees and at a given speed which is really a distance vector from the current location. SPRITETRAVEL has the following syntax attached to it:

SPRITETRAVEL *handle%*, *direction!*, *speed!*

Let's modify the code again to see SPRITETRAVEL in action.

```
'$INCLUDE:'spritetop.bi'

DIM dksheet%
DIM mario%
DIM background&
DIM count%

SCREEN _NEWIMAGE(800, 600, 32)
CLS
dksheet% = SPRITESHEETLOAD("dkong.png", 64, 64, _RGB(0, 255, 0))
mario% = SPRITENEW(dksheet%, 1, SAVE)
SPRITEANIMATESET mario%, 1, 3
SPRITEANIMATION mario%, ANIMATE, FORWARDLOOP
background& = _LOADIMAGE("backg.png", 32)
_PUTIMAGE (0, 0), background&
```

```
SPRITEPUT 399, 299, mario%
FOR count% = 0 TO 359
    _LIMIT 32
    SPRITETRAVEL mario%, count%, 2
    SPRITEPUT SPRITEAX(mario%), SPRITEAY(mario%), mario%
    _DISPLAY
NEXT count%
SPRITEANIMATION mario%, NOANIMATE, FORWARDLOOP
SPRITESET mario%, 16
SPRITEPUT SPRITEAX(mario%), SPRITEAY(mario%), mario%
_DISPLAY

'$INCLUDE:'sprite.bi'
```

It appears that even running in circles takes the life out of mario%! The sprite library keeps track of the x and y positions of all sprites that have been created. The x and y positions that it keeps track of are SINGLE values. You may have noticed that SPRITEPUT's x and y syntax values were followed by exclamation points (!) indicating that they are SINGLE precision. So far, we have been passing INTEGERs (%) to SPRITEPUT, but SPRITEPUT will allow for even finer precision than INTEGERs allow.

SPRITETRAVEL uses these stored SINGLE x and y values to calculate where the next x and y values should be. After the calculations are finished SPRITETRAVEL saves the new values to the sprite's x! and y! values that the library is keeping track of. We added a SPRITEPUT command before the FOR...NEXT loop to seed the x! and y! values giving SPRITETRAVEL initial values to work with. We then changed the count values in the FOR...NEXT loop to count from 0 to 359, a full circle. The SPRITETRAVEL command is then told to move the mario% sprite in the direction that count% currently contains at a distance, or speed, of 2. SPRITETRAVEL calculates the new x! and y! values and then stores them back into mario%.

Now, there needs to be a way to get the actual x! and y! values that were computed by SPRITETRAVEL so we can tell SPRITEPUT where to place mario% on the screen. We can do this by using two new commands called SPRITEAX() and SPRITEAY(). SPRITEAX returns the actual x! location of a sprite and SPRITEAY returns the actual y! location of a sprite. By giving these values to SPRITEPUT mario% is updated to the new calculated locations.

SPRITEAX() – get a sprite's actual current x value stored in the library.

The syntax for the SPRITEAX command is:

actualx! = SPRITEAX(*handle%*)

SPRITEAY() – get a sprite's actual current y value stored in the library.

The syntax for the SPRITEAY command is:

actualy! = SPRITEAY(*handle%*)

If you need the x and y location of a sprite on the screen, there is no need to convert the actual x! and y! values to integers. The sprite library is already doing this for you and saving the results. You can use two very similar commands to get the integer x and y values of the sprite's screen location.

SPRITEX() – get a sprite's current x location on the screen.

The syntax for the SPRITEX command is:

screenx% = SPRITEX(*handle%*)

SPRITEY() – get a sprite's current y location on the screen.

The syntax for the SPRITEY command is:

screeny% = SPRITEY(*handle%*)

There are going to be times when you would like to set a sprite in motion on its own. The sprite library has commands to do this as well. The library command SPRITEMOTION, along with some supporting commands, will allow you to set a sprite in automatic motion.

SPRITEMOTION() – enable or disable a sprite's auto-motion feature.

The syntax for SPRITEMOTION is very basic as it relies on a few other commands to set up motion parameters.

SPRITEMOTION *handle%*, *behavior%*

First, let's make a few modifications to our code and then cover what these new commands are doing.

```
'$INCLUDE:'spritetop.bi'

DIM dksheet%
DIM mario%
DIM background&
DIM count%

SCREEN _NEWIMAGE(800, 600, 32)
CLS
dksheet% = SPRITESHEETLOAD("dkong.png", 64, 64, _RGB(0, 255, 0))
mario% = SPRITENEW(dksheet%, 1, SAVE)
SPRITEANIMATESET mario%, 1, 3
SPRITEANIMATION mario%, ANIMATE, FORWARDLOOP
SPRITESPEEDSET mario%, 1
SPRITEDIRECTIONSET mario%, 45
background& = _LOADIMAGE("backg.png", 32)
_PUTIMAGE (0, 0), background&
SPRITEPUT 299, 299, mario%
SPRITEMOTION mario%, MOVE
FOR count% = 1 TO 200
    _LIMIT 32
    SPRITEPUT MOVE, MOVE, mario%
    _DISPLAY
NEXT count%
SPRITEMOTION mario%, DONTMOVE
SPRITEANIMATION mario%, NOANIMATE, FORWARDLOOP
SPRITESET mario%, 16
SPRITEPUT SPRITEX(mario%), SPRITEY(mario%), mario%
```

```
_DISPLAY

'$INCLUDE:'sprite.bi'
```

Here we have set mario% on a path at a 45 degree angle and told the sprite to move automatically. SPRITESPEEDSET has been used to tell the mario% sprite to move at a speed of 1. SPRITEDIRECTIONSET is then used to tell the mario% sprite to move at a 45 degree angle using the speed provided by SPRITESPEEDSET. It's important to note that SPRITESPEEDSET must be set before SPRITEDIRECTIONSET when being used for the very first time, otherwise SPRITEDIRECTIONSET will not have a seeded speed value to use and will use zero instead, effectively calculating no forward motion. The mario% sprite is then placed on the screen at 299, 299 to seed the x and y values of the sprite. The SPRITEMOTION command turns on the auto-motion feature of a sprite. Here we have told the mario% sprite to MOVE. The FOR...NEXT loop count values have been changed to count 200 times. Inside the FOR...NEXT loop we use SPRITEPUT once again to move mario% across the screen, but the x and y coordinates are replaced with the constant MOVE. When SPRITEMOTION is used to turn on auto-motion for mario% SPRITEPUT is alerted to this and effectively ignores the x and y values passed in. In reality you could have just used the coordinates 0, 0 if you like, but, by placing the constants MOVE, MOVE in their place the programmer is reminded that the sprite's auto-motion has been turned on. Every time SPRITEPUT is used from now on with mario% the sprite will move according to the values set in SPRITESPEEDSET and SPRITEDIRECTIONSET. Finally, we turn off mario%'s auto-motion by again using SPRITEMOTION but this time using the constant DONTMOVE.

The speed and direction of an auto-moving sprite can be changed at any time by simply changing the values for SPRITESPEEDSET and SPRITEDIRECTIONSET. To see this in action, add the following line after the _LIMIT 32 command line located inside the FOR...NEXT loop:

```
IF count% = 100 THEN SPRITEDIRECTIONSET mario%, 135
```

When the FOR...NEXT loop reaches 100 the mario% sprite will change direction to a 135 degree heading.

These auto-motion commands were instrumental in the creation of the Asteroids game. The asteroids were given a random speed and direction and sent along their merry way; only checking on them occasionally to see if they crossed the screen or were hit by another object. If they crossed the screen, auto-motion was turned off, the asteroids moved to their new location, and then auto-motion turned back on. If they collided with another object, auto-motion was turned off and the sprite moved off-screen for later recycling. This made a very quick and easy way to handle many objects moving in all different directions at the same time.

SPRITESPEEDSET() – setting a sprite's auto-motion speed.

As seen in the previous example code, SPRITESPEEDSET sets up SPRITEMOTION's speed, or distance vector, value to be used in auto-motion. The syntax for SPRITESPEEDSET is:

SPRITESPEEDSET *handle%, value!*

Notice that *value!* is SINGLE in scope, allowing for very precise speed, or distance, movements if desired. SPRITEMOTION also relies on the command SPRITEDIRECTIONSET to set up a direction vector value for a sprite.

SPRITEDIRECTIONSET() – setting a sprite's auto-motion direction angle.

The syntax for SPRITEDIRECTIONSET:

SPRITEDIRECTIONSET *handle%*, *value!*

Again, take note that the *value!* passed to SPRITEDIRECTIONSET is SINGLE in scope, allowing for partial degree changes for very fine directional control if desired. While a sprite is in auto-motion mode it may be required that you need to get the current direction value of the sprite. The command to retrieve this information is SPRITEDIRECTION.

SPRITEDIRECTION() – get a sprite's current auto-motion direction angle.

To retrieve a sprite's current auto-motion direction angle use the following syntax:

direction! = SPRITEDIRECTION(*handle%*)

In an upcoming modification to our code you'll see an example of this command in use.

Now that we have our sprite in motion, with a desired direction and speed, let's through a little spin on it shall we? One last command that can be used to set a parameter that SPRITEMOTION can use is SPRITESPINSET, which allows for the automatic rotation of a sprite.

SPRITESPINSET() – setting a sprite's auto-motion spin direction and speed.

To put a little spin on your sprite use the following syntax:

SPRITESPINSET *handle%*, *value!*

The *value!* passed in can be positive or negative in degrees. To get a clockwise spin pass in a positive value and for a counter-clockwise spin pass in a negative value. Also, just as SPRITEDIRECTIONSET has a counterpart command for retrieving sprite direction values, so too does SPRITESPINSET called SPRITEROTATION which we will discuss in detail in a little bit.

It's time to modify the code again to add a little spin to mario%'s adventures.

```
'$INCLUDE:'spritetop.bi'

DIM dksheet%
DIM mario%
DIM background&
DIM count%

SCREEN _NEWIMAGE(800, 600, 32)
CLS
dksheet% = SPRITESHEETLOAD("dkong.png", 64, 64, _RGB(0, 255, 0))
mario% = SPRITENEW(dksheet%, 1, SAVE)
SPRITEANIMATESET mario%, 1, 3
```

```
SPRITEANIMATION mario%, ANIMATE, FORWARDLOOP
SPRITESPEEDSET mario%, 1
SPRITEDIRECTIONSET mario%, 45
SPRITESPINSET mario%, 2
background& = _LOADIMAGE("backg.png", 32)
_PUTIMAGE (0, 0), background&
SPRITEPUT 299, 299, mario%
SPRITEMOTION mario%, MOVE
FOR count% = 1 TO 200
    _LIMIT 32
    IF count% = 100 THEN SPRITEDIRECTIONSET mario%, 135
    SPRITEPUT MOVE, MOVE, mario%
    LOCATE 5, 1
    PRINT "Mario current spin =";SPRITEROTATION(mario%)
    _DISPLAY
NEXT count%
SPRITEMOTION mario%, DONTMOVE
SPRITEANIMATION mario%, NOANIMATE, FORWARDLOOP
SPRITEROTATE mario%, 0
SPRITESET mario%, 16
SPRITEPUT SPRITEX(mario%), SPRITEY(mario%), mario%
_DISPLAY

'$INCLUDE:'sprite.bi'
```

Here we have set `mario%` to spin clockwise 2 degrees each time it is drawn using `SPRITEPUT`. A new command called `SPRITEROTATE` is used to set `mario%` back to a rotational spin of zero degrees after his journey is complete. Inside of the `FOR...NEXT` loop a `PRINT` statement is used to show `mario%`'s current spin angle using `SPRITEROTATION`.

SPRITEROTATE() – rotate a sprite at a given angle.

`SPRITEROTATE` can be used to manually rotate a sprite from `0` to `360` degrees. `SPRITEROTATE` will also alter the value that a sprite's auto-motion value contains, allowing you to make manual changes to a sprite currently using `SPRITESPINSET`. The syntax for `SPRITEROTATE` is:

`SPRITEROTATE` *`handle%`*, *`degrees!`*

The value for *`degrees!`* must be a value between `0` and `359.99`...

If you need to retrieve the current rotational value of a sprite the `SPRITEROTATION` command can be used to achieve this.

SPRITEROTATION() – get the current rotation angle of a sprite.

`SPRITEROTATION` can be used to retrieve the current rotation angle of a sprite in degrees. This command is especially useful when you need to know the current rotational angle of a sprite that is under auto-motion control or when the angle has been set by an upcoming command called `SPRITEANGLE`, that sets a sprite's angle based on the direction of other sprites. The syntax for `SPRITEROTATION` is:

`angle! = SPRITEROTATION(`*`handle%`*`)`

The angle! that is returned will be a SINGLE value ranging from 0 to 359.99...

As mentioned earlier the command that can be used to retrieve the angle between two sprites in SPRITEANGLE. SPRITEANGLE can also be used to set a sprite's angle based on another sprite.

SPRITEANGLE() – get the angle in degrees between two sprites.

The syntax for the SPRITEANGLE command is:

angle! = SPRITEANGLE(*handle1%*, *handle2%*)

The angle! that is returned will be a SINGLE value ranging from 0 to 359.99...

Let's update our code again to include an example of SPRITEANGLE at work. Another sprite will be created that will always point toward mario% while he moves along his path.

```
'$INCLUDE:'spritetop.bi'

DIM dksheet%
DIM mario%
DIM barrel%
DIM background&
DIM count%

SCREEN _NEWIMAGE(800, 600, 32)
CLS
dksheet% = SPRITESHEETLOAD("dkong.png", 64, 64, _RGB(0, 255, 0))
mario% = SPRITENEW(dksheet%, 1, SAVE)
barrel% = SPRITENEW(dksheet%, 67, SAVE)
SPRITEANIMATESET mario%, 1, 3
SPRITEANIMATION mario%, ANIMATE, FORWARDLOOP
SPRITESPEEDSET mario%, 1
SPRITEDIRECTIONSET mario%, 45
SPRITESPINSET mario%, 2
background& = _LOADIMAGE("backg.png", 32)
_PUTIMAGE (0, 0), background&
SPRITEPUT 199, 299, mario%
SPRITEMOTION mario%, MOVE
FOR count% = 1 TO 500
    _LIMIT 32
    IF count% = 250 THEN SPRITEDIRECTIONSET mario%, 135
    SPRITEPUT MOVE, MOVE, mario%
    SPRITEROTATE barrel%, SPRITEANGLE(barrel%, mario%)
    SPRITEPUT 399, 299, barrel%
    LOCATE 5, 1
    PRINT "Mario current spin  =";SPRITEROTATION(mario%)
    PRINT "Barrel current spin =";SPRITEROTATION(barrel%)
    _DISPLAY
NEXT count%
SPRITEMOTION mario%, DONTMOVE
SPRITEANIMATION mario%, NOANIMATE, FORWARDLOOP
SPRITEROTATE mario%, 0
SPRITESET mario%, 16
SPRITEPUT SPRITEX(mario%), SPRITEY(mario%), mario%
_DISPLAY
```

```
'$INCLUDE:'sprite.bi'
```

Another sprite, number 67 on dksheet%, the barrel, has been added to the program with the addition of another SPRITENEW command creating barrel%. Next we move mario% back a little bit and give him some more distance to travel by increasing the values in the FOR...NEXT loop. Inside the loop the SPRITEROTATE command is used to rotate the barrel at an angle that points toward mario% using SPRITEANGLE as the angle to set. The first sprite specified in SPRITEANGLE can be thought of as the "from" sprite, and the second sprite the "to" sprite. With SPRITEANGLE we are retrieving the angle from barrel% to mario%. Had we reversed these sprites then the barrel%'s angle would be 180 degrees off, because we would have gotten the angle from mario% to barrel%.

SPRITEANGLE is what makes it possible for the small saucer in the Asteroids game to be so deadly because it can be used to calculate a dead-on angle between two sprites. This will be a handy command if you are creating games with gun turrets or enemy opponents that need to have accuracy shooting at either the player or other sprites.

There are two other commands related to sprite motion, SPRITEREVERSEX and SPRITEREVERSEY. They simply change the x and y vector direction of a sprite under auto-motion control. Say for instance a sprite hits the boundary of the screen on either the right or left side. Using the command SPRITEREVERSEX will make the sprite appear to bounce off the edge of the screen.

SPRITEREVERSEX() – reversing a sprite's auto-motion x vector direction.

The syntax for SPRITEREVERSEX is:

SPRITEREVERSEX *handle%*

SPRITEREVERSEY() – reversing a sprite's auto-motion y vector direction.

The syntax for SPRITEREVERSEY is:

SPRITEREVERSEY *handle%*

Many Sprites from One

In most games there are multiple identical sprites used all over. For instance, in Donkey Kong there are many rolling barrels coming down the platforms at any given time. Instead of creating a new variable for each one of these barrels, a sprite array can be created using the SPRITECOPY command.

SPRITECOPY() – make an identical copy of a sprite.

The SPRITECOPY command will use an existing sprite as a master to produce duplicates from. The syntax of the SPRITECOPY command is:

newhandle% = SPRITECOPY(*handle%*)

Let's go ahead and start a new programming project to show how to create a sprite array using SPRITECOPY. You can go ahead and simply disregard the last project as a copy of it was included with this library called “mario.bas”. This next project is also included called “barrels.bas” if you do not wish to type the next few lines in. Also, the font size in the code below was reduced to keep from word-wrapping the lines.

```
'$INCLUDE:'spritetop.bi'

CONST NUMBARRELS = 100

DIM barrels%(NUMBARRELS)
DIM barrel%
DIM dksheet%
DIM count%
DIM background&

SCREEN _NEWIMAGE(800, 600, 32)
CLS
RANDOMIZE TIMER
background& = _LOADIMAGE("backg.png", 32)
dksheet% = SPRITESHEETLOAD("dkong.png", 64, 64, _RGB(0, 255, 0))
barrel% = SPRITENEW(dksheet%, 38, DONTSAVE)
SPRITEANIMATESET barrel%, 38, 41
SPRITEANIMATION barrel%, ANIMATE, FORWARDLOOP
FOR count% = 1 TO NUMBARRELS
    barrels%(count%) = SPRITECOPY(barrel%)
    SPRITEPUT INT(RND(1) * 600) + 100, INT(RND(1) * 400) + 100, barrels%(count%)
    SPRITESPEEDSET barrels%(count%), RND(3) - RND(3) + 1
    SPRITEDIRECTIONSET barrels%(count%), INT(RND(1) * 360)
    SPRITEMOTION barrels%(count%), MOVE
NEXT count%
SPRITEFREE barrel%
DO
    _LIMIT 32
    _PUTIMAGE (0, 0), background&
    FOR count% = 1 TO NUMBARRELS
        SPRITEPUT MOVE, MOVE, barrels%(count%)
        IF SPRITEX(barrels%(count%)) <= 0 THEN SPRITEREVERSEX barrels%(count%)
        IF SPRITEX(barrels%(count%)) >= _WIDTH(_DEST) THEN SPRITEREVERSEX barrels%(count%)
        IF SPRITEY(barrels%(count%)) <= 0 THEN SPRITEREVERSEY barrels%(count%)
        IF SPRITEY(barrels%(count%)) >= _HEIGHT(_DEST) THEN SPRITEREVERSEY barrels%(count%)
    NEXT count%
    _DISPLAY
LOOP UNTIL INKEY$ <> ""

'$INCLUDE:'sprite.bi'
```

Now that’s what I call a barrel of fun! In this code example we have created a `100` barrel sprite array from a single sprite, `barrel%`, using the `SPRITECOPY` command. After we define the sprite sheet we define a sprite called `barrel%`. Some basic animation is then set up for the sprite. It’s important to note that `SPRITECOPY` will copy every aspect of the sprite being copied. So, the animation that we set up for the `barrel%` will also be copied to the array of sprites with `SPRITECCOPY`. We would not want to set up any motion settings yet as that would yield `100` barrels all moving in the same direction.

The sprite array is created in the `FOR...NEXT` loop that follows. Each barrel is placed on the screen to seed the x and y position values of each barrel, a random speed between 1 and 4 is set for each barrel and a random direction between 0 and 359 is given to each barrel. Finally, each sprite is told to `MOVE` with the `SPRITEMOTION` command. When we are finished setting up the sprite array a new command called `SPRITEFREE` is used to free the original `barrel%` sprite from memory since it is no longer needed. It’s good programming practice with both this library and QB64’s graphics commands to free any resources that are not needed any longer. This keeps from creating memory overflows and leaks that may cause your programs to crash.

Next, we set up a loop that continues until a key is pressed. Inside the loop we set up a frames per second limit of 32, draw the background (essentially clearing the screen), and then set up a FOR...NEXT loop to cycle through all the barrels in the sprite array. Each barrel is moved automatically by SPRITEPUT and then checked for a collision with one of the four screen borders. If a screen border is reached, SPRITEREVERSEX and SPRITEREVERSEY are used to reverse the direction vector of the sprite to make it appear as though it bounced off the edge. Finally, after all this has been done the results are displayed on the screen and the process starts over.

For fun, try changing the value of the constant NUMBARRELS to 1000 and see what happens. The sprite library is fairly quick and should handle 1000 barrels just fine. Now, change the value to 10 as this next section will require less of a cluttered screen.

SPRITEFREE – removes a sprite from memory and frees its resources.

SPRITEFREE allows you to free up resources that are not in use any longer. The syntax for the SPRITEFREE command is:

SPRITEFREE *handle%*

This is an especially important command to use on sprites that were declared to SAVE the background image. The background images can potentially use lots of memory causing you to very quickly run out of RAM and crashing your program. In the Asteroids game every time a new level is achieved the old asteroid sprites are cleared from memory and recreated instead of simply creating more on top of the old ones. This negates the chance of an expert player reaching level 100 from running out of RAM and terrorizing the family in anger over the issue. You have to think of your players! ☺

You can also use sprites to build your background scenes with SPRITESTAMP. SPRITESTAMP lets you use a sprite as a stamp pad to place images of sprites anywhere on the screen you like.

SPRITESTAMP() – stamp a sprite image onto the background.

SPRITESTAMP acts much the same way SPRITEPUT does, in the fact that if a sprite is rotated, flipped or zoomed (flipping and zooming detailed in a bit) SPRITESTAMP will see this and use the modified sprite. The main difference between SPRITESTAMP and SPRITEPUT is that SPRITESTAMP places the image on the background with no regard for saving it. SPRITESTAMP does however support sprites with transparency so the background will still be preserved for transparent pixels. The syntax for SPRITESTAMP is:

SPRITESTAMP *x%*, *y%*, *handle%*

Modify your code to look like the following or simply load “barrels2.bas” to see how the girders were used from the Donkey Kong sprite sheet to create platforms for the barrels to roll on.

```
'$INCLUDE:'spritetop.bi'

CONST NUMBARRELS = 10
DIM barrels%(NUMBARRELS)
DIM barrel%
```

```
DIM dksheet%
DIM count%
DIM girder%
DIM background&

SCREEN _NEWIMAGE(800, 600, 32)
CLS
RANDOMIZE TIMER
background& = _LOADIMAGE("backg.png", 32)
dksheet% = SPRITESHEETLOAD("dkong.png", 64, 64, _RGB(0, 255, 0))
barrel% = SPRITENEW(dksheet%, 38, DONTSAVE)
girder% = SPRITENEW(dksheet%, 52, DONTSAVE)
_PUTIMAGE (0, 0), background&
FOR count% = 0 TO 12
    SPRITESTAMP count% * 64, 100, girder%
    SPRITESTAMP count% * 64, 200, girder%
    SPRITESTAMP count% * 64, 300, girder%
NEXT count%
_FREEIMAGE background&
background& = _COPYIMAGE(_DEST)
SPRITEANIMATESET barrel%, 38, 41
SPRITEANIMATION barrel%, ANIMATE, FORWARDLOOP
FOR count% = 1 TO NUMBARRELS
    barrels%(count%) = SPRITECOPY(barrel%)
    SPRITEPUT INT(RND(1) * 600) + 100, (INT(RND(1) * 3) + 1) * 100 - 32, barrels%(count%)
    SPRITESPEEDSET barrels%(count%), RND(3) - RND(3) + 3
    SPRITEDIRECTIONSET barrels%(count%), 90
    SPRITEMOTION barrels%(count%), MOVE
NEXT count%
SPRITEFREE barrel%
DO
    _LIMIT 32
    _PUTIMAGE (0, 0), background&
    FOR count% = 1 TO NUMBARRELS
        SPRITEPUT MOVE, MOVE, barrels%(count%)
        IF SPRITEX(barrels%(count%)) <= 0 THEN SPRITEREVERSEX barrels%(count%)
        IF SPRITEX(barrels%(count%)) >= _WIDTH(_DEST) THEN SPRITEREVERSEX barrels%(count%)
    NEXT count%
    _DISPLAY
LOOP UNTIL INKEY$ <> ""

'$INCLUDE:'sprite.bi'
```

In the code above we created a new sprite called `girder%` and used a `FOR...NEXT` loop to place 3 rows of `girder%`s on the background. This new background image was saved, a little changing to `SPRITDIRECTIONSET` for each barrel, and the y vector checks removed since the barrels are only moving in the x vector and we get barrels rolling across platforms.

You may have noticed that the Donkey Kong sprite sheet seems incomplete. There is an image of Mario standing to the right, but not to the left, and it seems that Mario is only able to run to the right. Also, if Mario climbs it appears that only his right foot will ever raise, what about his left foot? `SPRITEFLIP` can appear to make more sprites from the ones you have by flipping a sprite either horizontally, vertically or both at the same time.

SPRITEFLIP() – flip a sprite horizontally, vertically, or both.

The syntax for `SPRITEFLIP` is:

`SPRITEFLIP` *`handle%`*, *`behavior%`*

The flipping *behavior%* can be described using one of four constant values: NONE to reset the sprite back to its default view, HORIZONTAL to flip the sprite horizontally, VERTICAL to flip the sprite vertically and BOTH to flip the sprite both horizontally and vertically.

Let's examine a piece of code that uses SPRITEFLIP to make Mario run in the correct direction. You can either type the code below in by hand or load the included file called "mario2.bas".

```
'$INCLUDE:'spritetop.bi'

DIM dksheet%
DIM mario%
DIM background&
DIM count%
DIM girder%

SCREEN _NEWIMAGE(800, 600, 32)
CLS
dksheet% = SPRITESHEETLOAD("dkong.png", 64, 64, _RGB(0, 255, 0))
mario% = SPRITENEW(dksheet%, 1, SAVE)
girder% = SPRITENEW(dksheet%, 52, DONTSAVE)
SPRITEANIMATESET mario%, 1, 3
SPRITEANIMATION mario%, ANIMATE, FORWARDLOOP
SPRITESPEEDSET mario%, 3
SPRITEDIRECTIONSET mario%, 90
background& = _LOADIMAGE("backg.png", 32)
_PUTIMAGE (0, 0), background&
FOR count% = 0 TO 12
    SPRITESTAMP count% * 64, 300, girder%
NEXT count%
SPRITEPUT 199, 268, mario%
SPRITEMOTION mario%, MOVE
DO
    _LIMIT 16
    IF SPRITEX(mario%) <= 32 THEN
        SPRITEREVERSEX mario%
        SPRITEFLIP mario%, NONE
    END IF
    IF SPRITEX(mario%) >= _WIDTH(_DEST) - 32 THEN
        SPRITEREVERSEX mario%
        SPRITEFLIP mario%, HORIZONTAL
    END IF
    SPRITEPUT MOVE, MOVE, mario%
    _DISPLAY
LOOP UNTIL INKEY$ <> ""
SPRITEMOTION mario%, DONTMOVE
SPRITEANIMATION mario%, NOANIMATE, FORWARDLOOP
SPRITESET mario%, 16
SPRITEPUT SPRITEX(mario%), SPRITEY(mario%), mario%
_DISPLAY

'$INCLUDE:'sprite.bi'
```

There is nothing new going on before the DO...LOOP, just as before we are assigning a sprite sheet, sprites based off the sprite sheet and setting up the background with a row of girder%s going across. The real magic is happening inside the IF...END IF statements that are contained within the

DO...LOOP. When mario% reaches the right edge of the screen SPRITEREVERSEX reverses the x direction vector, sending the mario% sprite in the opposite direction. But, before leaving the IF statement SPRITEFLIP flips the mario% sprite horizontally, effectively turning it into a mirror image of itself. Notice also how each of the animation images are flipped as well. Once SPRITEFLIP has been applied to a sprite any image associated with that sprite will also be flipped. Even when you press a key to end mario%'s journey and lay him to rest, the last image of him is flipped accordingly.

When mario% reaches the left edge of the screen SPRITEFLIP is again used to set the sprite back to its default view by using NONE as the flipping behavior. As you can see, by using SPRITEFLIP you can create sprites that do not appear on your sprite sheet, keeping the size of the sprite sheet to a minimum while maximizing the images available to you.

Sprite Collision

Perhaps the easiest way to detect if sprites have collided is to imagine a circle surrounding each sprite with a radius that equals the farthest point of the sprite from the center. If one sprite's radius plus another sprite's radius is greater than the distance between the two then a collision can be assumed. This can be worked out by finding the difference between the center point x and y values, producing their vectors xv and yv, and then using Pythagoras to compute the distance between the two, $distance = \sqrt{xv^2} + yv^2$. If the distance is less than the two radii added together you have a collision. This works great for round objects, such as balls or slightly out of round objects such as a dented sphere, but more accuracy is required for most sprites because of them being inherently square or rectangular in shape.

To this end the sprite library offers two different types of collision detection the programmer. The first, and fastest, is box detection, where the sprites are treated as squares or rectangles and if either square or rectangular touches another a collision is assumed. This type of detection obviously works well on sprites that use most of their rectangular area for an image, or for very fast moving sprites that will probably always overlap when they get close to each other. The second type of collision detection offered is pixel perfect and is the slower of the two offered. When two sprites that have been defined to have pixel perfect detection enter into a box detection state, that is their rectangular areas collide, another procedure takes over that scans the affected area pixel by pixel looking for two pixels that are overlapping. For sprites that contain large areas of transparent pixels this can be a hindrance, as this large area will tend to collide well before an actual pixel collision happens. This is why in the sprite sheet creation section of this documentation it was recommended not to pad sprites with transparent pixels if they were not going to be used for static purposes. Always try to make your sprites as close to their rectangle size as possible.

The first command that deals with collision detection in the sprite library is SPRITECOLLIDE. The SPRITECOLLIDE command can be used to check two individual sprites for collision, or check one sprite against all sprites that are currently on the screen for collision.

SPRITECOLLIDE() – get the status of a sprite colliding with others.

The syntax for the SPRITECOLLIDE command is:

```
hit% = SPRITECOLLIDE(handle1%, handle2%)
```

handle1% must always be the sprite that you are testing collision with. *handle2%* can either be another specific sprite to check collision with, or the constant value ALLSPRITES, indicating that you want to test the sprite contained in *handle1%* with all sprites currently on the screen. If you specify two sprites to check collision with, and a collision occurs, then hit% will contain the value of *handle2%*. If you specify ALLSPRITES as the sprite for *handle2%* and a collision occurred with any of the sprites currently on screen, hit% will contain the sprite handle of the sprite that collided with *handle1%*. If no collisions occur in either scenario hit% will contain the value of zero.

Before collision detection can happen between sprites they need to be told what type of collision detection they will use. The command SPRITECOLLIDETYPE is used to tell a sprite what type of collision detection it will use when being checked with SPRITECOLLIDE.

SPRITECOLLIDETYPE() – set the type of collision detection to use with a sprite.

The syntax for the SPRITECOLLIDETYPE command is:

```
SPRITECOLLIDETYPE handle%, behavior%
```

handle% is the sprite that is getting a collision type set for and behavior% can be one three settings: NODETECT to turn collision detection off, BOXDETECT to use collision box or rectangular collision detection and PIXELDETECT to have the sprite use pixel perfect collision detection. It's important to remember that BOXDETECT always takes precedence over PIXELDETECT. If two sprites are being checked for collision and one is PIXELDETECT and the other BOXDETECT, rectangular collision detection will be used for both. Only when two sprites that have been defined using PIXELDETECT are checked for collision will pixel perfect detection be used.

SPRITECOLLIDEWITH() – which sprite collided with this sprite?

The last command that deals with collision detection is SPRITECOLLIDEWITH. When two sprites have collided after a check for collision has been performed the sprite they hit will be saved. You can use SPRITECOLLIDEWITH to get the sprite handle of the sprite that was hit. This may seem like a redundant command as SPRITECOLLIDE returns the handle of the sprite that collision occurred with. A good way to see which sprite collided with which is through a SELECT CASE statement. SPRITECOLLIDE can be used as a boolean with an IF statement when called, and since it saves the collision handle if a collision occurs, a check against SPRITECOLLIDEWITH can be done. For example:

```
IF SPRITECOLLIDE(mario%, ALLSPRITES) THEN
    SELECT CASE SPRITECOLLIDEWITH(mario%)
        CASE barrel%

        CASE fireguy%
```

and so on. There is no need to enter the SELECT CASE statement if no collision occurs and if one does, SPRITECOLLIDEWITH will yield the result of the collision. The syntax for the SPRITECOLLIDEWITH command is:

`hit% = SPRITECOLLIDEWITH(`*`handle%`*`)`

`hit%` will contain the last sprite handle that collided with the sprite in question. It's important to note that the handle value will not change until another collision happens. If you don't check for collision again and use `SPRITECOLLIDEWITH` to retrieve the sprite handle, it will still contain the value that was there from the last collision or no value if a collision has never happened.

Let's revisit the sample code that had barrels flying all over the screen. But this time Mario will be standing in the middle and if a barrel hits him the he jumps up in the air and the barrel speeds away at twice its normal speed. You can either type in the following code or load "collision.bas" which has been provided.

```
'$INCLUDE:'spritetop.bi'

CONST NUMBARRELS = 25

DIM barrels%(NUMBARRELS)
DIM barrel%
DIM mario%
DIM dksheet%
DIM count%
DIM background&

SCREEN _NEWIMAGE(800, 600, 32)
CLS
RANDOMIZE TIMER
background& = _LOADIMAGE("backg.png", 32)
dksheet% = SPRITESHEETLOAD("dkong.png", 64, 64, _RGB(0, 255, 0))
barrel% = SPRITENEW(dksheet%, 38, DONTSAVE)
mario% = SPRITENEW(dksheet%, 1, DONTSAVE)
_PUTIMAGE (0, 0), background&
SPRITEANIMATESET barrel%, 38, 41
SPRITEANIMATION barrel%, ANIMATE, FORWARDLOOP
SPRITECOLLIDETYPE barrel%, PIXELDETECT
SPRITECOLLIDETYPE mario%, PIXELDETECT
FOR count% = 1 TO NUMBARRELS
    barrels%(count%) = SPRITECOPY(barrel%)
    SPRITEPUT INT(RND(1) * 200) + 50, INT(RND(1) * 100) + 50, barrels%(count%)
    SPRITESPEEDSET barrels%(count%), RND(3) - RND(3) + 3
    SPRITEDIRECTIONSET barrels%(count%), INT(RND(1) * 360)
    SPRITEMOTION barrels%(count%), MOVE
NEXT count%
SPRITEFREE barrel%
DO
    _LIMIT 32
    _PUTIMAGE (0, 0), background&
    FOR count% = 1 TO NUMBARRELS
        SPRITEPUT 399, 299, mario%
        SPRITEPUT MOVE, MOVE, barrels%(count%)
        IF SPRITEX(barrels%(count%)) <= 0 THEN SPRITEREVERSEX barrels%(count%)
        IF SPRITEX(barrels%(count%)) >= _WIDTH(_DEST) THEN SPRITEREVERSEX barrels%(count%)
        IF SPRITEY(barrels%(count%)) <= 0 THEN SPRITEREVERSEY barrels%(count%)
        IF SPRITEY(barrels%(count%)) >= _HEIGHT(_DEST) THEN SPRITEREVERSEY barrels%(count%)
    NEXT count%
    IF SPRITECOLLIDE(mario%, ALLSPRITES) THEN
        SPRITEPUT MOVE, MOVE, SPRITECOLLIDEWITH(mario%)
        SPRITESET mario%, 4
    ELSE
        SPRITESET mario%, 1
    END IF
    _DISPLAY
LOOP UNTIL INKEY$ <> ""

'$INCLUDE:'sprite.bi'
```

While watching `mario%` jumping over the fast moving barrels did you notice something? You may have notice that at times some of the barrels that were clearly on top of him were not moving fast. This is because `SPRITECOLLIDE` will only detect one sprite collision. If there happens to be two or more sprites colliding with another sprite at the same time, the first collision seen is the one returned. This was done to maintain speed during collision detection subroutines. As soon as `SPRITECOLLIDE` sees a collision and it was told to check `ALLSPRITES` it returns the first collision it sees and ignores all others. If you need to see multiple collisions then you will have to check sprites individually in a loop of your own design. Just try to keep the number of collision checks to a minimum as you will notice a performance hit when the number of checks exceeds 50 or so. Trial and error on your part will yield the best results for your particular needs.

The Remaining Commands

The remaining commands in the sprite library are pretty self explanatory, so I am simply going to finish by listing each command and its syntax with a brief description of what it does. Please don't hesitate to contact me if you have any questions or comments relating to these last few commands.

SPRITEHIDE() – hiding a sprite from view.

`SPRITEHIDE` *`handle%`*

`SPRITEHIDE` will hide a sprite from view, but only if the sprite was defined as having the `SAVE` behavior when created with `SPRITENEW`. A hidden sprite will still retain its values, such as x and y position, but will not interact with commands such as `SPRITECOLLIDE`. I find it useful to use `SPRITEHIDE` as sort of a boolean operator as well, using `SPRITESHOWING` to test whether a sprite is still active in the game or not.

SPRITESHOW() – bring a sprite out of hiding.

`SPRITESHOW` *`handle%`*

`SPRITESHOW` is used to reverse the effect that `SPRITEHIDE` has on it.

SPRITESHOWING() – get the status of a sprite either hiding or showing.

`hidden% = SPRITESHOWING(handle%)`

`SPRITESHOWING` will return true (-1) if the sprite is currently shown or false (0) if not.

SPRITEZOOM() – change the size of a sprite.

`SPRITEZOOM` *`handle%, zoom%`*

`SPRITEZOOM` is used to resize a sprite from 1% to no upper limit. Supplying a value of 100 to *`zoom%`* will restore the sprite to its original size. When a sprite has been resized you can use the `SPRITECURRENTWIDTH` and `SPRITECURRENTHEIGHT` commands to get its width and height as well as `SPRITEX1`, `SPRITEY1`, `SPRITEX2` and `SPRITEY2` to get its upper left and lower right corner dimensions. This command would come in handy for games like 1942, where a plane would need to go in close to the ground for bombing runs and loops, but come back up for dogfights with other planes.

SPRITEZOOMLEVEL() – get a sprite's current zoom level or size.

`zoom% = SPRITEZOOMLEVEL(`*`handle%`*`)`

SPRITEZOOMLEVEL will retrieve the current zoom percentage value of a sprite.

SPRITESCORESET() – giving a sprite some value.

`SPRITESCORESET` *`handle%`*, *`score!`*

The idea behind SPRITESCORESET is to be able to assign a value, or points, to a sprite. In the Asteroids game a score value is set for each asteroid on the screen. When an asteroid is destroyed, its score value is added to the player's score. I've also found many other uses for the score value. Also in Asteroids the bullets are given a score value and each time the bullet moves the score value is decreased. When the score value reaches zero it signifies that the bullet has flown its maximum length and it's time to remove the bullet. You'll find many uses I'm sure for this general numerical register.

SPRITESCORE() – retrieve the value of a sprite.

`score! = SPRITESCORE(`*`handle%`*`)`

SPRITESCORE will retrieve a sprite's score value as set by SPRITESCORESET.

SPRITEX1() – the upper left x coordinate of a sprite.

SPRITEY1() – the upper left y coordinate of a sprite.

SPRITEX2() – the lower right x coordinate of a sprite.

SPRITEY2() – the lower right y coordinate of a sprite.

These four commands will retrieve the upper left and lower right coordinates of a sprite. These can be especially helpful when you want to find the corner coordinates of zoomed and/or rotated sprites, as rotated sprite sizes are ever changing. The syntax for all four of these commands is:

`x! = SPRITEX1(`*`handle%`*`)`

SPRITEMOUSE() – return the status of current mouse and sprite interaction.

`interaction% = SPRITEMOUSE(`*`handle%`*`)`

SPRITEMOUSE will return a mouse and sprite interaction value. There are four values that can be returned for `interaction%`:

- 0 = no mouse interaction – the constant NOMOUSE has been set to use for this test.

- 1 = left mouse button clicked on sprite – the constant `MOUSELEFT` has been set to use for this test.
- 2 = right mouse button clicked on sprite – the constant `MOUSERIGHT` has been set to use for this test.
- 3 = mouse pointer hovering over sprite – the constant `MOUSEHOVER` has been set to use for this test.

An example of code using this command might appear as:

```
mousestatus% = SPRITEMOUSE(mysprite%)
IF mousestatus% THEN
    SELECT CASE mousestatus%
        CASE MOUSELEFT

        CASE MOUSERIGHT

        CASE MOUSEHOVERING
```

and so on.

SPRITEMOUSEX() – the x location of the mouse on the sprite itself.

SPRITEMOUSEY() – the y location of the mouse on the sprite itself.

These two commands will reveal the x and y locations of the mouse pointer on the sprite itself. If the sprite the mouse pointer is currently on is 64 x 64 pixels, then the value range they may return would 0 to 63 accordingly. The syntax for both commands is:

`smousex% = SPRITEMOUSEX(`*`handle%`*`)`

SPRITEMOUSEAX() – the actual x location of the mouse on the screen.

SPRITEMOUSEAY() – the actual y location of the mouse on the screen.

These two commands will return the x and y locations of the mouse pointer of the screen behind the sprite that it is currently interacting with. The syntax for both commands is:

`smouseax% = SPRITEMOUSEAX(`*`handle%`*`)`

SPRITECURRENTWIDTH() – get the current width of a sprite.

SPRITECURRENTHEIGHT() – get the current height of a sprite.

These commands will yield the width and height of a sprite respectively. These are especially useful for getting the size of sprites that have been zoomed and/or rotated. The syntax for both commands is:

`swidth% = SPRITECURRENTWIDTH(`*`handle%`*`)`

Command Quick Reference

SUBROUTINES

SPRITEPUT(x!, y!, handle%) - Places a sprite on screen at coordinates INT(x!), INT(y!).

SPRITESHOW(handle%) - Unhide a sprite from view.

SPRITEHIDE(handle%) - Hide a sprite from view.

SPRITEZOOM(handle%, zoom%) - Change the size (zoom level) of a sprite.

SPRITEROTATE(handle%, degrees!) - Rotates a sprite from 0 to 360 degrees.

SPRITEFLIP(handle%, behavior%) - Flips a sprite horizontally, vertically, both or resets to no flipping.

SPRITESET(handle%, cell%) - Sets a sprite's image to a new image number on sprite sheet.

SPRITEANIMATESET(handle%, startcell%, endcell%) - Sets a sprite's animation sequence start and end sprite sheet cells.

SPRITEANIMATION(handle%, onoff%, behavior%) - Turns on or off automatic sprite animation with specified behavior.

SPRITENEXT(handle%) - Go to next cell of sprite's animation sequence.

SPRITEPREVIOUS(handle%) - Go to previous cell of sprite's animation sequence.

SPRITESTAMP(x%, y%, handle%) - Places a sprite on the background as if using a sprite stamp pad.

SPRITEFREE(handle%) - Removes a sprite from memory, freeing its resources.

SPRITECOLLIDETYPE(handle%, behavior%) - Sets the type of collision detection used for a sprite.

SPRITESCORESET(handle%, value!) - Sets the score value of a sprite.

SPRITETRAVEL(handle%, direction!, speed!) - Moves a sprite in the direction and speed indicated.

SPRITEDIRECTIONSET(handle%, direction!) - Sets a sprite's auto-motion direction angle.

SPRITESPEEDSET(handle%, speed!) - Sets a sprite's auto-motion speed in pixels.

SPRITEMOTION(handle%, behavior%) - Enables or disables a sprite's auto-motion feature.

SPRITESPINSET(handle%, spin!) - Sets a sprite's auto-motion spin direction.

SPRITEREVERSEY(handle%) - Reverses the y vector auto-motion value of a sprite.

SPRITEREVERSEX(handle%) - Reverses the x vector auto-motion value of a sprite.

SPRITETRAVEL(handle%, direction!, speed!) - Moves a sprite in the direction and speed indicated.

FUNCTIONS

SPRITEROTATION(handle%) - Gets the current rotation angle of a sprite in degrees.

SPRITENEW(sheet%, cell%, behavior%) - Creates a new sprite given the sheet and images that make up the sprite.

SPRITESHEETLOAD(filename$, spritewidth%, spriteheight%, transparent&) - Loads a sprite sheet into the sprite sheet array and assigns an integer handle value pointing to the sheet.

SPRITEX(handle%) - Returns the screen x location of a sprite. (integer) centered

SPRITEY(handle%) - Returns the screen y location of a sprite. (integer) centered

SPRITEAX(handle%) - Returns the actual x location of a sprite. (single) centered

SPRITEAY(handle%) - Returns the actual y location of a sprite. (single) centered

SPRITEX1(handle%) - Returns the upper left x screen position of the sprite.

SPRITEY1(handle%) - Returns the upper left y screen position of the sprite.

SPRITEX2(handle%) - Returns the lower right x screen position of the sprite.

SPRITEY2(handle%) - Returns the lower right y screen position of the sprite.

SPRITECOPY(handle%) - Makes a copy of a sprite and returns the newly created sprite's handle.

SPRITEMOUSE(handle%) - Returns the status of the current sprite and mouse pointer interaction.

SPRITEMOUSEX(handle%) - Returns the x location of the mouse on the sprite itself.

SPRITEMOUSEY(handle%) - Returns the y location of the mouse on the sprite itself.

SPRITEMOUSEAX(handle%) - Returns the x location of the mouse on the screen.

SPRITEMOUSEAY(handle%) - Returns the y location of the mouse on the screen.

SPRITECOLLIDE(handle%, handle2%) - Returns the status of collisions with other sprites.

SPRITEANGLE(handle%, handle2%) - Retrieves the angle in degrees between two sprites.

SPRITECOLLIDEWITH(handle%) - Returns the sprite number that collided with the specified sprite.

SPRITESCORE(handle%) - Retrieves the score value from a sprite.

SPRITESHOWING(handle%) - Returns the status of a sprite being hidden or not.

SPRITEDIRECTION(handle%) - Returns the direction angle a sprite's auto-motion has been set to.

SPRITEZOOMLEVEL(handle%) - Returns a sprite's current zoom level.

SPRITECURRENTHEIGHT(handle%) - Returns the current height of a sprite.

SPRITECURRENTWIDTH (handle%) - Returns the current width of a sprite.

CONSTANTS

Make sure not to use constants or variables with these names:

GLOBAL constants

NOVALUE = -32767

SPRITEFLIP constants

NONE = 0
HORIZONTAL = 1
VERTICAL = 2
BOTH = 3

SPRITENEW constants

SAVE = -1
DONTSAVE = 0

SPRITESHEETLOAD constants

NOTRANSPARENCY = -1
AUTOTRANSPARENCY = -2

SPRITEANIMATION constants

ANIMATE = -1
NOANIMATE = 0
FORWARDLOOP = 0
BACKWARDLOOP = 1
BACKFORTHLOOP = 2

SPRITEMOUSE constants

NOMOUSE = 0
MOUSELEFT = 1
MOUSERIGHT = 2
MOUSEHOVER = 3

<u>SPRITECOLLIDETYPE constants</u>

NODETECT = 0
BOXDETECT = 1
PIXELDETECT = 2

<u>SPRITECOLLIDE constants</u>

ALLSPRITES = -1

<u>SPRITEMOTION constants</u>

MOVE = -1
DONTMOVE = 0

TYPE DECLARATIONS

Make sure not to use the following as type declarations, constants or variables:

TYPE SPRITE

TYPE SHEET

ARRAYS

Make sure not to use the following array names as constant or variable names:

sprite(1) AS SPRITE

sheet(1) AS SHEET