The Atari ST M68000 tutorial part 11 – of making the mountain move to Mohammed

Well well, finally, as promised, we will delve into the technique of sprites; the essence of a platform or shoot- em- up game, and lots of other stuff. In fact, anything that needs something moving that is not 3D or real time rendered (that is, it's being drawn while the program runs, and not stored previously as a picture). It was really challenging and great fun to code this one, and it's probably the most satisfying coding experience ever, I hope I can convey the knowledge it brought me.

In the last tutorial, we learned something on pixels, in order to be able to address a single pixel anywhere, the data must be shifted into a correct position. Why is this? Because, each instruction except the bit instructions, deal with at least byte size. What it means is that if we use instructions with byte size, all pixels "snap" at 8 pixels, because that's the minimum addressable size. However, by shifting the data before using it in graphic instructions, we can in a way address any pixel we want to.

I actually suggest you load up the pre- assembled program, and both the picture files that comes with the tutorials, the pictures being AUTUMN.PI1 and SPRITE.PI1. A little note on the pictures, they are in STe palette, meaning that they will look a bit ugly on a ST, but STeem should handle this nicely. Yes, the character seen is the same one as in TUT9: Kenshin. He's the main character in a Japanimation, a former assassin for the government who now tries to atone by living a quiet life and helping people. This series is awesome and has given me much inspiration, the first Kenshin OVA series is one of the most beautiful pieces of art I've ever seen.

So, after you've been impressed by the Tai Ji symbol (a.k.a. Yin and Yang symbol, Yin and Yo in Japanese) bouncing around the screen, you are eager to learn for yourself, right? As you can see, the background is provided in the AUTUMN.PI1, and the bouncing ball, which is the sprite, is in SPRITE.PI1. Actually, only 14 colours are used for the background, the last two being reserved for the sprite, this isn't necessary and the sprite may well share colours with the background. The sprite seems to appear twice, in the SPRITE.PI1, there are two balls, one of them is the sprite mask, if confusion occurs, just read on.

Painting the background is easy, just smack in the pixel data and set the palette, bouncing will be dealt with later, what we need to focus on now is getting the sprite nicely on the screen, and being able to put it anywhere on the screen, preferably expressing the location in X and Y coordinates for human compatibility. How exactly to put the sprite data on screen, the most obvious choice is a move instruction. This won't do at all though, check this out.

Screen memory

%00000000	%00001110	first word
%00000000	%00000000	second word
%00000000	%00001011	third word
%00000000	%01010101	fourth word

Pixel colours

$00000000	$0808595C	

Sprite data

%00000000	%00000001	first word
%00000000	%00100000	second word
%00000000	%00000000	third word
%00000000	%00001010	fourth word

Pixel colours

$00000000	$00208081

Now, if we move the sprite data onto the graphics memory, we get

Screen memory

%00000000	%00000001	first word
%00000000	%00100000	second word
%00000000	%00000000	third word
%00000000	%00001010	fourth word

Pixel colours

$00000000	$00208081

Move instructions destroy all data and replace it with new, in other words, the background is completely lost and the sprite has taken over completely. Doing it like this will also create an ugly squared looking sprite, since the sprite background will not be transparent (actually rectangular, but more on this below). This will not do. Or instructions, on the other hand, will not overwrite the original data, we try an or instruction with the above configuration

Screen memory

%00000000	%00001111	first word
%00000000	%00100000	second word
%00000000	%00001011	third word
%00000000	%01011111	fourth word

Pixel colours

$00000000	$0828D9DD

Dang! By or:ing in the sprite, we mixed the sprite with the background, this is also bad since the sprite will not look as it should, although it will create quite a nice effect and is good if you simply want a “colour distortion” effect, but we don’t want that now. An exclusive or would only flip the colours around in strange ways, and an and instruction clears data. But wait, if we clear out the sprite data, with an and, leaving the background intact, and then or in the sprite, it would all work. The sprite mask has the same look as the sprite, but is only two colours. Colour 15 where the background is, making sure all bits there are set, and colour 0 where the real sprite form is, making sure all bits are cleared. Have a look at SPRITE.PI1,

and you will see clearly (well, ok, in the picture, the mask is colour 15 and the background is colour 0, but it will get inverted later, read on ...).

```
Sprite mask
%11111111          11010100  first word
%11111111          11010100  second word
%11111111          11010100  third word
%11111111          11010100  fourth word

Pixel colours
$FFFFFFFF  FF0F0F00
```

All pixels that were colour 0 (background) in the sprite, are now colour 15 (F), and all pixels that had one colour or another in the sprite are now colour 0. By and:ing the sprite mask with the background, we will make sure to clear out all sprite pixels (since they get and:ed with 0) and keeping the status of all other bits (since they are and:ed with 1). It is imperative that you understand this step, if you don't, reread the Boolean algebra part in TUT9, check some external sources and think again, or send me an e-mail :) After applying the mask, the screen memory will look like this

```
Screen memory
%00000000          %00000100          first word
%00000000          %00000000          second word
%00000000          %00000000          third word
%00000000          %01010100          fourth word

Pixel colours
$00000000          $08080900
#bbbbbbbb          #bbsbsbss  b = background, s = sprite
```

As you can see, the background has been preserved, while everything concerning the sprite is wiped out. Now is the time to or in the sprite data; this instruction will in no way affect the background (since the background colour in the sprite is 0). This is what it will look like after the or operation:

```
Screen memory
%00000000          %00000101          first word
%00000000          %00100000          second word
%00000000          %00000000          third word
%00000000          %01011110          fourth word

Pixel colours
$00000000          $08288981
#bbbbbbbb          #bbsbsbss  b = background, s = sprite
```

To summarise; first we take an inverted version of our sprite with only two colours, and and that with the background. This clears all pixels that are concerned with the sprite and leaves the background intact. After the

mask is applied, it is safe to or in the sprite data, since the previous clearing of the sprite pixels, there is no risk of mixing the sprite with the background. The background in the sprite is colour 0, thus the or instruction will have no effect on the background, the background part in the mask is colour 15 (all 1's) and thus the and instruction will not affect the background.

OK, now we know how to put the sprite on screen, but we are still faced with the problem of not being able to put it anywhere. To solve this, the sprite and mask data must be shifted. Like with the putpixel, in order to put the sprite at say 0,2, we need to shift the sprite data right two bits. With the putpixel, we shifted "real time", but there is much more data involved in a sprite, so we'll be pre-shifting the sprite instead. When using the pre-shifted method, we assign a storage area that is 16 times larger than the sprite data, and store the sprite in that area shifted in all possible sixteen combinations we need.

I see a big '?' in your face right now. Think about it, in the putpixel routine, we could end up shifting the pixel 15 bits to the right, at most, so what we do here with the sprite is to store all those possibilities after one another. When the time comes to put the sprite out, instead of shifting the original sprite data, all we have to do is access the storage area with the correct offset. An offset is the value added to the starting address of something. For example, the middle of the screen is the screen address with an offset of 100*160+80 = 16080 bytes.

```
Sprite data
$00001111          ...

Sprite storage area
$00001111          ...     first position, offset 0
$00000111          ...     second position, offset 1
$00000011          ...     third position, offset 2
$00000001          ...     fourth position, offset 3
```

(the offset number is completely fictional, it's not even an even number)

Let's say we want to put the sprite at 0,2, we know what we have to do. We have to point to the start of the screen memory, shift the sprite data right by 2, and put it in place. Say a0 points to the screen memory, and a1 to the sprite storage area, then we just need to add offset 2 to a1, and a1 will point to correctly shifted sprite data. Pre-shifting is way faster than loading up the sprite data in a1, and then shifting it, especially since the sprite data consists of several words that all need to be shifted. The downside of course is loss of memory.

Now, there is a problem here, if we have a 32*32 pixel sprite, like in the sample program, the data for the sprite is 16*32 bytes, arranged like this:

```
Sprite data
First 16    Last 16 pixels      (W = word)
WWWW        WWWW                first line
WWWW        WWWW                second line
WWWW        WWWW                third line
```

... and so on for a total of 32 lines

When we begin to shift, we want the last bit that goes out the first word, to be shifted in as the first bit in the fifth word. This is comparable to the tutorials on scrolling. The last bit that go out the fifth word, should not go into the first bit of the ninth word, because then a pixel from the first line would go into the second line, but there is no room to shift it out right on the first line. So, we have to add a buffer to every line so that no data will be lost in the shift. In the last shift, the first four words will be all but empty, and the buffer will be all but full. So the sprite storage area will have to look like this.

```
Sprite storage area
First 16    Middle 16   Last 16 pixels          (B = buffer, word size)
WWWW        WWWW        BBBB        first line
WWWW        WWWW        BBBB        second line
WWWW        WWWW        BBBB        third line
```

... and so on for a total of 32 lines and 16 such blocks to cover all possible shifts

Even though the sprite storage area covers a total of 48 pixels, 16 of these will be 0, thus not affecting the background. See the sprite as a 48 pixel wide block, with only 32 pixels coloured. Within this 48 pixel block, the 32 colour pixels will be shifted more and more to the right as X coordinates increase, then when it becomes critical, the block will move 16 pixels to the right in one sweep, and the 32 pixel colour area will be reset, starting the procedure all over again. Run the TUT11BLK.PRG, to see this clear.

Alright, theory part on pre- shifting done, now we need it in direct coding practice as well. First off, we'll need a good instruction with which to shift. Sure, lsr seems a good choice, but we need to be able to preserve the bit that gets shifted out, and lsr doesn't preserve anything. The instruction roxr, for ROtate eXtended Right, is good in this case. The extended bit is rotated in from the left, and the bit rotated out the right is saved in the carry and extended flag. So, by roxr:ing with one each time, we will save what we shift out, and shift it in the next time around. (btw, when speaking about the user bits in the status register, flags and bits are used synonymous) Looki looki

```
                            (X = extended flag)
                            d0 = %00001101      X = 0
roxr        #1,d0           d0 = %00000110      X = 1
roxr        #1,d0           d0 = %10000011      X = 0
roxr        #1,d0           d0 = %01000001      X = 1
```

What we do is to first copy the sprite data to the sprite storage area, then we take the data from the storage area, rotate extended right with one, and save that data into the next position of the storage area. What we have to think about when coding this is that the data from the first word, goes into the fifth word and so on. In code, it looks like this

```
move.l      #spr_dat,a0                 original sprite data
```

```
            add.l       #34,a0              skip palette
            move.l      #sprite,a1          storage of pre-shifted sprite

            move.l      #32-1,d0            32 scan lines per sprite
first_sprite
            move.l      (a0)+,(a1)+                 move from original to
pre-shifted
            move.l      (a0)+,(a1)+
            move.l      (a0)+,(a1)+
            move.l      (a0)+,(a1)+                 32 pixels moved
            add.l       #8,a1               jump over end words
            add.l       #144,a0             jump to next scan line
            dbf         d0,first_sprite
```

First, point to the sprite data, jump over the palette and load up the sprite storage area, which is a ds.l 3072: 16 bytes per line, plus 8 for the buffer totalling 24 bytes per scan line. The sprite is 32 lines and there should be 16 such blocks. This adds up to 24*32*16 = 12288 bytes, which is 3072 long words. In the loop, just copy data from the sprite picture to the storage area, the buffer word area is skipped since it contains nothing at this time. Now comes the challenging part, writing the generic pre-shift.

```
            move.l      #sprite,a0          point to beginning of storage
area
            move.l      #sprite,a1          point to beginning of storage
area
            add.l       #768,a1             point to next sprite position

            move.l      #15-1,d1            15 sprite positions left
positions
            move.l      #32-1,d2            32 scan lines per sprite
line
            move.l      #4-1,d3             4 bit planes
plane
            move.w      (a0),d0             move one word
            roxr        #1,d0               pre-shift
            move.w      d0,(a1)             put it in place

            move.w      8(a0),d0            move one word
            roxr        #1,d0               pre-shift
            move.w      d0,8(a1)            put it in place

            move.w      16(a0),d0           move one word
            roxr        #1,d0               pre-shift
            move.w      d0,16(a1)           put it in place

            add.l       #2,a0               next bit plane, also clears X flag
            add.l       #2,a1               next bit plane

            dbf         d3,plane
```

```
add.l     #16,a1              next scan line
add.l     #16,a0              next scan line

dbf       d2,line

dbf       d1,positions
```

First off, load up the storage area in a0 and a1, and make a1 point to the next storage area. This one is empty and should contain the sprite data shifted one bit to the right. Since we have already filled the first position in the storage area, 15 positions are left. 32 lines to each sprite and 4 bit planes to each line. Since all these are treated the same way, we only need one big loop so to speak.

Now comes the fun part, put the first word in d0, this word comes from the previous storage position. Rotate it, and put it in at the next storage position. Now the extended flag holds the bit that was shifted out the right, and this one needs to be shifted in on the left in the first word in the next word cluster. So, a byte offset of 8 (4 words) is added when fetching and storing the next word. The buffer must also come into play, so the last word will get a byte offset of 16. Now, we have pre- shifted three words.

By adding 2 to both a1 and a0 we will be at the next bit plane. It will also clear the extended flag, which is good because otherwise a bit from the last word might come over to the first word on the next bit plane, which is undesirable. Repeat for all four bit planes. We have now moved a line of the sprite. After the four bit planes have been rotated, a0 and a1 will point to the first word in the second 16 pixel cluster, or 8 bytes from the beginning of the data. By adding 16, we will point to the next scan line (16+8 = 24). Repeat for 15 positions. Pretty compact explanation, yes? A graphical representation follows

```
Storage area with data, beginning at $0
16          16          16 pixels
WWWW     WWWW     WWWW
... for 32 lines
0  2  4  6    8 10 12 14 16 18 20 22          byte offset

Storage area without data, beginning at $768
16          16          16 pixels
0000        0000        0000        (each 0 is word size)
... for 32 lines
0  2  4  6    8 10 12 14 16 18 20 22          byte offset

a0 = $0
a1 = $768

move.w    (a0),d0
roxr      d0                  C = leftmost bit from W offset 0
move.w    d0,(a1)             put rotation in 0 at offset 0
```

```
            move.w      8(a0),d0            as you see, first word of second
cluster
            roxr        d0                  bit preserved and shifted from
offset 0
            move.w      d0,8(a1)            put it at offset 8

            move.w      16(a0),d0           first word last cluster
            roxr        d0                  rotate, carry bit may now be set
            move.w      d0,16(a1)           at offset 16

            add.l       #2,a0               next bit plane, watch offset
            add.l       #2,a1               also clears X flag
```

Finally, a0 and a1 will both be at offset 8, the first word of the first bit plane, by adding 16 to this, the offset will be 24, the value for a whole line, effectively putting us at the beginning of the next line. That concludes the pre-shift of the sprite.

The mask data has to be pre-shifted a bit differently. Where the sprite colour is, we need the mask to be 0, and where the background is, the mask must be 1, as explained above. A look at the sprite picture will show that the sprite colour area is colour 15, all 1's, and the background is colour 0, all 0's. For the mask to be correctly pre-shifted, we need to invert it, making the background all 1's and the sprite colour area all 0's. When shifting, we must also always be shifting in 1's, not 0's as the case was with the sprite data.

The instruction not, for NOT :), will take any value and invert it, this means changing all 1's to 0's and all 0's to 1's. In order to have all bits except those concerning the sprite colour area set, we must make sure to put 1's in the buffer area. Also, at the beginning of each plane loop, we must also make sure that the highest bit of d0 is set, so that 1's are shifted in. Other than that, the sprite and mask pre-shift share ideas. The mask area is as big as the sprite area. Even though this isn't necessary since all bit planes in the sprite look alike, we could have reduced the size by ¾, but for ease of understanding, this was not done.

```
            move.l      #spr_dat,a0
            add.l       #34+160*32,a0       skip palette and sprite
            move.l      #mask,a1            load up mask part

            move.l      #32-1,d0            32 scan lines per sprite
first_mask
            move.l      (a0)+,(a1)          move from original to pre-
shifted
            not.l       (a1)+               invert the mask data
            move.l      (a0)+,(a1)
            not.l       (a1)+               invert the mask data
            move.l      (a0)+,(a1)
            not.l       (a1)+               invert the mask data
            move.l      (a0)+,(a1)
            not.l       (a1)+               invert the mask data
            move.l      #$ffffffff,(a1)+    fill last two words...
```

```
            move.l      #$ffffffff,(a1)+        ... with all 1's

            add.l       #144,a0                 jump to next scan line
            dbf         d0,first_mask
* the picture mask has been copied to first position in pre-shift

            move.l      #mask,a0                point to beginning of storage
area
            move.l      #mask,a1                point to beginning of storage
area
            add.l       #768,a1                 point to next mask position

            move.l      #15-1,d1                15 sprite positions left
positions_mask
            move.l      #32-1,d2                32 scan lines per sprite
line_mask
            move.l      #4-1,d3                 4 bit planes
plane_mask
            move.w      (a0),d0                 move one word
            roxr        #1,d0                   pre-shift
            or.w        #%1000000000000000,d0       make sure most
significant bit set
            move.w      d0,(a1)                 put it in place


            move.w      8(a0),d0                move one word
            roxr        #1,d0                   pre-shift
            move.w      d0,8(a1)                put it in place

            move.w      16(a0),d0               move one word
            roxr        #1,d0                   pre-shift
            move.w      d0,16(a1)               put it in place

            add.l       #2,a1                   next bit plane
            add.l       #2,a0                   next plane, clears X flag (bad)

            dbf         d3,plane_mask

            add.l       #16,a1                  next scan line
            add.l       #16,a0                  next scan line

            dbf         d2,line_mask

            dbf         d1,positions_mask
```

Unlike the sprite pre-shift where we could set up the storage area with direct memory moves from the sprite picture to the storage area, here we move data, and then perform the not instruction to invert the data. Also, instead of just skipping the buffer area like in the sprite, here we fill it with 1’s. The bit plane loop is almost identical, with the one exception that the first

shift must be guaranteed to shift in a 1, not a 0. A simple or instruction will make sure the most significant bit is set. That was that, all pre-shifting done.

The method which we use to get the coordinates is the exact one found in tutorial 10. So when we send in our coordinates, we will be provided with a pointer to the screen address, and the number of shifts to be done in d0. The number in d0 is an offset for the sprite data and mask data. By putting the address to the sprite data in an address register, multiplying d0 with 768 and adding that to the address register, we will get a pointer to correctly shifted sprite data. The reason for the number being 768 is that it is the size of a sprite block.

OK, now comes the problem of actually moving the sprite. We can put a sprite at any coordinate we want, but we can't move it yet. A simple bounce routine here, the sprite will move with a certain X speed and a certain Y speed, and change direction when it hits "walls" (edges of the screen). What we need is a heading, and a speed. For simplicity, we express the heading as either 1 or 0 for both X and Y respectively. 1 is towards bottom right and 0 is towards upper left. X heading is either right or left, and Y heading either up or down. The X and Y speed is how many pixels to move the sprite in desired direction each VBL. So with an X heading of 1, and an X speed of 2, the sprite would move 2 pixels right each VBL.

What the move routine needs to do is to add X and Y coordinates in accordance with heading and speed, as well as checking for wall hits. When a wall hit occurs, the sprite must change direction. A change in direction simply means flipping between 1 or 0 in heading. This might be a good time to tell about the equ, for EQUals method. Any label can have an equ applied to it, meaning that whenever one uses the label, it is replaced by the equ. Easy huh?

```
number    equ       2
          move.l    #number,d0                    same as move.l #2,d0
```

One can say that equ's, are constants. It's good practice to have as many equ's as possible, because if you realize you have to change a constant, you only need to change it in one place instead of every place the constant appears. X speed and Y speed is a good example (unless you want variable speed), X coordinate is a terrible thing, since it needs to change all the time. Actually, I think it's best to express the move routine in pseudo code first.

```
If (x_coord > 319 – 32 – x_speed + 1) Then
          x_heading = 0
If (x_coord < 0) Then
          x_heading = 1
If (y_coord > 199 – 32 – y_speed + 1) Then
          y_heading = 0
If (y_coord < 0) Then
          y_heading = 1
```

First we check to see if the heading needs change, as long as the sprite is in any way outside the screen coordinates, we need to change the heading. Since we check the heading before we move the sprite, and move the sprite before drawing it, the sprite will never be drawn off screen. The only

trouble here is where all numbers come from. Think of it first without x_speed added, every VBL the sprite just moves one pixel. Then the formula is x_coord > 319 – 32. This is easy to grasp, the X coordinate must not be more than the screen can hold, which is 319, minus the width of the sprite itself of course, which is 32.

The so called “hot spot” of the sprite is the upper left corner. This is the point against which all sprite coordinates are measured. We say that the sprite is at coordinates 13,13, but this really means that the sprite hot spot is at 13,13. Exactly what pixels the sprite inhabits is unknown to us, since the sprite can have any form, but for simplicity, we think of the sprite as a square, with the coordinates in the upper left corner. Thus, when seeing if the sprite hits the right wall, we take the coordinates of the upper left corner, the hot spot, and add the width of the sprite.

The x_speed is also to be taken into account. Imagine the sprite moving with 100 pixels per VBL, then the sprite will be way outside the screen if it’s anywhere over the right half of the screen, so the sprite is only ok if it’s on the left half of the screen, obviously, the speed must be taken into account. Think of the speed as just enlargement to the sprite. The Y check works exactly the same way, but with a different max coordinate for obvious reasons. It looks a bit different in assembly language though.

```
            cmp       #319-32-x_speed+1,x_coord
            blt       x_right_ok              see if x is < 319-32 for width
            move.w    #0,x_heading                    if x >=319, change
heading
x_right_ok

            cmp       #0,x_coord
            bgt       x_left_ok               see if x is > 0
            move.w    #1,x_heading                    if x <=0, change
heading
x_left_ok

            cmp       #199-32-y_speed+1,y_coord
            blt       y_low_ok                see if y is < 199-32 for lines
            move.w    #0,y_heading                    if y >=199, change
heading
y_low_ok

            cmp       #0,y_coord
            bgt       y_high_ok               see if y is > 0
            move.w    #1,y_heading                    if y <=0, change
heading
y_high_ok
```

We check if the X coordinate is lesser than the number, and if it is, it’s ok and a little branch will skip the changing of the X heading. Whereas the pseudo code’s If statements took place if the check was true, our checks affect if the statements are false. This may look messy, but it’s really quite simple,

just take a second look at it. We also need to update the coordinates, here's some more pseudo code.

```
If (x_heading = 0) Then
        x_coordinate = x_coordinate – x_speed
Else
        x_coordinate = x_coordinate + x_speed
If (y_heading = 0) Then
        y_coordinate = y_coordinate – y_speed
Else
        y_coordinate = y_coordinate + y_speed
```

No problem there, just change the coordinates according to speed and heading. In assembly language it becomes more troublesome though.

```
        cmp     #0,x_heading            check x heading
        bne     x_move_right    if 1, move right, otherwise left
        sub.w   #x_speed,x_coord        move sprite left
        bra     x_move_done     done moving sprite in x
x_move_right
        add.w   #x_speed,x_coord        move sprite right
x_move_done

        cmp     #0,y_heading            check y heading
        bne     y_move_down     if 1, move down, otherwise up
        sub.w   #y_speed,y_coord        move sprite up
        bra     y_move_done     done moving sprite in y
y_move_down
        add.w   #y_speed,y_coord        move sprite down
y_move_done
```

First, a check to see if X heading is 0, if it is, move to the left, otherwise move to the right. If we move to the left, we subtract the X coordinate by the X speed, and we must also make sure to jump past the move to the right. The Y part is exactly as the X part. Again, just look one more time at the code and if it seems confusing, write it down on paper if you must and go through the different possible branches, it's not too complex once you structuralize it.

Hoah, this takes time to explain, hope you're still with me 'cus we are almost done. Now we know how to pre- shift, apply the sprite and move it. One would think that we have all we need, there is just one more thing to take into account. If we would apply the things we know and fire away, we would have a sprite that moves over the screen and leaves a trail. The damn thing will never go away, making it most ugly. Why? Because the background must be restored when the sprite has passed it.

So, on every VBL, the background must first be restored, then it must be saved after the sprite coordinates are updated, since the save and restore routine is dependent on the sprite coordinates. Then we can paint the sprite. The save routine just copies a sprite sized block from the screen

memory into a save buffer, and the restore routine copies the data from the buffer onto the screen.

What we have now is a main routine that restores background, moves the sprite (rather updates the sprite coordinates), saves the background, applies the mask and lastly paints the sprite. All of this is so fast, that we don't even have to bother with double buffering, so we pull a fast one and just skip that. Here comes the entire source code, don't panic, most of the stuff will be familiar.

```
x_speed     equ         2           how many x coord to move each VBL
y_speed     equ         1           how many y coord to move each VBL

            jsr         initialise

* pre-shifting sprite
            move.l      #spr_dat,a0                 original sprite data
            add.l       #34,a0              skip palette
            move.l      #sprite,a1          storage of pre-shifted sprite

            move.l      #32-1,d0            32 scan lines per sprite
first_sprite
            move.l      (a0)+,(a1)+                 move from original to
pre-shifted
            move.l      (a0)+,(a1)+
            move.l      (a0)+,(a1)+
            move.l      (a0)+,(a1)+                 32 pixels moved
            add.l       #8,a1               jump over end words
            add.l       #144,a0             jump to next scan line
            dbf         d0,first_sprite
* the picture sprite has been copied to first position in pre-shift

            move.l      #sprite,a0          point to beginning of storage
area
            move.l      #sprite,a1          point to beginning of storage
area
            add.l       #768,a1             point to next sprite position

            move.l      #15-1,d1            15 sprite positions left
            positions
            move.l      #32-1,d2            32 scan lines per sprite
line
            move.l      #4-1,d3             4 bit planes
plane
            move.w      (a0),d0             move one word
            roxr        #1,d0               pre-shift
            move.w      d0,(a1)             put it in place


            move.w      8(a0),d0            move one word
            roxr        #1,d0               pre-shift
```

```
            move.w      d0,8(a1)            put it in place

            move.w      16(a0),d0           move one word
            roxr        #1,d0               pre-shift
            move.w      d0,16(a1)           put it in place

            add.l       #2,a0               next bit plane, also clears X flag
            add.l       #2,a1               next bit plane

            dbf         d3,plane

            add.l       #16,a0              next scan line
            add.l       #16,a1              next scan line

            dbf         d2,line

            dbf         d1,positions
* pre-shift of sprite done, all 16 sprite possitions saved in sprite


* pre-shifting mask
            move.l      #spr_dat,a0
            add.l       #34+160*32,a0       skip palette and sprite
            move.l      #mask,a1            load up mask part

            move.l      #32-1,d0            32 scan lines per sprite
first_mask
            move.l      (a0)+,(a1)          move from original to pre-
shifted
            not.l       (a1)+               invert the mask data
            move.l      (a0)+,(a1)
            not.l       (a1)+               invert the mask data
            move.l      (a0)+,(a1)
            not.l       (a1)+               invert the mask data
            move.l      (a0)+,(a1)
            not.l       (a1)+               invert the mask data
            move.l      #$ffffffff,(a1)+    fill last two words...
            move.l      #$ffffffff,(a1)+    ... with all 1's

            add.l       #144,a0             jump to next scan line
            dbf         d0,first_mask
* the picture mask has been copied to first position in pre-shift

            move.l      #mask,a0            point to beginning of storage
area
            move.l      #mask,a1            point to beginning of storage
area
            add.l       #768,a1             point to next mask position

            move.l      #15-1,d1            15 sprite positions left
```

```
positions_mask
            move.l      #32-1,d2                32 scan lines per sprite
line_mask
            move.l      #4-1,d3                 4 bit planes
plane_mask
            move.w      (a0),d0                 move one word
            roxr        #1,d0                   pre-shift
            or.w        #%1000000000000000,d0       make sure most
significant bit set
            move.w      d0,(a1)                 put it in place

            move.w      8(a0),d0                move one word
            roxr        #1,d0                   pre-shift
            move.w      d0,8(a1)                put it in place

            move.w      16(a0),d0               move one word
            roxr        #1,d0                   pre-shift
            move.w      d0,16(a1)               put it in place

            add.l       #2,a0                   next bit plane, clears X flag (bad)
            add.l       #2,a1                   next bit plane
            dbf         d3,plane_mask

            add.l       #16,a0                  next scan line
            add.l       #16,a1                  next scan line

            dbf         d2,line_mask

            dbf         d1,positions_mask
* pre-shift of mask done, all 16 sprite possitions saved in mask


            movem.l     bg+2,d0-d7
            movem.l     d0-d7,$ff8240

            move.l      #bg+34,a0               pixel part of background
            move.l      $44e,a1                 put screen memory in a1
            move.l      #7999,d0                8000 longwords to a screen
pic_loop
            move.l      (a0)+,(a1)+                 move one longword
to screen
            dbf         d0,pic_loop                 background painted

            jsr         save_background         something in restore buffer

            move.l      $70,old_70                  backup $70
            move.l      #main,$70               put in main routine

            move.w      #7,-(a7)
            trap        #1
```

```
            addq.l      #2,a7                   wait keypress

            move.l      old_70,$70                      restore old $70

            jsr         restore

            clr.l       -(a7)
            trap        #1                      exit

main
            movem.l     d0-d7/a0-a6,-(a7)       backup registers

            jsr         restore_background
            jsr         move_sprite
            jsr         save_background
            jsr         apply_mask
            jsr         put_sprite

            movem.l     (a7)+,d0-d7/a0-a6       restore registers

            rte


move_sprite
* moves the sprite one pixel in x and y
* see if any headings need to be changed
            cmp         #319-32-x_speed+1,x_coord
            blt         x_right_ok              see if x is < 319-32 for width
            move.w      #0,x_heading                    if x >=319, change
heading
x_right_ok

            cmp         #0,x_coord
            bgt         x_left_ok               see if x is > 0
            move.w      #1,x_heading                    if x <=0, change
heading
x_left_ok

            cmp         #199-32-y_speed+1,y_coord
            blt         y_low_ok                see if y is < 199-32 for lines
            move.w      #0,y_heading                    if y >=199, change
heading
y_low_ok

            cmp         #0,y_coord
            bgt         y_high_ok               see if y is > 0
            move.w      #1,y_heading                    if y <=0, change
heading
y_high_ok
* all eventual heading changes now made
```

```
* move sprite coordinates (change coordinates)
            cmp         #0,x_heading                    check x heading
            bne         x_move_right            if 1, move right, otherwise left
            sub.w       #x_speed,x_coord        move sprite left
            bra         x_move_done             done moving sprite in x
x_move_right
            add.w       #x_speed,x_coord        move sprite right
x_move_done

            cmp         #0,y_heading                    check y heading
            bne         y_move_down             if 1, move down, otherwise up
            sub.w       #y_speed,y_coord        move sprite up
            bra         y_move_done             done moving sprite in y
y_move_down
            add.w       #y_speed,y_coord        move sprite down
y_move_done
* finnished moving sprite

            rts


apply_mask
* applies the mask to the background
            jsr         get_coordinates
            move.l      #mask,a0
            mulu        #768,d0                 multiply position with size
            add.l       d0,a0                   add value to mask pointer

            move.l      #32- 1,d7               mask is 32 scan lines
maskloop
            rept        6                       mask is 6*4 bytes width
            move.l      (a0)+,d0                mask data in d0
            move.l      (a1),d1                 background data in d1
            and.l       d0,d1                   and mask and picture data
            move.l      d1,(a1)+                move masked data to
background
            endr
            add.l       #136,a1                 next scan line
            dbf         d7,maskloop

            rts


put_sprite
* paints the sprite to the screen
            jsr         get_coordinates
            move.l      #sprite,a0
            mulu        #768,d0                 multiply position with size
            add.l       d0,a0                   add value to sprite pointer
```

```
            move.l      #32-1,d7            sprite is 32 scan lines
bgloop
            rept        6                   sprite is 6*4 bytes width
            move.l      (a0)+,d0            sprite data in d0
            move.l      (a1),d1             background data in d1
            or.l        d0,d1               or sprite and background data
            move.l      d1,(a1)+            move ored sprite data to
background
            endr
            add.l       #136,a1
            dbf         d7,bgloop

            rts


save_background
* saves the background into bgsave
            jsr         get_coordinates
            move.l      #bgsave,a0

            move.l      #32-1,d7            sprite is 32 scan lines
bgsaveloop
            rept        6                   sprite is 6*4 bytes width
            move.l      (a1)+,(a0)+                 copy background to
save buffer
            endr
            add.l       #136,a1             next scan line
            dbf         d7,bgsaveloop

            rts


restore_background
* restores the background using data from bgsave
            jsr         get_coordinates
            move.l      #bgsave,a0

            move.l      #32-1,d7            sprite is 32 scan lines
bgrestoreloop
            rept        6                   sprite is 6*4 bytes width
            move.l      (a0)+,(a1)+                 copy save buffer to
background
            endr
            add.l       #136,a1             next scan line
            dbf         d7,bgrestoreloop

            rts
```

```
get_coordinates
* makes a1 point to correct place on screen
* sprite position in d0.b
            move.l      $44e,a1             screen memory in a1
            move.w      y_coord,d0                  put y coordinate in
d0
            mulu        #160,d0             160 bytes to a scan line
            add.l       d0,a1               add to screen pointer
            move.w      x_coord,d0                  put x coordinate in
d0
            divu.w      #16,d0              number of clusters in low, bit in
high
            clr.l       d1                          clear d1
            move.w      d0,d1               move cluster part to d1
            mulu.w      #8,d1               8 bytes to a cluster
            add.l       d1,a1               add cluster part to screen
memory
            clr.w       d0                  clear out the cluster value
            swap        d0                  bit to alter in low part of d0

            rts


            include     initlib.s

            section data
x_coord                 dc.w        0
y_coord                 dc.w        0
x_heading   dc.w        1
y_heading   dc.w        1

spr_dat     incbin      SPRITE.PI1
bg          incbin      AUTUMN.PI1
old_70      dc.l        0


            section bss
sprite      ds.l        3072        32/2+8*32 bytes * 16 positions / 4 for long
mask        ds.l        3072        same as above
bgsave      ds.l        192         32/2+8*32 bytes / 4 for long
```

The longest source to date, I'm truly starting to doubt the wisdom of putting the source code here as well as in a separate file. Anyways, starting from beginning going down, this is what it's all about. The first two lines are the X and Y speed, you may play around with these values to your hearts content, of course, setting them both to the same value will make the sprite move in 45 degrees, while any other values will make the sprite move differently.

Then, the pre- shifting of the sprite and the mask, this has been dealt with extensively and there is nothing more to add. After this, the

background is also prepared, it's just another put-degas-file-in-screen-memory. Note here, that there is a background save, this is to make sure something is in the save buffer before starting the main routine, otherwise the main routine would start off by "restoring" a blank area, effectively deleting a sprite sized block of the screen.

All preparations are done, just install the main routine, as described in tutorial 9. Put our main routine in the $70 vector, to have it executed every VBL. Wait for a key press, during which the main routine will execute continuously, and make a clean exit. Now, take note of how nice and tidy the main routine is, it just consists of subroutine calls, making the structure of the program very easy to read, and isolating each major part of the program for ease of reading.

Each subroutine in turn relies upon the get_coordinates routine, which translates the x_coord and y_coord into data intelligible to the program. As you can see in the comments at the start of the get_coordinates subroutine, what the routine does is to put a pointer to the screen memory in a0 and the sprite position (offset) in d0.b (meaning the least significant 8 bits of the d0 register). Since each subroutine relies upon the get_coordinates routine, if a bug is detected in the coordinate routine, it will only have to be dealt with in one place.

The save/restore background routines are short and simple and do little. They begin by calling the get_coordinates routine in order to get a screen pointer, the sprite position is uninteresting since they both deal with the entire sprite block.

The sprite and mask routines are very similar. Both begin by calling the get_coordinates routine, in order to get a screen pointer and a sprite position. Then either the sprite or mask area is loaded as appropriate, and the sprite position applied as an offset. Then comes a loop of moving data from the background and sprite or mask. Then this data is either and:ed or or:ed as appropriate. The result is put back in the screen memory.

Well, that is that. I haven't gone into everything in minute detail, but by now you shouldn't have to be baby nursed through every operation. The source code has many fun things you can do with it yourself, so test around some in the critical areas. The obvious change is the speed change, then, you can try commenting out some things in the main routine, and change an or to a move in the sprite routine for example. I think it's a good idea to play around some with the source code, and try to predict the changes, in this way, you'll really understand the underlying mechanics.

The next tutorial will probably be a very small one, I'm even considering of calling it tutorial 11 part B, and might cover the well known "infinite trail" of sprites, since it's ridiculously easy. Somewhere soon I suppose I'll do one on joystick and perhaps also mouse operation. I won't promise anything though. Big thanks again go out to Bruno Padinha, for providing valuable feedback and hitting me on the head. Damn, now I have to think of a good quote as well, this part is the hardest :)

perihelion of poSTmortem, 2002-07-06

“Is it possible that we two, you and I, have grown so old and inflexible that we have outlived our usefulness? Would that constitute … a joke?”

- Star Trek VI, the Undiscovered Country

Last edited 2002-07-12