**Store your sprite data:**  
  
I find it easiest to store the Character sprites first.  Point the first one to Pic1 and let the rest be relative to that pointer.  
After that, point the "ground tile" to Pic0.  This tile will be represented by "00" later in the data.  This tile is also the one you can always walk on.  
The other tiles will also be relative to this pointer, increasing by 8 to get to the next tile.  This is more organized if you comment above each piece of data, saying what it is and what number it is.  Remember to count in Hex here.

**Code:**[[Select]](javascript:void(0))

:.CHARACTER  
:.HEADFRONT  
:[7E7E99A5A5423C42]→Pic1  
:.BODYFRONT  
:[81A5A5A5425A6666]  
:[81A5A5A5425A6760]  
:[81A5A5A5425AE606]  
:  
:.HEADRIGHT  
:[3C7EF1C585423C42]  
:.BODYRIGHT  
:[4A4A45452624241C]  
:[4A4A45452654B2EC]  
:[4A4A45252624241C]  
:  
:.HEADLEFT  
:[3C7E8FA3A1423C42]  
:.BODYLEFT  
:[5252A2A264242438]  
:[5252A2A2642C4A76]  
:[5252A2A464242438]  
:  
:.HEADBACK  
:[7E7EB9B191423C42]  
:.BODYBACK  
:[81A5A5A5425A6666]  
:[81A5A5A5425A6760]  
:[81A5A5A5425AE606]  
:  
:  
:.STORE YOUR SPRITE DATA FIRST, I'M ASSUMMING YOU WANT MONOCHROME SPRITES  
:.EXAMPLE SPRITES  
:.GROUND TILE,0 THIS IS THE NUMBER THAT WILL REPRESENT THE TILE  
:[00A04000000A0400]→Pic0  
:  
:.TREE,1  
:[186689A14A3C2442]  
:  
:.HOUSE,2,3,4,5,6,7,8,9,A SOME OBJECTS NEED MORE THAN ONE TILE, AND THE A IS 10 IN BASE 16  
:[03060A1A2A6AAAAA]  
:[FF00000000000000]  
:[C060505854565555]  
:[ABACABB4A8D2A5C2]  
:[FF00FF000000007E]  
:[D535D52D154BA543]  
:[407F407F407F403F]  
:[8181A1A1818181FF]  
:[02FE02FE02FE02FC]  
:  
:.PATH,B  
:[8181818181818181]  
:  
:.BLACK BOUDARIES,C  
:[FFFFFFFFFFFFFFFF]  
  
Next, keeping the number for each tile in mind, write them out and lay it out like a map, as in:

**Code:**[[Select]](javascript:void(0))

:.MAP DATA, REMBER THOSE NUMBERS? ADD 0's RIGHT BEFORE THEM, AND STORE THEM TO A POINTER  
:[0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C]→GDB1  
:[0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C]  
:[0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C]  
:[0C0C0C0C010101010101010101010101010101010C0C0C0C]  
:[0C0C0C0C010000000000000000000000000000010C0C0C0C]  
:[0C0C0C0C010000000000000000000000000000010C0C0C0C]  
:[0C0C0C0C010000000000000000000000000000010C0C0C0C]  
:[0C0C0C0C010000000203040000000000000000010C0C0C0C]  
:[0C0C0C0C010000000506070000000000000000010C0C0C0C]  
:[0C0C0C0C0100000008090A0000000000000000010C0C0C0C]  
:[0C0C0C0C01000000000B000000000000000000010C0C0C0C]  
:[0C0C0C0C01000000000B000000000000000000010C0C0C0C]  
:[0C0C0C0C01000000000B000000000000000000010C0C0C0C]  
:[0C0C0C0C010000000000000000000000000000010C0C0C0C]  
:[0C0C0C0C010000000000000000000000000000010C0C0C0C]  
:[0C0C0C0C010000000000000000000000000000010C0C0C0C]  
:[0C0C0C0C010000000000000000000000000000010C0C0C0C]  
:[0C0C0C0C010000000000000000000000000000010C0C0C0C]  
:[0C0C0C0C010101010101010101010101010101010C0C0C0C]  
:[0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C]  
:[0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C]  
  
Looking back, remember the "0C"s are boundaries, "01"s are trees, and "00"s are ground tile, etc.  
Too keep it organized, I suggest entering the data for each row on a separate line.  Point the first one to GDB1, and let the rest be relative.  
When you enter another map, point it to a different GDB, and when you switch maps, Copy a map to a temp pointer, such as GDB0, and just always use GDB0 when mapping.  
  
<hr>  
  
**Movement engine:**  
  
Keep in mind that this engine is written such that your character stays in the same location on screen, and the map moves around him.  
  
  
Zero the pointers you need to be 0 first

**Code:**[[Select]](javascript:void(0))

:.P AND Q ARE THE LOCATION OR THE TOP CORNER OF THE MAP DATA  
:0→P→Q→V→H→I→J→θ→{r1}  
The {r1} is the one in VARS>Y-VARS>Polar>r1, its the variable telling how much the character pointer should add to to display a sprite facing the direction your moving.  
The P and Q are the "psuedo-coordinates" of the top corner of the screen on the map data.  We're basically pretending its an array.  So P=0 and Q=0 would correspond to the tile in the top left corner being a "0C" tile, or, a black one.  And the rest of the tiles shown on the screen will be relative to that.  Just look at the "array", and go down Q rows and go right P bytes, a byte being two digits.  
  
V, H, I, and J are just variable to make sure each time you press a arrow key, you only go to a direction 8 pixels at a time, but with smooth scrolling.  The smooth scrolling will be better explained later on.

**Code:**[[Select]](javascript:void(0))

:Repeat getKey(15)  
:  
:Z→Y  
:  
:Q\*24+P+126→Z  
:  
:.P+(getKey(3) and (({Z+1+GDB1}=0) or ({Z+1+GDB1}=11)))-(getKey(2) and (({Z-1+GDB1}=0) or ({Z-1+GDB1}=11)))→P  
:  
:.Q\*24+P+101→Z  
:.Q+(getKey(1) and (({Z+24+GDB1}=0) or ({Z+24+GDB1}=11)))-(getKey(4) and (({Z-24+GDB1}=0) or ({Z-24+GDB1}=11)))→Q  
:  
:If getKey(3) and (I=0) and (J=0) and (({Z+1+GDB1}=0) or ({Z+1+GDB1}=11))  
:1→I  
:32→{r1}  
:End  
:If getKey(2) and (I=0) and (J=0) and (({Z-1+GDB1}=0) or ({Z-1+GDB1}=11))  
:0-1→I  
:64→{r1}  
:End  
:If getKey(1) and (I=0) and (J=0) and (({Z+24+GDB1}=0) or ({Z+24+GDB1}=11))  
:1→J  
:0→{r1}  
:End  
:If getKey(4) and (I=0) and (J=0) and (({Z-24+GDB1}=0) or ({Z-24+GDB1}=11))  
:0-1→J  
:96→{r1}  
:End  
:  
:If getKey(1) or (getKey(2)) or (getKey(3)) or (getKey(4)  
:θ+1→θ  
:Else  
:0→θ  
:End  
:  
:H+I+I→H  
:V+J+J→V  
:  
:If H-8=0 or (H+8=0  
:P+I→P  
:0→H→I  
:End  
:If V-8=0 or (V+8=0  
:Q+J→Q  
:0→V→J  
:End  
:  
:.If Y≠Z  
:sub(MAP)  
:.End  
:  
:Pt-Off(40,32,Pic1+8+(θ/4^2\*8+8\*(θ≠0))+{r1}  
:Pt-Off(40,24,Pic1+{r1}  
:  
:DispGraph  
:  
:End  
  
The parts that are commented out are for scrolling by 8 pixels at a time (i.e. not smooth scrolling)  Uncomment those and comment/delete the If clauses with getkey(1-4) in them.  (but not the one that involves storing things into theta)  
  
Smooth Scrolling:  
H and V are constantly being incremented by I and J respectively.  When you aren't moving, I and J are 0, when you move, I and J change to (-)1 depending where you're going.  When H or V equal 8, they get reset to 0 and 8 is added/subtracted to/from P or Q, depending on whether you went**H**orizontally or **V**ertically.  
  
Collision detection:  
Collision detection is a simple matter of finding out where you are on our "array" and checking whats above, below, and beside you.  to find the number of bytes you are away from GDB1, you would take Q, multiply it by you "array's" width, add P, and add 101, in this case, since that gets you from the top corner of the screen to the place where your body is.  Now, just check if {Z (the variable with your distance from GDB1) plus GDB1 +/- 1/[width of your array]}  To check to your left/right, +/- 1, to check above and below, +/- 24, in this case, that being the width of our "array."  
  
<hr>  
**Map Drawing**  
  
You may have noticed the "sub(MAP)" in the above code, calling the subroutine "MAP."  
  
The map routine is very simple.  Use the vars P and Q to find the distance from GDB1 to the top left corner of your screen, as shown in below source.  And then use two for loops to increment by 1 to find the next tile, until you reach 12 tiles, and then skip to the next line, and display 12 tiles, and display 8 rows of twelve tiles.

**Code:**[[Select]](javascript:void(0))

:Lbl MAP  
:  
:Q\*24+P→Z  
:  
:ClrDraw  
:For(B,0,9  
:For(A,0,13  
:Pt-On(A\*8-8-H,B\*8-8-V,{B\*24+A+Z+GDB1}\*8+Pic0  
:End  
:End  
  
Attached below is a screenshot of the Example program and the source/executable