Scroll Vertical Scroll technique in Basic Tower

Each stage is a text string 60 lines height, and 13 characters wide. At the end of each line we add a Carriage Return, to jump to the next line of the string in the screen.

With this data, we get 60 * (13+1) = 840 bytes for the whole stage.
This is true, assuming that the stage is monochrome, and we don't need color codes within the string.

So, in order to draw the top frame, starting at line 1, and a height of 20 lines, we have to point z$ string (this is the string that will be printed) to the address at which the string begins (first line), and set a size of 20*(13+1) = 280 bytes.

If we want to scroll down one line, we have to set the z$ string to point to the address where the second line starts, and set a size of 280bytes.

So on, if we continue scrolling down, until the last line, this frame will set z$ to the start of line 41, and height of 20 lines, 280bytes

In Sinclair Basic, there is no direct way of using pointers, so to tell the interpreter at which address a string starts and its size, we use DefAdd: we have to create a structure of a DEF FN instruction where we can set the starting address and the size of a chunk of bytes. In BasicTower, defadd data starts at 38912, and I use 15 variables. One of them is z$, and it gets the string of the frame to be printed on the screen. The address is at 38930+4 y +5 and the size is at 38930+6 y +7.

Then we execute PRINT AT 0,0; z$; and the frame will be printed instantly (line 141 for up and line 167 for down, there is where magic happens).

If we want to use color codes within de string size will easily double or triple or even more. So to reduce the amount of bytes to be printed I use this optimization:

I preprocess offline each stage, checking if the line to print has changed from last frame, and I only print it if so. If is is the same, I skip that line just printing a Carriage Return, or the line until the last character that is different.

For example, imagine there are 5 lines with bricks, representing a wall, from lines 10 to 14. After scrolling down, we will get all bricks in line 9, but in lines 10,11,12 and 13 there will be those bricks already printed on the screen, so I can skip those just printing 4 Carriage Return, saving many many bytes, which means fasters prints.

That preprocess shrinks the size of the 60-line stage to a maximum of 768bytes, and each frame of 20 lines, including color codes, is around 300bytes.

Another small optimization is instead of using 4 pokes to set address and size of the frame, I use a LET instruction of DefAdd to fast copy those 4 bytes (line 141)

We get the sensation of background animation swapping UDG Banks, among the 4 available (line 110).

Other DefAdd variables I use are:
3 (i$, y$ y f$) for the string that prints the ghost,
5 (c$, k$ y g$) for the string that prints the gargoyles,
3 (p$, q$ y r$) to copy address and size of the frame
1 (e$) for the string that prints the hero and restores the backgroud of last position
2 (o$ y d$) to copy all the info of this stage to the working zone